# Micro Focus Net ExpressR

# 分散コンピューティング

**第 5 版**
**2003 年 5 月**

---

**このマニュアルでは、Web サービス、Java、COM オブジェクト、および Microsoft Transaction Server で実装される分散オブジェクトと Net Express の連携について説明します。**

# 第 1 章 : はじめに

この章では、分散コンピューティングの概念と仕組みを紹介し、Net Express における分散コンピューティングへの対応について説明します。

## 概要

Net Express では、COBOL アプリケーションを Web サービスや COM、EJB といった COBOL 以外の技術と連携させ、分散型システムを開発するための幅広いツールが提供されます。

このマニュアルは、これらの技術ごとに構成が分かれており、 Web サービス、Java、COM、XML といった個々の技術に、それぞれ 1 つの部があてられています。

Net Express で利用できる分散コンピューティング用の各種ツールは、COBOL アプリケーションがすでに完成していることを前提にして機能します。これらのツールは COBOL アプリケーションのフロントエンドやインターフェイスを作成し、クライアントソフトウェアからの呼び出しを可能にします。 Net Express には、次の 3 種類の分散コンピューティング用ツールがあります。

- Interface Mapping Toolkit
- クラスウィザードとメソッドウィザード (および専用の COBOLBean)
- インターネットアプリケーションウィザード

Net Express のマニュアルには、COBOL アプリケーションと連携するクライアントプログラムの開発方法も記載されています。

## 分散コンピューティングとは？

分散コンピューティング環境では、1 つのアプリケーションから他のアプリケーションが呼び出されます。これらのアプリケーションは、それぞれ別にロードされ、実行されます。 呼び出されるアプリケーションはサーバまたはサービスと呼ばれ、呼び出し側のアプリケーションはクライアントと呼ばれます。 これら 2 つのアプリケーションは多くの場合、互いに異なるマシンで動作し、開発言語や動作プラットフォームが異なる場合もあります。また、呼び出しはネットワークやイントラネット、インターネットを介して実行されます。

通常、クライアントはユーザ入力を受け入れ、データをユーザに返します。一方、サーバはデータが格納されているデータベースへのアクセスと更新を行います。クライアントのインスタンスは複数実行される場合が一般的で、それぞれのインスタンスがオフィス内、あるいは世界中のコンピュータで使用されているユーザインターフェイスに該当します。クライアントアプリケーションに複数の種類が存在し、それぞれデータを異なる形式でユーザに提示する場合もあります。

クライアントとサーバは通常、同じ組織内で 1 つのまとまりとして設計、開発されますが さまざまなクライアントで使用できる汎用的なサーバアプリケーションを開発するための技術もあります。 そのような技術に基づくサーバアプリケーションはメモリに常駐し、このサーバのイン

ターフェイスに対応し、適切なアクセス権を持つあらゆるクライアントからアクセスできます。そのようなサーバは「コンポーネント」、「再利用可能ソフトウェア」、「サービス」などの名前で呼ばれます。

このクライアント/サーバモデルの利点の 1 つは、クライアントとサーバをそれぞれ別に開発し、ディプロイできるという点です。 サーバはアプリケーションサーバとしてディプロイします。アプリケーションサーバとは、クライアントから送信される要求を待機して受け付けるソフトウェアのことです (COM の場合には、Windows 自体がアプリケーションサーバとして機能します)。したがって、いったんサーバがディプロイされれば、それを使用するクライアントを作成できます。

このクライアント/サーバモデルのもう 1 つの利点は、クライアントからアプリケーションサーバに送信される要求の数が増加しても、ロードするサーバコンポーネントのコピー数を、それに応じて増やすことができるという点です。このような柔軟性は「スケーラビリティ」と呼ばれます。 アプリケーションサーバは、セキュリティなどその他のニーズにも効果的に対応できます。

# サーバ、エンタープライズサーバ、およびアプリケーションサーバ

サービスやサーバアプリケーションは、さまざまな種類のソフトウェア環境 (Web サーバなど) で実行できます。通常、サービスやアプリケーションサーバが動作するマシンも、サーバと呼ばれます。

サービスを公開するほとんどの技術には、アプリケーションサーバが関わります。 アプリケーションサーバとは、サービスのディプロイ先となるソフトウェア環境の一種です。 アプリケーションサーバでは、クライアントから送信されたサービス呼び出し要求の着信を監視するリスナと呼ばれる機能が提供されます。 リスナは検出した各要求を、対象のサービスに転送します。アプリケーションサーバは、セキュリティやスケーラビリティといった課題にも対応することが可能であるため、サービスの開発に要する時間を大幅に短縮できます。

Net Express で開発したアプリケーションを実行するには、ランタイム環境が必要です。 Micro Focus は次の 3 種類のランタイム環境を提供しています。

- Enterprise Server for Windows

  Enterprise Server は、COBOL サービス、COBOL/J2EE アプリケーション、および COBOL Web サービスを、Windows、LINUX、UNIX の各プラットフォームでディプロイするためのスケーラブルな管理トランザクション環境です。 Net Express の Interface Mapping Toolkit で作成した COBOL サービスは、Enterprise Server 環境にディプロイして運用します。 Enterprise Server では、着信要求の COBOL への転送、処理負荷に応じた追加コピーのロード、セキュリティ対応などの処理が実行されます。

  Enterprise Server for Windows には、Application Server for Net Express が含まれています。

- Application Server for Net Express

  Application Server は、Enterprise Server の軽量バージョンです。Windows、LINUX、UNIX の各プラットフォーム版があり、Net Express で作成した COBOL アプリケーションに、高いパフォーマンスと堅牢性、および移植性を与えることが実証されています。Application Server には、Net Express で開発したアプリケーションに必要なランタイム支援ファイル群が含まれています。

- Run Time System for Net Express

  Run Time System は、スタンドアロンのクライアントアプリケーションを実行するための Application Server の特別な販売形態です。ソフトウェアとしては Application Server とまったく同一ですが、添付しているライセンスが 1 同時ユーザの実行のみを許容しています。

# 分散コンピューティングに利用できる技術

Net Express では、多彩な技術を COBOL と連携させて分散システムを実装できます。具体的には、次の技術を利用できます。

- Web サービス
- COM
- EJB
- JavaBean
- CGI
- XML

サーバに使用するコンポーネントの種類を決定すると、続いてクライアントに使用する技術を決定します。クライアントに使用する技術にはサーバとの互換性が不可欠ですが、その点を考慮しても幅広い選択肢があります。詳しくは、後述する『クライアントアプリケーションの作成』を参照してください。

## Web サービス

Web サービスとは、ソフトウェアコンポーネントをイントラネットやインターネットを介して呼び出し可能にする一連の規格と方式の総称です。Web サービスのコンポーネントとクライアントの間では、XML を含む可読形式のデータが交換されます。そのため、クライアントアプリケーション側ではサービスの言語やディプロイ方式をいっさい関知する必要がなく、名前とパラメータだけでサービスを呼び出すことができます。

Web サービスでは、クライアントとサーバ間の通信に XML が使用されます。XML レイヤによって、クライアントとサーバは互いの動作環境に関係なく連携できる汎用的で明確な言語が提供されます。

クライアントとサーバ間で XML を交換する手段として最も一般的なのは HTTP による搬送ですが、それ以外にもさまざまな方式が使用できます。HTTP を使用する利点は、ファイアウォールへの対応が容易になることです。クライアントから着信する要求は、XML で記述され、HTTP 標準ヘッダが付けられた要求を含んでおり、通常的な方法で搬送されます。ただし、XML の構成自体は、SOAP などのプロトコルを適用することによって変更できるため、実際にはより複雑になることがあります。

Net Express では、COBOL アプリケーションを Web サービスとして公開できます。Interface Mapping Toolkit を使用して、COBOL プログラムと Web サービスのインターフェイスをマッピングします。その結果、クライアントアプリケーションは COBOL プログラムに、Web サービスとしてアクセスすることが可能になります。

詳細については、このマニュアルの『COBOL Web サービス』のパート、および『入門書』の『インターフェイスマッピングの概要』のパートを参照してください。

## COM

COM は、アプリケーションの機能を他のアプリケーションに公開し、他のアプリケーションで利用可能にする Microsoft Windows の機能です。Net Express には、COM と COBOL を連携させるためのツール群があり、Windows で動作するクライアントアプリケーションは、COBOL で実装された機能を COM オブジェクトを介して利用できます。

COM オブジェクトは Windows プラットフォームにディプロイします。クライアントと COM オブジェクト間のインターフェイスのコーディングには、Windows のプログラミングに関する、ある程度以上のスキルが必要になります。

COM の技術は、次のような過程を経て進化してきました。

- 初期の COM － 1 台のマシン上でアプリケーションから他のアプリケーションを呼び出し可能にする Windows の機能として登場しました。
- DCOM (Distributed COM) － Windows マシン上のアプリケーションから、他の Windows マシン上のアプリケーションの呼び出しが可能になりました。
- Microsoft Transaction Server (MTS) － MTS は Windows の拡張機能の 1 つです。この MTS を通じて、COM オブジェクトがトランザクション処理に対応し、アプリケーションサーバに近い機能が実現できるようになりました。
- COM+ － Windows の新しいバージョンには COM+ が使用されています。COM+ とは、Windows に標準で含まれている COM、DCOM、および MTS のすべての機能の総称です。COM という用語が、COM+ の意味で使用されることも少なくありません。

COM オブジェクトは、Interface Mapping Toolkit を使用して COBOL のインターフェイスにマッピングできます。詳細については、このマニュアルの『COBOL での COM の利用』の章と、『入門書』の『インターフェイスマッピングの概要』のパートを参照してください。

COM オブジェクトは、クラスウィザードとメソッドウィザードを使用して作成することもできます。詳細については、このマニュアルの『COM オブジェクトの作成』の章と、『入門書 - その他のトピック』の『クラスウィザードとメソッドウィザード』を参照してください。

## EJB

Net Express には、1 つの COBOL アプリケーションを 1 つ以上のサービスとして公開するためのツールがあります。 COBOL プログラムを外部の EJB インターフェイスにマッピングするには、Interface Mapping Toolkit を使用します。 このマッピングを行うと、クライアントアプリケーションから EJB を介して、COBOL プログラムの機能を利用できるようになります。

EJB を使用する場合は、COBOL サービスの実行環境である Enterprise Server に加え、EJB を実行するために J2EE 準拠のアプリケーションサーバ (IBM WebSphere、BEA WebLogic など) が必要です。 EJB は Java 標準仕様の一部であり、EJB を使用するクライアントは、JSP や Java Swing といった Java 型のツールで記述する必要があります。 また、クライアントと EJB のインターフェイスのコーディングには、Java に関する専門的なスキルが求められます。

詳細については、このマニュアルの『Java インターフェイスのマッピングとリソースアダプタの使用』の章と、『入門書』の『インターフェイスマッピングの概要』のパートを参照してください。

## JavaBean

JavaBean も、COBOL プログラムと連携可能な Java 技術の 1 つです。 厳密に言えば、JavaBean は技術の名称であり、呼び出し対象のコンポーネントはビーン (Bean) と呼ばれますが、ここでは通例に従ってコンポーネント自体を JavaBean と呼びます。

JavaBean は、Java のランタイム環境である Java VM (Java 仮想マシン) がインストールされたマシンにディプロイします (Java VM は Web からダウンロードできます)。 すべての Java VM の仕様は Java 2 Standard Edition (J2SE) と呼ばれる標準に準拠しており、それゆえ Java VM 環境下への JavaBean のディプロイは、J2SE へのディプロイとも呼ばれます。

クライアントと JavaBean とのインターフェイスのコーディングには、Java に関する専門的なスキルが必要です。 クライアントは、リモートプロシージャコールの仕組みを使用して、JavaBean を直接呼び出す必要があります。

JavaBean はアプリケーションサーバの環境下で動作するわけではないため、スケーラビリティやセキュリティなどの面で、アプリケーションサーバの利点が活かされることはありません。また、クライアントからの呼び出しに備えて、JavaBean がロードされた状態に維持されることもありません。

JavaBean は、クラスウィザードとメソッドウィザードを使用して作成します。 詳細については、このマニュアルの『Java からの Object COBOL の呼び出し』の章と、『入門書 - その他のトピック』の『クラスウィザードとメソッドウィザードの概要』を参照してください。

## サーバ側プログラム (CGI)

サーバ側プログラムは、Web アプリケーションに従来から使用されている技術の 1 つです。サーバ側プログラムは Web サーバの制御下で動作し、 Web サーバの一種の拡張機能と見なすことができます。 サーバ側プログラムの仕組みは、共通ゲートウェイインターフェイス

(Common Gateway Interface) と呼ばれる通信規格として広く普及しており、サーバ側プログラムも通常は CGI プログラム、あるいは CGI と呼ばれます。

CGI プログラムを使用する場合には、クライアントアプリケーションを別途作成する必要はありません。CGI プログラムは Web ブラウザにフォームを送信し、それがユーザインターフェイスになります。 CGI はクライアントからの呼び出しに備えて、ディプロイ先に登録されるわけではないため、コンポーネントやサービスとは見なされません。

CGI を使用するには Web サーバが必要です。さらに、アプリケーションのビルド方法によっては、COBOL プログラムを実行するために Application Server for Net Express が必要になる場合もあります。

詳細については、このマニュアルの『CGI ベースのインターネットプログラミングの概要』のパートと、『入門書 - その他のトピ ック』の『CGI ベースのアプリケーションの概要』を参照してください。

### XML

XML (eXtensible Markup Language) は、Web 上でデータを交換する言語として広く使用されています。 XML の言語仕様は、DTD (Document Type Definition) と呼ばれる XML スキーマ内で定義されたルールセットに準拠します。 Net Express は XML スキーマに対応しており、XML ドキュメントと COBOL アプリケーション間でデータをやり取りすることが可能です。

Net Express には XML 拡張構文セットが含まれており、COBOL プログラム内に記述できるほか、CBL2XML ウィザードやコマンド行ユーティリティで自動生成することもできます。 この拡張構文セットを使用すれば、索引ファイルなど、従来的なファイルベースの手法に頼ることなく、COBOL プログラムから XML インスタンスドキュメントに直接、入出力を行うことが可能になります。

詳細については、このマニュアルの『XML データと COBOL レコードへの格納』のパートと、『入門書』の『XML 対応 COBOL の概要』を参照してください。

## 分散コンピューティング用のツール

Net Express では、分散型システムを実現するために、次のツールを使用できます。

- Interface Mapping Toolkit
- クラスウィザードとメソッドウィザード (および専用の COBOLBean)
- インターネットアプリケーションウィザード

### Interface Mapping Toolkit

Interface Mapping Toolkit を使用すれば、Web サービス、EJB 、COM といった技術を通じて、既存の COBOL プログラムの機能を COBOL 以外で開発したクライアントに公開できます。

そのためには、COBOL プログラム内のエントリポイントとデータ項目を、外部インターフェイスにマッピングします。

クライアントから Web サービスまたは EJB を介して呼び出されると、COBOL プログラムが Enterprise Server の環境下で実行されます。 マッピングしたインターフェイスを介してクライアントから着信する要求は、Enterprise Server によって受け入れられます。 Enterprise Server は、この要求を COBOL ランタイムシステムに渡し、応答をインターフェイス経由でクライアントに返信します。

クライアントから COM オブジェクト経由で要求を行う場合、アプリケーションのビルド方法によっては、COBOL プログラムの実行に Application Server が必要になることもあります。 COM はすべてのマッピング情報を内部に保持しており、Enterprise Server を使用しません。

Interface Mapping Toolkit は、次の各要素から構成されています。

- マッピングウィザード ― COBOL プログラムから連絡節とエントリポイントを抽出し、マッピング可能な状態にします。
- インターフェイスマッパー ― ドラッグアンドドロップ操作によって、COBOL プログラム内のデータ項目とエントリポイントを、定義したインターフェイスにマッピングします。
- ディプロイツール ― 定義したマッピング情報を Enterprise Server にディプロイするツール。 必要なインターフェイス (WSDL ファイル、EJB、または COM オブジェクト) の生成も行います。 WSDL ファイルとは、Web サービスを定義するファイルです。 EJB は Java アーカイブファイル (.jar ファイル) として、 COM オブジェクトはすべてのマッピング情報を含んだプロセス内 .dll ファイルとして、それぞれ生成されます。

Interface Mapping Toolkit の使用を検討する際には、次の点に留意してください。

- Web サービス、COM オブジェクト、および EJB が作成できるが、JavaBean は作成できない。
- COBOL インターフェイスを Web サービスや EJB にマッピングした場合、エンドユーザ側のマシンには Enterprise Server for Windows をインストールする必要がある。

『入門書』の『インターフェイスマッピングの概要』を参照してください。

## クラスウィザードとメソッドウィザード

クラスウィザードとメソッドウィザードを使用すれば、Object COBOL クラスを COBOL アプリケーションのフロントエンドとして作成し、COM としてディプロイすることができます。 フロントエンドとして Java クラスを作成することもできます。その場合、COBOL アプリケーションは、あたかも Java アプリケーションであるかのように振る舞います。また、作成した Java クラスは JavaBean としてディプロイできます。

クラスウィザードとメソッドウィザードの使用を検討する際には、次の点に留意してください。

- COM オブジェクトと JavaBean は作成できるが、EJB を作成したり、Web サービスを直接作成することはできない。

COM オブジェクトを作成した後、Microsoft SOAP Toolkit (Microsoft の Web サイトからダウンロード可能) を使用すれば、COM オブジェクトを Web サービスとして公開できます。 なお、Windows XP では必要な機能が .NET フレームワークに組み込まれているため、SOAP Toolkit を使用する必要はありません。

- Application Server for Net Express を使用すれば、 Enterprise Server for Windows は不要。

  フロントエンドクラスは COBOL アプリケーションのラッパーであり、独立したエージェントではありません。 COM オブジェクトは Windows 上にディプロイし、JavaBean は Java のアプリケーションサーバ、または Java VM の環境下にディプロイします。 また、COBOL アプリケーションは Application Server for Net Express の環境下にディプロイします。

クラスウィザードとメソッドウィザードで作成した Java クラスでは状態管理がいっさい行われないため、状態管理が必要であれば CobolBean クラスを使用してください。 このクラスは標準で提供されており、 COBOL プログラムをスレッドセーフに書き直したり、Java アプリケーションからパラメータを取り込むことなく、CobolBean のインスタンスを COBOL プログラムの記憶域節のインスタンスに関連付けることができます。

『入門書 - その他のトピック』の『クラスウィザードとメソッドウィザードの概要』には、クラスウィザードとメソッドウィザードのチュートリアルが記載されています。 これらのウィザードの詳細については、このマニュアルの『Java からの Object COBOL の呼び出し』と『COM オブジェクトの作成』の章を参照してください。 CobolBean の詳細については、このマニュアルの『Java からの手続き型 COBOL の呼び出し』の章にある『インスタンスデータとしての CobolBean の使用』の項を参照してください。

## インターネットアプリケーションウィザード

インターネットアプリケーションウィザードは、サーバ側プログラムを生成するためのツールです。 生成されたプログラムは、Web サーバ上にディプロイします。 サーバ側プログラムは Web フォームを生成し、そのフォームが Web サーバによってブラウザに送信されます。

インターネットアプリケーションウィザードを使用する前に、フォームを作成します。フォームの作成には、付属ツールの HTML ページウィザードと Form Designer を使用できます。 フォームの作成後、インターネットアプリケーションウィザードを使用して、そのフォームをアプリケーションのインターフェイスとして使用するサーバ側プログラムを作成します。 生成されるサーバ側プログラムは COBOL で記述されており、生成後にアプリケーションにビルドします。

インターネットアプリケーションウィザードでプログラムの連絡節を指定し、Web フォームとサーバ側プログラムを生成させる方法もあります。 また、データベースの SQL 仕様を指定して、フォームを含むデータベース管理アプリケーション全体を生成させることも可能です。

アプリケーションのテストには、標準で付属している軽量 Web サーバ、Solo を使用できます。 実際にアプリケーションを運用する際には、この Web サーバの使用は推奨されません。

インターネットアプリケーションウィザードの使用を検討する際には、次の点に留意してください。

- サービスは作成できない。
- Application Server を使用すれば、Enterprise Server は不要。
- 生成されたプログラムは、Web サーバが動作しているあらゆる環境にディプロイできる。
- 既存のアプリケーションやデータベースを基に、標準的な機能を持つシンプルな Web アプリケーション全体を、きわめて速やかに作成できる。
- スケーラビリティやセキュリティといった運用面のニーズには、開発者側で対処する必要がある。
- クライアントを別に作成する必要がない。

詳細については、このマニュアルの『CGI ベースのインターネットプログラミングの概要』のパートと、『入門書 - その他のトピック』の『CGI ベースのアプリケーションの概要』を参照してください。

# クライアントアプリケーションの作成

クライアントアプリケーションの作成に利用できる技術は、使用したコンポーネントの種類 (Web サービス、EJB、JavaBean、COM) によって左右されます。

Web サービスのクライアントはインターネットアプリケーションウィザードで生成できますが、それ以外のコンポーネントを使用した場合には、Net Express 付属のツールでクライアントを生成することはできません。『入門書』の『クライアントの作成の概要』の章には、COBOL アプリケーションのクライアントの作成例と詳細情報が記載されています。

## Web サービスのクライアント

Web サービスはあらゆる種類のクライアントで使用できます。Web サービスの詳細情報を提供すれば、ネットワークやインターネット上のさまざまな場所で、あらゆる利用者が、そのサービスを利用するクライアントアプリケーションを作成できるようになります。Web サービスの詳細情報は、Web サービス記述言語ファイル (WSDL ファイル) として提供します。このファイルには、Web サービスの名前、クライアントから Web サービスに渡すべき入力データ、Web サービスがクライアントに返す出力データなどの情報が記述されています。

Web サービスとクライアント間では、データがファイルとして HTTP でやり取りされるため、ファイアウォール越しに Web サービスを呼び出すことも可能です。典型的なクライアントとしては、Web サービスとは別に作成され、リモートサイトで実行されて、インターネット経由で Web サービスを呼び出すクライアントなどがあります。

Interface Mapping Toolkit には、インターネット経由で Web サービスを利用するシンプルなクライアントを生成する機能があります。生成したクライアントはテスト用途や、実際に使用するクライアントのひな型として使用できます。

### EJB や JavaBean のクライアント

EJB や JavaBean のクライアントは、Java 標準に準拠した技術を使って実装する必要があります。Java アプリケーションとして実装する場合には、特に制約はありません。インターネットで実行するクライアントは、JSP (Java Server Pages) で実装できます。JSP は Web サーバで動作し、リモートブラウザに HTML ページを表示します。また、GUI アプリケーションとして実装する場合は、Java Swing を使用できます。

JSP は Web サーバ上で実行されるスクリプトです。JSP には HTML と、HTML を生成する命令が含まれ、Web ページを動的に生成できます。JSP の記述には Java を使用します。

Java Swing は GUI オブジェクトを表示するための Java クラスセットです。

### COM オブジェクトのクライアント

COM オブジェクトのクライアントは Windows 上で実行します。実装には ASP、ASP.NET、または WS.NET を使用できます (ASP は ASP.NET によって代替されつつあります)。

典型的なクライアントとしては、サイト内の専用マシンで動作し、複数のリモートユーザに Web ブラウザでインターフェイスを提示するアプリケーションなどがあります。

### CGI プログラムのクライアント

CGI プログラムには、それ自体にクライアント部分が含まれています。実質的には、リモートマシン上の Web ブラウザで、CGI プログラムが生成した Web フォームを使用しているユーザがクライアントになります。

## まとめ

次の表は、COBOL プログラムの公開に使用できる技術、それぞれの技術に利用できるツール、および生成したコンポーネントやサービス、またはインターフェイスのディプロイ先になる環境を一覧しています。

| 技術 | Net Express のツール | ディプロイ先の環境 |
|---|---|---|
| Web サービス | Interface Mapping Toolkit | Enterprise Server、マッピングされたインターフェイス |
| COM オブジェクト | Interface Mapping Toolkit | Windows (COM オブジェクト)、Application Server (COBOL アプリケーション) |
| COM オブジェクト | クラスウィザードとメソッドウィザード | Windows |
| EJB | Interface Mapping Toolkit | Enterprise Server、マッピングされたインターフェイス、および J2EE 準拠のアプリケーションサーバ (EJB) |
| JavaBean | クラスウィザードとメソ | Java VM (JavaBean)、Application Server (COBOL ア |

| | | |
|---|---|---|
| | ッドウィザード | プリケーション) |
| CGI プログラム | インターネットアプリケーションウィザード | Web サーバ (CGI)、Application Server または Enterprise Server (COBOL アプリケーション) |

# 第 2 章 : Interface Mapping Toolkit

この章では Interface Mapping Toolkit について説明します。Interface Mapping Toolkit とは、Net Express でサービス (Web サービス、COM オブジェクト、EJB) を開発する際に使用する主要ツール群の総称です。

## 2.1 概要

Interface Mapping Toolkit を使用すると、既存の COBOL アプリケーションをサービスとしてディプロイできます。サービスとは、別のクライアントアプリケーションから常に呼び出し可能な状態に維持できる、メモリ常駐型のソフトウェアコンポーネントのことです。 サービスの作成では、標準的なサービスの実装方式から既存プログラムを呼び出すためのインターフェイスの作成が作業の中心になります。 サービスの実装方式は、EJB、COM、および Web サービスです。 このインターフェイスと既存プログラム内のパラメータ間のマッピングを定義します。サービスとして再利用するプログラムを、呼び出し可能なプログラムとして最初から設計していること (連絡節を持っていること) が前提になります。

サービスとして再利用するプログラムは、多くの場合、数年間にわたって実際に運用されてきたアプリケーションに含まれます。 そのため、この章ではそのようなプログラムを「レガシープログラム」と呼んでいる場合があります。 ただし、最初からサービスのベースとして使用するために作成したプログラムを使用することも可能です。

Interface Mapping Toolkit の主な構成要素を次に示します。

- 「サービスインターフェイス」ウィンドウ

  作成したサービスのツリー階層を表示します。

- マッピングウィザード

  作成するサービスの基本的なプロパティ (名前、サービスの実装方式、レガシー COBOL プログラムの名前など) を定義するウィザード。

- インターフェイスマッパー

  クライアントとサービスの間でやり取りされるデータと、COBOL プログラムのパラメータとのマッピングを定義するグラフィックドラッグアンドドロップエディタ。

- ディプロイツール

  作成したサービスを運用先にコピーするツールと、ディプロイオプションを設定するダイアログボックス。

- Client Generation

作成した Web サービスを使用する COBOL クライアントアプリケーションの作成ツール。

これらのツールについては、『入門書』で紹介しています。この章では、これらのツールの基本概念を示します。

## 2.2 「サービスインターフェイス」ウィンドウ

インターフェイスとマッピングの定義情報を含むファイルを Interface Mapping Toolkit で生成し、Net Express のリポジトリに格納します。なお、「インターフェイス」と「マッピング」という 2 つの用語は、実際には区別しないで使用されることが少なくありません。

「サービスインターフェイス」ウィンドウには、特定のサービスインターフェイスリポジトリ (.mpr ファイル) に格納されているサービスインターフェイス (マッピング) が表示されます。これら一群のサービスインターフェイスは、サービスインターフェイスグループと呼ばれます。

サービスインターフェイスグループは、常に Net Express のプロジェクトに関連付けられるわけではありませんが、マッピングウィザードで新しいサービスインターフェイスを作成するときにプロジェクトが開いていると、生成されたサービスインターフェイスは、そのプロジェクトのプロジェクトフォルダ内のサービスインターフェイスグループに追加されます。サービスインターフェイスグループが存在しない場合は、新しく作成されます。また、[ファイル > サービスインターフェイスを開く] をクリックしてリポジトリを開くときに、「開く」ダイアログボックスに最初に表示されるのは、現在開いているか、または最後に開いた Net Express プロジェクトのプロジェクトフォルダの内容です。

ツリー階層には、サービスの種類 (Web サービス、EJB、または COM インターフェイス) と、各サービス内に含まれる処理も表示されます。

サービスインターフェイスの名前を右クリックするとメニューが表示され、サービスの編集やディプロイを行うことができます。

## 2.3 マッピングウィザード

マッピングウィザードでは、COBOL アプリケーションを Web サービス、EJB、または COM インターフェイスとして公開し、クライアントアプリケーションからサービスとして呼び出し可能にするための設定を行います。サービスとして公開する COBOL アプリケーションは、副プログラムである (連絡節を持っている) ことが前提になります。

マッピングウィザードでは、サービスの名前や種類 (Web サービス、EJB、COM インターフェイス) など、サービスの基本的なプロパティを定義します。ウィザードの最後にインターフェイスマッパーが開き、クライアントとサービスの間でやり取りされるデータと、COBOL プログラムで使用されるパラメータとのマッピングを定義します。

『入門書』には、マッピングウィザードとインターフェイスマッパーの詳しい使用方法を示すチュートリアルが記載されています。フィールドに関する情報はポップアップヘルプに表示されま

す。特定のフィールドのポップアップヘルプを表示するには、コンテキストヘルプボタン ?をクリックし、続いてそのフィールドをクリックします。 このウィザードの特定状況での動作について、次に説明します。

### 2.3.1 サービスインターフェイスグループが見つからない場合の動作

プロジェクトを開いていない状態で [ファイル > サービスインターフェイスの新規作成] をクリックすると、サービスインターフェイスグループを新たに作成して、新しいサービスをその中に含めるかどうかを尋ねるダイアログボックスが表示されます。 サービスを既存のサービスインターフェイスグループに追加するには [キャンセル] をクリックし、既存のグループ (またはそのグループに関連付けられているプロジェクト) を開いた後で、[ファイル > サービスインターフェイスの新規作成] をもう一度クリックしてください。

### 2.3.2 終了時の警告

既存のサービスで使用されているプログラムを使用するサービスを作成した場合には、ウィザードで [終了] をクリックしたときに、プログラムの更新を通知する警告メッセージが表示されます。 このメッセージは、プログラムがマッピングウィザードの使用中に再コンパイルされたことを示します。 プログラムのソースコード自体に、通知なく変更が加えられることはありません。

### 2.3.3 生成されるファイル

Web サービス、EJB、COM インターフェイスのいずれを作成する場合も、Interface Mapping Toolkit によって 2 つの XML ファイルが生成されます。1 つは COBOL プログラムの説明を含むファイル、もう 1 つはサービスの説明を含むファイルです。 これらのファイルは WSDL ファイルとは異なります。WSDL ファイルはディプロイツールによって、Web サービス用に生成される XML ファイルです。 生成される 2 つのファイルは、どちらも Interface Mapping Toolkit の内部で使用されます。

## 2.4 インターフェイスマッパー

標準的な COBOL プログラム は、いくつかの関連機能を実行します。 たとえば、ファイルメンテナンスプログラムではレコードの追加、返却、削除などを実行することができます。 プログラムをサービスとして公開する場合は、クライアントアプリケーションでこれらの機能をいつでも呼び出すことができます。 サービスの用語では、これらの機能は「オペレーション」と呼ばれます。

インターフェイスマッパーは Interface Mapping Toolkit の中核となるツールです。 インターフェイスマッパーは、プログラムのどの機能をサービスの処理として公開するかと定義するのに使用します。各処理では、サービスとクライアントの間でやり取りされるメッセージに含まれるフィールドを定義します。 これらのフィールドは、インターフェイスフィールドと呼ばれます。

サービスが呼び出されると、インターフェイスフィールドの値が COBOL プログラムの連絡節内の項目に渡され、 サービスの終了時には、連結節内の項目のデータがインターフェイスフィールドに返されます。

Net Express でデフォルトマッピングを生成することも可能です。デフォルトマッピングでは、COBOL プログラムの正規構造を反映したフィールドだけが生成されます。 COBOL プログラムに関する十分な知識があれば、マッピングを明示的に定義することによって、より効率的で使いやすいインターフェイスを実現できます。 マッピングの明示的な定義はインターフェイスマッパーを使用して行います。 最初にデフォルトマッピングを作成して変更を加える方法と、インターフェイスマッパーが空の状態で、インターフェイスとマッピングをすべて明示的に定義する方法のどちらを使用してもかまいません。

### 2.4.1 インターフェイスマッパーのレイアウト

インターフェイスマッパーには、使用時における処理が表示されます。 インターフェイスマッパーのウィンドウは、次の 5 つのペインから構成されます。

- オペレーションの詳細

  処理の名前、全体として処理に適用される他の情報がタイトルバーの近くに表示されます。

- 「連絡節」ペイン

  左側のペインで、COBOL プログラムの連絡節が表示されます。

- 「インターフェイスフィールド」ペイン

  右上のペインで、表示している処理で、サービスとの間でやり取りされるデータフィールドが表示されます。

- 「再利用マッピング」ペイン

  中央右側のペインで、複合データ型を定義します。

- 「COBOL 既定値 (VALUE)」ペイン

  右下のペイン で、連絡節の指定項目に、処理の呼び出し時に割り当てる値を定義します。

各ペインの詳細について、次に説明します。

### 2.4.2 オペレーションの詳細

インターフェイスマッパーには、使用時におけるオペレーションの詳細が表示されます。表示しているオペレーションの名前が右上に表示され、ドロップダウンリストで他のオペレーションを選択できます。

左上には、表示中のオペレーションが呼び出されたときに、プログラムの実行が開始されるエントリポイントが表示されます。

新しいオペレーションを追加するには、[オペレーション > 新規作成] をクリックします。新しいオペレーションの名前を指定し、プログラムのエントリポイントを選択します。

Net Express が作成するデフォルトマッピングでは、各エントリポイントが 1 つのオペレーションになります。シングルエントリポイントが EVALUATE 文を示す場合には、一般的な改善策として異なる機能から選択します。これらの各機能を個別のオペレーションにすることができます。このようなプログラムではほとんどの場合には、1 つのパラメータは、必要な機能を決定するために EVALUATE 文で評価されるコードです。詳細については、『COBOL 既定値 (VALUE)』の項を参照してください。

### 2.4.3 連絡節

左側ペインで、COBOL プログラムの連絡節を COBOL データのツリー階層で表示します。

右側ペインで新しいエントリを作成する最も簡単な方法は、このペインから項目をドラッグする方法です。詳細は、右側ペインの該当する箇所を参照してください。

このペインを右クリックしてポップアップメニューを使用し、新しいインターフェイスフィールドを作成することもできます。

### 2.4.4 インターフェイスフィールド

「インターフェイスフィールド」ペインでは、現在表示されているオペレーションのインターフェイスフィールドを定義します。

このペインには、現在表示されているオペレーションのインターフェイスフィールドが表示されます。別のオペレーションのインターフェイスフィールドを表示するには、インターフェイスマッパーの右上のドロップダウンリストを使用してそのオペレーションを表示します。

[フィールド] メニューを使用すれば、右側のペイン内に新しいエントリを定義できます (「インターフェイスフィールド」ペインでポップアップメニューを使用する方法や、「連絡節」ペインでポップアップメニューを使用する方法、および「連絡節」ペインからデータ項目をドラッグする方法もあります)。最も便利な方法はドラッグする方法です。

データ項目をドラッグしてフィールドを定義すると、そのフィールドはドラッグしたデータ項目にマッピングされます。グループ項目を一括してドラッグし、インターフェイスフィールドのグループを定義することもできます (Web サービスでは、インターフェイスフィールドはグループ化されます)。

Net Express が作成するデフォルトマッピングでは、各オペレーションのインターフェイスフィールドは、オペレーションのエントリポイントで使用される連絡節内の項目から自動的に定義されます。 各インターフェイスフィールドは、連絡節内の対応する項目にマッピングされます。

## プロパティ

インターフェイスフィールドの大部分のプロパティは、「インターフェイスフィールド」ペインの列に表示されます。任意の列をダブルクリックすれば、該当するプロパティの値を更新できます。プロパティの値を変更するには、既存の値を上書きするか、ドロップダウンリストから値を選択します。 一部のプロパティでは、[フィールド > プロパティ] をクリックする必要があります。

### 名前

インターフェイスフィールドの名前はサービスの外部定義 (Web の場合は WSDL) に記述されており、サービスとの間でやり取りされるメッセージ内で、インターフェイスフィールドを識別するキーワードとして使用されます。 デフォルトでは、ドラッグアンドドロップで作成したフィールドは、対応する COBOL 項目と同じ名前になります。 デフォルトで割り当てられる名前は一意です。 この名前は、必要に応じて、わかりやすい名前に変更できます。

### 方向

インターフェイスフィールドには、送信方向に応じて次の 4 つの種類があります。

- 入力 ― サービスで受信されるメッセージ内に含めるべきフィールド
- 出力 ― サービスから返されるメッセージに含まれるフィールド
- 入出力 ― 入出力両方のメッセージに含まれるフィールド
- 出力リターン ― 出力フィールドの一種。戻り値を格納する。

Web サービスには入出力メッセージの概念はありません。また、3 種類のサービスの中で、戻り値を定義しているのは COM だけです。

サービスが呼び出されると、入力フィールドの値が COBOL プログラムの連絡節内の対応する項目に渡され、 サービスの終了時には連絡節内の項目の値が対応する出力フィールドに渡されます。

同じグループ内の各フィールドの方向は、常に同じ値になる必要があります。グループの方向は、グループ内のすべての項目に適用されます。

Net Express が作成するデフォルトマッピングでは、BY VALUE パラメータにマッピングされたフィールドは入力になります。 BY REFERENCE パラメータにマッピングされたフィールドは入出力になります (Web サービスの場合は、BY REFERENCE パラメータごとに 1 対の入力フィールドと出力フィールドが宣言されます)。 一般的な改善策として、出力時に値が渡される一方、入力時には無視される BY REFERENCE パラメータは、 デフォルトマッピングでは入出力フィールドにマッピングされますが、明示的に出力フィールドにマッピングすることができます。

### 型

COBOL では、データ項目の型は PICTURE 文字列で厳密に定義します。 XML (Web サービスとクライアント間のデータ交換に使用される) と Java、および COM は、いずれも専用のデータ型セットを定義します。 デフォルトでは、インターフェイスマッパーはプログラム内の各データ型を、作成するサービス (XML、Java、または COM) の最も近いデータ型にマッピングします。 このプロパティのドロップダウンドンメニューには、使用するサービスで提供されている原始データ型と、「再利用マッピング」ペインで定義した複合型がすべて含まれています。

#### Occurs

COBOL の OCCURS 句に対応するプロパティ。

このエントリをドラッグアンドドロップで作成すると、デフォルトではドラッグした COBOL 項目と同じ値になります。

### マッピング

マッピングでは、インターフェイスフィールドに対応する COBOL データ項目を定義します。 マッピングはインターフェイスフィールドの「マッピング」ダイアログボックス (フィールド > マッピング) に次のような形式で表示されます。

```
data-name -> interface-field-name
```

矢印はマッピングの方向を示します。この方向はインターフェイスフィールドの方向に従います。方向プロパティを変更することのみで変更できます。 入力フィールドの場合は、サービスが呼び出されたときに、インターフェイスフィールドのデータがデータ項目に渡されます。 出力フィールドや出力リターンフィールドの場合には、制御がサービスから呼び出し側プログラムに戻るときに、データ項目からデータがフィールドに渡されます。 入出力フィールドの場合には、これらの両方の方向にデータが渡されます (二重矢印で表されます)。

ドラッグアンドドロップでインターフェイスフィールドを作成した場合には、デフォルトでは、そのフィールドはドラッグした COBOL 項目にマッピングされます。

フィールドは複数マッピングできます。入力フィールドを 1 対多でマッピングすると、そのフィールドで受信されたデータは、マッピングされている全データ項目に渡されます。 出力フィールドの場合は、1 対多のマッピングは無意味です (複数の COBOL 項目のデータが出力フィールドに渡され、先に渡された値が後に渡される値によって上書きされるため) 。ただし、インターフェイスフィールドが Occurs プロパティを持っている場合は例外です。 その場合には、マッピングごとに添字の異なる値を指定できます。

「マッピング」ダイアログボックスには、インターフェイスフィールドのマッピングが、リストボックスに表示されます。マッピングを追加するには、「COBOL フィールド」フィールドに任意の COBOL データ項目の名前を指定し、[追加] をクリックします。 指定したマッピングがリストボックスに追加されます。

マッピング機能の主旨は、特定のインターフェイスフィールドのマッピングを定義することです。したがって、このダイアログボックスでは通常、「フィールド名」の値は変更できません。 ただし、

インターフェイスフィールドが Occurs プロパティを持つ場合には、フィールド名を編集して添字を指定できます。 添字以外を変更すると、[追加] をクリックしたときにエラーメッセージが表示されます。

Occurs プロパティを持つ項目の場合、デフォルトでは各オカレンスが COBOL データ項目の対応するオカレンスにマッピングされます。「Occurs」列の値が連絡節の OCCURS 句の値に一致しないと、「マッピング」ダイアログボックスが表示される前に警告メッセージが表示されます。 その場合には、「マッピング」ダイアログボックスで、オカレンスごとにマッピングを指定する必要があります。

### 2.4.5 再利用マッピング

「再利用マッピング」ペインは、主に他のペインの補助的な用途で使用します。 たとえば、インターフェイスフィールドのグループを作成して複数のオペレーションで使用する場合、「インターフェイスフィールド」ペインでグループを作成し、それを他のオペレーションに逐次コピーする方法は使用できません。「インターフェイスフィールド」ペインには表示中のオペレーションのフィールドしか表示されないからです。 かわりに「再利用マッピング」ペインでグループを作成すれば、このペインから各オペレーションの「インターフェイスフィールド」ペインにグループをコピーできます。 別のオペレーションを表示しても、「再利用マッピング」ペインの表示内容は変更されません。

[フィールド] メニューを使用すれば、右側のペイン内に新しいエントリを定義できます (「再利用マッピング」ペインでポップアップメニューを使用する方法、「連絡節」ペインからデータ項目をドラッグする方法もあります)。 最も便利な方法はドラッグする方法です。

「再利用マッピング」ペイン内で項目を作成すると、実際には複合データ型が定義されます。 すなわち、Web サービスでは WSDL ファイル内で複合データ型が定義され、 EJB では Sun の CCI (Common Client Interface) で定義されている CustomRecord インターフェイスに準拠したカスタムレコードが生成され、 COM では新しい複合データ型が生成されます (グループ構造内の各項目がプロパティとして公開されます)。

いずれの場合も、インターフェイスマッパーには複合データ型が、COBOL データ階層と同様の形式で表示されます。「再利用マッピング」ペイン内の各項目は、グループ項目、基本項目の違いに関係なく、すべて複合型です。 定義した複合型を「インターフェイスフィールド」ペインにドラッグすると、その型のインターフェイスフィールドが作成されます。「インターフェイスフィールド」ペインの「型」列をダブルクリックすると、定義したすべてのデータ型と、サービスの実装方式 (Web サービス、EJB、または COM) 固有のデータ型が、その列に一覧されます。

「再利用マッピング」ペインに表示されるプロパティは、インターフェイスフィールドのプロパティと同様です。ただし、「再利用マッピング」ペイン内の項目は型宣言であり、実際のフィールドではないため、方向プロパティはありません。 大部分のプロパティは、「再利用マッピング」ペインの列に表示されます。任意の列をダブルクリックすれば、該当するプロパティの値を更新できます。 また、[フィールド > プロパティ] をクリックする方法や、エントリを右クリックして [プロパティ] をクリックする方法でも更新できます。

「再利用マッピング」ペインのエントリのマッピングの更新は、インターフェイスフィールドの更新と同じように、[フィールド > マッピング] をクリックします。

### Web サービスでの「再利用マッピング」ペインの活用

Web サービスを作成しているときには、上記以外の理由でも「再利用マッピング」ペインが役立ちます。グループ項目を「インターフェイスフィールド」ペインにドラッグすると、COBOL のデータ階層に似たインターフェイスフィールド階層が生成されます。Interface Mapping Toolkit は、各グループフィールドを、WSDL ファイル内で複合型として宣言します。同じグループを複数のオペレーションに作成すると、同じ複合型が繰り返し宣言されますが、このような宣言は WSDL では不正です。WSDL では、それぞれの複合型が互いに一意の名前を持つ必要があります。

この問題は、グループフィールドを作成するたびに名前を変更すれば回避できますが、「再利用マッピング」ペインにグループ項目をドラッグすれば、より効率的かつ容易に対処できます。「再利用マッピング」ペインに項目をドラッグすると複合型が宣言されます。続いて「インターフェイスフィールド」ペインにドラッグすれば、その型のインターフェイスフィールドを必要な数だけ、型宣言を伴わずに作成できます。

## 2.4.6 COBOL 既定値 (VALUE)

「COBOL 既定値 (VALUE)」ペインでは、連絡節の指定項目に、現在表示されているオペレーションの呼び出し時に割り当てる値を定義します。

このペインには、現在表示されているオペレーションの事前設定された COBOL 値が表示されます。別のオペレーションのインターフェイスフィールドを表示するには、インターフェイスマッパーの右上のドロップダウンリストを使用してそのオペレーションを表示します。

同じエントリポイントから開始するオペレーションが複数存在し、パラメータの値によって実行するオペレーションを選択するときに事前設定された COBOL 値を使用します。パラメータにマッピングするインターフェイスフィールドを定義するかわりに、特定のオペレーションを実行させる値を、このペインに配置します。

[フィールド] メニューを使用すれば、右側のペイン内に新しいエントリを定義できます (「COBOL 既定値 (VALUE)」ペインでポップアップメニューを使用する方法、「連絡節」ペインからデータ項目をドラッグする方法もあります)。最も便利な方法はドラッグする方法です。ダイアログボックスが自動的に表示されます。このダイアログボックスでは、オペレーションの開始時に項目に設定される値を指定できます。

グループフィールドをドラッグすると、各基本項目に対してダイアログボックスが順に表示されます。すべての基本項目に対して値を設定したくない場合は、[キャンセル] をクリックします。

事前設定された COBOL 値エントリには、値を割り当てるプロパティは 1 つのみです。プロパティは「COBOL 既定値 (VALUE)」ペインの列に表示され、変更する場合にはプロパティをダブルクリックします。また、[フィールド > プロパティ] をクリックする方法や、エントリを右クリックして [プロパティ] をクリックする方法でも更新できます。

## 2.5 ディプロイツール

ディプロイツールは、WSDL ファイル、必要に応じて EJB または COM オブジェクトを生成し、マッピング情報とレガシーアプリケーションをディプロイ用フォルダにコピーします。

[Net Express] メニューの 2 つの機能でこのツールを使用します。[サービス > 設定] を使用してディプロイオプションを設定してから [サービス > ディプロイ] をクリックしてディプロイします。

COM インターフェイスの場合は、インターフェイスを含むダイナミックリンクライブラリ (.dll ファイル) がビルドされ、Net Express のディプロイ用フォルダにコピーされます。

Web サービスや EJB の場合には、インターフェイスファイルがアプリケーションファイルとともに、指定した Enterprise Server マシンのディプロイ用フォルダにコピーされます。

ディプロイの詳細については、『Enterprise Server ディプロイガイド』を参照してください。以下の項では、ディプロイツールのいくつかの側面について詳しく説明します。

### 2.5.1 データファイル

実装する際には、レガシープログラムからデータファイルを検出できるようにする必要があります。

COM の場合は、クライアントアプリケーションが開始されたフォルダでクライアントとサービスの両方が実行されます。クライアントとサービスをこのフォルダに置くことでデータファイルを検索することができます。

Web サービスや EJB の場合には、データファイルをプログラムとともにディプロイできます。「設定」ダイアログボックスの「アプリケーションファイル」タブで、ディプロイするファイルを指定します。デフォルトでは、現在のサービス実行ディレクトリがサービスの起動ディレクトリとして設定され、同じディレクトリ内にディプロイされたファイルがサービスで検出可能になります。

データファイルが他のアプリケーションでも使用される場合は、そのアプリケーションとサービス用に、それぞれ別のコピーを用意してください。このような状況が発生するのは、作成したサービスと並行してレガシーアプリケーションの運用を継続する場合です。明示的に別コピーを作成する方法以外でも、この問題は回避可能です。

たとえば、SELECT 文で環境変数を使ってファイル名を指定 ($MYDATADIR\myfile.dat など) したうえで、環境変数にコピー元ファイルの場所を登録する方法があります。COM の場合、環境変数はオペレーティングシステムで設定できます。

Enterprise Server はオペレーティングシステムの環境設定を継承するため、オペレーティングシステムで環境変数を設定する方法は、Web サービスと EJB でも有効です。また、サービスのディプロイ先となる Enterprise Server で、環境変数をローカルに設定することもできます。その場合、Enterprise Server のサーバ定義はオペレーティングシステムに依存しないため、リモート管理が可能です。詳細は『Enterprise Server 構成と管理』を参照してください。環境

変数は個々のサービスごとに設定することも可能です (後述する『Enterprise Server のランタイム環境』を参照してください)。

サービスのディプロイ先となるサーバとディレクトリが特定できる場合には、SELECT 文で絶対パスを指定できます。

### 2.5.2 設定

ディプロイ設定は、[サービス > 設定] で指定できます。

COM インターフェイスの場合は、使用するスレッドモデルを指定できます。スレッドモデルの詳細については、『COM オートメーションサーバの実行』の章を参照してください。

Web サービスや EJB の場合は、ディプロイ関連の情報の範囲を指定できます。ダイアログボックス内のフィールドに関する情報はポップアップヘルプに表示されます。特定のフィールドのポップアップヘルプを表示するには、コンテキストヘルプボタン ? をクリックし、続いてそのフィールドをクリックします。これらの設定の一部について、次に説明します。

#### エンタープライズサーバ名

「エンタープライズサーバ名」フィールドには、サービスをディプロイするときに受け入れ先となる Enterprise Server の名前を指定します。この Enterprise Server は、[ディプロイ] のクリック時に [開始] 状態になっている必要があります。

このフィールドでは、サービスの実行環境になる Enterprise Server は指定しません。

- 方法

通常は、どちらの用途にも ESDEMO で対応できます。

#### Enterprise Server のランタイム環境

[Enterprise Server 実行時環境の構成] ボタンを使用すれば、サービスの実行環境 (スイッチ、チューナー、環境変数など) を設定できます。これらの設定は、現在のサービスに限り、サービスが動作する Enterprise Server を対象として [Enterprise Server Administration] で指定した環境設定より優先されます。

設定結果を有効にするには、「Enterprise Server 実行時環境の使用」チェックボックスを選択します。このチェックボックスを選択しないと、このダイアログボックスの設定内容は無視されます。

# クライアント生成

クライアント生成は Web サービスのみで利用可能です。詳細については、『Interface Mapping Toolkit による COBOL Web サービスの作成』の章を参照してください。

# 第 3 章 : Enterprise Server へのアプリケーションの実装

この章では、COBOL アプリケーションを Enterprise Server の環境下で動作するサービスとして公開する場合の留意点について説明します。

## アプリケーションコンテナ

Enterprise Server 環境下にサービスとしてディプロイした COBOL アプリケーションは、常にサーバのサービス実行プロセス内で実行されます。 COBOL アプリケーションを実行する部分のサービス実行プロセスは、厳密に管理された COBOL ランタイム環境であり、アプリケーションコンテナとも呼ばれます。 Enterprise Server は、アプリケーションの呼び出しから次の呼び出しまでの間、一貫した状態を維持することによって、例外処理の動作に一貫性を与えます。

クライアントからの要求が着信すると、呼び出し対象のアプリケーションプログラムがロードされ、呼び出しが実際に実行される前に、コンテナによって実行環境がセットアップされます。この環境は、Interface Mapping Toolkit を使用して、ディプロイ設定の Enterprise Server ランタイム設定に指定した情報に基づいて構築されます。 この情報には、アプリケーションに必要な環境変数やチューナー、およびスイッチが含まれています。 サービスのディプロイ時にランタイム設定情報を指定しなかった場合には、デフォルト環境がセットアップされます。

クライアントからの要求には、短期実行と長期実行の要求があります。 短期実行のクライアント要求とは、クライアントとサービス間のやり取りが一度だけ実行される要求のことです。クライアントからの要求が着信するとサービスが実行され、応答がクライアントに返されます。クライアントから Web サービスに送信される要求は、常に短期実行の要求です。 一方、長期実行のクライアント要求とは、同じクライアントによって繰り返し実行されるサービス要求 (それぞれのサービス呼び出しの間でデータを保持する必要がある) か、トランザクションの一部を構成する要求のことです。 したがって、クライアントが WebLogic などの J2EE サーバで実行されているステートフルな JavaBean の場合、この Bean の実行が続く限り、サービスも継続して実行されます。

### 短期実行要求

アプリケーションが短期実行要求で呼び出された場合には、そのアプリケーションが終了 (正常終了、ランタイムシステムエラーによる終了の違いを問わない) すると、応答が生成されてクライアントに返される前にアプリケーションコンテナが初期状態に戻ります。

サービス実行プロセスは通常、クライアントに応答を送信した後、Enterprise Server から他の要求の処理を通知されるまで待機します。 ただし、次の条件のいずれかに該当する場合は、アプリケーションの実行終了後に、サービス実行プロセスも終了します。

- コンテナの完全性がなんらかの形で損なわれた (エラー 114 などのランタイムシステムエラーが発生したり、アプリケーションによってロードされた .dll ファイルや .lbr ファイルをメモリから解放できない場合など)
- サービスをディプロイする前に、「Enterprise Server 実行時環境の構成」ダイアログボックスの「ディプロイメントの特性」タブで「アプリケーション終了後のコンテナの再利用」チェックボックスの選択を解除した。このチェックボックスの選択を解除するのは、コンテナを安全に再利用できる状態にしないまま、アプリケーションが終了してしまう可能性が高い場合だけです。たとえば、アプリケーションが複数の言語で開発されており、COBOL 以外の言語で作成したモジュールが、アプリケーションの終了前にリソースの解放を行わない場合などです。
- サービスをディプロイする前に、「Enterprise Server 実行時環境の構成」ダイアログボックスの「ディプロイメントの特性」タブで「回復不可能でない実行時システムエラー後のコンテナの再利用」チェックボックスの選択を解除しており、かつ重大エラー以外のランタイムシステムエラー (エラー 173 の Program not found など) で アプリケーションが終了した。このチェックボックスの選択を解除するのは、アプリケーションの実行中に発生するランタイムエラーは、すべてコンテナの完全性を損なう重大なエラーだと判断した場合だけです。

これらの理由でコンテナが終了した場合には、Enterprise Server によって新しいサービス実行プロセスが開始されます。

## 長期実行要求

長期実行要求は、トランザクションの開始時か、ステートフルなサービスの呼び出しが要求されたときに開始されます。 長期実行要求ではトランザクションが終了するか、ステートフルな処理が不要になった旨の通知がクライアントから送信されるまで、クライアントとサーバ間の接続が一貫して維持されます。 ただし、長期実行要求の実行中にトランザクションが開始される場合 (トランザクション要求にステートフルなサービス要求が先行した場合など) や、トランザクションの最初のサービス要求としてステートフルなサービスが要求される場合もあります。 そのような場合はトランザクションが終了しても、サーバがステートフルモードで動作しているため、長期実行要求は終了しません。

長期実行要求がいったん開始されると、サービス実行プロセスは、その要求の発信元クライアントから着信する要求だけを処理するようになります。 そのため、他のクライアントからの要求にも応答できるように、Enterprise Server は既存のサービス実行プロセスでステートフル要求の処理が開始されるたびに、新しいサービス実行プロセスを開始します。

ステートフル処理の実行中には、サービス要求が正常に終了しても、アプリケーションコンテナは初期状態に戻りません。 ステートフルな処理が不要になった旨の通知をクライアントから受信するまで、コンテナはクライアント状態のまま複数のサービス要求にわたって維持されます (したがって、ユーザプログラムはキャンセルされず、環境変数やチューナー、およびスイッチは初期状態にリセットされず、ユーザライブラリはメモリ内に維持されます)。ステートフルな処理が不要になった旨の通知をクライアントから受信すると、アプリケーションコンテナは初期状態に戻ります。

また、サービス要求によってランタイムシステムエラーが発生した場合には、ステートフルモードは暗黙的に終了します。

Enterprise Server は、サービス実行プロセスの数が設定値を超えないように管理します。したがって、ステートフルなクライアント要求が完了したときに、Enterprise Server がその要求を実行していたサービス実行プロセスを終了させることもあります。

# リソース管理

サービスとして公開している COBOL アプリケーションがデータベースやファイルなどの外部リソースを使用する場合、それらのリソースは COBOL アプリケーション自体で管理されるか、あるいはアプリケーションコンテナが管理を代行します。使用するリソースをアプリケーション自体で管理するサービスはアプリケーション管理サービス、コンテナに管理を任せるサービスはコンテナ管理サービスと呼ばれます。

ディプロイするアプリケーションのリソース管理の種類は、[サービス] メニューの [設定] で指定できます。

## アプリケーション管理サービス

サービスは、次のいずれかの条件に該当する場合、自身でリソースを管理する必要があります。

- アプリケーションがトランザクションロジック (COMMIT、ROLLBACK) を含み、ファイルやデータベースにトランザクション内でアクセスする。
- アプリケーションが Enterprise Server でサポートされていないリソースマネージャを使用する (『コンテナ管理サービス』を参照)。

トランザクション以外の方法でファイルにアクセスする (COMMIT と ROLLBACK を含まない) アプリケーションによるサービスは、アプリケーション管理にする必要があります。

トランザクション処理を行うアプリケーション管理サービスは、必要なすべてのトランザクションロジックを含み、すべてのリソースをコミットまたはロールバックした状態で完了する必要があります。

アプリケーション管理サービスの実行方法は、Net Express や Application Server といった従来的な実行環境でのアプリケーションの実行方法と類似していますが、サービスの実行終了後もプロセスが継続されるという点で両者は大きく異なります。

## コンテナ管理サービス

サービスとして動作し、データベースやファイルを使用する一方で、トランザクションロジックをいっさい含まないアプリケーションは、リソース管理をアプリケーションコンテナに任せることができます。

アプリケーションコンテナでリソースを管理するには、必要なリソースマネージャを事前に指定しておく必要があります。リソースマネージャは Enterprise Server で指定します。指定できるのは XA 準拠のリソースマネージャだけであり、具体的には次の 3 種類のデータベースリソースマネージャが指定可能です。

- IBM DB2
- Oracle 8
- Oracle 9

リソースマネージャの指定方法については、『Enterprise Server 構成と管理』マニュアルの『構成』の章の『リソースマネージャ』を参照してください。

アプリケーションコンテナは、「Enterprise Server 実行時環境の構成」ダイアログボックスの「ディプロイメントの特性」タブで設定された情報に基づいて、リソースの処理方法 (コミット/ロールバック) を決定します。このダイアログボックスを表示するには、ディプロイするマッピングを含む .mpr ファイルを右クリックして [Enterprise Server 実行時環境の使用] を選択し、[Enterprise Server 実行時環境の構成] をクリックします。また、アプリケーションが呼び出しの成否を示す戻り値を返すかどうか、および呼び出しが成功した場合の戻り値の最大値も指定できます。呼び出しの成否を示す戻り値を返さないように指定すると、アプリケーションコンテナは呼び出しがすべて成功したものと解釈します。

# COBOL アプリケーションに適用される制約

サービスとして公開する場合、COBOL アプリケーションにはいくつかの制約が適用されます。制約のほとんどは、プログラムをリモートユーザから呼び出される自己完結型のコンポーネントとして使用するために生じる、必然的なものです。

COBOL アプリケーションをサービスとして公開する作業に着手する前に、アプリケーションがサービスに適しているかどうかを検討してください。公開するすべての機能を調べ、この項で説明する条件を満たしていることを確認してください。検討の結果、アプリケーションに多少の変更が必要になることもあります。

## ユーザ入出力

サービスは関数として機能する必要があります。つまり、データを受け取る手段は入力パラメータのみ、処理結果を返す手段は出力パラメータのみに限定し、ユーザとの直接的なやり取りを含めるべきではありません。これは、データの取得や出力の手段として ACCEPT 文と DISPLAY 文を使用出来ないことを意味します。

Enterprise Server 環境下でサービスとして動作しているアプリケーションでは、ANSI ACCEPT 文は常に空白文字を返し、ANSI DISPLAY 文はすべて無視されます。また、ANSI ACCEPT や ANSI DISPLAY 以外の画面出力やキーボード入力処理が存在すると、ランタイムシステムエラー 197 が生成され、アプリケーションは終了します。

## ランタイムエラーと終了

サービスは常に、適切な手順を経て終了する必要があります。適切な手順とは、中断されることなく実行され、クライアントに処理結果を返してから終了し、サーバに制御を戻すことです。

したがって、どのようなエラーが発生した場合でも停止することなく、エラーの通知を出力パラメータで返してから終了すべきです。プログラムがクラッシュしたり、中断されることがないように、FILE STATUS 句や宣言手順などの手段を通じて、あらゆるエラー状態を確実に検出させる必要があります。

同様の理由で、COBOL アプリケーションでは STOP RUN 文を使用すべきではありません。サービスの実行中に STOP RUN 文が呼び出されると、プログラムが適切な手順を経ずに、ただちに終了してしまいます。

## マルチスレッド

Enterprise Server のサービス実行プロセスでは、シングルスレッドのランタイムシステムを使用するアプリケーションのみが実行可能であるため、マルチスレッドアプリケーションをサービスとして公開することはできません。

## 状態の保持

短期実行クライアント要求で呼び出されるサービスの場合、アプリケーションは同じクライアントからの前回以前の呼び出しを関知することなく、Enterprise Server で実行されます。呼び出し時のデータや状態を次の呼び出しまで保持する必要があるときには、データを作業ファイルに保存する方法など、開発者自身が適切な方法をプログラムに盛り込む必要があります。

## ファイル操作

Micro Focus がファイル操作用に提供している File Handler API を使用する場合、この API を通じて開いたすべてのファイルを、アプリケーションの終了時に確実に閉じてください。ファイルを開いたままの状態で残すと、以降のサービス要求でサービス実行エラーが発生することがあります。

## リソースの解放

アプリケーションで使用したすべてのリソースは、COBOL ランタイムシステムで管理される部分を除き、すべて終了前に解放する必要があります。リソースを正しく解放しないと、コンテナによる以降のサービス呼び出し時に、サービスの正しい動作が保証されません。他のアプリケーションによって使用されているサードパーティ製モジュールなど、解放できないリソースが存在する場合には、その旨をサービスのディプロイ時に通知してください。この通知を行っておけば、Enterprise Server 内のサービス実行プロセスは、そのアプリケーションの終了まで待機するようになります。

# 第 4 章 : COBOL Web サービス

この章では、Net Express で Microsoft SOAP Toolkit を使用して、既存の Web サービスを利用するアプリケーションを開発する方法と、COBOL Web サービスそのものを作成する方法を説明します。

## 概要

「Web サービス」とは、ソフトウェアコンポーネントを Web 経由で呼び出すための一連の標準と仕組みの総称です。呼び出されるコンポーネント自体は Web サービスと呼ばれます、Web サービスとそれを使用するアプリケーション (クライアントアプリケーション) の間では、すべての情報が XML ファイルとして、HTTP などの標準プロトコルで搬送されるため、Web サービスの呼び出しと通信がきわめて容易になります。クライアントアプリケーション側では、コンポーネントのディプロイ形態や使用されている言語といった詳細を、いっさい関知する必要がありません。

コンポーネントを Web サービスとして公開すると、原理的には Web にアクセスできる世界中のあらゆる個人が、そのコンポーネントを呼び出せるようになります。たとえば、信託会社はクレジットカードの有効性を検証する Web サービスを、さまざま販売業者に広く提供しています。ただし、実際には Web サービスは社内イントラネットで使用されることが多く、内部向けのアプリケーションの一部として、あるいはそのようなアプリケーション間の差異を橋渡しする手段として、活用されています。

Web サービスは、クライアントとサービスにリンクするために必要な XML ファイルの形式を定義する次の 3 つの標準を核として形成されています。

- WSDL (Web Services Description Language)

  Web サービスのクライアントの開発者には、この形式で記述された情報を含むファイルが提供されます。このファイルには、サービスに渡すべきデータと戻り値が記述されています。

- SOAP (Simple Object Access Protocol)

  クライアントとサービスの間では、この形式でデータがやり取りされます。

- UDDI (Universal Description, Discovery and Integration)

  Web に公開されるサービスのリストは、この形式で記述されます。

WSDL、SOAP、および UDDI の最新仕様については、W3C の Web サイト (http://www.w3c.org) をご覧ください。

## COBOL Web サービス

Net Express では、次に挙げる複数の手段によって、COBOL プログラムを Web サービスとして公開できます。

- Interface Mapping Toolkit を使用して呼び出しインターフェイスを Web サービスのインターフェイスにマッピングし、WSDL ファイルを作成する。

  Web サービスは、Enterprise Server をインストールした環境にディプロイします。

  この方法は、『Interface Mapping Toolkit による COBOL Web サービスの作成』の章で説明しています。『入門書』の『インターフェイスマッピングの概要 』の章以降には、いくつかのチュートリアルが記載されています。

- Interface Mapping Toolkit または クラスウィザードとメソッドウィザードを使用してプログラムを COM オブジェクトとしてディプロイし、Microsoft SOAP Toolkit で呼び出しインターフェイスを抽出して、それを基に WSDL ファイルを作成する。

  COM オブジェクトは Application Server をインストールした環境にディプロイします。

  この方法は、『Interface Mapping Toolkit による COBOL Web サービスの作成』の章で説明しています。『入門書 - その他のトピック』の『Web サービスとしての COM オブジェクトのディプロイ』にはチュートリアルが記載されています。 SOAP Toolkit は、 Net Express Links のリンク先にある Microsoft の Web サイトからダウンロードできます。

- Java ベースのツールを使用する。

  Java のクラスを Web サービスとして公開したり、Java アプリケーションから Web サービスを呼び出す手段として、IBM Web Services Toolkit など、さまざまなツールキットが利用できます。 Micro Focus COBOL には Java との連携をサポートする機能が含まれており、これらのツールキットを COBOL から利用することが可能です。

  この方法については、弊社マニュアルでは特に説明していません。 COBOL と Java の連携については、『Java と COBOL の連携』の章を参照してください。

『はじめに』の章に、これらの方法の長所と比較を示しています。

# 第 5 章 : Interface Mapping Toolkit による COBOL Web サービスの作成

Net Express では、COBOL アプリケーションを Web サービスとして公開できます。そのために提供されているツールの集まりが Interface Mapping Toolkit です。Interface Mapping Toolkit は、Net Express でサポートされる 3 種類のサービス、すなわち Web サービス、EJB、および COM オブジェクトのいずれにも使用可能です (Interface Mapping Toolkit については、『分散コンピューティング』の『はじめに』の『Interface Mapping Toolkit』の章で説明しています)。

この章では、Interface Mapping Toolkit の Web サービスに関連する側面に焦点を当てます。

## 概要

『COBOL Web サービスの概要』の章で説明しているように、Net Express ではいくつかの方法で Web サービスを作成できます。その中でも、Interface Mapping Toolkit を使用する方法が、Enterprise Server と組み合わせて完全に特化したソリューションを作成できるという点で推奨されます。この方法とその他の方法は『はじめに』の章で詳しく比較されています。

Interface Mapping Toolkit には、次に挙げる Web サービス関連機能があります。

- WSDL ファイルの生成

  Web サービスとそのインターフェイスを XML 形式で定義したファイル。ディプロイツールで生成されます。Web サービスのクライアント開発者に、このファイルを提供します。

- クライアント生成
- 作成した Web サービスを使用する COBOL クライアントアプリケーションの生成

## WSDL ファイルの生成

WSDL は、Web サービスのクライアントインターフェイスを正式に記述する業界標準の Web サービス記述言語です。Interface Mapping Toolkit は WSDL を完全にサポートしており、あらゆる言語で作成したクライアントが、オペレーティングシステムの違いに関係なく、Interface Mapping Toolkit で作成した Web サービスにアクセスできます。

通常、WSDL ファイルを開発者自身で作成するには WSDL に関する知識が必要になりますが、Interface Mapping Toolkit では WSDL ファイルがディプロイ時に自動生成されるため、その WSDL ファイルをクライアント側に提供するだけです。

WSDL ファイルに基づいて Web サービスのクライアントを生成するツールは、数多くのベンダーから提供されており、生成されるクライアントの言語も多岐にわたります。Interface Mapping Toolkit にも、そのようなツールが含まれています。『クライアント生成』の項を参照してください。

# クライアント生成

Interface Mapping Toolkit には、Web サービスを使用する COBOL クライアントアプリケーションを生成する機能があります。生成されたアプリケーションは、実際の運用に使用するアプリケーションのひな型や例として利用できます。このアプリケーションは 2 つの COBOL プログラムから構成されており、一方のプログラムは ANSI 標準の ACCEPT 文と DISPLAY 文を使用するシンプルなコマンド行ユーザインターフェイスを備えています。このプログラムから呼び出されるもう一方のプログラムはプロキシと呼ばれ、クライアントアプリケーションから Web サービスを呼び出す方法を示します。

## マッピングと WSDL

[サービス > クライアントを生成] をクリックすると、[マッピングを使用] と [WSDL を使用] という 2 つのオプションが表示されます。この 2 つの違いを理解することは重要です。

Web のクライアントは通常、サービスを定義した WSDL ファイルから作成します。WSDL ファイルは Web (イントラネットやインターネット) からダウンロードできる場合もあります。クライアントアプリケーションを作成する場合、通常は WSDL ファイルを読み取ってクライアント (またはサービスを呼び出す部分) を生成するソフトウェアツールを使用します。[クライアントを生成] の [WSDL を使用] は、そのようなツールの 1 つです。

Net Express で作成した Web サービスをディプロイする際に生成されるファイルの中には WSDL ファイルも含まれており、このファイルを [クライアントを生成] の [WSDL を使用] の入力として使用できます。ただし、このオプションは Interface Mapping Toolkit を使用して作成した Web サービス専用ではありません。

その用途専用のオプションが [マッピングを使用] です。このオプションを使用すると、クライアントが WSDL を使用せずに、マッピングから直接生成されます。常に有効な [WSDL を使用] オプションとは異なり、[マッピングを使用] オプションが有効になるのは、「サービスインターフェイス」ウィンドウで Web サービスを選択した場合だけです。Web サービスを右クリックし、ポップアップメニューで [クライアントを生成] オプションを選択すると、[マッピングを使用] オプションがただちに使用できます。

[マッピングを使用] の利点は、レガシープログラムに似たデータ定義を持つクライアントが生成される点です。

COBOL では、PICTURE 句を使用してデータ項目を細かく制御できます。それに対して、WSDL ファイルに使用されている XML では、データ項目は固定長のデータ型だけに制約されます。Net Express で WSDL ファイルを生成すると、それぞれのインターフェイスフィールドに、対応するデータ項目の PICTURE 句と互換性がある XML データ型が割り当てられますが、データ項目の正確な定義まで反映されるわけではありません。その後、クライアントを生成するときには、インターフェイスフィールドの XML データ型を反映した PICTURE 文字列がデータ項目に割り当てられます。その結果、クライアント内のデータ項目の定義は、レガシープログラム内の対応するデータ項目と互換性はあるものの、必ずしも同一ではなくなります。—

方、[マッピングを使用] を使用した場合には、生成されるクライアント内のデータ定義は、レガシープログラムを直接ベースにしたものになります。

## プロジェクトへの追加

WSDL ファイルからクライアントを生成する際に使用するウィザードは 2 つのページから構成され、最初のページで WSDL ファイルを指定し、次のページでクライアントを追加するプロジェクトを選択します。マッピングから生成する場合は、このウィザードの 2 ページ目にあたるダイアログボックスだけを使用します。

デフォルトでは、クライアントは現在開いているプロジェクトに追加されます。通常は、このプロジェクトにサービスも含まれます。ただし、生成したクライアントとサービスへの作業を、それぞれ個別に実施する方が望ましい場合には、クライアントの追加先として、異なるプロジェクトを指定することもできます。

ウィザードで [完了] をクリックしたとき (ダイアログボックスの場合は [OK ] をクリックしたとき) にクライアントが生成されることに留意してください。「既存の Net Express プロジェクトを使用」を選択して [完了] (または [OK]) をクリックすると、続行するかどうかを尋ねるメッセージが表示されます。そこで [はい] をクリックすると指定したプロジェクトにクライアントが追加され、[いいえ] を選択するとプロジェクトには追加されません。ところが、どちらの場合でも、指定したプロジェクトのフォルダ内にはクライアントが生成されています。

# 第 6 章 : Microsoft SOAP Toolkit による COBOL Web サービスの作成

この章では、Net Express で Microsoft SOAP Toolkit を使用して、既存の Web サービスを利用するアプリケーションを開発する方法と、COBOL Web サービスそのものを作成する方法を説明します。

## 概要

Microsoft SOAP Toolkit は、Net Express Links のリンク先にある Microsoft の Web サイトからダウンロードできます。このツールキットには、既存の Web サービスにアクセスするための一連の COM コンポーネントと、既存の COM コンポーネントから WSDL ファイルを生成し、COM コンポーネントを Web サービスとして公開するための WSDL ジェネレータが含まれています。Net Express では、COBOL プログラムから COM コンポーネントにアクセスしたり、COBOL で COM コンポーネントを作成することが可能なため、Microsoft SOAP Toolkit を COBOL プログラムから利用することができます。

Micro Focus COBOL での COM と Java の連携は、オブジェクト指向プログラミング用の COBOL 拡張構文を利用して実現されますが、オブジェクト指向プログラミングの知識は必ずしも前提になりません。Web サービスを作成する際には、必要なすべてのオブジェクト指向コードが、Net Express のクラスウィザードとメソッドウィザードによって生成されるからです。また、COBOL プログラムから Web サービスを呼び出す場合には、新たに追加された **invoke** 動詞を使用できます。

この章の内容は、Net Express での COBOL プログラムと COM の連携に関する基本的な知識を前提としています (『***分散コンピューティング***』の『COBOL での COM の利用』のパートを参照)。

この章は 2 つの項から構成されています。1 つは COBOL Web サービスの作成に関する項、もう 1 つは COBOL プログラムからの Web サービスの呼び出しに関する項です。

## COBOL Web サービスの作成

Net Express で Web サービスを作成する場合は、Interface Mapping Toolkit の使用が推奨されます。この方法とその他の方法は『はじめに』の章で詳しく比較されていますが、端的に言えば Enterprise Server 環境にディプロイしない Web サービスの作成には、Microsoft SOAP Toolkit を使用できます。その場合、Net Express で COM オブジェクトを作成して、Microsoft SOAP Toolkit で Web サービスとして公開します。

『***入門書 - その他のトピック***』のチュートリアルを利用すれば、Microsoft SOAP Toolkit を使用した場合の COBOL Web サービスの作成方法の詳細を知ることができます。SOAP

Toolkit と WSDL ジェネレータの使用方法の詳細については、Microsoft SOAP Toolkit に付属しているユーザガイドを参照してください。

# COBOL からの Web サービスの利用

この項では、Microsoft SOAP Toolkit を使用して COBOL から Web サービスを呼び出す方法を説明します。

ここでは一例として、米国 xmethods 社 (www.xmethods.net) が提供している Web サービスにアクセスするサンプルを示します。同社の Web サービスはカリフォルニア州運輸交通局の Web サイトにアクセスし、同州の道路交通情報を返します。この Web サービスは **getTraffic** というメソッドを公開しており、このメソッドはコンマで区切られた一連のハイウェイ番号を受け取り、それぞれのハイウェイの道路交通情報を列挙した文字列を返します。

SOAP Toolkit には、WSDL ファイル内の情報に基づいて、Web サービスが公開しているメソッドをあたかも自身のメソッドであるかのようにクライアントアプリケーションに提供する COM があります。つまり、COM コンポーネントの呼び出しをサポートするあらゆる開発言語で、Web サービスに SOAP 要求を送信するプログラムを作成できます。そのような言語には Micro Focus COBOL も含まれます。

上記の Web サービスにアクセスするシンプルな COBOL アプリケーションの一例を次に示します。

```
$set ooctrl(+p)
class-control.
     MSSOAP is class "$OLE$MSSOAP.SoapClient"
     olesup is class "olesup".
 working-storage section.
 01  Soap object reference.
 78  WSDL-Url value
     z"http://www.xmethods.net/sd/2001/CATrafficService.wsdl".
 01  WSML-Url pic x value x"00".
 01 Traffic-Conditions pic x(5000).

 procedure division.
     invoke MSSOAP "new" returning Soap
     invoke Soap "mssoapinit" using WSDL-Url
                                    z"CATrafficService"
                                    z"CATrafficPort"
                                    WSML-Url
     invoke olesup "setDispatchType" using by value 0 size 4
     invoke Soap "getTraffic" using z"101"
     returning Traffic-Conditions
     display Traffic-Conditions
     invoke Soap "finalize" returning Soap
```

```
    stop run.
```

このプログラムを実行すると、カリフォルニア州のハイウェイ 101 の道路交通情報が出力されます。次に出力例を示します。

```
 reported as of Friday, September 14, 2001 at 11:21 .
 Slow for the Cone Zone

 US 101
    [ LOS ANGELES & VENTURA CO.'S]
     NO TRAFFIC RESTRICTIONS ARE REPORTED FOR THIS AREA.

    [ CENTRAL COAST]
     NO TRAFFIC RESTRICTIONS ARE REPORTED FOR THIS AREA.

    [ SAN FRANCISCO BAY AREA]
     NO TRAFFIC RESTRICTIONS ARE REPORTED FOR THIS AREA.

    [ NORTHWEST CALIFORNIA]
     1-WAY CONTROLLED TRAFFIC AT VARIOUS LOCATIONS FROM 2.2 MI SOUTH
     TO 8.8 MI NORTH OF THE JCT OF SR 1 (MENDOCINO CO) FROM 0800
     HRS TO 1600 HRS THRU 9/14/01 DUE TO MAINTENANCE
```

このプログラムは、SOAP Toolkit を使用して Web サービスにアクセスするあらゆる COBOL プログラムに共通の基本構造を持っています。具体的なポイントを次に示します。

- ooctrl(+p) コンパイラ指令 ― この指令によって、**invoke** 文に指定した各パラメータの種類に関する情報を、COBOL ランタイムシステムで利用できるようにしています。
- **$OLE$MSSOAP.SoapClient** クラスの宣言を含む**クラス制御**節の定義 ― プリフィックスの $OLE$ によって、MSSOAP.SoapClient クラスが標準の COBOL クラスではなく COM コンポーネントであることを COBOL ランタイムシステムに示しています。
- **オブジェクト参照**型の変数の宣言によるクラスへの参照の取得 ― このプログラムでは、該当する変数は **Soap** です。
- **new** メソッドの呼び出しによる **MSSOAP.SoapClient** コンポーネントの新しいインスタンスの生成 ― この処理を行っているのは次の行です。

  ```
      invoke MSSOAP "new" returning Soap
  ```

- **mssoapinit** メソッドの呼び出し ― サービスの WSDL ファイルの URL、サービスの名前、およびポート名を指定して呼び出しています。 サービス名とポート名は WSDL ファイル内で指定されています。 Mssoapinit によって、Web サービス内の各メソッドを、あたかも **MSSOAP.SoapClient** クラスのメソッドのように呼び出すことが可能になります。
- Web サービス内のメソッドによる必要な処理の実行 ― このプログラムでは、次の行です。
- ```
      invoke Soap "getTraffic" using z"101"
  ```
  ```

```
                                    returning Traffic-Conditions
```

このメソッドは NULL で終端する文字列 (z"101") 1 つを引数として受け取り、処理結果を含む文字列を返します。使用できるパラメータの種類の詳細は、『*分散コンピューティング*』の『OLE データ型』を参照してください。

- **finalize** メソッドによる Web サービスの使用終了。

**olesup** クラスの宣言と次の行の存在はやや不可解に見えるかもしれません。

```
invoke olesup "setDispatchType" using by value 0 size 4
```

この行が必要になるのは、名前の先頭に get が付くメソッド (**getTraffic**) を使用しているためです。Micro Focus COBOL では通常、COM コンポーネントにアクセスするメソッド名の先頭に get や put が付く場合、当該メソッドはプロパティ GET またはプロパティ PUT として解釈されます。**setDispatchType** メソッドの呼び出しは、**getTraffic** が通常のメソッド呼び出しとして扱われるように、このデフォルトの振る舞いを無効化しています。

この行が必要になるのは、メソッド名の先頭に get または put が付く場合だけです。その他のメソッド名の場合、この行をメソッド呼び出しの直前に挿入する必要はありません。**olesup** クラスと **setDispatchType** メソッドの詳細については、Net Express のオンラインマニュアルを参照してください。

# 第 7 章 Java と COBOL の連携

この章では、Java プログラムと COBOL プログラムの間で、一方から他方を呼び出すための各種の方法を紹介します。

## 概要

COBOL と Java は、いくつかの方法で連携させることができます。Interface Mapping Toolkit や、各種のウィザードおよび GUI ツールを使用して Java から COBOL を呼び出すインターフェイスを作成できるほか、標準で付属しているサポートライブラリを使用して、呼び出しを独自にコーディングすることも可能です。Java では RMI (Remote Method Invocation) や JNDI (Java Naming and Directory Interface) といった技術がサポートされており、これらの技術を利用して、さまざまな種類のマシンにオブジェクトを分散配置できます。

COBOL と Java を連携させる方法を次に示します。

- Java インターフェイスを手続き型 COBOL にマッピングする。

  Interface Mapping Toolkit を使用し、COBOL プログラムを変更することなく公開できます。作成したインターフェイスをディプロイする際には、生成された EJB を J2EE 準拠のアプリケーションサーバ、マッピング情報を Enterprise Server にそれぞれディプロイします。実行時には、J2EE アプリケーションサーバ環境で動作している EJB が COBOL を呼び出し、Enterprise Server の制御下で COBOL プログラムが実行されます。詳細は『*Java インターフェイスのマッピングとリソースアダプタの使用*』の章を参照してください。

- com.microfocus.cobol.RuntimeSystem クラスのサポート関数を使用して Java から手続き型 COBOL を呼び出す。

  このクラスは CobolBean クラスを含むように拡張されており、手続き型 COBOL から JNI (Java Native Interface) 経由で Java を呼び出すことも可能です。この方法は、Java 2 Standard Edition (J2SE) との組み合わせに最適です。詳細は『*Java からの手続き型 COBOL の呼び出し*』の章を参照してください。

- Object COBOL から Java オブジェクトドメインを介して Java を呼び出す。

  この方法は、Java 2 Standard Edition (J2SE) との組み合わせに最適です。詳細は『*Java からの Object COBOL の呼び出し*』の章を参照してください。

- Java から Object COBOL を呼び出す。

  クラスウィザードを使用して、Object COBOL クラス内の各メソッドに対応する関数を含む Java ラッパーを作成します。この方法は、Java 2 Standard Edition (J2SE) との

組み合わせに最適です。詳細は『*Object COBOL からの Java の呼び出し*』の章を参照してください。

Interface Mapping Toolkit を使用して公開した COBOL プログラムの実行には Enterprise Server for Windows が必須ですが、その他の方法で COBOL と Java のプログラムを連携させるアプリケーションでは、Enterprise Server for Windows の代わりに Application Server for Net Express (または Application Server for Server Express) を使用できます。

Java 言語では、COBOL で使用されるデータ型とは異なる、Java 用のデータ型が定義されています。COBOL から Java (または Java から COBOL) が呼び出されると、COBOL ランタイムシステムによって COBOL と Java 間でデータ型が自動的に変換されます。直接変換できない COBOL データ型 (編集済みのフィールドなど) は、環境によっては文字列型に変換されることがあります。詳細は『*Java データ型*』の章を参照してください。

# Java と COBOL の環境のセットアップ

Java アプリケーションを実行するマシンには、Java ランタイムシステムをインストールする必要があります。Java と COBOL の両方を使ってアプリケーションを開発する場合は、Java 開発環境も必要になります。開発環境としては、Sun が提供している Java SDK (Software Development Kit) のほか、Sun または Microsoft のランタイム環境に基づくあらゆる Java IDE が使用可能です。

Net Express は現在、Windows 環境で次の Java ランタイムシステムをサポートしています。

- Sun Java SDK 1.2.2、1.3 (開発環境)
- Sun Java Runtime Environment (JRE) 1.2.2、1.3、1.4 (アプリケーションと共に配布できるランタイム環境)

COBOL プログラムと Java プログラムを作成して、互いに連携させる場合には、COBOL や Java のランタイムシステムで使用される次の環境変数を、あらかじめ設定しておく必要があります。

- **COBJVM**

  COBOL プログラムから Java の呼び出しを行う場合には、使用している Java ランタイムシステムを COBOL のランタイムシステムに通知する必要があります。この通知を行うには、環境変数 COBJVM を次の値に設定します。

  - SUN (Sun のランタイムシステム)

  次に設定例を示します。

  ```
  set cobjvm=SUN
  ```

- **PATH**

Sun の Java ランタイムシステムを使用している場合には、**jvm.dll** ファイルを含むディレクトリをシステムの PATH に登録する必要があります。このファイルの場所は、使用している JDK (Java Development Kit) のバージョンによって異なります。次に設定例を示します。

```
set path=%path%;c:\jdk1.3.1\jre\bin\subdirectory
```

*subdirectory* の値は **client**、**classic**、**hotspot**、**server** のいずれかです。

この環境変数にディレクトリを登録すると、その中に含まれている **jvm.dll** ファイルが COBOL ランタイムシステムで検出されるようになります。**jvm.dll** ファイルは、Sun の Java ランタイムシステムに含まれる他のファイルと依存関係にあるため、他の場所には移動しないでください。

- **CLASSPATH**

  Java プログラムから COBOL を呼び出す場合には、COBOL ランタイムシステムとのインターフェイスを実装する Java クラスが必要になります。**mfcobol.jar** を含むディレクトリを、Java ランタイムシステムの環境変数 CLASSPATH に登録してください。CLASSPATH には、カレントディレクトリ (.) も登録されている必要があります。次に設定例を示します。

  ```
  set classpath=%classpath%;.;c:\program files\Micro Focus\net
  express\base\bin\mfcobol.jar
  ```

  Java のクラスパスは、Java プログラムを実行するコマンド行で -classpath スイッチを使用して設定することも可能です。次に設定例を示します。

  ```
  java -classpath ".;c:\program files\Micro Focus\net
  express\base\bin\mfcobol.jar;%CLASSPATH%" MyClass
  ```

# Java を呼び出す COBOL プログラムのコンパイル

Java を呼び出す COBOL プログラムは、必ず次の指令を指定してコンパイルしてください。

```
ooctrl(+p-f)
```

この指令によって、次の 2 つの処理が実行されます。

- COBOL ランタイムシステムが COBOL と Java ドメイン間でデータを正しく変換できるように、必要な文を呼び出すための型情報を追加する。
- コンパイラによって呼び出されるメソッドの名前の小文字変換を防止する (Java のメソッド名は大文字、小文字が区別される)。

## CobolBean インターフェイス経由で呼び出される COBOL プログラムのコンパイル

Java から CobolBean.cobcall*() メソッドで呼び出される COBOL プログラムは、必ず DATA-CONTEXT コンパイラ指令付きでコンパイルしてください。この指令を指定すると、作成された CobolBean のインスタンスごとに、ランタイムシステムが新しいアプリケーション記憶域を作成できるようになります。

## Net Express での Java プログラムのコンパイル

Java プログラムから呼び出す Object COBOL クラスを Net Express IDE で作成すると、クラスウィザードによって Object COBOL 用の Java ラッパークラスが Net Express プロジェクトに追加されます。この Java ラッパークラスは、プロジェクトのリビルド時に IDE によってコンパイルされます。ただし、この機能を有効にするには、**net express\base\bin** ディレクトリ内の **mfj.cfg** ファイルを編集して、Java コンパイラのファイル名を絶対パスで登録しておく必要があります。

**net express\base\bin** ディレクトリに **mfj.cfg** が存在しない場合は、新しく作成してください。Net Express の IDE で Java プログラムを最初にコンパイルするときに、使用している PC のレジストリか、環境変数 PATH に Java コンパイラの場所が登録されている場合、**mfj.cfg** ファイルは Net Express によって自動的に生成されます。

**mfj.cfg** ファイルには、使用する Java コンパイラの場所のほか、Java コンパイラのコマンド行引数も指定できます。次に **mfj.cfg** ファイル内での指定の一例を示します。

```
d:\jdk\bin\javac.exe -verbose
```

このように指定すると、Net Express IDE で Java プログラムをコンパイルするたびに、**d:\jdk\bin** ディレクトリ内の **javac.exe** が -verbose オプション付きで実行されます。

## マルチスレッドランタイムシステムへのリンク

Java から呼び出す COBOL プログラムは、マルチスレッドのランタイムシステムにリンクする必要があります。

Net Express の「**プロジェクトのビルド設定**」ダイアログボックスで「**リンク**」タブを選択し、「**マルチスレッド**」をクリックしてください。プログラムのデバッグ時には、[**アニメート**] メニューの [**設定**] をクリックし、さらに [**マルチスレッドランタイムの使用**] をクリックします。

## Java VM 未サポートエラー

Java から COBOL プログラムを呼び出すと、Object COBOL の Java サポートによって一連の **cbljvm_*.dll** モジュールのいずれか 1 つが、Java 仮想マシン (Java VM) とのインターフェ

イスとしてロードされます。実際にロードされるモジュールは、使用している Java VM によって左右されます。すなわち、Java プログラムを実行する Java VM の名前が照会され、該当するモジュールが選択されます。

使用している Java VM がサポート対象に含まれていない場合、重大エラー (Java 仮想マシン未サポートエラー) が発生します。Java システムプロパティの 1 つ (com.microfocus.cobol.cobjvm) に、サポートされている Java VM の名前を設定すれば、その Java VM を強制的にロードさせることが可能です。たとえば、**cbljvm_sun.dll** に含まれている Sun の Java VM 用のサポートモジュールをロードさせるには、このプロパティの値を sun に設定します。

このプロパティは、次のように Java プログラムを実行するコマンド行で指定できます。

```
java -Dcom.microfocus.cobol.cobjvm=name class
```

*name* には、使用するサポートモジュールの名前を指定します。たとえば、Sun の Java VM のサポートモジュールをロードして、Java プログラム myclass を実行する場合には、次のように入力します。

```
java -Dcom.microfocus.cobol.cobjvm=sun myclass
```

Java 仮想マシン未サポートエラーが発生した場合には、Sun の Java VM のサポートモジュールを指定してください。Sun の Java VM と名前が異なるだけで、実質的には同じ Java VM は少なくありません。

---



---

# 第 8 章 : Java インターフェイスのマッピングとリソースアダプタの使用

この章では、Micro Focus のリソースアダプタを使用するアプリケーションの開発手順を説明します。

## Java インターフェイスのマッピングの概要

手続き型 COBOL で作成したプログラムは、Interface Mapping Toolkit を使用すれば EJB として公開できます。

最初にインターフェイスマッパーでインターフェイスを COBOL プログラムにマッピングします。COBOL のエントリポイントまたはプログラム ID、および新しいインターフェイスに使用するパラメータを選択します。 Interface Mapping Toolkit によってマッピング情報がすべて保存され、COBOL サービスの呼び出しに使用できる EJB が生成されます。

続いて、作成したサービスインターフェイスをディプロイします。ディプロイする具体的な項目は次のとおりです。

- マッピング情報と COBOL プログラム ― これらの項目はエンタープライズサーバにディプロイします。これらの作業には Interface Mapping Toolkit のディプロイツールが使用できるほか、手作業でもディプロイできます。
- 生成された EJB ― J2EE 準拠のサードパーティ製アプリケーションサーバ (WebSphere、WebLogic など) にディプロイします。

  サービスインターフェイスは、生成された EJB をディプロイしない環境や、J2EE 準拠のアプリケーションサーバを使用しない環境でも利用可能です。ただし、その場合は接続管理が行われず、リソースアダプタを呼び出すコードを独自に作成する必要があります。

- リソースアダプタ ― EJB からエンタープライズサーバへのメッセージ送信を可能にするコンポーネント。Net Express に標準で付属しています。リソースアダプタは J2EE 準拠のアプリケーションサーバにディプロイします。

最後に、EJB を呼び出す Java クライアント (JSP など) を作成します。

実行時には、Java クライアントアプリケーションから要求を送信し、生成された EJB を呼び出します。EJB は受信した要求をリソースアダプタを介してエンタープライズサーバに渡し、公開されている COBOL プログラムがエンタープライズサーバによって実行されます。マッピング情報は実行時に、公開されている COBOL プログラムとの情報のやり取りに使用されます。COBOL プログラムからの応答はエンタープライズサーバによって、リソースアダプタを介して EJB に送信され、そこからクライアントに返されます。

ディプロイ方法の詳細については、『Enterprise Server ディプロイガイド』を参照してください。

# リソースアダプタの概要

Net Express には複数のリソースアダプタ (J2EE コネクタ) が付属しており、これらのリソースアダプタによって、J2EE アプリケーションサーバ上の EJB とエンタープライズサーバ上の COBOL プログラム間の通信が可能になります。

EJB にエンタープライズサーバ上の COBOL プログラムとの通信を開始させるには、J2EE アプリケーションサーバ上にリソースアダプタをディプロイする必要があります。詳細については、『Enterprise Server ディプロイガイド』の『EJB とリソースアダプタ』の章を参照してください。

Micro Focus のリソースアダプタは JCA (Java Connector Architecture) に準拠しており、個々のアダプタが API として CCI (Common Client Interface) を実装しています。JCA と CCI はどちらも Sun が策定した規格であり、EJB やその他の J2EE コンポーネントは準拠が義務付けられています。

要求がリソースアダプタを経由して送信される基本的な過程は次のとおりです。

1. 生成された EJB にクライアントからの要求が着信します。
2. 生成された EJB が CCI の実装を通じて、着信した要求をリソースアダプタに送信します。
3. リソースアダプタが Micro Focus バイナリプロトコルを使用して、受信した要求からバイナリ要求を生成し、エンタープライズサーバに送信します。
4. エンタープライズサーバはリソースアダプタからの要求の着信を監視しており、要求を検出すると適切な処理を実行します。リスナーとその設定については、『構成と管理』を参照してください。

Micro Focus のリソースアダプタは API として CCI を使用するため、生成された EJB を使用しないで、リソースアダプタを直接呼び出すことも可能です。後述する『J2SE 環境における管理されない状態での接続』の項を参照してください。

# 生成された EJB の呼び出し

生成された EJB を呼び出して、COBOL サービスに必要なパラメータを EJB に渡すには、Java でクライアントソフトウェアを作成する必要があります。このクライアントは JSP やサーブレット、またはその他のモジュールとして実装できます。

EJB インターフェイスは Interface Mapping Toolkit で定義されており、通常は Java 開発ツールに EJB をインポートして、そこでインターフェイスを調べます。その際には特に、出力パラメータに注意してください。マッピングされたインターフェイスで (グループ項目や対応するカスタムレコードではなく) 複数の出力パラメータが渡される場合、これらの出力パラメータは EJB インターフェイス内のコンテナオブジェクトを通じて返されます。Java 開発ツールでは、そのような出力パラメータの有無も確認できます。

実行時には、Java クライアントソフトウェアが EJB を呼び出し、EJB とエンタープライズサーバ間の情報のやり取りを通じて COBOL サービスが実行されます。

# J2SE 環境における管理されない状態での接続

Java から COBOL プログラムにメッセージを送信する手段としては、Interface Mapping Toolkit で EJB を生成する方法が最も簡単です。この方法では、生成した EJB を J2EE 準拠のアプリケーションサーバ環境にディプロイし、リソースアダプタとエンタープライズサーバ間の接続をアプリケーションサーバによって管理します。

リソースアダプタを呼び出してエンタープライズサーバに要求を送信するコードを独自に作成すれば、EJB と J2EE アプリケーションサーバを使用しないで、COBOL プログラムにメッセージを送信することができます。その場合、接続は管理されない状態になります。作成したコードは、J2EE アプリケーションサーバの環境ではなく、J2EE サーバのライブラリを含む Java 2 Standard Edition (J2SE) 環境で実行されます。

管理されない状態での接続をディプロイするために必要な設定は、Micro Focus のクラス群を含むディレクトリの classpath への登録だけです。詳細については、『Enterprise Server ディプロイガイド』の『EJB とリソースアダプタ』の章を参照してください。

## シンプルな CCI プログラムの作成

Java で作成するクライアントは、CCI に準拠する必要があります。CCI では、リソースアダプタから Enterprise Server などの EIS (Enterprise Information Server) に接続するために必要な API が定義されています。また、Micro Focus のリソースアダプタ専用の拡張クラスも使用可能です。

クライアントによる処理の大まかな流れは次のとおりです。

1. Java クラスと Micro Focus クラスをインポートする。
2. 接続を確立する。
3. 対話用のオブジェクトを作成する。
4. 対話の設定 (InteractionSpec) を行う。
5. 対話通信を実行する。
6. 結果を取得する。
7. 接続を閉じる。

CCI で定義されている各クラスと、それらのメソッドの一部を次に示します。

- ManagedConnectionFactory クラス ― createConnectionFactory()
- ConnectionFactory クラス ― get Connection()、getRecordFactory()
- Connection クラス ― createInteraction()、getResultSetInfo、close()
- Interaction クラス ― getConnection()、execute()、getWarnings()
- Record クラス ― getRecordName()、clone()
- InteractionSpec クラス ― SYNC-SEND、SYNC_RECEIVE ....

com.microfocus.cobol.connector.cci package には、Micro Focus による CCI の拡張が含まれています (この拡張については、NetExpress\base\docs\mfcoboldocs.zip 内の『J2EE Connector Class Library Reference』で説明されています)。
com.microfocus.cobol.connector.cci package には、次のようなクラスとメソッドがあります。

- CobolConnectionFactory クラス ― getConnection()、getRecordFactory()
- CobolInteraction クラス ― execute()、getWarnings()
- CobolInteractionSpec クラス ― getInteractionVerb()、setArgument()
- CustomRecord クラス ― getRecordName()、clone()、hashCode()
- IndexedRecord クラス ― lastIndexOf()、contains()、lastElement()

## リソースアダプタに接続するプログラムのサンプル

以下のいくつかの項では、COBOL サービスを要求する Java プログラムのコード例を示します。この Java プログラムは、CCI のインターフェイスを使用してリソースアダプタとの間で情報をやり取りし、リソースアダプタによって COBOL サービスに要求を渡します。また、COBOL グループ項目を表す CCI カスタムレコードの使用方法も示しています。この例では、Calculate という名前の COBOL プログラムからエントリポイント ADD を介し、再利用可能レコード Calculator を使用して、add サービスがマッピングされます。

COBOL の Calculate プログラムは、インターフェイスマッパーで Java インターフェイスにマッピングする必要があります。Calculator パラメータを「再利用マッピング」ペインにドラッグし、続いて add オペレーションの「インターフェイスフィールド」ペインにドラッグします。subtract、divide、multiply の各オペレーションでも、同じ操作を行います。このようにインターフェイスをマッピングするのは、アプリケーションサーバで管理されないクラスによって、CCI カスタムレコードの使用方法を示しているためです。

このプログラム例では、次の各 Java パッケージに含まれるクラスを使用します。

| | |
|---|---|
| javax.resource.cci | J2EE のコネクタアーキテクチャ仕様で定義されている CCI インターフェイス群 |
| com.microfocus.cobol.connector.spi | SPI (Service Provider Interface) インターフェイスを実装する Micro Focus リソースアダプタ専用のクラス群 |
| com.microfocus.cobol.connector.cci | CCI インターフェイスを実装する Micro Focus リソースアダプタ専用のクラス群 |

これらのパッケージは、コードにインポートすることが推奨されます。次のような文をコード内に記述してください。

```
import com.microfocus.cobol.connector.spi.*;
import com.microfocus.cobol.connector.cci.*;
import javax.resource.cci.*;
```

## サービスの一例

次にサービスのコード例を示します。コードに続いて、主な行を詳しく説明しています。

```
1       try {
2         // CobolNoTxManagedConnectionFactory のインスタンスを生成
3        mcf = new CobolNoTxManagedConnectionFactory();
4        // 必要なフィールドを設定
5        mcf.setServerHost("localhost");
6        mcf.setServerPort("9003");

7        // ConnectionFactory のインスタンスを生成 (JNDI は使用しない)
8        cxf = (javax.resource.cci.ConnectionFactory)
                mcf.createConnectionFactory();

9       // Cobol Connection ハンドルを取得
10       connection = cxf.getConnection();
11       initialize(connection, cxf, true);

12       // 対話通信のセットアップ
13       interaction = connection.createInteraction();

14       // 新しい対話設定の作成
15       CobolInteractionSpec iSpec =
              new CobolInteractionSpec();
16       iSpec.setFunctionName("myservice.add");

17       javax.resource.cci.RecordFactory rf =
              cxf.getRecordFactory();
18       Calculator calc = new Calculator();
19       calc.setArg1(new java.math.BigDecimal(10));
20       calc.setArg2(new java.math.BigDecimal(20));
21       iSpec.setArgument(0, com.microfocus.cobol.
              RuntimeProperties.BY_REFERENCE);

22       interaction.execute(iSpec, calc, calc);
23       System.out.println(
              "Input - Arg 1 was" + calc.getArg1());
24       System.out.println(
              "Input - Arg 2 was" + calc.getArg2());
25       System.out.println(
              "Result was" + calc.getResult());
26       System.out.println(
              "Memory was" + calc.getStorage());

28       interaction.close();
29       connection.close();
30       } catch( javax.resource.ResourceException re) {
```

```
31          re.printStackTrace();
32          Exception le = re.getLinkedException();
33        }
```

3〜6 行目

```
mcf = new CobolNoTxManagedConnectionFactory();
```

前述した ManagedConnectionFactory クラスのインスタンスを生成しています。続いて、Java プログラムとリソースアダプタ間の管理されない接続を対象に、このオブジェクトを使って次の属性を設定します。

- `mcf.setServerHost("localhost");`

  リソースアダプタが位置しているマシンの名前。

- `mcf.setServerPort("9003");`

  エンタープライズサーバがリソースアダプタからのメッセージ着信を監視するポート。

8 行目

```
cxf = (javax.resource.cci.ConnectionFactory)
          mcf.createConnectionFactory();
```

3 行目で生成した CobolNoTxManagedConnectionFactory を使用して javax.resource.cci.ConnectionFactory クラスをインスタンス化しています。

10 行目

```
connection = cxf.getConnection();
```

javax.resource.cci.Connection クラスをインスタンス化しています。ConnectionFactory オブジェクトの getConnection メソッドを呼び出すと、CobolNoTxManagedConnectionFactory の設定が反映された Connection インスタンスのハンドルが取得されます。この接続ハンドルは常に 1 つの ManagedConnection にマッピングされます。

11 行目

```
initialize(connection, cxf, true);
```

接続を初期化しています。この処理は省略できません。引数 true はエンタープライズサーバ上の COBOL プログラムの状態を指定しています。呼び出すたびにプログラムを初期状態で実行する場合は true、サービス呼び出しの間、同じ状態を保持させるには false に設定します。

13 行目

```
interaction = connection.createInteraction();
```

Connection オブジェクトの createInteraction() メソッドを呼び出し、javax.resource.cci.Interaction をインスタンス化しています。 Interaction オブジェクトの生成には Connection オブジェクトを使用します。生成した Interaction オブジェクトでエンタープライズサーバとの通信を行います。

15 行目

```
CobolInteractionSpec iSpec =
    new CobolInteractionSpec();
```

必要な対話通信の種類に応じて、専用のクラスをインスタンス化し、設定します。 この例では、com.microfocus.cobol.connector.cci.CobolInteractionSpec クラスを使用しています。このクラスで、パラメータを渡す要求と方法を作成します。 このクラスのインスタンスに、関数の名前と、索引レコード内の各引数の方向 (入力、出力、入出力) または引数 direction (カスタムレコードの場合) が保持されます。

16 行目

```
iSpec.setFunctionName("myservice.add");
```

エンタープライズサーバで実行するプログラムの名前を、setFunctionName メソッドで指定しています。

18～21 行目

```
Calculator calc = new Calculator();
calc.setArg1(new java.math.BigDecimal(10));
calc.setArg2(new java.math.BigDecimal(20));
iSpec.setArgument(0,
      com.microfocus.cobol.
      RuntimeProperties.BY_REFERENCE);
```

カスタムレコードクラスのオブジェクトを生成します。 単独のカスタムレコードを引数として設定する場合は、このコードで十分です。 カスタムレコードは EJB の生成時に自動的に生成され ソースファイルが myproject\repos\myservice.deploy\packageName に格納されています。

複数のカスタムレコード、またはカスタムレコードと Java の基本データ型を 1 つの引数として渡す場合には、索引レコードを使用する必要があります。 たとえば、整数型の値とカスタムレコードをパラメータとして渡す場合には、次のコードを使用します。

```
    javax.resource.cci.RecordFactory rf =
          cxf.getRecordFactory();
```

```
javax.resource.cci.IndexedRecord iRec =
      rf.createIndexedRecord("IndexedIn");
javax.resource.cci.IndexedRecord oRec =
      rf.createIndexedRecord("IndexedOut");
iRec.add(new Integer(5));
iRec.add(calc);
iSpec.setArgument(0,
      com.microfocus.cobol.
      RuntimeProperties.BY_REFERENCE);
```

このコードによって引数の方向が設定されます。各引数には 0 を基数とする連番が付けられ、この番号が最初の引数として setArgument に渡されます。方向の有効値は次のとおりです。

```
com.microfocus.cobol.RuntimeProperties.BY_REFERENCE
com.microfocus.cobol.RuntimeProperties.BY_VALUE
com.microfocus.cobol.RuntimeProperties.OUTPUT_ONLY
```

22 行目

```
interaction.execute(iSpec, calc, calc);
```

対話通信を実行します。Interaction オブジェクトの execute() メソッドを呼び出すと、要求がエンタープライズサーバに送信されます。

28～29 行目

```
interaction.close();
```

最後に Interaction と Connection をクローズします。Connection オブジェクトの close() は接続を終了するメソッドです。このメソッドを実行すると、リソースアダプタとの接続が解除されます。このメソッドを実行する前に、必ず Interaction オブジェクトの close() メソッドを実行してください。

## initialize メソッドの一例

initialize メソッドのコードの一例を次に示します。

```
private void initialize(
                javax.resource.cci.Connection con,
                javax.resource.cci.ConnectionFactory cf,
                boolean isInitial) {
      try {
      javax.resource.cci.Interaction ix =
           con.createInteraction();
      com.microfocus.cobol.connector.cci.CobolInteractionSpec
           iSpec = new
```

```
            com.microfocus.cobol.connector.cci.CobolInteractionSpec();
        iSpec.setFunctionName("initialize");
        javax.resource.cci.RecordFactory rf =
            cf.getRecordFactory();
        javax.resource.cci.IndexedRecord irec =
            rf.createIndexedRecord("beanArgs");
        irec.add(new Boolean(isInitial));
        javax.resource.cci.Record orec = ix.execute(iSpec, irec);
        ix.close();
    } catch(javax.resource.ResourceException ex) {
        throw new javax.ejb.EJBException(
            "initialize threw ResourceException: ", ex);
    }
}
```

## カスタムレコードの一例

上記のサンプルでは、COBOL サービス (myservice.add) でグループレコード Calculator を使用しています。リソースアダプタの CCI では、このグループレコードはカスタムレコードとして表されます。このグループレコードに対応するカスタムレコードクラス、Calculator.java をのコード例を次に示します。

```
/*****************************************************
 This is a file generated by Micro Focus Net Express 4.0
 This file represents the Custom Class for Calculator
 *****************************************************/

public class Calculator extends
       com.microfocus.cobol.connector.cci.CustomRecord {

    private java.math.BigDecimal arg1 =
            java.math.BigDecimal.valueOf(0);
    private java.math.BigDecimal arg2 =
            java.math.BigDecimal.valueOf(0);
    private java.math.BigDecimal result =
            java.math.BigDecimal.valueOf(0);
    private java.math.BigDecimal storage =
            java.math.BigDecimal.valueOf(0);
    public Calculator() {
    }
    public java.math.BigDecimal getArg1() {
        return arg1;
    }
    public void setArg1(java.math.BigDecimal ___p) {
        arg1 = ___p ;
    }
```

```
    public java.math.BigDecimal getArg2() {
        return arg2;
    }
    public void setArg2(java.math.BigDecimal ___p) {
        arg2 = ___p ;
    }
    public java.math.BigDecimal getResult() {
return result;
    }
    public void setResult(java.math.BigDecimal ___p) {
        result = ___p ;
    }
    public java.math.BigDecimal getStorage() {
        return storage;
    }
    public void setStorage(java.math.BigDecimal ___p) {
        storage = ___p ;
    }
    public Object[] getParameters() {
        Object[] objs = new Object[4];
        objs[0] = arg1;
        objs[1] = arg2;
        objs[2] = result;
        objs[3] = storage;
        return objs;
    }
    public void setParameters(Object[] objs) {
        arg1 = (java.math.BigDecimal)objs[0];
        arg2 = (java.math.BigDecimal)objs[1];
        result = (java.math.BigDecimal)objs[2];
        storage = (java.math.BigDecimal)objs[3];
    }
}
```

## COBOL プログラムのサンプル (Calculator)

サービスのサンプルで使用される COBOL プログラム、CALCULATOR のコードを次に示します。

```
      $set intlevel(4)
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CALCULATE.

       ENVIRONMENT DIVISION.

       DATA DIVISION.
```

```
WORKING-STORAGE SECTION.
01 CALMEMORY           pic s9(9) comp-5 VALUE 0.

LINKAGE SECTION.
01 CALCULATOR.
   05 ARG1         pic S9(19)V9(19) comp-3.
   05 ARG2         pic S9(19)V9(19) comp-3.
   05 RESULT       pic S9(19)V9(19) comp-3.
   05 STORAGE      pic S9(19)V9(19) comp-3.

procedure division.
    exit program.

entry "ADD" using CALCULATOR.
    move ARG1 to RESULT
    add  ARG2 to RESULT
    add  RESULT to CALMEMORY
    move CALMEMORY TO STORAGE
    exit program.

entry "SUBTRACT" using CALCULATOR.
    move ARG1 to RESULT
    subtract ARG2 from RESULT
    add  RESULT to CALMEMORY
    move CALMEMORY TO STORAGE

    exit program.

entry "MULTIPLY" using CALCULATOR.
    move ARG1 to RESULT
    multiply ARG2 by RESULT
    add  RESULT to CALMEMORY
    move CALMEMORY TO STORAGE

    exit program.

entry "DIVIDE" using CALCULATOR.
    move ARG1 to RESULT
    divide ARG2 into RESULT
    add  RESULT to CALMEMORY
    move CALMEMORY TO STORAGE

    exit program.
```

# 第 9 章 : Java からの手続き型 COBOL の呼び出し

この章では、com.microfocus.cobol.RuntimeSystem クラスに含まれる COBOL サポートを使用して、Java から手続き型 COBOL のレガシープログラムにアクセスする方法を説明します。

## 9.1 概要

手続き型の COBOL プログラムは、Java プログラムや EJB から呼び出すことができます (COBOL で作成した EJB に限らず、さまざまな言語で作成した EJB から呼び出し可能です) 。この呼び出しに使用できる各種の方法については、『*Java と COBOL の連携*』の章で概要を示しています。

この章で説明するのは、com.microfocus.cobol.RuntimeSystem クラスと CobolBean クラスに含まれる COBOL サポートを使用して呼び出しを行う方法です。CobolBean クラスは、実際には RuntimeSystem クラスの拡張であり、一連の RuntimeSystem.cobcall*() メソッドを使用します。ここで紹介する手法は、Object COBOL と Enterprise Server を使用せずに、Java や EJB から COBOL プログラムを呼び出すときに使用します。

Java オブジェクトには JNI (Java Native Interface) を介するか、Object COBOL の Java ドメインを使用してアクセスできます。

com.microfocus.cobol.RuntimeSystem クラスの COBOL サポートは一連の関数から構成されており、COBOL プログラムのロード、呼び出し、およびキャンセルを実行できます。また、Java 配列を使用して、COBOL プログラムにパラメータを渡すことも可能です。com.microfocus.cobol.RuntimeSystem クラスの関数群が、配列からデータを取り出し、そのデータを COBOL プログラムに渡します。図 9-1 は、Java プログラムから COBOL プログラムを呼び出し、2 つのパラメータを渡す処理を示しています。

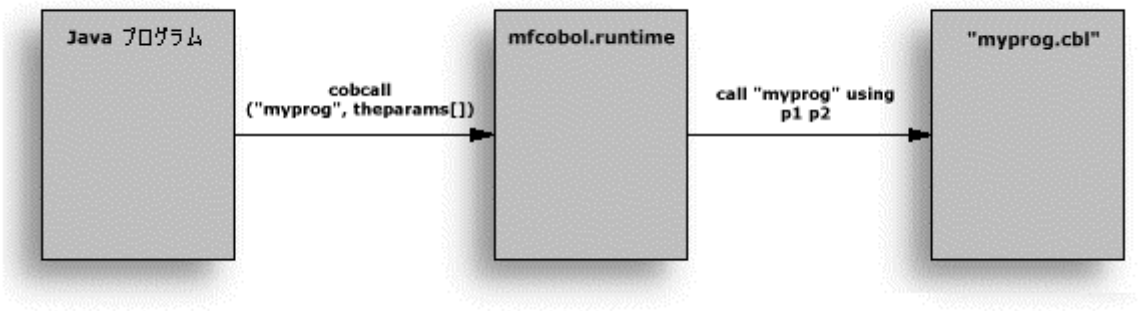



**図 9-1 com.microfocus.cobol.RuntimeSystem クラスを介した Java からの COBOL プログラムの呼び出し**

ここで説明する情報を活用するには、Java 言語に関する基本レベル以上の知識が必要になります。Java 言語の基本的な知識を得るには、Net Express Links のリンク先にある Sun の Java Web サイトが役立ちます。

COBOL と Java の環境設定の方法については、『*Java と COBOL の連携*』の章の『*Java と COBOL の環境のセットアップ*』を参照してください。

# 9.2 Java プログラムの作成

Java プログラムは **mfcobol.jar** ファイル内の **RuntimeSystem.class** で定義されている関数群を使用して COBOL エントリポイントを呼び出します。このクラスには cobcall_*returntype*() という名前の一連の関数が含まれています (*returntype* の部分は、呼び出し対象の COBOL プログラムまたはエントリポイントによって返される値のデータ型によって変化します。たとえば、整数型を返す COBOL プログラムの呼び出しには cobcall_int() を使用します)。

Java プログラムで COBOL サポートを使用するには、Java ソースファイルの先頭に次の文を記述してください。

```
import com.microfocus.cobol.*;
```

## 9.2.1 マルチスレッド関連の留意点

Java ランタイムシステムはマルチスレッド環境であるため、Java から呼び出す COBOL プログラムは、呼び出し元の Java プログラムがマルチスレッドプログラムかどうかに関係なく、常にマルチスレッドの COBOL ランタイムシステムとリンクする必要があります。COBOL プログラムをマルチスレッドの Java プログラムから呼び出す場合には、使用中の COBOL データに他のスレッドがアクセスしてデータが壊れてしまわないように対処する必要があります (あるいは、CobolBean インターフェイスを使用して、COBOL プログラムの作業場所節を CobolBean クラスの特定インスタンスと関連付ける方法を検討します)。

この問題には、次の方法で対処できます。

- COBOL プログラムを SERIAL 指令付きでコンパイルする。

  SERIAL を使用すると、複数のスレッドによる COBOL コードへのアクセスが、COBOL ランタイムシステムによってシリアル化され、プログラムにアクセスできるスレッドが一度に 1 つだけに限定されます。これは最も安全性の高い選択肢ですが、潜在的なオーバーヘッドも最も大きくなります。この方法は、COBOL プログラムを介して共有リソース (共有プリンタなど) にアクセスする場合や、Java から呼び出した COBOL プログラムが、マルチスレッドに対応していない他の COBOL プログラムを呼び出す場合に適しています。

- COBOL プログラムを REENTRANT(2) 指令付きでコンパイルする。

REENTRANT(2) を使用すると、互いに独立したユーザデータ領域と FD ファイル領域が COBOL ランタイムシステムによってスレッドごとに割り当てられ、プログラム内での競合の発生やデータ破損が防止されます。ただし、REENTRANT(2) を指定したプログラムでは、マルチスレッドプログラムから呼び出されたり、他の共有リソースにアクセスする際のスレッドセーフが保証されません。したがって、そのような場合の安全性は SERIAL に劣ります。REENTRANT(2) 指令では、スレッドが直前のスレッドの終了を待たずに実行されるため、パフォーマンス面では SERIAL より優れています。

- COBOL プログラムで作業場所節のかわりにスレッドローカルの記憶域節 (Thread-Local-Storage) を使用する。

  Thread-Local-Storage はスレッドごとに割り当てられるため、スレッドで使用しているデータに他のスレッドがアクセスし、データが破損する事態が回避されます。この方法は効率面でも優れていますが、レガシーコードに常に使用できるとは限りません。

- COBOL のマルチスレッド構文またはマルチスレッドランタイムシステムの呼び出しを使用して、COBOL プログラムをマルチスレッド化する。

  この方法は、スレッドローカルなデータと、スレッド間で共有するデータをユーザ側で指定できるため、効率面で非常に優れています。ただし、この方法をレガシー COBOL コードで使用するには、Java ランタイム環境とレガシープログラムを仲介する COBOL ドライバプログラムを作成する必要があります。このドライバプログラムによってレガシープログラムへのアクセスを制御します。また、2 つのスレッドが同じコードに同時にアクセスしないように、ドライバプログラムにはセマフォまたは同等の手法を使用する必要があります。

  マルチスレッド環境ではプログラムが複数のスレッドによって共有されるため、COBOL プログラムのキャンセル処理は、十分に注意して実装する必要があります。キャンセル処理の手段として、CANCEL と com.microfocus.cobol.RuntimeSystem.cobcancel() メソッドのどちらを使用する場合も同様です。可能であれば、キャンセル処理はいっさい使用しないでください。

### 9.2.2 COBOL プログラムやライブラリのロード

COBOL プログラムをライブラリファイルにリンクしている場合や、COBOL プログラム内のエントリポイントを公開する場合には、Java プログラムから COBOL プログラムを呼び出す前に、プログラム自体やライブラリファイルをロードする必要があります。

```
{    if (RuntimeSystem.cobload("mycbl") != 0)
        System.out.println("Could not load library
") ;
     else
        System.out.println("Library loaded successfully
") ;
}
```

### 9.2.3 cobcall() 関数の使用方法

必要なライブラリやプログラムをロードした後 (詳細は前項を参照)、Java アプリケーションで cobcall_ 関数を使用して COBOL プログラムを呼び出します。 cobcall_ は **RuntimeSystem.class** に含まれる関数群で、いずれも静的関数として定義されているため、事前に **RuntimeSystem.class** をインスタンス化する必要はありません。 cobcall_ 関数には 2 ～3 つの引数を指定します (3 番目の引数は省略可能)。

引数の指定方法 1

- COBOL プログラムの名前を示す文字列
- COBOL プログラムに渡す各パラメータを格納した Java オブジェクト配列。

  COBOL プログラムの手続き部見出しの USING に指定する順序で格納します。 デフォルトでは、パラメータは参照渡し (BY REFERENCE) されます。

- 各パラメータの USAGE 句を指定する Java 整数配列。 BY REFERENCE、BY VALUE、または BY CONTENT を、**com.microfocus.cobol.RuntimeSystem** で静的に割り当てられている値で指定します。

引数の指定方法 2

- COBOL プログラムの名前を示す文字列
- ParameterList() 内のパラメータリスト

Java と COBOL の間では、『*Java データ型*』の章に説明しているルールに従ってデータ型が変換されます。『*Java プログラムの作成*』の項で前述したように cobcall_ 関数にはいくつかの種類があり、COBOL プログラムやエントリポイントの戻り値のデータ型に応じて、対応する Java データ型の名前を持つ関数を使用します。 たとえば、COBOL プログラムが符号付き整数型 (pic s9(9) comp-5 など) の値を返す場合、この値は Java のデータ型では int 型に相当するため、この COBOL プログラムの呼び出しには cobcall_int 関数を使用します。 デフォルトでは、すべてのパラメータは参照渡しされます。

コピーファイル **javatypes.cpy** には、Java のデータ型に対応する一連の COBOL データ型が定義されています。 Java プログラムとのパラメータのやり取りに使用する COBOL データ項目は、このファイル内で定義されているデータ型を使用して宣言することが推奨されます。 これらのデータ型を使用すれば、他の COBOL プラットフォームとの互換性を維持できます。

cobcall_ の各関数は、『*Class Library Reference*』の『*Java Classes for Run-time Support*』に一覧されています。

### 9.2.4 Java プログラムからの呼び出し例

この項では、Java プログラムから COBOL プログラムを呼び出す 2 つの短いコード例を示します。 最初の例は次の処理を実行します。

- COBOL プログラムへのデータの参照渡し (デフォルト)
- cobcall_int() による整数型の戻り値の取得

最初にシンプルな COBOL のサブルーチン、**legacy.cbl** を示します。

```
 working-storage section.
 copy "javatypes.cpy".
 01 wsResult       jint.

 linkage section.
 01 wsOperand1     jint. *> この型は javatypes.cpy で定義済み
 01 wsOperand2     jint.
 01 wsOperation    pic x.

 procedure division using wsOperand1 wsOperand2 wsOperation.
     evaluate wsOperation
      when "a"
         add wsOperand1 to wsOperand2 giving wsResult
      when "s"
         subtract wsOperand1 from wsOperand2 giving wsResult
     end-evaluate

     exit program returning wsResult.
```

次に、このサブルーチンを呼び出す Java プログラムを示します。

```
import com.microfocus.cobol.* ;
import com.microfocus.cobol.lang.* ;

class SimpleCall
{
   public static void main(String argv[]) throws Exception
   {
       int i = RuntimeSystem.cobcall("legacy",
                          new ParameterList()
                                  .add((int)4)
                                  .add((int)7)
                                  .add((byte)'a'));

       System.out.println(i) ;
   }
}
```

2 つ目の例では、複数の異なる USAGE 句に相当する配列を使用して COBOL プログラムへデータを渡す方法を示します。 使用されている cobcall() 関数は値を戻しませんが、最初のパラメータとしてオブジェクトを渡し、COBOL プログラムから同じオブジェクト型を返すことが

できます。次の SimpleCall2 は、最初のパラメータを参照で、2 番目のパラメータを値で、3 番目のパラメータを内容 (by content) で、それぞれ渡します。

```
import com.microfocus.cobol.* ;
import com.microfocus.cobol.lang.* ;

class SimpleCall2
{
   public static void main(String argv[]) throws Exception
   {
       RuntimeSystem.cobcall(null, "usages",
                        new ParameterList()
                                .add((int)1, RuntimeSystem.BY_REFERENCE)
                                .add((int)2, RuntimeSystem.BY_VALUE)
                                .add((int)3, RuntimeSystem.BY_CONTENT));

   }
}
```

### 9.2.5 Java オブジェクト内のデータメンバの変更

この項の例は前項の例に似ていますが、COBOL プログラムから返される値を使用して、Java オブジェクト内のデータメンバの 1 つの値を変更しています。

```
 thread-local-storage section.
 copy "javatypes.cpy".

 linkage section.
 01 wsOperand1              jint.
 01 wsOperand2              jint.
 01 wsOperation             pic x.
 01 wsResult                jint.

 procedure division using wsOperand1 wsOperand2 wsOperation
                          wsResult.
     evaluate wsOperation
     when "a"
         add wsOperand1 to wsOperand2 giving weResult
     when "s"
         subtract wsOperand1 from wsOperand2 giving weResult
     end-evaluate

     exit program returning wsResult
```

この Java クラス SimpleCall2 は、**legacy2.cbl** が **legacy2.*ext*** というライブラリファイルに組み込まれていることを前提としています。このサブルーチンが別のライブラリファイルに組み込

まれている場合には、legacy2 を呼び出す前に RuntimeSystem.cobload() 関数を使用して、そのライブラリファイルをロードする必要があります。

```
import com.microfocus.cobol.* ;

class SimpleCall2
{
   Integer simpleInteger1;
   Integer simpleInteger2;
   Integer simpleResult;

   public SimpleCall2(int a, int b)
   {
        simpleInteger1 = new Integer(a);
        simpleInteger2 = new Integer(b);
        simpleResult   = new Integer(0);
   }


   public String toString()
   {
        return new String(

                  "simple1Integer1 = "+simpleInteger1+

                  "simple1Integer2 = "+simpleInteger2+

                  "simpleResult    = "+simpleResult);




   }



   public static void main(String argv[]) throws Exception
   {

       SimpleCall2 firstDemo = new SimpleCall2(4,7);
       Object theParams[] = { firstDemo.simpleInteger1,
                              firstDemo.simpleInteger2,
                              new Byte((byte) 'a'),
                              firstDemo.simpleResult };
```

```
       System.out.println("Before call
"+firstDemo) ;
       int i = RuntimeSystem.cobcall_int("legacy2", theParams) ;
       System.out.println("After call
"+firstDemo) ;
   }
}
```

### 9.2.6 インスタンスデータとしての CobolBean の使用

com.microfocus.cobol.CobolBean クラスを使用すれば、COBOL プログラムをスレッドセーフになるように、あるいは Java アプリケーションからパラメータを取り込むように書き直すことなく、CobolBean のインスタンスを COBOL プログラムの記憶域節のインスタンスに関連付けることができます。

たとえば、次の COBOL プログラムは、複数の Java クラスから cobcall() で呼び出されると、同じデータを共有します。

```
     $set intlevel"4" data-context
      working-storage section.
      01 address-rec pic x(30).

      linkage section.
      01 lnk-address-rec pic x(30).

      procedure division.
         goback.
      entry "setAddressBook" using lnk-address-rec.
         move lnk-address-rec to address-rec
         exit program returning 0.
      entry "getAddressBook" using lnk-address-rec.
         move address-rec to lnk-address-rec
         exit program returning 0.
```

com.microfocus.cobol.CobolBean を拡張する Java クラスを作成すれば、cobcall() のインスタンス (data-context) バージョンを利用できます。CobolBean.cobcall() で呼び出す COBOL プログラムは、re-entrant(1/2), serial などのスレッド化指令ではなく、常に data-context 指令を使用してコンパイルする必要があります。data-contetxt 指令を指定すると、COBOL ランタイムが CobolBean.cobcall() から使用されるときに、常に新しい COBOL 作業場所節が割り当てられます。

次の例では、COBOL プログラムの共有データ bean1.getAddress() が bean2.getAddress() と等しければ、通常は問題が発生しますが、Java プログラムが CobolBean の cobcall() を使用しているため、COBOL 作業場所節が CobolBean と関連付けられます。

```
import com.microfocus.cobol.*;
import com.microfocus.cobol.lang.*;

class MyBean extends com.microfocus.cobol.CobolBean
{
   private StringBuffer address = new StringBuffer(30);

   public MyBean() throws Exception
   {
     super();
     super.cobload("addbook");
   }

   public String getAddress() throws Exception
   {
     Pointer addressPointer = new Pointer(this.address.toString(),30);
     super.cobcall("getAddressBook", new
ParameterList().add(addressPointer));
     this.address.setLength(0);
     this.address.append(addressPointer.toString());
     return address.toString();
   }

   public void setAddress(String address) throws Exception
   {
     super.cobcall("setAddressBook", new ParameterList().add(new
Pointer(address,30)));
   }

   public static void main(String[] args) throws Exception
   {
      MyBean bean1 =new MyBean();
      bean1.setAddress("Mr A");

      MyBean bean2 =new MyBean();
      bean2.setAddress("Mr B");

      System.out.println("bean1.getAddress="+bean1.getAddress());
      System.out.println("bean2.getAddress="+bean2.getAddress());
   }
}
```

### 9.2.7 COBOL プログラムのキャンセル

メモリリークを防ぐため、Java オブジェクトがガーベッジコレクタによって破棄される前に、Java からロードした COBOL プログラムをすべてキャンセルする必要があります。Java オブジェクトの Finalize メソッドから、次の呼び出しを使用します。

```
RuntimeSystem.cobcancel("program")
```

ここで *program* は、RuntimeSystem.cobload() の呼び出しでロードされる COBOL プログラムの名前です。次に Java プログラム内の Finalize メソッドの一例を示します。

```
private void Finalize()
{
    try
    {
        RuntimeSystem.cobcancel("demoFinalizers");
        System.out.println("demoFinalizers - finalize'ed");
    }
    catch(Exception e)
    {
        System.out.println(
                "Error during finalize : "+e.getMessage());
    }
}
```

---

**警告:**

- プログラムをキャンセルするのは、そのプログラムが他のプログラムでいっさい使用されていない場合だけに限定すべきです。ただし、サーバ/マルチスレッドプログラムでは、その見極めが困難な場合もあります。
- メソッド名は Finalize であり、finalize ではありません。finalize にするとガーベッジコレクション中にインスタンスが呼び出され、ガーベッジコレクションの処理中にメモリブロックを起す可能性があります。

---

# 9.3 Java プログラムからの文字列操作

Java プログラムから COBOL に文字列を渡すときには、Java の String クラスまたは StringBuffer クラスを使用します。この処理では、次のいずれかの手法を使用できます。

- Pointer クラス ― COBOL のソースコードに変更を加えることなく、Java の文字列を COBOL プログラムに渡すことができます。Pointer クラスは com.microfocus.cobol.lang で定義されています。
- mf-jstring ― Java の文字列を mf-jstring と呼ばれるユーザ定義型に変換できます (mf-jstring は javatypes.cpy で定義されています)。このユーザ定義型に変換すれば、

データ、文字列長、文字列または文字列バッファの容量など、Java 文字列の場合と同様の情報を利用できます。

- CobolNational ― UTF-16 形式の文字列に変換する Java クラス。変換した文字列は COBOL の PIC N(...) usage is national フィールドに渡すことができます。

### 9.3.1 Pointer クラス

Java の文字列を COBOL に渡す際に常に問題になるのが文字列長です。COBOL では、文字列は常に固定長になるからです。COBOL プログラムで定義済みの固定長フィールドに変更を加えるには、そのインスタンスを Pointer クラスの新しいインスタンスでラップする必要があります。Pointer クラスは com.microfocus.cobol.lang で定義されています。

pic x(20) を定義している COBOL プログラムのコード例を次に示します。.

showbytes.cbl:

```
       working-storage section.
       linkage section.
       01 lnk-string     pic x(20).
       procedure division using lnk-string.
          display "lnk-string = " lnk-string.
          move "Hello from COBOL - 1" to lnk-string.
          exit program returning 0.
```

対応する Java のコードは次のようになります。

showbytes.java:

```
import com.microfocus.cobol.RuntimeSystem;
import com.microfocus.cobol.lang.Pointer;
import com.microfocus.cobol.lang.ParameterList;

public class showbytes
{
  public static void main(String[] args) throws Exception
  {
   String myString = new String("Hello to COBOL");
   Pointer ptr2String = new Pointer(myString, 20);

   RuntimeSystem.cobcall("showbytes", new ParameterList().add(ptr2String));

   myString = ptr2String.toString();

   System.out.println("myString = ["+myString+"]");
  }
```

```
}
```

### 9.3.2 mfjstring

COBOL プログラムでは Java から渡された文字列を変更することはできませんが、StringBuffer の内容は変更可能です。文字列操作を容易にするため、**javatypes.cpy** で MF-JSTRING 型が定義されています。定義の内容は次のとおりです。

```
01 mf-jstring is typedef.
    03 len                  jint.
    03 capacity             jint.
    03 ptr2string           pointer.
```

len フィールドは文字列の長さ、capacity フィールドはバッファのサイズをそれぞれ指定します。String の場合、capacity フィールドの値は常に 0 です。ptr2tring フィールドは、実際の文字列の先頭アドレスへのポインタです。StringBuffer の場合は、既存のバッファを変更できるほか、新しいバッファを割り当てて、その先頭アドレスを指すように ptr2string を指定できます。

次の COBOL プログラムは、Java から渡された String と StringBuffer を連結し、結果を StringBuffer に格納して返します。

```
 program-id. StringAdd.
 thread-local-storage section.
 copy "javatypes.cpy".
 01 lsNewPtr                 pointer.
 01 lsSize         jint. *> この型は javatypes.cpy で定義済み
 01 i              jint.
 01 lsStatus       jint.
 01 lsBigBuffer              pic x(1024).
 linkage section.
 01 lnkJString               mf-jstring.
 01 lnkJStringBuffer         mf-jstring.

 01 lnkString                pic x(256).
 01 lnkStringbuffer          pic x(256).
 procedure division using lnkJString lnkJStringBuffer.
     set address of lnkStringBuffer to
                              ptr2string of lnkJStringBuffer
     set address of lnkString to ptr2string of lnkJString
*>   lnkJStringBuffer が Java StringBuffer であることを確認
     if capacity of lnkJStringBuffer > 0
         add len of lnkJString to len of lnkJStringBuffer
                                    giving lsSize
*>       連結バッファがオーバーフローしないことを確認
```

```
        if lsSize < 1024
            move len of lnkJString to i
            move lnkString(1:i) to lsBigBuffer
            add 1 to i
            move lnkStringBuffer to lsBigBuffer(i:lsSize)
            set ptr2string of lnkJStringBuffer
                                to address of lsBigBuffer
            move lsSize to len of lnkJStringBuffer
        end-if
    end-if
```

次の Java クラス StringsToCobol は、文字列 "fred" および "ginger" を StringAdd に渡し、結果を表示します。

```
import com.microfocus.cobol.*;

class StringsToCobol {

  public static void main(String[] args) throws Exception
  {
      StringBuffer sb1 = new StringBuffer("ginger") ;
      Object theParams[] = {"fred" ,
                            sb1} ;

       RuntimeSystem.cobcall_int("stringadd", theParams) ;
       System.out.println(sb1) ;
  }
}
```

### 9.3.3 CobolNational

CobolNational Java クラスを使用すれば、UTF-16 文字列を生成して COBOL の PIC N(...) usage is national フィールドに渡すことができます。

たとえば、次の COBOL プログラム (**hellonat**) は文字列 "Hello From Java" を受け取り、"pic n(40) usage is national" として定義したデータ項目に格納します。

```
 $set unicode(portable)
  working-storage section.
  linkage section.
  01 lnk-natstring   pic n(40) usage is national.
  procedure division using lnk-natstring.
      display "Java said to COBOL [" lnk-natstring "]"
      move "Hello From Java" to lnk-natstring
      exit program returning 0.
```

文字列 "Hello From Java" を COBOL プログラム **hellonat** に渡す Java コードは次のとおりです。

```
CobolNational cobnat = new CobolNational("Hello From Java", 40);
  RuntimeSystem.cobcall("hellonat", new ParameterList().add(cobnat));
  System.out.println(cobnat.toString());
```

## 9.4 カスタムレコードによってグループ項目を渡す方法

com.microfocus.cobol.lang で定義されている CustomRecord インターフェイスを使用すれば、cobcall() や cobrcall() で呼び出したレガシー COBOL プログラムにグループ項目を渡すことができます。

CustomRecord インターフェイスの定義を次に示します。

```
package com.microfocus.cobol.lang;
public interface CustomRecord
{
  public Object[] getParameters();
  public void setParameters(Object[] parms);
}
```

***NetExpress*\Base\Demo\Javademo\Cobol** ディレクトリ内のサンプル、**recorddemo** では、customerDetails が次のように定義されています。

```
  01 customerDetails.
         03 customerName       pic x(30).
         03 customerAddress    pic x(30).
         03 customerRef        pic 9(6).
```

Java による CustomRecord インターフェイスの実装例を次に示します。

```
import com.microfocus.cobol.lang.*;
import java.text.*;

public class RecordData implements com.microfocus.cobol.lang.CustomRecord
{
    private String customerName;
    private StringBuffer customerAddress;
    private int customerRef;
    RecordData(String name, String address, int ref)
    {
      customerName = name;
      customerAddress = new StringBuffer(address);
```

```
      customerRef = ref;
    }
    public String getCustomerName()
    {
      return this.customerName;
    }
    public String getCustomerAddress()
    {
      return this.customerAddress.toString();
    }
    public int getCustomerRef()
    {
      return this.customerRef;
    }
    public Object[] getParameters()
    {
      String strCustomerRef = Integer.toString(this.customerRef);
      while(strCustomerRef.length() <  6)
      {
         strCustomerRef = "0"+strCustomerRef;
      }
      customerAddress.setLength(30);        /* 必ず正しい長さを指定すること
*/
      customerAddress.ensureCapacity(30);
      return new ParameterList()
        .add(new Pointer(this.customerName,30))
        .add(this.customerAddress)
        .add(strCustomerRef.getBytes())
        .getArguments();
    }
    public void setParameters(Object[] parms)
    {
      Pointer ptr = (Pointer)parms[0];
      this.customerName = ptr.toString();
      this.customerAddress = (StringBuffer)parms[1];
      byte[] byteCustomerRef = (byte[])parms[2];
      this.customerRef = Integer.parseInt(new String(byteCustomerRef));
    }
    public String toString()
    {
      return "Customer Name    : "+this.customerName+"
"+
             "Customer Address : "+this.customerAddress+"
"+
             "Customer Ref     : "+this.customerRef;
    }
```

```
}
```

Java プログラム内では、次のようなコードで COBOL プログラムを呼び出します。

```
cobcall(“RecordDemo”, new ParameterList().add( new RecordData(myname,
myaddress, myref)));
```

『*Java データ型*』の章の『ParameterList() によるパラメータの追加』を参照してください。

# 9.5 COBOL での JNI の使用

JNI (Java Native Interface) は、Java のオブジェクトとクラスを、Java 以外の言語で作成したプログラムで利用するためのインターフェイスです。**com.microfocus.cobol.RuntimeSystem** クラス内の cobcall() 関数の一種を使用して、呼び出し対象の COBOL プログラムに JNI ポインタを渡します。JNI ポインタとは、Java 以外のコードから Java ランタイムシステムへのアクセスを可能にする Java 関数群のテーブルを指すポインタです。

COBOL で JNI を容易に利用できるようにするため、**javatypes.cpy** でデータ型 JNINativeInterface が定義されています。JNINativeInterface は、JNI Java 関数群を指す一連の手続きポインタで構成されるグループデータ項目です。COBOL での JNI の用途としては、COBOL プログラムからの Java 例外のスローなどがあります。

---

**注記**：COBOL プログラムから Java にアクセスする手段としては、ドメインサポートも使用できます (『*COBOL からの Java の呼び出し*』を参照)。

---

## 9.5.1 例外スローの一例

JNI の関数を使用して例外をスローする COBOL プログラムの一例を次に示します。

```
 identification division.
 program-id.  "except".
 special-names.
************************************************
* JNI 呼び出しに使用する呼び出し規約をここで
* 指定します。 Win32 システムの場合は 74 を
* 指定します (stdcall の呼び出し規約に対応
* しています。)
************************************************
$if UNIX defined
     call-convention 0 is javaapi.
$else
```

```
     call-convention 74 is javaapi.
$end
 local-storage section.
 copy "javatypes.cpy".
 01 JavaException              pointer.
 01 ExceptionClass             pointer.

 linkage section.
 01 JEnv                       pointer.
 01 jobject                    pointer.
 01 lnk-JNINativeInterface     JNINativeInterface.

*  最初のパラメータ (JEnv) は、JNI 関数テーブルへのポインタ
*  です。 JNI 関数呼び出しでは常に、このポインタを最初の
*  パラメータとして参照渡しする必要があります。
*
 procedure division
      using by reference JEnv.

*    JEnv で渡されたポインタを JNINativeInterface
*    構造体の先頭アドレスにマッピングして、JNI 関数
*    の呼び出しを可能にします。
     set address of lnk-JNINativeInterface to JEnv

* 例外クラスへの参照を取得
      call javaapi FindClass
             using by reference JEnv
                   by reference
                   z"java/lang/IllegalArgumentException"
             returning ExceptionClass
      end-call

* 例外クラスが見つからない場合は、そのまま終了
     if ExceptionClass = NULL
             exit program
     end-if

* 新しい例外をスローする
     call javaapi ThrowNew
             using by reference JEnv
                   by value ExceptionClass
                   by reference z"Thrown from COBOL code"
     end-call
     exit program
     .
```

次に呼び出し側の Java プログラムを示します。

```
import com.microfocus.cobol.*;

class testexcept
{
    public static void main(String argv[]) throws Exception
    {
        try
        {   /* JNIEnv で渡すため、最後の引数は
               true に設定する */
            RuntimeSystem.cobcall(null,"throwex",null,null,true);
        }
        catch(Exception e)
        {
            System.out.println(
                      "PASS - exception caught ("+ e + ")");
        }
    }
}
```

## 9.6 サンプルプログラム

Net Express には、Java からの 手続き型 COBOL 呼び出しのさまざまな側面を示すサンプルプログラムが付属しています。 サンプルプログラムは ***NetExpress*\Base\Demo\Javademo\Cobol** 配下の複数のディレクトリに格納されています。これらのディレクトリ内の各ファイルは、いずれもサンプル関連ファイルであり、**readme.txt** ファイルに詳細情報が記載されています。

サンプルプログラムを格納している各ディレクトリと、格納されているサンプルの機能の概要を次に示します。

| ディレクトリ名 | 概要 |
|---|---|
| **arrays** | COBOL で Java 配列を読み取る。 |
| | COBOL で Java 配列を更新する。 |
| **pi** | pi を計算する。 |
| | 数値を文字列として Java に返す。 |
| **primtypes** | Java の原始データ型を COBOL に渡す。 |
| | Java の原始データ型を COBOL で更新する。 |
| **record** | シンプルな COBOL Java オブジェクトを作成する。 |

DataType インターフェイスを実装する Java オブジェクト から COBOL プログラムにデータを渡し、構造体に格納する。

パラメータを参照で (BY REFERENCE) 渡す。

---



---

# 第 10 章 : Object COBOL からの Java の呼び出し

この章では、Object COBOL の Java ドメインを使用して Java メソッドを呼び出す COBOL プログラムの作成方法を説明します。

## 概要

Micro Focus の Java サポートを使用すれば、Object COBOL のプログラムやクラスから Java オブジェクトにメッセージを送信することができます。

Java サポートは Object COBOL の Java ドメインを通じて実現されます。Java ドメインによって、COBOL プログラム内で Java クラスを宣言し、そのクラスにメッセージを送信することが可能になります。 図 10-1 に示すように、Java ドメインでは Java オブジェクトごとに、対応する COBOL プロキシオブジェクトを作成します。 宣言したクラス自体が、Java クラスの静的メソッドのプロキシになります。

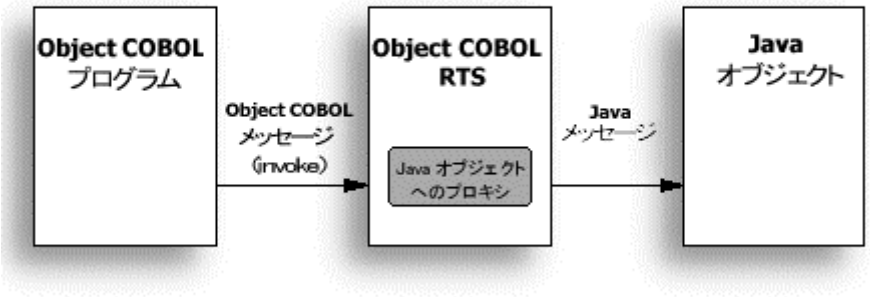



**図 10-1 Java プロキシ**

この用途に使用する COBOL プログラムにはクラス制御節が必要で、Java メソッドを呼び出すときには、常に INVOKE 動詞を使用する必要があります。ただし、必ずしも Object COBOL のクラスとして実装する必要はありません。

COBOL プログラムからメッセージ付きで送信されたパラメータは、COBOL ランタイムシステムによって COBOL から Java のデータ型に変換されます。 また、Java メソッドがパラメータを返す場合には、そのパラメータが Java データ型から COBOL データ型に変換されます。

この章で説明する手法を使用すれば、JDBC、Sockets、Swing など、あらゆる Java API を呼び出すことができます。

ここで説明する情報を活用するには、Java 言語に関する基本レベル以上の知識が必要になります。Java 言語の基本的な知識を得るには、Net Express Links のリンク先にある Sun の Java Web サイトが役立ちます。

COBOL プログラムを Java から呼び出すこともできます。その場合には、Object COBOL の Java ドメインは使用しません。方法については、『*Java からの手続き型 COBOL の呼び出し*』で説明しています。また、Java クラスから COBOL にメッセージを送信することも可能です。方法については、『*Java からの Object COBOL の呼び出し*』を参照してください。

# Java クラスの宣言

COBOL プログラムで使用する各 Java クラスは、COBOL のクラス制御節で個別に宣言する必要があります。宣言には、Java クラスが属するパッケージの名前を、直前に **$java$** を付けて省略せずに記述します。このプリフィックスによって COBOL ランタイムシステムに、クラスを Java ドメインからロードするように指示します。

宣言はクラス制御節に、次の形式で記述します。

```
repository.
     class COBOL-classname as "$java$class-name"
    .
```

各部の意味は次のとおりです。

| | |
|---|---|
| *COBOL-classname* | COBOL プログラム内でクラスを識別する名前 |
| java | Java ドメイン名 |
| *class-name* | Java ドメインでクラスを識別する名前。通常は、当該ドメインのオブジェクトモデルに登録されているクラス名を使用する。 |

次に例を示します。

```
repository.
     class jRectangle as "$java$java.awt.Rectangle"
     .
```

この例は、java.awt パッケージ内の Rectangle クラスの COBOL プロキシオブジェクトとして jRectangle を宣言しています。java.awt パッケージは、Java の classpath に登録されている必要があります。この条件を満たさないとランタイムエラーになり、プログラムは正しく実行できません。

# Java オブジェクトのインスタンス化

Java の各クラスには、オブジェクトをインスタンス化するためのコンストラクタメソッドが 1 つ以上含まれています。Java では、コンストラクタメソッドはクラスと同じ名前を持ちます。これらのコンストラクタメソッドは、COBOL プロキシオブジェクトで new というメソッド名にマッピングされ、COBOL 側から呼び出し可能になります。

同じ Java クラスでもコンストラクタが異なれば、インスタンスを初期化するために必要なパラメータの数や組み合わせが異なります。たとえば、Java の Rectangle クラスは、次に示す 2 通りの方法を含む、いくつかの異なる方法でインスタンス化できます。

```
Rectangle r1 = new Rectangle ()
             // 起点座標 (0,0)、幅 0、高さ 0 の矩形
Rectangle r2 = new Rectangle(4, 5, 10, 20)
            // 起点座標 (4,5)、幅 10、高さ 20 の矩形
```

これに相当する COBOL のコードは次のようになります。

```
working-storage section.
01 r1                  object reference.
01 r2                  object reference.
...

procedure division.
...
    invoke jRectangle "new" returning r1
         *> 起点座標 (0,0)、幅 0、高さ 0 の矩形
    invoke jRectangle "new" using 4, 5, 10, 20
                        returning r2
         *> 起点座標 (4,5)、幅 10、高さ 20 の矩形
```

COBOL ランタイムシステムは、パラメータの数と型を基に、Java クラス内の適切なコンストラクタを呼び出します。Java は型チェックが厳密であるため、指定するパラメータの型は、Java クラスの定義に必ず一致させてください。COBOL データ型と Java データ型のマッピングについては、『*Java データ型*』の章で説明しています。コピーファイル **javatypes.cpy** には、Java のデータ型に直接的に対応する一連のデータ型が定義されており、Java と COBOL 間でデータをやり取りするときには、これらのデータ型を COBOL 側で使用することが推奨されます。

# Java メソッドの呼び出し

Java オブジェクト内のメソッドは、いずれもメソッドと同じ名前のメッセージを送信することによって呼び出します。また、COBOL プログラムのクラス制御節で宣言したクラス名にメッセージを送信して、Java クラスの静的メソッドを呼び出すこともできます。Java ではメソッド名の大文字、小文字が区別されるため、COBOL プログラム内に記述するメッセージ名は、Java の対応するメソッドと大文字、小文字の使い分けまで一致させる必要があります。

Java にはメソッドのオーバーロード機能があり、渡されるパラメータの数と型に応じて、同じ名前のメソッドの実装を変えることができます。この機能に必要な COBOL 側の処理は内部的に実行されるため、常に適切な Java メソッドが呼び出されます。

たとえば、Rectangle クラスには互いに異なるパラメータを受け取る 3 種類の add() メソッドがあります。次の Java コードは、Rectangle の add() メソッドを呼び出す 3 通りの方法を示しています。

```
Rectangle r1 = new Rectangle(0,0,0,0) ;
Point pt = new Point(6,6) ;
Rectangle r2 = new Rectangle(3,4,9,9) ;
r1.add(4,5) ; // r1 を r1 自体と ポイント 4,5 を含む
              // 最小の矩形に変更する
r1.add(pt) ; // r1 を r1 自体と ポイント pt を含む
              // 最小の矩形に変更する
r1.add(r2) ; // r1 と r2 を結合した矩形を r1 にする
```

これに相当する COBOL のコードは次のようになります。

```
repository.
    class jRectangle as"$java$java.awt.Rectangle"
    class jPoint as "$java$java.awt.Point"
    .
working-storage section.
01 r1                object reference.
01 r2                object reference.
01 pt                object reference.

procedure division.
    invoke jRectangle "new" returning r1
    invoke jPoint "new" using 4 5 returning pt
    invoke jRectangle "new" using 3 4 9 9 returning r2
    invoke r1 "add" using 4 5
    invoke r1 "add" using pt
    invoke r1 "add" using r2
```

r2 と pt はどちらもオブジェクト参照型のデータ項目ですが、COBOL ランタイムシステムは表現される Java オブジェクトの型を判断して適切な Java メソッドを呼び出します。

# Java 変数の利用

Object COBOL プロキシで set*name* メソッドと get*name* メソッドを呼び出せば、Java クラスのパブリックメンバと静的変数にアクセスできます。Java では変数名の大文字、小文字が区別されるため、COBOL プログラム内に記述する変数名部分 (*name*) は、Java コード内での宣言に大文字、小文字の使い分けまで一致させる必要があります。

たとえば、次の Java クラスに含まれるパブリック変数 (classVal、instVal) にアクセスする場合を考えてみましょう。

```
public class x {
    static int classVal;
    int instVal;
};
```

次の COBOL コード例は、静的変数 classVal を設定し、続いてメンバ instVal の値を取得します。

```
$set ooctrl(-f+p)
repository.
     class x as "$Java$x"
     .

 working-storage section.
 copy "javatypes.cpy".
 01 anX                  object reference.
 01 anInt                jint.
 procedure division.
     invoke x "setclassVal" using by value 4
     invoke x "new" returning anX
     invoke anX "getinstVal" returning anInt
```

# Java 例外の処理

Java からスローされた例外は、javexpt クラスに発生した Object COBOL 例外として COBOL に返されます。デフォルトでは、COBOL ランタイムシステムは例外を受信すると、その旨を警告するメッセージを表示して終了しますが、 COBOL プログラムにイベントハンドラを追加して、例外をトラップすることも可能です。

次の手順は、『*Class Library Reference*』に記載されている例外処理の情報に、すでに目を通していることを前提としています。

Java 例外をトラップする手順は次のとおりです。

1. クラス制御節で、Exceptionmanager、JavaExceptionManager、および Callback (または EntryCallback) の各クラスを宣言します。

```
2.   repository.
3.       ...
4.       class JavaExceptionManager as "javaexpt"
5.       class ExceptionManager as "exptnmgr"
6.       class Callback as "callback"
7.       class EntryCallback as "entrycll"
```

```
...
```

8. 例外ハンドラ (クラスのメソッドまたはエントリポイント) を作成した後、対応する Callback または EntryCallback のコードを記述します。Callback を作成する場合は、次のように記述します。

```
9.    invoke Callback "new" using anObject z"methodName"
                          returning aHandler
```

EntryCallback を作成する場合は、次のように記述します。

```
invoke EntryCallback "new" using  z"entryPointname"
                          returning aHandler
```

10. コールバックを JavaExceptionManger クラスに登録します。次に例を示します。

```
11.  invoke ExceptionManager "register"
               using JavaExceptionManager aHandler
```

以上の手順を実施すると、Object COBOL の Java ドメインを経由して呼び出している Java クラスからスローされた Java 例外が、常に作成した例外ハンドラに送信されるようになります。

Java プログラムからスローされた例外をキャッチする COBOL プログラムの一例を次に示します。

```
$set ooctrl (+p-f)
 program-id. ExceptionCatcher.

 class-control.
     SimpleClass is class "$JAVA$SimpleClass"
     EntryCallback is class "entrycll"
     JavaExceptionManager is class "javaexpt"
     ExceptionManager is class "exptnmgr"
     .

 working-storage section.
 01 theInstance                  object reference.
 01 wsCallback                   object reference.
 local-storage section.
 01 filler pic x.    *> ローカルエントリポイントをコールバックに
                    *> 使用するためのダミーの記憶域
 linkage section.
 01 lnkException                 object reference.

 procedure division.
*>---例外ハンドラのセットアップ
      invoke EntryCallback "new" using z"JException"
                             returning wsCallback
```

```
      invoke ExceptionManager "register"
                                    using javaexceptionmanager
                                          wsCallback
*>---クラスのインスタンス化
      invoke SimpleClass "new" returning theInstance

      display "instantiated"
      invoke theInstance "TestException"
      display "excepted"
      stop run.


 entry "Jexception" using lnkException.
      invoke lnkException "display"
      .
```

このプログラムでは局所記憶域節によって再帰呼び出しが可能になっており、COBOL ランタイムシステムが EntryCallback を再帰的に呼び出します。局所記憶域節がないとランタイムエラーになり、このプログラムは正しく実行できません。

呼び出される SimpleClass の Java コードは次のとおりです。

```
import java.lang.* ;

public class SimpleClass {

  public SimpleClass() {
  }

  public void TestException() throws Exception
  {
     Exception e = new Exception ("Test error" );
     throw e;
  }
}
```

# ネイティブ Java オブジェクトへのアクセス

Object COBOL の Java ドメインを使用する場合、Java オブジェクトに直接アクセスすることはありません。『概要』で説明したように、常にプロキシを経由してアクセスします。一方、javasup クラスの getJavaObject メソッドを使用すると、実際の Java オブジェクトへのポインタを取得することができます。このメソッドを JNI (Java Native Interface) と組み合わせて使用すれば、Object COBOL の Java ドメインでは利用できない Java の機能も利用可能になります。

JNI ポインタを取得するには、javasup クラスの getEnv メソッドを呼び出します。 JNI ポインタは関数テーブルへのポインタです。次の例に示すように、コピーファイル **javatypes.cpy** で定義されている JNINativeInterface 構造体を使用すれば、JNI 関数テーブルに容易にアクセスできます。

```
 working-storage section.
 01 JEnv                          pointer.
 linkage section.
 01 lnk-JNINativeInterface        JNINativeInterface.

*>
 procedure division.
     invoke javasup "getEnv" returning jEnv
*>    JEnv で渡されたポインタを JNINativeInterface
*>    構造体の先頭アドレスにマッピングして、JNI 関数の
*>    呼び出しを可能にします。
     set address of lnk-JNINativeInterface to JEnv
*>
```

以上の処理を行うと、JNINativeInterface の型定義で指定されている名前を使用して、JNI 関数を呼び出すことが可能になります。 実際の呼び出し例は、『*Java からの手続き型 COBOL の呼び出し*』の章の『*例外スローの一例*』に記載されています。 JNI の詳細については、Net Express Links のリンク先にある Sun の Java Web サイトをご覧ください。

javasup クラスの詳細については、『*Class Library Reference*』を参照してください。

# Java オブジェクトの終了

Java ランタイムには、不要になったオブジェクトを自動的に破棄するガーベッジコレクタがあります。ガーベッジコレクタは、他のいっさいのオブジェクトから参照されていないオブジェクトを削除します。 COBOL プログラムで Java オブジェクトへのプロキシを作成すると、その Java オブジェクトへの参照が COBOL ランタイムシステムで保持され、Java ガーベッジコレクタによるオブジェクトの破棄が防止されます。

Java オブジェクトが不要になったときには、そのオブジェクトへの参照を COBOL プログラム内で明示的に解放して、ガーベッジコレクタで破棄させる必要があります。この処理を行わないと、アプリケーションでメモリリークが発生します。具体的には、次の呼び出しを実行します。

```
invoke javaobject "finalize" returning javaobject
```

returning パラメータは必須です。このパラメータを省略すると、COBOL ランタイムシステムは COBOL プロキシを破棄するかわりに、呼び出した Java オブジェクトにメッセージを渡します。このメッセージを受信した Java オブジェクトは、次の関数を呼び出します。

```
public void finalize()
```

この関数は、すべての Java クラスに継承または実装されており、Java ガーベッジコレクタによって、オブジェクトを破棄する前に呼び出されます。

通常、Java クラスの finalize() メソッドには戻り値はありません。ただし、使用する Java クラスが値を戻す finalize() メソッドを実装しているようであれば、Object COBOL プロキシを破棄する手段として、finalize のかわりに次の delete メソッドを使用してください。

```
invoke javaobject "delete" returning javaobject
```

---



---

# 第 11 章 : Java からの Object COBOL の呼び出し

この章では、Java プログラムから COBOL オブジェクトにアクセスする方法を説明します。この方法は、Java プログラムから手続き型 COBOL にアクセスする方法とはやや異なります (手続き型 COBOL へのアクセス方法については、『*Java からの手続き型 COBOL の呼び出し*』の章を参照してください)。

## 概要

オブジェクト指向 COBOL (Object COBOL) で作成したクラスは、あたかも Java クラスであるかのように Java プログラムから呼び出すことが可能です。そのためには、Object COBOL クラスのラッパークラスを Java で作成し、このラッパークラスを通じて Object COBOL クラス内の各メソッドに対応する関数を提供します。Net Express では、クラスウィザードとメソッドウィザードによって COBOL コードと同時に Java コードを生成できるため、この作業も容易に行うことができます。

Java ラッパークラス内の関数は、対応するメソッドに渡す各パラメータを Java の配列に格納し、Java クラス com.microfocus.cobol.RuntimeSystem のいずれか 1 つのメンバ関数の呼び出しを通じて Object COBOL 内のメソッドを呼び出し、処理結果を返します。この処理の流れを次に示します。

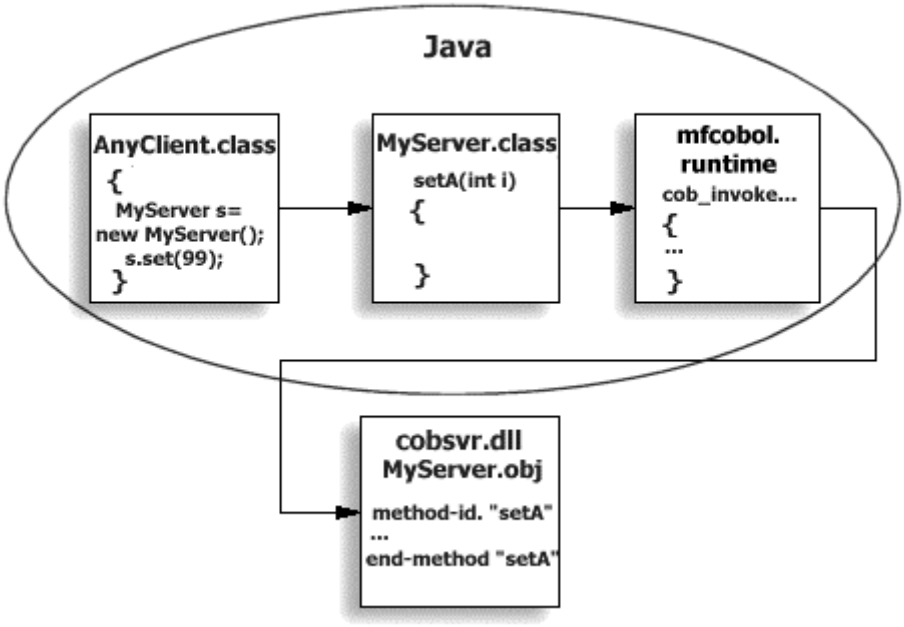



**図 11-1 Java の関数呼び出しから COBOL のメソッドが呼び出されるまでの処理の流れ**

以下の内容では、Object COBOL クラスと対応する Java ラッパーの作成方法を説明し、Net Express のウィザードで生成されるコードの種類を示します。

ここで説明する情報を活用するには、Java 言語に関する基本レベル以上の知識が必要になります。Java 言語の基本的な知識を得るには、Net Express Links のリンク先にある Sun の Java Web サイトが役立ちます。

# 作業環境のセットアップ

実際の作業に着手する前に、COBOL と Java のランタイムシステムが連携できるように、作業環境をセットアップする必要があります。具体的な手順については、『*Java と COBOL の連携*』の章にある『*Java と COBOL の環境のセットアップ*』を参照してください。

# Object COBOL での Java クラスの作成

Java プログラムから呼び出し可能なクラスを Object COBOL で作成するには、次の手順に従います。

1. プログラム内で次の指令を設定します。

   ```
   $set ooctrl(+p-f)
   ```

2. Object COBOL クラスは javabase を継承する必要があります。
3. Object COBOL クラスの Java ラッパークラスを作成します。ラッパークラスには、Object COBOL クラスのファイル名と同じ名前を付けます。
4. Java ラッパークラスには、Object COBOL クラス内に作成した各メソッドに対応するメソッドが必要です。
5. 作成したクラスを Windows プラットフォームにディプロイする場合は、.dll ファイルにパッケージ化する必要があります。

   .int ファイルや .gnt ファイルにコンパイルすることも可能ですが、これらのファイル形式では、複数のクラスを 1 つのファイルにパッケージ化することができません。

Net Express のクラスウィザードを使用して Object COBOL 内で Java クラスを作成すると、適切な継承と指令を持つ Object COBOL クラスが生成され、同時に Java ラッパークラスも生成されます。詳細は『*クラスウィザード*』を参照してください。

## COBOL サポートのインポート

Java ラッパークラスが Object COBOL クラスと通信するために必要なサポート機能は、**com.microfocus.cobol** パッケージに含まれています。したがって、ラッパーには次の文を記述する必要があります。

```
import com.microfocus.cobol.* ;
```

さらに、**mfcobol.jar** ファイルを含むディレクトリを、Java の classpath に必ず登録します。このファイルが見つからないと、Java プログラムはコンパイルできず、実行することもできません。『*Java と COBOL の連携*』の章の『*Java と COBOL の環境のセットアップ*』を参照してください。

## ラッパークラス

Java ラッパークラスでは、**microfocus.cobol.RuntimeObject** または **microfocus.cobol.RuntimeSystem** を拡張する必要があります。この拡張は、Java ラッパークラスのインスタンスで表される Object COBOL のインスタンスの存在期間に次のような影響を与えます。

- ラッパーが **microfocus.cobol.RuntimeObject** を継承する場合、ラッパーオブジェクトが Java のガーベッジコレクタで破棄されると、対応する Object COBOL インスタンスも終了する。
- ラッパーが **microfocus.cobol.RuntimeSystem** を継承する場合、メソッドの実行が終了すると Object COBOL インスタンスも終了し、インスタンスデータが失われる。

ラッパークラスは、ラッピング対象の COBOL クラスのライブラリとファイル名で初期化される必要があります。この初期化を実行するには、Java ラッパークラスに次のコードを含めます。

```
static
{
   cobloadclass("libname", "filename", "fullJavaClassName") ;
}
```

説明

| | |
|---|---|
| *libname* | Object COBOL クラスを含む **.dll** ファイルの名前 Object COBOL クラスを **.int** 形式や **.gnt** 形式のコードで実行している場合など、**.dll** ファイルとしてパッケージ化していない場合には、このパラメータの指定は省略できます。 |
| *filename* | Object COBOL クラスのファイル名。 |
| *FullJavaClassName* | Object COBOL クラスに対応する Java クラスの名前。 |

**microfocus.cobol.RuntimeSystem** では 3 種類の cobloadclass() メソッドが定義されています。これらのメソッドは、ライブラリ、COBOL ファイル、および Java ラッパークラスを識別する方法が、互いに少し異なっています。

## メソッドの追加と削除

Java ラッパークラスでは、COBOL クラスに追加した各メソッドに対応する関数を定義する必要があります。 Net Express のメソッドウィザードで COBOL クラスにメソッドを追加すると、対応するメソッドが Java ラッパーに自動的に追加されます。 ただし、いったん追加したメソッド

を COBOL クラスから削除した場合は、対応するメソッドを Java ラッパークラスから手作業で削除する必要があります。

詳細については、『*クラスウィザード*』を参照してください。

---

**注記**: メソッドウィザードで String 型または StringBuffer 型のパラメータを受け取るメソッドを追加すると、これらのパラメータ値の格納先として 32 バイト長のデータ項目 (PIC X(32)) が生成されます。これらのパラメータで渡した文字列が 32 バイトより短い場合、データ項目内では文字列の末尾にヌルバイト (x"00") が付加され、ヌルバイトから 32 番目のバイトまでの内容は未定義になります。したがって、渡された文字列を処理する前に、次のようなコードを追加して文字列の長さを特定する必要があります。

```
MOVE 0 TO received-length
INSPECT passed-string TALLYING received-length FOR CHARACTERS BEFORE
INITIAL x"00"
MOVE passed-string(1:received-length) TO actual-string
```

---

以下の内容では、Java 関数をメソッドウィザードを使用せず、手作業でラッパークラス内に記述する方法を説明します。

Java 関数には、次の処理が必要です。

1. COBOL メソッドの呼び出しで発生する可能性がある例外を宣言する。

   デフォルトでは、COBOL で例外が発生したときにスローされる Exception と、COBOL ランタイムシステムによってスローされる COBOLException を宣言します。Java クラスを Enterprise Java Bean としてディプロイするか、Java RMI (リモートメソッド起動) を使用する場合は、RemoteException も宣言する必要があります。

2. Java から COBOL メソッドに渡すパラメータを格納する Java 配列を作成する。

   Java のパラメータは Java 配列で COBOL ランタイムシステムに渡されます。

3. **RuntimeSystem.java** クラスの cobinvoke_ メソッドまたは cobinvokestatic_ メソッドを呼び出す。

   Java 関数の戻り値の型に応じて、対応する種類の cobinvoke_ または cobinvokestatic_ を使用します (たとえば、戻り値が int 型であれば cobinvoke_int を使用します)。また、Object COBOL のインスタンスを呼び出すには cobinvoke_ メソッド、Object COBOL のクラスを呼び出すには cobinvokestatic_ メソッドを使用します。

具体的な cobinvoke_ 関数の種類は、『*Java Run-time Class Library Reference*』に一覧されています。

COBOL メソッドの戻り値の型にマッピングする Java データ型を決定し、適切な cobinvoke_ 関数を使用する必要があります。詳細については、『*Java データ型*』の章を参照してください。

Object COBOL クラスのメソッドは、Java ラッパーの静的関数にマッピングされます。

COBOL インスタンスのメソッドと、それに対応する Java ラッパーの関数のサンプルコードを次に示します。まず、COBOL のメソッドのサンプルコードを示します。

```
 method-id. "myMethod".
 local-storage section.
*>---ユーザコード： 局所記憶域として必要なデータ項目を記述します。
 linkage Section.
 01 myParameter            pic x(4) comp-5.
 01 myReturnValue          pic x(4) comp-5.

 procedure division using by reference myParameter
                    returning myReturnValue.
*>---ユーザコード： メソッドを実装するコードを記述します。
     exit method.
 end method "myMethod".
```

次に Java ラッパーのメソッドのサンプルコードを示します。

```
public  int myMethod (Integer myParameter) throws Exception,
                    CobolException, RemoteException
 {
    // パラメータは配列で COBOL に渡されます。
    Object[] params = {myParameter};
    return ((int) cobinvoke_int ("myMethod", params)); }
```

通常、Java ラッパークラスのメソッドには、COBOL クラスの対応するメソッドと同じ名前を付けますが、これは必須ではありません。ただし、同じ名前を使用すれば、Java ラッパー内でのメソッドのオーバーロードが可能になります。

メソッドのオーバーロードでは、パラメータの型や数が異なるメソッドを複数、同じ名前で定義できます。メソッドのオーバーロードは Java でサポートされている機能であり、COBOL ではサポートされていませんが、複数のオーバーロード関数を Java ラッパーに追加して、それぞれの関数で COBOL クラス内の異なる名前のメソッドを呼び出すことができます。

たとえば、Java ラッパークラスでは次のようなオーバーロード関数を定義できます。

```
public int add (int a, int b)
{...}
public int add (int a, int b, int c)
{...}
```

これらの関数は、それぞれ COBOL クラス内の異なる名前のメソッド (add2 と add3 など) にマッピングすることが可能です。

Net Express のメソッドウィザードで、Java クラスとして生成した COBOL クラスにメソッドを追加する場合は、Java ラッパーと COBOL クラスの対応するメソッドに、互いに異なる名前を付けることができます。

## COBOL からの例外のスロー

COBOL コードから Java 例外をスローすることが可能です。 あらゆる Java クラスを例外オブジェクトとして使用できます。 例外をスローするメソッドは javasup クラスで定義されています。詳細については、『*Class Library Reference*』を参照してください。.

この用途には、次のいずれかのメソッドを使用できます。

```
invoke javasup "throwException" using aCOBOLProxy
```

説明

*aCOBOLProxy* Java の Throwable オブジェクトに対応する COBOL プロキシ

```
invoke javasup "throwNewException" using javaclassname
description
```

説明

*javaclassname* スローする Java 例外の名前

*description* 例外の説明

## BY REFERENCE パラメータの使用方法

メソッドに渡すパラメータを BY REFERENCE として宣言すると、デフォルトでは Object COBOL から呼び出し側 Java クラスに制御が戻ったときに、それらのパラメータに加えられたすべての変更が対応する Java オブジェクトに反映されます。

この Java オブジェクトの更新は、次の静的変数を使用することによって回避できます。

```
static boolean mfcobol.runtimeProperties.updateByRefParams;
```

この変数のデフォルト値は true です。false に設定すると、Java オブジェクトは更新されません。

---

**注記**: 旧バージョンの Net Express では、BY REFERENCE で定義されたパラメータが更新されても、デフォルトでは呼び出し側 Java オブジェクトには反映されません。旧バージョンの動作を前提としている Java コードがある場合には、mfcobol.runtimeProperties.updateByRefParams を false に設定するように、コードを変更する必要があります。

---

Net Express には、BY REFERENCE パラメータの処理方法の違いを示すサンプルプログラムが付属しています。 ***NetExpress*\Base\Demo\Javademo\OOCOBOL\Record** フォルダには、このサンプルプログラムに関連する全ファイルが格納されており、**readme.txt** ファイルにプログラムの詳細情報が記載されています。

### Java から渡されたデータの COBOL 構造体への格納

Net Express では、COBOL プログラムや Object COBOL のメソッドの連絡節で構造体を定義し、Java クラスから渡されるデータの格納先としてしようすることができます。COBOL プログラムや Object COBOL のメソッドから、この構造体を使って Java のデータにアクセスする方法の詳細については、『*Java データ型*』の章の『*構造体の使用方法*』を参照してください。

## Java プログラムを扱う際の留意点

この項では、Net Express で Java プログラムを扱うときに、特に留意すべき情報を示します。

- Net Express で Java のビルドに使用されるデフォルトの指令は、`-d %TARGETDIR -classpath "%$CLASSPATH";%TARGETDIR %FILENAME` です。この指令に含まれる `"%$CLASSPATH";%TARGETDIR` によって、現在指定されているターゲットディレクトリが環境変数 CLASSPATH の値の末尾に追加されます (CLASSPATH に登録済みのディレクトリはそのまま維持されます)。
- 複数のクラスを 1 つの **.java** プログラム内で指定した場合には、その **.java** プログラムとベース名が同じ **.class** ファイルだけが自動的にプロジェクトに追加されます。その他の **.class** ファイルは手作業で追加する必要があります。
- デバッグモードでは Java ビルド指令として -g (デバッグ情報の生成) を指定し、リリースモードでは代わりに -O (コードの最適化) を指定することが推奨されます。これらの指令を指定するには、[**プロジェクト**] メニューの [**ビルド設定**] をクリックし、.java ファイルごとに「コンパイル」タブを選択してリストに指令を追加します。

## JAR ファイルの作成

JAR (Java ARchive) 形式のファイルは、Java アプリケーションを配布する手段として広く使用されています。 Net Express で作成した Java クラスは、Zip ファイルや実行可能ファイルへのパッケージ化と同じ要領で JAR ファイルにして配布できます。

Net Express では、JAR ファイルの作成時に使用する設定を変更できます。 Net Express が JAR ファイルを作成するときには、デフォルトでは Jar コマンドで次のオプションが使用されます。

- c ― 新規アーカイブを作成する。
- v ― 詳細情報を出力する。
- 0 ― 圧縮を使用しない (格納専用)。
- f ― アーカイブ名を指定する。

デフォルト以外のオプションを指定する必要があるときには、他のビルド設定と同じ方法でオプションを変更できます。

# サンプルプログラム

Net Express には、Java からの Object COBOL 呼び出しのさまざまな側面を示すサンプルプログラムが付属しています。 サンプルプログラムは *NetExpress*\Base\Demo\Javademo\OOCOBOL 配下の複数のディレクトリに格納されています。 これらのディレクトリ内の各ファイルは、いずれもサンプル関連ファイルであり、**readme.txt** ファイルに詳細情報が記載されています。

サンプルプログラムを格納している各ディレクトリと、格納されているサンプルの機能の概要を次に示します。

| ディレクトリ名 | 概要 |
|---|---|
| **array** | Object COBOL で Java 配列を受信し、配列内のデータにアクセスする。 |
| | Object COBOL で Java 配列を作成して Java クラスに渡す。 |
| **record** | シンプルな COBOL Java オブジェクトを作成する。 |
| | DataType インターフェイスを実装する Java オブジェクト から Object COBOL のメソッドにデータを渡し、構造体に格納する。 |
| | パラメータを参照で (BY REFERENCE) 渡す。 |
| **simple** | シンプルな COBOL Java オブジェクトを作成する。 |
| | Object COBOL のクラスとインスタンスのメソッドを Java から呼び出す。 |

| | |
|---|---|
| **sort** | **シンプルな COBOL Java オブジェクトを作成する。** |
| | **Java で Object COBOL メソッドを呼び出す。** |
| | **Java で Object COBOL メソッドを呼び出す。** |
| | **Java オブジェクトをパラメータとして処理し、結果を返す。** |

# 第 12 章 : Java データ型

この章では、COBOL と Java のデータ型の対応関係について説明します。

## 12.1 概要

Java 言語では、COBOL で使用されるデータ型とは異なる、Java 用のデータ型が定義されています。図 12-1 に示すように、COBOL から Java (または Java から COBOL) が呼び出されると、COBOL ランタイムシステムによって COBOL と Java 間でデータ型が自動的に変換されます。

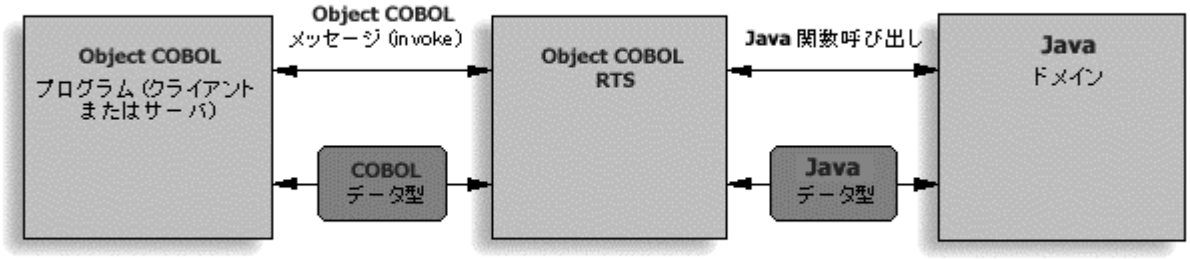



**図 12-1 データ型の変換**

数値データ型は、COBOL プログラムが手続き型、オブジェクト指向のどちらで作成されている場合も同じように変換されますが、オブジェクトや文字列の場合はこれらの作成手法によって変換結果が異なります。

Object COBOL では Java オブジェクトは COBOL オブジェクト参照として扱われますが、手続き型 COBOL ではポインタとして扱われます。

## 12.2 Java データ型の変換規則

COBOL から Java にデータを送信するときには、そのデータが適切な Java データ型に変換されます。同様に、Java プログラムが COBOL にデータを返信するときにも、Java データ型から COBOL データ型への変換が行われます。次の表は、Java と COBOL の間でデータがやり取りされるときに実行される型変換を示しています。一部のデータ型は、手続き型 COBOL (『*Java からの手続き型 COBOL の呼び出し*』の章を参照) と Object COBOL の Java ドメイン (『*Object COBOL からの Java の呼び出し*』および『*Java からの Object*

*COBOL の呼び出し*』の章を参照) では扱いが異なる場合があります。表中の「COBOL」欄は、.cobcall(…) で手続き型 COBOL に適用される規則、「Object COBOL」欄は .cobinvoke() で Object COBOL Java ドメインに適用される規則を示しています。

| Java データ型 | COBOL | Object COBOL | 説明 |
|---|---|---|---|
| byte | pic s99 comp-5 | pic s99 comp-5 pic s99 comp | 1 バイトの符号付き整数 |
| short | pic s9(4) comp-5 | pic s9(4) comp-5 pic s9(4) comp | 2 バイトの符号付き整数 |
| int | pic s9(9) comp-5 | pic s9(9) comp-5 pic s9(9) comp | 4 バイトの符号付き整数 |
| long | pic s9(18) comp-5 | pic s9(18) comp-5 pic s9(18) comp [1] | 8 バイトの符号付き整数 |
| boolean | pic 99 comp-5 | pic 99 comp-5 | 0 は false、0 以外の値は true |
| char | (Unicode) pic 9(4) comp-5 | pic s9(4) comp | Java の文字はすべてダブルバイトの Unicode 文字で表される。 |
| float | comp-1 [1] | comp-1 [1] | 浮動小数点数値 |
| double | comp-2 [1] | comp-2 [1] | 倍精度浮動小数点数値 |
| String | mf-jstring [2] pointer [3] | pic x(n) | 容量は常に 0。COBOL プログラムに渡される文字列は読み取り専用で、変更できないものと考える必要がある。 |

| | | | |
|---|---|---|---|
| StringBuffer | mf-jstring [2]<br><br>pointer [3] | pic x(n) | 容量はバッファの合計サイズで、サイズはバッファに格納されている文字列の長さ |
| objects | ポインタ | オブジェクト参照 | あらゆる Java オブジェクト。手続き型 COBOL では、返されたポインタを JNI 呼び出しに使用できる (『*Java からの手続き型 COBOL の呼び出し*』の章の『*COBOL での JNI の使用*』を参照)。 |
| object[] | ポインタ | jarray クラスのインスタンスへのオブジェクト参照 | Java オブジェクトの配列。手続き型 COBOL では、返されたポインタを JNI 呼び出しに使用できる (『*Java からの手続き型 COBOL の呼び出し*』の章の『*COBOL での JNI の使用*』を参照)。jarray は、Java 配列の内容にアクセスするための Object COBOL クラス (『*Jarray クラスの使用方法*』の項を参照)。 |
| DataType | 構造体 | 構造体 | 複合データ構造体 |
| Comp-3 | comp-3 | comp-3 | COMP(UTATIONAL)-3 または PACKED-DECIMAL |
| Pointer(StringBuffer)<br>Pointer(String, int capacity) | pic x(99) | pic x(99) | String や StringBuffer をラップすることによって、正しいサイズが割り当てられた状態で COBOL プログラムに渡し、COBOL 側で Java オブジェクトを安全に編集できるようにする。<br><br>StringBuffer の容量は、COBOL 側で編集可能な最大長として使用される。 |
| java.sql.Date(long) | 0x jyear pic | same as | JDBC で使用される標準 |

| | | | |
|---|---|---|---|
| CobolTime(time) | 9(4)<br>0x filler pic x<br>0x jmonth pic 99<br>0x filler pic x<br>0x jday pic 99 | for cobcall | SQL Date 型 |
| java.sql.time(long)<br>CobolDate(date) | 0x jhour pic 99<br>0x filler pic x<br>0x jminute pic 99<br>0x filler pic x<br>0x jseconds pic 99 | same as for cobcall | JDBC で使用される標準 SQL Time 型 |
| CobolBigDecimal(java.math.BigDecimal) | pic s9(19)v9(19) comp-3 | same as for cobcall | COBOL 側での BigDecimal 値の更新を可能にする CobolBigDecimal ラッパークラス。CobolBigDecimal() のコンストラクタは初期値をとる。更新後の値を取得するには、getValue() メソッドを使用する。 |
| CobolBigInteger(java.math.BigInteger) | pic S9(38) comp-3 | same as for cobcall | COBOL 側での BigInteger 値の更新を可能にする CobolBigInteger ラッパークラス。CobolBigInteger() のコンストラクタは初期値をとる。更新後の値を取得するには、getValue() メソッドを使用する。 |
| CustomRecord | 構造体 | 構造体 | Java でオブジェクト配列として表される複合データ構造体 |
| CobolNational | pic x(*n*)<br>usage is national. | same as for cobcall | UTF-16 文字列を生成して COBOL に渡すための Java クラス。 |

「COBOL」欄と「Object COBOL」欄のどちらにも含まれない COBOL データ型への変換はサポートされていません。ここに挙げられていない型の COBOL データ項目を使用するには、

pic x(n) usageis national.same as forcobcall 型のデータ項目を中間データ項目として使用し、このデータ項目との間で必要に応じてデータを転記します。

脚注

1. 参照渡しのみ。浮動小数点小数点型の場合、この制約が適用されるのは UNIX プラットフォームに限定されます。long 型の場合は、32 ビットプラットフォームでは参照渡しのみ、64 ビットプラットフォームでは参照渡しと値渡しが可能です。
2. .cobcall(...) で mf-jstring を使用します。受信側の COBOL プログラムで Java の String または StringBuffer にアクセスするには、ptr2string ポインタを使用する必要があります。mf-jstring を使用すれば、COBOL プログラムから Java オブジェクトのサイズと容量にアクセスできます。
3. .cobcall(...) でポインタを使用します。

# 12.3 ユーザ定義データ型の型定義

コピーファイル **javatypes.cpy** には、Java データ型に対応する一連の COBOL データ型が定義されており、Java で使用するデータ項目を COBOL プログラム内で宣言するときに利用できます。これらの型定義を次の表に一覧します。いずれも先頭文字が j になっています。

このコピーファイルは、**Net Express\source** ディレクトリ内にあります。これらの型を使用するすべてのプログラムは、intlevel"4" 指令でコンパイルする必要があります。

| 型定義 | COBOL 型 |
|---|---|
| jbyte | pic s99 comp-5 |
| jshort | pic s9(4) comp-5 |
| jint | pic s9(9) comp-5 |
| jlong | pic s9(18) comp-5 |
| jboolean<br>(JNI-TRUE：true<br>JNI-FALSE：false) | pic 99 comp-5 |
| jchar | pic 9(4) comp-5 |
| jfloat | comp-1 |
| jdouble | comp-2 |
| jobject | ポインタ |
| jbigdecimal | pic s9(19)v9(19) comp-3 [1] |
| jbiginteger | pic s9(38) comp-3 [1] |
| mf-sql-date<br>(javatypes.cpy での定義) | 0x jyear pic 9(4)<br>0x filler pic x<br>0x jmonth pic 99<br>0x filler pic x |

| | |
|---|---|
| | 0x jday pic 99 |
| mf-sql-time<br>(javatypes.cpy での定義) | 0x jhour pic 99<br>0x filler pic x<br>0x jminute pic 99<br>0x filler pic x<br>0x jseconds pic 99 |

脚注

1. コンパイラ指令として intlevel 4 を使用します。

# 12.4 Jarray クラスの使用方法

jarray クラスは、Object COBOL で Java 配列を操作するためのラッパーです。詳細は『*Java ドメインクラスライブラリリファレンス*』で説明されています。

次の COBOL プログラムは Java オブジェクトから配列を受け取り、次元を検出して、配列の内容を表示します。

```
$set ooctrl(+p-f)
 Program-id. ReadArray.
 class-control.
     arraydemo is class "$Java$arraydemo"
     .

 thread-local-storage section.
 01 aJavaObj             object reference.

 01 theTotal         pic 9(9).
 01 CDims            pic x(4) comp-5.
 01 Dims.
   03 Dims-entry     pic x(4) comp-5 occurs 256.
 01 Bounds.
   03 Bounds-entry   pic x(4) comp-5 occurs 256.
 01 ind0             pic x(4) comp-5.
 01 ind1             pic x(4) comp-5.
 01 arrayElement     pic x(4) comp-5.
 01 wsTable          object reference.
 01 wsResult         pic x(4) comp-5.

 procedure division.
   invoke arraydemo "new" returning aJavaObj
   invoke aJavaObj "getArray" returning wsTable
```

```
*> 配列内の要素数を検出
     invoke wsTable "getDimensions" returning CDims
     display "The array has " CDims " dimension(s)"

*> 各次元の要素数を取得
     display "dimensions are " with no advancing
     invoke wsTable "getBounds" using Bounds
     perform varying ind0 from 1 by 1 until ind0 > CDims
          display Bounds-entry(ind0) with no advancing
          if ind0 < CDims
             display " by " with no advancing
          end-if
     end-perform
     display " "

*> 配列内の各要素を表示
     perform varying ind0 from 0 by 1
             until ind0 = Bounds-entry(1)
         move ind0 to Dims-entry(1)
         perform varying ind1 from 0 by 1
                   until ind1 = Bounds-entry(2)
             move ind1 to Dims-entry(2)
             invoke wsTable "getElement" using
                                  by value CDims
                                  by reference Dims
                                  by reference arrayElement
             display "Element " ind0 ","
                   ind1 " is " arrayElement
*> 配列の内容を変更
             add 50 to arrayElement
             invoke wsTable "putElement" using
                                   by value CDims
                                   by reference Dims
                                   by reference arrayElement
         end-perform
     end-perform
```

この ReadArray プログラムで使用される arraydemo クラスの実装を次に示します。このプログラムは 2 次元配列を生成します。

```
import com.microfocus.cobol.*;
import java.io.*;

public class arraydemo extends RuntimeSystem
{
    int myArray[][];
```

```
    public arraydemo()
    {
        myArray = new int[5][2];
        int i,j;
        for (i = 0; i < 5; i++)
        {
            for (j = 0; j < 2; j++)
                myArray[i][j] = i * 100 + j;
        }
    }

    public int[][] getArray()
    {
        return myArray;
    }
}
```

## 12.5 ParameterList() によるパラメータの追加

com.microfocus.cobol.lang で定義されている ParameterList() クラスは、Java と COBOL の間でパラメータをやり取りする手段として役立ちます。

ParameterList() は通常、次の形式で使用します。

```
   RuntimeSystem.cobcall("myProgram",
    new ParameterList()
        .add(myOneInt,RuntimeSystem.BY_VALUE)
        .add(mySecondParameter,RuntimeSystem.BY_REFERENCE)
                .add(99)            // 数値 99 の参照渡し
                );
```

説明

| | |
|---|---|
| *myProgram* | 呼び出すプログラムの名前 |
| *myOneInt* | 値渡しする整数型パラメータ |
| *mySecondParameter* | 参照渡しするパラメータ |
| *99* | 参照渡しする数値 |

## 12.6 構造体の使用方法

COBOL プログラムや Object COBOL のメソッドでは、連絡節で構造体を使用し、Java クラスから渡されたデータをその中に格納することが可能です。ただし、対応する Java 側のパラメ

ータが、**com.microfocus.cobol.lang.Datatype** または **com.microfocus.cobol.lang.CustomRecord** を実装したオブジェクトであることが前提になります。次にこれらのインターフェイスの定義を示します。

```
package com.microfocus.cobol.lang;
public interface DataType
{
   void synchronizeData();
   byte[] getBytes();
}
```

および

```
package com.microfocus.cobol.lang;
public interface CustomRecord
{
   public Object[] getParameters();
   public void setParameters(Object[] parms);
}
```

**com.microfocus.cobol.lang.Datatype/com.microfocus.cobol.lang.CustomRecord** を使用すれば、複雑な構造のデータを COBOL のプログラムやメソッドに渡すことができます。多数の基本データ型でインターフェイスを作り直す必要はありません。

**com.microfocus.cobol.lang.Datatype** インターフェイスを実装したクラスの一例が、**mfcobol.jar** で定義されている **com.microfocus.cobol.lang.Pointer** です。
com.microfocus.cobol.lang.Pointer のコンストラクタを次に示します。

```
/* String initString から Pointer オブジェクトを作成 */
public Pointer(String initString);
/* String initString から Pointer オブジェクトを作成 */
public Pointer(String initString);
/* capacity で指定されるバイト数の Pointer オブジェクトを作成 (initString の
文字数が capacity より小さい場合、余剰部分は空白文字でパディングされる) */
public Pointer(String initString, int capacity);
```

### 12.6.1 com.microfocus.cobol.lang.CustomRecordinterface によるカスタムレコードの作成

.cobcall ではカスタムレコードを使用することが可能です。カスタムレコードは、レガシー COBOL プログラムに容易にグループ項目を渡すことができるように設計されています。

CustomRecord インターフェイスの定義を次に示します。

```
package com.microfocus.cobol.lang;
public interface CustomRecord
```

```
{
  public Object[] getParameters();
  public void setParameters(Object[] parms);
}
```

RecordDemo2 の customerDetails は次のように定義されています。

```
  01 customerDetails.
         03 customerName       pic x(30).
         03 customerAddress    pic x(30).
         03 customerRef        pic 9(6).
```

Java による CustomRecord インターフェイスの実装例を次に示します。

```
import com.microfocus.cobol.lang.*;
import java.text.*;
public class RecordData implements com.microfocus.cobol.lang.CustomRecord
{
    private String customerName;
    private StringBuffer customerAddress;
    private int customerRef;
    RecordData(String name, String address, int ref)
    {
        customerName = name;
        customerAddress = new StringBuffer(address);
        customerRef = ref;
    }
    public String getCustomerName()
    {
      return this.customerName;
    }
    public String getCustomerAddress()
    {
      return this.customerAddress.toString();
    }
    public int getCustomerRef()
    {
      return this.customerRef;
    }
    public Object[] getParameters()
    {
     String strCustomerRef = Integer.toString(this.customerRef);
     while(strCustomerRef.length() <  6)
     {
        strCustomerRef = "0"+strCustomerRef;
     }
```

```
     customerAddress.setLength(30);         /* 必ず正しい長さを指定すること */
     customerAddress.ensureCapacity(30);
     return new ParameterList()
        .add(new Pointer(this.customerName,30))
        .add(this.customerAddress)
        .add(strCustomerRef.getBytes())
        .getArguments();
    }

    public void setParameters(Object[] parms)
    {
      Pointer ptr = (Pointer)parms[0];
      this.customerName = ptr.toString();
      this.customerAddress = (StringBuffer)parms[1];
      byte[] byteCustomerRef = (byte[])parms[2];
      this.customerRef = Integer.parseInt(new String(byteCustomerRef));
    }
    public String toString()
    {
        return "Customer Name    : "+this.customerName+"
"+
               "Customer Address : "+this.customerAddress+"
"+
               "Customer Ref     : "+this.customerRef;
    }
}
```

Net Express には、Java から COBOL の構造体にデータを渡す方法を示すサンプルプログラムが付属しています。サンプルプログラムの関連ファイルは、**Net Express\Base\Demo\Javademo\Oocobol\RecordDemo** フォルダと **Net Express\Base\Demo\Javademo\Cobol\RecordDemo** フォルダに含まれています。これらのフォルダには、サンプルプログラムの詳細を記載した **readme.txt** ファイルもあります。

---

**注記**：Server Express の 2.0.11 以降のリリースでは、この機能が一部の UNIX プラットフォームでも利用できるようになりました。特定の UNIX プラットフォームでの利用の可否については、Micro Focus のアンサーラインにお尋ねください。

---

---

# 第 13 章 : COBOL での COM の利用

この章では、COM の概要と Net Express での活用方法を説明します。また、インターフェイスマッパーを使用して COM コンポーネントを作成する方法についても説明しています。

## COM (Component Object Model)

COM とは、Microsoft Windows の機能の 1 つであり、アプリケーションの機能を再利用可能な形で他のアプリケーションに公開するための技術仕様です。COM として実装された市販パッケージは、カスタムソフトウェアと組み合わせて、新しいアプリケーションの作成に利用できます。Microsoft Office の各アプリケーションと、Microsoft BackOffice のほとんどのアプリケーションは、COM オートメーションを通して機能を公開しており、これらのアプリケーションの一部を再利用して、共通の機能を実行することができます。たとえば、作成しているアプリケーションに、本格的なレポートを生成して印刷する機能が必要なときには、Microsoft Word の機能を利用できます。

Net Express では、Micro Focus COM サポートを通じて、COBOL のプログラムやクラスから COM オブジェクトにメッセージを送信する (COBOL を COM クライアントとして使用する) ことができます。また、COM オブジェクトを作成し、COM オートメーションを通して COBOL クラスを操作することもできます。

COM コンポーネントの説明では、サーバ、クラス、クライアント、オブジェクトなどの用語が独自の意味で使用されるため、混乱の原因になります。 このマニュアルでは、これらの用語を次の意味で使用しています。

- COM クラス ― 既存の COBOL プログラムや COBOL アプリケーションを基に作成する COBOL クラス
- COM サーバ (COM オートメーションサーバ) ― COM クラスを使用して作成されたアプリケーション
- COM クライアント ― COM サーバアプリケーションにアクセスするあらゆるプログラム
- COM オブジェクト ― クライアントで受信される COM サーバアプリケーションのインスタンス
- COM コンポーネント ― COM オブジェクトと同義

COM については、次の各章でも説明しています。

- COM オブジェクトの作成

  COM オブジェクトの作成手順を説明しています。 COM オブジェクトが公開する機能は、他のアプリケーションから制御できます。Micro Focus COM オートメーションサーバはデュアルインターフェイスをサポートしており、Net Express の インターフェイスマッパー、クラスウィザード、およびメソッドウィザードでは、そのサポートに必要なタイプライブラリを自動的に作成して更新できます。

- **COM オートメーションサーバの実行**

  COM オートメーションサーバをクライアントと通信できるように設定するための必要条件について説明しています。

- **COM クライアントの作成**

  オートメーションサーバとして動作している COM オブジェクトにアクセスする COM クライアントを、COBOL で作成する方法について説明しています。

- **COM コンポーネントの配布**

  DCOM (Distributed Common Object Model) の概要と、DCOM による COM コンポーネントの配布方法について説明しています。

- **COM データ型**

  COBOL データ型を COM データ型にマッピングする方法を説明しています。

- **Microsoft Component Services**

  Net Express を使用して、COBOL で COM+ コンポーネントを作成する方法を説明しています。

- **.NET との連携**

  .NET 環境で運用できる COM オブジェクトを Net Express で作成する方法について説明しています。

---

**注記**: COM クライアントと COM オブジェクトは、それぞれ OLE オートメーションクライアント、OLE オートメーションサーバという名前で呼ばれていた技術です。

---

# インターフェイスマッパー

インターフェイスマッパーを使用すると、プロセス内 COM オブジェクト (.dll) を作成することができます。そのために必要な情報のほとんどは、『***分散コンピューティング***』の『*Interface Mapping Toolkit*』の章で説明しています。インターフェイスマッパーで COM オブジェクトを作成するときには、次の点に留意してください。

- **インターフェイスマッパーで作成できるのはプロセス内 COM オブジェクトであり、プロセス外 COM オブジェクトではない。**

- 他のプログラムに従属し、連絡節を含んでいるプログラムから作業を開始する。
- COM は COBOL の構造に対応しておらず、内部のデータフィールドのみが処理される。

  「COBOL エントリポイント」ウィンドウから右ペイン内のいずれかのウィンドウに構造をドラッグすると、構造ではなく、内部のデータフィールドのリストが表示されます。

- 「再利用マッピング」ウィンドウ内で複数のフィールドをグループにまとめることによって、新しいデータ型を作成できる。
- 入力フィールド、出力フィールド、入出力フィールドのいずれも定義できる。

運用環境で使用できる COM オブジェクトをインターフェイスマッパーで生成するには、「サービスインターフェイス」ウィンドウで対象オブジェクトを右クリックし、ポップアップメニューの [**ディプロイ**] をクリックします。 Net Express によって ***serviceinterfacename*.dll** ファイルが、 プロジェクトファイル (**.app** ファイル) が存在しているフォルダのサブフォルダ (***projectname*\repos |*serviceinterfacename*.deploy**) 内に生成されます。詳細については、ヘルプトピックの『インプロセス COM オブジェクトの登録方法』を参照してください。

上記の操作を行うと、COM オブジェクトがディプロイ (運用環境に配置) されます。ただし、この COM オブジェクトにアクセスするには、COBOL アプリケーションを実行している必要があります。 詳細については、ヘルプトピックの『概要 - アプリケーションの実行』を参照してください。

# プロジェクトに着手する前に

COM の技術は急速に進歩しており、プロジェクトで使用するサードパーティ製ソフトウェアで、必要な機能がサポートされていない可能性もあります。プロジェクトに着手する前に、使用するサードパーティ製ソフトウェアをすべて最新リリースにアップデートし、必要な機能がすべてサポートされている状態にしてください。

具体的なチェックポイントは次のとおりです。

- 使用しているリリースのオペレーティングシステムで、必要なレベルの機能がサポートされていること。

  COM のレベル (オートメーションサーバ、DCOM、COM+) とオペレーティングシステムによるサポートついては、EnterpriseLink Links のリンク先にある Microsoft の Web サイトを参照してください。

- 使用するサードパーティ製のソフトウェアや言語ツールが、いずれも必要なレベルの機能をサポートしていること。

Net Express には、COBOL で COM クライアントを作成する方法を示すサンプルとして、**excel.app** と **word.app** が付属しています。前者のプロジェクトは **base\demo\comdemos\excel** フォルダ内、後者は **base\demo\comdemos\word** フォルダ

内にあります。これらのサンプルの実行方法については、それぞれのフォルダにある **readme.txt** を参照してください。

# 追加情報

COM は幅広い領域に関わる技術であり、このような短い章で詳細まで語ることは不可能です。COM に関する知識を高めれば、Net Express のプロジェクトにも役に立つでしょう。次にいくつかの参考文献を紹介します。

- 『***Inside COM***』(Dale Rogerson 著)
- 『***Essential COM***』(Don Box 著)
- 『***Understanding COM+***』(David S. Platt 著)

また、*Microsoft's MSDN Library* に収録されている COM 関連のドキュメントも、情報源として活用できます。

# 第 14 章 : COM オブジェクトの作成

この章では、Object COBOL のクラスから COM オブジェクトを作成する方法について説明します。

---

**注記**: この章では、すべてのメソッドとクラスを Net Express のクラスウィザードとメソッドウィザードで作成します。インターフェイスマッパーでプロセス内 COM オブジェクトを作成する方法については、『COM と COBOL の連携』の章の『*インターフェイスマッパー*』を参照してください。

---

## Object COBOL で COM オブジェクトを作成するための条件

Object COBOL のクラスが次の各条件を満たす場合、そのクラスから COM オブジェクトを作成できます。

- OLEBase または OLEBase-s を継承する。

  これらのクラスによって、オートメーションを介して COM と連携するために必要なコードが提供されます。OLEBase-s はシングルユーザサーバを指定します。つまり、OLEBase-s に接続するクライアントごとに新しいサーバが生成されます。OLEBase はマルチユーザサーバ用です。ほとんどのサーバはマルチユーザサーバです。プロセス内サーバは、シングルユーザ、マルチユーザのどちらにも該当しません。

  また、COBOL クラスは olebase クラスを継承する必要があります。COM は継承をサポートしていないため、これらのクラスからサブクラスを作成することはできません。

- OOCTRL(+P) コンパイラ指令付きでコンパイルしている。

  OOCTRL(+P) 指令を指定してコンパイルすると、オートメーションを介して COM と連携するために必要な型チェック追加情報が生成されます。

  **注記**: OOCTRL(+P) 指令を指定すると、メソッドの呼び出しや宣言に含めることができるパラメータの数が 31 個 までに制限されます (そのほかに、必要に応じて RETURNING パラメータを使用できます)。

- クラスの CLSID、ProgID、およびプログラムパスを Windows に登録している。

  クライアントは、Windows のレジストリを通じてクラスの場所を特定します。登録形式は COM オブジェクトの実行方法によって異なります (次項を参照してください)。

Net Express のクラスウィザードを使用して、クラスのひな型 (スケルトン)、レジストリエントリ、およびタイプライブラリを作成します (必要に応じてトリガプログラムも作成できます)。クラスウィザードを開くには、[**ファイル**] メニューの [**新規作成**] をクリックし、「新規作成」ダイアログで [**クラス**] を選択します。

# COM オブジェクトの処理

COM サーバアプリケーションは、次のいずれかの方法で実行できます。

- プロセス外

  オブジェクトが、それをクライアントしているプロセスとは異なるプロセス空間で動作します。プロセス外オブジェクトは、単独の **.exe** プログラムとしてビルドされます。プロセス外のオブジェクトがクラッシュしても、クライアントはエラーから回復できます。

- プロセス内

  オブジェクトがクライアントと同じプロセス空間にロードされます。プロセス内オブジェクトは **.dll** プログラムとしてビルドされます。プロセス内 COM オブジェクトには、プロセス外のオブジェクトに比べ、より高速にアクセスできます。保護違反などの原因でオブジェクトがクラッシュしたときには、ほとんどの場合、クライアントも同時にクラッシュします。

- リモート

  クライアントとオブジェクトが別のマシンで実行されます。メッセージはネットワーク経由でやり取りされます。リモート COM オブジェクトは、プロセス内、プロセス外のどちらのサーバとしてもビルドできます。ローカルで動作する COM オブジェクトを開発し、ディプロイする準備が整った段階でリモートオブジェクトに変換することができます。詳細については、『*COM コンポーネントの分散配置*』の章を参照してください。

  プロセス内サーバとしてビルドしたリモート COM オブジェクトは、クライアントのプロセス空間では実行されず、プロキシクライアントによってリモートマシンにロードされます。その結果、プロセス内サーバは一貫してロードされた状態になり、以降のクライアントごとに再ロードする必要がないため、パフォーマンスが向上します。

  リモート COM オブジェクトは、COM の拡張機能である DCOM (Distributed Component Object Model) でサポートされます。すべてのバージョンの Windows で DCOM がサポートされているわけではありません。詳細については、『*COBOL での COM の利用*』の章の『*プロジェクトに着手する前に*』を参照してください。

実行方法は COM オブジェクトのソースコードではなく、ビルドと登録の方法によって左右されます。次の 2 つの図は、プロセス内オブジェクト、プロセス外オブジェクト、およびリモートオブジェクトを示しています。

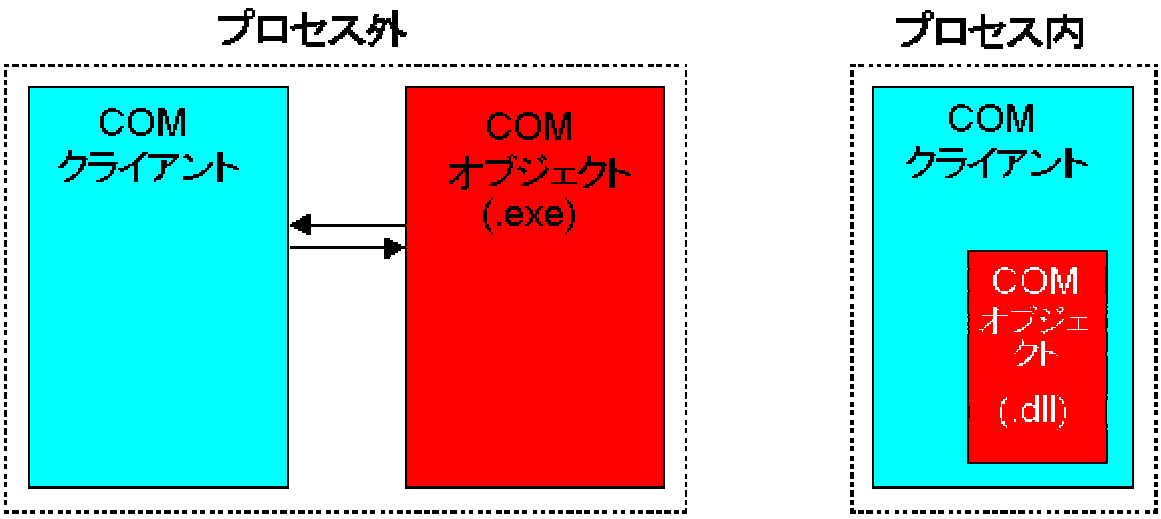



**図 14-1 プロセス外とプロセス内の COM オブジェクト**

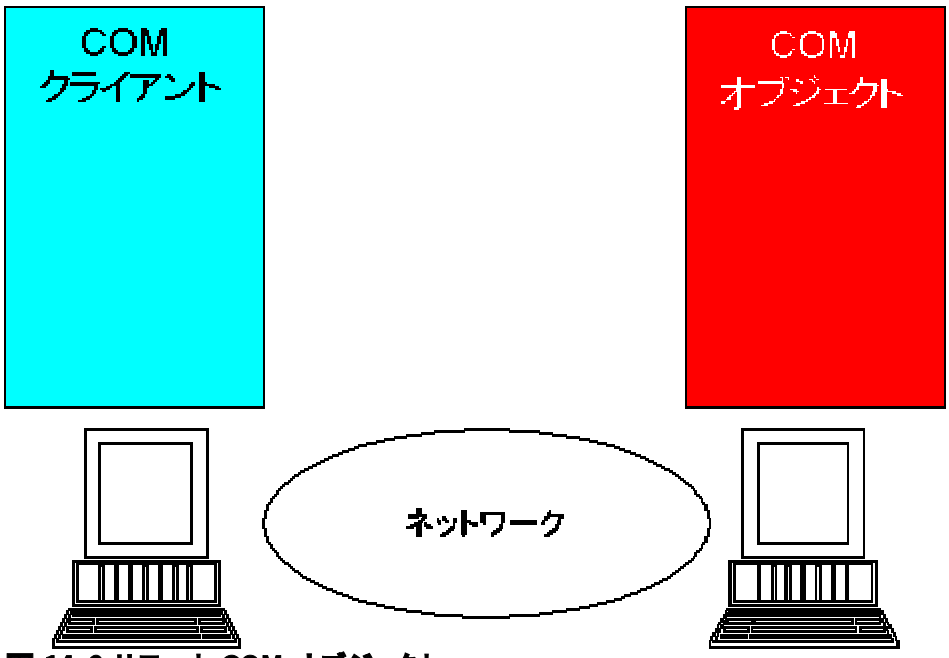



**図 14-2 リモート COM オブジェクト**

# COM オブジェクトに使用するクラスの作成

Object COBOL の COM オートメーションサーバは、Object COBOL クラスとして実装します。このクラスのインスタンスが COM オブジェクトになります。ここでは、クラスの作成手順を次の 3 段階に分けて説明します。

1. サーバクラスのスケルトンの生成
2. COM オブジェクトへのメソッドの追加
3. COM オブジェクトへのプロパティの追加

### サーバクラスのスケルトンの作成

Net Express のクラスウィザードでは、COM オートメーションクラス、トリガプログラム、およびレジストリエントリのひな型 (スケルトン) を生成できます。ウィザードを実行する前に、次の決定を行う必要があります。

- クラスの名前とファイル名
- サーバの実行方法 (プロセス内/プロセス外)
- サーバの実行場所 (ローカル/リモート)

  ローカルサーバとして開発したサーバを、完成した時点でリモートサーバに変更することもできます。ローカルサーバとリモートサーバの違いは、Windows への登録方法だけです。

- タイプライブラリの生成の有無

  タイプライブラリには、クラスのクライアントの実行に役立つ情報が含まれています。

  デュアルインターフェイス対応のライブラリを作成する場合 (デフォルト) には、タイプライブラリを生成すればクライアントを早い段階でタイプライブラリに結合できるため、パフォーマンスが大幅に向上します。タイプライブラリには、常に最新のクラスを反映する必要があります。後述する『*タイプライブラリの更新*』を参照してください。また、デュアルインターフェイスクライアントはタイプライブラリを通じて、早い段階でクラスに結合されるため、タイプライブラリを変更した場合は、これらのクライアントを再リンクする必要があります。タイプライブラリとクライアントに最新の変更が反映されていないと、プログラムが実行時にパラメータの不一致によってクラッシュする可能性があります。

続いてクラスウィザードを開きます。このウィザードを開く方法については、ヘルプトピックの『COM コンポーネントクラスの作成方法』を参照してください。

クラスウィザードの最後のページで [**完了**] ボタンをクリックすると、次の各ファイルが生成または更新され、プロジェクトに追加されます。

- Object COBOL クラス (*filename*.cbl)
- クラスの Windows レジストリエントリを含む *filename*.reg ファイル (オプション)
- クラスのトリガを含む *filename*Trigger.cbl プログラム (オプション)

タイプライブラリの生成を要求した場合には、次の各ファイルも生成されます。

- クラスの詳細情報を含む *filename_if*.idl ファイル
- タイプライブラリの詳細情報を含む *filename*.idl ファイル
- クラスのタイプライブラリを含む *filename*.rc ファイル (作成するサーバを、すでに .rc ファイルを持っている既存の .dll ファイルや .exe ファイルの一部にする場合を除く)

クラスをプロセス内サーバとしてビルドする場合は **.dll** ファイル、プロセス外サーバとしてビルドする場合は **.exe** ファイルが、それぞれクラスウィザードによって確実に生成されます。

---

**注記**: クラスウィザードでプロセス内サーバを作成する場合は、**dllserver.obj** を **.dll** ファイルにリンクする指令も、Net Express プロジェクトに追加されます。 **dllserver.obj** は Net Express のリンクライブラリに含まれており、このリンクライブラリはデフォルトでは **base\lib** にインストールされています。 ヘルプトピックの『アウトオブプロセス COM サーバからインプロセス COM サーバへの変換方法』を参照してください。

---

### COM オブジェクトへのメソッドの追加

COM オートメーションサーバクラスに追加するメソッドは、名前が set または get で始まるメソッドを除き、すべて COM オブジェクトのメソッドとして公開されます (『*タイプライブラリの更新*』を参照してください)。メソッドとの間でやり取りする各パラメータは、当該メソッドの連絡節で宣言します。COBOL のデータ型と COM オートメーションのデータ型は、『*COM データ型*』の章で説明するように、Object COBOL ランタイムシステムによって相互変換されます。

インスタンスメソッドを追加する方法については、ヘルプトピックの『クラスへのメソッドの追加』を参照してください。

メソッドを追加または削除した場合には、そのオブジェクトのタイプライブラリも更新する必要があります。この処理は、メソッドウィザードを使用すると自動的に実行されます。詳細については、『*タイプライブラリの更新*』を参照してください。

すべてのインスタンスメソッドは、次のコメント間に配置されます。

```
*> OCWIZARD - start standard instance methods
...
*> OCWIZARD - end standard instance methods
```

---

**注記**: COM はクラスオブジェクトとクラスメソッドをサポートしていないため、COM オートメーションサーバに記述したクラスメソッドはすべて無視されます。ただし、新しいインスタンスの作成をクラスオブジェクトに通知する必要がある場合は、new という名前でクラスメソッドを作成し、その中に通知コードを記述します。COM 以外の Object COBOL クラスの場合とは異なり、この new メソッドに新しいインスタンスを作成するコードを含める必要はありません。

---

### COM オブジェクトへのプロパティの追加

COM オブジェクトのオートメーションプロパティは、set メソッドと get メソッドを記述して定義します。クライアントとの間で実行される COM プロパティの設定と取得の処理は、すべて set メソッドと get メソッドにマッピングされます。 プロパティの設定や取得は、次の 2 通りの方法で有効化できます。

- メソッドウィザードでメソッドを追加する際に、「メソッドの種類」ページの [**プロパティの設定**] (または [**プロパティの表示**]) オプションボタンをクリックする。
- メソッドに set*propertyname* または get*propertyname* という名前を付ける (*propertyname* はプロパティ名)

プロパティの形式を次に示します。

```
[id(DISPID_CLASSNAME_SETPROPERTYNAME), propput]
    HRESULT PropertyName (parameter_descriptions);
[id(DISPID_CLASSNAME_GETPROPERTYNAME), propget]
    HRESULT PropertyName (parameter_descriptions);
```

説明

| | |
|---|---|
| *CLASSNAME* | クラス名 |
| *METHODNAME* | メソッド名 |
| *property_name* | プロパティ名 |
| *parameter_descriptions* | メソッドで使用されるパラメータに関する 1 行以上の説明 |

set メソッドにはプロパティの設定元を指定する USING パラメータ、get メソッドにはプロパティの値を返すための RETURNING パラメータが、それぞれ 1 つあります。COBOL のデータ型と COM オートメーションのデータ型は、『*COM データ型*』の章で説明するように、Object COBOL ランタイムシステムによって相互変換されます。

インスタンスメソッドを追加する方法については、ヘルプトピックの『クラスへのメソッドの追加』を参照してください。

メソッドを追加または削除した場合には、そのオブジェクトのタイプライブラリも更新する必要があります。この処理は、メソッドウィザードを使用すると自動的に実行されます。詳細については、『*タイプライブラリの更新*』を参照してください。

次の例は、COM オートメーションクライアントから LoanTermYears というプロパティを設定するメソッドと、YearPayment というプロパティを取得するメソッドを示しています。

```
method-id. "SetLoanTermYears".
linkage section.
01 NewLoanTermYears pic 99.
procedure division using NewLoanTermYears.
    move NewLoanTermYears to LoanTermYears
```

```
     exit method.
end method "SetLoanTermYears".

method-id. "GetYearPayment".
linkage section.
01 CurrentYearPayment pic $9(7).99.
procedure division returning CurrentYearPayment.
     move YearPayment to CurrentYearPayment
     exit method.
end method "GetYearPayment".
```

### タイプライブラリの更新

Net Express のクラスウィザードで生成されるタイプライブラリには、次の情報が含まれています。

- クラス内の各メソッドの名前、ディスパッチ ID、およびパラメータのデータ型
- 各プロパティの名前とデータ型

Net Express の COM レジストリファイルジェネレータを使用すれば、作成済みの COM オートメーションクラスのタイプライブラリも、必要に応じて逐次作成できます。詳細については、ヘルプトピックの『COM レジストリファイルの生成方法』を参照してください。

プロパティやメソッドを追加するには、メソッドウィザードを使用する必要があります。メソッドウィザードを開くと、そのたびに新しいインターフェイス情報がタイプライブラリに自動的に反映されます。

また、プロパティやメソッドを削除したり、メソッド内のパラメータやプロパティの設定、取得に使用するパラメータのデータ型を変更することもできます。詳細については、次のヘルプトピックを参照してください。

タイプライブラリからのメソッドの削除方法

メソッドまたはプロパティ set や get によるパラメータのデータ型の変更方法

## COM オブジェクトのデバッグ

作成した COM オブジェクトは Net Express でデバッグします。プロセス内とプロセス外の COM オブジェクトでは、デバッグの手順がやや異なります。

### プロセス外オブジェクトのデバッグ

COM クライアントとサーバが互いに別プロセスで動作している場合、サーバをデバッグするには Net Express のコピーを起動する必要があります。この場合、デバッグは次の 2 通りの方法で実行できます。

- Net Express のデバッガを介してサーバのトリガを起動する方法
- CBL_DEBUGBREAK の呼び出しを追加する方法

トリガを起動してデバッグする手順は次のとおりです。

1. Net Express を起動し、サーバアプリケーションのトリガプログラムのアニメートを開始します。
2. 次のいずれかの文に到達するまで、トリガプログラムをステップ実行します。

   ```
   invoke olesup "becomeServer"
   invoke anEventManager "run"
   ```

3. 上記の文に到達すると、Net Express のアニメータが応答を停止します。

   サーバがイベントループに入り、COM クライアントからメッセージを受信するまで待機します。この段階で、クライアントアプリケーションの実行やデバッグを行うことができます。クライアントがサーバにメッセージを送信すると、呼び出された COM サーバメソッドの先頭文に指定されている実行ポイントで、Net Express アニメータが動作を再開します。

   この段階で、サーバメソッドをデバッグできます。EXIT METHOD 文に到達するとイベントループに戻り、サーバが次のメッセージを受信するまで Net Express のアニメータが再度応答を停止します。

呼び出しを追加してデバッグする手順は次のとおりです。

1. ソースコード内でデバッグを開始する位置を決定します。
2. 次の文を追加します。

   ```
   CALL "CBL_DEBUGBREAK"
   ```

3. サーバをリビルドします。
4. クライアントプログラムを実行します。

   サーバの実行が CBL_DEBUGBREAK の呼び出しに到達すると、アニメートを開始するかどうかを尋ねるメッセージボックスが表示されます。

通常の方法でクライアントを実行すると、サーバだけがデバッグされます。クライアントはデバッガ内で起動することも可能です。その場合には、クライアントからサーバへのメッセージ送信をステップ実行するに伴い、2 つのデバッガ間で制御が切り替えられます。Object COBOL で作成したクライアントをデバッグするには、Net Express のインスタンスをもう 1 つ起動する必要があります。その他の言語でクライアントが作成されている場合には、その言語製品のベンダーが提供しているデバッグツールを使用してください。

### プロセス内サーバ

プロセス内サーバは、クライアントと同じアドレス空間にロードされます。Object COBOL で作成したクライアントは Net Express でデバッグできます。クライアントからサーバへのメッセージ送信をステップ実行するに伴って、サーバのコードが Net Express のアニメータに表示されます。

その他の言語でクライアントが作成されている場合には、new クラスメソッドを Object COBOL の COM オートメーションサーバに追加し、new メソッド内に次の文を記述します。

```
call "CBL_DEBUGBREAK"
```

クライアントがサーバを起動すると、Net Express が自動的に起動し、サーバコードのデバッグが可能になります。

# 第 15 章 : COM オートメーションサーバの実行

この章では、COM オブジェクトを登録して、COM オートメーションサーバとして機能させる方法を説明します。

## COM オブジェクトの登録

COM オブジェクトを COM オートメーションサーバとして機能させ、クライアントからのメッセージを受信できる状態にするには、Windows システムに登録する必要があります。Object COBOL オートメーションサーバは、次の 4 種類の方法で登録できます。

- サーバを実行する。

  Object COBOL オートメーションサーバが、実行時に自動的に登録されます。ただし、この方法は単独で動作するプロセス外サーバのみで有効であり、登録される情報も最小限です。

- Microsoft の **regsvr32** ユーティリティを使用してプロセス内サーバ (**.dll** ファイル) を有効化する。

  この方法ではサーバが永続的に登録されますが、サーバの **.dll** ファイルに追加コードが必要になります (この追加コードは、クラスウィザードで自動的に生成されます)。インターフェイスマッパーで作成した COM オブジェクトは、この方法で登録する必要があります。詳細については、後述する『*自己登録の有効化*』を参照してください。

- プロセス外サーバ (**.exe** ァイル) の起動時に、永続的に自己登録できるようにする。

  詳細については、『*自己登録の有効化*』を参照してください。

- サーバの詳細情報を Windows のシステムレジストリに直接入力する。

  サーバはシステムに永続的に登録されます。

1 番目の方法は、サーバの開発およびデバッグ時に特に役立ちます。クライアントによるサーバの起動要求の送信タイミングが特定できない場合は、それ以外の方法を使用します。レジストリに登録したサーバは、クライアントが新しいサーバのインスタンスを要求したときに、Windows によって自動的に起動されます。

Visual Basic など一部の開発環境では、作成した COM オブジェクトが開発マシンに自動的に登録されます。この処理はバックグラウンドで、いっさいの通知を伴わずに実行されます。ただし、そのオブジェクトのディプロイ先の各マシンでは、明示的に登録する必要があります。

Net Express のクラスウィザードで COM オートメーションサーバを作成すると、そのオブジェクトの登録に必要なすべての情報を含む **.reg** ファイルが生成されます。これは ASCII テキストファイルで、Windows のレジストリに挿入される各エントリを含んでいます。このファイルの形式については、『*レジストリエントリの編集*』を参照してください。

この情報によって、クライアントはオブジェクトの ProgID から一意の ID (オブジェクトの CLSID) を取得できます。通常、Object COBOL クラスの ProgID はクラスのファイル名ですが、queryClassInfo メソッドを編集すれば、異なる値にすることも可能です (次項を参照)。クライアントは CLSID を使用してレジストリ内を検索し、COM オートメーションサーバの起動に必要な情報を含む他のエントリを検出します。

**.reg** ファイル内の情報は、2 通りの方法で Windows レジストリに登録できます。 具体的な操作は次のとおり。

- 対象の **.reg** ファイルを Net Express の [**プロジェクト**] メニューで右クリックし、ポップアップメニューの [**登録**] をクリックする。
- コマンドプロンプトで次のコマンドを実行する。

  regedit *filename*.reg

## 自己登録の有効化

COM オートメーションサーバは、登録に必要なすべての情報を含むようにビルドできます。登録情報は、各 COM オートメーション クラス内の 2 つのメソッドに格納されます。

- queryClassInfo

  クラスに関する情報を返すメソッド

- queryLibraryInfo

  クラスタイプライブラリに関する情報を返すメソッド

これらのメソッドは、COM オートメーションクラスの作成時にクラスウィザードによって生成されます (queryLibraryInfo は、タイプライブラリを同時に生成しない場合は生成されません)。これらのメソッドで返される情報の一部を変更すれば、クラスの自己登録の詳細情報を変更できます。

queryClassInfo メソッドは次のように定義されています。

```
method-id. queryClassInfo.
linkage section.
01 theProgId            pic x(256).
01 theClassId           pic x(39).
01 theInterfceId        pic x(39).
01 theVersion           pic x(4) comp-5.
```

```
 01 theDescription        pic x(256).
 01 theThreadingModel     pic x(20).
 procedure division using by reference theProgId
                          by reference theClassId
                          by reference theInterfceId
                          by reference theVersion
                          by reference theDescription
                          by reference theThreadModel.
     move z"{clsid}" to theClassId
     move z"{clsid}" to theInterfceId
     move z"A description of your class" to theDescription
*> DLL を自動登録する場合は、レジストリエントリ
*> ThreadingModel を指定します。
     move z"Apartment" to theThreadModel
     exit method.
 end method queryClassInfo.
```

{*clsid*} は CLSID を表す 128 ビット長の文字列 ({F08FCA68-286C-11D2-831F-B01A09C10000} など) です。この値は変更しないでください。CLSID は Windows API の呼び出しによって生成され、すべてのクラスに一意の値が割り当てられています。CLSID は常に同じ値であることを前提に、クラスとそのインターフェイスの登録情報の一部として、さまざまな場所で使用されるため、値を変更すると予期しない結果につながります。クラスの ProgID は、ヌル値で終端する新しい ProgID の文字列をデータ項目 **theProgId** に転記することによって変更できます。バージョン番号は、現時点では COBOL ランタイムシステムで使用されません。

queryLibraryInfo メソッドは次のように定義されています。

```
 method-id. queryLibraryInfo.
 linkage section.
 01 theLibraryName        pic x(512).
 01 theMajorVersion       pic x(4) comp-5.
 01 theMinorVersion       pic x(4) comp-5.
 01 theLibraryId          pic x(39).
 01 theLocale             pic x(4) comp-5.
 procedure division using by reference theLibraryName
                          by reference theMajorVersion
                          by reference theMinorVersion
                          by reference theLibraryId
                          by reference theLocale.
                                       *> 現在は使用されません。
     move 1 to theMajorVersion
     move 0 to theMinorVersion
     move z"{clsid}" to theLibraryId
     exit method.
 end method queryLibraryInfo.
```

queryClassInfo メソッドでは、ライブラリ ID の値は変更しないでください。theTypeLibrary の値を次のいずれかに設定すれば、クラスのタイプライブラリの場所を指定できます。

| | |
|---|---|
| ヌル文字列 | タイプライブラリがクラスの **.dll** ファイルまたは **.exe** ファイルにリソースとして組み込まれていることを示します。クラスウィザードでクラス作成した場合、タイプライブラリはデフォルトで、この場所に格納されます。 |
| *n* | 数値。タイプライブラリがクラスの **.dll** ファイルまたは **.exe** ファイルのリソースであることを示します。 |
| *path* | タイプライブラリのファイル名を含む絶対パスを指定します。 |

**theMajorVersion** と **theMinorVersion** に値を設定し、タイプライブラリのメジャーバージョン番号とマイナーバージョン番号を指定することもできます。

次の 2 つの項では、プロセス内サーバとプロセス外サーバの登録手順を説明します。

**プロセス外サーバ**

プロセス外サーバの場合は、自己登録および自己登録解除を有効にすることが推奨されます。自己登録は通常、コマンド行スイッチで制御します。 詳細については、ヘルプトピックの『実行可能ファイルを使用したアウトオブプロセス COM オブジェクトの登録方法』を参照してください。

olesup クラスの registerServer メソッドと unRegisterServer メソッドを使用すれば、自己登録と自己登録解除を容易に行うことができます。サーバのトリガプログラムに、登録と登録解除のコマンド行スイッチを検出するためのコードを追加する必要があります。

トリガプログラムが登録スイッチを検出すると、次のコードが実行されます。

```
invoke olesup "registerServer" using theClass commandline
                              returning error-code
```

olesup の registerServer メソッドは、queryClassInfo メソッドと queryLibraryInfo メソッドを使用して、theClass にクラスとタイプライブラリの情報を問い合わせます。

トリガプログラムが登録解除スイッチを検出したときに、次のコードを実行します。

```
invoke olesup "unregisterServer" using theClass
                              returning error-code
```

説明

| データ項目 | データ型 | 説明 |
|---|---|---|
| *theClass* | OBJECT | 登録するクラス |

| | REFERENCE | |
|---|---|---|
| *commandline* | PIC X(256) | サーバを起動するコマンド行 (サーバのパスを含める) |
| *error-code* | PIC X(4) COMP-5 | 登録に成功すると 0 を返し、失敗すると OLE エラーコードを返す |

**プロセス内サーバ**

Microsoft regsvr32 ユーティリティを使用すれば、プロセス内サーバ (**.dll** ファイルとしてビルドしたサーバ) をコマンド行から登録または登録解除できます。 詳細については、ヘルプトピックの『インプロセス COM オブジェクトの登録方法』を参照してください。

クラスライブラリウィザードによって自動生成される queryClassInfo メソッドと queryLibraryInfo メソッドに加え、プロセス内サーバの **.dll** ファイルには、regsvr32 ユーティリティによる登録が機能するように、DllOleLoadClasses という名前のエントリポイントを含める必要があります。クラスライブラリウィザードでクラスのトリガプログラムを生成するオプションを選択した場合は、このエントリポイントも自動的に生成されます。

次に DllOleLoadClasses のコード例を示します。

```
*> COM サーバトリガ (プロセス内サーバ専用)
*> 警告：このファイルには、他のエントリポイントは追加しないでください。
$set case
 linkage section.
 01 loadReason    pic x(4) comp-5.
 procedure division.
 entry "DllOleLoadClasses" using by value loadReason.
*> OCWIZARD — クラスを起動します。
*> 呼び出されたクラスは、COM クライアントから使用できるクラスとして登録されます。
     call "myserver"
*> OCWIZARD — クラスを終了します。
      exit program.
        .
```

この例では、ファイル名 **myserver** で呼び出された 1 つのクラスだけを登録します。

DllOleLoadClasses には、必要に応じて独自のコードを追加できます。パラメータ **loadReason** は、COBOL ランタイムシステムがこのエントリポイントを呼び出した理由を示します。値は次のいずれかです。

1 サーバをロードする

2 クラスを登録する

3 サーバの登録を解除する

# レジストリエントリの編集

Net Express では、COM オブジェクトを容易に登録できる **.reg** ファイルが生成されます。この項では、サーバのパスを変更する必要が生じた場合に備え、このファイルの形式について説明します。**.reg** ファイルは、次の形式で記述された情報を含む ASCII ファイルです。

```
REGEDIT
[HKEY_CLASSES_ROOT\classname]

@ = description

[HKEY_CLASSES_ROOT\classname\Clsid]

@ = uuid

[HKEY_CLASSES_ROOT\CLSID\uuid]

@ = description

[HKEY_CLASSES_ROOT\CLSID\uuid\ProgID]

@ = classname

[HKEY_CLASSES_ROOT\CLSID\uuid\servertype]

@ = startup
```

**説明**

*classname* プログラム ID (ProgID)。クライアントは、この ID で COM オートメーションサーバを識別します。ProgID の値は通常、ベースファイル名です。

*description* サーバに関する簡単なコメント。

*uuid* Windows API の theCreateGUID() 関数で生成された一意の数値 ID。この ID は、UUIDGEN ツールを実行して生成することも可能です (詳細については後述します)。

*startup* サーバを起動するトリガプログラム。トリガプログラムを **.exe** ファイルとしてコンパイルする場合は、トリガプログラムの名前のみを指定します。**.int** ファイルまたは **.gnt** ファイルとしてコンパイルした場合には、COBOL の run コマンドを使用する必要があります。

このパラメータの値は、**.exe** ファイル名、バッチファイル名、COBOL の run コマンドのいずれかです。次に run コマンドの記述例を示します。

```
run mytrigger
```

**.exe** ファイルまたはバッチファイルの名前を指定する場合、該当するファイルは環境変数 PATH で設定されるシステムパス内のディレクトリに存在するか、あるいは *startup* パラメータの一部としてパスを明示的に指定する必要があります。

*servertype* COM サーバの種類。通常的な種類には、LocalServer32 と InProcServer32 があります。

UUIDGEN ツールは、Microsoft Platform SDK の一部です。UUIDGEN を実行すると、そのたびに一意の 16 進数が生成されます。

Windows レジストリに情報を入力するには、コマンドプロンプトで次のようなコマンドを実行します。

```
regedit registryfile
```

*registryfile* には、レジストリエントリを含むファイルの名前を指定します。

別の オートメーションサーバは、他の Windows システムにインストールするたびに登録する必要があります。それぞれのマシンで同じレジストリファイルを使用できますが、レジストリファイルに含まれるパスがマシンごとに異なる場合には、その部分を編集する必要があります。

---

**注記:**

レジストリファイルの形式と構造は一様ではありません。詳細については、Microsoft Platform SDK のドキュメントを参照してください。

---

# スレッドオプションの設定

この項では、マルチスレッドサーバをビルドして実行するために必要なすべての情報を示します。Object COBOL のサーバは通常、シングルスレッドサーバとしてビルドされます。シングルスレッドの COBOL プログラムは、シングルスレッド以外の COM スレッドオプションでは実行できません。

Net Express でマルチスレッドプログラムをビルドする方法の詳細については、Net Express のツールバーで [**ヘルプ > ヘルプトピック**] をクリックし、表示されるウィンドウの「目次」タブで『**プログラミング > マルチスレッドプログラミング**』を順にクリックして表示される情報を参照してください。

COM サーバには、次のスレッドオプションがあります。

- シングルスレッド
- アパートメント ― 各オブジェクトがシングルスレッドで動作する。
- フリー ― 完全なマルチスレッド。メソッドの違いを問わず、あらゆるスレッドからオブジェクトが呼び出し可能であるため、特定のスレッド専用のデータは使用できない。
- 両方 ― アパートメントまたはマルチスレッド (プロセス内サーバのみ)

プロセス外サーバとプロセス内サーバでは、COM によるスレッド管理の方法が異なります。

### プロセス外サーバのスレッド管理

プロセス外サーバは、シングルスレッド用にビルドするとシングルスレッドで動作 (COBOL プログラムのデフォルト) し、マルチスレッド用にビルドするとフリースレッドで動作します。

マルチスレッドのプロセス外サーバをアパートメントスレッドで動作させるには、サーバの起動コード内で、クラスの実行前に次の文を追加してください。

```
invoke olesup "singleThread" returning aBoolean
```

*aBoolean* は PIC X(4) COMP-5 で、呼び出しに成功すると 1 に設定されます。

### プロセス内サーバのスレッド管理

Net Express のクラスウィザードとインターフェイスマッパーでは、プロセス内サーバは通常、アパートメントスレッドで動作するようにビルドされます。プロセス内サーバのスレッドオプションを変更するには、レジストリの該当エントリを編集する必要があります。

1. サーバのレジストリエントリ内で InProcServer32 キーを見つけます。

   このキーはレジストリ内の [HKEY_CLASSES_ROOT\{*clsid*}\InProcServer32] にあります (*clsid* はサーバの CLSID)。

2. このキーで ThreadingModel という名前が付いた値を、次のいずれかに変更します。
   - Single
   - Apartment (デフォルト)
   - Free
   - Both

---

**注記**: Net Express が生成するレジストリファイルには、ThreadingModel という名前が付いた値はありません。また、regsvr32 の実行時に生成されるレジストリエントリにも、この値は含まれません。

このキーは Net Express が生成したレジストリファイルに追加できるほか、レジストリを直接編集したり、Windows API を使用してサーバの登録時にレジストリを編集するコードを記述する方法でも追加できます。コードを記述する場合は、クラスウィザードが生成したプロセス内ト

リガプログラムのエントリポイント DllLoadClasses に、作成したコードを追加します。詳細については、『*プロセス内サーバ*』を参照してください。

---

# 終了オプションの設定

プロセス外サーバは通常、最後のクライアントがログアウトすると、自動的に終了します。ところが、サーバがデスクトップにウィンドウを表示している場合など、この振る舞いが望ましくない場合もあります。この振る舞いは、OLESup クラスの setOLETerminateOption メソッドを使用すれば変更できます。

```
invoke olesup "setOLETerminateOption" using by value option
```

0 ウィンドウが表示されていない場合のみサーバを終了する。

1 コンソールウィンドウ以外のウィンドウが表示されていない場合のみサーバを終了する。
(Object COBOL の GUI アプリケーションは、DISPLAY 文を使用している場合、コンソールウィンドウを開きます)。

2 開いているウィンドウの有無に関係なくサーバを終了する (デフォルト)。

# 第 16 章 : COM クライアントの作成

この章では、COBOL による COM クライアントの作成について説明します。

## 概要

COBOL プログラムは COM オブジェクトに対して、次の操作を実行できます。

COM オブジェクトへのメッセージ送信

- COM プロパティの設定または取得

これらの操作によって、COM オートメーションインターフェイスを持つあらゆる Windows アプリケーションを、COBOL からダイレクトに制御することが可能になります。このような形で COM オブジェクトを使用するプログラムは、「COM クライアント」と呼ばれます。

COM オブジェクトは、クライアントとは異なる言語で作成できます。クライアントとオブジェクトの間で交換できるデータの型と形式は、COM の仕様に従って定義されます。COBOL プログラムから COM オブジェクトに送信されるすべてのデータには、『COM データ型』の章で紹介する規則に準じた型が、厳密に適用されます。

COBOL の COM オートメーションサポートでは、COBOL クライアントが使用する各 COM オブジェクトにプロキシが割り当てられ、これらの COM オブジェクトが COBOL オブジェクトとして表されます。COM クライアントはプロキシにメッセージを送信し、COBOL ランタイムシステムによってメッセージが COM オブジェクトに転送されます。

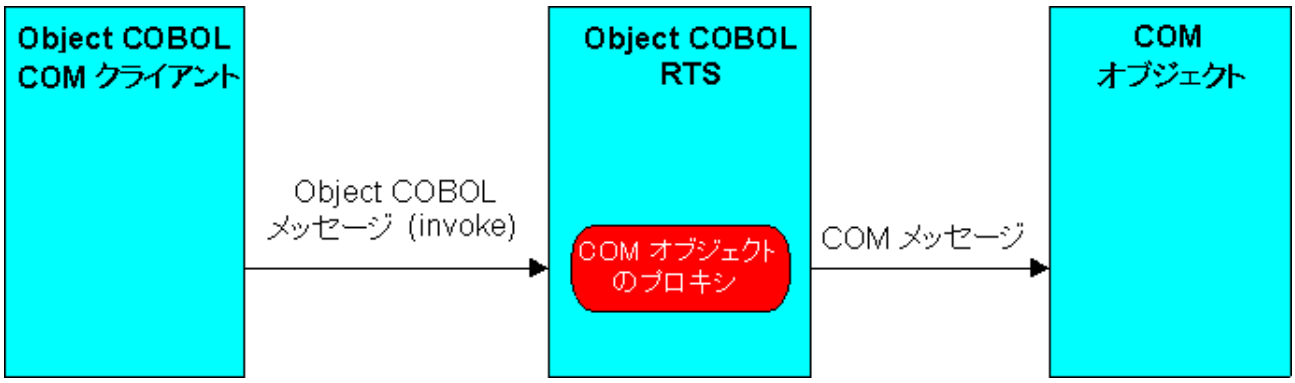



**図 16-1 クライアントから COM プロキシへのメッセージ送信**

COM クライアントの作成手順を、次の各項を通じて説明します。

- COM クライアントとしてのプログラムの有効化
- COM プロキシオブジェクトの作成

- COM オブジェクトへのメッセージ送信
- COM プロキシオブジェクトの終了
- COM オートメーションの例外
- タイプライブラリ情報の使用

# COM クライアントとしてのプログラムの有効化

任意の COBOL プログラムを COM クライアントとして設定できます。COBOL クラスを作成する必要はありません。そのために必要な手順は次のとおりです。

1. プログラムのコード内にコンパイラ指令 OOCTRL(+P) を設定します。

   ```
   $set ooctrl(+p)
   ```

   **注記**:この指令を追加すると、メソッドの呼び出しや宣言に含めることができるパラメータの数が 31 個 までに制限されます (そのほかに、必要に応じて RETURNING パラメータを使用できます)。

2. クラス制御節で、使用する COM オブジェクトを COBOL クラス名に 1 対 1 でマッピングします。
3. `  class-control.`
4. 

   ```
   class-name IS CLASS "$OLE$windows-registry-name"
   ```

   *windows-registry-name* は、COM オブジェクトの Windows システムレジストリ内での登録サーバ名です。ProgID と CLSID のどちらを指定してもかまいません。ProgID は名前を示す文字列、CLSID は 16 バイトの数値です。全世界のすべての COM オブジェクトは、互いに異なる CLSID を持っています。CLSID は、COM オブジェクトの新しいバージョンのリリースごとに変更される可能性が高いため、通常は ProgID を使用します。

   **例**

   ```
   class-control
       Word is class "$OLE$word.basic"
       HTMLHelp is class
                  "$OLE${adb880a6-d8ff-11cf-9377-00aa003b7a11}"
       .
   ```

5. COM オブジェクトのプロキシに割り当てるハンドルの格納領域として、BJECT REFERENCE データ項目を宣言します。

**例**

```
$set ooctrl(+P)
 class-control.
```

```
*> comploan は COM オブジェクトの名前 (ProgID)
     comploan is class "$OLE$comploan"
     ...
     .

 working-storage section.
 01 theCompLoanServer            object reference.
```

# COM プロキシオブジェクトの作成

使用する COM オブジェクトごとに、プロキシオブジェクトを 1 つずつ作成します。プロキシオブジェクトの作成時には、COBOL ランタイムシステムによって次の 2 つの処理が実行されます。

- 必要な COM オブジェクトの検索を Windows に要求する (Windows が COM オブジェクトを検索し、見つかったオブジェクトが動作していなければ起動します)。
- COM オブジェクトのプロキシを生成し、オブジェクトハンドルをプロキシに割り当てる。

プロキシを作成するには、COBOL のプロキシクラスに new メッセージを送信します。この段階で、プロキシを介して COM オブジェクトにメッセージを送信することが可能になります。

**例**

```
 working-storage section.
 01 theCompLoanServer            object reference.
 ...
 procedure division.
 ...
 invoke comploan "new" returning theCompLoanServer
 ...
```

プロキシの作成時に、サーバの場所を指定することもできます。 サーバの場所を指定するには、newWithServer メッセージをサーバのマシン名とともに COBOL プロキシクラスに送信します。

**例**

```
 working-storage section.
 01 theCompLoanServer            object reference.
  ...

 procedure division.
  ...
     invoke comploan "newWithServer"
                                  using z"machine1"
                                returning theCompLoanServer
```

...

# COM オブジェクトへのメッセージ送信

プロキシを作成すると、COM オブジェクトにメッセージを送信し、プロパティを設定または取得することが可能になります。メッセージ送信とプロパティの取得/設定操作では、INVOKE 文を使ってプロキシオブジェクトにメッセージを送信します。get または set で始まる名前のメッセージをプロキシオブジェクトに送信すると、COBOL ランタイムシステムによって、COM オブジェクトのプロパティを取得または設定するメッセージに自動的に変換されます (次の図を参照)。

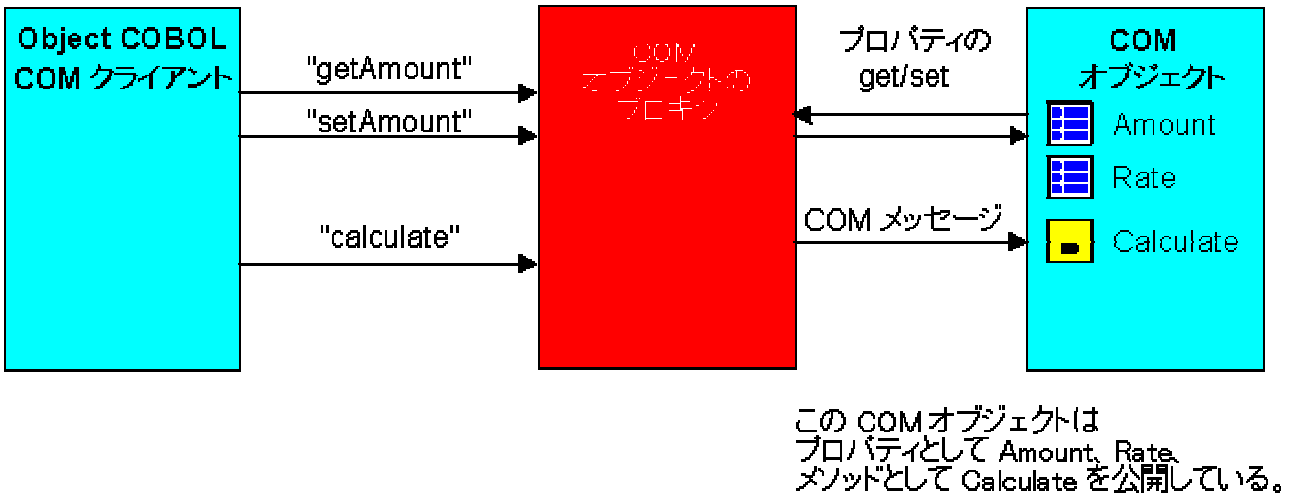



**図 16-2 メッセージ送信とプロパティの設定**

メッセージ送信の構文は次のとおりです。

```
invoke proxyObject "messagename" [using param-1 [param-2..]]
                                 [returning result]
```

| | |
|---|---|
| *proxyObject* | COM オブジェクトのプロキシ。プロキシを作成して COM オブジェクトを起動する方法については、『*COM プロキシオブジェクトの作成*』を参照してください。 |
| *messagename* | 送信するメッセージ。 |
| *param-1* | メッセージに必要なパラメータ。『*COM データ型*』の章で説明するように、COBOL データ型は COM データ型に変換されます。すべてのパラメータは、参照で渡されます (COBOL のデフォルト)。 |
| *result* | メッセージを送信したメソッドが戻り値を返す場合、その値が格納されます。『*COM データ型*』の章で説明するように、COBOL データ型は COM データ型に変換されます。 |

プロパティを設定するには、次のように記述します。

invoke *proxyObject* "set*PropertyName*" using *value*

*proxyObject* COM オブジェクトのプロキシ。プロキシを作成して COM オブジェクトを起動する方法については、『*COM プロキシオブジェクトの作成*』を参照してください。

*PropertyName* 設定するプロパティの名前。

*value* プロパティに設定する値。『*COM データ型*』の章で説明するように、COBOL データ型は COM データ型に変換されます。値は参照で渡されます (COBOL のデフォルト)。

プロパティを取得するには、次のように記述します。

invoke *proxyObject* "get*PropertyName*" returning *value*

*proxyObject* COM オブジェクトのプロキシ。プロキシを作成して COM オブジェクトを起動する方法については、『*COM プロキシオブジェクトの作成*』を参照してください。

*PropertyName* 値を取得するプロパティの名前。

*value* プロパティの値。『*COM データ型*』の章で説明するように、COBOL データ型は COM データ型に変換されます。取得した値が格納されるメモリ領域は、COM によって上書きされる可能性があるため、値を保持する必要があるときには取得後にコピーしてください。

次の例は、プロパティを設定し、メッセージを送信した後、プロパティを取得します。

```
working-storage section.
 ...
 01 theCompLoanServer       object reference.
 01 AnAmount                pic 9(7).99.
 01 ARate                   pic 99.99.
 01 Amount20                pic $9(7).99.
 ...

 procedure division.
 ...
      invoke theCompLoanServer "SetLoanTermYears" using
           "20"
      invoke theCompLoanServer "calculate"
      invoke theCompLoanServer "GetYearPayment" returning
           Amount20
 ...
```

## COM メッセージタイプの強制的な適用

ランタイムシステムは通常、COBOL から受信したメッセージが set または get で始まる場合、それらのメッセージをすべてプロパティを設定または取得する操作に変換します。ただし、この動作は変更可能です。また、先頭に set または get が付いていないメッセージを、プロパティを設定または取得する操作に強制的に変換することもできます。特定のメッセージタイプを強制的に適用するには、メッセージ "setDispatchType" を OLEsup クラス (ファイル名は **OLEsup**) に送信します。

```
invoke OLEsup "setDispatchType" using by value lsType
```

説明

*lsType*　PIC X(4) COMP-5

value 0 = 次のメッセージでメソッドを呼び出す。
value 1 = 次のメッセージでプロパティの値を設定する。
value 2 = 次のメッセージでプロパティの値を取得する。

指定したメッセージタイプは、次に送信する COM メッセージだけに適用され、それ以降は "setDispatchType" を送信するまでデフォルトの動作に戻ります。OLEsup クラスには、COM オートメーションプログラミングに役立ついくつかの関数が含まれています。OLEsup クラス の詳細については、『*COM Automation Class Library*』を参照してください。

# タイプライブラリ情報の使用方法

COM オブジェクトのタイプライブラリには、クライアントの作成に役立つ情報が定義されています。タイプライブラリアシスタントを使用すれば、あらゆるライブラリ用の COBOL コピーファイルを作成できます (タイプライブラリアシスタントを使用するには、Net Express の [**ツール**] メニューで [**タイプライブラリアシスタント**] をクリックします)。

タイプライブラリ内の情報に応じて、COBOL コピーファイルには次の情報が生成されます。

- タイプライブラリで定義されている各データ型に対応する COBOL 型定義
- ライブラリで定義されているオートメーションインターフェイスごとに次の情報が生成される。
  - タイプライブラリで定義されている各インターフェイスの IID (インターフェイス ID) を含むレベル 78 データ項目
  - インターフェイス記述を示すコメント
  - インターフェイスの名前を含むコメント化されたポインタデータ項目
  - インターフェイスの各プロパティの設定方法と取得方法を示すコメント
  - インターフェイスの各メソッドの呼び出し方法を示すコメント
- タイプライブラリで定義されているクラスごとに次の情報が生成される。
  - ライブラリで定義されている各クラスの CLSID と ProgID を含むレベル 78 データ項目
  - クラスを使用するためにクラス制御節に記述すべきエントリを示すコメント
  - クラス記述を示すコメント

- 各列挙型のデータ項目

# COM プロキシオブジェクトの終了

COBOL ランタイムシステムが、COM オブジェクトを操作するために作成されたプロキシオブジェクトを、自動的に終了させることはありません。プロキシオブジェクトを終了するには、COM オブジェクトの操作を終了した後、"finalize" メッセージをプロキシに送信する必要があります。不可視オブジェクトではないか、明示的なメソッド呼び出し (Microsoft Word を終了する "quit" の呼び出しなど) を必要とする場合を除き、COM オブジェクトは通常、接続がすべて解除されると自動的に終了します。

**例**

```
invoke theCompLoanServer "finalize"
                                    returning theCompLoanServer
```

# COM オートメーションの例外

COM クライアントは COM エラーの通知を COBOL 例外として受信します。オブジェクトで認識できないメッセージを送信した場合など、COM オブジェクトで COM エラーが発生したときには、常にエラーが例外として COBOL で作成された COM クライアントに返されます。デフォルトでは、例外を受信したクライアントは、その旨を警告するメッセージを表示して終了しますが、例外をトラップして独自のエラー処理コードで処理することも可能です。

次の手順説明は、『*Exception Handling Frameworks*』の記載情報に関する知識を前提としています。

COM 例外をトラップする手順は次のとおりです。

1. COM クライアントのクラス制御節で、ExceptionManager、OLEExceptionManager、OLEsup の各クラスと、Callback または EntryCallback (ステップ 2 を参照) を宣言します。

```
2.   class-control.
3.        ...
4.        OLEExceptionManager is class "oleexpt"
5.        ExceptionManager is class "exptnmgr"
6.        olesup is class "olesup"
7.        Callback is class "callback"
8.        EntryCallback is class "entrycll"
        ...
```

9. 例外ハンドラ (メソッドまたはエントリポイント) を作成した後、対応する Callback または EntryCallback を作成するコードを記述します。Callback を作成する場合は、次のように記述します。

```
10.  invoke Callback "new" using anObject z"methodName"
```

```
                    returning aHandler
```

EntryCallback を作成する場合は、次のように記述します。

```
invoke EntryCallback "new" using  z"entryPointname"
                    returning aHandler
```

11. Callback または EntryCallback を OLEExceptionManager クラスに登録します。次に記述例を示します。
12. 
```
invoke ExceptionManager "register"
          using OLEExceptionManager aHandler
```

以上のコードを追加すると、クライアントに送信される COM 例外が、すべて例外ハンドラに送信されるようになります。

例外ハンドラは 2 つのパラメータを受け取ります。最初のパラメータは OLEExeptionManager のオブジェクトハンドル、2 番目のパラメータは例外 ID です。COM エラーコードを取得するには、例外 ID から基本 COM 例外エラー番号を取り出す必要があります。基本 COM 例外エラー番号の値は、"getBaseOleException" メッセージを OLEsup クラスに送信すれば取得できます。具体的には、次の行を追加します。

```
invoke OLEsup "getBaseOLEException" returning lsBase
```

説明

*lsBase*  PIC X(4) COMP-5

COBOL ランタイムシステムから ExceptionManager を介して返される例外番号と、その簡単な説明を次に示します。

| 値 | 説明 |
|---|---|
| 0 | サーバで定義されている COM 例外 |
| 1 | パラメータ数が一致しません |
| 2 | COM データ型が一致しません |
| 3 | COM の名前が見つかりません |
| 4 | メモリ不足で COM 操作を実行できません |
| 5 | 名前がメソッドです |
| 6 | 名前がプロパティです |
| 7 | COM オートメーションエラー |
| 8 | COM サーバが利用できません |
| 9 | COM サーバで例外が発生しました |

例外メソッドで OLEsup クラスに "getLastSCode" メッセージを送信すれば、発生した COM エラーに関する詳しい情報を得ることができます。

```
invoke OLEsup "getLastSCode" returning lsErrorCode
```

説明

*lsErrorCode*　PIC X(4) COMP-5

COM エラーコードについては、Microsoft Platform SDK のドキュメントを参照してください。

次に示す最初のコード例 (**COM-err1.cbl**) は、COM 例外を処理するシンプルな COBOL サブルーチンです。2 番目のコード例は、このサブルーチンを EntryCallback でラップし、COM 例外ハンドラとして登録しています。

例外ハンドラの一例

```
class-control.
     OLEsup is class "OLEsup"
     .
 working-storage section.
 01 wsOffset                 pic x(4) comp-5.
 01 wsOLEException           pic x(4) comp-5.

 linkage section.
 01 lnkExceptionObject       object reference.
 01 lnkExceptionNumber       pic x(4) comp-5.
 procedure division using lnkExceptionObject
                          lnkExceptionNumber.
     ...
     invoke OLEsup "getBaseOLEException" returning wsOffset
     subtract wsOffset from lnkExceptionNumber
     giving wsOLEException  *> COM 例外
     ...
     exit program.
```

例外ハンドラの登録例

```
class-control.
    ...
     EntryCallback is class "entrycll"
     OLEexpt is class "OLEexpt"
     ExceptionManager is class "exptnmgr"
          .
```

```
working-storage section.
...
01 exceptionHandler        object reference.
...
procedure division.
...
*>   COM 例外ハンドラのセットアップ
     invoke EntryCallBack "new" using z"exception-section"
                            returning exceptionHandler
     invoke ExceptionManager "register" using OLEexpt
                                              exceptionHandler
     ...
```

# 第 17 章 : COM コンポーネントの分散配置

この章では、DCOM (Distributed COM) の概要と、DCOM を通じて COM コンポーネントを分散配置する方法を説明します。

## DCOM の使用方法

DCOM を使用すれば、ネットワーク上のさまざまなマシンに COM のクライアントとオブジェクトを分散配置できます。クライアントマシンとサーバマシンに正しいレジストリ情報を登録することによって、Object COBOL のあらゆる COM オブジェクトをネットワーク全体で実行することが可能になります。サーバが存在するリモートマシンは、クライアントマシンに登録したレジストリエントリによって検出されます。

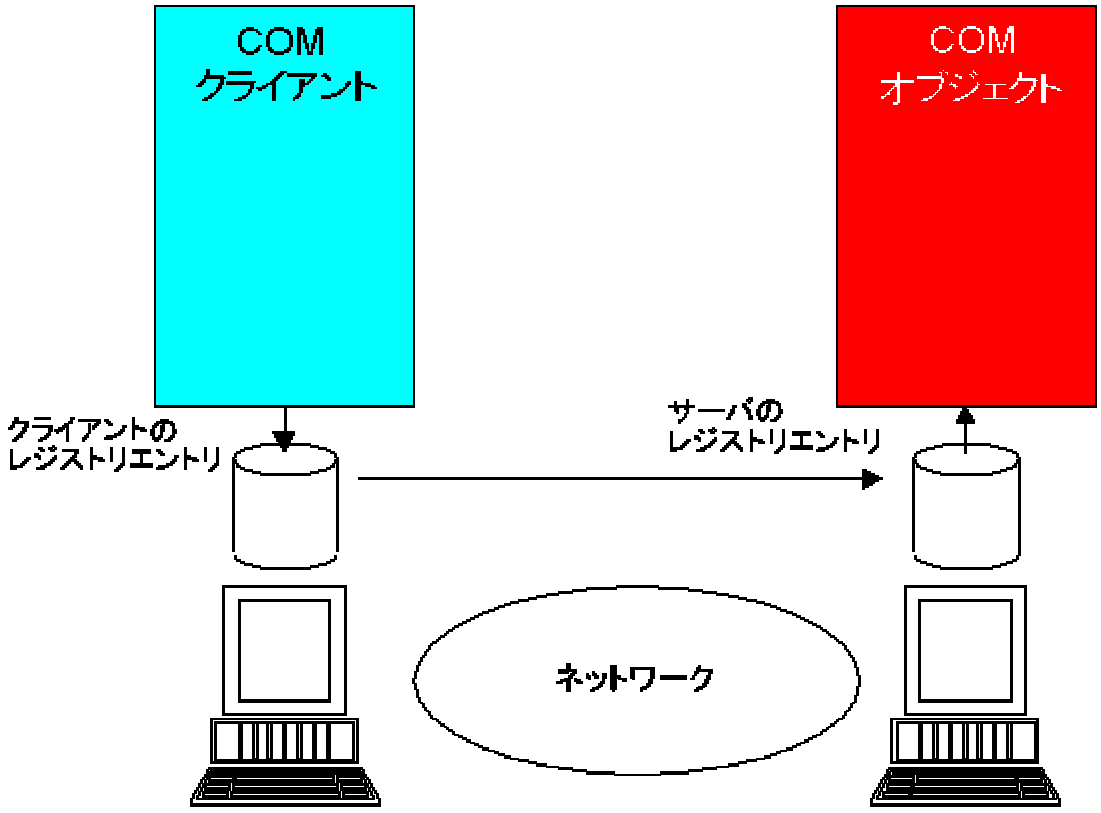



**図 17-1 リモート COM オブジェクト**

DCOM を利用するには、COM オブジェクトごとに 2 つの .reg ファイルを用意する必要があります。1 つはクライアントマシンへの登録用、もう 1 つはリモートサーバマシンへの登録用です。オートメーションサーバクラスを作成するときに、Net Express クラスウィザードでクラスをリモートとして指定すると、これらのファイルは自動的に生成されます。ただし、COM オブジェクトをローカルサーバとして開発し、完成後にリモートサーバとしてディプロイするケースも少なくありません。

詳細については、ヘルプトピックの『COM レジストリファイルの生成方法』を参照してください。

クライアントのレジストリファイルは、特定のサーバにアクセスするすべてのクライアントマシンに、そのサーバを登録するときに使用します。一方、サーバのレジストリファイルは、サーバをサーバマシンに登録するときに使用します。 サーバマシンには、Net Express または Application Server をインストールする必要があります。

## DCOM とセキュリティ

分散配置したサーバの起動や実行、およびサーバへのアクセス時には、Windows NT のセキュリティ設定に起因する問題が発生することがあります。 この項では、Windows NT で DCOM を使用する場合にチェックすべき主なポイントを示します。

- サーバの起動アクセス許可とアクセス許可を、クライアントからサーバを使用できるように設定する。

  DCOM では、このルールを守ることが必須です。Microsoft の DCOMCNFG ユーティリティを使用すれば、サーバごとに値を設定できます。すべてが正常に機能していることを確認するには、起動アクセス許可とアクセス許可をカスタマイズして各リストに Everyone を追加し、サーバと同じドメインにあるクライアントを使用します。その後、セキュリティを調整します。

- サーバを実行するユーザアカウントを設定する。

  この場合も、Microsoft の DCOMCNFG ユーティリティを使用してください。

  次のいずれかのアカウントを選択できます。

| | |
|---|---|
| 対話ユーザ | 現在サーバにログオンしているユーザアカウントでサーバを起動します。マルチユーザサーバの場合、起動すべきサーバは 1 つだけです。サーバは、画面への自動アクセス権を持っています。 |
| 起動したユーザ | クライアントアプリケーションを実行しているマシンのアカウントでサーバが起動します。マルチユーザサーバの場合、新しいクライアントがログオンするとサーバが起動します。 |
| 次のユーザ | 指定したユーザアカウントでサーバが起動します。マルチユーザサーバの場合、起動すべきサーバは 1 つだけです。 |

セキュリティの詳細については、Microsoft のドキュメントを参照してください。

---

# 第 18 章 : COM データ型

COM オートメーションでは、COM クライアントと COM サーバ間の通信に使用する一連のデータ型が定義されています。この章では、これらの COM データ型と COBOL データ型とのマッピングについて説明します。

## COM オブジェクト間のデータ交換

COM オートメーションでは、COM オブジェクト間のデータのやり取りに使用されるデータ型が独自に定義されています。次の図に示すように、Object COBOL は COM メッセージの送受信時に、COBOL データ型と COM データ型を自動的に変換します。

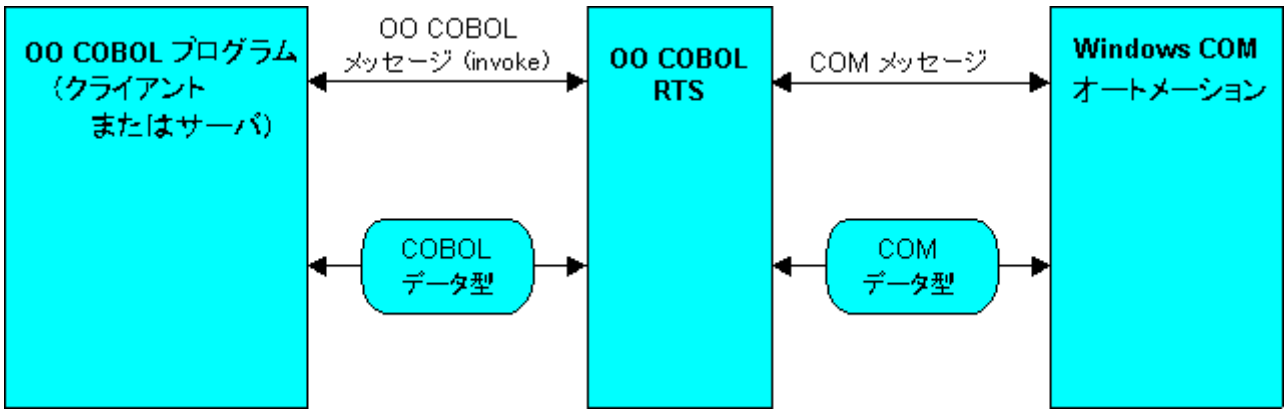



**図 18-1 COM オートメーションを介したデータのやり取り**

Object COBOL の COM コンポーネントは、次のデータ型をサポートしています。

- 2 バイトおよび 4 バイトの整数型
- 4 バイトおよび 8 バイトの浮動小数点型
- 文字列
- COM オブジェクト
- OLEVariant
- OLESafeArray

OLESafeArray と OLEVariant はどちらも複合データ型で、前者は Object COBOL の OLESafeArray クラスのインスタンス、後者は OLEVariant クラスのインスタンスとして、それぞれ Object COBOL に渡されます。他のデータ型は、相当する COBOL データ型に直接変換されます。

## COM データ型の変換規則

COBOL プログラムが COM オブジェクトにデータを送信するとき、そのデータは適切な COM データ型に変換されます。同様に、COBOL プログラムが COM オブジェクトから受信するデータは、常に COBOL データ型に変換されます。Object COBOL プログラムが COM コンポーネントを介して送受信するデータは、次に示すマッピングに従って型変換されます。

| COM データ型 | COBOL データ型 | 説明 |
|---|---|---|
| VT_I8 | PIC X(8) COMP-5<br>PIC X(8) COMP-X<br>PIC S9(18) COMP-5 | 8 バイトの整数 |
| VT_I2 | PIC X(2) COMP-5<br>PIC X(2) COMP-X<br>PIC S9(4) COMP-5 | 2 バイトの整数 |
| VT_I4 | PIC X(4) COMP-5<br>PIC X(4) COMP-X<br>PIC S9(9) COMP-5 | 4 バイトの整数 |
| VT_DISPATCH | オブジェクト参照 (OLEBase のインスタンス。不要になったオブジェクト参照は、受信側プログラムで解放する必要がある) | COM オブジェクトハンドル (『*オブジェクト参照*』の項を参照) |
| VT_R4 | COMP-1 | 4 バイトの浮動小数点数値 |
| VT_R8 | COMP-2 | 8 バイトの浮動小数点数値 |
| VT_DATE | COMP-2 | バイナリ形式の日付 (詳細は Microsoft の COM ドキュメントを参照) |
| VT_CY | COMP-3 | 通貨データ |
| VT_BOOL | PIC X(2) COMP-5 | BOOL 値 |
| SafeArray<br><br>SafeArray に対応する COM データ型は VT_ARRAY と VT_*datatype* の論理和になる (VT_*datatype* は、SafeArray がサポートするいずれか 1 つデータ型) | オブジェクト参照 (COMSafeArray のインスタンス) | *n* 次元の固定長配列 (『*SafeArray*』の項を参照) |
| Object COBOL のメソッドに渡される VT_VARIANT | オブジェクト参照 (OLEVariant のインスタンス) | Variant 型データ (データ自体と型情報を含み、この表に記載されている他のいずれか 1 つの型のデ |

| | | |
|---|---|---|
| | | ータを格納できる『*Variant*』の項を参照)<br><br>**注記**:この変換はメソッド呼び出しで渡される Variant だけに適用される (起動済みのメソッドが VT_VARIANT を返すことはない) |
| Variant の内容を基に、次のいずれかに変換される<br><br>• IDispatch<br>• SafeArray<br>• BSTR | オブジェクト参照 (Object COBOL のメソッドから返される OLEVariant のインスタンス) | このマッピングは、COM コンポーネントを介して呼び出された Object COBOL メソッドが、戻り値として OLEVariant のインスタンスの送信を試みる場合のみ実行され、かわりに Variant 型データが返される |
| VT_BSTR | PIC X(*n*) または CharacterArray のインスタンス<br><br>ランタイムシステムによって、COBOL データ項目が PIC X(*n*) として宣言されている場合は前者、OBJECT REFERENCE として宣言される場合は後者に変換される | データ長を先頭につけた文字列 |

COM コンポーネントは、これらのデータ型を数値コードで識別します。たとえば、VARIANT データ型には、この方法で識別されたデータ型の情報が格納されます。 これらの数値コードは、いずれも コピーファイル **mfole.cpy** 内で Level-78 データ項目として定義されています (このコピーファイルは **base\source** ディレクトリ内にあります)。

# オブジェクト参照

Object COBOL プログラムに COM オブジェクトが渡されると、ランタイムシステムによって COM ハンドル (VT_DISPATCH) がプロキシ Object COBOL ハンドル (オブジェクト参照) に変換されます。このプロキシオブジェクトは olebase のインスタンスであり、この Object COBOL ハンドルを使用して COM オブジェクトにメッセージを送信したり、他の COM オブジェクトにパラメータを渡すことができます。

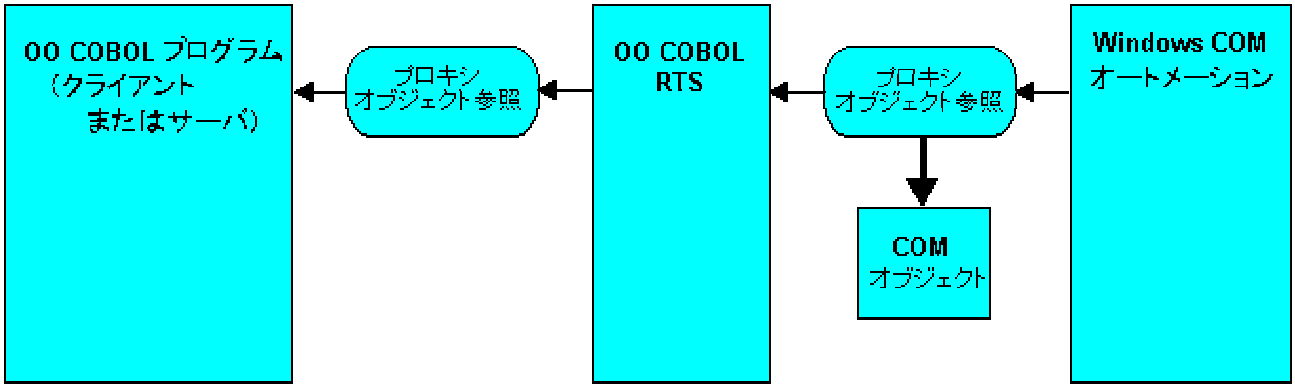



**図 18-2 VT_DISPATCH からプロキシ参照への変換**

オブジェクトハンドルを他の COM オブジェクトに送信するときには、ランタイムシステムが次の規則に従ってオブジェクトハンドルの変換を試みます。

- オブジェクトハンドルがプロキシオブジェクトのハンドルの場合、元の VT_DISPATCH 値に変換する。
- CharacterArray のハンドルの場合、VT_BSTR に変換する。
- OLEVariant のハンドルの場合、VT_VARIANT に変換する。
- SafeArray のハンドルの場合、VT_ARRAY に変換する。
- 上記以外のハンドルの場合、不明な型による例外が発生する。

オブジェクトハンドルのクラスは、olesup に getClass メッセージを送信すれば特定できます。このメッセージへの応答として返されるクラスのオブジェクトハンドルを、プログラムのクラス制御節で宣言したクラス名と照合してクラスを特定します。この処理を実装するコードの一例を次に示します。

```
class-control.
     CharacterArray is class "chararry"
     olesup is class "olesup"
     SafeArray is class "olesafea"
     OLEVariant is class "olevar"
     ...
     .
working-storage section.
   01 anObject              object reference.
   01 aClass                object reference.
   ...
 procedure division.
     ...
     invoke olesup "getClass" using by value anObject
                              returning aClass
     if aClass = CharacterArray
         display "CharacterArray"
     else
```

```
        if aClass = olebase
            display "COM object"
        else
            if aClass = SafeArray
                display "SafeArray"
            else
                if aClass = OLEVariant
                    display "Variant"
                else
                    display "Not supported by COM " &
                            "Automation"
                end-if
            end-if
        end-if
    end-if
    ...
```

# Variant

OLEVariant は、データとその型情報を格納してラップする COM データ型です。COM プログラミングでは、Variant 型は次のようにさまざまな用途に使用されます。

- パラメータとして渡されるデータの型を特定できない状態でメソッドや関数を作成する場合
- データ型の異なる複数のデータを 1 つの SafeArray に格納する場合

  SafeArray 内の各要素は、すべて同じサイズになる必要があります。Variant を格納する SafeArray を作成すれば、各要素が Variant にラップされるため、さまざまな型のデータを配列内に格納できます。

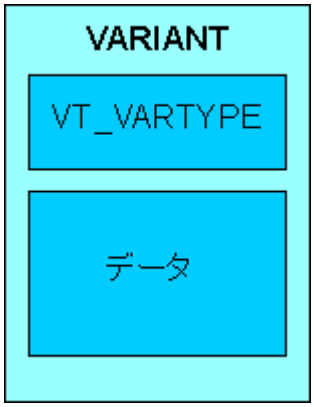



**図 18-3 OLEVariant による型情報とデータのラップ**

コピーファイル **mfole.cpy** では、OLEVariant を格納するデータ構造体 VARIANT が、新しい COBOL データ型として定義されています。このコピーファイルでは、VARIANT に格納できる

各種データ型に対応する値を定義した Level-78 データ項目セットも定義されています。次のコード例は VARIANT データ項目の VARIANT-vartype フィールドをチェックし、**mfole.cpy** で定義されている Level-78 データ項目群と照合して、VARIANT データ項目に整数または文字列が含まれているかどうかを判定します。

```
 working-storage section.
 COPY "MFOLE.CPY".
 01 aVariant          usage VARIANT.
 01 vtype             pic x(2) comp-5.
 ...
 procedure division.
      ...
*> VARIANT-vartype は VARIANT データ項目のフィールドの
*> 1 つで、型情報を格納しています。
      move VARIANT-vartype of aVariant to vType
      evaluate vType
         when VT-I2
             display "2 バイトの整数"
         when VT-I4
             display "4 バイトの整数"
         when VT-BSTR
             display "文字列"
         when other
             display "その他の型"
      end-evaluate
```

Object COBOL の COM オートメーションサポートには、VARIANT データの送受信を可能にする OLEVariant クラスがあります。この OLEVariant クラスには、文字列またはオブジェクト参照を Object COBOL ネイティブのデータ型として格納している VARIANT データにアクセスするメソッドが含まれています。他の型の VARIANT データを処理するには、VARIANT データ項目を宣言し、そのデータ項目に OLEVariant オブジェクトのデータを読み込みます。

**mfole.cpy** で定義されている VARIANT 型を使用すれば、データ項目の構造を COBOL で直接表現できるため、OLEVariant クラスは不要に思えるかもしれませんが、COM コンポーネントは VARIANT の割り当て、操作、解放に Windows API 関数セットを使用しており、OLEVariant クラスを使用すれば、COBOL からこれらの関数へのインターフェイスがシンプルになります。

この章には、OLEVariant クラスのメソッドの一部を示しています。OLEVariant のすべてのメソッドについては、Net Express ヘルプの『COM オートメーションクラスライブラリ』を参照してください。Net Express のツールバーで [**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「目次」タブで「**Reference > OO COBOL > OO Class Library Reference > COM Automation Class Library**」を順にクリックして、「**OLEVariant**」をダブルクリックします。

**base\demo\comdemos\objects** フォルダには、Variant、SafeArray などのオブジェクトの使用方法を示すサンプルプロジェクト **objects.app** があります。このサンプルの実行方法については、同じフォルダ内にある **readme.txt** を参照してください。

---

**注記:** OLEVariant データ型の詳しい説明は、Microsoft Platform SDK のヘルプに記載されています。 Microsoft Platform SDK は Net Express に付属しています。

---

## OleVariant クラスを使用する前に

Object COBOL クラスで OleVariant を使用するには、そのクラスのクラス制御節で Object COBOL クラスの OleVariant を宣言し、**mfole.cpy** を作業場所節にコピーする必要があります。

```
class-control.
     ...
    class OleVariant is class "olevar".
     ...

working-storage section.
copy "mfole.cpy".
...
```

コピーファイル **mfole.cpy** には、OLEVariant を使用するときに必要になる COBOL データ構造体の型定義が含まれています。

## OLEVariant のインスタンスの作成

OLEVariant のインスタンスを初期化する方法は、格納するデータの型によって異なります。数字データの場合は VARIANT データ項目に直接入力し、それを使用して OLEVariant インスタンスを初期化できます。文字列は Windows API 呼び出しで生成される OLE BSTRINGS として格納されます。COM オブジェクトの場合、Object COBOL プログラムで利用可能な Object COBOL プロキシ参照は使用せず、COM オブジェクト自体の VT_DISPATCH 参照を使用して格納する必要があります。OLEVariant クラスには、BSTRING を自動生成するメソッドと、プロキシ参照を格納前に VT_DISPATCH にマップし直すメソッドがあります。

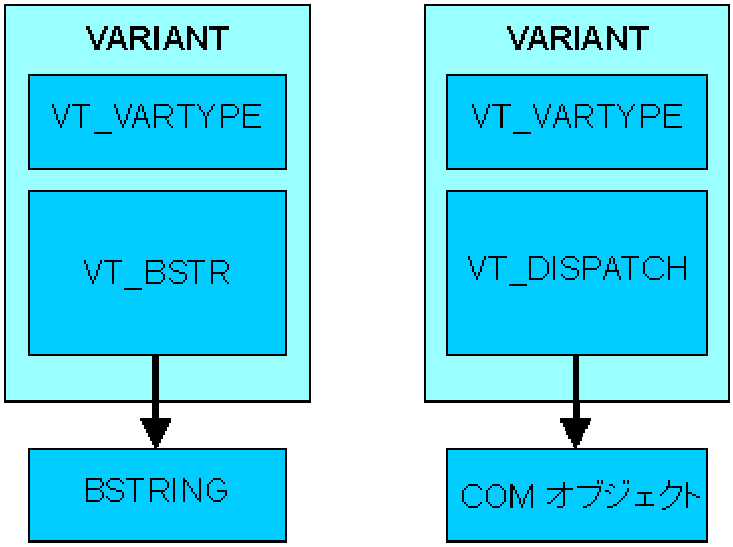



**図 18-4 Windows の文字列およびオブジェクト用の Variant**

OLEVariant オブジェクトは、上記以外にも 2 通りの方法で生成できます。文字列や COM オブジェクトを格納する場合と、その他いずれかの variant データ型を格納する場合では、生成方法が異なります。

文字列または OLEVariant オブジェクトを格納する OLEVariant インスタンスを生成するには、次の手順に従います。

1. プログラムの作業場所節にコピーファイル **mfole.cpy** を含めます。
2. OLEVarinat クラスに new メッセージを送信します。この結果、初期化されていない OLEVariant インスタンスが返されます。
3. PIC X(*n*) 内の文字列を格納するには setString メッセージ、Object COBOL CharacterArray 内の文字列を格納するには setChararry メッセージ、COM オブジェクトを格納するには setOLEObject メッセージを、このインスタンスにそれぞれ送信します。

その他の型のデータを格納する OLEVariant インスタンスを生成するには、次の手順に従います。

1. プログラムの作業場所節にコピーファイル **mfole.cpy** を含めます。
2. VARIANT データ項目を宣言して初期化します。
    1. VARIANT データ項目の VARIANT-VARTYPE フィールドのデータ型識別子を、格納するデータ型に転記します。データ型識別子は **mfole.cpy** 内で Level-78 として定義されており、いずれの識別子も先頭に "VT-" が付いています。
    2. VARIANT データ項目内の適切なフィールドにデータを転記します。VARIANT データフィールドには、VARIANT-VT-I2 のような名前が付けられています。各フィールドの具体名については、コピーファイル **mfole.cpy** 内の VARIANT 型定義を参照してください。VARIANT-VT-BSTR など一部のデータ型では、構造体へのポインタとしてデータを格納します。

3. OLEVariant クラスに newWithData メッセージを送信し、VARIANT データ項目をパラメータとして渡します。

次の例では 2 つの OLEVariant を生成し、一方に文字列型、もう一方には 4 バイトの整数型を格納しています。

```
 working-storage section.
 copy "mfole.cpy".
 01 aCharArray           object reference.
 01 aVariantinstance     object reference.
 01 variantinstance1     object reference.
 01 variantinstance2     object reference.
 01 vType                pic 9(4) comp-5.
 01 strLength            pic x(4) comp-5.
 01 winStatus            pic x(4) comp-5.
 01 aNumber              pic s9(9) comp-5.
 01 aString              pic x(12) value "I'm a string".
 01 vData2               VARIANT.
 01 vDataDisplay         VARIANT.
 ...
 procedure division.
      ...
*>---OLEVariant のインスタンスを生成して文字列を格納します。
*>    最初に空の variant インスタンスを作成します。
      invoke OLEVariant "new" returning variantInstance1
*>---文字列データを設定します。
      move length of aString to strLength
      invoke variantInstance1 "setString" using
                                        by value strLength
                                     by reference aString
                                        returning winStatus
      ...
*>---OLEVariant インスタンスを生成し、4 バイトの整数を格納
*>    します。
      move vt-I4 to variant-vartype of vData2
      move 99 to variant-vt-i4 of vData2
      invoke OLEVariant "newWithData" using vData2
                                  returning variantInstance2
      ...
```

## OLEVariant からのデータの読み取り

OLEVariant のインスタンスからデータを読み取る手順は、次のとおりです。

1. インスタンスから情報を取り込むために VARIANT 型のデータ項目を宣言します。

2. getVariant メッセージを OLEVariant のインスタンスに送信し、インスタンス内の VARIANT 構造体のコピーを取得します。
3. VARIANT-VARTYPE フィールドを評価し、データ型を特定します。
4. データ型が OLEBSTRING の場合は、getString メソッドを使用して COBOL PIC X(*n*) として取得するか、getCharArray メソッドを使用して CharacterArray として取得する必要があります。データ型が OLE VT_DISPATCH (オブジェクト参照) の場合は、getOLEObject メソッドを使用してプロキシ COBOL オブジェクト参照を取得します。

   他のデータ型については、VARIANT データ項目の適切なフィールドからデータを直接読み取ることができます。

データ型は OLEVariant インスタンスに getType メッセージを送信する方法でも特定できます。詳細については、オンラインヘルプの『COM オートメーションクラスライブラリリファレンス』を参照してください。

次のコード例は、VARIANT データ項目からデータを読み取る方法を示しています。

```
working-storage section.
copy "mfole.cpy".
01 aCharArray           object reference.
01 aVariantinstance     object reference.
01 strLength            pic x(4) comp-5.
01 winStatus            pic x(4) comp-5.
01 aNumber              pic s9(9) comp-5.
01 vDataDisplay         VARIANT.
...
procedure division.
    ...
    evaluate variant-vartype of vDataDisplay
                                      *> データ型識別子を取得
     when vt-bstr
            *> これらの定数は MFOLE.CPY 内で定義されています。
        invoke aVariantInstance "getCharArray"
                               using aCharArray
                           returning winStatus
        invoke aCharArray "display"
     when vt-I2
        move variant-vt-i2 of vDataDisplay to aNumber
        display aNumber
     when vt-I4
        move variant-vt-i4 of vDataDisplay to aNumber
        display aNumber
     when other
        display "文字列型、2 バイト整数型、4 バイト整数型のどれにも該当しません"
    end-evaluate
```

...

# SafeArray

OLE SafeArray は、複数のプロセス間で安全にやり取りできる *n* 次元の固定長配列です。COM は SafeArray を操作するための API を備えており、SafeArray の生成や破棄、および SafeArray に格納されているデータの操作を実行できます。Object COBOL では、OleSafeArray クラスを介して SafeArray を操作できます。OleSafeArray クラスは、Object COBOL の COM コンポーネントサポートで定義されています。

次元の先頭要素のインデックスは、その次元の下限になります。次元の下限は、SafeArray を定義するときに任意の整数として定義できます。たとえば、次に示す 6×7 要素の 2 次元 SafeArray で、2 つの次元の下限を 0 に設定すると赤いセルのアドレスは 2, 1 になり、下限を 1 に設定すると 3, 2 になります。

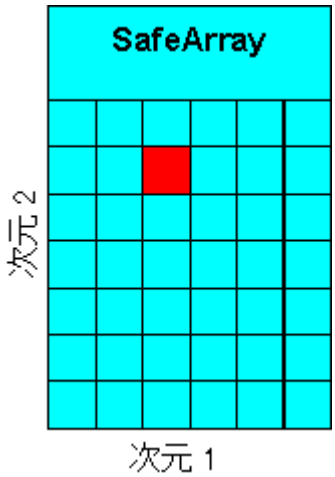



**図 18-5 2 次元の SafeArray**

この章には、OLEVariant クラスのメソッドの一部を示しています。OLEVariant のすべてのメソッドについては、Net Express ヘルプの『COM オートメーションクラスライブラリ』を参照してください。 Net Express の [**Help**] メニューで [**ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**Reference** > **OO Class Library**」を順にクリックして、『Class Library Reference』へのショートカットボタンをクリックします。

**base\demo\comdemos\objects** フォルダ内には Variant や SafeArray などのオブジェクトの使用方法を示すサンプルの **objects.app** プロジェクトがあります。このサンプルの実行方法については、同じフォルダ内の **readme.txt** を参照してください。

## SafeArray を使用する前に

Object COBOL クラスで SafeArray を使用するには、そのクラスのクラス制御節で Object COBOL クラスの OleSafeArray を宣言し、**mfole.cpy** を作業場所節にコピーする必要があります。

```
class-control.
    ...
    class OleSafeArray is class "olesafea".
    ...

working-storage section.
copy "olesafea.cpy".
...
```

## SafeArray の生成

SafeArray を生成する前に、次の情報を設定する必要があります。

- 配列の次元数

  1 以上の任意の数を指定できます。

- 各次元の境界

  境界は、次元の下限 (配列の先頭要素のインデックス) と要素数で指定します。SAFEARRAYBOUND 型データ項目のテーブルを使用して、各データ項目の境界を宣言します。このデータ型は、コピーファイル **olesafea.cpy** で定義されています。

- SafeArray に格納するデータの型

  SafeArray に格納するデータは、すべて同じ型で統一する必要があります。ただし、SafeArray は variant 配列として生成することも可能で、その場合には variant オブジェクトに異なる型を格納できます。SafeArray に格納できるデータ型は、コピーファイル **olesafea.cpy** でレベル 78 データ項目として定義されています。

SafeArray を生成するには、次のようなコードを作成します。

```
invoke OleSafeArray "new" using by value vType
                            by value dimensions
                          by reference saBounds(1)
                             returning aSafeArray
```

各パラメータの意味は次のとおりです。

| パラメータ | COBOL データ型 | 説明 |
|---|---|---|
| *dimensions* | pic x(4) comp-5 | SafeArray の次元数 |
| *saBounds* | SAFEARRAYBOUND occurs *n* | *n* は SafeArray の次元数 (*dimensions* の値)<br><br>SAFEARRAYBOUND 型のデータ項目には、llBounds と cElements という 2 つの要素が含ま |

れており、これらの要素で次元の下限と要素数を設定できます。

| | | |
|---|---|---|
| *vType* | PIC X(4) COMP-5 | SafeArray に格納するデータの型<br><br>格納できる COM データ型はコピーファイル **olesafea.cpy** で Level-78 として定義されています。 |

次の例は、3×2 の要素を持つ 2 次元の SafeArray を設定しています。

```
 program-id. tplolec.
 object section.
 class-control.
     OleSafeArray is class "olesafea"
     .
 working-storage section.
 copy "mfole.cpy".
 copy "olesafea.cpy".
 01 saBound              SAFEARRAYBOUND occurs 2.
 01 intSafeArray         object reference.
 01 varType              pic 9(4) comp-5.
 01 dimensions           pic x(4) comp-5.
 procedure division.
*>---データ型を 4 バイトの整数型に設定。
*>   VT-I4 は MFOLE.CPY で定義されて
*>   いる整数用の COM データ型
     move VT-I4 to varType
*>---配列を 2 次元として設定。
     move 2 to dimensions
*>---下限 0、要素数 3×2 の配列として
*>   定義。(下限は、次元の先頭要素の
*>   インデックス)
     move 3 to cElements of saBound(1)
                                        *>   cElements は次元のサイズ
                                        *>   を指定する SAFEARRAY 型
                                        *>   の副項目
                                        *>
     move 0 to llBound of saBound(1)
                                        *>   llBound は次元の下限
                                        *>   を指定する SAFEARRAY 型の
                                        *>   の副項目
                                        *>

     move 2 to cElements of saBound(2)
     move 0 to llBound of saBound(2)
```

```
      invoke ComSafeArray "new" using
                            by value varType
                            by value dimensions
                            by reference saBound(1)
                            returning intSafeArray
```

## SafeArray に関する情報の取得

SafeArray に対応する ComSafeArray のインスタンスを調べれば、次の情報が得られます。

- 次元数 (getDims メソッド)
- 各次元の下限と上限の境界 (getLBound メソッドと getUBound メソッド)

  これらのメソッドは Windows の状態コードも返します。存在しない次元のサイズの取得を試みた場合など、処理が失敗するとゼロ以外の状態コードが返されます。

- 配列に格納されているデータの型 (getVarType メソッド)
- 要素のサイズ (getElementSize メソッド)

すべてのデータは、PIC X(4) COMP-5 型で返されます。プログラムに外部から渡されたサイズ不明の SafeArray を処理するには、これらの情報が必要になります。

次のコード例では、SafeArray の次元数と、最初の次元のサイズを調べています。

```
 working-storage section.
 ...
 01 dimensions           pic x(4) comp-5.
 01 dimensionSize        pic x(4) comp-5.
 01 intSafeArray         object reference.
 01 lBound               pic x(4) comp-5.
 01 uBound               pic x(4) comp-5.
 01 hResult              pic x(4) comp-5.
 01 varType              pic x(4) comp-5.
 ...
 procedure division.
 ...
*>---配列の次元数を取得
       invoke intSafeArray "getDim" returning dimensions
       ...
       move 1 to dimensions
*>---hResult は SafeArray への照会で返される Windows の
*>    状態コード。0 は照会の成功、0 以外は失敗を示します。
*>    エラーコードは、コピーファイル MFOLE.CPY で
*>    Level-78 データ項目として定義されています。
      invoke intSafeArray "getLBound"
                                  using by value dimensions
```

```
                                        by reference lBound
                                returning hResult
    invoke intSafeArray "getUBound"
                                    using by value dimensions
                                        by reference uBound
                                returning hResult
*>---次元のサイズを計算
    subtract lBound from uBound
    add 1 to uBound giving dimensionSize
*>---配列に格納されているデータの型を取得
    invoke intSafeArray "getVarType" returning varType
    ...
```

## SafeArray の要素の読み取りと書き込み

OLESafeArray クラスには、格納されているデータの型に応じて、個々の要素の読み取りと書き込みを行うための各種メソッドがあります。

- 文字列
  - putCharArray メソッド と getCharArray メソッドは、Object COBOL の CharacterArray を使用して SafeArray との間で文字列をやり取りします。
  - putString メソッドと getString メソッドは、PIC X(*n*) データ項目を使用して SafeArray との間で文字列をやり取りします。getString で文字列を取り込む場合、取り込みに使用するデータ項目には十分な領域を割り当てておく必要があります。領域が不足するとデータが破壊されます。

    どちらのメソッドでも、文字列長は PIC X(4) COMP-5 データ項目で値渡しされます。

- COM オブジェクト
  - putOLEObject メソッドと getOLEObject メソッドは、SafeArray との間で COM オブジェクトをやり取りします。
- VT_BSTR 型と VT_DISPATCH 型の VARIANT
  - VT_BSTR ― getCharArrayFromVariant メソッドと putCharArrayAsVariant メソッドは CharacterArray として文字列にアクセスし、getStringFromVariant メソッドと putStringAsVariant メソッドは PIC X(*n*) としてアクセスします。
  - VT_DISPATCH ― putOLEObjectAsVariant メソッドと getOLEObjectFromVariant メソッドで COM オブジェクトにアクセスできます。

  先頭に get が付く各メソッドは、いずれもアクセス先の要素の型をチェックし、意図された型と異なる場合にエラーコードを返します。

- その他のデータ型

  getElement メソッドは、指定した要素のデータをメモリ領域にコピーします。このメソッドには ポインタを渡します (ポインタが、データを十分に格納できるサイズのデータ項

目を指していることを確認してください)。putElement メソッドは、引数として渡したポインタで指定される領域からデータをコピーします。getElementSize メソッドを使用すると、SafeArray の要素のサイズを取得できます。

これらのメソッドはいずれも、対象の要素を特定する索引テーブルと実際のデータを格納するデータ項目 (getElement メソッドと putElement メソッドの場合はポインタ) を引数として受け取り、エラーコードを返す点で互いに良く似ています。文字列にアクセスするメソッドでは、これらの引数に加え、文字列の長さを指定する PIC X(4) COMP-5 データ項目を値渡しする必要があります。

次のコード例は、putElement メソッドを使用して 3×2 の SafeArray に数値データを格納し、getCharArray メソッドを使用して VT_BSTR 型の SafeArray から文字列を取り込みます。

```
working-storage section.
copy "olesafea.cpy".
01 saBound              SAFEARRAYBOUND occurs 2.
01 intSafeArray         object reference.
01 strSafeArray         object reference.
01 iIndex               pic x(4) comp-5 occurs 2.
01 hIndex               pic x(4) comp-5.
01 vIndex               pic x(4) comp-5.
01 iValue               pic x(4) comp-5.
01 hResult              pic x(4) comp-5.
01 theData              POINTER.
...
procedure division.
    ...
    move 9 to iValue
    move 10 to strLength
    set theData to address of iValue *> putElement はアドレスポインタで
                          *> 指定される領域からデータを読み取ります。
    perform varying hIndex from 0 by 1 until hIndex = 3
        perform varying vIndex from 0 by 1 until vIndex = 2
            move hIndex to iIndex(1)
            move vIndex to iIndex(2)
            invoke intSafeArray "putElement"
                                          using iIndex(1)
                                     by value theData
                                    returning hResult
            subtract 1 from iValue
        end-perform
    end-perform
    ...
    perform varying hIndex from 0 by 1 until hIndex = 3
       perform varying vIndex from 0 by 1 until vIndex = 2
          move hIndex to iIndex(1)
```

```
            move vIndex to iIndex(2)
            invoke strSafeArray "getCharArray"
                                          using iIndex(1)
                                                aCharArray
                                        returning hResult
            invoke aCharArray "display"
        end-perform
        display " "
    end-perform
```

## SafeArray 内のデータへの直接的なアクセス

OLESafeArray のインスタンスでラップした Windows SafeArray のデータ構造体を格納したメモリ領域に直接アクセスすることも可能です。ただし、この操作を正しく行うには、SafeArray の内部構造に関する十分な知識が必要になります。

メモリ領域に直接アクセスするための手順は次のとおりです。

- SafeArray データをロックし、accessData メソッドを使用して、データ領域へのアドレスポインタを取得する。
- アドレスポインタを使用して、SafeArray を直接操作する。
- SafeArray の操作終了後、unAccessData メソッドを使用してロックを解除する。

詳細については、『OLESafeArray』のヘルプトピック内の『OLESafeArray Method accessData』および『OLESafeArray Method UnaccessData』を参照してください。

---



---

# 第 19 章 : Microsoft Component Services との連携

この章では、Net Express と Microsoft Component Services を連携させる方法について説明します。

## トランザクション処理

Microsoft Component Services は、大企業やインターネット、イントラネット環境向けに、スケーラブルで堅牢な高性能サーバアプリケーションを開発、ディプロイ (運用環境に配置)、および管理するためのトランザクション処理システムを提供する技術です。Microsoft Component Services は COM+ とも呼ばれます。 Component Services では、分散型アプリケーションを開発するためのアプリケーションプログラミングモデルが定義されています。また、開発したアプリケーションをディプロイし、管理するためのランタイム環境も提供されます。

Component Services を使用すれば、まとまった関数群を実行するコンポーネント単位にトランザクションを細分化できます。コンポーネントは、正常に終了したかどうかをコンポーネントサーバに通知します。トランザクションは、構成する全コンポーネントがすべて正常に終了するとコミットされ、正常に終了しなかったコンポーネントがあればロールバックされます。

この章では、Net Express を使用して、COBOL で COM+ コンポーネントを作成する方法を説明しています。Component Services のセットアップや管理の方法、および開発時に利用できるより高度な機能については、ここでは説明しません。詳細については、Microsoft COM+ の付属ドキュメントを参照してください。

## COM+ コンポーネントの作成

Microsoft Component Services で使用するコンポーネントは、常にプロセス内 COM サーバとしてコンパイルし、リンクする必要があります。この項では、コンポーネントの構造とビルド方法について説明します。 Net Express ではクラスウィザードを使用して COM+ を生成できます (クラスウィザードについては、Net Express の [**ヘルプ**] メニューで [**ヘルプトピック**] をクリックし、「**キーワード**」タブで「クラスウィザード」のトピックを選択して、表示されるページを参照してください)。COBOL で COM オートメーションサーバを作成する方法の詳細は、『*COM オブジェクトの作成*』の章を参照してください。

### COM+ コンポーネントの構造

COM+ で使用されるすべてのコンポーネントは、次に示す基本構造を持っています。

```
$set ooctrl(+P)
 class-id.
     component-name inherits from olebase.
```

```
object section.
class-control.
    Olebase is class "olebase"
    objectcontext is class "objectcontext".
object.

method-id. "method-name".

local-storage section.

01 Context              object reference.

 << メソッドローカルなすべてのデータ項目 >>

linkage section.

 << メソッドの全パラメータの定義 >>

procedure division using linkage-section-items.
*   オブジェクトのコンテキストを取得。Transaction Server
*   の制御下で実行していない場合、コンテキストは NULL に
*   設定されます。
     invoke objectcontext "GetObjectContext"
                                          returning Context

 << このコンポーネントに必要な処理を実行 >>

*   処理が成功した場合は SetComplete を呼び出し、
*   失敗した場合には SetAbort を呼び出します。
     if everything-ok
         if Context not = NULL
             invoke Context "setComplete"
         end-if
     else
         if Context NOT = NULL
             invoke context "setAbort"
         end-if
     end-if

*   コンポーネントの終了を確認
     if Context not = NULL
         invoke Context "finalize" returning context
     end-if
     exit method.
end method "method-name".
```

```
end object.
end class component-name.
```

| | |
|---|---|
| *component-name* | コンポーネントに使用される名前。COM のプログラム ID として使用され、この名前によって Component Services で識別されます。 |
| *method-name* | コンポーネントで呼び出すメソッドの名前。コンポーネントには複数のメソッドを含めることができます。 |

## objectcontext クラス

objectcontext クラスを使用すれば、Component Services で提供される機能を利用できます。objectcontext は、次のようにクラス制御節に含めます。

```
class-control.
    Combase is class "combase"
    objectcontext is class "objectcontext".
```

---

**注記**: *objectcontext* クラスは **objectcontext.dll** ファイルに含まれています。アプリケーションを配布するときには、このファイルを必ず COM+ コンポーネントにインクルードしてください。

---

## コンテキストオブジェクト

COM+ の各コンポーネントには、関連付けられたコンテキストオブジェクトがあります。.コンテキストオブジェクトとは、インスタンスの実行コンテキストを提供する拡張可能な COM+ オブジェクトです。インスタンスの実行コンテキストには、トランザクションやアクティビティ、およびセキュリテイのプロパティも含まれます。COM+ コンポーネントが生成されると、対応するコンテキストオブジェクトが自動的に生成されます。また、COM+ コンポーネントが解放されると、コンテキストオブジェクトも Component Services によって解放されます。

コンポーネントのコンテキストオブジェクトを取得するには、クラスメソッド *GetObjectContext* を使用します。このメソッドは、コンテキストオブジェクトのオブジェクト参照を返します。次に使用例を示します。

```
   01 Context           object reference.

   invoke objectcontext "GetObjectContext" returning Context
```

取得したコンテキストオブジェクトは、対応するコンポーネント専用であり、他のオブジェクトに渡すことはできません。

コンポーネントが Component Services の制御下で実行されていない場合 (COM+ パッケージにインポートされていない場合) には、オブジェクトコンテキストとして NULL が返されます。

## COM+ コンポーネントの終了

コンポーネントを終了する前に、対応するオブジェクトコンテキストを解放する必要があります。オブジェクトコンテキストを解放するには、finalize メソッドを使用します。次に使用例を示します。

```
invoke Context "finalize"
    returning context
```

この処理を実行すると、コンポーネントの参照がすべて解放されたときに、コンポーネントがメモリから正しくクリアされます。

## Objectcontext のメソッド

次に示す *objectcontext* クラスの各メソッドは、このコンポーネントのコンテキストを取得すると利用可能になります。これらのメソッドは、同じ名前の COM+ メソッドに直接マッピングされます。 各メソッドの詳しい機能については、Microsoft のドキュメントを参照してください。

### CreateInstance

使用例

```
01 NewObject        object reference.
invoke Context "CreateInstance"
    using z"Account"
    returning NewObject
```

現在のコンテキストを使用して、COM+ コンポーネントの新しいインスタンスを生成します。コンポーネントのプログラム ID が名前として渡されます。この名前は NULL で終端する必要があります。CreateInstance は、生成したインスタンスのオブジェクト参照を返します。新しいインスタンスは、現在のコンポーネントのコンテキストを使用します。

CreateInstance で生成したコンポーネントのインスタンスは、同じコンポーネントの new メソッドで生成したインスタンスと同じように使用できます。

### DisableCommit

使用例

```
invoke Context "DisableCommit"
```

コンポーネントの更新トランザクションが不整合で、現在の状態ではコミットできないことを宣言します。

### EnableCommit

使用例

```
    invoke Context "EnableCommit"
```

現在のコンポーネントの処理は必ずしも完了していないものの、その更新トランザクションに整合性があるため、現在の状態でコミットできることを宣言します。

### IsInTransaction

使用例

```
    01 ReturnValue        pic x(4) comp-5.
    invoke Context "IsInTransaction"
returning ReturnValue
```

トランザクションで現在のコンポーネントが実行されているときに 0 以外の値、現在のコンポーネントが実行されていない場合は 0 を返します。

### IsCallerInRole

使用例

```
    01 ReturnValue        pic x(4) comp-5.
    invoke Context "IsCallerInRole"
           using z"Managers" ReturnValue
```

コンポーネントの直接の呼び出し元が、それ自体またはグループの一部として、指定された役割 (Role) を持っているかどうかを示します。呼び出し元が指定された役割を持たない場合、0 が返されます。呼び出し元が役割を持っているか、セキュリティが有効化されていない場合は、0 以外の値が返されます。役割は NULL で終端する文字列として指定する必要があります。

### IsSecurityEnabled

使用例

```
    01 ReturnValue        pic x(4) comp-5.
    invoke Context "IsSecurityEnabled"
           returning ReturnValue
```

このコンポーネントに対してセキュリティが有効化されていない場合は 0、セキュリティが有効な場合は 0 以外の値を返します。

### receiveConstructorString

使用例

```
        method-id. "receiveConstructorString".
        local-storage section.
        linkage section.
        01 arrayObject      object reference.
        procedure division using by value arrayObject.
      *>  ここに文字列を処理するコードを追加
            exit method.
        end method "receiveConstructorString".
```

コンポーネントのプロパティで指定された初期化文字列を取得します。
ReceiveContructorString は CharacterArray オブジェクトのオブジェクト参照を返します。

**SetAbort**

使用例

```
    invoke Context "SetAbort"
```

オブジェクトが実行されているトランザクションの中断を宣言します。

**SetComplete**

使用例

```
    invoke Context "SetComplete"
```

現在のコンポーネントが処理を終了したことを宣言します。オブジェクトがトランザクションのスコープ内で実行されている場合は、そのブジェクトの更新トランザクションがコミット可能であることも示します。

### COM+ コンポーネントのビルド

Net Express ではクラスウィザードを使用して COM+ を生成できます (クラスウィザードについては、Net Express の [**ヘルプ**] メニューで [**ヘルプトピック**] をクリックし、「**キーワード**」タブで「クラスウィザード」のトピックを選択して、表示されるページを参照してください)。生成されたコンポーネントをビルドするには、『*COM コンポーネントの分散配置*』の章の『*プロセス内サーバの作成*』の項に記載されている手順に従います。

## COM+ コンポーネントの実行とデバッグ

COM+ コンポーネントは、COM+ 自体の制御下で動作するため、コンポーネントの動作環境やコンポーネントの動作中にログオンしているユーザを限定することはできません。

Net Express をインストールしていない PC でコンポーネントを使用するには、COBOL ランタイムファイルを含むディレクトリをシステムパスに登録する必要があります。このディレクトリを

パスに登録すれば、コンポーネントが必ず COBOL ランタイムファイルを検出できるようになります。

コンポーネントをデバッグする場合は、そのコンポーネントのコード内で、デバッグするメソッドの手続き部の先頭に次の行を追加してください。

```
call "CBL_DEBUGBREAK"
```

この行を追加すると、コンポーネントの実行時に Net Express デバッガが起動します。
Net Express がソースファイルとデバッグファイルを検出できるように、次の環境変数をシステム環境変数に追加してください。:

COBIDY — コンポーネントの **.idy** ファイルを含むディレクトリを設定します。

COBCPY — コンポーネントのソースファイルを含むディレクトリを設定します。

# COM+ コンポーネントのサンプル

Net Express には、COM+ で提供されている *Account* オブジェクトを COBOL で実装した COM+ コンポーネントがサンプルとして付属しています。このサンプルは、**base\demo\mtx** ディレクトリ内に **account.cbl** という名前で格納されています。

---



---

# 第 20 章 : .NET との連携

この章では、.NET 環境から呼び出し可能な COBOL COM オブジェクトを作成し、Visual Studio .NET 開発プラットフォームにインポートする過程について説明します。 Visual Studio .NET では、COM オートメーションサーバとして動作している COBOL COM にアクセスする .NET クライアントを、C# や VB.NET など任意の VS.NET 言語で作成できます。

## .NET に最適化した COM オブジェクトの作成

インターフェイスマッパーで作成する COM オブジェクトはすべてプロセス内 COM オブジェクトであり、.NET 環境でただちに使用可能です。

クラスウィザードとメソッドウィザードでプロセス内 COM オブジェクト (.dll) やプロセス外 COM オブジェクト (.exe) を作成するときには、次のオプションを必ず選択してください (どちらもデフォルトで選択されています)。

- 「タイプライブラリを作成する (.TLB)」チェックボックス
- 「マルチスレッドオプション」の「アパートメント」(プロセス内 COM オブジェクトを作成する場合のみ)

## Visual Studio .NET への COM オブジェクトのインポート

COM オブジェクトを Visual Studio .NET にインポートするには、そのオブジェクトの参照を追加します。 その結果、COM オブジェクトに含まれている COM インターフェイスの詳細情報が .NET 環境で利用できるようになります。 インポートした COM のインターフェイスが .NET のプロジェクトクラスライブラリに追加され、C# や Visual Basic (VB) でインスタンス化できるようになり、 編集画面に intellisense 情報 (メソッドの構文やパラメータ候補など) も表示されるようになります。

プロセス内またはプロセス外の COM オブジェクトを参照として追加するには、Visual Studio .NET でソリューションエクスプローラを使用します。 COM オブジェクトの .dll ファイルの絶対パスを指定するか、マシンに登録されている COM オブジェクトのリストから選択します。 この操作を行うと、インポートした COM オブジェクトのインスタンス化や、intellisense 情報の表示が可能になる前に、インターフェイス情報が .NET 開発環境で確認できるようになります。

COM オブジェクトのインターフェイス情報や GUID を変更した場合には、いったんオブジェクト参照を削除し、変更済みの COM オブジェクトの参照を追加しなおす必要があります。

詳細については、Microsoft Visual Studio .NET ドキュメントで、参照の追加に関するページを参照してください。

## .NET での COM の利用

.NET クライアントから COM オブジェクトにアクセスするには、そのオブジェクトが登録済みで、サーバで利用できる状態になっており、かつ COBOL アプリケーションが動作しているか、使用しているマシンで動作可能な状態になっている必要があります。

## プロセス内オブジェクト

インターフェイスマッパーのディプロイツールや、クラスウィザードとメソッドウィザードを使用してプロセス内 COM オブジェクトをディプロイした後、そのオブジェクトをサーバマシンのオペレーティングシステムに登録します。 オブジェクトをディプロイするたびに、登録を行ってください。オペレーティングシステムで COM オブジェクトの識別に使用されるグローバル一意識別子 (GUID) は、ディプロイするたびに再生成されます。 GUID は .dll ファイル内に格納され、オブジェクトの登録時にレジストリに登録されます。

以前に登録した COM オブジェクトを登録しなおすときには、必ず既存の登録情報をレジストリから削除してください。プロセス内 COM オブジェクトの登録および登録解除の詳細については、『インプロセス COM オブジェクトの登録方法』のヘルプトピックを参照してください。

## プロセス外オブジェクト

プロセス内 COM オブジェクトの場合と同様、プロセス外 COM オブジェクトもディプロイのたびに登録する必要があります。プロセス外 COM オブジェクトの登録および登録解除の詳細については、『実行可能ファイルを使用したアウトオブプロセス COM オブジェクトの登録方法』のヘルプトピックを参照してください。

## レガシーアプリケーションの利用

COM オブジェクトから呼び出すレガシー COM アプリケーションは、COM オブジェクトで検出できるようにしておく必要があります。レガシーアプリケーションの絶対パスを環境変数 COBDIR に登録すれば、容易かつ確実に、ランタイムシステムにレガシーアプリケーションを検出させることができます。 詳細については、『概要 - アプリケーションの実行』を参照してください。

# 第 21 章 : IBM WebSphere との連携

IBM WebSphere Extended Edition CICS の実行環境と Net Express の開発環境は、互いに密接に連携させ、組み合わせて使用することができます。 この章では、Net Express による WebSphere への対応と、その利用方法を説明します。

『*WebSphere アプリケーションの作成*』の項は、WebSphere アプリケーションのプロジェクトを正しく作成し、コンパイル、ビルド、およびディプロイを行うために必要なすべての情報を網羅しています。 その他の項は、Net Express での WebSphere 対応に重点を置いています。

WebSphere のドキュメントも参照してください。

## WebSphere アプリケーションの作成

WebSphere アプリケーションの作成では、標準的なアプリケーションの作成手順に加え、次の作業を行う必要があります。

- コンパイルする前に、WebSphere COBOL ソース (**.ccp**) を CICSTCL で変換して標準 COBOL ソース (**.cbl**) を生成する。
- コンパイル時には、必ず CALL-RECOVERY 指令と DATA-CONTEXT 指令を使用する。
- デバッグ用には中間コードまたはダイナミックリンクライブラリ (DLL) として、実行用には DLL としてビルドする。 DLL にリンクする際には、リンクライブラリファイル **cicsprcbmfnt.lib** をインクルードする。
- WebSphere で使用するには、ディプロイする前に DLL の名前を変更する。

以降の説明に従って WebSphere プロジェクトを作成すれば、これらの作業はいずれも Net Express によって自動的に実行されます。

WebSphere の COBOL ソースの拡張子は **.ccp** です。 Net Express のプロジェクトは、このソースファイルを使って作成します。 ビルドすると **.cbl** ファイルが生成されますが、このファイルは編集しないでください。**.cbl** ファイルはプロジェクトをリビルドするたびに再生成されるため、編集した内容も失われてしまいます。 **.cbl** ファイルは、誤って編集されないように、生成された時点で読み取り専用になっています。

プロジェクトの作成に着手する前に、Net Express が cicstcl.exe を検出して実行できるように、必ずシステムの PATH に **opt\cics\bin** フォルダを登録してください。**cicstcl.exe** によって **.ccp** ファイルが **.cbl** ファイルに変換されます。

### WebSphere プロジェクトの作成

Net Express で WebSphere プロジェクトを作成する手順は次のとおりです。

1. Net Express で [**ファイル** > **新規作成** > **プロジェクト**] をクリックします。 ウィザードを最後まで実行し、新しい空のプロジェクトを作成します。
2. 「プロジェクト」ウィンドウ左側のビルドペイン内を右クリックし、[**プロジェクトへファイルを追加**] をクリックします。
3. ダイアログボックスの「**ファイルの種類**」ドロップダウンメニューで「**IBM WSEE/TX Series Program (.ccp)**」を選択します。
4. アプリケーションに必要な WebSphere のソースファイル (拡張子 **.ccp** のファイル) をすべて選択し、[**追加**] をクリックします。

   Net Express によって各ファイルのビルド情報が設定されます。基本的なコンパイラ指令が追加され、各 **.ccp** ファイルから DLL が生成されます。

   ソースファイルを追加するときには、この手順を必要に応じて繰り返します。 CICS 以外の **.cbl** ファイルを追加する場合は、それらの各プログラムのビルド設定に DATA-CONTEXT 指令と CALL-RECOVERY 指令を明示的に追加してください (ステップ 9、10、11 を参照)。

   DLL のかわりに中間ファイルをデバッグに使用する場合は、後述する『*中間コードの作成*』を参照してください。

5. [**プロジェクト** > **プロパティ** > **IDE**] をクリックし、環境変数名 (COBCPY) を「変数」フィールドに、CICS のインクルードフォルダ (通常は **drive\opt\cics\include**) と **;%cobcpy%** を連結した文字列を「値」フィールドにそれぞれ入力し、[**設定**] をクリックします。
6. 続いて、環境変数名 LIB を「変数」フィールドに、CICS の LIB フォルダ (通常は **drive\opt\cics\lib**) に **;%lib%** を連結した文字列を「値」フィールドにそれぞれ入力し、[**設定**] をクリックします。
7. ダイアログボックスを閉じます。
8. [**プロジェクト** > **すべてをリビルド**] をクリックしてアプリケーションをビルドします。構文エラーが発生した場合は、ビルドが成功するまで修正を加えます。 次の警告メッセージが表示される場合もあります。

   ```
   Not supported for target platform.
   ```

   このメッセージは無視してかまいません。

アプリケーションをデバッグするには、デバッグセッションを開始するプログラムのビルド設定に、コンパイラ指令 INITCALL(CBL_DEBUGBREAK) を追加する必要があります。

9. ビルドペイン内で対象の **.cbl** ファイルを右クリックし、[**ビルド設定**] をクリックします。
10. 「コンパイル」タブをクリックし、「指令」ペイン内のセミコロン (;) の直前に INITCALL(CBL_DEBUGBREAK) と入力します。
11. [**OK**] をクリックします。

WebSphere で使用するには、実行可能ファイルの拡張子を **.cbmfnt** に変更する必要があります。 次の手順に従って実行可能ファイルの名前を変更し、適切なフォルダにコピーして、ディプロイできる状態にします。

12. [**プロジェクト > ディプロイ**] をクリックし、「**ディプロイする選択されたファイル**」ダイアログボックスを開きます。
13. [**追加**] をクリックして「**ディプロイするファイルの選択**」ダイアログボックスを開きます。
14. 「**ディプロイするファイル**」フィールドに、ディプロイする **.dll** ファイルの名前を SMI コロン (;) で区切って入力するか、[**参照**] をクリックしてディプロイするファイルを含むフォルダに移動し、それらのファイルをすべて選択します (ステップ 8 で説明した要領でプロジェクトをビルドするまで、[参照] をクリックしてもファイルは表示されません)。
15. 「**場所**」フィールドにコピー先フォルダの名前を入力します (すでに存在するフォルダの名前を入力します)。
16. ダイアログボックスを閉じます。
17. [**プロジェクト > すべてをリビルド**] をクリックします。 すべてのファイルがビルドされてコピーされ、実行可能ファイルの名前が変更されます。 以降のビルドでは、変更を加えたファイルがコピーされ、名前が変更されます。

これでプロジェクトをディプロイできる状態になります。 ソースファイル (**.ccp** ファイル) の編集、デバッグ用のビルド、およびアプリケーションのデバッグは通常の方法で行います。

## ソースプールからのファイルの追加

WebSphere ソースファイル (**.ccp** ファイル) は、ソースプール (「プロジェクト」ウィンドウ右側のペイン) から随時追加できます。 ただし、ビルドペインにドラッグしても、実行されるのは **.ccp** ファイルから **.cbl** ファイルへの変換だけであり、 DLL ファイルを生成するためのパッケージ化は明示的に実行する必要があります。次に手順を示します。

1. ソースペインで、右クリックして [**ソースプールへファイルを追加**] を選択します。
2. 追加する **.ccp** ファイルを選択して [**追加**] をクリックします。
3. ファイルをビルドペインにドラッグします。
4. 「**コンパイルのファイルタイプを選択**」ダイアログボックスで [**OK**] をクリックします。
5. 生成された **.cbl** ファイル (または選択した複数の **.cbl** ファイル) をビルドペインで右クリックし、[**選択されたファイルをパッケージ化**] をクリックして、さらに [**ダイナミックリンクライブラリ (DLL)**] をクリックします。
6. 生成された **.dll** ファイルを右クリックし、[**ビルド設定**] をクリックします。
7. 「**リンク**」タブをクリックし、「**カテゴリ**」選択ボックスの矢印をクリックして「**高度**」をクリックします。
8. 「**これらの LIB とリンクする**」フィールドに **cicsprcbmfnt.lib** と入力します。
9. [**閉じる**] をクリックします。

## 中間ファイルの生成

デバッグには DLL のかわりに中間ファイル (**.int** ファイル) を使用できます。 中間ファイルを生成する手順は次のとおりです。

1. ビルドペインにファイルを追加する前に、すべての **.ccp** ファイルをソースプールに追加します。追加した各ファイルが変換され、**.cbl** ファイルがソースプールに追加されます。
2. DLL としてパッケージ化する場合とは異なり、ここでソースプール内の全 **.cbl** ファイルを選択し、ビルドペインにドラッグします。
3. 「**コンパイルのファイルタイプを選択**」ダイアログボックスで [**OK**] をクリックします。
4. **.ccp** ファイルごとに 2 つのビルドツリーが形成されます。最上位に **.cbl** ファイルが位置するツリーで、その **.cbl** ファイルを右クリックして [**ビルドタイプから削除**] をクリックします。
5. [**OK**] をクリックして「ビルド構造から削除」ダイアログボックスを閉じます。
6. ビルドペイン内の **.cbl** ファイルごとに、右クリックして [**ビルド設定**] をクリックします。
7. 「コンパイル」タブをクリックし、「指令」ペイン内のセミコロン (;) の直前に CALL-RECOVERY 指令と DATA-CONTEXT 指令を入力します。
8. [**OK**] をクリックします。

### WebSphere アプリケーションのデバッグ

WebSphere アプリケーションのプロジェクトを作成すると、続いてデバッグを開始します。デバッグの手順は次のとおりです。

1. Net Express で [**アニメート > 設定**] をクリックします。パネル右側の「その他」の下にある「**アニメーションのアタッチメントを待つ**」チェックボックスを選択します。
2. 「**アニメーションの開始位置**」フィールドの内容を消去して [**OK**] をクリックします。
3. [**アニメート > アニメート開始**] をクリックしてデバッグを開始します。
4. 「**アニメーションの起動**」ダイアログボックスで、「**実行可能ファイルまたはプログラム名**」フィールドが空欄になっていることを確認し、[**OK**] をクリックします。
5. WebSphere CICS システムで、通常の方法でトランザクションを実行します。トランザクションで INITCALL(CBL_DEBUGBREAK) 指令付きのプログラムが実行されるときに、メッセージが表示されます。
6. [**はい**] をクリックします。Net Express でデバッガが有効になり、アプリケーションのデバッグを開始できます

   ([**はい**] をクリックしたときに Net Express が動作していないと、プロセスがハングアップしたような状態になることがあります)。

# WebSphere への対応

この項では、Net Express による WebSphere アプリケーションへの対応について、ある程度まで詳しく説明します。ここに記載されている情報を読み、理解することは、WebSphere アプリケーションの作成に必須ではありません。

Net Express による WebSphere 対応は、主に次の処理を通じて実現されます。

- 呼び出し先およびリンク先プログラム (再帰も含む) の COBOL 作業場所 (Working-Storage) を複数のレベルにわたって管理する。

- COBOL サブルーチンを一貫した、予測可能なかたちで動的にロードする。
- WebSphere と COBOL ランタイムシステム間でやり取りされるメッセージの処理機能を強化する。

## WebSphere COBOL の記憶域管理

DATA-CONTEXT コンパイラ指令付きでコンパイルされ、サブルーチンにリンクされる (またはサブルーチンを呼び出す) プログラムは、WebSphere との組み合わせで、COBOL の作業場所 (Working-Storage) のインスタンスが LINK の各レベルで一意になるように動作します。WebSphere アプリケーションの通常的なコンパイルでは DATA-CONTEXT 指令が自動的に設定され、開発者側で明示的に実施すべき操作は特にありません。ただし、トランザクションの一部として CICS 以外の COBOL サブプログラムを呼び出す場合には、そのプログラムの作業場所が正しく管理されるように、DATA-CONTEXT 指令付きでコンパイルする必要があります。

## WebSphere COBOL の実行可能ファイルのロード

CICS 変換された標準的なプログラムは、関連付けられた PPT エントリ (PD スタンザ) で指定されたフォルダか、WebSphere 標準のセマンティックスに従って検出されたフォルダから、トランザクションコード (または LINK や XCTL) で起動します。LINK と XCTL を使用する環境では、COBOL RTS の通常的なロード規則はいっさい使用されません。

## COBOL サブプログラムの動的なロード

COBOL サブルーチンの動的な呼び出し (最終的に新しい実行可能ファイルがロードされる) が CICS プラットフォームで実行されると、最初に PPT エントリ (PD スタンザ) の解決が試みられます。PPT で指定される物理モジュールには通常、PPT 名に一致するエントリポイントが含まれています。一致する PPT エントリが存在しない場合のみ、COBOL RTS は通常の規則に従ってモジュールのロードを試みます。

**var\cics_regions*region-name*\environment** に環境変数のエントリを追加すれば、環境変数の設定を WebSphere のリージョンごとに変えることができます。WebSphere リージョンのアプリケーションサーバのカレントフォルダは通常、**var\cics_regions\region--name\dumps\dir1** です。

---

**注記**：WebSphere のオンライン環境で検出できる COBOL プログラムは、変換前の状態で **.cbmfnt**、**.gnt**、**.int** のいずれかの拡張子を持つファイルに限定されます。

---

## COBOL ランタイムシステムのエラーメッセージ

範囲外のインデックスを指定したり、ゼロによる除算を試みたり、あるいは CALL-RECOVERY 指令を指定しない状態でモジュールの検出に失敗した場合など、COBOL ランタイムシステムエラーが発生したときには、次に示す新しい Abend コードが表示されます。 COBOL RTS エラーメッセージのテキストは、console log に出力されます。

A583 - Any COBOL RTS error.

# SFS との EXTFH インターフェイス

WebSphere には SFS との EXTFH インターフェイスが含まれており、WebSphere のオンライン環境で作成し、更新したファイルと同一のファイルに対して COBOL アプリケーションをバッチ実行する場合には、このインターフェイスを使用する必要があります。

詳細については、WebSphere の ReadMe を参照してください。

---



---

# 第 22 章 : COM での XML の使用

この章では、Net Express の XML 対応について説明します。XML の概要、XML スキーマ、および Net Express COBOL における XML の位置付けを示します。

## Net Express と XML

Net Express では、XML によるデータ交換の利点を、COBOL の既存および新規のアプリケーションで容易に活用することができます。

Net Express 自体の COBOL 構文が XML 拡張構文によって一段と拡張され、XML ドキュメントを操作するために必要なあらゆる構文が使用可能です。XML 拡張構文は、Net Express の XML プリプロセッサによって処理されます。詳細については、『XML 拡張構文』の章を参照してください)。

cbl2xml コマンド行ユーティリティと CBL2XML ウィザードによって、次の処理を行うことができます。

- XML 構文を含む COBOL プログラムの生成
- 既存の COBOL レコードに基づく XML スキーマの生成
- COBOL レコードのスキーマへのマッピング
- XML スキーマと XML ドキュメントの検証

cbl2xml の詳細については、『XML 構文を含む COBOL 構造の生成』の章を参照してください。

XML 対応の COBOL アプリケーションの作成手順を次に示します。この手順は、cbl2xml を利用するか否かに関係なく共通です。

- COBOL コードへの XML 拡張構文の追加
- XML 構文を含む COBOL データレコードの生成
- Net Express でのコードのコンパイル (XML プリプロセッサによる前処理を含む)
- アプリケーションのテストとディプロイ

XML 技術の概要については、後述する『XML の概要』を参照してください。

### 基本的な実装

XML データコードはレコードに基づいており、大部分のコードが COBOL のデータレコードとして容易に表現できます。次に例を示します。

まず、XML のストリームの一例を示します。

```
<?xml version="1.0"?>
<group elementNumber=1235 >
```

```
   <elementAlpha>Alpha value</elementAlpha>
</group>
```

この XML ストリームは、XML 拡張構文を含む次の COBOL コードに直接格納できます。

```
01  xmls-group                    identified by "group".
    05  xmls-Number     pic 9(18) identified by "elementNumber" is attribute.
    05  xmls-Alpha      pic x(80) identified by "elementAlpha".
```

**例 22-1 XML データと COBOL レコードへの格納**

これはシンプルな例に過ぎません。Net Express では可変タグの指定や、不規則でネストしたオカレンス、異なるデータ型の混在、オプションの要素といった、より複雑なさまざまな構文にも対応できます。

COBOL アプリケーション内で XML レコードを表現すれば、そのレコードをアプリケーション内で送受信できます。レコードのやり取りには、OPEN、READ、WRITE、CLOSE など、標準的な入出力動詞を使用します。変換やコード化、および必要なすべての入力処理は、ランタイムシステムで自動的に実行されます。たとえば、例 22-2 に示した XML ストリームは、次の短いコードだけで生成し、ファイル (out-xml) に格納できます。

```
     select xml-stream assign "out.xml"
           organization  is xml   document-type is "group"
                                  file status is xml-bookdb-status.
     xd xml-stream.
     01  xmls-group                identified by "group".
         05  xmls-Number pic 9(18) identified by "elementNumber"
                                   is attribute.
         05  xmls-Alpha  pic x(80) identified by "elementAlpha".
working-storage section.
     01  xml-bookdb-status         pic s9(9) comp.

     move 1235                     to xmls-Number
     move "Alpha value"            to xmls-Alpha
     open output xml-stream
     write xmls-group
     close xml-stream
stop run.
```

# XML PARSE

Net Express は、IBM による XML 実装、XML PARSE に完全対応しています。

XML PARSE 文は XML の処理機構を提供します。すなわち、XML PARSE 文を使用した XML ドキュメントのデータは、システムの XML パーサによって構成要素単位に分割され、ユ

ーザルーチンで使用できるようになります。細分化された各要素には、その要素の特徴を示すタグ (イベント) が付加されます。

### 特殊レジスタ

XML パーサとユーザルーチン間の通信には、次の 4 つの特殊レジスタが使用されます。

- XML-CODE ― XML 解析の状態を特定するレジスタ
- XML-EVENT ― 各 XML イベントの名前 (START-OF-DOCUMENT など) を受信するレジスタ
- XML-TEXT ― 英数字で構成されるドキュメントから XML ドキュメント要素を受信するレジスタ
- XML-NTEXT ― ローカライズされたドキュメントから XML ドキュメント要素を受信するレジスタ

特殊レジスタの詳細については、『言語リファレンス』の『COBOL 言語の概念』の章にある『特殊レジスタ』の項を参照してください。

### ユーザルーチン

プログラムには、予想される種類のドキュメント要素 (イベント) を処理するユーザルーチン (通常は EVALUATE 文を含むルーチン) を含める必要があります。パーサはドキュメント全体の処理が完了するか、重大なエラーが発生するまで、連続的に要素情報を送信します。不正な形式の XML が検出されると、エラーコードが特殊レジスタ XML-CODE に格納され、呼び出し元の文に制御が戻ります。重大エラー以外のエラーであれば、ユーザルーチン側で XML-CODE をゼロにリセットし、実行を継続することができます。また、通常のイベントで XML-CODE を -1 に設定することも可能です。-1 に設定するとパーサが停止します (例外イベントは生成されません)。

詳細については、ヘルプトピックの『XML PARSE 文』を参照してください。

## XML の概要

XML (eXtensible Markup Language) は、Web 上でデータを交換する手段として、急速に普及しています。XML の言語仕様は DTD (Document Type Definition) や XML スキーマで定義されたルールセットに準拠しており、これらのルールによって XML ドキュメントの構造が決定されます。XML スキーマは DTD に比べ、XML ドキュメントの構造を定義する手段として、いっそう先進的で強力、かつ支配的な技術です。Net Express では XML スキーマが広範にわたってサポートされており、COBOL アプリケーションによる XML ドキュメントの入出力が可能になっています。

XML の仕様は W3C によって策定されており、同団体の Web サイトが唯一のソースです。ここでは、Net Express でデータ交換を行うために必要な XML の部分だけに焦点を当てます。

### XML スキーマ

XML スキーマ (通常は .xsd ファイル) の一部は要素定義から構成されており、スキーマを参照する XML ドキュメントの構造と内容が、これらの定義によって決定されます。また、XML 要素には属性を関連付けることが可能であり、これらの属性によって要素に含まれるデータがいっそう詳しく定義されます。

次に、シンプルな XML スキーマの一例を示します。

```
<?xml version="1.0" encoding="utf-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
   <element name="employee">
      <complexType>
         <sequence>
            <element name="emp-name">
               <simpleType>
                  <restriction base="string">
                     <length value="12"/>
                  </restriction>
               </simpleType>
            </element>
            <element name="emp-number">
               <simpleType>
                  <restriction base="string">
                     <length value="8"/>
                  </restriction>
               </simpleType>
            </element>
         </sequence>
      </complexType>
   </element>
</schema>
```

**例 22-3 XML スキーマの一例**

このスキーマは employee という要素と、その 2 つの子要素 (emp-name、emp-number) を定義しています。それぞれの子要素には、さらに 2 つの子要素 (restriction、length) が含まれています。Net Express の XML 拡張構文では、これらの要素の base 属性と value 属性は、いずれも COBOL レコードの PIC 句に変換されます。

## XML ドキュメント

XML ドキュメント (.xml ファイル) は、スキーマで宣言されている要素群から構成され、さらにデータを含んでいます。Net Express では、XML 拡張構文を使用することによって、COBOL プログラム内で XML インスタンスドキュメントの入力や出力を行うことができます。

# 第 23 章 : XML 拡張構文

この章では、COBOL プログラムによる XML の入出力を可能にする XML 拡張構文について説明します。

## 概要

Net Express では、COBOL のデータ構造と、XML スキーマ内の要素定義の関連付けがサポートされており、 COBOL の XML 拡張構文を使用して、COBOL の定義内に XML 構造への参照を直接記述することが可能です。 XML 構造は、XML 形式での情報の入出力に使用されます。

## COBOL で XML を操作するための構文

Net Express では、COBOL プログラム内で XML 操作専用の構文を使用できます。 以降の段落と句の説明は、いずれも COBOL プログラム内で XML の読み取りや書き込みに使用される場合を前提としています。

### SELECT 段落

SELECT 段落は、ASSIGN 句を使用して XML ストリームにファイル名を割り当てることができるように拡張されています。 ASSIGN 句によって XML ストリームの経路と、ストリームに適用するスキーマを指定します。 このストリームの実際の形式は、ファイル節の XML 記述子 (XD) 段落内のレコード定義によって決定されます。

XML 構文 — SELECT 句

### XML 記述子 (XD) 段落

XML 形式のファイルごとに、操作する XML ストリームの形式を定義するレコード定義が必要です。 これらのレコード定義は、原始プログラムのファイル節内の XD 段落に含まれます。

XML 構文 — XD ファイル段落

### IDENTIFIED BY 句と IS ATTRIBUTE 句

XML のスキーマまたはドキュメント内で定義されている XML の要素 (タグ) や属性にレコードをマッピングする COBOL データレコードの宣言に使用します。

COBOL データレコードと XML タグを IDENTIFIED BY で対応付け、さらに IS ATTRIBUTE を追加して XML タグの属性を対応付けます。次に一例を示します。

次の内容を持つ XML ドキュメントを想定します。

```
<company_name type="accounting">Webber Lynch</company_name>
```

この場合、対応する COBOL レコードは、IDENTIFIED BY 句と IS ATTRIBUTE 句を使用して次のように記述できます。

```
    05  company identified by "company_name".
        10  company-name-value   pic X(30).
        10  company-type    pic x(10) identified by "type" is attribute.
```

### XML 構文 ― IDENTIFIED BY 句

XML 拡張構文を使用するプログラムでは、既知のタグ名だけでなく、タグ名と属性名の変数も操作できます。COBOL データ名をタグ名や属性名で置き換えることによって、タグや属性を指定します。その結果、出力用に開いた XML ファイルに、XML ドキュメントを動的、かつ任意の複雑の度合いで出力できます。また、入力用に開いたファイルから、そのような複雑なドキュメントを読み取ることができます。たとえば、次に示すプログラムはルートタグと単一階層のネストタグを含む XML ストリームを、ネストタグの数や、タグあたりの属性数に関係なく読み取ります。

```
0010 $set preprocess(prexml) o(foo.pp) warn endp
0020 $set sourceformat(free) case
0030
0040 select doc                 assign address of mybuf
0050                            organization is xml
0060                            document-type is external doc-type
0070                            file status is doc-status.
0080 xd doc.
0090 01  root-tag            identified by root-tag-name.
0100     10  root-tag-name        pic x(80).
0110     10  root-tag-val         pic x(80).
0120     10  root-tag-attr        identified by root-tag-attr-name
0130                              is attribute.
0140         15  root-tag-attr-name pic x(80).
0150         15  root-tag-attr-val  pic x(80).
0160
0170     10  sub-tag              identified by sub-tag-name.
0180         15  sub-tag-name          pic x(80).
0190         15  sub-tag-val           pic x(80).
0200         15  sub-tag-attr          identified by sub-tag-attr-name
0210                                   is attribute.
0220             20  sub-tag-attr-val  pic x(80).
0230             20  sub-tag-attr-name pic x(80).
0240
0250 working-storage section.
0260 01  doc-type            pic x(80).
0270 01  doc-status          pic s9(9) comp.
```

```
0280 01  mybuf.
0290     10  pic x(300) value
0300     '<library location="here" '
0310     &         'xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" '
0320     &         'xsi:schemaLocation="library.xsd">'
0330     &    '<book published="yes" goodreading="ok">'
0340     &       'booktext'
0350     &    '</book>'
0360     &    'is a place'
0370     & '</library>'.
0380
0390 procedure division.
0400     open input doc
0410     read doc
0420     display "Document type is: " doc-type
0430     display "Tag name: " root-tag-name
0440     display "Tag value:" root-tag-val
0450     start doc key root-tag-attr
0460     *>
0470     *> ループしてすべての属性の名前を出力
0480     *>
0490
0500     perform until exit
0510        read doc next key root-tag-attr
0520        if  doc-status not = 0
0530            exit perform
0540        end-if
0550        display "Attribute name : " root-tag-attr-name
0560        display "Attribute value: " root-tag-attr-val
0570     end-perform
0580
0590     *> ループしてすべてのサブタグを出力
0600     start doc key sub-tag
0610     perform until exit
0620        read doc  next key sub-tag *> インデックスのデフォルト値は 1
0630        if  doc-status not = 0
0640            exit perform
0650        end-if
0660        display " Sub tag name: " sub-tag-name
0670        display " Sub tag value:" sub-tag-val
0680
0690        start doc key sub-tag-attr index is 1
0700        perform until exit
0710            read doc next key sub-tag-attr
0720            if  doc-status not = 0
0730                exit perform
```

```
0740            end-if
0750            display " Sub tag attribute name : " sub-tag-attr-name
0760            display " Sub tag attribute value: " sub-tag-attr-val
0770        end-perform
0780    end-perform
0790
0800    close doc
0810
0820    stop run.
```

0040～0070 行

```
 select doc                 assign address of mybuf
                            organization is xml
                            document-type is external doc-type
                            file status is doc-status.
```

XML 入力のソースとしてファイル名とバッファ (mybuf) を割り当て、ファイル形式 (XML)、XML スキーマ名を指定する変数のデータ名、およびファイル状態を示すデータ名を指定しています。

0080～0230 行

```
0080 xd doc.
01  root-tag            identified by root-tag-name.
10  root-tag-name        pic x(80).
    10  root-tag-val         pic x(80).
    10  root-tag-attr        identified by root-tag-attr-name
                              is attribute.
        15  root-tag-attr-name pic x(80).
        15  root-tag-attr-val  pic x(80).

    10  sub-tag              identified by sub-tag-name.
        15  sub-tag-name          pic x(80).
        15  sub-tag-val           pic x(80).
        15  sub-tag-attr          identified by sub-tag-attr-name
                                   is attribute.
            20  sub-tag-attr-val  pic x(80).
            20  sub-tag-attr-name pic x(80).
```

バッファ領域のレコード構造をセットアップしています。

0280～0370 行

```
01  mybuf.
10  pic x(300) value
```

```
    '<library location="here" '
    &         'xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" '
    &         'xsi:schemaLocation="library.xsd">'
    &    '<book published="yes" goodreading="ok">'
    &       'booktext'
    &    '</book>'
    &    'is a place'
    & '</library>'.
```

静的バッファに格納する XML ストリームを指定しています。このプログラムでは XML ストリームをコード内で指定していますが、標準入力を使用して XML をバッファ (mybuf) に格納することもできます。

0400～0440 行

```
open input doc
    read doc
    display "Document type is: " doc-type
    display "Tag name: " root-tag-name
    display "Tag value:" root-tag-val
```

バッファ領域を開いて XML レコードを読み取り、ドキュメントタイプ、ルートタグ名、およびルートタグの値を表示します。

0450 行

```
start doc key root-tag-attr
```

バッファ内で、ルートタグの最初の属性が開始する位置を設定します。

0500～0570 行

```
perform until exit
       read doc next key root-tag-attr
       if  doc-status not = 0
           exit perform
       end-if
       display "Attribute name : " root-tag-attr-name
       display "Attribute value: " root-tag-attr-val
    end-perform
```

ルートタグ内の属性情報をループして読み取り、属性の名前と値を順次表示します。

0600 行目

```
0600     start doc key sub-tag
```

バッファ内で、最初のネストタグ (ルート以外のタグ) の開始位置を設定します。

0610～0780 行

```
perform until exit
       read doc  next key sub-tag *> インデックスのデフォルト値は 1
       if  doc-status not = 0
           exit perform
       end-if
       display " Sub tag name: " sub-tag-name
       display " Sub tag value:" sub-tag-val

       start doc key sub-tag-attr index is 1
       perform until exit
           read doc next key sub-tag-attr
           if  doc-status not = 0
               exit perform
           end-if
           display " Sub tag attribute name : " sub-tag-attr-name
           display " Sub tag attribute value: " sub-tag-attr-val
       end-perform
   end-perform
```

ルート以外のすべてのタグにわたってループし、XML タグの名前と値、および属性の名前と値を順次表示します。

**例 23-1 IDENTIFIED BY 句と IS ATTRIBUTE 句の使用方法**

## PROCESSING-INSTRUCTION 句

PROCESSING-INSTRUCTION 句は、タグに対応するデータ内に、XML 処理命令を直接埋め込むためだけに使用します。次に使用例を示します。

```
    01  my-tag    identified by "my_tag".
        05  my-tag-data1    pic x(80).
        05  my-tag-pi       pic x(80) IS PROCESSING-INSTRUCTION.
        05  my-tag-data2    pic x(80).
    ...
        move 'data' to my-tag-data1 my-tag-data2
        move 'somepi' to my-tag-pi
        write my-tag
```

このコードは、次の XML コードを生成します。

```
    <my_tag>data<?somepi?>data</my_tag>
```

## COUNT IN 句

『IDENTIFIED BY 句と IS ATTRIBUTE 句』の項で示したように、出現回数が特定されない既知のタグや不明なタグにも、XML 用に拡張された各種の動詞を通じて対処することが可能です。一方、それほど複雑ではなく、出現回数があらかじめ分かっている場合には、非常にシンプルな拡張構文で直接割り当てを行うことができます。

出現回数が分かっているタグや属性には、OCCURS 句や COUNT IN 句を使用します。OCCURS 句では、タグや属性の出現回数の許容上限を指定します。COUNT IN 句では、受信回数を特定したり、送信回数を指定することができます。

COUNT IN 句の使用例を次に示します。

```
0010  $set preprocess(prexml.int) o(foo.pp) warn endp
0020  $set sourceformat(free) case
0030
0040  select library-file
0050        assign address of mybuf
0060        organization  is xml
0070        document-type is "library.xsd"
0080        file status is library-status.
0090
0100  data division.
0110
0120  xd library-file.
0130  01  library                     identified by "library".
0140    05  book                     identified by "book"
0150                                    occurs 10 times
0160                                    count in library-book-count.
0170        10  book-title  pic x(80) identified by "title".
0180        10  book-author pic x(80) identified by "author".
0190        10  book-toc             identified by "toc".
0200            15  book-toc-section  occurs 20 times
0210                                   count in book-section-count
0220                                   identified by "section"
0230                                   pic x(20).
0240
0250  working-storage section.
0260  01  library-status  pic s999.
0270  01  mybuf.
0280      10  pic x(150) value
0290     '<library>'
0300     & '<book><title>book1</title>'
0310     &'       <toc><section>b1section1</section>'
```

```
0320     & '           <section>b1section2</section>'
0330     & '</toc></book>'.
0340      10 pic x(150) value
0350       '<book><title>book2</title>'
0360     & '      <toc><section>b2section1</section>'
0370     & '           <section>b2section2</section>'
0380     & '</toc></book>'
0390     & '</library>'.
0400
0410  open input library-file
0420  read library-file
0430  perform until library-book-count = 0
0440    display "Book title         = '" book-title(library-book-count) "'"
0450    display "Number of Sects = " book-section-count(library-book-count)
0460    subtract 1 from library-book-count
0470  end-perform
0480  close library-file
0490  stop run.
```

0040〜0080 行

```
select library-file
        assign address of mybuf
        organization  is xml
        document-type is "library.xsd"
        file status is library-status.
```

XML 入力のソースとしてファイル名とバッファ (mybuf) を割り当て、ファイル形式 (XML)、XML スキーマ名を指定するファイル名 (library.xsd)、およびファイル状態を示すデータ名を指定しています。

0120〜0230 行

```
  xd library-file.
  01  library                    identified by "library".
    05  book                     identified by "book"
                                   occurs 10 times
                                   count in library-book-count.
        10  book-title  pic x(80) identified by "title".
        10  book-author pic x(80) identified by "author".
        10  book-toc              identified by "toc".
            15  book-toc-section  occurs 20 times
                                   count in book-section-count
                                   identified by "section"
                                   pic x(20).
```

バッファ領域のレコード構造をセットアップしています。この例では、データ名 book は XML タグ <book> で識別され、出現回数は 10 回、暗黙的に定義されている library-book-count フィールドで出現がカウントされます。book-toc-section データ項目も同様に定義されており、暗黙的に定義されている book-section-count でカウントされます。

0270～0390 行

```
01  mybuf.
    10  pic x(150) value
   '<library>'
   & '<book><title>book1</title>'
   &'       <toc><section>b1section1</section>'
   & '           <section>b1section2</section>'
   & '</toc></book>'.
    10 pic x(150) value
     '<book><title>book2</title>'
   & '      <toc><section>b2section1</section>'
   & '           <section>b2section2</section>'
   & '</toc></book>'
   & '</library>'.
```

静的バッファに読み取る XML ストリームを指定しています。このプログラムでは XML ストリームをコード内で指定していますが、標準入力を使用して XML をバッファ (mybuf) に格納することもできます。

0430～0470 行

```
perform until library-book-count = 0
  display "Book title         = '" book-title(library-book-count) "'"
  display "Number of Sects = " book-section-count(library-book-count)
  subtract 1 from library-book-count
end-perform
```

すべてのレコードにわたってループし、レコード内に含まれる全書籍のタイトルとセクション番号をレコードごとに表示します。book-section-count のインデックスとして library-book-count を使用している点に注意してください。

### 例 23-2 OCCURS 句と COUNT IN 句の使用方法

COUNT IN 句は、要素定義に関連するタグが XML ストリームに含まれているかどうかを判定する手段としても非常に役立ちます。たとえば、一連の XD レコードのうち、READ 文の実行中に読み取られるレコードを特定する手段として、きわめて一般的に使用されます。次に一例を示します。

```
0010  select book-or-author-file
0020        assign address of mybuf
```

```
0030        organization  is xml
0040        document-type is "bookdb.xsd"
0050        file status is xml-bookdb-status.
0060
0070  data division.
0080  xd book-or-author-file.
0090  01  book-rec                        identified by "book"
0100                                       count in book-count.
0110      05  price-val                   pic 999.99 identified by "price"
0120                                        is attribute
0130                                        count in price-count.
0140      05  title-val                   pic x(80) identified by "title".
0150
0160  01  author-rec                      identified by "author"
0170                                        count in author-count.
0180      05  author-val                  pic x(80).
0190
0200  working-storage section.
0210  01  xml-bookdb-status  pic s999.
0220  01  mybuf  pic x(256) value
0230     '<book price="123.00"><title>A title</title></book>'.
0240
0250  procedure division.
0260     open input book-or-author-file
0270     read book-or-author-file
0280     evaluate true
0290     when book-count = 1
0300         if  price-count not = 0
0310             display "got <book price=" price-val "><title>" title-val
0320                     "</title></book>"
0330         else
0340             display "got <book><title>" title-val "</title></book>"
0350         end-if
0360     when author-count = 1
0370         display "got <author>" author-val "</author>"
0380     end-evaluate
0390     close book-or-author-file
0400     stop run.
```

0010〜0050 行

```
  select book-or-author-file
        assign address of mybuf
        organization  is xml
        document-type is "bookdb.xsd"
        file status is xml-bookdb-status.
```

XML 入力のソースとしてファイル名とバッファ (mybuf) を割り当て、ファイル形式 (XML)、XML スキーマ名を指定するファイル名 (library.xsd)、およびファイル状態を示すデータ名を指定しています。

0080～0180 行

```
  xd book-or-author-file.
  01  book-rec                       identified by "book"
                                      count in book-count.
      05  price-val                  pic 999.99 identified by "price"
                                       is attribute
                                       count in price-count.
      05  title-val                  pic x(80) identified by "title".

  01  author-rec                     identified by "author"
                                       count in author-count.
      05  author-val                 pic x(80).
```

この XD は 01 レベルの項目を 2 つ定義しています。1 つは書籍レコード (book-rec)、もう 1 つは著作者レコード (author-rec) です。 COUNT IN で定義した情報は、読み取られているレコードを特定するときに使用します。

0220～0230 行

```
  01  mybuf  pic x(256) value
     '<book price="123.00"><title>A title</title></book>'.
```

静的バッファに読み取る XML ストリームを指定しています。このプログラムでは XML ストリームをコード内で指定していますが、標準入力を使用して XML をバッファ (mybuf) に格納することもできます。

0260～0380 行

```
      open input book-or-author-file
      read book-or-author-file
      evaluate true
      when book-count = 1
          if  price-count not = 0
              display "got <book price=" price-val "><title>" title-val
                      "</title></book>"
          else
              display "got <book><title>" title-val "</title></book>"
          end-if
      when author-count = 1
          display "got <author>" author-val "</author>"
      end-evaluate
```

EVALUATE 句によって、book-count または author-count の値を基に、読み取られているレコードが特定されます。

**例 23-3 COUNT IN による読み取りレコードの特定**

[XML 構文 — COUNT IN 句]

## NAMESPACE 句

多くの XML ドキュメントは名前空間を持っています。NAMESPACE 句を使用すれば、名前空間を COBOL で設定したり、識別することができます。NAMESPACE 句はデータ記述項で指定します。NAMESPACE 句の使用例を次に示します。

```
0010  xd book-file.
0020  78  book-ns              value "http://xml.microfocus.com/book.xsd".
0030  01  book                identified by "book"
0040                           namespace is book-ns.
0050      05  publisher pic x(80)
0060                           identified by "name"
0070                           namespace is
0080                             "http://xml.microfocus.com/publisher.xsd".
0090
0100      05  author           identified by "name"
0110                           namespace is
0120                            "http://xml.microfocus.com/author.xsd".
0130          10  author-first identified by "firstname"
0140                           pic x(80).
0150          10  author-last  identified by "lastname"
0160                           pic x(80).
0170      05  title            identified by "title"
0180                           pic x(80).
```

0020 行

```
78  book-ns              value "http://xml.microfocus.com/book.xsd".
```

名前空間とその値を定義しています。

0030～0040 行

```
01  book                identified by "book"
                          namespace is book-ns.
```

book-ns 内の値に等しい名前空間を持つデータレコードを定義しています。

0050～0080 行

```
05  publisher pic x(80)
                          identified by "name"
                          namespace is
                            "http://xml.microfocus.com/publisher.xsd".
```

名前空間を特定の URI (Uniform Resource Identifier) に設定しています。URI とは Web 上でアクセスする特定の一点を意味し、この例では指定された XML スキーマファイルを指します。

0100～0120 行

```
05  author          identified by "name"
                    namespace is
                    "http://xml.microfocus.com/author.xsd".
```

名前空間を特定の URI に設定しています。

0130～0160 行

```
10  author-first identified by "firstname"
                 pic x(80).
10  author-last  identified by "lastname"
                 pic x(80).
```

これらのデータ項目は author グループに属しているため、author と同じ名前空間を使用します。

0170～0180 行

```
05  title            identified by "title"
                          pic x(80).
```

title は book グループに属し、名前空間を指定されていないため、book と同じ名前空間を継承します。

### 例 23-4 NAMESPACE の使用方法

入力時には、タグと名前空間の両方が、XML ストリーム内のタグおよび名前空間と一致する必要があります。出力時には、タグは指定された名前空間で修飾されます。例 23-5 のコードは、次のような XML ストリームを生成します。

```
<?xml version="1.0" encoding="utf-8" ?>
<book xmlns="http://xml.microfocus.com/book.xsd">
   <name xmlns="http://xml.microfocus.com/publisher.xsd">
      Just a publisher
   </name>
   <name xmlns="http://xml.microfocus.com/author.xsd">
```

```
      <firstname>AuthorFirstname</firstname>
      <lastname>AuthorFirstname</lastname>
   </name>
   <title>
      This is the title
   </title>
</book>
```

特定の NAMESPACE の値はデータ名に置き換えることが可能で、それによって XML 内の名前空間の動的な検出が容易になります。この場合、出力では名前空間の照合は行われず、かわりに一致するタグの名前空間が、NAMESPACE で指定されるデータ項目内に追加されます。 出力では、データ項目で参照される名前空間が、出力タグの修飾に使用されます。たとえば、次のレコードと XML ストリーム入力を例に挙げます。まず、レコード例を示します。

```
     xd generic-file.
     01  generic-tag                  identified by generic-tag-name
                                       namespace is generic-tag-namespace.
         05  generic-tag-name          pic x(80).
         05  generic-tag-namespace     pic x(80).
         05  generic-tag-value         pic x(80).
```

続いて XML ストリーム入力の例を示します。

```
   <name xmlns="http://xml.microfocus.com/publisher.xsd">
      Just a publisher
   </name>
```

この場合、generic-tag-name として "name"、generic-tag-namespace として "http://xml.microfocus.com/publisher.xsd"、generic-tag-value として "Just a publisher" が得られます。

XML 構文 － NAMESPACE 句

## 拡張された COBOL 動詞

使用頻度の高い COBOL 動詞の一部も、XML を処理できるように拡張されています。具体的には次のとおりです。

- START
- OPEN
- READ
- WRITE
- REWRITE
- DELETE
- CLOSE

# CGI ベースのアプリケーション向けの手法

ASSIGN "http://...." 構文を使用すれば、CGI ベースの XML 対応アプリケーションのクライアントを、XML 拡張構文を直接記述して作成できます。次に、ローカルホストで動作する CGI ベースのプログラム、calccgi.exe のクライアントプログラムのコード例を示します。

```
0010  $set preprocess(prexml) o(calc.pp) endp
0020
0030  program-id. "calc".
0040  select xml-calc
0050        assign "http://localhost/cgibin/calccgi.exe"
0060        organization is xml
0070        document-type is "calc.xsd"
0080        file status is xml-calc-status.
0090
0100  data division.
0110  xd  xml-calc.
0120     01 xml-record identified by "request"
0130                                        count in xml-record-check.
0140                05 xml-num-1 pic 99 identified by "num1".
0150                05  xml-num-2 pic 99 identified by "num2".
0160                05  xml-func pic x(3) identified by "func".
0170                05  xml-result pic ----9 identified by "result".
0180
0190  working-storage section.
0200  01  xml-calc-status pic s9(9) comp-5.
0210
0220  procedure division.
0230
0240     open i-o xml-calc
0250     if xml-calc-status not = 0
0260               display "Error opening channel"
0270               stop run
0280     end-if
0290
0300     move "subtract" to xml-func
0310     move 2 to xml-num-1
0320     move 3 to xml-num-2
0330     perform doit
0340
0350     move "add" to xml-func
0360     perform doit
0370
0380     close xml-calc
0390     stop run.
```

```
0400
0410  doit section.
0420
0430    write xml-record
0440    display "Function call :" xml-func
0450     if xml-calc-status < 0
0460               exhibit named xml-calc-status
0470               stop run
0480     end-if
0490
0500     read xml-calc
0510
0520     if xml-calc-status < 0
0530               exhibit named xml-calc-status
0540               stop run
0550     end-if
0560
0570     if xml-record-check > 0 *> 出力
0580               exhibit named xml-num-1
0590               exhibit named xml-num-2
0600               exhibit named xml-func
0610               exhibit named xml-result
0620     end-if.
```

0040～0080 行

```
select xml-calc
       assign "http://localhost/cgibin/calccgi.exe"
       organization is xml
       document-type is "calc.xsd"
       file status is xml-calc-status.
```

CGI アプリケーションを xml-calc に割り当てています。

**例 23-6 XML 拡張構文を使用したクライアント側プログラム**

次の例は、このクライアントに対応するサーバ側プログラムのコードです。ファイル名 ":CGI:" の使用方法に着目してください。この ":CGI:" の割り当てによって、ランタイムシステムは CGI 標準の環境変数 CONTENT_LENGTH の値に従って動作するようになります。

```
0010      $set preprocess(prexml) o(calccgi.pp) endp
0020
0030    program-id. "calccgi".
0040
0050    select xml-calc assign ":CGI:"
0060            organization is xml
```

```
0070            document-type is "calc.xsd"
0080            file status is xml-calc-status.
0090
0100    data division.
0110 xd xml-calc.
0120    01 xml-record identified by "request"
0130                            count in xml-record-check.
0140              05 xml-num-1 pic 99 identified by "num1".
0150              05 xml-num-2 pic 99 identified by "num2".
0160              05 xml-func pic x(3) identified by "func".
0170              05 xml-result pic ----9 identified by "result".
0180
0190    working-storage section.
0200    01 xml-calc-status pic s9(9) comp-5.
0210
0220    procedure division.
0230
0240              open input xml-calc
0250              if xml-calc-status not = 0
0260                      display "Error opening input channel"
0270                      stop run
0280              end-if
0290
0300              read xml-calc
0310
0320              if xml-calc-status < 0
0330                      exhibit named xml-calc-status
0340                      display "Error reading channel "
0350                      stop run
0360              end-if
0370
0380              if xml-record-check > 0 *> 入力有効
0390                      close xml-calc
0400
0410                      open output xml-calc
0420                      if xml-calc-status not = 0
0430                              display "Error opening output channel"
0440                              stop run
0450                      end-if
0460
0470                      evaluate true
0480                      when xml-func = "add"
0490                              add xml-num-2 to xml-num-1
0500                                              giving xml-result
0510                      when xml-func = "sub"
0520                              subtract xml-num-2 from xml-num-1
```

```
0530                                            giving xml-result
0540                        when xml-func = "mul"
0550                                multiply xml-num-1 by xml-num-2
0560                                            giving xml-result
0570                        when xml-func = "div"
0580                                divide xml-num-1 by xml-num-2
0590                                            giving xml-result
0600                        end-evaluate
0610
0620                        write xml-record key is plain-text "Content-type:
text/xml" & x"0d0a"
0630
0640                        write xml-record
0650                end-if
0660                close xml-calc.
0670                stop run.
```

0050〜0080 行

```
select xml-calc assign ":CGI:"
        organization is xml
        document-type is "calc.xsd"
        file status is xml-calc-status.
```

ファイル名 ":CGI:" の割り当ては重要です。CGI プログラムへの入力が定義されていないため、CONTENT_LENGTH で指定された値を超えるサイズのデータの読み取りが試みられる可能性があるためです。この ":CGI:" の割り当てによって、ランタイムシステムは CGI 標準の環境変数 CONTENT_LENGTH の値に従って動作するようになります。

0380 行

```
if xml-record-check > 0 *> 入力有効
```

xml-record-check には、ストリーム内のデータ項目数が保持されています。この値がゼロより大きければ、READ 文の実行によってデータが見つかったことを意味します。

0390〜0410 行

```
close xml-calc

open output xml-calc
```

XML 入力ストリームを閉じ、出力ストリームとして再度開きます。CGI アプリケーションは通常、":CGI:" を入力モードで開いて XML データを読み取った後、いったん入力ストリームを閉じ、":CGI:" を出力モードで開いて出力します。

0620 行

```
write xml-record key is plain-text "Content-type: text/xml"
```

一部の Web サーバでは、CGI 出力の先頭にプリフィックスとして、plain-text の値 (Content-type: text/xml) を付ける必要があります。

**例 23-7 XML 拡張構文を使用したサーバ側の CGI アプリケーション**

# XML 対応 COBOL プログラムのコンパイル

XML 拡張構文を使用して XML に対応させた COBOL プログラムをコンパイルまたはビルドするには、Net Express の XML プリプロセッサによるコードの前処理が必要になります。この処理は、プログラム内に記述した XML 構文を、Net Express のコンパイラに正しく認識させるために不可欠です。

XML 対応の COBOL プログラムは、次のどちらかの方法でコンパイルまたはビルドします。

- Net Express — XML プリプロセッサを使用するように指定するコードを追加する必要がある。
- コマンド行 — COBOL プログラムのコンパイル前に XML プリプロセッサを呼び出すコマンド行引数を使用する必要がある。

## Net Express でのビルド

Net Express で XML 対応の COBOL プログラムをビルドするには、あらかじめ次の行をコードの先頭 (見出し部の前) に追加しておく必要があります。

```
$set p(prexml) [{OUT | O} outname] [WARN] [{PREPROCESS | P} ppname] endp
```

各パラメータの意味は次のとおりです。

| | |
|---|---|
| {OUT ¦ O} outname | プリプロセッサから出力されるソースファイルの名前を outname に指定します。このソースファイルには、XML 構文を処理して生成された COBOL ネイティブのコードが出力されます。 |
| WARN | 解析の診断警告を PREXML に出力させます。 |
| {PREPROCESS ¦ P} ppname | ソースファイルを読み込んで COPY 文を処理するため、ネストしたプリプロセッサを呼び出します。 |

次に例を示します。

```
$set p(prexml) warn endp
```

## コマンド行でのコンパイル

コード内で $set 文を使用している XML 対応プログラムは、XML 非対応のプログラムと同じ要領で、コマンド行でコンパイルできます。コード内で $set 文を使用していないか、$set 文のパラメータで指定した値を変更する場合は、次のように入力してコンパイルします。

```
cobol programname.cbl p(prexml)
        [{OUT | O} outname]
        [WARN]
        [{PREPROCESS | P} ppname]
        endp
```

programname にはコンパイルする COBOL プログラムのファイルプリフィックスを指定します。その他のパラメータは、$set 文のパラメータと共通です。これらのパラメータについては、前述した『Net Express でのビルド』を参照してください。

次に例を示します。

```
cobol foo.cbl p(prexml) warn o(foo.pp) endp
```

# 第 24 章 : XML 構文を含む COBOL 構造の生成

この章では、コマンド行ユーティリティ cbl2xml と CBL2XML ウィザードを使用して、XML 構文を含む COBOL データ構造を生成する方法について説明します。

## 24.1 CBL2XML の機能

cbl2xml ユーティリティと CBL2XML ウィザードは、どちらを使用しても、次の処理を実行できます。

- 既存の COBOL レコードから、XML 構文を含む COBOL レコードと、対応する XML スキーマを生成する。
- 既存の XML スキーマを基に、XML 構文を含む COBOL レコードを生成する。
- XML スキーマと XML ドキュメントを検証する。

### 24.1.1 cbl2xml ユーティリティの実行形式

cbl2xml はコマンド行で実行され、COBOL レコードを読み取って複数のファイルを生成します。このユーティリティのコマンド行では、大文字、小文字は区別されません。コマンド行構文は次のとおりです。

```
cbl2xml filename
        [-c cpyFile]
        [-x schemaFile]
        [-m mapFile]
        [-p prefix]
        [-d directiveFile]
        [-v validationFile]
        [-noprompt]
        [-nocountin]
```

各パラメータの意味を次に説明します。

filename　　cbl2xml の入力ファイル。次のいずれかを指定します。

- 従来的な COBOL データ構造を XML 構文を含む COBOL 構造に変換する場合は、cbl2xml で読み取る COBOL ファイルの名前を指定する。プログラムファイル (拡張子 .cbl) やコピーファイル (拡張子 .cpy) を指定できる。
- XML スキーマを XML 構文を含む COBOL 構造に変換する場合は、cbl2xml で読み取る XML スキーマファイルの名前を指定する (スキ

ーマファイルの拡張子は通常 .xsd)。

| | |
|---|---|
| cpyFile | オプション。COBOL コピーファイル (.cpy ファイル) の名前を指定します。cbl2xml は、このファイルに XML 拡張構文を含む COBOL を出力します。このパラメータを省略すると、入力ファイルとして指定された COBOL のプログラムファイルまたはコピーファイルの名前に -cbl2xml を連結した名前のファイルが、cbl2xml によって生成されます。たとえば、入力ファイルとして account.cbl という名前の COBOL プログラムファイルを指定した場合、cbl2xml によってデフォルトで生成される XML 拡張構文を含むコピーファイル名は account-cbl2xml.cpy になります。 |
| schemaFile | オプション。生成するスキーマファイルの名前を指定します。このパラメータを省略すると、入力ファイルとして指定された COBOL のプログラムファイルと同じベース名に拡張子 .xsd を付けた名前のスキーマファイルが、cbl2xml によって生成されます。たとえば、入力ファイルとして account.cbl という名前の COBOL プログラムファイルを指定した場合、cbl2xml によってデフォルトで生成されるスキーマファイル名は account.xsd になります。 |
| mapFile | オプション。生成されるマップファイル (または既存のマップファイル) の名前。次のルールが適用されます。<br><br>• 指定した名前のマップファイルが存在しない場合、入力ファイルとして指定した COBOL ファイル内の情報を基にマップファイルが生成される。<br>• 既存のマップを指定すると、cbl2xml によって上書きを確認するメッセージが表示され、yes と応答するとマップファイル内の新しい情報が、生成された COBOL ファイルに適用される。<br><br>マップファイルの詳細については、『COBOL データの XML 要素へのマッピング』を参照してください。 |
| prefix | オプション。生成されるデータ名の先頭に付加するプリフィックスを指定します。このパラメータは、COBOL ファイル内のデータレコードと同じ構造を持つデータレコードを、異なるデータ名で生成する場合に使用します。このパラメータを指定すれば、元のデータレコードを保持したまま、XML 用のデータレコードを生成できます。<br><br>プリフィックスは文字列として指定します。たとえば、-p xml- と指定すると、生成される各データレコードの名前の先頭に xml- が付加されます。 |
| directiveFile | オプション。XML に対応させる COBOL プログラムやコピーファイルのコンパイルで常に特定のコンパイラ指令を使用する場合は、コンパイラ指令を含む指令ファイルを、このパラメータで cbl2xml を使用するたびに指定します。<br><br>指令ファイルの詳細については、ヘルプトピックの『 Compiler directives in a user directives file』を参照してください。 |
| validationFile | 検証する XML スキーマまたは XML ドキュメントのファイル名。次のルール |

が適用されます。

- スキーマを指定した場合、W3C の Web サイト (http://www.w3.org/2001/XMLSchema) で規定されている XML の表記規則に従ってスキーマ自体の妥当性が検証される。
- XML ドキュメントを指定した場合、その中で指定している XML スキーマを基に、XML ドキュメントの妥当性が検証される。

続いて、-noprompt 引数と -nocountin 引数の使用方法を説明します。

-noprompt　cbl2xml に既存ファイルを自動的に上書きさせる場合に指定します。

警告：-noprompt を指定すると、cbl2xml が生成したファイルと同じ名前の既存ファイルは、それがユーザ側で編集したファイルや、サードパーティから提供されたスキーマであっても、すべて上書きされます。これらのファイルが上書きされてかまわない場合を除いて、このパラメータは使用しないでください。

-nocountin　オプション。生成される COBOL データ項目に COUNT IN 句を生成しないように指定します。このオプションは、CBL2XML マッピングファイルで指定されているあらゆる値より優先されます。

### 24.1.2 CBL2XML ウィザード

コマンドプロンプトより GUI が扱いやすいようであれば、CBL2XML ウィザードを使用して、同じ結果を得ることができます。同じ結果が得られるのは、このウィザードが基本的に cbl2xml ユーティリティのフロントエンドであるためです。この章の例で示している入出力は、cbl2xml ユーティリティと CBL2XML ウィザードのどちらを使用する場合も共通です。

方法

### 24.1.3 コピーファイルとスキーマの生成

XML 拡張構文を含むコピーファイルと XML スキーマは、既存のコピーファイルや COBOL プログラムから生成できます。生成されたコピーファイルには、既存の COBOL プログラムを XML に対応させるための情報が含まれています。また、生成されたスキーマファイルに準拠し、データを含む XML ドキュメントを生成すれば、それを COBOL プログラムの入力として使用できます。コピーファイルとスキーマファイルを生成するには、次の形式で cbl2xml を実行します。

```
cbl2xml filename
        [-c cobolFile]
        [-x schemaFile]
        [-d directiveFile]
```

```
        [-nocountin]
```

注記：CBL2XML ウィザードでコピーファイルとスキーマを生成するには、「COBOL を XML 構文 COBOL に変換する」オプションを使用します。

生成されるスキーマに準拠する XML 情報を COBOL プログラムから利用できるようにするため、cbl2xml は新しいレコードを含むコピーファイルを生成します。この新しいレコードに、XML の操作に必要な XML 拡張構文が含まれています。次にレコード生成の例を示します (この例は、パラメータとして -c、-x のどちらも指定していないことを前提としています)。

cbl2xml で処理する前の、元のコピーファイル record.cpy の内容は次のとおりです。

```
       01 A-FILE-RECORD.
         05 A-FILE-ITEM PIC XX.
```

次に示す XML 拡張構文を含む新しいコピーファイル record-cbl2xml.cpy が生成されます。

```
        01 a-file-record identified by "a-file-record" count in
           a-file-record-count.
         02 a-file-item pic X(2) identified by "a-file-item" count in
            a-file-item-count.
```

**例 24-1 XML 拡張構文を含むコピーファイルの生成**

XML 拡張構文の詳細については、『XML 拡張構文』の章を参照してください。

プログラムを XML に対応させるには、この record-cbl2xml.cpy の名前を record.cpy に変更するか、元の record.cpy 内のレコードを record-cbl2xml.cpy 内のレコードで置き換える必要があります。

COBOL プログラム全体を XML に対応させる手順も、XML 拡張構文を含むコピーファイルの生成と同様です。この場合、cbl2xml はプログラム内で見つかった各レコードと等価の情報を含むスキーマおよびコピーファイルを生成します。

record.cpy を基に cbl2xml が生成したスキーマ (record.xsd) を次に示します。

```
<?xml version="1.0" encoding="utf-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
   <element name="a-file-record">
      <complexType>
         <sequence>
            <element name="a-file-item">
               <simpleType>
                  <restriction base="string">
                     <length value="2"/>
```

```
                </restriction>
              </simpleType>
            </element>
          </sequence>
        </complexType>
    </element>
</schema>
```

**例 24-2 COBOL ファイル定義から生成された XML スキーマ**

次の表は、このスキーマに含まれる各種のタグの意味を、表現するレコードのコンテキストで説明しています。

| スキーマ内のタグ | 説明 |
|---|---|
| <element name="a-file-record"> | グループレベルのデータ名 (A-FILE-RECORD) の定義。この要素が、グループ内でデータ項目を定義しているすべての要素の親要素になります。 |
| <complexType> | グループレコードに 1 つ以上の従属データ項目が含まれることを示しています。 |
| <sequence> | 従属レコードを定義する一連のタグの開始を示しています。 |
| <element name="a-file-item"> | 従属データ項目 A-FILE-ITEM のデータ名の定義。要素タグが、他の要素タグによって囲まれている (子要素) ため、この項目は従属データ項目になります。 |
| <simpleType> | データ項目に従属データ項目が存在しないことを示しています。 |
| <restriction base="string"> | データの制約を定義しています。この場合、データは COBOL データレコードの PIC XX の定義に従い、文字列に限定されています。 |
| <length value="2"/> | レコードの長さを定義しています。この場合、COBOL データレコードの PIC XX の定義の従い、長さは 2 に指定されています。 |

## 24.1.4 XML スキーマに基づく XML 対応 COBOL の生成

前述した『コピーファイルとスキーマの生成』で示した形式を使用して cbl2xml で生成したスキーマは、既存の COBOL レコードに基づいており、COBOL レコードから取り込まれた要素名を含んでいます。実際の用途では、このようなスキーマは十分ではありません。サードパーティとの間で XML 情報を交換するには、相手側が認識できる要素名を使用する必要があるからです。多くの業界では業界標準のスキーマが提供されており、その中には業界全体にわたって情報交換に一貫して使用される要素群が含まれています。サードパーティのスキーマに準拠する XML 拡張構文を含む COBOL レコードを生成するには、次の形式で cbl2xml を実行します。

```
cbl2xml XMLschema
                [-c cobolFile]
```

```
                [-d directiveFile]
                [-prompt | -noprompt]
```

注記：CBL2XML ウィザードで XML 対応の COBOL プログラムを生成するには、「XML スキーマを COBOL に変換する」オプションを使用します。

ここでは一例として、ある業界で widgets.xsd という名前の業界標準スキーマが提供されており、このスキーマが製品「Widget」の注文情報を企業間で交換するために使用されている場合を想定します。 まず、widgets.xsd の内容を次に示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
        <xsd:element name="Availability" type="xsd:string"/>
        <xsd:element name="Base_Price" type="xsd:string"/>
        <xsd:element name="Base_Price_And_Unit">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="Base_Price"/>
                                <xsd:element ref="Unit"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="Catalogue_Id" type="xsd:int"/>
        <xsd:element name="Widgets">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="Product_Id"
minOccurs="0"/>
                                <xsd:element ref="Company_Id"/>
                                <xsd:element ref="Catalogue_Id"/>
                                <xsd:element ref="SKU"/>
                                <xsd:element ref="Product_Series"
minOccurs="0"/>
                                <xsd:element ref="Availability"/>
                                <xsd:element ref="Sale"/>
                                <xsd:element ref="Base_Price_And_Unit"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="Company_Id" type="xsd:int"/>
        <xsd:element name="Product_Id" type="xsd:int"/>
        <xsd:element name="Product_Series" type="xsd:string"/>
        <xsd:element name="SKU" type="xsd:string"/>
        <xsd:element name="Sale" type="xsd:string"/>
        <xsd:element name="Unit" type="xsd:string"/>
```

```
</xsd:schema>
```

**例 24-3 製品「Widgets」用の業界標準 XML スキーマ**

cbl2xml はこのスキーマを読み取り、次に示すコピーファイル (widgets.cpy) を生成します。

```
       01 Widgets identified by "Widgets".
        02 Product-Id PIC X(80) identified by "Product_Id" count in
           Product-Id-count.
        02 Company-Id PIC X(80) identified by "Company_Id" count in
           Company-Id-count.
        02 Catalogue-Id PIC X(80) identified by "Catalogue_Id" count in
           Catalogue-Id-count.
        02 SKU PIC X(80) identified by "SKU" count in SKU-count.
        02 Product-Series PIC X(80) identified by "Product_Series" count
            in Product-Series-count.
        02 Availability PIC X(80) identified by "Availability" count in
           Availability-count.
        02 Sale PIC X(80) identified by "Sale" count in Sale-count.
        02 Base-Price-And-Unit identified by "Base_Price_And_Unit" count
            in Base-Price-And-Unit-count.
         03 Base-Price PIC X(80) identified by "Base_Price" count in
           Base-Price-count.
         03 Unit PIC X(80) identified by "Unit" count in Unit-count.
```

**例 24-4 widgets.xsd を基に生成された XML 対応の COBOL コピーファイル**

コピーファイルを生成すれば、Widget に関する情報の送受信に使用する COBOL プログラムに、そのファイルをインクルードできます。

## 24.1.5 COBOL データから XML 要素へのマッピング

『XML スキーマに基づく XML 対応 COBOL の生成』の項の例では、スキーマ内で検出された要素名を基に COBOL データレコードが生成されます。 ただし、既存の COBOL レコードを、サードパーティのスキーマとの通信専用に使用することも可能です。 そのためには、マップファイルを使用して、COBOL レコードとスキーマの要素を直接的に関連付ける必要があります。

マップファイルを生成するには、次の形式で cbl2xml を実行します。

```
cbl2xml filename -m mapFile
```

注記：CBL2XML ウィザードでマップファイルの生成や読み取りを行うには、「XML スキーマを COBOL に変換する」オプションを使用し、「高度な構成オプション」ページでマップファイルを指定します。

アプリケーションを XML に対応させ、既存の COBOL レコードとスキーマを基に、XML ドキュメントで情報を交換する手順は次のとおりです。

1. cbl2xml でマップファイルを生成します。新しいマップファイルを生成する場合は、存在しないファイルの名前を指定します。指定した名前のマップファイルが見つからない場合、cbl2xml は新しいマップファイルを生成します。
2. マップファイルを編集します。生成された要素の名前を、データ交換に使用するスキーマ内の既存要素の名前に変更してください。
3. ステップ 1 と同じ形式で cbl2xml を再実行します。ただし、今回は編集した既存マップファイルの名前を指定します。指定された名前のマップファイルが見つかると、cbl2xml はそのファイル内のマッピングを COBOL レコードに適用します。

この項の以下の内容では、上記手順の詳細を説明します。

指定したマップファイルが存在しない場合、cbl2xml は COBOL ファイルからデータ構造を取り込んで XML に変換し、その情報をマップファイルとして保存します。

次に一例を示します。

Product_info.cpy に次のレコードが含まれているものとします。

```
       01  PRODUCT-INFO.
           05 PROD-ID     PIC 9(6).
           05 COMPANY-ID  PIC 9(4).
           05 CATALOG-NUM PIC 9(8).
           05 SKU-REF     PIC X(10).
```

通信に使用する widget.xsd 内の要素定義を次に示します。

```
        <xsd:element name="Widgets">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="Product_Id"
minOccurs="0"/>
                                <xsd:element ref="Company_Id"/>
                                <xsd:element ref="Catalogue_Id"/>
                                <xsd:element ref="SKU"/>
                                <xsd:element ref="Product_Series"
minOccurs="0"/>
                                <xsd:element ref="Availability"/>
                                <xsd:element ref="Sale"/>
                                <xsd:element ref="Base_Price_And_Unit"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
```

### 例 24-5 COBOL レコードと XML スキーマ

cbl2xml が product_info.cpy から生成するマップファイルを次に示します。

```
<?xml version="1.0" encoding="utf-8"?>
<xml-cobol-mapping-file
xmlns="http://xml.microfocus.com/schema/xml/v1.0/mfxmlmap.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xml.microfocus.com/schema/xml/v1.0/mfxmlmap.xsd
mfxmlmap.xsd">
   <xml-cobol-mapping>
      <cobol-name>product-info</cobol-name>
      <cobol-count-in>true</cobol-count-in>
      <xml-name>product-info</xml-name>
      <xml-type>group</xml-type>
      <xml-attribute>false</xml-attribute>
      <xml-namespace></xml-namespace>
   </xml-cobol-mapping>
   <xml-cobol-mapping>
      <cobol-name>product-info.prod-id</cobol-name>
      <cobol-count-in>true</cobol-count-in>
      <xml-name>prod-id</xml-name>
      <xml-type>string</xml-type>
      <xml-attribute>false</xml-attribute>
      <xml-namespace></xml-namespace>
   </xml-cobol-mapping>
   <xml-cobol-mapping>
      <cobol-name>product-info.company-id</cobol-name>
      <cobol-count-in>true</cobol-count-in>
      <xml-name>company-id</xml-name>
      <xml-type>string</xml-type>
      <xml-attribute>false</xml-attribute>
      <xml-namespace></xml-namespace>
   </xml-cobol-mapping>
   <xml-cobol-mapping>
      <cobol-name>product-info.catalog-num</cobol-name>
      <cobol-count-in>true</cobol-count-in>
      <xml-name>catalog-num</xml-name>
      <xml-type>string</xml-type>
      <xml-attribute>false</xml-attribute>
      <xml-namespace></xml-namespace>
   </xml-cobol-mapping>
   <xml-cobol-mapping>
      <cobol-name>product-info.sku-ref</cobol-name>
      <cobol-count-in>true</cobol-count-in>
      <xml-name>sku-ref</xml-name>
```

```
        <xml-type>string</xml-type>
        <xml-attribute>false</xml-attribute>
        <xml-namespace></xml-namespace>
    </xml-cobol-mapping>
</xml-cobol-mapping-file>
```

**例 24-6 生成されたマップファイル**

マップファイルは XML ドキュメントです。cbl2xml によって生成される各種のタグには、次に示すように COBOL レコードに関する特定の情報が含まれています。

| マップファイル内のタグ名 | 説明 |
|---|---|
| <xml-cobol-mapping-file> | このファイルが、COBOL プログラム内に XML 構造を生成するために使用されるマップファイルであることを示しています。この要素が、マップファイルのルート要素です。 |
| <xml-cobol-mapping> | 1 つの COBOL データ項目に関する情報を含んでいます。<cobol-name>、<xml-name>、<xml-type>、<xml-attribute>、および <xml-namespace> の親要素です。 |
| <cobol-name> | 1 つの COBOL レコードから取り込まれたデータ項目とその親項目の名前を連結した文字列が含まれています。親項目とデータ項目の名前は、ピリオド (.) で連結されています (例：a-file-record.a-file-item)。 |
| <cobol-count-in> | <cobol-name> 内の COBOL 項目に COBOL XML の COUNT IN 句を生成するかどうかが示されています。有効値は true または false、デフォルト値は true です。この値は、-nocountin コマンド行オプションを指定した場合には無視されます。 |
| <xml-name> | <cobol-name> 内のレコード名に対応する XML 要素名を指定しています。 |
| <xml-type> | COBOL 項目のマッピング先の XML 基本データ型。このパラメータは、PIC 句を持つ COBOL データ項目だけに適用されます。 |
| <xml-attribute> | <xml-name> 内で指定された要素を、XML スキーマ内で属性として扱うかどうかを示しています。有効値は true または false です。 |
| <xml-namespace> | <xml-name> 内で指定された要素の XML 名前空間を指定するオプションの要素。グループレベルで指定された名前空間は、そのグループに属する子要素に自動的に継承されます。 |

マップファイルを生成した後、そのファイルを編集し、スキーマ内の XML 要素名と COBOL レコード内の COBOL 名を関連付けることによって、独自のマッピングを作成できます。具体的

には、マップファイル内の <xml-name> で指定されている名前を、スキーマ内の要素名に合致するように編集します。この例では、例 24-4 に示したマップファイル内の <xml-name> を次のように変更します。

| 変更前 | 変更後 |
|---|---|
| <xml-name>product-info</xml-name> | <xml-name>Widgets</xml-name> |
| <xml-name>prod-id</xml-name> | <xml-name>Product_Id</xml-name> |
| <xml-name>company-id</xml-name> | <xml-name>Company_Id</xml-name> |
| <xml-name>catalog-num</xml-name> | <xml-name>Catalogue_Id</xml-name> |
| <xml-name>sku-ref</xml-name> | <xml-name>SKU</xml-name> |

この時点で、編集したマップファイルを指定して cbl2xml を再実行します。その結果、スキーマとコピーファイルが生成されます。生成されたスキーマは、マップファイルに反映した既存要素の数にも左右されますが、元のスキーマにかなり似たものになります。このスキーマはデフォルトで生成され、重要性も低いため、必ずしも使用する必要はありません。生成されたコピーファイルには、元の COBOL レコードに、マップファイル内の情報に基づく適切な XML 拡張構文が付加されています。この例では、cbl2xml は次のコピーファイルを生成します。

```
01 product-info identified by "Widgets" count in Widgets-count.
 02 prod-id pic 9(6) identified by "Product_Id" count in
    Product_Id-count.
 02 company-id pic 9(4) identified by "Company_Id" count in
    Company_Id-count.
 02 catalog-num pic 9(8) identified by "Catalogue_Id" count in
    Catalogue_Id-count.
 02 sku-ref pic X(10) identified by "SKU" count in SKU-count.
```

この新しいレコードをコピーし、COBOL プログラム内に貼り付けて、元のレコードを上書きします。これで COBOL レコードは XML 対応になり、Widgets.xsd スキーマに準拠する XML ドキュメントをやり取りできるようになります。

### 24.1.6 XML 対応レコードへのアクセス

XML 対応レコードの作成後は、プログラムの手続き部から、それらのレコードにアクセスする必要があります。この用途には、通常の COBOL 動詞を使用できます (Net Express では、これらの動詞は XML 用に拡張されています。詳細については、『XML 拡張構文』の章を参照してください)。

手続き部で widgets.xml ファイルの内容を出力するプログラムの一例を次に示します。

```
$set p(prexml) endp
select xml-stream assign "widgets.xml"
                  organization is xml
                  document-type is "widgets.xsd"
```

```
                    file status is file-status-spec.

xd xml-stream.
       copy "widgets.cpy".

data division.
working-storage section.
01 file-status-spec PIC S9(9).

procedure division.
  open output xml-stream
  write widgets
  display file-status-spec
  close xml-stream
stop run.
```

**例 24-7 XML を使用する COBOL プログラム**

### 24.1.7 XML ドキュメントの検証

cbl2xml では、XML スキーマ (.xsd ファイル) や XML ドキュメント (.xml ファイル) の形式の妥当性を検証することもできます。 XML ドキュメントを検証する場合、cbl2xml はドキュメントの内容を、そのドキュメント内で参照されているスキーマを基準にチェックします。 検証を行う場合、cbl2xml は次の形式で実行します。

```
cbl2xml -v filename
```

filename には、検証するスキーマまたは XML ドキュメントのファイル名を指定します。

注記：CBL2XML ウィザードでファイルを検証するには、「XML スキーマを検査する」オプションまたは「XML インスタンスドキュメントを検査する」オプションを使用します。

# 第 25 章 : クライアント/サーバ結合 (CSBIND)

この章では、クライアント/サーバ結合 (CSBIND) のしくみについて説明します。 また、開発したプログラムを汎用クライアント/サーバモジュールに接続する方法についても説明しています。

Dialog System のアプリケーションで CSBIND を使用する場合は、『*Dialog System ユーザガイド*』オンラインマニュアルの『*クライアント/サーバ結合の使用*』の章を参照してください。 Dialog System 専用のコードサンプルが記載されています。

## 25.1 はじめに

クライアント/サーバ結合 (CSBIND) は、クライアント数が 300 以下の環境を対象とした、シンプルな軽量リモートプロシージャコール (RPC) のメカニズムです。CSBIND を使用すれば、通信プログラミングに伴う手続きに煩わされることなく、クライアント/サーバアーキテクチャを実装できます。 CSBIND には AAI (アプリケーション間インターフェイス) コンポーネントが含まれており、このコンポーネントを通じて旧バージョンの Net Express との互換性が確保されます。

CSBIND によって実現される機能とパフォーマンスは、 さまざまなビジネスニーズに適しています。しかも、CSBIND は実装が容易で、運用のコスト効率にも優れています。

CSBIND を使用すれば、アプリケーションに通信用のコードを含める必要がなくなります。 CSBIND には **mfclient**、**mfserver**、**aaiclnt**、**aaisrvr** という 4 つのモジュールがあり、これらのモジュールによって通信に必要なコードが完全に網羅されます (AAI を使用する場合、必要なモジュールは **aaiclnt** と **aaisrvr** の 2 つだけです)。

これらのモジュールが、設定ファイル内の情報に基づいて通信リンクとデータ転送を管理します。通信リンクで接続されたユーザプログラムと連携して動作することもあります。 **mfclient** モジュールは、ユーザインターフェイスを処理するプログラムなど、ユーザ側で作成したクライアントプログラムから呼び出され、**mfserver** はユーザ側で作成したサーバプログラム (データアクセスやアプリケーションロジックを管理するプログラムなど) を呼び出します。

ユーザプログラム (クライアント、サーバ) は、コピーファイル **mfclisrv.cpy** を使用して、汎用クライアント/サーバモジュールとの通信を行います。

Net Express には、CSBIND の使用方法を紹介するためのサンプルアプリケーションが付属しています。 このサンプルはクライアントとサーバ間でファイルデータを転送するアプリケーションで、Net Express の **demo** ディレクトリ内の **Csbind** サブディレクトリにインストールされています。

## 25.2 CSBIND のしくみ

CSBIND を CCI とともに使用する環境では、**mfserver** のコピーがサーバ階層で動作します。このコピーは特定のクライアント専用ではなく、一致するサーバ名を使用しているクライアントが存在すれば、該当するすべてのクライアントとの間で通信を行います。**mfserver** は、クライアントからの要求を受信して接続を確立または切断するモジュールであり、その他の処理は行わないため、あらかじめ定義されているデフォルト値のままで実行できます。

CSBIND での情報の流れを次に示します。ここではクライアント/サーバプログラムとして、サンプルアプリケーション **Csbind** に含まれるプログラムを使用しています。

1. ユーザ側で作成したクライアントプログラム (**netxcli.cbl**) がアプリケーションの初期化コードを実行し、ユーザにファイル名の入力を求め、入力されたファイルが存在すれば **mfclient** を呼び出します。
2. **mfclient** は最初の呼び出し時に、設定ファイル (**netxdem.cfg**) から情報を読み取ります。
3. **mfserver** が接続要求を受信します。
4. **mfserver** がクライアントごとにセカンダリサーバのプロセスを生成します。このセカンダリサーバの名前は、ベースサーバ名に ID 番号を連結した文字列 (**mfserver01** など) を基に、内部的に生成されます。設定ファイル内でサーバ名を明示的に指定することもできます。
5. **mfserver** がセカンダリサーバ名 (以下、**mfserver01** とします) を **mfclient** に返信し、このクライアントとの対話が終了します。
6. **mfclient** がセカンダリサーバ **mfserver01** に接続し、設定ファイルから取得したパラメータを LNK-PARAM-BLOCK を介して渡します。詳細については、後述する『*CSBIND のコピーファイル*』を参照してください。
7. **mfserver01** が、ユーザ側で作成されたサーバプログラム **netxserv.cbl** を呼び出し、連絡節を通じて **mfclient** から受信したパラメータを渡します。
8. **netxserv.cbl** は、最初に呼び出された時点でアプリケーション初期化コードを実行し、そのまま終了します。制御が **mfserver01** に戻ります。
9. **mfserver01** が **mfclient** に制御を戻します。
10. **mfclient** は、接続が確立されたことを確認した後、ユーザ側のクライアントプログラム (**netxcli.cbl**) に制御を戻します。
11. **netxcli.cbl** がエンドユーザによって指定されたファイルを開き、レコードを 1 件読み込んで **mfclient** を呼び出し、連絡節を介してレコードのデータを渡します。
12. **mfclient** が、自身の内部バッファを介して、受信したデータを **mfserver01** に渡します。
13. **mfserver01** は、このデータをサーバプログラム **netxserv.cbl** に渡します。netxserv.cbl はファイルを作成し、その中に受信したレコードを書き込みます。

    以上のレコードの読み取りと書き込みが、クライアントプログラム **netxcli.cbl** でファイルの末尾が検出されるまで繰り返されます。netxcli.cbl はファイルの末尾に達すると、ファイルを閉じて処理終了をサーバに通知します。この通知を受信したサーバは、ファイルのコピーが作成されたディレクトリの名前を返信します。

14. **netxcli.cbl** はサーバ上に作成されたファイルの名前を表示し、LNK-CNTRL-FLAG を client-ending に設定して **mfclient** を呼び出した後、終了します。

- **mfclient** が client-ending パラメータを **mfserver01** に渡します。このパラメータは mfserver01 を介してサーバプログラム **netxserv.cbl** に渡され、netxserv.cbl が終了します。

- **mfclient** がベースサーバ **mfserver** に、セカンダリサーバ **mfserver01** の終了を通知します。

AAI を使用する場合には、CSBIND の処理内容が異なります。この場合もアプリケーションから **mfclient** を呼び出しますが、**mfclient** では **aaiclnt** の呼び出しのみが実行され、クライアント階層での AAI とのやり取りは、すべて **aaiclnt** で処理されます。 CSBIND では、AAI に完全対応した製品と、AAI 対応の軽量 (Thin) クライアントのどちらも使用できます。 サーバ階層では、AAI が要求を受信するたびに **mfserver** のインスタンスを生成し、**mfserver** が **aaisrvr** を呼び出します (AAI が直接 **aaisrvr** を呼び出すように設定することも可能です)。 詳細については、AAI のマニュアルを参照してください。

# 25.3 CSBIND の使用方法

アプリケーションで CSBIND を使用するには、次の作業を行う必要があります。

- **mfclient** および **mfserver** の動作と通信接続を制御するための設定ファイル (**.cfg** ファイル) を作成する。 CSBIND の動作に影響を与える次のような情報を、このファイルにパラメータとして指定します。
    - 使用する通信プロトコル
    - **mfserver** から呼び出すユーザプログラムの名前

- ユーザ側で作成するクライアントプログラムに、**mfclient** モジュールを呼び出すコードを追加する。
- ユーザ側で作成するサーバプログラムに、**mfserver** モジュールからの呼び出しを可能にするコードを追加する。

これらの作業について、以下の項で詳しく説明します。

# 25.4 CSBIND のコピーファイル

**mfclient** と **mfserver** は、コピーファイル **mfclisrv.cpy** 内で定義されているパラメータブロック (LNK-PARAM-BLOCK) を介して相互通信を行います。 また、要求に応じて呼び出したユーザプログラムに情報を渡すときにも、同じパラメータブロックを使用します。 この **mfclisrv.cpy** ファイルは、**mfclient** と **mfserver** を使用するすべての COBOL プログラムでインクルードする必要があります。

**mfclisrv.cpy** ファイルは、Net Express のインストールディレクトリ内の SOURCE サブディレクトリにインストールされています。

# 25.5 CSBIND の設定ファイル

CSBIND では、**mfclient** および **mfserver** の動作と通信リンクを、設定ファイルで制御します。

CSBIND の設定ファイルは、拡張子が **.cfg** の ASCII ファイルです。 設定ファイル名を特に指定しなければ、デフォルトの設定ファイル名 (**mfclisrv.cfg**) が使用されます。

設定ファイルはアプリケーションごとに作成できるほか、1 つの設定ファイルに複数のエントリを記述することもできます。 複数のエントリを記述する場合は、それぞれのアプリケーションを表すタグを入力し、それに続いて当該アプリケーション用の各パラメータを入力します。さらに、LNK-PARAM-BLOCK パラメータブロックの LNK-TAGNAME フィールドを使用して、クライアントプログラム内でタグ名を指定する必要があります。 この方法でデフォルトの設定ファイル (**mfclisrv.cfg**) 内に複数のエントリを記述すれば、コマンド行入力を簡潔化できます。

クライアント/サーバアプリケーションの設定が正しくない場合、設定ファイル内で無効なエントリが見つかると、プログラムは当該エントリの詳細情報を示すメッセージを画面表示して終了します。

このエラーは **mfclisrv.log** ファイルにも記録されます。このファイルは現在のディレクトリ (または MFLOGDIR 環境変数で指定されるディレクトリ) に作成されます。 このため、端末に直接接続されていないサーバでもメッセージを記録できます。

### 25.5.1 設定ファイルのパラメータ

**mfclisrv.cfg** 設定ファイル内に指定できる全パラメータを、以下にアルファベット順で示します。指定を省略したパラメータについては、デフォルト値が使用されます。

aaiservice=*name*
: PIC X(128) (デフォルト：MFCLISRV)

  **CSBIND** で AAI を使用する場合の接続先 AAI サービスの名前。 AAI 管理システムの論理サービス名エントリです。

cblksize=*nnnn*
: PIC X(4) COMP-X (デフォルト：0)

  Dialog System の制御ブロックのサイズ。 Dialog System を使用する場合のみ指定します。

clierrprog=*name*
: PIC X(128) (デフォルト：なし)

  **mfclient** の代わりに通信エラーを処理するプログラムの名前。 **mfclient** エラーを呼び出し元プログラムで処理する場合は SAME と指定します。

  このパラメータは AAI ではサポートされていません。

commsapi=*api*
: PIC X(4) (デフォルト：CCI)

  通信 API。有効値は CCI、AAI、NONE。 2 階層型アプリケーションの開発で、通信プログラムを使用せずに 1 台の PC 上でテストを実行す

る場合は、NONE を指定します。 その場合、データは **mfclient** とユーザのサーバプログラム間で直接交換され、**mfserver** と通信に必要な要素はバイパスされます。

**注記**：ユーザのサーバプログラムとして既存アプリケーションを使用している場合、このオプションは指定できません。

compress=*nnn* PIC 9(3) (デフォルト：000)

圧縮ルーチン番号。たとえば、001 を指定すると CBLDC001 ルーチンが使用されます。 0 を指定するとデータ圧縮は行われません。

dblksize=*nnnn* PIC X(4) COMP-X (デフォルト：0)

ユーザ側で作成したクライアントプログラムとサーバプログラム間のデータ交換に使用されるユーザデータ領域のサイズ。 Dialog System アプリケーションの場合は、この値が Dialog System のデータブロックのサイズになります。

**警告**：実際に使用しているデータブロックのサイズより小さい値を指定すると、予期しない結果につながります。

eblksize=*nnnn* PIC X(4) COMP-X (デフォルト：0)

Dialog System のイベントブロック (オプション) のサイズ。

このパラメータは AAI ではサポートされていません。

machinename=*name* PIC X(34)

*servername* パラメータで指定したサーバが動作しているマシン名。このパラメータを指定すれば、指定された名前と合致するサーバの検索を省略できます。

このパラメータは AAI ではサポートされていません。

maxtimeout=*nnnn* PIC X(4) COMP-5 (デフォルト：0)

**mfclient** がサーバへの接続を試み、最終的に失敗するまでに許容されるタイムアウトの回数。 デフォルト値 (0) は無制限を意味し、接続が成功するまで試行が継続されます。

maxtrans=*nn* PIC 99 (デフォルト：0)

転送パッケージの最大サイズ (KB 単位)。 有効な値は 1～62。バッファ全体の長さが指定値を超える場合、バッファは *maxtrans* サイズずつ分割して転送されます。

このパラメータは AAI ではサポートされていません。

midconfig=*filename* PIC X(128) (デフォルト：なし)

**mfserver** から呼び出されたときに、**mfclient** が階層間ソリューションの一部として使用する設定ファイルの名前。このパラメータを指定すると、**mfserver** が一種のルータとして機能し、ローカルの **mfclient** モジュールにデータを渡します。**mfclient** モジュールは、この設定ファイルを使用して他のサーバを検索し、通信を行います。この方法を使用すると、プロトコルと通信 API をマシンごとに変更できます。

このパラメータは AAI ではサポートされていません。

protocol=*protocol* PIC X(8) (デフォルト：CCITC32)

使用する CCI プロトコル。このパラメータは、*commsapi* が CCI に設定されている場合のみ有効です。サポートされている CCI プロトコルのいずれかを指定します。Novell IPX を使用する場合は CCIIX32、NetBEUI は CCINB32、動的データ交換 (DDE) は CCIDE32、TCP/IP は CCITC32 をそれぞれ指定します。なお、*commsapi* の値がデフォルト (CCI) の場合に *protocol* を指定しないと、デフォルトプロトコルの TCP/IP (CCITC32) が CSBIND で使用されます。

このパラメータは AAI ではサポートされていません。

scrntype=*type* PIC X(4) (デフォルト：なし)

ユーザプログラムが GUI とコマンド行インターフェイスの両方をサポートしている場合に、使用しているインターフェイスの種類をクライアントプログラムに通知する手段として使用できます。*scrntype* の値は CSBIND モジュールでは検証されません。

このパラメータは AAI ではサポートされていません。

servername=*server* PIC X(14) (デフォルト：MFCLISRV)

通信に使用するサーバ名。

このパラメータは AAI ではサポートされていません。

setenv=*name value* PIC X(148) (デフォルト：なし)

あらゆるサーバプログラムの実行前に設定すべき環境変数を指定します。指定形式は次のとおりです。

variable name PIC X(20)

variable value PIC X(128)

変数名 (name) 前と値 (value) のフィールドは 1 つ以上の空白で区切

ります。 *setenv* は 9 個まで指定できます。

このパラメータは AAI ではサポートされていません。

srvanim=*x*　PIC X(16)　(デフォルト：N)

サーバ上でのユーザプログラムのアニメートの可否を指定します。値は Y または N です。Y を指定するとサーバ上でのアニメートが可能になり、COBSW=+A が動的に設定されるため、**mfserver** をいったん停止し、COBSW=+A を指定して再起動する必要がなくなります。

UNIX システムでは、x,*filename* と指定することによって、複数のセッションにわたるユーザプログラムのアニメートを有効化できます。詳細については、後述する『*アプリケーションのアニメート*』を参照してください。

このパラメータは AAI ではサポートされていません。

srverrprog=*name*　PIC X(128)　(デフォルト：なし)

**mfserver** の代わりに通信エラーを処理するプログラムの名前。*srvprog* で指定した **mfserver** エラーの処理プログラムと同じ場合は SAME と指定します。

このパラメータは AAI ではサポートされていません。

srvprog=*name*　PIC X(128)

**mfserver** から呼び出されるユーザプログラムの名前。

srvtier=*name*　PIC X(128)　(デフォルト：mfserver)

サーバ階層プログラムの名前。指定した場合、**mfserver** はこのプログラムから呼び出されます。

このパラメータは AAI ではサポートされていません。

subserver=*base-name*　PIC X(14)　(デフォルト：なし)

サブサーバプロセスとの通信に使用するベース名。デフォルトのサーバ名は **mfclisrv** で、この名前に 00001 から始まる番号を連結した文字列 (**mfclisrv00001**、**mfclisrv00002**、...) がサブサーバ名になります。このパラメータを **newserv** に設定すると、サブサーバ名は **newserv00001**、**newserv00002**、... となり、ベースサーバ名を変更することなく、個々のアプリケーションごとに異なるサブサーバ名を使用できます。

このパラメータは AAI ではサポートされていません。

timeout=*nnnn*　PIC X(4) COMP-5　(デフォルト：120 秒)

タイムアウトの長さ。このパラメータに設定した値は、システムのタイムアウトのデフォルト値より優先されます。タイムアウトは 1/10 秒単位で指定します。 実行中の呼び出しには、最初の呼び出し時点で有効だったタイムアウトがそのまま使用されます。

このパラメータは AAI ではサポートされていません。

ublksize=*nnnn* PIC X(4) COMP-X (デフォルト:0)

オプションのユーザデータブロックのサイズ。

useraudit=*x* PIC X (デフォルト:N)

クライアントの接続および切断の詳細情報をログに記録するかどうかを指定します。値は Y または N。 このパラメータを Y に設定すると、*日付*、*時刻*、*接続/切断の指示子*、*サーバ名*、*マシン名*、および *プロトコル* がクライアントとサーバの両方でログに記録されます。 これらの情報はシステムログファイルに記録されます。後述する『*監査トレールの作成*』および『*システムログファイル*』を参照してください。

## 25.5.2 設定ファイルで指定すべき最小限のパラメータ

設定ファイルで指定すべき最小限のパラメータは、次の要因によって左右されます。

- 使用するアプリケーション
- 動作しているサーバ数
- 使用する通信プロトコル

常に指定すべき必須パラメータを次に示します。

*dblksize* データ転送に使用するユーザデータ領域のサイズ。

*srvprog* **mfserver** から呼び出され、サーバ側の処理を実行するプログラムの名前。 サーバマシン上で動作し、データアクセスや業務ロジックを実行するユーザプログラムを指定します。

複数のサーバが動作する環境では、使用するサーバの名前を *servername* で指定する必要があります。

TCP/IP を使用しない場合は、*protocol* を必ず設定します。

設定ファイルで指定しないパラメータについては、デフォルト値が使用されます。

## 25.5.3 設定ファイルの場所の特定

設定ファイルの場所は特に決まっておらず、必要に応じて任意の場所に置くことができます。**mfclient** と **mfserver** は、どちらも次の順序で設定ファイルを見つけ出します。

1. MFCSCFG 環境変数をチェックし、ファイル名が設定されている場合には、そのファイルを使用します。
2. コマンド行でファイル名が指定されている場合には、そのファイル名を MFCSCFG 環境変数で指定されたファイル名より優先して使用します。
3. どちらにもファイル名が指定されていない場合、デフォルトのファイル名 (**mfclisrv.cfg**) に合致するファイルを現在のディレクトリ内で検索します。
4. **mfclisrv.cfg** が見つからなければ、各設定パラメータのデフォルト値を使用します。

MFCSCFG 環境変数とコマンド行のどちらでも、ファイル名は 128 文字以下の絶対パスで指定します。

## 25.6 クライアントプログラムの mfclient への接続

**mfclient** モジュールと **mfserver** モジュールは、**mfclisrv.cpy** コピーファイルに記述されたパラメータブロックを通じて互いに情報を渡します。 これらのモジュールは、ユーザプログラムに情報を渡すときにも、同じパラメータブロック (LNK-PARAM-BLOCK) を使用します。

ユーザプログラムには、**mfclient** にパラメータを渡すためのコードを追加する必要があります。次にコード例を示します。

```
$SET ANS85

 WORKING-STORAGE SECTION. ......

*--- クライアントプログラムの作業場所節とサーバ
*--- プログラムの連絡節では、必ず mfclisrv.cpy
*--- をインクルードします。

 COPY "MFCLISRV.CPY".

 LINKAGE SECTION. ......

*--- INPUT-REC は、ユーザのクライアントプログラムと
*--- サーバプログラム間のデータ転送に使用される領域
*--- です。このデータ領域のサイズは、CSBIND の設定
*--- ファイル内で指定できます。指定した値は mfclient
*--- モジュールによって読み取られます。mfclient は
*--- 必要なサイズのメモリを INPUT-REC に割り当て、
*--- INPUT-REC を指すポインタをユーザプログラムに
*--- 返します (下記のコードを参照)。

 01 INPUT-REC                PIC X(32767)
```

```
 PROCEDURE DIVISION.

 CLIENT-CONTROL SECTION.

     PERFORM UNTIL END-CONNECTION

*--- LNK-CLIENT には、文字列 mfclient が格納され
*--- ています。最初のループで mfclient を初期化し、
*--- サーバとの接続を確立します。

         CALL LNK-CLIENT USING LNK-PARAM-BLOCK

         EVALUATE TRUE

*--- mfclient から返されたアドレスを割り当てることによって、
*--- ユーザデータ領域 (INPUT-REC) をアクセス可能にします。

          WHEN START-CONNECTION
             SET ADDRESS OF INPUT-REC TO LNK-DBLOCK-PTR

          WHEN END-CONNECTION
             EXIT PERFORM

          WHEN OTHER

*--- アプリケーションのクライアントロジック (ユーザインターフェイス
*--- データの表示や受け入れなど) を実行します。

              ......

         END-EVALUATE

*--- ユーザ定義のフラグ (EXIT-FLG-TRUE など) を TRUE に設定して、
*--- クライアント処理の終了を示します。たとえば、 アプリケーション
*--- インターフェイスの [終了] プッシュボタンをクリックしたときに、
*--- このフラグを立てます。

         IF EXIT-FLG-TRUE
            SET CLIENT-ENDING TO TRUE
         END-IF

     END-PERFORM.

 CLIENT-CONTROL-END.
     STOP RUN.
```

アプリケーションのクライアント数を制御したり、エラーメッセージの表示をプログラム側で処理できるようにするには、ユーザプログラムの最初の EVALUATE 文に次のようなコードを追加する必要があります。

```
           WHEN TOO-MANY-CLIENTS
              PERFORM OVER-CLIENT-LIMIT

           WHEN COMMS-ERROR
              PERFORM SHOW-ERROR

           ......

 OVER-CLIENT-LIMIT SECTION.
     DISPLAY SPACES AT 0101 WITH BACKGROUND-COLOR 7
        "MAXIMUM NUMBER OF CLIENTS EXCEEDED - SESSION ENDED"
        AT 1012 WITH FOREGROUND-COLOR 4
     SET EXIT-FLG-TRUE
     SET CLIENT-ENDING TO TRUE
     EXIT.

SHOW-ERROR SECTION.
     DISPLAY LNK-ERROR-LOC AT 2201
     DISPLAY LNK-ERROR-MSG AT 2301
             WITH SIZE LNK-ERROR-MSG-LEN.
     EXIT.
```

非同期要求を処理するには、EVALUATE 文に次のようなコードを追加する必要があります。

---

**注記**：AAI を使用している場合は、ASYNC-REQUEST は不要です。

---

```
WHEN START-CONNECTION
   PERFORM GET-USER-INPUT
   IF MAKE-ASYNC-REQUEST <* ユーザの非同期オプション
      SET ASYNC-REQUEST TO TRUE
   END-IF

WHEN ASYNC-OK
   SET TEST-ASYNC-RESULT TO TRUE
   PERFORM DELAY-LOOP

WHEN ASYNC-INCOMPLETE
   DISPLAY "REQUEST STILL BEING PROCESSED " AT 1010
```

```
   PERFORM DELAY-LOOP
   SET TEST-ASYNC-RESULT TO TRUE

WHEN RESULT-OK
   DISPLAY "REQUEST-COMPLETED " AT 1010
   PERFORM GET-USER-INPUT

WHEN ASYNC-NOT-STARTED
WHEN ASYNC-FAILED
   DISPLAY "ASYNCHRONOUS REQUEST FAILURE  " AT 1010
   PERFORM SHOW-ERROR
   PERFORM GET-USER-INPUT

WHEN COMMS-ERROR
      PERFORM SHOW-ERROR
```

## 25.7 サーバプログラムの mfserver サーバへの接続

**mfclient** モジュールと **mfserver** モジュールは、**mfclisrv.cpy** コピーファイルに記述されたパラメータブロックを通じて互いに情報を渡します。これらのモジュールは、ユーザプログラムに情報を渡すときにも同じパラメータブロックを使用します。

ユーザのサーバプログラムの連絡節には、この **mfclisrv.cpy** をインクルードする必要があります。また、手続き部の見出しには LNK-PARAM-BLOCK パラメータブロックを含め、**mfserver** からパラメータを受け取れるようにします。

次にサーバコードの一例を示します。

```
 LINKAGE SECTION.

 01 INPUT-REC                 PIC X(32767).

 COPY "MFCLISRV.CPY".*--- LNK-PARAM-BLOCK は mfclisrv.cpy 内のレコード定義です。

 PROCEDURE DIVISION USING LNK-PARAM-BLOCK.

 CONTROLLING SECTION.

*-----------------------------------------------------------*
*  ユーザデータ領域を、LNK-PARAM-BLOCK 内
*  のポインタに関連付けます。
*-----------------------------------------------------------*

     SET ADDRESS OF INPUT-AREA TO LNK-DBLOCK-PTR.
```

```
      EVALUATE TRUE

       WHEN START-CONNECTION
          PERFORM PROGRAM-INITIALIZE

       WHEN OTHER
          PERFORM PROGRAM-BODY

      END-EVALUATE.
      EXIT PROGRAM.

 PROGRAM-INITIALIZE SECTION.

*--- ここにサーバプログラムの初期化コード
*--- (アプリケーションのデータファイルを
*--- 開く処理など) を記述します。

      ......

 PROGRAM-BODY SECTION.

*--- ここにアプリケーションの処理コード
*--- (データファイルの読み取り/書き込み
*--- など) を記述します。

      ......
```

エラーメッセージの表示をユーザプログラム側で処理するには、プログラムの最初の EVALUATE 文に次のようなコードを追加します。

```
           WHEN COMMS-ERROR
                    PERFORM SHOW-ERROR

           ......

SHOW-ERROR SECTION.
      DISPLAY LNK-ERROR-LOC AT 2201
      DISPLAY LNK-ERROR-MSG AT 2301 WITH SIZE LNK-ERROR-MSG-LEN.
      EXIT.
```

## 25.8 CSBIND アプリケーションの実行

**mfserver** モジュールは **.int** コード形式で提供されており、それ自体で実行できるほか、**.gnt** コードに変換することも可能です。

UNIX システムで Micro Focus COBOL 3.2 (またはそれ以降のバージョン) を使用している場合は、この **.int** モジュールを UNIX に移動し、他のプログラムとリンクして実行可能モジュールを作成することができます。

**mfclient** と **mfserver** は、標準の COBOL プログラムとして実行され、どちらも手動で起動します。

CSBIND を実行するには、まずサーバプログラムを起動し、続いてクライアントプログラムを起動します。

UNIX でサーバプログラムを起動するコマンド行は次のとおりです。

```
cobrun mfserver [-b] [-p protocol] [-s server-name] [-v]
```

Windows では、次のコマンド行を使用します。

```
run mfserver [-b] [-p protocol] [-s server-name] [-v]
```

これらのコマンド行について説明します。

| | |
|---|---|
| -b | サーバをバックグラウンドプロセスとして実行します。UNIX システムでは、コマンド行の末尾に & 記号を付ける必要があります。サーバが生成したエラーメッセージは、すべてログファイル (**mfclisrv.log**) に出力されます。 |
| -p *protocol* | 通信プロトコルを指定します。 |
| -s *server-name* | サーバ名としてデフォルト (**mfclisrv**) 以外の名前を使用する場合に、そのサーバ名を指定します。 |
| -v | **mfserver** のバージョン番号を表示します。 |

クライアントプログラムを起動するコマンド行は次のとおりです。

```
run program-name [config-filename]
```

このコマンド行について説明します。

| | |
|---|---|
| *program-name* | ユーザインターフェイスプログラム名を指定します。 |
| *config-filename* | 使用する設定ファイルの名前を指定します。 |

## 25.9 アプリケーションのアニメート処理

クライアントプログラムには、Net Express 標準のアニメート機能を使用して、アニメート処理を施すことができます。

サーバプログラムをアニメートするには、設定ファイル内に **srvanim=y** パラメータを指定します (『*設定ファイルのパラメータ*』を参照してください)。

**srvanim=y** を指定した場合、PC サーバではクライアントがサーバに接続すると、自動的にアニメータが起動します。

UNIX サーバでは、アニメータは **mfserver** を起動した端末で実行されるため、**mfserver** をフォアグラウンドモードで実行する必要があります。ところが、**mfserver** はバックグラウンドプロセスとして実行する方が望ましく、フォアグラウンドモードでは問題が発生する可能性があります。また、**mfserver** を通じてアニメートを実行できるユーザが、一度に 1 ユーザだけに限定されてしまいます。これらの問題を回避するには、**srvanim=x**,*filename* を設定します (*filename* は **touch** コマンドで作成したファイル)。アニメータの出力を表示する端末で **COBANIM_2=animator** を設定し、続いて **anim** *filename* コマンドを実行します。標準入出力に使用する端末では、設定ファイルに **srvanim=x**,*filename* を追加しておけば、通常の方法でアプリケーションを実行できます。

# 25.10 サーバの管理

**mfcsmgr** プログラムを実行して必要なパラメータと値を渡すと、サーバの動作を変更できます。設定できるパラメータは、場所とアクションという 2 つのグループに分類されます。場所グループのパラメータはターゲットサーバの場所を指定し、設定値には **m**、**p**、**s** があります。アクショングループのパラメータはターゲットサーバへの影響を制御し、設定値には **a**、**c**、**o**、**r**、**t** があります。一部のアクションパラメータは、組み合わせて指定することができません。該当するのは **o**、**r**、**t** の 3 つです。

**mfcsmgr** プログラムは、AAI の管理下で動作しているアプリケーションには無効です。

## 25.10.1 mfserver のシャットダウン

生成されたサーバプロセスは、クライアントの正常終了時にクライアントプログラムによって終了されます。 最初に起動したサーバは、対応するクライアントが終了しても動作を継続します。そのため、このサーバは明示的に終了させる必要があります。

## 25.10.2 許可パスワードの管理

アクティブなサーバを誤ってシャットダウンすることがないように、各サーバにはパスワードを割り当てることができます。このパスワードは、サーバを終了したり、サーバのパラメータを変更する前に入力する必要があります。

## 25.10.3 クライアントの最大接続数の設定

サーバがサポートするクライアント数はデフォルトでは 30 ですが、**mfcsmgr** の機能を使用してクライアントの最大数を減らすことができます。 指定した値はパスワードファイルに保存され、サーバが起動するたびに使用されます。

## 25.10.4 サーバパラメータのオーバライド

サーバの *servername*、*protocol*、*machinename* 各パラメータには、優先する値を明示的に指定し、設定済みの値の代わりに使用する (オーバライドする) ことができます。 サーバは設定ファイルを使用しないため、優先するパラメータ値は **mfcsmgr** プログラムで指定します。 このプログラムを実行すると、指定した機能を処理するために、クライアントとサーバがわずかな時間だけ情報を交換します。

**mfcsmgr** のコマンドライン構文を次に示します。

### *Windows の場合*

```
run mfcsmgr [-a] [-c nnnnn] [-d] [-i filename]
            [- m machine-name] [-p protocol] [-s server-name]
            [-o m, machine-name] [-o p, protocol] [-o r]
            [-o s, server-name] [-t] [-v]
```

### *UNIX の場合*

```
cobrun mfcsmgr [-a] [-c nnnnn] [-d] [-i filename]
               [- m machine-name] [-p protocol]
               [-s server-name] [-o m, machine-name]
               [-o p, protocol] [-o r] [-o s, server-name]
               [-t] [-v]
```

上記の構文について次に説明します。

| | |
|---|---|
| -a | ターゲットサーバの認証パスワードを変更します。 |
| -c,*nnnnn* | サーバがサポートするクライアントの最大数を設定します。 |
| -d | -d クライアントの終了時に、オーバライド用のローカルファイル (**mfcsovrd.cfg**) を削除します。 |
| -i *filename* | クライアントが存在する場合に、指定したファイルをクライアントのローカルディレクトリにインストールします。 |
| -m *machine-name* | ターゲットサーバを実行しているマシン名を指定します。 名前のサーバが複数のプラットフォームで動作している場合、この指定は必須です。 |
| -p *protocol* | 使用する通信パラメータ。TCP/IP (CCITC32)、NetBEUI (CCINB32)、Novell IPX (CCIIX32)、動的データ交換 (CCIDE32) のいずれかを指定します。 |

| | |
|---|---|
| -s *server-name* | デフォルト (**mfclisrv**) 以外のサーバ名を指定します。 |
| -o m,*machine-name* | 優先値を使用するサーバが動作しているマシンの名前を指定します。 |
| -o p, *protocol* | 優先値を使用するサーバへのアクセスに使用するプロトコル。 |
| -o r | アクティブな優先パラメータをリセットし、元の設定に戻します。 |
| -o s,*server-name* | ターゲットサーバへの接続を、このサーバへの接続に変更します。 |
| -t | サーバを終了します。 |
| -v | **mfclient** のバージョン番号を表示します。 |

上記の各フラグは、いずれも - または / を付けて指定します。大文字と小文字は区別されません。

# 25.11 高度なトピック

## 25.11.1 システムログファイル

**mfclient** と **mfserver** は、どちらもエラーとメッセージのログを保持します。 すべてのログエントリは、日付と時刻のスタンプを付加され、**mfclisrv.log** ファイルに保存されます。このログファイルはプログラムが起動したディレクトリ、または MFLOGDIR 環境変数で指定されているディレクトリにあります。 このログを定期的にチェックして、システムが正しく動作していることを確認してください。

## 25.11.2 監査トレールの作成

クライアントがサーバに接続した日付と時刻の履歴 (監査トレール) を作成できます。 この機能は、設定ファイルに **useraudit=y** を追加するだけで有効になります。

監査トレールは前項で説明したシステムログファイルに記録されます。

## 25.11.3 設定ファイルのエントリのオーバライド

クライアントの起動時には、設定ファイルの読み取り後に CSBIND によってオーバライドファイル (**mfcsovrd.cfg**) が検索されます。 検索先はシステムログファイルの格納先ディレクトリです。 ファイルが見つかると、設定ファイルで指定された各パラメータが、オーバライドファイルの内容に従ってオーバライドされます。 オーバライドファイルの記述形式は通常の設定ファイルと同じですが、**override-cntrl** というパラメータが 1 つだけ追加されています。 このパラメータにはオーバライドの対象として、サーバ名とタグ名のいずれかを指定します。 サーバ名を指定すると、そのサーバを使用する全クライアントのパラメータが、オーバライドファイルの内容に従って変更されます。 タグ名を指定すると、そのタグ名を使用するクライアントのパラメータだけがオーバライドされます。

クライアント側のパラメータによるオーバライド機能は、サーバが使用不能になり、アプリケーションを他のマシンで実行する必要が生じた場合などに使用します。 設定ファイルを個別に編集することも可能ですが、この機能を使用すれば、1 つのオーバライドファイルだけで、任意の数のアプリケーションを他のサーバでも使用できるようになります。

オーバライド機能はサーバでも使用できますが、サーバマシンが起動済みで動作している必要があります。 この場合も、オーバライドの対象になるのはサーバ名またはタグ名です。

オーバライドファイルが見つかると、その旨がログファイルに記録されます。さらに、オーバライドの対象になるパラメータもすべて記録されます。

オーバライド機能を使用して、クライアントの接続先サーバを変更する設定の一例を次に示します。

```
[override-cntrl]
override=servername
[oldserver]
servername=newserver
```

この例では、**[override-cntrl]** セクションでサーバ名をオーバライド対象 (**override=servername**) として指定し、**[oldserver]** で元のサーバ名、**servername=newserver** で新しいサーバ名をそれぞれ指定しています。 この結果、ログファイルには次のようなエントリが記録されます。

```
20/04/1998 11:01:02 Using Local File: mfcsovrd.cfg
Overriding Entries for Servername:OLDSERVER
servername=newserver
20/04/1998 11:01:02 Override Completed:
```

次に、指定したタグを使用するすべてのクライアントを対象に、サーバを変更する設定の一例を示します。

```
[override-cntrl]
override=tagname
[mf-clisrv]
servername=newserv
```

この例では、**[override=cntrl]** セクションでタグ名をオーバライド対象 (**override=tagname**) として指定し、続いてタグ名 (**[mf-clisrv]**)、その下にオーバライドするパラメータ (**servername=newserv**) を指定しています (この場合、オーバーライドするパラメータはサーバ名です)。 この結果、ログファイルには次のようなエントリが記録されます。

```
20/04/1998 11:04:02 Using Local File: mfcsovrd.cfg
Overriding Entries for Tagname:MF-CLISRV
servername=newserv
20/04/1998 11:04:02 Override Completed:
```

### 25.11.4 Dialog System のコールアウトの使用

CSBIND で Dialog System のコールアウト機能を使用するには、接続のいずれか一端でコールアウト要求に対応するように、ユーザクライアントプログラムに多少の変更を加えする必要があります。具体的には、ユーザクライアントプログラムに以下の 2 行と指令を追加します。

プログラムの冒頭に次の行を追加します。

```
$set ans85 linkcount(128)
```

作業場所節に次の行を追加します。

```
 01  lnk-param-block-addr          POINTER EXTERNAL.
```

手続き部に次の指令を追加します。

```
     SET lnk-param-block-addr
              TO ADDRESS OF lnk-param-block
```

これらの行を追加するのは、ダイアログによって呼び出され、コールアウト要求を処理するプログラムが、ユーザクライアントプログラムと同じパラメータエントリにアクセスできるようにするためです。その実現には、外部データポインタが使用されます。コールアウトプログラムには、次の例のようなコードが必要です。

```
$set ans85
 WORKING-STORAGE SECTION.

 01  lnk-param-block-addr          POINTER EXTERNAL.

 LINKAGE SECTION.

 COPY "mfclisrv.cpy".

 01  user-data-block              PIC X(30).

 PROCEDURE DIVISION.
 Controlling SECTION.
     SET ADDRESS OF lnk-param-block TO lnk-param-block-addr
     SET ADDRESS OF user-data-block TO lnk-ublock-ptr
     SET ds-callout TO TRUE
     MOVE "callosrv" TO user-data-block
     CALL lnk-client USING lnk-param-block
     EXIT PROGRAM.
```

サーバでコールアウト要求を受信するプログラムの名前は、user-data-block で設定されます。このプログラムの名前が、Dialog System で使用されるいずれかの制御ブロック内のデータの

みで特定される場合には、次のプログラムのようなコードを使用して、該当するブロックへのアクセスを可能にする必要があります。

```
$set ans85 linkcount(128)
 WORKING-STORAGE SECTION.

 01  lnk-param-block-addr         POINTER EXTERNAL.

 LINKAGE SECTION.

 COPY "mfclisrv.cpy".

 01  user-data-block              PIC X(30).

     COPY "DS-CNTRL.MF".
     COPY "CUSTOMER.CPB".

 PROCEDURE DIVISION USING ds-control-block customer-data-block.
 Controlling SECTION.
     SET ADDRESS OF lnk-param-block TO lnk-param-block-addr
     SET ADDRESS OF user-data-block TO lnk-ublock-ptr
     SET ds-callout TO TRUE
     EVALUATE cust-callout-flg
      WHEN 1
         MOVE "callprog1" TO user-data-block
      WHEN 2
         MOVE "callprog2" TO user-data-block
     END-EVALUATE
     CALL lnk-client USING lnk-param-block
     EXIT PROGRAM.
```

サーバ側のコールアウトプログラムは、次のようなコードになります。

```
$set ans85

 LINKAGE SECTION.

     COPY "DS-CNTRL.MF".
     COPY "CUSTOMER.CPB".

 PROCEDURE DIVISION USING ds-control-block customer-data-block.
 Controlling SECTION.
       ..... * コールアウト
           ..... * 要求を処理
               ..... * するコード
           EXIT PROGRAM.
```

### 25.11.5 Dialog System の既存プログラムのサーバモジュールとしての使用

CCI とともに使用する場合、CSBIND では、既存の Dialog System プログラムをユーザサーバモジュールとして使用することができます。この機能を有効にするには、srvtier=<*your program*> を指定します (詳細については、前述した『*設定ファイルのパラメータ*』を参照してください)。

この機能では、**mfserver** を Dialog System の API に準拠するパラメータセットで呼び出す必要があります。ユーザプログラムは **mfserver** を呼び出してクライアントにデータを転送し、サーバの制御要素になります。この機能で使用できるユーザプログラムは、Dialog System プログラムだけに限定されません。Dialog System 以外のプログラムも、Dialog System で使用される API に準拠すれば、同じ用途に使用できます。

CSBIND は通常、API の最初のパラメータに含まれる特定項目をチェックし、見つかった値に基づいて特定の機能を実行します。この動作を無効化するには、該当する項目を HIGH-VALUES に設定します。**mfserver** は、次の 2 つのパラメータで呼び出す必要があります。

- 最初のパラメータ ― ダイアログ制御ブロックを表す値。長さは 6～399 バイト。6 番目のバイトがチェックされるため、このバイトを HIGH-VALUES に設定し、Dialog System への呼び出しを回避します。このパラメータの他の部分は、必要に応じて使用または無視できます。
- 2 番目のパラメータ ― 任意の長さのユーザデータブロック。

これらの項目のサイズは、どちらもアプリケーションに使用する設定ファイル内で定義します。プログラムと必要な設定ファイルの例を次に示します。

- o nodscli.cbl ― クライアントモジュール
- ```
  $set ans85
  WORKING-STORAGE SECTION.
  01  w-prog-id                pic x(30) value
       '"@(#) nodscli.cbl 1.1.1"'.

   copy "mfclisrv.cpy".

  01  ws-scrn-pos.
      03  ws-line              pic 99 value 07.
      03  ws-col               pic 99 value 1.

  LINKAGE SECTION.
  01  ds-control-block         pic x(6).

  01  NODS-DATA-BLOCK.
      03  NODS-PARAM           PIC 99.
      03  NODS-DATA            PIC X(120).
  ```
  ```

```
    procedure division.
    000-control section.
    * 最初にサーバに接続
        call lnk-client using lnk-param-block
        if start-connection
            set address of ds-control-block to lnk-cblock-ptr
            set address of nods-data-block to lnk-dblock-ptr
        end-if

        display spaces at 0101
                "Starting Test Session" at 0510
        perform 010-run-test until nods-param = 10
    * 切断要求を送信
        set client-ending to true
             call lnk-client using lnk-param-block
        display "Test Session Completed" at 2010
        stop run.

    010-run-test section.
    * サーバからデータを取得
        call lnk-client using lnk-param-block
        add 1 to ws-line
        display nods-data(1:80) at ws-scrn-pos
        exit.
```

nodssrv.cbl ― サーバモジュール

```
  $set ans85
    working-storage section.
    01  w-prog-id                pic x(30) value
        '"@(#) nodssrv.cbl 1.1.1"'.

    01  dummy-cntrl-block        pic x(6) value high-values.

    01  NODS-DATA-BLOCK.
       03  NODS-PARAM           PIC 99.
       03  NODS-DATA            PIC X(120).

        copy "mfclisrv.cpy".

    01  ws-sub                   pic 99.
    01  ws-table.
       03  ws-letts              pic x occurs 10.

    procedure division.
    controlling section.
```

```
*
        move "ABCDEFGHIJ" to ws-table
        move 0 to nods-param
    *
    * クライアントとの接続を確立
    * (クライアントのタイムアウトや、サーバ検出を目的としたネットワーク
    * ポーリングを回避するためにも、この処理はできる限り早い段階で実行
    * する必要があります。)
    *
        call "mfserver" using dummy-cntrl-block nods-data-block
    *
    * mfserver の呼び出しは、クライアントがシャットダウン要求を送信
    * した後で実行してください。 このシステムでは、nods-param の値
    * が 10 になるとクライアントがシャットダウン要求を送信し、この
    * プログラム内のループを抜けます。
    *
        perform varying nods-param from 1 by 1
                until nods-param > 10
            perform varying ws-sub from 1 by 1
                    until ws-sub > 80
                move ws-letts(nods-param)
                            to nods-data(ws-sub:1)
            end-perform
            call "mfserver" using dummy-cntrl-block
                                  nods-data-block
        end-perform
        stop run.
```

nods.cfg ― 設定ファイル

```
    ************************************************************
    * Micro Focus - クライアントサーバモジュール設定ファイル       *
    ************************************************************

    [mf-clisrv]
    srvtier=nodssrv
    cblksize=6
    dblksize=122
    servername=nods
```

サーバプログラムが呼び出される側のプログラムで、クライアントにデータを渡すたびに EXIT PROGRAM を使用する場合とは異なり、上記のサーバプログラムは、動作を継続しながらクライアントにデータを渡します。既存のプログラムが Dialog System API に準拠できない場合でも、次のような短いブリッジプログラムを介在させれば、上記の方法を使用できます。

```
 WORKING-STORAGE SECTION.
```

```
01  dummy-control-block          PIC X(6) VALUE HIGH-VALUES.
01  grouped-data-block.
  03  ws-param-1                PIC X(50).
   03  ws-param-2                PIC X(120).
   03  ws-param-3                PIC X(18).
   03  ws-param-4                PIC X(48).

LINKAGE SECTION.

01  lnk-param-1                  PIC X(50).
01  lnk-param-2                  PIC X(120).
01  lnk-param-3                  PIC X(18).
01  lnk-param-4                  PIC X(48).

PROCEDURE DIVISION USING
        lnk-param-1 lnk-param-2
        lnk-param-3 lnk-param-4.

    MOVE lnk-param-1 TO ws-param-1
    MOVE lnk-param-2 TO ws-param-2
    MOVE lnk-param-3 TO ws-param-3
    MOVE lnk-param-4 TO ws-param-4

    CALL "mfserver" USING dummy-control-block
                          grouped-data-block

    MOVE ws-param-1 TO lnk-param-1
    MOVE ws-param-2 TO lnk-param-2
    MOVE ws-param-3 TO lnk-param-3
    MOVE ws-param-4 TO lnk-param-4

    EXIT PROGRAM.
```

### 25.11.6 インライン設定機能の使用

CSBIND では、通信条件を設定ファイルで制御することができます。これは CSBIND の最も優れた特長の 1 つですが、設定ファイルのエントリを変更すれば、エンドユーザがアプリケーションの動作に影響を与えうるという点で、望ましくない場合があります。

設定パラメータは、すべてクライアントプログラム内で指定することも可能です。この方法はインライン設定と呼ばれます。インライン設定では設定ファイルは使用されないため、エンドユーザがアプリケーションの動作に影響を与える可能性はありません。また、この場合も複雑な通信コードの記述は不要で、必要なパラメータをクライアントプログラム内に記述するだけです。インライン設定では、**load-inline-cfg** と **end-inline-cfg** の間に設定パラメータを指定します。指定した各パラメータは **lnk-error-msg** フィールドにロードされ、**mfclient** が呼び出さ

れて処理されます。設定パラメータは、処理がメインループに入る前に指定する必要があります。インライン設定の一例を次に示します。

インライン設定は、AAI を使用する場合にはサポートされません。

```
 WORKING-STORAGE SECTION.

  COPY "mfclisrv.cpy".

 LINKAGE SECTION.

 01 INPUT-REC                PIC X(32767)

 PROCEDURE DIVISION.

 Client-Control SECTION.

     SET load-inline-cfg TO TRUE
     MOVE "clierrprog=same" TO lnk-error-msg
     CALL lnk-client USING lnk-param-block

     MOVE "srverrprog=same" TO lnk-error-msg
     CALL lnk-client USING lnk-param-block

     MOVE "servername=mainserv" TO lnk-error-msg
     CALL lnk-client USING lnk-param-block

     SET end-inline-cfg TO TRUE

*   メインループの処理は、サーバとの接続が切断
*   されるまで継続します。

     PERFORM UNTIL End-Connection

* lnk-client には、文字列 mfclient が格納
*   されています。最初のループでシステムを
*   初期化し、サーバとの接続を確立します。

     CALL lnk-client USING lnk-param-block

     EVALUATE TRUE

      WHEN start-connection

      ........ * これ以降のコードは、
      ........... * 標準的な内容です。
```

外部設定ファイルとインライン設定を組み合わせて使用することもできます。この方法では、外部設定ファイルとインライン設定のそれぞれの長所が活かされます。すなわち、エンドユーザによるシステムの動作の制御を可能にしながら、最終的な動作はクライアントプログラムで制御できます。設定ファイルとインライン設定では、前者が先に処理されるため、設定ファイルで望ましくないパラメータが設定されても、インライン設定でオーバーライドすることが可能です。設定パラメータの読み取りは **use-combined-cfg** で開始され、**end-inline-cfg** で終了します。

```
 WORKING-STORAGE SECTION.

 COPY "mfclisrv.cpy".

 LINKAGE SECTION.

 01 INPUT-REC               PIC X(32767)

 PROCEDURE DIVISION.

 Client-Control SECTION.

     SET use-combined-cfg TO TRUE
     CALL lnk-client USING lnk-param-block

     SET load-inline-cfg TO TRUE
     MOVE "servername=mainserv" TO lnk-error-msg
          CALL lnk-client USING lnk-param-block

     SET end-inline-cfg TO TRUE

*   メインループの処理は、サーバとの接続が切断
*   されるまで継続します。

     PERFORM UNTIL End-Connection

* lnk-client には、文字列 mfclient が格納
*   されています。最初のループでシステムを
*   初期化し、サーバとの接続を確立します。

     CALL lnk-client USING lnk-param-block

     EVALUATE TRUE
      WHEN start-connection

      ........ * これ以降のコードは、
      ........... * 標準的な内容です。
```

### 25.11.7 縮小データ転送機能

CSBIND では、設定ファイルで指定されたデータブロックを確実に格納できるサイズのバッファが割り当てられます。クライアントプログラムからサーバプログラムに (または逆方向に) 制御が移動するときには、このバッファが常にプログラム間で転送されますが、この転送によってネットワークに許容範囲を超える負荷がかかる場合があります。たとえば、22 KB のレコード領域を格納した 24 KB のバッファが割り当てられている場合、ファイルのレコードキーの長さが 10 バイトであれば、クライアントとサーバ間でやり取りされるキーが 10 バイトであるにもかかわらず、24K のバッファが使用されます。実際の転送に必要なメモリのサイズは、データサイズより 1〜2 バイト大きくなりますが、それを勘案してもバッファサイズは大きすぎます。そこで、ネットワーク上を転送されるデータ量を制御し、最小限に抑制する手段として、縮小データ転送 (RDT) 機能が提供されています。なお、CSBIND で AAI を使用する場合には、RDT 機能は使用できません。

RDT 機能を使用するには、制御フラグ (*use-rdt*) と次の 3 つのパラメータが必要です。

| | |
|---|---|
| *lnk-usr-fcode* | ユーザファンクション番号。受信側にこのデータの処理方法を示します。 |
| *lnk-usr-retcode* | バッファの転送開始ポイント。転送の開始ポイントとして、4 つのデータ領域のいずれか 1 つを指定します。有効な値は、1 (cblock)、2 (dblock)、3 (eblk)、4 (ublk)。同じアドレス領域は 11〜14 の数字でも指定できますが、これらの数字は同時にデータ圧縮を示します。データ圧縮を使用するには、設定ファイルでデータ圧縮を有効化しておく必要があります。データ圧縮が無効な場合は、デフォルトオプション (圧縮なし) が使用されます。また、0 と指定するとデータ転送は実行されません。0 を指定すれば、IF 文を多用してコードの制御フローを変更することなく NULL 操作を選択できるため、特定の機能をローカルで処理し、ネットワークトラフィックの増加を防ぐ手段として効果的です。 |
| *lnk-data-length* | 転送するデータのサイズ。 |

ここでは RDT 機能の活用例として、索引ファイルを使用してレコード (顧客詳細情報) の追加、削除、読み取りを行うアプリケーションを想定します。索引ファイルのレコードキーは顧客コードです。このアプリケーションには、顧客レコードの操作だけでなく、顧客情報をインターフェイス画面から消去する機能もあります。このアプリケーションのコードの一部を次に示します。*user-data-block* 領域には、6 バイトのレコードキーが格納されます。クライアントとサーバ間の顧客詳細情報レコードの交換には、データブロック領域 (*customer-data-block*) が使用されます。この領域のサイズは *dblksize* です。*customer-data-block* 内の 6 バイトのデータ項目 *customer-c-code* にレコードキーが格納されます。

クライアント側のコードは次のようになります。

```
EVALUATE TRUE

 WHEN customer-load-flg-true
```

```
*--- ユーザがインターフェイスで顧客コードを入力して Load オプション
*--- を選択したときに、指定されたコードに関連する顧客の詳細情報を読み
*--- 取って表示する処理

           MOVE customer-c-code TO user-data-block
           SET use-rdt TO TRUE
           MOVE 1 TO lnk-usr-fcode
           MOVE 4 TO lnk-usr-retcode
           MOVE 6 TO lnk-data-length

         WHEN customer-del-flg-true

*--- ユーザが顧客コードを入力して DELETE オプションを選択したときに
*--- 顧客レコードをファイルから削除する処理

           MOVE customer-c-code TO user-data-block
           SET use-rdt TO TRUE
           MOVE 2 TO lnk-usr-fcode
           MOVE 4 TO lnk-usr-retcode
           MOVE 6 TO lnk-data-length
           initialize customer-data-block

         WHEN customer-clr-flg-true

*--- ユーザが CLEAR オプションを選択したときに、表示中の
*--- 顧客情報を画面から消去する処理

           SET use-rdt TO TRUE
           MOVE 0 TO lnk-usr-retcode
           initialize customer-data-block
           PERFORM Set-Up-For-Refresh-Screen
       END-EVALUATE
```

画面消去はローカルで完結し、サーバに接続する必要がないため、CLEAR オプションが選択されたときには NULL 操作を使用します。続いて、RDT 機能に必要な処理を含むサーバプログラムのコード例の一部を示します。CSBIND で *send-via-rdt* フラグが設定され、*lnk-usr-fcode* の値がチェック可能になります。サーバ側のコードは次のようになります。

```
         WHEN send-via-rdt
           EVALUATE lnk-usr-fcode
            WHEN 1

*--- LOAD 機能が実行されると、サーバプログラムはデータ
*--- ファイルから顧客情報を読み取り、customer-data-block
*--- データ領域に格納して、RDT 機能を使用せずにクライアント
```

```
*--- に返送します。   RDT フラグが設定されていない限り、
*--- CSBIND はクライアントとサーバの間で、常に設定ファイル
*--- の dblksize で指定されるサイズのデータ領域全体を転送
*--- します。
               MOVE user-data-block TO customer-c-code
               SET customer-load-flg-true TO TRUE
               PERFORM ....... rest of program.......
            WHEN 2
               MOVE user-data-block TO customer-c-code
               SET customer-del-flg-true TO TRUE
               PERFORM ....... rest of program.......
          END-EVALUATE
```

標準で付属しているサンプルアプリケーション (**Csbind**) では、RDT 機能を使用しています。

## 25.11.8 サーバ制御のファイル管理機能

クライアント/ サーバソリューションには、特にクライアント数が増加した場合に、クライアントプログラムの更新やその他の変更が煩雑になりがちだという問題があります。

オーバライド機能 (『*設定ファイルのエントリのオーバライド*』を参照) を使用すれば、サーバの保守作業中にクライアントの設定ファイルを個別に変更することなく、クライアントアプリケーションの接続先サーバを切り替えることができます。オーバライド機能はローカル、リモートのどちらでも実行できますが、ローカルのオーバライドファイルをクライアントごとにインストールしたり削除する作業には手間がかかります。

**mfcsmgr** プログラムを使用すると、-i オプションまたは -d オプションを使用して、クライアントにオーバライドファイルをインストールまたは削除できます (『*サーバの管理*』を参照)。インストールや削除は、指定したクライアントプログラムの終了時に実行されます。

**mfcsmgr** プログラムで -i オプションを使用してインストールできるのは、オーバライドファイルだけではありません。実際には、あらゆる種類のファイルをクライアントシステム (クライアントのローカルディレクトリ) にインストールすることができます。したがって、この方法で新しいスクリーンセットやプログラムファイルをインストールすれば、各クライアントに更新データを一元的に配布することが可能になります。一方、削除オプションの機能は、セキュリティ上の理由から限定されており、削除できるのはオーバライドファイル (**mfcsovrd.cfg**) だけです。

-i オプションを使用してインストールするファイルは、サーバ上に位置している必要があります。インストールするファイルが **mfserver** を起動したディレクトリ内にある場合は、ファイル名だけを指定してインストールできます。その他のディレクトリにある場合は、ファイル名が抽出可能な完全パス名で指定する必要があります。具体的には、/u/live/update/newprog.int、d:\testprog.int、$LIVE/newtest.int などが有効で、$newfile は無効です。

この機能は、プログラムに変更を加えずに使用できます。クライアントにはファイルが転送されたことを示すメッセージが表示され、システムログファイルに詳しい情報が記録されます。

**mfcsmgr** の他の機能と同様、サーバ制御のファイル管理機能は AAI ではサポートされていません。

## 25.12 CSBIND 使用上の留意点

CSBIND には制限事項はほとんどありませんが、次の点に注意してください。

- CSBIND モジュールは **.int** (または **.gnt**) 形式で提供されており、『アプリケーションのディプロイ可能なプラットフォーム』で定義されている各プラットフォームで実行できます。UNIX プラットフォームでは、CSBIND モジュールをユーザ側で作成したアプリケーションプログラムとリンクして、実行可能オブジェクトを作成することもできます。
- 並行して接続可能なクライアントの最大数は 30 です。ただし、サーバの処理能力やネットワークプロトコル、エンドユーザが必要とするパフォーマンスなどの要因によって、この数が抑えられる場合もあります。 たとえば、UNIX では **mfserver** が生成できるサブプロセスの数によって、最大数が制限されます。 生成可能なサブプロセスの数は、ユーザが UNIX マシンにログインするときに割り当てられる一意のプロセス ID で左右されます。 CSBIND を通じて接続したクライアントは、常にベースサーバのサブプロセスになります。ただし、サブプロセスの許容数を変更したり、複数のベースサーバを実行することによって、この制限は緩和できます。 数百人に及ぶユーザが並行して直接ログインできるように設定されている UNIX マシンは、サブプロセス数の制限を緩和した場合に、ユーザの最大数と同じ数のクライアントをサポートできる必要があります。
- CSBIND にはデータ回復機能がないため、 ネットワークがダウンするとデータが失われます。 クライアント/サーバの両端で障害の発生が検出され、ログに記録されますが、データは回復できません。 この問題は、接続のいずれか一端のユーザプログラムを終了させる RTS エラー全般に共通しており、 この点に関する CSBIND の機能は、標準的な RTS を超えるものではありません。

---



---

# 第 26 章 : CGI ベースのインターネットプログラミングの概要

この章では、CGI ベースのインターネットアプリケーションの概要と、その構成要素について説明します。

## 1.1 概要

インターネットやイントラネット向けのアプリケーションの開発方法として、最も古くから支持されてきた手法の 1 つがサーバ側プログラムです。 また、Interface Mapper Toolkit やクラスウィザードとメソッドウィザードで生成されるような、コンポーネントベースのサービスを使用する手法も、有力な選択肢の 1 つです。 両者の特徴と違いについては、『はじめに』の章を参照してください。.

サーバ側プログラムの仕組みは、共通ゲートウェイインターフェイス (Common Gateway Interface) と呼ばれる通信規格として広く普及しており、サーバ側プログラム自体も通常は CGI プログラム、あるいは CGI と呼ばれます。

『***分散コンピューティング***』のこのパートでは、このサーバ側プログラムによるインターネットやイントラネット向けのアプリケーションの作成について説明します。

インターネットアプリケーションは、標準的なインターネットプロトコルでクライアントをサーバに接続するクライアント/サーバ型のアプリケーションであり、Web で公開されているインターネットアプリケーションや、社内で運用されているイントラネットアプリケーションも、まったく同じ方法で構築できます。イントラネットアプリケーションとは、組織内のイントラネットで運用されるタイプのインターネットアプリケーションであり、利用者も組織内のスタッフに限定されます。なお、このマニュアルで言及する「インターネットアプリケーション」には、インターネット上で運用されている本来の意味でのインターネットアプリケーションのほか、イントラネットアプリケーションも含まれます。

インターネットアプリケーションは、"Thin client" と呼ばれる軽量のクライアント側プログラムと、"Thick Server" と呼ばれる処理部のサーバ側プログラムで構成されます。この構成において、クライアント側はエンドユーザへの表示と、エンドユーザとの対話を行い、その役割はユーザインターフェイスに限定されます。クライアントは、インターネットにアクセスするための標準的なツールである Web ブラウザで実行されます。すべての処理はサーバ側で行われ、ここに組織のデータも置かれます。

インターネットアプリケーションは、クライアントとサーバ間の通信に標準的なインターネットプロトコルを使用するため、クロスプラットフォーム環境で運用可能です。Micro Focus COBOL で作成したサーバ側プログラムは Windows NT や UNIX のサーバで実できます (ただし、UNIX でアプリケーションを実行するには、Micro Focus COBOL for UNIX を購入する必要があります)。

インターネットアプリケーションのサーバ側プログラムは、同じマシンで動作している Web サーバソフトウェアを介してクライアントと通信します。COBOL のサーバ側プログラムと、それを実行する Web サーバ間のインターフェイスを明示的にコーディングする必要はありません。次に挙げる 3 種類の業界標準 Web サーバインターフェイスのいずれかを、コードを変更することなく利用できます。

- CGI (Common Gateway Interface)
- ISAPI (Internet Server Application Program Interface)
- NSAPI (Netscape Server Application Program Interface)

これらのインターフェイスについては、『CGI、ISAPI、および NSAPI プログラム』の章で詳しく説明します。 Net Express のインターネットアプリケーションウィザードでは、ビルドするアプリケーションで使用するサーバ側プログラムとして、CGI (すべての Web サーバに対応) と ISAPI (Microsoft IIS に最適化) のいずれかを選択できます。 開発とデバッグには CGI を使用することが推奨されます。COBOL CGI プログラムとしてビルドしたプログラムは、Net Express のコンパイラ設定とビルド設定を変更すれば、それ以降は ISAPI プログラムまたは NSAPI プログラムとしてリビルドできます。

クライアント側のユーザインターフェイスは、次の構成要素をさまざまに組み合わせて作成できます。

- 標準的な HTML フォーム。フォームをサポートするあらゆる Web ブラウザで使用可能なため、Windows、UNIX、Macintosh、および OS/2 のエンドユーザプラットフォームでディプロイできます。
- ActiveX コントロール。ActiveX 対応の Web ブラウザで使用できます (ActiveX を使用できるのは、現時点で 32 ビットの Windows プラットフォームのみ)。
- Java アプレット。Java 対応のあらゆる Web ブラウザで使用できます (Internet Explorer と Netscape Navigator を含む)。

詳細については、『フォームと HTML』の章を参照してください。

Form Designer では、クライアント側の機能を拡張するスクリプトも容易に作成できます。共通の検証関数を実行したり (『フォームの検証』の章を参照)、スクリプトアシスタントを使用してスクリプトを追加することができます (『クライアント側のプログラミング』の章を参照)。

# 1.2 インターネットアプリケーションの内容

インターネットアプリケーションは、クライアント/サーバ型アプリケーションであり、次の 2 つの構成要素に分けられます。

- フォーム
- サーバ側プログラム

フォームは、エンドユーザ側に表示される部分です。フォームは Web ブラウザに表示され、エンドユーザがデータを入力するためのコントロール類があります。次の図は、フォームの一例を示しています。

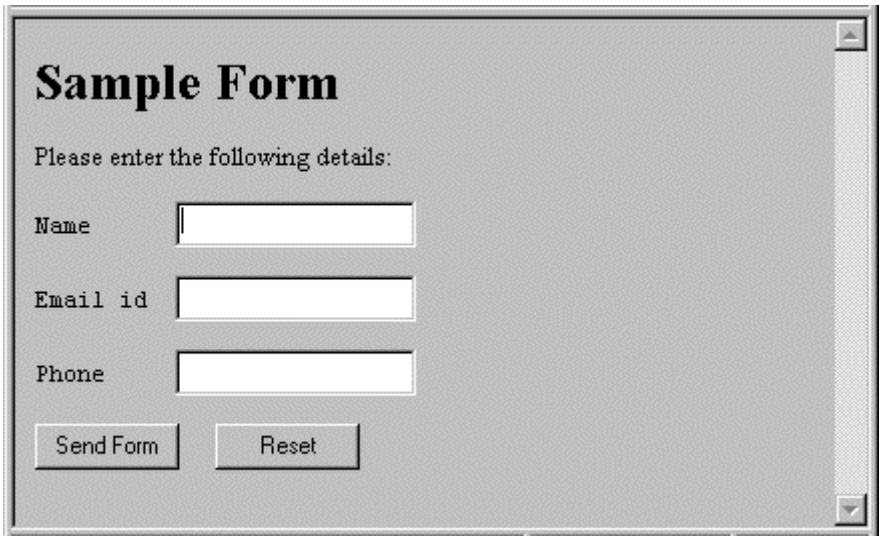



**図 26-1 インターネットアプリケーションのフォームの一例**

このフォームでエンドユーザが [**Send Form**] ボタンをクリックすると、フォームに入力された情報が連結され、サーバ側プログラムに送信されます。サーバ側プログラムは、フォームや Web ページ内のリンクで呼び出された場合のみ動作します。サーバ側プログラムは、フォームの情報を処理し、エンドユーザにページを返します。

プログラムの処理内容に応じて、結果がフォームまたはテキスト形式で表示されます。

### 1.2.1 より複雑なアプリケーション

前述したアプリケーションは、非常にシンプルな例に過ぎません。実際のアプリケーションは、より複雑になる場合がほとんどであり、互いにリンクされた複数のフォームとサーバ側プログラムで構成される場合もあります。サーバ側プログラムは、次の 2 種類に分類されます。

- 対称型

  対称型のサーバ側プログラムは、入力と出力に同じフォームを使用します。たとえば、対称型ののデータベース照会/更新プログラムでは、レコード入力や SQL 照会用のフィールドセットが表示され、これらのフィールドに照会データを入力すると、プログラムの処理結果が同じフィールドに表示されます。

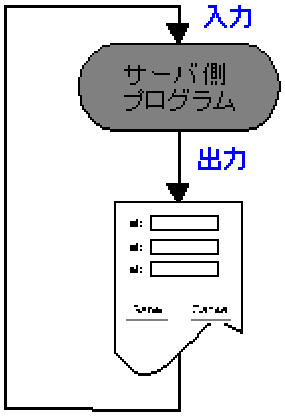



**図 26-2 対称型のサーバ側プログラム**

- **非対称型**

  **非対称型のサーバ側プログラムは、入力と出力に異なるフォームを使用します。たとえば、最初のフォームに顧客の詳細情報を入力すると、新規注文用の別のフォームが表示される注文入力プログラムなどがあります。**

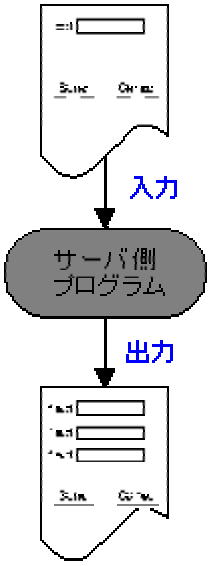



**図 26-3 非対称型のサーバ側プログラム**

**非対称型のサーバ側プログラムでは、異なるフォームとサーバ側プログラムを連鎖させて、複雑なアプリケーションを構築することが可能です。次に示すアプリケーションでは、最初のプログラムの出力を受けて 2 番目のサーバ側プログラムが起動し、プログラム内部の処理ロジックに応じて異なるフォームが出力されます。**

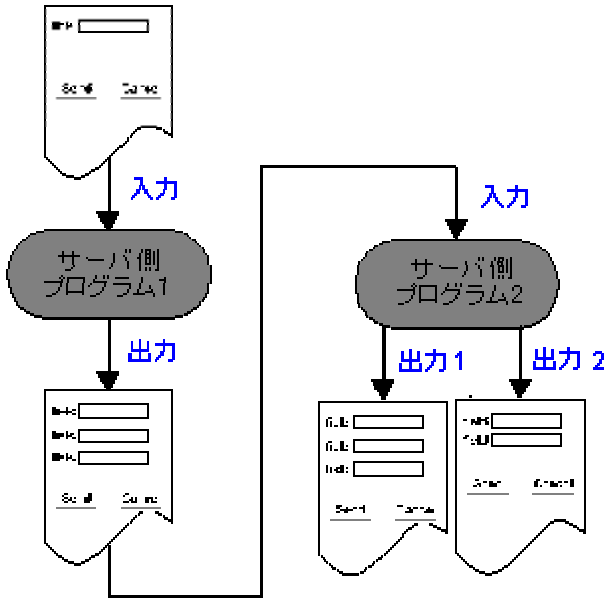



**図 26-4 より複雑なアプリケーション**

---

**注記:**

Web ブラウザに表示されるのは HTML ページです。1 つの HTML ページは複数のフォームを含む場合もありますが、サーバ側プログラムが受信できるのは 1 つのフォームからの入力だけです。サーバ側プログラムは Web ブラウザにページを返します。このページにも複数のフォームを含めることができます。

サーバ側プログラムへの入力の基本単位は常にフォーム、出力の基本単位は常にページです。ほとんどの場合、ページに含まれるフォームは 1 つだけであり、アプリケーションの作成中にはページよりフォームでの作業が多くなるため、上図ではフォームだけを示しています。

---

## 1.3 実行フロー

インターネットアプリケーションは、従来の CICS アプリケーションに似た性格を持っています。フォームを BMS 入力画面に、サーバ側プログラムを CICS トランザクションに置き換えて考えれば、両者の制御フローは非常に良く似ています。CICS トランザクションと同様、サーバ側プログラムが動作するのは、データを処理して結果を返すまでの間だけです。複雑な CICS アプリケーションは、複数の入力画面と複数のトランザクションで構成されます。エンドユーザからはアプリケーションが常に実行されているように見えますが、実際のアプリケーション内では、短時間だけ実行されて終了するトランザクションが連続的に実行されています。

対称型のサーバ側プログラムに基づくアプリケーションの主な処理の流れを次に示します (対称型サーバ側プログラムの定義については、前項を参照してください)。

1. エンドユーザが、サーバ側プログラムを起動するリンクを Web ページでクリックします。
2. サーバ側プログラムが実行され、アプリケーションのフォームを含む HTML ページを返します。
3. エンドユーザがフォームに情報を入力し、ボタンをクリックして送信します。

   この操作によって、サーバ側プログラムが再実行されます。

4. サーバ側プログラムがデータを取得し、フォームとしてエンドユーザのブラウザに返します。

これ以外にも、多様な処理の流れが考えられます。たとえば、Net Express に付属している CGI のサンプルアプリケーション (**Net Express\base\demo\cgi\cgiprg1.app** に格納されています) では、最初に入力フォームが表示され、このフォームを送信すると簡単な HTML ページを返す CGI プログラムが実行されます。前項で簡単に説明した非対称型のサーバ側プログラムでは、処理フローはいっそう複雑になりがちです。ただし、その場合も基本的な特徴、すなわちサーバ側プログラムが結果を返すまでの短い間しか実行されないという点は変わりません。

---



---

# 第 27 章 : CGI ベースのインターネットアプリケーションの開発

この章では、CGI ベースのインターネットアプリケーションの開発手順の概要と、Net Express のツールを活用して、インターネットアプリケーションを速やかに開発する方法について説明します。

## 2.1 概要

Net Express を導入すると、ただちにインターネットアプリケーションを作成できます。
Net Express には、次の各作業に役立つツール群が含まれています。

- HTML ページとサーバ側プログラムの作成
- インターネットアプリケーションのサブルーチンとレガシーコードの再利用
- アプリケーションのデバッグとテスト

Net Express では、次の 3 通りのアプローチでインターネットアプリケーションを開発できます。

- Form Designer とインターネットアプリケーションウィザードを使用して、アプリケーションを新規に開発する (『CGI ベースのアプリケーションの新規作成』の章で説明しています)。
- インターネットアプリケーションウィザードを使用して、既存のサブルーチンを再利用する (『レガシーコードの再利用』の章で説明しています)。
- インターネットアプリケーションウィザードを使用して、SQL データベースにアクセスするアプリケーションを開発する (『CGI ベースのデータアクセスアプリケーション』の章で説明しています)。

## 2.2 支援ツール

この項では、インターネットアプリケーションの作成を支援する Net Express の 2 つのツールについて簡単に紹介します。

### 2.2.1 HTML ページの作成

Form Designer を使用すると、インターネットアプリケーションのユーザインターフェイスを作成できます。Form Designer は、WYSIWYG の HTML エディタと HTML フォームペインタです。Form Designer では、標準的な HTML フォームコントロール、ActiveX コントロール、および Java アプレットを使用できます。

Form Designer では、ページの各 HTML フォームコントロールに COBOL データ型を指定できます。ページ内の各コントロールのデータ項目は、アプリケーションのサーバプログラムを作

成するときに宣言されます。Form Designer の基礎知識については、『入門書 - その他のトピック』のサンプルを参照してください。

Form Designer でインターネットアプリケーションを作成する手順を次に示します。

1. NetExpress のプロジェクトを作成します。アプリケーションのすべてのファイルが、この中に格納されます。
2. Form Designer を起動し、作成する HTML ページの種類を入力して、ページ内にフォームをペイント (視覚的に作成) します。既存の HTML ページをテンプレートとして使用することもできます。
3. インターネットアプリケーションウィザードを使用して、サーバ側のプログラムのひな型 (スケルトン) を作成します。
4. Net Express の IDE でサーバ側プログラムを編集し、機能を追加します。
5. プログラムをビルドします。
6. Solo Web サーバでアプリケーションのテストとデバッグを行います (アプリケーションのテストとデバッグには Solo 以外の Web サーバも使用できますが、Web サーバの設定や使用方法を十分に把握していない場合、Solo は使いやすさの点で最適です)。
7. Form Designer でフォームを編集・調整します。フォームコントロールに変更を加えると、コードジェネレータによってサーバ側プログラム内のデータ宣言に変更が反映されます (アプリケーションのコードは変更されません)。
8. サーバ側プログラムを編集・調整します。

Form Desinger の詳しい使用方法については、『CGI ベースのアプリケーションの新規作成』の章を参照してください。

### 2.2.2 サーバ側プログラムの作成

インターネットアプリケーションウィザードでは、インターネットアプリケーションのサーバ側の COBOL プログラムを生成します。プログラムは次の 3 通りの方法で生成できます。

- Form Designer で作成したフォームを基にサーバプログラムを生成する。
- 既存のコードからサーバプログラムとフォームを生成する。
- データベースからサーバプログラムとフォームを生成する。

インターネットアプリケーションウィザードの基本的な使用方法については、『入門書 - その他のトピ ック』のサンプルを参照してください。

### 2.2.3 インターネットアプリケーションのデバッグとテスト

作成したインターネットアプリケーションは、Solo Web サーバを使用して、開発マシン上でデバッグできます。Solo は Net Express 専用に設計され、作業中のプロジェクトと同じフォルダを使用するように自動的に設定されます。Solo を使用すれば、Web サーバの設定や実行に関する知識はいっさい必要ありません。

Solo の基本的な使用方法については、『入門書 - その他のトピ ック』のサンプルを参照してください。 Solo に関する説明は、Net Express のオンラインヘルプにあります。[**ヘルプ > ヘ**

**ルプトピック**] をクリックし、オンラインヘルプの目次で「**プログラミング > CGI ベースのアプリケーション > CGI ベースのアプリケーションのデバッグ**」を順にダブルクリックしてください。

Solo Web サーバは、インターネットアプリケーションの運用には適していません。アプリケーションを実際に運用するときには、市販の Web サーバソフトウェアを使用してください。また、アプリケーションのユニットテストも、運用に使用する Web サーバソフトウェアで実施する必要があります。

開発したアプリケーションを運用するため、Web サーバ上にディプロイする方法については、『CGI ベースのアプリケーションのディプロイ』の章で説明しています。

---



---

# 第 28 章 : フォームと HTML

この章では、HTML フォームの基本的な概念について説明します。また、Form Designer の出力オプションについても説明しています。

## 3.1 概要

フォームは、ユーザとアプリケーション間のインターフェイスです。最も単純なフォームには、いくつかのコントロール (エントリフィールド、オプションボタン、チェックボックスなど) と、フォーム送信用のプッシュボタンがあります。このプッシュボタンをクリックするとサーバ側プログラムの起動要求が行われ、フォームに入力したデータが連結されて Web サーバに送信されます。

このような単純なフォームの作成では、GUI やイベント駆動型プログラミングに関する知識はまったく必要ありません。フォームは Form Designer で視覚的に作成できるため、必要になるのは COBOL の知識だけです。

エンドユーザによるデータ入力時に複数のコントロールが連携するような、より複雑なフォームも作成できます。たとえば、オプションボタンとフィールドを連携させて、特定のオプションボタンを選択した場合のみ、対応するフィールドを有効にすることができます。このようなフォームを作成するには、HTML Web ページのスクリプト言語である JavaScript の知識が必要になりますが、その場合も Form Designer を使用すれば、Script Assistant でイベントハンドラを容易にセットアップして、フォームで発生するイベントを処理できます。イベントハンドラのセットアップについては、『クライアント側のプログラミング』の章で説明しています。

## 3.2 コントロールとデータ

フォーム内の各コントロールは、いずれも名前と値を持っています。エンドユーザによるフォームの送信時には、コントロールごとに対になった名前と値の集まりとして、フォーム内の情報がサーバ側プログラムに送信されます。COBOL 構文の拡張機能を使用すると、フォーム内のコントロールの名前を、サーバ側プログラム内の COBOL データ項目に直接関連付けることができます。この関連付けを行えば、フォーム内のデータがサーバ側プログラムに渡されるときに、コントロールの値が対応するデータ項目の値として設定されます。関連付けの方法については、『サーバ側のプログラミング』の章で説明しています。

COBOL と Form Designer には現在、次に挙げる各種コントロールの支援機能があり、これらのコントロールを容易に使用できます。

- エントリフィールド
- 選択ボックス
- オプションボタン
- チェックボックス
- プッシュボタン
- 入力イメージ (HTML 4.0 のみ)

いずれも HTML フォームで使用できる基本的なコントロールです。フォームの視覚的な作成については、『CGI ベースのアプリケーションの新規作成』の章で説明しています。

## 3.3 HTML と Java、および ActiveX

HTML ページには、標準的な HTML フォームコントロール以外にも、ActiveX コントロールと Java アプレットを追加できます。ActiveX コントロールと Java アプレットは、どちらも HTML ページの対話操作性を向上させる技術です。次の表は、ActiveX コントロールと Java アプレットの主な特徴と相違点を示しています。これらの技術はインターネットで利用できますが、それ以上に社内イントラネットなど、すべてのユーザのマシンに適切なブラウザをインストールできる環境に適しています。

| | ActiveX コントロール | Java アプレット |
|---|---|---|
| **対応プラットフォーム** | Windows の Internet Explorer のみ。 | Internet Explorer と Netscape Navigator (複数のオペレーティングシステムに対応)。<br><br>Java の全機能が、すべてのブラウザでサポートされるわけではない。 |
| **セキュリティ** | ActiveX コントロールの有効/無効をユーザのブラウザで選択できるほか、ActiveX コントロールの実行を個別に拒否することも可能。ActiveX コントロールには、発行元を認証するデジタル署名を付与することができる。 | Java アプレットの有効/無効をユーザのブラウザで選択できる。厳密なセキュリティモデルに基づき、アプレットによる PC 上の情報の変更や読み取りが防止される。 |
| **機能** | Windows API に制限なしでアクセスできるため、基本的には PC で実行される他のプログラムやアプリケーションと同様の機能を実装することが可能。 | 洗練された高度な GUI アプリケーションを作成できるが、Java アプレットの機能はセキュリティモデルによって制限される。 |

Form Designer を使用すれば、ActiveX コントロールや Java アプレットを、HTML コントロールの場合と同じ方法でページに追加できます。ActiveX コントロールや Java アプレットのプロパティは、Form Designer の Property Editor で設定できます。

---

**ヒント**：Java アプレットをフォームに追加するときに問題が発生するようであれば、***Net Express Links*** のリンク先にある Microsoft Service Packs Web サイトから最新の Microsoft Java 仮想マシンをダウンロードし、インストール後に再試行してください (最新の Java 仮想マシンは Internet Explorer 6 に付属しています)。

---

# 3.4 フォーム出力の種類

Form Designer とインターネットアプリケーションウィザードでは、フォームの配置情報を HTML ページとして保存するときに、次のいずれかの形式を選択できます。

- クロスプラットフォーム形式

  フォーム内の各要素が、HTML のテーブルを使用して配置されます。

- ダイナミック HTML

  フォーム内の各要素のサイズと位置が、スタイル属性によって正確に指定されます。

クロスプラットフォーム形式で保存した HTML ページは、さまざまな種類のブラウザで表示できます。ダイナミック HTML では、フォーム内の各要素をより正確に配置できますが、Internet Explorer 4.0 (およびそれ以降のバージョン) 以外のブラウザではサポートされていません。

Form Designer (ページウィザード) とインターネットアプリケーションウィザードでは、フォームの作成開始時に形式を選択できます。いったん指定した形式は、Form Designer で HTML ページを開き、「ページのプロパティ」を変更して保存すれば、切り替えることが可能です。「**ページのプロパティ**」ダイアログボックスを表示するには、[**ページ**] メニューの [**プロパティ**] をクリックします。

---



---

# 第 29 章 : CGI ベースのアプリケーションの新規作成

この章では、Form Designer でインターフェイスを作成し、インターネットアプリケーションウィザードでサーバ側のスケルトンプログラムを生成して、インターネットアプリケーションを新規作成する方法について説明します。

## 4.1 概要

Net Express では、インターネットやイントラネットで運用するアプリケーションを手軽に作成できます。必要なフォームは視覚的に配置して作成 (ペイント) できるほか、サーバ側プログラムを自動生成する機能もあります。インターネットアプリケーションウィザードで生成されるスケルトン (ひな型) のアプリケーションは、フォームから送信されたデータを受信して COBOL データ項目に格納し、さらにデータ項目に含まれている情報をフォームに出力します。開発者側でコードを記述するのは、入力と出力の間でデータを処理する部分だけです。

インターネットアプリケーションウィザードでは、次の 2 種類のプログラムを作成できます。

- 対称型 ― 入力フォームと同じフォームを出力するアプリケーション
- 非対称型 ― 入力フォームとは異なるフォームを出力するアプリケーション

この章では、次の手順に従ってアプリケーションの作成方法を示します。

1. フォームの作成
2. スケルトンアプリケーションの実行
3. 非対称型のサーバ側プログラムの作成 (すべてのアプリケーションが非対称型のサーバ側プログラムを使用するわけではありません)
4. サーバ側プログラムの機能の拡張

これらの手順をシンプルな例を通じて説明します。この章では、[**OK**] ボタンをどこでクリックするか、という細かいレベルの説明ではなく、全体的な手順の流れを示すことに重点を置いています。各手順の細かい説明については、オンラインヘルプを参照してください。作業に着手する前に、Form Designer と インターネットアプリケーションウィザードの操作に慣れておきたい方は、『入門書 - その他のトピック』のサンプルセッションを利用できます。

## 4.2 Form Designer

この項では、Form Designer の概要を説明します。Net Express で新しい HTML ページを作成したり、既存の HTML ページを編集するときには、Form Designer が起動します。

### 4.2.1 Form Designer のウィンドウ

次の図は、Net Express の Form Designer で HTML ページを開いた状態を示しています。

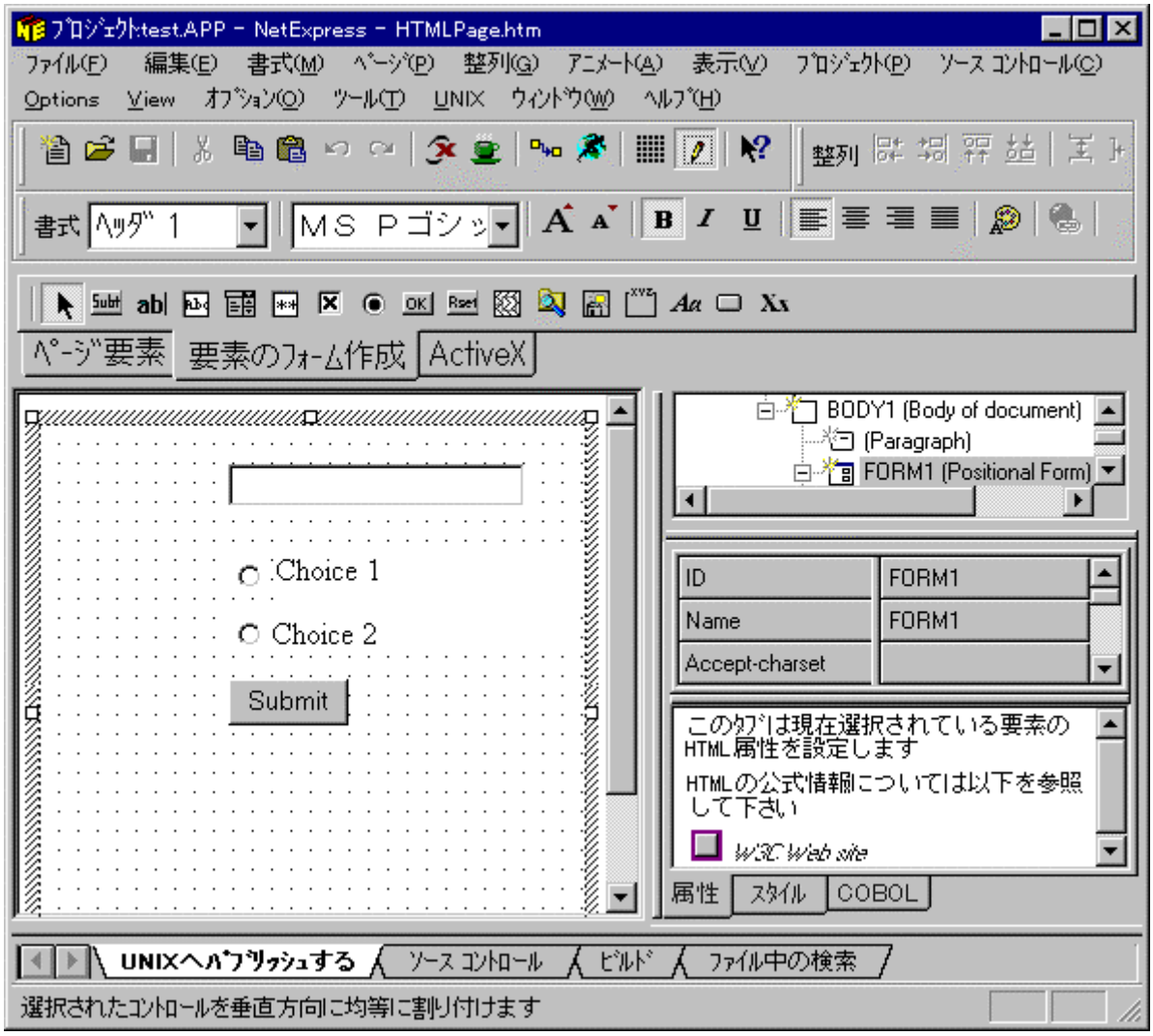



**図 29-1 Form Designer**

Form Designer の主な構成要素は次のとおりです。

- **ページデザイン領域**

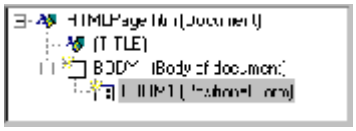

現在作業しているページやフォームを編集するための領域です。

- **プロパティエディタ**

**ページツリービューで選択している要素の全プロパティが表示されます。通常、ページデザイン領域とページツリービューの一方で要素を選択すると、他方でも同じ要素が選択されます。ただし、<BODY> などページデザイン領域に現れない要素は例外です。**

**プロパティは種類ごとに異なるペインに表示され、下部のタブをクリックしてペインを切り替えます。「属性」ペインには HTML 属性、「スタイル」ペインには要素の CSS スタイル設定が表示されます。フォームまたはフォーム要素を選択している場合には、リンクされているデータ項目のプロパティを「COBOL」ペインで設定できます (『プロパティとリンクデータ項目』を参照してください)。また、Java アプレットや ActiveX コントロールを選択している場合には、公開されている各プロパティを「プロパティ」ペインで設定できます。**

- **ページツリービュー**

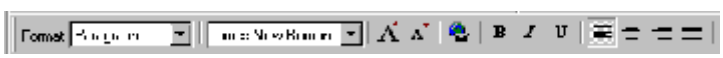


**ページ内の全要素が階層表示されます。ページデザイン領域内の要素は、ページツリービュー内の対応する項目をクリックして選択することもできます。また、ページツリービュー内を右クリックし、ポップアップメニューで [要素をフィルタ処理] をクリックすれば、特定の種類の要素を非表示にすることも可能です。**

- **構成要素パレット**

**このパレットを使用すると、ページやフォームに要素を追加できます。要素の追加に使用する各種のアイコンは、種類ごとに異なるタブに表示されます。たとえば、[ページ要素] タブには、ページのあらゆる位置に配置できる HTML 要素のアイコンが含まれており、[フォーム要素] タブには、HTML フォームだけに配置可能な HTML 要素のアイコンが表示されます。**

- **書式ツールバー**



選択したテキストの書式を変更するツールバー。段落から見出しへの変更、フォントと色の変更、太字、斜体、下線の使用、およびテキストの整列を実行するためのツールがあります。ハイパーリンクを追加することもできます。

### 4.2.2 配置編集とフロー編集

Form Designer には HTML エディタと HTML フォームペインタという 2 つの側面があり、この両面のニーズに対応するため、2 つの編集モードを備えています。

- 配置編集モード

  コントロールやテキスト ラベルを位置フォームの任意の位置にドロップできます。フォーム内の要素は、マウスをドラッグして移動できます。

- フロー編集モード

  テキストエディタと同様に、テキストの入力や挿入、上書き、および削除を行うことができます。

フロー編集がデフォルトの編集モードです。Form Designer を配置編集モードに切り替えることができるのは、ページで位置フォームを選択している場合だけです。HTML ページウィザードで新しいページを作成するときに**位置フォーム**テンプレートを選択すると、Form Designer によって単一の位置指定フォームを含む空のページが生成されます。

「**フォーム要素**」パレットの [位置フォーム] ボタンをクリックすれば、HTML ページ内の任意の位置に位置フォームを挿入できます (Form Designer によって、現在のテキストカーソル位置にフォームが挿入されます)。位置フォームの上または下をクリックすると、そのフォームの直または後にテキストや画像を追加できます。

フロー編集モードでは、「**フォーム要素**」パレットで任意のフォーム要素をクリックして、編集中のページにフォーム要素を追加することもできます。この場合、ページの現在のカーソル位置に HTML フォームを挿入するかどうかを尋ねるダイアログボックスが表示されます。フォームの配置編集はできませんが、段落、空白文字、テーブルなどを使用して、フォーム内に要素を配置できます。

### 4.2.3 プロパティとリンクデータ項目

Form Designer では、フォーム内のコントロールを COBOL サーバ側プログラムのデータ項目にリンクすることができます。Form Designer で作成したフォームを基に、インターネットアプリケーションウィザードでサーバ側プログラムを生成すると、フォーム内の名前を付けられた各コントロールに対応するデータ項目 (リンクデータ項目) を含むプログラムが生成されます。

インターネットアプリケーションウィザードは、**Name** 属性を参照 (コントロールが **Name** 属性を持たない場合は **ID** 属性) し、そのコントロールを表す同じ名前のデータ項目をサーバ側プログラム内に生成します。**Name** 属性と **ID** 属性は、どちらもプロパティエディタの「**属性**」ペインで設定します。データ項目の PICTURE 値は、プロパティエディタの「**COBOL**」ペインで設定し

た COBOLPicture プロパティから取得されます。COBOLPicture プロパティが空白の場合、リンクデータ項目は生成されません。

フォームをサーバ側プログラムの入力として送信するときには、フォーム内のコントロールの値が、対応するデータ項目に格納されます。フォームがサーバ側プログラムから出力されるときには、サーバ側プログラム内のデータ項目の値が、フォーム内の対応するコントロールに設定されます。

次の図は、入力フィールド、2 つのオプションボタン、および 2 つの送信ボタンを持つ入力フォームと、3 つの入力フィールドを持つ出力フォームを示しています。これらのフォームの間に、インターネットアプリケーションウィザードで生成されたリンクデータ項目を持つサーバ側プログラムが位置しています。

アプリケーションのエンドユーザが送信ボタンをクリックすると、入力フォーム内のコントロールの値がサーバ側プログラムに渡されます。2 つのオプションボタンは同じ名前を共有しており、サーバ側プログラムの 1 つのデータ項目に対応しています。同様に、送信ボタンも 1 つの名前とデータ項目を共有しています。サーバ側プログラムから返信される出力フォーム内の各フィールドには、リンクされているデータ項目の値が設定されています。

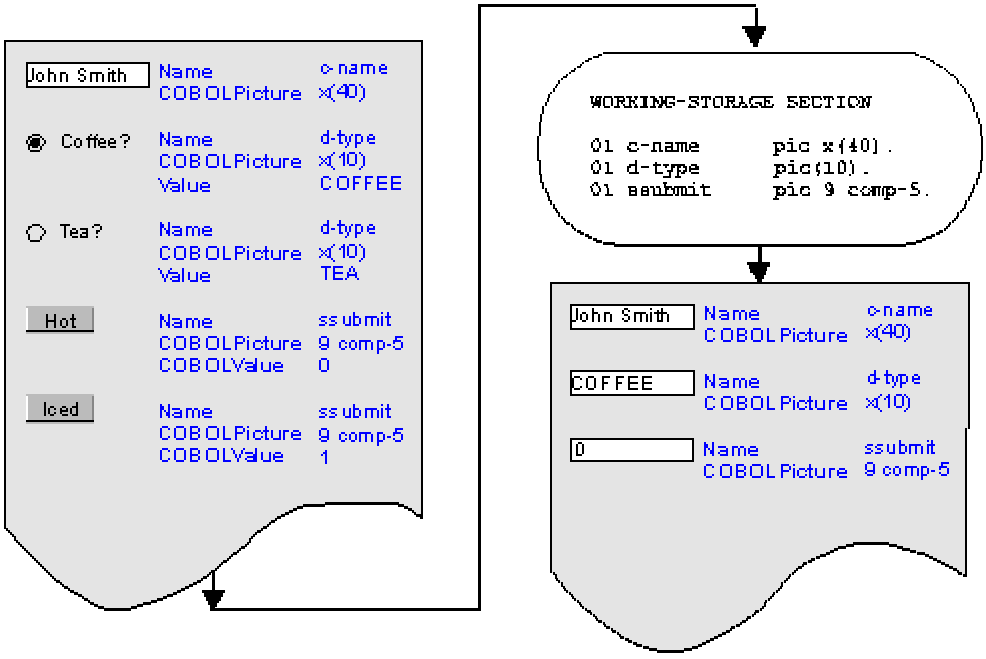



**図 29-2 コントロールとデータ項目の対応**

## 4.3 フォームとサーバ側プログラムの作成

この項では、フォームとサーバ側プログラムを作成するための基本的な手順を、例を交えて説明します。Form Designer の主な役割は、COBOL サーバ側プログラムでの使用に適したフ

ォームの作成ですが、その他のあらゆる HTML ページも読み込んで編集できます。ページウィザードで新しいページを作成するときには、任意の HTML ページをテンプレートとして使用できます。また、Net Express のプロジェクトディレクトリに HTML ページをコピーしてプロジェクトに追加し、Form Designer で直接編集することも可能です。

このように、Form Designer によるアプリケーションのページのデザインと作成は柔軟性に富んでおり、専門のレイアウトデザイナーが作成したページを基に、Form Designer でフォームを追加することもできます。ただし、ページ内でサーバ側プログラムとの通信に使用されるのはフォームだけであるため、この章の説明では分かりやすさを優先して、1 つの位置フォームだけを含む HTML ページを使用します。

フォームを作成してサーバ側プログラムを生成する前に、次の選択を行う必要があります。

- アプリケーションの種類 (非対称型アプリケーションまたは対称型アプリケーション)

  非対称型アプリケーションは入力フォームを含むページと出力ページが異なり、対称型アプリケーションは入力と出力に同じフォームおよびページを使用します)。

  対称型アプリケーションと非対称型アプリケーションについては、『CGI ベースのインターネットプログラミングの概要』の章を参照してください。

- 入力と出力に使用するフォーム

非対称型アプリケーションでは、2 つのページが同じサーバ側プログラムを共有します。サーバ側プログラムを生成するときには、HTML ページと、その中に含まれ、プログラムへの入力として使用するフォーム、および別の出力ページを指定します。非対称型アプリケーションの追加情報については、『非対称型のサーバ側プログラムの作成』の項を参照してください。対称型アプリケーションのサーバ側プログラムを作成する場合には、入力と出力に同じフォームとページを指定します。

以下の説明は、全体的な手順の流れを示すことに重点を置いています。各手順の細かい説明については、オンラインヘルプを参照してください。 Form Designer と関連するウィザードの操作に慣れておきたい方は、『入門書 - その他のトピック』のサンプルセッションを利用できます。

1. アプリケーションファイルの保存先になる Net Express のプロジェクトを作成します。
2. [**ファイル > 新規作成**] をクリックし、「**新規作成**」ダイアログボックスで「HTML ページ」を選択してページウィザードを開きます。
3. ページウィザードに情報を入力します。

   「HTML ページの新規作成」ページの「全般」タブと「フレーム」タブに表示される標準テンプレートファイルを、作成するページのベースとして使用できます。また、[既存のページ] タブをクリックして既存の HTML ファイルを選択し、それを基にページを作成することも可能です。

4. ページやその中のフォームを編集します。

5. フォームを保存します。
6. アプリケーションの入力フォームと出力フォームのページを作成した後、[**ファイル > 新規作成**] をクリックして「インターネットアプリケーション」を選択し、インターネットアプリケーションウィザードを開きます。
7. インターネットアプリケーションウィザードで [サーバプログラム] を選択し、[次へ] をクリックします。
8. 「サーバプログラムの生成」ページで、サーバプログラムの生成に使用する入力ページ (1 つ以上の入力フォームを含むページ) を 1 つ、出力ページを 1 つ以上選択します。対称型アプリケーションの場合、入力と出力に同じページを選択します。

### 4.3.1 サンプルフォームの作成

以下の内容では、Net Express で CGI ベースのインターネットアプリケーションを作成する基本的な手順を、サンプルアプリケーションを通じて説明します。この項では、サンプルアプリケーションのフォーム部分を作成します。ここで作成するのは、社内イントラネットにコーヒーマシンが接続されていることを想定したホットドリンクのオーダーフォームです。完成したフォームは、次のようになります。

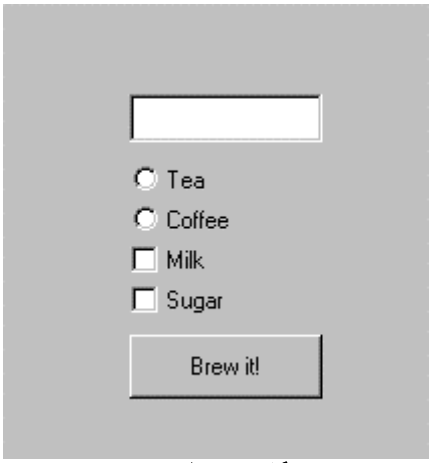



**図 29-1 ドリンクオーダーフォーム**

以下の手順説明では、個々の作業の詳細については説明しません。作業に着手する前に、Form Designer の操作に慣れておきたい方は、『入門書 - その他のトピック』のサンプルセッションを利用できます。Form Designer で実行するほとんどの作業の手順は、Form Designer のヘルプに詳しく説明されています。[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「目次」タブで「**プログラミング > CGI ベースのアプリケーション > Form Designer**」を順にダブルクリックして、表示される情報を参照してください。Form Designer の大部分のダイアログボックスや入力フィールドでは、コンテキストヘルプも使用できます。ダイアログボックスで疑問符 (?) のアイコンをクリックし、続いてヘルプの対象項目をクリックします。フィールド内にカーソルを置いた状態で **F1** キーを押す方法もあります。

ここで使用しているサンプルアプリケーションは、**Net Express\base\demo\bevord_h** ディレクトリに格納されています。

ドリンクフォームを作成する手順を次に示します。

1. Net Express を起動し、**bevord.app** という名前で空プロジェクトを作成します。
2. [**ファイル > 新規作成**] をクリックし、「**新規作成**」ダイアログボックスで「**HTML ページ**」を選択してページウィザードを開きます。

   ページウィザードに次の情報を入力します。

   - テンプレート ― 位置フォーム
   - HTML 出力 ― クロスプラットフォーム
   - HTML ファイル名 ― **bevord_h**

   [**完了**] をクリックすると **bevord_h.htm** ファイルが生成され、Form Designer に表示されます。

3. 「**フォーム要素**」パレットを使用して、フォームにテキストエントリフィールドを追加します。プロパティエディタで次のプロパティを設定します。
   - **Name** ― **customername** (「**属性**」ペイン)
   - **COBOLPicture** ― **X(60)** (「**COBOL**」ペイン)

   インターネットアプリケーションウィザードでサーバ側プログラム **bev.cbl** を生成する際には、データ項目 customername が生成されます。フォームを送信するときには、このエントリフィールドの入力値が、customername 項目に渡されます。

4. 「フォーム要素」パレットの [テキスト] ボタンを使用し、テキストエントリフィールドの左側にスパン要素を追加し、その内側をクリックします。Name と入力します。

   スパン要素を使用して、位置フォームにラベルを追加します。

5. フォームにオプションボタンを追加します。キャプションを Tea に変更します。

   デフォルトでは、コントロールをフォームにドロップするときにキャプションの文字列が選択状態になっており、そのまま上書きできます。それ以降にキャプションを変更する際には、キャプション文字列内をクリックし、テキストカーソルを表示させます。その後、変更する文字列部分を選択して上書きします。

   オプションボタン自体をクリック (またはページツリービューでオプションボタンを選択) し、次のプロパティを設定します。

   - **Name** ― **beveragetype**
   - **Value** ― **TEASELECTED**
   - **COBOLPicture** ― **X(20)**

   このオプションボタンがフォーム送信時に選択されていると、データ項目 beveragetype が TEASELECTED に設定されます。

6. フォームに 2 つ目のオプションボタンを追加します。キャプションを Coffee に変更し、次のプロパティを設定します。

- **Name** ― **beveragetype**
- **Value** ― **COFFEESELECTED**
- **COBOLPicture** ― **X(20)**

このオプションボタンがフォーム送信時に選択されていると、データ項目 beveragetype が COFFEESELECTED に設定されます。

7. フォームにチェックボックスを追加します。キャプションを Milk に変更し、次のプロパティを設定します。
   - **Name** ― **milkrequest**
   - **Value** ― **1**
   - **COBOLPicture** ― **9**

   このチェックボックスがフォーム送信時に選択されていると、データ項目 milkrequest が 1 に設定されます。

8. フォームに 2 つ目のチェックボックスを追加します。キャプションを Sugar に変更し、次のプロパティを設定します。
   - **Name** ― **sugarrequest**
   - **Value** ― **1**
   - **COBOLPicture** ― **9**

   このチェックボックスがフォーム送信時に選択されていると、データ項目 sugarrequest が 1 に設定されます。

9. 送信ボタンを追加します。ボタンのキャプションを Brew It! に設定し、次のプロパティを設定します。
   - **Name** ― **Submit**
   - **Value** ― **1**
   - **COBOLPicture** ― **9**

   この設定によって、ボタンをクリックしたときに 1 に設定される COBOL データ項目 submit の宣言が、**bev_h.cbl** の生成時に追加されます。

10. ツールバーの [**保存**] ボタンをクリックしてフォームを保存します。

    ページの全設定が **bevord_h.htm** ファイルに保存され、 サーバ側プログラム内にリンクデータ項目を生成するための情報が **bevord_h.mff** に保存されます。

    これらのファイルはプロジェクトに追加されます (**.mff** ファイルは「プロジェクト」ウィンドウ右側のソースプールだけに追加され、**.htm** ファイルは「プロジェクト」ウィンドウの両方のペインに追加されます)。Net Express では、.mff ファイルと .htm ファイルのどちらを開いても、Form Designer が開きます。

    以上の操作で、作成したフォームを使用するシンプルな CGI (サーバ側プログラム) を生成する準備が整います。

11. [**ファイル**] メニューの [**新規作成**] をクリックし、「インターネットアプリケーション」を選択してインターネットアプリケーションウィザードを開きます。
12. ウィザードに次の情報を入力します。
    - サーバプログラム
    - 入力ファイル － **bevord_h.htm**
    - 入力フォーム － **FORM1** (作成したフォームのデフォルト名)
    - 出力フォーム － **bevord_h.htm**
    - ファイル名 － **bev_h.cbl**

    ウィザードの [**完了**] をクリックすると、次のサーバ側プログラム用ファイルが生成されます。

    - **bev.cbl**
    - **bev.cpf**
    - **bev.cpv**
    - **bev.cpy**

    **.cbl** ファイルがプログラムで、その他はフォームデータのコピーファイルです。これらのファイルの詳細は、この章で後述します。

フォームが実際にエンドユーザ側でどのように表示されるかを確認するには、関連付けられた CGI アプリケーションを実行する必要があります。Windows のエクスプローラでフォームファイルをダブルクリックした場合など、Web ブラウザにフォームを直接ロードすると、エントリフィールドには次の文字列が表示されます。

```
:f-customername
```

この文字列は、CGI プログラムからフォームに送信されるデータを表すパラメータです。CGI プログラムを実行してフォームを表示した場合は、このエントリフィールドには空白文字か、プログラムから返された情報が表示されます。プログラムの実行については、次の項で説明します。

# 4.4 スケルトンアプリケーションの実行

前項までの作業で、次の項目を含む実行可能なインターネットアプリケーションのスケルトンが作成されました。

- フォーム
- サーバ側プログラム

サーバ側プログラムをビルドして実行し、問題がないかどうかをテストできます。

## 4.4.1 サーバ側プログラムのビルド

この章では、サーバ側プログラムをビルドする方法を示します。実行とアニメートが最も容易な CGI プログラムとしてビルドします。CGI としてビルドしたプログラムは、動作の検証後に ISAPI プログラムや NSAPI プログラムに変更できます。詳細については、『CGI、ISAPI および NSAPI プログラム』の章を参照してください。

サーバ側プログラムのビルド方法は、次のとおりです。

- [**プロジェクト**] メニューで [**リビルド**] をクリックします。

アプリケーションがビルドされ、実行可能な状態になります。

### 4.4.2 アプリケーションの実行

アプリケーションを実行するには、ローカルマシンで Web サーバを起動する必要があります。Solo 以外の Web サーバを使用する場合は、『アプリケーションのディプロイ』の章の説明に従って、Net Express アプリケーションをデバッグできるように Web サーバを設定します。他の Web サーバを使用する方法は、Net Express のオンラインヘルプにも簡単に説明されています。[**ヘルプ**] メニューの [**ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**プログラミング > CGI ベースのアプリケーション > CGI ベースのアプリケーションのデバッグ > 方法 > CGI ベースのアプリケーションのアニメート**」を順にダブルクリックします。

アプリケーションを実行する前に、ローカルマシンでのインターネットアプリケーションの実行に慣れておきたい方は、『入門書 - その他のトピック』に記載されている Solo Web サーバ用のサンプルセッションを利用できます。

Solo を使用する場合、アプリケーションは次の方法で実行します。

- [**アニメート > 実行**] をクリックします。.

### 4.4.3 アプリケーションのデバッグ

Net Express のアニメータを使用すれば、プログラムをフローに従ってステップ実行できます。デバッグを開始する前に、アニメータの操作に慣れておきたい方は、『***入門書***』の『Net Express の使用方法』の章に記載されているサンプルセッションを利用できます。

インターネットアプリケーションのアニメート手順を次に示します。

1. アニメート設定は Form Designer で行っており、すでに Solo Web サーバを使用してアプリケーションをアニメートできる状態になっています。[**アニメート > 設定**] をクリックすれば、「**アニメーションの起動**」フィールドにサーバ側プログラムの URL が次のように設定されていることを確認できます。
2. 
3. `http://127.0.0.1/cgi-bin/`*program*`.exe`

   *program*.exe は サーバ側プログラムの名前です。Solo 以外の Web サーバを使用する場合は、URL を次のように変更してください。

http://*machine.domainname*/cgi-bin/*program*.exe

Solo 以外の Web サーバを使用する方法については、[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**プログラミング > CGI ベースのアプリケーション > CGI ベースのアプリケーションのデバッグ > 方法 > CGI ベースのアプリケーションのアニメート**」を順にダブルクリックして、表示される説明を参照してください。

4. Web サーバが動作していなければ、ここで起動します。Solo を使用する場合は、Net Express によって自動的に起動されるため、このステップは省略できます。
5. [**アニメート > アニメート開始**] をクリックします。

   Web ブラウザが起動してサーバ側プログラムの URL にアクセスし、Web サーバにプログラムの実行要求が送信されます。プログラムを起動するとアニメータも起動します。この時点でサーバ側プログラムのステップ実行が可能になります。

   EXEC HTML 文をステップ実行するたびに、プログラムから Web ブラウザにページが出力されます。

---

**注記**：この説明は対称型アプリケーションの場合です。非対称型アプリケーションの場合、「**アニメーションの起動**」フィールドはアプリケーションの入力ページの URL に設定されており、入力ページの送信ボタンをクリックするとデバッガに COBOL のコードが表示されます。

---

### 4.4.4 手順の要約

インターネットアプリケーションのサンプルとして生成したサーバ側のスケルトンプログラムを実行すると、問題がないかどうかデバッグしてテストできます。前述したフォームの作成手順と同様、ここでも手順の各作業の詳細については説明しません。 次の手順を開始する前に、Solo Web サーバの使用方法に慣れておきたい方は、『入門書 - その他のトピック』のサンプルセッションを利用できます。

サンプルをビルドして実行する方法は、次のとおりです。

1. Net Express でプロジェクトをリビルドします。
2. [**アニメート > 実行**] をクリックします。.

デバッガでコードの実行を確認する手順は、次のとおりです。

1. [**アニメート > アニメート開始**] をクリックします。

   Web ブラウザが起動してプログラムの URL にアクセスし、Net Express のアニメータ内でプログラムの実行が開始されます。Web ブラウザにただちにフォームが表示され、

Net Express のアニメータでコードの実行開始を確認できない場合には、ブラウザの [**更新**] ボタンまたは [**再読み込み**] ボタンをクリックしてください。

2. アニメータの強力なデバッグ機能を使用して、プログラムのコードをステップ実行します。

   アニメータの使用方法の詳細については、Net Express の [**ヘルプ**] メニューで [**ヘルプトピック**] をクリックし、「目次」タブで「開発環境 > プログラムのデバッグおよびアニメーション」の順に選択してください。

## 4.5 非対称型のサーバ側プログラムの作成

『インターネットプログラミングの概要』の章では、サーバ側プログラムを次のように対称型と非対称型に分類しました。

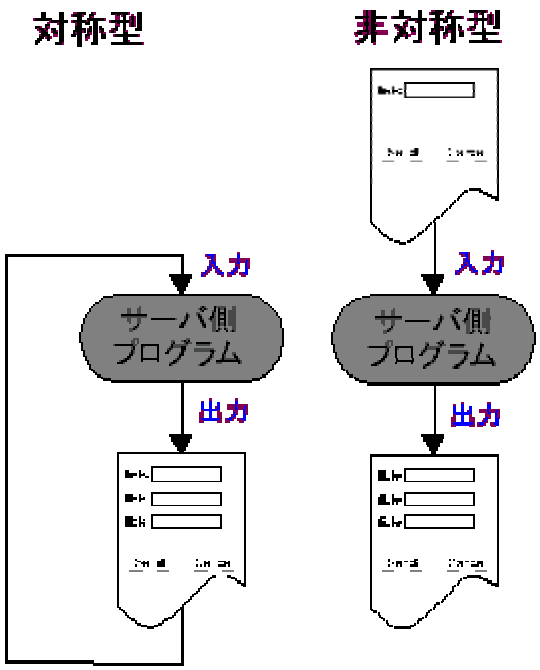



**図 29-2 対称型アプリケーションと非対称型アプリケーション**

この章の前述した内容では、対称型プログラムの作成手順を紹介してきました。 この項では、Form Designer を使って非対称型プログラムを作成する方法に目を向けます。非対称型プログラムの開発では、入力フォームと出力フォーム、およびこれらのフォームを仲介するサーバ側プログラムを作成する必要があります。

非対称型アプリケーションの大まかな開発手順は次のとおりです。

1. ページウィザードで入力フォームを作成する。
2. ページウィザードで出力フォームを作成する。
3. インターネットアプリケーションウィザードを開始してサーバプログラムを選択し、さらに入力フォームと出力フォームを選択する。

インターネットアプリケーションウィザードで [**完了**] をクリックすると、サーバ側のスケルトンプログラムが自動的に生成され、プロジェクトに追加されます。このプログラムには、入力フォームからの入力を受け付け、出力フォームを返すためのコードが含まれています。

### 4.5.1 非対称型アプリケーションの一例

対称型アプリケーションと非対称型アプリケーションの違いを示すため、前述したドリンクオーダーアプリケーションに 2 つ目のフォームを追加します。このフォームに、入力フォームの選択結果を集計して出力します。この項では、出力フォームの作成手順を示します。生成されたアプリケーションに、入力データを処理して出力フォーム用に集計するコードを追加する手順は、次項以降で示しています。

出力フォームは次のように表示されます。

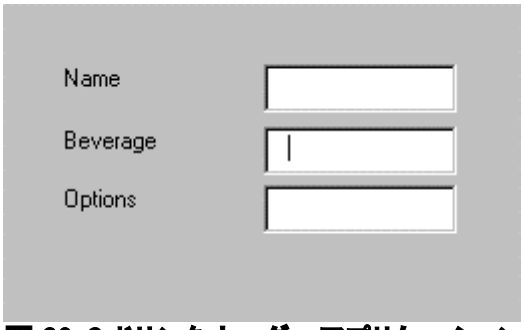



**図 29-3 ドリンクオーダーアプリケーションの出力フォーム例**

このフォームは他のサーバ側プログラムには接続しないため、プッシュボタンはありません。

出力フォームの作成手順は次のとおりです。

1. Net Express を起動し、前述した手順で作成した **bevord.app** プロジェクトをロードします。
2. [**ファイル**] メニューの [**新規作成**] をクリックし、「**新規作成**」ダイアログボックスで「HTML ページ」を選択してページウィザードを開きます。位置フォームを **bevsum_h** というファイル名で作成します。
3. フォームに Name、Beverage、Options の各ラベルを追加します ([フォーム要素] メニューの [テキスト] を使用して、テキスト挿入用のスパン要素をフォームに追加します)。
4. フォームに 3 つのテキストエントリコントロールを追加し、プロパティを次のように設定します。
    a. **Name** ― **customerout**、**COBOLPicture** ― **X(60)**
    b. **Name** ― **beverageout**、**COBOLPicture** ― **X(60)**
    c. **Name** ― **optionsout**、**COBOLPicture** ― **X(60)**

    サーバ側プログラムを再生成するときに、データ項目 CUSTOMEROUT、BEVERAGEOUT、および OPTIONSOUT が自動的に宣言されます。

5. フォームを保存します。

6. インターネットアプリケーションウィザードを開きます。
7. ウィザードに次の情報を入力します。
    - サーバプログラム
    - 入力ファイル － **bevord_h.htm**
    - 入力フォーム － **FORM1**
    - 出力フォーム － **bevsum_h.htm**
    - プログラムファイル名 － **bev_h.cbl**

ウィザードで [**完了**] をクリックすると、サーバ側プログラムのスケルトンが再生成されます。既存のサーバ側プログラムを上書きするかどうか確認するダイアログボックスが表示されます。既存のサーバ側プログラムにビジネスロジックを追加した場合は上書きします。ここで別の名前を指定してプログラムを生成すれば、既存のプログラムを残しておくことができます。

# 4.6 サーバ側プログラムへの機能の追加

この項では、インターネットアプリケーションウィザードで生成されたサーバ側のスケルトンプログラムの構成について説明します。対称型アプリケーションの場合、サーバ側のスケルトンプログラムは、エンドユーザが入力したデータを含むフォームをエコーバックするだけです。サーバ側のスケルトンプログラムを実用可能なアプリケーションにするには、機能を追加する必要があります。

インターネットアプリケーションウィザードでは、次の各ファイルが生成されます。

- ***program*.cbl**

  サーバ側プログラムのスケルトンコードを含む COBOL ソースファイル。生成されたファイルのうち、編集するのはこのファイルだけです。

- ***program*.cpy**

  HTML コントロールの COBOL Picture プロパティに指定した PICTURE 文字列の変数を含む COBOL コピーファイル。これらのデータ項目の名前は、Form Designer で設定した名前と同じになります。

  非対称型アプリケーションのプログラムを生成する場合、このコピーファイルで宣言されるデータ項目は、入力フォームと出力フォームのリンクデータ項目を統合したものになります。つまり、入力フォームと出力フォームで名前が同じコントロールには、リンクデータ項目が 1 つだけ生成されます。

- ***program*.cpf**

  フォームとプログラム間のデータ送信に使用されるデータ項目を宣言する COBOL コピーファイル。データ項目はすべて PIC X(n) 型で、フォーム内の対応するコントロールの Name プロパティの値が名前になります。

非対称型アプリケーションのプログラムを生成する場合、このコピーファイルで宣言されるデータ項目は、入力フォームと出力フォームのリンクデータ項目を統合したものになります。

- ***program*.cpv**

  フォームとプログラムの間でやり取りされる形式 (**.cpf** ファイル内のデータ項目) と、各コントロールの COBOL Picture プロパティで指定した形式 (**.cpy** ファイル内の変数) の間でデータ変換を行うコードが含まれた COBOL コピーファイル。

  入力フォームを指定してプログラムを生成した場合、そのフォームから渡されるデータが入力変換され、フォームに渡されるデータが出力変換されます。

アプリケーションに機能を追加するには、生成された ***program*.cbl** ファイルに変更を加えます。サーバ側プログラムに関連付けたフォームを保存すると、そのたびに上記の各コピーファイルが再生成されます。したがって、サーバ側プログラムには常に、使用するフォームとの通信に必要なすべてのデータ項目が含まれています。

主となる 一方、プログラムのソースコードを含む .cbl ファイルは自動更新されないため、プログラムに加えた変更はそのまま維持されます。上記の各コピーファイルは編集しないでください。コピーファイルに変更を加えても、サーバ側プログラムに関連付けたフォームを編集して保存すると、その変更は失われてしまいます。

### 4.6.1 コード例

ここまでの内容では、入力フォームと出力フォームを作成し、サーバ側プログラムを生成しました。この項では、入力フォームに入力された全データを取り込んで集計し、出力フォームに送信するコードを、生成されたサーバ側プログラムに追加します。このコードは非常にシンプルですが、サーバ側プログラム内のどの部分を、どのように変更できるかを示しており、これらのポイントは複雑なアプリケーションにもそのまま適用できます。

プログラムに機能を追加する手順は次のとおりです。

1. Net Express IDE の「プロジェクト」ウィンドウで **bev.cbl** ファイルをダブルクリックし、編集を開始します。
2. PROCESS-BUSINESS-LOGIC 節を見つけ、次のコメントの下の行にカーソルを移動します。

```
3.
4.    *> Add application business logic here
```

5. 次のコードを追加します。

```
6.
7.     *> 顧客名を出力
8.        move CustomerName to CustomerOut
9.     *> 飲み物の種類を出力
10.       if BeverageType = "TEASELECTED"
11.           move "Tea" & x"0" to BeverageOut
```

```
12.       end-if
13.       if BeverageType = "COFFEESELECTED"
14.           move "Coffee" & x"0" to BeverageOut
15.       end-if  *> ミルクと砂糖の選択を集計
16.       if MilkRequest = 1
17.           if SugarRequest = 1
18.               move "Milk and sugar" to OptionsOut
19.                    else
20.               move "Milk" to OptionsOut
21.           end-if
22.       else
23.           if SugarRequest = 1
24.               move "Sugar" to OptionsOut
25.           end-if
26.       end-if
```

27. 変更を保存し、アプリケーションをリビルドします。
28. 入力フォームの URL を指定してアプリケーションを起動し、再実行します。

    フォームにデータを入力して送信ボタンをクリックすると、入力内容を集計した出力フォームが表示されます。

# 第 30 章 : CGI ベースのデータアクセスアプリケーション

この章では、インターネットアプリケーションウィザードを使って、実用的な SQL アプリケーションを簡単に作成する方法を説明します。また、作成したアプリケーションの機能を拡張する方法についても説明しています。

## 5.1 概要

インターネットアプリケーションウィザードでは、SQL データベースにアクセスするための HTML フォームとサーバ側コード一式を生成できます。このウィザードは、Open ESQL アシスタント を使用して SQL クエリーコードを生成し、サーバ側プログラムに埋め込みます。インターネットアプリケーションウィザードで生成されるデータアクセスアプリケーションでは、エンドユーザは次のデータベース機能の一部、またはすべてを利用できます。

- クエリー
- 更新
- 追加
- 削除

利用可能にする機能は、アプリケーションの作成中に指定します。

次の 2 種類のビューを生成してデータベースを表示できます。

- 単一レコードビュー (フォームビュー)

  エントリフィールドにクエリーの各列を表示するフォーム。エンドユーザがデータベースの情報を更新するときには、このビューを使用します。

- テーブルビュー

  レコードのグループを表示するテーブル。一度に表示されるレコード数は、アプリケーションの生成中に指定できます。デフォルトでは 10 レコードです。

各ビューは、それぞれ個別のフォームと、サーバー側プログラムによって処理されます。ウィザードが生成するサーバ側プログラムは、入力と出力に同じフォームを使用します。

単一レコードビューとテーブルビューの両方を持つアプリケーションを生成すると、2 つのビューは互いにリンクされます (単一レコードビューのフォームに、テーブルビューを表示するためのプッシュボタンが付けられ、テーブルビュー内の各レコードには、単一レコードビューを表示するためのハイパーテキストリンクが追加されます)。

COBOL アプリケーションと SQL アプリケーションの詳細については、オンラインマニュアル『データベースアクセス』を参照してください。

# 5.2 データアクセスアプリケーションの作成

インターネットアプリケーションウィザードでデータアクセスアプリケーションを作成する方法を学ぶ一番の近道は、実際に試してみることです。ここでは、インターネットアプリケーションウィザードを使用して、2 つのプログラムを作成します。1 つは顧客情報テーブルを照会して顧客の詳細情報を取得するプログラム、もう 1 つは注文テーブルに照会して注文の詳細情報を取得するプログラムです。 プログラムの作成に着手する前に、データソースを作成する必要があります。 この方法については、ヘルプの『方法』を参照してください。 このヘルプでは、2 つのデータソース (UserSample1、UserSample2) の作成を推奨しています。

データアクセスに必要なすべての ESQL コードを含む 2 つの基本的なプログラムを生成した後、これらに変更を加えて注文処理システムを作成します。顧客レコードとその顧客の注文を直接リンクする操作も行います。つまり、インターネットアプリケーションウィザードでデータベースアクセス用の簡単なアプリケーションを生成する方法だけでなく、生成されたアプリケーションをカスタマイズし、拡張する方法も説明しています。

## 5.2.1 アプリケーションの生成

この項では、次の処理を行う 2 つの簡単なアプリケーションを生成します。

- 顧客の照会/更新
- 注文の照会/更新

顧客の照会では、エンドユーザからデータベースに顧客情報を照会できます。注文照会/更新では、エンドユーザが注文を検索し、更新できます。

### 5.2.1.1 顧客照会

Net Express で開いているプロジェクトがある場合は、この時点でプロジェクトを閉じます。Net Express のプロジェクトを閉じないと、インターネットアプリケーションウィザードで生成されたすべてのファイルが、そのプロジェクトに追加されてしまいます。ここで生成するアプリケーションのファイルは、すべて 1 つの専用プロジェクトに格納します。開いているプロジェクトがなければ、インターネットアプリケーションウィザードによって、新しいプロジェクトの名前と場所の指定を求めるメッセージが表示されます。

この章では、[OK] ボタンをどこでクリックするか、という細かいレベルの説明ではなく、全体的な手順の流れを示すことに重点を置いています。 作業に着手する前に、Form Designer と インターネットアプリケーションウィザードの操作に慣れておきたい方は、『入門書 - その他のトピック』のサンプルセッションを利用できます。

顧客の照会を生成する手順は次のとおりです。

1. インターネットアプリケーションウィザードを開始します。[**ファイル**] メニューの [**新規作成**] をクリックし、「**新規作成**」ダイアログボックスで [**インターネットアプリケーション**] を選択します。
2. 次の情報を記入して新規プロジェクトを作成し、[**次へ**] ボタンをクリックします。

   プロジェクト名　　**orders**

   プロジェクトのパス　**Net Express\base\workarea\orders**

3. **Wizard の入力**で「**SQL データベース**」を選択し、[**次へ**] をクリックします。
4. 次の情報を入力して [**次へ**] ボタンをクリックします。

   ファイルの基本名　　**customer**

   生成されるフォームのタイトル　**顧客の詳細情報**

5. インターネットアプリケーションウィザードによって、アプリケーションが生成されます。生成するファイルがすでに存在する場合は、その旨を通知するメッセージが表示され、該当するファイルを維持または上書きを選択できます。
6. 「**データの選択**」ダイアログボックスで Net Express 4.0 Sample 2 をダブルクリックし、続いて「Customer」テーブルをダブルクリックして選択します。「Customer」テーブルを右クリックし、[**すべてのカラムを選択**] を選択します。

   照会結果を確認するには、[**クエリー実行**] ボタンをクリックします。

7. [**次へ**] ボタンをクリックして「**アプリケーションスタイル**」ダイアログボックスを表示します。このダイアログボックスでは、データを表示するビューの種類を選択できます。
8. 次のオプションを選択して [**次へ**] ボタンをクリックします。
   - データを単一レコードビューで表示
   - データをテーブルビューで表示
     - テーブル中の項目を単一レコードビューへホットリンク
     - 1 ページに 10 レコードずつページング
9. 「フォームの編集」ダイアログボックスでは、アプリケーションのユーザにデータベースの更新を許可するかどうかを指定します。[**はい**] オプションボタンをクリックし、「**挿入**」、「**更新**」、「**削除**」の各チェックボックスを選択します。[**次へ**] ボタンをクリックします。
10. 最後のダイアログボックスで、「**設定を省略値とする**」チェックボックスと「**ファイルを NetExpress プロジェクトに追加する**」チェックボックスが選択されていることを確認し、[**完了**] ボタンをクリックします。

インターネットアプリケーションウィザードで、次の各ファイルが生成されます。

- フォームビュー
  - **customerform.cbl** ― SQL コードを含むソースファイル
  - **customerform.cpf** ― フォーム内のコントロールの変数を含むコピーファイル
  - **customerform.cpy** ― ビジネスロジックの変数を含むコピーファイル

    - **customerform.cpv** ― ビジネスロジックとフォーム内のコントロール間で変数値を変換するコードを含むコピーファイル
    - **customerform.htm** ― フォームの HTML ファイル
    - **customerform.mff** ― フォーム編集用の Form Designer ファイル
- テーブルビュー
    - **customerlist.cbl** ― SQL コードを含むソースファイル
    - **customerlist.cpf** ― フォーム内のコントロールの変数を含むコピーファイル
    - **customerlist.cpy** ― ビジネスロジックの変数を含むコピーファイル
    - **customerlist.cpv** ― ビジネスロジックとフォーム内のコントロール間で変数値を変換するコードを含むコピーファイル
    - **customerlist.htm** ― フォームの HTML ファイル
    - **customerlist.mff** ― フォーム編集用の Form Designer ファイル

---

**注記**:

- 上記以外にも 2 つのファイル (**sqlca.cpy**、**sqlda.cpy**) がプロジェクトに追加されます。これらのファイルは埋め込み SQL の標準的なコピーファイルであり、編集する必要はありません。詳細については、Net Express オンラインヘルプの索引で「**sqlca**」と「**sqlda**」のエントリを見つけ、対応するページに記載されている情報を参照してください。
- データベースにアクセスするアプリケーションの生成で、データソースのテーブルに主キーや一意の索引キーが定義されている場合には、生成したアプリケーションでも、それらのキーを使用する必要があります。明示的に選択しない場合でも、これらのキーはインターネットアプリケーションウィザードによって自動的に選択され、その旨を通知するダイアログボックスが表示されます。
- 「**生成するファイルの名前を入力してください**」というレベルが付いたフィールドに入力するファイル名には、%、#、(、)、@ は使用しないでください。これらの記号を含むファイル名は、Web ブラウザや Net Express で正しく処理されない場合があります。

---

以上の手順を終了すると、データベースを照会するアプリケーションのファイル一式が生成されます。

---

**注記**: 異なる SQL 照会でアプリケーションを再生成する場合は、再生成の実行前に **.cbl** ファイルを削除してください。ビジネスロジックへの変更を維持するため、インターネットアプリケーションウィザードでは既存の **.cbl** ファイルは上書きされません。**.cbl** ファイルを削除しておけば、新しい **.cbl** ファイルが生成されます。

---

### 5.2.2 生成したアプリケーションの実行

ここでは、生成されたアプリケーションをビルドして実行します。

1. [**プロジェクト**] メニューで [**リビルド**] をクリックします。
2. [**アニメート**] メニューで [**実行**] をクリックします。
3. 「アニメーションの起動」ダイアログボックスで、実行可能プログラムの名前を次のように変更します。

   HTTP://127.0.0.1/cgi-bin/customerlist.exe

顧客リストのサーバ側プログラムが実行され、顧客データベース内の最初の 10 レコードが表示されます。[**<**] または [**>**] をクリックすると、前または次の 10 レコードを表示できます。先頭または末尾の 10 レコードを表示するには、[**<<**] または [**>>**] をクリックします。

単独の顧客レコード表示するには、左側の列にある顧客 ID (CustID) をクリックします。この操作を行うとフォームビューが表示され、レコードの編集やフィルタ処理オプションの変更を行うことができます。

たとえば、ニューヨーク州の全顧客を確認する手順は、次のとおりです。

1. [**画面をクリア**] ボタンをクリックします。
2. [**地域**] フィールドに **NY** と入力します。
3. [**フィルタ条件**] ドロップダウンリストで「**地域**」を選択します。
4. [**クエリー**] ボタンをクリックします。ニューヨーク州の最初の顧客のレコードが表示されます。

   表示されたビューでレコードを 1 件ずつ閲覧する代わりに、[テーブルビュー] をクリックして複数のレコードを一覧することもできます。テーブルビューでは、地域によるフィルタが適用されていることが、ステータス行に示されます。

レコードを追加するには、次の手順に従います。

1. フォームビューで [**画面をクリア**] ボタンをクリックします。
2. 顧客の詳細情報を入力します。
3. [**レコード挿入**] ボタンをクリックします。

   フォームが再表示され、レコードが正常に挿入されたことがステータス行に示されます。

レコードを更新するには、次の手順に従います。

1. フォームで住所など、データの一部を変更します。
2. [**レコードの更新**] ボタンをクリックします。

   フォームが再表示され、レコードが正常に更新されたことがステータス行に示されます。

レコードの削除方法は、次のとおりです。

1. フォームビューで [**レコードの削除**] ボタンをクリックします。

### 5.2.3 2 つ目のフォームセットの生成

ここでは、注文レコードの照会と更新に使用する 2 つ目のフォームセットを生成します。

1. インターネットアプリケーションウィザードを開きます。
2. **Wizard の入力**で「**SQL データベース**」を選択し、[**次へ**] をクリックします。
3. 次の情報を入力して [**次へ**] ボタンをクリックします。

   | | |
   |---|---|
   | ファイルのベース名 | **order** |
   | 生成されるフォームのタイトル | **注文の詳細情報** |

4. 「**データの選択**」ダイアログボックスで **UserSample2** をダブルクリックし、続いて「**Orders**」テーブルをダブルクリックして選択します。「Orders」テーブルを右クリックして [**すべてのカラムを選択**] を選択し、[**次へ**] をクリックします。
5. 「**アプリケーションスタイル**」ダイアログボックスで、フォームとして単一レコードビューとテーブルビューの両方を選択し、1 ページあたりのレコード数を前回と同様、10 レコードに設定します。
6. 「**フォームの編集**」ダイアログボックスで前回と同じ操作を行い、レコードの追加、更新、および削除をユーザに許可します。
7. ウィザードの最後のページで [**完了**] をクリックします。

   インターネットアプリケーションウィザードによって、注文情報を表示するためのファイルセットが生成されます。

8. アプリケーションをリビルドします。
9. このアプリケーションでフォームビューを実行します。
   - [**フィルタ条件**] を「**CustID**」に設定します。
   - [画面をクリア] ボタンをクリックします。
   - [**CustID**] フィールドに「**MERRG**」と入力して [**クエリー**] をクリックします。
   - [**テーブルビュー**] をクリックし、この顧客の注文リストを表示します。

次の項では、このアプリケーションをカスタマイズして、顧客と注文の照会フォームをリンクします。

### 5.2.4 アプリケーションのカスタマイズ

データベースにアクセスするサーバ側プログラムは COBOL で記述されているため、編集して機能を追加できます。また、HTML フォームの編集には Form Designer を使用できます。この項では、サンプルアプリケーションのカスタマイズ方法を説明します。

ここまでの操作で 2 対のフォームを作成しました。一方は顧客情報のデータベースを照会するフォーム、もう一方は注文情報のデータベースを照会するフォームです。以下の内容では、顧客照会フォームに顧客の注文リストへ直接リンクするプッシュボタンを追加します。

追加するプッシュボタンでサーバ側プログラム **orderform.exe** を起動し、表示している顧客の ID (CustID) を渡します。

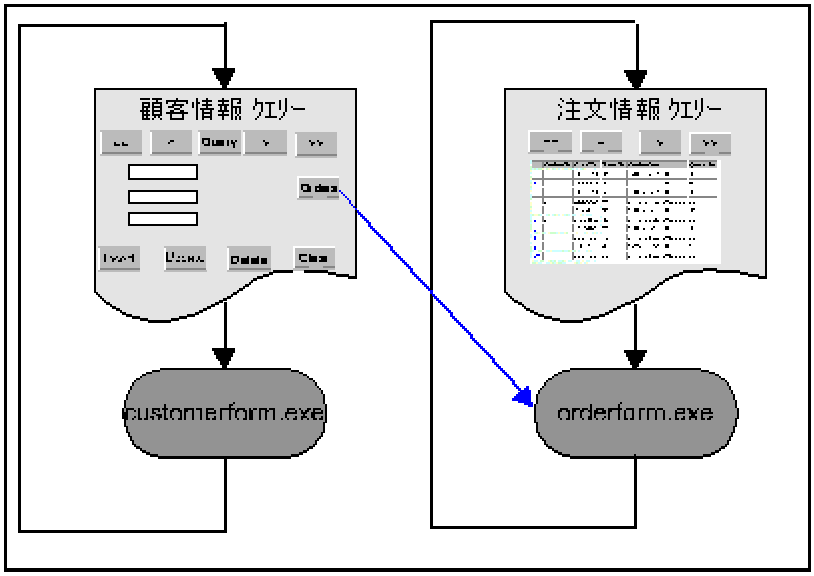



**図 30-1 顧客照会フォームから注文リストへのリンク**

この機能を追加するには、フォームとサーバ側プログラムをカスタマイズする必要があります。以下の 2 つの項では、この方法について説明します。

### 5.2.4.1 フォームのカスタマイズ

サーバ側プログラム **orderlist.exe** を起動するリンクを顧客照会フォームに追加します。**orderlist.exe** には、次の情報を渡す必要があります。

- フォームに表示されている顧客の ID (CustID)
- 顧客 ID でフィルタ処理する命令

サーバ側プログラムには、それを起動するリンクを通して情報を渡すことができます。この情報はフォームのコントロールから渡されるデータと同様、名前と値の組み合わせで渡されます。情報を渡すために必要なすべての変更は、**customerform.htm** と **orderlist.htm** を Form Designer で編集することによって実装できます。

必要な変更内容は次のとおりです。

- **customerform.htm** のフォームにプッシュボタンを追加し、ハイパーリンク経由で **orderlist.exe** を起動する JavaScript をスクリプトアシスタントで追加する。

注文リストを顧客ごとにフィルタ処理するための顧客 ID と命令も、このリンクで送信します。

- 受信した顧客 ID を格納するダミーのエントリフィールドを **orderlist.htm** に追加する。

  **orderlist.htm** を保存すると、**orderlist.cbl** でフォームから送信されたデータの受信に使用されるすべてのフォーム変数とコードが、Form Designer によって再生成されます。新しいデータも名前と値の組み合わせとしてサーバ側プログラムに渡されるため、サーバ側プログラムはハイパーリンクで起動したときに、このデータをフォームから送信されたデータと同じ方法で受信します。コマンド行で送信されるパラメータと同じ名前のダミーエントリフィールドを追加すると、顧客 ID の受信に必要なすべてのコードを自動生成できます。ダミーフィールドには、フォームに表示されない非表示フィールドを使用します。

**customerform.htm** は次の手順で編集します。

1. Net Express のプロジェクトウィンドウで **customerform.htm** をダブルクリックして Form Designer を起動し、フォームをロードします。
2. フォームに入力ボタンをドロップします (必ず入力ボタンを使用してください。誤って送信ボタンを使用すると、ボタンをクリックするたびに **customerform.exe** が再実行されます)。
3. ボタンのキャプションを「**注文**」に変更します。
4. 入力ボタンを右クリックし、コンテキストメニューで [**イベント**] をクリックします。
5. 「**Orders**」入力ボタンの **onclick** イベントを選択し、[**ハンドラの新規作成**] をクリックします。
6. 最下部のペインで中括弧の間に次の文字列を入力します。

   ```
   window.location.href="orderlist.exe?CustID=:A-CustID&Action=cquery"
   ```

   **注記**: コントロール名、CustID、および Action は、大文字と小文字が区別されます。ここに記載されているとおり、正確に入力してください。

   上記のように記述すると、**orderlist.exe?CustID=:f-A-CustID&Action=cquery** というハイパーテキストリンクを設定した場合と同じ処理が、プッシュボタンをクリックするたびに実行されます。送信文字列の最初の部分 (**orderlist.exe**) で、リソース **orderlist.exe** をサーバに要求します。疑問符 (?) は、それ以降の部分が名前と値のペアであることを示します。また、アンパサンド (&) は名前と値の区切りを示します。

   - CustID=:A-CustID は、CustID という名前と :f-A-CustID という値を持つパラメータによって送信することを示します。このフォームはサーバ側プログラム **customerform.exe** によって、EHTML を使用して出力されます。:f-A-CustID は EHTML の代入マーカーで、フォーム出力時に COBOL データ項目 f-A-CustID の値で置き換えられます。COBOL データ項目 f-A-CustID は COBOL プログラムによって、「CustomerID」フィールドの値を設定するために使用されます。その結果、HTML パラメータ CustID に Customer ID の値が設定されます。

フォームフィールドの値の設定に使用される COBOL 変数を識別するには、フォームで CustID のフィールドをクリックします。「CustID」フィールドの **Name** プロパティは A_CustID に設定されており、この設定によってサーバ側プログラム内に A-CustID と f-A-CustID という 2 つの変数が生成されます (アンダースコアは COBOL データ名には使用できないため、ハイフンに変換されます)。コントロールと同じ名前を持つ変数が、COBOL プログラム内での操作対象です。この変数には、コントロールの COBOLPicture プロパティで指定される PICTURE 文字列が格納されています。

先頭に f- が付いている変数は、フォームと COBOL プログラム間のデータ転記に使用されます。フォームデータは常に文字列リテラルであるため、この変数は PIC X(n) の PICTURE 文字列になります。生成されたコードには、入力用と出力用の 2 つのデータ項目間でデータを転記するためのルーチンセットと、必要な変換ルーチンが常に含まれています。

- Action=cquery は、Action という名前の HTML パラメータの値を cquery に設定します。

  **orderlist.htm** 内の各送信ボタンの Group name プロパティは、Action に設定されており、プログラムを実行するためにクリックした送信ボタンの値が、変数 Action としてサーバ側プログラムに渡されます。**orderlist.exe** は、起動後に COBOL データ項目 Action の値をチェックし、クリックされたボタンを特定します。この場合は Action=cquery の指定によって、cquery という値を持つボタンが特定されるため、Action が cquery の場合に Customer ID の値でフィルタ処理を行う照会のコードを orderlist.cbl に追加できます。

7. [**OK**] をクリックしてスクリプトアシスタントを閉じます。
8. 変更を保存します。

**orderlist.htm** のカスタマイズ手順は次のとおりです。

1. Form Designer にファイルをロードします。
2. フォーム内の任意の位置に非表示フィールドコントロールをドロップし、**Name** プロパティを CustID に設定します。COBOLPicture プロパティを PIC X(5) に設定します。これが **customerform.htm** に含まれる「CustID」フィールドの PICTURE プロパティの値になります。

   フォームを保存すると、CustID というデータ項目と、変数 CustID から値を読み込むためのコードが生成されます。

3. フォームへの変更を保存します。

   この操作によってフォームだけでなく、フォームのデータ入力と出力に関連するサーバ側プログラム (**orderlist.cbl**) 内のコードもすべて再生成され、CustID の値がある場合に、その値が COBOL 変数 CustID に格納されるようになります。

#### 5.2.4.2 サーバ側プログラムのカスタマイズ

続いて、サーバ側プログラム **orderlist.cbl** に変更を加える必要があります。このプログラムには、次の 2 つの項目を追加する必要があります。

- cquery アクションを検出するために EVALUATE 文に追加する条件
- CustID でフィルタ処理する SQL 照会に必要な変数を設定する新しい節

通常、これらの項目はサーバ側プログラム **orderform.exe** によって設定され、状態ファイルに格納されます。ただし、**orderlist.exe** が customerform から起動された場合には、このデータは設定されません。

サーバ側プログラムは次の手順でカスタマイズします。

1. テキスト編集ペインで **orderlist.cbl** を開きます。
2. 手続き部の見出しを見つけ、その数行下にある EVALUATE 文の位置まで移動します。
3. EVALUATE 文の WHEN OTHER 句の直前に新しい条件を追加します。

```
  when "cquery"
     perform DoQueryByCustomer
```

5. CustID の値でフィルタ処理するコードをプログラムに追加します。プログラムのメインセクションの後に、次の節を追加してください。

```
 DoQueryByCustomer section.
      move "A.CustID" to s-filter-field
                                    *> アプリケーションのロジックでは
                                    *> 大文字と小文字が区別されます。
                                    *> "A.CustID" は表示されている
                                    *> とおりに入力してください。
      move "=" to s-filter-op
      move custID to filter-A-Custid
      move "??" to search-op
      move low-values to sort-spec
      perform Do-SQL-Query
      perform SQL-Open
      perform Next-DataTable
      if no-data
          perform setup-status
          string
             " - no data to display" delimited size
             into frmesStatus pointer stat-index
      end-if
     exit.
```

25. 変更を保存します。

上記のコードは、「CustID」フィールドの値でフィルタ処理するデータベースクエリーを Do-SQL-Query で作成するための変数を設定しています。それぞれの変数には次のような意味があります。

| 変数 | 説明 |
|---|---|
| s-filter-field | フィルタを適用するカラム。値は "A.*columnname*" です。先頭の "A." は、将来的に機能を拡張し、テーブルを結合するクエリーを作成できることを示します。 |
| s-filter-op | 比較演算子。"="、">="、">"、"<="、"<"、"!=" のいずれかを設定できます。 |
| filter-A-CustID | フィルタ条件として使用する CustID の値。データウィザードで作成したクエリーの各カラムに対応するフィルタフィールドがあります。フィルタフィールド名は *columnname* です。 |
| search-op | 検索方向。"??" は、最初の合致データを検索するための命令です。"?>" と "?<" は、それぞれ次と直前の合致データを検索するための命令です。次と直前の合致は、検索順序のカラムと、アプリケーションの状態レコードの内容によって定義されます。 |
| sort-spec | テーブルのソートの有無。DESC または空白文字を設定します。 |

詳細については、『生成 CGI コードリファレンス』を参照してください。[**ヘルプ > ヘルプトピック**] をクリックし、「**目次**」タブで [**リファレンス > 生成された CGI コード**] をダブルクリックします。

#### 5.2.4.3 カスタマイズしたアプリケーションの実行

カスタマイズしたアプリケーションは、次の手順で実行します。

1. [**プロジェクト**] メニューの [**リビルド**] をクリックして、変更したプログラムを再コンパイルします。
2. [**アニメート**] メニューで [**実行**] をクリックします。
3. 「アニメーションの起動」ダイアログボックス内のフィールドを次のように変更し、[OK] をクリックします。

   ```
   http://127.0.0.1/cgi-bin/customerform.exe
   ```

   Web ブラウザに、顧客フォームビューが表示されます。

4. このフォームでは、カスタマイズ前と同様、顧客の詳細情報を表示したり、更新することができます。
5. 特定の顧客の注文を確認するには、[**注文**] ボタンをクリックします。

   選択した顧客の注文リストが表示されます。

# 第 31 章 : レガシーコードの再利用

この章では、既存の COBOL サブルーチンを再利用してインターネットアプリケーションを作成する方法を説明します。

## 6.1 概要

インターネットアプリケーションウィザードを使用すれば、既存の COBOL サブルーチンを速やかにインターネットアプリケーションに変換できます。連絡節でパラメータを受け付け、結果を返すようにコーディングしている COBOL プログラムの大部分が再利用可能です。インターネットアプリケーションウィザードで生成されるのは、フォームから送信された入力値を受信し、その値でサブルーチンを呼び出す仲介型のプログラムです。このプログラムはさらに、呼び出したプログラムから返された処理結果を元のフォーム (または別のフォーム) に返信し、表示させます。レガシーアプリケーションには、既存のユーザインターフェイスロジックを他のロジックから分離して、入力パラメータと出力パラメータに連絡節を介してアクセスできるように、多少の変更が必要になる場合もあります。

インターネットアプリケーションウィザードを実行して「**ウィザードへの入力**」で「**COBOL ソースファイル**」を選択すると、インターネットアプリケーションのベースとして使用する COBOL プログラムの選択を求めるメッセージが表示されます。ソースファイルを選択すると、連絡節で宣言されている全データ項目が表示されます。同じデータ項目は、手続き部見出しの USING 句でも指定されています。これらのデータ項目から、入力パラメータ、出力パラメータ、または入出力パラメータとして使用する項目を選択します。

以上の操作を行うと、インターネットアプリケーションウィザードによって、次のフォームとプログログラムがアプリケーション用に生成されます。

- 入力フォーム
- 出力フォーム (入力フォームと同じフォームを使用するように設定した場合を除く)
- フォームとサブルーチン間の通信を仲介する COBOL プログラム

フォームの形式として HTML と DHTML (Dynamic HTML) のいずれか一方を選択できます。HTML はクロスプラットフォーム環境に適しています (両者の違いについては、『フォームと HTML』の章を参照してください)。インターネットアプリケーションウィザードで生成された基本的なフォームには、Form Designer を使用して変更を加えることができます。

サブルーチンを使用するための手順を次に示します。

1. 新しい Net Express プロジェクトを作成し、サブルーチンをプログラムファイルとして追加します。
2. インターネットアプリケーションウィザードを起動し、「**ウィザードへの入力**」で「**COBOL ソースファイル**」を選択します。
3. インターネットアプリケーションウィザードに表示されるメッセージに従って、必要な情報を入力します。

4. ウィザードの「**パラメータの割り当て**」ページで、入力パラメータと出力パラメータを選択します (プログラムの連絡節で指定されているパラメータのリストが表示されます)。パラメータに対応するコントロールの種類と、コントロールを表示するフォーム (入力フォーム、出力フォーム、またはその両方) を、パラメータごとに選択できます。
5. ウィザードの最後のページで [**完了**] をクリックします。

   以上の手順でアプリケーション用のファイルが生成され、Net Express プロジェクトに追加されます。

6. インターネットアプリケーションウィザードで生成されたフォームに、必要に応じて Form Designer で変更を加えます。
7. アプリケーションの機能を調整したり、ステップ 6 でフォームに追加したコントロールの処理を記述する必要があるときには、生成されたコードを編集します。
8. アプリケーションをリビルドします。

以下の内容では、この手順の各ステップについて説明します。インターネットアプリケーションウィザードでプログラムを生成し、生成したプログラムを編集する手順は、具体例を通じて示しています。 この章では、[OK] ボタンをどこでクリックするか、という細かいレベルの説明ではなく、全体的な手順の流れを示すことに重点を置いています。各手順の細かい説明については、オンラインヘルプを参照してください。作業に着手する前に、インターネットアプリケーションウィザードの操作に慣れておきたい方は、『入門書 - その他のトピック』のサンプルセッションを利用できます。

## 6.2 インターネットアプリケーションの作成

インターネットアプリケーションの作成に着手する前に、次の点を決定しておく必要があります。

- フォームの形式としてクロスプラットフォーム形式と DHTML のどちらを使用するか。
- 入力と出力に同じフォームを使用する対称型アプリケーションと、それぞれ別のフォームを使用する非対称型アプリケーションのどちらを採用するか。

  対称型アプリケーションと非対称型アプリケーションについては、『インターネットプログラミングの概要』の章を参照してください。

以下の説明は、全体的な手順の流れを示すことに重点を置いています。各手順の詳しい説明については、Net Express のオンラインヘルプを参照してください。 インターネットアプリケーションウィザードの操作に慣れておきたい方は、『入門書 - その他のトピック』のサンプルセッションを利用できます。

インターネットアプリケーションウィザードでアプリケーションを生成する手順は、次のとおりです。

1. Net Express プロジェクトを作成し、再利用するレガシーサブルーチンのソースファイルをプロジェクトに追加します。
2. インターネットアプリケーションウィザードを開きます。

3. 「**ウィザードへの入力**」で「**COBOL ソースファイル**」を選択します。
4. アプリケーションに再利用するサブルーチンを含むファイルの名前の入力を求めるメッセージが表示されます。ファイル名を直接入力するか、[**参照**] ボタンをクリックしてファイルを選択します。
5. アプリケーションのインターフェイスとして、入力と出力に同じフォームを使用する対称型と、それぞれ別のフォームを使用する非対称型のいずれかを選択し、さらにフォームの形式としてクロスプラットフォーム形式と DHTML のいずれかを選択します。
6. 入力フォームと出力フォームのタイトルの入力を求めるメッセージが表示されます。また、生成する各ファイルのデフォルトのファイル名も表示されます。デフォルトのファイル名は変更できます。
7. 生成するフォームの HTML コントロールとプログラムのパラメータのマッピングを求めるメッセージが表示されます。

   この手順は、次項の『データの選択』で詳しく説明しています。

8. [完了] をクリックします。フォームと COBOL サーバ側プログラムが生成されます (生成されたフォームは、このサーバ側プログラムを介して元の COBOL プログラムと連携します)。

## 6.3 データの選択

インターネットアプリケーションウィザードの「パラメタの割り当て」ページは、次のように表示されます。


**図 31-1 インターネットアプリケーションウィザードの「パラメタの割り当て」ページ**

**ページ左側には、選択したプログラムの連絡節の各データ項目 (手続き部見出しの USING 句に指定されているデータ項目) が、次の情報とともにツリー形式で表示されます。**

| フィールド | 説明 |
|---|---|
| レベル番号 | データ項目のレベル番号。ここに表示されるレベル番号は、必ずしもソースコード内のレベル番号と一致しているとは限りません。グループ項目は、横に表示されるプラス記号 (+) で識別できます。<br><br>このプラス記号 (+) をクリックするとグループ項目が展開され、そのグループ内の副項目の階層が表示されます。<br><br>たとえば、01、02、04 の各レベルを持つグループ項目を展開すると、レベル 01、03、05 にグループ項目が表示されます。 |
| フィールド名 | 表示しているサブルーチン内の COBOL データ項目の名前 |

**データ型**　　**データ項目の COBOL PICTURE 文字列**

**テーブルのデータ項目は、Occurs *n* の形式で表示されます (*n* は項目の OCCURS 値)。再定義されるデータ項目は Redefined、他のデータ項目を再定義するデータ項目は Redefines という文字列で示されています。**

**右側の「パラメタの割り当て」ペインには、データとフォーム内のコントロールのマッピングに関する情報が表示されます。.**

**ラベル名**　　**データ項目に対応するフォーム内のコントロールのキャプション**

**コントロールタイプ**　　**データ項目に対応するフォーム内のコントロールの種類**

**インターネットアプリケーションウィザードで生成できるコントロールは、テキストエントリフィールド (このペイン内での表示は Edit)、オプションボタン (RadioButton)、チェックボックス (CheckBox) の 3 種類です。**

**データ方向 (I/O)**　　**データの送信方向。フォームからプログラムへの送信 (入力パラメータ)、プログラムからフォームへの送信 (結果)、または両方向。「I/O」欄をクリックし、リストから次のいずれかの項目を選択します。**

| | |
|---|---|
| **N/A** | **フォームで使用しないデータ項目** |
| **In** | **入力パラメータ。入力と出力に別のフォームを使用するアプリケーションでは、当該フィールドは入力フォームだけに表示されます。** |
| **Out** | **処理結果。入力と出力に別のフォームを使用するアプリケーションでは、当該フィールドは出力フォームだけに表示されます。** |
| **Both** | **入力パラメータとしてプログラムに渡され、処理結果として返されるデータ項目。入力と出力に別のフォームを使用するアプリケーションでは、当該フィールドは入力、出力の両方のフォームに表示されます。** |

**初期状態では、すべてのパラメータのデータ方向が Both (グループ項目の場合は N/A) に設定されています。アプリケーションを作成するには、ユーザがフォームで閲覧または編集するすべてのデータ項目について、ラベル、コントロールタイプ、およびデータ方向を指定する必要があります。**

このページで設定した内容は、割り当てファイル (**.aht** ファイル) として保存できます。アプリケーション生成後に設定内容の変更が必要になった場合には、同じ **.cbl** ファイルを使用してインターネットアプリケーションウィザードをもう一度開始し、保存した **.aht** ファイルを再ロードすれば、前回の設定結果から作業を開始できます。

### 6.3.1 エントリフィールドへのフィールドの割り当て

テキストエントリフィールドの使用方法は非常にシンプルです。フォームからプログラムにデータを送信する際に、エントリフィールドの内容が、対応する COBOL データ項目に格納されます。インターネットアプリケーションウィザードで生成されるコードには基本的な検証機能が含まれており、COBOL の数値データ項目に対応するエントリフィールドに入力されたデータが正しく変換できないと、Web ブラウザにエラーメッセージが表示されます。

プログラムからフォームにデータが返信される際には、データ項目の内容がエントリフィールドに表示されます。インターネットアプリケーションウィザードでは、数値データをエントリフィールドに表示可能なリテラルに変換するためのコードも生成されます。

パスワードフィールドは、テキストエントリフィールドとまったく同じように処理されます。両者の唯一の違いは、パスワードフィールドに入力された文字がアスタリスク (*) として表示されるという点です。ラベルと編集不可フィールドも、エントリフィールドと同様に扱われますが、エンドユーザがフィールドの内容を変更することはできません (編集不可フィールドは、Web ブラウザによってはサポートされていない場合があります)。また、Web ブラウザには表示されない非表示フィールドと呼ばれるフィールドもあります。

### 6.3.2 オプションボタンへのフィールドの割り当て

単一の COBOL データ項目には、オプションボタンのグループを割り当てることができます。それぞれのオプションボタンは特定の値に関連付けられており、フォームからプログラムにデータが送信される際に、選択されているオプションボタンの値が COBOL データ項目に格納されます。また、プログラムからフォームにデータが返信される際には、COBOL データ項目の値と一致するオプションボタンが選択されます。オプションボタンとその値を設定するには、「**コントロールタイプ**」フィールドで [**Selects**] をクリックし、表示される「**コントロールマップ**」タブを使用します。

オプションボタンには、対応するデータ項目に適した値を割り当ててください。たとえば、数値データ項目に対応する 3 つのオプションボタンに、それぞれ "X"、"Y" "Z" を割り当てると、どのオプションボタンを選択してもランタイムエラーメッセージが表示されてしまいます。

### 6.3.3 チェックボックスへのフィールドの割り当て

COBOL データ項目にはチェックボックスを割り当てることもできます。データがフォームからプログラムに送信される際に、チェックボックスが選択されていると、そのチェックボックスに関連付けられた値が COBOL データ項目に格納されます。一方、データがプログラムからフォームに送信される際には、COBOL データ項目とチェックボックスの値が一致すれば、チェックボックスが選択された状態で表示されます。チェックボックスのラベルと値を設定するには、「**コ**

**ントロールタイプ**」フィールドで [**Checkbox**] をクリックし、表示される「**コントロールマップ**」タブを使用します。

### 6.3.4 COBOL テーブルの割り当て

OCCURS 句付きで宣言した COBOL データ項目に割り当てることができるコントロールは、テキストエントリフィールドだけです (パスワードフィールド、非表示フィールド、ラベルフィールド、および編集不可フィールドも含まれます)。OCCURS データ項目がグループ項目の場合、そのグループ内の副項目にも同じコントロールのサブセットを使用できます。

COBOL テーブル内の各コントロールは HTML のテーブル内に、OCCURS 句の値と同じ数だけ繰り返して配置されます。

### 6.3.5 選択コントロールへのフィールドの割り当て

COBOL データ項目には選択コントロールも割り当てることができます。選択コントロールには、一連のラベルと値のペアを設定します。Web ブラウザでは選択コントロール内にラベルが表示され、フォームの送信時に選択されたラベルに対応する値が COBOL データ項目に渡されます。

## 6.4 アプリケーションの生成

アプリケーションに必要なデータの選択が完了すれば、アプリケーションのファイルを生成できます。インターネットアプリケーションウィザードの最後のページで [**完了**] をクリックすると、次の表に示す各ファイルが生成されます。

この表中のファイル名は、インターネットアプリケーションウィザードがデフォルトで割り当てる名前です。これらの名前は、ウィザードの途中のページで変更できます。

| ファイル | 説明 |
|---|---|
| *program*_server.cbl | フォームと元のサブルーチンを仲介するプログラム |
| *program*_server.cpf | フォームで送受信するデータのコピーファイル |
| *program*_server.cpl | サブルーチンのパラメータデータ |
| *program*_server.cpv | Web ブラウザとサブルーチン間でデータを変換するルーチン |
| *program*_server.cpy | アプリケーションデータ |
| *program*_input.htm | アプリケーションの入力フォーム |
| *program*_input.mff | 入力フォーム用の Form Designer ファイル (このファイルを Form Designer にロードすれば、インターネットアプリケーションウィザードで |

生成されたフォームのレイアウトを変更できます)

***program*_output.htm** アプリケーションの出力フォーム (非対称型アプリケーションを選択した場合のみ生成されます)

***program*_output.mff** 出力フォーム用の Form Designer ファイル (非対称型アプリケーションを選択した場合のみ生成されます。このファイルを Form Designer にロードすれば、インターネットアプリケーションウィザードで生成されたフォームのレイアウトを変更できます)

インターネットアプリケーションウィザードでファイルが生成されると、実際に使用可能なアプリケーションが完成します。アプリケーションが生成されると、Form Designer でフォームに変更を加えたり、生成されたコードを編集することができます。

# 6.5 サンプルアプリケーションの作成

この項では、インターネットアプリケーションウィザードを使用して、2 つのサブルーチンに基づく小規模なアプリケーションを作成します。最初のサブルーチン (**b-write.cbl**) が索引ファイルにレコードを書き込み、もう 1 つのサブルーチン (**b-read.cbl**) がキーを基にレコードを検索して取り込みます。各レコードは書籍情報のエントリで、次のフィールドから構成されています。

- Title (副キー)
- Author (副キー)
- Type
- Reprinting Date
- ISBN (主キー)

この章では、[OK] ボタンをどこでクリックするか、という細かいレベルの説明ではなく、全体的な手順の流れを示すことに重点を置いています。インターネットアプリケーションウィザードや Net Express の詳しい操作手順については、オンラインヘルプを参照してください。インターネットアプリケーションウィザードの操作に慣れておきたい方は、『入門書 - その他のトピック』のサンプルセッションを利用できます。

2 つのサブルーチンは **Net Express\base\demo\bookapp** にあります。最初に、ファイルに書き込みを行うアプリケーションとフォームを作成します。

1. Net Express を起動し、**Net Express\base\demo\bookapp** フォルダにある **bookwrit.app** プロジェクトをロードします。
2. 新しいインターネットアプリケーションを作成します。インターネットアプリケーションウィザードで、次のオプションを選択してください。
    - COBOL ソースファイル
    - COBOL 原始プログラムとして **b-write.cbl** を選択
    - 単一入出力フォーム
    - DHTML
    - 入力タイトルとして「書籍レコードの追加」を入力

3. インターネットアプリケーションウィザードの「**パラメタの割り当て**」ページには、01 レベルのデータ項目が 2 つ表示されます。


**図 31-2 インターネットアプリケーションウィザードに表示された連絡節のデータ項目**

4. LNK-FILE-STATUS のラベルを File Status に変更し、データ方向を **Out** に設定します。
5. ツリービューで LNK-B-DETAILS の横にある **+** をクリックして、副項目を表示します。
6. [**パラメタの割り当て**] タブのフィールドラベルを、上から順に次のように設定します。
    - Title
    - Type
    - Author
    - Reprinting Date
    - ISBN
7. すべてのフィールドのデータ方向を **In** に設定します。
8. ツリービューで任意のエントリを右クリックし、ポップアップメニューで [**割り当てファイルの保存**] を選択します。ファイル名を入力して [**OK**] をクリックします。

この操作を行うと、実行したデータ割り当てがすべて保存されます。同じアプリケーションを再生成するときには、COBOL サブルーチンをロードした後、ここで保存した割り当てファイルを再ロードします。割り当てファイルの基本名がサブルーチンファイルと同じ場合には、インターネットアプリケーションウィザードによって割り当てファイルが自動的に再ロードされます。

9. ウィザードの最後のページで [**完了**] をクリックします。

   インターネットアプリケーションウィザードによってファイルが生成され、プロジェクトに追加されます。

10. Net Express の [**プロジェクト**] で [**リビルド**] をクリックし、アプリケーションをビルドします。

続いて、ファイルをロードするアプリケーションを作成します。

1. **Net Express\base\demo\bookapp**. フォルダにある **bookread.app** プロジェクトをロードします。
2. 新しいインターネットアプリケーションを作成します。インターネットアプリケーションウィザードで、次のオプションを選択してください。
   - COBOL ソースファイル
   - COBOL 原始プログラムとして **b-read.cbl** を選択
   - 単一入出力フォーム
   - DHTML
   - 入力タイトルとして「書籍レコードの読み取り」を入力
3. LNK-FILE-STATUS のラベルを File Status に変更し、データ方向を **Out** に設定します。
4. ツリービューで LNK-B-DETAILS の横にある **+** をクリックして、副項目を表示します。
5. [**パラメタの割り当て**] タブでフィールドラベルを次のように設定します。
   - Title
   - Type
   - Author
   - Reprinting Date
   - ISBN
6. すべてのフィールドのデータ方向を **Both** に設定します (1 つのフィールドを検索キーとして入力すると、プログラムが該当するエントリの全フィールドのデータを返します)。
7. 割り当てを **b-read.aht** として保存します。
8. [**終了**] をクリックします。
9. Net Express でアプリケーションをリビルドします。

以上の手順を終了すると、書籍レコードファイルの更新と読み取りを行うシンプルなアプリケーションが完成します。

### 6.5.1 サンプルアプリケーションのテスト

この項では、作成したサンプルアプリケーションをテストし、実際にレコードの追加と読み取りを実行します。

1. アプリケーションを起動します。Web サーバと Web ブラウザを起動して Web ブラウザにアプリケーションの URL を入力するか、Net Express のアニメータで起動します。
   - Web サーバとブラウザでアプリケーションを起動する手順は次のとおりです。
     1. Solo Web サーバーを起動します。

Solo 以外の Web サーバを使用する方法については、[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの [**目次**] タブで「**プログラミング > CGI ベースのアプリケーション > CGI ベースのアプリケーションのデバッグ > 方法 > CGI ベースのアプリケーションのアニメート**」を順にダブルクリックして、表示される説明を参照してください。

2. Web ブラウザのアドレスフィールドに次の URL を入力します。

   ```
   http://127.0.0.1/cgi-bin/b-write_server.exe
   ```

- Net Express のアニメータでアプリケーションを起動するには、[**アニメート > 実行**] をクリックします。ソースコードにブレークポイントを設定しておけば、アプリケーションの実行がその位置で停止し、コードのデバッグを行うことができます。

どちらの方法でアプリケーションを起動しても、Web ブラウザには次のようなフォームが表示されます。

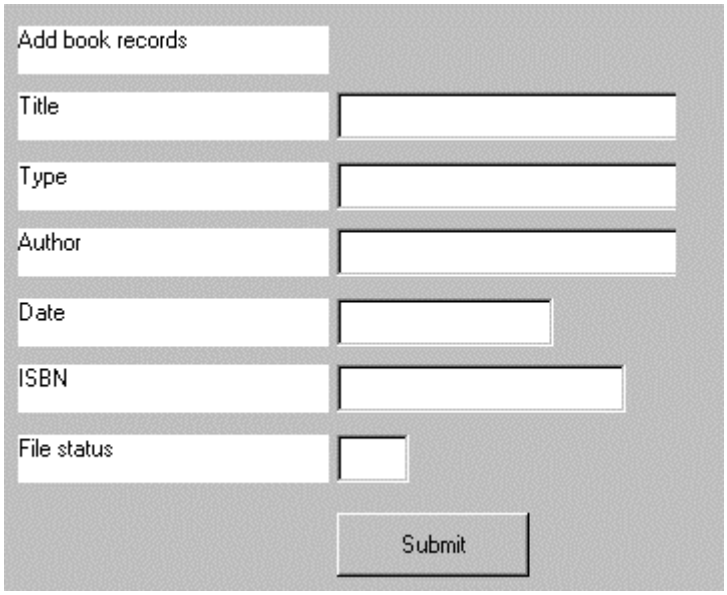



**図 31-3「書籍レコードの追加」フォーム**

2. フォームの各フィールドに次の情報を入力し、[**Submit**] ボタンをクリックします。

| | |
|---|---|
| **Title** | **Moby Dick** |
| **Type** | **Fiction** |
| **Author** | **Melville** |
| **Reprinting Date** | **8feb96** |

ISBN 1

3. [Submit] ボタンをクリックすると、しばらくして大文字に変換された入力値を含むフォームが表示されます。b-write サブルーチンは、b-read サブルーチンによるキー照合を効率化するため、すべてのフィールドを大文字に変換します。レコードが正常に追加されると、ファイル状態フィールドに「00」が表示されます。
4. さらに数冊の書籍に関する情報を入力し、レコードとして追加します。

必要な数のレコードを追加した後、レコードの検索を開始します。

1. Web ブラウザのアドレスフィールドに次の URL を入力します。

   ```
   http://127.0.0.1/cgi-bin/b-read_server.exe
   ```

   レコードを追加したフォームに似た別のフォームが表示されます。

2. 「ISBN」フィールドに 1 と入力し、[Submit] ボタンをクリックします。

   この操作の結果、*Moby Dick* のレコードが表示されます。

# 6.6 アプリケーションの拡張

これまでの項では、フォームから直接取得したデータを 1 つのサブルーチンで処理する基本的なアプリケーションの作成方法を示しました。この項では、インターネットアプリケーションウィザードで生成されたフォームと COBOL プログラムを編集して、アプリケーションの機能を拡張する方法について説明します。

## 6.6.1 COBOL プログラムの編集

インターネットアプリケーションウィザードは、アプリケーションの生成時に次の COBOL ソースコードファイルを生成します。

| ファイル | 説明 |
|---|---|
| *program*_server.cbl | フォームと元のサブルーチンを仲介するプログラム |
| *program*_server.cpf | フォームで送受信するデータのコピーファイル |
| *program*_server.cpl | サブルーチンのパラメータデータ |
| *program*_server.cpv | Web ブラウザとサブルーチン間でデータを変換するルーチン |
| *program*_server.cpy | アプリケーションデータ |

編集するファイルは *program*_server.cbl だけです。他のファイルは、いずれもインターネットアプリケーションウィザードで同じアプリケーションを再生成するたびに自動的に上書きされます。*program*_server.cbl ファイルには、次の情報が格納されています。

- フォームとサブルーチンに関する全データの宣言
- フォームの入力データを受け付けるロジック
- フォームの入力データを英数字文字列からレガシーサブルーチンで正しく処理できる形式に変換するロジック
- サブルーチンの呼び出し (CALL)
- サブルーチンの出力を表示可能な英数字に変換するロジック
- フォームを出力するロジック

このプログラムには、サブルーチン呼び出し (CALL) の前後どちらにも独自のアプリケーションロジックを追加できます。インターネットアプリケーションウィザードによって、これらの場所を示すコードに "TO DO" というコメントが挿入されます。

### 6.6.2 フォームの編集

インターネットアプリケーションウィザードで生成されたフォームのデフォルトレイアウトは、フォームを Form Designer にロードして変更できます。また、フォームにコントロールを追加することもできます。コントロールを追加した場合には、インターネットアプリケーションウィザードで生成された COBOL プログラムにロジックを追加する必要があります。

フォームを編集するには、Net Express の「プロジェクト」ウィンドウで、フォームの .htm ファイルをダブルクリックします。この操作によって Form Designer が起動し、フォームがロードされます。フォームにコントロールを追加して保存すると、そのコントロールに対応するデータ項目がプログラムのコピーファイルに追加されます。

### 6.6.3 サンプルアプリケーションの拡張

この章でビルドしたサンプルアプリケーションでは、ファイル状態が 2 文字のコードとしてエントリフィールドに表示されるため、このコードの意味がわからないとファイル状態を見分けることができません。この項では、次のいずれかの状態通知メッセージを返すように、レコード読み取り用アプリケーションを変更します。

- Record found (レコードが見つかった場合)
- Record not found (レコードが見つからなかった場合)
- File operation failed (上記 2 つに該当しないファイル状態)

変更の大まかな手順は次のとおりです。

1. インターネットアプリケーションウィザードで、サブルーチン **b-read** への割り当てから LNK-FILE-STATUS フィールドを除外します。

   アプリケーションを再生成すると、このフィールドはフォームに存在しなくなります。

2. **b_read_server.htm** を Form Designer にロードし、メッセージ用の新しいフィールドを追加します。
3. サブルーチンから返されるファイル状態コードを評価し、フォームに適切なメッセージを返すコード **b_read_server.cbl** を追加します。

#### 6.6.3.1 インターネットアプリケーションウィザードでの割り当ての変更

インターネットアプリケーションウィザードで、次の手順に従って **b-read.cbl** への割り当てを変更します。

1. Net Express を起動し、**Net Express\base\demo\bookapp** フォルダにある **bookread.app** プロジェクトをロードします。
2. 新しいインターネットアプリケーションを作成します。インターネットアプリケーションウィザードで、次のオプションを選択してください。
   - COBOL ソースファイル
   - COBOL 原始プログラムとして **b-read.cbl** を選択
   - 単一入出力フォーム
   - DHTML
   - 入力タイトルとして「書籍レコードの読み取り」を入力
3. 「パラメタの割り当て」ページのツリービューで任意の項目を右クリックします。ポップアップメニューで [**割り当てファイルをロード**] をクリックし、「**ファイルを開く**」ダイアログボックスで **b-read.aht** を選択します。

   このアプリケーションを最初に作成したときに割り当てた内容が再ロードされます。

4. LNK-FILE-STATUS の「I/O」欄の値を N/A に変更します。
5. ウィザードの最後のページで [**完了**] をクリックします。

このアプリケーションのコピーファイルとフォームが、LNK-FILE-STATUS フィールドを除外した状態で再生成されます。

#### 6.6.3.2 メッセージフィールドの追加

次の手順でフォームに新しいフィールドを追加します。

1. Net Express の「プロジェクト」ウィンドウで **b-read_server.htm** をダブルクリックし、Form Designer を起動します。
2. フォームの最下部にテキストボックスを追加します。
3. 構成要素ツリービューで ID を resultfield に変更します。
4. プロパティを次のように設定します。
   - MaxLength を 30 に設定します。
   - COBOL Picture を X(30) に設定します。
5. 変更を保存し、Form Designer を閉じます。

#### 6.6.3.3 サンプルプログラムの変更

追加したフィールドにエラーメッセージを返すように、次の手順でプログラムを変更します。

1. Net Express の「プロジェクト」ウィンドウで **b-read_server.cbl** をダブルクリックし、編集を開始します。
2. コード内で "Add post-call application business logic here" というコメントがある位置まで移動します。
3. このコメントの後に、次のコードを追加します。

```
4.   evaluate lnk-file-status
5.       when "00"
6.           move "Read OK" to resultfield
7.       when "23"
8.           move "Not found" to resultfield
9.       when other
10.          move "Operation failed" to resultfield
11.  end-evaluate
```

12. プロジェクトをリビルドします。
13. アプリケーションを再実行します。

    以上の操作を行うと、アプリケーションがファイル状態を示すメッセージを返すようになります。

---



---

# 第 32 章 : サーバ側のプログラミング

この章では、インターネットアプリケーションのサーバ側プログラムを作成する方法について説明します。ここで説明する内容は、ISAPI、NSAPI、CGI の違いに関係なく、サーバ側プログラムの作成全般に適用されます。

## 7.1 概要

Net Express では COBOL の機能が拡張されており、他の開発言語を使用する場合に比べ、はるかに容易にインターネットアプリケーションのサーバ側プログラムを作成できます。フォームのデータは ACCEPT 動詞で受信し、処理結果は DISPLAY 動詞か埋め込み HTML (EHTML) で Web ブラウザに送信できます。EHTML を使用すれば、COBOL プログラム内で動的に HTML ページを生成できます。この章では、主に次の 4 つのトピックについて説明しています。

- リソースの競合
- サーバ側プログラムへの入力
- サーバ側プログラムからの出力
- アプリケーション状態の保持

Form Designer で HTML を作成すれば、データを読み取って結果を返すサーバ側プログラムのスケルトン (ひな型) を、インターネットアプリケーションウィザードで生成できます。こうして生成されたスケルトンには、必要に応じてデータを処理するコードを追加できます。このように Net Express では、入力を受信して出力を返すための構文の詳しい知識がなくてもインターネットアプリケーションを作成できます (ただし、より高度なアプリケーションを作成する場合には、そのような知識が役立ちます)。

インターネットアプリケーションウィザードで生成されたサーバ側プログラムは、ACCEPT を使用してフォームからのデータ入力を受信し、EHTML を使用してフォームを出力します。

## 7.2 リソースの競合

サーバ側プログラムは、複数のユーザが並行して実行できます。サーバ側プログラムが CGI プログラムの場合、プログラムは呼び出されるたびに別プロセスとして動作します。一方、ISAPI や NSAPI のサーバ側プログラムは、呼び出されるたびに別スレッドで動作します。サーバ側プログラムがファイルやデータベースなどの共有リソースにアクセスする場合には、リソース競合に対処するためのロジックを含める必要があります。

COBOL には共有ファイル関連の多彩な関数群があります。 詳細については、『*ファイル操作*』の『ファイルの共有』の章を参照してください。

## 7.3 サーバ側プログラムへの入力

フォーム内の各コントロールには名前と値があります。『フォームと HTML』の章で説明したように、エンドユーザによるフォームの送信時には、対になった名前と値の集まりとしてフォーム内の情報がサーバ側プログラムに送信されます。COBOL の拡張構文を使用すると、フォーム内のコントロールの名前を、サーバ側プログラム内の COBOL データ項目に直接関連付けることができます。この関連付けを行えば、フォーム内のデータがサーバ側プログラムに渡されるときに、コントロールの値が対応するデータ項目の値として設定されます。

次のフォームには name という名前のフィールドがあり、エンドユーザによって値 "Bob" が入力されています。

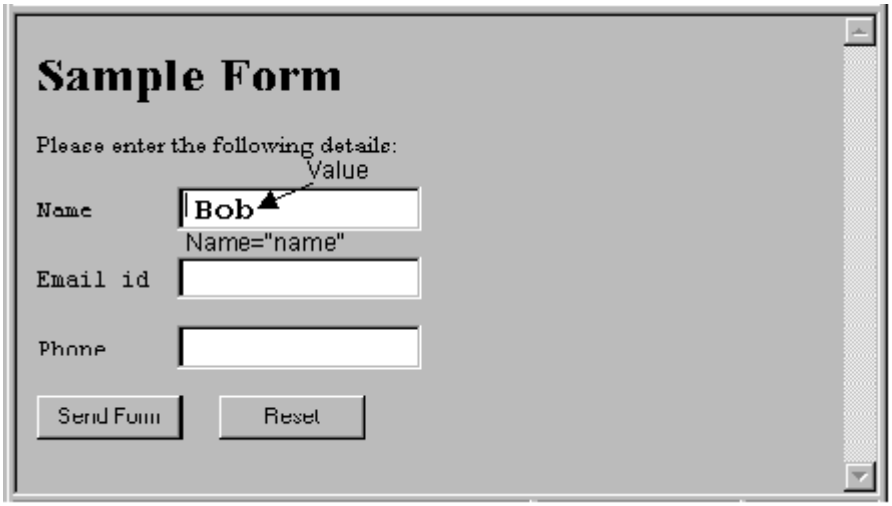



**図 32-1 フォーム内のフィールドの名前**

サーバ側プログラムは、作業場所節の次のような宣言を通じて、名前を持つ各コントロールと関連付けられます。

```
01 inputdata is external-form.
   03 name-field      pic x(30) identified by "name".
   03 email           pic x(15) identified by "emailid".
   03 phone-no        pic x(30) identified by "phone".
```

エンドユーザが [**Send Form**] ボタン (Submit コントロール) をクリックするとサーバ側プログラムが起動し、次の文が実行されたときに値 "Bob" がデータ項目 NAME-FIELD に格納されます。

```
accept input-form
```

Form Designer でフォームを作成し、サーバ側アプリケーションのスケルトンを自動生成すると、フォーム内の各コントロールに対応するデータ項目の宣言を含むコピーファイルも同時に生成されます。

### 7.3.1 構文

COBOL データ項目を CGI 入力にマッピングするには、次の構文を使用します。

*level-number data-name-1* IS EXTERNAL-FORM.

この宣言は、グループ項目 *data-name-1* とその中の各フィールドに、HTML フォームから入力された値を設定できることを示しています。基本データ項目を NAME 属性にマッピングするには、次の構文を使用します。

*sub-level-number data-name-2 picture-string*
IDENTIFIED BY *name*.

各パラメータの意味は次のとおりです。

*sub-level-number* COBOL データ項目のレベル番号

*data-name-2* COBOL データ名

*picture-string* COBOL PICTURE 文字列。PIC X(*n*)、PIC 9(*n*)、または数値編集フィールドです。*n* 文字より長い文字列は超過部分が切り捨てられ、*n* 文字より短い文字列は左側から不足文字数だけ空白文字でパディングされます。

*name* フォーム内のコントロールに対応する Name プロパティまたは Groupname プロパティの値

### 7.3.2 例

次の例は 3 つのデータ項目を、フォーム内のそれぞれ異なるコントロールにマッピングしています。

```
01 input-form is external-form.
   03  name-field pic x(30) identified by "Name".
   03  phone-no   pic x(30) identified by "Phone".
   03  email      pic x(15) identified by "EmailID".
```

## 7.4 サーバ側プログラムの出力

Net Express では、次に示す 2 通りの方法で、COBOL プログラムから Web ブラウザに HTML を出力できます。どちらの方法でも、HTML 出力の一部を COBOL 変数の値で置き換えることができます。

- 埋め込み HTML (EHTML)

  COBOL プログラム内の EXEC HTML～END-EXEC 間に HTML を記述します。HTML 文は原始プログラムに直接記述できるほか、コピーファイルとしてインクルードすることもできます。

EHTML を使用すれば、HTML 出力をプログラムで完全に制御できます。HTML ソースがサーバ側プログラムにリンクされるため、HTML コピーファイルに変更を加えた場合には、プログラムのコンパイルとリンクを再実行する必要があります。

- DISPLAY

  COBOL の DISPLAY 動詞を使用して、外部ファイルとして保存されている HTML ページを表示します。

### 7.4.1 EHTML の使用方法

埋め込み HTML (EHTML) を使用すれば、COBOL プログラムから HTML を直接出力することができます。インターネットアプリケーションウィザードで生成されるサーバ側プログラムも、EHTML を使用してエンドユーザに結果を返します。

EHTML では HTML ページ全体をコピーファイルとして、特別なデータ宣言をいっさい使用することなくプログラムにインクルードできます。また、部分的な HTML ページをコピーファイルとして使用したり、HTML 文を個別出力することによって、HTML ページの生成をプログラムで完全に制御できるため、HTML ページの生成がきわめて柔軟になります。

HTML を COBOL プログラム内に埋め込むには、EXEC HTML 文を使用します。埋め込み部分は EHTML プリプロセッサ (htmlp) によって処理されます。

EXEC HTML と END-EXEC を使用すれば、HTML を COBOL ソースコード内に直接埋め込むことができます。キーワード EXEC と HTML は同じ行に記述する必要があります。この点を除けば、書式上の制約はありません。 ただし、COPY ...REPLACING 文で使用されるコピーファイルには、EXEC HTML は 使用すべきではありません。 同様に、REPLACE 文のスコープ内では、EXEC HTML 文を使用しないでください。 このルールに従わずに EXEC HTML 文を使用すると、プログラムが意図したとおりに動作しなくなり、構文エラーが発生します。

次に一般的な書式を示します。

```
EXEC HTML
  [htmloutput]
  [copy "file.htm".]
END-EXEC
```

各パラメータの意味は次のとおりです。

*htmloutput*　Web ブラウザに出力する HTML コード

*file.htm*　　HTML コードを含むファイル

EHTML プリプロセッサは、CGI プログラムを起動した Web サーバに HTML コードを直接出力するだけで、HTML コードの検証や解析は行いません。

固定フォーマットの COBOL ソースコード内に埋め込まれた HTML は、EHTML プリプロセッサによって、各行の 8 文字目を 1 文字目に変換して出力されます。このため、HTML の <PRE> タグも、COBOL ソースコードの書式を気にすることなく使用できます。ただし、インクルードされるコピーファイルの拡張子が **htm** の場合には、この書式変換は行われません。プリプロセッサによる書式変換を常に無効にするには、プリプロセッサ指令 NOAUTOFORMAT を指定します。

#### 7.4.1.1 代入マーカー

代入マーカーを使用すると、HTML 出力の一部を、COBOL プログラム内の変数のデータで置き換えることができます。代入マーカーは、次のようにコロン (:) を先頭に付けた COBOL データ名として記述します。

**:***data-name*

EHTML 内では、常に表示項目を使用する必要があります。バイナリデータ項目を使用すると、判読できないデータがフォームに表示されてしまいます。

コロンの直後に空白文字や区切り文字マーカーを入れると、コロンは代入マーカーの接頭辞とは見なされず、そのまま出力されます。コロンの直前にバックスラッシュ文字 (\) を付けて、コロンを定数として出力させることもできます。

次に代入マーカーの使用例を示します。

```
 working-storage section.
 01 acct-code    pic 9(8).
 ...
 procedure division.
 ...
*> 最初のコロンは直後に空白文字があるため、EXEC
*> HTML で代入マーカーとして処理されることはあり
*> ません。 2 つ目のコロンは直後にデータ名が位置
*> しており、代入マーカーとして処理されます。
*> 3 つ目のコロンは直前にバックスラッシュがある
*> ため、通常のコロンとして処理されます。
EXEC HTML
     Account Code: :acct-code <BR>
     \:acct-code
END-EXEC
```

グループ項目のデータ名と基本項目のデータ名をピリオド (.) で区切って、代入マーカーを修飾できます。次に一例を示します。

```
 working-storage section.
 01 customer.
  03 name         pic x(80).
```

```
  03 acct-code    pic 9(8).
 ...
 procedure division.
 ...
*> EXEC HTML ブロック内の customer.acct-code は、
*> COBOL ソース内の customer の acct-code に相当します。
 EXEC HTML
     Account Code: :customer.acct-code <BR>
 END-EXEC
```

代入マーカーは部分参照することも可能です。

```
 working-storage section.
 01 acct-code    pic 9(8).
 ...
 procedure division.
 ...
*> acct-code の最初の 4 文字を部分参照して
*> 出力します。
 EXEC HTML
     Account Code: :acct-code(1:4) <BR>
 END-EXEC
```

---

**注記**: :*data-name* は、特定の文字列の直後に記述されている場合、代入マーカーとして処理されません。該当する文字列は、オンラインヘルプに一覧されています。[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**リファレンス > 埋め込み HTML > 代入マーカー**」の順にダブルクリックしてください。

---

#### 7.4.1.2 EHTML プリプロセッサ指令

EHTML プリプロセッサを実行するには、次のコンパイラ指令を設定する必要があります。

```
preprocess(htmlpp) [preprocessor-directives] endp
```

インターネットアプリケーションウィザードで生成されたすべてのプログラムには、先頭に EHTML プリプロセッサを実行するための $SET 文が記述されています。

```
$SET preprocess(htmlpp) endp
```

EHTML プリプロセッサ指令は、**htmlpp.dir** という名前の ASCII ファイルに記述して、Net Express システムの $COBDIR に登録されているディレクトリ内に保存する方法でも設定できます。

EHTML プリプロセッサ指令の詳細は、オンラインリファレンスに記載されています。[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**リファレンス > 埋め込み HTML**」の順にダブルクリックしてください。

## 7.4.2 DISPLAY の使用方法

HTML ページは、DISPLAY 動詞を使用して Web ブラウザに送信できます。HTML ページ内には、プログラムから渡される変数データを格納するための代入マーカーが含まれる場合があります。代入マーカーは標準 HTML ではありません。COBOL システムは、代入マーカーを変数データに置換してから、Web ブラウザにページを送信します。代入マーカーは、次のどちらかの形式で記述されます。

%%*name*%%
%%!s *name*%%

*name* はデータ宣言の IDENTIFIED BY 句で COBOL データ項目にマッピングされている名前です。データ項目をそのまま出力する場合は %%*name*%% を使用し、データ項目から後続の空白を削除する場合は %%!s *name*%% を使用します。通常、平文テキストを Web ブラウザに出力すると自動的に後続の空白文字が削除されますが、DISPLAY 動詞を使用してフォーム要素に値を格納する場合には、この機能を使用して後続の空白文字を削除する必要があります。

次に COBOL のグループデータ項目を出力ページにマッピングする構文を示します。

```
level-number data-name-1 IS EXTERNAL-FORM
                              identified by "pagefile.htm".
```

各部の意味は次のとおりです。

*level-number*　グループ項目の COBOL データ項目レベル番号
*data-name-1*　COBOL データ名
*pagefile.htm*　出力する HTML ページのファイル名

この宣言は、HTML フォーム ***pagefile.htm*** にグループ項目 data-name-1 とその中の各フィールドのデータセットを使用するように指定しています。続いて、基本データ項目をフォーム名にマッピングする構文を示します。

```
sub-level-number data-name-2 picture-string
                                        IDENTIFIED BY name.
```

各部の意味は次のとおりです。

*sub-level-number* COBOL データ項目のレベル番号
*data-name-2*　COBOL データ名
*picture-string*　COBOL PICTURE 文字列。PIC X(*n*)、PIC 9(*n*)、または数字編集フィールド

です。*n* 文字より長い文字列は超過部分が切り捨てられ、*n* 文字より短い文字列は左側から不足文字数だけ空白文字でパディングされます。

*name* 出力する HTML ページ内の代入マーカーの名前

#### 7.4.2.1 例

次の例は 3 つのデータ項目を、フォーム内のそれぞれ異なるコントロールにマッピングしています。

```
01 output-form is external-form identified by "outpage1.htm".
    03  name-field pic x(30) identified by "Name".
    03  phone-no   pic x(30) identified by "Phone".
    03  email      pic x(15) identified by "EmailID".
```

次の例は、上記の名前に代入マーカーを設定した HTML ページを示しています。

```
<HTML><BODY>
<H1>This is the output-form </H1>
<P>Dear %%Name%%,</P>
<P>You entered the following information</P>
<UL>
<LI>'phone number - <B>%%Phone%%</B></LI>
<LI>Email id - <B>%%EmailID%%</B></LI>
</UL>
</BODY></HTML>
```

## 7.5 アプリケーション状態の保持

Web ベースのアプリケーションで対処すべき課題の 1 つが、アプリケーション状態の保持です。CGI プログラム自体は、前回の処理内容や呼び出し元のクライアントを記憶できません。サーバ側プログラムを使用するクライアントごとにアプリケーション状態を保持する手段として、次の 2 通りの方法が利用できます。

- 非表示フィールド

  エンドユーザの Web ブラウザに送信されるフォーム内に、ブラウザに表示されないフィールドを作成し、その中に状態情報を設定します。非表示フィールドは、フォームをサポートするあらゆるブラウザで使用できます。ただし、この方法ではセッション間で状態を保持することはできません。エンドユーザのマシンでフォームが閉じられると、ただちに状態情報は失われてしまいます。

- cookie

  エンドユーザがブラウザで他のページやサイトに移動しても、アプリケーションに戻ったときに移動前の状態が保持されています。また、セッション間でも状態を保持できま

す。ただし、エンドユーザが cookie をサポートするブラウザを使用していることが条件になります。

どちらの方法にも、次の 2 つの問題があります。

- ネットワーク経由で大量の状態情報を送信すると、アプリケーションの実行速度が低下することがある。

  特に、ダイヤルアップで接続しているインターネットクライアントの場合は、この傾向が顕著に現れます。

- セキュリティやその他の理由で、イントラネットやインターネットでの状態情報の送信が望ましくない場合がある。

Net Express には上記の 2 つの方法に加え、すべての状態情報をサーバに格納し、この情報にサーバ側プログラムから素早くアクセスするための機能があります。ランタイムシステムが提供している一連の call-by-name ルーチンを使用すれば、サーバ側プログラムは接続するクライアントごとに一意のクライアント ID を要求し、それぞれのクライアントの状態データを索引ファイルに保存できます。

次の図のサーバ側プログラムは、フォームの送信時にクライアント ID を含む cookie をクライアント側ブラウザに送信します。このクライアント側ブラウザは Web への次回の要求送信時に、前回受信した cookie をサーバ側に渡します。サーバ側プログラムは受信した cookie 内のクライアント ID を使用して、合致する状態データを格納したレコードを取り込みます (このサーバ側プログラムは、最初のサーバ側プログラムと必ずしも一致する必要はありません)。

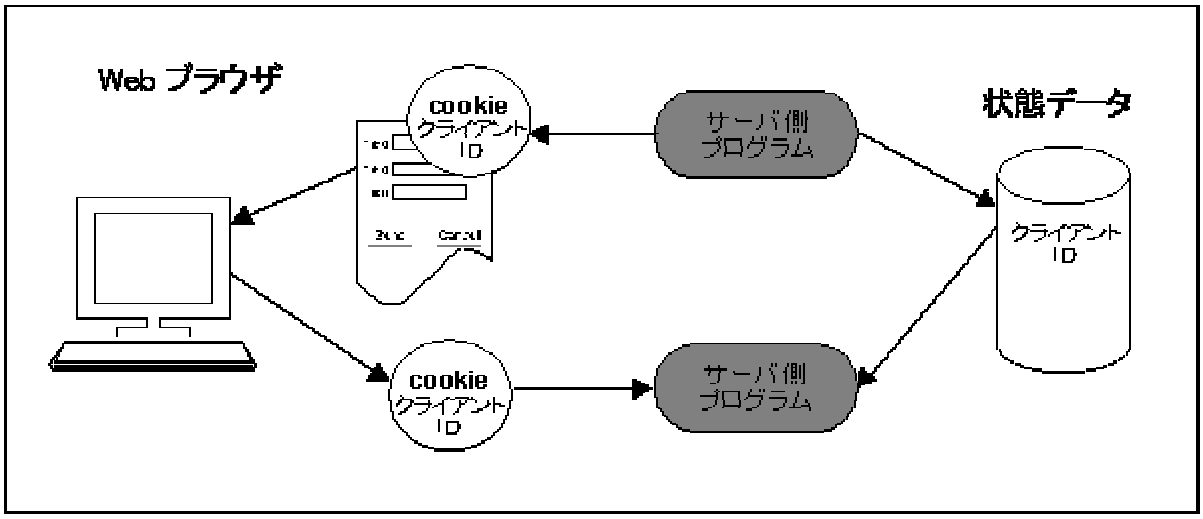



**図 32-2 サーバ側の状態ファイル**

**注記**：状態ファイルに重要な情報 (クレジットカード情報など) を格納する場合には、暗号化などのセキュリティ対策を追加実装するか、アプリケーションへのアクセス経路を、セキュリティが確保されたネットワークリンクだけに限定する必要があります。フォームを再送信する前に他のクライアントを偽装し、他人の状態レコードに不正にアクセスするユーザがいないとも限りません。そのようなユーザが他人の状態情報を閲覧できるかどうかは、アプリケーションの設計と Web ブラウザに返される情報の種類によって決まります。

---

以下の 3 つの項では、次のトピックについて説明します。

- 非表示フィールドの使用方法
- cookie の使用方法
- サーバ側の状態保持機能の使用方法

## 7.5.1 非表示フィールド

HTML フォームには非表示フィールドを配置できます。HTML フォームに非表示フィールドを追加してみてください。追加したフィールドは HTML 入力 (エントリフィールド) と同様に機能しますが、エンドユーザ側でフォームをロードしたブラウザには表示されません。

このフォームを入力フォームや出力フォームとして使用するサーバ側プログラムを生成すると、エントリフィールドの場合と同様に、非表示フィールドに対応する COBOL データ項目が生成されます。フォームの出力時にアプリケーションの状態情報を非表示フィールドに格納すれば、フォームが他のサーバ側プログラムに送信されたときに、そのサーバ側プログラムに状態情報を渡すことができます。

## 7.5.2 cookie

cookie とは名前と値を組み合わせた情報であり、HTML ページとともに Web ブラウザに送信されます。cookie にアプリケーションの状態情報を記録すれば、それを受信したクライアントが次回にサーバ側プログラムにアクセスするときに、サーバ側プログラム側で状態情報を取り込むことができます。Netscape Navigator の 2.0 以降、および Microsoft Internet Explorer の 3.0 以降の各バージョンは、いずれも cookie をサポートしていますが、他のブラウザではサポートされていない場合があります。

cookie の詳しい仕様については、Netscape Web サイトを参照してください。

デフォルトでは、エンドユーザが Web ブラウザを閉じるまでの間、cookie はブラウザで保持されます。cookie には有効期限を設定することもできます。その場合、cookie は有効期限が失効するまでの間、ブラウザのキャッシュ内に保持されます。

Form Designer で HTML ページを作成しているときに、[**ページ**] メニューの [**cookie**] オプションを使用すれば、そのページに cookie を関連付けることができます。cookie には属性として、名前、値、COBOL PICTURE、有効期限、パス、ドメイン、およびセキュリティを設定できます。ドメイン属性とパス属性は、複数のページ間での cookie の共有に使用します。「cookie」ダイ

アログボックスで [**OK**] をクリックすると、サーバ側とクライアント側の両方で cookie の設定と読み取りに必要なコードが、ほぼ完全な形で Form Designer によって生成されます。

- クライアント側では、ページのロード時やアンロード時に呼び出される JavaScript コードが生成されます。このコードはページのロード時に cookie の値を取得し、その値を JavaScript 変数に格納して、アンロード時に cookie に保存します。
- サーバ側では、「cookie」ダイアログボックスで指定した COBOL PICTURE データ項目が cookie ごとに生成されます。また、cookie を読み取るための ACCEPT 文も生成されます。HTML ページがサーバで受け付けられると cookie 内の情報がデータ項目に格納され、出力ページが Web ブラウザに送信されるときに、データ項目の値が cookie に格納されます。

Form Designer の Cookie Editor の詳細については、オンラインヘルプを参照してください。[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**キーワード**」タブに **cookie** と入力して、「**該当するトピック**」ダイアログボックスで「**cookie を追加する**」を選択します。

ページの表示中に cookie に新しい値を格納するには、値を cookie に割り当てる JavaScript コードをページに追加する必要があります。JavaScript コードの作成方法の詳細については、『クライアント側のプログラミング』の章を参照してください。

#### 7.5.2.1 cookie の使用例

この例では HTML ページを作成して cookie と関連付け、ページへのアクセス回数を cookie に記録します。カウンタの値はサーバ側プログラムで増やします。ここでは [OK] ボタンをどこでクリックするか、といった細かいレベルの説明ではなく、全体的な手順の流れを示すことに重点を置いています。インターネットアプリケーションウィザードや Net Express の詳しい操作手順については、オンラインヘルプを参照してください。作業に着手する前に、インターネットアプリケーションウィザードの操作に慣れておきたい方は、『入門書 - その他のトピック』のサンプルセッションを利用できます。

cookie を使用するアプリケーション例を作成し、cookie を設定する手順は次のとおりです。

1. 新しいプロジェクトを PAGECNTR.app という名前で作成します。
2. 空白のテンプレートを選択して、HTML ページを新規作成します。
3. 見出しを記述した後、ユーザにページのアクセス回数を知らせるフォームをペイントします。このフォームには送信ボタンも含めます。次の図で "page" と "times" の間にあるフィールドは、テキストエントリフィールドです。このフィールドに "hitcounter" という名前を付けます。図 32-3 に、作成したフォームの例を示します。

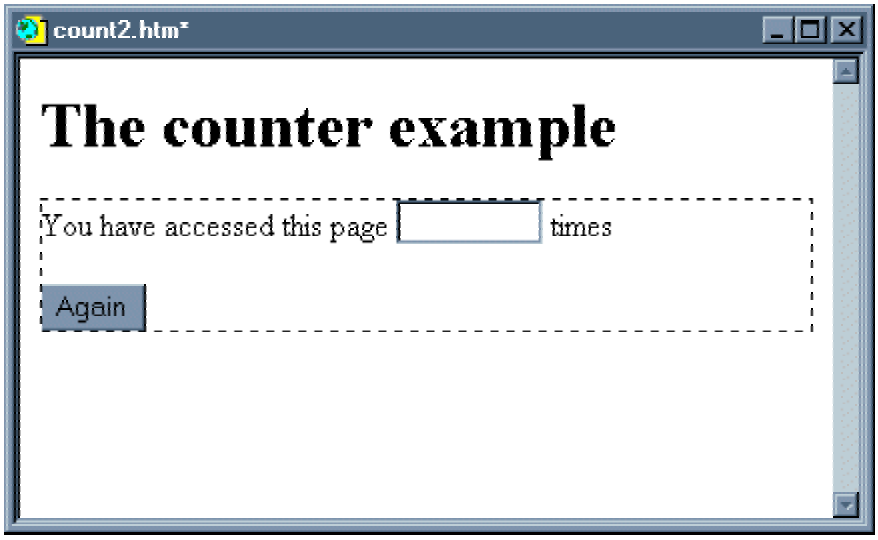



**図 32-3 cookie の使用例**

4. [**ページ**] メニューの [**Cookies**] オプションを使用して、ページに cookie を追加します。cookie の「**Name**」は cookiecounter、「**Value**」は 0、「**COBOLPicture**」は 9999 にそれぞれ設定します。
5. [**ファイル**] メニューの [**新規作成**] をクリックし、「インターネットアプリケーション」を選択して、インターネットアプリケーションウィザードを起動します。
6. 最初のページで「サーバプログラム」を選択します。
7. 「サーバプログラムの生成」ページの「入力ファイル/フォーム」リストと「出力フォーム」リストの両方で HTML ページを選択します。
8. [**次へ**] をクリックし、最後のページで [**完了**] をクリックします。インターネットアプリケーションウィザードによってサーバ側プログラムと、フォームデータ用の関連コピーファイルが生成されます。
9. 生成された COBOL プログラムを開き、process-business-logic 節に移動します。
10. 既存のコードの前に、次のコードを入力します。

```
add 1 to cookiecounter
move cookiecounter to hitcounter
```

11. COBOL ファイルを保存し、プロジェクトをリビルドします。
12. アプリケーションを実行します。Solo を使用している場合は、[**アニメート > 実行**] をクリックします。フォームで [**Again**] をクリックするたびに、hitcounter フィールドの値が 1 ずつ増加します (アプリケーションの実行方法の詳細については、『*CGI ベース のアプリケーションの新規作成*』の章の『アプリケーションの実行』を参照してください。

## 7.5.3 サーバ側の状態保持機能

サーバ側の状態保持機能を使用すると、アプリケーションの状態に関する情報を、指定した形式のレコードとして保存することができます。この機能を使用する場合は、セッションの開始時にクライアント ID を割り当てるため、アプリケーションの最初のフォームを常にサーバ側プログラムから出力する必要があります。

クライアント ID を cookie に保存する場合、状態情報がユーザの現在のセッションの終了後も保存されるように有効期限情報を設定できます。そのように設定すると、アプリケーションを永続化できます。この場合の永続化とは、アプリケーションのセッションをいったん終了した後、翌日にセッションを再開したときに、前回作業していた状態から作業を開始できることを意味します。永続化するアプリケーションのプログラムは、クライアント ID を含む cookie をチェックし、当該クライアントによる実行が初めてかどうかを判定する必要があります。その結果、クライアント ID の値が空白文字であれば ID を割り当て、新しいセッションを開始します。それ以外の値の場合は、クライアント ID に合致する状態情報を復元し、適切なフォームを表示する必要があります。

---

**注記**: サーバ側の状態保持機能は、sstate と呼ばれるモジュールで実装されています。sstate はソースコードでも提供されているため、サブルーチンに変更を加えて、暗号化などの機能を追加することができます。

sstate のソースコード (**sstate.cbl**) は、**Net Express\base\demo\sstate** フォルダにあります。変更を加える場合には、新しい Net Express プロジェクトを作成し、その中に **sstate.cbl** をコピーしてください。変更した sstate モジュールをアプリケーションで使用するには、**sstate.obj** と **sstate.gnt** をビルドします。 **sstate.obj** は **Net Express\base\lib**、**sstate.gnt** は **Net Express\base\bin** にそれぞれコピーします。

データアクセスウィザードで生成されたアプリケーションは、すべて sstate に依存しています。sstate とのインターフェイスには変更を加えないでください。インターフェイスを変更すると、データアクセスウィザードで作成したアプリケーションが意図したようには動作しなくなる可能性があります。すべての sstate 呼び出しへのインターフェイスは、オンラインヘルプで説明されています。[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**リファレンス > ライブラリルーチン > 関数ごとのライブラリルーチン > 状態の維持**」の順にダブルクリックしてください。

---

### 7.5.3.1 クライアント状態レコードの格納と取り込み

次のフローチャートは、サーバ側の状態保持機能を使用するためのロジックを示しています。

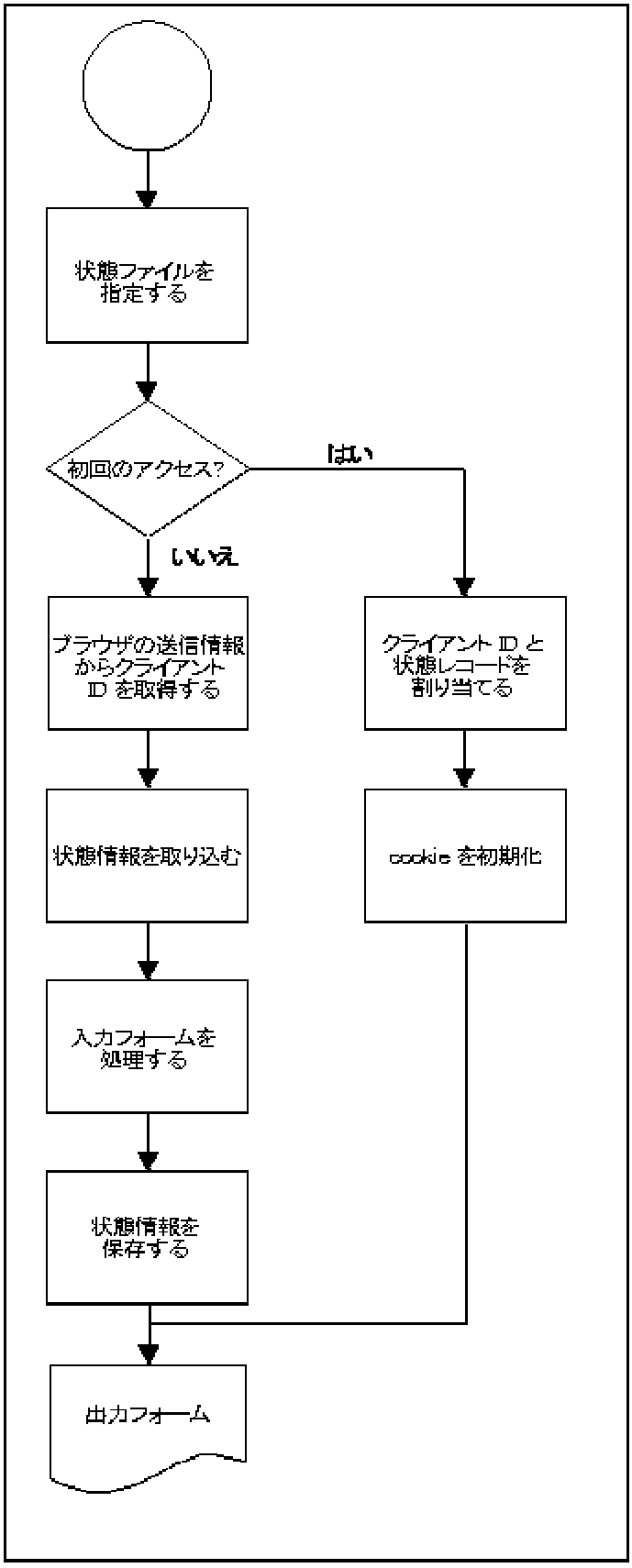

状態ファイルを
指定する
初回のアクセス?
はい
いいえ
ブラウザの送信情報
からクライアント
ID を取得する
クライアント ID と
状態レコードを
割り当てる
状態情報を取り込む
cookie を初期化
入力フォームを
処理する
状態情報を
保存する
出力フォーム

**図 32-4 サーバ側の状態保持機能の使用手順**

このフローチャートの大まかな流れを次に説明します。

1. 状態ファイルの名前を指定します。

   MF_CLIENT_STATE_FILE ルーチンを呼び出します。

2. 初めてアクセスしたクライアントの場合、MF_CLIENT_STATE_ALLOCATE ルーチンを呼び出して新しいクライアント ID を割り当て、さらに cookie または非表示コントロールを初期化してステップ 6 に進みます。

   アクセス済みのクライアントの場合は、ブラウザから返された cookie または非表示コントロールからクライアント ID を読み取り、ステップ 3 に進みます。

3. クライアント ID に合致するレコードを読み取ります。

   MF_CLIENT_STATE_RESTORE ルーチンを呼び出します。

4. 入力フォームを読み取り、必要に応じて状態情報を使用して処理します。
5. クライアントの状態を保存します。

   MF_CLIENT_STATE_SAVE ルーチンを呼び出します。

6. 出力フォームをブラウザに送信します。

ここで言及した各ルーチンについては、オンラインリファレンスを参照してください。[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**リファレンス > ライブラリルーチン > 関数ごとのライブラリルーチン > 状態の維持**」の順にダブルクリックしてください。

次に、状態保持ルーチンを使用しているスケルトンプログラムの一例を示します。太字のコメントは、上図のフローチャート内のボックスに直接関連する部分を示しています。

```
working-storage section.
...
01 state-filename    pic x(255) value
                     "MF-STATE-SAVE.DAT".
01 client-length     pic xxxx comp-x.
01 client-state.
  03 ...
  *> クライアント状態レコードの形式は
  *> 任意に指定できます。レコード長は
  *> client-length フィールドに格納します。
01 state-status       pic x comp-x.
01 browserinput is external-form.
  03 name-field       pic x(30) identified by "name".
```

```
  03 phone-no        pic x(30) identified by "phone".
  03 email           pic x(15) identified by "emailid".
*> cookie
  03 client-id       pic x(30) identified by "clientid".
 ...

procedure division.
   *> 状態ファイルを指定します。
   call "MF_CLIENT_STATE_FILE" using state-filename
                                     state-status
   ...
   *> ブラウザの送信情報からクライアント ID
   *> を取得します。
   accept browserinput
   move length of client-state to client-length
   if client-id = spaces
   *> cookie が空 ･･･ 初回のアクセス
   *> 状態レコードを割り当て、新しいクライアント
   *> ID を取得します。
   call "MF_CLIENT_STATE_ALLOCATE" using client-id
                                         client-length
                                         state-status
   *> cookie を初期化します。cookie を使用して
   *> クライアント ID をユーザマシンに格納する場合
   *> は、cookie の出力に使用するデータ項目を
   *> クライアント ID で初期化します。cookie は
   *> プログラムのフォーム出力とともに送信されます。
   ...
   else
   *>  状態情報を取り込みます。
   call "MF_CLIENT_STATE_RESTORE" using client-id
                                        client-state
                                        client-length
                                        state-status
   end-if
   ...
   *> 入力フォームを処理します。この時点
   *> までに、入力フォーム内のすべての情報
   *> と、状態保持ファイル内の状態情報の
   *> 取得が完了しています。
   ...
   *>  状態情報を保存します。
   call "MF_CLIENT_STATE_RESTORE" using client-id
                                        client-state
                                        client-length
                                        state-status
```

```
*> 出力フォームを送信します。  クライアント ID をフォームの
*> 非表示コントロールに格納する場合は、この情報で当該コントロール
*> を初期化する必要があります。 クライアント ID が cookie に
*> 格納されている場合には、ユーザへの再送信は不要です。
...
```

#### 7.5.3.2 クライアント状態レコードの削除

不要になったクライアント状態レコードは、標準で提供されている 2 つのルーチンで削除できます。1 つは特定のレコード 1 件を削除するルーチン、もう 1 つは特定の日数が経過した全レコードを削除するルーチンです。

1 つのレコードを削除するには、"MF_CLIENT_STATE_DELETE" を呼び出します。次に使用例を示します。

```
 working-storage section.
 ...
 01 state-status             pic x comp-x.
 01 client-id                pic x(30).
 01 state-filename           pic x(255) value
                             "MF-STATE-SAVE.DAT".

 procedure division.
      ...
*>  クライアント状態ファイルを開きます。
     call "MF_CLIENT_STATE_FILE" using state-filename
      ...
     call "MF_CLIENT_STATE_DELETE " using client-id
                                          server-status
```

特定の日数が経過したレコードをすべて削除するには、"MF_CLIENT_STATE_PURGE" を呼び出します。次に使用例を示します。

```
 working-storage section.
 ...
 01 state-status             pic x comp-x.
 01 age-in-days              pic x(4) comp-x.
 01 state-filename           pic x(255) value
                             "MF-STATE-SAVE.DAT".

 procedure division.
      ...
*>  クライアント状態ファイルを開きます。
     call "MF_CLIENT_STATE_FILE" using state-filename
      ...
*> 生成から 6 日以上経過した全レコードを削除します。
```

```
move 5 to age-in-days
call "MF_CLIENT_STATE_PURGE " using age-in-days
                                    server-status
```

これらのルーチンについては、オンラインリファレンスで詳しく説明しています。[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**リファレンス > ライブラリルーチン > 関数ごとのライブラリルーチン > 状態の維持**」の順にダブルクリックしてください。

---



---

# 第 33 章 : CGI、ISAPI、および NSAPI プログラム

この章では、サーバ側プログラムに使用できる 3 種類の API の相違点について説明し、これらの API をコンパイラ指令を変更するだけで切り替える方法を示します。

## 8.1 概要

サーバ側プログラムの実行には、現時点で次の 3 種類の API を使用できます。

- CGI (Common Gateway Interface)

  従来から広く使用されており、すべての Web サーバでサポートされている API。このタイプの API はデバッグも最も簡単であるため、サーバ側プログラムは CGI で実装することが推奨されます。

- ISAPI (Internet Server API)

  Microsoft で開発された API。CGI プログラムに比べ、より高速に実行されます。ISAPI は、Microsoft Internet Server や、その他一部のベンダの Web サーバでサポートされています。

- NSAPI (Netscape Server API)

  Netscape 社で開発された API。CGI プログラムに比べ、より高速に実行されます。NSAPI は、Netscape の Web サーバや、その他一部のベンダの Web サーバでサポートされています。

ISAPI や NSAPI を使用して実装したサーバ側プログラムは、次の 2 つの理由によって、標準的な CGI アプリケーションより高速に起動します。

- Web サーバによって別スレッドとして起動される。

  標準的な CGI プログラムは別プロセスとして起動されます。スレッドの起動は、新しいプロセスを起動する場合に比べ、より高速です。

- サーバ側プログラムがダイナミックリンクライブラリ (.dll ファイル) としてコンパイルされる。

  いったん実行されるとサーバによってメモリ内に保持されるため、それ以降の実行はさらに高速になります。

ISAPI プログラムと NSAPI プログラムはプロセスとしては実行されず、サーバとは別のスレッドとして実行されます。したがって、これらのプログラムは、マルチスレッドに対応する必要があります。REENTRANT(2) コンパイラ指令を設定すれば、あらゆる COBOL プログラムをマルチスレッド化できます。 コンパイラ指令の設定方法については、[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**リファレンス > コンパイラ指令**」を順に選択すれば、詳細情報を参照できます。.

# 8.2 ISAPI または NSAPI を使用するサーバ側プログラムの作成

ISAPI や NSAPI を使用するサーバ側プログラムは、CGI プログラムの場合と同様、入出力に ACCEPT/DISPLAY と EHTML を使用する方法で作成できます。サーバ側プログラムは CGI プログラムとして開発し、デバッグして正常に機能することを確認した段階で、ISAPI プログラムや NSAPI プログラムに変更することが推奨されます。ISAPI を実行する Web サーバは、バグに比較的厳密であるため、ISAPI プログラムがクラッシュすると、Web サーバソフトウェアがハングアップし、再起動が必要になる場合があります。 ISAPI アプリケーションのビルドおよびテストの方法については、[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**プログラミング > CGI ベースのアプリケーション > CGI ベースのアプリケーションのデバッグ**」を順にダブルクリックすれば、詳細情報を参照できます。

サーバ側プログラムの開発時に推奨される手順を次に示します。

1. サーバ側プログラムを CGI プログラムとして作成します。データファイルなどのリソースに対する競合は必ず許可してください。リソースの競合は、すべてのサーバ側プログラムに影響を与えます。

   ISAPI プログラムは、Web サーバのサービスプロセスの一部として実行されます。つまり、サービス内で実行されるため、Windows NT のシェルへのアクセス (ウィンドウやダイアログボックスの作成など) を試みるべきではありません。

   ---

   **注記**:ISAPI や NSAPI を使用するプログラムは Web サーバのプロセス内で実行されるため、処理の終了時に EXIT PROGRAM または GOBACK を使用して、制御を戻す必要があります。 CGI プログラムのように STOP RUN 文を使用すると、サーバがハングする原因になります。

   ---

2. プログラムを CGI プログラムとしてビルドし、デバッグします。
3. プログラムで保護違反の原因になるようなエラーが発生しないことを確認した後、後述する手順に従って ISAPI プログラムまたは NSAPI プログラムとしてリビルドします。
4. 使用する API をサポートする Web サーバでプログラムをテストします。
5. プログラムをディプロイします。

## 8.2.1 ISAPI プログラムからの環境変数へのアクセス

ISAPI プログラムから Windows の環境変数にアクセスするには、プログラム内にライブラリルーチン PC_ISAPI_GET_EXT_BLOCK の呼び出しを記述します。この呼び出しは ISAPI 拡張ブロックを返します。このブロック内には、ISAPI の getservervariable ルーチンを指すポインタを格納したフィールドがあり、このポインタを使って getservervariable ルーチンを呼び出すことができます。なお、この手法を使用するには、**Net Express\base\source\isapiext.cpy** にあるコピーファイルをプログラムにインクルードする必要があります。

ISAPI プログラムと CGI プログラムでは、環境変数の命名規則が異なる点に留意してください。ISAPI 環境変数の詳細については、Net Express Links から Microsoft の Web に移動し、掲載されている ISAPI 拡張のドキュメントを参照してください。

PC_ISAPI_GET_EXT_BLOCK を呼び出すプログラムの一例を次に示します。

```
$set sourceformat(free) webserver(isapi) reentrant(1) case

 copy "isapiext.cpy"

 special-names.
     call-convention 74 is WINAPI.
 working-storage section.
 thread-local-storage section.
 01 ext-block usage extension-control-block.
 01 option-name pic x(255).
 01 option-value pic x(255).
 01 temp-1 pic 9(9) comp-5.

 procedure division.

 move length of extension-control-block to cbsize of ext-block
 call "PC_ISAPI_GET_EXT_BLOCK" using ext-block
 if retrun-code not = 0
     display "Not ISAPI environment"
     exit program
     stop run
 end-if

 move z"REQUEST_METHOD" to option-name *> NULL で終端する文字列
 move length of option-name to temp-1
 move space to option-value(1:1)
 call WINAPI GetServerVariable using
                              by value     connid
                              by reference option-name
                                           option-value
```

```
                                                temp-1
*> temp-1 には、option-value 内の文字列（末尾の NULL バイトを含む）の
*> 長さが格納されています。
 subtract 1 from temp-1

 exit program
 .
```

PC_ISAPI_GET_EXT_BLOCK ルーチンの詳細については、『*PC_ISAPI_GET_EXT_BLOCK*』を参照してください。

---

**注記**：PC_ISAPI_GET_EXT_BLOCK ルーチンは、getservervariable 手続きポインタの取得だけに使用してください。 その他の用途に使用すると、予期せぬ結果につながるおそれがあります。

---

## 8.2.1 ISAPI 用コンパイラ指令の設定

ISAPI プログラムをリビルドする前に、次の各コンパイラ指令を設定します。

- WEBSERVER(ISAPI)

  このプログラムが ISAPI アプリケーションであることをコンパイラに通知します。

- CASE

  外部シンボル (呼び出されたプログラムの名前を含む) の大文字変換を禁止します。この指定を省略すると、エンドユーザが ISAPI プログラムを実行しようとしたときに、見つからない旨を知らせるメッセージが Web ブラウザに表示されます。

- REENTRANT(2)

  このプログラムの複数のコピーを確実かつ安全に実行できるようにします。

アプリケーションで Open ESQL を使用する場合には、次の指令も設定する必要があります。

- SQL(THREAD=ISOLATE)

  SQL リソースとトランザクションのスレッド間共有を禁止します。

インターネットアプリケーションウィザードで生成されるデータアクセスアプリケーションは、必ず Open ESQL を使用します。

プログラムの冒頭に $SET 文を記述すれば、そのプログラムをコンパイルするたびに、設定したコンパイラ指令が自動的に使用されます。次に記述例を示します。

```
$set webserver(isapi) case reentrant(2)
```

ドル記号 ($) が必ずソースコードの 7 文字目に位置するように記述してください。ただし、SOURCEFORMAT"FREE" 指令を設定する場合には、この制約は適用されません。 コンパイラ指令の設定方法については、[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**リファレンス > コンパイラ指令**」を順に選択すれば、詳細情報を参照できます。

## 8.2.2 NSAPI 用コンパイラ指令の設定

NSAPI プログラムをリビルドする前に、次の各コンパイラ指令を設定します。

- WEBSERVER(NSAPI,*entry-point-name*)

  このプログラムが NSAPI アプリケーションであることをコンパイラに通知します。*entry_point_name* には、ユーザが定義したエントリポイント名を指定します。*entry_point_name* で指定される名前の非表示エントリポイントがコンパイラによって生成され、NSAPI Web サーバによるプログラム起動が可能になります。

  このアプリケーションを NSAPI サーバにディプロイする際には、サーバの **obj.conf** ファイルを編集し、`Init fn` の `funcs` 属性の値を *entry_point_name* に設定してください。『CGI ベースのアプリケーションのディプロイ』の章の『NSAPI サーバ設定ファイルへの変更』を参照してください。

- CASE

  外部シンボル (呼び出されたプログラムの名前を含む) の大文字変換を禁止します。この指定を省略すると、エンドユーザが NSAPI プログラムを実行しようとしたときに、見つからない旨を知らせるメッセージが Web ブラウザに表示されます

- REENTRANT(2)

  このプログラムの複数のコピーを確実かつ安全に実行できるようにします。

アプリケーションで Open ESQL を使用する場合には、次の指令も設定する必要があります。

- SQL(THREAD=ISOLATE)

  SQL リソースとトランザクションのスレッド間共有を禁止します。

インターネットアプリケーションウィザードで生成されるデータアクセスアプリケーションは、必ず Open ESQL を使用します。

プログラムの冒頭に $SET 文を記述すれば、そのプログラムをコンパイルするたびに、設定したコンパイラ指令が自動的に使用されます。次に記述例を示します。

```
$set webserver(nsapi,run_update_1) case reentrant(2)
```

ドル記号 ($) が必ずソースコードの 7 文字目に位置するように記述してください。ただし、SOURCEFORMAT"FREE" 指令を設定する場合には、この制約は適用されません。 コンパイラ指令の設定方法については、[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**リファレンス > コンパイラ指令**」を順に選択すれば、詳細情報を参照できます。

## 8.2.3 ISAPI/NSAPI プログラムのリンク

CGI プログラムは **.exe** ファイルとしてビルドし、ISAPI プログラムや NSAPI プログラムは **.dll** ファイル (ダイナミックリンクライブラリ) としてビルドします。詳細については、『CGI ベースのアプリケーションのディプロイ』の章の『共有ランタイムシステムを使用する ISAPI/NSAPI プログラムのビルド』を参照してください。

CGI として開発したサーバ側プログラムを ISAPI または NSAPI の **.dll** ファイルとしてリビルドした場合には、そのプログラムを参照している Web ページやフォーム内の URL もすべて変更する必要があります。CGI プログラムは次のような URL で参照されています。

*/share-name/program*.exe

ISAPI/NSAPI へのリビルド後、URL を次のように変更します。:

*/share-name/program*.dll

---



---

# 第 34 章 : Form Designer の出力の編集

この章では、HTML ページの作成時に Form Designer が出力するファイルの内容と、Form Designer を他の HTML 編集ツールと併用する方法について説明します。HTML については説明していません。

## 9.1 概要

Form Designer では HTML ページ内のあらゆる部分を表示、編集できますが、ページの作成に他のツールを使用して、フォームの追加やクライアント側スクリプトの作成だけに Form Designer を使用することも可能です。

Form Designer で開くファイルは、実際には次の 2 種類のファイルから構成されます。

- **.htm** ファイル

  ページ内のすべてのテキストとレイアウト情報を含む HTML ファイル。フォームのレイアウトに関するすべての情報が、標準的な HTML の形式 (テーブルまたは DHTML) でページ内に記述されています (『フォームと HTML』の章を参照してください)。

- **.mff** ファイル

  Form Designer 独自形式のファイル。インターネットアプリケーションウィザードを使用して、ページ内のフォームから生成した COBOL データ項目を、サーバ側プログラム内で宣言するために必要な情報が含まれています。具体的には、ページ内の各 HTML コントロールに対応する COBOLPicture プロパティなどが格納されています。

Form Designer では、他の HTML エディタで作成した HTML ファイルを開くことができます。また、その逆の操作も可能です。

### 9.1.1 他の HTML エディタで作成したページを Form Designer で開くには

他の HTML エディタで作成した有効な HTML ページは、Net Express のプロジェクトディレクトリにコピーし、プロジェクトに追加してダブルクリックすれば、Form Designer で開くことができます。開いたページに対応する **.mff** ファイルが存在しなければ、Form Designer によって自動的に .mff ファイルが生成されます。Form Designer で開いたページには、必要に応じて変更を加えたり、新しいフォームを追加することができます。開いたページに必要なフォームがすでに含まれている場合は、フォーム内の各コントロールに対応する COBOL プロパティを設定するだけで、そのページをインターネットアプリケーションウィザードで使用できるようになります。

---

**注記**: コントロールのデータ名には、HTML コントロールの Name 属性の値が使用されます。Form Designer でページを開くときには、Name 属性の値と COBOL の予約語との重複はチェックされません。COBOL の予約語と同じ名前を持つコントロールがあると、このフォームからインターネットアプリケーションウィザードで生成した COBOL アプリケーションは構文エラーになります。その場合には、コントロールの名前を変更してください。

---

## 9.1.2 Form Designer で作成したページを他の HTML エディタで開くには

Form Designer で作成した **.htm** ファイルは、標準の HTML で記述されているため、情報を失わずに他の HTML エディタで開くことができます。ただし、情報を独自形式のファイルに格納するタイプのエディタでは、フォームをインポートして、独自形式のファイルに **.htm** ファイル内の情報を反映させる必要があります。

---



---

# 第 35 章 : クライアント側のプログラミング

この章では、サーバー側プログラムを実行せずに、フォームのイベント処理やその他の処理を行う方法について説明します。

この章で説明する機能の使用方法については、オンラインヘルプで手順を追って詳しく説明しています。[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**プログラミング > CGI ベースのアプリケーション > Form Designer > JavaScript とイベント処理 > 概要 - スクリプトアシスタント**」を順にクリックしてください。

## 10.1 概要

『***分散コンピューティング***』のこのパートでは、主にフォームから送信されたデータを処理して結果を返すサーバ側プログラムの作成方法に重点を置いていますが、対話型のコードはフォーム自体に記述することも可能です。そのようなフォームを作成するには、JavaScript や JScript を使用します。前者は Netscape の Web ブラウザ、後者は Microsoft の Web ブラウザでサポートされており、JScript は JavaScript に基づく派生言語です。フォームに対話型コードを追加するときには、JavaScript や JScript で関数を作成し、それをフォーム内のイベントに関連付けます。イベントに関連付けられた関数はイベントハンドラと呼ばれます。たとえば、エンドユーザが特定のチェックボックスをクリックしたときに画像を表示するイベントハンドラなどを作成できます。

フォーム内の対話型コードの顕著な活用例の 1 つが、フォームに入力されたデータの検証です。ただし、Net Express には検証用の JavaScript 関数が付属しており、必要に応じて呼び出すことができるため、通常的な検証で JavaScript や JScript のコードを作成する必要はほとんどありません。フォームの検証方法については、『フォームの検証』の章で説明しています。

Form Designer のスクリプトアシスタントでは、マウスポインタのポイント & クリック操作だけでイベントハンドラを作成できます。JavaScript や JScript の知識を持たなくても、この方法で効果的なイベントハンドラを作成し、フォームに追加することができます。たとえば、デフォルトのデータをフォームに設定するコードや、自動的に他の Web ページに移動するコードなどを作成できます。

スクリプトアシスタントで生成されるコードは、JavaScript と JScript の両方の構文規則に従っていますが、Microsoft Internet Explorer で機能するコードが Netscape Navigator では機能しなかったり、その逆のケースが発生することがあります。これは、2 種類のブラウザで採用されているドキュメントオブジェクトモデルが互いに異なるためです。

イベントハンドラのコードは、条件文やループなどを加えるうちに、非常に複雑になることがあります。そのように複雑で高度なコードを作成するには、この章で説明する JavaScript や JScript の知識だけでは十分ではありません。Internet Explorer のユーザを主なターゲットとしてコードを作成する場合は、Net Express Links のリンク先にある Microsoft JScript ドキュメントを参照してください。主に Netscape Web ブラウザのユーザ向けにコードを作成する場合

も、同様に Net Express Links のリンク先にある Netscape 社の JavaScript ドキュメントを参照してください。また、後述する『ブラウザ間の互換性』にも留意してください。

この章で JavaScript に関して言及している内容は、JScript にも同様に適用できます。

## 10.2 JavaScript の概要

JavaScript はオブジェクトの概念に基づき、メソッド、プロパティ、およびスタイルを内包したオブジェクトを扱う言語の一種です。オブジェクトにはフォームのコントロールだけでなく、テキストのあらゆる要素やドキュメント自体、およびウィンドウも含まれます。メソッドとは、「フォーカスを失う」、「選択される」など、オブジェクト自体で実行できる動作のことです。プロパティとスタイルは、オブジェクトの属性です。Dynamic HTML (DHTNL) の登場によって、フィールド内容の設定、テキストや背景の色変更、新しいウィンドウの呼び出し、HTML の追加要素の作成など、表示内容や振る舞いをいっそう広範囲にわたって制御できるようになりました。

イベントハンドラは、HTML コントロールと ActiveX コントロールを対象として作成できます。HTML コントロールのイベントハンドラを作成すると、HTML 文書内には次の 2 箇所に JavaScript が挿入されます。

- JavaScript の関数は、HTML 文書の <HEAD> セクション内の <SCRIPT> タグと </SCRIPT> タグの間に挿入される。
- 関数の呼び出しは、フォームに含まれるコントロールの属性値として、HTML 文書の <BODY> セクション内に挿入される。

ActiveX コントロールのイベントハンドラを作成した場合には、JavaScript は HTML 文書内の 1 箇所のみ、<HEAD> セクション内の <SCRIPT>～</SCRIPT> 間に挿入されます。

Java アプレットのイベントハンドラを作成し、アプレットに含まれるメソッドを呼び出したり、スタイルやプロパティの取得と設定を行うこともできます。Java アプレットを制御する JavaScript のコードも、HTML 文書の <HEAD> セクション内の <SCRIPT>～</SCRIPT> 間だけに挿入されます。

JavaScript のコードは、HTML 文書内に記述するかわりに、拡張子 .js の別ファイルとして保存することもできます。別ファイルとして保存すれば、複数のフォームから関数を呼び出すことが可能になります。1 つのフォームで、同じファイル内に記述された関数と、.js ファイル内の関数の両方を呼び出すこともできます。

スクリプトアシスタントを使用すると、HTML 文書の右側に JavaScript コードが自動生成されます。

## 10.3 スクリプトアシスタント

Form Designer のスクリプトアシスタントを使用すると、次の処理を行うベントハンドラを容易に作成できます。

- メソッドの呼び出し
- プロパティの取得と設定
- スタイルの取得と設定

スクリプトアシスタントには、次の 5 つのタブがあります。

- 「イベント」タブ
- 「メソッド」タブ
- 「プロパティ」タブ
- 「スタイル」タブ
- 「スクリプト」タブ

「イベント」、「メソッド」、「プロパティ」、「スタイル」の 4 つのタブでは、マウスポインタのポイント & クリック操作だけでイベントハンドラを作成できます。「スクリプト」タブでは、コードを直接入力して編集できます。

タブの内容とスクリプトアシスタントの動作は、作成するスクリプトの対象が (HTML コントロールまたは ActiveX コントロール) によって異なります。

### 10.3.1 「イベント」、「メソッド」、「プロパティ」、「スタイル」の各タブ

「イベント」、「メソッド」、「プロパティ」、「スタイル」の各タブは表示内容が似ており、いずれも 4 つのペインから構成されます。次の 3 つのペインは、これらの各タブに共通です。

- オブジェクトビューを含む左上のペイン
- 右側中段のヘルプペイン
- イベントハンドラのコードを表示する下部のペイン

右上にある 4 つ目のペインには、選択しているタブに応じて、イベント、メソッド、プロパティ、またはスタイルのビューが表示されます。ヘルプペインには、選択したイベント、メソッド、プロパティ、またはスタイルに関する情報が表示されます。

各タブには、タブを操作するためのコントロールが 1 つ以上あります。

図 35-1 は、HTML コントロールを含むフォームの「イベント」タブを示しています。

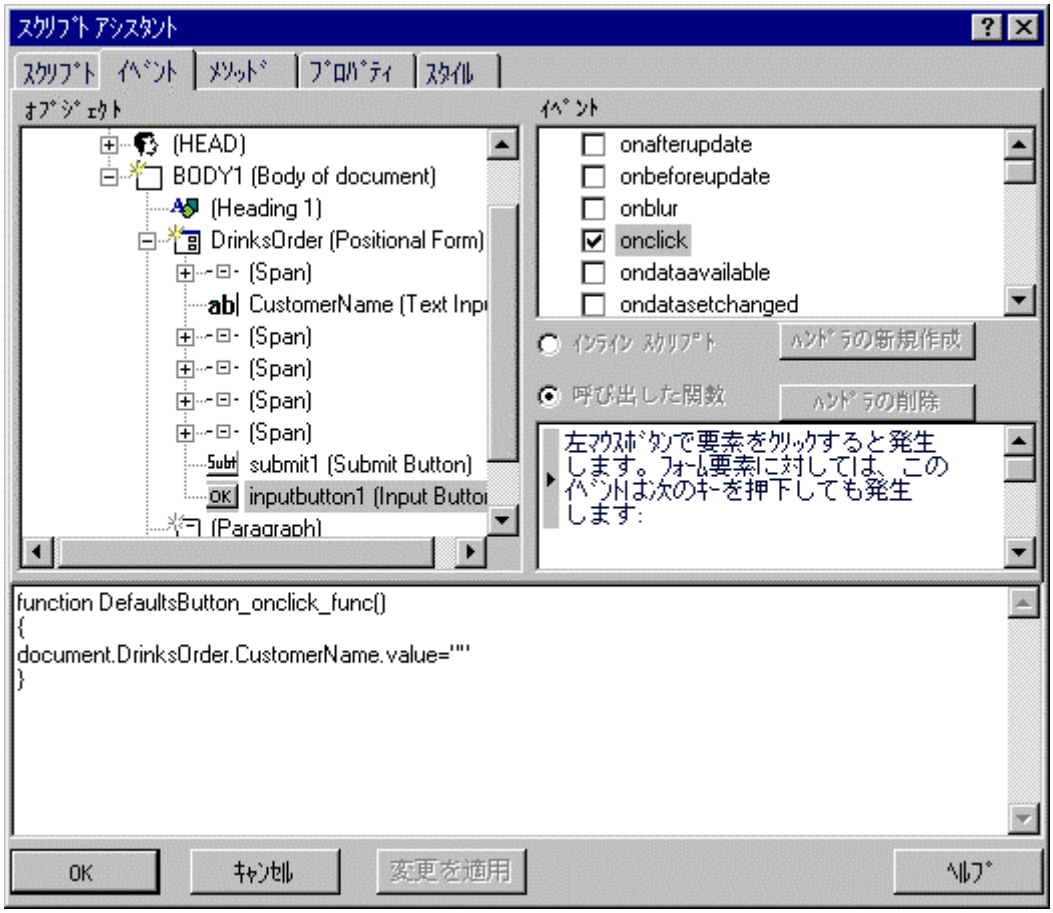



**図 35-1「イベント」タブ**

**「イベント」タブには、次のコントロールがあります。**

- **新しいイベントハンドラを作成するコントロール**
- **イベントハンドラを削除するコントロール。関数のみを削除するか、関数とそれを呼び出すコード (インラインスクリプト) の両方を削除するかを選択できます。**
- **関数またはインラインスクリプトを表示するコントロール**

**「イベント」タブを最初に表示した時点では、これらのコントロールはグレー表示され、無効になっています。[ハンドラの新規作成] を選択する前に、オブジェクトとイベントを選択する必要があります。**

**次の図は「メソッド」タブを示しています。**

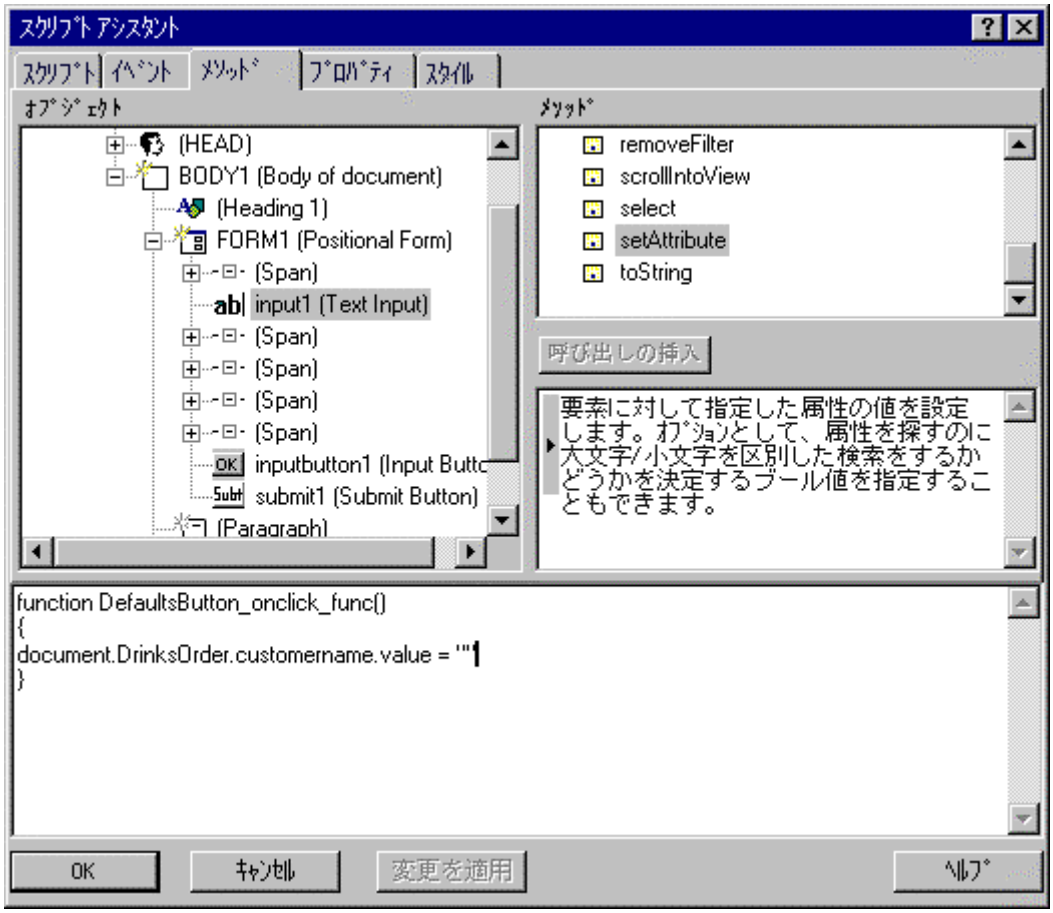



**図 35-2「メソッド」タブ**

**「メソッド」タブには、メソッド呼び出しを挿入するコントロールがあります。このコントロールは、メソッドを選択するまでグレー表示され、無効になっています。**

**下の図は「プロパティ」タブを示しています。「スタイル」タブは、右上のペインにプロパティのかわりにスタイルが一覧される点を除き、「プロパティ」タブと同じです。**

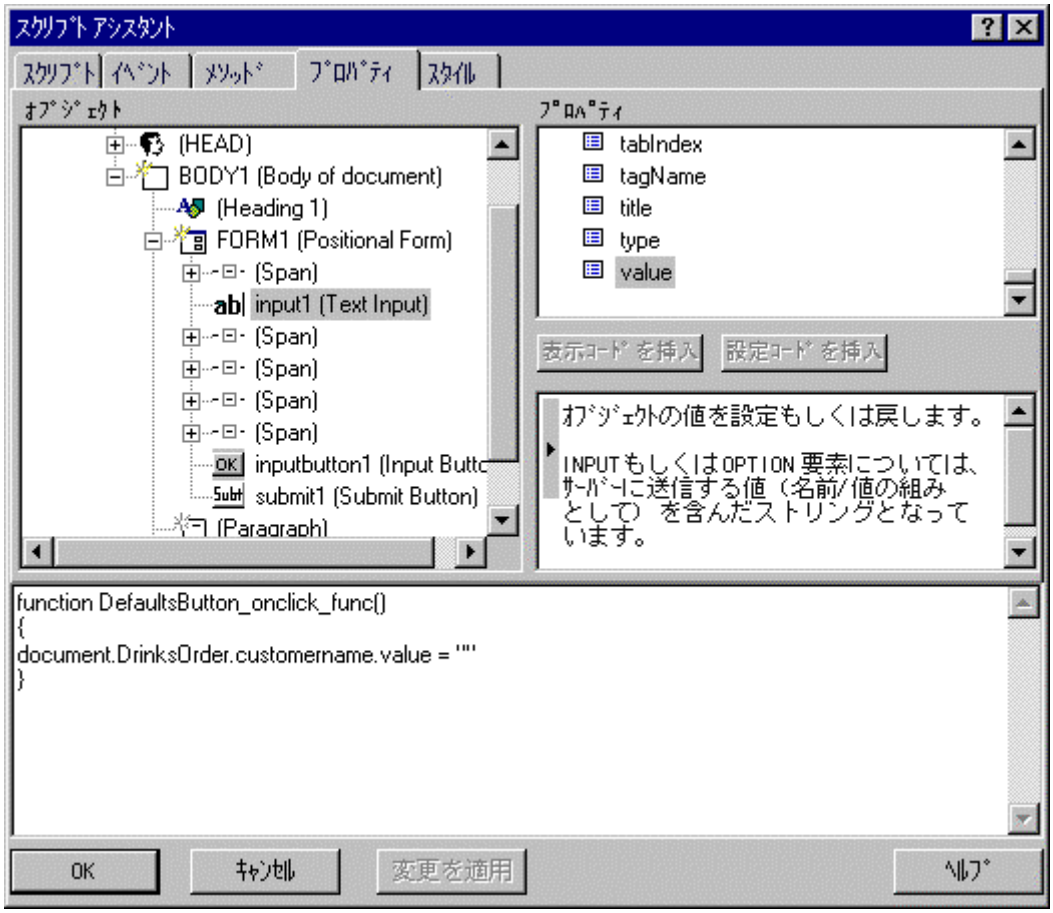



**図 35-3「プロパティ」タブ**

「プロパティ」タブと「スタイル」タブには、次のコントロールがあります。

- プロパティまたはスタイルを取得するコードを挿入するコントロール
- プロパティまたはスタイルを設定するコードを挿入するコントロール

これらのコントロールは、プロパティまたはスタイルを選択するまでグレー表示され、無効になっています。設定できないプロパティもあります。

以降の項では、次の各ビューの詳細を説明します。

- オブジェクトビュー
- イベントビュー
- メソッドビュー
- プロパティビュー
- スタイルビュー
- イベントハンドラコードビュー

### 10.3.1.1 オブジェクトビュー

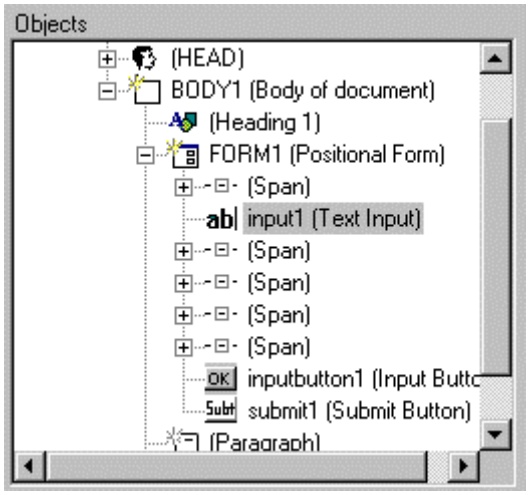



**図 35-4 オブジェクトビュー**

オブジェクトビューには、HTML 文書内のすべての要素がツリー表示されます。要素の横にあるプラス記号 (+) をクリックすると、その要素に属する要素が表示されます。

### 10.3.1.2 イベントビュー

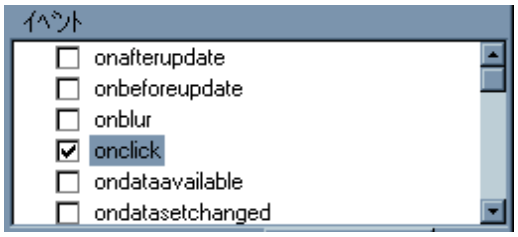



**図 35-5 イベントビュー**

イベントビューには、オブジェクトビューで選択したフォームコントロールで使用可能なすべてのイベントが一覧表示されます。上の図は、HTML のオプションボタンで利用できるイベントの一部を示しています。

### 10.3.1.3 メソッドビュー

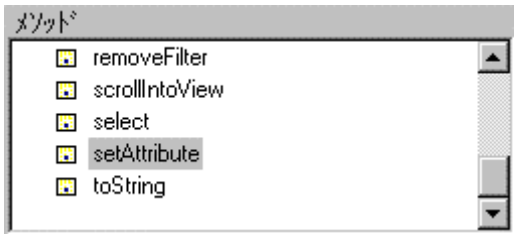



**図 35-6 メソッドビュー**

メソッドビューには、オブジェクトビューで選択した HTML 要素で使用可能なすべてのメソッドが一覧表示されます。上の図は、オプションボタンで利用できるメソッドの一部を示しています。

コントロールでメソッドを呼び出すと、何らかの動作が発生します。たとえば、テキストフィールドで blur メソッドを呼び出すと、そのフィールド以外にフォーカスが移動します。

#### 10.3.1.4 プロパティビュー

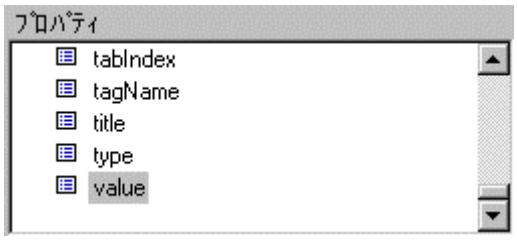



**図 35-7 プロパティビュー**

プロパティとは、コントロールの状態に関する情報のことです。コントロールの状態は、プロパティを取得することによって確認できます。また、プロパティを変更するとコントロールの状態も変更されます。たとえば、ほとんどの ActiveX コントロールと HTML コントロールには Value プロパティがありますが、このプロパティを取得すれば、テキストフィールドの内容を知ることができます。また、Value プロパティに異なる値を設定すると、テキストフィールドの内容を変更できます。上の図は、オプションボタンのプロパティの一部を示しています。

#### 10.3.1.5 スタイルビュー

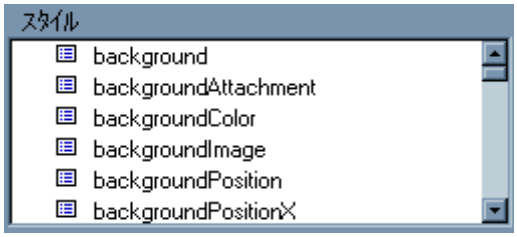



**図 35-8 スタイルビュー**

スタイルとは、要素の表示形式に関する情報のことです。プロパティの場合と同じように、要素の表示形式はスタイルを取得すれば確認できます。要素の表示形式を変更するには、スタイルに異なる値を設定します。たとえば、ほとんどの ActiveX コントロールと HTML コントロールには背景色のスタイルがありますが、このスタイルの値を取得すれば背景色を知ることができます。また、このスタイルに任意の色を設定すると、背景色が変更されます。上の図は、HTML プッシュボタンコントロールのスタイルの一部を示しています。

プロパティと大部分のスタイルは、ダイアログボックスに文字列を入力して設定します。ただし、色スタイルを設定する場合には、48 の基本色と 16 のカスタム色を含むダイアログボックスが表示されます。このダイアログボックスではカスタム色を定義できますが、JavaScript でサポートされている色の範囲は限定されており、サポートされていないカスタム色を設定すると、

サポート範囲内で最近似の色が使用されます。サポートされている色の詳細については、Netscape または Internet Explorer のリファレンスを参照してください。

#### 10.3.1.6 イベントハンドラビュー

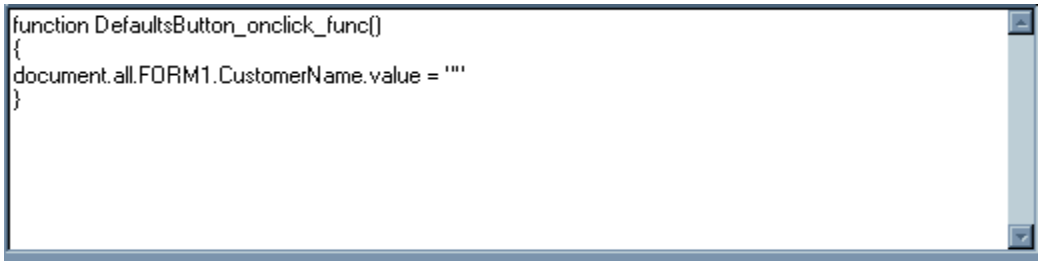



**図 35-9 イベントハンドラビュー**

イベントハンドラビューには、イベントビューで現在選択しているイベントのイベントハンドラのコードが表示されます。上の図は、プッシュボタンの Click イベントを処理する JavaScript イベントハンドラを示しています。このコードは、他のコントロール (テキストフィールド) の Value プロパティをヌル文字列に設定します。

オブジェクトビューでイベントをクリックすると、イベントハンドラビューには、そのイベントを処理するイベントハンドラの現在のコードが表示されます。このコードには、次の 2 通りの方法で変更を加えることができます。

- ペイン内のコードを直接編集する。
- メソッド、プロパティ、またはスタイルを適切なビューでダブルクリックし、メソッドを呼び出すコードや、プロパティまたはスタイルを取得/設定するコードを追加する。

##### 10.3.1.6.1 HTML コントロールのイベントハンドラ

HTML コントロールのイベントを処理するコードは、関数とその呼び出しで構成されます。この 2 つの部分は、互いに切り替えて表示できます。

関数は次の形式で記述します。

function *controlname_event_*func()
{*JavaScript code*
}

各部の意味は次のとおりです。

*controlname* — イベントの検出対象コントロールの名前

*event* — 処理対象のイベント

*Javascript code* — メソッドの呼び出しやプロパティの取得、設定、またはその他の処理を実行するためのコード

次に使用例を示します。

```
function DefaultsButton_onclick_func()
{CustomerName.value=""

}
```

関数名の末尾のカッコ "()" 内には、イベントに応じて引数を含めることができます。スクリプトアシスタントで生成された関数には、このカッコ内にダミーの引数が記述されており、適切な値に置き換える必要があります。

#### 10.3.1.6.2 ActiveX コントロールのイベントハンドラ

ActiveX コントロールのイベントを処理するコードは、メソッドの呼び出し、プロパティやスタイルの取得/設定、あるいはその他の処理 (明示的に記述する場合のみ) で構成され、function で始まる行や関数呼び出しの行はありません。

---

### 注記

- タブを切り替えてオブジェクトを選択する際には、操作の順序に留意する必要があります。最初にイベントハンドラの対象となるコントロールとイベントを選択して [**ハンドラの新規作成**] をクリックします。続いて、「メソッド」タブ、「プロパティ」タブ、または「スタイル」タブに適宜切り替え、イベントハンドラで扱うオブジェクトと、呼び出し、取得、設定などを行うメソッド、プロパティ、スタイルを選択します。タブを切り替える前にイベントハンドラで扱うオブジェクトを選択すると、イベントハンドラのコードが抹消され、新しいコードを作成できなくなります。
- イベントハンドラの対象コントロールの名前は、ハンドラの作成後には変更しないでください。この名前はコード内で使用されており、変更してしまうとコードが正しく実行できなくなります。いったんイベントハンドラを削除すれば、コントロールの名前を変更して、イベントハンドラを再作成できます。

---

## 10.3.2 「スクリプト」タブ

「スクリプト」タブでは、JavaScript の関数を直接編集できます。このタブは次の用途に使用します。

- 他のタブを使用して生成されたコードの内容を確認する。
- 新しいコードを入力して関数を追加する。
- 関数を削除する。

- スクリプトを追加または削除する。

JavaScript に十分に習熟するまでは他のタブを使用し、ポイント & クリック操作でコードを生成することが推奨されます。

次の図は「スクリプト」タブを示しています。

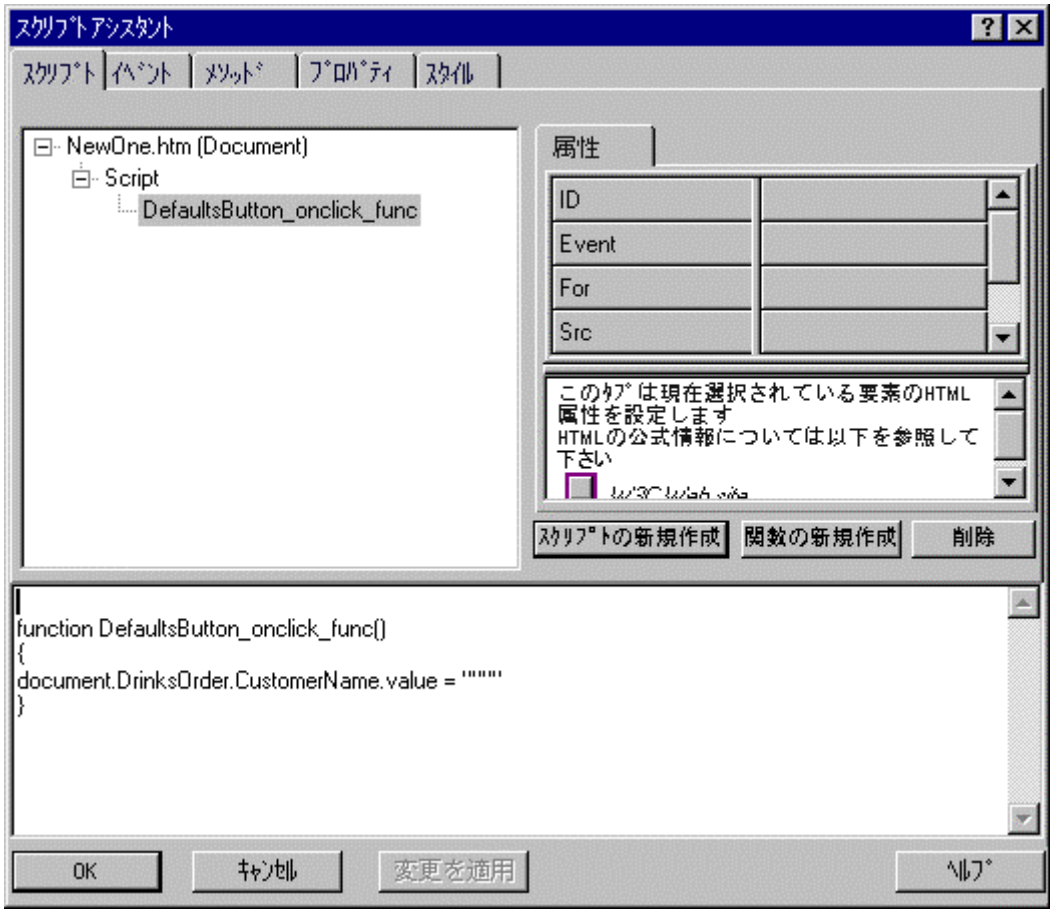



**図 35-10「スクリプト」タブ**

「スクリプト」タブには、次の 4 つのペインがあります。

- 左上のペイン ― スクリプトビューのペイン。文書内のすべてのスクリプトと関数がツリー形式で表示されます。文書名の横にあるプラス記号 (+) をクリックすると文書内のスクリプトが一覧され、スクリプトをクリックすると、そのスクリプト内の関数が表示されます。
- 右上のペイン ― 選択したスクリプトの属性が一覧表示されます。
- 右側中段のペイン ― 選択した属性に関するヘルプが表示されます。
- 下部のペイン ― 選択したスクリプト内の関数のコードが表示されます。

**注記**：Net Express に付属している検証用の JavaScript 関数を呼び出した場合は、その関数もスクリプトビューに表示されます (『フォームの検証』の章を参照してください)。これらの関数は複数の **.js** ファイルとして提供されており、**mfcommon.js** ファイル内の共通関数を使用します。

---

# 10.4 ブラウザ間の互換性

Netscape と Microsoft の Web ブラウザを対象としてコードを作成する場合は、次に示す互換性の問題に留意する必要があります。

- ActiveX コントロールは Internet Explorer だけで動作する。
- アプレットのイベントは Internet Explorer だけで使用できる。
- Internet Explorer では文書内のフォームとフォーム内のコントロールを連番で識別できるが、Netscape のブラウザでは名前の付いていないフォームやコントロールは操作できない。
- スクリプトアシスタントに表示されるイベントの一部は、ブラウザによっては機能しないことがあります。

DHTML とドキュメントオブジェクトモデルに対応していない旧バージョンの Web ブラウザが使用されている可能性も考慮する必要があります。

# 10.5 例

## 10.5.1 例 1：プロパティを設定するイベントハンドラの追加

この項では、第 29 章 (『*CGI ベースのアプリケーションの新規作成*』) で例として使用している飲み物注文フォームに、簡単なイベントハンドラを追加する方法を示します。最初に 2 つ目のプッシュボタンをフォームに追加し、そのボタンがクリックされたときにフォームにデフォルト値を設定するイベントハンドラを追加します。新しいフォームは次のようになります。

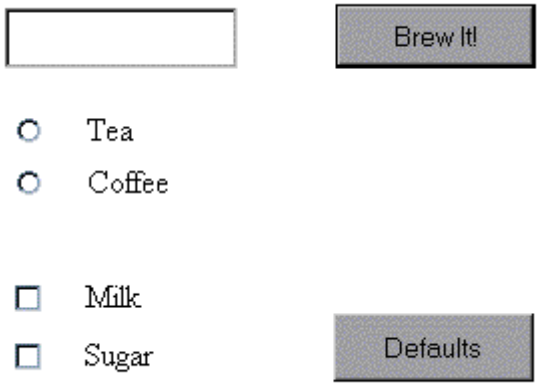



**図 35-11 プッシュボタンを追加した飲み物注文フォーム**

デフォルト設定は次のとおりです。

- 名前フィールドは空白
- 「Coffee」オプションボタンを選択
- 「Milk」チェックボックスを選択
- 「Suger」チェックボックスを選択解除

この例では、HTML フォームにイベントハンドラを追加する方法を示します。『*CGI ベースのアプリケーションの新規作成*』の章に記載されているサンプルを作成していない場合でも、**Net Express\base\demo\bevord_h\bev_h.app** をロードすれば、同じアプリケーションの HTML バージョンを利用できます。

新規ボタンを追加する手順は、次のとおりです。

1. Net Express を起動し、『*CGI ベースのアプリケーションの新規作成*』の章で作成した **bevord_h.app** プロジェクト (または上記の HTML バージョン) を開きます。
2. Form Designer を起動し、**bevord_h.htm** フォームを開きます (Net Express の「プロジェクト」ウィンドウで **bevord_h.htm** をダブルクリックすると、簡単に開ことができます)。
3. フォームにコマンドボタンを追加し、ID プロパティと Name プロパティを DefaultsButton に設定します。ボタンの Value プロパティを Defaults に設定します。
4. スクリプトアシスタントを起動し、オブジェクトビューで DefaultsButton を選択します。
5. イベントビューで onclick イベントをクリックし、[ハンドラの新規作成] をクリックします。次の JavaScript コードがイベントハンドラビューに表示されます。

```
6.   function DefaultsButton_onclick_func1()
7.   {
}
```

   カーソルが閉じかっこの直前に置かれます。

8. DefaultsButton コントロールを選択したまま、「プロパティ」タブに切り替えます。
9. オブジェクトビューで CustomerName コントロールを選択します。

10. Value プロパティをクリックし、[設定コードを挿入] をクリックします。
11. 文字列の入力を求めるダイアログボックスが表示されます。
12. フィールドを空白に設定するため、何も入力せずに [**OK**] ボタンをクリックします。次の JavaScript コードが、イベントハンドラビューのカーソル位置に挿入されます。

    ```
    document.DrinksOrder.CustomerName.value = ""
    ```

    エンドユーザが [Defaults] ボタンをクリックするたびに、このコードが実行され、CustomerName フィールドがクリアされます。

13. オブジェクトビューで CoffeeSelected オプションボタンを選択します。
14. Checked プロパティをクリックし、[設定コードを挿入] をクリックします。
15. プロパティとして True または False の選択を求めるダイアログボックスが表示されます。True をクリックし、[**OK**] ボタンをクリックします。イベントハンドラビューの関数内で、カーソルが位置している位置に、次の JavaScript コードが挿入されます。

    ```
    document.DrinksOrder.DrinkSelected(1).Checked = True
    ```

    エンドユーザが [Defaults] ボタンをクリックするたびに、このコードが実行され、「Coffee」オプションボタンが選択されます。「Tea」オプションボタンの選択を解除するコードを追加する必要はありません。2 つのオプションボタンは同じグループ内にあるため、一方を選択すると、他方は自動的に選択解除されます。

16. 「MilkSelected」チェックボックスについても同じ手順を繰り返します。Checked プロパティを True に設定します。
17. SugarSelected チェックボックスを対象に、同じ手順を繰り返します (Checked プロパティは False に設定します)。

    この時点で、イベントハンドラビューには次のコードが表示されています。

    ```
    function DefaultsButton_onclick_func()
    {
    document.DrinksOrder.CustomerName.value = ""
    document.DrinksOrder.DrinkSelected[1].checked = true
    document.DrinksOrder.MilkSelected.checked = true
    document.DrinksOrder.SugarSelected.checked = false
    }
    ```

    **図 35-12 イベントハンドラビューに表示されるコード**

18. [OK] ボタンをクリックしてスクリプトアシスタントを閉じ、フォームを保存します。アプリケーションを再実行するか、Form Designer ツールバーの [テスト] ボタン  をクリックして、Web ブラウザにフォームを表示します。[**Defaults**] ボタンをクリックするたびに、フォームの各コントロールがデフォルト値に設定されます。

### 10.5.2 例 2：プロパティを取得するイベントハンドラの追加

この項では、コントロールのプロパティを取得し、他のコントロールのプロパティを設定する簡単なイベントハンドラの作成方法を示します。2 つの入力フィールドとプッシュボタンを含む新しいフォームを作成した後、イベントハンドラを追加します。次のようなフォームを作成します。


**図 35-13 プロパティを取得するフォームの一例**

フォームとイベントハンドラの作成手順は次のとおりです。

1. Net Express を起動し、新しいプロジェクトを作成します。
2. 新規の HTML ページを作成します。
3. **mff** ファイルを開き、位置フォームを作成します。
4. 位置フォームにテキストフィールドを追加します。テキストフィールドの名前を InputName に変更します。
5. プッシュボタンを追加し、名前を CopyButton に設定します。ボタンのラベルを Copy に変更します。
6. 2 つ目のテキストフィールドを追加し、フィールド名を CopyName に変更します。
7. スクリプトアシスタントを起動し、オブジェクトビューで CopyButton を選択します。
8. イベントビューで onclick イベントをクリックし、[ハンドラの新規作成] をクリックします。イベントハンドラビューに、次の JavaScript コードが表示されます。

```
function CopyButton_onclick_func1()
{
}
```

   カーソルが閉じかっこの直前に置かれます。

11. CopyButton コントロールを選択したまま、「プロパティ」タブに切り替えます。
12. オブジェクトビューで CopyName コントロールを選択します。
13. Value プロパティをクリックし、[設定コードを挿入] をクリックします。文字列の入力を求めるダイアログボックスが表示されます。デフォルト (文字列なし) をそのまま使用します。イベントハンドラビューのカーソル位置に、次の JavaScript コードが挿入されます。

```
document.all.FORM1.CopyName.value = ""
```

14. デフォルト値 ("") を選択して削除します。カーソルは削除位置に残しておきます。

15. オブジェクトビューで InputName コントロールを選択します。
16. Value プロパティをクリックし、[表示コードを挿入] をクリックします。値を取得する JavaScript コードがカーソル位置に挿入されます。その結果、次のように CopyName の値を InputName の値に設定する文が形成されます。

    ```
    document.all.FORM1.CopyName.value = document.all.FORM1.InputName.value
    ```

17. [**OK**] ボタンをクリックしてスクリプトアシスタントを閉じ、フォームを保存します。アプリケーションを再実行するか、Form Designer ツールバーの [テスト] ボタン  をクリックして、Web ブラウザにフォームを表示します。最初のテキストフィールドに名前を入力します。[**Copy**] ボタンをクリックすると、入力した名前が 2 番目のテキストフィールドに表示されます。

### 10.5.3 例 3：色を設定するイベントハンドラの追加

この項では、マウスポインタを重ねたときにボタンの色を変化させる簡単なイベントハンドラの作成方法を示します。例 2 で作成した CopyButton コントロールの初期の背景色を設定した後、イベントハンドラを追加します。

フォームとイベントハンドラを次の手順で編集します。

1. Net Express を起動し、例 2 で作成したプロジェクトと **.htm** ファイルを開きます。
2. CopyButton コントロールを選択し、その背景色スタイルを Pale Green に変更します。
3. スクリプトアシスタントを起動し、オブジェクトビューで CopyButton を選択します。
4. イベントビューで onmouseover イベントをクリックし、[ハンドラの新規作成] をクリックします。イベントハンドラビューに次の JavaScript コードが挿入されます。

```
5.   function CopyButton_onmouseover_func()
6.   {
}
```

   カーソルが閉じかっこの直前に置かれます。

7. CopyButton コントロールを選択したまま、「スタイル」タブに切り替えます。
8. backgroundColor スタイルをクリックし、[設定コードを挿入] をクリックします。「色の設定」ダイアログボックスが表示されます。
9. 3 行目の左から 3 列目にある明るい緑のセルを選択して [**OK**] をクリックします。イベントハンドラビューのカーソル位置に、次の JavaScript コードが挿入されます。

   ```
   document.all.FORM1.CopyButton.style.backgroundColor = 0x0000FF00
   ```

10. イベントビューで onmouseout イベントをクリックし、[ハンドラの新規作成] をクリックします。イベントハンドラビューに次の JavaScript コードが挿入されます。

```
11.  function CopyButton_onmouseout_func()
12.  {
}
```

カーソルが閉じかっこの直前に置かれます。

13. CopyButton コントロールを選択したまま、「スタイル」タブに切り替えます。
14. backgroundColor スタイルをクリックし、[設定コードを挿入] をクリックします。「色の設定」ダイアログボックスが表示されます。
15. [**色の定義>>**] をクリックします。
16. 色円環の緑色の部分をクリックします。ステップ 2 で設定した Pale Green とほぼ同じ色が選択されるように、ダイアログボックス右側の細長い矩形内の上部をクリックします。[**OK**] をクリックします。イベントハンドラビューのカーソル位置に、色を再設定する JavaScript コードが挿入されます。
17. [**OK**] ボタンをクリックしてスクリプトアシスタントを閉じ、フォームを保存します。アプリケーションを再実行するか、Form Designer ツールバーの [テスト] ボタン  をクリックして、Web ブラウザにフォームを表示します。マウスポインタを [**Copy**] ボタン上に移動すると緑色が濃くなり、[Copy] ボタンの外に移動すると明るい緑色に戻ります。

---



---

# 第 36 章 : フォームの検証

この章では、Net Express に付属している検証用の JavaScript 関数を呼び出して、フォームのエントリフィールドに入力されたデータを検証する方法と、独自に作成した検証関数を追加する方法について説明します。

オンラインヘルプには、この章で説明している機能の使用方法が、手順を追って詳しく説明されています。[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**プログラミング > CGI ベースのアプリケーション > Form Designer > JavaScript とイベント処理 > 概要 - フォーム検証**」を順にクリックしてください。

## 11.1 標準で付属している検証関数

Net Express には、次の各 HTML コントロールを検証する関数が標準で付属しています。

- テキスト入力
- テキストエリア
- パスワード

これらの関数では、フィールドが必要かどうか、およびステータス行やポップアップにメッセージを表示するかどうかを指定できます。ステータス行のメッセージは、検証するフィールドにフォーカスがあるときに表示されます。一方、ポップアップのメッセージは、フィールド以外にフォーカスが移ったとき、および検証が失敗したときに表示されます。メッセージのテキストは変更できます。

フィールドの検証関数は、次のタイミングで実行されます。

- フィールドの内容が変更されたとき
- フォームが送信されたとき

Net Express には、次に挙げる検証関数があります。

| 関数名 | 検証の内容 |
|---|---|
| American Express card number | American Express カードの番号形式との一致を検証する。 |
| Credit card number | 主要クレジットカード (American Express、Mastercard、Visa) との番号形式の一致を検証する。 |
| e-mail address | E メールアドレス形式との一致を検証する。 |
| Mastercard card number | Mastercard カードの番号形式との一致を検証する。 |
| Number | 符号を含まない数値かどうかを検証する。 |
| Number (decimal) | 小数点を含む数値かどうかを検証する (符号も許容される)。 |

| | |
|---|---|
| Number (signed positive) | 正の整数かどうかを検証する。 |
| Number (signed) | 符号付きの整数かどうかを検証する。 |
| Phone number - international | 国際電話の電話番号形式との一致を検証する。 |
| US phone number | 米国の電話番号形式 (10 桁) との一致を検証する。3 桁目と 6 桁目の直後に空白文字が 1 つ挟まれていても許容される。 |
| US social security number | 米国の社会保険番号の形式 (9 桁) との一致を検証する。3 桁目と 5 桁目の直後に空白文字が 1 つ挟まれていても許容される。 |
| US state code | 米国の州を表す 2 文字の略称との一致を検証する。 |
| US zip code | 米国の郵便番号形式 (5 桁または 9 桁) との一致を検証する。 |
| Visa card number | Visa カードの番号形式との一致を検証する。 |

**注記**:主要クレジットカード (American Express、Mastercard、Visa) は特定の個人に発行されているため、番号の有効性は検証されません。

## 11.2 検証例

この項では、検証関数を呼び出す 3 つの例を照会します。これらの例は、**.htm** ファイルを開いており、その中に検証の対象になりうるフィールド (テキストエントリフィールド、テキストエリアフィールド、パスワードフィールド) が 1 つ以上含まれていることを前提としています。

次の図は「確認」ダイアログボックスを示しています。

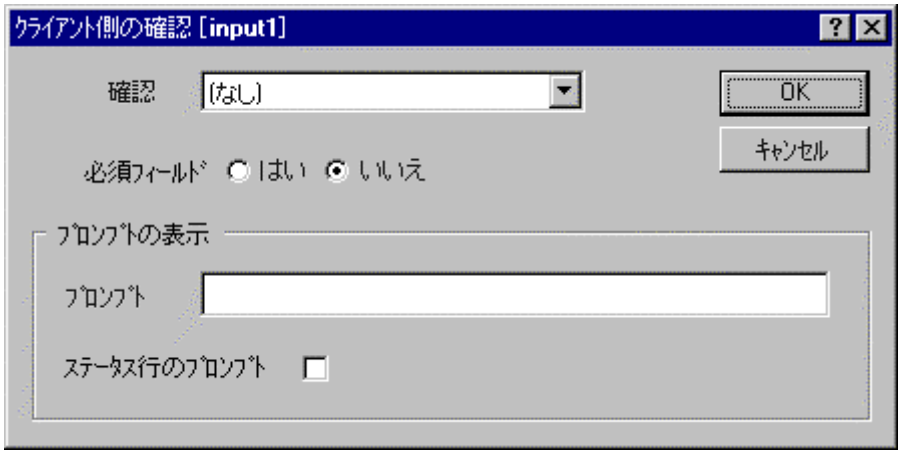



**図 36-1「確認」ダイアログボックス**

### 11.2.1 小数点を含む数値の検証

小数点を含む数値かどうかを検証する関数を呼び出し、対象のフィールドをオプションにする手順を次に示します。

1. 検証するフィールドを選択します。
2. フィールド内を右クリックし、ポップアップメニューから [**確認**] を選択します (または [編集] メニューから [**確認**] を選択します)。「確認」ダイアログボックスが表示されます。
3. 「**確認**」ドロップダウンリストで **Number (decimal)** をクリックします。
4. 「**必須フィールド**」の「**いいえ**」オプションボタンをクリックします。この操作によって、選択したフィールドがオプションフィールドになります。
5. 小数点を含む数値の検証ルーチンに関連付けられたメッセージテキストが「**プロンプト**」フィールドに表示されます。このテキストは必要に応じて編集できます。メッセージをステータス行に表示する場合には、「**ステータス行のプロンプト表示**」をクリックします。
6. [**OK**] をクリックします。

エンドユーザはこのフィールドを空白のまま残すことができますが、小数点を含む数値以外の値を入力した場合には、有効な値をフィールドに再入力するように求めるメッセージが表示されます。

### 11.2.2 Visa カード番号の検証

Visa カード番号の検証関数を呼び出し、フィールドを必須フィールドにする手順を次に示します。

1. 検証するフィールドを選択します。
2. フィールド内を右クリックし、ポップアップメニューから [**確認**] を選択します (または [編集] メニューから [**確認**] を選択します)。「確認」ダイアログボックスが表示されます。
3. 「**確認**」ドロップダウンリストで **Visa card number** をクリックします。
4. 「**必須フィールド**」の「**はい**」オプションボタンをクリックします。この操作によって、選択したフィールドが必須フィールドになります。
5. Visa カード番号の検証関数に関連付けられたメッセージテキストが「**プロンプト**」に表示されます。このテキストは必要に応じて編集できます。メッセージをステータス行に表示する場合には、「**ステータス行のプロンプト表示**」をクリックします。
6. [**OK**] をクリックします。

エンドユーザはこのフィールドに、有効な値を入力する必要があります。有効な値を入力しないと、フィールドに戻って有効な値を入力するように求めるメッセージが表示されます。

### 11.2.3 必須フィールドへの入力確認

「確認」ダイアログボックスでは、標準で付属している検証関数を使用しない場合でも、特定のフィールドを必須フィールドに指定することができます。手順は次のとおりです。

1. 必須フィールドにするフィールドを選択します。
2. フィールド内を右クリックし、ポップアップメニューから [**確認**] を選択します (または [編集] メニューから [**確認**] を選択します)。「確認」ダイアログボックスが表示されます。
3. 「**確認**」ドロップダウンリストで **なし** をクリックします。
4. 「**必須フィールド**」の「**はい**」オプションボタンをクリックします。この操作によって、選択したフィールドが必須フィールドになります。
5. 「確認」ドロップダウンリストの「なし」に関連付けられているメッセージテキストが「**プロンプト**」に表示されます。このテキストは必要に応じて編集できます。メッセージをステータス行に表示する場合には、「**ステータス行のプロンプト表示**」をクリックします。
6. [**OK**] をクリックします。

エンドユーザは、このフィールドに値を入力する必要があります。入力しない場合、フィールドに戻って値を入力するように求めるメッセージが表示されます。

# 11.3 独自の検証関数の追加

標準で付属している検証関数が適切でない場合には、検証関数を独自に作成して、テキスト入力やテキストエリア、およびパスワードのフィールドに適用することができます。そのためには、JavaScript や JScript に関する十分な理解が必要になります。詳細については、『クライアント側のプログラミング』の章を参照してください。

独自に作成した検証関数は、次の 2 通りの方法でフォームに適用できます。

- 他の関数を実行するためのコードと同様に、フォーム内にイベントハンドラとして記述する。

  検証関数は通常、onchange イベントで呼び出します。フォームの onsubmit イベントで呼び出す場合も考えられます。この方法は、検証関数を特定のフォームだけで使用する場合に適しています。

- 作成した検証関数を、Net Express 付属の検証関数に追加する。

  この方法で追加した検証関数は、どの入力フォームでも必要に応じて使用できます。

  標準で付属している検証関数群は 1 つの **.js** ファイル内に記述されており、**bin** ディレクトリ内の **validate.cpt** という名前のファイルから参照されます。独自の検証関数を作成した場合は、その関数を **.js** ファイルとして保存し、**validate.cpt** ファイルを更新する必要があります。独自の検証関数は、テキストエディタやスクリプトアシスタントを使用して新しく作成することもできますが、付属の検証ルーチンのいずれかをコピーして、必要な機能に合わせて編集することもできます。 関数をコピーして編集する手順は、オンラインヘルプに詳しく説明されています。[**ヘルプ > ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**プログラミング > CGI ベースのアプリケーション > Form Designer > JavaScript とイベント処理 > 概要 - フォーム検証 > 方法 > 検証関数の追加**」を順にクリックしてください。

# 第 37 章 : CGI ベースのアプリケーションのディプロイ

この章では、作成したアプリケーションを Web サーバ上にディプロイし、イントラネットやインターネットで運用できる状態にする方法を説明します。また、開発段階で Solo 以外の Web サーバを使用して、アプリケーションをデバッグする方法についても説明しています。

## 12.1 概要

アプリケーションを Web サーバ上に配置し、インターネットやイントラネットで使用できる状態にすることを、ここでは「アプリケーションのディプロイ」と呼んでいます。Net Express の アプリケーションを Form Designer とインターネットアプリケーションウィザードで開発し、Web サーバとして Solo を使用している場合は、環境、Web 共有リソース、および URL がすべて適切に設定されるため、Web サーバの設定を変更する必要はありません。

アプリケーションを運用環境の Web サーバにディプロイした後、次の点を確認する必要があります。

- サーバの Web 共有リソースのパスとアプリケーションの URL
- 配布するフォームファイル
- アプリケーションのプロジェクトビルド設定

この章では、作成したアプリケーションのディプロイ手順を詳しく説明します。また、CGI プログラムを ISAPI プログラムや NSAPI プログラムに変換する手順についても説明しています。すべてのサーバ側プログラムは CGI プログラムとして開発し、ディプロイする準備ができた段階で、ISAPI または NSAPI に変換することが推奨されます。

アプリケーションを Web サーバにディプロイする前に、アプリケーションの配布に関する使用許諾条件に必ず目を通してください。 使用許諾条件は、ReadMe ヘルプ内の エンドユーザライセンス契約に記載されています。 ReadMe ヘルプは [**スタート**] メニューから開くことができます。

### 12.1.1 他の Web サーバを使用したデバッグ

Net Express で開発したプログラムは、Solo 以外の Web サーバを使用する環境でもデバッグできます。デバッグ中には、アプリケーションに変更を加えながらデバッグする作業が容易になるように、Web サーバに共有リソースとして **COBOL** と **CGI-BIN** を設定することが推奨されます。Solo 以外の Web サーバでのディプロイとデバッグの相違点については、後述する『ディプロイとデバッグの手引き』で説明しています。

### 12.1.2 作業例

『ディプロイとデバッグの例』の章には、各種の Web サーバ環境におけるデバッグとディプロイの作業例がいくつか記載されており、以下の内容をより良く理解するための参考資料として利用できます。

## 12.2 サポートされているサーバ

Net Express では、Web プログラムのインターフェイスとして、CGI、ISAPI、または NSAPI を使用するアプリケーションを作成できます。したがって、Net Express アプリケーションは、これらのインターフェイスに完全準拠したあらゆる Web サーバで実行可能です。このマニュアルの執筆時点で、Net Express アプリケーションの動作が試験を通じて確認されているサーバは次のとおり。

Windows NT

- Netscape Enterprise Server
- Microsoft Internet Information Server V4.0
- Netscape Fasttrack Server 2.01
- Microsoft パーソナル Web サーバ

UNIX

- Apache (Micro Focus Object COBOL 4.1 for Unix を使用)

## 12.3 ディプロイとデバッグの手引き

以下の 3 つの項では、Solo 以外の Web サーバでアプリケーションを実行するために必要な手順の概要を示します。それぞれの手順の詳細は、その後の各項で説明しています。アプリケーションを運用環境の Web サーバにディプロイする場合のみ、あるいはアプリケーションのデバッグ時のみに適用すべき情報は、その旨を文中で明記しています。アプリケーションを運用環境の Web サーバにディプロイする場合と、Solo 以外の Web サーバでデバッグする場合の主な違いは次のとおりです。

- デバッグ時には Net Express のデフォルトの Web 共有リソース (**COBOL**、**CGI-BIN**) を使用するが、ディプロイ時には Web 共有リソースの柔軟性を高める必要がある。
- ディプロイ時には、開発に使用したマシンから別のマシンにアプリケーションをコピーする。デバッグ時には、Net Express のライセンスコピーがインストールされている開発マシンを使用する。
- デバッグ時には、サーバ設定にいくつかの変更を加える必要がある (そのような変更はディプロイ時には不要)。

次の 3 つの項は、Solo 以外の Web サーバを使ったディプロイ手順またはデバッグ手順の概要を示しています。

- Windows サーバでのディプロイ手順
- UNIX サーバでのディプロイ手順

- デバッグの手順

### 12.3.1 Windows サーバでのディプロイ手順

この項では、アプリケーションを運用環境の Web サーバにディプロイする手順の概要を示します。この手順は、運用環境の Web サイトで、Web 共有リソースとして Net Express のデフォルト (**COBOL**、**CGI-BIN**) 以外のリソースを使用する場合を想定しています。

1. 使用するランタイムシステムを選択します。

   Form Designer とインターネットアプリケーションウィザードで開発したアプリケーションには、動的に結合されるシングルスレッドの共有ランタイムシステムがデフォルトで使用されます。アプリケーションの種類によっては、その他のランタイムシステムが望ましい場合や、その他のランタイムシステムを使用すべき場合があります。

   『ランタイムシステムの選択』の項を参照してください。

2. Web サーバの共有リソースとアクセス権を設定します。

   この手順は、基本的には Web サーバと Web サイトの設定作業の一部であり、通常は一度だけ実施します。ただし、ここで設定した内容は、Net Express で作成したアプリケーションを当該サーバにディプロイするための作業内容に影響を与えます。

   『Web サーバの設定』の項を参照してください。

3. Net Express で作成したアプリケーションの全ファイルを、ディプロイ用にコピーします。

   使用している Web サーバ、選択した Web 共有リソース名、およびアプリケーションのビルド方法によっては、COBOL のソースファイルと HTML ファイルに多少の変更が必要になることがあります。そのような場合には、開発に使用したソースとは別にコピーを作成し、そのコピーに変更を加えることが推奨されます。

   『ディプロイ用コピーの作成』の項を参照してください。

4. Web 共有リソース名に応じて、必要であればアプリケーションの URL を変更します。

   Net Express では、**COBOL** と **CGI-BIN** が Web 共有リソースとして使用されることを前提として、コードが自動生成されます。その他の Web 共有リソース名を使用する場合は、COBOL コードと HTML コードを共有リソース名に合わせて更新する必要があります。

   『アプリケーションの URL 変更』の項を参照してください。

5. アプリケーションでサーバ側の状態機構 (『サーバ側のプログラミング』の章を参照) を使用する場合は、CGI に sstate モジュールを追加する必要があります。データアクセスウィザードで生成したアプリケーションは、すべてこの機構を使用します。

『ディプロイ済みアプリケーションへの sstate の追加』の項を参照してください。

6. サーバの API を選択し、ターゲット Web サーバ用にアプリケーションをリビルドします。

   アプリケーションをリビルドして、ステップ 4 で追加したすべての変更を反映します。この段階で、サーバ API として CGI のかわりに ISAPI や NSAPI を選択してアプリケーションをリビルドすることも可能です。

   『アプリケーションのリビルド』の項を参照してください。

7. Web サーバにアプリケーションをインストールします。

   『アプリケーションのディプロイ』の項を参照してください。

8. Object COBOL のオブジェクト指向機能を使用する ISAPI アプリケーションの場合は、Net Express ランタイムシステムの設定を変更する必要があります。また、Windows NT でのサーバアクセス権の設定が必要になることもあります。

   『ISAPI に準拠した Object COBOL アプリケーションのディプロイ』の項を参照してください。

9. NSAPI アプリケーションの場合のみ、Netscape サーバの設定ファイルの一部を変更する必要があります。

   『NSAPI アプリケーションを実行する前に』の項を参照してください。

### 12.3.2 UNIX サーバでのディプロイ手順

この項では、アプリケーションを UNIX 上の Web サーバで運用する場合のディプロイ手順の概要を示します。以下の手順は、運用環境の Web サイトにディプロイする際に、Web 共有リソースとして Net Express のデフォルト (**COBOL**、**CGI-BIN**) 以外のリソースを使用する場合を想定しています。

1. UNIX サーバに Micro Focus COBOL for UNIX の 4.1 以降のバージョンをインストールします。
2. UNIX サーバに SCP をインストールします (詳細については、『***UNIX オプションユーザガイド***』を参照してください)。
3. Web サーバの共有リソースとアクセス権を設定します。

   この手順は、基本的には Web サーバと Web サイトの設定作業の一部であり、通常は一度だけ実施します。ただし、ここで設定した内容は、Net Express で作成したアプリケーションを当該サーバにディプロイするための作業内容に影響を与えます。

   『Web サーバの設定』の項を参照してください。

4. Net Express で作成したアプリケーションの全ファイルを、ディプロイ用にコピーします。

使用している Web サーバ、選択した Web 共有リソース名、およびアプリケーションのビルド方法によっては、COBOL のソースファイルと HTML ファイルに多少の変更が必要になることがあります。そのような場合には、開発に使用したソースとは別にコピーを作成し、そのコピーに変更を加えることが推奨されます。

『ディプロイ用コピーの作成』の項を参照してください。

5. アプリケーションでサーバ側の状態機構 (『サーバ側のプログラミング』の章を参照) を使用する場合は、CGI に sstate モジュールを追加する必要があります。

   『ディプロイ済みアプリケーションへの sstate の追加』の項を参照してください。

6. Web 共有リソース名に応じて、必要であればアプリケーションの URL を変更します。

   Net Express では、**COBOL** と **CGI-BIN** が Web 共有リソースとして使用されることを前提として、コードが自動生成されます。その他の Web 共有リソース名を使用する場合は、COBOL コードと HTML コードを共有リソース名に合わせて更新する必要があります。UNIX では実行可能ファイルの拡張子が異なります。また、大文字と小文字が区別される点も考慮してください。

   『アプリケーションの URL 変更』の項を参照してください。

7. Web サーバにアプリケーションをコピーします。

   『UNIX へのアプリケーションのパブリッシュ』の項を参照してください。

### 12.3.3 デバッグの手順

この項では、Solo 以外の Web サーバでアプリケーションの開発とデバッグを行うための手順の概要を示します。以下の手順は、アプリケーションの開発とデバッグで Net Express のデフォルトの Web 共有リソース (**COBOL** と **CGI-BIN**) を使用することを前提としています。デフォルトの Web 共有リソースを使用すれば、新しいフォームやプログラムを開発する際の作業量が軽減されます。

1. 使用するランタイムシステムを選択します。

   Form Designer とインターネットアプリケーションウィザードで開発したアプリケーションには、シングルスレッドの共有ランタイムシステムがデフォルトで使用されます。アプリケーションの種類によっては、その他のランタイムシステムが望ましい場合や、その他のランタイムシステムを使用すべき場合があります。

   『ランタイムシステムの選択』の項を参照してください。

2. Web サーバの共有リソースとアクセス権を設定します。

フォームと HTML ページ用に COBOL、プログラム用に CGI-BIN という 2 つの共有リソースが必要です。

『Web サーバの設定』の項を参照してください。

3. サーバの API を選択し、ターゲット Web サーバ用にアプリケーションをリビルドします。

   この段階で、サーバ API として CGI のかわりに ISAPI や NSAPI を選択してアプリケーションをリビルドすることも可能です。 Net Express では、これらの API のいずれを使用するアプリケーションもデバッグできます。ISAPI や NSAPI で保護違反が発生すると、通常は Web サーバを閉じて再起動する必要があるため、最初の開発には CGI を使用することが推奨されます。

   『アプリケーションのリビルド』の項を参照してください。

4. Object COBOL のオブジェクト指向機能を使用する ISAPI アプリケーションの場合は、Net Express ランタイムシステムの設定を変更する必要があります。また、Windows NT でのサーバアクセス権の設定が必要になることもあります。

   『ISAPI に準拠した Object COBOL アプリケーションのディプロイ』の項を参照してください。

5. NSAPI アプリケーションの場合のみ、Netscape サーバの設定ファイルの一部を変更する必要があります。

   『NSAPI アプリケーションを実行する前に』の項を参照してください。

この手順を最後まで済ませると、サーバ側プログラムに Solo 以外の Web サーバを使用して、アプリケーションの編集、ビルド、デバッグのサイクルを実行できます。

### 12.3.4 ランタイムシステムの選択

Form Designer とインターネットアプリケーションウィザードで開発したアプリケーションには、動的に結合されるシングルスレッドの共有ランタイムシステムがデフォルトで使用されます。つまり、実行可能ファイルは起動時に共有ランタイムシステムに動的にリンクされ、シングルスレッドで動作します。実行可能プログラムは、Windows のレジストリ内でランタイムシステムを検索します。

ただし、作成するアプリケーションが次のいずれかに該当する場合には、ランタイムシステムを変更する必要があります。

- アプリケーションを ISAPI プログラムまたは NSAPI プログラムとしてディプロイする

  ISAPI や NSAPI を使用するプログラムは、マルチスレッドで動作する必要があります。『共有ランタイムシステムを使用する ISAPI/NSAPI プログラムのビルド』の項に従ってプログラムをリンクしてください。

- 共有ランタイムシステムを必要としないスタンドアローン型の実行可能プログラムを作成する

  1 つのサーバで実行する COBOL CGI プログラムが 1〜2 つに過ぎず、COBOL ランタイムシステムを別ファイルとしてサーバ上に配置するオーバーヘッドを回避する必要がある場合を除き、この方法は推奨されません。『静的ランタイムシステムを使用する CGI プログラムのビルド』の説明に従ってプログラムをリンクしてください。

---

**注記**: マルチスレッドで動作する CGI アプリケーションも作成可能です。その場合には、マルチスレッドの共有または静的ランタイムシステムとリンクします。方法については、『共有ランタイムシステムを使用する CGI プログラムのビルド』と『静的ランタイムシステムを使用する CGI プログラムのビルド』の項を参照してください。

---

### 12.3.5 Web サーバの設定

この項では、インターネットアプリケーションを実行するための Web サーバの設定方法を説明します。サーバによって細かい手順が異なるため、詳細については使用するサーバのマニュアルを参照してください。

ディプロイ時とデバッグ時ではサーバの設定方法が異なるため、次の 2 つの項に分けて説明しています。

- ディプロイ用の Web サーバ設定
- デバッグ用の Web サーバ設定

#### 12.3.5.1 ディプロイ用の Web サーバ設定

この項では、UNIX での Web サーバの設定については説明していません。Web 共有リソースに関する情報は、すべてのオペレーティングシステムに共通ですが、Application Server に関する記載情報は Windows NT と Windows 95 だけが対象です。UNIX へのディプロイについては、『*UNIX オプションユーザガイド*』を参照してください。

アプリケーションをディプロイする際には、次の手順で Web サーバを設定します。

1. 共有ランタイムシステムを使用するアプリケーションをディプロイする場合は、ターゲットマシンに Application Server をインストールします。その結果、Net Express で作成し、ターゲットマシンで実行するアプリケーションに応じて、適切なランタイムシステムが設定されます。

   ---

**注記**: UNIX サーバでは、Object COBOL for UNIX 4.1 をインストールする必要があります。

2. Web サーバの管理ツールを起動します (具体的な方法については、Web サーバのマニュアルを参照してください)。
3. 以下の表に示す各 Web 共有リソースを設定します。Web 共有リソースとは、Web サーバが動作するマシン上にあり、特定の URL にマッピングされているディレクトリやフォルダのことです。

   この表は、Net Express で各種の共有リソースに使用されるデフォルトの名前を示しています。Net Express のデフォルト名以外の Web 共有リソース名を使用する場合は、『アプリケーションの URL 変更』の説明に従ってアプリケーションのファイルを編集してください。なお、ISAPI アプリケーションや NSAPI アプリケーションをディプロイする場合、および UNIX サーバにディプロイする場合には、実行可能ファイル名がやや異なるため、常に Web 共有リソース名に合わせてアプリケーションのファイルを編集する必要があります。

| Web 共有リソース | Net Express でのデフォルト名 | アクセス権 | 説明 |
|---|---|---|---|
| *bin-share* | /CGI-BIN/ | 実行 | アプリケーションの実行可能ファイル (Windows の .exe ファイルと .dll ファイル) を格納する。 |
| *form-share* | /COBOL/ | 読み取り専用 | アプリケーションのフォームと HTML ページ (.htm ファイル) を格納する。 |
| *image-share* | /COBOL/ | 読み取り専用 | 多数の画像を使用するアプリケーションでは、それらの画像ファイルを独立した共有リソースに格納すれば、アプリケーションの保守が容易になります。<br><br>HTML 画像の Image Source プロパティや、ActiveX イメージコントロールの Picture Path プロパティを、共有リソースの URL に合わせて変更する必要があります。 |

**注記**

- 上記の Web 共有リソースの設定が、すべてのサーバに適用できるとは限りません。たとえば、Netscape FastTrack は CGI と Document という 2 つの共有リソースの概念を持っており、FastTrack サーバの管理時には実行可能ファイル用に CGI 共有リソース、その他のファイル用に Document 共有リソースを設定します。FastTrack のインストール時には、ドキュメントと CGI 用のデフォルト Web 共有リソースが複数設定されます。これらの Web 共有リソースもアプリケーションのディプロイに使用できます。
- Netscape など一部の Web サーバでは、Web 共有リソース名の大文字と小文字が区別されます。たとえば、**/cgi-bin/** と **/CGI-BIN/** は異なる共有リソースとして扱われます。

---

Net Express のデフォルトの Web 共有リソース名を使用し、ランタイムシステムを変更せず、しかもディプロイ時に ISAPI または NSAPI を使用しない場合は、『アプリケーションのディプロイ』の項に進んでください。

### 12.3.5.2 デバッグ用の Web サーバの設定

Net Express でアプリケーションをデバッグする際には、次の手順で Web サーバを設定します。

1. Web サーバの管理ツールを起動します (具体的な方法については、Web サーバのマニュアルを参照してください)。
2. 以下の表に示す各 Web 共有リソースを設定します。Web 共有リソースとは、Web サーバが動作するマシン上にあり、特定の URL にマッピングされているディレクトリやフォルダのことです。

   この表は、Net Express で各種の共有リソースに使用されるデフォルトの名前を示しています。Net Express のデフォルト名以外の Web 共有リソース名を使用する場合は、『アプリケーションの URL 変更』の説明に従ってアプリケーションのファイルを編集してください。

| Web 共有リソース名 | アクセス権 | 位置 |
|---|---|---|
| **/CGI-BIN/** | 実行 | Net Express プロジェクトの実行可能ファイル (**.exe** ファイル、**.dll** ファイル) を格納するディレクトリ。これはソースディレクトリのサブディレクトリの 1 つです。 Net Express では、デフォルトで **debug** と **release** という 2 つのサブディレクトリがあり、前者がデバッグビルド用、後者がリリースビルド用です。 |
| **/COBOL/** | 読 | プロジェクトのソースディレクトリ。 |

み
取り
専
用

3. Web サーバが Windows NT のサービスとして動作している場合には、匿名ログオンを拒否するように設定します。また、Web サーバが特定のユーザアカウントとして動作する場合は、そのユーザアカウントに管理者権限と、「サービスとしてログオン」のユーザ権利を与える必要があります。

   Web サーバは通常、匿名ログオンで実行されるように設定されており、すべてのユーザがイントラネットやインターネットを介して、名前とパスワードを指定せずにアクセスできます。匿名ログオンでは、Web サーバの実行権限は制限されており、デバッガを起動することはできません。

   匿名ログオンを許可しない場合、他のユーザは適切なユーザ名とパスワードを通知しない限り、Web サーバに接続できません。ただし、管理者がローカルで接続する (Web サーバが動作しているマシンで接続する) 場合には、Web サーバに同じ権限が与えられるため、アニメータ (Net Express のデバッガ) の起動が可能になります。

---

**注記**: Web サーバとして Windows 2000 の ISS を使用しており、アプリケーション保護レベルを「中」または「高」に設定している場合、デバッガを起動可能にするには、デスクトップのセキュリティを変更する必要があります。 Net Express のコマンドプロンプトで次のコマンドを実行します。

```
cblcored -i
```

現在のユーザがログアウトすると、デスクトップのセキュリティは標準に戻ります。

---

## 12.3.6 ディプロイ用のコピーの作成

---

**注記**: アプリケーションをデバッグ用に設定している場合には、この手順はスキップしてください。

---

次のいずれか 1 つ以上の項目に該当する場合には、コピーを作成する必要があります。

- Net Express のデフォルトとは異なる Web 共有リソース名を使用するため、アプリケーションのディプロイ前にアプリケーションファイルを変更する必要がある (『Web サーバの設定』の項を参照)
- デフォルト以外のランタイムシステムを使用する (『ランタイムシステムの選択』の項を参照)
- UNIX サーバにディプロイする (『UNIX サーバでのディプロイ手順』の項を参照)
- アプリケーションでサーバ側状態機構を使用する (『サーバ側のプログラミング』の章を参照)

アプリケーションの開発に使用したファイルはできる限り編集せず、それらのファイルのコピーを作成して、コピーを編集対象にしてください。開発マシンにコピー用の新しいディレクトリを作成し、プロジェクト ディレクトリにあるすべてのファイルを、そのディレクトリ内にコピーします。Net Express でプロジェクトを開いている場合は、そのプロジェクトを閉じます。**.app** ファイル (プロジェクトファイル) は、Net Express で開いているときにはコピーできません。

### 12.3.7 アプリケーションの URL の変更

---

**注記**: アプリケーションをデバッグ用に設定している場合には、この手順はスキップしてください。

---

Net Express では、HTML ページ (**.htm** ファイル) が **COBOL**、実行可能ファイルが **CGI-BIN** という Web 共有リソース内にそれぞれ格納されていることを前提として、アプリケーションが生成されます。他の Web 共有リソース名を使用している場合には、次の URL を変更する必要があります。

- すべての入力フォームに含まれる CGI の URL
- すべてのフォームに含まれる画像ファイルの URL
- DISPLAY 動詞で HTML ページを出力する各 COBOL ソースファイルに含まれる出力ページの URL

  インターネットアプリケーションウィザードによって生成されるアプリケーションは DISPLAY 動詞を使用しません。この変更が必要になるのは、アプリケーション内で明示的に DISPLAY 動詞を使用した場合だけです (『サーバ側のプログラミング』の章を参照してください)。

入力フォームに含まれる CGI の URL を変更するには、入力フォームごとに次の手順を実行します。

1. 「プロジェクト」ウィンドウの左ペインでフォームの **.htm** ファイル右クリックし、コンテキストメニューの [**編集**] をクリックして、そのフォームを Form Designer にロードします。
2. Action プロパティをクリックして、次のように修正します。

*|binshare|program.ext*

各パラメータの意味は次のとおりです。

| | |
|---|---|
| *bin-share* | 実行ファイルの Web 共有リソース |
| *program* | プログラムのファイル名 |
| *ext* | プログラムファイルの拡張子。Windows プラットフォームの CGI プログラムの拡張子は **.exe** です (アプリケーションを ISAPI プログラムとしてディプロイする場合、この値は **.dll** になります。NSAPI プログラムの拡張子はユーザ側で定義します)。<br><br>UNIX サーバでは、CGI はシェルスクリプトから実行されます。シェルスクリプトの拡張子は **.sh** です。 |

画像ファイルの URL を変更するには、画像を含む各フォームを Form Designer にロードし、それぞれの画像に対して次の手順を実行します。

1. 画像をクリックします。
2. Image Source プロパティ (HTML 画像) または Image Path プロパティ (ActiveX イメージコントロール) をクリックし、次のように変更します。

   *|image-share|image.ext*

   各パラメータの意味は次のとおりです。

| | |
|---|---|
| *image-share* | 画像ファイルの Web 共有リソース |
| *image* | 画像ファイルの名前 |
| *ext* | 画像のファイル拡張子 |

DISPLAY 動詞で出力される HTML フォームのパスを変更する手順は次のとおりです。

1. COBOL ソースコード内で IS EXTERNAL FORM IDENTIFIED BY 句を検索します。
2. ファイル名を次のように修正します。

   "*fullpath**htmlfile*.htm"

   *fullpath* には、Web サーバが動作しているマシン上の **.htm** ファイルの出力先を示す絶対パスを指定します (Web 共有リソース名ではありません)。

### 12.3.8 ディプロイしたアプリケーションへの sstate の追加

サーバ側のサポート機構 (『サーバ側のプログラミング』の章を参照) では、sstate と呼ばれるモジュールが使用されます。このモジュールは、COBOL ランタイムシステムの一部ではあり

ません (このモジュールのソースコードは標準で付属しており、必要に応じて変更を加えることができます)。そのため、sstate を使用するアプリケーションをディプロイするときには、常に sstate を CGI、ISAPI、または NSAPI の実行可能プログラムにリンクする必要があります。アプリケーションでこの機構を使用しない場合には、この項はスキップしてください。

---

**注記**: インターネットアプリケーションウィザードで生成されるデータアクセスアプリケーションは、この機構を使用します。

この項では、作業手順を Windows プラットフォームと UNIX プラットフォームに分けて説明します。

---

#### Windows

Windows では、**sstate.obj** を実行可能プログラムに直接リンクできます。

1. Net Express の「プロジェクト」ウィンドウの左ペインで CGI のメイン実行可能ファイル (**.exe** ファイル) を右クリックし、コンテキストメニューの [**ビルド設定**] をクリックします。
2. 「**リンク**」タブをクリックします。
3. [**カテゴリ > 高度な設定**] を順にクリックします。
4. 「**これらの OBJ とリンクする**」に **sstate** と入力し、[**OK**] をクリックします。

#### UNIX

UNIX でディプロイするには、**sstate.int** を Net Express のプロジェクトに追加し、UNIX オプションとともに発行します。発行した **sstate.int** は、CGI に自動的にリンクされます。

1. [**プロジェクト > プロジェクトへファイルを追加**] を順に選択します。
2. 「開く」ダイアログボックスで、**Net Express\unix\cgi-sup** フォルダに移動します (このフォルダは、UNIX オプションをインストールした場合のみ利用可能です)。
3. [**ファイルタイプ > すべてのファイル**] を順に選択し、リストから **sstate.int** を選択します。
4. このファイルがプロジェクトディレクトリ以外のディレクトリに位置していることを警告するメッセージボックスが表示されます。[**OK**] ボタンをクリックして、プロジェクトディレクトリにコピーします。

### 12.3.9 アプリケーションのリビルド

この項では、アプリケーションをターゲットマシンにディプロイするために、アプリケーションに含まれるプログラムをリビルドする手順を説明します。次のいずれか 1 つ以上の項目に該当する場合は、アプリケーションをリビルドする必要があります。

- ターゲット Web サーバでデフォルト以外の共有リソース名を使用するために URL を変更した
- ランタイムシステムを変更する必要がある (詳細については、『ランタイムシステムの選択』の項を参照してください)
- アプリケーションを ISAPI または NSAPI 用にリビルドする
- マルチスレッド CGI を使用する

  CGI をマルチスレッド化する必要があるのは、CGI プログラムをマルチスレッドアプリケーションとして作成した場合のみです。詳細については、Net Express の [**ヘルプ**] メニューで [**ヘルプトピック**] をクリックし、オンラインヘルプの索引でマルチスレッドのエントリを見つけ、対応するページに記載されている情報を参照してください。

アプリケーションのビルド方法について、次の各項で説明します。

- 静的ランタイムシステムを使用する CGI プログラムのビルド
- 共有ランタイムシステムを使用する CGI プログラムのビルド
- 共有ランタイムシステムを使用する ISAPI/NSAPI プログラムのビルド

#### 12.3.9.1 静的ランタイムシステムを使用する CGI プログラムのビルド

静的ランタイムシステムの使用が推奨されるのは、Web サーバに少数 (1〜2 つ) の COBOL CGI プログラムを実装し、COBOL ランタイムシステムを別ファイルとしてサーバ上に配置するオーバーヘッドを回避する方が望ましい場合だけです。

CGI プログラムを静的ランタイムシステム用にビルドする手順は次のとおりです。

1. Net Express の「プロジェクト」ウィンドウの左ペインで CGI のメイン実行可能ファイル (**.exe** ファイル) を右クリックし、コンテキストメニューの [**ビルド設定**] をクリックします。
2. 「**リンク**」タブをクリックします。
3. 「**静的**」と「**シングルスレッド**」を選択します。
4. アプリケーションをリビルドします。

アプリケーションをマルチスレッドランタイムシステムにリンクするには、上記のステップ 3 で「**マルチスレッド**」オプションボタンをクリックします。この操作が必要になるのは、マルチスレッドを利用する CGI アプリケーションを作成した場合のみです。 詳細については、Net Express の [**ヘルプ**] メニューで [**ヘルプトピック**] をクリックし、オンラインヘルプの索引で「**マルチスレッド**」のエントリを見つけ、対応するページに記載されている情報を参照してください。

#### 12.3.9.2 共有ランタイムシステムを使用する CGI プログラムのビルド

CGI を共有ランタイムシステムとリンクしてビルドする場合は、ランタイムシステムファイルを提供する Net Express の Application Server を、アプリケーションのディプロイ先マシンにインストールする必要があります。

Form Designer とインターネットアプリケーションウィザードで作成したアプリケーションは、動的に結合される共有ランタイムシステムと常にリンクされます。デフォルトのリンク設定をいったん変更した後、元に戻すには、「ビルド設定」ダイアログボックスを使用します。

1. Net Express の「プロジェクト」ウィンドウの左ペインで CGI のメイン実行可能ファイル (**.exe** ファイル) を右クリックし、コンテキストメニューの [**ビルド設定**] をクリックします。
2. 「**リンク**」タブをクリックします。
3. 「**静的**」と「**シングルスレッド**」を選択します。

   マルチスレッドを利用する CGI アプリケーションを作成した場合には、「**マルチスレッド**」オプションボタンをクリックすれば、マルチスレッドランタイムシステムとリンクできます。 詳細については、Net Express の [**ヘルプ**] メニューで [**ヘルプトピック**] をクリックし、オンラインヘルプの索引で「**マルチスレッド**」のエントリを見つけ、対応するページに記載されている情報を参照してください。

4. 「**動的**」チェックボックスを選択します。

   この操作によって、実行可能プログラムは Windows のレジストリ内を検索し、Application Server または Net Express ランタイムシステムを検出できるようになります。

5. アプリケーションをリビルドします。

#### 12.3.9.3 共有ランタイムシステムを使用する ISAPI/NSAPI プログラムのビルド

ISAPI や NSAPI を使用するプログラムをビルドするには、Net Express のプロジェクトを次のように変更する必要があります。

- **.exe** 実行可能ファイルではなく、**.dll** 実行可能ファイルとしてプログラムをビルドする。
- コンパイル時に ISAPI プログラムや NSAPI プログラムに必要なエントリポイントを生成するコンパイラ指令を追加し、マルチスレッドの共有ランタイムシステムにプログラムをリンクする (ISAPI プログラムと NSAPI プログラムはマルチスレッドプログラム)
- 追加ライブラリをリンクする (NSAPI のみ)

---

**注記**

- ISAPI/NSAPI アプリケーションがリレーショナルデータベースを使用する場合、データベースへのアクセスに使用する ODBC ドライバには、スレッドセーフであることが求められます。 Microsoft Access はスレッドセーフではないため、ISAPI/NSAPI サーバ側プログラムでの使用は推奨されません。
- ISAPI/NSAPI Web サーバによって並行してロードされる可能性がある複数の COBOL **.dll** ファイル内に、同じエントリポイント名を含めることはできません。

---

プログラムを **.dll** 実行可能ファイルとしてビルドする手順は次のとおりです。

1. 「プロジェクト」ウィンドウの左ペインで **.exe** を右クリックし、コンテキストメニューで [**ビルドタイプから削除**] をクリックします。
2. 「ビルド構造から削除」ダイアログボックスで、「**ビルド構造の全体を削除します**」チェックボックスの選択を解除し、「**このターゲットをすべてのビルドタイプから削除します**」を選択して [**削除**] ボタンをクリックします。
3. 「プロジェクト」ウィンドウの左ペインで、元の **.exe** ファイルのビルド時にリンクした全 **.obj** ファイルを選択し、右クリックしてコンテキストメニューから [**選択されたファイルをパッケージ化 > 動的リンクライブラリ**] を順に選択します。
4. 「新規に定義: 動的リンクライブラリ」ダイアログボックスで [**作成**] をクリックします

コンパイラ指定とビルド設定を変更する手順は次のとおりです。

1. ISAPI または NSAPI のコンパイラ指令を、下記の表を参考にして設定します。

   プログラムの冒頭に $SET 文を記述すると、そのプログラムをコンパイルするたびに、設定したコンパイラ指令が自動的に使用されます。

   次に使用例を示します。

   ```
   $set webserver(nsapi,myentry) case reentrant(2)
   ```

   ドル記号 ($) は、必ずソースコードの 7 文字目に記述してください。ただし、SOURCEFORMAT(FREE) 指令を設定した場合には、この制約は適用されません。コンパイラ指令の設定の詳細については、Net Express の [**ヘルプ**] メニューで [**ヘルプトピック**] をクリックし、オンラインヘルプの「**目次**」タブで「**リファンレス**」、「**コンパイラ指令**」を順にクリックして、対応するページに記載されている情報を参照してください。

   ISAPI プログラムと NSAPI プログラムの指令

   | 指令 | 説明 |
   |---|---|
   | WEBSERVER(ISAPI) | このプログラムが ISAPI アプリケーションであることをコンパイラに通知します。<br><br>この指令は、アプリケーションを起動するために実行されるアプリケーションのメイン **.dll** のみに適用します。この指令は、メインの **.dll** から呼び出されるサブモジュールには適用しないでください。適用するとランタイムエラーが発生します。 |
   | WEBSERVER(NSAPI, *entry_point_name*) | このプログラムが NSAPI アプリケーションであることをコンパイラに通知します。*entry_point_name* はエントリポイント名で、任意の文字列を指定できます。*entry_point_name* で指定される名前の非表示エントリ |

| | |
|---|---|
| | ポイントがコンパイラによって生成され、NSAPI Web サーバによるプログラム起動が可能になります。*entry_point_name* にはハイフン (-) は使用しないでください。*entry_point_name* には、プログラムの PROGRAM-ID 見出しと同じ名前は指定できません。<br><br>この指令は、アプリケーションを起動するために実行されるアプリケーションのメイン **.dll** のみに適用します。この指令は、メインの **.dll** から呼び出されるサブモジュールには適用しないでください。適用するとランタイムエラーが発生します。 |
| CASE | 外部シンボル (呼び出されたプログラムの名前を含む) の大文字変換を禁止します。この指定を省略すると、エンドユーザがプログラムを実行しようとしたときに、見つからない旨を知らせるメッセージが Web ブラウザに表示されます。 |
| REENTRANT(2) | このプログラムの複数のコピーを確実かつ安全に実行できるようにします。 |
| SQL(THREAD=ISOLATE) | Open ESQL アプリケーション (インターネットアプリケーションウィザードで作成したデータアクセスアプリケーションを含む) のみに必要な指令。スレッドリソースとトランザクションのスレッド間共有を禁止します。 |

この指令を指定すれば、Open ESQL アプリケーション (インターネットアプリケーションウィザードで生成したデータアクセスアプリケーションなど) のパフォーマンスが向上する場合があります。

2. プロジェクトのビルド設定を変更する手順は次のとおりです。
   a. Net Express の「プロジェクト」ウィンドウの左ペインで ISAPI 用または NSAPI 用の **.dll** ファイルを右クリックし、コンテキストメニューの [**ビルド設定**] をクリックします。
   b. 「**リンク**」タブをクリックします。
   c. [**リンクウィザード**] をクリックします。このウィザードを操作してサーバ側プログラムに適した設定を選択します (このプログラムはマルチスレッドです)。この操作を通じて、動的に結合される共有マルチスレッドランタイムシステムを使用するようにプログラムが設定されます。

以上の手順の終了後、プロジェクトをリビルドして ISAPI または NSAPI の **.dll** プログラムを作成します。

---

**注記**: ISAPI プログラムをビルドした場合は、ISAPI を起動するすべてのフォームの Action プロパティを ***program*.exe** から ***program*.dll** に変更する必要があります (『アプリケーションの

URL の変更』の項を参照してください)。Action プロパティは、NSAPI プログラムでも変更する必要があります。この場合は、新しい MIME タイプを指定します。NSAPI プログラムに必要なその他の変更については、『NSAPI アプリケーションを実行する前に』の項で説明します。

---

## 12.3.10 NSAPI アプリケーションを実行する前に

この項では、NSAPI サーバ側プログラムをビルドした後、ディプロイやデバッグの前に実施すべき追加手順を示します。

- accnsapi モジュールの設定
- NSAPI サーバ設定ファイルへの変更
- フォームの Action プロパティの変更

これらの手順を実施した後、Web サーバが動作しているマシンにファイルをディプロイし、サーバを再起動して NSAPI プログラムをロードします。

### 12.3.10.1 accnsapi モジュールの設定

サーバ側プログラムが NSAPI Web サーバとの間でデータをやり取りするには、**accnsapi.dll** モジュールが必要です。Netscape サーバで使用されるエントリポイント名はバージョンごとに異なるため、ビルドする accnsapi.dll モジュールのバージョンは、使用している Web サーバのバージョンに一致させる必要があります。

accnsapi をビルドして設定する手順は次のとおりです。

1. accnsapi モジュールは、一連のエントリポイント名を通じて Netscape サーバと通信します。これらのエントリポイント名は、Netscape サーバに付属している **.lib** ファイル内で定義されています。
   - **libhttpd.lib** ― Netscape FastTrack 用
   - **libhttpd.lib** ― Netscape Enterprise Server 用
   - **httpd.lib** ― Netscape Commerce 用

   これらのファイルは、Netscape サーバソフトウェアを含むディレクトリのディレクトリ (**server\nsapi\examples**) 内にあります。適切なファイルを **Net Express\base\lib** ディレクトリにコピーします (このディレクトリにコピーすれば、ファイルのリビルド時にリンカによって確実に検出されます)。

2. Net Express には、さまざまな Netscape サーバ用の、複数のバージョンの **accnsapi.obj** が付属しています。
   - FastTrack

     デフォルトバージョンの **accnsapi.obj** は、FastTrack 用に設定されているため、変更する必要はありません。

- Commerce Server

  **Net Express\base\lib** ディレクトリ内で、デフォルトバージョンの **accnsapi.obj** のファイル名を変更し (上書き防止)、**accnscs.obj** のファイル名を **accnsapi.obj** に変更します。

- Enterprise Server

  **Net Express\base\lib** ディレクトリ内で、デフォルトバージョンの **accnsapi.obj** のファイル名を変更し (上書き防止)、**accenter.obj** のファイル名を **accnsapi.obj** に変更します。

3. Net Express のコマンドプロンプトを開きます (Windows の [**スタート**] メニューで [**プログラム** > **Micro Focus Net Express** > **Net Express コマンドプロンプト**] の順に選択します)。
4. **Net Express\base\lib** ディレクトリに移動し、次のコマンドを実行します。

   ```
   cbllink -d -j accnsapi.obj netscape.lib
   ```

   ***netscape*.lib** には **libhttpd.lib** または **httpd.lib** を指定します (上記のステップ 1 を参照してください)。

5. **accnsapi.dll** を **Net Express\base\lib** から **Net Express\base\bin** にコピーします。

### 12.3.10.2 NSAPI サーバ設定ファイルへの変更

この項では、Netscape 社の Fasttrack サーバを対象として、サーバ設定ファイルの変更方法を説明します。その他のサーバ製品を使用している場合は、その製品に付属しているマニュアルを参照してください。

次の設定が行われるように変更する必要があります。

- 物理的な実行可能ファイルとエントリポイントを、サーバの初期化時にロードされるモジュールとして定義する。
- NSAPI プログラムのエントリポイントをサービスとして定義し、新しい MIME タイプと関連付ける。
- 新しい MIME タイプを特定の拡張子と関連付ける。

変更すべき設定ファイルは次の 2 つです。

- **obj.conf**
- **mime.types**

**obj.conf** ファイルに次の手順で変更を加えます。

1. 次の行を追加して、サーバの起動時にサーバ側プログラムがロードされるようにします。
2. `Init fn="load-modules" shlib="`*executable_file*`.dll" funcs="`*entry-point-name*`"`

   各パラメータの意味は次のとおりです。

   | | |
   |---|---|
   | *executable_file* | サーバ側プログラムの物理パスとファイル名<br><br>**注記**: Windows プラットフォームにディプロイする場合でも、パスの区切り文字には円記号 (\) ではなく、通常のスラッシュ (/) を使用してください。 |
   | *entry-point-name* | プログラムのコンパイル時に WEBSERVER(NSAPI,*entry-point-name*) 指令で指定したエントリポイント名。『共有ランタイムシステムを使用する ISAPI/NSAPI プログラムのビルド』の項を参照してください。 |

3. <OBJECT name="default"> タグと </OBJECT> タグの間に次の行を追加し、プログラムのエントリポイントを新しい MIME タイプに関連付けます。
4. `Service fn="`*entry-point-name*`" method="(GET|POST)" type="magnus-internal/`*new-type*`"`

   各パラメータの意味は次のとおりです。

   | | |
   |---|---|
   | *new-type* | プログラムに関連付ける新しい MIME タイプの名前 |
   | *entry-point-name* | ステップ 1 で指定したエントリポイント名 |

**mime.types** ファイルに次の手順で変更を加えます。

1. 次の行を追加し、上記の手順で定義した新しい MIME タイプを拡張子に関連付けます。

   `type=magnus-internal/`*new-type* `exts=`*extension*

   各パラメータの意味は次のとおりです。

   | | |
   |---|---|
   | *new-type* | 上記の手順でプログラムと関連付けた MIME タイプ名 |
   | *extension* | プログラムのファイル拡張子 (拡張子の長さは 3 文字に制限されませ |

ん。)

#### 12.3.10.3 Action プロパティの変更

NSAPI プログラムは、前項で定義した新しい MIME タイプが Web ブラウザから要求されるたびに起動します。NSAPI プログラムを呼び出す各フォームで、新しい MIME タイプを要求するように Action プロパティを変更する必要があります。

Action プロパティを変更する手順は、次のとおりです。

1. Form Designer にフォームファイルをロードします。
2. ページ内のフォームごとに Action プロパティをクリックし、次のように変更します。

   *program*.*extension*

   *program* にはプログラムのエントリポイント、*extension* には前項で定義した新しい MIME タイプの拡張子を指定します。

3. ページ内の各フォームに変更を加えた後、**.htm** ファイルを保存し、新しいバージョンとしてディプロイします。

### 12.3.11 ISAPI に準拠した Object COBOL アプリケーションのディプロイ

Object COBOL ではオブジェクト指向 (OO) の ISAPI アプリケーションを作成できますが、作成したアプリケーションをディプロイするときに、いくつかの追加手順を実行する必要があります。なお、オブジェクト指向の ISAPI アプリケーションでは、COM コンポーネントを使用することが可能です。

Object COBOL で作成した ISAPI アプリケーションを実行するには、その前に Object COBOL のデフォルトの例外処理機構を無効にする必要があります。デフォルトの例外ハンドラは、OO 例外が発生すると、エラーメッセージを表示してプロセスを終了します。ISAPI アプリケーションは Web サーバと同じプロセスで動作するため、このような対応は望ましくありません。デフォルトのハンドラを無効にする手順は次のとおりです。

1. 新しい ASCII ファイルを **cobopt.cfg** という名前で作成します。
2. そのファイル内に次の行を記述します。

   ```
   set signal_regime(0)=1
   ```

   この行を記述した後、ファイルを保存します。

3. Application Server のランタイムシステムファイルディレクトリ (デフォルトでは、**program files\micro focus\appserver**) に、このファイルを移動します。

#### 12.3.11.1 COM と ISAPI

COM の詳細については、本マニュアル (『*分散コンピューティング*』) の第 4 部を参照してください。

ISAPI の **.dll** は Windows NT のサービスとして動作し、特定ユーザアカウントの設定やアクセス権は適用されません。これは、ISAPI の **.dll** 内で COM を使用するときに重要な意味を持ちます。

プロセス内 COM サーバを使用する場合は、そのサーバの **.dll** ファイルがロード可能であれば、アプリケーションからアクセスできます。

COM のローカルサーバを使用する場合には、サーバの実行可能ファイルにアクセスし、起動するために必要なアクセス権が与えられていることを確認する必要があります。

そのための手順を次に示します。

1. COM サーバのレジストリエントリがあることを確認します。

   サードパーティ製の COM サーバは、インストール時にレジストリに自動登録される場合もありますが、製品マニュアルを参照してユーザ自身で登録すべき可能性もあります。

   Object COBOL で独自に作成した COM サーバは、ユーザ自身で登録する必要があります。 Net Express の [**ツール**] メニューで [**COM レジストリファイルジェネレータ**] をクリックし、「OCREG」ダイアログボックスの [**ヘルプ**] ボタンをクリックして、表示される説明に従います。

2. **regedit.exe** (Windows 95) または **regedt32.exe** (Windows NT) を実行し、サーバの APPID 設定を追加します。
   a. HKEY_CLASSES_ROOT\CLSID に達するまでレジストリツリーを順次展開します。
   b. 左側のペインで CLSID を右クリックし、コンテキストメニューで [**新規 > キー**] を順に選択します。新規キーとしてサーバの UUID を入力します。

      UUID は、COM サーバのレジストリエントリの作成時に生成される値です。ステップ 1 で作成した **.reg** ファイルをテキストエディタで開き、表示される 16 進 long 値をすべてコピーします (中かっこ "{}" も含みます)。続いて、キーの名前を変更し、クリップボードの値を貼り付けます。

   c. 新しいキーを右クリックし、コンテキストメニューで [**新規** > **文字列**] の順にクリックします。右側のペインの「名前」列で、新しい名前として APPID と入力します。
   d. APPID エントリをダブルクリックし、値として UUID をもう一度貼り付けます。

続いて、サーバを実行するためのアカウントアクセス権を設定します。

1. **dcomcnfg.exe** を実行し、サーバを実行するアカウントの ID を設定します。

a. リストからサーバアプリケーションを選択し、プロパティを表示します。
b. 「**ID**」タブを選択します。
c. 「**ユーザ**」を選択します。
d. このアプリケーションを実行するアカウントのユーザ名とパスワードを入力します。

   このアカウントによって、アプリケーション実行中に適用されるセキュリティ権限が決定されます。

2. **dcomcnfg.exe** を使用して、上記のステップで選択したユーザアカウントに、サーバを起動するアクセス権が与えられていることを確認します。
   a. リストからアプリケーションを選択し、そのプロパティを表示します。
   b. 「**セキュリティ**」タブを選択し、ステップ 1 で選択したユーザアカウント、またはそのユーザアカウントが属するいずれかのグループが、サーバを起動する権限を持っていることを確認します。

      「**デフォルト**」オプションボタンが選択されている場合には「アプリケーションのプロパティ」ウィンドウを閉じ、メインウィンドウの「**デフォルトのセキュリティ**」タブをクリックして、[**デフォルトを編集**] ボタンをクリックしてください。

      選択したユーザがデフォルト設定に含まれていない場合には、そのユーザを追加します。アプリケーションの「セキュリティ」タブに戻り、「**独自の起動アクセス権を使用**」オプションボタンをクリックし、続いて [**編集**] ボタンをクリックしてリストにユーザを追加します。

3. サーバを含むフォルダに、サーバを実行するユーザのアクセス権が含まれていることを確認します。
   a. Windows のエクスプローラで、そのフォルダを選択します。
   b. フォルダの「プロパティ」ダイアログを表示します。
   c. 「**セキュリティ**」タブを選択し、[**Permissions**] をクリックします。

アクセス権を正しく設定すると、ISAPI の **.dll** ファイルから COM を介してローカルサーバに接続できるようになります。

### 12.3.12 XDB をインクルードするインターネットアプリケーションのリンク

作成したインターネットアプリケーションで SQL オプションサーバにアクセスするために、OpenESQL プリコンパイラのかわりに XDB プリコンパイラを使用する場合には、XDB データベースアクセス用の **.lib** ファイルをインクルードするように、ビルド設定を変更する必要があります。

手順は次のとおりです。

1. NetExpress プロジェクトを開き、[**ビルドタイプ**] を [一般リリースビルド] に設定します。
2. 対象の **.exe** ファイル (または **.dll** ファイル) を右クリックします。

3. シンボルが変換できないことを示す _XDBINTRF エラーが表示された場合は、次の手順を試みてください。
    1. [ビルド設定 ...] を選択して「リンク」タブをクリックします。
    2. [カテゴリ] を [高度な設定] に設定します。
    3. 「これらの LIB とリンクする」テキストボックスに次のファイル名を入力します。

        ```
        xdbintrf.lkib
        ```

4. オブジェクトのリンクを再試行します。
5. エラーが解消されないようであれば、SQL オプションのディレクトリが環境変数 LIB に登録されているかどうかをチェックしてください。[**プロジェクト**] メニューで [**プロパティ > IDE > インポート**] の順にクリックし、表示されるリストボックスをスクロールして、LIB 環境変数の値を確認します。

### 12.3.13 XDB アプリケーションの配布

作成したアプリケーションをを、SQL オプションアクセスロジックを含む **.exe** ファイル (または **.dll** ファイル) として他のマシンに配布する場合には、次の **.dll** ファイルの配布も必要になることがあります。

- **xdbintrf.dll**
- **xdbcrun.dll**

## 12.4 アプリケーションのディプロイ

この項には、アプリケーションを他のマシンの Web サーバでディプロイするときに必要になる情報を記載しています。

『ディプロイとデバッグの手引き』の項に従って Web サーバの設定とアプリケーション変更を完了すれば、他の Web サーバマシンにアプリケーションをディプロイする準備が整います。

次の説明に従って、プロジェクト内のファイルをコピーします。

- 入力フォームの **.htm** ファイルを、Web 共有リソース ***form-share*** にマッピングされたフォルダにコピーする。

    フォームが常に、いずれか 1 つのサーバ側プログラムから EHTML で出力される場合には、入力フォームがプログラムの実行コードに埋め込まれているため、Web サーバにコピーする必要はありません。インターネットアプリケーションウィザードによって生成されたすべてのコードは、フォームを EHTML で出力します。

- 入力および出力ページ用の **.js** ファイルを、Web 共有リソース ***form-share*** にマッピングされたフォルダにコピーする。

これらのファイルには、cookie の設定や取り込み、クライアント側検証など、JavaScript で記述された各種の関数が含まれています。

- アプリケーションに画像ファイルが含まれている場合、Web 共有リソース ***image-share*** にマッピングされたフォルダに画像ファイルをコピーする。
- Web 共有リソース ***bin-share*** にマッピングされたフォルダに、アプリケーションの実行可能ファイルをコピーする。

Web 共有リソースは、『ディプロイ用の Web サーバ設定』の項で設定しています。

次の 2 つの項では、特定の種類のアプリケーションだけに該当する追加情報を説明します。

- ISAPI/NSAPI アプリケーション内の依存関係の識別

  ISAPI/NSAPI アプリケーションをディプロイする場合は、この項に目を通してください。

- ODBC データソースの使用方法

  ODBC データソースを使用する SQL アプリケーションをディプロイする場合には、この項の情報に目を通してください (該当するアプリケーションには、インターネットアプリケーションウィザードで生成されるデータアクセスアプリケーションも含まれます)。

## 12.4.1 ISAPI/NSAPI アプリケーション内の依存関係の検出

この項では、次のいずれか 1 つ以上の条件に該当する ISAPI/NSAPI アプリケーションをディプロイする際に留意すべき情報を紹介しています。

- アプリケーションが 1 つのメイン **.dll** ファイルから構成され、このファイルから他の **.dll** ファイル内の 1 つ以上の副プログラムが呼び出される。

  『ISAPI/NSAPI アプリケーションでの副プログラムの検出』の項を参照してください。

- アプリケーションが拡張 DISPLAY 構文を通じて HTML ページを出力する (拡張 DISPLAY 構文については、『サーバ側プログラミング』の章を参照)。

  『ISAPI/NSAPI アプリケーションでの HTML ファイルの検出』の項を参照してください。

ISAPI や NSAPI を使用するプログラムは CGI プログラムとは異なり、メインプログラムと同じディレクトリに配置するだけでは、依存するファイルをいっさい見つけることができません。ISAPI/NSAPI アプリケーションは Web サーバプロセス内のスレッドとして動作するため、その作業ディレクトリはメイン **.dll** プログラムが位置するディレクトリではなく、常に Web サーバの作業ディレクトリになります。

### 12.4.1.1 ISAPI/NSAPI アプリケーションでの副プログラムの検出

ISAPI/NSAPI アプリケーションに、自身が呼び出す .dll ファイルを確実に検出させるには、そのアプリケーションをディプロイする各マシンで次の手順を実行します。

1. すべての .dll ファイルを 1 つのディレクトリにコピーします。

   ISAPI や NSAPI のメイン .dll ファイル用ディレクトリを使用できます。

2. マシンの Net Express または Application Server のパスに、次の手順でコピー先ディレクトリのパスを追加します。
   a. コマンドプロンプトでレジストリエディタを起動します (Windows 95 では regedit、Windows NT では regedt32)。
   b. 次のキーを見つけます。

      HKEY_LOCAL_MACHINE/SOFTWARE/Micro Focus/NetExpress/*version-number*/COBOL/*version-number*/Environment

   c. .dll をコピーしたディレクトリのパスを、PATH の値の末尾に追加します。

#### 12.4.1.2 ISAPI/NSAPI アプリケーションでの HTML ファイルの検出

ISAPI/NSAPI アプリケーションに、拡張 DISPLAY 構文を通じて出力する HTML ファイルを確実に検出させるには、アプリケーションをディプロイする各マシンで次の手順を実行します。

1. すべての HTML ファイルを 1 つのディレクトリにコピーします。
2. マシンの Net Express または Application Server の COBDATA に、次の手順でコピー先ディレクトリのパスを登録します。
   a. コマンドプロンプトでレジストリエディタを起動します (Windows 95 では regedit、Windows NT では regedt32)。
   b. 次のキーを見つけます。

      HKEY_LOCAL_MACHINE/SOFTWARE/Micro Focus/NetExpress/*version-number*/COBOL/*version-number*/Environment

   c. COBDATA が存在しない場合には、新規に作成します。
   d. 値を次のように設定します。

      ;;%COBDIR%;*html-dir*

      *html-dir* に、HTML ファイルをコピーしたディレクトリのパスを指定します。先頭の 2 つのセミコロン (;) は必須です。省略しないでください。

### 12.4.2 ODBC データソース

アプリケーションで ODBC データソースを使用する場合には、そのデータソースがサーバ側プログラムから認識できる状態にする必要があります。サーバ側プログラムは通常、起動元

の Web サーバのアクセス権を継承します。次の 2 点に留意し、Web サーバが ODBC データソースに確実にアクセスできようにしてください。

- ODBC データソースをユーザデータソースではなく、システムデータソースとして設定する。

  Web サーバはユーザデータソースにはアクセスできません。

- ODBC データソースにユーザ DSN ではなく、システム DSN を割り当てる。

  ユーザ DSN も使用できますが、適切なアクセス権限の割り当てが困難になる可能性があります。

## 12.5 UNIX へのアプリケーションのパブリッシュ

『UNIX サーバへのディプロイ手順』の項に示した手順に従って設定とアプリケーションの変更を行うと、UNIX サーバにアプリケーションを発行できるようになります。

1. Net Express でアプリケーションのディプロイ用コピーを開きます。
2. UNIX サーバで、アプリケーションのすべてのファイルを格納するディレクトリを作成します。
3. [**UNIX**] メニューで [**設定**] をクリックし、アプリケーションのサーバオプションを設定します。

   詳細については、『*UNIX オプションユーザガイド*』を参照してください。

4. [**UNIX**] メニューで [**パブリッシュ**] をクリックして、サーバにプロジェクトファイルを送信し、サーバで実行可能プログラムをビルドします。
5. アプリケーションの CGI プログラムを実行するシェルスクリプトを作成します。

   シェルスクリプトは、COBOL プログラムを実行する前に COBOL 環境を設定するために必要です。

   a. 次のようなスクリプトを作成します。
   b.
   ```
   #!/bin/sh
   ./mfenv.sh cginame
   ```

   c. スクリプトを ***cginame*.sh** という名前で保存します。
6. 次の説明に従って、発行先ディレクトリ内のファイルをコピーします。
   - 入力フォームの **.htm** ファイルを、Web 共有リソース ***form-share*** にマッピングされたディレクトリにコピーする。
   - DISPLAY 動詞によって CGI プログラムから出力されるフォームの **.htm** ファイルを、サーバ側プログラムと同じディレクトリにコピーする。

サーバ側プログラムから常に EHTML で出力されるフォームをコピーする必要はありません。EHTML で出力されるフォームは、プログラムの実行コードに埋め込まれています。インターネットアプリケーションウィザードによって生成されたすべてのコードは、フォームを EHTML で出力します。

- アプリケーションに画像ファイルが含まれている場合、Web 共有リソース ***image-share*** にマッピングされたフォルダに画像ファイルをコピーする。
- Web 共有リソース ***bin-share*** にマッピングされたフォルダに、アプリケーションの実行可能ファイルをコピーする。ステップ 5 で作成したシェルスクリプトと **mfenv.sh**、および CGI 実行可能ファイルも必ずコピーしてください。

これらの Web 共有リソースは、『ディプロイ用の Web サーバ設定』の項で設定しています。

# 12.6 アプリケーションのデバッグ

この項では、開発マシンで Solo 以外の Web サーバを使用してアプリケーションをデバッグするときに留意すべき情報を示しています。

『*ディプロイとデバッグの手引き*』の説明に従って Web サーバの設定とアプリケーションの変更を完了すると、アプリケーションをデバッグする準備が整います。CGI プログラムと ISAPI/NSAPI プログラムのデバッグ手順はやや異なるため、2 つの項に分けて説明します。

## 12.6.1 CGI プログラムのアニメート

この項は、『*ディプロイとデバッグの手引き*』の項で説明されている Web サーバの設定とアプリケーションの変更が、すでに完了していることを前提としています。CGI プログラムは、Solo 以外の Web サーバで最も簡単にアニメートできるプログラムです。CGI プログラムをアニメートするには、Net Express の IDE でプロジェクトを多少変更する必要があります。この変更を加えれば、デバッグを開始できます。

プロジェクトをアニメートするための設定手順は、次のとおりです。

1. CGI の **.exe** ファイルを右クリックし、コンテキストメニューで [**ビルド設定**] をクリックします。「**リンク**」タブをクリックし、選択した **.exe** ファイルが GUI アプリケーションではなく、キャラクタ型アプリケーションとしてビルドされることを確認します。必要に応じて、設定を変更し、リビルドします。
2. Net Express の [**オプション**] メニューで [**プロジェクト**] をクリックし、「プロジェクト」ダイアログボックスを表示します。
3. 「**Web サーバとして SOLO を使う**」チェックボックスの選択を解除して [**OK**] をクリックします。

**注記**: この設定は、現在のプロジェクトと、以降にロードする全プロジェクトに適用されます。Solo を使用するプロジェクトでは、変更したオプションを元に戻してください。

---

4. Net Express の [**アニメート**] メニューで [**設定**] をクリックし、「アニメート設定」ダイアログボックスを表示します。
5. 「**アニメート可能なアタッチメントの待機**」チェックボックスを選択します。
6. アプリケーションを開始する URL を「**アニメーションの開始位置**」に入力します。URL は次のいずれかの形式で入力します。

   http://*machinename.domain*/COBOL/*inputform*.htm
   http://*machinename.domain*/CGI-BIN/*program*.exe

   各パラメータの意味は次のとおりです。

| パラメータ | 説明 |
|---|---|
| *machinename.domain* | イントラネットまたはインターネット上でのマシンの位置 (**www.microfocus.com**、**dev1.slithy.com** など) |
| *inputform* | アプリケーションを起動するフォームを含むファイルの名前 |
| *program* | アプリケーションを起動するプログラムの名前 |

プログラムは次の操作でアニメートします。

- Net Express の [**アニメート**] メニューで [**アニメート開始**] または [**ステップ**] をクリックします。

  この操作によって、デフォルトの Web ブラウザが起動し、アプリケーションの開始ページまたは起動プログラムがロードされます。CGI プログラムの実行が開始されると、ただちに Net Express によるアニメートがスタートし、実行する最初の行がハイライト表示されます。この時点でプログラムのデバッグを開始できます。

## 12.6.2 ISAPI/NSAPI プログラムのアニメート

この項は、『ディプロイとデバッグの手引き』の項で説明されている Web サーバの設定とアプリケーションの変更 (プログラムを ISAPI/NSAPI プログラムとしてリビルドするための変更を含む) が、すでに完了していることを前提としています。ISAPI/NSAPI プログラムの実行やデバッグには Solo は使用できません。これらのプログラムをデバッグする場合は、ISAPI または NSAPI をサポートする Web サーバを使用する必要があります。ISAPI/NSAPI プログラムをデバッグするには、Net Express の IDE の設定を変更し、各 ISAPI/NSAPI プログラムの先頭に文を追加する必要があります。

プロジェクトをアニメートするための設定手順は、次のとおりです。

1. Net Express の [**オプション**] メニューで [**プロジェクト**] をクリックし、「プロジェクト」ダイアログボックスを表示します。
2. 「**Web サーバとして SOLO を使う**」チェックボックスの選択を解除して [**OK**] をクリックします。

---

**注記**: この設定は、現在のプロジェクトと、以降にロードする全プロジェクトに適用されます。Solo を使用するプロジェクトでは、変更したオプションを元に戻してください。

---

3. 各 ISAPI/NSAPI プログラムの先頭に次の文を追加します。

   ```
   call "CBL_DEBUGBREAK"
   ```

   この文が実行されると Net Express が起動し、デバッガが開始されます。

4. プロジェクトをリビルドします。

   アプリケーションの **.dll** ファイルは、Web サーバにロードされている間はロックされており、上書きできません。このファイルをビルドするディレクトリに Web 共有リソースがマッピングされている場合は、Net Express でプロジェクトをリビルドする前に、Web サーバを停止する必要があります。また、Web 共有リソースがその他のディレクトリにマッピングされている場合には、リビルドした **.dll** ファイルに上書きコピーする前に、Web サーバを停止する必要があります。

アプリケーションをアニメートする手順は次のとおりです。

1. NSAPI Web サーバを使用している場合は、Web サーバを再起動して NSAPI **.dll** プログラムをロードします。
2. Web ブラウザを起動し、アプリケーションを開始するフォームまたはプログラムの URL を入力します。

   ISAPI プログラムが CALL "CBL_DEBUGBREAK" 文を実行すると、デバッガを起動するかどうかを尋ねるメッセージボックスが表示されます。

3. [**はい**] をクリックします。

   「IDY ファイル エラー」ダイアログボックスが表示されます。デバッグに必要な **.idy** ファイルはデバッガで自動認識されないため、このダイアログボックスで場所を指定する必要があります。

4. [**参照**] をクリックし、デバッグ用の **.idy** ファイルを「開く」ダイアログボックスで選択します。このファイルは Net Express によって、アプリケーションの実行可能ファイルと同じディレクトリに作成されます。実行可能ファイルは通常、ソースファイル (プロジェク

ト) ディレクトリの **debug** (ビルドタイプによっては **release**) サブディレクトリに格納されています。

以上の手順を終了すると、プログラムのデバッグを開始できます。

---

## 注記

- Net Express による ISAPI/NSAPI プログラムの実行中にアニメートを停止すると、Web サーバも停止します。その場合には Web サーバを再起動してください。ISAPI/NSAPI プログラムの実行中またはアニメート中に STOP RUN 文に到達した場合も同様です。ISAPI/NSAPI プログラムでは STOP RUN 文は使用せず、常に EXIT PROGRAM 文を使用してください。
- ISAPI または NSAPI の **.dll** ファイルは、いったんロードされると、Web サーバが停止するまでメモリに維持されます。上記のステップ 2 で、デバッガの起動を確認するメッセージに対して [**いいえ**] をクリックすると、それ以降の実行時には Web サーバを再起動して **.dll** ファイルを再ロードしない限り、このメッセージは表示されなくなります。同じメッセージで [**はい**] をクリックした場合には、それ以降にプログラムを実行するたびに、Net Express 内で自動的にアニメートが実行されます。

---

---



---

# 第 38 章 : ディプロイとデバッグの例

この付録では、Net Express で開発したアプリケーションを Web サーバでデバッグし、ディプロイするための手順を、いくつかの作業例を通じて示します。

## 39.1 概要

この付録は、Net Express に付属しているサンプルアプリケーションの 1 つを、いくつかの種類の Web サーバでデバッグし、ディプロイする手順を示す作業例から構成されており、『CGI ベースのアプリケーションのディプロイ』の章の補足情報として利用できます。

使用する Web サーバを使った作業例が含まれていない場合も、全般的な手順は各種サーバでほぼ共通であるため、作業例を通じて得られた情報を活用できます。

### 39.1.1 Microsoft Internet Server への ISAPI アプリケーションのディプロイ

この項では、Net Express に付属している Form Designer のサンプルアプリケーションのディプロイ手順を、作業例を通じて示します。この例では、次の作業手順を知ることができます。

- Web 共有リソースを Net Express のデフォルト (**COBOL**、**CGI-BIN**) 以外の名前で作成する。
- 新しい Web 共有リソース名を使用するようにアプリケーションを変更する。
- 共有ランタイムシステムを使用する ISAPI アプリケーションとしてリビルドする。
- Web サーバにアプリケーションをコピーする。

アプリケーションのディプロイ時には、より簡単な方法を採ることも可能です。 たとえば、Net Express のデフォルトの Web 共有リソースを使用すれば、アプリケーションで使用する共有リソースの名前を変更する必要はありません。 また、ISAPI と共有ランタイムを使用しない場合や、インターネットアプリケーションウィザードで ISAPI アプリケーションを作成した場合には、アプリケーションをリビルドする必要はありません。

この例では、Net Express に付属しているサンプルアプリケーションの 1 つを、Microsoft Internet Server で ISAPI プログラムとしてディプロイする方法について説明します。

#### 39.1.1.1 Internet Server の Web 共有リソースの設定

この項では、アプリケーションを構成するファイルの種類に応じて、対応する Web 共有リソースを作成します。

1. Windows のエクスプローラで、**netxapps** という新しいフォルダを作成します。
2. **netxapps** 内に次の 4 つのサブフォルダを作成します。
    - nx-bin
    - nx-html
    - nx-images

3. Internet Service Manager を起動します。
4. 「Microsoft Internet Service Manager」ウィンドウで [WWW] プロセスをダブルクリックします。
5. 「WWW サービスのプロパティ」ダイアログボックスの「**ディレクトリ**」タブをクリックします。
6. バイナリファイルを格納する Web 共有リソースとして **nx-bin** を作成します。
    a. [**追加**] をクリックします。
    b. 「**ディレクトリ**」フィールドに ***x*:\netxapps\nx-bin** と入力します (***x*** には、ステップ 1 でフォルダを作成したドライブを指定します)。
    c. 「**エイリアス**」フィールドに **nx-bin** と入力します。
    d. 「アクセス」プロパティを「実行」のみに設定します (「**読み取り**」チェックボックスの選択を解除し、「**実行**」チェックボックスを選択します)。
    e. [**OK**] をクリックします。
7. さらに 2 つの共有リソースを**読み取り**専用で作成します。
    - エイリアス：nx-html
      ディレクトリ：***x*:\netxapps\nx-html**
    - エイリアス：nx-html
      ディレクトリ：***x*:\netxapps\nx-img**

#### 39.1.1.2 アプリケーションの実装可能バージョンの作成

アプリケーションソースを開発マシンのディプロイ用ディレクトリにコピーします。

1. 入力フォームの Action プロパティを次のように変更します。
    a. 「プロジェクト」ウィンドウの左側のペインで **bevord_h.htm** を右クリックし、コンテキストメニューの [**編集**] をクリックします。
    b. ページ内のフォームを選択し、Action プロパティをクリックして、値を次のように変更します。

        ```
        /nx-bin/bev_h.dll
        ```

2. **bevord_h.htm** を保存して Form Designer を終了します。

インターネットアプリケーションウィザードで ISAPI アプリケーションを作成した場合には、すでに ISAPI アプリケーションとしてビルドされているため、次の手順をスキップして『*アプリケーションのディプロイ*』に進んでください。 それ以外の場合は、次の手順に従って ISAPI アプリケーションをビルドする必要があります。

1. 実行可能ファイルを **.exe** から **.dll** に変更します。
    a. 「プロジェクト」ウィンドウの左側のペインで **bev_h.exe** を右クリックし、コンテキストメニューの [**ビルドタイプから削除**] をクリックします。
    b. 「ビルド構造から削除」ダイアログボックスで、「**ビルド構造の全体を削除します**」チェックボックスの選択を解除し、「**このターゲットをすべてのビルドタイプから削除します**」を選択して [**削除**] ボタンをクリックします。
    c. 「プロジェクト」ウィンドウの左側のペインで **bev_h.obj** を右クリックし、コンテキストメニューで [**選択されたファイルをパッケージ化 > 動的リンクライブラリ**] の順にクリックします。

d. 「新規に定義:動的リンクライブラリを定義」ダイアログボックスで [**作成**] をクリックします。

2. プログラムのビルド設定を次の手順で変更します。
   a. 「プロジェクト」ウィンドウの左側のペインで **bev_h.dll** を右クリックし、コンテキストメニューの [**ビルド設定**] をクリックします。
   b. 「**リンク**」タブをクリックし、「**共有**」と「**マルチスレッド**」のチェックボックスを選択します。「**動的**」チェックボックスも必ず選択してください。
   c. [**閉じる**] をクリックします。
3. プログラムを ISAPI 用にコンパイルするため、次の手順で SET 文を追加します。
   a. 「プロジェクト」ウィンドウの左側のペインで **bev_h.cbl** を右クリックし、コンテキストメニューの [**編集**] をクリックします。
   b. プログラムの先頭にある $SET PREPROCESS 指令の直後に次の文を追加します (プリプロセッサ指令は、プログラムの最初の SET 文に記述する必要があります)。

      ```
      $set webserver(isapi) case reentrant(2)
      ```

      先頭のドル記号 ($) が 7 文字目に位置するように記述してください。

   c. 変更を保存し、ファイルを閉じます。
4. プログラムをリビルドします。

#### 39.1.1.3 アプリケーションのディプロイ

この段階で Web サーバにアプリケーションをディプロイします。

1. **.dll** ファイルを ***x*:\netxapps\nx-bin** にコピーします。
2. **bevord_h.htm** と **bevsum_h.htm** を ***x*:\netxapps\nx-html** にコピーします。

アプリケーションを実行します。

- Web ブラウザを起動し、次の URL を入力します (*machine* はマシン名、*domain* はドメイン名)。

  ```
  http://machine.domain/nx-html/bevord_h.htm
  ```

# 39.2 各種 Web サーバでのデバッグ

以下の 3 つの項では、Net Express に付属しているサンプルプログラムを、複数の種類の Web サーバでアニメートする作業例を示します。具体的な作業例は次のとおりです。

- Microsoft Internet Server での CGI プログラムのアニメート
- Microsoft Internet Server での ISAPI プログラムのアニメート
- Netscape FastTrack での NSAPI プログラムのアニメート

これらのサンプルプログラムは、いずれもオペレーティングシステムとして Windows NT Server を使用して作成されています。

### 39.2.1 Microsoft Internet Server での CGI プログラムのアニメート

ここでは、Net Express に付属している飲み物注文 HTML アプリケーションを、Windows NT Server 上の Microsoft Internet Server を使用してアニメートする方法を示します。

次の手順で Internet Server を再設定します。

1. Microsoft Internet Service Manager を起動します。
2. [WWW] サービスを選択し、[**プロパティ > サービスのプロパティ**] を順にクリックして、「WWW サービスのプロパティ」ダイアログボックスを表示します。
3. 「**サービス**」タブで、「**匿名を認める**」チェックボックスの選択を解除します。

---

**注記**: この設定によって、他のマシンからこの Web サーバにアクセスするユーザに、ユーザ名とパスワードの提示が義務付けられます。アニメータを実行できる権限を Web サーバに与えるには、匿名ログオンを禁止する必要があります。

オペレーティングシステムと Web サーバの設定によっては、「**NT チャレンジ/レスポンス**」または「**基本 (テキストをクリア)**」チェックボックスの選択が必要になる場合もあります。Web ブラウザとして Microsoft Internet Explorer を使用する場合は「**NT チャレンジ/レスポンス**」、Netscape Navigator を使用する場合は「**基本**」をそれぞれ使用してください。

以上の手順を行った後、CGI を実行してもアニメートされない場合は、次の手順に従ってください。

---

- 「WWW サービスのプロパティ」ダイアログボックスの「ディレクトリ」タブをクリックします。
- 共有リソースとして **COBOL** と **CGI-BIN** がある場合は削除します (共有リソースをクリックし、[**削除**] ボタンをクリックします)。
- アプリケーションのソースディレクトリにマッピングした COBOL の共有リソースを追加します。
    a. [**追加**] ボタンをクリックし、「ディレクトリのプロパティ」ダイアログボックスを表示します。
    b. 「**ディレクトリ**」フィールドに、次のように入力します。

        ```
        x:\Program Files\Micro Focus\Net Express\base\demo\bevord_h
        ```

        x は Net Express をインストールしたドライブです。

    c. 「**エイリアス**」フィールドに、次のように入力します。

COBOL

d. 「アクセス」を「読み取り」に設定します。
e. [**OK**] をクリックします。

- アプリケーションのソースディレクトリにマッピングされた共有リソース CGI-BIN を追加します。
  a. [**追加**] ボタンをクリックし、「ディレクトリのプロパティ」ダイアログボックスを表示します。
  b. 「**ディレクトリ**」フィールドに、次のように入力します。

     x:\Program Files\Micro Focus\Net Express\base\demo\bevord_h\debug

     x は Net Express をインストールしたドライブです。このプロジェクトを初めてビルドする場合は、このディレクトリが存在しないため、変更の適用時にサーバエラーが発生します。エラーが発生した場合には、[OK] ボタンをクリックして変更を適用する前に、Windows のエクスプローラで上記の場所に **debug** フォルダを作成してください。

  c. 「**エイリアス**」フィールドに、次のように入力します。

     CGI-BIN

  d. 「アクセス」を「実行」に設定します。
  e. [**OK**] をクリックします。
- [**OK**] をクリックして、Web サーバに変更を適用します。

アプリケーションをリビルドし、アニメートします。

1. Net Express を起動し、サンプルプロジェクト **bevord_h.app** をロードします。
2. Net Express が Solo を使用しないように、次の手順で設定を変更します。
   a. [**オプション > プロジェクト**] の順にクリックします。
   b. [**Web サーバーとして SOLO を使う**」チェックボックスの選択を解除します。
   c. [**OK**] をクリックします。
3. アプリケーションを起動する URL を、Solo 用の URL から、使用する Web サーバに適した URL に変更します。
   a. [**プロジェクト > プロパティ**] をクリックします。
   b. 「**アニメート開始位置**」フィールドの値を次のように変更します。

      http://*servername*/COBOL/bevord_h.htm

      *servername* には、この Web サーバへのアクセスに使用するサーバ名を指定します。

   c. [**OK**] をクリックします。
4. [**プロジェクト > リビルド**] をクリックし、アプリケーションをビルドします。

5. [**アニメート > アニメート開始**] をクリックしてアプリケーションを実行します。
6. Web ブラウザで [**Brew it!**] ボタンをクリックし、CGI を実行します。この操作でアニメータが起動します。

### 39.2.2 Microsoft Internet Server での ISAPI プログラムのアニメート

ここでは、Net Express に付属している飲み物注文 HTML アプリケーションの ISAPI バージョンを、Windows NT Server 上の Microsoft Internet Server を使用してアニメートする手順を示します。この手順は、次の各段階に分けられます。

- Web サーバの設定
- ISAPI **.dll** としてのプログラムのリビルド
- プログラムのアニメート

Web サーバの設定手順は次のとおりです。

1. Microsoft Internet Service Manager を起動します。
2. [WWW] サービスを選択し、[**プロパティ > サービスのプロパティ**] を順にクリックして、「WWW サービスのプロパティ」ダイアログボックスを表示します。
3. 「**サービス**」タブで、「**匿名を認める**」チェックボックスの選択を解除します。

   この設定によって、他のマシンからこの Web サーバにアクセスするユーザに、ユーザ名とパスワードの提示が義務付けられます。この設定は必須であり、省略すると Web サーバは Net Express デバッガを起動できません。

   オペレーティングシステムと Web サーバの設定によっては、「NT チャレンジ/レスポンス」または「基本 (テキストをクリア)」チェックボックスの選択が必要になる場合もあります。Web ブラウザとして Microsoft Internet Explorer を使用する場合は「NT チャレンジ/レスポンス」,Netscape Navigator を使用する場合は「基本」をそれぞれ使用してください。

   以上の手順を行った後、CGI を実行してもアニメートされない場合は、次の手順を実行します。

4. 「WWW サービスのプロパティ」ダイアログボックスの「**ディレクトリ**」タブをクリックします。
5. 共有リソースとして **COBOL** と **CGI-BIN** がある場合は削除します (共有リソースをクリックし、[**削除**] ボタンをクリックします)。
6. Net Express のソースディレクトリを **COBOL** という名前の Web 共有リソースとして設定します。
   a. [**追加**] ボタンをクリックし、「ディレクトリのプロパティ」ダイアログボックスを表示します。
   b. 「**ディレクトリ**」フィールドに、次のように入力します。

      ```
      x:\Program Files\Micro Focus\Net Express\base\demo\bevord_h
      ```

      x は Net Express をインストールしたドライブです。

c. 「**エイリアス**」フィールドに、次のように入力します。

```
COBOL
```

d. 「アクセス」を「読み取り」に設定します。
e. [**適用**] をクリックします。

7. Net Express の実行可能プログラムのディレクトリを、**COBOL** という名前の Web 共有リソースとして設定します。
   a. このプロジェクトを初めてビルドする場合には、実行可能プログラムのディレクトリは存在しません。Windows のエクスプローラで **x:\Program Files\Micro Focus\Net Express\base\demo\bevord_h** フォルダを開きます (x は Net Express をインストールしたドライブ)。このフォルダ内に **debug** という名前のサブフォルダが存在しなければ、Windows のエクスプローラで **debug** フォルダを新しく作成してください。
   b. [**追加**] ボタンをクリックし、「ディレクトリのプロパティ」ダイアログボックスを表示します。
   c. 「**ディレクトリ**」フィールドに、次のように入力します。

```
x:\Program Files\Micro
Focus\Net Express\base\demo\bevord_h\debug
```

   d. 「**エイリアス**」フィールドに、次のように入力します。

```
COBOL
```

   e. 「アクセス」を「読み取り」に設定します。
   f. [**適用**] をクリックします。

プログラムを ISAPI 用の **.dll** としてリビルドする手順は次のとおりです。

1. **bevord_h.app** プロジェクトを Net Express の IDE にロードします。
2. **bev_h.exe** のかわりに **bev_h.dll** を生成するように、プロジェクトのビルド情報を変更します。
   a. 「プロジェクト」ウィンドウの左側のペインで **bev_h.exe** を右クリックし、コンテキストメニューの [**ビルドタイプから削除**] をクリックします。
   b. 「ビルド構造から削除」ダイアログボックスで、「**ビルド構造の全体を削除します**」チェックボックスの選択を解除し、「**このターゲットをすべてのビルドタイプから削除します**」を選択して [**削除**] ボタンをクリックします。
   c. 「プロジェクト」ウィンドウの左側のペインで **bev_h.obj** を右クリックし、コンテキストメニューで [**選択されたファイルをパッケージ化** > **動的リンクライブラリ**] の順に選択します。
   d. 「新規に定義:動的リンクライブラリを定義」ダイアログボックスで [**作成**] をクリックして **bev_h.dll** をプロジェクトリストに追加します。
3. マルチスレッドの共有ランタイムシステムを使用するように、**.dll** ファイルのビルド設定を変更します。

a. Net Express の「プロジェクト」ウィンドウの左ペインで **bev_h.dll** ファイルを右クリックし、コンテキストメニューから [**ビルド設定**] を選択します。
b. 「**リンク**」タブをクリックします。
c. 「**共有**」および「**マルチスレッド**」チェックボックスを選択します。「**動的**」チェックボックスも必ず選択してください。
d. [**閉じる**] をクリックします。

4. このプログラムを ISAPI プログラムとしてビルドするコンパイラ指令を追加します。**bev_h.cbl** の先頭にある $SET 文の直後に次の文を追加します。

```
$set webserver(isapi) case reentrant(2)
```

先頭のドル記号 ($) が 7 文字目に位置するように記述してください。

5. プログラムの実行時にデバッガを起動する文を、次の手順で追加します。
a. 「プロジェクト」ウィンドウで **bev_h.cbl** をダブルクリックし、テキスト編集ペインにロードします。
b. プログラムの手続き部に移動します。
c. 次の文を手続き部の先頭に追加します。

```
call "CBL_DEBUGBREAK"
```

6. [**プロジェクト > リビルド**] を順にクリックし、ISAPI プログラムをビルドします。
7. **bev_h.exe** のかわりに **bev_h.dll** を呼び出すように、入力フォームの Action プロパティを変更します。
a. **bevord_h.htm** をダブルクリックして Form Designer を起動し、ページをロードします。
b. フォームを選択して **Action** プロパティをクリックし、値を次のように変更します。

```
CGI-BIN/bev_h.dll
```

c. フォームを保存し、Form Designer を終了します。
8. Net Express IDE を終了します。

続いて、次の手順でプログラムをアニメートします。

1. Web ブラウザを起動します。
2. アプリケーションの最初のページの URL を入力します。

```
http://machinename/cobol/bevord_h.htm
```

*machinename* は、Web サーバが動作するマシン名です。

3. フォームで [**Brew It!**] ボタンをクリックします。

アプリケーションをデバッグするかどうかを尋ねるダイアログボックスが表示されます。

4. [**はい**] をクリックします。

   「IDY ファイルエラー」ダイアログボックスが表示されます。デバッグに必要な **.idy** ファイルはデバッガで自動認識されないため、このダイアログボックスで場所を指定する必要があります。

5. [**参照**] ボタンをクリックし、プログラムのデバッグに使用する **bev_h.idy** ファイルを「開く」ダイアログボックスで選択します。このファイルは Net Express によって、アプリケーションの実行可能ファイルと同じディレクトリに作成されます。

   以上の手順を終了すると、プログラムのデバッグを開始できます。

デバッガを停止すると Web サーバも停止し、再起動が必要になります。EXIT PROGRAM 文をステップ実行すると、Net Express IDE は開いたままで維持されますが、デバッガはプログラムを再実行するまで停止します。また、Web サーバの動作中は **.dll** ファイルが開いたままの状態になるため、リビルドするには Web サーバを停止する必要があります。

### 39.2.3 NSAPI プログラムのアニメート

ここでは、Net Express に付属している飲み物注文 HTML アプリケーションの NSAPI バージョンを、Windows NT 上の Netscape FastTrack を使用してアニメートする手順を示します。この手順は、次の各段階に分けられます。

- Web サーバの設定
- NSAPI **.dll** としてのプログラムのリビルド
- プログラムのアニメート

サーバは次の手順で設定します。

1. Windows の [**スタート**] メニューで、[**Netscape > Netscape サーバー管理**] を順に選択します。
2. Web サーバに COBOL 共有リソースを追加します。
   a. 最上部のフレームで [**コンテンツ管理**] をクリックします。
   b. 左側のフレームで [**追加ドキュメントディレクトリ**]をクリックします。

      この操作によって、「追加ドキュメントディレクトリ」フォームが表示されます。

   c. 「URL」フィールドに **COBOL** と入力します。
   d. プロジェクトのソースコードディレクトリの絶対パスを「**エイリアス**」フィールドに入力します。
   e. [**OK**] をクリックします。

      「変更の保存と適用」ページが表示されます。

   f. 「変更の保存と適用」ページで [**変更の保存と適用**] ボタンをクリックします。

**注記**: NSAPI アプリケーションの場合は、CGI-BIN 共有リソースを作成して NSAPI プログラムを検出させる必要はありません。NSAPI プログラムは Web サーバの起動時に、サーバ設定ファイル内で明示的に定義されているパスを使用してロードされます。

3. NSAPI プログラムをロードし、対応する新しい MIME タイプを定義するように、次の手順で Web サーバの設定ファイルを編集します。
   a. Netscape のサーバファイル **obj.conf** をテキストエディタで開きます。

      このファイルは **netscape\server\httpd-*name*\config** にあります (***name*** はサーバマシンの名前)。

   b. 次の行を追加して、サーバの起動時にサーバ側プログラムがロードされるようにします。
   c.
```
Init fn="load-modules" shlib="x:/Program Files/Micro
Focus/Net Express/base/demo/debug/bev_h.dll"
funcs="beverage"
```

      *x* は、Net Express をインストールしたドライブです。

   d. <OBJECT name="default"> タグと <OBJECT> タグの間に次の行を追加し、プログラムのエントリポイントを新しい MIME タイプ (bev) に関連付けます。

```
Service fn="beverage" method="(GET|POST)" type="magnus-
internal/bev"
```

4. **mime.types** (**obj.conf** と同じディレクトリ内にあります) を編集し、定義したばかりの新しい MIME タイプを拡張子と関連付けます。 次の行を **mime.types** に追加してください (magnus-internal/bev の拡張子を cobolcgi として定義します)。

```
type=magnus-internal/bev exts=cobolcgi
```

   この結果、Web ブラウザが **beverage.cobolcgi** を要求したときに、**bev_h.dll** が実行されるようになります。

5. Netscape サーバ管理ブラウザの最上部にあるフレームで [**適用**] ボタンをクリックします。

   「**変更の適用**」フィールドが表示されます。

6. [**構成ファイルの読み取り**] ボタンをクリックします。

プログラムを NSAPI 用にリビルドする手順は次のとおりです。

1. プロジェクト **bevord_h.app** を Net Express の IDE にロードします。

2. **bev_h.exe** のかわりに **bev_h.dll** を生成するように、プロジェクトのビルド情報を変更します。
    a. 「プロジェクト」ウィンドウの左側のペインで **bev_h.exe** を右クリックし、コンテキストメニューの [**ビルドタイプから削除**] をクリックします。
    b. 「ビルド構造から削除」ダイアログボックスで、「**ビルド構造の全体を削除します**」チェックボックスの選択を解除し、「**このターゲットをすべてのビルドタイプから削除します**」を選択して [**削除**] ボタンをクリックします。
    c. 「プロジェクト」ウィンドウの左側のペインで **bev_h.obj** を右クリックし、コンテキストメニューで [**選択されたファイルをパッケージ化 > 動的リンクライブラリ**] の順にクリックします。
    d. 「新規に定義:動的リンクライブラリを定義」ダイアログボックスで [**作成**] をクリックして **bev_h.dll** をプロジェクトリストに追加します。
3. マルチスレッドの共有ランタイムシステムを使用し、NSAPI に必要なモジュールやライブラリにリンクするように、プログラムの [ビルド設定] を変更します。
    a. Net Express の「プロジェクト」ウィンドウの左ペインで **bev_h.dll** ファイルを右クリックし、コンテキストメニューの [**ビルド設定**] をクリックします。
    b. 「**リンク**」タブをクリックします。
    c. 「**共有**」および「**マルチスレッド**」チェックボックスを選択します。「**動的**」チェックボックスも必ず選択してください。
    d. [**カテゴリ**] ドロップダウンメニューから「**高度な設定**」を選択します。
    e. 「**これらの OBJ とリンクする**」に **accnsapi.obj** と入力します。

       このファイルには 3 つのバージョンがあり、Netscape サーバの種類に応じて、適切なバージョンを使用する必要があります。

       - FastTrack

         **accnsapi.obj** のデフォルトバージョンは FastTrack 用であり、変更する必要はありません。

       - Commerce Server

         **Net Express\base\lib** ディレクトリ内で、デフォルトバージョンの **accnsapi.obj** のファイル名を変更し (上書き防止)、**accnscs.obj** のファイル名を **accnsapi.obj** に変更します。

       - Enterprise Server

         **Net Express\base\lib** ディレクトリ内で、デフォルトバージョンの **accnsapi.obj** のファイル名を変更し (上書き防止)、**accenter.obj** のファイル名を **accnsapi.obj** に変更します。

    f. 「**これらの LIB とリンクする**」フィールドに **libhttpd.lib** と入力します (Commerce Server を使用する場合は **httpd.lib** と入力してください)。
4. **libhttpd.lib** ファイルを、Net Express のコンパイラが検出できるディレクトリにコピーします。

このファイルは、Netscape サーバソフトウェアがインストールされているディレクトリのサブディレクトリ (**server\nsapi\examples**) にあります。このディレクトリ内の **libhttpd.lib** を、*x*:**\Program Files\Micro Focus\Net Express\base\bin** (*x*: は Net Express のインストール先ドライブ名) にコピーします。

5. このプログラムを NSAPI 用にビルドするコンパイラ指令を追加します。**bev_h.cbl** の先頭にある $SET 文の直後に次の文を追加します。

   ```
   $set webserver(nsapi,beverage) case reentrant(2)
   ```

   先頭のドル記号 ($) が 7 文字目に位置するように記述してください。

6. プログラムの実行時にデバッガを起動する文を、次の手順で追加します。
   a. 「プロジェクト」ウィンドウで **bev_h.cbl** をダブルクリックし、テキスト編集ペインにロードします。
   b. プログラムの手続き部に移動します。
   c. 次の文を手続き部の先頭に追加します。

      ```
      call "CBL_DEBUGBREAK"
      ```

7. [**プロジェクト > リビルド**] を順にクリックし、NSAPI プログラムをビルドします。
8. **bev_h.exe** のかわりに **bev_h.dll** を呼び出すように、入力フォームの Action プロパティを変更します (サーバの設定時に、新しい MIME の拡張子として **.dll** を定義しています)。
   a. **bevord_h.htm** をダブルクリックして Form Designer を起動し、ページをロードします。
   b. フォームを選択して Action プロパティをクリックし、値を次のように変更します。

      ```
      beverage.cobolcgi
      ```

   c. フォームを保存して Form Designer を終了します。
9. Net Express IDE を終了します。
10. Netscape サーバにプログラムをロードします (いったんサーバを停止して [Netscape サーバー管理] から再起動してください)。

続いて、次の手順でプログラムをアニメートします。

1. Web ブラウザを起動します。
2. アプリケーションの最初のページの URL を入力します。

   http://*machinename*/COBOL/bevord_h.htm

   *machinename* は、Web サーバが動作するマシン名です。

3. フォームで [**Brew It!**] ボタンをクリックします。

アプリケーションをデバッグするかどうかを尋ねるダイアログボックスが表示されます。

4. [**はい**] をクリックします。

   「IDY ファイルエラー」ダイアログボックスが表示されます。デバッグに必要な **.idy** ファイルはデバッガで自動認識されないため、このダイアログボックスで場所を指定する必要があります。

5. [**参照**] ボタンをクリックし、プログラムのデバッグに使用する **bev_h.idy** ファイルを「開く」ダイアログボックスで選択します。このファイルは Net Express によって、アプリケーションの実行可能ファイルと同じディレクトリに作成されます。

   以上の手順を終了すると、プログラムのデバッグを開始できます。

デバッガを停止すると Web サーバも停止し、再起動が必要になります。EXIT PROGRAM 文をステップ実行すると、Net Express IDE は開いたままの状態で維持されますが、デバッガはプログラムを再実行するまで停止します。また、Web サーバの動作中は **.dll** ファイルが開いたままの状態になるため、リビルドするには Web サーバを停止する必要があります。

---



---