Nextra
クライアント開発者ガイド

Version 6.5

目 次

第1章　はじめに

この章では、『クライアント開発者ガイド』の利用方法、対象読者、トピックの簡単な概要および表記規則について説明します。

本書の利用方法

本書『クライアント開発者ガイド』では、UNIX、Windows のプラットフォーム上の GUI クライアントを 3 層分散アプリケーションに組み込む方法を説明します。

対象読者

本書は、3 層分散アプリケーション用のテキストベースクライアントおよび GUI クライアントを設計、構築する開発者を対象としています。

本書では、まず、C、COBOL または Perl による UNIX 上でのクライアントの構築プロセスを説明します。

それから、PC クライアントを 3 層分散アプリケーションに組み込むプロセス、特に、PowerBuilder, Visual Basic, Visual C++, Java, VB .NET, C#, Delphi で作成されるクライアントの組み込みについて説明します。

前提知識

『サーバ開発者ガイド』で説明されている内容をご理解いただいてから本書をお読みください。

特に、Nextra で作成できるオープン分散アプリケーションを実現する 3 層構造アーキテクチャについてご理解ください。基本的な知識に加え、3 層アプリケーションのプレゼンテーション層の機能についてもご理解ください。

Nextra ツールの使用方法と機能の詳細は『リファレンス』で説明されています。

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
sub-text	ユーザが指定する必要がある値を示します。	*text*.def
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**
[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。	[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。

デバッグのヒント

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。

第2章　概要

GUI クライアントを完成するには、いくつかの手順があります。この章では、GUI クライアントの開発プロセスを示し、別の章で詳しい説明を行います。

はじめに

GUI クライアントをオープン分散環境へ組み込む前に、以下のことを検討してください。

ネットワークの必要条件

Nextra ランタイム機能は、ネットワーク内のさまざまなマシンに存在する、クライアントとサーバのプロセス間のコネクションを確立することによって動作します。このため、プラットフォームには、マシン間でメッセージをやりとりするためのネットワークハードウェアとソフトウェアの組み合わせが必要になります。

GUIの機能

GUI クライアントは、それ自身がソフトウェアの強力な一部となる可能性がありますが、アプリケーションの機能の大部分を GUI クライアントに含めるべきではないという点に注意してください。1 つのフロントエンドに含まれる操作ルーチンや計算処理が多いほど、クライアントを別の GUI 環境に移植する際に変更しなくてはならないプログラムコードが増えます。

代わりに、こういった機能は可能なかぎりサーバでコーディングするようにします。これによって、クライアントは自分自身の処理に専念することができます。

GUIクライアント開発プロセス

GUI クライアントの作成には、主に次の 4 つの手順があります。クライアントに必要なファイルの設定、クライアントのコーディング、クライアント・スタブの準備、そしてクライアント用の環境ファイルの作成です。

ここでは、クライアントのおおよその設計とローカルバグを見逃さないためのコードの初期テストを終了しているものとして説明を進めます。ここでの設計には、インタフェースのルック＆フィール、つまり、ユーザと GUI との対話、画面表示の順序、そしてそれぞれの画面の設計が含まれます。

GUIクライアントの設定

Nextra の関数を GUI クライアントが使用できるようにするために、関数を定義するファイルへのアクセス手段をクライアントに与える必要があります。

Windows GUI のクライアントは、次の 3 種類のファイルへのアクセスを必要とします。

- **ダイナミックリンクライブラリ(DLL)**
  クライアントが RPC(リモートプロシージャコール)を行えるようにします。DLL には、クライアントの環境を設定し、サーバと相互に通信してメモリの管理を処理するオープン分散環境関数が含まれています。

- **ローカル GUI API**
  DLL のインタフェース・ファイルとして機能し、GUI が DLL 内の関数を呼び出すことを可能にします。

- **追加ファイル**
  DLL 関数を GUI 用語で宣言する外部関数宣言と、グローバル変数宣言が含まれます。

GUIクライアントプログラムの作成

全ての GUI のクライアントには、上述のファイルだけでなく、命令、つまり何をすべきかを指示するコードが必要です。3 層分散アプリケーションには、ローカルとリモートの 2 種類の命令があります。

ローカルコードが GUI のクライアントの操作を行い、リモートコードは他のプログラム、すなわちサーバ内で呼び出されたプロシージャの応答を行います。

ローカルコードの作成を終了したら、リモートコード、すなわち RPC の作成に専念してください。各インタフェースには、関連する.def の拡張子をもつ IDL（定義）ファイルがあります。引数の数やその他のシンタックスのリファレンスとして、IDL ファイルを使用してください。

クライアント・スタブの準備

クライアント・スタブは、GUI クライアントがサーバと通信するためのアプリケーションの一部です。これらのスタブのコードは **RPCMake** で生成されます。

GUI クライアントには、通信するサーバ毎にスタブが必要になります。たとえば、アプリケーションに 4 つのサーバがある場合、GUI クライアントには 4 つのスタブが必要です。これらのスタブは、それぞれのサーバに関連する 4 つの IDL ファイルから生成されるものです。

スタブを作成したら、それを GUI のクライアントにロードすることができます。

この作業には、クライアントにスタブを取り込むための GUI への読み込みがあります。

環境ファイルの編集

GUI のクライアントを完成させるための次のステップは、環境ファイルが予測される位置にあり、有効な設定情報を含んでいるかどうかを確認することです。

次のステップ

上記の手順を終えたら、クライアントはサーバで実行されるどの関数でも呼び出すことができるようになります。

しかし、GUI のクライアントがこれで本当に完成されたわけではありません。コードの 1 つ 1 つが完全であれば完成ですが、そうでなければデバッグプロセスが始まるのです。

統合された 3 層構造の分散型アプリケーションの一部分として GUI クライアントをテストしてください。

これで、完成されたクライアントをエンドユーザに手渡すことができます。

第3章　C, COBOLクライアント

この章では、C、COBOL クライアントプログラムを構築する手順を説明します。この章では、『サーバ開発者ガイド』および『はじめにお読みください』で説明した概念を使用します。このため、この 2 冊の内容を前もって理解しておいてください。また、C、COBOL のプログラミング知識があることも前提となっています。

クライアントプログラムの作成

クライアントプログラムは、フロントエンドユーザインタフェース、あるいは別のサーバに対し 1 つ以上のリモートプロシージャを発行してクライアントの働きをするサーバから構成されます。

リモート関数呼び出しをクライアントにコーディングする際のシンタックスは、ローカル関数呼び出しの場合と全く同じです。

必要であれば、IDL ファイルで、その関数呼び出しのパラメータを再確認してください。引数の個数と順番は、IDL ファイル中におけるその関数の引数のものと一致していなければいけません。ただし、引数の名前は、IDL ファイル中の引数名と一致していなくてもかまいません。

通常、サーバコードまたはクライアントプログラムを作成する両方の開発チームが共通のインタフェースを用いて並行してコードを書けるようにするため、事前に IDL ファイルを作成する必要があります。

インクルードファイル

クライアントプログラムは、生成されたヘッダファイル (*interface*.h) を RPC ヘッダファイル `dceinc.h` の後にインクルードしなければなりません。

Nextra環境の設定

全てのクライアントは、ランタイム RPC の実行に先だって環境を初期化するために、初期関数を呼び出さなくてはなりません。

関数 dce_setenv()を呼び出し、クライアントプログラムで環境を設定します。この関数は、どの RPC の実行よりも先に呼び出さなくてはなりません。dce_setenv()は以下の 3 つの引数をとります。

1. 環境ファイル名
2. ユーザのログイン名
3. ユーザのセキュリティパスワード

上記2, 3については、どちらの引数にも NULL を設定してください。

例）dec_setenv (*envfile_name*, NULL, NULL)

注）一部のプラットフォームでは NULL を文字型配列に入れて関数に渡す必要があります。

dce_setenv()の詳細については、『リファレンス』の「ファイル仕様」と「Nextra API」の章を参照してください。

RPCステータスのチェック

各呼び出しの直後に RPC のステータスをチェックすることは非常に重要です。エラーが発生しなかったかどうか、また、返された値が正しいかどうかを検証する方法は他にはありません。このチェックには、関数 `dce_error()`, `dce_errnum()`, `dce_errstr()` を使用します。これらの関数によって得られる情報は、RPC 実行中に発生したエラーを判断するために使用されます。これら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。

COBOLサーバへの文字列の引き渡し

変数を COBOL サーバに渡す場合には、その変数が静的変数であることを確認してください。COBOL では、どの変数も一定の長さであることが前提となっているため、必要に応じて文字列に空白を埋め込まなければなりません。

クライアントプログラムの例

Cコードの例

C で作成されたクライアントプログラムの例を次に示します。このプログラムを使うと、『サーバ開発者ガイド』の「ファンクショナリティ・サーバの構築」で説明されている basics の例にアクセスすることができます。

```
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);
}
```

COBOLコードの例

COBOL で作成されたクライアントプログラムの例を次に示します。このプログラムを使うと、『サーバ開発者ガイド』の「ファンクショナリティ・サーバの構築」で説明されている cobserv の例にアクセスすることができます。

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBCLI.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CBUFFER     PIC X(150).
01 XNULL       PIC S9(9) VALUE 0 BINARY.
01 RV          PIC 9(9) BINARY SYNC.
01 X           PIC S9(9) BINARY SYNC.
01 Y           PIC S9(9) BINARY SYNC.
01 Z           PIC S9(9) BINARY SYNC.
01 LOWER-BUFFERPIC X(100).
01 UPPER-BUFFERPIC X(100).
PROCEDURE DIVISION.
A000-MAIN.
*******************
* Call dce_setenv *
*******************
STRING "client.env" DELIMITED BY SIZE,
LOW-VALUE
DELIMITED BY SIZE INTO CBUFFER.
CALL "dce_setenv_ref"
USING CBUFFER, XNULL, XNULL, RV.
IF RV = 0
DISPLAY "CANNOT SET ENV.".
***************************
* Prompt user for inputs, *
* then display the result *
***************************
DISPLAY "ENTER X: ".
ACCEPT X.
DISPLAY "ENTER Y: ".
ACCEPT Y.
CALL "CADD" USING X, Y, Z.
CALL "dce_errnum" GIVING ERRNUM.
IF ERRNUM <> 0 THEN
DISPLAY "ERROR DURING RPC.".
ELSE
DISPLAY "Z = ", Z.
END-IF.
*******************************
* Prompt user to input string *
* to convert to upper case.   *
*******************************
DISPLAY "PLEASE ENTER THE STRING TO
CAPITALIZE:".
ACCEPT LOWER-BUFFER.
DISPLAY "The string entered was:".
DISPLAY LOWER-BUFFER.
CALL "CLOWER" USING LOWER-BUFFER, UPPER-BUFFER.
CALL "dce_errnum" GIVING ERRNUM.
```

```
IF ERRNUM <> 0 THEN
DISPLAY "ERROR DURING RPC.".
ELSE
DISPLAY "Z = ", Z.
END-IF.
*****************************
* Prompt user to input string *
* to convert to upper case.   *
*****************************
DISPLAY "PLEASE ENTER THE STRING TO
CAPITALIZE:".
ACCEPT LOWER-BUFFER.
DISPLAY "The string entered was:".
DISPLAY LOWER-BUFFER.
CALL "CLOWER" USING LOWER-BUFFER, UPPER-BUFFER.
CALL "dce_errnum" GIVING ERRNUM.
IF ERRNUM <> 0 THEN
DISPLAY "ERROR DURING RPC.".
ELSE
DISPLAY "The capitalized string is:".
DISPLAY UPPER-BUFFER.
END-IF.
****************
* End program. *
****************
STOP RUN.
END PROGRAM COBCLI.
```

C の関数引数のメモリ割り当て

Nextra 環境でコードを作成する際は、動的な関数引数のメモリ割り当てが最小限で済むようになっています。明示的に関数引数のためのメモリを解放しなくてはならないのは、動的配列として宣言されている出力変数(サーバからの値)だけです。この場合は、クライアント・スタブが動的に割り当てた、サーバ・スケルトンからの配列値のためのメモリを、クライアントプログラムで解放することが必要です。
動的配列値を保持する出力引数のメモリを解放するには、解放関数 dce_release()を使用します。 dce_release()は引数がありません。この関数は、動的配列値を保持するサーバの出力引数のために、クライアント・スタブが割り当てた全てのメモリを解放します。

例

dce_release()の使い方を説明するために、次のクライアントプログラムを見てみましょう。このクライアントプログラムは、file_server サーバから送られたバイナリファイルのために、クライアント・スタブが割り当てたメモリを解放するものです。(このサーバについては、『サーバ開発者ガイド』の「C メモリ割り当ての例」で説明されています。) コマンドラインで、

このクライアントは環境ファイルの名前とサーバマシンから送られたファイル名を受け取ります。

```
#include <stdio.h>
#include <string.h>
#include <dceinc.h>
#include "fileserver.h"

void main(int argc, char **argv)
{
int returnval;
long index=0;
char *recv_array, *filename, s[1000], msg[100];
FILE *outfile;

/* Set the Nextra environment */
if (!dce_setenv(argv[argc-1], NULL, NULL)) {
    fprintf (stderr, "Could not set env\n");
    exit(1);
}

/* Get filename to retrieve */
filename=(char *)dce_malloc(30);
strcpy(filename, argv[argc-2]);
printf("%s\n",filename);

/* get file from remote host */
if ((returnval=send_file(filename, &index,
 &recv_array))==1){
    if (outfile=fopen("dup","wb")){
        if (!(fwrite(recv_array, 1, index,
            outfile))) {
        printf("Could not write\n");
        fclose(outfile);
    } else {
        printf("Could not duplicate \n");
        dce_release();
    }
} else {
    printf("Could not get file \n");
    free(filename);
}
}
```

ボールド体で示されたテキスト(**dce_release()**)は、コンストレインド配列として定義された出力変数のためのメモリを解放します。
クライアントがコンストレインド配列中の情報を使用した後で、dce_release()が呼び出されている点に注意してください。

Windowsの静的配列

Windows において、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場合には、その配列を格納するために十分な大きさの HEAPSIZE を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];
```

配列がこのように定義されている場合には、アプリケーションのモジュール定義ファイルの HEAPSIZE は、これらの配列のサイズより大きな値を持たなければなりません。

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

値の合計を計算するには、実際の変数名をファイルに入力するのではなく、これら 3 つの数字を合計することに注意してください。したがって、ARRAY_SIZE1=6000 および ARRAY_SIZE2=7000 である場合は、たとえば、次のような行をモジュール定義ファイルに入力します。

```
HEAPSIZE 16384
```

COBOLコーディングの問題

Nextra 関数を呼び出すときには、ポインタパラメータはリファレンス渡し（アドレス渡し）、レギュラーパラメータは値渡しにします。

NULLターミネーション

COBOL のコードでは、Nextra 関数に渡される全ての文字列は NULL で終わらなければなりません。コンパイラがサポートしている場合には、定数も次のように NULL で終わることができます。

```
01 USER-NAME PIC X(10) VALUE

"smith" & X"00".
```

値が未知である変数を NULL で終わらせる方法は 2 つあります。1 つめの方法は、それぞれの変数について、dce_null_terminate( )を呼び出すことです。文字列の中に空白があると、この方法はうまくいきません。この関数は、次のように呼び出します。

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

2 つめの方法は、変数と定数の両方で使用されます。

```
STRING "ode.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.
```

ネーミングルール

各コンパイル言語に対応するクライアントプログラムのファイル名には拡張子が付きます。この拡張子は、コードが書かれた言語を表すものです。

表 3.1：クライアントプログラムのネーミングルール

言語	拡張子
C	.c
COBOL	.cbl
Perl	.pl

たとえば、上記の C プログラムの名前の例は cclient.c となります。

クライアントプログラムのデバッグ

ランタイム分散アプリケーション中のクライアントとして取り込む前に、クライアントプログラムをテストしてください。初期段階でコードをテストすることによって、クライアントプログラム自身のシンタックス上、またはプロシージャ上のエラーについて問題を特定できます。

クライアントプログラムのシンタックス上のエラーを検査するには、全ての RPC を一時的にコメントアウトしてからプログラムをコンパイルし、実行することによってテストしてください。他には、ローカル関数を一時的に代用する方法もあります。代用する関数には、リモート関数と同じパラメータを返す関数を使用してください。

次のステップ

クライアントプログラムを作成し、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クライアント・スタブの生成に進んでください。

クライアント・スタブの生成

クライアントがアクセスするサーバごとに IDL ファイルをすでに作成しているので、その手順は省略してクライアント・スタブの生成に進むことができます。

RPCMake ユーティリティを使用して、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。**RPCMake** ユーティリティでは、以下の情報を指定する必要があります。

- IDL ファイルの名前
- クライアント・スタブをインプリメントする言語

クライアント・スタブを生成するには、コマンドラインから **RPCMake** を呼び出すか、あるいはユーティリティのグラフィックフロントエンドを呼び出します。前者は処理を速く行うことができますが、Nextra ツールの使用に慣れていないユーザにとっては、後者の方が **RPCMake** を簡単に使用できます。

コマンドラインからのRPCMakeの呼び出し

コマンドラインから **RPCMake** を呼び出すには、次のシンタックスを使用します。

```
> rpcmake -d file. def -c language1
```

ここで、*file*.def は IDL ファイル名、*language1* はクライアント・スタブを作成する言語の省略形になります。

言語を指定する際は、C 言語では c を、COBOL 言語では mfcobol を使用します。クライアント・スタブの言語は、クライアントプログラムで使用する言語と同じでなければなりません。一方、クライアント・スタブの言語は、サーバコードが作成された言語と一致しなくてもかまいません。

RPCMake の最もシンプルな形式を見てきましたが、RPCMake を 1 回呼び出すだけで、クライアント・スタブを複数の言語で生成することができます。同じコマンドラインで、異なる-c オプションを必要なだけ指定することができます。

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

RPCMake 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。

RPCMake がエラーを通知した場合は、次の 2 つの点を確認してください。

(1) ODEDIR 環境変数が、使用する通信プロトコルのための適切なディレクトリに設定されているかどうか、また、PATH 環境変数に $ODEDIR/bin ディレクトリが含まれているかどうかを確認してください。

(2) IDL ファイルを指定する際は、**RPCMake** が呼び出されたディレクトリに IDL ファイルが存在するかどうかを確認するか、IDL ファイルのフルパス名を指定してください。

RPCMakeが生成するファイル

RPCMake のセッションを終了すると、コマンドラインの-c または-s オプション、あるいは 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作られます。

C クライアント・スタブでは、ヘッダファイルも合わせて生成されます。クライアントプログラムの先頭において、#include 文を使ってヘッダファイルをインクルードしてください。

#include "interface.h"

例

次のような **RPCMake** コマンドを発行したとします。

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

すると、次のファイルが生成されます。

```
cserver.h

cserver_c.c
```

ネーミングルール

スタブ・スケルトン名の接頭語は、IDL ファイルで指定したインタフェース名によって決まります。スタブ・スケルトン名の拡張子は、スタブがクライアント・スタブであるかサーバ・スケルトンであるかを表します。また、コードがインプリメントされている言語も表します。

表 3.2：スタブ・スケルトンのネーミングルール

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

次のステップ

クライアント・スタブを作成したら、次のステップに進むことができます。

C あるいは COBOL を使用する場合、クライアントをコンパイルする必要があります。

クライアントのコンパイル

COBOLでのコンパイル

COBOL で作成されたクライアントの場合、次の手順はクライアントの実行ファイルのコンパイルです。UNIX の MicroFocus COBOL では、次のようになります。

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

x オプションはコンパイラに、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指示します。e "RPCMAIN"オプションは、エントリ関数名を「RPCMAIN」と指定するものです。関数名はクライアント・スタブに "RPCMAIN"とハードコードされるため、この情報は重要です。

cobol_files は、cblserv.cbl のような COBOL ソースファイルのリストです。

libraries は、必要なライブラリのリストです。Nextra COBOL プログラムでは、少なくとも -L($ODEDIR)/lib -lrpccobol -lrpc になります。

executable_name は、作成される実行ファイルの名前です。

Cでのコンパイル

C で書かれたクライアントでは、次のステップはクライアントの実行ファイルのコンパイルになります。ANSI C で書かれたクライアントのコンパイルは次の手順で行います。

- 先にコンパイルされたオブジェクトファイルを、リンクの際にランタイムライブラリ librpc.*ext* と共にリンクします。このライブラリは、UNIX 環境ではインストレーションディレクトリ$ODEDIR/lib、Windows 環境では%ODEDIR%\LIB に置かれます。
- コンパイルの際には、プラットフォーム独自のフラグをコンパイラに渡す必要があります。

UNIX C Makefileの例

C クライアントコンパイルの例を示します。`make client` を入力したときに、次の `makefile` が UNIX 上のクライアントをコンパイルします。

```
# Substitute the name of your interface below
INTF = interface

# substitute the name of your client code below
# (minus the .c)
CLNT = client

# the rest of the file should not need to be changed
# $ODEDIR/bin/getplatform is a shell script that helps
# make this Makefile portable.

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

SOBJ = $(SERV).o $(INTF)_s.o
COBJ = $(CLNT1).o $(INTF)_c.o
LIBS = `getplatform libdir` `getplatform lib`
INCS = -I$(ODEDIR)/include `getplatform inc`

.c.o:
$(CC) -c $< $(INCS)
client: $(COBJ)
$(LD) -o $(CLNT1) $(COBJ) $(LIBS)
$(INTF)_s.c $(INTF)_c.c $(INTF)_c.pl: server.def
```

```
rpcmake.real -d server.def -c c -c pl -s c -y
```

クライアントについては、ソースコードファイル名から.c 拡張子を除いたファイル名を使ってください。この例では cclient.c なので、次のように指定します。

```
CLNT = cclient
```

WindowsでのCクライアントのコンパイル

次の makefile の例は、nmake を使って呼び出すことができます。この例では、クライアントとサーバの両方をコンパイルしています。

```
# Substitute the name of your server code below (minus
the .c)
SERV = basics
# substitute the name of your client program below (minus
the .c)
CLNT = cclient

# the rest of the file should not need to be changed
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: 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

クライアントのコンパイルに成功したら、次のステップである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

環境ファイルには、最低 1 つのブローカの位置を指定する属性が含まれなければなりません。オプションとして、環境ファイルにデバッグ、エラーロギングおよびコンフィギュレーションに関連する他の属性設定を含むこともできます。クライアントは起動時に環境ファイルをチェックし、サーバ位置情報を問い合わせるためにブローカにアクセスします。

環境ファイル作成の詳細は、『リファレンス』の「環境ファイルの作成」を参照してください。

次のステップ

環境ファイルを正しく作成できたら、クライアントを実行することができます。次のステップである、実行時のクライアントのテストに進んでください。

クライアントのテスト

次のステップは、作成したクライアントのテストです。ここまでに、以下のデバッグタスクを実行しておいてください。

- ローカルプログラムとしてのサーバコードのテスト
- **RPCDebug** によるサーバのテスト
- ローカルプログラムとしてのクライアントプログラムのテスト

次の点を検証するために、クライアントをテストします。

- 最新のクライアント・スタブが生成されたかどうか
- クライアントの実行ファイルが正しくコンパイルされたかどうか

問題点を上記の原因に特定するために、原則として、サーバが実行されている同じプラットフォームから、クライアントをテストします。しかし、コンパイルされたクライアントは、サ

ーバが実行されているプラットフォームと互換性がない可能性もあります。したがって、ランタイムエラーを突きとめる際、ネットワーク上の原因を想定する必要があります。ネットワークの問題としては、次のものがあります。

- クライアントマシンがサーバマシンと通信できない。
- クライアントプラットフォームが、これ以上ソケット接続を作成できない。

Windows上のクライアント

Windows 上で C クライアントを使用する場合は、コンパイラの標準インクルードディレクトリに `dceinc.h` をコピーするか、コンパイラのインクルードパスに$ODEDIR/includeディレクトリを追加してください。

テスト環境のセットアップ

新しいクライアントのためにテスト環境をセットアップするには、サーバとクライアントを次の順序で起動する必要があります。

1. ブローカを起動します。

2. サーバを起動します。

   『サーバ開発者ガイド』の「サーバのテスト」を参照してください。

3. クライアントを起動します。

RPCの呼び出し

クライアントの機能を実行してください。その堅牢性とパフォーマンスが満足できるものであれば、モジュールを分散アプリケーションに取り込んでください。

最後に （クライアントの実装）

クライアントのテストを完了した後、クライアントの実行ファイル、関連する環境ファイルをクライアントが実行されるプラットフォームに移動してください。

第4章　Delphiクライアント

この章では、3 層環境で Delphi を使用してクライアントを構築する手順を説明します。

この章を読む前に、『サーバ開発者ガイド』および本書の第1章、第2章で説明されている内容をご理解ください。

Delphiクライアントの開発

この節の開発プロセスの説明では、ローカルコードの初期テストと同じように、クライアントのおおまかな設計が終わっていることを前提とします。

Delphi クライアントの開発は、C や Perl のような文字ベースのプログラミング言語で作成されるクライアントの開発プロセスと同じです。

1. Delphi がアクセスする必要なファイルを設定します。

   このステップではいくつかのファイルが扱われます。DLL、外部関数宣言とグローバル変数の宣言を含むファイルです。

2. Delphi コードを作成します。

   コーディングには、Delphi コード および Nextra サーバ呼び出し(RPC)を使用します。

クライアントでは、起動時に最初のフォームが dce_setenv()を呼び出して、RPC の環境を設定しなければなりません。

デバッグ

分散アプリケーションに進む前に、クライアントをローカルアプリケーションとしてテストしてください。RPC をローカル関数に置き換えて、正しく実行できるかどうかを確認します。

3. Delphi クライアント・スタブを準備します。

クライアント・スタブには、Delphi クライアントが DLL を使ってサーバとブローカにアクセスするための Delphi 固有のコードが含まれます。クライアント・スタブの準備には 3 つのステップがあります。Nextra ユーティリティである **RPCMake** を使って、各サーバのためのクライアント・スタブを生成し、Delphi クライアントがあるマシンにスタブを送り、クライアントアプリケーションのプロジェクトにクライアント・スタブを追加します。

4. 環境ファイルを作成、編集します。

起動呼び出しである dce_setenv()が正しいファイルを指しているかどうかを確認します。

デバッグ

分散アプリケーションのフロントエンドとして、完成した GUI クライアントをテストしてください。

必要なファイルのロード

RPC を実行するために、Delphi は 2 つのファイルにアクセスする必要があります。ここでは、それらのファイルを使用可能にする方法について説明しています。

1. RPC ランタイムライブラリにアクセスします。

Delphi クライアントが RPC を実行するためには、Nextra ランタイムライブラリ（DLL）にアクセスしなければなりません。Delphi はこの librpc.dll を使用して、Delphi の外部関数として宣言されている関数を実行します。

2. 外部関数宣言およびグローバル関数宣言をインポートします。

librpc_dp*X*.pas は、DLL 中の関数のヘッダファイルとして機能します。odeconst.pas には、グローバル変数が含まれています。次の手順で、これらのファイルをプロジェクトに追加します。

A. Delphi を起動し、新規のプロジェクトを開きます。

B. 「プロジェクト」メニューから「追加」を選びます。ダイアログボックスより、追加するファイルを選択します。

C. ダイアログボックスで、Nextra をインストールしたディレクトリの下にある ClientLib ディレクトリ、さらにその下の delphi ディレクトリを選択します。 ファイルのリストから librpc_dp*X*.pas を選択して「OK」をクリックします。

D.　B と C の手順を繰り返して、ここで odeconst.pas をロードします。

E. プロジェクトを保存します。

この操作により、GUI クライアントが設定されて、クライアントプログラムをコーディングできるようになります。

Delphiクライアントプログラムの作成

RPC は、他の Delphi 関数呼び出しと同じシンタックスを使用します。RPC のシンタックスは次の 1 行になります。

```
rv := function_name(argument1, argument2, ..., argumentn);
```

たとえば、basics サーバ (『サーバ開発者ガイド』の「サーバのチュートリアル」参照)は、次のように書かれます。

```
var
rv, first, second : Integer;

 rv := add(first, second);
```

クライアントの設定

Nextra クライアントでは、RPC をコーディングする前に dce_setenv()を呼び出さなければなりません。この呼び出しは、プロジェクトファイルのアプリケーション起動前の部分に入れることをお勧めします。dce_setenv()は、クライアントのための正しい環境を準備します。dce_setenv()の呼び出しのシンタックスは次のとおりです。

```
rv := dce_setenv
     (AnsiString(path_of_environment_file),
      AnsiString(user_name),
      AnsiString(passwd));
```

ここで、path_of_environment_file は環境ファイルのフルパス名です。

たとえば、次のようなコードとなります。

```
rv := dce_setenv
      (AnsiString('c:\dpexample\client.env'), '', '');
```

文字列のCOBOLサーバへの引き渡し

変数を COBOL サーバに渡す場合は、その変数が静的変数であることを確認します。COBOL では、どの変数も一定の長さであることが前提となっているため、必要に応じて文字列に空白を埋め込まなければなりません。

エラー処理

RPC エラーのメッセージを受け取るには、エラー処理関数を明示的に呼び出さなければなりません。

DLL 関数 dce_errnum() または dce_error()を使用します。これらの関数は、それぞれの RPC の後に置きます。エラーが発生すると、Delphi はメッセージボックスをポップアップして、何が起こったかを通知します。この機能はクライアントの開発段階では役立ちますが、ユーザがアプリケーションを手にする前に変更した方が良い場合もあります。

エラー処理機能をコードに追加しない場合は、クライアントのログファイルでエラーを発見することができます。

GUIクライアントのローカルデバッグ

RPC を呼び出す Delphi コードを書いたら、クライアントマシン内でローカルにテストしてください。（このテストは RPC 部分を加える前にできます。）

デバッグの間は、RPC をコメントアウトします。その代わりに、実際の RPC を含むライブラリを、ローカル関数を含むローカルテストライブラリに置き換えます。ローカル関数を使うことによって、クライアント機能全体をテストできます。このテストで問題が起きたときには、その原因が RPC 部分にはないことがわかり、RPC を含めた後にテストするよりも原因の切り分けが容易になります。

Delphi のクライアントコードを作成し、ローカルのテストが終了したならば、クライアント・スタブを準備します。

Delphiクライアント・スタブの準備

スタブの生成

Delphi クライアント・スタブを生成するには、開発マシン上で **RPCMake** ユーティリティを使用します。次のように入力します。

```
> rpcmake -d def_file -c delphi
```

クライアント・スタブのファイル名は、インタフェース名に_c.pas が付いたものになります。たとえば、dpserver_c.pas は、dpserver というインタフェースのクライアント・スタブになります。

スタブの転送

開発マシンで **RPCMake** を使用してクライアント・スタブを生成した後で、Delphi プロジェクトが含まれている PC にそれを転送する必要があります。

スタブの取り込み

PC にスタブを転送したら、スタブをアプリケーションに取り込むことができます。

Delphi アプリケーションでクライアント・スタブを記録する方法は 2 つあります。全てのスタブのテキストを 1 つのユニットにまとめる方法か、またはそれぞれのスタブについて個別のユニットを作成する方法です。

全てのスタブを 1 つのユニットに入れる最初の方法の場合には、そのユニットに含まれる各 IDL ファイル中のそれぞれの関数名はユニークでなければなりません。Delphi では、同じユニットの中で関数名が重複することは許されていません。たとえば、dpserver と perlserver の両方に add()という関数を使用することはできません。この方法を使用する場合には、最初に全てのスタブを作成してから、それらを Delphi に読み込む手順で行うことをお勧めします。

それぞれのスタブについて個別のユニットを作成する2番目の方法の場合には、クライアントで使用可能な関数セットの中で、関数名がユニークである必要があるだけです。

クライアント・スタブをインポートするには、まず Delphi でプロジェクトを開きます。

A. Delphi を起動し、新規のプロジェクトを開きます。

B. 「プロジェクト」メニューから「追加」を選びます。ダイアログボックスより、追加するファイルを選択します。

C. ダイアログボックスで、クライアント・スタブが置いてあるディレクトリを選択します。ファイルのリストからクライアント・スタブを選択して「OK」をクリックします。

クライアント・スタブが複数ある場合には、B, C を繰り返します。

D. プロジェクトを保存します。

クライアント・スタブのインポートが終了したら、次にクライアントの環境ファイルを設定します。

環境ファイル

クライアントの設定

クライアントが呼び出す環境ファイルを作成または編集する必要があります。

ASCII エディタを使用して、環境ファイルを開きます。環境ファイルの中の DCE_BROKER 属性が、有効なブローカのホスト名とポート番号を指していることを確認します。そのポートで待つブローカが、クライアントがアクセスするものでなければなりません。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。

環境ファイル名は Delphi ランタイムにコーディングされるため、dce_setenv()呼び出しで指定されているドライブとパスの情報が、アプリケーションのエンドユーザにとって正しいものであることを、アプリケーションを配布する前に保証しなければなりません。

GUIクライアントのデバッグ

GUI クライアント開発の全てのステップを終えたら、分散環境の設定でテストしてください。コメントアウトしていた RPC の部分を元に戻し、直前のデバッグセッションで用いたローカル関数をコメントアウトします。そして、ブローカを起動し、サーバ、GUI クライアントを順に起動します。

アプリケーションが、正しい環境ファイルを使用していることを確認するには、呼び出しのシンタックスをチェックして、ファイルのパス情報が正しいことを確認します。

問題を診断するためにログファイルを使用する方法は次のとおりです。

- 環境ファイルを編集して、DCE_DEBUGLEVEL を ERROR, DEBUG に設定します。

アプリケーションでいくつか処理を実行して、ログファイルを検証します。最初の 10 行ほどのところに、環境ファイルに合った BROKERHOST と BROKERPORT の値があれば、問題があるのは環境ファイルではありません。

ログファイルに何も入力されていない場合には、dce_setenv()を再度チェックして、正しいファイルが読み込まれていることを確認します。呼び出しが正しい場合には、エラーが発生したのは DLL へのアプリケーションリンクの前であり、ネットワークまたは RPC ランタイムライブラリが原因であることが考えられます。

dce_setenv()が呼び出されたのにアプリケーションが動作しない場合には、ログファイルを見て、DLL がどこで hosts ファイルを探しているかを調べます。hosts ファイルの位置をチェックして、RPC ランタイムライブラリにとって正しい位置にあることを確認します。環境ファイルでブローカのホストとして指定されているマシンは、正しい IP アドレスで hosts ファイルにリストされていなければなりません。また、リストするときには、次の正しい形式に準拠していなければなりません。

```
IP_address hostname
```

ブローカのホスト名の代わりに IP アドレスをリストしている場合には、この形式は適用されません。

クイックチェックリスト

アプリケーションを初めてデバッグするときの参考のために、Delphi アプリケーションの必要条件である、以下のリストを使用してください。

1. DLL へのアクセス権を与えましたか？

   .DLL ファイルは、PC からの RPC 呼び出しを行うためのメカニズムを提供するため、Windows サーチパスの中になければなりません。詳細については、「必要なファイルのロード」を参照してください。

2. DLL の外部関数宣言およびグローバル変数定義をインポートしましたか？

.pas ファイルは、DLL にあるいくつかの関数のヘッダファイルとして機能します。詳細については、「必要なファイルのロード」を参照してください。

3. 起動関数に正しい引数を指定しましたか？

   起動関数に正しい引数が入力されていないと、Delphi クライアントは正しく実行できない可能性があります。dce_setenv()の詳細については、「Delphi クライアントプログラムの作成」を参照してください。

4. RPC を行う前に起動関数を呼び出しましたか？

   起動関数の呼び出しは、RPC の作成前に実行しなければなりません。この呼び出しは、最初のフォームの Form Load イベントの中に入力することをお勧めします。この環境が設定されていないと、他の RPC も正しく動作しないためです。詳細については、「Delphi クライアントプログラムの作成」を参照してください。

5. 環境ファイルの使用方法を検証しましたか？

   ユーザが指定したとおりに環境を設定するためには、Delphi クライアントが正しい環境ファイルを探さなければなりません。詳細については、『リファレンス』の「環境ファイル」を参照してください。

データタイプマッピング

表 4.1 に、IDL ファイルと Delphi のデータタイプの対応を示します。

表 4.1: データタイプマッピング

シンプルデータ型	
IDL	Delphi
int	Integer
long	Longint
float	Single
double	Double
char	Char
1次元配列	
IDL	Delphi
int [ ]	array of Integer
long [ ]	array of Longint
float [ ]	array of Single
double [ ]	array of Double
char [ ]	String
void [ ]	array of Byte
2次元配列	
IDL	Delphi
char [ ][ ]	Variant

第5章　PowerBuilderクライアント

この章では、3 層環境で PowerBuilder を使用してクライアントを構築する手順を説明します。

この章を読む前に、『サーバ開発者ガイド』および本書の第1章、第2章で説明されている内容をご理解ください。

はじめに

PowerBuilder クライアントの開発を始める前に、次の条件を確認してください。ここで確認しておけば、後で発生する時間と労力を節約できます。

プラットフォームの必要条件

開発パッケージを PowerBuilder がインストールされている Windows システムにインストールします。ソフトウェアのバージョン情報については「ネットワークの必要条件」および「ソフトウェアの必要条件」を参照してください。

また、クライアント・スタブを生成するには、Nextra 開発パッケージがインストールされている UNIX または Windows プラットフォームにアクセスする必要があります。

PowerBuilderクライアントの開発

この節の開発プロセスの説明は、ローカルコードの初期テストと同じように、クライアントのおおまかな設計が終了していることを前提とします。

PowerBuilder クライアントの開発プロセスは、C や Perl のような文字ベースのプログラミング言語で作成されるクライアントの開発プロセスと同じです。プロセスには 4 つの主な作業があります。

1. PowerBuilder クライアントが必要とするファイルを設定します。

このステップでは、Nextra ランタイム ライブラリ(librpc.dll)、ローカル PowerBuilder API(librpc_pb.txt)、グローバル外部関数宣言(odepb.txt)、そしてグローバル定数宣言(odeconst.txt)が扱われます。

2. PowerBuilder コードを作成します。

コーディングでは、PowerBuilder コードおよび Nextra サーバ呼出し(RPC)を使用します。

クライアントでは、起動時に Open イベントスクリプトが dce_setenv()を呼び出して、環境ファイル、ユーザ名、およびパスワードを指定しなければなりません。(現在の version ではユーザ名とパスワードにはブランク ("")を指定してください。) PowerBuilder アプリケーション終了時にリソースを解放するために用いる関数は dce_close_env()です。

デバッグ

分散アプリケーションに進む前に、クライアントをローカルアプリケーションとしてテストしてください。RPC をローカル関数に置き換えて、クライアントに構築した機能をテストします。

3. PowerBuilder クライアント・スタブを準備します。

クライアント・スタブには、PowerBuilder クライアントが DLL を使ってサーバとブローカにアクセスするための PowerBuilder 固有のコードが含まれます。クライアント・スタブの準備には 3 つのステップがあります。Nextra ユーティリティである **RPCMake** を使って、各サーバのためのクライアント・スタブを生成し、PowerBuilder クライアントがあるマシンにスタブを送り、クライアントアプリケーションにクライアント・スタブを取り込みます。

4. 環境ファイルを作成し、確認します。

起動呼び出しである dce_setenv()が正しい環境ファイルを指しているかどうかを確認します。

デバッグ

分散アプリケーションのフロントエンドとして、完成した GUI クライアントをテストしてください。

必要なファイルのロード

RPC を実行するためには、PowerBuilder は Nextra 関数が含まれている DLL および関数宣言が含まれているファイルにアクセスする必要があります。

RPCランタイムライブラリへのアクセス

PowerBuilder は、PowerBuilder の外部関数として宣言されている関数を実行するために、Nextra ランタイムライブラリ（DLL）を参照します。
Nextra クライアントでは、このファイルの名前は `librpc.dll` です。

PowerBuilder クライアントは PATH 変数により DLL の位置を認識できなければなりません。

関数宣言の統合

ここでは、RPC を実行するために PowerBuilder が必要とする外部関数にアクセスする方法について説明します。この方法では、Nextra ライブラリを PowerBuilder のライブラリサーチパスに追加し、Nextra 関数を PowerBuilder の外部関数として宣言します。

スクラッチから始める方法

1. ローカル PowerBuilder API を取り込みます。

   librpc_pb*X*.pbl ファイルは、`PowerBuilder` が解釈できる方法により `DLL` 中の関数のいくつかを定義します。 このファイルを `PowerBuilder` アプリケーションライブラリとして追加する方法は次のとおりです。

   A． ユーザアプリケーションを選択、右クリックをして「プロパティ」を選択します。

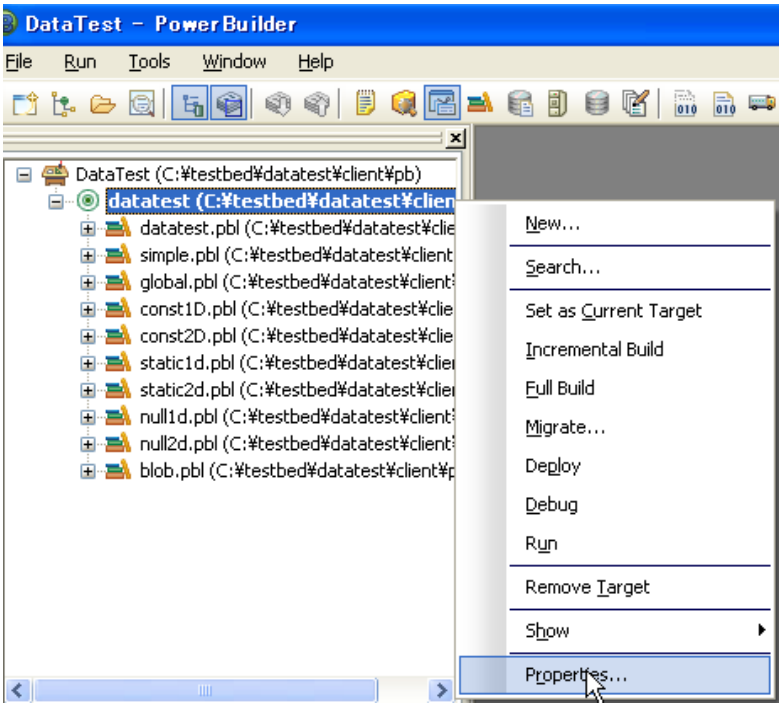

図 5.1：「プロパティの選択」

B. 「ライブラリ」リストボックスに、ローカル API ファイル(librpc_pb*X*.pbl)を選択、追加します。

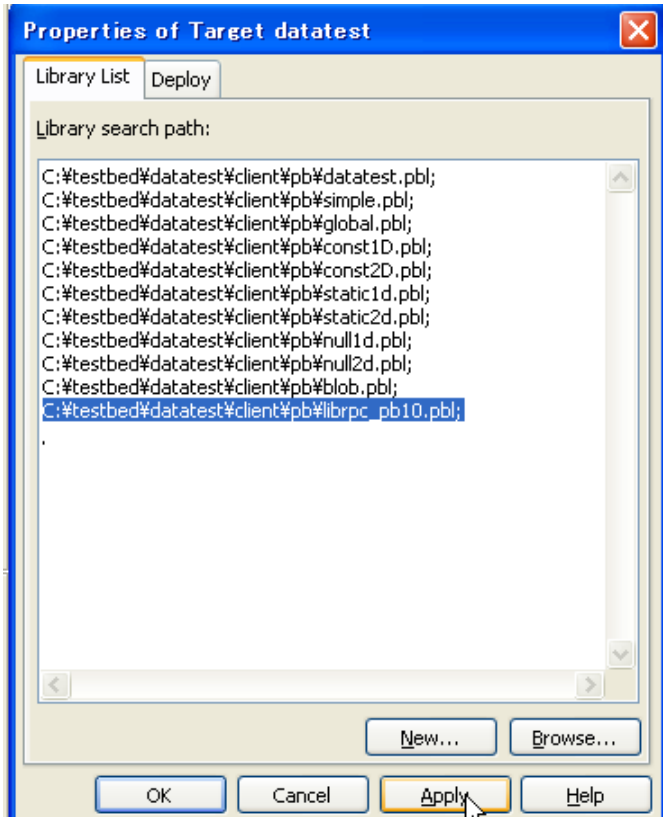

図 5.2： Nextra PBL ファイルのライブラリサーチパスへの追加

C.アプリケーションに追加されました。

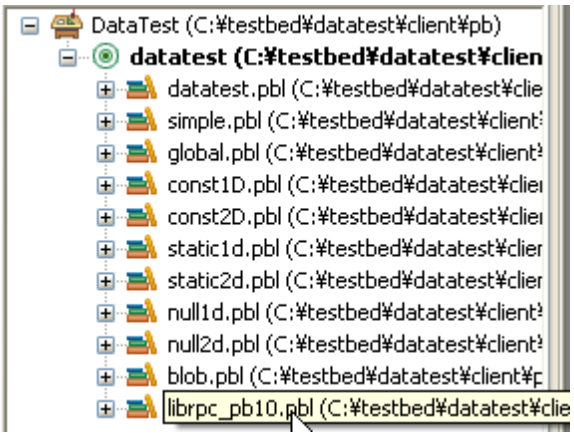

図 5.3： Nextra PBL ファイルの追加

2. odepb.txt という名前のテキストファイルから外部関数宣言をインポートします。

このファイルは、PowerBuilder が個別ライブラリなしで理解できる DLL のヘッダファイルとして機能します。

次の手順で、このファイルをアプリケーションに追加します。

A. 「メモ帳」を使用して odepb.txt を開き、その内容全体をクリップボードにコピーします。(「全ての範囲を選択」を選び、続いて「編集」メニューから「コピー」を選びます。)

B. PowerBuilder に入り、アプリケーションを選択します。

C. 「編集」メニューを選択します。

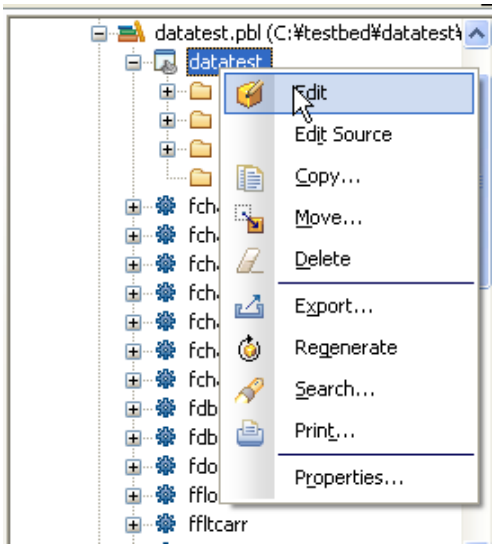

図 5.4: 編集メニューの選択

D. 「グローバル外部関数の宣言」を選択します。

E. カーソルがスクリプトのウィンドウに現れたら、<Ctrl-V>を入力して、odepb.txt をスクリプトに貼り付けます。

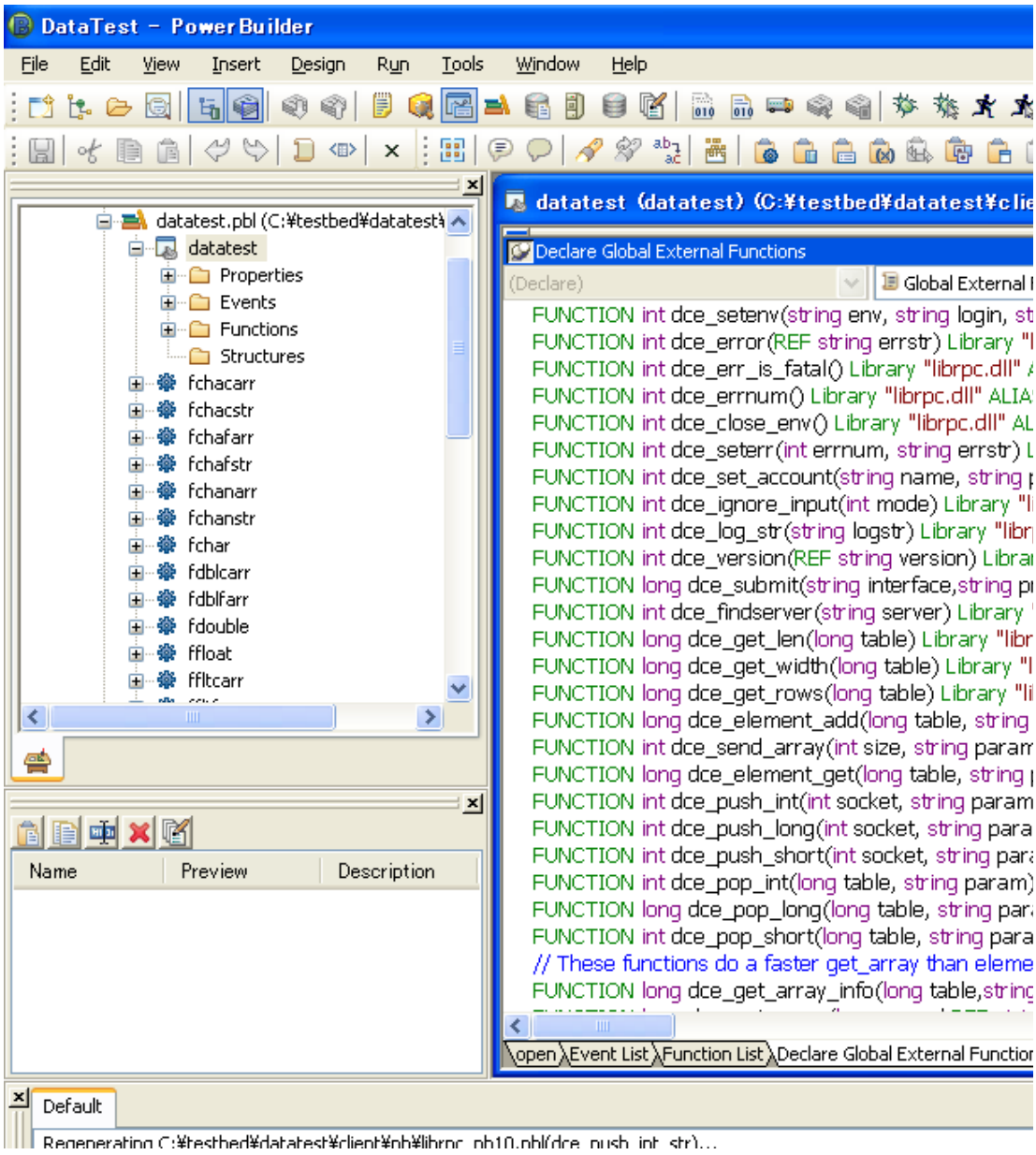

図 5.5: グローバル外部関数の宣言への貼り付け

この操作により、外部関数がアプリケーション内で宣言されました。

3. odeconst.txt テキストファイルからグローバル定数宣言をインポートします。

次の点を除いて、ステップ 2 と同じ手順になります。

A. 「メモ帳」を使用して odeconst.txt を開き、その内容全体をクリップボードにコピーします。

B. PowerBuilder に入り、アプリケーションを選択します。

C. 「編集」メニューを選択します。

D. 「インスタンス変数の宣言」を選択します。

E. カーソルがスクリプトのウィンドウに現れたら、<Ctrl-V>を押して、odeconst.txt をスクリプトに貼り付けます。

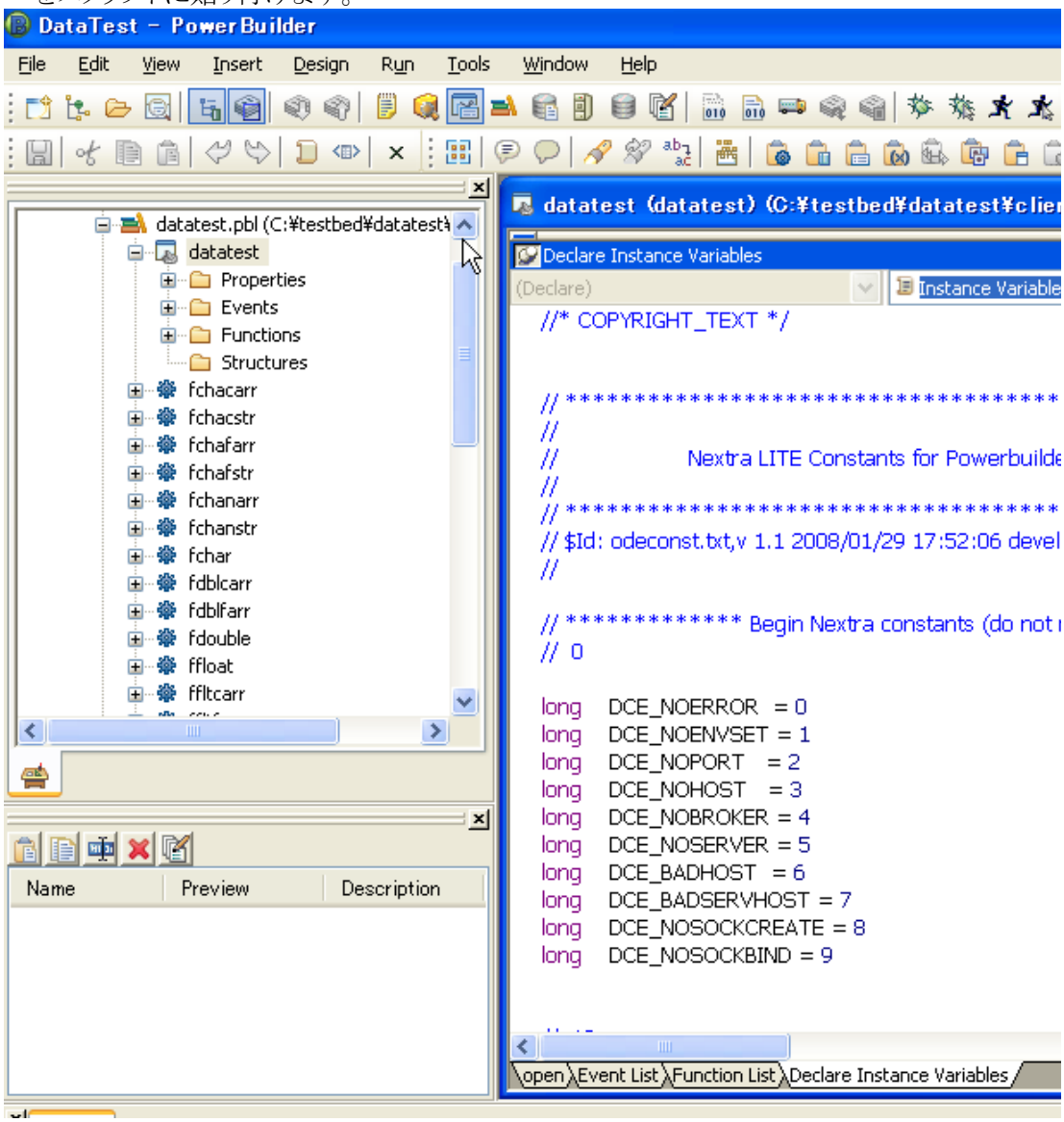

図 5.6:インスタンス変数の宣言への貼り付け

この操作により、グローバル定数がアプリケーション内で宣言されました。

PowerBuilderクライアントプログラムの作成

RPC は、他の PowerBuilder 関数呼び出しと同じシンタックスを使用します。RPC を実行するための PowerScript のシンタックスは次の 1 行になります。

```
rv=function_name (argument1, argument2, ..., argumentn)
```

たとえば、5 フィールドのデータとエラーメッセージを返す get_info というプロシージャに対する呼び出しは、次のようになります。

```
rv=get_info (PhoneNum, LastName, FirstName, HouseNum,
Street, errstr)
```

クライアントの設定

クライアントでは、RPC をコーディングする前に、dce_setenv()を呼び出さなければなりません。この呼び出しをスクリプトの OPEN イベントの中に入れることをお勧めします。dce_setenv()は、クライアントのために正しい環境を準備するものです。dce_setenv()の呼び出しの例は次のとおりです。

```
int rv
      rv = dce_setenv
      ("path_of_environment_file","user_name","passwd")
      if rv = 0 then dce_showerror()
return rv
```

ここで、path_of_environment_file は、環境ファイルのフルパスです。

user_name と *passwd* には、ブランク (“”)を指定してください。

環境ファイルを移動すると、dce_setenv()呼び出しが変更され、PowerBuilder の実行可能プログラムが再構築されるので、dce_setenv()呼び出しで指定するパスのデータがアプリケーションのエンドユーザにとって正しいかどうかを確認します。

コーディングする際の最後に用いる RPC は dce_close_env()でなければなりません。これは、アプリケーションの最後のフォームに含まれる必要があります。この関数は、セッションの間、クライアントに割り当てられていたリソースを解放します。

文字列のCOBOLサーバへの引き渡し

変数を COBOL サーバに渡す場合は、その変数が静的変数であることを確認します。COBOL では、どの変数も一定の長さであることが前提となっているため、必要に応じて文字列に空白を埋め込まなければなりません。

エラー処理

RPC エラーのメッセージを受け取るには、エラー処理の関数を明示的に呼び出さなければなりません。

DLL 関数 dce_error()または dce_error()を呼び出すローカル RPC API 関数 dce_showerror()を使用します。この関数は、それぞれの RPC の後に置きます。エラーが発生すると、PowerBuilder はメッセージボックスをポップアップして、何が起こったかを通知します。この機能はクライアントの開発段階では役に立ちますが、ユーザにアプリケーションを提供する前に解除した方が良い場合もあります。

エラー処理機能をコードに追加しない場合には、クライアントのログファイルでエラーを発見することができます。PowerBuilder は終了するまでクライアントログファイルの内容全体を書き込まないため、このログファイルを調べる前にアプリケーションを終了する必要があります。 以下のサンプルを参考にしてください。

```
integer rc
string dce_ErrorStr

dce_ErrorStr = Space(250)

/* RPC call */
lower2upper(lower, upper)

rc = dce_error(dce_ErrorStr)
if rc <> 0 then
   MessageBox("ODE ERROR", dce_ErrorStr)
End if
```

GUIクライアントのローカルデバッグ

RPC を呼び出す PowerBuilder コードを書いたら、クライアントマシン内でローカルにテストしてください。（このテストは RPC 部分を加える前にできます。）

デバッグの間は、RPC をコメントアウトします。その代わりに、実際の RPC を含むライブラリを、ローカル関数を含むローカルテストライブラリに置き換えます。ローカル関数を使うことによって、クライアント機能全体をテストできます。このテストで問題が起きたときには、

その原因が RPC 部分にはないことがわかり、RPC を含めた後にテストするよりも原因の切り分けが容易になります。

PowerBuilder のクライアントコードを作成し、ローカルのテストが終了したならば、クライアント・スタブを準備します。

PowerBuilderクライアント・スタブの準備

スタブの生成

PowerBuilder クライアント・スタブを生成するには、開発マシン上で **RPCMake** ユーティリティを使用します。

```
> rpcmake -d def_file -c pb32
```

生成されたクライアント・スタブのファイル名は、インタフェース名に、1 から振られる数字、さらに.srf32 が続きます。たとえば、myfun1.srf32 は、「myfunctions」というインタフェースの IDL ファイルに、最初に定義された関数のためのクライアント・スタブになります。

スタブの転送

開発マシン上で **RPCMake** を使ってクライアント・スタブを生成したら、PowerBuilder プロジェクトを含む PC にクライアント・スタブを転送する必要があります。

スタブの取り込み

PC にスタブを転送したら、スタブを PowerBuilder に取り込むことができます。

A.「アプリケーション」を選択後、右クリック、「インポート」を選択します。

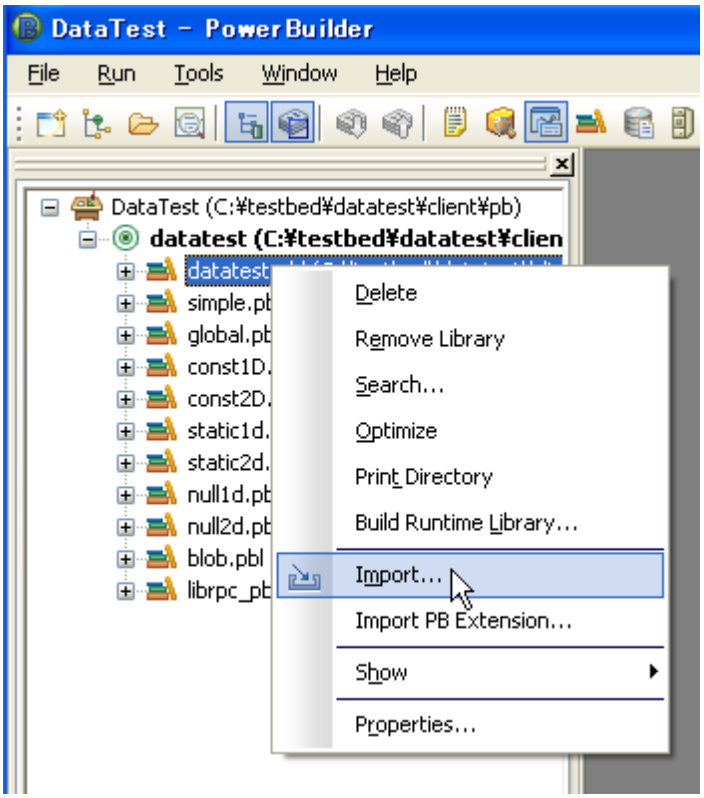

図 5.7:インポートを選択

B. 表示されるダイアログボックスで、スタブファイルを含むディレクトリを選択し、取り込むもファイルを選択します

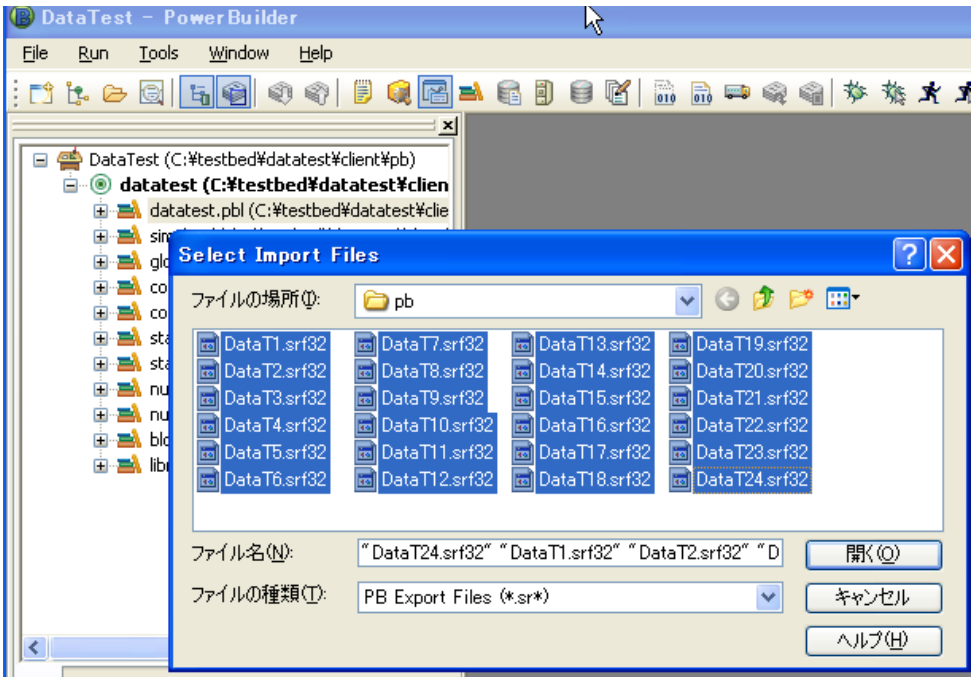

図 5.8： PB クライアント・スタブの取り込み

PB クライアント・スタブを取り込むことができたならば、クライアントのために環境ファイルを設定します。

環境ファイル

クライアントの設定

クライアントが使用する環境ファイルを作成または編集する必要があります。

ASCII エディタを使用して、環境ファイルを開きます。環境ファイル中の DCE_BROKER 属性が、有効なブローカのホスト名およびポート番号を指していることを確認します。そのポートで待つブローカが、クライアントがアクセスするものでなければなりません。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。

環境ファイル名は PowerBuilder ランタイムにコーディングされるため、`dce_setenv()`呼び出しで指定されているドライブとパスの情報が、アプリケーションのエンドユーザにとって正しいものであることを、アプリケーションを配布する前に保証しなければなりません。

GUIクライアントのデバッグ

開発の全ての手順を終了した後、分散環境で GUI クライアントをテストします。コメントアウトしていた RPC の部分を元に戻します。代わりに、ローカル関数が含まれているローカルテストライブラリを外して、実際の RPC が含まれているライブラリを入れます。その後、ブローカ、サーバ、GUI クライアントを順次起動します。

アプリケーションが、正しい環境ファイルを使用していることを確認するには、呼び出しのシンタックスをチェックして、ファイルのパス情報が正しいことを確認します。

問題を診断するためにログファイルを使用する方法は、次のとおりです。

- 環境ファイルを編集して、DCE_DEBUGLEVEL を ERROR, DEBUG に設定します。

アプリケーションで処理をいくつか実行して、ログファイルを検証します。最初の 10 行ほどのところに、環境ファイルに合った BROKERHOST と BROKERPORT の値があれば、問題があるのは環境ファイルではありません。

ログファイルに何も入力されていない場合には、dce_setenv()を再度チェックして、正しいファイルが読み込まれていることを確認します。呼び出しが正しい場合は、エラーが発生したのは DLL へのアプリケーションリンクの前であり、ネットワークまたは RPC ランタイムライブラリが原因であることが考えられます。

dce_setenv()が呼び出されたのにアプリケーションが動作しない場合には、ログファイルを見て、DLL がどこで hosts ファイルを探しているかを調べます。hosts ファイルの位置をチェックして、RPC ランタイムライブラリにとって正しい位置にあることを確認します。環境ファイルでブローカのホストとして指定されているマシンは、正しい IP アドレスで hosts ファイルにリストされていなければなりません。また、リストするときには、次の正しい形式に準拠しなければなりません。

```
IP_address hostname
```

ブローカのホスト名の代わりに IP アドレスをリストしている場合には、この形式は適用されません。

クイックチェックリスト

アプリケーションを初めてデバッグするときの参考のために、PowerBuilder アプリケーションの必要条件である、以下のリストを使用してください。

1. DLL へのアクセス権を与えましたか？

   .DLL ファイルは、PC からの RPC を行うためのメカニズムを提供するため、Windows サーチパスの中になければなりません。詳細については、「必要なファイルのロード」を参照してください。

2. PowerBuilder のローカル API をアプリケーションに統合しましたか？

   このローカル API は、DLL 関数のいくつかを PowerBuilder が理解できる用語に定義します。詳細については、「必要なファイルのロード」を参照してください。

3. テキストファイルから、DLL のグローバ外部関数宣言とグローバル定数宣言をインポートしましたか？

   odepb.txt ファイルは、PowerBuilder がそのまま理解できる DLL 中の関数のヘッダファイルとして機能し、odeconst.txt にはグローバル定数宣言が含まれています。詳細については「必要なファイルのロード」を参照してください。

4. 起動関数に正しい引数を入力しましたか？

起動関数に正しい引数が指定されていないと、PowerBuilder クライアントは正しく実行しない可能性があります。dce_setenv()の詳細については、「PowerBuilder クライアントプログラムの作成」を参照してください。

5. RPC を行う前に、起動関数を呼び出しましたか？

起動関数の呼び出しは、スクリプトの OPEN イベントの中に入れることをお勧めします。環境データがないと、他の RPC も正しく動作しないためです。詳細については「PowerBuilder クライアントプログラムの作成」を参照してください。

6. 環境ファイルの使用方法を検証しましたか？

希望どおりの環境を設定するためには、PowerBuilder クライアントは正しい位置にある正しい環境ファイルを見なければなりません。詳細については、『リファレンス』の「環境ファイル」を参照してください。

第6章　Visual Basicクライアント

この章では、3 層環境で Visual Basic を使用してクライアントを構築する手順を説明します。

この章を読む前に、『サーバ開発者ガイド』および本書の第1章、第2章で説明されている内容をご理解ください。

はじめに

Visual Basic クライアントの開発を始める前に、次の条件を確認してください。ここで確認しておけば、後で発生する時間と労力を節約できます。

制限事項

3 層アプリケーションのためのクライアント開発に使用する場合、Visual Basic にはいくつかの制限事項があります。Visual Basic クライアントプログラムを作成する場合は、以下の点に注意してください。

Visual Basicに依存するもの

メモリ割り当て

Visual Basic では、文字列配列の文字列についてのみ、メモリを動的に割り当てることができます。

void型の返り値

Visual Basic では、`void` 型の返り値をとる、IDL ファイル中で宣言されるサブルーチンがサポートされません。IDL ファイルで `void` 型の返り値をとる関数を宣言すると、**RPCMake** はその関数の代わりにサブルーチンにスタブを作成します。これらのツールは、関数の構造を変更して Visual Basic と互換性のあるスタブを作成します。

VB6 で作成したモジュールがランタイムエラーになる場合

最適化を指定して作成したモジュールを使用すると、ランタイムエラーなる場合があります。「最適化なし(O)」を指定して、再ビルドしてください。VB6 IDE の「プロジェクト(P)メニュー」より、xxx のプロパティ(E) → コンパイル → ネイティブコードコンパイル(N) → 最適化なし(O)のオプションボタンを選択してください。

Visual Basicクライアントの開発

この節の開発プロセスの説明では、ローカルコードの初期テストと同じように、クライアントのおおまかな設計が終わっていることを前提とします。

Visual Basic クライアントの開発は、C や Perl のような文字ベースのプログラミング言語で作成されるクライアントの開発プロセスと同じです。

1. Visual Basic がアクセスする必要なファイルを設定します。

   このステップではいくつかのファイルが扱われます。DLL、外部関数宣言とグローバル変数の宣言を含むファイルです。

2. Visual Basic コードを作成します。

   コーディングには、Visual Basic コード および Nextra サーバ呼び出し(RPC)を使用します。

   クライアントでは、起動時に最初のフォームが dce_setenv()を呼び出して、RPC の環境を設定しなければなりません。Visual Basic アプリケーション終了時にリソースを解放するための関数は dce_close_env()です。

デバッグ

分散アプリケーションに進む前に、クライアントをローカルアプリケーションとしてテストしてください。RPC をローカル関数に置き換えて、正しく実行できるかどうかを確認します。

3. Visual Basic クライアント・スタブを準備します。

   クライアント・スタブには、Visual Basic クライアントが DLL を使ってサーバとブローカにアクセスするための Visual Basic 固有のコードが含まれます。クライアント・スタブの準備には 3 つのステップがあります。Nextra ユーティリティである **RPCMake** を使って、各サーバのためのクライアント・スタブを生成し、Visual

Basic クライアントがあるマシンにスタブを送り、クライアントアプリケーションにクライアント・スタブを取り込みます。

4. 環境ファイルを作成、編集します。

起動呼び出しである dce_setenv()が正しいファイルを指しているかどうかを確認します。

デバッグ

分散アプリケーションのフロントエンドとして、完成した GUI クライアントをテストしてください。

必要なファイルのロード

RPC を実行するために、Visual Basic は 2 つのファイルにアクセスする必要があります。ここでは、それらのファイルを使用可能にする方法について説明しています。

1. RPC ランタイムライブラリにアクセスします。

Visual Basic クライアントが RPC を実行するためには、Nextra ランタイムライブラリ(DLL)にアクセスしなければなりません。Visual Basic は librpc.dll、および librpcvb.dll を使用して、Visual Basic の外部関数として宣言されている関数を実行します。

2. 外部関数宣言およびグローバル関数宣言をインポートします。

librpc_vb*X*.bas(VB6 の場合は、librpc_vb6.bas)は、DLL 中の関数のヘッダファイルとして機能します。odeconst.bas には、グローバル変数が含まれています。次の手順で、これらのファイルをプロジェクトに追加します。

A. Visual Basic を起動し、新規のプロジェクトを開きます。

B. 「ファイル」メニューから「ファイルの追加...」を選びます。ダイアログボックスが、追加するファイル名の入力のプロンプトを表示します。

C. ダイアログボックスで、Nextra をインストールしたディレクトリより librpc_vb*X*.bas を選択して「OK」をクリックします。

D. B と C の手順を繰り返して、ここで odecosnt.bas をロードします。

E. プロジェクトを保存します。

この操作により、GUI クライアントが設定されて、クライアントプログラムをコーディングできるようになります。

Visual Basicクライアントプログラムの作成

RPC は、他の Visual Basic 関数呼び出しと同じシンタックスを使用します。RPC のシンタックスは次の 1 行になります。

```
rv = function_name (argument1, argument2, ..., argumentn)
```

たとえば、basics サーバ (『サーバ開発者ガイド』の「サーバのチュートリアル」参照)内 add 関数に対する VB クライアント RPC コールは、以下のように記述されます。

```
Dim first as long
Dim second as long
Dim rv as long

rv = add(first, second)
```

IDL ファイル中での void 返り値

Visual Basic では、void 型の返り値をとる関数は使えません。詳細については、「void 型の返り値」を参照してください。

クライアントの設定

Nextra クライアントでは、RPC をコーディングする前に dce_setenv()を呼び出さなければなりません。この呼び出しは、最初のフォームの中に入れることをお勧めします。dce_setenv()は、クライアントのための正しい環境を準備します。dce_setenv()の呼び出しのシンタックスは以下の通りです。

```
rv% = dce_setenv
("path_of_environment_file","user_name","passwd")
if (rv% = 0) Then
    Call dce_showerror()
End If
```

ここで、*path_of_environment_file* は環境ファイルのフルパス名です。 *user_name, passwd* には、空ストリング(“”)を指定してください。

たとえば、Object を Form に、Procedure を Load に設定した後に、次のようなコードを追加することができます。

```
Sub Form_Load
   Call dce_setenv("c:\vbexampl\client.env","","")
End Sub
```

環境ファイルの位置を変更すると dce_setenv()呼び出しが変更され、Visual Basic の実行可能プログラムが再構築されるため、dce_setenv()呼び出しで指定するパスのデータがアプリケーションのエンドユーザにとって正しくなるように確認します。

コーディングする際の最後に用いる RPC は dce_close_env()でなければなりません。これは、アプリケーションの最後のフォームに含まれる必要があります。 dce_close_env()は、セッションの間、クライアントに割り当てられていたリソースを解放します。

文字列のCOBOLサーバへの引き渡し

変数を COBOL サーバに渡す場合は、その変数が静的変数であることを確認します。COBOL では、どの変数も一定の長さであることが前提となっているため、必要に応じて文字列に空白を埋め込まなければなりません。

エラー処理

RPC エラーのメッセージを受け取るには、エラー処理関数を明示的に呼び出さなければなりません。

DLL 関数 dce_error()または dce_error()を呼び出すローカル RPC API 関数 dce_showerror を使用します。この関数は、それぞれの RPC の後に置きます。エラーが発生すると、Visual Basic はメッセージボックスをポップアップして、何が起こったかを通知します。この機能はクライアントの開発段階では役立ちますが、ユーザがアプリケーションを手にする前に変更した方が良い場合もあります。

エラー処理機能をコードに追加しない場合は、クライアントのログファイルでエラーを発見することができます。Visual Basic は終了するまでクライアントログファイルの内容全体を書き込まないため、このログファイルを調べる前にアプリケーションを終了する必要があります。

GUIクライアントのローカルデバッグ

RPC を呼び出す Visual Basic コードを書いたら、クライアントマシン内でローカルにテストしてください。（このテストは RPC 部分を加える前にできます。）

デバッグの間は、RPC をコメントアウトします。その代わりに、実際の RPC を含むライブラリを、ローカル関数を含むローカルテストライブラリに置き換えます。ローカル関数を使うことによって、クライアント機能全体をテストできます。このテストで問題が起きたときには、その原因が RPC 部分にはないことがわかり、RPC を含めた後にテストするよりも原因の切り分けが容易になります。

Visual Basic のクライアントコードを作成し、ローカルのテストが終了したならば、クライアント・スタブを準備します。

Visual Basicクライアント・スタブの準備

スタブの生成

Visual Basic クライアント・スタブを生成するには、開発マシン上で **RPCMake** ユーティリティを使用します。次のように入力します。

```
> rpcmake -d def_file -c vb32
```

クライアント・スタブのファイル名は、インタフェース名に`_c.vb32` が付いたものになります。たとえば、`cserver_c.vb32` は、`cserver` というインタフェースのクライアント・スタブになります。

スタブの転送

開発マシンで **RPCMake** を使用してクライアント・スタブを生成した後で、Visual Basic プロジェクトが含まれている PC にそれを転送する必要があります。

スタブの取り込み

PC にスタブを転送したら、スタブをアプリケーションに取り込むことができます。

Visual Basic アプリケーションでクライアント・スタブを記録する方法は 2 つあります。全てのスタブのテキストを 1 つのモジュールにロードする方法か、またはそれぞれのスタブについて個別のモジュールを作成する方法です。

全てのスタブを 1 つのモジュールに入れる最初の方法の場合には、そのモジュールに含まれる各 IDL ファイル中のそれぞれの関数名はユニークでなければなりません。Visual Basic では、同じモジュールの中で関数名が重複することは許されていません。たとえば、cserver と perlserver の両方に add()という関数を使用することはできません。この方法を使用する場合には、最初に全てのスタブを作成してから、それらを Visual Basic にロードすることをお勧めします。

それぞれのスタブについて個別のモジュールを作成する2番目の方法の場合には、クライアントで使用可能な関数セットの中で、関数名がユニークである必要があるだけです。

クライアント・スタブをインポートするには、まず Visual Basic でプロジェクトを開きます。

1. 「ファイル」メニューから「新規コードモジュール」を選択します。

   (Visual Basic はこの新しいファイルに module1.bas という名前を付けます。より覚えやすい名前に変更することをお勧めします)

2. 新しいモジュールをダブルクリックします。

3. 「ファイル」メニューから「テキストの読み込み」を選択します。

   表示されるダイアログボックスから、目的のクライアント・スタブ (.vb32 ファイル)を選択し、「置換」ボタンをクリックします。

   全てのスタブを 1 つのモジュールにロードする場合は次のようにします。

   - テキストをロードするときに、最初のスタブについて「置換」ボタンをクリックします。Visual Basic はモジュールをクリアして、最初のスタブの内容を入れます。
   - 続く全てのスタブについて、「追加」ボタンをクリックします。それぞれのスタブに個別のモジュールを使用する場合には、スタブをロードするときに「置換」ボタンを使用します。

4. Form を保存します。

   「ファイル」メニューの「Form の上書き保存」 または「名前を付けて Form の保存」を使用します。

5. プロジェクトを保存します。

   「ファイル」メニューの「名前を付けてプロジェクトの保存」を使用します。

クライアント・スタブのインポートが終了したら、次にクライアントの環境ファイルを設定します。

環境ファイル

クライアントの設定

クライアントが呼び出す環境ファイルを作成または編集する必要があります。

ASCII エディタを使用して、環境ファイルを開きます。環境ファイルの中の DCE_BROKER 属性が、有効なブローカのホスト名とポート番号を指していることを確認します。そのポートで待つブローカが、クライアントがアクセスするものでなければなりません。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。

環境ファイル名は Visual Basic ランタイムにコーディングされるため、dce_setenv() 呼び出しで指定されているドライブとパスの情報が、アプリケーションのエンドユーザにとって正しいものであることを、アプリケーションを配布する前に保証しなければなりません。

GUIクライアントのデバッグ

GUI クライアント開発の全てのステップを終えたら、分散環境の設定でテストしてください。コメントアウトしていた RPC の部分を元に戻し、直前のデバッグセッションで用いたローカル関数をコメントアウトします。そして、ブローカを起動し、サーバ、GUI クライアントを順に起動します。

アプリケーションが、正しい環境ファイルを使用していることを確認するには、呼び出しのシンタックスをチェックして、ファイルのパス情報が正しいことを確認します。

問題を診断するためにログファイルを使用する方法は次のとおりです。

- 環境ファイルを編集して、DCE_DEBUGLEVEL を ERROR, DEBUG に設定します。

アプリケーションでいくつか処理を実行して、ログファイルを検証します。最初の 10 行ほどのところに、環境ファイルに合った BROKERHOST と BROKERPORT の値があれば、問題があるのは環境ファイルではありません。

ログファイルに何も入力されていない場合には、dce_setenv()を再度チェックして、正しいファイルが読み込まれていることを確認します。呼び出しが正しい場合には、エラーが発生したのは DLL へのアプリケーションリンクの前であり、ネットワークまたは RPC ランタイムライブラリが原因であることが考えられます。

dce_setenv()が呼び出されたのにアプリケーションが動作しない場合には、ログファイルを見て、DLL がどこで hosts ファイルを探しているかを調べます。hosts ファイルの位置をチェックして、RPC ランタイムライブラリにとって正しい位置にあることを確認します。環境ファイルでブローカのホストとして指定されているマシンは、正しい IP アドレスで hosts ファイルにリストされていなければなりません。また、リストするときには、次の正しい形式に準拠していなければなりません。

IP_address hostname

ブローカのホスト名の代わりに IP アドレスをリストしている場合には、この形式は適用されません。

クイックチェックリスト

アプリケーションを初めてデバッグするときの参考のために、Visual Basic アプリケーションの必要条件である、以下のリストを使用してください。

1. DLL へのアクセス権を与えましたか？

   .DLL ファイルは、PC からの RPC 呼び出しを行うためのメカニズムを提供するため、Windows サーチパスの中になければなりません。詳細については、「必要なファイルのロード」を参照してください。

2. DLL の外部関数宣言およびグローバル変数定義をインポートしましたか？

   .BAS ファイルは、DLL にあるいくつかの関数のヘッダファイルとして機能します。詳細については、「必要なファイルのロード」を参照してください。

3. 起動関数に正しい引数を指定しましたか？

   起動関数に正しい引数が入力されていないと、Visual Basic クライアントは正しく実行できない可能性があります。dce_setenv()の詳細については、「Visual Basic クライアントプログラムの作成」を参照してください。

4. RPC を行う前に起動関数を呼び出しましたか？

   起動関数の呼び出しは、RPC の作成前に実行しなければなりません。この呼び出しは、最初のフォームの Form Load イベントの中に入力することをお勧めします。この環

境が設定されていないと、他の RPC も正しく動作しないためです。詳細については、「Visual Basic クライアントプログラムの作成」を参照してください。

5. 環境ファイルの使用方法を検証しましたか？

ユーザが指定したとおりに環境を設定するためには、Visual Basic クライアントが正しい環境ファイルを探さなければなりません。詳細については、『リファレンス』の「環境ファイル」を参照してください。

第7章　Visual C++クライアント

この章を読む前に、『サーバ開発者ガイド』および本書の第1章、第2章で説明されている内容をご理解ください。

はじめに

Visual C++クライアントの開発を始める前に、次の条件を確認してください。ここで確認しておけば、後で発生する時間と労力を節約できます。

Microsoft Foundation Class Librariesの使用

Foundation Class Libraries を使用する場合、/Aw コンパイラオプションを使用しなければならないかもしれません。このオプションはアプリケーションに SS!=DS を指定します。

制限事項

3 層分散アプリケーションのためのクライアント開発に使用する場合、Visual C++にはいくつかの制限事項があります。Visual C++クライアントプログラムを作成する場合には、以下の点に注意してください。

開発の概要

この節では、ローカルコードの初期テストと同じように、クライアントのおおまかな設計が終了していることを前提として、開発プロセスを説明しています。

Visual C++クライアント作成の開発プロセスは、他の GUI 環境の開発プロセスと同じです。プロセスには、次の 4 つのメインタスクが含まれます。

1. IDL ァイルを作成し、Visual C++クライアント・スタブを準備します。

クライアント・スタブの準備には、次のステップがあります。IDL ファイルの作成、RPCMake を使用しての VC++クライアン・スタブの生成、そしてクライアント・スタブのアプリケーションへのインポートがあります。

2. Visual C++コードを作成します。

コーディングには、ローカル Visual C++コーディングおよび Nextra サーバ呼び出し(RPC)を使用します。

クライアントでは、クライアントプログラムは起動時に dce_setenv()を呼び出して、環境ファイル、ユーザ名およびパスワードを指定しなければなりません。(現在の version では、ユーザ名とパスワードには NULL を指定してください。)Visual C++アプリケーションが完了する前にリソースを解放するための関数は dce_close_env()です。

デバッグ

分散アプリケーションに進む前に、クライアントをローカルアプリケーションとしてテストしてください。RPC をローカル関数に置き換えて、正しく実行できるかテストします。

3. RPC ライブラリとヘッダファイルをアプリケーションに追加します。

4. 環境ファイルを作成して、確認します。

dce_setenv()の呼び出しが、環境ファイルの正しいパス名とファイル名を指定しているかどうか確認してください。

デバッグ

分散アプリケーションのフロントエンドとして、完成した Visual C++クライアントをテストしてください。

Visual C++クライアント・スタブの準備

Visual C++クライアント・スタブの準備には、3 つの段階があります。IDL ファイルの作成、スタブの生成、そして Visual C++へのロードです。

IDLファイル

IDL ファイルは、クライアントが使用できる各関数およびそのパラメータに関する情報を含んでいます。

スタブの生成

Visual C++クライアント・スタブを生成する場合、Nextra の **RPCMake** を使用して C スタブを生成します。次のように入力します。

> rpcmake -d *def_file* -c c

Nextra の **RPCMake** は、*server*_c.c ファイル（C クライアント・スタブ）および *server*.h（ヘッダファイル）を作成します。

スタブとヘッダファイルの転送

次に、生成したスタブとヘッダファイルを Visual C++マシンに移します。クライアントを実行するマシン上でスタブを生成した場合を除き、ファイルを転送することが必要です。

スタブのロード

スタブを、アプリケーションを含むディレクトリに転送した後、Visual C++内でアプリケーションをオープンし、スタブファイルを Visual C++にロードすることができます。

Visual C++プロジェクトをオープンすると、「プロジェクト」メニューに進み、*server*_c.c をそのプロジェクト内のファイルのリストに追加することができます。

クライアント・スタブを準備した後で、クライアント用の環境ファイルのセットアップを行うことができます。

Visual C++クライアントプログラムの作成

サーバを呼出す前に、初期化関数 dce_setenv()を呼び出さなければなりません。

環境のセットアップ

dce_setenv()は、クライアントに対して適切な環境を準備します。dce_setenv()の呼び出しの例は次のとおりです。

```
int rv
rv = dce_setenv
("path_of_environment_file","user_name","passwd")
if (dce_errnum() != DCE_NOERROR)
{
      exit(dce_errnum());
}
```

ここで、*path_of_environment_file* は環境ファイルのフルパス名で、ドライブ、パス、ファイル名を含みます。 *user_name* と *passwd* には、NULL を指定してください。

環境ファイルの位置を変更すると、文字列が変更され Visual C++の実行ファイルを再構築することになるため、環境ファイルに文字列を入れる場合、dce_setenv()呼び出しで指定する PATH のデータがアプリケーションのエンドユーザにとって正しくなるように確認してください。多くの場合、環境ファイル用の属性を使用して、後から使用するファイルを入力できる形式にする方が簡単です。

アプリケーションは、dce_close_env()を最後の関数として呼び出さなければなりません。この関数を呼び出すことにより、セッションの間、クライアントに割り当てられていたリソースを解放します。

文字列のCOBOLサーバへの引き渡し

変数を COBOL サーバに渡す場合は、その変数は固定サイズ変数であることを確認してください。COBOL では、どの変数も一定の長さであることが前提となっているため、必要に応じて文字列に空白を埋め込まなければなりません。

エラー処理

RPC エラーのメッセージを受け取るには、エラー処理関数を明示的に呼び出さなければなりません。

ローカル RPC API 関数の dce_errnum()と dce_errstr()を使用します。これらの関数は各 RPC の後に置きます。エラーが発生すると、dce_errnum()が設定され、dce_errstr()がエラーメッセージを返すことができます。

GUIクライアントのローカルデバッグ

RPC を呼び出す Visual C++コードを作成したら、クライアントマシン内でローカルにテストしてください。（このテストは RPC 部分を加える前にできます。）

デバッグの間は、RPC をコメントアウトします。その代わりに、実際の RPC を含むライブラリを、ローカル関数を含むローカルテストライブラリに置き換えます。ローカル関数を使うことによって、クライアント機能全体をテストできます。このテストで問題が起きたときには、その原因が RPC 部分にはないことがわかり、RPC を含めた後にテストするよりも原因の切り分けが容易になります。

Visual C++のクライアントコードを作成し、ローカルのテストが終了したならば、クライアント・スタブを準備します。

コンパイラオプション

各種のコンパイラオプションについては、Visual C++のマニュアルを参照してください。

ヘッダファイルのインクルード

Nextra ヘッダファイルをソースファイルにインクルードする必要もあります。ヘッダファイルは、Nextra 関数を使用する全てのソースファイルに必要であり、スタブ内に自動的にインクルードされることに注意してください。

```
#include "interface.h"
#include “dceinc.h”
```

環境ファイル

次に、クライアントが使用する環境ファイルを作成または編集する必要があります。

クライアントのセットアップ

ASCII エディタを使用して、環境ファイルを作成します。環境ファイル内の DCE_BROKER 属性が有効なブローカのホスト名とポート番号を指していることを確認してください。そのポートで待っているブローカは、クライアントがアクセスするものでなければなりません。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。

GUIクライアントのリモート構築とデバッグ

Visual C++クライアント開発の全てのステップを終了したら、クライアントを構築して分散環境の設定でテストしてください。コメントアウトしていた RPC 部分を元に戻し、直前のデバッグセッションで使用したローカル関数をコメントアウトします。それから、ブローカ、サーバ、GUI クライアントを順に起動します。

クイックチェックリスト

アプリケーションを初めてデバッグするときの参考のために、Visual C++アプリケーションの必要条件である、以下のリストを使用することができます。

1. 最初の RPC を行う前にクライアントプログラムの中で dce_setenv()を呼び出しましたか？

   この関数により Nextra ランタイム環境をセットアップした後、RPC を正常に実行することができます。詳細については、「Visual C++クライアントプログラムの作成」を参照してください。

2. Nextra ライブラリ(librpc.dll)に対するアクセス権を与えましたか？

   これらのライブラリは、Windows 上の Nextra アプリケーションに RPC を行うためのメカニズムを提供します。

3. クライアントが正しい環境ファイルを使用していることを検証しましたか？

   Visual C++クライアントは、dce_setenv()の呼び出しで環境ファイルを指定するか、あるいは実行時にユーザから情報を受け取ります。

第8章　Javaクライアント

この章を読む前に、『サーバ開発者ガイド』および本書の第1章、第2章で説明されている内容をご理解ください。

はじめに

特長

ユーザは、**rpcmake** コマンドを用いて作成した Java　クライアント・スタブを Java クライアントから容易に呼び出して、Nextra に接続することができます。

また、Java スレッドの機能を用いることにより、負荷分散を行うことができます。

アーキテクチャ

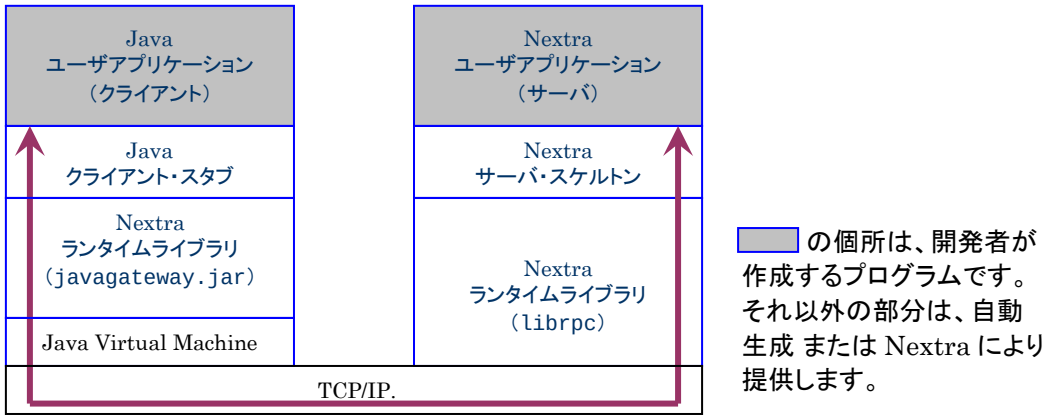

図 8.1: Java クライアント使用時のアーキテクチャ

プラットフォームの必要条件

完成した Java アプリケーションを実行する各プラットフォームは、Nextra Java ランタイムライブラリ（`javagateway.jar`）および JDK1.6.x のバージョンが必要です。

データタイプマッピング

表 8.1 に、Java クライアント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。

表 8.1：データタイプマッピング

Java Client	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

Javaクライアントと Nextra間でのデータの扱いについて

表 8.2 に、Java クライアントと Nextra 間でのデータ型の扱いの対応関係を示します。

表 8.2：Java クライアントと Nextra 間におけるデータ型の扱い

シンプルデータ型		
IDL	OUT パラメータの取得	返り値の取得
short	getShort(“パラメータ名”)	getShortReturn()
int	getInt(“パラメータ名”)	getIntReturn()
long	getInt(“パラメータ名”)	getIntReturn()
float	getFloat(“パラメータ名”)	getFloatReturn()
double	getDouble(“パラメータ名”)	getDoubleReturn()
char	getChar(“パラメータ名”)	getCharReturn()
object	getObject(“パラメータ名”)	getObjectReturn()
1 次元配列		
IDL	OUT パラメータの取得	返り値の取得
short	getShortArray(“パラメータ名”)	なし
int	getIntArray(“パラメータ名”)	なし
long	getIntArray(“パラメータ名”)	なし
float	getFloatArray(“パラメータ名”)	なし
double	getDoubleArray(“パラメータ名”)	なし
char	getString(“パラメータ名”)	なし
void	getByteArray(“パラメータ名”)	なし
2 次元配列		
IDL	OUT パラメータの取得	返り値の取得
char	getStringArray(“パラメータ名”)	なし

- getLong(“パラメータ名”)または getLongReturn() などを Java クライアントプログラム内で使用しないでください。コンパイルはできますが、Java と Nextra での long データの定義の違いにより、java.lang.ClassCastException が発生する恐れがあります。
- 詳しい使用方法については、「データ型別使用例」を参照してください。

引数Argumentの使用例

この章の最後にある「データ型別使用例」を参照してください。

返り値

char, short, long, int, float, double, object の関数リターン返り値の取得方法については、この章の最後にある「データ型別使用例」を参照してください。

Javaクライアント・スタブの生成

クライアント・スタブの生成は、RPC Developer にて生成する方法と、コマンドラインにて生成する方法があります。生成されるファイル名は、*interfacename*_c.java になります。

RPC Developer にある RPCMake タブにて、クライアント言語として Java を選択してください。または、コマンドラインでは以下のように使用されます。

```
> rpcmake -d file.def -c java
```

ここで、*file*.def は IDL ファイルとなります。

Javaクライアントの記述方法

Nextra 開発パッケージ、「samples」ディレクトリ内を参照してください。Nextra 開発パッケージでは、以下の Nextra エラーに対する Exception をサポートしています。

表 8.3： Nextra エラーに対する Exception

Error No.	Symbol	Exception
4	DCE_NOBROKER	DceNoBrokerException
5	DCE_NOSERVER	DceNoServerException
6	DCE_BADHOST	DceBadHostException
7	DCE_BADSERVHOST	DceBadServerHostException
13	DCE_NOSUCHFUNC	DceNoSuchFuncException
14	DCE_LOCALHOSTUNKN	DceLocalHostUnknownException
17	DCE_NOMEMORY	DceNoMemoryException
24	DCE_PEERERROR	DceConnectionResetByPeerException
25	DCE_LOSTSERVER	DceLostServerException
26	DCE_BADTCPINIT	DceBadTcpInitException
27	DCE_IPCINITBAD	DceBadIpcInitException
28	DCE_IPCCANTCLOSE	DceIpcCantCloseException
31	DCE_CANTFORK	DceCannotForkException
32	DCE_BADPORT	DceBadPortException
33	DCE_NOINTERFACE	DceNoInterfaceException
36	DCE_SIGINT	DceSignalInterruptException
37	DCE_SERVERFAILED	DceServerFailedException
38	DCE_BADINTERFACE	DceBadInterfaceException
43	DCE_MAXCAPACITY	DceMaxCapacityException
45	DCE_FSERROR	DceFileSystemException
46	DCE_RPCTIMEOUT	DceRPCTimeOutException
52	DCE_UNAVAILABLE	DceUnavailableException
58	DCE_MAXRPCECEEDED	DceMaxRPCExceededException
59	DCE_ASYNCRPCNOTFOUND	DceAsyncRPCNotFoundException
61	DCE_CANCELRPCERROR	DceCancelRPCErrorException
64	DCE_ASYNCINPROGRESS	DceAsyncInProgressException
67	DCE_THREADCREATIONFAILED	DceThreadCreationFailedException

68	DCE_THREADLOCKFAILED	DceThreadLockFailedException
69	DCE_UNABLE2PROCESS	DceUnable2ProcessException
その他、オブジェクトクライアントがサポートする Exception		
Server または Broker が QUEUE 溢れの場合		DceQueueFullException
上記の Exception のスーパークラス		DceException

Java プログラムでは、必ず以下を import してください。

```
import com.inspire.rpc.client.*;

import com.inspire.rpc.shared.*
```

クライアント環境の設定について、RPC の実行前に以下のようにして環境を設定してください。

Environment.dce_setenv("*filename*.env");

バリアブル・ネームド・サーバの記述について

クライアント・スタブ中の各メソッドの 1 番目の引数として、"dce_server"が生成されます。したがって、Java クライアントプログラムからメソッドを使用する場合は、必ず最初の引数に当該サーバ名を指定して呼び出してください。

デディケイテッド・サーバについて

クライアントプログラムを終了する前に、dce_dedDisconnect メソッドをクライアントプログラムから呼び出し、デディケイテッド・サーバ子プロセスを終了してください。あるいは、サーバ環境ファイルに「DCE_SVR_TIMEOUT」属性を指定して子プロセスのタイムアウトによる終了を行ってください。

環境ファイルについて

環境ファイル中には、以下の環境ファイル属性が使えます。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」環境ファイル属性を参照してください。以下は Java クライアント特有の環境ファイル属性です。
* GW_PUTNULL(JG exclusive)
* GW_TRIM(JG exclusive)

属性	説明
GW_PUTNULL デフォルト=false	2 次元文字配列の最後に null を付加する。null を追加する場合は”true”を指定。
GW_TRIM デフォルト=false	文字列末尾の空白文字を削除する。削除する場合は”true”を指定。

エンコーディング指定 DCE_LANG について

Nextra を UNIX 上でお使いのユーザは「SJIS」、Windows でお使いのユーザは、指定しないか、または「MS932」と指定してください。エンコーディングに関しては、サンマイクロシステムズのページ を参照してください。

一般的な環境ファイルは、以下のようになります。

```
DCE_BROKER=HOSTNAME1,PORT#1
           [HOSTNAME2,PORT#2]
DCE_CLN_TIMEOUT=60 ←秒
DCE_LANG=MS932 ←注)
```

サンプルプログラム

開発パッケージの「samples」ディレクトリ内に、単純なデータタイプ、一次元データタイプ、二次元データタイプの転送を行うサンプルがあります。

データ型別使用例 (Java)

注意事項

“バリアブル・ネームド・サーバ”でない場合の例です。“バリアブル・ネームド・サーバ”については、「バリアブル・ネームド・サーバの記述について」を参照してください。

rpcmake により生成されたクライアント・スタブのクラス名を「*interface*_c」とします。

RPCTable は、リモートメソッドの実行結果を格納するオブジェクトです。

RPCTable 中のアウトプットデータを収得するためには、データ型に応じて、**get<オブジェクトタイプ>("パラメータ名")**メソッドを使用してください。

RPCTable 中の返り値を収得するためには、データ型に応じて、**get<オブジェクトタイプ>Return()**メソッドを使用してください。

表 8.4：データ型別使用例

次元	配列	型	IDL ファイル内 関数例	パラメータと返り値の扱い方

Simple		short	short FuncShort(
			[in] short shVar,
			[out] short shVarOut);
				short shVar = 1;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncShort(shVar, "shVarOut");
				short output = params.getShort("shVarOut");
				short returnVal = params.getShortReturn();

		long	long FuncLong(
			[in] long lVar,
			[out] long lVarOut);
				int lVar = 1;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncLong(lVar, "lVarOut");
				int output = params.getInt("lVarOut");
				int returnValue = params.getIntReturn();

		int	int FuncInteger(
			[in] int nVar,
			[out] int nVarOut);
				int nVar = 1;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncInteger(nVar, "nVarOut");
				int output = params.getInt("nVarOut");
				int returnValue = params.getIntReturn();

		float	float FuncFloat(
			[in] float fVar,
			[out] float fVarOut);
				float fVar = 1.0;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncFloat(fVar, "fVarOut");
				float output = params.getFloat("fVarOut");
				float returnValue = params.getFloatReturn();

		double	double FuncDouble(
			[in] double dVar,
			[out] double dVarOut);
				double dVar = 1.0;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncDouble(dVar, "dVarOut");
				double output = params.getDouble("dVarOut");
				double returnValue = params.getDoubleReturn();

		char	char FuncChar(
			[in] char cVar,
			[out] char cVarOut);
				char cVar = 'X';
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncChar(cVar, "cVarOut");
				char output = params.getChar("cVarOut");
				char returnValue = params.getCharReturn();

		Object	object FuncObj(
			[in] object oVar,
			[out] object oVarOut);
				List oVar = new LinkedList();
				…
				Book book = new Book();
				…
				oVar.add(book);
				interface_c stub = new interface_c();
				RPCTable table = stub.FuncObj(oVar, “oVarOut”);
				
				List output = (List)table.getObject(“oVarOut”);
				
				List returnValue = (List) table.getObjectReturn();

1 Dimensional	Constrained array	short	void FuncConstrainedShortArray(
			[in] short nRowDim1,
			[in] short nRowDim2,
			[in] short shArrVar[nRowDim1],
			[out] short shArrVarOut[nRowDim2]);
				short[] shArrVar = new short[2];
				shArrVar[0] = 1;
				shArrVar[1] = 10;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncConstrainedShortArray(2, 2, shArrVar, "shArrVarOut");
				short[] output = params.getShortArray("shArrVarOut");

		long	void FuncConstrainedLongArray(
			[in] long nRowDim1,
			[in] long nRowDim2,
			[in] long lArrVar[nRowDim1],
			[out] long lArrVarOut[nRowDim2]);
				int[] lArrVar = new int[2];
				lArrVar[0] = 1;
				lArrVar[1] = 10;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncConstrainedLongArray(2, 2, lArrVar, "lArrVarOut");
				int[] output = params.getIntArray("lArrVarOut");

		int	void FuncConstrainedIntArray(
			[in] int nRowDim1,
			[in] int nArrVar[nRowDim1],
			[in] int nRowDim2,
			[out] int nArrVarOut[nRowDim2]);
				int[] nArrVar = new int[2];
				nArrVar[0] = 1;
				nArrVar[1] = 10;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncConstrainedIntArray(2, nArrVar, 2, "nArrVarOut");
				int[] output = params.getIntArray("nArrVarOut");

		float	void FuncConstrainedFloatArray(
			[in] int nRowDim1,
			[in] float fArrVar[nRowDim1],
			[in] int nRowDim2,
			[out] float fArrVarOut[nRowDim2]);
				float[] fArrVar = new float[2];
				fArrVar[0] = 1.0;
				fArrVar[1] = 10.0;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncConstrainedFloatArray(2, fArrVar, 2, "fArrVarOut");
				float[] output = params.getFloatArray("fArrVarOut");

		double	void FuncConstrainedDoubleArray(
			[in] int nRowDim1,
			[in] double dArrVar[nRowDim1],
			[in] int nRowDim2,
			[out] double dArrVarOut[nRowDim2]);
				double[] dArrVar = new double[2];
				dArrVar[0] = 1.0;
				dArrVar[1] = 10.0;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncConstrainedDoubleArray(2, dArrVar, 2, "dArrVarOut");
				double[] output = params.getDoubleArray("dArrVarOut");

		char	void FuncConstrainedCharArray([in] int nColDim1, [in] char cArrVar[nColDim1], [in] int nColDim2, [out] char cArrVarOut[nColDim2]);	String cArrVar = "データ"; interface_c stub = new interface_c(); RPCTable params = stub.FuncConstrainedCharArray(6, cArrVar, 7, "cArrVarOut"); String output = params.getString("cArrVarOut");

		void	void FuncConstrainedVoidArray(
			[in] int nRowDim1,
			[in] void byteArrVar[nRowDim1],
			[out] int nRowDim2,
			[out] void byteArrVarOut[nRowDim2]);
				byte[] byteVarArr = readFile("X.gif");
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncConstrainedVoidArray(2, byteArrVar, “lOut”, "byteArrVarOut");
				byte[] output = params.getByteArray("byteArrVarOut");

	Fixed array	short	void FuncFixedLengthShortArray(
			[in] short shArrVar[10],
			[out] short shArrVarOut[10]);
				short[] shArrVar = new short[2];
				shArrVar[0] = 1;
				shArrVar[1] = 10;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncFixedLengthShortArray(shArrVar, "shArrVarOut");
				short[] output = params.getShortArray("shArrVarOut");

		long	void FuncFixedLengthLongArray(
			[in] long lArrVar[10],
			[out] long lArrVarOut[10]);
				int[] lArrVar = new int[2];
				lArrVar[0] = 1;
				lArrVar[1] = 10;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncFixedLengthLongArray(lArrVar, "lArrVarOut");
				int[] output = params.getIntArray("lArrVarOut");

		int	void FuncFixedLengthIntArray(
			[in] int nArrVar[10],
			[out] int nArrVarOut[10]);
				int[] nArrVar = new int[2];
				nArrVar[0] = 1;
				nArrVar[1] = 10;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncFixedLengthIntArray(nArrVar, "nArrVarOut");
				int[] output = params.getIntArray("nArrVarOut");

		float	void FuncFixedLengthFloatArray(
			[in] float fArrVar[10],
			[out] float fArrVarOut[10]);
				float[] fArrVar = new float[2];
				fArrVar[0] = 1.0;
				fArrVar[1] = 10.0;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncFixedLengthFloatArray(fArrVar, "fArrVarOut");
				float[] output = params.getFloatArray("fArrVarOut");

		double	void FuncFixedLengthDoubleArray(
			[in] double dArrVar[10],
			[out] double dArrVarOut[10]);
				double[] dArrVar = new double[2];
				dArrVar[0] = 1.0;
				dArrVar[1] = 10.0;
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncFixedLengthDoubleArray(dArrVar, "dArrVarOut");
				double[] output = params.getDoubleArray("dArrVarOut");

		char	void FuncFixedLengthCharArray([in] char cArrVar[10], [out] char cArrVarOut[10]);	String cArrVar = "データ"; interface_c stub = new interface_c(); RPCTable params = stub.FuncFixedLengthCharArray(cArrVar, "cArrVarOut"); String output = params.getString("cArrVarOut");

		void	void FuncFixedLengthVoidArray(
			[in] void byteArrVar[5973],
			[out] void byteArrVarOut[5973]);
				byte[] byteVarArr = readFile("X.gif");
				interface_c stub = new interface_c();
				RPCTable params = stub.FuncFixedLengthVoidArray(byteArrVar, "byteArrVarOut");
				byte[] output = params.getByteArray("byteArrVarOut");

	Null terminated array	char	void FuncNullTerminatedArray(
			[in] char cArrVar[],
			[out] char cArrVarOut[]);
				String cArrVar = "データ";
				interface_c stub = new interface_c();
				RPCTable params =
				stub.FuncNullTerminatedArray(cArrVar,
				"cArrVarOut");
				String output =
				params.getString("cArrVarOut");

2 dimensional	Fixed array	char	void FuncFixedCharArray(
			[in] char sVar[10][20],
			[out] char sVarOut[10][30]);
				String[] sVar = new String[2];
				sVar[0] = "配列 1";
				sVar[1] = "配列 2";
				interface_c stub = new interface_c();
				RPCTable params =
				stub.FuncFixedCharArray(sVar, "sVarOut");
				String[] output =
				params.getStringArray("sVarOut");

	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])
			;
				String[] sVar = new String[2];
				sVar[0] = "配列 1";
				sVar[1] = "配列 2";
				interface_c stub = new interface_c();
				RPCTable params =
				stub.FuncConstrainedCharArray(2, 5, sVar,
				2, 6, "sVarOut");
				String[] output =
				params.getStringArray("sVarOut");

	Null terminated array	char	void FuncNullTerminatedArray(
			[in] char sVar[][],
			[out] char sVarOut[][]);
				String[] sVar = new String[2];
				sVar[0] = "配列 1";
				sVar[1] = "配列 2";
				interface_c stub = new interface_c();
				RPCTable params =
				stub.FuncNullTerminatedArray(sVar,
				"sVarOut");
				String[] output =
				params.getStringArray("sVarOut");

第9章　VB .NET, C# クライアント

この章では、3 層環境で .NET Framework を使用して Visual Basic .NET または C#クライアントを構築する手順を説明します。

この章を読む前に、『サーバ開発者ガイド』および本書の第1章、第2章で説明されている内容をご理解ください。

アーキテクチャ

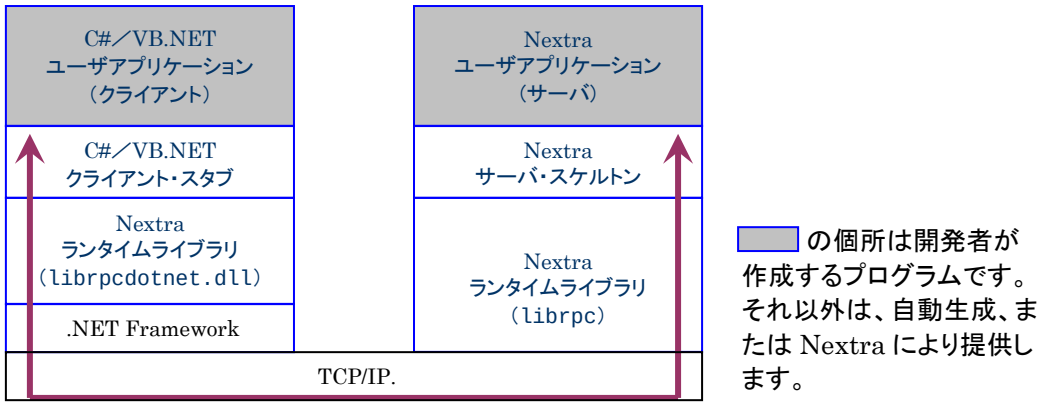

図 9.1：VB .NET, C# 使用時のアーキテクチャ

プラットフォームの必要条件

開発パッケージを、Visual Studio .NET (または .NET Framework SDK)がインストールされている Windows システムにインストールします。ソフトウェアのバージョン情報については、「ソフトウェアの必要条件」を参照してください。

完成した .NET Framework アプリケーションを実行する各 PC に必要な条件は次のとおりです。

- Microsoft .NET Framework Version 4.0 以降 再頒布可能パッケージ
- Nextra ランタイムライブラリ (librpcdotnet.dll)

データタイプマッピング

表 9.1：データタイプマッピング

シンプルデータ型		
IDL	VB .NET Client	C# Client
short	Short	short
int	Integer	int
long	Integer	int
float	Single	float
double	Double	double
char	Char	char
1 次元配列		
IDL	VB .NET Client	C# Client
short[]	Short()	short[]
int[]	Integer()	int[]
long[]	Integer()	int[]
float[]	Single()	float[]
double[]	Double()	double[]
char[]	String	string
void[]	Byte()	byte[]
2 次元配列		
IDL	VB .NET Client	C# Client
char[][]	String()	string[]

- 詳しい使用方法については、「データ型別使用例」を参照してください。

メソッドのパラメータについて

VB .NET の場合、IDL ファイルにて `[in]` と指定されたパラメータはスタブでは `ByVal` が指定され、`[out]` と指定されたパラメータはスタブでは `ByRef` が指定されます。

C# の場合、IDL ファイルにて `[in]` と指定されたパラメータは通常の値渡し、`[out]` と指定されたパラメータはスタブでは `out` が指定されます。そのため、呼び出し側でも `out` を使用してください。

詳しくはこの章の最後にある「データ型別使用例」を参照してください。

返り値

`char`, `short`, `long`, `int`, `float`, `double` の関数リターン返り値の取得方法については、この章の最後にある「データ型別使用例」を参照してください。

.NETクライアント・スタブの生成

クライアント・スタブの生成は、RPC Developer にて生成する方法と、コマンドラインにて生成する方法があります。生成されるファイル名は、VB .NET の場合 *interfacename*_c.vb、C# の場合 *interfacename*_c.cs になります。

RPC Developer にある RPCMake タブにて、クライアント言語として VB .NET または C# を選択してください。または、コマンドラインでは以下のように使用されます。

[VB .NET の場合]

```
> rpcmake -d file.def -c vb.net
```

[C# の場合]

```
> rpcmake -d file.def -c cs
```

スタブとライブラリをプロジェクトに追加

Visual Studio にて、メニューより [プロジェクト]－[参照の追加] を選択し、Nextra のインストールディレクトリの `tcp\bin` 内にある `librpcdotnet.dll` を追加してください。

次に、メニューより [プロジェクト]－[既存項目の追加] を選択し、クライアント・スタブを追加してください。

.NETクライアントの記述方法

Nextra 開発パッケージの「`samples`」ディレクトリ内を参照してください。

以下の Nextra エラーに対する `Exception` をサポートしています。

表 9.2： Nextra エラーに対する Exception

Error No.	Symbol	Exception
4	DCE_NOBROKER	DceNoBrokerException
5	DCE_NOSERVER	DceNoServerException
6	DCE_BADHOST	DceBadHostException
7	DCE_BADSERVHOST	DceBadServerHostException
13	DCE_NOSUCHFUNC	DceNoSuchFuncException
14	DCE_LOCALHOSTUNKN	DceLocalHostUnknownException
17	DCE_NOMEMORY	DceNoMemoryException
24	DCE_PEERERROR	DceConnectionResetByPeerException
25	DCE_LOSTSERVER	DceLostServerException
26	DCE_BADTCPINIT	DceBadTcpInitException
27	DCE_IPCINITBAD	DceBadIpcInitException
28	DCE_IPCCANTCLOSE	DceIpcCantCloseException
31	DCE_CANTFORK	DceCannotForkException
32	DCE_BADPORT	DceBadPortException
33	DCE_NOINTERFACE	DceNoInterfaceException
36	DCE_SIGINT	DceSignalInterruptException
37	DCE_SERVERFAILED	DceServerFailedException
38	DCE_BADINTERFACE	DceBadInterfaceException
43	DCE_MAXCAPACITY	DceMaxCapacityException
45	DCE_FSERROR	DceFileSystemException
46	DCE_RPCTIMEOUT	DceRPCTimeOutException
52	DCE_UNAVAILABLE	DceUnavailableException
58	DCE_MAXRPCECEEDED	DceMaxRPCExceededException
59	DCE_ASYNCRPCNOTFOUND	DceAsyncRPCNotFoundException
61	DCE_CANCELRPCERROR	DceCancelRPCErrorException
64	DCE_ASYNCINPROGRESS	DceAsyncInProgressException
67	DCE_THREADCREATIONFAILED	DceThreadCreationFailedException
68	DCE_THREADLOCKFAILED	DceThreadLockFailedException
69	DCE_UNABLE2PROCESS	DceUnable2ProcessException
その他、オブジェクトクライアントがサポートする Exception		
Server または Broker が QUEUE 溢れの場合		DceQueueFullException
上記の Exception のスーパークラス		DceException

クライアントプログラム内でライブラリ中のクラスにアクセスできるよう、以下の文を記述してください。

[VB .NET の場合]

```
Imports com.inspire.rpc.cleint

Imports com.inspire.rpc.shared
```

[C# の場合]

```
using com.inspire.rpc.client;

using com.inspire.rpc.shared;
```

クライアント環境の設定について、RPC の実行前に以下のようにして環境を設定してください。

[VB .NET の場合]

Environemnt.dce_steenv("*filename*.env")

[C# の場合]

Environemnt.dce_setenv("*filename*.env");

バリアブル・ネームド・サーバの記述について

クライアント・スタブ中各メソッドの 1 番目の引数として、"dce_server"が生成されます。よって、クライアントプログラムからメソッドを使用する場合は、必ず最初の引数に当該サーバ名を指定して呼び出してください。

デディケイテッド・サーバについて

通常の Nextra クライアントでは、スタブに生成された dce_dedDisconnect メソッドをクライアントプログラムから呼び出し、デディケイテッド・サーバ子プロセスを終了してください。あるいは、サーバ環境ファイルに「DCE_SVR_TIMEOUT」属性を指定して子プロセスのタイムアウトによる終了を行ってください。

環境ファイルについて

環境ファイル中には、以下の環境ファイル属性が使えます。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」環境ファイル属性を参照してください。以下は Java クライアント特有の環境ファイル属性です。
* GW_PUTNULL(.NET exclusive)
* GW_TRIM(.NET exclusive)

属性	説明
GW_PUTNULL デフォルト=false	2 次元文字配列の最後に null を付加する。null を追加する場合は”true”を指定。
GW_TRIM デフォルト=false	文字列末尾の空白文字を削除する。削除する場合は”true”を指定。

エンコーディング指定 DCE_LANG について

Nextra を UNIX 上でお使いのユーザは「SJIS」、Windows でお使いのユーザは、指定しないか、または「MS932」と指定してください。

一般的な環境ファイルは、以下のようになります。

```
DCE_BROKER=HOSTNAME1,PORT#1
      [HOSTNAME2,PORT#2]
DCE_CLN_TIMEOUT=60 ←秒
```

サンプルプログラム

開発パッケージの「samples」ディレクトリ内に、単純なデータタイプ、一次元データタイプ、二次元データタイプのサンプルがあります。

データ型別使用例 (VB .NET, C#)

注意事項

“バリアブル・ネームド・サーバ”ではない場合の例です。“バリアブル・ネームド・サーバ”については、「バリアブル・ネームド・サーバの記述について」を参照してください。

rpcmake により生成されたクライアント・スタブのクラス名を「*interface*_c」とし、下記のコードが書かれているものと仮定します。

[VB.NET の場合]

```
Dim stub As New interface_c()
```

[C#の場合]

```
interface_c stub = new interface_c();
```

表 9.3：データ型別使用例

次元	配列	型	IDL ファイル内 関数例	パラメータと返り値の扱い方

Simple		short	short FuncShort(
			[in] short shVar,
			[out] short shVarOut
			);
				[VB]
				Dim shVar As Short = 1
				Dim shVarOut As Short
				Dim ret As Short =
				stub.FuncShort(shVar, shVarOut)
				
				[C#]
				short shVar = 1;
				short shVarOut;
				short ret = stub.FuncShort(shVar, out shVarOut);

		long	long FuncLong(
			[in] long lVar,
			[out] long lVarOut);
				[VB]
				Dim lVar As Integer = 1
				Dim lVarOut As Integer
				Dim ret As Integer
				=_stub.FuncLong(lVar, lVarOut)
				
				[C#]
				int lVar = 1;
				int lVarOut;
				int ret = stub.FuncLong(lVar, out lVarOut);

		int	int FuncInteger(
			[in] int nVar,
			[out] int nVarOut);
				[VB]
				Dim nVar As Integer = 1
				Dim nVarOut As Integer
				Dim ret As Integer
				=_stub.FuncInteger(nVar, nVarOut)
				
				[C#]
				int nVar = 1;
				int nVarOut;
				int ret = stub.FuncInteger(nVar, out nVarOut);

		float	float FuncFloat(
			[in] float fVar,
			[out] float fVarOut);
				[VB]
				Dim fVar As Single = 1.0F
				Dim fVarOut As Single
				Dim ret As Single =
				_stub.FuncFloat(fVar, fVarOut)
				
				[C#]
				float fVar = 1.0;
				float fVarOut;
				float ret = stub.FuncFloat(fVar, out fVarOut);

		double	double FuncDouble(
			[in] double dVar,
			[out] double dVarOut);
				[VB]
				Dim dVar As Double = 1.0
				Dim dVarOut As Double
				Dim ret As Double =
				_stub.FuncDouble(dVar, dVarOut)
				
				[C#]
				double dVar = 1.0;
				double dVarOut;
				double ret = stub.FuncDouble(dVar, out dVarOut);

		char	char FuncChar([in] char cVar, [out] char cVarOut);	[VB] Dim cVar As Char = "X" Dim cVarOut As Char Dim ret As Char = _stub.FuncChar(cVar, cVarOut) [C#] char cVar = 'X'; char cVarOut; char ret = stub.FuncChar(cVar, out cVarOut);

1 Dimensional	Constrained array	short	void FuncConstrainedShortArray(
			[in] short nRowDim1,
			[in] short shArrVar[nRowDim1],
			[in] short nRowDim2,
			[out] short shArrVarOut[nRowDim2]);
				[VB]
				Dim shArrVar(1) As Short
				shArrVar(0) = 1
				shArrVar(1) = 10
				Dim shArrVarOut() As Short
				stub.FuncConstrainedShortArray(2,shArrVar, 2,shArrVarOut)
				
				[C#]
				short[] shArrVar = new short[2];
				shArrVar[0] = 1;
				shArrVar[1] = 10;
				short[] shArrVarOut;
				stub.FuncConstrainedShortArray(2,shArrVar,2,outshArrVarOut);

		long	void FuncConstrainedLongArray(
			[in] long nRowDim1,
			[in] long lArrVar[nRowDim1],
			[in] long nRowDim2,
			[out] long lArrVarOut[nRowDim2]);
				[VB]
				Dim lArrVar(1) As Integer
				lArrVar(0) = 1
				lArrVar(1) = 10
				Dim lArrVarOut() As Integer
				stub.FuncConstrainedLongArray(2,lArrVar,2,lArrVarOut)
				
				[C#]
				int[] lArrVar = new int[2];
				lArrVar[0] = 1;
				lArrVar[1] = 10;
				int[] lArrVarOut;
				stub.FuncConstrainedLongArray(2,lArrVar,2,outArrVarOut);

		int	void FuncConstrainedIntArray(
			[in] int nRowDim1,
			[in] int nArrVar[nRowDim1],
			[in] int nRowDim2,
			[out] int nArrVarOut[nRowDim2]);
				[VB]
				Dim nArrVar(1) As Integer
				nArrVar(0) = 1
				nArrVar(1) = 10
				Dim nArrVarOut() As Integer
				stub.FuncConstrainedIntArray(2,nArrVar,2,nArrVarOut)
				
				[C#]
				int[] nArrVar = new int[2];
				nArrVar[0] = 1;
				nArrVar[1] = 10;
				int[] nArrVarOut;
				stub.FuncConstrainedIntArray(2,nArrVar,2,out nArrVarOut);

		float	void FuncConstrainedFloatArray(
			[in] int nRowDim1,
			[in] float fArrVar[nRowDim1],
			[in] int nRowDim2,
			[out] float fArrVarOut[nRowDim2]);
				[VB]
				Dim fArrVar(1) As Single
				fArrVar(0) = 1.0
				fArrVar(1) = 10.0
				Dim fArrVarOut() As Single
				stub.FuncConstrainedFloatArray(2,fArrVar,2,fArrVarOut)
				
				[C#]
				float[] fArrVar = new float[2];
				fArrVar[0] = 1.0;
				fArrVar[1] = 10.0;
				float[] fArrVarOut;
				stub.FuncConstrainedFloatArray(2,fArrVar,2,out fArrVarOut);

		double	void FuncConstrainedDoubleArray(
			[in] int nRowDim1,
			[in] double dArrVar[nRowDim1],
			[in] int nRowDim2,
			[out] double
			dArrVarOut[nRowDim2]);
				[VB]
				Dim dArrVar(1) As Double
				dArrVar(0) = 1.0
				dArrVar(1) = 10.0
				Dim dArrVarOut() As Double
				stub.FuncConstrainedDoubleArray(2,dArrVar,2,dArrVarOut)
				
				[C#]
				double[] dArrVar = new double[2];
				dArrVar[0] = 1.0;
				dArrVar[1] = 10.0;
				double[] dArrVarOut;
				stub.FuncConstrainedDoubleArray(2,dArrVar,2,out dArrVarOut);

		char	void FuncConstrainedCharArray([in] int nColDim1, [in] char cArrVar[nColDim1], [in] int nColDim2, [out] char cArrVarOut[nColDim2]);	[VB] Dim cArrVar As String = "データ" Dim cArrVarOut As String stub.FuncConstrainedCharArray(6,cArrVar,7,cArrVarOut) [C#] string cArrVar = "データ"; string cArrVarOut; stub.FuncConstrainedCharArray(6,cArrVar,7,out cArrVarOut);

		void	void FuncConstrainedVoidArray(
			[in] int nRowDim1,
			[in] void byteArrVar[nRowDim1],
			[out] int nRowDim2,
			[out] void
			byteArrVarOut[nRowDim2]);
				[VB]
				Dim byteVarArr() As Byte =
				readFile("X.gif")
				Dim nRowDim2 As Integer
				Dim byteArrVarOut() As Byte
				stub.FuncConstrainedVoidArray(_byteArrVar.Length, byteArrVar, out nRowDim2, out byteArrVarOut)
				
				[C#]
				byte[] byteVarArr = readFile("X.gif");
				int nRowDim2;
				byte[] byteArrVarOut;
				stub.FuncConstrainedVoidArray(byteArrVar.Length, byteArrVar,out nRowDim2, out byteArrVarOut);

	Fixed array	short	void FuncFixedLengthShortArray(
			[in] short shArrVar[2],
			[out] short shArrVarOut[10]);
				[VB]
				Dim shArrVar(1) As Short
				shArrVar(0) = 1
				shArrVar(1) = 10
				Dim shArrVarOut() As Short
				stub.FuncFixedLengthShortArray(shArrVar, shArrVarOut)
				
				[C#]
				short[] shArrVar = new short[2];
				shArrVar[0] = 1;
				shArrVar[1] = 10;
				short[] shArrVarOut;
				stub.FuncFixedLengthShortArray(shArrVar,out shArrVarOut);

		long	void FuncFixedLengthLongArray(
			[in] long lArrVar[2],
			[out] long lArrVarOut[10]);
				[VB]
				Dim lArrVar(1) As Integer
				lArrVar(0) = 1
				lArrVar(1) = 10
				Dim lArrVarOut() As Integer
				stub.FuncFixedLengthLongArray(lArrVar, lArrVarOut)
				
				[C#]
				int[] lArrVar = new int[2];
				lArrVar[0] = 1;
				lArrVar[1] = 10;
				int[] lArrVarOut;
				stub.FuncFixedLengthLongArray(lArrVar,
				out lArrVarOut);

		int	void FuncFixedLengthIntArray(
			[in] int nArrVar[2],
			[out] int nArrVarOut[10]);
				[VB]
				Dim nArrVar(1) As Integer
				nArrVar(0) = 1
				nArrVar(1) = 10
				Dim nArrVarOut() As Integer
				stub.FuncFixedLengthIntArray(nArrVar, nArrVarOut)
				
				[C#]
				int[] nArrVar = new int[2];
				nArrVar[0] = 1;
				nArrVar[1] = 10;
				int[] nArrVarOut;
				stub.FuncFixedLengthIntArray(nArrVar,
				out nArrVarOut);

		float	void FuncFixedLengthFloatArray(
			[in] float fArrVar[2],
			[out] float fArrVarOut[10]);
				[VB]
				Dim fArrVar(1) As Single
				fArrVar(0) = 1.0
				fArrVar(1) = 10.0
				Dim fArrVarOut() As Single
				stub.FuncFixedLengthFloatArray(fArrVar, fArrVarOut);
				
				[C#]
				float[] fArrVar = new float[2];
				fArrVar[0] = 1.0;
				fArrVar[1] = 10.0;
				float[] fArrVarOut;
				stub.FuncFixedLengthFloatArray(fArrVar, out fArrVarOut);

		double	void FuncFixedLengthDoubleArray(
			[in] double dArrVar[2],
			[out] double dArrVarOut[10]);
				[VB]
				Dim dArrVar(1) As Double
				dArrVar(0) = 1.0
				dArrVar(1) = 10.0
				Dim dArrVarOut() As Double
				stub.FuncFixedLengthDoubleArray(dArrVar, dArrVarOut)
				
				[C#]
				double[] dArrVar = new double[2];
				dArrVar[0] = 1.0;
				dArrVar[1] = 10.0;
				double[] dArrVarOut;
				stub.FuncFixedLengthDoubleArray(dArrVar,out dArrVarOut);

		char	void FuncFixedLengthCharArray([in] char cArrVar[6], [out] char cArrVarOut[10]);	[VB] Dim cArrVar As String = "データ" Dim cArrVarOut As String stub.FuncFixedLengthCharArray(cArrVar, cArrVarOut) [C#] string cArrVar = "データ"; string cArrVarOut; stub.FuncFixedLengthCharArray(cArrVar, out cArrVarOut);

		void	void FuncFixedLengthVoidArray(
			[in] void byteArrVar[5973],
			[out] void byteArrVarOut[5973]);
				[VB]
				Dim byteVarArr() As Byte
				=readFile("X.gif")
				Dim byteArrVarOut() As Byte
				stub.FuncFixedLengthVoidArray(byteArrVar,byteArrVarOut)
				
				[C#]
				byte[] byteVarArr = readFile("X.gif");
				byte[] byteArrVarOut;
				stub.FuncFixedLengthVoidArray(byteArrVar, out byteArrVarOut);

	Null terminated array	char	void FuncNullTerminatedArray([in] char cArrVar[], [out] char cArrVarOut[]);	[VB] Dim cArrVar As String = "データ" Dim cArrVarOut As String stub.FuncNullTerminatedArray(cArrVar, cArrVarOut) [C#] string cArrVar = "データ"; string cArrVarOut; stub.FuncNullTerminatedArray(cArrVar, out cArrVarOut);

2 dimensional	Fixed array	char	void FuncFixedCharArray(
			[in] char sVar[2][5],
			[out] char sVarOut[10][30]);
				[VB]
				Dim sVar(1) As String
				sVar(0) = "配列 1"
				sVar(1) = "配列 2"
				Dim sVarOut() As String
				stub.FuncFixedCharArray(sVar, sVarOut)
				
				[C#]
				string[] sVar = new string[2];
				sVar[0] = "配列 1";
				sVar[1] = "配列 2";
				string[] sVarOut;
				stub.FuncFixedCharArray(sVar, out sVarOut);

	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]);
				[VB]
				Dim sVar(1) As String
				sVar(0) = "配列 1"
				sVar(1) = "配列 2"
				Dim sVarOut() As String
				stub.FuncConstrainedCharArray(2,5,sVar,2,6,sVarOut)
				
				[C#]
				string[] sVar = new string[2];
				sVar[0] = "配列 1";
				sVar[1] = "配列 2";
				string[] sVarOut;
				stub.FuncConstrainedCharArray(2, 5, sVar, 2, 6,out sVarOut);

	Null terminated array	char	void FuncNullTerminatedArray(
			[in] char sVar[][],
			[out] char sVarOut[][]);
				[VB]
				Dim sVar(1) As String
				sVar(0) = "配列 1"
				sVar(1) = "配列 2"
				Dim sVarOut() As String
				stub.FuncNullTerminatedArray(sVar, sVarOut)
				
				[C#]
				string[] sVar = new string[2];
				sVar[0] = "配列 1";
				sVar[1] = "配列 2";
				string[] sVarOut;
				stub.FuncNullTerminatedArray(sVar, out sVarOut);

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
クライアント開発者ガイド

2015 年 11 月 08 日　v6.5 1st Edition
2011 年 09 月 15 日　v6 1st Edition
2010 年 11 月 15 日　Delphi クライアントの変更
2010 年 9 月 9 日　C# librpcdotnet.dll 対応
2009 年 1 月 16 日　オブジェクト クライアントの環境ファイル属性、及び Exception の追加
2008 年 9 月 8 日　v5 2nd Edition
2008 年 1 月 31 日　PB クライアントの変更
2007 年 4 月 17 日　第 2 版発行
2004 年 12 月 8 日　Java、VB .NET, C# クライアントの環境ファイル属性の追加
2004 年 7 月 10 日　「第 9 章 Delphi クライアント」追加
2003 年 7 月 22 日　「第 8 章 VB.NET, C#クライアント」追加
2003 年 4 月 18 日　初版発行

著者　Inspire International Inc.