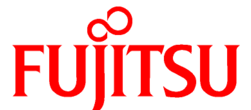


FUJITSU Software
Interstage Application Server

アプリケーション作成ガイド (コンポーネントトランザクションサービス編)

Windows(32)/Solaris(32)/Linux(32)

B1WS-1047-04Z0(00)
2014年10月

まえがき

本書の目的

本書は、コンポーネントトランザクションサービスを利用して分散アプリケーション開発を行うために必要なプログラミングの方法と、業務運用を実現するためのプログラミングの手法、手順、および定義を説明しています。
本書は、アプリケーションの作成、運用を行う方を対象にしています。

前提知識

本書を読む場合、以下の知識が必要です。

- C言語に関する基本的な知識
- C++言語に関する基本的な知識
- COBOLに関する基本的な知識
- Java言語に関する基本的な知識
- インターネットに関する基本的な知識
- オブジェクト指向技術に関する基本的な知識
- 分散オブジェクト技術(CORBA)に関する基本的な知識
- リレーショナルデータベースに関する基本的な知識
- 使用するOSに関する基本的な知識

本書の構成

本書は以下の構成になっています。

第1章 分散アプリケーションを作成するための基礎知識
アプリケーションを作成する上で必要な基礎知識を説明しています。

第2章 サーバアプリケーションの作成(C言語)
C言語を使用したサーバアプリケーションの作成方法について説明しています。

第3章 サーバアプリケーションの作成(C++言語)
C++言語を使用したサーバアプリケーションの作成方法について説明しています。

第4章 サーバアプリケーションの作成(COBOL)
COBOLを使用したサーバアプリケーションを作成する方法について説明しています。

第5章 Interstageの特徴的な機能
Interstageの特徴的な機能について説明しています。

第6章 C++言語の提供クラス
IDL定義を元に生成されるクラスおよびコンポーネントトランザクションサービスが提供するクラスについて説明しています。

第7章 スナップショット機能
スナップショットの出力例について説明しています。

第8章 ワークユニットが提供する運用支援機能の使用方法
ワークユニットが提供する運用支援機能について説明します。

付録A ワークユニット定義の記述形式
ワークユニット定義の設定について説明しています。

付録B トランザクションアプリケーションのサンプルプログラム(基本編)
トランザクションアプリケーションの基本的なサンプルプログラムについて説明しています。

付録C トランザクションアプリケーションのサンプルプログラム(各種データ型編)
トランザクションアプリケーションにおいて各種データ型を使用したサンプルプログラムについて説明しています。

付録D トランザクションアプリケーションのサンプルプログラム（サーバアプリケーション間連携編）
サーバアプリケーション間連携を行うサンプルプログラムについて説明しています。

付録E トランザクションアプリケーションのサンプルプログラム（プロセスバインド機能編）
プロセスバインド機能を使用したサンプルプログラムについて説明しています。

輸出許可

本ドキュメントを輸出または第三者へ提供する場合は、お客様が居住する国および米国輸出管理関連法規等の規制をご確認のうえ、必要な手続きをおとりください。

著作権

Copyright 1999-2014 FUJITSU LIMITED

2014年10月 第4版
2012年8月 初版

目　次

第1章 分散アプリケーションを作成するための基礎知識....1
1.1 アプリケーションの作成の流れ....1
1.2 基本的な運用パターン....2
1.3 トランザクションアプリケーション作成上の注意点....3

第2章 サーバアプリケーションの作成(C言語)....9
2.1 サーバアプリケーションの開発....9
2.2 IDLファイルの作成....13
2.3 サーバアプリケーションのソースの作成....18
2.4 ソースのコンパイル・リンク....19
2.4.1 IDLファイルのコンパイル....19
2.4.2 スタブとクライアントアプリケーションのソースとのコンパイル・リンク....20
2.4.3 スケルトンとサーバアプリケーションのソースとのコンパイル・リンク....20
2.4.4 スレッドモードのアプリケーションのコンパイルとリンク....25
2.4.5 プロセスモードのアプリケーションのコンパイルとリンク....26
2.4.6 継承について....29
2.5 ワークユニット定義の作成....31
2.6 アプリケーションの登録....32
2.7 アプリケーションのテスト....33

第3章 サーバアプリケーションの作成(C++言語)....39
3.1 サーバアプリケーションの開発....39
3.2 IDLファイルの作成....43
3.3 サーバアプリケーションのソースの作成....48
3.4 ソースのコンパイル・リンク....51
3.4.1 IDLファイルのコンパイル....51
3.4.2 スタブとクライアントアプリケーションのソースとのコンパイル・リンク....52
3.4.3 スケルトンとサーバアプリケーションのソースとのコンパイル・リンク....52
3.4.4 スレッドモードのアプリケーションのコンパイルとリンク....56
3.4.5 プロセスモードのアプリケーションのコンパイルとリンク....58
3.4.6 継承について....61
3.5 ワークユニット定義の作成....64
3.6 アプリケーションの登録....64
3.7 アプリケーションのテスト....65

第4章 サーバアプリケーションの作成(COBOL)....70
4.1 サーバアプリケーションの開発....70
4.2 IDLファイルの作成....74
4.3 サーバアプリケーションのソースの作成....79
4.4 ソースのコンパイル・リンク....80
4.4.1 IDLファイルのコンパイル....80
4.4.2 スタブとクライアントアプリケーションのソースとのコンパイル・リンク....80
4.4.3 スケルトンとサーバアプリケーションのソースとのコンパイル・リンク....80
4.4.4 スレッドモードのアプリケーションのコンパイルとリンク....83
4.4.5 プロセスモードのアプリケーションのコンパイルとリンク....84
4.4.6 継承について....86
4.5 ワークユニット定義の作成....88
4.6 アプリケーションの登録....89
4.7 アプリケーションのテスト....90

第5章 Interstageの特徴的な機能....97
5.1 セションIDを採番するためのトランザクションアプリケーションの作成....97
5.1.1 セションID....97
5.1.2 プログラミングの流れ....97
5.1.2.1 IDL定義....97

5.1.2.2 ワークユニット定義....97
5.1.2.3 プログラミングイメージ....97
5.2 プロセスバインド機能を使用したトランザクションアプリケーションの作成....98
5.2.1 概要....98
5.2.2 プログラミングの流れ....99
5.2.2.1 プログラミングの概要....99
5.2.2.2 C言語・COBOLの場合の通信イメージ....102
5.2.2.3 C++言語の場合の通信イメージ....103
5.2.3 注意点....104
5.2.3.1 クライアント異常を考慮したトランザクションアプリケーション....104
5.2.3.2 その他の注意点....106
5.3 セション情報管理機能を使用したトランザクションアプリケーションの作成....106
5.3.1 概要....106
5.3.2 セション情報域を操作するオブジェクト....107
5.3.2.1 コンポーネントトランザクションサービスの環境定義....107
5.3.2.2 プログラミングの流れ....107
5.3.2.3 提供インクルード....121
5.3.2.4 提供ライブラリ....121
5.3.2.5 IDL....122
5.3.3 セション情報管理の事象通知リスナオブジェクト....125
5.3.3.1 プログラミングの流れ....125
5.3.3.2 事象通知リスナ作成に使用する提供インクルード....130
5.3.3.3 事象通知リスナ作成に使用する提供ライブラリ....131
5.3.3.4 COBOLを使用する場合の注意事項....133
5.3.4 セション情報管理の注意事項....135

第6章 C++言語の提供クラス....136
6.1 IDL定義を元に生成されるクラス....136
6.1.1 クラスマッピング....136
6.1.2 構造データ型のマッピング形態....138
6.1.2.1 構造体型....138
6.1.2.2 配列型....142
6.1.2.3 シーケンス型....146
6.2 標準提供クラス....159
6.2.1 String_var, WString_varクラス....160
6.2.2 可変長データ領域獲得関数....162
6.2.3 可変長データ領域解放関数....163

第7章 スナップショット機能....164
7.1 機能概要....164
7.2 ファイル出力のスナップショット....164
7.3 メモリ出力のスナップショット....164
7.3.1 ロギング情報の取得手順....164
7.3.2 スナップショット取得/参照のためのコマンド....165
7.3.3 取得情報を格納するメモリ....165
7.3.4 ロギング情報を取得できるワークユニットの数....166
7.4 スナップショット情報の出力例....166
7.4.1 パラメタ情報の出力例....167
7.4.1.1 long型の表示....167
7.4.1.2 unsigned long型の表示....168
7.4.1.3 short型の表示....168
7.4.1.4 unsigned short型の表示....168
7.4.1.5 long long型の表示....168
7.4.1.6 float型の表示....168
7.4.1.7 double型の表示....168
7.4.1.8 long double型の表示....169
7.4.1.9 char型の表示....169
7.4.1.10 wchar型の表示....169

7.4.1.11 octet型の表示....169
7.4.1.12 boolean型の表示....169
7.4.1.13 string型の表示....169
7.4.1.14 wstring型の表示....170
7.4.1.15 sequence型データ(要素に基本型)の表示....170
7.4.1.16 struct型データの表示....171
7.4.1.17 array型データの表示....171
7.4.2 ユーザ例外情報の出力例....173
7.4.2.1 例外の表示....173

第8章 ワークユニットが提供する運用支援機能の使用方法....174
8.1 ワークユニット出口機能の使用方法....174
8.1.1 概要....174
8.1.2 プログラミングの流れ....174
8.1.3 ソースのコンパイル・リンク....178
8.1.4 注意事項....178
8.2 プロセス回収出口機能の使用方法....179
8.2.1 概要....179
8.2.2 プログラミングの流れ....179
8.2.3 ソースのコンパイル・リンク....183
8.2.4 注意事項....183
8.3 ワークユニットプロセス情報通知機能の使用方法....184
8.3.1 概要....184
8.3.2 プログラミングの流れ....184
8.3.3 注意事項....186
8.4 ユーティリティワークユニットのプロセス停止出口機能の使用方法....186
8.4.1 概要....186
8.4.2 プログラミングの流れ....186
8.4.3 ソースのコンパイル・リンク....189
8.4.4 注意事項....189

付録A ワークユニット定義の記述形式....190
A.1 常駐機能....190
A.2 非常駐機能....192
A.3 マルチオブジェクト常駐機能....195
A.4 複数リソースへのアクセス....198

付録B トランザクションアプリケーションのサンプルプログラム(基本編)....202
B.1 サーバアプリケーション....202
B.1.1 ファイル構成....202
B.1.2 コンパイルおよびワークユニット定義の登録....204
B.1.3 ワークユニットの起動....206
B.1.4 注意点....206
B.2 CORBAクライアントアプリケーション....207
B.2.1 ファイル構成....207
B.2.2 コンパイル....208
B.2.3 クライアントアプリケーションの実行....209
B.2.4 注意点....209

付録C トランザクションアプリケーションのサンプルプログラム(各種データ型編)....211
C.1 概要....211
C.2 ファイル構成....211
C.2.1 ディレクトリ構成....212
C.2.2 ファイル構成....212
C.3 アプリケーションのコンパイル....218
C.3.1 IDL定義ファイルのコンパイル....219
C.3.2 クライアント／サーバアプリケーションのコンパイル....220
C.4 ワークユニットの起動....222

C.4.1 IDLインタフェース情報の登録......222
C.4.2 ネーミングサービスへの登録......222
C.4.3 ワークユニット定義の登録......223
C.4.4 ワークユニットの起動......223
C.5 クライアントアプリケーションの実行......223
C.6 注意点......223

付録D トランザクションアプリケーションのサンプルプログラム(サーバアプリケーション間連携編)......225
D.1 概要......225
D.2 ファイル構成......225
D.2.1 ディレクトリ構成......226
D.2.2 ファイル構成......226
D.3 アプリケーションのコンパイル......230
D.3.1 IDL定義ファイルのコンパイル......230
D.3.2 スタブ／スケルトンファイルの修正......232
D.3.3 クライアント／サーバアプリケーションのコンパイル......233
D.4 ワークユニットの起動......234
D.4.1 IDLインタフェース情報の登録......234
D.4.2 ネーミングサービスへの登録......234
D.4.3 ワークユニット定義の登録......235
D.4.4 ワークユニットの起動......236
D.5 クライアントアプリケーションの実行......236
D.6 注意点......236

付録E トランザクションアプリケーションのサンプルプログラム(プロセスバインド機能編)......238
E.1 概要......238
E.2 ファイル構成......242
E.2.1 ディレクトリ構成......242
E.2.2 ファイル構成......243
E.3 アプリケーションのコンパイル......246
E.3.1 IDL定義ファイルのコンパイル......246
E.3.2 スタブ／スケルトンファイルの修正......247
E.3.3 クライアント／サーバアプリケーションのコンパイル......248
E.4 ワークユニットの起動......249
E.4.1 IDLインタフェース情報の登録......250
E.4.2 ネーミングサービスへの登録......250
E.4.3 ワークユニット定義の登録......250
E.4.4 ワークユニットの起動......251
E.5 クライアントアプリケーションの実行......251
E.6 注意点......251

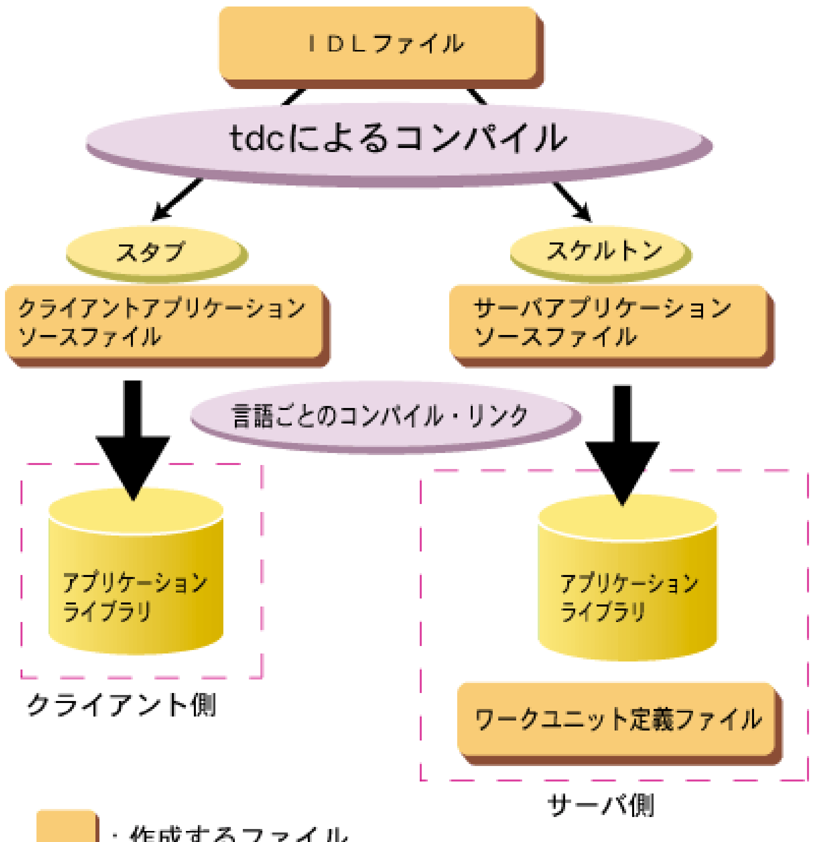
索引......252

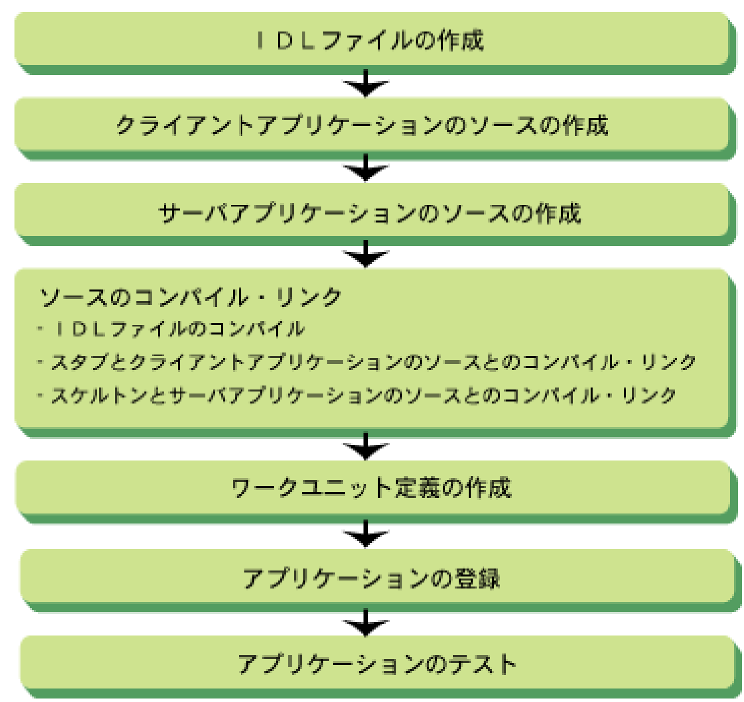
第1章 分散アプリケーションを作成するための基礎知識

コンポーネントトランザクションサービスを利用したアプリケーションを作成する上で必要な基礎知識を説明します。

1.1 アプリケーションの作成の流れ

ローカルトランザクション運用を行うために必要なアプリケーションの作成の流れを以下に示します。





注意

スタブとスケルトンは、必ず同一のIDLファイルから生成されたものを使用してください。

アプリケーションを作成する上で必要な内容については、“OLTPサーバ運用ガイド”を参照してください。

クライアントとサーバのインタフェース情報が不一致となった場合には、コンポーネントトランザクションサービスの動作が不定となります。インタフェース情報チェック機能を使用することを推奨します。インタフェース情報チェック機能の詳細は、“OLTPサーバ運用ガイド”の“インタフェース情報チェック機能を使用した運用”を参照してください。

1.2 基本的な運用パターン

クライアント側とサーバ側でデータをやり取りする場合のプロトコルは、CORBAで規定されているIIOPが使用できます。代表的な運用パターンとして以下について説明します。

- CORBAクライアントとローカルトランザクションアプリケーションの連携

CORBAクライアントとローカルトランザクションアプリケーションの連携

CORBAの分散オブジェクト環境でのクライアントと、サーバシステム内でデータベース管理システムに対してトランザクション運用を行うアプリケーションが通信します。
アプリケーションが、1つのデータベースを利用する業務に適しています。

Windows32 Solaris32

サーバアプリケーションを作成できる言語は、C言語、C++言語、COBOLです。

Linux32

サーバアプリケーションを作成できる言語は、C言語、C++言語です。

クライアントアプリケーションの作成

クライアント側のアプリケーションは、CORBAクライアントアプリケーションとして作成します。詳細については、“アプリケーション作成ガイド(CORBAサービス編)”を参照してください。
なお、コンポーネントトランザクションサービスを使用する際に、アプリケーションが例外復帰した場合の対処については、“メッセージ集”の“コンポーネントトランザクションサービスの例外情報”を参照してください。

1.3 トランザクションアプリケーション作成上の注意点

トランザクションアプリケーションを作成する上で、注意すべき点があります。これら作成上の誤りによりInterstageまたはアプリケーションが動作不良となることがあります。

- データ領域の獲得／解放について
- シグナルについて
- コンパイルとリンクについて
- インタフェース情報が異なる場合に動作不良となる
- アプリケーションで受信した文字列型データが不完全となる
- アプリケーション間連携時にアプリケーションが動作不良となる
- プロセスバインド機能を使用した場合、レスポンスが悪くなる場合がある
- 標準コード変換(FSUNiconv)の使用について

以下の機能を使用して、これらの問題回避または検証を行うことをお勧めします。

1) データ領域の獲得／解放について

以下のデータ型を使用する場合は、各データ型に応じた専用の関数により領域獲得を行う必要があります。専用の関数で獲得していない場合は、アプリケーション動作時にメモリアクセスエラーを引き起こす要因となります。また、クライアントおよびスタブ内で動的に獲得した領域は、不要になった時点で各データ型に応じた解放関数により、明示的な解放操作が基本的に必要です。解放漏れの場合、アプリケーションプロセスでのメモリリークの要因となります。

- 文字列型、ワイド文字列型、シーケンス型、構造体、配列

  以下のデータ型を使用する場合、クライアントアプリケーションのinパラメタ、inoutパラメタ、およびサーバアプリケーションのoutパラメタ、inoutパラメタ、復帰値として、NULLポインタは設定できません。誤って設定した場合、メモリアクセスエラーを招く要因となります。

- 文字列型、シーケンス型、構造体、共用体、配列

  COBOLアプリケーションにおいて以下の型のデータ設定・取得操作は、必ず専用APIを使用してください。各APIの詳細については、“リファレンスマニュアル(API編)”を参照してください。

データ型	データ設定API	データ取得API
文字列型	TDSTRINGSET	TDSTRINGGET
ワイド文字列型	TDWSTRINGSET	TDWSTRINGGET
シーケンス型	TDSEQUENCEELEMENTSET	TDSEQUENCEELEMENTGET

2) シグナルについて

トランザクションアプリケーションでは、シグナル処理を使用できません。

シグナルを受信した場合、アプリプロセスの異常終了などの誤動作となる可能性があります。なお、明にシグナル処理を行わない場合でも、使用する連携製品のライブラリなどでシグナル処理を行う可能性があるため、注意してください。

3) コンパイルとリンクついて

- トランザクションアプリケーションには、サーバアプリケーションプロセスの形態として「プロセスモード」と「スレッドモード」の2つのモードがあります。各モード用のライブラリが提供されているため、リンク時は、モードに応じたライブラリを指定してください。
  実行モジュールにおいて、プロセスモード用ライブラリとスレッドモード用ライブラリを混在してリンクすると、アプリケーションの動作が不定となり、場合によっては、アプリケーションが異常終了する可能性があります。

- Linux for Intel64(32ビット互換)でサンプルプログラムのコンパイル／リンクを行う場合は、gcc/g++コマンド実行時に「-m32 -mtune=i386」オプションを指定する必要があります。Makefileのgcc/g++コマンドに指定するオプションを修正してください。

4) インタフェース情報が異なる場合に動作不良となる

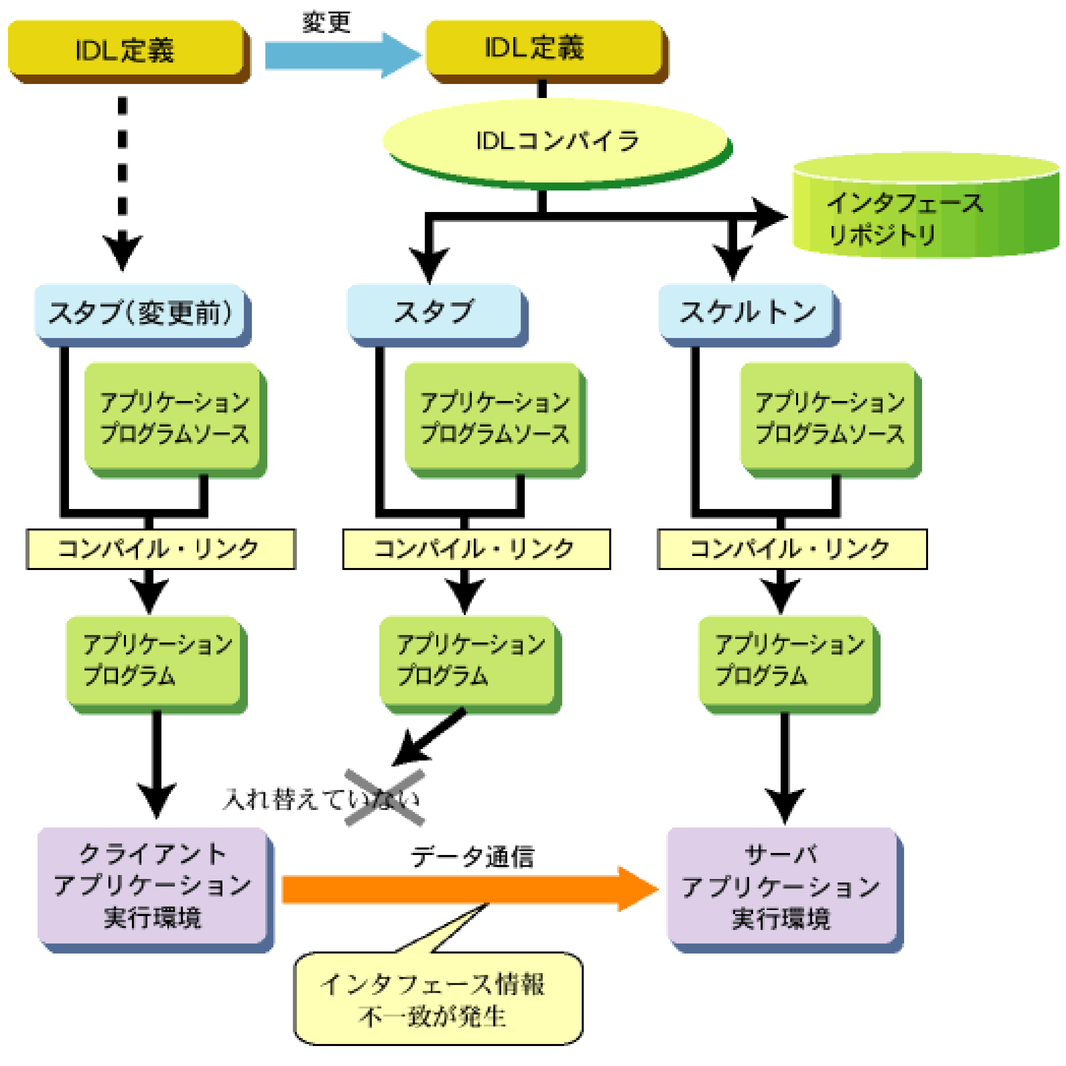
クライアントアプリケーションを作成する際に、IDLコンパイラでIDL定義をコンパイルすることによりスタブファイルを生成します。また、CORBA通信を可能とするために、トランザクションアプリケーションが動作するサーバが参照するインタフェースリポジトリにインタフェース情報を登録します。
このとき使用するIDL定義は、本来同じものを使用しなければなりません。
しかし、下記のような場合、これらの関係に差異が生じることがあります。

- IDL定義の修正を要するアプリケーションを変更したが、一部の操作端末でクライアントアプリケーションを入れ替えていない場合

上記条件でクライアントから処理要求を行うと、Interstageが下記の現象により動作不良となります。

- サーバアプリケーションで受信した、クライアントアプリケーションからの要求データが不当な値となる
- メモリ不足エラーが発生する
- 処理要求が無応答となる
- Interstageが異常終了する

これを回避するため、Interstageではインタフェース情報チェック機能を提供します。
インタフェース情報チェック機能については、“OLTPサーバ運用ガイド”の“インタフェース情報チェック機能を使用した運用”を参照してください。



IDL不一致の発生する環境

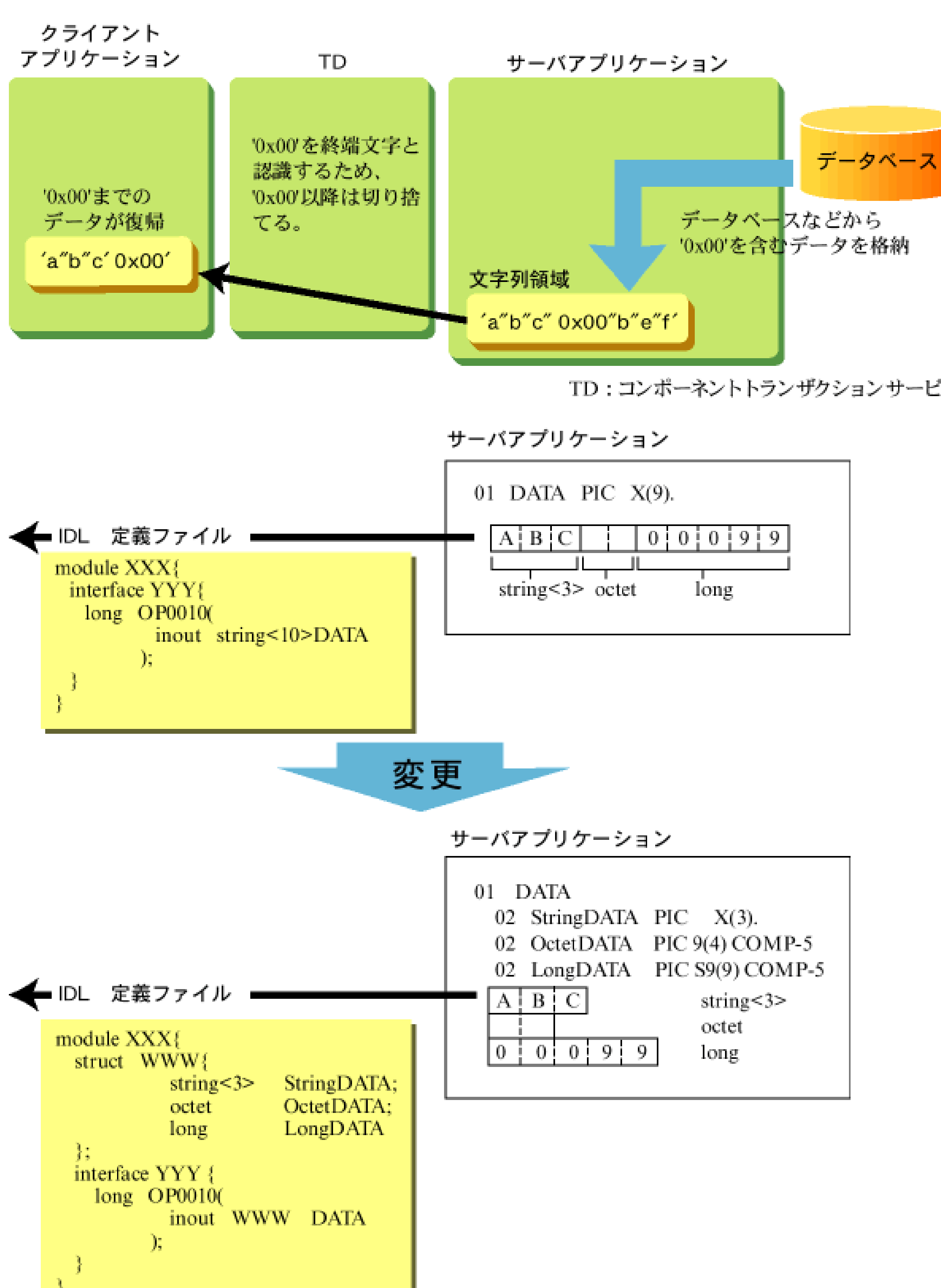
5) アプリケーションで受信した文字列型データが不完全となる

クライアントアプリケーションからサーバアプリケーションへ、またはサーバアプリケーションからクライアントアプリケーションへ処理要求を行った場合に、IDL定義に文字列型で定義をしたデータが途切れることがあります。

これは文字列型と定義されているデータに0x00が含まれているために、Interstageのデータ解析処理が、0x00を終端文字と認識することにより、以降のデータを不要と判断して切り捨てるために発生します。

本現象が発生した場合は、スナップショットでデータを確認するなどして、アプリケーションを修正してください。

本現象となる処理イメージ図とプログラム例を以下に示します。



上記の理由より、データの切り捨てが発生した場合は、IDL定義にて文字列型で定義するのではなく、構造体型として各データ項目を定義するなどし、現象を回避してください。

6) アプリケーション間連携時にアプリケーションが動作不良となる

サーバアプリケーション間連携時には、以下の処理が考えられます。

- クライアントからの要求データを中継アプリケーションで受信し、さらにサーバアプリケーションへデータを通知する

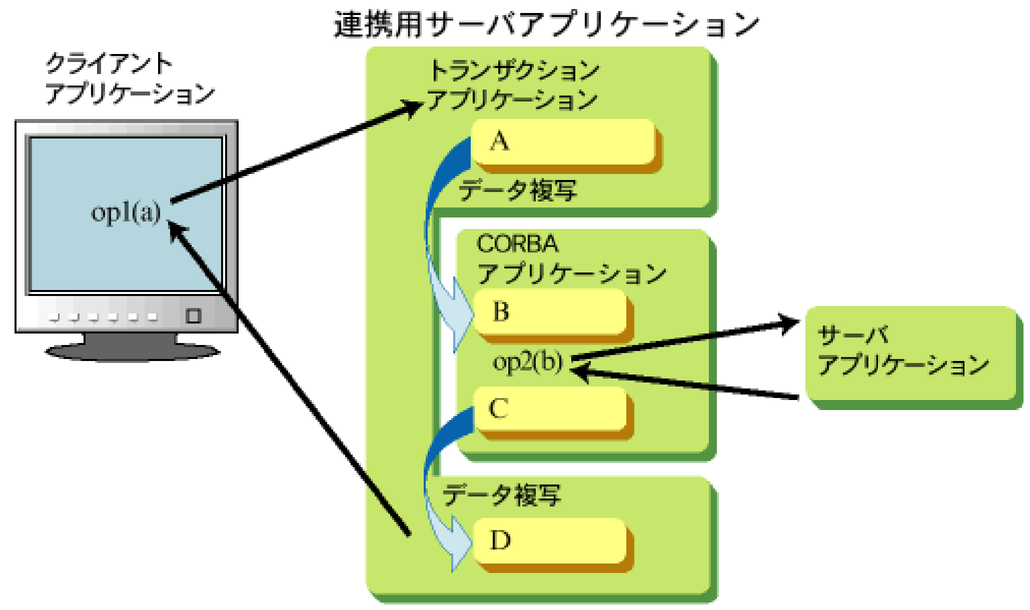
中継アプリケーションは、クライアントとなるCORBAアプリケーションからはトランザクションアプリケーションとして、中継先のトランザクションアプリケーションからはCORBAアプリケーションとして位置づけられます。

このとき、クライアントからのデータを受信したデータは、コンポーネントトランザクションサービスが管理する領域に格納されます。このデータをサーバアプリケーションへ通知する場合は、CORBAサービスが管理する領域に一度複写する必要があります。

これを行わずコンポーネントトランザクションサービスが管理する領域を、CORBAサービスのリクエストのパラメタに設定すると、以下の現象が発生します。

- アプリケーションに通知するデータが不当な値となる
- アプリケーションが異常終了する

下図に示すように、連携用サーバアプリケーションは、トランザクションアプリケーションとCORBAアプリケーションから構成されます。これらは同一オブジェクト上で動作しますが、トランザクションアプリケーション論理で獲得した領域(A)を、CORBAアプリケーションで発行するオペレーション(OP2)のパラメタに指定することはできません。必ずCORBAアプリケーション論理で獲得した領域(B)に複写し、オペレーションのパラメタに指定してください。



A : td_XXX_alloc()で獲得した領域
B : CORBA_XXX_alloc()で獲得した領域
C : CORBA_XXX_alloc()で獲得した領域
D : td_XXX_alloc()で獲得した領域

7) プロセスバインド機能を使用した場合、レスポンスが悪くなる場合がある

プロセスバインド機能では、セション開始要求を受け取ったサーバアプリケーションプロセスに、そのセションIDに対する処理がバインドされます。通常、セション開始要求は、ワークユニット内で最後に要求待ちとなったサーバアプリケーションプロセスに振り分けられます。そのため、クライアントからの要求数が少ない場合、すべてのクライアントが特定のプロセスにバインドされ、その結果、1プロセスにセションが集中して性能劣化を引き起こすことがあります。

これを回避するためには、“セション開始要求(初回呼び出し)のプロセスへの振り分け方式”を参照してください。

8) 標準コード変換(FSUNiconv)の使用について Solaris32

コード変換関数は、OSのlibc.soにも存在します。そのため、アプリケーションから富士通の標準コード変換(FSUNiconv)を使用してコード変換する場合、以下の対処を実施してください。

- CまたはCOBOLのアプリケーションの場合

  tdlinknormapmコマンドを使用してFSUNiconvがリンクされたextp_apmTDNORMを再作成する必要が有ります。
  “リファレンスマニュアル（コマンド編）”より“コンポーネントトランザクションサービス運用コマンド”の“tdlinknormapm”を参照し、extp_apmTDNORMを再作成してください。
  以下にコマンドの使用例を示します。

```
tdlinknormapm -t thread -p "-L/opt/FSUNiconv/lib -licv" -o TDNORMICV
```

注意

上記の場合、-oオプションで指定した"TDNORMICV"をワークユニット定義の“APM”ステートメントに設定する必要があります。
なお、-oオプションを指定しない場合は、"TDNORM"が指定されたものと見なし、標準の extp_apmTDNORMが置き換えられます。そのため、あらかじめextp_apmTDNORMをバックアップしておく必要があります。

- C++のアプリケーションの場合

  アプリケーションのリンク時に、“libicv.so”が“libc.so”より先にリンクされるようにリンクオプションを指定してください。

第2章 サーバアプリケーションの作成(C言語)

ローカルトランザクション運用を行うためのアプリケーションの作成方法について、説明します。

2.1 サーバアプリケーションの開発

サーバアプリケーションを記述するために必要な以下の事項を説明します。

- サーバアプリケーションの構造
- サーバアプリケーションの入出力情報
- クライアントアプリケーションへの復帰値

Solaris32 Linux32

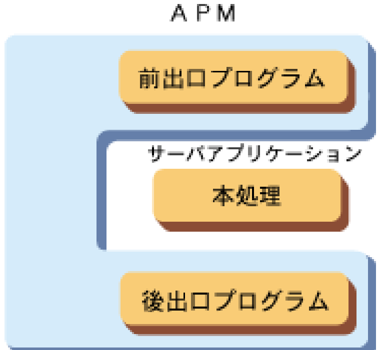
- サーバアプリケーションプロセスの形態

サーバアプリケーションの構造

一般のアプリケーションは、前処理、本処理、後処理という3つの構成で成り立っています。前処理として行う処理は、使用するデータベースの接続処理やデータベースのオープン処理などです。後処理として行う処理は、接続中データベースの切断処理やオープン中データベースのクローズ処理などです。前処理および後処理で行うことは、使用するデータベース管理システムごとに決まっています。そのため、Interstageでは前出口プログラムおよび後出口プログラムの機能を提供します。この機能を実現するのがAPM(Application Program Manager)です。

APMを使用することにより、サーバアプリケーション開発者は、本処理だけを開発することで、サーバアプリケーションの作成が可能となります。

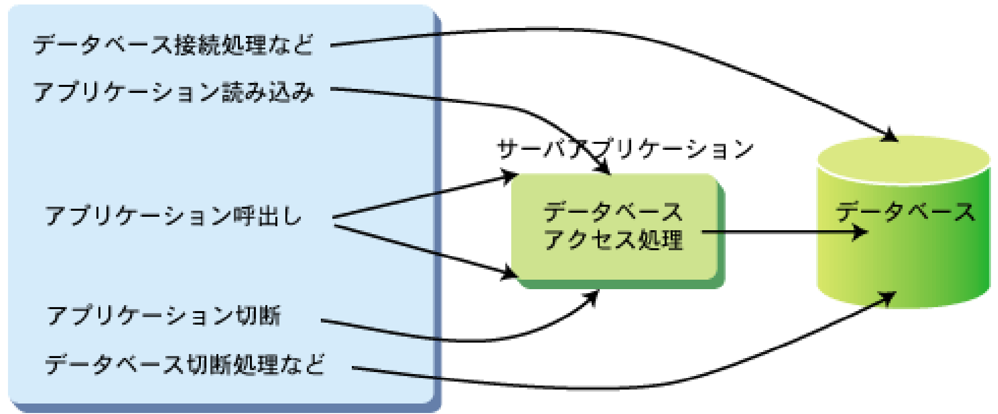
サーバアプリケーションの構造を以下に示します。



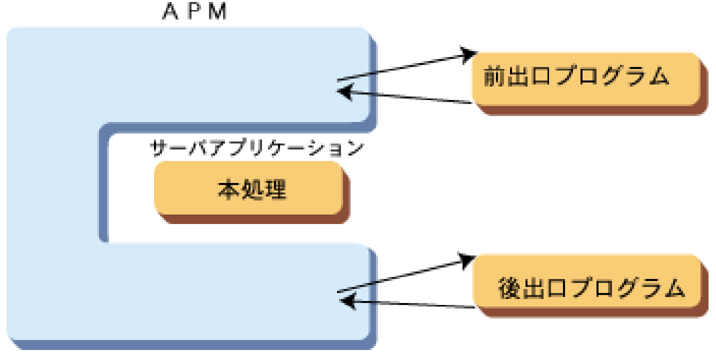
なお、データベース管理システムが提供するトランザクションを使用する場合には、サーバアプリケーション自身がデータベースとの接続や切断処理を行う必要があります。このような前処理および後処理を、出口プログラムとして本処理部分とは別に作成することができます。

この出口プログラムを作成することで、データ処理ごとに行っていたデータベースとの接続や切断処理を出口プログラムで行うことができるようになるため、効率の良い処理を構築することができます。

データベース管理システムが提供するトランザクションを使用する場合のサーバアプリケーションの構造を以下に示します。



APMとサーバアプリケーションの関係は、主プログラムと副プログラムという関係となります。また、サーバアプリケーションはAPMと動的結合により実行されます。
APMとサーバアプリケーションの関係について以下に示します。



サーバアプリケーションと、一般のアプリケーションとの違いを以下に示します。

- 副プログラムとして作成する必要があります。
- スレッドに関する命令は記述できません。
- 実行プログラムは、動的リンクライブラリの形式で作成する必要があります。

Solaris32 Linux32

- シグナルに関する命令は記述できません。

注意

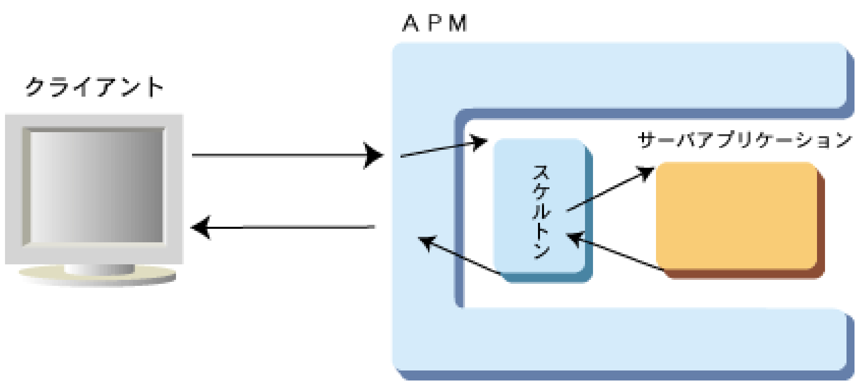
前出口プログラムは、ワークユニット起動時にアプリケーションプロセス(APM)単位で実行され、後出口プログラムは、ワークユニット停止時にアプリケーションプロセス(APM)単位で実行されます。ただし、ワークユニット強制停止時およびワークユニット異常終了時は、後出口プログラムは実行されません。
このとき、前出口プログラムおよび後出口プログラムの復帰値が0の場合は正常、0以外の場合は異常とします。復帰値が0以外の場合、ワークユニット起動時では、ワークユニットの起動失敗となり、ワークユニット停止時では、警告メッセージを出力し、ワークユニット停止処理は正常に終了します。
また、サーバアプリケーションの前処理、後処理の復帰値はすべてlong型にする必要があります。
以下の処理はInterstageが行うため、サーバアプリケーションで考慮する必要はありません。

- CORBAの初期化メソッドの呼び出し
- メソッドの登録
- サーバの活性化/非活性化

ただし、サーバアプリケーションが例外を使用する場合は、CORBAの初期化メソッドの呼び出しが必要です。

サーバアプリケーションの入出力情報

サーバアプリケーションは、クライアントからの呼び出し時に、スケルトンを経由して、IDLで定義した入力インタフェースで呼び出されます。また、サーバアプリケーションの処理が完了すると、スケルトンを経由してIDLで定義した出力インタフェースがクライアントに返却されます。
サーバアプリケーションの動作概要を以下に示します。



サーバアプリケーションの入出力インタフェースはCORBAで規定されています。CORBAのデータ型と言語とのマッピングを以下に示します。

CORBAデータ型			型宣言
基本データ型	整数型	・long long ・long ・unsigned long ・short ・unsigned short	・CORBA_long_long ・CORBA_long ・CORBA_unsigned_long ・CORBA_short ・CORBA_unsigned_short
	浮動小数点型	・float ・double ・long double	・CORBA_float ・CORBA_double ・CORBA_long_double
	文字型	・char ・wchar	・CORBA_char ・CORBA_wchar
	オクテット	・octet	・CORBA_octet
	ブーリアン	・boolean	・CORBA_boolean
	文字列型	・string ・wstring	・CORBA_string ・CORBA_wstring
構造データ型	構造体	・struct	――――――
	配列	――――――	――――――
	シーケンス	・sequence	――――――

サーバアプリケーションからスケルトンに、outパラメタまたはinoutパラメタのstring型のデータを受け渡す場合はTD_string_alloc、wstring型のデータを受け渡す場合はTD_wstring_allocにより領域を獲得する必要があります。また、string型およびwstring型データの場合には、サーバのアプリケーション用APIを利用する必要があります。サーバのアプリケーション用APIの詳細は、"リファレンスマニュアル(API編)"を参照してください。
なお、サーバアプリケーションでは、コンテキスト情報を使用することはできません。サーバアプリケーションに対しては、第1引数と最後の引数でシステムパラメタ、第2引数以降IDLで定義したパラメタというインタフェースでデータの入出力を行います。

また、IDL定義にoutパラメタを定義している場合、サーバアプリケーションが処理を終了する際には、必ずoutパラメタに復帰データを設定しなければなりません。outパラメタに復帰データが設定されていない場合、サーバアプリケーションプロセスが異常終了することがあります。

クライアントアプリケーションへの復帰値

クライアントアプリケーションへの復帰値は、Interstageにより規定されています。

復帰値	意味	アプリケーションの指定可否
0～10000	サーバアプリケーション任意	可
10001	システム異常検出(メモリ不足等)	不可
10002	システムで異常検出	不可
Windows32 Solaris32 10003	AIM連携のホスト間通信処理で異常検出	不可
10004	以下のいずれかの異常を検出 ― サーバアプリケーションでの異常または、 ― オペレーション名が不当	不可
Windows32 Solaris32 10005	AIM連携のセション継続機能において異常検出	不可
10006	アクセス権がないユーザからの要求	不可
10007	プロセスバインド機能において異常検出(注)	不可
10008	最大キューイング数に達した	不可

(注)IPCOMによる負荷分散を行っている環境でプロセスバインド機能を使用した場合にも復帰します。

復帰値0～10000

クライアントアプリケーションとサーバアプリケーションで任意に使用することができます。Interstageはこれらの復帰値を正常とみなします。

復帰値10001～10008

Interstageがクライアントに通知する復帰値であり、システムで異常が検出されたことを意味します。クライアントに復帰値10001～10008が通知された場合は、同時にその異常の詳細がエラーログに記録されます。

また、復帰値が10004で、エラーログに詳細が出力されていない場合は、クライアントから呼び出したオペレーションが、サーバのスケルトンに定義されていない(上記表の「オペレーション名が不当」に該当する)可能性があります。クライアントのスタブとサーバのスケルトンが同一のIDL定義から生成されているか、不一致がないかを確認してください。

なお、エラーログは、以下のファイルに採取されます。エラーログの内容については“メッセージ集”の“コンポーネントトランザクションサービスが出力するログメッセージ”を参照してください。

Windows32

```
C:¥INTERSTAGE¥td¥trc¥lorb¥errlog0
(トランザクションアプリケーションの場合)

C:¥INTERSTAGE¥td¥trc¥rorb¥errlog0
(AIM連携の場合)
```

Solaris32

```
/var/opt/FSUNtd/trc/lorb/errlog0
(トランザクションアプリケーションの場合)

/var/opt/FSUNtd/trc/rorb/errlog0
(AIM連携の場合)
```

Linux32

```
/var/opt/FJSVtd/trc/lorb/errlog0
(トランザクションアプリケーションの場合)
```

注意

クライアントアプリケーションでは、復帰値を参照する前に例外を判定する必要があります。例外が発生している場合は、復帰値は不定となります。クライアントアプリケーションでの例外処理については、“アプリケーション作成ガイド(CORBAサービス編)”を参照してください。

サーバアプリケーションプロセスの形態 Solaris32 Linux32

サーバアプリケーションプロセスには以下の2つの形態があります。

スレッドモード

サーバアプリケーションプロセスがマルチスレッドで動作する形態です。ただし、サーバアプリケーションはプライマリスレッド上でのみ動作し、マルチスレッドでは動作しません。スレッドモードではInterstageの制御用スレッドが複数起動します。

通常はスレッドモードを使用してください。

プロセスモード

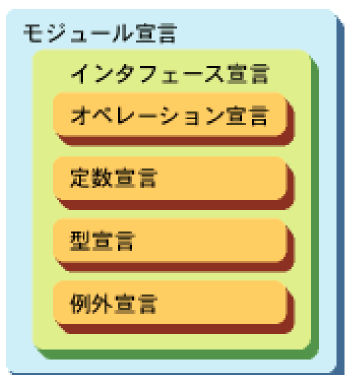
サーバアプリケーションプロセスがシングルスレッドで動作する形態です。本モードは、サーバアプリケーションより呼び出される他製品のライブラリが、マルチスレッドで動作するプロセス上で呼び出すことができない場合にのみ使用してください。

2.2 IDLファイルの作成

ローカルトランザクション運用におけるIDLファイルの文法の概要について説明します。IDLファイルの文法の詳細については、“アプリケーション作成ガイド（CORBAサービス編）”を参照してください。

IDLファイルの記述形式

ローカルトランザクション運用におけるIDLファイルの記述形式を以下に示します。



モジュール宣言

モジュール宣言では、IDL定義のオペレーション名や型名などが、ほかのIDL定義と重複しないように、オブジェクトのグループ化を定義します。
モジュール宣言の書式を以下に示します。

```
module モジュール名  {
        interface インタフェース名m {
            ・・
       } ;
        interface インタフェース名n {
            ・・
```

```
        } ;
    } ;
```

モジュール宣言の中に新たなモジュール宣言をすることもできます。これをモジュール宣言の入れ子と呼びます。モジュール宣言のmodule A宣言内にmodule B宣言をする例を以下に示します。

```
  module A  {
   interface a1 {
       ・・
    } ;
    module B  {
       interface b1 {
        ・・
       } ;
    } ;
  } ;
```

インタフェース宣言

インタフェース宣言では、アプリケーションへの入力と出力を規定するためのインタフェースを定義します。
インタフェース宣言の書式を以下に示します。

```
interface  インタフェース名[:継承するインタフェース名]  {
      オペレーション宣言 ;
      定数宣言 ;
      型宣言 ;
      例外宣言 ;

 } ;
```

※ [ ]内は省略可能です。

インタフェース宣言は、モジュール宣言内に複数個記述できます。トランザクション運用でインタフェース宣言を記述するときは、モジュール宣言を必ず記述する必要があります。

オペレーション宣言

オペレーション宣言では、アプリケーションに対応したオペレーション名、復帰値の型、パラメタのデータ型を定義します。
オペレーション宣言の書式を以下に示します。

```
 復帰値のデータ型  オペレーション名（
  [ パラメタタイプ  データ型  パラメタ名[ , . . ]]
）  [  raises （例外構造体名[ , . .]）] ;
```

※ [ ]内は省略可能です。

オペレーション宣言は、インタフェース宣言内に複数個記述できます。

定数宣言

インタフェースの定義内で使用する定数を定義します。
定数宣言の書式を以下に示します。

```
const データ型 定数名 ＝ 定数式 ;
```

データ型と型宣言

トランザクション運用で使用できるデータ型とその型宣言について示します。

基本データ型の宣言

データ型宣言では、インタフェース内で使用されるデータ型を定義できます。基本データ型のうち、整数型、浮動小数点型、文字型、オクテット型、ブーリアン型については、型宣言にtypedefを利用することもできます。
文字列型を除く基本データ型宣言の書式を以下に示します。

```
typedef long long          データ型名 ;
typedef long               データ型名 ;
typedef short              データ型名 ;
typedef unsigned long      データ型名 ;
typedef unsigned short     データ型名 ;
typedef float              データ型名 ;
typedef double             データ型名 ;
typedef long double        データ型名 ;
typedef char               データ型名 ;
typedef wchar              データ型名 ;
typedef octet              データ型名 ;
typedef boolean            データ型名 ;

long long                  データ名 ;
long                       データ名 ;
short                      データ名 ;
unsigned long              データ名 ;
unsigned short             データ名 ;
float                      データ名 ;
double                     データ名 ;
long double                データ名 ;
char                       データ名 ;
wchar                      データ名 ;
octet                      データ名 ;
boolean                    データ名 ;
```

文字列型の型宣言の書式を以下に示します。

```
typedef string<サイズ>     データ型名;
typedef string             データ型名;
typedef wstring<サイズ>    データ型名;
typedef wstring            データ型名;

string<サイズ>             データ名 ;
string                     データ名 ;
wstring<サイズ>            データ名 ;
wstring                    データ名 ;
```

構造体

構造体の宣言の書式を以下に示します。

```
struct  データ名 {
        構造体メンバの宣言
 };
```

構造体メンバの宣言の書式を以下に示します。

```
基本データ型     メンバ名 ;
```

構造体メンバは1つ以上必要であり、空の構造体は定義できません。
構造体の記述例を以下に示します。

```
module A {
        struct S {
                string  name;
```

```
                        short   number;
                        long    value;
            };
    };
```

配列

配列宣言では1次元固定長の配列を定義します。配列宣言の書式を以下に示します。

```
typedef データ型 識別子 [配列サイズ];
```

配列の次元は1階層までに制限されています。配列サイズは正の整数定数式で指定します。配列サイズはコンパイル時に固定されます。
配列宣言の例を以下に示します。

```
typedef long A[5];
```

シーケンス型

IDL言語でシーケンス型sequenceを指定した場合、以下の構造体でデータを宣言します。

```
struct
{
        CORBA_unsigned_long     _maximum;       /* シーケンスの最大長 */
        CORBA_unsigned_long     _length;        /* シーケンスの長さ   */
        CORBA_Type              *_buffer;       /* シーケンスのデータ */
};
```

_maximumはシーケンスの最大長、_lengthはシーケンスの長さ、_bufferはシーケンスのデータを指します。シーケンス構造体の領域を獲得するための関数(モジュール名、インタフェース名、構造体およびallocを"_"でつないだ名前の関数(以降XX_alloc関数と呼びます))がTDコンパイラで生成されます。
また、シーケンスのデータを獲得するための関数(モジュール名、インタフェース名、構造体名およびallocbufを"_"でつないだ関数(以降XX_allocbuf関数と呼びます))がTDコンパイラで生成されます。
シーケンス型の宣言の書式を以下に示します。

```
typedef sequence<データ型名 > データ名;
```

または

```
typedef sequence<データ型名 , シーケンス要素数 > データ名;
```

シーケンス型の要素として指定するデータ型は基本データ型だけに制限されています。
シーケンス型の記述例を以下に示します。

```
typedef sequence<long > Q;
```

例外宣言

例外宣言では、オペレーション実行中に例外が発生したときに例外情報を受け渡すための例外構造体名を定義します。オペレーション宣言のraises式を定義する場合は、この例外構造体名を指定します。
例外宣言の書式を以下に示します。

```
exception  例外構造体名 {
        データ型   メンバ名 ;      //構造体メンバを宣言
                    :
} ;
```

構造体メンバを複数定義する場合は、各メンバをセミコロン(“;”)で区切ります。メンバとして定義できるデータ型を以下に示します。

- 基本データ型

・ 配列(配列要素のデータ型は基本データ型のみ、次元数は1次元まで)

また、例外にはシステムでの異常終了を通知するシステム例外とサーバアプリケーションでの異常終了を通知するユーザ例外がありますが、トランザクションアプリケーションではシステム例外は使用できません。
トランザクションアプリケーションでは、CORBAの初期化メソッドの呼び出しは必要ありませんが、例外を使用する場合、CORBAの初期化メソッドの呼び出しが必要となります。

例外の使用方法については、“アプリケーション作成ガイド(CORBAサービス編)”の各言語ごとのアプリケーションの例外処理を参照してください。

注意

コンポーネントトランザクションサービスにおける利用範囲

トランザクション運用で使用するIDLファイルの文法は、OMGで規定しているIDLの文法に準拠していますが、以下に示す項目が制限されています。

- モジュール宣言などでは、次の宣言が利用できません。
  - 属性宣言
  - コンテキスト
- 以下のデータ型は使用できません。
  - 基本データ型の列挙型(enum)とany型
  - 共用体型
- オペレーション宣言において、利用できる型は以下に限られています。
  - long
  - oneway void

データ型の使用方法

各データ型の使用方法については、“アプリケーション作成ガイド(CORBAサービス編)”の各言語ごとのデータ型に対するマッピングを参照してください。

なお、クライアントアプリケーションにVisual Basicを使用し、サーバアプリケーションにOLE CORBA-ゲートウェイ経由で文字列型のデータを通信する場合には、IDL定義の記述に注意が必要です。“アプリケーション作成ガイド(CORBAサービス編)”のCOM/CORBA連携プログラミングを参照してください。

CORBAアプリケーションとトランザクションアプリケーションでは、使用するAPIが以下に示す点で異なります。

```
CORBA_string_alloc ==> TD_string_alloc
CORBA_free ==> TD_free
CORBA_wstring_alloc ==> TD_wstring_alloc
```

また、トランザクションアプリケーションでは、リリースフラグの参照APIおよびリリースフラグの設定APIが用意されていません。データの扱いは、CORBA_FALSEが設定されているものとして扱ってください。

オペレーションの使用方法

トランザクションアプリケーションでは、オペレーションの型として、longおよびoneway voidのみ用意しています。

oneway voidの場合、以下の注意が必要となります。

- オペレーションを実装するプログラムは、void型となります。
- クライアントアプリケーションへの復帰は、サーバアプリケーションの状態にかかわらず、サーバアプリケーションへデータを送信し、送信処理が成功した時点で正常復帰します。
- outパラメタ、inoutパラメタは使用できません。
- ユーザ例外を使用できません。

- グローバルトランザクション連携機能は使用できません。

2.3 サーバアプリケーションのソースの作成

サーバアプリケーションの処理を、オペレーティングシステムに添付されているエディタなどを使用して記述します。

ローカルトランザクション運用の場合は、リソースマネージャが提供するトランザクション命令と、リソースマネージャを使用するために必要な、データベースの結合処理や切り離し文などを記述してください。記述するトランザクション命令については、使用するリソースマネージャのマニュアルを参照してください。

IDL定義とサーバアプリケーション名の関係を以下に示します。

```
モジュール名_インタフェース名_オペレーション名
```

注意

- サーバアプリケーションにおいて、引数に指定するオブジェクト情報は設定されていません。
- セション情報管理機能を使用する場合は、“5.3 セション情報管理機能を使用したトランザクションアプリケーションの作成”を参照してください。
- トランザクションアプリケーションにおいて、イベントサービスを利用する場合、使用可能なイベントサービスの機能は以下になります。
  - Pushモデルのサプライヤ

サーバアプリケーションにインクルードするヘッダファイル

IDLコンパイルにより生成される以下のヘッダファイルをサーバアプリケーションにインクルードしてください。

ヘッダファイル	種別
TD_オブジェクト名.h	スケルトン用ヘッダファイル
IDLファイル名.h	スタブ用ヘッダファイル(注)

(注)
中継用サーバアプリケーションを作成するときに必要です。スタブ側のIDLファイルのコンパイルにより生成されます。

出口プログラムの作成

出口プログラムは以下の形式で作成してください。なお、ここでの出口プログラムとは、前出口プログラム、後出口プログラム、および、プロセスバインド機能を使用する場合の異常出口プログラムを指します。

引数	なし
復帰値	long型 正常終了:0 異常終了:0以外

復帰値が0以外の場合は、出口プログラム異常終了とみなし、出口プログラム種別ごとに以下の動作となります。

- 前出口プログラムの場合、ワークユニットの起動に失敗します。
- 後出口プログラムの場合、警告メッセージを出力し、ワークユニットの停止処理は成功します。
- プロセスバインド機能の異常出口の場合、アプリケーションプロセスが異常終了します。その後、アプリケーション異常時の自動再起動が設定されている場合は、アプリケーションプロセスが自動再起動され、自動再起動が設定されていない場合は、ワークユニットが異常終了します。

出口プログラム作成時の注意点 Windows32

出口プログラムを使用する場合、出口プログラムをDLLからエクスポートするために、関数のプロトタイプ宣言に__declspec(dllexport)キーワードを付加してください。または、モジュール定義ファイル(.DEF)を使用して、出口プログラムをエクスポートしてください。

Microsoft(R) Visual C++ .NETまたは、Microsoft(R) Visual C++ 2005を使用してアプリケーションを作成する場合の注意点 Windows32

アプリケーションより標準出力または標準エラー出力に向けて出力されたデータは、カレントフォルダ配下のstdoutファイルまたはstderrファイルに出力されます。

しかし、Microsoft(R) Visual C++ .NETまたは、Microsoft(R) Visual C++ 2005を使用してビルドされたアプリケーションでは、標準出力または標準エラー出力に向けて出力されたデータが、カレントフォルダ配下のstdoutファイルまたはstderrファイルに出力されません。

これを回避し正しく出力するためには、アプリケーションにおいて以下の対処を実施してください。

プログラムの先頭に以下のコードを追加してください。(注1)

```
freopen("stdout", "w", stdout);
freopen("stderr", "w", stderr);
```

なお、前出口プログラムを使用される場合は、前出口プログラムの先頭に追加してください。前出口プログラムに追加した場合、本処理および後出口プログラムへの対処は必要ありません。

前出口プログラムを使用されない場合は、本処理の先頭に追加し、かつ、初回呼び出し時のみ実行するよう対処してください。

注1) Microsoft(R) Visual C++ 2005を使用してビルドした場合、"warning C4996: 'freopen' が古い形式として宣言されました。"という警告が出力される場合がありますが、動作上の問題はありません。

2.4 ソースのコンパイル・リンク

2.4.1 IDLファイルのコンパイル

IDLファイルをコンパイルすることにより、クライアント、サーバそれぞれのアプリケーションの言語に合わせたスタブファイルとスケルトンファイルが作成されます。IDLファイルのコンパイルには、tdcコマンドを使用します。tdcコマンドのオプションは、使用する言語により異なります。tdcコマンドの詳細については、"リファレンスマニュアル(コマンド編)"を参照してください。tdcコマンドを使用した例を以下に示します。

Windows32

```
>tdc -c -mc tdsample1.idl
```

Solaris32

```
%OD_HOME=/opt/FSUNod
%export OD_HOME
%TD_HOME=/opt/FSUNtd
%export TD_HOME
%tdc -c -mc tdsample1.idl
```

Linux32

```
%OD_HOME=/opt/FJSVod
%export OD_HOME
%TD_HOME=/opt/FJSVtd
%export TD_HOME
%tdc -c -mc tdsample1.idl
```

注意

トランザクションアプリケーションでは、インタフェースリポジトリにインタフェース情報を登録する必要があります。インタフェースリポジトリに登録したインタフェース情報とスタブに埋め込まれるインタフェース情報は必ず一致している必要があります。IDLコンパイラでインタフェース情報を生成する際には、インタフェース情報チェック機能を使用することを推奨します。インタフェース情報チェック機能についての詳細は、"OLTPサーバ運用ガイド"の"インタフェース情報チェック機能を使用した運用"を参照してください。

アプリケーション間連携時に、以下の条件を満たす場合、コンパイル時に関数の二重定義エラーが発生します。

- 1つのクライアントアプリケーションから、複数のサーバアプリケーションを呼び出し可能とする。

・ 各々のサーバアプリケーションで使用するIDL定義に同一種別のデータ型(基本データ型は除く)が宣言されている。

この場合は出力された関数名を変更するか、IDLcコマンド(-lcまたは-lsオプションを指定)を使用してスタブファイルを生成してください。IDLcコマンドの詳細については、“リファレンスマニュアル(コマンド編)”を参照してください。

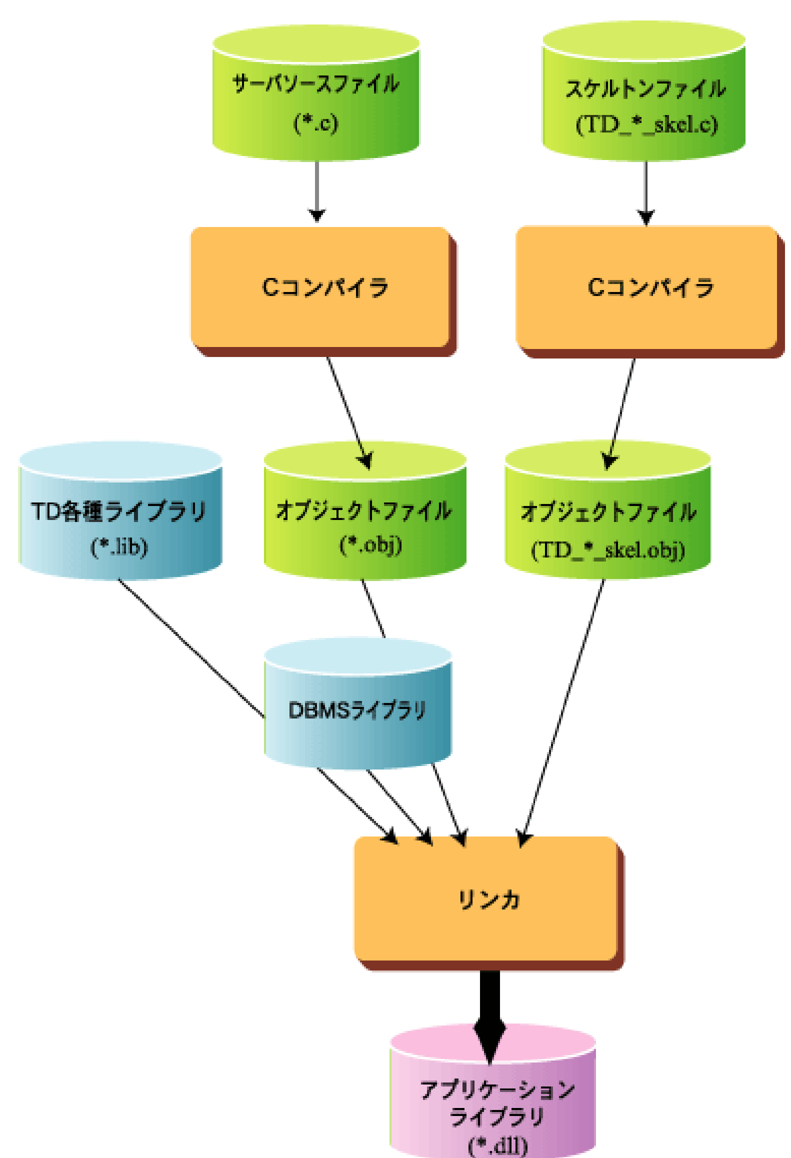
2.4.2 スタブとクライアントアプリケーションのソースとのコンパイル・リンク

記述したクライアントアプリケーションのソースとスタブをコンパイルします。コンパイル方法の詳細については、クライアントアプリケーションを動作させるオペレーティングシステムやミドルウェアのマニュアルを参照してください。

2.4.3 スケルトンとサーバアプリケーションのソースとのコンパイル・リンク

サーバアプリケーションのソースとスケルトンのコンパイル方法、およびリンク方法について説明します。
C言語でアプリケーションを作成した場合について、コンパイルとリンクの手順の流れを以下に示します。

Windows32



サーバアプリケーションは、APMと動的に結合します。このため、サーバアプリケーションは共用ライブラリとし、動的結合ができる状態にしておく必要があります。この共用ライブラリには、サーバアプリケーションが使用するライブラリを結合してください。データベース管理システムのライブラリ以外で、サーバアプリケーションのリンク時に指定するライブラリを以下に示します。

[リンク時に指定するライブラリ]

ライブラリ名	格納場所	用途
f3fmalcapi.lib	INTERSTAGEインストールフォルダ¥td¥lib	コンポーネントトランザクションサービスランタイム(必須)
libextpapiskl.lib	INTERSTAGEインストールフォルダ¥extp¥lib	コンポーネントトランザクションサービスランタイム(必須)
odwin.lib	INTERSTAGEインストールフォルダ¥odwin¥lib	CORBAサービスランタイム(注)
odcn.lib	INTERSTAGEインストールフォルダ¥odwin¥lib	CORBAサービスランタイム(注)
odif.lib	INTERSTAGEインストールフォルダ¥odwin¥lib	CORBAサービスランタイム(注)

(注)
中継用サーバアプリケーションを作成するとき、および、例外を使用するときに必要です。

サーバアプリケーションとスケルトンをVisual C++でコンパイル後、リンクする場合のコンパイルオプションの例を以下に示します。

[コンパイルオプションの例]

Visual C++でコンパイルする場合は、[プロジェクト]-[設定]-[C/C++]の「カスタマイズ」で、以下の表に示すオプションを設定してください。

カテゴリー	項目	設定値
コード生成	Processer	80386を推奨
	構造体メンバのアライメント	4バイト
	使用するランタイム	マルチスレッド(DLL)
最適化		デフォルトを推奨
プリプロセッサ	プリプロセッサの定義	"OM_PC","OM_WIN32_BUILD", "__STDC__" を追加

また、[ツール]-[オプション]-[ディレクトリ]の「インクルードファイル」、「ライブラリファイル」にそれぞれINTERSTAGEインストールフォルダの下の"include"、"lib"フォルダを登録してください。

登録例)

(INTERSTAGEをC:¥INTERSTAGEにインストールした場合)

- インクルードファイル
  C:¥INTERSTAGE¥odwin¥include
  C:¥INTERSTAGE¥td¥include
  C:¥INTERSTAGE¥extp¥include
- ライブラリファイル
  C:¥INTERSTAGE¥odwin¥lib
  C:¥INTERSTAGE¥td¥lib
  C:¥INTERSTAGE¥extp¥lib

リンク時の注意事項

サーバアプリケーションのリンク時には、上記のライブラリのほかに、データベースや連携する他製品のライブラリなど、サーバアプリケーションで必要なライブラリがすべてリンクされていることを確認してください。また、リンクしたライブラリの格納先をワークユニット定義のアプリケーション使用パス(Path for Applicationセクション)に設定してください。
リンクが完全でない場合やワークユニット定義のアプリケーション使用パスが完全でない場合は、ワークユニット起動処理において実行ファイルのオープン処理に失敗し、メッセージ(EXTP4640,EXTP4643,EXTP4645,EXTP4508)が出力されます。

以下に必要なワークユニット定義の設定例を示します。

ワークユニット定義の設定例

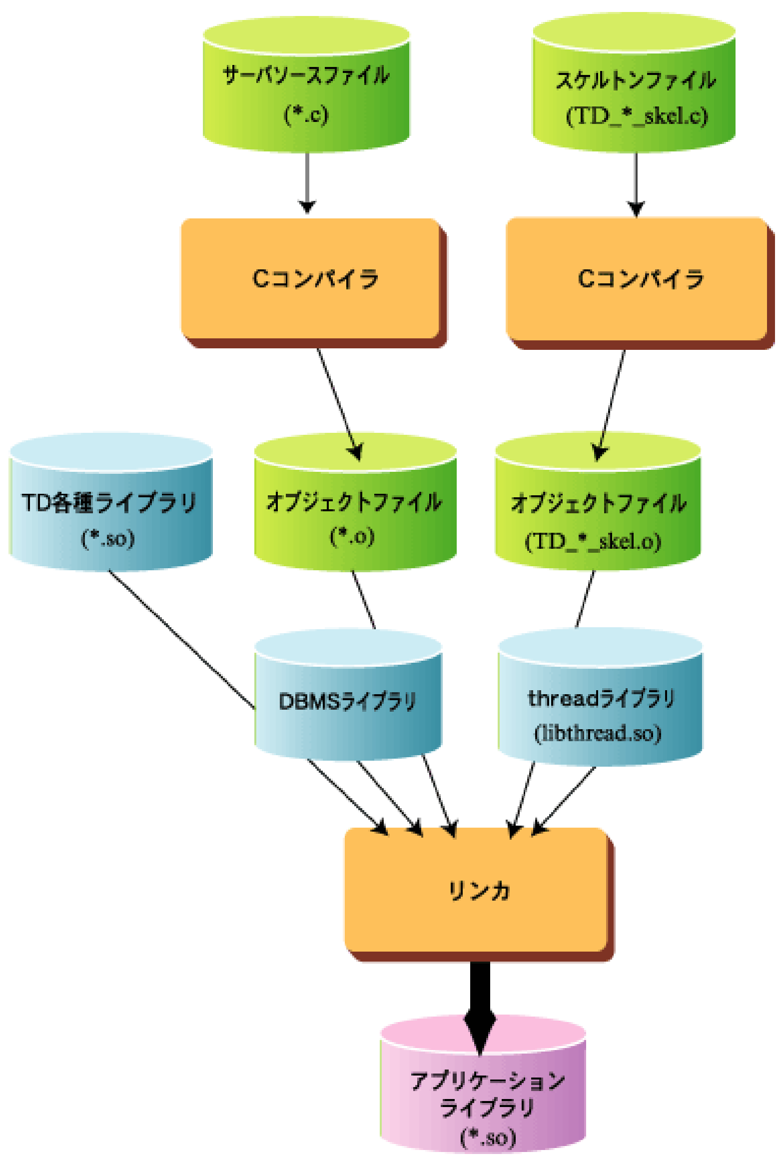
サーバアプリケーションの実行ファイルが"application1.dll"、格納先がc:¥application¥bin、サーバアプリケーションにリンクされているライブラリの格納先が、それぞれ、"d:¥product1¥bin"、"e:¥product2¥bin"の場合の定義例を以下に示します。

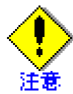
```
[WORK UNIT]
Name: SAMPLEWU
Kind: ORB
[APM]
Name: TDNORM

[Control Option]
…
Path: c:¥application¥bin
Path for Application: d:¥product1¥bin
Path for Application: e:¥product2¥bin
…

[Application Program]
…
Executable File : application1.dll
```

サーバアプリケーションにリンクされているライブラリの格納先が複数存在する場合は、アプリケーション使用パス(Path for Applicationセクション)を複数設定してください。

また、出口プログラムにリンクされているライブラリの格納先についても、同様にアプリケーション使用パス(Path for Applicationセクション)に設定してください。

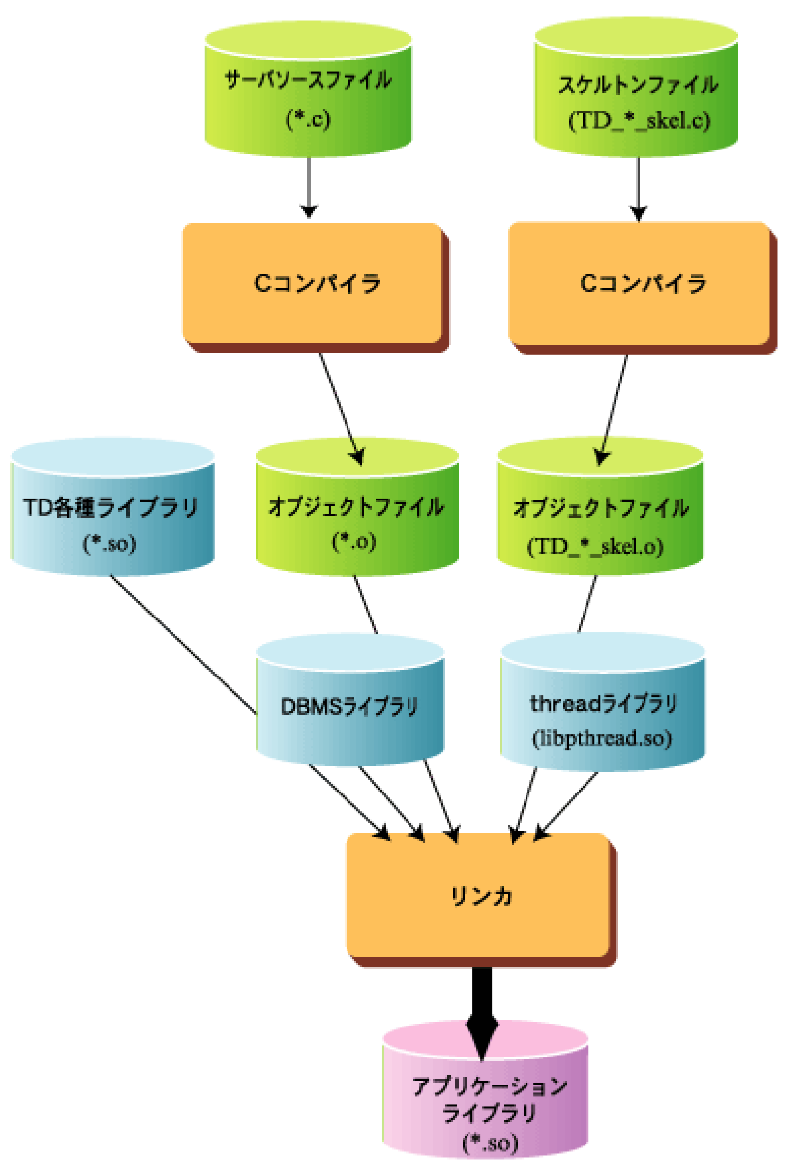
Solaris32

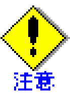




スレッドライブラリ(libthread.so)は、スレッドモードで作成する場合のみ必要です。

Linux32





スレッドライブラリ(libpthread.so)は、スレッドモードで作成する場合のみ必要です。

サーバアプリケーションは、APMと動的に結合します。このため、サーバアプリケーションは共用ライブラリとし、動的結合ができる状態にしておく必要があります。この共用ライブラリには、サーバアプリケーションが使用するライブラリを結合してください。データベース管理システムのライブラリ以外で、サーバアプリケーションのリンク時に指定するライブラリを以降に示します。

APMの再作成

コード変換用関数などのように、ユーザアプリケーションから呼び出されているシンボルが、ユーザがアプリケーションにリンクしたライブラリとlibc.soなどのシステムライブラリとの両方に格納されているとき、ユーザアプリケーションを実行すると、無条件にlibc.soなどのシステムライブラリに格納されたシンボルが呼び出されます。これは、APM実行モジュールの先頭にlibc.soなどのシステムライブラリが結合されているためです。この場合、ユーザアプリケーションに結合されたライブラリのシンボルを呼び出すためには、APM実行モジュールの先頭にユーザアプリケーションが使用するライブラリを結合する必要があります。

APM実行モジュールの先頭にユーザアプリケーションが使用するライブラリを結合するためには、tdlinknormapmコマンドを使用し、APMを再作成してください。tdlinknormapmコマンドの使用方法については“リファレンスマニュアル(コマンド編)”を参照してください。

2.4.4 スレッドモードのアプリケーションのコンパイルとリンク

[コンパイル時に指定するオプション] Linux32

```
-D-REENTRANT（スレッド版のみ）
```

[リンク時に指定するオプション] Linux32

```
-shared -fpic　または
-shared -fPIC
```

[コンパイル時に指定するインクルードパス] Solaris32 Linux32

IDLコンパイルを実施したディレクトリ
TDのインストールディレクトリ/include
ODのインストールディレクトリ/include

[リンク時に指定するライブラリ] Solaris32

ライブラリ名	格納場所	用途
libthread.so	/usr/lib	スレッドライブラリ(必須)(注1)
libtdalcapi.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libextpapiskl.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libOM.so	ODのインストールディレクトリ/lib	ODランタイム(注2)
libOMcn.so	ODのインストールディレクトリ/lib	ODランタイム(注3)
libsocket.so	/usr/lib	ソケットライブラリ(注2)
libnsl.so	/usr/lib	TLIライブラリ(注2)

(注1)
スレッドライブラリは必ず結合するライブラリの中で先頭に指定してください。

(注2)
中継用サーバアプリケーションを作成するとき、および、例外を使用するときに必要です。

(注3)
中継用サーバアプリケーションを作成するときに必要です。

なお、リンカにより結合処理を行う場合、以下のオプションを指定してください。

```
-dy -G
```

サーバアプリケーションとスケルトンをコンパイル後、リンクする手順について例を以下に示します。

[リンク時に指定するライブラリ] Linux32

ライブラリ名	格納場所	用途
libpthread.so	/usr/lib	スレッドライブラリ(必須)(注1)
libtdalcapi.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libextpapiskl.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libOM.so	ODのインストールディレクトリ/lib	ODランタイム(注2)
libOMcn.so	ODのインストールディレクトリ/lib	ODランタイム(注3)
libnsl.so	/usr/lib	TLIライブラリ(注2)

(注1)
スレッドライブラリは必ず結合するライブラリの中で先頭に指定してください。

(注2)

中継用サーバアプリケーションを作成するとき、および、例外を使用するときに必要です。

(注3)

中継用サーバアプリケーションを作成するときに必要です。

サーバアプリケーションとスケルトンをコンパイル後、リンクする手順について例を以下に示します。

[コンパイル・リンク手順の例]

Solaris32

```
%cc -c -D_REENTRANT -I/opt/FSUNod/include -I/opt/FSUNtd/include tdsample1_s.c
%cc -c -D_REENTRANT -I/opt/FSUNod/include -I/opt/FSUNtd/include
TD_TDSAMPLE1_INTF_skel.c
%cc -G -o libtdsample1.so tdsample1_s.o TD_TDSAMPLE1_INTF_skel.o -lthread
 -L/opt/FSUNtd/lib -ltdalcapi -L/opt/FSUNextp/lib -lextpapiskl
%
```

Linux32

```
%gcc -c -D_REENTRANT -I/opt/FJSVod/include -I/opt/FJSVtd/include tdsample1_s.c
%gcc -c -D_REENTRANT -I/opt/FJSVod/include -I/opt/FJSVtd/include
TD_TDSAMPLE1_INTF_skel.c
%gcc -shared -fpic -o libtdsample1.so tdsample1_s.o TD_TDSAMPLE1_INTF_skel.o -lpthread
 -L/opt/FJSVtd/lib -ltdalcapi -L/opt/FJSVextp/lib -lextpapiskl
%
```

2.4.5 プロセスモードのアプリケーションのコンパイルとリンク

[コンパイル時に指定するインクルードパス] Solaris32 Linux32

IDLコンパイルを実施したディレクトリ
TDのインストールディレクトリ/include
ODのインストールディレクトリ/include

アプリケーションをプロセスモードで動作させる場合は、以下のオプションは指定しないでください。

Solaris32

```
-mt -D_REENTRANT
```

Linux32

```
-D_REENTRANT
```

サーバアプリケーションのリンク時に指定するライブラリを以下に示します。

[リンク時に指定するライブラリ] Solaris32

ライブラリ名	格納場所	用途
libtdalcapi_nt.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libextpapiskl_nt.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libOM.so	ODのインストールディレクトリ/lib/nt	ODランタイム(注1)
libOMcn.so	ODのインストールディレクトリ/lib	ODランタイム(注2)
libsocket.so	/usr/lib	ソケットライブラリ(注1)

ライブラリ名	格納場所	用途
libnsl.so	/usr/lib	TLIライブラリ(注1)

(注1)
中継用サーバアプリケーションを作成するとき、および、例外を使用するときに必要です。

(注2)
中継用サーバアプリケーションを作成するときに必要です。

なお、リンカにより結合処理を行う場合、以下のオプションを指定してください。

```
-dy -G
```

サーバアプリケーションとスケルトンをコンパイル後、リンクする手順について例を以下に示します。

[リンク時に指定するライブラリ] Linux32

ライブラリ名	格納場所	用途
libtdalcapi_nt.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libextpapiskl_nt.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libOM.so	ODのインストールディレクトリ/lib/nt	ODランタイム(注1)
libOMcn.so	ODのインストールディレクトリ/lib	ODランタイム(注2)
libnsl.so	/usr/lib	TLIライブラリ(注1)

(注1)
中継用サーバアプリケーションを作成するとき、および、例外を使用するときに必要です。

(注2)
中継用サーバアプリケーションを作成するときに必要です。

サーバアプリケーションとスケルトンをコンパイル後、リンクする手順について例を以下に示します。

[コンパイル・リンク手順の例]

Solaris32

```
%cc -c -I/opt/FSUNod/include -I/opt/FSUNtd/include tdsample1_s.c
%cc -c -I/opt/FSUNod/include -I/opt/FSUNtd/include TD_TDSAMPLE1_INTF_skel.c
%cc -G -o libtdsample1_nt.so tdsample1_s.o TD_TDSAMPLE1_INTF_skel.o
 -L/opt/FSUNtd/lib -ltdalcapi_nt -L/opt/FSUNextp/lib -lextpapiskl_nt
%
```

注意

作成したアプリケーションライブラリにスレッドライブラリが結合されていないことを確認してください。
アプリケーションのコンパイル／リンクの手順に誤りがあった場合や、libthread.soがリンクされている場合、ワークユニット起動／停止処理またはクライアントとの通信が無応答状態となる可能性があります。その場合、以下の対処を行ってください。

1. 当該ワークユニット配下で動作しているアプリケーションプロセスのプロセスIDを特定します。

```
% ps -ef | grep ワークユニット名
```

2. 当該ワークユニット配下で動作しているアプリケーションプロセスの強制停止

```
% kill -9 プロセスID
```

注意

作成したアプリケーションライブラリにスレッドライブラリが結合されていないことを確認してください。
アプリケーションのコンパイル／リンクの手順に誤りがあった場合や、libpthread.soがリンクされている場合、ワークユニット起動／停止処理またはクライアントとの通信が無応答状態となる可能性があります。その場合、以下の対処を行ってください。

1. 当該ワークユニット配下で動作しているアプリケーションプロセスのプロセスIDを特定します。

```
% ps -ef | grep ワークユニット名
```

2. 当該ワークユニット配下で動作しているアプリケーションプロセスの強制停止

```
% kill -9 プロセスID
```

リンク時の注意事項

サーバアプリケーションのリンク時には、上記のライブラリのほかに、データベースや連携する他製品のライブラリなど、サーバアプリケーションで必要なライブラリがすべてリンクされていることを確認してください。また、リンクしたライブラリの格納先をワークユニット定義のアプリケーション使用ライブラリパス(Library for Applicationセクション)に設定してください。
リンクが完全でない場合やワークユニット定義のアプリケーション使用ライブラリパスが完全でない場合は、ワークユニット起動処理において実行ファイルのオープン処理に失敗し、メッセージ(EXTP4640,EXTP4643,EXTP4645,EXTP4508)が出力されます。

Solaris32

プロセスモードの場合でリンク時にlibOM.soを指定する場合には、必ず"/opt/FSUNod/lib/nt"を指定しなければなりません。

Linux32

プロセスモードの場合でリンク時にlibOM.soを指定する場合には、必ず"/opt/FJSVod/lib/nt"を指定しなければなりません。

以下に必要なワークユニット定義の設定例を示します。

ワークユニット定義の設定例

サーバアプリケーションの実行ファイルが"application1.so"、格納先が/application/lib、サーバアプリケーションにリンクされているライブラリの格納先が、それぞれ、"/product1/lib"、"/product2/lib"で、プロセスモードでかつリンク時にlibOM.soを指定する場合の定義例を以下に示します。

Solaris32

```
[WORK UNIT]
Name: SAMPLEWU
Kind: ORB
[APM]
Name: TDNORMnt

[Control Option]
…
Path: /application/lib
Library for Application: /product1/lib
Library for Application: /product2/lib
Library for Application: /opt/FSUNod/lib/nt
…

[Application Program]
…
Executable File : application1.so
```

Linux32

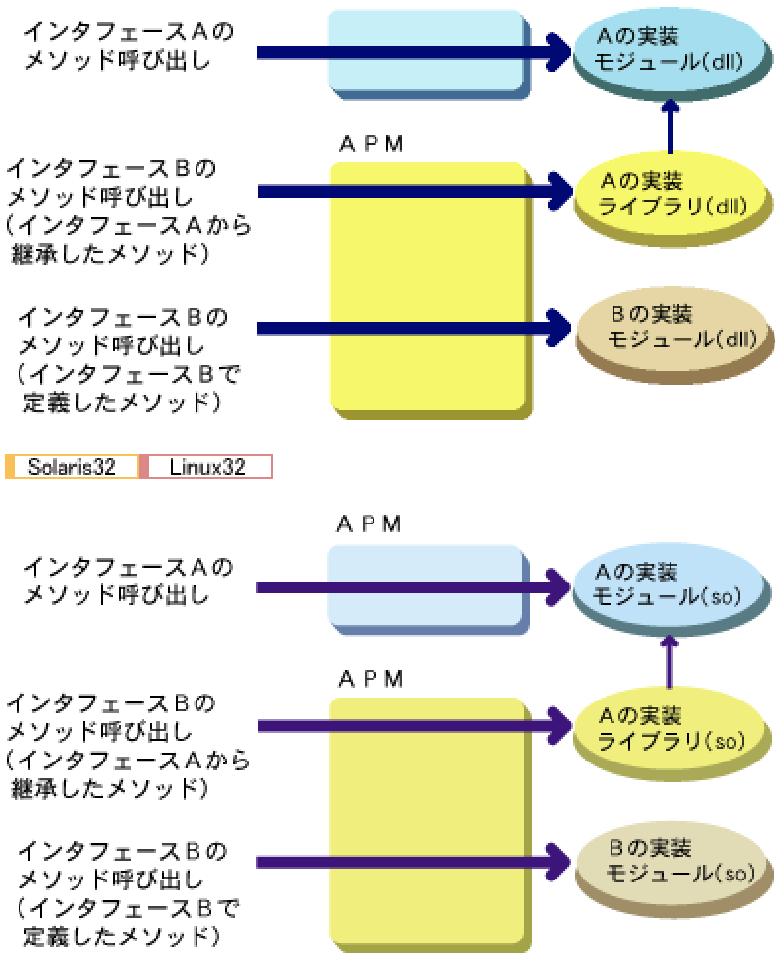
```
[WORK UNIT]
Name: SAMPLEWU
Kind: ORB
[APM]
```

```
Name: TDNORMnt

[Control Option]
…
Path: /application/lib
Library for Application: /product1/lib
Library for Application: /product2/lib
Library for Application: /opt/FJSVod/lib/nt
…

[Application Program]
…
Executable File : application1.so
```

サーバアプリケーションにリンクされているライブラリの格納先が複数存在する場合は、アプリケーション使用ライブラリパス(Library for Applicationセクション)を複数設定してください。
また、出口プログラムにリンクされているライブラリの格納先についても、同様にアプリケーション使用ライブラリパス(Library for Application セクション)に設定してください。

2.4.6 継承について

あるインタフェースで定義したオペレーションを、別のインタフェースに引き継ぐことを可能にするための機能です。継承を利用した場合、クライアントからは継承を意識せずに、継承したオペレーションを呼び出すことができます。
メソッドの呼び出しイメージについて以下に示します。



継承は、IDL定義ファイルに継承の指定を記述することによって使用することができます。継承先インタフェースは、継承元インタフェースのスコープ名をコロン(":")に続いて指定します。継承の指定例を以下に示します。
この例では、インタフェース“A”が継承元インタフェース、インタフェース“B”が継承先インタフェースとなります。

```
interface A {
        long op1(in long a);
};
interface B:X::A {
        long op2(in long b);
};
```

X:module名
A、B:interface名

継承先のIDL定義内に継承元のIDL定義をincludeする記述が必要です。include方法の例を以下に示します。

A.idl

```
module X{
 interface A{
  long op1();
 };
};
```

B.idl

```
#include "A.idl"
module Y{
 interface B:X::A{
  long op2();
 };
};
```

X::Aにop2を追加したY::Bを作成する方法

継承を使用する場合、モジュール作成時に継承元のインタフェースを実装しているライブラリをリンクします。継承を使用したアプリケーションの作成時に必要な注意点を、以下に示します。

継承元のインタフェースの状態

継承先のIDL定義をIDLコンパイルする場合、継承元は以下の状態であることが考えられます。ここでは、以下の状態の注意点を説明します。

- 継承元のインタフェースがインタフェースリポジトリに登録されていない場合
- 継承元のインタフェースがインタフェースリポジトリに登録されている場合

継承元のインタフェースがインタフェースリポジトリに登録されていない場合

tdcコマンド実行時に-Iオプションを指定してください。tdcコマンドの詳細については、“リファレンスマニュアル(コマンド編)”を参照してください。

継承元のインタフェースがインタフェースリポジトリに登録されている場合

継承先のIDL定義に継承元のIDL定義をincludeする記述を追加します。tdcコマンドは、-Iオプションと共に、-updateオプションを設定して実行してください。tdcコマンドの詳細については、“リファレンスマニュアル(コマンド編)”を参照してください。

継承元のライブラリの所在

継承先のサーバアプリケーションをコンパイルする場合の注意点を説明します。

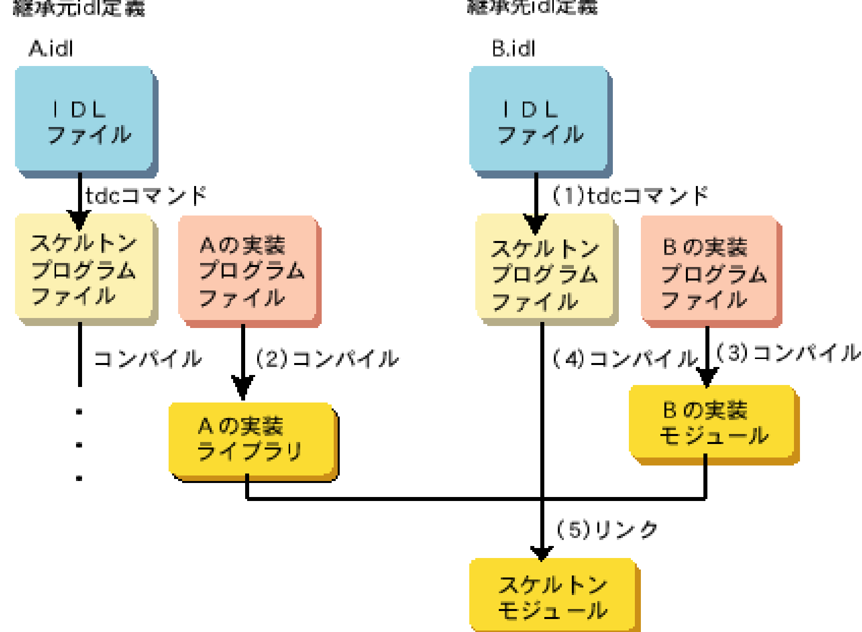
- 継承元インタフェースが実装されているライブラリのリンク
- 継承元インタフェースが実装されているライブラリのパスの設定

継承元インタフェースが実装されているライブラリのリンク

継承を使用する場合のサーバアプリケーション作成方法

(1) 継承先のIDL定義ファイル(B.idl)をtdcコマンドでコンパイルします。
(2) 継承元となるAの実装部は、スケルトンと別ライブラリとして作成します。
(3) 継承先となるBの実装部をコンパイルし、ライブラリを作成します。

(4) (1)で生成されたスケルトンをコンパイルします。
(5) (2)(3)(4)で生成されたライブラリをリンクします。



Windows32

ライブラリ

ダイナミックリンクライブラリ(.lib)

モジュール

ダイナミックリンクライブラリ(.dll)

Solaris32 Linux32

ライブラリ

ダイナミックリンクライブラリ(.so)

モジュール

ダイナミックリンクライブラリ(.so)

継承元インタフェースが実装されているライブラリのパスの設定

継承元インタフェースが実装されているライブラリのパスの設定は、ワークユニット定義で設定します。

Windows32

"Path for Application:"ステートメントに、継承元インタフェースへのパスを記述します。

Solaris32 Linux32

"Library for Application:"ステートメントに、継承元インタフェースへのパスを記述します。

2.5 ワークユニット定義の作成

ローカルトランザクション運用におけるワークユニット定義の概要について説明します。ワークユニット定義の詳細については、"付録A ワークユニット定義の記述形式"および"OLTPサーバ運用ガイド"を参照してください。

定義情報の設定

ワークユニット定義ファイルに記述する定義情報について、以下に示します。

ワークユニット名

ワークユニットを操作するためのワークユニット名を[WORK UNIT]セクションで設定します。ワークユニット名は、ワークユニット単位に1つ設定することができます。

APM名

APM名を[APM]セクションに設定します。"TDNORM"と指定してください。

Solaris32 Linux32

ただし、プロセスモードで動作させる場合は、"TDNORMnt"を指定してください。
また、APM再作成コマンドで作成したAPMを使用する場合は、再作成したAPM名を指定してください。

制御オプション

アプリケーションが動作するためのカレントディレクトリやアプリケーションが格納されているライブラリパスなどの環境情報を[Control Option]セクションで設定します。

アプリケーション情報

ワークユニットで動作させるアプリケーション名やオブジェクト名などの情報を、[Application Program]セクションで設定します。
ワークユニットにアプリケーションを追加する場合、追加するアプリケーション単位に[Application Program]セクションに情報を記載して追加します。たとえば、ワークユニットに4つの別々なアプリケーションを動作させる場合には、4つの[Application Program]セクションを記載します。
ワークユニットからアプリケーションを削除する場合は、不要となるアプリケーションに対応する[Application Program]セクションを削除します。

非常駐アプリケーション情報

非常駐形態で動作させるアプリケーションの多重度などの情報を[Nonresident Application Process]セクションで設定します。

Solaris32

なお、非常駐形態で運用する場合は、[Control Option]セクションの"Environment Variable:"ステートメントに環境変数"EXTP_NONRESIDENT_OPTION=on"を設定してください。環境変数が設定されていない場合、サーバアプリケーションの処理が完了しても、サーバアプリケーションはメモリ上から消去されません。

マルチオブジェクト常駐アプリケーション情報

マルチオブジェクト常駐形態で動作させるアプリケーションの多重度などの情報を[Multiresident Application Process]セクションで設定します。

2.6 アプリケーションの登録

サーバアプリケーションをほかのアプリケーションからアクセス可能にするためには、目的のアプリケーションを識別するためのオブジェクトリファレンスを作成する必要があります。また、同時に作成したオブジェクトリファレンスをネーミングサービスに登録することによって、他のアプリケーションからのアクセスが可能になります。

トランザクションアプリケーションの場合、オブジェクトリファレンスの作成とネーミングサービスへの登録は、次の2つの方法で行うことができます。

- ワークユニット起動による自動登録
- OD_or_admコマンド(注)による手動登録

ワークユニット起動による自動登録の場合は、ワークユニットを起動するとワークユニットで指定された各アプリケーションごとにオブジェクトリファレンスが作成され、ネーミングサービスに登録されます。この場合、ワークユニットを停止すると、オブジェクトリファレンスは、自動的にネーミングサービスから削除され、無効となります。
OD_or_admコマンドによる手動登録の場合は、ワークユニットの起動前に、事前にOD_or_admコマンドによりオブジェクトリファレンスの作成とネーミングサービスへの登録を行います。この場合は、ワークユニットの起動、停止にかかわらず、オブジェクトリファレンスは有効です。

2つの方法とも、ネーミングサービスには以下の名前でオブジェクトリファレンスを登録します。

```
モジュール名::インタフェース名
```

また、インプリメンテーションリポジトリIDとしては、以下のIDを使用します。

Windows32 Solaris32

ワークユニット種別がORBの場合　　: FUJITSU-Interstage-TDLC
ワークユニット種別がWRAPPERの場合　: FUJITSU-Interstage-TDRC

Linux32

ワークユニット種別がORBの場合　　: FUJITSU-Interstage-TDLC

なお、2つの方法のどちらを使用するかは、ワークユニット定義で指定する必要があります。詳細は"OLTPサーバ運用ガイド"を参照してください。

注) ロードバランス機能を使用する場合は、odadministerlbコマンドを使用します。

注意

手動登録する場合、OD_or_admコマンドを実行する前に、Interstageが起動されている必要があります。OD_or_admコマンドの詳細については、"リファレンスマニュアル(コマンド編)"を参照してください。

2.7 アプリケーションのテスト

作成したアプリケーションのテスト方法を、以下の内容で説明します。

Solaris32

- サーバアプリケーションのテスト方法

Windows32 Solaris32 Linux32

- クライアントアプリケーションのテスト方法
- スナップショットによるテスト支援
- 運用環境への移行
- 運用環境におけるテスト

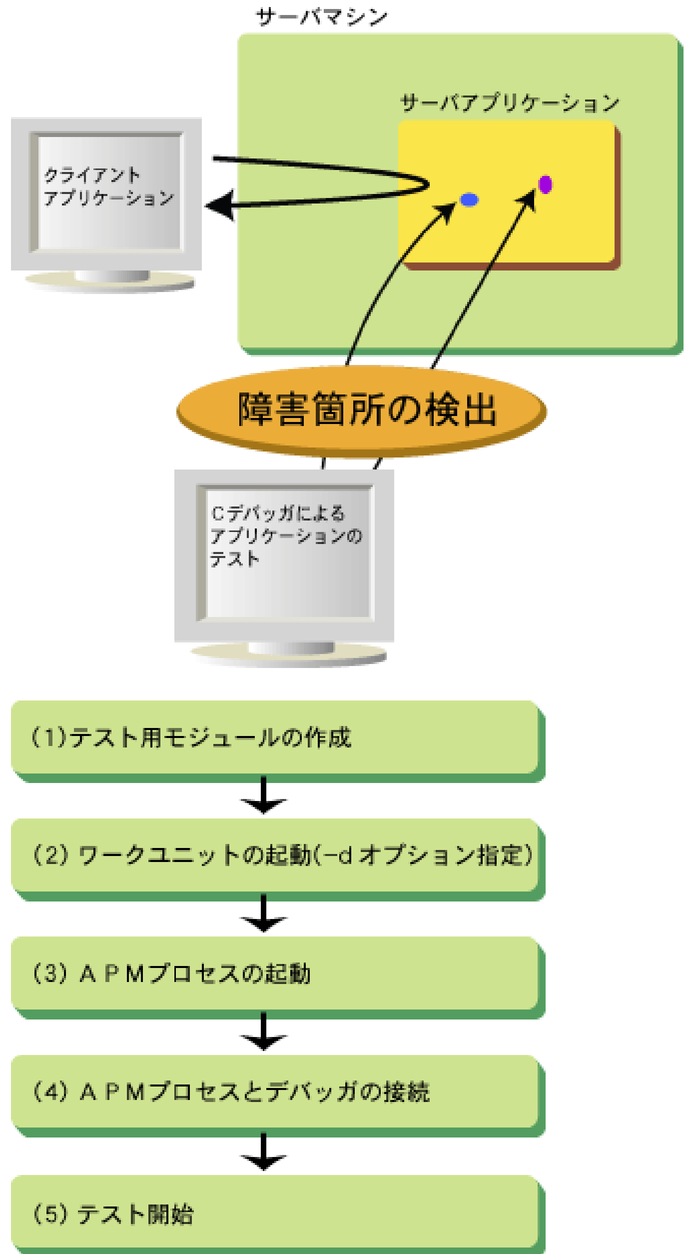
サーバアプリケーションのテスト方法 Solaris32

サーバアプリケーションのテストを行う場合、実際にクライアントアプリケーションと結合して行います。このとき、サーバアプリケーションをデバッガ配下で動作させることで、サーバアプリケーションが正しく作成されているか確認できます。

注意

サーバアプリケーションのテストは、Solarisの場合だけが可能です。

Cデバッガと連携するには、APM配下のアプリケーションをdbx配下などで動かせるようにする必要があります。Cデバッガと連携するときの動作の概要と、サーバアプリケーションをデバッガ配下でテストする場合の手順を、以下に示します。



テスト用モジュールの作成

テストを行うサーバアプリケーションを、-gオプションおよび-xsオプションを指定してコンパイルします。コンパイル方法の詳細については、関連するマニュアルを参照してください。
テスト用モジュールのコンパイル例を以下に示します。

```
%cc -c -g -xs -D_REENTRANT -I/opt/FSUNod/include -I/opt/FSUNtd/include tdsample1_s.c
```

ワークユニットの起動

テストを行うサーバアプリケーションのワークユニット定義を作成し、tdstartwuコマンドでデバッグオプション-dを指定してワークユニットを起動します。-dオプションを指定してワークユニットを起動した場合、ワークユニットの動作環境まで作成し、APMプロセスを起動せずにコマンドは復帰します。tdstartwuコマンドの詳細については“リファレンスマニュアル（コマンド編）”を参照してください。
デバッグ用のワークユニットの起動例を、以下に示します。

```
%tdstartwu -d TDSAMPLE1
```

APMプロセスの起動

起動したワークユニットのテスト対象のAPMプロセスを起動します。APMプロセスを起動する際に指定するパラメタの指定方法を以下に示します。

```
ファイル名  APMモジュール名  業務システム名 ワークユニット名 オブジェクト名 種別
```

ファイル名

"EXTPのインストールディレクトリ/FSUNextp/bin/extp_apmenv"を指定します。

APMモジュール名

"EXTPのインストールディレクトリ/FSUNextp/bin/extp_apmXXX"を指定します。
XXX : ワークユニット定義で指定したAPM名を指定します。
また、拡張システムの場合は、以下のように設定します。
XXX : “APM名_拡張システム名”を設定します。

業務システム名

デフォルトシステムの場合は、"td001"を指定します。拡張システムの場合は、システム名を設定します。

ワークユニット名

本APMが動作するワークユニット名を指定します。

オブジェクト名

ワークユニット定義で指定した本APMのオブジェクトを指定します。

動作システム種別

"T"を指定します。

APMプロセスの起動前に、ワークユニット定義で指定したすべての環境変数を設定しておいてください。
APMプロセスの起動例を以下に示します。

```
/opt/FSUNextp/bin/extp_apmenv  /opt/FSUNextp/bin/extp_apmTDNORM  td001
TDSAMPLE1  TDSAMPLE1/INTF  T
```

APMプロセスとデバッガの接続

起動したAPMプロセスのプロセスIDを指定してデバッガを起動します。Cデバッガの起動方法については、関連するマニュアルを参照してください。

テストの実施

クライアントアプリケーションからサーバアプリケーションを呼び出し、処理を実行することで、サーバアプリケーションの動作状態をデバッガから確認することができます。以上により、デバッガ配下でサーバアプリケーションを実行することができ、ステップ単位でデバッグすることができます。なお、Cプログラムのデバッグ方法の詳細については、関連するマニュアルを参照してください。

アプリケーションのテストを終了する場合は、以下の手順で行います。

- サーバアプリケーションのデバッグを終了し、クライアントアプリケーションからの要求待ちの状態にします。
- ワークユニットを停止します。
- デバッガを終了します。

Cデバッガにより、サーバアプリケーションをデバッグする場合の注意事項について示します。

- サーバアプリケーションをデバッガ配下で起動しデバッグを行っているときは、再デバッグ(rerun)を行わないでください。再デバッグを行う場合は一度デバッガを停止し、その後再起動してください。
- サーバアプリケーションのデバッグ中にワークユニットを停止しないでください。必ずサーバアプリケーションが復帰し、要求待ちの状態になっている状態でワークユニットを停止してください。
- ワークユニット定義におけるプロセス多重度には必ず1を指定してください。
- 1つのワークユニットに対しては、デバッガは1つだけ起動するようにしてください。
- デバッガを終了する場合は、必ずデバッガによる実行状態が完了している状態で行ってください。

- 後出口プログラムのデバッグ中は、ほかのワークユニットの起動処理または停止処理が保留されます。
- デバッガによるステップ実行中でも時間監視は行われますので、ワークユニット定義で時間監視を行わないか、またはデバッグに十分な時間を指定してください。
- デバッガを使用している場合は、tdstopwuコマンドで-cオプション、およびisstopコマンドで-c、-fおよび-s（Windows版）オプションは投入できません。
- ワークユニット定義のアプリケーション使用ライブラリパス、およびアプリケーションライブラリパスに指定されているパスを環境変数(LD_LIBRARY_PATH)に設定し、デバッガを起動してください。

クライアントアプリケーションのテスト方法

クライアントアプリケーションのテスト方法は、クライアントアプリケーションを動作させるオペレーティングシステムやミドルウェアによって異なります。お使いのオペレーティングシステムやミドルウェアごとに推奨される方法でテストを行ってください。

スナップショットによるテスト支援

スナップショットを使用して、クライアントからの要求に対する入出力情報をワークユニット単位に取得することにより、アプリケーションのデバッグを行うことができます。
詳細は、"第7章 スナップショット機能"を参照してください。

運用環境への移行

開発環境でテストした資材を運用環境へ移行するための作業手順と、運用環境でのテスト方法について説明します。

移行手順

クライアント資材の移行

クライアント資材は、サーバ資材を移行してサーバアプリケーションがコンパイルされるまでに移行してください。

サーバ資材の移行

サーバ資材を運用環境に移行する手順について、以下に示します。

1.Interstageシステムの停止

2.アプリケーションソースの移行

3.定義ファイルの移行

4.IDLファイルのコンパイルの実施

5.Interstageシステムの停止

6.サーバアプリケーションのコンパイルを実施

7.オブジェクトリファレンスの登録

8.ワークユニット定義の登録

9.Interstageシステムの起動

10.ワークユニットの起動

1. 開発環境のInterstageシステムを停止します。
2. 開発環境からサーバアプリケーションのプログラムソースを運用環境に複写します。
3. 開発環境からIDLファイル、ワークユニット定義ファイルを運用環境に複写します。
4. 3.で複写したIDLファイルをもとに、tdcコマンドを実行し、スタブファイル、スケルトンファイルを作成します。
5. 運用環境のInterstageシステムの停止
6. 2.で複写したサーバアプリケーションプログラムソースと4.で作成したスケルトンファイルにより、サーバアプリケーションを作成します。
7. ワークユニット定義の"ネーミングサービスの登録形態"が"MANUAL"の場合、オブジェクトリファレンスを登録します。
8. 3.で複写したワークユニット定義を登録します。
9. 運用環境のInterstageシステムを起動します。

10. 運用環境のワークユニットを起動します。

運用環境におけるテスト

開発環境では各モジュールの単体テストを行いますが、運用環境ではシステム全体として以下に示すテストが必要です。

- システム負荷テスト
- 業務に沿った運用テスト
- 業務に則した性能テスト

それぞれのテスト方法について、以下に示します。

システム負荷テスト

システム負荷テストは業務を遂行するために、システム上のすべてのコンポーネントを含めたテストを実施します。システムの負荷をあげるためには、システムへのデータ入力の頻度（呼量と呼びます）をあげると、実施できます。たとえば、多数のCORBAクライアントからの入力の場合、CORBAクライアントを高速なマシン上で動作させると、呼量が増加し、負荷があげられます。

Windows32 Solaris32

また、Web連携の負荷をあげるときは、WebStoneなどをサーバに設定し、同様に呼量を増加させることができます。

業務に沿った運用テスト

業務に沿った運用テストは、実際の業務を想定したテストを実施し、システムとして運用に問題がないかを確認します。したがって、システムで利用する製品および業務アプリケーションすべてを動作させます。たとえば、受注業務の運用テストを実施する場合、受注業務で利用する全製品とアプリケーションを動作させて、受注業務の開始/終了、データ入力とその処理などを実施します。

業務に則した性能テスト

性能テストは、業務運用中の性能について測定し、問題がないかを確認するテストです。

第3章 サーバアプリケーションの作成（C++言語）

ローカルトランザクション運用を行うためのアプリケーションの作成方法について、説明します。

3.1 サーバアプリケーションの開発

サーバアプリケーションを記述するために必要な以下の項目を説明します。

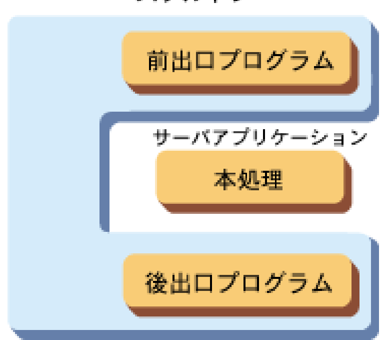
- サーバアプリケーションの構造
- サーバアプリケーションの入出力情報
- クライアントアプリケーションへの復帰値

Solaris32 Linux32

- サーバアプリケーションプロセスの形態

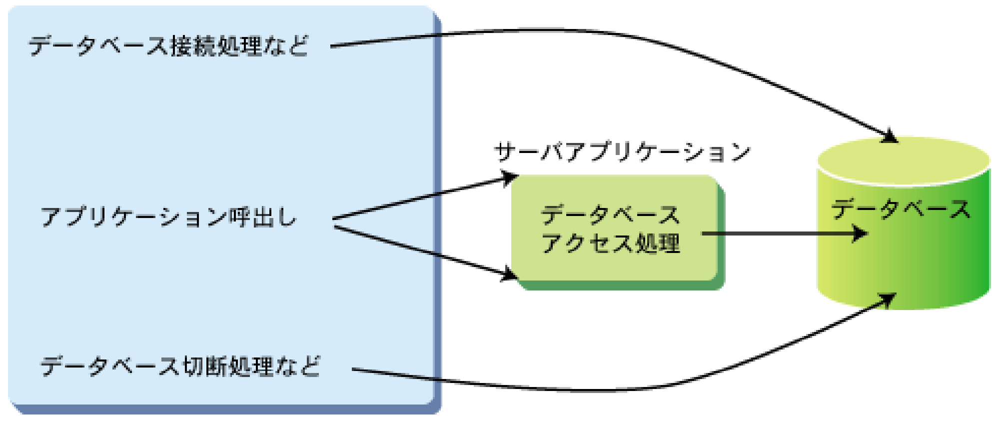
サーバアプリケーションの構造

一般のアプリケーションは、前処理、本処理、後処理という3つの構成で成り立っています。前処理として行う処理は、使用するデータベースの接続処理やデータベースのオープン処理などです。後処理として行う処理は、接続中データベースの切断処理やオープン中データベースのクローズ処理などです。前処理および後処理で行うことは、使用するデータベース管理システムごとに決まっています。そのため、Interstageでは前出口プログラムおよび後出口プログラムの機能を提供します。
スケルトンを使用することにより、サーバアプリケーション開発者は、本処理だけを開発することで、サーバアプリケーションの作成が可能となります。
サーバアプリケーションの構造を以下に示します。

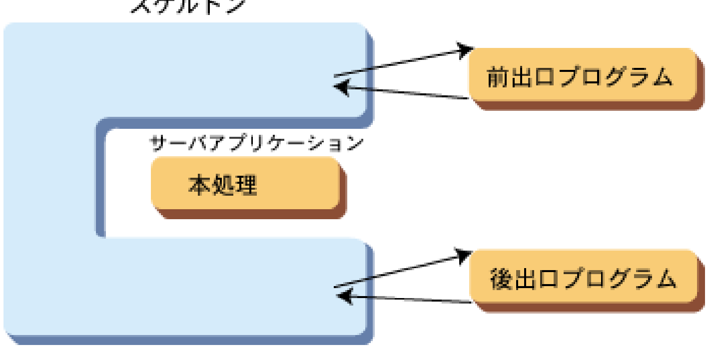


なお、データベース管理システムが提供するトランザクションを使用する場合には、サーバアプリケーション自身がデータベースとの接続や切断処理を行う必要があります。このような前処理および後処理を、出口プログラムとして本処理部分とは別に作成することができます。
この出口プログラムを作成することで、データ処理ごとに行っていたデータベースとの接続や切断処理を出口プログラムで行うことができるようになるため、効率の良い処理を構築することができます。

データベース管理システムが提供するトランザクションを使用する場合の、サーバアプリケーションの構造を以下に示します。



スケルトンとサーバアプリケーションの関係は、主プログラムと副プログラムという関係となります。また、サーバアプリケーションはスケルトンと静的結合により実行されます。スケルトンとサーバアプリケーションの関係について以下に示します。



サーバアプリケーションと、一般のアプリケーションとの違いを以下に示します。

- 副プログラムとして作成する必要があります。
- スレッドに関する命令は記述できません。
- 実行プログラムは、スケルトンと静的結合した実行形式プログラムとする必要があります。

Solaris32 Linux32

- シグナルに関する命令は記述できません。

注意

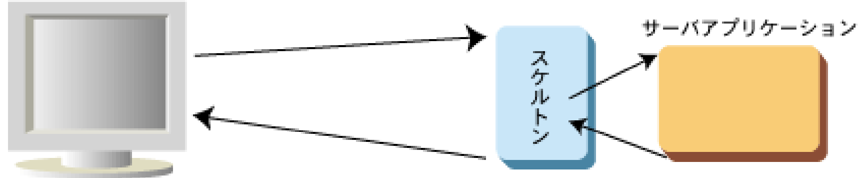
前出口プログラムは、ワークユニット起動時にアプリケーションプロセス単位で実行され、後出口プログラムは、ワークユニット停止時にアプリケーションプロセス単位で実行されます。ただし、ワークユニット強制停止時およびワークユニット異常終了時は、後出口プログラムは実行されません。
このとき、前出口プログラムおよび後出口プログラムの復帰値が0の場合は正常、0以外の場合は異常とします。復帰値が0以外の場合、ワークユニット起動時では、ワークユニットの起動失敗となり、ワークユニット停止時では、警告メッセージを出力し、ワークユニット停止処理は正常に終了します。
また、サーバアプリケーションの前処理、後処理の復帰値はすべてlong型にする必要があります。
以下の処理はInterstageが行うため、サーバアプリケーションで考慮する必要はありません。

- CORBAの初期化メソッドの呼び出し
- メソッドの登録
- サーバの活性化/非活性化

ただし、サーバアプリケーションが例外を使用する場合は、CORBAの初期化メソッドの呼び出しが必要です。

サーバアプリケーションの入出力情報

サーバアプリケーションは、クライアントからの呼び出し時に、スケルトンを経由して、IDLで定義した入力インタフェースで呼び出されます。また、サーバアプリケーションの処理が完了すると、スケルトンを経由してIDLで定義した出力インタフェースがクライアントに返却されます。
サーバアプリケーションの動作概要を以下に示します。



サーバアプリケーションの入出力インタフェースはCORBAで規定されています。CORBAのデータ型と言語とのマッピングを以下に示します。

CORBAデータ型			型宣言
基本データ型	整数型	・long long ・long ・unsigned long ・short ・unsigned short	・CORBA::LongLong ・CORBA::Long ・CORBA::ULong ・CORBA::Short ・CORBA::UShort
	浮動小数点型	・float ・double ・long double	・CORBA::Float ・CORBA::Double ・CORBA::LongDouble
	文字型	・char ・wchar	・CORBA::Char ・CORBA::WChar
	オクテット	・octet	・CORBA::Octet
	ブーリアン	・boolean	・CORBA::Boolean
	文字列型	・string ・wstring	・CORBA::Char* ・CORBA::WChar*
構造データ型	構造体	・struct	――――――――
	配列	――――――――	――――――――
	シーケンス	・sequence	――――――――

サーバアプリケーションからスケルトンに、outパラメタまたはinoutパラメタのstring型のデータを受け渡す場合はTD::string_alloc、wstring型のデータを受け渡す場合はTD::wstring_allocにより領域を獲得する必要があります。また、string、wstring型データおよび構造データ型の場合には、サーバのアプリケーション用APIを利用する必要があります。サーバのアプリケーション用APIの詳細は、C++の提供クラスを参照してください。
なお、サーバアプリケーションでは、コンテキスト情報を使用することはできません。サーバアプリケーションに対しては、第1引数と最後の引数でシステムパラメタ、第2引数以降IDLで定義したパラメタというインタフェースでデータの入出力を行います。

また、IDL定義にoutパラメタを定義している場合、サーバアプリケーションが処理を終了する際には、必ずoutパラメタに復帰データを設定しなければなりません。outパラメタに復帰データが設定されていない場合、サーバアプリケーションプロセスが異常終了することがあります。

クライアントアプリケーションへの復帰値

クライアントアプリケーションへの復帰値は、Interstageにより規定されています。

復帰値	意味	アプリケーションの指定可否
0～10000	サーバアプリケーション任意	可
10001	システム異常検出(メモリ不足等)	不可
10002	システムで異常検出	不可
Windows32 Solaris32 10003	AIM連携のホスト間通信処理で異常検出	不可
10004	以下のいずれかの異常を検出 ・ サーバアプリケーションでの異常または、 ・ プロセスバインド機能使用時にセションIDが参照できないまたは、 ・ プロセスバインド機能使用時にスケルトンで領域処理獲得に失敗または、 ・ オペレーション名が不当	不可
Windows32 Solaris32 10005	AIM連携のセション継続機能において異常検出	不可
10006	アクセス権がないユーザからの要求	不可
10007	プロセスバインド機能において異常検出(注)	不可
10008	最大キューイング数に達した	不可

(注)IPCOMによる負荷分散を行っている環境でプロセスバインド機能を使用した場合にも復帰します。

復帰値0～10000

クライアントアプリケーションとサーバアプリケーションで任意に使用することができます。Interstageはこれらの復帰値を正常とみなします。

復帰値10001～10008

Interstageがクライアントに通知する復帰値であり、システムで異常が検出されたことを意味します。クライアントに復帰値10001～10008が通知された場合は、同時にその異常の詳細がエラーログに記録されます。

また、復帰値が10004で、エラーログに詳細が出力されていない場合は、クライアントから呼び出したオペレーションが、サーバのスケルトンに定義されていない(上記表の「オペレーション名が不当」に該当する)可能性があります。クライアントのスタブとサーバのスケルトンが同一のIDL定義から生成されているか、不一致がないかを確認してください。

なお、エラーログは、以下のファイルに採取されます。エラーログの内容については“メッセージ集”の“コンポーネントトランザクションサービスが出力するログメッセージ”を参照してください。

Windows32

```
INTERSTAGEインストールフォルダ¥td¥trc¥lorb¥errlog0
(トランザクションアプリケーションの場合)

INTERSTAGEインストールフォルダ¥td¥trc¥rorb¥errlog0
(AIM連携の場合)
```

Solaris32

```
/var/opt/FSUNtd/trc/lorb/errlog0
(トランザクションアプリケーションの場合)
```

```
/var/opt/FSUNtd/trc/rorb/errlog0
（AIM連携の場合）
```

Linux32

```
/var/opt/FJSVtd/trc/lorb/errlog0
(トランザクションアプリケーションの場合)
```

注意

クライアントアプリケーションでは、復帰値を参照する前に例外を判定する必要があります。例外が発生している場合は、復帰値は不定となります。クライアントアプリケーションでの例外処理については、“アプリケーション作成ガイド(CORBAサービス編)”を参照してください。

サーバアプリケーションプロセスの形態 Solaris32 Linux32

サーバアプリケーションプロセスには以下の2つの形態があります。

スレッドモード

サーバアプリケーションプロセスがマルチスレッドで動作する形態です。ただし、サーバアプリケーションはプライマリスレッド上でのみ動作し、マルチスレッドでは動作しません。スレッドモードではInterstageの制御用スレッドが複数起動します。

通常はスレッドモードを使用してください。

プロセスモード

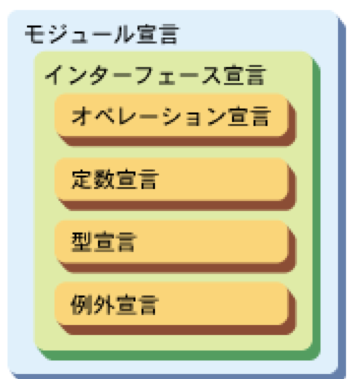
サーバアプリケーションプロセスがシングルスレッドで動作する形態です。本モードは、サーバアプリケーションより呼び出される他製品のライブラリが、マルチスレッドで動作するプロセス上で呼び出すことができない場合にのみ使用してください。

3.2 IDLファイルの作成

ローカルトランザクション運用におけるIDLファイルの文法の概要について説明します。IDLファイルの文法の詳細については、“アプリケーション作成ガイド（CORBAサービス編）”を参照してください。

IDLファイルの記述形式

ローカルトランザクション運用におけるIDLファイルの記述形式を、以下に示します。



モジュール宣言

モジュール宣言では、IDL定義のオペレーション名や型名などが、ほかのIDL定義と重複しないように、オブジェクトのグループ化を定義します。
モジュール宣言の書式を以下に示します。

```
module モジュール名  {
        interface インタフェース名m {
             ・・
       } ;
```

```
        interface インタフェース名n {
            ・・
        } ;
    } ;
```

モジュール宣言の中に新たなモジュール宣言をすることもできます。これをモジュール宣言の入れ子と呼びます。モジュール宣言のmodule A宣言内にmodule B宣言をする例を以下に示します。

```
module A  {
  interface a1 {
      ・・
  } ;
  module B  {
     interface b1 {
      ・・
     } ;
  } ;
} ;
```

インタフェース宣言

インタフェース宣言では、アプリケーションへの入力と出力を規定するためのインタフェースを定義します。
インタフェース宣言の書式を以下に示します。

```
interface  インタフェース名[:継承するインタフェース名]  {
      オペレーション宣言 ;
      定数宣言 ;
      型宣言 ;
      例外宣言 ;

 } ;
```

※ [ ]内は省略可能です。

インタフェース宣言は、モジュール宣言内に複数個記述できます。トランザクション運用では、モジュール宣言を記述せずにインタフェース宣言を記述することはできません。

オペレーション宣言

オペレーション宣言では、アプリケーションに対応したオペレーション名、復帰値の型、パラメタのデータ型を定義します。
オペレーション宣言の書式を以下に示します。

```
復帰値のデータ型  オペレーション名（
  [ パラメタタイプ  データ型  パラメタ名[ , . . ]]
）  [  raises  (例外構造体名[ , . .])）] ;
```

※ [ ]内は省略可能です。

オペレーション宣言は、インタフェース宣言内に複数個記述できます。

定数宣言

インタフェースの定義内で使用する定数を定義します。
定数宣言の書式を以下に示します。

```
const データ型 定数名 ＝ 定数式;
```

データ型と型宣言

トランザクション運用で使用できるデータ型とその型宣言について示します。

基本データ型の宣言

データ型宣言では、インタフェース内で使用されるデータ型を定義できます。基本データ型のうち、整数型、浮動小数点型、文字型、オクテット型、ブーリアン型については、型宣言にtypedefを利用することもできます。

文字列型を除く基本データ型宣言の書式を以下に示します。

```
typedef long long        データ型名 ;
typedef long             データ型名 ;
typedef short            データ型名 ;
typedef unsigned long    データ型名 ;
typedef unsigned short   データ型名 ;
typedef float            データ型名 ;
typedef double           データ型名 ;
typedef long double      データ型名 ;
typedef char             データ型名 ;
typedef wchar            データ型名 ;
typedef octet            データ型名 ;
typedef boolean          データ型名 ;

long long                データ名 ;
long                     データ名 ;
short                    データ名 ;
unsigned long            データ名 ;
unsigned short           データ名 ;
float                    データ名 ;
double                   データ名 ;
long double              データ名 ;
char                     データ名 ;
wchar                    データ名 ;
octet                    データ名 ;
boolean                  データ名 ;
```

文字列型の型宣言の書式を以下に示します。

```
typedef string<サイズ>    データ型名;
typedef string           データ型名;
typedef wstring<サイズ>   データ型名;
typedef wstring          データ型名;

string<サイズ>            データ名 ;
string                   データ名 ;
wstring<サイズ>           データ名 ;
wstring                  データ名 ;
```

構造体

構造体の宣言の書式を以下に示します。

```
struct  データ名 {
          構造体メンバの宣言
 };
```

構造体メンバの宣言の書式を以下に示します。

```
基本データ型 メンバ名 ;
```

構造体メンバは1つ以上必要であり、空の構造体は定義できません。

構造体の記述例を以下に示します。

```
module A {
        struct S {
                string  name;
                short   number;
                long    value;
        };
 };
```

配列

配列宣言では1次元固定長の配列を定義します。配列宣言の書式を以下に示します。

```
typedef データ型 識別子 [配列サイズ];
```

配列の次元は1階層までに制限されています。配列サイズは正の整数定数式で指定します。配列サイズはコンパイル時に固定されます。

配列宣言の例を以下に示します。

```
typedef long A[5];
```

シーケンス型

IDL言語でシーケンス型sequenceを指定した場合、C++言語では最大長_maximum、シーケンス長_length、バッファポインタ_bufferをprivateデータに持ったC++のクラスで定義します。シーケンスのデータを獲得するための関数("モジュール名::インタフェース名::シーケンス名::allocbuf"で定義された関数(以降XX::allocbuf関数と呼びます))がTDコンパイラで生成されます。

```
class データ名
{
        public:
                データ名();                     //デフォルトコンストラクタ
                データ名( CORBA::ULong max);   //maximumコンストラクタ
                データ名( CORBA::ULong max,
                        CORBA::ULong length,
                        CORBA::Type *data,
                        CORBA::Boolean release = CORBA_TRUE );
//T *dataコンストラクタ
                データ名( const sampleseq &s );
                                //コピーコンストラクタ
                データ名();   //デストラクタ
                static CORBA::Type *allocbuf( CORBA::ULong );
                                // allocbuf
                static void freebuf( CORBA::Type* );
                                // freebuf
                sampleseq &operator=( const sampleseq &s );
                                //割当てオペレータ
                CORBA::ULong maximum() const;
                                // maximumアクセス関数
                void length( CORBA::ULong );
                                // lengthアクセス関数
                CORBA::ULong length() const;
                                // lengthアクセス関数
                CORBA::Type &operator[]( CORBA::ULong index );
                                // _bufferのindex番目の要素を取得
                const CORBA::Type &operator[]( CORBA::ULong index )
                                             const;
                                // _bufferのindex番目の要素を取得
        private:
                CORBA::ULong _maximum;          // 配列の最大個数
                CORBA::ULong _length;           // 配列の個数
                CORBA::Type *_buffer;           // 配列の値
```

```
        CORBA::Boolean _release;        // releaseフラグ
};
```

シーケンス型の宣言の書式を以下に示します。

```
typedef sequence<データ型名 > データ名;
```

または

```
typedef sequence<データ型名 , シーケンス要素数 > データ名;
```

シーケンス型の要素として指定するデータ型は基本データ型だけに制限されています。
シーケンス型の記述例を以下に示します。

```
typedef sequence<long > Q;
```

例外宣言

例外宣言では、オペレーション実行中に例外が発生したときに例外情報を受け渡すための例外構造体名を定義します。オペレーション宣言のraises式を定義する場合は、この例外構造体名を指定します。
例外宣言の書式を以下に示します。

```
exception 例外構造体名 {
        データ型   メンバ名 ;      //構造体メンバを宣言
                   :
} ;
```

構造体メンバを複数定義する場合は、各メンバをセミコロン(“;”)で区切ります。メンバとして定義できるデータ型を以下に示します。

- 基本データ型
- 配列(配列要素のデータ型は基本データ型のみ、次元数は1次元まで)

また、例外にはシステムでの異常終了を通知するシステム例外とサーバアプリケーションでの異常終了を通知するユーザ例外がありますが、トランザクションアプリケーションではシステム例外は使用できません。
トランザクションアプリケーションでは、CORBAの初期化メソッドの呼び出しは必要ありませんが、例外を使用する場合、CORBAの初期化メソッドの呼び出しが必要となります。

例外の使用方法については、“アプリケーション作成ガイド(CORBAサービス編)”の各言語ごとのアプリケーションの例外処理を参照してください。

注意

コンポーネントトランザクションサービスにおける利用範囲

トランザクション運用で使用するIDLファイルの文法は、OMGで規定しているIDLの文法に準拠していますが、以下に示す項目が制限されています。

- モジュール宣言などでは、次の宣言が利用できません。
  - 属性宣言
  - コンテキスト
- 以下のデータ型は使用できません。
  - 基本データ型の列挙型(enum)とany型
  - 共用体型
  - オペレーション宣言において、利用できる型は以下に限られています。
    - long

- oneway void

データ型の使用方法

各データ型の使用方法については、"アプリケーション作成ガイド(CORBAサービス編)"の各言語ごとのデータ型に対するマッピングを参照してください。

なお、クライアントアプリケーションにVisual Basicを使用し、サーバアプリケーションにOLE CORBA-ゲートウェイ経由で文字列型のデータを通信する場合には、IDL定義の記述に注意が必要です。"アプリケーション作成ガイド(CORBAサービス編)"のCOM/CORBA連携プログラミングを参照してください。

CORBAアプリケーションとトランザクションアプリケーションでは、使用するAPIが以下に示す点で異なります。

```
CORBA::string_alloc ==> TD::string_alloc
CORBA::string_free  ==> TD::string_free
CORBA::wstring_alloc ==> TD::wstring_alloc
CORBA::wstring_free  ==> TD::wstring_free
```

オペレーションの使用方法

トランザクションアプリケーションでは、オペレーションの型として、longおよびoneway voidのみ用意しています。

oneway voidの場合、以下の注意が必要となります。

- オペレーションを実装するプログラムは、void型となります。
- クライアントアプリケーションへの復帰は、サーバアプリケーションの状態にかかわらず、サーバアプリケーションへデータを送信し、送信処理が成功した時点で正常復帰します。
- outパラメタ、inoutパラメタは使用できません。
- ユーザ例外を使用できません。
- グローバルトランザクション連携機能は使用できません。

3.3 サーバアプリケーションのソースの作成

サーバアプリケーションの処理を、オペレーティングシステムに添付されているエディタなどを使用して記述します。
ローカルトランザクション運用の場合は、リソースマネージャが提供するトランザクション命令と、リソースマネージャを使用するために必要なデータベースの結合処理や切り離し文などを記述してください。記述するトランザクション命令については、使用するリソースマネージャのマニュアルを参照してください。
IDL定義とサーバアプリケーション名の関係を以下に示します。

```
モジュール名_インタフェース名_impl::オペレーション名
```

サーバアプリケーションにインクルードするヘッダファイル

IDLコンパイルにより生成される以下のヘッダファイルをサーバアプリケーションにインクルードしてください。

ヘッダファイル	種別
TD_ IDLファイル名.h	スケルトン用ヘッダファイル
IDLファイル名.h	スタブ用ヘッダファイル(注)

(注)
中継用サーバアプリケーションを作成するときに必要です。スタブ側のIDLファイルのコンパイルにより生成されます。

出口プログラムの作成

出口プログラムは以下の形式で作成してください。なお、ここでの出口プログラムとは、前出口プログラム、後出口プログラム、および、プロセスバインド機能を使用する場合の異常出口プログラムを指します。

引数	なし
復帰値	long型 正常終了:0 異常終了:0以外

復帰値が0以外の場合は、出口プログラム異常終了とみなし、出口プログラム種別ごとに以下の動作となります。

- 前出口プログラムの場合、ワークユニットの起動に失敗します。
- 後出口プログラムの場合、警告メッセージを出力し、ワークユニットの停止処理は成功します。
- プロセスバインド機能の異常出口の場合、アプリケーションプロセスが異常終了します。その後、アプリケーション異常時の自動再起動が設定されている場合は、アプリケーションプロセスが自動再起動され、自動再起動が設定されていない場合は、ワークユニットが異常終了します。

以下に出口プログラムの設定方法について示します。

出口プログラムの設定方法

C++言語で出口プログラムを使用する場合、以下の方法を使用してください。C++言語の場合は、C言語またはCOBOL(COBOLは、Windows(R)版、Solaris版のみ)のように、ワークユニット定義に前出口プログラム名、後出口プログラム名および異常出口プログラム名を記述する必要はありません。ただし、出口プログラム最大処理時間については、ワークユニット定義に記述してください。
なお、異常出口プログラムは、プロセスバインド機能を使用する場合にのみ有効となります。

ヘッダの修正

tdcコマンドによりスケルトンのソースを生成する場合、スケルトンのソースと共に、以下の形式の名前でincludeファイルが出力されます。

```
TD_オブジェクト名_proto.h
```

includeファイルの修正

出口プログラムを使用する場合、TD_オブジェクト名_proto.hのファイルを修正する必要があります。

[前出口プログラムを使用する場合]

前出口プログラムを使用する場合は、includeファイル内の"long ApmInit();"のコメントを削除してください。

```
class ExtpApmUser:public ExtpApmBase{

 public:
   long  ApmInit();

 //long ApmRecover();
 //long ApmDestroy();
 };
```

[後出口プログラムを使用する場合]

後出口プログラムを使用する場合は、includeファイル内の"long ApmDestroy();"のコメントを削除してください。

```
class ExtpApmUser:public ExtpApmBase{

 public:
   //long  ApmInit();

   //long ApmRecover();
   long ApmDestroy();
 };
```

[異常出口プログラムを使用する場合]

異常出口プログラムを使用する場合は、includeファイル内の"long ApmRecover();"のコメントを削除してください。

```
class ExtpApmUser:public ExtpApmBase{

 public:
   //long  ApmInit();

   long ApmRecover();
   //long ApmDestroy();
 };
```

[前出口プログラム、後出口プログラムおよび、異常出口プログラムを使用する場合]

前出口プログラム、後出口プログラムおよび、異常出口プログラムを使用する場合は、includeファイル内の"long ApmInit();"、"long ApmDestroy();"および、"long ApmRecover();"のコメントを削除してください。

```
class ExtpApmUser:public ExtpApmBase{

 public:
   long ApmInit();
   long ApmRecover();
   long ApmDestroy();
 };
```

出口プログラムの作成

前出口プログラム名は"ApmInit"、後出口プログラム名は"ApmDestroy"、異常出口は"ApmRecover"という固定名で作成する必要があります。そのため、出口プログラムを作成する場合、以下のように記述する必要があります。

[前出口プログラムを作成する場合]

```
#include "TD_オブジェクト名_proto.h"

 long ExtpApmUser::ApmInit(){

   // 前出口の処理
 };
```

[後出口プログラムを作成する場合]

```
#include "TD_オブジェクト名_proto.h"
 long ExtpApmUser::ApmDestroy(){

   // 後出口の処理
 };
```

[異常出口プログラムを作成する場合]

```
#include "TD_オブジェクト名_proto.h"
 long ExtpApmUser::ApmRecover(){

   // 異常出口の処理
 };
```

注意

- サーバアプリケーションにおいて、引数に指定するオブジェクト情報は設定されていません。

- セション情報管理機能を使用する場合は、“5.3 セション情報管理機能を使用したトランザクションアプリケーションの作成”を参照してください。
- トランザクションアプリケーションにおいて、イベントサービスを利用する場合、使用可能なイベントサービスの機能は以下になります。
  - Pushモデルのサプライヤ

Microsoft(R) Visual C++ .NETまたは、Microsoft(R) Visual C++ 2005を使用してアプリケーションを作成する場合の注意点 Windows32

アプリケーションより標準出力または標準エラー出力に向けて出力されたデータは、カレントフォルダ配下のstdoutファイルまたはstderrファイルに出力されます。
しかし、Microsoft(R) Visual C++ .NETまたは、Microsoft(R) Visual C++ 2005を使用してビルドされたアプリケーションでは、標準出力または標準エラー出力に向けて出力されたデータが、カレントフォルダ配下のstdoutファイルまたはstderrファイルに出力されません。
これを回避し正しく出力するためには、アプリケーションにおいて以下の対処を実施してください。

プログラムの先頭に以下のコードを追加してください。(注1)

```
freopen("stdout", "w", stdout);
freopen("stderr", "w", stderr);
```

なお、前出口プログラムを使用される場合は、前出口プログラムの先頭に追加してください。前出口プログラムに追加した場合、本処理および後出口プログラムへの対処は必要ありません。
前出口プログラムを使用されない場合は、本処理の先頭に追加し、かつ、初回呼び出し時のみ実行するよう対処してください。

注1) Microsoft(R) Visual C++ 2005を使用してビルドした場合、“warning C4996: 'freopen' が古い形式として宣言されました。”という警告が出力される場合がありますが、動作上の問題はありません。

3.4 ソースのコンパイル・リンク

3.4.1 IDLファイルのコンパイル

IDLファイルをコンパイルすることにより、クライアント、サーバそれぞれのアプリケーションの言語に合わせたスタブファイルとスケルトンファイルが作成されます。IDLファイルのコンパイルには、tdcコマンドを使用します。tdcコマンドのオプションは、使用する言語により異なります。tdcコマンドの詳細については、“リファレンスマニュアル(コマンド編)”を参照してください。tdcコマンドを使用した例を以下に示します。

Windows32

```
>tdc -cpp -mvcpp tdsample1.idl
```

Solaris32

```
%OD_HOME=/opt/FSUNod
%export OD_HOME
%TD_HOME=/opt/FSUNtd
%export TD_HOME
%tdc -cpp -mcpp tdsample1.idl
```

Linux32

```
%OD_HOME=/opt/FJSVod
%export OD_HOME
%TD_HOME=/opt/FJSVtd
%export TD_HOME
%tdc -cpp -mcpp tdsample1.idl
```

注意

トランザクションアプリケーションでは、インタフェースリポジトリにインタフェース情報を登録する必要があります。インタフェースリポジトリに登録したインタフェース情報とスタブに埋め込まれるインタフェース情報は必ず一致している必要があります。IDLコンパイラでインタフェース情報を生成する際には、インタフェース情報チェック機能を使用することを推奨します。インタフェース情報チェック機能についての詳細は、"OLTPサーバ運用ガイド"の"インタフェース情報チェック機能を使用した運用"を参照してください。

アプリケーション間連携時に、以下の条件を満たす場合、コンパイル時に関数の二重定義エラーが発生します。

- 1つのクライアントアプリケーションから、複数のサーバアプリケーションを呼び出し可能とする。
- 各々のサーバアプリケーションで使用するIDL定義に同一種別のデータ型(基本データ型は除く)が宣言されている。

この場合は出力された関数名を変更するか、IDLcコマンド(-lcまたは-lsオプションを指定)を使用してスタブファイルを生成してください。IDLcコマンドの詳細については、"リファレンスマニュアル(コマンド編)"を参照してください。

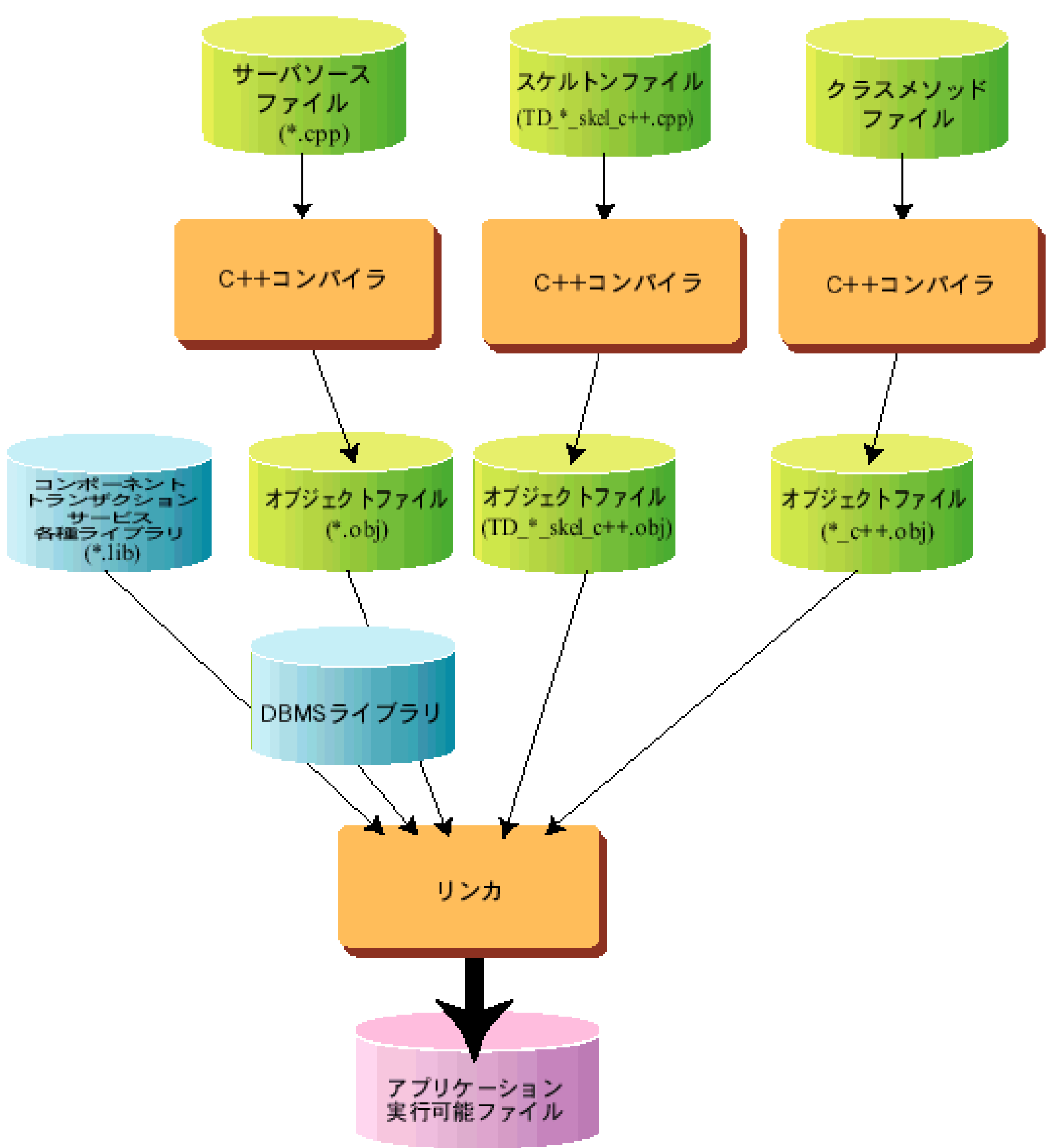
3.4.2 スタブとクライアントアプリケーションのソースとのコンパイル・リンク

記述したクライアントアプリケーションのソースとスタブをコンパイルします。コンパイル方法の詳細については、クライアントアプリケーションを動作させるオペレーティングシステムやミドルウェアのマニュアルを参照してください。

3.4.3 スケルトンとサーバアプリケーションのソースとのコンパイル・リンク

サーバアプリケーションのソースとスケルトンのコンパイル方法、およびリンク方法について説明します。
C++言語でアプリケーションを作成した場合について、コンパイルとリンクの手順の流れを以下に示します。

Windows32



サーバアプリケーション実行可能ファイルは、スケルトンとサーバアプリケーションが使用するライブラリで静的に結合します。データベース管理システムのライブラリ以外で、サーバアプリケーションのリンク時に指定するライブラリを以下に示します。

[リンク時に指定するライブラリ]

ライブラリ名	格納場所	用途
odsvcpp.lib	INTERSTAGEインストールフォルダ¥odwin¥lib	CORBAサービスランタイム(注)
odwincpp.lib	INTERSTAGEインストールフォルダ¥odwin¥lib	CORBAサービスランタイム(必須)
f3fmalcapi.lib	INTERSTAGEインストールフォルダ¥td¥lib	コンポーネントトランザクションサービスランタイム(必須)
f3fmapicpp.lib	INTERSTAGEインストールフォルダ¥td¥lib	コンポーネントトランザクションサービスランタイム(必須)
libextpapiskl.lib	INTERSTAGEインストールフォルダ¥extp¥lib	コンポーネントトランザクションサービスランタイム(必須)
libextpapmcom.lib	INTERSTAGEインストールフォルダ¥extp¥lib	コンポーネントトランザクションサービスランタイム(必須)

ライブラリ名	格納場所	用途
libextpapmbase.lib	INTERSTAGEインストールフォルダ¥extp¥lib	コンポーネントトランザクションサービスランタイム(必須)
LibextpapmlibTDNORM.lib	INTERSTAGEインストールフォルダ¥extp¥lib	コンポーネントトランザクションサービスランタイム(必須)

(注)
中継用サーバアプリケーションを作成するとき、および、例外を使用するときに必要です。
odsvcpp.libをリンクする場合、odwincpp.libをリンクする必要はありません。

サーバアプリケーションとスケルトンをVisual C++でコンパイル後、リンクする場合のオプションの例を以下に示します。

[コンパイルオプションの例]

Visual C++でコンパイルする場合は、[プロジェクト]-[設定]-[C/C++]の「カスタマイズ」で、以下の表に示すオプションを設定してください。

カテゴリー	項目	設定値
コード生成	CPU	80386を推奨
	構造体メンバのアライメント	4バイト
	使用するランタイム	マルチスレッド(DLL)
最適化		デフォルトを推奨
プリプロセッサ	プリプロセッサの定義	"OM_PC","OM_WIN32_BUILD", "__STDC__" を追加

また、[ツール]-[オプション]-[ディレクトリ]の「インクルードファイル」、「ライブラリファイル」にそれぞれINTERSTAGEインストールフォルダの下の"include"、"lib"フォルダを登録してください。

登録例)

(INTERSTAGEをC:¥INTERSTAGEにインストールした場合)

- インクルードファイル
  C:¥INTERSTAGE¥odwin¥include
  C:¥INTERSTAGE¥td¥include
  C:¥INTERSTAGE¥extp¥include
- ライブラリファイル
  C:¥INTERSTAGE¥odwin¥lib
  C:¥INTERSTAGE¥td¥lib
  C:¥INTERSTAGE¥extp¥lib

リンク時の注意事項

サーバアプリケーションのリンク時には、上記のライブラリのほかに、データベースや連携する他製品のライブラリなど、サーバアプリケーションで必要なライブラリがすべてリンクされていることを確認してください。また、リンクしたライブラリの格納先をワークユニット定義のアプリケーション使用パス(Path for Applicationセクション)に設定してください。
リンクが完全でない場合やワークユニット定義のアプリケーション使用パスが完全でない場合は、ワークユニット起動処理において実行ファイルのオープン処理に失敗し、メッセージ(EXTP4640,EXTP4643,EXTP4645,EXTP4508)が出力されます。

以下に必要なワークユニット定義の設定例を示します。

ワークユニット定義の設定例

サーバアプリケーションの実行ファイルが"application1.exe"、格納先がc:¥application¥bin、サーバアプリケーションにリンクされているライブラリの格納先が、それぞれ、"d:¥product1¥bin"、"e:¥product2¥bin"の場合の定義例を以下に示します。

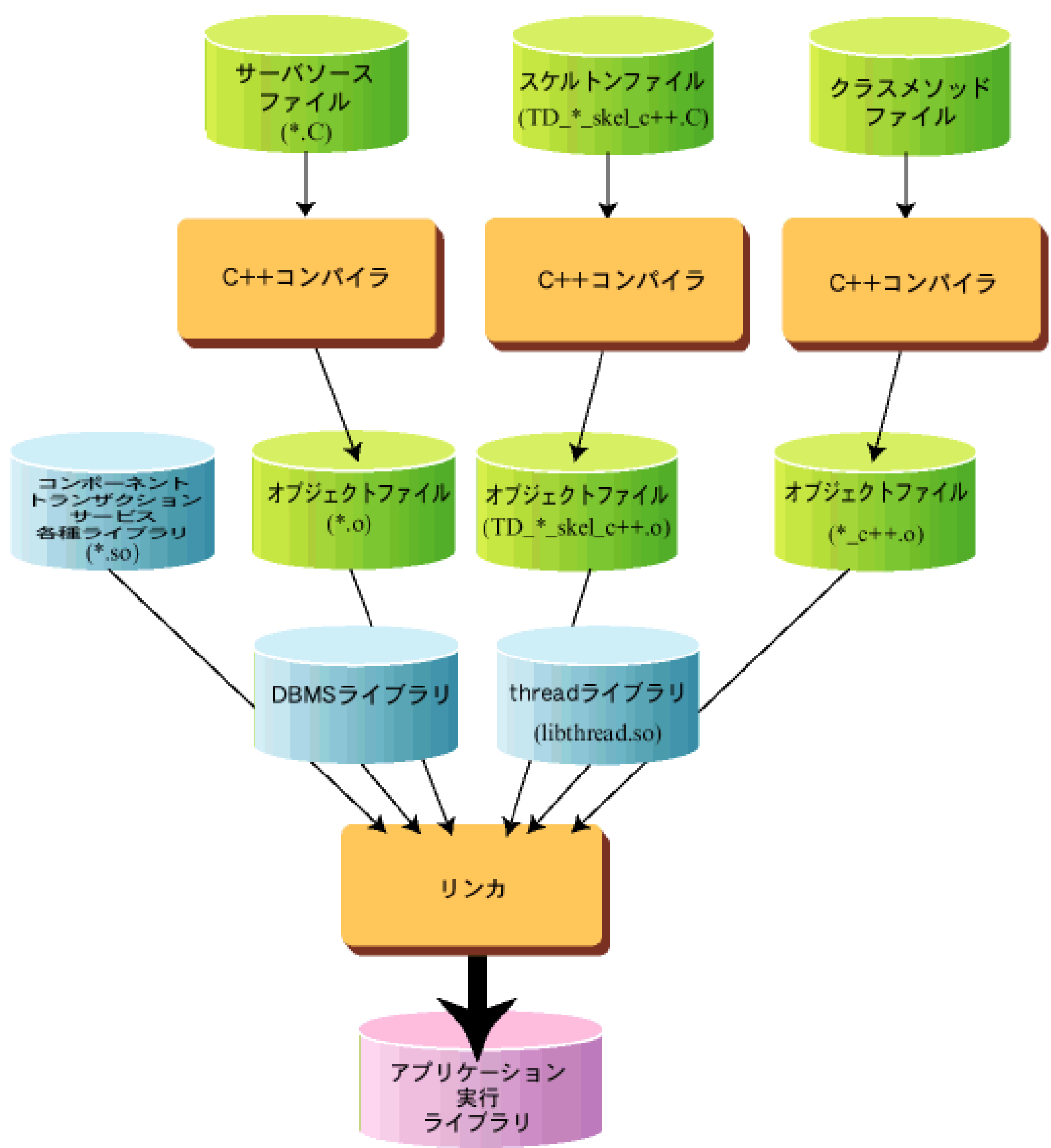
```
[WORK UNIT]
Name: SAMPLEWU
Kind: ORB
```

```
[APM]
Name: TDNORM

[Control Option]
…
Path: c:¥application¥bin
Path for Application: d:¥product1¥bin
Path for Application: e:¥product2¥bin
…

[Application Program]
…
Executable File : application1.exe
```

サーバアプリケーションにリンクされているライブラリの格納先が複数存在する場合は、アプリケーション使用パス(Path for Applicationセクション)を複数設定してください。

また、出口プログラムにリンクされているライブラリの格納先についても、同様にアプリケーション使用パス(Path for Applicationセクション)に設定してください。

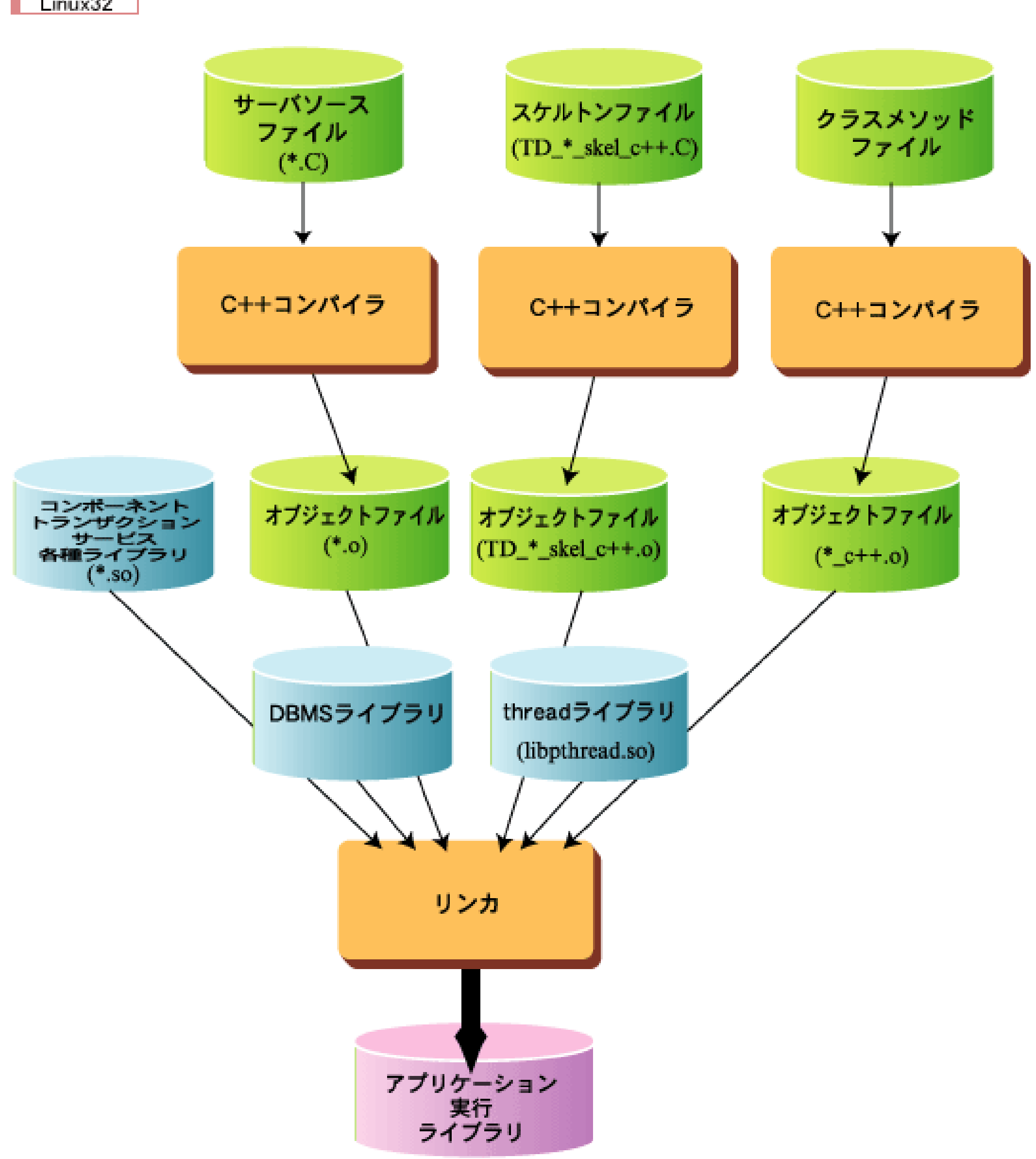
Solaris32



注意

スレッドライブラリ(libthread.so)は、スレッドモードで作成する場合のみ必要です。

Linux32



注意

スレッドライブラリ(libpthread.so)は、スレッドモードで作成する場合のみ必要です。

サーバアプリケーション実行可能ファイルは、スケルトンとサーバアプリケーションが使用するライブラリで静的に結合します。データベース管理システムのライブラリ以外で、サーバアプリケーションのリンク時に指定するライブラリを以降に示します。

3.4.4 スレッドモードのアプリケーションのコンパイルとリンク

[コンパイル時に指定するインクルードパス] Solaris32 Linux32

IDLコンパイルを実施したディレクトリ
TDのインストールディレクトリ/include
ODのインストールディレクトリ/include
EXTPのインストールディレクトリ/include

[リンク時に指定するライブラリ]

Solaris32

Forte Developer 6シリーズ、Sun ONE Studio 7シリーズ、Sun ONE Studio 8シリーズまたはSun Studio 9以降を使用する場合

ライブラリ名	格納場所	用途
libsocket.so	/usr/lib	ソケットライブラリ(必須)
libnsl.so	/usr/lib	TLIライブラリ(必須)
libthread.so	/usr/lib	スレッドライブラリ(必須)
libOM.so	ODのインストールディレクトリ/lib	ODランタイム(注)
libOMcncpp50.so	ODのインストールディレクトリ/lib	ODランタイム(注)
libextpapiskl.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libtdalcapi.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libtdapicpp50.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libOMcpp50.so	ODのインストールディレクトリ/lib	ODランタイム(必須)
Libextpapmcom50.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libextpapmbase50.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libextpapmlibTDNORM50.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)

Linux32

ライブラリ名	格納場所	用途
libnsl.so	/usr/lib	TLIライブラリ(必須)
libpthread.so	/usr/lib	スレッドライブラリ(必須)
libC.so	/usr/lib	Cライブラリ(必須)
libOM.so	ODのインストールディレクトリ/lib	ODランタイム(注)
libOMcncpp.so	ODのインストールディレクトリ/lib	ODランタイム(注)
libextpapiskl.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libtdalcapi.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libtdapicpp.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libOMcpp.so	ODのインストールディレクトリ/lib	ODランタイム(必須)
Libextpapmcom.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libextpapmbase.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libextpapmlibTDNORM.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)

(注)

中継用サーバアプリケーションを作成するとき、および、例外を使用するときに必要です。

サーバアプリケーションとスケルトンをコンパイル後、リンクする手順について例を以下に示します。

[コンパイル・リンク手順の例]

Solaris32

Forte Developer 6シリーズ、Sun ONE Studio 7シリーズ、Sun ONE Studio 8シリーズまたはSun Studio 9以降を使用する場合

```
%CC -c -D_REENTRANT -I/opt/FSUNod/include -I/opt/FSUNtd/include tdsample1_s.C
%CC -c -D_REENTRANT -I/opt/FSUNod/include -I/opt/FSUNtd/include -I/opt/FSUNextp/include
TD_TDSAMPLE1_INTF_skel_c++.C
```

```
%CC -c -D_REENTRANT -I/opt/FSUNod/include -I/opt/FSUNtd/include TD_tdsample1_c++.C
%CC -o tdsample1_s tdsample1_s.o TD_TDSAMPLE1_INTF_skel_c++.o TD_tdsample1_c++.o
 -lsocket -lnsl -lthread -L/opt/FSUNod/lib -lOMcpp50 -L/opt/FSUNtd/lib -ltdalcapi
 -ltdapicpp50 -L/opt/FSUNextp/lib -lextpapiskl -lextpapmlibTDNORM50 -lextpapmcom50
 -lextpapmbase50
%
```

注意

Forte Developer 6シリーズ、Sun ONE Studio 7シリーズ、Sun ONE Studio 8シリーズまたはSun Studio 9以降を使用する場合には、リンクオプションに"-lC"および"-lCrun"を指定しないでください。

3.4.5 プロセスモードのアプリケーションのコンパイルとリンク

[コンパイル時に指定するインクルードパス] Solaris32 Linux32

IDLコンパイルを実施したディレクトリ
TDのインストールディレクトリ/include
ODのインストールディレクトリ/include
EXTPのインストールディレクトリ/include

アプリケーションをプロセスモードで動作させる場合は、以下のオプションは指定しないでください。

Solaris32

```
-mt -D_REENTRANT
```

Linux32

```
-D_REENTRANT
```

[リンク時に指定するライブラリ]

Solaris32

Forte Developer 6シリーズ、Sun ONE Studio 7シリーズ、Sun ONE Studio 8シリーズまたはSun Studio 9以降を使用する場合を使用する場合

ライブラリ名	格納場所	用途
libsocket.so	/usr/lib	ソケットライブラリ(必須)
libnsl.so	/usr/lib	TLIライブラリ(必須)
libOM.so	ODのインストールディレクトリ/lib/nt	ODランタイム(注)
libOMcncpp50.so	ODのインストールディレクトリ/lib	ODランタイム(注)
libextpapiskl_nt.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libtdalcapi_nt.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libtdapicpp50_nt.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libOMcpp50.so	ODのインストールディレクトリ/lib	ODランタイム(必須)
libextpapmcom50_nt.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libextpapmbase50_nt.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libextpapmlibTDNORM50_nt.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)

Linux32

ライブラリ名	格納場所	用途
libnsl.so	/usr/lib	TLIライブラリ(必須)
libC.so	/usr/lib	Cライブラリ(必須)
libOM.so	ODのインストールディレクトリ/lib/nt	ODランタイム(注)
libOMcncpp.so	ODのインストールディレクトリ/lib	ODランタイム(注)
libextpapiskl_nt.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libtdalcapi_nt.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libtdapicpp_nt.so	TDのインストールディレクトリ/lib	TDランタイム(必須)
libOMcpp.so	ODのインストールディレクトリ/lib	ODランタイム(必須)
libextpapmcom_nt.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libextpapmbase_nt.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)
libextpapmlibTDNORM_nt.so	EXTPのインストールディレクトリ/lib	TDランタイム(必須)

(注)
中継用サーバアプリケーションを作成するとき、および、例外を使用するときに必要です。

[コンパイル・リンク手順の例]

Solaris32

Forte Developer 6シリーズ、Sun ONE Studio 7シリーズ、Sun ONE Studio 8シリーズまたはSun Studio 9以降を使用する場合を使用する場合

```
%CC -c -I/opt/FSUNod/include -I/opt/FSUNtd/include tdsample1_s.C
%CC -c -I/opt/FSUNod/include -I/opt/FSUNtd/include -I/opt/FSUNextp/include
TD_TDSAMPLE1_INTF_skel_c++.C
%CC -c -I/opt/FSUNod/include -I/opt/FSUNtd/include TD_tdsample1_c++.C
%CC -o tdsample1_s_nt tdsample1_s.o TD_TDSAMPLE1_INTF_skel_c++.o TD_tdsample1_c++.o
-lsocket -lnsl -L/opt/FSUNod/lib/nt -L/opt/FSUNod/lib -lOMcpp50 -L/opt/FSUNtd/lib
-ltdalcapi_nt -ltdapicpp50_nt -L/opt/FSUNextp/lib -lextpapiskl_nt
-lextpapmlibTDNORM50_nt -lextpapmcom50_nt -lextpapmbase50_nt
%
```

注意

Forte Developer 6シリーズ、Sun ONE Studio 7シリーズ、Sun ONE Studio 8シリーズまたはSun Studio 9以降を使用してアプリケーションを作成する場合、リンクオプションに"-lC"および"-lCrun"を指定しないでください。

作成したアプリケーションライブラリにスレッドライブラリが結合されていないことを確認してください。
アプリケーションのコンパイル/リンクの手順に誤りがあった場合や、libthread.soがリンクされている場合、ワークユニット起動/停止処理またはクライアントとの通信が無応答状態となる可能性があります。その場合、以下の対処を行ってください。

1. 当該ワークユニット配下で動作しているアプリケーションプロセスのプロセスIDを特定します。

```
% ps -ef | grep ワークユニット名
```

2. 当該ワークユニット配下で動作しているアプリケーションプロセスを強制停止します。

```
% kill -9 プロセスID
```

Linux32

```
%g++ -c -I/opt/FJSVod/include -I/opt/FJSVtd/include tdsample1_s.C
%g++ -c -I/opt/FJSVod/include -I/opt/FJSVtd/include -I/opt/FJSVextp/include
```

```
TD_TDSAMPLE1_INTF_skel_c++.C
%g++ -c -I/opt/FJSVod/include -I/opt/FJSVtd/include TD_tdsample1_c++.C
%g++ -o tdsample1_s_nt tdsample1_s.o TD_TDSAMPLE1_INTF_skel_c++.o TD_tdsample1_c++.o
-lnsl -lC -L/opt/FJSVod/lib/nt -L/opt/FJSVod/lib -lOMcpp
-L/opt/FJSVtd/lib -ltdalcapi_nt -ltdapicpp_nt -L/opt/FJSVextp/lib -lextpapiskl_nt
-lextpapmlibTDNORM_nt -lextpapmcom_nt -lextpapmbase_nt
%
```

注意

作成したアプリケーションライブラリにスレッドライブラリが結合されていないことを確認してください。
アプリケーションのコンパイル／リンクの手順に誤りがあった場合や、libpthread.soがリンクされている場合、ワークユニット起動／停止処理またはクライアントとの通信が無応答状態となる可能性があります。その場合、以下の対処を行ってください。

1. 当該ワークユニット配下で動作しているアプリケーションプロセスのプロセスIDを特定します。

```
% ps -ef | grep ワークユニット名
```

2. 当該ワークユニット配下で動作しているアプリケーションプロセスを強制停止します。

```
% kill -9 プロセスID
```

リンク時の注意事項

サーバアプリケーションのリンク時には、上記のライブラリのほかに、データベースや連携する他製品のライブラリなど、サーバアプリケーションで必要なライブラリがすべてリンクされていることを確認してください。また、リンクしたライブラリの格納先をワークユニット定義のアプリケーション使用ライブラリパス(Library for Applicationセクション)に設定してください。

Solaris32

サーバアプリケーションがC++言語の場合には、必ず"/opt/FSUNod/lib/nt"を指定しなければなりません。

Linux32

サーバアプリケーションがC++言語の場合には、必ず"/opt/FJSVod/lib/nt"を指定しなければなりません。

リンクが完全でない場合やワークユニット定義のアプリケーション使用ライブラリパスが完全でない場合は、ワークユニット起動処理において実行ファイルのオープン処理に失敗し、メッセージ(EXTP4640,EXTP4643,EXTP4645,EXTP4508)が出力されます。

以下に必要なワークユニット定義の設定例を示します。

ワークユニット定義の設定例

サーバアプリケーションの実行ファイルが"application1"、格納先が/application/bin、サーバアプリケーションにリンクされているライブラリの格納先が、それぞれ、"/product1/lib"、"/product2/lib"の場合の定義例を以下に示します。

Solaris32

```
[WORK UNIT]
Name: SAMPLEWU
Kind: ORB
[APM]
Name: TDNORM

[Control Option]
…
Path: /application/bin
Library for Application: /product1/lib
Library for Application: /product2/lib
Library for Application: /opt/FSUNod/lib/nt
…

[Application Program]
```

```
…
Executable File : application1
```

Linux32

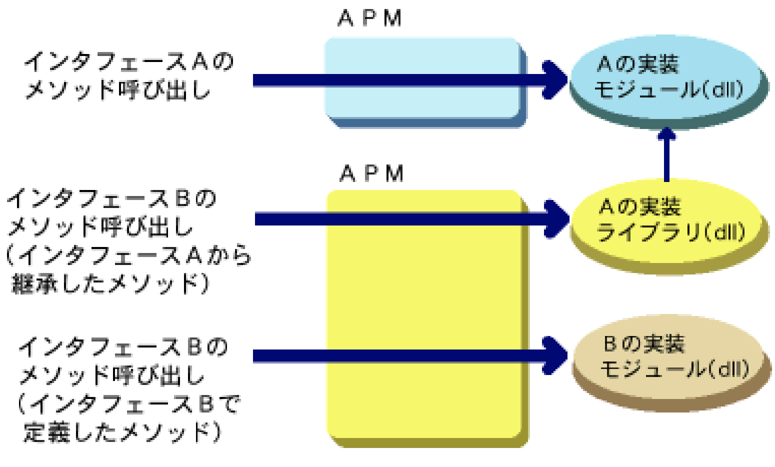
```
[WORK UNIT]
Name: SAMPLEWU
Kind: ORB
[APM]
Name: TDNORM

[Control Option]
…
Path: /application/bin
Library for Application: /product1/lib
Library for Application: /product2/lib
Library for Application: /opt/FJSVod/lib/nt
…

[Application Program]
…
Executable File : application1
```

サーバアプリケーションにリンクされているライブラリの格納先が複数存在する場合は、アプリケーション使用ライブラリパス(Library for Applicationセクション)を複数設定してください。
また、出口プログラムにリンクされているライブラリの格納先についても、同様にアプリケーション使用ライブラリパス(Library for Applicationセクション)に設定してください。

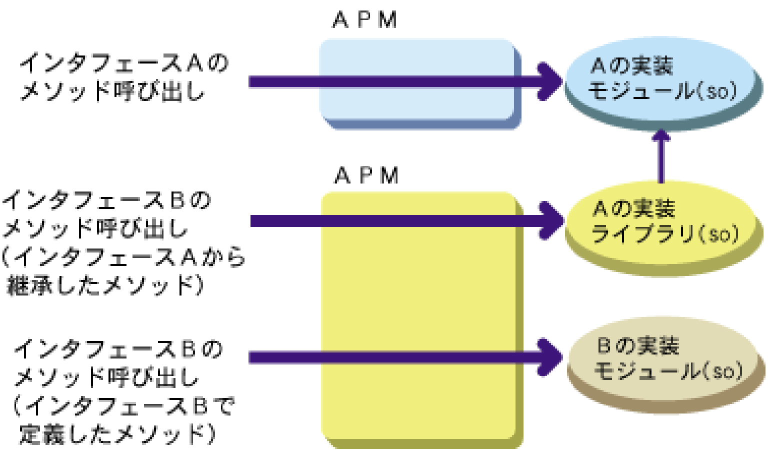
3.4.6 継承について

あるインタフェースで定義したオペレーションを、別のインタフェースに引き継ぐことを可能にするための機能です。継承を利用した場合、クライアントからは継承を意識せずに、継承したオペレーションを呼び出すことができます。
メソッドの呼び出しイメージについて以下に示します。

Windows32



Solaris32 Linux32



継承は、IDL定義ファイルに継承の指定を記述することによって使用することができます。継承先インタフェースは、継承元インタフェースのスコープ名をコロン(":")に続いて指定します。継承の指定例を以下に示します。
この例では、インタフェース“A”が継承元インタフェース、インタフェース“B”が継承先インタフェースとなります。

```
interface A {
        long op1(in long a);
};
interface B:X::A {
        long op2(in long b);
};
```

X：module名
A、B：interface名

継承先のIDL定義内に継承元のIDL定義をincludeする記述が必要です。include方法の例を以下に示します。

A.idl

```
module X{
 interface A{
   long op1();
 };
};
```

B.idl

```
#include "A.idl"
module Y{
 interface B:X::A{
   long op2();
 };
};
```

X::Aにop2を追加したY::Bを作成する方法

継承を使用する場合、モジュール作成時に継承元のインタフェースを実装しているライブラリをリンクします。継承を使用したアプリケーションの作成時に必要な注意点を、以下に示します。

継承元のインタフェースの状態

継承先のIDL定義をIDLコンパイルする場合、継承元は以下の状態であることが考えられます。ここでは、以下の状態の注意点を説明します。

- 継承元のインタフェースがインタフェースリポジトリに登録されていない場合
- 継承元のインタフェースがインタフェースリポジトリに登録されている場合

継承元のインタフェースがインタフェースリポジトリに登録されていない場合

tdcコマンド実行時に-Iオプションを指定してください。tdcコマンドの詳細については、“リファレンスマニュアル(コマンド編)”を参照してください。

継承元のインタフェースがインタフェースリポジトリに登録されている場合

継承先のIDL定義に継承元のIDL定義をincludeする記述を追加します。tdcコマンドは、-Iオプションと共に、-updateオプションを設定して実行してください。tdcコマンドの詳細については、“リファレンスマニュアル（コマンド編）”を参照してください。

継承元のライブラリの所在

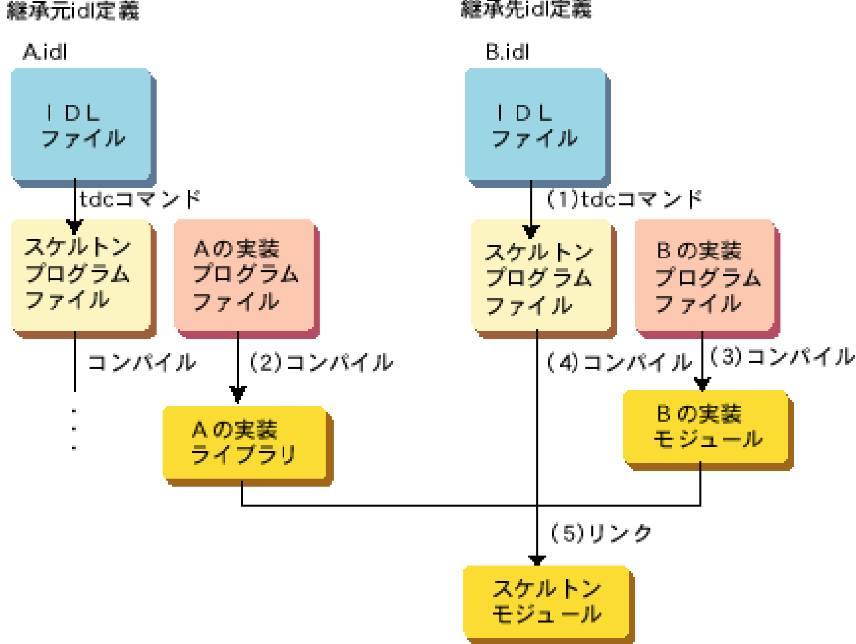
継承先のサーバアプリケーションをコンパイルする場合の注意点を説明します。

- 継承元インタフェースが実装されているライブラリのリンク
- 継承元インタフェースが実装されているライブラリのパスの設定

継承元インタフェースが実装されているライブラリのリンク

継承を使用する場合のサーバアプリケーション作成方法

(1) 継承先のIDL定義ファイル（B.idl）をtdcコマンドでコンパイルします。
(2) 継承元となるAの実装部は、スケルトンと別ライブラリとして作成します。
(3) 継承先となるBの実装部をコンパイルし、ライブラリを作成します。
(4) (1)で生成されたスケルトンをコンパイルします。
(5) (2)(3)(4)で生成されたライブラリをリンクします。



Windows32

ライブラリ

ダイナミックリンクライブラリ（.lib）

モジュール

実行モジュール(.exe)

Solaris32 Linux32

ライブラリ

ダイナミックリンクライブラリ（.so）

モジュール

実行モジュール

継承元インタフェースが実装されているライブラリのパスの設定

継承元インタフェースが実装されているライブラリのパスの設定は、ワークユニット定義で設定します。

Windows32

“Path for Application:”ステートメントに、継承元インタフェースへのパスを記述します。

Solaris32 Linux32

“Library for Application:”ステートメントに、継承元インタフェースへのパスを記述します。

3.5 ワークユニット定義の作成

ローカルトランザクション運用におけるワークユニット定義の概要について説明します。ワークユニット定義の詳細については、“付録A ワークユニット定義の記述形式”および“OLTPサーバ運用ガイド”を参照してください。

定義情報の設定

ワークユニット定義ファイルに記述する定義情報について、以下に示します。

ワークユニット名

ワークユニットを操作するためのワークユニット名を[WORK UNIT]セクションで設定します。ワークユニット名は、ワークユニット単位に1つ設定することができます。

APM名

APM名を[APM]セクションに設定します。無条件に“TDNORM” と指定してください。

制御オプション

アプリケーションが動作するためのカレントディレクトリやアプリケーションが格納されているライブラリパスなどの環境情報を[Control Option]セクションで設定します。

アプリケーション情報

ワークユニットで動作させるアプリケーション名やオブジェクト名などの情報を、[Application Program]セクションで設定します。
ワークユニットにアプリケーションを追加する場合、追加するアプリケーション単位に[Application Program]セクションに情報を記載して追加します。たとえば、ワークユニットに4つの別々なアプリケーションを動作させる場合には、4つの[Application Program]セクションを記載します。
ワークユニットからアプリケーションを削除する場合は、不要となるアプリケーションに対応する[Application Program]セクションを削除します。

なお、C++アプリケーションでは、非常駐運用およびマルチオブジェクト常駐運用を行うことはできません。

3.6 アプリケーションの登録

サーバアプリケーションをほかのアプリケーションからアクセス可能にするためには、目的のアプリケーションを識別するためのオブジェクトリファレンスを作成する必要があります。また、同時に作成したオブジェクトリファレンスをネーミングサービスに登録することによって、ほかのアプリケーションからのアクセスが可能になります。

トランザクションアプリケーションの場合、オブジェクトリファレンスの作成とネーミングサービスへの登録は、次の2つの方法で行うことができます。

- ワークユニット起動による自動登録
- OD_or_admコマンド(注)による手動登録

ワークユニット起動による自動登録の場合は、ワークユニットを起動するとワークユニットで指定された各アプリケーションごとにオブジェクトリファレンスが作成され、ネーミングサービスに登録されます。この場合、ワークユニットを停止すると、オブジェクトリファレンスは、自動的にネーミングサービスから削除され、無効となります。
OD_or_admコマンドによる手動登録の場合は、ワークユニットの起動前に、事前にOD_or_admコマンドによりオブジェクトリファレンスの作成とネーミングサービスへの登録を行います。この場合は、ワークユニットの起動、停止にかかわらず、オブジェクトリファレンスは有効です。

2つの方法とも、ネーミングサービスには以下の名前でオブジェクトリファレンスを登録します。

```
モジュール名::インタフェース名
```

また、インプリメンテーションリポジトリIDとしては、以下のIDを使用します。

Windows32 Solaris32

```
ワークユニット種別がORBの場合　　: FUJITSU-Interstage-TDLC
ワークユニット種別がWRAPPERの場合　: FUJITSU-Interstage-TDRC
```

Linux32

```
ワークユニット種別がORBの場合　　: FUJITSU-Interstage-TDLC
```

なお、2つの方法のどちらを使用するかは、ワークユニット定義で指定する必要があります。詳細は“OLTPサーバ運用ガイド”を参照してください。

注）ロードバランス機能を使用する場合は、odadministerlbコマンドを使用します。

注意

手動登録する場合、OD_or_admコマンドを実行する前に、Interstageが起動されている必要があります。
OD_or_admコマンドの詳細については、“リファレンスマニュアル(コマンド編)”を参照してください。

3.7 アプリケーションのテスト

作成したアプリケーションのテスト方法を、以下の内容で説明します。

Solaris32

- サーバアプリケーションのテスト方法

Windows32 Solaris32 Linux32

- クライアントアプリケーションのテスト方法
- スナップショットによるテスト支援
- 運用環境への移行
- 運用環境におけるテスト

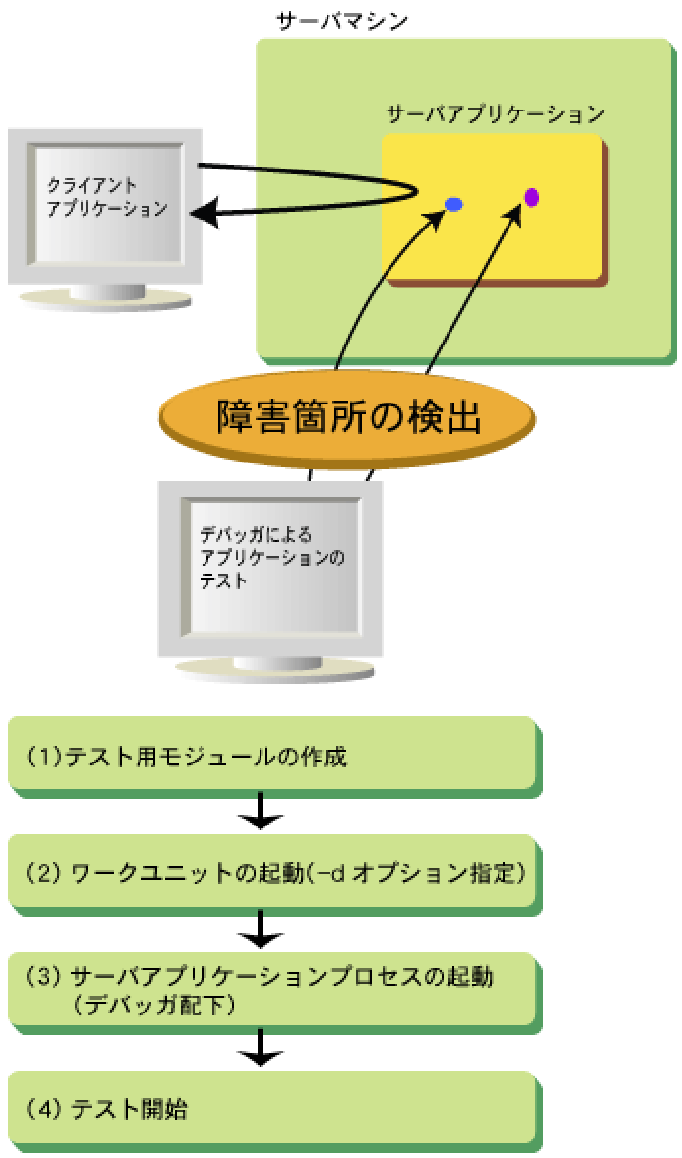
サーバアプリケーションのテスト方法 Solaris32

サーバアプリケーションのテストを行う場合、実際にクライアントアプリケーションと結合して行います。このとき、サーバアプリケーションをデバッガ配下で動作させることで、サーバアプリケーションが正しく作成されているか確認できます。

注意

サーバアプリケーションのテストは、Solarisの場合だけが可能です。

デバッガと連携するには、アプリケーションをdbx配下などで動かせるようにする必要があります。デバッガと連携するときの動作の概要と、サーバアプリケーションをデバッガ配下でテストする場合の手順を、以下に示します。



テスト用モジュールの作成

テストを行うサーバアプリケーションを、-gオプションおよび-xsオプションを指定してコンパイルします。コンパイル方法の詳細については、関連するマニュアルを参照してください。
テスト用モジュールのコンパイル例を以下に示します。

```
%CC -c -g -xs -D_REENTRANT -I/opt/FSUNod/include -I/opt/FSUNtd/include tdsample1_s.C
```

ワークユニットの起動

テストを行うサーバアプリケーションのワークユニット定義を作成し、tdstartwuコマンドでデバッグオプション-dを指定してワークユニットを起動します。-dオプションを指定してワークユニットを起動した場合、ワークユニットの動作環境まで作成し、サーバアプリケーションプロセスを起動せずにコマンドは復帰します。tdstartwuコマンドの詳細については"リファレンスマニュアル(コマンド編)"を参照してください。
デバッグ用のワークユニットの起動例を、以下に示します。

```
%tdstartwu -d TDSAMPLE1
```

サーバアプリケーションプロセスの起動

dbxコマンドなどをサーバアプリケーションを指定して起動することにより、サーバアプリケーションをそのデバッガ配下で起動します。サーバアプリケーションプロセスを起動する際のdbxコマンドの指定方法を以下に示します。

```
%dbx サーバアプリケーション
```

テストの実施

デバッガ配下でサーバアプリケーションを起動後、デバッガ画面でサーバアプリケーションの実行を行ってください。サーバアプリケーションの実行方法を以下に示します。

```
(dbx)run　業務システム名　ワークユニット名　オブジェクト名　T
```

業務システム名

デフォルトシステムの場合は、"td001"を指定します。拡張システムの場合は、システム名を設定します。

ワークユニット名

本サーバアプリケーションが動作するワークユニット名を指定します。

オブジェクト名

ワークユニット定義で指定した、本サーバアプリケーションのオブジェクト名を指定します。

オブジェクト名

“T”を指定します。

サーバアプリケーションの実行後、クライアントアプリケーションからサーバアプリケーションを呼び出し、処理を実行することで、サーバアプリケーションの動作状態をデバッガから確認することができます。これにより、アプリケーションを実行することができ、ステップ単位でデバッグすることができます。なお、C++プログラムのデバッグ方法の詳細については、関連するマニュアルを参照してください。

アプリケーションのテストを終了する場合は、以下の手順で行います。

- サーバアプリケーションのデバッグを終了し、クライアントアプリケーションからの要求待ちの状態にします。
- ワークユニットを停止します。

デバッガにより、サーバアプリケーションをデバッグする場合の注意事項について示します。

- サーバアプリケーションをデバッガ配下で起動しデバッグを行っているときは、再デバッグ（rerun）を行わないでください。再デバッグを行う場合は一度デバッガを停止し、その後再起動してください。
- サーバアプリケーションのデバッグ中にワークユニットを停止しないでください。必ずサーバアプリケーションが復帰し、要求待ちの状態になっている状態でワークユニットを停止してください。
- ワークユニット定義におけるプロセス多重度には必ず1を指定してください。
- 1つのワークユニットに対しては、デバッガは1つだけ起動するようにしてください。
- デバッガを終了する場合は、必ずデバッガによる実行状態が完了している状態で行ってください。
- 前出口プログラム、および後出口プログラムのデバッグ中は、ほかのワークユニットの起動処理または停止処理が保留されます。
- デバッガによるステップ実行中でも時間監視は行われますので、ワークユニット定義で時間監視を行わないか、またはデバッグに十分な時間を指定してください。
- デバッガを使用している場合は、tdstopwuコマンドで-cオプション、およびisstopコマンドで-c、-fおよび-s（Windows版）オプションは投入できません。
- ワークユニット定義のアプリケーション使用ライブラリパス、およびアプリケーションライブラリパスに指定されているパスを環境変数(LD_LIBRARY_PATH)に設定し、デバッガを起動してください。

クライアントアプリケーションのテスト方法

クライアントアプリケーションのテスト方法は、クライアントアプリケーションを動作させるオペレーティングシステムやミドルウェアによって異なります。お使いのオペレーティングシステムやミドルウェアごとに推奨される方法でテストを行ってください。

スナップショットによるテスト支援

スナップショットを使用して、クライアントからの要求に対する入出力情報をワークユニット単位に取得することにより、アプリケーションのデバッグを行うことができます。

詳細は、"第7章 スナップショット機能"を参照してください。

運用環境への移行

開発環境でテストした資材を運用環境へ移行するための作業手順と、運用環境でのテスト方法について説明します。

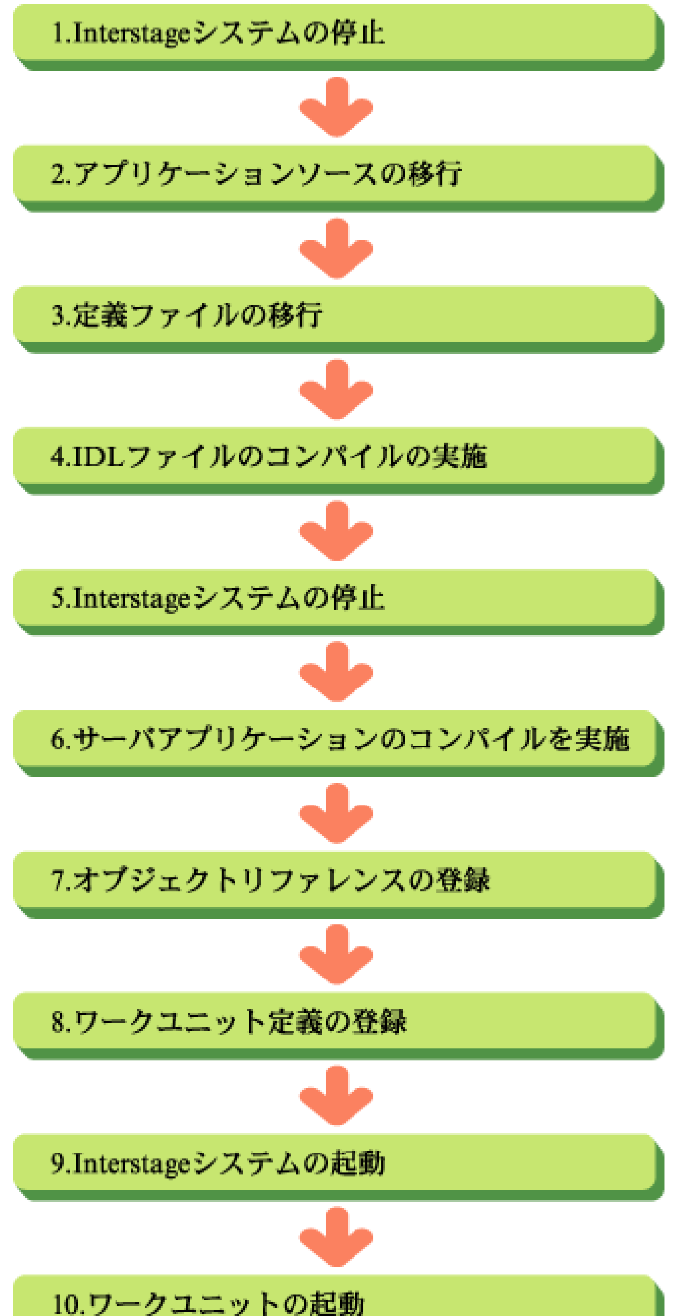
移行手順

クライアント資材の移行

クライアント資材は、サーバ資材を移行してサーバアプリケーションがコンパイルされるまでに移行してください。

サーバ資材の移行

サーバ資材を運用環境に移行する手順について、以下に示します。



1. 開発環境のInterstageシステムを停止します。
2. 開発環境からサーバアプリケーションのプログラムソースを運用環境に複写します。
3. 開発環境からIDLファイル、ワークユニット定義ファイルを運用環境に複写します。
4. 3.で複写したIDLファイルをもとに、tdcコマンドを実行し、スタブファイル、スケルトンファイルを作成します。
5. 運用環境のInterstageシステムの停止
6. 2.で複写したサーバアプリケーションプログラムソースと4.で作成したスケルトンファイルにより、サーバアプリケーションを作成します。
7. ワークユニット定義の"ネーミングサービスの登録形態"が"MANUAL"の場合、オブジェクトリファレンスを登録します。
8. 3.で複写したワークユニット定義を登録します。
9. 運用環境のInterstageシステムを起動します。
10. 運用環境のワークユニットを起動します。

運用環境におけるテスト

開発環境では各モジュールの単体テストを行いますが、運用環境ではシステム全体として以下に示すテストが必要です。

- システム負荷テスト
- 業務に沿った運用テスト
- 業務に則した性能テスト

それぞれのテスト方法について、以下に示します。

システム負荷テスト

システム負荷テストは業務を遂行するために、システム上のすべてのコンポーネントを含めたテストを実施します。システムの負荷をあげるためには、システムへのデータ入力の頻度(呼量と呼びます)をあげると、実施できます。たとえば、多数のCORBAクライアントからの入力の場合、CORBAクライアントを高速なマシン上で動作させると、呼量が増加し、負荷があげられます。

Windows32 Solaris32

また、Web連携の負荷をあげるときは、WebStoneなどをサーバに設定し、同様に呼量を増加させることができます。

業務に沿った運用テスト

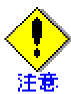
業務に沿った運用テストは、実際の業務を想定したテストを実施し、システムとして運用に問題がないかを確認します。したがって、システムで利用する製品および業務アプリケーションすべてを動作させます。たとえば、受注業務の運用テストを実施する場合、受注業務で利用する全製品とアプリケーションを動作させて、受注業務の開始/終了、データ入力とその処理などを実施します。

業務に則した性能テスト

性能テストは、業務運用中の性能について測定し、問題がないかを確認するテストです。

第4章 サーバアプリケーションの作成（COBOL）

ローカルトランザクション運用を行うためのサーバアプリケーションの作成方法について、説明します。



Linux版でCOBOLアプリケーションは使用できません。

4.1 サーバアプリケーションの開発

サーバアプリケーションを記述するために必要な以下の事項を説明します。

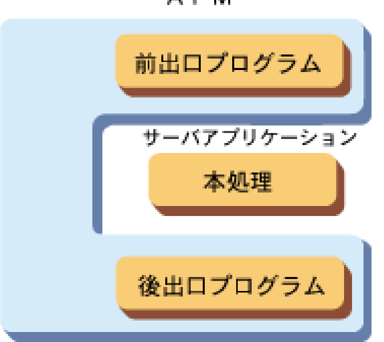
- サーバアプリケーションの構造
- サーバアプリケーションの入出力情報
- クライアントアプリケーションへの復帰値

Solaris32

- サーバアプリケーションプロセスの形態

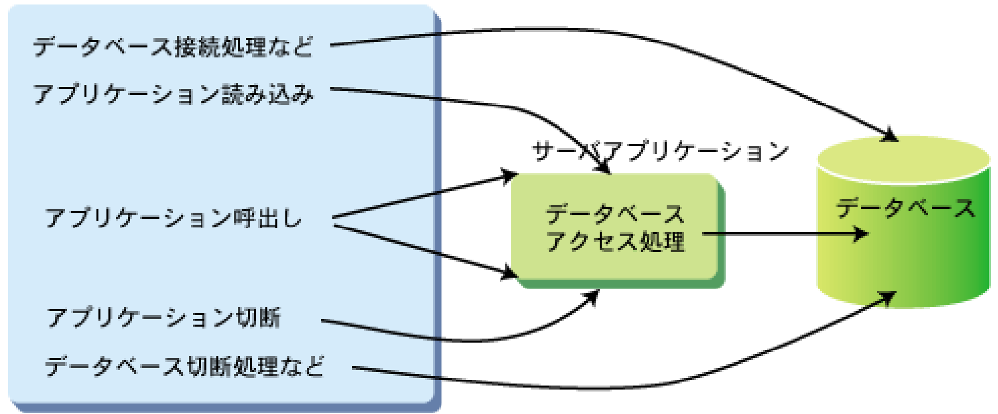
サーバアプリケーションの構造

一般のアプリケーションは、前処理、本処理、後処理という3つの構成で成り立っています。前処理として行う処理は、使用するデータベースの接続処理やデータベースのオープン処理などです。後処理として行う処理は、接続中データベースの切断処理やオープン中データベースのクローズ処理などです。
前処理および後処理で行うことは、使用するデータベース管理システムごとに決まっています。そのため、Interstageでは前出口プログラムおよび後出口プログラムの機能を提供します。この機能を実現するのがAPM(Application Program Manager)です。
APMを使用することにより、サーバアプリケーション開発者は、本処理だけを開発することで、サーバアプリケーションの作成が可能となります。
サーバアプリケーションの構造を以下に示します。

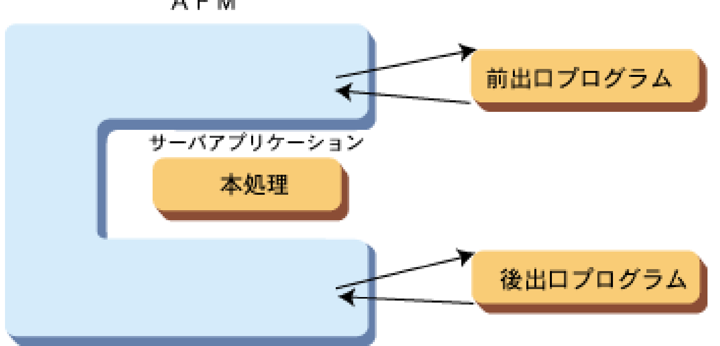


なお、データベース管理システムが提供するトランザクションを使用する場合には、サーバアプリケーション自身がデータベースとの接続や切断処理を行う必要があります。このような前処理および後処理を、出口プログラムとして本処理部分とは別に作成することができます。
この出口プログラムを作成することで、データ処理ごとに行っていたデータベースとの接続や切断処理を出口プログラムで行うことができるようになるため、効率の良い処理を構築することができます。

データベース管理システムが提供するトランザクションを使用する場合のサーバアプリケーションの構造を以下に示します。



APMとサーバアプリケーションの関係は、主プログラムと副プログラムという関係となります。また、サーバアプリケーションはAPMと動的結合により実行されます。APMとサーバアプリケーションの関係について以下に示します。



サーバアプリケーションと、一般のアプリケーションとの違いを以下に示します。

- 副プログラムとして作成する必要があります。
- スレッドに関する命令は記述できません。
- 実行プログラムは、動的リンクライブラリの形式で作成する必要があります。
- STOP RUN文を記述することはできません。
- 関数名、および変数名の長さは30文字以内です。
- アンダースコア"_"は、ハイフン"-"に置き換える必要があります。

注意

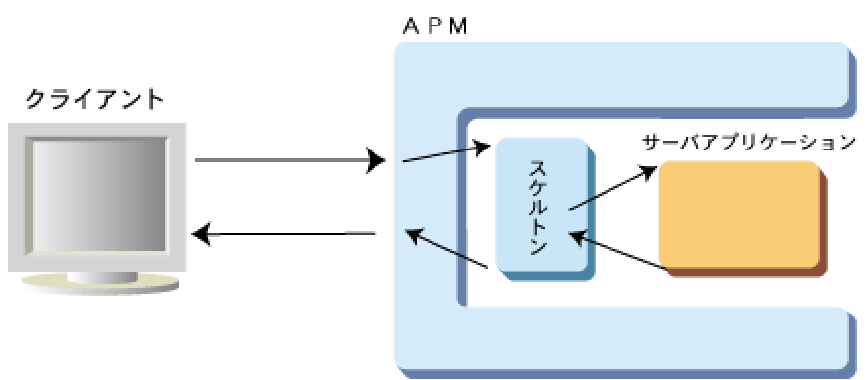
前出口プログラムは、ワークユニット起動時にアプリケーションプロセス(APM)単位で実行され、後出口プログラムは、ワークユニット停止時にアプリケーションプロセス(APM)単位で実行されます。ただし、ワークユニット強制停止時およびワークユニット異常終了時は、後出口プログラムは実行されません。
このとき、前出口プログラムおよび後出口プログラムの復帰値(PROGRAM-STATUS)が0の場合は正常、0以外の場合は異常とします。復帰値が0以外の場合、ワークユニット起動時では、ワークユニットの起動失敗となり、ワークユニット停止時では、警告メッセージを出力し、ワークユニット停止処理は正常に終了します。
以下の処理はInterstageが行うため、サーバアプリケーションで考慮する必要はありません。

- CORBAの初期化メソッドの呼び出し
- メソッドの登録
- サーバの活性化/非活性化

ただし、サーバアプリケーションが例外を使用する場合は、CORBAの初期化メソッドの呼び出しが必要です。

サーバアプリケーションの入出力情報

サーバアプリケーションは、クライアントからの呼び出し時に、スケルトンを経由して、IDLで定義した入力インタフェースで呼び出されます。また、サーバアプリケーションの処理が完了すると、スケルトンを経由してIDLで定義した出力インタフェースがクライアントに返却されます。
サーバアプリケーションの動作概要を以下に示します。



サーバアプリケーションの入出力インタフェースはCORBAで規定されています。CORBAのデータ型と言語とのマッピングを以下に示します。

CORBAデータ型			型宣言	COBOL表記
基本データ型	整数型	・long long ・long ・unsigned long ・short ・unsigned short	・CORBA-LONG-LONG ・CORBA-LONG ・CORBA-UNSIGNED-LONG ・CORBA-SHORT ・CORBA-UNSIGNED-SHORT	・PIC S9(18) COMP-5 ・PIC S9(9) COMP-5 ・PIC 9(9) COMP-5 ・PIC S9(4) COMP-5 ・PIC 9(4) COMP-5
	浮動小数点型	・float ・double	・CORBA-FLOAT ・CORBA-DOUBLE	・COMP-1 ・COMP-2
	文字型	・char ・wchar	・CORBA-CHAR ・CORBA-WCHAR	・PIC X(1) ・PIC N(1)
	オクテット	・octet	・CORBA-OCTET	・PIC 9(4) COMP-5
	ブーリアン	・boolean	・CORBA-BOOLEAN	・PIC 9(9) COMP-5
	文字列型	・string ・wstring	―――――――	・PIC X(n) ・PIC N(n)
構造データ型	構造体	・struct	―――――――	・集団項目
	配列	・array	―――――――	・OCCURS句
	シーケンス	・sequence		・OCCURS句

CORBA-STRING型およびCORBA-WSTRING型のデータを入力とする場合は、COBOLプログラムで扱えるように変換処理を行う必要があります。変換処理を行うには、TDSTRINGGETまたはTDWSTRINGGETを呼び出します。この変換処理はTDSTRINGSETまたはTDWSTRINGSETを呼び出して行います。TDSTRINGGET、TDWSTRINGGET、TDSTRINGSETおよびTDWSTRINGSETの詳細については、"リファレンスマニュアル(API編)"を参照してください。
また、サーバアプリケーションからスケルトンに、outパラメタ、またはinoutパラメタの文字列型のデータを受け渡す場合、TDSTRINGALLOC または TDWSTRINGALLOC により領域を獲得する必要があります。TDSTRINGALLOC および TDWSTRINGALLOCの詳細については、"リファレンスマニュアル(API編)"を参照してください。

なお、サーバアプリケーションでは、コンテキスト情報を使用することはできません。サーバアプリケーションに対しては、第1引数と最後の1つ前の引数でシステムパラメタ、第2引数以降IDLで定義したパラメタおよび最後の引数で復帰値というインタフェースでデータの入出力を行います。これらのインタフェースのパラメタは、COBOLプログラムのLINKAGE SECTIONで指定します。LINKAGE SECTIONにおける記述形式については、COBOLのマニュアルを参照してください。

また、IDL定義にoutパラメタを定義している場合、サーバアプリケーションが処理を終了する際には、必ずoutパラメタに復帰データを設定しなければなりません。outパラメタに復帰データが設定されていない場合、サーバアプリケーションプロセスが異常終了することがあります。

クライアントアプリケーションへの復帰値

クライアントアプリケーションへの復帰値は、Interstageにより規定されています。

復帰値	意味	アプリケーションの指定可否
0～10000	サーバアプリケーション任意	可
10001	システム異常検出(メモリ不足等)	不可
10002	システムで異常検出	不可
10003	AIM連携のホスト間通信処理で異常検出	不可
10004	以下のいずれかの異常を検出 ― サーバアプリケーションでの異常または、 ― オペレーション名不当	不可
10005	AIM連携のセション継続機能において異常検出	不可
10006	アクセス権がないユーザからの要求	不可
10007	プロセスバインド機能において異常検出(注)	不可
10008	最大キューイング数に達した	不可

(注)IPCOMによる負荷分散を行っている環境でプロセスバインド機能を使用した場合にも復帰します。

復帰値0～10000

クライアントアプリケーションとサーバアプリケーションで任意に使用することができます。Interstageはこれらの復帰値を正常とみなします。

なお、サーバアプリケーションの本処理からの復帰値は、プログラムの最後の引数に設定してください。

復帰値10001～10008

Interstageがクライアントに通知する復帰値であり、システムで異常が検出されたことを意味します。クライアントに復帰値10001～10008が通知された場合は、同時にその異常の詳細がエラーログに記録されます。

また、復帰値が10004で、エラーログに詳細が出力されていない場合は、クライアントから呼び出したオペレーションが、サーバのスケルトンに定義されていない(上記表の「オペレーション名が不当」に該当する)可能性があります。クライアントのスタブとサーバのスケルトンが同一のIDL定義から生成されているか、不一致がないかを確認してください。

なお、エラーログは、以下のファイルに採取されます。エラーログの内容については“メッセージ集”の“コンポーネントトランザクションサービスが出力するログメッセージ”を参照してください。

Windows32

INTERSTAGEインストールフォルダ¥td¥trc¥lorb¥errlog0
(トランザクションアプリケーションの場合)

INTERSTAGEインストールフォルダ¥td¥trc¥rorb¥errlog0
(AIM連携の場合)

Solaris32

/var/opt/FSUNtd/trc/lorb/errlog0
(トランザクションアプリケーションの場合)

/var/opt/FSUNtd/trc/rorb/errlog0
(AIM連携の場合)

注意

- クライアントアプリケーションでは、復帰値を参照する前に例外を判定する必要があります。例外が発生している場合は、復帰値は不定となります。クライアントアプリケーションでの例外処理については、“アプリケーション作成ガイド(CORBAサービス編)”を参照してください。
- COBOLでは復帰値としてPROGRAM-STATUSが復帰します。サーバアプリケーション終了時にPROGRAM-STATUSに値を設定していないと、クライアント復帰する値は不定となります。

サーバアプリケーションプロセスの形態 Solaris32

サーバアプリケーションプロセスには以下の2つの形態があります。

スレッドモード

サーバアプリケーションプロセスがマルチスレッドで動作する形態です。ただし、サーバアプリケーションはプライマリスレッド上でのみ動作し、マルチスレッドでは動作しません。スレッドモードではInterstageの制御用スレッドが複数起動します。

通常はスレッドモードを使用してください。

プロセスモード

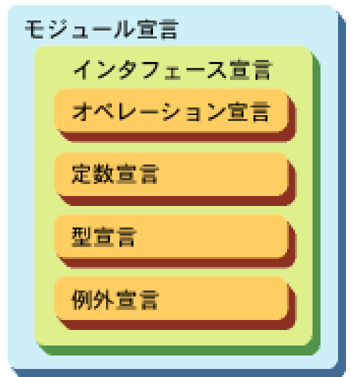
サーバアプリケーションプロセスがシングルスレッドで動作する形態です。本モードは、サーバアプリケーションより呼び出される他製品のライブラリが、マルチスレッドで動作するプロセス上で呼び出すことができない場合にのみ使用してください。

4.2 IDLファイルの作成

ローカルトランザクション運用におけるIDLファイルの文法の概要について説明します。IDLファイルの文法の詳細については、“アプリケーション作成ガイド（CORBAサービス編）”を参照してください。

IDLファイルの記述形式

ローカルトランザクション運用におけるIDLファイルの記述形式を以下に示します。



モジュール宣言

モジュール宣言では、IDL定義のオペレーション名や型名などが、ほかのIDL定義と重複しないように、オブジェクトのグループ化を定義します。
モジュール宣言の書式を以下に示します。

```
module モジュール名  {
       interface インタフェース名m {
           ・・
      } ;
       interface インタフェース名n {
           ・・
       } ;
 } ;
```

モジュール宣言の中に新たなモジュール宣言をすることもできます。これをモジュール宣言の入れ子と呼びます。モジュール宣言のmodule A宣言内にmodule B宣言をする例を以下に示します。

```
module A  {
  interface a1 {
      ・・
  } ;
  module B  {
     interface b1 {
       ・・
     } ;
  } ;
} ;
```

インタフェース宣言

インタフェース宣言では、アプリケーションへの入力と出力を規定するためのインタフェースを定義します。
インタフェース宣言の書式を以下に示します。

```
interface  インタフェース名[:継承するインタフェース名]  {
      オペレーション宣言 ;
      定数宣言 ;
      型宣言 ;
      例外宣言 ;

} ;
```

※ [ ]内は省略可能です。

インタフェース宣言は、モジュール宣言内に複数個記述できます。トランザクション運用では、インタフェース宣言を記述するときは、モジュール宣言を必ず記述する必要があります。

オペレーション宣言

オペレーション宣言では、アプリケーションに対応したオペレーション名、復帰値の型、パラメタのデータ型を定義します。
オペレーション宣言の書式を以下に示します。

```
復帰値のデータ型  オペレーション名 (
  [ パラメタタイプ  データ型  パラメタ名[ , . . ]]
)  [  raises  (例外構造体名[ , . .]) ] ;
```

※ [ ]内は省略可能です。

オペレーション宣言は、インタフェース宣言内に複数個記述できます。

定数宣言

インタフェースの定義内で使用する定数を定義します。
定数宣言の書式を以下に示します。

```
const データ型 定数名 = 定数式;
```

データ型と型宣言

トランザクション運用で使用できるデータ型とその型宣言について示します。

基本データ型の宣言

データ型宣言では、インタフェース内で使用されるデータ型を定義できます。基本データ型のうち、整数型、浮動小数点型、文字型、オクテット型、ブーリアン型については、型宣言にtypedefを利用することもできます。

文字列型を除く基本データ型宣言の書式を以下に示します。

```
typedef long long       データ型名 ;
typedef long            データ型名 ;
typedef short           データ型名 ;
typedef unsigned long   データ型名 ;
typedef unsigned short  データ型名 ;
typedef float           データ型名 ;
typedef double          データ型名 ;
typedef char            データ型名 ;
typedef wchar           データ型名 ;
typedef octet           データ型名 ;
typedef boolean         データ型名 ;
typedef wchar           データ型名 ;
long long               データ名 ;
long                    データ名 ;
short                   データ名 ;
unsigned long           データ名 ;
unsigned short          データ名 ;
float                   データ名 ;
double                  データ名 ;
char                    データ名 ;
wchar                   データ名 ;
octet                   データ名 ;
boolean                 データ名 ;
```

文字列型の型宣言の書式を以下に示します。

```
typedef string<サイズ>    データ型名;
typedef string            データ型名;
typedef wstring<サイズ>   データ型名;
typedef wstring           データ型名;

string<サイズ>            データ名 ;
string                    データ名 ;
wstring<サイズ>           データ名 ;
wstring                   データ名 ;
```

構造体

構造体の宣言の書式を以下に示します。

```
struct  データ名 {
         構造体メンバの宣言
};
```

構造体メンバの宣言の書式を以下に示します。

```
基本データ型 メンバ名 ;
```

構造体メンバは1つ以上必要であり、空の構造体は定義できません。
構造体の記述例を以下に示します。

```
module A {
        struct S {
                 string  name;
                 short   number;
                 long    value;
        };
};
```

配列

配列宣言では1次元固定長の配列を定義します。配列宣言の書式を以下に示します。

```
typedef データ型 識別子 [配列サイズ];
```

配列の次元は1階層までに制限されています。配列サイズは正の整数定数式で指定します。配列サイズはコンパイル時に固定されます。
配列宣言の例を以下に示します。

```
typedef long A[5];
```

シーケンス型

IDL言語でシーケンス型sequenceを指定した場合、以下の構造体でデータを宣言します。

```
struct
 {
        CORBA_unsigned_long     _maximum;       /* シーケンスの最大長 */
        CORBA_unsigned_long     _length;        /* シーケンスの長さ   */
        CORBA_Type              *_buffer;       /* シーケンスのデータ */
 };
```

_maximumはシーケンスの最大長、_lengthはシーケンスの長さ、_bufferはシーケンスのデータを指します。シーケンス構造体の領域を獲得するための関数(モジュール名、インタフェース名、構造体およびALLOCを"-"でつないだ名前の関数(以降XX-ALLOC関数と呼びます))がTDコンパイラで生成されます。
また、シーケンスのデータを獲得するための関数(モジュール名、インタフェース名、構造体名およびALLOCBUFを"-"でつないだ関数(以降XX-ALLOCBUF関数と呼びます))がTDコンパイラで生成されます。
シーケンス型の宣言の書式を以下に示します。

```
typedef sequence<データ型名 > データ名;
```

または

```
typedef sequence<データ型名 , シーケンス要素数 > データ名;
```

シーケンス型の要素として指定するデータ型は基本データ型だけに制限されています。
シーケンス型の記述例を以下に示します。

```
typedef sequence<long > Q;
```

例外宣言

例外宣言では、オペレーション実行中に例外が発生したときに例外情報を受け渡すための例外構造体名を定義します。オペレーション宣言のraises式を定義する場合は、この例外構造体名を指定します。
例外宣言の書式を以下に示します。

```
exception  例外構造体名 {
        データ型    メンバ名 ;       //構造体メンバを宣言
                    :
} ;
```

構造体メンバを複数定義する場合は、各メンバをセミコロン(“;”)で区切ります。メンバとして定義できるデータ型を以下に示します。

- 基本データ型
- 配列(配列要素のデータ型は基本データ型のみ、次元数は1次元まで)

また、例外にはシステムでの異常終了を通知するシステム例外とサーバアプリケーションでの異常終了を通知するユーザ例外がありますが、トランザクションアプリケーションではシステム例外は使用できません。
トランザクションアプリケーションでは、CORBAの初期化メソッドの呼び出しは必要ありませんが、例外を使用する場合、CORBAの初期化メソッドの呼び出しが必要となります。

例外の使用方法については、“アプリケーション作成ガイド（CORBAサービス編）”の各言語ごとのアプリケーションの例外処理を参照してください。

注意

コンポーネントトランザクションサービスにおける利用範囲

トランザクション運用で使用するIDLファイルの文法は、OMGで規定しているIDLの文法に準拠していますが、以下に示す項目が制限されています。

- モジュール宣言などにおいて、次の宣言が利用できません。
  - 属性宣言
  - コンテキスト
- 以下のデータ型は使用できません。
  - 浮動小数点型のlong double
  - 基本データ型の列挙型（enum）とany型
  - 共用体型
- オペレーション宣言において、利用できる型は以下に限られています。
  - long
  - oneway void

データ型の使用方法

各データ型の使用方法については、“アプリケーション作成ガイド（CORBAサービス編）”の各言語ごとのデータ型に対するマッピングを参照してください。
なお、クライアントアプリケーションにVisual Basicを使用し、サーバアプリケーションにOLE CORBA-ゲートウェイ経由で文字列型のデータを通信する場合には、IDL定義の記述に注意が必要です。“アプリケーション作成ガイド（CORBAサービス編）”のCOM/CORBA連携プログラミングを参照してください。

CORBAアプリケーションとトランザクションアプリケーションでは、使用するAPIが以下に示す点で異なります。

```
CORBA-STRING-ALLOC           ==> TDSTRINGALLOC
CORBA-FREE                   ==> TDFREE
CORBA-STRING-SET             ==> TDSTRINGSET
CORBA-STRING-GET             ==> TDSTRINGGET
CORBA-WSTRING-ALLOC          ==> TDWSTRINGALLOC
CORBA-WSTRING-SET            ==> TDWSTRINGSET
CORBA-WSTRING-GET            ==> TDWSTRINGGET

CORBA-SEQUENCE-ELEMENT-GET  ==> TDSEQUENCEELEMENTGET
CORBA-SEQUENCE-ELEMENT-SET  ==> TDSEQUENCEELEMENTSET
```

また、トランザクションアプリケーションでは、リリースフラグの参照APIおよびリリースフラグの設定APIが用意されていません。データの扱いは、CORBA-FALSE-VALUEが設定されているものとして扱ってください。

オペレーションの使用方法

トランザクションアプリケーションでは、オペレーションの型として、longおよびoneway voidのみ用意しています。

oneway voidの場合、以下の注意が必要となります。

- オペレーションを実装するプログラムは、PROCEDURE DIVISION USINGに復帰情報を記述しません。
- クライアントアプリケーションへの復帰は、サーバアプリケーションの状態にかかわらず、サーバアプリケーションへデータを送信し、送信処理が成功した時点で正常復帰します。
- outパラメタ、inoutパラメタは使用できません。

- ユーザ例外を使用できません。
- グローバルトランザクション連携機能は使用できません。

4.3 サーバアプリケーションのソースの作成

サーバアプリケーションの処理を、オペレーティングシステムに添付されているエディタなどを使用して記述します。
ローカルトランザクション運用の場合は、リソースマネージャが提供するトランザクション命令と、リソースマネージャを使用するために必要な、データベースの結合処理や切り離し文などを記述してください。記述するトランザクション命令については、使用するリソースマネージャのマニュアルを参照してください。
セション情報管理機能を使用して、セション型の業務システムを構築する場合は、サーバ側のオブジェクトがセション情報管理オブジェクトへアクセスする必要があるため、CORBAクライアントの初期化が必要です。
セション情報管理機能に使用するAPIについては、"リファレンスマニュアル（API編）"を参照してください。
IDL定義とサーバアプリケーション名の関係を以下に示します。

```
モジュール名-インタフェース名-オペレーション名
```

注意

- サーバアプリケーションにおいて、引数に指定するオブジェクト情報は設定されていません。
- セション情報管理機能を使用する場合は、"5.3 セション情報管理機能を使用したトランザクションアプリケーションの作成"を参照してください。
- トランザクションアプリケーションにおいて、イベントサービスを利用する場合、使用可能なイベントサービスの機能は以下になります。
  - Pushモデルのサプライヤ

COBOL登録集からの複写

IDL定義に定数宣言を記述した場合、IDLコンパイルにより以下のCOBOL登録集ファイルが出力されます。本ファイルが出力された場合は、COPY文を使用して登録集原文をサーバアプリケーションに複写してください。

```
TD-オブジェクト名-H.cbl
```

出口プログラムの作成

出口プログラムは以下の形式で作成してください。なお、ここでの出口プログラムとは、前出口プログラム、後出口プログラム、および、プロセスバインド機能を使用する場合の異常出口プログラムを指します。

引数	なし
復帰値（PROGRAM-STATUS）	正常終了：0 異常終了：0以外

復帰値が0以外の場合は、出口プログラム異常終了とみなし、出口プログラム種別ごとに以下の動作となります。

- 前出口プログラムの場合、ワークユニットの起動に失敗します。
- 後出口プログラムの場合、警告メッセージを出力し、ワークユニットの停止処理は成功します。
- プロセスバインド機能の異常出口の場合、アプリケーションプロセスが異常終了します。その後、アプリケーション異常時の自動再起動が設定されている場合は、アプリケーションプロセスが自動再起動され、自動再起動が設定されていない場合は、ワークユニットが異常終了します。

4.4 ソースのコンパイル・リンク

4.4.1 IDLファイルのコンパイル

IDLファイルをコンパイルすることにより、クライアント、サーバそれぞれのアプリケーションの言語に合わせたスタブファイルとスケルトンファイルが作成されます。IDLファイルのコンパイルには、tdcコマンドを使用します。tdcコマンドのオプションは、使用する言語により異なります。
tdcコマンドの詳細については、"リファレンスマニュアル(コマンド編)"を参照してください。tdcコマンドを使用した例を以下に示します。

Windows32

```
>tdc tdsample1.idl
```

Solaris32

```
%OD_HOME=/opt/FSUNod
%export OD_HOME
%TD_HOME=/opt/FSUNtd
%export TD_HOME
%tdc tdsample1.idl
```

注意

トランザクションアプリケーションでは、インタフェースリポジトリにインタフェース情報を登録する必要があります。インタフェースリポジトリに登録したインタフェース情報とスタブに埋め込まれるインタフェース情報は必ず一致している必要があります。IDLコンパイラでインタフェース情報を生成する際には、インタフェース情報チェック機能を使用することを推奨します。インタフェース情報チェック機能についての詳細は、"OLTPサーバ運用ガイド"の"インタフェース情報チェック機能を使用した運用"を参照してください。

アプリケーション間連携時に、以下の条件を満たす場合、コンパイル時に関数の二重定義エラーが発生します。

- 1つのクライアントアプリケーションから、複数のサーバアプリケーションを呼び出し可能とする。
- 各々のサーバアプリケーションで使用するIDL定義に同一種別のデータ型(基本データ型は除く)が宣言されている。

この場合は出力された関数名を変更するか、IDLcコマンド(-lcまたは-lsオプションを指定)を使用してスタブファイルを生成してください。IDLcコマンドの詳細については、"リファレンスマニュアル(コマンド編)"を参照してください。

4.4.2 スタブとクライアントアプリケーションのソースとのコンパイル・リンク

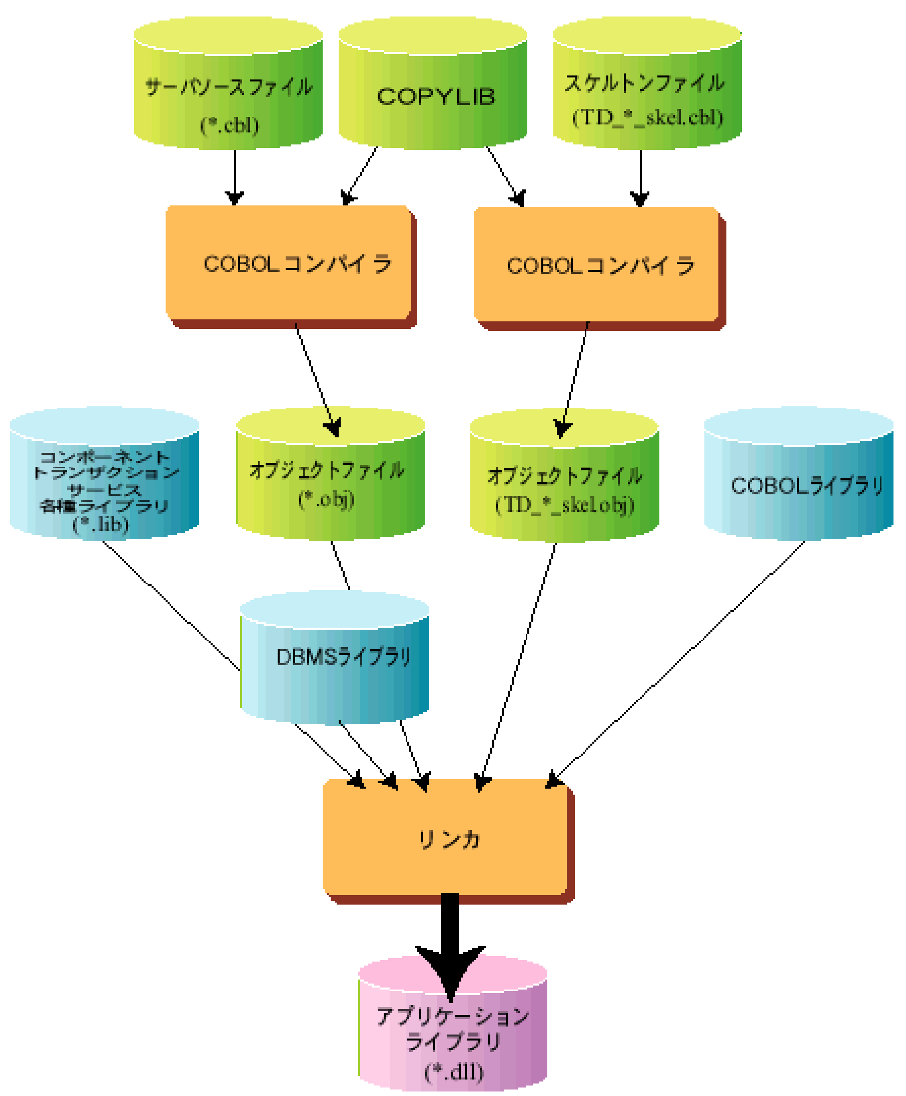
記述したクライアントアプリケーションのソースとスタブをコンパイルします。コンパイル方法の詳細については、クライアントアプリケーションを動作させるオペレーティングシステムやミドルウェアのマニュアルを参照してください。

4.4.3 スケルトンとサーバアプリケーションのソースとのコンパイル・リンク

サーバアプリケーションのソースとスケルトンのコンパイル方法、およびリンク方法について説明します。

COBOLでアプリケーションを作成した場合について、コンパイルとリンクの手順の流れを以下に示します。

Windows32



サーバアプリケーションは、APMと動的に結合します。このため、サーバアプリケーションは共用ライブラリとし、動的結合を可能とする構造(ダイナミックリンクライブラリ)としなければならず、動的リンク構造の実行可能プログラムとします。動的プログラム構造とすることはできません。この共用ライブラリには、サーバアプリケーションが使用するライブラリを結合してください。データベース管理システムのライブラリ以外で、サーバアプリケーションのリンク時に指定するライブラリを以下に示します。

[リンク時に指定するライブラリ]

ライブラリ名	格納場所	用途
f3fmalcapi.lib	INTERSTAGEインストールフォルダ¥td¥lib	コンポーネントトランザクションサービスランタイム(必須)
libextpapiskl.lib	INTERSTAGEインストールフォルダ¥extp¥lib	コンポーネントトランザクションサービスランタイム(必須)
odcobcbl.lib	INTERSTAGEインストールフォルダ¥odwin¥lib	CORBAサービスランタイム(注)

(注):
中継用サーバアプリケーションを作成するとき、および、例外を使用するときに必要です。

サーバアプリケーションとスケルトンをコンパイル後、リンクする場合のオプションの例を以下に示します。

[コンパイルオプションの例]

以下の表に示すコンパイルオプションを設定してください。

項目	設定値
登録集名	CORBA="INTERSTAGEインストールフォルダ¥odwin¥include¥cobol"
SSOUT	任意の文字列(注)

(注)
サーバアプリケーションでDISPLAY文を使用している場合に必要です。また、ワークユニット定義の環境変数ステートメントに、“任意の文字列”を環境変数として、DISPLAY文を出力するファイル名を指定します。指定されたファイルは、ワークユニット定義のカレントフォルダステートメントで指定されたフォルダ配下の“ワークユニット名¥プロセスID”フォルダ配下に出力されます。以下に設定方法を示します。

Environment Variable: 任意の文字列=ファイル名

リンク時の注意事項

サーバアプリケーションのリンク時には、上記のライブラリの他に、データベースや連携する他製品のライブラリなど、サーバアプリケーションで必要なライブラリがすべてリンクされていることを確認してください。また、リンクしたライブラリの格納先をワークユニット定義のアプリケーション使用パス(Path for Applicationセクション)に設定してください。
リンクが完全でない場合やワークユニット定義のアプリケーション使用パスが完全でない場合は、ワークユニット起動処理において実行ファイルのオープン処理に失敗し、メッセージ(EXTP4640,EXTP4643,EXTP4645,EXTP4508)が出力されます。

以下に必要なワークユニット定義の設定例を示します。

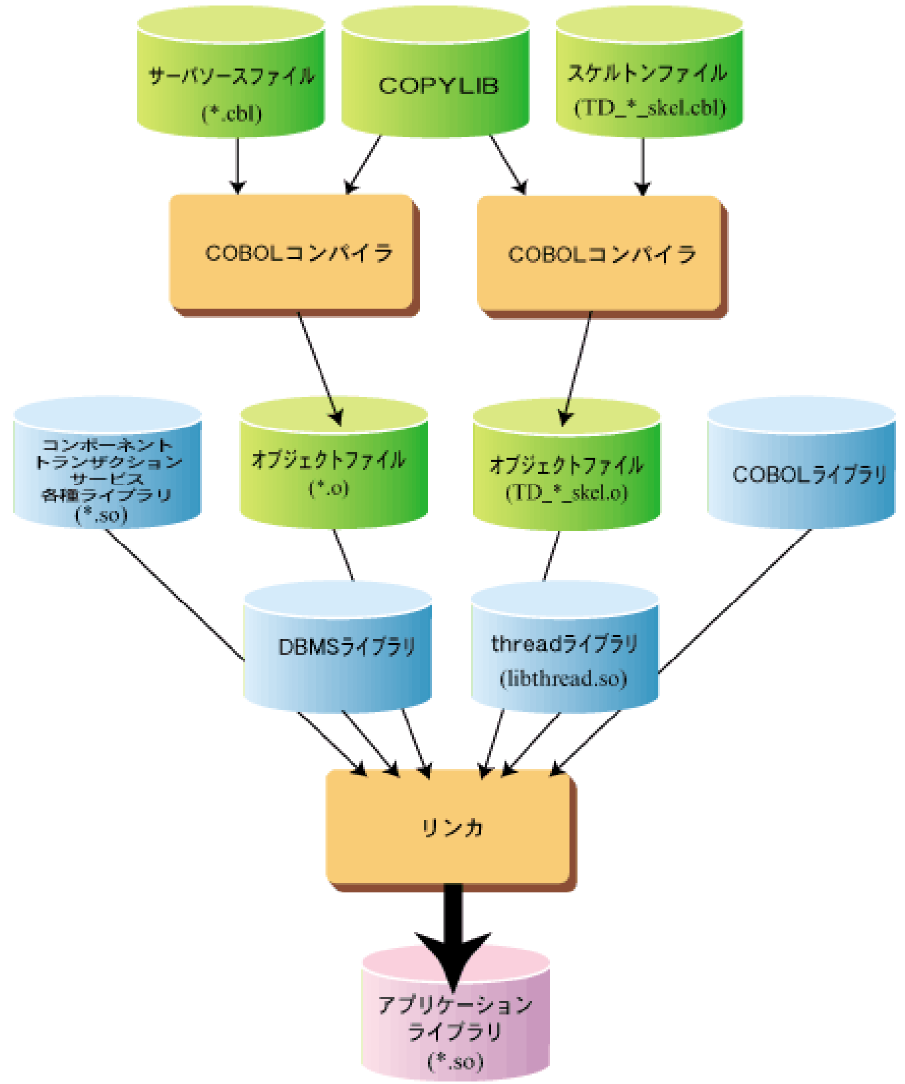
ワークユニット定義の設定例

サーバアプリケーションの実行ファイルが“application1.dll”、格納先がc:¥application¥bin、サーバアプリケーションにリンクされているライブラリの格納先が、それぞれ、“d:¥product1¥bin”、“e:¥product2¥bin”の場合の定義例を以下に示します。

```
[WORK UNIT]
Name: SAMPLEWU
Kind: ORB
[APM]
Name: TDNORM

[Control Option]
…
Path: c:¥application¥bin
Path for Application: d:¥product1¥bin
Path for Application: e:¥product2¥bin
…

[Application Program]
…
Executable File : application1.dll
```

サーバアプリケーションにリンクされているライブラリの格納先が複数存在する場合は、アプリケーション使用パス(Path for Applicationセクション)を複数設定してください。
また、出口プログラムにリンクされているライブラリの格納先についても、同様にアプリケーション使用パス(Path for Applicationセクション)に設定してください。

Solaris32



注意

スレッドライブラリ(libthread.so)は、スレッドモードで作成する場合のみ必要です。

サーバアプリケーションは、APMと動的に結合します。このため、サーバアプリケーションは共用ライブラリとし、動的結合を可能とする構造(ダイナミックリンクライブラリ)としなければならず、動的リンク構造の実行可能プログラムとします。動的プログラム構造とすることはできません。この共用ライブラリには、サーバアプリケーションが使用するライブラリを結合してください。データベース管理システムのライブラリ以外で、サーバアプリケーションのリンク時に指定するライブラリを以降に示します。

APMの再作成

コード変換用関数などのように、ユーザアプリケーションから呼び出されているシンボルが、ユーザがアプリケーションにリンクしたライブラリとlibc.soなどのシステムライブラリとの両方に格納されているとき、ユーザアプリケーションを実行すると、無条件にlibc.soなどのシステムライブラリに格納されたシンボルが呼び出されます。これは、APM実行モジュールの先頭にlibc.soなどのシステムライブラリが結合されているためです。この場合、ユーザアプリケーションに結合されたライブラリのシンボルを呼び出すためには、APM実行モジュールの先頭にユーザアプリケーションが使用するライブラリを結合する必要があります。
APM実行モジュールの先頭にユーザアプリケーションが使用するライブラリを結合するためには、tdlinknormapmコマンドを使用し、APMを再作成してください。tdlinknormapmコマンドの使用方法については"リファレンスマニュアル(コマンド編)"を参照してください。

4.4.4 スレッドモードのアプリケーションのコンパイルとリンク

Solaris32

[リンク時に指定するライブラリ]

ライブラリ名	格納場所	用途
libthread.so	/usr/lib	スレッドライブラリ(必須)(注1)
libtdalcapi.so	TDのインストールディレクトリ/lib	TD ランタイム(必須)
libextpapiskl.so	EXTPのインストールディレクトリ/lib	TD ランタイム(必須)
librcobol.so	COBOLのインストールディレクトリ/lib	COBOL ランタイム(必須)
libOMcblMT.so	ODのインストールディレクトリ/lib	ODランタイム(注2)

(注1)
スレッドライブラリは必ず結合するライブラリの中で先頭に指定してください。

(注2)
中継用サーバアプリケーションを作成するとき、および、例外を使用するときに必要です。

なお、リンカにより結合処理を行う場合、以下のオプションを指定してください。

```
-dy -G
```

サーバアプリケーションとスケルトンをコンパイル後、リンクする手順について例を以下に示します。

[コンパイル・リンク手順の例]

```
%CORBA=/opt/FSUNod/include/COBOL
%export CORBA
%cobol -c tdsample1_s.cbl
%cobol -c TD_TDSAMPLE1_INTF_skel.cbl
%cobol -G -dy -Tm -o libtdsample1.so -lthread -L/opt/FSUNtd/lib -ltdalcapi
-L/opt/FSUNextp/lib -lextpapiskl tdsample1_s.o
TD_TDSAMPLE1_INTF_skel.o
```

注意

アプリケーションを作成する場合には、以下の注意が必要です。

- リンクオプションに-Tmオプションを指定してください。また、ソースのコンパイルとリンクは分けて実施し、リンク時のオプションにのみ-Tmオプションを指定してください。
- リンクオプションに-Tmオプションを指定することで、自動的にlibrcobol.soをリンクします。
このため、リンクライブラリにlibrcobol.soを指定する必要はありません。

4.4.5 プロセスモードのアプリケーションのコンパイルとリンク

Solaris32

サーバアプリケーションのリンク時に指定するライブラリを以下に示します。

[リンク時に指定するライブラリ]

ライブラリ名	格納場所	用途
libtdalcapi_nt.so	TDのインストールディレクトリ/lib	TD ランタイム(必須)
libextpapiskl_nt.so	EXTPのインストールディレクトリ/lib	TD ランタイム(必須)
libcobol.so	COBOLのインストールディレクトリ/lib	COBOL ランタイム(必須)
libOMcbl.so	ODのインストールディレクトリ/lib	ODランタイム(注)

(注)
中継用サーバアプリケーションを作成するとき、および、例外を使用するときに必要です。

なお、リンカにより結合処理を行う場合、以下のオプションを指定してください。

```
-dy -G
```

サーバアプリケーションとスケルトンをコンパイル後、リンクする手順について例を以下に示します。

[コンパイル・リンク手順の例]

```
%cobol -c TD_TDSAMPLE1_INTF_skel.cbl
%cobol -G -o libtdsample1_nt.so -L/opt/FSUNtd/lib -ltdalcapi_nt
-L/opt/FSUNextp/lib -lextpapiskl_nt tdsample1_s.o
TD_TDSAMPLE1_INTF_skel.o
```

注意

作成したアプリケーションライブラリにスレッドライブラリが結合されていないことを確認してください。

アプリケーションのコンパイル／リンクの手順に誤りがあった場合や、libthread.soがリンクされている場合、ワークユニット起動／停止処理またはクライアントとの通信が無応答状態となる可能性があります。その場合、以下の対処を行ってください。

1. 当該ワークユニット配下で動作しているアプリケーションプロセスのプロセスIDを特定します。

```
% ps -ef | grep ワークユニット名
```

2. 当該ワークユニット配下で動作しているアプリケーションプロセスの強制停止

```
% kill -9 プロセスID
```

リンク時の注意事項

サーバアプリケーションのリンク時には、上記のライブラリの他に、データベースや連携する他製品のライブラリなど、サーバアプリケーションで必要なライブラリがすべてリンクされていることを確認してください。また、リンクしたライブラリの格納先をワークユニット定義のアプリケーション使用ライブラリパス(Library for Applicationセクション)に設定してください。
リンクが完全でない場合やワークユニット定義のアプリケーション使用ライブラリパスが完全でない場合は、ワークユニット起動処理において実行ファイルのオープン処理に失敗し、メッセージ(EXTP4640,EXTP4643,EXTP4645,EXTP4508)が出力されます。
プロセスモードの場合でリンク時にlibOM.soを指定する場合には、必ず"/opt/FSUNod/lib/nt"を指定しなければなりません。

以下に必要なワークユニット定義の設定例を示します。

ワークユニット定義の設定例

サーバアプリケーションの実行ファイルが"application1.so"、格納先が/application/lib、サーバアプリケーションにリンクされているライブラリの格納先が、それぞれ、"/product1/lib"、"/product2/lib"で、プロセスモードでかつリンク時にlibOM.soを指定するの場合の定義例を以下に示します。

```
[WORK UNIT]
Name: SAMPLEWU
Kind: ORB
[APM]
Name: TDNORMnt

[Control Option]
…
Path: /application/lib
Library for Application: /product1/lib
Library for Application: /product2/lib
Library for Application: /opt/FSUNod/lib/nt
…
Environment Variable: MANPATH=/opt/FJSVCOBop/man/%L:/opt/FJSVCOBop/man
Environment Variable: NLSPATH=/opt/FJSVCOBop/lib/nls/%1/%c/%N.cat:
/opt/FJSVCOBop/lib/nls/C/%N.cat;
…

[Application Program]
```

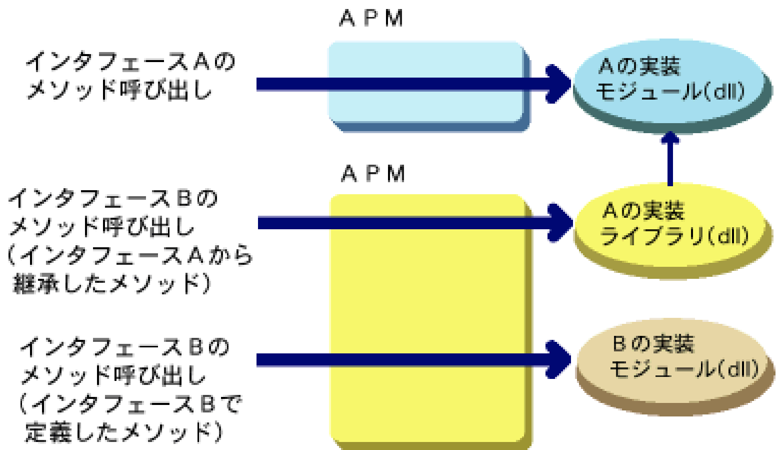
```
…
Executable File : application1.so
```

サーバアプリケーションにリンクされているライブラリの格納先が複数存在する場合は、アプリケーション使用ライブラリパス(Library for Applicationセクション)を複数設定してください。
また、出口プログラムにリンクされているライブラリの格納先についても、同様にアプリケーション使用ライブラリパス(Library for Application セクション)に設定してください。

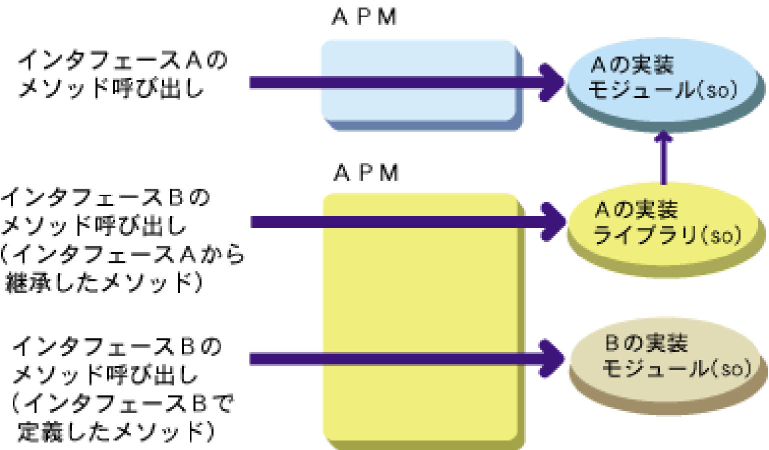
4.4.6 継承について

あるインタフェースで定義したオペレーションを、別のインタフェースに引き継ぐことを可能にするための機能です。継承を利用した場合、クライアントからは継承を意識せずに、継承したオペレーションを呼び出すことができます。
メソッドの呼び出しイメージについて以下に示します。

Windows32



Solaris32



継承は、IDL定義ファイルに継承の指定を記述することによって使用することができます。継承先インタフェースは、継承元インタフェースのスコープ名をコロン(":")に続いて指定します。継承の指定例を以下に示します。
この例では、インタフェース“A”が継承元インタフェース、インタフェース“B”が継承先インタフェースとなります。

```
interface A {
        long op1(in long a);
    };
interface B:X::A {
        long op2(in long b);
 };
```

X：module名
A、B：interface名

継承先のIDL定義内に継承元のIDL定義をincludeする記述が必要です。include方法の例を以下に示します。

A.idl

```
module X{
 interface A{
   long op1();
 };
};
```

B.idl

```
#include "A.idl"
module Y{
 interface B:X::A{
   long op2();
 };
};
```

X::Aにop2を追加したY::Bを作成する方法

継承を使用する場合、モジュール作成時に継承元のインタフェースを実装しているライブラリをリンクします。継承を使用したアプリケーションの作成時に必要な注意点を、以下に示します。

継承元のインタフェースの状態

継承先のIDL定義をIDLコンパイルする場合、継承元は以下の状態であることが考えられます。ここでは、以下の状態の注意点を説明します。

- 継承元のインタフェースがインタフェースリポジトリに登録されていない場合
- 継承元のインタフェースがインタフェースリポジトリに登録されている場合

継承元のインタフェースがインタフェースリポジトリに登録されていない場合

tdcコマンド実行時に -I オプションを指定してください。tdcコマンドの詳細については、"リファレンスマニュアル（コマンド編）"を参照してください。

継承元のインタフェースがインタフェースリポジトリに登録されている場合

継承先のIDL定義に継承元のIDL定義をincludeする記述を追加します。tdcコマンドは、-Iオプションと共に、-updateオプションを設定して実行してください。tdcコマンドの詳細については、"リファレンスマニュアル（コマンド編）"を参照してください。

継承元のライブラリの所在

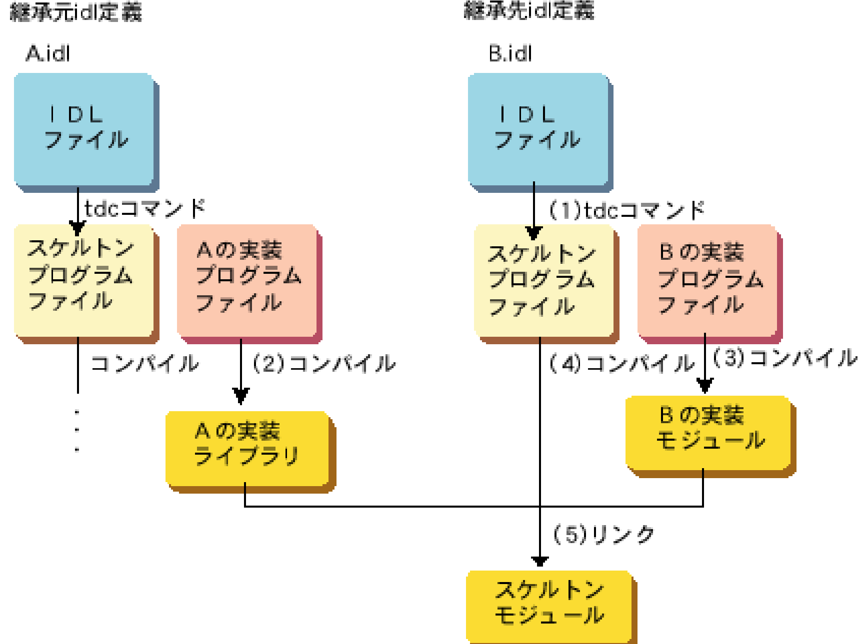
継承先のサーバアプリケーションをコンパイルする場合の注意点を説明します。

- 継承元インタフェースが実装されているライブラリのリンク
- 継承元インタフェースが実装されているライブラリのパスの設定

継承元インタフェースが実装されているライブラリのリンク

継承を使用する場合のサーバアプリケーション作成方法

(1) 継承先のIDL定義ファイル（B.idl）をtdcコマンドでコンパイルします。
(2) 継承元となるAの実装部は、スケルトンと別ライブラリとして作成します。
(3) 継承先となるBの実装部をコンパイルし、ライブラリを作成します。
(4) (1)で生成されたスケルトンをコンパイルします。
(5) (2)(3)(4)で生成されたライブラリをリンクします。



Windows32

ライブラリ

ダイナミックリンクライブラリ(.lib)

モジュール

ダイナミックリンクライブラリ(.dll)

Solaris32

ライブラリ

ダイナミックリンクライブラリ(.so)

モジュール

ダイナミックリンクライブラリ(.so)

継承元インタフェースが実装されているライブラリのパスの設定

継承元インタフェースが実装されているライブラリのパスの設定は、ワークユニット定義で設定します。

Windows32

“Path for Application:”ステートメントに、継承元インタフェースへのパスを記述します。

Solaris32

“Library for Application:”ステートメントに、継承元インタフェースへのパスを記述します。

4.5 ワークユニット定義の作成

ローカルトランザクション運用におけるワークユニット定義の概要について説明します。ワークユニット定義の詳細については、“付録A ワークユニット定義の記述形式”および“OLTPサーバ運用ガイド”を参照してください。

定義情報の設定

ワークユニット定義ファイルに記述する定義情報について、以下に示します。

ワークユニット名

ワークユニットを操作するためのワークユニット名を[WORK UNIT]セクションで設定します。ワークユニット名は、ワークユニット単位に1つ設定することができます。

APM名

APMの名前を[APM]セクションに設定します。APM名を“TDNORM”と指定してください。

Solaris32

ただし、プロセスモードで動作させる場合は、“TDNORMnt”を指定してください。
また、APM再作成コマンドで作成したAPMを使用する場合は、再作成したAPM名を指定してください。

制御オプション

アプリケーションが動作するためのカレントディレクトリやアプリケーションが格納されているライブラリパスなどの環境情報を[Control Option]セクションで設定します。

Solaris32

なお、COBOLアプリケーションを実行するためには以下の環境変数が必要です。使用するCOBOLのマニュアルを参照し、“Environment Variable:”ステートメントに環境変数を設定してください。また、下記以外のCOBOLの環境変数については、COBOLマニュアルを参照し、必要に応じて設定してください。

- NLSPATH
- MANPATH
- LM_LICENSE_FILE

アプリケーション情報

ワークユニットで動作させるアプリケーション名やオブジェクト名などの情報を、[Application Program]セクションで設定します。
ワークユニットにアプリケーションを追加する場合、追加するアプリケーション単位に[Application Program]セクションに情報を記載して追加します。たとえば、ワークユニットに4つの別々なアプリケーションを動作させる場合には、4つの[Application Program]セクションを記載します。
ワークユニットからアプリケーションを削除する場合は、不要となるアプリケーションに対応する[Application Program]セクションを削除します。

非常駐アプリケーション情報

非常駐形態で動作させるアプリケーションの多重度などの情報を[Nonresident Application Process]セクションで設定します。

Solaris32

なお、非常駐形態で運用する場合は、[Control Option]セクションの“Environment Variable:”ステートメントに環境変数“EXTP_NONRESIDENT_OPTION=on”を設定してください。環境変数が設定されていない場合、サーバアプリケーションの処理が完了しても、サーバアプリケーションはメモリ上から消去されません。

マルチオブジェクト常駐アプリケーション情報

マルチオブジェクト常駐形態で動作させるアプリケーションの多重度などの情報を[Multiresident Application Process]セクションで設定します。

4.6 アプリケーションの登録

サーバアプリケーションを他のアプリケーションからアクセス可能にするためには、目的のアプリケーションを識別するためのオブジェクトリファレンスを作成する必要があります。また、同時に作成したオブジェクトリファレンスをネーミングサービスに登録することによって、他のアプリケーションからのアクセスが可能になります。

トランザクションアプリケーションの場合、オブジェクトリファレンスの作成とネーミングサービスへの登録は、次の2つの方法で行うことができます。

- ワークユニット起動による自動登録
- OD_or_admコマンド(注)による手動登録

ワークユニット起動による自動登録の場合は、ワークユニットを起動するとワークユニットで指定された各アプリケーションごとにオブジェクトリファレンスが作成され、ネーミングサービスに登録されます。この場合、ワークユニットを停止すると、オブジェクトリファレンスは、自動的にネーミングサービスから削除され、無効となります。
OD_or_admコマンドによる手動登録の場合は、ワークユニットの起動前に、事前にOD_or_admコマンドによりオブジェクトリファレンスの作成とネーミングサービスへの登録を行います。この場合は、ワークユニットの起動、停止にかかわらず、オブジェクトリファレンスは有効です。

2つの方法とも、ネーミングサービスには以下の名前でオブジェクトリファレンスを登録します。

```
モジュール名::インタフェース名
```

また、インプリメンテーションリポジトリIDとしては、以下のIDを使用します。

```
ワークユニット種別がORBの場合　　: FUJITSU-Interstage-TDLC
ワークユニット種別がWRAPPERの場合　: FUJITSU-Interstage-TDRC
```

なお、2つの方法のどちらを使用するかは、ワークユニット定義で指定する必要があります。詳細は“OLTPサーバ運用ガイド”を参照してください。

注）ロードバランス機能を使用する場合は、odadministerlbコマンドを使用します。

注意事項

手動登録する場合、OD_or_admコマンドを実行する前に、Interstageが起動されている必要があります。
OD_or_admコマンドの詳細については、“リファレンスマニュアル（コマンド編）”を参照してください。

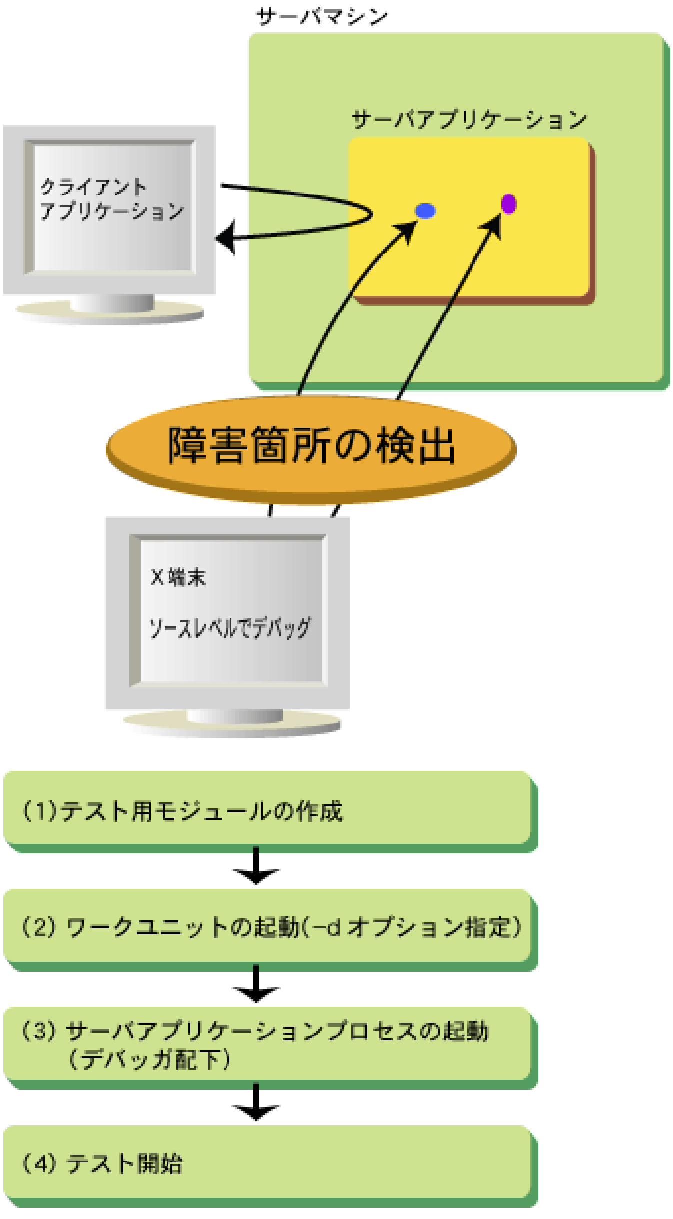
4.7 アプリケーションのテスト

作成したアプリケーションのテスト方法を、以下の内容で説明します。

- サーバアプリケーションのテスト方法
- クライアントアプリケーションのテスト方法
- スナップショットによるテスト支援
- 運用環境への移行
- 運用環境におけるテスト

サーバアプリケーションのテスト方法

サーバアプリケーションのテストを行う場合、実際にクライアントアプリケーションと結合して行います。このとき、サーバアプリケーションをデバッガ配下で動作させることで、サーバアプリケーションが正しく作成されているか確認できます。
COBOLデバッガと連携するときの動作の概要と、サーバアプリケーションをデバッガ配下でテストする場合の手順を、以下に示します。



テスト用モジュールの作成

Windows32

テストを行うサーバアプリケーションを、TESTオプションを指定してコンパイルします。リンクする場合は、リンクオプション"/DEBUG"、および"/DEBUGTYPE:COFF"を指定します。コンパイル方法の詳細についてはCOBOLのマニュアルを参照してください。

Solaris32

テストを行うサーバアプリケーションを、TESTオプションを指定してコンパイルします。コンパイル方法の詳細についてはCOBOLのマニュアルを参照してください。
テスト用モジュールの作成例を以下に示します。

```
%CORBA=/opt/FSUNod/include/COBOL
%export CORBA
%cobol -WC,"TEST" -c tdsample1_s.cbl
%cobol -WC,"TEST" -c TD_TDSAMPLE1_INTF_skel.cbl
%cobol -G -dy -o libtdsample1.so -lthread -L/opt/FSUNtd/lib -ltdalcapi
-L/opt/FSUNextp/lib -lextpapiskl -L/opt/FJSVCOBop/lib -lrcobol tdsample1_s.o
TD_TDSAMPLE1_INTF_skel.o
```

ワークユニットの起動

テストを行うサーバアプリケーションのワークユニット定義を作成し、tdstartwuコマンドでデバッグオプション-dを指定してワークユニットを起動します。-dオプションを指定してワークユニットを起動した場合、ワークユニットの動作環境まで作成し、APMプロセスを起動せずにコマンドは復帰します。tdstartwuコマンドの詳細については“リファレンスマニュアル（コマンド編）”を参照してください。

デバッグ用のワークユニットの起動例を、以下に示します。

Windows32

```
>tdstartwu -d TDSAMPLE1
>
```

Solaris32

```
%tdstartwu -d TDSAMPLE1
```

COBOLデバッガの起動 Windows32

コマンドプロンプトからwinsvdコマンドでCOBOLデバッガを起動し、APMプロセスをCOBOLデバッガ配下で起動します。APMプロセスをCOBOLデバッガ配下で起動する際のwinsvdコマンドのパラメタのうち、基本的なパラメタの指定方法を以下に示します。

```
> winsvd  /G開始プログラム名  /Sソースファイル格納フォルダ名
/Dデバッグ情報格納フォルダ名  APMモジュール名  業務システム名
ワークユニット名  オブジェクト名  動作システム種別
```

/G開始プログラム名

デバッグを開始する外部プログラム名を指定します。

/Sソースファイル格納フォルダ

COBOLソースプログラムの格納フォルダを指定します。

/Dデバッグ情報ファイル格納フォルダ

デバッグ情報ファイルの格納フォルダを指定します。デバッグ情報ファイルはコンパイル時に生成される拡張子が“.svd”のファイルです。

APMモジュール名

“C:¥INTERSTAGE¥extp¥bin¥extp_apmXXX”を指定します。

XXX : ワークユニット定義で指定したAPM名を指定します。

業務システム名

“td001”を指定します。

ワークユニット名

本APMが動作するワークユニット名を指定します。

オブジェクト名

ワークユニット定義で指定した本APMのオブジェクト名を指定します。

動作システム種別

“T”を指定します。

APMプロセスの起動 Solaris32

端末から、テスト対象のCOBOLプログラムを指定したsvdコマンドにより、COBOLデバッガを起動し、APMプロセスをそのデバッガ配下で起動します。APMプロセスを起動する際のsvdコマンドのパラメタのうち、基本的なパラメタの指定方法を、以下に示します。

```
% svd  -p 開始プログラム名  -s ソースファイル格納ディレクトリ名
-k デバッグ情報格納ディレクトリ名 ファイル名 APMモジュール名 業務システム名
ワークユニット名  オブジェクト名  動作システム種別
```

-p 開始プログラム名

デバッグを開始する外部プログラム名を指定します。

-s ソースファイル格納ディレクトリ名

COBOLソースプログラムの格納ディレクトリを指定します。

-k デバッグ情報格納ディレクトリ名

デバッグ情報ファイルの格納ディレクトリを指定します。デバッグ情報ファイルはコンパイル時に生成される拡張子が".svd"のファイルです。

ファイル名

"EXTPのインストールディレクトリ/FSUNextp/bin/extp_apmenv"を指定します。

APMモジュール名

"EXTPのインストールディレクトリ/FSUNextp/bin/extp_apmXXX"を指定します。

XXX : ワークユニット定義で指定したAPM名を指定します。
また、拡張システムの場合は、以下のように設定します。
XXX : "APM名_拡張システム名"を設定します。

業務システム名

デフォルトシステムの場合は、"td001"を指定します。拡張システムの場合は、システム名を設定します。

ワークユニット名

本APMが動作するワークユニット名を指定します。

オブジェクト名

ワークユニット定義で指定した本APMのオブジェクト名を指定します。

動作システム種別

"T"を指定します。

APMプロセスの起動例を以下に示します。

```
% PATH=/opt/FJSVCOBop/bin:${PATH}; export PATH
% LD_LIBRARY_PATH=/opt/FJSVCOBop/lib:/usr/dt/lib:/usr/openwin/lib:${LD_LIBRARY
_PATH};export LD_LIBRARY_PATH
% MANPATH=/opt/FJSVCOBop/man/%L:/opt/FJSVCOBop/man:${MANPATH}; export MANPATH
% NLSPATH=/opt/FJSVCOBop/lib/nls/%1/%c/%N.cat:/opt/FJSVCOBop/lib/nls/C/%N.cat;
export NLSPATH
% XUSERFILESEARCHPATH=/opt/FJSVCOBop/lib/app-defaults/%L/%N:/opt/FJSVCOBop/lib
/app-defaults/%N; export XUSERFILESEARCHPATH
%
% svd -p TDSAMPLE1 -s /opt/FSUNtd/sample/SERVER -k /opt/FSUNtd/sample/SERVER
/opt/FSUNextp/bin/extp_apmenv /opt/FSUNextp/bin/extp_apmTDNORM td001 TDSAMPLE1
TDSAMPLE1/INTF T
```

COBOLデバッガの詳細については、COBOLのマニュアルを参照してください。

クライアントアプリケーションのテスト方法

クライアントアプリケーションのテスト方法は、クライアントアプリケーションを動作させるオペレーティングシステムやミドルウェアによって異なります。お使いのオペレーティングシステムやミドルウェアごとに推奨される方法でテストを行ってください。

スナップショットによるテスト支援

スナップショットを使用して、クライアントからの要求に対する入出力情報をワークユニット単位に取得することにより、アプリケーションのデバッグを行うことができます。
詳細は、"第7章 スナップショット機能"を参照してください。

実行時のエラーメッセージ情報の取得によるデバッグ Solaris32

COBOLサーバアプリケーション開発時に、ワークユニット定義に以下を追加することで、COBOLランタイムからのメッセージが標準エラー出力に出力されデバッグが容易になります。

Environment Variable:NLSPATH=COBOL_HM/lib/nls/%L/%N.cat:COBOL_HM/lib/nls/C/%N.cat

注）
COBOL_HM：COBOLコンパイラのインストールディレクトリを指定します。

運用環境への移行

開発環境でテストした資材を運用環境へ移行するための作業手順と、運用環境でのテスト方法について説明します。

移行手順

クライアント資材の移行

クライアント資材は、サーバ資材を移行してサーバアプリケーションがコンパイルされるまでに移行してください。

サーバ資材の移行

サーバ資材を運用環境に移行する手順について、以下に示します。

1.Interstageシステムの停止

2.アプリケーションソースの移行

3.定義ファイルの移行

4.IDLファイルのコンパイルの実施

5.Interstageシステムの停止

6.サーバアプリケーションのコンパイルを実施

7.オブジェクトリファレンスの登録

8.ワークユニット定義の登録

9.Interstageシステムの起動

10.ワークユニットの起動

1. 開発環境のInterstageシステムを停止します。
2. 開発環境からサーバアプリケーションのプログラムソースを運用環境に複写します。
3. 開発環境からIDLファイル、ワークユニット定義ファイルを運用環境に複写します。
4. 3.で複写したIDLファイルをもとに、tdcコマンドを実行し、スタブファイル、スケルトンファイルを作成します。
5. 運用環境のInterstageシステムの停止
6. 2.で複写したサーバアプリケーションプログラムソースと4.で作成したスケルトンファイルにより、サーバアプリケーションを作成します。
7. ワークユニット定義の“ネーミングサービスの登録形態”が“MANUAL”の場合、オブジェクトリファレンスを登録します。
8. 3.で複写したワークユニット定義を登録します。
9. 運用環境のInterstageシステムを起動します。

10. 運用環境のワークユニットを起動します。

運用環境におけるテスト

開発環境では各モジュールの単体テストを行いますが、運用環境ではシステム全体として以下に示すテストが必要です。

- システム負荷テスト
- 業務に沿った運用テスト
- 業務に則した性能テスト

それぞれのテスト方法について、以下に示します。

システム負荷テスト

システム負荷テストは業務を遂行するために、システム上のすべてのコンポーネントを含めたテストを実施します。システムの負荷をあげるためには、システムへのデータ入力の頻度(呼量と呼びます)をあげると、実施できます。たとえば、多数のCORBAクライアントからの入力の場合、CORBAクライアントを高速なマシン上で動作させると、呼量が増加し、負荷があげられます。また、Web連携の負荷をあげるときは、WebStoneなどをサーバに設定し、同様に呼量を増加させることができます。

業務に沿った運用テスト

業務に沿った運用テストは、実際の業務を想定したテストを実施し、システムとして運用に問題がないかを確認します。したがって、システムで利用する製品および業務アプリケーションすべてを動作させます。たとえば、受注業務の運用テストを実施する場合、受注業務で利用する全製品とアプリケーションを動作させて、受注業務の開始/終了、データ入力とその処理などを実施します。

業務に則した性能テスト

性能テストは、業務運用中の性能について測定し、問題がないかを確認するテストです。

第5章 Interstageの特徴的な機能

Interstageの特徴的な機能を使用したトランザクションアプリケーションの作成方法について、以下に示す内容で説明します。

- セションIDを採番するためのトランザクションアプリケーションの作成
- プロセスバインド機能を使用したトランザクションアプリケーションの作成
- セション情報管理機能を使用したトランザクションアプリケーションの作成

5.1 セションIDを採番するためのトランザクションアプリケーションの作成

5.1.1 セションID

セションIDとは、トランザクションアプリケーションにおいて、クライアントアプリケーションを識別するための識別情報です。

セションIDは、以下の機能で使用できます。

- プロセスバインド機能
- セション情報管理機能

Windows32 Solaris32

- AIM連携のセション継続機能
  注：AIM連携のセション継続機能については、"NETSTAGE Director ユーザーズガイド"を参照してください。

セションIDの形式は、オクテット型の配列で配列サイズは48バイトです。

セションIDは、トランザクションアプリケーションで"セションID採番API"を発行して採番します。セションID採番APIの詳細については、"リファレンスマニュアル（API編）"を参照してください。

5.1.2 プログラミングの流れ

5.1.2.1 IDL定義

IDL定義の記述例を示します。

```
typedef octet SessionID [48] ;          ← セション IDの形式です。
  module A {                               SessionID という名前で宣言し
        interface B {                      ています。
                long op0(
                        …
                        out SessionID outSessionID,
                        … );
                …
        } ;
} ;
```

5.1.2.2 ワークユニット定義

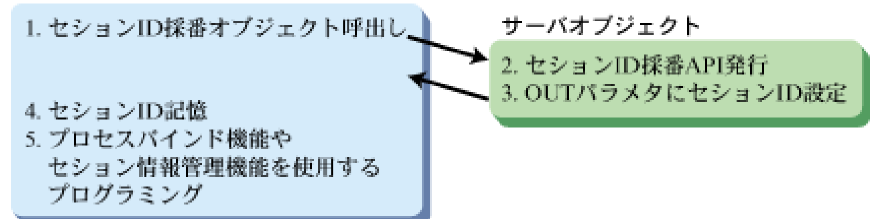
一般のワークユニット定義と相違ありません。

5.1.2.3 プログラミングイメージ

セションIDの採番方法には以下の方法があります。

- セションID採番用のオブジェクトを個別に作成し、セションID採番APIを発行して採番
- プロセスバインド機能、セション情報管理機能を使用するオブジェクトのメソッドとして作成し、セションID採番APIを発行して採番

セションIDを採番するオブジェクトのイメージを以下に示します。



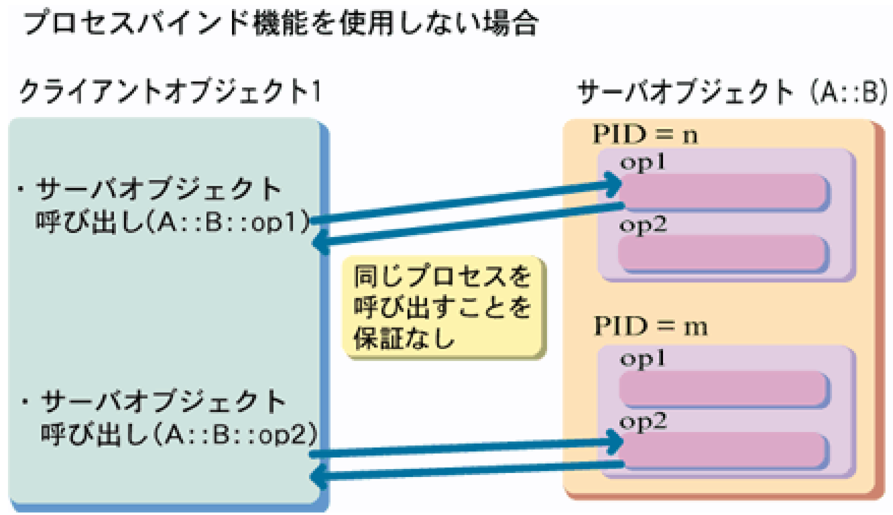
5.2 プロセスバインド機能を使用したトランザクションアプリケーションの作成

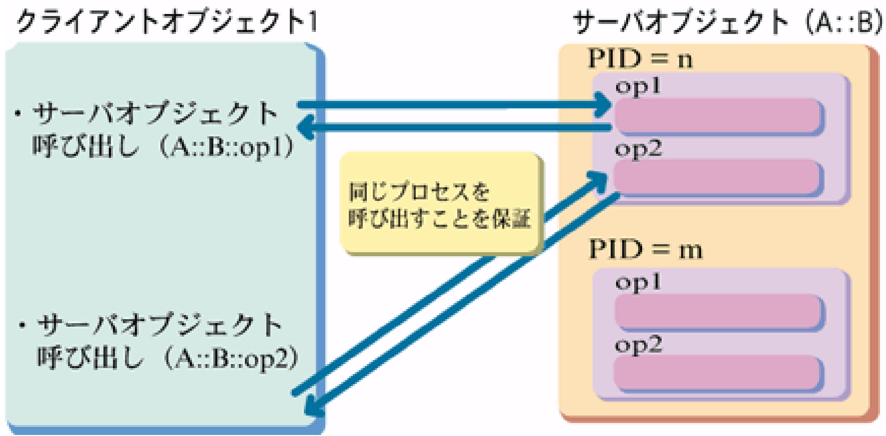
プロセスバインド機能を使用してトランザクションアプリケーションを作成する方法について説明します。

5.2.1 概要

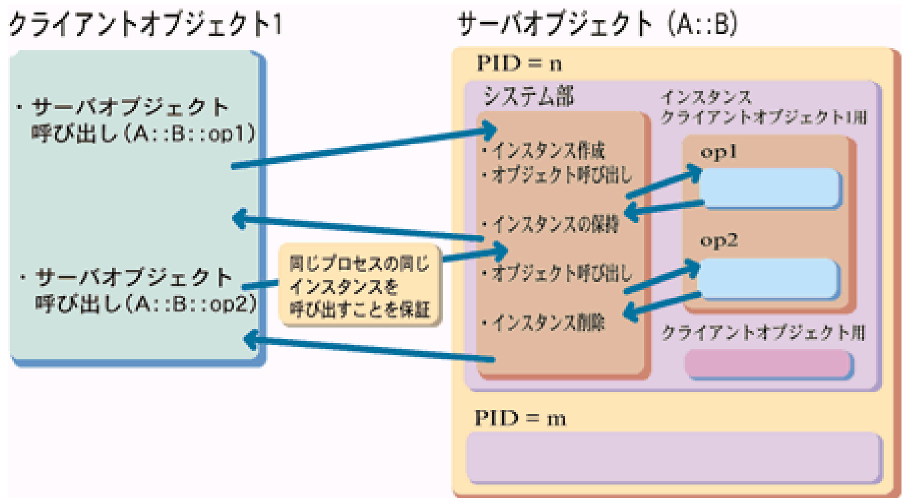
クライアントオブジェクトから複数回呼び出されるサーバオブジェクトにおいて、ワークユニット定義で多重度に2以上を定義している場合、毎回の呼び出しで動作するサーバオブジェクトの処理は、同じプロセスで処理が行われる保証はありません。

本機能を使用することにより、クライアントオブジェクトから複数回呼び出されるサーバオブジェクトにおいて、ワークユニット定義で多重度に2以上を定義している場合でも、初回呼び出し時に動作したプロセスのサーバオブジェクトで処理を行うことが保証されます。





C++言語で作成されたオブジェクトのバインドは、プロセス内に作成されるインスタンスをバインドします。

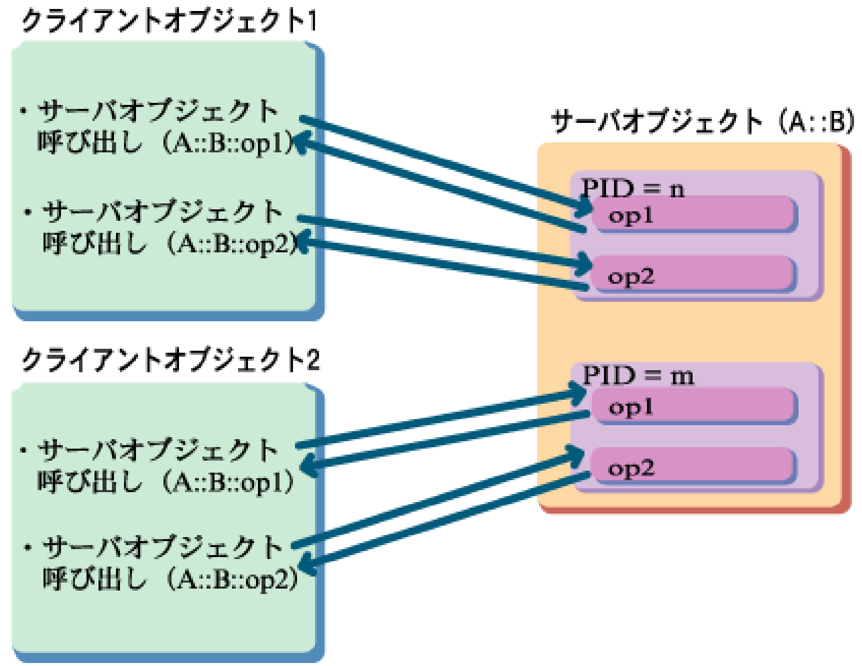


セション開始要求(初回呼び出し)のプロセスへの振り分け方式

プロセスバインド機能では、セション開始要求を受け取ったサーバアプリケーションプロセスに、そのセションIDに対する処理がバインドされます。通常、セション開始要求は、ワークユニット内で最後に要求待ちとなったサーバアプリケーションプロセスに振り分けられます。そのため、クライアントからの要求数が少ない場合、すべてのクライアントが特定のプロセスにバインドされ、その結果、1プロセスにセションが集中して性能劣化を引き起こすことがあります。

これを回避するためには、クライアントオブジェクトからのセション開始要求を、サーバオブジェクトのプロセスに均等に割り当てる必要があります。本機能を使用する場合は、クライアントからの要求を、要求待ちのサーバアプリケーションプロセスに振り分ける方式として、必ず、FIFO方式を設定してください。FIFO方式にすることにより、セション開始要求が各プロセスに均等に割り振られ、性能劣化を防止することができます。

要求待ちのサーバアプリケーションプロセスに振り分ける方式については、"OLTPサーバ運用ガイド"の"キュー制御"を参照してください。



5.2.2 プログラミングの流れ

5.2.2.1 プログラミングの概要

クライアントオブジェクトとサーバオブジェクト内のプロセスをバインドするために、セションIDを使用します。

セションIDをサーバオブジェクトの呼び出し時に、パラメタに設定することによりシステムがクライアントオブジェクトとサーバオブジェク

ト内のプロセスをバインドします。
セションID採番方法は、"5.1 セションIDを採番するためのトランザクションアプリケーションの作成"を参照してください。
セションIDのパラメタへの設定方法は以下のようになります。

1. オクテット型の配列で配列サイズは48バイトのパラメタを含むオペレーションをIDL定義に定義します。
2. IDL定義で定義したパラメタ名をワークユニット定義に定義します。
3. サーバオブジェクト呼び出し時に、採番したセションIDを1.および2.で定義したパラメタに設定します。
4. サーバオブジェクトでは、バインドを継続したい場合、セション継続APIを発行します。
5. 2回目以降の呼び出し時も、3.の手順を行います。
6. バインドを継続する必要がなくなった場合、セション継続APIを発行せずにサーバオブジェクトから復帰します。

IDL定義

IDL定義の記述例を示します。

```
typedef octet SessionID [48] ;
   module A
         interface B
                  long op1(
                           …
                           in SessionID inSessionID,
                           … );
                  long op2(
                           …
                           in SessionID inSessionID,
                           … );
                  long op3(
                           …
                           in SessionID inSessionID,
                           … );
        } ;
} ;
```


ワークユニット定義

ワークユニット定義の記述例を示します。

```
[WORK UNIT]
Name:…
Kind:ORB
[APM]
Name:TDNORM
[Control Option]
  …

[Application Program]
Destination: A/B
Application Language: CPP              ←── プログラミング言語を指定
Bind Type: INSTANCE                    ←── プロセスバインド機能の使用を指定
Method Name to Begin Session: op1      ←── セションを開始するメソッド(オペレーション)
SessionID Param: inSessionID           ←── IDL 定義で定義したセション ID 用のパラメタ
Request Assignment Mode: FIFO          ←── 要求メッセージ振り分け方式を指定
```

プロセスバインド管理機能を使用するために必要なワークユニット定義を以下に示します。
［Application Program］セクションに以下の定義を設定します。

Bind Type：バインド形式

バインド形式を指定します。

"DISABLE"：プロセスバインド機能を使用しない。
"PROCESS "：プロセス
"INSTANCE"：インスタンス

プロセスバインド機能を使用する場合、必須です。

Application Language:(アプリケーション言語)がC言語または、COBOL（COBOLはWindows(R)版、Solaris版のみです）の場合、"PROCESS "を設定してください。Application Language:(アプリケーション言語)がC++言語の場合、"INSTANCE"設定してください。

SessionID Param：セションID通知パラメタ

IDL定義で定義したセションIDを設定するためのパラメタ名を設定します。

プロセスバインド機能を使用する場合、必須です。

Method Name to Begin Session：セションを開始するメソッド名

セションを開始するメソッド名を設定します。

プロセスバインド機能を使用する場合、必須です。

Maximum Session Active Time for Client：クライアント思考時間の最大時間

クライアント思考時間の最大時間を秒単位で設定します。

0～86400の整数値。

本ステートメントは省略可能です。

本ステートメントを省略した場合、省略値として300が設定されます。

0を指定した場合は時間監視を行いません。

Recover Exit Program：異常出口プログラム

クライアント思考時間超過となった場合に呼び出される異常出口プログラム名を設定します。

本ステートメントは、Application Language:(アプリケーション言語)がC言語または、COBOL（COBOLはWindows(R)版、Solaris版のみです）の場合のみ有効です。Application Language:(アプリケーション言語)がC++言語の場合、本ステートメントは無視されます。

本ステートメントは省略可能です。

Request Assignment Mode：要求メッセージ振り分け方式

クライアントからの要求メッセージを、要求待ちのサーバアプリケーションプロセスに振り分ける方式を指定します。

プロセスバインド機能を使用する場合、本ステートメントに必ず“FIFO”を指定してください。

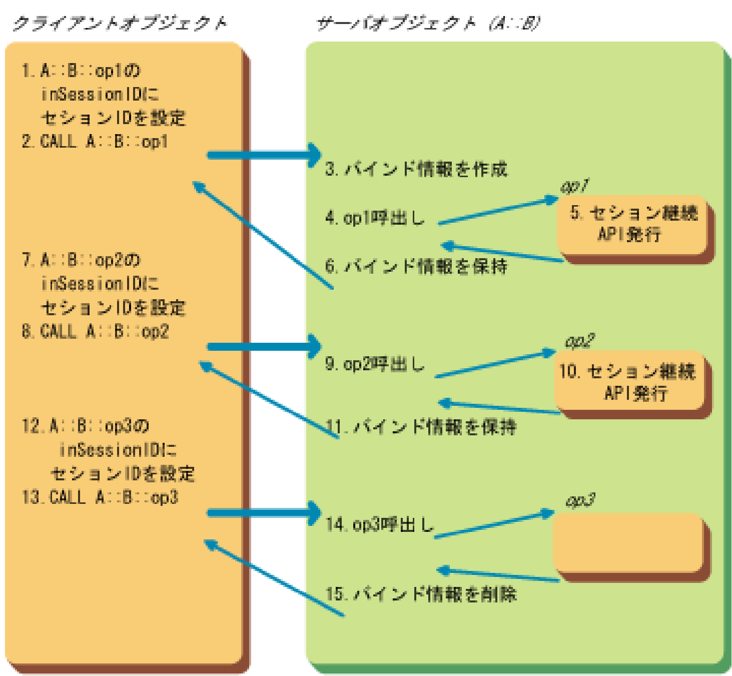
ワークユニット定義時の注意事項

- アプリケーション開発言語がC言語、COBOL（COBOLはWindows(R)版、Solaris版のみです）の場合、Bind Type:ステートメントに“INSTANCE”を設定することはできません。ワークユニット定義の登録に失敗します。
- アプリケーション開発言語がC++言語の場合、Bind Type:ステートメントに“PROCESS”を設定することはできません。ワークユニット定義の登録に失敗します。
- Bind Type:ステートメント に“PROCESS”または、“INSTANCE”を設定し、SessionID Paramステートメントを設定していない場合、ワークユニット定義の登録に失敗します。
- Bind Type:ステートメントに“PROCESS”または、“INSTANCE”を設定し、Method Name to Begin Sessionステートメントを設定していない場合、ワークユニット定義の登録に失敗します。
- Method Name to Begin Session:ステートメントに設定したメソッドのパラメタにSessionID Paramステートメントで設定したパラメタがない場合、ワークユニットの起動に失敗します。

- SessionID Param:ステートメントで設定したパラメタ名がIDL定義内でセションIDの形式と異なった型として定義されている場合、ワークユニットの起動に失敗します。
- Request Assignment Mode:ステートメントを省略した場合、または“LIFO”を指定した場合、1プロセスにセションが集中し性能劣化を引き起こすことがあります。

5.2.2.2 C言語・COBOLの場合の通信イメージ

C言語・COBOLの場合の通信イメージを以下に示します。
なお、COBOLはWindows(R)版、Solaris版のみ使用できます。

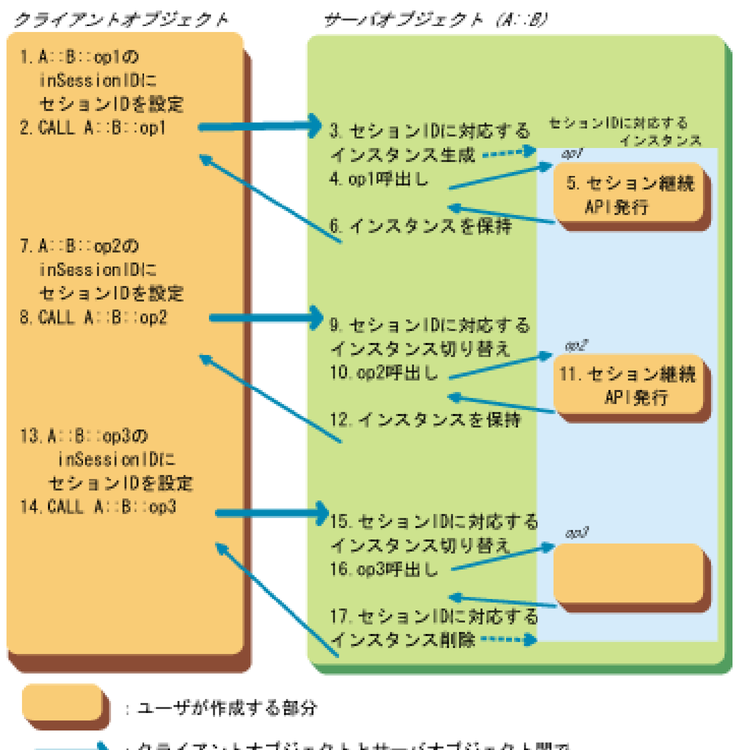


1. セションIDをA::B::op1のワークユニット定義に定義したパラメタに設定します。
2. A::B::op1を呼び出します。
3. バインド情報を作成します。
4. op1をスケジュールします。
5. セション継続APIを発行します。
6. セション継続APIが発行されているので、バインド情報を削除せずに保持します。
7. セションIDをA::B::op2のワークユニット定義に定義したパラメタに設定します。
8. A::B::op2を呼び出します。
9. op2をスケジュールします。
10. セション継続APIを発行します。
11. セション継続APIが発行されているので、バインド情報を削除せずに保持します。
12. セションIDをA::B::op3のワークユニット定義に定義したパラメタに設定します。
13. A::B::op3を呼び出します。

14. op3をスケジュールします。
15. セション継続APIが発行されていないので、バインド情報を削除します。

5.2.2.3 C++言語の場合の通信イメージ

C++言語の場合の通信イメージを以下に示します。



1. セションIDをA::B::op1のワークユニット定義に定義したパラメタに設定します。
2. A::B::op1を呼び出します
3. セションIDに対応したインスタンスを生成します。
4. 生成したインスタンスのop1をスケジュールします。
5. セション継続APIを発行します。
6. セション継続APIが発行されているので、インスタンスを削除せずに保持します。
7. セションIDをA::B::op2のワークユニット定義に定義したパラメタに設定します。
8. A::B::op2を呼び出します。
9. セションIDに対応し、保持されているインスタンスに切り替えます。
10. op2をスケジュールします。
11. セション継続APIを発行します。
12. セション継続APIが発行されているので、インスタンスを削除せずに保持します。
13. セションIDをA::B::op3のワークユニット定義に定義したパラメタに設定します。
14. A::B::op3を呼び出します
15. セションIDに対応し、保持されているインスタンスに切り替えます。

16. op3をスケジュールします。

17. セション継続APIが発行されていないので、セションIDに対応するインスタンスを削除します。

5.2.3 注意点

プロセスバインド機能を使用する場合、特別な注意が必要となります。
ここでは、プロセスバインド機能を使用するアプリケーションを作成する場合に必要となる注意点を説明しています。

5.2.3.1 クライアント異常を考慮したトランザクションアプリケーション

クライアントオブジェクトの異常やクライアントの電源断などにより、クライアントオブジェクトが異常終了した場合、トランザクションアプリケーション内に異常終了したクライアントオブジェクト用の領域が残ってしまいます。
このために、プロセスバインド機能ではクライアント思考時間監視機能と異常出口を提供しています。

クライアント思考時間監視

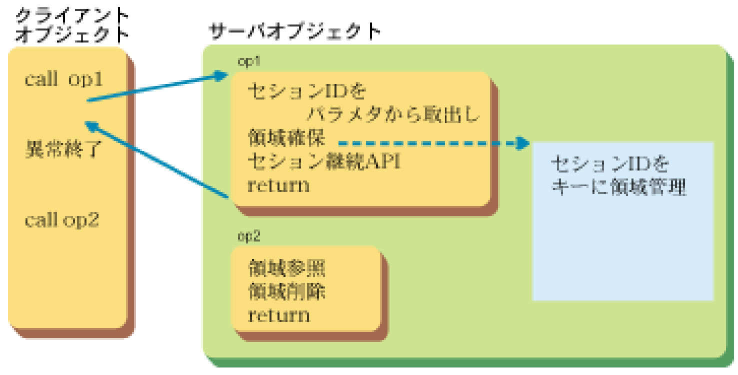
クライアント思考時間監視は、サーバオブジェクトを呼び出し、復帰した後、次の同一トランザクションアプリケーションを呼び出すまでの時間を監視します。
クライアントに制御が渡り、監視時間内に次の同一トランザクションアプリケーションが呼び出されない場合、システムは、クライアントにバインドされているトランザクションアプリケーションの異常出口をスケジュールします。
異常出口の登録方法については、トランザクションアプリケーション作成言語がC言語または、COBOL（COBOLはWindows(R)版、Solaris版のみです）の場合、本章を、アプリケーション作成言語がC++言語の場合、"3.3 サーバアプリケーションのソースの作成"を参照してください。

異常出口

クライアント思考時間がワークユニット定義で定義した値（秒）を超過すると、異常出口がスケジュールされます。
異常出口では、セションID参照APIを発行することによりセションID（クライアント識別子）を参照することができます。
セションIDをキーにトランザクションアプリケーション内のタイムアウトとなったクライアントオブジェクト用の領域を解放してください。

クライアントオブジェクト領域の解放

異常出口では、タイムアウトとなったクライアントのセションIDをセションID参照APIを発行することにより知ることができます。そのため、プロセスバインド機能を使用しているトランザクションアプリケーションではメソッド間にまたがって使用する領域を獲得する場合、セションIDをキーに獲得した領域の管理が必要となります。



op1でクライアントオブジェクト固有の領域として獲得した領域は、op2で参照後、削除されるロジックとなっています。また、クライアントオブジェクトがop1を呼び出した後、op2を呼び出すまでに異常終了した場合、op1でクライアントオブジェクト固有の領域として獲得した領域は、サーバオブジェクトプロセスが終了するまで解放されなくなってしまいます。
領域をセションIDをキーに管理することにより、異常出口で領域を解放することが可能となります。

C言語・COBOLで作成したオブジェクトにおける異常出口の作成と登録

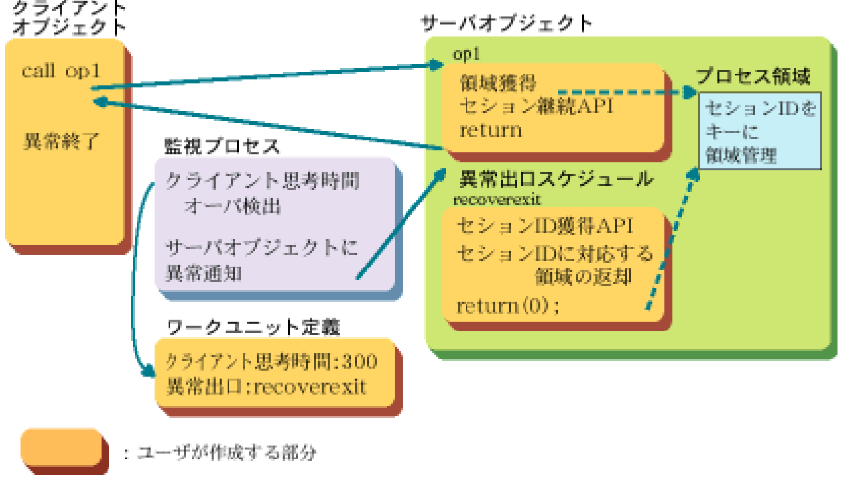
異常出口関数を作成します。異常出口では、以下のような処理を行うことが可能です。

- セションID参照APIを発行することにより、セションIDの参照が可能です。
- セションIDをキーに領域の解放が可能です。

- 標準エラー出力へのメッセージを出力することができます。

異常出口が正常終了(復帰値＝0)した場合、次の通信を行います。
異常出口が異常終了(復帰値＝0以外)した場合、異常出口が動作したトランザクションアプリケーションプロセスが異常終了します。
サーバプロセス異常終了時の動作は、“5.2.3.2 その他の注意点”を参照してください。
ワークユニット定義にて、異常出口名を定義します。詳細は、“OLTPサーバ運用ガイド”を参照してください。

なお、COBOLはWindows(R)版、Solaris版のみ使用できます。

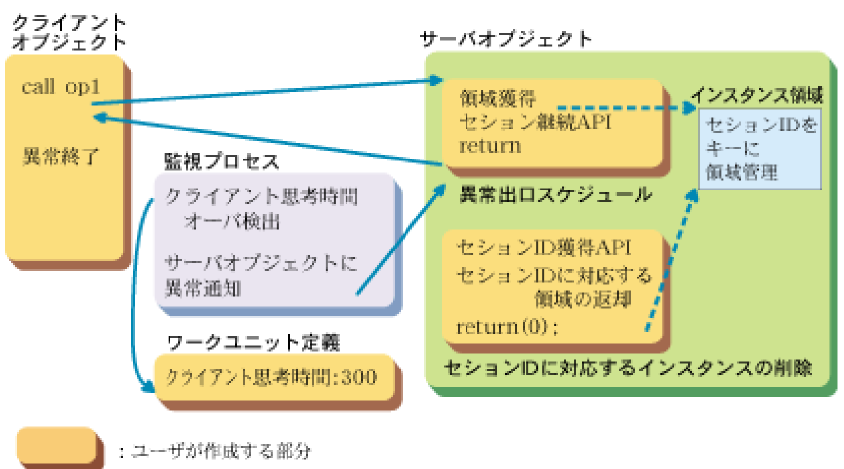


C++言語で作成したオブジェクトにおける異常出口の作成と登録

異常出口関数を作成します。関数名は、“ApmRecover”固定です。
異常出口では、以下のような処理を行うことが可能です。

- セションID参照APIを発行することにより、セションIDの参照が可能です。
- セションIDをキーに領域の解放が可能です。
- 標準エラー出力へのメッセージを出力することができます。

異常出口が正常終了(復帰値＝0)した場合、次の通信を行います。
異常出口が異常終了(復帰値＝0以外)した場合、異常出口が動作したトランザクションアプリケーションのプロセスが異常終了します。
サーバプロセス異常終了時の動作は、“5.2.3.2 その他の注意点”を参照してください。
tdcコマンド実行時に生成される“TD_オブジェクト名_proto.h”の異常出口(ApmRecover)のコメントアウト記述を外します。



5.2.3.2 その他の注意点

- ワークユニット配下のユーザオブジェクトにインスタンスが残っている状態で、ワークユニットの通常停止を実行した場合、ワークユニットの通常停止はリジェクトされます。
  この場合、ワークユニット配下のユーザオブジェクトのインスタンスがすべて削除されるまで待つか、必要に応じて、ワークユニットを強制停止してください。
  同様に、活性変更コマンドもリジェクトされます。
- ワークユニット定義で定義したパラメタ名のパラメタに値が設定されていない場合（0パディング）、バインドの必要がないと判断します（インスタンス管理を行いませんが、オブジェクト呼び出しは正常に行われます。インスタンスの継続は行えません。）。
- バインドの開始は、“Method Name to Begin Session:”ステートメント に指定したメソッドとなります。
- セションIDを採番するメソッドが、プロセスバインド機能を使用するオブジェクト内に含まれる場合、セションIDを採番するメソッドをプロセスバインドの範囲に含むことはできません。“Method Name to Begin Session:”ステートメントには、セションIDを採番した後に呼び出すメソッド名を指定してください。
- サーバオブジェクト異常時（プロセス異常終了時）には、サーバオブジェクトにバインドされているインスタンスの情報がすべて破棄されます。この場合、サーバオブジェクトが異常となる原因となったクライアントオブジェクトは、アプリケーション異常（10004）で復帰します。異常となったサーバオブジェクトにバインドされている他のクライアントオブジェクトは、次回の要求時にセション異常（10007）で復帰します。
- クライアント思考時間オーバによりインスタンスが削除されることがあるため、インスタンス管理機能を使用するオブジェクトでは、インスタンス内で獲得した領域は、デストラクタで返却するように作成してください。インスタンス用の領域をデストラクタで返却しない場合、メモリリークが発生します。
- プロセスバインド機能を使用してインスタンスを保持している場合、そのインスタンスはInterstageシステム定義ファイルのシステム規模で決定する同時接続クライアント数のうち、1つの接続を占有した状態となります。そのため、サーバアプリケーションが同時に使用するインスタンス数は、同時接続クライアント数以下になるように設計してください。この同時接続クライアント数を超えて要求を受けると、クライアントオブジェクトには復帰値10002（システムで異常検出）で復帰します。
- セションIDはワークユニットごとに管理します。そのため、同一ワークユニット内で同じセションIDを使用して複数のオブジェクトに対してインスタンスを生成することはできません。

5.3 セション情報管理機能を使用したトランザクションアプリケーションの作成

セション情報管理機能を使用してトランザクションアプリケーションを作成する方法について説明します。

5.3.1 概要

セション情報管理機能は、トランザクションアプリケーションで、クライアントごとの複数要求でプロセス間にまたがった情報（セション情報）を管理する機能です。セション情報管理機能は、CORBAオブジェクトであるセション情報管理オブジェクト(以降、SMOと呼びます)で機能が提供されます。
セション情報は、セションごと、または、クライアントごとに管理します。

セションは、セションIDにより識別されます。セションIDは、トランザクションアプリケーションのAPIを使用してサーバオブジェクトで獲得します。

クライアントは、要求元のクライアントを特定する情報であるクライアント識別子で識別されます。クライアント識別子は要求元クライアントのプロセスごとにInterstageが自動的に獲得します。サーバ側では各要求ごとにコンポーネントトランザクションサービスのAPIにより要求元のクライアント識別子を取得できます。

以下に示す、セション情報管理を使用するための、2形態のオブジェクトの作成について説明します。

- セション情報域を操作するオブジェクト
- セション情報管理の事象通知リスナオブジェクト

注意

サーバオブジェクトでトランザクションアプリケーションのAPIを使用してセションIDを獲得する場合、以下の各言語ごとに以下のAPIを使用します。

C

TD_getsessionid()

C++

TD::getsessionid

COBOL Windows32 Solaris32

TDGETSESSIONID

5.3.2 セション情報域を操作するオブジェクト

セション情報管理機能を使用するトランザクションアプリケーションのサーバオブジェクトの作成方法を説明します。

5.3.2.1 コンポーネントトランザクションサービスの環境定義

セション情報管理を使用する場合、コンポーネントトランザクションサービスの環境定義の[SYSTEM ENVIRONMENT]セクションに次の定義を行います。定義の詳細については、“チューニングガイド”を参照してください。

Using Session Information Management Object

セション情報管理オブジェクトを使用することを以下のように指定します。

```
Using Session Information Management Object:YES
```

Name of Session Information Management Object

セション情報管理オブジェクト(SMO)のネーミングサービスへの登録名を指定します。1つのネーミングサービスを運用するドメイン内の1つのサーバだけで、SMOを運用する場合は省略可能です。この場合は省略値である“ISTD::SMO”が使用されます。

```
Name of Session Information Management Object:オブジェクト名
```

5.3.2.2 プログラミングの流れ

(1)CORBAサービスの初期化

セション情報管理を使用するトランザクションアプリケーションでは、前出口プログラムでCORBAサービスの初期化をする必要があります。CORBAサービスの初期化の方法については、“付録D トランザクションアプリケーションのサンプルプログラム(サーバアプリケーション間連携編)”の各言語ごとのサンプルプログラムを参照してください。

注意

前出口プログラムとSMOを操作する処理は1つのライブラリとして作成する必要があります。本処理を前処理と別のライブラリとして作成する場合であっても、SMOの各APIの呼び出す処理は前出口プログラムと同一のライブラリとして作成することが必要です。

(2)セション情報管理オブジェクトのオブジェクトリファレンスの獲得

次の処理により前出口プログラムでSMOのオブジェクトリファレンスを獲得します。

SMOのネーミングサービス登録名の獲得

SMO名獲得APIによりSMOのネーミングサービスへの登録名を獲得します。
各言語ごとのSMO名獲得APIの使用方法については、“リファレンスマニュアル(API編)”を参照してください。

ネーミングサービスからのSMOのオブジェクトリファレンスの獲得

ネーミングサービスへ登録したSMOの登録名をもとに、ネーミングサービスからSMOのオブジェクトリファレンスを獲得します。

注意

本処理で獲得したセション情報管理オブジェクトのオブジェクトリファレンスは、セション情報管理オブジェクトへのすべての要求が終了した時点で解放する必要があります。

(3)クライアント識別子の獲得

セション情報域を、要求元のクライアントごとに獲得する場合に、クライアントを特定するクライアント識別子をコンポーネントトランザクションサービスから受け取り、セション情報管理の各操作時に通知する必要があります。

クライアントを特定する情報を"クライアント識別子"と呼びます。サーバプログラムはクライアントからの要求を受信するたびに、クライアント識別子獲得APIによりコンポーネントトランザクションサービスからクライアント識別子を獲得できます。

各言語ごとのクライアント識別子獲得APIの使用方法については、"リファレンスマニュアル（API編）"を参照してください。

なお、セションごとにセション情報域を獲得する場合は、クライアント識別子の獲得は行いません。セションIDは、オペレーションのパラメタとして持ち回ります。

(4)セション情報域の生成

セション情報管理を使用するサーバオブジェクトは、セションの開始となる要求を受信した場合に、その要求のクライアント識別子またはセションIDを使用して、セション情報管理にセション情報域の獲得を依頼します。

セション情報域の生成は、セション情報管理のcreate_info()オペレーション（クライアント識別子の場合）またはcreate_info2()オペレーション（セションIDの場合）を使用して行います。各言語での使用方法を以下に示します。

注意

以降の使用方法で示すプログラミング例は、簡単化のため、エラー処理は省略しています。SMOの各オペレーションの呼び出しはCORBAアプリケーションのオペレーション呼び出しとなるため、復帰値の判定の前に例外処理が必要です。CORBAのクライアントアプリケーションでの例外処理については、"アプリケーション作成ガイド(CORBAサービス編)"のクライアントアプリケーションの例外処理を参照してください。

以降のセションIDを使用する場合のすべてのプログラミング例において、オペレーション名は例として示しているだけで、IDLを規定するものではありません。

セションIDを使用する場合

C言語

```
long MOD1_INTF1_ope1(..., ISTD_SMO_SessionId SessionId, ...) {

CORBA_Environment       CoENV;
CORBA_Object            CoOBJ;          /* SMO の OR                    */
CORBA_unsigned_long     SlotNo;         /* スロット番号                 */
CORBA_unsigned_long     size;           /* セション情報域長             */
CORBA_unsigned_long     lifetime;       /* 未使用時間監視間隔           */
int                     ret;
long                    i;


SlotNo    = 1;                          /* スロット番号の設定           */
size      = 8;                          /* セション情報域長の設定       */
lifetime  = 60;                         /* 未使用時間監視間隔の設定     */

/* セション情報域の生成 */
i = ISTD_SMO_create_info2( (ISTD_SMO)CoOBJ,
                           SessionId,
                           SlotNo,
                           size,
                           lifetime,
                           &CoENV);
```

C++言語

```
long MOD1_INTF1_impl::ope1(..., ISTD::SMO::SessionId SessionId, ...) {

CORBA::Environment_ptr  env;
ISTD::SMO_ptr           smo;            // SMOのOR
CORBA::Ulong            SlotNo;         // スロット番号
CORBA::Ulong            size;           // セション情報域長
```

```
    CORBA::Ulong            lifetime;        // 未使用時間監視間隔

    CORBA::Ulong            I;

    CORBA::Octet            *buf;

    long                    length;


    SlotNo    = 1;                            // スロット番号の設定
    size      = 8;                            // セション情報域長の設定
    lifetime  = 60;                           // 未使用時間監視間隔の設定

    // セション情報域の生成
    I = smo->create_info2(SessionId,
                          SlotNo,
                          size,
                          lifetime,
                          *env);
```

COBOL Windows32 Solaris32

```
     WORKING-STORAGE SECTION.
      COPY CONST IN CORBA.

      01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY OBJ-EXT.

      01 RETCODE PIC S9(9) COMP-5.
      01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY SLOTNO.
      01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY SIZE1.
      01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY LIFETIME.
      01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY COENV.

    LINKAGE SECTION.
      01 SESSID.
        02 FILLER OCCURS 48.
          03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY ISTD-SMO-SESSIONID-V.

      01 COPY LONG IN CORBA REPLACING CORBA-LONG BY RET.
      01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY ENV.

    PROCEDURE DIVISION USING
                        ...
                        SESSID
                        ...
                        ENV
                        RET.
    MAIN.

   * --- スロット番号 ---
        COMPUTE SLOTNO = 1.
   * --- セション情報域の大きさ ---
        COMPUTE SIZE1 = 8.
   * --- 時間監視 ---
        COMPUTE LIFETIME = 600.

   * --- セション情報域の生成 ---

        CALL "ISTD-SMO-CREATE-INFO2" USING
          OBJ-EXT
          SESSID
```

```
        SLOTNO
        SIZE1
        LIFETIME
        COENV
        RETCODE
```

クライアント識別子を使用する場合

C言語

```
CORBA_Environment       CoENV;
CORBA_Object            CoOBJ;          /* SMO の OR                 */
ISTD_SMO_ClientId       *ClientId;      /* クライアント識別子        */
CORBA_unsigned_long     SlotNo;         /* スロット番号              */
CORBA_unsigned_long     size;           /* セション情報域長          */
CORBA_unsigned_long     lifetime;       /* 未使用時間監視間隔        */
int                     ret;

long                    i;

/* クライアント識別子の取得 */
ClientId = ISTD_SMO_ClientId_alloc();    /* クライアント識別子領域の獲得 */
ClientId->_maximum = ClientId->_length = ISTD_SMO_ClientIdLen;

ClientId->_buffer  = ISTD_SMO_ClientId_allocbuf( ISTD_SMO_ClientIdLen );
memset( ClientId->_buffer, 0x00, 48 );

ret = TD_get_client_id(ClientId);       /* クライアント識別子獲得      */

SlotNo    = 1;                          /* スロット番号の設定          */
size      = 8;                          /* セション情報域長の設定      */
lifetime  = 60;                         /* 未使用時間監視間隔の設定    */


/* セション情報域の生成 */


i = ISTD_SMO_create_info( (ISTD_SMO)CoOBJ,
                          ClientId,
                          SlotNo,
                          size,
                          lifetime,
                          &CoENV);
```

C++言語

```
CORBA::Environment_ptr  env;
ISTD::SMO_ptr           smo;            // SMOのOR
ISTD::SMO::ClientId     *ClientId;      // クライアント識別子
CORBA::Ulong            SlotNo;         // スロット番号
CORBA::Ulong            size;           // セション情報域長
CORBA::Ulong            lifetime;       // 未使用時間監視間隔
CORBA::Ulong            i;
CORBA::Octet            *buf;
long                    length;

// クライアント識別子の獲得
buf = ISTD::SMO::ClientId::allocbuf(48);   // クライアント識別子型のバッファ獲得
ClientId = new ISTD::SMO::ClientId(48 ,    // クライアント識別子型の獲得
                                   buf,
                                   CORBA_FALSE);
```

```
ret = TD::get_client_id((char *)buf, 48, &length); // クライアント識別子獲得

SlotNo    = 1;                          // スロット番号の設定
size      = 8;                          // セション情報域長の設定
lifetime  = 60;                         // 未使用時間監視間隔の設定

// セション情報域の生成
I = smo->create_info(*ClientId,
                     SlotNo,
                     size,
                     lifetime,
                     *env);
```

COBOL Windows32 Solaris32

```
 WORKING-STORAGE SECTION.
  COPY CONST IN CORBA.

  01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY OBJ-EXT.

  01 ISTD-CLIENTID.
    02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-MAXIMUM.
    02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-LENGTH.
    02 SEQ-BUFFER USAGE IS POINTER.
    02 FILLER OCCURS 48.
      03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY CLIENTID-VALUE.
  01 RETCODE PIC S9(9) COMP-5.
  01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY SLOTNO.
  01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY SIZE1.
  01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY LIFETIME.
  01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY COENV.

PROCEDURE DIVISION USING
MAIN.

* --- クライアント識別子の獲得 ---
    MOVE FUNCTION ADDR(CLIENTID-VALUE(1)) TO SEQ-BUFFER OF ISTD-CLIENTID.
    MOVE 48 TO SEQ-MAXIMUM OF ISTD-CLIENTID.
    CALL "TDGETCLIENTID" USING
      ISTD-CLIENTID
      RETCODE.

* --- スロット番号 ---
    COMPUTE SLOTNO = 1.
* --- セション情報域の大きさ ---
    COMPUTE SIZE1 = 8.
* --- 時間監視 ---
    COMPUTE LIFETIME = 600.

* --- セション情報域の生成 ---
    CALL "ISTD-SMO-CREATE-INFO" USING
      OBJ-EXT
      ISTD-CLIENTID
      SLOTNO
      SIZE1
      LIFETIME
      COENV
      RETCODE.
```

(5)セション情報域への情報の書き込み

セション情報管理を使用するサーバオブジェクトは、セション中の後続のオペレーション呼び出し時まで保持したい情報を、セション情報域に書き込めます。セション情報域への情報の書き込みは、セション情報管理のset_info()オペレーション（クライアント識別子の場合）またはset_info2()オペレーション（セションIDの場合）を使用して行います。各言語での使用方法を以下に示します。

注意

以降の使用方法で示すプログラミング例は、簡単化のため、エラー処理は省略しています。また、セション情報域の生成は完了しているものとします。

セションIDを使用する場合

C言語

```
long MOD1_INTF1_ope1(..., ISTD_SMO_SessionId SessionId, ...) {

CORBA_Environment      CoENV;
CORBA_Object           CoOBJ;           /* SMO の OR                    */
CORBA_unsigned_long    SlotNo;          /* スロット番号                 */
ISTD_SMO_SessionInfo   *data1;          /* セション情報                 */

/* セション情報の作成 */
data1 = ISTD_SMO_SessionInfo_alloc();
data1->_length = data1->_maximum = 8;
data1->_buffer = ISTD_SMO_SessionInfo_allocbuf( data1->_maximum );
data1->_buffer[0] = 1;
SlotNo           = 1;

/* セション情報域へのセション情報の設定 */
ISTD_SMO_set_info2( (ISTD_SMO)CoOBJ,
                    SessionId,
                    SlotNo,
                    data1,
                    &CoENV);
```

C++言語

```
long MOD1_INTF1_impl::ope1(..., ISTD::SMO::SessionId SessionId, ...) {

CORBA::Environment_ptr  env;
ISTD::SMO_ptr           smo;            // SMOのOR
CORBA::ULong            SlotNo;         // スロット番号
ISTD::SMO::SessionInfo  *data1;         // セション情報
CORBA::Octet            *buf;

// スロット番号の設定
SlotNo = 1;

// 設定するセション情報の作成
buf = ISTD::SMO::SessionInfo::allocbuf(8);
*buf = 1;
data1 = new ISTD::SMO::SessionInfo( (CORBA::ULong)8,
                                    (CORBA::ULong)8,
                                    (CORBA::Octet*)buf,
                                    (CORBA::Boolean)CORBA_FALSE );

// セション情報域へのセション情報の設定
smo->set_info2( SessionId,
                SlotNo,
                *data1,
                *env );
```

COBOL Windows32 Solaris32

```
 WORKING-STORAGE SECTION.
  COPY CONST IN CORBA.

  01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY OBJ-EXT.

  01 DATA1.
    02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-MAXIMUM.
    02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-LENGTH.
    02 SEQ-BUFFER USAGE IS POINTER.
    02 FILLER OCCURS 8.
      03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY SEQ-VALUE.

  01 RETCODE PIC S9(9) COMP-5.
  01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY SLOTNO.
  01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY COENV.

 LINKAGE SECTION.
  01 TEMP-SEQ-VALUE.
    02 FILLER OCCURS 8.
      03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY SEQ-VALUE.

  01 SESSID.
    02 FILLER OCCURS 48.
      03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY ISTD-SMO-SESSIONID-V.

  01 COPY LONG IN CORBA REPLACING CORBA-LONG BY RET.
  01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY ENV.

 PROCEDURE DIVISION USING
                    ...
                    SESSID
                    ...
                    ENV

                    RET.

 MAIN.


* --- スロット番号 ---
     COMPUTE SLOTNO = 1.

* --- set_info ---
     MOVE 8 TO SEQ-MAXIMUM OF DATA1
     MOVE 8 TO SEQ-LENGTH OF DATA1

     CALL "CORBA-SEQUENCE-OCTET-ALLOCBUF" USING
       SEQ-LENGTH OF DATA1
       SEQ-BUFFER OF DATA1.

     SET ADDRESS OF TEMP-SEQ-VALUE TO SEQ-BUFFER OF DATA1.
     MOVE 1 TO SEQ-VALUE OF TEMP-SEQ-VALUE(1).

* --- CALL ISTD-SMO-SET-INFO2 ---
     CALL "ISTD-SMO-SET-INFO2" USING
       OBJ-EXT
       SESSID
       SLOTNO
       DATA1
       COENV.
```

クライアント識別子を使用する場合

C言語

```
CORBA_Environment      CoENV;
CORBA_Object           CoOBJ;           /* SMO の OR                */
ISTD_SMO_ClientId      *ClientId;       /* クライアント識別子       */
CORBA_unsigned_long    SlotNo;          /* スロット番号             */
ISTD_SMO_SessionInfo   *data1;          /* セション情報             */

/* セション情報の作成 */
data1 = ISTD_SMO_SessionInfo_alloc();
data1->_length = data1->_maximum = 8;
data1->_buffer = ISTD_SMO_SessionInfo_allocbuf( data1->_maximum );
data1->_buffer[0] = 1;
SlotNo            = 1;

/* セション情報域へのセション情報の設定 */
ISTD_SMO_set_info( (ISTD_SMO)CoOBJ,
                   ClientId,
                   SlotNo,
                   data1,
                   &CoENV);
```

C++言語

```
CORBA::Environment_ptr  env;
ISTD::SMO_ptr           smo;            // SMOのOR
ISTD::SMO::ClientId     *ClientId;      // クライアント識別子
CORBA::ULong            SlotNo;         // スロット番号
ISTD::SMO::SessionInfo  *data1;         // セション情報
CORBA::Octet            *buf;

// スロット番号の設定
SlotNo = 1;

// 設定するセション情報の作成
buf = ISTD::SMO::SessionInfo::allocbuf(8);
*buf = 1;
data1 = new ISTD::SMO::SessionInfo( (CORBA::ULong)8,
                                    (CORBA::ULong)8,
                                    (CORBA::Octet*)buf,
                                    (CORBA::Boolean)CORBA_FALSE );

// セション情報域へのセション情報の設定
smo->set_info( *ClientId,
               SlotNo,
               *data1,
               *env );
```

COBOL Windows32 Solaris32

```
WORKING-STORAGE SECTION.
 COPY CONST IN CORBA.

 01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY OBJ-EXT.

 01 ISTD-CLIENTID.
   02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-MAXIMUM.
   02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-LENGTH.
   02 SEQ-BUFFER USAGE IS POINTER.
   02 FILLER OCCURS 48.
```

```
        03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY CLIENTID-VALUE.
    01 RETCODE PIC S9(9) COMP-5.

    01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY SLOTNO.
    01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY COENV.

    01 DATA1.
      02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-MAXIMUM.
      02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-LENGTH.
      02 SEQ-BUFFER USAGE IS POINTER.
      02 FILLER OCCURS 8.
        03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY SEQ-VALUE.

  LINKAGE SECTION.

    01 TEMP-SEQ-VALUE.
      02 FILLER OCCURS 8.
        03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY SEQ-VALUE.

  PROCEDURE DIVISION USING
  MAIN.

 * --- スロット番号 ---
      COMPUTE SLOTNO = 1.

 * --- set_info ---
      MOVE 8 TO SEQ-MAXIMUM OF DATA1
      MOVE 8 TO SEQ-LENGTH OF DATA1

      CALL "CORBA-SEQUENCE-OCTET-ALLOCBUF" USING
        SEQ-LENGTH OF DATA1
        SEQ-BUFFER OF DATA1.


      SET ADDRESS OF TEMP-SEQ-VALUE TO SEQ-BUFFER OF DATA1.
      MOVE 1 TO SEQ-VALUE OF TEMP-SEQ-VALUE(1).

 * --- CALL ISTD-SMO-SET-INFO ---
      CALL "ISTD-SMO-SET-INFO" USING
        OBJ-EXT
        ISTD-CLIENTID
        SLOTNO
        DATA1
        COENV.
```

(6)セション情報域からの情報の読み込み

セション情報管理を使用するサーバオブジェクトは、以前に呼び出された際にセション情報域に書き込んだセション情報を読み出すことができます。セション情報の読み込みは、セション情報管理のget_info()オペレーション(クライアント識別子の場合)またはget_info2()オペレーション(セションIDの場合)を使用して行います。各言語での使用方法を以下に示します。

注意

以降の使用方法で示すプログラミング例は、簡単化のため、エラー処理は省略しています。また、セション情報域の生成は完了しているものとします。

セションIDを使用する場合

C言語

```
long MOD1_INTF1_ope1(..., ISTD_SMO_SessionId SessionId, ...) {

CORBA_Environment      CoENV;
```

```
CORBA_Object           CoOBJ;          /* SMO の OR                        */
CORBA_unsigned_long    SlotNo;         /* スロット番号                     */
ISTD_SMO_SessionInfo  *data1;          /* セション情報                     */

SlotNo   = 1;

/* セション情報域からのセション情報の取り出し */

ISTD_SMO_get_info2( (ISTD_SMO)CoOBJ,
                    SessionId,
                    SlotNo,
                    &data1,
                    &CoENV);
```

C++言語

```
long MOD1_INTF1_impl::ope1(..., ISTD::SMO::SessionId SessionId, ...) {

CORBA::Environment_ptr  env;
ISTD::SMO_ptr           smo;            // SMOのOR
CORBA::ULong            SlotNo;         // スロット番号
ISTD::SMO::SessionInfo  *data1;         // セション情報

// スロット番号の設定
SlotNo = 1;

// セション情報域からのセション情報の取り出し
smo->get_info2(  SessionId,
                 SlotNo,
                 data1,
                 *env );
```

COBOL Windows32 Solaris32

```
 WORKING-STORAGE SECTION.
  COPY CONST IN CORBA.

  01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY OBJ-EXT.

  01 RETCODE PIC S9(9) COMP-5.

  01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY SLOTNO.
  01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY COENV.

  01 DATA1.
    02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-MAXIMUM.
    02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-LENGTH.
    02 SEQ-BUFFER USAGE IS POINTER.
    02 FILLER OCCURS 8.
      03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY SEQ-VALUE.
  01 DATA2-P USAGE POINTER.

LINKAGE SECTION.
  01 DATA2.
    02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-MAXIMUM.
    02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-LENGTH.
    02 SEQ-BUFFER USAGE IS POINTER.
    02 FILLER OCCURS 100.
      03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY SEQ-VALUE.
  01 TEMP-SEQ-VALUE.
    02 FILLER OCCURS 8.
      03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY SEQ-VALUE.
```

```
  01 SESSID.
    02 FILLER OCCURS 48.
      03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY ISTD-SMO-SESSIONID-V.

  01 COPY LONG IN CORBA REPLACING CORBA-LONG BY RET.
  01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY ENV.

PROCEDURE DIVISION USING
                    ...
                    SESSID
                    ...
                    ENV
                    RET.

MAIN.

* --- スロット番号 ---
    COMPUTE SLOTNO = 1.

    CALL "ISTD-SMO-GET-INFO2" USING
      OBJ-EXT
      SESSID
      SLOTNO
      DATA2-P
      COENV.

    SET ADDRESS OF DATA2 TO DATA2-P
    SET ADDRESS OF TEMP-SEQ-VALUE TO SEQ-BUFFER OF DATA2.
```

クライアント識別子を使用する場合

C言語

```
CORBA_Environment      CoENV;
CORBA_Object           CoOBJ;            /* SMO の OR                    */
ISTD_SMO_ClientId      *ClientId;        /* クライアント識別子           */
CORBA_unsigned_long    SlotNo;           /* スロット番号                 */
ISTD_SMO_SessionInfo   *data1;           /* セション情報                 */

SlotNo   = 1;

/* セション情報域からのセション情報の取り出し */
ISTD_SMO_get_info( (ISTD_SMO)CoOBJ,
                   ClientId,
                   SlotNo,
                   &data1,
                   &CoENV);
```

C++言語

```
CORBA::Environment_ptr  env;
ISTD::SMO_ptr           smo;             // SMOのOR
ISTD::SMO::ClientId     *ClientId;       // クライアント識別子
CORBA::ULong            SlotNo;          // スロット番号
ISTD::SMO::SessionInfo  *data1;          // セション情報

// スロット番号の設定
SlotNo = 1;

// セション情報域からのセション情報の取り出し
smo->get_info( *ClientId,
```

```
                SlotNo,
                data1,
                *env );
```

COBOL Windows32 Solaris32

```
     WORKING-STORAGE SECTION.
      COPY CONST IN CORBA.

      01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY OBJ-EXT.

      01 ISTD-CLIENTID.
        02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-MAXIMUM.
        02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-LENGTH.
        02 SEQ-BUFFER USAGE IS POINTER.
        02 FILLER OCCURS 48.
          03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY CLIENTID-VALUE.
      01 RETCODE PIC S9(9) COMP-5.

      01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY SLOTNO.

      01 DATA1.
        02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-MAXIMUM.
        02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-LENGTH.
        02 SEQ-BUFFER USAGE IS POINTER.
        02 FILLER OCCURS 8.
          03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY SEQ-VALUE.
      01 DATA2-P USAGE POINTER.

    LINKAGE SECTION.
      01 DATA2.
        02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-MAXIMUM.
        02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-LENGTH.
        02 SEQ-BUFFER USAGE IS POINTER.
        02 FILLER OCCURS 100.
          03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY SEQ-VALUE.
      01 TEMP-SEQ-VALUE.
        02 FILLER OCCURS 8.
          03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY SEQ-VALUE.

    PROCEDURE DIVISION USING
    MAIN.

   * --- スロット番号 ---
        COMPUTE SLOTNO = 1.

        CALL "ISTD-SMO-GET-INFO" USING
          OBJ-EXT
          ISTD-CLIENTID
          SLOTNO
          DATA2-P
          COENV.

        SET ADDRESS OF DATA2 TO DATA2-P
        SET ADDRESS OF TEMP-SEQ-VALUE TO SEQ-BUFFER OF DATA2.
```

(7)セション情報域の削除

セション情報管理を使用するサーバオブジェクトは、セション情報域が不要となった時点で、セション情報域を削除します。セション情報域の削除は、セション情報管理のdelete_info()オペレーション（クライアント識別子の場合）またはdelete_info2()オペレーション（セションIDの場合）を使用して行います。各言語での使用方法を以下に示します。

注意

以降の使用方法で示すプログラミング例は、簡単化のため、エラー処理は省略しています。また、セション情報域の生成は完了しているものとします。

セションIDを使用する場合

C言語

```
long MOD1_INTF1_ope1(..., ISTD_SMO_SessionId SessionId, ...) {

CORBA_Environment      CoENV;
CORBA_Object           CoOBJ;           /* SMO の OR                    */
CORBA_unsigned_long    SlotNo;          /* スロット番号                   */

SlotNo   = 1;

/* セション情報域の削除 */
ISTD_SMO_delete_info2( (ISTD_SMO)CoOBJ,
                       SessoinId,
                       SlotNo,
                       &CoENV);
```

C++言語

```
long MOD1_INTF1_impl::ope1(..., ISTD::SMO::SessionId SessionId, ...) {

CORBA::Environment_ptr  env;
ISTD::SMO_ptr           smo;            // SMOのOR
ISTD::SMO::SessionId    *SessionId;     // セションＩＤ
CORBA::ULong            SlotNo;         // スロット番号

// スロット番号の設定
SlotNo = 1;

// セション情報域の削除
smo->delete_info2( SessionId,
                   SlotNo,
                   *env );
```

COBOL Windows32 Solaris32

```
 WORKING-STORAGE SECTION.
  COPY CONST IN CORBA.

  01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY OBJ-EXT.

  01 RETCODE PIC S9(9) COMP-5.

  01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY SLOTNO.
  01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY COENV.

LINKAGE SECTION.
  01 SESSID.
    02 FILLER OCCURS 48.
      03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY ISTD-SMO-SESSIONID-V.

  01 COPY LONG IN CORBA REPLACING CORBA-LONG BY RET.
  01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY ENV.

PROCEDURE DIVISION USING
                     ...
                     SESSID
```

```
                    ...
                    ENV
                    RET.

 MAIN.

* --- スロット番号 ---
    COMPUTE SLOTNO = 1.

* --- セション情報域の削除 ---
    CALL "ISTD-SMO-DELETE-INFO2" USING
      OBJ-EXT
      SESSID
      SLOTNO
      COENV.
```

クライアント識別子を使用する場合

C言語

```
CORBA_Environment      CoENV;
CORBA_Object           CoOBJ;          /* SMO の OR                */
ISTD_SMO_ClientId      *ClientId;      /* クライアント識別子          */
CORBA_unsigned_long    SlotNo;         /* スロット番号               */

SlotNo   = 1;

/* セション情報域の削除 */
ISTD_SMO_delete_info( (ISTD_SMO)CoOBJ,
                      ClientId,
                      SlotNo,
                      &CoENV);
```

C++言語

```
CORBA::Environment_ptr  env;
ISTD::SMO_ptr           smo;           // SMOのOR
ISTD::SMO::ClientId     *ClientId;     // クライアント識別子
CORBA::ULong            SlotNo;        // スロット番号

// スロット番号の設定
SlotNo = 1;

// セション情報域の削除
smo->delete_info( *ClientId,
                  SlotNo,
                  *env );
```

COBOL Windows32 Solaris32

```
WORKING-STORAGE SECTION.
 COPY CONST IN CORBA.

 01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY OBJ-EXT.

 01 ISTD-CLIENTID.
   02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-MAXIMUM.
   02 COPY LONG IN CORBA REPLACING CORBA-LONG BY SEQ-LENGTH.
   02 SEQ-BUFFER USAGE IS POINTER.
   02 FILLER OCCURS 48.
     03 COPY OCTET IN CORBA REPLACING CORBA-OCTET BY CLIENTID-VALUE.
 01 RETCODE PIC S9(9) COMP-5.
```

```
      01 COPY ULONG IN CORBA REPLACING CORBA-UNSIGNED-LONG BY SLOTNO.
      01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY COENV.

    PROCEDURE DIVISION USING
    MAIN.

   * --- スロット番号 ---
        COMPUTE SLOTNO = 1.

   * --- セション情報域の削除 ---
        CALL "ISTD-SMO-DELETE-INFO" USING
          OBJ-EXT
          ISTD-CLIENTID
          SLOTNO
          COENV.
```

5.3.2.3 提供インクルード

セション情報管理を使用するためのインクルードを提供します。

C言語

ISTD_smo.h

C++言語

ISTD_smocpp.h

格納場所は、以下です。

Windows32

INTERSTAGEのインストールフォルダ¥td¥include

Solaris32 Linux32

コンポーネントトランザクションサービスのインストールディレクトリ/usr/include

5.3.2.4 提供ライブラリ

セション情報管理は、セション情報管理オブジェクトの各機能を使用するためのライブラリを提供します。各プログラミング言語のライブラリ名を次に示します。

Solaris32 Linux32

格納場所は、コンポーネントトランザクションサービスのインストールディレクトリ/usr/libです。

Windows32

C言語

F3FMsmo.lib

(F3FMsmosv.lib)

C++言語

F3FMsmocpp.lib

(F3FMsmocppsv.lib)

COBOL

F3FMSMOCBL.lib

(F3FMSMOCBLSV.lib)

格納場所は、以下のフォルダです。

```
INTERSTAGEのインストールフォルダ¥td¥lib
```

注意事項

()内はサーバアプリケーション開発用ライブラリです。CORBAサービスのサーバアプリケーション開発用ライブラリを使用している場合には、()内のライブラリを使用してください。CORBAサービスのライブラリとセション情報管理ライブラリの種類(クライアント/サーバ)が異なる場合、動作は保証されません。

Solaris32

C言語

libtdsmo.so

(libtdsmo_nt.so 注1)

C++言語

libtdsmocpp.so

(libtdsmocpp_nt.so 注1)

libtdsmocpp50.so 注2

(libtdsmocpp50_nt.so 注1 注2)

COBOL

libtdsmocbl.so

(libtdsmocbl_mt.so 注3)

注1

()内はプロセスモードで動作するアプリケーションを作成するためのものです。アプリケーションをプロセスモードで動作させる場合は、()内のライブラリを使用してください。

注2

C++言語のアプリケーションを作成する際に、Sun WorkShop Compilers C++ 5.0またはWS Compilers C++ 6を使用する場合は、libtdsmocpp.so、libtdsmocpp_nt.soのかわりにlibtdsmocpp50.so、libtdsmocpp50_nt.soを使用するようにしてください。

注3

PowerCOBOL97を使用して、マルチスレッドのCOBOL言語のアプリケーションを作成する場合は、libtdsmocbl.soのかわりにlibtdsmocbl_mt.soを使用するようにしてください。

Linux32

C言語

libtdsmo.so

(libtdsmo_nt.so 注1)

C++言語

libtdsmocpp.so

(libtdsmocpp_nt.so 注1)

注1

()内はプロセスモードで動作するアプリケーションを作成するためのものです。アプリケーションをプロセスモードで動作させる場合は、()内のライブラリを使用してください。

5.3.2.5 IDL

セション情報管理が実装するIDLは、次のとおりです。
インタフェースの詳細については、“リファレンスマニュアル(API編)”を参照してください。

```
module ISTD {

    interface SMO {

        // クライアント識別子のデータ型
        const unsigned long     ClientIdLen = 48;            // クライアント識別子長
        typedef sequence<octet,ClientIdLen> ClientId;        // クライアント識別子型

    // セションIDのデータ型

     const unsigned long      SessionIdLen = 48;            // セションID長
     typedef octet            SessionId[SessionIdLen];      // セションID型

        // ユーザ例外処理の定義
        exception InvalidArgument { string reason; };       // パラメタ異常
        exception ProcessFailed   {};                       // 処理異常
        exception AlreadyExist    {};                       // 既登録
        exception NotExist        {};                       // 対象セション情報域なし
        exception NotRegistered   {};                       // 対象リスナ未登録

        // create_info の復帰値
        const unsigned long     TD_OK_NEW   = 0;            // 新規生成
        const unsigned long     TD_OK_OLD   = 1;            // 既存セション情報域返却

        // セション情報域型
        typedef sequence<octet> SessionInfo;

        //
        // --- operations for client id ---
        //
        long create_info(       // セション情報域の獲得
            in  ClientId               CLIENT_ID,
            in  unsigned long          slotno,
            in  unsigned long          size,
            in  unsigned long          lifetime
            ) raises( InvalidArgument, ProcessFailed, AlreadyExist );

        void set_info(          // セション情報域へのデータを書込
            in  ClientId               CLIENT_ID,
            in  unsigned long          slotno,
            in  SessionInfo            data
            ) raises( InvalidArgument, NotExist );

        void get_info(          // セション情報域からのデータを読込
            in  ClientId               CLIENT_ID,
            in  unsigned long          slotno,
            out SessionInfo            data
            ) raises( InvalidArgument, NotExist );

        void delete_info(       // セション情報域の解放
            in  ClientId               CLIENT_ID,
            in  unsigned long          slotno
            ) raises( InvalidArgument, NotExist );

        void add_listener(      // リスナの登録
            in  Object                 LISTENER_OR
            ) raises( InvalidArgument );

        void del_listener(      // リスナの抹消
            in  Object                 LISTENER_OR
            ) raises( InvalidArgument, NotRegistered );

        //
```

```
// --- operatoins for session id ---
//
long create_info2(      // セション情報域の獲得
      in  SessionId              SESSION_ID,
      in  unsigned long          slotno,
      in  unsigned long          size,
      in  unsigned long          lifetime
      ) raises( InvalidArgument, ProcessFailed, AlreadyExist );

void set_info2(         // セション情報域へのデータを書込
      in  SessionId              SESSION_ID,
      in  unsigned long          slotno,
      in  SessionInfo            data
      ) raises( InvalidArgument, NotExist );

void get_info2(         // セション情報域からのデータを読込
      in  SessionId              SESSION_ID,
      in  unsigned long          slotno,
      out SessionInfo            data
      ) raises( InvalidArgument, NotExist );

void delete_info2(      // セション情報域の解放
      in  SessionId              SESSION_ID,
      in  unsigned long          slotno
      ) raises( InvalidArgument, NotExist );

void add_listener2(     // リスナの登録
      in  Object                 LISTENER_OR
      ) raises( InvalidArgument );

void del_listener2(     // リスナの抹消
      in  Object                 LISTENER_OR
      ) raises( InvalidArgument, NotRegistered );

};  // SMO


interface SMO_LISTENER { // 事象通知リスナインタフェース（クライアント識別子用）

    // クライアント識別子のデータ型
    const unsigned long     ClientIdLen = 48;            // クライアント識別子長
    typedef sequence<octet,ClientIdLen> ClientId;        // クライアント識別子型

    // セション情報域型
    typedef sequence<octet> SessionInfo;

    // 事象通知データ型（クライアント識別子用）
    struct SlotInfo {
        ClientId           CLIENT_ID;
        unsigned long      slotno;
        SessionInfo        data;
    };
    typedef sequence<SlotInfo> DelSlotInfo;

    void timeout(          // イベント受信
        in  DelSlotInfo           data
        );

}; // SMO_LISTENER

interface SMO_LISTENER2 { // 事象通知リスナインタフェース（セションID用）

// セションIDのデータ型
```

```
        const unsigned long      SessIdLen = 48;              // セションID長
        typedef octet            SessionId[SessIdLen];        // セションID型

          // セション情報域型
          typedef sequence<octet> SessionInfo;

          // 事象通知データ型 (セションID用)
          struct SlotInfo {
              SessionId          SESSION_ID;
              unsigned long      slotno;
              SessionInfo        data;
          };
          typedef sequence<SlotInfo> DelSlotInfo;

          void timeout(          // イベント受信
              in  DelSlotInfo           data
              );

    }; // SMO_LISTENER2

};
```

注意

本IDLは、セション情報管理のインタフェース情報を示すためのものです。本IDLをIDLコンパイラに適用する必要はありません。また、本IDLをIDLコンパイラに適用して生成されたスタブファイル、およびスケルトンファイルは、セション情報管理との通信には使用できません。セション情報管理のスタブおよびスケルトンは、提供ライブラリに含まれます。

5.3.3 セション情報管理の事象通知リスナオブジェクト

セション情報管理機能で、セション情報域の未使用時間監視を使用した場合に、未使用時間を超過したセション情報域の発生の事象通知をSMOから受け付けるオブジェクト(事象通知リスナオブジェクト)の作成方法について示します。

5.3.3.1 プログラミングの流れ

(1) CORBAサービスの初期化

CORBAサーバアプリケーションとして作成する必要があります。そのため、最初に、CORBAサービスの初期化が必要です。

注意事項 Windows32

プログラミング言語がC言語またはC++言語の場合、CORBAサービスの初期化直後に次の関数を発行する必要があります。

セションIDの場合

ISTD_SMO_LISTENER2_init();

クライアント識別子の場合

ISTD_SMO_LISTENER_init();

(2) 事象通知リスナ実装関数の作成

SMOからの事象通知を受け取るために、以下に示すインタフェース実装関数名およびパラメタ形式でSMO_LISTENERインタフェース(クライアント識別子の場合)またはSMO_LISTENER2インタフェース(セションIDの場合)のtimeout()オペレーションを実装する必要があります。各プログラミング言語での実装形式を示します。
なお、事象通知のデータ形式については、"5.3.2.5 IDL"を参照してください。

セションIDを使用する場合

C言語

```
void
ISTD_SMO_LISTENER2_timeout(
    CORBA_Object                        obj,
    ISTD_SMO_LISTENER2_DelSlotInfo      *slot,
    CORBA_Environment                   *env
    )
{

    事象通知に対する処理を記述してください。

}
```

C++言語

```
void
ISTD_SMO_LISTENER2_impl::timeout(
    const ISTD::SMO_LISTENER2::DelSlotInfo      &slot,
    CORBA::Environment                          &env  )
    throw( CORBA::Exception )
{

    事象通知に対する処理を記述してください。

}
```

COBOL Windows32 Solaris32

```
IDENTIFICATION DIVISION.
PROGRAM-ID. "ISTD-SMO-LISTENER2-TIMEOUT".

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    ARGUMENT-NUMBER IS ARG-C
    ARGUMENT-VALUE  IS ARG-V
    SYMBOLIC CONSTANT
    COPY SYMBOL-CONST IN CORBA.
    .

DATA DIVISION.
 WORKING-STORAGE SECTION.
  COPY CONST IN CORBA.

LINKAGE SECTION.
  01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY OBJ.
  01 DATA-P USAGE POINTER.
  01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY ENV.

PROCEDURE DIVISION USING
  OBJ
  DATA-P
  ENV.

MAIN.

  事象通知に対する処理を記述してください。

MAIN-END.

END PROGRAM  "ISTD-SMO-LISTENER2-TIMEOUT".
```

クライアント識別子を使用する場合

C言語

```
void
ISTD_SMO_LISTENER_timeout(
    CORBA_Object                    obj,
    ISTD_SMO_LISTENER_DelSlotInfo   *slot,
    CORBA_Environment               *env
    )
{

    事象通知に対する処理を記述してください。

}
```

C++言語

```
void
ISTD_SMO_LISTENER_impl::timeout(
    const ISTD::SMO_LISTENER::DelSlotInfo     &slot,
    CORBA::Environment                        &env  )
    throw( CORBA::Exception )
{

    事象通知に対する処理を記述してください。

}
```

COBOL Windows32 Solaris32

```
IDENTIFICATION DIVISION.
PROGRAM-ID. "ISTD-SMO-LISTENER-TIMEOUT".

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    ARGUMENT-NUMBER IS ARG-C
    ARGUMENT-VALUE  IS ARG-V
    SYMBOLIC CONSTANT
    COPY SYMBOL-CONST IN CORBA.
    .

DATA DIVISION.
 WORKING-STORAGE SECTION.
  COPY CONST IN CORBA.

LINKAGE SECTION.
  01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY OBJ.
  01 DATA-P USAGE POINTER.
  01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY ENV.

PROCEDURE DIVISION USING
  OBJ
  DATA-P
  ENV.

MAIN.
```

```
    事象通知に対する処理を記述してください。

   MAIN-END.

   END PROGRAM  "ISTD-SMO-LISTENER-TIMEOUT".
```

(3) 事象通知リスナの登録

SMOから事象通知を受け取るためには、事象通知を受け取るリスナアプリケーションのオブジェクトリファレンスを、SMOに登録する必要があります。リスナアプリケーションの登録は、SMOのadd_listener()オペレーション(クライアント識別子の場合)またはadd_listener2()オペレーション(セションIDの場合)を使用して行います。

各プログラミング言語での使用方法を以下に示します。

セションIDを使用する場合

C言語

```
    CORBA_Environment       CoENV;
    CORBA_Object            CoOBJ;    /* SMO の OR          */
    CORBA_Object            LiOBJ;    /* リスナ の OR       */

    ISTD_SMO_add_listener2( CoOBJ,
                            LiOBJ,
                            &CoEnv );
```

C++言語

```
    CORBA::Environment_ptr  env;
    ISTD::SMO_ptr           smo;           // SMO の OR
    CORBA::Object*          obj_listener;  // リスナ の OR

    smo->add_listener2(  obj_listener,
                         *env);
```

COBOL Windows32 Solaris32

```
  WORKING-STORAGE SECTION.
   COPY CONST IN CORBA.
    01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY COOBJ.
    01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY LSNOBJ.
    01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY COENV.

  PROCEDURE DIVISION USING
  MAIN.

      CALL "ISTD-SMO-ADD-LISTENER2" USING
        COOBJ
        LSNOBJ
        COENV.
```

クライアント識別子を使用する場合

C言語

```
    CORBA_Environment       CoENV;
    CORBA_Object            CoOBJ;    /* SMO の OR          */
    CORBA_Object            LiOBJ;    /* リスナ の OR       */

    ISTD_SMO_add_listener( CoOBJ,
```

```
                                   LiOBJ,
                                   &CoEnv );
```

C++言語

```
            CORBA::Environment_ptr  env;
            ISTD::SMO_ptr           smo;            // SMO の OR
            CORBA::Object*          obj_listener;   // リスナ の OR

            smo->add_listener(  obj_listener,
                                *env);
```

COBOL Windows32 Solaris32

```
          WORKING-STORAGE SECTION.
           COPY CONST IN CORBA.
            01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY COOBJ.
            01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY LSNOBJ.

            01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY COENV.

          PROCEDURE DIVISION USING

          MAIN.

              CALL "ISTD-SMO-ADD-LISTENER" USING
                COOBJ
                LSNOBJ
                COENV.
```

(4) 事象通知リスナの登録抹消

SMOからの事象通知受け取りを停止する場合、SMOに登録したリスナアプリケーションのオブジェクトリファレンスの登録抹消を行う必要があります。リスナアプリケーションの登録抹消はSMOのdel_listener()オペレーション(クライアント識別子の場合)またはdel_listener2()オペレーション(セションIDの場合)を使用して行います。

各プログラミング言語での使用方法を以下に示します。

セションIDを使用する場合

C言語

```
            CORBA_Environment       CoENV;
            CORBA_Object            CoOBJ;    /* SMO の OR          */
            CORBA_Object            LiOBJ;    /* リスナ の OR       */

            ISTD_SMO_del_listener2( CoOBJ,
                                    LiOBJ,
                                    &CoEnv );
```

C++言語

```
            CORBA::Environment_ptr  env;
            ISTD::SMO_ptr           smo;            // SMO の OR
            CORBA::Object*          obj_listener;   // リスナ の OR

            smo->del_listener2(  obj_listener,
                                 *env);
```

COBOL Windows32 Solaris32

```
WORKING-STORAGE SECTION.
 COPY CONST IN CORBA.
  01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY COOBJ.
  01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY LSNOBJ.
  01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY COENV.

PROCEDURE DIVISION USING
MAIN.

    CALL "ISTD-SMO-DEL-LISTENER2" USING
      COOBJ
      LSNOBJ
      COENV.
```

クライアント識別子を使用する場合

C言語

```
CORBA_Environment       CoENV;
CORBA_Object            CoOBJ;    /* SMO の OR          */
CORBA_Object            LiOBJ;    /* リスナ の OR       */

ISTD_SMO_del_listener( CoOBJ,
                       LiOBJ,
                       &CoEnv );
```

C++言語

```
CORBA::Environment_ptr  env;
ISTD::SMO_ptr           smo;           // SMO の OR
CORBA::Object*          obj_listener;  // リスナ の OR

smo->del_listener(  obj_listener,
                    *env);
```

COBOL Windows32 Solaris32

```
WORKING-STORAGE SECTION.
 COPY CONST IN CORBA.
  01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY COOBJ.
  01 COPY OBJECT IN CORBA REPLACING CORBA-OBJECT BY LSNOBJ.
  01 COPY ENVIRONMENT IN CORBA REPLACING CORBA-ENVIRONMENT BY COENV.

PROCEDURE DIVISION USING
MAIN.

    CALL "ISTD-SMO-DEL-LISTENER" USING
      COOBJ
      LSNOBJ
      COENV.
```

5.3.3.2 事象通知リスナ作成に使用する提供インクルード

事象通知リスナを実装するプログラムでは、以下のインクルードファイルをインクルードしてください。()内は、事象通知リスナの登録および登録抹消を行う場合に必要なインクルードです。

C言語

セションIDの場合

ISTD_smo_listener2.h

(ISTD_smo.h)

クライアント識別子の場合

ISTD_smo_listener.h

(ISTD_smo.h)

C++言語

セションIDの場合

ISTD_smo_listener2cpp.h

(ISTD_smocpp.h)

クライアント識別子の場合

ISTD_smo_listenercpp.h

(ISTD_smocpp.h)

格納場所は、以下です。

Windows32

INTERSTAGEのインストールフォルダ¥td¥include

Solaris32 Linux32

コンポーネントトランザクションサービスのインストールディレクトリ/usr/include

5.3.3.3 事象通知リスナ作成に使用する提供ライブラリ

事象通知リスナを実装するためには、作成したプログラムに以下の提供ライブラリをリンクする必要があります。
()内のライブラリは、事象通知リスナのSMOへの登録および登録抹消を行う場合にリンクする必要があります。

Windows32

C言語

セションIDの場合

F3FMsmolistener2.lib

(F3FMsmo.lib/F3FMsmosv.lib)

クライアント識別子の場合

F3FMsmolistener.lib

(F3FMsmo.lib/F3FMsmosv.lib)

C++言語

セションIDの場合

F3FMsmolistener2cpp.lib

(F3FMsmocpp.lib/F3FMsmocppsv.lib)

クライアント識別子の場合

F3FMsmolistenercpp.lib

(F3FMsmocpp.lib/F3FMsmocppsv.lib)

COBOL

セションIDの場合

F3FMsmolistener2cbl.lib

(F3FMSMOCBL.lib/F3FMSMOCBLSV.lib)

クライアント識別子の場合

F3FMsmolistenercbl.lib

(F3FMSMOCBL.lib/F3FMSMOCBLSV.lib)

格納場所は、以下のフォルダです。

INTERSTAGEのインストールフォルダ¥td¥lib

Solaris32

C言語

セションIDの場合

libtdsmolistener2.so/libtdsmolistener2_nt.so 注1

(libtdsmo.so/libtdsmo_nt.so 注1)

クライアント識別子の場合

libtdsmolistener.so/libtdsmolistener_nt.so 注1

(libtdsmo.so/libtdsmo_nt.so 注1)

C++言語

セションIDの場合

libtdsmolistener2cpp.so/libtdsmolistener2cpp_nt.so 注1

(libtdsmocpp.so/libtdsmocpp_nt.so 注1)

libtdsmolistener2cpp50.so/libtdsmolistener2cpp50_nt.so 注1、注2

(libtdsmocpp50.so/libtdsmocpp50_nt.so 注1、注2)

クライアント識別子の場合

libtdsmolistenercpp.so/libtdsmolistenercpp_nt.so 注1

(libtdsmocpp.so/libtdsmocpp_nt.so 注1)

libtdsmolistenercpp50.so/libtdsmolistenercpp50_nt.so 注1、注2

(libtdsmocpp50.so/libtdsmocpp50_nt.so 注1、注2)

COBOL

セションIDの場合

libtdsmolistener2cbl.so/libtdsmolistener2cbl_mt.so 注3

(libtdsmocbl.so/libtdsmocbl_mt.so) 注3

クライアント識別子の場合

libtdsmolistenercbl.so/libtdsmolistenercbl_mt.so 注3

(libtdsmocbl.so/libtdsmocbl_mt.so) 注3

注1

ファイル名に“_nt”の付いたライブラリは、プロセスモードで動作するアプリケーションを作成するためのものです。アプリケーションをプロセスモードで動作させる場合は、ファイル名に“_nt”の付いたライブラリを使用してください。

注2

C++言語のアプリケーションを作成する際に、Sun WorkShop Compilers C++ 5.0またはWS Compilers C++ 6を使用する場合は、libtdsmolistener2cpp.so、libtdsmolistener2cpp_nt.so、libtdsmolistenercpp.so、libtdsmolistenercpp_nt.so、libtdsmocpp.so、libtdsmocpp_nt.soのかわりにlibtdsmolistener2cpp50.so、libtdsmolistener2cpp50_nt.so、libtdsmolistenercpp50.so、libtdsmolistenercpp50_nt.so、libtdsmocpp50.so、libtdsmocpp50_nt.soを使用するようにしてください。

注3

PowerCOBOL97を使用して、マルチスレッドのCOBOL言語のアプリケーションを作成する場合は、libtdsmolistener2cbl.so、libtdsmolistenercbl.so、libtdsmocbl.soのかわりにlibtdsmolistener2cbl_mt.so、libtdsmolistenercbl_mt.so、libtdsmocbl_mt.soを使用するようにしてください。

Linux32

C言語

セッションIDの場合

libtdsmolistener2.so/libtdsmolistener2_nt.so 注1

(libtdsmo.so/libtdsmo_nt.so 注1)

クライアント識別子の場合

libtdsmolistener.so/libtdsmolistener_nt.so 注1

(libtdsmo.so/libtdsmo_nt.so 注1)

C++言語

セッションIDの場合

libtdsmolistener2cpp.so/libtdsmolistener2cpp_nt.so 注1

(libtdsmocpp.so/libtdsmocpp_nt.so 注1)

クライアント識別子の場合

libtdsmolistenercpp.so/libtdsmolistenercpp_nt.so 注1

(libtdsmocpp.so/libtdsmocpp_nt.so 注1)

注1

ファイル名に"_nt"の付いたライブラリは、プロセスモードで動作するアプリケーションを作成するためのものです。アプリケーションをプロセスモードで動作させる場合は、ファイル名に"_nt"の付いたライブラリを使用してください。

5.3.3.4 COBOLを使用する場合の注意事項

COBOLにより事象通知リスナを作成する場合、以下の点に注意してください。
なお、COBOLはWindows(R)版、Solaris版のみ使用できます。

(1) 事象通知リスナ実装関数のコンパイル方法

Windows32

COBOLのサーバアプリケーションはダイナミックリンクライブラリ(dll)の形態で利用されます。事象通知リスナのdllの名称は以下のようにしてください。

セッションIDの場合

ISTD-SMO-LISTENER2.dll

クライアント識別子の場合

ISTD-SMO-LISTENER.dll

事象通知リスナ実装関数をコンパイルする場合、コンパイルオプションにNAMEオプションを指定してコンパイルしてください。ここで作成されたオブジェクトと提供ライブラリ(F3FMsmolistener2cbl.libまたはF3FMsmolistenercbl.lib)をリンクすることでdllを作成することができます。

また、dll作成の際にはモジュール定義ファイルが必要です。モジュール定義ファイルのひな型(F3FMsmolistener2cbl.defおよびF3FMsmolistenercbl.def)は、以下のフォルダに格納されています。

```
INTERSTAGEのインストールフォルダ¥td¥lib
```

Solaris32

COBOLのサーバアプリケーションはsoライブラリの形態で利用されます。事象通知リスナのsoライブラリの名称は以下のようにしてください。

セションIDの場合

libISTD-SMO-LISTENER2.so

クライアント識別子の場合

libISTD-SMO-LISTENER.so

コンパイル方法

セションIDの場合

```
cobol -G -lcobol -L$OD_HOME/lib -lOMcbl -L$TD_HOME/usr/lib
      -ltdsmolistener2cbl -o libISTD-SMO-LISTENER2.so
      ISTD_SMO_LISTENER2.cbl
```

クライアント識別子の場合

```
cobol -G -lcobol -L$OD_HOME/lib -lOMcbl -L$TD_HOME/usr/lib
      -ltdsmolistenercbl -o libISTD-SMO-LISTENER.so
      ISTD_SMO_LISTENER.cbl
```

$OD_HOME:CORBAサービスのインストールディレクトリ
$TD_HOME:コンポーネントトランザクションサービスのインストールディレクトリ

(2) OD_impl_instでの登録

OD_impl_instで指定する定義ファイルに以下の指定が必要です。

セションIDの場合

Windows32

```
rep_id                       = IDL:ISTD/SMO_LISTENER2:1.0
IDL:ISTD/SMO_LISTENER2:1.0   = ISTD-SMO-LISTENER2.dll
```

Solaris32

```
rep_id                       = IDL:ISTD/SMO_LISTENER2:1.0
IDL:ISTD/SMO_LISTENER2:1.0 = /user/libISTD-SMO-LISTENER2.so
```

クライアント識別子の場合

Windows32

```
rep_id                       = IDL:ISTD/SMO_LISTENER:1.0
IDL:ISTD/SMO_LISTENER2:1.0   = ISTD-SMO-LISTENER.dll
```

Solaris32

```
rep_id                      = IDL:ISTD/SMO_LISTENER:1.0
IDL:ISTD/SMO_LISTENER:1.0 = /user/libISTD-SMO-LISTENER.so
```

(3) 実行時の注意事項

COBOLアプリケーションはノンスレッドプログラムだけが可能となります。

Windows32

マルチスレッドプログラムは使用できません。

Solaris32

このため、LD_LIBRARY_PATHに、CORBAサービスのノンスレッドアプリケーション用のライブラリのパスを指定する必要があります。ノンスレッド用のライブラリについては、"アプリケーション作成ガイド(CORBAサービス編)"を参照ください。

5.3.4 セション情報管理の注意事項

セション情報管理を使用したトランザクションアプリケーションを構築する場合、次のことに注意してください。

(1) クライアント識別子についての注意事項

クライアント識別子の識別単位

セション情報管理を使用するトランザクションアプリケーションは、各オペレーションの呼び出しごとに、呼び出し元のクライアントを一意に識別できるクライアント識別子が取得できます。
このクライアント識別子は、クライアントオブジェクトを実装したプロセス単位の識別子です。1プロセスで、複数のクライアント側のオブジェクトを実装している場合は、複数のオブジェクトからのオペレーション呼び出しに対して、同一のクライアント識別子が通知されます。
したがって、セション情報管理機能を使用する、単一のセション型業務のクライアントオブジェクトは、1つのプロセス内に実装するように作成してください。

クライアント識別子の中継

サーバ側で、クライアントからの要求が、複数のトランザクションアプリケーションを中継されて処理される場合、クライアントからの要求を直接受け付けるトランザクションアプリケーションで、クライアント識別子をコンポーネントトランザクションサービスから受け取り、以降のトランザクションアプリケーションにユーザデータとして伝搬するように作成してください。

クライアント識別子の有効期間

クライアントとサーバ間で通信エラー(クライアント側にCOMM_FAILURE例外で通知される)が発生した場合、当該クライアント識別子は無効となります。
また、クライアントとサーバ間の無通信時間が一定時間以上となると、クライアント識別子は無効となります。無通信時間監視時間は、デフォルトで600秒です。この値は、CORBAサービス環境定義ファイルの以下の定義項目を設定することでカストマイズ可能です。

```
Period_idle_con_timeout
```

この定義項目は、デフォルトが120です。この定義項目と、period_tick(デフォルト値は、5です)を乗算した値が、無通信監視時間となります。なお、0を指定すると無通信監視は行われません。

(2) セション情報域の有効範囲

セション情報域は、SMOによりメモリ上で管理される領域です。したがって、セション情報域は、獲得後、以下の範囲で有効です。

- 明示的なセション情報域の削除まで
- Interstageの停止まで
- 未使用時間監視によるセション情報域の削除まで

(3) 事象通知リスナの有効範囲

コンポーネントトランザクションサービスを再起動する場合、事象通知リスナは、コンポーネントトランザクションサービスの再起動後に、SMOに再度登録する必要があります。

第6章 C++言語の提供クラス

tdcコマンドによりC++言語のスケルトンを生成した場合、IDL定義はトランザクションアプリケーション用ヘッダファイルにclassとしてマッピングされます。

ここでは、IDL定義を元に生成されるクラス、およびコンポーネントトランザクションサービスが提供するクラスについて、以下に示す内容で説明します。

6.1 IDL定義を元に生成されるクラス

クラスマッピングおよびIDL定義に構造データ型が記述されている場合のマッピング形態について、以下に示す内容で説明します。

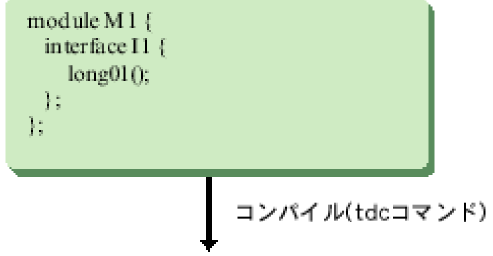
6.1.1 クラスマッピング

IDL定義を元に生成されるクラスを、以下に示します。

- モジュールクラス
- インタフェースクラス
- インプリメンテーションクラス

module宣言、およびinterface宣言はそれぞれclassにマッピングされます。オペレーションは、インタフェースクラスの仮想メンバ関数、およびインプリメンテーションクラスのメンバ関数としてマッピングされます。マッピングの例、および各クラスの概要を以下に示します。

IDL定義ファイル



トランザクションサーバアプリケーション用ヘッダファイル

```
class M1 {
    class I1:public virtual TD{
        ・・・
        virtual CORBA::Long01()=0;
        ・・・
    };
};

class M1_I1_impl:public M1::I1
{
        ・・・
        CORBA::Long01();
        ・・・
} ;
```

インタフェースクラス
モジュールクラス
インプリメンテーションクラス

モジュールクラス

モジュールクラスには、そのコンポーネント内にインタフェースクラスが実装されます。また、IDL定義のmodule宣言で記述されたモジュール名が、モジュールクラス名となります。

インタフェースクラス

インタフェースクラスは、TDクラスを継承し、IDL定義に記述されたオペレーションが仮想メンバ関数（図中のvirtual CORBA::Long O1()）として実装されます。IDL定義に記述されたデータ型によっては、このクラスのコンポーネント内にデータ型のクラス、および、メンバ関数が生成されます。また、IDL定義のinterface宣言で記述されたインタフェース名が、インタフェースクラス名となります。

インプリメンテーションクラス

インプリメンテーションクラスは、インタフェース実装関数（図中のCORBA::Long O1()）をメンバ関数に持つクラスです。このクラスは、インタフェースクラスを継承し、オブジェクト単位に生成されます。アプリケーションは、このインタフェース実装関数として作成します。インプリメンテーションクラス名は、"モジュール名_インタフェース名_impl"となります。

IDL定義に基づいてクラスが生成されますが、module宣言、interface宣言のネスト構造、および変数の有効範囲等も、IDL定義に基づいてマッピングされます。module宣言がネストしている場合、およびinterface宣言が継承している場合のトランザクションアプリケーション用ヘッダファイルを、以下に示します。

[module宣言がネストしている場合]

IDL定義ファイル

```
module M1 {
  interface I1 {
      long01();
  };
  module M2 {
    interface I2 {
        long02();
    };
  };
};
```

トランザクションサーバアプリケーション用ヘッダファイル

```
class M1 {
   class I1 :public virtual TD{
       ・・・
       virtual CORBA::Long01()=0;
       ・・・
   };
   class M2{
       class I2 :public virtual TD {
            ・・・
            virtual CORBA::Long02();
            ・・・
            };
   };
};
```

[継承している場合]

IDL定義ファイル

```
module M1 {
  interface I1 {
      long O1();
  };
  module M2 {
    interface I2:M1::I1 {
        long O2();
    };
  };
};
```

トランザクションサーバアプリケーション用ヘッダファイル

```
class M1 {
   class I1 :public virtual TD{
       ・・・
       virtual CORBA::Long O1()=0;
       ・・・
   };
   class M2{
     class I2 :public virtual M1::I1,
          public virtual TD {
            ・・・
            virtual CORBA::Long O2()=0;
            ・・・
            };
   };
};
```

[継承している場合(**#include**ディレクティブを使用)]

IDL定義ファイル

```
module M1 {
  interface I1 {
      long01();
  };
};
```

inheritance.idl

```
#include"base.idl"
module M2{
  interface I2:M1::I1{
      long02();
  };
};
```

トランザクションサーバアプリケーション用ヘッダファイル

```
class M1 {
    class I1 :public virtual TD {
        ・・・
        virtual CORBA::Long01()=0;
        ・・・
    };
};

class M2{
    class I2 :public virtual M1::I1,
        public virtual TD {
        ・・・
        virtual CORBA::Long02()=0;
        ・・・
    };
};
```

6.1.2 構造データ型のマッピング形態

IDL定義で構造データ型を記述した場合、データ操作用のクラスおよびメンバ関数が生成されます。

構造データ型には、以下の3つがあります。

- 構造体型
- 配列型
- シーケンス型

6.1.2.1 構造体型

(1)マッピング

IDL言語で構造体型structを指定した場合、C++言語でもstructでデータを宣言します。

IDL定義ファイル

```
module M1 {
  interface I1 {
    struct FStruct{
          long para1;
          long para2;
    }
    struct VStruct{
          long para3;
          string para4;
    };
      ・・・
  };
};
```

トランザクションサーバアプリケーション用ヘッダファイル

```
class M1 {
    class I1:public virtual TD {
        struct FStruct{
                CORBA::Long para1;
                CORBA::Long para2;
        }
        struct VStruct{
                CORBA::Long para3;
                TD::String_var para4;
        };
        ・・・
        };
};
```

- **IDL定義**

```
module  M1{
    interface I1{
        struct  f_STRUCT {
            long        ef1;
            long        ef2;
        };
```

```
        struct  v_STRUCT {
            long        ev1;
            string      ev2;
        };

        long OPE1 (
            in      f_STRUCT  para1,
            out     f_STRUCT  para2,
            inout   f_STRUCT  para3,
            in      v_STRUCT  para4,
            out     v_STRUCT  para5,
            inout   v_STRUCT  para6);
    };
};
```

・トランザクションアプリケーション用ヘッダファイル

```
class   M1
{
public:
    class   I1 : public virtual TD
    {
    public:
        struct f_STRUCT{
            CORBA::Long ef1;
            CORBA::Long ef2;
        };

        class f_STRUCT_var
        {
        public:
            f_STRUCT_var();
            f_STRUCT_var( f_STRUCT* );
            f_STRUCT_var( const f_STRUCT & );
            f_STRUCT_var( const f_STRUCT_var & );
            ~f_STRUCT_var();
            f_STRUCT_var &operator=( f_STRUCT * );
            f_STRUCT_var &operator=( const f_STRUCT & );
            f_STRUCT_var &operator=( const f_STRUCT_var & );
            f_STRUCT *operator->() const;
            operator f_STRUCT*() const;
            protected:
            f_STRUCT    *_ptr;
        };

        struct v_STRUCT{
            CORBA::Long ev1;
            TD::String_var  ev2;
        };

        class v_STRUCT_var
        {
        public:
            v_STRUCT_var();
            v_STRUCT_var( v_STRUCT* );
            v_STRUCT_var( const v_STRUCT & );
            v_STRUCT_var( const v_STRUCT_var & );
            ~v_STRUCT_var();
            v_STRUCT_var &operator=( v_STRUCT * );
            v_STRUCT_var &operator=( const v_STRUCT & );
            v_STRUCT_var &operator=( const v_STRUCT_var & );
            v_STRUCT *operator->() const;
            operator v_STRUCT*() const;
            protected:
```

```
            v_STRUCT    *_ptr;
        };
        ・・・
    };
    ・・・
};
```

構造体を宣言すると、上記のように“構造体名_var”クラスが生成されます。

(2)メンバ変数/メンバ関数

・ 構造体のポインタ(*_ptr)

・ デフォルトコンストラクタ
(M1::I1::f_STRUCT_var::f_STRUCT_var()／M1::I1::v_STRUCT_var::v_STRUCT_var())

インスタンス生成時、新規T*データを作成し初期化します。

(使用例)

```
// STRUCT_var型のインスタンス宣言
M1::I1::f_STRUCT_var fstr_v;
M1::I1::v_STRUCT_var *vstr_v = new M1::I1::v_STRUCT_var;
// いずれも暗黙のうちにデフォルトコンストラクタが呼ばれる
```

・ **T***コンストラクタ
(M1::I1::f_STRUCT_var::f_STRUCT_var(f_STRUCT *)／M1::I1::v_STRUCT_var::v_STRUCT_var(v_STRUCT *))

インスタンス生成時、パラメタで指定されたTポインタをメンバ変数_ptrに設定します。

(使用例)

```
M1::I1::f_STRUCT_var *fstr_v1 = new M1::I1::f_STRUCT_var;
// fstr_v1を使用した処理
・・・
M1::I1::f_STRUCT_var fstr_v2(fstr_v1);
// T*コンストラクタ
```

・ コピーコンストラクタ
(M1::I1::f_STRUCT_var::f_STRUCT_var(const f_STRUCT &)／M1::I1::v_STRUCT_var::v_STRUCT_var (const v_STRUCT &))

インスタンス生成時、指定されたパラメタの_ptrのデータのコピーを作成し、自身の_ptrに設定します。

(使用例)

```
M1::I1::f_STRUCT fstr_v1;
// fstr_v1を使用した処理
・・・
M1::I1::f_STRUCT_var fstr_v2(fstr_v1);
// コピーコンストラクタ
```

・ コピーコンストラクタ**(var型)**
(M1::I1::f_STRUCT_var::f_STRUCT_var(const f_STRUCT_var &)／M1::I1::v_STRUCT_var::v_STRUCT_var (const v_STRUCT_var &))

インスタンス生成時、指定されたパラメタの_ptrのデータのコピーを作成し、自身の_ptrに設定します。

(使用例)

```
M1::I1::f_STRUCT_var fstr_v1;
// fstr_v1を使用した処理
```

```
・・・
M1::I1::f_STRUCT_var fstr_v2(fstr_v1);
// コピーコンストラクタ
```

- デストラクタ
（ M1::I1::f_STRUCT_var::~f_STRUCT_var ()／M1::I1::v_STRUCT_var::~v_STRUCT_var() ）

インスタンス破壊時、_ptrの領域を解放します。

- **T***代入演算子
（ M1::I1::f_STRUCT_var & M1::I1::f_STRUCT_var::operator=(f_STRUCT *) ／ M1::I1::v_STRUCT_var & M1::I1::v_STRUCT_var::operator=(v_STRUCT *) ）

（使用例）

```
M1::I1::f_STRUCT *fstr_1 = new M1::I1::f_STRUCT;
// fstr_1を使用した処理
・・・
M1::I1::f_STRUCT_var fstr_v2;
fstr_v2 = fstr_1;
// T*代入演算子
```

- 代入演算子
（ M1::I1::f_STRUCT_var & M1::I1::f_STRUCT_var::operator=(const f_STRUCT &) ／ M1::I1::v_STRUCT_var &M1::I1::v_STRUCT_var::operator=(const v_STRUCT &) ）

（使用例）

```
const M1::I1::f_STRUCT *fstr_1 = new M1::I1::f_STRUCT;
// fstr_1を使用した処理
・・・
M1::I1::f_STRUCT_var fstr_v2 = fstr_1;
// 代入演算子
```

- 代入演算子**(var**型**)**
（ M1::I1::f_STRUCT_var &M1::I1::f_STRUCT_var::operator=(const f_STRUCT_var &) ／ M1::I1::v_STRUCT_var &M1::I1::v_STRUCT_var::operator=(const M1::I1::v_STRUCT_var &) ）

（使用例）

```
M1::I1::f_STRUCT *fstr_1 = new M1::I1::f_STRUCT;
// fstr_1を使用した処理
・・・
M1::I1::f_STRUCT_var fstr_v2;
fstr_v2 = fstr_1;

M1::I1::f_STRUCT_var fstr_v3 = fstr_v2;
// 代入演算子
```

- 交換演算子
（ M1::I1::f_STRUCT_var::operator M1::I1::f_STRUCT*() const／M1::I1::v_STRUCT_var::operator M1::I1::v_STRUCT*() const ）

（使用例）

```
M1::I1::f_STRUCT *fstr_1 = new M1::I1::f_STRUCT;
// fstr_1を使用した処理
・・・

M1::I1::f_STRUCT_var fstr_v2;
fstr_v2 = fstr_1;
```

```
M1::I1::f_STRUCT_var *fstr_v3 = (M1::I1::f_STRUCT*)fstr_v2;
// 交換演算子
```

- ポインタ演算子
  （ M1::I1::f_STRUCT *M1::I1::f_STRUCT_var::operator->() const／M1::I1::v_STRUCT *M1::I1::v_STRUCT_var::operator->() const ）

  （使用例）

```
M1::I1::f_STRUCT *fstr_1 = new M1::I1::f_STRUCT;
fstr_1->ef1 = 1;
fstr_1->ef2 = 2;

M1::I1::f_STRUCT_var fstr_v2;
fstr_v2 = fstr_1;

fstr_v2->ef1 = 3;
fstr_v2->ef2 = 4;
// ポインタ演算子
```

(3)領域獲得/解放

固定長データ

- inパラメタ …領域を獲得、および解放するための特別な関数を使用する必要はありません。
- outパラメタ …領域を獲得、および解放するための特別な関数を使用する必要はありません。
- inoutパラメタ…領域を獲得、および解放するための特別な関数を使用する必要はありません。

可変長データ

- inパラメタ …領域を獲得、および解放するための特別な関数を使用する必要はありません。
- outパラメタ …構造体の領域はC++言語のnew演算子、可変長のデータ領域は可変長データ域獲得関数(TD::string_alloc())によって行います。なお、ここで獲得した領域はスケルトンにより解放されます。
- inoutパラメタ…領域を獲得、および解放するための特別な関数を使用する必要はありません

6.1.2.2 配列型

(1)マッピング

IDL言語で配列を指定した場合、C++言語でも配列でデータを宣言します。

- **IDL定義**

```
module M1 {
        interface I1 {
        typedef  long    FARRAY[10];
        typedef  string  VARRAY[10];
           ・・・
        };
};
```

- トランザクションアプリケーション用ヘッダファイル

```
class   M1
{
public:
    class   I1 : public virtual TD
    {
    public:
```

```
        typedef CORBA::Long FARRAY[10];
        typedef CORBA::Long FARRAY_slice;

        class FARRAY_var
        {
        public:
            FARRAY_var();
            FARRAY_var( FARRAY_slice* );
            FARRAY_var( const FARRAY_var & );
            ~FARRAY_var();
            FARRAY_var &operator=( FARRAY_slice* );
            FARRAY_var &operator=( const FARRAY_var & );

            operator FARRAY_slice*() const;
            FARRAY_slice &operator[]( CORBA::ULong );
            FARRAY_slice &operator[]( CORBA::ULong ) const;
            protected:
            FARRAY_slice *_ptr;
        };

        static FARRAY_slice *FARRAY_alloc();
        static void FARRAY_free( FARRAY_slice * );

        typedef TD::String_var VARRAY[10];
        typedef TD::String_var VARRAY_slice;

        class M1::I1::VARRAY_var
        {
        public:
            VARRAY_var();
            VARRAY_var( VARRAY_slice* );
            VARRAY_var( const VARRAY_var & );
            ~VARRAY_var();
            VARRAY_var &operator=( VARRAY_slice* );
            VARRAY_var &operator=( const VARRAY_var & );

            operator VARRAY_slice*() const;
            VARRAY_slice &operator[]( CORBA::ULong );
            VARRAY_slice &operator[]( CORBA::ULong ) const;
            protected:
            VARRAY_slice *_ptr;
        };

        static VARRAY_slice *VARRAY_alloc();
        static void VARRAY_free( VARRAY_slice * );
         ・・・
    };
     ・・・
};
```

(2)メンバ変数/メンバ関数

- 配列領域獲得関数
  (M1::I1::FARRAY_slice * M1::I1::FARRAY_alloc()／ M1::I1::VARRAY_slice * M1::I1::VARRAY_alloc())

  IDL定義で記述された配列の領域を獲得します。

  （使用例）

```
M1::I1::FARRAY_slice *fix = M1::I1::FARRAY_alloc();
M1::I1::VARRAY_slice *var = M1::I1::VARRAY_alloc();
```

・ 配列領域解放関数
(void M1::I1::FARRAY_free(M1::I1::FARRAY_slice *)／ void M1::I1::VARRAY_free(M1::I1::VARRAY_slice *))
配列領域獲得関数、および、配列要素複写関数で獲得した領域を解放します。

（使用例）

```
M1::I1::FARRAY_slice *fix = M1::I1::FARRAY_alloc();
M1::I1::VARRAY_slice *var = M1::I1::VARRAY_alloc();
        // fix,varを使用した処理
・・・
M1::I1::FARRAY_free(fix);
M1::I1::VARRAY_free(var);
```

(3)配列_varクラスのメンバ変数/メンバ関数

配列を宣言すると、“配列名_var”クラスが生成されます。

・ 配列のポインタ（*_ptr）

・ デフォルトコンストラクタ
(M1::I1::FARRAY_var::FARRAY_var()／M1::I1::VARRAY_var::VARRAY_var())

インスタンス生成時、新規T*データを作成し初期化します。

（使用例）

```
// 配列_var型のインスタンス宣言
M1::I1::FARRAY_var farr_v;
M1::I1::FARRAY_var *farr_v = new M1::I1::FARRAY_var;
// いずれも暗黙のうちにデフォルトコンストラクタが呼ばれる
```

・ **T***コンストラクタ
(M1::I1::FARRAY_var::FARRAY_var(M1::I1::FARRAY_slice*) ／ M1::I1::VARRAY_var::VARRAY_var(M1::I1::VARRAY_slice *))

インスタンス生成時、パラメタで指定されたTポインタをメンバ変数_ptrに設定します。

（使用例）

```
M1::I1::FARRAY_var *farr_v1 = new M1::I1::FARRAY_var;
// farr_v1を使用した処理
・・・
M1::I1::FARRAY_var farr_v2(farr_v1);
// T*コンストラクタ
```

・ コピーコンストラクタ
(M1::I1::FARRAY_var::FARRAY_var(const M1::I1::FARRAY_var &)／M1::I1::VARRAY_var::VARRAY_var(const M1::I1::VARRAY_var &))

インスタンス生成時、指定されたパラメタの_ptrのデータのコピーを作成し、自身の_ptrに設定します。

（使用例）

```
M1::I1::FARRAY_var farr_v1;
// farr_v1を使用した処理
・・・
M1::I1::FARRAY_var farr_v2(farr_v1);
// コピーコンストラクタ
```

・ デストラクタ
(M1::I1::FARRAY_var::~FARRAY_var()／M1::I1::VARRAY_var::~VARRAY_var())

インスタンス破壊時、_ptrの領域を解放します。

- **T***代入演算子
（ M1::I1::FARRAY_var &M1::I1::FARRAY_var::operator=(M1::I1::FARRAY_slice *)／ M1::I1::VARRAY_var &M1::I1::VARRAY_var::operator=(M1::I1::VARRAY_slice *)）

（使用例）

```
M1::I1::FARRAY *farr_1 = new M1::I1::FARRAY;
// farr_1を使用した処理
・・・
M1::I1::FARRAY_var farr_v2;
farr_v2 = farr_1;
// T*代入演算子
```

- 代入演算子
（ M1::I1::FARRAY_var &M1::I1::FARRAY_var::operator=(const FARRAY_var &)／M1::I1::VARRAY_var &M1::I1::VARRAY_var::operator=(const VARRAY_var &)）

（使用例）

```
M1::I1::FARRAY *farr_1 = new M1::I1::FARRAY;
// farr_1を使用した処理
・・・
M1::I1::FARRAY_var farr_v2;
farr_v2 = farr_1;

M1::I1::FARRAY_var farr_v3 = farr_v2;
// 代入演算子
```

- 交換演算子
（M1::I1::FARRAY_var::operator M1::I1::FARRAY_slice *() const／M1::I1::VARRAY_var::operator M1::I1::VARRAY_slice *() const）

（使用例）

```
M1::I1::FARRAY *farr_1 = new M1::I1::FARRAY;
// farr_1を使用した処理
・・・

M1::I1::FARRAY_var farr_v2;
farr_v2 = farr_1;

M1::I1::FARRAY_var *farr_v3 = (M1::I1::FARRAY*)farr_v2;
// 交換演算子
```

- 添字演算子
（ M1::I1::FARRAY_slice &M1::I1::FARRAY_var::operator[](CORBA::ULong index)／M1::I1::VARRAY_slice &M1::I1::VARRAY_var::operator[](CORBA::ULong index)）

（使用例）

```
M1::I1::FARRAY_slice *farr_1;
farr_1 = FARRAY_alloc();
// farr_1を使用した処理
・・・
M1::I1::FARRAY_var farr_v1;
farr_v1 = farr_1;
long d = varr_1[0];    // 交換演算子
```

・ 添字演算子(constあり)
(M1::I1::FARRAY_slice &M1::I1::FARRAY_var::operator[](CORBA::ULong) const ／ M1::I1::VARRAY_slice &M1::I1::VARRAY_var::operator[](CORBA::ULong) const

(使用例)

```
M1::I1::FARRAY_slice *farr_1;
farr_1 = FARRAY_alloc();
// farr_1を使用した処理
・・・
const M1::I1::FARRAY_var farr_v1;
farr_v1 = farr_1;
const long d = varr_1[0];    // 添字演算子
```

(4)領域獲得/解放

固定長データ

- inパラメタ …領域を獲得、および解放するための特別な関数を使用する必要はありません。
- outパラメタ …領域を獲得、および解放するための特別な関数を使用する必要はありません。
- inoutパラメタ…領域を獲得、および解放するための特別な関数を使用する必要はありません。

可変長データ

- inパラメタ …領域を獲得、および解放するための特別な関数を使用する必要はありません。
- outパラメタ …配列領域獲得関数によりデータ域の獲得を行います。なお、ここで獲得した領域はスケルトンにより解放されます。
- inoutパラメタ…領域を獲得、および解放するための特別な関数を使用する必要はありません。

6.1.2.3 シーケンス型

(1)マッピング

IDL言語でシーケンス型sequenceを指定した場合、C++言語では_maximum(最大長)、_length(シーケンス長)、_buffer(バッファポインタ)をprivateデータに持ったC++のクラスにマッピングします。

IDL定義

```
module M1 {
        interface I1 {
                typedef sequence <long> SequenceLong;
・・・
        };
    interface I2 {
        typedef sequence <string> SequenceStr;
        ・・・
    };
    interface I3 {
        typedef sequence <string,10> BSequenceStr;
        ・・・
    };

};
```

トランザクションアプリケーション用ヘッダファイル

```
class   M1
{
public:
    class   I1 : public virtual TD
    {
```

```
public:
    class SequenceLong
    {
    public:
        SequenceLong();
        SequenceLong( CORBA::ULong );
        SequenceLong( CORBA::ULong max, CORBA::ULong length,
             CORBA::Long *data, CORBA::Boolean release = CORBA_TRUE );
        SequenceLong( const SequenceLong &s );
        ~SequenceLong();

        static CORBA::Long *allocbuf( CORBA::ULong );
        static void freebuf( CORBA::Long* );

        SequenceLong &operator=( const SequenceLong &s );

        CORBA::ULong maximum() const;

        void length( CORBA::ULong );
        CORBA::ULong length() const;

        CORBA::Long &operator[]( CORBA::ULong index );
        const CORBA::Long &operator[]( CORBA::ULong index ) const;

    private:

        CORBA::ULong _maximum;
        CORBA::ULong _length;
        CORBA::Long *_buffer;
        CORBA::Boolean _release;
        CORBA::Long wkrtn;
    };

    class SequenceLong_var
    {
    public:
        SequenceLong_var();
        SequenceLong_var( SequenceLong_ptr );
        SequenceLong_var( const SequenceLong_var & );
        ~SequenceLong_var();

        SequenceLong_var &operator=( SequenceLong_ptr );
        SequenceLong_var &operator=( const SequenceLong_var & );

        operator SequenceLong_ptr() const;
        operator SequenceLong_ptr&();
        SequenceLong_ptr operator->() const;

        CORBA::Long &operator[]( CORBA::ULong );
        const CORBA::Long &operator[]( CORBA::ULong ) const;
        protected:
        SequenceLong    *_ptr;
        CORBA::Long wkrtn;
    };
};
class   I2 : public virtual TD
{
public:
    class SequenceStr
    {
    public:
        SequenceStr();
        SequenceStr( CORBA::ULong );
```

```
        SequenceStr( CORBA::ULong max, CORBA::ULong length,
            TD::String_var *data, CORBA::Boolean release = CORBA_TRUE );
        SequenceStr( const SequenceStr &s );
        ~SequenceStr();

        static TD::String_var *allocbuf( CORBA::ULong );
        static void freebuf( TD::String_var* );

        SequenceStr &operator=( const SequenceStr &s );

        CORBA::ULong maximum() const;

        void length( CORBA::ULong );
        CORBA::ULong length() const;


        TD::String_var &operator[]( CORBA::ULong index );

        const TD::String_var &operator[]( CORBA::ULong index ) const;


    private:

        CORBA::ULong _maximum;
        CORBA::ULong _length;
        TD::String_var *_buffer;
        CORBA::Boolean _release;
        TD::String_var *wkrtn;
    };

    class SequenceStr_var
    {

    public:
        SequenceStr_var();
        SequenceStr_var( SequenceStr_ptr );
        SequenceStr_var( const SequenceStr_var & );
        ~SequenceStr_var();

        SequenceStr_var &operator=( SequenceStr_ptr );
        SequenceStr_var &operator=( const SequenceStr_var & );

        operator SequenceStr_ptr() const;
        operator SequenceStr_ptr&();
        SequenceStr_ptr operator->() const;

        TD::String_var &operator[]( CORBA::ULong );
        const TD::String_var &operator[]( CORBA::ULong ) const;
        protected:
        SequenceStr *_ptr;
        TD::String_var *wkrtn;
    };
};
class   I3 : public virtual TD
{
public:
    class BSequenceStr
    {
    public:
        BSequenceStr();
        BSequenceStr( CORBA::ULong length, TD::String_var *data,
        CORBA::Boolean release = CORBA_TRUE );
        BSequenceStr( const BSequenceStr &s );
```

```
            ~BSequenceStr();

            static TD::String_var *allocbuf( CORBA::ULong );
            static void freebuf( TD::String_var* );

            BSequenceStr &operator=( const BSequenceStr &s );

            CORBA::ULong maximum() const;

            void length( CORBA::ULong );
            CORBA::ULong length() const;

            TD::String_var &operator[]( CORBA::ULong index );
            const TD::String_var &operator[]( CORBA::ULong index ) const;

        private:
            const CORBA::ULong _maximum;
            CORBA::ULong _length;
            TD::String_var *_buffer;
            CORBA::Boolean _release;
            TD::String_var *wkrtn;
        };

        class BSequenceStr_var
        {
        public:
            BSequenceStr_var();
            BSequenceStr_var( BSequenceStr_ptr );
            BSequenceStr_var( const BSequenceStr_var & );
            ~BSequenceStr_var();

            BSequenceStr_var &operator=( BSequenceStr_ptr );
            BSequenceStr_var &operator=( const BSequenceStr_var & );

            operator BSequenceStr_ptr() const;
            operator BSequenceStr_ptr&();
            BSequenceStr_ptr operator->() const;

            TD::String_var &operator[]( CORBA::ULong );
            const TD::String_var &operator[]( CORBA::ULong ) const;
            protected:
            BSequenceStr    *_ptr;
            TD::String_var *wkrtn;
        };
        ・・・
    };
    ・・・
};
```

(2)メンバ変数/メンバ関数

・ 配列の最大個数(_maximum)

・ 使用する配列の個数(_length)

・ 配列の値(_buffer)

・ リリースフラグ(_release)
未サポート。CORBA_TRUEだけが設定可能です。

・ デフォルトコンストラクタ
（M1::I1::SequenceLong::SequenceLong() ／ M1::I2::SequenceStr::SequenceStr() ／ M1::I3::BSequenceStr::BSequenceStr()）

インスタンス生成時、シーケンス長_lengthを0で初期化します。また、サイズ指定ありのシーケンスの場合、最大長_maximumは0で初期化し、最大値が指定されたバウンディッド・シーケンスの場合、最大長_maximumは指定された最大値を設定します。

（使用例）

```
// 固定長データ
M1::I1::SequenceLong *seq1 = new M1::I1::SequenceLong;
              // デフォルトコンストラクタSequenceLongが呼ばれ
              // seq1が初期化されます。

// 可変長データ
M1::I2::SequenceStr *seq2 = new M1::I2::SequenceStr;
               // デフォルトコンストラクタSequenceStrが呼ばれ
               // seq2が初期化されます。

// 可変長データ（最大値が指定されたバウンディッド・シーケンス）
M1::I3::BSequenceStr *seq3 = new M1::I3::BSequenceStr;

              // デフォルトコンストラクタBSequenceStrが呼ばれ
              // seq3が初期化されます。
```

・ **maximum**コンストラクタ
(M1::I1::SequenceLong::SequenceLong(CORBA::ULong max) ／ M1::I2::SequenceStr::SequenceStr(CORBA::ULong max))

インスタンス生成時、サイズ指定なしのシーケンスの場合、最大長_maximumを0で初期化します。

（使用例）

```
// 固定長データ
M1::I1::SequenceLong *seq1 = new M1::I1::SequenceLong(10);
           // maximumコンストラクタが呼ばれ
           // seq1の最大長が10に設定される

// 可変長データ
M1::I2::SequenceStr *seq2 = new M1::I2::SequenceStr(11);
           // maximumコンストラクタが呼ばれ
           // seq1の最大長が11に設定される
```

・ **T *data**コンストラクタ
(M1::I1::SequenceLong::SequenceLong(CORBA::ULong max, CORBA::ULong length, CORBA::Long *data, CORBA::Boolean release) ／ M1::I2::SequenceStr::SequenceStr(CORBA::ULong max, CORBA::ULong length, TD::String_var *data, CORBA::Boolean release) ／ M1::I3::BSequenceStr::BSequenceStr(CORBA::ULong length, TD::String_var *data, CORBA::Boolean release))

インスタンス生成時、サイズ指定あり/なしの両方において、パラメタで指定したシーケンス長、最大長およびバッファポインタを_length、_maximum、_bufferに設定します。ただし、最大値が指定されたバウンディッド・シーケンスの場合、_maximumは指定された最大値を設定します。

（使用例）

```
// 固定長データ
long data[] = {1,2,3,4,5};
M1::I1::SequenceLong  seq1(10,5,data,CORBA_TRUE);
      // T ＊dataコンストラクタが呼ばれ、
      // パラメタで指定した値で初期化される

// 可変長データ
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I2::SequenceStr::allocbuf(4) ;
```

```
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I2::SequenceStr seq1(10,4,wkdata,CORBA_TRUE);
      // T *dataコンストラクタが呼ばれ、
      // パラメタで指定した値で初期化される

// 可変長データ（最大値が指定されたバウンディッド・シーケンス）
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I3::BSequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I3::BSequenceStr seq1(4,wkdata,CORBA_TRUE);
      // T *dataコンストラクタが呼ばれ、
      // パラメタで指定した値で初期化される
```

- コピーコンストラクタ
  (M1::I1::SequenceLong::SequenceLong(const SequenceLong &) ／ M1::I2::SequenceStr::SequenceStr(const SequenceStr &) ／ M1::I3::BSequenceStr::BSequenceStr(const BSequenceStr &))

  インスタンス生成時、パラメタで指定された値を_length、_maximum、_bufferに設定します。ただし、最大値が指定されたバウンディッド・シーケンスの場合、_maximumは指定された最大値を設定します。

  （使用例）

```
// 固定長データ
long data[] = {1,2,3,4,5};
M1::I1::SequenceLong s1(10,5,data,CORBA_TRUE);
M1::I1::SequenceLong s2(s1);
    // コピーコンストラクタが呼ばれ、s1のデータがs2にコピーされる

// 可変長データ
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I2::SequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I2::SequenceStr seq1(10,4,wkdata,CORBA_TRUE);
M1::I2::SequenceStr seq2(seq1);
    // コピーコンストラクタが呼ばれ、seq1のデータがseq2にコピーされる

// 可変長データ（最大値が指定されたバウンディッド・シーケンス）
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I3::BSequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I3::BSequenceStr seq1(4,wkdata,CORBA_TRUE);
M1::I3::BSequenceStr seq2(seq1);
    // コピーコンストラクタが呼ばれ、seq1のデータがseq2にコピーされる
```

- デストラクタ
  (M1::I1::SequenceLong::~SequenceLong() ／ M1::I2::SequenceStr_var::~SequenceStr() ／ M1::I3::BSequenceStr::~BSequenceStr())

  インスタンス破壊時、_bufferの値に対する領域を解放します。

- 配列領域獲得関数
  (CORBA::Long * M1::I1::SequenceLong::allocbuf(CORBA::ULong) ／ TD::String_var * M1::I2::SequenceStr::allocbuf(CORBA::ULong) ／ TD::String_var * M1::I3::BSequenceStr::allocbuf(CORBA::ULong))

T *dataコンストラクタに渡すことができるシーケンスの要素の配列を割り当てます。

（使用例）

```
// 固定長データ
CORBA::Long *data1 = M1::I1::SequenceLong::allocbuf(5);
//data1を使用する処理
                 ･
M1::I1::SequenceLong::freebuf(data1);
        // 獲得した配列の領域を解放する

// 可変長データ
TD::String_var *data2 = M1::I2::SequenceStr::allocbuf(4);
//data2を使用する処理
                 ･
M1::I2::SequenceStr::freebuf(data2);
        // 獲得した配列の領域を解放する

// 可変長データ（最大値が指定されたバウンディッド・シーケンス）
TD::String_var *data3 = M1::I3::BSequenceStr::allocbuf(4);
      // ４つの配列領域を獲得する
```

- 配列領域解放関数
（static void M1::I1::SequenceLong::freebuf(CORBA::Long*) ／ static void M1::I2::SequenceStr::freebuf(TD::String_var *) ／ M1::I3::BSequenceStr::freebuf(TD::String_var *)）

allocbuf関数によって割り当てられた配列を解放します。

（使用例）

```
// 固定長データ
CORBA::Long *data1 = M1::I1::SequenceLong::allocbuf(5);
//data1を使用する処理

M1::I1::SequenceLong::freebuf(data1);

        // 獲得した配列の領域を解放する

// 可変長データ
TD::String_var *data2 = M1::I2::SequenceStr::allocbuf(4);
//data2を使用する処理

M1::I2::SequenceStr::freebuf(data2);
        // 獲得した配列の領域を解放する

// 可変長データ（最大値が指定されたバウンディッド・シーケンス）
TD::String_var *data3 = M1::I3::BSequenceStr::allocbuf(4);
//data2を使用する処理

M1::I3::BSequenceStr::freebuf(data3);
        // 獲得した配列の領域を解放する
```

- 代入演算子
（ M1::I1::SequenceLong &M1::I1::SequenceLong::operator=(const SequenceLong &) ／ M1::I2::SequenceStr &M1::I2::SequenceStr::operator(const SequenceStr &) ／ M1::I3::BSequenceStr &M1::I3::BSequenceStr::operator=(const BSequenceStr &)）

パラメタで指定された値を_bufferに設定します。

（使用例）

```
// 固定長データ
```

```
long data[] = {1,2,3,4,5};
M1::I1::SequenceLong s1(10,5,data,CORBA_TRUE);
M1::I1::SequenceLong s2;
s2 = s1;  // 代入演算子が呼ばれ、s1のデータがs2にコピーされる

// 可変長データ
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I2::SequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I2::SequenceStr seq1(10,4,wkdata,CORBA_TRUE);
M1::I2::SequenceStr seq2;
seq2 = seq1;   // 代入演算子が呼ばれ、seq1のデータがseq2にコピーされる

// 可変長データ (最大値が指定されたバウンディッド・シーケンス)
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I3::BSequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I3::BSequenceStr seq1(4,wkdata,CORBA_TRUE);
M1::I3::BSequenceStr seq2;
seq2 = seq1;   // 代入演算子が呼ばれ、seq1のデータがseq2にコピーされる
```

- **maximum()**アクセス関数(**maximum**獲得)
  (CORBA::ULong M1::I1::SequenceLong::maximum() const ／ CORBA::ULong M1::I2::SequenceStr::maximum() const ／ CORBA::ULongM1::I3::BSequenceStr::maximum() const)

  サイズ指定のないシーケンスの場合、現在使用可能なバッファの総数を返します。サイズが指定されたシーケンスの場合、IDLで与えられたシーケンスの大きさを返します。

  (使用例)

```
// 固定長データ
CORBA::ULong max;
M1::I1::SequenceLong s1(5);
max = s1.maximum();  // maxの値は５

// 可変長データ
M1::I2::SequenceStr s2(4);
max = s2.maximum() ;  // maxの値は４

// 可変長データ (最大値が指定されたバウンディッド・シーケンス)
M1::I3::BSequenceStr s3;  //デフォルトコンストラクタで最大値に10が設定される
max = s3.maximum() ;  // maxの値は10
```

- **length()**アクセス関数(**length**設定)
  (void M1::I1::SequenceLong::length(CORBA::ULong) ／ void M1::I2::SequenceStr::length(CORBA::ULong) ／ void M1::I3::BSequenceStr::length(CORBA::ULong))

  length(Ulong)関数はパラメタで指定した値をシーケンス長に設定します。

  (使用例)

```
// 固定長データ
long data[] = {1,2,3,4,5};
M1::I1::SequenceLong s1(10,5,data,CORBA_TRUE);
s1.length(5);  //シーケンス長を5に設定

// 可変長データ
char *data[] = {"sunday","monday","tuesday","wednesday"};
```

```
TD::String_var *wkdata = M1::I2::SequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I2::SequenceStr seq1(10,4,wkdata,CORBA_TRUE);
seq1.length(4) ; //シーケンス長を4に設定

// 可変長データ (最大値が指定されたバウンディッド・シーケンス)
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I3::BSequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I3::BSequenceStr seq1(4,wkdata,CORBA_TRUE);
seq1.length(4) ; //シーケンス長を4に設定
```

- **length()アクセス関数(length獲得)**
  (CORBA::ULong M1::I1::SequenceLong::length() const ／ CORBA::ULong M1::I2::SequenceStr::length() const ／ CORBA::ULong M1::I3::BSequenceStr::length() const

  シーケンス長を返します

  (使用例)

```
// 固定長データ
CORBA::ULong len;
long data[] = {1,2,3,4,5};
M1::I1::SequenceLong s1(10,5,data,CORBA_TRUE);
len = s1.length();  // lenの値は５

// 可変長データ
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I2::SequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I2::SequenceStr seq1(10,4,wkdata,CORBA_TRUE);
len = seq1.length() ;  // lenの値は４

// 可変長データ (最大値が指定されたバウンディッド・シーケンス)
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I3::BSequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I3::BSequenceStr seq1(4,wkdata,CORBA_TRUE);
len = seq1.length() ;  // lenの値は４
```

- 添字演算子
  (CORBA::Long & M1::I1::SequenceLong::operator[](CORBA::ULong) ／ M1::I2::SequenceStr::operator[](CORBA::ULong) ／ TD::String_var &M1::I3::BSequenceStr::operator[](CORBA::ULong))

  与えられたインデックスに対応する_bufferの内容を返します。

  (使用例)

```
// 固定長データ
long data[] = {1,2,3,4,5};
M1::I1::SequenceLong s1(10,5,data,CORBA_TRUE);
long  l = s1[3];  // lの値は４

// 可変長データ
char *data[] = {"sunday","monday","tuesday","wednesday"};
```

```
TD::String_var *wkdata = M1::I2::SequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I2::SequenceStr seq1(10,4,wkdata,CORBA_TRUE);
TD::String_var  d = seq1[3];   // dの値はwednesday

// 可変長データ (最大値が指定されたバウンディッド・シーケンス)
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I3::BSequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I3::BSequenceStr seq1(4,wkdata,CORBA_TRUE);
TD::String_var  d = seq1[3];   // dの値はwednesday
```

- 添字演算子(**const**あり)
  (const CORBA::Long &M1::I1::SequenceLong::operator[](CORBA::ULong) const ／ const TD::String_var &M1::I2::SequenceStr::operator[](CORBA::ULong) const ／ const TD::String_var & M1::I3::BSequenceStr::operator[] (CORBA::ULong) const)

  与えられたインデックスに対応する_bufferの内容を返す。

  (使用例)

```
// 固定長データ
long data[] = {1,2,3,4,5};
M1::I1::SequenceLong s1(10,5,data,CORBA_TRUE);
const M1::I1::SequenceLong s2(s1);
long l = s2[3];  // lの値は4

// 可変長データ
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I2::SequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I2::SequenceStr seq1(10,4,wkdata,CORBA_TRUE);
const M1::I2::SequenceStr seq2(seq1);

const TD::String_var d = seq2[3];  // dの値はwednesday


// 可変長データ (最大値が指定されたバウンディッド・シーケンス)
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I3::BSequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I3::BSequenceStr seq1(4,wkdata,CORBA_TRUE);
const M1::I3::BSequenceStr seq2(seq1);
const TD::String_var d = seq2[3];  // dの値はwednesday
```

(3) シーケンス_varクラスのメンバ変数/メンバ関数

シーケンスを宣言すると、"シーケンス名_var"クラスが生成されます。

- シーケンスクラスのポインタ(*_ptr)

- デフォルトコンストラクタ
  (M1::I1::SequenceLong_var::SequenceLong_var() ／ M1::I2::SequenceStr_var::SequenceStr_var() ／

M1::I3::BSequenceStr_var::BSequenceStr_var() ）

インスタンス生成時、新規T*データを作成し初期化します。

（使用例）

```
// Sequence_var型のインスタンス宣言
M1::I1::SequenceLong_var seq1;
M1::I1::SequenceLong_var *seq2 = new M1::I1::SequenceLong_var;
// いずれも暗黙のうちにデフォルトコンストラクタが呼ばれる
```

- **T***コンストラクタ
（ M1::I1::SequenceLong_var::SequenceLong_var(SequenceLong_ptr *) ／ M1::I2::SequenceStr_var::SequenceStr_var(SequenceStr_ptr *)) ／ M1::I3::BSequenceStr_var::BSequenceStr_var(BSequenceStr_ptr *) ）

インスタンス生成時、パラメタで指定されたTポインタをメンバ変数_ptrに設定します。

（使用例）

```
M1::I1::SequenceLong_var *seq1 = new M1::I1::SequenceLong_var;
// seq1を使用した処理
・・・
M1::I1::SequenceLong_var seq2(seq1);
// T*コンストラクタ
```

- コピーコンストラクタ
（ M1::I1::SequenceLong_var::SequenceLong_var(const SequenceLong_var &) ／ M1::I2::SequenceStr_var::SequenceStr_var(const SequenceStr_var &) ／ M1::I3::BSequenceStr_var(const BSequenceStr_var &) ）

インスタンス生成時、指定されたパラメタの_ptrのデータのコピーを作成し、自身の_ptrに設定します。

（使用例）

```
M1::I1::SequenceLong_var seq1;
// seq1を使用した処理
・・・
M1::I1::SequenceLong_var seq2(seq1);
// コピーコンストラクタ
```

- デストラクタ
（ M1::I1::SequenceLong_var::~SequenceLong_var() ／ M1::I2::SequenceStr_var::~SequenceStr_var() ／ M1::I3::BSequenceStr_var::~BSequenceStr_var()）

インスタンス破壊時、_ptrの領域を解放します。

- **T***代入演算子
（ M1::I1::SequenceLong_var &M1::I1::SequenceLong_var::operator=(SequenceLong_ptr *) ／ M1::I2::SequenceStr_var &M1::I2::SequenceStr_var::operator=(SequenceStr_ptr *) ／ M1::I3::BSequenceStr_var &M1::I3::BSequenceStr_var::operator=(BSequenceStr_ptr *) ）

（使用例）

```
M1::I1::SequenceLong *seq1 = new M1::I1::SequenceLong;
// seq1を使用した処理
・・・
M1::I1::SequenceLong_var seq2;
seq2 = seq1;
// T*代入演算子
```

- 代入演算子
（ M1::I1::SequenceLong_var &M1::I1::SequenceLong_var::operator=(const SequenceLong_var &) ／ M1::I2::SequenceStr_var

&M1::I2::SequenceStr_var::operator=(const SequenceStr_var &) ／ M1::I3::BSequenceStr_var &M1::I3::BSequenceStr_var::operator=(const BSequenceStr_var &) ）

（使用例）

```
M1::I1::SequenceLong *seq1 = new M1::I1::SequenceLong;
// seq1を使用した処理
・・・
M1::I1::SequenceLong_var seq2;
seq2 = seq1;

M1::I1::SequenceLong_var seq3 = seq2;
// 代入演算子
```

- ポインタ演算子
  （ M1::I1::SequenceLong *M1::I1::SequenceLong::operator->() const ／ M1::I2::SequenceStr *M1::I2::SequenceStr::operator->() const ／ M1::I3::BSequenceStr *M1::I3::BSequenceStr::operator->() const ）

（使用例）

```
M1::I1::SequenceLong *seq1 = new M1::I1::SequenceLong;

seq1->ef1 = 1;
seq1->ef2 = 2;

M1::I1::SequenceLong_var seq_2;
seq2 = seq1;
seq2->ef1 = 3;
seq2->ef2 = 4;
// ポインタ演算子
```

- 交換演算子
  （ operator M1::I1::SequenceLong_ptr *() const ／ operator M1::I2::SequenceStr_ptr *() const ／ operator M1::I3::BSequenceStr_ptr *() const ）

（使用例）

```
M1::I1::SequenceLong *seq1 = new M1::I1::SequenceLong;
// seq1を使用した処理
・・・
M1::I1::SequenceLong_var seq2;
seq2 = seq1;

M1::I1::SequenceLong_var *seq3 = (M1::I1::SequenceLong*)seq2;
// 交換演算子
```

- 添字演算子（**const**なし）
  （ CORBA::Long &M1::I1::SequenceLong::operator[](CORBA::ULong) ／ TD::String_var &M1::I2::SequenceStr::operator[](CORBA::ULong) ／ TD::String_var & M1::I3::BSequenceStr_var::operator[](CORBA::ULong) ）

  与えられたインデックスに対応する_bufferの内容を返します。

（使用例）

```
// 固定長データ
long data[] = {1,2,3,4,5};
M1::I1::SequenceLong *seq1 = new M1::I1::SequenceLong (10,5,data,CORBA_TRUE);
M1::I1::SequenceLong_var vseq1;
vseq1 = seq1;
long  l = vseq1[3];  // lの値は4

// 可変長データ
char *data[] = {"sunday","monday","tuesday","wednesday"};
```

```
TD::String_var *wkdata = M1::I2::SequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}

M1::I2::SequenceStr *seq1 = new M1::I2::SequenceStr(10,4,wkdata,CORBA_TRUE);
M1::I2::SequenceStr_var vseq1;
vseq1 = seq1;
TD::String_var  d =vseq1[3];   // dの値はwednesday

// 可変長データ（最大値が指定されたバウンディッド・シーケンス）
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I3::BSequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I3::BSequenceStr *seq1 = new M1::I3::BSequenceStr(4,wkdata,CORBA_TRUE);
M1::I3::BSequenceStr vseq1;
vseq1 = seq1;
TD::String_var  d = vseq1[3];   // dの値はwednesday
```

- 添字演算子(**const**あり)
  (const CORBA::Long &M1::I1::SequenceLong::operator[](CORBA::ULong) const ／ const TD::String_var &M1::I2::SequenceStr::operator[](CORBA::ULong) const ／ const TD::String_var &M1::I3::BSequenceStr_var::operator[] (CORBA::ULong) const)

  与えられたインデックスに対応する_bufferの内容を返す。

  （使用例）

```
// 固定長データ
long data[] = {1,2,3,4,5};
M1::I1::SequenceLong *seq1 = new M1::I1::SequenceLong (10,5,data,CORBA_TRUE);
const M1::I1::SequenceLong_var vseq1(seq1);
const long  l = vseq1[3];  // lの値は4

// 可変長データ
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I2::SequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I2::SequenceStr *seq1 = new M1::I2::SequenceStr(10,4,wkdata,CORBA_TRUE);
const M1::I2::SequenceStr vseq1(seq1);
const TD::String_var  d = vseq1[3];   // dの値はwednesday

// 可変長データ（最大値が指定されたバウンディッド・シーケンス）
char *data[] = {"sunday","monday","tuesday","wednesday"};
TD::String_var *wkdata = M1::I3::BSequenceStr::allocbuf(4) ;
for (cnt = 0;cnt < 4;cnt++) {
    wkdata[cnt] = (const CORBA::Char *)data[cnt];
}
M1::I3::BSequenceStr *seq1 = new M1::I3::BSequenceStr(4,wkdata,CORBA_TRUE);

const M1::I3::BSequenceStr_var vseq1(seq1);
const TD::String_var  d = vseq1[3];   // dの値はwednesday
```

(4)領域獲得/解放

固定長データ

- inパラメタ …領域を獲得、および解放するための特別な関数を使用する必要はありません。

- outパラメタ …領域を獲得、および解放するための特別な関数を使用する必要はありません。
- inoutパラメタ…領域を獲得、および解放するための特別な関数を使用する必要はありません。

可変長データ

- inパラメタ …領域を獲得、および解放するための特別な関数を使用する必要はありません。
- outパラメタ …シーケンスクラスのインスタンスはC++のnew演算子、シーケンスのデータ領域はallocbuf関数(XX::allocbuf())により領域の獲得を行います。なお、ここで獲得した領域はスケルトンにより解放されます。
- inoutパラメタ…領域を獲得、および解放するための特別な関数を使用する必要はありません。

6.2 標準提供クラス

アプリケーションでデータ操作等を行うためのメソッドを、TDではクラスとして標準提供します。これをTDクラスと呼びます。TDクラスは、インタフェースクラスの基本クラス(スーパークラス)となります。

```
class TD
{
public:
        class String_var
        {
        public:
                String_var();
                String_var( char * );
                String_var( const char * );
                String_var( const String_var & );
                ~String_var();

                String_var      &operator=( char * );
                String_var      &operator=( const char * );
                String_var      &operator=( const String_var & );

                operator        char*();
                operator        const   char*() const;

                char    &operator[]( CORBA::ULong );
                char    operator[]( CORBA::ULong )     const;

        private:
                char    *_ptr;
        };

       class   WString_var
       {
        public:
                WString_var();
                WString_var( CORBA::WChar * );
                    WString_var( const CORBA::WChar * );
                    WString_var( const WString_var & );
                ~WString_var();
                    WString_var &operator=( CORBA::WChar * );
                WString_var &operator=( const CORBA::WChar * );
                WString_var &operator=( const WString_var & );

                operator    CORBA::WChar*();
                operator    const   CORBA::WChar*()    const;
                CORBA::WChar   &operator[]( CORBA::ULong );
                CORBA::WChar   operator[]( CORBA::ULong ) const;
                private:
                CORBA::WChar   *_ptr;
                };
```

```
        static void *string_alloc(unsinged long);
        static void *wstring_alloc(unsigned long);
        static void string_free(void *);
        static void wstring_free(void *);
};
```

TDクラスは、以下のメンバで構成されています。

- 可変長データ領域獲得関数(string_alloc()、wstring_alloc())
- 可変長データ領域解放関数(string_free()、wstring_free())

各メンバの詳細を、以下に示します。

6.2.1 String_var, WString_varクラス

String_var, WString_varクラスは文字列操作を簡易化するためのいくつかのメンバ関数を提供します。メンバ関数の詳細について、以下に示します。

- デフォルトコンストラクタ
  (TD::String_var() / TD::WString_var())

  _ptrをNULLに初期化します。

  (使用例)

```
TD::String_var data;
TD::WString_var data;
```

char *(CORBA::WChar *)コンストラクタ
(TD::String_var(char *) ／ TD::WString_var(CORBA::WChar *) , TD::String_var(const char *) / TD::WString_var(const CORBA::WChar *))

char*(CORBA::WChar *)コンストラクタはconstなしとconstありが定義されます。constなしの場合は、指定されたパラメタを_ptrに設定します。constありの場合は、指定されたパラメタのコピーを_ptrに設定します。

(使用例)

```
CORBA::Char *str1 = TD::string_alloc(5);
strcpy(str1,"test");

CORBA::WChar *str2 = TD::wstring_alloc(5);
wcscpy(str2,"あいうえ");

// char*(CORBA::WChar *)コンストラクタ(constなし)
TD::String_var data1(str1 );
TD::WString_var data2(str2 );

// char*(CORBA::WChar *)コンストラクタ(constあり)
TD::String_var, TD::WString_var data1( (const CORBA::Char *)str1 );
TD::WString_var data2( (const CORBA::WChar *)str2 );
```

- コピーコンストラクタ
  (TD::String_var (const TD::String_var &) / TD::WString_var (const TD::WString_var &))

  指定されたパラメタのpに設定されているデータ自身を_ptrに複写します。

  (使用例)

```
CORBA::Char *str1 = TD::string_alloc(5);
strcpy(str1,"test");
```

```
CORBA::WChar *str2 = TD::wstring_alloc(5);
wcscpy(str2,” あいうえ”);

// char*(CORBA::WChar *)コンストラクタ(constなし)
TD::String_var data1(str1 );

TD::WString_var data2(str2 );

// コピーコンストラクタ
TD::String_var data12(data1 );
TD::WString_var data22(data2 );
```

- デストラクタ
(TD::String_var() / TD::WString_var())

_ptrに設定されているデータの解放を行います。

(使用例)

```
CORBA::Char *str = TD::string_alloc(5);
strcpy(str1,"test");
CORBA::WChar *str = TD::wstring_alloc(5);
wcscpy(str2,” あいうえ”);

// char*(CORBA::WChar *)コンストラクタ(constなし)
TD::String_var *data1 =new TD::WString_var(str1 );
TD::WString_var *data2 =new TD::WString_var(str2 );

// デストラクタが起動され_ptr（この例ではstrの指す領域）が解放される
delete data1;
delete data2;
```

- **char *(CORBA::WChar *)代入演算子**
(TD::String_var &operator= (char *) / TD::WString_var &operator= (CORBA::WChar *) , TD::String_var (const char *) / TD::WString_var &operator= (const CORBA::WChar *))

char *(CORBA::WChar *)代入演算子は、constなしとconstありが定義されます。constなしの場合は、右辺で指定された(CORBA::WChar *) *ポインタを_ptrに代入します。constありの場合は、右辺で指定されたchar *(CORBA::WChar *)ポインタの指すデータのコピーを作成し、そのポインタを_ptrに設定します。

ptrにデータが設定されている場合は、constなし、constありいずれも、設定されている_ptrのデータを解放したあとで、それぞれの処理を実行します。

(使用例)

```
CORBA::Char *str1 =TD::string_alloc(5);
strcpy(str1,"test");
TD::String_var data1;

CORBA::WChar *str2 =TD::string_alloc(5);
wcscpy(str2,” あいうえ”);
TD::WString_var data2;

// char*(CORBA::WChar *)代入演算子(constなし)
data1 =str1;

data2 =str2;

// char *(CORBA::WChar *)代入演算子(constあり)
data1 =(const CORBA::Char *)"new data1";
data2 =(const CORBA::WChar *)"new data2";
```

・ 代入演算子
(TD::String_var & operator=(const TD::String_var &) / TD::WString_var & operator=(const TD::WString_var &))

代入演算子は、右辺で指定されたTD::String_var, TD::WString_varクラスの_ptrの設定されているデータのコピーを作成し、_ptrに設定します。すでに_ptrにデータが設定されている場合には、そのデータを解放したあと、処理を行います。

(使用例)

```
CORBA::Char *str1 = TD::string_alloc(5);
strcpy(str1,"test");
TD::String_var data1;

CORBA::WChar *str2 = TD::wstring_alloc(5);
wcscpy(str2," あいうえ" );
TD::WString_var data2;

// char*(CORBA::WChar *)代入演算子(constなし)
data1 = str1;
TD::String_var data12;

data2 = str2;
TD::WString_var data22;

// 代入演算子
data12 = data1;
data22 = data2;
```

・ 交換演算子
(operator char*()／operator const CORBA::WChar*() const , operator char*()／operator const CORBA::WChar*() const)

・ []演算子
(char &operator[](ULong)／CORBA::WChar operator[](ULong) const , char &operator[](ULong) ／ CORBA::WChar operator[](ULong) const)

[]演算子は、_ptrに設定されているパラメタで指定されたデータ+1番目のデータの参照値を返します。

(使用例)

```
CORBA::Char *str1 = TD::string_alloc(5);
strcpy(str1,"test");
TD::String_var data1;

CORBA::WChar *str2 = TD::wstring_alloc(5);
wcscpy(str2,"あいうえ");
TD::WString_var data2;

// char*(CORBA::WChar *)代入演算子(constなし)
data1 = str1;
data2 = str2;

// []演算子3文字目's'が返る
CORBA::Char x = data1[2];

// []演算子3文字目'う'が返る
CORBA::WChar x = data2[2];
```

6.2.2 可変長データ領域獲得関数

(TD::string_alloc(unsigned long length)、TD::wstring_alloc(unsigned long length))

lengthで指定された文字数＋1の動的なString領域の獲得を行います。正常に領域が獲得できた場合は、獲得したデータのポインタを返します。異常終了した場合には、NULLポインタを返します。

（使用例）

```
CORBA::Char *str = TD::string_alloc(5);
CORBA::WChar *wstr = TD::wstring_alloc(5);
```

6.2.3 可変長データ領域解放関数

指定されたポインタの領域を解放します。

（使用例）

```
CORBA::Char *str = TD::string_alloc(5);
CORBA::WChar *wstr = TD::wstring_alloc(5);
TD::string_free(str);
TD::wstring_free(wstr);
```

第7章 スナップショット機能

7.1 機能概要

スナップショットを使用して、クライアントからの要求に対する入出力情報をワークユニット単位にロギングすることにより、アプリケーションのデバッグを行うことができます。
ロギング情報をワークユニット単位にファイルに取得することにより、開発初期時のアプリケーションのデバッグを目的とします。

スナップショット機能はロギング情報の出力先により、以下の2種類があります。

- ファイル出力
- メモリ出力

7.2 ファイル出力のスナップショット

ワークユニット単位で、スナップショット情報の取得を定義します。出力されるスナップショットの単位は、ワークユニット配下で動作するサーバアプリケーションのプロセス単位です。
ワークユニット定義にスナップショット取得指定を定義することにより、スナップショット情報が出力されます。ファイル出力のスナップショットでは、ワークユニットの起動から停止までの範囲でスナップショット情報をロギングします。ロギング情報の出力先は、ワークユニット定義のスナップショット出力パスで指定されたディレクトリに、以下に示すファイル名で作成されます。

Windows32 Linux32

取得するワークユニット名．アプリケーションの実行プロセスID

Solaris32

デフォルトシステムの場合

取得するワークユニット名．アプリケーションの実行プロセスID

拡張システムの場合

取得するワークユニット名_システム名．アプリケーションの実行プロセスID

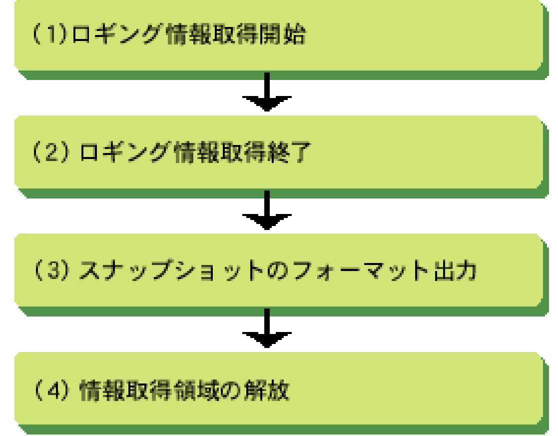
スナップショット出力パスを指定していない場合は、カレントディレクトリ配下に作成されます。ワークユニット定義の定義方法については、“OLTPサーバ運用ガイド”を参照してください。

7.3 メモリ出力のスナップショット

ワークユニット単位で、スナップショット情報を取得します。出力されるスナップショットの単位は、ワークユニット配下で動作するサーバアプリケーションすべてです。
ロギング情報をワークユニット単位にメモリに取得します。
運用中のワークユニットに対して、スナップショット取得開始コマンドを投入することにより、スナップショット情報のロギングを開始し、スナップショット取得終了コマンドを投入することにより、スナップショット情報のロギングを終了します。ロギングされたスナップショット情報は、スナップショット情報フォーマットコマンドにより、ファイルに出力されます。必要のなくなったスナップショット情報は、スナップショット情報削除コマンドで削除してください。

7.3.1 ロギング情報の取得手順

ロギング情報の取得手順を、以下に示します。



(1) サーバアプリケーションが動作しているのを確認後、tdstartsnapコマンドを投入してロギング情報の取得を開始します。

(2) 目的のロギング情報が取得できたところで、tdstopsnapコマンドを投入してロギング情報の取得を終了します。

(3) tdformsnapコマンドを投入し、取得したロギング情報をファイルに出力します。このロギング情報により、サーバアプリケーションのインタフェースを確認します。

(4) ロギング情報の取得のために獲得した領域（共有メモリ）を、tdfreesnapコマンドを投入して解放します。

7.3.2 スナップショット取得/参照のためのコマンド

ロギング情報の取得は、以下に示す5つのコマンドで操作を行います。なお、それぞれのコマンド文法の詳細については“リファレンスマニュアル（コマンド編）”を参照してください。

tdstartsnapコマンド

ロギング情報の取得を開始します。投入前に、ロギング情報を取得するワークユニットが起動されている必要があります。

tdstopsnapコマンド

ロギング情報の取得を終了します。ロギング情報を取得しているワークユニットを停止する前に投入してください。
tdstopsnapコマンドを投入する前に取得対象のワークユニットを停止した場合、ワークユニットの停止と同時にスナップショットの取得を終了します。この場合、tdstopsnapコマンドを投入する必要はありません。

tdformsnapコマンド

メモリに蓄積されたロギング情報を、ファイルへ出力します。作成されるディレクトリは、コマンド投入時のカレントディレクトリです。

Windows32 Linux32

ファイル名は、“取得するワークユニット名.コマンドの実行プロセスID”となります。

Solaris32

ファイル名は、デフォルトシステムの場合、“取得するワークユニット名.コマンドの実行プロセスID”、拡張システムの場合、“取得するワークユニット名_システム名.コマンドの実行プロセスID”となります。
tdformsnapコマンドは、ロギング情報の取得を終了させてから投入してください。

tdlistwusnapコマンド

ロギング情報をメモリに取得してあるワークユニットの一覧を表示します。

tdfreesnapコマンド

ロギング情報をメモリから削除します。

7.3.3 取得情報を格納するメモリ

取得情報を格納するメモリは、ワークユニット単位で固定です。このため、メモリが満杯になった場合、同一ワークユニット内の古い情報から上書きされます。

ワークユニットごとに割り当てられるメモリの大きさは128Kバイトです。

7.3.4 ロギング情報を取得できるワークユニットの数

ロギング情報を取得できるワークユニットの数を以下に示します。

システム規模	ロギングできるワークユニット数
SMALL	8
MODERATE	16
LARGE	24
SUPER	32

7.4 スナップショット情報の出力例

ログファイルの出力形式は以下のとおりです。

Windows32 Linux32

```
SNAP START: TIME: 10:51:02.159574               (1)
MODULE NAME      : SNAP10_OBJ1                  (2)
OPERATION NAME   : OPE1                         (3)
INPUT INFORMATION                               (4)
PARAMETER NUMBER:10                             (5)
VARIABLES
  PARAM0001: ATTRIBUTE   :long                  (6)
             DATA LENGTH  :4                    (7)
             DATA       :1                      (8)
              :
              :
SNAP START: TIME: 13:08:03.317794               (9)
MODULE NAME      : SNAP10_OBJ1                  (2)
OPERATION NAME   : OPE1                         (3)
RETURN INFORMATION                             (10)
RETURN VALUES : 0                              (11)
OUTPUT INFORMATION                             (12)
PARAMETER NUMBER:10                             (5)
VARIABLES
  PARAM0001: ATTRIBUTE   :short                 (6)
             DATA LENGTH  :2                    (7)
             DATA       :100                    (8)
              :
              :
```

(1) アプリケーションの実行開始時間
(2) オブジェクト名
(3) オペレーション名
(4) アプリケーション呼び出し時のパラメタ。タイプがinまたはinoutのパラメタ情報を表示。
(5) パラメタ数
(6) PARAMxxxx は xxxx番目のパラメタを表す。ATTRIBUTEはデータ属性。
(7) データ長(バイト数)
(8) データの値。整数以外は16進数で表示。
(9) アプリケーションからの復帰時間
(10) アプリケーションからの復帰時の情報
(11) 復帰値
(12) アプリケーションの復帰時のパラメタ。タイプがoutまたはinoutのパラメタ情報を表示またはユーザ例外情報を表示。

Solaris32

```
SNAP START: TIME: 10:51:02.159574               (1)
SYSTEM NAME      : system01                     (2)
```

```
MODULE NAME      : SNAP10_OBJ1                  (3)
OPERATION NAME   : OPE1                         (4)
INPUT INFORMATION                               (5)
PARAMETER NUMBER:10                             (6)
VARIABLES
  PARAM0001: ATTRIBUTE   :long                  (7)
             DATA LENGTH  :4                    (8)
             DATA      :1                       (9)
              :
              :
SNAP START: TIME: 13:08:03.317794              (10)

MODULE NAME      : SNAP10_OBJ1                  (3)

OPERATION NAME   : OPE1                         (4)

RETURN INFORMATION                             (11)
RETURN VALUES : 0                              (12)
OUTPUT INFORMATION                             (13)
PARAMETER NUMBER:10                             (6)
VARIABLES
  PARAM0001: ATTRIBUTE   :short                 (7)
             DATA LENGTH  :2                    (8)
             DATA      :100                     (9)
              :
              :
```

(1) アプリケーションの実行開始時間
(2) システム名。拡張システムの場合のみ表示。
(3) オブジェクト名
(4) オペレーション名
(5) アプリケーション呼び出し時のパラメタ。タイプがinまたはinoutのパラメタ情報を表示。
(6) パラメタ数
(7) PARAMxxxx は xxxx番目のパラメタを表す。ATTRIBUTEはデータ属性。
(8) データ長(バイト数)
(9) データの値。整数以外は16進数で表示。
(10) アプリケーションからの復帰時間
(11) アプリケーションからの復帰時の情報
(12) 復帰値
(13) アプリケーションの復帰時のパラメタ。タイプがoutまたはinoutのパラメタ情報を表示またはユーザ例外情報を表示。

属性別のパラメタ情報の出力例については、“7.4.1 パラメタ情報の出力例”を参照してください。
ユーザ例外情報の出力例については、ユーザ例外情報の出力例を参照してください。

7.4.1 パラメタ情報の出力例

7.4.1.1 long型の表示

```
PARAM0001 : ATTRIBUTE     : long             (1)
            DATA LENGTH : 4                  (2)
            DATA        : 2147483647         (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) データを表示

7.4.1.2 unsigned long型の表示

```
PARAM0001 : ATTRIBUTE    : unsigned long        (1)
            DATA LENGTH : 4                     (2)
            DATA        : 4294967295            (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) データを表示

7.4.1.3 short型の表示

```
PARAM0001 : ATTRIBUTE    : short                (1)
            DATA LENGTH : 2                     (2)
            DATA        : 32767                 (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) データを表示

7.4.1.4 unsigned short型の表示

```
PARAM0001 : ATTRIBUTE    : unsigned short       (1)
            DATA LENGTH : 2                     (2)
            DATA        : 65535                 (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) データを表示

7.4.1.5 long long型の表示

```
PARAM0001 : ATTRIBUTE    : long long            (1)
            DATA LENGTH : 8                     (2)
            DATA        : 9223300000000000000   (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) データを表示

7.4.1.6 float型の表示

```
PARAM0001 : ATTRIBUTE    : float                (1)
            DATA LENGTH : 4                     (2)
            DATA        : 1.700000E+007         (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) データを表示

7.4.1.7 double型の表示

```
PARAM0001 : ATTRIBUTE    : double              (1)
            DATA LENGTH : 8                    (2)
            DATA        : 1.700000E+308        (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) データを表示

7.4.1.8 long double型の表示

```
PARAM0001 : ATTRIBUTE    : long double         (1)
            DATA LENGTH : 8                   (2)
            DATA        : 1.700000E+308       (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) データを表示

7.4.1.9 char型の表示

```
PARAM0001 : ATTRIBUTE    : char               (1)
            DATA LENGTH : 1                  (2)
            DATA        : 0000 * 41          (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) *の左にバイト数を表示
  *の右にデータを16進で表示

7.4.1.10 wchar型の表示

```
PARAM0001 : ATTRIBUTE    : wchar              (1)
            DATA LENGTH : 2                  (2)
            DATA        : 0000 * 82a0        (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) *の左にバイト数を表示
  *の右にデータを16進で表示

7.4.1.11 octet型の表示

```
PARAM0001 : ATTRIBUTE    : octet                (1)
            DATA LENGTH : 1                    (2)
            DATA        : 0000 * 00            (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) *の左にバイト数を表示
  *の右にデータを16進で表示

7.4.1.12 boolean型の表示

```
PARAM0001 : ATTRIBUTE    : boolean            (1)
            DATA LENGTH : 1                  (2)
            DATA        : 0000 * 01          (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) *の左にバイト数を表示
  *の右にデータを16進で表示

7.4.1.13 string型の表示

```
PARAM0001 : ATTRIBUTE     : string               (1)
            DATA LENGTH : 25                   (2)
            DATA        : 0000 * 534e4150 20424153 49432054 59504520
```

```
                        0010 * 44415441 20544553 00
                                                   (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) *の左にバイト数を表示
*の右にデータを16進で表示

7.4.1.14 wstring型の表示

```
PARAM0002 : ATTRIBUTE   : wstring                  (1)
            DATA LENGTH : 22                       (2)
            DATA        : 0000 * 82a082a2 82a482a6 82a88140 82a982ab
                          0010 * 82ad82af 0000
                                                   (3)
```

(1) データ属性を表示
(2) データ長を表示
(3) *の左にバイト数を表示
*の右にデータを16進で表示

7.4.1.15 sequence型データ(要素に基本型)の表示

－ 要素が数値型の場合の表示

```
PARAM0001 : ATTRIBUTE   : sequence                 (1)
            SEQUENCE MAX LENGTH : 3                (2)
            SEQUENCE LENGTH     : 3                (3)
            ATTRIBUTE           : long             (4)
            SIZE of SEQUENCE    : 4                (5)
            DATA                : 1024             (6)
            DATA                : 2048             (7)
            DATA                : 4096             (8)
```

(1) データ属性を表示
(2) シーケンスの最大長を表示
(3) シーケンス長を表示
(4) シーケンスのデータ属性を表示
(5) シーケンスのデータ長を表示
(6)～(8) シーケンスのデータを表示

－ 要素が文字型の場合の表示

```
PARAM0001 : ATTRIBUTE           : sequence         (1)
            SEQUENCE MAX LENGTH : 2                (2)
            SEQUENCE LENGTH     : 2                (3)
            ATTRIBUTE           : string           (4)
            SIZE of SEQUENCE    : 16               (5)
            DATA LENGTH         : 16               (6)
            DATA        : 0000 * 50617261 6d322064 61746131 00000000
                                                   (7)
            DATA LENGTH         : 16               (8)
            DATA        : 0000 * 50617261 6d322064 61746132 00000000
                                                   (9)
```

(1) データ属性を表示
(2) シーケンスの最大長を表示
(3) シーケンス長を表示
(4) シーケンスのデータ属性を表示
(5) シーケンスのデータ長を表示
(6)(8) データ長を表示

(7)(9) *の左にバイト数を表示
*の右にデータを16進で表示

7.4.1.16 struct型データの表示

```
PARAM0001 : ATTRIBUTE     : struct                (1)
            NUMBER        : 4                     (2)
            ATTRIBUTE     : long                  (3)
            DATA LENGTH   : 4                     (4)
            DATA          : 13579                 (5)
            ATTRIBUTE     : char                  (6)
            DATA LENGTH   : 1                     (7)
            DATA          : 0000 * 41             (8)
            ATTRIBUTE     : octet                 (9)
            DATA LENGTH   : 1                    (10)
            DATA          : 0000 * 61            (11)
            ATTRIBUTE     : string               (12)
            DATA LENGTH   : 35                   (13)
            DATA          : 0000 * 61626364 65663031 32333435 36373839
                            0010 * 41424344 45463031 32333435 36373839
                            0020 * 58595a       (14)
```

(1) データ属性を表示
(2) 構造体のメンバ数を表示
(3) 第1メンバデータ属性を表示
(4) 第1メンバデータ長を表示
(5) 第1メンバデータを表示
(6) 第2メンバデータ属性を表示
(7) 第2メンバデータ長を表示
(8) 第2メンバデータを表示
*の左にバイト数を表示
(9) 第3メンバデータ属性を表示
(10) 第3メンバデータ長を表示
(11) 第3メンバデータを表示
*の左にバイト数を表示
(12) 第4メンバデータ属性を表示
(13) 第4メンバデータ長を表示
(14) 第4メンバのデータを表示
*の左にバイト数を表示
*の右にデータを16進で表示

7.4.1.17 array型データの表示

― 要素が数値型の場合の表示

```
PARAM0001 : ATTRIBUTE    : array                 (1)
            ATTRIBUTE    : long                  (2)
            DATA LENGTH  : 4                     (3)
            DIMENSIONS   : 1                     (4)
            ELEMENTS                             (5)
              1 NUMBER   : 3                     (6)
            DATA         : -2147483648           (7)
            DATA         : 0                     (8)
            DATA         : 100                   (9)
```

(1) データ属性を表示
(2) 配列の要素のデータ属性を表示
(3) 配列の要素のデータ長を表示
(4) 配列の次元数を表示
(5) 次元ごとの要素数を表示

(6) 1次元目の要素数を表示
(7)～(9) データを表示

－ 要素が文字型の場合の表示

```
PARAM0001 : ATTRIBUTE    : array                (1)
             ATTRIBUTE   : string               (2)
             DIMENSIONS  : 1                    (3)
             ELEMENTS                           (4)
                1 NUMBER : 2                    (5)
             DATA LENGTH : 20                   (6)
             DATA        : 0000 * 41414141 41000000 00000000 00000000
                           0010 * 00000000      (7)
             DATA LENGTH : 20                   (8)
             DATA        : 0000 * 42424242 42420000 00000000 00000000
                           0010 * 00000000      (9)
```

(1) データ属性を表示
(2) 配列の要素のデータ属性を表示
(3) 配列の要素のデータ長を表示
(4) 配列の次元数を表示
(5) 次元ごとの要素数を表示
(6)(8) 文字列長
(7)(9) *の左にバイト数を表示
*の右にデータを16進で表示

－ 要素が構造体型の場合の表示

```
PARAM0001 : ATTRIBUTE    : array                (1)
             ATTRIBUTE   : struct               (2)
             DIMENSIONS  : 1                    (3)
             ELEMENTS                           (4)
                1 NUMBER : 2                    (5)
             NUMBER      : 2                    (6)
             ATTRIBUTE   : long                 (7)
             DATA LENGTH : 4                    (8)
             DATA        : 100                  (9)
             ATTRIBUTE   : string               (10)
             DATA LENGTH : 16                   (11)
             DATA        : 0000 * 41727261 79206f66 20537472 75637400
                                                (12)
             NUMBER      : 2                    (13)
             ATTRIBUTE   : long                 (14)
             DATA LENGTH : 4                    (15)
             DATA        : 200                  (16)
             ATTRIBUTE   : string               (17)
             DATA LENGTH : 16                   (18)
             DATA        : 0000 * 41727261 79206f66 20537472 7563747f
                                                (19)
```

(1) データ属性を表示
(2) 配列要素のデータ属性を表示
(3) 配列の次元数を表示
(4) 要素数の表示
(5) 1次元目の要素数を表示
(6)(13) 構造体型のメンバ数を表示
(7)(14) 構造体型の第1メンバ属性を表示
(8)(15) 構造体型の第1メンバ属性長を表示
(9)(16) 構造体型の第1メンバデータを表示
(10)(17) 構造体型の第2メンバを表示
(11)(18) 構造体の第2メンバデータ長を表示

(12)(19) 構造体の第2メンバデータを表示
*の左にバイト数を表示
*の右にデータを16進で表示

7.4.2 ユーザ例外情報の出力例

スナップショットによって採取されるログファイルにおけるユーザ例外情報について、出力例を以下に示します。

7.4.2.1 例外の表示

```
USER EXCEPTION INFORMATION                         (1)
major INFORMATION : 1                              (2)
ID LENGTH         : 24                             (3)
ID NAME           : IDL:EXCEP/OBJ1/EXP2:1.0        (4)
MEMBER NUMBER     : 3                              (5)
VARIABLES
    PARAM0001 : ATTRIBUTE   : long                 (6)
                DATA LENGTH : 4
                DATA        : 200
    PARAM0002 : ATTRIBUTE   : string
                DATA LENGTH : 11
                DATA        : 0000 * 45584345 50204253 545200
    PARAM0003 : ATTRIBUTE   : wstring

                DATA LENGTH : 22

                DATA        : 0000 * 82a082a2 82a482a6 82a88140 82a982ab

                              0010 * 82ad82af 0000
```

(1) ユーザ例外情報見出しを表示
(2) メジャー情報を表示
(3) ID名長を表示
(4) ID名を表示
(5) パラメタ数を表示
(6) パラメタの表示
※パラメタ表示については前述の記事を参照してください。

第8章 ワークユニットが提供する運用支援機能の使用方法

ワークユニットが提供する運用支援機能について、以下に示す内容で説明します。

- ワークユニット出口機能の使用方法
- プロセス回収出口機能の使用方法
- ワークユニットプロセス情報通知機能の使用方法

Solaris32 Linux32

- ユーティリティワークユニットのプロセス停止出口機能の使用方法

注意

ワークユニット出口機能およびプロセス回収出口機能を使用する場合は、運用中にワークユニット定義で指定したカレントディレクトリのディスク領域で、領域不足が発生しないよう十分注意してください。
領域不足が発生すると、ワークユニット出口およびプロセス回収出口の実行に失敗する場合があります。

8.1 ワークユニット出口機能の使用方法

ワークユニット出口機能の使用方法について説明します。

8.1.1 概要

ワークユニット出口は、ワークユニット単位で資源の獲得／解放などの処理を行うための出口プログラムです。ワークユニット起動時に、アプリケーションプロセスの起動前に1度呼び出されます。また、ワークユニット停止時およびワークユニット異常終了時に、ワークユニット配下のプロセスの回収処理が終了したあとに1度呼び出されます。
また、アプリケーションの前出口プログラムおよび後出口プログラムは、アプリケーションプロセスの起動および停止時に呼び出される出口プログラムであるのに対し、ワークユニット出口プログラムは、ワークユニットの起動、停止および異常終了時に呼び出される出口プログラムです。
これにより、ワークユニット単位で共有メモリなどの資源を獲得する場合、ワークユニット出口より共有メモリの獲得／解放処理を行うことが可能となります。
CORBAアプリケーション、トランザクションアプリケーション、EJBアプリケーション、一般アプリケーションのワークユニット（ユーティリティワークユニット）で使用することができます。
ユーティリティワークユニットはSolaris版、Linux版のみ使用することができます。

また、ワークユニット出口機能とプロセス回収出口機能を併用する場合は、実行モジュールは同一とする必要があります。

8.1.2 プログラミングの流れ

以下に示す形式で作成してください。なお、ワークユニット出口の作成言語は、ワークユニット定義の“kind”（ワークユニット種別）および、“Application Language”（アプリケーション言語）にかかわらず、C言語のみ有効です。

関数様式

[形式]

```
long 出口名 ( char      *sysname,
              char      *wuname,
              char      *username,
              long      mode,
              long      *userdata1,
              long      *userdata2)
```

[引数]

パラメタ	設定内容	入出力
char *sysname	システム名（注1）	入力値

パラメタ	設定内容	入出力
char *wuname	ワークユニット名	入力値
char *username	ワークユニット起動ユーザ名	入力値
long mode	状態遷移モード	入力値
long *userdata1	復帰情報1設定領域アドレス	入出力値（注2）
long *userdata2	復帰情報2設定領域アドレス	入出力値（注2）

(注1)

Windows32 Linux32

“default”が設定されます。

Solaris32

デフォルトシステムの場合は“default”が設定されます。拡張システムの場合は拡張システムのシステム名が設定されます。

(注2)

状態遷移モードが1（ワークユニット起動）の場合、出口プログラムで復帰情報を設定します。それ以外の状態モードでは、ワークユニット起動時に設定された復帰情報が出口プログラムの入力値として設定されます。

状態遷移モード

状態遷移モードでは、ワークユニット出口の呼出し契機を通知します。以下のいずれかが設定されます。

状態	設定値	呼出し契機
ワークユニット起動	1	ワークユニット起動時
ワークユニット通常停止	2	ワークユニット通常停止時
ワークユニット強制停止	3	ワークユニット強制停止時 Interstageの起動サービス強制停止時 Interstageの全強制停止時
ワークユニット異常終了	4	ワークユニット異常終了時
ワークユニット起動失敗	5	ワークユニット起動失敗時

ワークユニット出口は、各呼出し契機で同一の出口プログラムが呼び出され、状態遷移モードにより呼出し契機を通知します。したがって、出口プログラムでは、必要に応じて状態遷移モードごとに処理を作成し、設定されている状態遷移モードを判断して処理の振り分けを行ってください。

復帰情報1，2

ワークユニット起動時のユーザ情報を設定する領域です。ワークユニット起動時の情報を、その他の状態で呼び出された場合に持ちまわることができます。

状態遷移モードが“ワークユニット起動”の場合に、ワークユニット出口の出力値として復帰情報を設定してください。その他のモードの場合は、ワークユニット起動時に設定された復帰情報が、入力値として設定されます。
これにより、ワークユニット起動時の復帰情報を、その他の呼出し契機に使用することができます。また、ワークユニット起動時に設定された復帰情報は、プロセス回収出口にも入力値として設定されます。さらに、ワークユニットプロセス情報通知機能を使用して、サーバアプリケーションプログラムに情報を通知することができます。

[復帰値]

復帰値	意味
0	正常復帰
0以外	異常復帰

出口プログラムが正常復帰する場合は、復帰値に0を設定し、異常の場合は0以外を設定してください。
ワークユニット起動時にワークユニット出口が異常復帰（復帰値が0以外で復帰）した場合、メッセージが出力され、ワークユニットの起動は失敗します。

また、ワークユニット起動以外の契機で呼び出されたワークユニット出口が異常復帰した場合は、メッセージのみ出力し、動作異常とはなりません。

プログラミング例

ワークユニット出口プログラムの記述例を以下に示します。

```
long wuexit( char    *sysname,
             char    *wuname,
             char    *username,
             long     mode,
             long    *userdata1,
             long    *userdata2)
{

    long    data1, data2;

    /* 状態遷移モードごとに処理を振り分ける */
    switch(mode){
      case    1:
        /* ワークユニット起動時の処理を記述 */
        /* 復帰情報を設定して復帰 */
        *userdata1 = data1;
        *userdata2 = data2;
        break;

      case    2:
        /* 起動時の情報の取出し */
        data1 = *userdata1;
        data2 = *userdata2;
        /* ワークユニット通常停止時の処理を記述 */
        break;

      case    3:
         /* 起動時の情報の取出し */
        data1 = *userdata1;
        data2 = *userdata2;
        /* ワークユニット強制停止時の処理を記述 */
        break;

      case    4:
        /* 起動時の情報の取出し */
        data1 = *userdata1;
        data2 = *userdata2;
        /* ワークユニット異常終了時の処理を記述 */
        break;

      case    5:
        /* 起動時の情報の取出し */
        data1 = *userdata1;
        data2 = *userdata2;
        /* ワークユニット起動失敗時の処理を記述 */
        break;
    }

    return(0);
}
```

ワークユニット定義

ワークユニット定義の記述例を示します。

Windows32

```
[WORK UNIT]
Name: ･･･
Kind: ORB
[APM]
Name: TDNORM

[Control Option]
･･･
WorkUnit Exit Program: wuexit                                 ← 出口プログラム名を指定
Executable File of Exit Program for Salvage: libwuexit.DLL    ← 実行ファイル名を指定
Maximum Processing Time for Exit Program: 60                  ← 出口プログラム最大処理時間
･･･
```

Solaris32 Linux32

```
[WORK UNIT]
Name: ･･･
Kind: ORB
[APM]
Name: TDNORM

[Control Option]
･･･
WorkUnit Exit Program: wuexit                                 ← 出口プログラム名を指定
Executable File of Exit Program for Salvage: libwuexit.so     ← 実行ファイル名を指定
Maximum Processing Time for Exit Program: 60                  ← 出口プログラム最大処理時間
･･･
```

ワークユニット出口機能を使用するために必要なワークユニット定義を以下に示します。
[Control Option]セクションに以下の定義を設定します。

WorkUnit Exit Program:ワークユニット出口プログラム名

ワークユニット出口プログラム名を設定します。
31バイト以内の英数字とアンダースコアが使用できます。

Executable File of Exit Program for Salvage:ワークユニット出口プログラム実行ファイル名

ワークユニット出口プログラムが格納されている実行ファイル名を指定します。

Windows32

31バイト以内の制御文字(ShiftJISの0x00～0x1f,0x7f)を除く文字が使用できます。ただし、半角および全角英文字の大文字と小文字は区別されません。

Solaris32 Linux32

31バイト以内の空白文字と半角カナを除く文字が使用できます。

“WorkUnit Exit Program”ステートメントが設定された場合、本ステートメントは省略できません。

Maximum Processing Time for Exit Program:出口プログラム最大処理時間

出口プログラムの最大処理時間の監視値(秒)を設定します。
1～1800の整数値。
本ステートメントは省略可能です。本ステートメントを省略した場合、省略値として300が設定されます。

標準出力および標準エラー出力

出口プログラムが動作する時のカレントディレクトリは、以下のような構成になっており、カレントディレクトリ配下に標準出力用のファイルと標準エラー出力用のファイルが設定されています。

標準出力には、stdoutファイルが割り当てられています。出口プログラムで標準出力にデータを出力した場合に、出力先として使用されます。
標準エラー出力には、stderrファイルが割り当てられています。出口プログラムで標準エラー出力にデータを出力した場合に、出力先として使用されます。

Windows32

カレントフォルダ:xxx¥yyy¥zzz_exit
xxx :ワークユニット定義で指定されたフォルダ
yyy :当該ワークユニット名
zzz :ワークユニット出口プログラムの実行プロセスid

Solaris32 Linux32

カレントディレクトリ:xxx/yyy/zzz_exit
xxx :ワークユニット定義で指定されたディレクトリ
yyy :当該ワークユニット名
zzz :ワークユニット出口プログラムの実行プロセスid

Solaris32

なお、マルチシステム機能の拡張システムを使用している場合は、上記yyyは“当該ワークユニット名.拡張システム名”となります。

8.1.3 ソースのコンパイル・リンク

ワークユニット出口プログラムは、動的リンクライブラリの形式で作成してください。

Windows32

Visual C++でコンパイルする場合のコンパイルオプションの例を以下に示します。

[コンパイルオプションの例]

Visual C++でコンパイルする場合は、[プロジェクト]-[設定]-[C/C++]の「カスタマイズ」で、以下の表に示すオプションを設定してください。

カテゴリー	項目	設定値
コード生成	Processer	80386を推奨
	構造体メンバのアライメント	4バイト
	使用するランタイム	マルチスレッド(DLL)
最適化		デフォルトを推奨
プリプロセッサ	プリプロセッサの定義	"OM_PC","OM_WIN32_BUILD", "__STDC__" を追加

Solaris32

また、コンパイルおよびリンクオプションに“-mt”を指定し、スレッド・セイフな形式で作成してください。

Linux32

また、コンパイルオプションに“-D-REENTRANT”を指定し、リンクオプションに“-lpthread”を指定して、スレッド・セイフな形式で作成してください。

8.1.4 注意事項

ワークユニット出口機能を使用する場合、次のことに注意してください。

- ワークユニット出口プログラムは、呼出し契機ごとに、新規に出口用のプロセスが起動され呼び出されます。したがって、外部変数などを使用して、起動時のデータを停止時に引き継ぐことはできません。
- ワークユニット起動時にワークユニット出口プログラムで異常が発生し、出口を呼び出すプロセスが異常終了した場合、メッセージが出力されワークユニット起動は失敗します。
  また、ワークユニット起動以外の契機で呼び出されたワークユニット出口プログラムが異常終了した場合は、メッセージのみ出力し動作異常とはなりません。
- ワークユニットの起動／停止の契機で呼び出される場合、ワークユニット出口が復帰するまで起動／停止処理は終了しません。ループ防止のために、必ず出口プログラム最大処理時間を設定してください。
- 最大処理時間を超過した場合、出口を呼び出すプロセスは強制停止され、メッセージが出力されます。ワークユニットの起動時には、ワークユニット起動が失敗します。その他の契機で呼び出された場合は、動作異常とはなりません。

8.2 プロセス回収出口機能の使用方法

プロセス回収出口機能の使用方法について説明します。

8.2.1 概要

プロセス回収出口は、アプリケーションプロセス終了後にアプリケーションプロセス内で獲得した資源を解放するための出口プログラムです。ワークユニット停止やアプリケーション異常などによる、アプリケーションプロセスの終了を契機に、アプリケーションプロセスとは別プロセスよりプロセス回収出口が呼び出されます。
これにより、共有メモリなどに作成したアプリケーションプロセスの資源を解放することが可能となります。
本機能はCORBAアプリケーション、トランザクションアプリケーション、EJBアプリケーション、一般アプリケーションのワークユニット(ユーティリティワークユニット)で使用することができます。ただし、ユーティリティワークユニットでは、ワークユニット停止コマンドを使用してワークユニットを停止した場合、および、ワークユニット異常終了により強制停止されるプロセスに対しては、プロセス回収出口は呼び出されません。
ユーティリティワークユニットはSolaris版、Linux版のみ使用することができます。

また、ワークユニット出口機能とプロセス回収出口機能を併用する場合は、実行モジュールは同一とする必要があります。

8.2.2 プログラミングの流れ

以下に示す形式で作成してください。なお、プロセス回収出口の作成言語は、ワークユニット定義の“kind”(ワークユニット種別)および、“Application Language”(アプリケーション言語)にかかわらず、C言語のみ有効です。

関数様式

[形式]

```
long 出口名 ( char      *sysname,
              char      *wuname,
              char      *username,
              long      mode,
              long      user_data1,
              long      user_data2,
              long      pid)
```

[引数]

パラメタ	設定内容	入出力
char *sysname	システム名(注)	入力値
char *wuname	ワークユニット名	入力値
char *username	ワークユニット起動ユーザ名	入力値
long mode	回収モード	入力値
long user_data1	ワークユニット出口復帰情報1設定値	入力値

パラメタ	設定内容	入出力
long user_data2	ワークユニット出口復帰情報2設定値	入力値
long pid	アプリケーションプロセスのプロセス番号	入力値

(注)

Windows32 Linux32

“default”が設定されます。

Solaris32

デフォルトシステムの場合は“default”が設定されます。拡張システムの場合は拡張システムのシステム名が設定されます。

回収モード

回収モードでは、プロセス回収出口の呼出し契機を通知します。以下のいずれかが設定されます。

状態	設定値	呼出し契機
ワークユニット通常停止	1	ワークユニット通常停止によるプロセス停止
ワークユニット強制停止	2	ワークユニット強制停止 Interstageの起動サービス強制停止時 Interstageの全強制停止時
アプリケーション異常終了	3	アプリケーション異常終了によるプロセス停止 Solaris32 Linux32 ユーティリティワークユニットの自動停止モードを設定している場合のプロセス停止
アプリケーションタイムアウト	4	アプリケーションタイムアウトによるプロセス停止
プロセス回収	5	異常終了、ワークユニット起動失敗など別プロセスが起因する異常によるプロセスの停止、および、tdmodifyprocnumによるプロセス停止

出口プログラムでは、必要に応じて回収モードごとに処理を作成し、設定されている呼出し契機を判断して処理の振り分けを行ってください。

ワークユニット出口復帰情報1，2

ワークユニット起動時に、ワークユニット出口の出力値として返された復帰情報が設定されます。

［復帰値］

復帰値	意味
0	正常復帰
0以外	異常復帰

出口プログラムが正常復帰する場合は、復帰値に0を設定し、異常の場合は0以外を設定してください。
ただし、プロセス回収出口が異常復帰した場合でも、メッセージのみ出力され動作異常とはなりません。

プログラミング例

プロセス回収出口プログラムの記述例を以下に示します。

```
long recoverexit( char    *sysname,
                  char    *wuname,
                  char    *username,
                  long     mode,
                  long     userdata1,
                  long     userdata2,
                  long     pid)
{
```

```
    /* 状態遷移モードごとに処理を振り分ける */
    switch(mode){
      case    1:
        /* ワークユニット通常停止時の処理を記述 */
        break;

      case    2:
        /* ワークユニット強制停止時の処理を記述 */
        break;

      case    3:
        /* アプリケーション異常終了時の処理を記述 */
        break;

      case    4:
        /* アプリケーションタイムアウト時の処理を記述 */
        break;

      case    5:
        /* プロセス回収時の処理を記述 */
        break;
    }

    return(0);
}
```

ワークユニット定義

ワークユニット定義の記述例を示します。

Windows32

```
[WORK UNIT]
Name: ･･･
Kind: ORB
[APM]
Name: TDNORM

[Control Option]
･･･
Executable File of Exit Program for Salvage: libexit.DLL        ← 実行ファイル名を指定
Maximum Processing Time for Exit Program: 60                    ← 出口プログラム最大処理時間
･･･

[Application Program]
･･･
Exit Program for Process Salvage: procexit                      ← 出口プログラム名を指定
Executable File of Exit Program for Salvage: libexit.DLL        ← 実行ファイル名を指定
Maximum Processing Time for Exit Program: 60                    ← 出口プログラム最大処理時間
･･･
```

Solaris32 Linux32

```
[WORK UNIT]
Name: ･･･
Kind: ORB
[APM]
Name: TDNORM

[Control Option]
･･･
Executable File of Exit Program for Salvage: libexit.so          ← 実行ファイル名を指定
Maximum Processing Time for Exit Program: 60                     ← 出口プログラム最大処理時間
･･･

[Application Program]
･･･
Exit Program for Process Salvage: procexit                       ← 出口プログラム名を指定
Executable File of Exit Program for Salvage: libexit.so          ← 実行ファイル名を指定
Maximum Processing Time for Exit Program: 60                     ← 出口プログラム最大処理時間
･･･
```

プロセス回収出口機能を使用するために必要なワークユニット定義を以下に示します。
［Control Option］セクション、および、アプリケーションの運用形態（常駐運用、非常駐運用、マルチオブジェクト常駐運用）にあわせて、それぞれ、[Application Program]セクション、[Nonresident Application Process]セクション、[Multiresident Application Process]セクションに設定します。

Exit Program for Process Salvage:プロセス回収出口プログラム名

プロセス回収出口プログラム名を設定します。
31バイト以内の英数字とアンダースコアが使用できます。
アプリケーションの運用形態にあわせて、[Application Program]セクション、[Nonresident Application Process]セクション、[Multiresident Application Process]セクションのいずれかに設定します。

Executable File of Exit Program for Salvage:出口プログラム実行ファイル名

プロセス回収出口プログラムが格納されている実行ファイル名を指定します。

Windows32

31バイト以内の制御文字（ShiftJISの0x00～0x1f,0x7f）を除く文字が使用できます。ただし、半角および全角英文字の大文字と小文字は区別されません。

Solaris32 Linux32

31バイト以内の空白文字と半角カナを除く文字が使用できます。

“Exit Program for Process Salvage”ステートメントが設定された場合、本ステートメントを［Control Option］セクションまたはアプリケーションの運用形態にあわせたセクション（[Application Program]、[Nonresident Application Process]、[Multiresident Application Process]）に設定する必要があります。なお、両方のセクションに設定された場合は、運用形態別のセクションの設定が有効となります。

Maximum Processing Time for Exit Program:出口プログラム最大処理時間

出口プログラムの最大処理時間の監視値（秒）を設定します。
1～1800の整数値。
本ステートメントが［Control Option］セクションおよびアプリケーションの運用形態にあわせたセクションの両方に設定された場合は、運用形態別のセクションの設定が有効となります。
本ステートメントは省略可能です。本ステートメントを省略した場合、省略値として300が設定されます。

標準出力および標準エラー出力

プロセス回収出口プログラムが動作する時のカレントディレクトリは、以下のような構成になっており、カレントディレクトリ配下に標準出力用のファイルと標準エラー出力用のファイルが設定されています。

標準出力には、stdoutファイルが割り当てられています。プロセス回収出口プログラムで標準出力にデータを出力した場合に、出力先として使用されます。

標準エラー出力には、stderrファイルが割り当てられています。プロセス回収出口プログラムで標準エラー出力にデータを出力した場合に、出力先として使用されます。

Windows32

```
カレントフォルダ:xxx¥yyy¥zzz_exit
 xxx  :ワークユニット定義で指定されたフォルダ
 yyy  :当該ワークユニット名
 zzz  :プロセス回収出口プログラムの実行プロセスid
```

Solaris32 Linux32

```
カレントディレクトリ:xxx/yyy/zzz_exit
 xxx  :ワークユニット定義で指定されたディレクトリ
 yyy  :当該ワークユニット名
 zzz  :プロセス回収出口プログラムの実行プロセスid
```

Solaris32

なお、マルチシステム機能の拡張システムを使用している場合は、上記yyyは"当該ワークユニット名.拡張システム名"となります。

8.2.3 ソースのコンパイル・リンク

プロセス回収出口プログラムは、動的リンクライブラリの形式で作成してください。

Windows32

Visual C++でコンパイルする場合のコンパイルオプションの例を以下に示します。

[コンパイルオプションの例]

Visual C++でコンパイルする場合は、[プロジェクト]-[設定]-[C/C++]の「カスタマイズ」で、以下の表に示すオプションを設定してください。

カテゴリー	項目	設定値
コード生成	Processer	80386を推奨
	構造体メンバのアライメント	4バイト
	使用するランタイム	マルチスレッド(DLL)
最適化		デフォルトを推奨
プリプロセッサ	プリプロセッサの定義	"OM_PC","OM_WIN32_BUILD", "__STDC__" を追加

Solaris32

また、コンパイルおよびリンクオプションに"-mt"を指定し、スレッド・セイフな形式で作成してください。

Linux32

また、コンパイルオプションに"-D-REENTRANT"を指定し、リンクオプションに"-lpthread"を指定して、スレッド・セイフな形式で作成してください。

8.2.4 注意事項

プロセス回収出口機能を使用する場合、次のことに注意してください。

- プロセス回収出口プログラムは、呼出し契機ごとに、新規に出口用のプロセスが起動され呼び出されます。したがって、外部変数などを使用してアプリケーションプログラムからデータを引き継ぐことはできません。

- プロセス回収出口プログラムで異常が発生し、出口を呼び出すプロセスが異常終了した場合、メッセージが出力されます。動作異常とはなりません。
- ワークユニットの停止契機で呼び出される場合、プロセス回収出口プログラムが復帰するまで停止処理は終了しません。ループ防止のために、必ず、出口プログラム最大処理時間を設定してください。
- 最大処理時間を超過した場合、出口を呼び出すプロセスは強制停止され、メッセージが出力されます。動作異常とはなりません。

8.3 ワークユニットプロセス情報通知機能の使用方法

ワークユニットプロセス情報通知機能の使用方法について説明します。

8.3.1 概要

アプリケーションプロセスの起動時に、プロセス固有の情報が特定の環境変数に設定されます。アプリケーションプログラムでは、自プロセスの情報を環境変数より獲得することができます。これにより、アプリケーションプログラムは、プロセスごとの資源を独自に管理するための情報を獲得することができます。
また、アプリケーション異常やタイムアウトによりアプリケーションプロセスが終了した場合、プロセス再起動時に終了前の情報を引き継いだり、現プロセスの起動状態を獲得することができます。

以下に採取可能な情報を記載します。

設定情報	内容
プロセス通番	ワークユニット内でユニークなプロセスのシリアル番号であり、プロセス再起動時にも、異常終了したプロセスと同じ番号が設定されます。ただし、プロセス多重度変更機能によるプロセスの増減を行った場合、ユニークな番号であることは保証されますが、連続した番号は保証されません。
Solaris32 システム名	デフォルトシステムの場合は“default”が設定されます。拡張システムの場合は拡張システムのシステム名が設定されます。
ワークユニット名	ワークユニット名
起動ユーザ名	ワークユニット起動コマンドを実行したユーザ名
プロセス起動回数	プロセスがワークユニット起動による初回起動なのか、プロセス異常終了による再起動なのかを識別する情報
ワークユニット出口復帰情報	ワークユニット起動時のワークユニット出口で、出力値として設定されたワークユニット出口復帰情報1，2

8.3.2 プログラミングの流れ

以下に示す方法でアプリケーションに組み込んでください。

設計方法

プロセス通番

共有メモリなどで、ワークユニット内のプロセスごとにブロックを割り当て、情報を保持している場合、プロセス異常終了後に再起動したプロセスで、異常終了前のプロセスと同一の共有メモリのブロックを使用したい場合などに、環境変数よりプロセス通番を獲得します。

システム名 Solaris32

アプリケーション内でInterstageのAPIを使用している場合、入力値として設定する必要のある場合に、環境変数よりシステム名を獲得します。

ワークユニット名

アプリケーション内でワークユニット名を意識したアプリケーションの処理(ワークユニットにより動作を切り分けているなど)を行っている場合、環境変数よりワークユニット名を獲得します。

起動ユーザ名

アプリケーションが起動されたユーザ名を意識したアプリケーションの処理(ユーザ名により認証を行っているなど)を行っている場合、環境変数より起動ユーザ名を獲得します。

プロセス起動回数

自プロセスがワークユニット起動による初回起動か、アプリケーション異常終了後の再起動による起動かを知る必要がある場合に、環境変数よりプロセス起動回数を獲得します。

ワークユニット出口復帰情報

ワークニット内で共有メモリなどを使用する場合、ワークユニット起動時に呼び出されるワークユニット出口の復帰情報に、共有メモリIDなどを設定し復帰することで、ワークユニット配下のアプリケーションプロセスで共有メモリ操作が可能となります。その場合、環境変数よりワークユニット出口復帰情報を獲得し、共有メモリのマッピング処理などを行います。

環境変数名の詳細

以下の環境変数に情報を設定します。

環境変数名	設定情報	最大データ長
IS_APL_SERIALNUM	プロセス通番	10バイト
Solaris32 IS_APL_SYSNAME	システム名	8バイト
IS_APL_WUNAME	ワークユニット名	36バイト
IS_APL_USRNAME	起動ユーザ名	48バイト
IS_APL_STARTNUM	プロセス起動回数	10バイト
IS_APL_INFO1	ワークユニット出口復帰情報1	10バイト
IS_APL_INFO2	ワークユニット出口復帰情報2	10バイト

アプリケーション作成方法

アプリケーション上でプロセス情報を獲得したい場合は、環境変数より情報を取り出す必要があります。
環境変数情報を格納する領域は、必ず最大データ長よりも大きな領域を用意してください。

[C、C++の場合の記述例]

Windows32

GetEnvironmentVariable()を使用して環境変数の値を取得します。

```
#include <windows.h>
long FUNC()
{
    long    rtn,  length=11;
    char    proc_num[11];
    rtn = GetEnvironmentVariable( “IS_APL_SERIALNUM” ,proc_num,length);
              :
```

Solaris32 Linux32

getenv()を使用して環境変数の値を取得します。

```
#include <stdlib.h>
long FUNC()
{
    char    *proc_num;
    proc_num = getenv( “IS_APL_SERIALNUM” );
              :
```

[COBOLの場合の記述例] Windows32 Solaris32

環境変数の値を参照するためには、DISPLAY文とACCEPT文をこの順に実行します。
DISPLAY文では、参照したい環境変数の名前を一意名または定数に指定し、機能名ENVIRONMENT-NAMEに対応付けた呼び名をUPON指定に書きます。DISPLAY文を実行すると、DISPLAY文の一意名または定数に設定した名前を持つ環境変数が参照可能になります。
ACCEPT文では、機能名ENVIRONMENT-VALUEに対応付けた呼び名をFROM指定に書きます。ACCEPT文を実行すると、DISPLAY文の実行によって参照可能になった環境変数の値が、ACCEPT文の一意名に設定されます。

```
ENVIRONMENT     DIVISION.
CONFIGURATION   SECTION.
SPECIAL-NAMES.
    ENVIRONMENT-NAME  IS ENVNAME
    ENVIRONMENT-VALUE  IS ENVVAL.
DATA            DIVISION.
WORKING-STORAGE  SECTION.
01  PROC_NUM  PIC  X(11).
PROCEDURE     DIVISION.
    DISPLAY  "IS_APL_SERIALNUM"  UPON  ENVNAME.
    ACCEPT  PROC_NUM  FROM  ENVVAL.
            :
```

8.3.3 注意事項

ワークユニットプロセス情報通知機能を使用する場合、次のことに注意してください。

- C、C++言語において獲得した環境変数領域は解放しないでください。
- 環境変数の値は変更しないでください。

8.4 ユーティリティワークユニットのプロセス停止出口機能の使用方法

ユーティリティワークユニットのプロセス停止出口機能の使用方法について説明します。
ユーティリティワークユニットはSolaris版、Linux版のみ使用することができます。

8.4.1 概要

プロセス停止出口プログラムは、ユーティリティワークユニットのプロセスを停止するための出口プログラムです。プロセス停止出口プログラムが設定されると、ユーティリティワークユニットの通常停止、強制停止およびワークユニット異常終了時のプロセス停止時に、起動されているプロセス単位にプロセス停止出口プログラムが呼び出されます。
プロセス停止出口プログラムでは、通知されるプロセス番号のプロセスを停止する必要があります。
プロセス停止出口プログラムが設定されていない場合は、ワークユニットを停止してもプロセスは停止しません。その場合、利用者が直接プロセスを停止する必要があります。
本機能は、ユーティリティワークユニットでのみ使用することができます。

8.4.2 プログラミングの流れ

以下に示す形式で作成してください。なお、プロセス停止出口の作成言語はC言語のみ有効です。

関数様式

[形式]

```
long 出口名 ( char      *wuname,
              char      *username,
              long      pid,
              char      *sysname,
              long      mode)
```

[引数]

パラメタ	設定内容	入出力
char *wuname	ワークユニット名	入力値
char *username	ワークユニット起動ユーザ名	入力値
long pid	停止するユーティリティワークユニットのプロセス番号	入力値
char *sysname	システム名(注)	入力値
long mode	停止モード	入力値

(注)

Solaris32

デフォルトシステムの場合は“default”が設定されます。拡張システムの場合は拡張システムのシステム名が設定されます。

Linux32

“default”が設定されます。

停止モード

停止モードでは、ユーティリティワークユニットの停止モードを通知します。以下のどちらかが設定されます。

状態	設定内容
ワークユニット通常停止	1
ワークユニット強制停止、および、 ワークユニット異常時のプロセス停止	2

プロセス停止出口プログラムでは、必要に応じて停止モードごとに処理を作成し、設定されている停止モードを判断して処理の振り分けを行ってください。

[復帰値]

復帰値	意味
0	正常復帰
0以外	異常復帰

出口プログラムが正常復帰する場合は、復帰値に0を設定し、異常の場合は0以外を設定してください。
ただし、プロセス停止出口が異常復帰した場合でも、メッセージのみ出力され動作は異常となりません。

プログラミング例

プロセス停止出口プログラムの記述例を以下に示します。

```
long stopexit( char     *wuname,
               char     *username,
               long      pid,
               char     *sysname,
               long      mode)
{

    /* 停止モードごとに処理を振り分ける */
    switch(mode){
      case    1:
        /* ワークユニット通常停止時の処理を記述 */
        /* アプリケーションの仕様にあわせて通知されたプロセスの通常停止処理を行います */
        break;

      case    2:
        /* ワークユニット強制停止時の処理を記述 */
```

```
        /* アプリケーションの仕様にあわせて通知されたプロセスの強制停止処理を行います */
        break;
    }

    return(0);
}
```

ワークユニット定義

ワークユニット定義の記述例を示します。

```
[WORK UNIT]
Name: …
Kind: UTY

[Control Option]
…
Executable File of Exit Program for Salvage: libexit.so      ← 実行ファイル名を指定
Maximum Processing Time for Exit Program: 60                 ← 出口プログラム最大処理時間
…

[Application Program]
…
Exit Program for Terminating Process: termexit               ← 出口プログラム名を指定
Executable File of Exit Program for Salvage: libexit.so      ← 実行ファイル名を指定
Maximum Processing Time for Exit Program: 60                 ← 出口プログラム最大処理時間
…
```

プロセス停止出口機能を使用するために必要なワークユニット定義を以下に示します。
[Control Option]セクションおよび[Application Program]セクションに以下の定義を設定します。

Exit Program for Terminating Process:プロセス停止出口プログラム名

プロセス停止出口プログラム名を設定します。
31バイト以内の英数字とアンダースコアが使用できます。

Executable File of Exit Program for Salvage:出口プログラム実行ファイル名

プロセス停止出口プログラムが格納されている実行ファイル名を指定します。
31バイト以内の空白文字と半角カナを除く文字が使用できます。
[Application Program]セクションに、"Exit Program for Process Salvage"ステートメントが設定された場合、本ステートメントを[Control Option]セクションまたは[Application Program]セクションのどちらかに設定する必要があります。なお、両方のセクションに設定された場合は、[Application Program]セクションの設定が有効となります。

Maximum Processing Time for Exit Program:出口プログラム最大処理時間

出口プログラムの最大処理時間の監視値(秒)を設定します。
1~1800の整数値。
本ステートメントが[Control Option]セクションおよび[Application Program]セクションの両方に設定された場合は、[Application Program]セクションの設定が有効となります。
本ステートメントは省略可能です。本ステートメントを省略した場合、省略値として300が設定されます。

標準出力および標準エラー出力

プロセス停止出口プログラムが動作する時のカレントディレクトリは、以下のような構成になっており、カレントディレクトリ配下に標準出力用のファイルと標準エラー出力用のファイルが設定されています。

標準出力には、stdoutファイルが割り当てられています。プロセス停止出口プログラムで標準出力にデータを出力した場合に、出力先として使用されます。
標準エラー出力には、stderrファイルが割り当てられています。プロセス停止出口プログラムで標準エラー出力にデータを出力した場合に、出力先として使用されます。

カレントディレクトリ：xxx/yyy/zzz_exit
xxx ：ワークユニット定義で指定されたディレクトリ
yyy ：当該ワークユニット名
zzz ：プロセス停止出口プログラムの実行プロセスid

Solaris32

なお、マルチシステム機能の拡張システムを使用している場合は、上記yyyは“当該ワークユニット名.拡張システム名”となります。

8.4.3 ソースのコンパイル・リンク

プロセス停止出口プログラムは、動的リンクライブラリの形式で作成してください。

Solaris32

また、コンパイルおよびリンクオプションに“-mt”を指定し、スレッド・セイフな形式で作成してください。

Linux32

また、コンパイルオプションに“-D-REENTRANT”を指定し、リンクオプションに“-lpthread”を指定して、スレッド・セイフな形式で作成してください。

8.4.4 注意事項

プロセス停止出口機能を使用する場合、次のことに注意してください。

- プロセス停止出口プログラムは、呼出し契機ごとに新規にプロセスが起動され呼び出されます。したがって、外部変数などを使用してアプリケーションプロセスからデータを引き継ぐことはできません。
- プロセス停止出口が復帰するまでワークユニット停止コマンドは終了しません。ループ防止のために、必ず、出口プログラム最大処理時間を設定してください。
- ワークユニット異常終了となる場合、すでに停止済のプロセスに対してもプロセス停止出口プログラムの呼び出しを行います。停止済のプロセスに対しても呼び出されても問題のないように、考慮する必要があります。

付録A　ワークユニット定義の記述形式

ここでは、以下に示す機能を使用する時に必要なワークユニット定義の設定について説明します。

- 常駐機能
- 非常駐機能
- マルチオブジェクト常駐機能
- 複数のリソースへのアクセス

A.1　常駐機能

[WORK UNIT]

定義名・種別	●：必須 ○：必要な時に指定
Name: ワークユニット名	●
Kind: ワークユニット種別	●

[APM]

APM名	●：必須 ○：必要な時に指定
Name: APM名	●

[Control Option]

制御オプション	●：必須 ○：必要な時に指定
Path: アプリケーションライブラリパス (注1)	●
Current Directory:カレントディレクトリ	●
Remove Directory: APM（サーバアプリケーション）のカレントディレクトリ削除の有無	○
Application Retry Count:連続異常終了回数	○
Snapshot:スナップショット取得指定	○
Path for Snapshot:スナップショット出力パス	○
Windows32 Path for Application:アプリケーション使用パス (注2)	○
Solaris32 Linux32 Library for Application:アプリケーション使用ライブラリパス (注2)	○
Environment Variable:環境変数 (注3)	○
Registration to Naming Service:ネーミングサービスの登録形態	○
Using Load Balance:ロードバランス機能使用の有無	○
Windows32 Solaris32 Using Notification of User Information:ユーザ識別情報の通知の有無	○
Windows32 Solaris32 Access Control:アクセス制御実施の有無	○
Windows32 Solaris32 Access Control Base DN: アクセス制御対象エントリの基点識別名	○
Maximum Processing Time for Exit Program:出口プログラム最大処理時間	○
WorkUnit Exit Program:ワークユニット出口プログラム名	○

制御オプション	●:必須 ○:必要な時に指定
Executable File of Exit Program for Salvage: プロセス回収出口プログラム実行ファイル名	○
Solaris32 Linux32 WorkUnit Automatic Stop Mode:自動停止モード	○

注1)
アプリケーションライブラリパスを複数定義する場合は、"Path:"ステートメントを繰り返し記述します。

注2)
アプリケーション使用ライブラリパスを複数定義する場合は、"Library for Application:"ステートメントを繰り返し記述します。

注3)
環境変数を複数定義する場合は、"Environment Variable:"ステートメントを繰り返し記述します。

[Application Program]

アプリケーション情報 (注1)	●:必須 ○:必要な時に指定
Destination:あて先名(オブジェクト名)	●
Destination Priority; オブジェクトの優先度	○
Executable File:実行ファイル名	●
Application Language:アプリケーション言語	●
Code Conversion For String:コード変換情報(string)	○
Concurrency:プロセス多重度	○
Maximum Processing Time:アプリケーション最大処理時間	○
Maximum Processing Time for Exit Program:出口プログラム最大処理時間	○
Maximum Queuing Message: 最大キューイング数	○
Queuing Message To Notify Alarm: 監視キューイング数	○
Queuing Message To Notify Resumption: 監視再開キューイング数	○
Environment Variable:環境変数 (注2)	○
Form:常駐、非常駐の形態	RESIDENTを指定する
Pre Exit Program:前出口プログラム	○
Post Exit Program:後出口プログラム	○
Recovery Exit Program: 異常出口プログラム	○
Executable File for Exit Program:出口プログラム実行ファイル	○
Windows32 Solaris32 Access Control: アクセス制御実施の有無	○
Windows32 Solaris32 Access Control Base DN: アクセス制御対象エントリの基点識別名	○
Windows32 Solaris32 Type of User Identification: ユーザ識別情報種別	○
Windows32 Solaris32 User Name Param:ユーザ名通知パラメタ	○
Windows32 Solaris32 User Base DN: ユーザ名検索基点識別名	○
Windows32 Solaris32 User DN Param:ユーザ識別名通知パラメタ	○

アプリケーション情報 (注1)	●:必須 ○:必要な時に指定
Windows32 Solaris32 Password Param:パスワード通知パラメタ	○
Bind Type:バインド形式	○
SessionID Param:セションID通知パラメタ	○
Method Name to Begin Session:セションを開始するメソッド(オペレーション)	○
Maximum Session Active Time for Client:クライアント思考時間の最大時間	○
Windows32 Solaris32 Maximum Processing Time for WRAPPER:AIMアプリケーション監視時間	○
Maximum Memory for EJB Application:EJBアプリケーション最大メモリ量	○
CLASSPATH for Application:アプリケーション使用クラスパス	○
Java Command Option:Javaコマンド指定オプション	○
Exit Program for Process Salvage:プロセス回収出口プログラム名	○
Executable File of Exit Program for Salvage:プロセス回収出口プログラム実行ファイル名	○

注1)
アプリケーション情報を複数定義する場合は、[Application Program] セクションを繰り返し記述します。

注2)
環境変数を複数定義する場合は、“Environment Variable:”ステートメントを繰り返し記述します。

[Resource Manager]

リソースマネージャ情報(注1)	●:必須 ○:必要な時に指定
Name:リソース定義名(注2)	○
RM:データベース・システムの名前(注3)	○

注1)
グローバルトランザクションを使用しない場合、[Resource Manager]セクションは省略することができます。

注2)
otssetrscを利用して登録したOTS用のリソース定義に記載されているリソース定義名(NAME)を指定します。

注3)
otssetrscを利用して登録したOTS用のリソース定義に記載されているリソースマネージャ名(RMNAME)を指定します。

A.2 非常駐機能

[WORK UNIT]

定義名・種別	●:必須 ○:必要な時に指定
Name: ワークユニット名	●
Kind: ワークユニット種別	●

[APM]

APM名	●:必須 ○:必要な時に指定
Name: APM名	●

[Control Option]

制御オプション	●：必須 ○：必要な時に指定
Path: アプリケーションライブラリパス (注1)	●
Current Directory:カレントディレクトリ	●
Remove Directory: APM（サーバアプリケーション）のカレントディレクトリ削除の有無	○
Application Retry Count:連続異常終了回数	○
Snapshot:スナップショット取得指定	○
Path for Snapshot:スナップショット出力パス	○
Windows32 Path for Application:アプリケーション使用パス (注2)	○
Solaris32 Linux32 Library for Application:アプリケーション使用ライブラリパス (注2)	○
Environment Variable:環境変数 (注3)	○
Registration to Naming Service:ネーミングサービスの登録形態	○
Using Load Balance:ロードバランス機能使用の有無	○
Windows32 Solaris32 Using Notification of User Information:ユーザ識別情報の通知の有無	○
Windows32 Solaris32 Access Control:アクセス制御実施の有無	○
Windows32 Solaris32 Access Control Base DN:アクセス制御対象エントリの基点識別名	○
Maximum Processing Time for Exit Program:出口プログラム最大処理時間	○
WorkUnit Exit Program:ワークユニット出口プログラム名	○
Executable File of Exit Program for Salvage: プロセス回収出口プログラム実行ファイル名	○
Solaris32 Linux32 WorkUnit Automatic Stop Mode:自動停止モード	○

注1)
アプリケーションライブラリパスを複数定義する場合は、"Path:"ステートメントを繰り返し記述します。

注2)
アプリケーション使用ライブラリパスを複数定義する場合は、"Library for Application:"ステートメントを繰り返し記述します。

注3)
環境変数を複数定義する場合は、"Environment Variable:"ステートメントを繰り返し記述します。

[Application Program] (注1)

アプリケーション情報	●：必須 ○：必要な時に指定
Destination:あて先名（オブジェクト名）	●
Destination Priority; オブジェクトの優先度	○
Executable File:実行ファイル名	●

アプリケーション情報	●：必須 ○：必要な時に指定
Application Language:アプリケーション言語	●
Code Conversion For String:コード変換情報(string)	○
Maximum Processing Time:アプリケーション最大処理時間	○
Maximum Queuing Message: 最大キューイング数	○
Queuing Message To Notify Alarm: 監視キューイング数	○
Queuing Message To Notify Resumption: 監視再開キューイング数	○
Environment Variable:環境変数 (注2)	○
Form:常駐、非常駐の形態	NONRESIDENTを指定する
Windows32 Solaris32 Access Control: アクセス制御実施の有無	○
Windows32 Solaris32 Access Control Base DN: アクセス制御対象エントリの基点識別名	○
Windows32 Solaris32 Type of User Identification: ユーザ識別情報種別	○
Windows32 Solaris32 User Name Param:ユーザ名通知パラメタ	○
Windows32 Solaris32 User Base DN: ユーザ名検索基点識別名	○
Windows32 Solaris32 User DN Param:ユーザ識別名通知パラメタ	○
Windows32 Solaris32 Password Param:パスワード通知パラメタ	○
Bind Type:バインド形式	○
SessionID Param:セションID通知パラメタ	○
Method Name to Begin Session:セションを開始するメソッド(オペレーション)	○
Maximum Session Active Time for Client:クライアント思考時間の最大時間	○
Windows32 Solaris32 Maximum Processing Time for WRAPPER:AIMアプリケーション監視時間	○
Maximum Memory for EJB Application:EJBアプリケーション最大メモリ量	○
CLASSPATH for Application:アプリケーション使用クラスパス	○
Java Command Option:Javaコマンド指定オプション	○
Exit Program for Process Salvage:プロセス回収出口プログラム名	○
Executable File of Exit Program for Salvage:プロセス回収出口プログラム実行ファイル名	○

注1)
アプリケーション情報を複数定義する場合は、[Application Program] セクションを繰り返し記述します。

注2)
環境変数を複数定義する場合は、"Environment Variable:"ステートメントを繰り返し記述します。

[Resource Manager]

リソースマネージャ情報(注1)	●:必須 ○:必要な時に指定
Name:リソース定義名(注2)	○
RM:データベース・システムの名前(注3)	○

注1)
グローバルトランザクションを使用しない場合、[Resource Manager]セクションは省略することができます。

注2)
otssetrscを利用して登録したOTS用のリソース定義に記載されているリソース定義名(NAME)を指定します。

注3)
otssetrscを利用して登録したOTS用のリソース定義に記載されているリソースマネージャ名(RMNAME)を指定します。

[Nonresident Application Process]

非常駐セクション	●:必須 ○:必要な時に指定
Concurrency:プロセス多重度	●
Pre Exit Program:前出口プログラム名	○
Post Exit Program:後出口プログラム名	○
Executable File for Exit Program:出口プログラム実行ファイル名	○
Dynamic Link Library:動的リンクライブラリ名	○
Maximum Processing Time for Exit Program:出口プログラム最大処理時間	○
Exit Program for Process Salvage:プロセス回収出口プログラム名	○
Executable File of Exit Program for Salvage:プロセス回収出口プログラム実行ファイル	○

A.3 マルチオブジェクト常駐機能

[WORK UNIT]

定義名・種別	●:必須 ○:必要な時に指定
Name: ワークユニット名	●
Kind: ワークユニット種別	●

[APM]

APM名	●:必須 ○:必要な時に指定
Name: APM名	●

[Control Option]

制御オプション	●:必須 ○:必要な時に指定
Path: アプリケーションライブラリパス (注1)	●
Current Directory:カレントディレクトリ	●

制御オプション	●:必須 ○:必要な時に指定
Remove Directory: APM(サーバアプリケーション)のカレントディレクトリ削除の有無	○
Application Retry Count:連続異常終了回数	○
Snapshot:スナップショット取得指定	○
Path for Snapshot:スナップショット出力パス	○
Windows32 Path for Application:アプリケーション使用パス (注2)	○
Solaris32 Linux32 Library for Application:アプリケーション使用ライブラリパス (注2)	○
Environment Variable:環境変数 (注3)	○
Registration to Naming Service:ネーミングサービスの登録形態	○
Using Load Balance:ロードバランス機能使用の有無	○
Windows32 Solaris32 Using Notification of User Information:ユーザ識別情報の通知の有無	○
Windows32 Solaris32 Access Control:アクセス制御実施の有無	○
Windows32 Solaris32 Access Control Base DN: アクセス制御対象エントリの基点識別名	○
Maximum Processing Time for Exit Program:出口プログラム最大処理時間	○
WorkUnit Exit Program:ワークユニット出口プログラム名	○
Executable File of Exit Program for Salvage: プロセス回収出口プログラム実行ファイル名	○
Solaris32 Linux32 WorkUnit Automatic Stop Mode:自動停止モード	○

注1)
アプリケーションライブラリパスを複数定義する場合は、“Path:”ステートメントを繰り返し記述します。

注2)
アプリケーション使用パスを複数定義する場合は、“Path for Application:”ステートメントを繰り返し記述します。

注3)
環境変数を複数定義する場合は、“Environment Variable:”ステートメントを繰り返し記述します。

[Application Program] (注1)

アプリケーション情報	●:必須 ○:必要な時に指定
Destination:あて先名(オブジェクト名)	●
Destination Priority; オブジェクトの優先度	○
Executable File:実行ファイル名	●
Application Language:アプリケーション言語	●
Code Conversion For String:コード変換情報(string)	○

アプリケーション情報	●:必須 ○:必要な時に指定
Maximum Processing Time:アプリケーション最大処理時間	○
Maximum Queuing Message: 最大キューイング数	○
Queuing Message To Notify Alarm: 監視キューイング数	○
Queuing Message To Notify Resumption: 監視再開キューイング数	○
Environment Variable:環境変数 (注2)	○
Form:常駐、非常駐の形態	MULTIRESIDENTを指定する
Windows32 Solaris32 Access Control: アクセス制御実施の有無	○
Windows32 Solaris32 Access Control Base DN: アクセス制御対象エントリの基点識別名	○
Windows32 Solaris32 Type of User Identification: ユーザ識別情報種別	○
Windows32 Solaris32 User Name Param:ユーザ名通知パラメタ	○
Windows32 Solaris32 User Base DN: ユーザ名検索基点識別名	○
Windows32 Solaris32 User DN Param:ユーザ識別名通知パラメタ	○
Windows32 Solaris32 Password Param:パスワード通知パラメタ	○
Bind Type:バインド形式	○
SessionID Param:セションID通知パラメタ	○
Method Name to Begin Session:セションを開始するメソッド(オペレーション)	○
Windows32 Solaris32 Maximum Session Active Time for Client:クライアント思考時間の最大時間	○
Maximum Processing Time for WRAPPER:AIMアプリケーション監視時間	○
Maximum Memory for EJB Application:EJBアプリケーション最大メモリ量	○
CLASSPATH for Application:アプリケーション使用クラスパス	○
Java Command Option:Javaコマンド指定オプション	○
Exit Program for Process Salvage:プロセス回収出口プログラム名	○
Executable File of Exit Program for Salvage:プロセス回収出口プログラム実行ファイル名	○

注1)

アプリケーション情報を複数定義する場合は、[Application Program] セクションを繰り返し記述します。

注2)
環境変数を複数定義する場合は、“Environment Variable:”ステートメントを繰り返し記述します。

[Resource Manager]

リソースマネージャ情報(注1)	●:必須 ○:必要な時に指定
Name:リソース定義名(注2)	○
RM:データベース・システムの名前(注3)	○

注1)
グローバルトランザクションを使用しない場合、[Resource Manager]セクションは省略することができます。

注2)
otssetrscを利用して登録したOTS用のリソース定義に記載されているリソース定義名(NAME)を指定します。

注3)
otssetrscを利用して登録したOTS用のリソース定義に記載されているリソースマネージャ名(RMNAME)を指定します。

[Multiresident Application Process]

マルチオブジェクト常駐セクション	●:必須 ○:必要な時に指定
Concurrency:プロセス多重度	●
Pre Exit Program:前出口プログラム名	○
Post Exit Program:後出口プログラム名	○
Executable File for Exit Program:出口プログラム実行ファイル名	○
Maximum Processing Time for Exit Program:出口プログラム最大処理時間	○
Recover Exit Program: 異常出口プログラム名	○
Exit Program for Process Salvage:プロセス回収出口プログラム名	○
Executable File of Exit Program for Salvage:プロセス回収出口プログラム実行ファイル	○

A.4 複数リソースへのアクセス

[WORK UNIT]

定義名・種別	●:必須、○:必要な時に指定
Name: ワークユニット名	●
Kind: ワークユニット種別	●

[APM]

APM名 tdlinkapmコマンドで作成したAPM名を指定する	●:必須、○:必要な時に指定
Name: APM名	●

[Control Option]

制御オプション	●:必須、○:必要な時に指定
Path:アプリケーションライブラリパス (注1)	●
Current Directory:カレントディレクトリ	○
Application Retry Count:連続異常終了回数	○
Snapshot:スナップショット取得指定	○

制御オプション	●:必須、○:必要な時に指定
Path for Snapshot:スナップショット出力パス	○
Windows32 Path for Application:アプリケーション使用パス (注2)	○
Solaris32 Linux32 Library for Application:アプリケーション使用ライブラリパス (注2)	○
Environment Variable:環境変数 (注3)	○
Resistration to Naming Service:ネーミングサービスの登録形態	○
Using Load Balance:ロードバランス機能使用の有無	○
Windows32 Solaris32 Using Notification of User Information:ユーザ識別情報の通知の有無	○
Windows32 Solaris32 Access Control:アクセス制御実施の有無	○
Windows32 Solaris32 Access Control Base DN: アクセス制御対象エントリの基点識別名	○
Maximum Processing Time for Exit Program:出口プログラム最大処理時間	○
WorkUnit Exit Program:ワークユニット出口プログラム名	○
Executable File of Exit Program for Salvage: プロセス回収出口プログラム実行ファイル名	○
Solaris32 Linux32 WorkUnit Automatic Stop Mode:自動停止モード	○

注1)
アプリケーションライブラリパスを複数定義する場合は、“Path”ステートメントを繰り返し記述します。

注2)
アプリケーション使用ライブラリパスを複数定義する場合は、“Library for Application:”ステートメントを繰り返し記述します。

注3)
環境変数を複数定義する場合は、“Environment Variable:”ステートメントを繰り返し記述します。

[Application Program] (注1)

アプリケーション情報	●:必須、○:必要な時に指定
Destination:あて先名(オブジェクト名)	●
Windows32 Solaris32 PSYS:DPCF通信パス名	○
Executable File:実行ファイル名	●
Application Language:アプリケーション言語	●
Code Conversion For String:コード変換情報(string)	○
Concurrency:プロセス多重度	○
Maximum Processing Time:アプリケーション最大処理時間	○
Maximum Processing Time for Exit Program:出口プログラム最大処理時間	○
Environment Variable:環境変数 (注2)	○
Form:常駐、非常駐の形態	○
Pre Exit Program:前出口プログラム	○
Post Exit Program:後出口プログラム	○

アプリケーション情報	●:必須、○:必要な時に指定
Executable File for Exit Program:出口プログラム実行ファイル	○
Windows32 Solaris32 Access Control: アクセス制御実施の有無	○
Windows32 Solaris32 Access Control Base DN: アクセス制御対象エントリの基点識別名	○
Windows32 Solaris32 Type of User Identification: ユーザ識別情報種別	○
Windows32 Solaris32 User Name Param:ユーザ名通知パラメタ	○
Windows32 Solaris32 User Base DN: ユーザ名検索基点識別名	○
Windows32 Solaris32 User DN Param:ユーザ識別名通知パラメタ	○
Windows32 Solaris32 Password Param:パスワード通知パラメタ	○
Bind Type:バインド形式	○
SessionID Param:セションID通知パラメタ	○
Method Name to Begin Session:セションを開始するメソッド(オペレーション)	○
Maximum Session Active Time for Client:クライアント思考時間の最大時間	○
Windows32 Solaris32 Maximum Processing Time for WRAPPER:AIMアプリケーション監視時間	○
Maximum Memory for EJB Application:EJBアプリケーション最大メモリ量	○
CLASSPATH for Application:アプリケーション使用クラスパス	○
Java Command Option:Javaコマンド指定オプション	○
Exit Program for Process Salvage:プロセス回収出口プログラム名	○
Executable File of Exit Program for Salvage:プロセス回収出口プログラム実行ファイル名	○

注1)
アプリケーション情報を複数定義する場合は、[Application Program] セクションを繰り返し記述します。

注2)
環境変数を複数定義する場合は、"Environment Variable:"ステートメントを繰り返し記述します。

[Resource Manager]

[Resource Manager]セクションはn個指定することができます。

リソースマネージャ情報	●:必須、○:必要な時に指定
Name:リソース定義名(注1)	●
RM:データベース・システムの名前(注2)	●

注1)
otssetrscを利用して登録したOTS用のリソース定義に記載されているリソース定義名(NAME)を指定します。

注2)

otssetrscを利用して登録したOTS用のリソース定義に記載されているリソースマネージャ名(RMNAME)を指定します。

[Nonresident Application Process] (注1)

非常駐セクション	●:必須、○:必要な時に指定
Concurrency:プロセス多重度	●
Pre Exit Program:前出口プログラム名	○
Post Exit Program:後出口プログラム名	○
Executable File for Exit Program: 出口プログラム実行ファイル名	○
Dynamic Link Library:動的リンクライブラリ名 (注2)	○
Maximum Processing Time for Exit Program:出口プログラム最大処理時間	○
Exit Program for Process Salvage:プロセス回収出口プログラム名	○
Executable File of Exit Program for Salvage:プロセス回収出口プログラム実行ファイル	○

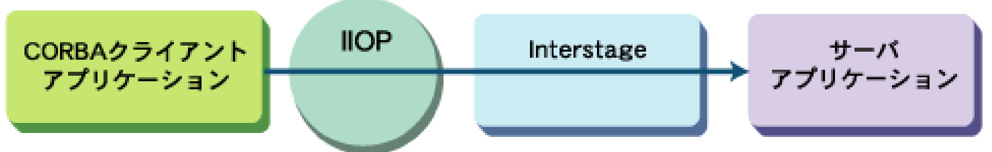
注1)

[Application Program]セクションの“Form:”ステートメントにNONRESIDENTを指定した場合には必須です。

注2)

動的リンクライブラリを複数定義する場合は、“Dynamic Link Library:”ステートメントを繰り返し記述します。

付録B　トランザクションアプリケーションのサンプルプログラム（基本編）



B.1　サーバアプリケーション

B.1.1　ファイル構成

本アプリケーションのファイル構成を以下に示します。

Solaris32 Linux32
なお、本サンプルプログラムにはスレッドモードとプロセスモードの2つの形態のアプリケーションがあります。

C言語サンプル

Windows32

No	ファイルの概要	ファイル名	格納パス
(3)	IDLファイル	tdsample1.idl	(インストールパスはデフォルト) C:¥Interstage¥td¥sample¥C_SERVER
(4)	サーバアプリケーションプログラム	tdsample1_s.c	
(5)	ワークユニット定義ファイル	tdsample1.wu	
(7)	プロジェクトファイル	C_Server.vcproj	
(8)	ソリューションファイル	C_Server.sln	

Solaris32 Linux32

No	ファイルの概要	ファイル名	格納パス
(1)	Makefile（スレッドモード）	Makefile	Solaris32 (インストールパスはデフォルト) /opt/FSUNtd/sample/C_SERVER Linux32 /opt/FJSVtd/sample/C_SERVER
(2)	Makefile（プロセスモード）	Makefile_nt	
(3)	IDLファイル	tdsample1.idl	
(4)	サーバアプリケーションプログラム	tdsample1_s.c	
(5)	ワークユニット定義ファイル（スレッドモード）	tdsample1.wu	
(6)	ワークユニット定義ファイル（プロセスモード）	tdsample1_nt.wu	

C++言語サンプル

Windows32

No	ファイルの概要	ファイル名	格納パス
(3)	IDLファイル	tdsample1.idl	(インストールパスはデフォルト) C:¥Interstage¥td¥sample¥CPP_SERVER
(4)	サーバアプリケーションプログラム	tdsample1_s.cpp	

No	ファイルの概要	ファイル名	格納パス
(5)	ワークユニット定義ファイル	tdsample1.wu	
(7)	プロジェクトファイル	Cpp_Server.vcproj	
(8)	ソリューションファイル	Cpp_Server.sln	

Solaris32 Linux32

No	ファイルの概要	ファイル名	格納パス
(1)	Makefile(スレッドモード)	Makefile	Solaris32 (インストールパスはデフォルト) /opt/FSUNtd/sample/CPP_SERVER Linux32 /opt/FJSVtd/sample/CPP_SERVER
(2)	Makefile(プロセスモード)	Makefile_nt	
(3)	IDLファイル	tdsample1.idl	
(4)	サーバアプリケーションプログラム	tdsample1_s.C	
(5)	ワークユニット定義ファイル(スレッドモード)	tdsample1.wu	
(6)	ワークユニット定義ファイル(プロセスモード)	tdsample1_nt.wu	

COBOLサンプル

Windows32

No	ファイルの概要	ファイル名	格納パス
(3)	IDLファイル	tdsample1.idl	(インストールパスはデフォルト) C:¥Interstage¥td¥sample¥SERVER
(4)	サーバアプリケーションプログラム	tdsample1_s.cbl	
(5)	ワークユニット定義ファイル	tdsample1.wu	
(7)	プロジェクトファイル	Server.prj	
(9)	翻訳オプションファイル	Server.cbi	

Solaris32

No	ファイルの概要	ファイル名	格納パス
(1)	Makefile(スレッドモード)	Makefile	(インストールパスはデフォルト) /opt/FSUNtd/sample/SERVER
(2)	Makefile(プロセスモード)	Makefile_nt	
(3)	IDLファイル	tdsample1.idl	
(4)	サーバアプリケーションプログラム	tdsample1_s.cbl	
(5)	ワークユニット定義ファイル(スレッドモード)	tdsample1.wu	
(6)	ワークユニット定義ファイル(プロセスモード)	tdsample1_nt.wu	

(1)Makefile(スレッドモード)

本アプリケーションのバイナリファイルをスレッドモードで生成するためのMakefileです。

環境に合わせて一部修正する必要があります。

Solaris32

なお、本Makefileは、COBOLの場合Sun日本語COBOL用コンパイル環境に加えてFujitsu PowerCOBOL97用コンパイル環境を、

C++の場合Sun WorkShop Compilers C++4.2環境に加えてSun WorkShop 5.0シリーズ、Forte Developer 6シリーズ、Sun ONE Studio 7シリーズ、Sun ONE Studio 8シリーズまたはSun Studio 9環境を用意しています。

(2)Makefile(プロセスモード)

本アプリケーションのバイナリファイルをプロセスモードで生成するためのMakefileです。

スレッドモードと同様に環境に合わせて一部修正する必要があります。

(3) IDLファイル

本サンプルが使用するIDLファイルです。

(4)サーバアプリケーションプログラム

本サーバアプリケーションをコンパイルすることにより、サーバアプリケーションとして利用できるようになります。

(5)ワークユニット定義ファイル

ワークユニット定義を行うための入力ファイルです。環境に合わせて一部修正する必要があります。
Solaris版、Linux版の場合はスレッドモードで行うための入力ファイルです。

(6)ワークユニット定義ファイル(プロセスモード)

ワークユニット定義をプロセスモードで行うための入力ファイルです。実行環境に合わせて一部修正する必要があります。

(7) プロジェクトファイル

Microsoft(R) Visual C++(R)またはCOBOL97上でサーバアプリケーションをコンパイルするためのプロジェクトファイルです。

(8) ソリューションファイル

Microsoft(R) Visual C++(R)上でプロジェクト構成を管理するためのファイルです。

(9) 翻訳オプションファイル

COBOL97でプロジェクト管理機能を使用してコンパイルするための翻訳オプション情報が格納されているファイルです。

B.1.2 コンパイルおよびワークユニット定義の登録

本アプリケーションをコンパイルする手順を説明します。
手順は、各言語共通です。
なお、本アプリケーションは任意のディレクトリに複写して使用してください。また、複写後の環境に合わせて各ファイルをカストマイズする必要があります。

Windows32

なお、本アプリケーションは、Microsoft(R) Visual C++(R)またはCOBOL97のプロジェクトを使用してコンパイルすることを前提としています。

COBOLのサンプルを使用する場合は、空白を含まないフォルダに複写してください。

以下の記号を使用して説明します。

- $CURRENTは、本アプリケーションで使用するファイルが存在するフォルダを示します。
- サーバアプリケーションのコンパイルはプロジェクトを使用して実施します。

```
>cd $CURRENT
>tdc -mc tdsample1.idl ---------------(a)
サーバアプリケーションのコンパイル  ---(b)
>isaddwudef tdsample1.wu           ---(c)
```

(a) IDLファイルをコンパイルします。

tdcコマンドを実行することで、サーバアプリケーションを作成するうえで必要なファイルが生成されます。サーバアプリケーションの開発言語によって作成されるファイルが異なるため、tdcコマンド実行時にサーバアプリケーションの開発言語を以下に示すオプションで指定してください。

- C言語：-mc
- C++言語：-mvcpp
- COBOL：-mcobol

なお、tdcコマンドはIDL定義をシステムに登録する機能も備えているため、すでにIDL定義が登録されている場合、tdcコマンドが異常終了します。その場合は、以下のオプションを指定してください。

```
-update
```

例：C言語の場合

```
>tdc -mc -update tdsample1.idl
```

(b) サーバアプリケーションをコンパイルします。

サーバアプリケーションのコンパイルはMicrosoft(R) Visual C++(R) または COBOL97のプロジェクトを使用して行います。コンパイルはプロジェクトのビルドを実行することにより、サーバアプリケーションの作成に必要なファイルのコンパイル、リンクを行います。
コンパイル時の注意事項については、"B.1.4 注意点"を参照してください。

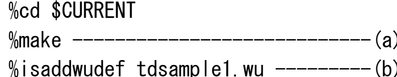
また、IDLファイルの内容を変更した場合、再度 (a)のIDLファイルのコンパイルを実施した後、サーバアプリケーションのリコンパイルを実施する必要があります。

プロジェクトのビルドが正常に終了した場合、プロジェクトファイルと同じフォルダに、libtdsample1.dll、tdsample1_s.exeまたはTDSAMPLE1-INTF.dllが作成されます。

(c) ワークユニットを登録します。

isaddwudefコマンドを実行することにより、ワークユニット定義を登録します。
ワークユニット定義がすでに登録されている場合には、以下のコマンドを実行してください。

```
>isaddwudef -o tdsample1.wu
```



以下の記号を使用して説明します。

- "%"は、一般ユーザ時のプロンプトを示します。
- $CURRENTは、本アプリケーションで使用するファイルが存在するディレクトリを示します。

スレッドモード

```
%cd $CURRENT
%make ----------------------------(a)
%isaddwudef tdsample1.wu ---------(b)
```

プロセスモード

```
%cd $CURRENT
%make -f Makefile_nt --------------(a)
%isaddwudef tdsample1_nt.wu -------(b)
```

(a) makeコマンドを実行します。

makeコマンドを実行することにより、tdcコマンドの実行、および、サーバアプリケーションの作成に必要なファイルのコンパイル、リンクを行います。

Solaris32

COBOLのプロセスモードの場合、｢nt｣ディレクトリが作成され、｢nt｣ディレクトリ配下に中間ファイルが出力されるようになっています。

makeコマンドが正常に終了した場合、makeコマンドを実行したカレントディレクトリに、以下のファイルが作成されます。

言語	スレッドモード	プロセスモード
C	libtdsample1.so	libtdsample1_nt.so
C++	tdsample1_s	tdsample1_s_nt
Solaris32 COBOL	libtdsample1.so	libtdsample1_nt.so

makeコマンドが失敗する場合には、以下のコマンドを実行してください。

```
%make clean                    （スレッドモード）
%make -f Makefile_nt clean       （プロセスモード）
```

注意

Solaris32

Sun Studio 12 Update 1以降を使用してサンプルプログラムを翻訳する場合は、Makefileのccコマンドのパスを修正してください。
コンパイラのインストール先が｢/opt/sunstudio12.1｣である場合のMakefileの修正例を以下に示します。

直接指定する場合

```
CC = /opt/sunstudio12.1/bin/cc
```

環境変数PATHに指定する場合

```
CC = cc
```

(b) isaddwudefコマンドを実行します。

isaddwudefコマンドを実行することにより、ワークユニット定義を登録します。
ワークユニット定義がすでに登録されている場合には、以下のコマンドを実行してください。

```
%isaddwudef -o tdsample1.wu        （スレッドモード）
%isaddwudef -o tdsample1_nt.wu      （プロセスモード）
```

B.1.3　ワークユニットの起動

ワークユニットの起動方法を以下に示します。

- ワークユニットの起動は、事前にInterstageが起動されている必要があります。

```
%isstartwu TDSAMPLE1
```

以上の手順により、本アプリケーションはサーバアプリケーションとしてクライアントアプリケーションと通信可能となります。クライアントアプリケーションの使用方法は、“B.2 CORBAクライアントアプリケーション”を参照してください。

B.1.4　注意点

本アプリケーションが動作している場合、“付録D トランザクションアプリケーションのサンプルプログラム(サーバアプリケーション間連携編)”に関する作業を一切行わないでください。

■Microsoft(R) Visual C++(R)でコンパイルする場合の注意点 Windows32

- 本サンプルプログラムのプロジェクトファイルは、インストールパスがデフォルトの場合(C:¥Interstage)を想定して提供しています。デフォルト以外のフォルダにインストールした場合は、FileViewのLibrary Filesを変更する必要があります。
- 参照するインクルードファイルを追加する必要があります。
  ツール(T)－オプション(O)のディレクトリを選択し、表示するディレクトリ(S):インクルードファイルに標準設定されているフォルダに加え、以下のフォルダを追加してください。

```
Interstageインストールフォルダ¥td¥include
Interstageインストールフォルダ¥odwin¥include
Interstageインストールフォルダ¥extp¥include
```

■COBOL97でコンパイルする場合の注意点 Windows32

本サンプルプログラムのプロジェクトファイルは、インストールパスがデフォルトの場合(C:¥Interstage)を想定して提供しています。デフォルト以外のフォルダにインストールした場合は、ライブラリファイルおよび登録集格納先のフォルダを変更する必要があります。

■COBOLコンパイラ変更時の注意点 Solaris32

ワークユニット定義ファイルにおいてアプリケーション使用ライブラリパスにCOBOLランタイムライブラリのパスを指定しているため、コンパイラの変更に合わせてライブラリパスを変更する必要があります。

■Linux for Intel64(32ビット互換)でコンパイルする場合の注意点 Linux32

gcc/g++コマンド実行時に「-m32 -mtune=i386」オプションを指定する必要があります。MakefileまたはMakefile_ntのgcc/g++コマンドに指定するオプションを修正してください。

B.2 CORBAクライアントアプリケーション

B.2.1 ファイル構成

本アプリケーションのファイル構成を以下に示します。

Windows32

No	ファイルの概要	ファイル名	格納パス
(2)	IDLファイル	tdsample1.idl	(インストールパスはデフォルト) C:¥Interstage¥td¥sample ¥CORBA_CL
(3)	クライアントアプリケーションプログラム	tdsample1_c.c	
(4)	プロジェクトファイル	Corba_CL.vcproj	
(5)	ソリューションファイル	Corba_CL.sln	

Solaris32 Linux32

No	ファイルの概要	ファイル名	格納パス
(1)	Makefile	Makefile	Solaris32 (インストールパスはデフォルト) /opt/FSUNtd/sample/CORBA_CL Linux32 /opt/FJSVtd/sample/CORBA_CL
(2)	IDLファイル	tdsample1.idl	
(3)	クライアントアプリケーションプログラム	tdsample1_c.c	

(1)Makefile

本アプリケーションのバイナリファイルを生成するためのMakefileです。環境に合わせて一部修正する必要があります。

(2)IDLファイル

本サンプルで使用するIDLファイルです。

(3)クライアントアプリケーションプログラム

本クライアントアプリケーションをコンパイルすることにより、サーバアプリケーションと通信を行うことができます。

(4) プロジェクトファイル

Microsoft(R) Visual C++(R)上でサーバアプリケーションをコンパイルするためのプロジェクトファイルです。

(5) ソリューションファイル

Microsoft(R) Visual C++(R)上でプロジェクト構成を管理するためのファイルです。

B.2.2 コンパイル

本アプリケーションをコンパイルする手順を説明します。
本アプリケーションは任意のディレクトリに複写して使用してください。なお、複写後の環境に合わせて各ファイルをカスタマイズする必要があります。

以下の記号を使用して説明します。

・ $CURRENTは、本アプリケーションで使用するファイルが存在するフォルダを示します。

Windows32

なお、本アプリケーションは、Microsoft(R) Visual C++(R)上のプロジェクトを使用してコンパイルすることを前提としています。

クライアントアプリケーションのコンパイルはプロジェクトを使用して実施します。

```
>cd $CURRENT
>tdc -c tdsample1.idl ----------------------(a)

 クライアントアプリケーションのコンパイル  ---(b)
```

(a) IDLファイルをコンパイルします。

tdcコマンドを実行することで、クライアントアプリケーションを作成するうえで必要なファイルが生成されます。
なお、tdcコマンドはIDL定義をシステムに登録する機能も備えているため、すでにIDL定義が登録されている場合、tdcコマンドが異常終了します。その場合は、以下のオプションを指定してください。

```
-update
```

例：C言語の場合

```
>tdc -c -update tdsample1.idl
```

(b) クライアントアプリケーションをコンパイルします。

クライアントアプリケーションのコンパイルはMicrosoft(R) Visual C++(R) のプロジェクトを使用して行います。コンパイルはプロジェクトのビルドを実行することにより、クライアントアプリケーションの作成に必要なファイルのコンパイル、リンクを行います。
また、IDLファイルの内容を変更した場合、再度(a)のIDLファイルのコンパイルを実施した後、クライアントアプリケーションのリコンパイルを実施する必要があります。
プロジェクトのビルドが正常に終了した場合、プロジェクトファイルと同じフォルダに、tdsample1_c.exeが作成されます。

Solaris32 Linux32

```
%cd $CURRENT
%make ---------------(a)
```

$CURRENTは、本アプリケーションで使用するファイルが存在するディレクトリを示します。

(a) makeコマンドを実行します。

makeコマンドを実行することにより、tdcコマンドの実行、および、クライアントアプリケーションの作成に必要なファイルのコンパイル、リンクを行います。
makeコマンドが正常に終了した場合、makeコマンドを実行したカレントディレクトリに、tdsample1_c が作成されます。makeが失敗する場合には、以下のコマンドを実行してください。

```
%make clean
```

Solaris32
Sun Studio 12 Update 1以降を使用してサンプルプログラムを翻訳する場合は、Makefileのccコマンドのパスを修正してください。
コンパイラのインストール先が「/opt/sunstudio12.1」である場合のMakefileの修正例を以下に示します。

直接指定する場合

```
CC = /opt/sunstudio12.1/bin/cc
```

環境変数**PATH**に指定する場合

```
CC = cc
```

B.2.3 クライアントアプリケーションの実行

本クライアントアプリケーションの実行手順を以下に示します。

Windows32
本クライアントアプリケーションを実行することにより、サーバアプリケーションと通信を行います。

Solaris32 Linux32
本クライアントアプリケーションを実行することにより、スレッドモード／プロセスモードのサーバアプリケーションと通信を行います。

```
%cd $CURRENT
%tdsample1_c
```

$CURRENTは、本アプリケーションで使用するファイルが存在するディレクトリを示します。

B.2.4 注意点

本アプリケーションが動作している場合、"付録D トランザクションアプリケーションのサンプルプログラム(サーバアプリケーション間連携編)"に関する作業を一切行わないでください。

■Microsoft(R) Visual C++(R)でコンパイルする場合の注意点 Windows32

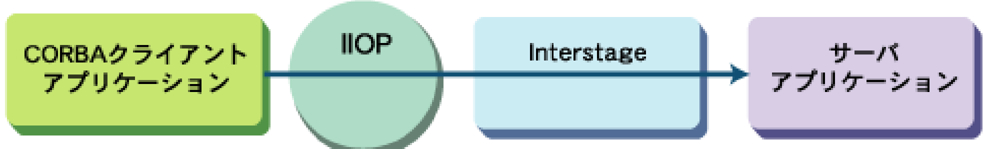
- 本サンプルプログラムのプロジェクトファイルは、インストールパスがデフォルトの場合(C:¥Interstage)を想定して提供しています。デフォルト以外のフォルダにインストールした場合は、FileViewのLibrary Filesを変更する必要があります。
- 参照するインクルードファイルを追加する必要があります。
  ツール(T)－オプション(O)のディレクトリを選択し、表示するディレクトリ(S):インクルードファイルに標準設定されているフォルダに加え、下記のフォルダを追加してください。

```
Interstageインストールフォルダ¥td¥include
Interstageインストールフォルダ¥odwin¥include
Interstageインストールフォルダ¥extp¥include
```

■Linux for Intel64(32ビット互換)でコンパイルする場合の注意点 Linux32

gcc/g++コマンド実行時に｢-m32 -mtune=i386｣オプションを指定する必要があります。MakefileまたはMakefile_ntのgcc/g++コマンドに指定するオプションを修正してください。

付録C　トランザクションアプリケーションのサンプルプログラム(各種データ型編)



C.1　概要

本サンプルは、各種データ型を操作するための機能に特化したものです。本機能に影響のない異常時の対応等の処理は簡略化されています。
以下に提供されているデータ型を示します。

- long型
- string型
- sequence型
- exception型

本サンプルを使用するにあたり、提供ファイルを任意のディレクトリに複写し、複写先の環境に合わせて各ファイルをカスタマイズすることをお勧めします。

Windows32

なお、COBOLのサンプルを使用する場合は、空白を含まないフォルダに複写してください。

説明にあたり、コマンド等の例題には、以下の記号を使用して説明します。

- 例題はlong型を対象に記述しています。
- $CURRENTは、本アプリケーションで使用する機能ごとのディレクトリを示します。

Windows32

- ‘コンパイルする’は各プロジェクトファイルを使用して実施します。

Solaris32　Linux32

- “%”は、一般ユーザ時のプロンプトを示します。

C.2　ファイル構成

プログラムなどの資産は、以下のディレクトリ配下に格納されています。

Windows32 (インストールパスはデフォルト)

```
C:¥Interstage¥td¥sample¥data
```

Solaris32 (インストールパスはデフォルト)

```
/opt/FSUNtd/sample/data（注）
```

Linux32

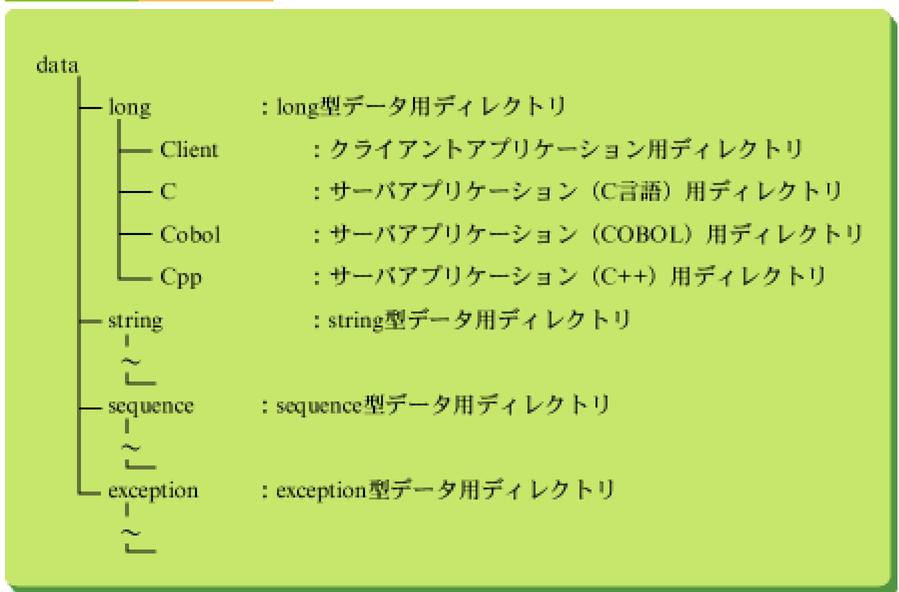
```
/opt/FJSVtd/sample/data（注）
```

注）サーバアプリケーションプログラムには、スレッドモードとプロセスモードの2種類の形態のアプリケーションがあります。

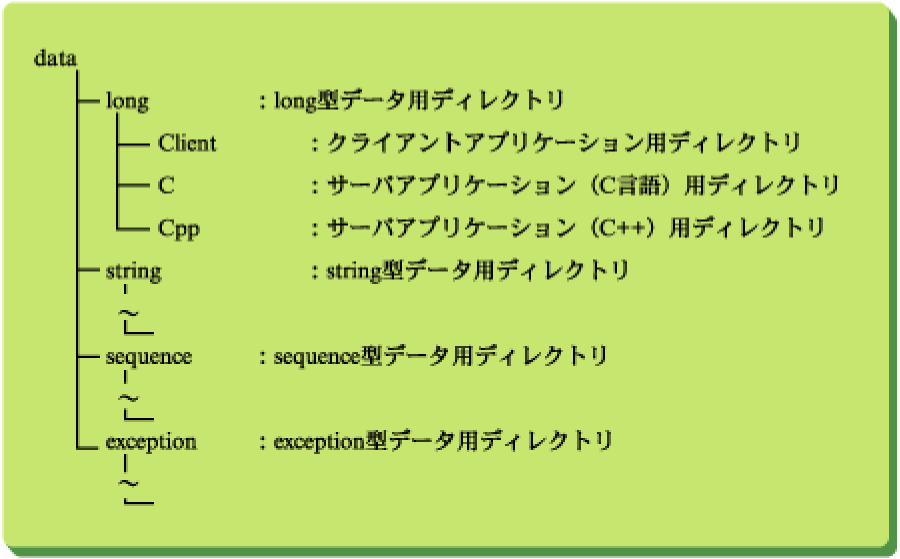
C.2.1 ディレクトリ構成

本サンプルのディレクトリ構成を以下に示します。

Windows32 Solaris32



Linux32



各ディレクトリには、プログラムやワークユニット定義ファイルなど、アプリケーションごとに必要な資産が格納されています。
また、「long」等のデータ型用ディレクトリ直下には、クライアント、サーバを問わず、各アプリケーションに関連のある共通ファイルが格納されています。

C.2.2 ファイル構成

本サンプルのファイル構成を以下に示します。

共通ファイル

本サンプルの共通ファイルの構成を以下に示します。

ディレクトリ	No	ファイル名	ファイルの概要
long	(1)	typlng.idl	IDLファイル
string	(1)	typstr.idl	IDLファイル
sequence	(1)	typseq.idl	IDLファイル
exception	(1)	typexp.idl	IDLファイル

クライアントアプリケーション用ディレクトリ

本ディレクトリには、クライアントアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Windows32

ディレクトリ	No	ファイル名	ファイルの概要
long	(4)	typlng_c.c	クライアントアプリケーションプログラム
	(9)	Cl_typlng.vcproj	プロジェクトファイル
	(10)	Cl_typlng.sln	ソリューションファイル
string	(4)	typstr_c.c	クライアントアプリケーションプログラム
	(9)	Cl_typstr.vcproj	プロジェクトファイル
	(10)	Cl_typstr.sln	ソリューションファイル
sequence	(4)	typseq_c.c	クライアントアプリケーションプログラム
	(9)	Cl_typseq.vcproj	プロジェクトファイル
	(10)	Cl_typseq.sln	ソリューションファイル
exception	(4)	typexp_c.c	クライアントアプリケーションプログラム
	(9)	Cl_typexp.vcproj	プロジェクトファイル
	(10)	Cl_typexp.sln	ソリューションファイル

Solaris32 Linux32

ディレクトリ	No	ファイル名	ファイルの概要
long	(2)	Makefile	Makefile
	(4)	typlng_c.c	クライアントアプリケーションプログラム
string	(2)	Makefile	Makefile
	(4)	typstr_c.c	クライアントアプリケーションプログラム
sequence	(2)	Makefile	Makefile
	(4)	typseq_c.c	クライアントアプリケーションプログラム
exception	(2)	Makefile	Makefile
	(4)	typexp_c.c	クライアントアプリケーションプログラム

サーバアプリケーション(C言語)用ディレクトリ

本ディレクトリには、開発言語がC言語のサーバアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Solaris32 Linux32

なお、本サンプルプログラムにはスレッドモードとプロセスモードの2種類の形態のアプリケーションがあります。

Windows32

ディレクトリ	No	ファイル名	ファイルの概要
long	(5)	typlng_s.c	サーバアプリケーションプログラム
	(7)	typlng.wu	ワークユニット定義ファイル(スレッドモード)
	(9)	C_typlng.vcproj	プロジェクトファイル
	(10)	C_typlng.sln	ソリューションファイル
string	(5)	typstr_s.c	サーバアプリケーションプログラム
	(7)	typstr.wu	ワークユニット定義ファイル(スレッドモード)
	(9)	C_typstr.vcproj	プロジェクトファイル
	(10)	C_typstr.sln	ソリューションファイル
sequence	(5)	typseq_s.c	サーバアプリケーションプログラム
	(7)	typseq.wu	ワークユニット定義ファイル(スレッドモード)
	(9)	C_typseq.vcproj	プロジェクトファイル
	(10)	C_typseq.sln	ソリューションファイル
exception	(5)	typexp_s.c	サーバアプリケーションプログラム
	(6)	typexp_pre.c	前出口プログラム
	(7)	typexp.wu	ワークユニット定義ファイル(スレッドモード)
	(9)	C_typexp.vcproj	プロジェクトファイル
	(10)	C_typexp.sln	ソリューションファイル

Solaris32 Linux32

ディレクトリ	No	ファイル名	ファイルの概要
long	(2)	Makefile	Makefile(スレッドモード)
	(3)	Makefile_nt	Makefile(プロセスモード)
	(5)	typlng_s.c	サーバアプリケーションプログラム
	(7)	typlng.wu	ワークユニット定義ファイル(スレッドモード)
	(8)	typlng_nt.wu	ワークユニット定義ファイル(プロセスモード)
string	(2)	Makefile	Makefile(スレッドモード)
	(3)	Makefile_nt	Makefile(プロセスモード)
	(5)	typstr_s.c	サーバアプリケーションプログラム
	(7)	typstr.wu	ワークユニット定義ファイル(スレッドモード)
	(8)	typstr_nt.wu	ワークユニット定義ファイル(プロセスモード)
sequence	(2)	Makefile	Makefile(スレッドモード)
	(3)	Makefile_nt	Makefile(プロセスモード)
	(5)	typseq_s.c	サーバアプリケーションプログラム
	(7)	typseq.wu	ワークユニット定義ファイル(スレッドモード)
	(8)	typseq_nt.wu	ワークユニット定義ファイル(プロセスモード)
exception	(2)	Makefile	Makefile(スレッドモード)
	(3)	Makefile_nt	Makefile(プロセスモード)
	(5)	typexp_s.c	サーバアプリケーションプログラム
	(6)	typexp_pre.c	前出口プログラム

ディレクトリ	No	ファイル名	ファイルの概要
	(7)	typexp.wu	ワークユニット定義ファイル(スレッドモード)
	(8)	typexp_nt.wu	ワークユニット定義ファイル(プロセスモード)

サーバアプリケーション(C++)用ディレクトリ

本ディレクトリには、開発言語がC++のサーバアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Solaris32 Linux32

なお、本サンプルプログラムにはスレッドモードとプロセスモードの、2種類の形態のアプリケーションがあります。

Windows32

ディレクトリ	No	ファイル名	ファイルの概要
long	(5)	typlng_s.cpp	サーバアプリケーションプログラム
	(7)	typlng.wu	ワークユニット定義ファイル(スレッドモード)
	(9)	Cpp_typlng.vcproj	プロジェクトファイル
	(10)	Cpp_typlng.sln	ソリューションファイル
string	(5)	typstr_s.cpp	サーバアプリケーションプログラム
	(7)	typstr.wu	ワークユニット定義ファイル(スレッドモード)
	(9)	Cpp_typstr.vcproj	プロジェクトファイル
	(10)	Cpp_typstr.sln	ソリューションファイル
sequence	(5)	typseq_s.cpp	サーバアプリケーションプログラム
	(7)	typseq.wu	ワークユニット定義ファイル(スレッドモード)
	(9)	Cpp_typseq.vcproj	プロジェクトファイル
	(10)	Cpp_typseq.sln	ソリューションファイル
exception	(5)	typexp_s.cpp	サーバアプリケーションプログラム
	(7)	typexp.wu	ワークユニット定義ファイル(スレッドモード)
	(9)	Cpp_typexp.vcproj	プロジェクトファイル
	(10)	Cpp_typexp.sln	ソリューションファイル

Solaris32 Linux32

ディレクトリ	No	ファイル名	ファイルの概要
long	(2)	Makefile	Makefile(スレッドモード)
	(3)	Makefile_nt	Makefile(プロセスモード)
	(5)	typlng_s.C	サーバアプリケーションプログラム
	(7)	typlng.wu	ワークユニット定義ファイル(スレッドモード)
	(8)	typlng_nt.wu	ワークユニット定義ファイル(プロセスモード)
string	(2)	Makefile	Makefile(スレッドモード)
	(3)	Makefile_nt	Makefile(プロセスモード)
	(5)	typstr_s.C	サーバアプリケーションプログラム
	(7)	typstr.wu	ワークユニット定義ファイル(スレッドモード)
	(8)	typstr_nt.wu	ワークユニット定義ファイル(プロセスモード)
sequence	(2)	Makefile	Makefile(スレッドモード)

ディレクトリ	No	ファイル名	ファイルの概要
	(3)	Makefile_nt	Makefile（プロセスモード）
	(5)	typseq_s.C	サーバアプリケーションプログラム
	(7)	typseq.wu	ワークユニット定義ファイル（スレッドモード）
	(8)	typseq_nt.wu	ワークユニット定義ファイル（プロセスモード）
exception	(2)	Makefile	Makefile（スレッドモード）
	(3)	Makefile_nt	Makefile（プロセスモード）
	(5)	typexp_s.C	サーバアプリケーションプログラム
	(7)	typexp.wu	ワークユニット定義ファイル（スレッドモード）
	(8)	typexp_nt.wu	ワークユニット定義ファイル（プロセスモード）

サーバアプリケーション（COBOL）用ディレクトリ Windows32 Solaris32

本ディレクトリには、開発言語がCOBOLのサーバアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Solaris32

なお、本サンプルプログラムにはスレッドモードとプロセスモードの、2種類の形態のアプリケーションがあります。

Windows32

ディレクトリ	No	ファイル名	ファイルの概要
long	(5)	typlng_s.cbl	サーバアプリケーションプログラム
	(7)	typlng.wu	ワークユニット定義ファイル（スレッドモード）
	(11)	Cbl_typlng.prj	プロジェクトファイル
	(12)	Cbl_typlng.cbi	翻訳オプションファイル
string	(5)	typstr10_s.cbl	サーバアプリケーションプログラム
	(5)	typstr20_s.cbl	サーバアプリケーションプログラム
	(7)	typstr.wu	ワークユニット定義ファイル（スレッドモード）
	(11)	Cbl_typstr.prj	プロジェクトファイル
	(12)	Cbl_typstr.cbi	翻訳オプションファイル
sequence	(5)	LG0010_s.cbl	サーバアプリケーションプログラム
	(5)	LG0020_s.cbl	サーバアプリケーションプログラム
	(5)	SG0010_s.cbl	サーバアプリケーションプログラム
	(5)	SG0020_s.cbl	サーバアプリケーションプログラム
	(5)	SG0030_s.cbl	サーバアプリケーションプログラム
	(5)	SG0040_s.cbl	サーバアプリケーションプログラム
	(7)	typseq.wu	ワークユニット定義ファイル（スレッドモード）
	(11)	Cbl_typseq.prj	プロジェクトファイル
	(12)	Cbl_typseq.cbi	翻訳オプションファイル
exception	(5)	typexp_s.cbl	サーバアプリケーションプログラム
	(6)	typexp_pre.cbl	前出口プログラム
	(7)	typexp.wu	ワークユニット定義ファイル（スレッドモード）
	(11)	Cbl_typexp.prj	プロジェクトファイル

ディレクトリ	No	ファイル名	ファイルの概要
	(12)	Cbl_typexp.cbi	翻訳オプションファイル

Solaris32

ディレクトリ	No	ファイル名	ファイルの概要
long	(2)	Makefile	Makefile(スレッドモード)
	(3)	Makefile_nt	Makefile(プロセスモード)
	(5)	typlng_s.cbl	サーバアプリケーションプログラム
	(7)	typlng.wu	ワークユニット定義ファイル(スレッドモード)
	(8)	typlng_nt.wu	ワークユニット定義ファイル(プロセスモード)
string	(2)	Makefile	Makefile(スレッドモード)
	(3)	Makefile_nt	Makefile(プロセスモード)
	(5)	typstr10_s.cbl	サーバアプリケーションプログラム
	(5)	typstr20_s.cbl	サーバアプリケーションプログラム
	(7)	typstr.wu	ワークユニット定義ファイル(スレッドモード)
	(8)	typstr_nt.wu	ワークユニット定義ファイル(プロセスモード)
sequence	(2)	Makefile	Makefile(スレッドモード)
	(3)	Makefile_nt	Makefile(プロセスモード)
	(5)	LG0010_s.cbl	サーバアプリケーションプログラム
	(5)	LG0020_s.cbl	サーバアプリケーションプログラム
	(5)	SG0010_s.cbl	サーバアプリケーションプログラム
	(5)	SG0020_s.cbl	サーバアプリケーションプログラム
	(5)	SG0030_s.cbl	サーバアプリケーションプログラム
	(5)	SG0040_s.cbl	サーバアプリケーションプログラム
	(7)	typseq.wu	ワークユニット定義ファイル(スレッドモード)
	(8)	typseq_nt.wu	ワークユニット定義ファイル(プロセスモード)
exception	(2)	Makefile	Makefile(スレッドモード)
	(3)	Makefile_nt	Makefile(プロセスモード)
	(5)	typexp_s.cbl	サーバアプリケーションプログラム
	(6)	typexp_pre.cbl	前出口プログラム
	(7)	typexp.wu	ワークユニット定義ファイル(スレッドモード)
	(8)	typexp_nt.wu	ワークユニット定義ファイル(プロセスモード)

(1) IDLファイル

本サンプルで使用するIDLファイルです。クライアントアプリケーション、サーバアプリケーション共通のファイルになります。

(2)Makefile(スレッドモード) Solaris32 Linux32

本アプリケーションのバイナリファイルをスレッドモードで生成するためのMakefileです。

環境に合わせて一部修正する必要があります。

Solaris32

なお、本Makefileは、COBOLの場合Sun日本語COBOL用コンパイル環境に加えてFujitsu PowerCOBOL97用コンパイル環境を、

C++の場合Sun WorkShop Compilers C++4.2環境に加えてSun WorkShop 5.0シリーズ、Forte Developer 6シリーズ、Sun ONE Studio 7シリーズ、Sun ONE Studio 8シリーズまたはSun Studio 9環境を用意しています。

(3)Makefile(プロセスモード) Solaris32 Linux32

本アプリケーションのバイナリファイルをプロセスモードで生成するためのMakefileです。

スレッドモードと同様に環境に合わせて一部修正する必要があります。

(4)クライアントプログラム

本プログラムをコンパイルすることにより、クライアントアプリケーションとして利用できるようになります。

(5)サーバアプリケーションプログラム

本プログラムをコンパイルすることにより、サーバアプリケーションとして利用できるようになります。

(6) 前出口プログラム

ワークユニット起動時に呼び出される前処理出口のアプリケーションです。CORBAサービスに対して以下の処理を行います。
・ORBの初期化
・BOAの初期化

(7)ワークユニット定義ファイル(スレッドモード)

サーバアプリケーション用のワークユニット定義を行うための入力ファイルです。実行環境に合わせて一部修正する必要があります。
Solaris版、Linux版の場合はスレッドモードで行うための入力ファイルです。

(8)ワークユニット定義ファイル(プロセスモード) Solaris32 Linux32

サーバアプリケーション用のワークユニット定義をプロセスモードで行うための入力ファイルです。実行環境に合わせて一部修正する必要があります。

(9) C言語／C++コンパイル用プロジェクトファイル Windows32

Microsoft(R) Visual C++(R)以降でクライアント／サーバアプリケーションをコンパイルするためのプロジェクトファイルです。

(10) ソリューションファイル Windows32

Microsoft(R) Visual C++(R)以降でプロジェクト構成を管理するためのファイルです。

(11) COBOLコンパイル用プロジェクトファイル Windows32

COBOL97 V50L10以降でサーバアプリケーションをコンパイルするためのプロジェクトファイルです。

(12) COBOL翻訳オプションファイル Windows32

COBOL97 V50L10以降でプロジェクト管理機能を使用してコンパイルするための翻訳オプション情報が格納されているファイルです。

C.3 アプリケーションのコンパイル

クライアントおよびサーバのアプリケーションをコンパイルする手順を説明します。

Windows32

なお、本アプリケーションは、Microsoft(R) Visual C++(R)、またはCOBOL97 のプロジェクトを使用してコンパイルすることを前提としています。

アプリケーションのコンパイルは以下の手順で実施します。

1. IDL定義ファイルのコンパイル
2. アプリケーションのコンパイル

Windows32

```
>cd $CURRENT¥long

>tdc -c -mc typlng.idl     ----------(1)
アプリケーションのコンパイル          --(2)
```

Solaris32 Linux32

例：Cの場合

```
%cd $CURRENT/long

%tdc -c -mc typlng.idl    ----------(1)
%cd $CURRENT/long/C

%make    --------------------------(2)
```

なお、本サンプルのアプリケーション用Makefileは、IDL定義ファイルのコンパイルも実施するように構成されてます。

C.3.1 IDL定義ファイルのコンパイル

IDL定義ファイルからプログラム作成に必要なスタブ／スケルトンファイルを生成します。スタブ・スケルトンファイルは、IDLコンパイラにより生成されます。

本サンプルの場合、トランザクションアプリケーション用IDLコンパイラであるtdcコマンドでIDL定義ファイルをコンパイルしてください。

なお、アプリケーションの開発言語によって、生成されるスタブ・スケルトンファイルが異なるため、IDL定義ファイルのコンパイル時には、アプリケーションの開発言語に合わせて、以下に示すオプションを指定してください。

Windows32

- クライアントアプリケーション
  - C言語：-c
  - C++：-vcpp
  - COBOL：-cobol
- サーバアプリケーション
  - C言語：-mc
  - C++：-mvcpp
  - COBOL：-mcobol

Solaris32

- クライアントアプリケーション
  - C言語：-c
  - C++：-cpp
  - COBOL：-cobol
- サーバアプリケーション
  - C言語：-mc
  - C++：-mcpp
  - COBOL：-mcobol

Linux32

- クライアントアプリケーション
  - C言語：-c
  - C++：-cpp

- サーバアプリケーション
  - C言語:-mc
  - C++:-mcpp

なお、IDLコンパイラはIDL定義ファイルで記述されたIDLインタフェース情報をシステムに登録する機能も備えているため、すでにIDLインタフェース情報が登録されている場合は、IDLコンパイルコマンドが異常終了します。その場合は、以下のオプションを指定し、すでに登録されているIDLインタフェース情報を削除した後に、再度、IDL定義ファイルのコンパイルを実施するようにしてください。

```
-delete
```

例：C言語の場合

```
%tdc -c -mc -delete typlng.idl

%tdc -c -mc typlng.idl
```

C.3.2 クライアント／サーバアプリケーションのコンパイル

Windows32

クライアント／サーバアプリケーションのコンパイルはMicrosoft(R) Visual C++(R) または COBOL97のプロジェクトを使用して行います。コンパイルはこのプロジェクトのビルドを実行することにより、各アプリケーションの作成に必要なファイルのコンパイル、リンクを行います。
プロジェクトのビルドが正常に終了した場合、プロジェクトファイルと同じフォルダに以下のアプリケーションが作成されます。
コンパイル時の注意事項については、"C.6 注意点"を参照してください。

フォルダ	アプリケーション種別	アプリケーション名
long	クライアントアプリケーション	typlng_c.exe
	サーバアプリケーション(C言語)	libtyplng_c.dll
	サーバアプリケーション(C++)	typlng_cpp.exe
	サーバアプリケーション(COBOL)	libtyplng-cbl.dll
string	クライアントアプリケーション	typstr_c.exe
	サーバアプリケーション(C言語)	libtypstr_c.dll
	サーバアプリケーション(C++)	typstr_cpp.exe
	サーバアプリケーション(COBOL)	libtypstr-cbl.dll
sequence	クライアントアプリケーション	typseq_c.exe
	サーバアプリケーション(C言語)	libtypseq_c.dll
	サーバアプリケーション(C++)	typseq_cpp.exe
	サーバアプリケーション(COBOL)	libtypseq-cbl.dll
exception	クライアントアプリケーション	typexp_c.exe
	サーバアプリケーション(C言語)	libtypexp_c.dll
	サーバアプリケーション(C++)	typexp_cpp.exe
	サーバアプリケーション(COBOL)	libtypexp-cbl.dll

Solaris32 Linux32

クライアント／サーバアプリケーションのコンパイルはmakeコマンドを実行することにより、各アプリケーションの作成に必要なファイルのコンパイル、リンクを行います。

サーバアプリケーションの場合は、スレッドモードとプロセスモードの2種類のアプリケーション形態がありますので、アプリケーション形態ごとのMakefileを使用してコンパイルを実施するようにしてください。

スレッドモードの場合

```
%make
```

プロセスモードの場合

```
%make -f Makefile_nt
```

Solaris32

COBOLのプロセスモードの場合、「nt」ディレクトリが作成され、「nt」ディレクトリ配下に中間ファイルが出力されるようになっています。

注意

Solaris32
Sun Studio 12 Update 1以降を使用してサンプルプログラムを翻訳する場合は、Makefileのccコマンドのパスを修正してください。
コンパイラのインストール先が「/opt/sunstudio12.1」である場合のMakefileの修正例を以下に示します。

直接指定する場合

```
CC = /opt/sunstudio12.1/bin/cc
```

環境変数PATHに指定する場合

```
CC = cc
```

makeコマンドが正常に終了した場合、makeコマンドを実行したディレクトリに以下のアプリケーションが作成されます。

ディレクトリ	アプリケーション種別	形態	アプリケーション名
long	クライアントアプリケーション	－	typlng_c
	サーバアプリケーション（C言語）	スレッド	libtyplng_c.so
		プロセス	libtyplng_cnt.so
	サーバアプリケーション（C++）	スレッド	typlng_cpp
		プロセス	typlng_cppnt
	Solaris32 サーバアプリケーション（COBOL）	スレッド	libtyplng-cbl.so
		プロセス	libtyplng-cblnt.so
string	クライアントアプリケーション	－	typstr_c
	サーバアプリケーション（C言語）	スレッド	libtypstr_c.so
		プロセス	libtypstr_cnt.so
	サーバアプリケーション（C++）	スレッド	typstr_cpp
		プロセス	typstr_cppnt
	Solaris32 サーバアプリケーション（COBOL）	スレッド	libtypstr-cbl.so
		プロセス	libtypstr-cblnt.so
sequence	クライアントアプリケーション	－	typseq_c
	サーバアプリケーション（C言語）	スレッド	libtypseq_c.so
		プロセス	libtypseq_cnt.so
	サーバアプリケーション（C++）	スレッド	typseq_cpp
		プロセス	typseq_cppnt
	Solaris32 サーバアプリケーション（COBOL）	スレッド	libtypseq-cbl.so
		プロセス	libtypseq-cblnt.so

ディレクトリ	アプリケーション種別	形態	アプリケーション名
exception	クライアントアプリケーション	－	typexp_c
	サーバアプリケーション(C言語)	スレッド	libtypexp_c.so
		プロセス	libtypexp_cnt.so
	サーバアプリケーション(C++)	スレッド	typexp_cpp
		プロセス	typexp_cppnt
	Solaris32 サーバアプリケーション(COBOL)	スレッド	libtypexp-cbl.so
		プロセス	libtypexp-cblnt.so

C.4 ワークユニットの起動

ワークユニットの起動は、以下の手順で実施します。

1. IDLインタフェース情報の登録
2. ネーミングサービスへの登録
3. ワークユニット情報の登録
4. ワークユニットの起動

なお、「1.」、「2.」および「3.」の登録作業は、「4. ワークユニットの起動」の前であれば順番は関係ありません。

C.4.1 IDLインタフェース情報の登録

IDLインタフェース情報の登録は、IDLコンパイルコマンドで行います。
通常、アプリケーションの作成時に行ったtdcコマンドで登録されています。ただし、アプリケーション作成後、「-delete」指定のtdcコマンドで登録情報を削除した場合等は、IDLコンパイルコマンドで再登録する必要があります。
なお、インタフェース情報の登録のみを行いたい場合は、「-R」指定のtdcコマンドで実施することが可能です。詳細については、"リファレンスマニュアル(コマンド編)"を参照してください。

C.4.2 ネーミングサービスへの登録

ネーミングサービスへの登録は、Interstage起動後にOD_or_admコマンドで行います。
本サンプルは、以下に示すコマンドで登録してください。

```
% OD_or_adm -c IDL:TYPLNG/INTF:1.0 -a FUJITSU-Interstage-TDLC -n TYPLNG::INTF
```

ワークユニット停止後、ネーミングサービスの登録を削除する場合は、以下に示すコマンドで実施してください。

```
% OD_or_adm -d -n TYPLNG::INTF
```

OD_or_admコマンドで指定するパラメタを以下に示します。

フォルダ	インタフェースリポジトリID	ネーミングサービスに登録する名称
long	IDL:TYPLNG/INTF:1.0	TYPLNG::INTF
string	IDL:TYPSTR/INTF:1.0	TYPSTR::INTF
sequence	IDL:TYPSEQ/INTF:1.0	TYPSEQ::INTF
exception	IDL:TYPEXP/INTF:1.0	TYPEXP::INTF

本サンプルのワークユニット定義は、ネーミングサービスの登録形態が「MANUAL」になっているために手動で登録する必要があります。ワークユニット定義のネーミングサービスの登録形態が「AUTO」または省略した場合は、本工程を実施する必要はありません。
なお、ワークユニット定義、および、OD_or_admコマンドの詳細については、"OLTPサーバ運用ガイド"および"リファレンスマニュアル(コマンド編)"を参照してください。

C.4.3 ワークユニット定義の登録

isaddwudefコマンドで、ワークユニット定義ファイルを元にワークユニット定義の情報を登録します。

本サンプルには、開発言語ごとにワークユニット定義ファイルを各ディレクトリ上にスレッドモード、プロセスモードそれぞれ提供していますので、開発言語に合わせてワークユニット定義を登録するようにしてください。

なお、各コマンドの詳細については、"リファレンスマニュアル(コマンド編)"を参照してください。

例:C言語(スレッド)の場合

```
%cd $CURRENT/C
%isaddwudef typlng.wu
```

C言語(プロセス)の場合 Solaris32 Linux32

```
%cd $CURRENT/C
%isaddwudef typlng_nt.wu
```

C.4.4 ワークユニットの起動

isstartwuコマンドで、ワークユニットを起動します。

```
%isstartwu TYPLNG
```

以下に指定するワークユニット名を示します。

ディレクトリ	ワークユニット名
long	TYPLNG
string	TYPSTR
sequence	TYPSEQ
exception	TYPEXP

上記の手順を実施することにより、ワークユニットで定義したアプリケーションは、サーバアプリケーションとしてクライアントアプリケーションからの依頼を受け付けることが可能となります。

C.5 クライアントアプリケーションの実行

本クライアントアプリケーションの実行手順を以下に示します。本クライアントアプリケーションを実行することにより、サーバアプリケーションと通信を行います。

```
%cd $CURRENT/Client
%typlng_c
```

C.6 注意点

■Microsoft(R) Visual C++(R)でコンパイルする場合の注意点 Windows32

- 本サンプルプログラムのプロジェクトファイルは、インストールパスがデフォルトの場合(C:¥Interstage)を想定して提供しています。インストールパスをデフォルトから変更した場合は、FileViewのLibrary Filesを変更する必要があります。

- 参照するインクルードファイルを追加する必要があります。
  ツール(T)－オプション(O)のフォルダを選択し、表示するフォルダ(S)：インクルードファイルに標準設定されているフォルダに加え、以下のフォルダを追加してください。

```
Interstageインストールフォルダ¥td¥include
Interstageインストールフォルダ¥odwin¥include
Interstageインストールフォルダ¥extp¥include
```

■Microsoft(R) Visual C++(R).NETでコンパイルする場合の注意点 Windows32

sequenceフォルダのサンプルプログラムをコンパイルすると、以下のエラーが出力されます。コンパイル時に「/force」オプションを指定してください。

```
error LNK2005: _CORBA_sequence_string_allocbuf は既に typseq_stub.obj で定義されています。
```

■COBOL97でコンパイルする場合の注意点 Windows32

本サンプルプログラムのプロジェクトファイルは、インストールパスがデフォルトの場合(C:¥Interstage)を想定して提供しています。デフォルト以外のフォルダにインストールした場合は、ライブラリファイルおよび登録集格納先のフォルダを変更する必要があります。

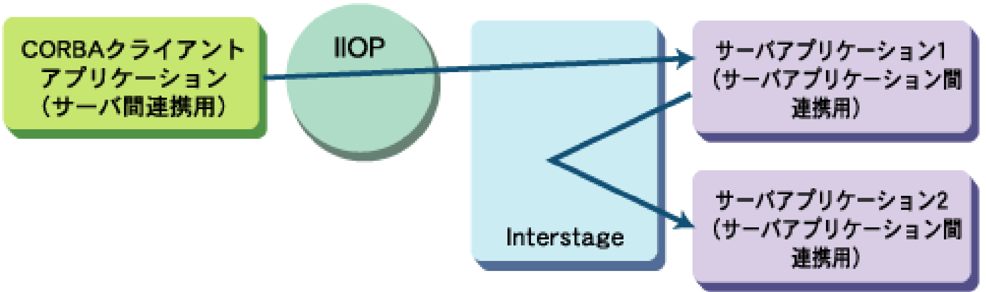
■COBOLコンパイラ変更時の注意点 Solaris32

ワークユニット定義ファイルにおいてアプリケーション使用ライブラリパスにCOBOLランタイムライブラリのパスを指定しているため、コンパイラの変更に合わせてライブラリパスを変更する必要があります。

■Linux for Intel64(32ビット互換)でコンパイルする場合の注意点 Linux32

gcc/g++コマンド実行時に「-m32 -mtune=i386」オプションを指定する必要があります。MakefileまたはMakefile_ntのgcc/g++コマンドに指定するオプションを修正してください。

付録D　トランザクションアプリケーションのサンプルプログラム(サーバアプリケーション間連携編)



D.1　概要

本サンプルは、サーバアプリケーション間連携機能に特化したものです。本機能に影響のない異常時の対応等の処理は簡略化されています。

本サンプルを使用するにあたり、提供ファイルを任意のディレクトリに複写し、複写先の環境に合わせて各ファイルをカストマイズすることをお勧めします。

Windows32

なお、COBOLのサンプルを使用する場合は、空白を含まないフォルダに複写してください。

説明にあたり、以下の記号を使用して説明します。

- $CURRENTは、本アプリケーションで使用する機能ごとのディレクトリを示します。
- “%”は、一般ユーザ時のプロンプトを示します。
- クライアントアプリケーションから依頼を受け付けるサーバアプリケーションを『中継用サーバアプリケーション』、中継用サーバアプリケーションから依頼を受け付けるサーバアプリケーションを『末端用サーバアプリケーション』と仮称し説明していきます。

Windows32

- ‘コンパイルする’は各プロジェクトファイルを使用して実施します。

D.2　ファイル構成

プログラムなどの資産は、以下のディレクトリ配下に格納されています。

Windows32 (インストールパスはデフォルト)

```
C:¥Interstage¥td¥sample¥tds2tds
```

Solaris32 (インストールパスはデフォルト)

```
/opt/FSUNtd/sample/tds2tds （注）
```

Linux32

```
/opt/FJSVtd/sample/tds2tds （注）
```

注）サーバアプリケーションプログラムには、スレッドモードとプロセスモードの2種類の形態のアプリケーションがあります。

D.2.1 ディレクトリ構成

本サンプルのディレクトリ構成を以下に示します。

Windows32 Solaris32

```
tds2tds
 ├─ Client    ： クライアントアプリケーション用ディレクトリ
 ├─ C         ： 中継用サーバアプリケーション(C言語)用ディレクトリ
 ├─ Cobol     ： 中継用サーバアプリケーション(COBOL)用ディレクトリ
 ├─ Cpp       ： 中継用サーバアプリケーション(C++)用ディレクトリ
 └─ Server2   ： 末端用サーバアプリケーション(C言語)用ディレクトリ
```

Linux32

```
tds2tds
 ├─ Client    ： クライアントアプリケーション用ディレクトリ
 ├─ C         ： 中継用サーバアプリケーション(C言語)用ディレクトリ
 ├─ Cpp       ： 中継用サーバアプリケーション(C++)用ディレクトリ
 └─ Server2   ： 末端用サーバアプリケーション(C言語)用ディレクトリ
```

各ディレクトリには、プログラムやワークユニット定義ファイルなど、アプリケーションごとに必要な資産が格納されています。
また、「tds2tds」ディレクトリ直下には、クライアント、サーバを問わず、各アプリケーションに関連のある共通ファイルが格納されています。

D.2.2 ファイル構成

本サンプルのファイル構成を以下に示します。

共通ファイル

本サンプルの共通ファイルの構成を以下に示します。

No	ファイル名	ファイルの概要
(1)	tds2tds.idl	IDLファイル
(2)	tds2tds2.idl	IDLファイル2

クライアントアプリケーション用ディレクトリ

本ディレクトリには、クライアントアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Windows32

No	ファイル名	ファイルの概要
(5)	tds2tds_c.c	クライアントアプリケーションプログラム
(10)	Cl_tds2tds.vcproj	プロジェクトファイル
(11)	Cl_tds2tds.sln	ソリューションファイル

Solaris32 Linux32

No	ファイル名	ファイルの概要
(3)	Makefile	Makefile
(5)	tds2tds_c.c	クライアントアプリケーションプログラム

中継用サーバアプリケーション(C言語)用ディレクトリ

本ディレクトリには、開発言語がC言語の中継用サーバアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Solaris32 Linux32

なお、本サンプルプログラムにはスレッドモードとプロセスモードの2種類の形態のアプリケーションがあります。

Windows32

No	ファイル名	ファイルの概要
(6)	tds2tds_s.c	サーバアプリケーションプログラム
(7)	tds2tds_pre.c	前出口プログラム
(8)	tds2tds.wu	ワークユニット定義ファイル
(10)	C_tds2tds.vcproj	プロジェクトファイル
(11)	C_tds2tds.sln	ソリューションファイル

Solaris32 Linux32

No	ファイル名	ファイルの概要
(3)	Makefile	Makefile(スレッドモード)
(4)	Makefile_nt	Makefile(プロセスモード)
(6)	tds2tds_s.c	サーバアプリケーションプログラム
(7)	tds2tds_pre.c	前出口プログラム
(8)	tds2tds.wu	ワークユニット定義ファイル(スレッドモード)
(9)	tds2tds_nt.wu	ワークユニット定義ファイル(プロセスモード)

中継用サーバアプリケーション(C++)用ディレクトリ

本ディレクトリには、開発言語がC++の中継用サーバアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Solaris32 Linux32

なお、本サンプルプログラムにはスレッドモードとプロセスモードの2種類の形態のアプリケーションがあります。

Windows32

No	ファイル名	ファイルの概要
(6)	tds2tds_s.cpp	サーバアプリケーションプログラム
(7)	tds2tds_pre.cpp	前出口プログラム
(8)	tds2tds.wu	ワークユニット定義ファイル
(10)	Cpp_tds2tds.vcproj	プロジェクトファイル
(11)	Cpp_tds2tds.sln	ソリューションファイル

Solaris32 Linux32

No	ファイル名	ファイルの概要
(3)	Makefile	Makefile（スレッドモード）
(4)	Makefile_nt	Makefile（プロセスモード）
(6)	tds2tds_s.C	サーバアプリケーションプログラム
(7)	tds2tds_pre.C	前出口プログラム
(8)	tds2tds.wu	ワークユニット定義ファイル（スレッドモード）
(9)	tds2tds_nt.wu	ワークユニット定義ファイル（プロセスモード）

中継用サーバアプリケーション（COBOL）用ディレクトリ Windows32 Solaris32

本ディレクトリには、開発言語がCOBOLの中継用サーバアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Solaris32

なお、本サンプルプログラムにはスレッドモードとプロセスモードの2種類の形態のアプリケーションがあります。

Windows32

No	ファイル名	ファイルの概要
(6)	tds2tds_s.cbl	サーバアプリケーションプログラム
(7)	tds2tds_pre.cbl	前出口プログラム
(8)	tds2tds.wu	ワークユニット定義ファイル
(12)	Cbl_tds2tds.prj	プロジェクトファイル
(13)	Cbl_tds2tds.cbi	翻訳オプションファイル

Solaris32

No	ファイル名	ファイルの概要
(3)	Makefile	Makefile（スレッドモード）
(4)	Makefile_nt	Makefile（プロセスモード）
(6)	tds2tds_s.cbl	サーバアプリケーションプログラム
(7)	tds2tds_pre.cbl	前出口プログラム
(8)	tds2tds.wu	ワークユニット定義ファイル（スレッドモード）
(9)	tds2tds_nt.wu	ワークユニット定義ファイル（プロセスモード）

末端用サーバアプリケーション用ディレクトリ

本ディレクトリには、中継用サーバアプリケーションからの依頼を受け付ける末端用サーバアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Solaris32 Linux32

なお、本サンプルプログラムではスレッドモードのみ提供しています。

Windows32

No	ファイル名	ファイルの概要
(6)	tds2tds21_s.c	サーバアプリケーション1プログラム
(6)	tds2tds22_s.c	サーバアプリケーション2プログラム

No	ファイル名	ファイルの概要
(8)	tds2tds2.wu	ワークユニット定義ファイル
(10)	C_tds2tds21.vcproj	サーバアプリケーション1用プロジェクトファイル
(10)	C_tds2tds22.vcproj	サーバアプリケーション2用プロジェクトファイル
(11)	C_tds2tds2.sln	ソリューションファイル

Solaris32 Linux32

No	ファイル名	ファイルの概要
(3)	Makefile	Makefile
(6)	tds2tds21_s.c	サーバアプリケーション1プログラム
(6)	tds2tds22_s.c	サーバアプリケーション2プログラム
(8)	tds2tds2.wu	ワークユニット定義ファイル

(1) IDLファイル

本サンプルで使用するIDLファイルです。クライアントアプリケーション、中継用サーバアプリケーション共通のファイルになります。

(2) IDLファイル2

本サンプルで使用するIDLファイルです。中継用サーバアプリケーション、末端用サーバアプリケーション共通のファイルになります。

(3) Makefile(スレッドモード) Solaris32 Linux32

本アプリケーションのバイナリファイルをスレッドモードで生成するためのMakefileです。

環境に合わせて一部修正する必要があります。

Solaris32

なお、本Makefileは、COBOLの場合Sun日本語COBOL用コンパイル環境に加えてFujitsu PowerCOBOL97用コンパイル環境を、C++の場合Sun WorkShop Compilers C++4.2環境に加えてSun WorkShop 5.0シリーズ、Forte Developer 6シリーズ、Sun ONE Studio 7シリーズ、Sun ONE Studio 8シリーズまたはSun Studio 9環境を用意しています。

(4) Makefile(プロセスモード) Solaris32 Linux32

本アプリケーションのバイナリファイルをプロセスモードで生成するためのMakefileです。

スレッドモードと同様に環境に合わせて一部修正する必要があります。

(5) クライアントプログラム

本プログラムをコンパイルすることにより、クライアントアプリケーションとして利用できるようになります。

(6) サーバアプリケーションプログラム

本プログラムをコンパイルすることにより、サーバアプリケーションとして利用できるようになります。

(7) 前出口プログラム

ワークユニット起動時に呼び出される前出口のアプリケーションです。CORBAサービスに対して以下の処理を行います。
・ORBの初期化
・NamingServiceから通信先サーバアプリケーションのObjectリファレンスの取得

(8) ワークユニット定義ファイル(スレッドモード)

サーバアプリケーション用のワークユニット定義を行うための入力ファイルです。実行環境に合わせて一部修正する必要があります。
Solaris版、Linux版の場合はスレッドモードで行うための入力ファイルです。

(9)ワークユニット定義ファイル(プロセスモード) Solaris32 Linux32

サーバアプリケーション用のワークユニット定義をプロセスモードで行うための入力ファイルです。実行環境に合わせて一部修正する必要があります。

(10) C言語／C++コンパイル用プロジェクトファイル Windows32

Microsoft(R) Visual C++(R)以降上でクライアント／サーバアプリケーションをコンパイルするためのプロジェクトファイルです。

(11) ソリューションファイル Windows32

Microsoft(R) Visual C++(R)以降でプロジェクト構成を管理するためのファイルです。

(12) COBOLコンパイル用プロジェクトファイル Windows32

COBOL97 V50L10以降でサーバアプリケーションをコンパイルするためのプロジェクトファイルです。

(13) COBOL翻訳オプションファイル Windows32

COBOL97 V50L10以降でプロジェクト管理機能を使用してコンパイルするための翻訳オプション情報が格納されているファイルです。

D.3 アプリケーションのコンパイル

クライアント、中継用サーバおよび末端用サーバのアプリケーションをコンパイルする手順を説明します。

Windows32

なお、本アプリケーションは、Microsoft(R) Visual C++(R)、またはCOBOL97 のプロジェクトを使用してコンパイルすることを前提としています。

アプリケーションのコンパイルは以下の手順で実施します。

1. IDL定義ファイルのコンパイル
2. スタブ／スケルトンファイルの修正
3. アプリケーションのコンパイル

Windows32

例：C++の場合

```
    >cd $CURRENT
>tdc -c -mvcpp tds2tds.idl     --(1)
クライアントアプリケーションのコンパイル     --(2)
>tdc -vcpp -mc tds2tds2.idl     --(3)
スタブ／スケルトンファイルの修正     --(4)
中継用、末端用サーバアプリケーションのコンパイル     --(5)
```

Solaris32 Linux32

例：C++の場合

```
    %cd $CURRENT
%tdc -c -mcpp tds2tds.idl     --(1)
%tdc -cpp -mc tds2tds2.idl     --(2)
スタブ／スケルトンファイルの修正     --(3)
%cd $CURRENT/Cpp
%make     --(4)
```

なお、本サンプルのサーバアプリケーション（C++）用以外のMakefileは、IDL定義ファイルのコンパイルも実施するように構成されてます。

D.3.1 IDL定義ファイルのコンパイル

IDL定義ファイルからプログラム作成に必要なスタブ／スケルトンファイルを生成します。スタブ・スケルトンファイルは、IDLコンパイラにより生成されます。
本サンプルの場合、トランザクションアプリケーション用IDLコンパイラであるtdcコマンドでIDL定義ファイルをコンパイルしてください。

なお、アプリケーションの開発言語によって、生成されるスタブ・スケルトンファイルが異なるため、IDL定義ファイルのコンパイル時には、アプリケーションの開発言語に合わせて、以下に示すオプションを指定してください。

Windows32

- クライアントアプリケーション
  - C言語:-c
  - C++:-vcpp
  - COBOL:-cobol
- サーバアプリケーション
  - C言語:-mc
  - C++:-mvcpp
  - COBOL:-mcobol

Solaris32

- クライアントアプリケーション
  - C言語:-c
  - C++:-cpp
  - COBOL:-cobol
- サーバアプリケーション
  - C言語:-mc
  - C++:-mcpp
  - COBOL:-mcobol

Linux32

- クライアントアプリケーション
  - C言語:-c
  - C++:-cpp
- サーバアプリケーション
  - C言語:-mc
  - C++:-mcpp

本サンプルでは、「tds2tds.idl」がクライアント6中継用サーバアプリケーション用、「tds2tds2.idl」が中継用サーバ6末端用サーバ用のため、クライアントアプリケーションをコンパイルする場合は「tds2tds.idl」を、末端用サーバアプリケーションをコンパイルする場合は「tds2tds2.idl」を、中継用サーバアプリケーションをコンパイルする場合は両方のIDL定義ファイルをコンパイルしておく必要があります。

IDL定義ファイル「tds2tds2.idl」をコンパイルする場合、開発言語のオプション指定は、中継用サーバアプリケーションはクライアントアプリケーションとして開発言語を指定するようにしてください。

Windows32 Solaris32

例：中継用サーバがCOBOLの場合

```
%tdc -c -mcobol tds2tds.idl

%tdc -cobol -mc tds2tds2.idl
```

Linux32

例：中継用サーバがC++の場合

```
%tdc -c -mcpp tds2tds.idl
```

```
%tdc -cpp -mc tds2tds2.idl
```

なお、IDLコンパイラはIDL定義ファイルで記述されたIDLインタフェース情報をシステムに登録する機能も備えているため、すでにIDLインタフェース情報が登録されている場合は、IDLコンパイルコマンドが異常終了します。その場合は、以下のオプションを指定し、すでに登録されているIDLインタフェース情報を削除した後に、再度、IDL定義ファイルのコンパイルを実施するようにしてください。

```
-delete
```

Windows32 Solaris32

例：COBOLの場合

```
%tdc -c -mcobol -delete tds2tds.idl
```

```
%tdc -c -mcobol tds2tds.idl
```

Linux32

例：C++の場合

```
%tdc -c -mcpp -delete tds2tds.idl
```

```
%tdc -c -mcpp tds2tds.idl
```

D.3.2 スタブ／スケルトンファイルの修正

IDLコンパイルコマンドにより作成されたスタブ／スケルトンと呼ばれるファイルは、本来、ユーザが修正する必要のないファイルです。しかし、以下で示すような場合など、この生成されたファイルを修正することで、比較的容易にアプリケーションを作成することができるようになります。

- 開発言語がC++のサーバアプリケーションで、前出口、異常出口、後出口処理を使用する場合
- 開発言語がC++のサーバアプリケーションで、プロセスバインドを使用し、かつ、セション中のみ有効は変数を使用したい場合

開発言語がC++の本サーバアプリケーションの場合、前出口処理を使用するため、以下で説明するようにファイルを修正してください。

1) ‘TD_モジュール名_インタフェース名_proto.h (TD_TDS2TDS_INTF_proto.h)’ の修正

```
class ExtpApmUser : public ExtpApmBase {
public:
      // long ApmInit();
      // long ApmRecover();
      // long ApmDestroy();
};
```

修正

```
class ExtpApmUser : public ExtpApmBase {
public:
        long ApmInit();     ← 修正
      // long ApmRecover();
      // long ApmDestroy();
};
```

この修正は、サーバアプリケーションに前出口を登録するものです。
開発言語がC++で前出口、後出口および異常出口を使用する場合は、この生成ファイルを修正する必要があります。

D.3.3 クライアント／サーバアプリケーションのコンパイル

Windows32

クライアント／サーバアプリケーションのコンパイルはMicrosoft(R) Visual C++(R) または COBOL97のプロジェクトを使用して行います。コンパイルはこのプロジェクトのビルドを実行することにより、各アプリケーションの作成に必要なファイルのコンパイル、リンクを行います。
プロジェクトのビルドが正常に終了した場合、プロジェクトファイルと同じフォルダに以下のアプリケーションが作成されます。
コンパイル時の注意事項については、“D.6 注意点”を参照してください。

アプリケーション種別	アプリケーション名
クライアントアプリケーション	tds2tds_c.exe
中継用サーバアプリケーション（C言語）	libtds2tds_c.dll
中継用サーバアプリケーション（C++）	tds2tds_cpp.exe
中継用サーバアプリケーション（COBOL）	libtds2tds-cbl.dll
末端用サーバアプリケーション	libtds2tds21_c.dll
	libtds2tds22_c.dll

Solaris32 Linux32

クライアント／サーバアプリケーションのコンパイルはmakeコマンドを実行することにより、各アプリケーションの作成に必要なファイルのコンパイル、リンクを行います。

中継用サーバアプリケーションの場合は、スレッドモードとプロセスモードの2種類のアプリケーション形態がありますので、アプリケーション形態ごとのMakefileを使用してコンパイルを実施するようにしてください。

スレッドモードの場合

```
%make
```

プロセスモードの場合

```
%make -f Makefile_nt
```

Solaris32

COBOLのプロセスモードの場合、「nt」ディレクトリが作成され、「nt」ディレクトリ配下に中間ファイルが出力されるようになっています。

注意

Solaris32

Sun Studio 12 Update 1以降を使用してサンプルプログラムを翻訳する場合は、Makefileのccコマンドのパスを修正してください。
コンパイラのインストール先が「/opt/sunstudio12.1」である場合のMakefileの修正例を以下に示します。

直接指定する場合

```
CC = /opt/sunstudio12.1/bin/cc
```

環境変数PATHに指定する場合

```
CC = cc
```

makeコマンドが正常に終了した場合、makeコマンドを実行したディレクトリに以下のアプリケーションが作成されます。

アプリケーション種別	形態	アプリケーション名
クライアントアプリケーション	−	tds2tds_c
中継用サーバアプリケーション(C言語)	スレッド	libtds2tds_c.so
	プロセス	libtds2tds_cnt.so
中継用サーバアプリケーション(C++)	スレッド	tds2tds_cpp
	プロセス	tds2tds_cppnt
Solaris32 中継用サーバアプリケーション(COBOL)	スレッド	libtds2tds-cbl.so
	プロセス	libtds2tds-cblnt.so
末端用サーバアプリケーション	−	libtds2tds2_c.so

D.4 ワークユニットの起動

ワークユニットの起動は、以下の手順で実施します。

1. IDLインタフェース情報の登録
2. ネーミングサービスへの登録
3. ワークユニット情報の登録
4. ワークユニットの起動

なお、「1.」、「2.」および「3.」の登録作業は、「4. ワークユニットの起動」の前であれば順番は関係ありません。また、本サンプルの場合、ワークユニットの起動は、「末端用サーバアプリケーション」、「中継用サーバアプリケーション」の順で起動するようにしてください。

D.4.1 IDLインタフェース情報の登録

IDLインタフェース情報の登録は、IDLコンパイルコマンドで行います。
通常、アプリケーションの作成時に行ったtdcコマンドで登録されています。ただし、アプリケーション作成後、「-delete」指定のtdcコマンドで登録情報を削除した場合等は、IDLコンパイルコマンドで再登録する必要があります。
なお、インタフェース情報の登録のみを行いたい場合は、「-R」指定のIDLコンパイルコマンドで実施することが可能です。詳細については、"リファレンスマニュアル(コマンド編)"を参照してください。

D.4.2 ネーミングサービスへの登録

ネーミングサービスへの登録は、Interstage起動後にOD_or_admコマンドで行います。
本サンプルは、以下に示すコマンドで登録してください。

中継用サーバアプリケーション

```
% OD_or_adm -c IDL:TDS2TDS/INTF:1.0 -a FUJITSU-Interstage-TDLC -n TDS2TDS::INTF
```

末端用サーバアプリケーション

```
% OD_or_adm -c IDL:TDS2TDS2/INTF:1.0 -a FUJITSU-Interstage-TDLC -n TDS2TDS2::INTF
% OD_or_adm -c IDL:TDS2TDS2/INTF2:1.0 -a FUJITSU-Interstage-TDLC -n TDS2TDS2::INTF2
```

ワークユニット停止後、ネーミングサービスの登録を削除する場合は、以下に示すコマンドで実施してください。

中継用サーバアプリケーション

```
% OD_or_adm -d -n TDS2TDS::INTF
```

末端用サーバアプリケーション

```
% OD_or_adm -d -n TDS2TDS2::INTF
% OD_or_adm -d -n TDS2TDS2::INTF2
```

本サンプルのワークユニット定義は、ネーミングサービスの登録形態「Registration to Naming Service」が「MANUAL」になっています。そのため、手動で登録する必要があります。ワークユニット定義のネーミングサービスの登録形態が「AUTO」または省略した場合は、本工程を実施する必要はありません。

なお、ワークユニット定義、および、OD_or_admコマンドの詳細については、“OLTPサーバ運用ガイド”および“リファレンスマニュアル(コマンド編)”を参照してください。

D.4.3 ワークユニット定義の登録

isaddwudefコマンドで、ワークユニット定義ファイルを元にワークユニット定義の情報を登録します。

本サンプルには、中継用サーバアプリケーションは開発言語ごとにスレッドモード、プロセスモードそれぞれの形態を、末端用サーバアプリケーションはC言語用のスレッドモードのワークユニット定義ファイルを各ディレクトリ上に提供していますので、中継用サーバアプリケーション、末端用サーバアプリケーションそれぞれのワークユニット定義を登録するようにしてください。

なお、各コマンドの詳細については、“リファレンスマニュアル(コマンド編)”を参照してください。

Windows32

例：

中継用サーバアプリケーション(COBOL用)

```
>cd $CURRENT¥Cobol
>isaddwudef tds2tds.wu
```

末端用サーバアプリケーション

```
>cd $CURRENT¥Server2
>isaddwudef Server2¥tds2tds2.wu
```

Solaris32

例：

中継用サーバアプリケーション(COBOLスレッド用)

```
%cd $CURRENT/Cobol
%isaddwudef tds2tds.wu
```

中継用サーバアプリケーション(COBOLプロセス用)

```
%cd $CURRENT/Cobol
%isaddwudef tds2tds_nt.wu
```

末端用サーバアプリケーション

```
%cd $CURRENT/Server2
%isaddwudef Server2/tds2tds2.wu
```

Linux32

例：

中継用サーバアプリケーション(C言語スレッド用)

```
%cd $CURRENT/C
%isaddwudef tds2tds.wu
```

中継用サーバアプリケーション(C言語プロセス用)

```
%cd $CURRENT/C
%isaddwudef tds2tds_nt.wu
```

末端用サーバアプリケーション

```
%cd $CURRENT/Server2
%isaddwudef Server2/tds2tds2.wu
```

D.4.4 ワークユニットの起動

isstartwuコマンドで、ワークユニットを起動します。
本サンプルでは、末端用サーバアプリケーション用、中継用サーバアプリケーション用の順で起動していきます。

末端用サーバアプリケーション

```
%isstartwu TDS2TDS2
```

Windows32 Solaris32

中継用サーバアプリケーション(COBOL用)

```
%isstartwu TDS2TDS
```

Linux32

中継用サーバアプリケーション(C言語用)

```
%isstartwu TDS2TDS
```

上記の手順を実施することにより、ワークユニットで定義したアプリケーションは、サーバアプリケーションとしてクライアントアプリケーションからの依頼を受け付けることが可能となります。

D.5 クライアントアプリケーションの実行

本クライアントアプリケーションの実行手順を以下に示します。
本クライアントアプリケーションを実行することにより、クライアント6中継用サーバおよび中継用サーバ6末端用サーバと通信を行います。

```
%cd $CURRENT/Client
%tds2tds_c
```

D.6 注意点

■Microsoft(R) Visual C++(R)でコンパイルする場合の注意点 Windows32

- 本サンプルプログラムのプロジェクトファイルは、インストールパスがデフォルトの場合(C:¥Interstage)を想定して提供しています。インストールパスをデフォルトから変更した場合は、FileViewのLibrary Filesを変更する必要があります。
- 参照するインクルードファイルを追加する必要があります。
  ツール(T)－オプション(O)のフォルダを選択し、表示するフォルダ(S):インクルードファイルに標準設定されているフォルダに加え、以下のフォルダを追加してください。

```
Interstageインストールフォルダ¥td¥include
Interstageインストールフォルダ¥odwin¥include
Interstageインストールフォルダ¥extp¥include
```

- C++のサンプルプログラムのコンパイル時に警告メッセージが出力されることがありますが、動作上は問題ありません。

■COBOL97でコンパイルする場合の注意点 Windows32

本サンプルプログラムのプロジェクトファイルは、インストールパスがデフォルトの場合（C:¥Interstage）を想定して提供しています。デフォルト以外のフォルダにインストールした場合は、ライブラリファイルおよび登録集格納先のフォルダを変更する必要があります。

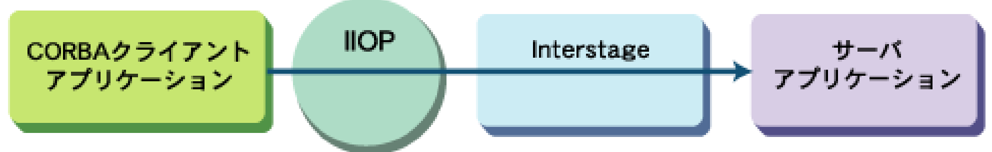
■COBOLコンパイラ変更時の注意点 Solaris32

ワークユニット定義ファイルにおいてアプリケーション使用ライブラリパスにCOBOLランタイムライブラリのパスを指定しているため、コンパイラの変更に合わせてライブラリパスを変更する必要があります。

■Linux for Intel64(32ビット互換)でコンパイルする場合の注意点 Linux32

gcc/g++コマンド実行時に「-m32 -mtune=i386」オプションを指定する必要があります。MakefileまたはMakefile_ntのgcc/g++コマンドに指定するオプションを修正してください。

付録E　トランザクションアプリケーションのサンプルプログラム（プロセスバインド機能編）



E.1　概要

本サンプルは、プロセスバインド機能に特化したものです。本機能に影響のない異常時の対応などの処理は簡略化されています。

機能として、初期化処理でセションが開始され、後処理でセションが終了します。その間、クライアントからの参照オペレーション、加算オペレーションおよび減算オペレーションの依頼を受け付け、その処理結果をクライアントに通知するものです。クライアントからの依頼オペレーションは、サーバでセションIDごとに管理している【変数】の値を通知、またはその【変数】に加算／減算するものです。

Windows32　Solaris32

なお、開発言語がC言語、またはCOBOLの本サンプルの場合、セション管理できるクライアント数は3つです。そのため、4つ目以降のクライアントからの依頼は、失敗してしまいます。

Linux32

なお、開発言語がC言語の本サンプルの場合、セション管理できるクライアント数は3つです。そのため、4つ目以降のクライアントからの依頼は、失敗してしまいます。

IDL定義

```
module PRCBIND{

    const    unsigned long SIDLEN=48;          セションID定義
    typedef octet          SessionID[SIDLEN];

    interface INTF{
        long OP0000(  out SessionID SesID);
        long OP0010(  in  SessinID SesID,
                      in string<8>INP10);
        long OP0020(  in  SessionID SesID,
                      out long   OUTP20);
        long OP0030(  in  SessionID  SesID,
                      in  long    INP30,
                      out long    OUTP30);
        long OP0040(  in  SessionID  SesID,
                      in  long    INP40,
                      out  long  OUTP40);
        long OP0090(  in  SessionID  SesID);
    };
};
```

OP0000

セションID獲得用オペレーション。
サーバで採番したセションIDをクライアントに通知します。獲得したセションIDは、クライアントオブジェクトとサーバオブジェクト内のプロセスをバインドするために、入力パラメタとして「OP0000」～「OP0090」で使用します。

OP0010

初期化用オペレーション。
サーバアプリケーション内で管理している【変数】の初期値を設定しています。また、開発言語がC言語、COBOLの場合、セションIDをキーとする管理テーブルの初期化処理も行います。なお、COBOLはWindows(R)版、Solaris版のみです。
本オペレーションより、セションが開始されます。

OP0020

参照用オペレーション。
サーバアプリケーション内で管理している【変数】をクライアントに通知するものです。

OP0030

加算用オペレーション。
サーバアプリケーション内で管理している【変数】に、クライアントからの入力値を加算し、その結果をクライアントに通知するものです。

OP0040

減算用オペレーション。
サーバアプリケーション内で管理している【変数】に、クライアントからの入力値で減算し、その結果をクライアントに通知するものです。

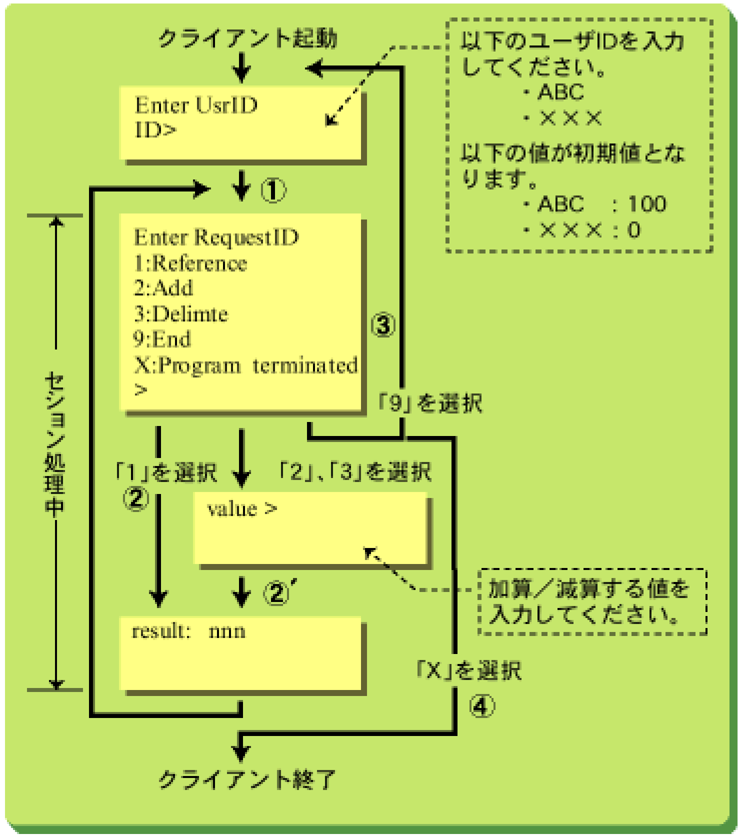
OP0090

後処理用オペレーション。

サーバアプリケーション内で管理している【変数】の初期化を行っています。また、開発言語がC言語、COBOLの場合、セションIDをキーとする管理テーブルの解放処理も行います。なお、COBOLはWindows(R)版、Solaris版のみです。
本オペレーションで、セションが終了します。

操作手順

以下に本サンプルのクライアントアプリケーションの操作手順を示します。



IDL定義ファイルのオペレーション発行契機は、以下のようになります。

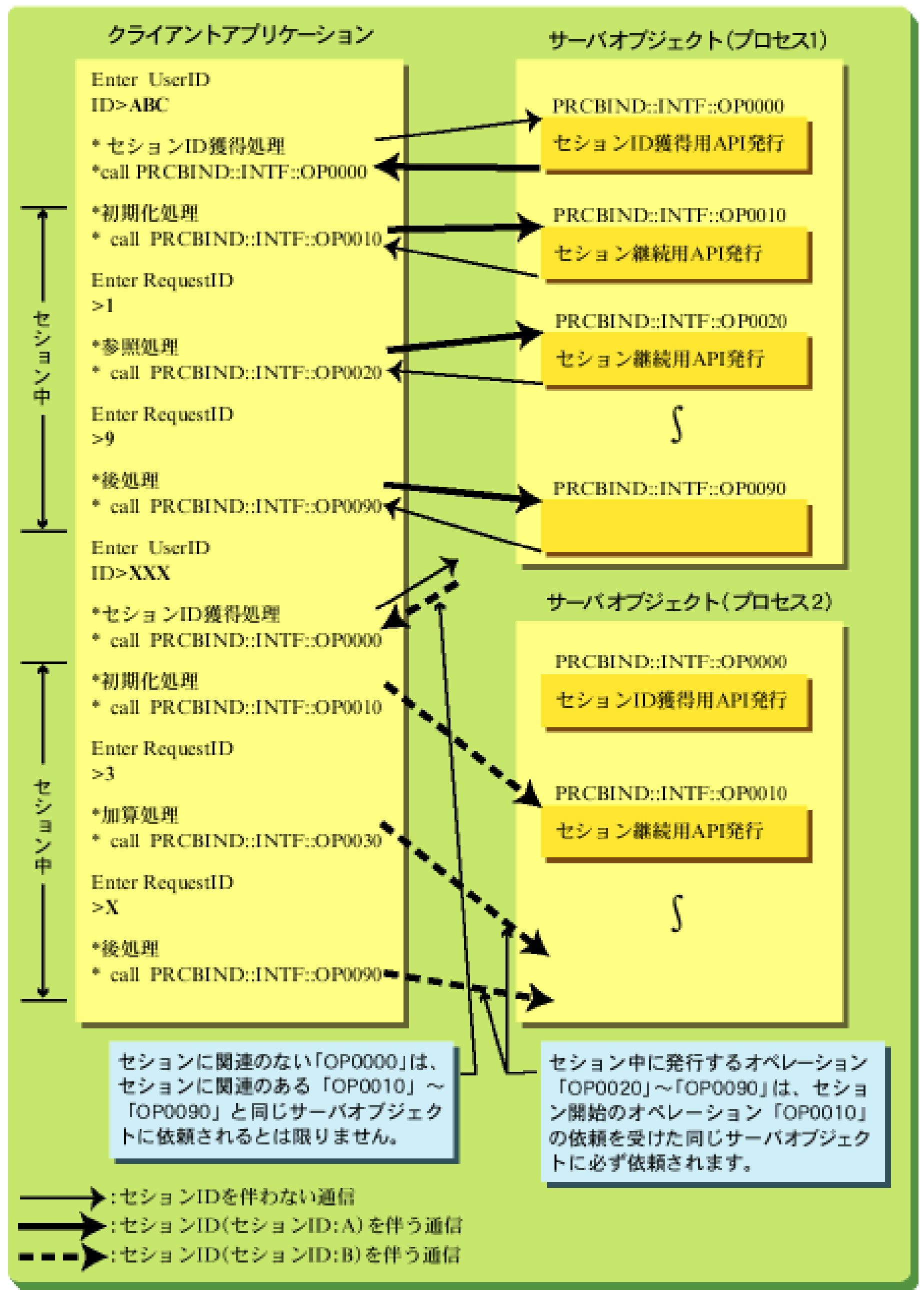
1. ユーザIDが入力されると((1))、セションID獲得オペレーション「OP0000」を発行します。そして、獲得したセションIDで初期化依頼用オペレーション「OP0010」を発行し、セションを開始します。

2. リクエストIDが入力されると((2))、以下のように依頼種別ごとにオペレーションを発行します。このオペレーションは、セションが終了しない限り繰り返し発行することができます。

- 「1(Reference)」:参照用オペレーション「OP0020」
- 「2(Add)」:加算用オペレーション「OP0030」
- 「3(Delimte)」:減算用オペレーション「OP0040」

3. ただし、「9(End)」、「X(Program Terminated)」の終了用のリクエストIDが入力されると((3)、(4))、後処理用のオペレーション「OP0090」を発行し、セションを終了します。

プロセスバインドとセション管理の関連



ワークユニット定義で多重度に2以上を定義している場合、サーバオブジェクトは複数存在することになります。その場合、クライアントアプリケーションからの各オペレーションは、どのサーバオブジェクトに依頼されるか解かりません。
プロセスバインド機能は、セションIDをキーに、セション中に依頼されたオペレーションは、必ずセションを開始したサーバオブジェクトに依頼されるように制御されます。

本サンプルを使用するにあたり、提供ファイルを任意のディレクトリに複写し、複写先の環境に合わせて各ファイルをカストマイズすることをお勧めします。

Windows32

なお、COBOLのサンプルを使用する場合は、空白を含まないフォルダに複写してください。

説明にあたり、コマンドなどの例題には、以下の記号を使用して説明します。

- $CURRENTは、本アプリケーションで使用する機能ごとのディレクトリを示します。

Windows32

- ‘コンパイルする’は各プロジェクトファイルを使用して実施します。

Solaris32 Linux32

- “%”は、一般ユーザ時のプロンプトを示します。

E.2 ファイル構成

プログラムなどの資産は、以下のディレクトリ配下に格納されています。

Windows32 (インストールパスはデフォルト)

```
C:¥Interstage¥td¥sample¥prcbind
```

Solaris32 (インストールパスはデフォルト)

```
/opt/FSUNtd/sample/prcbind (注)
```

Linux32

```
/opt/FJSVtd/sample/prcbind (注)
```

注）サーバアプリケーションプログラムには、スレッドモードとプロセスモードの2種類の形態のアプリケーションがあります。

E.2.1 ディレクトリ構成

本サンプルのディレクトリ構成を以下に示します。

Windows32

```
Prcbind
 ├─ Client   ：クライアントアプリケーション用フォルダ
 ├─ C        ：サーバアプリケーション（C言語）用フォルダ
 ├─ Cobol    ：サーバアプリケーション（COBOL）用フォルダ
 └─ Cpp      ：サーバアプリケーション（C++）用フォルダ
```

Solaris32

```
prcbind
 ├─ Client   ：クライアントアプリケーション用ディレクトリ
 ├─ C        ：サーバアプリケーション（C言語）用ディレクトリ
 ├─ Cobol    ：サーバアプリケーション（COBOL）用ディレクトリ
 └─ Cpp      ：サーバアプリケーション（C++）用ディレクトリ
```

Linux32

```
prcbind
  ├─ Client   ：クライアントアプリケーション用ディレクトリ
  ├─ C        ：サーバアプリケーション（C言語）用ディレクトリ
  └─ Cpp      ：サーバアプリケーション（C++）用ディレクトリ
```

各ディレクトリには、プログラムやワークユニット定義ファイルなど、アプリケーションごとに必要な資産が格納されています。
また、「prcbind」ディレクトリ直下には、クライアント、サーバを問わず、各アプリケーションに関連のある共通ファイルが格納されています。

E.2.2 ファイル構成

本サンプルのファイル構成を以下に示します。

共通ファイル

本サンプルの共通ファイルの構成を以下に示します。

No	ファイル名	ファイルの概要
(1)	prcbind.idl	IDLファイル

クライアントアプリケーション用ディレクトリ

本ディレクトリには、クライアントアプリケーションに関連する資産が格納されています。以下にその構成を示します。

No	ファイル名	ファイルの概要
(2)	Makefile	Makefile
(4)	prcbind_c.c	クライアントアプリケーションプログラム

サーバアプリケーション（C言語）用ディレクトリ

本ディレクトリには、開発言語がC言語のサーバアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Solaris32 Linux32

なお、本サンプルプログラムにはスレッドモードとプロセスモードの2種類の形態のアプリケーションがあります。

Windows32

No	ファイル名	ファイルの概要
(3)	prcbind_s.c	サーバアプリケーションプログラム
(4)	prcbind_rcv.c	異常出口プログラム
(5)	prcbind.wu	ワークユニット定義ファイル
(6)	C_prcbind.vcproj	プロジェクトファイル
(7)	C_prcbind.sln	ソリューションファイル

Solaris32 Linux32

No	ファイル名	ファイルの概要
(2)	Makefile	Makefile（スレッドモード）

No	ファイル名	ファイルの概要
(3)	Makefile_nt	Makefile（プロセスモード）
(5)	prcbind_s.c	サーバアプリケーションプログラム
(6)	prcbind_rcv.c	異常出口プログラム
(7)	prcbind.wu	ワークユニット定義ファイル（スレッドモード）
(8)	prcbind_nt.wu	ワークユニット定義ファイル（プロセスモード）

サーバアプリケーション（C++）用ディレクトリ

本ディレクトリには、開発言語がC++のサーバアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Solaris32 Linux32

なお、本サンプルプログラムにはスレッドモードとプロセスモードの2種類の形態のアプリケーションがあります。

Windows32

No	ファイル名	ファイルの概要
(3)	prcbind_s.cpp	サーバアプリケーションプログラム
(4)	prcbind_rcv.cpp	異常出口プログラム
(5)	prcbind.wu	ワークユニット定義ファイル
(6)	Cpp_prcbind.vcproj	プロジェクトファイル
(7)	Cpp_prcbind.sln	ソリューションファイル

Solaris32 Linux32

No	ファイル名	ファイルの概要
(2)	Makefile	Makefile（スレッドモード）
(3)	Makefile_nt	Makefile（プロセスモード）
(5)	prcbind_s.C	サーバアプリケーションプログラム
(6)	prcbind_rcv.C	異常出口プログラム
(7)	prcbind.wu	ワークユニット定義ファイル（スレッドモード）
(8)	prcbind_nt.wu	ワークユニット定義ファイル（プロセスモード）

サーバアプリケーション（COBOL）用ディレクトリ Windows32 Solaris32

本ディレクトリには、開発言語がCOBOLのサーバアプリケーションに関連する資産が格納されています。以下にその構成を示します。

Solaris32

なお、本サンプルプログラムにはスレッドモードとプロセスモードの2種類の形態のアプリケーションがあります。

Windows32

No	ファイル名	ファイルの概要
(3)	prcbind_s.cbl	サーバアプリケーションプログラム
(4)	prcbind_rcv.cbl	異常出口プログラム
(5)	prcbind.wu	ワークユニット定義ファイル
(8)	Cbl_prcbind.prj	プロジェクトファイル
(9)	Cbl_prcbind.cbi	翻訳オプションファイル

Solaris32

No	ファイル名	ファイルの概要
(2)	Makefile	Makefile（スレッドモード）
(3)	Makefile_nt	Makefile（プロセスモード）
(5)	prcbind_s.cbl	サーバアプリケーションプログラム
(6)	prcbind_rcv.cbl	異常出口プログラム
(7)	prcbind.wu	ワークユニット定義ファイル（スレッドモード）
(8)	prcbind_nt.wu	ワークユニット定義ファイル（プロセスモード）

(1) IDLファイル

本サンプルで使用するIDLファイルです。クライアントアプリケーション、サーバアプリケーション共通のファイルになります。

(2) Makefile（スレッドモード） Solaris32

本アプリケーションのバイナリファイルをスレッドモードで生成するためのMakefileです。

環境に合わせて一部修正する必要があります。

なお、本Makefileは、COBOLの場合Sun日本語COBOL用コンパイル環境に加えてFujitsu PowerCOBOL97用コンパイル環境を、C++の場合Sun WorkShop Compilers C++4.2環境に加えてSun WorkShop 5.0シリーズ、Forte Developer 6シリーズ、Sun ONE Studio 7シリーズ、Sun ONE Studio 8シリーズまたはSun Studio 9環境を用意しています。

(3) Makefile（プロセスモード） Solaris32

本アプリケーションのバイナリファイルをプロセスモードで生成するためのMakefileです。

スレッドモードと同様に環境に合わせて一部修正する必要があります。

(4)クライアントプログラム

本プログラムをコンパイルすることにより、クライアントアプリケーションとして利用できるようになります。

(5)サーバアプリケーションプログラム

本プログラムをコンパイルすることにより、サーバアプリケーションとして利用できるようになります。

(6)異常出口プログラム

本プログラムをコンパイルすることにより、サーバアプリケーションの異常出口として利用できるようになります。異常出口とは、クライアント思考監視時間内に同一のトランザクションアプリケーションが呼び出されない場合に、システムからスケジューリングされるトランザクションアプリケーションの出口処理です。

(7)ワークユニット定義ファイル（スレッドモード）

サーバアプリケーション用のワークユニット定義を行うための入力ファイルです。実行環境に合わせて一部修正する必要があります。
Solaris版、Linux版の場合はスレッドモードで行うための入力ファイルです。

(8)ワークユニット定義ファイル（プロセスモード） Solaris32

サーバアプリケーション用のワークユニット定義をプロセスモードで行うための入力ファイルです。実行環境に合わせて一部修正する必要があります。

(6) C言語／C++コンパイル用プロジェクトファイル Windows32

Microsoft(R) Visual C++(R)以降でクライアント／サーバアプリケーションをコンパイルするためのプロジェクトファイルです。

(7) ソリューションファイル Windows32

Microsoft(R) Visual C++(R)以降でプロジェクト構成を管理するためのファイルです。

(8) COBOLコンパイル用プロジェクトファイル Windows32

COBOL97 V50L10以降でサーバアプリケーションをコンパイルするためのプロジェクトファイルです。

(9) COBOL翻訳オプションファイル Windows32

COBOL97 V50L10以降でプロジェクト管理機能を使用してコンパイルするための翻訳オプション情報が格納されているファイルです。

E.3 アプリケーションのコンパイル

クライアントおよびサーバのアプリケーションをコンパイルする手順を説明します。

Windows32

なお、本アプリケーションは、Microsoft(R) Visual C++(R)、またはCOBOL97 のプロジェクトを使用してコンパイルすることを前提としています。

アプリケーションのコンパイルは以下の手順で実施します。

1. IDL定義ファイルのコンパイル
2. スタブ／スケルトンファイルの修正
3. アプリケーションのコンパイル

Windows32

```
>cd $CURRENT
>tdc -c -mc prcbind.idl     ----------(1)
スタブ／スケルトンファイルの修正     --(2)
アプリケーションのコンパイル         --(3)
```

Solaris32 Linux32

例：C++の場合

```
%cd $CURRENT
%tdc -c -mcpp prcbind.idl     ----------(1)
スタブ／スケルトンファイルの修正     --(2)
%cd $CURRENT/Cpp
%make    ------------------------(3)
```

なお、本サンプルのサーバアプリケーション（C++）用以外のMakefileは、IDL定義ファイルのコンパイルも実施するように構成されてます。

E.3.1 IDL定義ファイルのコンパイル

IDL定義ファイルからプログラム作成に必要なスタブ／スケルトンファイルを生成します。スタブ・スケルトンファイルは、IDLコンパイラにより生成されます。

本サンプルの場合、トランザクションアプリケーション用IDLコンパイラであるtdcコマンドでIDL定義ファイルをコンパイルしてください。
なお、アプリケーションの開発言語によって、生成されるスタブ・スケルトンファイルが異なるため、IDL定義ファイルのコンパイル時には、アプリケーションの開発言語に合わせて、以下に示すオプションを指定してください。

Windows32

- クライアントアプリケーション
  - C言語：-c
  - C++：-vcpp
  - COBOL：-cobol
- サーバアプリケーション
  - C言語：-mc
  - C++：-mvcpp

- COBOL：-mcobol

Solaris32

- クライアントアプリケーション
  - C言語：-c
  - C++：-cpp
  - COBOL：-cobol
- サーバアプリケーション
  - C言語：-mc
  - C++：-mcpp
  - COBOL：-mcobol

Linux32

- クライアントアプリケーション
  - C言語：-c
  - C++：-cpp
- サーバアプリケーション
  - C言語：-mc
  - C++：-mcpp

なお、IDLコンパイラはIDL定義ファイルで記述されたIDLインタフェース情報をシステムに登録する機能も備えているため、すでにIDLインタフェース情報が登録されている場合は、IDLコンパイルコマンドが異常終了します。その場合は、以下のオプションを指定し、すでに登録されているIDLインタフェース情報を削除した後に、再度、IDL定義ファイルのコンパイルを実施するようにしてください。

```
-delete
```

例：C言語の場合

```
%tdc -c -mc -delete prcbind.idl

%tdc -c -mc prcbind.idl
```

E.3.2 スタブ／スケルトンファイルの修正

IDLコンパイルコマンドにより作成されたスタブ／スケルトンと呼ばれるファイルは、本来、ユーザが修正する必要のないファイルです。しかし、以下で示すような場合など、この生成されたファイルを修正することで、比較的容易にアプリケーションを作成することができるようになります。

- 開発言語がC++のサーバアプリケーションで、前出口、異常出口、後出口処理を使用する場合
- 開発言語がC++のサーバアプリケーションで、プロセスバインドを使用し、かつ、セション中のみ有効な変数を使用したい場合

開発言語がC++の本サーバアプリケーションの場合、異常出口処理を使用し、かつプロセスバインド機能においてセション中のみ有効な変数を使用するため、以下で説明するようにファイルを修正してください。

1) ‘TD_モジュール名_インタフェース名_proto.h (TD_PRCBIND_INTF_proto.h)’ の修正

```
class ExtpApmUser : public ExtpApmBase {
public:
        // long ApmInit();
        // long ApmRecover();
        // long ApmDestroy();
};
```

修正

```
class ExtpApmUser : public ExtpApmBase {
public:
        // long ApmInit();
          long ApmRecover();  ← 修正
        // long ApmDestroy();
};
```

この修正は、サーバアプリケーションに異常出口を登録するものです。
開発言語がC++で前出口、後出口および異常出口を使用する場合は、この生成ファイルを修正する必要があります。

2) 'TD_IDLファイル名.h (TD_prcbind.h)' の修正

```
// Implementation Class
class PRCBND_INTF_impl : public virtual PRCBND::INTF
{
public:
        PRCBND_INTF_impl(){}
```

修正

```
// Implementation Class
class PRCBND_INTF_impl : public virtual PRCBND::INTF
{
public:
        CORBA::Long      dVle;        ← 追加

        PRCBND_INTF_impl(){}
```

本修正は、当該サーバアプリケーションでセション中のみ有効な変数の定義、すなわち、サーバオブジェクトのプロセス内に作成されたインスタンスにインスタンス変数を定義するものです。
サーバオブジェクトのインスタンス生成の詳細については、"5.2 プロセスバインド機能を使用したトランザクションアプリケーションの作成"を参照してください。

E.3.3 クライアント／サーバアプリケーションのコンパイル

Windows32

クライアント／サーバアプリケーションのコンパイルはMicrosoft(R) Visual C++(R) または COBOL97のプロジェクトを使用して行います。コンパイルはこのプロジェクトのビルドを実行することにより、各アプリケーションの作成に必要なファイルのコンパイル、リンクを行います。
プロジェクトのビルドが正常に終了した場合、プロジェクトファイルと同じフォルダに以下のアプリケーションが作成されます。
コンパイル時の注意事項については、"E.6 注意点"を参照してください。

アプリケーション種別	アプリケーション名
クライアントアプリケーション	prcbind_c.exe
サーバアプリケーション(C言語)	libprcbind_c.dll
サーバアプリケーション(C++)	prcbind_cpp.exe
サーバアプリケーション(COBOL)	libprcbind-cbl.dll

Solaris32 Linux32

クライアント／サーバアプリケーションのコンパイルはmakeコマンドを実行することにより、各アプリケーションの作成に必要なファイルのコンパイル、リンクを行います。

サーバアプリケーションの場合は、スレッドモードとプロセスモードの2種類のアプリケーション形態がありますので、アプリケーション形態ごとのMakefileを使用してコンパイルを実施するようにしてください。

スレッドモードの場合

```
%make
```

プロセスモードの場合

```
%make -f Makefile_nt
```

Solaris32

COBOLのプロセスモードの場合、「nt」ディレクトリが作成され、「nt」ディレクトリ配下に中間ファイルが出力されるようになっています。

注意

Solaris32

Sun Studio 12 Update 1以降を使用してサンプルプログラムを翻訳する場合は、Makefileのccコマンドのパスを修正してください。
コンパイラのインストール先が「/opt/sunstudio12.1」である場合のMakefileの修正例を以下に示します。

直接指定する場合

```
CC = /opt/sunstudio12.1/bin/cc
```

環境変数PATHに指定する場合

```
CC = cc
```

makeコマンドが正常に終了した場合、makeコマンドを実行したディレクトリに以下のアプリケーションが作成されます。

アプリケーション種別	形態	アプリケーション名
クライアントアプリケーション	－	prcbind_c
サーバアプリケーション（C言語）	スレッド	libprcbind_c.so
	プロセス	libprcbind_cnt.so
サーバアプリケーション（C++）	スレッド	prcbind_cpp
	プロセス	prcbind_cppnt
Solaris32 サーバアプリケーション（COBOL）	スレッド	libprcbind-cbl.so
	プロセス	libprcbind-cblnt.so

E.4 ワークユニットの起動

ワークユニットの起動は、以下の手順で実施します。

1. IDLインタフェース情報の登録
2. ネーミングサービスへの登録
3. ワークユニット情報の登録
4. ワークユニットの起動

なお、「1.」、「2.」および「3.」の登録作業は、「4. ワークユニットの起動」の前であれば順番は関係ありません。

E.4.1　IDLインタフェース情報の登録

IDLインタフェース情報の登録は、IDLコンパイルコマンドで行います。

通常、アプリケーションの作成時に行ったtdcコマンドで登録されています。ただし、アプリケーション作成後、「-delete」指定のtdcコマンドで登録情報を削除した場合などは、IDLコンパイルコマンドで再登録する必要があります。

なお、インタフェース情報の登録のみを行いたい場合は、「-R」指定のIDLコンパイルコマンドで実施することが可能です。詳細については、“リファレンスマニュアル（コマンド編）”を参照してください。

E.4.2　ネーミングサービスへの登録

ネーミングサービスへの登録は、Interstage起動後にOD_or_admコマンドで行います。

本サンプルは、以下に示すコマンドで登録してください。

```
% OD_or_adm -c IDL:PRCBIND/INTF:1.0 -a FUJITSU-Interstage-TDLC -n PRCBIND::INTF
```

ワークユニット停止後、ネーミングサービスの登録を削除する場合は、以下に示すコマンドで実施してください。

```
% OD_or_adm -d -n PRCBIND::INTF
```

本サンプルのワークユニット定義は、ネーミングサービスの登録形態「Registration to Naming Service」が「MANUAL」になっています。そのため、手動で登録する必要があります。ワークユニット定義のネーミングサービスの登録形態が「AUTO」または省略した場合は、本工程を実施する必要はありません。

なお、ワークユニット定義、およびOD_or_admコマンドの詳細については、“OLTPサーバ運用ガイド”および“リファレンスマニュアル（コマンド編）”を参照してください。

E.4.3　ワークユニット定義の登録

isaddwudefコマンドで、ワークユニット定義ファイルを元にワークユニット定義の情報を登録します。

本サンプルには、開発言語ごとにワークユニット定義ファイルを各ディレクトリ上にスレッドモード、プロセスモードそれぞれ提供していますので、開発言語に合わせてワークユニット定義を登録するようにしてください。

なお、各コマンドの詳細については、“リファレンスマニュアル（コマンド編）”を参照してください。

Windows32

例：C言語の場合

```
>cd $CURRENT¥C
>isaddwudef prcbind.wu
```

Solaris32　Linux32

例：C言語（スレッド）の場合

```
%cd $CURRENT/C
%isaddwudef prcbind.wu
```

C言語（プロセス）の場合

```
%cd $CURRENT/C
%isaddwudef prcbind_nt.wu
```

E.4.4　ワークユニットの起動

isstartwuコマンドで、ワークユニットを起動します。

```
%isstartwu PRCBIND
```

上記の手順を実施することにより、ワークユニットで定義したアプリケーションは、サーバアプリケーションとしてクライアントアプリケーションからの依頼を受け付けることが可能となります。

E.5　クライアントアプリケーションの実行

本クライアントアプリケーションの実行手順を以下に示します。本クライアントアプリケーションを実行することにより、サーバアプリケーションと通信を行います。
クライアントアプリケーションでの操作手順については、“E.1 概要“を参照してください。

```
%cd $CURRENT/Client
%prcbind_c
```

E.6　注意点

■Microsoft(R) Visual C++(R)でコンパイルする場合の注意点 Windows32

- 本サンプルプログラムのプロジェクトファイルは、インストールパスがデフォルトの場合（C:¥Interstage）を想定して提供しています。インストールパスをデフォルトから変更した場合は、FileViewのLibrary Filesを変更する必要があります。
- 参照するインクルードファイルを追加する必要があります。
  ツール(T)－オプション(O)のフォルダを選択し、表示するフォルダ(S)：インクルードファイルに標準設定されているフォルダに加え、以下のフォルダを追加してください。

```
Interstageインストールフォルダ¥td¥include
Interstageインストールフォルダ¥odwin¥include
Interstageインストールフォルダ¥extp¥include
```

■COBOL97でコンパイルする場合の注意点 Windows32

本サンプルプログラムのプロジェクトファイルは、インストールパスがデフォルトの場合（C:¥Interstage）を想定して提供しています。デフォルト以外のフォルダにインストールした場合は、ライブラリファイルおよび登録集格納先のフォルダを変更する必要があります。

■COBOLコンパイラ変更時の注意点 Solaris32

ワークユニット定義ファイルにおいてアプリケーション使用ライブラリパスにCOBOLランタイムライブラリのパスを指定しているため、コンパイラの変更に合わせてライブラリパスを変更する必要があります。

■Linux for Intel64(32ビット互換)でコンパイルする場合の注意点 Linux32

gcc/g++コマンド実行時に「-m32 -mtune=i386」オプションを指定する必要があります。MakefileまたはMakefile_ntのgcc/g++コマンドに指定するオプションを修正してください。

索　引

[A]

APM........................9,70
APMプロセスの起動........................34

[C]

CORBAサービスの初期化........................107,125

[I]

IDL........................122
IDLインタフェース情報........................222,234,250
IDL定義........................97,100
IDLファイル........................13,43,74
IDLファイルの記述形式........................13,43,74
includeファイル........................49

[S]

SMO........................106,107

[T]

tdcコマンド........................19
tdformsnap........................165
tdfreesnap........................165
tdstartsnap........................165
tdstopsnap........................165

[あ]

後出口プログラム........................9,39,70
アプリケーションのテスト........................33,65,90
アプリケーションの登録........................32,64,89
異常出口........................104
異常出口プログラム........................49
インタフェース情報........................19,52,80
インタフェース宣言........................14,44,75
エラーログ........................12,42,73
オブジェクトリファレンスの作成........................32
オペレーション宣言........................14,44,75
オペレーションの使用方法........................17,48,78

[か]

基本データ型の宣言........................14,45,75
クライアントアプリケーションへの復帰値........................12,41,73
クライアントオブジェクト領域の解放........................104
クライアント識別子の獲得........................108
クライアント思考時間監視........................104
継承........................29,61,86
構造体........................15,45,76
コンポーネントトランザクションサービスにおける利用範囲
........................17,47,78

[さ]

サーバアプリケーションの入出力情報........................11,41,72
事象通知リスナオブジェクト........................125
事象通知リスナ実装関数の作成........................125
事象通知リスナの登録........................128
事象通知リスナの登録抹消........................129
自動登録........................32
手動登録........................32
シーケンス型........................16,46,77
スナップショット........................36,68,93
スナップショット機能........................164
セションID........................97
セション情報域からの情報の読み込み........................115
セション情報域の削除........................118
セション情報域の生成........................108
セション情報域への情報の書き込み........................112
セション情報管理オブジェクトのオブジェクトリファレンスの獲得
........................107
セション情報管理機能........................106
ソースのコンパイル・リンク........................19,51,80

[た]

提供インクルード........................121,130
提供ライブラリ........................121,131
定数宣言........................14,44,75
出口プログラムの作成........................18,48,79
テスト用モジュールの作成........................34
データ型と型宣言........................14,44,75
データ型の使用方法........................17,48,78
トランザクションアプリケーション........................3

[な]

ネーミングサービスへの登録........................32

[は]

配列........................16,46,77
ファイル出力のスナップショット........................164
プロセス回収出口機能........................179
プロセスバインド機能........................98

[ま]

前出口プログラム........................9,39,70
メモリ出力のスナップショット........................164
モジュール宣言........................13,43,74

[や]

ユーティリティワークユニットのプロセス停止出口機能........................186

[ら]

例外宣言........................16,47,77
ローカルトランザクションアプリケーション........................2

[わ]

ワークユニット定義........................31,64,88,97,100,176,188
ワークユニット出口機能........................174
ワークユニットの起動........................34
ワークユニットプロセス情報通知機能........................184