プロジェクトの概要と全体構造

年金管理システムのITアーキテクチャ

取締役 執行役員 生産品質強化本部長
萩原高行

1．はじめに

住友信託銀行殿（以下STB殿）の従来の年金管理システムは、メインフレーム上で稼動していた。メインフレームで動いていたアプリケーションをダウンサイジングする場合は、何としてもパフォーマンス問題を解決する必要がある。特に、Windowsプラットホームで動作させる場合には、UNIXを利用する場合とは異なり、IAサーバー*1の処理性能の限界に縛られる。Webベースのオンライン処理は、Webサーバーのロードバランシング機能を利用することにより複数サーバーでの処理が可能であるが、バッチ処理については相当の工夫が必要となる。そのため、年金管理システムのアーキテクチャを検討する際に最も強く意識したのは、個別サーバーの能力に依存しない処理方法の確立であった。

実際、本プロジェクト推進の過程でも一時、バッチ処理のパフォーマンス問題がクローズアップされたことがあったが、本稿で述べるバッチ処理フレームワーク*2を用いていたためチューニングを迅速に行うことができ、結果的には、実行時間をメインフレームよりも短縮することに成功している。今回のシステム構築で採用した方法は、汎用的に利用できる手法であり、今後の大規模システム構築時にも利用可能であると確信している。

本稿では、まず年金管理システムのアーキテクチャ確定に向けての要件となる代表的な処理フローを紹介した後、設計上の基本方針とアーキテクチャ上の重要要素を説明し、最後に、本プロジェクトから得られた知見と将来に向けての展望を述べる。

2．代表的な処理フロー

年金管理システムは、年金の加入員と受給権者の加入履歴に関わる情報を管理するのが基本的な使命となる。新規加入届や脱退届など各種の適用届や、受給権者への送金先変更などの指図書が、システムの入力トランザクションとなる。データの発生源は企業や受給権者本人だが、基金等の委託者が取りまとめてトランザクションを発生させる。

本システムでは、委託者からのそれらのトランザクションについて、1件単位の入力とExcelによる一括入力をサポートしている。基本的な処理は、それらの入力の適正性をオンラインでチェックし、加入員情報や受給権者情報を保存している業務DBをバッチで更新する仕組みとなる（次ページ図1参照）。

3．基本方針

3.1　前提事項

今回、年金管理システムを構築するに当たっては、STB殿のご指定により、.NET Framework（バージョン1.0）を利用することとした。記述言語は、開発を担当するCACから提案することになっていたため、Visual Basic.NET（以下VB.NET）を採用した。OSおよびDBMSは、ご指定によりWindows 2000 ServerとSQL Server

＊1）Intel Architectureサーバーの略で、PCサーバーとも呼ばれる。

＊2）フレームワークは、アプリケーションの枠組みの規定を行うものであり、本システムでは、フレームワークはスーパークラスの集合で構成している。通常のアプリケーションはフレームワークを構成するクラスのサブクラスを利用して実装することになる。

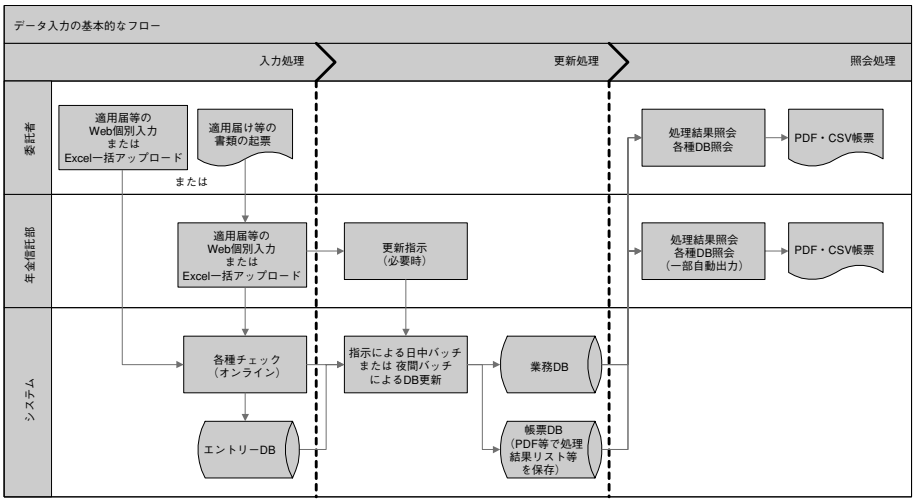

図1　データ入力の基本的なフロー

2000を利用した。当初、バッチ処理の部分についてはCOBOLを利用する方針を立てたが、オンライン処理とバッチ処理共通の機能の実現に難があったため、VB.NETで全面的に記述することとした。

3.2　アーキテクチャ考案時のポイント

年金管理システムは、以下のような業務上の特徴を有している。

・多数の対象者の履歴レコードを対象とした大量のバッチ処理を行う必要がある
・繁忙期はあるが、オンラインの同時ユーザー数は2桁程度でトランザクション量は比較的小さい
・届書、指図書が代表的なトランザクションとなるため、1トランザクションのデータ量、処理量は大きい
・基金や企業等の委託者をまたがる処理はなく、1委託者あたりの対象者は数万人以下（例外あり）

IAサーバーを用いたメインフレームからのダウンサイジングはSTB殿の既定方針である。その方針に沿った選択肢の1つとして、1台のサーバーの能力に頼るスケールアップ型のアーキテクチャがあるが、大量のバッチ処理で実用に耐えるシステムの構築には、この選択はリスクが高すぎると我々は判断した。万が一、十分なパフォーマンスが得られなかった場合に、CPUやメモリーの追加をしたあげく、最高のハードウェアでも要件に達しなければ、その時点でプロジェクトは失敗となる。したがって、「バッチ処理を減らす」か「バッチ処理のスケールアウトモデルを確立する」か、あるいはその他の方法で現実解を見出す必要があった。

一方、1トランザクションのデータ量が大きいことと処理が重いことから、オンラインのレスポンスにも悲観的な予想があった。加えて、入力帳票のデータ量が多く、しかも複雑なため、画面によっては複数入力者によるベリファイ*3を実現する必要があり、また入力帳票の入力順とバッチでの処理順序が異なるケースがあった。これらの条件により、オンライン処理時に直接データベース更新まで行うのは無理があることがわかった。結果として、何らかの適切な方法で入力帳票を一時的にプールする仕組みを構築することとした。

3.3　コアとなる処理方式とその基本方針

以上の検討を経て、年金管理システムのコアとなる処理方式として、まず次の2点について方針を策定した。

・バッチ処理フレームワーク
・エントリーDB

上記の2方式でInputとProcessの部分はカバーできるが、Outputの部分についても統一的な管理が望ましいと考え、さらに運用時、特に障害対応時の耐性を考慮し、次の2点についても方針を策定した。

・帳票処理フレームワーク
・DBバックアップ方式

1）バッチ処理フレームワーク

狙いは、バッチ処理をその処理量に応じたサーバー台数の追加でまかなえるバッチ処理方式の確立である。今回の対象業務には委託者をまたがる処理が必要なかったため、この特性を活かし、各バッチを委託者単位に分割、複数のバッチサーバーで処理することとした。サーバーの割り当て処理を行うためのツールにはMSMQを利用し、MSMQに与える電文はXMLで表現することとした。

2）エントリーDB

狙いは、複数種類のオンライン入力帳票を統一的に管理

＊3）本稿では確認再入力機能のことを指す。

できるオンライン処理方式の確立である。入力帳票をXML表現のテキストに変換し、適当なキーを付与してエントリーDB（特性によって全部で6種のテーブルにマッピングされる）に保管する。これをオンバッチや夜間バッチで適当な単位で取り出して入力帳票を再現して利用することとした。

3）帳票処理フレームワーク

狙いは、複数種類の出力帳票（ホスト転送ファイル、CSV帳票、PDF帳票等）を統一的に管理できる帳票管理方式の確立である。出力帳票を他システムから受け取るファイル等とあわせてSQLサーバーにバイナリデータとして格納することにし、そのために帳票管理DBを作成した。

4）DBバックアップ方式

狙いは、夜間バッチ処理障害時に適当な時点へ迅速に回復できるDBバックアップ方式の確立である。SANのスナップショット機能を利用し、夜間バッチ中の開始時点、開局前時点等複数のチェックポイントに戻せるようにした。

その他、セキュリティを確保するための監査証跡記録機能やWeb画面を効率的に作成するためのフレームワーク機能の実現などについても検討を進めた。それらについては本誌別稿で取り上げられているので、そちらを参照されたい。

3.4　実装方式策定時のポイント

Microsoftは、.NET Framework を、「次世代のソフトウェア アプリケーションと XML Web サービスを構築および実行するための Windows コンポーネント」*4としている。我々は、.NET Frameworkを採用した以上、XML Webサービスの時代を強く意識すべきだと考えた。本システムでは、XML Webサービスの実装は要件上必要ではなかったが、将来の移行に備え、基本的にデータ表現にはXMLを利用することとし、重要なデータ要素についてはXSD*5による定義を行うこととした。

もはや当然のこととして触れられることも少ないが、.NET Frameworkはプログラミングパラダイムとしてオブジェクト指向が採用されている。今回のプロジェクトメンバーの大半はホスト時代から年金業務に関わってきたベテランであったこともあり慎重論が強かったが、.NET Framework本来のメリットを引き出すため、内部設計以降については全面的にオブジェクト指向を採用した。

アプリケーションの作成という観点から見ると、オブジェクト指向の採用には良い面と悪い面がある。良い面としては、十分に検討された設計に基づいた実装を行えばコードが共有されて量を削減できる。逆に、依存するコンポーネントが増えてしまうと共用コンポーネントの変更時のテストが難しい。また、年金管理の業務では委託者ごとに微妙なカスタマイズが必要なケースが多く、COBOLによる実装の時代には類似コードをコピーして修正したバッチを作成するケースが多かったが、同様の手法を採ることは難しい。そのため、設計時にどの程度のカスタマイズに対応できるように考慮しておくかが重要となるが、それには業務に関する精緻な考察が必要となり、設計者への負担が大きくなる傾向があった。

4．オンライン処理アーキテクチャ（エントリーDB）

年金管理システムのオンライン処理は、すべてASP.NETによるWeb実装である。

セキュリティを確保するためにRSAのSecurIDを利用しており、SecurID認証のリクエストは、STB殿標準のISAPI*6フィルタを利用してログオン画面にリダイレクトさせている。

ユーザー（基金等の委託者）は、住友信託銀行のホームページ経由でまず年金管理システムのログオン画面（次ページ図3、図2の①）でユーザーIDを入力した上で、SecurIDのワンタイムパスワードを投入し、ログオンする。この処理の実施時に、RSA SecurIDの認証サーバーであるACE/Serverにワンタイムパスワードの確認を行い、Cookieを発行する（図2の②）。このCookieがなければISAPIフィルタでリダイレクトされ、セキュリティが確保されることになる。

ログオン認証をパスし、入力画面（図4）に至った後は、画面入力を行い、実行ボタンを押下することで、サーバー（IIS）に入力データを送信する（図2の③）。

入力データを受け取ったIISのプログラムは、当該データをXMLフォーマットに変換した上で、エントリーDBに保管する（図2の④）。入力帳票は約400あるが、XMLフォーマットに変更することで差異が吸収できるため、基本的に1つのテーブルに収めることができる。これにより、画面ごとに別個の収納テーブルを用意する必要がなくなる

＊4）Microsoftのホームページによる。

＊5）XMLスキーマ定義言語（XML Schema Definition Language）の略。XSDを定義することにより、XMLに制約を与えることができ、定義されたSchemaに準拠しているかどうかをチェックすることも可能になる。XSDを利用することにより、不要なタグや数値項目に英字が含まれていないかといったチェックが合理化できる。

＊6）Internet Server API の略。本システムでは、Webサーバーでアプリケーションが呼び出される前にセキュリティチェック機能を実装するために用いられている。

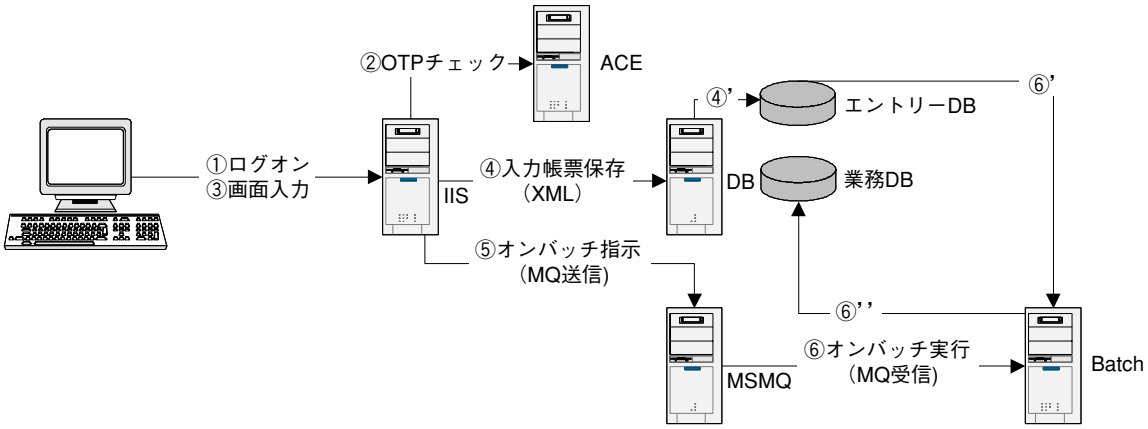

図2　オンライン処理アーキテクチャ

図3　ログオン画面

図5　Excelアップロード画面

図4　入力画面の一例

とともに処理の一元化を図ることも可能となった。

入力画面の中には即時に処理を実行する必要のあるものもあり、そのような画面では後述のバッチフレームワークを応用したオンバッチの機能を利用して後継処理を行っている（図2の⑤以降）。

委託者からの入力リクエストには繰り返し入力を必要とするものがあり、その実現のためにはExcelを利用している。ただし、各委託者で異なるバージョンのExcelが利用されていることを考慮し、Excelのファイルをそのままアップロードするのではなく、Excelマクロを用いてXMLファイルを生成した後にアップロードする方式を採用した。ユーザーは、Excelアップロードの画面（図5）でXMLファイルをアップロードしてデータを投入するが、投入後のデータ処理は通常の入力画面とまったく同様となり、データはエントリーDBに保存されることとなる。

なお、将来Excel 2003が十分に普及した後には、XML Webサービスによる実装によりExcelからの直接入力も可能となる。現在アップロード機能で利用されているXMLファイルの中には、トランザクション種別の情報とデータの双方が含まれており、機能上Webサービスによる実装は容易である。ただし、現在の実装ではワンタイムパスワードを利用しているため、認証のために何らかの対応が必要となる。クライアントサイド証明書を発行してISAPIフィルタをバイパスさせる、XML電文の中に電子署名を含める、Passport認証に依存して実装するといったいくつかの方法が考えられるが、いずれのケースでも利用者の協力を

必要とするため、技術的な観点のみから決定することはできない。

Excelアップロード以外のすべての画面でも入力トランザクションには対応するXMLスキーマを定義しており、将来はInfoPath等を用いた入力や、委託者側の人事システムからWebサービスを利用したシステム間連携が実現できる拡張性を確保してある。

5．バッチ処理アーキテクチャ（バッチ処理フレームワーク）

年金管理システムのバッチ処理は、STB殿の標準ツールであるJP1を利用して管理している（図6の①）。ただし、JP1が管理しているのは呼び出しの部分だけである。すべてのアプリケーションプログラムは、VB.NETで作成したバッチ生成.exeにバッチIDを引数として渡して起動している。

バッチ生成.exeは、当該バッチの委託者ごとの分割処理を制御するキーコンポーネントだが、その役割は単純である。別稿で詳説されているが、引数として渡されたバッチIDをキーにしてアセンブリを検索し、当該バッチクラスを生成した上で、そのメンバーメソッドである「対象委託者取得」を呼び出す（図6の②）。そして自分自身の情報を管理用テーブルに登録（図6の③）した上で、この関数の戻り値に基づき、必要な数だけ、MQのメッセージを送信することだけがその役割である（図6の④）。

バッチ生成と並んでバッチ処理フレームワークを構成するコアコンポーネントは「バッチ制御」というサービスである。このバッチ制御には、オンバッチをつかさどる日中バッチ制御と夜間バッチをつかさどる夜間バッチ制御がある。バッチ制御サービスは、基本的には単純ループを構成しており、MSMQからメッセージを受け取り、メッセージの内容に基づき委託者キーで分割された子バッチを実行し、実行が終わったらまたMSMQのメッセージを受け取りに行くというものである（図6の⑤）。

厳密には、バッチ制御は、すべての子バッチの実行状況をモニタリング（図6の⑦）して、すべてのバッチが正常に終わったかどうかについてJP1に戻り値を返している。もちろん、バッチ制御サービスも子バッチが終わるたびに結果を管理用のDBであるバッチ管理DBに登録している（図6の⑥）。

2004年1月の時点では、バッチサーバーを10台並列で動作させているが、並列度に応じたパフォーマンス向上が実現できている。委託者ごとに処理対象者の人数が大きく異なるため、非常に処理対象の大きい子バッチの実行時間が、ジョブの実行時間と一致する。

バッチ処理が何らかの理由で異常終了した場合の対応は重要な問題である。JP1のジョブとしても複数のバッチが並列して実行されるため、ジョブごとにDBのバックアップやリカバリポイントを設定することは現実的でない。また、単一のジョブについても委託者ごとの分割を行って複数の子バッチとして実行するため、ある子バッチでは正常終了しているが、別の子バッチでは異常終了しているといった状況にも対応できる必要がある。

本システムでは、できる限りリカバリ処理を単純化するために子バッチは全て1トランザクションとして扱い、異常終了した場合はその更新が全てロールバックされるように設計した。通常、バッチ処理をトランザクションとして実行することは困難であるが、今回のフレームワークでは個別のバッチを細分化し多くの子バッチが1分以内に終了するため実現が可能となった。この仕組みにより、ある委託者に対してはバッチ処理が成功してジョブが完了した状態になり、異常終了した委託者に対しては、バッチが実行されなかったのと同様の状態になる。その上で、リカバリのために、完了済みの子バッチと未完了の子バッチを分析

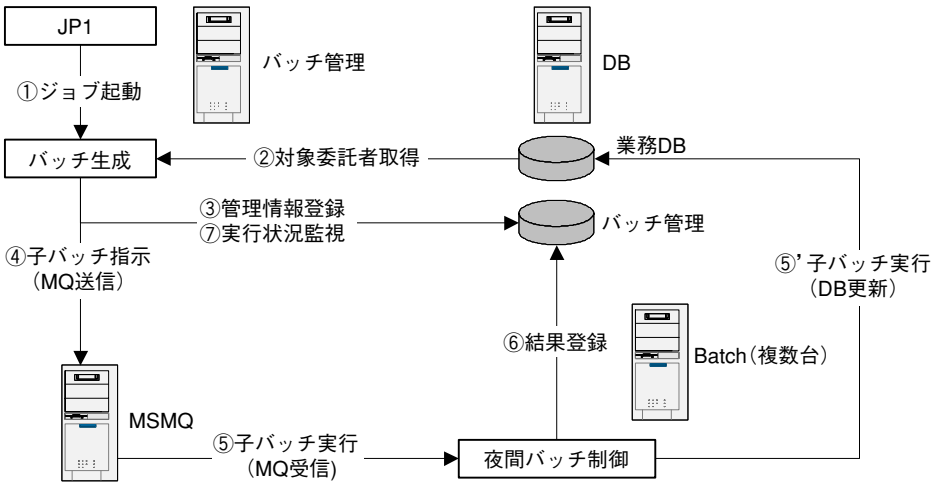

図6　バッチ処理アーキテクチャ概要

し、未完了の子バッチのみを再実行する機能を実装した。この機能を用いることにより、異常終了の原因となるデータの除去を行うか、プログラムのバグを修正したものをリリースした上で、失敗した委託者のみの再実行を行ってリカバリ処理を行うことができるようにした。

また、数ヵ月に1度と発生頻度は低いが、処理のタイミング等に起因してテーブルのロックエスカレーションによるデッドロックが発生して子バッチが異常終了するケースがある。このようなケースでも、単に再実行を行うことにより問題を解決することができる。

6．印刷処理アーキテクチャ（帳票管理フレームワーク）

年金管理システムは、PDF、CSVその他の帳票ファイルを生成する。紙に出力して、お客様にお送りするものもあれば、EUC（End User Computing）用途で電子媒体のままお送りするものもある。また、法的に保管が義務付けられるものもあるため、システム情報の扱いについて検討を行い、事前印刷帳票等特殊なものを除き、最終的に顧客に提供される帳票のイメージをファイルとして保存できる仕組みを実現することとした。

システムの管理情報は基本的に全てDBに納められているが、これらの大量の帳票出力情報には、Windowsのファイルとしての管理が必要となる。そのため、再出力や訂正版の作成などDB上の管理情報と物理ファイルの対応付けが運用時に向けた懸念事項であった。

今回のシステム実装では、何種類かの実装方法について検討した上で、最終的にDBMS内の管理テーブルに生成したファイルをそのまま収める方式を採用することとした。この方法をとることにより、管理情報と物理ファイルを完全に対応させることができるようになる。また、PDF、CSV等、オンラインで照会でき、取得できるファイルをWindowsのファイルシステム上におく必要がなくなるため、Webサーバー上のファイルシステムのアクセス制御についても考慮が不要となる。ASP.NETのアプリケーションで権限をコントロールした上で、DBMSからファイル情報を取り出して、ストリームとしてクライアントに提供している。

もう一つ、管理上重要な課題となったのは、社内利用者のオンバッチで出力する帳票の扱いである。利用者ごとにプリンタは身近なものを設定し、オンバッチが終了し次第、出力をする必要がある一方で、履歴を残すという観点から出力ファイルはPDFで作成してDBMSに保存しなければならない。ライセンス上、サーバーからPDFをそのままプリントアウトできないため、中継して処理するPCと管理アプリケーションを作成して対応することとした。Adobe Acrobatがライセンス制約上サーバーで動作させることができないため、専用クライアントPC（ダイレクトプリントPC ＝ DPPC）を準備して、MSMQを利用したプログラムを作成して対応した。

実際の作成と印刷の流れとしては、実行指示に基づいてバッチサーバー上で出力ファイルを一時的に作成（図7の①、②）した後に、帳票DBに保存（図7の③）、もしその後のプリントが必要な場合は、ダイレクトプリント用のMSMQに対してメッセージを送信する（図7の④）。

一方、DPPCは非同期にダイレクトプリント用のMQを監視しており、メッセージの到着を検知（図7の⑤）したらそのメッセージ内容（対象データと対象プリンタ）に従って、帳票データ（PDF）をDBから取得（図7の⑥）し

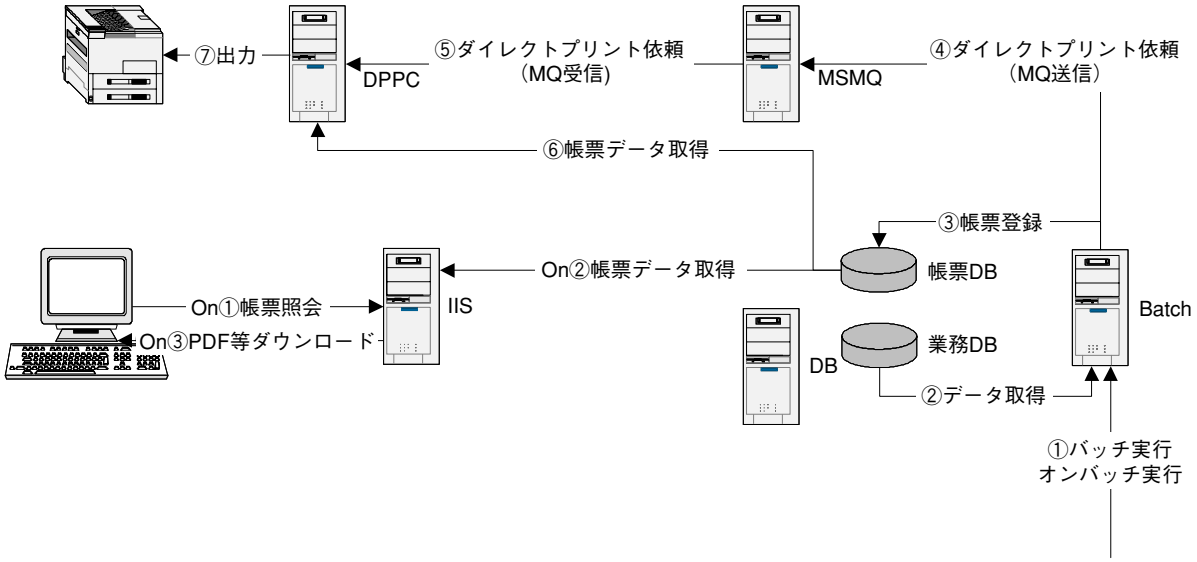

図7　帳票管理アーキテクチャ概要

てプリンタに出力する（図7の⑦）。

ダイレクト出力の対象であるか否かに関わらず、適切な権限を有する利用者は帳票照会機能を利用（図7のOn①）して、帳票DBの内容を取得、ダウンロードすることが可能である（図7のOn②、On③）。

なお、PDFファイルの作成には、ウイングアーク テクノロジーズ（株）*7のSuper Visual Formade（以下SVF）を利用した。当初は複数のバッチサーバーから1台のSVF用のサーバーを利用してPDF作成を行っていたが、パフォーマンス向上の目的で最終的には各バッチサーバーにSVFをインストールして対応した。

年金管理システムの実運用では、帳票周りのリカバリ処理でしばしば苦労を経験した。次章でも触れるが、データ不整合が発見された場合には、当日のバッチの頭に戻って再実行を行える仕組みを作成したが、出てしまった帳票はなくなる訳ではない。DBは完全にロールバックされているため、再実行すると同じIDで（データ修正等のために）結果の異なった帳票が出力されることになる。そのため、既出力帳票の廃棄運用には十分注意する必要があった。また、PDF自身はDB内にあるものの、バグ等の原因で内容は正しくない状態になっているケースがある場合も考慮が難しい。修正後の帳票を顧客に再配布した場合に、間違いがある前の帳票と新しい帳票をどう管理するかが課題となった。基本的には正式版のみを管理すれば良いが、顧客から、訂正前版で問い合わせを受けた場合への考慮が必要となる。現在は運用側でこの問題をカバーしており、業務上の不具合は発生していないが、システムとしての扱いについては、今後さらに改善の余地がある。

外字ファイルの配布ミスなどがあると、せっかく帳票ファイルが保存されていても中身が正しくないケースが発生する。この場合はリプリント処理を行っても意味がない。当初は、レンダリング前のデータについてもXML化して保管することを考えていたが、パフォーマンスの観点から見送ったため、現時点ではレンダリング時の不具合を回避するためにはジョブの再実行を行うしか選択肢がない。実際の作成時から時間が経過した後の再実行はコストの高いデータのリカバリ処理を必要とするため、そのようなケースを最少化するための拡張が望ましいと考えている。

7．DB管理アーキテクチャ

繰り返しになるが、年金管理システムはバッチ処理が重く、万が一障害が発生したときのリカバリに時間がかかると翌日のオンラインに影響が出る懸念がある。実運用では、よほど特別なイベントがない限り00:00前に終了しているが、設計時に最も心配したのはバッチのパフォーマンスと障害時のリカバリ対策であった。DBのリカバリ処理を短縮するためにSAN（Storage Area Network）のDDR（Dynamic Data Replication）技術を用いている。高速ディスクとしてもSANは非常に魅力的であるが、運用面ではスナップショット機能の利用は大きなポイントとなる。

本システムでは、夜間バッチ開始時やオンライン開始時など、重要と思われるポイントでスナップショットを取得しておき、必要があればその時点に短時間で戻せるように構成している。また、取得したスナップショットを他のサーバーで読み出せるようにしてDBのイメージをそのまま利用することもできるため、前日のバッチ更新終了後のイメージを汎用検索サーバーとして有権限者に公開する用途にも利用できている。テープへのバックアップについても表側のディスクで処理を行っている最中にスナップ

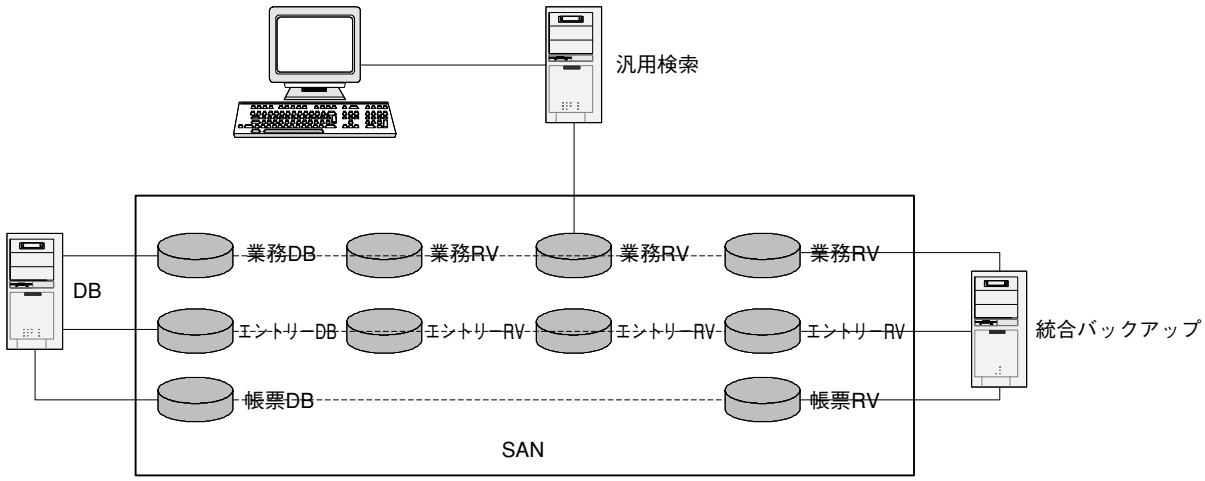

図8　DB管理アーキテクチャ概要

*7）翼システム株式会社情報企画事業部より、2004年３月24日をもって分社、設立された。

ショットのイメージから取得することができ、全体のバッチ処理時間の短縮に貢献している。

DB管理およびSANの応用に関しては、本誌別稿「高信頼性と高速バックアップを実現するSANの活用」で詳細を説明しているのでそちらを参照されたい。

8．実装後に判明した問題点と反省点

今回のシステムの実装に当たり、クラスの実装にはWeb層、ビジネスロジック層、DBアクセス層など複数の階層からなる構造を導入している。階層分けの方針は明示して設計を行ったが、オブジェクト指向実装の良さを十分に享受できたかという観点では悩ましい部分が少なからずあった。アプリケーションの観点から見た設計の良否についても議論の余地があるが、ここではアプリケーションを支えるフレームワークとしての配慮に絞って反省点をまとめておく。一度作成してしまったシステムのフレームワークの見直しは事実上不可能であるから、大規模なリライトを行わない限り本システムでは採用できないが、今後類似のシステムでフレームワークを設計する際の参考にしていただきたい。

8.1　アプリケーション特性の吸収

年金管理業務では、履歴の管理が重要なキーワードとなる。入社や昇給、休職や退職など年金に関わるイベントを加入者一人ひとりに対して管理していくシステムなので、履歴の管理はアプリケーションの重要な機能そのものと言っても良い。当然、アプリケーション設計の段階で様々な履歴管理の機能がDB設計にも機能設計にも含まれている。

アーキテクチャ設計（フレームワーク策定）の段階では、これらの履歴管理の機能はアプリケーションで考慮されるべきものとしてフレームワーク上の配慮は行わなかった。しかし、実運用の段階になって、トランザクションの誤投入などに起因してデータの修正が発生するケース、およびプログラムのバグなどによる過去日付での帳票の出し直しが必要となるケースの2種類のリカバリ処理への配慮が必要であることが判明した。データの修正についてはアプリケーション機能として実現されているが、残念ながら履歴レコードが適切に管理されていても任意の過去日付でデータを取り出しに行く機能は実現されていない。

データアクセスのコンポーネントに適切なパターンを利用しておけば汎用的に対応できたので、運用時の利用イメージの分析が十分でなかったことが悔やまれる。データベースとアプリケーションのマッピングの問題、特に履歴に関する制御は、年金管理システムだけでなく多くの事務処理システムで問題となる機能であり、フレームワークの考案時に十分な検討を行う必要があると思われた。

8.2　実装時のADO.NETの多用

今回のシステムでは、DBアクセスの部分だけでなく多くのコンポーネントでADO.NETのDatasetを多用した。ADO.NETは開発生産性を向上させる非常に強力な機能だが、Datasetを利用するとどうしても関連するDB（テーブル）の物理実装が透けて見えてしまう。このため、十分に考慮しておかないと、注意してDBアクセスコンポーネントを介してDBアクセスしていたとしても、個々のアプリケーションのクラスにデータストアの構造情報が漏れ出してしまう。一度、その構造情報を前提にしたプログラムが書かれてしまえばデータストアの実装を変更することは難しくなってしまうので、可能であれば、アプリケーションで利用するデータ表現はDatasetとは別にデータの論理構造に基づいたクラスを定義することを推奨する。

今回、入力帳票の内部表現にもDatasetを利用し、繰り返し項目の表現にもDBテーブル同様の構造を持たせた実装を行ったが、入力帳票の論理構造を表わすためには必ずしもDatasetを用いる必要はなく、むしろ型が明確に定義できる通常のクラスを利用した方が後日のバグ修正やメンテナンスをより容易にしたのではないかと考えている。

8.3　メモリー空間の不足

バッチ処理をトランザクションとして実装する際に、同時に読み込めるサイズであればADO.NETを利用して処理対象となるレコードを全部取り込んでしまうようにした。ほとんどのプログラムについては、分割後の対象レコードのサイズが限られるため、一度読み込んでしまえば後のデータベースアクセスもなく、プログラムもシンプルになり処理も高速化した。しかし、残念ながら32ビットのメモリー空間は既に十分広いとは言えず、物理メモリーは余分に積めてもデータの取り回しの空間が不足するという事態が発生した。いったん、メモリー空間内に全データが取り込めないと判明した場合は、バッチを委託者単位より細かく分割することで対処するか、処理ループの中でDBアクセスを繰り返す必要が生じてしまう。32ビット空間に入りきるか入らないかで大きくアプリケーションの構造が変わってしまうのが難点であった。64ビットに対応した.NET Frameworkがリリースされればこの問題は解決すると思われるので、プラットホームの進化に期待したい。

9．他システムへの応用と今後の課題

今回のシステム構築経験で得られたノウハウおよび実際のクラスの一部は、これから開発する多くの事務処理システムに応用が可能である。4つのコア要素である「バッチ

処理フレームワーク」、「エントリーDB」、「帳票処理フレームワーク」、「DBバックアップ方式」はいずれも広範に適用可能な技術である。

9.1 バッチ処理フレームワークの今後

現在のバッチ処理フレームワークは、年金管理システムのアプリケーション特性を直接実装しているため、委託者という並列実行単位に縛られた実装になっている。しかし、オブジェクトのインタフェースの切り出しを行えば、年金管理システムに固有な部分を切り離すことが可能で、任意の並列実行のキーを与えられるようになる。現在の処理対象委託者を取得するメソッドを任意の並列化キー取得メソッドに置き換えることで任意のバッチ処理に応用できる。

また、並列化キーと共に、対象となるDBへのコネクションストリングを渡す仕様に拡張すれば、DBサーバーも単一の大型サーバーを利用しなくてはならないわけではない。例えば処理対象のレコードがさらに膨大となった場合には、DBサーバーも複数台立てて再配置を行うことも可能になる。

さらに将来の拡張としては、バッチキューの優先制御機能の追加を考慮すべきと考えている。現在は、オンバッチの更新順序を保証するためにオンバッチは1台のバッチサーバーで処理するように実装しているが、実際には、委託者が異なるために更新が衝突せずに並列実行できるケースが大半である。実行時の競合回避のアルゴリズムを確立すれば、さらなる効率化も可能となる。

9.2 エントリーDBの今後

エントリーDBで採用した入力トランザクションをXMLフォーマットで統一的に扱うことができるようにするという手法は、XML Webサービスの時代にも応用が効く技術である。しかし、オンラインの処理が一部非同期化してしまうため、場合によっては直感と異なる動作に違和感を覚えるケースもある。DBレコードの世代管理を行い、関連レコードの照会が発生した場合には、オンバッチの処理を待つ（滞留している関連オンバッチを優先的に処理してしまう）といったフレームワーク上のサポートが必要と考える。

9.3 帳票管理フレームワークの今後

帳票によっては数十MBの大きさのファイルでもSQLサーバー上のレコードの一部として自然に扱うことができることは十分に実証できたので、運用時の負荷を軽減するツールとして他システムでも十分に利用できると考えている。反面、フレームワークを実現するクラスの実装は、PDFやCSVといった出力ファイルのフォーマットからの独立性がまだ十分ではなく、改善の余地がある。

またOffice 2003以降であれば、OfficeファイルをXML形式で扱うことができるため、PDFを利用しなくても高度なレイアウトの出力ファイルをXML形式で生成することも可能となる。あらかじめExcelでレイアウトを作ってXMLファイルとして保存したものをサーバーにテンプレートとして置いておき、データ部分だけをプログラムから埋め込む形にすれば、帳票作成時の柔軟性とパフォーマンス向上が期待できる。

9.4 DBバックアップ方式の今後

頻繁なスナップショット取得は、ディスクの同期待ちリスクも有している。今回の実装では夜間4回の同期ポイントを設けたが、やや過剰であったかもしれないと考えている。デバイスの速度向上やコストとの兼ね合いで、都度設計を見直す必要はあるが、他のシステムでも同様の実装を検討することにより対障害性を向上させることができる。

10. おわりに

年金管理システムは、VB.NETのコードライン数で300万行を超える、.NET Frameworkのアプリケーションとしては大きなアプリケーションである。これだけの大きさのアプリケーションでアーキテクチャ設計を誤れば、多大な開発工数が失われるだけでなく、プロジェクトそのものが失敗するリスクも有していた。実際、ベテランのアプリケーションエンジニアからは委託者単位でバッチが並列化できるわけなどないといった声もあり、その効果が理解されるまでには少なからぬ時間を要している。結果として、プロジェクトは無事カットオーバーでき、反省点は多々あるもののアーキテクチャ設計に関わったメンバーはまずは一息という心境である。

末筆ながら、STB殿関係者の皆様には、今回の大規模開発の機会を頂いたことに深く感謝申し上げたい。また、新しいプラットホームだけに多くの局面で問題が発生したが、Microsoft社の手厚いサポートの結果、最後まで到達できたことについても感謝したい。