Nextra
サーバ開発者ガイド

Version 6

1st Edition

目 次

第 1 章 概要

この章では、本書の利用方法、対象読者および Nextra の使用方法について説明します。

本書の利用方法

本書『サーバ開発者ガイド』を参考に、3 層分散システムの設計と構築を行ってください。本書では、分散アプリケーションの構築の手順を紹介します。まず、この章を読んで、対象読者に該当し、本書が必要に応じた適切なマニュアルであることを確認してください。

対象読者

本書は、3 層分散アプリケーションを扱う全ての開発者と管理者を対象にしています。本書を通じて、サーバの構築、アプリケーションの設定について基礎知識を得ることができます。

前提知識

読者には、クライアント／サーバコンピューティングについての基礎知識と、2 層構造アーキテクチャ固有の制限事項について前提知識があるものとします。本書を読む前に、『はじめにお読みください』をお読みください。

開発に必要な各言語（C, COBOL, SQL, Java）についての知識も必要です。

本書の位置付け

本書には、チュートリアルと簡単な言語の説明がありますので、オープンクライアント／サーバアプリケーションを構築して実行するために必要な中核となるスキルを得ることができます。本書を知識習得の出発点としてご利用ください。

短い期間でご理解いただくために、本書では基礎的なことに絞って説明しています。ソフトウェアの機能とオプションのほとんどについては、本書では解説していません。これらについては、『運用／設定ガイド』、および『リファレンス』で説明されています。アプリケーションを作成する場合は、この 2 冊を参照してください。

表記規則

文中の表記規則

本書で使用する規則を理解しておくと、ユーティリティの使用方法などを容易に理解できます。

形式	説明	例

`terminal`	OS やサードパーティのユーティリティ、ファイル名、または変数の定数値を示します。	`telnet`
		
		`cust.def`

`sub-text`	ユーザが指定する必要がある値を示します。	*`server`*`_c.pl` `-e` *`environment_file`*
`bold`	本文中では Nextra ユーティリティを示します。サンプル中では、強調される部分を示します。	**`broker`** **`RPCMake`**

`[brackets]`	|がない場合は、オプションテキストを示します。|がある場合は、いずれか 1 つを選択することを示します。	`[-d` *`def_file`*`]`
		
		`[NONE|ERROR|WARN|DEBUG]`

次の形式で区別されているパラグラフは、コード例です。

```
#include <stdio.h>

main( ) {
      int i ;
      printf("the number is %d\n",i);
}
```

本書で使用するシンボル

本書では、次のようなシンボルを使用しています。

警告メッセージ

このシンボルに続くメッセージに、特別な注意を払う必要があることを示しています。このメッセージには重要な情報が含まれており、この情報を正しく理解してから先に進んでください。

ヒントメッセージ

このシンボルに続く本文は、必須ではありませんが状況に応じて役立つ手順であることを示しています。

オプションメッセージ

このシンボルに続く本文はオプションであることを示しています。内容は、追加機能または代替手法の概要、ある概念を理解するために役立つプロセスステップの詳細などです。

デバッグ方法

このシンボルに続く本文は、プロジェクトの現在のステップをデバッグする手順が含まれていることを示しています。この方法はあくまで参考であり、別の有効なデバッグ方法の使用を妨げるものではありません。

Nextraの旧バージョンとの互換性

旧バージョン(Nextra5.2 以前)

Nextra5.2 以前に開発したアプリケーションは、Nextra6 で提供される RPC ライブラリを用いて再コンパイルしてください。

言語サポート

開発パッケージは、多種多様なプログラミング言語、DB、およびプラットフォームをサポートします。Nextra の本リリースでは、以下のプログラミング言語と DB がサポートされます。

クライアント言語/GUI:

- Java
- ANSI C
- C++
- C#.NET
- COBOL
- Visual Basic (.NET)
- PowerBuilder
- Delphi

サーバ言語:

- ANSI C
- C++
- COBOL
- Java

データベース:

- Microsoft SQL Server
- Oracle

- DB2
- HiRDB

クライアントプラットフォーム:

- Windows
- AIX
- HP-UX
- Solaris
- Linux

サーバプラットフォーム:

- Windows
- AIX
- HP-UX
- Solaris
- Linux

第 2 章　基礎知識

この章では、3 層分散アプリケーションの動作について、基本的な事柄から高度な知識までを説明します。

この章は、Nextra の動作について理解する必要のある開発者と管理者向けの内容になっています。既に Nextra のトレーニングを受けた方にとっては、この章は復習のためのクイックリファレンスとしてお使いいただけます。

RPC (Remote Procedure Call)

オープン分散環境では、異なったマシンで実行されているプログラムは、リモートプロシージャコール (RPC)を通じて互いに通信します。RPC では、1 つのプログラム(クライアント)が別のプログラム(サーバ)に処理の実行を要求します。サーバは、クライアントコード内のローカルプロシージャのように、クライアントからのパラメータを受信してプロシージャを実行し、パラメータと戻り値またはどちらか一方を返します。

Nextra では、通信プログラムを書くことに注意を払わなくてよいので、クライアント／サーバアプリケーションの構築は非常に簡単です。その代わりに、各リモート関数とパラメータのデータ型を定義する、IDL（Interface Definition Language）ファイルを記述します。IDL ファイルによって、クライアントとサーバのための通信プログラム（スタブ・スケルトン）が自動的に生成されます。スタブ（Stub）・スケルトン（Skeleton）は、RPC ランタイムライブラリに結びつけられて、転送のためのデータのパッケージ化(データマーシャリング)の処理を全て行い、転送を実行し、RPC を完全に透過的なものとします。内部の複雑な処理は全てスタブ・スケルトンが行うので、開発者はアプリケーションのコーディングに専念することができます。スタブ・スケルトンにより、リモート関数呼び出しは、ローカル関数呼び出しと同じように容易に記述できます。

ネーミング・サービス（Naming service）

ネーミング・サービスは、ネットワーク上でのサーバリソースの位置情報を提供します。「電話番号案内」を呼び出して、連絡したい電話番号を知るのと同じ方法で、クライアントプログラムは、ネーミング・サービスに照会して必要なインタフェースを提供するサーバの位置情報を検索することができます。

ネーミング・サービスを提供するのは、ブローカ（broker）であり、各アプリケーションで動作する全てのサーバの位置情報を保持する特殊なサーバ（Naming server）です。

サーバプログラムが起動されると、ネーミングサーバであるブローカ（broker）に、ホスト名とポート番号を登録します。ブローカはこの情報をメモリ上に保持し、クライアントからの問い合わせがあるたびに情報を渡します。

電話の場合で考えてみましょう。通常は、相手に直接電話をかけますが、電話番号が分からなければ、104 に電話をかけオペレータに問い合わせることができます。これと同じように、クライアントは通常、RPC を直接サーバに送りますが、サーバの位置情報がわからない場合は、ブローカの助けを借りるのです。

ブローカは、他のサーバとは異なり、スケルトンが必要ありません。これは、RPC ランタイムライブラリに内蔵された固定インタフェースを持っているためです。

RPC使用時の処理過程

図 2.1 に、RPC が発行されたときの処理の流れを示します。クライアントコード（開発者が記述）がリモート関数を呼び出すとき、クライアント・スタブには同名の関数が存在し、この関数は RPC ランタイムライブラリで定義された一連の関数を呼び出します。

まず、クライアント・スタブは、その関数を扱うサーバのネットワークアドレスについて内部テーブルを検索します。テーブルに適切なサーバ位置情報がない場合、クライアント・スタブはブローカに問い合わせます。

ブローカは、サーバアドレスをクライアント・スタブに送り、クライアント・スタブはサーバ位置情報の内部テーブルを更新します。この情報を元に、クライアントはサーバに接続して、呼び出される関数名と引数をサーバに送ります。クライアント・スタブは、サーバ・スケルトンが応答を返すまで待ちます。

関数名と引数を受け取ると、サーバ・スケルトンは指定されたサーバ関数(図 2.1 ではローカル関数)を、受け取った引数で呼び出し、結果をクライアント・スタブに返します。

クライアント・スタブはサーバとの接続を終了し、クライアントコードに対して RPC の結果を返します。

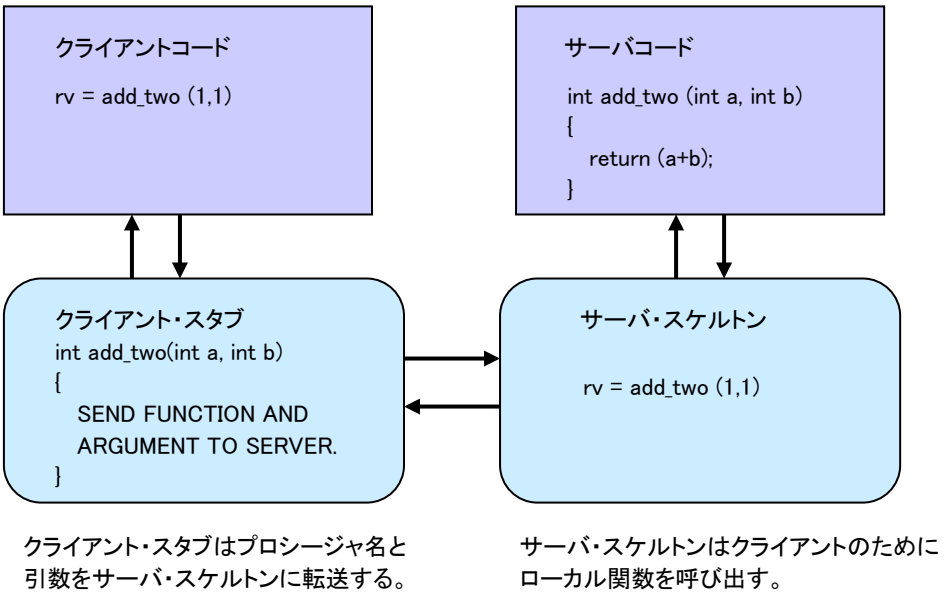


図 2.1 : リモート関数呼び出し

RPCを使うことの利点

RPC 技術によって、サーバコードを共用リソースとして使用できるようになり、たくさんの異なるクライアントが同じサーバにアクセスできます。クライアントとサーバは同じ言語で書かれている必要も、同じ OS やハードウェアプラットフォーム上で実行される必要もありません。

たとえば、Windows 上で Visual Basic クライアントプログラムを実行し、そのクライアントを IBM 互換 PC 上で実行されている PowerBuilder アプリケーションが使用しているサーバにアクセスさせることができます。サーバを C や COBOL で作成し、AIX、HP-UX、Solaris、Windows または Linux 上で実行できます。クライアントプログラムに必要なことは、RPC のリクエスト先を知っていることだけです。

3 層構造アーキテクチャ

アプリケーションプログラマとして Nextra を使用する場合、オープンなオブジェクト指向パラダイムをサポートするコードを作成することも仕事の 1 つになります。より良い開発者になれるだけではなく、今までにはなかった発想、解決策を生み出せるようになります。

3 層構造での開発により、以下のメリットを享受できます:

1. オープン
2. モジュール化
3. スケーラブル
4. 高パフォーマンス
5. 容易な修正
6. 統合的なアプリケーションの作成

3 層構造アプローチにより、理想的なオープン分散環境を手に入れることができます。

3 層構造とは

3 層構造アーキテクチャは、アプリケーションを論理的に 3 つの異なる層(tier)、つまり、プレゼンテーション層(Presentation tier)、ファンクショナリティ層(Functionality tier)、データ層(Data tier)に分割します。プレゼンテーション層は、コンピュータオペレータがアプリケーションと対話するためのプログラムです。通常の 3 層構造環境では、クライアントロジックの関数は、出力の送信とユーザからの入力の受信の他、データの編集および確認に制限されます。ファンクショナリティ層は、会社が在庫管理、注文、経理などの関数を実行するようなビジネス・ロジックから成ります。3 層構造環境では、この層はアプリケーションが実行する、事実上全てのデータ処理を含んでいます。データ層は、RDBMS、フラットファイル、メインフレームまたは企業のデータが格納されている他のリソースから成ります。

3 層構造アーキテクチャは、図 2.2 に示すように、一方ではデータ層とファンクショナリティ層の間、もう一方ではファンクショナリティ層とプレゼンテーション層の間の、明確に定義されたインタフェースに依存します。アプリケーションレベルとモジュールレベルの両方で、インタフェースが明確に定義されているため、異なる GUI へ変更したり、アプリケーションサービスを修正したり、新しい DB への移動などの変更が非常に容易になります。

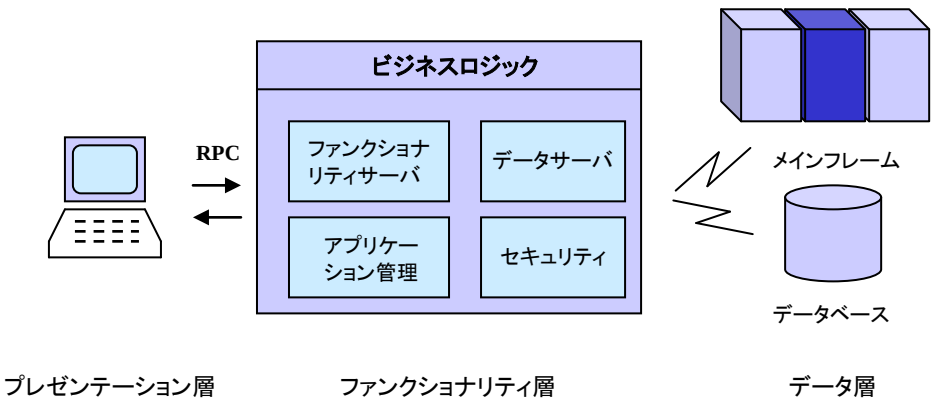


図 2.2 : 3 層構造アーキテクチャ

中間(ファンクショナリティ)層は、多くの形式をとる可能性のある論理的な区分です。たとえば、ある組織において、会社のアプリケーションごとに生のデータアクセスを提供するDB サーバ群を持つとします。各アプリケーション用の別のサーバ群は、アプリケーションのエンドユーザの必要に応じて、さまざまな方法で返された情報を処理します。ビジネス・ロジックは、単にデータを取得してユーザインタフェースに渡す単一サーバではありません。他のサービスの関数を使用できるサービスのセットになることもあります。また、新しいアプリケーションは、必要に応じてあらゆる既存のサービスを使用できます。3 層構造技法により、このような柔軟性と再利用性が向上します。

図 2.3 に示すように、ファンクショナリティ層自体を地理的な要求に応じて物理的に分散することができます。この図では、可能なデータパスが 1 つしかない点に注意してください。別のアプリケーションのサーバが、図中の DB アクセス・サーバを使用することもできます。

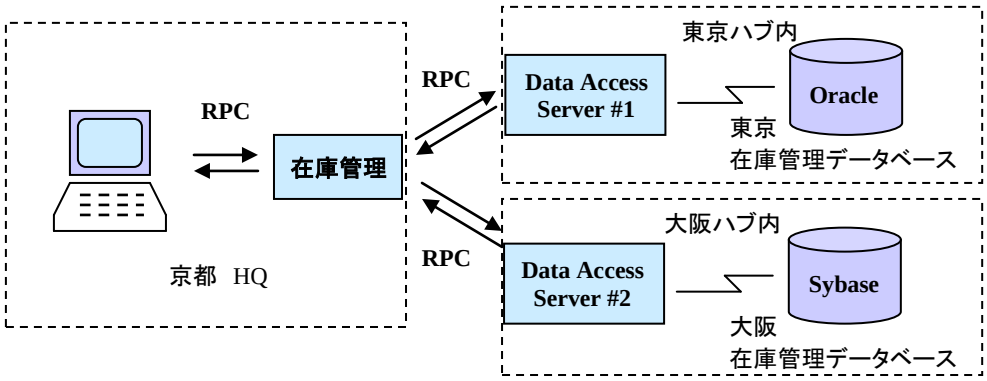


図 2.3 : データパスの例

3 層構造の利点

3 層構造アーキテクチャの根本原理は単純です。アプリケーションは、組織が所有する全てのハードウェアとソフトウェア資源の長所を利用し、短所を回避しなければなりません。たとえば、PowerBuilder または Visual Basic などの GUI を実行している安価な PC は見事なユーザインタフェースですが、ビジネス・ロジックを追加しすぎるとマシンのパフォーマンスが極端に劣化することがあります。これとは逆に、UNIX マシンは UNIX の知識がないエンドユーザが利用することは難しいですが、すばらしい処理能力を提供します。

したがって、エンドユーザが扱いやすい Windows マシンを純粋なユーザインタフェースとして使用し、UNIX マシンの処理能力を使用して、サーバとしてアプリケーションデータを処理するシステムを構築すると双方の利点を生かせます。また、会社が既存のメインフレームや他の DB に大型の投資をしているのであれば、これらの既存のデータソースを分散環境に統合して使うことも必要でしょう。Nextra を導入して3層構造を実現すると、これらのケースの問題点を容易に解決できます。

2 層構造アーキテクチャ

次に、伝統的な 2 層構造のクライアント／サーバアーキテクチャとの違いを説明します。まず、3 層構造でのユーザインタフェースには、2層構造においてパフォーマンスを遅くする原因となるデータ処理が、ほとんど、あるいは全く含まれません。なぜなら、ユーザインタフェース部分は、その特色であるデータ表示しか行わないからです。さらに、PowerBuilder、Visual Basic、JSP、HTML などのクライアントから同一のサーバにアクセスさせることができます。また、クライアントを他の言語に移植する際も、サーバサイドの変更をせずに行える利点があります。第 2 に、サーバプログラムはクライアントになることも可能であり、プレゼンテーション層に属する GUI と同じサーバにアクセスすることも可能です。第 3 に、アプリケーションの自由度の確保です。たとえば、会社が別の会社を買収し、その会社が異なる RDBMS を使用している場合、3 層構造アーキテクチャでは、既存の RDBMS やアプリケーションなどを容易に組み入れることができます。

再利用性

さらに、3 層構造アーキテクチャには、2 層構造アーキテクチャにはない再利用性が保証されています。新しく大きなアプリケーションを構築するとき、データ処理とデータアクセス手法のライブラリを構築して、適切な場合に頻繁に再利用することができます。3 層構造アーキテクチャの導入は本当の意味での投資であり、より短い開発周期でリターンを見込むことができます。既存のサーバコードベースが大きくなるにつれて、新しいアプリケーションの開発期間はより短くなります。

プレゼンテーション層（Presentation tier）

ユーザインタフェースを設計する際には、エンドユーザからできる限り多くの情報を得てください。ユーザインタフェースは、実際に使用するエンドユーザの希望に添ったものでなければなりません。その情報を設計プロセスに含める時期が早いほど、大きな変更が少なくてすみます。また、試作品を使用する機会をエンドユーザに何度も提供してください。多くの変更が発生するでしょう。エンドユーザからの情報が何もない状態でプロトタイプを作成しないでください。

フロントエンドをコーディングするときには、できる限り機能（ファンクショナリティ）を含めないようにしてください。複雑な仕事はファンクショナリティ層にて処理させてください。GUI にてコーディングしたロジックは、異なる GUI では使用できず再コーディングする必要があります。機能をファンクショナリティ層に任せることにより、部門や組織がどんなに多くの異なる GUI を使用しても、ファンクショナリティ層には全く影響しません。

ファンクショナリティ層（Functionality tier）

Nextra は、分散アプリケーション開発者の作業を容易にし、分散アプリケーションの品質を向上させる目的で設計されています。特定の製品に閉じ込められる（Lock in）心配もありません。中間層の作成手法をマスターしてください。この層は、アプリケーションの 90 パーセントの仕事をする場所です。この層での処理・応答時間がアプリケーション全体に影響しますので、この層の設計には特に注意してください。

DBアクセス・サーバ

DB アクセス・サーバを使用することにより、開発者は煩わしい DB 操作言語を記述することなく、SQL ステートメントを記述するだけで、クライアントから DB を操作することが可能です。

GUI クライアントから DB アクセス・サーバの関数を直接呼び出すこともできます。

トランザクション・サーバ

この種類のサーバでは、単一トランザクションの中で複数の DB にアクセスできます。

トランザクション・サーバは、サポートされる RDBMS に格納されている全てのデータについて、コミット（commit）とアボート（abort）の機能を提供します。このタイプのサーバは、2 つ以上の DB 操作が必要な状況で使用されます。

ファンクショナリティ・サーバ

これはビジネス・ロジックとも呼ばれます。ファンクショナリティ・サーバは、各アプリケーションに必要なデータ処理手法(ビジネスルール)を提供します。ファンクショナリティ・サーバを形成するには、データ処理ロジックと IDL ファイルを作成します。

プログラムのモジュール化は、オープン分散環境では強力な武器となります。サーバをモジュール化しておけば、将来のシステム再構築の際に繰り返し使用できます。可能な限り柔軟に扱えるようにサーバを作成してください。新しいアプリケーションを構築する際には、既存のサーバが再利用可能かどうかをまず調べてください。

ファンクショナリティ層は、データにアクセスする手法の 1 セットのライブラリとみなすべきです。アプリケーションの処理に関わらず、同じ DB を使用するアプリケーションは同じ DB アクセス・サーバを使用できます。アプリケーション固有の処理は、異なるサーバのセットに分割してください。

データ層

Nextra は、この層にはあまり作用しません。リモートシステム(メインフレームまたは他の既存のプラットフォーム)と RDBMS は、データ層に含まれます。これらは両方とも情報のソースです。

DB を作成するときは、伝統的な DB 設計ツールと技法によって設計します。DB に多くのファンクショナリティをコーディングしないことをお勧めします。ほとんどの仕事はファンクショナリティ層に位置するビジネス・ロジックにさせてください。データ層の応答時間には注意が必要ですので、設計(正規化の程度、インデックス使用の有無など)を慎重に行ってください。

第 3 章 サーバのチュートリアル

本章では、サンプルプログラムを通じて、分散アプリケーションを構築する方法を説明します。ここでの目的は、分散アプリケーションがどのように構築されるのか、Nextra ユーティリティを使用すると分散アプリケーションの構築がいかに容易であるかを理解していただくことです。

チュートリアルを始めるにあたって

このチュートリアルを始める前に、次のソフトウェアが必要です。

- Nextra ソフトウェアパッケージ：開発パッケージ
- サードパーティ製品：各言語コンパイラ

初めての分散アプリケーション

起動の前に

UNIX

環境変数 ODEDIR に、Nextra をインストールしたディレクトリの tcp サブディレクトリを指定します。PATH の前には必ず、$ODEDIR をインクルードしてください。

［sh または ksh］

```
ODEDIR=/usr/nextra/tcp

PATH=$ODEDIR/bin:$ODEDIR/../cmn/bin:$PATH

export ODEDIR PATH
```

Windows

Windows では、コントロールパネル中の「システム」アプレットを使って、与えられたユーザ名の環境変数を設定してください。 `ODEDIR=`*`install_dir`*`\tcp` を設定し、次を `PATH` 内に含めてください。

```
"%ODEDIR%\bin;%ODEDIR%\..\cmn\bin"
```

ファイルの位置

必要なサンプルファイルは全て、UNIX または Windows 上のディレクトリ $ODEDIR/../samples (Windows では C:¥ODEDIR¥..¥samples)にあります。これらのディレクトリにあるファイルは、このチュートリアルの後半で解説します。

オリジナルファイルを上書きしないように、これらのサンプルファイルは自分の作業ディレクトリにコピーしたものを使用してください。

手順の概要

1. サーバコードを作成します。(実際の開発では、このステップは次のステップと入れ替わります。)
2. IDL ファイルを作成します。
3. RPCMake を使って、クライアントコード(スタブ)、サーバコード(スケルトン)を生成します。
4. サーバまたはクライアントをコンパイルします。
5. 環境ファイルを作成します。
6. クライアントとサーバを実行します。

サーバコードの作成

分散アプリケーションの作成を始めるにあたり、最も迅速な方法は、ローカルなものとしてコーディングすることです。ローカルプログラムが動作するようになれば、それをサーバ部分とクライアント部分に 2 つに分けることができます。

このチュートリアルでは、サーバは、2 つの機能を含む短いプログラムから構成されます。2 つの整数を加算する機能と、小文字の文字列を大文字に変換する機能です。次の例で

は、C で記述された 2 つのローカルプログラムについて説明します。このプログラムでは、ユーザの入力を促し、加算機能と、小文字から大文字への変換機能を実行します。

Cサーバ

まず、加算する整数と変換する文字列の入力を促す、通常の C プログラムをコーディングすることから始めます。そして、加算と変換を行う関数も取り込みます。このローカル C プログラムは「cprog.c」です。

cprog.c

```
#include <stdio.h>
#include <ctype.h>

/* ------------------------------------ */
/* this is the code for the add function */
/* ------------------------------------ */
int add(int x, int y)
{
  return(x+y);
}
/* ----------------------------------------------*/
/* this is the code for the lower2upper function */
/* --------------------------------------------- */
int lower2upper(char *before, char **after)
{
  int i=0;
  char *a;
  *after = (char *)malloc(strlen(before) + 1);
  if (*after==NULL) {
    printf("Could not allocate memory!");
    exit(1);
  }
  a = *after;
  for (i=0; i<strlen(before);i++){
    a[i] = toupper(before[i]);
  }
  a[i] = '\0';
  return i;
}
main(int argc, char **argv)
{
  int one,two,result;
  char instring[100], *outstring;
  char inbuf[100];

  printf("\nPlease enter two numbers to add
```

```
    (e.g. 3,6): ");
      gets(inbuf);
      sscanf(inbuf,"%d,%d", &one, &two);
      result = add(one,two);
      printf("\nThe result is %d\n", result);
      printf("\nPlease enter the string to capitalize:\n");
    gets(instring);
    lower2upper(instring,&outstring);
    printf("\nThe string is <%s>\n", outstring);
    free(outstring);
  }
```

ローカルプログラムをコンパイルし実行します。コンパイラがデフォルトで ANSI コードを認識できない場合は、必ず ANSI オプションを使用してください。

表 3.1 : コンパイラコマンド

OS	C コンパイラコマンド
HP-UX	`cc -Aa`
IBM AIX	`cc`
Windows	`CL`

以下は、UNIX 環境でのコンパイラコマンドのシンタックスです。Windows の互換コンパイラでは、UNIX コンパイラとはフラグと引数のシンタックスがそれぞれ異なります。

```
> command cprog.c -o testprog
```

プログラムをコンパイルしたら、実行ファイル名を入力して実行してください。(上記の UNIX の例では testprog)

作成したコードにバグがなければ、ローカルアプリケーションは入力を促し、予測どおりの結果を返します。

次のステップ

ローカルエラーのないコードを作成するという、開発プロセスの最初のステップが終了しました。

次に、アプリケーションを分散する最初のステップを紹介します。このステップでは、サーバとクライアントの間のインタフェースを定義します。

ローカルコードでは、インタフェースは内蔵のものとなります。メインプログラムルーチンをクライアント、そして、それ以外のルーチンをサーバと考えることができます。

IDLファイルの作成

IDLファイルとは

分散アプリケーションのクライアントは、main()ルーチンがローカルプログラムにあり、サーバはサブルーチンの役割を果たします。このため、やりとりする関数のパラメータをクライアントとサーバが確認する手段が必要になります。 Nextra では、ファイルに書かれた関数定義を使うことによって、この情報をクライアントとサーバに直接組み込みます。

このファイルは IDL（Interface Definition Language）ファイルと呼ばれ、サーバが実行できる各プロシージャの入力、出力を記述するために使われます。このパラメータによって、必要なマーシャリング（スタブとスケルトンのデータパケット交換）と通信ルーチンを生成するために十分な情報が Nextra ツールに与えられます。この通信ルーチンを使って、クライアントはサーバにサービスを要求できるのです。

IDLファイルの作成方法

最初のステップで見たローカルプログラム（cprog.cs)には、サーバのための IDL ファイルを作成するために必要な情報が全て含まれています。プロシージャの入力と出力を調べることによって、IDL ファイルに含めなくてはならない情報が分かります。各プロシージャには戻り値があり、入力または出力パラメータを持ちます。

IDL は、プログラミング言語とプラットフォームに依存しません。

例

前のステップのローカルプログラムに対応した IDL ファイル basics.def を以下に示します。

basics.def

```
# interface definition file for: basics
[uuid(uuid) version(1.0)]
interface basics {
  int add (
```

```
        [in] int x,
        [in] int y);
      int lower2upper (
        [in] char s1[],
        [out] char s2[]);
}
```

簡単な説明

最初の行はコメントです。

2 行目は、UUID を使うことによって、「インタフェース」(IDL におけるサーバ名)を識別します。UUID とバージョン番号を指定してください。 Nextra ではサーバを識別するのに、UUID ではなくサーバ名を使います。このため、正しい形式であれば、任意の UUID を使うことができます。 現在のバージョンでは、UUID は無視されます。

3 行目は、サーバ(またはインタフェース)の名前です。同じネットワークのサーバの中でユニークな名前であれば、任意の名前を付けることができます。

中括弧{ }で囲まれたコードは、全ての関数を記述します。add 関数は int データ型の値を返します。関数は、同じ型の入力パラメータを 2 つとりますが、出力パラメータはありません。lower2upper 関数は、入力パラメータと出力パラメータを 1 つずつとり、整数の戻り値を持ちます。関数パラメータには、サーバコードと同じ名前を付ける必要はありません。しかし、関数名についてはコードと同じものを使う必要があります。これは、関数が名前によって識別されるためです。

次のステップ

IDL ファイルは、次のステップ(通信ロジックの生成)で使用されます。**RPCMake** ユーティリティは、IDL ファイルを読み込み、クライアントとサーバが互いに通信するのに必要なネットワーク通信ロジック(スタブ・スケルトン)を生成します。

クライアント・スタブとサーバ・スケルトンの作成

IDL ファイルで指定された情報に基づいて、ネットワーク通信ロジック(スタブ・スケルトン)を作成する準備ができました。**RPCMake** ユーティリティが、このサービスを提供します。スタブ・スケルトンがローレベルの通信を処理するので、開発者が通信部分を意識する必要はありません。

makeによる自動的なスタブ・スケルトンの生成

C のサンプルコードに含まれる Makefile には、make コマンド実行時に、自動的にスタブ・スケルトンを生成するように記述されています。make コマンドを使用する場合は、ここでスタブ・スケルトンを生成する必要はありません。ここでスタブ・スケルトンを生成しても、Makefile 中のコマンドが、コンパイル時にスタブ・スケルトンを上書きします。もちろん、ここでスタブ・スケルトンを作成しても問題はありません。

RPCMake コマンドラインでは、IDL ファイル名を指定するために-d オプションを、クライアント・スタブの言語を指定するために-c オプションを、そして、サーバ・スケルトンの言語を指定するために-s オプションを使用します。次の例を参照してください。

C サーバ

C 言語でサーバとクライアントのスタブを生成するには、次のコマンドを入力します。

```
> rpcmake -d basics.def -c c -s c -c perl
```

RPCMake は `basics_s.c` というサーバ・スケルトン、`basics_c.c` というクライアント・スタブ、および `basics.h` というヘッダファイルを生成します。

また、Perl クライアント・スタブ `basics_c.pl` も生成します。後で、**RPCDebug** テストユーティリティで使用します。

RPCMake GUI

RPCMake を呼び出すときに、コマンドラインでオプションを省略して実行すると、**RPCMake** の GUI が起動されます。GUI を使用すると、ボタンのクリックによって、さまざまな言語でスタブ・スケルトンを生成することができます。また、IDL ファイルの検索に便利な「定義ファイル」ウィンドウもあります。

RPCMake の詳細については、『リファレンス』を参照してください。

サーバコードのコンパイルの準備

ローカルコードからサーバコードとクライアントコードを切り離す作業は、ローカルコードのコピーを作成して行ってください。決して、完成しているローカルコードを直接変更しないでください。(これまでの作業の確定)

C サーバ

ローカルの C プログラムをネットワークサーバに変更するには、ローカルプログラムをわずかに書き換えるだけですみます。サーバコードのサンプルは basics.c です。basics.c と cprog.c を比べると、変更がきわめて少ないことが分かります。

必要な手順

必要な手順は次の 2 つのみです。

- 各メモリ割り当て関数を、対応する **dce_** メモリ割り当て関数に置き換える 。
- main()ルーチンを削除する。

これで、ローカルプログラムのコードをコンパイルしてサーバにするための書き換えが終了しました。

例

完成されたサーバソースコード `basics.c` は次のようになります。

```
basics.c

  #include <stdio.h>
  #include <ctype.h>
  #include <string.h>
  #include "dceinc.h"

  /* ------------------------------------ */
  /* this is the code for the add function */
  /* ------------------------------------ */
  int     add(int x, int y)
  {
```

```
    return(x+y);
}

/* ------------------------------------------------ */
/* this is the code for the lower2upper function    */
/* ------------------------------------------------ */
int lower2upper(char *before, char **after)
{
    size_t i=0;
    char *a;

    *after = (char *)dce_malloc(strlen(before) + 1);
    a = *after;
    for (i=0; i<strlen(before); i++) {
       a[i] = toupper(before[i]);
    }
    a[i] = '\0';
    return i;
}
```

クライアントコードのコンパイル準備

ローカルコードからクライアントコードを切り離す作業は、ローカルコードのコピーを作成して行ってください。決して、完成しているローカルコードを直接変更しないでください。(これまでの作業の確定)

Cクライアント

クライアントコードが必要とするのは、ソースコードに追加する標準起動ルーチンだけです。必要に応じて、**RPCMake** が生成するヘッダファイルをインクルードすることもできます。この場合、クライアント関数呼び出しのパラメータがサーバコードのパラメータに一致しないと、コンパイラから警告を受けます。

必要な手順

クライアントソースコードについて、次の作業が必要です。

- ローカルプログラムからサーバルーチンを削除する。
- free()呼び出しを、dce_release()に置き換える。これは、サーバから動的配列を受信するときのみ、必要な手順である。
- 標準起動ルーチンを作成する。
- RPC でエラーが生じたときに、エラーメッセージを出すのための処理部を加える。

例

完成したクライアントソースコード cclient.c をローカル C プログラムと比べてください。クライアントコードは、**ボールド体**で書かれた標準起動ルーチンを含みます。

cclient.c

```
#include <stdio.h>
#include "dceinc.h"
#include "basics.h"

main(int argc, char **argv)
{
    int one,two,result;
    char envfile[100];
    char instring[100], *outstring;
    char inbuf[100];

    printf("Please enter the name of the env file: ");
    gets(envfile);
    printf("File: %s", envfile);
    if (!dce_setenv(envfile,NULL,NULL)) {
        printf("Error: %s\n", dce_errstr());
        exit(1);
    }
    printf("\nPlease enter two numbers
                              to add (e.g. 3,6): ");
    gets(inbuf);
    sscanf(inbuf,"%d,%d", &one, &two);
    result = add(one,two);
    if (dce_errnum()!=0)  {
        printf("Error: %s\n", dce_errstr());
        exit(1);
    }
    printf("\nThe result is %d\n", result);
    printf("\nPlease enter the string
                                 to capitalize:\n");
    gets(instring);
    lower2upper(instring,&outstring);
    if (dce_errnum()!=0)  {
        printf("Error: %s\n", dce_errstr());
        exit(1);
    }
    printf("\nThe string is <%s>\n", outstring);
    dce_release();
    return (0);
}
```

これで、コード作成は終了です。サーバとクライアントのコンパイルの準備ができました。

サーバとクライアントの実行ファイル作成

UNIX 環境で C の実行ファイルをコンパイルするコマンドは、Makefile というファイルにあります。このファイルはサンプルファイルと同じディレクトリにあります。

必要な手順

C サーバをコンパイルするには、make server と入力します。

C クライアントをコンパイルするには、make client と入力します。

両方をコンパイルするには、make all と入力します。

出力ファイルは basics と cclient であり、分散アプリケーションを構成する実行ファイルになります。

Windows では、アプリケーションをコンパイルするためにサンプルの makefile を使用してください。

環境ファイルの編集

サーバまたはクライアントを実行する前に、環境ファイルを用意する必要があります。

サーバとクライアントは、このファイルを使ってブローカの位置を決め、デバッグ出力レベルを設定します。その一方、ブローカは、このファイルを使ってクライアントとサーバからの通信を受けるポートを決定します。ブローカ起動時に、このファイル(サンプルでは broker.env) を指定する必要があります。

必要な手順

サンプルの環境ファイルを見てください。DCE_BROKER が指定する IP アドレスが、ブローカを実行するマシンを指すように変更してください。(ここでは、今使っているマシンを

使ってください。) マシンのネットワーク名が分からない場合は、hostname コマンドで調べてください。このコマンドの結果を、下記のサンプル環境ファイルの 193.1.1.60 と置き換えてください。

ブローカを実行しようとして、エラーが発生した場合は、別のポート番号で試してください。使用できるポートは、2048～65535 の中の未使用ポートです。UNIX マシンでは、次のコマンドで使用中のポートが分かります。

```
> netstat -an | grep port_num
```

指定したポートが表示された場合、そのポートは使用中ということになります。そのポートに関する応答が何もない場合、そのポートは空いています。

変更を環境ファイルに保存して、server.env と client.env という名前でコピーを 2 つ作ります。DCE_LOG 属性には、それぞれ別のログファイル名を指定してください。broker.env で指定したログファイルをブローカが使用して、サーバとクライアントは指定した別のログファイルを使用します。

例

サンプルファイルに含まれている環境ファイルは次のとおりです。

```
DCE_BROKER=193.1.1.60,7800
DCE_DEBUGLEVEL=DEBUG,DEBUG
DCE_LOG=broker.log
```

この設定により、高レベルのデバッグ情報が broker.log というカレントディレクトリのファイルに書き込まれ、ブローカはマシン 193.1.1.60 のポート番号 7800 で接続を待ちます。

環境ファイルの詳細については、『リファレンス』の「ファイル仕様」の章を参照してください。

分散アプリケーションの起動

ここで、全てが正しく動作していることを確認します。クライアントとサーバの実行ファイル、および環境ファイルが同じディレクトリにあるかどうか確認してください。このディレクトリからブローカを起動します。

```
> broker -e broker.env &（UNIX ブローカ用）
```

> start /b broker -e broker.env（Windows ブローカ用）

このコマンドはブローカをバックグラウンドプロセスとして起動します。ブローカは環境ファイルを読み込んで、ファイルに指定されたポートで起動します。

C サーバ

C サーバを構築したら、次のコマンドで起動します。

> basics -e server.env &　　（UNIX 用）
> start /b basics –e servere.env（Windows 用）

サーバは、環境ファイルを読み込んでブローカにアクセスし、使用可能サーバとして登録します。

Cクライアント

C ユーザインタフェースを構築したら、次のコマンドで起動します。

> cclient

このコマンドはクライアントを起動して、環境ファイルの入力を促し、その要求を満たすことができるサーバの名前と位置情報について、ブローカにコンタクトします。それから入力が促されます。手順に沿って、動作中の分散アプリケーションを確認してください。

C クライアントを使用して Java サーバにアクセスしたり、.NET クライアントを使用して C サーバにアクセスしたりすることができます。このような相互運用性がオープン分散環境なのです。

次のステップ

次に、加算する数字と、変換する小文字の文字列を入力して、結果を調べます。全てのプログラムが同じマシンで動作していますが、RPC 呼び出しにて相互通信を行っています。

RPCDebugの試行

クライアントを使用する代わりに、RPCDebug ユーティリティ(汎用クライアント)を使って新しいサーバのテストを行えます。

```
> rpcdebug
```

Windows では、「RPC Developer」アイコンをダブルクリックして起動してください。

このユーティリティによって、サーバが提供する各関数の動的変数を使って呼び出せるようになり、サーバ関数が正しく動作しているかどうかをテストできます。

1. ウィンドウが表示されたら、「環境ファイル」ボタンをクリックして、環境ファイルをロードします。

2. 次に、「定義ファイル」ボタンをクリックして、サーバに対応する IDL をロードします。

3. 「RPC Functions」ボックスから、テストする関数を選択します。関数名をクリックしてください。add、または lower2upper を選択します。

4. 選択した関数の入力値を入力します。「Inputs」ボックスで、適切なフィールドに値を入力します。

5. 「実行」ボタンをクリックします。

6. 始めは数秒かかりますが、「Outputs」ウィンドウに関数の戻り値が表示されます。

次に、入力を変更するか、別の関数を試してください。「実行」をクリックすると、ほぼ同時に応答があります。これは、クライアント・スタブ (RPCDebug は生成した Perl クライアント・スタブを使います) が使用可能なサーバのリストを内部に保管しており、最初の RPC 実行時、または適切なサーバが見つからない場合にのみブローカに問い合わせるからです。図 3.1 に、RPCDebug ウィンドウを示します。

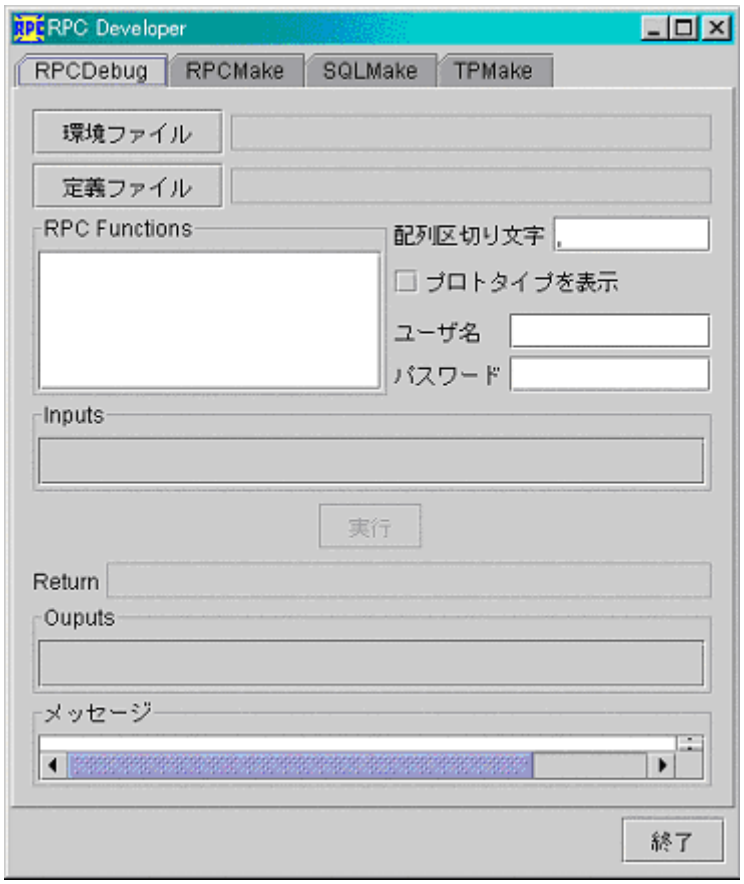


図 3.1 : RPCDebug メインウィンドウ

最後に

ここでひと休みして、最初の分散アプリケーションの完成を祝ってください。☺

第 4 章 チュートリアルの解説

この章では、前章での作業について解説します。

サーバコードの作成

サーバコードには、サーバ・スケルトンとそこから呼び出されるサーバ関数のみが含まれます。これは、生成されたサーバ・スケルトンが、サーバ関数を呼び出すメインプログラムルーチンを含んでいるためです。メインルーチンはクライアントから引数を受け取り、実行する適切なサーバ関数を選択します。

サーバのコーディング方法を理解したら、チュートリアルで示したようなローカルプログラムを半分に切り離す方法ではなく、サーバコードを直接作成したくなりませんか？

サーバコードでの重要な変更点は 、**dce_**という接頭辞の付いたメモリ割り当て関数の置き換えでした。これについては、「Cの関数引数のためのメモリ割り当て」を参照してください。

また、実際の開発では、クライアントをサーバの開発と並行して進められるように IDL ファイルを先に作成します。

IDLファイルの作成

IDL ファイルは、アプリケーションにとって最も重要な部分です。それは、正しく通信を行うために、クライアントもサーバもこのファイルに依存するためです。

正しい IDL ファイルがあれば、サポートされる言語またはスクリプトによって通信ルーチンを生成することができます。

IDL ファイルは、各サーバのアプリケーションプログラミングインタフェース (API) を定義するので、開発チームの別のメンバーが並行して作業することができます。IDL ファイルに定義された関数を使えば、ユーザインタフェースとサーバロジックを同時に開発することができます。これが、サーバとクライアントのコーディングに先だって、IDL ファイルを作成する理由です。

スケルトンの生成

サーバは 1 つのサーバ・スケルトンを使用します。 サーバ・スケルトンは、サーバコードが実行できる RPC だけを扱うことが必要です。

クライアントがアクセスできるサーバは 1 つだけではないので、クライアントは必要な数のクライアント・スタブを持つことができます。クライアントがアクセスするサーバごとに、クライアント・スタブを生成してください。

サーバが、別のサーバにとってのクライアントとして動作するようにするなら、その別のサーバの IDL ファイルからクライアント・スタブを生成して、C の場合はコンパイルして、クライアント・スタブをサーバコードにインクルードします。

スタブ・スタブの言語は、インクルードまたはコンパイルされる、クライアントまたはサーバコードと同じでなければなりません。

環境ファイルの作成

ブローカ、サーバ、クライアントは、起動時に環境ファイル中の設定を調べて、設定をカスタマイズします。その際、コマンドラインオプションよりも、環境ファイルの設定を優先します。

環境ファイルには、多くの環境属性を指定できます。属性の種類については『リファレンス』を参照してください。

アプリケーションの起動

ネーミング・サービス(ブローカ)、サーバプログラム、およびクライアントプログラムは、決まった順序で起動しなくてはなりません。

アプリケーションの起動では、ブローカに登録できなければサーバが起動されないので、ブローカを先に起動しなくてはなりません。

ブローカが起動されていたら、サーバを起動することができます。そして最後にクライアントを起動してください。

まとめ

以上で、Nextra ランタイムの動作方法と、基礎的なアプリケーションの作成方法を理解されたことでしょう。

『サーバ開発者ガイド』の残りの部分では、クライアントおよびサーバアプリケーションエレメントの構築方法の詳細を説明します。Nextra ツールの機能と能力については『リファレンス』および『運用／設定ガイド』を参照してください。

第 5 章 開発プロセス

この章では、Inspire International Inc. が推奨するアプリケーション仕様と設計プロセスについて説明します。また、Nextra 分散アプリケーションの心臓部であるクライアント／サーバコンビネーションを実現するための技術的なステップについても説明します。

この章の後半で説明されるプロセスは、残りの章で解説されています。

Nextra開発パラダイム

Inspire International Inc.が提供する教育コースでは、パイロットアプリケーションや基本的なアプリケーションの仕様を定義するさまざまな手段として、ユーザインタフェースの設計を紹介しています。しかし、規模が大きくなり、複数ユーザアプリケーションになれば、さらに多くのことが必要になります。このようなシステムは処理速度を最適化して、データの重複を避け、参照整合性を保証しなくてはなりません。さらに、良いアプリケーションと呼ばれるためには、使いやすいものでなければなりません。このため、厳密なデータフロー解析とデータモデル定義が必要になります。

複雑なシステムでは、ユーザインタフェースの設計は、開発プロセスの中の重要なステップです。エンドユーザからの情報収集と「実在チェック」を継続して行うことによって、データがどのように見えればよいか、またデータフロー中でのユーザの役割が何であるかをユーザが認識するために、フロントエンドの設計は役に立ちます。自然と、ユーザインタフェースはパラメータに合うように最適化されます。

しかし、実際のデータモデルとデータフローでは、システムの効率を第一に考えなくてはなりません。と言っても、データドリブンのアプリケーション開発に戻るということではありません。今日のソフトウェア開発の能力と柔軟性をもってすれば、両者の良い部分を持つアプリケーションを開発できます。つまり、非常に使いやすいユーザインタフェースと、処理が最適化されたデータモデルの両立が可能なのです。

使いやすいユーザインタフェースと最適化されたデータモデルという 2 つのゴールを考えると、分散アプリケーションの初期設計には 3 つの重要な要素があげられ、ます。

- 開発途中からユーザに参加してもらう。
- データモデルを定義する。
- データフローを構造的に解析する。

これらのプロセスの詳細を説明することは本書の対象外となっていますので、あらかじめご了承ください。次に示すガイドラインとヒントは、アプリケーションの構造を決定するのに役立つでしょう。

開発中のユーザの参加

開発者は、開発中にエンドユーザと密な連絡をとるようにしてください。パイロットシステムが稼働したときのテスト時間を節約できるだけではなく、新しい観点が提供されることや、より洗練された解決策が生まれることもあるからです。さらに、ユーザがこの段階で変更を指定できるようにすれば、アプリケーションを製品化する時点で行うよりも簡単に変更することができます。全ての段階において、開発チームのメンバーは、ユーザの仕事をより簡単に、そして効率的にするためにシステムを開発しているのだということを念頭に置かなくてはなりません。

データモデルの定義

ユーザには、名前、住所、誕生日、過去の購入品、口座情報といったアクセスする必要のある データの一括リストを作成してもらいます。さらに、ユーザは情報の種類だけではなく各種の情報量も開発者に知らせることが必要です。住所が 20 文字なのか 100 文字なのか、コメントフィールドは 40 文字なのか 4000 文字なのかというようなことです。

全てのユーザがデータセットについて合意したら、データエントリが互いにどのように関連するか、そしてその関連性の深さ (つまり、1:1、1:n なのか、必須:任意なのかなど)を決めるのは設計チームの責任になります。データモデルは、次の事柄のバランスがとれるように設計しなくてはなりません。

- コーディングを最小にする:洗練された方法でユーザにデータを渡す。
- データの冗長性を最小にする:データストレージを節約し、データの矛盾発生を防ぐ。
- パフォーマンスを最大にする:インデックスを使って、DB トランザクションを高速化する。
- 的確な整合性:データの破壊を防ぐ DB アクセス手法で設計する。

3 層構造の出現によって最初の 3 つにはほとんど変更がありませんが、分散アプリケーションで参照整合性をどのように含めるかという新しい問題が発生します。次に示すガイドラインは、データ整合性をどのように保証するかを設計チームが判断するのに役立つでしょう。

データフローの構造解析

ユーザと開発者は、ユーザがどのようにデータを利用し、ユーザとDBの間でどのように処理すべきかを分析しなくてはなりません。

このプロセスでの設計チームは、どの階層をファンクショナリティ層にするかを決定しなくてはなりません。ここでの目的は、ユーザインタフェース中のデータ処理を最小にして、ファンクショナリティ層中のサーバにコーディングされるプロセスを最大限にすることです。これで、モジュールアーキテクチャを目指す設計手法に基づいて、ファンクショナリティ層がサーバに分けられます。また、このアプローチによって、最大限にモジュール化されたシステムを構築できます。ユーザインタフェース、サーバ、そしてDBを追加したり削除したりする際は、他のモジュールへの影響を最小にして、新しいモジュール自身のコードを最小にすることが必要です。

たとえば、 Visual Basic ユーザインタフェースが複雑な演算から得られるフィールドを渡し、2番目のフィールドと掛け合わせ、結果を3番目のフィールドに格納するようなコードを含んでいるような場合を考えてください。同じインタフェースの PowerBuilder バージョンを作成するには、全てのコードを移植して書き直さなくてはなりません。これとは対照的に、ファンクショナリティ部分がサーバに集中していれば、新しい PowerBuilder インタフェースでは、新しい RPC を作成するだけですみます。もちろん、サーバコードを変更する必要はありません。

開発プロセスの概要

ここでは、サーバとクライアントを構築し、それらのやりとりをデバッグするために必要なステップを説明します。

詳細については、「ファンクショナリティ・サーバ」、および『クライアント開発者ガイド』を参照してください。

環境変数の設定

『はじめにお読みください』の「インストール後の環境設定」を参照してください。

分散アプリケーションの構築

ユーザインタフェースの設計を終え、分散アプリケーションのアーキテクチャ設計を終えて適切な環境変数が設定されていることを確認したら、個々のクライアント／サーバのために次のステップを実行してください。

サーバの構築

1. IDL(定義)ファイルを作成します。

サーバが処理するリモート関数のシンタックスを定義します。サーバが実行する RPC ごとの名前、戻り値の型、パラメータの型です。

2. サーバコードを作成します。

RPC 経由でサーバが提供する関数のセットを実行するコードを作成します。

	**デバッグ** 一時的にプログラムに 1 セットのローカル関数を追加して実行して、サーバコードをローカルプログラムとしてテストしてください。シンタックスエラーをチェックして、関数呼び出しが正しい値を返すことを確認してください。

3. サーバ・スケルトンを生成します。

サーバ・スケルトンを自動的に生成するために、IDL ファイルを **RPCMake** ユーティリティの入力として指定します。 必要に応じて、テストするプラットフォームにスケルトンを移動してください。サーバがコンパイル言語 (C または COBOL)でインプリメントされる場合は、サーバの実行ファイルをコンパイルしてください。

4. 環境ファイルを作成、編集します。

テストプラットフォーム上の環境ファイルは、ブローカが実行される位置を反映します。これで、サーバは実行可能です。

	**デバッグ** ランタイム分散アプリケーションにインプリメントする前に、新しいサー

	バをテストしてください。

テストは次のいずれかで行います。

- 新しいサーバに対して、リモート関数呼び出しを行うテストクライアントを構築して実行する。
- インタラクティブに呼び出しを行うために、**RPCDebug** ユーティリティを使用する。

サーバがコンパイル言語でインプリメントされる場合は、サーバの実行ファイルと環境ファイルを実行プラットフォームに移動してください。

サーバがインタプリタ言語でインプリメントされる場合は、サーバ・スケルトン、サーバコードを含むファイル、そして環境ファイルを実行プラットフォームに移動してください。

クライアントの構築

1. クライアントコードを作成します。

RPC を発行するコードを作成してください（特定の RPC のシンタックスを確認するには、サーバの IDL ファイルを参照してください）。

	**デバッグ** RPC をコメントアウトして実行することによって、クライアントコードをローカルプログラムとしてテストして、シンタックスエラーをチェックしてください。

クライアントが GUI の場合には、GUI 設計とプロトタイプの段階で、コーディングがほぼ終わっていることが必要です。

2. クライアント・スタブを生成します。

クライアント・スタブを自動的に生成するために、クライアントがアクセスするサーバごとに、サーバの IDL ファイルを **RPCMake** ユーティリティの入力に指定します。必要に応じて、スタブをクライアントテストプラットフォームに移動します。

3. 必要なファイルを使ってクライアントプログラムを設定します。

クライアント・スタブにリンクしたりアクセスしたりするのと同じように、クライアントプログラムは Nextra ランタイムライブラリにリンク、またはアクセスすることが必要です。クライアントがコンパイル言語でインプリメントされる場合は、クライアントの実行ファイルをコンパイルしてください。

4. 環境ファイルを作成、または編集します。

クライアントテストプラットフォーム上に環境ファイルを用意します。

以上で、クライアントはそのスタブを含むサーバが実行する関数を起動できるようになります。

クライアントがコンパイル言語でインプリメントされる場合は、クライアントの実行ファイルと環境ファイルを実行プラットフォームに移動してください。
クライアントがインタプリタ言語でインプリメントされる場合は、クライアント・スタブ、クライアントコードを含むファイル、そして環境ファイルを実行プラットフォームに移動してください。

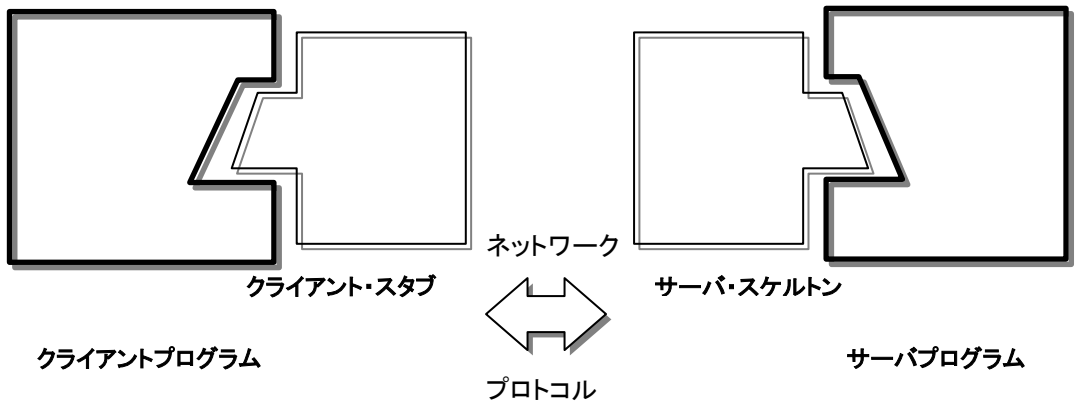
分散アプリケーションのデバッグ

ローカルアプリケーションと比較して、分散アプリケーションではエラーの要因が 2 つ追加されます。開発者が作成したコードが、シンタックスの上で正しいだけではいけないのです。

- 通信ロジックが正しく生成されているか？
- クライアントとサーバのプラットフォームは、TCP/IP で通信できる状態にあるか ？

サーバとクライアントの構築の説明では、順を追ってデバッグプロセスが入っていて、開発者はモジュールごとに分散アプリケーションをテストしました。ここでは、デバッグプロセスを総合的な観点から見てみましょう。

ネットワーク上に分散されたアプリケーションをデバッグする際に忘れてはならないのは、エラーの切り分けです。もっと大きなスケールで見れば、実行状態のシステムに追加する前に、新しいパーツは全てエラーフリーでなければなりません。開発者のレベルでは、開発者が開発ステップごとにデバッグを行い、生成されたコードにエラーが残るのを防ぎ、次のステップの基盤をしっかりつくらなければなりません。また、適切なデバッグ手法(つまり、エラーの切り分け)を用いれば、エラーが 1 つか 2 つのソースに限定されるので、時間を節約することができます。



1. ローカルプログラムとしてサーバをテスト
2. ローカルクライアントからサーバをテスト
3. ローカルプログラムとしてクライアントコードをテスト
4. クライアント／サーバインタフェースをテスト

図 5.1 :　デバッグ手法

このデバッグ手法に完全に沿っていれば、開発者は各デバッグステップでエラーを分離して、開発に費やす時間を大幅に削減できます。

1. サーバをローカルプログラムとしてテストします。

サーバ・スケルトンを生成する前に、サーバコードをローカルプログラムとしてテストすると、サーバコード中のシンタックス、およびプロシージャの問題を切り分けることができるので、エラーがネットワーク通信の問題からくるものかどうかを気にする必要がありません。

2. サーバを個々のプラットフォームからテストします。

ステップ 1 の後、かつサーバを作成する前に、個々のプラットフォームからサーバの実行ファイルをテストすると、エラーが IDL ファイルにあるのか、サーバコードにあるのかを特定できるので、エラーがネットワーク通信の問題からくるものかどうかを気にする必要がありません。

3. クライアントコードをローカルプログラムとしてテストします。

クライアント・スタブを生成する前に、クライアントコードをローカルプログラムとしてテストすると、クライアントコード中のシンタックス、およびプロシージャの問題を切り分けることができるので、エラーがネットワーク通信の問題からくるものかどうかを気にする必要がありません。

4. クライアント/サーバインタフェースをテストします。

ステップ 1、2、3 を終えて、クライアント/サーバ インタフェースをテストすると、問題がネットワーク通信によるものか、あるいはクライアントの関数呼び出しとサーバの関数定義の相違によるものかを切り分けることができます。

上記のステップでは、エラーを迅速に見つけ解決するのに役立つ一般的なデバッグのアプローチを説明しました。忘れてはならないステップを以下にまとめます。

- 別のコードを生成する前に、自分が記述したコードのシンタックスと機能を確認する。
- 可能であれば、実際のプロシージャの入力、出力およびデータ型について、IDL ファイルをクロスチェックする。
- スタブ生成(必要であればコンパイルも)後、**RPCDebug** ユーティリティを使って、新しいサーバが正しく動作するかどうかを確認する。
- 自動的に関数呼び出しを行うクライアントを作成して、全てのサーバコード内のロジックを実行する (このステップは、アプリケーションに合わせて調整しなければならない)。

第 6 章 ファンクショナリティ・サーバ

この章では、ファンクショナリティ・サーバに焦点を絞って、サーバの構築とデバッグの一般的なプロセスを説明します。DB アクセス・サーバの構築の詳細については、本書の該当する章を参照してください。トランザクション・サーバの構築については、該当マニュアルを参照してください。

この章では、前章で説明した概念を参照するので、これより先に進む前に、前章をお読みになることをお勧めします。

なお、Java言語によるアプリケーション(ファンクショナリティ)サーバの構築の詳細については、この章を一読後、「第 9 章 Javaアプリケーション・サーバの構築」も併せてお読みください。

IDLファイルの作成

IDL ファイルは、与えられたサーバが実行可能なファンクションを定義するテキストファイルです。

IDL ファイルには二重の意味があります。第 1 に、IDL ファイルを **RPCMake** ユーティリティに与えることによってスタブが自動的に生成されるので、面倒な通信ロジック部分のコーディングが不要になります。第 2 に、サーバが IDL ファイル中に定義されるので、サーバ自身のインプリメンテーションとは無関係に、サーバインタフェースの仕様を形式化でき、別のプログラマがその仕様どおりにコーディングすることが可能になります。分散環境内で、クライアントとサーバの並行開発が可能となります。

IDL ファイル内の定義は言語に依存しませんので、COBOL で書かれたサーバと、同じ機能を実行する C で書かれた別のサーバがあれば、両方のサーバで同じ IDL を使用することができます。

サーバ情報の収集

サーバの IDL を作成するには、次の情報を集める必要があります。

1. サーバ名、およびサーバのユニバーサル固有識別番号 (UUID)

ランタイムライブラリは名前でインタフェースを識別しますので、インタフェース名を指定しなければなりません。

インタフェース名は、英数文字またはアンダースコア(_)の組み合わせでなければなりません。

個々のインタフェース名は、ネットワーク上の全てのインタフェース中でユニークでなければなりません。

定義ファイルには UUID フィールドが含まれなければなりませんが、内容は現在の Nextra バージョンでは無視されます。(使用されません。)

2. サーバのバージョン番号

バージョン番号を個々のサーバに関連付ける必要があります。このバージョン は必ず指定しなければなりませんが、値は無視されます。(使用されません。)

3. サーバが扱う、全ての関数名と戻り値の型

IDL で指定する関数名は、サーバコードの実際の関数名に一致しなければなりません。関数名は、クライアントが呼び出す全ての関数の中でユニークなものでなければなりません。

4. 各関数の全ての入力、出力パラメータの名前とデータ型

出力パラメータは、サーバからクライアントに渡されます。入力パラメータはクライアントからサーバにのみ渡さ れます。

パラメータ名を選ぶときは、パラメータの「並び」がサーバコード中の対応するパラメータに一致するかどうかについてだけ注意してください。IDL ファイルで宣言されたパラメータ名は、サーバコード中の対応するパラメータに一致していなくてもかまいません。変数は名前ではなく、位置によって参照されます。

属性に関するリストは、『リファレンス』の「データ型情報」を参照してください。

ファイルのシンタックス

情報をまとめたら、次の形式に従って IDL ファイルを作成してください。

```
# RPC Interface Definition File for: interface_name
[uuid(uuid) version(ver)]
```

```
interface <<$>>interface_name {
    <<declarations>>
    data_type procedure_name (
    <<[in]|[out]>> data_type variable_name<<[]>> <<,
    ...
   <<[in]|[out]>> data_type variable_name<<[]>> >>);
    data_type procedure_name (
    <<[in]|[out]>> data_type variable_name<<[]>> <<,
    ...
    <<[in]|[out]>> data_type variable_name<<[]>> >>;
}
```

表 6.1: IDL ファイルのシンタックス

シンボル	意味
#	行の最初が非空白文字の場合、#は行の残りの部分がコメントであることを示します。
uuid	UUID。この値は形式上必要ですが無視されます。
ver	サーバのバージョン番号。この値は形式上必要ですが無視されます。
interface_name	サーバを識別するための、ネットワーク上での全てのサーバの中でユニークな名前。インタフェース名は、文字で始まり、25 文字を超えてはいけません。各 IDL ファイルには、1 つのインタフェース定義のみが許されます。名前の先頭にドル記号 ″$″ が付いている場合、サーバ名は変数となり、起動時にサーバ名を指定できます。詳細については、『運用／設定ガイド』の「バリアブル・ネームド・サーバ(Variable named server)の概要」を参照してください。
data_type	データ型宣言。*data_type* が *procedure_name* より前にくる場合、関数の戻り値のデータ型を宣言します。 *data_type* が *variable_name* より前にくる場合、入力変数、出力変数、または戻り値のデータ型を宣言します。 有効な戻り値:`char`, `double`, `float`, `int`, `short`, `long`, `void`
procedure_name	指定したクライアントから呼び出される関数の間でユニークなリテラル文字列。プロシージャ名は文字で始まり、80 文字を超えてはいけません。サーバプログラムが実行できる関数の名前を付けます。クライアント・プログラムはこの関数を *procedure_name* によって呼び出し、この関数はサーバコードでは *procedure_name* となります。名前を `dce_`, `DCE_`, `ODE_`, `ode_`で始めることはできません。
`[in]|[out]`	このいずれかの値は関連する変数が入力変数か出力変数かを指定します。入力は値がクライアントからサーバに渡されることを意味します。ブラケット[ ] が必要です。

variable_name<<[]>>	関数内でユニークなリテラル文字列。変数名は文字で始め、80 文字を超えてはいけません。変数は dce_で始めてはいけません。クライアントとサーバコードで、対応する変数は同じ名前ではなくてもかまいません。しかし、変数は位置によって参照されるため、プロシージャの定義の最初の変数は、プロシージャパラメータリストとクライアントの呼び出しパラメータリスト中の、最初の変数に対応します。他の変数は同様に参照されます。 オプションの括弧を付けると、配列変数を示します。数字の値、または他の変数値を括弧内に挿入すると、固定長配列(*注 1)またはコンストレインド配列。 (*注 1)を宣言することになります。空の括弧は、NULL ターミネーテッド配列(*注 2)を示します。(char データ型のみで許される。) (*注 2)IDL データ型を参照。

定義ファイルの例

C, COBOL サーバコードに関連する定義ファイルを以下に示します。

C の定義ファイル

```
server.def

#IDL file for:cserver (or cblserv)
[uuid(123) version(1.0)]

interface cserver {
    long c_add (
        [in] long x,
        [in] long y,
        [out] long z);

}
```

クライアントが、いずれかのサーバに交互にコンタクトする場合は、異なる言語で作成されていても、同じ名前 (cserver)でサーバを呼び出すことができます。両者を区別したい場合は、異なった interface 名を使って IDL ファイルを分離します。(バリアブル・ネームド・サーバを作成することもできます。詳細は『運用／設定ガイド』と『リファレンス』を参照してください。)

ネーミングルール

IDL ファイル名には、拡張子.def をつけることをお勧めしますが、この拡張子は任意です。たとえば、cserver.def は上記の IDL ファイルの名前です。

データ型の宣言

データ型の宣言を説明するために、この後に出てくる send_file 関数をインプリメントするサーバコードに関連する IDL ファイルを以下に示します。

```
file_server.def

#IDL for: file_server
[uuid(123) version(1.0)]

interface file_server {
    short send_file (
        [in] char filename[],
        [out] long arraysize,
        [out] char outarray[arraysize]);
}
```

COBOLサイズ情報のインクルード

COBOL サイズファイル、および IDL ファイルは共に NULL ターミネーテッド配列と互換性のある COBOL スタブ・スケルトンを生成するために、**RPCMake** ユーティリティに対する入力として使用されます。

COBOL サイズファイルには、IDL ファイルに現れるものと同じ関数名を含みます。個々の NULL ターミネーテッド配列について、次のシンタックスに従って、関数ラベルの下に COBOL サイズファイル中のエントリを追加してください。

```
#cobol variable size [number]
```

表 6.2： COBOL サイズファイルのシンタックス

variable	定義ファイル(.def)中の任意の変数名。(たとえば、select 文の中の列名。) SQL 文の中のカラム名にエイリアスが使用されている場合は、variable 名は列名ではなくエイリアスに対応します。
size *(列: COLUMN)*	配列の各要素のバイト数 。(たとえば、カラム中で最も大きな要素のサイズ、または列定義で許される最大サイズ。) 1/2 次元共通です。

	文字型での注意: データの最終バイトを NULL 終了しなければいけません。よって、COBOL ユーザロジック内で、dce_null_terminate を使用して NULL 終了してください。
number *(行:ROW)*	2 次元文字配列のみでの使用。配列の行数になります。 配列の行数。デフォルトは 200 または rcmax で設定される値 です。

固定長文字配列について

定義ファイル(.def)に直接、サイズを指定したもので COBOL サイズファイルを使用しない場合です。コンストレインド文字配列と違い、データの最終バイトを NULL 終了する必要はありませんが、COBOL ユーザロジック内では size - 1 バイト数しか利用できません。

関数ラベルの下に次のステートメントを挿入することによって、指定した関数の全ての配列について、*number*を設定できます。

```
#cobol rcmax number
```

COBOLサイズファイルの例

```
get_tt:
#cobol rcmax 400
#cobol tool_type_name 40 rcmax
get_tl:
#cobol tool_name 40
```

解説:

- 関数 get_tt で宣言された変数 tool_type_name は 2 次元コンストレインド文字配列(Constrained string array)として、400 行、40 列が割り当てられます。
- 関数 get_tl で宣言された変数 tool_name は 2 次元コンストレインド文字配列として、200 行、40 列が割り当てられます。

次のステップ

サポートされるデータ型の詳細については、『リファレンス』の「データ型情報」の章を参照してください。また、IDL ファイルの詳細については『リファレンス』の「ファイル仕様」の章を参照してください。

これで、次のステップであるサーバコードの生成に進むことができます。

サーバコードの作成

サーバコードは、プロシージャ、または関数定義内にパッケージ化されたファンクショナリティのユニットから構成されます。また、サーバコードは、サーバの IDL ファイルに定義された各プロシージャと同じだけの関数定義を含みます。各サーバ関数は、IDL ファイルに定義された戻り値とパラメータの型を使用します。

インクルードファイル

Cインクルードの問題

サーバ・スケルトン作成時に生成されたヘッダファイル(詳細は、「サーバ・スケルトンの生成」を参照)、および標準のRPCライブラリ用ヘッダであるdceinc.hをインクルードする必要があります。

サーバコードの例

Cの例

ANSI C で作成されたサーバコードの例を示します。

```
cserver.c

#include "cserver_s.h"
#include < dceinc.h >
long c_add(long x, long y, long *z)
{
```

```
    *z=x+y;

}
```

この短いサンプル cserver.c はサーバコードの要件を満たしています。このサーバコードを、後のステップで生成されるサーバ・スケルトンとリンクします。

COBOLの例

COBOL で作成されたサーバコードの例を示します。

cblserv.cbl

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CADD.
DATA DIVISION.
LINKAGE SECTION.
01 X PIC S9(9) BINARY SYNC.
01 Y PIC S9(9) BINARY SYNC.
01 Z PIC S9(9) BINARY SYNC.

PROCEDURE DIVISION USING X, Y, Z.
  DO-C-ADD.
  MOVE 0 TO Z.
  ADD X, Y TO Z.
  GOBACK.
END PROGRAM CADD.
```

このファイルを cblserv.cbl と呼びます。

一般的なコーディングの問題

予約語

OEC_, oec_, dce_, DCE_で始まる関数名と変数名はツールの中で予約されています。これらの文字で始まる名前を使うとエラーとなるか、予測できない動作になる可能性があります。

Cのポインタデータ型－文字列配列の引渡し

文字列配列を渡すときは、配列が NULL ターミネーテッドであることを確認することが重要です。

たとえば、3 つの文字列（*instring1*, *instring2*, *instring3*）の配列をサーバに送り、サーバコードが *outstring1*, *outstring2*, *outstring3* に対する各文字列を個別に処理するようにしたい場合は、次のようにメモリを outstring の配列に割り当ててください。

- *pointer1*－> *outstring1*
- *pointer2*－> *outstring2*
- *pointer3*－> *outstring3*
- *pointer4*－> *通常は未定義*

pointer4 が未定義のままだと、サーバが SIGSEGV エラーでクラッシュする原因となります。よって次のように指定しなければなりません。

- *pointer4 は NULL を設定する。*

COBOLの問題

サーバは PROGRAM モジュールから成り、これは RPCMAIN モジュールによりディスパッチされます。

戻り値

IDL ファイルで COBOL サーバファンクションの宣言に戻り値がある場合は、その関数の引数リストの最後に特別の整数引数を追加しなければなりません。COBOL では、C のような戻り値をサポートしません。さらに、戻り値を特別な変数に明示的に割り当てなければならないため、コードの Linkage セクション中で変数を宣言するように注意してください。この変数の値は戻り値としてクライアントに再びマーシャリングされて、呼び出しプログラムにとっては、COBOL ルーチンが C のような戻り値をサポートするように見えます。DCE_RESULT は戻り値を保持する変数です。

Nextra 関数を呼び出すときは、ポインタ引数はリファレンス渡しに、レギュラー引数は値渡しにしてください。

NULLターミネーション

サイズファイルにて列サイズを指定する文字型(Constrained String Array:コンストレインド文字配列)を利用する場合、COBOL ユーザロジック内では、全ての文字列を NULL 終了しなければなりません。

変数を COBOL ユーザロジック内で NULL 終了(ターミネーテッド:Terminated)にするには、2 つの方法があります。1 つは、個々の変数に対して dce_null_terminate を呼び出す方法です。

```
CALL "dce_null_terminate" USING login.
CALL "dce_null_terminate" USING passwd.
```

2 つめの方法は、変数と定数の両方に使用することができます。

```
STRING "server.env" DELIMITED BY SIZE, LOW-VALUE
                        DELIMITED BY SIZE INTO ENVFILE.
CALL "dce_setenv" USING ENVFILE, DCE-NULL, DCE-NULL
giving RV.
IF RV = 0
  DISPLAY "SET ENV WITH ", ENVFILE, " FAILED."
  STOP RUN
END-IF.
```

C配列に変換されるCOBOL配列の例

size ファイルを使用して char 型の配列変数を宣言するとき(コンストレインド配列)は、NULL ターミネータキャラクタ分の 1 バイトが必要な点に注意してください。たとえば、IDL ファイルで、

```
[in] char item_in[],
[out] char item_out[],
```

と定義されている変数に対して、10 文字の文字列を COBOL サーバから入出力する場合は、COBOL サーバコードの LINKAGE SECTION に以下のように記述します。

```
01 item_in PIC X(10).
01 item_out PIC X(10).
```

COBOL サイズファイル:

```
#cobol item_in 11
#cobol item_out 11
```

というように、COBOL サーバコードで指定したサイズに NULL 用の 1 バイトをプラスしたサイズを指定しなければなりません。仮に、この例で

	COBOL サイズファイル: `#cobol item_out 10` とすると、COBOL サーバから文字列“1234567890”を出力する場合、クライアント側が受け取る文字列は“123456789\n”となります。

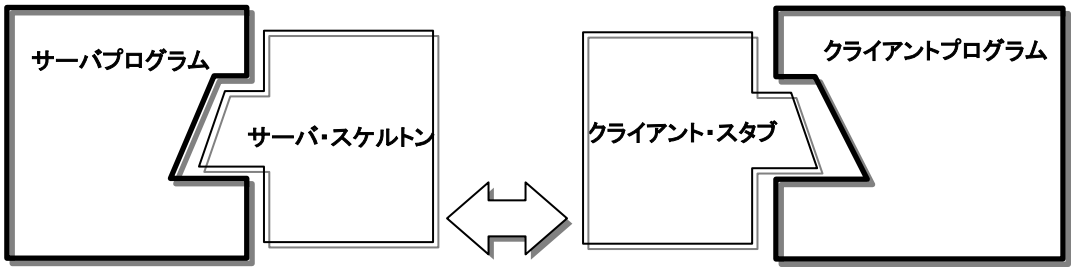
Cの関数引数のためのメモリ割り当て

C 言語とそのスーパーセットは、ポインタとして定義されるデータ構造を含むプログラムに、メモリ管理機能をエンコードする 必要があります。ポインタで参照されるデータのために、メモリを割り当てる(関数群の malloc を使用)コードと、使用後にそれを解放するコードを作成しなければなりません。

ローカルの C プログラムでは、メモリリークを防ぐために、メモリを割り当てたり、解放したりすることができます。これに比べて、クライアント／サーバモデルでは、それぞれのメモリスペースにアクセスする 2 つのモジュールについて処理が必要です。つまり、クライアントコードとクライアント・スタブはクライアントマシンのメモリにアクセスし、サーバコードとサーバ・スケルトンはサーバマシンのメモリにアクセスするのです。

標準の malloc 関数を使用すると、オープン分散環境では問題が発生します。たとえば、クライアントが返す動的引数のためにサーバがメモリスペースを割り当てるような場合、ネットワーク経由で引数が送り返されるまで、メモリスペースを解放してはいけません。しかし、ネットワークに送り出す前に、サーバコードはスケルトンに結果を返すので、サーバコードはメモリを解放することができません。スケルトンだけが解放できます。

同じように、クライアント・スタブが入力動的配列のためのメモリを割り当てる場合は、クライアント・スタブはメモリを解放することができません。これは、スタブが結果を返した後で、クライアントコードがその引数を使うためです。クライアントコードが引数を使い終えた後、メモリスペースを解放できるのです。



1. サーバプログラムが指定されたメモリを割り当てる。
2. サーバ・スケルトンはネットワーク経由で情報を送り、サーバが割り当てたメモリを解放する。
3. クライアント・スタブはサーバ・スケルトンから情報を受け取り、メモリを割り当てる。

4. クライアントプログラムは受け取ったデータを読み出し、クライアント・スタブが割り当てたメモリを解放する。

図 6.1: Nextra を使ったメモリの割り当て

Nextra では、メモリ割り当ての問題を解決するために、独自の割り当て関数を提供します。これらの関数 (dce_malloc(), dce_realloc(), dce_calloc())は、標準の C の関数と同じ引数と戻り値をとります。これらの関数を使用して、動的出力引数(NULL ターミネーテッド、およびコンストレインド配列)のために、サーバマシンのメモリを割り当ててください。 ネットワーク上に出力値を送り出すと、スケルトンは自動的に別の Nextra 関数 dce_release()を呼び出します。この関数によって、Nextra はサーバコード中のメモリ管理関数が割り当てた全てのメモリを解放します。

クライアント側では、クライアント・スタブは dce_malloc()関数群を使って、受け取る動的引数のためのメモリを割り当てます。メモリ使用後に解放するために、クライアントコードに dce_release()の呼び出しを追加してください。

Nextra は、サーバに渡された引数のための割り当てを全て自動的に処理します。このため、表 6.3 には動的引数の型だけが示されています。なお、クライアントからサーバに情報を送る場合は、Nextra メモリ関数を使う必要はありません。これは、共有リソースとしてのサーバの性質によるものです。

表 6.3 : 動的データ型

IDL データ型では、サーバコード(パラメータ x のみ)には dce_malloc()が、クライアントコードには dce_release()が必要になる。	
[out] char *x* []	[out] long *n* [out] any_type *x* [*n*]
[out] char *x* [][]	[out] long *n* [out] char *x* [*n*][*n*]

Cのメモリ割り当ての例

dce_malloc()関数の使い方を説明するために、次のサーバコードのサンプルを見ていきます。このコードは、クライアントが要求したファイルを送るために、コンストレインド配列にメモリを割り当てます。(バイナリファイルをクライアントに送る手順を特に説明するために、エラー処理の部分は除かれています。)

```
#include <stdio.h>
#include <string.h>
#include "file_server_s.h"
```

```
short send_file(char *filename, long *arraysize, char
**out_array)
{
    long filedescr, returnval, maxlen, packet=1000,
                          offset=0;
    char *buffer;
    FILE *fileptr;

    maxlen=packet;
    buffer=(char *)dce_malloc(maxlen);

    fileptr=fopen(filename,"r");
    if (fileptr == NULL) {
       printf("ERROR: unable to
                        open file <%s>\n",filename);
    }
    filedescr=fileno(fileptr);
    while
  ((returnval=read(filedescr,&(buffer[offset]),packet))>0)
    {
       offset+=returnval;
       maxlen+=packet;
       if (returnval<packet)
          break;
       buffer=(char *)dce_realloc(buffer, maxlen);
    }
    *out_array=buffer;
    *arraysize=offset;
    fclose(fileptr);
    return 1;

}
```

このコードは、出力引数のためのメモリをコンストレインド配列として定義します。通常は `malloc()`を使用するところで、`dce_malloc()`関数を呼び出す点に注意してください。このサーバのためのサーバ・スケルトンは、ファイルをネットワーク上に送り出した後、`dce_release` を呼び出します。これで、上記のコード中の `buffer` に割り当てられたメモリが解放されます。

Windowsの静的配列

Windows のアプリケーションは、オプションでコンパイル時にモジュール定義ファイルを使用することができます。Windows の、クライアントまたはサーバのいずれかで静的配列を使

用するときは、定義ファイルの HEAPSIZE を配列に適応する大きさに設定しなければなりません。例を示します。

```
char array1[ARRAY_SIZE1];
char array2[ARRAY_SIZE2];
```

上記のように配列を定義した場合、アプリケーションのモジュール定義ファイルの HEAPSIZE 文は、配列サイズより大きい値を持たなければなりません。

```
HEAPSIZE ARRAY_SIZE1 + ARRAY_SIZE2 + (other variable space)
```

ファイルに実際の変数名を入れるのではなく、合計値を得るために、3 つの数を足す点に注意してください。このため、ARRAY_SIZE1=6000 とおよび ARRAY_SIZE2=7000 の場合は、たとえば、モジュール定義ファイルに次のような行を入力します。

```
HEAPSIZE 16384
```

ネーミングルール

サーバコードを含むファイル名は、そのコードが書かれた言語に応じた拡張子が付きます。

表 6.4： サーバコードのネーミングルール

言語	拡張子
C	.c
COBOL	.cbl

サーバ名には、次の文字だけを使用することができます。

- 英大文字 (A-Z)
- 英小文字 (a-z)
- 整数 (0-9)
- アンダースコア (_)

サーバコードのデバッグ

早い段階でコードをテストすれば、サーバコードで発生するシンタックス、またはプロシージャのエラーを突き止め、後で発生する問題を回避することができます。

一時的にローカル関数呼び出しを追加して、サーバコードをテストしてください。

1. 定義した関数を呼び出すメインルーチンを追加します。

   このルーチンは、プロシージャをローカルに呼び出します。これにより、ネットワークトランスポートを追加してシステムが複雑になる前に、エラーを突き止めることができます。

2. サーバをコンパイルします。

3. プログラムを実行します。

   シンタックスエラーをチェックしてください。関数呼び出しから返された値が予測通りかどうかを確認してください。

Cの例

たとえば、次の行を前出の C のサンプルに追加することができます。

```
main()
{

    long a,rv;
    rv=c_add(1,2, &a);
    printf("The return value of add is %d \n", a);

}
```

このプログラムを C コンパイラでコンパイルして、正しい結果が返されるかどうかを確認します。

次のステップ

サーバコードの作成と、ローカルプログラムとしてのテストを終えたら、次のステップであるスタブ・スケルトンの生成に進むことができます。

サーバ・スケルトンの生成

IDL ファイルを作成したら、インタフェースのためのサーバ・スケルトンとクライアント・スタブを生成するために、**RPCMake** ユーティリティを使用します。**RPCMake** には、次の情報を指定する必要があります。

- IDL ファイル名
- サーバ・スケルトンを作成する言語

サーバごとにサーバ・スケルトンは 1 つだけ使用します。サーバ・スケルトンは、サーバロジックが記述された言語を指定します。

サーバ・スケルトンを生成するには、コマンドラインから **RPCMake** を呼び出すか、GUI を起動して生成します。

コマンドラインからのRPCMakeの呼び出し

RPCMake を呼び出して、サーバ・スケルトンを生成するには、次のシンタックスを使用します。

```
> rpcmake -d file.def -s language1 [-p file.procs] [-size
file.size] [-k]
```

ここで、*file*.def は定義ファイル名、*language1* はサーバ・スケルトンを生成する言語の省略形になります。C の場合は c、COBOL の場合は mfcobol を使用します。

COBOL サーバ・スケルトンを生成し、コードが NULL ターミネーテッド配列を処理する場合は、COBOL サイズファイルを指定するために、-size オプションを指定しなければなりません。

RPCMake の最も簡単な形式について見てきました。1 回の **RPCMake** 呼び出しで、複数の言語のクライアント・スタブ、サーバ・スケルトンを生成することができます。それぞれ-c、または-s オプションを使用して複数の言語を指定することができます。**RPCMake** の機能の詳細については、『リファレンス』の「Nextra ユーティリティ」の章を参照してください。

問題が発生した場合

RPCMake でエラーが発生したら、2 つの点を確認してください。(1)ODEDIR 環境変数が、使用する Nextra ツールキットに正しく設定されているか、また PATH 環境変数に $ODEDIR/bin ディレクトリを含んでいるかどうかを確認する。(2) IDL ファイル指定の際に、IDL ファイルが **RPCMake** を起動したディレクトリにあることを確認するか、定義ファイルのフルパス名を指定する。

C の例

RPCMake の使用方法を説明するために、次のコマンドで IDL ファイル `cserver.def` から C のサーバ・スケルトンを生成してください。

```
> rpcmake -d cserver.def -s c
```

COBOLの例

次のコマンドは、`cblserv.def` の定義を使って、`cblserv.cbl` で処理されるサーバ・コードのための COBOL サーバ・スケルトンを生成します。

```
> rpcmake -d cblserv.def -s mfcobol -size cblserv.size
```

GUIを使ったRPCMakeの呼び出し

GUI で **RPCMake** を起動するには、次のコマンドを入力します。

```
> rpcmake
```

RPCMake が起動されると、次のウィンドウが表示されます。

図 6.2： RPCMake の GUI 画面

1. 定義(IDL)ファイルの名を指定します。

定義(IDL)ファイル名をテキストボックスに入力するか、「定義ファイル」ボタンをクリックします。クリックした場合は、別のウィンドウが表示されます。

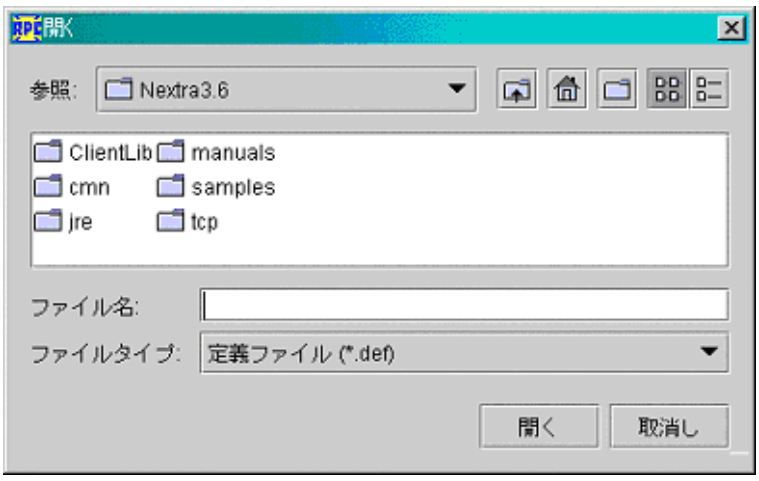


図 6.3：ファイル選択ウィンドウ

このウィンドウで、指定する定義（IDL）ファイルを選択して「開く」をクリックします。

2. Perl を使用する場合は、最初の RPCMake ウィンドウに戻って、プロシージャ名を入力します。

Perl サーバ・スケルトンを生成しない場合は、「プロシージャファイル」フィールドは空白にしておきます。Perl を使用する場合は、Perl サーバ・スケルトンを含むファイル名を入力してください。

3. COBOL の場合は、「サイズファイル」フィールドに COBOL サイズファイルの名前を入力します。

COBOL サーバ・スケルトンを生成しない場合は、「サイズファイル」フィールドは空白にしておきます。COBOL を使用する場合は、サイズ（NULL ターミネーテッド配列サイズ情報を含む）ファイル名を入力してください。

4. サーバ・スケルトン、およびクライアント・スタブ言語を選択します。

「サーバ」の下の言語オプションの 1 つをクリックしてください。サーバ・スケルトンとサーバコードの言語は同じでなければなりません。そして、「クライアント」の下の言語オプションをクリックして、クライアント・スタブの言語を選択してください。

5. 必要に応じてファイルを編集します。

「編集」ボタンをクリックすると、ファイルを編集するためのテキストエディタが起動されます。

6. スタブ・スケルトンを生成します。

「スタブ・スケルトン生成」ボタンをクリックして、選択したスタブ・スケルトンを生成してください。

7. RPCMake を終了します。

「終了」ボタンをクリックしてください。

生成されたファイル

RPCMake を終了すると、コマンドラインで指定した-s オプション、または GUI で選択した言語ボタンによる新しいファイルがカレントディレクトリに生成されます。C でスタブを生成した場合は、ディレクトリにはスタブとヘッダファイルが生成されます。ファイル *server*.h

をクライアントコードにインクルードします。ファイル *server*_s.h をサーバコードにインクルードします。

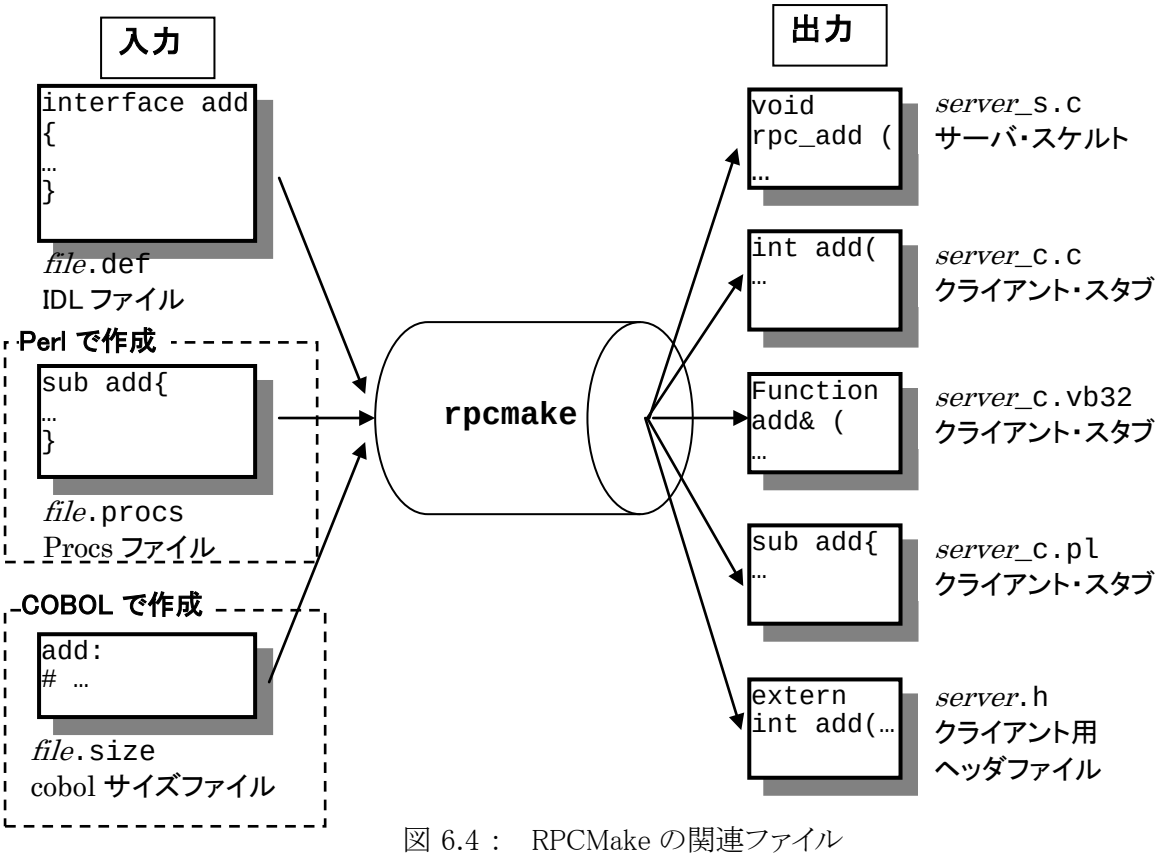


図 6.4： RPCMake の関連ファイル

例

前出 C の例では、**RPCMake** により次のファイルが生成されます。

```
cserver_c.c
cserver_s.c
cserver.h
```

ネーミングルール

スタブ・スケルトン名の接頭語は、定義(IDL)ファイルで指定したインタフェース名からネーミングされます。スタブ・スケルトン名のネーミングにより、それがサーバ・スケルトンなのかクライアント・スタブなのか、さらにインプリメントされた言語が何であるかを識別できます。

表 6.5 : スタブ・スケルトン ネーミングルール

言語	サーバ・スケルトン拡張子	クライアント・スタブ拡張子
C	_s.c	_c.c
COBOL	_s.cbl	_c.cbl
Perl	_s.pl	_c.pl
Java	_s.java	_c.java

次のステップ

サーバ・スケルトンを作成したら、次のステップ、サーバのコンパイルに進むことができます。

サーバのコンパイル

COBOLでのコンパイル

COBOL で作成されたサーバの場合、次のステップは、サーバの実行ファイルをコンパイルすることになります。

```
> cob -xe "RPCMAIN" cobol_files libraries -o executable_name
```

x オプションは、オブジェクトファイルではなく、実行ファイルを生成するようにコンパイラに指定するものです。e "RPCMAIN"オプションは、エントリファンクションが「RPCMAIN」という名前であることを示します。この情報は重要です。なぜなら、この関数名はサーバ・スケルトンで「RPCMAIN」となるようにハードコーディングされるからです。

cobol_files は、たとえば `cblserv.cbl` のような COBOL ソースファイルのリストになります。

libraries は必要なライブラリのリストになります。Nextra COBOL プログラムでは、少なくとも `librpc.`*ext*、`librpccobol.`*ext* が必要になります。

executable_name は作成される実行ファイルの名前になります。

コンパイルのオプション

開発パッケージ付属のサンプルコード内にある、Makefile（makefile.nt<Windows 用>）を参照してください。

C／C++でのコンパイル

C／C++で サーバを作成した場合は、次のステップで、サーバの実行ファイルをコンパイルします。ANSI C 言語ファミリーでサーバを作成した場合は、以下のステップでコンパイルします。

- 先にコンパイルしたオブジェクトファイルを RPC ランタイムライブラリと一緒にリンクする必要があります。UNIX では、インストレーションディレクトリ$ODEDIR/lib に librpc.*ext* があります。
- コンパイルプロセスの全てのステップでは、開発者はプラットフォーム固有のフラグをコンパイラに指定しなくてはなりません。

UNIX makefileの例

C サーバコンパイルのサンプルでは、`make server` と入力すれば、次の `makefile` が UNIX 環境でサーバをコンパイルします。

```
SERV = server_name

CC = `getplatform cc`
LD = `getplatform ld`

SOBJ = $(SERV).o $(SERV)_s.o
COBJ = $(CLNT).o $(SERV)_c.o
EXTRALIBS = `getplatform netlib`
LIBS = `getplatform libdir` `getplatform lib`
INCS = -I$(ODEDIR)/include `getplatform inc`

.c.o:
	$(CC) -c $< $(INCS)

all: server

server: $(SOBJ)
	$(LD) -o $(SERV) $(SOBJ) $(LIBS)

clean:
```

```
        rm -f *.o
rm -f $(SERV) $(CLNT) *_[sc].*
```

server_name には、拡張子.c を削除したソースコードファイル名を指定してください。このサンプルでは、`cserver.c` ですが、次のように指定します。

```
SERV = cserver
```

compiler_command には、プラットフォーム固有のコンパイラコマンドを指定します。次の表に、使用するコマンドをまとめます。

表 6.6： コンパイラコマンド

OS	CC の設定
HP-UX	`cc -Aa -D_POSIX_SOURCE`
IBM AIX	`cc`
Solaris	`acc -D__unix -Dunix -D_REENTRANT -D_POSIX_SOURCE -D_solaris_ -DODEDCE`
Windows	`CL`

高度なMakefile の記述方法

必要であれば、Makefile を使って、コンパイルだけではなく別の作業を行うことができます。次の行を Makefile に追加して、RPCMake を呼び出すことによりサーバ・スケルトンを同時に生成することができます。

```
$(SERV)_s.c: $(SERV).def
   rpcmake.real -d $(SERV).def -s c -y
```

或いは次の行を追加します。そうすると、make clean、または make clenaup とコマンドラインで指定することによって、不要なファイルが削除されます。

```
        clean:

rm -f *.o

        cleanup: clean

rm -f $(SERV) $(CLNT) *_[sc].*
```

Windows makefileの例

nmake を使用して呼び出すことができる makefile の例を示します。例では、クライアントとサーバの両方をコンパイルしています。

```
#THE FOLLOWING NEED TO BE SET:
# ODEDIR= Directory where Entra tools are installed
# SERV= Server Executable name
# CLNT= Client Executable name

SERV = basics
CLNT = cclient

CC = cl /nologo /I $(ODEDIR)/include
LD = cl

SOBJ = $(SERV).obj $(SERV)_s.obj
COBJ = $(CLNT).obj $(SERV)_c.obj
LIBS = -link /SUBSYSTEM:console /NOLOGO
$(ODEDIR)/lib/librpc.lib
INCS = -I$(ODEDIR)/include

.c.o:
  $(CC) -c $< $(INCS)

all: server client

server: $(SOBJ)
  $(LD) -o $(SERV) $(SOBJ) $(LIBS)

client: $(COBJ)
  $(LD) -o $(CLNT) $(COBJ) $(LIBS)

$(SERV)_s.c $(SERV)_c.c $(SERV).h: $(SERV).def
  rpcmake -d $(SERV).def -c c -s c -y

clean:
  del *.sbr *.exe *.obj *_[sc].* $(SERV).h *_c.c *_s.c
```

次のステップ

サーバのコンパイルに成功したら、次のステップ、環境ファイルの作成に進むことができます。

環境ファイルの作成

環境ファイルには、少なくとも 1 つのブローカの位置を指定する属性が含まれます。オプションでは、デバッグ、エラーロギング、コンフィギュレーションに関連する属性の設定を含めることもできます。起動時に、サーバは環境ファイルを参照して、登録するブローカの位置を調べます。

プラットフォームごとに環境ファイルが 1 つしか必要ない場合であっても、サーバごとに環境ファイルを作成することをお勧めします。こうすれば、それぞれのサーバからのログを別々のファイルに保存することが出来ます。

基本的なシンタックス

環境ファイルを作成するための基本的なシンタックスは次のとおりです。

```
DCE_BROKER=broker_host, port_num
[DCE_LOG=log_file]
[DCE_LOG_MAXSIZE=size]
[DCE_DEBUGLEVEL= normlevel, errlevel]
```

[ ]内のテキストはオプションですが、指定することをお勧めします。イタリック体で示した内容は、次の表のとおりです。

表 6.7 : 基本的な環境ファイルの属性

属性	説明
broker_host	ブローカが実行されているマシンの IP アドレスまたはマシン名。
port_num	ブローカが接続を待っているポート番号。使用するポート番号が分からない場合は、10000 の辺りの値を使用します。環境ファイルで同じホストとポートが指定されているかどうかを確認してください。
log_file	サーバからのログが出力されるファイル。分からない場合は `server.log` を使用してください。
size	*log_file* の最大サイズ(バイト単位)。デフォルトは 1000000 または 1MB。
normlevel	正常操作時のデバッグレベル。分からない場合は、WARNING を使用してください。
errlevel	発生前、およびエラー条件中に使われるデバッグレベル。分からない場合は、DEBUG を使用してください。

環境ファイル名に使用できるのは、次の文字のみです。

- 英大文字 (A-Z)
- 英小文字 (a-z)
- 整数 (0-9)
- アンダースコア (_)

環境ファイルには、多くのオプション属性と用途があります。環境ファイルの詳細については、『リファレンス』の「ファイル仕様」の章を参照してください。

次のステップ

エラーフリーの環境ファイルを作成したら、サーバを実行できるようになります。次のステップである、ランタイム時のサーバのテストに進んでください。

サーバのテスト

製品アプリケーションでサーバを実行する前に、広範囲のテストが必要です。RPCDebug ユーティリティの GUI を使用すると、入力引数を指定してサーバの RPC を実行し、テストすることができます (代わりに、**rpctest** を使用して、同じサーバプログラムにテキストベースインタフェースでアクセスすることもできます)。

ここでは、次のことを確認するためにサーバをテストします。

- IDL(定義)ファイルが正しく作成されているかどうか (IDL ファイルでの関数宣言が、サーバコードでの関数宣言に一致するかどうか)。
- 最新のサーバ・スケルトンが生成されているかどうか。
- サーバの実行ファイルが正しくコンパイルされたかどうか。

また、問題を特定するために、サーバを実行するのと同じプラットフォームから、サーバのテストを行ってください。サーバが正しく動作することを確認してから、後の段階でネットワーク関連のデバッグを行います。

以下のステップでは、RPCDebug の使用法を示します。RPCDebug は IDL ファイルを使用します。

設定

次のステップに沿って、サーバのテスト環境を設定してください。

1. RPCDebug で使用する Perl クライアント・スタブを生成します。

RPCMake を使用して、Perl クライアント・スタブを生成してください。サーバが書かれている言語に関わらず Perl を使用します。次のように入力してください。

> rpcmake -d *file*.def -c perl

この Perl クライアント・スタブを使用して、テストユーティリティ(RPCDebug)はサーバと通信を行います。

2. ブローカを起動します。

ブローカがまだ実行されていない場合は、サーバの環境ファイルをコピーして、DCE_LOG の設定を別のファイル、たとえば broker.log に変更してください。そして、新しい環境ファイルを使って、ブローカをバックグラウンドプロセスとして起動してください。

> broker -e *broker_env_file* & [UNIX の場合]

> start /b broker -e *broker_env_file* [Windows の場合]

ブローカで問題が発生したと思われる場合は、ログファイル (環境ファイルで DCE_LOG 属性を指定したもの)を調べてください。ログファイルには、ブローカで発生した全てのエラーが記録されます。

3. サーバを起動します。

サーバをバックグラウンドプロセスとして起動します。

> *server_name* -e *server_env_file* & [UNIX の場合]

> start /b *server_name* -e *server_env_file* [Windows の場合]

サーバ起動時に問題が発生したと思われる場合は、ログファイル (環境ファイルの DCE_LOG 属性で指定されたもの)を調べます。ログファイルには発生したサーバエラーが全て記録されます。

4. RPCDebug を起動します。

サーバの環境ファイルをコピーして、ログファイルを別のファイル、たとえば debug.log のように変更します。このファイルと、生成した Perl クライアント・スタブの両方をカレントディレクトリに置くか、RPCDebug のファイル選択ウィンドウで指定しなければなりません。

```
> rpcdebug
```

あるいは、テキストベースのインタフェースを使用します。次のように入力して起動します。

```
> rpctest -e debug_env_file client_stub_name
```

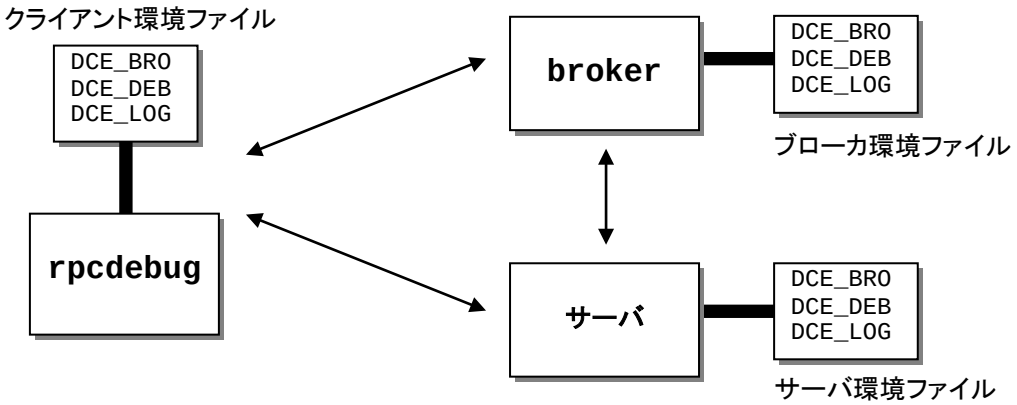


図 6.5： サーバのテストのための構成

RPCのテスト

RPCDebugの使用

RPCDebug コマンドを入力すると、次のウィンドウが表示されます。

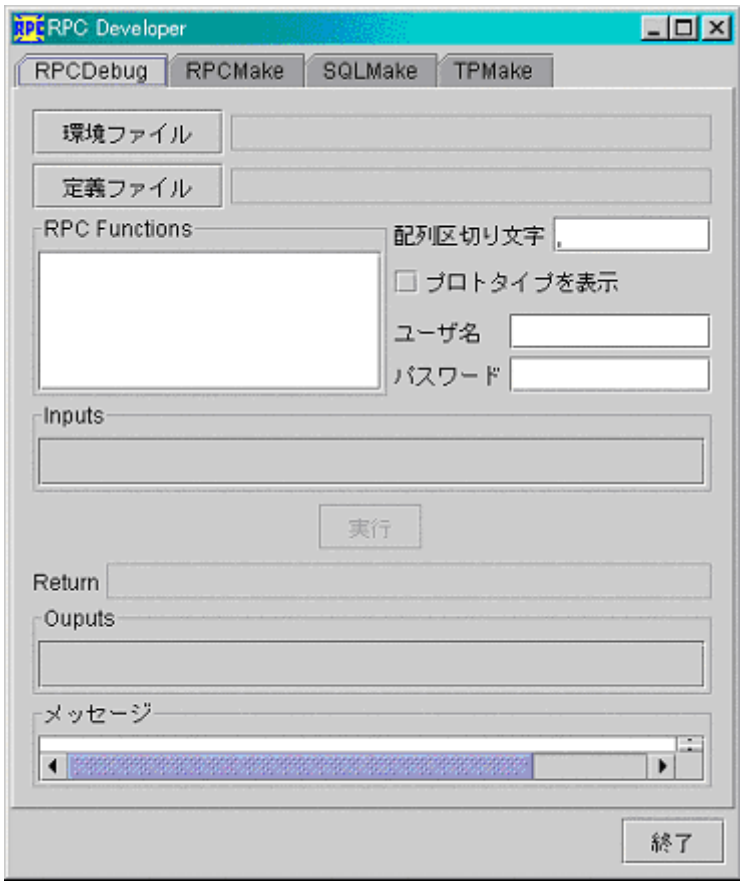


図 6.6： RPCDebug の GUI 画面

1. 「定義ファイル」ボタンをクリックして IDL（定義）ファイルを選択して、ファイル選択ウィンドウを表示します。

2. ファイル選択ウィンドウで、起動したサーバに対応する IDL ファイルを選択します。

   正しい IDL ファイルを選択したら「了解」をクリックしてください。

3. 「RPC Functions」リストボックス中の名前をクリックして、テストする関数を選択します。

4. 「入力」ボックスで、各入力引数の値を入力します。

5. 「実行」ボタンをクリックして、関数呼び出しを実行します。

出力変数とその値が「出力」ボックスに表示されます。値が予測どおりのものであるかどうか確認してください。

6. 別の関数をテストします。

「RPC Functions」ボックス中の別の関数をクリックして、新しい入力値を指定します。そして、「実行」ボタンを再度クリックしてください。

7. ユーティリティを終了します。

サーバ中の全ての関数をテストしたら、「終了」ボタンをクリックして終了ください。

RPCtestの使用

RPCtest ユーティリティを起動すると、次のプロンプトが表示されます。

```
rpctest>
```

rpctest>プロンプトで RPC を実行し、標準 Perl シンタックスに準拠した結果を表示することができます。

次のシンタックスで関数呼び出しを実行してください。

```
rpctest> &func_name(in_arg1, in_arg2, ..., in_argN, *out_arg1,
*out_arg2, ..., *out_argN)
```

関数のパラメータが分からない場合は、IDL ファイル中の関数定義を調べてください。*in_arg* が[in]パラメータに、 *out_arg* が[out]パラメータにそれぞれ対応します。この対応は、位置に依存します。*in_arg1* は IDL 中の最初の入力引数で、*out_arg1* は最初の出力引数です。関数呼び出しでのパラメータ名は、サーバコード中の名前と一致しなくてもかまいません。重要なのは位置だけです。

最後の関数呼び出しの結果を出力するには、show と入力します。RPCtest シェルを終了するには、exit と入力します。

関数呼び出しの詳細については、『リファレンス』の「Nextra ユーティリティ」の章の rpctest を参照してください。

サーバの修正

予測しない値を関数が返した場合は、コードを調べる必要があります。テストユーティリティを終了して、コードを修正し、再度テストしてください。

時間を節約するために、サーバの IDL ファイルを変更しないかぎり、サーバ・スケルトンを生成しなおす必要はないという点に注意してください。つまり、サーバコードを変更しても、関数名、戻り値の型、パラメータの数、データ型、順序を変更しないかぎり、サーバ・スケルトンを再生成する必要がないということです。このいずれかを変更した場合は、IDL ファイルも変更する必要があります。この場合を除き、サーバの実行ファイルを再コンパイルするだけでかまいません。

Perl では、修正前のサーバを停止し、新しいサーバを起動するだけです。サーバ・スケルトンは自動的に新しいサーバコードにインクルードされます (サーバコードファイルの名前は変更しないものとします)。

最後に:サーバの実行

サーバのテストが完全に終わったら、サーバプログラムと関連する環境ファイルを、実際にサーバが実行されるプラットフォームに移動します。

第 7 章 DBアクセス・サーバ

この章では、DB アクセス・サーバの機能を紹介し、DB アクセス・サーバ作成の開発プロセスについてまとめます。サーバ開発のステップ毎の詳しい説明は後の章にあります。

なお、ANSI-SQL (Structured Query Language)を使った DB アクセスルーチンのコーディングについての知識が必要です。

DBアクセス・サーバとは

DB アクセス・サーバを使うと、RDBMS にアクセスし、操作するサーバを構築することが容易になります。Nextra 準拠の形式で、SQL ステートメントをコーディングするだけですみます。DB アクセス・サーバはバイナリとして提供され、DB 層との必要な機能を提供します。

DB アクセス・サーバは、次の RDBMS をサポートしています。

Oracle、Microsoft SQL Server、DB/2、HiRDB

コンポーネント

DB アクセス・サーバは、次のコンポーネントから構成されます。

- SQLMake
- dbcommon.h
- RDBMS 固有のサーバ起動ユーティリティ

SQLMake はグラフィカルユーティリティです。**SQLMake** を呼び出して、Nextra 準拠の形式で記述された SQL ステートメントの入力ファイルに基づいて自動的にインタフェース定義を生成します。**SQLMake** は、コマンドラインから直接呼び出すこともできます。これらのユーティリティのそれぞれは、**SQLMake** を使用してクライアント・スタブを生成します。

ヘッダファイル `dbcommon.h` には、DB サーバが返す全てのエラーコードの定義が含まれています。

作成するコードを最小にするために、RDBMS とのインタフェースに関する全ての機能が集約されています。この機能を持つ RDBMS 固有の起動ユーティリティを使って、**SQLMake** で生成されたクライアント・スタブを持つクライアントから DB アクセス・サーバを使用します。RDBMS に応じて、起動ユーティリティは次のいずれかになります。

ora_start*xx*(***xx*** は Oracle のバージョン)、 **sql_start**(SQL Server)、**db2_start**(DB2)、hirdb_start(HIRDB)

本書では起動ユーティリティを *DB*_start と呼びます。

使用方法

図 7.1 に、DB アクセス・サーバの使用モデルを示します。DB アクセス・サーバを使って RDBMS との 3 層コネクティビティをインプリメントするには、SQL ステートメント(一部修正し、Nextra 準拠のフォーマットにする必要あり)を作成し、そのコードを **SQLMake** に渡します。**SQLMake** は IDL ファイルとクライアント・スタブを生成します。RDBMS 固有の *DB*_start を起動するときに、SQL ステートメントファイルを渡すことによって、DB アクセス・サーバを起動できます。

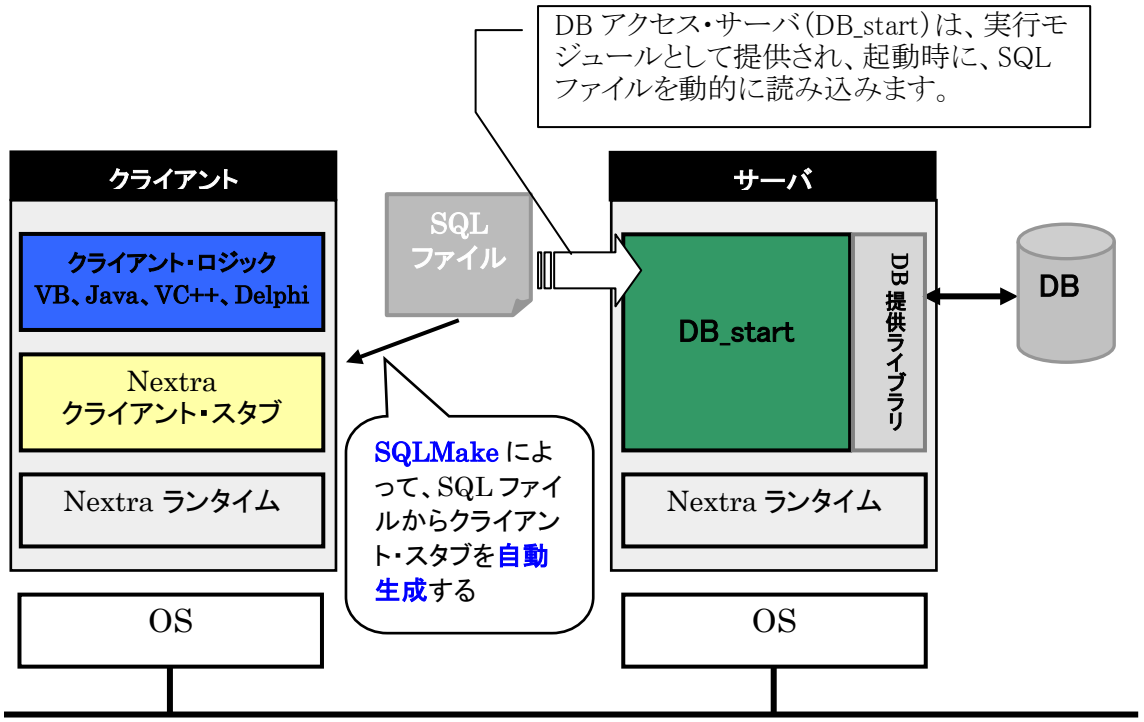


図 7.1 ： DB アクセス・サーバの使用

始める前に

プラットフォームの必要条件

Nextra 開発パッケージまたは運用パッケージを、DB アクセス・サーバが実行される全てのマシンにインストールしておくことが必要です。特に、RDBMS 固有の起動ユーティリティ(UNIX プラットフォームでは$ODEDIR/bin にあります)が必要です。
さらに、アクセスする RDBMS とは別のマシン上で DB サーバを実行する場合は、RDBMS 固有のネットワークソフトウェアを両方のシステムにインストールして、DB アクセス・サーバが DB に接続できるようにしなければなりません。

環境変数の設定

Nextra ツールを使用する前に、DB アクセス・サーバを開発、およびインプリメントする各プラットフォーム上で、環境変数を設定してください。環境変数の詳細については、「環境変数の設定」を参照してください。

RDBMS環境変数

全ての RDBMS に対して、PATH 変数が実行可能なユーティリティがあるディレクトリを指しているかどうかを確認してください(通常は *home_directory*/bin)。

小数位の数を、浮動小数点のカラムから返されるデータに設定するには、PRECISION 環境変数を使用します。小数点の右に表示する位置の値を設定します。PRECISION を設定しない場合、またはゼロ(0)を設定した場合、デフォルトは 6 小数位になります。

PRECISION 変数に 'e' を追加すると、全ての値について指数表記を指定することができます。PRECISION 変数に 'g' を指定すると、非常に大きい、または非常に小さい値についてのみ、指数表記を指定することができます。

表 7.1 : PRECISON を用いたときの値の例

PRECISION=5	PRECISION=4	PRECISION=4e	PRECISION=4g
7648.65431	7648.6543	7.6486e3	7648.6543
4.12457	4.1246	4.1246e0	4.1246
675391.00000	675391.0000	6.7539e5	6.7539e5
.02100	.0210	2.1000e-2	.0210

RDBMS固有の環境変数

SQL Server、Oracle の場合:

- DB_LOGIN を DB ユーザ名に設定する。
- DB_PWD を DB ユーザのパスワードに設定する。

別のユーザとパスワードを使用する場合は、別のセッションで sql_start、ora_start を実行する必要があります。

Oracle についてのみ、DB_PWD と DB_LOGIN の別の組み合わせを指定できます。たとえば、スラッシュ(/)をセパレータとして、ユーザ名とパスワードの両方を指定できます。

```
DB_LOGIN=user_name/password
DB_PWD=
```

或いは、sqlnet がコンフィギュレーション済みで、リモート DB にアクセスする場合は、次のように入力します。

```
DB_LOGIN=user_name/password@database_alias
DB_PWD=
```

また、設定する必要がある RDBMS 固有の環境変数を表 7.2 に示します。

表 7.2： RDBMS 固有の変数

RDBMS	データベース固有の変数
SQL Server	DSQUERY=*name_of_remote_SQL_server* PATH に *SQL_Server_direcory*\DLL を含めます。
DB2	DB2INSTANCE=*user_name_of_RDBMS_owner*
Oracle	ORACLE_SID=*database* ORACLE_HOME=*dir*

デディケイテッド・サーバの環境変数

Oracle では、デディケイテッド・サーバを使用するために、環境変数 DB_NOCOMMIT を設定しなければなりません。

DB_NOCOMMIT が 1 に設定されている場合、サーバは非選択 SQL ステートメントでは暗黙的なコミットを実行しません。0 に設定されている場合には、これはデフォルト値ですが、*DB*_start サーバが暗黙的なコミットを実行します。

デディケイテッド・サーバの詳細については、『リファレンス』の各 *DB*_start を参照してください。

ランタイムの概要

RDBMS 固有の起動ユーティリティが実行されているとき、必要なコマンドラインオプションを使って DB セッションを開始します。

```
> ora_start -q query_file -d db_name -e env_file [-s interface]
```

表 7.3： 必要な *DB*_start コマンドのオプション

フラグ	説明
`-q` *query_file*	SQLMake で使われた SQL ファイルの名前。
`-d` *db_name*	サーバがアクセスする DB 名。
`-e` *env_file*	環境ファイル名を示します。
`-s` *interface*	このオプションは、起動する DB アクセス・サーバのインタフェース名を示します。ただし、このオプションを使わない場合は、インタフェース名を SQL ファイルに指定するか、サーバの環境ファイル属性 DCE_SERVERNAME に設定します。

指定できる全てのオプションについては、『リファレンス』を参照してください。 起動コマンドが実行されると、次のようになります。

- まず、ユーティリティが環境ファイルを読み込みます。 環境ファイル属性 `DCE_SERVERPORT` が設定されていなければ、ユーティリティは、使用可能なポートを割り当てることを OS にリクエストします。
- 次に、ユーティリティは SQL ファイルを解析します。
- そして、デディケイテッド・サーバでない場合は、RDBMS へのログインを初期化します。
- 最後に、ユーティリティはブローカを DB アクセス・サーバとして登録して、クライアントからの要求を待ちます。

クライアントがサーバに対して関数呼び出しを行うたびに、サーバは、結果ステータスを示す特定の値を返します。値は次に示す正、または負の整数です。

値	意味
`>= 0`	関数呼び出しは成功しました。戻り値は、関数によって選択、挿入または更新された DB の行数です。
`-1000000<errnum<-1`	DB マネージャ内部で発生したエラーのために、関数呼び

	出しは失敗しました。戻り値は、関連する RDBMS 固有のエラー番号になります。エラーの意味については、RDBMS のドキュメント参照。
<= -1000000	DB マネージャ以外で発生したエラーのために、関数呼び出しは失敗しました。発生したエラーを調べるには、dce_errnum()または dce_errstr()を呼び出してください。

DB へのログインセッションは、サーバプロセスが停止されたときに終了します。この状態になると、サーバは接続をクローズして、終了します。

サポートされるSQL

DB アクセス・サーバは、ストアドプロシージャの起動を含む、全ての ANSI SQL ステートメントをサポートします。唯一の例外は、ROLLBACK がソフトウェアによって現在サポートされていないという点です。これは、DB アクセス・サーバが関数呼び出しのたびに暗黙の COMMIT を実行するためです。

サポートされるデータ型

RDBMS にアクセスするための RPC ライブラリでサポートされる SQL '92 のエントリレベルのデータ型は、character, integer, small integer, float, real および double です。これらの型に加えて、拡張レベルの Binary Large OBject (BLOB) 型が、全ての RDBMS でサポートされています。binary は BLOB データを表すために使用されます。サポートされるデータ型を、クライアント・スタブは対応する IDL データ型にマップし、*DB*_start は RDBMS 固有のデータ型にマップします。decimal と numeric のデータ型は対応する IDL 型を持たないため、サポートされません。サポートされていないデータ型は全て、文字列として扱われます。

開発プロセス

ここでは、Nextra 環境で DB アクセス・サーバをインプリメントするために必要なステップをまとめます。

新しいサーバの構築－プロセスの概要

DB アクセス・サーバごとに次のステップを実行します。

1. RPC に変換される ANSI SQL ステートメントを作成します。

	**デバッグ** RDBMS にテキストをパイプで入力し、結果をファイルにリダイレクトすることによって、SQL シンタックスが正しいかどうかテストしてください。出力ファイルを見て、それぞれの SQL ステートメントが成功したかどうか確認してください。

2. SQL ステートメントを Nextra 準拠の形式に変換します。

3. COBOL スタブを生成する場合は、COBOL サイズの情報を追加します。

4. SQLMake ユーティリティを使って、IDL ファイルとクライアント・スタブを生成します。

   RDBMS 固有の起動ユーティリティでは、既に全てのサーバ・スケルトン機能を組み込んでいるので、サーバ・スケルトンを生成する必要はありません。

5. 環境ファイルを作成します。

   これで、DB アクセス・サーバを起動できるようになります。

	**デバッグ** 分散アプリケーションに組み込む前に、新しいサーバをテストしてください。 DB アクセス・サーバをテストしたら、DB サーバが実行されるプラットフォームに、環境ファイルを移してください。

第 8 章 DBアクセス・サーバの構築

この章では、DB アクセス・サーバの構築とデバッグの固有プロセスについて説明します。プロセスの概要については、「開発プロセスの概要」を参照してください。

既に説明した概念をベースに説明しますので、この章を読む前、または DB アクセス・サーバを構築する前に、前の章をお読みになることをお勧めします。また、ANSI-SQL (Structured Query Language)を使った DB アクセスルーチンのコーディングについての知識も必要です。

DBアクセス・サーバの構築

Nextra ツールを使うと、新しい SQL ステートメントをサーバに非常に効率よく組み込むことができます。

SQLステートメントの作成

RDBMS 固有のシンタックスを使って、SQL ステートメントをファイルに入力してください。Oracle ファイルの例を示します。

```
select
      base_cost,
      indust_strg
from
      tools
where
      tool_name='WonderTool';

insert
      into tools
values
       ('WonderTool',500,8);


delete from
      tools
where
      tool_name = 'WonderTool';
```

SQL ステートメントの順序に対する決まりごとはありません。デリミタ(;)ごとに 1 つのステートメントだけが許されます。RDBMS によっては、異なるデリミタを使用するものがあります。使用する特定の RDBMS の詳細については、そのドキュメンテーションを参照してください。

SQLステートメントのローカルテスト

全ての SQL ステートメントをファイルに入力したら、RDBMS 固有のインタラクティブ SQL ユーティリティを使って、ファイルを RDBMS にパイプ入力することによって、シンタックスをテストします。

```
> cat test_file | DB_access_method > resultfile
```

意味は次のとおりです。

表 8.1： シンタックスをテストする SQL ファイル

test_file	SQL ステートメントを含むファイル名。
DB_access_method	RDBMS 固有のインタラクティブアクセスプログラムを起動する文字列 (つまり、 Oracle では sqlplus)です。必要に応じて、引数(ユーザ名、パスワード、および／または DB 名)を渡します。
resultfile	DB マネージャが SQL ステートメントを実行した後の出力を受け取るファイル名です。

このステップは省略しないでください

SQL ステートメントがインタラクティブなユーティリティを使用した入力に対して有効ではない場合、SQL ステートメントはサーバコードとしては機能しません。作業を続ける前に、DB に対して SQL ステートメントをテストする必要があります。

注意:DB と OS のための正確なシンタックスは多少異なることがあります。考え方としては、SQL ステートメントを DB にパイプして、出力をファイル、プリンタ、画面にリダイレクトすることになります。このプランは、どの方法で実行してもかまいません。 Windows の場合の正しいシンタックスの例については、開発パッケージにバンドルされているサンプルを参照してください。

別の方法としては、SQL ファイルを実行ファイルにして(UNIX 環境では `chmod +x` *filename*)、インタラクティブ SQL ユーティリティーを起動するためのコマンドを先頭に追加する方法があります。その例を示します。

[Oracle では]

```
pathname/sqlplus user/passwd <
statements
!
```

[MS SQL Server では]
次の行を含むファイルを作成する

```
use db_name
go
statement
go

...
```

インタラクティブ SQL ユーティリティを次のように起動する。

```
> pathname\isql - U user -P password -S server <<filename>
!
```

この方法で、出力をファイルにリダイレクトするには、次のコマンドを実行します。

```
> test_file > resultfile
```

いずれの場合でも、テストの実行を終えたら、エラーメッセージの出力ファイルを調べ、予測どおりのデータが返されたかどうか確認してください。エラーや、誤ったデータがある場合は、SQL ステートメントを編集して、テストを再実行します。出力ファイルの内容が満足できるものであれば、開発プロセスの次のステップに進むことができます。

SQLステートメントの修正

SQL ステートメントのシンタックスをテストしたら、次のステップは個々の SQL ステートメントを Nextra 準拠の形式に変更することです。それには、次のステップが必要です。

1. 個々の SQL ステートメントに、関数ラベル（およびオプション関数）の接頭語を付けます。

関数ラベルは、クライアントコードが名前で個々の照会を呼び出せるようにするために必要です。関数ラベルの形式は次のとおりです。

name:

ここで、*name* は、アプリケーションで使用可能な全ての関数名の中でユニークな関数名です。

2. リテラル文字列について記述引数変数を置き換えます。

リテラル値が入る場所に、ドル記号$を付けてフラグを立てた変数名を挿入することによって、変数を置き換えます。データ型属性を使用していない場合には、文字列リテラルの変数の置き換えにはクォーテーション記号が必要です。キャラクタフィールドの場合は、クォーテーション中を変数名に置き換えるだけです。数字フィールドの場合、クォーテーションは不要です。

3. 各変数のデータ型を示す属性文字列を挿入します。

SQL '92 のエントリレベルのデータ型は、numeric 型と decimal 型以外は全てサポートされています。この 2 つの型には相当する IDL 型がないので、IDL ファイルジェネレータによって文字列にマップされます。変数にデータ型が指定されていない場合、この変数は文字列(char)とみなされます。有効なデータ型、およびこれに対応する大文字、小文字の区別なしの属性文字を以下に示します。

データ型	属性文字列
character	char, character
integer	int, integer
small integer	smallint, small int, small integer
real	real
float	float
double	double
binary	bin, binary, BLOB

データ型を示すには、次の形式を使用してください。

入力	出力
input[*datatype*]	[*datatype*] *output*

ここで、*input* と *output* は変数名です。*datatype* は上記のリスト内のサポートされるデータ型のうちの 1 つです。

属性句とその形式の詳細については、『リファレンス』の「SQL ファイル」中の記述を参照してください。

4. **(オプション) 実行中に SQL ステートメントを動的に入力したい場合は、adhoc 関数を追加します。**

関数名はユニークでなければならないので、アプリケーションが adhoc 関数を含む 2 つの DB アクセス・サーバを含む場合は、別々の名前(たとえば、 adhoc1 と adhoc2)を付ける必要がある点に注意してください。

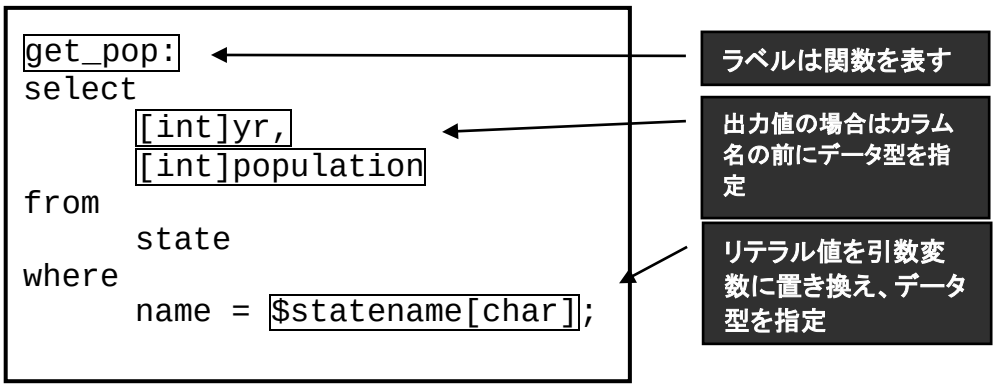


図 8.1 : 関数ラベル、引数変数、データ型指定の例

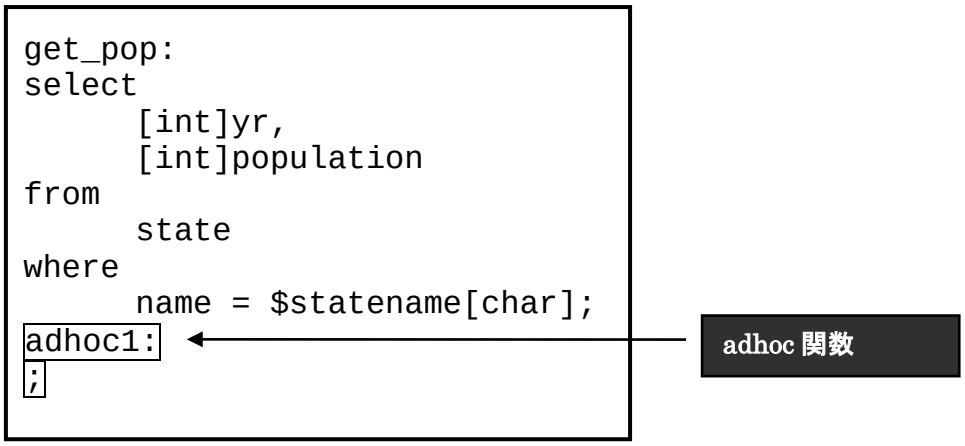


図 8.2 : SQL ファイルの adhoc 関数

全ての SQL ステートメントを Nextra 準拠の形式に変換したら、SQL ファイルはシンタックスの上では完成したことになります。いくつかの異なるサーバが動作する場合は、内容によって区別できるようにサーバファイルに名前を付けなければなりません。SQL ファイルについては、*database_name*.sql、または *server_name*.sql と命名するとよいでしょう。

使用可能な関数属性については、『リファレンス』の「ファイル仕様」の章を参照してください。

警告

DB アクセス・サーバがアクセスできる DB は 1 つだけです。つまり、1 つの RDBMS の下で 2 つの異なる DB にアクセスする必要がある場合は、DB アクセス・サーバを 2 つ起動しなければならないことになります。

COBOLサイズ情報の挿入

DB アクセス・サーバが COBOL クライアントによってアクセスされる場合、全ての NULL ターミネーテッド配列変数のサイズ属性が、SQL ファイル内に指定されなければなりません。

関数で使用される変数は全て、そのサイズを指定するためにステートメントファイル内に対応する 1 行が必要です。この行は次の形式を持っています。

```
#COBOL [output type]variable  size1 [size2]
#COBOL variable [input type]  size1 [size2]
```

ここで、*variable* は変数の名前です。変数の種類によって、*output type* または *input type* のいずれかが含まれます。これらは変数のデータ型を示します。*size1* と *size2* は整数です。binary 型の変数（BLOB）は静的割り当てに使用できないため、COBOL では使用することはできません。単純なデータ型（int、float など）は固定サイズであるため、*size1* だけを設定する必要があります。これは配列が格納することができる値の最大数を設定します。文字列では、両方の値を設定する必要があります。通常、*size1* は変数内に格納されている値を持つデータ列と同じサイズに、また *size2* は返したい行の最大数にします。 size2 の値は、デフォルトの 200 か'rcmax'に設定された値です。

たとえば、図 8.1 の get_pop 照会を COBOL 用に直した SQL ステートメントを以下に示します。

```
get_pop:
#cobol yr 10
#cobol population 10
#cobol statename 11
select
  [int]yr,
  [int]population
from
  state
where
  name = $statename[char];
```

yr と population は最大 10 個の整数の配列です。statename は最大 10 バイトの文字列の配列です。11 と指定したのは、NULL 用の 1 バイトをプラスしたからです。

スタブとIDLファイルの生成

変数名

IDL(定義)ファイルを生成するために、SQL ファイル内の select 文を解析しているときに、SQLMake は、文の中の select-list 項目ごとに最も直感的な関数引数名を使用しようとします。式中の全ての有効な文字列トークンを連結し、それぞれの間にアンダースコアを挿入し、それを名前として使用します。これが 前の select-list 式と矛盾する場合、ユーティリティは文字列_xxx を付加します。(xxx は、項目の select 文の中の位置に対応する数字の文字列) エレメントにエイリアスが指定されている場合には、ユーティリティは最初にそれを使用しようとします。ここで、DB アクセス・サーバが実行する全ての RPC の定義を含む IDL ファイルを生成する必要があります。次に、IDL ファイルを使って、クライアント・スタブを生成してください。DB アクセス・サーバはサーバ・スケルトンを必要としません。SQLMake のグラフィカルインタフェースを使用して、IDL ファイルを生成します。 図 8.3 に、QLMake GUI の例を示します。

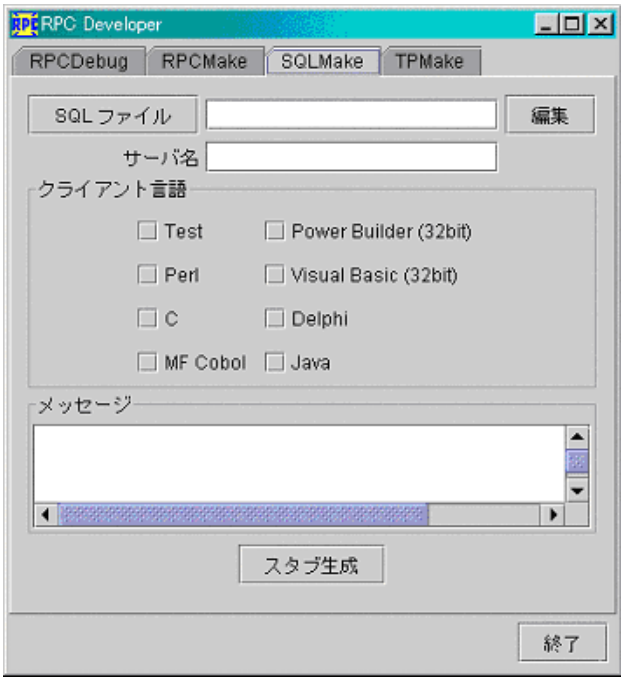


図 8.3： SQLMake の GUI 画面

IDL ファイルと 1 つ以上のクライアント・スタブを生成するには、次のステップを実行します。

1. 「SQL ファイル」フィールドに SQL ファイルの名前を入力します。

2. 「サーバ名」フィールドに、インタフェース(サーバ)名を入力します。

   このフィールドは IDL ファイル名の接頭語も決定します。バリアブル・ネームド・サーバ名を入力する場合は、ドル記号$を前に付けるだけです。GUI では、クォーテーションの指定は不要です。

3. 「クライアント言語」の下のボタンを 1 つ以上クリックして、クライアント・スタブ言語を選択します。

   この時点では、クライアント・スタブの生成は必要ではないのですが、ここで生成しないと、後で生成することになります。

4. 「スタブ生成」ボタンをクリックして、ファイルを作成します。

5. 「終了」ボタンをクリックして、終了します。

インタフェースオプション

「SQL ファイル」ボタンをクリックすると、「SQL ファイル選択ウィンドウ」ダイアログボックスが表示されます。これを使って、ファイルを選択できます。

「編集」ボタンをクリックすると、スクリーンエディタが表示されます。これを使うと、SQLMake ユーティリティを終了せずに、テキストファイルを編集することができます

出力ファイル

図 8.4 に示すように、SQLMake は IDL ファイルと、指定したクライアント・スタブを生成します。

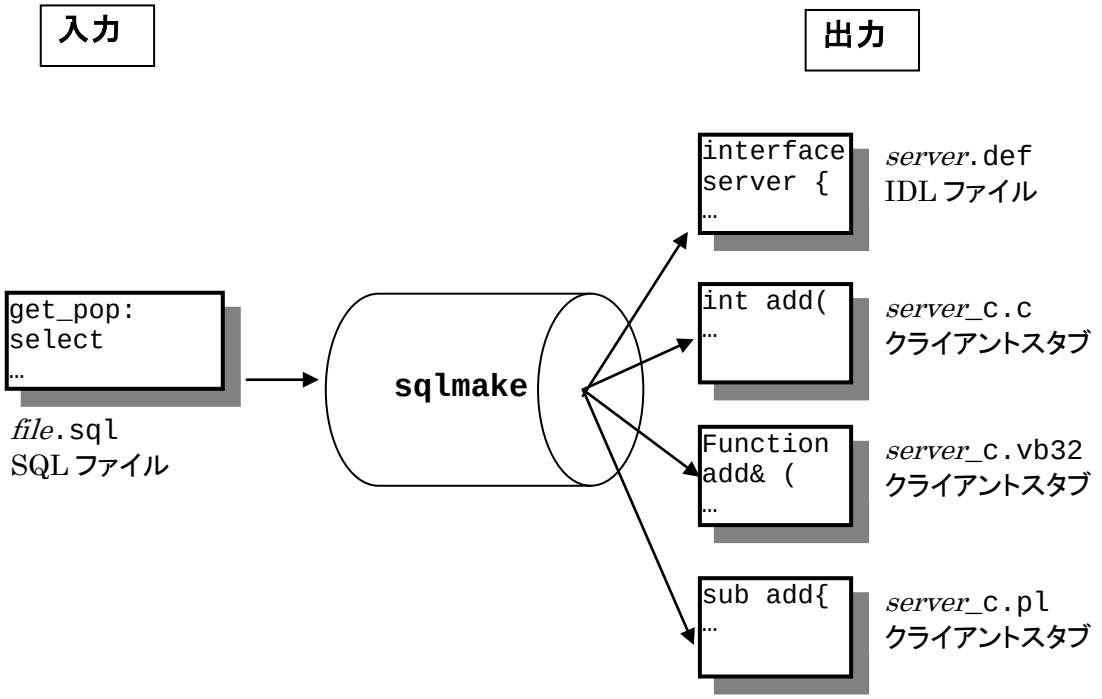


図 8.4： SQLMake の関連ファイル

SQLMakeの起動

コマンドラインから、スタブや IDL ファイルを生成するには、以下のシンタックスを使用します。

> sqlmake [-q *query_file*] [-s ['$]*server_name*[']] [-c *lang1*] [-c *lang2* ...] [-y] [-h | -help] [-v]

意味は次のとおりです。

表 8.2： SQLMake コマンドのオプション

query_file	Nextra 準拠の SQL ファイル名。
server_name	オプション。SQL ファイルにインタフェース名を指定しなかった場合は、ここで指定しなければいけません。*server_name* は 25 文字を超えてはいけません。 1 つ以上の異なるインタフェース名を SQL ファイルに指定する場合、コマンドラインオプションは接頭語をとります。可変名を使用する場合は、$記号で始めて、引数をシングルクォートで囲みます。 複数の同一サーバの詳細については、『運用／設定ガイド』の「バリアブル・ネームド・サーバの概要」を参照してください。
lang	クライアント・スタブを生成する言語を示す省略形。SQLMake は、

	RPCMake を呼び出してスタブを生成するが、言語を -c フラグで渡し、生成された IDL ファイルを RPCMake ユーティリティに渡します。同時に複数のクライアント・スタブ言語を指定できます。このフラグはオプションです。詳細については、『リファレンス』の「Nextra ユーティリティ」の章の RPCMake を参照してください。
-y	オプション。既存の IDL ファイルやスタブが、新しい IDL ファイルまたはスタブに上書きされるときに通常行われる確認のプロンプト表示を行いません。GUI を呼び出した場合は、プロンプトは表示されません。
-h, -help	ヘルプメッセージを表示してユーティリティを終了します。
-v	ユーティリティのバージョン番号を表示して終了します。

SQLMake は、IDL ファイルを *server_name*.def とします。クライアント・スタブは *server_name*_c.*lang* になります。ここで、*lang* は選択した言語です。

環境ファイルの作成

「環境ファイルの作成」を参照して、環境ファイルを作成してください。

DBアクセス・サーバのテスト

分散アプリケーションにインプリメントする前に、DB アクセス・サーバをテストしてください。「サーバのテスト」で説明したデバッグ手法を使用してください。

1. **「サーバのテスト」の説明に沿って、Perl クライアント・スタブを生成して、ブローカを起動します。**

2. **DB アクセス・スサーバを起動します。**

   起動するコマンドは次のようになります。

   ```
   > DB_start -q query_file -d db_name -e env_file -s
   server_name
   ```

   DB アクセス・サーバの起動オプションの詳細については『リファレンス』を参照してください。

3. **「サーバのテスト」の手順に沿って、RPCDebug を起動します。**

各関数を順に選択して、必要な入力値を与えてください。「実行」ボタンをクリックして、返された値が予測どおりかどうか調べてください。

adhoc 関数をインクルードしている場合は、入力として、完全な ANSI-SQL ステートメントを与えます。「実行」ボタンをクリックして、結果を調べてください (adhoc は SQL ステートメントを直接 RDBMS に送るので、エラーが発生した場合は、SQL ステートメントに誤りがあったことになります)。

サーバコードの配布

DB アクセス・サーバが実行される マシンに SQL ファイルを移します。クライアント・スタブがこの時点で生成されていれば、クライアントインタフェースが置かれるプラットフォームにクライアント・スタブを移します。クライアント・スタブが生成されていない場合は、クライアント構築プロセスについて、『クライアント開発者ガイド』を参照してください。

特殊機能の使用

ここでは、次の 4 つの追加機能について説明します。

- 非デフォルト(任意)の DB のアクセス
- 複数 DB にアクセスするデディケイテッド・サーバ
- ストアド・プロシージャ
- DB カーソル (デディケイテッド・サーバのみ)

非デフォルト(任意)DBのアクセス

*DB*_start は、デフォルト DB(環境変数で指定可能)だけではなく、適切な RDBMS の下で実行されているあらゆる DB に接続できます。

各 *DB*_start によってアクセスできる DB についての制限はありません。たとえば、デフォルトとして指定されていない Oracle データベースにアクセスしたり、デフォルトとして指定された以外の SQL サーバの下で実行されている SQL Server データベースにアクセスできます。各 DB エンジンの「デフォルト」は次のようになります。

Oracle	環境変数 ORACLE_SID でデフォルト DB を指定します。
MS SQL Server	デフォルトはローカル SQL になります。DSQUERY 環境変数が

	設定されていれば、優先して使用されます。
DB2	環境変数 DB2INSTANCE でデフォルト DB を指定します

別の DB にアクセスするには、別のインスタンスを設定する必要があります。

各 RDBMS 固有の情報については、以下の説明を参照してください。

Oracle

デフォルト DB 以外の DB に接続するには、関連する全てのマシンは(1 つのマシンしかない場合でも)、Oracle SQL-Net を実行していることが必要です。また、tnsnames.ora ファイルで指定される、DB 名と SQL-Net エイリアスは同じではなくてもかまいません。

たとえば、デフォルト DB を *massachusetts* に指定したとします。

```
ORACLE_SID=massachusetts
```

デフォルト DB で DB アクセス・サーバを起動するには、次のように入力するだけです。

> ora_start -e *data.env* -s *oraserver* -d *massachusetts* -q *query_file*

別の DB を使用して起動するには、次のように入力してください。

> ora_start -e *data.env* -s *oraserver* -d *alias* -q *query_file*

ora_start は tnsnames.ora ファイルを調べて、*alias* を参照し、そのエイリアスに対して指定された DB にアクセスします。

Windows上のSQL Server

デフォルト(通常はローカル)サーバ以外の SQL Server にアクセスするには、環境変数 DSQUERY を設定する必要があります。

この SQL Server は、SQL Server プログラムグループにある SQL クライアント設定ユーティリティを使用して定義しなければなりません。SQL クライアント設定ダイアログボックスで、次のステップに従ってください。

1. 「Advanced」ボタンをクリックします。

   「Advanced」ダイアログボックスが表示されます。

2. 最初のフィールドに、サーバを識別するエイリアスを入力します。

   定義された SQL サーバの間でユニークであれば、どの名前でもかまいません。

3. 2 番目のフィールドでは、DLL を常に dbmssocn にしなければなりません。

4. 3 番目のフィールドに、指定したエイリアス名と関連づける SQL Server のホスト名とポートを入力します。

   正しいシンタックスは hostname,portnum です。

5. 変更を保存して、ユーティリティを終了します。

   DB アクセス・サーバを起動するときは、(-d オプションを使用して)DB を指定しなければなりません。この DB は、環境変数 DSQUERY で指定した SQL Server で使用可能なものです。

たとえば、デフォルト SQL Server の下で実行されるデータベース bigdb にアクセスするサーバを起動する場合は、次のようになります。

```
> sql_start -e data.env -s server_name -d bigdb -q query_file
```

別の SQL サーバの下で実行される DB にアクセスするには、そのサーバ名に対して DSQUERY を設定して、同じような起動コマンドを入力します。

```
> sql_start -e data.env -s server_name -d tech -q query_file
```

各デディケイテッド・サーバの子プロセスによる別のDBへのアクセス

*DB*_start のデディケイテッド・サーバは、サーバの各デディケイテッド・サーバの子プロセスを使って、別の DB に接続することができます。

デディケイテッド・サーバを実行している場合、各クライアントは接続する DB を、sql_prepare_*interface*() RPC に指定することができます。次の SQL ステートメントは、dbserver という同じデディケイテッド・サーバに接続する別のクライアントのもので、別の DB に接続するデディケイテッド・サーバの子プロセスを要求しています。

```
sql_prepare_dbserver("purpledb", "beth", "v98S34tJ");
sql_ prepare_dbserver("reddb", "marco", "eWr8uin348");
```

ストアドプロシージャの使用

Nextra ツールキットは、Oracle のためのストアドプロシージャの呼び出しをサポートします。

Oracle では、ストアドプロシージャからの出力は返されません。

DB アクセス・サーバが実行する他の関数と同じように、ストアドプロシージャを呼び出す関数は、SQL ファイル中で定義されていなければなりません。

たとえば、次のストアドプロシージャを Oracle で定義するものとします。

```
  create procedure listname (person_age in integer, in
raise real) as
  BEGIN
  update namelist
  set salary=salary*raise where age > person_age;
  END listename;
```

次の方法で SQL ファイル中のプロシージャを呼び出すことができます。

```
update_namelist:
        {BEGIN listname($a, $b)\; END\; \;};
```

参照:『リファレンス』の「SQL ファイル」の章

デディケイテッド・サーバにおけるカーソルの使用

デディケイテッド・サーバでは、カーソルを使用することができます。環境ファイル属性 DCE_DEDICATED が 0 より大きな値に設定されている状態で、*DB*_start ユーティリティを起動した場合、接続するクライアントごとにコピーが作成され、各サーバは 1 つのクライアントのために RPC を実行することになります。通常のサーバと比べて、この種のサーバは「ステータス」を維持しているといえます。つまり、クライアント要求と対応する応答を適切に記録できるということです。DB アクセス・サーバでカーソルを使用することは、この機能を活用する 1 つの方法です。

カーソルとアドバンスドDBアクセス・サーバ

SQLMake が生成した IDL ファイルには、デディケイテッド・サーバが使う 2 つの関数、sql_prepare_<*interface*>()と sql_set_max_rows_<*interface*>()が含まれます。この関数はデディケイテッド・サーバでのみ使用されるもので、通常の DB アクセス・サーバから呼び出されることはありません。クライアントが 1 つ以上のデディケイテッド・サーバにアクセスする

場合に、別のサーバの同じような関数と区別するために、関数名にはインタフェース名が付けられます。

デディケイテッド・サーバでのセッションの開始

デディケイテッド・サーバでは、2 つの異なった方法で DB にログインできます。それは、暗示的(デフォルト)、または明示的 (オーバーライド)な方法です。

デフォルトの DB のログインとパスワードは、環境変数の DB_LOGIN と DB_PWD で与えられます。sql_prepare_*interface*()が、デディケイテッド・サーバが受け取る最初の RPC でない場合、デディケイテッド・サーバは通常のサーバと同じように、コマンドラインの-d オプションで指定した DB 名で、DB_LOGIN と DB_PWD の組み合わせを使って、DB へのログインを初期化します。環境変数が設定されていなければ、ログインとパスワードは NULL になります。ログインに失敗すると、デディケイテッド・サーバは DB と接続できずに停止してしまいます。

このサーバに対する最初の RPC として sql_prepare_<*interface*>()を呼び出すことによって、このデフォルトをオーバーライドすることができます。関数は、アクセスする DB、ユーザ名、およびパスワードの 3 つの引数をとります。その C のプロトタイプを次に示します。

long sql_prepare_<*interface*> (char * *dbname*, char * *login*, char * *passwd*)

この関数呼び出しを受け取ると、指定された名前とパスワードを使って、サーバは特定の DB へのログインを初期化します。関数が正常終了した場合、この関数をさらに呼び出すと、エラー (DBALREADYCONN: 関数が 2 回呼び出された)が返されます。最初の呼び出しに失敗すると、この関数の次の呼び出しをサーバは受け付けます。戻り値 DCPSUCCESS によって sql_prepare_<*interface*>() が正常終了するまで、サーバは他の RPC を受け付けません(エラーコード DBUSELOGINRPC: sql_prepare 失敗)。

この関数の詳細については、『リファレンス』を参照してください。

カーソル値の設定

次に、カーソル機能を使うために、クライアントは関数 sql_set_max_rows_<*interface*>()を呼び出します。

この関数は、1 回の照会ごとに検索する行の最大数という引数だけを使用します。この関数は、*maxrows* が 0 か正の値で、関数が正常終了した場合に、値 DCPSUCCESS を返します。*maxrows* が無効(負)の場合は、関数は DBINVALIDPARAM を返します。関数が別の理由でエラーとなった場合は、対応するエラーコードが返されます。

たとえば、照会でトータル 500 行を選択し、値 50 で sql_set_max_rows_<*interface*>()を呼び出すと、照会が最初に行われたときの先頭の 50 行が返されます。その他の行は、子プロセス内でキャッシュされます。同じ照会を続けて行うと、次の 50 行が返されます。全ての行が返されると、キャッシュはリフレッシュされます。

この関数の詳細については『リファレンス』を参照してください。

カーソルがデディケイテッド・サーバでのみサポートされる理由

同じサーバに対する同じ照会を別のクライアントが行うことは避けられないため、通常のサーバではカーソルは使用できません。同じ照会を行うということは、オリジナルのクライアントが次に照会を行うときに、カーソルが 50 行先に設定されるということです。あるいは、別のクライアントが異なった照会を行って、最初のクライアントが全体を検索し終える前に、カーソルをゼロに戻してしまう可能性もあります。デディケイテッド・サーバがサービスするのは 1 つのクライアントだけなので、別のクライアントが照会を妨げることはありません。

sql_set_max_rows_<*interface*>()を一度も呼び出さない場合、または sql_set_max_rows_<*interface*>(0)が呼び出された場合は、サーバは、最初の照会で選択された全ての行を返します。同様に、選択された行数よりも *maxrows* が大きい場合は、最初の照会で、選択された全ての行が返されます。最初の照会が選択された全ての行を返す前に、2 回目の照会が呼び出されると、2 回目の最初の *maxrows* 行が検索されて、最初の照会のカーソルはその先頭にリセットされます。一度に返される行数を変更するには、新しい値で sql_set_max_rows_<*interface*>()を呼び出す必要があります。

たとえば、2 つの RPC、*rpc1* と *rpc2* があるとします。両方とも合計で 20 行を返し、sql_set_max_rows_<*interface*>(4)が呼び出されているとします。*rpc1* に対する最初の呼び出しは行 A1-A4 を返します。*rpc1* に対する次の呼び出しは A5-A8 を返します。*rpc2* に対する呼び出しは B1-B4 を返します。rpc1 に対する別のクライアントからの呼び出しでは A1-A4 を返します。sql_set_max_rows_<*interface*>(8)を呼び出してから、*rpc1* を再度呼び出すと、A5-A12 が返されます。

注：sql_set_max_rows_<*interface*>(4)を使って *rpc1* が 6 回呼び出されると、5 回目の呼び出しは 17-20 行を返しますが、6 回目の呼び出しでは 0 が返されます。7 回目の呼び出しは子プロセス内のキャッシュをリフレッシュして、行 1-4 を返します。

全部で 19 行しかない場合は、5 回目の呼び出しは行 17-19 を返し、6 回目の呼び出しは行 1-4 を返します。

一連の照会の最後で、全選択が完了した場合、照会の次の実行では 0 行が返されます。このようになっていないと、全選択が返されたときに、行数が必ずしも返されないという状態になります。この処理では、照会が *maxrows* 行より少ない行数を返す場合、その照会では全選択が返されます。

例:

```
#include dceinc.h
/* This example is for regular data access servers */
.
..
...
main
{
char * db, * login, * passwd;
int rv;

get_info_from_user(db, login, passwd);
rv = sql_prepare_interface(db, login, passwd);
check_for_error(rv);
rv = sql_set_max_rows_interface(100);
...

..
```

カーソルとバリアブル・ネームドDBサーバ

バリアブル・ネームドインタフェースでは、全ての RPC についての特別パラメータがスタブに含まれます。このパラメータはパラメータリストの最初のパラメータになり、関数呼び出しが送られるインタフェースを指定します。

バリアブル・ネームド・サーバでは、IDL ファイル中でインタフェースは可変名になり、先頭の'$'が取り除かれたものになります。たとえば、-s '$var_name'フラグを使って SQLMake で IDL ファイルを生成した場合、カーソル関数は、sql_prepare_var_name()、および sql_set_max_rows_var_name()になります。

参照:『運用／設定ガイド』の「デディケイテッド・サーバ」
Nextra ライブラリ関数:『リファレンス』の「Nextra API」の章の sql_set_max_rows_()、sql_prepare_()

第 9 章 Javaアプリケーション・サーバの構築

この章では、Java アプリケーション・サーバの構築とデバッグの固有プロセスについて説明します。

既に説明した概念をベースに説明しますので、この章を読む前に、「第 6 章 ファンクショナリティ・サーバ」をお読みになることをお勧めします。

クライアント側の開発については、『クライアント開発者ガイド』を参照してください。

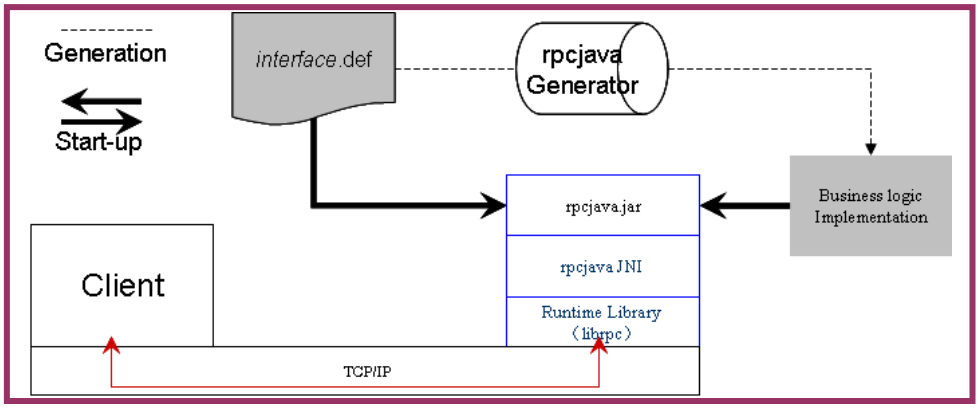
Javaアプリケーション・サーバの特長

ユーザは、IDL ファイルから生成された Java インタフェースクラスを元にビジネス・ロジックの実装を行うだけで、Java 分散アプリケーションの構築、起動ができます。

このウルトラ・ライトウエイトの Java アプリケーション・サーバの特長は

1. EJB ではないのでコンテナは不要であり、軽量で、しかも EJB に比べ最適化されています。
2. 複数マシン上に、並列にサーバプロセスを複数起動することにより、アプリケーション・クラスタリングを実現します。
3. クライアントとの間で、大量の高速通信を可能とします。
4. Java クライアントばかりでなく、.NET、VB、C、COBOL、Delhi、PB もサポートされます。

アーキテクチャ



の個所は、開発者が作成するプログラムです。
それ以外の部分は、自動生成 または Nextra により提供します。

図 9.1： Java アプリケーション・サーバ アーキテクチャ

必要条件

プラットフォームの必要条件

- Nextra ランタイムライブラリ（librpc）
- JDK1.6.x のバージョン

制限事項

デディケイテッド・サーバ（環境ファイル属性 DCE_DEDICATED にて指定）は、クラスパス以外の JVM オプションは子プロセスに引き継がれません。

開発プロセス

Java アプリケーション・サーバの作成には、主に次の7つの手順があります。

環境設定

『はじめにお読みください』の「第 2 章 インストールの方法」にある”インストール後の環境設定”に記述されている設定は既に行われているとします。

Windows

jvm.dll が存在するディレクトリを PATH へ設定。

UNIX

$ODEDIR/lib、そして jvm.[*txt*]が存在するディレクトリを共有ライブラリパスへ設定。

IDLファイルの記述

テキストエディタにて、IDL（定義）ファイル（.def）を記述してください。IDL ファイルは、フラットファイルです。IDL ファイルの記述方法は、『リファレンス』「第 2 章 ファイル仕様」にある「インタフェース定義言語（IDL）ファイル」を参照してください。

Javaアプリケーション・サーバ（rpcjava）による、Javaインタフェースクラスの生成

通常の Nextra クライアントやサーバと違い、サーバサイド Java ではスタブの生成は必要ありません。ただし、インタフェースクラスの生成が必要になります。生成されるファイル名は、*interfacename*.java になります。生成する方法は、GUI とコマンドラインの 2 通りがあります。

Nextra開発ツールを使用する方法

>rpcmake

GUI 起動後、IDL(.def)ファイル、サーバ言語で「Java」を選択して生成してください。

コマンドラインにて生成する方法

>rpcjava -h …HELP が表示されます。

```
>rpcjava -f file.def [-d directory] [-c classname] [vm
options …]
```

Javaインタフェースクラスの実装

interfacename.javaの実装を行ってください。実装時の注意点については、、「実装時の注意点」を参照してください。

環境ファイルの準備

環境ファイル属性については、『リファレンス』「第2章 ファイル仕様」の「環境ファイル属性」を参照してください。

rpcjavaの起動

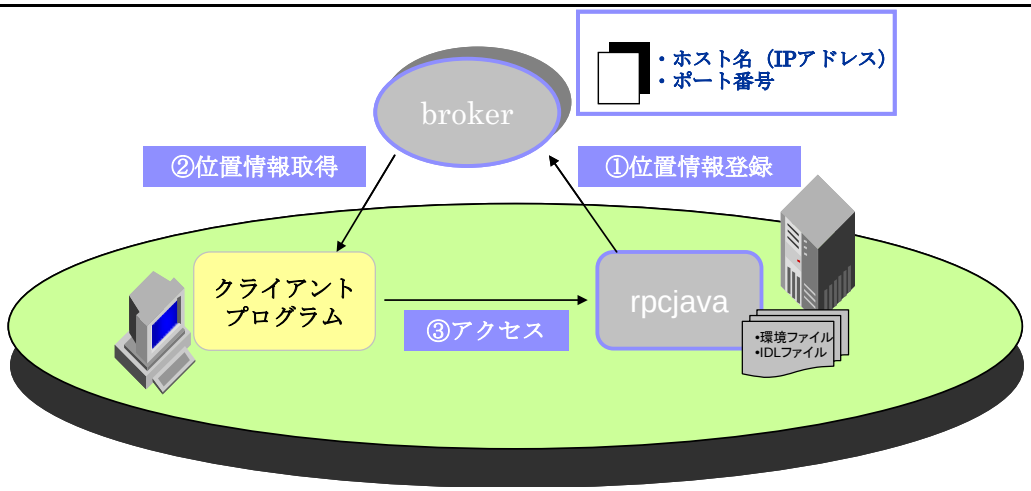


図 9.2： rpcjava 稼動イメージ

最初に broker を起動します。

>broker -e env_file

＊ broker のオプションなどの詳細については、『リファレンス』「第4章 Nextra ユーティリティ」の「broker」を参照してください。

次に、rpcjava の起動を行います。この際、環境ファイルと IDL ファイルの読み込みが必要になります。rpcjava は、起動時に、図 9.2 の「①位置情報登録」を broker に対して行います。

>rpcjava -e *env_file* -d *file*.def [-i *server implement class* | -f *server factory*] [-DMULTI=true] [server options …] [vm options …]

デディケイテッド・サーバとシステムプロパティの取得

デディケイテッド・サーバによりクライアント毎に作成される子プロセスにて、システムプロパティを使用して値を取得したい場合。

String lDelay = System.getProperty("L2U_DELAY");

この場合、L2U_DELAY を rpcjava 起動コマンドに”-DDED_”に続き以下のように指定してください。

例: > rpcjava -DDED_L2U_DELAY=10 …

この例のような指定を行い起動することにより、子プロセスにて、この例では、10 という値を取得することができます。

RPCDebug によるテスト

クライアントプログラムが作成されていなくても、RPCDebug があれば rpcjava のテストを行うことができます。RPCDebug は、テスト目的で、サービスに対するユニバーサルなクライアントとして機能します。RPCDebug は、図 9.2 の中の「クライアントプログラム」にあたります。詳しくは、『リファレンス』「第4章 Nextra ユーティリティ」の「RPCDebug」を参照してください。

実装時の注意点

Exception

パッケージにある「samples」ディレクトリ内を参照してください。インタフェースクラスの実装において、rpcjava で使用できる Exception は以下の通りになります。

Exception	説明	使用方法
RPCException	サーバ起動時に IO エラーなどが起こった場合に throw されます。	サーバ実装クラスから制御する必要はありません。
IDLFormatException	IDL ファイルのフォーマットが適切でない場合に throw されます。	サーバ実装クラスから制御する必要はありません。

ServerException	サーバ処理に失敗した場合にサーバ実装クラスから throw することができます。
	Exit フラグを true に設定した場合、サーバアプリケーションは終了します。
		try {
		//Business logic
		} (catch Exception e) {
		ServerException se = new
		ServerException(“message”, e);
		se.setExit(true);//Exit flag
		throw se;
		}

バリアブル・ネームド・サーバの記述について

実行時に”-s”で指定、または環境ファイル中の DCE_SERVERNAME 属性でサーバ名を指定することにより、起動時にサーバ名を変更できます。前提として、IDL ファイルの interface 名では「$*variable_named_sever*」と指定しなければいけません。

文字コード（Locale）について

環境ファイル属性「DCE_LOCALE」にて指定します。デフォルトは、SJIS になりますが、指定できるエンコーディングについては、以下を参照してください。

http://java.sun.com/j2se/1.4/ja/docs/ja/guide/intl/encoding.doc.html
http://java.sun.com/j2se/1.4.2/docs/guide/intl/encoding.doc.html

ログファイルへの出力

com.inspire.rpc.RPCLog クラスを使用して RPC ログにメッセージを出力することが出来ます。サーバ実装クラスからは com.inspire.rpc.server.ServerContext の getLog()を使用して RPCLog オブジェクトを取得することが出来ます。

```
import com.inspire.rpc.server.ServerContext;
import com.inspire.rpc.server.ServerException;

public BasicsImpl implements Basics {

    private ServerContext ctx;

    public void setServerContext(ServerContext ctx) {
        this.ctx = ctx;
    }

    public int add(int x, int y) throws ServerException {
        ctx.getLog().debug("add", "message");
        return x + y;
    }
}
```

AppMinderにて起動する際の「-classpath」の指定について

rpcjava を Windows 上で起動、しかも AppMinder から起動する場合、さらに「-classpath」を指定する場合には、「-classpath」以降のディレクトリはダブルクオーテーション「""」で囲む必要があります。

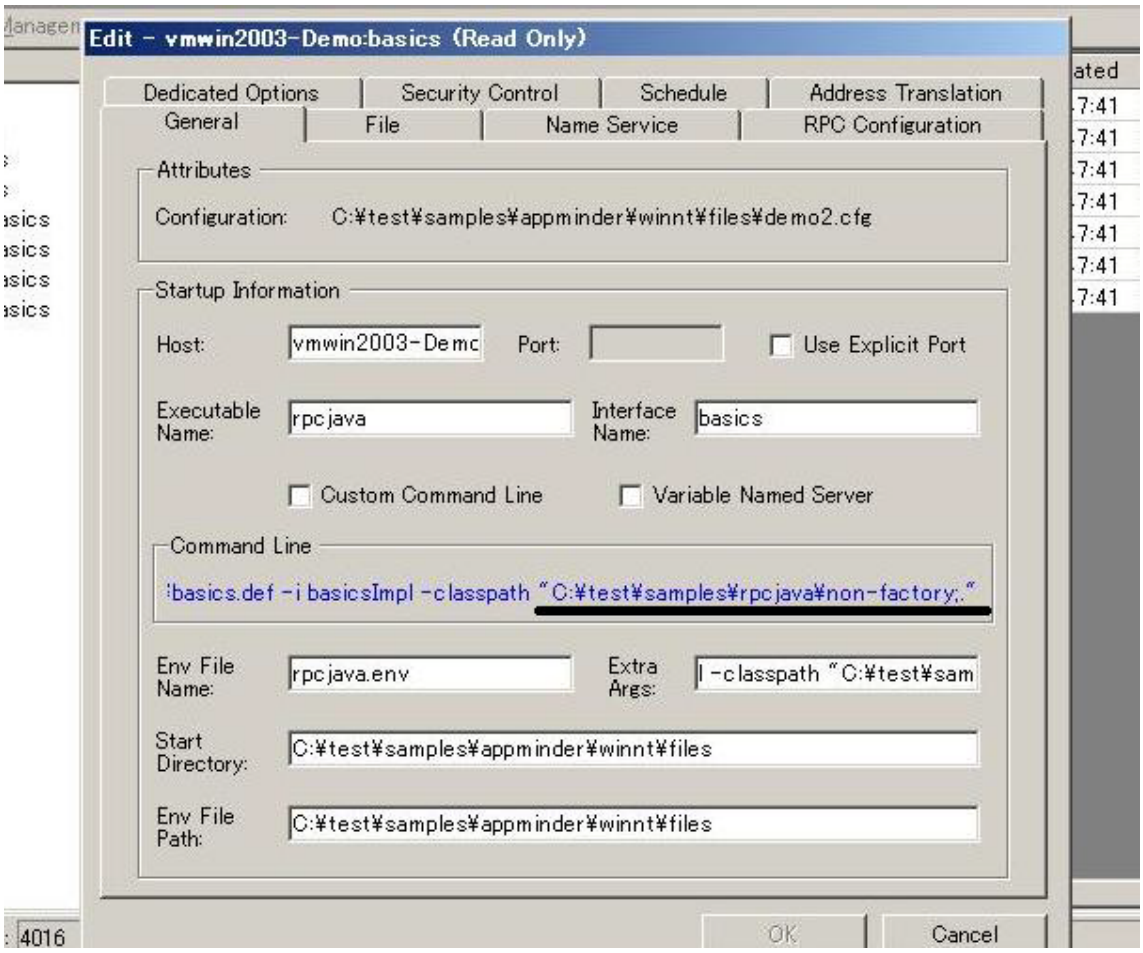


図 9.3： AmViewer(ビューア)での「-classpath」指定の例

OS環境変数への設定

DCE_LISTEN_QUEUES

各サーバ・ソケットにで保留される TCP バックログ 数を指定するための環境変数です。デフォルト値は 5 で、1 から INT_MAX まで指定可能です。更に、環境ファイル属性「DCE_LISTEN_QUEUES」としても設定可能です。

API

Nextra 製品がインストールされている「docs/api/rpcjava」ディレクトリを参照してください。

サンプルプログラム

Nextra 製品がインストールされている「samples/datatest/server/java」ディレクトリ内に、使用できる全データタイプのサンプルがあります。

データタイプマッピング

表 9.1 に、rpcjava で使用可能なシンプルデータ型および IDL ファイルで宣言するデータ型との対応を示します。使用できるデータ型別についての説明は、表 9.2 を参照してください。

表 9.1： データタイプマッピング

Java アプリケーション・サーバ内	IDL ファイルでの宣言
short	short
int	int
int	long
float	float
double	double
char	char
void	void
Object	object

表 9.2： データ型別使用例

次元	配列	型	IDL ファイル内 関数例	パラメータの扱い方については、製品 samples の中の、以下の method を参照してください。
Simple		short	short FuncShort([in] short shVar, [out] short shVarOut);	public short FuncShort(short shVar, ShortOut shVarOut)
		long	long FuncLong([in] long lVar, [out] long lVarOut);	public int FuncLong(int lVar, IntOut lVarOut)
		int	int FuncInteger([in] int nVar, [out] int nVarOut);	public int FuncInteger(int nVar, IntOut nVarOut)
		float	float FuncFloat([in] float fVar, [out] float fVarOut);	public float FuncFloat(float fVar, FloatOut fVarOut)
		double	double FuncDouble([in] double dVar, [out] double dVarOut);	public double FuncDouble(double dVar, DoubleOut dVarOut)
		char	char FuncChar([in] char cVar, [out] char cVarOut);	public char FuncChar(char cVar, CharOut cVarOut)
		Object	object FuncObj([in] object oVar, [out] object oVarOut);	public Object FuncObj(Object oVar, ObjectOut oVarOut)
1 Dimen sional	Const raine d	short	void FuncConstrainedShortArray([in] short nRowDim1, [in] short nRowDim2, [in] short shArrVar[nRowDim1], [out] short shArrVarOut[nRowDim2]);	public void FuncConstrainedShortArray(short nRowDim1, short nRowDim2, short[] shArrVar, ShortArrayOut shArrVarOut)

		long	void FuncConstrainedLongArray([in] long nRowDim1, [in] long nRowDim2, [in] long lArrVar[nRowDim1], [out] long lArrVarOut[nRowDim2]);	public void FuncConstrainedLongArray(int nRowDim1, int nRowDim2, int[] lArrVar, IntArrayOut lArrVarOut)
		int	void FuncConstrainedIntArray([in] int nRowDim1, [in] int nArrVar[nRowDim1], [in] int nRowDim2, [out] int nArrVarOut[nRowDim2]);	public void FuncConstrainedIntArray(int nRowDim1, int[] nArrVar, int nRowDim2, IntArrayOut nArrVarOut)
		float	void FuncConstrainedFloatArray([in] int nRowDim1, [in] float fArrVar[nRowDim1], [in] int nRowDim2, [out] float fArrVarOut[nRowDim2]);	public void FuncConstrainedFloatArray(int nRowDim1, float[] fArrVar, int nRowDim2, FloatArrayOut fArrVarOut)
		double	void FuncConstrainedDoubleArray([in] int nRowDim1, [in] double dArrVar[nRowDim1], [in] int nRowDim2, [out] double dArrVarOut[nRowDim2]);	public void FuncConstrainedDoubleArray(int nRowDim1, double[] dArrVar, int nRowDim2, DoubleArrayOut dArrVarOut)
		char	void FuncConstrainedCharArray([in] int nColDim1, [in] char cArrVar[nColDim1], [in] int nColDim2, [out] char cArrVarOut[nColDim2]);	public void FuncConstrainedCharArray(int nColDim1, String cArrVar, int nColDim2, StringOut cArrVarOut)
		void	void FuncConstrainedVoidArray([in] int nRowDim1, [in] void byteArrVar[nRowDim1], [out] int nRowDim2, [out] void byteArrVarOut[nRowDim2]);	public void FuncConstrainedVoidArray(int nRowDim1, byte[] byteArrVar, IntOut nRowDim2, ByteArrayOut byteArrVarOut)
	Fixed array	short	void FuncFixedLengthShortArray([in] short shArrVar[10], [out] short shArrVarOut[10]);	public void FuncFixedLengthShortArray(short[] shArrVar, ShortArrayOut shArrVarOut)
		long	void FuncFixedLengthLongArray([in] long lArrVar[10], [out] long lArrVarOut[10]);	public void FuncFixedLengthLongArray(int[] lArrVar, IntArrayOut lArrVarOut)
		int	void FuncFixedLengthIntArray([in] int nArrVar[10], [out] int nArrVarOut[10]);	public void FuncFixedLengthIntArray(int[] nArrVar, IntArrayOut nArrVarOut
		float	void FuncFixedLengthFloatArray([in] float fArrVar[10], [out] float fArrVarOut[10]);	public void FuncFixedLengthFloatArray(float[] fArrVar, FloatArrayOut fArrVarOut)
		double	void FuncFixedLengthDoubleArray([in] double dArrVar[10], [out] double dArrVarOut[10]);	public void FuncFixedLengthDoubleArray(double[] dArrVar, DoubleArrayOut dArrVarOut)
		char	void FuncFixedLengthCharArray([in] char cArrVar[10], [out] char cArrVarOut[10]);	public void FuncFixedLengthCharArray(String cArrVar, StringOut cArrVarOut)
		void	void FuncFixedLengthVoidArray([in] void byteArrVar[5973], [out] void byteArrVarOut[5973]);	public void FuncFixedLengthVoidArray(byte[] byteArrVar, ByteArrayOut byteArrVarOut)
	NT*) array	char	void FuncNullTerminatedArray([in] char cArrVar[], [out] char cArrVarOut[]);	public void FuncNullTerminatedArray(String cArrVar, StringOut cArrVarOut)
2 dimensional	Constrained array	char	void FuncConstrainedCharArray([in] int nRowDim1, [in] int nColDim1, [in] char sVar[nRowDim1][nColDim1], [in] int nRowDim2, [in] int nColDim2, [out] char sVarOut[nRowDim2][nColDim2]);	public void FuncConstrainedCharArray(int nRowDim1, int nColDim1, String[] sVar, int nRowDim2, int nColDim2, StringArrayOut sVarOut)
	Fixed array	char	void FuncFixedCharArray([in] char sVar[10][20], [out] char sVarOut[10][30]);	public void FuncFixedCharArray(String[] sVar, StringArrayOut sVarOut)
	NT*) array	char	void FuncNullTerminatedArray([in] char sVar[][], [out] char sVarOut[][]);	public void FuncNullTerminatedArray(String[] sVar, StringArrayOut sVarOut)

*) NT=Null Terminated

ご注意

商標権に関する注意

Nextra 製品は、全て Inspire International Inc. の商標または登録商標です。その他記載のブランドおよび製品名は、該当する会社の商標または登録商標です。

著作権に関する注意

Nextra
サーバ開発者ガイド

2011 年 9 月 15 日 v6 1st Edition
2008 年 10 月 15 日 v5 2nd Edition
2007 年 7 月 10 日 第 9 章 Java アプリケーション・サーバの構築
2007 年 4 月 16 日 第 3 版 2008.10.15 [-DMULTI=true] added 発行
2006 年 11 月 20 日 環境変数の設定に追加
2005 年 11 月 11 日 COBOL に関する記述の更新
2004 年 7 月 19 日 第 2 版発行
2003 年 4 月 18 日 初版

著者 Inspire International Inc.