VisiNotify ガイド

Borland VisiBroker® 7.0

Borland®
Excellence Endures™

Borland Software Corporation
20450 Stevens Creek Blvd., Suite 800
Cupertino, CA 95014 USA
www.borland.com

2006 年 5 月 11 日初版発行

著者：Borland Software Corporation

発行：ボーランド株式会社

PDF

目次

第 1 章

Borland VisiBroker の概要

Borland は，CORBA 開発者に向けて，業界最先端の VisiBroker オブジェクトリクエストブローカー（ORB）を活用するために *VisiBroker for Java*，*VisiBroker for C++*，および *VisiBroker for .NET* を提供しています。この 3 つの VisiBroker は CORBA 2.6 仕様の実装です。

VisiBroker の概要

VisiBroker は，CORBA が Java オブジェクトと Java 以外のオブジェクトの間でやり取りする必要がある分散配布で使用されます。幅広いプラットフォーム（ハードウェア，オペレーティングシステム，コンパイラ，および JDK）で使用できます。VisiBroker は，異種環境の分散システムに関連して一般に発生するすべての問題を解決します。

VisiBroker は次のコンポーネントからなります。

- VisiBroker for Java，VisiBroker for C++，および VisiBroker for .NET（業界最先端のオブジェクトリクエストブローカーの 3 つの実装）。
- VisiNaming Service - Interoperable Naming Specification バージョン 1.3 の完全な実装。
- GateKeeper - ファイアウォールの背後の CORBA サーバーとの接続を管理するプロキシサーバー。
- VisiBroker Console - CORBA 環境を簡単に管理できる GUI ツール。
- コモンオブジェクトサービス - VisiNotify（通知サービス仕様の実装），VisiTransact（トランザクションサービス仕様の実装），VisiTelcoLog（Telecom ログサービス仕様の実装），VisiTime（タイムサービス仕様の実装），VisiSecure など。

VisiBroker の機能

VisiBroker には次の機能があります。

- セキュリティと Web 接続性を容易に装備できます。
- J2EE プラットフォームにシームレスに統合できます（CORBA クライアントが EJB に直接アクセスできる）。
- 堅牢なネーミングサービス（VisiNaming）とキャッシュ，永続的ストレージ，および複製によって高可用性を実現します。
- プライマリサーバーにアクセスできない場合に，クライアントをバックアップサーバーに自動的にフェイルオーバーします。
- CORBA サーバークラスタ内で負荷分散を行います。
- OMG CORBA 2.6 仕様に完全に準拠します。
- Borland JBuilder 統合開発環境と統合されます。
- Borland AppServer などの他の Borland 製品と最適に統合されます。

VisiBroker のマニュアル

VisiBroker のマニュアルセットは次のマニュアルで構成されています。

- *Borland VisiBroker インストールガイド* — VisiBroker をネットワークにインストールする方法について説明します。このマニュアルは，Windows または UNIX オペレーティングシステムに精通しているシステム管理者を対象としています。
- *Borland VisiBroker セキュリティガイド* — VisiSecure for VisiBroker for Java および VisiBroker for C++ など，VisiBroker のセキュリティを確保するための Borland のフレームワークについて説明しています。
- *Borland VisiBroker for Java 開発者ガイド* — Java による VisiBroker アプリケーションの開発方法について記載されています。Visibroker ORB の設定と管理，およびプログラミングツールの使用方法について説明します。また, IDL コンパイラ, スマートエージェント，ロケーションサービス，ネーミングサービス，イベントサービス，オブジェクトアクティベーションデーモン (OAD), Quality of Service (QoS), インターフェースリポジトリ，および Web サービスサポートについても説明します。
- *Borland VisiBroker for C++ 開発者ガイド* — C++ による VisiBroker アプリケーションの開発方法について記載されています。Visibroker ORB の設定と管理，およびプログラミングツールの使用方法について説明します。また, IDL コンパイラ, スマートエージェント，ロケーションサービス，ネーミングサービス，イベントサービス，OAD, QoS，プラグイン可能トランスポートインターフェース，RT CORBA 拡張機能，Web サービスサポート，およびインターフェースリポジトリについても説明します。
- *Borland VisiBroker for .NET 開発者ガイド* — .NET 環境による VisiBroker アプリケーションの開発方法について記載されています。
- *Borland VisiBroker for C++ API リファレンス* — VisiBroker for C++ に付属するクラスとインターフェースについて説明します。
- *Borland VisiBroker VisiTime ガイド* — Borland による OMG Time Service 仕様の実装について説明します。
- *Borland VisiBroker VisiNotify ガイド* — Borland による OMG 通知サービス仕様の実装について説明します。通知メッセージフレームワークの主な機能として, 特に Quality of Service (QoS) のプロパティ, フィルタリング, および Publish/Subscribe Adapter (PSA) の使用方法が記載されています。

- *Borland VisiBroker VisiTransact ガイド* — Borland による OMG Object Transaction Service 仕様の実装および Borland Integrated Transaction Service コンポーネントについて説明します。
- *Borland VisiBroker VisiTelcoLog ガイド* — Borland による OMG Telecom Log Service 仕様の実装について説明します。
- *Borland VisiBroker GateKeeper ガイド* — Web ブラウザやファイアウォールによるセキュリティ制約の下で，VisiBroker GateKeeper を使用して，VisiBroker のクライアントがネットワークを介してサーバーとの通信を確立する方法について説明します。

通常，マニュアルにアクセスするには，VisiBroker とともにインストールされるヘルプビューアを使用します。ヘルプは，スタンドアロンのヘルプビューアからアクセスすることも，VisiBroker コンソールからアクセスすることもできます。どちらの場合も，ヘルプビューアを起動すると独立したウィンドウが表示されるため，このウィンドウからヘルプビューアのメインツールバーにアクセスしてナビゲーションや印刷を行ったり，ナビゲーションペインにアクセスすることができます。ヘルプビューアのナビゲーションペインには，すべての VisiBroker ブックとリファレンス文書の目次，完全なインデックス，および包括的な検索を実行できるページがあります。

重要 Web サイト http://www.borland.com/techpubs には，PDF 版のマニュアルと最新の製品マニュアルがあります。

スタンドアロンヘルプビューアからの VisiBroker オンラインヘルプトピックへのアクセス

製品がインストールされているコンピュータでスタンドアロンのヘルプビューアからオンラインヘルプにアクセスするには，次のいずれかの手順を実行します。

Windows

- ［スタート｜プログラム｜ Borland VisiBroker ｜ Help Topics］の順に選択します。
- または，コマンドプロンプトを開き，製品のインストールディレクトリの ¥bin ディレクトリに移動し，次のコマンドを入力します。

```
help
```

UNIX コマンドシェルを開き，製品のインストールディレクトリの /bin ディレクトリに移動し，次のコマンドを入力します。

```
help
```

ヒント UNIX システムにインストールするときの指定で，PATH エントリのデフォルトに bin を含まないようにします。カスタムインストールオプションを選択して PATH エントリのデフォルトを変更せず，PATH に現在のディレクトリのエントリがない場合は，./help を使用してヘルプビューアを起動できます。

VisiBroker コンソールからの VisiBroker オンラインヘルプトピックへのアクセス

VisiBroker コンソールから VisiBroker オンラインヘルプトピックにアクセスするには，［Help ｜ Help Topics］を選択します。

［Help］メニューには，オンラインヘルプ内のいくつかの文書へのショートカットもあります。ショートカットの 1 つを選択すると，ヘルプトピックビューアが起動し，［Help］メニューで選択した項目が表示されます。

マニュアルの表記規則

VisiBroker のマニュアルでは，文中の特定の部分を表すために，次の表に示す書体と記号を使用します。

表 1.1　マニュアルの表記規則

表記規則	用途
italic	新規の用語およびマニュアル名に使用されます。
`computer`	ユーザーやアプリケーションが提供する情報，サンプルコマンドライン，およびコードです。
bold computer	本文では，ユーザーが入力する情報を示します。サンプルコードでは，重要なステートメントを強調表示します。
[ ]	省略可能な項目。
...	繰り返しが可能な直前の引数。
|	二者択一の選択。

プラットフォームの表記

VisiBroker マニュアルでは，次の記号を使用してプラットフォーム固有の情報を示します。

表 1.2　プラットフォームの表記

記号	意味
Windows	サポートされているすべての Windows プラットフォーム
Win2003	Windows 2003 のみ
WinXP	Windows XP のみ
Win2000	Windows 2000 のみ
UNIX	すべての UNIX プラットフォーム
Solaris	Solaris のみ
Linux	Linux のみ

Borland サポートへの連絡

ボーランド社は各種のサポートオプションを用意しています。それらにはインターネット上の無償サービスが含まれており，大規模な情報ベースを検索したり，他の Borland 製品ユーザーからの情報を得ることができます。さらに Borland 製品のインストールに関するサポートから有償のコンサルタントレベルのサポートおよび高レベルなアシスタンスに至るまでの複数のカテゴリから，電話サポートの種類を選択できます。

Borland のサポートサービスの詳細や Borland テクニカルサポートへの問い合わせについては，Web サイト `http://support.borland.com` で地域を選択してください。

ボーランド社のサポートへの連絡にあたっては，次の情報を用意してください。

- 名前
- 会社名およびサイト ID
- 電話番号
- ユーザー ID 番号（米国のみ）
- オペレーティングシステムおよびバージョン
- Borland 製品名およびバージョン
- 適用済みのパッチまたはサービスパック
- クライアントの言語とそのバージョン（使用している場合）
- データベースとそのバージョン（使用している場合）

- 発生した問題の詳細な内容と経緯
- 問題を示すログファイル
- 発生したエラーメッセージまたは例外の詳細な内容

オンラインリソース

ネットワーク上の次のサイトから情報を得ることができます。

Web サイト http://www.borland.com/jp/

オンラインサポート http://support.borland.com（ユーザー ID が必要）

リストサーバー 電子ニュースレター（英文）を購読する場合は，次のサイトに用意されているオンライン書式を使用してください。

http://www.borland.com/products/newsletters

Web サイト

定期的に http://www.borland.com/jp/products/visibroker/index.html をチェックしてください。VisiBroker 製品チームによるホワイトペーパー，競合製品の分析，FAQ の回答，サンプルアプリケーション，最新ソフトウェア，最新のマニュアル，および新旧製品に関する情報が掲載されます。

特に，次の URL をチェックすることをお勧めします。

- http://www.borland.com/products/downloads/download_visibroker.html（最新の VisiBroker ソフトウェアおよび他のファイル）
- http://www.borland.com/techpubs（マニュアルの更新および PDF）
- http://info.borland.com/devsupport/bdp/faq/（VisiBroker の FAQ）
- http://community.borland.com（英語，開発者向けの弊社 Web ベースニュースマガジン）

Borland ニュースグループ

Borland VisiBroker を対象とした数多くのニュースグループに参加できます。VisiBroker などの Borland 製品のユーザーによるニュースグループへの参加については，http://www.borland.com/newsgroups を参照してください。

メモ これらのニュースグループはユーザーによって管理されているものであり，ボーランド社の公式サイトではありません。

第 2 章

VisiNotify の概要

この章では，OMG Event/Notification Service のアーキテクチャと，Borland のインプリメンテーション VisiNotify の概要について説明します。

メモ　このマニュアルは，VisiNotify に付属しているアプリケーションサンプルと，OMG Web サイトから入手できる『OMG Event/Notification Specification』とともに使用してください。

OMG Event/Notification Service の通信モデル

CORBA 環境では，コア ORB が，オブジェクト指向のクライアント／サーバーアプリケーションを作成する場合の分散フレームワークになります。コア ORB でサポートされる通信モデルは，1 対 1 の同期通信を直接（少なくとも概念上）行うクライアント／サーバーアプリケーション用です。アプリケーションに必要な条件の中には，次のようにコア ORB ではサポートできない機能があります。

- 多対多，分離された通信など，分散 publish/subscribe アプリケーションの設計
- 同期通信より実質的にはスループットが大きい，非同期にバッファリングされた単方向のイベント配布
- イベント／接続の信頼性などの Quality of Service（QoS）
- イベントフィルタリング

上で説明した要件は，従来からある CORBA 以外のアプリケーション用メッセージ指向ミドルウェア（MOM）によってサポートされています。OMG Event/Notification Service では，CORBA アプリケーションと同じ要件を処理することができます。

publish/subscribe アプリケーションでは，通信に含まれるオブジェクトは任意です。publish/subscribe 通信には，イベントサプライヤ（プロバイダとパブリッシャ）とイベントコンシューマ（オブザーバとサブスクライバ）の 2 種類のオブジェクトがあります。また，イベント転送モデルには，イベントプッシュとイベントプルの 2 種類あります。publish/subscribe 通信に含まれるオブジェクトは，メッセージミッドウェアによって互いに分離されます。これらのオブジェクトは適切に動作し，ほかのオブジェクトの存在と状態に依存することはありません。イベントサプライヤは，コンシューマの存在に関係なく，チャネルにイベントを転送します。

メモ　このインスタンスにおける分離とは，セキュリティではなく独立していることを意味します。サプライヤが暗黙的にコンシューマの存在を通知できる場合でも，分離が破棄されたということにはなりません。

単方向のイベント配布では，イベントは上流から下流に流れます。具体的には，イベントはサプライヤからチャネルへ，次にチャネルからサブスクライブしたコンシューマへと流れます。イベント転送は，**非同期**でバッファリングされています。サプライヤは，コンシューマからではなくメッセージミドルウェアから認知されます。つまり，メッセージミドルウェアを介してルーティングされるイベント転送では，ルーティングしない同期メソッドの呼び出しより，スループットが大幅によくなります。

OMG Event/Notification Service のオブジェクトモデル

OMG Event/Notification Service の主なアーキテクチャは，**チャネル**です。イベントは，イベントチャネルに送信され，受信者にレプリケートされます。複数の独立したチャネルを作成して，特定のアプリケーションで使用できます。イベントは，サプライヤからイベントチャネルにプッシュまたはプルされます。イベントは，下流へ向かってチャネル内を流れます。下流の末端にあるイベントは，プロキシサプライヤでバッファリングされ，コンシューマによってプッシュまたはプルされます。アプリケーションレベルのイベントサプライヤまたはイベントコンシューマは，プロキシオブジェクトを使って接続され，チャネルにイベントを転送したり，チャネルからイベントを転送します。

チャネルの**下流**の末端（コンシューマ側）では，次のようになります。

- 各プッシュコンシューマは，専用のプロキシプッシュサプライヤを作成して，接続する必要があります。次に，プッシュコンシューマは，チャネルコールバックがイベントを送信するのを非アクティブに待機します。
- プルコンシューマは，プロキシプルサプライヤに対してアクティブに要求を行い，チャネルからイベントを取得します。

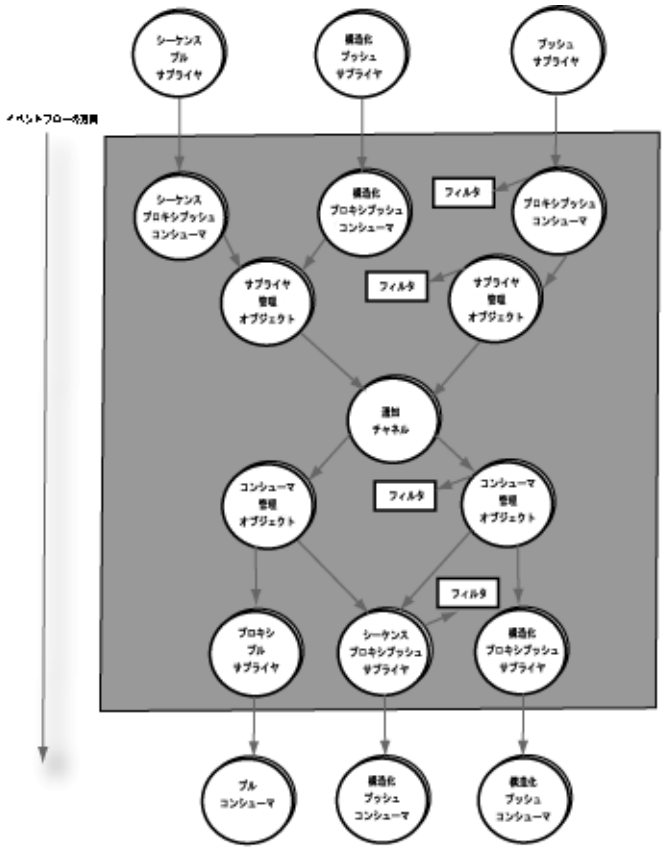
プロキシサプライヤは，通常，チャネルサーバー内にあり，コンシューマ管理からのアプリケーションによって作成されます。コンシューマ管理は，デフォルトで作成されるか，チャネルからのアプリケーションによって作成されます。各チャネルには，デフォルトのコンシューマ管理があります。この作成プロセスにより，［チャネル］-［コンシューマ管理］-［サプライヤプロキシ］階層が形成されます。

チャネルの**上流**の末端（サプライヤ側），次のようになります。

- プッシュサプライヤは，プロキシプッシュコンシューマに対してアクティブに要求を行い，チャネルにイベントをプッシュします。
- 各プルサプライヤは，専用のプロキシプルコンシューマを作成して，接続する必要があります。次に，プルサプライヤは，チャネルコールバックがイベントを取得するのを非アクティブに待機します。

プロキシコンシューマは，通常，チャネルサーバー内にあり，サプライヤ管理からのアプリケーションによって作成されます。サプライヤ管理は，デフォルトで作成されるか，チャネルからのアプリケーションによって作成されます。各チャネルには，デフォルトのサプライヤ管理があります。この作成プロセスにより，［チャネル］-［サプライヤ管理］-［コンシューマプロキシ］階層が形成されます。

この図は，通知通信モデル内のイベントフロー（上流と下流）を示しています。



OMG 通知サービスは，従来のメッセージ指向ミドルウェアと同様に，Qualify of Service（QoS）を定義およびサポートします。VisiNotify は，OMG によって定義されたほとんどの QoS ポリシーを，追加の VisiNotify 拡張機能とともにサポートします。これらの QoS ポリシーの中で，最も重要な 2 つの QoS は，イベントの永続性と接続の永続性です。イベントの永続性（または信頼性）では，チャネル内のバッファリングされたイベントは，一時的に永続的リポジトリに保存されるので，保守のためのシャットダウンや不慮のシステムクラッシュによるイベントの損失を防ぐことができます。接続の永続性（または信頼性）では，OMG は 2 つの QoS 機能を定義します。1 つめの機能は，チャネル，管理，およびプロキシの画像です。これらの現在の設定は，永続的リポジトリに保存されるので，チャネルサーバーは，チャネルを再起動したときにこれらのオブジェクトを復元できます。2 つめの機能では，チャネルがサプライヤをプルし，コンシューマをプッシュするために，転送接続を再確立します。

OMG Event/Notification Service で定義されるもう 1 つの重要なサービスは，イベントフィルタリングです。アプリケーションは，管理レベルまたはプロキシレベルでフィルタオブジェクトを追加して，不要なイベントを選択的にフィルタアウトします。

OMG Event/Notification Service では，**型なし**，**構造化**，**シーケンス**，および**型付き**の 4 種類の通知チャネルが定義されます。最初の 3 つのチャネルのイベントインターフェースは，OMG Event ／ Notification 仕様によって定義済みで，「定義済み」チャネルと呼ばれます。型付きチャネルのイベントインターフェースは，OMG Event ／ Notification 仕様ではなく，ユーザーアプリケーションによって定義済みで，「ユーザー定義」の型付きチャネルと呼ばれます。VisiNotify では，4 種類のチャネルすべてをサポートします。ただし，シーケンスプリングは除きます。

型なしチャネルのイベントは，CORBA *Anys* で示されます。イベントを送信するには，型なしコンシューマまたはプロキシ型なしコンシューマオブジェクトを使用して，入力パラメータとして CORBA::Any を持つ push() オペレーションを呼び出します。構造化チャネルのイベ

ントおよびシーケンスチャネルのイベントは，tructuredEvent　IDL　構造体または StructuredEvent IDL シーケンスで示されます。イベントを送信するには，各コンシューマまたはプロキシコンシューマの push_structured_event() または push_structured_events() を呼び出します。型付きチャネルには，定義済みのイベントインターフェースはありません。イベントインターフェースは，ユーザーアプリケーションによって OMG IDL インターフェースとして定義されます。イベントを送信するには，コンシューマまたはプロキシコンシューマの型付きインターフェースの擬似でないオペレーションを呼び出します。

メモ　上記のイベントタイプを持つサプライヤアプリケーションおよびコンシューマアプリケーションのサンプルについては，このマニュアルの第 3 章「サプライヤ／コンシューマアプリケーションの開発」を参照してください。

VisiNotify の機能

Borland VisiNotify は，商用の OMG Event/Notification Service のインプリメンテーションです。VisiNotify は，ユーザーレベルではなく ORB レベルで実装され，一般的なネーミングサービス機構によってネーミングサービスに登録されます。詳細については，「VisiNaming サービスの使い方」(Java) および「VisiNaming サービスの使い方」(C++) を参照してください。VisiNotify は，このユニークな設計のおかげで，さらに効率的に動作し，ユーザーレベルではサポートが難しい機能または不可能な機能を提供します。次に，VisiNotify の主な機能を示します。

優れたスループットとスケーラビリティ

VisiNotify は，GIOP メッセージレベルで動作するように設計されています。VisiNotify は，受信したイベントペイロードを下流のコンシューマに直接渡します。受信したイベントをレプリケートする場合，フィルタやイベントレベルの QoS がストリーム内にない限り，VisiNotify はイベントをアンマーシャルしません。フィルタやイベントレベルの QoS がある場合でも，VisiNotify はイベントをリマーシャルしない場合があります。このユニークな設計により，VisiNotify では，CPU の利用率を低く抑え，イベントスループットを高くすることができます。GIOP 1.0 および 1.1 を介してクライアント接続を処理する場合は，連続した高度な技術を使用して，ペイロードの位置を調整します。この場合，イベントのデマーシャリングやリマーシャリングは不要です。

VisiBroker 5.1 における Borland のイベントバッファリングおよびバッチ技術を利用すれば，VisiNotify で表示されるスループットは，市場のどのようなユーザーレベルの通知サービス製品よりかなり高くなります。イベントバッファリングおよびバッチにより，VisiNotify のスループットが最適化されます。ユーザーレベルのバッチ技術とは異なり，シーケンスイベントなどのイベントバッファリングやバッチ技術は，ユーザーアプリケーションに対して完全に透過的で，イベントタイプに対する制限もありません。すべてのイベントタイプ（型なし，構造化，シーケンス，型付き）は，バッファリングされ，まとめて送信されます。そのため，VisiNotify は，最小のイベントサイズ，最低のマーシャリング／デマーシャリングコストの型付きイベントとイベントバッチを組み合わせることによって，最善の終端間イベントスループットに達することができます。

ユーザーレベルのインプリメンテーションでは，イベントバッファリング／バッチは，アプリケーションに対して透過的ではありません。また，特定のイベントタイプ，つまり構造化イベントだけが一括して送信されます。Borland のイベントバッファリング／バッチ技術と比べて，シーケンスチャネルを使用したイベントバッチには利点がありません。そのため，VisiNotify のシーケンスチャネルに対するサポートは，次のように制限されています。

- 終端間のプッシュモデルのシーケンスチャネルだけをサポートします。フィルタの制限とイベントレベルの QoS ポリシーは，シーケンス内の最初のイベントに対してのみ評価され，その結果はイベント全体に適用されます。
- 最大バッチサイズの設定は無視されます。

- シーケンスプルはサポートされません。

メモ：実際の業界における使用例（ITU-T CORBA/TMN 通知）に基づいたスループット用ベンチマークテストスィートは，この VisiNotify バージョンに同梱されています（examples/vbroker/notify/bench_cpp and bench_java）。

イベントの永続性のパフォーマンスの向上

多くのユーザーレベルのチャネル製品では，永続性をサポートするために DynAny を使用して，イベントからイベントをアンパックします。VisiNotify は，イベントメッセージペイロードをアンマーシャリングまたはアンパックしないで，永続的ストレージに直接ダンプします。このユニークな設計により，イベントの永続性のオーバーヘッドを最小限に抑えることができます。デフォルトの設定では，VisiNotify のイベント永続性のオーバーヘッドは，5 ～ 15 % です。

valuetype のサポート

VisiNotify は，イベントの valuetype をサポートする唯一の通知チャネルです。イベントストリームにフィルタが存在する場合でも，VisiNotify は，指定されたイベント内の最初の valuetype の前にある属性を使用して，フィルタの状況を評価することができます。

型付きチャネルのサポート

型付きチャネルのサポートについては，第 3 章「サプライヤ／コンシューマアプリケーションの開発」で説明しています。

VisiNotify は，Dynamic Invocation Interface（DII）および Dynamic Skeleton Interface（DSI）を使用しない初めての OMG Typed Event/Notification インプリメンテーションです。型付きイベントストリーム内にフィルタの制限がない限り，VisiNotify は Interface Repository には依存しません。つまり，VisiNotify の型付きチャネルは，あらゆる型付きチャネルまたは型なしチャネルのインプリメンテーションより著しく速いことを意味しています。

フィルタ未使用時には，VisiNotify は IR に依存しないので，obtain_typed_..._consumer/supplier() を呼び出すためのキーパラメータに，イベントインターフェースのリポジトリ ID を使用する必要はありません。したがって，アプリケーションは，代替のフィルタリング戦略としてプロキシキーを選択できます。アプリケーションは，プロキシキーを使用して，特定の型付きチャネルを複数の論理チャネルに分割できます。この方法は，制約のある言語解析ベースのフィルタリングより効率的で柔軟性があります。

Publish/Subscribe Adapter（PSA）

Publish/Subscribe Adapter の機能については，第 4 章「Publish/Subscribe Adapter（PSA）の使い方」の章で説明します。

PSA は，VisiBroker 5.1 でサポートされるプログラミングモデルおよびソフトウェアコンポーネントです。あらゆる OMG Event/Notification Service の上位で動作します。PSA の基本的な概念は，publish/subscribe 通信に対して，高レベルのオブジェクト指向抽出を提供することです。PSA は，型付きイベント／通知アプリケーションのコードを単純化するだけでなく，typed プルに対して明快なソリューションを提供します。また，接続インターフェースの相違点を直接処理しないように，アプリケーションを保護します。PSA を使用しないで，さまざまなイベントタイプ（型なし，構造化，シーケンス，型付き）や転送モードを使用するということは，さまざまな接続インターフェースを意味します。

Pull<I> インターフェースを使用しない型付きプル

PSA の明快な機能の 1 つに，切り取った Pull<I> インターフェースのかわりに，ユーザー定義された元の <I> インターフェースを使用した型付きプルをサポートする機能があります。

明示的 RMI と EJB のサポート

明示的 RMI と EJB のサポートについては，第 3 章「サプライヤ／コンシューマアプリケーションの開発」で説明しています。

VisiNotify は，2 種類の RMI と EJB 接続構成があります。最初の構成は，メッセージミドルウェアとして VisiNotify の型付きチャネルを使用する型付きイベントの RMI ／ EJB アプリケーションです。この場合，ユーザー定義の RMI インターフェースや EJB リモートインターフェースは，型付きイベントインターフェースの定義になります。すべてのサプライヤは，RMI アプリケーションです。RMI コールが，イベントを VisiNotify 型付きチャネルにプッシュします。接続されたすべてのコンシューマも RMI アプリケーションで，その RMI リファレンスは，型付きイベントチャネルに接続されています。

2 番めの構成では，構造化イベントチャネルを使用します。この構成のサプライヤは，すべて CORBA アプリケーションで，構造化イベントチャネルに CosNotification::StructuredEvent を送信します。下流の末端のコンシューマアプリケーションには，構造化コンシューマとして接続された CORBA アプリケーションもあれば，構造化イベント EJB Bean になるコンシューマもあります。構造化イベント EJB Bean は，通常のセッションやエンティティ Bean と変わりません。構造化イベント Bean とそのリモートインターフェースは，入力パラメータとして org.omg.CosNotification.StructuredEvent を使用して，push_structured_event() オペレーションを実装および宣言します。VisiNotify は，**subtool** というユーティリティを提供して，特定の Visinotify 構造イベントチャネルに構造化イベント Bean のリモートインターフェースを接続します。

この 2 つの構成は，イベント駆動型の J2EE アプリケーション用に純粋な代替のオブジェクト指向のソリューションを提供します。次に，Java Message Service (JMS) と Message Driven Bean（MDB）を比較した場合の利点を示します。

- 静的なタイプセーフの RMI スタブとスケルトンがメッセージのパック／アンパックを実行する。
- イベントがユーザー定義の Java RMI インターフェースによって表される。

接続の永続性

VisiNotify は，OMG 仕様で定義されているように，接続の永続性をサポートします。

- チャネルの再起動後，永続的なチャネル，管理，およびプロキシを復元します。
- 破棄された／失われた転送接続を再確立して，コンシューマをプッシュしたり，サプライヤをプルします。

VisiNotify は，永続的なチャネル，管理，およびプロキシだけでなく，それらの現在の設定と ID（ChannelID，AdminID，および ProxyID）も復元します。また，転送接続も再確立します。また，VisiNotify には，接続されたプッシュコンシューマやプルサプライヤへの接続が切断された場合に，自動的にプロキシを中断された状態にできる拡張機能があります。これは，ループで転送接続を再確立するよりは推奨される構成です。

自己適応型非同期フローの制御

OMG Notification Service 1.0 では，イベントストリームに最低 1 人のコンシューマがいれば，チャネルは，プルサプライヤからのイベントメッセージをサポートします。OMG Notification Service 1.3 の OMG では，チャネルに接続されているコンシューマがいる

かどうかに関係なく，プロキシがプルする必要があります。OMG から作成された引数，つまりこのイベントのプルは，コンシューマがいるかどうかをサプライヤにはわからないようにして，コンシューマからサプライヤを保護します。

この 2 つの構成では，必要がない密接なプルに対して，システムとネットワークリソースを浪費することになります。自己適応型非同期フローの制御を使用すれば，プロキシプルコンシューマがプルするのは，戻されたイベントが，下流内の少なくとも 1 人のコンシューマに渡される場合だけです。このインプリメンテーションでは，各論理チャネルに投票スロットを適用する必要があります。上流のプロキシプルコンシューマがプルするのは，その投票スロットの数字がゼロ以外の場合だけです。下流の各プロキシサプライヤ（プッシュまたはプル）では，その論理チャネルの投票スロットに対して票が 1 つあります。キューにあるイベントの数が，低いウォーターマークより低い場合は，プルに投票します。プルのキューにある保留中のイベントが，高いウォーターマークより高い場合は，その投票を取り消します。これにより，上流のプロキシプルコンシューマがイベントをプルバックして，下流のプロキシコンシューマによってただちにイベントが破棄または拒否されるのを防ぎます。高いウォーターマークと低いウォーターマークを設定すると，アプリケーションは，OMG Notification Service 1.0 または 1.3 の動作を取得することもできます。

QoS とフィルタのサポート

VisiNotify は，OMG QoS および VisiNotify の拡張機能をサポートします。また，構造化チャネル，シーケンスチャネル，および型付きチャネルに，最適化された OMG 準拠のフィルタのサポートを提供します。詳細については，第 5 章「Quality of Service とフィルタの設定」の章を参照してください。

オンデマンドのスレッド

内部的には，すべてのチャネルおよびアクティブなプロキシ（プロキシプルコンシューマおよびプロキシプッシュコンシューマ）に，スレッドが必要です。ただし，スレッドは，専用のサーバントとして割り当てられるわけではありません。階層的にチャネルやプロキシより上にあるその他のオブジェクトがアイドル状態のときは，スレッドはリサイクルされます。VisiNotify は，スレッドを動的に提供します。

第 3 章

サプライヤ／コンシューマアプリケーションの開発

この章では，OMG Notification Service を使用したサプライヤアプリケーションとコンシューマアプリケーションの開発方法について説明します。説明する項目は次のとおりです。

定義済み Event/Notification Service の使い方

OMG Notification Service では，**型なし**イベントチャネル，**構造化**イベントチャネル，および**シーケンス**イベントチャネルの 3 種類の定義済みチャネルを指定できます。定義済みチャネルは，ユーザーレベルのインプリメンテーションが容易であるという点で優れています。そのため，販売されているほとんどすべての通知サービス製品が，定義済みチャネルをサポートしています。次に，定義済みチャネルの欠点を示します。

- ユーザー定義の型付きチャネルより遅い。
- 通常，イベントサイズが大きい。
- イベントへのユーザーデータのパック，イベントからのユーザーデータのアンパックに必要な，タイプセーフではない動的な手動コードが多い。
- 広く採用されている単一の正式なイベント記述言語がない。

これらの理由から，新しい CORBA アプリケーションでは，定義済みの型なしチャネル，構造化チャネル，およびシーケンスチャネルは不適切です。ただし，これらのチャネルは，OMG 準拠および既存のアプリケーション用の VisiNotify ではサポートされます。新しいアプリケーションでは，OMG Typed Notification Service を使用することを検討してください。詳細については，23 ページの「Typed Event/Notification Service の使い方」を参照してください。

プッシュコンシューマアプリケーションの開発

プッシュコンシューマは，基本的には CORBA コールバックサーバーアプリケーションです。プッシュコンシューマは，プッシュコンシューマオブジェクトインプリメンテーションを提供します。プッシュコンシューマオブジェクトインプリメンテーションは，定義済み（型なし，構造化，またはシーケンス）のプッシュコンシューマインターフェースをサポートします。コンシューマアプリケーションは，このコンシューマオブジェクトをチャネルに接続して，イベントを受信します。

プッシュコンシューマアプリケーションを開発するには，次の 2 つの作業を実行します。

1 定義済み（型なし，構造化，またはシーケンス）のプッシュコンシューマインターフェースをサポートする，通常のプッシュコンシューマサーバーオブジェクトを実装します。これには，次の操作を行います。

- コンシューマサーバントを実装する。
- POA でサーバントをアクティブ化する。
- POA マネージャをアクティブ化する。

2 コンシューマオブジェクトをチャネルに接続します。これには，次の操作を行います。

- チャネルリファレンスを取得する。
- チャネルからコンシューマ管理を取得する。
- プロキシプッシュサプライヤを取得する。
- コンシューマオブジェクトをプロキシプッシュサプライヤに接続する。

プッシュコンシューマアプリケーションの開発を具体的に説明するために，構造化プッシュコンシューマを使用します。

C++ に関するメモ プッシュコンシューマのサンプルは，examples/vbroker/notify/basic_cpp/structPushConsumer.C にあります。

```
// 1. プッシュコンシューマサーバントを実装します。
   class StructuredPushConsumerImpl : public POA_CosNotifyComm::StructuredPushConsumer,
                                      public virtual PortableServer::RefCountServantBase
   {
      ...
     public:
      ...
      void push_structured_event(const CosNotification::StructuredEvent& event) { ... }
      ...
   };

   // コンシューマサーバー
   int main(int argc, char** argv)
   {
       // orb および POA を取得します ...
       ...

       // プッシュコンシューマサーバントを割り当てます。
       StructuredPushConsumerImpl* servant = new StructuredPUshConsumerImpl;

       // 2. POA でコンシューマサーバントをアクティブ化します。
       poa->activate_object(servant);

       // 3. POA をアクティブ化します。
       poa->the_POAManager()->activate();

       ...
       // 4. 何らかの方法でチャネルを取得します。
       CosNotifyChannelAdmin::EventChannel_var channel = ...;
```

```
    // 5. ここでは，デフォルト管理を使用します。
    CosNotifyChannelAdmin::ConsumerAdmin_var admin
                                        = channel->default_consumer_admin();

    // 6. この管理からプロキシプッシュサプライヤを取得します。
    CosNotifyChannelAdmin::ProxyID pxy_id;
    CosNotifyChannelAdmin::ProxySupplier_var proxy
         = admin->obtain_notification_push_supplier(
                 CosNotifyChannelAdmin::STRUCTURED_EVENT, pxy_id);

    CosNotifyChannelAdmin::StructuredProxyPushSupplier_var supplier
         = CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(proxy);

    // 7. コンシューマオブジェクトリファレンスを取得し，プロキシに接続します。
    CORBA::Object_var obj = poa->servant_to_reference(servant);
    CosNotifyComm::StructuredPushConsumer_var consumer
         = CosNotifyComm::StructuredPushConsumer::_narrow(obj);
    supplier->connect_structured_push_consumer(consumer);

    // 作業ループ
    orb->run();
}
```

Java に関するメモ プッシュコンシューマのサンプルは，examples/vbroker/notify/basic_java/StructPushConsumer.java にあります。

```
import org.omg.CosNotifyComm.*;
import org.omg.CosNotifyChannelAdmin,*;
import org.omg.CosNotification.*;

public class StructuredPushConsumer extends StructuredPushConsumerPOA
{
   ...
  // 1. プッシュコンシューマサーバントを実装します。
  public void push_structured_event(StructuredEvent event) { ... }
   ...

  public static int main(String[] args) {
    // orb および POA を取得します ...
    ...

    // プッシュコンシューマサーバントを割り当てます。
    StructuredPushConsumer servant = new StructuredPUshConsumer();

    // 2. POA でコンシューマサーバントをアクティブ化します。
    poa.activate_object(servant);

    // 3. POA をアクティブ化します。
    poa.the_POAManager().activate();

    ...
    // 4. 何らかの方法でチャネルを取得します。
    EventChannel channel = ...;

    // 5. ここでは，デフォルト管理を使用します。
    ConsumerAdmin admin = channel.default_consumer_admin();

    // 6. この管理からプロキシプッシュサプライヤを取得します。
    ProxyIDHolder pxy_id = new ProxyIDHolder();
    ProxySupplier proxy = admin.obtain_notification_push_supplier(
                ClientType.STRUCTURED_EVENT, pxy_id);

    StructuredProxyPushSupplier supplier
         = StructuredProxyPushSupplierHelper.narrow(proxy);

    // 7. コンシューマオブジェクトリファレンスを取得し，プロキシに接続します。
```

```
        org.omg.CORBA::Object obj = poa.servant_to_reference(servant);
        StructuredPushConsumer consumer = StructuredPushConsumerHelper.narrow(obj);
        supplier.connect_structured_push_consumer(consumer);

        // 作業ループ
        orb.run();
    }
}
```

プルコンシューマアプリケーションの開発

プルコンシューマは，基本的には CORBA クライアントです。プルコンシューマは，チャネル内でプロキシオブジェクトを取得して，プロキシに要求をアクティブに送信して，バッファリングされたイベントを取得します。

プルコンシューマアプリケーションを開発するには，次の 2 つの作業を実行します。

1 （オプション）定義済み（型なし，構造化，またはシーケンス）のプルコンシューマインターフェースをサポートする，プルコンシューマサーバーオブジェクトを実装します。これには，次の操作を行います。

- コンシューマサーバントを実装する。
- POA でサーバントをアクティブ化する。
- POA マネージャをアクティブ化する。

2 プロキシコンシューマリファレンスを取得し，そこからイベントを取得します。これには，次の操作を行います。

- チャネルリファレンスを取得する。
- チャネルからコンシューマ管理を取得する。
- プロキシプルサプライヤを取得する。
- コンシューマオブジェクト（または null）をプロキシプルサプライヤに接続する。
- プロキシプルサプライヤからイベントをアクティブにプルする。

プルコンシューマアプリケーションの開発を具体的に説明するために，構造化プルコンシューマを使用します。

C++ に関するメモ プッシュコンシューマのサンプルは，examples/vbroker/notify/basic_cpp/structPullConsumer.C にあります。

```
// コンシューマクライアント
   int main(int argc, char** argv)
   {
       // orb を取得します ...
       ...

       // 1．何らかの方法でチャネルを取得します。
       CosNotifyChannelAdmin::EventChannel_var channel = ...;

       // 2．ここでは，デフォルト管理を使用します。
       CosNotifyChannelAdmin::ConsumerAdmin_var admin
                                          = channel->default_consumer_admin();

       // 3．この管理からプロキシプルサプライヤを取得します。
       CosNotifyChannelAdmin::ProxyID pxy_id;
       CosNotifyChannelAdmin::ProxySupplier_var proxy
            = admin->obtain_notification_pull_supplier(
                    CosNotifyChannelAdmin::STRUCTURED_EVENT, pxy_id);

       CosNotifyChannelAdmin::StructuredProxyPullSupplier_var supplier
```

```
        = CosNotifyChannelAdmin::StructuredProxyPullSupplier::_narrow(proxy);

    // 4. プロキシに接続します。
    supplier->connect_structured_pull_consumer(NULL);

    // 5. プロキシプルサプライヤからイベントをプルします。
    for(int i=0;i<100;i++) {
       CosNotification::StructuredEvent_var event;
       event = supplier->pull_structured_event();
       ...
    }

    // 6. 正しくクリーンアップします。
    supplier->disconnect_structured_pull_supplier();
}
```

Java に関するメモ プルコンシューマのサンプルは，examples/vbroker/notify/basic_java/StructPullConsumer.java にあります。

```
import org.omg.CosNotifyComm.*;
import org.omg.CosNotifyChannelAdmin,*;
import org.omg.CosNotification.*;

public class StructuredPullConsumer
{
   ...
 public static int main(String[] args) {
    // orb を取得します ...
    ...

    // 1. 何らかの方法でチャネルを取得します。
    EventChannel channel = ...;

    // 2. ここでは，デフォルト管理を使用します。
    ConsumerAdmin admin = channel.default_consumer_admin();

    // 3. この管理からプロキシプルサプライヤを取得します。
    ProxyIDHolder pxy_id = new ProxyIDHolder();
    ProxySupplier proxy = admin.obtain_notification_pull_supplier(
                ClientType.STRUCTURED_EVENT, pxy_id);

    StructuredProxyPullSupplier supplier
         = StructuredProxyPullSupplierHelper.narrow(proxy);

    // 4. プロキシに接続します。
    supplier.connect_structured_pull_consumer(null);

    // 5. プロキシプルサプライヤからイベントをプルします。
    for(int i=0;i<100;i++) {
       StructuredEvent event = supplier.pull_structured_event();
       ...
    }

    // 6. 正しくクリーンアップします。
    supplier.disconnect_structured_pull_supplier();
 }
}
```

プッシュサプライヤアプリケーションの開発

プッシュサプライヤアプリケーションは，基本的には CORBA クライアントです。プッシュサプライヤアプリケーションは，コンシューマプロキシオブジェクトに対してアクティブに要求を行い，チャネルにイベントを送信します。

プッシュサプライヤアプリケーションの開発するには，次の 2 つの作業を実行します。

1 （オプション）定義済み（型なし，構造化，またはシーケンス）のプルコンシューマインターフェースをサポートする，プッシュサプライヤサーバーオブジェクトを実装します。これには，次の操作を行います。

- コンシューマサーバントを実装する。
- POA でサーバントをアクティブ化する。
- POA マネージャをアクティブ化する。

2 プロキシサプライヤリファレンスを取得し，そこにイベントを送信します。これには，次の操作を行います。

- チャネルリファレンスを取得する。
- チャネルからサプライヤ管理を取得する。
- プロキシプッシュコンシューマを取得する。
- サプライヤオブジェクト（または null）をプロキシプッシュコンシューマに接続する。
- プロキシプッシュコンシューマにイベントをアクティブにプッシュします。

プッシュサプライヤアプリケーションの開発を具体的に説明するために，構造化プッシュサプライヤを使用します。

C++ に関するメモ プッシュサプライヤのサンプルは，examples/vbroker/notify/basic_cpp/structPushSupplier.C にあります。

```
// プッシュサプライヤクライアント
int main(int argc, char** argv)
{
    // orb を取得します ...
    ...

    // 1. 何らかの方法でチャネルを取得します。
    CosNotifyChannelAdmin::EventChannel_var channel = ...;

    // 2. ここでは，デフォルト管理を使用します。
    CosNotifyChannelAdmin::SupplierAdmin_var admin
                                        = channel->default_supplier_admin();

    // 3. この管理からプロキシプッシュコンシューマを取得します。
    CosNotifyChannelAdmin::ProxyID pxy_id;
    CosNotifyChannelAdmin::ProxyConsumer_var proxy
         = admin->obtain_notification_push_consumer(
                 CosNotifyChannelAdmin::STRUCTURED_EVENT, pxy_id);

    CosNotifyChannelAdmin::StructuredProxyPushConsumer_var supplier
         = CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(proxy);

    // 4. プロキシに接続します。
    supplier->connect_structured_push_supplier(NULL);

    // 5. プロキシプッシュコンシューマにイベントをプッシュします。
    for(int i=0;i<100;i++) {
       CosNotification::StructuredEvent_var event = ...;
       Consumer->push_structured_event(event);
    }

    // 6. 正しくクリーンアップします。
    consumer->disconnect_structured_push_consumer();
}
```

Java に関するメモ プッシュサプライヤのサンプルは，examples/vbroker/notify/basic_java/StructPushSupplier.java にあります。

```
import org.omg.CosNotifyComm.*;
import org.omg.CosNotifyChannelAdmin,*;
import org.omg.CosNotification.*;

public class StructuredPushSupplier
{
   ...
 public static int main(String[] args) {
    // orb を取得します ...
    ...

    // 1．何らかの方法でチャネルを取得します。
    EventChannel channel = ...;

    // 2．ここでは，デフォルト管理を使用します。
    ConsumerAdmin admin = channel.default_supplier_admin();

    // 3．この管理からプロキシコンシューマを取得します。
    ProxyIDHolder pxy_id = new ProxyIDHolder();
    ProxyConsumer proxy = admin.obtain_notification_push_consumer (
                ClientType.STRUCTURED_EVENT, pxy_id);

    StructuredProxyPushConsumer consumer
         = StructuredProxyPushConsumerHelper.narrow(proxy);

    // 4．プロキシに接続します。
    consumer.connect_structured_push_supplier(null);

    // 5．プロキシプッシュコンシューマにイベントをプッシュします。
    for(int i=0;i<100;i++) {
       StructuredEvent event = ...;
       Consumer.push_structured_event(event);
    }

    // 6．正しくクリーンアップします。
    consumer.disconnect_structured_push_consumer();
  }
}
```

プルサプライヤアプリケーションの開発

プルサプライヤアプリケーションは，CORBA コールバックサーバーです。プルサプライヤアプリケーションは，プルサプライヤオブジェクトインプリメンテーションを提供します。プルサプライヤオブジェクトインプリメンテーションは，定義済み（型なし，構造化，またはシーケンス）のプルサプライヤインターフェースをサポートします。サプライヤアプリケーションは，このサプライヤオブジェクトをチャネルに接続して，イベントを送信します。

プルサプライヤアプリケーションを開発するには，次の 2 つの作業を実行します。

1 定義済み（型なし，構造化，またはシーケンス）のプルサプライヤインターフェースをサポートする，通常のプルサプライヤサーバーオブジェクトを実装します。これには，次の操作を行います。

- サプライヤサーバントを実装する。
- POA でサーバントをアクティブ化する。
- POA マネージャをアクティブ化する。

2 サプライヤオブジェクトをチャネルに接続します。これには，次の操作を行います。

- チャネルリファレンスを取得する。
- チャネルからサプライヤ管理を取得する。
- プロキシプルコンシューマを取得します。
- プルサプライヤオブジェクトをプロキシプルコンシューマに接続する。

プルサプライヤアプリケーションの開発を具体的に説明するために，構造化プルサプライヤを使用します。

C++ に関するメモ プルサプライヤのサンプルは，examples/vbroker/notify/basic_cpp/structPullSupplier.C にあります。

```
// 1. プルサプライヤサーバントを実装します。
class StructuredPullSupplierImpl : public POA_CosNotifyComm::StructuredPullSupplier,
                                   public virtual PortableServer::RefCountServantBase
{
   ...
  public:
   ...
    CosNotification::StructuredEvent* pull_structured_event() { ... }
    CosNotification::StructuredEvent* try_pull_structured_event(
                                  CORBA::Boolean& has_event) { ... }
   ...
};

// サプライヤサーバー
int main(int argc, char** argv)
{
    // orb および POA を取得します ...
    ...

    // プルサプライヤサーバントを割り当てます。
    StructuredPullSupplierImpl* servant = new StructuredPullSupplierImpl;

    // 2. POA でコンシューマサーバントをアクティブ化します。
    poa->activate_object(servant);

    // 3. POA をアクティブ化します。
    poa->the_POAManager()->activate();

    ...
    // 4. 何らかの方法でチャネルを取得します。
    CosNotifyChannelAdmin::EventChannel_var channel = ...;

    // 5. ここでは，デフォルト管理を使用します。
    CosNotifyChannelAdmin::SupplierAdmin_var admin
                                        = channel->default_supplier_admin();

    // 6. この管理からプロキシプルコンシューマを取得します。
    CosNotifyChannelAdmin::ProxyID pxy_id;
    CosNotifyChannelAdmin::ProxyConsumer_var proxy
         = admin->obtain_notification_pull_consumer(
                CosNotifyChannelAdmin::STRUCTURED_EVENT, pxy_id);

    CosNotifyChannelAdmin::StructuredProxyPullConsumer_var supplier
         = CosNotifyChannelAdmin::StructuredProxyPullConsumer::_narrow(proxy);

    // 7. サプライヤオブジェクトリファレンスを取得し，プロキシに接続します。
    CORBA::Object_var obj = poa->servant_to_reference(servant);
    CosNotifyComm::StructuredPullSupplier_var supplier
         = CosNotifyComm::StructuredPullSupplier::_narrow(obj);
    consumer->connect_structured_pull_supplier(supplier);

    // 作業ループ
```

```
    orb->run();
}
```

Java に関するメモ プルサプライヤのサンプルは，examples/vbroker/notify/basic_java/StructPullSupplier.java にあります。

```
import org.omg.CosNotifyComm.*;
import org.omg.CosNotifyChannelAdmin,*;
import org.omg.CosNotification.*;

public class StructuredPullSupplier extends StructuredPullSupplierPOA
{
   ...
  // 1. プッシュコンシューマサーバントを実装します。
  public StructuredEvent pull_structured_event() { ... }
  public StructuredEvent try_pull_structured_event(
                                    org.omg.CORBA.BooleanHolder has_event) {...}
   ...

  public static int main(String[] args) {
    // orb および POA を取得します ...
    ...

    // プルサプライヤサーバントを割り当てます。
    StructuredPullSupplier servant = new StructuredPullSupplier();

    // 2. POA でサプライヤサーバントをアクティブ化します。
    poa.activate_object(servant);

    // 3. POA をアクティブ化します。
    poa.the_POAManager()Activate();

    ...
    // 4. 何らかの方法でチャネルを取得します。
    EventChannel channel = ...;

    // 5. ここでは，デフォルト管理を使用します。
    ConsumerAdmin admin = channel.default_supplier_admin();

    // 6. この管理からプロキシプルコンシューマを取得します。
    ProxyIDHolder pxy_id = new ProxyIDHolder();
    ProxyConsumer proxy = admin.obtain_notification_pull_consumer(
                ClientType.STRUCTURED_EVENT, pxy_id);

    StructuredProxyPullConsumer consumer
         = StructuredProxyPullConsumerHelper.narrow(proxy);

    // 7. サプライヤオブジェクトリファレンスを取得し，プロキシに接続します。
    org.omg.CORBA::Object obj = poa.servant_to_reference(servant);
    StructuredPullSupplier supplier = StructuredPullSupplierHelper.narrow(obj);
    consumer.connect_structured_pull_supplier(supplier);

    // 作業ループ
    orb.run();
  }
}
```

Typed Event/Notification Service の使い方

OMG Event/Notification Service の定義済みイベント（型なし，構造化，シーケンス）は，メッセージ指向の方法を表します。この方法の欠点を次に示します。

- ユーザー定義の型付きチャネルより遅い。

- 通常，イベントサイズが大きい。
- イベントへのユーザーデータのパック，イベントからのユーザーデータのアンパックに必要なタイプセーフではない動的な手動コードが多い。
- 広く採用されている単一の正式なイベント記述言語がない。

そのため，新しいアプリケーションの開発時には，ユーザー定義の型付きイベントおよび OMG Typed Event/Notification Service の使用をお勧めします。OMG Typed Event/Notification Service を使用すれば，イベントインターフェースは，OMG ではなく，OMG IDL 言語を使用したユーザーアプリケーションによって事前に定義されます。次に，この方法を使用した場合の利点を示します。

- アプリケーションのイベントスループットが著しく高くなる。
- イベントサイズがかなり小さくなる。
- イベントパックおよびイベントアンパックの操作は，タイプセーフの IDL で生成された静的スタブ／スケルトンコードを利用する。
- イベントが正式に IDL によって定義される。

OMG Typed Event/Notification Service の使用については，小さな欠点があります。これには以下のような問題点が含まれます。

- Typed Event/Notification Service への接続は，（定義済みの）Event/Notification Service への接続より若干複雑である。アプリケーションは，追加のハンドラインプリメンテーションを提供したり，追加の接続オペレーション（get_typed_consumer() など）を実行する必要がある。以上の欠点は，先に説明した利点があることを考えれば，些細な欠点と言えます。
- Event/Notification Service を直接使用して，型付きプルを行う方法は，OMG では適切に定義されていません。OMG ソリューションには，実質的な作業が必要です。

VisiBroker Publish/Subscribe Adapter（PSA）アーキテクチャを使用すれば，この 2 つの問題を解決することができます。詳細については，第 4 章「Publish/Subscribe Adapter（PSA）の使い方」を参照してください。PSA を使用すると，通知サービスや型付き通知サービスへの接続手順を単純化して，統一することができます。また，型付きプルに対して明快なソリューションを提供します。

メモ この章では，OMG Typed Notification Service を使用して，型付きプッシュアプリケーションを直接開発する方法についてのみ説明します。型付きプルおよび PSA については，PSA の章で説明します。

この章で紹介するすべてのサンプルでは，ユーザー定義のイベントタイプとして，次の IDL インターフェース定義が使用されます。

```
// TMN.idl：型付きイベントの定義

// ユーザー定義のプラグマ
# pragma prefix "example.borland.com"

// ユーザー定義のモジュール
module TMN {

     // ユーザー定義のイベントインターフェース
     interface TypedEvent {
           void attributeValueChange(...);
           void qosAlarm(...);
           ...
     };
};
```

型付きプッシュコンシューマアプリケーションの開発

型付きプッシュコンシューマは，基本的には CORBA コールバックサーバーアプリケーションです。型付きプッシュコンシューマは，型付きプッシュコンシューマオブジェクトインプリメンテーションを提供します。型付きプッシュコンシューマオブジェクトインプリメンテーションは，ユーザー定義の IDL インターフェースをサポートします。コンシューマアプリケーションは，このコンシューマオブジェクトを型付きチャネルに接続して，型付きイベントを受信します。

型付きプッシュコンシューマアプリケーションを開発するには，次の 2 つの作業を実行します。

1 ユーザー定義の IDL インターフェースをサポートする, 通常のコンシューマサーバーオブジェクトを実装します。これには，次の操作を行います。

- <I> インターフェースサーバントなど, ユーザー定義の型付きコンシューマサーバントを実装する。
- ハンドラサーバントを実装する。ハンドラサーバントは，CosTypedNotifyComm::TypedPushConsumer インターフェースとその get_typed_consumer() オペレーションをサポートします。これにより，ユーザー定義の型付きコンシューマオブジェクトのリファレンスが戻されます（<I> インターフェースを戻すなど）。
- POA でユーザー定義の型付きサーバントをアクティブ化して，そのリファレンスを取得する。
- アクティブ化したハンドラオブジェクトをユーザー定義の型付きコンシューマオブジェクトリファレンス（<I> インターフェースなど）に渡す。
- POA マネージャをアクティブ化する。

2 コンシューマオブジェクトをチャネルに接続します。これには，次の操作を行います。

- 型付きチャネルリファレンスを取得する。
- チャネルから型付きコンシューマ管理を取得する。
- 型付きプロキシプッシュサプライヤを取得する。
- ハンドラオブジェクトを型付きプロキシプッシュサプライヤに接続する。

次のサンプルは，定義済みのイベントインターフェースの手順との比較です。型付きイベントを使用するには，プッシュコンシューマアプリケーションに追加のインプリメンテーションが必要です。

C++ に関するメモ 型付きプッシュコンシューマのサンプルは，examples/vbroker/notify/basic_cpp/typedPushConsumer.C にあります。

```
// 1．ユーザー定義の型付きコンシューマサーバントを実装します。
class TMNTypedEventImpl : public POA_TMN::TypedEvent,
                          public virtual PortableServer::RefCountServantBase
{
   ...
  public:
   ...
   void attributeValueChange (...) { ... }
   void qosAlarm(...) { ... }
   ...
};

// 2．ハンドラサーバントを実装します。
class HandlerImpl : public POA_CosTypedNotifyComm::TypedPushConsumer,
                    public virtual PortableServer::RefCountServantBase
{
    CORBA::Object_var   _the_typed_consumer; // I インターフェース
```

```
  public:
    HandlerImpl(CORBA::Object_ptr ref)
      : _the_typed_consumer(CORBA::Object::_duplicate(ref)) {}

    CORBA::Object_ptr get_typed_consumer() {
              // <I> インターフェースを戻します。
              return CORBA::Object::_duplicate(_the_typed_consumer); }
    ...
};

// 型付きコンシューマサーバー
int main(int argc, char** argv)
{
    // orb および POA を取得します ...
    ...

    // プッシュコンシューマサーバントを割り当てます。
    TMNTypedEventImpl* servant = new TMNTypedEventImpl;

    // 3. POA で型付きコンシューマをアクティブ化します。
    poa->activate_object(servant);

    // 4. 型付きコンシューマリファレンスを取得します。
    CORBA::Object_var obj = poa->servant_to_reference(servant);

    // 5. ハンドラサーバントを割り当て，型付きコンシューマリファレンスを渡します。
    HandlerImpl* handler = new HandlerImpl(obj);

    // 6. POA でハンドラオブジェクトをアクティブ化します。
    poa->activate_object(handler);

    // 7. POA をアクティブ化します。
    poa->the_POAManager()->activate();

    ...
    // 8. 何らかの方法で型付きチャネルを取得します。
    CosTypedNotifyChannelAdmin::TypedEventChannel_var channel = ...;

    // 9. ここでは，デフォルト管理を使用します。
    CosTypedNotifyChannelAdmin::TypedConsumerAdmin_var admin
                                        = channel->default_consumer_admin();

    // 10. イベントリポジトリ ID「IDL:example.borland.com/TMN/TypedEvent:1.0」を
    //    キーとして使用して，この管理からプロキシプッシュサプライヤを
    //    取得します。
    CosNotifyChannelAdmin::ProxyID pxy_id;
    CosTypedNotifyChannelAdmin::ProxySupplier_var proxy
         = admin->obtain_typed_notification_push_supplier(
             "IDL:example.borland.com/TMN/TypedEvent:1.0", pxy_id);

    // 11. ハンドラオブジェクトリファレンスを取得し，プロキシに接続します。
    CORBA::Object_var ref = poa->servant_to_reference(handler);
    CosTypedNotifyComm::TypedPushConsumer_var consumer
         = CosTypedNotifyComm::TypedPushConsumer::_narrow(ref);
    proxy->connect_typed_push_consumer(consumer);

    // 作業ループ
    orb->run();
}
```

Java に関するメモ 型付きプッシュコンシューマのサンプルは，examples/vbroker/notify/basic_java/TypedPushConsumerImpl.java にあります。

```
// 1. ユーザー定義の型付きコンシューマサーバントを実装します。
class TMNTypedEventImpl extends TMN.TypedEventPOA {
```

```
   ...
   public void attributeValueChange (...) { ... }
   public void qosAlarm(...) { ... }
   ...
}

// 2. ハンドラサーバントを実装します。
public class TypedPushConsumerImpl
         extends org.omg.CosTypedNotifyComm.TypedPushConsumer {
    org.omg.CORBA.Object _the_typed_consumer = null; // I インターフェース

    TypedPushConsumerImpl(org.omg.CORBA.Object ref) {
       _the_typed_consumer = ref;
    }
    org.omg.CORBA.Object get_typed_consumer() {
        // <I> インターフェースを戻します。
        return _the_typed_consumer; }
    ...


    public static void main(int argc, char** argv) {
       // orb および POA を取得します ...
       ...

       // プッシュコンシューマサーバントを割り当てます。
       TMNTypedEventImpl servant = new TMNTypedEventImpl();

       // 3. POA で型付きコンシューマをアクティブ化します。
       poa.activate_object(servant);

       // 4. 型付きコンシューマリファレンスを取得します。
       org.omg.CORBA.Object obj = poa.servant_to_reference(servant);

       // 5. ハンドラサーバントを割り当て，型付きコンシューマリファレンスを渡します。
       TypedPushConsumerImpl handler = new TypedPushConsumerImpl(obj);

       // 6. POA でハンドラオブジェクトをアクティブ化します。
       poa.activate_object(handler);

       // 7. POA をアクティブ化します。
       poa.the_POAManager()Activate();

       ...
       // 8. 何らかの方法で型付きチャネルを取得します。
       org.omg.CosTypedNotifyChannelAdmin.TypedEventChannel channel = ...;

       // 9. ここでは，デフォルト管理を使用します。
       org.omg.CosTypedNotifyChannelAdmin.TypedConsumerAdmin admin
                                       = channel.default_consumer_admin();

       // 10. イベントリポジトリ ID「IDL:example.borland.com/TMN/TypedEvent:1.0」を
       //    キーとして使用して，この管理からプロキシプッシュサプライヤを
       //    取得します。
       org.omg.CosNotifyChannelAdmin.ProxyIDHolder pxy_id_holder
         = new org.omg.CosNotifyChannelAdmin.ProxyIDHolder();
       org.omg.CosTypedNotifyChannelAdmin.ProxySupplier proxy
         = admin.obtain_typed_notification_push_supplier(
             "IDL:example.borland.com/TMN/TypedEvent:1.0", pxy_id_holder);

       // 11. ハンドラオブジェクトリファレンスを取得し，プロキシに接続します。
       org.omg.CORBA.Object ref = poa.servant_to_reference(handler);
       org.omg.CosTypedNotifyComm.TypedPushConsumer consumer
         = org.omg.CosTypedNotifyComm.TypedPushConsumerHelper.narrow(ref);
       proxy.connect_typed_push_consumer(consumer);

       // 作業ループ
```

```
            orb.run();
        }
    }
```

型付きプッシュサプライヤアプリケーションの開発

型付きプッシュサプライヤアプリケーションは，CORBA クライアントです。型付きプッシュサプライヤアプリケーションは，型付きコンシューマプロキシオブジェクトに対してアクティブに要求を行い，チャネルに型付きイベントを送信します。

型付きプッシュサプライヤアプリケーションを開発するには，次の 2 つの作業を実行します。

1 （オプション）型付きプッシュサプライヤサーバーオブジェクトを実装します。これには，次の操作を行います。

- サプライヤサーバントを実装する。
- POA でサーバントをアクティブ化する。
- POA マネージャをアクティブ化する。

2 プロキシサプライヤリファレンスを取得し，そこにイベントを送信します。これには，次の操作を行います。

- 型付きチャネルリファレンスを取得する。
- 型付きチャネルからサプライヤ管理を取得する。
- 型付きプロキシプッシュコンシューマを取得する。
- 型付きプロキシプッシュコンシューマで get_typed_consumer() を呼び出し，<I> インターフェースリファレンスを取得する。
- <I> インターフェースリファレンスにイベントをアクティブにプッシュする。

次のサンプルは，定義済みのイベントインターフェースの手順との比較です。型付きイベントを使用するには，get_typed_consumer() などの追加の手順が必要です。

C++ に関するメモ 型付きプッシュサプライヤのサンプルは，examples/vbroker/notify/basic_cpp/typedPushSupplier.C にあります。

```
// 型付きプッシュサプライヤクライアント
int main(int argc, char** argv)
{
    // orb を取得します ...
    ...

    // 1. 何らかの方法で型付きチャネルを取得します。
    CosTypedNotifyChannelAdmin::TypedEventChannel_var channel = ...;

     // 2. ここでは，デフォルト管理を使用します。
    CosTypedNotifyChannelAdmin::TypedSupplierAdmin_var admin
                                        = channel->default_supplier_admin();

     // 3. イベントリポジトリ ID「IDL:example.borland.com/TMN/TypedEvent:1.0」を
    //    キーとして使用して，この管理からプロキシプッシュサプライヤを
    //    取得します。
    CosTypedNotifyChannelAdmin::ProxyID pxy_id;
    CosTypedNotifyChannelAdmin::TypedProxyPushConsumer_var proxy
         = admin->obtain_typed_notification_push_consumer(
                 "IDL:example.borland.com/TMN/TypedEvent:1.0", pxy_id);

     // 4. プロキシに接続します。
    proxy->connect_typed_push_supplier(NULL);

     // 5. <I> インターフェースを取得します。
    CORBA::Object_var obj = proxy->get_typed_consumer();
```

```
    TMN::TypedEvent_var consumer = TMN::TypedEvent::_narrow(obj);


     // 6. イベントを <I> インターフェースにプッシュします。
    for(int i=0;i<100;i++) {
       consumer->attributeValueChange(...);
       consumer->qosAlarm(...);
    }

     // 7. バッファリングされたイベントを消去します。
    consumer->_non_existent();

     // 8. 正しくクリーンアップします。
    proxy->disconnect_typed_push_consumer();
}
```

Java に関するメモ 型付きプッシュコンシューマのサンプルは，examples/vbroker/notify/basic_java/TypedPushSupplier.java にあります。

```
import org.omg.CosTypedNotifyComm.*;
import org.omg.CosTypedNotifyChannelAdmin,*;
import org.omg.CosNotification.*;

public class TypedPushSupplierImpl
{
   ...

   // 型付きプッシュサプライヤクライアント
   public static void main(String[] args) {
      // orb を取得します ...
      ...

       // 1. 何らかの方法で型付きチャネルを取得します。
      org.omg.CosTypedNotifyChannelAdmin.TypedEventChannel_var channel = ...;

       // 2. ここでは，デフォルト管理を使用します。
      org.omg.CosTypedNotifyChannelAdmin.TypedSupplierAdmin admin
                                        = channel.default_supplier_admin();

       // 3. イベントリポジトリ ID「IDL:example.borland.com/TMN/TypedEvent:1.0」を
       //    キーとして使用して，この管理からプロキシプッシュサプライヤを
      //    取得します。
      Org.omg.CosTypedNotifyChannelAdmin.ProxyIDHolder pxy_id
                        = new org.omg.CosTypedNotifyChannelAdmin.ProxyIDHolder();
      CosTypedNotifyChannelAdmin::TypedProxyPushConsumer_var proxy
         = admin.obtain_typed_notification_push_consumer(
                 "IDL:example.borland.com/TMN/TypedEvent:1.0", pxy_id);

       // 4. プロキシに接続します。
      proxy.connect_typed_push_supplier(null);

       // 5. <I> インターフェースを取得します。
      org.omg.CORBA.Object obj = proxy.get_typed_consumer();
      TMN.TypedEvent consumer = TMN.TypedEventHelper.narrow(obj);


       // 6. イベントを <I> インターフェースにプッシュします。
      for(int i=0;i<100;i++) {
         consumer.attributeValueChange(...);
         consumer.qosAlarm(...);
      }

       // 7. バッファリングされたイベントを消去します。
      consumer._non_existent();
```

```
            // 8. 正しくクリーンアップします。
            proxy.disconnect_typed_push_consumer();
        }
    }
```

VisiNotify を使用した RMI ／ EJB アプリケーションの開発

J2EE 1.3 の導入に伴い, RMI-over-IIOP は J2EE インプリメンテーションで標準化されました。これにより，CORBA 環境と J2EE 環境の間の相互運用と相互接続をシームレスに行うことができます。J2EE は，基本的には，クライアント／サーバーアプリケーション用のフレームワークです。ただし，J2EE テクノロジでは，publish/subscribe アプリケーションは十分にサポートされていません。J2EE で定義されているソリューションは，Java Messaging Service（JMS）および Message Driver Bean（MDB）だけです。JMS は，完全にメッセージ指向のサービスで，主に既存のメッセージミドルウェアとの統合や相互接続に使用されます。MDB は，JMS の使用に続いて定義されます。MDB を使用すると，既存のメッセージミドルウェアは，JMS を介してエンタープライズ Bean にメッセージを送信できます。この点で，JMS ベースのソリューションと MDB ベースのソリューションは，通常，既存のメッセージ指向のミドルウェアが持つ同じ欠点を共有しています。これには以下のような問題点が含まれます。

- ユーザー定義のオブジェクト指向の型付きチャネルより遅い。
- 比較的イベントサイズが大きい。
- イベントへのユーザーデータのパック，イベントからのユーザーデータのアンパックに必要な，標準でないまたはタイプセーフでない動的な手動コードが多い。
- 広く採用されている単一の正式なイベント記述言語がない。

OMG Typed Event/Notification Service では，これらの問題を解決することができます。型付き通知は，RMI ／ EJB アプリケーションの publish/subscribe ミドルウェアとして使用できます。また，VisiNotify では，OMG 型付きチャネルと RMI ／ EJB 間で直接接続することができます。さらに,（標準マーシャリングまたはカスタムマーシャリングのどちらかにおける）CORBA の valuetype だけでなく，Java のシリアライズ可能なオブジェクトを直接サポートします。OMG Typed Event/Notification Service, J2EE 1.3, および VisiNotify それぞれの拡張機能にあるこれらの標準機能を使用すれば，OMG Notification Service を JMS プロバイダとしてマッピングした後，JMS ／ MDB を使用するのではなく，イベント駆動型の RMI ／ EJB アプリケーションを通常のオブジェクト指向のアプリケーションとして開発できます。次に，この方法の利点を示します。

- パフォーマンスが著しく改善される。
- イベントサイズが小さくなる。
- 静的なタイプセーフの RMI スタブとスケルトンがメッセージのパック／アンパックを実行する。
- イベントがユーザー定義の Java RMI インターフェースによって表される。

ここでは，RMI ／ EJB 環境における OMG Typed Event/Notification Service VisiNotify の使用方法について説明します。

この節のすべてのサンプルでは, RMI サーバーインターフェースまたは EJB コンシューマ Bean リモートインターフェースとして, このユーザー定義の Java RMI リモートインターフェースが使用されます。.

```
package TMN;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Notification extends Remote {
        void attributeValueChange(...) throws RemoteException;
```

```
        void qosAlarm(...) throws RemoteException;
        ...
}
```

RMI 型付きコンシューマの開発

RMI 型付きプッシュコンシューマは，基本的には，OMG Typed Notification Service に接続された RMI コールバックアプリケーションです。型付きプッシュコンシューマ RMI オブジェクトは，ユーザー定義の RMI インターフェースを実装します。RMI 型付きコンシューマは，わずかな違いはありますが，CORBA の型付きコンシューマとよく似ています。次に似ている点を示します。

- RMI オブジェクトは，POA で明示的にアクティブ化する必要がない。
- アプリケーションは，<I> インターフェースとして RMI オブジェクトの CORBA オブジェクトリファレンスを取得する必要がある（次のサンプルのステップ 4 を参照）。

このサンプルコードは，RMI 型付きプッシュコンシューマを示しています。

```
// 1．ユーザー定義の型付きコンシューマ RMI オブジェクトを実装します。
class RMINotifyImpl
      extends PortableRemoteObject
      implements TMN.Notification {
   ...
   public void attributeValueChange (...) { ... }
   public void qosAlarm(...) { ... }
   ...
}

// 2．ハンドラサーバントを実装します。
public class TypedPushConsumerImpl
        extends org.omg.CosTypedNotifyComm.TypedPushConsumer {
    org.omg.CORBA.Object _the_typed_consumer = null; // <I> インターフェース

    TypedPushConsumerImpl(org.omg.CORBA.Object ref) {
       _the_typed_consumer = ref;
    }
    org.omg.CORBA.Object get_typed_consumer() {
        // <I> インターフェースを戻します。
        return _the_typed_consumer; }
    ...


    public static void main(int argc, char** argv) {
       // orb および POA を取得します ...
       ...

        // 3．RMI コンシューマオブジェクトを割り当てます。
       RMINotifyImpl consumer = new RMINotifyImpl();

        // 4．RMI コンシューマの CORBA オブジェクトリファレンスを取得します。
       org.omg.CORBA.Object corba_obj
                = javax.rmi.CORBA.Util.getTie(consumer).thisObject();

        // 5．ハンドラサーバントを割り当て，型付きコンシューマリファレンスを渡します。
       TypedPushConsumerImpl handler = new TypedPushConsumerImpl(corba_obj);

        // 6．POA でハンドラオブジェクトをアクティブ化します。
       poa.activate_object(handler);

        // 6．POA をアクティブ化します。
       poa.the_POAManager()Activate();

       ...
```

```
        // 7. 何らかの方法で型付きチャネルを取得します。
        org.omg.CosTypedNotifyChannelAdmin.TypedEventChannel channel = ...;

        // 8. ここでは，デフォルト管理を使用します。
        org.omg.CosTypedNotifyChannelAdmin.TypedConsumerAdmin admin
                                        = channel.default_consumer_admin();

        // 9. この管理からプロキシプッシュサプライヤを取得します。
        org.omg.CosNotifyChannelAdmin.ProxyIDHolder pxy_id_holder
          = new org.omg.CosNotifyChannelAdmin.ProxyIDHolder();
        org.omg.CosTypedNotifyChannelAdmin.ProxySupplier proxy
          = admin.obtain_typed_notification_push_supplier( "RMI.Test" , pxy_id_holder);

        // 10. ハンドラオブジェクトリファレンスを取得し，プロキシに接続します。
        org.omg.CORBA.Object ref = poa.servant_to_reference(handler);
        org.omg.CosTypedNotifyComm.TypedPushConsumer consumer
          = org.omg.CosTypedNotifyComm.TypedPushConsumerHelper.narrow(ref);
        proxy.connect_typed_push_consumer(consumer);

        // 作業ループ
        orb.run();
    }
}
```

RMI 型付きサプライヤの開発

RMI 型付きサプライヤは，CORBA の同等品とよく似ています。ただし，get_typed_consumer() から戻される I リファレンスを，一致する RMI スタブにナローイングする必要がある点を除きます（次のサンプルのステップ 6 を参照）。

このサンプルコードは，RMI 型付きプッシュサプライヤを示しています。

```
import org.omg.CosTypedNotifyComm.*;
import org.omg.CosTypedNotifyChannelAdmin,*;
import org.omg.CosNotification.*;

public class TypedPushSupplierImpl
{
   ...

   // 型付きプッシュサプライヤクライアント
   public static void main(String[] args) {
      // orb を取得します ...
      ...

      // 1. 何らかの方法で型付きチャネルを取得します。
      org.omg.CosTypedNotifyChannelAdmin.TypedEventChannel_var channel = ...;

      // 2. ここでは，デフォルト管理を使用します。
      org.omg.CosTypedNotifyChannelAdmin.TypedSupplierAdmin admin
                                        = channel.default_supplier_admin();

      // 3. この管理から型付きプロキシプッシュコンシューマを取得します。
      Org.omg.CosTypedNotifyChannelAdmin.ProxyIDHolder pxy_id
                        = new org.omg.CosTypedNotifyChannelAdmin.ProxyIDHolder();
      CosTypedNotifyChannelAdmin::TypedProxyPushConsumer_var proxy
         = admin.obtain_typed_notification_push_consumer(
                 "RMI.Test, pxy_id);

      // 4. プロキシに接続します。
      proxy.connect_typed_push_supplier(null);

      // 5. <I> インターフェースを取得します。
      org.omg.CORBA.Object obj = proxy.get_typed_consumer();
```

```
        // 6. CORBA オブジェクトリファレンスを RMI スタブにナローイングします。
        TMN.Notification consumer = (TMN.Notification)PortableRemoteObject.
                                        narrow(obj, TMN.Notification.class);

        // 7. イベントを <I> RMI スタブにプッシュします。
        for(int i=0;i<100;i++) {
           consumer.attributeValueChange(...);
           consumer.qosAlarm(...);
        }

        // 8. バッファリングされたイベントを消去します。
        com.inprise.vbroker.orb.BufferedEvents.flush();

        // 9. 正しくクリーンアップします。
        proxy.disconnect_typed_push_consumer();
    }
}
```

EJB Bean を型付き通知コンシューマとして開発する

EJB 型付きイベント Bean は，MDB 以外であれば，どのような種類の Bean（セッションまたはエンティティ，ステートレスまたはステートフル）でもかまいません。EJB 型付きイベント Bean は，関連付けられた特定のユーザー定義リモートインターフェースで宣言されているように，イベントオペレーションを実装します。

このサンプルコードは，ユーザー定義の TMN.Notification リモートインターフェースのプッシュコンシューマとして，EJB Bean を表します。

```
import javax.ejb.*;

// 1. Bean のインプリメンテーション
public class TMNNotifyBean implements SessionBean {
   ...
   // Bean のリモートインターフェースで宣言したオペレーションを実装します。
   public void attributeValueChange (...) { ... }
   public void qosAlarm(...) { ... }
   ...
}
```

この型付き EJB Bean インプリメンテーションを構築および配布すると，次の操作を実行できるようになります。

- JNDI からホームインターフェースを取得する。
- ホームインターフェースからリモートインターフェースを取得する。
- リモートインターフェースを OMG 型付き通知チャネルに接続する。

このバージョンでは，コマンドラインユーティリティの *subtool* が追加され，JNDI 名を確認することで，型付き RMI コンシューマとして EJB Bean をサブスクライブできるようになりました。型付きイベント EJB Bean を OMG 型付き通知チャネルに接続するには，次の subtool コマンドを使用します。

```
% subtool [-channel <ior>|-admin <ior>] ¥
         -home <jndi_name> ¥
         -type typed ¥
         -key <proxy_key>
```

オプションの意味は次のとおりです。

- -channel オプションまたは -admin オプションは，サブスクライブポイントとしてチャネルまたはコンシューマ管理オブジェクトを指定します。
- -home <jndi_name> は，Bean のホームインターフェースの JNDI 名を subtool に伝えます。

- -type typed オプションは，Bean のリモートインターフェースに型付きコンシューマとして接続するように subtool に伝えます。
- -key <proxy_key> オプションは，どのキーパラメータを obtain_typed_notification_push_supplier() に使用するかを subtool に伝えます。

このサンプルは，subtool を使用して，型付きイベント Bean を OMG 型付き通知チャネルにサブスクライブする方法を示しています。

```
% subtool  -channel  corbaloc::127.0.0.1:14100/default_channel ¥
        -home  stock_home  -type  typed  -key  Stock
```

EJB Bean を構造化通知コンシューマとして開発する

MDB 以外であれば，任意のタイプの EJB Bean（セッションまたはエンティティ，ステートレスまたはステートフル）を EJB 構造化イベント Bean として使用できます。この EJB 構造化イベント Bean は，VisiNotify Structured Notification Service に接続して，RMI 以外の CORBA アプリケーションから生じた構造化イベントを受け取ることができます。EJB 構造化イベント Bean は，必須のオペレーションの中でも特に，void push_structured_event（org.omg.CosNotification.StructuredEvent）オペレーションを実装します。このオペレーションは，EJB 構造化イベント Bean とそのリモートインターフェース内ではオーバーロードしません。

型付きイベント Bean とは異なり，構造化イベント Bean は，VisiNotify の拡張機能によってサポートされます。VisiNotify は，接続されたコンシューマが構造化イベント EJB Bean であることを検知すると，特殊な方法で，CORBA 構造化サプライヤアプリケーションに送信された StructuredEvent 構造を StructuredEvent 値ボックスに変換します。

このサンプルは，構造化プッシュコンシューマとして EJB Bean を示しています。

```
import javax.ejb.*;

 // Bean のインプリメンテーション
public class MyStructuredNotifyBean implements SessionBean {
   ...
    public void push_structured_event(
                        org.omg.CosNotification.StructuredEvent event) { ... }
   ...
}
```

この構造化イベント Bean を構築および配布したら，そのリモートインターフェースを特定の Visinotify の構造化イベントチャネルに接続します。この Bean のリモートインターフェースは，push_structured_event() オペレーションを宣言する必要があります。このサンプルコードは，次の接続を示しています。

```
import java.rmi.Remote;
import java.rmi.RemoteException;

 // Bean のリモートインターフェース
public interface MyStructuredInterface extends Remote {
    public void push_structured_event(
                org.omg.CosNotification.StructuredEvent event)  throws RemoteException;
        ...
}
```

ORB 型システムにより，この構造化イベント Bean のリモートインターフェースを構造化イベントコンシューマとして OMG Notification Service 構造化チャネルに直接接続することは禁じられています。構造化イベント Bean を VisiNotify 構造化通知チャネルチャネルに接続するには，subtool コマンドを使用します。

```
% subtool [-channel <ior>|-admin <ior>] ¥
          -home <jndi_name> ¥
          -type struct
```

オプションの意味は次のとおりです。

- -channel オプションまたは -admin オプションは，サブスクライブポイントとしてチャネルまたはコンシューマ管理オブジェクトを指定します。
- -home <jndi_name> は，Bean のホームインターフェースの JNDI 名を subtool に伝えます。
- -type struct オプションは，Bean のリモートインターフェースに構造化コンシューマとして接続するように subtool に伝えます。

このサンプルは，subtool を使用して，構造化イベント Bean を VisiNotify Structured Notification Channel にサブスクライブする方法を示しています。

```
% subtool  -channel  corbaloc::127.0.0.1:14100/default_channel ¥
            -home  stock_home  -type struct
```

VisiBroker イベントバッファリング／バッチ

イベントバッファリングおよびバッチは，VisiBroker 5.1 で実装されるメカニズムで，VisiNotify のイベントスループットを最適化します。デフォルトでは，イベントは，VisiNotify に大きなバッチメッセージとして書き出される前に，サプライヤ側のスタブ内でバッファリングされます。また，VisiNotify は，コンシューマが VisiNotify 5.1 の上位で動作していることを検出すると，一緒にイベントをバッファリングおよびバッチしようとします。

サプライヤ側のイベントバッファリングを無効にする

サプライヤアプリケーションでは，vbroker.orb.supplier.eventBatch を false に設定することにより，サプライヤ側のイベントバッファリングを無効にすることができます。次に例を示します。

```
% typedPushSupplier ...-Dvbroker.orb.supplier.eventBatch=false
```

または

```
% vbj ...StructPushSupplier ...-Dvbroker.orb.supplier.eventBatch=false
```

コンシューマ側のイベントバッファリングを無効にする

コンシューマアプリケーションでも，vbroker.orb.consumer.eventBatch を false に設定することにより，VisiNotify がイベントを一括して送信できないようにすることができます。次に例を示します。

```
% typedPushConsumer ...-Dvbroker.orb.consumer.eventBatch=false
```

または

```
% vbj ...StructPushConsumerImpl ...-Dvbroker.orb.consumer.eventBatch=false
```

サプライヤアプリケーション内のバッファリングされたイベントを消去する

次の状態が生じると，サプライヤ側の VisiBroker ランタイムはイベントを消去します。

イベントバッファがいっぱいの場合：これはスタブ単位のレベルの消去です。このレベルのデフォルトのイベントバッファサイズは，32K です。特定のサプライヤアプリケーションでは，vbroker.orb.supplier.eventBufferSize を使用して，このサイズを 8K ～ 64K の間で指定できます。たとえば，次のようになります。

```
% typedPushSupplier ...-Dvbroker.orb.supplier.eventBufferSize=48000
```

バッファリングされたイベントの数が，最大バッチサイズを超えた場合：これはスタブ単位のレベルの消去です。スタブがそのバッファ内に保持できるデフォルトのイベント最大

数は，128 です。サプライヤアプリケーションでは，vbroker.orb.supplier.maxBatchSize を使用して，このサイズを 256 未満の値に指定できます。次に例を示します。

```
% vbj ...UntypedPushSupplier ...-Dvbroker.orb.supplier.eventBatchSize=32
```

内部バッファの消去がタイムアウトの場合：イベントが，一括消去されます。タイムアウトになると，すべてのスタブにあるバッファリングされたすべてのイベントが消去されます。デフォルトのタイムアウトの間隔は 2,000 ミリ秒（2 秒）です。サプライヤアプリケーションでは，vbroker.orb.supplier.eventBatchTimerInterval を使用して，この時間を 100 ミリ秒（0.1 秒）～ 10,000 ミリ秒（10 秒）の間の値に指定できます。たとえば，次のようになります。次に例を示します。

```
% typedPushSupplier ...-Dvbroker.orb.supplier.eventBatchTimerInterval=5000
```

サプライヤがバッファリングできないオペレーションをスタブで呼び出した場合：これはスタブ単位のレベルの消去です。次に，各スタブでバッファリング可能なオペレーションを示します。

- 型なしプロキシコンシューマスタブでは, push() オペレーションだけがバッファリング可能です。
- 構造化プロキシコンシューマスタブでは, push_structured_event() オペレーションだけがバッファリング可能です。
- シーケンスプロキシコンシューマスタブでは，push_structured_events() オペレーションだけがバッファリング可能です。

メモ disconnect_..._push_consumer() オペレーションまたは _non_existent() オペレーションをプロキシスタブ （上述）で呼び出すと，バッファリングされたすべてのイベントが消去されます。

- 型付きチャネルの <I> インターフェーススタブについては，擬似でないすべてのオペレーションがバッファリング可能です。

したがって, サプライヤアプリケーションでは, <I> インターフェーススタブで _non_existent() オペレーションを呼び出すと，そのバッファリングされたイベントを消去することができます。型付きプロキシコンシューマスタブで disconnect_typed_push_consumer() を呼び出しても，対応する <I> スタブ内のバッファは消去されません。アプリケーションは，<I> インターフェーススタブの call _non_existent() を明示的に呼び出してから，プロキシスタブの disconnect_typed_push_consumer() を呼び出す必要があります。

BufferedEvent.flush() を呼び出す Java アプリケーション

Java サプライヤアプリケーションは，明示的に com.inprise.vbroker.orb.BufferedEvents.flush() を呼び出して消去することができます。これは，一括のイベント消去です。これは，Java.rmi.Remote インターフェースには，イベントの消去に使用できる擬似オペレーションがないので，VisiBroker RMI アプリケーションをサポートするためです。この静的メソッドを呼び出すと，すべてのスタブ内のすべてのイベントが消去されます。

VisiNotify の初期リファレンス

デフォルトでは，コマンドラインで -Dvbroker.notify.listener.port=<port> を使用しない限り，VisiNotify は TCP ポート番号 14100 を使用します。したがって，OMG Notification Service で指定されているように，Channel ファクトリと型付きチャネルファクトリの URL は次のようになります。

```
corbaloc::<host>:14100/NotificationService
corbaloc::<host>:14100/TypedNotificationService
```

ここで，<host> には，VisiNotify ホストマシンのドメイン名またはドット付きの IP アドレスを指定します。VisiNotify サーバーは，デフォルトのチャネルも作成します。このデフォルトのチャネルの URL は次のとおりです。

```
corbaloc::<host>:14100/default_channel
```

この URL は，サプライヤアプリケーションとコンシューマアプリケーションの ORB に登録することができます。この場合，次に示す 2 つの OMG 標準化構成を使用します。

1 -ORBInitRef ORB_init() コマンドラインオプション。例

```
-ORBInitRef NotificationService=corbaloc::127.0.0.1:14100/NotificationService
```

または

```
-ORBInitRef TypedNotificationService=corbaloc::127.0.0.1:14100/
TypedNotificationService
```

2 ORB::register_initial_reference()。例

```
orb.register_initial_reference("TypedNotificationService",
 orb.string_to_object(
  "corbaloc::127.0.0.1:14100/TypedNotificationService");
```

これらを初期サービスとして登録すると，resolve_initial_reference() を使用できるようになります。

第 4 章

Publish/Subscribe Adapter（PSA）の使い方

この章では，VisiBroker Publish/Subscribe Adapter（PSA）について説明します。PSA は，主として OMG Event/Notification Service とともに動作するプログラミングモデルおよびコンポーネントです。PSA は，低レベルの OMG Notification Service インターフェースを使用するアプリケーションと相互運用できます。

はじめに

「最良のクライアント／サーバーミドルウェア製品の 1 つ」である CORBA は，OMG オブジェクト指向の ORB アーキテクチャをベースとする従来のクライアント／サーバーアプリケーションを堅固にサポートします。ただし，publish/subscribe アプリケーションのサポートという点では，CORBA にはいくつかの欠点があります。多くのエンタープライズビジネスアプリケーションでは，publish/subscribe 通信モデルは，クライアント／サーバーモデルと同じくらい重要です。CORBA ミドルウェアインフラストラクチャで publish/subscribe 通信モデルを直接サポートすることにより，開発者は，システムソリューションの再設計ではなく，ビジネスロジックの実装に集中できます。これにより，開発作業を大幅に軽減することができます。

それにもかかわらず，ORB レベルの publish/subscribe 通信モデルのサポートは，サードパーティの ORB ベンダーとともに ORB 内では，実質的に省略されてきました。CORBA 開発分野においては，publish/subscribe 通信モデルは，「2 流のサブジェクト」と考えられています。その結果，アプリケーション開発者は，Event/Notification Service などの COS レベルのソリューションに頼らざるを得なくなります。COS レベルのソリューションは，オブジェクト指向ではなくメッセージ指向です。COS Event/Notification Service の publish/subscribe は，レプリケートされたクライアント／サーバー通信としてモデリングされます。次に，このモデリングの欠点について説明します。

- オブジェクトが，非常に低レベルで抽出される点。セマンティクスの大きなすき間は，アプリケーション開発者が埋める必要があります。コンシューマプロキシ，サプライヤプロキシなど，クライアント／サーバー通信の低レベルの概念およびオブジェクトを直接操作し，相互接続を直接再調整する必要があります。
- オブジェクトモデルにおける緊密な結合の使用。チャネル接続モデル，メッセージ形式（構造化，型付きなど），およびメッセージ転送モデル（プッシュ／プル）は直交します

が，緊密に結合しています。これらのコンポーネントの 1 つに変更を加えると，ほかの部分も影響を受けます。特に構造化チャネルから型付きチャネルに変更した場合や，型付きプッシュから型付きプルに変更した場合です。

メモ CORBA 以外にも，新しい通信モデルを同じオブジェクト抽象レベルでサポートしない配布オブジェクトミドルウェアがあります。たとえば，RMI ／ EJB 環境内では，元の Java オブジェクトモデルや RMI オブジェクトモデルではなく，メッセージ指向のモデル（つまり，JMS ／ MDB）が使用されます。

この章で説明している Publish/Subscribe Adapter（PSA）を使用すれば，上述の問題に対処できます。PSA は，主に，OMG 準拠の Notification Service の上位で動作するプログラミングモデルおよびソフトウェアコンポーネントです。したがって，PSA は，サードパーティの OMG Notification Service インプリメンテーションとともに使用することもできます。また，低レベルの OMG Notification Service インターフェースを使って直接構築されたアプリケーションとも互換性があります。

PSA の基本的な機能の 1 つは，チャネル接続に関連した詳細を非表示にすることです。通常，CORBA publish/subscribe アプリケーションは，アプリケーションのコンシューマオブジェクトが，特定のチャネルからイベントを受信できるようにすることを主な目的として設計されます。チャネルは，通常，チャネルリファレンスまたはコンシューマ管理リファレンスによって指定します。コンシューマオブジェクトは，通常，その POA とオブジェクト ID によって指定します。直接 OMG Notification Service を使用した場合，コンシューマオブジェクトをチャネルに接続するには，複数の手順が必要になります。一方，PSA を使用した場合，この接続を完了するために必要なオペレーションは 1 つだけです。

PSA の基本的な概念を説明するために，このサンプルでは，型付きイベントコンシューマアプリケーションのコーディング方法を示します。型付きイベントは，IDL インターフェースによって定義されているものとします。

```
// TMN.idl : 型付きイベントの定義
#pragma prefix "examples.borland.com"
module TMN {
     interface TypedEvent {
           void attributeValueChange(...);
           ...
     };
};
```

まず，型付きイベントコンシューマがイベントを受け取るには，ユーザー定義のイベントインターフェーススケルトンである POA_TMN::TypedEvent から派生するサーバントインプリメンテーションを提供する必要があります。

```
// 1. 型付きサーバントを実装します。
# include "TMNEvents_s.hh"
class TMNTypedEventImpl : public POA_TMN::TypedEvent,
                          public PortableServer::RefCountServantBase
{
   public:
     void attributeValueChange(...) { ... };
     ...
};
```

次に，POA でこのサーバントをアクティブ化します。

```
int main(argc, argv)
{
     ...
     // 2. orb および POA 環境を取得します。
     CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
     CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
     PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
     // 3. 型付きサーバントを割り当てます。
     TMNTypedEventImpl* servant = new TMNTypedEventImpl();
     // 4. POA でこれをアクティブ化します。
     poa->activate_object(servant);
```

この時点で，型付きイベントアプリケーションは，通常の型付きコンシューマアプリケーションとして処理され，特に追加される作業はありません。これが，通常のクライアント／サーバーのサンプルであれば，アプリケーションは，POA からオブジェクトリファレンスを作成して，クライアントにそのオブジェクトリファレンスを渡すという作業が発生します。いずれにしても，これは publish/subscribe コンシューマのサンプルであるので，アプリケーションは，イベントパブリッシャであるクライアントに直接リファレンスを渡す必要はありません。そのかわり，コンシューマは，特定のチャネルやコンシューマ管理のオブジェクトリファレンスに接続する必要があります。

PSA では，コンシューマをチャネルに「接続」するのではなく，「サブスクライブ」するだけです。

```
    // 5. 何らかの方法で，このコンシューマにチャネルリファレンスが与えられます。
    CORBA::Object_var channel = ... ;
    // 6. コンシューマサーバントのオブジェクト ID を取得します。
    PortableServer::ObjectId_var oid = poa->servant_to_id(servant);
    // 7. POA を PSA にナローイングします。
    PortableServerExt::PSA_var psa = PortableServerExt::PSA::_narrow(poa);
    // 8. チャネルにサブスクライブします。
    PortableServerExt::SubjectScheme scheme = {
           PortableServerExt::CHANNEL_ADDR,
           PortableServerExt::TYPED_SUBJECT,
           (const char*)"IDL:example.borland.com/TMN/TypedEvent:1.0",
           PortableServerExt::PUSH_EVENT };
    psa->subscribe(scheme, channel, oid, CORBA::NameValuePairSeq());
    // 9. コンシューマの作業ループ
    poa->the_POAManager()->activate();
    orb->run();
}
```

上のコードで示したように，アプリケーションは，PSA を使用して，型付きサーバントインプリメンテーションを作成するだけです。get_typed_consumer() をサポートするために，CosTypedNotifyComm::TypedPushConsumer サーバントは必要ありません。また，「接続」には複数（6 手順）のオペレーションが必要ですが，サブスクライブに必要な手順は 1 つだけです。

次に，Java コードによる同等のサンプルを示します。

```
import com.inprise.vbroker.PortableServerExt.*;
// 1. 型付きサーバントを実装します。
public class TMNTypedEventImpl : extend TMN.TypedEventPOA,
{
    public void attributeValueChange(...) { ... }
};
public class TypedPushConsumerImpl
{
    public static void main(String[] args)
    {
       ...
       // 2. orb および PSA 環境を取得します。
       org.omg.CORBA.ORB orb = ORB_init(args, null);
       org.omg.PortableServer.POA poa
         = org.omg.PortableServer.POA.orb.resolve_initial_references("RootPOA");
       // 3. 型付きサーバントを割り当てます。
       TMNTypedEventImpl servant = new TMNTypedEventImpl();
       // 4. これをルート PSA でアクティブ化します。
       poa.activate_object(servant);
       // 5. 何らかの方法で，このコンシューマにチャネルリファレンスが与えられます。
       org.omg.CORBA.Object channel = ...;
       // 6. コンシューマサーバントのオブジェクト ID を取得します。
       org.omg.PortableServer.ObjectId oid = psa.servant_to_id(servant);
       // 7. org.omg.PortableServer.POA を com.inprise.vbroker.PSA にナローイングします。
       PSA psa = PSA.narrow(poa);
       // 8. チャネルにサブスクライブします。
       SubjectScheme scheme = new SubjectScheme(
            SubjectAddressScheme.CHANNEL_ADDR,
```

```
                SubjectInterfaceScheme.TYPED_SUBJECT,
                "IDL:example.borland.com/TMN/TypedEvent:1.0",
                SubjectDeliveryScheme.PUSH_EVENT);
            psa.subscribe(scheme, channel, oid, null);
             // 9. 作業ループ
             poa.the_POAManager().activate();
             orb.run();
        }
    }
```

このサンプルは，PSA が OMG Event/Notification Service や Typed Event/Notification Service とともに動作するしくみを明確に示しています。さらに，管理／プロキシなどの低レベルの通知サービスオブジェクトやオペレーションから，CORBA publish/subscribe アプリケーションを保護することで，このアプリケーションを単純化するしくみについても説明しています。

この章の後半では，PSA がイベントインターフェースから接続ロジックを分離して，モデルを転送する方法について説明します。PSA 内にあるサブスクライブなどの接続ロジックが，イベントインターフェースや転送モデルの影響を受けることはありません。たとえば，構造化コンシューマを型付きコンシューマに変更したり，型付きコンシューマをプッシュからプルに変更する場合は，コンシューマのサブスクライブロジックを変更する必要はありません。サブジェクトスキームのフラグを変更するだけです。このような変更を PSA を使用しないで行うには，コンシューマ接続ロジックのコードを大きく修正する必要があります。また，この章では，さまざまなアプリケーション用のサンプルを提供し，PSA の機能と用途について説明します。

PSA リファレンスと PSA インターフェース IDL

PSA は，POA の拡張機能で，POA で定義されたすべてのオペレーションをサポートします。VisiBroker 5.1 の POA リファレンスは，「RootPOA」および「RootPSA」をとる resolve_initial_references() (実際には同じ内部リファレンス返す) によって PSA リファレンスにナローイングできます。

このサンプルコードは，ルート PSA の取得方法を示しています。

C++

```
// C++ でルート PSA を取得します。
CORBA::Object_var ref = orb->resolve_initial_references("RootPSA");
PortableServerExt::PSA_var psa = PortableServerExt::_narrow(ref);
```

Java

```
// Java でルート PSA を取得します。
// パブリッシャ／サブスクライバアダプタを取得します。
org.omg.CORBA.Object ref = orb.resolve_initial_references("RootPOA");
PSA psa = PSAHelper.narrow(ref);
```

PSA は，PortableServerExt モジュールで定義され，（間接的に）PortableServer::POA から派生します。

```
module PortableServerExt {
      interface POA : PortableServer::POA {
      readonly attribute CORBA::PolicyList the_policies;
      };
      enum SubjectAddressScheme {
           SUBSCRIBE_ADMIN_ADDR,
           PUBLISH_ADMIN_ADDR,
           CHANNEL_ADDR,
           SUBJECT_ADDR
      };
      enum SubjectInterfaceScheme {
           TYPED_SUBJECT,
           UNTYPED_SUBJECT,
           STRUCTURED_SUBJECT,
           SEQUENCE_SUBJECT
     };
     enum SubjectDeliveryScheme {
```

```
        PUSH_EVENT,
        PULL_EVENT
    };
    typedef string SubjectInterfaceId;
    struct SubjectScheme {
        SubjectAddressScheme address_scheme;
        SubjectInterfaceScheme interface_scheme;
  SubjectInterfaceId  interface_id;
  SubjectDeliveryScheme delivery_scheme;
};
typedef Object     Subject;
typedef CORBA::OctetSequence PublishSubscribeDesc;
typedef PublishSubscribeDesc SubscribeDesc;
typedef PublishSubscribeDesc PublishDesc;
exception InvalidSubjectScheme { long error; };
exception InvalidSubscribeDesc { long error; };
exception InvalidPublishDesc { long error; };
exception InvalidProperties { CORBA::StringSequence names; };
exception ChannelException { string repository_id; }
// パブリッシャ／サブスクライバアダプタ
interface PSA : POA {
  // サブジェクトオブザーバを登録します。
  SubscribeDesc subscribe(
    in SubjectScheme   the_subject_scheme,
    in Subject     the_subject,
    in PortableServer::ObjectId the_observer_id,
    in CORBA::NameValuePairSeq the_properties )
        raises( InvalidSubjectScheme,
          InvalidProperties,
          ChannelException );
  // サブジェクトプロバイダを登録します。
  PublishDesc publish(
    in SubjectScheme  the_subject_scheme,
    in Subject   the_subject,
    in PortableServer::ObjectId the_pullable_publisher_id,
    in CORBA::NameValuePairSeq the_properties )
        raises( InvalidSubjectScheme,
          InvalidProperties,
          ChannelException );
  // サブジェクトからオブザーバの登録を解除します。
  void unsubscribe(
   in SubscribeDesc  the_subscribe_desc )
       raises( InvalidSubscribeDesc,
         ChannelException );
  // プロバイダの登録を解除します（プルモード）。
  void unpublish(
   in PublishDesc   the_publish_desc)
       raises( InvalidPublishDesc,
         ChannelException );
  // サブジェクトを一時停止して，
  // 登録済みオブザーバにプッシュするか，登録済みプロバイダから
  // プルします。
  void suspend(
    in PublishSubscribeDesc  the_desc)
       raises( ChannelException );
  // サブジェクトを再開して，
  // 登録済みオブザーバにプッシュするか，登録済みプロバイダから
  // プルします。
  Void resume(
    in PublishSubscribeDesc  the_desc)
       raises( ChannelException );
  //  （型付き）イベントをプルして，登録済みオブザーバに
  // 送信します。
  unsigned long pull_and_dispatch(
    in SubscribeDesc  the_subscribe_desc,
    in unsigned long  max_count,
```

```
    in boolean   block_pulling,
    in Boolean   async_dispatch)
       raises( InvalidSubscribeDesc,
         InvalidSubjectScheme,
         ChannelException );

  // （型付き）イベントをプルして，指定されたビジターによる
  // イベントへの「ビジット」を受け入れます。
  Unsigned long pull_and_visit(
    in SubscribeDesc  the_subscribe_desc,
    in unsigned long  max_count,
    in Boolean   block_pulling,
    in PortableServer::Servant the_visitor)
       raises( InvalidSubscribeDesc,
         InvalidSubjectScheme,
         ChannelException );
  Subject the_subject_addr(
    in PublishSubscribeDesc  the_desc)
       raises( InvalidSubjectScheme );
  // 低レベルなアクセス
  Object the_proxy_addr(
    in PublishSubscribeDesc  the_desc)
       raises( InvalidSubjectScheme );
 };
 ...
};
```

VisiBroker（5.1 以降）では，すべての POA が内部的に PSA として実装されます。したがって，VisiBroker 5.1 のあらゆる POA リファレンスは，必ず PSA リファレンスにナローイングされます。

このサンプルコードは，POA を PSA にナローイングする方法を示しています。

C++

```
// C++ で POA を PSA にナローイングします。
PortableServer::POA_var poa = parent_poa->create_POA(...);
PortableServerExt::PSA_var psa = PortableServerExt::_narrow(poa);
```

Java

```
// Java で POA を PSA にナローイングします。
org.omg.PortableServer.POA poa = parent_poa.create_POA(...);
com.inprise.vbroker.PortableServerExt.PSA psa
      = com.inprise.vbroker.PortableServerExt.PSAHelper.narrow(poa);
```

ユーザーサンプル

次のサンプルでは，COS 通知メソッドで記述されたアプリケーションコードと PSA を比較しています。パラメータは次のとおりです。

- 構造化プッシュコンシューマ
- 型付きプッシュコンシューマ
- 構造化プッシュサプライヤ
- 型付きプッシュサプライヤ

構造化プッシュコンシューマ

次の表は，通知接続メソッド（左列）および PSA（右列）で記述した，同じ構造化プッシュコンシューマアプリケーションを比較しています。主な相違点は次のとおりです。

- PSA の接続コードは，3 手順から 1 手順に単純化されている。
- PSA を使用したプッシュコンシューマは，通常のサーバーアプリケーションとよく似ている。

このサンプルコードは，チャネルに構造化コンシューマを接続／サブスクライブする方法を C++ で示しています。

Notification Service インターフェースを使用した構造化プッシュコンシューマ（basic/cpp/structPushConsumer.C）

```
// コンシューマサーバントを実装します。
class StructuredPushConsumerImpl :
 public POA_CosNotifyComm::
                 StructuredPushConsumer,
 Public PortableServer::RefCountServantBase
{
  public:
   void push_structured_event(...) {...}
   ...
};
using namespace CosNotifyChannelAdmin;
int main(int argc, char** argv)
{
// orb および POA 環境を取得します。
  CORBA::ORB_ptr orb
         = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb-> resolve_initial_references("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
// チャネルリファレンスを取得します。
  EventChannel_var channel = ...;
  // コンシューマサーバントを割り当てます。
  StructuredPushConsumerImpl* servant = new StructuredPushConsumerImpl();
  // これをルート POA でアクティブ化します。
  poa->activate_object(servant);
  // コンシューマオブジェクトリファレンスを取得します。
  CORBA::Object_var obj = poa->servant_to_reference(servant);
  CosNotifyComm::StructuredPushConsumer_var
     consumer = CosNotifyComm::StructuredPushConsumer::_narrow(obj);
  // チャネルに接続します。
  // 1．デフォルト管理を取得します。
  ConsumerAdmin_var admin =
     channel->default_consumer_admin();
  // 2．プロキシを作成します。
  ProxyID proxy_id;
  ProxySupplier_var proxy = admin->
     obtain_notification_push_supplier(STRUCTURED_EVENT, proxy_id);
  // スタブにナローイングします。
  StructuredProxyPushSupplier_var supplier = StructuredProxyPushSupplier::
                          _narrow(proxy);
  // 3．プロキシサプライヤに接続します。
  supplier-> connect_structured_push_consumer(consumer);
  // 作業ループ
  orb->run();
}
```

PSA を使用した構造化プッシュコンシューマ（psa/cpp/structPushConsumer.C）

```
// コンシューマサーバントを実装します。
class StructuredPushConsumerImpl :
 public POA_CosNotifyComm::
                 StructuredPushConsumer,
 Public PortableServer::RefCountServantBase
{
  public:
    void push_structured_event(...) {...}
    ...
};
// チャネル型固有の名前空間はありません。
int main(int argc, char** argv)
{
  // orb および PSA 環境を取得します。
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb-> Resolve_initial_references("RootPSA");
  PortableServerExt::PSA_var psa = PortableServerExt::PSA::_narrow(obj);
  // チャネルリファレンスを取得します。
```

```
  CORBA::Object_var channel = ...;
  // コンシューマサーバントを割り当てます。
  StructuredPushConsumerImpl* servant = new StructuredPushConsumerImpl();
  // これをルート PSA でアクティブ化します。
  psa->activate_object(servant);
  // コンシューマオブジェクト ID を取得します。
  PortableServer::ObjectId_var oid = poa->servant_to_id(servant);
  // チャネルにサブスクライブします。

  // サブジェクトスキームを指定します。
  PortableServerExt::SubjectScheme scheme =
  { PortableServerExt::CHANNEL_ADDR, PortableServerExt::STRUCTURED_SUBJECT,
    (const char*)"", PortableServerExt::PUSH_EVENT };
  // 1. サブスクライブします。
   psa->subscribe(scheme, channel, oid, CORBA::NameValuePairSeq());
  // 作業ループ
  orb->run();
}
```

このサンプルコードは，チャネルに構造化コンシューマを接続／サブスクライブする方法を **Java** で示しています。

Notification Service インターフェースを使用した構造化プッシュコンシューマ（basic_java/StructPushConsumerImpl.java）

```
Import org.omg.CORBA.*;
import org.omg.PortableServer.*;
Import org.omg.CosNotifyComm.*;
// コンシューマサーバントを実装します。
public class StructuredPushConsumerImpl : extend StructuredPushConsumerPOA
{
 public void push_structured_event(...) {...}
   ...
 public static void main(String[] args){
  // orb および POA 環境を取得します。
  ORB orb = ORB_init(args, null);
  Object obj
      = orb.resolve_initial_references("RootPOA");
  POA poa = POA.narrow(obj);
  // チャネルリファレンスを取得します。
  EventChannel channel = ...;
  // コンシューマサーバントを割り当てます。
  StructuredPushConsumerImpl servant = New StructuredPushConsumerImpl();
  // これをルート POA でアクティブ化します。
  poa.activate_object(servant);
  // コンシューマオブジェクトリファレンスを取得します。
  Object ref
     = poa.servant_to_reference(servant);
  StructuredPushConsumer consumer = StructuredPushConsumer.narrow(ref);
  // チャネルに接続します。
  // 1. デフォルト管理を取得します。
  ConsumerAdmin admin = channel.default_consumer_admin();
  // 2. プロキシを作成します。
  ProxyID proxy_id;
  ProxySupplier proxy
     = admin.obtain_notification_push_supplier(STRUCTURED_EVENT, proxy_id);
  // スタブにナローイングします。
  StructuredProxyPushSupplier supplier
     = StructuredProxyPushSupplier. narrow(proxy);
  // 3. プロキシサプライヤに接続します。
  supplier. Connect_structured_push_consumer(Consumer);
  // 作業ループ
  orb.run();
}
```

PSA を使用した構造化プッシュコンシューマ（psa_java/structPushConsumer.java）

```
import org.omg.CORBA.*;
Import com.inprise.vbroker.
                     PortableServerExt.*;
// コンシューマサーバントを実装します。
public class StructuredPushConsumerImpl :
```

```
extend CosNotifyComm.StructuredPushConsumer
{
 public void push_structured_event(...) {...}
    ...
 public static void main(String[] args) {
  // orb および PSA 環境を取得します。
  ORB orb = ORB_init(args, null);
  Object obj
      = orb.resolve_initial_references("RootPSA");
  PSA psa = PSA.narrow(obj);
  // チャネルリファレンスを取得します。
  CORBA.Object. channel = ...;
  // コンシューマサーバントを割り当てます。
  StructuredPushConsumerImpl servant = New StructuredPushConsumerImpl();
  // これをルート PSA でアクティブ化します。
  psa.activate_object(servant);
  // コンシューマオブジェクト ID を取得します。
  org.omg.PortableServer.ObjectId oid = psa.servant_to_id(servant);
  // チャネルにサブスクライブします。

  // サブジェクトスキームを指定します。
  SubjectScheme scheme = new SubjectScheme(SubjectAddressScheme.CHANNEL_ADDR,
    SubjectInterfaceScheme.STRUCTURED_SUBJECT,(const char*)"",
    SubjectDeliveryScheme.PUSH_EVENT);
 // 1. サブスクライブします。
  psa.subscribe(
     scheme, channel, oid, null);
  // 作業ループ
  orb.run();
}
```

型付きプッシュコンシューマ

次の表は，通知接続メソッド（左列）および PSA（右列）で記述したコードを示しています。主な相違点は次のとおりです。

- PSA の接続コードは，6 手順から 1 手順に単純化されている。
- PSA　を使用した型付きコンシューマアプリケーションには，get_typed_consumer() にプロキシオブジェクトを提供するアプリケーションが必要ない。PSA は，透過的にこの機能を提供します。
- PSA を使用した型付きプッシュコンシューマは，通常のサーバーアプリケーションとよく似ている。
- PSA を使用した型付きプッシュコンシューマは，PSA 型付きプッシュコンシューマとほとんど同じ。PSA 型付きプッシュコンシューマでは，PSA が，異なるチャネルのあらゆる変更からアプリケーションを保護します。

このサンプルコードは，チャネルにコンシューマを接続／サブスクライブする方法を **C++** で示しています。

Notification Service インターフェースを使用した型付きプッシュコンシューマ（basic/cpp/typedPushConsumer.C）

```
// プロキシコンシューマサーバントを実装します。
class TypedPushConsumerImpl :
 public POA_CosTypedNotifyComm::
                TypedPushConsumer,
 Public PortableServer::RefCountServantBase
{
   CORBA::Object_var _I;
  Public:
   TypedPushConsumerImpl(
        CORBA::Object_ptr I) : _I(
        CORBA::Object::_duplicate(I)) {}
   CORBA::Object_ptr get_typed_consumer() {
     return CORBA::Object::_duplicate(_I);
```

```
   }
   ...
};
// 型付きサーバントを実装します。
class TMNTypedEventImpl :
 public POA_TMN::TypedEvent,
 public PortableServer::RefCountServantBase
{
  public:
    void attributeValueChange(...);
    ...
};
using namespace CosTypedNotifyChannelAdmin;
int main(int argc, char** argv)
{

  // orb および POA 環境を取得します。
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb-> resolve_initial_references("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
 // チャネルリファレンスを取得します。
  TypedEventChannel channel = ... ;
  // 型付きサーバントを割り当てます。
  TMNTypedEventImpl* typed_servant = new TMNTypedEventImpl();
  // POA でこれをアクティブ化します。
  poa->activate_object(typed_servant);
  // そのリファレンスを取得します。
  CORBA::Object_var typed_ref = poa->servant_to_reference(typed_servant);
  // チャネルに接続します。
  // 1. プロキシコンシューマを割り当てます。
  TypedPushConsumerImpl* servant = new TypedPushConsumerImpl(typed_ref);
  // 2. これをルート POA でアクティブ化します。
  poa->activate_object(servant);
  // 3. コンシューマオブジェクトリファレンスを取得します。/
  obj = poa->servant_to_reference(servant);
  CosTypedNotifyComm::TypedPushConsumer_var
     consumer = CosTypedNotifyComm::TypedPushConsumer::_narrow(obj);
  // 4. デフォルト管理を取得します。
  TypedConsumerAdmin_var admin = channel->default_consumer_admin();
  // 5. プロキシを作成します。
  CosNotifyChannelAdmin::ProxyID proxy_id;
  TypedProxySupplier_var proxy = admin-> obtain_notification_push_supplier(
          "IDL:example.borland.com/"
          "TMN/TypedEvent:1.0", proxy_id);
  // 6. プロキシサプライヤに接続します。
  proxy->connect_typed_push_consumer(consumer);
  // 作業ループ
  orb->run();
}
```

PSA を使用した型付きプッシュコンシューマ（psa/cpp/typedPushConsumer.C）

```
// プロキシコンシューマを実装する必要はありません。
// PSA は get_typed_consumer() をサポートする
// プロキシを透過的に提供します。
// 型付きサーバントを実装します。
class TMNTypedEventImpl :
 public POA_TMN::TypedEvent,
 public PortableServer::RefCountServantBase
{
  public:
    void attributeValueChange (...) ;
    ...
};

// チャネル型固有の名前空間はありません。
int main(int argc, char** argv)
{
  // orb および PSA 環境を取得します。
```

```
  CORBA::ORB_ptr orb
          = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb-> Resolve_initial_references("RootPSA");
  PortableServerExt::PSA_var psa = PortableServerExt::PSA::_narrow(obj);
  // チャネルリファレンスを取得します。
  CORBA::Object_var channel = ... ;
  // 型付きサーバントを割り当てます。
  TMNTypedEventImpl* typed_servant = new TMNTypedEventImpl();
  // これをルート PSA でアクティブ化します。
  psa->activate_object(typed_servant);
  // そのオブジェクト ID を取得します。
  PortableServer::ObjectId_var oid = poa->servant_to_id(typed_servant);
  // チャネルにサブスクライブします。
  // サブジェクトスキームを指定します。
  PortableServerExt::SubjectScheme scheme = { PortableServerExt::CHANNEL_ADDR,
    PortableServerExt::TYPED_SUBJECT, (const char*)"IDL:example.borland.com"
                "TMN/TypedEvent:1.0", PortableServerExt::PUSH_EVENT };
  // 1. サブスクライブします。
   psa->subscribe(scheme, channel, oid, CORBA::NameValuePairSeq());
  // 作業ループ
  orb->run();
}
```

このサンプルコードは，チャネルに型付きコンシューマを接続／サブスクライブする方法を **Java** で示しています。

Notification Service インターフェースを使用した型付きプッシュコンシューマ（basic_java/TypedPushConsumerImpl.C）

```
import org.omg.CORBA.*;
Import org.omg.PortableServer.*;
Import org.omg.CosNotifyComm.*;
// 型付きサーバントを実装します。
public class TMNTypedEventImpl : extend TMN.TypedEventPOA,
{
    public void attributeValueChange(...);
       ...
  }
  public class TypedPushConsumerImpl : extend TypedPushConsumerPOA
  {
    Object _I = null;
    public TypedPushConsumerImpl(Object I) { _I = I; }
    // get_typed_consumer() を実装します。
    public object get_typed_consumer({return _I; }
      ...
    public static void main(String[] args){
     // orb および POA 環境を取得します。
     ORB orb = ORB_init(args, null);
     Object obj = orb.resolve_initial_references( "RootPOA");
     POA poa = POA.narrow(obj);
     // チャネルリファレンスを取得します。
     TypedEventChannel channel = ... ;
     // 型付きサーバントを割り当てます。
     TMNTypedEventImpl typed_servant = new TMNTypedEventImpl();
     // POA でこれをアクティブ化します。
     poa.activate_object(typed_servant);

     // そのリファレンスを取得します。
     Object typed_ref
       = poa.servant_to_reference( typed_servant);
     // チャネルに接続します。
     // 1. プロキシコンシューマを割り当てます。
     TypedPushConsumerImpl servant = New TypedPushConsumerImpl(typed_ref);
     // 2. これをルート POA でアクティブ化します。
     poa.activate_object(servant);
     // 3. コンシューマオブジェクトリファレンスを取得します。
     obj = poa->servant_to_reference(servant);
     TypedPushConsumer Consumer = TypedPushConsumer.narrow(obj);
     // 4. デフォルト管理を取得します。
```

```
    TypedConsumerAdmin admin = Channel.default_consumer_admin();
    // 5. プロキシを作成します。
    CosNotifyChannelAdmin::ProxyID proxy_id;
    TypedProxySupplier proxy = admin. Obtain_notification_push_supplier(
           "IDL:example.borland.com/"
           "TMN/TypedEvent:1.0", proxy_id);
   // 6. プロキシサプライヤに接続します。
   proxy.connect_typed_push_consumer(consumer);
   // 作業ループ
   orb.run();
}
```

PSA を使用した型付きプッシュコンシューマ（psa_java/TypedPushConsumerImpl.C）

```
import org.omg.CORBA.*;
Import com.inprise.vbroker.
                      PortableServerExt.*;
// 型付きサーバントを実装します。
public class TMNTypedEventImpl : extend TMN.TypedEventPOA,
{
public void attributeValueChange(...);
    ...
}
public class TypedPushConsumerImpl
{
 // プロキシコンシューマを実装する必要はありません。
 // PSA は get_typed_consumer() を
 // サポートするプロキシを
 // 透過的に提供します。
 Public static void main(String args) {
  // orb および PSA 環境を取得します。
  ORB orb = ORB_init(args, null);
  Object obj
   = orb.resolve_initial_references("RootPSA");
  PSA psa = PSA.narrow(obj);
 // チャネルリファレンスを取得します。
  Object channel = ... ;
  // 型付きサーバントを割り当てます。
  TMNTypedEventImpl typed_servant = new TMNTypedEventImpl();
  // これをルート PSA でアクティブ化します。
  psa.activate_object(typed_servant);
  // そのオブジェクト ID を取得します。
  PortableServer::ObjectId oid = psa.servant_to_id(typed_servant);

   // チャネルにサブスクライブします。
   // サブジェクトスキームを指定します。
   SubjectScheme scheme = new
    (SubjectAddressScheme.CHANNEL_ADDR, SubjectInterfaceScheme.TYPED_SUBJECT,
     (const char*)"IDL:example.borland.com" "TMN/TypedEvent:1.0",
     SubjectDeliveryScheme.PUSH_EVENT);
   // 1. サブスクライブします。
   psa.subscribe( scheme, channel, oid, null);
  // 作業ループ
  orb.run();
}
```

この 2 つのサンプルは，PSA が，構造化コンシューマおよび型付きコンシューマ両方の通知チャネルに接続する手順を徹底的に簡略化および統一する方法を明確に示しています。また，PSA がイベント形式の選択と接続ロジックを分離する方法についても説明しています。低レベルの COS Notificatin Service を直接使用した場合，構造化チャネルと型付きチャネル間のアプリケーションコードは大きく異なります。一方，PSA を使用した場合は，2 つのサンプルのサブスクライブロジックの違いはほとんどありません。

構造化プッシュサプライヤおよび型付きプッシュサプライヤのサンプル

次に示す 2 つのサンプルでは，型付きプッシュサプライヤと構造化プッシュサプライヤが，通知接続メソッド（左列）および PSA（右列）で記述されています。PSA を使用した型付きプッシュサプライヤは，PSA 構造化プッシュサプライヤとほとんど同じです。どちらの場合も，PSA は，構成が異なる各チャネルからアプリケーションを保護します。

チャネルへの構造化サプライヤ

このサンプルコードは，チャネルに構造化サプライヤを接続／サブスクライブする方法を C++ で示しています。

通知サービスインターフェースを使用した構造化プッシュサプライヤ（basic_cpp/structPushSupplier.C）

```
Using namespace CosNotifyChannelAdmin;
int main(int argc, char** argv)
{
  // orb および POA 環境を取得します。
  CORBA::ORB_ptr orb
          = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb-> Resolve_initial_references("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
  // チャネルリファレンスを取得します。
  EventChannel channel = ... ;
  // チャネルに接続します。
  // 1. デフォルト管理を取得します。
  ConsumerAdmin_var admin = Channel->default_supplier_admin();
  // 2. プロキシを作成します。
  ProxyID proxy_id;
  ProxyConsumer_var proxy = admin->
     obtain_notification_push_consumer(STRUCTURED_EVENT, proxy_id);
  // 3. StructuredProxyConsumer を取得します。
  StructuredProxyPushConsumer_var consumer
     = StructuredProxyPushConsumer::_narrow(proxy);
  // 型付きイベントインターフェースをプッシュします。
  for(;;) {
     consumer->push_structured_event(...);
     ...
  }
  ...
}
```

通知サービスインターフェースを使用した構造化プッシュサプライヤ（basic_cpp/structPushSupplier.C）

```
int main(int argc, char** argv)
{
  // orb および PSA 環境を取得します。
  CORBA::ORB_ptr orb
          = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb-> Resolve_initial_references("RootPSA");
  PSA_var psa = PSA::_narrow(obj);

  // チャネルリファレンスを取得します。
  CORBA::Object_var channel = ... ;
  // チャネルにパブリッシュします。
  // 1. 公開します。
  PortableServerExt::SubjectScheme scheme = { PortableServerExt::CHANNEL_ADDR,
     PortableServerExt::STRUCTURED_SUBJECT, (const char*)"",
     PortableServerExt::PUSH_EVENT };
  PortableServerExt::PublishDesc_var desc = psa->publish(scheme, channel,
     PortableServer::ObjectId(), CORBA::NameValuePairSeq());
  // 2. StructuredProxyConsumer を取得します。
  CORBA::Object_var obj = psa->the_subject_addr(desc);
  StructuredProxyPushConsumer_var consumer
     = StructuredProxyPushConsumer::_narrow(proxy);

  // 型付きイベントインターフェースをプッシュします。
```

```
  for(;;) {
     consumer->push_structured_event(...);
     ...
  }
  ...
}
```

チャネルへの型付きサプライヤ

このサンプルコードは，チャネルに型付きサプライヤを接続／サブスクライブする方法を C++ で示しています。

Notification Service インターフェースを使用した型付きプッシュサプライヤ（basic_cpp/typedPushSupplier.C）

```
using namespace CosTypedNotifyChannelAdmin;
int main(int argc, char** argv)
{
  // orb および POA 環境を取得します。
  CORBA::ORB_ptr orb
          = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb-> Resolve_initial_references("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
  // チャネルリファレンスを取得します。
  TypedEventChannel channel = ... ;
  // チャネルに接続します。
  // 1. デフォルト管理を取得します。
  TypedConsumerAdmin_var admin = Channel->default_supplier_admin();
  // 2. プロキシを作成します。
  CosNotifyChannelAdmin::ProxyID proxy_id;
  TypedProxyConsumer_var proxy = admin->obtain_notification_push_consumer(
          "IDL:example.borland.com/" "TMN/TypedEvent:1.0", proxy_id);
  // 3. I インターフェースを取得します。
  CORBA::Object_var obj = proxy->get_typed_consumer();
  TMN::TypedEvent_var consumer = TMN::TypedEvent::_narrow(obj);
  // 型付きイベントインターフェースをプッシュします。
  for(;;) {
     consumer->attributeValueChange(...);
     ...
  }
  ...
}
```

PSA を使用した型付きプッシュサプライヤ（psa_cpp/typedPushSupplier.C）

```
int main(int argc, char** argv)
{
  // orb および PSA 環境を取得します。
  CORBA::ORB_ptr orb
          = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb-> Resolve_initial_references("RootPSA");
  PSA_var psa = PSA::_narrow(obj);

  // チャネルリファレンスを取得します。
  CORBA::Object_var channel = ... ;
  // チャネルにパブリッシュします。
  // 1. 公開します。
 PortableServerExt::SubjectScheme scheme =
  { PortableServerExt::CHANNEL_ADDR, PortableServerExt::TYPED_SUBJECT,
    (const char*)"IDL:example.borland.com" "TMN/TypedEvent:1.0",
    PortableServerExt::PUSH_EVENT };
 PortableServerExt::PublishDesc_var desc = psa->publish(scheme, channel,
      PortableServer::ObjectId(), CORBA::NameValuePairSeq());
  // 2. I インターフェースを取得します。
  CORBA::Object_var obj = psa->the_subject_addr(desc);
  TMN::TypedEvent_var consumer = TMN::TypedEvent::_narrow(obj);

  // 型付きイベントインターフェースをプッシュします。
  for(;;) {
     consumer->attributeValueChange(...);
     ...
```

```
    }
    ...
}
```

これらのサンプルは，Notification Service インターフェースを使用した場合，構造化サプライヤアプリケーションと型付きサプライヤアプリケーションの接続コードと接続手順の間に著しい違いがあることを示しています。一方，PSA を使用した場合は，両方のアプリケーションの接続コードと接続手順はほとんど同じになります。

メモ　この章で使用するすべてのサンプルは，同梱の VisiNotify と PSA サンプルを凝縮したものです。これらのサンプルはディレクトリに置かれます。examples/vbe/notify/basic_<cpp|java> and examples/vbe/notify/psa_<cpp|Java>.

PSA を使ってサブジェクトをサブスクライブする

PSA のサブスクライブオペレーションを使用すると，コンシューマオブジェクトは，通知／イベントソースに結び付けられ，（プッシュまたはプル）イベントメッセージを受信することができます。これは，次に示す有効な publish/subscribe 構成をすべて含む非常に幅広い概念です。

- OMG 通知／イベントチャネルに接続する。
- マルチキャストグループを結合して，内部コンシューマ ID マッピングに外部キーを確立する。
- IIOP 以外のメッセージ指向ミドルウェアに接続する。

PSA は，低レベルの転送メカニズムやチャネル／メッセージ形式の種類に関係なく，単一のプログラミングモデルを使用して，これらの構成をサポートします。このバージョンでは，PSA は，OMG 通知／イベントチャネル（全 4 種類のチャネル）のサブスクライブだけをサポートします。PSA サブスクライブオペレーションは，次のように定義されます。

```
SubscribeDesc subscribe(
    in SubjectScheme   the_subject_scheme,
    in Subject     the_subject,
    in PortableServer::ObjectId the_observer_id,
    in CORBA::NameValuePairSeq the_properties )
         raises( InvalidSubjectScheme,
                  InvalidProperties,
                  ChannelException );
```

PSA を COS Notification の上位で使用すると，このオペレーションは，コンシューマ管理およびプロキシサプライヤを取得し，接続を行う，すべての低レベルのオペレーションを実行します。型付きサブジェクトをサブスクライブするために，PSA は内部でハンドラプロキシオブジェクトを作成および管理して，get_typed_consumer() オペレーションをサポートします。また，アプリケーションは，アプリケーションが指定された型付き <I> インターフェースをサポートするオブザーバサーバントインプリメンテーションを提供するだけです。

SubjectScheme

subscribe() への最初のパラメータは SubjectScheme で，次のように定義されます。

```
struct SubjectScheme {
SubjectAddressScheme  address_scheme;
 SubjectInterfaceScheme interface_scheme;
 SubjectInterfaceId  interface_id;
 SubjectDeliveryScheme delivery_scheme;
};
```

SubjectScheme パラメータは，サブジェクトリファレンスのアドレススキーム，インターフェーススキーム，インターフェースリポジトリ ID（型付きチャネル用のみ），および配布スキームを指定できます。

address_scheme フィールドには，サブジェクトリファレンスを指定します。たとえば，アドレスを指定すれば，プッシュイベントやアドレスに直接使用して，サブスクライブだけ行うことができます。現在，このフィールドには，サブスクライブ用の 3 つの値，**SUBSCRIBE_ADMIN_ADDR**，**CHANNEL_ADDR**，および **SUBJECT_ADDR** があります。それぞれの値は，subscribe() オペレーションへのサブジェクトリファレンスが，OMG Notification Consumer Admin，OMG Notification Channel（または型付きチャネル），または直接的なイベントプッシュアドレスであることを示しています。

address_scheme フィールドの 3 つの値を使用すると，アプリケーションは，次に示す方法でサブスクライブすることができます。

- **SUBSCRIBE_ADMIN_ADDR** - subscribe() へのサブジェクトリファレンスは，OMG Notification Consumer Admin リファレンスです。PSA は，管理で obtain_<...>_supplier() を呼び出し，管理でプロキシを割り当てて，プロキシで connect_<...>_consumer() を呼び出します。プロキシに接続されたコンシューマリファレンスは，null（プルモードコンシューマ用），または observer_id パラメータを使用してこの PSA から生成されたプッシュコンシューマオブジェクトリファレンスのどちらかになります。型付きチャネルでは，get_typed_consumer() および get_typed_supplier() は，自動的に PSA によって処理されます。
- **CHANNEL_ADDR** - subscribe() へのサブジェクトリファレンスは，OMG Notification Channel（または型付きチャネル）です。PSA は，チャネルで calls _get_default_consumer_admin() を呼び出すだけで，デフォルトの管理を取得し，このコンシューマ管理リファレンスを介して，接続として処理します。
- **SUBJECT_ADDR** - subscribe() へのサブジェクトリファレンスは，直接的なイベントプッシュアドレスです。たとえば，マルチキャスト IOR や型付き <I> インターフェースです。型付き以外のチャネルでは，プロキシプッシュコンシューマになります。PSA は，_get_MyAdmin()/_get_MyChannel()/_get_default_consumer_admin() を呼び出して，コンシューマ管理を介して，接続として処理します。型付きチャネルについては，これはすでにプッシュ <I> インターフェースになっています。PSA はリファレンスの中からコンシューマ管理コンポーネントを探して（現在はサポートされていません），コンシューマ管理を介して，接続として処理します。

また，アプリケーションは，**SubjectInterfaceScheme** および **SubjectDeliveryScheme** を指定する必要があります。

次に **SubjectInterfaceScheme** の有効な値を示します。

- **TYPED_SUBJECT** - サブジェクトは，マルチキャストまたは OMG 型付き通知チャネルを使用します。
- **UNTYPED_SUBJECT** - サブジェクトは，OMG 型なし通知チャネルを使用します。
- **STRUCTURED_SUBJECT** - サブジェクトは，OMG 構造化通知チャネルを使用します。
- **SEQUENCE_SUBJECT** - サブジェクトは，OMG シーケンス通知チャネルを使用します。

次に **SubjectDeliveryScheme** の有効な値を示します。

- **PUSH_EVENT** - サブジェクトは，マルチキャストまたは OMG Push Notification モード（4 種類ある OMG イベントのいずれか）を使用します。
- **PULL_EVENT** - サブジェクトは，OMG Pull Notification モード（4 種類ある OMG イベントのいずれか）を使用します。

型付きチャネルに接続するには，<I> インターフェースのリポジトリ ID も指定する必要があります。このリポジトリは，暗黙的なイベントフィルタとして使用されます。

Subscribe() へのサブジェクトリファレンス，オブザーバ ID，およびプロパティ

subscribe() への 2 番めと 3 番めのパラメータは，サブジェクトのリファレンスおよび非アクティブなコンシューマオブジェクトのオブジェクト ID です。サブジェクトリファレンスの解釈は，subscribe() への最初のパラメータとして SubjectScheme によって指定されます。これは，先に説明したとおりです。非アクティブなコンシューマオブジェクト ID は，送信するコンシューマオブジェクト（受信したイベント）を指定します。

コンシューマオブジェクトには，**非アクティブ**と**アクティブ**の 2 種類あります。すべてのプッシュコンシューマは非アクティブコンシューマで，プルコンシューマ（pull_and_dispatch() を使用する型付きコンシューマを除く）はすべてアクティブコンシューマです。

非アクティブコンシューマは，有効なオブジェクト ID を使ってサブスクライブする必要があります。コンシューマサーバントは，サブスクライブ中の PSA（POA など）内で，サーバントマネージャなどによってアクティブ化するか，アクティブ化できるようにする必要があります。一方，アクティブコンシューマでは，subscribe() を呼び出すために有効なオブジェクト ID は必要ありません。事実，PSA では，アクティブコンシューマをサブスクライブするために subscribe() が呼び出されると，実際のオブジェクト ID パラメータは無視されます。また，アクティブコンシューマは，アクティブ化したり，アクティブ化できるようにする必要はありません。

Subscribe() のサンプル

サンプル

このサンプルは，プッシュコンシューマとしてチャネルリファレンスを介して，型付きサービスに接続する方法を示しています。

C++

```
PortableServerExt::SubjectScheme scheme = {
  PortableServerExt::CHANNEL_ADDR,
  PortableServerExt::UNTYPED_SUBJECT,
  (const char*)"",
  PortableServerExt::PUSH_EVENT };
PortableServerExt::SubscribeDesc_var desc psa->subscribe(
  scheme, channel, observer_oid, CORBA::NameValuePairSeq());
```

Java

```
// プッシュコンシューマとしてチャネルリファレンスを介して型なしサービスに接続する Java コード。
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.UNTYPED_SUBJECT,
 "",
 SubjectDeliveryScheme.PUSH_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, observer_oid, null);
```

サンプル

このサンプルは，プルコンシューマとしてチャネルリファレンスを介して，型なしサービスに接続する方法を示しています。

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::UNTYPED_SUBJECT,
 (const char*)"",
 PortableServerExt::PULL_EVENT };
PortableServerExt::SubscribeDesc_var desc = psa->subscribe(
scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java

```
// プッシュコンシューマとしてチャネルリファレンスを介して型なしサービスに接続する Java コード。
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
```

```
 SubjectInterfaceScheme.UNTYPED_SUBJECT,
 "",
 SubjectDeliveryScheme.PULL_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, null, null);
```

サンプル

このサンプルは，プッシュコンシューマとしてチャネルリファレンスを介して，構造化サービスに接続する方法を示しています。

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::STRUCTURED_SUBJECT,
 (const char*)"",
 PortableServerExt::PUSH_EVENT };
PortableServerExt::SubscribeDesc_var desc psa->subscribe(
scheme, channel, observer_oid, CORBA::NameValuePairSeq());
```

Java

```
// プッシュコンシューマとしてチャネルリファレンスを介して構造化サービスに接続する Java コード。
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.STRUCTURED_SUBJECT,
 "",
 SubjectDeliveryScheme.PUSH_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, observer_oid, null);
```

サンプル

このサンプルは，プルコンシューマとしてチャネルリファレンスを介して，構造化サービスに接続する方法を示しています。

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::STRUCTURED_SUBJECT,
 (const char*)"",
 PortableServerExt::PULL_EVENT };
PortableServerExt::SubscribeDesc_var desc = psa->subscribe(
 scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java

```
// プルコンシューマとしてチャネルリファレンスを介して構造化サービスに接続する Java コード。
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.STRUCTURED_SUBJECT,
 "",
 SubjectDeliveryScheme.PULL_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, null, null);
```

サンプル

このサンプルは，プッシュコンシューマとしてチャネルリファレンスを介して，型付きサービスに接続する方法を示しています。

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::TYPED_SUBJECT,
 (const char*)"IDL:example.borland.com/TMN/TypedEvent:1.0",
 PortableServerExt::PUSH_EVENT };
PortableServerExt::SubscribeDesc_var desc psa->subscribe(
 scheme, channel, observer_oid, CORBA::NameValuePairSeq());
```

Java

```
// プッシュコンシューマとしてチャネルリファレンスを介して型付きサービスに接続する Java コード。
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.TYPED_SUBJECT,
 "IDL:example.borland.com/TMN/TypedEvent:1.0",
```

```
 SubjectDeliveryScheme.PUSH_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, observer_oid, null);
```

サンプル

このサンプルは，プルコンシューマとしてチャネルリファレンスを介して，型付きサービスに接続する方法を示しています。

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::TYPED_SUBJECT,
 (const char*)"IDL::example.borland.com/TMN/TypedEvent:1.0",
 PortableServerExt::PULL_EVENT };
PortableServerExt::SubscribeDesc_var desc = psa->subscribe(
 scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java

```
// プルコンシューマとしてチャネルリファレンスを介して型付きサービスに接続する Java
コード。
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.TYPED_SUBJECT,
 "IDL:example.borland.com/TMN/TypedEvent:1.0",
 SubjectDeliveryScheme.PULL_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, null, null);
```

サブスクライブデスクリプタと the_subject_addr()

```
Object the_subject_addr(in PublishSubscribeDesc the_desc);
```

subscribe() オペレーションが正しく行われると，サブスクライブデスクリプタが戻されます。サブスクライブデスクリプタには，unsubscribe()，suspend()，resume() など，その他の subscribe() オペレーションを実装するためのすべての情報／マッピングが保存されています。また，このデスクリプタは，永続的リポジトリに保存して，後で同じコンシューマプロセスセッションや，新しく再起動されたコンシューマプロセスセッションにロードすることができます。ただし，このデスクリプタの形式は，デスクリプタを作成した特定の ORB 内向けです。したがって，オブジェクトキーのように，サブスクライブデスクリプタは，デスクリプタを作成した ORB で使用する必要があります。

サブスクライブされたプッシュコンシューマのチャネルは，指定したオブザーバ ID でアクティブされたコンシューマサーバントに，イベントをアクティブにプッシュします。subscribe() オペレーションが正しく行われると，型なし／構造化／シーケンスプルコンシューマを持つアプリケーションは，PSA の the_subject_addr() から，サブスクライブデスクリプタとともにプルアドレス（プロキシプルサプライヤ）を取得することができます。サブスクライブデスクリプタは，PSA subscribe() メソッドからパラメータとして戻されます。

サンプル

このサンプルは，サブスクライブデスクリプタからプロキシ型なし／構造化／シーケンスプルサプライヤを取得する方法を示しています。

C++

```
CORBA::Object_var proxy_pull_supplier = psa->the_subject_addr(the_desc);
```

Java

```
org.omg.CORBA.Object proxy_pull_supplier = psa.the_subject_addr(the_desc);
```

指定したプロキシにこのリファレンスをナローイングすると，pull()/try_pull()/pull_structured_event() および try_pull_structured_event() を使ってサプライヤからイベントをプルすることができます。

型付きプルコンシューマについては，64 ページの「型付きプルのサポート」を参照してください。

サブジェクトのサブスクライブ解除

PSA unsubscribe() は，接続されたチャネルからコンシューマの接続を解除して，すべてのローカルリソースをクリーンアップします。これは，必要に応じて行われ，マルチキャストの場合は，オブザーバ ID マッピングへのサブジェクトキーが削除されます。コンシューマが，型なしチャネルや型付きチャネルに接続されている場合は，PSA はプロキシに対して disconnect_push/pull_supplier() を呼び出します。

コンシューマが構造化チャネルやシーケンスチャネルに接続されている場合は，PSA はそれぞれ，disconnect_structured_push/pull_supplier() と disconnect_sequence_push/pull_supplier() を呼び出します。

このサンプルコードは，サブジェクトのサブスクライブを解除する方法を示しています。

```
void unsubscribe(in SubscribeDesc the_subscribe_desc)
```

サブジェクトの公開

PSA モデルにおける**公開**は，（プッシュまたはプル）イベントメッセージを送信する通知／イベントチャネルに，サプライヤオブジェクトやサプライヤソースを結び付けるオペレーションとして定義されます。

これは，次に示す有効な publish/subscribe 構成をすべて含む非常に幅広い概念です。

- OMG 通知／イベントチャネルに接続する。
- マルチキャストチャネルに結び付ける。ただし，ネイティブ UDP マルチキャストである publish() は，ラッパー公開デスクリプタを作成し，返すだけです。
- IIOP 以外のメッセージ指向ミドルウェアに接続する。

PSA は，低レベルの転送メカニズムやチャネル／メッセージ形式の種類に関係なく，単一のプログラミングモデルを使用して，多くの publish/subscribe 構成をサポートします。このバージョンでは，PSA は，OMG 通知／イベントチャネル（全 4 種類のチャネル）への接続だけをサポートします。次に，特定のサブジェクトに対してオブザーバなどのコンシューマをサブスクライブする PSA オペレーションを示します。

```
PublishDesc publish(
  in SubjectScheme the_subject_scheme,
  in Subject the_subject,
  in PortableServer::ObjectId  the_provider_id,
  in CORBA::NameValuePairSeq  the_properties )
     raises( InvalidSubjectScheme,
       InvalidProperties,
       ChannelException );
```

COS Notification の上位で公開オペレーションを使用すると，このオペレーションは，サプライヤ管理およびプロキシコンシューマを取得し，接続を行うすべてのオペレーションを実行します。また，公開オペレーションを型付きサブジェクトとともに使用すると，PSA は，プロキシコンシューマで get_typed_consumer() を呼び出して，<I> リファレンスを取得することもできます。

SubjectScheme

SubjectScheme は，publish() への最初のパラメータで，次のように定義されます。

```
struct SubjectScheme {
 SubjectAddressScheme address_scheme;
 SubjectInterfaceScheme interface_scheme;
 SubjectInterfaceId     interface_id;
 SubjectDeliveryScheme delivery_scheme;
};
```

SubjectScheme パラメータでは，サブジェクトリファレンスで使用するアドレススキーム，インターフェーススキーム，インターフェースリポジトリ ID（型付きチャネルのみ），および配布スキームを指定できます。

address_scheme フィールドには，直接イベントをプッシュできるアドレスかどうか，サブスクライブだけが可能なアドレスかどうかなど，サブジェクトリファレンスを指定します。現在，VisiBroker では，このフィールドに 3 つの値を指定できます。それぞれの値は，subscribe() オペレーションへのサブジェクトリファレンスが，OMG Notification Consumer Admin，OMG Notification Channel（または型付きチャネル），または直接的なイベントプッシュアドレスであることを示しています。

次に，サポートされるサブスクライブの 3 つのアドレススキームを示します。

- **PUBLISH_ADMIN_ADDR** - publish() へのサブジェクトリファレンスは，OMG Notification Consumer Admin リファレンスです。PSA は，管理リファレンスで obtain_<...>_consumer() を呼び出して，管理でプロキシを割り当て，プロキシで connect_<...>_supplier() を呼び出します。プロキシに接続されたサプライヤリファレンスは，null（プッシュサプライヤ用），または provider_id パラメータを持つこの PSA から生成されたプルサプライヤリファレンスのどちらかになります。型付きチャネルの場合，get_typed_consumer() オペレーションと get_typed_supplier() インプリメンテーションは，PSA によって自動的に処理されます。
- **CHANNEL_ADDR** - publish() へのサブジェクトリファレンスは，OMG Notification Channel（または型付きチャネル）です。PSA は，チャネルで _get_default_supplier_admin() を呼び出すだけで，デフォルトのサプライヤ管理を取得できます。次に，このコンシューマ管理リファレンスを介して，接続として処理します。
- **SUBJECT_ADDR** - subscribe() へのサブジェクトリファレンスは，直接的なイベントプッシュアドレスです。たとえば，マルチキャスト IOR や型付き <I> インターフェースです。これは，ごく簡単な例です。PSA は，パブリッシャデスクリプタをラップして返します。

また，アプリケーションは，**SubjectInterfaceScheme** および **SubjectDeliveryScheme** を指定する必要もあります。

次に **SubjectInterfaceScheme** の有効な値を示します。

- **TYPED_SUBJECT** - サブジェクトは，マルチキャストまたは OMG 型付き通知チャネルを使用します。
- **UNTYPED_SUBJECT** - サブジェクトは，OMG 型なし通知チャネルを使用します。
- **STRUCTURED_SUBJECT** - サブジェクトは OMG 構造化通知チャネルを使用します。
- **SEQUENCE_SUBJECT** - サブジェクトは，OMG シーケンス通知チャネルを使用します。

次に **SubjectDeliveryScheme** の有効な値を示します。

- **PUSH_EVENT** - サブジェクトは，マルチキャストまたは OMG プッシュ通知モード（4 種類ある OMG イベントのいずれか）を使用します。
- **PULL_EVENT** - サブジェクトは，OMG プル通知モード（4 種類ある OMG イベントのいずれか）を使用します。

型付きチャネルに接続するには，<I> インターフェースのリポジトリ ID も指定する必要があります。このリポジトリは，get_typed_consumer() から戻された型付きプッシュリファレンスを**スタブにナローイングする場合に使用します**。これは，プッシュイベントおよびプルイベントのフィルタリングキーとしても使用されます。

Publish() へのサブジェクトリファレンス，プロバイダ ID，およびプロパティ

サブジェクトリファレンスの解釈は, publish() オペレーションへの最初のパラメータとして SubjectScheme によって指定されます。publish() への 2 番めと 3 番めのパラメータは，サブジェクトのリファレンスと，非アクティブなサプライヤ（サプライヤなど）オブジェクトのオブジェクト ID です。非アクティブなサプライヤオブジェクト ID は，イベントをプルして公開するために PSA で使用するサプライヤオブジェクトを指定します。

サプライヤオブジェクトには, 非アクティブとアクティブの 2 種類あります。すべてのプッシュサプライヤはアクティブで，すべてのプルサプライヤは非アクティブです。

非アクティブサプライヤは，有効なオブジェクト ID を使って公開する必要があります。公開サーバントは，公開中の PSA（POA など）内で，サーバントマネージャなどによってアクティブ化するか，アクティブ化できるようにする必要があります。一方，アクティブサプライヤでは，publish() を呼び出すために有効なオブジェクト ID は必要ありません。事実，PSA では，アクティブサプライヤを公開するために publish() が呼び出されると，実際のオブジェクト ID パラメータは無視されます。また，アクティブサプライヤは，アクティブ化したり，アクティブ化できるようにする必要はありません。アクティブサプライヤには，サーバントインプリメンテーションも必要ありません。

メモ アクティブサプライヤには，サーバントインプリメンテーションも必要ありません。

publish() のサンプル

サンプル

このサンプルは，名前空間 PortableServerExt を使用して，プッシュサプライヤとしてチャネルリファレンスを介し，型なしサービスに接続する方法を示しています。

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::UNTYPED_SUBJECT,
 PortableServerExt::(const char*)"",
 PortableServerExt::PUSH_EVENT };
PortableServerExt::PublishDesc_var desc psa->publish(
 Scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java

```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.UNTYPED_SUBJECT,
 "",
 SubjectDeliveryScheme.PUSH_EVENT);
Byte[] desc = psa.publish(scheme, channel, null, null);
```

スキームで指定されているように，特定のサブジェクトリファレンスは，実際には COS Notification Service チャネルリファレンスです。PSA は次のオペレーションを内部的に実行します。

- このチャネルからデフォルトのサプライヤ管理を取得する。
- 管理から型なしプロキシプッシュコンシューマを取得する。
- 戻されたサブスクライブデスクリプタ内のプロキシプッシュコンシューマのリファレンスをカプセル化する。

サンプル

このサンプルは，名前空間 PortableServerExt を使用して，プルサプライヤとしてチャネルリファレンスを介し，型なしサービスに接続する方法を示しています。

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::UNTYPED_SUBJECT,
 (const char*)"",
```

```
 PortableServerExt::PULL_EVENT };
PortableServerExt::PublishDesc_var desc = psa->publish(
 scheme, channel, provider_id, CORBA::NameValuePairSeq());
```

Java

```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.UNTYPED_SUBJECT,
 "",
 SubjectDeliveryScheme.PULL_EVENT);
PublishDesc desc = psa.publish(scheme, channel, provider_id, null);
```

スキームで指定されているように，特定のサブジェクトリファレンスは，実際には COS Notification Service チャネルリファレンスです。PSA は次のオペレーションを実行します。

- このチャネルからデフォルトのサプライヤ管理を取得する。
- 管理から型なしプロキシプルコンシューマを取得する。
- プロキシリファレンス上で，`get_typed_consumer()` を呼び出し，<I> インターフェースを取得する。
- このプルサプライヤリファレンスを使ってプロキシに接続する。
- 戻されたサブスクライブデスクリプタ内のプロキシプルコンシューマのリファレンスをカプセル化する。

サンプル

このサンプルは，名前空間 PortableServerExt を使用して，プッシュサプライヤとしてチャネルリファレンスを介し，構造化サービスに接続する方法を示しています。

C++

```
SubjectScheme scheme = {
 CHANNEL_ADDR, STRUCTURED_SUBJECT, (const char*)"", PUSH_EVENT };
PublishDesc_var desc = psa->publish(
scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java

```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.STRUCTURED_SUBJECT,
 "",
 SubjectDeliveryScheme.PUSH_EVENT);
PublishDesc desc = psa.publish(scheme, channel, null, null);
```

スキームで指定されているように，特定のサブジェクトリファレンスは，実際には COS Notification Service チャネルリファレンスです。PSA は次のオペレーションを実行します。

- このチャネルからデフォルトのサプライヤ管理を取得する。
- 管理から構造化プロキシプッシュコンシューマを取得する。
- 戻されたサブスクライブデスクリプタ内のプロキシプッシュコンシューマのリファレンスをカプセル化する。

サンプル

このサンプルは，C++ コードで，名前空間 PortableServerExt を使用して，プッシュサプライヤとしてチャネルリファレンスを介し，構造化サービスに接続する方法を示しています。

C++

```
PortableServerExt::SubjectScheme scheme = {
  PortableServerExt::CHANNEL_ADDR,
  PortableServerExt::STRUCTURED_SUBJECT,
 (const char*)"",
  PortableServerExt::PULL_EVENT };
PortableServerExt::PublishDesc_var desc = psa->publish(
 scheme, channel, provider_id, CORBA::NameValuePairSeq());
```

Java
```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.STRUCTURED_SUBJECT,
 "",
 SubjectDeliveryScheme.PULL_EVENT);
Byte[] desc = psa.publish(scheme, channel, provider_id, null);
```

スキームで指定されているように，特定のサブジェクトリファレンスは，実際には COS Notification Service チャネルリファレンスです。PSA は次のオペレーションを実行します。

- このチャネルからデフォルトのサプライヤ管理を取得する。
- 管理から構造化プロキシプルコンシューマを取得する。
- オブジェクト ID として provider_id パラメータを使用し，PSA からプルサプライヤオブジェクトリファレンスを作成する。
- このプルサプライヤリファレンスを使ってプロキシに接続する。
- 戻されたサブスクライブデスクリプタ内のプロキシプルコンシューマのリファレンスをカプセル化する。

サンプル

このサンプルは，名前空間 PortableServerExt を使用して，プッシュサプライヤとしてチャネルリファレンスを介し，型付きサービスに接続する方法を示しています。

C++
```
PortableServerExt::SubjectScheme scheme = {
  PortableServerExt::CHANNEL_ADDR,
  PortableServerExt::TYPED_SUBJECT,
  (const char*)"IDL:example.borland.com/TMN/TypedEvent:1.0",
  PortableServerExt::PUSH_EVENT };
PortableServerExt::PublishDesc_var desc psa->publish(
  scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java
```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.TYPED_SUBJECT,
 "",
 SubjectDeliveryScheme.PUSH_EVENT);
PublishDesc desc = psa.publish(scheme, channel, null, null);
```

スキームで指定されているように，特定のサブジェクトリファレンスは，実際には COS Notification Service チャネルリファレンスです。PSA は次のオペレーションを実行します。

- このチャネルからデフォルトのサプライヤ管理を取得する。
- 管理から型付きプロキシプルコンシューマを取得する。
- プロキシリファレンスで get_typed_consumer() を呼び出して，**インターフェースリファレンス**を取得する。
- このプルサプライヤリファレンスを使ってプロキシに接続する。
- 戻されたサブスクライブデスクリプタ内のプロキシプッシュコンシューマのリファレンスおよび <I> インターフェースリファレンスをカプセル化する。

サンプル

このサンプルは，名前空間 PortableServerExt を使用して，プルサプライヤとしてチャネルリファレンスを介し，型付きサービスに接続する方法を示しています。

C++
```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::TYPED_SUBJECT,
 (const char*)"IDL::example.borland.com/TMN/TypedEvent:1.0",
 PortableServerExt::PULL_EVENT };
```

```
PortableServerExt::PublishDesc_var desc = psa->publish(
       scheme, channel, provider_id, CORBA::NameValuePairSeq());
```

Java

```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.TYPED_SUBJECT,
 "",
 SubjectDeliveryScheme.PULL_EVENT);
PublishDesc desc = psa.publish(scheme, channel, provider_id, null);
```

スキームで指定されているように，特定のサブジェクトリファレンスは，実際には COS Notification Service チャネルリファレンスです。PSA は次のオペレーションを実行します。

- このチャネルからデフォルトのサプライヤ管理を取得する。
- 管理から型付きプロキシプルコンシューマを取得する。
- オブジェクト ID として provider_id パラメータを使用し，この PSA から型付きプルサプライヤオブジェクトリファレンスを作成する。
- 内部のプロキシサプライヤインプリメンテーションを作成して，get_typed_supplier() メソッドからリファレンスを戻す。
- このプロキシインプリメンテーションのリファレンスを作成する。
- このプロキシインプリメンテーションのリファレンスを使ってプロキシコンシューマに接続する。
- 戻されたサブスクライブデスクリプタ内のプロキシプルコンシューマのリファレンスをカプセル化する。

公開デスクリプタと the_subject_addr()

```
Object the_subject_addr(in PublishSubscribeDesc the_desc);
```

publish() オペレーションを正しく行うと，公開デスクリプタが戻されます。公開デスクリプタには，unpublish()，suspend()，resume() など，その他の publish() オペレーションを実装するための情報／マッピングが保存されています。このデスクリプタは，永続的リポジトリに保存して，後で同じサプライヤプロセスセッションや，新しく再起動されたサプライヤセッションに再ロードすることができます。ただし，このデスクリプタの形式は，デスクリプタを作成した特定の ORB 内用です。したがって，オブジェクトキーのように，サブスクライブデスクリプタは，デスクリプタを作成した ORB で使用する必要があります。

公開された非アクティブサプライヤ（プルなど）のチャネルは，指定したプロバイダ ID でアクティブ化したサプライヤサーバントからイベントをプルします。

publish() オペレーションが正しく行われると，アクティブ（プッシュ）サプライヤを持つアプリケーションは，公開デスクリプタを使用する PSA の the_subject_addr() から，プッシュアドレス（プロキシプッシュコンシューマまたは <I> インターフェースリファレンスと入力されたチャネル）を取得することができます。公開デスクリプタは，PSA publish() メソッドからパラメータとして戻されます。

サンプル

このサンプルは，サブスクライブデスクリプタからプロキシ型なし／構造化／シーケンスプルサプライヤを取得する方法を示しています。

C++
```
CORBA::Object_var proxy_pull_supplier = psa->the_subject_addr(the_desc);
```

Java
```
org.omg.CORBA.Object proxy_pull_supplier = psa.the_subject_addr(the_desc);
```

指定したプロキシや <I> インターフェーススタブにこのリファレンスをナローイングすると，アプリケーションは，接続されたチャネルにイベントをプッシュできます。

メモ 型付きプルサプライヤについては，64 ページの「型付きプルのサポート」を参照してください。

サブジェクトの公開解除

```
void unpublish(in PublishDesc the_publish_desc)
        raises( InvalidPublishDesc,
                ChannelException );
```

PSA unpublish() は，接続されたチャネルからサプライヤの接続を解除して，すべてのローカルリソースをクリーンアップします。サプライヤが型なしチャネルや型付きチャネルに接続されている場合は，PSA はプロキシに対して disconnect_push/pull_consumer() を呼び出します。コンシューマが構造化チャネルやシーケンスチャネルに接続されている場合は，PSA はそれぞれ，disconnect_structured_push/pull_consumer() と disconnect_sequence_push/pull_consumer() を呼び出します。

型付きプルのサポート

Notification Service の大きな問題の 1 つは，型付きプルの処理における問題です。OMG で定義されたプログラミングモデルにより，使用するのが困難です。理解に苦しむ「Pull<I>」インターフェースを使用するにはプルコンシューマ，「Pull<I>」サーバントを実装するにはプルサプライヤが必要です。型付きプッシュ <I> コンシューマ／サプライヤから型付き「Pull<I>」コンシューマ／サプライヤに変更するには，アプリケーションの設計を大幅に改良し，コードを変更する必要があります。また，「Pull<I>」インターフェースは，オペレーション専用です。たとえば，型付きイベントをチャネルからプルするには，イベントに関連付けられたオペレーションでプルコンシューマを選択する必要があります。これは，型付きプッシュコンシューマまたは構造化プルコンシューマのどちらにも対

応していません。型付きプッシュコンシューマでは，コンシューマをプッシュしても，次に受信されるイベントに関連付ける必要があるオペレーションを指定することはできません。構造化イベントの場合，構造化プルコンシューマは，次に戻されるイベントの type_name（型付きイベントのオペレーションと同等）では選択できません。

PSA は上述の問題をすべて解決します。次に，PSA 固有の利点を示します。

- 型付きプッシュコンシューマのように，型付きプルコンシューマは，理解に苦しむ "Pull<I>" インターフェースのかわりに，元の <I> インターフェースを実装します。これにより，PSA のモデルを直感的かつ容易に使用できるようになり，既存のツール（通常の IDL プリコンパイラなど）を使用して，タイプセーフのコードを生成できます。PSA を使って開発されたプルコンシューマアプリケーションは，"Pull<I>" インターフェースを使用した開発より，簡単に理解できます。
- 型付きプッシュサプライヤのように，型付きプルサプライヤは，型固有の "Pull<I>" サーバントを実装するかわりに，元の <I> インターフェースを使用します。
- 戻されたイベントのオペレーションでは，型付きプルコンシューマを選択できません。これは，型付きプッシュコンシューマおよび構造化プルコンシューマと一致しています。
- PSA は，アクティブ型付きプルコンシューマおよび非アクティブ型付きプルコンシューマをサポートするので，アプリケーションの異なる要件を満たすことができます。

次に，PSA がサポートする 3 種類の型付きプルインプリメンテーションを示します。

- 非アクティブ型付きプルコンシューマ
- アクティブ型付きプルコンシューマ
- 型付きプルサプライヤ

非アクティブ型付きプルコンシューマ

コードレベルの非アクティブ型付きプルコンシューマは，型付きプッシュコンシューマに似ています。実際，型付きプッシュコンシューマアプリケーションは，ほとんどコードを変更せずに，非アクティブ型付きプルコンシューマアプリケーションに変更することができます。非アクティブ型付きコンシューマを作成するには，コンシューマオブジェクトが POA で利用できる必要があり，サブジェクトに関連付けられたオブジェクト ID とともに，コンシューマオブジェクトをサブジェクトにサブスクライブする必要があります。次に，非アクティブ型付きプルコンシューマと型付きプッシュコンシューマの違いについて説明します。

- 通常の型付きプッシュコンシューマ：型付きイベントは，チャネルからコンシューマプロセスに非同期にプッシュされ，プッシュコンシューマオブジェクトにディスパッチされます。
- 非アクティブ型付きプルコンシューマ:型付きイベントは，コンシューマアプリケーションによってチャネルからコンシューマサーバーに同期的にプルバックされ，アクティブ型付きプッシュコンシューマであるかのように，PSA によって，非アクティブ型付きコンシューマオブジェクトにディスパッチされます。

したがって，非アクティブ型付きプルコンシューマをサブスクライブするには，有効なオブジェクト ID が PSA subscribe() オペレーションに必要です。subscribe() の次に，アプリケーションは PSA の pull_and_dispatch() メソッドを使用して，チャネルから型付きイベントをプルして，非アクティブコンシューマにディスパッチします。非アクティブ型付きプルコンシューマは，コンシューマアプリケーションから受信されるイベントを制御するとともに，非アクティブコンシューマを使用するアプリケーション向けに設計されています。

サンプル

このサンプルは，非アクティブ型付きプルコンシューマのものです。

C++
```
// (examples/vbroker/notify/psa_cpp/typedPullConsumer1.C)
// アクティブなビジターを実装します。
# include "TMNEvents_s.hh"
class TMNTypedEventVisitor : public POA_TMN::TypedEvent
{
 ...
   public:
 void attributeValueChange(...) { ... }
 ...
 void qosAlarm(...) { ... }
};
int main(argc, argv)
{
   ...
   // 型付きプルコンシューマとしてチャネルにサブスクライブします。
   PortableServerExt::SubjectScheme scheme = {
         PortableServerExt::CHANNEL_ADDR,
            PortableServerExt::TYPED_SUBJECT,
                (const char*)"IDL::example.borland.com/TMN/TypedEvent:1.0",
                  PortableServerExt::PULL_EVENT };
   PortableServerExt::SubscribeDesc_var desc = psa->subscribe(
      scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
   // ビジターインスタンスを作成します。
   TMNTypedEventVisitor visitor;
    // ブロックモードを使用して，最大 100 個のイベントをプルおよびビジットします。
    psa->pull_and_visit(desc, 100, (CORBA::Boolean)1, &visitor);
    ...
 }
```

プッシュされたコンシューマアプリケーションとの唯一の違いは, 型付きイベントを取得する方法です。ORB run() でブロックして，チャネルがイベントを非同期に送信するのを待機するのではなく, PSA を使用した, アプリケーションによる明示的なプル（pull_and_dispatch() を使用）が非アクティブ型付きコンシューマに必要です。

非アクティブ型付きプルコンシューマのロジックおよび手順は，次のようにまとめることができます。

- POA スケルトンから派生したコンシューマサーバントインプリメンテーションを記述する。
- POA でサーバントをアクティブ化して，そのオブジェクト ID を取得する。
- SubjectDeliveryScheme に PULL_EVENT を指定して，チャネルにサブスクライブする。
- サブスクライブデスクリプタをパラメータとして使用して，サブスクライブ PSA で pull_and_dispatch() を呼び出す。
- プルバックされたイベントが，指定したコンシューマサーバントに非同期にディスパッチされる。

アクティブ型付きプルコンシューマ

アクティブ型付きプルコンシューマのコンシューマサーバントは，POA には登録されず，POA をアクティブ化する必要もありません。返信された型付きイベントは，POA_<I> サーバントスケルトンから派生したビジターインプリメンテーションによって直接ビジットされます（visitor パターンを考慮）。visitor インプリメンテーションは，直接 pull_and_visit() の各コールで指定されるので，POA に関連付けたり，POA で登録する必要はありません。アクティブ型付きプルコンシューマは，従来の型付きプルに似ていますが，"Pull<I>" スタブではなく POA_<I> を実装して，イベントに逆にビジットする点で異なります。

サンプル

このサンプルは，アクティブ型付きプルコンシューマのものです。

C++

```
// (examples/vbroker/notify/psa_cpp/typedPullConsumer1.C)
// アクティブなビジターを実装します。
# include "TMNEvents_s.hh"
class TMNTypedEventVisitor : public POA_TMN::TypedEvent
{
 ...
   public:
 void attributeValueChange(...) { ... }
 ...
 void qosAlarm(...) { ... }
};
int main(argc, argv)
{
   ...
   // 型付きプルコンシューマとしてチャネルにサブスクライブします。
   PortableServerExt::SubjectScheme scheme = {
         PortableServerExt::CHANNEL_ADDR,
         PortableServerExt::TYPED_SUBJECT,
         (const char*)"IDL::example.borland.com/TMN/TypedEvent:1.0",
         PortableServerExt::PULL_EVENT };
   PortableServerExt::SubscribeDesc_var desc = psa->subscribe(
      scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
   // ビジターインスタンスを作成します。
   TMNTypedEventVisitor visitor;
    // ブロックモードを使用して，最大 100 個のイベントをプルおよびビジットします。
    psa->pull_and_visit(desc, 100, (CORBA::Boolean)1, &visitor);
    ...
 }
```

Java

```
// (examples/vbroker/notify/psa_java/TypedPullConsumer1.java)
import com.inprise.vbroker.PortableServerExt.*;
// アクティブなビジターを実装します。
class TMNTypedEventVisitor extends TMN.TypedEventPOA {
{
   public void attributeValueChange(...) { ... }
       ...
   public void qosAlarm(...) { ... }
};
public class TypedPullConsumer1 {
   public static void main(String[] args) {
      ...
      // 型付きプルコンシューマとしてチャネルにサブスクライブします。
      SubjectScheme scheme = new SubjectScheme(
         SubjectAddressScheme.CHANNEL_ADDR,
         SubjectInterfaceScheme.TYPED_SUBJECT,
         "IDL::example.borland.com/TMN/TypedEvent:1.0",
         SubjectDeliveryScheme.PULL_EVENT };
      SubscribeDesc desc = psa.subscribe(scheme, channel, null, null);
      // ビジターインスタンスを作成します。
       TMNTypedEventVisitor visitor = new TMNTypedEventVisitor();
       // ブロックモードを使用して，最大 100 個のイベントをプルおよびビジットします。
       psa.pull_and_visit(desc, 100, true, visitor);
   }
      }
```

アクティブ型付きプルコンシューマのロジックおよび手順は，次のようにまとめることができます。

- POA スケルトンから派生したビジター（サーバント）インプリメンテーションを記述する。
- SubjectDeliveryScheme に PULL_EVENT を指定して，チャネルにサブスクライブする。
- サブスクライブデスクリプタおよびパラメータとしてビジターインスタンスを使用して，サブスクライブ PSA で pull_and_visit() を呼び出す。

- プルバックされたイベントが，指定したビジターで同期的に呼び出される。

型付きプルサプライヤ

PSA は，「ピギーバックリフレクティブコールバック」技術を使用して，型付きプルサプライヤをサポートしています。リフレクティブコールバックを使用すると，プルサプライヤをプルしても，最初にプッシュモード用に定義した <I> インターフェース内でイベントを発行できます。

ピギーバックを使用しないシンプルなリフレクティブコールバックは，次のように動作します。

- プルサプライヤは，定義済み型付きの特定されないコールバックハンドラを実装，インスタンス化，およびアクティブ化します。たとえば，TypedCallback::PullEvent ハンドラは，pull_typed_event() などの単一のオペレーションだけを持ちます。オペレーションには，入力パラメータとしてイベント受信者オブジェクトリファレンスがあります。
- プルサプライヤを公開すると，このコールバックハンドラのリファレンスがチャネルに接続されます。

次に，チャネルがサプライヤをプルした場合の特性を示します。

- チャネルはイベント受信オブジェクトを準備して，そのリファレンスを取得します。
- チャネルは，イベント受信者リファレンスを使用して，プルサプライヤの TypedCallback::PullEvent ハンドラの pull_typed_event() オペレーションをコールバックします。
- プルサプライヤは，イベント受信リファレンスを <I> インターフェースにナローイングします。プルサプライヤは，<I> インターフェースで定義されたオペレーションを呼び出して，特定のイベントを型付きチャネルに送信します。
- チャネルは，イベント受信者からイベントを取得して，コンシューマに送信します。

シンプルなリフレクティブコールバックの利点は，プルサプライヤ側 ORB に特別なサポートが必要ない点です。欠点は，プルオペレーションごとにリモート往復が 2 つ必要であるという点です。最初の往復には，チャネルからサプライヤへのコールバックが必要です。2 番めの往復には，サプライヤからイベント受信者へのコールバックが必要です。

PSA と VisiNotify では，ピギーバックリフレクティブコールバックをサポートおよび実装しています。ピギーバックリフレクティブコールバックは，次のようなメカニズムを持つ，シンプルなリフレクティブコールバックのバリエーションです。

- PSA は，内部エージェントオブジェクト（別の内部シングルトン POA に存在）を作成します。このオブジェクトは，入力パラメータを使用しないで pull_typed_event() メソッドをサポートします。
- プルサプライヤは，定義済みアプリケーションインターフェースに依存しない TypedCallback::PullEvent ハンドラを実装，インスタンス化，およびアクティブ化します。
- アプリケーションがプルサプライヤを公開すると，ORB 内部エージェントにポイントされており，コールバックハンドラリファレンスをカプセル化するコールバックリファレンスは，チャネルに接続されます。アプリケーションハンドラは，実際はチャネルに接続されていません。
- エージェントの pull_typed_event() メソッドへのチャネルコールバックには，入力パラメータがありません。
- エージェントは，オブジェクト ID からの実際のハンドラのリファレンスを解決して，ローカルイベント受信者を作成します。
- エージェントは，このローカルイベント受信者リファレンスを入力パラメータとして使用して，ハンドラの pull_typed_event() をローカルで呼び出します。

- アプリケーションは，このローカルイベント受信者リファレンスを <I> にナローイングして，型付きイベントをローカルで発行します。
- エージェントは，ローカルイベント受信者からのイベントをアンパックして，チャネルの pull_typed_event() コールからの応答として送り返します。

ピギーバックリフレクティブコールバックは，アプリケーションに対して透過的です。たとえば，アプリケーションは，シンプルなリフレクティブコールバックにもピギーバックリフレクティブコールバックにも依存しません。ピギーバックリフレクティブコールバックでは，プルごとに必要な往復は 1 回だけですが，プルサプライヤ側の ORB のサポートが必要です。VisiBroker PSA では，ピギーバックリフレクティブコールバックがサポートされています。VisiNotify は，効果的な理由から，ピギーバックリフレクティブコールバックだけを使用します。

サンプル

このサンプルは，型付きプルコンシューマのものです。

C++

```
// (examples/vbroker/notify/psa_cpp/typedPullSupplier.C)
// ピギーバックダブルコールバックを使って TypedCallback::PullEvent
// ハンドラを実装します。このハンドラは，リモートプロキシプルコンシューマ
// ではなく，ローカルな PSA によってコールバックされます。したがって，
// イベント受信者もローカルオブジェクトです。
# include <TypedCallback_s.hh>
# include "TMNEvents_c.hh"
class PullEventHandlerImpl : public POA_TypedCallback::PullEvent,
                      public virtual PortableServer::RefCountServantBase
 {
     public:
  // 型付きプル
  void pull_typed_event(
      CORBA::Object_ptr event_receiver,
      CORBA::Boolean  block)
  {
   // 型付きスタブにナローイングします。
   TMN::TypedEvent_ptr stub
= TMN::TypedEvent::_narrow(event_receiver);
   // コールバックを反映して，
// attributeValueChange イベントを発行します。
   stub->attributeValueChange(...);
  }
 };
 ...
// サプライヤハンドラサーバントを作成して，PSA でアクティブ化します。
PullEventHandler* handler = new PullEventHandlerImpl;
psa->activate_object_with_id(handler_id, handler);
// 実際の <I> インターフェースリポジトリ ID ではなく，handler_id を
// 使用して，型付きプルサプライヤとしてチャネルにパブリッシュします。
PortableServerExt::SubjectScheme scheme = {
   PortableServerExt::CHANNEL_ADDR,
   PortableServerExt::TYPED_SUBJECT,
   (const char*)"IDL::example.borland.com/TMN/TypedEvent:1.0",
   PortableServerExt::PULL_EVENT };
PortableServerExt::SubscribeDesc_var desc = psa->publish(
   scheme, channel, handler_id, CORBA::NameValuePairSeq());
 // PSA をアクティブ化し，プルを待ちます。
 psa->the_POAManager()->activate();
 orb->run();
```

Java

```
// (examples/vbroker/notify/psa_java/TypedPullSupplier.java)
 import com.inprise.vbroker.PortableServerExt.*;
// ピギーバックダブルコールバックを使って TypedCallback::PullEvent
// ハンドラを実装します。このハンドラは，リモートプロキシプルコンシューマ
// ではなく，ローカルな PSA によってコールバックされます。したがって，
// イベント受信者もローカルオブジェクトです。
class PullEventHandler
```

```
        extends com.borland.vbroker.TypedCallback.PullEventPOA {
      ...
    public void pull_typed_event(
            org.omg.CORBA.Object event_receiver,
            Boolean   block) {
      // 型付きスタブにナローイングします。
      TMN.TypedEvent stub = TMN.TypedEventHelper.narrow(event_receiver);
      // コールバックを反映して，attributeValueChange イベントを発行します。
      stub.attributeValueChange(...);
    }
  }
  public class TypedPullSupplier {
    ...
    public static void main(String[] args) {
       ...
      // サプライヤハンドラサーバントを作成して，PSA でアクティブ化します。
      PullEventHandler handler = new PullEventHandler();
      psa.activate_object_with_id(handler_id, handler);
      // 実際の <I> インターフェースリポジトリ ID ではなく，handler_id を
      // 使用して，型付きプルサプライヤとしてチャネルにパブリッシュします。
      SubjectScheme scheme = new SubjectScheme(
         SubjectAddressScheme.CHANNEL_ADDR,
         SubjectInterfaceScheme.TYPED_SUBJECT,
         "IDL::example.borland.com/TMN/TypedEvent:1.0",
         SubjectDeliveryScheme.PULL_EVENT);
      SubscribeDesc desc = psa.publish(scheme, channel, handler_id, null);
       // PSA をアクティブ化し，プルを待ちます。
      psa.the_POAManager()Activate();
       orb->run();
    }
  }
```

型付きプルコンシューマのロジックおよび手順は，次のようにまとめることができます。

- POA スケルトンから派生した TypedCallback::PullEvent サプライヤサーバントインプリメンテーションを記述する。リフレクティブコールバックを使用するこのサーバントの pull_typed_event() オペレーションは, 元の IDL インターフェーススタブを使って型付きイベントを生成する。
- POA で PullEvent サーバントをアクティブ化して, そのオブジェクト ID を取得する。
- SubjectDeliveryScheme に PULL_EVENT を指定し, publish() パラメータとしてオブジェクト ID　を使用して，このコールバックをチャネルに公開する。
- POA をアクティブ化して，プルリクエストを処理する。

追加項目とまとめ

ここでは，PSA に関連するその他の情報について説明します。

ChannelException

the_subject_addr() and the_proxy_addr() の例外など，PSA オペレーションのほとんどは，PortableServerExt::ChannelException を生成します。この例外には文字列メンバーがあります。これは，低レベルの CORBA ユーザー例外のリポジトリです。たとえば，特定のプッシュコンシューマサブスクライブデスクリプタをパラメータとして使用しながら，suspend() を 2 回呼び出すと，ChannelException を取得できます。このとき，repository_id メンバーは「IDL:omg.org/CosNotifyChannelAdmin/ConnectionAlreadyInactive」になります。

Notification Service 例外を生成するために，PSA オペレーションを宣言しないのは，PSA フレームワークを汎用的にするためです。PSA の現在のインプリメンテーションは，

OMG Notification Service または Typed Notification Service とともに動作していますが，このインプリメンテーションは単純です。マルチキャストなど，その他の publish/subscribe インフラストラクチャに対応するようにサポートを拡張します。

PSA で Notification Service QoS を設定する

PSA アプリケーション内で接続されたプロキシに QoS ポリシーを設定するには，subscribe() および publish() のプロパティパラメータを使用するという方法があります。この方法は，VisiBroker 5.1.x では実装されていません。

QoS ポリシーを設定する別の方法としては，プロキシリファレンスを直接取得する方法があります。_proxy_addr() メソッドを使ってサブスクライブ／公開オペレーションを行ったら，プロキシリファレンスで set_qos() を使ってポリシーを変更します。

PSA のまとめ

次の一覧は，PSA の概念と機能に関するまとめです。

- PSA は，publish/subscribe システムに，直感的な高レベルのオブジェクトの抽象化を提供して，接続，管理，プロキシなど，低レベルのオブジェクトからアプリケーションを保護します。
- PSA は，一流のサブジェクトとして publish/subscribe をサポートして，POA モデルに似た高レベルのプログラム的なモデルを提供します。PSA を使用した CORBA publish/subscribe アプリケーションの開発は，POA ベースのクライアント／サーバーアプリケーションの開発に似ています。
- PSA は，直交するオブジェクトを分離するので，アプリケーションは，オブジェクトインプリメンテーションやロジックインプリメンテーションを別々に変更できます。たとえば，型付きコンシューマをプッシュからプルに変更する場合，サブスクライブのフラグを変更するだけで，コードを受信するメッセージを変更する必要はありません。PSA を使用しないでこのような変更を行うには，大きな修正が伴います。
- PSA を使用すれば，アプリケーションは，面倒でエラーしやすい理解に苦しむ "Pull<I>" ではなく，初めは型付きイベントプッシュ用に定義されたものとまったく同じ IDL <I> インターフェースを使用して，型付きイベントプルを行うことができます。
- PSA ／ VisiNotify における型付きイベントプルモデルは，構造化プルだけでなく，ほかのイベントプルモデルにも対照的です。
- PSA は，自動的に get_typed_consumer()/get_typed_supplier() および <I> インターフェースを処理して，マッピングを転送します。これは，型付きイベント／通知サービスを使用する場合のアプリケーションコードを大幅に簡略化します。型付き通知アプリケーションは，<I> インターフェースオブザーバを実装およびインストールするだけです。
- PSA は，高レベルのプログラミングモデルですが，OMG Notification Service で定義されている，QoS の照会や修正など，低レベルの機能にもアプリケーションはアクセスできます。
- 高レベルの抽象化により，PSA は，下にあるメッセージミドルウェアが OMG Notification Service であるとはみなしません。同じ PSA プログラミングモデルで，さまざまなマルチキャスト転送や OMG 以外のメッセージミドルウェアを透過的にサポートします。

第 5 章

Quality of Service とフィルタの設定

ここでは，イベントタイプを使用し，Filters プロパティおよび QoS プロパティを設定して，通知チャネルを設定する方法について説明します。

Quality of Service（QoS）のプロパティ

QoS で設定されているポリシーにより，アプリケーションは，実行時にチャネルのサービスパラメータを動的に調整することができます。VisiNotify は，独自の QoS ポリシーを指定するだけでなく，OMG 固有の QoS のサブセットもサポートします。次に，VisiNotify がサポートする QoS のプロパティを示します。

Priority

Priority QoS の設定と取得は，OMG 標準にしたがってサポートされています。優先順位は short 値で表されます。最も低い優先順位は -32,767，最も高い優先順位は 32,767 です。すべてのイベントのデフォルトの優先順位は 0 です。優先順位はメッセージレベル，プロキシレベル，管理レベル，およびチャネルレベルで設定できます。このプロパティを ConsumerAdmin またはプロキシサプライヤごとに設定しても意味がありません。

EventReliability

EventReliability QoS は，OMG 標準にしたがってサポートされています。パフォーマンス的な理由から，各プロキシサプライヤは，関連するコンシューマにどのイベントを送信したかを永続的に記憶しているわけではありません。したがって，イベントチャネルがクラッシュしたり，再起動した場合は，イベントが何度もコンシューマに送信されます。

VBPersistentDbType

このプロパティは，チャネルがイベントを保存する際に使用する永続的ストレージの種類を指定します。VisiNotify は，このプロパティの値にしたがって，永続的なイベントをメモリマップファイルまたはフラットファイルに保存します。(CORBA::Short)1 の値を指定すると，メモリマップ永続化になります。(CORBA::Short)2 の値を指定すると，フラットファイル永続化になります。

デフォルトは，メモリマップ永続化です。

VBPersistentCommitSyncPolicy

VBPersistentCommitSyncPolicy プロパティでは，チャネルが，永続的ストレージにイベントを適切にコミットした後にだけ，サプライヤを確認するかどうかを指定します。

次の定数値を設定できます。

- **True** - チャネルは，永続的ストレージにイベントを適切にコミットした後にだけ，サプライヤを確認します。
- **False（デフォルト）** - チャネルは，永続的ストレージにイベントをコミットする直前に，サプライヤ（push() コールからの戻りなど）を確認し，後で遅延コミットを実行します。

VBPersistentStorageOverflowBlockTimeout

新しいイベントがチャネル内に到着し，永続化する必要がある場合があります。ただし，永続的ストレージがすでにいっぱいの可能性もあります。この問題を防ぐには，永続的ストレージ内のスペースが空くまでサプライヤをブロックします。

VBPersistentStorageOverflowBlockTimeout プロパティでは，永続的ストレージのスペースが空くまで，サプライヤをブロックして待機させる時間を指定します。指定した時間が過ぎると，チャネルは新しいイベントを保持するために，1 つ以上のイベントを BestEffort にダウングレードしようとします
（74 ページの「VBPersistentStorageOverflowBlockTimeout」を参照）。

このプロパティのデフォルト値は 0 です。つまり，チャネルはブロックされませんが，かわりに，VBPersistentOverflowDowngradePolicy にしたがって，キューに入っているイベントをただちにダウングレードしようとします。

VBPersistentOverflowDowngradePolicy

VBPersistentOverflowDowngradePolicy プロパティは，VBPersistentStorageOverflowBlockTimeout で設定した時間を過ぎても永続的ストレージのスペースが空かない場合に，チャネルが既存のイベントをダウングレードして，新しい（永続的な）イベント用のスペースを作る方法を制御します。イベントがダウングレードされるということは，つまりメッセージ／チャネルの設定に関係なく，イベントの EventReliability が自動的に BestEffort に設定されるということです。

次の定数値を設定できます。

- **AnyOrder（デフォルト）** - Lifo が使用されます。
- **FifoOrder** - キューに入っているイベントは，先に受信されたものから先にダウングレードされます。
- **LifoOrder** - 新しいイベントがダウングレードされます。

メモ　イベント自身が永続的ストレージに収まらない場合は，ただちにダウングレードされます。

ConnectionReliability

ConnectionReliability は，OMG 標準にしたがってサポートされます。

ConnectionReliability が（適切なチャネル／管理／プロキシで）Persistent に設定されている場合は，VisiNotify は次を復元しようとします。

1 元のポリシー，ID，（チャネルの）IOR を持つすべての永続的なチャネルおよび含まれるすべてのイベント

2 元のポリシー，ID，および IOR を持つすべての永続的な管理

3 元のポリシー，ID，IOR，および結び付けられたサプライヤ／コンシューマを持つすべての永続的なプロキシ

プロキシの ConnectionReliability が through set_qos() を介して明示的に指定されていない場合は，アクティブなプロキシにはデフォルト値が使用されます。たとえば，プロキシのプル型コンシューマとプッシュ型のサプライヤが永続的であると，非アクティブプロキシのそれは，vbroker.notify.channel.passiveProxyPersistenceMask に記述されます。

MaxEventsPerConsumer

MaxEventsPerConsumer は，OMG 標準にしたがってサポートされます。

DiscardPolicy

永続的ストレージ管理など，インプリメンテーションを実現するために，OMG の AnyOrder，FifoOrder，および LifoOrder だけがサポートされます。

OrderPolicy

この QoS プロパティは，特定のプロキシが別のプロキシまたはコンシューマに配信するためにバッファリングしたイベントを並べ替える際に使用するポリシーを設定します。AnyOrder，FifoOrder（デフォルト），および PriorityOrder を使用できます。

メモ このプロパティをメッセージごとに設定しても意味がありません。

VBQueueLowWaterMark

プロキシサプライヤのキューに入っている保留中のイベント数が，VBQueueHighWaterMark レベルを超えた後（保留中のイベント数がこの値以下に下がったとき），このプロキシは必要に応じて，チャネルへのイベントのプッシュ／プルをアンブロックしたり，スピードアップするようにチャネルに通知します。詳細については，「フロー制御」を参照してください。

VBQueueLowWaterMark のデフォルト値は 32 です。

VBQueueHighWaterMark

プロキシサプライヤのキューに入っている保留中のイベント数が設定値を超えると，このプロキシは必要に応じて，チャネルへのイベントのプッシュ率およびプル率をブロックしたり，スローダウンするようにチャネルに通知します。詳細については，「フロー制御」を参照してください。

VBQueueHighWaterMark のデフォルト値は，VisiNotify によって算出され，チャネルキューのサイズ（チャネルの管理プロパティ）のユーザー定義の設定と，VBQueueLowWaterMark の設定によって異なります。

VBProxyPushSupplierThreadModel

各プロキシプッシュサプライヤには，キューにあるイベントを接続先のプッシュコンシューマにプッシュするためのスレッドが必要です。このプロパティは，プロキシがイベントをプッシュするために専用のスレッドを使用するか，スレッドプールを使用するかを指定します。

有効な値は“dedicated”または“pool”で，“pool”がデフォルト値です。無効な値は単に無視されます。詳細は，スレッドプールの設定に関する静的プロパティのセクションを

参照してください。このプロパティをチャネルまたはコンシューマ管理で設定すると，すべてのサブオブジェクトがこの値を継承します。このプロパティをサプライヤ管理または他のタイプのプロキシで設定しても意味はなく，単に無視されます。

VBProxyPushSupplierQueuePreemptWaterMark

このプロパティはスレッドプールの動作を微調整するために使用されます。このプロパティは，VBProxyPushSupplierThreadModel が “pool” に設定され，プロキシプッシュサプライヤにサービスを提供するスレッドプールのスレッド数が有限の値になるように制限されている場合にのみ適用されます。プロキシプッシュサプライヤオブジェクトは，接続先のコンシューマにイベントをプッシュするためのスレッドをスレッドプールから取得します。このプロキシオブジェクトに多数の保留中のイベントがある場合は，スレッドが独占され，他のプロキシがスレッドを取得できなくなります。このような状況を制御するため，各プロキシオブジェクトのキューにウォーターマークを設定し，ウォーターマークに達した場合は，別のプロキシプッシュサプライヤオブジェクトに優先的にスレッドを提供するようにできます。

デフォルト値は，キューのサイズに基づいて VisiNotify によって決定されます。

VBReceivedEventsCount

受け取ったイベントの数を示します。set_qos API を使ってこのプロパティに任意の値を設定すると，カウンタが 0 にリセットされます。実際に渡された値は無視されます。

VBPendingEventsCount

これは，キュー内にある保留中のイベントの数を示す読み取り専用プロパティです。

VBDiscardedEventsCount

キューのオーバーフローのために破棄されたイベントの数を示します。set_qos API を使ってこのプロパティに任意の値を設定すると，カウンタが 0 にリセットされます。実際に渡された値は無視されます。

VBForwardedEventsCount

下流に転送されたイベントの数を示します。set_qos API を使ってこのプロパティに任意の値を設定すると，カウンタが 0 にリセットされます。実際に渡された値は無視されます。

VBFilteredEventsCount

フィルタに一致しなかったために破棄されたイベントの合計数を示します。set_qos API を使ってこのプロパティに任意の値を設定すると，カウンタが 0 にリセットされます。実際に渡された値は無視されます。

QoS プロパティの管理と検証

QoS ポリシーの管理では，次のインターフェースおよびメソッドがサポートされます。

Interface CosNotification::QoSAdmin

このインターフェースは，チャネル，サプライヤ／コンシューマ管理，およびプロキシサプライヤ／コンシューマによってサポートされています。このインターフェースを使用すると，これらのオブジェクトのクライアントは QoS プロパティを取得および設定できるようになります。

ただし，サポートのレベルにはいくつかの制約があります。

- set_qos() が VisiBroker 固有の QoS に渡され，プロパティ値が不正な場合は，set_qos() は暗黙的に無視され，例外は生成されません。例外は，OMG 固有の QoS に対してのみ生成されます。
- QoS 変更の伝達は, VisiBroker 固有の QoS ではなく OMG 固有の QoS のチャネル／管理／プロキシ階層だけに放散されます。

構造化イベントのヘッダーにある QoS を検証する

これは現在サポートされていません。

QoS ネゴシエーション

次の QoS ネゴシエーション API は，現在サポートされていません。

- CosNotification::QoSAdmin::validate_qos()
- CosNotifyChannelAdmin::ProxySupplier::validate_event_qos()
- CosNotifyChannelAdmin::ProxyConsumer::validate_event_qos()

チャネル管理のプロパティ

チャネル管理では，次のインターフェースがサポートされます。

Interface CosNotification::AdminPropertiesAdmin

このインターフェースは，通知および型付き通知イベントチャネルによってサポートされます。これは，チャネルの管理プロパティを取得および設定する場合に使用します。

次に，サポートされる OMG 定義のプロパティを示します。

- MaxQueueLength
- MaxConsumers
- MaxSuppliers
- RejectNewEvents

VBPersistentStorageSize

永続的なイベント（たとえば，EventReliability が persistent に設定されたチャネル）は，永続的ストレージに保存する必要があります。この管理プロパティを使用すると，ストレージの容量を制限して，VisiNotify が空き容量を使い過ぎないようにすることができます。VisiNotify は，ファイルに永続的なイベントを保存します。この管理プロパティでは，このファイルの最大サイズをキロバイトで指定します。

VBPersistentStorageSize のデフォルト値は 1024 です。その型は `CORBA::Ulong` です。

Static プロパティ

QoS プロパティとは異なり，Static プロパティを設定できるのは，Notification Service の起動時だけです。サービスの実行中に設定することはできません。Static プロパティは，-D<property_name>=<property_value> を使用して，VisiBroker ORB のほかのプロパティのように指定します。

次のプロパティがサポートされます。

vbroker.notify.console

```
vbroker.notify.console = <Boolean>
```

このプロパティは，Notification Service を制御して，VisiBroker コンソールに「Notification Service is ready」というメッセージを表示します。

次に，vbroker.notify.console プロパティでサポートされる値を示します。

- **True（デフォルト）-** メッセージを出力します。
- **False -** メッセージを出力しません。

vbroker.notify.listener.port

```
vbroker.notify.listener.port = <ULong>
```

これは，vbroker.se.iiop_tp.scm.iiop_tp.listener.port のエリアスです。

vbroker.notify.listener.port プロパティのデフォルト値は 14100 です。

vbroker.notify.factory.name

```
vbroker.notify.factory.name = <string>
```

vbroker.notify.factory.name プロパティは，Notification Service で生成されるデフォルトのファクトリ名を指定します。アプリケーションは，resolve_initial_references() ではなく _bind() を実行して，ファクトリへのリファレンスを取得します。

このプロパティのデフォルト値は VisiNotifyChannelFactory です。

vbroker.notify.channel.name

```
vbroker.notify.channel.name = <string>
```

vbroker.notify.channel.name プロパティは，Notification Service で生成されるデフォルトのチャネル名を指定します。アプリケーションは，デフォルトチャネルへのリファレンスを明示的に作成するのではなく，_bind() を実行して取得します。

このプロパティのデフォルト値は default_channel です。

vbroker.notify.channel.threadMaxIdle

```
vbroker.notify.channel.threadMaxIdle = <ULong>
```

vbroker.notify.channel.threadMaxIdle プロパティは，チャネル／プロキシプッシュサプライヤが threadMaxIdle 秒間待機している間に，イベントがキューに到達しなかった場合は，チャネルはイベントを待機するスレッドを解放します。新しいイベントが到達すると，チャネルはスレッドを再起動します。

このプロパティのデフォルト値は 3 秒です。

vbroker.notify.enableEventQoS

```
vbroker.notify.enableEventQoS = <Boolean>
```

vbroker.notify.enableEventQoS プロパティには，チャネルでイベントレベルの QoS を利用して，イベントを配布するかどうかを指定します。True に設定すると，チャネルのパフォーマンスは著しく低下します。

次の値を指定できます。

- **True** - チャネルは, EventReliability などのイベント配布時は, イベントレベルの QoS を利用します。
- **False（デフォルト）** - チャネルは，イベント配布時のイベントレベルの QoS を無視します。かわりに，プロキシ／管理／チャネルの QoS 設定が採用されます。

vbroker.notify.dir

```
vbroker.notify.dir = <string>
```

vbroker.notify.dir には，VisiNotify の永続的ストレージルートのファイルディレクトリまたはデータベーステーブル名を指定します。ConnectionPersistence QoS が適切なレベルに設定されている場合，VisiNotify は，次のオブジェクトを（EventReliability および ConnectionReliability QoS ポリシーによって異なる）リポジトリに保存します。

- イベント
- チャネル
- コンシューマ管理とサプライヤ管理
- プロキシ
- チャネル管理プロパティ，QoS，フィルタ

このプロパティのデフォルト値は「./visinotify.dir」です。

vbroker.notify.ir

```
vbroker.notify.ir = <string>
```

vbroker.notify.ir プロパティには，VisiNotify で使用する IR を指定します。指定された文字列は，IOR 文字列または URL 文字列（corbaloc など）のどちらかになります。

このプロパティのデフォルト値は null です。この場合，VisiNotify は，osagent を使って IR にバインドしようとします。

vbroker.notify.channel.persistentStorageSize

```
vbroker.notify.channel.persistentStorageSize = <ULong>
```

vbroker.notify.channel.persistentStorageSize プロパティは，VBPersistentStorageSize チャネル管理プロパティに似ていますが，チャネルを初めて起動したときにだけ使用するという点で異なります。したがって，VisiNotify は，永続的ストレージから現在の設定を取得します。

このプロパティのデフォルト値は VBPersistentStorageSize です。

vbroker.notify.channel.persistentCommitPolicy

```
vbroker.notify.channel.persistentCommitPolicy = <Boolean>
```

vbroker.notify.channel.persistentCommitPolicy プロパティは，VBPersistentCommitSyncPolicy に似ていますが，チャネルを初めて起動したときにだけ使用するという点で異なります。したがって，VisiNotify は，永続的ストレージから現在の設定を取得します。

このプロパティのデフォルト値は VBPersistentCommitSyncPolicy です。

vbroker.notify.channel.persistentOverflowBlockTimeout

```
vbroker.notify.channel.persistentOverflowBlockTimeout = <ULong>
```

vbroker.notify.channel.persistentOverflowBlockTimeout プロパティは，VBPersistentStorageOverflowBlockTimeout に似ていますが，チャネルを初めて起動したときにだけ使用するという点で異なります。したがって，VisiNotify は，永続的ストレージから現在の設定を取得します。

このプロパティのデフォルト値は VBPersistentStorageOverflowBlockTimeout です。

vbroker.notify.channel.persistentDowngradePolicy

```
vbroker.notify.channel.persistentDowngradePolicy = <ULong>
```

vbroker.notify.channel.persistentDowngradePolicy プロパティは，VBPersistentOverflowDowngradePolicy に似ていますが，チャネルを初めて起動したときにだけ使用するという点で異なります。したがって，VisiNotify は，永続的ストレージから現在の設定を取得します。

有効な値は次のとおりです。

- AnyOrder (0)
- FifoOrder (1)
- LifoOrder (4)

値が，これ以外の値に設定された場合は，チャネルは暗黙的に値 0（AnyOrder）を採用します。

vbroker.notify.channel.persistentEvent

```
vbroker.notify.channel.persistentEvent = <Boolean>
```

vbroker.notify.channel.persistentEvent プロパティは，EventReliability に似ていますが，チャネルを初めて起動したときにだけ使用するという点で異なります。したがって，VisiNotify は，永続的ストレージから現在の設定を取得します。

True を指定すると，チャネルの EventReliability は Persistent に設定され，True を指定しないと，BestEffort に設定されます。

vbroker.notify.channel.iorFile

```
vbroker.notify.channel.iorFile = <string>
```

vbroker.notify.channel.iorFile プロパティには，VisiNotify がデフォルトチャネルの IOR をダンプできるファイル名を指定します。このプロパティは，3.x バージョンと同じ構文，-ior < ファイル名 > オプションを使用します。

このプロパティのデフォルト値は null です。

vbroker.notify.channel.passiveProxyPersistenceMask

```
vbroker.notify.channel.passiveProxyPersistenceMask = <Boolean>
```

一般に，非アクティブのプロキシ（プロキシプッシュコンシューマやプロキシプルサプライヤ）を保存する必要はありません。システムがクラッシュしたり再起動した後は，非アクティブなプロキシのユーザーは存在しなくなるからです。

このプロパティは，次の設定を使用して，非アクティブなプロキシのデフォルト ConnectionReliability 設定を派生させるために使用します。

- let admin's persistence setting = 1 if admin's ConnectionReliability = Persistent, else let it be 0.
- default persistence of proxy = (this property setting) && (its admin's persistence setting)

このデフォルトの永続性の値が True に設定されている場合は，非アクティブなプロキシのデフォルト ConnectionReliability は Persistent に設定されます。永続性の値が True 以外に設定されている場合は，BestEffort に設定されます。

このプロパティのデフォルト値は False です。

vbroker.notify.channel.maxDelay

```
vbroker.notify.channel.maxDelay = <ULong>
```

vbroker.notify.channel.maxDelay プロパティは，遅延時間（ミリ秒単位）を制御する設定です。この遅延時間は，プロキシプッシュサプライヤがコンシューマにイベントを配布するときに，条件付きで適用する時間です。このプロパティは，チャネルのパフォーマンスを調整する場合にも使用できます。

このプロパティのデフォルト値は 2000 ミリ秒です。最低値は 20 ミリ秒，最大値は 2000 ミリ秒です。

vbroker.notify.threadPool.threadMax

```
vbroker.notify.threadPool.threadMax = <ULong>
```

このプロパティは，任意の時点でスレッドプール内に存在できるスレッドの最大数を指定します。

このプロパティのデフォルト値は 0（無制限）です。

vbroker.notify.threadPool.threadMin

```
vbroker.notify.threadPool.threadMin = <ULong>
```

このプロパティは，任意の時点でスレッドプール内に存在できるスレッドの最小数を指定します。

このプロパティのデフォルト値は 0 です。

vbroker.notify.threadPool.threadMaxIdle

```
vbroker.notify.threadPool.threadMaxIdle = <ULong>
```

このプロパティは，スレッドがスレッドプール内にアイドル状態で存在できる時間（秒数）を指定します。アイドル時間が経過したスレッドはガベージコレクションによって回収されます。

このプロパティのデフォルト値は 300 秒です。

vbroker.log.enable

vvbroker.log.enable = <Boolean>

このサーバーのデバッグログステートメントを表示するには，このプロパティを true に設定します。デバッグログフィルタのさまざまなソース名オプションについては，『VisiBroker for C++ 開発者ガイド』の「デバッグログのプロパティ」を参照してください。

サポートのレベル

次の表は，各 QoS プロパティのサポートのレベルを示しています。

プロパティ	サポートされる値	メッセージ	プロキシ	管理	チャネル	コメント
1. 一般						
Priority	Short の値	はい	はい	はい	はい	値の範囲は -32767 から +32767 です。
2. イベントの永続性						
EventReliability	• BestEffort • Persistent	はい	いいえ	いいえ	はい	Persistent イベントでは，同じイベントが何度もコンシューマに配布される場合があります。
VBPersistentDbType	Short の値	いいえ	いいえ	いいえ	はい	値 1（デフォルト）は，メモリマップ永続化になります。値 2 は，フラットファイル永続化になります。
VBPersistentCommitSyncPolicy (extension)	• False（デフォルト） • True	いいえ	いいえ	いいえ	はい	
VBPersistentStorageOverflowBlockTimeout (extension)	あらゆる ULong の値	いいえ	いいえ	いいえ	はい	単位：秒
VBPersistentStorageOverflowDowngradePolicy (extension)	• AnyOrder (0) • FifoOrder (1) • LifoOrder (4)	いいえ	いいえ	いいえ	はい	
3. 接続の永続性						
ConnectionReliability	• BestEffort • Persistent	いいえ	はい	はい	はい	
4. キューオーバーフローの処理						

プロパティ	サポートされる値	メッセージ	プロキシ	管理	チャネル	コメント
MaxEventsPerConsumer	OMG 仕様ごと	いいえ	○(プロキシサプライヤのみ)	○(コンシューマ管理のみ)	はい	いいえ
DiscardPolicy	• AnyOrder(デフォルト) • FifoOrder • LifoOrder	いいえ	○(プロキシサプライヤのみ)	○(コンシューマ管理のみ)	はい	
5. イベントの永続性						
StopTime	サポートされていません	はい	いいえ	いいえ	いいえ	
Timeout	サポートされていません	はい	はい	はい	はい	
StopTimeSupported	サポートされていません	いいえ	はい	はい	はい	
6. イベントの配信						
StartTime	サポートされていません	はい				
StartTimeSupported	サポートされていません	いいえ	はい	はい	はい	
OrderPolicy	• AnyOrder • FifoOrder(デフォルト) • PriorityOrder	いいえ	はい	はい	はい	
MaximumBatchSize	サポートされていません	いいえ	はい	はい	はい	
PacingInterval	サポートされていません	いいえ	はい	はい	はい	
7. フロー制御						
VBQueueLowWaterMark (extension)	ULong 値	いいえ	○(プロキシサプライヤのみ)	○(コンシューマ管理のみ)	はい	
VBQueueHighWaterMark (extension)	ULong 値	いいえ	○(プロキシサプライヤのみ)	○(コンシューマ管理のみ)	はい	

フィルタオブジェクトを使用したイベントフィルタリング

OMG 通知サービス仕様では，2 種類のフィルタを定義しています。

- 転送フィルタ
- マッピングフィルタ

転送フィルタを使用すると，クライアントによって設定された制限を満たすイベントが転送されます。そのため，コンシューマは，転送フィルタを使用すれば，必要なイベントだけを受信できます。転送フィルタオブジェクトは，CosNotifyFilter::Filter インターフェースを実装します。

マッピングフィルタを使用すると，コンシューマは，制限を満たすイベントの優先順位プロパティと存続期間プロパティを変更できます。マッピングフィルタオブジェクトは，CosNotifyFilter::MappingFilter インターフェースを実装します。ただし，VisiNotify は現在マッピングフィルタをサポートしていません。

イベントのフィルタリング

VisiNotify のイベントフィルタリングは，構造化イベント，型付きイベント，およびシーケンスイベントで実行されます。型なしイベントでは，フィルタリングはサポートされていません。シーケンスイベントでは，VisiNotify はシーケンス内の最初のイベントだけをフィルタリングします。シーケンスの最初のイベントがフィルタを満たさない場合は，シーケンス全体が破棄されます。

メモ　各イベント （構造化，型付き，シーケンス）の詳細については，「OMG 通知サービス仕様」の「セクション 2」を参照してください。

転送フィルタの評価

フィルタオブジェクトは，コンシューマ／サプライヤプロキシまたはコンシューマ／サプライヤ管理オブジェクトなどのターゲットオブジェクトに結び付けられます。指定したすべてのフィルタオブジェクトには一連の制約があり，各制約は Extended Trader Constraint Language（TCL）で表されます。制約式は，TRUE（イベントが制約を満たす）または FALSE（イベントが制約を満たさない）を求めます。

制約が 1 つでも TRUE に設定されていれば，フィルタオブジェクトはイベントをただちに転送します。ターゲットオブジェクトに結び付けられたフィルタが FALSE に設定されている場合，イベントは破棄されます。制約式を記述する方法については 86 ページの「フィルタ制約式の記述」を，Extended TCL の詳細については 87 ページの「Extended Trader Constraint Language（Extended TCL）」を参照してください。

転送フィルタオブジェクトが管理オブジェクトに登録されている場合は，その管理オブジェクトに関連付けられているすべてのプロキシオブジェクトが転送フィルタを適用します。プロキシオブジェクトまたは管理オブジェクトに適用されているフィルタがない場合は，受信されたすべてのイベントが，次の配布ポイントに転送されます。

フィルタが，管理オブジェクトに管理オブジェクトのプロキシとともに結び付けられると，管理オブジェクトの作成に使用したセマンティクスが **AND** または **OR** かによって，イベント転送の方法は異なります。AND セマンティクスを使って作成された管理オブジェクトは，イベントが管理とそのプロキシフィルタの両方を渡す必要があることを示します。OR セマンティクスを使って作成された管理オブジェクトは，イベントが管理またはそのプロキシフィルタのどちらかを渡す必要があることを示します。

コンシューマ管理を作成するには，チャネルの `new_for_consumers()` を呼び出し，値 `AND_OP`（AND セマンティクス用）または `OR_OP`（OR セマンティクス用）を渡して，コンシューマ管理オブジェクトのフィルタ間グループ演算子のセマンティクスを設定します。同様に，サプライヤ管理を作成するには，`new_for_suppliers()` を呼び出します。チャネルで `default_consumer_admin()` または `default_supplier_admin()` を呼び出すと，デフォルトのコンシューマ管理またはサプライヤ管理が，それぞれ ADN セマンティクスとともに戻されます。

メモ　AND セマンティクスや OR セマンティクスで使用されるメソッドの詳細については，OMG 通知サービス仕様（セクション 3.4「The CosNotifyChannelAdmin Module」）を参照してください。

転送フィルタの使い方

転送フィルタを適用するには，次の手順にしたがいます。

1 **転送フィルタファクトリを取得します。**VisiNotify は，デフォルトのフィルタファクトリを提供します。フィルタファクトリへのリファレンスを取得するには，次のチャネルでメソッド `default_filter_factory()` を呼び出します。

```
CosNotifyFilter::FilterFactory_var ffact = channel->default_filter_factory();
```

2 **転送フィルタオブジェクトを作成します。**VisiNotify は，OMG 通知サービスによって指定された Extended Trader Constraint Language だけをサポートします。制約を指

定するフィルタを作成するには，ステップ 1 で取得したフィルタファクトリオブジェクトでメソッド create_filter(EXTENDED_TCL) を呼び出します。

```
CosNotifyFilter::Filter_var filter = ffact->create_filter( "EXTENDED_TCL" );
```

3 **制約を作成します。**特定のフィルタオブジェクトには，一連の制約を関連付けることができます。制約式は，Extended TCL で記述されます。

次に，一連の制約とシンプルな制約式の作成方法について説明します。

```
CosNotifyFilter::ConstraintExpSeq constraints;
constraints.length(1); // 1 つの制約を格納します。
constraints[0].constraint_expr = CORBA::string_dup( "$balance == 123.45");
```

メモ Extended TCL の詳細については，87 ページの「Extended Trader Constraint Language（Extended TCL）」と OMG 通知サービス仕様（セクション 2.4「The Default Filter Constraint Language」）を参照してください。

4 **制約をフィルタオブジェクトに追加します。**一連の制約を追加するには，ステップ 2 で取得したフィルタオブジェクトの内，ステップ 3 で作成した一連の制約を通過したフィルタオブジェクトでメソッド add_constraints を呼び出します。

```
filter->add_constraints( constraints );
```

メモ 制約の修正や，フィルタオブジェクトからの制約の取得など，その他のオペレーションの詳細については，OMG 通知サービス仕様（セクション 3.2.1「The Filter Interface」）を参照してください。

5 **フィルタをターゲットオブジェクトに追加します。**ターゲットオブジェクトは，管理オブジェクトまたはプロキシオブジェクトになります。ターゲットオブジェクトは，フィルタオブジェクトが結び付けられる前に作成する必要があります。このサンプルは，構造化プッシュサプライヤプロキシを示しています。

```
// 構造化プッシュサプライヤプロキシを作成します。
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxySupplier_var proxy
= admin->obtain_notification_push_supplier
CosNotifyChannelAdmin::STRUCTURED_EVENT, proxy_id);
CosNotifyChannelAdmin::StructuredProxyPushSupplier_var supplier
= CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(proxy);
```

フィルタオブジェクトをターゲットオブジェクトに結び付けるには，ターゲットオブジェクトで add_filter を呼び出します。add_filter オペレーションは，フィルタオブジェクトを受け取り，特定のターゲットオブジェクトに固有のフィルタ ID を戻します。このサンプルは，構造化プッシュサプライヤプロキシで呼び出され，ステップ 2 で作成されたフィルタオブジェクトに渡される add_filter を示しています。

```
CORBA::Long filter_id;
Filter_id = supplier->add_filter( filter );
```

メモ フィルタの修正や，ターゲットオブジェクトからのフィルタの取得など，その他のオペレーションの詳細については，OMG 通知サービス仕様（セクション 3.2.4「The FilterAdmin Interface」）を参照してください。

転送フィルタの制限

次のフィルタオブジェクトメソッドは，現在サポートされていません。

- attach_callback
- detach_callback
- get_callbacks

メモ これらのメソッドと共有サブスクリプションの詳細については，OMG 通知サービス仕様（セクション 2.6.5「Obligations on Filter」）を参照してください。

フィルタ制約式の記述

概要

制約式はブール式です。つまり，TRUE または FALSE のいずれかに評価されます。通常，制約式はイベントデータを参照します。イベントデータには，アプリケーションがフィルタリングするかどうかの判断の基準にするフィルタリング可能なデータが含まれています。

構造化イベントの内容

次に示すように，構造化イベントは CosNotification.idl で定義されます。

```
...
typedef string Istring;
typedef Istring PropertyName;
typedef any PropertyValue;

struct Property {
  PropertyName name;
  PropertyValue value;
};
typedef sequence<Property> PropertySeq;

typedef PropertySeq OptionalHeaderFields;
typedef PropertySeq FilterableEventBody;

struct EventType {
  string domain_name;
  string type_name;
};
struct EventType {
  string domain_name;
  string type_name;
};
struct FixedEventHeader {
  EventType event_type;
  string event_name;
};

struct EventHeader {
  FixedEventHeader fixed_header;
  OptionalHeaderFields variable_header;
};

struct StructuredEvent {
  EventHeader header;
  FilterableEventBody filterable_data;
  any remainder_of_body;
};
...
```

型付きイベントの内容

型付きイベントには，名前／値の組のシーケンスが含まれています。シーケンスの最初の項目は CosNotification::EventType を参照します。これには，型付きインターフェースの名前を参照する domain_name と，そのインターフェース内のオペレーションの名前を参照する type_name が含まれています。シーケンスの 2 番め以降の項目は，フィルタリング可能なデータです。各項目には，型付きインターフェース内のオペレーションの入力パラメータを参照する名前と，そのオペレーションのパラメータ値を参照する値が含まれています。

たとえば，アプリケーションで，次の IDL で示される型付きイベントを使用できます。

```
interface foo {
  void bar( in string first, in long second );
};
```

この例では，型付きイベント foo::bar を受け取ります。名前／値の組のシーケンスの 2 番めの項目では first という名前と文字列値が対になり，3 番めの項目では second という名前と long 値が対になります。

メモ　構造化イベントと型付きイベントの詳細については，OMG 通知サービス V1.0 仕様（セクション 2.2「Structured Events」とセクション 2.7「Filtering Typed Events」）を参照してください。

Extended Trader Constraint Language（Extended TCL）

OMG 通知サービス V1.0 は，Extended Constraint Language をデフォルトのフィルタ制約言語として指定します。Extended TCL は，OMG Trading Service の Trader Constraint Language（TCL）に基づいており，一部拡張と変更が追加されています。

メモ　TCL に加えられた変更の詳細については，OMG 通知サービス V1.0 仕様（セクション 2.4.1）を参照してください。

Extended TCL で記述された制約式は，TRUE 値または FALSE 値のどちらかに評価されます。この 2 つの値は TCL の予約語です。Extended TCL の TRUE の値は 1，FALSE の値は 0 です。そのため，次のような式を記述できます。

```
TRUE + TRUE
```

この式の結果は 2 になります。部分式を指定するには，次に示すように，式を括弧で囲みます。

```
(TRUE + TRUE) == 2
```

イベントデータへのアクセス

Extended TCL では，イベント内の複合データ型（IDL 型の struct，enum，union，および any）を参照できます。イベントは $ で表されます。また，イベント内の属性は .（ピリオド）を使って参照されます。これは，今日使用されている C++ や Java のプログラミング構造に似ています。

たとえば，構造化イベントの固定ヘッダーの event_name 属性を参照するには，次のように記述します。

```
$.header.fixed_header.event_name
```

型付きイベントでは，アプリケーションに最初のパラメータとして first という名前の文字列をとる bar という名前のオペレーションを含む foo という名前のインターフェースがある場合，first を参照するには，次のように記述します。

```
$.first
```

メモ　イベントデータが存在しないか，オペレーションの両方のオペランドのデータ型が一致しない（'A String' == 3.14 など）場合，制約式は FALSE と評価されます。

表記の短縮

Extended TCL の実行時変数を使用して，イベント内の特定の予約済み属性やフィルタリング可能なデータを参照できます。実行時変数は，識別子名の前に $ を付加して表します。たとえば，$event_name は，実際は $.header.fixed_header.event_name と同じです。実行時変数を使用した場合，識別子は，イベント内の予約済み属性に対応付けられます。識別子がイベント内の予約済み属性でない場合は，フィルタリング可能なデータに対応付けられます。

メモ　一般イベントをフィルタリングする際の表記の短縮については，OMG 通知サービス V1.0 仕様（セクション 2.4.5）を参照してください。

位取り記数法

現在のバージョンの VisiNotify は，位取り記数法をサポートしていません。

等価演算子，関係演算子，論理演算子

Extended TCL では，標準の TCL で使用される演算子と同じ演算子および OMG 通知サービス V1.0 仕様で追加された演算子を使用できます。

メモ　次の表の演算子は，TRUE または FALSE のどちらかに評価されます。

表 5.1　等価演算子，関係演算子，論理演算子

演算子	説明	サンプル
==	等価	($.one + $.two) == 3
!=	不等価	($.one + $.two) != 4
<	より小さい	($.one + $.two) < 3
<=	以下	($.one + $.two) <= 3
>	より大きい	($.one + $.two) > 1
>=	以上	($.one + $.two) >= 2
in	左オペランドが単純なプリミティブ型であり，それが同じプリミティブ型のシーケンスである右オペランドに含まれているかどうかをチェックします。	$.one in $.list_of_nums
~	左オペランド文字列が右オペランド文字列に含まれているかどうかをチェックします。	'Notify' ~ 'VisiNotify'
exist	識別子が存在するかどうかをチェックします。	exist $.one
and	論理積	($.one == 1) and ($.two == 2)
or	論理和	($.one == 1) or ($.two == 2)
not	論理否定	not exist $.one
default	ディスクリミネータ付きの共用体データだけに適用されます。ディスクリミネータ付きの共用体にデフォルトのメンバーが含まれているかどうかをチェックします。	Default $.myUnion

算術演算子

次の表の演算子の結果の型は，オペランドの型によって異なります。厳密に型指定されたオペランドによって最終的なデータ型が決定されます。

メモ　文字データを算術演算で使用できます。長さ 1 の文字列も，文字とみなされます。

表 5.2　算術演算子

演算子	説明	サンプル
+	加算	$.one + $.two
-	減算	$.one - $.two
*	乗算	$.one * $.two
/	除算	$.two / $.one

添字演算子

配列およびシーケンスにアクセスするには，添字演算子 [*n*] を使用します。たとえば，配列の 2 番めの要素にアクセスするには，次のように記述します。

```
$myArray[1]
```

名前／値の組の検索

イベントでは，名前／値の組のシーケンス（フィルタリング可能なデータなど）がよく使用されます。たとえば，フィルタリング可能なデータにアクセスするには，次のような式を記述します。

```
$.filterable_data[2].name = = "balance" and $.filterable_data[2].value > 100)
```

このような式は長くなることがあるため，Extended TCL では，次のように短縮して記述することもできます。

```
$.filterable_data(balance) > 100
```

予約済みの暗黙的メンバー

Extended TCL では，イベントと複合データで予約済みメンバー属性を使用できます。次の表は，予約済みメンバー属性の名前とその目的を示します。

表 5.3　複合データの予約済み属性

属性	説明	サンプル
_length	配列またはシーケンスの長さ	$.mySequence._length
_d	ディスクリミネータ付きの共用体のディスクリミネータ	$.myUnion._d
_type_id	スコープのない IDL 型名	$.myData._type_id
_repos_id	リポジトリ ID	$.myData._repository_id

索引

記号

B

C

E

J

O

P

Q

R

S

ふ

へ

ま