

第 2 章

BizTalk アーキテクチャ

本章では後続の章からも参照される BizTalk のアーキテクチャについて説明します。本章の目的は後続の章のための準備とし、アーキテクチャの個々の詳細については後続の章に譲ります。

後続の章では BizTalk の主な機能領域について扱いますが、ここでは基本的な内容の説明ではなく、その機能がどのように動作し、その機能をプロジェクト内でどのように使用すべきかについて検討します。

2.1 | なぜ BizTalk なのか

BizTalk のアーキテクチャについて検討を始める前に、少し立ち止まって、BizTalk 無しにソリューションを実装する場合はどうするかについて考えてみましょう。これは開発者たちが単にコードを書きたがっているような場合に対し、筆者が顧客と共にしばしば行うアプローチです。勿論、コードを書くことはできるでしょうが、往々にして悪魔は細部に宿るものです。

BizTalk はすべてのソフトウェアの問題を解決する万能薬というわけではありません。適切な使い方をすべきであることを覚えておくことが重要です。やみくもに使用するのではなく、「道理に合った場合でのみ」使用すべきです。筆者はいつも、顧客に対してそれぞれのユースケースで BizTalk からどんな価値を引き出したいのか検討してもらっています。その上で顧客が答えを出しかねているようなら、おそらく BizTalk は最適なソリューションではないでしょう。

架空のシナリオを使用して、BizTalk がなぜ現状のように設計されているのか、その背後にある論理の一部を示してみましょう。

2.1.1 | フェーズ 1

このシステムでは旅行日程予約のリクエストを Windows ファイルシステム（1つのディレクトリ）上に個別のファイルとして渡します。これらのファイルは収集され、その内容をサードパーティー独自のデータ構造から内部で取り扱い可能なフォーマットに変換しなくて

はなりません。

それが済んだら、サードパーティーサプライヤーの、航空便予約、ホテル予約、およびレンタカー予約用 Web サービスにインターフェイスで接続する必要があります。

■ カスタム開発の場合

出発点はサーバー上で自動実行可能な Windows サービスです。この Windows サービスは、いったん開始されると、.NET Framework に含まれる FileSystemWatcher クラスを使用して、ファイルの到着があるかどうかディレクトリを監視することができます。

FileSystemWatcher クラスはあらゆるディレクトリ監視を扱い、ファイルが到着するとイベントを発生させます。するとサービスは複数のファイルを同時に処理するために、.NET ThreadPool クラスを使用して処理スレッドを生成します。これらの各処理スレッドはファイルの内容をメモリに読み込み、再処理を避けるためにそのファイルをディスクから削除します。

ファイル (XML フォーマット) の内容は、標準スキーマに基づいた一般的な XML ドキュメントに変換される必要があります。そのためには、ソース XML ドキュメントから目的 XML ドキュメントに変換する XSLT ドキュメントを作成する必要があります。この XSLT ドキュメントは、Visual Studio 2005 に組み込まれた標準の XML エディタを使用して手動で作成しなければなりません。

XSLT ドキュメントを作成したら、XslTransform クラスを使用して2つのフォーマット間の変換を実行できます。メッセージ内容へのアクセスを簡単にするには、シリアル化可能クラスを使用して、そのメッセージ用に厳密に型指定されたクラスを作成するとよいでしょう。

次のステップでは、XML ドキュメント内の関連する部分を使用して、3つの予約用 Web サービスを呼び出します (例えば、メッセージの「Flights」部分の内容が航空便予約 Web サービスに渡される必要があります)。

プロジェクトに Web 参照を追加して、それぞれの参照を順に呼び出すことができます。そのためには、データをソースメッセージから Web サービスが必要とするフォーマットに手動で変換しなければなりません。これは、一般的には、データをソースメッセージに対応するクラスインスタンスからそれぞれの Web サービスが必要とするフォーマットに対応するクラスインスタンスに手動で割り当てることによって行うことができます。

以上で、問題なく3つの Web サービスを呼び出すことが可能となります。

■ BizTalk 開発の場合

最初に、ファイルフォーマットを定義するために使用される事前定義済み XML スキーマを BizTalk プロジェクトに追加して、新しい BizTalk オーケストレーションを作成します。

この BizTalk オーケストレーションに、BizTalk に対してメッセージをオーケストレーションに「配信」するよう指示するための、論理的な一方向のポートにリンクされた受信シェイプを作成します。3つの各予約用 Web サービスに対する Web 参照を追加します。

次のステップでは、ファイルから取り出したソースデータに基づいて、Web サービスを呼び出すために必要なメッセージを構築します。3つの変換シェイプをオーケストレーション上に追加し、それぞれの変換シェイプに対応させて、オーケストレーションに対して提供されたファイルメッセージを各予約用 Web サービスに渡すメッセージに変換するための、グラフィカルなマップを作成します。マップ内での変換の指定は簡単で、送信元 XML ドキュ

メントと送信先 XML ドキュメント間に接続を描くだけです。

このソリューションを BizTalk に配置すると、Web サービスの送信ポートは自動的に作成されます。あとは、ハードディスクドライブからファイルを取り出すために、ファイルアダプタを使用して新しい受信場所を作成する必要があります。

■ フェーズ 1 のまとめ

既にお分りのとおり、BizTalk ソリューションではカスタムコードはまったく必要ありません。必要なのはフェーズ 1 の要件を満たす構成と配置だけです。一方、カスタムソリューションでは、問題を解決するためにかなりの量のコードが必要となります。

カスタムソリューションの開発労力はそれほど大きなものではありません。そのためこの開発競争では圧倒的な勝者は存在しません。しかし、かなりの量のテスト、特にネットワーク通信およびその他に不具合が発生した場合のエラー処理に関するテストが必要となります。BizTalk ソリューションの場合は、これを「プラットフォーム」が処理してくれるため、そういったことがまったく必要ありません。一般的に、このような「不幸な」シナリオを扱うには、かなりの開発労力と事前の設計上の検討が必要となります。

2.1.2 フェーズ 2

フェーズ 1 によって作成されたソリューションを配置してしばらく経つと、やがてビジネス機能に対するソリューションの重要性が高まってきます。そのため、将来の負荷増に対応するため、また、ソリューション自身もしくはソリューションが依存する予約用 Web サービスで発生したエラーからの復帰を確実にするために、ソリューションをスケールアウトする必要があります。

■ カスタム開発の場合

カスタム開発者は、上記の新しい要件を満たすために設計を白紙に戻さなければならないでしょう。フェーズ 1 の設計では、ディレクトリから取り出したメッセージは、処理中および予約用 Web サービスの呼び出し中に、メモリに保持されるようになっています。

このシナリオに対してハードウェア障害を追加することを検討してみましょう。メッセージはメモリに保持されるので、電源障害が発生した場合、そのメッセージは失われてしまうでしょう。ファイルシステムはトランザクションに対応していないので、軽量トランザクションも使用できません。

障害が発生した場合にメッセージが失われずに後の処理のために永続的ストアに残るように、メッセージの処理中はトランザクションストアが使用されなければなりません。SQL Server データベースがこの場合の素直な選択です。少なくとも、処理の成功を待ってペンディング状態の「進行中の」メッセージを保持し、処理が成功した時点でメッセージを破棄できるように、データベーステーブルを設計する必要があります。

このデータベースは高いレベルの挿入操作および削除操作に対応できるように設計する必要があります。さらに、そのデータベースがボトルネックにならないように、ストレステストやインデックスチューニングを行わなければなりません。設計が適切でない場合、データベースはすぐにボトルネックとなるでしょう。

Windows サービスは、「ペンディング」処理スレッドが定期的に起動して、データベーステーブル内に長期間のペンディング処理のために残っているメッセージがないかどうか確認

するように設計する必要があるでしょう。残っている場合、そのメッセージは再処理する必要があります。エラーが発生している可能性が高いからです。

復元性のシナリオは、Windows サービスの実装だけではなく、依存する Web サービスにも関わります。例えば、航空便予約用 Web サービスが利用不能で、リトライ可能な回数を超えた場合、その Web サービスが利用可能になるまで、処理を「一時停止」する必要があります。

さらに、航空便予約用 Web サービスが数時間ダウンした場合、メモリ内に処理を待っているメッセージが大量に残ったままになる可能性があります。これでは継続性があるとも復元性があるとも言えないことは明らかです。そのため、Windows サービスが、ペンディングされたメッセージを格納するために、先ほど触れたデータベースをさらに活用する必要があります。

フェーズ1のもう1つの問題は競合状態です。新しい拡張要件を導入する際に、Windows サービスを実行する複数のマシンが、何らかの形のロックングを行わなければ、同一のファイルを2度処理してしまうことは簡単にありうるでしょう（例えば、FileSystemWatcherは複数のマシンに対して同時にイベントを呼び出すことができます）。ただし、ファイルを開く時に排他的ロックを使用すれば、このリスクは軽減されるでしょう。

また、FileSystemWatcherにはファイルが書き込みを終えたタイミミングの認識に関する問題もあります。一般的に、FileSystemWatcherは、ファイルシステムイベントに依存するため、同じファイルについて、書き込みが終了する前に、複数のイベントを報告します。そのファイルにアクセスしようとする、書き込みがまだ完了していないので、ロックングの問題が発生する可能性があります。そのため、そのファイルをキューに入れたあとで実際の処理を開始する必要があります。

以上で、ファイルを処理するために同一のディレクトリを「監視」するように設定された複数のマシンすべてに Windows サービスを配置することができます。このように設計すれば、あるマシンに障害が発生しても、別のマシンがシームレスに処理を続行し、すべてのメモリ内ファイルはデータベーステーブル内で保護されるでしょう。

■ BizTalk 開発の場合

このシナリオの問題を解決するには、別の BizTalk サーバーを BizTalk Server グループに追加します。その結果、オリジナルの BizTalk サーバーに障害が発生した場合でも簡単に回復できます。このサーバーは、SQL Server 上にホストされた同一の BizTalk メッセージボックスを使用します。

BizTalk メッセージボックスをホスティングする SQL Server は、Windows プラットフォームに組み込まれた標準的なクラスタリングサポートを使用して、アクティブ/パッシブ構成にクラスタリングする必要があります。また、共有ディスクアレイを活用する必要があります。

この共有 BizTalk メッセージボックスアーキテクチャによって、複数のサーバー間でシームレスな負荷分散が可能となり、カスタムコードをまったく使用せずに、簡単にスケラビリティを高められます。

依存関係にある予約用 Web サービスが使用不能となった場合、BizTalk アダプタが、設定済みのリトライ間隔に基づいて自動的にリトライを実行し、リトライの最大回数に達したらメッセージを一時停止します。停止したメッセージは、アドミニストレータが標準の BizTalk 管理ツールを使用して再開します。

■ フェーズ 2 のまとめ

このケースで明らかになったのは、これらの新しい要件が、事実上あらゆるシステムに当てはまるもので（フェーズ 1 は特に現実の世界の話ではありませんでしたが、役に立つ比較を示しています）、カスタム開発者はソリューション全体の設計を白紙に戻し、大量のカスタム開発を含む全面的な再設計を行わなければならない、またそれはテストおよびサポートが必要だということです。

BizTalk ソリューションでは、フェーズ 2 用に（またはフェーズ 1 用に）コードを書く必要は一切なく、サーバーを追加することによって既に組み込まれている復元性と拡張性を活用するだけです。これは既存のソリューションを停止せずに実行することができます。このソリューションでは BizTalk の圧倒的勝利です。開発コストはきわめて低く抑えることができ、筆者の経験では開発コストよりも大きな割合を占めるサポートコストも、Microsoft 社が BizTalk プラットフォームおよびそのコンポーネントに対してサポートしてくれるので、大幅に削減されます。

2.1.3 フェーズ 3

フェーズ 2 のソリューションはしばらくの間稼動し続け、ソリューションに責任を負う企業と予約サービスのプロバイダの双方にとって次第に重要なものとなってきました。同産業が成長していることから、メッセージの負荷が短期間に大幅に増大することが予想されています。

この重要性を考慮して、SLA（Service Level Agreement；サービスレベルアグリーメント）を適切に定義するための契約書が作成されます。また、処理に関する紛争が発生したのを受けて、メッセージを追跡して処理上の問題および SLA 違反を特定するという新しい要件が導入されました。

■ カスタム開発の場合

カスタム開発者はフェーズ 2 のソリューションにメッセージ追跡を導入する必要があります。この追跡システムは、受け取ったメッセージが変換される前の詳細な状態と変換後にメッセージの変更された点の両者を変換前後で比較できる状態で記録し、変換によってメッセージのコアペイロードが変更されていないことを確認する必要があります。

フェーズ 2 で導入したデータベースを活用するのが素直な選択でしょう。活用にあたっては、メッセージ本文を格納できるデータ型（image または text）を使用して、最も単純な形式の別のテーブルにメッセージ本体を保管することが必要となります。フェーズ 2 と同様、このテーブルには大きな負荷がかかります。そのため、処理中のボトルネックにならないように、厳しいテストおよびチューニングが必要となります。

SLA 監視を実装するには、処理のあらゆる主要な段階（支払いの到着時、支払い変更の開始および終了時など）で正確な時刻取得機能を実装する必要があります。さらに、各予約用 Web サービスコールのどちらかの側でタイムスタンプを記録して、コールの継続時間を計算したり、予約サービスのサプライヤーが取り決められた SLA に違反したタイミングを特定したりする必要があります。そのため、追跡ポータルを実装して、1 次サポート担当者と開発者が、処理中に記録された追跡データに簡単にアクセスできるようにする必要があります。

カスタムソリューションに対して最適なパフォーマンスを実現するには、広範囲に及ぶパ

パフォーマンスチューニングおよびベンチマーキングが必要となるでしょう。また、パフォーマンス特性によっては、何らかの形の負荷調整をソリューション全体に実装して、処理システムまたは依存関係にある複数のソリューションの過負荷を避けなければなりません。これらすべてのアクティビティには、ベテラン開発者が持つ広範囲に及ぶ開発に関する専門知識が求められます。

■ BizTalk 開発の場合

BizTalk では、デフォルトで基本的なメッセージ追跡がオンになっています。メッセージ本文追跡は、受信メッセージを記録するために受信場所のレベルで、また、変換済みメッセージを記録するためにオーケストレーションのレベルでオンにしなければなりません。

BAM (Business Activity Monitoring ; ビジネスアクティビティ監視) は、SLA の要件を満たすための記録に使用することができます。追跡するビジネスマイルストーンは BAM アドインを使用して Excel のスプレッドシート内に定義します。パフォーマンスが高く拡張性の高い方法でこの情報を格納するために必要なデータベースインフラストラクチャは、BAM 管理ツールによって自動的に作成され、開発やコーディングはまったく必要ありません。また、Microsoft 社によって完全にサポートされています。

BizTalk 開発者は、TPE (Tracking Profile Editor ; 追跡プロファイルエディタ) を使用するだけで、オーケストレーションの BAM がマイルストーンを記録する地点にマークを付けることができます。これは、追跡を開始する稼働中のシステムに簡単に配置することができます。

ビルトインの BAM ポータルは、記録された BAM データへの簡単なインターフェイスとなります。これについても、コーディングは一切必要ありません。

BizTalk メッセージングエンジンには、数多くの調整用コントロールが組み込まれており、これによりメッセージ処理に対するきめ細かい制御を可能とし、また、メッセージングエンジンを過負荷から保護します。BizTalk 管理ツールを使用することによって、メッセージングエンジンを停止させずにすべての設定を行うことができます。

■ フェーズ 3 のまとめ

フェーズ 2 からフェーズ 3 への移行で明白なことは、カスタムソリューションでは、新しい要件を満たすために、またしても大きな変更を行わなければならないということです。一方、BizTalk ソリューションでは、依然として全体に渡ってカスタムコードが不要、かつ完全にサポートされていることから、BizTalk の有用性は明白です。また、BizTalk がなぜ現状のように設計されているのか、その現実世界の例も示すことができました。



メモ

確かに不自然で明らかに簡素化されたシナリオでしたが、コードをまったく必要としなかったことは強調するに値します。一般的なソリューションでカスタムコードが不要であることは稀です。カスタムコードを書かなければならないということは弱点でもなければ、それ自体避けるべきものでもありません。しかし、コードの記述を最小限に抑えることはきわめて重要です。正しく作成することは困難であることに加え、時に高い負荷がかかる場合、パフォーマンスおよび拡張性の問題の原因となるからです。

2.2 | アーキテクチャの概要

2

図 2-1 は、BizTalk アーキテクチャを示しています。本図は、本書の中でたびたび使用しています。

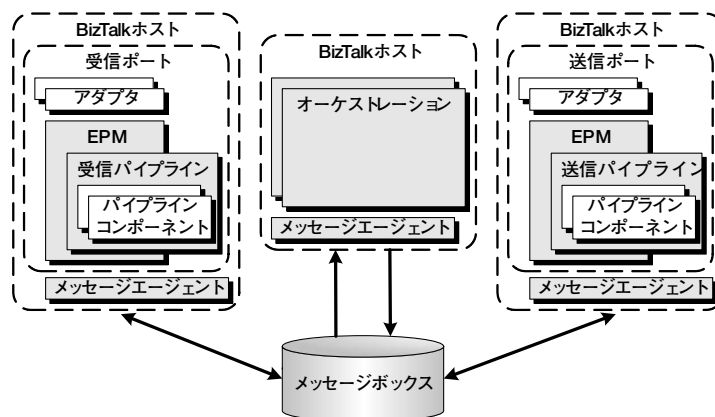


図 2-1 BizTalk アーキテクチャ

左側の受信ポートは、アダプタ、EPM (End Point Manager ; エンドポイントマネージャ)、および受信パイプラインで構成されており、送信元システムから BizTalk へメッセージを送信し、また、そのメッセージを BizTalk メッセージボックスへ格納します。

中央のオーケストレーションは BizTalk のオプション機能であり、必要に応じてそれを活用してメッセージを処理できます。オーケストレーションによって処理されるメッセージは、BizTalk メッセージボックスから取り出され、オーケストレーションから送信されたすべてのメッセージが BizTalk メッセージボックスに格納されます。

右側の送信ポートは、アダプタ、エンドポイントマネージャ、および送信パイプラインで構成されており、BizTalk メッセージボックスからの送信メッセージを収集し、また、そのメッセージを送信先システムに転送します。

以降の節では、アーキテクチャの構成要素がどのように連携するのか理解を深めるために、主要なコンポーネントを一通り見てみましょう。

2.3 | メッセージ

メッセージはすべての BizTalk ソリューションの中を流れる血液のようなものです。型を問わず、すべてのデータは BizTalk 内ではメッセージとなります。一般的に、メッセージは XML ドキュメントとなりますが、バイナリ形式のメッセージも処理することができます。ただしその場合、使用できる機能の数は制限されます。例えば、バイナリ形式のメッセージに対して BizTalk マッパーやルールエンジンを使用することはできません。

1つの BizTalk メッセージには、最低1つのパートが含まれます。これは一般的にはメッセージの本文です。1つのメッセージ内に複数のデータを保持したい場合、マルチパートメッセージを作成できます。

BizTalk はまた、メッセージ本文を追跡する機能を備えています。これは、HAT (Health and Activity Tracking ; 状態と動作状況の追跡) ツールを使用して表示できます。HAT は、大きなオーバーヘッドが生じるため、デフォルトでは無効となっています。本当に必要な場合のみ使用するべきです。



メモ

代替アプローチの詳細は「第6章 ビジネスアクティビティ監視」を参照してください。BizTalk メッセージの詳細は「第4章 パイプライン」と「第5章 オーケストレーション」で扱います。

2.4 アダプタ

アダプタは BizTalk と外界とを接続するものです。このアーキテクチャにより、アダプタを除く BizTalk エンジン全体は、リモートシステムが課すどんな要件または通信規則からも分離されています。

大抵の場合、アダプタはトランスポートと共に動作し、送信または受信されるデータについて基礎的な情報を一切持ちません。アダプタは回線からバイトストリームを受信し、BizTalk エンジンに送ります。アダプタはほとんどデータに依存しません。つまりアダプタは、受信するものが XML であろうがフラットファイルであろうが、あるいはバイナリデータであろうが一切関知しません。

受信の場合、アダプタはデータをリスニングまたはポーリングして、BizTalk メッセージでラッピングします。リスニングの場合、受信データのエンドポイント (または URL) をリスニングして、必要な方法 (認証、トランザクション、ハンドシェイクなど) でリモートシステムと通信します。ポーリングの場合は、通常 BizTalk に受信される必要がある新しいデータのエンドポイントを定期的に確認する処理が行われます。

送信の場合、アダプタは BizTalk メッセージボックスからメッセージを受信し、そのメッセージに関連するトランスポートまたはプロトコルを使用して、リモートシステムに送信します。

例えば、SQL アダプタは BizTalk メッセージボックスから XML メッセージを取り出し、最終的には、すべて XML メッセージの内容に基づいて、リモートの SQL Server データベース内のレコードを作成します。SQL アダプタはメッセージを BizTalk からリモートシステムに転送します。残りの BizTalk エンジンはいかなるエンドポイントの仕様も要求しません。完全に分離されています。

アダプタは、メッセージを受信したら、そのメッセージを処理するために BizTalk メッセージングエンジンに渡します。メッセージングエンジンは、メッセージの受信後および送信前に、パイプライン内でメッセージを処理します。

**メモ**

アダプタの詳細については、「第 3 章 アダプタ」を参照してください。

2

2.5 | パイプライン

パイプラインは受信メッセージおよび送信メッセージに対するデータ処理と検証の機能を提供します。受信メッセージは受信パイプラインを使用し、送信メッセージは送信パイプラインを使用します。パイプラインの標準的な使用法は、受信データを XML に正規化し、送信データを必要なフォーマットに非正規化することです。ただしこれは、決してそうでなければならないということではありません。よくある誤解は、BizTalk では常にデータを XML フォーマットに正規化する必要があるというものです。これは真実ではありません。

BizTalk では、ほとんどのシナリオに適した 4 つのパイプラインが用意されています。

以下のパイプラインは受信側で使用できます。

- XmlReceive - XmlReceive パイプラインは、XML 逆アセンブラパイプラインコンポーネントを搭載する。
- PassThruReceive - PassThruReceive パイプラインは、パイプラインコンポーネントを一切搭載しない。

以下のパイプラインは送信側で使用できます。

- XmlTransmit - XmlTransmit パイプラインは、XML アセンブラパイプラインコンポーネントを搭載する。
- PassThruTransmit - PassThruTransmit パイプラインは、パイプラインコンポーネントを一切搭載しない。

表 2-1 および表 2-2 に示すとおり、パイプラインの実行は複数の処理ステージに分割されています。各処理ステージは特定のタイプの処理を実行するように設計されています。各ステージはオプションで必要に応じたパイプラインコンポーネントを格納できることに注意してください。パイプラインの詳細は「第 4 章 パイプライン」で扱います。

これらのパイプラインステージを通じたメッセージフローおよびパイプライン処理は、BizTalk におけるすべての処理と同様、ストリームベースとなっています。BizTalk はこれを利用して、メッセージ全体をメモリにロードせざるをえない状態を回避します。大きなメッセージをメモリにロードすると、メモリ不足状態が発生して、BizTalk プロセスが停止することもあるからです。

表 2-1 受信パイプラインステージ

ステージ	内容
デコード	デコードステージでは後続のステージがメッセージを処理するために必要なすべての処理が実行される。よく行われる例としては、復号化や圧縮解除がある。これらは逆アセンブルや検証の前に実行される必要がある。
逆アセンブル	コンポーネントにもよるが、逆アセンブルステージでは、メッセージのタイプの特定、XML への変換、(必要に応じて) メッセージの分割などが行われる。
検証	検証ステージではメッセージの検証が実行される。一般的に、XML スキーマを参照して XML メッセージが検証される。
パーティの解決	パーティの解決ステージは、メッセージの送信者を特定するために使用される。一般的に、アダプタが提供する X.509 証明書または Windows ユーザーを使用して行われる。

表 2-2 送信パイプラインステージ

ステージ	内容
プリアセンブル	プリアセンブルステージでは、後続のステージがメッセージを処理するために必要なすべての処理が実行される。多くの場合このステージは、アセンブルステージによってフラットファイルに変更される前の XML メッセージに対する、最後の変更を行うために使用される。
アセンブル	アセンブルステージは逆アセンブルステージの逆で、メッセージを XML から一般的にはフラットファイルのような異なるフォーマットに変換する。また、複数のメッセージをアセンブルして、1つの出力メッセージに変換することも可能である。
エンコード	エンコードステージは、メッセージに対する最後の変更を実行するために使用される。一般的に、エンコード、暗号化、圧縮などである。

パイプラインコンポーネントが呼び出されると、メッセージストリームからメッセージデータを読み込み、必要な正規化または変換を実行することができます。そのため、フラットファイル逆アセンブルパイプラインコンポーネントの場合、ストリームからフラットファイルの内容を読み込み、XML メッセージ作成のためのフラットファイルスキーマ構文解析または変換を適用します。

このフラットファイル逆アセンブルパイプラインコンポーネントから、新しいストリームが返されます。しかし、そのストリームにはフラットファイルフォーマットではなく新しい XML ドキュメントが格納されています。次にこれは、パイプラインを経由して、次のパイプラインコンポーネントに渡されます。

パイプラインコンポーネントの一般的な用途は、フラットファイルのアセンブル/逆アセンブル、メッセージのバッチ処理/デバッチ処理^{*1}、XML スキーマを参照するメッセージの検証、デジタル署名の確認、暗号化/復号化などです。

数多くのパイプラインコンポーネントが BizTalk に含まれています。新しいコンポーネントは、簡単に作成できます。カスタムパイプラインコンポーネント開発はよく行われるもので、カスタムアダプタよりもはるかに必要とされます。



メモ

パイプラインおよびパイプラインコンポーネントの構築方法の詳細については「第 4 章 パイプライン」を参照してください。

*1 監修注：バッチ処理とは、受信したメッセージを複数のメッセージに分割する処理です。デバッチ処理は、逆に複数のメッセージを結合して単一のメッセージを生成する処理です。

2.6 サブスクリプション

2

メッセージがアダプタおよび関連するパイプラインによって処理されたら、当事者へのルーティングを行う必要があります。BizTalk オーケストレーションまたは送信ポートは、サブスクリプションを介して特定のメッセージ内にその当事者を登録することができます。

1つのサブスクリプションは、サブスクライバが必要なメッセージを確実に取得できるための数多くの条件で構成されています。メッセージが処理される際、メッセージの内容、経由したアダプタ、および経由した BizTalk ポートの内容を基にメタデータが生成され、メッセージの「コンテキスト」に書き込まれます。これらのコンテキストプロパティは、メッセージのルーティング方法を決定するために、サブスクリプションを参照して評価されます。

一般的な XMLReceive パイプラインでは、XML 逆アセンブラコンポーネントは「MessageType」コンテキストプロパティを追加します。XML メッセージの場合、このプロパティには関連する XML スキーマの名前空間およびルートノードが格納され、それらによってメッセージタイプが一意に識別されます。

すると BizTalk オーケストレーションと送信ポートは、特定のスキーマタイプのすべてのメッセージや、メッセージを受信した受信ポートの名前やメッセージを送信する顧客の名前など、コンテキストプロパティの任意の組み合わせによってサブスクライブできるようになります。

これらのサブスクリプションは、BizTalkMsgBoxDB データベースの Subscription テーブルに保存され、BizTalk メッセージエージェントによってメッセージがメッセージボックスに対して発行される際に評価されます。これは BizTalk ホストの一部として実行されます。

図 2-2 は、BizTalk Server 2006 管理コンソールのサブスクリプションクエリ機能を示しています。現在 BizTalk に登録されているサブスクリプションのすべてが表示されています。これは、待機中のメッセージを知る際や、そのメッセージの最終的な送信先（オーケストレーションまたは送信ポート）を判断する際に役に立ちます。

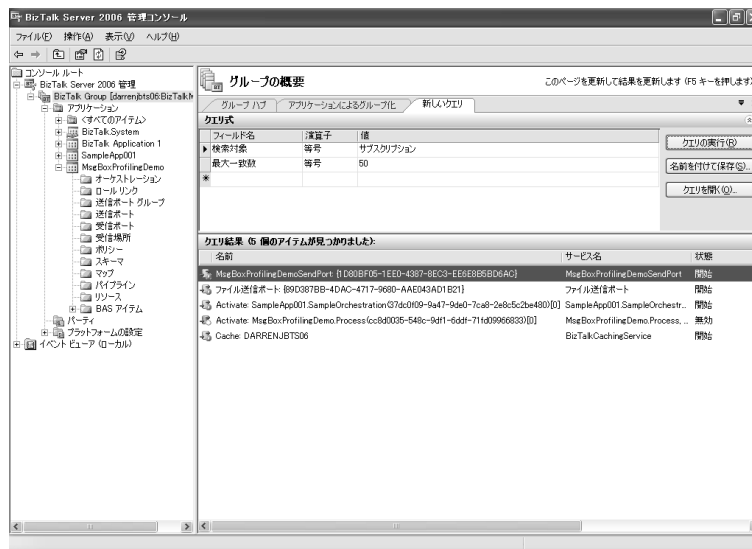


図 2-2 BizTalk Server 2006 管理コンソールのサブスクリプションクエリ機能

サブスクリプションをダブルクリックすると、サブスクリプションの正確な詳細を知ることができます。例えば、図 2-3 では、MsgBoxProfilingDemoRecvPort という名の受信ポートからのすべてのメッセージをサブスクライブする送信ポートサブスクリプションが示されています。



メモ

サブスクリプションの詳細は「第 3 章 アダプタ」、「第 5 章 オーケストレーション」、および「第 11 章 管理」で扱います。



図 2-3 サブスクリプションの詳細画面

2.7 メッセージボックス

メッセージがアダプタからパイプラインに渡されたら、BizTalk ホストインスタンスの一部として実行された BizTalk メッセージエージェントはこのメッセージをサブスクライブしたのが誰なのかを評価し、そのメッセージを BizTalk メッセージボックスにコミットします。図 2-4 にこれまで本文中で解説してきたエンジンコンセプトのすべてを示します。

BizTalk メッセージボックスは、SQL Server データベースとして実装されており、1つの BizTalk グループ内のすべての BizTalk サーバー間で共有されています。非常に負荷の高いまたは遅延が問題となるシナリオでは、複数のメッセージボックスを利用できます。ただし、複数のメッセージボックスにまたがる分散トランザクションを使用することが要求されること、およびマスタメッセージボックス上でメッセージ発行を無効にする必要があることから、一般的に、スケーラビリティを確保するためにメッセージボックスは1つから3つまでにする必要があります（これはパフォーマンスおよび拡張性のテストを行う必要があります）。

メッセージボックスは、すべての BizTalk ソリューションの中心であり、それなしでは何も機能しません。オーケストレーションの実行、退避、復元、およびメッセージ追跡に伴う受信および送信メッセージはすべて、BizTalk メッセージボックスに依存しています。

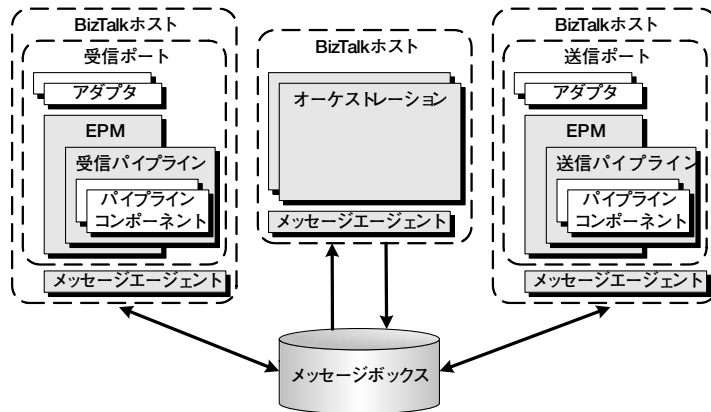


図 2-4 BizTalk エンジンコンセプト

メッセージボックスの詳細について解説する時にわかりますが、メッセージボックスは一種のパブリッシュ/サブスクライブ設計となっています。メッセージは BizTalk メッセージボックスに対して発行 (パブリッシュ) されます。特定のメッセージに対して複数のサブスクライバが存在する場合、それぞれのサブスクライバがメッセージのコピーを受信 (サブスクライブ) します。

BizTalk がメッセージを受信する場合、メッセージがアダプタおよび関連するパイプラインを通過してから、メッセージエージェントがそのメッセージを受け取ります。メッセージエージェントは、まずメッセージコンテキストから昇格したプロパティを、bts_InsertProperty ストアドプロシージャを使用して BizTalkMsgBoxDb データベース内の MessageProps テーブルに書き込みます (以上のすべてはメッセージボックスデータベースに対する単一のラウンドトリップ内で行われます)。

使用可能となる昇格したプロパティは、アダプタとパイプラインの組み合わせによって異なります。例えば、昇格したプロパティで最も一般的なものとしては、MessageType があります。これは、オーケストレーションサブスクリプションに対して使用されるメッセージタイプを提供します。

次にメッセージエージェントは、BizTalk メッセージボックスに保存されている bts_FindSubscriptions ストアドプロシージャを呼び出し、このメッセージと一致するサブスクリプションを検索します。このストアドプロシージャは Subscription テーブルおよび Predicate テーブルと、前のステップでプロパティを挿入した MessageProps テーブルとを結合して、可能性のあるサブスクライバを特定します。

サブスクリプションの、可能な演算子の組み合わせすべてに合う一連の Predicate テーブルが存在します (EqualsPredicates、ExistsPredicates、GreaterThanOrEqualPredicates、GreaterThanOrEqualPredicates、LessThanOrEqualPredicates、LessThanPredicates、および NotEqualPredicates)。例えば、以下のサブスクリプションは、 = 演算子を使用しているので、EqualsPredicates テーブルに置かれます。

```
ReceivePortName = "MsgBoxProfilingDemoRecvPort"
```

サブスクリプションの種類に応じて、前述のテーブルのいくつかを用いてサブスクリプシ

ョンを表現します。

図 2-5 には、MsgBoxProfilingDemoRecvPort という名の受信ポートによって処理されたすべてのメッセージをサブスクライブするように送信ポートが設定された時の、EqualsPredicates テーブルの内容が示されています。これは図 2-3 で示したサブスクリプションと同じです。

id	uid	uidPopID	vValue	uidPredicateGroupID
1	34	133445B0-B87A-482E-9A98-0A82CFED3896	{8CDF518D-D07A-4ECC-9604-67866A2E4D8C}	825F4A90-973C-4529-9A20-802114
2	32	133445B0-B87A-482E-9A98-0A82CFED3896	{F230A0A3-E89E-47D0-97F9-5A7A286AD829}	CEEF9C4D-1501-4A03-8887-DD9EF
3	48	A20B8C25-EF73-4C89-9866-5661E330CCE4	{1D808F05-1EE0-4387-9EC3-EE6E8858D6AC}	A74EF3FC-50DC-4296-8BFE-88EB9
4	44	A20B8C25-EF73-4C89-9866-5661E330CCE4	{89D38789-4D4C-4717-9680-AAE0434D1621}	B6CDE4C3-A971-417C-9106-D000D0
5	47	78809A3A-344E-4019-9465-89AD2AED18E5	MsgBoxProfilingDemoRecvPort	7C5F38E4-4A72-4033-9214-D08826
6	30	B8F6E8DC-7D03-47E8-AA27-879EA30CA688	CacheRefresh	90B619AE-0D41-4E20-A404-CC0A28
7	33	F4E068C3-48AE-49EC-8CCA-F83E542348B2	http://MsgBoxProfilingDemo.Message#Root	CEEF9C4D-1501-4A03-8887-DD9EF
8	35	F4E068C3-48AE-49EC-8CCA-F83E542348B2	http://SampleApp001.ReceiveSchema#Root	825F4A90-973C-4529-9A20-802114

図 2-5 EqualsPredicates テーブルの内容

次にメッセージエージェントは bts_InsertMessage ストアドプロシージャを呼び出して、BizTalk メッセージを、パートの数などメッセージに関する基本的なメタデータと共に、Spool テーブルに挿入します。このストアドプロシージャの内部で int_InsertPart ストアドプロシージャが呼び出され、メッセージの本文がそのまま、Parts テーブルに置かれます。

int_EvaluateSubscriptions ストアドプロシージャも bts_InsertMessage ストアドプロシージャから呼び出され、サブスクリプションの比較中に、メッセージに対する参照に関連するホストキューテーブルに挿入します。このテーブル名は、命名規則 <ホスト名>Q に従います。例えば、デフォルトの BizTalkServerApplication ホストの名称は、BizTalkServerApplicationQ となります。これは、現在どの作業がシステム内のどのホストによって処理待ちとなっているか分析する際に役に立ちます。詳細については本章の「2.12 ホスト」を参照してください。

マルチパートメッセージの場合、BizTalk メッセージエージェントは、そのメッセージに対する追加パート用に、再度 bts_InsertMessage ストアドプロシージャを呼び出し、続けて Parts テーブルに置かれます。

図 2-6 は、SQL Server Management Studio ツールによってキャプチャされた SQL Server Profiler によるトレースを示しています。このトレースから、メッセージが BizTalk によって受信された時にストアドプロシージャが呼び出されていることがわかります。左側のブックマークで強調表示されたトレース行は、今まで解説してきたインタラクションを示しています。

RPC:Completed	exec bt_GetDocSpecInfoById @id=N' {3F882321-A06E-4742-B3F1-9495203845E8}'
Audit Logout	
RPC:Completed	exec [dbo].[bts_InsertProperty] @uidBatchID=N' {8C6A91A9-5283-4B47-AF46-01E9A8B96184}', @nOrderID=0, @uidMessageID=N' {943675F5-6DA
RPC:Completed	exec [dbo].[bts_InsertProperty] @uidBatchID=N' {8C6A91A9-5283-4B47-AF46-01E9A8B96184}', @nOrderID=0, @uidMessageID=N' {943675F5-6DA
RPC:Completed	exec [dbo].[bts_InsertProperty] @uidBatchID=N' {8C6A91A9-5283-4B47-AF46-01E9A8B96184}', @nOrderID=0, @uidMessageID=N' {943675F5-6DA
RPC:Completed	exec [dbo].[bts_InsertProperty] @uidBatchID=N' {8C6A91A9-5283-4B47-AF46-01E9A8B96184}', @nOrderID=0, @uidMessageID=N' {943675F5-6DA
RPC:Completed	exec [dbo].[bts_FindSubscriptions] @uidBatchID=N' {8C6A91A9-5283-4B47-AF46-01E9A8B96184}'
RPC:Completed	declare @p17 int set @p17=1 exec [dbo].[bts_InsertMessage] @uidMessageID=N' {943675F5-6DAC-4B40-BE63-40CCFE1FE86B}', @uidBatch
RPC:Completed	declare @p17 int set @p17=0 exec [dbo].[bts_InsertMessage] @uidMessageID=N' {943675F5-6DAC-4B40-BE63-40CCFE1FE86B}', @uidBatch
RPC:Completed	exec [dbo].[bts_InsertTrackingData] @uidServiceID=N' {D878C2D5-80AE-4C05-866B-CCA77437D811}', @uidInstanceID=N' {FA25687F-7020-4A6
SQL:BatchStarting	IF @@TRANCOUNT > 0 COMMIT TRAN
SQL:BatchCompleted	IF @@TRANCOUNT > 0 COMMIT TRAN
RPC:Completed	declare @p4 bigint set @p4=0 exec TDDS_GetTrackingData @DestinationID=0, @PartitionID=1, @LastReadSeqNum=-1, @MaxSeqNum=@p4 out

図 2-6 SQL Server Profiler によるトレース

SQL Server Profiler によるトレースの詳細を以下に示します。InsertProperty ストアドプロシージャが、昇格したそれぞれのプロパティ用に呼び出されているということが一目瞭然です。uidPropertyID 値は、BizTalk に登録されたコンテキストプロパティを表す ID で、BizTalkMgmtDb (BizTalk 管理データベース) 内の bt_DocumentSpec テーブルに登録されています。

```

exec [dbo].[bts_InsertProperty]
@uidBatchID=N' {8C6A91A9-5283-4B47-AF46-01E9A8B96184}',
@nOrderID=0, @uidMessageID=N' {943675F5-6DAC-4B40-BE63-40CCFE1FE86B}',
@uidPropertyID=N' {70715D47-142A-41B4-8FDC-9E3DDBA78BD13}',
@vtValue=N'c:_btsport_MsgBoxProfilingDemo_Receive_*.xml'
go

exec [dbo].[bts_InsertProperty]
@uidBatchID=N' {8C6A91A9-5283-4B47-AF46-01E9A8B96184}',
@nOrderID=0, @uidMessageID=N' {943675F5-6DAC-4B40-BE63-40CCFE1FE86B}',
@uidPropertyID=N' {133445B0-B87A-482E-9A98-0A82CFED3896}',
@vtValue=N' {F2930AA3-EB9E-47DD-97F8-5A742B64DB29}'
go

exec [dbo].[bts_InsertProperty]
@uidBatchID=N' {8C6A91A9-5283-4B47-AF46-01E9A8B96184}',
@nOrderID=0, @uidMessageID=N' {943675F5-6DAC-4B40-BE63-40CCFE1FE86B}',
@uidPropertyID=N' {798D8A34-3A4E-4DD9-8AB5-99AD2AEB16E5}',
@vtValue=N'MsgBoxProfilingDemoRecvPort'
go

exec [dbo].[bts_InsertProperty]
@uidBatchID=N' {8C6A91A9-5283-4B47-AF46-01E9A8B96184}',
@nOrderID=0, @uidMessageID=N' {943675F5-6DAC-4B40-BE63-40CCFE1FE86B}',
@uidPropertyID=N' {7D0D2D40-4906-4AB0-9578-D4593F7848D7}',
@vtValue=N'FILE'
go

exec [dbo].[bts_InsertProperty]
@uidBatchID=N' {8C6A91A9-5283-4B47-AF46-01E9A8B96184}',

```

```

@nOrderID=0,@uidMessageID=N'{943675F5-6DAC-4B40-BE63-40CCFE1FE86B}',
@uidPropertyID=N'{F4E068C3-48AE-49EC-8CCA-FB3B542348B2}',
@vtValue=N'http://MsgBoxProfilingDemo.Message#Root'
go

exec [dbo].[bts_FindSubscriptions]
@uidBatchID=N'{8C6A91A9-5283-4B47-AF46-01E9A8B96184}'
go

exec [dbo].[bts_InsertMessage]
@uidMessageID=N'{943675F5-6DAC-4B40-BE63-40CCFE1FE86B}',
@uidBatchID=N'{8C6A91A9-5283-4B47-AF46-01E9A8B96184}',
@uidSubscriptionID=N'{1D80BF05-1EE0-4387-8EC3-EE6E8B5BD6AC}',
@uidPredicateGroupID=N'{AD2E63DD-42C0-46A7-88EE-50387DBDE38E}',
@uidInstanceID=NULL,
@fMessageRoutedTwice=1,
@nRefCount=1,@numParts=0,@OriginatorSID=NULL,@OriginatorPID=NULL,
@dtExpiration='1899-12-30 00:00:00:000'',
@fTrackMessage=0,@nvcMessageType=NULL,@uidPartID=NULL,@nvcPartName=NULL,
@nPartSize=0,@fSuccess=@p17 output,@imgPart=NULL,@imgPropBag=NULL,
@fPartExistsInDB=0,@imgContext=NULL
go

exec [dbo].[bts_InsertMessage]
@uidMessageID=N'{943675F5-6DAC-4B40-BE63-40CCFE1FE86B}',
@uidBatchID=N'{8C6A91A9-5283-4B47-AF46-01E9A8B96184}',
@uidSubscriptionID=NULL,@uidPredicateGroupID=NULL,@uidInstanceID=NULL,
@fMessageRoutedTwice=0,@nRefCount=0,@numParts=1,@OriginatorSID=NULL,
@OriginatorPID=N's-1-5-7',@dtExpiration='1899-12-30 00:00:00:000'',
@fTrackMessage=0,@nvcMessageType=N'http://MsgBoxProfilingDemo.Message#Root'
,
@uidPartID=N'{F90F9C67-DE14-4E80-9EDE-D50283404FEC}',@nvcPartName=N'body',
@nPartSize=149,@fSuccess=@p17 output,@imgPart=(略)
go

```

これにより、InsertProperty ストアドプロシージャを呼び出す（上の図でボールドで強調されている）際に、プロパティを表す GUID が uidPropertyID パラメータとして渡されていることが分かります。この GUID に対応するプロパティ名は ReceivePortName です。以下は、GUID からプロパティ名を解決する SQL クエリです。

```
SELECT * FROM bt_DocumentSpec WHERE id LIKE '798D8A34-3A4E-4DD9-8AB5-99AD2AEB16E5'
```

結果※²:

```
http://schemas.microsoft.com/BizTalk/2003/system-properties#ReceivePortName
```

※2 監修注：この結果は msgtype 列に格納されています。

上記の SQL Server Profiler によるトレースで分かるとおり、bts_InsertMessage ストアドプロシージャの最初の実行の際には uidSubscriptionID パラメータに SubscriptionID を渡しています。この値は、bts_FindSubscriptions ストアドプロシージャから返されたこのメッセージに対するサブスクライバを表す ID です。

渡された値は {1D80BF05-1EE0-4387-8EC3-EE6E8B5BD6AC} であり、これは、図 2-7 でも分かるとおり、送信ポートサブスクリプションと一致します。



図 2-7 送信ポートサブスクリプションで表示させた SubscriptionID

これらすべての完了により、メッセージルーティングは完結するため、この処理の最初のステップで挿入された、昇格したプロパティは削除されます。

サブスクリプションの処理は複雑であるため、デバッグ中に役立つように参考として説明してきました。要するに、いったんメッセージがメッセージエージェントに到着すると、サブスクライバの評価、メッセージの挿入、そしてホストインスタンスキューへのメッセージ参照の追加が行われ、その結果、サブスクライバがメッセージを処理する準備ができる、ということだけ覚えておけば十分です。

各ホストインスタンスは定期的にデータベースに対してポーリングを行い、キューの中に新しい作業があるかどうか調べます。複数のスレッド（内部では「デキュースレッド」と呼ばれます）がホストインスタンスによって使用され、さらに復元性と拡張性のために複数のマシンで指定されたホストインスタンスを実行することができます。

次に、これらのデキュースレッドが bts_DequeueMessages_<ホストインスタンス名> ストアドプロシージャを使用し、処理のためのメッセージを取り出します。bts_DequeueMessages を実行するたびに、設定された上限数（バッチサイズ）以内の、可能な限り多くのメッセージを取り出します。

デフォルトのバッチサイズは 20 に設定されており、サービスクラスごとに設定可能です。サービスクラスの設定に関する詳細は、「第 9 章 パフォーマンスとスケーラビリティ」で解説します。また、ホストインスタンスによる bts_DequeueMessages ストアドプロシージャの呼び出し頻度も設定できます。低所要時間シナリオでは、これによって待ち時間を短縮

することが可能です（ただし、SQL Server の負荷が増大します）。

BizTalk のアーキテクチャとそれによるメッセージボックスの使用は、メッセージをその内容に基づいてルーティングできるということを意味します。一般的にこれは、コンテンツベースのルーティング（content-based routing ; CBR）と呼ばれています。CBR は、非常に強力なコンセプトであり、BizTalk 内で非常に簡単に設定できます。受信場所を設定し、1 つ以上の送信ポートを受信メッセージのプロパティがサブスクリブされるように設定します。すると、必要となるトランスポートが使用されて、そのメッセージがリモートシステムまたはエンドポイントに送信されます。

CBR は、適切な BizTalk 使用方法の 1 つであり、カスタムパイプラインコンポーネントと組み合わせれば、あなたが抱える問題に対する解決策として、十分な答えを出すことができます。しかし、ほとんどのシナリオでは、BizTalk を活用して複雑なビジネスプロセスを実装することができるオーケストレーションを使用します。

2.8 オーケストレーション

BizTalk オーケストレーションでは、「オーケストレーションデザイナー」と呼ばれるビジュアルデザインツールを使用して、ビジネスプロセスを実装することができます。オーケストレーションデザイナーでは、複数の「シェイプ」の組み合わせによってビジネスプロセスを表現できます。シェイプはツールボックスから選択して、デザイン画面の上にドラッグします。

図 2-8 に、オーケストレーションデザイナーを示します。左側のツールボックスに使用可能なシェイプがあります。

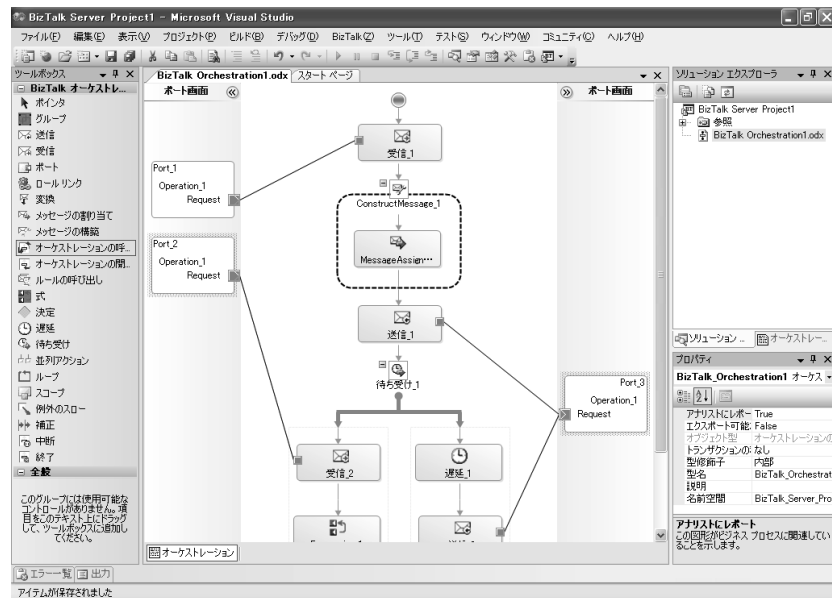


図 2-8 オーケストレーションデザイナー

デザイン画面は Visual Studio が提供するもので、開発者を対象とする設計となっています。また、Visio アドインも用意されており、ビジネスプロセスを簡素化した概念を使用して表現することができます。その結果作成された Visio ダイアグラムは、メインのオーケストレーションデザイナーにインポートして完全に実装することができます。

オーケストレーションを作成したら、それを .NET コードにコンパイルし、最終的には .NET アセンブリに変換します。オーケストレーションは .NET side by side (SxS) 機能も活用しています。これにより複数のバージョンのオーケストレーションをいつでも配置できます。

オーケストレーションの種類には、メッセージをサブスクライブするものと、他のオーケストレーションによって呼び出されるものがあります。メッセージをサブスクライブするオーケストレーションの場合、そのオーケストレーションは最上部に受信シェイプを置く必要があります。

次に開発者は、Visual Studio とオーケストレーションの種類ペインを使用して BizTalk メッセージを作成し、リクエストされているメッセージのスキーマを指定して、受信シェイプがこのメッセージを使用するように設定します。次のステップでは、オーケストレーションデザイン画面上にポートを作成して、これを受信シェイプとリンクさせます。

このプロセスによって、オーケストレーションに必要なサブスクリプションが効果的に描かれ、メッセージをオーケストレーションヘルディングすることが可能になります。

オーケストレーションエンジンは、メッセージが到着する際にオーケストレーションを作成します。オーケストレーションの実行中は、通常の場合、「ウエイト」ポイントに達します。これらのウエイトポイントは通常の場合受信シェイプです（例えば、オーケストレーションは、メッセージの到着までずっと待機するか、さもなければ遅延シェイプによって事前に定義された期間だけ待機します）。

これらの期間中メモリ内にオーケストレーションが残っていると、使用されるシステムリソースが増えすぎる原因となり、多くのシナリオではメモリ不足やリソース不足につながりかねません。例えば、スレッドプールにスレッドが入りきらないなどです。これに対処するために、オーケストレーションエンジンは、オーケストレーションを「退避」させることを決定できます。これは、メッセージボックスに対するオーケストレーションの状態をシリアライズし、その後メモリからオーケストレーションを削除するものです。オーケストレーションエンジンは、待っていたメッセージが到着した時や遅延期間の期限が切れた時に、自動的に、オーケストレーションを復元させます。

オーケストレーションの処理はステートレスです。オーケストレーションの状態はすべて、オーケストレーションインスタンス内に保存されています。つまり、オーケストレーションのインスタンスは、どの特定のマシンとも強い関係を持ちません。そのため、1つのオーケストレーションがあるマシンで退避させられ、その後、別のマシンで復元させられることもありえます。



メモ

「第 5 章 オーケストレーション」では、オーケストレーションのコンセプトのすべてを扱います。実装可能なより複雑なビジネスプロセスもいくつか紹介します。

2.9 エンタープライズシングルサインオン

SSO (Enterprise Single Sign-On ;エンタープライズシングルサインオン) は、BizTalk ソリューションにとって不可欠な要素であり、情報の回復と安全な保存を提供するものです。デフォルトでは、BizTalk は SSO を使用して、アダプタ設定や、リモートシステムにアクセスするために必要な (アダプタが取り出すことができる) ユーザー名とパスワードの安全で一極集中のストレージを提供します。

SSO を使用することによってこのような構成データを安全に保管することができます。構成データは集中保管されるので、一度に変更することが可能で、すべての BizTalk Server が起動時にこの構成データを取り出します。



メモ

SSO の詳細は「第 11 章 管理」および「第 12 章 エンドツーエンドのシナリオ」で解説します。

2.10 ビジネスアクティビティ監視

BAM (Business Activity Monitoring ;ビジネスアクティビティ監視) は BizTalk が持つ極めて有用で強力な機能です。BizTalk Server 2004 で導入され、BizTalk Server 2006 ではさらに機能が拡張されています。

BizTalk を、ビジネス上の問題を解決するために使用する際、多くの場合、そのソリューション内の処理はブラックボックスになってしまいます。そのため、ビジネスユーザーやテクニカルサポート担当者が、何が行われているか知ることは大変難しくなっています。これは BizTalk ソリューションだけに当てはまることではなく、ビジネスデータやビジネスロジックの処理を含む多くの IT プロジェクトで言えることです。このような処理を見えるようにすることは、そのようなユーザー達にとって非常に重要です。一般的に言って彼等は、処理のバックログが発生している場所とその原因を知る必要があります。また、処理中のビジネストランザクションの数と種類についても知る必要があります。

ある時点のある特定のリクエストまたはメッセージの場所を知ることは、難しいかもしれませんが、数多くの BizTalk 管理ツールが用意されていますが、これらのツールから得られる情報はきわめて技術的なものです。つまり、オーケストレーションやメッセージの情報を得ることはできますが、開発者以外の者にとっては分かりにくいのです。また、これらのツールでは、そのツールが現在処理のどの段階にあるかということを正確に知ることはまったくできません。

そこでビジネスアクティビティ監視の登場です。BAM を使用すれば、どんなメッセージングまたはビジネスプロセスについても、あらゆるデータ利用者が理解できるような形でその情報を完全に示すことができます。最初のステップで必要なのは、収集するデータの「ウィッシュリスト」です。これはビジネスマイルストーンおよびビジネスデータの観点から記述します。ビジネスマイルストーンの観点からの場合、タイムスタンプとして表現し、ビジネスデータの観点からの場合、テキスト文字列、整数、または桁数に制限のない数値のデー

タ型として表現します。

ウィッシュリストを作成したら、この計測データを保管する場所を指定する必要があります。SQL Serverはそのための理想的な場所です。トランザクション処理が可能で、かつ高い負荷に対処できる能力があるからです。

BAM管理ツールは、この計測データを高いパフォーマンスで保存するために必要なあらゆるデータベースインフラストラクチャを提供します。システムパフォーマンス全体に大きな影響を与えることなく、大量の計測データを収集することができます。高い負荷がかかるシステムに対して追跡機能を実装済みのユーザーであれば、これが解決するに値しない小さな問題ではないことを理解してもらえましょう。

ウィッシュリストとデータベースインフラストラクチャを作成したら、このインフラストラクチャにデータを格納する必要があります。大抵の場合、TPE (Tracking Profile Editor ; 追跡プロファイルエディタ) を使用して、グラフィック上のドラッグアンドドロップ操作によって、BizTalk エンジンに対してタイムスタンプとメッセージデータを収集し、ウィッシュリストとマッチングさせるように指示することができます。このプロセスによって追跡プロファイルが作成されます。これらすべては、既存のシステムに対しても、必要に応じて後から行うことができます。

これで追跡プロファイルをデータ収集の実行時に配置することができます。また、システムを少しも停止させることなく後から修正することもできます。

データをデータベースインフラストラクチャに格納したら、そのデータを当事者が利用できるようにする必要があります。BizTalk Server 2006にはBAMポータル(後に解説します)が導入されています。BAMポータルは、BAMが収集したデータにアクセスできる業界標準のWebインターフェイスを提供します。データはSQLデータベースに格納されるので、BAMポータルでは要件を満たせない場合は、当然、SQL Server Reporting Servicesなどのツールを使用して独自のポータルを作成することや、SQLクエリを使用してデータを既存のツールやアプリケーションに組み込むことが可能です。

ウィッシュリストを作成する段階でメジャーとディメンション(「第6章 ビジネスアクティビティ監視」で解説します)を定義することもできます。これらは、OLAP (Online Analytical Processing ; オンライン分析処理) キューブを提供するため、および、データウェアハウスを提供するために使用します。OLAP キューブはBAMインフラストラクチャが自動的に設定します。データウェアハウスは、何らかのビジネスインテリジェンスツールまたはExcelのピボットテーブル機能を使用して問い合わせを行うことができます。



メモ

このテーマについては「第6章 ビジネスアクティビティ監視」で詳細に解説しますが、BizTalk以外のサーバー上でBAMコンポーネントを活用でき、その結果、効果的にエンドツーエンドで企業規模の機能を提供するということは、本章においても注目すべきことです。

2.11 ルールエンジン

どんなBizTalkソリューションにも、条件付きロジックの要素があります。これらのソリューションの多くは、C#のようなプログラミング言語でビジネスロジックを効果的に実装

できます。

このビジネスロジックが変化する場合、またはそのビジネスルールの所有者が独自の観点から変更を管理する必要がある場合、ビジネスロジックをコードから分離できるルールエンジンは、非常に強力な機能になる可能性があります。

BizTalk Server 2004ではBRE (Business Rules Engine ; ビジネスルールエンジン) が導入されました。BizTalk Server 2006で大きな変更はありません。BREは標準的な前方推論ルールエンジンであり、Rete アルゴリズムを実装しています (これが実際に何を意味するかは「第7章 ビジネスルールエンジン」で扱います)。

技術者ではないビジネスユーザーは、ビジネスルール作成ツールを使用してルールを定義できます。このGUI (Graphical User Interface ; グラフィカルユーザーインターフェイス) ツール内で、ポリシーを定義できます。ポリシーとは、複数のルールをグループ化するために使用されるコンセプトです。これらのルールは何らかのデータソースの組み合わせに依存しています。有効なデータソースはXMLドキュメント、SQL Server データベーステーブルの列、または.NET クラスです。

データソースはビジネスユーザーにとって悩みの種です。なぜなら、ビジネスユーザーは、自分達がルールを評価する上で依存するデータの格納場所も、その取得方法も知らないからです。例えば、XPath ステートメントを書いているユーザーを想像してみてください。これを避けるため、BREは「ボキャブラリ」を使用します。これにより、開発者がデータにビジネスに適した名前をマッピングすることができます。ビジネスユーザーはデータの格納場所や格納方法を知らなくても、単純に論理データアイテムを自分のルールの上にドラッグするだけでよいのです。

ポリシーはバージョン化されており、ダウンタイムを発生させることなく BizTalk Server 環境に配置することができます。ポリシーはSQL Server のルールエンジンデータベースに格納されています。BizTalk Server はデータベースのラウンドトリップを避けるためにメモリ内にポリシーのキャッシュを保持します。その結果、パフォーマンスが向上します。

ルールの実行は追跡することができ、あるルールが特定のシナリオ内で使用された理由とポリシー実行の結果についての明快なトレーサビリティを実現できます。

2.12 | ホスト

BizTalk 「ホスト」は、ランタイムプロセスの論理的表現です。各ホストは受信アダプタ、送信アダプタ (およびそれらに関連する受信ポートおよび送信ポート)、オーケストレーションなどで構成されます。

BizTalk ホストを定義したら、そのインスタンスを作成できます。インスタンスを作成する時は、ホストインスタンスを実行する物理サーバーの名前と、「サービスアカウント」の名前を指定します。ホストはインプロセスでも分離プロセスでも実行することが可能です。インプロセスホストインスタンスはWindows サービスとして実行されます。BizTalk ツールは、新しいホストが作成されると自動的にこのサービスを作成します。分離ホストインスタンスは、ASP.NET ワーカープロセスなどの BizTalk 以外のプロセス内で実行されます。

1つのホストは複数のインスタンスを持つことができます。これにより、BizTalk ホストを複数のサーバー上で実行できます。さらに各 BizTalk Server はホストのどんな組み合わせ

せも実行することができます。

複数のマシン上でホストを実行することは、BizTalk Server グループ内での負荷分散という意味で重要です。パフォーマンステストに伴うよくある問題は、ソリューションがデフォルトの BizTalkServerApplication ホスト内にすべてのアダプタおよびオーケストレーションと共に配置される場合に発生します。異なる種類の処理からのリソース要求が競合するからです。

BizTalk エンジンの大部分は.NET で書かれており、それ自身はビルトインの CLR スレッドプールを利用するので、同じホスト内で SOAP アダプタとオーケストレーションを実行すると、両者はスレッドのような有限のリソースを取り合うでしょう。複数のオーケストレーションが実行され、各オーケストレーションが1つのスレッドプールスレッドを使用する場合、SOAP アダプタに送信メッセージを処理するためのスレッドを「枯渇させる」ことは簡単です。その結果、SOAP アダプタはエラーを起こし、後でメッセージをリトライします。たいていの場合、これはオーケストレーションエンジンが.NET のスレッドプールをあまりにも頻繁に使用するという事実が原因です。

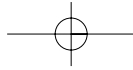
まず考えられる経験的な解決法、および一般的なベストプラクティスは、受信用、オーケストレーション用、および送信用に、それぞれ別のホストを使用することです。それぞれの BizTalk サーバー上でそれぞれのホストを実行することができます。もちろんこれは、数多くのソリューションを検討する上での出発点であるべきです。パフォーマンステストを行えば、ソリューションにとって最適なホストとホストインスタンスの組み合わせが分かるでしょう。

デフォルトでは、BizTalk には、BizTalkServerApplication と BizTalkServerIsolatedHost という、2つのホストが用意されています。BizTalkServerApplication ホストは、自分で作成するすべてのアダプタまたはオーケストレーション用、および BizTalk が生成する送信 SOAP メッセージ用のデフォルトホストです。一方、BizTalkServerIsolatedHost は、SOAP アダプタ経由で受信されたメッセージ用に使用されます。

この分割にはいくつかの相応の理由があります。受信 SOAP メッセージは、他のアダプタとは異なり、BizTalkServer エンジンではなく、IIS (Internet Information Services) によって処理されます。また、他のアダプタからのメッセージよりもはるかに信頼性が低いので、IIS から受信するのだと考えることもできるでしょう。多くの場合、受信 SOAP メッセージは、インターネットのような信頼性の低いネットワーク経由で受信されます。SOAP アダプタなど、多くのアダプタは、BizTalk NT Service 以外のプロセスによってホストされる必要があります。この分離ホストモデルは、これを可能にし、事実上アダプタが BizTalk の外にホストされる可能性があることを意味しています。このシナリオでは、アダプタは実際には BizTalk メッセージングエンジン、続いてメッセージエージェントをそのプロセスアドレス空間にロードします。これにより、アダプタが BizTalk によってホストされたものか、分離されたアダプタかに関わらず、同じプロセス外通信が要求されることが確実にになります。

この分割によって、BizTalk メッセージボックスに対する最低限のパーミッションしか持たない異なるサービスアカウントの元で分離されたホストを実行することが可能となり、これは非常に安全なソリューションとなります。もちろん、異なるサービスアカウントを持つ BizTalk ホストインスタンスをそれぞれ設定して、同じアプローチを配置全体に使用することが可能です。

追跡ホストについても言及する価値があります。これは、すべてのメッセージ追跡データを BizTalk メッセージボックスから追跡データベースに、そして「第6章 ビジネスアクテ



「イビティ監視」でも解説するように、すべてのBAMデータをBizTalkメッセージボックスからBAMPrimaryImportデータベースに移動するものです。

どのホストも追跡処理を実行できます。デフォルトでは、BizTalkServerApplicationホストが追跡ホストになるように設定されています。もちろん、必要に応じて、追跡を実行するための専用ホストを設定することも可能です。

2.13 | まとめ

本章では、架空の統合シナリオを利用して、BizTalkを使用しないでソリューションを作成することがいかに複雑な作業となるか、一方、BizTalkを使用すれば、同じソリューションをいかに簡単に作成できるかを強調してきました。このシナリオは不自然なものではありませんが、ソリューションの設計においてBizTalkを使用するかどうか決定する時に考慮すべきことが明らかになったと思います。

次に本章ではBizTalkの主要なアーキテクチャ要素を一通り紹介し、どのようにすべてのアーキテクチャ要素が組み合わされているのかを解説し、後続の章の準備を行いました。次の章ではBizTalkアダプタについて扱い、その基本コンセプトとマシン内で利用可能な各種アダプタについて詳しく解説します。

