トランザクショナル分散オブジェクト基盤

TPBroker Object Transaction Monitor
プログラマーズガイド

文法・操作書

3000-3-774-20

マニュアルの購入方法

このマニュアル，および関連するマニュアルをご購入の際は，
巻末の用紙をご利用ください。

HITACHI

■対象製品

• **適用** OS：AIX 5L

P-1M64-E111 TPBroker Object Transaction Monitor 03-03

P-1M64-E121 Cosminexus TPBroker Object Transaction Monitor 03-03

P-1M64-E211 TPBroker Object Transaction Monitor - Client 03-03

P-1M64-E511 TPBroker Object Transaction Monitor - Connector for Object Request Broker 01-00

• **適用** OS：HP-UX 11.0，HP-UX 11i，HP-UX 11i V2

P-1B64-E121 TPBroker Object Transaction Monitor 03-03 ※

P-1B64-E131 Cosminexus TPBroker Object Transaction Monitor 03-03 ※

P-1B64-E221 TPBroker Object Transaction Monitor - Client 03-03 ※

P-1B64-E521 TPBroker Object Transaction Monitor - Connector for Object Request Broker 01-00

• **適用** OS：HP-UX 11i V2 (IPF)

P-1J64-E111 TPBroker Object Transaction Monitor 03-03 ※

P-1B64-E211 TPBroker Object Transaction Monitor - Client 03-03 ※

P-1J64-E511 TPBroker Object Transaction Monitor - Connector for Object Request Broker 01-00

• **適用** OS：Solaris

P-9D64-E111 TPBroker Object Transaction Monitor 03-03 ※

P-9D64-E121 Cosminexus TPBroker Object Transaction Monitor 03-03 ※

P-9D64-E211 TPBroker Object Transaction Monitor - Client 03-03 ※

P-9D64-E511 TPBroker Object Transaction Monitor - Connector for Object Request Broker 01-00

• **適用** OS：Windows Server 2003，Windows XP，Windows 2000，Windows NT 4.0

P-2464-E114 TPBroker Object Transaction Monitor 03-03 ※

P-2464-E124 Cosminexus TPBroker Object Transaction Monitor 03-03 ※

P-2464-E214 TPBroker Object Transaction Monitor - Client 03-03 ※

P-2464-E514 TPBroker Object Transaction Monitor - Connector for Object Request Broker 01-00

※印のプログラムプロダクトについては，発行時期をご確認ください。

これらのプログラムプロダクトのほかにも，このマニュアルをご利用になれる場合があります。詳細は「ソフトウェア添付資料」または「Readme ファイル」でご確認ください。

■輸出時の注意

本製品を輸出される場合には，外国為替および外国貿易法ならびに米国の輸出管理関連法規などの規制をご確認の上，必要な手続きをお取りください。

なお，ご不明な場合は，弊社担当営業にお問い合わせください。

■商標類

■発行

2002 年 2 月（第 1 版）3000-3-774
2005 年 6 月（第 3 版）3000-3-774-20

■著作権

変更内容

変更内容（3000-3-774-20） TPBroker Object Transaction Monitor　03-03，TPBroker Object Transaction Monitor - Client　03-03，Cosminexus TPBroker Object Transaction Monitor　03-03

追加・変更機能	変更個所
スケジュール用キューの長さをキューごとに変更する機能を追加した。 これに伴い，TSCRootAcceptor クラスに setQueueLength メソッドまたは TSCRAcceptor-setQueueLength 副プログラムを追加した。	TSCRootAcceptor（C++），TSCRootAcceptor（Java），TSCRootAcceptor（COBOL）
スケジュール用キューの長さを取得する機能を追加した。 これに伴い，TSCRootAcceptor クラスに getQueueLength メソッドまたは TSCRAcceptor-getQueueLength 副プログラムを追加した。	TSCRootAcceptor（C++），TSCRootAcceptor（Java），TSCRootAcceptor（COBOL）
内容コードを追加した。	付録 D.1，付録 D.7

単なる誤字・脱字などはお断りなく訂正しました。

変更内容（3000-3-774-10） TPBroker Object Transaction Monitor　03-01，TPBroker Object Transaction Monitor - Client　03-01，Cosminexus TPBroker Object Transaction Monitor　03-01

追加・変更機能
一つのクライアントアプリケーションからのすべてのリクエストを同一のインスタンスで受け付ける機能を追加した。
tscidl2j コマンドに -package オプションを追加した。
内容コードに追加・変更があった。

はじめに

このマニュアルは，TPBroker Object Transaction Monitor，Cosminexus TPBroker Object Transaction Monitor，TPBroker Object Transaction Monitor - Client，および TPBroker Object Transaction Monitor - Connector for Object Request Broker を使用してアプリケーションプログラムを作成する方法について説明したものです。
なお，このマニュアルでは，TPBroker Object Transaction Monitor および Cosminexus TPBroker Object Transaction Monitor を OTM と表記し，TPBroker Object Transaction Monitor - Client を OTM - Client と表記しています。また，TPBroker Object Transaction Monitor - Connector for Object Request Broker を OTM - Connector for ORB と表記しています。これらすべてを総称するときは，TPBroker OTM と表記します。

■対象読者

TPBroker OTM を使用するシステム管理者およびアプリケーションプログラマを対象としています。

■マニュアルの構成

このマニュアルは，次に示す章と付録から構成されています。

第 1 章　IDL 文法
OTM のトランザクションフレームジェネレータで使用する IDL 文法について説明しています。

第 2 章　アプリケーションプログラムの作成（C++）
C++ でアプリケーションプログラムを作成する方法について説明しています。

第 3 章　アプリケーションプログラミングインタフェース（C++）
C++ で使用するクラスライブラリについて説明しています。

第 4 章　アプリケーションプログラムの作成（Java）
Java でアプリケーションプログラムを作成する方法について説明しています。

第 5 章　アプリケーションプログラミングインタフェース（Java）
Java で使用するクラスライブラリについて説明しています。

第 6 章　アプリケーションプログラムの作成（COBOL）
COBOL でアプリケーションプログラムを作成する方法について説明しています。

第 7 章　アプリケーションプログラミングインタフェース（COBOL）
COBOL で使用する副プログラムについて説明しています。

第 8 章　コマンドリファレンス
アプリケーションプログラムの作成時に使用するコマンドについて説明しています。

付録 A　エラーコード一覧

OTM で障害が発生したときに，障害の内容を示すエラーコードについて説明しています。

付録 B　場所コード一覧

OTM で障害が発生した場所を示す場所コードについて説明しています。

付録 C　完了状態一覧

OTM で障害が発生したときに，メソッド（副プログラム）の呼び出しが完了しているかどうかを示す完了コードについて説明しています。

付録 D　内容コード一覧

例外で使用する内容コードについて説明しています。

■関連マニュアル

このマニュアルの関連マニュアルを次に示します。必要に応じてお読みください。

・TPBroker Version 3

トランザクショナル分散オブジェクト基盤
TPBroker　ユーザーズガイド
㊙㊐㊓㊗　(3000-3-660)

トランザクショナル分散オブジェクト基盤
TPBroker for C++
ユーザーズガイド
㊙㊐㊓㊗　(3000-3-771)

トランザクショナル分散オブジェクト基盤
TPBroker for Java
OTSガイド
㊐㊓㊗　(3000-3-661)

トランザクショナル分散オブジェクト基盤
TPBroker for the Java™
Platform　ユーザーズガイド
㊙㊐㊓㊗　(3000-3-772)

・TPBroker OTM Version 3

トランザクショナル分散オブジェクト基盤
TPBroker Object
Transaction Monitor
ユーザーズガイド
㊙㊐㊗　(3000-3-773)

トランザクショナル分散オブジェクト基盤
TPBroker Object
Transaction Monitor
プログラマーズガイド
㊓㊗　(3000-3-774)

・VisiBroker for C++ Version 3

VisiBroker for C++
プログラマーズガイド
（AIX, HP-UX, Solaris用）
㊙㊐㊓㊗　(3000-3-651)

VisiBroker for C++
リファレンス
（AIX, HP-UX, Solaris用）
㊓　(3000-3-652)

VisiBroker for C++
プログラマーズガイド
（Windows(R)用）
㊙㊐㊓㊗　(3000-3-678)

VisiBroker for C++
リファレンス
（Windows(R)用）
㊓　(3000-3-679)

・VisiBroker for Java Version 3

VisiBroker for Java
プログラマーズガイド
㊙㊐㊓㊗　(3000-3-656)

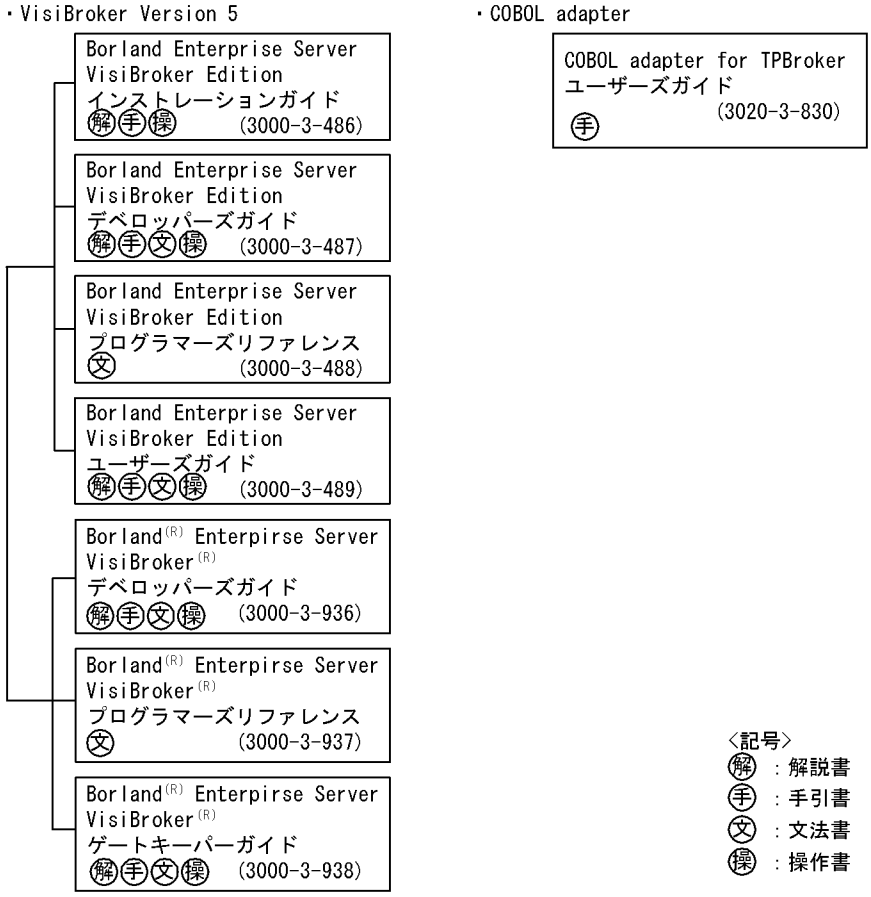
VisiBroker for Java
リファレンス
㊓　(3000-3-657)

・VisiBroker for Java Version 4

VisiBroker Version 4
VisiBroker for Java
プログラマーズガイド
㊙㊐㊓㊗　(3000-3-827)

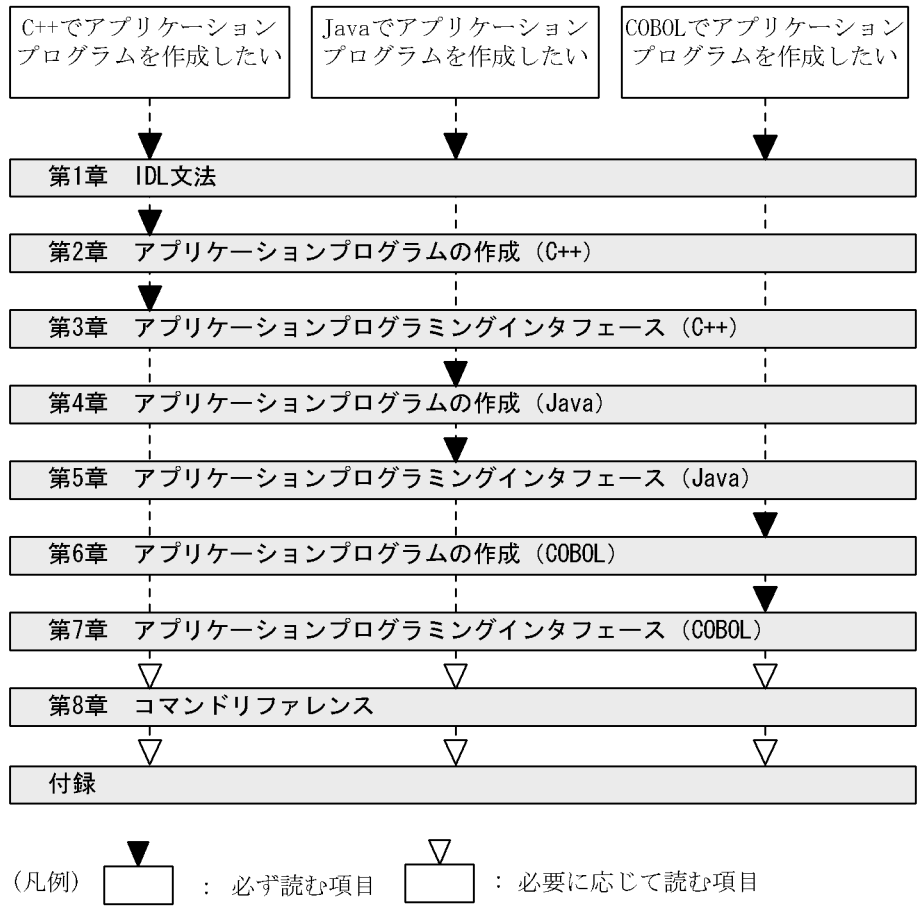
VisiBroker Version 4
VisiBroker for Java
リファレンス
㊓　(3000-3-828)

〈記号〉
㊙：解説書
㊐：手引書
㊓：文法書
㊗：操作書



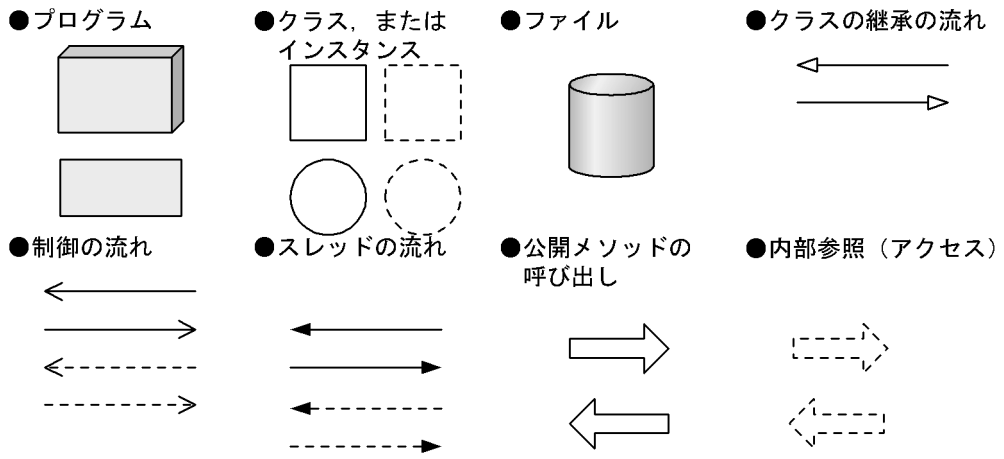
■読書手順

このマニュアルは，利用目的に合わせて直接章を選択して読むことができます。利用目的別に次の流れに従ってお読みいただくことをお勧めします。



■図中で使用する記号

このマニュアルの図中で使用する記号を，次のように定義します。



■このマニュアルでの表記

このマニュアルで使用する製品名称の略称を次に示します。

名称	略称
AIX 5L	AIX
HP-UX 11.0	HP-UX
HP-UX 11i	
HP-UX 11i V2	
HP-UX 11i V2 (IPF)	
Itanium(R) Processor Family	IPF
Java™	Java
Microsoft(R) Windows(R) 2000 Advanced Server Operating System	Windows 2000
Microsoft(R) Windows(R) 2000 Datacenter Server Operating System	
Microsoft(R) Windows(R) 2000 Professional Operating System	
Microsoft(R) Windows(R) 2000 Server Operating System	
Microsoft(R) Windows NT(R) Server Network Operating System Version 4.0	Windows NT
Microsoft(R) Windows NT(R) Workstation Operating System Version 4.0	
Microsoft(R) Windows Server(TM) 2003, Enterprise Edition	Windows Server 2003
Microsoft(R) Windows Server(TM) 2003, Standard Edition	
Microsoft(R) Windows(R) XP Professional Operating System	Windows XP

- Windows 2000，Windows NT，Windows Server 2003，および Windows XP で機能差がない場合，Windows と表記しています。
- AIX，HP-UX，および Solaris を総称して UNIX と表記しています。

このマニュアルで使用する英略語の一覧を示します。

英略語	英字での表記
CORBA	Common Object Request Broker Architecture
DLL	Dynamic Linking Library
IDL	Interface Definition Language
ORB	Object Request Broker
OS	Operating System
OTS	Object Transaction Service
TPBroker	TPBroker for C++，または TPBroker for Java
TSC	TPBroker Schedule Control

■文法の記号

このマニュアルで使用する記号の意味を示します。
C++，Java，COBOL のインタフェースで次の表に示す記号を使用した場合，各言語の文法規則に従います。

文法記述記号	意味
[ ]	この記号で囲まれている項目は省略してもよいことを示します。 （例）tscidl2cbl [-idl2cobol] これは，tscidl2cbl と指定するか，または tscidl2cbl -idl2cobol と指定することを示します。
...	この記号で示す直前の項目を繰り返し指定できることを示します。 （例）-I ディレクトリ [: ディレクトリ ...] これは，-I オプションに，ディレクトリを繰り返し指定できることを示します。
|	この記号で区切られた項目を選択できることを示します。 （例）-format 1 | 2 これは，-format オプションに 1 か 2 のどちらかを指定できることを示します。
～	この記号のあとにユーザ指定値の属性を示します。
《 》	ユーザが指定を省略したときの省略値を示します。
＜ ＞	ユーザ指定値の構文要素を示します。
(( ))	ユーザ指定値の指定範囲を示します。
＜英字＞	アルファベット（A ～ Z，a ～ z）と＿（アンダースコア）
＜英数字＞※	英字と数字（0 ～ 9）
＜文字列＞	任意の文字の配列
＜符号なし整数＞※	数字（0 ～ 9）
＜2 進数字＞	数字（0 または 1）

注
　すべて 1 バイトコードで指定してください。

注※
　数字は 10 進数表現で指定してください。

■常用漢字以外の漢字の使用について

このマニュアルでは，常用漢字を使用することを基本としていますが，次に示す用語については，常用漢字以外の漢字を使用しています。
個所（かしょ） 雛形（ひながた） 閉塞（へいそく）

■謝辞

COBOL 言語仕様は，CODASYL（the Conference on Data Systems Languages：データシステムズ言語協議会）によって，開発された。原開発者に対し謝意を表すとともに，CODASYL の要求に従って次の謝辞を掲げる。なお，この文章は，COBOL の原仕様書「CODASYL COBOL JOURNAL OF DEVELOPMENT 1984」の謝辞の一部を再掲するものである。

いかなる組織であっても，COBOL の原仕様書とその仕様の全体又は一部分を複製すること，マニュアルその他の資料のための土台として原仕様書のアイデアを利用することは自由である。ただし，その場合には，その刊行物のまえがきの一部として，次の謝辞を掲載しなければならない。書評などに短い文章を引用するときは，"COBOL" という名称を示せば謝辞全体を掲載する必要はない。

COBOL は産業界の言語であり，特定の団体や組織の所有物ではない。
CODASYL COBOL 委員会又は仕様変更の提案者は，このプログラミングシステムと言語の正確さや機能について，いかなる保証も与えない。さらに，それに関連する責任も負わない。

次に示す著作権表示付資料の著作者及び著作権者

FLOW-MATIC（Sperry Rand Corporation の商標），Programming for the Univac(R) I and II, Data Automation Systems，Sperry Rand Corporation 著作権表示 1958 年，1959 年；
IBM Commercial Translator Form No.F 28-8013，IBM 著作権表示 1959 年；
FACT，DSI 27A 5260-2760，Minneapolis-Honeywell 著作権表示 1960 年

は，これらの全体又は一部分を COBOL の原仕様書中に利用することを許可した。この許可は，COBOL 原仕様書をプログラミングマニュアルや類似の刊行物に複製したり，利用したりする場合にまで拡張される。

目次

図目次

表目次

1 IDL 文法

この章では，OTM のトランザクションフレームジェネレータで使用する IDL 文法について説明します。

1.1 IDL 定義一覧

ユーザは**ユーザ定義** IDL **インタフェース**を IDL ファイルに定義します。その後，IDL ファイルをトランザクションフレームジェネレータで変換してトランザクションフレームを出力します。トランザクションフレームは，TSC ユーザプロキシ，TSC ユーザアクセプタ，TSC ユーザスケルトンなどを含むプログラミングコードです。

各 IDL 定義をトランザクションフレームジェネレータで使用できるかどうかを表 1-1 に示します。さらに，ほかの IDL ファイルに #include 文で指定される IDL ファイルで IDL 定義を使用できるかどうかを表 1-2 に，operation 定義を使用できるかどうかを表 1-3 に示します。

表 1-1 IDL 定義の使用の可否

種別	使用の可否
module 定義	○
interface 定義	○（継承定義は使用できません）
operation 定義	○（context 指定は使用できません）
typedef 定義	○（array または sequence だけ使用できます）
struct 定義	○
union 定義	○（COBOL では使用できません）
enum 定義	○
constant 定義	×
attribute 定義	×
exception 定義	○

（凡例）
○：使用できます。
×：使用できません。

表 1-2 #include 文で指定される IDL 定義の使用の可否

種別	使用の可否
module 定義	○
interface 定義	×
operation 定義	×
typedef 定義	○（array または sequence だけ使用できます）
struct 定義	○

種別	使用の可否
union 定義	○（COBOL では使用できません）
enum 定義	○
constant 定義	×
attribute 定義	×
exception 定義	○

（凡例）
○：使用できます。
×：使用できません。

表 1-3　operation 定義の使用の可否

定義内容	使用の可否
in 引数	○
out 引数	○
inout 引数	○
リターン値	○（void も使用できます）
raises 定義	○
context 定義	×
oneway 定義	○（セション呼び出しでは使用できません）

（凡例）
○：使用できます。
×：使用できません。

1.2 IDL データ型

IDL データ型は operation 定義の引数およびリターン値，struct 定義のメンバ，union 定義のメンバ，array 型の要素，sequence 型の要素，ならびに exception 定義のメンバに使用できます。

array 型および sequence 型は，必ず typedef してください。

IDL データ型を使用できるかどうかを次の表に示します。

表 1-4 IDL データ型の使用の可否

データ型	使用の可否（C++）	使用の可否（Java）	使用の可否（COBOL）
Primitive	○※1		
struct	○		
sequence	○（最大長指定は使用できません）	○（最大長指定は使用できません）	○（最大長指定は使用できません。要素として使用できるのは Primitive 型と struct 型だけです）
array	○		
union	○※2		×
enum	○		
interface	×（データ型として使用できません）		
any	○（Primitive 型だけ使用できます）	○（Primitive 型だけ使用できます）	×
typedef	○（array 型と sequence 型だけ使用できます）		
Object Reference	×		
fixed	×		

（凡例）

○：使用できます。

×：使用できません。

注※ 1

詳細については，表 1-5 を参照してください。

注※ 2

union 型の case ラベルには，次に示すデータ型または値を使用できません。

- long long 型
- unsigned short 型で 32767 を超える値
- char 型で非表示文字（ASCII コードで 31 以下および 127 以上）の値

表 1-5　Primitive 型データの使用の可否

データ型	使用の可否（C++）	使用の可否（Java）	使用の可否（COBOL）
short	○		
long	○		
unsigned short	○	○（short と同様に扱われます）	○
unsigned long	○	○（long と同様に扱われます）	○
float	○		
double	○		
char	○		
boolean	○		
octet	○		
string	○		
long long	○		
unsigned long long	○	○（long long と同様に扱われます）	○
long double	×		
wchar ※	○	○	×
wstring ※	○（最大長指定は使用できません）	○（最大長指定は使用できません）	×

（凡例）

○：使用できます。

×：使用できません。

注※

wchar 型および wstring 型を使用しても文字コードは変換されません。異なる言語やプラットフォーム間では wchar 型および wstring 型は使用しないでください。

1.3 IDL 文法の注意事項

- IDL 文法に誤りがある場合，エラーが出力されます。IDL 文法が正しくても，トランザクションフレームジェネレータが使用できない定義をする場合，または構文以外の規則に違反がある場合，動作が常に同じになるという保証はありません。
- プリプロセッサに対する指定は，#include 文だけ記述できます。
- IDL 中での前方参照はできません。
- 使用する各プログラミング言語の予約語を，IDL 中の識別子として使用できません。
- IDL 中で使用する識別子は一意になるようにしてください。
- 大文字と小文字に関係なく，"TSC" で始まる識別子は使用できません。
- COBOL 言語中の識別子の最大長は 30 文字です。OTM が生成する識別子も含めて 30 文字以内にしてください。
- COBOL 言語では，一つの IDL ファイル中に複数の interface 定義を記述できません。

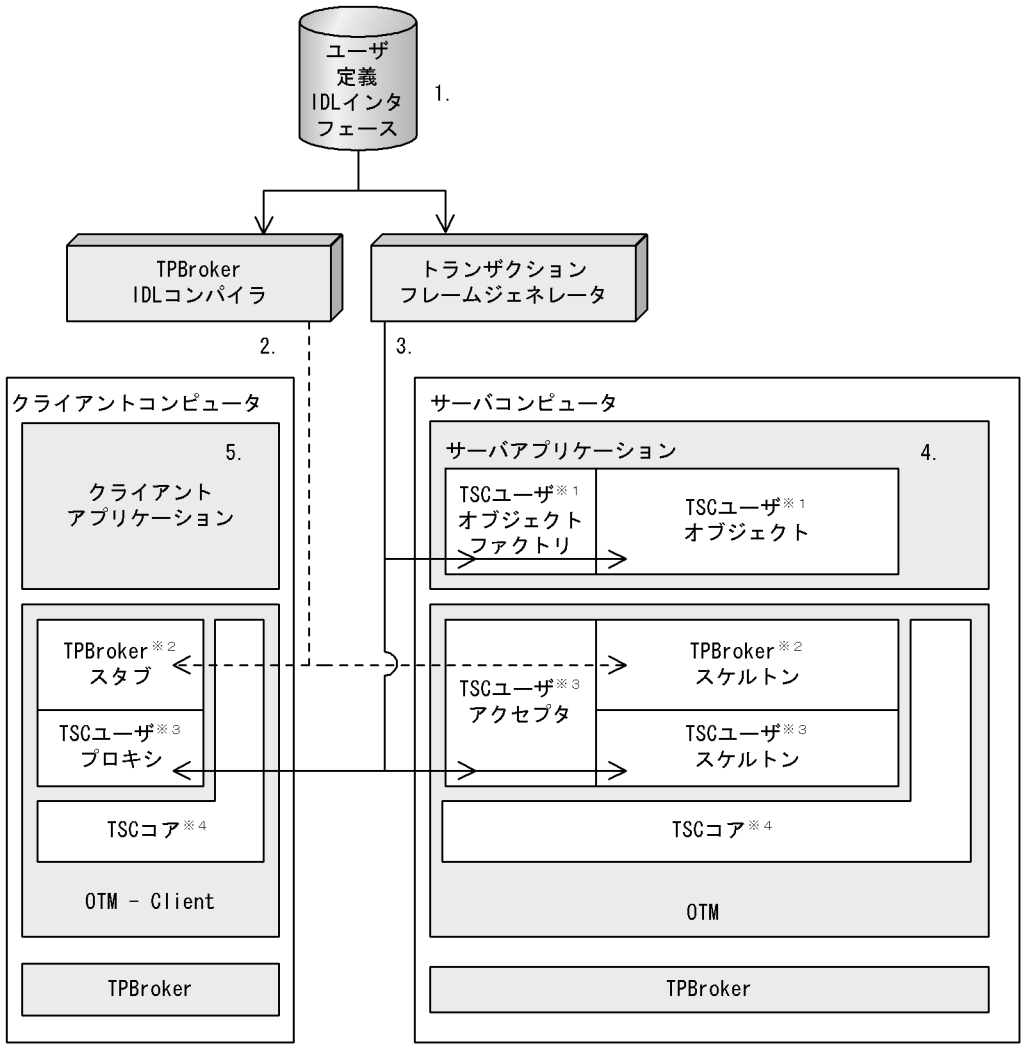
2 アプリケーションプログラムの作成（C++）

この章では，C++ でアプリケーションプログラムを作成する方法について説明します。

2.1　アプリケーションプログラムの作成手順（C++）

C++ でアプリケーションプログラムを作成する手順を次の図に示します。

図 2-1　アプリケーションプログラムを作成する手順（C++）



注※ 1

トランザクションフレームジェネレータが生成するクラスの雛形（**雛形クラス**）です。クラスのシグネチャ，およびコードの一部はすでに記述されています。自動生成されたままの状態では利用できないクラスなので，ユーザはコードを追加する必要があります。

注※ 2

TPBroker の IDL コンパイラが生成するクラス（TPBroker **クラス**）です。

注※ 3

トランザクションフレームジェネレータが生成する基底クラス，およびそのまま使用できるクラス（**ユーザ定義** IDL **インタフェース依存クラス**）です。TPBroker スタブクラスまたは TPBroker スケルトンクラスと継承関係にあります。

注※ 4

システム提供クラスです。OTM の機能の中心となるクラスです。

図中の 1. ～ 5. の手順は次のとおりです。

1. **ユーザ定義** IDL **インタフェースを定義します。**
2. TPBroker **の** IDL **コンパイラを使用して** TPBroker **クラスを生成します。**
3. **トランザクションフレームジェネレータを使用してユーザ定義** IDL **インタフェース依存クラスおよび雛形クラスを生成します。**
4. **次に示すコードを記述してサーバアプリケーションを作成します。**
   - TSC ユーザオブジェクト（雛形クラスの編集）
   - TSC ユーザオブジェクトファクトリ（雛形クラスの編集）
   - サービス登録処理
5. **次に示すコードを記述してクライアントアプリケーションを作成します。**
   - サービス利用処理
   - TSC ユーザプロキシ呼び出し

次に，図中のオブジェクトについて説明します。

● TPBroker **スタブ**

TPBroker の IDL コンパイラが生成するスタブのオブジェクトです。

● TSC **ユーザプロキシ**

クライアントアプリケーションが TSC ユーザオブジェクトを呼び出すための代理オブジェクトです。ユーザ定義 IDL インタフェース依存のオブジェクトです。

● TSC **ユーザオブジェクトファクトリ**

サーバアプリケーションで，TSC ユーザオブジェクトを生成するオブジェクトです。実装についてはユーザが記述します。雛形のオブジェクトです。

● TSC **ユーザオブジェクト**

サーバアプリケーションで，サービスを提供するオブジェクトです。このオブジェクトは，TSC ユーザスケルトンを継承し，実装についてはユーザが記述します。雛形のオブジェクトです。

● TSC **ユーザアクセプタ**

TSC ユーザプロキシから呼び出し要求を受けて，TSC ユーザオブジェクトにリクエストを配送するオブジェクトです。ユーザ定義 IDL インタフェース依存のオブジェクトです。

● **TPBroker スケルトン**

TPBroker の IDL コンパイラが生成するスケルトンのオブジェクトです。

● **TSC ユーザスケルトン**

TSC ユーザオブジェクトのスケルトンです。ユーザ定義 IDL インタフェース依存のオブジェクトです。

2.2　同期型呼び出しをするアプリケーションプログラム（C++）

同期型呼び出しをするアプリケーションプログラムの C++ での作成例を示します。これらのサンプルコードは製品で提供されています。

● ユーザ定義 IDL インタフェースの例

ユーザ定義 IDL インタフェースの例を次に示します。

```
//
// "ABCfile.idl"
//

typedef sequence<octet> sampleOctetSeq;

interface ABC
{
  void call(in sampleOctetSeq in_data,
            out sampleOctetSeq out_data);
};
```

● IDL コンパイラが生成するクラス

TPBroker の IDL コンパイラはユーザ定義 IDL インタフェースから次のファイルを生成します。

- ABCfile_c.hh
- ABCfile_c.cc
- ABCfile_s.hh
- ABCfile_s.cc

● トランザクションフレームジェネレータが生成するクラス

OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。

表 2-1　同期型呼び出しをするアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（C++）

分類	ファイル名	生成するクラス名（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• ABCfile_TSC_c.hh • ABCfile_TSC_c.cc	• ABC_TSCprxy（TSC ユーザプロキシ）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• ABCfile_TSC_s.hh • ABCfile_TSC_s.cc	• ABC_TSCsk（TSC ユーザスケルトン） • ABC_TSCacpt（TSC ユーザアクセプタ）
雛形クラス	• ABCfile_TSC_t.hh • ABCfile_TSC_t.cc	• ABC_TSCimpl（TSC ユーザオブジェクト） • ABC_TSCfactimpl（TSC ユーザオブジェクトファクトリ）

2.2.1 同期型呼び出しをするクライアントアプリケーションの例（C++）

同期型呼び出しをするクライアントアプリケーションの処理の流れとコードの例を示します。

（1） サービス利用処理の流れ

1. TPBroker の初期化処理
2. TPBroker OTM の初期化処理
3. TSC デーモンへの接続
4. TSC ユーザプロキシの生成および各種設定
5. TSC ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）
6. TSC ユーザプロキシの削除
7. TSC デーモンへの接続解放
8. TPBroker OTM の終了処理

（2） サービス利用処理のコード

```
//
// "Client.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <corba.h>

#include <tscadm.h>
#include <tscproxy.h>
#include <tscexcept.h>

#include "ABCfile_TSC_c.hh"

#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

extern void callService(ABC_TSCprxy_ptr abc);

int main(int argc, char** argv)
{

  ////////
  // 1, TPBrokerの初期化処理
  ////////

  CORBA::ORB_ptr orb = 0;

  try
  {
```

```
  // ORBの初期化
  orb = CORBA::ORB_init(argc, argv);
}
catch(CORBA::SystemException& se)
{
  // 例外処理
  cerr << se << endl;
  exit(1);
}

////////
// 2, TPBroker OTMの初期化処理
////////

try
{
  // TSCの初期化
  TSCAdm::initClient(argc, argv, orb);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  exit(1);
}

////////
// 3, TSCデーモンへの接続
////////

TSCDomain_ptr tsc_domain = 0;

try
{
  tsc_domain = new TSCDomain(0, 0);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}
```

```
TSCClient_ptr tsc_client = 0;

try
{
  tsc_client = TSCAdm::getTSCClient(tsc_domain,
  TSCAdm::TSC_ADM_REGULATOR);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 4, TSCユーザプロキシの生成および各種設定
////////

// ユーザ定義IDLインタフェース"ABC"用のTSCProxy生成
ABC_TSCprxy_ptr my_proxy = 0;

try
{
  my_proxy = new ABC_TSCprxy(tsc_client);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::releaseTSCClient(tsc_client);
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}
```

```
////////
// 5, TSCユーザプロキシのメソッド呼び出し
//      （サーバ側のオブジェクトの呼び出し）
////////

try
{
  callService(my_proxy);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_proxy;
    TSCAdm::releaseTSCClient(tsc_client);
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 6, TSCユーザプロキシの削除
////////

delete my_proxy;

////////
// 7, TSCデーモンへの接続解放
////////

try
{
  TSCAdm::releaseTSCClient(tsc_client);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
```

```
      exit(1);
    }
    exit(1);
  }

  ////////
  // 8, TPBroker OTMの終了処理
  ////////

  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    exit(1);
  }

  exit(0);

};
```

(3) TSC ユーザプロキシを呼び出すコード

```
//
// "callService.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <corba.h>

#include <tscadm.h>
#include <tscproxy.h>
#include <tscexcept.h>

#include "ABCfile_TSC_c.hh"

#define SEND_MESSAGE_LENGTH  (256)
#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

void callService(ABC_TSCprxy_ptr abc)
{

  ////////
  // サービスの呼び出し
  ////////

  // in引数の準備
  sampleOctetSeq  req_data;
  req_data.length(SEND_MESSAGE_LENGTH);
```

```
  for(int i=0; i<SEND_MESSAGE_LENGTH; ++i)
  {
    req_data[i] = (unsigned char)(i%256);
  };

  // out引数の準備
  sampleOctetSeq*  res_data;

  try
  {
    // サーバのメソッドの呼び出し
    abc->call(req_data, res_data);
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    throw;
  }

}
```

2.2.2 同期型呼び出しをするサーバアプリケーションの例（C++）

同期型呼び出しをするサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス ABC_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス ABC_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。

（1）TSC ユーザオブジェクト（ABC_TSCimpl）と TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のヘッダのコード

```
//
// "ABCfile_TSC_t.hh"
//
#ifndef _ABCfile_TSC_T_HDR
#define _ABCfile_TSC_T_HDR

#include <tscobject.h>

#include "ABCfile_TSC_s.hh"

class ABC_TSCfactimpl : public TSCObjectFactory
{
public:
  // コンストラクタの引数の数および型を変更することもできます。
  ABC_TSCfactimpl();
```

```
  virtual ~ABC_TSCfactimpl();

  virtual TSCObject_ptr create();
  virtual void destroy(TSCObject_ptr tsc_object);

};

class ABC_TSCimpl : public ABC_TSCsk
{
private:

public:
  // コンストラクタの引数の数および型を変更することもできます。
  ABC_TSCimpl();
  ABC_TSCimpl(const char* _tpbroker_object_name);

  virtual ~ABC_TSCimpl();

  void call(const sampleOctetSeq& in_data,
                sampleOctetSeq*& out_data);

  // メソッドが呼ばれた回数
  CORBA::Long m_counter;

};

#endif  // _ABCfile_TSC_T_HDR
```

（2）TSC ユーザオブジェクト（ABC_TSCimpl）と TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のコード

```
//
//  "ABCfile_TSC_t.cpp"
//
#include "ABCfile_TSC_t.hh"

ABC_TSCfactimpl::ABC_TSCfactimpl()
{
  // Constructor of factory implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのコンストラクタのコードを記述
  // します。引数の数および型を変更することもできます。
}

ABC_TSCfactimpl::~ABC_TSCfactimpl()
{
  // Destructor of factory implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのデストラクタのコードを記述
  // します。
}

TSCObject_ptr
ABC_TSCfactimpl::create()
{
  // Method to create user object.
  // Write user own code.
```

```
  // サーバオブジェクトを生成するコードを記述します。
  // 必要に応じて変更してください。
  return new ABC_TSCimpl();
}

void
ABC_TSCfactimpl::destroy(TSCObject_ptr tsc_obj)
{
  // Method to destroy user object.
  // Write user own code.
  // ここに後処理のコードを記述します。
  // 必要に応じて変更してください。
  delete tsc_obj;
}

ABC_TSCimpl::ABC_TSCimpl()
{
  // Constructor of implementation.
  // Write user own code.
  // TSCユーザオブジェクトのコンストラクタのコードを記述します。
  // 引数の数および型を変更することもできます。
}

ABC_TSCimpl::ABC_TSCimpl(const char* _tpbroker_object_name)
                        : ABC_TSCsk(_tpbroker_object_name)
{
  // Constructor of implementation.
  // Write user own code.
  // TSCユーザオブジェクトのコンストラクタのコードを記述します。
  // 引数の数および型を変更することもできます。
}

ABC_TSCimpl::~ABC_TSCimpl()
{
  // Destructor of implementation.
  // Write user own code.
  // ユーザオブジェクトのデストラクタのコードを記述します。
}

void ABC_TSCimpl::call(const sampleOctetSeq& in_data,
                               sampleOctetSeq*& out_data)
{
  // Operation "::ABC::call".
  // Write user own code.
  // ユーザメソッドのコードを記述します。
  out_data = new sampleOctetSeq();
  out_data->length(0);

  // メソッドが呼ばれた回数を増加させます。
  //（このメソッドの処理は引数の値と無関係です）
  printf("Call method in ABC_TSCimpl¥n");
}
```

（3）サービス登録処理の流れ

1. TPBroker **の初期化処理**

2. TPBroker OTM **の初期化処理**

3. TSC デーモンへの接続
4. TSC ユーザオブジェクトファクトリ，TSC ユーザアクセプタの生成（new），および各種設定
5. TSC ルートアクセプタの生成および各種設定
6. TSC ルートアクセプタの活性化
7. 実行制御の受け渡し
8. TSC ルートアクセプタの非活性化
9. TSC ルートアクセプタの削除
10. TSC ユーザオブジェクトファクトリおよび TSC ユーザアクセプタの削除（delete）
11. TSC デーモンへの接続解放
12. TPBroker OTM の終了処理

（4） サービス登録処理のコード

```
//
// "Server.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <tscadm.h>
#include <tscobject.h>
#include <tscexcept.h>

#include "ABCfile_TSC_t.hh"

#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

int main(int argc, char** argv)
{

  ////////
  // 1, TPBrokerの初期化処理
  ////////

  CORBA::ORB_ptr orb = 0;
  try
  {
    // ORBの初期化
    orb = CORBA::ORB_init(argc, argv);
  }
  catch(CORBA::SystemException& se)
  {
    // 例外処理
    cerr << se << endl;
    exit(1);
  }
```

```
////////
// 2, TPBroker OTMの初期化処理
////////

try
{
  // TSCの初期化
  TSCAdm::initServer(argc, argv, orb);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  exit(1);
}

////////
// 3, TSCデーモンへの接続
////////

TSCDomain_ptr tsc_domain = 0;

try
{
  tsc_domain = new TSCDomain(0, 0);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

TSCServer_ptr tsc_server = 0;

try
{
  // TSCデーモンの参照オブジェクトを取得
  tsc_server = TSCAdm::getTSCServer(tsc_domain);
}
catch(TSCSystemException& se)
```

```
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 4, TSCユーザオブジェクトファクトリ, TSCユーザアクセプタ(new),
//    および各種設定
////////

// ABC_TSCfactimplの生成
TSCObjectFactory_ptr my_obj_fact = new ABC_TSCfactimpl();

// TSCAcceptorの生成
TSCAcceptor_ptr my_acpt = 0;

try
{
  my_acpt = new ABC_TSCacpt(my_obj_fact);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 5, TSCルートアクセプタの生成および各種設定
////////
```

```
// TSCRootAcceptorの生成
TSCRootAcceptor_ptr my_rt_acpt = 0;

try
{
  my_rt_acpt = TSCRootAcceptor::create(tsc_server);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

try
{
  // TSCRootAcceptorへの登録
  my_rt_acpt->registerAcceptor(my_acpt);

  // TSCRootAcceptorのパラレルカウント指定もできます。
  // デフォルト値は1
  // オプション引数でデフォルト値を変更することもできます。
  // my_rt_acpt->setParallelCount(5);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
```

```
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 6, TSCルートアクセプタの活性化
////////

try
{
  // オブジェクトの活性化
  my_rt_acpt->activate("serviceX");
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 7, 実行制御の受け渡し
////////

try
{
  TSCAdm::serverMainloop();
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    my_rt_acpt->deactivate();
```

```
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 8, TSCルートアクセプタの非活性化
////////

try
{
  // オブジェクトの非活性化
  my_rt_acpt->deactivate();
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 9, TSCルートアクセプタの削除
////////

TSCRootAcceptor::destroy(my_rt_acpt);

////////
// 10, TSCユーザオブジェクトファクトリおよびTSCユーザアクセプタの
//     削除(delete)
////////

delete my_acpt;
delete my_obj_fact;
```

```
    ////////
    // 11, TSCデーモンへの接続解放
    ////////

    try
    {
      TSCAdm::releaseTSCServer(tsc_server);
    }
    catch(TSCSystemException& se)
    {
      // 例外処理
      fprintf(stderr, ERR_FORMAT,
         se.getErrorCode(), se.getDetailCode(),
         se.getPlaceCode(), se.getCompletionStatus(),
         se.getMaintenanceCode1(), se.getMaintenanceCode2(),
         se.getMaintenanceCode3(), se.getMaintenanceCode4());
      try
      {
        TSCAdm::endServer();
      }
      catch(TSCSystemException& se)
      {
        exit(1);
      }
      exit(1);
    }

    delete tsc_domain;

    ////////
    // 12, TPBroker OTMの終了処理
    ////////

    try
    {
      TSCAdm::endServer();
    }
    catch(TSCSystemException& se)
    {
      // 例外処理
      fprintf(stderr, ERR_FORMAT,
         se.getErrorCode(), se.getDetailCode(),
         se.getPlaceCode(), se.getCompletionStatus(),
         se.getMaintenanceCode1(), se.getMaintenanceCode2(),
         se.getMaintenanceCode3(), se.getMaintenanceCode4());
      exit(1);
    }

    exit(0);

  };
```

2.2.3　同期型呼び出しをするアプリケーションプログラムの実行時の処理（C++）

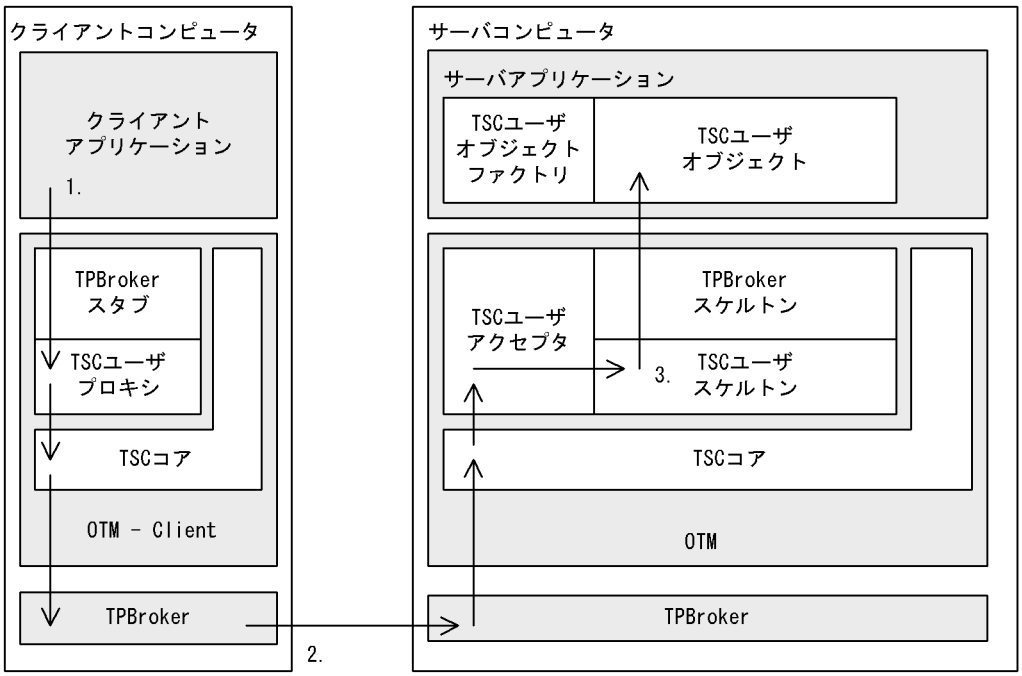
同期型呼び出しをするアプリケーションプログラムを実行した場合の処理シーケンスを示します。

（1）クライアントアプリケーションの開始（C++）

クライアントアプリケーションを開始すると，サービスの利用処理が実行されます。ABC_TSCprxy のメソッドを呼び出すと，ABC_TSCsk を経由し，ユーザが実装した ABC_TSCimpl のメソッドが呼び出されます。

クライアントアプリケーションの開始の流れを次の図に示します。

図 2-2　同期型呼び出しをするクライアントアプリケーションの開始（C++）



1. TSC **ユーザプロキシのメソッドが呼び出されます。**
2. **スケジューリングおよび** TPBroker **の通信が実行されます。**
3. TSC **ユーザオブジェクトのメソッドが呼び出されます。**

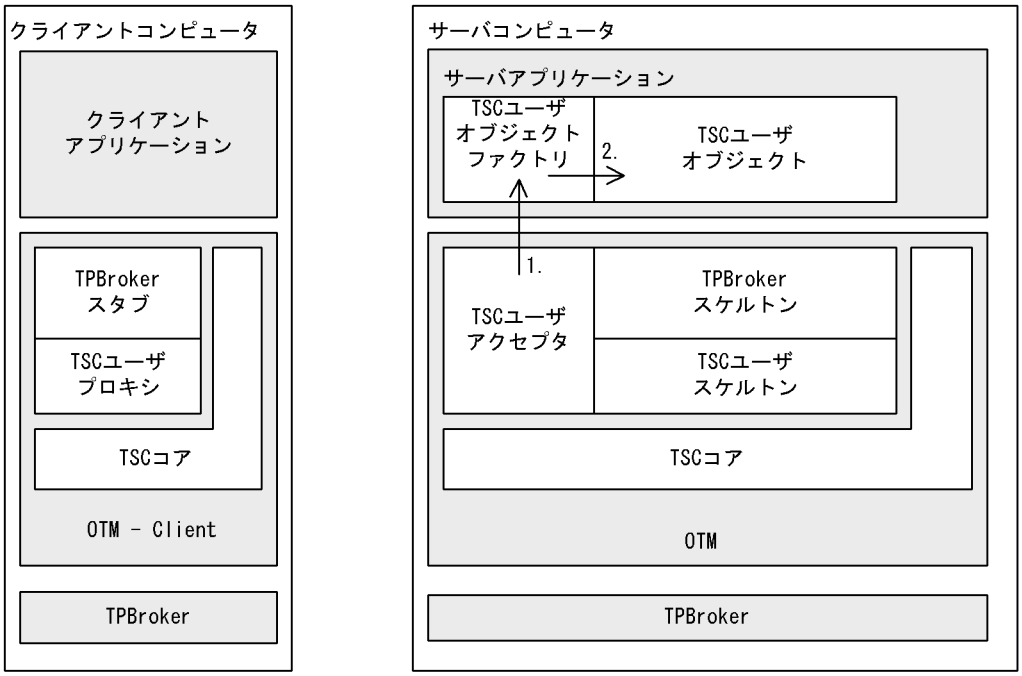
（2）サーバアプリケーションの開始（C++）

サーバアプリケーションを開始すると，サービスの登録処理が実行されます。TSCRootAcceptor の activate メソッドを呼び出すと，ABC_TSCfactimpl に

ABC_TSCimpl の create が呼び出されます。

サーバアプリケーションの開始の流れを次の図に示します。

図 2-3　同期型呼び出しをするサーバアプリケーションの開始（C++）



1. TSC **ユーザオブジェクトの活性化処理が実行されます。**
2. TSC **ユーザオブジェクトのインプリメンテーションが生成されます。**

2.3　非応答型呼び出しをするアプリケーションプログラム（C++）

非応答型呼び出しをするアプリケーションプログラムの C++ での作成例を示します。これらのサンプルコードは製品で提供されています。

● ユーザ定義 IDL インタフェースの例

ユーザ定義 IDL インタフェースの例を次に示します。

```
//
// "XYZfile.idl"
//

interface XYZ
{
  oneway void callOnly(in long in_data);
};
```

● IDL コンパイラが生成するクラス

TPBroker の IDL コンパイラはユーザ定義 IDL インタフェースから次のファイルを生成します。

- XYZfile_c.hh
- XYZfile_c.cc
- XYZfile_s.hh
- XYZfile_s.cc

● トランザクションフレームジェネレータが生成するクラス

OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。

表 2-2　非応答型呼び出しをするアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（C++）

分類	ファイル名	生成するクラス名（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• XYZfile_TSC_c.hh • XYZfile_TSC_c.cc	• XYZ_TSCprxy（TSC ユーザプロキシ）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• XYZfile_TSC_s.hh • XYZfile_TSC_s.cc	• XYZ_TSCsk（TSC ユーザスケルトン） • XYZ_TSCacpt（TSC ユーザアクセプタ）
雛形クラス	• XYZfile_TSC_t.hh • XYZfile_TSC_t.cc	• XYZ_TSCimpl（TSC ユーザオブジェクト） • XYZ_TSCfactimpl（TSC ユーザオブジェクトファクトリ）

2.3.1 非応答型呼び出しをするクライアントアプリケーションの例（C++）

非応答型呼び出しをするクライアントアプリケーションの処理の流れとコードの例を示します。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

（1） サービス利用処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
6. TSC **ユーザプロキシの削除**
7. TSC **デーモンへの接続解放**
8. TPBroker OTM **の終了処理**

（2） サービス利用処理のコード

```
//
// "Client.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <corba.h>

#include <tscadm.h>
#include <tscproxy.h>
#include <tscexcept.h>

#include "XYZfile_TSC_c.hh"

#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

extern void callOnlyService(XYZ_TSCprxy_ptr xyz);

int main(int argc, char** argv)
{

  ////////
  // 1, TPBrokerの初期化処理
  ////////

  CORBA::ORB_ptr orb = 0;

  try
  {
```

```
  // ORBの初期化
  orb = CORBA::ORB_init(argc, argv);
}
catch(CORBA::SystemException& se)
{
  // 例外処理
  cerr << se << endl;
  exit(1);
}

////////
// 2, TPBroker OTMの初期化処理
////////

try
{
  // TSCの初期化
  TSCAdm::initClient(argc, argv, orb);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  exit(1);
}

////////
// 3, TSCデーモンへの接続
////////

TSCDomain_ptr tsc_domain = 0;

try
{
  tsc_domain = new TSCDomain(0, 0);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}
```

```
TSCClient_ptr tsc_client = 0;

try
{
  tsc_client = TSCAdm::getTSCClient(tsc_domain,
               TSCAdm::TSC_ADM_REGULATOR);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 4, TSCユーザプロキシの生成および各種設定
////////

// ユーザ定義IDLインタフェース"XYZ"用のTSCPrxy生成
XYZ_TSCprxy_ptr my_proxy = 0;

try
{
  my_proxy = new XYZ_TSCprxy(tsc_client);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::releaseTSCClient(tsc_client);
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
 }
```

```
////////
// 5, TSCユーザプロキシのメソッド呼び出し
//      （サーバ側のオブジェクトの呼び出し）
////////

try
{
  callOnlyService(my_proxy);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_proxy;
    TSCAdm::releaseTSCClient(tsc_client);
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 6, TSCユーザプロキシの削除
////////

delete my_proxy;

////////
// 7, TSCデーモンへの接続解放
////////

try
{
  TSCAdm::releaseTSCClient(tsc_client);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
```

```
      exit(1);
    }
    exit(1);
  }

  ////////
  // 8, TPBroker OTMの終了処理
  ////////

  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    exit(1);
  }

  exit(0);

};
```

(3) TSC ユーザプロキシを呼び出すコード

```
//
// "callOnlyService.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <corba.h>

#include <tscadm.h>
#include <tscproxy.h>
#include <tscexcept.h>

#include "XYZfile_TSC_c.hh"

#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

void callOnlyService(XYZ_TSCprxy_ptr xyz)
{

  ////////
  // サービスの呼び出し
  ////////

  // Setup of in-argument
  CORBA::Long req_data = 100;

  try
```

```
  {
    // サーバのメソッド呼び出し
    xyz->callOnly(req_data);
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    throw;
  }

}
```

2.3.2 非応答型呼び出しをするサーバアプリケーションの例（C++）

非応答型呼び出しをするサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス XYZ_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス XYZ_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。

（1）TSC ユーザオブジェクト（XYZ_TSCimpl）と TSC ユーザオブジェクトファクトリ（XYZ_TSCfactimpl）のヘッダのコード

```
//
// "XYZfile_TSC_t.hh"
//
#ifndef _XYZfile_TSC_T_HDR
#define _XYZfile_TSC_T_HDR

#include <tscobject.h>

#include "XYZfile_TSC_s.hh"

class XYZ_TSCfactimpl : public TSCObjectFactory
{
public:
  // コンストラクタの引数の数および型を変更することもできます。
  XYZ_TSCfactimpl();
  virtual ~XYZ_TSCfactimpl();

  virtual TSCObject_ptr create();
  virtual void destroy(TSCObject_ptr tsc_object);

};
```

```
class XYZ_TSCimpl : public XYZ_TSCsk
{
private:

public:
  // コンストラクタの引数の数および型を変更することもできます。
  XYZ_TSCimpl();
  XYZ_TSCimpl(const char* _tpbroker_object_name);

  virtual ~XYZ_TSCimpl();

  void callOnly(CORBA::Long in_data);

};

#endif  // _XYZfile_TSC_T_HDR
```

（2）TSC ユーザオブジェクト（XYZ_TSCimpl）と TSC ユーザオブジェクトファクトリ（XYZ_TSCfactimpl）のコード

```
//
// "XYZfile_TSC_t.cpp"
//
#include "XYZfile_TSC_t.hh"

XYZ_TSCfactimpl::XYZ_TSCfactimpl()
{
  // Constructor of factory implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのコンストラクタのコードを記述して
  // ください。
  // コンストラクタの引数の数および型を変更することもできます。
}

XYZ_TSCfactimpl::~XYZ_TSCfactimpl()
{
  // Destructor of factory implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのデストラクタのコードを記述して
  // ください。
}

TSCObject_ptr
XYZ_TSCfactimpl::create()
{
  // Method to create user object.
  // Write user own code.
  // サーバオブジェクトを生成するコードを記述します。
  // 必要に応じて変更してください。
  return new XYZ_TSCimpl();
}

void
XYZ_TSCfactimpl::destroy(TSCObject_ptr tsc_obj)
{
  // Method to destroy user object.
  // Write user own code.
```

```
  // 後処理のコードを記述します。
  // 必要に応じて変更してください。
  delete tsc_obj;
}

XYZ_TSCimpl::XYZ_TSCimpl()
{
  // Constructor of implementation.
  // Write user own code.
  // ユーザオブジェクトのコンストラクタのコードを記述してください。
  // コンストラクタの引数の数および型を変更することもできます。
}

XYZ_TSCimpl::XYZ_TSCimpl(const char* _tpbroker_object_name)
      : XYZ_TSCsk(_tpbroker_object_name)
{
  // Constructor of implementation.
  // Write user own code.
  // ユーザオブジェクトのコンストラクタのコードを記述してください。
  // コンストラクタの引数の数および型を変更することもできます。
}

XYZ_TSCimpl::~XYZ_TSCimpl()
{
  // Destructor of implementation.
  // Write user own code.
  // ユーザオブジェクトのデストラクタのコードを記述してください。
}

void
XYZ_TSCimpl::callOnly(CORBA::Long in_data)
{
  // Operation "::XYZ::callOnly".
  // Write user own code.
  // ユーザメソッドのコードを記述します。
  printf("Call method in XYZ_TSCimpl. In_data = %d¥n",
         in_data);
}
```

（3） サービス登録処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザオブジェクトファクトリ，**TSC **ユーザアクセプタの生成（**new**），および各種設定**
5. TSC **ルートアクセプタの生成および各種設定**
6. TSC **ルートアクセプタの活性化**
7. **実行制御の受け渡し**
8. TSC **ルートアクセプタの非活性化**

9. TSC ルートアクセプタの削除

10. TSC ユーザオブジェクトファクトリおよび TSC ユーザアクセプタの削除（delete）

11. TSC デーモンへの接続解放

12. TPBroker OTM の終了処理

（4） サービス登録処理のコード

```
//
//  "Server.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <tscadm.h>
#include <tscobject.h>
#include <tscexcept.h>

#include "XYZfile_TSC_t.hh"

#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

int main(int argc, char** argv)
{

  ////////
  // 1, TPBrokerの初期化処理
  ////////

  CORBA::ORB_ptr orb = 0;
  try
  {
    // ORBの初期化
    orb = CORBA::ORB_init(argc, argv);
  }
  catch(CORBA::SystemException& se)
  {
    // 例外処理
    cerr << se << endl;
    exit(1);
  }

  ////////
  // 2, TPBroker OTMの初期化処理
  ////////

  try
  {
    // TSCの初期化
    TSCAdm::initServer(argc, argv, orb);
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
```

```
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  exit(1);
}

////////
// 3, TSCデーモンへの接続
////////

TSCDomain_ptr tsc_domain = 0;

try
{
  tsc_domain = new TSCDomain(0, 0);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

TSCServer_ptr tsc_server = 0;

try
{
  // TSCデーモンの参照オブジェクトを取得
  tsc_server = TSCAdm::getTSCServer(tsc_domain);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
```

```
    exit(1);
  }
  exit(1);
}

////////
// 4, TSCユーザオブジェクトファクトリ, TSCユーザアクセプタ(new),
//    および各種設定
////////

// XYZ_TSCfactimplの生成
TSCObjectFactory_ptr my_obj_fact = new XYZ_TSCfactimpl();

// TSCAcceptorの生成
TSCAcceptor_ptr my_acpt = 0;

try
{
  my_acpt = new XYZ_TSCacpt(my_obj_fact);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 5, TSCルートアクセプタの生成および各種設定
////////

// TSCRootAcceptorの生成
TSCRootAcceptor_ptr my_rt_acpt = 0;

try
{
  my_rt_acpt = TSCRootAcceptor::create(tsc_server);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
```

```
      se.getPlaceCode(), se.getCompletionStatus(),
      se.getMaintenanceCode1(), se.getMaintenanceCode2(),
      se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

try
{
  // TSCRootAcceptorへの登録
  my_rt_acpt->registerAcceptor(my_acpt);

  // TSCRootAcceptorのパラレルカウント指定もできます。
  // デフォルト値は1
  // オプション引数でデフォルト値を変更することもできます。
  // my_rt_acpt->setParallelCount(5);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 6, TSCルートアクセプタの活性化
////////

try
{
```

```
  // オブジェクトの活性化
  my_rt_acpt->activate("serviceX");
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 7, 実行制御の受け渡し
////////

try
{
  TSCAdm::serverMainloop();
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    my_rt_acpt->deactivate();
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}
```

```
////////
// 8, TSCルートアクセプタの非活性化
////////

try
{
  // オブジェクトの非活性化
  my_rt_acpt->deactivate();
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 9, TSCルートアクセプタの削除
////////

TSCRootAcceptor::destroy(my_rt_acpt);

////////
// 10, TSCユーザオブジェクトファクトリおよびTSCユーザアクセプタの
//     削除(delete)
////////

delete my_acpt;
delete my_obj_fact;

////////
// 11, TSCデーモンへの接続解放
////////

try
{
  TSCAdm::releaseTSCServer(tsc_server);
}
catch(TSCSystemException& se)
{
  // 例外処理
```

```
      fprintf(stderr, ERR_FORMAT,
         se.getErrorCode(), se.getDetailCode(),
         se.getPlaceCode(), se.getCompletionStatus(),
         se.getMaintenanceCode1(), se.getMaintenanceCode2(),
         se.getMaintenanceCode3(), se.getMaintenanceCode4());
      try
      {
        TSCAdm::endServer();
      }
      catch(TSCSystemException& se)
      {
        exit(1);
      }
      exit(1);
    }

    delete tsc_domain;

    ////////
    // 12, TPBroker OTMの終了処理
    ////////

    try
    {
      TSCAdm::endServer();
    }
    catch(TSCSystemException& se)
    {
      // 例外処理
      fprintf(stderr, ERR_FORMAT,
         se.getErrorCode(), se.getDetailCode(),
         se.getPlaceCode(), se.getCompletionStatus(),
         se.getMaintenanceCode1(), se.getMaintenanceCode2(),
         se.getMaintenanceCode3(), se.getMaintenanceCode4());
      exit(1);
    }

    exit(0);

  };
```

2.4　セション呼び出しをするアプリケーションプログラム（C++）

セション呼び出しをするアプリケーションプログラムの C++ での作成例を示します。これらのサンプルコードは製品で提供されています。

ユーザ定義 IDL インタフェースの例，および IDL コンパイラが生成するクラスは，同期型呼び出しの場合と同様です。「2.2　同期型呼び出しをするアプリケーションプログラム（C++）」を参照してください。

● **トランザクションフレームジェネレータが生成するクラス**
OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。

表 2-3　セション呼び出しをするアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（C++）

分類	ファイル名	生成するクラス名（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• ABCfile_TSC_p.hh • ABCfile_TSC_p.cc	• ABC_TSCspxy（TSC ユーザプロキシ）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• ABCfile_TSC_s.hh • ABCfile_TSC_s.cc	• ABC_TSCsk（TSC ユーザスケルトン） • ABC_TSCacpt（TSC ユーザアクセプタ）
雛形クラス	• ABCfile_TSC_t.hh • ABCfile_TSC_t.cc	• ABC_TSCimpl（TSC ユーザオブジェクト） • ABC_TSCfactimpl（TSC ユーザオブジェクトファクトリ）

2.4.1　セション呼び出しをするクライアントアプリケーションの例（C++）

セション呼び出しをするクライアントアプリケーションの処理の流れとコードの例を示します。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

（1）サービス利用処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
6. TSC **ユーザプロキシの削除**

7. TSC デーモンへの接続解放

8. TPBroker OTM の終了処理

（2） サービス利用処理のコード

```
//
// "Client.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <corba.h>

#include <tscadm.h>
#include <tscproxy.h>
#include <tscexcept.h>

#include "ABCfile_TSC_p.hh"

#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

extern void callSessionService(ABC_TSCspxy_ptr abc);

int main(int argc, char** argv)
{

  ////////
  // 1, TPBrokerの初期化処理
  ////////

  CORBA::ORB_ptr orb = 0;

  try
  {
    // ORBの初期化
    orb = CORBA::ORB_init(argc, argv);
  }
  catch(CORBA::SystemException& se)
  {
    // 例外処理
    cerr << se << endl;
    exit(1);
  }

  ////////
  // 2, TPBroker OTMの初期化処理
  ////////

  try
  {
    // TSCの初期化
    TSCAdm::initClient(argc, argv, orb);
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
```

```
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  exit(1);
}

////////
// 3, TSCデーモンへの接続
////////

TSCDomain_ptr tsc_domain = 0;

try
{
  tsc_domain = new TSCDomain(0, 0);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

TSCClient_ptr tsc_client = 0;

try
{
  tsc_client = TSCAdm::getTSCClient(tsc_domain,
  TSCAdm::TSC_ADM_REGULATOR);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
```

```
    exit(1);
  }
  exit(1);
}

////////
// 4, TSCユーザプロキシの生成および各種設定
////////

// ユーザ定義IDLインタフェース"ABC"用のTSCspxy生成
ABC_TSCspxy_ptr my_proxy = 0;

try
{
  my_proxy = new ABC_TSCspxy(tsc_client);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::releaseTSCClient(tsc_client);
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 5, TSCユーザプロキシのメソッド呼び出し
//     （サーバ側のオブジェクトの呼び出し）
////////

try
{
  callSessionService(my_proxy);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_proxy;
    TSCAdm::releaseTSCClient(tsc_client);
    TSCAdm::endClient();
```

```
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 6, TSCユーザプロキシの削除
////////

delete my_proxy;

////////
// 7, TSCデーモンへの接続解放
////////

try
{
  TSCAdm::releaseTSCClient(tsc_client);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 8, TPBroker OTMの終了処理
////////

try
{
  TSCAdm::endClient();
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  exit(1);
```

```
  }

  exit(0);

};
```

(3) TSC ユーザプロキシを呼び出すコード

```
//
// "callSessionService.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <corba.h>

#include <tscadm.h>
#include <tscproxy.h>
#include <tscexcept.h>

#include "ABCfile_TSC_p.hh"

#define SEND_MESSAGE_LENGTH  (256)
#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

void callSessionService(ABC_TSCspxy_ptr abc)
{

  ////////
  // 1, セションの開始
  ////////

  try
  {
    abc->_TSCStart();
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    throw;
  }

  ////////
  // 2, サービスの呼び出し
  ////////

  // in引数の準備
  sampleOctetSeq  req_data;
  req_data.length(SEND_MESSAGE_LENGTH);
  for(int i=0; i<SEND_MESSAGE_LENGTH; ++i)
  {
```

```
    req_data[i] = (unsigned char)(i%256);
  };

  // out引数の準備
  sampleOctetSeq*  res_data;

  try
  {
    for(int i=0; i<3; ++i)
    {
      // サーバのメソッドの呼び出し
      abc->call(req_data, res_data);
    }
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    try
    {
      abc->_TSCStop();
    }
    catch(TSCSystemException& se)
    {}
    throw;
  }

  ////////
  // 3, セションの停止
  ////////

  try
  {
    abc->_TSCStop();
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    throw;
  }

}
```

2.4.2 セション呼び出しをするサーバアプリケーションの例（C++）

同期型呼び出しの場合と同様です。「2.2.2　同期型呼び出しをするサーバアプリケーショ

ンの例（C++)」を参照してください。

2.5　TSCContext を利用するアプリケーションプログラム（C++）

TSCContext を利用するアプリケーションプログラムの C++ での作成例を示します。これらのサンプルコードは製品で提供されています。

ユーザ定義 IDL インタフェースの例，IDL コンパイラが生成するクラス，およびトランザクションフレームジェネレータが生成するクラスは，同期型呼び出しの場合と同様です。「2.2　同期型呼び出しをするアプリケーションプログラム（C++）」を参照してください。

2.5.1　TSCContext を利用するクライアントアプリケーションの例（C++）

TSCContext を利用するクライアントアプリケーションの処理の流れとコードの例を示します。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

（1）サービス利用処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **コンテキストへのユーザデータの設定**
6. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
7. TSC **ユーザプロキシの削除**
8. TSC **デーモンへの接続解放**
9. TPBroker OTM **の終了処理**

（2）サービス利用処理のコード

```
//
// "Client.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <corba.h>

#include <tscadm.h>
#include <tscproxy.h>
```

```
#include <tscexcept.h>
#include <tsccontext.h>

#include "ABCfile_TSC_c.hh"

#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

extern void callService(ABC_TSCprxy_ptr abc);

int main(int argc, char** argv)
{

  ////////
  // 1, TPBrokerの初期化処理
  ////////

  CORBA::ORB_ptr orb = 0;

  try
  {
    // ORBの初期化
    orb = CORBA::ORB_init(argc, argv);
  }
  catch(CORBA::SystemException& se)
  {
    // 例外処理
    cerr << se << endl;
    exit(1);
  }

  ////////
  // 2, TPBroker OTMの初期化処理
  ////////

  try
  {
    // TSCの初期化
    TSCAdm::initClient(argc, argv, orb);
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    exit(1);
  }

  ////////
  // 3, TSCデーモンへの接続
  ////////

  TSCDomain_ptr tsc_domain = 0;

  try
```

```
{
  tsc_domain = new TSCDomain(0, 0);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

TSCClient_ptr tsc_client = 0;

try
{
  tsc_client = TSCAdm::getTSCClient(tsc_domain,
               TSCAdm::TSC_ADM_REGULATOR);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
   try
   {
     TSCAdm::endClient();
   }
   catch(TSCSystemException& se)
   {
     exit(1);
   }
   exit(1);
 }

////////
// 4, TSCユーザプロキシの生成および各種設定
////////

// ユーザ定義IDLインタフェース"ABC"用のTSCProxy生成
ABC_TSCprxy_ptr my_proxy = 0;

try
{
  my_proxy = new ABC_TSCprxy(tsc_client);
```

```
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::releaseTSCClient(tsc_client);
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 5, TSCコンテキストへのユーザデータの設定
////////

// TSCコンテキストの取得
TSCContext_ptr ctx = my_proxy->_TSCContext();

// ユーザIDの取得
ctx->setUserData((unsigned char*)"UserID:1111", 12,
                 TSC_FALSE);

////////
// 6, TSCユーザプロキシのメソッド呼び出し
//     （サーバ側のオブジェクトの呼び出し）
////////

try
{
  callService(my_proxy);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_proxy;
    TSCAdm::releaseTSCClient(tsc_client);
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
```

```
  }
  exit(1);
}

////////
// 7, TSCユーザプロキシの削除
////////

delete my_proxy;

////////
// 8, TSCデーモンへの接続解放
////////

try
{
  TSCAdm::releaseTSCClient(tsc_client);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 9, TPBroker OTMの終了処理
////////

try
{
  TSCAdm::endClient();
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  exit(1);
}

exit(0);
```

```
};
```

（3） TSC ユーザプロキシを呼び出すコード

同期型呼び出しの場合と同様です。「2.2.1(3)　TSC ユーザプロキシを呼び出すコード」を参照してください。

2.5.2 TSCContext を利用するサーバアプリケーションの例（C++）

TSCContext を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス ABC_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス ABC_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。

（1） TSC ユーザオブジェクト（ABC_TSCimpl）と TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のヘッダのコード

```
//
// "ABCfile_TSC_t.hh
//
#ifndef _ABCfile_TSC_T_HDR
#define _ABCfile_TSC_T_HDR

#include <tscobject.h>

#include "ABCfile_TSC_s.hh"

class ABC_TSCfactimpl : public TSCObjectFactory
{
public:
  // コンストラクタの引数の数および型を変更することもできます。
  ABC_TSCfactimpl();
  virtual ~ABC_TSCfactimpl();

  virtual TSCObject_ptr create();
  virtual void destroy(TSCObject_ptr tsc_object);

};

class ABC_TSCimpl : public ABC_TSCsk
{
private:

public:
  // コンストラクタの引数の数および型を変更することもできます。
  ABC_TSCimpl();
  ABC_TSCimpl(const char* _tpbroker_object_name);
```

```
  virtual ~ABC_TSCimpl();

  void call(const sampleOctetSeq& in_data,
                sampleOctetSeq*& out_data);

};

#endif  // _ABCfile_TSC_T_HDR
```

（2）TSC ユーザオブジェクト（ABC_TSCimpl）と TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のコード

```
//
// "ABCfile_TSC_t.cpp"
//
#include "ABCfile_TSC_t.hh"

#include <tsccontext.h>

ABC_TSCfactimpl::ABC_TSCfactimpl()
{
  // Constructor of factory implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのコンストラクタのコードを記述
  // します。引数の数および型を変更することもできます。

}

ABC_TSCfactimpl::~ABC_TSCfactimpl()
{
    // Destructor of factory implementation.
    // Write user own code.
    // TSCユーザオブジェクトファクトリのデストラクタのコードを記述
    // します。
}

TSCObject_ptr
ABC_TSCfactimpl::create()
{
  // Method to create user object.
  // Write user own code.
  // サーバオブジェクトを生成するコードを記述します。
  // 必要に応じて変更してください。
  return new ABC_TSCimpl();
}

void
ABC_TSCfactimpl::destroy(TSCObject_ptr tsc_obj)
{
  // Method to destroy user object.
  // Write user own code.
  // ここに後処理のコードを記述します。
  // 必要に応じて変更してください。
  delete tsc_obj;
}

ABC_TSCimpl::ABC_TSCimpl()
```

```
{
  // Constructor of implementation.
  // Write user own code.
  // TSCユーザオブジェクトのコンストラクタのコードを記述します。
  // 数の数および型を変更することもできます。
}

ABC_TSCimpl::ABC_TSCimpl(const char* _tpbroker_object_name)
       : ABC_TSCsk(_tpbroker_object_name)
{
  // Constructor of implementation.
  // Write user own code.
  // TSCユーザオブジェクトのコンストラクタのコードを記述します。
  // 数の数および型を変更することもできます。
}

ABC_TSCimpl::~ABC_TSCimpl()
{
  // Destructor of implementation.
  // Write user own code.
  // TSCユーザオブジェクトのデストラクタのコードを記述します。
}

void
ABC_TSCimpl::call(const sampleOctetSeq& in_data, sampleOctetSeq*&
out_data)
{
  // Operation "::ABC::call".
  // Write user own code.
  // ユーザメソッドのコードを記述します。

  // TSCコンテキストの取得
  TSCContext_ptr ctx = _TSCContext();

  // ユーザデータの取得
  TSCUChar* data = ctx->getUserData();
  TSCInt data_size = ctx->getUserDataLength();
  out_data = new sampleOctetSeq();
  out_data->length(0);
  printf("Call method in ABC_TSCimpl¥n");
}
```

(3) サービス登録処理の流れ・コード

同期型呼び出しの場合と同様です。「2.2.2(3)　サービス登録処理の流れ」,「2.2.2(4)　サービス登録処理のコード」を参照してください。

2.6　TSCThread を利用するアプリケーションプログラム（C++）

TSCThread を利用するアプリケーションプログラムの C++ での作成例を示します。これらのサンプルコードは製品で提供されています。

ユーザ定義 IDL インタフェースの例，IDL コンパイラが生成するクラス，およびトランザクションフレームジェネレータが生成するクラスは，同期型呼び出しの場合と同様です。「2.2　同期型呼び出しをするアプリケーションプログラム（C++）」を参照してください。

2.6.1　TSCThread を利用するクライアントアプリケーションの例（C++）

同期型呼び出しの場合と同様です。「2.2.1　同期型呼び出しをするクライアントアプリケーションの例（C++）」を参照してください。

2.6.2　TSCThread を利用するサーバアプリケーションの例（C++）

TSCThread を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス ABC_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス ABC_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。さらに，TSC ユーザスレッドとして TSCThread の派生クラスを記述し，TSC ユーザスレッドファクトリとして TSCThreadFactory の派生クラスを記述します。

（1）TSC ユーザオブジェクト（ABC_TSCimpl）と TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のヘッダのコード

```
//
// "ABCfile_TSC_t.hh"
//
#ifndef _ABCfile_TSC_T_HDR
#define _ABCfile_TSC_T_HDR

#include <tscobject.h>

#include "ABCfile_TSC_s.hh"

class ABC_TSCfactimpl : public TSCObjectFactory
```

```
{
public:
  // コンストラクタの引数の数および型を変更することもできます。
  ABC_TSCfactimpl();
  virtual ~ABC_TSCfactimpl();

  virtual TSCObject_ptr create();
  virtual void destroy(TSCObject_ptr tsc_object);

};

class ABC_TSCimpl : public ABC_TSCsk
{
private:

public:
  // コンストラクタの引数の数および型を変更することもできます。
  ABC_TSCimpl();
  ABC_TSCimpl(const char* _tpbroker_object_name);

  virtual ~ABC_TSCimpl();

  void call(const sampleOctetSeq& in_data,
                sampleOctetSeq*& out_data);

};

#endif  // _ABCfile_TSC_T_HDR
```

（2）TSC ユーザオブジェクト（ABC_TSCimpl）と TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のコード

```
//
// "ABCfile_TSC_t.cpp"
//
#include "ABCfile_TSC_t.hh"

#include "UserThread.h"

ABC_TSCfactimpl::ABC_TSCfactimpl()
{
  // Constructor of factory implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのコンストラクタのコードを記述
  // します。引数の数および型を変更することもできます。
}

ABC_TSCfactimpl::~ABC_TSCfactimpl()
{
  // Destructor of factory implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのデストラクタのコードを記述
  // します。
}

TSCObject_ptr
ABC_TSCfactimpl::create()
{
```

```
  // Method to create user object.
  // Write user own code.
  // サーバオブジェクトを生成するコードを記述します。
  // 必要に応じて変更してください。
  return new ABC_TSCimpl();
}

void
ABC_TSCfactimpl::destroy(TSCObject_ptr tsc_obj)
{
  // Method to destroy user object.
  // Write user own code.
  // ここに後処理のコードを記述します。
  // 必要に応じて変更してください。
  delete tsc_obj;
}

ABC_TSCimpl::ABC_TSCimpl()
{
  // Constructor of implementation.
  // Write user own code.
  // TSCユーザオブジェクトのコンストラクタのコードを記述します。
  // 引数の数および型を変更することもできます。
}

ABC_TSCimpl::ABC_TSCimpl(const char* _tpbroker_object_name)
       : ABC_TSCsk(_tpbroker_object_name)
{
  // Constructor of implementation.
  // Write user own code.
  // TSCユーザオブジェクトのコンストラクタのコードを記述します。
  // 引数の数および型を変更することもできます。
}

ABC_TSCimpl::~ABC_TSCimpl()
{
  // Destructor of implementation.
  // Write user own code.
  // TSCユーザオブジェクトのデストラクタのコードを記述します。
}

void ABC_TSCimpl::call(const sampleOctetSeq& in_data,
                              sampleOctetSeq*& out_data)
{
  // Operation "::ABC::call".
  // Write user own code.
  // ユーザメソッドのコードを記述します。

  // TSCユーザスレッドを取得
  // ここで取得したオブジェクトを消去してはいけません。
  TSCThread_ptr my_thr = this->_TSCThread();

  // ユーザクラスにキャスト
  // （仮想継承を使用している場合は，dynamic_castを利用して
  // ください）
  UserTImpl* my_thr_impl = (UserTImpl*)my_thr;

  // UserTImplのメソッドを呼び出し，値を取得します。
```

```
  CORBA::Long thr_value = my_thr_impl->getValue();

  // メソッドが呼ばれた回数を増加させます。
  // （このメソッドの処理は引数の値と無関係です）
  out_data = new sampleOctetSeq();
  out_data->length(0);
  printf("Call method in ABC_TSCimpl¥n");

}
```

（3）TSC ユーザスレッド（UserTImpl）と TSC ユーザスレッドファクトリ（UserFactImpl）のヘッダのコード

```
//
// "UserThread.h"
//

#include <tscobject.h>
#include <tscexcept.h>

class UserTImpl : public TSCThread
{
public:
  UserTImpl(CORBA::Long init_info);
  virtual ~UserTImpl();

  TSCInt getValue();

protected:
  //
  TSCInt m_value;
};


class UserTFactImpl : public TSCThreadFactory
{
public:
  UserTFactImpl();
  virtual ~UserTFactImpl();

  virtual TSCThread_ptr create();

  virtual void destroy(TSCThread_ptr tsc_object);
};
```

（4）TSC ユーザスレッド（UserTImpl）と TSC ユーザスレッドファクトリ（UserFactImpl）のコード

```
//
// "UserThread.cpp
//
#include <tscobject.h>
#include <tscexcept.h>

#include "UserThread.h"

UserTImpl::UserTImpl(CORBA::Long init_value)
```

```
  : m_value(init_value)
{}

UserTImpl::~UserTImpl()
{}

//
//ユーザ定義のメソッド
//
TSCInt
UserTImpl::getValue()
{
  return m_value;
}

UserTFactImpl::UserTFactImpl()
{}

UserTFactImpl::~UserTFactImpl()
{}

TSCThread_ptr
UserTFactImpl::create()
{
  //TSCユーザスレッドを生成します。
  TSCThread_ptr usr_thr = new UserTImpl(222);
  return usr_thr;
}

void
UserTFactImpl::destroy(TSCThread_ptr tsc_thr)
{
  //TSCユーザスレッドを削除します。
  delete tsc_thr;
}
```

（5） サービス登録処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザオブジェクトファクトリ，**TSC **ユーザアクセプタの生成（**new**），および各種設定**
5. TSC **ユーザスレッドファクトリの生成（**new**）および各種設定**
6. TSC **ルートアクセプタの生成および各種設定**
7. TSC **ルートアクセプタの活性化**
8. **実行制御の受け渡し**
9. TSC **ルートアクセプタの非活性化**
10. TSC **ルートアクセプタの削除**

11. TSC ユーザオブジェクトファクトリおよび TSC ユーザアクセプタの削除（delete）

12. TSC デーモンへの接続解放

13. TPBroker OTM の終了処理

（6） サービス登録処理のコード

```
//
//  "Server.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <tscadm.h>
#include <tscobject.h>
#include <tscexcept.h>

#include "ABCfile_TSC_t.hh"
#include "UserThread.h"

#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

int main(int argc, char** argv)
{

  ////////
  // 1, TPBrokerの初期化処理
  ////////

  CORBA::ORB_ptr orb = 0;
  try
  {
    // ORBの初期化
    orb = CORBA::ORB_init(argc, argv);
  }
  catch(CORBA::SystemException& se)
  {
    // 例外処理
    cerr << se << endl;
    exit(1);
  }

  ////////
  // 2, TPBroker OTMの初期化処理
  ////////

  try
  {
    // TSCの初期化
    TSCAdm::initServer(argc, argv, orb);
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
```

```
      se.getErrorCode(), se.getDetailCode(),
      se.getPlaceCode(), se.getCompletionStatus(),
      se.getMaintenanceCode1(), se.getMaintenanceCode2(),
      se.getMaintenanceCode3(), se.getMaintenanceCode4());
  exit(1);
}

////////
// 3, TSCデーモンへの接続
////////

TSCDomain_ptr tsc_domain = 0;

try
{
  tsc_domain = new TSCDomain(0, 0);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
      se.getErrorCode(), se.getDetailCode(),
      se.getPlaceCode(), se.getCompletionStatus(),
      se.getMaintenanceCode1(), se.getMaintenanceCode2(),
      se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

TSCServer_ptr tsc_server = 0;

try
{
  // TSCデーモンの参照オブジェクトを取得
  tsc_server = TSCAdm::getTSCServer(tsc_domain);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
      se.getErrorCode(), se.getDetailCode(),
      se.getPlaceCode(), se.getCompletionStatus(),
      se.getMaintenanceCode1(), se.getMaintenanceCode2(),
      se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
```

```
  }
  exit(1);
}

////////
// 4, TSCユーザオブジェクトファクトリ, TSCユーザアクセプタ（new）,
//    および各種設定
////////

// ABC_TSCfactimplの生成
TSCObjectFactory_ptr my_obj_fact = new ABC_TSCfactimpl();

// TSCAcceptorの生成
TSCAcceptor_ptr my_acpt = 0;

try
{
  my_acpt = new ABC_TSCacpt(my_obj_fact);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

//////
// 5, TSCユーザスレッドファクトリの生成（new）および各種設定
//////
TSCThreadFactory_ptr  my_thr_fact = new UserTFactImpl();

////////
// 6, TSCルートアクセプタの生成および各種設定
////////

// TSCRootAcceptorの生成
TSCRootAcceptor_ptr my_rt_acpt = 0;

try
{
  my_rt_acpt = TSCRootAcceptor::create(tsc_server,
                                       my_thr_fact);
}
```

```
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

try
{
  // TSCRootAcceptorへの登録
  my_rt_acpt->registerAcceptor(my_acpt);

  // TSCRootAcceptorのパラレルカウント指定もできます。
  // デフォルト値は1
  // オプション引数でデフォルト値を変更することもできます。
  // my_rt_acpt->setParallelCount(5);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}
```

```
////////
// 7, TSCルートアクセプタの活性化
////////

try
{
  // オブジェクトの活性化
  my_rt_acpt->activate("serviceX");
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    my_rt_acpt->deactivate();
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 8, 実行制御の受け渡し
////////

try
{
  TSCAdm::serverMainloop();
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
    se.getErrorCode(), se.getDetailCode(),
    se.getPlaceCode(), se.getCompletionStatus(),
    se.getMaintenanceCode1(), se.getMaintenanceCode2(),
    se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    my_rt_acpt->deactivate();
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
```

```
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 9, TSCルートアクセプタの非活性化
////////

try
{
  // オブジェクトの非活性化
  my_rt_acpt->deactivate();
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 10, TSCルートアクセプタの削除
////////

TSCRootAcceptor::destroy(my_rt_acpt);

////////
// 11, TSCユーザオブジェクトファクトリおよびTSCユーザアクセプタの
//     削除(delete)
////////

delete my_acpt;
delete my_obj_fact;

////////
// 12, TSCデーモンへの接続解放
////////
```

```
  try
  {
    TSCAdm::releaseTSCServer(tsc_server);
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    try
    {
      TSCAdm::endServer();
    }
    catch(TSCSystemException& se)
    {
      exit(1);
    }
    exit(1);
  }

  delete tsc_domain;

  ////////
  // 13, TPBroker OTMの終了処理
  ////////

  try
  {
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    exit(1);
  }

  exit(0);

};
```

2.7　ユーザ例外通知を利用するアプリケーションプログラム（C++）

ユーザ例外通知を利用するアプリケーションプログラムの C++ での作成例を示します。これらのサンプルコードは製品で提供されています。

● ユーザ定義 IDL インタフェースの例

ユーザ定義 IDL インタフェースの例を次に示します。

```
//
// "EEEfile.idl"
//

exception UserExcept {
  long value;
};

interface EEE
{
  void call() raises (UserExcept);
};
```

● IDL コンパイラが生成するクラス

TPBroker の IDL コンパイラはユーザ定義 IDL インタフェースから次のファイルを生成します。

- EEEfile_c.hh
- EEEfile_c.cc
- EEEfile_s.hh
- EEEfile_s.cc

● トランザクションフレームジェネレータが生成するクラス

OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。

表 2-4　ユーザ例外通知を利用するアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（C++）

分類	ファイル名	生成するクラス名（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• EEEfile_TSC_c.hh • EEEfile_TSC_c.cc	• EEE_TSCprxy（TSC ユーザプロキシ）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• EEEfile_TSC_s.hh • EEEfile_TSC_s.cc	• EEE_TSCsk（TSC ユーザスケルトン） • EEE_TSCacpt（TSC ユーザアクセプタ）
雛形クラス	• EEEfile_TSC_t.hh • EEEfile_TSC_t.cc	• EEE_TSCimpl（TSC ユーザオブジェクト） • EEE_TSCfactimpl（TSC ユーザオブジェクトファクトリ）

2.7.1 ユーザ例外通知を利用するクライアントアプリケーションの例（C++）

ユーザ例外通知を利用するクライアントアプリケーションの処理の流れとコードの例を示します。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

（1） サービス利用処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
6. TSC **ユーザプロキシの削除**
7. TSC **デーモンへの接続解放**
8. TPBroker OTM **の終了処理**

（2） サービス利用処理のコード

```
//
// "Client.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <corba.h>

#include <tscadm.h>
#include <tscproxy.h>
#include <tscexcept.h>

#include "EEEfile_TSC_c.hh"

#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

extern void callService(EEE_TSCprxy_ptr eee);

int main(int argc, char** argv)
{

  ////////
  // 1, TPBrokerの初期化処理
  ////////

  CORBA::ORB_ptr orb = 0;

  try
  {
```

```
  // ORBの初期化
  orb = CORBA::ORB_init(argc, argv);
}
catch(CORBA::SystemException& se)
{
  // 例外処理
  cerr << se << endl;
  exit(1);
}

////////
// 2, TPBroker OTMの初期化処理
////////

try
{
  // TSCの初期化
  TSCAdm::initClient(argc, argv, orb);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  exit(1);
}

////////
// 3, TSCデーモンへの接続
////////

TSCDomain_ptr tsc_domain = 0;

try
{
  tsc_domain = new TSCDomain(0, 0);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}
```

```
TSCClient_ptr tsc_client = 0;

try
{
  tsc_client = TSCAdm::getTSCClient(tsc_domain,
                    TSCAdm::TSC_ADM_REGULATOR);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 4, TSCユーザプロキシの生成および各種設定
////////

// ユーザ定義IDLインタフェース"EEE"用のTSCProxy生成
EEE_TSCprxy_ptr my_proxy = 0;

try
{
  my_proxy = new EEE_TSCprxy(tsc_client);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::releaseTSCClient(tsc_client);
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}
```

```
////////
// 5, TSCユーザプロキシのメソッド呼び出し
//      （サーバ側のオブジェクトの呼び出し）
////////

try
{
  callService(my_proxy);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_proxy;
    TSCAdm::releaseTSCClient(tsc_client);
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 6, TSCユーザプロキシの削除
////////

delete my_proxy;

////////
// 7, TSCデーモンへの接続解放
////////

try
{
  TSCAdm::releaseTSCClient(tsc_client);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
```

```
      exit(1);
    }
    exit(1);
  }

  ////////
  // 8, TPBroker OTMの終了処理
  ////////

  try
  {
    TSCAdm::endClient();
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    exit(1);
  }

  exit(0);

};
```

(3) TSC ユーザプロキシを呼び出すコード

```
//
// "callService.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <corba.h>

#include <tscadm.h>
#include <tscproxy.h>
#include <tscexcept.h>

#include "EEEfile_TSC_c.hh"

#define SEND_MESSAGE_LENGTH  (256)
#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

void callService(EEE_TSCprxy_ptr eee)
{

  ////////
  // サービスの呼び出し
  ////////

  try
  {
    // サーバのメソッドの呼び出し
```

```
    eee->call();
  }
  catch(UserExcept& se)
  {
    cout << "catch = " << se.value << endl;
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr, ERR_FORMAT,
       se.getErrorCode(), se.getDetailCode(),
       se.getPlaceCode(), se.getCompletionStatus(),
       se.getMaintenanceCode1(), se.getMaintenanceCode2(),
       se.getMaintenanceCode3(), se.getMaintenanceCode4());
    throw;
  }

}
```

2.7.2 ユーザ例外通知を利用するサーバアプリケーションの例（C++）

ユーザ例外通知を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス EEE_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス EEE_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。

（1）TSC ユーザオブジェクト（EEE_TSCimpl）と TSC ユーザオブジェクトファクトリ（EEE_TSCfactimpl）のヘッダのコード

```
//
// "UserExcept_TSC_t.hh"
//
#ifndef _EEEfile_TSC_T_HDR
#define _EEEfile_TSC_T_HDR

#include <tscobject.h>

#include "EEEfile_TSC_s.hh"

class EEE_TSCfactimpl : public TSCObjectFactory
{
public:
  // コンストラクタの引数の数および型を変更することもできます。
  EEE_TSCfactimpl();
  virtual ~EEE_TSCfactimpl();

  virtual TSCObject_ptr create();
  virtual void destroy(TSCObject_ptr tsc_object);
```

```
};

class EEE_TSCimpl : public EEE_TSCsk
{
private:

public:
  // コンストラクタの引数の数および型を変更することもできます。
  EEE_TSCimpl();
  EEE_TSCimpl(const char* _tpbroker_object_name);

  virtual ~EEE_TSCimpl();

  void call();

};

#endif  // _EEEfile_TSC_T_HDR
```

(2) TSC ユーザオブジェクト（EEE_TSCimpl）と TSC ユーザオブジェクトファクトリ（EEE_TSCfactimpl）のコード

```
//
// "UserExcept_TSC_t.cpp"
//
#include "EEEfile_TSC_t.hh"

EEE_TSCfactimpl::EEE_TSCfactimpl()
{
  // Constructor of factory implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのコンストラクタのコードを記述して
  // ください。
  // コンストラクタの引数の数および型を変更することもできます。
}

EEE_TSCfactimpl::~EEE_TSCfactimpl()
{
  // Destructor of factory implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのデストラクタのコードを記述して
  // ください。
}

TSCObject_ptr
EEE_TSCfactimpl::create()
{
  // Method to create user object.
  // Write user own code.
  // サーバオブジェクトを生成するコードを記述します。
  // 必要に応じて変更してください。
  return new EEE_TSCimpl();
}

void
EEE_TSCfactimpl::destroy(TSCObject_ptr tsc_obj)
{
```

```
  // Method to destroy user object.
  // Write user own code.
  // 後処理のコードを記述します。
  // 必要に応じて変更してください。
  delete tsc_obj;
}

EEE_TSCimpl::EEE_TSCimpl()
{
  // Constructor of implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのコンストラクタのコードを記述して
  // ください。
  // コンストラクタの引数の数および型を変更することもできます。
}

EEE_TSCimpl::EEE_TSCimpl(const char* _tpbroker_object_name)
      : EEE_TSCsk(_tpbroker_object_name)
{
  // Constructor of implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのコンストラクタのコードを記述して
  // ください。
  // コンストラクタの引数の数および型を変更することもできます。
}

EEE_TSCimpl::~EEE_TSCimpl()
{
  // Destructor of implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのデストラクタのコードを記述して
  // ください。
}

void
EEE_TSCimpl::call()
{
  // Operation "::EEE::call".
  // Write user own code.

  throw UserExcept(123456);
}
```

（3） サービス登録処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザオブジェクトファクトリ，**TSC **ユーザアクセプタの生成（**new**），および各種設定**
5. TSC **ルートアクセプタの生成および各種設定**
6. TSC **ルートアクセプタの活性化**

7. 実行制御の受け渡し
8. TSC ルートアクセプタの非活性化
9. TSC ルートアクセプタの削除
10. TSC ユーザオブジェクトファクトリおよび TSC ユーザアクセプタの削除（delete）
11. TSC デーモンへの接続解放
12. TPBroker OTM の終了処理

（4） サービス登録処理のコード

```
//
// "Server.cpp"
//
#include <stdio.h>
#include <iostream.h>

#include <tscadm.h>
#include <tscobject.h>
#include <tscexcept.h>

#include "EEEfile_TSC_t.hh"

#define ERR_FORMAT
    "EC=%d,DC=%d,PC=%d,CS=%d,MC1=%d,MC2=%d,MC3=%d,MC4=%d¥n"

int main(int argc, char** argv)
{

  ////////
  // 1, TPBrokerの初期化処理
  ////////

  CORBA::ORB_ptr orb = 0;
  try
  {
    // ORBの初期化
    orb = CORBA::ORB_init(argc, argv);
  }
  catch(CORBA::SystemException& se)
  {
    // 例外処理
    cerr << se << endl;
    exit(1);
  }

  ////////
  // 2, TPBroker OTMの初期化処理
  ////////

  try
  {
    // TSCの初期化
    TSCAdm::initServer(argc, argv, orb);
  }
```

```
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  exit(1);
}

////////
// 3, TSCデーモンへの接続
////////

TSCDomain_ptr tsc_domain = 0;

try
{
  tsc_domain = new TSCDomain(0, 0);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

TSCServer_ptr tsc_server = 0;

try
{
  // TSCデーモンの参照オブジェクトを取得
  tsc_server = TSCAdm::getTSCServer(tsc_domain);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCAdm::endServer();
```

```
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 4, TSCユーザオブジェクトファクトリ, TSCユーザアクセプタ（new）,
//    および各種設定
////////

// EEE_TSCfactimplの生成
TSCObjectFactory_ptr my_obj_fact = new EEE_TSCfactimpl();

// TSCAcceptorの生成
TSCAcceptor_ptr my_acpt = 0;

try
{
  my_acpt = new EEE_TSCacpt(my_obj_fact);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 5, TSCルートアクセプタの生成および各種設定
////////

// TSCRootAcceptorの生成
TSCRootAcceptor_ptr my_rt_acpt = 0;

try
{
  my_rt_acpt = TSCRootAcceptor::create(tsc_server);
}
catch(TSCSystemException& se)
{
```

```
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

try
{
  // TSCRootAcceptorへの登録
  my_rt_acpt->registerAcceptor(my_acpt);

  // TSCRootAcceptorのパラレルカウント指定もできます。
  // デフォルト値は1
  // オプション引数でデフォルト値を変更することもできます。
  // my_rt_acpt->setParallelCount(5);
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
    exit(1);
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 6, TSCルートアクセプタの活性化
```

```
////////

try
{
  // オブジェクトの活性化
  my_rt_acpt->activate("serviceX");
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 7, 実行制御の受け渡し
////////

try
{
  TSCAdm::serverMainloop();
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    my_rt_acpt->deactivate();
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
```

```
    exit(1);
  }
  exit(1);
}

////////
// 8, TSCルートアクセプタの非活性化
////////

try
{
  // オブジェクトの非活性化
  my_rt_acpt->deactivate();
}
catch(TSCSystemException& se)
{
  // 例外処理
  fprintf(stderr, ERR_FORMAT,
     se.getErrorCode(), se.getDetailCode(),
     se.getPlaceCode(), se.getCompletionStatus(),
     se.getMaintenanceCode1(), se.getMaintenanceCode2(),
     se.getMaintenanceCode3(), se.getMaintenanceCode4());
  try
  {
    TSCRootAcceptor::destroy(my_rt_acpt);
    delete my_acpt;
    delete my_obj_fact;
    TSCAdm::releaseTSCServer(tsc_server);
    TSCAdm::endServer();
  }
  catch(TSCSystemException& se)
  {
    exit(1);
  }
  exit(1);
}

////////
// 9, TSCルートアクセプタの削除
////////

TSCRootAcceptor::destroy(my_rt_acpt);

////////
// 10, TSCユーザオブジェクトファクトリおよびTSCユーザアクセプタの
//      削除（delete）
////////

delete my_acpt;
delete my_obj_fact;

////////
// 11, TSCデーモンへの接続解放
////////

try
{
  TSCAdm::releaseTSCServer(tsc_server);
```

```
    }
    catch(TSCSystemException& se)
    {
      // 例外処理
      fprintf(stderr, ERR_FORMAT,
         se.getErrorCode(), se.getDetailCode(),
         se.getPlaceCode(), se.getCompletionStatus(),
         se.getMaintenanceCode1(), se.getMaintenanceCode2(),
         se.getMaintenanceCode3(), se.getMaintenanceCode4());
      try
      {
        TSCAdm::endServer();
      }
      catch(TSCSystemException& se)
      {
        exit(1);
      }
      exit(1);
    }

    delete tsc_domain;

    ////////
    // 12, TPBroker OTMの終了処理
    ////////

    try
    {
      TSCAdm::endServer();
    }
      catch(TSCSystemException& se)
    {
      // 例外処理
      fprintf(stderr, ERR_FORMAT,
         se.getErrorCode(), se.getDetailCode(),
         se.getPlaceCode(), se.getCompletionStatus(),
         se.getMaintenanceCode1(), se.getMaintenanceCode2(),
         se.getMaintenanceCode3(), se.getMaintenanceCode4());
      exit(1);
    }

    exit(0);

  };
```

2.8　TSCWatchTime を利用するアプリケーションプログラム（C++）

TSCWatchTime を利用するアプリケーションプログラムの C++ での作成例を示します。

ユーザ定義 IDL インタフェースの例，IDL コンパイラが生成するクラス，およびトランザクションフレームジェネレータが生成するクラスは，同期型呼び出しの場合と同様です。「2.2　同期型呼び出しをするアプリケーションプログラム（C++）」を参照してください。

2.8.1　TSCWatchTime を利用するクライアントアプリケーションの例（C++）

同期型呼び出しの場合と同様です。「2.2.1　同期型呼び出しをするクライアントアプリケーションの例（C++）」を参照してください。

2.8.2　TSCWatchTime を利用するサーバアプリケーションの例（C++）

TSCWatchTime を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス ABC_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス ABC_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。さらに，TSCThread の派生クラスと TSCThreadFactory の派生クラスを記述します。

（1）TSC ユーザオブジェクト（ABC_TSCimpl）と TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のヘッダのコード

```
//
// "ABCfile_TSC_t.hh"
//

#ifndef _ABCfile_TSC_T_HDR
#define _ABCfile_TSC_T_HDR

#include <tscobject.h>

#include "ABCfile_TSC_s.hh"

class ABC_TSCfactimpl : public TSCObjectFactory
{
public:
```

```
  // コンストラクタの引数の数および型を変更することもできます。
  ABC_TSCfactimpl();

  virtual ~ABC_TSCfactimpl();

  virtual TSCObject_ptr create();
  virtual void destroy(TSCObject_ptr tsc_object);

};

class ABC_TSCimpl : public ABC_TSCsk
{
private:

public:
  // コンストラクタの引数の数および型を変更することもできます。
  ABC_TSCimpl();

  virtual ~ABC_TSCimpl();

  void call(const OctetSeq& in_data,OctetSeq*& out_data);

  // メソッドが呼ばれた回数
  CORBA::Long m_counter;

};

#endif  // _ABCfile_TSC_T_HDR
```

（2）TSC ユーザオブジェクト（ABC_TSCimpl）と TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のコード

```
//
//  "ABCfile_TSC_t.cpp"
//


#include <tscadm.h>
#include <tscexcept.h>

#include "ABCfile_TSC_t.hh"

ABC_TSCfactimpl::ABC_TSCfactimpl()
{
  // Constructor of factory implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのコンストラクタのコードを記述
  // します。引数の数および型を変更することもできます。
}

ABC_TSCfactimpl::~ABC_TSCfactimpl()
{
  // Destructor of factory implementation.
  // Write user own code.
  // TSCユーザオブジェクトファクトリのデストラクタのコードを記述
  // します。

}
```

```
TSCObject_ptr
ABC_TSCfactimpl::create()
{
  // Method to create user object.
  // Write user own code.
  // サーバオブジェクトを生成するコードを記述します。
  // 必要に応じて変更してください。
    return new ABC_TSCimpl();
}

void
ABC_TSCfactimpl::destroy(TSCObject_ptr tsc_obj)
{
  // Method to destroy user object.
  // Write user own code.
  // ここに後処理のコードを記述します。
  // 必要に応じて変更してください。
  delete tsc_obj;
}

ABC_TSCimpl::ABC_TSCimpl()
{
  // Constructor of implementation.
  // Write user own code.

  ////////
  // 時間監視の処理
  ////////
  TSCWatchTime_ptr watch_time = 0;
  try
  {
    // 監視時間60秒の時間監視オブジェクト生成
    watch_time = new TSCWatchTime(60);

    // 時間監視の開始
    watch_time->start();

    // TSCユーザオブジェクトのコンストラクタのコードを記述します。
    // 引数の数および型を変更することもできます。

    // 時間監視の中断
    watch_time->stop();

    // 時間監視オブジェクトの解放
    delete watch_time;
  }
  catch(TSCSystemException& se)
  {
    // 例外処理
    fprintf(stderr,"EC=%d,DC=%d,PC=%d,MC1=%d,MC2=%d¥n",
              se.getErrorCode(), se.getDetailCode(),
              se.getPlaceCode(), se.getMaintenanceCode1(),
              se.getMaintenanceCode2());

    delete watch_time;

    throw;
```

```
  }
}


ABC_TSCimpl::~ABC_TSCimpl()
{
  // Destructor of implementation.
  // Write user own code.
  // ユーザオブジェクトのデストラクタのコードを記述します。
}

void ABC_TSCimpl::call(const OctetSeq& in_data,
                              OctetSeq*& out_data)
{
  // Operation "::ABC::ABCCall".
  // Write user own code.
  // ユーザメソッドのコードを記述します。

  // メソッドが呼ばれた回数を増加させます。
  // （このメソッドの処理は引数の値と無関係です）
  m_counter++;
  out_data = new OctetSeq();
  out_data->length(0);

}
```

(3) サービス登録処理の流れ・コード

同期型呼び出しの場合と同様です。「2.2.2(3)　サービス登録処理の流れ」,「2.2.2(4)　サービス登録処理のコード」を参照してください。

3 アプリケーションプログラミングインタフェース（C++）

この章では，C++ のクラスライブラリについて説明します。ユーザ定義 IDL インタフェース依存クラスと雛形クラスでは，対応するユーザ定義 IDL インタフェースの名称を "ABC" と仮定します。なお，各クラスの詳細は，アルファベット順に示します。

クラスの一覧（C++）

各クラスは次のように分類されます。

- システム提供クラス
- システム提供例外クラス
- ユーザ定義 IDL インタフェース依存クラス
- 雛形クラス

各クラスの一覧を次の表に示します。

表 3-1　クラス一覧（C++）

分類	クラス
システム提供クラス	• TSCAcceptor • TSCAdm • TSCClient • TSCContext • TSCDomain • TSCObject • TSCObjectFactory • TSCProxyObject • TSCRootAcceptor • TSCServer • TSCSessionProxy • TSCThread • TSCThreadFactory • TSCWatchTime
システム提供例外クラス	• TSCSystemException

分類	クラス	
	• TSCSystemException の派生クラス	• TSCBadContextException • TSCBadInvOrderException • TSCBadOperationException • TSCBadParamException • TSCBadTypecodeException • TSCCommFailureException • TSCDataConversionException • TSCFreeMemException • TSCImpLimitException • TSCInitializeException • TSCInternalException • TSCIntfReposException • TSCInvFlagException • TSCInvIdentException • TSCInvObjrefException • TSCMarshalException • TSCNoImplementException • TSCNoMemoryException • TSCNoPermissionException • TSCNoResourcesException • TSCNoResponseException • TSCObjAdapterException • TSCObjectNotExistException • TSCPersistStoreException • TSCTransientException • TSCUnknownException
ユーザ定義 IDL インタフェース依存クラス（基底クラス）	• ABC_TSCacpt（TSCAcceptor） • ABC_TSCprxy（TSCProxyObject） • ABC_TSCsk（TSCObject） • ABC_TSCspxy（TSCSessionProxy）	
雛形クラス（基底クラス）	• ABC_TSCfactimpl（ABC_TSCfactsk） • ABC_TSCimpl（ABC_TSCsk）	

基本データ型（C++）

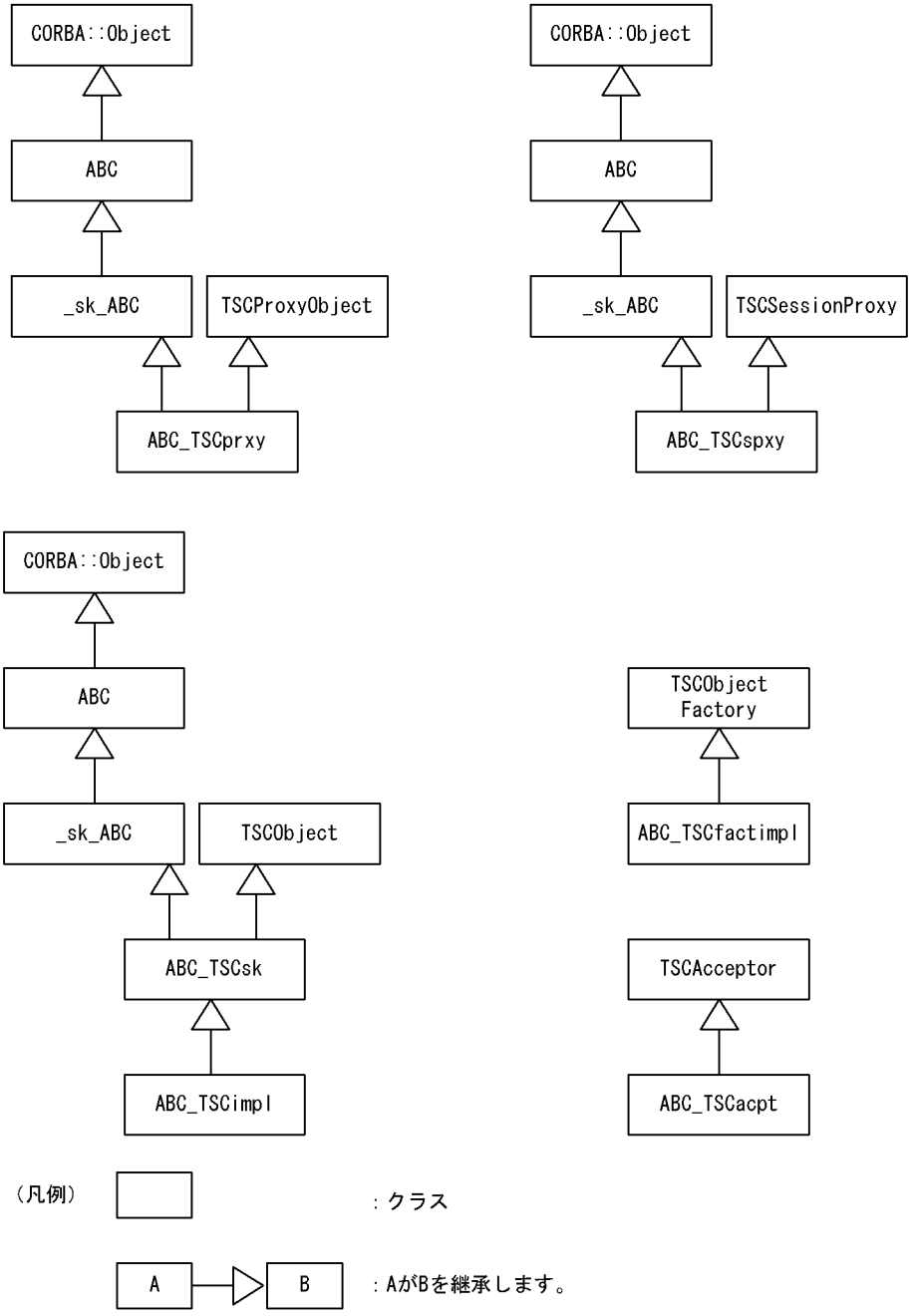
OTM で定義している，各クラスで使用する基本データ型を次に示します。

```
typedef short           TSCShort;
typedef int             TSCInt;
typedef long            TSCLong;
typedef unsigned short  TSCUShort;
typedef unsigned int    TSCUInt;
typedef unsigned long   TSCULong;
typedef char            TSCChar;
typedef char            TSCBool;
typedef unsigned char   TSCUChar;
```

クラス関連図（C++）

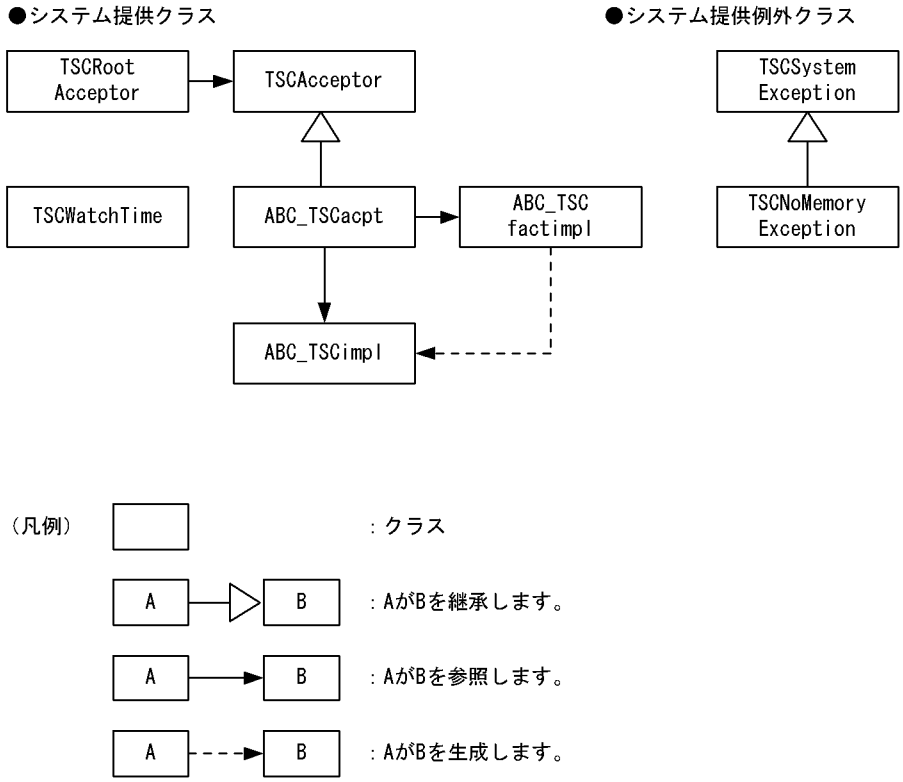
システム提供クラスから派生するユーザ定義 IDL インタフェース依存クラスの関連を次の図に示します。

図 3-1　システム提供クラスから派生するユーザ定義 IDL インタフェース依存クラス（C++）



サーバアプリケーションを作成する際に使用するシステム提供クラスおよびシステム提供例外クラスの関連を次の図に示します。

図 3-2　システム提供クラスおよびシステム提供例外クラス（C++）



公開メソッド呼び出しと内部参照（C++）

システム提供クラスのインスタンス関連図を基に，OTM 上でのインスタンス間の公開メソッド呼び出し，および内部参照（アクセス）について説明します。

● **公開メソッド呼び出し**
OTM のシステム提供クラスのインスタンスが，ほかのインスタンスの公開メソッドを呼び出すことです。

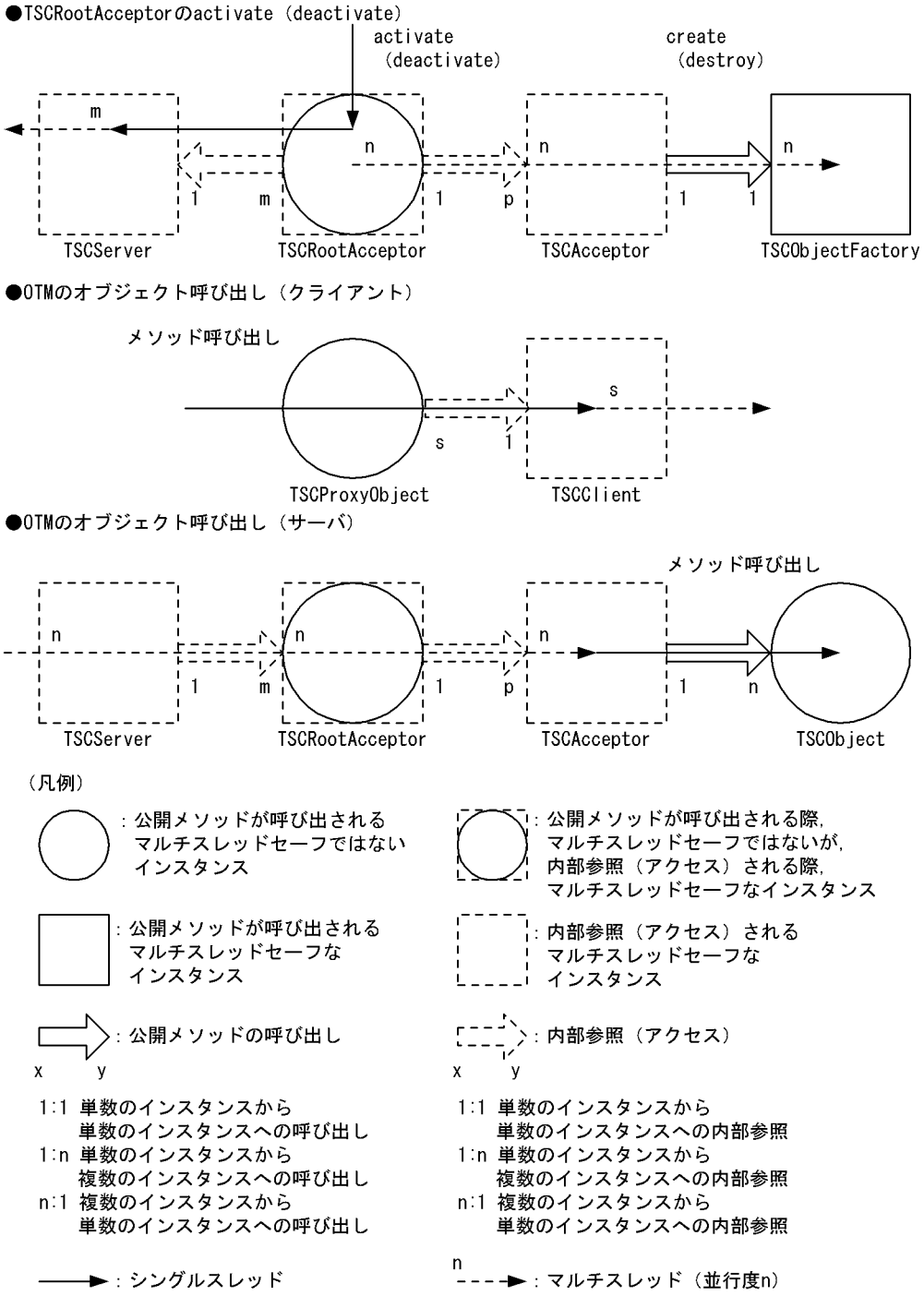
● **内部参照（アクセス）**
OTM のシステム提供クラスのインスタンスが，ほかのインスタンスの非公開メソッドを内部的にアクセスすることです。またはそのインスタンスを内部的に参照することです。

図中のインスタンス数を次の表に示します。また，システム提供クラスのインスタンス関連を図 3-3 に示します。

表 3-2　インスタンス数（C++）

インスタンス	インスタンス数
TSC ルートアクセプタ	m（TSCServer 単位）
パラレルカウント	n（TSCRootAcceptor 単位）
TSC ユーザアクセプタ	p（TSCRootAcceptor 単位）
TSC ユーザオブジェクトファクトリ	1（TSCAcceptor 単位）
TSC ユーザプロキシ	s（TSCClient 単位）

図 3-3　システム提供クラスのインスタンス関連図（C++）



ABC_TSCacpt（C++）

ABC_TSCacpt はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCacpt は，TSC ユーザアクセプタの実装クラスです。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCacpt を自動生成します。次に ABC_TSCacpt の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。

形式

```
class ABC_TSCacpt;
typedef ABC_TSCacpt* ABC_TSCacpt_ptr;

class ABC_TSCacpt : public TSCAcceptor
{
 public:
  ABC_TSCacpt(TSCObjectFactory_ptr tsc_object_fact);
  ABC_TSCacpt(TSCObjectFactory_ptr tsc_object_fact,
              const char* tsc_acpt_name);

  virtual ~ABC_TSCacpt();
};
```

コンストラクタ

● ABC_TSCacpt(TSCObjectFactory_ptr tsc_object_fact)

項目	型・意味	
引数	TSCObjectFactory_ptr tsc_object_fact	TSCObjectFactory オブジェクト
例外	TSCBadParamException TSCNoMemoryException	

TSC ユーザオブジェクトファクトリとして tsc_object_fact を保持します。さらに，デフォルトの TSC アクセプタ名称で ABC_TSCacpt を生成します。

サーバアプリケーションの開始時にコマンドオプション引数 -TSCAcceptor を指定しない場合，TSC アクセプタ名称のデフォルト値は " 指定なし " となります。コマンドオプション引数 -TSCAcceptor を指定する場合は，TSC アクセプタ名称のデフォルト値はその指定値となります。

● ABC_TSCacpt(TSCObjectFactory_ptr tsc_object_fact,
const char* tsc_acpt_name)

項目	型・意味	
引数	TSCObjectFactory_ptr tsc_object_fact	TSCObjectFactory オブジェクト
	const char* tsc_acpt_name	TSC アクセプタ名称
例外	TSCBadParamException TSCNoMemoryException	

TSC ユーザオブジェクトファクトリとして tsc_object_fact を保持します。さらに，デフォルトの TSC アクセプタ名称で ABC_TSCacpt を生成します。ただし，tsc_acpt_name には，1 ～ 31 文字の TSC アクセプタ名称を指定してください。

デストラクタ

● virtual ~ABC_TSCacpt()

ABC_TSCacpt を削除します。

ABC_TSCfactimpl（C++）

ABC_TSCfactimpl は雛形クラスです。

ABC_TSCfactimpl は，TSC ユーザオブジェクトファクトリの実装クラスです。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCfactimpl を生成します。ユーザはこの雛形クラスに処理依存のコードを記述します。次に ABC_TSCfactimpl の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。

形式

*斜体*で示している部分は，ユーザが実装のコードを記述する必要があるメソッドです。**太字**で示している部分は，引数の型および数を変更できるメソッドで，ユーザが実装のコードを記述する必要があります。

```
class ABC_TSCfactimpl : public TSCObjectFactory
{
 public:
  ABC_TSCfactimpl(...);

  virtual ~ABC_TSCfactimpl;

  virtual TSCObject_ptr create();
  virtual void destroy(TSCObject_ptr tsc_object);
};
```

コンストラクタ

● ABC_TSCfactimpl()

引数の型や数を含めて，ユーザがコードを記述する必要があります。複数のコンストラクタを生成できます。

コールバックメソッド

● virtual TSCObject_ptr create()

典型的なコードを生成します。必要に応じて変更してください。

● virtual void destroy(TSCObject_ptr tsc_object)

典型的なコードを生成します。必要に応じて変更してください。

ABC_TSCimpl（C++）

ABC_TSCimpl は雛形クラスです。

ABC_TSCimpl は，TSC ユーザオブジェクトの実装クラスです。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCimpl を生成します。ユーザはこの雛形クラスに処理依存のコードを記述します。次に ABC_TSCimpl の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。

ABC_TSCimpl は，TPBroker のスケルトンである _sk_ABC も継承します。

ユーザ定義 IDL インタフェースのマッピング

ユーザ定義 IDL インタフェース内に定義されたオペレーションの C++ 言語へのマッピングは，TPBroker と同じです。

形式

*斜体*で示している部分は，ユーザが実装のコードを記述する必要があるメソッドです。**太字**で示している部分は，引数の型および数を変更できるメソッドで，ユーザが実装のコードを記述する必要があります。

```
class ABC_TSCimpl : public ABC_TSCsk
{
public:
  ABC_TSCimpl(...);

  virtual ~ABC_TSCimpl();

  //ユーザ定義IDLインタフェース依存のメソッド群
  virtual ... xxx(...);
};
```

コンストラクタ

● ABC_TSCimpl(...)

引数の型や数を含めて，ユーザがコードを記述する必要があります。複数のコンストラクタを生成できます。

コールバックメソッド

● virtual ... xxx(...)

ユーザ定義 IDL インタフェースのオペレーション定義に従ったメソッドです。ユーザがメソッドのコードを実装します。メソッドの引数・戻り値の型や数のマッピングは，TPBroker と同じです。

ABC_TSCprxy（C++）

ABC_TSCprxy はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCprxy は，TSC ユーザプロキシの実装クラスです。ユーザ定義 IDL インタフェースに従って，ユーザデータをバイト配列データに変換し，TSCProxyObject を呼び出します。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCprxy を自動生成します。次に ABC_TSCprxy の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。

ABC_TSCprxy は，TPBroker のスケルトンである _sk_ABC も継承します。また，ABC_TSCprxy の派生クラスも同様に，TPBroker のスケルトンを継承します。

ユーザ定義 IDL インタフェースのマッピング

ユーザ定義 IDL インタフェース内に定義されたオペレーションの C++ 言語へのマッピングは，TPBroker と同じです。

形式

```
class ABC_TSCprxy;
typedef ABC_TSCprxy* ABC_TSCprxy_ptr;

class ABC_TSCprxy : public _sk_ABC, public TSCProxyObject
{
 public:
  ABC_TSCprxy(TSCClient_ptr tsc_client);
  ABC_TSCprxy(TSCClient_ptr tsc_client,
              const char* tsc_acpt_name);

  virtual ~ABC_TSCprxy();

  //ユーザ定義IDLインタフェース依存のメソッド群
  virtual ... xxx(...);
};
```

コンストラクタ

● ABC_TSCprxy(TSCClient_ptr tsc_client)

項目	型・意味	
引数	TSCClient_ptr tsc_client	接続する TSCClient
例外	TSCBadParamException TSCNoMemoryException	

tsc_client と接続する ABC_TSCprxy を生成します。

● ABC_TSCprxy(TSCClient_ptr tsc_client,
const char* tsc_acpt_name)

項目	型・意味	
引数	TSCClient_ptr tsc_client	接続する TSCClient
	const char* tsc_acpt_name	TSC アクセプタ名称
例外	TSCBadParamException TSCNoMemoryException	

tsc_client と接続する TSC アクセプタ名称が tsc_acpt_name の ABC_TSCprxy を生成します。

メソッド

● virtual ... xxx(...);

項目	型・意味
例外	TSCSystemException（各種例外）

ユーザ定義 IDL インタフェースのオペレーション定義に従ったメソッドです。メソッドの引数・戻り値の型や数のマッピングは，TPBroker と同じです。

ABC_TSCsk（C++）

ABC_TSCsk はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCsk は，TSC ユーザスケルトンの実装クラスです。クライアント側から送信されたバイト配列データとともに呼び出されます。ユーザ定義 IDL インタフェースに従って，そのバイト配列データをユーザデータに分解し，TSC ユーザオブジェクトの実装メソッドを呼び出します。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCsk を自動生成します。次に ABC_TSCsk の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。
- 直接，このクラスのインスタンスを生成できません。
- ユーザは ABC_TSCsk クラスを継承して，実装クラスを記述する必要があります。

ABC_TSCsk は，TPBroker のスケルトンである _sk_ABC も継承します。また，ABC_TSCsk の派生クラスも同様に，TPBroker のスケルトンを継承します。

ユーザ定義 IDL インタフェースのマッピング

ユーザ定義 IDL インタフェース内に定義されたオペレーションの C++ 言語へのマッピングは，TPBroker と同じです。

形式

```
class ABC_TSCsk
 : public _sk_ABC, public virtual TSCObject
{
 public:

  virtual ... xxx(...)=0;

 protected:
  ABC_TSCsk();

 virtual ~ABC_TSCsk();

};
```

コンストラクタ

● ABC_TSCsk()

ABC_TSCsk を生成します。

デストラクタ

● virtual ~ABC_TSCsk()

ABC_TSCsk を削除します。

コールバックメソッド

● virtual ... xxx(...)

ユーザ定義 IDL インタフェースのオペレーション定義に従ったメソッドです。メソッドの引数・戻り値の型や数のマッピングは，TPBroker と同じです。

ABC_TSCspxy（C++）

ABC_TSCspxy はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCspxy は，セション呼び出し用の TSC ユーザプロキシの実装クラスです。ユーザ定義 IDL インタフェースに従って，ユーザデータをバイト配列データに変換し，TSCSessionProxy を呼び出します。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCspxy を自動生成します。ABC_TSCspxy は，次の点を除いて ABC_TSCprxy と同様の働きをします。

- TSCSessionProxy を継承します。
- トランザクションフレームジェネレータに -TSCspxy オプションを指定したときだけ生成されます。
- oneway のオペレーションを定義したユーザ定義 IDL からは生成できません。

次に ABC_TSCspxy の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。

ABC_TSCspxy は，TPBroker のスケルトンである _sk_ABC も継承します。また，ABC_TSCspxy の派生クラスも同様に，TPBroker のスケルトンを継承します。

ユーザ定義 IDL インタフェースのマッピング

ユーザ定義 IDL インタフェース内に定義されたオペレーションの C++ 言語へのマッピングは，TPBroker と同じです。

形式

```
class ABC_TSCspxy;
typedef ABC_TSCspxy* ABC_TSCspxy_ptr;

class ABC_TSCspxy : public _sk_ABC, public TSCSessionProxy
{
 public:
  ABC_TSCspxy(TSCClient_ptr tsc_client);
  ABC_TSCspxy(TSCClient_ptr tsc_client,
              const char* tsc_acpt_name);

  virtual ~ABC_TSCspxy();

  //ユーザ定義IDLインタフェース依存のメソッド群
  virtual ... xxx(...);
};
```

コンストラクタ

● ABC_TSCspxy(TSCClient_ptr tsc_client)

項目	型・意味	
引数	TSCClient_ptr tsc_client	接続する TSCClient
例外	TSCBadParamException TSCNoMemoryException	

tsc_client と接続する ABC_TSCspxy を生成します。

● ABC_TSCspxy(TSCClient_ptr tsc_client,
const char* tsc_acpt_name)

項目	型・意味	
引数	TSCClient_ptr tsc_client	接続する TSCClient
	const char* tsc_acpt_name	TSC アクセプタ名称
例外	TSCBadParamException TSCNoMemoryException	

tsc_client と接続する TSC アクセプタ名称が tsc_acpt_name の ABC_TSCspxy を生成します。

メソッド

● virtual ... xxx(...);

項目	型・意味
例外	TSCSystemException（各種例外）

ユーザ定義 IDL インタフェースのオペレーション定義に従ったメソッドです。メソッドの引数・戻り値の型や数のマッピングは，TPBroker と同じです。

TSCAcceptor（C++）

TSCAcceptor はシステム提供クラスです。

TSCAcceptor は，TSC ユーザプロキシを使用したクライアント側からの TSC ユーザオブジェクトのメソッド呼び出し要求に対して，サーバ側の TSC ユーザオブジェクトのメソッドを呼び出すためのクラスです。TSC ルートアクセプタから TSC ユーザオブジェクトのメソッド呼び出し要求を受け取り，TSC ユーザオブジェクトのメソッドを呼び出します。これに伴って，TSC ユーザオブジェクトを管理したり，スレッドと TSC ユーザオブジェクト間を対応づけたりします。また，TSC サービス識別子を使用して，TSC ユーザオブジェクトが提供するサービスを識別します。

ユーザは TSC ユーザオブジェクトの管理オブジェクトとして，TSCAcceptor クラスのインスタンスを生成します。次に TSCAcceptor の特徴を示します。

- TSCObjectFactory を保持することで，TSC ユーザオブジェクト（TSCObject）を管理します。
- TSCAcceptor が提供できるサービスを TSC **サービス識別子**の列で表現します。TSC サービス識別子は，**インタフェース名称**の列と TSC **アクセプタ名称**から構成されます。ただし，TSC アクセプタ名称がない場合もあります。

TSC ユーザオブジェクト（TSCObject）の管理

TSCObjectFactory による TSCObject の管理

TSCRootAcceptor が active 状態に遷移するとき，登録されている TSCAcceptor はオブジェクト管理開始通知を受けます。また，TSCRootAcceptor が non-active 状態に遷移するとき，登録されている TSCAcceptor はオブジェクト管理終了通知を受けます。それぞれの通知とともに，TSCAcceptor は TSCObjectFactory を使用して次に示すように動作します。

- **TSCRootAcceptor からのオブジェクト管理開始通知**
  TSCAcceptor は，TSCRootAcceptor からオブジェクト管理開始通知を受けると，TSCRootAcceptor が保持する各スレッド上で TSCObjectFactory の create を呼び出します。さらに，この呼び出しで返される TSCObject を TSCRootAcceptor が保持する各スレッドに割り当てます。これによって，TSCObjectFactory の create 呼び出しで返される TSCObject は，スレッドに対応づけて管理されます。
- **TSCRootAcceptor からのオブジェクト管理終了通知**
  TSCAcceptor は，TSCRootAcceptor からオブジェクト管理終了通知を受けると，TSCRootAcceptor が保持する各スレッド上で割り当てられている TSCObject を引数に，TSCObjectFactory の destroy を呼び出します。これによって，対応づけられているスレッド上の管理対象から，該当する TSCObject が外されます。

TSCRootAcceptor からの TSCObject の呼び出し

TSCRootAcceptor は，クライアント側からの TSC ユーザオブジェクト呼び出しを受け取ると，該当するサービスを提供する TSCAcceptor に振り分けます。さらに，TSCRootAcceptor が管理するスレッド上で TSC ユーザオブジェクト呼び出し要求を受け取ると，同じスレッド上で同じスレッドに割り当てられている TSCObject を呼び出します。

TSC サービス識別子によるサービスの識別

TSC サービス識別子の構成

TSCAcceptor が提供できるサービスの種類は，TSC サービス識別子によって表されます。TSC サービス識別子は，インタフェース名称の列と TSC アクセプタ名称から構成されます。

- **TSCAcceptor のインタフェース名称の列**
  TSCAcceptor のインタフェース名称の列は，TSCAcceptor や管理する TSCObject が提供するサービスのインタフェースの種類を表します。
  TSCObject が単数のインタフェースをサポートする場合，TSCAcceptor や管理する TSCObject が提供するインタフェースの種類は，単数のインタフェース名称で表されます。TSCObject が複数のインタフェースをサポートする場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，複数のインタフェース名称が列で表されます。
- **TSCAcceptor の TSC アクセプタ名称**
  TSC アクセプタ名称も，TSCAcceptor や管理する TSCObject が提供するサービスのインタフェースの種類を表します。ただし，同じインタフェースを提供する TSCAcceptor または TSCObject の間の実装内容の違いを識別するために使用します。したがって，TSC アクセプタ名称を設定しないで，"TSC アクセプタ名称なし " とすることもできます。

サービスの種類の表現方法

TSCAcceptor が提供できるサービスの種類は，TSC サービス識別子の列によって表されます。TSCAcceptor は TSC サービス識別子の列で示されるサービスを提供できます。

- **TSCAcceptor に TSC アクセプタ名称が設定されていない場合**
  TSCObject が単数のインタフェースをサポートする場合，TSCAcceptor や管理する TSCObject が提供できるサービスの種類は，次のように表されます。

  単数のTSCアクセプタ名称なしTSCサービス識別子

  TSCObject が複数のインタフェースをサポートする場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，TSC サービス識別子の列で次のように表されます。

複数のTSCアクセプタ名称なしTSCサービス識別子

- **TSCAcceptor に TSC アクセプタ名称が設定されている場合**
  TSCObject が単数のインタフェースをサポートする場合，TSCAcceptor や管理する TSCObject が提供できるサービスの種類は，次のように表されます。

  単数のTSCアクセプタ名称なしTSCサービス識別子，および
  単数のTSCアクセプタ名称ありTSCサービス識別子

  TSCObject が複数のインタフェースをサポートする場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，TSC サービス識別子の列で次のように表されます。

  複数のTSCアクセプタ名称なしTSCサービス識別子，および
  複数のTSCアクセプタ名称ありTSCサービス識別子（ただし，TSCサービス識別子中の TSCアクセプタ名称は同じ）

サービスの種類の表現例

- **TSCAcceptor のインタフェース名称が "ABC" で，TSC アクセプタ名称がない場合**
  次のように表される場合，TSCAcceptor は，"ABC::" への要求だけを受け付けることができます。

  ```
  "ABC::"
  ```

- **TSCAcceptor のインタフェース名称が "ABC" で，TSC アクセプタ名称が "abc" の場合**
  次のように表される場合，TSCAcceptor は，"ABC::" と "ABC::abc" への要求メッセージを受け付けることができます。

  ```
  "ABC::abc"
  "ABC::"
  ```

形式

```
class TSCAcceptor;
typedef TSCAcceptor_ptr *TSCAcceptor;

class TSCAcceptor
{
 public:
  //インタフェース名称列
  const char* const* getInterfaceName();

  //TSCアクセプタ名称
  const char* getAcceptorName();
};
```

インクルードファイル

```
#include <tscobject.h>
```

メソッド

● const char* const* getInterfaceName()

項目	型・意味
戻り値	インタフェース名称の列
例外	ありません。

インタフェース名称の列を取得します。

TSC アクセプタ名称のメモリ領域の管理責任は TSCAcceptor クラスにあるので，ユーザは削除しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● const char* getAcceptorName()

項目	型・意味
戻り値	TSC アクセプタ名称
例外	ありません。

TSC アクセプタ名称を取得します。

TSC アクセプタ名称のメモリ領域の管理責任は TSCAcceptor クラスにあるので，ユーザは削除しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

TSCAcceptor の生成と削除

TSCAcceptor を生成するには，引数を指定しないで，または TSC アクセプタ名称を指定して，new オペレータで生成します。

TSCAcceptor を削除するときは，delete オペレータを使用します。TSCAcceptor クラスのインスタンスへの内部参照（アクセス）があるときは削除できないため，インスタンスへの内部参照（アクセス）をなくした状態で削除してください。

TSCAcceptor の派生クラス型のインスタンスについても同様です。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCAcceptor クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
getInterfaceName	できます。
getAcceptorName	できます。

インスタンスの公開メソッド呼び出し規則

TSCAcceptor クラスのインスタンスが，ほかのクラスのインスタンスの公開メソッドを呼び出す規則を次に示します。

タイミング	公開メソッド呼び出し
TSCRootAcceptor からのオブジェクト管理開始通知，またはオブジェクト管理終了通知のとき	コンストラクタで指定した TSCObjectFactory 型のインスタンス
登録先の TSCRootAcceptor が active 状態のとき	active 状態に遷移するときに管理を開始した TSCObject 型のインスタンス

インスタンスへの内部参照（アクセス）規則

TSCAcceptor クラス型のインスタンスを解放したあと，このインスタンスを内部参照するインスタンスからのアクセスは，メモリアクセス違反となります。OTM は，その際の動作を保証しません。

また，複数のスレッド上から同時に TSCAcceptor クラスの同じインスタンスを内部参照（アクセス）できます。

TSCAdm（C++）

TSCAdm はシステム提供クラスです。

TSCAdm は，アプリケーションプログラムの初期化処理，TSCClient の取得，および TSCServer の取得をするクラスです。

形式

```
class TSCAdm
{
 public:
  static void initServer(TSCInt argc,
                         TSCChar *argv[],
                         CORBA::ORB_ptr orb);
  static void initClient(TSCInt argc,
                         TSCChar *argv[],
                         CORBA::ORB_ptr orb);

  static void serverMainloop();
  static void shutdown();

  static void endServer();
  static void endClient();

  static TSCClient_ptr getTSCClient(TSCDomain_ptr tsc_domain,
                                    TSCInt way);
  static TSCClient_ptr getTSCClient(TSCDomain_ptr tsc_domain);

  static TSCServer_ptr getTSCServer(TSCDomain_ptr tsc_domain);

  static void releaseTSCClient(TSCClient_ptr tsc_client);
  static void releaseTSCServer(TSCServer_ptr tsc_server);

  static TSCInt get_status();
};
```

インクルードファイル

```
#include <tscadm.h>
```

メソッド

● static void initServer(TSCInt argc,
TSCChar *argv[],
CORBA::ORB_ptr orb)

項目	型・意味	
引数	TSCInt argc	コマンド引数の数
	TSCChar *argv[]	コマンド引数
	CORBA::ORB_ptr orb	ORB のリファレンス

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException

サーバアプリケーションの初期化処理を実行します。このメソッドは，プロセスで 1 回だけ発行できます。TSCAdm::endServer() メソッド，または TSCAdm::endClient() メソッドの発行によって終了処理したあとでも，このメソッドは発行できません。

このメソッドの argc 引数にはプロセスの main() 関数の第 1 引数を，argv 引数には main() 関数の第 2 引数をそれぞれそのまま指定してください。プロセス開始時にコマンドラインで指定された情報を削除または変更して argc 引数および argv 引数に指定すると，正しく動作しない場合があります。tscstartprc コマンドを使用して開始したサーバアプリケーションのコマンドラインには，tscstartprc コマンドに指定したコマンドライン引数がすべて渡されます。

● static void initClient(TSCInt argc,
TSCChar *argv[],
CORBA::ORB_ptr orb)

項目	型・意味	
引数	TSCInt argc	コマンド引数の数
	TSCChar *argv[]	コマンド引数
	CORBA::ORB_ptr orb	ORB のリファレンス
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException	

クライアントアプリケーションの初期化処理を実行します。TSCAdm::initServer() メソッドを発行すれば，TSCAdm::initClient() メソッドを発行しなくてもクライアントアプリケーションの機能を使用できるようになります。

このメソッドの argc 引数にはプロセスの main() 関数の第 1 引数を，argv 引数には

main() 関数の第 2 引数をそれぞれそのまま指定してください。プロセス開始時にコマンドラインで指定された情報を削除または変更して argc 引数および argv 引数に指定すると，正しく動作しないことがあります。

このメソッドは，TSCAdm::endClient() メソッドの発行によって終了処理をした場合は再発行できますが，TSCAdm::endServer() メソッドの発行によって終了処理をしたあとは再発行できません。TSCAdm::endClient() メソッドの発行後にこのメソッドを再発行した場合，2 回目以降の発行で指定した argc 引数および argv 引数の値は無効となり，初回の発行で指定した値が有効になります。ただし，このメソッドを複数回発行するとクライアントアプリケーションの性能に影響を与えるため，推奨できません。

● static void serverMainloop()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCCommFailureException

リクエストを受信待ち状態にします。このメソッドはサーバアプリケーションでだけ発行できます。

● static void shutdown()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException

リクエストの受信待ち状態を解除します。このメソッドはサーバアプリケーションでだけ発行できます。

● static void endServer()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCInternalException TSCNoMemoryException

サーバアプリケーションの終了処理を実行します。

このメソッドはプロセスで 1 回だけ発行できます。このメソッド発行後はそのプロセスで OTM の機能は使用しないでください。

● static void endClient()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCInternalException TSCNoMemoryException

クライアントアプリケーションの終了処理を実行します。

このメソッドの発行後は，そのプロセスで OTM または OTM - Client の機能を使用できません。また，このメソッドで例外が発生した場合は，クライアントアプリケーションを終了させる必要があります。

● static TSCClient_ptr getTSCClient(TSCDomain_ptr tsc_domain,
TSCInt way)

項目	型・意味	
引数	TSCDomain_ptr tsc_domain	TSCDomain のリファレンス
	TSCInt way	接続経路
戻り値	TSCClient オブジェクトリファレンス	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException TSCTransientException	

指定した TSCDomain を基に，リクエストする TSC デーモンの TSCClient のリファレンスを取得するメソッドです。

way には，接続経路として，次に示すどちらかを指定します。

- TSCAdm::TSC_ADM_DIRECT
  TSC デーモンに直結してリクエストします。ただし，シングルスレッドライブラリを使用するアプリケーションプログラムの場合は，TSC デーモンに直結したリクエストはできません。
- TSCAdm::TSC_ADM_REGULATOR
  TSC レギュレータを経由してリクエストします。この場合，TSC レギュレータによってコネクションを集約します。

ファイル検索方式でマルチノードリトライ接続を実行する場合，このメソッドに指定した TSCDomain および way の組み合わせに一致する情報が，接続先情報ファイル中に記述されていなければなりません。一致する情報が接続先情報ファイルにない場合，

TSCBadParamException 例外が発生します。なお，ファイル検索方式でマルチノードリトライ接続を実行するには，アプリケーションプログラムの開始時に，次に示すようにコマンドオプション引数を指定します。

- -TSCRetryReference に接続先情報ファイルを指定し，かつ，-TSCRetryWay に "0000" または "0001" を指定します。
- -TSCRetryReference に接続先情報ファイルを指定して，-TSCRetryWay の指定を省略します。この場合，-TSCRetryWay には，"0000" が仮定されます。

● static TSCClient_ptr getTSCClient(TSCDomain_ptr tsc_domain)

項目	型・意味	
引数	TSCDomain_ptr tsc_domain	TSCDomain のリファレンス
戻り値	TSCClient オブジェクトリファレンス	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException TSCTransientException	

指定した TSCDomain を基に，リクエストする TSC デーモンの TSCClient のリファレンスを取得するメソッドです。接続経路には，アプリケーションプログラムの開始時に，コマンドオプション引数 -TSCRequestWay に指定した値を使用します。なお，シングルスレッドライブラリを使用するアプリケーションプログラムの場合，TSC デーモンに直結したリクエストはできません。

ファイル検索方式でマルチノードリトライ接続を実行する場合，このメソッドに指定した TSCDomain，およびコマンドオプション引数 -TSCRequestWay の組み合わせに一致する情報が，接続先情報ファイル中に記述されていなければなりません。一致する情報が接続先情報ファイルにない場合，TSCBadParamException 例外が発生します。なお，ファイル検索方式でマルチノードリトライ接続を実行するには，アプリケーションプログラムの開始時に，次に示すようにコマンドオプション引数を指定します。

- -TSCRetryReference に接続先情報ファイルを指定し，かつ，-TSCRetryWay に "0000" または "0001" を指定します。
- -TSCRetryReference に接続先情報ファイルを指定して，-TSCRetryWay の指定を省略します。この場合，-TSCRetryWay には，"0000" が仮定されます。

● static TSCServer_ptr getTSCServer(TSCDomain_ptr tsc_domain)

項目	型・意味	
引数	TSCDomain_ptr tsc_domain	TSCDomain のリファレンス
戻り値	TSCServer オブジェクトリファレンス	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException	

指定した TSCDomain を基に，自サーバへリクエストを振り分ける TSC デーモンの TSCServer のリファレンスを取得するメソッドです。

● static void releaseTSCClient(TSCClient_ptr tsc_client)

項目	型・意味	
引数	TSCClient_ptr tsc_client	TSCClient オブジェクトリファレンス
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException	

TSCClient を解放します。

● static void releaseTSCServer(TSCServer_ptr tsc_server)

項目	型・意味	
引数	TSCServer_ptr tsc_server	TSCServer オブジェクトリファレンス
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException	

TSCServer を解放します。

● static TSCInt get_status()

項目	型・意味
戻り値	プロセスステータス

運用管理で管理するプロセスステータスを返します。プロセスステータスを示す定数を表 3-3，表 3-4 に示します。

表 3-3　TSCAdm クラスで検出するクライアントアプリケーションのプロセスステータス（C++）

定数名	状態	内容
TSCAdm::TSC_ADM_PRC_LIVING	オンライン稼働中	initClient() を発行してから，終了要求を受け付けるまでの状態
TSCAdm::TSC_ADM_PRC_DYING	正常終了処理中	終了要求を受け付けてから，endClient() を発行するまでの状態
TSCAdm::TSC_ADM_PRC_DEAD	終了	initClient() の発行以前，または endClient() の発行以降の状態

クライアントアプリケーションの状態遷移については，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のクライアントアプリケーションの状態の検出に関する説明を参照してください。

表 3-4　TSCAdm クラスで検出するサーバアプリケーションのプロセスステータス（C++）

定数名	状態	内容
TSCAdm::TSC_ADM_PRC_LIVING	正常開始処理中	initServer() を発行してから，serverMainloop() を発行するまでの状態
TSCAdm::TSC_ADM_PRC_ACTIVE	オンライン稼働中	serverMainloop() を発行してから，終了要求を受け付けるまでの状態
TSCAdm::TSC_ADM_PRC_DYING	正常終了処理中	終了要求を受け付けてから，endServer() を発行するまでの状態
TSCAdm::TSC_ADM_PRC_DEAD	終了	initServer() の発行以前，または endServer() の発行以降の状態

サーバアプリケーションの状態遷移については，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のサーバアプリケーションの状態の検出に関する説明を参照してください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCAdm クラスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
initServer	できます。※
initClient	できます。※
serverMainloop	できます。※
shutdown	できます。※
endServer	できます。※
endClient	できます。※
getTSCClient	できます。
getTSCServer	できます。
releaseTSCClient	できます。
releaseTSCServer	できます。
get_status	できます。

注※

複数のスレッド上から同時に呼び出すことはできますが，有効となるのは一つの呼び出しだけです。

TSCClient（C++）

TSCClient はシステム提供クラスです。

TSCClient は，TSC デーモン中のクライアントアプリケーション管理部分を表すクラスです。クライアントアプリケーション側からの TSC ユーザオブジェクトの呼び出し要求は，TSCClient を経由して TSC デーモンに渡されます。

ユーザは，クライアントアプリケーションが TSC デーモンと接続するときに TSCClient を取得します。クライアントアプリケーションと TSC デーモンの接続には，TSC デーモンと直結する方法と，TSC レギュレータを経由する方法があります。次に TSCClient の特徴を示します。

- 属性として TSC ドメイン名称と TSC 識別子を持ちます。

TSC デーモンに直結する場合の TSCClient の取得

クライアントアプリケーションと TSC デーモン間の直結の接続は，クライアントアプリケーションプロセス内で TSCClient を最初に取得するときに確立されます。その後，同じ TSC デーモンに対して TSCClient を取得する場合は，その接続を共有します。逆に，取得したすべての TSCClient を解放すると接続が切断されます。

一つのクライアントアプリケーションから複数の TSC デーモンへ接続を確立することもできます。また，TSC ユーザオブジェクト呼び出し要求が，この接続を経由して TSC デーモンに渡される場合，並行して処理されます。

ただし，シングルスレッドライブラリを使用するアプリケーションプログラムの場合，TSC デーモンに直結してリクエストできません。

TSC レギュレータを経由する場合の TSCClient の取得

TSC レギュレータを経由する場合のクライアントアプリケーションと TSC デーモン間の接続は，TSCClient を取得するたびに確立されます。逆に，TSCClient を解放するたびに，割り当てられた接続が切断されます。

一つのクライアントアプリケーションから複数の TSC デーモンへ接続を確立することもできます。また，TSC ユーザオブジェクト呼び出し要求がこの一つの接続を経由して TSC デーモンに渡される場合，並行して処理されないで順番に処理されます。

形式

```
class TSCClient;
typedef TSCClient* TSCClient_ptr;

class TSCClient
{
 public:
```

```
  const char* getTSCDomainName();
  const char* getTSCID();
};
```

インクルードファイル

```
#include <tscproxy.h>
```

メソッド

● const char* getTSCDomainName()

項目	型・意味
戻り値	TSC ドメイン名称
例外	ありません。

TSC ドメイン名称を返します。

TSC ドメイン名称のメモリ領域の管理責任は TSCClient クラスにあるので，ユーザは削除しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● const char* getTSCID()

項目	型・意味
戻り値	TSC 識別子
例外	ありません。

TSC 識別子を返します。

TSC ドメイン名称のメモリ領域の管理責任は TSCClient クラスにあるので，ユーザは削除しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

TSCClient の取得と解放

TSC デーモンと直結する場合，TSCClient を取得するには，TSCAdm::TSC_ADM_DIRECT を引数に指定して TSCAdm::getTSCClient を発行します。TSC レギュレータを経由する場合は，TSCAdm::TSC_ADM_REGULATOR を引数に指定して TSCAdm::getTSCClient を発行します。

TSCClient を解放するときは，TSCAdm::releaseTSCClient を発行します。TSCClient クラスのインスタンスへの内部参照（アクセス）があるときは解放できないため，インスタンスへの内部参照（アクセス）をなくした状態で解放してください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCClient クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
getTSCDomainName	できます。
getTSCID	できます。

インスタンスへの内部参照（アクセス）規則

TSCClient クラスのインスタンスを解放したあと，このインスタンスを内部参照するインスタンスからのアクセスは，メモリアクセス違反となります。OTM は，その際の動作を保証しません。

また，複数のスレッド上から同時にこのクラスの同じインスタンスを内部参照できます。

TSCContext（C++）

TSCContext はシステム提供クラスです。

TSCContext は，TSC ユーザプロキシを使用してクライアント側からサーバ側の TSC ユーザオブジェクトのメソッドを呼び出すとき，暗黙的にサーバ側に渡すユーザデータのコンテナクラスです。次に TSCContext の特徴を示します。

- ユーザデータを保持します。

TSCContext によるユーザデータの取得

TSCProxyObject を使用してクライアント側からサーバ側の TSC ユーザオブジェクトのメソッドを呼び出すときに，TSCContext によって引数以外のユーザデータをサーバ側に渡すことができます。

クライアント側では，TSCProxyObject の _TSCContext によって TSCContext を取得し，送信したいユーザデータを設定します。サーバ側では，オブジェクトが呼び出されている間，TSCObject の _TSCContext を使用して TSCContext を取得します。この TSCContext が保持しているユーザデータは，クライアント側の TSCContext に設定したユーザデータと同じ内容です。

形式

```
class TSCContext
{
 public:
  void setUserData(TSCUChar* user_data,
                   TSCInt user_data_length,
                   TSCBool release);
  TSCUChar* getUserData();
  TSCInt getUserDataLength();
  TSCBool getUserDataRelease();
};
```

インクルードファイル

```
#include <tsccontext.h>
```

メソッド

● void setUserData(TSCUChar* user_data,
TSCInt user_data_length,
TSCBool release)

項目	型・意味	
引数	TSCUChar* user_data	ユーザデータ
	TSCInt user_data_length	ユーザデータサイズ
	TSCBool release	リリースフラグ
戻り値	ありません。	

ユーザデータを設定します。

リリースフラグ（release）が TSC_FALSE の場合，user_data のメモリ領域の管理責任はユーザにあります。TSC_TRUE の場合は，user_data のメモリ管理責任は TSCContext クラスにあるので，ユーザは削除しないでください。また，TSC_TRUE の場合は，TSCContext クラスは user_data を delete オペレータで削除します。そのため，user_data を new オペレータで生成してください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● TSCUChar* getUserData()

項目	型・意味
戻り値	ユーザデータ

ユーザデータを取得します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● TSCInt getUserDataLength()

項目	型・意味
戻り値	ユーザデータのサイズ

ユーザデータのサイズを取得します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● TSCBool getUserDataRelease()

項目	型・意味
戻り値	リリースフラグ

ユーザデータのリリースフラグを取得します。

TSCContext の生成と削除

TSCContext のインスタンスは，new オペレータで生成しないでください。また，delete オペレータで削除しないでください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCContext クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
setUserData	できません。
getUserData	できません。
getUserDataLength	できません。

TSCDomain（C++）

TSCDomain はシステム提供クラスです。

TSCDomain は，文字列の TSC ドメイン情報，および TSC 識別子を管理するホルダクラスです。

形式

```
class TSCDomain;
typedef TSCDomain* TSCDomain_ptr;

class TSCDomain
{
 public:
  TSCDomain(TSCChar *domain_name);
  TSCDomain(TSCChar *domain_name, TSCChar *tscid);
};
```

インクルードファイル

```
#include <tscadm.h>
```

コンストラクタ

● TSCDomain(TSCChar *domain_name)

項目	型・意味	
引数	TSCChar *domain_name	TSC ドメイン名称
例外	TSCBadParamException TSCInitializeException	

このコンストラクタを使用する場合，TSC ドメイン名称で TSC ドメインを管理します。

domain_name に文字列を指定する場合は，先頭が "TSC" または "tsc" ではない 1 〜 31 文字の英数字の文字列を指定してください。domain_name に NULL を指定する場合は，アプリケーションプログラムの開始時に指定するコマンドオプション引数 -TSCRetryReference の指定の有無によって管理する情報が異なります。

- コマンドオプション引数 -TSCRetryReference を指定しない場合
  initServer または initClient の argv 内の "-TSCDomain" オプションの指定値を使用します。
- コマンドオプション引数 -TSCRetryReference を指定する場合
  コマンドオプション引数 -TSCRetryWay の指定内容によって動作が異なります。
  -TSCRetryWay の指定値と接続方式の関係については，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のマルチノードリトライ接続の接続対象に関する説明を参照してください。

● TSCDomain(TSCChar *domain_name,
TSCChar *tscid)

項目	型・意味	
引数	TSCChar *domain_name	TSC ドメイン名称
	TSCChar *tscid	TSC 識別子
例外	TSCBadParamException TSCInitializeException	

このコンストラクタを使用する場合，TSC ドメイン名称と TSC 識別子で TSC ドメインを管理します。

domain_name または tscid に文字列を指定する場合は，先頭が "TSC" または "tsc" ではない 1 ～ 31 文字の英数字の文字列を指定してください。なお，tscid に IP アドレスを指定する場合は，ピリオド (.) も使用できます。domain_name または tscid に NULL を指定する場合は，アプリケーションプログラムの開始時に指定するコマンドオプション引数 -TSCRetryReference の指定の有無によって管理する情報が異なります。

- コマンドオプション引数 -TSCRetryReference を指定しない場合
  initServer または initClient の argv 内の "-TSCDomain" オプションおよび "-TSCID" オプションの指定値を使用します。
- コマンドオプション引数 -TSCRetryReference を指定する場合
  コマンドオプション引数 -TSCRetryWay の指定内容によって動作が異なります。
  -TSCRetryWay の指定値と接続方式の関係については，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のマルチノードリトライ接続の接続対象に関する説明を参照してください。

TSCObject（C++）

TSCObject はシステム提供クラスです。

TSCObject は，OTM での（サーバ）オブジェクトの基本クラスおよびインタフェースです。

ユーザは TSCObject を継承させて，クライアント側にサービスを提供するクラスを定義します。また，サービスを提供するオブジェクトとして，その派生クラスのインスタンスを生成します。次に TSCObject の特徴を示します。

- 直接，TSCObject のインスタンスを生成できません。
- TSCObject を生成する，TSCObjectFactory の実装クラスを記述する必要があります。
- 属性としてインタフェース名称の列を持ちます。
- クライアント側から TSCProxyObject を使用して TSCObject を呼び出すときにユーザデータを TSCContext に指定すると，サーバ側で TSCContext から同じデータを取得できます。
- TSCRootAcceptor を生成するときに TSCThreadFactory を指定すると，TSCThread を取得できます。

TSCObject が提供するサービスのインタフェース名称

TSCObject が提供するインタフェースの種類は，TSCAcceptor のインタフェース名称の列で表されます。

TSCObject が単数のインタフェースを提供する場合，提供するインタフェースの種類は，単数のインタフェース名称で表されます。TSCObject が複数のインタフェースを提供する場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，複数のインタフェース名称が列で表されます。

TSCObject の呼び出し時の TSCContext の取得

TSCProxyObject を使用してクライアント側から TSCObject のメソッドを呼び出すときに，ユーザは引数以外のデータを TSCContext として送信できます。クライアント側で引数以外のデータを TSCContext として送信すると，サーバ側では TSCObject のサービス提供メソッドが呼び出されている間に _TSCContext を呼び出すことによって，クライアント側で指定した TSCContext を取得できます。

TSCObject の呼び出し時の TSCThread の取得

TSCObject の呼び出し時の TSCThread の取得は，TSCThreadFactory を引数として TSCRootAcceptor を生成する場合を前提にします。この場合，サーバ側で TSCObject のサービス提供メソッドが呼び出されている間に _TSCThread を呼び出すことによって，実行制御を持つスレッドに割り付けられている TSCThread を取得できます。

形式

```
class TSCObject
typedef TSCObject* TSCObject_ptr

class TSCObject
{
 public:
  //インタフェース名称
  const char* const* _TSCInterfaceName();

  //TSCコンテキスト
  TSCContext_ptr _TSCContext();

  //TSCユーザスレッド
  TSCThread_ptr _TSCThread();
};
```

インクルードファイル

```
#include <tscobject.h>
```

メソッド

● const char* const* _TSCInterfaceName()

項目	型・意味
戻り値	インタフェース名称列

インタフェース名称の列を取得します。

インタフェース名称のメモリ領域の管理責任は TSCObject クラスにあるので，ユーザは削除しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● TSCContext_ptr _TSCContext()

項目	型・意味
戻り値	呼び出し元で指定された TSCContext

TSCContext を取得します。クライアント側から呼び出すときに TSCContext に指定したユーザデータを取得できます。

戻り値の TSCContext のメモリ領域の管理責任は TSCObject クラスにあるので，ユーザは削除しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● TSCThread_ptr _TSCThread()

項目	型・意味
戻り値	TSC ユーザスレッド

このオブジェクトに割り当てられているスレッドに対応する TSC ユーザスレッドを返します。

戻り値の TSCThread のメモリ領域の管理責任は TSCObject クラスにあるので，ユーザは削除しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

TSCObject の派生クラスの生成と削除

TSCObject の派生クラスは，new オペレータで生成し，delete オペレータで削除します。OTM が TSCObject の公開メソッドを呼び出しているときは削除できないため，公開メソッドを呼び出していない状態で削除してください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCObject クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
_TSCInterfaceName	できません。
_TSCContext	できません。
_TSCThread	できません。
クライアント側からのオブジェクト呼び出し※	できません。

注※

このメソッドは OTM が呼び出します。

TSCObjectFactory（C++）

TSCObjectFactory はシステム提供クラスです。

TSCObjectFactory は，OTM が TSC ユーザオブジェクトの管理を開始するとき，または管理対象から外すときに呼び出されるオブジェクトのインタフェースです。

ユーザは TSCObjectFactory を継承させて，TSC ユーザオブジェクトを生成，または削除するクラスを定義します。また，TSC ユーザオブジェクトのファクトリとして，派生クラスのインスタンスを生成します。次に TSCObjectFactory の特徴を示します。

- 直接，TSCObjectFactory のインスタンスを生成できません。
- ユーザは TSCObjectFactory クラスを継承して，実装クラスを記述する必要があります。

OTM からの呼び出しによる TSCObject の管理

OTM は，TSCObjectFactory の create を呼び出すことで，返される TSCObject の管理を開始します。逆に，該当する TSCObject を引数に TSCObjectFactory の destroy を呼び出すことで，TSCObject を管理対象から外します。

形式

```
class TSCObjectFactory;
typedef TSCObjectFactory* TSCObjectFactory_ptr;

class TSCObjectFactory
{
 public:
  TSCObjectFactory();
  virtual ~TSCObjectFactory();

  virtual TSCObject_ptr create() = 0;
  virtual void destroy(TSCObject_ptr tsc_object) = 0;
};
```

インクルードファイル

```
#include <tscobject.h>
```

コンストラクタ

● TSCObjectFactory()

TSCObjectFactory を生成します。

デストラクタ

● virtual ~TSCObjectFactory()

TSCObjectFactory を削除します。

コールバックメソッド

● virtual TSCObject_ptr create() = 0

項目	型・意味
戻り値	管理対象の TSC ユーザオブジェクト
例外	各種 TSCSystemException

TSCObject を返します。TSC ユーザオブジェクトを生成するコードを記述できます。OTM がこのメソッドを呼び出した結果，返された TSCObject が管理対象となります。

管理対象とする TSCObject のメモリ管理責任は OTM にあるので，ユーザは削除しないでください。なお，OTM は，複数のスレッド上から同時にこのメソッドを呼び出すので，ユーザはマルチスレッド環境に対応するリエントラントなコードを記述する必要があります。

また，create 呼び出しに失敗した場合は，各種 TSCSystemException によって通知する形でコードを記述してください。

● virtual void destroy(TSCObject_ptr tsc_object) = 0

項目	型・意味	
引数	TSCObject_ptr tsc_object	管理対象から外す TSC ユーザオブジェクト
例外	ありません。	

TSCObject を消去する前の処理のコードを記述できます。OTM が TSC ユーザオブジェクトを管理対象から外すとき，該当する TSCObject を引数に指定して，このメソッドを呼び出します。

管理対象から外された TSCObject のメモリ管理責任はユーザにあります。なお，OTM は，複数のスレッド上から同時にこのメソッドを呼び出すので，ユーザはマルチスレッド環境に対応するリエントラントなコードを記述する必要があります。

また，このメソッドから通知した例外は無視されます。

TSCObjectFactory の派生クラスの生成と削除

TSCObjectFactory の派生クラスは，new オペレータで生成し，delete オペレータで削除します。OTM が TSCObject の公開メソッドを呼び出しているとき，またはユーザが TSCObject の公開メソッドを呼び出しているときは削除できないため，公開メソッドを

呼び出していない状態で削除してください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCObjectFactory クラスのインスタンスのメソッドを呼び出す規則を次に示します。なお，これらのメソッドは，OTM が呼び出します。

メソッド	複数のスレッド上からの同時呼び出し
create	できます。
destroy	できます。

TSCProxyObject（C++）

TSCProxyObject はシステム提供クラスです。

TSCProxyObject は TSCObject の代理クラスです。TSCProxyObject を呼び出すと，OTM のスケジューリング機構を経由して TSCObject が呼び出されます。

ユーザは，サーバアプリケーションの TSC ユーザオブジェクトが提供するサービスを利用するときに，TSCProxyObject クラスのインスタンスを生成して呼び出します。次に TSCProxyObject の特徴を示します。

- TSCProxyObject を使用して利用できるサービスを TSC **サービス識別子**で表現します。TSC サービス識別子は，**インタフェース名称**と TSC **アクセプタ名称**から構成されます。ただし，TSC アクセプタ名称がない場合もあります。
- 属性として，タイムアウト値（呼び出し時の監視時間）とプライオリティ値（メソッド呼び出し時の優先順位）を持ちます。
- TSCContext を登録できます。

TSC サービス識別子によるサービスの識別

TSC サービス識別子の構成

TSCProxyObject を使用して利用するサービスの種類は，TSC サービス識別子によって表されます。TSC サービス識別子は，インタフェース名称と TSC アクセプタ名称から構成されます。

- **TSCProxyObject のインタフェース名称**
  TSCProxyObject のインタフェース名称は，TSCProxyObject を使用して呼び出すサービスのインタフェースの種類を表します。
- **TSCProxyObject の TSC アクセプタ名称**
  TSCProxyObject の TSC アクセプタ名称も，TSCProxyObject を使用して呼び出すサービスのインタフェースの種類を表します。ただし，同じインタフェースを提供するサービスで，実装内容の違いを識別するために使用します。したがって，TSC アクセプタ名称を設定しないで，"TSC アクセプタ名称なし " とすることもできます。

サービスの種類の表現方法

TSCProxyObject によって利用するサービスの種類は，TSC サービス識別子によって表されます。TSC サービス識別子は，インタフェース名称と TSC アクセプタ名称から構成されます。

- **TSCAcceptor に TSC アクセプタ名称が設定されていない場合**
  TSCProxyObject が呼び出すサービスのインタフェースの種類は，次のように表されます。

TSCアクセプタ名称なしTSCサービス識別子

- **TSCAcceptor に TSC アクセプタ名称が設定されている場合**
  TSCProxyObject が呼び出すサービスの種類は，次のように表されます。

  TSCアクセプタ名称ありTSCサービス識別子

サービスの種類の表現例

- **TSCProxyObject のインタフェース名称が "ABC" で，TSC アクセプタ名称がない場合**
  次のように表される場合，TSCProxyObject は，"ABC::" への要求メッセージを生成し，"ABC::" としてサービスを提供している TSCObject と TSCAcceptor を呼び出すことができます。

  ```
  "ABC::"
  ```

  サーバアプリケーション側で，TSCProxyObject に "ABC::" と指定してサービスを呼び出した場合，その要求を受け付ける TSCObject と TSCAcceptor は，次に示すどちらかです。
  - インタフェース名称 "ABC:" だけ指定された TSCAcceptor，およびその TSCAcceptor に管理される TSCObject
  - インタフェース名称 "ABC" と任意の TSC アクセプタ名称が指定された TSCAcceptor，およびその TSCAcceptor に管理される TSCObject

  つまり，TSCProxyObject でインタフェース名称だけ指定して呼び出した場合，サーバ側では，同じインタフェース名称を持つ TSCAcceptor に管理されるすべての TSCObject が，TSC アクセプタ名称の値に関係なく呼び出されます。
- **TSCProxyObject のインタフェース名称が "ABC" で，TSC アクセプタ名称が "abc" の場合**
  次のように表される場合，TSCProxyObject は "ABC::abc" への要求メッセージを生成し，"ABC::abc" としてサービスを提供している TSCObject と TSCAcceptor を呼び出すことができます。

  ```
  "ABC::abc"
  ```

  サーバアプリケーション側で，TSCProxyObject に "ABC::abc" と指定してサービスを呼び出した場合，その要求を受け付ける TSCObject と TSCAcceptor は，インタフェース名称 "ABC" と TSC アクセプタ名称 "abc" が指定された TSCAcceptor，およびその TSCAcceptor に管理される TSCObject です。
  つまり，TSCProxyObject にインタフェース名称と TSC アクセプタ名称を指定して呼び出した場合，サーバ側では，同じインタフェース名称と同じ TSC アクセプタ名称を持つ TSCAcceptor に管理される TSCObject が呼び出されるため，TSC アクセプタ名称によって呼び出す TSCObject を選択できます。

形式

```
class TSCProxyObject
```

```
{
 public:
  const char* _TSCInterfaceName();
  const char* _TSCAcceptorName();

  TSCInt _TSCTimeout();
  void _TSCTimeout(TSCInt timeout);

  TSCInt _TSCPriority();
  void _TSCPriority(TSCInt priority);

  TSCContext_ptr _TSCContext();
};
```

インクルードファイル

```
#include <tscproxy.h>
```

メソッド

● const char* _TSCInterfaceName()

項目	型・意味
戻り値	インタフェース名称

インタフェース名称を取得します。

インタフェース名称のメモリ領域の管理責任は TSCProxyObject クラスにあるので，ユーザは削除しないでください。

● const char* _TSCAcceptorName()

項目	型・意味
戻り値	TSC アクセプタ名称

TSC アクセプタ名称を取得します。

TSC アクセプタ名称のメモリ領域の管理責任は TSCProxyObject クラスにあるので，ユーザは削除しないでください。

● TSCInt _TSCTimeout()

項目	型・意味（単位）
戻り値	タイムアウト時間（秒）

タイムアウト値（呼び出し時の監視時間）を取得します。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● void _TSCTimeout(TSCInt timeout)

項目	型・意味（単位）	
引数	TSCInt timeout	タイムアウト時間（秒）
戻り値	ありません。	
例外	TSCBadParamException	

タイムアウト値（呼び出し時の監視時間）を秒単位で設定します。"0" を指定した場合，時間監視をしません。監視時間は，メソッド呼び出しごとに変更できます。

アプリケーションプログラムの開始時に -TSCTimeOut オプションを指定しない場合は，監視時間のデフォルト値は "180"（秒）です。-TSCTimeOut オプションを指定する場合は，監視時間のデフォルト値は -TSCTimeOut オプションの指定値になります。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● TSCInt TSCPriority()

項目	型・意味
戻り値	プライオリティ値

プライオリティ値（メソッド呼び出し時の優先順位）を取得します。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● void _TSCPriority(TSCInt priority)

項目	型・意味	
引数	TSCInt priority	プライオリティ値
戻り値	ありません。	
例外	TSCBadParamException	

プライオリティ値（メソッド呼び出し時の優先順位）を設定します。

priority に 1 ～ 8 の値を指定することで，キューイング取り出し時の優先順位を変更できます。priority に指定する値が小さいほど優先度は高くなります。プライオリティ値はリクエスト単位に変更できます。

アプリケーションプログラムの開始時に -TSCRequestPriority オプションを指定しない場合は，プライオリティ値のデフォルト値は "4" です。-TSCRequestPriority オプションを指定する場合は，プライオリティ値のデフォルト値は -TSCRequestPriority オプションの指定値になります。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● TSCContext_ptr _TSCContext()

項目	型・意味
戻り値	TSC コンテキスト

TSCContext を取得します。

戻り値の TSCContext のメモリ領域の管理責任は TSCProxyObject クラスにあるので，削除しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCProxyObject クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
_TSCInterfaceName	○
_TSCAcceptorName	○
_TSCTimeout()	○
_TSCTimeout(TSCInt)	×
_TSCPriority()	○
_TSCPriority(TSCInt)	×
_TSCContext	×
クライアント側からのオブジェクト呼び出し	×

（凡例）

○：できます。

×：できません。

インスタンスの内部参照（アクセス）規則

TSCProxyObject クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

メソッド	複数のスレッド上からの内部参照
_TSCInterfaceName	ありません。
_TSCAcceptorName	ありません。
_TSCTimeout()	ありません。
_TSCTimeout(TSCInt)	ありません。
_TSCPriority()	ありません。

メソッド	複数のスレッド上からの内部参照
_TSCPriority(TSCInt)	ありません。
_TSCContext	ありません。
クライアント側からのオブジェクト呼び出し	生成時に指定した TSCClient 型のインスタンス

TSCRootAcceptor（C++）

TSCRootAcceptor はシステム提供クラスです。

TSCRootAcceptor は，サーバオブジェクトの実行空間を表現するオブジェクトです。クライアント側からの TSC ユーザオブジェクト呼び出し要求を受け付けて，適切な TSC ユーザアクセプタに振り分けます。また，パラレルカウント（常駐するスレッド数）に合わせてスレッドを管理します。

ユーザは，クライアント側にサービスを提供する TSC ユーザオブジェクトの実行空間を構築するときに，サーバアプリケーション内で TSCRootAcceptor クラスのインスタンスを生成します。次に TSCRootAcceptor の特徴を示します。

- 複数の TSCAcceptor を登録できます。
- スレッドの生成や削除などをして，スレッドを管理します。
- 属性として TSC **ルートアクセプタ状態**を持ちます。TSC ルートアクセプタ状態には active 状態と non-active 状態の 2 種類があります。
- TSC **ルートアクセプタ登録名称**を指定して active 状態に遷移できます。
- 属性として**パラレルカウント**（常駐するスレッド数）を持ちます。
- TSCThreadFactory を登録できます。
- 属性として**スケジュール用キューの長さ**を持ちます。

TSCAcceptor の登録

TSCObject の実行空間の構成

TSCRootAcceptor の実行空間で TSCObject を実行させる場合，例えば，
TSCRootAcceptor の管理するスレッド上で TSCObject を実行させる場合，その
TSCObject に対応する TSCAcceptor を TSCRootAcceptor に登録します。
TSCRootAcceptor には複数の TSCAcceptor を登録できます。

提供するサービスの管理

TSCRootAcceptor には複数の TSCAcceptor を登録できます。TSCRootAcceptor は，各 TSCAcceptor の複数の TSC サービス識別子を集め，かつ，重複を削除したものを属性として管理します。TSCRootAcceptor が提供できるサービスの種類は，この TSC サービス識別子によって表現されます。つまり，TSC サービス識別子の列で表されるサービスを提供することになります。

TSC ルートアクセプタ状態

TSC ルートアクセプタ状態ごとのスレッドの管理方式

TSC ルートアクセプタ状態には active 状態と non-active 状態の 2 種類があります。各状態の遷移中の場合も含めて，それぞれの状態のときのスレッドの管理方式を次に示し

ます。

- non-active **状態**
  non-active 状態のオブジェクトが管理するスレッドはありません。
  non-active 状態のときは，TSCAcceptor の登録および削除ができます。TSCAcceptor の登録時には，登録識別子が返されます。TSCAcceptor を削除するときには，その登録識別子を指定します。
  TSCRootAcceptor を生成した時点では non-active 状態です。
- non-active **状態から** active **状態に遷移中**
  パラレルカウント数と同数のスレッドを生成します。また，登録されているすべての TSCAcceptor にオブジェクト管理開始通知を出します。すべての TSCAcceptor がオブジェクトの管理を開始したあと，クライアント側からの TSC ユーザオブジェクト呼び出し要求を受け付けることができます。
- active **状態**
  クライアント側にサービスを提供できる状態です。クライアント側から TSC ユーザオブジェクト呼び出し要求を受け取ると，適切な TSCAcceptor に振り分けます。
  non-active 状態から active 状態に遷移するときに生成されたスレッドは TSCRootAcceptor に管理されています。ユーザは，TSC 経由以外でこれらの TSC ユーザオブジェクトにアクセスしないでください。また，active 状態のときは，TSCAcceptor の登録および削除はできません。
- active **状態から** non-active **状態に遷移中**
  登録されているすべての TSCAcceptor にオブジェクト管理終了通知を出します。また，TSCRootAcceptor の管理下にあるスレッドを削除します。

active **状態での障害通知**

サーバアプリケーション内や TSC デーモンとの接続に障害が発生してスレッドが存続できなくなる場合，そのスレッド上でオブジェクト管理終了通知を TSCAcceptor に渡します。その後，そのスレッドを削除します。なお，このときはまだ active 状態です。

障害が解除されると，再度，スレッドを生成します。その後，そのスレッド上でオブジェクト管理開始通知を TSCAcceptor に渡します。つまり，障害が発生してから解除されるまでの間は，TSCRootAcceptor は，スレッド数がパラレルカウント（常駐するスレッド数）以下の状態で存続します。

TSC **ルートアクセプタ登録名称**

TSCRootAcceptor が active 状態に遷移すると，TSCRootAcceptor と関連づけられている TSCServer に TSC ルートアクセプタ登録名称が登録されます。以降，クライアント側のメソッド呼び出し要求が TSCRootAcceptor に振り分けられるようになります。

TSCRootAcceptor を active 状態に遷移させるときに，TSC ルートアクセプタ登録名称を指定することもできます。activate の呼び出し時に，TSC ルートアクセプタ登録名称を指定する場合と指定しない場合について，それぞれ次に示します。

- **TSC ルートアクセプタ登録名称を指定して activate を呼び出した場合**
  TSC ルートアクセプタ登録名称を "abc" とすると，関連づけられている TSCServer に "abc" として登録されます。
- **TSC ルートアクセプタ登録名称を指定しないで activate を呼び出した場合**
  関連づけられている TSCServer に，デフォルトの TSC ルートアクセプタ登録名称で登録されます。デフォルトの TSC ルートアクセプタ登録名称は "default" ですが，サーバアプリケーションの開始時に指定するコマンドオプション引数 -TSCRootAcceptor によって変更できます。

また，同じ TSC ルートアクセプタ登録名称で，TSCRootAcceptor を同じ TSC デーモンに登録できます。同じ TSC ルートアクセプタ登録名称で登録した場合，スケジュール用キュー（配送機構）が共有されます。ただし，スケジュール用キューを共有する場合，登録する TSCRootAcceptor 間で提供できるサービス内容が一致している必要があります。つまり，同じ TSC ルートアクセプタ登録名称で登録する TSCRootAcceptor は，同じ TSC サービス識別子の列を持っている必要があります。

TSC ユーザスレッドファクトリによる TSC ユーザスレッドの管理

TSCThreadFactory を引数として TSCRootAcceptor を生成した場合を前提にします。この場合，TSCRootAcceptor は，non-active 状態から active 状態に遷移する過程でスレッドを生成したあと，TSCThreadFactory の create を呼び出して，戻り値である TSCThread をそのスレッドに割り当てます。また，active 状態から non-active 状態に遷移する過程で，スレッドごとに割り当てた TSCThread を引数に TSCThreadFactory の destroy を呼び出したあと，スレッドを削除します。

形式

```
class TSCRootAcceptor;
typedef TSCRootAcceptor* TSCRootAcceptor_ptr;

class TSCRootAcceptor
{
 public:

  static TSCRootAcceptor_ptr create(TSCServer_ptr tsc_server);

  static TSCRootAcceptor_ptr create(TSCServer_ptr tsc_server,
                                    TSCThreadFactory_ptr tsc_thr_fact);

  static void destroy(TSCRootAcceptor_ptr tsc_rt_acpt);

  //TSCAcceptorの追加
  TSCInt registerAcceptor(TSCAcceptor_ptr tsc_acpt);

  //TSCAcceptorの削除
  void cancelAcceptor(TSCInt reg_id);

  //パラレルカウントの設定
  void setParallelCount(TSCInt p_count);
```

```
  TSCInt getParallelCount();

  //TSCRootAcceptorの活性化
  TSCInt activate();
  TSCInt activate(const char* rt_acpt_reg_name);

  //TSCRootAcceptorの非活性化
  TSCInt deactivate();

  //スケジュール用キュー長の設定
  void setQueueLength(TSCInt length);
  TSCInt getQueueLength();
};
```

インクルードファイル

```
#include <tscobject.h>
```

メソッド

● static TSCRootAcceptor_ptr create(TSCServer_ptr tsc_server)

項目	型・意味	
引数	TSCServer_ptr tsc_server	接続した TSCServer
戻り値	生成された TSCRootAcceptor	
例外	TSCBadParamException TSCNoMemoryException	

TSCServer と関連づけられた TSCRootAcceptor を生成します。

引数で渡す TSCServer の解放責任，および戻り値で返される TSCRootAcceptor の削除責任はユーザにあるので，適切な状態のときに解放および削除してください。

なお，このメソッドを複数のスレッドから同時に呼び出すことができます。

● static TSCRootAcceptor_ptr create(TSCServer_ptr tsc_server,
TSCThreadFactory_ptr tsc_thr_fact)

項目	型・意味	
引数	TSCServer_ptr tsc_server	接続した TSCServer
	TSCThreadFactory_ptr tsc_thr_fact	TSC ユーザスレッドファクトリオブジェクト
戻り値	生成された TSCRootAcceptor	
例外	TSCBadParamException TSCNoMemoryException	

tsc_server と関連づけられた tsc_thr_fact を保持する TSCRootAcceptor を生成します。

引数で渡す TSCServer の解放責任，引数で渡す TSCThreadFactory のメモリ領域の管

理責任，および戻り値で返される TSCRootAcceptor の削除責任はユーザにあるので，適切な状態のときに解放および削除してください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● static void destroy(TSCRootAcceptor_ptr tsc_rt_acpt)

項目	型・意味	
引数	TSCRootAcceptor_ptr tsc_rt_acpt	削除する TSCRootAcceptor
戻り値	ありません。	
例外	ありません。	

tsc_rt_acpt を削除します。削除した TSCRootAcceptor にはアクセスしないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● TSCInt registerAcceptor(TSCAcceptor_ptr tsc_acpt)

項目	型・意味	
引数	TSCAcceptor_ptr tsc_acpt	登録する TSCAcceptor
戻り値	TSC ユーザアクセプタの登録識別子	
例外	TSCBadParamException TSCNoMemoryException TSCNoPermissionException	

TSCAcceptor を登録します。active 状態のときは登録できません。

登録する TSCAcceptor のメモリ領域の管理責任はユーザにあるので，適切な状態のときに削除してください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● void cancelAcceptor(TSCInt reg_id)

項目	型・意味	
引数	TSCInt reg_id	削除する TSC ユーザアクセプタの登録識別子
戻り値	ありません。	
例外	TSCBadParamException TSCNoPermissionException	

TSCAcceptor オブジェクトの登録を削除します。reg_id には registerAcceptor で戻された値を指定してください。activate 状態のときと，active 状態のときは登録を削除できません。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● void setParallelCount(TSCInt p_count)

項目	型・意味	
引数	TSCInt p_count	パラレルカウント
戻り値	ありません。	
例外	TSCBadParamException TSCNoPermissionException	

パラレルカウント（常駐するスレッド数）を設定します。active 状態のときは設定できません。

サーバアプリケーションの開始時にコマンドオプション引数 -TSCParallelCount を指定しない場合，パラレルカウントのデフォルト値は "1" です。コマンドオプション引数 -TSCParallelCount を指定する場合は，パラレルカウントのデフォルト値はその指定値となります。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● TSCInt getParallelCount()

項目	型・意味
戻り値	パラレルカウント

パラレルカウント（常駐するスレッド数）を取得します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● TSCInt activate()

項目	型・意味
戻り値	常に 0
例外	TSCBadInvOrderException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException ユーザが通知する各種例外

デフォルトの TSC ルートアクセプタ登録名称で，active 状態に遷移します。

サーバアプリケーションの開始時に指定するコマンドオプション引数 -TSCRootAcceptor を指定しない場合，TSC ルートアクセプタ登録名称のデフォルト値は "default" です。コマンドオプション引数 -TSCRootAcceptor を指定した場合，TSC ルートアクセプタのデフォルト値はその指定値となります。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● TSCInt activate(const char* rt_acpt_reg_name)

項目	型・意味	
引数	const char* rt_acpt_reg_name	TSC ルートアクセプタの登録名称
戻り値	常に 0	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException ユーザが通知する各種例外	

指定した TSC ルートアクセプタ登録名称で，active 状態に遷移します。
rt_acpt_reg_name に文字列を指定する場合は，1 ～ 31 文字で指定してください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● TSCInt deactivate()

項目	型・意味
戻り値	常に 0
例外	TSCBadInvOrderException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException

non-active 状態に遷移します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● void setQueueLength(TSCInt length)

項目	型・意味	
引数	TSCInt length	スケジュール用キューの長さ
戻り値	ありません。	
例外	TSCBadParamException TSCNoPermissionException	

生成するスケジュール用キューの長さを指定します。指定できる範囲は 1 ～ 32767 です。
active 状態のときは指定できません。

このメソッドでスケジュール用キューの長さを指定しない場合，生成されるスケジュール用キューの長さは，tscstartprc コマンドまたはサーバアプリケーションの開始コマンドの -TSCQueueLength オプションで指定した長さになります。スケジュール用キュー

を共有する場合，すでに生成されているスケジュール用キューの長さが有効になります。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● TSCInt getQueueLength()

項目	型・意味
戻り値	スケジュール用キューの長さ

non-active 状態の場合は，setQueueLength() メソッド，または tscstartprc コマンドもしくはサーバアプリケーションの開始コマンドの -TSCQueueLength オプションで指定したスケジュール用キューの長さを取得します。setQueueLength() メソッドが未発行で，かつ tscstartprc コマンドまたはサーバアプリケーションの開始コマンドの -TSCQueueLength オプションが指定されていないときの戻り値は "0" となります。

active 状態の場合は，現在有効になっているスケジュール用キューの長さを取得します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

TSCRootAcceptor の生成と削除

TSCServer を引数に指定した TSCRootAcceptor の create で生成し，TSCRootAcceptor を引数に指定した TSCRootAcceptor の destroy で削除します。TSCRootAcceptor クラスのインスタンスへの内部参照（アクセス）があるときは削除できないため，インスタンスへの内部参照（アクセス）をなくした状態で削除してください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCRootAcceptor クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
create(TSCServer_ptr)	○
create(TSCServer_ptr,TSCThreadFactory_ptr)	○
destroy	○
registerAcceptor	×
cancelAcceptor	×
setParallelCount	×
getParallelCount	○
activate()	×
activate(const char*)	×
deactivate()	×

メソッド	複数のスレッド上からの同時呼び出し
setQueueLength	×
getQueueLength	○

（凡例）
○：できます。
×：できません。

インスタンスの公開メソッド呼び出し規則

TSCRootAcceptor クラスのインスタンスがほかのクラスのインスタンスの公開メソッドを呼び出す規則を次に示します。

メソッド	公開メソッド呼び出し
activate()	生成時に指定された TSCThreadFactory 型のインスタンス
activate(const char*)	生成時に指定された TSCThreadFactory 型のインスタンス
deactivate()	生成時に指定された TSCThreadFactory 型のインスタンス

インスタンスの内部参照（アクセス）規則

TSCRootAcceptor クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

メソッド	内部参照
registerAcceptor	ありません。
cancelAcceptor	ありません。
setParallelCount	ありません。
getParallelCount	ありません。
activate()	生成時に指定された TSCServer 型のインスタンス 登録されている TSCAcceptor 型のインスタンス
activate(const char*)	生成時に指定された TSCServer 型のインスタンス
deactivate()	生成時に指定された TSCServer 型のインスタンス
setQueueLength	ありません。
getQueueLength	ありません。
active 状態のとき	registerAcceptor の引数で指定された TSCAcceptor 型のインスタンス （TSCRootAcceptor に登録されている TSCAcceptor 型のインスタンス）

なお，TSCRootAcceptor クラスのインスタンスを解放したあと，このインスタンスを内部参照するインスタンスからのアクセスは，メモリアクセス違反となります。OTM はこのときの動作を保証しません。また，複数のスレッド上から同時に，TSCRootAcceptor クラスの同じインスタンスを内部参照できます。

TSCServer（C++）

TSCServer はシステム提供クラスです。

TSCServer は，TSC デーモン中のサーバアプリケーション管理部分を参照するクラスです。サーバアプリケーション側の機能操作の要求は，TSCServer を経由して TSC デーモンに渡されます。また，TSC ユーザプロキシを使用したクライアントアプリケーション側からの TSC ユーザオブジェクト呼び出し要求を受けて，TSC ルートアクセプタに振り分けます。

ユーザはサーバアプリケーションが TSC デーモンと接続するときに，TSCServer クラスのインスタンスを取得します。次に TSCServer の特徴を示します。

- 属性として TSC ドメイン名称と TSC 識別子を持ちます。

TSCServer と接続

サーバアプリケーションと TSC デーモン間の接続は，サーバアプリケーションプロセス内で TSCServer を最初に取得するときに確立されます。その後，同じ TSC デーモンに対して TSCServer を取得する場合は，その接続を共有します。逆に，取得したすべての TSCServer を解放すると接続が切断されます。

一つのサーバアプリケーションから複数の TSC デーモンへ接続を確立することもできます。また，サーバアプリケーション側の機能操作の要求が，この接続を経由して TSC デーモンに渡される場合，並行して処理されないで順番に処理されます。TSC デーモンからのオブジェクト呼び出し要求が，この接続を経由してサーバアプリケーションに配送される場合は，並行して処理されます。

形式

```
class TSCServer;
typedef TSCServer* TSCServer_ptr;

class TSCServer
{
public:
  const char* getTSCDomainName();
  const char* getTSCID();
};
```

インクルードファイル

```
#include <tscobject.h>
```

メソッド

● const char* getTSCDomainName()

項目	型・意味
戻り値	TSC ドメイン名称

TSC ドメイン名称を返します。

TSC ドメイン名称のメモリ領域の管理責任は TSCServer クラスにあるので，ユーザは削除しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● const char* getTSCID()

項目	型・意味
戻り値	TSC 識別子

TSC 識別子を返します。

TSC 識別子のメモリ領域の管理責任は TSCServer クラスにあるので，ユーザは削除しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

TSCServer の取得と解放

TSCServer は TSCAdm の getTSCServer で取得し，TSCAdm の releaseTSCServer で解放します。TSCServer クラスのインスタンスへの内部参照（アクセス）があるときは解放できないため，TSCServer クラスのインスタンスへの内部参照（アクセス）をなくした状態で解放してください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCServer クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
getTSCDomainName	できます。
getTSCID	できます。

インスタンスの内部参照（アクセス）規則

TSCServer クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

タイミング	内部参照
getTSCDomainName	ありません。
getTSCID	ありません。
関連づけがある TSCRootAcceptor が active 状態	関連づけがある TSCRootAcceptor 型のインスタンス

なお，TSCServer クラスのインスタンスを解放したあと，このインスタンスを内部参照するインスタンスからのアクセスは，メモリアクセス違反となります。OTM は，この際の動作を保証しません。また，複数のスレッド上から同時に，TSCServer クラスの同じインスタンスを内部参照できます。

TSCSessionProxy（C++）

TSCSessionProxy はシステム提供クラスです。

TSCSessionProxy は，TSCObject の代理クラスです。TSCSessionProxy を呼び出すと，OTM のスケジューリング機構を経由してステートフルに TSCObject が呼び出されます。TSCSessionProxy の TSCProxyObject との違いを次に示します。

TSCSessionProxy の特徴

- TSCProxyObject の持つ属性に加え，セション呼び出しインターバル監視時間（呼び出しと呼び出しの間の監視時間）が属性に追加されます。
- _TSCStart() メソッドでセションを確立します。
  セションを確立すると，TSCSessionProxy とサーバアプリケーションのインスタンスを対応づけます。_TSCStart() メソッド発行後は，_TSCStop() メソッドの発行まで，対応づけたインスタンスにリクエストを要求します。対応づけられるサーバアプリケーションのインスタンスは TSCObject です。TSCObject と対応づけた場合は，次のことに注意してください。
  - TSCObject のインスタンスを保持しているスレッドも対応づける。
  - TSCObject のインスタンスはセションを確立していないほかのクライアントアプリケーションからのリクエストを受け付けられない。
  - すべての TSCObject のインスタンスが対応づけられた場合は，新しいクライアントアプリケーションからのリクエストを受け付けられない。
- _TSCStop() メソッドを発行すると，TSCSessionProxy とサーバアプリケーションのインスタンスとのセションを解放します。
- コンストラクタに指定する TSC アクセプタ名称は，_TSCStart() メソッド発行時に対応づけるインスタンスを決定するために使用されます。
- TSC アクセプタ名称を指定していないコンストラクタで生成した場合は，_TSCStart() メソッドで任意のインスタンスに対応づけ，対応づけた TSC アクセプタ名称を _TSCStop() メソッド発行時まで引き継ぎます。

形式

```
class TSCSessionProxy
{
 public:
  void _TSCStart();
  void _TSCStop();

  TSCInt _TSCSessionInterval();
  void _TSCSessionInterval(TSCInt session_interval);

  const char* _TSCInterfaceName();
  const char* _TSCAcceptorName();
```

```
  TSCInt _TSCTimeout();
  void _TSCTimeout(TSCInt timeout);

  TSCInt _TSCPriority();
  void _TSCPriority(TSCInt priority);

  TSCContext_ptr _TSCContext();
}
```

インクルードファイル

```
#include <tscproxy.h>
```

メソッド

● void _TSCStart()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResponseException TSCObjectNotExistException TSCTransientException

セションを開始します。

セションを開始すると TSC ユーザプロキシとサーバアプリケーションのインスタンスを対応づけます。また，アクセプタ名称を指定していないコンストラクタでインスタンスを生成した場合は，_TSCStart() メソッドで対応づけたインスタンスのアクセプタ名称を _TSCStop() メソッドの発行時まで引き継ぎます。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

_TSCStart() メソッドが正常終了した場合は，セション呼び出しが成功しても失敗しても，必ず _TSCStop() メソッドを発行してください。

● void _TSCStop()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResponseException TSCTransientException

セションを解放します。

セションを解放すると TSC ユーザプロキシとサーバアプリケーションのインスタンスの対応づけも解放されます。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● TSCInt _TSCSessionInterval()

項目	型・意味（単位）
戻り値	セション呼び出しインターバル監視時間（秒）

セション呼び出しインターバル監視時間を取得します。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● void _TSCSessionInterval(TSCInt session_interval)

項目	型・意味（単位）	
引数	TSCInt session_interval	セション呼び出しインターバル監視時間（秒）
戻り値	ありません。	
例外	TSCBadParamException	

セション呼び出しインターバル監視時間を秒単位で設定します。メソッド呼び出しごとに変更できます。

アプリケーションプログラムの開始時に -TSCSessionInterval オプションを指定しない場合は，監視時間のデフォルト値は "180"（秒）です。-TSCSessionInterval オプションを指定する場合は，監視時間のデフォルト値は -TSCSessionInterval オプションの指定値になります。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● const char* _TSCInterfaceName()

項目	型・意味
戻り値	インタフェース名称

インタフェース名称を取得します。

インタフェース名称のメモリ領域の管理責任は TSCSessionProxy クラスにあるので，ユーザは削除しないでください。

● const char* _TSCAcceptorName()

項目	型・意味
戻り値	TSC アクセプタ名称

TSC アクセプタ名称を取得します。

取得する TSC アクセプタ名称を次に示します。

- セション呼び出し中の場合
  セション呼び出し中のサーバアプリケーションのアクセプタ名称
- セション呼び出し中でない場合
  TSCSessionProxy 生成時に指定したアクセプタ名称

TSC アクセプタ名称のメモリ領域の管理責任は TSCSessionProxy クラスにあるので，ユーザは削除しないでください。また，セション呼び出し中に取得した TSC アクセプタ名称のメモリ領域は _TSCStop() メソッド発行後には参照できません。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● TSCInt _TSCTimeout()

項目	型・意味（単位）
戻り値	タイムアウト時間（秒）

タイムアウト値（呼び出し時の監視時間）を取得します。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● void _TSCTimeout(TSCInt timeout)

項目	型・意味（単位）	
引数	TSCInt timeout	タイムアウト時間（秒）
戻り値	ありません。	
例外	TSCBadParamException	

タイムアウト値（呼び出し時の監視時間）を秒単位で設定します。"0" を指定した場合は，時間監視をしません。監視時間は，メソッド呼び出しごとに変更できます。

アプリケーションプログラムの開始時に -TSCTimeOut オプションを指定しない場合は，監視時間のデフォルト値は "180"（秒）です。-TSCTimeOut オプションを指定する場合は，監視時間のデフォルト値は -TSCTimeOut オプションの指定値になります。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

タイムアウト値が有効になるメソッドを次に示します。

- _TSCStart() メソッド
- クライアント側からのオブジェクト呼び出しメソッド
- _TSCStop() メソッド

● TSCInt _TSCPriority()

項目	型・意味
戻り値	プライオリティ値

プライオリティ値（メソッド呼び出し時の優先順位）を取得します。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● void _TSCPriority(TSCInt priority)

項目	型・意味	
引数	TSCInt priority	プライオリティ値
戻り値	ありません。	
例外	TSCBadParamException	

プライオリティ値（メソッド呼び出し時の優先順位）を設定します。

priority に 1 ～ 8 の値を指定することで，キューイング取り出し時の優先順位を変更できます。priority に指定する値が小さいほど優先度は高くなります。プライオリティ値はリクエスト単位に変更できます。

アプリケーションプログラムの開始時に -TSCRequestPriority オプションを指定しない場合は，プライオリティ値のデフォルト値は "4" です。-TSCRequestPriority オプションを指定する場合は，プライオリティ値のデフォルト値は -TSCRequestPriority オプションの指定値になります。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

プライオリティ値が有効になるのは _TSCStart() メソッドだけです。

● TSCContext_ptr _TSCContext()

項目	型・意味
戻り値	TSC コンテキスト

TSC コンテキストを取得します。

戻り値の TSC コンテキストのメモリ領域の管理責任は TSCSessionProxy クラスにあるので，ユーザは削除しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCSessionProxy クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
_TSCStart()	×
_TSCStop()	×
_TSCSessionInterval()	○
_TSCSessionInterval(TSCInt)	×
_TSCInterfaceName()	○
_TSCAcceptorName()	○
_TSCTimeout()	○
_TSCTimeout(TSCInt)	×
_TSCPriority()	○
_TSCPriority(TSCInt)	×
_TSCContext	×
クライアント側からのオブジェクト呼び出し	×

（凡例）

○：できます。

×：できません。

インスタンスの内部参照（アクセス）規則

TSCSessionProxy クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

メソッド	複数のスレッド上からの内部参照
_TSCStart()	ありません。
_TSCStop()	ありません。
_TSCSessionInterval()	ありません。
_TSCSessionInterval(TSCInt)	ありません。
_TSCInterfaceName()	ありません。
_TSCAcceptorName()	ありません。
_TSCTimeout()	ありません。
_TSCTimeout(TSCInt)	ありません。
_TSCPriority()	ありません。
_TSCPriority(TSCInt)	ありません。
_TSCContext	ありません。
クライアント側からのオブジェクト呼び出し	生成時に指定した TSCClient 型のインスタンス

セション呼び出し機能使用中のメソッド呼び出し規則

セション呼び出し機能使用中（_TSCStart() メソッド呼び出し完了から _TSCStop() メソッド呼び出し完了まで）の，TSCSessionProxy クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	セション呼び出し中での呼び出し
_TSCStart()	×
_TSCStop()	○
_TSCSessionInterval()	○
_TSCSessionInterval(TSCInt)	○
_TSCInterfaceName()	○
_TSCAcceptorName()	○
_TSCTimeout()	○
_TSCTimeout(TSCInt)	○
_TSCPriority()	○
_TSCPriority(TSCInt)	○
_TSCContext	○
クライアント側からのオブジェクト呼び出し	○

（凡例）

○：できます。

×：できません。

TSCSystemException（C++）

TSCSystemException はシステム提供例外クラスです。

TSCSystemException は，OTM での例外用の基本クラスです。次に TSCSystemException の特徴を示します。

- 属性としてエラーコード，内容コード，場所コード，完了状態，および四つの保守コードを持ちます。
- ユーザはこのクラスのインスタンスを生成できません。
- ユーザはこのクラスの型で，派生クラスの例外を catch できます。

エラーコード

エラーコードは，TSCSystemException クラスの派生クラスに対応するコードです。TSCSystemException クラスの型で例外を catch した場合，エラーコードを参照することで例外の種類を知ることができます。

定数値の順に並べたエラーコードの一覧を次の表に示します。エラーコードに対応する各派生クラスについては，この章の「TSCSystemException の派生クラス（C++）」を参照してください。

表 3-5　エラーコードの一覧（C++）

エラーコード	例外クラス
BAD_PARAM	TSCBadParamException
NO_MEMORY	TSCNoMemoryException
COMM_FAILURE	TSCCommFailureException
NO_PERMISSION	TSCNoPermissionException
INTERNAL	TSCInternalException
MARSHAL	TSCMarshalException
INITIALIZE	TSCInitializeException
NO_IMPLEMENT	TSCNoImplementException
BAD_OPERATION	TSCBadOperationException
NO_RESOURCES	TSCNoResourcesException
NO_RESPONSE	TSCNoResponseException
BAD_INV_ORDER	TSCBadInvOrderException
TRANSIENT	TSCTransientException
OBJECT_NOT_EXIST	TSCObjectNotExistException
UNKNOWN	TSCUnknownException
INV_OBJREF	TSCInvObjrefException

エラーコード	例外クラス
IMP_LIMIT	TSCImpLimitException
BAD_TYPECODE	TSCBadTypecodeException
PERSIST_STORE	TSCPersistStoreException
FREE_MEM	TSCFreeMemException
INV_IDENT	TSCInvIdentException
INV_FLAG	TSCInvFlagException
INTF_REPOS	TSCIntfReposException
BAD_CONTEXT	TSCBadContextException
OBJ_ADAPTER	TSCObjAdapterException
DATA_CONVERSION	TSCDataConversionException

個々のエラーコードの詳細については，「付録 A　エラーコード一覧」を参照してください。

内容コード

内容コードは，障害の詳しい情報を示すコードです。エラーコード間で内容コードが重複することはないため，ユニークな定数を持ちます。したがって，TSCSystemException クラスの型で例外を catch する場合，内容コードを参照することで詳細な情報を知ることができます。ユーザが TSCSystemException の派生クラスのインスタンスを生成して throw する場合，ユーザ用に割り当てられている値を内容コードに設定してください。

内容コードの分類を次の表に示します。

表 3-6　内容コードの分類（C++）

分類	内容コードの範囲
ユーザアプリケーション用	0 ～ 999
OTM システム予約	1000 ～ 30000

個々の内容コードの詳細については，「付録 D　内容コード一覧」を参照してください。

場所コード

場所コードは障害が発生した場所を示します。ユーザが TSCSystemException の派生クラスのインスタンスを生成して throw する場合，PLACE_CODE_USER_AP を設定してください。

場所コードの一覧を次の表に示します。

表 3-7　場所コードの一覧（C++）

場所コード	場所
PLACE_CODE_USER_AP	ユーザアプリケーション
PLACE_CODE_SERV	OTM のサーバ機能部分
PLACE_CODE_DAEMON	TSC デーモン
PLACE_CODE_CLNT	OTM のクライアント機能部分
PLACE_CODE_CLNT_REG	TSC レギュレータ
PLACE_CODE_STUB	TSC ユーザプロキシ（スタブ）
PLACE_CODE_SKELTON	TSC ユーザスケルトン（スケルトン）
PLACE_CODE_ORBGW	TSCORB コネクタ

完了状態

完了状態は，障害が発生したときにメソッド呼び出しが完了しているかどうかを示します。

完了状態の一覧を次の表に示します。

表 3-8　完了状態の一覧（C++）

完了状態	説明
COMPLETED_NO	メソッド呼び出しが完了していません。
COMPLETED_MAYBE	メソッド呼び出しの完了状態を決定できません。
COMPLETED_YES	メソッド呼び出しが完了しています。

形式

```
class TSCSystemException : public CORBA::SystemException
{

 public:
  //各種情報取得
  TSCInt getErrorCode();
  TSCInt getDetailCode();
  TSCInt getPlaceCode();
  TSCInt getCompletionStatus();
  TSCInt getMaintenanceCode1();
  TSCInt getMaintenanceCode2();
  TSCInt getMaintenanceCode3();
  TSCInt getMaintenanceCode4();

  //エラーコード
  static const TSCInt BAD_PARAM = 1;
  static const TSCInt NO_MEMORY = 2;
  static const TSCInt COMM_FAILURE = 3;
  static const TSCInt NO_PERMISSION = 4;
```

```
static const TSCInt INTERNAL = 5;
static const TSCInt MARSHAL = 6;
static const TSCInt INITIALIZE = 7;
static const TSCInt NO_IMPLEMENT = 8;
static const TSCInt BAD_OPERATION = 9;
static const TSCInt NO_RESOURCES = 10;
static const TSCInt NO_RESPONSE = 11;
static const TSCInt BAD_INV_ORDER = 12;
static const TSCInt TRANSIENT = 13;
static const TSCInt OBJECT_NOT_EXIST = 14;
static const TSCInt UNKNOWN = 15;
static const TSCInt INV_OBJREF = 16;
static const TSCInt IMP_LIMIT = 17;
static const TSCInt BAD_TYPECODE = 18;
static const TSCInt PERSIST_STORE = 19;
static const TSCInt FREE_MEM = 20;
static const TSCInt INV_IDENT = 21;
static const TSCInt INV_FLAG = 22;
static const TSCInt INTF_REPOS = 23;
static const TSCInt BAD_CONTEXT = 24;
static const TSCInt OBJ_ADAPTER = 25;
static const TSCInt DATA_CONVERSION = 26;


//
//詳細コード
//
//BAD_PARAM
static const TSCInt INVALID_TIMEOUT = 1001;
static const TSCInt INVALID_RT_ACPT_NAME = 1002;
static const TSCInt INVALID_PARALLEL_COUNT = 1003;
static const TSCInt INVALID_ACPT_NAME = 1004;
static const TSCInt OBJ_FACT_IS_NULL = 1005;
static const TSCInt ACPT_IS_NULL = 1006;
static const TSCInt INVALID_ACPT_REGID = 1007;
static const TSCInt INVALID_TSCID = 1008;
static const TSCInt INVALID_DOMAIN_NAME = 1009;
static const TSCInt INVALID_OP_PARAM = 1010;
static const TSCInt SERV_IS_NULL = 1011;
static const TSCInt CLNT_IS_NULL = 1012;
static const TSCInt ORB_IS_NULL = 1013;
static const TSCInt DOMAIN_IS_NULL = 1015;
static const TSCInt INVALID_REQUEST_WAY = 1016;
static const TSCInt INVALID_PRIORITY = 1017;
static const TSCInt THREAD_FACT_IS_NULL = 1018;
static const TSCInt PROXY_IS_NULL = 1020;
static const TSCInt OBJECT_IS_NULL = 1021;
static const TSCInt INVALID_FLAG = 1022;
static const TSCInt INVALID_WATCH_TIME = 1027;
static const TSCInt WATCH_TIME_IS_NULL = 1028;
static const TSCInt INVALID_RETRY_REQUIREMENT = 1029;
static const TSCInt INVALID_SESSION_INTERVAL = 1038;
static const TSCInt INVALID_QUEUE_LENGTH = 1047;

//NO_MEMORY
static const TSCInt MEM_ALLOC_FAILURE = 2001;

//COMM_FAILURE
```

```
static const TSCInt SEND_CLNT_FAILURE = 3004;
static const TSCInt SEND_THIN_CLNT_FAILURE = 3005;
static const TSCInt SEND_SERV_FAILURE = 3006;
static const TSCInt SEND_TSCD_FAILURE = 3007;
static const TSCInt BASIC_CONN_FAILURE = 3009;
static const TSCInt CONN_FAILURE = 3010
static const TSCInt INCOMPATIBLE_PROTOCOL = 3011;
static const TSCInt NOT_IGNORE_PROTOCOL = 3012;
static const TSCInt DEACTIVATE_FAILURE = 3021;

//NO_PERMISSION
static const TSCInt CALL_IN_HOLD = 4001;
static const TSCInt RT_ACPT_IS_ACTIVE = 4002;
static const TSCInt SERV_CONN_IN_END = 4005;
static const TSCInt ACTIVATE_IN_END = 4006;
static const TSCInt CLNT_CONN_IN_END = 4007;
static const TSCInt DIFF_THREAD_CALL = 4008;
static const TSCInt CALL_IN_END = 4009;
static const TSCInt ACPT_NOT_REGISTED = 4010;
static const TSCInt SERV_CONN_IN_START = 4011;
static const TSCInt CLNT_CONN_IN_START = 4012;
static const TSCInt TSCD_IS_NOT_MY_HOST = 4013;
static const TSCInt OVER_ACPT_REGI = 4014;
static const TSCInt NOT_SUPPORTED = 4015;
static const TSCInt ACTIVATE_IN_START = 4016;
static const TSCInt DEACTIVATE_IN_END = 4018;
static const TSCInt CLNT_DISCONN_IN_END = 4020;
static const TSCInt ACTIVATE_WITH_DIFF_PROP = 4022;
static const TSCInt CLNT_INIT_IN_END = 4023;
static const TSCInt SERV_INIT_IN_END = 4024;
static const TSCInt CLNT_INIT_IN_START = 4025;
static const TSCInt SERV_INIT_IN_START = 4026;
static const TSCInt NOT_ACCEPT_OBJECT = 4027;
static const TSCInt CLNT_COMMAND_START = 4028;
static const TSCInt WATCH_IS_STARTED = 4029;
static const TSCInt WATCH_IS_STOPPED = 4030;
static const TSCInt SAME_APID_EXIST = 4031;
static const TSCInt FILE_ACCESS_FAILURE = 4032;
static const TSCInt SESSION_IN_END = 4033;
static const TSCInt SESSION_IN_CALL = 4034;

//INTERNAL
static const TSCInt PROPERTIES_FAILURE = 5001;
static const TSCInt MSG_TYPE_FAILURE = 5002;
static const TSCInt MUTEX_FAILURE = 5003;
static const TSCInt SIG_COND_FAILURE = 5004;
static const TSCInt EVENT_FAILURE = 5005;
static const TSCInt SH_MEM_FAILURE = 5006;
static const TSCInt THREAD_CREATE_FAILURE = 5007;
static const TSCInt TSD_FAILURE = 5008;
static const TSCInt CBL_ADAPTER_ERROR = 5009;
static const TSCInt SYSTEM_TIME_FAILURE = 5010;
static const TSCInt PROGRAM_ERROR = 5999;

//MARSHAL
static const TSCInt INVALID_STREAM_LEN = 6001;
static const TSCInt INVALID_STREAM_VALUE = 6002;
static const TSCInt MARSHAL_OTHERS = 6003;
static const TSCInt REQ_MARSHAL_FAILURE = 6004;
```

```
static const TSCInt REQ_UNMARSHAL_FAILURE = 6005;
static const TSCInt REP_MARSHAL_FAILURE = 6006;
static const TSCInt REP_UNMARSHAL_FAILURE = 6007;
static const TSCInt UEXCEPT_MARSHAL_FAILURE = 6008;
static const TSCInt UEXCEPT_UNMARSHAL_FAILURE = 6009;
static const TSCInt MARSHAL_ERROR = 6010;

//INITIALIZE
static const TSCInt INVALID_DEF_TIMEOUT = 7002;
static const TSCInt INVALID_DEF_RT_ACPT = 7003;
static const TSCInt INVALID_DEF_PARALLEL_COUNT = 7004;
static const TSCInt LOAD_SHLIB_FAILURE = 7005;
static const TSCInt INVALID_DEF_TSCID = 7006;
static const TSCInt INVALID_DEF_DOMAIN_NAME = 7007;
static const TSCInt INVALID_DEF_WITH_SYSTEM = 7008;
static const TSCInt INVALID_ENV_TSCDIR = 7009;
static const TSCInt MTRACE_FAILURE = 7010;
static const TSCInt INVALID_DEF_NICE =7011;
static const TSCInt INVALID_DEF_PRIORITY = 7012;
static const TSCInt INVALID_DEF_ACPT = 7013;
static const TSCInt INVALID_PRC_KIND = 7014;
static const TSCInt INVALID_DEF_REQUEST_WAY = 7015;
static const TSCInt
    INVALID_DEF_CLIENT_MESSAGE_BUFFER_COUNT = 7016;
static const TSCInt INVALID_DEF_APID = 7017;
static const TSCInt INVALID_DEF_WATCH_TIME = 7018;
static const TSCInt INVALID_DEF_WATCH_METHOD = 7019;
static const TSCInt INVALID_DEF_MY_HOST = 7020;
static const TSCInt INVALID_DEF_REBIND_TIMES = 7021;
static const TSCInt INVALID_DEF_REBIND_INTERVAL = 7022;
static const TSCInt INVALID_DEF_RETRY_REFERENCE = 7024;
static const TSCInt INVALID_FILE_FORMAT = 7025;
static const TSCInt INVALID_DEF_RETRY_WAY = 7030;
static const TSCInt ALREADY_SHUTDOWN = 7031;
static const TSCInt INVALID_DEF_EXCEPT_CONVERT_FILE = 7034;
static const TSCInt ENTRY_FAILURE = 7035;
static const TSCInt INVALID_DEF_SESSION_INTERVAL = 7036;
static const TSCInt INVALID_DEF_QUEUE_LENGTH = 7040;

//NO_IMPLEMENT
static const TSCInt NO_SUCH_INTERF = 8001;
static const TSCInt NO_SUCH_ACPT = 8002;
static const TSCInt NO_SUCH_OP_NAME = 9001;

//NO_RESOURCES
static const TSCInt OVER_MAX_CLNT = 10001;
static const TSCInt OVER_MAX_SERV = 10002;
static const TSCInt OVER_ADM_MAX_CLNT = 10005;
static const TSCInt OVER_ADM_MAX_SERV = 10006;
static const TSCInt OVER_MAX_RT_ACPT_REGI = 10007;
static const TSCInt OVER_MAX_THIN_CLIENT = 10008;
static const TSCInt OVER_MAX_DISPATCH_PARALLEL = 10009;
static const TSCInt OVER_MAX_REQUEST_COUNT = 10010;
static const TSCInt OVER_MAX_ORB_CLIENT = 10011;

//NO_RESPONSE
static const TSCInt TIMED_OUT = 11001;
```

```
//BAD_INV_ORDER
static const TSCInt ALREADY_ACTIVE = 12002;
static const TSCInt ALREADY_DEACTIVE = 12003;
static const TSCInt CLNT_NOT_INITIALIZED = 12004;
static const TSCInt SERV_NOT_INITIALIZED = 12005;
static const TSCInt ALREADY_INITCLNT = 12006;
static const TSCInt ALREADY_INITSERV = 12007;
static const TSCInt ALREADY_SERV_ML = 12008;
static const TSCInt ALREADY_SESSION_START = 12013;
static const TSCInt SESSION_NOT_START = 12014;

//TRANSIENT
static const TSCInt REBIND_FAILURE = 13001;
static const TSCInt ALL_CONN_FAILURE = 13002;

//OBJECT_NOT_EXIST
static const TSCInt SERV_NO_SUCH_INTERF = 14001;
static const TSCInt SERV_NO_SUCH_ACPT = 14002;

//UNKNOWN
static const TSCInt COMMON_EXCEPTION = 15001;
static const TSCInt INVALID_USER_EXCEPTION = 15002;

//INV_OBJREF
static const TSCInt FACTORY_CREATE_FAILURE = 16001;
static const TSCInt THREAD_FACTORY_CREATE_FAILURE = 16002;
static const TSCInt FACTORY_DESTROY_FAILURE = 16003;
static const TSCInt THREAD_FACTORY_DESTROY_FAILURE = 16004;
static const TSCInt CREATED_OBJECT_IS_NULL = 16005;
static const TSCInt CREATED_THREAD_OBJECT_IS_NULL = 16006;

static const TSCInt TPBROKER_BAD_PARAM = 1998;
static const TSCInt TPBROKER_NO_MEMORY = 2998;
static const TSCInt TPBROKER_COMM_FAILURE = 3998;
static const TSCInt TPBROKER_NO_PERMISSION = 4998;
static const TSCInt TPBROKER_INTERNAL = 5998;
static const TSCInt TPBROKER_MARSHAL = 6998;
static const TSCInt TPBROKER_INITIALIZE = 7998;
static const TSCInt TPBROKER_NO_IMPLEMENT = 8998;
static const TSCInt TPBROKER_BAD_OPERATION = 9998;
static const TSCInt TPBROKER_NO_RESOURCES = 10998;
static const TSCInt TPBROKER_NO_RESPONSE = 11998;
static const TSCInt TPBROKER_BAD_INV_ORDER = 12998;
static const TSCInt TPBROKER_TRANSIENT = 13998;
static const TSCInt TPBROKER_OBJECT_NOT_EXIST = 14998;
static const TSCInt TPBROKER_UNKNOWN = 15998;
static const TSCInt TPBROKER_INV_OBJREF = 16998;
static const TSCInt TPBROKER_IMP_LIMIT = 17998;
static const TSCInt TPBROKER_BAD_TYPECODE = 18998;
static const TSCInt TPBROKER_PERSIST_STORE = 19998;
static const TSCInt TPBROKER_FREE_MEM = 20998;
static const TSCInt TPBROKER_INV_IDENT = 21998;
static const TSCInt TPBROKER_INV_FLAG = 22998;
static const TSCInt TPBROKER_INTF_REPOS = 23998;
static const TSCInt TPBROKER_BAD_CONTEXT = 24998;
static const TSCInt TPBROKER_OBJ_ADAPTER = 25998;
static const TSCInt TPBROKER_DATA_CONVERSION = 26998;

//場所コード
static const TSCInt PLACE_CODE_USER_AP = 1;
static const TSCInt PLACE_CODE_SERV = 2;
```

```
  static const TSCInt PLACE_CODE_DAEMON = 3;
  static const TSCInt PLACE_CODE_CLNT = 4;
  static const TSCInt PLACE_CODE_CLNT_REG = 5;
  static const TSCInt PLACE_CODE_STUB = 6;
  static const TSCInt PLACE_CODE_SKELTON = 7;
  static const TSCInt PLACE_CODE_ORBGW = 8;

  // 完了状態
  static const TSCInt COMPLETED_NO = -1;
  static const TSCInt COMPLETED_MAYBE = 0;
  static const TSCInt COMPLETED_YES = 1;
};
```

インクルードファイル

```
#include <tscexcept.h>
```

メソッド

● TSCInt getErrorCode()

項目	型・意味
戻り値	エラーコード

障害のエラーコードを返します。

● TSCInt getDetailCode()

項目	型・意味
戻り値	内容コード

障害の内容コードを返します。

● TSCInt getPlaceCode()

項目	型・意味
戻り値	場所コード

障害の場所コードを返します。

● TSCInt getCompletionStatus()

項目	型・意味
戻り値	完了状態

障害発生時のメソッド呼び出しの完了状態を返します。

● TSCInt getMaintenanceCode1()

項目	型・意味
戻り値	保守コード 1

障害の保守コード 1 を返します。

● TSCInt getMaintenanceCode2()

項目	型・意味
戻り値	保守コード 2

障害の保守コード 2 を返します。

● TSCInt getMaintenanceCode3()

項目	型・意味
戻り値	保守コード 3

障害の保守コード 3 を返します。

● TSCInt getMaintenanceCode4()

項目	型・意味
戻り値	保守コード 4

障害の保守コード 4 を返します。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCSystemException クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
getErrorCode	できます。
getDetailCode	できます。
getPlaceCode	できます。
getCompletionStatus	できます。
getMaintenanceCode1	できます。
getMaintenanceCode2	できます。
getMaintenanceCode3	できます。
getMaintenanceCode4	できます。

TSCSystemException の派生クラス（C++）

TSCSystemException の派生クラスはシステム提供例外クラスです。

次に TSCSystemException の特徴を示します。

- 各派生クラスはエラーコードと 1 対 1 に対応します。
- 各例外クラス間でも，内容コードの一意性は保証されます。
- TSC システム例外を生成する場合，TSCSystemException の派生クラスのインスタンスを生成します。
- 種類別に例外を catch することで，種類ごとに違った例外処理ができます。

各種例外クラス

各種例外クラスの一覧を，アルファベット順で次の表に示します。

表 3-9　OTM のシステム例外（C++）

例外名	説明
TSCBadContextException	コンテキストオブジェクトの処理中に障害が発生しました。
TSCBadInvOrderException	ルーチン呼び出しの順番が不正です。
TSCBadOperationException	オペレーションが無効です。
TSCBadParamException	無効パラメタが渡されました。
TSCBadTypecodeException	タイプコードが不正です。
TSCCommFailureException	通信障害が発生しました。
TSCDataConversionException	データ変換に失敗しました。
TSCFreeMemException	メモリの解放に失敗しました。
TSCImpLimitException	実装の制限を超えました。
TSCInitializeException	ORB 初期化障害が発生しました。
TSCInternalException	ORB 内部エラーが発生しました。
TSCIntfReposException	インタフェースリポジトリへのアクセス中に障害が発生しました。
TSCInvFlagException	不正なフラグが指定されました。
TSCInvIdentException	識別子の構文が不正です。
TSCInvObjrefException	無効なオブジェクトリファレンスが指定されました。
TSCMarshalException	スタブ，スケルトンで CDR マーシャルに失敗しました。
TSCNoImplementException	オペレーションの実装が使用できません。
TSCNoMemoryException	動的メモリの割り当て障害が発生しました。
TSCNoPermissionException	許可されていないオペレーションを実行しようとしました。
TSCNoResourcesException	リクエストを処理するための資源が不足しています。
TSCNoResponseException	リクエストに対する応答がありません。

例外名	説明
TSCObjAdapterException	オブジェクトアダプタが障害を検出しました。
TSCObjectNotExistException	該当するオブジェクトがありません。
TSCPersistStoreException	パーシステントストレージに障害が発生しました。
TSCTransientException	一時的な障害が発生しました。
TSCUnknownException	未知の例外が発生しました。

内容コード

ユーザが TSCSystemException の派生クラスのインスタンスを生成して throw する場合，ユーザアプリケーション用に割り当てられている値を内容コードに設定してください。

OTM の内容コードの分類を次の表に示します。

表 3-10　OTM の内容コードの分類（C++）

分類	内容コードの範囲
ユーザアプリケーション用	0 ～ 999
OTM システム予約	1000 ～ 30000

個々の内容コードの詳細については，「付録 D　内容コード一覧」を参照してください。

場所コード

場所コードは障害が発生した場所を示します。ユーザが TSCSystemException の派生クラスのインスタンスを生成して throw する場合，PLACE_CODE_USER_AP を設定してください。

OTM の場所コードの一覧を次の表に示します。

表 3-11　OTM の場所コード（C++）

場所コード	場所
PLACE_CODE_USER_AP	ユーザアプリケーション
PLACE_CODE_SERV	OTM のサーバ機能部分
PLACE_CODE_DAEMON	TSC デーモン
PLACE_CODE_CLNT	OTM のクライアント機能部分
PLACE_CODE_CLNT_REG	TSC レギュレータ
PLACE_CODE_STUB	スタブ
PLACE_CODE_SKELTON	スケルトン
PLACE_CODE_ORBGW	TSCORB コネクタ

完了状態

完了状態は，障害が発生したときにメソッド呼び出しが完了しているかどうかを示します。

OTM の完了状態の一覧を次の表に示します。

表 3-12　OTM の完了状態（C++）

完了状態	意味
COMPLETED_NO	メソッド呼び出しが完了していません。
COMPLETED_MAYBE	メソッド呼び出しの完了状態を決定できません。
COMPLETED_YES	メソッド呼び出しが完了しています。

形式

```
class TSCBadContextException
  : public TSCSystemException
{
public:
  TSCBadContextException(TSCInt detail_code,
                         TSCInt place_code,
                         TSCInt completion_status,
                         TSCInt maintenance_code1,
                         TSCInt maintenance_code2,
                         TSCInt maintenance_code3,
                         TSCInt maintenance_code4);
};

class TSCBadInvOrderException
  : public TSCSystemException
{
public:
  TSCBadInvOrderException(TSCInt detail_code,
                          TSCInt place_code,
                          TSCInt completion_status,
                          TSCInt maintenance_code1,
                          TSCInt maintenance_code2,
                          TSCInt maintenance_code3,
                          TSCInt maintenance_code4);
};

class TSCBadOperationException
  : public TSCSystemException
{
public:
  TSCBadOperationException(TSCInt detail_code,
                           TSCInt place_code,
                           TSCInt completion_status,
                           TSCInt maintenance_code1,
                           TSCInt maintenance_code2,
                           TSCInt maintenance_code3,
                           TSCInt maintenance_code4);
```

```
};

class TSCBadParamException
  : public TSCSystemException
{
public:
  TSCBadParamException(TSCInt detail_code,
                       TSCInt place_code,
                       TSCInt completion_status,
                       TSCInt maintenance_code1,
                       TSCInt maintenance_code2,
                       TSCInt maintenance_code3,
                       TSCInt maintenance_code4);
};

class TSCBadTypecodeException
  : public TSCSystemException
{
public:
  TSCBadTypecodeException(TSCInt detail_code,
                          TSCInt place_code,
                          TSCInt completion_status,
                          TSCInt maintenance_code1,
                          TSCInt maintenance_code2,
                          TSCInt maintenance_code3,
                          TSCInt maintenance_code4);
};

class TSCCommFailureException
  : public TSCSystemException
{
public:
  TSCCommFailureException(TSCInt detail_code,
                          TSCInt place_code,
                          TSCInt completion_status,
                          TSCInt maintenance_code1,
                          TSCInt maintenance_code2,
                          TSCInt maintenance_code3,
                          TSCInt maintenance_code4);
};

class TSCDataConversionException
  :public TSCSystemException
{
public:
  TSCDataConversionException(TSCInt detail_code,
                             TSCInt place_code,
                             TSCInt completion_status,
                             TSCInt maintenance_code1,
                             TSCInt maintenance_code2,
                             TSCInt maintenance_code3,
                             TSCInt maintenance_code4);
};

class TSCFreeMemException
  : public TSCSystemException
{
public:
```

```
  TSCFreeMemException(TSCInt detail_code,
                      TSCInt place_code,
                      TSCInt completion_status,
                      TSCInt maintenance_code1,
                      TSCInt maintenance_code2,
                      TSCInt maintenance_code3,
                      TSCInt maintenance_code4);
};

class TSCImpLimitException
  : public TSCSystemException
{
public:
  TSCImpLimitException(TSCInt detail_code,
                       TSCInt place_code,
                       TSCInt completion_status,
                       TSCInt maintenance_code1,
                       TSCInt maintenance_code2,
                       TSCInt maintenance_code3,
                       TSCInt maintenance_code4);
};

class TSCInitializeException
  : public TSCSystemException
{
public:
  TSCInitializeException(TSCInt detail_code,
                         TSCInt place_code,
                         TSCInt completion_status,
                         TSCInt maintenance_code1,
                         TSCInt maintenance_code2,
                         TSCInt maintenance_code3,
                         TSCInt maintenance_code4);
};

class TSCInternalException
  : public TSCSystemException
{
public:
  TSCInternalException(TSCInt detail_code,
                       TSCInt place_code,
                       TSCInt completion_status,
                       TSCInt maintenance_code1,
                       TSCInt maintenance_code2,
                       TSCInt maintenance_code3,
                       TSCInt maintenance_code4);
};

class TSCIntfReposException
  : public TSCSystemException
{
public:
  TSCIntfReposException(TSCInt detail_code,
                        TSCInt place_code,
                        TSCInt completion_status,
                        TSCInt maintenance_code1,
                        TSCInt maintenance_code2,
                        TSCInt maintenance_code3,
                        TSCInt maintenance_code4);
```

```
};

class TSCInvFlagException
  : public TSCSystemException
{
public:
  TSCInvFlagException(TSCInt detail_code,
                      TSCInt place_code,
                      TSCInt completion_status,
                      TSCInt maintenance_code1,
                      TSCInt maintenance_code2,
                      TSCInt maintenance_code3,
                      TSCInt maintenance_code4);
};

class TSCInvIdentException
  : public TSCSystemException
{
public:
  TSCInvIdentException(TSCInt detail_code,
                       TSCInt place_code,
                       TSCInt completion_status,
                       TSCInt maintenance_code1,
                       TSCInt maintenance_code2,
                       TSCInt maintenance_code3,
                       TSCInt maintenance_code4);
};

class TSCInvObjrefException
  : public TSCSystemException
{
public:
  TSCInvObjrefException(TSCInt detail_code,
                        TSCInt place_code,
                        TSCInt completion_status,
                        TSCInt maintenance_code1,
                        TSCInt maintenance_code2,
                        TSCInt maintenance_code3,
                        TSCInt maintenance_code4);
};

class TSCMarshalException
  : public TSCSystemException
{
public:
  TSCMarshalException(TSCInt detail_code,
                      TSCInt place_code,
                      TSCInt completion_status,
                      TSCInt maintenance_code1,
                      TSCInt maintenance_code2,
                      TSCInt maintenance_code3,
                      TSCInt maintenance_code4);
};

class TSCNoImplementException
  : public TSCSystemException
{
public:
```

```
  TSCNoImplementException(TSCInt detail_code,
                          TSCInt place_code,
                          TSCInt completion_status,
                          TSCInt maintenance_code1,
                          TSCInt maintenance_code2,
                          TSCInt maintenance_code3,
                          TSCInt maintenance_code4);
};

class TSCNoMemoryException
  : public TSCSystemException
{
public:
  TSCNoMemoryException(TSCInt detail_code,
                       TSCInt place_code,
                       TSCInt completion_status,
                       TSCInt maintenance_code1,
                       TSCInt maintenance_code2,
                       TSCInt maintenance_code3,
                       TSCInt maintenance_code4);
};

class TSCNoPermissionException
  : public TSCSystemException
{
public:
  TSCNoPermissionException(TSCInt detail_code,
                           TSCInt place_code,
                           TSCInt completion_status,
                           TSCInt maintenance_code1,
                           TSCInt maintenance_code2,
                           TSCInt maintenance_code3,
                           TSCInt maintenance_code4);
};

class TSCNoResourcesException
  : public TSCSystemException
{
public:
  TSCNoResourcesException(TSCInt detail_code,
                          TSCInt place_code,
                          TSCInt completion_status,
                          TSCInt maintenance_code1,
                          TSCInt maintenance_code2,
                          TSCInt maintenance_code3,
                          TSCInt maintenance_code4);
};

class TSCNoResponseException
  : public TSCSystemException
{
public:
  TSCNoResponseException(TSCInt detail_code,
                         TSCInt place_code,
                         TSCInt completion_status,
                         TSCInt maintenance_code1,
                         TSCInt maintenance_code2,
                         TSCInt maintenance_code3,
```

```
                          TSCInt maintenance_code4);
};

class TSCObjAdapterException
  : public TSCSystemException
{
public:
  TSCObjAdapterException(TSCInt detail_code,
                         TSCInt place_code,
                         TSCInt completion_status,
                         TSCInt maintenance_code1,
                         TSCInt maintenance_code2,
                         TSCInt maintenance_code3,
                         TSCInt maintenance_code4);
};

class TSCObjectNotExistException
  : public TSCSystemException
{
public:
   TSCObjectNotExistException(TSCInt detail_code,
                              TSCInt place_code,
                              TSCInt completion_status,
                              TSCInt maintenance_code1,
                              TSCInt maintenance_code2,
                              TSCInt maintenance_code3,
                              TSCInt maintenance_code4);
};

class TSCPersistStoreException
  : public CORBA::PERSIST_STORE, public TSCSystemException
{
public:
  TSCPersistStoreException(TSCInt detail_code,
                           TSCInt place_code,
                           TSCInt completion_status,
                           TSCInt maintenance_code1,
                           TSCInt maintenance_code2,
                           TSCInt maintenance_code3,
                           TSCInt maintenance_code4);
};

class TSCTransientException
  :public TSCSystemException
{
public:
  TSCTransientException(TSCInt detail_code,
                        TSCInt place_code,
                        TSCInt completion_status,
                        TSCInt maintenance_code1,
                        TSCInt maintenance_code2,
                        TSCInt maintenance_code3,
                        TSCInt maintenance_code4);
};

class TSCUnknownException
  : public TSCSystemException
{
```

```
public:
  TSCUnknownException(TSCInt detail_code,
                      TSCInt place_code,
                      TSCInt completion_status,
                      TSCInt maintenance_code1,
                      TSCInt maintenance_code2,
                      TSCInt maintenance_code3,
                      TSCInt maintenance_code4);
};
```

インクルードファイル

```
#include <tscexcept.h>
```

TSCThread（C++）

TSCThread はシステム提供クラスです。

TSCThread は，OTM が管理するスレッドに対応するオブジェクトのインタフェースです。

ユーザは，TSCThread を継承させて，必要な情報を保存するクラスをスレッド単位に定義します。スレッドに割り付けられるオブジェクトとして，派生クラスのインスタンスを生成します。次に TSCThread の特徴を示します。

- ユーザは TSCThread クラスを継承して，実装クラスを記述する必要があります。
- TSCThread を生成する TSCThreadFactory の実装クラスを記述する必要があります。

形式

```
class TSCThread
{
 public:
  TSCThread();
  virtual ~TSCThread();
};
```

インクルードファイル

```
#include <tscobject.h>
```

コンストラクタ

● TSCThread()

TSCThread を生成します。

デストラクタ

● ~TSCThread()

TSCThread を削除します。

TSCThreadFactory（C++）

TSCThreadFactory はシステム提供クラスです。

TSCThreadFactory は，TSCThread を生成または削除するメソッドを持つオブジェクトのインタフェースです。

ユーザは TSCThreadFactory を継承させて，TSC ユーザスレッドを生成および削除するクラスを定義し，TSC ユーザスレッドのファクトリとして派生クラスのインスタンスを生成します。次に TSCThreadFactory の特徴を示します。

- 直接，TSCThreadFactory クラスのインスタンスを生成できません。
- ユーザは TSCThreadFactory クラスを継承して，実装クラスを記述する必要があります。

OTM からの呼び出しによる TSCThread の管理

OTM は，TSCThreadFactory の create を呼び出すことで，返される TSCThread をそのスレッドに割り当ててスレッドを生成します。逆に，そのスレッドに割り当てた TSCThread を引数に destroy を呼び出すことで，生成したスレッドを削除します。

形式

```
class TSCThreadFactory;
typedef TSCThreadFactory* TSCThreadFactory_ptr;

class TSCThreadFactory
{
 public:
  TSCThreadFactory();
  virtual ~TSCThreadFactory();

  virtual TSCThread_ptr create() = 0;
  virtual void destroy(TSCThread_ptr tsc_thr) = 0;
};
```

インクルードファイル

```
#include <tscobject.h>
```

コンストラクタ

● TSCThreadFactory()

TSCThreadFactory を生成します。

デストラクタ

● virtual ~TSCThreadFactory()

TSCThreadFactory を削除します。

コールバックメソッド

● virtual TSCThread_ptr create()

項目	型・意味
戻り値	管理される TSCThread
例外	各種 TSCSystemException

TSCThread を返します。TSC ユーザスレッドを生成するコードを記述できます。OTM がこのメソッドを呼び出した結果，返された TSCThread が管理対象となります。

管理対象とする TSCThread のメモリ領域の管理責任は OTM にあるので，ユーザは削除しないでください。なお，OTM は，複数のスレッド上から同時にこのメソッドを呼び出すので，ユーザはマルチスレッド環境に対応するリエントラントなコードを記述する必要があります。

また，create 呼び出しに失敗した場合は，各種の TSCSystemException によって通知する形でコードを記述してください。

● virtual destroy(TSCThread_ptr tsc_thr)

項目	型・意味	
引数	TSCThread_ptr tsc_thr	管理対象から外す TSCThread
例外	ありません。	

TSCThread を消去する前の処理のコードを記述できます。OTM が TSC ユーザスレッドを管理対象から外すとき，該当する TSCThread を引数に指定して，このメソッドを呼び出します。

管理対象から外された TSCThread のメモリ管理責任はユーザにあります。ユーザは，必要に応じて削除してください。なお，OTM は，複数のスレッド上から同時にこのメソッドを呼び出すので，ユーザはマルチスレッド環境に対応するリエントラントなコードを記述する必要があります。

また，このメソッドから通知した例外は無視されます。

TSCThreadFactory の派生クラスの生成と削除

TSCThreadFactory の派生クラスは，new オペレータで生成し，delete オペレータで削除します。OTM が TSCThread の公開メソッドを呼び出しているとき，またはユーザが TSCThread の公開メソッドを呼び出しているときは削除できないため，公開メソッドを呼び出していない状態で削除してください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCThreadFactory クラス型のインスタンスのメソッドを呼び

出す規則を次に示します。なお，これらのメソッドは，OTM が呼び出します。

メソッド	複数のスレッド上からの同時呼び出し
create	できます。
destroy	できます。

TSCWatchTime（C++）

TSCWatchTime はシステム提供クラスです。

TSCWatchTime は，start() が発行されてから stop() が発行されるまでの時間を監視するクラスです。時間監視が終了する前に指定された監視時間が経過すると，エラーメッセージを出力してプロセスを異常終了します。この機能は，サーバアプリケーションの initServer() 発行後から endServer() を発行するまでの間有効です。

形式

```
class TSCWatchTime
{
public:

  TSCWatchTime();

  TSCWatchTime(TSCInt watch_time);

  ~TSCWatchTime();

  void start();
  void stop();
  void reset();
};
```

インクルードファイル

```
#include <tscadm.h>
```

コンストラクタ

● TSCWatchTime()

項目	型・意味
例外	TSCBadInvOrderException TSCInternalException TSCNoMemoryException

TSCWatchTime を生成します。

サーバアプリケーションの開始時にコマンドオプション引数 -TSCWatchTime で指定した監視時間（秒）が適用されます。

● TSCWatchTime(TSCInt watch_time)

項目	型・意味（単位）	
引数	TSCInt watch_time	監視時間（秒）
例外	TSCBadInvOrderException TSCBadParamException TSCInternalException TSCNoMemoryException	

TSCWatchTime を生成します。

引数に "0" を指定した場合，サーバアプリケーションの開始時にコマンドオプション引数 -TSCWatchTime で指定した監視時間（秒）が適用されます。

デストラクタ

● ~TSCWatchTime()

TSCWatchTime を削除します。

メソッド

● void start()

項目	型・意味（単位）
例外	TSCInternalException TSCNoPermissionException

時間監視を開始します。または，stop() で中断した時間監視を再開します。

● void stop()

項目	型・意味（単位）
例外	TSCInternalException TSCNoPermissionException

時間監視を中断します。start() を発行したスレッドと異なるスレッドでは，stop() を発行できません。

● void reset()

項目	型・意味（単位）
例外	TSCInternalException TSCNoPermissionException

監視時間をこのクラスの作成時に指定した値に戻します。start() の発行以降 stop() の発行までの間は発行できません。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCWatchTime クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
start	できます。
stop	できます。※
reset	できます。

注※

必ず start() と同一のスレッドで発行してください。

注意事項

- IDL ファイルに記述したユーザメソッドの呼び出しからリターンまでの実行時間を監視する場合には，TSCWatchTime クラスではなく，サーバアプリケーションの開始時に指定するコマンドオプション引数 -TSCWatchMethod を使用してください。
- TSCAdm::initServer() の発行前，または TSCAdm::endServer() の発行後には，TSCWatchTime クラスのインスタンスを生成できません。
- stop() を発行する前にデストラクタが発行されると，時間監視を終了します。
- start() を発行したスレッドと異なるスレッドでは，stop() を呼び出せません。
- stop() を発行したあとに start() によって処理を再開する場合には，前回経過した時間を監視時間から差し引いて処理をします。
  次に例を示します。

```
TSCWatchTime *wt = new TSCWatchTime(180);
    :
wt->start();
    :                       //(1) 60秒経過
wt->stop();
    :                       //(2)
wt->start();
```

例えば，180 秒の監視時間を設定してクラスを生成した場合に，(1) で示す範囲で 60 秒が経過すると，stop() の発行後，次の start() から stop() までの監視時間は 120 秒になります。180 秒の時間監視を設定したい場合には (2) で示す範囲で reset() を発行してください。その場合，再発行した start() から stop() までの監視時間は 180 秒になります。

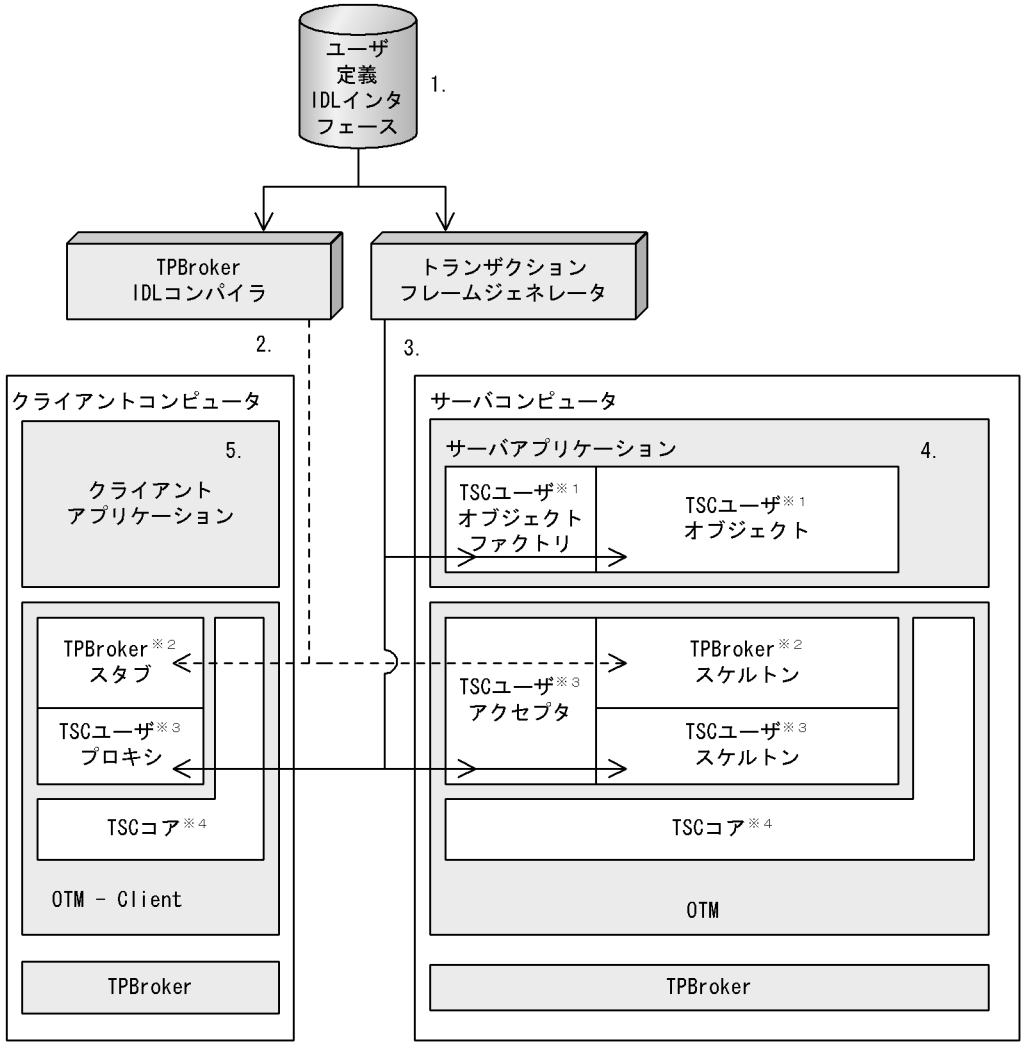
4 アプリケーションプログラムの作成（Java）

この章では，Java でアプリケーションプログラムを作成する方法について説明します。

4.1　アプリケーションプログラムの作成手順（Java）

Java でアプリケーションプログラムを作成する手順を次の図に示します。

図 4-1　アプリケーションプログラムを作成する手順（Java）



注※ 1

トランザクションフレームジェネレータが生成するクラスの雛形（**雛形クラス**）です。クラスのシグネチャ，およびコードの一部はすでに記述されています。自動生成されたままの状態では利用できないクラスなので，ユーザはコードを追加する必要があります。

注※ 2

TPBroker の IDL コンパイラが生成するクラス（TPBroker **クラス**）です。

注※ 3

トランザクションフレームジェネレータが生成する基底クラス，およびそのまま使用できるクラス（**ユーザ定義** IDL **インタフェース依存クラス**）です。TPBroker スタブクラスまたは TPBroker スケルトンクラスと継承関係にあります。

注※ 4

システム提供クラスです。OTM の機能の中心となるクラスです。

図中の 1. ～ 5. の手順は次のとおりです。

1. **ユーザ定義** IDL **インタフェースを定義します。**
2. TPBroker **の** IDL **コンパイラを使用して** TPBroker **クラスを生成します。**
   Cosminexus TPBroker for Java の ORB Version 4 を使用する場合は，-boa および -no_narrow_compliance オプションを指定して実行してください。
3. **トランザクションフレームジェネレータを使用してユーザ定義** IDL **インタフェース依存クラスおよび雛形クラスを生成します。**
   Cosminexus TPBroker for Java の ORB Version 4 を使用する場合は，-vbj4 オプションを指定して実行してください。
4. **次に示すコードを記述してサーバアプリケーションを作成します。**
   - TSC ユーザオブジェクト（雛形クラスの編集）
   - TSC ユーザオブジェクトファクトリ（雛形クラスの編集）
   - サービス登録処理
5. **次に示すコードを記述してクライアントアプリケーションを作成します。**
   - サービス利用処理
   - TSC ユーザプロキシ呼び出し

次に，図中のオブジェクトについて説明します。

● TPBroker **スタブ**

TPBroker の IDL コンパイラが生成するスタブのオブジェクトです。

● TSC **ユーザプロキシ**

クライアントアプリケーションが TSC ユーザオブジェクトを呼び出すための代理オブジェクトです。ユーザ定義 IDL インタフェース依存のオブジェクトです。

● TSC **ユーザオブジェクトファクトリ**

サーバアプリケーションで，TSC ユーザオブジェクトを生成するオブジェクトです。実装についてはユーザが記述します。雛形のオブジェクトです。

● TSC **ユーザオブジェクト**

サーバアプリケーションで，サービスを提供するオブジェクトです。このオブジェク

トは，TSC ユーザスケルトンを継承し，実装についてはユーザが記述します。雛形のオブジェクトです。

● **TSC ユーザアクセプタ**

TSC ユーザプロキシから呼び出し要求を受けて，TSC ユーザオブジェクトにリクエストを配送するオブジェクトです。ユーザ定義 IDL インタフェース依存のオブジェクトです。

● **TPBroker スケルトン**

TPBroker の IDL コンパイラが生成するスケルトンのオブジェクトです。

● **TSC ユーザスケルトン**

TSC ユーザオブジェクトのスケルトンです。ユーザ定義 IDL インタフェース依存のオブジェクトです。

4.2 同期型呼び出しをするアプリケーションプログラム（Java）

同期型呼び出しをするアプリケーションプログラムの Java での作成例を示します。これらのサンプルコードは製品で提供されています。

● ユーザ定義 IDL インタフェースの例

ユーザ定義 IDL インタフェースの例を次に示します。

```
//
// "ABCfile.idl"
//
typedef sequence<octet> OctetSeq;

interface ABC
{
  void call(in OctetSeq in_data, out OctetSeq out_data);
};
```

● IDL コンパイラが生成するクラス

TPBroker の IDL コンパイラはユーザ定義 IDL インタフェースから次のファイルを生成します。

- ABC.java
- ABCHelper.java
- ABCHolder.java
- ABCOperations.java
- OctetSeqHelper.java
- OctetSeqHolder.java
- _ABCImplBase.java
- _example_ABC.java ※1
- _st_ABC.java ※2
- _tie_ABC.java
- ABCPOA.java ※3
- ABCPOATie.java ※3
- _ABCStub.java ※3

注※ 1

Cosminexus TPBroker for Java の ORB Version 4 を使用する場合は，デフォルトでは生成されません。

注※ 2

Cosminexus TPBroker for Java の ORB Version 4 を使用する場合は生成されません。

注※ 3

Cosminexus TPBroker for Java の ORB Version 4 を使用する場合だけ生成されます。

● トランザクションフレームジェネレータが生成するクラス

OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。

表 4-1 同期型呼び出しをするアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（Java）

分類	生成するクラス名（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• ABC_TSCprxy（TSC ユーザプロキシ）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• ABC_TSCsk（TSC ユーザスケルトン） • ABC_TSCacpt（TSC ユーザアクセプタ）
雛形クラス	• ABC_TSCimpl（TSC ユーザオブジェクト） • ABC_TSCfactimpl（TSC ユーザオブジェクトファクトリ）

4.2.1 同期型呼び出しをするクライアントアプリケーションの例（Java）

同期型呼び出しをするクライアントアプリケーションの処理の流れとコードの例を示します。

（1） サービス利用処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
6. TSC **デーモンへの接続解放**
7. TPBroker OTM **の終了処理**

（2） サービス利用処理のコード

```
//
// "ClientAP.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;
```

```
public class ClientAP
{

  public static void main(String[] args)
  {
    ////////
    // 1, TPBrokerの初期化処理
    ////////
    org.omg.CORBA.ORB orb = null;
    try
    {
      // ORBの初期化
      orb = org.omg.CORBA.ORB.init(args, null);
    }
    catch(org.omg.CORBA.SystemException ce)
    {
      // 例外処理
      System.out.println(ce);
      System.exit(1);
    }

    ////////
    // 2, TPBroker OTMの初期化処理
    ////////
    // TSCの初期化
    try
    {
      TSCAdm.initClient(args, null, orb);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      System.exit(1);
    }

    ////////
    // 3, TSCデーモンへの接続
    ////////
    TSCDomain domain = null;
    try
    {
      domain = new TSCDomain(null, null);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      try
      {
        TSCAdm.endClient();
      }
      catch(TSCSystemException se)
      {
        System.exit(1);
      }
        System.exit(1);
```

```
}

TSCClient tsc_client = null;
try
{
  tsc_client =
    TSCAdm.getTSCClient(domain, TSCAdm.Regulator);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 4, TSCユーザプロキシの生成および各種設定
////////
// ユーザ定義IDLインタフェース"ABC"用のTSCProxy生成
ABC_TSCprxy my_proxy = null;
try
{
  my_proxy = new ABC_TSCprxy(tsc_client);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCClient(tsc_client);
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 5, TSCユーザプロキシのメソッド呼び出し
//     （サーバ側のオブジェクトの呼び出し）
////////
try
{
  callService.invoke(my_proxy);
}
catch(TSCSystemException tsc_se)
```

```
    {
      // 例外処理
      System.out.println(tsc_se);
      try
      {
        TSCAdm.releaseTSCClient(tsc_client);
        TSCAdm.endClient();
      }
      catch(TSCSystemException se)
      {
        System.exit(1);
      }
      System.exit(1);
    }

    ////////
    // 6, TSCデーモンへの接続解放
    ////////
    try
    {
      TSCAdm.releaseTSCClient(tsc_client);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      try
      {
        TSCAdm.endClient();
      }
      catch(TSCSystemException se)
      {
        System.exit(1);
      }
      System.exit(1);
    }

    ////////
    // 7, TPBroker OTMの終了処理
    ////////
    try
    {
      TSCAdm.endClient();
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      System.exit(1);
    }

    System.exit(0);

  }

}
```

（3） TSC ユーザプロキシを呼び出すコード

```
//
// "callService.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public
class callService
{
  public static void
  invoke(ABC_TSCprxy my_proxy)
  {

    //////
    //サービスの呼び出し
    //////
    //in引数の準備
    byte[] user_in = new byte[4];

    //out引数の準備
    OctetSeqHolder user_out = new OctetSeqHolder();

    try
    {
      //サーバのメソッド呼び出し
      my_proxy.call(user_in, user_out);
    }
    catch(TSCSystemException tsc_se)
    {
      //例外処理
      System.out.println(tsc_se);
      throw tsc_se;
    }
  }

}
```

4.2.2 同期型呼び出しをするサーバアプリケーションの例（Java）

同期型呼び出しをするサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス ABC_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス ABC_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。

（1） TSC ユーザオブジェクト（ABC_TSCimpl）のコード

```
//
// "ABC_TSCimpl.java"
//
```

```
import OctetSeqHelper;
import OctetSeqHolder;

// import classes used in this implementation, if necessary.
import java.lang.System;

public class ABC_TSCimpl extends ABC_TSCsk
{
  // Write class variables, if necessary

  public ABC_TSCimpl()
  {
    // Constructor of implementation.
    // Write user own code.
    //ユーザオブジェクトのコンストラクタのコードを記述します。
    //引数の数および型を変更できます。
    super();
  };

  public void call(byte[] in_data, OctetSeqHolder out_data)

  {
    // Operation "call".
    // Write user own code.
    //ユーザメソッドのコードを記述します。

    //メソッドが呼ばれた回数を増加させます。
    //（このメソッドの処理は引数の値と無関係です）
    m_counter++;
    out_data.value = new byte[4];

    System.out.println("Call method in ABC_TSCprxy");

  };

  //メソッドが呼ばれた回数
  protected int m_counter = 0;

};
```

（2）TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のコード

```
//
// "ABC_TSCfactimpl.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.TSCObject;
import JP.co.Hitachi.soft.TPBroker.TSC.TSCObjectFactory;

import OctetSeqHelper;
import OctetSeqHolder;

// import classes used in this implementation, if necessary.

public class ABC_TSCfactimpl
    implements TSCObjectFactory
{
  public ABC_TSCfactimpl()
```

```
  {
    // Constructor of implementation.
    // Write user own code.
    //TSCユーザオブジェクトファクトリのコードを記述します。
    //引数の数および型を変更することもできます。
  };

  public TSCObject create()
  {
    // Method to create user object.
    // Write user own code.
    //サーバオブジェクトを生成するコードを記述します。
    //必要に応じて変更してください。
    return new ABC_TSCimpl();
  };

  public void destroy(TSCObject tsc_obj)
  {
    // Method to destroy user object.
    // Write user own code.
    //後処理のコードを記述します。
    //必要に応じて変更してください。
  };

};
```

(3) サービス登録処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザオブジェクトファクトリ，**TSC **ユーザアクセプタの生成（**new**），および各種設定**
5. TSC **ルートアクセプタの生成および各種設定**
6. TSC **ルートアクセプタの活性化**
7. **実行制御の受け渡し**
8. TSC **ルートアクセプタの非活性化**
9. TSC **デーモンへの接続解放**
10. TPBroker OTM **の終了処理**

(4) サービス登録処理のコード

```
//
//  "ServerAP.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;
```

```
public class ServerAP
{

  public static void main(String[] args)
  {

    ////////
    // 1, TPBrokerの初期化処理
    ////////

    org.omg.CORBA.ORB orb = null;
    try
    {
      // ORBの初期化
      orb = org.omg.CORBA.ORB.init(args, null);
    }
    catch(org.omg.CORBA.SystemException ce)
    {
      // 例外処理
      System.out.println(ce);
      System.exit(1);
    }

    ////////
    // 2, TPBroker OTMの初期化処理
    ////////
    try
    {
      // TSCの初期化
      TSCAdm.initServer(args, orb);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      System.exit(1);
    }

    ////////
    // 3, TSCデーモンへの接続
    ////////
    TSCDomain domain = null;
    try
    {
      domain = new TSCDomain(null, null);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      try
      {
        TSCAdm.endServer();
      }
      catch(TSCSystemException se)
      {
        System.exit(1);
      }
```

```
  System.exit(1);
}

TSCServer tsc_server = null;
try
{
  // TSCデーモンの参照オブジェクトを取得
  tsc_server = TSCAdm.getTSCServer(domain);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 4, TSCユーザオブジェクトファクトリ, TSCユーザアクセプタの
//    生成（new）および各種設定
////////
// ABC_TSCfactimplの生成
TSCObjectFactory my_fact = new ABC_TSCfactimpl();

// TSCAcceptorの生成
TSCAcceptor my_acpt = null;
try
{
  my_acpt = new ABC_TSCacpt(my_fact);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 5, TSCルートアクセプタの生成および各種設定
////////

// TSCRootAcceptorの生成
```

```
TSCRootAcceptor my_rt_acpt = null;
try
{
  my_rt_acpt = TSCRootAcceptor.create(tsc_server);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

try
{
  // TSCRootAcceptorへの登録
  my_rt_acpt.registerAcceptor(my_acpt);

  // TSCRootAcceptorのパラレルカウント指定もできます。
  // デフォルト値は1
  // オプション引数でデフォルト値を変更することもできます。
  // my_rt_acpt.setParallelCount(5);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 6, TSCルートアクセプタの活性化
////////
try
{
  // オブジェクトの活性化
  my_rt_acpt.activate("serviceX");
}
catch(TSCSystemException tsc_se)
{
```

```
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 7, 実行制御の受け渡し
////////
try
{
  TSCAdm.serverMainloop();
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    my_rt_acpt.deactivate();
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 8, TSCルートアクセプタの非活性化
////////
try
{
  // オブジェクトの非活性化
  my_rt_acpt.deactivate();
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
```

```
            System.exit(1);
          }
          System.exit(1);
        }

        ////////
        // 9, TSCデーモンへの接続解放
        ////////
        try
        {
          TSCAdm.releaseTSCServer(tsc_server);
        }
        catch(TSCSystemException tsc_se)
        {
          // 例外処理
          System.out.println(tsc_se);
          try
          {
            TSCAdm.endServer();
          }
          catch(TSCSystemException se)
          {
            System.exit(1);
          }
          System.exit(1);
        }

        ////////
        // 10, TPBroker OTMの終了処理
        ////////
        try
        {
          TSCAdm.endServer();
        }
        catch(TSCSystemException tsc_se)
        {
          // 例外処理
          System.out.println(tsc_se);
          System.exit(1);
        }

        System.exit(0);

      }
    }
```

4.2.3 同期型呼び出しをするアプリケーションプログラムの実行時の処理（Java）

同期型呼び出しをするアプリケーションプログラムを実行した場合の処理シーケンスを示します。

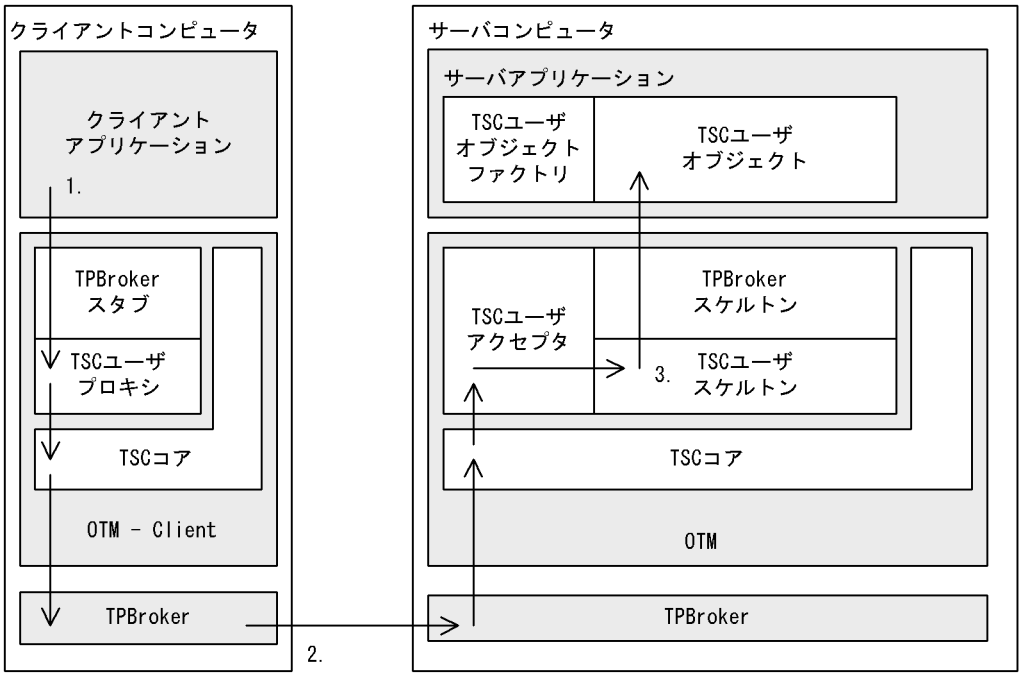
（1）クライアントアプリケーションの開始（Java）

クライアントアプリケーションを開始すると，サービスの利用処理が実行されます。

ABC_TSCprxy のメソッドを呼び出すと，ABC_TSCsk を経由し，ユーザが実装した ABC_TSCimpl のメソッドが呼び出されます。

クライアントアプリケーションの開始の流れを次の図に示します。

図 4-2　同期型呼び出しをするクライアントアプリケーションの開始（Java）



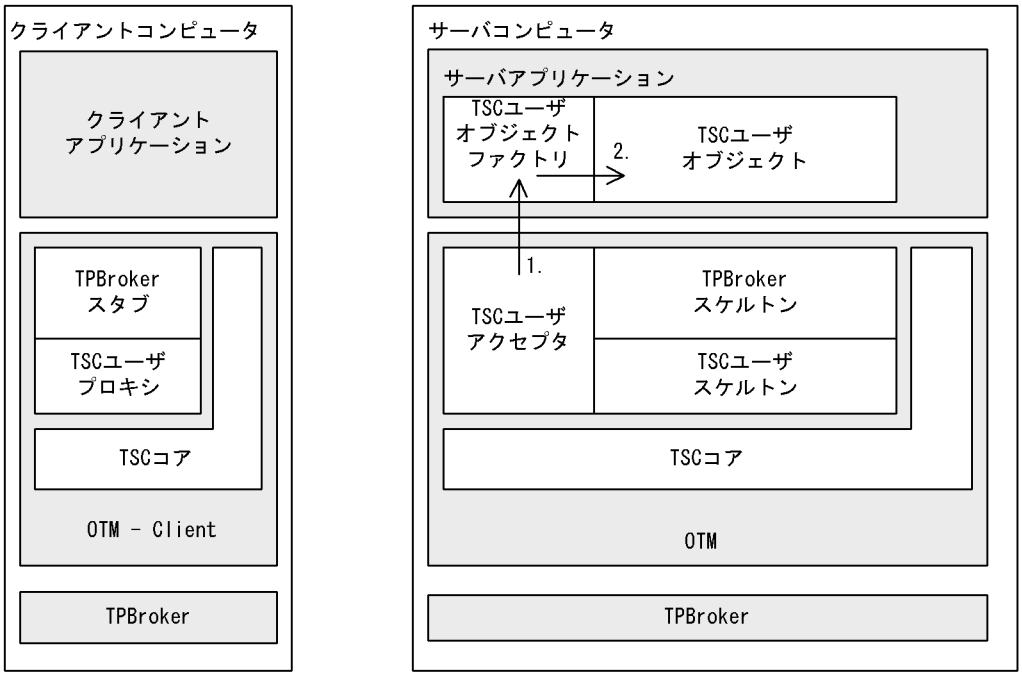
1. TSC **ユーザプロキシのメソッドが呼び出されます。**
2. **スケジューリングおよび** TPBroker **による通信が実行されます。**
3. TSC **ユーザオブジェクトのメソッドが呼び出されます。**

（2） サーバアプリケーションの開始（Java）

サーバアプリケーションを開始すると，サービスの登録処理が実行されます。TSCRootAcceptor の activate メソッドを呼び出すと，ABC_TSCfactimpl に ABC_TSCimpl の生成依頼が発行されます。

サーバアプリケーションの開始の流れを次の図に示します。

図 4-3　同期型呼び出しをするサーバアプリケーションの開始（Java）



1. TSC **ユーザオブジェクトの活性化処理が実行されます。**
2. TSC **ユーザオブジェクトのインプリメンテーションが生成されます。**

4.3 非応答型呼び出しをするアプリケーションプログラム（Java）

非応答型呼び出しをするアプリケーションプログラムの Java での作成例を示します。これらのサンプルコードは製品で提供されています。

● ユーザ定義 IDL インタフェースの例

ユーザ定義 IDL インタフェースの例を次に示します。

```
//
// "XYZfile.idl"
//

interface XYZ
{
  oneway void callOnly(in long in_data);
};
```

● IDL コンパイラが生成するクラス

TPBroker の IDL コンパイラはユーザ定義 IDL インタフェースから次のファイルを生成します。

- XYZ.java
- XYZHelper.java
- XYZHolder.java
- XYZOperations.java
- _XYZImplBase.java
- _example_XYZ.java ※ 1
- _st_XYZ.java ※ 2
- _tie_XYZ.java
- XYZPOA.java ※ 3
- XYZPOATie.java ※ 3
- _XYZStub.java ※ 3

注※ 1

Cosminexus TPBroker for Java の ORB Version 4 を使用する場合は，デフォルトでは生成されません。

注※ 2

Cosminexus TPBroker for Java の ORB Version 4 を使用する場合は生成されません。

注※ 3

Cosminexus TPBroker for Java の ORB Version 4 を使用する場合だけ生成されます。

● **トランザクションフレームジェネレータが生成するクラス**

OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。

表 4-2 非応答型呼び出しをするアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（Java）

分類	生成するクラス名（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• XYZ_TSCprxy（TSC ユーザプロキシ）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• XYZ_TSCsk（TSC ユーザスケルトン） • XYZ_TSCacpt（TSC ユーザアクセプタ）
雛形クラス	• XYZ_TSCimpl（TSC ユーザオブジェクト） • XYZ_TSCfactimpl（TSC ユーザオブジェクトファクトリ）

4.3.1 非応答型呼び出しをするクライアントアプリケーションの例（Java）

非応答型呼び出しをするクライアントアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。

（1） サービス利用処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
6. TSC **デーモンへの接続解放**
7. TPBroker OTM **の終了処理**

（2） サービス利用処理のコード

```
//
// "ClientAP.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public class ClientAP
{
  public static void main(String[] args)
  {
```

```
////////
// 1, TPBrokerの初期化処理
////////
org.omg.CORBA.ORB  orb = null;
try
{
  // ORBの初期化処理
  orb = org.omg.CORBA.ORB.init(args,null);
}
catch(org.omg.CORBA.SystemException ce)
{
  // 例外処理
  System.out.println(ce);
  System.exit(1);
}

////////
//2, TPBroker OTMの初期化処理
////////
// TSCの初期化
try
{
  TSCAdm.initClient(args,null,orb);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  System.exit(1);
}

////////
// 3, TSCデーモンへの接続
////////

TSCDomain domain = null;
try
{
  domain = new TSCDomain(null,null);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }

  System.exit(1);
}

TSCClient tsc_client = null;
```

```
try
{
  tsc_client =
    TSCAdm.getTSCClient(domain, TSCAdm.Regulator);
}
catch(TSCSystemException tsc_se)
{
  // Exception process
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 4, TSCユーザプロキシの生成および各種設定
////////
// ユーザ定義IDLインタフェース"XYZ"用のTSCProxy生成

XYZ_TSCprxy  my_proxy = null;
try
{
  my_proxy = new XYZ_TSCprxy(tsc_client);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCClient(tsc_client);
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
    System.exit(1);
}

//////
// 5, TSCユーザプロキシのメソッド呼び出し
//     （サーバ側のオブジェクトの呼び出し）
//////
try
{
  callOnlyService.invoke(my_proxy);
}
  catch(TSCSystemException tsc_se)
{
  System.out.println(tsc_se);
```

```
        try
        {
          TSCAdm.releaseTSCClient(tsc_client);
          TSCAdm.endClient();
        }
        catch(TSCSystemException se)
        {
          System.exit(1);
        }
        System.exit(1);

      }
      /////////
      // 6, TSCデーモンへの接続解放
      /////////
      try
      {
        TSCAdm.releaseTSCClient(tsc_client);
      }
      catch(TSCSystemException tsc_se)
      {
        // 例外処理
        System.out.println(tsc_se);
        try
        {
          TSCAdm.endClient();
        }
        catch(TSCSystemException se)
        {
          System.exit(1);
        }

        System.exit(1);
      }

      /////////
      //7, TPBroker OTMの終了処理
      /////////
      try
      {
        TSCAdm.endClient();
      }
      catch(TSCSystemException tsc_se)
      {
        // 例外処理
        System.out.println(tsc_se);
        System.exit(1);

      }

      System.exit(0);

    }
}
```

（3）TSC ユーザプロキシを呼び出すコード

```
//
// "callOnlyService.java"

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public
class callOnlyService
{
  public static void
  invoke(XYZ_TSCprxy my_proxy)
  {
    //////
    //サービスの呼び出し
    //////
    //in引数の準備
    int in_data = 0;

    try
    {
      //サーバのメソッド呼び出し
      my_proxy.callOnly(in_data);
    }
    catch(TSCSystemException tsc_se)
    {
      //例外処理
      System.out.println(tsc_se);
      throw tsc_se;
    }
  }

}
```

4.3.2 非応答型呼び出しをするサーバアプリケーションの例（Java）

非応答型呼び出しをするサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス XYZ_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス XYZ_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。

（1）TSC ユーザオブジェクト（XYZ_TSCimpl）のコード

```
//
//  "XYZ_TSCimpl.java"
//


// import classes used in this implementation, if necessary.
import java.lang.System;

public class XYZ_TSCimpl extends XYZ_TSCsk
```

```
{
  // Write class variables, if necessary

  public XYZ_TSCimpl()
  {
    // Constructor of implementation.
    // Write user own code.
    //TSCユーザオブジェクトのコンストラクタのコードを記述します。
    //引数の数および型を変更できます。
    super();
  };

  public void callOnly(int in_data)

  {
    // Operation "callOnly".
    // Write user own code.
    //ユーザメソッドのコードを記述します。

    //メソッドが呼ばれた回数を増加させます。
    //（このメソッドの処理は引数の値と無関係です）
    m_value += in_data;

    System.out.println("Call method in XYZ_TSCprxy");

  };

  //メソッドが呼ばれた回数
  protected int m_value;
};
```

（2） TSC ユーザオブジェクトファクトリ（XYZ_TSCfactimpl）のコード

```
//
// "XYZ_TSCfactimpl".java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.TSCObject;
import JP.co.Hitachi.soft.TPBroker.TSC.TSCObjectFactory;


// import classes used in this implementation, if necessary.

public class XYZ_TSCfactimpl
  implements TSCObjectFactory
{
  public XYZ_TSCfactimpl()
  {
    // Constructor of implementation.
    // Write user own code.
    //TSCユーザオブジェクトファクトリのコンストラクタのコードを記述します。
    //引数の数および型を変更できます。
  };

  public TSCObject create()
  {
    // Method to create user object.
    // Write user own code.
```

```
    //サーバオブジェクトを生成するコードを記述します。
    //必要に応じて修正してください。
    return new XYZ_TSCimpl();
  };

  public void destroy(TSCObject tsc_obj)
  {
    // Method to destroy user object.
    // Write user own code.
    //後処理のコードを記述します。
    //必要に応じて修正してください。
  };

};
```

（3） サービス登録処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザオブジェクトファクトリ，**TSC **ユーザアクセプタの生成（**new**），および各種設定**
5. TSC **ルートアクセプタの生成および各種設定**
6. TSC **ルートアクセプタの活性化**
7. **実行制御の受け渡し**
8. TSC **ルートアクセプタの非活性化**
9. TSC **デーモンへの接続解放**
10. TPBroker OTM **の終了処理**

（4） サービス登録処理のコード

```
//
// "ServerAP.Java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public class ServerAP
{

  public static void main(String[] args)
  {
    ////////
    // 1, TPBrokerの初期化処理
    ////////

    org.omg.CORBA.ORB orb = null;
    try
```

```
{
  // ORBの初期化
  orb = org.omg.CORBA.ORB.init(args,null);

}
catch(org.omg.CORBA.SystemException ce)
{
  // 例外処理
  System.out.println(ce);
  System.exit(1);
}

////////
// 2, TPBroker OTMの初期化処理
////////
try
{
  // TSCの初期化
  TSCAdm.initServer(args,orb);

}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  System.exit(1);
}

////////
// 3, TSCデーモンへの接続
////////
TSCDomain domain = null;

try
{
  domain = new TSCDomain(null,null);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

TSCServer tsc_server = null;
try
{
  // TSCデーモンの参照オブジェクトを取得
  tsc_server = TSCAdm.getTSCServer(domain);
}
```

```
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }

  System.exit(1);
}

//////
// 4, TSCユーザオブジェクトファクトリ，TSCユーザアクセプタの生成
//    （new）および各種設定
//////
// XYZ_TSCfactimplの生成
TSCObjectFactory  my_fact = new XYZ_TSCfactimpl();

// TSCAcceptorの設定
TSCAcceptor my_acpt = null;
try
{
  my_acpt = new XYZ_TSCacpt(my_fact);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}


////////
// 5, TSCルートアクセプタの生成および各種設定
////////

// TSCRootAcceptorの生成
TSCRootAcceptor my_rt_acpt = null;
try
{
  my_rt_acpt = TSCRootAcceptor.create(tsc_server);

}
catch(TSCSystemException tsc_se)
```

```
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }

  System.exit(1);
}

try
{
  // TSCRootAcceptorへの登録
  my_rt_acpt.registerAcceptor(my_acpt);

  // TSCRootAcceptorのパラレルカウント指定もできます
  // デフォルト値は1
  // オプション引数でデフォルト値を変更することもできます。
  // my_rt_acpt.setParallelCount(5);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);

}

////////
// 6, TSCルートアクセプタの活性化
////////
try
{
  // オブジェクトの活性化
  my_rt_acpt.activate("serviceX");
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
```

```
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

//////
// 7, 実行制御の受け渡し
//////
try
{
  TSCAdm.serverMainloop();
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    my_rt_acpt.deactivate();
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }

  System.exit(1);
}

////////
// 8, TSCルートアクセプタの非活性化
////////
try
{
  // オブジェクトの非活性化
  my_rt_acpt.deactivate();
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }

  System.exit(1);
}
```

```
        ////////
        // 9, TSCデーモンへの接続解放
        ////////
        try
        {
          TSCAdm.releaseTSCServer(tsc_server);
        }
        catch(TSCSystemException tsc_se)
        {
          // 例外処理
          System.out.println(tsc_se);

          try
          {
            TSCAdm.endServer();
          }
          catch(TSCSystemException se)
          {
            System.exit(1);
          }

          System.exit(1);
        }

        ////////
        // 10, TPBroker OTMの終了処理
        ////////
        try
        {
          TSCAdm.endServer();
        }
        catch(TSCSystemException tsc_se)
        {
          // 例外処理
          System.out.println(tsc_se);
          System.exit(1);

        }

        System.exit(0);

      }
    }
```

4.4 セション呼び出しをするアプリケーションプログラム（Java）

セション呼び出しをするアプリケーションプログラムの Java での作成例を示します。これらのサンプルコードは製品で提供されています。

ユーザ定義 IDL インタフェースの例，および IDL コンパイラが生成するクラスは，同期型呼び出しの場合と同様です。「4.2　同期型呼び出しをするアプリケーションプログラム（Java）」を参照してください。

● **トランザクションフレームジェネレータが生成するクラス**

OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。

表 4-3　セション呼び出しをするアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（Java）

分類	生成するクラス名（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• ABC_TSCspxy（TSC ユーザプロキシ）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• ABC_TSCsk（TSC ユーザスケルトン） • ABC_TSCacpt（TSC ユーザアクセプタ）
雛形クラス	• ABC_TSCimpl（TSC ユーザオブジェクト） • ABC_TSCfactimpl（TSC ユーザオブジェクトファクトリ）

4.4.1 セション呼び出しをするクライアントアプリケーションの例（Java）

セション呼び出しをするクライアントアプリケーションの処理の流れとコードの例を示します。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

（1） サービス利用処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
6. TSC **デーモンへの接続解放**

7. TPBroker OTM の終了処理

（2） サービス利用処理のコード

```
//
// "ClientAP.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public class ClientAP
{

  public static void main(String[] args)
  {
    ////////
    // 1, TPBrokerの初期化
    ////////
    org.omg.CORBA.ORB orb = null;
    try
    {
      // ORBの初期化
      orb = org.omg.CORBA.ORB.init(args, null);
    }
    catch(org.omg.CORBA.SystemException ce)
    {
      // 例外処理
      System.out.println(ce);
      System.exit(1);
    }

    ////////
    // 2, TPBroker OTMの初期化処理
    ////////
    // TSCの初期化
    try
    {
      TSCAdm.initClient(args, null, orb);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      System.exit(1);
    }

    ////////
    // 3, TSCデーモンへの接続
    ////////
    TSCDomain domain = null;
    try
    {
      domain = new TSCDomain(null, null);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
```

```
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
    System.exit(1);
}

TSCClient tsc_client = null;
try
{
  tsc_client =
    TSCAdm.getTSCClient(domain, TSCAdm.Regulator);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 4, TSCユーザプロキシの生成および各種設定
////////
// ユーザ定義IDLインタフェース"ABC"用のTSCspxy生成
ABC_TSCspxy my_proxy = null;
try
{
  my_proxy = new ABC_TSCspxy(tsc_client);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCClient(tsc_client);
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}
```

```
////////
// 5, TSCユーザプロキシのメソッド呼び出し
//     （サーバ側のオブジェクトの呼び出し）
////////
try
{
  callSessionService.invoke(my_proxy);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCClient(tsc_client);
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 6, TSCデーモンへの接続解放
////////
try
{
  TSCAdm.releaseTSCClient(tsc_client);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 7, TPBroker OTMの終了処理
////////
try
{
  TSCAdm.endClient();
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
```

```
      System.out.println(tsc_se);
      System.exit(1);
    }

    System.exit(0);

  }

}
```

（3）TSC ユーザプロキシを呼び出すコード

```
//
// "callSessionService.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public
class callSessionService
{
  public static void
  invoke(ABC_TSCspxy my_proxy)
  {

    ////////
    // 1, セションの開始
    ////////
    try
    {
      my_proxy._TSCStart();
    }
    catch(TSCSystemException tsc_se)
    {
      //例外処理
      System.out.println(tsc_se);
      throw tsc_se;
    }

    //////
    // 2, サービスの呼び出し
    //////
    //in引数の準備
    byte[] user_in = new byte[4];

    //out引数の準備
    OctetSeqHolder user_out = new OctetSeqHolder();

    try
    {
      for(int i=0; i<3; ++i)
      {
        //サーバのメソッド呼び出し
        my_proxy.call(user_in, user_out);
      }
    }
    catch(TSCSystemException tsc_se)
```

```
    {
      //例外処理
      System.out.println(tsc_se);
      try{
        my_proxy._TSCStop();
      }
      catch(TSCSystemException se)
      {}
      throw tsc_se;
    }

    ////////
    // 3, セションの停止
    ////////
    try
    {
      my_proxy._TSCStop();
    }
    catch(TSCSystemException tsc_se)
    {
      //例外処理
      System.out.println(tsc_se);
      throw tsc_se;
    }

  }

}
```

4.4.2 セション呼び出しをするサーバアプリケーションの例（Java）

同期型呼び出しの場合と同様です。「4.2.2　同期型呼び出しをするサーバアプリケーションの例（Java）」を参照してください。

4.5 TSCContext を利用するアプリケーションプログラム（Java）

TSCContext を利用するアプリケーションプログラムの Java での作成例を示します。これらのサンプルコードは製品で提供されています。

ユーザ定義 IDL インタフェースの例，IDL コンパイラが生成するクラス，およびトランザクションフレームジェネレータが生成するクラスは，同期型呼び出しの場合と同様です。「4.2　同期型呼び出しをするアプリケーションプログラム（Java）」を参照してください。

4.5.1 TSCContext を利用するクライアントアプリケーションの例（Java）

TSCContext を利用するクライアントアプリケーションの処理の流れとコードの例を示します。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

（1） サービス利用処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **コンテキストへのユーザデータの設定**
6. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
7. TSC **デーモンへの接続解放**
8. TPBroker OTM **の終了処理**

（2） サービス利用処理のコード

```
//
// "ClientAP.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public class ClientAP
{

  public static void main(String[] args)
  {
    ////////
```

```
// 1, TPBrokerの初期化処理
////////
org.omg.CORBA.ORB orb = null;
try
{
  // ORBの初期化
  orb = org.omg.CORBA.ORB.init(args, null);
}
catch(org.omg.CORBA.SystemException ce)
{
  // 例外処理
  System.out.println(ce);
  System.exit(1);
}

////////
// 2, TPBroker OTMの初期化処理
////////
// TSCの初期化
try
{
  TSCAdm.initClient(args, null, orb);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  System.exit(1);
}

////////
// 3, TSCデーモンへの接続
////////
TSCDomain domain = null;
try
{
  domain = new TSCDomain(null, null);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

TSCClient tsc_client = null;
try
{
  tsc_client =
```

```
    TSCAdm.getTSCClient(domain, TSCAdm.Regulator);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 4, TSCユーザプロキシの生成および各種設定
////////
// ユーザ定義IDLインタフェース"ABC"用のTSCProxy生成
ABC_TSCprxy my_proxy = null;
try
{
  my_proxy = new ABC_TSCprxy(tsc_client);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCClient(tsc_client);
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 5, TSCコンテキストへのユーザデータの設定
////////

// TSCContextの取得
TSCContext ctx = my_proxy._TSCContext();

// ユーザIDの設定
String user_id = "UserID:111";
ctx.setUserData(user_id.getBytes());

////////
// 6, TSCユーザプロキシのメソッド呼び出し
//     （サーバ側のオブジェクトの呼び出し）
////////
try
```

```
    {
      callService.invoke(my_proxy);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      try
      {
        TSCAdm.releaseTSCClient(tsc_client);
        TSCAdm.endClient();
      }
      catch(TSCSystemException se)
      {
        System.exit(1);
      }
      System.exit(1);
    }

    ////////
    // 7, TSCデーモンへの接続解放
    ////////
    try
    {
      TSCAdm.releaseTSCClient(tsc_client);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      try
      {
        TSCAdm.endClient();
      }
      catch(TSCSystemException se)
      {
        System.exit(1);
      }
      System.exit(1);
    }

    ////////
    // 8, TPBroker OTMの終了処理
    ////////
    try
    {
      TSCAdm.endClient();
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      System.exit(1);
    }

    System.exit(0);

  }
```

}

（3） TSC ユーザプロキシを呼び出すコード

同期型呼び出しの場合と同様です。「4.2.1(3)　TSC ユーザプロキシを呼び出すコード」を参照してください。

4.5.2 TSCContext を利用するサーバアプリケーションの例（Java）

TSCContext を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス ABC_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス ABC_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。

（1） TSC ユーザオブジェクト（ABC_TSCimpl）のコード

```
//
// "ABC_TSCimpl.java"
//

import OctetSeqHelper;
import OctetSeqHolder;
import JP.co.Hitachi.soft.TPBroker.TSC.TSCContext;

// import classes used in this implementation, if necessary.
import java.lang.System;

public class ABC_TSCimpl extends ABC_TSCsk
{
  // Write class variables, if necessary

  public ABC_TSCimpl()
  {
    // Constructor of implementation.
    // Write user own code.
    //TSCユーザオブジェクトのコンストラクタのコードを記述します。
    //引数の数および型を変更できます。
    super();
  };

  public void call(byte[] in_data, OctetSeqHolder out_data)

  {
    // Operation "call".
    // Write user own code.
    //ユーザメソッドのコードを記述します。

    //TSCコンテキストの取得
```

```
    TSCContext ctx = _TSCContext();

    //ユーザIDの取得
    byte[] data = ctx.getUserData();
    System.out.println(new String(data));

    //メソッドが呼ばれた回数を増加させます。
    //（このメソッドの処理は引数の値と無関係です）
    m_counter++;
    out_data.value = new byte[4];

    System.out.println("Call method in ABC_TSCprxy");

    };

  //メソッドが呼ばれた回数
  protected int m_counter = 0;

};
```

（2） TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のコード

同期型呼び出しの場合と同様です。「4.2.2(2)　TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のコード」を参照してください。

（3） サービス登録処理の流れ・コード

同期型呼び出しの場合と同様です。「4.2.2(3)　サービス登録処理の流れ」,「4.2.2(4) サービス登録処理のコード」を参照してください。

4.6　TSCThread を利用するアプリケーションプログラム（Java）

TSCThread を利用するアプリケーションプログラムの Java での作成例を示します。これらのサンプルコードは製品で提供されています。

ユーザ定義 IDL インタフェースの例，IDL コンパイラが生成するクラス，およびトランザクションフレームジェネレータが生成するクラスは，同期型呼び出しの場合と同様です。「4.2　同期型呼び出しをするアプリケーションプログラム（Java）」を参照してください。

4.6.1　TSCThread を利用するクライアントアプリケーションの例（Java）

同期型呼び出しの場合と同様です。「4.2.1　同期型呼び出しをするクライアントアプリケーションの例（Java）」を参照してください。

4.6.2　TSCThread を利用するサーバアプリケーションの例（Java）

TSCThread を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス ABC_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス ABC_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。さらに，TSC ユーザスレッドとして TSCThread の派生クラスを記述し，TSC ユーザスレッドファクトリとして TSCThreadFactory の派生クラスを記述します。

（1）TSC ユーザオブジェクト（ABC_TSCimpl）のコード

```
//
// "ABC_TSCimpl.java"
//

import OctetSeqHelper;
import OctetSeqHolder;
import JP.co.Hitachi.soft.TPBroker.TSC.TSCThread;

// import classes used in this implementation, if necessary.
import java.lang.System;

public class ABC_TSCimpl extends ABC_TSCsk
```

```
{
  // Write class variables, if necessary

  public ABC_TSCimpl()
  {
    // Constructor of implementation.
    // Write user own code.
    //TSCユーザオブジェクトのコンストラクタのコードを記述します。
    //引数の数および型を変更できます。
    super();
  };

  public void call(byte[] in_data, OctetSeqHolder out_data)

  {
    // Operation "call".
    // Write user own code.
    //ユーザメソッドのコードを記述します。

    //TSCユーザスレッドの取得
    TSCThread my_thr = this._TSCThread();

    //ユーザクラスにキャスト
    UserTImpl my_thr_impl = (UserTImpl)my_thr;

    //UserTImplのメソッドを呼び出し，値を取得します。
    int thr_value = my_thr_impl.getValue();


    //メソッドが呼ばれた回数を増加させます。
    //（このメソッドの処理は引数の値と無関係です）
    m_counter++;
    out_data.value = new byte[4];

    System.out.println("Call method in ABC_TSCprxy");

  };

  //メソッドが呼ばれた回数
  protected int m_counter = 0;

};
```

（2） TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のコード

同期型呼び出しの場合と同様です。「4.2.2(2) TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のコード」を参照してください。

（3） TSC ユーザスレッド（UserTImpl）のコード

```
//
// "UserTImpl.Java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public class UserTImpl
```

```
  implements TSCThread
{
  public UserTImpl(int init_info)
  {
    m_value = init_info;
  }
  public int getValue()
  {
    return m_value;
  }

  protected int m_value;
}
```

（4） TSC ユーザスレッドファクトリ（UserTFactImpl）

```
//
// "UserThreadFactory.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

class UserTFactImpl
  implements TSCThreadFactory
{
  public UserTFactImpl()
  {}
  public TSCThread create()
  {
    //TSCユーザスレッドを生成します。
    TSCThread usr_thr = new UserTImpl(222);
    return usr_thr;
  }

  public void destroy(TSCThread tsc_thr)
  {
    //後処理を記述します。
  }

};
```

（5） サービス登録処理の流れ

1. TPBroker の初期化処理
2. TPBroker OTM の初期化処理
3. TSC デーモンへの接続
4. TSC ユーザオブジェクトファクトリ，TSC ユーザアクセプタの生成（new），および各種設定
5. TSC ユーザスレッドファクトリの生成および各種設定
6. TSC ルートアクセプタの生成および各種設定
7. TSC ルートアクセプタの活性化

8. 実行制御の受け渡し

9. TSC ルートアクセプタの非活性化

10. TSC デーモンへの接続解放

11. TPBroker OTM の終了処理

（6） サービス登録処理のコード

```
//
// "ServerAP.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public class ServerAP
{

  public static void main(String[] args)
  {

    ////////
    // 1, TPBrokerの初期化処理
    ////////

    org.omg.CORBA.ORB orb = null;
    try
    {
      // ORBの初期化
      orb = org.omg.CORBA.ORB.init(args, null);
    }
    catch(org.omg.CORBA.SystemException ce)
    {
      // 例外処理
      System.out.println(ce);
      System.exit(1);
    }

    ////////
    // 2, TPBroker OTMの初期化処理
    ////////
    try
    {
      // TSCの初期化
      TSCAdm.initServer(args, orb);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      System.exit(1);
    }

    ////////
    // 3, TSCデーモンへの接続
    ////////
```

```
TSCDomain domain = null;
try
{
  domain = new TSCDomain(null, null);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

TSCServer tsc_server = null;
try
{
  // TSCデーモンの参照オブジェクトを取得
  tsc_server = TSCAdm.getTSCServer(domain);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 4, TSCユーザオブジェクトファクトリ, TSCユーザアクセプタの
//    生成（new）および各種設定
////////
// ABC_TSCfactimplの生成
TSCObjectFactory my_fact = new ABC_TSCfactimpl();

// TSCAcceptorの生成
TSCAcceptor my_acpt = null;
try
{
  my_acpt = new ABC_TSCacpt(my_fact);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
```

```
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 5, TSCユーザスレッドファクトリの生成および各種設定
////////

TSCThreadFactory my_thr_fact = new UserTFactImpl();

////////
// 6, TSCルートアクセプタの生成および各種設定
////////

// TSCRootAcceptorの生成
TSCRootAcceptor my_rt_acpt = null;
try
{
  my_rt_acpt =
    TSCRootAcceptor.create(tsc_server, my_thr_fact);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

try
{
  // TSCRootAcceptorへの登録
  my_rt_acpt.registerAcceptor(my_acpt);

  // TSCRootAcceptorのパラレルカウント指定もできます。
  // デフォルト値は1
  // オプション引数でデフォルト値を変更することもできます。
  // my_rt_acpt.setParallelCount(5);
}
catch(TSCSystemException tsc_se)
{
```

```
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 7, TSCルートアクセプタの活性化
////////
try
{
  // オブジェクトの活性化
  my_rt_acpt.activate("serviceX");
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 8, 実行制御の受け渡し
////////
try
{
  TSCAdm.serverMainloop();
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    my_rt_acpt.deactivate();
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
```

```
    System.exit(1);
  }
  System.exit(1);
}

////////
// 9, TSCルートアクセプタの非活性化
////////
try
{
  // オブジェクトの非活性化
  my_rt_acpt.deactivate();
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 10, TSCデーモンへの接続解放
////////
try
{
  TSCAdm.releaseTSCServer(tsc_server);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 11, TPBroker OTMの終了処理
////////
try
{
  TSCAdm.endServer();
```

```
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      System.exit(1);
    }

    System.exit(0);

  }
}
```

4.7 ユーザ例外通知を利用するアプリケーションプログラム（Java）

ユーザ例外通知を利用するアプリケーションプログラムの Java での作成例を示します。これらのサンプルコードは製品で提供されています。

● ユーザ定義 IDL インタフェースの例

ユーザ定義 IDL インタフェースの例を次に示します。

```
//
// "UserExcept.idl"
//

exception UserExcept {
    long value;
};

interface EEE {
    void call() raises(UserExcept);
};
```

● IDL コンパイラが生成するクラス

TPBroker の IDL コンパイラはユーザ定義 IDL インタフェースから次のファイルを生成します。

- EEE.java
- EEEHelper.java
- EEEHolder.java
- EEEOperations.java
- UserExcept.java
- UserExceptHelper.java
- UserExceptHolder.java
- _EEEImplBase.java
- _example_ EEE.java ※ 1
- _st_EEE.java ※ 2
- _tie_EEE.java
- EEEPOA.java ※ 3
- EEEPOATie.java ※ 3
- _EEEStub.java ※ 3

注※ 1

Cosminexus TPBroker for Java の ORB Version 4 を使用する場合は，デフォルトでは生成されません。

注※ 2

Cosminexus TPBroker for Java の ORB Version 4 を使用する場合は生成されません。

注※ 3

Cosminexus TPBroker for Java の ORB Version 4 を使用する場合だけ生成されます。

● **トランザクションフレームジェネレータが生成するクラス**

OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。

表 4-4 ユーザ例外通知を利用するアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（Java）

分類	生成するクラス名（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• EEE_TSCprxy（TSC ユーザプロキシ）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• EEE_TSCsk（TSC ユーザスケルトン） • EEE_TSCacpt（TSC ユーザアクセプタ）
雛形クラス	• EEE_TSCimpl（TSC ユーザオブジェクト） • EEE_TSCfactimpl（TSC ユーザオブジェクトファクトリ）

4.7.1 ユーザ例外通知を利用するクライアントアプリケーションの例（Java）

ユーザ例外通知を利用するクライアントアプリケーションの処理の流れとコードの例を示します。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

（1） サービス利用処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
6. TSC **デーモンへの接続解放**
7. TPBroker OTM **の終了処理**

（2） サービス利用処理のコード

```
//
// "ClientAP.java"
```

```
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public class ClientAP
{

  public static void main(String[] args)
  {
    ////////
    // 1, TPBrokerの初期化処理
    ////////
    org.omg.CORBA.ORB orb = null;
    try
    {
      // ORBの初期化
      orb = org.omg.CORBA.ORB.init(args, null);
    }
    catch(org.omg.CORBA.SystemException ce)
    {
      // 例外処理
      System.out.println(ce);
      System.exit(1);
    }

    ////////
    // 2, TPBroker OTMの初期化処理
    ////////
    // TSCの初期化
    try
    {
      TSCAdm.initClient(args, null, orb);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      System.exit(1);
    }

    ////////
    // 3, TSCデーモンへの接続
    ////////
    TSCDomain domain = null;
    try
    {
      domain = new TSCDomain(null, null);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      try
      {
        TSCAdm.endClient();
      }
      catch(TSCSystemException se)
      {
```

```
    System.exit(1);
  }
  System.exit(1);
}

TSCClient tsc_client = null;
try
{
  tsc_client =
    TSCAdm.getTSCClient(domain, TSCAdm.Regulator);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 4, TSCユーザプロキシの生成および各種設定
////////
// ユーザ定義IDLインタフェース"EEE"用のTSCProxy生成
EEE_TSCprxy my_proxy = null;
try
{
  my_proxy = new EEE_TSCprxy(tsc_client);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCClient(tsc_client);
    TSCAdm.endClient();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 5, TSCユーザプロキシのメソッド呼び出し
//     （サーバ側のオブジェクトの呼び出し）
////////
try
{
```

```
      callService.invoke(my_proxy);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      try
      {
        TSCAdm.releaseTSCClient(tsc_client);
        TSCAdm.endClient();
      }
      catch(TSCSystemException se)
      {
        System.exit(1);
      }
      System.exit(1);
    }

    ////////
    // 6, TSCデーモンへの接続解放
    ////////
    try
    {
      TSCAdm.releaseTSCClient(tsc_client);
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      try
      {
        TSCAdm.endClient();
      }
      catch(TSCSystemException se)
      {
        System.exit(1);
      }
      System.exit(1);
    }

    ////////
    // 7, TPBroker OTMの終了処理
    ////////
    try
    {
      TSCAdm.endClient();
    }
    catch(TSCSystemException tsc_se)
    {
      // 例外処理
      System.out.println(tsc_se);
      System.exit(1);
    }

    System.exit(0);

  }
```

```
}
```

（3）TSC ユーザプロキシを呼び出すコード

```
//
// "callService.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public
class callService
{
  public static void
  invoke(EEE_TSCprxy my_proxy)
  {

    //////
    //サービスの呼び出し
    //////
    try
    {
      //サーバのメソッド呼び出し
      my_proxy.call();
    }
    catch(UserExcept tsc_se)
    {
      System.out.println("catch" + tsc_se.value);
    }
    catch(TSCSystemException tsc_se)
    {
      //例外処理
      System.out.println(tsc_se);
      throw tsc_se;
    }
  }

}
```

4.7.2 ユーザ例外通知を利用するサーバアプリケーションの例（Java）

ユーザ例外通知を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス EEE_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス EEE_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。

（1）TSC ユーザオブジェクト（EEE_TSCimpl）のコード

```
//
```

```
// "EEE_TSCimpl.java"
//

import UserExcept;
import UserExceptHelper;
import UserExceptHolder;

public  EEE_TSCimpl extends EEE_TSCsk
{
  public
  EEE_TSCimpl()
  {
    // Constructor of implementation.
    // Write user own code.
    super();
    //TSCユーザオブジェクトのコンストラクタのコードを記述します。
    //引数の数および型を変更できます。

  }

  //ユーザ定義IDLインタフェース依存のメソッド

  public void
  call()
  {
    // Operation "call".
    // Write user own code.
    //ユーザメソッドのコードを記述します。

    throw new UserExcept(1);
  }
};
```

(2) TSC ユーザオブジェクトファクトリ（EEE_TSCfactimpl）のコード

```
//
// "EEE_TSCfactimpl".java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.TSCObject;
import JP.co.Hitachi.soft.TPBroker.TSC.TSCObjectFactory;

import UserExcept;
import UserExceptHelper;
import UserExceptHolder;

class EEE_TSCfactimpl : public TSCObjectFactory
{
  public
  EEE_TSCfactimpl()
  {
    // Constructor of implementation.
    // Write user own code.
    // TSCユーザオブジェクトファクトリのコンストラクタのコードを
    // 記述します。
    //引数の数および型を変更できます。
  }
```

```
  public TSCObject create()
  {
    // Method to create user object.
    // Write user own code.
    //サーバオブジェクトを生成するコードを記述します。
    //必要に応じて修正してください。
    return new EEE_TSCimpl();
  }

  public void destroy(TSCObject tsc_object)
  {
    // Method to destroy user object.
    // Write user own code.
    //後処理のコードを記述します。
    //必要に応じて修正してください。
  }
};
```

（3） サービス登録処理の流れ

1. TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザオブジェクトファクトリ，**TSC **ユーザアクセプタの生成（**new**），および各種設定**
5. TSC **ルートアクセプタの生成および各種設定**
6. TSC **ルートアクセプタの活性化**
7. **実行制御の受け渡し**
8. TSC **ルートアクセプタの非活性化**
9. TSC **デーモンへの接続解放**
10. TPBroker OTM **の終了処理**

（4） サービス登録処理のコード

```
//
// "ServerAP.java"
//

import JP.co.Hitachi.soft.TPBroker.TSC.*;

public class ServerAP
{

  public static void main(String[] args)
  {

    ////////
    // 1, TPBrokerの初期化処理
```

```
////////

org.omg.CORBA.ORB orb = null;
try
{
  // ORBの初期化
  orb = org.omg.CORBA.ORB.init(args, null);
}
catch(org.omg.CORBA.SystemException ce)
{
  // 例外処理
  System.out.println(ce);
  System.exit(1);
}

////////
// 2, TPBroker OTMの初期化処理
////////
try
{
  // TSCの初期化
  TSCAdm.initServer(args, orb);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  System.exit(1);
}

////////
// 3, TSCデーモンへの接続
////////
TSCDomain domain = null;
try
{
  domain = new TSCDomain(null, null);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

TSCServer tsc_server = null;
try
{
  // TSCデーモンの参照オブジェクトを取得
  tsc_server = TSCAdm.getTSCServer(domain);
```

```
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 4, TSCユーザオブジェクトファクトリ，TSCユーザアクセプタの
//    生成（new）および各種設定
////////
// EEE_TSCfactimplの生成
TSCObjectFactory my_fact = new EEE_TSCfactimpl();

// TSCAcceptorの生成
TSCAcceptor my_acpt = null;
try
{
  my_acpt = new EEE_TSCacpt(my_fact);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 5, TSCルートアクセプタの生成および各種設定
////////

// TSCRootAcceptorの生成
TSCRootAcceptor my_rt_acpt = null;
try
{
  my_rt_acpt = TSCRootAcceptor.create(tsc_server);
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
```

```
    System.out.println(tsc_se);
    try
    {
      TSCAdm.releaseTSCServer(tsc_server);
      TSCAdm.endServer();
    }
    catch(TSCSystemException se)
    {
      System.exit(1);
    }
    System.exit(1);
  }

  try
  {
    // TSCRootAcceptorへの登録
    my_rt_acpt.registerAcceptor(my_acpt);

    // TSCRootAcceptorのパラレルカウント指定もできます。
    // デフォルト値は1
    // オプション引数でデフォルト値を変更することもできます。
    // my_rt_acpt.setParallelCount(5);
  }
  catch(TSCSystemException tsc_se)
  {
    // 例外処理
    System.out.println(tsc_se);
    try
    {
      TSCAdm.releaseTSCServer(tsc_server);
      TSCAdm.endServer();
    }
    catch(TSCSystemException se)
    {
      System.exit(1);
    }
    System.exit(1);
  }

  ////////
  // 6, TSCルートアクセプタの活性化
  ////////
  try
  {
    // オブジェクトの活性化
    my_rt_acpt.activate("serviceX");
  }
  catch(TSCSystemException tsc_se)
  {
    // 例外処理
    System.out.println(tsc_se);
    try
    {
      TSCAdm.releaseTSCServer(tsc_server);
      TSCAdm.endServer();
    }
    catch(TSCSystemException se)
```

```
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 7, 実行制御の受け渡し
////////
try
{
  TSCAdm.serverMainloop();
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    my_rt_acpt.deactivate();
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 8, TSCルートアクセプタの非活性化
////////
try
{
  // オブジェクトの非活性化
  my_rt_acpt.deactivate();
}
catch(TSCSystemException tsc_se)
{
  // 例外処理
  System.out.println(tsc_se);
  try
  {
    TSCAdm.releaseTSCServer(tsc_server);
    TSCAdm.endServer();
  }
  catch(TSCSystemException se)
  {
    System.exit(1);
  }
  System.exit(1);
}

////////
// 9, TSCデーモンへの接続解放
////////
```

```
        try
        {
          TSCAdm.releaseTSCServer(tsc_server);
        }
        catch(TSCSystemException tsc_se)
        {
          // 例外処理
          System.out.println(tsc_se);
          try
          {
            TSCAdm.endServer();
          }
          catch(TSCSystemException se)
          {
            System.exit(1);
          }
          System.exit(1);
        }

        ////////
        // 10, TPBroker OTMの終了処理
        ////////
        try
        {
          TSCAdm.endServer();
        }
        catch(TSCSystemException tsc_se)
        {
          // 例外処理
          System.out.println(tsc_se);
          System.exit(1);
        }

        System.exit(0);

      }
    }
```

4.8　TSCWatchTime を利用するアプリケーションプログラム（Java）

TSCWatchTime を利用するアプリケーションプログラムの Java での作成例を示します。

ユーザ定義 IDL インタフェースの例，IDL コンパイラが生成するクラス，およびトランザクションフレームジェネレータが生成するクラスは，同期型呼び出しの場合と同様です。「4.2　同期型呼び出しをするアプリケーションプログラム（Java）」を参照してください。

4.8.1　TSCWatchTime を利用するクライアントアプリケーションの例（Java）

同期型呼び出しの場合と同様です。「4.2.1　同期型呼び出しをするクライアントアプリケーションの例（Java）」を参照してください。

4.8.2　TSCWatchTime を利用するサーバアプリケーションの例（Java）

TSCWatchTime を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス ABC_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス ABC_TSCfactimpl に TSC ユーザオブジェクトファクトリのコードを記述します。

（1）TSC ユーザオブジェクト（ABC_TSCimpl）のコード

```
//
// "ABC_TSCimpl.java"
//
import OctetSeqHelper;
import OctetSeqHolder;
import JP.co.Hitachi.soft.TPBroker.TSC.TSCWatchTime;


// import classes used in this implementation, if necessary.
import java.lang.System;

public class ABC_TSCimpl extends ABC_TSCsk
{
  // Write class variables, if necessary

  public ABC_TSCimpl()
  {
```

```
    // Constructor of implementation.
    // Write user own code.
    super();

    ////////
    //時間監視の処理
    ////////
    TSCWatchTime watch_time = null;
    try
    {
      //監視時間60秒の時間監視オブジェクト生成
      watch_time = new TSCWatchTime(60);

      //時間監視の開始
      watch_time.start();

      //TSCユーザオブジェクトのコンストラクタのコードを記述します。
      //引数の数および型を変更できます。

      //時間監視の中断
      watch_time.stop();
    }
    catch(TSCSystemException se)
    {
      //例外処理
      System.out.println(se);
      throw se;
    }

};

  public void call(byte[] in_data, OctetSeqHolder out_data)

  {
    // Operation "call".
    // Write user own code.
    //ユーザメソッドのコードを記述します。

    //メソッドが呼ばれた回数を増加させます。
    //（このメソッドの処理は引数の値と無関係です）
    m_counter++;
    out_data.value = new byte[4];

    };

  //メソッドが呼ばれた回数
  protected int m_counter = 0;

};
```

(2) TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のコード

同期型呼び出しの場合と同様です。「4.2.2(2) TSC ユーザオブジェクトファクトリ（ABC_TSCfactimpl）のコード」を参照してください。

（3） サービス登録処理の流れ・コード

同期型呼び出しの場合と同様です。「4.2.2(3)　サービス登録処理の流れ」,「4.2.2(4)　サービス登録処理のコード」を参照してください。

5 アプリケーションプログラミングインタフェース（Java）

この章では，Java で使用するクラスライブラリについて説明します。
ユーザ定義 IDL インタフェース依存クラスと雛形クラスでは，対応するユーザ定義 IDL インタフェースの名称を "ABC" と仮定します。なお，各クラスの詳細は，アルファベット順に示します。

クラスの一覧（Java）

各クラスは次のように分類されます。

- システム提供クラス
- システム提供例外クラス
- ユーザ定義 IDL インタフェース依存クラス
- 雛形クラス

各クラスの一覧を次の表に示します。

表 5-1　クラス一覧（Java）

分類	クラス
システム提供クラス	• TSCAcceptor • TSCAdm • TSCClient • TSCContext • TSCDomain • TSCObject • TSCObjectFactory • TSCProxyObject • TSCRootAcceptor • TSCServer • TSCSessionProxy • TSCThread • TSCThreadFactory • TSCWatchTime
システム提供例外クラス	• TSCSystemException

分類	クラス	
	• TSCSystemException の派生クラス	• TSCBadContextException • TSCBadInvOrderException • TSCBadOperationException • TSCBadParamException • TSCBadTypecodeException • TSCCommFailureException • TSCDataConversionException • TSCFreeMemException • TSCImpLimitException • TSCInitializeException • TSCInternalException • TSCIntfReposException • TSCInvFlagException • TSCInvIdentException • TSCInvObjrefException • TSCMarshalException • TSCNoImplementException • TSCNoMemoryException • TSCNoPermissionException • TSCNoResourcesException • TSCNoResponseException • TSCObjAdapterException • TSCObjectNotExistException • TSCPersistStoreException • TSCTransientException • TSCUnknownException
ユーザ定義 IDL インタフェース依存クラス（基底クラス）	• ABC_TSCacpt（TSCAcceptor） • ABC_TSCprxy（TSCProxyObject） • ABC_TSCsk（TSCObject） • ABC_TSCspxy（TSCSessionProxy）	
雛形クラス（基底クラス）	• ABC_TSCfactimpl（ABC_TSCfactsk） • ABC_TSCimpl（ABC_TSCsk）	

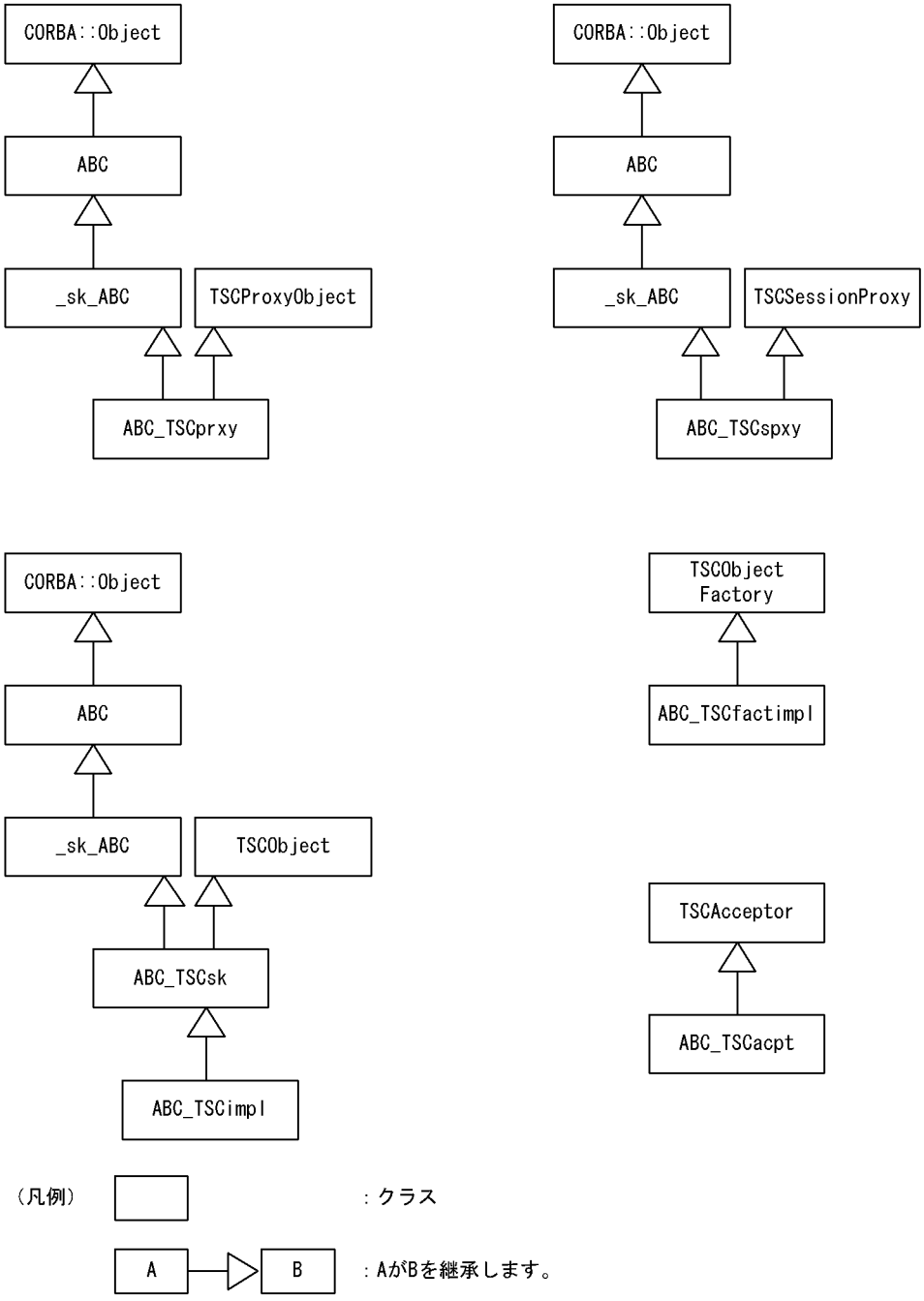
基本データ型（Java）

OTM の各クラスで使用する基本データ型は，Java が提供する標準型です。

クラス関連図（Java）

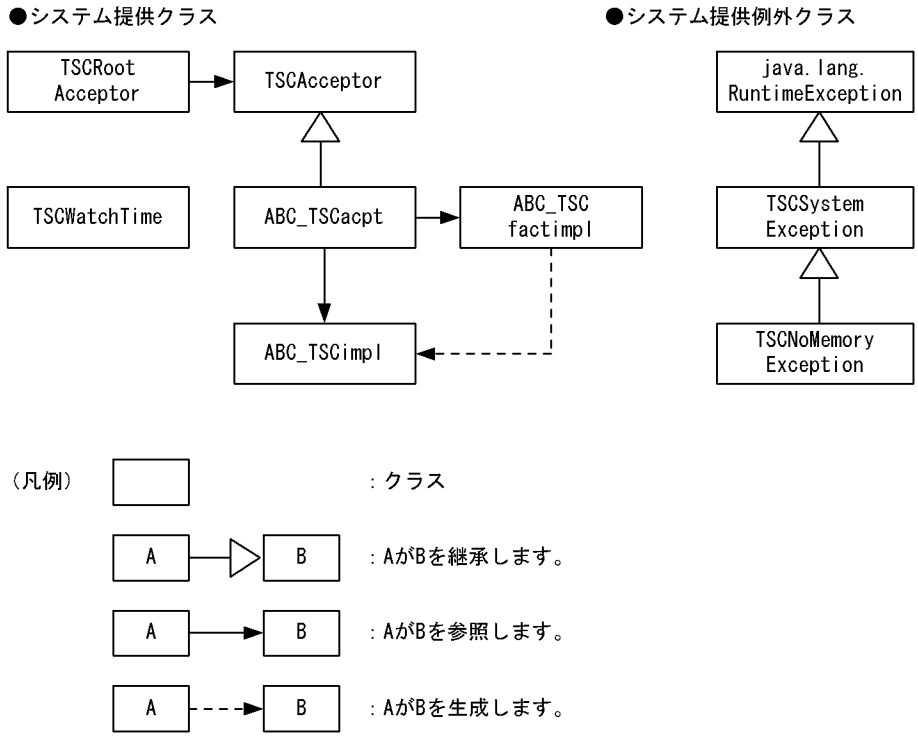
システム提供クラスから派生するユーザ定義 IDL インタフェース依存クラスの関連を次の図に示します。

図 5-1　システム提供クラスから派生するユーザ定義 IDL インタフェース依存クラス（Java）



サーバアプリケーションを作成する際に使用するシステム提供クラスおよびシステム提供例外クラスの関連を次の図に示します。

図 5-2　システム提供クラスおよびシステム提供例外クラス（Java）



公開メソッド呼び出しと内部参照（Java）

システム提供クラスのインスタンス関連図を基に，OTM 上でのインスタンス間の公開メソッド呼び出し，および内部参照（アクセス）について説明します。

● **公開メソッド呼び出し**
OTM のシステム提供クラスのインスタンスが，ほかのインスタンスの公開メソッドを呼び出すことです。

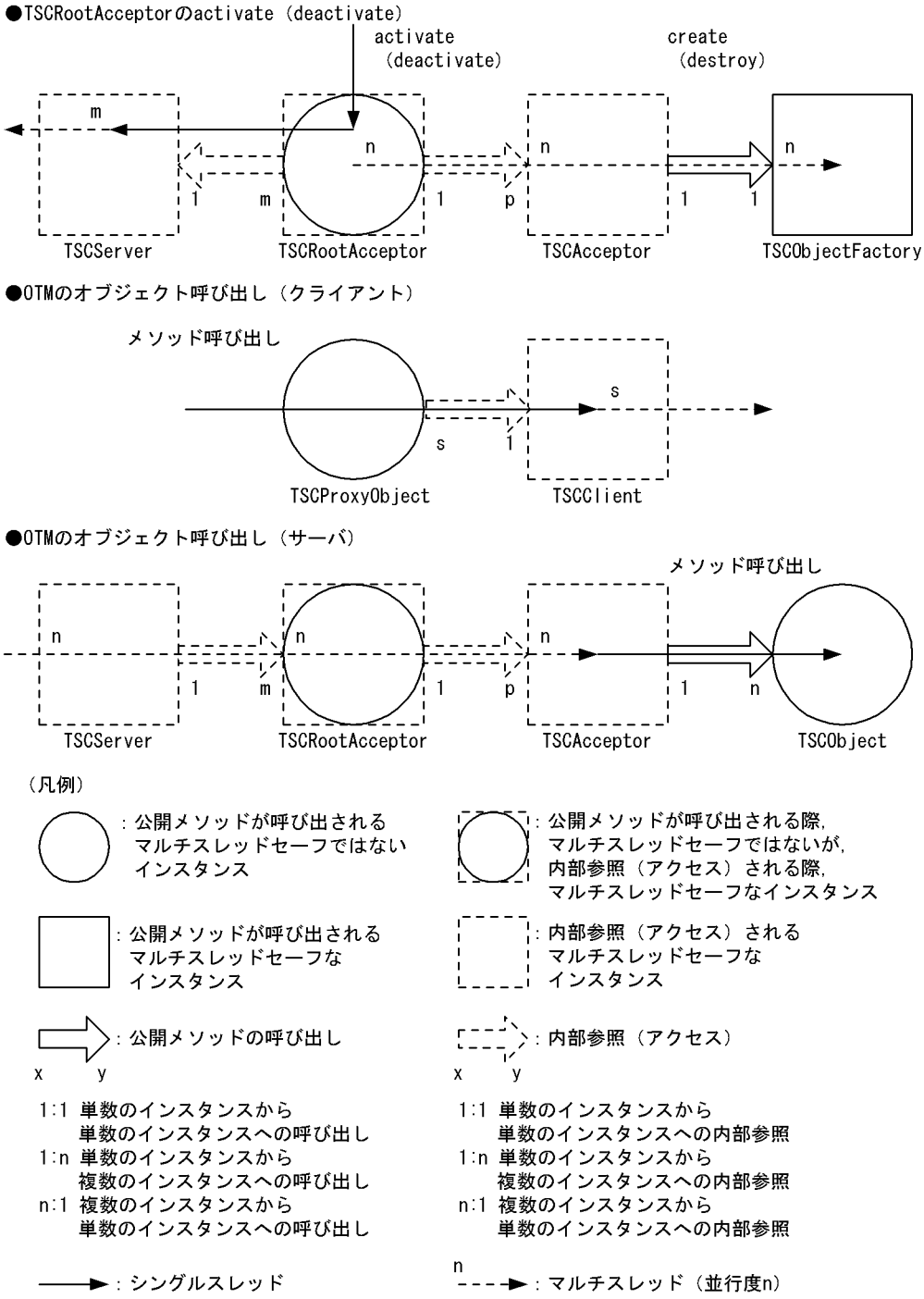
● **内部参照（アクセス）**
OTM のシステム提供クラスのインスタンスが，ほかのインスタンスの非公開メソッドを内部的にアクセスすることです。またはそのインスタンスを内部的に参照することです。

図中のインスタンス数を次の表に示します。また，システム提供クラスのインスタンス関連を図 5-3 に示します。

表 5-2　インスタンス数（Java）

インスタンス	インスタンス数
TSC ルートアクセプタ	m（TSCServer 単位）
パラレルカウント	n（TSCRootAcceptor 単位）
TSC ユーザアクセプタ	p（TSCRootAcceptor 単位）
TSC ユーザオブジェクトファクトリ	1（TSCAcceptor 単位）
TSC ユーザプロキシ	s（TSCClient 単位）

図 5-3　システム提供クラスのインスタンス関連図（Java）



ABC_TSCacpt（Java）

ABC_TSCacpt はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCacpt は，TSC ユーザアクセプタの実装クラスです。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCacpt を自動生成します。次に ABC_TSCacpt の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。

形式

```
package ***;

class ABC_TSCacpt extends TSCAcceptor
{
  public ABC_TSCacpt(TSCObjectFactory_ptr tsc_object_fact);
  public ABC_TSCacpt(TSCObjectFactory_ptr tsc_object_fact,
                              const char* tsc_acpt_name);

  ~ABC_TSCacpt();
}
```

コンストラクタ

● public ABC_TSCacpt(TSCObjectFactory_ptr tsc_object_fact)

項目	型・意味	
引数	TSCObjectFactory_ptr tsc_object_fact	TSCObjectFactory オブジェクト
例外	TSCBadParamException	

TSC ユーザオブジェクトファクトリとして tsc_object_fact を保持します。さらに，デフォルトの TSC アクセプタ名称で ABC_TSCacpt を生成します。

サーバアプリケーションの開始時にコマンドオプション引数 -TSCAcceptor を指定しない場合，TSC アクセプタ名称のデフォルト値は " 指定なし " となります。コマンドオプション引数 -TSCAcceptor を指定する場合は，TSC アクセプタ名称のデフォルト値はその指定値となります。

● public ABC_TSCacpt(TSCObjectFactory_ptr tsc_object_fact,
const char* tsc_acpt_name)

項目	型・意味	
引数	TSCObjectFactory_ptr tsc_object_fact	TSCObjectFactory オブジェクト
	const char* tsc_acpt_name	TSC アクセプタ名称
例外	TSCBadParamException	

TSC ユーザオブジェクトファクトリとして tsc_object_fact を保持します。さらに，デフォルトの TSC アクセプタ名称で ABC_TSCacpt を生成します。ただし，tsc_acpt_name には，1 〜 31 文字の TSC アクセプタ名称を指定してください。

デストラクタ

● ~ABC_TSCacpt()

ABC_TSCacpt を削除します。

ABC_TSCfactimpl（Java）

ABC_TSCfactimpl は雛形クラスです。

ABC_TSCfactimpl は，TSC ユーザオブジェクトファクトリの実装クラスです。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCfactimpl を生成します。ユーザはこの雛形クラスに処理依存のコードを記述します。次に ABC_TSCfactimpl の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。

形式

*斜体*で示している部分は，ユーザが実装のコードを記述する必要があるメソッドです。**太字**で示している部分は，引数の型および数を変更できるメソッドで，ユーザが実装のコードを記述する必要があります。

```
class ABC_TSCfactimpl implements TSCObjectFactory
{
  public ABC_TSCfactimpl(....);

  public TSCObject create();
  public void destroy(TSCObject tsc_object);

};
```

コンストラクタ

● public ABC_TSCfactimpl(....)

引数の型や数を含めて，ユーザがコードを記述する必要があります。複数のコンストラクタを生成できます。

コールバックメソッド

● public TSCObject create()

典型的なコードを生成します。必要に応じて変更してください。

● public void destroy(TSCObject tsc_object)

典型的なコードを生成します。必要に応じて変更してください。

ABC_TSCimpl（Java）

ABC_TSCimpl は雛形クラスです。

ABC_TSCimpl は，TSC ユーザオブジェクトの実装クラスです。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCimpl を生成します。ユーザはこの雛形クラスに処理依存のコードを記述します。次に ABC_TSCimpl の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。

ABC_TSCimpl は，TPBroker のスケルトンである _sk_ABC も継承します。

ユーザ定義 IDL インタフェースのマッピング

ユーザ定義 IDL インタフェース内に定義されたオペレーションの Java 言語へのマッピングは，TPBroker と同様です。

形式

*斜体*で示している部分は，ユーザが実装のコードを記述する必要があるメソッドです。**太字**で示している部分は，引数の型および数を変更できるメソッドで，ユーザが実装のコードを記述する必要があります。

```
package ***;
public class ABC_TSCimpl extends ABC_TSCsk
{
  public ABC_TSCimpl(....);

  //ユーザ定義IDLインタフェース依存のメソッド群
  ... xxx(...);

};
```

コンストラクタ

● public ABC_TSCimpl(....)

引数の型や数を含めて，ユーザがコードを記述する必要があります。複数のコンストラクタを生成できます。

コールバックメソッド

● ... xxx(...)

ユーザ定義 IDL インタフェースのオペレーション定義に従ったメソッドです。ユーザがメソッドのコードを実装します。メソッドの引数・戻り値の型や数のマッピングは，TPBroker と同じです。

ABC_TSCprxy（Java）

ABC_TSCprxy はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCprxy は，TSC ユーザプロキシの実装クラスです。ユーザ定義 IDL インタフェースの定義に従って，ユーザデータをバイト配列データに変換し，TSCProxyObject を呼び出します。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCprxy を自動生成します。次に ABC_TSCprxy の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。

ABC_TSCprxy は，TPBroker のスケルトンである _sk_ABC も継承します。また，ABC_TSCprxy の派生クラスも同様に，TPBroker のスケルトンを継承します。

ユーザ定義 IDL インタフェースのマッピング

ユーザ定義 IDL インタフェース内に定義されたオペレーションの Java 言語へのマッピングは，TPBroker と同じです。

形式

```
class ABC_TSCprxy extends _sk_ABC implements TSCProxyObject
{
  public ABC_TSCprxy(TSCClient tsc_client);
  public ABC_TSCprxy(TSCClient tsc_client,
                     String tsc_acpt_name);

  //ユーザ定義IDLインタフェース依存のメソッド群
  ...xxx(...);
};
```

コンストラクタ

● public ABC_TSCprxy(TSCClient tsc_client)

項目	型・意味	
引数	TSCClient_ptr tsc_client	接続する TSCClient
例外	TSCBadParamException	

tsc_client と接続する ABC_TSCprxy を生成します。

● public ABC_TSCprxy(TSCClient tsc_client,
String tsc_acpt_name)

項目	型・意味	
引数	TSCClient_ptr tsc_client	接続する TSCClient
	String tsc_acpt_name	TSC アクセプタ名称
例外	TSCBadParamException	

tsc_client と接続する TSC アクセプタ名称が tsc_acpt_name の ABC_TSCprxy を生成します。

メソッド

● ... xxx(...)

項目	型・意味
例外	TSCSystemException（各種例外）

ユーザ定義 IDL インタフェースのオペレーション定義に従ったメソッドです。メソッドの引数・戻り値の型や数のマッピングは，TPBroker と同じです。

ABC_TSCsk（Java）

ABC_TSCsk はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCsk は，TSC ユーザスケルトンの実装クラスです。クライアント側から送信されたバイト配列データとともに呼び出されます。ユーザ定義 IDL インタフェースに従って，そのバイト配列データをユーザデータに分解し，TSC ユーザオブジェクトの実装メソッドを呼び出します。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCsk を自動生成します。次に ABC_TSCsk の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。
- 直接，このクラスのインスタンスを生成できません。
- ユーザは ABC_TSCsk クラスを継承して，実装クラスを記述する必要があります。

ABC_TSCsk は，TPBroker のスケルトンである _sk_ABC も継承します。また，ABC_TSCsk の派生クラスも同様に，TPBroker のスケルトンを継承します。

ユーザ定義 IDL インタフェースのマッピング

ユーザ定義 IDL インタフェース内に定義されたオペレーションの Java 言語へのマッピングは，TPBroker と同じです。

形式

```
public class ABC_TSCsk
  extends _sk_ABC implements virtual TSCObject
{
  public abstruct ... xxx(...);

  protected ABC_TSCsk();

};
```

コンストラクタ

● protected ABC_TSCsk()

ABC_TSCsk を生成します。

コールバックメソッド

● public abstruct ... xxx(...)

ユーザ定義 IDL インタフェースのオペレーション定義に従ったメソッドです。メソッドの引数・戻り値の型や数のマッピングは，TPBroker と同じです。

ABC_TSCspxy（Java）

ABC_TSCspxy はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCspxy は，セション呼び出し用の TSC ユーザプロキシの実装クラスです。ユーザ定義 IDL インタフェースの定義に従って，ユーザデータをバイト配列データに変換し，TSCSessionProxy を呼び出します。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCspxy を自動生成します。
ABC_TSCspxy は，次の点を除いて ABC_TSCprxy と同様の働きをします。

- TSCSessionProxy を継承します。
- トランザクションフレームジェネレータに -TSCspxy オプションを指定したときだけ生成されます。
- oneway のオペレーションを定義したユーザ定義 IDL からは生成できません。

次に ABC_TSCspxy の特徴を示します。

- ユーザ定義 IDL インタフェース依存のコードを含みます。

ABC_TSCspxy は，TPBroker のスケルトンである _sk_ABC も継承します。また，ABC_TSCspxy の派生クラスも同様に，TPBroker のスケルトンを継承します。

ユーザ定義 IDL インタフェースのマッピング

ユーザ定義 IDL インタフェース内に定義されたオペレーションの Java 言語へのマッピングは，TPBroker と同じです。

形式

```
class ABC_TSCspxy extends _sk_ABC implements TSCSessionProxy
{
  public ABC_TSCspxy(TSCClient tsc_client);
  public ABC_TSCspxy(TSCClient tsc_client,
                         String tsc_acpt_name);

  //ユーザ定義IDLインタフェース依存のメソッド群
  ...xxx(...);
};
```

コンストラクタ

● public ABC_TSCspxy(TSCClient tsc_client)

項目	型・意味	
引数	TSCClient_ptr tsc_client	接続する TSCClient
例外	TSCBadParamException	

tsc_client と接続する ABC_TSCspxy を生成します。

● public ABC_TSCspxy(TSCClient tsc_client,
String tsc_acpt_name)

項目	型・意味	
引数	TSCClient_ptr tsc_client	接続する TSCClient
	String tsc_acpt_name	TSC アクセプタ名称
例外	TSCBadParamException	

tsc_client と接続する TSC アクセプタ名称が tsc_acpt_name の ABC_TSCspxy を生成します。

メソッド

● ... xxx(...)

項目	型・意味
例外	TSCSystemException（各種例外）

ユーザ定義 IDL インタフェースのオペレーション定義に従ったメソッドです。メソッドの引数・戻り値の型や数のマッピングは，TPBroker と同じです。

TSCAcceptor（Java）

TSCAcceptor はシステム提供クラスです。

TSCAcceptor は，TSC ユーザプロキシを使用したクライアント側からの TSC ユーザオブジェクトのメソッド呼び出し要求に対して，サーバ側の TSC ユーザオブジェクトのメソッドを呼び出すためのクラスです。TSC ルートアクセプタから TSC ユーザオブジェクトのメソッド呼び出し要求を受け取り，TSC ユーザオブジェクトのメソッドを呼び出します。これに伴って，TSC ユーザオブジェクトを管理したり，スレッドと TSC ユーザオブジェクト間を対応づけたりします。また，TSC サービス識別子を使用して，TSC ユーザオブジェクトが提供するサービスを識別します。

ユーザは TSC ユーザオブジェクトの管理オブジェクトとして，TSCAcceptor クラスのインスタンスを生成します。次に TSCAcceptor の特徴を示します。

- TSCObjectFactory を保持することで，TSC ユーザオブジェクト（TSCObject）を管理します。
- TSCAcceptor が提供できるサービスを TSC **サービス識別子**の列で表現します。TSC サービス識別子は，**インタフェース名称**の列と TSC **アクセプタ名称**から構成されます。ただし，TSC アクセプタ名称がない場合もあります。

TSC ユーザオブジェクト（TSCObject）の管理

TSCObjectFactory による TSCObject の管理

TSCRootAcceptor が active 状態に遷移するとき，登録されている TSCAcceptor はオブジェクト管理開始通知を受けます。また，TSCRootAcceptor が non-active 状態に遷移するとき，登録されている TSCAcceptor はオブジェクト管理終了通知を受けます。それぞれの通知とともに，TSCAcceptor は TSCObjectFactory を使用して次に示すように動作します。

- **TSCRootAcceptor からのオブジェクト管理開始通知**
  TSCAcceptor は，TSCRootAcceptor からオブジェクト管理開始通知を受けると，TSCRootAcceptor が保持する各スレッド上で，TSCObjectFactory の create を呼び出します。さらに，この呼び出しで返される TSCObject を TSCRootAcceptor が保持する各スレッドに割り当てます。これによって，TSCObjectFactory の create 呼び出しで返される TSCObject は，スレッドに対応づけて管理されます。
- **TSCRootAcceptor からのオブジェクト管理終了通知**
  TSCAcceptor は，TSCRootAcceptor からオブジェクト管理終了通知を受けると，TSCRootAcceptor が保持する各スレッド上で，割り当てられている TSCObject を引数に，TSCObjectFactory の destroy を呼び出します。これによって，対応づけられているスレッド上の管理対象から，該当する TSCObject が外されます。

TSCRootAcceptor からの TSCObject の呼び出し

TSCRootAcceptor は，クライアント側からの TSC ユーザオブジェクト呼び出しを受け取ると，該当するサービスを提供する TSCAcceptor に振り分けます。さらに，TSCAcceptor は，TSCRootAcceptor が管理するスレッド上で TSC ユーザオブジェクト呼び出し要求を受け取ると，同じスレッド上で同じスレッドに割り当てられている TSCObject を呼び出します。

TSC サービス識別子によるサービスの識別

TSC サービス識別子の構成

TSCAcceptor が提供できるサービスの種類は，TSC サービス識別子によって表されます。TSC サービス識別子は，インタフェース名称の列と TSC アクセプタ名称から構成されます。

- **TSCAcceptor のインタフェース名称の列**
  TSCAcceptor のインタフェース名称の列は，TSCAcceptor や管理する TSCObject が提供するサービスのインタフェースの種類を表します。
  TSCObject が単数のインタフェースをサポートする場合，TSCAcceptor や管理する TSCObject が提供するインタフェースの種類は，単数のインタフェース名称で表されます。TSCObject が複数のインタフェースをサポートする場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，複数のインタフェース名称が列で表されます。
- **TSCAcceptor の TSC アクセプタ名称**
  TSC アクセプタ名称も，TSCAcceptor や管理する TSCObject が提供するサービスのインタフェースの種類を表します。ただし，同じインタフェースを提供する TSCAcceptor または TSCObject の間の実装内容の違いを識別するために使用します。したがって，TSC アクセプタ名称を設定しないで，"TSC アクセプタ名称なし " とすることもできます。

サービスの種類の表現方法

TSCAcceptor が提供できるサービスの種類は，TSC サービス識別子の列によって表されます。TSCAcceptor は TSC サービス識別子の列で示されるサービスを提供できます。

- **TSCAcceptor に TSC アクセプタ名称が設定されていない場合**
  TSCObject が単数のインタフェースをサポートする場合，TSCAcceptor や管理する TSCObject が提供できるサービスの種類は，次のように表されます。

  単数のTSCアクセプタ名称なしTSCサービス識別子

  TSCObject が複数のインタフェースをサポートする場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，TSC サービス識別子の列で次のように表されます。

複数のTSCアクセプタ名称なしTSCサービス識別子

- **TSCAcceptor に TSC アクセプタ名称が設定されている場合**
  TSCObject が単数のインタフェースをサポートする場合，TSCAcceptor や管理する TSCObject が提供できるサービスの種類は，次のように表されます。

  単数のTSCアクセプタ名称なしTSCサービス識別子，および
  単数のTSCアクセプタ名称ありTSCサービス識別子

  TSCObject が複数のインタフェースをサポートする場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，TSC サービス識別子の列で次のように表されます。

  複数のTSCアクセプタ名称なしTSCサービス識別子，および
  複数のTSCアクセプタ名称ありTSCサービス識別子（ただし，TSCサービス識別子中のTSCアクセプタ名称は同じ）

サービスの種類の表現例

- **TSCAcceptor のインタフェース名称が "ABC" で，TSC アクセプタ名称がない場合**
  次のように表される場合，TSCAcceptor は，"ABC::" への要求だけを受け付けることができます。

  ```
  "ABC::"
  ```

- **TSCAcceptor のインタフェース名称が "ABC" で，TSC アクセプタ名称が "abc" の場合**
  次のように表される場合，TSCAcceptor は，"ABC::" と "ABC::abc" への要求メッセージを受け付けることができます。

  ```
  "ABC::abc"
  "ABC::"
  ```

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;

public class TSCAcceptor
{

  //インタフェース名称列
  String[] getInterfaceName();

  //TSCアクセプタ名称列
  String getAcceptorName();

};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCAcceptor;
```

メソッド

● String[] getInterfaceName()

項目	型・意味
戻り値	インタフェース名称の列

インタフェース名称の列を取得します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● String getAcceptorName()

項目	型・意味
戻り値	TSC アクセプタ名称

TSC アクセプタ名称を取得します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

TSCAcceptor または派生クラスの生成

TSCAcceptor または派生クラスを生成するには，引数を指定しないで，または TSC アクセプタ名称を指定して，new オペレータで生成します。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCAcceptor クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
getInterfaceName	できます。
getAcceptorName	できます。

インスタンスの公開メソッド呼び出し規則

TSCAcceptor クラスのインスタンスが，ほかのクラスのインスタンスの公開メソッドを呼び出す規則を次に示します。

タイミング	公開メソッド呼び出し
TSCRootAcceptor からのオブジェクト管理開始通知，またはオブジェクト管理終了通知のとき	コンストラクタで指定した TSCObjectFactory 型のインスタンス
登録先の TSCRootAcceptor が active 状態のとき	active 状態に遷移するときに管理を開始した TSCObject 型のインスタンス

インスタンスへの内部参照（アクセス）規則

複数のスレッド上から同時に TSCAcceptor クラスの同じインスタンスを内部参照（アクセス）できます。

TSCAdm（Java）

TSCAdm はシステム提供クラスです。

TSCAdm は，アプリケーションプログラムの初期化処理，TSCClient の取得，および TSCServer の取得をするクラスです。

形式

```
public class TSCAdm
{
  public static final int Direct=0;
  public static final int Regulator=1;

  public static void initServer(String[] args,
                       org.omg.CORBA.ORB orb);
  public static void initClient(String[] args,
                     java.util.Properties option,
                       org.omg.CORBA.ORB orb);
  public static void initClient(Applet app,
                   java.util.Properties option,
                     org.omg.CORBA.ORB orb);

  public static void serverMainloop();
  public static void shutdown();

  public static void endServer();
  public static void endClient();

  public static TSCClient getTSCClient(TSCDomain tsc_domain,
                                       int way);
  public static TSCClient getTSCClient(TSCDomain tsc_domain);
  public static TSCServer getTSCServer(TSCDomain tsc_domain);

  public static void releaseTSCClient(TSCClient tsc_client);
  public static void releaseTSCServer(TSCServer tsc_server);

  public static int get_status();

};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCAdm;
```

メソッド

● public static void initServer(String[] args,
org.omg.CORBA.ORB orb)

項目	型・意味	
引数	String[] args	コマンド引数
	org.omg.CORBA.ORB orb	ORB のリファレンス
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException	

サーバアプリケーションの初期化処理を実行します。このメソッドは，プロセスで 1 回だけ発行できます。TSCAdm.endServer() メソッド，または TSCAdm.endClient() メソッドの発行によって終了処理したあとでも，このメソッドは発行できません。

このメソッドの args 引数にはプロセスの main() メソッドの第 1 引数をそのまま指定してください。プロセス開始時にコマンドラインで指定された情報を削除または変更して args 引数に指定すると，正しく動作しないことがあります。tscstartprc コマンドを使用して開始したサーバアプリケーションのコマンドラインには，tscstartprc コマンドに指定したコマンドライン引数がすべて渡されます。

● public static void initClient(String[] args,
java.util.Properties option,
org.omg.CORBA.ORB orb)

項目	型・意味	
引数	String[] args	コマンド引数
	java.util.Properties option	OTM の動作をカスタマイズするために設定できるプロパティ※
	org.omg.CORBA.ORB orb	ORB のリファレンス
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException	

注※

option を使用する場合，例えば次のように記述します。

```
java.util.Properties opt = new java.util.Properties();
opt.put("-TSCDomain","hitachi");
```

クライアントアプリケーションの初期化処理を実行します。

args 引数および option 引数に同じオプションがある場合，args 引数のオプションが有効になります。

このメソッドの args 引数にはプロセスの main() メソッドの第 1 引数をそのまま指定してください。プロセス開始時にコマンドラインで指定された情報を削除または変更して args 引数に指定すると，正しく動作しないことがあります。

このメソッドは，TSCAdm.endClient() メソッドの発行によって終了処理をした場合は再発行できますが，TSCAdm.endServer() メソッドの発行によって終了処理をしたあとは再発行できません。TSCAdm.endClient() メソッドの発行後にこのメソッドを再発行した場合，2 回目以降の発行で指定した argc 引数および option 引数の値は無効となり，初回の発行で指定した値が有効になります。ただし，このメソッドを複数回発行するとクライアントアプリケーションの性能に影響を与えるため，推奨できません。

● public static void initClient(Applet app,
java.util.Properties option,
org.omg.CORBA.ORB orb)

項目	型・意味	
引数	Applet app	このメソッドが実行されるアプレット ※1
	java.util.Properties option	OTM の動作をカスタマイズするために設定できるプロパティ ※2
	org.omg.CORBA.ORB orb	ORB のリファレンス
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException	

注※ 1

アプレットタグにオプションを記述する場合，例えば次のようにアプレットタグに記述します。

```
<param name=-TSCDomain value=hitachi>
```

注※ 2

option を使用する場合，例えば次のように記述します。

```
java.util.Properties opt = new java.util.Properties();
opt.put("-TSCDomain","hitachi");
```

Java アプレットの場合に，Applet クラスを使用したクライアントアプリケーションの初期化処理を実行します。

このメソッドは OTM - Client だけで使用できます。OTM で使用した場合は，例外が返ります。

また，このメソッドは，TSCAdm.endClient() メソッドの発行によって終了処理をした場合は再発行できますが，TSCAdm.endServer() メソッドの発行によって終了処理をしたあとは再発行できません。TSCAdm.endClient() メソッドの発行後にこのメソッドを再発行した場合，2 回目以降の発行で指定した option 引数の値は無効となり，初回の発行で指定した値が有効になります。ただし，このメソッドの複数回発行するとクライアントアプリケーションの性能に影響を与えるため，推奨できません。

● public static void serverMainloop()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException

リクエストを受信待ち状態にします。このメソッドはサーバアプリケーションでだけ発行できます。

● public static void shutdown()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException

リクエストの受信待ち状態を解除します。このメソッドはサーバアプリケーションでだけ発行できます。

● public static void endServer()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCNoMemoryException

サーバアプリケーションの終了処理を実行します。

このメソッドはプロセスで 1 回だけ発行できます。このメソッド発行後はそのプロセスで OTM の機能は使用しないでください

● public static void endClient()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCNoMemoryException

クライアントアプリケーションの終了処理を実行します。

このメソッドの発行後は，そのプロセスで OTM または OTM - Client の機能を使用できません。また，このメソッドで例外が発生した場合は，クライアントアプリケーションを終了させる必要があります。

● public static TSCClient getTSCClient(TSCDomain tsc_domain,
int way)

項目	型・意味	
引数	TSCDomain tsc_domain	TSC ドメインのリファレンス
	int way	接続経路
戻り値	TSCClient オブジェクト	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException TSCTransientException	

指定した TSCDomain を基に，リクエストする TSC デーモンの TSCClient のリファレンスを取得するメソッドです。

way には，接続経路として，次に示すどちらかを指定します。

- TSCAdm.Direct
  TSC デーモンに直結してリクエストします。
- TSCAdm.Regulator
  TSC レギュレータを経由してリクエストします。この場合，TSC レギュレータによってコネクションを集約します。

ファイル検索方式でマルチノードリトライ接続を実行する場合，このメソッドに指定した TSCDomain および way の組み合わせに一致する情報が，接続先情報ファイル中に記述されていなければなりません。一致する情報が接続先情報ファイルにない場合，TSCBadParamException 例外が発生します。なお，ファイル検索方式でマルチノードリトライ接続を実行するには，アプリケーションプログラムの開始時に，次に示すようにコマンドオプション引数を指定します。

- -TSCRetryReference に接続先情報ファイルを指定し，かつ，-TSCRetryWay に "0000" または "0001" を指定します。
- -TSCRetryReference に接続先情報ファイルを指定して，-TSCRetryWay の指定を省略します。この場合，TSCRetryWay には，"0000" が仮定されます。

● public static TSCClient getTSCClient(TSCDomain tsc_domain)

項目	型・意味	
引数	TSCDomain tsc_domain	TSCDomain のリファレンス
戻り値	TSCClient オブジェクトリファレンス	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException TSCTransientException	

指定した TSCDomain を基に，リクエストする TSC デーモンの TSCClient のリファレンスを取得するメソッドです。接続経路には，アプリケーションプログラムの開始時に，コマンドオプション引数 -TSCRequestWay に指定した値を使用します。

ファイル検索方式でマルチノードリトライ接続を実行する場合，このメソッドに指定した TSCDomain，およびコマンドオプション引数 -TSCRequestWay の組み合わせに一致する情報が，接続先情報ファイル中に記述されていなければなりません。一致する情報が接続先情報ファイルにない場合，TSCBadParamException 例外が発生します。なお，ファイル検索方式でマルチノードリトライ接続を実行するには，アプリケーションプログラムの開始時に，次に示すようにコマンドオプション引数を指定します。

- -TSCRetryReference に接続先情報ファイルを指定し，かつ，-TSCRetryWay に "0000" または "0001" を指定します。
- -TSCRetryReference に接続先情報ファイルを指定して，-TSCRetryWay の指定を省略します。この場合，-TSCRetryWay には，"0000" が仮定されます。

● public static TSCServer getTSCServer(TSCDomain tsc_domain)

項目	型・意味	
引数	TSCDomain tsc_domain	TSCDomain のリファレンス

項目	型・意味
戻り値	TSCServer オブジェクト
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException

指定した TSCDomain を基に，自サーバへリクエストを振り分ける TSC デーモンの TSCServer のリファレンスを取得するメソッドです。

● public static void TSCClient releaseTSCClient(TSCClient tsc_client)

項目	型・意味	
引数	TSCClient tsc_client	TSCClient オブジェクト
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException	

TSCClient を解放します。

● public static void releaseTSCServer(TSCServer tsc_server)

項目	型・意味	
引数	TSCServer tsc_server	TSCServer オブジェクト
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException	

TSCServer を解放します。

● public static int get_status()

項目	型・意味
戻り値	プロセスステータス

運用管理で管理するプロセスステータスを返します。プロセスステータスを示す定数を

表 5-3，表 5-4 に示します。

表 5-3　TSCAdm クラスで検出するクライアントアプリケーションのプロセスステータス（Java）

定数名	状態	内容
TSCAdm.Living	オンライン稼働中	TSCAdm.initClient を発行してから，終了要求を受け付けるまでの状態
TSCAdm.Dying	正常終了処理中	終了要求を受け付けてから，TSCAdm.endClient を発行するまでの状態
TSCAdm.Dead	終了	TSCAdm.initClient の発行以前，または TSCAdm.endClient の発行以降の状態

クライアントアプリケーションの状態遷移については，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のクライアントアプリケーションの状態の検出に関する説明を参照してください。

表 5-4　TSCAdm クラスで検出するサーバアプリケーションのプロセスステータス（Java）

定数名	状態	内容
TSCAdm.Living	正常開始処理中	TSCAdm.initServer を発行してから，TSCAdm.serverMainloop を発行するまでの状態
TSCAdm.Active	オンライン稼働中	TSCAdm.serverMainloop を発行してから，終了要求を受け付けるまでの状態
TSCAdm.Dying	正常終了処理中	終了要求を受け付けてから，TSCAdm.endServer を発行するまでの状態
TSCAdm.Dead	終了	TSCAdm.initServer の発行以前，または TSCAdm.endServer の発行以降の状態

サーバアプリケーションの状態遷移については，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のサーバアプリケーションの状態の検出に関する説明を参照してください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCAdm クラスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
initServer	できます。※
initClient	できます。※
serverMainloop	できます。※
shutdown	できます。※
endServer	できます。※
endClient	できます。※
getTSCClient	できます。
getTSCServer	できます。
releaseTSCClient	できます。
releaseTSCServer	できます。
get_status	できます。

注※

複数のスレッド上から同時に呼び出すことはできますが，有効となるのは一つの呼び出しだけです。

TSCClient（Java）

TSCClient はシステム提供クラスです。

TSCClient は，TSC デーモン中のクライアントアプリケーション管理部分を表すクラスです。クライアントアプリケーション側からの TSC ユーザオブジェクトの呼び出し要求は，TSCClient を経由して TSC デーモンに渡されます。

ユーザは，クライアントアプリケーションが TSC デーモンと接続するときに TSCClient を取得します。クライアントアプリケーションと TSC デーモンの接続には，TSC デーモンと直結する方法と，TSC レギュレータを経由する方法があります。次に TSCClient の特徴を示します。

- 属性として TSC ドメイン名称と TSC 識別子を持ちます。

TSC デーモンに直結する場合の TSCClient の取得

クライアントアプリケーションと TSC デーモン間の直結の接続は，クライアントアプリケーションプロセス内で TSCClient を最初に取得するときに確立されます。その後，同じ TSC デーモンに対して TSCClient を取得する場合は，その接続を共有します。逆に，取得したすべての TSCClient を解放すると接続が切断されます。

一つのクライアントアプリケーションから複数の TSC デーモンへ接続を確立することもできます。また，TSC ユーザオブジェクト呼び出し要求が，この接続を経由して TSC デーモンに渡される場合，並行して処理されます。

TSC レギュレータを経由する場合の TSCClient の取得

TSC レギュレータを経由する場合のクライアントアプリケーションと TSC デーモン間の接続は，TSCClient を取得するたびに確立されます。逆に，TSCClient を解放するたびに，割り当てられた接続が切断されます。

一つのクライアントアプリケーションから複数の TSC デーモンへ接続を確立することもできます。また，TSC ユーザオブジェクト呼び出し要求がこの一つの接続を経由して TSC デーモンに渡される場合，並行して処理されないで順番に処理されます。

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;

public class TSCClient
{
  public String getTSCDomainName();
  public String getTSCID();
};
```

メソッド

● public String getTSCDomainName()

項目	型・意味
戻り値	TSC ドメイン名称
例外	ありません。

TSC ドメイン名称を返します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● public String getTSCID()

項目	型・意味
戻り値	TSC 識別子
例外	ありません。

TSC 識別子を返します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

TSCClient の取得と解放

TSC デーモンと直結する場合，TSCClient を取得するには，TSCAdm.Direct を引数に指定して TSCAdm.getTSCClient を発行します。TSC レギュレータを経由する場合は，TSCAdm.Regulator を引数に指定して TSCAdm.getTSCClient を発行します。

TSCClient を解放するときは，TSCAdm.releaseTSCClient を発行します。TSCClient クラスのインスタンスへの内部参照（アクセス）があるときは解放できないため，内部参照（アクセス）をなくした状態で解放してください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCClient クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
getTSCDomainName	できます。
getTSCID	できます。

インスタンスへの内部参照（アクセス）規則

TSCClient クラスのインスタンスを解放したあと，このインスタンスを内部参照するインスタンスからのアクセスは，通信障害となります。

また，複数のスレッド上から同時にこのクラスの同じインスタンスを内部参照できます。

TSCContext（Java）

TSCContext はシステム提供クラスです。

TSCContext は，TSC ユーザプロキシを使用してクライアント側からサーバ側の TSC ユーザオブジェクトのメソッドを呼び出すとき，暗黙的にサーバ側に渡すユーザデータのコンテナクラスです。次に TSCContext の特徴を示します。

- ユーザデータを保持します。

TSCContext によるユーザデータの取得

TSCProxyObject を使用して，クライアント側からサーバ側の TSC ユーザオブジェクトのメソッドを呼び出すときに，TSCContext によって引数以外のユーザデータをサーバ側に渡すことができます。

クライアント側では，TSCProxyObject の _TSCContext によって TSCContext を取得し，送信したいユーザデータを設定します。サーバ側では，オブジェクトが呼び出されている間，TSCObject の _TSCContext を使用して，TSCContext を取得します。この TSCContext が保持しているユーザデータは，クライアント側の TSCContext に設定したユーザデータと同じ内容です。

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;

public class TSCContext
{
  public void setUserData(byte[] user_data);
  public byte[] getUserData();
};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCContext;
```

メソッド

● public void setUserData(byte[] user_data)

項目	型・意味	
引数	byte[] user_data	ユーザデータ
戻り値	ありません。	

ユーザデータを設定します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● public byte[] getUserData()

項目	型・意味
戻り値	ユーザデータ

ユーザデータを取得します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

TSCContext の生成

TSCContext クラスのインスタンスは，new オペレータで生成しないでください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCContext クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
setUserData	できません。
getUserData	できません。

TSCDomain（Java）

TSCDomain はシステム提供クラスです。

TSCDomain は，文字列の TSC ドメイン情報，および TSC 識別子を管理するホルダクラスです。

形式

```
public class TSCDomain
{
  public TSCDomain(String domain_name);
  public TSCDomain(String domain_name, String tscid);
};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCDomain;
```

コンストラクタ

● public TSCDomain(String domain_name)

項目	型・意味	
引数	String domain_name	TSC ドメイン名称
例外	TSCBadParamException TSCInitializeException	

このコンストラクタを使用する場合，TSC ドメイン名称で TSC ドメインを管理します。

domain_name に文字列を指定する場合は，先頭が "TSC" または "tsc" ではない 1 ～ 31 文字の英数字の文字列を指定してください。domain_name に null を指定する場合は，アプリケーションプログラムの開始時に指定するコマンドオプション引数 -TSCRetryReference の指定の有無によって管理する情報が異なります。

- コマンドオプション引数 -TSCRetryReference を指定しない場合
  initServer または initClient の args 内の "-TSCDomain" オプションの指定値を使用します。
- コマンドオプション引数 -TSCRetryReference を指定する場合
  コマンドオプション引数 -TSCRetryWay の指定内容によって動作が異なります。
  -TSCRetryWay の指定値と接続方式の関係については，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のマルチノードリトライ接続の接続対象に関する説明を参照してください。

● public TSCDomain(String domain_name,
String tscid)

項目	型・意味	
引数	String domain_name	TSC ドメイン名称
	String tscid	TSC 識別子
例外	TSCBadParamException TSCInitializeException	

このコンストラクタを使用する場合，TSC ドメイン名称と TSC 識別子で TSC ドメインを管理します。

domain_name または tscid に文字列を指定する場合は，先頭が "TSC" または "tsc" ではない 1 ～ 31 文字の英数字の文字列を指定してください。なお，tscid に IP アドレスを指定する場合は，ピリオド (.) も使用できます。domain_name または tscid に null を指定する場合は，アプリケーションプログラムの開始時に指定するコマンドオプション引数 -TSCRetryReference の指定の有無によって管理する情報が異なります。

- コマンドオプション引数 -TSCRetryReference を指定しない場合
  initServer または initClient の args 内の "-TSCDomain" オプションおよび "-TSCID" オプションの指定値を使用します。
- コマンドオプション引数 -TSCRetryReference を指定する場合
  コマンドオプション引数 -TSCRetryWay の指定内容によって動作が異なります。
  -TSCRetryWay の指定値と接続方式の関係については，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のマルチノードリトライ接続の接続対象に関する説明を参照してください。

TSCObject（Java）

TSCObject はシステム提供クラスです。

TSCObject は，OTM での（サーバ）オブジェクトの基本クラスおよびインタフェースです。

ユーザは TSCObject を継承させて，クライアント側にサービスを提供するクラスを定義します。また，サービスを提供するオブジェクトとして，その派生クラスのインスタンスを生成します。次に TSCObject の特徴を示します。

- 直接，TSCObject のインスタンスを生成できません。
- TSCObject を生成する，TSCObjectFactory の実装クラスを記述する必要があります。
- 属性としてインタフェース名称の列を持ちます。
- クライアント側から TSCProxyObject を使用して TSCObject を呼び出すときにユーザデータを TSCContext に指定すると，サーバ側で TSCContext から同じデータを取得できます。
- TSCRootAcceptor を生成するときに TSCThreadFactory を指定すると，TSCThread を取得できます。

TSCObject が提供するサービスのインタフェース名称

TSCObject が提供するインタフェースの種類は，TSCAcceptor のインタフェース名称の列で表されます。

TSCObject が単数のインタフェースを提供する場合，提供するインタフェースの種類は，単数のインタフェース名称で表されます。TSCObject が複数のインタフェースを提供する場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，複数のインタフェース名称が列で表されます。

TSCObject の呼び出し時の TSCContext の取得

TSCProxyObject を使用してクライアント側から TSCObject のメソッドを呼び出すときに，ユーザは引数以外のデータを TSCContext として送信できます。クライアント側で引数以外のデータを TSCContext として送信すると，サーバ側では TSCObject のサービス提供メソッドが呼び出されている間に _TSCContext を呼び出すことによって，クライアント側で指定した TSCContext を取得できます。

TSCObject の呼び出し時の TSCThread の取得

TSCObject の呼び出し時の TSCThread の取得は，TSCThreadFactory を引数として TSCRootAcceptor を生成する場合を前提にします。この場合，サーバ側で TSCObject のサービス提供メソッドが呼び出されている間に _TSCThread を呼び出すことによって，実行制御を持つスレッドに割り付けられている TSCThread を取得できます。

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;

public class TSCObject
{

  //インタフェース名称
  public String[] _TSCInterfaceName();

  //TSCコンテキスト
  public TSCContext _TSCContext();

  //TSCユーザスレッド
  public TSCThread _TSCThread();

};
```

メソッド

● public String[] _TSCInterfaceName()

項目	型・意味
戻り値	インタフェース名称列

インタフェース名称の列を取得します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● public TSCContext _TSCContext()

項目	型・意味
戻り値	呼び出し元で指定された TSCContext

TSCContext を取得します。クライアント側から呼び出すときに指定したものと同じ内容の TSCContext を取得できます。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● public TSCThread _TSCThread()

項目	型・意味
戻り値	TSC ユーザスレッド

このオブジェクトに割り当てられているスレッドに対応する TSC ユーザスレッドを返します。

戻り値の TSCThread のメモリ領域の管理責任は TSCObject クラスにあるので，解放しないでください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

TSCObject の派生クラスのインスタンスの生成

TSCObject の派生クラスは，new オペレータで生成します。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCObject クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
_TSCInterfaceName	できません
_TSCContext	できません
_TSCThread	できません
クライアント側からのオブジェクト呼び出し※	できません。

注※

このメソッドは OTM が呼び出します。

TSCObjectFactory（Java）

TSCObjectFactory はシステム提供インタフェースです。

TSCObjectFactory は，OTM が TSC ユーザオブジェクトをの管理を開始するとき，または管理対象から外すときに呼び出されるオブジェクトのインタフェースです。

ユーザは TSCObjectFactory を継承させて，TSC ユーザオブジェクトを生成，または削除するクラスを定義します。また，TSC ユーザオブジェクトのファクトリとして，派生クラスのインスタンスを生成します。次に TSCObjectFactory の特徴を示します。

- 直接，TSCObjectFactory のインスタンスを生成できません。
- ユーザは TSCObjectFactory クラスを継承して，実装クラスを記述する必要があります。

OTM からの呼び出しによる TSCObject の管理

OTM は，TSCObjectFactory の create を呼び出すことで，返される TSCObject の管理を開始します。逆に，該当する TSCObject を引数に TSCObjectFactory の destroy を呼び出すことで，TSCObject を管理対象から外します。

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;

public interface TSCObjectFactory
{
  public TSCObject create();
  public void destroy(TSCObject tsc_object);
};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCObjectFactory;
```

コールバックメソッド

● public TSCObject create()

項目	型・意味
戻り値	管理対象の TSC ユーザオブジェクト
例外	各種 TSCSystemException

TSCObject を返します。TSC ユーザオブジェクトを生成するコードを記述できます。OTM がこのメソッドを呼び出した結果，返された TSCObject が管理対象となります。

管理対象とする TSCObject の管理責任は OTM にあるので，ユーザはなるべくアクセス

しないでください。なお，OTM は，複数のスレッド上から同時にこのメソッドを呼び出すので，ユーザはマルチスレッド環境に対応するリエントラントなコードを記述する必要があります。

また，create 呼び出しに失敗した場合は，各種 TSCSystemException によって通知する形でコードを記述してください。

● public void destroy(TSCObject tsc_object)

項目	型・意味	
引数	TSCObject tsc_object	管理対象から外す TSC ユーザオブジェクト
例外	ありません。	

TSCObject を消去する前の処理のコードを記述できます。OTM が TSC ユーザオブジェクトを管理対象から外すとき，該当する TSCObject を引数に指定して，このメソッドを呼び出します。

管理対象から外された TSCObject の管理責任はユーザにあります。なお，OTM は，複数のスレッド上から同時にこのメソッドを呼び出すので，ユーザはマルチスレッド環境に対応するリエントラントなコードを記述する必要があります。

また，このメソッドから通知した例外は無視されます。

TSCObjectFactory または派生クラスの生成

TSCObjectFactory，またはその派生クラスは，new オペレータで生成します。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCObjectFactory クラスのインスタンスのメソッドを呼び出す規則を次に示します。なお，これらのメソッドは，OTM が呼び出します。

メソッド	複数のスレッド上からの同時呼び出し
create	できます。
destroy	できます。

TSCProxyObject（Java）

TSCProxyObject はシステム提供クラスです。

TSCProxyObject は TSCObject の代理クラスです。TSCProxyObject を呼び出すと，OTM のスケジューリング機構を経由して TSCObject が呼び出されます。

ユーザは，サーバアプリケーションの TSC ユーザオブジェクトが提供するサービスを利用するときに，TSCProxyObject クラスのインスタンスを生成して呼び出します。次に TSCProxyObject の特徴を示します。

- TSCProxyObject を使用して利用できるサービスを TSC **サービス識別子**で表現します。TSC サービス識別子は，**インタフェース名称**と TSC **アクセプタ名称**から構成されます。ただし，TSC アクセプタ名称がない場合もあります。
- 属性として，タイムアウト値（呼び出し時の監視時間）とプライオリティ値（メソッド呼び出し時の優先順位）を持ちます。
- TSCContext を登録できます。

TSC サービス識別子によるサービスの識別

TSC サービス識別子の構成

TSCProxyObject を使用して利用するサービスの種類は，TSC サービス識別子によって表されます。TSC サービス識別子は，インタフェース名称と TSC アクセプタ名称から構成されます。

- **TSCProxyObject のインタフェース名称**
  TSCProxyObject のインタフェース名称は，TSCProxyObject を使用して呼び出すサービスのインタフェースの種類を表します。
- **TSCProxyObject の TSC アクセプタ名称**
  TSCProxyObject の TSC アクセプタ名称も，TSCProxyObject を使用して呼び出すサービスのインタフェースの種類を表します。ただし，同じインタフェースを提供するサービスで，実装内容の違いを識別するために使用します。したがって，TSC アクセプタ名称を設定しないで，"TSC アクセプタ名称なし " とすることもできます。

サービスの種類の表現方法

TSCProxyObject によって利用するサービスの種類は，TSC サービス識別子によって表されます。TSC サービス識別子は，インタフェース名称と TSC アクセプタ名称から構成されます。

- **TSCAcceptor に TSC アクセプタ名称が設定されていない場合**
  TSCProxyObject が呼び出すサービスのインタフェースの種類は，次のように表されます。

TSCアクセプタ名称なしTSCサービス識別子

- **TSCAcceptor に TSC アクセプタ名称が設定されている場合**
  TSCProxyObject が呼び出すサービスの種類は，次のように表されます。

  TSCアクセプタ名称ありTSCサービス識別子

サービスの種類の表現例

- **TSCProxyObject のインタフェース名称が "ABC" で，TSC アクセプタ名称がない場合**
  次のように表される場合，TSCProxyObject は，"ABC::" への要求メッセージを生成し，"ABC::" としてサービスを提供している TSCObject と TSCAcceptor を呼び出すことができます。

  ```
  "ABC::"
  ```

  サーバアプリケーション側で，TSCProxyObject に "ABC::" と指定してサービスを呼び出した場合，その要求を受け付ける TSCObject と TSCAcceptor は，次に示すどちらかです。
  - インタフェース名称 "ABC:" だけ指定された TSCAcceptor，およびその TSCAcceptor に管理される TSCObject
  - インタフェース名称 "ABC" と任意の TSC アクセプタ名称が指定された TSCAcceptor，およびその TSCAcceptor に管理される TSCObject

  つまり，TSCProxyObject でインタフェース名称だけ指定して呼び出した場合，サーバ側では，同じインタフェース名称を持つ TSCAcceptor に管理されるすべての TSCObject が，TSC アクセプタ名称の値に関係なく呼び出されます。
- **TSCProxyObject のインタフェース名称が "ABC" で，TSC アクセプタ名称が "abc" の場合**
  次のように表される場合，TSCProxyObject は "ABC::abc" への要求メッセージを生成し，"ABC::abc" としてサービスを提供している TSCObject と TSCAcceptor を呼び出すことができます。

  ```
  "ABC::abc"
  ```

  サーバアプリケーション側で，TSCProxyObject に "ABC::abc" と指定してサービスを呼び出した場合，その要求を受け付ける TSCObject と TSCAcceptor は，インタフェース名称 "ABC" と TSC アクセプタ名称 "abc" が指定された TSCAcceptor，およびその TSCAcceptor に管理される TSCObject です。
  つまり，TSCProxyObject にインタフェース名称と TSC アクセプタ名称を指定して呼び出した場合，サーバ側では，同じインタフェース名称と同じ TSC アクセプタ名称を持つ TSCAcceptor に管理される TSCObject が呼び出されるため，TSC アクセプタ名称によって呼び出す TSCObject を選択できます。

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;
```

```
public interface TSCProxyObject
{

  public String _TSCInterfaceName();
  public String _TSCAcceptorName();

  public int _TSCTimeout();
  public void _TSCTimeout(int timeout);

  public int _TSCPriority();
  public void _TSCPriority(int priority);
  public TSCContext _TSCContext();

};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCProxyObject;
```

メソッド

● public String _TSCInterfaceName()

項目	型・意味
戻り値	インタフェース名称

インタフェース名称を取得します。

● public String _TSCAcceptorName()

項目	型・意味
戻り値	TSC アクセプタ名称

TSC アクセプタ名称を取得します。

● public int _TSCTimeout()

項目	型・意味（単位）
戻り値	タイムアウト時間（秒）

タイムアウト値（呼び出し時の監視時間）を取得します。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● public void _TSCTimeout(int timeout)

項目	型・意味（単位）	
引数	int timeout	タイムアウト時間（秒）

項目	型・意味（単位）
戻り値	ありません。
例外	TSCBadParamException

タイムアウト値（呼び出し時の監視時間）を秒単位で設定します。"0" を指定した場合，時間監視をしません。監視時間は，メソッド呼び出しごとに変更できます。

アプリケーションプログラムの開始時に -TSCTimeOut オプションを指定しない場合は，監視時間のデフォルト値は "180"（秒）です。-TSCTimeOut オプションを指定する場合は，監視時間のデフォルト値は -TSCTimeOut オプションの指定値になります。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● public int _TSCPriority()

項目	型・意味
戻り値	プライオリティ値

プライオリティ値（メソッド呼び出し時の優先順位）を取得します。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● public void _TSCPriority(int priority)

項目	型・意味	
引数	int priority	プライオリティ値
戻り値	ありません。	
例外	TSCBadParamException	

プライオリティ値（メソッド呼び出し時の優先順位）を設定します。

priority に 1 ～ 8 の値を指定することで，キューイング取り出し時の優先順位を変更できます。priority に指定する値が小さいほど優先度は高くなります。プライオリティ値はリクエスト単位に変更できます。

アプリケーションプログラムの開始時に -TSCRequestPriority オプションを指定しない場合は，プライオリティ値のデフォルト値は "4" です。-TSCRequestPriority オプションを指定する場合は，プライオリティ値のデフォルト値は -TSCRequestPriority オプションの指定値になります。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● TSCContext _TSCContext()

項目	型・意味
戻り値	TSC コンテキスト

TSCContext を取得します。

TSCProxyObject または派生クラスの生成

TSCProxyObject，またはその派生クラスは，new オペレータで生成します。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCProxyObject クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
_TSCInterfaceName	○
_TSCAcceptorName	○
_TSCTimeout()	○
_TSCTimeout(TSCInt)	×
_TSCPriority()	○
_TSCPriority(TSCInt)	×
_TSCContext	×
クライアント側からのオブジェクト呼び出し	×

（凡例）
○：できます。
×：できません。

インスタンスの内部参照（アクセス）規則

TSCProxyObject クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

メソッド	複数のスレッド上からの内部参照
_TSCInterfaceName	ありません。
_TSCAcceptorName	ありません。
_TSCTimeout()	ありません。
_TSCTimeout(TSCInt)	ありません。
_TSCPriority()	ありません。
_TSCPriority(TSCInt)	ありません。

メソッド	複数のスレッド上からの内部参照
_TSCContext	ありません。
クライアント側からのオブジェクト呼び出し	生成時に指定した TSCClient 型のインスタンス

TSCRootAcceptor（Java）

TSCRootAcceptor はシステム提供クラスです。

TSCRootAcceptor は，サーバオブジェクトの実行空間を表現するオブジェクトです。クライアント側からの TSC ユーザオブジェクト呼び出し要求を受け付けて，適切な TSC ユーザアクセプタに振り分けます。また，パラレルカウント（常駐するスレッド数）に合わせてスレッドを管理します。

ユーザは，クライアント側にサービスを提供する TSC ユーザオブジェクトの実行空間を構築するときに，サーバアプリケーション内で TSCRootAcceptor クラスのインスタンスを生成します。次に TSCRootAcceptor の特徴を示します。

- 複数の TSCAcceptor を登録できます。
- スレッドの生成や削除などをして，スレッドを管理します。
- 属性として TSC **ルートアクセプタ状態**を持ちます。TSC ルートアクセプタ状態には active 状態と non-active 状態の 2 種類があります。
- TSC **ルートアクセプタ登録名称**を指定して active 状態に遷移できます。
- 属性として**パラレルカウント**（常駐するスレッド数）を持ちます。
- TSCThreadFactory を登録できます。
- 属性として**スケジュール用キューの長さ**を持ちます。

TSCAcceptor の登録

TSCObject の実行空間の構成

TSCRootAcceptor の実行空間で TSCObject を実行させる場合，例えば，
TSCRootAcceptor の管理するスレッド上で TSCObject を実行させる場合，その
TSCObject に対応する TSCAcceptor を TSCRootAcceptor に登録します。
TSCRootAcceptor には複数の TSCAcceptor を登録できます。

提供するサービスの管理

TSCRootAcceptor には複数の TSCAcceptor を登録できます。TSCRootAcceptor は，各 TSCAcceptor の複数の TSC サービス識別子を集め，かつ，重複を削除したものを属性として管理します。TSCRootAcceptor が提供できるサービスの種類は，この TSC サービス識別子によって表現されます。つまり，TSC サービス識別子の列で表されるサービスを提供することになります。

TSC ルートアクセプタ状態

TSC ルートアクセプタ状態ごとのスレッドの管理方式

TSC ルートアクセプタ状態には active 状態と non-active 状態の 2 種類があります。各状態の遷移中の場合も含めて，それぞれの状態のときのスレッドの管理方式を次に示し

ます。

- non-active **状態**
  non-active 状態のオブジェクトが管理するスレッドはありません。
  non-active 状態のときは，TSCAcceptor の登録および削除ができます。TSCAcceptor の登録時には，登録識別子が返されます。TSCAcceptor を削除するときには，その登録識別子を指定します。
  TSCRootAcceptor を生成した時点では non-active 状態です。
- non-active **状態から** active **状態に遷移中**
  パラレルカウント数と同数のスレッドを生成します。また，登録されているすべての TSCAcceptor にオブジェクト管理開始通知を出します。すべての TSCAcceptor がオブジェクトの管理を開始したあと，クライアント側からの TSC ユーザオブジェクト呼び出し要求を受け付けることができます。
- active **状態**
  クライアント側にサービスを提供できる状態です。クライアント側から TSC ユーザオブジェクト呼び出し要求を受け取ると，適切な TSCAcceptor に振り分けます。
  non-active 状態から active 状態に遷移するときに生成されたスレッドは TSCRootAcceptor に管理されています。ユーザは，TSC 経由以外でこれらの TSC ユーザオブジェクトにアクセスしないでください。また，active 状態のときは，TSCAcceptor の登録および削除はできません。
- active **状態から** non-active **状態に遷移中**
  登録されているすべての TSCAcceptor にオブジェクト管理終了通知を出します。また，TSCRootAcceptor の管理下にあるスレッドを削除します。

active **状態での障害通知**

サーバアプリケーション内や TSC デーモンとの接続に障害が発生してスレッドが存続できなくなる場合，そのスレッド上でオブジェクト管理終了通知を TSCAcceptor に渡します。その後，そのスレッドを削除します。なお，このときはまだ active 状態です。

障害が解除されると，再度，スレッドを生成します。その後，そのスレッド上でオブジェクト管理開始通知を TSCAcceptor に渡します。つまり，障害が発生してから解除されるまでの間は，TSCRootAcceptor は，スレッド数がパラレルカウント（常駐するスレッド数）以下の状態で存続します。

TSC **ルートアクセプタ登録名称**

TSCRootAcceptor が active 状態に遷移すると，TSCRootAcceptor と関連づけられている TSCServer に TSC ルートアクセプタ登録名称が登録されます。以降，クライアント側のメソッド呼び出し要求が TSCRootAcceptor に振り分けられるようになります。

TSCRootAcceptor を active 状態に遷移させるときに，TSC ルートアクセプタ登録名称を指定することもできます。activate の呼び出し時に，TSC ルートアクセプタ登録名称を指定する場合と指定しない場合について，それぞれ次に示します。

- TSC **ルートアクセプタ登録名称を指定して** activate **を呼び出した場合**
  TSC ルートアクセプタ登録名称を "abc" とすると，関連づけられている TSCServer に "abc" として，登録されます。
- TSC **ルートアクセプタ登録名称を指定しないで** activate **を呼び出した場合**
  関連づけられている TSCServer に，デフォルトの TSC ルートアクセプタ登録名称で登録されます。デフォルトの TSC ルートアクセプタ登録名称は "default" ですが，サーバアプリケーションの開始時に指定するコマンドオプション引数 -TSCRootAcceptor によって変更できます。

また，同じ TSC ルートアクセプタ登録名称で，TSCRootAcceptor を同じ TSC デーモンに登録できます。同じ TSC ルートアクセプタ登録名称で登録した場合，スケジュール用キュー（配送機構）が共有されます。ただし，スケジュール用キューを共有する場合，登録する TSCRootAcceptor 間で提供できるサービス内容が一致している必要があります。つまり，同じ TSC ルートアクセプタ登録名称で登録する TSCRootAcceptor は，同じ TSC サービス識別子の列を持っている必要があります。

TSC **ユーザスレッドファクトリによる** TSC **ユーザスレッドの管理**

TSCThreadFactory を引数として TSCRootAcceptor を生成した場合を前提にします。この場合，TSCRootAcceptor は，non-active 状態から active 状態に遷移する過程でスレッドを生成したあと，TSCThreadFactory の create を呼び出して，戻り値である TSCThread をそのスレッドに割り当てます。また，active 状態から non-active 状態に遷移する過程で，スレッドごとに割り当てた TSCThread を引数に TSCThreadFactory の destroy を呼び出したあと，スレッドを削除します。

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;

public class TSCRootAcceptor
{

  public static TSCRootAcceptor create(TSCServer tsc_server);

  public static TSCRootAcceptor create(TSCServer tsc_server,
                                       TSCThreadFactory tsc_thr_fact);


  //TSCAcceptorの追加
  public int registerAcceptor(TSCAcceptor tsc_acpt);

  //TSCAcceptorの削除
  public void cancelAcceptor(int reg_id);

  //パラレルカウントの設定
  public void setParallelCount(int p_count);
  public int getParallelCount();

  //TSCRootAcceptorの活性化
```

```
  public int activate();
  public int activate(String rt_acpt_req_name);

  //TSCRootAcceptorの非活性化
  public int deactivate();

  //スケジュール用キュー長の設定
  public void setQueueLength(int length);
  public int getQueueLength();
};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCRootAcceptor;
```

メソッド

● public static TSCRootAcceptor create(TSCServer tsc_server)

項目	型・意味	
引数	TSCServer tsc_server	接続した TSCServer
戻り値	生成された TSCRootAcceptor	
例外	TSCBadParamException TSCNoMemoryException	

tsc_server と関連づけられた，TSCRootAcceptor オブジェクトを生成します。

引数で渡す TSCServer の解放責任はユーザにあるので，適切な状態のときに解放してください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● public static TSCRootAcceptor create(TSCServer tsc_server,
TSCThreadFactory tsc_thr_fact)

項目	型・意味	
引数	TSCServer tsc_server	接続した TSCServer
	TSCThreadFactory tsc_thr_fact	TSC ユーザスレッドファクトリ
戻り値	生成された TSCRootAcceptor	
例外	TSCBadParamException TSCNoMemoryException	

tsc_server と関連づけられた，tsc_thr_fact を保持する TSCRootAcceptor オブジェクトを生成します。

引数で渡す TSCServer の解放責任はユーザにあるので，適切な状態のときに解放してください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● public int registerAcceptor(TSCAcceptor tsc_acpt)

項目	型・意味	
引数	TSCAcceptor tsc_acpt	登録する TSCAcceptor
戻り値	TSC ユーザアクセプタの登録識別子（正数）	
例外	TSCBadParamException TSCNoPermissionException	

TSCAcceptor を登録します。active の状態のときは登録できません。

● public void cancelAcceptor(int reg_id)

項目	型・意味	
引数	int reg_id	削除する TSC の登録識別子
戻り値	ありません。	
例外	TSCBadParamException TSCNoPermissionException	

TSCAcceptor の登録を削除します。reg_id には registerAcceptor で戻された値を指定してください。activate 状態のときは登録を削除できません。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● public void setParallelCount(int p_count)

項目	型・意味	
引数	int p_count	パラレルカウント
戻り値	ありません。	
例外	TSCBadParamException TSCNoPermissionException	

パラレルカウント（常駐するスレッド数）を設定します。

サーバアプリケーションの開始時にコマンドオプション引数 -TSCParallelCount を指定しない場合，パラレルカウントのデフォルト値は "1" です。コマンドオプション引数 -TSCParallelCount を指定する場合は，パラレルカウントのデフォルト値はその指定値となります。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● public int getParallelCount()

項目	型・意味
戻り値	パラレルカウント

パラレルカウント（常駐するスレッド数）を取得します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● public int activate()

項目	型・意味
戻り値	常に 0
例外	TSCBadInvOrderException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException ユーザが通知する各種例外

デフォルトの TSC ルートアクセプタ登録名称で，active 状態に遷移します。

サーバアプリケーションの開始時に指定するコマンドオプション引数 -TSCRootAcceptor を指定しない場合，TSC ルートアクセプタ登録名称のデフォルト値は "default" です。コマンドオプション引数 -TSCRootAcceptor を指定した場合，TSC ルートアクセプタのデフォルト値はその指定値となります。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● public int activate(String rt_acpt_req_name)

項目	型・意味	
引数	String rt_acpt_req_name	TSC ルートアクセプタの登録名称
戻り値	常に 0	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException ユーザが通知する各種例外	

指定した TSC ルートアクセプタ登録名称で，active 状態に遷移します。rt_acpt_name に文字列を指定する場合，1 ～ 31 文字で指定してください。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● public int deactivate()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException

deactive 状態に遷移します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● public void setQueueLength(int_length)

項目	型・意味	
引数	int length	スケジュール用キューの長さ
戻り値	ありません。	
例外	TSCBadParamException TSCNoPermissionException	

生成するスケジュール用キューの長さを指定します。指定できる範囲は 1 ～ 32767 です。active 状態のときは指定できません。

このメソッドでスケジュール用キューの長さを指定しない場合，生成されるスケジュール用キューの長さは，tscstartprc コマンドまたはサーバアプリケーションの開始コマンドの -TSCQueueLength オプションで指定した長さになります。スケジュール用キューを共有する場合，すでに生成されているスケジュール用キューの長さが有効になります。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● public int getQueueLength()

項目	型・意味
戻り値	スケジュール用キューの長さ

non-active 状態の場合は，setQueueLength() メソッド，または tscstartprc コマンドもしくはサーバアプリケーションの開始コマンドの -TSCQueueLength オプションで指定したスケジュール用キューの長さを取得します。setQueueLength() メソッドが未発行で，かつ tscstartprc コマンドまたはサーバアプリケーションの開始コマンドの -TSCQueueLength オプションが指定されていないときの戻り値は "0" となります。

active 状態の場合は，現在有効になっているスケジュール用キューの長さを取得します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

TSCRootAcceptor の生成

TSCServer を引数に指定した TSCRootAcceptor の create で生成します。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCRootAcceptor クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
create(TSCServer)	○
create(TSCServer, TSCThreadFactory)	○
registerAcceptor	×
cancelAcceptor	×
setParallelCount	×
getParallelCount	○
activate()	×
activate(String)	×
deactivate()	×
setQueueLength	×
getQueueLength	○

（凡例）

○：できます。

×：できません。

インスタンスの内部参照（アクセス）規則

TSCRootAcceptor クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

メソッド	内部参照
registerAcceptor	ありません。
cancelAcceptor	ありません。
setParallelCount	ありません。
getParallelCount	ありません。
activate()	生成時に指定された TSCServer 型のインスタンス 登録されている TSCAcceptor 型のインスタンス
activate(String)	生成時に指定された TSCServer 型のインスタンス 登録されている TSCAcceptor 型のインスタンス

メソッド	内部参照
deactivate()	生成時に指定された TSCServer 型のインスタンス 登録されている TSCAcceptor 型のインスタンス
setQueueLength	ありません。
getQueueLength	ありません。
active 状態	registerAcceptor の引数で指定された TSCAcceptor 型のインスタンス（TSCRootAcceptor に登録されている TSCAcceptor 型のインスタンス）

インスタンスの公開メソッド呼び出し規則

TSCRootAcceptor クラスのインスタンスがほかのクラスのインスタンスの公開メソッドを呼び出す規則を次に示します。

メソッド	公開メソッド呼び出し
activate()	生成時に指定された TSCThreadFactory 型のインスタンス
activate(String)	生成時に指定された TSCThreadFactory 型のインスタンス
deactivate()	生成時に指定された TSCThreadFactory 型のインスタンス

インスタンスへの内部参照（アクセス）規則

複数のスレッド上から同時に，TSCRootAcceptor クラスの同じインスタンスを内部参照できます。

TSCServer（Java）

TSCServer はシステム提供クラスです。

TSCServer は，TSC デーモン中のサーバアプリケーション管理部分を参照するクラスです。サーバアプリケーション側の機能操作の要求は，TSCServer を経由して TSC デーモンに渡されます。また，TSC ユーザプロキシを使用したクライアントアプリケーション側からの TSC ユーザオブジェクト呼び出し要求を受けて，TSC ルートアクセプタに振り分けます。

ユーザはサーバアプリケーションが TSC デーモンと接続するときに，TSCServer クラスのインスタンスを取得します。次に TSCServer の特徴を示します。

- 属性として TSC ドメイン名称と TSC 識別子を持ちます。

TSCServer と接続

サーバアプリケーションと TSC デーモン間の接続は，サーバアプリケーションプロセス内で TSCServer を最初に取得するときに確立されます。その後，同じ TSC デーモンに対して TSCServer を取得する場合は，その接続を共有します。逆に，取得したすべての TSCServer を解放すると接続が切断されます。

一つのサーバアプリケーションから複数の TSC デーモンへ接続を確立することもできます。また，サーバアプリケーション側の機能操作の要求が，この接続を経由して TSC デーモンに渡される場合，並行して処理されないで順番に処理されます。TSC デーモンからのオブジェクト呼び出し要求が，この接続を経由してサーバアプリケーションに配送される場合は，並行して処理されます。

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;

public class TSCServer
{
  public String getTSCDomainName();
  public String getTSCID();

};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCServer;
```

メソッド

● public String getTSCDomainName()

項目	型・意味
戻り値	TSC ドメイン名称

TSC ドメイン名称を返します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

● public String getTSCID()

項目	型・意味
戻り値	TSC 識別子

TSC 識別子を返します。

なお，このメソッドを複数のスレッド上から同時に呼び出すことができます。

TSCServer の取得と解放

TSCServer は TSCAdm の getTSCServer で取得し，TSCAdm の releaseTSCServer で解放します。TSCServer クラスのインスタンスへの内部参照（アクセス）があるときは解放できないため，TSCServer クラスのインスタンスへの内部参照（アクセス）をなくした状態で解放してください。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCServer クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
getTSCDomainName	できます。
getTSCID	できます。

インスタンスの内部参照（アクセス）規則

TSCServer クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

タイミング	内部参照
getTSCDomainName	ありません。

タイミング	内部参照
getTSCID	ありません。
関連づけがある TSCRootAcceptor が active 状態	関連づけがある TSCRootAcceptor 型のインスタンス

なお，TSCServer クラスのインスタンスを解放したあと，このインスタンスを内部参照するインスタンスからのアクセスは，通信障害となります。また，複数のスレッド上から同時に，TSCServer クラスの同じインスタンスを内部参照できます。

TSCSessionProxy（Java）

TSCSessionProxy はシステム提供クラスです。

TSCSessionProxy は，TSCObject の代理クラスです。TSCSessionProxy を呼び出すと，OTM のスケジューリング機構を経由してステートフルに TSCObject が呼び出されます。TSCSessionProxy の TSCProxyObject との違いを次に示します。

TSCSessionProxy の特徴

- TSCProxyObject の持つ属性に加え，セション呼び出しインターバル監視時間（呼び出しと呼び出しの間の監視時間）が属性に追加されます。
- _TSCStart() メソッドでセションを確立します。
  セションを確立すると，TSCSessionProxy とサーバアプリケーションのインスタンスを対応づけます。_TSCStart() メソッド発行後は，_TSCStop() メソッドの発行まで，対応づけたインスタンスにリクエストを要求します。対応づけられるサーバアプリケーションのインスタンスは TSCObject です。TSCObject と対応づけた場合は，次のことに注意してください。
  - TSCObject のインスタンスを保持しているスレッドも対応づける。
  - TSCObject のインスタンスはセションを確立していないほかのクライアントアプリケーションからのリクエストを受け付けられない。
  - すべての TSCObject のインスタンスが対応づけられた場合は，新しいクライアントアプリケーションからのリクエストを受け付けられない。
- _TSCStop() メソッドを発行すると，TSCSessionProxy とサーバアプリケーションのインスタンスとのセションを解放します。
- コンストラクタに指定する TSC アクセプタ名称は，_TSCStart() メソッド発行時に対応づけるインスタンスを決定するために使用されます。
- TSC アクセプタ名称を指定していないコンストラクタで生成した場合は，_TSCStart() メソッドで任意のインスタンスに対応づけ，対応づけた TSC アクセプタ名称を _TSCStop() メソッド発行時まで引き継ぎます。

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;
public interface TSCSessionProxy
{
   public void _TSCStart();
   public void _TSCStop();
   public int _TSCSessionInterval();
   public void _TSCSessionInterval(int _session_interval);
   public String _TSCInterfaceName();
   public String _TSCAcceptorName();
   public int _TSCTimeout();
```

```
    public void _TSCTimeout(int _timeout);
    public int _TSCPriority();
    public void _TSCPriority(int _priority);
    public TSCContext _TSCContext();
};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCSessionProxy;
```

メソッド

● public void _TSCStart()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResponseException TSCObjectNotExistException TSCTransientException

セションを開始します。

セションを開始すると TSC ユーザプロキシとサーバアプリケーションのインスタンスを対応づけます。また，アクセプタ名称を指定していないコンストラクタでインスタンスを生成した場合は，_TSCStart() メソッドで対応づけたインスタンスのアクセプタ名称を _TSCStop() メソッドの発行時まで引き継ぎます。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

_TSCStart() メソッドが正常終了した場合は，セション呼び出しが成功しても失敗しても，必ず _TSCStop() メソッドを発行してください。

● public void _TSCStop()

項目	型・意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResponseException TSCTransientException

セションを解放します。

セションを解放すると TSC ユーザプロキシとサーバアプリケーションのインスタンスの対応づけも解放されます。

なお，このメソッドを複数のスレッド上から同時に呼び出すことはできません。

● public int _TSCSessionInterval()

項目	型・意味（単位）
戻り値	セション呼び出しインターバル監視時間（秒）

セション呼び出しインターバル監視時間を取得します。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● public void _TSCSessionInterval(int _session_interval)

項目	型・意味（単位）	
引数	int _session_interval	セション呼び出しインターバル監視時間（秒）
戻り値	ありません。	
例外	TSCBadParamException	

セション呼び出しインターバル監視時間を秒単位で設定します。メソッド呼び出しごとに変更できます。

アプリケーションプログラムの開始時に -TSCSessionInterval オプションを指定しない場合は，監視時間のデフォルト値は "180"（秒）です。-TSCSessionInterval オプションを指定する場合は，監視時間のデフォルト値は -TSCSessionInterval オプションの指定値になります。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● public String _TSCInterfaceName()

項目	型・意味
戻り値	インタフェース名称

インタフェース名称を取得します。

● public String _TSCAcceptorName()

項目	型・意味
戻り値	TSC アクセプタ名称

TSC アクセプタ名称を取得します。

取得する TSC アクセプタ名称を次に示します。

- セション呼び出し中の場合
  セション呼び出し中のサーバアプリケーションのアクセプタ名称
- セション呼び出し中でない場合
  TSCSessionProxy 生成時に指定したアクセプタ名称

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● public int _TSCTimeout()

項目	型・意味（単位）
戻り値	タイムアウト時間（秒）

タイムアウト値（呼び出し時の監視時間）を取得します。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● public void _TSCTimeout(int _timeout)

項目	型・意味（単位）	
引数	int _timeout	タイムアウト時間（秒）
戻り値	ありません。	
例外	TSCBadParamException	

タイムアウト値（呼び出し時の監視時間）を秒単位で設定します。"0" を指定した場合は，時間監視をしません。監視時間は，メソッド呼び出しごとに変更できます。

アプリケーションプログラムの開始時に -TSCTimeOut オプションを指定しない場合は，監視時間のデフォルト値は "180"（秒）です。-TSCTimeOut オプションを指定する場合は，監視時間のデフォルト値は -TSCTimeOut オプションの指定値になります。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

タイムアウト値が有効になるメソッドを次に示します。

- _TSCStart() メソッド
- クライアント側からのオブジェクト呼び出しメソッド
- _TSCStop() メソッド

● public int _TSCPriority()

項目	型・意味
戻り値	プライオリティ値

プライオリティ値（メソッド呼び出し時の優先順位）を取得します。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● public void _TSCPriority(int _priority)

項目	型・意味	
引数	int _priority	プライオリティ値
戻り値	ありません。	
例外	TSCBadParamException	

プライオリティ値（メソッド呼び出し時の優先順位）を設定します。

_priority に 1 ～ 8 の値を指定することで，キューイング取り出し時の優先順位を変更できます。_priority に指定する値が小さいほど優先度は高くなります。プライオリティ値はリクエスト単位に変更できます。

アプリケーションプログラムの開始時に -TSCRequestPriority オプションを指定しない場合は，プライオリティ値のデフォルト値は "4" です。-TSCRequestPriority オプションを指定する場合は，プライオリティ値のデフォルト値は -TSCRequestPriority オプションの指定値になります。

このメソッドを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

プライオリティ値が有効になるのは _TSCStart() メソッドだけです。

● public TSCContext _TSCContext()

項目	型・意味
戻り値	TSC コンテキスト

TSC コンテキストを取得します。

TSCSessionProxy または派生クラスの生成

TSCSessionProxy，またはその派生クラスは，new オペレータで生成します。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCSessionProxy クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
_TSCStart()	×
_TSCStop()	×
_TSCSessionInterval()	○
_TSCSessionInterval(int)	×
_TSCInterfaceName()	○

メソッド	複数のスレッド上からの同時呼び出し
_TSCAcceptorName()	○
_TSCTimeout()	○
_TSCTimeout(int)	×
_TSCPriority()	○
_TSCPriority(int)	×
_TSCContext	×
クライアント側からのオブジェクト呼び出し	×

（凡例）
○：できます。
×：できません。

インスタンスの内部参照（アクセス）規則

TSCSessionProxy クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

メソッド	複数のスレッド上からの内部参照
_TSCStart()	ありません。
_TSCStop()	ありません。
_TSCSessionInterval()	ありません。
_TSCSessionInterval(int)	ありません。
_TSCInterfaceName()	ありません。
_TSCAcceptorName()	ありません。
_TSCTimeout()	ありません。
_TSCTimeout(int)	ありません。
_TSCPriority()	ありません。
_TSCPriority(int)	ありません。
_TSCContext	ありません。
クライアント側からのオブジェクト呼び出し	生成時に指定した TSCClient 型のインスタンス

セション呼び出し機能使用中のメソッド呼び出し規則

セション呼び出し機能使用中（_TSCStart() メソッド呼び出し完了から _TSCStop() メソッド呼び出し完了まで）の，TSCSessionProxy クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	セション呼び出し中での呼び出し
_TSCStart()	×
_TSCStop()	○
_TSCSessionInterval()	○
_TSCSessionInterval(int)	○
_TSCInterfaceName()	○
_TSCAcceptorName()	○
_TSCTimeout()	○
_TSCTimeout(int)	○
_TSCPriority()	○
_TSCPriority(int)	○
_TSCContext	○
クライアント側からのオブジェクト呼び出し	○

（凡例）

○：できます。

×：できません。

TSCSystemException（Java）

TSCSystemException はシステム提供例外クラスです。

TSCSystemException は，OTM での例外用の基本クラスです。次に TSCSystemException の特徴を示します。

- 属性としてエラーコード，内容コード，場所コード，完了状態，および四つの保守コードを持ちます。
- ユーザはこのクラスのインスタンスを生成できません。
- ユーザはこのクラスの型で，派生クラスの例外を catch できます。

エラーコード

エラーコードは TSCSystemException クラスの派生クラスに対応するコードです。TSCSystemException クラスの型で例外を catch した場合，エラーコードを参照することで例外の種類を知ることができます。

定数値の順に並べたエラーコードの一覧を次の表に示します。エラーコードに対応する各派生クラスについては，この章の「TSCSystemException の派生クラス（Java）」を参照してください。

表 5-5　エラーコードの一覧（Java）

エラーコード	例外クラス
BAD_PARAM	TSCBadParamException
NO_MEMORY	TSCNoMemoryException
COMM_FAILURE	TSCCommFailureException
NO_PERMISSION	TSCNoPermissionException
INTERNAL	TSCInternalException
MARSHAL	TSCMarshalException
INITIALIZE	TSCInitializeException
NO_IMPLEMENT	TSCNoImplementException
BAD_OPERATION	TSCBadOperationException
NO_RESOURCES	TSCNoResourcesException
NO_RESPONSE	TSCNoResponseException
BAD_INV_ORDER	TSCBadInvOrderException
TRANSIENT	TSCTransientException
OBJECT_NOT_EXIST	TSCObjectNotExistException
UNKNOWN	TSCUnknownException
INV_OBJREF	TSCInvObjrefException

エラーコード	例外クラス
IMP_LIMIT	TSCImpLimitException
BAD_TYPECODE	TSCBadTypecodeException
PERSIST_STORE	TSCPersistStoreException
FREE_MEM	TSCFreeMemException
INV_IDENT	TSCInvIdentException
INV_FLAG	TSCInvFlagException
INTF_REPOS	TSCIntfReposException
BAD_CONTEXT	TSCBadContextException
OBJ_ADAPTER	TSCObjAdapterException
DATA_CONVERSION	TSCDataConversionException

個々のエラーコードの詳細については，「付録 A　エラーコード一覧」を参照してください。

内容コード

内容コードは，障害の詳しい情報を示すコードです。エラーコード間で内容コードが重複することはないため，ユニークな定数を持ちます。したがって，TSCSystemException クラスの型で例外を catch する場合，内容コードを参照することで詳細な情報を知ることができます。ユーザが TSCSystemException の派生クラスのインスタンスを生成して throw する場合，ユーザ用に割り当てられている値を内容コードに設定してください。

内容コードの分類を次の表に示します。

表 5-6　内容コードの分類（Java）

分類	内容コードの範囲
ユーザアプリケーション用	0 ～ 999
OTM システム予約	1000 ～ 30000

個々の内容コードの詳細については，「付録 D　内容コード一覧」を参照してください。

場所コード

場所コードは障害が発生した場所を示します。ユーザが TSCSystemException の派生クラスのインスタンスを生成して throw する場合，PLACE_CODE_USER_AP を設定してください。

場所コードの一覧を次の表に示します。

表 5-7　場所コードの一覧（Java）

場所コード	場所
PLACE_CODE_USER_AP	ユーザアプリケーション
PLACE_CODE_SERV	OTM のサーバ機能部分
PLACE_CODE_DAEMON	TSC デーモン
PLACE_CODE_CLNT	OTM のクライアント機能部分
PLACE_CODE_CLNT_REG	TSC レギュレータ
PLACE_CODE_STUB	TSC ユーザプロキシ（スタブ）
PLACE_CODE_SKELTON	TSC ユーザスケルトン（スケルトン）
PLACE_CODE_ORBGW	TSCORB コネクタ

完了状態

完了状態は，障害が発生したときにメソッド呼び出しが完了しているかどうかを示します。

完了状態の一覧を次の表に示します。

表 5-8　完了状態の一覧（Java）

完了状態	説明
COMPLETED_NO	メソッド呼び出しが完了していません。
COMPLETED_MAYBE	メソッド呼び出しの完了状態を決定できません。
COMPLETED_YES	メソッド呼び出しが完了しています。

形式

```
public class TSCSystemException extends java.lang.RuntimeException
{
  //各種情報取得
  public int getErrorCode();
  public int getDetailCode();
  public int getPlaceCode();
  public int getCompletionStatus();
  public int getMaintenanceCode1();
  public int getMaintenanceCode2();
  public int getMaintenanceCode3();
  public int getMaintenanceCode4();

  //エラーコード
  public static final int BAD_PARAM = 1;
  public static final int NO_MEMORY = 2;
  public static final int COMM_FAILURE = 3;
  public static final int NO_PERMISSION = 4;
  public static final int INTERNAL = 5;
  public static final int MARSHAL = 6;
```

```
public static final int INITIALIZE = 7;
public static final int NO_IMPLEMENT = 8;
public static final int BAD_OPERATION = 9;
public static final int NO_RESOURCES = 10;
public static final int NO_RESPONSE = 11;
public static final int BAD_INV_ORDER = 12;
public static final int TRANSIENT = 13;
public static final int OBJECT_NOT_EXIST = 14;
public static final int UNKNOWN = 15;
public static final int INV_OBJREF = 16;
public static final int IMP_LIMIT = 17;
public static final int BAD_TYPECODE = 18;
public static final int PERSIST_STORE = 19;
public static final int FREE_MEM = 20;
public static final int INV_IDENT = 21;
public static final int INV_FLAG = 22;
public static final int INTF_REPOS = 23;
public static final int BAD_CONTEXT = 24;
public static final int OBJ_ADAPTER = 25;
public static final int DATA_CONVERSION = 26;

//
//詳細コード
//
//BAD_PARAM
public static final int INVALID_TIMEOUT = 1001;
public static final int INVALID_RT_ACPT_NAME = 1002;
public static final int INVALID_PARALLEL_COUNT = 1003;
public static final int INVALID_ACPT_NAME = 1004;
public static final int OBJ_FACT_IS_NULL = 1005;
public static final int ACPT_IS_NULL = 1006;
public static final int INVALID_ACPT_REGID = 1007;
public static final int INVALID_TSCID = 1008;
public static final int INVALID_DOMAIN_NAME = 1009;
public static final int INVALID_OP_PARAM = 1010;
public static final int SERV_IS_NULL = 1011;
public static final int CLNT_IS_NULL = 1012;
public static final int ORB_IS_NULL = 1013;
public static final int DOMAIN_IS_NULL = 1015;
public static final int INVALID_REQUEST_WAY = 1016;
public static final int INVALID_PRIORITY = 1017;
public static final int THREAD_FACT_IS_NULL = 1018;
public static final int PROXY_IS_NULL = 1020;
public static final int OBJECT_IS_NULL = 1021;
public static final int INVALID_FLAG = 1022;
public static final int INVALID_WATCH_TIME = 1027;
public static final int WATCH_TIME_IS_NULL = 1028;
public static final int INVALID_RETRY_REQUIREMENT = 1029;
public static final int INVALID_SESSION_INTERVAL = 1038;
public static final int INVALID_QUEUE_LENGTH = 1047;

//NO_MEMORY
public static final int MEM_ALLOC_FAILURE = 2001;

//COMM_FAILURE
public static final int SEND_CLNT_FAILURE = 3004;
public static final int SEND_THIN_CLNT_FAILURE = 3005;
public static final int SEND_SERV_FAILURE = 3006;
```

```
public static final int SEND_TSCD_FAILURE = 3007;
public static final int BASIC_CONN_FAILURE = 3009;
public static final int CONN_FAILURE = 3010;
public static final int INCOMPATIBLE_PROTOCOL = 3011;
public static final int NOT_IGNORE_PROTOCOL = 3012;
public static final int DEACTIVATE_FAILURE = 3021;

//NO_PERMISSION
public static final int CALL_IN_HOLD = 4001;
public static final int RT_ACPT_IS_ACTIVE = 4002;
public static final int SERV_CONN_IN_END = 4005;
public static final int ACTIVATE_IN_END = 4006;
public static final int CLNT_CONN_IN_END = 4007;
public static final int DIFF_THREAD_CALL = 4008;
public static final int CALL_IN_END = 4009;
public static final int ACPT_NOT_REGISTED = 4010;
public static final int SERV_CONN_IN_START = 4011;
public static final int CLNT_CONN_IN_START = 4012;
public static final int TSCD_IS_NOT_MY_HOST = 4013;
public static final int OVER_ACPT_REGI = 4014;
public static final int NOT_SUPPORTED = 4015;
public static final int ACTIVATE_IN_START = 4016;
public static final int DEACTIVATE_IN_END =4018;
public static final int CLNT_DISCONN_IN_END = 4020;
public static final int ACTIVATE_WITH_DIFF_PROP = 4022;
public static final int CLNT_INIT_IN_END = 4023;
public static final int SERV_INIT_IN_END = 4024;
public static final int CLNT_INIT_IN_START = 4025;
public static final int SERV_INIT_IN_START = 4026;
public static final int NOT_ACCEPT_OBJECT = 4027;
public static final int CLNT_COMMAND_START = 4028;
public static final int WATCH_IS_STARTED = 4029;
public static final int WATCH_IS_STOPPED = 4030;
public static final int SAME_APID_EXIST = 4031;
public static final int FILE_ACCESS_FAILURE = 4032;
public static final int SESSION_IN_END = 4033;
public static final int SESSION_IN_CALL = 4034;

//INTERNAL
public static final int PROPERTIES_FAILURE = 5001;
public static final int MSG_TYPE_FAILURE = 5002;
public static final int MUTEX_FAILURE = 5003;
public static final int SIG_COND_FAILURE = 5004;
public static final int EVENT_FAILURE = 5005;
public static final int SH_MEM_FAILURE = 5006;
public static final int THREAD_CREATE_FAILURE = 5007;
public static final int TSD_FAILURE = 5008;
public static final int CBL_ADAPTER_ERROR = 5009;
public static final int SYSTEM_TIME_FAILURE = 5010;
public static final int PROGRAM_ERROR = 5999;

//MARSHAL
public static final int INVALID_STREAM_LEN = 6001;
public static final int INVALID_STREAM_VALUE = 6002;
public static final int MARSHAL_OTHERS = 6003;
public static final int REQ_MARSHAL_FAILURE = 6004;
public static final int REQ_UNMARSHAL_FAILURE = 6005;
public static final int REP_MARSHAL_FAILURE = 6006;
```

```
public static final int REP_UNMARSHAL_FAILURE = 6007;
public static final int UEXCEPT_MARSHAL_FAILURE = 6008;
public static final int UEXCEPT_UNMARSHAL_FAILURE = 6009;
public static final int MARSHAL_ERROR = 6010;

//INITIALIZE
public static final int INVALID_DEF_TIMEOUT = 7002;
public static final int INVALID_DEF_RT_ACPT = 7003;
public static final int INVALID_DEF_PARALLEL_COUNT = 7004;
public static final int LOAD_SHLIB_FAILURE = 7005;
public static final int INVALID_DEF_TSCID = 7006;
public static final int INVALID_DEF_DOMAIN_NAME = 7007;
public static final int INVALID_DEF_WITH_SYSTEM = 7008;
public static final int INVALID_ENV_TSCDIR = 7009;
public static final int MTRACE_FAILURE = 7010;
public static final int INVALID_DEF_NICE = 7011;
public static final int INVALID_DEF_PRIORITY = 7012;
public static final int INVALID_DEF_ACPT = 7013;
public static final int INVALID_PRC_KIND = 7014;
public static final int INVALID_DEF_REQUEST_WAY = 7015;
public static final int
        INVALID_DEF_CLIENT_MESSAGE_BUFFER_COUNT = 7016;
public static final int INVALID_DEF_APID = 7017;
public static final int INVALID_DEF_WATCH_TIME = 7018;
public static final int INVALID_DEF_WATCH_METHOD = 7019;
public static final int INVALID_DEF_MY_HOST = 7020;
public static final int INVALID_DEF_REBIND_TIMES = 7021;
public static final int INVALID_DEF_REBIND_INTERVAL = 7022;
public static final int INVALID_DEF_RETRY_REFERENCE = 7024;
public static final int INVALID_FILE_FORMAT = 7025;
public static final int INVALID_DEF_RETRY_WAY = 7030;
public static final int ALREADY_SHUTDOWN = 7031;
public static final int INVALID_DEF_EXCEPT_CONVERT_FILE = 7034;
public static final int ENTRY_FAILURE = 7035;
public static final int INVALID_DEF_SESSION_INTERVAL = 7036;
public static final int INVALID_DEF_QUEUE_LENGTH = 7040;

//NO_IMPLEMENT
public static final int NO_SUCH_INTERF = 8001;
public static final int NO_SUCH_ACPT = 8002;
public static final int NO_SUCH_OP_NAME = 9001;

//NO_RESOURCES
public static final int OVER_MAX_CLNT = 10001;
public static final int OVER_MAX_SERV = 10002;
public static final int OVER_ADM_MAX_CLNT = 10005;
public static final int OVER_ADM_MAX_SERV = 10006;
public static final int OVER_MAX_RT_ACPT_REGI = 10007;
public static final int OVER_MAX_THIN_CLIENT = 10008;
public static final int OVER_MAX_DISPATCH_PARALLEL = 10009;
public static final int OVER_MAX_REQUEST_COUNT = 10010;
public static final int OVER_MAX_ORB_CLIENT = 10011;

//NO_RESPONSE
public static final int TIMED_OUT = 11001;

//BAD_INV_ORDER
public static final int ALREADY_ACTIVE = 12002;
```

```
public static final int ALREADY_DEACTIVE = 12003;
public static final int CLNT_NOT_INITIALIZED = 12004;
public static final int SERV_NOT_INITIALIZED = 12005;
public static final int ALREADY_INITCLNT = 12006;
public static final int ALREADY_INITSERV = 12007;
public static final int ALREADY_SERV_ML = 12008;
public static final int ALREADY_SESSION_START = 12013;
public static final int SESSION_NOT_START = 12014;

//TRANSIENT
public static final int REBIND_FAILURE = 13001;
public static final int ALL_CONN_FAILURE = 13002;

//OBJECT_NOT_EXIST
public static final int SERV_NO_SUCH_INTERF = 14001;
public static final int SERV_NO_SUCH_ACPT = 14002;

//UNKNOWN
public static final int COMMON_EXCEPTION = 15001;
public static final int INVALID_USER_EXCEPTION = 15002;

//INV_OBJREF
public static final int FACTORY_CREATE_FAILURE = 16001;
public static final int THREAD_FACTORY_CREATE_FAILURE = 16002;
public static final int FACTORY_DESTROY_FAILURE = 16003;
public static final int THREAD_FACTORY_DESTROY_FAILURE = 16004;
public static final int CREATED_OBJECT_IS_NULL = 16005;
public static final int CREATED_THREAD_OBJECT_IS_NULL = 16006;

public static final int TPBROKER_BAD_PARAM = 1998;
public static final int TPBROKER_NO_MEMORY = 2998;
public static final int TPBROKER_COMM_FAILURE = 3998;
public static final int TPBROKER_NO_PERMISSION = 4998;
public static final int TPBROKER_INTERNAL = 5998;
public static final int TPBROKER_MARSHAL = 6998;
public static final int TPBROKER_INITIALIZE = 7998;
public static final int TPBROKER_NO_IMPLEMENT = 8998;
public static final int TPBROKER_BAD_OPERATION = 9998;
public static final int TPBROKER_NO_RESOURCES = 10998;
public static final int TPBROKER_NO_RESPONSE = 11998;
public static final int TPBROKER_BAD_INV_ORDER = 12998;
public static final int TPBROKER_TRANSIENT = 13998;
public static final int TPBROKER_OBJECT_NOT_EXIST = 14998;
public static final int TPBROKER_UNKNOWN = 15998;
public static final int TPBROKER_INV_OBJREF = 16998;
public static final int TPBROKER_IMP_LIMIT = 17998;
public static final int TPBROKER_BAD_TYPECODE = 18998;
public static final int TPBROKER_PERSIST_STORE = 19998;
public static final int TPBROKER_FREE_MEM = 20998;
public static final int TPBROKER_INV_IDENT = 21998;
public static final int TPBROKER_INV_FLAG = 22998;
public static final int TPBROKER_INTF_REPOS = 23998;
public static final int  TPBROKER_BAD_CONTEXT = 24998;
public static final int TPBROKER_OBJ_ADAPTER = 25998;
public static final int TPBROKER_DATA_CONVERSION = 26998;

//場所コード
public static final int PLACE_CODE_USER_AP = 1;
public static final int PLACE_CODE_SERV = 2;
public static final int PLACE_CODE_DAEMON = 3;
public static final int PLACE_CODE_CLNT = 4;
```

```
  public static final int PLACE_CODE_CLNT_REG = 5;
  public static final int PLACE_CODE_STUB = 6;
  public static final int PLACE_CODE_SKELTON = 7;
  public static final int PLACE_CODE_ORBGW = 8;

// 完了状態
  public static final int COMPLETED_NO = -1;
  public static final int COMPLETED_MAYBE = 0;
  public static final int COMPLETED_YES = 1;
};
```

メソッド

● public int getErrorCode()

項目	型・意味
戻り値	エラーコード

障害のエラーコードを返します。

● public int getDetailCode()

項目	型・意味
戻り値	内容コード

障害の内容コードを返します。

● public int getPlaceCode()

項目	型・意味
戻り値	場所コード

障害の場所コードを返します。

● public int getCompletionStatus()

項目	型・意味
戻り値	完了状態

障害発生時のメソッド呼び出しの完了状態を返します。

● public int getMaintenanceCode1()

項目	型・意味
戻り値	保守コード 1

障害の保守コード 1 を返します。

● public int getMaintenanceCode2()

項目	型・意味
戻り値	保守コード 2

障害の保守コード 2 を返します。

● public int getMaintenanceCode3()

項目	型・意味
戻り値	保守コード 3

障害の保守コード 3 を返します。

● public int getMaintenanceCode4()

項目	型・意味
戻り値	保守コード 4

障害の保守コード 4 を返します。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCSystemException クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
getErrorCode	できます。
getDetailCode	できます。
getPlaceCode	できます。
getCompletionStatus	できます。
getMaintenanceCode1	できます。
getMaintenanceCode2	できます。
getMaintenanceCode3	できます。
getMaintenanceCode4	できます。

TSCSystemException の派生クラス（Java）

TSCSystemException の派生クラスはシステム提供例外クラスです。

次に TSCSystemException の特徴を示します。

- 各派生クラスはエラーコードと 1 対 1 に対応します。
- 各例外クラス間でも，内容コードの一意性は保証されます。
- TSC システム例外を生成する場合，TSCSystemException の派生クラスのインスタンスを生成します。
- 種類別に例外を catch することで，種類ごとに違った例外処理ができます。

各種例外クラス

各種例外クラスの一覧を，アルファベット順で次の表に示します。

表 5-9　OTM のシステム例外（Java）

例外名	説明
TSCBadContextException	コンテキストオブジェクトの処理中に障害が発生しました。
TSCBadInvOrderException	ルーチン呼び出しの順番が不正です。
TSCBadOperationException	オペレーションが無効です。
TSCBadParamException	無効パラメタが渡されました。
TSCBadTypecodeException	タイプコードが不正です。
TSCCommFailureException	通信障害が発生しました。
TSCDataConversionException	データ変換に失敗しました。
TSCFreeMemException	メモリの解放に失敗しました。
TSCImpLimitException	実装の制限を超えました。
TSCInitializeException	ORB 初期化障害が発生しました。
TSCInternalException	ORB 内部エラーが発生しました。
TSCIntfReposException	インタフェースリポジトリへのアクセス中に障害が発生しました。
TSCInvFlagException	不正なフラグが指定されました。
TSCInvIdentException	識別子の構文が不正です。
TSCInvObjrefException	無効なオブジェクトリファレンスが指定されました。
TSCMarshalException	スタブ，スケルトンで CDR マーシャルに失敗しました。
TSCNoImplementException	オペレーションの実装が使用できません。
TSCNoMemoryException	動的メモリの割り当て障害が発生しました。
TSCNoPermissionException	許可されていないオペレーションを実行しようとしました。
TSCNoResourcesException	リクエストを処理するための資源が不足しています。

例外名	説明
TSCNoResponseException	リクエストに対する応答がありません。
TSCObjAdapterException	オブジェクトアダプタが障害を検出しました。
TSCObjectNotExistException	該当するオブジェクトがありません。
TSCPersistStoreException	パーシステントストレージに障害が発生しました。
TSCTransientException	一時的な障害が発生しました。
TSCUnknownException	未知の例外が発生しました。

内容コード

ユーザが TSCSystemException の派生クラスのインスタンスを生成して throw する場合，ユーザアプリケーション用に割り当てられている値を内容コードに設定してください。

OTM の内容コードの分類を次の表に示します。

表 5-10　OTM の内容コード（Java）

分類	内容コードの範囲
ユーザアプリケーション用	0 ～ 999
OTM システム予約	1000 ～ 30000

個々の内容コードの詳細については，「付録 D　内容コード一覧」を参照してください。

場所コード

場所コードは障害が発生した場所を示します。ユーザが TSCSystemException の派生クラスのインスタンスを生成して throw する場合，PLACE_CODE_USER_AP を設定してください。

OTM の場所コードの一覧を次の表に示します。

表 5-11　OTM の場所コード（Java）

場所コード	場所
PLACE_CODE_USER_AP	ユーザアプリケーション
PLACE_CODE_SERV	OTM のサーバ機能部分
PLACE_CODE_DAEMON	TSC デーモン
PLACE_CODE_CLNT	OTM のクライアント機能部分
PLACE_CODE_CLNT_REG	TSC レギュレータ
PLACE_CODE_STUB	スタブ

場所コード	場所
PLACE_CODE_SKELTON	スケルトン
PLACE_CODE_ORBGW	TSCORB コネクタ

完了状態

完了状態は，障害が発生したときにメソッド呼び出しが完了しているかどうかを示します。

OTM の完了状態の一覧を次の表に示します。

表 5-12　OTM の完了状態（Java）

完了状態	説明
COMPLETED_NO	メソッド呼び出しが完了していません。
COMPLETED_MAYBE	メソッド呼び出しの完了状態を決定できません。
COMPLETED_YES	メソッド呼び出しが完了しています。

形式

```
public class TSCBadContextException extends TSCSystemException
{
  public TSCBadContextException(int detail_code,
                                int place_code,
                                int completion_status,
                                int maintenance_code1,
                                int maintenance_code2,
                                int maintenance_code3,
                                int maintenance_code4);
};

public class TSCBadInvOrderException extends TSCSystemException
{
  public TSCBadInvOrderException(int detail_code,
                                 int place_code,
                                 int completion_status,
                                 int maintenance_code1,
                                 int maintenance_code2,
                                 int maintenance_code3,
                                 int maintenance_code4);
};

public class TSCBadOperationException extends TSCSystemException
{
  public TSCBadOperationException(int detail_code,
                                  int place_code,
                                  int completion_status,
                                  int maintenance_code1,
                                  int maintenance_code2,
                                  int maintenance_code3,
                                  int maintenance_code4);
```

```
};

public class TSCBadParamException extends TSCSystemException
{
  public TSCBadParamException(int detail_code,
                              int place_code,
                              int completion_status,
                              int maintenance_code1,
                              int maintenance_code2,
                              int maintenance_code3,
                              int maintenance_code4);
};

public class TSCBadTypecodeException extends TSCSystemException
{
  public TSCBadTypecodeException(int detail_code,
                                 int place_code,
                                 int completion_status,
                                 int maintenance_code1,
                                 int maintenance_code2,
                                 int maintenance_code3,
                                 int maintenance_code4);
};

public class TSCCommFailureException extends TSCSystemException
{
  public TSCCommFailureException(int detail_code,
                                 int place_code,
                                 int completion_status,
                                 int maintenance_code1,
                                 int maintenance_code2,
                                 int maintenance_code3,
                                 int maintenance_code4);
};

public class TSCDataConversionException extends TSCSystemException
{
  public TSCDataConversionException(int detail_code,
                                    int place_code,
                                    int completion_status,
                                    int maintenance_code1,
                                    int maintenance_code2,
                                    int maintenance_code3,
                                    int maintenance_code4);
};

public class TSCFreeMemException extends TSCSystemException
{
  public TSCFreeMemException(int detail_code,
                             int place_code,
                             int completion_status,
                             int maintenance_code1,
                             int maintenance_code2,
                             int maintenance_code3,
                             int maintenance_code4);
};

public class TSCImpLimitException extends TSCSystemException
{
```

```
  public TSCImpLimitException(int detail_code,
                              int place_code,
                              int completion_status,
                              int maintenance_code1,
                              int maintenance_code2,
                              int maintenance_code3,
                              int maintenance_code4);
};

public class TSCInitializeException extends TSCSystemException
{
  public TSCInitializeException(int detail_code,
                                int place_code,
                                int completion_status,
                                int maintenance_code1,
                                int maintenance_code2,
                                int maintenance_code3,
                                int maintenance_code4);
};

public class TSCInternalException extends TSCSystemException
{
  public TSCInternalException(int detail_code,
                              int place_code,
                              int completion_status,
                              int maintenance_code1,
                              int maintenance_code2,
                              int maintenance_code3,
                              int maintenance_code4);
};

public class TSCIntfReposException extends TSCSystemException
{
  public TSCIntfReposException(int detail_code,
                               int place_code,
                               int completion_status,
                               int maintenance_code1,
                               int maintenance_code2,
                               int maintenance_code3,
                               int maintenance_code4);
};

public class TSCInvFlagException extends TSCSystemException
{
  public TSCInvFlagException(int detail_code,
                             int place_code,
                             int completion_status,
                             int maintenance_code1,
                             int maintenance_code2,
                             int maintenance_code3,
                             int maintenance_code4);
};

public class TSCInvIdentException extends TSCSystemException
{
  public TSCInvIdentException(int detail_code,
                              int place_code,
                              int completion_status,
                              int maintenance_code1,
```

```
                                int maintenance_code2,
                                int maintenance_code3,
                                int maintenance_code4);
};

public class TSCInvObjrefException extends TSCSystemException
{
  public TSCInvObjrefException(int detail_code,
                               int place_code,
                               int completion_status,
                               int maintenance_code1,
                               int maintenance_code2,
                               int maintenance_code3,
                               int maintenance_code4);
};

public class TSCMarshalException extends TSCSystemException
{
  public TSCMarshalException(int detail_code,
                             int place_code,
                             int completion_status,
                             int maintenance_code1,
                             int maintenance_code2,
                             int maintenance_code3,
                             int maintenance_code4);
};

public class TSCNoImplementException extends TSCSystemException
{
  public TSCNoImplementException(int detail_code,
                                 int place_code,
                                 int completion_status,
                                 int maintenance_code1,
                                 int maintenance_code2,
                                 int maintenance_code3,
                                 int maintenance_code4);
};

public class TSCNoMemoryException extends TSCSystemException
{
  public TSCNoMemoryException(int detail_code,
                              int place_code,
                              int completion_status,
                              int maintenance_code1,
                              int maintenance_code2,
                              int maintenance_code3,
                              int maintenance_code4);
};

public class TSCNoPermissionException extends TSCSystemException
{
  public TSCNoPermissionException(int detail_code,
                                  int place_code,
                                  int completion_status,
                                  int maintenance_code1,
                                  int maintenance_code2,
                                  int maintenance_code3,
                                  int maintenance_code4);
```

```
};

public class TSCNoResourcesException extends TSCSystemException
{
  public TSCNoResourcesException(int detail_code,
                                 int place_code,
                                 int completion_status,
                                 int maintenance_code1,
                                 int maintenance_code2,
                                 int maintenance_code3,
                                 int maintenance_code4);
};

public class TSCNoResponseException extends TSCSystemException
{
  public TSCNoResponseException(int detail_code,
                                int place_code,
                                int completion_status,
                                int maintenance_code1,
                                int maintenance_code2,
                                int maintenance_code3,
                                int maintenance_code4);
};

public class TSCObjAdapterException extends TSCSystemException
{
  public TSCObjAdapterException(int detail_code,
                                int place_code,
                                int completion_status,
                                int maintenance_code1,
                                int maintenance_code2,
                                int maintenance_code3,
                                int maintenance_code4);
};

public class TSCObjectNotExistException extends TSCSystemException
{
  public TSCObjectNotExistException(int detail_code,
                                    int place_code,
                                    int completion_status,
                                    int maintenance_code1,
                                    int maintenance_code2,
                                    int maintenance_code3,
                                    int maintenance_code4);
};

public class TSCPersistStoreException extends TSCSystemException
{
  public TSCPersistStoreException(int detail_code,
                                  int place_code,
                                  int completion_status,
                                  int maintenance_code1,
                                  int maintenance_code2,
                                  int maintenance_code3,
                                  int maintenance_code4);
};

public class TSCTransientException extends TSCSystemException
```

```
{
  public TSCTransientException(int detail_code,
                               int place_code,
                               int completion_status,
                               int maintenance_code1,
                               int maintenance_code2,
                               int maintenance_code3,
                               int maintenance_code4);
};

public class TSCUnknownException extends TSCSystemException
{
  public TSCUnknownException(int detail_code,
                             int place_code,
                             int completion_status,
                             int maintenance_code1,
                             int maintenance_code2,
                             int maintenance_code3,
                             int maintenance_code4);
};
```

TSCThread（Java）

TSCThread はシステム提供インタフェースです。

TSCThread は，OTM が管理するスレッドを表すオブジェクトのインタフェースです。

ユーザは，TSCThread を継承させ，必要な情報を保存するクラスをスレッド単位に定義します。スレッドに割り付けられるオブジェクトとして，派生クラスのインスタンスを生成します。次に TSCThread の特徴を示します。

- ユーザは TSCThread クラスを継承して，実装クラスを記述する必要があります。
- TSCThread を生成する TSCThreadFactory の実装クラスを記述する必要があります。

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;

public interface TSCThread
{
};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCThread;
```

TSCThreadFactory（Java）

TSCThreadFactory はシステム提供インタフェースです。

TSCThreadFactory は，TSCThread を生成または削除するメソッドを持つオブジェクトのインタフェースです。

ユーザは TSCObjectFactory を継承させて，TSC ユーザスレッドを生成および削除するクラスを定義し，TSC ユーザスレッドのファクトリとして派生クラスのインスタンスを生成します。次に TSCObjectFactory の特徴を示します。

- 直接，TSCThreadFactory クラスのインスタンスを生成できません。
- ユーザは TSCThreadFactory クラスを継承して，実装クラスを記述する必要があります。

OTM からの呼び出しによる TSCThread の管理

OTM は，TSCThreadFactory の create を呼び出すことで，返される TSCThread をそのスレッドに割り当ててスレッドを生成します。逆にそのスレッドに割り当てた TSCThread を引数に destroy を呼び出すことで，生成したスレッドを削除します。

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;

public interface TSCThreadFactory
{
  public TSCThread create();
  public void destroy(TSCThread tsc_thr);

};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCThreadFactory;
```

コールバックメソッド

● public TSCThread create()

項目	型・意味
戻り値	管理される TSCThread
例外	各種 TSCSystemException

TSCThread を返します。TSC ユーザスレッドを生成するコードを記述できます。OTM がこのメソッドを呼び出した結果，返された TSCThread が管理対象となります。

なお，OTM は，複数のスレッド上から同時にこのメソッドを呼び出すので，ユーザはマ

ルチスレッド環境に対応するリエントラントなコードを記述する必要があります。

また，create 呼び出しに失敗した場合は，各種の TSCSystemException によって通知する形でコードを記述してください。

● public void destroy(TSCThread tsc_thr)

項目	型・意味	
引数	TSCThread tsc_thr	管理対象から外す TSCThread
例外	ありません。	

TSCThread を消去する前の処理のコードを記述できます。OTM が TSC ユーザスレッドを管理対象から外すとき，該当する TSCThread を引数に指定して，このメソッドを呼び出します。

OTM は，複数のスレッド上から同時にこのメソッドを呼び出すので，ユーザはマルチスレッド環境に対応するリエントラントなコードを記述する必要があります。

また，このメソッドから通知した例外は無視されます。

TSCThreadFactory の派生クラスの生成

TSCThreadFactory の派生クラスは，new オペレータで生成します。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCThreadFactory クラス型のインスタンスのメソッドを呼び出す規則を次に示します。なお，これらのメソッドは，OTM が呼び出します。

メソッド	複数のスレッド上からの同時呼び出し
create	できます。
destroy	できます。

TSCWatchTime（Java）

TSCWatchTime はシステム提供クラスです。

TSCWatchTime は，start() が発行されてから stop() が発行されるまでの時間を監視するクラスです。時間監視が終了する前に指定された監視時間が経過するとエラーメッセージを出力して，プロセスを異常終了します。この機能は，サーバアプリケーションの initServer() 発行後から endServer() を発行するまでの間有効です。

形式

```
package JP.co.Hitachi.soft.TPBroker.TSC;

public class TSCWatchTime
{
  public TSCWatchTime();

  public TSCWatchTime(int watch_time);

  public void start();
  public void stop();
  public void reset();

};
```

インポートクラス

```
import JP.co.Hitachi.soft.TPBroker.TSC.TSCWatchTime;
```

コンストラクタ

● public TSCWatchTime()

項目	型・意味
例外	TSCBadInvOrderException TSCInternalException TSCNoMemoryException

TSCWatchTime を生成します。

サーバアプリケーションの開始時にコマンドオプション引数 -TSCWatchTime で指定した監視時間（秒）が適用されます。

● public TSCWatchTime(int watch_time)

項目	型・意味（単位）	
引数	int watch_time	監視時間（秒）
例外	TSCBadInvOrderException TSCBadParamException TSCInternalException TSCNoMemoryException	

TSCWatchTime を生成します。

引数に "0" を指定した場合，サーバアプリケーションの開始時にコマンドオプション引数 -TSCWatchTime で指定した監視時間（秒）が適用されます。

メソッド

● public void start()

項目	型・意味（単位）
例外	TSCInternalException TSCNoPermissionException

時間監視を開始します。または，stop() で中断した時間監視を再開します。

● public void stop()

項目	型・意味（単位）
例外	TSCInternalException TSCNoPermissionException

時間監視を中断します。start() を発行したスレッドと異なるスレッドでは，stop() を発行できません。

● public void reset()

項目	型・意味（単位）
例外	TSCInternalException TSCNoPermissionException

監視時間をこのクラスの生成時に指定した値に戻します。start() の発行以降 stop() の発行までの間は発行できません。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCWatchTime クラスのインスタンスのメソッドを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
start	できます。
stop	できます。※
reset	できます。

注※

必ず start() と同一のスレッドで発行してください。

注意事項

- IDL ファイルに記述したユーザメソッドの呼び出しからリターンまでの実行時間を監視する場合には，TSCWatchTime クラスではなく，サーバアプリケーションの開始時に指定するコマンドオプション引数 -TSCWatchMethod を使用してください。
- TSCAdm::initServer() の発行前，または TSCAdm::endServer() の発行後には，TSCWatchTime クラスのインスタンスを生成できません。
- stop() を発行する前にこのクラスのインスタンスへの参照を失うと，ガベージコレクションが実行されるまで，時間監視は終了しません。
- start() を発行したスレッドと異なるスレッドでは，stop() を呼び出せません。
- stop() を発行したあとに start() によって処理を再開する場合には，前回経過した時間を監視時間から差し引いて処理をします。
  次に例を示します。

```
TSCWatchTime wt = new TSCWatchTime(180);
    :
wt.start();
    :                       //(1) 60秒経過
wt.stop();
    :                       //(2)
wt.start();
```

例えば，180 秒の監視時間を設定してクラスを生成した場合に，(1) で示す範囲で 60 秒が経過すると，stop() の発行後，次の start() から stop() までの監視時間は 120 秒になります。180 秒の時間監視を設定したい場合には (2) で示す範囲で reset() を発行してください。その場合，再発行した start() から stop() までの監視時間は 180 秒になります。

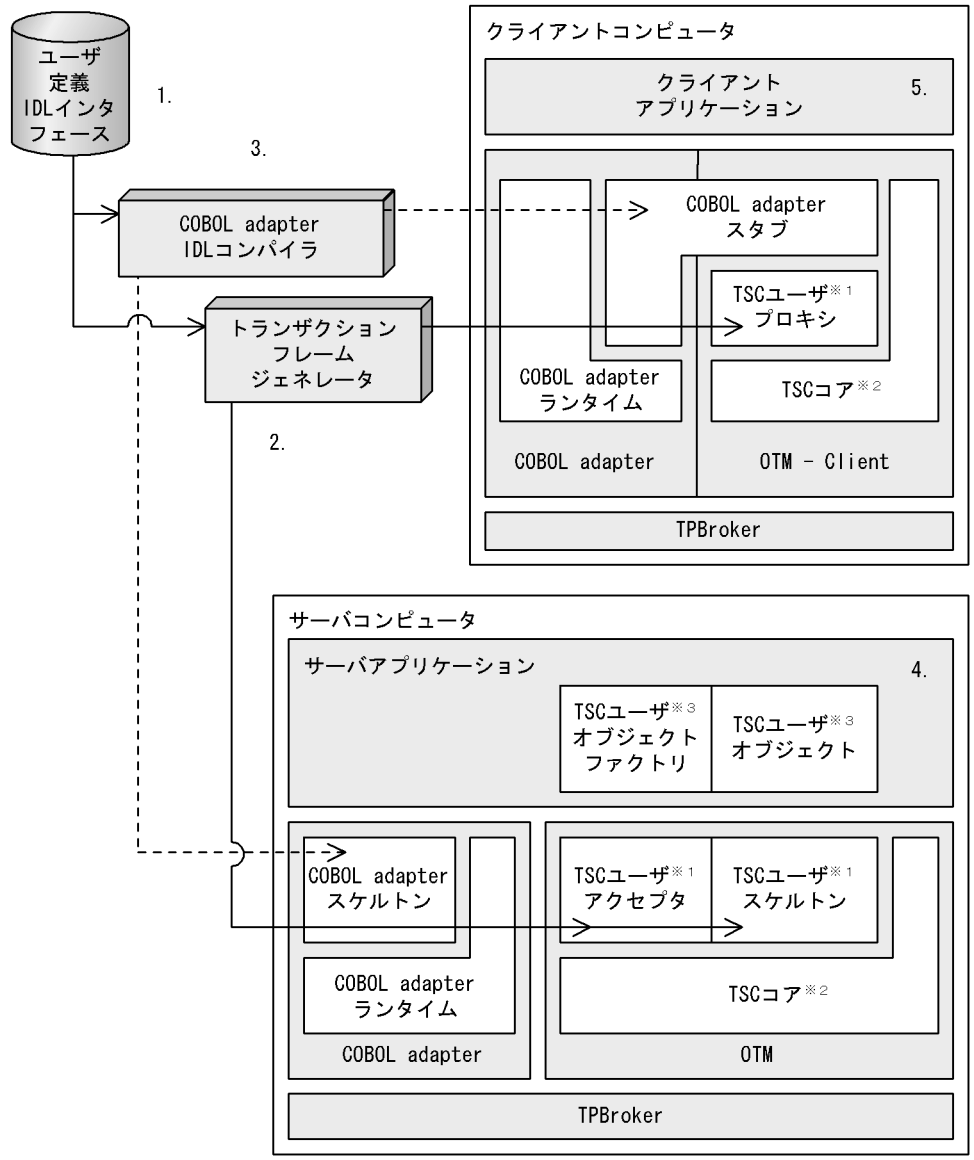
6 アプリケーションプログラムの作成（COBOL）

この章では，COBOL でアプリケーションプログラムを作成する方法について説明します。

6.1 アプリケーションプログラムの作成手順（COBOL）

COBOL でアプリケーションプログラムを作成する手順を次の図に示します。

図 6-1 アプリケーションプログラムを作成する手順（COBOL）



注

"COBOL adapter" は，"COBOL adapter for TPBroker" を表します。

注※ 1

トランザクションフレームジェネレータが生成する基底クラス，およびそのまま使用できるクラス（**ユーザ定義** IDL **インタフェース依存クラス**）です。

注※ 2

システム提供クラスです。OTM の機能の中心となるクラスです。

注※ 3

トランザクションフレームジェネレータが生成するクラスの雛形（**雛形クラス**）です。クラスのシグネチャ，およびコードの一部はすでに記述されています。自動生成されたままの状態では利用できないクラスなので，ユーザはコードを追加する必要があります。

図中の 1. ～ 5. の手順は次のとおりです。

1. **ユーザ定義** IDL **インタフェースを定義します。**
2. **トランザクションフレームジェネレータを使用してユーザ定義** IDL **インタフェース依存クラスおよび雛形クラスを生成します。**
3. COBOL adapter for TPBroker **を使用してソースを生成します。**
   COBOL adapter for TPBroker の IDL コンパイラについては，マニュアル「COBOL adapter for TPBroker　ユーザーズガイド」を参照してください。
4. **次に示すコードを記述してサーバアプリケーションを作成します。**
   - TSC ユーザオブジェクト（雛形クラスの編集）
   - TSC ユーザオブジェクトファクトリ（雛形クラスの編集）
   - 初期化処理
   - サービス登録処理
5. **次に示すコードを記述してクライアントアプリケーションを作成します。**
   - 初期化処理
   - サービス登録処理

次に，図中のオブジェクトについて説明します。

● COBOL adapter **スタブ**

COBOL adapter for TPBroker の IDL コンパイラが生成するスタブのオブジェクトです。

● TSC **ユーザプロキシ**

クライアントアプリケーションが TSC ユーザオブジェクトを呼び出すための代理オブジェクトです。ユーザ定義 IDL インタフェース依存のオブジェクトです。

● TSC **ユーザオブジェクトファクトリ**

サーバアプリケーションで，TSC ユーザオブジェクトを生成するオブジェクトです。実装についてはユーザが記述します。雛形のクラスです。

● TSC **ユーザオブジェクト**

サーバアプリケーションで，サービスを提供するオブジェクトです。このオブジェクトは，TSC ユーザスケルトンを継承し，実装についてはユーザが記述します。雛形のクラスです。

● COBOL adapter **スケルトン**

COBOL adapter for TPBroker の IDL コンパイラが生成するスケルトンのオブジェクトです。

● TSC **ユーザアクセプタ**

TSC ユーザプロキシから呼び出し要求を受けて，TSC ユーザオブジェクトにリクエストを配送するオブジェクトです。ユーザ定義 IDL インタフェース依存のオブジェクトです。

● TSC **ユーザスケルトン**

TSC ユーザオブジェクトのスケルトンです。ユーザ定義 IDL インタフェース依存のオブジェクトです。

6.2 同期型呼び出しをするアプリケーションプログラム（COBOL）

同期型呼び出しをするアプリケーションプログラムの COBOL での作成例を示します。これらのサンプルコードは製品で提供されています。

● ユーザ定義 IDL インタフェースの例

ユーザ定義 IDL インタフェースの例を次に示します。

```
//
// "ABCfile.idl"
//

typedef sequence<octet> OctetSeq;

interface CBLClass
{
  void call(in OctetSeq in_data, out OctetSeq out_data);
};
```

● IDL コンパイラが生成するクラス

COBOL adapter for TPBroker の IDL コンパイラはユーザ定義 IDL インタフェースから次のファイルを生成します。

- ABCfile_c.ocb
- ABCfile_s.ocb

● トランザクションフレームジェネレータが生成するクラス

OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。ただし，実際に生成されるのはクラスを使用するための副プログラムです。副プログラムの名称は次のとおりになります。

"クラス名"－"メソッド名"

表 6-1 同期型呼び出しをするアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（COBOL）

分類	ファイル名	副プログラムのプリフィックス（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• ABCfile_TSC_c.ocb	• CBLClass_TSCprxy（TSC ユーザプロキシ）

分類	ファイル名	副プログラムのプリフィックス（オブジェクト名）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• ABCfile_TSC_s.ocb	• CBLClass_TSCsk（TSC ユーザスケルトン） • CBLClass_TSCacpt（TSC ユーザアクセプタ）
雛形クラス	• ABCfile_TSC_t.ocb	• CBLClass_TSCimpl（TSC ユーザオブジェクト） • CBLClass_TSCfact（TSC ユーザオブジェクトファクトリ） • TSCCBLThreadFactory（TSC ユーザスレッドファクトリ） • TSCCBLThread（TSC ユーザスレッド）

6.2.1 同期型呼び出しをするクライアントアプリケーションの例（COBOL）

同期型呼び出しをするクライアントアプリケーションの処理の流れとコードの例を示します。

（1） サービス利用処理の流れ

1. COBOL adapter for TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
6. TSC **ユーザプロキシの削除**
7. TSC **デーモンへの接続解放**
8. TPBroker OTM **の終了処理**

（2） サービス利用処理のコード

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CLIENT.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.

DATA DIVISION.
WORKING-STORAGE SECTION.

01 ORB-PTR                USAGE POINTER.
01 DOMAIN-PTR             USAGE POINTER.
01 CLIENT-PTR             USAGE POINTER.
01 CLIENT-WAY             PIC S9(9) COMP.
01 MY-DOMAIN-NAME         PIC X(10).
01 MY-DOMAIN-NAME-PTR     USAGE POINTER.
01 MY-DOMAIN-NAME-LEN     PIC S9(9) COMP.
```

```
 01 MY-TSCID               PIC X(10).
 01 MY-TSCID-PTR           USAGE POINTER.
 01 MY-TSCID-LEN           PIC S9(9) COMP.
 01 MY-DOMAIN-FLAG         PIC S9(9) COMP VALUE 1.
 01 ACCEPTOR-NAME          PIC X(20).
 01 ACCEPTOR-NAME-PTR      USAGE POINTER VALUE NULL.
 01 ACCEPTOR-NAME-LEN      PIC S9(9) COMP.
 01 PROXY-PTR              USAGE POINTER.

 01 INIT-CLIENT-FLAG       PIC S9(9) COMP.
 01 DOMAIN-CREATE-FLAG     PIC S9(9) COMP.
 01 GET-CLIENT-FLAG        PIC S9(9) COMP.
 01 TSCPRXY-CREATE-FLAG    PIC S9(9) COMP.

 01 CORBA-ENVIRONMENT.
  02 MAJOR                 PIC 9(9) COMP.
   88 CORBA-NO-EXCEPTION      VALUE 0.
   88 CORBA-USER-EXCEPTION    VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION VALUE 2.
  02 EXCEP                 USAGE POINTER.
  02 FUNC-NAME             PIC X(256).

 01 ERR-CODE               PIC S9(9) COMP.

* Sequence
 01 TYPE-CODE-PTR          USAGE POINTER.
 01 ELEMENT-NUMBER         PIC 9(9) COMP.
 01 SETOCTETVAL            PIC X.
 01 in_data                USAGE POINTER.
 01 out_data               USAGE POINTER.

 LINKAGE SECTION.
 01 ARGC                   PIC 9(9) COMP.
 01 ARGV                   USAGE POINTER.

 PROCEDURE DIVISION USING
         BY VALUE ARGC
         BY VALUE ARGV.

 MOVE 0 TO RETURN-CODE.
 MOVE 0 TO INIT-CLIENT-FLAG.
 MOVE 0 TO DOMAIN-CREATE-FLAG.
 MOVE 0 TO GET-CLIENT-FLAG.
 MOVE 0 TO TSCPRXY-CREATE-FLAG.

* ORBの初期化処理
 CALL 'CORBA-STUB-INIT-ABCfile'.

* 1. COBOL adapter for TPBrokerの初期化処理
 CALL 'CORBA_orb_init' USING
         BY REFERENCE ARGC
         BY REFERENCE ARGV
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING ORB-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
```

```
                BY REFERENCE MAJOR
                BY REFERENCE CORBA-ENVIRONMENT
        CALL 'CORBA_FreeException' USING
                EXCEP OF CORBA-ENVIRONMENT
        MOVE 1 TO RETURN-CODE
        GO TO PROG-END
    END-IF.
    DISPLAY 'Success CORBA_orb_init'.

   * 2. TPBroker OTMの初期化処理
    CALL 'TSCAdm-initClient' USING
            BY REFERENCE ARGC
            BY REFERENCE ARGV
            BY VALUE     ORB-PTR
            BY REFERENCE CORBA-ENVIRONMENT.
   * 例外チェック
    IF NOT CORBA-NO-EXCEPTION THEN
        CALL 'OTM-EXCEPTION-HANDLER' USING
                BY REFERENCE MAJOR
                BY REFERENCE CORBA-ENVIRONMENT
        CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
        MOVE 1 TO RETURN-CODE
        GO TO PROG-END
    END-IF.
    DISPLAY 'Success TSCAdm-initClient'.
    MOVE 1 TO INIT-CLIENT-FLAG.

   * 3. TSCデーモンへの接続
   * (1) TSCDomainの生成
    SET MY-DOMAIN-NAME-PTR TO NULL.
    SET MY-TSCID-PTR TO NULL.
    MOVE 1 TO MY-DOMAIN-FLAG.
    CALL 'TSCDomain-NEW' USING
            BY VALUE     MY-DOMAIN-NAME-PTR
            BY VALUE     MY-TSCID-PTR
            BY VALUE     MY-DOMAIN-FLAG
            BY REFERENCE CORBA-ENVIRONMENT
        RETURNING DOMAIN-PTR.
   * 例外チェック
    IF NOT CORBA-NO-EXCEPTION THEN
        CALL 'OTM-EXCEPTION-HANDLER' USING
                BY REFERENCE MAJOR
                BY REFERENCE CORBA-ENVIRONMENT
        CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
        MOVE 1 TO RETURN-CODE
        GO TO PROG-END
    END-IF.
    DISPLAY 'Success TSCDomain-NEW:DomainName = '
                                        MY-DOMAIN-NAME.
    MOVE 1 TO DOMAIN-CREATE-FLAG.

   * (2) TSCClientの取得
    MOVE 1 TO CLIENT-WAY.
    CALL 'TSCAdm-getTSCClient' USING
            BY VALUE     DOMAIN-PTR
```

```
         BY VALUE     CLIENT-WAY
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING CLIENT-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success TSCAdm-getTSCClient'.
 MOVE 1 TO GET-CLIENT-FLAG.

* 4. TSCユーザプロキシの生成および各種設定
*    IDLインタフェース"CBLClass"用のTSCProxy生成
 SET ACCEPTOR-NAME-PTR TO NULL.
 CALL 'CBLClass_TSCprxy-NEW' USING
         BY VALUE     CLIENT-PTR
         BY VALUE     ACCEPTOR-NAME-PTR
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING PROXY-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success CBLClass_TSCprxy-NEW'.
 MOVE 1 TO TSCPRXY-CREATE-FLAG.

* 5. TSCユーザプロキシのメソッド呼び出し
*    （サーバ側のオブジェクトの呼び出し）
*    in属性ユーザデータの確保と設定
*    Octet TypeCodeオブジェクトの生成
 CALL 'Create_CORBA_TypeCode' USING
         BY VALUE     10
         BY VALUE     1
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING TYPE-CODE-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

* Octet型Sequenceオブジェクトの生成
 MOVE 999999999 TO ELEMENT-NUMBER.
```

```
 CALL 'CORBA-SeqAlloc' USING
         BY REFERENCE ELEMENT-NUMBER
         BY REFERENCE TYPE-CODE-PTR
         BY REFERENCE in_data
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

 MOVE 1 TO ELEMENT-NUMBER.
 MOVE 'A' TO SETOCTETVAL.
 CALL 'CORBA-SeqSet' USING
         BY REFERENCE in_data
         BY REFERENCE ELEMENT-NUMBER
         BY REFERENCE SETOCTETVAL
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

* TypeCodeオブジェクトの解放
 CALL 'CORBA_TypeCode__release' USING
         BY VALUE TYPE-CODE-PTR
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

**************************************************
* サーバメソッドの呼び出し
**************************************************
 DISPLAY 'Start CBLClass-call'.
 CALL 'CBLClass-call' USING
         BY VALUE     PROXY-PTR
         BY VALUE     in_data
         BY REFERENCE out_data
```

```
          BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success CBLClass-call'.

 PROG-END.

* 6. TSCユーザプロキシの削除
*    領域の解放
     IF TSCPRXY-CREATE-FLAG = 1 THEN
         CALL 'CBLClass_TSCprxy-DEL' USING
                 BY VALUE PROXY-PTR
                 BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
         IF NOT CORBA-NO-EXCEPTION THEN
             CALL 'OTM-EXCEPTION-HANDLER' USING
                     BY REFERENCE MAJOR
                     BY REFERENCE CORBA-ENVIRONMENT
             CALL 'TSCSysExcept-DELETE' USING
                     BY VALUE EXCEP OF CORBA-ENVIRONMENT
             MOVE 1 TO RETURN-CODE
         END-IF
     END-IF.

* 7. TSCデーモンへの接続解放
     IF GET-CLIENT-FLAG = 1 THEN
         CALL 'TSCAdm-releaseTSCClient' USING
                 BY VALUE CLIENT-PTR
                 BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
         IF NOT CORBA-NO-EXCEPTION THEN
             CALL 'OTM-EXCEPTION-HANDLER' USING
                     BY REFERENCE MAJOR
                     BY REFERENCE CORBA-ENVIRONMENT
             CALL 'TSCSysExcept-DELETE' USING
                     BY VALUE EXCEP OF CORBA-ENVIRONMENT
             MOVE 1 TO RETURN-CODE
         ELSE
             DISPLAY 'Success TSCAdm-releaseTSCClient'
         END-IF
     END-IF.

     IF DOMAIN-CREATE-FLAG = 1 THEN
         CALL 'TSCDomain-DELETE' USING
                 BY VALUE DOMAIN-PTR
                 BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
         IF NOT CORBA-NO-EXCEPTION THEN
             CALL 'OTM-EXCEPTION-HANDLER' USING
```

```
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            END-IF
        END-IF.

    * 8. TPBroker OTMの終了処理
        IF INIT-CLIENT-FLAG = 1 THEN
            CALL 'TSCAdm-endClient' USING
                BY REFERENCE CORBA-ENVIRONMENT
    * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            ELSE
                DISPLAY 'Success TSCAdm-endClient'
            END-IF
        END-IF.

     END PROGRAM CLIENT.
```

6.2.2 同期型呼び出しをするサーバアプリケーションの例（COBOL）

同期型呼び出しをするサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス CBLClass_TSCimpl にコードを記述します。また，雛形クラス CBLClass_TSCfact に TSC ユーザオブジェクトファクトリのコードを記述します。

（1）TSC ユーザオブジェクト（CBLClass_TSCimpl）と TSC ユーザオブジェクトファクトリ（CBLClass_TSCfact）のコード

```
*****************************************************
* Operation 'call'
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'call'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 CORBA-ENVIRONMENT.
  02 MAJOR                    PIC 9(9) COMP.
   88 CORBA-NO-EXCEPTION           VALUE 0.
```

```
   88 CORBA-USER-EXCEPTION       VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION     VALUE 2.
  02 EXCEP                    USAGE POINTER.
  02 FUNC-NAME                PIC X(256).

 01 TSC-SEQMAXLEN PIC 9(9) USAGE COMP.
 01 TSC-TC-1 USAGE POINTER.
 01 TSC-SEQENV.
   02 MAJOR PIC 9(9) USAGE COMP.
   02 EXCEP USAGE POINTER.
   02 FUNC-NAME PIC X(256).
 01 TSC-TC-2 USAGE POINTER.

* 必要に応じてデータ宣言を追加できます。
 01 ERR-CODE                  PIC S9(9) COMP.
 01 TYPE-CODE-PTR             USAGE POINTER.
 01 MY-OUT-DATA-LEN           PIC S9(9) COMP.
 01 ELEMENT-NUMBER            PIC S9(9) COMP.
 01 SETOCTETVAL               PIC X.

 LINKAGE SECTION.
 01 in_data USAGE POINTER.
 01 out_data USAGE POINTER.

* Do not change signature of this sub-program.
 PROCEDURE DIVISION
     USING
         BY VALUE in_data
         BY REFERENCE out_data.

* Write user own code.
* ユーザメソッドのコードを記述します。
 DISPLAY 'call method in CBLClass'.

* OUT属性引数の作成
* Octet TypeCode オブジェクトの作成
 CALL 'Create_CORBA_TypeCode' USING
         BY VALUE     10
         BY VALUE     1
         BY REFERENCE CORBA-ENVIRONMENT
         RETURNING TYPE-CODE-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
             BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

* Octet型Sequenceオブジェクトの生成
 MOVE 999999999 TO MY-OUT-DATA-LEN.
 CALL 'CORBA-SeqAlloc' USING
         BY REFERENCE MY-OUT-DATA-LEN
         BY REFERENCE TYPE-CODE-PTR
         BY REFERENCE out_data
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
```

```
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
             BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

 MOVE 1 TO ELEMENT-NUMBER.
 MOVE 'A' TO SETOCTETVAL.
 CALL 'CORBA-SeqSet' USING
         BY REFERENCE in_data
         BY REFERENCE ELEMENT-NUMBER
         BY REFERENCE SETOCTETVAL
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
             BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

 CALL 'CORBA_TypeCode__release' USING
     BY VALUE TYPE-CODE-PTR
     BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
             BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

 END PROGRAM 'call'.


**************************************************
* Constructor of 'CBLClass_TSCimpl'
**************************************************
* Constructor of OTM Object
* Implement.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING SKELETON-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
* コンストラクタの引数の数および型を変更することもできます。
```

```
* This sub-program must return a pointer
* that 'CBLClass_TSCsk-NEW' sub-program returns.
     CALL 'CBLClass_TSCsk-NEW'
          RETURNING SKELETON-POINTER.
 END PROGRAM 'CBLClass_TSCimpl-NEW'.

*****************************************************
* Destructor of 'CBLClass_TSCimpl'
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION USING
          BY VALUE SKELETON-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。

* This sub-program must call
* 'CBLClass_TSCsk-DEL' sub-program.
     CALL 'CBLClass_TSCsk-DEL' USING
          BY VALUE SKELETON-POINTER.
 END PROGRAM 'CBLClass_TSCimpl-DEL'.

*****************************************************
* Constructor of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
* 引数の数および型を変更することもできます。

* This sub-program must return a pointer that
* 'CBLClass_TSCfact-get' sub-program returns.
     CALL 'CBLClass_TSCfact-get'
          RETURNING FACTORY-POINTER.
 END PROGRAM 'CBLClass_TSCfact-NEW'.
```

```
*****************************************************
* Destructor of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。

* This sub-program must call
* 'CBLClass_TSCfact-rls'.sub-program.
     CALL 'CBLClass_TSCfact-rls' USING
         BY VALUE FACTORY-POINTER.
 END PROGRAM 'CBLClass_TSCfact-DEL'.


*****************************************************
* 'create' method of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-crt'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
* Do not change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER
     RETURNING OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを生成するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-NEW' returns.
     CALL 'CBLClass_TSCimpl-NEW'
         RETURNING OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-crt'.


*****************************************************
* 'destroy' method of CBLClass_TSCfact
```

```
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-dst'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
* Do not change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER
         BY VALUE OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを削除するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-DEL' returns.
     CALL 'CBLClass_TSCimpl-DEL' USING
         BY VALUE OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-dst'.


*****************************************************
* TSCCBLThread-beginThread of
*     TSCCBLThread
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-beginThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 BEGIN-THREAD-PTR USAGE POINTER.
 01 END-THREAD-PTR USAGE POINTER.
* Do not change signature of this sub-program.
 LINKAGE SECTION.
 01 THREAD-FACTORY-ID PIC S9(9) COMP.
 PROCEDURE DIVISION USING
         BY VALUE THREAD-FACTORY-ID.

* Write user own code.
* スレッド開始処理を記述します。
* 必要に応じて変更してください。

 END PROGRAM 'TSCCBLThread-beginThread'.


*****************************************************
* TSCCBLThread-endThread of
*     TSCCBLThreadFactory
*****************************************************
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-endThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* Do not change signature of this sub-program.
 LINKAGE SECTION.
 01 THREAD-FACTORY-ID PIC S9(9) COMP.
 PROCEDURE DIVISION USING
         BY VALUE THREAD-FACTORY-ID.

* Write user own code.
* スレッド終了処理を記述します。
* 必要に応じて変更してください。

 END PROGRAM 'TSCCBLThread-endThread'.
```

(2) サービス登録処理の流れ

1. COBOL adapter for TPBroker の初期化処理
2. TPBroker OTM の初期化処理
3. TSC デーモンへの接続
4. TSC ユーザアクセプタの生成および各種設定
5. TSC ルートアクセプタの生成および各種設定
6. TSC ルートアクセプタの活性化
7. 実行制御の受け渡し
8. TSC ルートアクセプタの非活性化
9. TSC ルートアクセプタの削除
10. TSC ユーザオブジェクトファクトリおよび TSC ユーザアクセプタの削除
11. TSC デーモンへの接続解放
12. TPBroker OTM の終了処理

(3) サービス登録処理のコード

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CORBA-SERVER-MAIN.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 CLASS CBLClass IS 'ABCfile_s'.
 DATA DIVISION.
* The following are the ORB pointers.
* These are necessary for all servers.
 WORKING-STORAGE SECTION.
```

```
 01 ORB-PTR                USAGE POINTER.

 01 DOMAIN-PTR             USAGE POINTER.
 01 SERVER-PTR             USAGE POINTER.
 01 FACTORY-PTR            USAGE POINTER.
 01 ACCEPTOR-PTR           USAGE POINTER.
 01 R-ACCEPTOR-PTR         USAGE POINTER.
 01 THREAD-FACT-PTR        USAGE POINTER.
 01 THREAD-FACT-ID         PIC S9(9) COMP.
 01 MY-DOMAIN-NAME         PIC X(10).
 01 MY-DOMAIN-NAME-PTR     USAGE POINTER.
 01 MY-DOMAIN-NAME-LEN     PIC S9(9) COMP.
 01 MY-TSCID               PIC X(10).
 01 MY-TSCID-PTR           USAGE POINTER.
 01 MY-TSCID-LEN           PIC S9(9) COMP.
 01 MY-DOMAIN-FLAG         PIC S9(9) COMP.
 01 ACCEPTOR-NAME-PTR      USAGE POINTER.
 01 ACCEPTOR-ID            PIC S9(9) COMP.
 01 P-COUNT                PIC S9(9) COMP.
 01 DEACT-MODE             PIC S9(9) COMP.
 01 R-ACCEPTOR-NAME        PIC X(10).
 01 R-ACCEPTOR-NAME-PTR    USAGE POINTER.
 01 R-ACCEPTOR-NAME-LEN    PIC S9(9) COMP VALUE 30.

 01 INIT-SERVER-FLAG       PIC S9(9) COMP.
 01 DOMAIN-CREATE-FLAG     PIC S9(9) COMP.
 01 GET-SERVER-FLAG        PIC S9(9) COMP.
 01 TSCFACT-CREATE-FLAG    PIC S9(9) COMP.
 01 TSCACPT-CREATE-FLAG    PIC S9(9) COMP.
 01 RACPT-CREATE-FLAG      PIC S9(9) COMP.
 01 RACPT-ACTIVATE-FLAG    PIC S9(9) COMP.

 01 CORBA-ENVIRONMENT.
  02 MAJOR PIC 9(9) USAGE COMP.
   88 CORBA-NO-EXCEPTION     VALUE 0.
   88 CORBA-USER-EXCEPTION   VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION VALUE 2.
  02 EXCEP USAGE POINTER.
  02 FUNC-NAME PIC X(256).

 LINKAGE SECTION.
 01 ARGC PIC S9(9) USAGE COMP.
 01 ARGV USAGE POINTER.

 PROCEDURE DIVISION USING
         BY VALUE ARGC
         BY VALUE ARGV.

     MOVE 0 TO RETURN-CODE.
     MOVE 0 TO INIT-SERVER-FLAG.
     MOVE 0 TO DOMAIN-CREATE-FLAG.
     MOVE 0 TO GET-SERVER-FLAG.
     MOVE 0 TO TSCFACT-CREATE-FLAG.
     MOVE 0 TO TSCACPT-CREATE-FLAG.
     MOVE 0 TO RACPT-CREATE-FLAG.
     MOVE 0 TO RACPT-ACTIVATE-FLAG.

* 1. COBOL adapter for TPBrokerの初期化処理
```

```
* First, call the skeleton and class initializers.
     CALL 'CORBA-SKEL-INIT-ABCfile'.
     CALL 'CBLClass-CLASS-INIT' USING BY VALUE CBLClass.

* ORBの初期化
     CALL 'CORBA_orb_init' USING
         BY REFERENCE ARGC
         BY REFERENCE ARGV
         BY REFERENCE CORBA-ENVIRONMENT
         RETURNING    ORB-PTR.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'CORBA_FreeException' USING
                 EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success CORBA_orb_init'.

* 2. TPBroker OTMの初期化処理
     CALL 'TSCAdm-initServer' USING
             BY REFERENCE ARGC
             BY REFERENCE ARGV
             BY VALUE      ORB-PTR
             BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCAdm-initServer'.
     MOVE 1 TO INIT-SERVER-FLAG.

* 3. TSCデーモンへの接続
     SET MY-DOMAIN-NAME-PTR TO NULL.
     SET MY-TSCID-PTR TO NULL.
     MOVE 1 TO MY-DOMAIN-FLAG.
     CALL 'TSCDomain-NEW' USING
             BY VALUE MY-DOMAIN-NAME-PTR
             BY VALUE MY-TSCID-PTR
             BY VALUE MY-DOMAIN-FLAG
             BY REFERENCE CORBA-ENVIRONMENT
         RETURNING DOMAIN-PTR.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
```

```
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
            GO TO PROG-END
        END-IF.
        DISPLAY 'Success TSCDomain-NEW'.
        MOVE 1 TO DOMAIN-CREATE-FLAG.

    * TSCServerの取得
        CALL 'TSCAdm-getTSCServer' USING
                BY VALUE DOMAIN-PTR
                BY REFERENCE CORBA-ENVIRONMENT
            RETURNING SERVER-PTR.
    * 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
            GO TO PROG-END
        END-IF.
        DISPLAY 'Success TSCAdm-getTSCServer'.
        MOVE 1 TO GET-SERVER-FLAG.

    * 4. TSCユーザアクセプタの生成および各種設定
    * CBLClass_TSCfactの生成
        CALL 'CBLClass_TSCfact-NEW'
            RETURNING FACTORY-PTR.
        DISPLAY 'Success CBLClass_TSCfact-NEW'.
        MOVE 1 TO TSCFACT-CREATE-FLAG.

    * TSCAcceptorの生成
        SET ACCEPTOR-NAME-PTR TO NULL.
        CALL 'CBLClass_TSCacpt-NEW' USING
                BY VALUE FACTORY-PTR
                BY VALUE ACCEPTOR-NAME-PTR
                BY REFERENCE CORBA-ENVIRONMENT
            RETURNING ACCEPTOR-PTR.
    * 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
            GO TO PROG-END
        END-IF.
        DISPLAY 'Success CBLClass_TSCacpt-NEW'.
        MOVE 1 TO TSCACPT-CREATE-FLAG.

    * 5. TSCルートアクセプタの生成および各種設定
        SET THREAD-FACT-PTR TO NULL.
        CALL 'TSCRAcceptor-create' USING
                BY VALUE SERVER-PTR
                BY VALUE THREAD-FACT-PTR
```

```
            BY REFERENCE CORBA-ENVIRONMENT
        RETURNING R-ACCEPTOR-PTR.
* 例外チェック
    IF NOT CORBA-NO-EXCEPTION THEN
        CALL 'OTM-EXCEPTION-HANDLER' USING
                BY REFERENCE MAJOR
                BY REFERENCE CORBA-ENVIRONMENT
        CALL 'TSCSysExcept-DELETE' USING
            BY VALUE EXCEP OF CORBA-ENVIRONMENT
        MOVE 1 TO RETURN-CODE
        GO TO PROG-END
    END-IF.
    DISPLAY 'Success TSCRAcceptor-create'.
    MOVE 1 TO RACPT-CREATE-FLAG.

* TSCアクセプタの登録
    CALL 'TSCRAcceptor-registerAcceptor' USING
            BY VALUE R-ACCEPTOR-PTR
            BY VALUE ACCEPTOR-PTR
            BY REFERENCE CORBA-ENVIRONMENT
        RETURNING ACCEPTOR-ID.
* 例外チェック
    IF NOT CORBA-NO-EXCEPTION THEN
        CALL 'OTM-EXCEPTION-HANDLER' USING
                BY REFERENCE MAJOR
                BY REFERENCE CORBA-ENVIRONMENT
        CALL 'TSCSysExcept-DELETE' USING
            BY VALUE EXCEP OF CORBA-ENVIRONMENT
        MOVE 1 TO RETURN-CODE
        GO TO PROG-END
    END-IF.
    DISPLAY 'Success TSCRAcceptor-registerAcceptor'.

* 6. TSCルートアクセプタの活性化
    MOVE 'serviceX' TO R-ACCEPTOR-NAME.
    CALL 'CORBA_string_set' USING
            BY REFERENCE R-ACCEPTOR-NAME-PTR
            BY REFERENCE R-ACCEPTOR-NAME-LEN
            BY REFERENCE R-ACCEPTOR-NAME.
    CALL 'TSCRAcceptor-activate' USING
            BY VALUE R-ACCEPTOR-PTR
            BY VALUE R-ACCEPTOR-NAME-PTR
            BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
    IF NOT CORBA-NO-EXCEPTION THEN
        CALL 'OTM-EXCEPTION-HANDLER' USING
                BY REFERENCE MAJOR
                BY REFERENCE CORBA-ENVIRONMENT
        CALL 'TSCSysExcept-DELETE' USING
            BY VALUE EXCEP OF CORBA-ENVIRONMENT
        MOVE 1 TO RETURN-CODE
        GO TO PROG-END
    END-IF.
    DISPLAY 'Success TSCRAcceptor-activate'.
    MOVE 1 TO RACPT-ACTIVATE-FLAG.

* 7. 実行制御の受け渡し
```

```
     DISPLAY 'Start TSCAdm-serverMainloop'.
     CALL 'TSCAdm-serverMainloop' USING
             BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCAdm-serverMainloop'.

 PROG-END.

* 8. TSCルートアクセプタの非活性化
     IF RACPT-ACTIVATE-FLAG = 1 THEN
         MOVE 0 TO DEACT-MODE
         CALL 'TSCRAcceptor-deactivate' USING
                 BY VALUE R-ACCEPTOR-PTR
                 BY VALUE DEACT-MODE
                 BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
         IF NOT CORBA-NO-EXCEPTION THEN
             CALL 'OTM-EXCEPTION-HANDLER' USING
                     BY REFERENCE MAJOR
                     BY REFERENCE CORBA-ENVIRONMENT
             CALL 'TSCSysExcept-DELETE' USING
                     BY VALUE EXCEP OF CORBA-ENVIRONMENT
             MOVE 1 TO RETURN-CODE
         ELSE
             DISPLAY 'Success TSCRAcceptor-deactivate'
         END-IF
     END-IF.

* 9. TSCルートアクセプタの削除
     IF RACPT-CREATE-FLAG = 1 THEN
         CALL 'TSCRAcceptor-destroy' USING
                 BY VALUE R-ACCEPTOR-PTR
                 BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
         IF NOT CORBA-NO-EXCEPTION THEN
             CALL 'OTM-EXCEPTION-HANDLER' USING
                     BY REFERENCE MAJOR
                     BY REFERENCE CORBA-ENVIRONMENT
             CALL 'TSCSysExcept-DELETE' USING
                     BY VALUE EXCEP OF CORBA-ENVIRONMENT
             MOVE 1 TO RETURN-CODE
         ELSE
             DISPLAY 'Success TSCRAcceptor-destroy'
         END-IF
     END-IF.

* 10. ユーザオブジェクトファクトリおよびTSCユーザアクセプタの削除
     IF TSCACPT-CREATE-FLAG = 1 THEN
```

```
            CALL 'CBLClass_TSCacpt-DEL' USING
                    BY VALUE ACCEPTOR-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
    * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            END-IF
        END-IF.

    * 領域の解放
        IF TSCFACT-CREATE-FLAG = 1 THEN
            CALL 'CBLClass_TSCfact-DEL' USING
                    BY VALUE FACTORY-PTR
        END-IF.

    * 11. TSCデーモンへの接続解放
        IF GET-SERVER-FLAG = 1 THEN
            CALL 'TSCAdm-releaseTSCServer' USING
                    BY VALUE SERVER-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
    * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            ELSE
                DISPLAY 'Success TSCAdm-releaseTSCServer'
            END-IF
        END-IF.

        IF DOMAIN-CREATE-FLAG = 1 THEN
            CALL 'TSCDomain-DELETE' USING
                    BY VALUE DOMAIN-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
    * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            END-IF
        END-IF.

    * 12. TPBroker OTMの終了処理
        IF INIT-SERVER-FLAG = 1 THEN
            CALL 'TSCAdm-endServer' USING
                    BY REFERENCE CORBA-ENVIRONMENT
```

```
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        ELSE
            DISPLAY 'Success TSCAdm-endServer'
        END-IF
    END-IF.

END PROGRAM CORBA-SERVER-MAIN.
```

6.2.3 例外処理のコードの例（COBOL）

アプリケーションプログラムで例外を検出した場合に呼び出される副プログラムとしての例外処理の例を示します。なお，この例外処理はクライアントアプリケーションおよびサーバアプリケーションに共通です。

```
IDENTIFICATION DIVISION.
PROGRAM-ID. OTM-EXCEPTION-HANDLER.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.

DATA DIVISION.
WORKING-STORAGE SECTION.

COPY TSCSysExcept.

01 OTM-ERROR-CODE            PIC S9(9) COMP.
01 OTM-DETAIL-CODE           PIC S9(9) COMP.
01 OTM-PLACE-CODE            PIC S9(9) COMP.
01 OTM-COMPLET-CODE          PIC S9(9) COMP.
01 OTM-MAINTENANCE-CODE1     PIC S9(9) COMP.
01 OTM-MAINTENANCE-CODE2     PIC S9(9) COMP.
01 OTM-MAINTENANCE-CODE3     PIC S9(9) COMP.
01 OTM-MAINTENANCE-CODE4     PIC S9(9) COMP.

LINKAGE SECTION.
01 EXCEPTION-TYPE        PIC 9(9) COMP.
01 CORBA-ENVIRONMENT.
 02 MAJOR                PIC 9(9) COMP.
  88 CORBA-NO-EXCEPTION     VALUE 0.
  88 CORBA-USER-EXCEPTION   VALUE 1.
  88 CORBA-SYSTEM-EXCEPTION VALUE 2.
 02 EXCEP                 USAGE POINTER.
 02 FUNC-NAME             PIC X(256).

PROCEDURE DIVISION USING
        BY REFERENCE EXCEPTION-TYPE
        BY REFERENCE CORBA-ENVIRONMENT.

    DISPLAY '!!!! An Exception Occurs in '
```

```
            FUNC-NAME OF CORBA-ENVIRONMENT ' !!!!'

        IF CORBA-SYSTEM-EXCEPTION
            DISPLAY 'OTM System Exception Occurs.'

            CALL 'TSCSysExcept-getErrorCode' USING
                    BY VALUE EXCEP
                RETURNING OTM-ERROR-CODE
            DISPLAY 'Error-code is ' OTM-ERROR-CODE

            CALL 'TSCSysExcept-getDetailCode' USING
                    BY VALUE EXCEP
                RETURNING OTM-DETAIL-CODE
            DISPLAY 'Detail-code is ' OTM-DETAIL-CODE

            CALL 'TSCSysExcept-getPlaceCode' USING
                    BY VALUE EXCEP
                RETURNING OTM-PLACE-CODE
            DISPLAY 'Place-code is ' OTM-PLACE-CODE

            CALL 'TSCSysExcept-getCompletion' USING
                    BY VALUE EXCEP
                RETURNING OTM-COMPLET-CODE
            DISPLAY 'Completion-code is ' OTM-COMPLET-CODE

            CALL 'TSCSysExcept-getMaintenance1' USING
                    BY VALUE EXCEP
                RETURNING OTM-MAINTENANCE-CODE1
            DISPLAY 'Mainte-code1 is ' OTM-MAINTENANCE-CODE1

            CALL 'TSCSysExcept-getMaintenance2' USING
                    BY VALUE EXCEP
                RETURNING OTM-MAINTENANCE-CODE2
            DISPLAY 'Mainte-code2 is ' OTM-MAINTENANCE-CODE2

            CALL 'TSCSysExcept-getMaintenance3' USING
                    BY VALUE EXCEP
                RETURNING OTM-MAINTENANCE-CODE3
            DISPLAY 'Mainte-code3 is ' OTM-MAINTENANCE-CODE3

            CALL 'TSCSysExcept-getMaintenance4' USING
                    BY VALUE EXCEP
                RETURNING OTM-MAINTENANCE-CODE4
            DISPLAY 'Mainte-code4 is ' OTM-MAINTENANCE-CODE4
        ELSE
      * ユーザ例外処理
            DISPLAY 'User Exception Occurs.'
        END-IF.

     END PROGRAM OTM-EXCEPTION-HANDLER.


     IDENTIFICATION DIVISION.
     PROGRAM-ID. EXCEPTION-HANDLER.
     ENVIRONMENT DIVISION.
     CONFIGURATION SECTION.

     DATA DIVISION.
```

```
WORKING-STORAGE SECTION.

01 EXCEPTION-NAME            PIC X(50).
01 EXCEPTION-NAME-PTR        USAGE POINTER.
01 EXCEPTION-NAME-LEN        PIC S9(9) COMP VALUE 50.
01 MINOR-CODE                PIC S9(9) COMP.
01 COMPLETED-CODE            PIC S9(9) COMP.

LINKAGE SECTION.
01 EXCEPTION-TYPE        PIC 9(9) COMP.
01 CORBA-ENVIRONMENT.
 02 MAJOR                PIC 9(9) COMP.
  88 CORBA-NO-EXCEPTION     VALUE 0.
  88 CORBA-USER-EXCEPTION   VALUE 1.
  88 CORBA-SYSTEM-EXCEPTION VALUE 2.
 02 EXCEP                USAGE POINTER.
 02 FUNC-NAME            PIC X(256).


PROCEDURE DIVISION USING
        BY REFERENCE EXCEPTION-TYPE
        BY REFERENCE CORBA-ENVIRONMENT.

    DISPLAY '!!!! An Exception Occurs in '
        FUNC-NAME OF CORBA-ENVIRONMENT

    IF CORBA-SYSTEM-EXCEPTION
        DISPLAY 'CORBA System Exception Occurs.'

    CALL 'CORBA-get-exception-name' USING
            BY REFERENCE EXCEP
        RETURNING EXCEPTION-NAME-PTR.
    CALL 'CORBA_string_get' USING
            BY REFERENCE EXCEPTION-NAME-PTR
            BY REFERENCE EXCEPTION-NAME-LEN
            BY REFERENCE EXCEPTION-NAME.

    DISPLAY 'Exception name is' EXCEPTION-NAME.

    IF CORBA-SYSTEM-EXCEPTION
        CALL 'CORBA-SysExcept-get-minor' USING
                BY VALUE EXCEP
            RETURNING MINOR-CODE
        DISPLAY 'Minor-code is ' MINOR-CODE

        CALL 'CORBA-SysExcept-get-completed' USING
                BY VALUE EXCEP
            RETURNING COMPLETED-CODE
        DISPLAY 'Completed-code is ' COMPLETED-CODE

    ELSE
        DISPLAY 'User Exception Occurs.'
    END-IF.

END PROGRAM EXCEPTION-HANDLER.
```

6.2.4 同期型呼び出しをするアプリケーションプログラムの実行時の処理（COBOL）

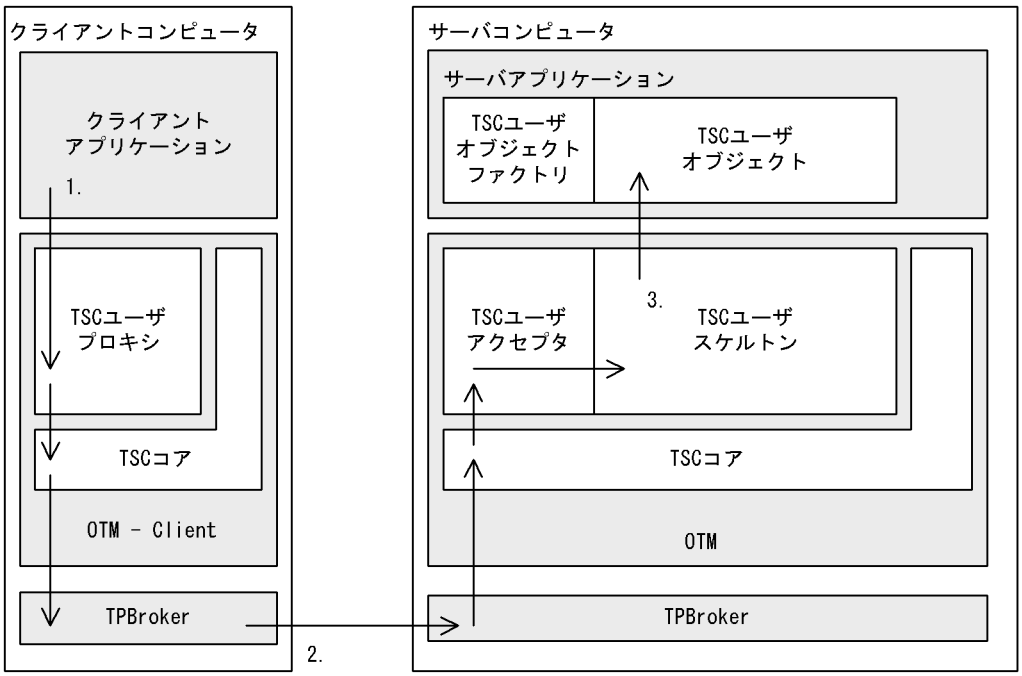
同期型呼び出しをするアプリケーションプログラムを実行した場合の処理シーケンスを示します。

（1） クライアントアプリケーションの開始（COBOL）

クライアントアプリケーションを開始すると，サービスの利用処理が実行されます。CBLClass_TSCprxy のメソッドを呼び出すと，CBLClass_TSCsk を経由し，ユーザが実装した CBLClass_TSCimpl のメソッドが呼び出されます。

クライアントアプリケーションの開始の流れを次の図に示します。

図 6-2 同期型呼び出しをするクライアントアプリケーションの開始（COBOL）



1. TSC **ユーザプロキシのメソッドが呼び出されます。**
2. **スケジューリングおよび** TPBroker **の通信が実行されます。**
3. TSC **ユーザオブジェクトのメソッドが呼び出されます。**

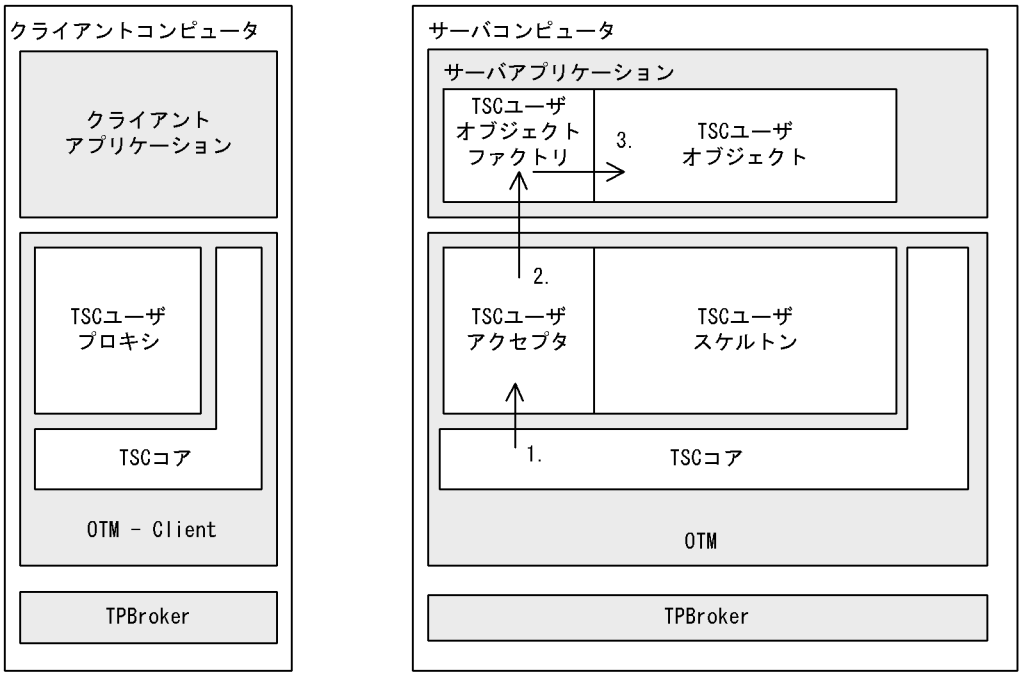
（2） サーバアプリケーションの開始（COBOL）

サーバアプリケーションを開始すると，サービスの登録処理が実行されます。TSCRAcceptor-activate 副プログラムを呼び出すと，CBLClass_TSCfact に

CBLClass_TSCimpl の生成依頼が発行されます。

サーバアプリケーションの開始の流れを次の図に示します。

図 6-3　同期型呼び出しをするサーバアプリケーションの開始（COBOL）



1. TSC **ユーザオブジェクトの活性化処理が実行されます。**
2. TSC **ユーザオブジェクトのインプリメンテーションが生成されます。**
3. TSC **ユーザオブジェクトが生成されます。**

6.3 非応答型呼び出しをするアプリケーションプログラム（COBOL）

非応答型呼び出しをするアプリケーションプログラムの COBOL での作成例を示します。これらのサンプルコードは製品で提供されています。

● ユーザ定義 IDL インタフェースの例

ユーザ定義 IDL インタフェースの例を次に示します。

```
//
// "XYZfile.idl"
//

interface CBLClass
{
  oneway void callOnly(in long in_data);
};
```

● IDL コンパイラが生成するクラス

COBOL adapter for TPBroker の IDL コンパイラはユーザ定義 IDL インタフェースから次のファイルを生成します。

- XYZfile_c.ocb
- XYZfile_c.ocb

● トランザクションフレームジェネレータが生成するクラス

OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。ただし，実際に生成されるのはクラスを使用するための副プログラムです。副プログラムの名称は次のとおりになります。

"クラス名"－"メソッド名"

表 6-2 非応答型呼び出しをするアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（COBOL）

分類	ファイル名	副プログラムのプリフィックス（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• XYZfile_TSC_c.ocb	• CBLClass_TSCprxy（TSC ユーザプロキシ）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• XYZfile_TSC_s.ocb	• CBLClass_TSCsk（TSC ユーザスケルトン） • CBLClass_TSCacpt（TSC ユーザアクセプタ）
雛形クラス	• XYZfile_TSC_t.ocb	• CBLClass_TSCimpl（TSC ユーザオブジェクト） • CBLClass_TSCfact（TSC ユーザオブジェクトファクトリ） • TSCCBLThreadFactory（TSC ユーザスレッドファクトリ） • TSCCBLThread（TSC ユーザスレッド）

6.3.1 非応答型呼び出しをするクライアントアプリケーションの例（COBOL）

非応答型呼び出しをするクライアントアプリケーションの処理の流れとコードの例を示します。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

なお，非応答型呼び出しをするクライアントアプリケーションの例外処理は，同期型呼び出しの場合と同様です。「6.2.3　例外処理のコードの例（COBOL）」を参照してください。

（1） サービス利用処理の流れ

1. COBOL adapter for TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
6. TSC **ユーザプロキシの削除**
7. TSC **デーモンへの接続解放**
8. TPBroker OTM **の終了処理**

（2） サービス利用処理のコード

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CLIENT.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.

DATA DIVISION.
WORKING-STORAGE SECTION.

01 ORB-PTR                USAGE POINTER.
01 DOMAIN-PTR             USAGE POINTER.
01 CLIENT-PTR             USAGE POINTER.
01 CLIENT-WAY             PIC S9(9) COMP.
01 MY-DOMAIN-NAME         PIC X(10).
01 MY-DOMAIN-NAME-PTR     USAGE POINTER.
01 MY-DOMAIN-NAME-LEN     PIC S9(9) COMP.
01 MY-TSCID               PIC X(10).
01 MY-TSCID-PTR           USAGE POINTER.
01 MY-TSCID-LEN           PIC S9(9) COMP.
01 MY-DOMAIN-FLAG         PIC S9(9) COMP VALUE 1.
01 ACCEPTOR-NAME          PIC X(20).
01 ACCEPTOR-NAME-PTR      USAGE POINTER VALUE NULL.
01 ACCEPTOR-NAME-LEN      PIC S9(9) COMP.
01 PROXY-PTR              USAGE POINTER.
```

```
 01 INIT-CLIENT-FLAG      PIC S9(9) COMP.
 01 DOMAIN-CREATE-FLAG    PIC S9(9) COMP.
 01 GET-CLIENT-FLAG       PIC S9(9) COMP.
 01 TSCPRXY-CREATE-FLAG   PIC S9(9) COMP.

 01 CORBA-ENVIRONMENT.
  02 MAJOR                PIC 9(9) COMP.
   88 CORBA-NO-EXCEPTION     VALUE 0.
   88 CORBA-USER-EXCEPTION   VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION VALUE 2.
  02 EXCEP                 USAGE POINTER.
  02 FUNC-NAME             PIC X(256).

 01 ERR-CODE               PIC S9(9) COMP.

* Sequence
 01 in_data                USAGE POINTER.

 LINKAGE SECTION.
 01 ARGC                   PIC 9(9) COMP.
 01 ARGV                   USAGE POINTER.

 PROCEDURE DIVISION USING
         BY VALUE ARGC
         BY VALUE ARGV.

 MOVE 0 TO RETURN-CODE.
 MOVE 0 TO INIT-CLIENT-FLAG.
 MOVE 0 TO DOMAIN-CREATE-FLAG.
 MOVE 0 TO GET-CLIENT-FLAG.
 MOVE 0 TO TSCPRXY-CREATE-FLAG.

* ORBの初期化処理
 CALL 'CORBA-STUB-INIT-XYZfile'.

* 1. COBOL adapter for TPBrokerの初期化処理
 CALL 'CORBA_orb_init' USING
         BY REFERENCE ARGC
         BY REFERENCE ARGV
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING ORB-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success CORBA_orb_init'.

* 2. TPBroker OTMの初期化処理
 CALL 'TSCAdm-initClient' USING
         BY REFERENCE ARGC
         BY REFERENCE ARGV
```

```
          BY VALUE     ORB-PTR
          BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success TSCAdm-initClient'.
 MOVE 1 TO INIT-CLIENT-FLAG.

* 3. TSCデーモンへの接続
* (1) TSCDomainの生成
 SET MY-DOMAIN-NAME-PTR TO NULL.
 SET MY-TSCID-PTR TO NULL.
 MOVE 1 TO MY-DOMAIN-FLAG.
 CALL 'TSCDomain-NEW' USING
          BY VALUE     MY-DOMAIN-NAME-PTR
          BY VALUE     MY-TSCID-PTR
          BY VALUE     MY-DOMAIN-FLAG
          BY REFERENCE CORBA-ENVIRONMENT
     RETURNING DOMAIN-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success TSCDomain-NEW:DomainName = '
                                      MY-DOMAIN-NAME.
 MOVE 1 TO DOMAIN-CREATE-FLAG.

* (2) TSCClientの取得
 MOVE 1 TO CLIENT-WAY.
 CALL 'TSCAdm-getTSCClient' USING
          BY VALUE     DOMAIN-PTR
          BY VALUE     CLIENT-WAY
          BY REFERENCE CORBA-ENVIRONMENT
     RETURNING CLIENT-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
```

```
 DISPLAY 'Success TSCAdm-getTSCClient'.
 MOVE 1 TO GET-CLIENT-FLAG.

* 4. TSCユーザプロキシの生成および各種設定
*    IDLインタフェース"CBLClass"用のTSCProxy生成
 SET ACCEPTOR-NAME-PTR TO NULL.
 CALL 'CBLClass_TSCprxy-NEW' USING
         BY VALUE     CLIENT-PTR
         BY VALUE     ACCEPTOR-NAME-PTR
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING PROXY-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success CBLClass_TSCprxy-NEW'.
 MOVE 1 TO TSCPRXY-CREATE-FLAG.

* 5. TSCユーザプロキシのメソッド呼び出し
*    （サーバ側のオブジェクトの呼び出し）
 MOVE 100 TO in_data.
**************************************************
* サーバメソッドの呼び出し
**************************************************
 DISPLAY 'Start CBLClass-callOnly'.
 CALL 'CBLClass-callOnly' USING
         BY VALUE     PROXY-PTR
         BY VALUE     in_data
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success CBLClass-callOnly'.

 PROG-END.

* 6. TSCユーザプロキシの削除
*    領域の解放
     IF TSCPRXY-CREATE-FLAG = 1 THEN
         CALL 'CBLClass_TSCprxy-DEL' USING
                 BY VALUE PROXY-PTR
                 BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
         IF NOT CORBA-NO-EXCEPTION THEN
```

```
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        END-IF
    END-IF.

* 7. TSCデーモンへの接続解放
    IF GET-CLIENT-FLAG = 1 THEN
        CALL 'TSCAdm-releaseTSCClient' USING
                BY VALUE CLIENT-PTR
                BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        ELSE
            DISPLAY 'Success TSCAdm-releaseTSCClient'
        END-IF
    END-IF.

    IF DOMAIN-CREATE-FLAG = 1 THEN
        CALL 'TSCDomain-DELETE' USING
                BY VALUE DOMAIN-PTR
                BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        END-IF
    END-IF.

* 8. TPBroker OTMの終了処理
    IF INIT-CLIENT-FLAG = 1 THEN
        CALL 'TSCAdm-endClient' USING
                BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        ELSE
            DISPLAY 'Success TSCAdm-endClient'
        END-IF
```

```
        END-IF.

    END PROGRAM CLIENT.
```

6.3.2 非応答型呼び出しをするサーバアプリケーションの例（COBOL）

非応答型呼び出しをするサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス CBLClass_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス CBLClass_TSCfact に TSC ユーザオブジェクトファクトリのコードを記述します。

なお，非応答型呼び出しをするサーバアプリケーションの例外処理は，同期型呼び出しの場合と同様です。「6.2.3　例外処理のコードの例（COBOL）」を参照してください。

（1）TSC ユーザオブジェクト（CBLClass_TSCimpl）と TSC ユーザオブジェクトファクトリ（CBLClass_TSCfact）のコード

```
*****************************************************
* Operation 'callOnly'
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'callOnly'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 CORBA-ENVIRONMENT.
  02 MAJOR                    PIC 9(9) COMP.
   88 CORBA-NO-EXCEPTION          VALUE 0.
   88 CORBA-USER-EXCEPTION        VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION      VALUE 2.
  02 EXCEP                    USAGE POINTER.
  02 FUNC-NAME                PIC X(256).

 LINKAGE SECTION.
 01 in_data PIC S9(9) COMP.

* Do not change signature of this sub-program.
 PROCEDURE DIVISION
     USING
         BY VALUE in_data.

* Write user own code.
* ユーザメソッドのコードを記述します。
 DISPLAY 'callOnly method in CBLClass'.

 END PROGRAM 'callOnly'.
```

```
*****************************************************
* Constructor of 'CBLClass_TSCimpl'
*****************************************************
* Constructor of OTM Object Implement.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING SKELETON-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
* コンストラクタの引数の数および型を変更することもできます。

* This sub-program must return a pointer
* that 'CBLClass_TSCsk-NEW' sub-program returns.
     CALL 'CBLClass_TSCsk-NEW'
         RETURNING SKELETON-POINTER.
 END PROGRAM 'CBLClass_TSCimpl-NEW'.

*****************************************************
* Destructor of 'CBLClass_TSCimpl'
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE SKELETON-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。

* This sub-program must call
* 'CBLClass_TSCsk-DEL' sub-program.
     CALL 'CBLClass_TSCsk-DEL' USING
         BY VALUE SKELETON-POINTER.
 END PROGRAM 'CBLClass_TSCimpl-DEL'.

*****************************************************
* Constructor of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
```

```
 PROGRAM-ID. 'CBLClass_TSCfact-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
* 引数の数および型を変更することもできます。

* This sub-program must return a pointer that
* 'CBLClass_TSCfact-get' sub-program returns.
     CALL 'CBLClass_TSCfact-get'
         RETURNING FACTORY-POINTER.
 END PROGRAM 'CBLClass_TSCfact-NEW'.


*****************************************************
* Destructor of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
* This sub-program must call
* 'CBLClass_TSCfact-rls'.sub-program.
     CALL 'CBLClass_TSCfact-rls' USING
         BY VALUE FACTORY-POINTER.
 END PROGRAM 'CBLClass_TSCfact-DEL'.


*****************************************************
* 'create' method of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-crt'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
```

```
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
* Do not change signature of this sub-program.
 PROCEDURE DIVISION USING
          BY VALUE FACTORY-POINTER
      RETURNING OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを生成するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-NEW' returns.
     CALL 'CBLClass_TSCimpl-NEW'
          RETURNING OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-crt'.


*****************************************************
* 'destroy' method of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-dst'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
* Do not change signature of this sub-program.
 PROCEDURE DIVISION USING
          BY VALUE FACTORY-POINTER
          BY VALUE OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを削除するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-DEL' returns.
     CALL 'CBLClass_TSCimpl-DEL' USING
          BY VALUE OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-dst'.


*****************************************************
* TSCCBLThread-beginThread of
*     TSCCBLThread
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-beginThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
```

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 BEGIN-THREAD-PTR USAGE POINTER.
 01 END-THREAD-PTR USAGE POINTER.
* Do not change signature of this sub-program.
 LINKAGE SECTION.
 01 THREAD-FACTORY-ID PIC S9(9) COMP.
 PROCEDURE DIVISION USING
         BY VALUE THREAD-FACTORY-ID.

* Write user own code.
* スレッド開始処理を記述します。
* 必要に応じて変更してください。

 END PROGRAM 'TSCCBLThread-beginThread'.


*****************************************************
* TSCCBLThread-endThread of
*      TSCCBLThreadFactory
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-endThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* Do not change signature of this sub-program.
 LINKAGE SECTION.
 01 THREAD-FACTORY-ID PIC S9(9) COMP.
 PROCEDURE DIVISION USING
         BY VALUE THREAD-FACTORY-ID.

* Write user own code.
* スレッド終了処理を記述します。
* 必要に応じて変更してください。

 END PROGRAM 'TSCCBLThread-endThread'.
```

（2） サービス登録処理の流れ

1. COBOL adapter for TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザアクセプタの生成および各種設定**
5. TSC **ルートアクセプタの生成および各種設定**
6. TSC **ルートアクセプタの活性化**
7. **実行制御の受け渡し**
8. TSC **ルートアクセプタの非活性化**

9. TSC ルートアクセプタの削除

10. TSC ユーザオブジェクトファクトリおよび TSC ユーザアクセプタの削除

11. TSC デーモンへの接続解放

12. TPBroker OTM の終了処理

（3） サービス登録処理のコード

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CORBA-SERVER-MAIN.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 CLASS CBLClass IS 'XYZfile_s'.
 DATA DIVISION.
* The following are the ORB pointers.
* These are necessary for all servers.
 WORKING-STORAGE SECTION.
 01 ORB-PTR        USAGE POINTER.

 01 DOMAIN-PTR            USAGE POINTER.
 01 SERVER-PTR            USAGE POINTER.
 01 FACTORY-PTR           USAGE POINTER.
 01 ACCEPTOR-PTR          USAGE POINTER.
 01 R-ACCEPTOR-PTR        USAGE POINTER.
 01 THREAD-FACT-PTR       USAGE POINTER.
 01 THREAD-FACT-ID        PIC S9(9) COMP.
 01 MY-DOMAIN-NAME        PIC X(10).
 01 MY-DOMAIN-NAME-PTR    USAGE POINTER.
 01 MY-DOMAIN-NAME-LEN    PIC S9(9) COMP.
 01 MY-TSCID              PIC X(10).
 01 MY-TSCID-PTR          USAGE POINTER.
 01 MY-TSCID-LEN          PIC S9(9) COMP.
 01 MY-DOMAIN-FLAG        PIC S9(9) COMP.
 01 ACCEPTOR-NAME-PTR     USAGE POINTER.
 01 ACCEPTOR-ID           PIC S9(9) COMP.
 01 P-COUNT               PIC S9(9) COMP.
 01 DEACT-MODE            PIC S9(9) COMP.
 01 R-ACCEPTOR-NAME       PIC X(10).
 01 R-ACCEPTOR-NAME-PTR   USAGE POINTER.
 01 R-ACCEPTOR-NAME-LEN   PIC S9(9) COMP VALUE 30.

 01 INIT-SERVER-FLAG      PIC S9(9) COMP.
 01 DOMAIN-CREATE-FLAG    PIC S9(9) COMP.
 01 GET-SERVER-FLAG       PIC S9(9) COMP.
 01 TSCFACT-CREATE-FLAG   PIC S9(9) COMP.
 01 TSCACPT-CREATE-FLAG   PIC S9(9) COMP.
 01 RACPT-CREATE-FLAG     PIC S9(9) COMP.
 01 RACPT-ACTIVATE-FLAG   PIC S9(9) COMP.

 01 CORBA-ENVIRONMENT.
  02 MAJOR PIC 9(9) USAGE COMP.
   88 CORBA-NO-EXCEPTION     VALUE 0.
   88 CORBA-USER-EXCEPTION   VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION VALUE 2.
```

```
  02 EXCEP USAGE POINTER.
  02 FUNC-NAME PIC X(256).

 LINKAGE SECTION.
 01 ARGC PIC S9(9) USAGE COMP.
 01 ARGV USAGE POINTER.

 PROCEDURE DIVISION USING
         BY VALUE ARGC
         BY VALUE ARGV.

     MOVE 0 TO RETURN-CODE.
     MOVE 0 TO INIT-SERVER-FLAG.
     MOVE 0 TO DOMAIN-CREATE-FLAG.
     MOVE 0 TO GET-SERVER-FLAG.
     MOVE 0 TO TSCFACT-CREATE-FLAG.
     MOVE 0 TO TSCACPT-CREATE-FLAG.
     MOVE 0 TO RACPT-CREATE-FLAG.
     MOVE 0 TO RACPT-ACTIVATE-FLAG.

* 1. COBOL adapter for TPBrokerの初期化処理
* First, call the skeleton and class initializers.
     CALL 'CORBA-SKEL-INIT-XYZfile'.
     CALL 'CBLClass-CLASS-INIT' USING BY VALUE CBLClass.

* ORBの初期化
     CALL 'CORBA_orb_init' USING
         BY REFERENCE ARGC
         BY REFERENCE ARGV
         BY REFERENCE CORBA-ENVIRONMENT
         RETURNING    ORB-PTR.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'CORBA_FreeException' USING
                 EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success CORBA_orb_init'.

* 2. TPBroker OTMの初期化処理
     CALL 'TSCAdm-initServer' USING
             BY REFERENCE ARGC
             BY REFERENCE ARGV
             BY VALUE      ORB-PTR
             BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
```

```
            GO TO PROG-END
        END-IF.
        DISPLAY 'Success TSCAdm-initServer'.
        MOVE 1 TO INIT-SERVER-FLAG.

    * 3. TSCデーモンへの接続
        SET MY-DOMAIN-NAME-PTR TO NULL.
        SET MY-TSCID-PTR TO NULL.
        MOVE 1 TO MY-DOMAIN-FLAG.
        CALL 'TSCDomain-NEW' USING
                BY VALUE MY-DOMAIN-NAME-PTR
                BY VALUE MY-TSCID-PTR
                BY VALUE MY-DOMAIN-FLAG
                BY REFERENCE CORBA-ENVIRONMENT
            RETURNING DOMAIN-PTR.
    * 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
            GO TO PROG-END
        END-IF.
        DISPLAY 'Success TSCDomain-NEW'.
        MOVE 1 TO DOMAIN-CREATE-FLAG.

    * TSCServerの取得
        CALL 'TSCAdm-getTSCServer' USING
                BY VALUE DOMAIN-PTR
                BY REFERENCE CORBA-ENVIRONMENT
            RETURNING SERVER-PTR.
    * 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
            GO TO PROG-END
        END-IF.
        DISPLAY 'Success TSCAdm-getTSCServer'.
        MOVE 1 TO GET-SERVER-FLAG.

    * 4. TSCユーザアクセプタの生成および各種設定
    * CBLClass_TSCfactの生成
        CALL 'CBLClass_TSCfact-NEW'
            RETURNING FACTORY-PTR.
        DISPLAY 'Success CBLClass_TSCfact-NEW'.
        MOVE 1 TO TSCFACT-CREATE-FLAG.

    * TSCAcceptorの生成
        SET ACCEPTOR-NAME-PTR TO NULL.
        CALL 'CBLClass_TSCacpt-NEW' USING
                BY VALUE FACTORY-PTR
```

```
                BY VALUE ACCEPTOR-NAME-PTR
                BY REFERENCE CORBA-ENVIRONMENT
            RETURNING ACCEPTOR-PTR.
      * 例外チェック
          IF NOT CORBA-NO-EXCEPTION THEN
              CALL 'OTM-EXCEPTION-HANDLER' USING
                      BY REFERENCE MAJOR
                      BY REFERENCE CORBA-ENVIRONMENT
              CALL 'TSCSysExcept-DELETE' USING
                      BY VALUE EXCEP OF CORBA-ENVIRONMENT
              MOVE 1 TO RETURN-CODE
              GO TO PROG-END
          END-IF.
          DISPLAY 'Success CBLClass_TSCacpt-NEW'.
          MOVE 1 TO TSCACPT-CREATE-FLAG.

      * 5. TSCルートアクセプタの生成および各種設定
      *    TSCRootAcceptorの生成
          SET THREAD-FACT-PTR TO NULL.
          CALL 'TSCRAcceptor-create' USING
                  BY VALUE SERVER-PTR
                  BY VALUE THREAD-FACT-PTR
                  BY REFERENCE CORBA-ENVIRONMENT
              RETURNING R-ACCEPTOR-PTR.
      * 例外チェック
          IF NOT CORBA-NO-EXCEPTION THEN
              CALL 'OTM-EXCEPTION-HANDLER' USING
                      BY REFERENCE MAJOR
                      BY REFERENCE CORBA-ENVIRONMENT
              CALL 'TSCSysExcept-DELETE' USING
                  BY VALUE EXCEP OF CORBA-ENVIRONMENT
              MOVE 1 TO RETURN-CODE
              GO TO PROG-END
          END-IF.
          DISPLAY 'Success TSCRAcceptor-create'.
          MOVE 1 TO RACPT-CREATE-FLAG.

      * TSCユーザアクセプタの登録
          CALL 'TSCRAcceptor-registerAcceptor' USING
                  BY VALUE R-ACCEPTOR-PTR
                  BY VALUE ACCEPTOR-PTR
                  BY REFERENCE CORBA-ENVIRONMENT
              RETURNING ACCEPTOR-ID.
      * 例外チェック
          IF NOT CORBA-NO-EXCEPTION THEN
              CALL 'OTM-EXCEPTION-HANDLER' USING
                      BY REFERENCE MAJOR
                      BY REFERENCE CORBA-ENVIRONMENT
              CALL 'TSCSysExcept-DELETE' USING
                  BY VALUE EXCEP OF CORBA-ENVIRONMENT
              MOVE 1 TO RETURN-CODE
              GO TO PROG-END
          END-IF.
          DISPLAY 'Success TSCRAcceptor-registerAcceptor'.

      * 6. TSCルートアクセプタの活性化
          MOVE 'serviceX' TO R-ACCEPTOR-NAME.
```

```
     CALL 'CORBA_string_set' USING
             BY REFERENCE R-ACCEPTOR-NAME-PTR
             BY REFERENCE R-ACCEPTOR-NAME-LEN
             BY REFERENCE R-ACCEPTOR-NAME.
     CALL 'TSCRAcceptor-activate' USING
             BY VALUE R-ACCEPTOR-PTR
             BY VALUE R-ACCEPTOR-NAME-PTR
             BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCRAcceptor-activate'.
     MOVE 1 TO RACPT-ACTIVATE-FLAG.

* 7. 実行制御の受け渡し
     DISPLAY 'Start TSCAdm-serverMainloop'.
     CALL 'TSCAdm-serverMainloop' USING
             BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCAdm-serverMainloop'.

 PROG-END.

* 8. TSCルートアクセプタの非活性化
     IF RACPT-ACTIVATE-FLAG = 1 THEN
         MOVE 0 TO DEACT-MODE
         CALL 'TSCRAcceptor-deactivate' USING
                 BY VALUE R-ACCEPTOR-PTR
                 BY VALUE DEACT-MODE
                 BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
         IF NOT CORBA-NO-EXCEPTION THEN
             CALL 'OTM-EXCEPTION-HANDLER' USING
                     BY REFERENCE MAJOR
                     BY REFERENCE CORBA-ENVIRONMENT
             CALL 'TSCSysExcept-DELETE' USING
                     BY VALUE EXCEP OF CORBA-ENVIRONMENT
             MOVE 1 TO RETURN-CODE
         ELSE
             DISPLAY 'Success TSCRAcceptor-deactivate'
         END-IF
```

```
        END-IF.

    * 9. TSCルートアクセプタの削除
        IF RACPT-CREATE-FLAG = 1 THEN
            CALL 'TSCRAcceptor-destroy' USING
                    BY VALUE R-ACCEPTOR-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
    * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                        BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            ELSE
                DISPLAY 'Success TSCRAcceptor-destroy'
            END-IF
        END-IF.

    * 10. TSCユーザオブジェクトファクトリおよび
    *     TSCユーザアクセプタの削除
        IF TSCACPT-CREATE-FLAG = 1 THEN
            CALL 'CBLClass_TSCacpt-DEL' USING
                    BY VALUE ACCEPTOR-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
    * 例外処理
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            END-IF
        END-IF.

    * 領域の解放
        IF TSCFACT-CREATE-FLAG = 1 THEN
            CALL 'CBLClass_TSCfact-DEL' USING
                    BY VALUE FACTORY-PTR
        END-IF.

    * 11. TSCデーモンへの接続解放
        IF GET-SERVER-FLAG = 1 THEN
            CALL 'TSCAdm-releaseTSCServer' USING
                    BY VALUE SERVER-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
    * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            ELSE
```

```
            DISPLAY 'Success TSCAdm-releaseTSCServer'
        END-IF
    END-IF.

    IF DOMAIN-CREATE-FLAG = 1 THEN
        CALL 'TSCDomain-DELETE' USING
                BY VALUE DOMAIN-PTR
                BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        END-IF
    END-IF.

* 12. TPBroker OTMの終了処理
    IF INIT-SERVER-FLAG = 1 THEN
        CALL 'TSCAdm-endServer' USING
                BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        ELSE
            DISPLAY 'Success TSCAdm-endServer'
        END-IF
    END-IF.

 END PROGRAM CORBA-SERVER-MAIN.
```

6.4 セション呼び出しをするアプリケーションプログラム（COBOL）

セション呼び出しをするアプリケーションプログラムの COBOL での作成例を示します。これらのサンプルコードは製品で提供されています。

ユーザ定義 IDL インタフェースの例，および IDL コンパイラが生成するクラスは，同期型呼び出しの場合と同様です。「6.2　同期型呼び出しをするアプリケーションプログラム（COBOL）」を参照してください。

● トランザクションフレームジェネレータが生成するクラス

OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。ただし，実際に生成されるのはクラスを使用するための副プログラムです。副プログラムの名称は次のとおりになります。

"クラス名"－"メソッド名"

表 6-3　セション呼び出しをするアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（COBOL）

分類	ファイル名	生成するクラス名（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	ABCfile_TSC_p.ocb	CBLClass_TSCspxy（TSC ユーザプロキシ）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	ABCfile_TSC_s.ocb	CBLClass _TSCsk（TSC ユーザスケルトン） CBLClass _TSCacpt（TSC ユーザアクセプタ）
雛形クラス	ABCfile_TSC_t.ocb	CBLClass_TSCimpl（TSC ユーザオブジェクト） CBLClass_TSCfact（TSC ユーザオブジェクトファクトリ） TSCCBLThread（TSC ユーザスレッド） TSCCBLThreadFactory（TSC ユーザスレッドファクトリ）

6.4.1 セション呼び出しをするクライアントアプリケーションの例（COBOL）

セション呼び出しをするクライアントアプリケーションの処理の流れとコードの例を示します。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

（1）サービス利用処理の流れ

1. COBOL adapter for TPBroker **の初期化処理**

2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
6. TSC **ユーザプロキシの削除**
7. TSC **デーモンへの接続解放**
8. TPBroker OTM **の終了処理**

（2）サービス利用処理のコード

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CLIENT.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.

       DATA DIVISION.
       WORKING-STORAGE SECTION.

       01 ORB-PTR                USAGE POINTER.
       01 DOMAIN-PTR             USAGE POINTER.
       01 CLIENT-PTR             USAGE POINTER.
       01 CLIENT-WAY             PIC S9(9) COMP.
       01 MY-DOMAIN-NAME         PIC X(10).
       01 MY-DOMAIN-NAME-PTR     USAGE POINTER.
       01 MY-DOMAIN-NAME-LEN     PIC S9(9) COMP.
       01 MY-TSCID               PIC X(10).
       01 MY-TSCID-PTR           USAGE POINTER.
       01 MY-TSCID-LEN           PIC S9(9) COMP.
       01 MY-DOMAIN-FLAG         PIC S9(9) COMP VALUE 1.
       01 ACCEPTOR-NAME          PIC X(20).
       01 ACCEPTOR-NAME-PTR      USAGE POINTER VALUE NULL.
       01 ACCEPTOR-NAME-LEN      PIC S9(9) COMP.
       01 SPROXY-PTR              USAGE POINTER.

       01 INIT-CLIENT-FLAG       PIC S9(9) COMP.
       01 DOMAIN-CREATE-FLAG     PIC S9(9) COMP.
       01 GET-CLIENT-FLAG        PIC S9(9) COMP.
       01 TSCSPRXY-START-FLAG    PIC S9(9) COMP.
       01 TSCSPRXY-CREATE-FLAG   PIC S9(9) COMP.

       01 CORBA-ENVIRONMENT.
        02 MAJOR                 PIC 9(9) COMP.
         88 CORBA-NO-EXCEPTION      VALUE 0.
         88 CORBA-USER-EXCEPTION    VALUE 1.
         88 CORBA-SYSTEM-EXCEPTION VALUE 2.
        02 EXCEP                 USAGE POINTER.
        02 FUNC-NAME             PIC X(256).

       01 ERR-CODE               PIC S9(9) COMP.

      * Sequence
```

```
 01 TYPE-CODE-PTR         USAGE POINTER.
 01 ELEMENT-NUMBER        PIC 9(9) COMP.
 01 SETOCTETVAL           PIC X.
 01 in_data               USAGE POINTER.
 01 out_data              USAGE POINTER.

 LINKAGE SECTION.
 01 ARGC                  PIC 9(9) COMP.
 01 ARGV                  USAGE POINTER.

 PROCEDURE DIVISION USING
        BY VALUE ARGC
        BY VALUE ARGV.

 MOVE 0 TO RETURN-CODE.
 MOVE 0 TO INIT-CLIENT-FLAG.
 MOVE 0 TO DOMAIN-CREATE-FLAG.
 MOVE 0 TO GET-CLIENT-FLAG.
 MOVE 0 TO TSCSPRXY-START-FLAG.
 MOVE 0 TO TSCSPRXY-CREATE-FLAG.

* ORBの初期化処理
 CALL 'CORBA-STUB-INIT-ABCfile'.

* 1. COBOL adapter for TPBrokerの初期化処理
 CALL 'CORBA_orb_init' USING
        BY REFERENCE ARGC
        BY REFERENCE ARGV
        BY REFERENCE CORBA-ENVIRONMENT
    RETURNING ORB-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
    CALL 'EXCEPTION-HANDLER' USING
            BY REFERENCE MAJOR
            BY REFERENCE CORBA-ENVIRONMENT
    CALL 'CORBA_FreeException' USING
            EXCEP OF CORBA-ENVIRONMENT
    MOVE 1 TO RETURN-CODE
    GO TO PROG-END
 END-IF.
 DISPLAY 'Success CORBA_orb_init'.

* 2. TPBroker OTMの初期化処理
 CALL 'TSCAdm-initClient' USING
        BY REFERENCE ARGC
        BY REFERENCE ARGV
        BY VALUE     ORB-PTR
        BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
    CALL 'OTM-EXCEPTION-HANDLER' USING
            BY REFERENCE MAJOR
            BY REFERENCE CORBA-ENVIRONMENT
    CALL 'TSCSysExcept-DELETE' USING
            BY VALUE EXCEP OF CORBA-ENVIRONMENT
    MOVE 1 TO RETURN-CODE
    GO TO PROG-END
 END-IF.
```

```
 DISPLAY 'Success TSCAdm-initClient'.
 MOVE 1 TO INIT-CLIENT-FLAG.

* 3. TSCデーモンへの接続
* (1) TSCDomainの生成
 SET MY-DOMAIN-NAME-PTR TO NULL.
 SET MY-TSCID-PTR TO NULL.
 MOVE 1 TO MY-DOMAIN-FLAG.
 CALL 'TSCDomain-NEW' USING
         BY VALUE     MY-DOMAIN-NAME-PTR
         BY VALUE     MY-TSCID-PTR
         BY VALUE     MY-DOMAIN-FLAG
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING DOMAIN-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success TSCDomain-NEW:DomainName = '
                                       MY-DOMAIN-NAME.
 MOVE 1 TO DOMAIN-CREATE-FLAG.

* (2) TSCClientの取得
 MOVE 1 TO CLIENT-WAY.
 CALL 'TSCAdm-getTSCClient' USING
         BY VALUE     DOMAIN-PTR
         BY VALUE     CLIENT-WAY
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING CLIENT-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success TSCAdm-getTSCClient'.
 MOVE 1 TO GET-CLIENT-FLAG.

* 4. TSCユーザプロキシの生成および各種設定
*    IDLインタフェース"CBLClass"用のTSCspxy生成
 SET ACCEPTOR-NAME-PTR TO NULL.
 CALL 'CBLClass_TSCspxy-NEW' USING
         BY VALUE     CLIENT-PTR
         BY VALUE     ACCEPTOR-NAME-PTR
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING SPROXY-PTR.
* 例外チェック
```

```
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success CBLClass_TSCspxy-NEW'.
 MOVE 1 TO TSCSPRXY-CREATE-FLAG.

* 5. TSCユーザプロキシのメソッド呼び出し
*     （サーバ側のオブジェクトの呼び出し）
*     in属性ユーザデータの確保と設定
*     Octet TypeCodeオブジェクトの生成
 CALL 'Create_CORBA_TypeCode' USING
         BY VALUE     10
         BY VALUE     1
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING TYPE-CODE-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

* Octet型Sequenceオブジェクトの生成
 MOVE 999999999 TO ELEMENT-NUMBER.
 CALL 'CORBA-SeqAlloc' USING
         BY REFERENCE ELEMENT-NUMBER
         BY REFERENCE TYPE-CODE-PTR
         BY REFERENCE in_data
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

 MOVE 1 TO ELEMENT-NUMBER.
 MOVE 'A' TO SETOCTETVAL.
 CALL 'CORBA-SeqSet' USING
         BY REFERENCE in_data
         BY REFERENCE ELEMENT-NUMBER
         BY REFERENCE SETOCTETVAL
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
```

```
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

* TypeCodeオブジェクトの解放
 CALL 'CORBA_TypeCode__release' USING
         BY VALUE TYPE-CODE-PTR
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

**************************************************
* セションの開始
**************************************************
 CALL 'TSCSProxy-TSCStart' USING
         BY VALUE     SPROXY-PTR
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 MOVE 1 TO TSCSPRXY-START-FLAG.

**************************************************
* サーバメソッドの呼び出し
**************************************************
 PERFORM 3 TIMES
     DISPLAY 'Start CBLClass-call'
     CALL 'CBLClass-call' USING
             BY VALUE     SPROXY-PTR
             BY VALUE     in_data
             BY REFERENCE out_data
             BY REFERENCE CORBA-ENVIRONMENT
     IF NOT CORBA-NO-EXCEPTION THEN
         EXIT PERFORM
     END-IF
 END-PERFORM.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
```

```
      CALL 'OTM-EXCEPTION-HANDLER' USING
              BY REFERENCE MAJOR
              BY REFERENCE CORBA-ENVIRONMENT
      CALL 'TSCSProxy-TSCStop' USING
              BY VALUE     SPROXY-PTR
              BY REFERENCE CORBA-ENVIRONMENT
      MOVE 0 TO TSCSPRXY-START-FLAG
      CALL 'TSCSysExcept-DELETE' USING
              BY VALUE EXCEP OF CORBA-ENVIRONMENT
      MOVE 1 TO RETURN-CODE
      GO TO PROG-END
  END-IF.
  DISPLAY 'Success CBLClass-call'.

  PROG-END.

************************************************************
* セションの停止
************************************************************
      IF TSCSPRXY-START-FLAG = 1 THEN
          CALL 'TSCSProxy-TSCStop' USING
                  BY VALUE     SPROXY-PTR
                  BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
          IF NOT CORBA-NO-EXCEPTION THEN
              CALL 'OTM-EXCEPTION-HANDLER' USING
                      BY REFERENCE MAJOR
                      BY REFERENCE CORBA-ENVIRONMENT
              CALL 'TSCSysExcept-DELETE' USING
                      BY VALUE EXCEP OF CORBA-ENVIRONMENT
              MOVE 1 TO RETURN-CODE
              GO TO PROG-END
          END-IF
      END-IF.

* 6. TSCユーザプロキシの削除
*    領域の解放
      IF TSCPRXY-CREATE-FLAG = 1 THEN
          CALL 'CBLClass_TSCprxy-DEL' USING
                  BY VALUE PROXY-PTR
                  BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
          IF NOT CORBA-NO-EXCEPTION THEN
              CALL 'OTM-EXCEPTION-HANDLER' USING
                      BY REFERENCE MAJOR
                      BY REFERENCE CORBA-ENVIRONMENT
              CALL 'TSCSysExcept-DELETE' USING
                      BY VALUE EXCEP OF CORBA-ENVIRONMENT
              MOVE 1 TO RETURN-CODE
          END-IF
      END-IF.

* 7. TSCデーモンへの接続解放
      IF GET-CLIENT-FLAG = 1 THEN
          CALL 'TSCAdm-releaseTSCClient' USING
                  BY VALUE CLIENT-PTR
                  BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
```

```
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                        BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            ELSE
                DISPLAY 'Success TSCAdm-releaseTSCClient'
            END-IF
        END-IF.

        IF DOMAIN-CREATE-FLAG = 1 THEN
            CALL 'TSCDomain-DELETE' USING
                    BY VALUE DOMAIN-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
    * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                        BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            END-IF
        END-IF.

    * 8. TPBroker OTMの終了処理
        IF INIT-CLIENT-FLAG = 1 THEN
            CALL 'TSCAdm-endClient' USING
                    BY REFERENCE CORBA-ENVIRONMENT
    * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                        BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            ELSE
                DISPLAY 'Success TSCAdm-endClient'
            END-IF
        END-IF.

     END PROGRAM CLIENT.
```

6.4.2 セション呼び出しをするサーバアプリケーションの例（COBOL）

同期型呼び出しの場合と同様です。「6.2.2　同期型呼び出しをするサーバアプリケーションの例（COBOL）」を参照してください。

6.5 TSCContext を利用するアプリケーションプログラム（COBOL）

TSCContext を利用するアプリケーションプログラムの COBOL での作成例を示します。これらのサンプルコードは製品で提供されています。

ユーザ定義 IDL インタフェースの例，IDL コンパイラが生成するクラス，およびトランザクションフレームジェネレータが生成するクラスは，同期型呼び出しの場合と同様です。「6.2　同期型呼び出しをするアプリケーションプログラム（COBOL）」を参照してください。

6.5.1 TSCContext を利用するクライアントアプリケーションの例（COBOL）

TSCContext を利用するクライアントアプリケーションの処理の流れとコードの例を示します。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

なお，TSCContext を利用するクライアントアプリケーションの例外処理は，同期型呼び出しの場合と同様です。「6.2.3　例外処理のコードの例（COBOL）」を参照してください。

（1） サービス利用処理の流れ

1. COBOL adapter for TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **コンテキストへのユーザデータの設定**
6. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
7. TSC **ユーザプロキシの削除**
8. TSC **デーモンへの接続解放**
9. TPBroker OTM **の終了処理**

（2） サービス利用処理のコード

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CLIENT.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
```

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01 ORB-PTR                USAGE POINTER.
 01 DOMAIN-PTR             USAGE POINTER.
 01 CLIENT-PTR             USAGE POINTER.
 01 CLIENT-WAY             PIC S9(9) COMP.
 01 MY-DOMAIN-NAME         PIC X(10).
 01 MY-DOMAIN-NAME-PTR     USAGE POINTER.
 01 MY-DOMAIN-NAME-LEN     PIC S9(9) COMP.
 01 MY-TSCID               PIC X(10).
 01 MY-TSCID-PTR           USAGE POINTER.
 01 MY-TSCID-LEN           PIC S9(9) COMP.
 01 MY-DOMAIN-FLAG         PIC S9(9) COMP VALUE 1.
 01 ACCEPTOR-NAME          PIC X(20).
 01 ACCEPTOR-NAME-PTR      USAGE POINTER VALUE NULL.
 01 ACCEPTOR-NAME-LEN      PIC S9(9) COMP.
 01 PROXY-PTR              USAGE POINTER.

 01 CONTEXT-PTR            USAGE POINTER.
 01 CONTEXT-DATA-PTR       USAGE POINTER.
 01 CONTEXT-DATA-LEN       PIC S9(9) COMP.
 01 CONTEXT-DATA           PIC X(32).

 01 INIT-CLIENT-FLAG       PIC S9(9) COMP.
 01 DOMAIN-CREATE-FLAG     PIC S9(9) COMP.
 01 GET-CLIENT-FLAG        PIC S9(9) COMP.
 01 TSCPRXY-CREATE-FLAG    PIC S9(9) COMP.

 01 CORBA-ENVIRONMENT.
  02 MAJOR                PIC 9(9) COMP.
   88 CORBA-NO-EXCEPTION     VALUE 0.
   88 CORBA-USER-EXCEPTION   VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION VALUE 2.
  02 EXCEP                 USAGE POINTER.
  02 FUNC-NAME             PIC X(256).

 01 ERR-CODE               PIC S9(9) COMP.

*Sequence
 01 TYPE-CODE-PTR          USAGE POINTER.
 01 ELEMENT-NUMBER         PIC 9(9) COMP.
 01 SETOCTETVAL            PIC X.
 01 in_data                USAGE POINTER.
 01 out_data               USAGE POINTER.

 LINKAGE SECTION.
 01 ARGC                   PIC 9(9) COMP.
 01 ARGV                   USAGE POINTER.

 PROCEDURE DIVISION USING
         BY VALUE ARGC
         BY VALUE ARGV.

 MOVE 0 TO RETURN-CODE.
 MOVE 0 TO INIT-CLIENT-FLAG.
 MOVE 0 TO DOMAIN-CREATE-FLAG.
```

```
 MOVE 0 TO GET-CLIENT-FLAG.
 MOVE 0 TO TSCPRXY-CREATE-FLAG.

* ORBの初期化処理
 CALL 'CORBA-STUB-INIT-ABCfile'.

* 1. COBOL adapter for TPBrokerの初期化処理
 CALL 'CORBA_orb_init' USING
         BY REFERENCE ARGC
         BY REFERENCE ARGV
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING ORB-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success CORBA_orb_init'.

* 2. TPBroker OTMの初期化処理
 CALL 'TSCAdm-initClient' USING
         BY REFERENCE ARGC
         BY REFERENCE ARGV
         BY VALUE     ORB-PTR
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success TSCAdm-initClient'.
 MOVE 1 TO INIT-CLIENT-FLAG.

* 3. TSCデーモンへの接続
* (1) TSCDomainの生成
 SET MY-DOMAIN-NAME-PTR TO NULL.
 SET MY-TSCID-PTR TO NULL.
 MOVE 1 TO MY-DOMAIN-FLAG.
 CALL 'TSCDomain-NEW' USING
         BY VALUE     MY-DOMAIN-NAME-PTR
         BY VALUE     MY-TSCID-PTR
         BY VALUE     MY-DOMAIN-FLAG
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING DOMAIN-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
```

```
                BY REFERENCE MAJOR
                BY REFERENCE CORBA-ENVIRONMENT
        CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
        MOVE 1 TO RETURN-CODE
        GO TO PROG-END
 END-IF.
 DISPLAY 'Success TSCDomain-NEW:DomainName = '
                                        MY-DOMAIN-NAME.
 MOVE 1 TO DOMAIN-CREATE-FLAG.

* (2) TSCClientの取得
 MOVE 1 TO CLIENT-WAY.
 CALL 'TSCAdm-getTSCClient' USING
          BY VALUE     DOMAIN-PTR
          BY VALUE     CLIENT-WAY
          BY REFERENCE CORBA-ENVIRONMENT
        RETURNING CLIENT-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
        CALL 'OTM-EXCEPTION-HANDLER' USING
                BY REFERENCE MAJOR
                BY REFERENCE CORBA-ENVIRONMENT
        CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
        MOVE 1 TO RETURN-CODE
        GO TO PROG-END
 END-IF.
 DISPLAY 'Success TSCAdm-getTSCClient'.
 MOVE 1 TO GET-CLIENT-FLAG.

* 4. TSCユーザプロキシの生成および各種設定
*    IDLインタフェース"CBLClass"用のTSCProxy生成
 SET ACCEPTOR-NAME-PTR TO NULL.
 CALL 'CBLClass_TSCprxy-NEW' USING
          BY VALUE     CLIENT-PTR
          BY VALUE     ACCEPTOR-NAME-PTR
          BY REFERENCE CORBA-ENVIRONMENT
        RETURNING PROXY-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
        CALL 'OTM-EXCEPTION-HANDLER' USING
                BY REFERENCE MAJOR
                BY REFERENCE CORBA-ENVIRONMENT
        CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
        MOVE 1 TO RETURN-CODE
        GO TO PROG-END
 END-IF.
 DISPLAY 'Success CBLClass_TSCprxy-NEW'.
 MOVE 1 TO TSCPRXY-CREATE-FLAG.

* 5. TSCコンテキストへのユーザデータの設定
 CALL 'TSCProxyObject-TSCContextGet' USING
          BY VALUE PROXY-PTR
          BY REFERENCE CORBA-ENVIRONMENT
        RETURNING CONTEXT-PTR.
```

```
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 MOVE 'UserID:1111' TO CONTEXT-DATA.
 MOVE 32 TO CONTEXT-DATA-LEN.
 CALL 'CORBA_string_set' USING
         BY REFERENCE CONTEXT-DATA-PTR
         BY REFERENCE CONTEXT-DATA-LEN
         BY REFERENCE CONTEXT-DATA.
 CALL 'TSCContext_setUserData' USING
         BY VALUE CONTEXT-PTR
         BY VALUE CONTEXT-DATA-PTR
         BY VALUE CONTEXT-DATA-LEN
         BY REFERENCE CORBA-ENVIRONMENT.
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

* 6. TSCユーザプロキシのメソッド呼び出し
*     （サーバ側のオブジェクトの呼び出し）
*    in属性ユーザデータの確保と設定
*    Octet TypeCodeオブジェクトの生成
 CALL 'Create_CORBA_TypeCode' USING
         BY VALUE     10
         BY VALUE     1
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING TYPE-CODE-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

* Octet型Sequenceオブジェクトの生成
 MOVE 999999999 TO ELEMENT-NUMBER.
 CALL 'CORBA-SeqAlloc' USING
         BY REFERENCE ELEMENT-NUMBER
         BY REFERENCE TYPE-CODE-PTR
         BY REFERENCE in_data
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
```

```
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

 MOVE 1 TO ELEMENT-NUMBER.
 MOVE 'A' TO SETOCTETVAL.
 CALL 'CORBA-SeqSet' USING
         BY REFERENCE in_data
         BY REFERENCE ELEMENT-NUMBER
         BY REFERENCE SETOCTETVAL
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

* TypeCodeオブジェクトの解放
 CALL 'CORBA_TypeCode__release' USING
         BY VALUE TYPE-CODE-PTR
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.

************************************************
* サーバメソッドの呼び出し
************************************************
 DISPLAY 'Start CBLClass-call'.
 CALL 'CBLClass-call' USING
         BY VALUE     PROXY-PTR
         BY VALUE     in_data
         BY REFERENCE out_data
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
```

```
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
        MOVE 1 TO RETURN-CODE
        GO TO PROG-END
    END-IF.
    DISPLAY 'Success CBLClass-call'.

    PROG-END.

   * 7. TSCユーザプロキシの削除
   *    領域の解放
        IF TSCPRXY-CREATE-FLAG = 1 THEN
            CALL 'CBLClass_TSCprxy-DEL' USING
                    BY VALUE PROXY-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
   * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                        BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            END-IF
        END-IF.

   * 8. TSCデーモンへの接続解放
        IF GET-CLIENT-FLAG = 1 THEN
            CALL 'TSCAdm-releaseTSCClient' USING
                    BY VALUE CLIENT-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
   * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                        BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            ELSE
                DISPLAY 'Success TSCAdm-releaseTSCClient'
            END-IF
        END-IF.

        IF DOMAIN-CREATE-FLAG = 1 THEN
            CALL 'TSCDomain-DELETE' USING
                    BY VALUE DOMAIN-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
   * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                        BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            END-IF
        END-IF.
```

```
* 9. TPBroker OTMの終了処理
     IF INIT-CLIENT-FLAG = 1 THEN
         CALL 'TSCAdm-endClient' USING
                 BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
         IF NOT CORBA-NO-EXCEPTION THEN
             CALL 'OTM-EXCEPTION-HANDLER' USING
                     BY REFERENCE MAJOR
                     BY REFERENCE CORBA-ENVIRONMENT
             CALL 'TSCSysExcept-DELETE' USING
                     BY VALUE EXCEP OF CORBA-ENVIRONMENT
             MOVE 1 TO RETURN-CODE
         ELSE
             DISPLAY 'Success TSCAdm-endClient'
         END-IF
     END-IF.

 END PROGRAM CLIENT.
```

6.5.2 TSCContext を利用するサーバアプリケーションの例（COBOL）

TSCContext を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス CBLClass_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス CBLClass_TSCfact に TSC ユーザオブジェクトファクトリのコードを記述します。

また，TSCContext を使用する場合は，ユーザメソッドの第 1 引数に TSC ユーザオブジェクトのポインタを受け取る形式 2 の雛形ソースが必要です。形式 2 の雛形ソースを出力するには，tscidl2cbl コマンドの -format オプションに "2" を指定してください。形式 1 および形式 2 は，TSC ユーザオブジェクトに対してユーザが実装する副プログラムの形式です。詳細は，7 章の「ABC_TSCfactimpl（COBOL）」を参照してください。

TSCContext を利用するサーバアプリケーションの例外処理は，同期型呼び出しの場合と同様です。「6.2.3　例外処理のコードの例（COBOL）」を参照してください。

（1）TSC ユーザオブジェクト（CBLClass_TSCimpl）と TSC ユーザオブジェクトファクトリ（CBLClass_TSCfact）のコード

```
*************************************************
* Operation 'call'
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'call'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
```

```
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 CORBA-ENVIRONMENT.
  02 MAJOR                    PIC 9(9) COMP.
   88 CORBA-NO-EXCEPTION          VALUE 0.
   88 CORBA-USER-EXCEPTION        VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION      VALUE 2.
  02 EXCEP                    USAGE POINTER.
  02 FUNC-NAME                PIC X(256).

 01 TSC-SEQMAXLEN PIC 9(9) USAGE COMP.
 01 TSC-TC-1 USAGE POINTER.
 01 TSC-SEQENV.
   02 MAJOR PIC 9(9) USAGE COMP.
   02 EXCEP USAGE POINTER.
   02 FUNC-NAME PIC X(256).
 01 TSC-TC-2 USAGE POINTER.

* 必要に応じてデータ宣言を追加できます。
 01 ERR-CODE                  PIC S9(9) COMP.
 01 TYPE-CODE-PTR             USAGE POINTER.
 01 MY-OUT-DATA-LEN           PIC S9(9) COMP.
 01 ELEMENT-NUMBER            PIC S9(9) COMP.
 01 SETOCTETVAL               PIC X.

 01 CONTEXT-PTR         USAGE POINTER.
 01 CONTEXT-DATA-PTR    USAGE POINTER.
 01 CONTEXT-DATA-LEN    PIC S9(9) COMP.
 01 CONTEXT-DATA        PIC X(32).

 LINKAGE SECTION.
 01 TSC-OBJECT-PTR USAGE POINTER.
 01 in_data USAGE POINTER.
 01 out_data USAGE POINTER.

* Do not change signature of this sub-program.
 PROCEDURE DIVISION
     USING
         BY VALUE TSC-OBJECT-PTR
         BY VALUE in_data
         BY REFERENCE out_data.

* Write user own code.
* ユーザメソッドのコードを記述します。
 DISPLAY 'call method in CBLClass'.

 CALL 'TSCObject-TSCContextGet' USING
         BY VALUE TSC-OBJECT-PTR
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING CONTEXT-PTR.
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
```

```
    EXIT PROGRAM
END-IF.
CALL 'TSCContext-getUserData' USING
        BY VALUE CONTEXT-PTR
        BY REFERENCE CORBA-ENVIRONMENT
    RETURNING CONTEXT-DATA-PTR.
IF NOT CORBA-NO-EXCEPTION THEN
    CALL 'OTM-EXCEPTION-HANDLER' USING
            BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
            BY REFERENCE CORBA-ENVIRONMENT
    CALL 'TSCSysExcept-DELETE' USING
            BY VALUE EXCEP OF CORBA-ENVIRONMENT
    EXIT PROGRAM
END-IF.
CALL 'TSCContext-getUserDataLength' USING
        BY VALUE CONTEXT-PTR
        BY REFERENCE CORBA-ENVIRONMENT
    RETURNING CONTEXT-DATA-LEN.
IF NOT CORBA-NO-EXCEPTION THEN
    CALL 'OTM-EXCEPTION-HANDLER' USING
            BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
            BY REFERENCE CORBA-ENVIRONMENT
    CALL 'TSCSysExcept-DELETE' USING
            BY VALUE EXCEP OF CORBA-ENVIRONMENT
    EXIT PROGRAM
END-IF.
IF (CONTEXT-DATA-LEN IS > 32) THEN
    MOVE 32 TO CONTEXT-DATA-LEN
END-IF.
CALL 'CORBA_string_get' USING
        BY REFERENCE CONTEXT-DATA-PTR
        BY REFERENCE CONTEXT-DATA-LEN
        BY REFERENCE CONTEXT-DATA.
DISPLAY 'Context-data = ' CONTEXT-DATA.

* OUT属性引数の作成
* Octet TypeCode オブジェクトの作成
 CALL 'Create_CORBA_TypeCode' USING
        BY VALUE     10
        BY VALUE     1
        BY REFERENCE CORBA-ENVIRONMENT
        RETURNING TYPE-CODE-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
    CALL 'EXCEPTION-HANDLER' USING
            BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
            BY REFERENCE CORBA-ENVIRONMENT
    EXIT PROGRAM
 END-IF.

* Octet型Sequenceオブジェクトの生成
 MOVE 999999999 TO MY-OUT-DATA-LEN.
 CALL 'CORBA-SeqAlloc' USING
        BY REFERENCE MY-OUT-DATA-LEN
        BY REFERENCE TYPE-CODE-PTR
        BY REFERENCE out_data
        BY REFERENCE CORBA-ENVIRONMENT.
```

```
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
             BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

 MOVE 1 TO ELEMENT-NUMBER.
 MOVE 'A' TO SETOCTETVAL.
 CALL 'CORBA-SeqSet' USING
         BY REFERENCE in_data
         BY REFERENCE ELEMENT-NUMBER
         BY REFERENCE SETOCTETVAL
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
             BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

 CALL 'CORBA_TypeCode__release' USING
     BY VALUE TYPE-CODE-PTR
     BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
             BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

 END PROGRAM 'call'.


**************************************************
* Constructor of 'CBLClass_TSCimpl'
**************************************************
* Constructor of OTM Object
* Implement.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING SKELETON-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
```

```
* コンストラクタの引数の数および型を変更することもできます。

* This sub-program must return a pointer
* that 'CBLClass_TSCsk-NEW' sub-program returns.
     CALL 'CBLClass_TSCsk-NEW'
          RETURNING SKELETON-POINTER.
 END PROGRAM 'CBLClass_TSCimpl-NEW'.

*************************************************
* Destructor of 'CBLClass_TSCimpl'
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION USING
          BY VALUE SKELETON-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。

* This sub-program must call
* 'CBLClass_TSCsk-DEL' sub-program.
     CALL 'CBLClass_TSCsk-DEL' USING
          BY VALUE SKELETON-POINTER.
 END PROGRAM 'CBLClass_TSCimpl-DEL'.

*************************************************
* Constructor of CBLClass_TSCfact
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
* 引数の数および型を変更することもできます。

* This sub-program must return a pointer that
* 'CBLClass_TSCfact-get' sub-program returns.
     CALL 'CBLClass_TSCfact-get'
          RETURNING FACTORY-POINTER.
```

```
 END PROGRAM 'CBLClass_TSCfact-NEW'.


*****************************************************
* Destructor of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。

* This sub-program must call
* 'CBLClass_TSCfact-rls'.sub-program.
     CALL 'CBLClass_TSCfact-rls' USING
         BY VALUE FACTORY-POINTER.
 END PROGRAM 'CBLClass_TSCfact-DEL'.


*****************************************************
* 'create' method of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-crt'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
* Do not change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER
     RETURNING OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを生成するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-NEW' returns.
     CALL 'CBLClass_TSCimpl-NEW'
         RETURNING OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-crt'.
```

```
*************************************************
* 'destroy' method of CBLClass_TSCfact
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-dst'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
* Do not change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER
         BY VALUE OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを削除するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-DEL' returns.
     CALL 'CBLClass_TSCimpl-DEL' USING
         BY VALUE OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-dst'.


*************************************************
* TSCCBLThread-beginThread of
*     TSCCBLThread
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-beginThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 BEGIN-THREAD-PTR USAGE POINTER.
 01 END-THREAD-PTR USAGE POINTER.
* Do not change signature of this sub-program.
 LINKAGE SECTION.
 01 THREAD-FACTORY-ID PIC S9(9) COMP.
 PROCEDURE DIVISION USING
         BY VALUE THREAD-FACTORY-ID.

* Write user own code.
* スレッド開始処理を記述します。
* 必要に応じて変更してください。

 END PROGRAM 'TSCCBLThread-beginThread'.


*************************************************
* TSCCBLThread-endThread of
```

```
*     TSCCBLThreadFactory
**************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-endThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* Do not change signature of this sub-program.
 LINKAGE SECTION.
 01 THREAD-FACTORY-ID PIC S9(9) COMP.
 PROCEDURE DIVISION USING
         BY VALUE THREAD-FACTORY-ID.

* Write user own code.
* スレッド終了処理を記述します。
* 必要に応じて変更してください。

 END PROGRAM 'TSCCBLThread-endThread'.
```

(2) サービス登録処理の流れ・コード

同期型呼び出しの場合と同様です。「6.2.2(2)　サービス登録処理の流れ」,「6.2.2(3)　サービス登録処理のコード」を参照してください。

6.6　TSCThread を利用するアプリケーションプログラム（COBOL）

TSCThread を利用するアプリケーションプログラムの COBOL での作成例を示します。これらのサンプルコードは製品で提供されています。

ユーザ定義 IDL インタフェースの例，IDL コンパイラが生成するクラス，およびトランザクションフレームジェネレータが生成するクラスは，同期型呼び出しの場合と同様です。「6.2　同期型呼び出しをするアプリケーションプログラム（COBOL）」を参照してください。

6.6.1　TSCThread を利用するクライアントアプリケーションの例（COBOL）

同期型呼び出しの場合と同様です。詳細は「6.2.1　同期型呼び出しをするクライアントアプリケーションの例（COBOL）」を参照してください。例外処理については，「6.2.3　例外処理のコードの例（COBOL）」を参照してください。

6.6.2　TSCThread を利用するサーバアプリケーションの例（COBOL）

TSCThread を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス CBLClass_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス CBLClass_TSCfact に TSC ユーザオブジェクトファクトリのコードを記述します。

また，TSCThread を使用する場合は，ユーザメソッドの第 1 引数に TSC ユーザオブジェクトのポインタを受け取る形式 2 の雛形ソースが必要です。形式 2 の雛形ソースを出力するには，tscidl2cbl コマンドの -format オプションに "2" を指定してください。形式 1 および形式 2 は，TSC ユーザオブジェクトに対してユーザが実装する副プログラムの形式です。詳細は，7 章の「ABC_TSCfactimpl（COBOL）」を参照してください。

TSCThread を利用するサーバアプリケーションの例外処理は，同期型呼び出しの場合と同様です。「6.2.3　例外処理のコードの例（COBOL）」を参照してください。

（1）TSC ユーザオブジェクト（CBLClass_TSCimpl）と TSC ユーザオブジェクトファクトリ（CBLClass_TSCfact）のコード

```
*****************************************************
* Operation 'call'
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'call'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 CORBA-ENVIRONMENT.
  02 MAJOR                    PIC 9(9) COMP.
   88 CORBA-NO-EXCEPTION          VALUE 0.
   88 CORBA-USER-EXCEPTION        VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION      VALUE 2.
  02 EXCEP                    USAGE POINTER.
  02 FUNC-NAME                PIC X(256).

 01 TSC-SEQMAXLEN PIC 9(9) USAGE COMP.
 01 TSC-TC-1 USAGE POINTER.
 01 TSC-SEQENV.
   02 MAJOR PIC 9(9) USAGE COMP.
   02 EXCEP USAGE POINTER.
   02 FUNC-NAME PIC X(256).
 01 TSC-TC-2 USAGE POINTER.

* 必要に応じてデータ宣言を追加できます。
 01 ERR-CODE                  PIC S9(9) COMP.
 01 TYPE-CODE-PTR             USAGE POINTER.
 01 MY-OUT-DATA-LEN           PIC S9(9) COMP.
 01 ELEMENT-NUMBER            PIC S9(9) COMP.
 01 SETOCTETVAL               PIC X.

 01 THREAD-PTR          USAGE POINTER.
 01 THREAD-FACT-ID      PIC S9(9) COMP.

 LINKAGE SECTION.
 01 TSC-OBJECT-PTR USAGE POINTER.
 01 in_data USAGE POINTER.
 01 out_data USAGE POINTER.

* Do not change signature of this sub-program.
 PROCEDURE DIVISION
     USING
         BY VALUE TSC-OBJECT-PTR
         BY VALUE in_data
         BY REFERENCE out_data.

* Write user own code.
* ユーザメソッドのコードを記述します。
 DISPLAY 'call method in CBLClass'.

* TSCThreadの取得
 CALL 'TSCObject-TSCThreadGet' USING
         BY VALUE TSC-OBJECT-PTR
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING THREAD-PTR.
```

```
* Exception process
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
              BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
              BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
              BY VALUE EXCEP OF CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.
* Get ThreadFactoryID
 CALL 'TSCCBLThread-getThreadFactID' USING
         BY VALUE THREAD-PTR
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING THREAD-FACT-ID.
* Exception process
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
              BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
              BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
              BY VALUE EXCEP OF CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.
 DISPLAY 'Thread-Fact-ID = ' THREAD-FACT-ID.

* OUT属性引数の作成
* Octet TypeCode オブジェクトの作成
 CALL 'Create_CORBA_TypeCode' USING
         BY VALUE     10
         BY VALUE     1
         BY REFERENCE CORBA-ENVIRONMENT
         RETURNING TYPE-CODE-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
              BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
              BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

* Octet型Sequenceオブジェクトの生成
 MOVE 999999999 TO MY-OUT-DATA-LEN.
 CALL 'CORBA-SeqAlloc' USING
         BY REFERENCE MY-OUT-DATA-LEN
         BY REFERENCE TYPE-CODE-PTR
         BY REFERENCE out_data
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
              BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
              BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

 MOVE 1 TO ELEMENT-NUMBER.
 MOVE 'A' TO SETOCTETVAL.
```

```
 CALL 'CORBA-SeqSet' USING
         BY REFERENCE in_data
         BY REFERENCE ELEMENT-NUMBER
         BY REFERENCE SETOCTETVAL
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
             BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

 CALL 'CORBA_TypeCode__release' USING
     BY VALUE TYPE-CODE-PTR
     BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
             BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

 END PROGRAM 'call'.


**************************************************
* Constructor of 'CBLClass_TSCimpl'
**************************************************
* Constructor of OTM Object
* Implement.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING SKELETON-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
* コンストラクタの引数の数および型を変更することもできます。

* This sub-program must return a pointer
* that 'CBLClass_TSCsk-NEW' sub-program returns.
     CALL 'CBLClass_TSCsk-NEW'
         RETURNING SKELETON-POINTER.
 END PROGRAM 'CBLClass_TSCimpl-NEW'.

**************************************************
* Destructor of 'CBLClass_TSCimpl'
```

```
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE SKELETON-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。

* This sub-program must call
* 'CBLClass_TSCsk-DEL' sub-program.
     CALL 'CBLClass_TSCsk-DEL' USING
         BY VALUE SKELETON-POINTER.
 END PROGRAM 'CBLClass_TSCimpl-DEL'.

*****************************************************
* Constructor of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
* 引数の数および型を変更することもできます。

* This sub-program must return a pointer that
* 'CBLClass_TSCfact-get' sub-program returns.
     CALL 'CBLClass_TSCfact-get'
         RETURNING FACTORY-POINTER.
 END PROGRAM 'CBLClass_TSCfact-NEW'.


*****************************************************
* Destructor of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
```

```
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。

* This sub-program must call
* 'CBLClass_TSCfact-rls'.sub-program.
      CALL 'CBLClass_TSCfact-rls' USING
          BY VALUE FACTORY-POINTER.
 END PROGRAM 'CBLClass_TSCfact-DEL'.


*****************************************************
* 'create' method of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-crt'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
* Do not change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER
      RETURNING OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを生成するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-NEW' returns.
      CALL 'CBLClass_TSCimpl-NEW'
          RETURNING OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-crt'.


*****************************************************
* 'destroy' method of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-dst'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
```

```
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
* Do not change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER
         BY VALUE OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを削除するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-DEL' returns.
     CALL 'CBLClass_TSCimpl-DEL' USING
         BY VALUE OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-dst'.


*****************************************************
* TSCCBLThread-beginThread of
*     TSCCBLThread
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-beginThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 BEGIN-THREAD-PTR USAGE POINTER.
 01 END-THREAD-PTR USAGE POINTER.
* Do not change signature of this sub-program.
 LINKAGE SECTION.
 01 THREAD-FACTORY-ID PIC S9(9) COMP.
 PROCEDURE DIVISION USING
         BY VALUE THREAD-FACTORY-ID.

* Write user own code.
* スレッド開始処理を記述します。
* 必要に応じて変更してください。

 END PROGRAM 'TSCCBLThread-beginThread'.


*****************************************************
* TSCCBLThread-endThread of
*     TSCCBLThreadFactory
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-endThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* Do not change signature of this sub-program.
```

```
       LINKAGE SECTION.
       01 THREAD-FACTORY-ID PIC S9(9) COMP.
       PROCEDURE DIVISION USING
               BY VALUE THREAD-FACTORY-ID.

      * Write user own code.
      * スレッド終了処理を記述します。
      * 必要に応じて変更してください。

       END PROGRAM 'TSCCBLThread-endThread'.
```

(2) サービス登録処理の流れ

1. COBOL adapter for TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザアクセプタの生成および各種設定**
5. TSC **ユーザスレッドファクトリの生成および各種設定**
6. TSC **ルートアクセプタの生成および各種設定**
7. TSC **ルートアクセプタの活性化**
8. **実行制御の受け渡し**
9. TSC **ルートアクセプタの非活性化**
10. TSC **ルートアクセプタの削除**
11. TSC **ユーザオブジェクトファクトリおよび** TSC **ユーザアクセプタの削除**
12. TSC **デーモンへの接続解放**
13. TPBroker OTM **の終了処理**

(3) サービス登録処理のコード

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CORBA-SERVER-MAIN.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       REPOSITORY.
       CLASS CBLClass IS 'ABCfile_s'.
       DATA DIVISION.
      * The following are the ORB pointers.
      * These are necessary for all servers.
       WORKING-STORAGE SECTION.
       01 ORB-PTR               USAGE POINTER.

       01 DOMAIN-PTR            USAGE POINTER.
       01 SERVER-PTR            USAGE POINTER.
       01 FACTORY-PTR           USAGE POINTER.
       01 ACCEPTOR-PTR          USAGE POINTER.
```

```
 01 R-ACCEPTOR-PTR         USAGE POINTER.
 01 THREAD-FACT-PTR        USAGE POINTER.
 01 THREAD-FACT-ID         PIC S9(9) COMP.
 01 MY-DOMAIN-NAME         PIC X(10).
 01 MY-DOMAIN-NAME-PTR     USAGE POINTER.
 01 MY-DOMAIN-NAME-LEN     PIC S9(9) COMP.
 01 MY-TSCID               PIC X(10).
 01 MY-TSCID-PTR           USAGE POINTER.
 01 MY-TSCID-LEN           PIC S9(9) COMP.
 01 MY-DOMAIN-FLAG         PIC S9(9) COMP.
 01 ACCEPTOR-NAME-PTR      USAGE POINTER.
 01 ACCEPTOR-ID            PIC S9(9) COMP.
 01 P-COUNT                PIC S9(9) COMP.
 01 DEACT-MODE             PIC S9(9) COMP.
 01 R-ACCEPTOR-NAME        PIC X(10).
 01 R-ACCEPTOR-NAME-PTR    USAGE POINTER.
 01 R-ACCEPTOR-NAME-LEN    PIC S9(9) COMP VALUE 30.

 01 INIT-SERVER-FLAG       PIC S9(9) COMP.
 01 DOMAIN-CREATE-FLAG     PIC S9(9) COMP.
 01 GET-SERVER-FLAG        PIC S9(9) COMP.
 01 TSCFACT-CREATE-FLAG    PIC S9(9) COMP.
 01 TSCACPT-CREATE-FLAG    PIC S9(9) COMP.
 01 THREAD-CREATE-FLAG     PIC S9(9) COMP.
 01 RACPT-CREATE-FLAG      PIC S9(9) COMP.
 01 RACPT-ACTIVATE-FLAG    PIC S9(9) COMP.

 01 CORBA-ENVIRONMENT.
  02 MAJOR PIC 9(9) USAGE COMP.
   88 CORBA-NO-EXCEPTION     VALUE 0.
   88 CORBA-USER-EXCEPTION   VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION VALUE 2.
  02 EXCEP USAGE POINTER.
  02 FUNC-NAME PIC X(256).

 LINKAGE SECTION.
 01 ARGC PIC S9(9) USAGE COMP.
 01 ARGV USAGE POINTER.

 PROCEDURE DIVISION USING
         BY VALUE ARGC
         BY VALUE ARGV.

     MOVE 0 TO RETURN-CODE.
     MOVE 0 TO INIT-SERVER-FLAG.
     MOVE 0 TO DOMAIN-CREATE-FLAG.
     MOVE 0 TO GET-SERVER-FLAG.
     MOVE 0 TO TSCFACT-CREATE-FLAG.
     MOVE 0 TO TSCACPT-CREATE-FLAG.
     MOVE 0 TO THREAD-CREATE-FLAG.
     MOVE 0 TO RACPT-CREATE-FLAG.
     MOVE 0 TO RACPT-ACTIVATE-FLAG.

* 1. COBOL adapter for TPBrokerの初期化処理
* First, call the skeleton and class initializers.
     CALL 'CORBA-SKEL-INIT-ABCfile'.
     CALL 'CBLClass-CLASS-INIT' USING BY VALUE CBLClass.
```

```
* ORBの初期化
     CALL 'CORBA_orb_init' USING
         BY REFERENCE ARGC
         BY REFERENCE ARGV
         BY REFERENCE CORBA-ENVIRONMENT
         RETURNING    ORB-PTR.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'CORBA_FreeException' USING
                 EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success CORBA_orb_init'.

* 2. TPBroker OTMの初期化処理
     CALL 'TSCAdm-initServer' USING
             BY REFERENCE ARGC
             BY REFERENCE ARGV
             BY VALUE      ORB-PTR
             BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCAdm-initServer'.
     MOVE 1 TO INIT-SERVER-FLAG.

* 3. TSCデーモンへの接続
     SET MY-DOMAIN-NAME-PTR TO NULL.
     SET MY-TSCID-PTR TO NULL.
     MOVE 1 TO MY-DOMAIN-FLAG.
     CALL 'TSCDomain-NEW' USING
             BY VALUE MY-DOMAIN-NAME-PTR
             BY VALUE MY-TSCID-PTR
             BY VALUE MY-DOMAIN-FLAG
             BY REFERENCE CORBA-ENVIRONMENT
         RETURNING DOMAIN-PTR.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
                 BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
```

```
     DISPLAY 'Success TSCDomain-NEW'.
     MOVE 1 TO DOMAIN-CREATE-FLAG.

* TSCServerの取得
     CALL 'TSCAdm-getTSCServer' USING
             BY VALUE DOMAIN-PTR
             BY REFERENCE CORBA-ENVIRONMENT
         RETURNING SERVER-PTR.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
                 BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCAdm-getTSCServer'.
     MOVE 1 TO GET-SERVER-FLAG.

* 4. TSCユーザアクセプタの生成および各種設定
* CBLClass_TSCfactの生成
     CALL 'CBLClass_TSCfact-NEW'
         RETURNING FACTORY-PTR.
     DISPLAY 'Success CBLClass_TSCfact-NEW'.
     MOVE 1 TO TSCFACT-CREATE-FLAG.

* TSCAcceptorの生成
     SET ACCEPTOR-NAME-PTR TO NULL.
     CALL 'CBLClass_TSCacpt-NEW' USING
             BY VALUE FACTORY-PTR
             BY VALUE ACCEPTOR-NAME-PTR
             BY REFERENCE CORBA-ENVIRONMENT
         RETURNING ACCEPTOR-PTR.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
                 BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success CBLClass_TSCacpt-NEW'.
     MOVE 1 TO TSCACPT-CREATE-FLAG.

* 5. TSCユーザスレッドファクトリの生成および各種設定
* TSCThreadFactoryの生成
     MOVE 123 TO THREAD-FACT-ID.
     CALL 'TSCCBLThreadFactory-NEW' USING
             BY VALUE THREAD-FACT-ID
         RETURNING THREAD-FACT-PTR.
     MOVE 1 TO THREAD-CREATE-FLAG.

* 6. TSCルートアクセプタの生成および各種設定
```

```
      SET THREAD-FACT-PTR TO NULL.
      CALL 'TSCRAcceptor-create' USING
              BY VALUE SERVER-PTR
              BY VALUE THREAD-FACT-PTR
              BY REFERENCE CORBA-ENVIRONMENT
          RETURNING R-ACCEPTOR-PTR.
* 例外チェック
      IF NOT CORBA-NO-EXCEPTION THEN
          CALL 'OTM-EXCEPTION-HANDLER' USING
                  BY REFERENCE MAJOR
                  BY REFERENCE CORBA-ENVIRONMENT
          CALL 'TSCSysExcept-DELETE' USING
              BY VALUE EXCEP OF CORBA-ENVIRONMENT
          MOVE 1 TO RETURN-CODE
          GO TO PROG-END
      END-IF.
      DISPLAY 'Success TSCRAcceptor-create'.
      MOVE 1 TO RACPT-CREATE-FLAG.

* TSCアクセプタの登録
      CALL 'TSCRAcceptor-registerAcceptor' USING
              BY VALUE R-ACCEPTOR-PTR
              BY VALUE ACCEPTOR-PTR
              BY REFERENCE CORBA-ENVIRONMENT
          RETURNING ACCEPTOR-ID.
* 例外チェック
      IF NOT CORBA-NO-EXCEPTION THEN
          CALL 'OTM-EXCEPTION-HANDLER' USING
                  BY REFERENCE MAJOR
                  BY REFERENCE CORBA-ENVIRONMENT
          CALL 'TSCSysExcept-DELETE' USING
              BY VALUE EXCEP OF CORBA-ENVIRONMENT
          MOVE 1 TO RETURN-CODE
          GO TO PROG-END
      END-IF.
      DISPLAY 'Success TSCRAcceptor-registerAcceptor'.

* 7. TSCルートアクセプタの活性化
      MOVE 'serviceX' TO R-ACCEPTOR-NAME.
      CALL 'CORBA_string_set' USING
              BY REFERENCE R-ACCEPTOR-NAME-PTR
              BY REFERENCE R-ACCEPTOR-NAME-LEN
              BY REFERENCE R-ACCEPTOR-NAME.
      CALL 'TSCRAcceptor-activate' USING
              BY VALUE R-ACCEPTOR-PTR
              BY VALUE R-ACCEPTOR-NAME-PTR
              BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
      IF NOT CORBA-NO-EXCEPTION THEN
          CALL 'OTM-EXCEPTION-HANDLER' USING
                  BY REFERENCE MAJOR
                  BY REFERENCE CORBA-ENVIRONMENT
          CALL 'TSCSysExcept-DELETE' USING
              BY VALUE EXCEP OF CORBA-ENVIRONMENT
          MOVE 1 TO RETURN-CODE
          GO TO PROG-END
      END-IF.
```

```
     DISPLAY 'Success TSCRAcceptor-activate'.
     MOVE 1 TO RACPT-ACTIVATE-FLAG.

* 8. 実行制御の受け渡し
     DISPLAY 'Start TSCAdm-serverMainloop'.
     CALL 'TSCAdm-serverMainloop' USING
             BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCAdm-serverMainloop'.

 PROG-END.

* 9. TSCルートアクセプタの非活性化
     IF RACPT-ACTIVATE-FLAG = 1 THEN
         MOVE 0 TO DEACT-MODE
         CALL 'TSCRAcceptor-deactivate' USING
                 BY VALUE R-ACCEPTOR-PTR
                 BY VALUE DEACT-MODE
                 BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
         IF NOT CORBA-NO-EXCEPTION THEN
             CALL 'OTM-EXCEPTION-HANDLER' USING
                     BY REFERENCE MAJOR
                     BY REFERENCE CORBA-ENVIRONMENT
             CALL 'TSCSysExcept-DELETE' USING
                     BY VALUE EXCEP OF CORBA-ENVIRONMENT
             MOVE 1 TO RETURN-CODE
         ELSE
             DISPLAY 'Success TSCRAcceptor-deactivate'
         END-IF
     END-IF.

* 10. TSCルートアクセプタの削除
     IF RACPT-CREATE-FLAG = 1 THEN
         CALL 'TSCRAcceptor-destroy' USING
                 BY VALUE R-ACCEPTOR-PTR
                 BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
         IF NOT CORBA-NO-EXCEPTION THEN
             CALL 'OTM-EXCEPTION-HANDLER' USING
                     BY REFERENCE MAJOR
                     BY REFERENCE CORBA-ENVIRONMENT
             CALL 'TSCSysExcept-DELETE' USING
                     BY VALUE EXCEP OF CORBA-ENVIRONMENT
             MOVE 1 TO RETURN-CODE
         ELSE
             DISPLAY 'Success TSCRAcceptor-destroy'
         END-IF
```

```
    END-IF.

* 11. TSCユーザオブジェクトファクトリ
*     およびTSCユーザアクセプタの削除
    IF NOT THREAD-FACT-PTR = NULL
    AND THREAD-CREATE-FLAG = 1 THEN
        CALL 'TSCCBLThreadFactory-DELETE' USING
                BY VALUE THREAD-FACT-PTR
    END-IF.

    IF TSCACPT-CREATE-FLAG = 1 THEN
        CALL 'CBLClass_TSCacpt-DEL' USING
                BY VALUE ACCEPTOR-PTR
                BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        END-IF
    END-IF.

* 領域の解放
    IF TSCFACT-CREATE-FLAG = 1 THEN
        CALL 'CBLClass_TSCfact-DEL' USING
                BY VALUE FACTORY-PTR
    END-IF.

* 12. TSCデーモンへの接続解放
    IF GET-SERVER-FLAG = 1 THEN
        CALL 'TSCAdm-releaseTSCServer' USING
                BY VALUE SERVER-PTR
                BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        ELSE
            DISPLAY 'Success TSCAdm-releaseTSCServer'
        END-IF
    END-IF.

    IF DOMAIN-CREATE-FLAG = 1 THEN
        CALL 'TSCDomain-DELETE' USING
                BY VALUE DOMAIN-PTR
                BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
```

```
                    BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            END-IF
        END-IF.

* 13. TPBroker OTMの終了処理
        IF INIT-SERVER-FLAG = 1 THEN
            CALL 'TSCAdm-endServer' USING
                    BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            ELSE
                DISPLAY 'Success TSCAdm-endServer'
            END-IF
        END-IF.

 END PROGRAM CORBA-SERVER-MAIN.
```

6.7 ユーザ例外通知を利用するアプリケーションプログラム（COBOL）

ユーザ例外通知を利用するアプリケーションプログラムの COBOL での作成例を示します。これらのサンプルコードは製品で提供されています。

● ユーザ定義 IDL インタフェースの例

ユーザ定義 IDL インタフェースの例を次に示します。

```
//
// "EEEfile.idl"
//

exception UserExcept {
  long value;
};

interface CBLClass
{
  void call() raises (UserExcept);
};
```

● IDL コンパイラが生成するクラス

COBOL adapter for TPBroker の IDL コンパイラはユーザ定義 IDL インタフェースから次のファイルを生成します。

- EEEfile_c.ocb
- EEEfile_s.ocb

● トランザクションフレームジェネレータが生成するクラス

OTM のトランザクションフレームジェネレータは，ユーザ定義 IDL インタフェースから次の表に示すクラスを生成します。ただし，実際に生成されるのはクラスを使用するための副プログラムです。副プログラムの名称は次のとおりになります。

"クラス名"－"メソッド名"

表 6-4 ユーザ例外通知を利用するアプリケーションプログラムの作成時にトランザクションフレームジェネレータが生成するクラス（COBOL）

分類	ファイル名	副プログラムのプリフィックス（オブジェクト名）
クライアントアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• EEEfile_TSC_c.ocb	• CBLClass_TSCprxy（TSC ユーザプロキシ）

分類	ファイル名	副プログラムのプリフィックス（オブジェクト名）
サーバアプリケーション用のユーザ定義 IDL インタフェース依存クラス	• EEEfile_TSC_s.ocb	• CBLClass_TSCsk（TSC ユーザスケルトン） • CBLClass_TSCacpt（TSC ユーザアクセプタ）
雛形クラス	• EEEfile_TSC_t.ocb	• CBLClass_TSCimpl（TSC ユーザオブジェクト） • CBLClass_TSCfact（TSC ユーザオブジェクトファクトリ） • TSCCBLThreadFactory（TSC ユーザスレッドファクトリ） • TSCCBLThread（TSC ユーザスレッド）

6.7.1 ユーザ例外通知を利用するクライアントアプリケーションの例（COBOL）

ユーザ例外通知を利用するクライアントアプリケーションの処理の流れとコードの例を示します。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

なお，ユーザ例外通知を利用するクライアントアプリケーションの例外処理は，同期型呼び出しの場合と同様です。「6.2.3　例外処理のコードの例（COBOL）」を参照してください。

（1） サービス利用処理の流れ

1. COBOL adapter for TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザプロキシの生成および各種設定**
5. TSC **ユーザプロキシのメソッド呼び出し（サーバ側のオブジェクトの呼び出し）**
6. TSC **ユーザプロキシの削除**
7. TSC **デーモンへの接続解放**
8. TPBroker OTM **の終了処理**

（2） サービス利用処理のコード

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CLIENT.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.

       DATA DIVISION.
       WORKING-STORAGE SECTION.

       01 ORB-PTR                USAGE POINTER.
       01 DOMAIN-PTR             USAGE POINTER.
```

```
 01 CLIENT-PTR            USAGE POINTER.
 01 CLIENT-WAY            PIC S9(9) COMP.
 01 MY-DOMAIN-NAME        PIC X(10).
 01 MY-DOMAIN-NAME-PTR    USAGE POINTER.
 01 MY-DOMAIN-NAME-LEN    PIC S9(9) COMP.
 01 MY-TSCID              PIC X(10).
 01 MY-TSCID-PTR          USAGE POINTER.
 01 MY-TSCID-LEN          PIC S9(9) COMP.
 01 MY-DOMAIN-FLAG        PIC S9(9) COMP VALUE 1.
 01 ACCEPTOR-NAME         PIC X(20).
 01 ACCEPTOR-NAME-PTR     USAGE POINTER VALUE NULL.
 01 ACCEPTOR-NAME-LEN     PIC S9(9) COMP.
 01 PROXY-PTR             USAGE POINTER.

 01 INIT-CLIENT-FLAG      PIC S9(9) COMP.
 01 DOMAIN-CREATE-FLAG    PIC S9(9) COMP.
 01 GET-CLIENT-FLAG       PIC S9(9) COMP.
 01 TSCPRXY-CREATE-FLAG   PIC S9(9) COMP.

 01 CORBA-ENVIRONMENT.
  02 MAJOR                PIC 9(9) COMP.
   88 CORBA-NO-EXCEPTION     VALUE 0.
   88 CORBA-USER-EXCEPTION   VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION VALUE 2.
  02 EXCEP                 USAGE POINTER.
  02 FUNC-NAME             PIC X(256).

 01 ERR-CODE               PIC S9(9) COMP.

* Sequence
 01 EXCEPTION-NAME-PTR         USAGE POINTER.
 01 EXCEPTION-NAME-LEN         PIC S9(9) COMP.
 01 EXCEPTION-NAME             PIC X(30).
 01 EXCEPTION-HANDLE           USAGE POINTER.
 01 EXCEPTION-VALUE            PIC S9(9) COMP.

 LINKAGE SECTION.
 01 ARGC                   PIC 9(9) COMP.
 01 ARGV                   USAGE POINTER.

 PROCEDURE DIVISION USING
         BY VALUE ARGC
         BY VALUE ARGV.

 MOVE 0 TO RETURN-CODE.
 MOVE 0 TO INIT-CLIENT-FLAG.
 MOVE 0 TO DOMAIN-CREATE-FLAG.
 MOVE 0 TO GET-CLIENT-FLAG.
 MOVE 0 TO TSCPRXY-CREATE-FLAG.

* ORBの初期化処理
 CALL 'CORBA-STUB-INIT-EEEfile'.

* 1. COBOL adapter for TPBrokerの初期化処理
 CALL 'CORBA_orb_init' USING
         BY REFERENCE ARGC
         BY REFERENCE ARGV
         BY REFERENCE CORBA-ENVIRONMENT
```

```
     RETURNING ORB-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'CORBA_FreeException' USING
             EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success CORBA_orb_init'.

* 2. TPBroker OTMの初期化処理
 CALL 'TSCAdm-initClient' USING
         BY REFERENCE ARGC
         BY REFERENCE ARGV
         BY VALUE     ORB-PTR
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success TSCAdm-initClient'.
 MOVE 1 TO INIT-CLIENT-FLAG.

* 3. TSCデーモンへの接続
* (1) TSCDomainの生成
 SET MY-DOMAIN-NAME-PTR TO NULL.
 SET MY-TSCID-PTR TO NULL.
 MOVE 1 TO MY-DOMAIN-FLAG.
 CALL 'TSCDomain-NEW' USING
         BY VALUE     MY-DOMAIN-NAME-PTR
         BY VALUE     MY-TSCID-PTR
         BY VALUE     MY-DOMAIN-FLAG
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING DOMAIN-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success TSCDomain-NEW:DomainName = '
                                       MY-DOMAIN-NAME.
 MOVE 1 TO DOMAIN-CREATE-FLAG.
```

```
* (2) TSCClientの取得
 MOVE 1 TO CLIENT-WAY.
 CALL 'TSCAdm-getTSCClient' USING
         BY VALUE     DOMAIN-PTR
         BY VALUE     CLIENT-WAY
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING CLIENT-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success TSCAdm-getTSCClient'.
 MOVE 1 TO GET-CLIENT-FLAG.

* 4. TSCユーザプロキシの生成および各種設定
*    IDLインタフェース"CBLClass"用のTSCProxy生成
 SET ACCEPTOR-NAME-PTR TO NULL.
 CALL 'CBLClass_TSCprxy-NEW' USING
         BY VALUE     CLIENT-PTR
         BY VALUE     ACCEPTOR-NAME-PTR
         BY REFERENCE CORBA-ENVIRONMENT
     RETURNING PROXY-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
     MOVE 1 TO RETURN-CODE
     GO TO PROG-END
 END-IF.
 DISPLAY 'Success CBLClass_TSCprxy-NEW'.
 MOVE 1 TO TSCPRXY-CREATE-FLAG.

* 5. TSCユーザプロキシのメソッド呼び出し
*    （サーバ側のオブジェクトの呼び出し）
************************************************
* サーバメソッドの呼び出し
************************************************
 DISPLAY 'Start CBLClass-call'.
 CALL 'CBLClass-call' USING
         BY VALUE     PROXY-PTR
         BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'OTM-EXCEPTION-HANDLER' USING
             BY REFERENCE MAJOR
             BY REFERENCE CORBA-ENVIRONMENT
     IF CORBA-SYSTEM-EXCEPTION THEN
         CALL 'TSCSysExcept-DELETE' USING
```

```
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
            GO TO PROG-END
        END-IF
* ユーザ例外受信時の処理
        IF CORBA-USER-EXCEPTION THEN
            CALL 'CORBA-get-exception-name' USING
                    BY REFERENCE EXCEP
                RETURNING EXCEPTION-NAME-PTR
            MOVE 30 TO EXCEPTION-NAME-LEN
            CALL 'CORBA_string_get' USING
                    BY REFERENCE EXCEPTION-NAME-PTR
                    BY REFERENCE EXCEPTION-NAME-LEN
                    BY REFERENCE EXCEPTION-NAME
            DISPLAY 'Exception-name = ' EXCEPTION-NAME
            CALL 'CORBA-GetExceptionHandle' USING
                    BY REFERENCE EXCEP
                RETURNING EXCEPTION-HANDLE
            CALL 'GET-value-UserExcept' USING
                    BY VALUE EXCEPTION-HANDLE
                RETURNING EXCEPTION-VALUE
            DISPLAY 'Exception-value = ' EXCEPTION-VALUE
        END-IF
    END-IF.
    DISPLAY 'Success CBLClass-call'.

    PROG-END.

* 6. TSCユーザプロキシの削除
*    領域の解放
        IF TSCPRXY-CREATE-FLAG = 1 THEN
            CALL 'CBLClass_TSCprxy-DEL' USING
                    BY VALUE PROXY-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                        BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            END-IF
        END-IF.

* 7. TSCデーモンへの接続解放
        IF GET-CLIENT-FLAG = 1 THEN
            CALL 'TSCAdm-releaseTSCClient' USING
                    BY VALUE CLIENT-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                        BY VALUE EXCEP OF CORBA-ENVIRONMENT
```

```
                MOVE 1 TO RETURN-CODE
            ELSE
                DISPLAY 'Success TSCAdm-releaseTSCClient'
            END-IF
        END-IF.

        IF DOMAIN-CREATE-FLAG = 1 THEN
            CALL 'TSCDomain-DELETE' USING
                    BY VALUE DOMAIN-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
    * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                        BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            END-IF
        END-IF.

    * 8. TPBroker OTMの終了処理
        IF INIT-CLIENT-FLAG = 1 THEN
            CALL 'TSCAdm-endClient' USING
                    BY REFERENCE CORBA-ENVIRONMENT
    * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                        BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            ELSE
                DISPLAY 'Success TSCAdm-endClient'
            END-IF
        END-IF.

     END PROGRAM CLIENT.
```

6.7.2 ユーザ例外通知を利用するサーバアプリケーションの例（COBOL）

ユーザ例外通知を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス CBLClass_TSCimpl に TSC ユーザオブジェクトのコードを記述します。また，雛形クラス CBLClass_TSCfact に TSC ユーザオブジェクトファクトリのコードを記述します。

なお，ユーザ例外通知を利用するサーバアプリケーションの例外処理は，同期型呼び出しの場合と同様です。「6.2.3　例外処理のコードの例（COBOL）」を参照してください。

（1）TSC ユーザオブジェクト（CBLClass_TSCimpl）と TSC ユーザオブジェクトファクトリ（CBLClass_TSCfact）のコード

```
*****************************************************
* Operation 'call'
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'call'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 CORBA-ENVIRONMENT.
  02 MAJOR                    PIC 9(9) COMP.
   88 CORBA-NO-EXCEPTION          VALUE 0.
   88 CORBA-USER-EXCEPTION        VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION      VALUE 2.
  02 EXCEP                    USAGE POINTER.
  02 FUNC-NAME                PIC X(256).

* 必要に応じてデータ宣言を追加できます。
 01 EXCEPTION-PTR    USAGE POINTER.
 01 EXCEPTION-VALUE  PIC S9(9) COMP.

 LINKAGE SECTION.

* Do not change signature of this sub-program.
 PROCEDURE DIVISION.

* Write user own code.
* ユーザメソッドのコードを記述します。
 DISPLAY 'call method in CBLClass'.

* ユーザ例外の作成
      CALL 'sk-CBL-NEW-UserExcept'
          RETURNING EXCEPTION-PTR.

* ユーザ例外のデータの設定
      MOVE 123456 TO EXCEPTION-VALUE.
      CALL 'sk-SET-value-UserExcept' USING
          BY VALUE     EXCEPTION-PTR
          BY REFERENCE EXCEPTION-VALUE.

* ユーザ例外のthrow
      CALL 'CBL-THROW' USING
          BY VALUE     EXCEPTION-PTR.

 END PROGRAM 'call'.


*****************************************************
* Constructor of 'CBLClass_TSCimpl'
*****************************************************
* Constructor of OTM Object
* Implement.
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING SKELETON-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
* コンストラクタの引数の数および型を変更することもできます。

* This sub-program must return a pointer
* that 'CBLClass_TSCsk-NEW' sub-program returns.
     CALL 'CBLClass_TSCsk-NEW'
         RETURNING SKELETON-POINTER.
 END PROGRAM 'CBLClass_TSCimpl-NEW'.

*****************************************************
* Destructor of 'CBLClass_TSCimpl'
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE SKELETON-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。

* This sub-program must call
* 'CBLClass_TSCsk-DEL' sub-program.
     CALL 'CBLClass_TSCsk-DEL' USING
         BY VALUE SKELETON-POINTER.
 END PROGRAM 'CBLClass_TSCimpl-DEL'.

*****************************************************
* Constructor of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
```

```
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
* 引数の数および型を変更することもできます。

* This sub-program must return a pointer that
* 'CBLClass_TSCfact-get' sub-program returns.
     CALL 'CBLClass_TSCfact-get'
         RETURNING FACTORY-POINTER.
 END PROGRAM 'CBLClass_TSCfact-NEW'.


*****************************************************
* Destructor of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。

* This sub-program must call
* 'CBLClass_TSCfact-rls'.sub-program.
     CALL 'CBLClass_TSCfact-rls' USING
         BY VALUE FACTORY-POINTER.
 END PROGRAM 'CBLClass_TSCfact-DEL'.


*****************************************************
* 'create' method of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-crt'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
* Do not change signature of this sub-program.
```

```
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER
     RETURNING OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを生成するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-NEW' returns.
     CALL 'CBLClass_TSCimpl-NEW'
         RETURNING OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-crt'.


*****************************************************
* 'destroy' method of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-dst'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
* Do not change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER
         BY VALUE OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを削除するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-DEL' returns.
     CALL 'CBLClass_TSCimpl-DEL' USING
         BY VALUE OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-dst'.


*****************************************************
* TSCCBLThread-beginThread of
*     TSCCBLThread
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-beginThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 BEGIN-THREAD-PTR USAGE POINTER.
 01 END-THREAD-PTR USAGE POINTER.
```

```
* Do not change signature of this sub-program.
 LINKAGE SECTION.
 01 THREAD-FACTORY-ID PIC S9(9) COMP.
 PROCEDURE DIVISION USING
         BY VALUE THREAD-FACTORY-ID.

* Write user own code.
* スレッド開始処理を記述します。
* 必要に応じて変更してください。

 END PROGRAM 'TSCCBLThread-beginThread'.


*****************************************************
* TSCCBLThread-endThread of
*     TSCCBLThreadFactory
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-endThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* Do not change signature of this sub-program.
 LINKAGE SECTION.
 01 THREAD-FACTORY-ID PIC S9(9) COMP.
 PROCEDURE DIVISION USING
         BY VALUE THREAD-FACTORY-ID.

* Write user own code.
* スレッド終了処理を記述します。
* 必要に応じて変更してください。

 END PROGRAM 'TSCCBLThread-endThread'.
```

（2） サービス登録処理の流れ

1. COBOL adapter for TPBroker **の初期化処理**
2. TPBroker OTM **の初期化処理**
3. TSC **デーモンへの接続**
4. TSC **ユーザアクセプタの生成および各種設定**
5. TSC **ルートアクセプタの生成および各種設定**
6. TSC **ルートアクセプタの活性化**
7. **実行制御の受け渡し**
8. TSC **ルートアクセプタの非活性化**
9. TSC **ルートアクセプタの削除**
10. TSC **ユーザオブジェクトファクトリおよび** TSC **ユーザアクセプタの削除**

11. TSC デーモンへの接続解放

12. TPBroker OTM の終了処理

(3) サービス登録処理のコード

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CORBA-SERVER-MAIN.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       REPOSITORY.
       CLASS CBLClass IS 'EEEfile_s'.
       DATA DIVISION.
      * The following are the ORB pointers.
      * These are necessary for all servers.
       WORKING-STORAGE SECTION.
       01 ORB-PTR               USAGE POINTER.

       01 DOMAIN-PTR            USAGE POINTER.
       01 SERVER-PTR            USAGE POINTER.
       01 FACTORY-PTR           USAGE POINTER.
       01 ACCEPTOR-PTR          USAGE POINTER.
       01 R-ACCEPTOR-PTR        USAGE POINTER.
       01 THREAD-FACT-PTR       USAGE POINTER.
       01 THREAD-FACT-ID        PIC S9(9) COMP.
       01 MY-DOMAIN-NAME        PIC X(10).
       01 MY-DOMAIN-NAME-PTR    USAGE POINTER.
       01 MY-DOMAIN-NAME-LEN    PIC S9(9) COMP.
       01 MY-TSCID              PIC X(10).
       01 MY-TSCID-PTR          USAGE POINTER.
       01 MY-TSCID-LEN          PIC S9(9) COMP.
       01 MY-DOMAIN-FLAG        PIC S9(9) COMP.
       01 ACCEPTOR-NAME-PTR     USAGE POINTER.
       01 ACCEPTOR-ID           PIC S9(9) COMP.
       01 P-COUNT               PIC S9(9) COMP.
       01 DEACT-MODE            PIC S9(9) COMP.
       01 R-ACCEPTOR-NAME       PIC X(10).
       01 R-ACCEPTOR-NAME-PTR   USAGE POINTER.
       01 R-ACCEPTOR-NAME-LEN   PIC S9(9) COMP VALUE 30.

       01 INIT-SERVER-FLAG      PIC S9(9) COMP.
       01 DOMAIN-CREATE-FLAG    PIC S9(9) COMP.
       01 GET-SERVER-FLAG       PIC S9(9) COMP.
       01 TSCFACT-CREATE-FLAG   PIC S9(9) COMP.
       01 TSCACPT-CREATE-FLAG   PIC S9(9) COMP.
       01 RACPT-CREATE-FLAG     PIC S9(9) COMP.
       01 RACPT-ACTIVATE-FLAG   PIC S9(9) COMP.

       01 CORBA-ENVIRONMENT.
        02 MAJOR PIC 9(9) USAGE COMP.
         88 CORBA-NO-EXCEPTION     VALUE 0.
         88 CORBA-USER-EXCEPTION   VALUE 1.
         88 CORBA-SYSTEM-EXCEPTION VALUE 2.
        02 EXCEP USAGE POINTER.
        02 FUNC-NAME PIC X(256).

       LINKAGE SECTION.
```

```
 01 ARGC PIC S9(9) USAGE COMP.
 01 ARGV USAGE POINTER.

 PROCEDURE DIVISION USING
         BY VALUE ARGC
         BY VALUE ARGV.

     MOVE 0 TO RETURN-CODE.
     MOVE 0 TO INIT-SERVER-FLAG.
     MOVE 0 TO DOMAIN-CREATE-FLAG.
     MOVE 0 TO GET-SERVER-FLAG.
     MOVE 0 TO TSCFACT-CREATE-FLAG.
     MOVE 0 TO TSCACPT-CREATE-FLAG.
     MOVE 0 TO RACPT-CREATE-FLAG.
     MOVE 0 TO RACPT-ACTIVATE-FLAG.

* 1. COBOL adapter for TPBrokerの初期化処理
* First, call the skeleton and class initializers.
     CALL 'CORBA-SKEL-INIT-EEEfile'.
     CALL 'CBLClass-CLASS-INIT' USING
         BY VALUE CBLClass.

* ORBの初期化
     CALL 'CORBA_orb_init' USING
         BY REFERENCE ARGC
         BY REFERENCE ARGV
         BY REFERENCE CORBA-ENVIRONMENT
         RETURNING    ORB-PTR.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'CORBA_FreeException' USING
                 EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success CORBA_orb_init'.

* 2. TPBroker OTMの初期化処理
     CALL 'TSCAdm-initServer' USING
             BY REFERENCE ARGC
             BY REFERENCE ARGV
             BY VALUE      ORB-PTR
             BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCAdm-initServer'.
```

```
           MOVE 1 TO INIT-SERVER-FLAG.

       * 3. TSCデーモンへの接続
           SET MY-DOMAIN-NAME-PTR TO NULL.
           SET MY-TSCID-PTR TO NULL.
           MOVE 1 TO MY-DOMAIN-FLAG.
           CALL 'TSCDomain-NEW' USING
                   BY VALUE MY-DOMAIN-NAME-PTR
                   BY VALUE MY-TSCID-PTR
                   BY VALUE MY-DOMAIN-FLAG
                   BY REFERENCE CORBA-ENVIRONMENT
               RETURNING DOMAIN-PTR.
       * 例外チェック
           IF NOT CORBA-NO-EXCEPTION THEN
               CALL 'OTM-EXCEPTION-HANDLER' USING
                       BY REFERENCE MAJOR
                       BY REFERENCE CORBA-ENVIRONMENT
               CALL 'TSCSysExcept-DELETE' USING
                       BY VALUE EXCEP OF CORBA-ENVIRONMENT
               MOVE 1 TO RETURN-CODE
               GO TO PROG-END
           END-IF.
           DISPLAY 'Success TSCDomain-NEW'.
           MOVE 1 TO DOMAIN-CREATE-FLAG.

       * TSCServerの取得
           CALL 'TSCAdm-getTSCServer' USING
                   BY VALUE DOMAIN-PTR
                   BY REFERENCE CORBA-ENVIRONMENT
               RETURNING SERVER-PTR.
       * 例外チェック
           IF NOT CORBA-NO-EXCEPTION THEN
               CALL 'OTM-EXCEPTION-HANDLER' USING
                       BY REFERENCE MAJOR
                       BY REFERENCE CORBA-ENVIRONMENT
               CALL 'TSCSysExcept-DELETE' USING
                       BY VALUE EXCEP OF CORBA-ENVIRONMENT
               MOVE 1 TO RETURN-CODE
               GO TO PROG-END
           END-IF.
           DISPLAY 'Success TSCAdm-getTSCServer'.
           MOVE 1 TO GET-SERVER-FLAG.

       * 4. TSCユーザアクセプタの生成および各種設定
       * CBLClass_TSCfactの生成
           CALL 'CBLClass_TSCfact-NEW'
               RETURNING FACTORY-PTR.
           DISPLAY 'Success CBLClass_TSCfact-NEW'.
           MOVE 1 TO TSCFACT-CREATE-FLAG.

       * TSCAcceptorの生成
           SET ACCEPTOR-NAME-PTR TO NULL.
           CALL 'CBLClass_TSCacpt-NEW' USING
                   BY VALUE FACTORY-PTR
                   BY VALUE ACCEPTOR-NAME-PTR
                   BY REFERENCE CORBA-ENVIRONMENT
               RETURNING ACCEPTOR-PTR.
```

```
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
                 BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success CBLClass_TSCacpt-NEW'.
     MOVE 1 TO TSCACPT-CREATE-FLAG.

* 5. TSCルートアクセプタの生成および各種設定
     SET THREAD-FACT-PTR TO NULL.
     CALL 'TSCRAcceptor-create' USING
             BY VALUE SERVER-PTR
             BY VALUE THREAD-FACT-PTR
             BY REFERENCE CORBA-ENVIRONMENT
         RETURNING R-ACCEPTOR-PTR.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCRAcceptor-create'.
     MOVE 1 TO RACPT-CREATE-FLAG.

* TSCアクセプタの登録
     CALL 'TSCRAcceptor-registerAcceptor' USING
             BY VALUE R-ACCEPTOR-PTR
             BY VALUE ACCEPTOR-PTR
             BY REFERENCE CORBA-ENVIRONMENT
         RETURNING ACCEPTOR-ID.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCRAcceptor-registerAcceptor'.

* 6. TSCルートアクセプタの活性化
     MOVE 'serviceX' TO R-ACCEPTOR-NAME.
     CALL 'CORBA_string_set' USING
             BY REFERENCE R-ACCEPTOR-NAME-PTR
             BY REFERENCE R-ACCEPTOR-NAME-LEN
             BY REFERENCE R-ACCEPTOR-NAME.
```

```
     CALL 'TSCRAcceptor-activate' USING
             BY VALUE R-ACCEPTOR-PTR
             BY VALUE R-ACCEPTOR-NAME-PTR
             BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCRAcceptor-activate'.
     MOVE 1 TO RACPT-ACTIVATE-FLAG.

* 7. 実行制御の受け渡し
     DISPLAY 'Start TSCAdm-serverMainloop'.
     CALL 'TSCAdm-serverMainloop' USING
             BY REFERENCE CORBA-ENVIRONMENT.
* 例外チェック
     IF NOT CORBA-NO-EXCEPTION THEN
         CALL 'OTM-EXCEPTION-HANDLER' USING
                 BY REFERENCE MAJOR
                 BY REFERENCE CORBA-ENVIRONMENT
         CALL 'TSCSysExcept-DELETE' USING
             BY VALUE EXCEP OF CORBA-ENVIRONMENT
         MOVE 1 TO RETURN-CODE
         GO TO PROG-END
     END-IF.
     DISPLAY 'Success TSCAdm-serverMainloop'.

 PROG-END.

* 8. TSCルートアクセプタの非活性化
     IF RACPT-ACTIVATE-FLAG = 1 THEN
         MOVE 0 TO DEACT-MODE
         CALL 'TSCRAcceptor-deactivate' USING
                 BY VALUE R-ACCEPTOR-PTR
                 BY VALUE DEACT-MODE
                 BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
         IF NOT CORBA-NO-EXCEPTION THEN
             CALL 'OTM-EXCEPTION-HANDLER' USING
                     BY REFERENCE MAJOR
                     BY REFERENCE CORBA-ENVIRONMENT
             CALL 'TSCSysExcept-DELETE' USING
                     BY VALUE EXCEP OF CORBA-ENVIRONMENT
             MOVE 1 TO RETURN-CODE
         ELSE
             DISPLAY 'Success TSCRAcceptor-deactivate'
         END-IF
     END-IF.

* 9. TSCルートアクセプタの削除
     IF RACPT-CREATE-FLAG = 1 THEN
```

```
        CALL 'TSCRAcceptor-destroy' USING
                BY VALUE R-ACCEPTOR-PTR
                BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        ELSE
            DISPLAY 'Success TSCRAcceptor-destroy'
        END-IF
    END-IF.

* 10. ユーザオブジェクトファクトリおよびTSCユーザアクセプタの削除
    IF TSCACPT-CREATE-FLAG = 1 THEN
        CALL 'CBLClass_TSCacpt-DEL' USING
                BY VALUE ACCEPTOR-PTR
                BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        END-IF
    END-IF.

* 領域の解放
    IF TSCFACT-CREATE-FLAG = 1 THEN
        CALL 'CBLClass_TSCfact-DEL' USING
                BY VALUE FACTORY-PTR
    END-IF.

* 11. TSCデーモンへの接続解放
    IF GET-SERVER-FLAG = 1 THEN
        CALL 'TSCAdm-releaseTSCServer' USING
                BY VALUE SERVER-PTR
                BY REFERENCE CORBA-ENVIRONMENT
* 例外チェック
        IF NOT CORBA-NO-EXCEPTION THEN
            CALL 'OTM-EXCEPTION-HANDLER' USING
                    BY REFERENCE MAJOR
                    BY REFERENCE CORBA-ENVIRONMENT
            CALL 'TSCSysExcept-DELETE' USING
                BY VALUE EXCEP OF CORBA-ENVIRONMENT
            MOVE 1 TO RETURN-CODE
        ELSE
            DISPLAY 'Success TSCAdm-releaseTSCServer'
        END-IF
    END-IF.

    IF DOMAIN-CREATE-FLAG = 1 THEN
```

```
            CALL 'TSCDomain-DELETE' USING
                    BY VALUE DOMAIN-PTR
                    BY REFERENCE CORBA-ENVIRONMENT
      * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            END-IF
        END-IF.

      * 12. TPBroker OTMの終了処理
        IF INIT-SERVER-FLAG = 1 THEN
            CALL 'TSCAdm-endServer' USING
                    BY REFERENCE CORBA-ENVIRONMENT
      * 例外チェック
            IF NOT CORBA-NO-EXCEPTION THEN
                CALL 'OTM-EXCEPTION-HANDLER' USING
                        BY REFERENCE MAJOR
                        BY REFERENCE CORBA-ENVIRONMENT
                CALL 'TSCSysExcept-DELETE' USING
                    BY VALUE EXCEP OF CORBA-ENVIRONMENT
                MOVE 1 TO RETURN-CODE
            ELSE
                DISPLAY 'Success TSCAdm-endServer'
            END-IF
        END-IF.

       END PROGRAM CORBA-SERVER-MAIN.
```

6.8 TSCWatchTime を利用するアプリケーションプログラム（COBOL）

TSCWatchTime を利用するアプリケーションプログラムの COBOL での作成例を示します。

ユーザ定義 IDL インタフェースの例，IDL コンパイラが生成するクラス，およびトランザクションフレームジェネレータが生成するクラスは，同期型呼び出しの場合と同様です。「6.2　同期型呼び出しをするアプリケーションプログラム（COBOL）」を参照してください。

6.8.1 TSCWatchTime を利用するクライアントアプリケーションの例（COBOL）

同期型呼び出しの場合と同様です。「6.2.1　同期型呼び出しをするクライアントアプリケーションの例（COBOL）」を参照してください。例外処理については，「6.2.3　例外処理のコードの例（COBOL）」を参照してください。

6.8.2 TSCWatchTime を利用するサーバアプリケーションの例（COBOL）

TSCWatchTime を利用するサーバアプリケーションの処理の流れとコードの例を示します。*斜体*で示しているコードは，雛形クラスとして自動生成される部分です。**太字**で示しているコードは，同期型呼び出しのコードと異なる部分です。

サーバアプリケーションの作成時には，ユーザは，自動生成された雛形クラス CBLClass_TSCimpl にコードを記述します。また，雛形クラス CBLClass_TSCfact に TSC ユーザオブジェクトファクトリのコードを記述します。

なお，TSCWatchTime を利用するサーバアプリケーションの例外処理は，同期型呼び出しの場合と同様です。「6.2.3　例外処理のコードの例（COBOL）」を参照してください。

（1）TSC ユーザオブジェクト（CBLClass_TSCimpl）と TSC ユーザオブジェクトファクトリ（CBLClass_TSCfact）のコード

```
*****************************************************
* Operation 'call'
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'call'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
```

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 CORBA-ENVIRONMENT.
  02 MAJOR                    PIC 9(9) COMP.
   88 CORBA-NO-EXCEPTION          VALUE 0.
   88 CORBA-USER-EXCEPTION        VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION      VALUE 2.
  02 EXCEP                    USAGE POINTER.
  02 FUNC-NAME                PIC X(256).

 01 TSC-SEQMAXLEN PIC 9(9) USAGE COMP.
 01 TSC-TC-1 USAGE POINTER.
 01 TSC-SEQENV.
   02 MAJOR PIC 9(9) USAGE COMP.
   02 EXCEP USAGE POINTER.
   02 FUNC-NAME PIC X(256).
 01 TSC-TC-2 USAGE POINTER.

* 必要に応じてデータ宣言を追加できます。
 01 ERR-CODE                  PIC S9(9) COMP.
 01 TYPE-CODE-PTR             USAGE POINTER.
 01 MY-OUT-DATA-LEN           PIC S9(9) COMP.

 LINKAGE SECTION.
 01 in_data USAGE POINTER.
 01 out_data USAGE POINTER.

* Do not change signature of this sub-program.
 PROCEDURE DIVISION
     USING
         BY VALUE in_data
         BY REFERENCE out_data.

* Write user own code.
* ユーザメソッドのコードを記述します。
 DISPLAY 'call method in CBLClass'.

* OUT属性の引数を作成します。
* Octet TypeCode オブジェクトを作成します。
 CALL 'Create_CORBA_TypeCode' USING
         BY VALUE     10
         BY VALUE     1
         BY REFERENCE CORBA-ENVIRONMENT
         RETURNING TYPE-CODE-PTR.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
              BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
              BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

* Octet型のSequenceオブジェクトを生成します。
 MOVE 999999999 TO MY-OUT-DATA-LEN.
 CALL 'CORBA-SeqAlloc' USING
         BY REFERENCE MY-OUT-DATA-LEN
         BY REFERENCE TYPE-CODE-PTR
         BY REFERENCE out_data
```

```
          BY REFERENCE CORBA-ENVIRONMENT
          RETURNING ERR-CODE.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
              BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
              BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

 CALL 'CORBA_TypeCode__release' USING
     BY VALUE TYPE-CODE-PTR
     BY REFERENCE CORBA-ENVIRONMENT
   RETURNING ERR-CODE.
* 例外チェック
 IF NOT CORBA-NO-EXCEPTION THEN
     CALL 'EXCEPTION-HANDLER' USING
              BY REFERENCE MAJOR OF CORBA-ENVIRONMENT
              BY REFERENCE CORBA-ENVIRONMENT
     EXIT PROGRAM
 END-IF.

 END PROGRAM 'call'.


*****************************************************
* Constructor of 'CBLClass_TSCimpl'
*****************************************************
* Constructor of OTM Object
* Implement.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 MY-WATCH-TIME-PTR        USAGE POINTER.
 01 MY-WATCH-TIME            PIC S9(9) COMP.
 01 CORBA-ENVIRONMENT.
  02 MAJOR                  PIC 9(9) COMP.
   88 CORBA-NO-EXCEPTION         VALUE 0.
   88 CORBA-USER-EXCEPTION       VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION     VALUE 2.
  02 EXCEP                  USAGE POINTER.
  02 FUNC-NAME              PIC X(256).
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING SKELETON-POINTER.

* Write user own code, if necessary.

* 戻り値の初期化
     SET SKELETON-POINTER TO NULL.

* 時間監視60秒の時間監視オブジェクト生成
```

```
      MOVE 60 TO MY-WATCH-TIME.
      CALL 'TSCWatchTime-NEW' USING
              BY VALUE MY-WATCH-TIME
              BY REFERENCE CORBA-ENVIRONMENT
          RETURNING MY-WATCH-TIME-PTR.
*  例外チェック
      IF NOT CORBA-NO-EXCEPTION THEN
          CALL 'OTM-EXCEPTION-HANDLER' USING
                  BY REFERENCE MAJOR
                  BY REFERENCE CORBA-ENVIRONMENT
          CALL 'TSCSysExcept-DELETE' USING
              BY VALUE EXCEP OF CORBA-ENVIRONMENT
          GO TO PROG-END
      END-IF.

*  時間監視の開始
      CALL 'TSCWatchTime-start' USING
              BY VALUE MY-WATCH-TIME-PTR
              BY REFERENCE CORBA-ENVIRONMENT.
*  例外チェック
      IF NOT CORBA-NO-EXCEPTION THEN
          CALL 'OTM-EXCEPTION-HANDLER' USING
                  BY REFERENCE MAJOR
                  BY REFERENCE CORBA-ENVIRONMENT
          CALL 'TSCSysExcept-DELETE' USING
              BY VALUE EXCEP OF CORBA-ENVIRONMENT
          GO TO PROG-END
      END-IF.

*  必要に応じてユーザ独自のコードを追加できます。
*  コンストラクタの引数の数および型を変更することもできます。

*  時間監視の中断
      CALL 'TSCWatchTime-stop' USING
              BY VALUE MY-WATCH-TIME-PTR
              BY REFERENCE CORBA-ENVIRONMENT.
*  例外チェック
      IF NOT CORBA-NO-EXCEPTION THEN
          CALL 'OTM-EXCEPTION-HANDLER' USING
                  BY REFERENCE MAJOR
                  BY REFERENCE CORBA-ENVIRONMENT
          CALL 'TSCSysExcept-DELETE' USING
              BY VALUE EXCEP OF CORBA-ENVIRONMENT
          GO TO PROG-END
      END-IF.

*  This sub-program must return a pointer
*  that 'CBLClass_TSCsk-NEW' sub-program returns.
      CALL 'CBLClass_TSCsk-NEW'
          RETURNING SKELETON-POINTER.

 PROG-END.

 END PROGRAM 'CBLClass_TSCimpl-NEW'.

*****************************************************
*  Destructor of 'CBLClass_TSCimpl'
*****************************************************
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCimpl-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 SKELETON-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE SKELETON-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。

* This sub-program must call
* 'CBLClass_TSCsk-DEL' sub-program.
     CALL 'CBLClass_TSCsk-DEL' USING
         BY VALUE SKELETON-POINTER.
 END PROGRAM 'CBLClass_TSCimpl-DEL'.

*****************************************************
* Constructor of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-NEW'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
* You can change signature of this sub-program.
 PROCEDURE DIVISION
     RETURNING FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。
* 引数の数および型を変更することもできます。

* This sub-program must return a pointer that
* 'CBLClass_TSCfact-get' sub-program returns.
     CALL 'CBLClass_TSCfact-get'
         RETURNING FACTORY-POINTER.
 END PROGRAM 'CBLClass_TSCfact-NEW'.


*****************************************************
* Destructor of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-DEL'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
```

```
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER.

* Write user own code, if necessary.
* 必要に応じてユーザ独自のコードを追加できます。

* This sub-program must call
* 'CBLClass_TSCfact-rls'.sub-program.
     CALL 'CBLClass_TSCfact-rls' USING
         BY VALUE FACTORY-POINTER.
 END PROGRAM 'CBLClass_TSCfact-DEL'.


*****************************************************
* 'create' method of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-crt'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
* Do not change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER
     RETURNING OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを生成するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-NEW' returns.
     CALL 'CBLClass_TSCimpl-NEW'
         RETURNING OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-crt'.


*****************************************************
* 'destroy' method of CBLClass_TSCfact
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'CBLClass_TSCfact-dst'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 FACTORY-POINTER USAGE POINTER.
 01 OBJECT-POINTER USAGE POINTER.
```

```
* Do not change signature of this sub-program.
 PROCEDURE DIVISION USING
         BY VALUE FACTORY-POINTER
         BY VALUE OBJECT-POINTER.

* Write user own code, if necessary.
* サーバオブジェクトを削除するコードを記述します。
* 必要に応じて変更してください。

* This sub-program must return pointer that
* 'CBLClass_TSCimpl-DEL' returns.
     CALL 'CBLClass_TSCimpl-DEL' USING
         BY VALUE OBJECT-POINTER.
 END PROGRAM 'CBLClass_TSCfact-dst'.


*****************************************************
* TSCCBLThread-beginThread of
*     TSCCBLThread
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-beginThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 BEGIN-THREAD-PTR USAGE POINTER.
 01 END-THREAD-PTR USAGE POINTER.
* Do not change signature of this sub-program.
 LINKAGE SECTION.
 01 THREAD-FACTORY-ID PIC S9(9) COMP.
 PROCEDURE DIVISION USING
         BY VALUE THREAD-FACTORY-ID.

* Write user own code.
* スレッド開始処理を記述します。
* 必要に応じて変更してください。

 END PROGRAM 'TSCCBLThread-beginThread'.


*****************************************************
* TSCCBLThread-endThread of
*     TSCCBLThreadFactory
*****************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'TSCCBLThread-endThread'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* Do not change signature of this sub-program.
 LINKAGE SECTION.
 01 THREAD-FACTORY-ID PIC S9(9) COMP.
 PROCEDURE DIVISION USING
         BY VALUE THREAD-FACTORY-ID.
```

```
* Write user own code.
* スレッド終了処理を記述します。
* 必要に応じて変更してください。

 END PROGRAM 'TSCCBLThread-endThread'.
```

(2) サービス登録処理の流れ・コード

同期型呼び出しの場合と同様です。「6.2.2(2)　サービス登録処理の流れ」,「6.2.2(3)　サービス登録処理のコード」を参照してください。

7 アプリケーションプログラミングインタフェース（COBOL）

この章では，COBOL で使用する副プログラムについて説明します。
ユーザ定義 IDL インタフェース依存クラスと雛形クラスでは，対応するユーザ定義 IDL インタフェースの名称を "ABC" と仮定します。なお，各クラスの詳細は，アルファベット順に示します。

COBOL85 インタフェース

OTM の各機能はオブジェクト指向プログラミングのクラスとして実現されています。そのため，C++ および Java では，各機能を使用するアプリケーションプログラミングインタフェースはクラスのメソッドとして提供されます。一方，COBOL85 インタフェースでは，アプリケーションプログラミングインタフェースは副プログラムとして提供されます。

副プログラムによるクラスの使用

C++ または Java では，「TSCxxx クラスの yyy メソッド」の形式で機能が提供されます。COBOL85 インタフェースでは，多くの場合，「TSCxxx-yyy 副プログラム」の形式で機能が提供されます。幾つかの機能は，「TSCaaa-bbb 副プログラム」の形式で提供されます。この場合，aaa および bbb は，xxx および yyy と異なる文字列です。

TSCxxx クラスのインスタンスを生成する副プログラムとして TSCxxx-NEW が提供されます。また，インスタンスを削除する副プログラムとして TSCxxx-DELETE が提供されます。

TSCxxx-yyy の第 1 引数に TSCxxx-NEW の戻り値を指定することによって，該当するインスタンスに対するメソッド呼び出しと同じ働きをします。

例外処理

副プログラム実行中に例外が発生したかどうかを検知するために，呼び出し元では**例外情報集団項目**を利用します。例外情報集団項目は，各副プログラムの最終引数です。例外情報集団項目を利用することで，ユーザ定義 IDL インタフェースのオペレーション呼び出し時にユーザ例外が発生したかどうかも検知できます。

ユーザ例外を発生しないオペレーションを呼び出した場合でも，システム例外が発生することがあります。そのため，例外情報集団項目によって例外の有無を必ず確認してください。詳細は，この章の「TSCSystemException（COBOL）」を参照してください。

文字列の扱い

COBOL85 インタフェースとアプリケーションプログラムとのインタフェースで使用される文字列は，16 進表示の $(00)_{16}$ で表される文字で終了する文字列です。そこで，COBOL adapter for TPBroker で提供される文字列操作ラッパー関数を使用して，次のように処理する必要があります。

- in 属性の引数は，COBOL の文字列を 16 進表示の $(00)_{16}$ で表される文字で終了する文字列に変換します。
- out 属性の引数または戻り値は，16 進表示の $(00)_{16}$ で表される文字で終了する文字列を，COBOL の文字列に変換します。

ユーザが COBOL adapter for TPBroker の文字列操作ラッパー関数を使用して得た，16進表示の $(00)_{16}$ で表される文字で終了する文字列の領域は，特に断りがないかぎり，ユーザの責任で解放してください。また，ユーザが確保したのではない領域は，特に断りがないかぎりユーザが解放してはいけません。ただし，ユーザオペレーション引数のメモリ管理は，以降に示す規則に従ってください。なお，文字列領域の解放には COBOL adapter for TPBroker のラッパー関数を使用してください。文字列操作ラッパー関数の使用方法については，マニュアル「COBOL adapter for TPBroker　ユーザーズガイド」を参照してください。

ユーザオペレーション引数のメモリ管理規則

OTM の COBOL85 インタフェースでは，ユーザ定義 IDL インタフェースのオペレーション引数のうち，可変長の引数（string 型および sequence 型）は，COBOL adapter for TPBroker のラッパー関数を使用して領域を取得および解放する必要があります。領域の取得および解放の規則を次の表に示します。

表 7-1　領域の取得および解放（呼び出し元）

領域	取得者	解放者
in 属性	ユーザ	ユーザ
out 属性	OTM	ユーザ
inout 属性の入力引数	ユーザ	OTM（必要に応じて）
inout 属性の出力引数	OTM（必要に応じて）	ユーザ
戻り値	OTM	ユーザ

表 7-2　領域の取得および解放（呼び出し先）

領域	取得者	解放者
in 属性	OTM	OTM
out 属性	ユーザ	OTM
inout 属性の入力引数	OTM	ユーザ（必要に応じて）
inout 属性の出力引数	ユーザ（必要に応じて）	OTM
戻り値	ユーザ	OTM

クラスの一覧（COBOL）

各クラスは次のように分類されます。ただし，各クラスの機能は副プログラムとして提供されます。

- システム提供クラス
- システム提供例外クラス
- ユーザ定義 IDL インタフェース依存クラス
- 雛形クラス

各クラスの一覧を次の表に示します。

表 7-3　クラス一覧（COBOL）

分類	クラス
システム提供クラス	• TSCAcceptor • TSCAdm • TSCClient • TSCContext • TSCDomain • TSCObject • TSCProxyObject • TSCRootAcceptor • TSCServer • TSCSessionProxy • TSCCBLThread • TSCCBLThreadFactory • TSCWatchTime
システム提供例外クラス	• TSCSystemException
ユーザ定義 IDL インタフェース依存クラス（基底クラス）	• ABC_TSCacpt（TSCAcceptor） • ABC_TSCprxy（TSCProxyObject） • ABC_TSCsk（TSCObject） • ABC_TSCspxy（TSCSessionProxy）
雛形クラス（基底クラス）	• ABC_TSCfactimpl（ABC_TSCfactsk） • ABC_TSCimpl（ABC_TSCsk）

基本データ型（COBOL）

OTM の各クラスで使用する基本データ型は，COBOL が提供する標準型です。

説明の形式

以降では，各副プログラムをクラスごとに説明します。また，副プログラムの引数が，値を入力する項目なのか，値が出力される項目なのかの区別を示します。

ABC_TSCacpt（COBOL）

ABC_TSCacpt はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCacpt は，TSC ユーザアクセプタの実装クラスです。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCacpt を自動生成します。

各副プログラム名の "_TSCacpt" の部分は，tscidl2cbl コマンドの -TSCacpt_ext オプションによって変更できます。省略した場合，"_TSCacpt" が設定されます。

形式

```
CALL 'ABC_TSCacpt-NEW' USING
            BY VALUE     FACTORY-PTR
            BY VALUE     ACCEPTOR-NAME
            BY REFERENCE CORBA-ENVIRONMENT
         RETURNING       ACCEPTOR-PTR.

CALL 'ABC_TSCacpt-DEL' USING
            BY VALUE     ACCEPTOR-PTR
            BY REFERENCE CORBA-ENVIRONMENT.
```

副プログラム

● CALL 'ABC_TSCacpt-NEW' USING
BY VALUE FACTORY-PTR
BY VALUE ACCEPTOR-NAME
BY REFERENCE CORBA-ENVIRONMENT
RETURNING ACCEPTOR-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE FACTORY-PTR USAGE POINTER	(入力)TSCObjectFactory のポインタ
	BY VALUE ACCEPTOR-NAME USAGE POINTER	(入力)TSC アクセプタ名称のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ACCEPTOR-PTR USAGE POINTER	TSC ユーザアクセプタのポインタ

FACTORY-PTR で指定された TSCObjectFactory を持つ，TSC アクセプタ名称が ACCEPTOR-NAME の ABC_TSCacpt を生成します。ただし，ACCEPTOR-NAME には，1 ～ 31 文字の TSC アクセプタ名称を指定してください。ACCEPTOR-NAME が NULL の場合，ACCEPTOR-NAME は無視されます。

● CALL 'ABC_TSCacpt-DEL' USING
BY VALUE ACCEPTOR-PTR
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE ACCEPTOR-PTR USAGE POINTER	(入力)ABC_TSCacpt のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	

ABC_TSCacpt を削除します。

ABC_TSCfactimpl（COBOL）

ABC_TSCfactimpl は雛形クラスです。

ABC_TSCfactimpl は，TSC ユーザオブジェクトファクトリを実現するためのクラスです。実際の副プログラムのプリフィックスは "ABC_TSCfact" となります。各副プログラム名の "_TSCfact" の部分は，tscidl2cbl コマンドの -TSCfact_ext オプションによって変更できます。省略した場合，"_TSCfact" が設定されます。

形式

*斜体*で示している部分は，ユーザが実装のコードを記述する必要がある副プログラムです。**太字**で示している部分は，引数の型および数を変更できる副プログラムで，ユーザが実装のコードを記述する必要があります。

```
CALL 'ABC_TSCfact-NEW'
            ...
             RETURNING           FACTORY-PTR.

CALL 'ABC_TSCfact-crt' USING
                  BY VALUE       FACTORY-PTR
             RETURNING           OBJECT-PTR.
CALL 'ABC_TSCfact-dst' USING
                  BY VALUE       FACTORY-PTR
                  BY VALUE       SK-PTR.

CALL 'ABC_TSCfact-get' USING
             RETURNING           FACTORY-PTR.

CALL 'ABC_TSCfact-rls' USING
                  BY VALUE       FACTORY-PTR.

CALL 'ABC_TSCfact-DEL' USING
            BY VALUE FACTORY-PTR
             ....
```

副プログラム

● CALL 'ABC_TSCfact-NEW'

```
...
  RETURNING   FACTORY-PTR.
```

項目	型・意味	
戻り値	FACTORY-PTR USAGE POINTER	ABC_TSCfact-get の戻り値

ABC_TSCfact を作成します。引数の型や数を含めて，ユーザがコードを記述する必要があります。

名称を変更して複数の副プログラムを作成できますが，どの副プログラムの戻り値にも，

次に示す副プログラムを実行して得られる戻り値を設定する必要があります。

```
CALL 'ABC_TSCfact-get' USING
         RETURNING FACTORY-PTR.
```

● CALL 'ABC_TSCfact-crt' USING

BY VALUE FACTORY-PTR
RETURNING OBJECT-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE FACTORY-PTR USAGE POINTER	(入力)ABC_TSCfact-NEW の戻り値
戻り値	OBJECT-PTR USAGE POINTER	ABC_TSCimpl-NEW の戻り値

ABC_TSCimpl を作成するための副プログラムです。典型的なコードを生成するため，ユーザは必要に応じて変更してください。

この副プログラムの戻り値には，次に示す副プログラムまたは同等のユーザ実装副プログラムを実行して得られる戻り値を設定する必要があります。

```
CALL 'ABC_TSCimpl-NEW' ...
         RETURNING OBJECT-PTR.
```

● CALL 'ABC_TSCfact-dst' USING

BY VALUE FACTORY-PTR
BY VALUE SK-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE FACTORY-PTR USAGE POINTER	(入力)ABC_TSCfact-NEW の戻り値
	BY VALUE SK-PTR USAGE POINTER	(入力)ABC_TSCimpl-NEW の戻り値

ABC_TSCimpl を削除するための副プログラムです。典型的なコードを生成するため，ユーザは必要に応じて変更してください。

この副プログラムでは，次の副プログラムを実行する必要があります。

```
CALL 'ABC_TSCimpl-DEL' USING
             BY VALUE SK-PTR ... .
```

● CALL 'ABC_TSCfact-get'

RETURNING FACTORY-PTR.

項目	型・意味	
戻り値	FACTORY-PTR	OTM 内部のファクトリのポインタ

ユーザ定義 IDL インタフェース依存クラスの ABC_TSCfact と OTM を関連づけて，OTM 内部のファクトリのポインタを返します。この副プログラムは必ず ABC_TSCfact-NEW から呼び出して，その戻り値を ABC_TSCfact-NEW の戻り値に設定してください。この副プログラムはユーザが変更してはいけません。

● CALL 'ABC_TSCfact-rls'
　　BY VALUE　FACTORY-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE　FACTORY-PTR　USAGE POINTER	(入力)ABC_TSCfact のポインタ
戻り値	ありません。	

ユーザ定義 IDL インタフェース依存クラスの ABC_TSCfact と OTM の関連づけを解除します。この副プログラムは必ず ABC_TSCfact-DEL から呼び出してください。この副プログラムは変更できません。

● CALL 'ABC_TSCfact-DEL' USING
　　BY VALUE FACTORY-PTR
　　....

項目	型・(入出力の区別) 意味	
引数	BY VALUE　FACTORY-PTR　USAGE POINTER	(入力)ABC_TSCfact-NEW の戻り値
戻り値	ありません。	

ABC_TSCfact を削除します。引数の型や数を含めて，ユーザがコードを記述する必要があります。名称を変更して複数の副プログラムを作成できますが，どの副プログラムも，次のように発行する必要があります。

```
CALL 'ABC_TSCfact-rls' USING
            BY VALUE FACTORY-PTR.
```

ABC_TSCimpl（COBOL）

ABC_TSCimpl は雛形クラスです。

ABC_TSCimpl は，TSC ユーザオブジェクトを実現するためのクラスです。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCimpl を生成します。ユーザはこの雛形クラスに処理依存のコードを記述します。

各副プログラム名の "_TSCimpl" の部分は，tscidl2cbl コマンドの -TSCimpl_ext オプションによって変更できます。省略した場合，"_TSCimpl" が設定されます。

形式

*斜体*で示している部分は，ユーザが実装のコードを記述する必要がある副プログラムです。**太字**で示している部分は，引数の型および数を変更できる副プログラムで，ユーザが実装のコードを記述する必要があります。

```
 CALL 'ABC_TSCimpl-NEW' USING
                ...
          RETURNING       SK-PTR.
 CALL 'ABC_TSCimpl-DEL' USING
              BY VALUE       SK-PTR
              ....

*ユーザ定義IDLインタフェース依存の副プログラム群
*形式1
 CALL 'xxx' USING
                ...
          (RETURNING       ...).

*形式2
 CALL 'xxx' USING
              BY VALUE       SK-PTR
                ...
          (RETURNING       ...).
```

副プログラム

● CALL 'ABC_TSCimpl-NEW' USING

...
RETURNING SK-PTR.

項目	型・意味	
戻り値	SK-PTR USAGE POINTER	ABC_TSCsk-NEW の戻り値

TSC ユーザオブジェクトを生成します。引数の型や数を含めて，ユーザがコードを記述する必要があります。この副プログラムは ABC_TSCfact-create 副プログラムから呼ば

れます。名称を変更して複数の副プログラムを作成できます。

この副プログラムの戻り値には，次に示す副プログラムを実行して得られる戻り値を設定する必要があります。

```
CALL 'ABC_TSCsk-NEW'
          RETURNING SK-PTR.
```

● CALL 'ABC_TSCimpl-DEL' USING
　　BY VALUE　SK-PTR
　...

項目	型・(入出力の区別) 意味	
引数	BY VALUE　SK-PTR　USAGE POINTER	(入力)ABC_TSCsk-NEW の戻り値
戻り値	ありません。	

TSC ユーザオブジェクトを削除します。引数の型や数を含めて，ユーザがコードを記述する必要があります。この副プログラムは ABC_TSCfact-destroy 副プログラムから呼ばれます。

名称を変更して複数の副プログラムを作成できますが，どの副プログラムも，次のように発行する必要があります。

```
CALL 'ABC_TSCsk-DEL' USING
            BY VALUE SK-PTR.
```

ユーザが実装する副プログラム

ユーザ定義 IDL インタフェースのオペレーションに相当する副プログラムを，ユーザが実装する必要があります。インタフェースには次の二つがあります。使用する形式は，トランザクションフレームジェネレータ実行時に tscidl2cbl コマンドの -format オプションで選択できます。

形式 1

```
CALL 'xxx' USING
         ...
      (RETURNING)    ....
```

ユーザが実装するオペレーションです。この形式では，引数および戻り値はユーザ定義 IDL インタフェースのオペレーションと一致します。

形式 2

```
CALL 'xxx' USING
         BY VALUE    OBJECT-PTR
         ...
      (RETURNING)    ....
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE OBJECT-PTR USAGE POINTER	(入力) 対応する TSCObject のポインタ

ユーザが実装するオペレーションです。この形式では，ユーザ定義 IDL インタフェースのオペレーションにある引数の前に，TSCObject のポインタが渡されます。
TSCContext または TSCThread などの機能を使用する場合，この形式を使用してください。

ABC_TSCprxy（COBOL）

ABC_TSCprxy はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCprxy は，TSC ユーザプロキシの実装クラスです。ユーザ定義 IDL インタフェースに従って，ユーザデータをバイト配列データに変換し，TSCProxyObject を呼び出します。なお，ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCprxy を自動生成します。

各副プログラム名の "_TSCprxy" の部分は，tscidl2cbl コマンドの -TSCprxy_ext オプションによって変更できます。省略した場合，"_TSCprxy" が設定されます。

ユーザ定義 IDL インタフェースのマッピング

ユーザ定義 IDL インタフェース内に定義されたオペレーションの COBOL 言語へのマッピングは，TPBroker と同様です。

形式

```
 CALL 'ABC_TSCprxy-NEW' USING
             BY VALUE        CLIENT-PTR
             BY VALUE        ACCEPTOR-NAME
             BY REFERENCE    CORBA-ENVIRONMENT
          RETURNING       PROXY-HANDLE.

*ユーザ定義IDLインタフェース依存の副プログラム群
 CALL 'ABC-xxx' USING
             BY VALUE        PROXY-HANDLE
             ...
             BY REFERENCE    CORBA-ENVIRONMENT
           (RETURNING ...) .

 CALL 'ABC_TSCprxy-DEL' USING
             BY VALUE        PROXY-HANDLE
             BY REFERENCE    CORBA-ENVIRONMENT.
```

副プログラム

● CALL 'ABC_TSCprxy-NEW' USING

```
      BY VALUE        CLIENT-PTR
      BY VALUE        ACCEPTOR-NAME
      BY REFERENCE     CORBA-ENVIRONMENT
    RETURNING     PROXY-HANDLE.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE CLIENT-PTR USAGE POINTER	(入力) 接続する TSCClient のポインタ
	BY VALUE ACCEPTOR-NAME USAGE-POINTER	(入力)TSC アクセプタ名称のポインタ

項目	型・(入出力の区別) 意味	
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	PROXY-HANDLE USAGE-POINTER	ABC_TSCprxy のポインタ

CLIENT-PTR で示される TSCClient と接続する ABC_TSCprxy を生成します。この ABC_TSCprxy の TSC アクセプタ名称は ACCEPTOR-NAME です。
ACCEPTOR-NAME が NULL の場合は，ACCEPTOR-NAME は無視されます。

ユーザ定義 IDL インタフェース依存副プログラム

● CALL 'ABC-xxx' USING
BY VALUE PROXY-HANDLE
...
BY REFERENCE CORBA-ENVIRONMENT
(RETURNING ...).

項目	型・(入出力の区別) 意味	
引数	BY VALUE PROXY-HANDLE USAGE POINTER	(入力)ABC_TSCprxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ユーザ定義 IDL インタフェースに依存します。	
例外	TSCSystemException（各種例外） ユーザ定義 IDL インタフェース中の raises 句に定義されたユーザ例外	

ユーザ定義 IDL インタフェースで定義された，ABC インタフェース内の xxx オペレーションを呼び出します。PROXY-HANDLE と CORBA-ENVIRONMENT 以外の引数，および戻り値はユーザ定義 IDL インタフェースに依存します。

● CALL 'ABC_TSCprxy-DEL' USING
BY VALUE PROXY-HANDLE
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE PROXY-HANDLE USAGE POINTER	(入力)ABC_TSCprxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	

ABC_TSCprxy を削除します。

ABC_TSCsk（COBOL）

ABC_TSCsk はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCsk は，TSC ユーザスケルトンの実装クラスです。ユーザのオブジェクトインプリメントを作成する副プログラム（ABC_TSCimpl-NEW），または削除する副プログラム（ABC_TSCimpl-DEL）では，必ず TSC ユーザスケルトンの作成（ABC_TSCsk-NEW）または削除（ABC_TSCsk-DEL）を行ってください。

各副プログラム名の "_TSCsk" の部分は，tscidl2cbl コマンドの -TSCsk_ext オプションによって変更できます。省略した場合，"_TSCsk" が設定されます。

形式

```
CALL 'ABC_TSCsk-NEW' USING
         RETURNING      SK-PTR.

CALL 'ABC_TSCsk-DEL' USING
            BY VALUE      SK-PTR.
```

副プログラム

● CALL 'ABC_TSCsk-NEW'
　RETURNING　SK-PTR.

項目	型・意味	
戻り値	SK-PTR USAGE POINTER	ABC_TSCsk のポインタ

ABC_TSCsk を生成します。

● CALL 'ABC_TSCsk-DEL' USING
　BY VALUE　SK-PTR.

項目	型・意味
戻り値	ありません。

ABC_TSCsk を削除します。

ABC_TSCspxy（COBOL）

ABC_TSCspxy はユーザ定義 IDL インタフェース依存クラスです。

ABC_TSCspxy は，セション呼び出し用の TSC ユーザプロキシの実装クラスです。ユーザ定義 IDL インタフェースに従って，ユーザデータをバイト配列データに変換し，TSCSessionProxy を呼び出します。ユーザ定義 IDL インタフェースに従って，トランザクションフレームジェネレータが ABC_TSCspxy を自動生成します。ABC_TSCspxy は，次の点を除いて ABC_TSCprxy と同様の働きをします。

- TSCSessionProxy を継承します。
- トランザクションフレームジェネレータに -TSCspxy オプションを指定したときだけ生成されます。
- oneway のオペレーションを定義したユーザ定義 IDL からは生成できません。

各副プログラム名の "_TSCspxy" の部分は，tscidl2cbl コマンドの -TSCspxy_ext オプションによって変更できます。省略した場合，"_TSCspxy" が設定されます。

ユーザ定義 IDL インタフェースのマッピング

ユーザ定義 IDL インタフェース内に定義されたオペレーションの COBOL 言語へのマッピングは，TPBroker と同じです。

形式

```
 CALL 'ABC_TSCspxy-NEW' USING
              BY VALUE        CLIENT-PTR
              BY VALUE        ACCEPTOR-NAME
              BY REFERENCE    CORBA-ENVIRONMENT
           RETURNING       SPROXY-HANDLE.

*ユーザ定義IDLインタフェース依存の副プログラム群
 CALL 'ABC-xxx' USING
              BY VALUE        SPROXY-HANDLE
              ...
              BY REFERENCE    CORBA-ENVIRONMENT
            (RETURNING ...) .

 CALL 'ABC_TSCspxy-DEL' USING
              BY VALUE        SPROXY-HANDLE
              BY REFERENCE    CORBA-ENVIRONMENT.
```

副プログラム

● CALL 'ABC_TSCspxy-NEW' USING

BY VALUE　　CLIENT-PTR
BY VALUE　　ACCEPTOR-NAME
BY REFERENCE　　CORBA-ENVIRONMENT
RETURNING　　SPROXY-HANDLE.

項目	型・(入出力の区別) 意味	
引数	BY VALUE CLIENT-PTR USAGE POINTER	(入力) 接続する TSCClient のポインタ
	BY VALUE ACCEPTOR-NAME USAGE-POINTER	(入力)TSC アクセプタ名称のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	SPROXY-HANDLE USAGE-POINTER	ABC_TSCspxy のポインタ

CLIENT-PTR で示される TSCClient と接続する ABC_TSCspxy を生成します。この ABC_TSCspxy の TSC アクセプタ名称は ACCEPTOR-NAME です。
ACCEPTOR-NAME が NULL の場合は，ACCEPTOR-NAME は無視されます。

ユーザ定義 IDL インタフェース依存副プログラム

● CALL 'ABC-xxx' USING

```
BY VALUE        SPROXY-HANDLE
...
BY REFERENCE    CORBA-ENVIRONMENT
(RETURNING ...).
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE SPROXY-HANDLE USAGE POINTER	(入力)ABC_TSCspxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ユーザ定義 IDL インタフェースに依存します。	
例外	TSCSystemException（各種例外） ユーザ定義 IDL インタフェース中の raises に定義されたユーザ例外	

ユーザ定義 IDL インタフェースで定義された，ABC インタフェース内の xxx オペレーションを呼び出します。SPROXY-HANDLE と CORBA-ENVIRONMENT 以外の引数，および戻り値はユーザ定義 IDL インタフェースに依存します。

● CALL 'ABC_TSCspxy-DEL' USING

```
BY VALUE        SPROXY-HANDLE
BY REFERENCE    CORBA-ENVIRONMENT.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE SPROXY-HANDLE USAGE POINTER	(入力)ABC_TSCprxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	

ABC_TSCspxy を削除します。

TSCAcceptor（COBOL）

TSCAcceptor はシステム提供クラスです。

TSCAcceptor は ABC_TSCacpt の基底クラスです。各副プログラムは ABC_TSCacpt のインスタンスに対して発行します。したがって，各副プログラムの第 1 引数（ACCEPTOR-PTR）には，ユーザ定義 IDL インタフェース依存クラス（ABC_TSCacpt）のポインタを指定してください。

TSCAcceptor は，TSC ユーザプロキシを使用したクライアント側からの TSC ユーザオブジェクトの副プログラム呼び出し要求に対して，サーバ側の TSC ユーザオブジェクトの副プログラムを呼び出すためのクラスです。TSC ルートアクセプタから TSC ユーザオブジェクトの副プログラム呼び出し要求を受け取り，TSC ユーザオブジェクトの副プログラムを呼び出します。これに伴って，TSC ユーザオブジェクトを管理したり，スレッドと TSC ユーザオブジェクト間を対応づけたりします。また，TSC サービス識別子を使用して，TSC ユーザオブジェクトが提供するサービスを識別します。

ユーザは TSC ユーザオブジェクトの管理オブジェクトとして，TSCAcceptor クラスのインスタンスを生成します。次に TSCAcceptor の特徴を示します。

- TSCObjectFactory を保持することで，TSC ユーザオブジェクト（TSCObject）を管理します。
- TSCAcceptor が提供できるサービスを TSC **サービス識別子**の列で表現します。TSC サービス識別子は，**インタフェース名称**の列と TSC **アクセプタ名称**から構成されます。ただし，TSC アクセプタ名称がない場合もあります。

TSC ユーザオブジェクト（TSCObject）の管理

TSCObjectFactory による TSCObject の管理

TSCRootAcceptor が active 状態に遷移するとき，登録されている TSCAcceptor はオブジェクト管理開始通知を受けます。また，TSCRootAcceptor が non-active 状態に遷移するとき，登録されている TSCAcceptor はオブジェクト管理終了通知を受けます。それぞれの通知とともに，TSCAcceptor は TSCObjectFactory を使用して次に示すように動作します。

- **TSCRootAcceptor からのオブジェクト管理開始通知**
  TSCAcceptor は，TSCRootAcceptor からオブジェクト管理開始通知を受けると，TSCRootAcceptor が保持する各スレッド上で，TSCObjectFactory の create を呼び出します。さらに，この呼び出しで返される TSCObject を TSCRootAcceptor が保持する各スレッドに割り当てます。これによって，TSCObjectFactory の create 呼び出しで返される TSCObject は，スレッドに対応づけて管理されます。
- **TSCRootAcceptor からのオブジェクト管理終了通知**

TSCAcceptor は，TSCRootAcceptor からオブジェクト管理終了通知を受けると，TSCRootAcceptor が保持する各スレッド上で，割り当てられている TSCObject を引数に，TSCObjectFactory の destroy を呼び出します。これによって，対応づけられているスレッド上の管理対象から，該当する TSCObject が外されます。

TSCRootAcceptor からの TSCObject の呼び出し

TSCRootAcceptor は，クライアント側からの TSC ユーザオブジェクト呼び出しを受け取ると，該当するサービスを提供する TSCAcceptor に振り分けます。さらに，TSCAcceptor は，TSCRootAcceptor が管理するスレッド上で TSC ユーザオブジェクト呼び出し要求を受け取ると，同じスレッド上で同じスレッドに割り当てられている TSCObject を呼び出します。

TSC サービス識別子によるサービスの識別

TSC サービス識別子の構成

TSCAcceptor が提供できるサービスの種類は，TSC サービス識別子によって表されます。TSC サービス識別子は，インタフェース名称の列と TSC アクセプタ名称から構成されます。

- **TSCAcceptor のインタフェース名称の列**
  TSCAcceptor のインタフェース名称の列は，TSCAcceptor や管理する TSCObject が提供するサービスのインタフェースの種類を表します。
  TSCObject が単数のインタフェースをサポートする場合，TSCAcceptor や管理する TSCObject が提供するインタフェースの種類は，単数のインタフェース名称で表されます。TSCObject が複数のインタフェースをサポートする場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，複数のインタフェース名称が列で表されます。
- **TSCAcceptor の TSC アクセプタ名称**
  TSC アクセプタ名称も，TSCAcceptor や管理する TSCObject が提供するサービスのインタフェースの種類を表します。ただし，同じインタフェースを提供する TSCAcceptor または TSCObject の間の実装内容の違いを識別するために使用します。したがって，TSC アクセプタ名称を設定しないで，"TSC アクセプタ名称なし " とすることもできます。

サービスの種類の表現方法

TSCAcceptor が提供できるサービスの種類は，TSC サービス識別子の列によって表されます。TSCAcceptor は TSC サービス識別子の列で示されるサービスを提供できます。

- **TSCAcceptor に TSC アクセプタ名称が設定されていない場合**
  TSCObject が単数のインタフェースをサポートする場合，TSCAcceptor や管理する TSCObject が提供できるサービスの種類は，次のように表されます。

単数のTSCアクセプタ名称なしTSCサービス識別子

TSCObject が複数のインタフェースをサポートする場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，TSC サービス識別子の列で次のように表されます。

複数のTSCアクセプタ名称なしTSCサービス識別子

- **TSCAcceptor に TSC アクセプタ名称が設定されている場合**
  TSCObject が単数のインタフェースをサポートする場合，TSCAcceptor や管理する TSCObject が提供できるサービスの種類は，次のように表されます。

  単数のTSCアクセプタ名称なしTSCサービス識別子，および
  単数のTSCアクセプタ名称ありTSCサービス識別子

  TSCObject が複数のインタフェースをサポートする場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，TSC サービス識別子の列で次のように表されます。

  複数のTSCアクセプタ名称なしTSCサービス識別子，および
  複数のTSCアクセプタ名称ありTSCサービス識別子（ただし，TSCサービス識別子中のTSCアクセプタ名称は同じ）

サービスの種類の表現例

- **TSCAcceptor のインタフェース名称が "ABC" で，TSC アクセプタ名称がない場合**
  次のように表される場合，TSCAcceptor は，"ABC::" への要求だけを受け付けることができます。

  ```
  "ABC::"
  ```

- **TSCAcceptor のインタフェース名称が "ABC" で，TSC アクセプタ名称が "abc" の場合**
  次のように表される場合，TSCAcceptor は，"ABC::" と "ABC::abc" への要求メッセージを受け付けることができます。

  ```
  "ABC::abc"
  "ABC::"
  ```

形式

```
CALL 'TSCAcceptor-getInterfaceName USING
       BY VALUE      ACCEPTOR-PTR
       BY REFERENCE  CORBA-ENVIRONMENT
     RETURNING       INTERFACE-NAME.

CALL 'TSCAcceptor-getAcceptorName USING
       BY VALUE      ACCEPTOR-PTR
       BY REFERENCE  CORBA-ENVIRONMENT
     RETURNING       ACCEPTOR-NAME.
```

副プログラム

● CALL 'TSCAcceptor-getInterfaceName' USING
　　BY VALUE　ACCEPTOR-PTR

```
    BY REFERENCE  CORBA-ENVIRONMENT
RETURNING       INTERFACE-NAME.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE ACCEPTOR-PTR USAGE POINTER	(入力)ABC_TSCAcpt のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	INTERFACE-NAME USAGE POINTER	インタフェース名称
例外	TSCBadParamException TSCNoPermissionException	

インタフェース名称を取得します。

インタフェース名称のメモリ領域の管理責任は TSCAcceptor クラスにあるので，ユーザは解放しないでください。

なお，この副プログラムを複数のスレッド上で同時に呼び出すことができます。

● CALL 'TSCAcceptor-getAcceptorName' USING

```
    BY VALUE      ACCEPTOR-PTR
    BY REFERENCE  CORBA-ENVIRONMENT
RETURNING       ACCEPTOR-NAME
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE ACCEPTOR-PTR USAGE POINTER	(入力)ABC_TSCAcpt のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ACCEPTOR-NAME USAGE POINTER	TSC アクセプタ名称
例外	TSCBadParamException	

TSC アクセプタ名称を取得します。

TSC アクセプタ名称のメモリ領域の管理責任は TSCAcceptor クラスにあるので，ユーザは解放しないでください。

なお，この副プログラムを複数のスレッド上で同時に呼び出すことができます。

TSCAcceptor の生成と削除

TSCAcceptor は，TSCAcceptor-NEW で生成し，TSCAcceptor-DELETE で削除します。TSCAcceptor クラスのインスタンスへの内部参照（アクセス）があるときは削除できないため，インスタンスへの内部参照（アクセス）をなくした状態で解放してください。

マルチスレッド環境での副プログラム呼び出し規則

マルチスレッド環境で，TSCAcceptor クラスのインスタンスの副プログラムを呼び出す

規則を次に示します。

副プログラム	複数のスレッド上からの同時呼び出し
TSCAcceptor-getInterfaceName	できます。
TSCAcceptor-getAcceptorName	できます。

インスタンスの公開副プログラム呼び出し規則

TSCAcceptor クラスのインスタンスが，ほかのクラスのインスタンスの公開副プログラムを呼び出す規則を次に示します。

タイミング	公開副プログラム呼び出し
TSCRootAcceptor からのオブジェクト管理開始通知，またはオブジェクト管理終了通知のとき	コンストラクタで指定した TSCObjectFactory 型のインスタンス
登録先の TSCRootAcceptor が active 状態のとき	active 状態に遷移するときに管理を開始した TSCObject 型のインスタンス

インスタンスへの内部参照（アクセス）規則

TSCAcceptor クラス型のインスタンスを解放したあと，このインスタンスを内部参照するインスタンスからのアクセスは，メモリアクセス違反となります。OTM は，その際の動作を保証しません。

また，複数のスレッド上から同時に TSCAcceptor クラスの同じインスタンスを内部参照（アクセス）できます。

TSCAdm（COBOL）

TSCAdm はシステム提供クラスです。

TSCAdm は，アプリケーションの初期化処理，TSCClient の取得，および TSCServer の取得をするクラスです。

形式

```
CALL 'TSCAdm-initServer' USING
                  BY REFERENCE ARGC
                  BY REFERENCE ARGV
                  BY VALUE     ORB-PTR
                  BY REFERENCE CORBA-ENVIRONMENT.
CALL 'TSCAdm-initClient' USING
                  BY REFERENCE ARGC
                  BY REFERENCE ARGV
                  BY VALUE     ORB-PTR
                  BY REFERENCE CORBA-ENVIRONMENT.

CALL 'TSCAdm-serverMainloop' USING
                  BY REFERENCE CORBA-ENVIRONMENT.
CALL 'TSCAdm-shutdown' USING
                  BY REFERENCE CORBA-ENVIRONMENT.

CALL 'TSCAdm-endServer' USING
                  BY REFERENCE CORBA-ENVIRONMENT.
CALL 'TSCAdm-endClient' USING
                  BY REFERENCE CORBA-ENVIRONMENT.

CALL 'TSCAdm-getTSCClient' USING
                  BY VALUE     DOMAIN-PTR
                  BY VALUE     WAY
                  BY REFERENCE CORBA-ENVIRONMENT
              RETURNING        CLIENT-PTR.
CALL 'TSCAdm-getTSCServer' USING
                  BY VALUE     DOMAIN-PTR
                  BY REFERENCE CORBA-ENVIRONMENT
              RETURNING        SERVER-PTR.

CALL 'TSCAdm-releaseTSCClient' USING
                  BY VALUE     CLIENT-PTR
                  BY REFERENCE CORBA-ENVIRONMENT.
CALL 'TSCAdm-releaseTSCServer' USING
                  BY VALUE     SERVER-PTR
                  BY REFERENCE CORBA-ENVIRONMENT.

CALL 'TSCAdm-get_status' USING
                  BY REFERENCE CORBA-ENVIRONMENT.
              RETURNING        STATUS.
```

副プログラム

● CALL 'TSCAdm-initServer'　USING

```
BY REFERENCE     ARGC
BY REFERENCE     ARGV
BY VALUE         ORB-PTR
BY REFERENCE     CORBA-ENVIRONMENT.
```

項目	型・(入出力の区別) 意味	
引数	BY REFERENCE ARGC PIC S9(9) COMP	(入力) コマンド引数配列の要素数
	BY REFERENCE ARGV USAGE POINTER	(入力) コマンド引数配列のポインタ
	BY VALUE ORB-PTR USAGE POINTER	(入力)ORB のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException	

サーバアプリケーションの初期化処理を実行します。

この副プログラムは，プロセスで 1 回だけ発行できます。TSCAdm-endServer 副プログラム，または TSCAdm-endClient 副プログラムの発行によって終了処理したあとでも，この副プログラムは発行できません。

この副プログラムの ARGC にはプロセスの MAIN となる副プログラムの第 1 引数を，ARGV にはプロセスの MAIN となる副プログラムの第 2 引数をそれぞれそのまま指定してください。プロセス開始時にコマンドラインで指定された情報を削除または変更して ARGC および ARGV に指定すると，正しく動作しない場合があります。tscstartprc コマンドを使用して開始したサーバアプリケーションのコマンドラインには，tscstartprc コマンドに指定したコマンドライン引数がすべて渡されます。

● CALL 'TSCAdm-initClient'　USING

```
BY REFERENCE     ARGC
BY REFERENCE     ARGV
BY VALUE         ORB-PTR
BY REFERENCE     CORBA-ENVIRONMENT.
```

項目	型・(入出力の区別) 意味	
引数	BY REFERENCE ARGC PIC S9(9) COMP	(入力) コマンド引数配列の要素数
	BY REFERENCE ARGV USAGE POINTER	(入力) コマンド引数配列のポインタ
	BY VALUE ORB-PTR USAGE POINTER	(入力)ORB のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException	

クライアントアプリケーションの初期化処理を実行します。

この副プログラムの ARGC にはプロセスの MAIN となる副プログラムの第 1 引数を，ARGV にはプロセスの MAIN となる副プログラムの第 2 引数をそれぞれそのまま指定してください。プロセス開始時にコマンドラインで指定された情報を削除または変更して ARGC および ARGV に指定すると，正しく動作しない場合があります。

この副プログラムは，TSCAdm-endClient 副プログラムの発行によって終了処理をした場合は再発行できますが，TSCAdm-endServer 副プログラムの発行によって終了処理をしたあとは再発行できません。TSCAdm-endClient 副プログラムの発行後にこの副プログラムを再発行した場合，2 回目以降の発行で指定した ARGC および ARGV の値は無効となり，初回の発行で指定した値が有効になります。ただし，この副プログラムを複数回発行するとクライアントアプリケーションの性能に影響を与えるため，推奨できません。

● CALL 'TSCAdm-serverMainloop' USING
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	

リクエストを受信待ち状態にします。この副プログラムはサーバアプリケーションでだけ発行できます。

● CALL 'TSCAdm-shutdown' USING
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	

リクエストの受信待ち状態を解除します。この副プログラムはサーバアプリケーションでだけ発行できます。

● CALL 'TSCAdm-endServer' USING
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	

サーバアプリケーションの終了処理を実行します。

この副プログラムはプロセスで１回だけ発行できます。この副プログラム発行後はそのプロセスで OTM の機能は使用しないでください。

● CALL 'TSCAdm-endClient' USING
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	

クライアントアプリケーションの終了処理を実行します。

この副プログラムの発行後は，そのプロセスで OTM または OTM - Client の機能を使用できません。また，この副プログラムで例外が発生した場合は，クライアントアプリケーションを終了させる必要があります。

● CALL 'TSCAdm-getTSCClient' USING
BY VALUE DOMAIN-PTR
BY VALUE WAY
BY REFERENCE CORBA-ENVIRONMENT
RETURNING CLIENT-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE DOMAIN-PTR USAGE POINTER	(入力)TSC ドメインのポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目

項目	型・(入出力の区別) 意味	
戻り値	CLIENT-PTR USAGE POINTER	TSCClient オブジェクト
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException TSCTransientException	

指定した TSCDomain を基に，リクエストする TSC デーモンの TSCClient のリファレンスを取得します。

WAY には，接続経路として，次に示すどれかを指定します。

- 0
  TSC デーモンに直結してリクエストします。ただし，シングルスレッドライブラリを使用するアプリケーションプログラムの場合は，TSC デーモンに直結したリクエストはできません。
- 1
  TSC レギュレータを経由してリクエストします。この場合，TSC レギュレータによってコネクションを集約します。
- 2
  アプリケーションプログラムの開始時に，コマンドオプション引数 -TSCRequestWay に指定した接続経路に従います。

ファイル検索方式でマルチノードリトライ接続を実行する場合，この副プログラムに指定した TSCDomain および WAY（WAY が "2" の場合はコマンドオプション引数 -TSCRequestWay の指定値）の組み合わせに一致する情報が，接続先情報ファイル中に記述されていなければなりません。一致する情報が接続先情報ファイルにない場合，TSCBadParamException 例外が発生します。なお，ファイル検索方式でマルチノードリトライ接続を実行するには，アプリケーションプログラムの開始時に，次に示すようにコマンドオプション引数を指定します。

- -TSCRetryReference に接続先情報ファイルを指定し，かつ，-TSCRetryWay に "0000" または "0001" を指定します。
- -TSCRetryReference に接続先情報ファイルを指定して，-TSCRetryWay の指定を省略します。この場合，TSCRetryWay には，"0000" が仮定されます。

● CALL 'TSCAdm-getTSCServer' USING

```
    BY VALUE          DOMAIN-PTR
    BY REFERENCE      CORBA-ENVIRONMENT
  RETURNING           SERVER-PTR.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE DOMAIN-PTR USAGE POINTER	(入力)TSC ドメインのポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	SERVER-PTR USAGE POINTER	サーバオブジェクトのポインタ
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException	

指定した TSCDomain を基に，自サーバへリクエストを振り分ける TSC デーモンの TSCServer のポインタを取得します。

● CALL 'TSCAdm-releaseTSCClient' USING
BY VALUE CLIENT-PTR
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE CLIENT-PTR USAGE POINTER	(入力)TSCClient オブジェクト
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException	

TSCClient を解放します。

● CALL 'TSCAdm-releaseTSCServer' USING
BY VALUE SERVER-PTR
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE SERVER-PTR USAGE POINTER	(入力)TSCServer オブジェクト
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目

項目	型・(入出力の区別) 意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException

TSCServer を解放します。

● CALL 'TSCAdm-get_status'
　　BY REFERENCE　CORBA-ENVIRONMENT
　RETURNING　　STATUS.

項目	型・(入出力の区別) 意味	
引数	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	STATUS PIC S9(9) COMP	プロセスステータス

運用管理で管理するプロセスステータスを返します。プロセスステータスを示す定数を表 7-4，表 7-5 に示します。

表 7-4　TSCAdm クラスで検出するクライアントアプリケーションのプロセスステータス（COBOL）

定数	状態	内容
0	オンライン稼働中	TSCAdm-initClient を発行してから，終了要求を受け付けるまでの状態
2	正常終了処理中	終了要求を受け付けてから，TSCAdm-endClient を発行するまでの状態
3	終了	TSCAdm-initClient の発行以前，または TSCAdm-endClient の発行以降の状態

クライアントアプリケーションの状態遷移については，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のクライアントアプリケーションの状態の検出に関する説明を参照してください。

表 7-5　TSCAdm クラスで検出するサーバアプリケーションのプロセスステータス（COBOL）

定数	状態	内容
0	正常開始処理中	TSCAdm-initServer を発行してから，TSCAdm-serverMainloop を発行するまでの状態
1	オンライン稼働中	TSCAdm-serverMainloop を発行してから，終了要求を受け付けるまでの状態
2	正常終了処理中	終了要求を受け付けてから，TSCAdm-endServer を発行するまでの状態
3	終了	TSCAdm-initServer の発行以前，または TSCAdm-endServer の発行以降の状態

サーバアプリケーションの状態遷移については，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のサーバアプリケーションの状態の検出に関する説明を参照してください。

マルチスレッド環境での副プログラム呼び出し規則

マルチスレッド環境で，TSCAdm クラスのインスタンスの副プログラムを呼び出す規則を次に示します。

副プログラム	複数のスレッド上からの同時呼び出し
initServer	できます。※
initClient	できます。※
serverMainloop	できます。※
shutdown	できます。※
endServer	できます。※
endClient	できます。※
getTSCClient	できます。
getTSCServer	できます。
releaseTSCClient	できます。
releaseTSCServer	できます。
get_status	できます。

注※

複数のスレッド上から同時に呼び出すことはできますが，有効となるのは一つの呼び出しだけです。

TSCCBLThread（COBOL）

TSCCBLThread は，COBOL85 インタフェースのために用意された，TSCThread の派生クラスです。TSCThread を生成する TSCThreadFactory が TSCRootAcceptor に登録されている場合，TSCRootAcceptor が所有する実際のスレッドを表します。

COBOL85 インタフェースでは，TSCThread を継承した COBOL85 インタフェース用の TSCCBLThread クラスを使用します。複数の TSCThread を使用したい場合，ユーザは TSCThread の派生クラスを複数作成するのではなく，TSCCBLThreadFactory クラスに**スレッドファクトリ識別子**を指定します。

TSCCBLThread-beginThread 副プログラムまたは TSCCBLThread-endThread 副プログラム中で，該当するスレッドに応じて動作を切り分ける場合も，TSCThreadFactory クラスのスレッドファクトリ識別子を利用します。スレッドファクトリ識別子は，スレッドの開始および終了時に呼び出される TSCCBLThread-beginThread および TSCCBLThread-endThread が受け取る第 1 引数に該当します。

TSCCBLThread-beginThread 副プログラムまたは TSCCBLThread-endThread 副プログラムは，OTM のトランザクションフレームジェネレータによって雛形が生成されます。内部動作はユーザが記述します。

形式

*斜体*で示している部分は，ユーザが実装のコードを記述する必要がある副プログラムです。

```
CALL 'TSCCBLThread-beginThread' USING
            BY VALUE      THREAD-FACT-ID.
CALL 'TSCCBLThread-endThread' USING
            BY VALUE      THREAD-FACT-ID.

CALL 'TSCCBLThread-getThreadFactID' USING
            BY VALUE      THREAD-PTR
            BY REFERENCE  CORBA-ENVIRONMENT
         RETURNING        THREAD-FACT-ID.
```

副プログラム

● CALL 'TSCCBLThread-beginThread' USING
　　BY VALUE　THREAD-FACT-ID.

項目	型・(入出力の区別) 意味	
引数	BY VALUE　THREAD-FACT-ID　PIC S9(9) COMP	(入力) スレッドファクトリ識別子
戻り値	ありません。	

TSCRootAcceptor がスレッドを生成するときに呼び出します。THREAD-FACT-ID 引数には，該当する TSCCBLThreadFactory のスレッドファクトリ識別子が設定されます。これによって，ユーザは副プログラムの動作をスレッドファクトリ識別子ごとに切り替えることができます。

● CALL 'TSCCBLThread-endThread' USING
BY VALUE THREAD-FACT-ID.

項目	型・(入出力の区別) 意味	
引数	BY VALUE THREAD-FACT-ID PIC S9(9) COMP	(入力) スレッドファクトリ識別子
戻り値	ありません。	

TSCRootAcceptor がスレッドを削除するときに呼び出します。THREAD-FACT-ID 引数には，該当する TSCCBLThreadFactory のスレッドファクトリ識別子が設定されます。これによって，ユーザは副プログラムの動作をスレッドファクトリ識別子ごとに切り替えることができます。

● CALL 'TSCCBLThread-getThreadFactID' USING
BY VALUE THREAD-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING THREAD-FACT-ID.

項目	型・(入出力の区別) 意味	
引数	BY VALUE THREAD-PTR USAGE POINTER	(入力)TSCThread のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	THREAD-FACT-ID PIC S9(9) COMP	スレッドファクトリ識別子

THREAD-PTR で指定する TSCThread に対応するスレッドファクトリ識別子を取得します。ユーザオペレーション内で，TSCObject-TSCThreadGet で取得する TSCThread のスレッドファクトリ識別子を取得する場合に使用できます。

TSCCBLThreadFactory（COBOL）

TSCCBLThreadFactory は，COBOL85 インタフェースのために用意された，TSCThreadFactory の派生クラスです。

COBOL85 インタフェースでは，複数のスレッドファクトリを使用したい場合，ユーザは TSCThreadFactory クラスの派生クラスを複数作成するのではなく，TSCCBLThreadFactory クラスに**スレッドファクトリ識別子**を指定します。

TSCCBLThread-beginThread 副プログラムまたは TSCCBLThread-endThread 副プログラム中で，該当するスレッドに応じて動作を切り分ける場合も，TSCThreadFactory クラスのスレッドファクトリ識別子を利用します。スレッドファクトリ識別子は，スレッドの開始および終了時に呼び出される TSCCBLThread-beginThread および TSCCBLThread-endThread が受け取る第 1 引数に該当します。

形式

```
CALL 'TSCCBLThreadFactory-NEW' USING
          BY VALUE      THREAD-FACT-ID
       RETURNING        THREAD-FACT-PTR.

CALL 'TSCCBLThreadFactory-DELETE' USING
          BY VALUE      THREAD-FACT-PTR.
```

副プログラム

● CALL 'TSCCBLThreadFactory-NEW' USING
BY VALUE　THREAD-FACT-ID
RETURNING　THREAD-FACT-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE THREAD-FACT-ID PIC S9(9) COMP	(入力) スレッドファクトリ識別子
戻り値	THREAD-FACT-PTR USAGE POINTER	TSCCBLThreadFactory のポインタ

TSCCBLThreadFactory を生成します。スレッドファクトリには，THREAD-FACT-ID で指定されたスレッドファクトリ識別子が設定されます。このスレッドファクトリ識別子は TSCCBLThread-beginThread および TSCCBLThread-endThread に引数として渡されます。ユーザは，このスレッドファクトリ識別子によって TSCCBLThread-beginThread および TSCCBLThread-endThread の動作を切り替えることができます。スレッドファクトリ識別子には，ユーザが任意の値を設定できます。

● CALL 'TSCCBLThreadFactory-DELETE' USING
BY VALUE　THREAD-FACT-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE THREAD-FACT-PTR USAGE POINTER	(入力)TSCCBLThreadFactory のポインタ
戻り値	ありません。	

TSCCBLThreadFactory を削除します。

TSCClient（COBOL）

TSCClient はシステム提供クラスです。

ABC_TSCprxy を TSC デーモンと関連づける場合，ABC_TSCprxy を生成する時点で，TSCClient は ABC_TSCprxy-NEW 副プログラムの引数に渡されます。

TSCClient は，TSC デーモン中のクライアントアプリケーション管理部分を表すクラスです。クライアントアプリケーション側からの TSC ユーザオブジェクトの呼び出し要求は，TSCClient を経由して TSC デーモンに渡されます。

ユーザは，クライアントアプリケーションが TSC デーモンと接続するときに TSCClient を取得します。クライアントアプリケーションと TSC デーモンの接続には，TSC デーモンと直結する方法と，TSC レギュレータを経由する方法があります。次に TSCClient の特徴を示します。

- 属性として TSC ドメイン名称と TSC 識別子を持ちます。

TSC デーモンに直結する場合の TSCClient の取得

クライアントアプリケーションと TSC デーモン間の直結の接続は，クライアントアプリケーションプロセス内で TSCClient を最初に取得するときに確立されます。その後，同じ TSC デーモンに対して TSCClient を取得する場合は，その接続を共有します。逆に，取得したすべての TSCClient を解放すると接続が切断されます。

一つのクライアントアプリケーションから複数の TSC デーモンへ接続を確立することもできます。また，TSC ユーザオブジェクト呼び出し要求が，この接続を経由して TSC デーモンに渡される場合，並行して処理されます。

ただし，シングルスレッドライブラリを使用するアプリケーションプログラムの場合，TSC デーモンに直結してリクエストできません。

TSC レギュレータを経由する場合の TSCClient の取得

TSC レギュレータを経由する場合のクライアントアプリケーションと TSC デーモン間の接続は，TSCClient を取得するたびに確立されます。逆に，TSClient を解放するたびに，割り当てられた接続が切断されます。

一つのクライアントアプリケーションから複数の TSC デーモンへ接続を確立することもできます。また，TSC ユーザオブジェクト呼び出し要求がこの一つの接続を経由して TSC デーモンに渡される場合，並行して処理されないで順番に処理されます。

形式

```
CALL 'TSCClient-getTSCDomainName' USING
            BY VALUE       CLIENT-PTR
```

```
            BY REFERENCE   CORBA-ENVIRONMENT
          RETURNING        DOMAIN-NAME.

CALL 'TSCClient-getTSCID' USING
            BY VALUE       CLIENT-PTR
            BY REFERENCE   CORBA-ENVIRONMENT
          RETURNING        TSCID.
```

副プログラム

● CALL 'TSCClient-getTSCDomainName' USING
BY VALUE CLIENT-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING DOMAIN-NAME.

項目	型・(入出力の区別) 意味	
引数	BY VALUE CLIENT-PTR USAGE POINTER	(入力)TSCClient のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	DOMAIN-NAME USAGE POINTER	TSC ドメイン名称のポインタ
例外	TSCBadParamException	

TSC ドメイン名称を返します。

● CALL 'TSCClient-getTSCID' USING
BY VALUE CLIENT-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING TSCID.

項目	型・(入出力の区別) 意味	
引数	BY VALUE CLIENT-PTR USAGE POINTER	(入力)TSCClient のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	TSCID USAGE POINTER	TSC 識別子のポインタ
例外	TSCBadParamException	

TSC 識別子を返します。

マルチスレッド環境での副プログラム呼び出し規則

マルチスレッド環境で，TSCClient クラスのインスタンスの副プログラムを呼び出す規則を次に示します。

副プログラム	複数のスレッド上からの同時呼び出し
TSCClient-getTSCDomainName	できます。
TSCClient-getTSCID	できます。

インスタンスの副プログラム呼び出しの内部参照（アクセス）規則

TSCClient クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

副プログラム	内部参照
TSCClient-getTSCDomainName	ありません。
TSCClient-getTSCID	ありません。

インスタンスへの内部参照（アクセス）規則

TSCClient クラスのインスタンスを解放したあと，このインスタンスを内部参照するインスタンスからのアクセスは，メモリアクセス違反となります。OTM は，その際の動作を保証しません。

また，複数のスレッドから同時にこのクラスの同じインスタンスを内部参照できます。

TSCContext（COBOL）

TSCContext はシステム提供クラスです。

TSCContext は，TSC ユーザプロキシを使用してクライアント側からサーバ側の TSC ユーザオブジェクトの副プログラムを呼び出すとき，暗黙的にサーバ側に渡すユーザデータのコンテナクラスです。次に TSCContext の特徴を示します。

- ユーザデータを保持します。

TSCContext によるユーザデータの取得

TSCProxyObject を使用して，クライアント側からサーバ側の TSC ユーザオブジェクトの副プログラムを呼び出すときに，TSCContext によって引数以外のユーザデータをサーバ側に渡すことができます。

クライアント側では，TSCProxyObject-TSCContextGet によって TSCContext を取得し，送信したいユーザデータを設定します。サーバ側では，オブジェクトが呼び出されている間，TSCObject-TSCContextGet を使用して TSCContext を取得します。この TSCContext が保持しているユーザデータは，クライアント側の TSCContext に設定したユーザデータと同じ内容です。

形式

```
CALL 'TSCContext-setUserData' USING
            BY VALUE       CONTEXT-PTR
            BY VALUE       DATA-PTR
            BY VALUE       DATA-LENGTH
            BY REFERENCE   CORBA-ENVIRONMENT.

CALL 'TSCContext-getUserData' USING
            BY VALUE       CONTEXT-PTR
            BY REFERENCE   CORBA-ENVIRONMENT.
         RETURNING         DATA-PTR.

CALL 'TSCContext-getUserDataLength' USING
            BY VALUE       CONTEXT-PTR
            BY REFERENCE   CORBA-ENVIRONMENT.
         RETURNING         DATA-LENGTH.
```

副プログラム

● CALL 'TSCContext-setUserData' USING

```
BY VALUE     CONTEXT-PTR
BY VALUE     DATA-PTR
BY VALUE     DATA-LENGTH
BY REFERENCE  CORBA-ENVIRONMENT.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE CONTEXT-PTR USAGE POINTER	(入力)TSCContext のポインタ
	BY VALUE TA-PTR USAGE POINTER	(入力) ユーザデータ（文字列のポインタ）
	BY VALUE DATA-LENGTH PIC 9(9) COMP	(入力) ユーザデータの長さ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadParamException	

ユーザデータを設定します。ここで設定したユーザデータの領域の管理責任は，ユーザにあります。

● CALL 'TSCContext-getUserData' USING
BY VALUE CONTEXT-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING DATA-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE CONTEXT-PTR USAGE POINTER	(入力)TSCContext のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	DATA-PTR USAGE POINTER	ユーザデータ
例外	TSCBadParamException	

ユーザデータを取得します。ユーザが自分で設定したものでないかぎり，ここで取得したユーザデータをユーザが解放してはいけません。

● CALL 'TSCContext-getUserDataLength' USING
BY VALUE CONTEXT-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING DATA-LENGTH.

項目	型・(入出力の区別) 意味	
引数	BY VALUE CONTEXT-PTR USAGE POINTER	(入力)TSCContext のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	DATA-LENGTH PIC 9(9) COMP	ユーザデータの長さ
例外	TSCBadParamException	

ユーザデータの長さを取得します。

マルチスレッド環境での副プログラム呼び出し規則

マルチスレッド環境で，TSCContext クラスのインスタンスの副プログラムを呼び出す規則を次に示します。

副プログラム	複数のスレッド上からの同時呼び出し
TSCContext-setUserData	できません。
TSCContext-getUserData	できません。
TSCContext-getUserDataLength	できません。

TSCDomain（COBOL）

TSCDomain はシステム提供クラスです。

TSCDomain は，TSC ドメイン情報および TSC 識別子情報を管理するホルダクラスです。

形式

```
CALL 'TSCDomain-NEW' USING
                  BY VALUE     DOMAIN-NAME
                  BY VALUE     TSCID
                  BY VALUE     FLAG
                  BY REFERENCE CORBA-ENVIRONMENT
            RETURNING   DOMAIN-PTR.

CALL 'TSCDomain-DELETE' USING
                  BY VALUE      DOMAIN-PTR
                  BY REFERENCE  CORBA-ENVIRONMENT.
```

副プログラム

● CALL 'TSCDomain-NEW' USING

BY VALUE DOMAIN-NAME
BY VALUE TSCID
BY VALUE FLAG
BY REFERENCE CORBA-ENVIRONMENT
RETURNING DOMAIN-PTR

項目	型・(入出力の区別) 意味	
引数	BY VALUE DOMAIN-NAME USAGE POINTER	(入力)TSC ドメイン名称
	BY VALUE TSCID USAGE POINTER	(入力)TSC 識別子
	BY VALUE FLAG PIC S9(9) COMP	(入力)0 または 1 のフラグ値
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	DOMAIN-PTR USAGE POINTER	TSCDomain インスタンスのポインタ
例外	TSCBadParamException TSCInitializeException	

TSCDomain クラスのインスタンスを作成します。

FLAG の値によって動作が異なります。FLAG が "0" の場合，TSC ドメイン名称だけから TSCDomain を作成します。TSCID の指定は無視されます。FLAG が "1" の場合，TSC ドメイン名称および TSC 識別子から TSCDomain を作成します。

DOMAIN-NAME および TSCID に文字列を指定する場合は，先頭が "TSC" または "tsc"

ではない 1 ～ 31 文字の英数字の文字列を指定してください。なお，TSCID に IP アドレスを指定する場合は，ピリオド (.) も使用できます。また，DOMAIN-NAME に NULL を指定する場合，TSCID に NULL を指定する場合，または FLAG に 0 を指定する場合は，アプリケーションプログラムの開始時に指定するコマンドオプション引数 -TSCRetryReference の指定の有無によって管理する情報が異なります。

- コマンドオプション引数 -TSCRetryReference を指定しない場合
  DOMAIN-NAME に NULL を指定する場合，TSCAdm-initServer，または TSCAdm-initClient の ARGV 引数内の "-TSCDomain" オプションの指定値を使用します。
  FLAG の値が 1 で，かつ TSCID に NULL を指定する場合，TSCAdm-initServer，または TSCAdm-initClient の ARGV 引数内の "-TSCID" オプションの指定値を使用します。
- コマンドオプション引数 -TSCRetryReference を指定する場合
  コマンドオプション引数 -TSCRetryWay の指定内容によって動作が異なります。
  -TSCRetryWay の指定値と接続方式の関係については，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のマルチノードリトライ接続の接続対象に関する説明を参照してください。

● CALL 'TSCDomain-DELETE' USING
BY VALUE　　DOMAIN-PTR
BY REFERENCE　　CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE　DOMAIN-PTR　USAGE POINTER	(入力)TSCDomain のポインタ
	BY REFERENCE　CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadParamException TSCInitializeException	

TSCDomain クラスのインスタンスを削除します。

TSCObject（COBOL）

TSCObject はシステム提供クラスです。

TSCObject は ABC_TSCimpl の基底クラスです。各副プログラムは ABC_TSCimpl のインスタンスに対して発行します。したがって，各副プログラムの第 1 引数（OBJECT-PTR）には，ユーザ定義 IDL インタフェース依存クラス（ABC_TSCimpl）のポインタを指定してください。

TSCObject は，OTM での（サーバ）オブジェクトの基本クラスおよびインタフェースです。

ユーザは TSCObject を継承させて，クライアント側にサービスを提供するクラスを定義します。また，サービスを提供するオブジェクトとして，その派生クラスのインスタンスを生成します。次に TSCObject の特徴を示します。

- 直接，TSCObject のインスタンスを生成できません。
- TSCObject を生成する，TSCObjectFactory の実装クラスを記述する必要があります。
- 属性としてインタフェース名称の列を持ちます。
- クライアント側から TSCProxyObject を使用して TSCObject を呼び出すときにユーザデータを TSCContext に指定すると，サーバ側で TSCContext から同じデータを取得できます。
- TSCRootAcceptor を生成するときに TSCThreadFactory を指定すると，TSCThread を取得できます。

TSCObject が提供するサービスのインタフェース名称

TSCObject が提供するインタフェースの種類は，TSCAcceptor のインタフェース名称の列で表されます。

TSCObject が単数のインタフェースを提供する場合，提供するインタフェースの種類は，単数のインタフェース名称で表されます。

TSCObject が複数のインタフェースを提供する場合，例えば，ユーザ定義 IDL インタフェースで継承を利用した場合は，複数のインタフェース名称が列で表されます。

TSCObject の呼び出し時の TSCContext の取得

TSCProxyObject を使用してクライアント側から TSCObject の副プログラムを呼び出すときに，ユーザは引数以外のデータを TSCContext として送信できます。クライアント側引数以外のデータを TSCContext として送信すると，サーバ側では TSCObject のサービス提供副プログラムが呼び出されている間に TSCObject-TSCContextGet を呼び出すことによって，クライアント側で指定した TSCContext を取得できます。

TSCObject の呼び出し時の TSCThread の取得

TSCObject の呼び出し時の TSCThread の取得は，TSCThreadFactory を引数として TSCRootAcceptor を生成する場合を前提にします。この場合，サーバ側で TSCObject のサービス提供副プログラムが呼び出されている間に TSCObject-TSCThreadGet を呼び出すことによって，実行制御を持つスレッドに割り付けられている TSCThread を取得できます。

形式

```
 CALL 'TSCObject-TSCContextGet' USING
            BY VALUE        OBJECT-PTR
            BY REFERENCE    CORBA-ENVIRONMENT
          RETURNING         CONTEXT-PTR.

* TSCクライアント情報
 CALL 'TSCObject-TSCThreadGet' USING
            BY VALUE        OBJECT-PTR
            BY REFERENCE    CORBA-ENVIRONMENT
          RETURNING         THREAD-PTR.
```

副プログラム

● CALL 'TSCObject-TSCContextGet' USING

```
      BY VALUE        OBJECT-PTR
      BY REFERENCE    CORBA-ENVIRONMENT
    RETURNING         CONTEXT-PTR.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE OBJECT-PTR USAGE POINTER	(入力)ABC_TSCimpl のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	CONTEXT-PTR USAGE POINTER	TSCContext のポインタ
例外	TSCBadParamException	

クライアント側から呼び出すときに指定された TSCContext を取得します。クライアント側から呼び出すときに TSCContext を指定しない場合は，NULL を返します。

● CALL 'TSCObject-TSCThreadGet' USING

```
      BY VALUE        OBJECT-PTR
      BY REFERENCE    CORBA-ENVIRONMENT
    RETURNING         THREAD-PTR.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE OBJECT-PTR USAGE POINTER	(入力)ABC_TSCimpl のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目

項目	型・(入出力の区別) 意味	
戻り値	THREAD-PTR USAGE POINTER	TSC ユーザスレッドのポインタ
例外	TSCBadParamException	

現在，実行制御を持つスレッドの TSC ユーザスレッドを返します。

マルチスレッド環境での副プログラム呼び出し規則

マルチスレッド環境で，TSCObject クラスのインスタンスの副プログラムを呼び出す規則を次に示します。

副プログラム	複数のスレッド上からの同時呼び出し
_TSCInterfaceName	できません。
_TSCContext	できません。
_TSCThread	できません。
クライアント側からのオブジェクト呼び出し※	できません。

注※

このメソッドは OTM が呼び出します。

TSCProxyObject（COBOL）

TSCProxyObject はシステム提供クラスです。

TSCProxyObject は ABC_TSCprxy の基底クラスです。副プログラムは ABC_TSCprxy のインスタンスに対して発行します。したがって，各副プログラムの第 1 引数（PROXY-PTR）には，ユーザ定義 IDL インタフェース依存クラス（ABC_TSCprxy）のポインタを指定してください。

TSCProxyObject は TSC ユーザオブジェクトの代理クラスです。TSCProxyObject を呼び出すと，OTM のスケジューリング機構を経由して TSCObject が呼び出されます。

ユーザは，サーバアプリケーションの TSC ユーザオブジェクトが提供するサービスを利用するときに，TSCProxyObject クラスのインスタンスを生成して呼び出します。次に TSCProxyObject の特徴を示します。

- TSCProxyObject を使用して利用できるサービスを TSC **サービス識別子**で表現します。TSC サービス識別子は，**インタフェース名称**と TSC **アクセプタ名称**から構成されます。ただし，TSC アクセプタ名称がない場合もあります。
- 属性として，タイムアウト値（呼び出し時の監視時間）とプライオリティ値（メソッド呼び出し時の優先順位）を持ちます。
- TSCContext を登録できます。

TSC サービス識別子によるサービスの識別

TSC サービス識別子の構成

TSCProxyObject を使用して利用するサービスの種類は，TSC サービス識別子によって表されます。TSC サービス識別子は，インタフェース名称と TSC アクセプタ名称から構成されます。

- **TSCProxyObject のインタフェース名称**
  TSCProxyObject のインタフェース名称は，TSCProxyObject を使用して呼び出すサービスのインタフェースの種類を表します。
- **TSCProxyObject の TSC アクセプタ名称**
  TSCProxyObject の TSC アクセプタ名称も，TSCProxyObject を使用して呼び出すサービスのインタフェースの種類を表します。ただし，同じインタフェースを提供するサービスで，実装内容の違いを識別するために使用します。したがって，TSC アクセプタ名称を設定しないで，"TSC アクセプタ名称なし " とすることもできます。

サービスの種類の表現方法

TSCProxyObject によって利用するサービスの種類は，TSC サービス識別子によって表されます。TSC サービス識別子は，インタフェース名称と TSC アクセプタ名称から構成されます。

- **TSCAcceptor に TSC アクセプタ名称が設定されていない場合**
  TSCProxyObject が呼び出すサービスのインタフェースの種類は，次のように表されます。

  TSCアクセプタ名称なしTSCサービス識別子

- **TSCAcceptor に TSC アクセプタ名称が設定されている場合**
  TSCProxyObject が呼び出すサービスの種類は，次のように表されます。

  TSCアクセプタ名称ありTSCサービス識別子

サービスの種類の表現例

- **TSCProxyObject のインタフェース名称が "ABC" で，TSC アクセプタ名称がない場合**
  次のように表される場合，TSCProxyObject は，"ABC::" への要求メッセージを生成し，"ABC::" としてサービスを提供している TSCObject と TSCAcceptor を呼び出すことができます。

  ```
  "ABC::"
  ```

  サーバアプリケーション側で，TSCProxyObject に "ABC::" と指定してサービスを呼び出した場合，その要求を受け付ける TSCObject と TSCAcceptor は，次に示すどちらかです。
  - インタフェース名称 "ABC:" だけ指定された TSCAcceptor，およびその TSCAcceptor に管理される TSCObject
  - インタフェース名称 "ABC" と任意の TSC アクセプタ名称が指定された TSCAcceptor，およびその TSCAcceptor に管理される TSCObject

  つまり，TSCProxyObject でインタフェース名称だけ指定して呼び出した場合，サーバ側では，同じインタフェース名称を持つ TSCAcceptor に管理されるすべての TSCObject が，TSC アクセプタ名称の値に関係なく呼び出されます。
- **TSCProxyObject のインタフェース名称が "ABC" で，TSC アクセプタ名称が "abc" の場合**
  次のように表される場合，TSCProxyObject は "ABC::abc" への要求メッセージを生成し，"ABC::abc" としてサービスを提供している TSCObject と TSCAcceptor を呼び出すことができます。

  ```
  "ABC::abc"
  ```

  サーバアプリケーション側で，TSCProxyObject に "ABC::abc" と指定してサービスを呼び出した場合，その要求を受け付ける TSCObject と TSCAcceptor は，インタフェース名称 "ABC" と TSC アクセプタ名称 "abc" が指定された TSCAcceptor，およびその TSCAcceptor に管理される TSCObject です。
  つまり，TSCProxyObject にインタフェース名称と TSC アクセプタ名称を指定して呼び出した場合，サーバ側では，同じインタフェース名称と同じ TSC アクセプタ名称を持つ TSCAcceptor に管理される TSCObject が呼び出されるため，TSC アクセプタ名

称によって呼び出す TSCObject を選択できます。

形式

```
* タイムアウト
 CALL 'TSCProxyObject-TSCTimeoutSet' USING
            BY VALUE       PROXY-PTR
            BY VALUE       TIMEOUT
            BY REFERENCE   CORBA-ENVIRONMENT.
 CALL 'TSCProxyObject-TSCTimeoutGet' USING
            BY VALUE       PROXY-PTR
            BY REFERENCE   CORBA-ENVIRONMENT
          RETURNING        TIMEOUT.

* 優先度
 CALL 'TSCProxyObject-TSCPrioritySet' USING
            BY VALUE       PROXY-PTR
            BY VALUE       PRIORITY
            BY REFERENCE   CORBA-ENVIRONMENT.
 CALL 'TSCProxyObject-TSCPriorityGet' USING
            BY VALUE       PROXY-PTR
            BY REFERENCE   CORBA-ENVIRONMENT
          RETURNING        PRIORITY.

 CALL 'TSCProxyObject-TSCContextGet' USING
            BY VALUE       PROXY-PTR
            BY REFERENCE   CORBA-ENVIRONMENT
          RETURNING        CONTEXT-PTR.
```

副プログラム

● CALL 'TSCProxyObject-TSCTimeoutSet' USING

BY VALUE PROXY-PTR
BY VALUE TIMEOUT
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE PROXY-PTR USAGE POINTER	(入力)ABC_TSCprxy のポインタ
	BY VALUE TIMEOUT PIC S9(9) COMP	(入力) タイムアウト時間（秒）
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadParamException	

タイムアウト値（呼び出し時の監視時間）を秒単位で設定します。"0" を指定する場合，時間監視をしません。監視時間は，副プログラム呼び出しごとに変更できます。監視時間のデフォルト値は "180"（秒）です。また，initServer 発行時の引数 args に，-TSCTimeOut オプションを指定する場合は，監視時間のデフォルト値はその指定値となります。

この副プログラムを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● CALL 'TSCProxyObject-TSCTimeoutGet' USING
BY VALUE PROXY-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING TIMEOUT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE PROXY-PTR USAGE POINTER	(入力)ABC_TSCprxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	TIMEOUT PIC S9(9) COMP	タイムアウト時間（秒）
例外	TSCBadParamException	

タイムアウト値（呼び出し時の監視時間）を取得します。

この副プログラムを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● CALL 'TSCProxyObject-TSCPrioritySet' USING
BY VALUE PROXY-PTR
BY VALUE PRIORITY
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE PROXY-PTR USAGE POINTER	(入力)ABC_TSCprxy のポインタ
	BY VALUE PRIORITY PIC S9(9) COMP	(入力) プライオリティ値
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadParamException	

プライオリティ値（副プログラム呼び出し時の優先順位）を設定します。

PRIORITY に 1 ～ 8 の値を指定することで，キューイング取り出し時の優先順位を変更できます。PRIORITY に指定する値が小さいほど優先度は高くなります。この副プログラムはリクエスト単位に変更することもできます。プライオリティ値のデフォルト値は "4" です。また，initServer 発行時の引数 args に，-TSCRequestPriority オプションを指定する場合は，プライオリティ値のデフォルト値はその指定値となります。

この副プログラムを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● CALL 'TSCProxyObject-TSCPriorityGet' USING
BY VALUE PROXY-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING PRIORITY.

項目	型・(入出力の区別) 意味	
引数	BY VALUE PROXY-PTR USAGE POINTER	(入力)ABC_TSCprxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	PRIORITY PIC S9(9) COMP	プライオリティ値
例外	TSCBadParamException	

プライオリティ値（副プログラム呼び出し時の優先順位）を取得します。

この副プログラムを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● CALL 'TSCProxyObject-TSCContextGet' USING
BY VALUE PROXY-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING CONTEXT-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE PROXY-PTR USAGE POINTER	(入力)ABC_TSCprxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	CONTEXT-PTR USAGE POINTER	TSCContext のポインタ
例外	TSCBadParamException	

呼び出し時の TSCContext を取得します。ユーザはこの TSCContext にコンテキストデータを設定できます。

マルチスレッド環境での副プログラム呼び出し規則

マルチスレッド環境で，TSCProxyObject クラスのインスタンスの副プログラムを呼び出す規則を次に示します。

副プログラム	複数のスレッド上からの同時呼び出し
TSCProxyObject-TSCTimeoutGet	できます。
TSCProxyObject-TSCTimeoutSet	できません。
TSCProxyObject-TSCPriorityGet	できます。
TSCProxyObject-TSCPrioritySet	できません。

副プログラム	複数のスレッド上からの同時呼び出し
TSCProxyObject-TSCContextGet	できません。
クライアント側からのオブジェクト呼び出し	できません。

インスタンスの内部参照（アクセス）規則

TSCProxyObject クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

副プログラム	複数のスレッド上からの内部参照
TSCProxyObject-TSCTimeoutGet	ありません。
TSCProxyObject-TSCTimeoutSet	ありません。
TSCProxyObject-TSCPriorityGet	ありません。
TSCProxyObject-TSCPrioritySet	ありません。
TSCProxyObject-TSCContextGet	ありません。
クライアント側からのオブジェクト呼び出し	生成時に指定した TSCClient 型のインスタンス

TSCRootAcceptor（COBOL）

TSCRootAcceptor はシステム提供クラスです。

TSCRootAcceptor は，サーバオブジェクトの実行空間を表現するオブジェクトです。クライアント側からの TSC ユーザオブジェクト呼び出し要求を受け付けて，適切な TSC ユーザアクセプタに振り分けます。また，パラレルカウント（常駐するスレッド数）に合わせてスレッドを管理します。

ユーザは，クライアント側にサービスを提供する TSC ユーザオブジェクトの実行空間を構築するときに，サーバアプリケーション内で TSCRootAcceptor クラスのインスタンスを生成します。次に TSCRootAcceptor の特徴を示します。

- 複数の TSCAcceptor を登録できます。
- スレッドの生成や削除などをして，スレッドを管理します。
- 属性として TSC **ルートアクセプタ状態**を持ちます。TSC ルートアクセプタ状態には active 状態と non-active 状態の 2 種類があります。
- TSC **ルートアクセプタ登録名称**を指定して active 状態に遷移できます。
- 属性として**パラレルカウント**（常駐するスレッド数）を持ちます。
- TSCThreadFactory を登録できます。
- 属性として**スケジュール用キューの長さ**を持ちます。

TSCAcceptor の登録

TSCObject の実行空間の構成

TSCRootAcceptor の実行空間で TSCObject を実行させる場合，例えば，
TSCRootAcceptor の管理するスレッド上で TSCObject を実行させる場合，その
TSCObject に対応する TSCAcceptor を TSCRootAcceptor に登録します。
TSCRootAcceptor には複数の TSCAcceptor を登録できます。

提供するサービスの管理

TSCRootAcceptor には複数の TSCAcceptor を登録できます。TSCRootAcceptor は，各 TSCAcceptor の複数の TSC サービス識別子を集め，かつ，重複を削除したものを属性として管理します。TSCRootAcceptor が提供できるサービスの種類は，この TSC サービス識別子によって表されます。つまり，TSC サービス識別子の列で表されるサービスを提供することになります。

TSC ルートアクセプタ状態

TSC ルートアクセプタ状態ごとのスレッドの管理方式

TSC ルートアクセプタ状態には active 状態と non-active 状態の 2 種類があります。各状態の遷移中の場合も含めて，それぞれの状態のときのスレッドの管理方式を次に示し

ます。

- non-active 状態
  non-active 状態のオブジェクトが管理するスレッドはありません。
  non-active 状態のときは，TSCAcceptor の登録および削除ができます。TSCAcceptor の登録時には，登録識別子が返されます。TSCAcceptor を削除するときには，その登録識別子を指定します。
  TSCRootAcceptor を生成した時点では non-active 状態です。
- non-active 状態から active 状態に遷移中
  パラレルカウント数と同数のスレッドを生成します。また，登録されているすべての TSCAcceptor にオブジェクト管理開始通知を出します。すべての TSCAcceptor がオブジェクトの管理を開始したあと，クライアント側からの TSC ユーザオブジェクト呼び出し要求を受け付けることができます。
- active 状態
  クライアント側にサービスを提供できる状態です。クライアント側から TSC ユーザオブジェクト呼び出し要求を受け取ると，適切な TSCAcceptor に振り分けます。
  non-active 状態から active 状態に遷移するときに生成されたスレッドは TSCRootAcceptor に管理されています。ユーザは，TSC 経由以外でこれらの TSC ユーザオブジェクトにアクセスしないでください。また，active 状態のときは，TSCAcceptor の登録および削除はできません。
- active 状態から non-active 状態に遷移中
  登録されているすべての TSCAcceptor にオブジェクト管理終了通知を出します。また，TSCRootAcceptor の管理下にあるスレッドを削除します。

active 状態での障害通知

サーバアプリケーション内や TSC デーモンとの接続に障害が発生してスレッドが存続できなくなる場合，そのスレッド上でオブジェクト管理終了通知を TSCAcceptor に渡します。その後，そのスレッドを削除します。なお，このときはまだ active 状態です。

障害が解除されると，再度，スレッドを生成します。その後，そのスレッド上でオブジェクト管理開始通知を TSCAcceptor に渡します。つまり，障害が発生してから解除されるまでの間は，TSCRootAcceptor は，スレッド数がパラレルカウント（常駐するスレッド数）以下の状態で存続します。

TSC ルートアクセプタ登録名称

TSCRootAcceptor が active 状態に遷移すると，TSCRootAcceptor と関連づけられている TSCServer に TSC ルートアクセプタ登録名称が登録されます。以降，クライアント側の副プログラム呼び出し要求が TSCRootAcceptor に振り分けられるようになります。

TSCRootAcceptor を active 状態に遷移させるときに，TSC ルートアクセプタ登録名称を指定することもできます。activate の呼び出し時に，TSC ルートアクセプタ登録名称を指定する場合と指定しない場合について，それぞれ次に示します。

- TSC **ルートアクセプタ登録名称を指定して** activate **を呼び出した場合**
  TSC ルートアクセプタ登録名称を "abc" とすると，関連づけられている TSCServer に "abc" として，登録されます。
- TSC **ルートアクセプタ登録名称を指定しないで** activate **を呼び出した場合**
  関連づけられている TSCServer に，デフォルトの TSC ルートアクセプタ登録名称で登録されます。デフォルトの TSC ルートアクセプタ登録名称は "default" ですが，サーバアプリケーションの開始時に指定するコマンドオプション引数 -TSCRootAcceptor によって変更できます。

また，同じ TSC ルートアクセプタ登録名称で，TSCRootAcceptor を同じ TSC デーモンに登録できます。同じ TSC ルートアクセプタ登録名称で登録した場合，スケジュール用キュー（配送機構）が共有されます。ただし，スケジュール用キューを共有する場合，登録する TSCRootAcceptor 間で提供できるサービス内容が一致している必要があります。つまり，同じ TSC ルートアクセプタ登録名称で登録する TSCRootAcceptor は，同じ TSC サービス識別子の列を持っている必要があります。

TSC ユーザスレッドファクトリによる TSC ユーザスレッドの管理

TSCThreadFactory を引数として TSCRootAcceptor を生成した場合を前提にします。この場合，TSCRootAcceptor は，non-active 状態から active 状態に遷移する過程でスレッドを生成したあと，TSCThreadFactory の create を呼び出して，戻り値である TSCThread をそのスレッドに割り当てます。また，active 状態から non-active 状態に遷移する過程で，スレッドごとに割り当てた TSCThread を引数に TSCThreadFactory の destroy を呼び出したあと，スレッドを削除します。

COBOL インタフェースでは，プリフィックスが TSCRAcceptor の副プログラムとして提供されます。

形式

```
* TSCRootAcceptorの生成
 CALL 'TSCRAcceptor-create' USING
            BY VALUE       SERVER-PTR
            BY VALUE       THREAD-FACT-PTR
            BY REFERENCE   CORBA-ENVIRONMENT
         RETURNING         R-ACCEPTOR-PTR.

* TSCRootAcceptorの削除
 CALL 'TSCRAcceptor-destroy' USING
            BY VALUE       R-ACCEPTOR-PTR
            BY REFERENCE   CORBA-ENVIRONMENT.

* TSCAcceptorの追加
 CALL 'TSCRAcceptor-registerAcceptor'   USING
            BY VALUE       R-ACCEPTOR-PTR
            BY VALUE       ACCEPTOR-PTR
            BY REFERENCE   CORBA-ENVIRONMENT.
         RETURNING         ACCEPTOR-ID.
```

```
* TSCAcceptorの登録解除
 CALL 'TSCRAcceptor-cancelAcceptor'   USING
            BY VALUE       R-ACCEPTOR-PTR
            BY VALUE       ACCEPTOR-ID
            BY REFERENCE   CORBA-ENVIRONMENT.

* パラレルカウントの設定
 CALL 'TSCRAcceptor-setParallelCount'   USING
            BY VALUE       R-ACCEPTOR-PTR
            BY VALUE       P-COUNT
            BY REFERENCE   CORBA-ENVIRONMENT.
 CALL 'TSCRAcceptor-getParallelCount'   USING
            BY VALUE       R-ACCEPTOR-PTR
            BY REFERENCE   CORBA-ENVIRONMENT
        RETURNING          P-COUNT.

* TSCRootAcceptorの活性化
 CALL 'TSCRAcceptor-activate'   USING
            BY VALUE       R-ACCEPTOR-PTR
            BY VALUE       R-ACCEPTOR-NAME
            BY REFERENCE   CORBA-ENVIRONMENT.

* TSCRootAcceptorの非活性化
 CALL 'TSCRAcceptor-deactivate' USING
            BY VALUE       R-ACCEPTOR-PTR
            BY VALUE       MODE
            BY REFERENCE   CORBA-ENVIRONMENT.

* スケジュール用キュー長の設定
 CALL 'TSCRAcceptor-setQueueLength'   USING
            BY VALUE       R-ACCEPTOR-PTR
            BY VALUE       P-LENGTH
            BY REFERENCE   CORBA-ENVIRONMENT.
 CALL 'TSCRAcceptor-getQueueLength'   USING
            BY VALUE       R-ACCEPTOR-PTR
            BY REFERENCE   CORBA-ENVIRONMENT
         RETURNING         P-LENGTH.
```

副プログラム

● CALL 'TSCRAcceptor-create' USING

```
      BY VALUE      SERVER-PTR
      BY VALUE      THREAD-FACT-PTR
      BY REFERENCE  CORBA-ENVIRONMENT
    RETURNING       R-ACCEPTOR-PTR.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE SERVER-PTR USAGE POINTER	(入力) 接続する TSC デーモン
	BY VALUE THREAD-FACT-PTR USAGE-POINTER	(入力)TSCCBLThreadFactory のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目

項目	型・(入出力の区別) 意味	
戻り値	R-ACCEPTOR-PTR　USAGE POINTER	TSCRootAcceptor のポインタ
例外	TSCBadParamException TSCNoMemoryException	

TSCServer から要求を受信する TSCRootAcceptor オブジェクトを生成します。スレッドファクトリを関連づけない場合は，THREAD-FACT-PTR に NULL ポインタを設定してください。

引数で渡す TSCServer の解放責任，引数で渡す TSCCBLThreadFactory の解放責任，および戻り値で返される TSCRootAcceptor の削除責任はユーザにあるので，適切な状態のときに解放および削除してください。

なお，この副プログラムを複数のスレッドが同時に呼び出すことができます。

● CALL 'TSCRAcceptor-destroy'　USING
　　BY VALUE　　R-ACCEPTOR-PTR
　　BY REFERENCE　CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE　R-ACCEPTOR-PTR　USAGE POINTER	(入力)TSCRootAcceptor のポインタ
	BY REFERENCE　CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	ありません。	

TSCRootAcceptor オブジェクトを解放します。解放した TSCRootAcceptor のポインタは使用できません。

なお，この副プログラムを複数のスレッドが同時に呼び出すことができます。

● CALL 'TSCRAcceptor-registerAcceptor' USING
　　BY VALUE　　R-ACCEPTOR-PTR
　　BY VALUE　　ACCEPTOR-PTR
　　BY REFERENCE　CORBA-ENVIRONMENT
　　RETURNING　　ACCEPTOR-ID.

項目	型・(入出力の区別) 意味	
引数	BY VALUE　R-ACCEPTOR-PTR　USAGE POINTER	(入力)TSCRootAcceptor のポインタ
	BY VALUE　ACCEPTOR-PTR　USAGE POINTER	(入力) 登録する TSCAcceptor のポインタ
	BY REFERENCE　CORBA-ENVIRONMENT	(出力) 例外情報集団項目

項目	型・(入出力の区別) 意味	
戻り値	ACCEPTOR-ID PIC S9(9) COMP	登録した TSCAcceptor の登録識別子
例外	TSCBadParamException TSCNoMemoryException TSCNoPermissionException	

TSCAcceptor を登録します。activate 状態のときは登録できません。

登録する TSCAcceptor のメモリ領域の管理責任はユーザにあるので，適切な状態のときに削除してください。

なお，この副プログラムを複数のスレッド上で同時に呼び出すことはできません。

● CALL 'TSCRAcceptor-cancelAcceptor' USING

```
BY VALUE      R-ACCEPTOR-PTR
BY VALUE      ACCEPTOR-ID
BY REFERENCE  CORBA-ENVIRONMENT.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE R-ACCEPTOR-PTR USAGE POINTER	(入力)TSCRootAcceptor のポインタ
	BY VALUE ACCEPTOR-ID PIC S9(9) COMP	(入力) 削除する TSCAcceptor の識別子
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadParamException TSCNoPermissionException	

TSCAcceptor オブジェクトを削除します。activate 状態のときは削除できません。

なお，この副プログラムを複数のスレッド上で同時に呼び出すことはできません。

● CALL 'TSCRAcceptor-setParallelCount' USING

```
BY VALUE      R-ACCEPTOR-PTR
BY VALUE      P-COUNT
BY REFERENCE  CORBA-ENVIRONMENT.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE R-ACCEPTOR-PTR USAGE POINTER	(入力)TSCRootAcceptor のポインタ
	BY VALUE P-COUNT PIC S9(9) COMP	(入力) パラレルカウント
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目

項目	型・(入出力の区別) 意味
戻り値	ありません。
例外	TSCBadParamException TSCNoPermissionException

パラレルカウント（常駐するスレッド数）を設定します。パラレルカウントのデフォルト値は "1" です。また，TSCAdm-initServer 発行時の引数 ARGV に -TSCParallelCount オプションを指定した場合は，パラレルカウントのデフォルト値はその指定値となります。

なお，この副プログラムを複数のスレッド上で同時に呼び出すことはできません。

● CALL 'TSCRAcceptor-getParallelCount' USING
BY VALUE R-ACCEPTOR-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING P-COUNT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE R-ACCEPTOR-PTR USAGE POINTER	(入力)TSCRootAcceptor のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	P-COUNT PIC S9(9) COMP	パラレルカウント
例外	TSCBadParamException	

パラレルカウント（常駐するスレッド数）を取得します。

この副プログラムを複数のスレッド上で同時に呼び出すことができます。

● CALL 'TSCRAcceptor-activate' USING
BY VALUE R-ACCEPTOR-PTR
BY VALUE R-ACCEPTOR-NAME
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE R-ACCEPTOR-PTR USAGE POINTER	(入力)TSCRootAcceptor のポインタ
	BY VALUE R-ACCEPTOR-NAME USAGE POINTER	(入力)TSC ルートアクセプタ登録名称
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目

項目	型・(入出力の区別) 意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResourcesException

指定した TSC ルートアクセプタ登録名称で，active 状態に遷移します。ただし，このバージョンでは，TSC ルートアクセプタ識別子の指定は無視され，常にデフォルト値が適用されます。

TSC ルートアクセプタ登録名称に NULL を指定すると，デフォルトの TSC ルートアクセプタ登録名称が使用されます。デフォルトの TSC ルートアクセプタ登録名称は "default" です。また，TSCAdm-initServer 発行時の引数 ARGV に --TSCRootAcceptorName オプションを指定した場合は，TSC ルートアクセプタ登録名称のデフォルト値はその指定値となります。

なお，この副プログラムを複数のスレッド上で同時に呼び出すことはできません。

● CALL 'TSCRAcceptor-deactivate' USING
BY VALUE R-ACCEPTOR-PTR
BY VALUE MODE
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE R-ACCEPTOR-PTR USAGE POINTER	(入力)TSCRootAcceptor のポインタ
	BY VALUE MODE PIC S9(9) COMP	(入力) 非活性化モード
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInternalException TSCNoMemoryException	

non-active 状態に遷移します。ただし，このバージョンでは非活性化モードは無視されます。

この副プログラムを複数のスレッド上で同時に呼び出すことはできません。

● CALL 'TSCRAcceptor-setQueueLength' USING
BY VALUE R-ACCEPTOR-PTR

```
BY VALUE       P-LENGTH
BY REFERENCE   CORBA-ENVIRONMENT.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE R-ACCEPTOR-PTR USAGE POINTER	(入力)TSCRootAcceptor のポインタ
	BY VALUE P-LENGTH PIC S9(9) COMP	(入力) スケジュール用キューの長さ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadParamException TSCNoPermissionException	

生成するスケジュール用キューの長さを指定します。指定できる範囲は 1 ～ 32767 です。active 状態のときは指定できません。

この副プログラムでスケジュール用キューの長さを指定しない場合，生成されるスケジュール用キューの長さは，tscstartprc コマンドまたはサーバアプリケーションの開始コマンドの -TSCQueueLength オプションで指定した長さになります。スケジュール用キューを共有する場合，すでに生成されているスケジュール用キューの長さが有効になります。

なお，この副プログラムを複数のスレッド上から同時に呼び出すことはできません。

● CALL 'TSCRAcceptor-getQueueLength' USING

```
        BY VALUE       R-ACCEPTOR-PTR
        BY REFERENCE   CORBA-ENVIRONMENT
    RETURNING       P-LENGTH.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE R-ACCEPTOR-PTR USAGE POINTER	(入力)TSCRootAcceptor のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	P-LENGTH PIC S9(9) COMP	スケジュール用キューの長さ
例外	TSCBadParamException	

non-active 状態の場合は，TSCRAcceptor-setQueueLength，または tscstartprc コマンドもしくはサーバアプリケーションの開始コマンドの -TSCQueueLength オプションで指定したスケジュール用キューの長さを取得します。TSCRAcceptor-setQueueLength が未発行で，かつ tscstartprc コマンドまたはサーバアプリケーションの開始コマンドの -TSCQueueLength オプションが指定されていないときの戻り値は "0" となります。

active 状態の場合は，現在有効になっているスケジュール用キューの長さを取得します。

なお，この副プログラムを複数のスレッド上から同時に呼び出すことができます。

TSCRootAcceptor の生成と削除

TSCServer を引数に指定した TSCRAcceptor-create で生成し，TSCRootAcceptor を引数に指定した TSCRAcceptor-destroy で削除します。TSCRootAcceptor クラスのインスタンスへの内部参照（アクセス）があるときは削除ないため，インスタンスへの内部参照（アクセス）をなくした状態で削除してください。

マルチスレッド環境での副プログラム呼び出し規則

マルチスレッド環境で，TSCRootAcceptor クラスのインスタンスの副プログラムを呼び出す規則を次に示します。

副プログラム	複数のスレッド上からの同時呼び出し
TSCRAcceptor-create	○
TSCRAcceptor-destroy	○
TSCRAcceptor-registerAcceptor	×
TSCRAcceptor-cancelAcceptor	×
TSCRAcceptor-setParallelCount	×
TSCRAcceptor-getParallelCount	○
TSCRAcceptor-activate	×
TSCRAcceptor-deactivate	×
TSCRAcceptor-setQueueLength	×
TSCRAcceptor-getQueueLength	○

（凡例）
○：できます。
×：できません。

インスタンスの内部参照（アクセス）規則

TSCRootAcceptor クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

副プログラム	内部参照
TSCRAcceptor-registerAcceptor	ありません。
TSCRAcceptor-cancelAcceptor	ありません。
TSCRAcceptor-setParallelCount	ありません。
TSCRAcceptor-getParallelCount	ありません。
TSCRAcceptor-activate()	生成時に指定された TSCServer 型のインスタンス

副プログラム	内部参照
TSCRAcceptor-deactivate()	生成時に指定された TSCServer 型のインスタンス
TSCRAcceptor-setQueueLength	ありません。
TSCRAcceptor-getQueueLength	ありません。
active 状態のとき	TSCRAcceptor-registerAcceptor の引数で指定した TSCAcceptor 型のインスタンス （TSCRootAcceptor に登録されている TSCAcceptor 型のインスタンス）

なお，TSCRootAcceptor クラスのインスタンスを解放したあと，このインスタンスを内部参照するインスタンスからのアクセスは，メモリアクセス違反となります。OTM はこのときの動作を保証しません。また，複数のスレッド上から同時に TSCRootAcceptor クラスの同じインスタンスを内部参照できます。

TSCServer（COBOL）

TSCServer はシステム提供クラスです。

TSCServer は，TSC デーモン中のサーバアプリケーション管理部分を参照するクラスです。サーバアプリケーション側の機能操作の要求は，TSCServer を経由して TSC デーモンに渡されます。また，TSC ユーザプロキシを使用した，クライアントアプリケーション側からの TSC ユーザオブジェクト呼び出し要求を受けて，TSC ルートアクセプタに振り分けます。

ユーザはサーバアプリケーションが TSC デーモンと接続するときに，TSCServer クラスのインスタンスを取得します。次に TSCServer の特徴を示します。

- 属性として TSC ドメイン名称と TSC 識別子を持ちます。

TSCServer と接続

サーバアプリケーションと TSC デーモン間の接続は，サーバアプリケーションプロセス内で TSCServer を最初に取得するときに確立されます。その後，同じ TSC デーモンに対して TSCServer を取得する場合は，その接続を共有します。逆に，取得したすべての TSCServer を解放すると接続が切断されます。

一つのサーバアプリケーションから複数の TSC デーモンへ接続を確立することもできます。また，サーバアプリケーション側の機能操作の要求が，この接続を経由して TSC デーモンに渡される場合，並行して処理されないで順番に処理されます。TSC デーモンからのオブジェクト呼び出し要求が，この接続を経由してサーバアプリケーションに配送される場合は，並行して処理されます。

形式

```
CALL 'TSCServer-getTSCDomainName'   USING
           BY VALUE        SERVER-PTR
           BY REFERENCE    CORBA-ENVIRONMENT
        RETURNING          DOMAIN-NAME.
CALL 'TSCServer-getTSCID'    USING
           BY VALUE       SERVER-PTR
           BY REFERENCE   CORBA-ENVIRONMENT
        RETURNING         TSCID.
```

副プログラム

● CALL 'TSCServer-getTSCDomainName' USING
BY VALUE SERVER-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING DOMAIN-NAME.

項目	型・(入出力の区別) 意味	
引数	BY VALUE SERVER-PTR USAGE POINTER	(入力)TSCServer のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	DOMAIN-NAME USAGE POINTER	TSC ドメイン名称の文字列のポインタ
例外	TSCBadParamException	

TSC ドメイン名称を返します。

TSC ドメイン名称のメモリ領域の管理責任は TSCServer にあるので，ユーザは解放しないでください。

なお，この副プログラムを複数のスレッドが同時に呼び出すことができます。

● CALL 'TSCServer-getTSCID' USING
BY VALUE SERVER-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING TSCID.

項目	型・(入出力の区別) 意味	
引数	BY VALUE SERVER-PTR USAGE POINTER	(入力)TSCServer のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	TSCID USAGE POINTER	TSC 識別子の文字列のポインタ
例外	TSCBadParamException	

TSCServer インスタンスが保持する TSC 識別子を返します。

TSC 識別子のメモリ領域の管理責任は TSCServer にあるので，ユーザは解放しないでください。

なお，この副プログラムを複数のスレッドが同時に呼び出すことができます。

TSCServer の取得と解放

TSCAdm-getTSCServer で取得し，TSCAdm-releaseTSCServer で解放します。
TSCServer クラスのインスタンスへの内部参照（アクセス）があるときは解放できないため，TSCServer クラスのインスタンスへの内部参照（アクセス）をなくした状態で解放してください。

マルチスレッド環境での副プログラム呼び出し規則

マルチスレッド環境で，TSCServer クラスのインスタンスの副プログラムを呼び出す際

の規則を次に示します。

副プログラム	複数のスレッド上からの同時呼び出し
TSCServer-getTSCDomainName	できます。
TSCServer-getTSCID	できます。

インスタンスの内部参照（アクセス）規則

TSCServer クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

タイミング	内部参照
TSCServer-getTSCDomainName	ありません。
TSCServer-getTSCID	ありません。
関連づけがある TSCRootAcceptor が active 状態	関連づけがある TSCRootAcceptor 型のインスタンス

なお，TSCServer クラスのインスタンスを解放したあと，このインスタンスを内部参照するインスタンスからのアクセスは，メモリアクセス違反となります。OTM は，この際の動作を保証しません。また，複数のスレッド上から同時に，このクラスの同じインスタンスを内部参照できます。

TSCSessionProxy（COBOL）

TSCSessionProxy はシステム提供クラスです。

TSCSessionProxy は，ABC_TSCspxy の基底クラスです。副プログラムは ABC_TSCspxy のインスタンスに対して発行します。したがって，各副プログラムの第 1 引数（SESSION-PROXY-PTR）には，ユーザ定義 IDL インタフェース依存クラス（ABC_TSCspxy）のポインタを指定してください。

TSCSessionProxy は，TSCObject の代理クラスです。TSCSessionProxy を呼び出すと，OTM のスケジューリング機構を経由してステートフルに TSCObject が呼び出されます。TSCSessionProxy の TSCProxyObject との違いを次に示します。

TSCSessionProxy の特徴

- TSCProxyObject の持つ属性に加え，セション呼び出しインターバル監視時間（呼び出しと呼び出しの間の監視時間）が属性に追加されます。
- _TSCStart() メソッドでセションを確立します。
  セションを確立すると，TSCSessionProxy とサーバアプリケーションのインスタンスを対応づけます。_TSCStart() メソッド発行後は，_TSCStop() メソッドの発行まで，対応づけたインスタンスにリクエストを要求します。対応づけられるサーバアプリケーションのインスタンスは TSCObject です。TSCObject と対応づけた場合は，次のことに注意してください。
  - TSCObject のインスタンスを保持しているスレッドも対応づける。
  - TSCObject のインスタンスはセションを確立していないほかのクライアントアプリケーションからのリクエストを受け付けられない。
  - すべての TSCObject のインスタンスが対応づけられた場合は，新しいクライアントアプリケーションからのリクエストを受け付けられない。
- _TSCStop() メソッドを発行すると，TSCSessionProxy とサーバアプリケーションのインスタンスとのセションを解放します。
- コンストラクタに指定する TSC アクセプタ名称は，_TSCStart() メソッド発行時に対応づけるインスタンスを決定するために使用されます。
- TSC アクセプタ名称を指定していないコンストラクタで生成した場合は，_TSCStart() メソッドで任意のインスタンスに対応づけ，対応づけた TSC アクセプタ名称を _TSCStop() メソッド発行まで引き継ぎます。

形式

```
* セション呼び出し開始メソッド
CALL 'TSCSProxy-TSCStart' USING
            BY VALUE       SESSION-PROXY-PTR
            BY REFERENCE   CORBA-ENVIRONMENT.
```

```
* セション解放メソッド
CALL 'TSCSProxy-TSCStop' USING
            BY VALUE       SESSION-PROXY-PTR
            BY REFERENCE   CORBA-ENVIRONMENT.

* セション呼び出しインターバル監視時間
CALL 'TSCSProxy-TSCSessionIntvalSet' USING
            BY VALUE       SESSION-PROXY-PTR
            BY VALUE       SESSION-INTERVAL
            BY REFERENCE   CORBA-ENVIRONMENT.
CALL 'TSCSProxy-TSCSessionIntvalGet' USING
            BY VALUE       SESSION-PROXY-PTR
            BY REFERENCE   CORBA-ENVIRONMENT.
         RETURNING         SESSION-INTERVAL.

* タイムアウト
CALL 'TSCSProxy-TSCTimeoutSet' USING
            BY VALUE       SESSION-PROXY-PTR
            BY VALUE       TIMEOUT
            BY REFERENCE   CORBA-ENVIRONMENT.
CALL 'TSCSProxy-TSCTimeoutGet' USING
            BY VALUE       SESSION-PROXY-PTR
            BY REFERENCE   CORBA-ENVIRONMENT
          RETURNING        TIMEOUT.

* 優先度
CALL 'TSCSProxy-TSCPrioritySet' USING
            BY VALUE       SESSION-PROXY-PTR
            BY VALUE       PRIORITY
            BY REFERENCE   CORBA-ENVIRONMENT.
CALL 'TSCSProxy-TSCPriorityGet' USING
            BY VALUE       SESSION-PROXY-PTR
            BY REFERENCE   CORBA-ENVIRONMENT
         RETURNING         PRIORITY.

CALL 'TSCSProxy-TSCContextGet' USING
            BY VALUE       SESSION-PROXY-PTR
            BY REFERENCE   CORBA-ENVIRONMENT
          RETURNING        CONTEXT-PTR.
```

副プログラム

● CALL 'TSCSProxy-TSCStart' USING

BY VALUE　　SESSION-PROXY-PTR
BY REFERENCE　CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE SESSION-PROXY-PTR USAGE POINTER	(入力)ABC_TSCspxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目

項目	型・(入出力の区別) 意味
戻り値	ありません。
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResponseException TSCObjectNotExistException TSCTransientException

セションを開始します。

セションを開始すると TSC ユーザプロキシとサーバアプリケーションのインスタンスを対応づけます。また，アクセプタ名称を指定していないコンストラクタでインスタンスを生成した場合は，この副プログラムで対応づけたインスタンスのアクセプタ名称を _TSCStop() メソッド発行時まで引き継ぎます。

なお，この副プログラムを複数のスレッド上から同時に呼び出すことはできません。

この副プログラムが正常終了した場合は，セション呼び出しが成功しても失敗しても，必ず _TSCStop() メソッドを発行してください。

● CALL 'TSCSProxy-TSCStop' USING
BY VALUE SESSION-PROXY-PTR
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE SESSION-PROXY-PTR USAGE POINTER	(入力)ABC_TSCspxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadInvOrderException TSCBadParamException TSCCommFailureException TSCInitializeException TSCInternalException TSCNoMemoryException TSCNoPermissionException TSCNoResponseException TSCTransientException	

セションを解放します。

セションを解放すると TSC ユーザプロキシとサーバアプリケーションのインスタンスの対応づけも解放されます。

なお，この副プログラムを複数のスレッド上から同時に呼び出すことはできません。

● CALL 'TSCSProxy-TSCSessionIntvalSet' USING

```
BY VALUE        SESSION-PROXY-PTR
BY VALUE        SESSION-INTERVAL
BY REFERENCE    CORBA-ENVIRONMENT.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE SESSION-PROXY-PTR USAGE POINTER	(入力)ABC_TSCspxy のポインタ
	BY VALUE SESSION-INTERVAL PIC S9(9) COMP	(入力) セション呼び出しインターバル監視時間（単位：秒）
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadParamException	

セション呼び出しインターバル監視時間を秒単位で設定します。メソッド呼び出しごとに変更できます。

アプリケーションプログラムの開始時に -TSCSessionInterval オプションを指定しない場合は，監視時間のデフォルト値は "180"（秒）です。-TSCSessionInterval オプションを指定する場合は，監視時間のデフォルト値は -TSCSessionInterval オプションの指定値になります。

この副プログラムを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● CALL 'TSCSProxy-TSCSessionIntvalGet' USING

```
  BY VALUE        SESSION-PROXY-PTR
  BY REFERENCE    CORBA-ENVIRONMENT
RETURNING         SESSION-INTERVAL.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE SESSION-PROXY-PTR USAGE POINTER	(入力)ABC_TSCspxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	SESSION-INTERVAL PIC S9(9) COMP	セション呼び出しインターバル監視時間（単位：秒）
例外	TSCBadParamException	

セション呼び出しインターバル監視時間を取得します。

この副プログラムを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● CALL 'TSCSProxy-TSCTimeoutSet' USING
BY VALUE SESSION-PROXY-PTR
BY VALUE TIMEOUT
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE SESSION-PROXY-PTR USAGE POINTER	(入力)ABC_TSCspxy のポインタ
	BY VALUE TIMEOUT PIC S9(9) COMP	(入力) タイムアウト時間（単位：秒）
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadParamException	

タイムアウト値（呼び出し時の監視時間）を秒単位で設定します。"0" を指定した場合は，時間監視をしません。監視時間は，副プログラム呼び出しごとに変更できます。

initServer 発行時の引数 args に -TSCTimeOut オプションを指定しない場合は，監視時間のデフォルト値は "180"（秒）です。-TSCTimeOut オプションを指定する場合は，監視時間のデフォルト値は -TSCTimeOut オプションの指定値になります。

この副プログラムを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

タイムアウト値が有効になる副プログラムを次に示します。

- TSCSProxy-TSCStart
- クライアント側からのオブジェクト呼び出し
- TSCSProxy-TSCStop

● CALL 'TSCSProxy-TSCTimeoutGet' USING
BY VALUE SESSION-PROXY-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING TIMEOUT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE SESSION-PROXY-PTR USAGE POINTER	(入力)ABC_TSCspxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	TIMEOUT PIC S9(9) COMP	タイムアウト時間（単位：秒）
例外	TSCBadParamException	

タイムアウト値（呼び出し時の監視時間）を取得します。

この副プログラムを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● CALL 'TSCSProxy-TSCPrioritySet' USING
BY VALUE SESSION-PROXY-PTR
BY VALUE PRIORITY
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE SESSION-PROXY-PTR USAGE POINTER	(入力)ABC_TSCspxy のポインタ
	BY VALUE PRIORITY PIC S9(9) COMP	(入力) プライオリティ値
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadParamException	

プライオリティ値（副プログラム呼び出し時の優先順位）を設定します。

PRIORITY に 1 ～ 8 の値を指定することで，キューイング取り出し時の優先順位を変更できます。PRIORITY に指定する値が小さいほど優先度は高くなります。プライオリティ値はリクエスト単位に変更できます。

initServer 発行時の引数 args に，-TSCRequestPriority オプションを指定しない場合は，プライオリティ値のデフォルト値は "4" です。initServer 発行時の引数 args に，-TSCRequestPriority オプションを指定する場合は，プライオリティ値のデフォルト値は -TSCRequestPriority オプションの指定値になります。

この副プログラムを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

プライオリティ値が有効になるのは TSCSProxy-TSCStart() 副プログラムだけです。

● CALL 'TSCSProxy-TSCPriorityGet' USING
BY VALUE SESSION-PROXY-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING PRIORITY.

項目	型・(入出力の区別) 意味	
引数	BY VALUE SESSION-PROXY-PTR USAGE POINTER	(入力)ABC_TSCspxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	PRIORITY PIC S9(9) COMP	プライオリティ値
例外	TSCBadParamException	

プライオリティ値（副プログラム呼び出し時の優先順位）を取得します。

この副プログラムを複数のスレッド上から同時に呼び出した場合，結果は保証されません。

● CALL 'TSCSProxy-TSCContextGet' USING
BY VALUE SESSION-PROXY-PTR
BY REFERENCE CORBA-ENVIRONMENT
RETURNING CONTEXT-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE SESSION-PROXY-PTR USAGE POINTER	(入力)ABC_TSCspxy のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	CONTEXT-PTR USAGE POINTER	TSCContext のポインタ
例外	TSCBadParamException	

TSC コンテキストを取得します。ユーザはこの TSC コンテキストにコンテキストデータを設定できます。

マルチスレッド環境での副プログラム呼び出し規則

マルチスレッド環境で，TSCSessionProxy クラスのインスタンスの副プログラムを呼び出す規則を次に示します。

副プログラム	複数のスレッド上からの同時呼び出し
TSCSProxy-TSCStart	できません。
TSCSProxy-TSCStop	できません。
TSCSProxy-TSCSessionIntvalSet	できません。
TSCSProxy-TSCSessionIntvalGet	できます。
TSCSProxy-TSCTimeoutSet	できません。
TSCSProxy-TSCTimeoutGet	できます。
TSCSProxy-TSCPrioritySet	できません。
TSCSProxy-TSCPriorityGet	できます。
TSCSProxy-TSCContextGet	できません。
クライアント側からのオブジェクト呼び出し	できません。

インスタンスの内部参照（アクセス）規則

TSCSessionProxy クラスのインスタンスがほかのクラスのインスタンスを内部参照（アクセス）する規則を次に示します。

副プログラム	複数のスレッド上からの内部参照
TSCSProxy-TSCStart	ありません。
TSCSProxy-TSCStop	ありません。
TSCSProxy-TSCSessionIntvalSet	ありません。
TSCSProxy-TSCSessionIntvalGet	ありません。
TSCSProxy-TSCTimeoutSet	ありません。
TSCSProxy-TSCTimeoutGet	ありません。
TSCSProxy-TSCPrioritySet	ありません。
TSCSProxy-TSCPriorityGet	ありません。
TSCSProxy-TSCContextGet	ありません。
クライアント側からのオブジェクト呼び出し	生成時に指定した TSCClient 型のインスタンス

セション呼び出し機能使用中の副プログラム呼び出し規則

セション呼び出し機能使用中（TSCSProxy-TSCStart 副プログラム呼び出し完了から TSCSProxy-TSCStop 副プログラム呼び出し完了）の，TSCSessionProxy クラスのインスタンスの副プログラムを呼び出す規則を次に示します。

副プログラム	セション呼び出し中での呼び出し
TSCSProxy-TSCStart	×
TSCSProxy-TSCStop	○
TSCSProxy-TSCSessionIntervalSet	○
TSCSProxy-TSCSessionIntervalGet	○
TSCSProxy-TSCTimeoutSet	○
TSCSProxy-TSCTimeoutGet	○
TSCSProxy-TSCPrioritySet	○
TSCSProxy-TSCPriorityGet	○
TSCSProxy-TSCContextGet	○
クライアント側からのオブジェクト呼び出し	○

（凡例）

○：できます。

×：できません。

TSCSystemException（COBOL）

TSCSystemException はシステム提供例外クラスです。

COBOL85 インタフェースでの例外は，各副プログラムの**例外情報集団項目**（CORBA-ENVIRONMENT 引数）に渡されます。例外集団項目は，次に示す集団項目です。

```
01  CORBA-ENVIRONMENT.
  02  MAJOR        PIC 9(9) COMP.
    88 CORBA-NO-EXCEPTION         VALUE 0.
    88 CORBA-USER-EXCEPTION       VALUE 1.
    88 CORBA-SYSTEM-EXCEPTION     VALUE 2.
  02  EXCEP        USAGE POINTER.
  02  FUNC-NAME    PIC X(256).
```

ユーザのオペレーション呼び出し中，または OTM のシステム処理中に例外が発生した場合，処理終了時に例外情報集団項目中の MAJOR に例外種別が設定され，EXCEP に例外オブジェクトポインタが設定されます。

例外オブジェクトポインタは，次に示す例外オブジェクトを指します。

- **システム例外（CORBA-SYSTEM-EXCEPTION）の場合**
  例外種別が CORBA-SYSTEM-EXCEPTION の場合，例外オブジェクトポインタは OTM のシステム例外を指します。これは OTM 独自の型なので，COBOL adapter for TPBroker のラッパー関数を使用するアクセスはできません。
- **ユーザ例外（CORBA-USER-EXCEPTION）の場合**
  例外種別が CORBA-USER-EXCEPTION の場合，例外オブジェクトポインタは COBOL adapter for TPBroker の場合と同じ例外オブジェクトです。アクセスには COBOL adapter for TPBroker のラッパー関数を使用してください。

TSCSystemException は，TSC システム例外を扱うためのクラスです。ユーザは TSCSystemException クラスの副プログラムを使用して，例外情報を取得できます。なお，実際の副プログラムのプリフィックスは，"TSCSysExcept" となります。

形式

```
CALL 'TSCSysExcept-getErrorCode' USING
            BY VALUE     EXCEPT-PTR
         RETURNING       ERROR-CODE.

CALL 'TSCSysExcept-getDetailCode' USING
            BY VALUE      EXCEPT-PTR
         RETURNING        DETAIL-CODE.

CALL 'TSCSysExcept-getPlaceCode' USING
            BY VALUE      EXCEPT-PTR
```

```
          RETURNING         PLACE-CODE.

CALL 'TSCSysExcept-getCompletion' USING
             BY VALUE       EXCEPT-PTR
          RETURNING         COMPLETION-STATUS.

CALL 'TSCSysExcept-getMaintenance1' USING
             BY VALUE       EXCEPT-PTR
          RETURNING         MAINTENANCE1.

CALL 'TSCSysExcept-getMaintenance2' USING
             BY VALUE       EXCEPT-PTR
          RETURNING         MAINTENANCE2.

CALL 'TSCSysExcept-getMaintenance3' USING
             BY VALUE       EXCEPT-PTR
          RETURNING         MAINTENANCE3.

CALL 'TSCSysExcept-getMaintenance4' USING
             BY VALUE       EXCEPT-PTR
          RETURNING         MAINTENANCE4.

CALL 'TSCSysExcept-DELETE' USING
             BY VALUE       EXCEPT-PTR.
```

副プログラム

● CALL 'TSCSysExcept-getErrorCode'　USING
　　BY VALUE　EXCEPT-PTR
　RETURNING　 ERROR-CODE.

項目	型・(入出力の区別) 意味	
引数	BY VALUE　EXCEPT-PTR　USAGE POINTER	(入力) 例外オブジェクトポインタ
戻り値	ERROR-CODE　PIC S9(9) COMP	エラーコード

障害のエラーコードを返します。EXCEPT-PTR 引数が OTM のシステム例外でない場合，動作が常に同じになるという保証はありません。

エラーコードの値は TSCSysExcept-ERROR-CODE データ項目の条件名として，$TSCDIR/include/COBOL/TSCSysExcept.cbl ファイルに定義されています。ユーザは必要に応じてこのファイルを COPY できます。

エラーコードの値の順に並べた一覧を次の表に示します。

表 7-6　エラーコードの一覧（COBOL）

条件名	説明	値
TSCSysExcept_ERR_BAD_PARAM	無効な引数を指定し副プログラムを呼び出しました。	1
TSCSysExcept_ERR_NO_MEMORY	動的メモリの割り当て障害が発生しました。	2

条件名	説明	値
TSCSysExcept_ERR_COMM_FAILURE	通信障害が発生しました。	3
TSCSysExcept_ERR_NO_PERMISSION	許可されていない副プログラムを呼び出しました。	4
TSCSysExcept_ERR_INTERNAL	ORB 内部エラーが発生しました。	5
TSCSysExcept_ERR_MARSHAL	スタブ，スケルトンで CDR マーシャルに失敗しました。	6
TSCSysExcept_ERR_INITIALIZE	ORB 初期化障害が発生しました。	7
TSCSysExcept_ERR_NO_IMPLEMENT	オペレーションの実装が使用できません。	8
TSCSysExcept_ERR_BAD_OPERATION	オペレーションが無効です。	9
TSCSysExcept_ERR_NO_RESOURCES	リクエストを処理するための資源が不足しています。	10
TSCSysExcept_ERR_NO_RESPONSE	リクエストに対する応答がありません。	11
TSCSysExcept_ERR_BAD_INV_ORDER	副プログラムの発行順序が不正です。	12
TSCSysExcept_ERR_TRANSIENT	一時的な障害が発生しました。	13
TSCSysExcept_ERR_NOT_EXIST	該当するオブジェクトがありません。	14
TSCSysExcept_ERR_UNKNOWN	未知の例外が発生しました。	15
TSCSysExcept_ERR_INV_OBJREF	無効なオブジェクトリファレンスが指定されました。	16
TSCSysExcept_ERR_IMP_LIMIT	実装の制限を超えました。	17
TSCSysExcept_ERR_BAD_TYPECODE	タイプコードが不正です。	18
TSCSysExcept_ERR_PERSIST_STORE	パーシステントストレージに障害が発生しました。	19
TSCSysExcept_ERR_FREE_MEM	メモリの解放に失敗しました。	20
TSCSysExcept_ERR_INV_IDENT	識別子が不正です。	21
TSCSysExcept_ERR_INV_FLAG	不正なフラグが指定されました。	22
TSCSysExcept_ERR_INTF_REPOS	インタフェースリポジトリへのアクセス中に障害が発生しました。	23
TSCSysExcept_ERR_BAD_CONTEXT	コンテキストオブジェクトの処理中に障害が発生しました。	24
TSCSysExcept_ERR_OBJ_ADAPTER	オブジェクトアダプタが障害を検出しました。	25
TSCSysExcept_ERR_DATA_CONV	データ変換に失敗しました。	26

個々のエラーコードの詳細については，「付録 A　エラーコード一覧」を参照してください。

● CALL 'TSCSysExcept-getDetailCode'　USING

```
    BY VALUE   EXCEPT-PTR
  RETURNING    DETAIL-CODE.
```

項目	型・(入出力の区別) 意味	
引数	BY VALUE EXCEPT-PTR USAGE POINTER	(入力) 例外オブジェクトポインタ
戻り値	DETAIL-CODE PIC S9(9) COMP	内容コード

障害の内容コードを返します。EXCEPT-PTR 引数が OTM のシステム例外でない場合，動作が常に同じになるという保証はありません。

個々の内容コードの詳細については，「付録 D　内容コード一覧」を参照してください。

● CALL 'TSCSysExcept-getPlaceCode'　USING
　　BY VALUE　EXCEPT-PTR
　RETURNING　PLACE-CODE.

項目	型・(入出力の区別) 意味	
引数	BY VALUE EXCEPT-PTR USAGE POINTER	(入力) 例外オブジェクトポインタ
戻り値	DETAIL-CODE PIC S9(9) COMP	場所コード

障害の場所コードを返します。EXCEPT-PTR 引数が OTM のシステム例外でない場合，動作が常に同じになるという保証はありません。

場所コードの値は，TSCSysExcept-PLACE-CODE データ項目の条件名として，$TSCDIR/include/COBOL/TSCSysExcept.cbl ファイルに定義されています。ユーザは必要に応じてこのファイルを COPY できます。

場所コードの値の順に並べた一覧を次の表に示します。

表 7-7　場所コードの一覧（COBOL）

条件名	場所	値
TSCSysExcept_PLACE_USER_AP	ユーザアプリケーション	1
TSCSysExcept_PLACE_SERV	OTM のサーバ機能部分	2
TSCSysExcept_PLACE_DAEMON	TSC デーモン	3
TSCSysExcept_PLACE_CLNT	OTM のクライアント機能部分	4
TSCSysExcept_PLACE_CLNT_REG	TSC レギュレータ	5
TSCSysExcept_PLACE_STUB	スタブ	6
TSCSysExcept_PLACE_SKELTON	スケルトン	7
TSCSysExcept_PLACE_ORBGW	TSCORB コネクタ	8

● CALL 'TSCSysExcept-getCompletion'　USING
　　BY VALUE　EXCEPT-PTR

RETURNING　COMPLETION-STATUS.

項目	型・(入出力の区別) 意味	
引数	BY VALUE EXCEPT-PTR USAGE POINTER	(入力) 例外オブジェクトポインタ
戻り値	COMPLETION-STATUS PIC S9(9) COMP	完了状態

障害発生時の副プログラム呼び出しの完了状態を返します。EXCEPT-PTR 引数が OTM のシステム例外でない場合，動作が常に同じになるという保証はありません。

完了状態の値は，TSCSysExcept-COMPLETION-STATUS データ項目の条件名として，$TSCDIR/include/COBOL/TSCSysExcept.cbl ファイルに定義されています。ユーザは必要に応じてこのファイルを COPY できます。

完了状態の値の順に並べた一覧を次の表に示します。

表 7-8　完了状態の一覧（COBOL）

完了状態	説明	値
TSCSysExcept_COMPLETED_NO	副プログラム呼び出しが完了していません。	-1
TSCSysExcept_COMPLETED_MAYBE	副プログラム呼び出しの完了状態を決定できません。	0
TSCSysExcept_COMPLETED_YES	副プログラム呼び出しの処理が完了しています。	1

● CALL 'TSCSysExcept-getMaintenance1'　USING
　　BY VALUE　EXCEPT-PTR
　RETURNING　MAINTENANCE1.

項目	型・(入出力の区別) 意味	
引数	BY VALUE EXCEPT-PTR USAGE POINTER	(入力) 例外オブジェクトポインタ
戻り値	MAINTENANCE1 PIC S9(9) COMP	保守コード 1

障害の保守コード 1 を返します。EXCEPT-PTR 引数が OTM のシステム例外でない場合，動作が常に同じになるという保証はありません。

● CALL 'TSCSysExcept-getMaintenance2'　USING
　　BY VALUE　EXCEPT-PTR
　RETURNING　MAINTENANCE2.

項目	型・(入出力の区別) 意味	
引数	BY VALUE EXCEPT-PTR USAGE POINTER	(入力) 例外オブジェクトポインタ
戻り値	MAINTENANCE2 PIC S9(9) COMP	保守コード 2

障害の保守コード 2 を返します。EXCEPT-PTR 引数が OTM のシステム例外でない場合，動作が常に同じになるという保証はありません。

● CALL 'TSCSysExcept-getMaintenance3' USING
BY VALUE EXCEPT-PTR
RETURNING MAINTENANCE3.

項目	型・(入出力の区別) 意味	
引数	BY VALUE EXCEPT-PTR USAGE POINTER	(入力) 例外オブジェクトポインタ
戻り値	MAINTENANCE3 PIC S9(9) COMP	保守コード 3

障害の保守コード 3 を返します。EXCEPT-PTR 引数が OTM のシステム例外でない場合，動作が常に同じになるという保証はありません。

● CALL 'TSCSysExcept-getMaintenance4' USING
BY VALUE EXCEPT-PTR
RETURNING MAINTENANCE4.

項目	型・(入出力の区別) 意味	
引数	BY VALUE EXCEPT-PTR USAGE POINTER	(入力) 例外オブジェクトポインタ
戻り値	MAINTENANCE4 PIC S9(9) COMP	保守コード 4

障害の保守コード 4 を返します。EXCEPT-PTR 引数が OTM のシステム例外でない場合，動作が常に同じになるという保証はありません。

● CALL 'TSCSysExcept-DELETE' USING
BY VALUE EXCEPT-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE EXCEPT-PTR USAGE POINTER	(入力) 例外オブジェクトポインタ
戻り値	ありません。	

指定した TSCSystemException を解放します。EXCEPT-PTR 引数が OTM のシステム例外でない場合，動作が常に同じになるという保証はありません。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCSystemException クラスのインスタンスの副プログラムを呼び出す規則を次に示します。

副プログラム	複数のスレッド上からの同時呼び出し
TSCSysExcept-getErrorCode	できます。
TSCSysExcept-getDetailCode	できます。
TSCSysExcept-getPlaceCode	できます。
TSCSysExcept-getCompletionStatus	できます。
TSCSysExcept-getMaintenanceCode1	できます。
TSCSysExcept-getMaintenanceCode2	できます。
TSCSysExcept-getMaintenanceCode3	できます。
TSCSysExcept-getMaintenanceCode4	できます。

TSCWatchTime（COBOL）

TSCWatchTime はシステム提供クラスです。

TSCWatchTime は，TSCWatchTime-start が発行されてから TSCWatchTime-stop が発行されるまでの時間を監視するクラスです。時間監視が終了する前に指定された監視時間が経過するとエラーメッセージを出力して，プロセスを異常終了します。この機能は，サーバアプリケーションの TSCAdm-initServer 発行後から TSCAdm-endServer を発行するまでの間有効です。

形式

```
CALL 'TSCWatchTime-NEW' USING
           BY VALUE       WATCH-TIME
           BY REFERENCE   CORBA-ENVIRONMENT
        RETURNING         WATCH-TIME-PTR.

CALL 'TSCWatchTime-start' USING
           BY VALUE       WATCH-TIME-PTR
           BY REFERENCE   CORBA-ENVIRONMENT.

CALL 'TSCWatchTime-stop' USING
           BY VALUE       WATCH-TIME-PTR
           BY REFERENCE   CORBA-ENVIRONMENT.

CALL 'TSCWatchTime-reset' USING
           BY VALUE       WATCH-TIME-PTR
           BY REFERENCE   CORBA-ENVIRONMENT.
```

副プログラム

● CALL 'TSCWatchTime-NEW' USING

BY VALUE　WATCH-TIME
BY REFERENCE　CORBA-ENVIRONMENT
RETURNING　WATCH-TIME-PTR.

項目	型・(入出力の区別) 意味	
引数	BY VALUE　WATCH-TIME　PIC S9(9) COMP	(入力) 監視時間（秒）
	BY REFERENCE　CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	WATCH-TIME-PTR　USAGE POINTER	TSCWatchTime のポインタ
例外	TSCBadInvOrderException TSCBadParamException TSCInternalException TSCNoMemoryException	

TSCWatchTime を生成します。

引数に "0" を指定した場合，サーバアプリケーションの開始時にコマンドオプション引数

-TSCWatchTime で指定した監視時間（秒）が適用されます。

● CALL 'TSCWatchTime-start' USING
BY VALUE WATCH-TIME-PTR
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE WATCH-TIME-PTR USAGE POINTER	(入力)TSCWatchTime のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCInternalException TSCNoPermissionException	

時間監視を開始します。または，TSCWatchTime-stop で中断した時間監視を再開します。

● CALL 'TSCWatchTime-stop' USING
BY VALUE WATCH-TIME-PTR
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE WATCH-TIME-PTR USAGE POINTER	(入力)TSCWatchTime のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCInternalException TSCNoPermissionException	

時間監視を中断します。TSCWatchTime-start を発行したスレッドと異なるスレッドでは，TSCWatchTime-stop を発行できません。

● CALL 'TSCWatchTime-reset' USING
BY VALUE WATCH-TIME-PTR
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE WATCH-TIME-PTR USAGE POINTER	(入力)TSCWatchTime のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCInternalException TSCNoPermissionException	

監視時間をこのクラスの生成時に指定した値に戻します。TSCWatchTime-start の発行以降 TSCWatchTime-stop の発行までの間は発行できません。

● CALL 'TSCWatchTime-DELETE' USING
BY VALUE WATCH-TIME-PTR
BY REFERENCE CORBA-ENVIRONMENT.

項目	型・(入出力の区別) 意味	
引数	BY VALUE WATCH-TIME-PTR USAGE POINTER	(入力)TSCWatchTime のポインタ
	BY REFERENCE CORBA-ENVIRONMENT	(出力) 例外情報集団項目
戻り値	ありません。	
例外	TSCBadParamException TSCInternalException	

TSCWatchTime クラスのインスタンスを削除します。

マルチスレッド環境でのメソッド呼び出し規則

マルチスレッド環境で，TSCWatchTime クラスのインスタンスの副プログラムを呼び出す規則を次に示します。

メソッド	複数のスレッド上からの同時呼び出し
TSCWatchTime-NEW	できます。
TSCWatchTime-start	できます。
TSCWatchTime-stop	できます。※
TSCWatchTime-reset	できます。
TSCWatchTime-DELETE	できます。

注※
必ず TSCWatchTime-start と同一のスレッドで発行してください。

注意事項

● IDL ファイルに記述したユーザメソッドの呼び出しからリターンまでの実行時間を監視する場合には，TSCWatchTime クラスではなく，サーバアプリケーションの開始時に指定するコマンドオプション引数 -TSCWatchMethod を使用してください。

● TSCAdm-initServer の発行前，または TSCAdm-endServer の発行後には，TSCWatchTime クラスのインスタンスを生成できません。

● TSCWatchTime-stop を発行する前に TSCWatchTime-DELETE が発行されると，時間監視を終了します。

- TSCWatchTime-start を発行したスレッドと異なるスレッドでは，TSCWatchTime-stop を呼び出せません。
- TSCWatchTime-stop を発行したあとに TSCWatchTime-start によって処理を再開する場合には，前回経過した時間を監視時間から差し引いて処理をします。次に例を示します。

```
01 WATCH-TIME-PTR   USAGE POINTER.
01 WATCH-TIME       PIC S9(9) COMP.
  :
 MOVE 180 TO WATCH-TIME.
 CALL 'TSCWatchTime-NEW' USING
 BY VALUE           WATCH-TIME
 BY REFERENCE       CORBA-ENVIRONMENT
RETURNING WATCH-TIME-PTR.
  :
 CALL 'TSCWatchTime-start' USING
 BY VALUE           WATCH-TIME-PTR
 BY REFERENCE       CORBA-ENVIRONMENT.
  :               //(1) 60秒経過
 CALL 'TSCWatchTime-stop' USING
 BY VALUE           WATCH-TIME-PTR
 BY REFERENCE       CORBA-ENVIRONMENT.
  :              //(2)
 CALL 'TSCWatchTime-start' USING
 BY VALUE           WATCH-TIME-PTR
 BY REFERENCE       CORBA-ENVIRONMENT.
```

例えば，180 秒の監視時間を設定してクラスを生成した場合に，(1) で示す範囲で 60 秒が経過すると，TSCWatchTime-stop の発行後，次の TSCWatchTime-start から TSCWatchTime-stop までの監視時間は 120 秒になります。180 秒の時間監視を設定したい場合には (2) で示す範囲で TSCWatchTime-reset を発行してください。その場合，再発行した TSCWatchTime-start から TSCWatchTime-stop までの監視時間は 180 秒になります。

8 コマンドリファレンス

この章では，アプリケーションプログラムの作成時に使用するコマンドについて，コマンド名のアルファベット順に説明します。

コマンドの一覧

コマンドの一覧

アプリケーションプログラムの作成時に使用するコマンドの一覧を表 8-1 に示します。
その他の運用コマンドについては，マニュアル「TPBroker Object Transaction Monitor ユーザーズガイド」を参照してください。

なお，これらのコマンドは，OTM と OTM - Client の両方で使用できます。

表 8-1　アプリケーションプログラムの作成時に使用するコマンドの一覧

コマンド名	説明
tscidl2cpp	C++ 用トランザクションフレームの出力
tscidl2j	Java 用トランザクションフレームの出力
tscidl2cbl	COBOL 用トランザクションフレームの出力

tscidl2cbl（トランザクションフレームの出力（COBOL））

形式

```
tscidl2cbl [-h] | [-TSCclient_ext文字列] [-TSCserver_ext 文字列]
           [-TSCtemplate_ext 文字列] [-TSCsession_ext 文字列]
           [-TSCprxy_ext 文字列] [-TSCsk_ext 文字列]
           [-TSCacpt_ext 文字列] [-TSCfact_ext 文字列]
           [-TSCimpl_ext 文字列] [-TSCspxy_ext 文字列]
           [-client_ext文字列] [-server_ext文字列]
           [-idl2cobol] [-template] [-format 1|2] [-TSCspxy]
           [-TSCroot_dir パス名] [-root_dir パス名]
           [-TSCno_proxy] [-TSCno_skel] [-TSCidl2cblfix フラグ]
           [[-I ディレクトリ[:ディレクトリ...]]...]
           [-A] IDLファイル名称
```

機能

トランザクションフレームジェネレータです。ユーザ定義 IDL インタフェース依存クラスなどの COBOL ソースを出力します。

オプション

● -h

ヘルプメッセージを出力します。ファイルの読み込みおよび生成はしません。

● -TSCclient_ext **文字列**

～＜文字列＞《_TSC_c》

クライアント部分を出力するファイル名称に付加する文字列を指定します。省略した場合は "_TSC_c" が設定されます。

● -TSCserver_ext **文字列**

～＜文字列＞《_TSC_s》

サーバ部分を出力するファイル名称に付加する文字列を指定します。省略した場合は "_TSC_s" が設定されます。

● -TSCtemplate_ext **文字列**

～＜文字列＞《_TSC_t》

雛形部分を出力するファイル名称に付加する文字列を指定します。省略した場合は "_TSC_t" が設定されます。

-template オプションを指定しない場合は，-TSCtemplate_ext オプションは無視されま

す。

● -TSCsession_ext **文字列**

～＜文字列＞《_TSC_p》

セション用 TSC ユーザプロキシ部分を出力するファイル名称に付加する文字列を指定します。省略した場合は "_TSC_p" が設定されます。

-TSCspxy オプションを指定しない場合は，-TSCsession_ext オプションは無視されます。

● -TSCprxy_ext **文字列**

～＜文字列＞《_TSCprxy》

ユーザ定義 IDL インタフェース依存クラスの ABC_TSCprxy の名称に付加する文字列を指定します。省略した場合は "_TSCprxy" が設定されます。

● -TSCsk_ext **文字列**

～＜文字列＞《_TSCsk》

ユーザ定義 IDL インタフェース依存クラスの ABC_TSCsk の名称に付加する文字列を指定します。省略した場合は "_TSCsk" が設定されます。

● -TSCacpt_ext **文字列**

～＜文字列＞《_TSCacpt》

ユーザ定義 IDL インタフェース依存クラスの ABC_TSCacpt の名称に付加する文字列を指定します。省略した場合は "_TSCacpt" が設定されます。

● -TSCfact_ext **文字列**

～＜文字列＞《_TSCfact》

雛形クラスの ABC_TSCfactimpl の名称に付加する文字列を指定します。省略した場合は "_TSCfact" が設定されます。

● -TSCimpl_ext **文字列**

～＜文字列＞《_TSCimpl》

雛形クラスの ABC_TSCimpl の名称に付加する文字列を指定します。省略した場合は "_TSCimpl" が設定されます。

-template オプションを指定しない場合は，-TSCimpl_ext オプションは無視されます。

● -TSCspxy_ext **文字列**

～＜文字列＞《_TSCspxy》

セション用 TSC ユーザプロキシの名称に付加する文字列を指定します。省略した場合は "_TSCspxy" が設定されます。

-TSCspxy オプションを指定しない場合は，-TSCspxy_ext オプションは無視されます。

● -client_ext **文字列**

～＜文字列＞《_c》

idl2cobol コマンドで出力されるクライアント部分のファイルに付加する文字列を指定します。省略した場合は "_c" が設定されます。

idl2cobol コマンドで出力されるクライアント部分のファイルに付加する文字列を変更する場合は，-idl2cobol オプションを指定しないときでも必ず -client_ext オプションを指定してください。

● -server_ext **文字列**

～＜文字列＞《_s》

idl2cobol コマンドで出力されるサーバ部分のファイルに付加する文字列を指定します。省略した場合は "_s" が設定されます。

idl2cobol コマンドで出力されるサーバ部分のファイルに付加する文字列を変更する場合は，-idl2cobol オプションを指定しないときでも必ず -server_ext オプションを指定してください。

● -idl2cobol

内部で TPBroker および COBOL adapter for TPBroker のコマンドを実行して，COBOL adapter for TPBroker スタブおよび COBOL adapter for TPBroker スケルトンを出力します。

PATH 環境変数に COBOL adapter for TPBroker の idl2cobol が格納されているディレクトリを設定してください。

● -template

雛形クラスの COBOL ソースを出力します。

● -format 1|2

～＜符号なし整数＞ ((1 | 2))《1》

雛形クラスを，形式 1 で出力するか形式 2 で出力するかを指定します。省略した場合は形式 1 が設定されます。

形式 1 および形式 2 は，TSC ユーザオブジェクトに対してユーザが実装する副プログラムの形式です。詳細は，7 章の「ABC_TSCfactimpl（COBOL）」を参照してください。

ユーザオペレーション内で TSCContext または TSCThread を使用する場合，形式 2 を指定してください。-template オプションを指定しない場合でも，形式 2 を使用するときは必ず -format オプションを使用してください。

● -TSCspxy

セション用 TSC ユーザプロキシを持つクライアント部分を生成します。-TSCspxy オプションを指定した場合は，通常の TSC ユーザプロキシは生成されません。

● -TSCroot_dir **パス名**

～＜文字列＞

ソースファイルを出力するディレクトリを指定します。ディレクトリがない場合は，ディレクトリを作成します。

● -root_dir **パス名**

～＜文字列＞

idl2cobol コマンドで出力されるソースファイルを出力するディレクトリを指定します。

-idl2cobol オプションを指定しない場合は，-root_dir オプションは無視されます。

● -TSCno_proxy

TSC ユーザプロキシの生成をしません。

● -TSCno_skel

TSC ユーザスケルトンの生成をしません。

● -TSCidl2cblfix **フラグ**

～＜4 けたの 2 進数字＞《0000》

boolean 型データの送受信時に，COBOL 固有の値を，C++ および Java に共通する値に変換するかどうかを指定します。省略した場合は "0000"（変換しない）が設定されます。

"0001" を指定すると，生成したトランザクションフレームで boolean 型データを送受信するときに，COBOL 固有の値を C++ および Java に共通する値に変換します。次に示すアプリケーションプログラムと boolean 型のデータを送受信する場合は "0001" を指定してください。

- C++ または Java で作成された OTM のアプリケーションプログラム
- OTM - Connector for ORB を経由して接続する ORB クライアントアプリケーション

指定しない場合，または "0000" を指定した場合は，boolean 型データの送受信時に値を

変換しません。

また，COBOL で作成された OTM のアプリケーションプログラムと boolean 型のデータを送受信する場合は，通信相手となるアプリケーションプログラムの作成時に -TSCidl2cblfix オプションに指定した値と同じ値を指定してください。

● -I **ディレクトリ** [: **ディレクトリ** ...]

～＜文字列＞

インクルードファイルのサーチパスを指定します。複数指定する場合，UNIX の場合は：（コロン），Windows の場合は；（セミコロン）で区切るか，または -I オプションを複数回指定してください。-I オプションと -idl2cobol オプションを指定した場合，idl2cobol コマンドにも同じサーチパスが使用されます。

● -A

内部で osagent を開始しません。ローカルホスト内ですでに osagent が開始されている場合に -A オプションを指定してください。

コマンド引数

● IDL **ファイル名称**

入力の IDL ファイル名称を指定します。-h オプションを指定する場合だけ省略できます。IDL ファイル名称は必ずコマンドラインの最後に指定してください。また，IDL ファイル名称の拡張子は必ず ".idl" にしてください。

戻り値

このコマンドは次に示す戻り値をシェルに返してから，処理を終了します。

戻り値	意味
0	正常終了しました。 メッセージ KFOT70008-I が出力された場合，および -h オプションを指定した場合も 0 が返ります。
0 以外	コマンド処理中にエラーが発生したために異常終了しました。 -idl2cobol オプションを指定した場合は，idl2cobol コマンド処理中にエラーが発生したときも 0 以外を返します。出力されたメッセージに従って対策したあと，再度，コマンドを入力してください。

実行条件

● TSCDIR 環境変数を設定してください。

● TPBroker の共用ライブラリを使用できるようにしてください。

● PATH 環境変数に TPBroker の osagent，idl2ir，および irep が格納されているディレクトリを設定してください。Cosminexus TPBroker for Java の ORB Version 4 の

ディレクトリは設定しないでください。

注意事項

- オプションとコマンドオプション引数との間には，必ず空白を入れてください。
- OTM で使用できない定義が含まれている場合，不正な動作をする，または不正なファイルが出力されるときがあります。
- -idl2cobol オプションを指定しないで idl2cobol コマンドを実行する場合，idl2cobol コマンドで出力される，クライアント部分のファイルに付加する文字列，およびサーバ部分のファイルに付加する文字列は，tscidl2cbl コマンドで指定した文字列（または省略時の文字列）と同じにしてください。
- tscidl2cbl コマンドの実行中に，内部で使用している TPBroker のメッセージが出力される場合があります。
- オプションに必要なコマンドオプション引数が指定されていない場合は省略値が設定されます。

tscidl2cpp（トランザクションフレームの出力（C++））

形式

```
tscidl2cpp [-h] | [-TSCclient_ext文字列] [-TSCserver_ext 文字列]
    [-TSCtemplate_ext 文字列] [-TSCsession_ext 文字列]
    [-TSChdr_suffix拡張子名] [-TSCsrc_suffix拡張子名]
    [-client_ext文字列] [-server_ext文字列]
    [-hdr_suffix拡張子名] [-src_suffix拡張子名]
    [-idl2cpp] [-template] [-TSCspxy]
    [-TSCexport 文字列] [-TSCexport_skel 文字列]
    [-export 文字列] [-export_skel 文字列]
    [-TSCroot_dir パス名] [-TSCsrc_dir パス名] [-TSChdr_dir パス名]
    [-root_dir パス名] [-src_dir パス名] [-hdr_dir パス名]
    [-TSCno_proxy] [-TSCno_skel] [-TSCidl2cppfix フラグ]
    [[-I ディレクトリ[:ディレクトリ...]]...]
    [-A] IDLファイル名称
```

機能

トランザクションフレームジェネレータです。ユーザ定義 IDL インタフェース依存クラスなどの C++ ソースを出力します。

オプション

● -h

ヘルプメッセージを出力します。ファイルの読み込みおよび生成はしません。

● -TSCclient_ext **文字列**

～＜文字列＞《_TSC_c》

クライアント部分を出力するファイル名称に付加する文字列を指定します。省略した場合は "_TSC_c" が設定されます。

● -TSCserver_ext **文字列**

～＜文字列＞《_TSC_s》

サーバ部分を出力するファイル名称に付加する文字列を指定します。省略した場合は "_TSC_s" が設定されます。

● -TSCtemplate_ext **文字列**

～＜文字列＞《_TSC_t》

雛形部分を出力するファイル名称に付加する文字列を指定します。省略した場合は "_TSC_t" が設定されます。

-template オプションを指定しない場合は，-TSCtemplate_ext オプションは無視されます。

● -TSCsession_ext **文字列**

～＜文字列＞《_TSC_p**》**

セション用 TSC ユーザプロキシ部分を出力するファイル名称に付加する文字列を指定します。省略した場合は "_TSC_p" が設定されます。

-TSCspxy オプションを指定しない場合は，-TSCsession_ext オプションは無視されます。

● -TSChdr_suffix **拡張子名**

～＜文字列＞《hh**》**

ヘッダファイルの拡張子を指定します。省略した場合は "hh" が設定されます。

● -TSCsrc_suffix **拡張子名**

～＜文字列＞《cc**》**

ソースファイルの拡張子を指定します。省略した場合は "cc" が設定されます。

● -client_ext **文字列**

～＜文字列＞《_c**》**

idl2cpp コマンドで出力されるクライアント部分のファイルに付加する文字列を指定します。省略した場合は "_c" が設定されます。

idl2cpp コマンドで出力されるクライアント部分のファイルに付加する文字列を変更する場合は，-idl2cpp オプションを指定しないときでも必ず -client_ext オプションを指定してください。

● -server_ext **文字列**

～＜文字列＞《_s**》**

idl2cpp コマンドで出力されるサーバ部分のファイルに付加する名称を指定します。省略した場合は "_s" が設定されます。

idl2cpp コマンドで出力されるサーバ部分のファイルに付加する文字列を変更する場合は，-idl2cpp オプションを指定しないときでも必ず -server_ext オプションを指定してください。

● -hdr_suffix **拡張子名**

～＜文字列＞《.hh**》**

idl2cpp コマンドで出力されるヘッダファイルのサフィックス名称を指定します。省略した場合は -TSChdr_suffix オプションで指定する拡張子名が使用されます。どちらも指定されていない場合は ".hh" が設定されます。

● -src_suffix **拡張子名**

～＜文字列＞《.cc》

idl2cpp コマンドで出力されるソースファイルのサフィックス名称を指定します。省略した場合は -TSChdr_suffix オプションで指定する拡張子名が使用されます。どちらも指定されていない場合は ".cc" が設定されます。

● -idl2cpp

内部で TPBroker のコマンドを実行して，TPBroker スタブおよび TPBroker スケルトンを出力します。

PATH 環境変数に TPBroker の idl2cpp が格納されているディレクトリを設定してください。

● -template

雛形クラスの C++ ソースを出力します。

● -TSCspxy

セション用 TSC ユーザプロキシを持つクライアント部分を生成します。-TSCspxy オプションを指定した場合は，通常の TSC ユーザプロキシは生成されません。

● -TSCexport **文字列**

～＜文字列＞

TSC ユーザプロキシにエクスポートタグを指定します。-TSCexport オプションは，Windows で DLL 内のクラスにアクセスするための宣言に使用します。

● -TSCexport_skel **文字列**

～＜文字列＞

TSC ユーザプロキシ，TSC ユーザスケルトン，TSC ユーザアクセプタ，TSC ユーザオブジェクトファクトリ，および TSC ユーザオブジェクトにエクスポートタグを指定します。-TSCexport_skel オプションは Windows で DLL 内のクラスにアクセスするための宣言に使用します。

-template オプションを指定しない場合は，TSC ユーザオブジェクトファクトリおよび TSC ユーザオブジェクトへの指定は無視されます。

● -export **文字列**

～＜文字列＞

idl2cpp コマンドで出力されるスタブにエクスポートタグを指定します。-export オプションは Windows で DLL 内のクラスにアクセスするための宣言に使用します。

-idl2cpp オプションを指定しない場合は，-export オプションは無視されます。

● -export_skel **文字列**

～＜文字列＞

idl2cpp コマンドで出力されるスタブおよびスケルトンにエクスポートタグを指定します。-export_skel オプションは Windows で DLL 内のクラスにアクセスするための宣言に使用します。

-idl2cpp オプションを指定しない場合は，-export_skel オプションは無視されます。

● -TSCroot_dir **パス名**

～＜文字列＞

ソースファイルおよびヘッダファイルを出力するディレクトリを指定します。指定したディレクトリがない場合は，ディレクトリを作成します。

出力先を変更した場合，出力先ディレクトリをインクルードファイルのサーチパスに指定してください。

● -TSCsrc_dir **パス名**

～＜文字列＞

ソースファイルを出力するディレクトリを指定します。指定したディレクトリがない場合は，ディレクトリを作成します。

● -TSChdr_dir **パス名**

～＜文字列＞

ヘッダファイルを出力するディレクトリを指定します。指定したディレクトリがない場合は，ディレクトリを作成します。

出力先を変更した場合，出力ファイルのコンパイル時に，出力先ディレクトリをインクルードファイルのサーチパスに指定してください。

● -root_dir **パス名**

～＜文字列＞

idl2cpp コマンドで出力されるソースファイルおよびヘッダファイルを出力するディレクトリを指定します。

出力先を変更した場合，依存関係を修正し，出力先ディレクトリをインクルードファイルのサーチパスに指定する必要があります。

-idl2cpp オプションを指定しない場合は，-root_dir オプションは無視されます。

● -src_dir **パス名**

～＜文字列＞

idl2cpp コマンドで出力されるソースファイルを出力するディレクトリを指定します。

-idl2cpp オプションを指定しない場合は，-src_dir オプションは無視されます。

● -hdr_dir **パス名**

～＜文字列＞

idl2cpp コマンドで出力されるヘッダファイルを出力するディレクトリを指定します。

出力先を変更した場合，出力ファイルのコンパイル時に，出力先ディレクトリをインクルードファイルのサーチパスに指定してください。

-idl2cpp オプションを指定しない場合は，-hdr_dir オプションは無視されます。

● -TSCno_proxy

TSC ユーザプロキシの生成をしません。

● -TSCno_skel

TSC ユーザスケルトンの生成をしません。

● -TSCidl2cppfix **フラグ**

～＜4 けたの 2 進数字＞《0000》

トランザクションフレームの出力時に使用する領域の解放について指定します。省略した場合，または 0000 が指定された場合，該当する領域を解放しません。

上位 1 けた目～ 2 けた目
　"00" を指定してください。

上位 3 けた目
　上位 3 けた目に "1" を指定すると，OUT 属性および INOUT 属性で指定した引数の領域割り当て後に例外が発生したとき，領域が TSC ユーザスケルトンで解放されます。

上位 4 けた目
　上位 4 けた目に "1" を指定すると，INOUT 属性の string を使用するとき，IN 属性として使用した領域が TSC ユーザプロキシで解放されます。

両方とも指定する場合は，"0011" を指定してください。

● -I ディレクトリ [: ディレクトリ ...]

～＜文字列＞

インクルードファイルのサーチパスを指定します。複数指定する場合，UNIX の場合は：（コロン），Windows の場合は；（セミコロン）で区切るか，または -I オプションを複数回指定してください。-I オプションと -idl2cpp オプションを指定した場合，idl2cpp コマンドにも同じサーチパスが使用されます。

● -A

内部で osagent を開始しません。ローカルホスト内ですでに osagent が開始されている場合に -A オプションを指定してください。

コマンド引数

● IDL ファイル名称

入力の IDL ファイル名称を指定します。-h オプションを指定する場合だけ省略できます。IDL ファイル名称は必ずコマンドラインの最後に指定してください。また，IDL ファイル名称の拡張子は必ず ".idl" にしてください。

戻り値

このコマンドは次に示す戻り値をシェルに返してから，処理を終了します。

戻り値	意味
0	正常終了しました。 メッセージ KFOT70008-I が出力された場合，および -h オプションを指定した場合も 0 が返ります。
0 以外	コマンド処理中にエラーが発生したために異常終了しました。 -idl2cpp オプションを指定した場合は，idl2cpp コマンド処理中にエラーが発生したときも 0 以外を返します。出力されたメッセージに従って対策したあと，再度，コマンドを入力してください。

実行条件

● TSCDIR 環境変数を設定してください。

● TPBroker の共用ライブラリを使用できるようにしてください。

● PATH 環境変数に TPBroker の osagent，idl2ir，および irep が格納されているディレクトリを設定してください。Cosminexus TPBroker for Java の ORB Version 4 のディレクトリは設定しないでください。

注意事項

- オプションとコマンドオプション引数との間には，必ず空白を入れてください。
- OTM で使用できない定義が含まれている場合，不正な動作をする，または不正なファイルが出力されるときがあります。
- -idl2cpp オプションを指定しないで idl2cpp コマンドを実行する場合，idl2cpp コマンドで出力される，クライアント部分のファイルに付加する文字列，サーバ部分のファイルに付加する文字列，ヘッダファイルのサフィックス名称，およびソースファイルのサフィックス名称は，tscidl2cpp コマンドで指定した文字列（または省略時の文字列）と同じにしてください。
- tscidl2cpp コマンドの実行中に，内部で使用している TPBroker のメッセージが出力される場合があります。
- オプションに必要なコマンドオプション引数が指定されていない場合は省略値が設定されます。

tscidl2j（トランザクションフレームの出力（Java））

形式

```
tscidl2j  [-h] | [-idl2java] [-template] [-TSCspxy]
          [-package パッケージ名] [-TSCroot_dir パス名]
          [-root_dir パス名] [-TSCno_proxy] [-TSCno_skel]
          [[-Iディレクトリ[:ディレクトリ...]]...]
          [-A] [-vbj4] IDLファイル名称
```

機能

トランザクションフレームジェネレータです。ユーザ定義 IDL インタフェース依存クラスなどの Java ソースを生成します。

オプション

● -h

ヘルプメッセージを出力します。ファイルの読み込みおよび生成はしません。

● -idl2java

内部で TPBroker のコマンドを実行して，TPBroker スタブおよび TPBroker スケルトンを出力します。

PATH 環境変数に設定するディレクトリは，-vbj4 オプションの指定の有無によって異なります。-vbj4 オプションを指定する場合は，Cosminexus TPBroker for Java の ORB Version 4 の idl2java が格納されているディレクトリを設定してください。-vbj4 オプションを指定しない場合は，ORB Version 3 の idl2java が格納されているディレクトリを設定してください。

● -template

雛形クラスの Java ソースを出力します。

● -TSCspxy

セション用 TSC ユーザプロキシを持つクライアント部分を生成します。-TSCspxy オプションを指定した場合は，通常の TSC ユーザプロキシは生成されません。

● -package **パッケージ名**

～ < 文字列 >

IDL ファイルに定義したパッケージ名の前に，-package オプションで指定したパッケージ名を付加します。指定したパッケージ名のディレクトリがない場合は，ディレクトリ

が作成されます。指定したパッケージ名のディレクトリがある場合は，そのディレクトリの内容が更新されます。なお，-package オプションを指定しなくても，同一ユーザ定義 IDL インタフェースの相手とは通信できます。

-idl2java オプションを指定しないで -package オプションを指定した場合は，idl2java コマンドでも，-package オプションの指定と同じパッケージ名を，同じ IDL ファイルに対して指定してください。

● -TSCroot_dir **パス名**

～＜文字列＞

ソースファイルを出力するディレクトリを指定します。指定したディレクトリがない場合は，ディレクトリを作成します。

● -root_dir **パス名**

～＜文字列＞

idl2java コマンドで出力されるソースファイルを出力するディレクトリを指定します。

-idl2java オプションを指定しない場合は，-root_dir オプションは無視されます。

● -TSCno_proxy

TSC ユーザプロキシの生成をしません。

● -TSCno_skel

TSC ユーザスケルトンおよび TSC アクセプタの生成をしません。

● -I **ディレクトリ** [: **ディレクトリ** ...]

～＜文字列＞

インクルードファイルのサーチパスを指定します。複数指定する場合，UNIX の場合は：（コロン），Windows の場合は；（セミコロン）で区切るか，または -I オプションを複数回指定してください。-I オプションと -idl2java オプションを指定した場合，idl2java コマンドにも同じサーチパスが使用されます。

● -A

内部で osagent を開始しません。ローカルホスト内ですでに osagent が開始されている場合に -A オプションを指定してください。

● -vbj4

Cosminexus TPBroker for Java の ORB Version 4 に対応するトランザクションフレームを出力します。Cosminexus TPBroker for Java の ORB Version 4 に対応するアプリケーションプログラムを作成する場合に指定してください。

-vbj4 オプションを指定する場合は，PATH 環境変数に Cosminexus TPBroker for Java の ORB Version 4 の osagent および irep が格納されているディレクトリを設定してください。

コマンド引数

● IDL ファイル名称

入力の IDL ファイル名称を指定します。-h オプションを指定する場合だけ省略できます。IDL ファイル名称は必ずコマンドラインの最後に指定してください。また，IDL ファイル名称の拡張子は必ず ".idl" にしてください。

戻り値

このコマンドは次に示す戻り値をシェルに返してから，処理を終了します。

戻り値	意味
0	正常終了しました。 メッセージ KFOT70008-I が出力された場合，および -h オプションを指定した場合も 0 が返ります。
0 以外	コマンド処理中にエラーが発生したために異常終了しました。 -idl2java オプションを指定した場合は，idl2java コマンド処理中にエラーが発生したときも 0 以外を返します。出力されたメッセージに従って対策したあと，再度，コマンドを入力してください。

実行条件

● TSCDIR 環境変数を設定してください。

● TPBroker の共用ライブラリを使用できるようにしてください。

● PATH 環境変数に設定するディレクトリは，-vbj4 オプションの指定の有無によって異なります。

- -vbj4 オプションを指定する場合
  Cosminexus TPBroker for Java の ORB Version 4 の osagent，idl2ir，および irep が格納されているディレクトリを設定してください。ORB Version 3 のディレクトリは設定しないでください。
- -vbj4 オプションを指定しない場合
  ORB Version 3 の osagent，idl2ir，および irep が格納されているディレクトリを設定してください。Cosminexus TPBroker for Java の ORB Version 4 のディレクトリは設定しないでください。

注意事項

● OTM で使用できない定義が含まれている場合，不正な動作をする，または不正なファイルが出力されるときがあります。

- tscidl2j コマンドの実行中に，内部で利用している TPBroker のメッセージが出力される場合があります。
- オプションに必要なコマンドオプション引数が指定されていない場合は省略値が設定されます。
- Cosminexus TPBroker for Java の ORB Version 4 を使用する環境で tscidl2j コマンドを実行すると，TPBroker が "IDL 名称 .rollback" というファイルを作成します。作成する場所は，Java のシステムプロパティで取得されるホームディレクトリ下です。このファイルの有無は，アプリケーションプログラムの作成および動作には影響しないため，必要に応じて削除してください。ファイル名に設定される IDL 名称は，tscidl2j コマンドの引数に指定した IDL ファイル名からディレクトリ名と拡張子（.idl）を除いた名称です。例えば，"tscidl2j -vbj4 ABCfile.idl" を実行した場合，ABCfile.rollback という名称のファイルが作成されます。
- Cosminexus TPBroker for Java の ORB Version 4 に対応するアプリケーションプログラムを作成するときに，tscidl2j に -idl2java オプションを指定しない場合は，idl2java コマンドを別途実行する必要があります。この場合は，idl2java コマンドに -boa オプションおよび -no_narrow_compliance オプションを指定してください。
- 同一ホスト内で同一名称の IDL ファイルに対する tscidl2j，tscidl2cpp，tscidl2cbl が実行されている間は，-vbj4 オプションを指定した tscidl2j を実行しないでください。同一名称の IDL ファイルで，それぞれ定義内容が異なる場合，不正な内容のファイルが出力される場合があります。
- Cosminexus TPBroker for Java の ORB Version 4 を使用しない環境で，-vbj4 オプションを指定して tscidl2j を実行した場合，tscidl2j が正常に終了しても，ファイルが出力されなかったり，不正な内容のファイルが出力されたりする場合があります。

付録

付録 A　エラーコード一覧

エラーコードは，TSCSystemException クラスの派生クラスに対応する値です。

C++ または Java でアプリケーションプログラムを作成する場合のエラーコード一覧を表 A-1 に，COBOL の場合のエラーコード一覧を表 A-2 に示します。

表 A-1　エラーコード一覧（C++，Java）

値	エラーコード	障害内容
1	BAD_PARAM	無効パラメタが渡されました。
2	NO_MEMORY	動的メモリの割り当て障害が発生しました。
3	COMM_FAILURE	通信障害が発生しました。
4	NO_PERMISSION	許可されていないオペレーションを実行しようとしました。
5	INTERNAL	ORB 内部エラーが発生しました。
6	MARSHAL	スタブ，スケルトンで CDR マーシャルに失敗しました。
7	INITIALIZE	ORB 初期化障害が発生しました。
8	NO_IMPLEMENT	オペレーションの実装が使用できません。
9	BAD_OPERATION	オペレーションが無効です。
10	NO_RESOURCES	リクエストを処理するための資源が不足しています。
11	NO_RESPONSE	リクエストに対する応答がありません。
12	BAD_INV_ORDER	ルーチン呼び出しの順番が不正です。
13	TRANSIENT	一時的な障害が発生しました。
14	OBJECT_NOT_EXIST	該当するオブジェクトがありません。
15	UNKNOWN	未知の例外が発生しました。
16	INV_OBJREF	無効なオブジェクトリファレンスが指定されました。
17	IMP_LIMIT	実装の制限を超えました。
18	BAD_TYPECODE	タイプコードが不正です。
19	PERSIST_STORE	パーシステントストレージに障害が発生しました。
20	FREE_MEM	メモリの解放に失敗しました。
21	INV_IDENT	識別子の構文が不正です。
22	INV_FLAG	不正なフラグが指定されました。
23	INTF_REPOS	インタフェースリポジトリへのアクセス中に障害が発生しました。
24	BAD_CONTEXT	コンテキストオブジェクトの処理中に障害が発生しました。

値	エラーコード	障害内容
25	OBJ_ADAPTER	オブジェクトアダプタが障害を検出しました。
26	DATA_CONVERSION	データ変換に失敗しました。

表 A-2　エラーコード一覧（COBOL）

値	エラーコード	障害内容
1	TSCSysExcept_ERR_BAD_PARAM	無効な引数を指定し副プログラムを呼び出しました。
2	TSCSysExcept_ERR_NO_MEMORY	動的メモリの割り当て障害が発生しました。
3	TSCSysExcept_ERR_COMM_FAILURE	通信障害が発生しました。
4	TSCSysExcept_ERR_NO_PERMISSION	許可されていない副プログラムを呼び出しました。
5	TSCSysExcept_ERR_INTERNAL	ORB 内部エラーが発生しました。
6	TSCSysExcept_ERR_MARSHAL	スタブ，スケルトンで CDR マーシャルに失敗しました。
7	TSCSysExcept_ERR_INITIALIZE	ORB 初期化障害が発生しました。
8	TSCSysExcept_ERR_NO_IMPLEMENT	オペレーションの実装が使用できません。
9	TSCSysExcept_ERR_BAD_OPERATION	オペレーションが無効です。
10	TSCSysExcept_ERR_NO_RESOURCES	リクエストを処理するための資源が不足しています。
11	TSCSysExcept_ERR_NO_RESPONSE	リクエストに対する応答がありません。
12	TSCSysExcept_ERR_BAD_INV_ORDER	副プログラムの発行順序が不正です。
13	TSCSysExcept_ERR_TRANSIENT	一時的な障害が発生しました。
14	TSCSysExcept_ERR_NOT_EXIST	該当するオブジェクトがありません。
15	TSCSysExcept_ERR_UNKNOWN	未知の例外が発生しました。
16	TSCSysExcept_ERR_INV_OBJREF	無効なオブジェクトリファレンスが指定されました。
17	TSCSysExcept_ERR_IMP_LIMIT	実装の制限を超えました。
18	TSCSysExcept_ERR_BAD_TYPECODE	タイプコードが不正です。
19	TSCSysExcept_ERR_PERSIST_STORE	パーシステントストレージに障害が発生しました。
20	TSCSysExcept_ERR_FREE_MEM	メモリの解放に失敗しました。
21	TSCSysExcept_ERR_INV_IDENT	識別子が不正です。
22	TSCSysExcept_ERR_INV_FLAG	不正なフラグが指定されました。
23	TSCSysExcept_ERR_INTF_REPOS	インタフェースリポジトリへのアクセス中に障害が発生しました。
24	TSCSysExcept_ERR_BAD_CONTEXT	コンテキストオブジェクトの処理中に障害が発生しました。
25	TSCSysExcept_ERR_OBJ_ADAPTER	オブジェクトアダプタが障害を検出しました。
26	TSCSysExcept_ERR_DATA_CONV	データ変換に失敗しました。

付録 B　場所コード一覧

場所コードは，OTM で障害が発生した場所を示す値です。

C++ または Java でアプリケーションプログラムを作成する場合の場所コード一覧を表 B-1 に，COBOL の場合の場所コード一覧を表 B-2 に示します。

表 B-1　場所コード一覧（C++，Java）

値	場所コード	場所
1	PLACE_CODE_USER_AP	ユーザアプリケーション
2	PLACE_CODE_SERV	OTM のサーバ機能部分
3	PLACE_CODE_DAEMON	TSC デーモン
4	PLACE_CODE_CLNT	OTM のクライアント機能部分
5	PLACE_CODE_CLNT_REG	TSC レギュレータ
6	PLACE_CODE_STUB	TSC ユーザプロキシ（スタブ）
7	PLACE_CODE_SKELTON	TSC ユーザスケルトン（スケルトン）
8	PLACE_CODE_ORBGW	TSCORB コネクタ

表 B-2　場所コード一覧（COBOL）

値	条件名	場所
1	TSCSysExcept_PLACE_USER_AP	ユーザアプリケーション
2	TSCSysExcept_PLACE_SERV	OTM のサーバ機能部分
3	TSCSysExcept_PLACE_DAEMON	TSC デーモン
4	TSCSysExcept_PLACE_CLNT	OTM のクライアント機能部分
5	TSCSysExcept_PLACE_CLNT_REG	TSC レギュレータ
6	TSCSysExcept_PLACE_STUB	スタブ
7	TSCSysExcept_PLACE_SKELTON	スケルトン
8	TSCSysExcept_PLACE_ORBGW	TSCORB コネクタ

付録 C　完了状態一覧

完了状態は，障害が発生したときにメソッド（副プログラム）の呼び出しが完了しているかどうかを示す値です。

C++ または Java でアプリケーションプログラムを作成する場合の完了状態一覧を表 C-1 に，COBOL の場合の完了状態一覧を表 C-2 に示します。

表 C-1　完了状態一覧（C++，Java）

値	完了状態	説明
-1	COMPLETED_NO	メソッド呼び出しが完了していません。
0	COMPLETED_MAYBE	メソッド呼び出しの完了状態を決定できません。
1	COMPLETED_YES	メソッド呼び出しが完了しています。

表 C-2　完了状態一覧（COBOL）

値	完了状態	説明
-1	TSCSysExcept_COMPLETED_NO	副プログラム呼び出しが完了していません。
0	TSCSysExcept_COMPLETED_MAYBE	副プログラム呼び出しの完了状態を決定できません。
1	TSCSysExcept_COMPLETED_YES	副プログラム呼び出しの処理が完了しています。

付録 D　内容コード一覧

OTM の例外で使用する内容コードについて，定数値の順に説明します。なお，角括弧内［］は該当する処理または個所を表します。該当する処理がメソッドの場合，クラス名とメソッド名を "::" で区切って示します。例えば，"TSCProxyObject::_TSCTimeOut" は，TSCProxyObject クラスの _TSCTimeOut メソッドを表します。

付録 D.1　内容コード 1000 ～ 1999

1001　INVALID_TIMEOUT

障害内容

［TSCProxyObject::_TSCTimeOut］

［TSCSessionProxy::_TSCTimeOut］

引数に指定したタイムアウト値が不正です。

開発時の対策

TSCProxyObject::_TSCTimeOut，または TSCSessionProxy::_TSCTimeOut の引数に，規定の監視時間を指定してください。

1002　INVALID_RT_ACPT_NAME

障害内容

［TSCRootAcceptor::activate］

引数に指定した TSC ルートアクセプタ登録名称が不正です。

開発時の対策

TSCRootAcceptor::activate の引数に，規定の TSC ルートアクセプタ登録名称を指定してください。

1003　INVALID_PARALLEL_COUNT

障害内容

［TSCRootAcceptor::setParallelCount］

引数に指定したパラレルカウント（常駐するスレッド数）が不正です。

開発時の対策

TSCRootAcceptor::setParallelCount の引数に，規定のパラレルカウント値を指定してください。

1004　INVALID_ACPT_NAME

障害内容

［TSCAcceptor のコンストラクタ］

［TSCAcceptor の派生クラスのコンストラクタ］

［TSCSessionProxy のコンストラクタ］
［TSCSessionProxy の派生クラスのコンストラクタ］
引数に指定した TSC アクセプタ名称が不正です。

開発時の対策
TSCAcceptor，TSCAcceptor の派生クラス，TSCSessionProxy，または TSCSessionProxy の派生クラスのコンストラクタの引数に，規定の長さ（1 ～ 31 文字）の TSC アクセプタ名称を指定してください。

1005　OBJ_FACT_IS_NULL

障害内容
［TSCAcceptor のコンストラクタ］
［TSCAcceptor の派生クラスのコンストラクタ］
引数に指定した TSCObjectFactory が null です。

開発時の対策
TSCAcceptor，または TSCAcceptor の派生クラスのコンストラクタの引数に，new オペレータで生成した TSCObjectFactory を指定してください。

1006　ACPT_IS_NULL

障害内容
［TSCAcceptor のコンストラクタ］
［TSCAcceptor の派生クラスのコンストラクタ］
引数に指定した TSCAcceptor が null です。

開発時の対策
TSCRootAcceptor::registerAcceptor の引数に，new オペレータで生成した TSCAcceptor を指定してください。

1007　INVALID_ACPT_REGID

障害内容
［TSCRootAcceptor::cancelAcceptor］
引数に指定した TSCAcceptor の登録識別子が不正です。

開発時の対策
TSCRootAcceptor::cancelAcceptor の引数に，TSCRootAcceptor::registerAcceptor の戻り値として取得し，かつ，TSCAcceptor に登録されている TSC ユーザアクセプタの登録識別子を指定してください。

1008　INVALID_TSCID

障害内容
［TSCDomain のコンストラクタ］

引数に指定した TSC 識別子が不正です。

開発時の対策

TSCDomain のコンストラクタの引数に，規定の TSC 識別子を指定してください。

1009　INVALID_DOMAIN_NAME

障害内容

［TSCDomain のコンストラクタ］

引数に指定した TSC ドメイン名称が不正です。

開発時の対策

TSCDomain のコンストラクタの引数に，規定の TSC ドメイン名称を指定してください。

1010　INVALID_OP_PARAM

障害内容

［オブジェクトのユーザメソッド］

オブジェクトのユーザメソッドを呼び出すときの引数が不正です。ただし，オブジェクトのユーザメソッドの呼び出しは完了している可能性があります。

［スタブ（場所コード：PLACE_CODE_STUB)］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトを呼び出そうとしましたが，サーバ側に渡す in 引数または inout 引数が不正です。TSC ユーザオブジェクトのユーザメソッドは呼び出されていません。

［スケルトン（場所コード：PLACE_CODE_SKELTON)］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトを呼び出し，そのメソッドの処理は完了しましたが，クライアント側に戻す out 引数，inout 引数，または戻り値が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。さらに，TSC ユーザオブジェクトのユーザメソッドを呼び出すときの引数，またはユーザメソッドから戻るときの引数および戻り値を，IDL から各プログラミング言語へのマッピング，および各プログラミング言語での引数と戻り値の規則に従うようにしてください。

運用時の対策

クライアントアプリケーション，またはサーバアプリケーションがメソッド呼び出しの規則に違反している可能性が高いです。IDL から各プログラミング言語へのマッピング，および各プログラミング言語での引数と戻り値の規則に従うように，アプリケーションプログラムを修正してください。

1011　SERV_IS_NULL

障害内容

［TSCRootAcceptor::create］

引数に指定した TSCServer が null です。

開発時の対策

TSCRootAcceptor::create の引数に，TSCAdm::getTSCServer で取得した TSCServer を指定してください。

1012　CLNT_IS_NULL

障害内容

［TSCProxyObject のコンストラクタ］

［TSCProxyObject の派生クラスのコンストラクタ］

［TSCSessionProxy のコンストラクタ］

［TSCSessionProxy の派生クラスのコンストラクタ］

引数に指定した TSCClient が null です。

開発時の対策

TSCProxyObject，TSCProxyObject の派生クラス，TSCSessionProxy，または TSCSessionProxy の派生クラスのコンストラクタに，TSCAdm の getTSCClient で取得した TSCClient を指定してください。

1013　ORB_IS_NULL

障害内容

［TSCAdm::initServer］

［TSCAdm::initClient］

引数に指定した CORBA::ORB が null です。

開発時の対策

［TSCAdm::initServer］

TSCAdm::initServer の引数に，ORB の init で得た CORBA::ORB を指定してください。

［TSCAdm::initClient］

TSCAdm::initClient の引数に，ORB の init で得た CORBA::ORB を指定してください。

1015　DOMAIN_IS_NULL

障害内容

［TSCAdm::getTSCClient］

［TSCAdm::getTSCServer］

引数に指定した TSCDomain が null です。

開発時の対策

［TSCAdm::getTSCServer］

TSCAdm::getTSCServer の引数に，new オペレータで生成した TSCDomain を指定してください。

［TSCAdm::getTSCClient］

TSCAdm::getTSCClient の引数に，new オペレータで生成した TSCDomain を指定してください。

1016　INVALID_REQUEST_WAY

障害内容

［TSCAdm::getTSCClient］

引数に指定した接続経路が不正です。

開発時の対策

TSCAdm::getTSCClient の引数に，規定の接続経路を指定してください。

1017　INVALID_PRIORITY

障害内容

［TSCProxyObject::_TSCPriority］

［TSCSessionProxy::_TSCPriority］

引数に指定したプライオリティ値（メソッド呼び出し時の優先順位）が不正です。

開発時の対策

TSCProxyObject::_TSCPriority，または TSCSessionProxy::_TSCPriority の引数に，規定のプライオリティ値を指定してください。

1018　THREAD_FACT_IS_NULL

障害内容

［TSCRootAcceptor::create］

引数に指定した TSCThreadFactory が不正です。

開発時の対策

TSCRootAcceptor::create に，new オペレータで生成した TSCThreadFactory を指定してください。

1020　PROXY_IS_NULL

障害内容

［COBOL］

引数に指定した TSC ユーザプロキシのポインタが null です。

開発時の対策

生成した TSC ユーザプロキシのポインタを指定してください。

1021　OBJECT_IS_NULL

障害内容

［COBOL］

引数に指定した TSC ユーザオブジェクトのポインタが null です。

開発時の対策

生成した TSC ユーザオブジェクトのポインタを指定してください。

1022　INVALID_FLAG

障害内容

［TSCDomain-NEW］

引数に指定したフラグ値が不正です。

開発時の対策

TSCDomain-NEW 副プログラムの引数に，規定のフラグ値を指定してください。

1027　INVALID_WATCH_TIME

障害内容

［TSCWatchTime のコンストラクタ］

引数に指定した監視時間が不正です。

開発時の対策

TSCWatchTime のコンストラクタの引数に，規定の監視時間を指定してください。

1028　WATCH_TIME_IS_NULL

障害内容

［COBOL］

引数に指定した TSCWatchTime のポインタが null です。

開発時の対策

生成した TSCWatchTime オブジェクトのポインタを指定してください。

1029　INVALID_RETRY_REQUIREMENT

障害内容

［TSCAdm::getTSCClient］

TSCAdm::getTSCClient の引数に指定した，TSCDomain クラスのコンストラクタの TSC ドメイン名称，TSC 識別子，および接続経路（指定がない場合は，コマンド引数オプション -TSCRequestWay の指定値）を持つ接続対象が接続先情報ファイル中に含まれていないため，接続を実行できません。

開発時の対策

接続先情報ファイル中のレコードが接続対象となるよう TSCAdm::getTSCClient の引数を指定してください。マルチノードリトライ接続の対象となるための条件につ

いては，マニュアル「TPBroker Object Transaction Monitor　ユーザーズガイド」のマルチノードリトライ接続の接続対象に関する説明を参照してください。

運用時の対策

TSCAdm::getTSCClient に指定した TSC ドメイン名称（null が指定されている場合は任意），および接続経路に適応する接続対象が含まれる接続先情報ファイルを使用してください。

1038　INVALID_SESSION_INTERVAL

障害内容

［TSCSessionProxy::_TSCSessionInterval］

引数に指定したセション呼び出しインターバル監視時間が不正です。

開発時の対策

TSCSessionProxy::_TSCSessionInterval の引数に，規定のセション呼び出しインターバル監視時間を指定してください。

1047　INVALID_QUEUE_LENGTH

障害内容

スケジュール用キューの長さを指定するメソッドに指定したスケジュール用キューの長さが不正です。

開発時の対策

サーバアプリケーション内のメソッドで指定したスケジュール用キューの長さを見直し，規定のスケジュール用キューの長さを指定してください。

1998　TPBROKER_BAD_PARAM

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::BAD_PARAM の例外通知を受けました。

付録 D.2　内容コード 2000 ～ 2999

2001　MEM_ALLOC_FAILURE

障害内容

メモリ確保に失敗しました。

［オブジェクトのユーザメソッドの呼び出し］

TSC ユーザオブジェクトのユーザメソッドの処理が完了している可能性があります。

［その他のシステム提供メソッドの呼び出し］

システム提供メソッドの処理が完了している可能性があります。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

場所コードで示される個所でユーザメモリの確保に失敗しました。OTM のシステムを終了してください。その後，プロセス単位で確保できる最大ユーザメモリサイズを変更し，再度，システムを開始してください。

2998　TPBROKER_NO_MEMORY

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::NO_MEMORY の例外通知を受けました。

付録 D.3　内容コード 3000 ～ 3999

3004　SEND_CLNT_FAILURE

障害内容

TSC デーモンに直結するクライアントアプリケーションまたは TSC レギュレータと，TSC デーモン間の通信中に障害が発生しました。ただし，オブジェクトのユーザメソッドの呼び出しは完了している可能性があります。

開発時の対策

［TSC デーモンへの接続からのリトライ］

TSC デーモンに直結するクライアントアプリケーションまたは TSC レギュレータから TSC デーモンへの接続が切断されている可能性があります。再度，TSCAdm::getTSCClient の処理からやり直してください。

［ほかの TSC デーモンが開始済み］

ほかの TSC デーモンに対し，TSCAdm::getTSCClient の処理からやり直してください。

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［TSC デーモンの確認］

TSC デーモンが開始していることを確認してから，クライアントアプリケーションを再度，開始してください。

3005　SEND_THIN_CLNT_FAILURE

障害内容

クライアントアプリケーションと TSC レギュレータ間の通信中に障害が発生しました。ただし，オブジェクトのユーザメソッドの呼び出しは完了している可能性があります。

開発時の対策

［TSC レギュレータへの接続からのリトライ］

クライアントアプリケーションから TSC レギュレータへの接続が切断されている可能性があります。再度，TSCAdm::getTSCClient の処理からやり直してください。

［ほかの TSC デーモンが開始済み］

ほかの TSC デーモンに対し，TSCAdm::getTSCClient の処理からやり直してください。

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［TSC レギュレータおよび TSC デーモンの確認］

TSC レギュレータ，および TSC デーモンが開始していることを確認したあと，再度，クライアントアプリケーションを開始してください。

3006　SEND_SERV_FAILURE

障害内容

TSC デーモンとサーバアプリケーション間の通信中に障害が発生しました。ただし，オブジェクトのユーザメソッドの呼び出しは完了している可能性があります。

開発時の対策

［リトライ］

再度，オブジェクトを呼び出してください。ただし，前回のユーザメソッドの呼び出しは完了している可能性があります。

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［TSC ルートアクセプタの確認］

該当するインタフェース名称および TSC アクセプタ名称のオブジェクトを提供する，TSC ルートアクセプタが active 状態であることを確認し，再度，クライアントアプリケーションを開始してください。

［サーバアプリケーションの確認］

該当するインタフェースおよび TSC アクセプタ名称のオブジェクトを提供する，サーバアプリケーションプロセスが開始していることを確認したあと，再度，クライアントアプリケーションを開始してください。

3007　SEND_TSCD_FAILURE

障害内容

TSC デーモン間の通信中に障害が発生しました。ただし，オブジェクトのユーザメソッドの呼び出しは完了している可能性があります。

開発時の対策

［リトライ］

再度，オブジェクトを呼び出してください。ただし，前回のオブジェクトのユーザメソッドの呼び出しは完了している可能性があります。

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［TSC デーモンの確認］

該当するインタフェース名称および TSC アクセプタ名称のオブジェクトを提供しているサーバアプリケーションプロセスが接続している，同じ TSC ドメイン内の TSC デーモンが開始していることを確認してください。TSC デーモンが開始していない場合，TSC デーモンおよびサーバアプリケーションを開始したあと，クライアントアプリケーションを実行してください。

3009　BASIC_CONN_FAILURE

障害内容

［TSCAdm::initServer］

［TSCAdm::getTSCServer］

［TSCAdm::initClient］

［TSCAdm::getTSCClient］

通信先の tscd が見つかりませんでした。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［osagent の確認］

TPBroker の osagent プロセスが開始されているかどうかを確認してください。

［環境変数の確認］

osagent プロセス，OTM システムプロセス，およびアプリケーションプロセスの OSAGENT_PORT 環境変数が一致しているかどうかを確認してください。

［TSC デーモンおよび TSC レギュレータの確認］

接続または通信する TSC デーモンおよび TSC レギュレータが開始されているかどうかを確認してください。

［その他］
osagent プロセスを再開始してください。

3010　CONN_FAILURE

障害内容
通信路の接続に失敗しました。
［TSCAdm::initServer］
サーバアプリケーションの TSC デーモンへの登録要求の過程で障害が発生しました。その原因を次に示します。
- コマンドオプション引数 -TSCPort に，すでにほかのプロセスが使用しているポート番号を指定しています。
- コマンドオプション引数 -TSCPort に指定した値が不正です。

［TSCAdm::getTSCServer］
サーバアプリケーションと TSC デーモンの接続の過程で障害が発生しました。
［TSCAdm::initClient］
クライアントアプリケーションの TSC デーモンへの登録要求の過程で障害が発生しました。その原因を次に示します。
- コマンドオプション引数 -TSCPort に，すでにほかのプロセスが使用しているポート番号を指定しています。
- コマンドオプション引数 -TSCPort に指定した値が不正です。

［TSCAdm::getTSCClient］
クライアントアプリケーションと，TSC デーモンまたは TSC レギュレータの接続の過程で障害が発生しました。
［TSCAdm::releaseTSCServer］
サーバアプリケーションと TSC デーモンの接続切断の過程で障害が発生しました。
［TSCAdm::endServer］
サーバアプリケーションの TSC デーモンへの登録解除の過程で障害が発生しました。
［TSCAdm::releaseTSCClient］
クライアントアプリケーションと，TSC デーモンまたは TSC レギュレータの接続切断の過程で障害が発生しました。
［TSCAdm::endClient］
クライアントアプリケーションの TSC デーモンへの登録解除の過程で障害が発生しました。
［TSCRootAcceptor::acitvate］
TSCRootAcceptor の活性化（active 状態への遷移）の過程で通信障害がありました。
［TSCRootAcceptor::deactivate］
TSCRootAcceptor の非活性化（non-active 状態への遷移）の過程で通信障害がありました。

開発時の対策

［リトライ（TSCAdm::getTSCServer）］

再度，TSCAdm::getTSCServer からやり直してください。

［リトライ（TSCAdm::getTSCClient）］

再度，TSCAdm::getTSCClient からやり直してください。

［処理の終了（TSCAdm::releaseTSCServer）］

TSCAdm::releaseTSCServer が受け付けられる前に，障害が発生しました。OTM の処理を終了してください。

［処理の終了（TSCAdm::releaseTSCClient）］

TSCAdm::releaseTSCClient が受け付けられる前に，障害が発生しました。OTM の処理を終了してください。

［リトライ（TSCRootAcceptor::activate）］

再度，TSCAdm::getTSCServer からやり直してください。

［処理の終了（TSCRootAcceptor::deactivate）］

TSCRootAcceptor::deactivate が受け付けられる前に，障害が発生しました。OTM の処理を終了してください。

［処理の終了］

保守コード 1 および保守コード 2 を取得して，OTM の処理を終了してください。TSCRootAcceptor::deactivate でこの例外が発生し，完了状態が COMPLETED_MAYBE（0）の場合，TSCRootAcceptor::deactivate の処理が完了していないため，アプリケーションプロセスが異常終了する可能性があります。

運用時の対策

［最大コネクション数（ファイルディスクリプタ数）の確認］

プロセス単位で確立することができる最大コネクション数，または取得できるファイルディスクリプタ数を超えて，コネクションを確立しようとした可能性が高いです。OTM のシステムを終了してください。その後，プロセス単位で確立することができる最大コネクション数，または取得できるファイルディスクリプタ数の上限を変更し，再度，システムを開始してください。

［指定ポート番号の確認］

コマンドオプション引数 -TSCPort に指定したポート番号は，すでにほかのプロセスで使用されている可能性があります。コマンドオプション引数 -TSCPort に指定するポート番号をほかのプロセスが使用していないポート番号に変更し，再度，実行してください。

3011　INCOMPATIBLE_PROTOCOL

障害内容

OTM の通信プロトコル上で，バージョンの不一致がありました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処

理を終了してください。

運用時の対策

同じ TSC ドメインに含まれる OTM のシステムデーモン，またはアプリケーションプログラムの中で，バージョンの不一致があります。再度，確認してください。

3012　NOT_IGNORE_PROTOCOL

障害内容

［オブジェクトのユーザメソッド］

OTM の一部の機能についてバージョンの不一致がありました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

一部の機能について，場所コードで示されるプロセスのバージョンがほかのプロセスのバージョンと不一致です。再度，確認してください。

3021　DEACTIVATE_FAILURE

障害内容

［TSCRootAcceptor::deactivate］

TSCRootAcceptor::deactivate の処理中にエラーが発生しました。次の要因が考えられます。

- 通信処理中に TSC デーモンで通信障害が発生しました。
- 通信処理中に TSC デーモンでタイムアウトを検知しました。
- TSCObjectFactory::destroy のユーザ実装部分から例外が返されました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。エラーの詳細については KFOT75122-E メッセージの内容を参照してください。完了状態が COMPLETED_MAYBE（0）の場合，異常終了するおそれがあるため，TSCRootAcceptor::destroy，ユーザアクセプタの削除，またはユーザスレッドファクトリの削除は実行しないでください。

3998　TPBROKER_COMM_FAILURE

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::COMM_FAILURE の例外通知を受けました。

付録 D.4　内容コード 4000 ～ 4999

4001　CALL_IN_HOLD

障害内容

［オブジェクトのユーザメソッド］

該当するインタフェース名称，および TSC アクセプタ名称の TSC ユーザオブジェクトが提供するサービスが閉塞中です。または，サーバアプリケーションが異常終了しました。

開発時の対策

［リトライ（閉塞解除ができる場合）］

再度，オブジェクトのユーザメソッドを呼び出してください。

［処理の終了（閉塞解除ができない場合）］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

該当するインタフェース名称，および TSC アクセプタ名称の TSC ユーザオブジェクトが提供するサービスが閉塞中の場合は，それらのサービスを閉塞解除してから，再度，TSC ユーザオブジェクトのユーザメソッドを呼び出してください。

サーバアプリケーションが異常終了している場合は，必要に応じてサーバアプリケーションを再開始してから，再度，TSC ユーザオブジェクトのユーザメソッドを呼び出してください。

4002　RT_ACPT_IS_ACTIVE

障害内容

［TSCRootAcceptor::activate］

TSCRootAcceptor はすでに active 状態です。

開発時の対策

［TSCRootAcceptor の属性変更］

TSCRootAcceptor の属性変更を反映させたい場合，一度，TSCRootAcceptor ::deactivate によって non-active 状態にしてから，再度，TSCRootAcceptor::activate を発行してください。

4005　SERV_CONN_IN_END

障害内容

［TSCAdm::getTSCServer］

TSC デーモンの終了処理中に TSCAdm::getTSCServer を発行しました。

開発時の対策

［ほかの TSC デーモンに接続］

ほかの TSC デーモンに対し，TSCAdm::getTSCServer の処理からやり直してくだ

さい。
［処理の終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策
［TSC デーモンの再開始］
TSC デーモンを再開始したあと，TSCAdm::getTSCServer の処理からやり直してください。

4006　ACTIVATE_IN_END

障害内容
［TSCRootAcceptor::activate］
TSC デーモンの終了処理中に activate を発行しました。

開発時の対策
［ほかの TSC デーモンでの活性化（active 状態への遷移）］
ほかの TSC デーモンに対して，TSCAdm::getTSCServer の処理からやり直してください。
［処理の終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策
［TSC デーモンの再開始］
TSC デーモンを再開始したあと，TSCAdm::getTSCServer の処理からやり直してください。

4007　CLNT_CONN_IN_END

障害内容
［TSCAdm::getTSCClient］
TSC デーモンの終了処理中に TSCAdm::getTSCClient を発行しました。

開発時の対策
［ほかの TSC デーモンでのリトライ］
ほかの TSC デーモンに対し，TSCAdm::getTSCClient の処理からからやり直してください。
［処理の終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策
［TSC デーモンの再開始］
TSC デーモンを再開始したあと，TSCAdm::getTSCClient の処理からやり直してく

ださい。

4008　DIFF_THREAD_CALL

障害内容

［TSCWatchTime::stop］

時間監視を開始したスレッドと異なるスレッドで呼び出されました。

開発時の対策

スレッド制御が不正です。監視を開始したスレッドで呼び出してください。

4009　CALL_IN_END

障害内容

［オブジェクトのユーザメソッド］

TSC デーモンまたはサーバアプリケーションの終了処理中にオブジェクトを発行しました。オブジェクトのユーザメソッドの呼び出しは完了していません。

開発時の対策

［ほかの TSC デーモンでのリトライ］

ほかの TSC デーモンに対し，TSCAdm::getTSCClient の処理からやり直してください。

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［TSC デーモンの再開始］

TSC デーモンを再開始したあと，TSCAdm::getTSCClient の処理からやり直してください。

［サーバアプリケーションの確認］

該当するインタフェースおよび TSC アクセプタ名称のオブジェクトを提供するサーバアプリケーションプロセスが開始していることを確認したあと，再度クライアントアプリケーションを開始してください。

4010　ACPT_NOT_REGISTED

障害内容

［TSCRootAcceptor::activate］

TSC ユーザアクセプタが登録されていない状態で activate を発行しました。

開発時の対策

［TSC ユーザアクセプタの登録］

TSCAcceptor::registerAcceptor によって TSC ユーザアクセプタを登録してください。

4011　SERV_CONN_IN_START

障害内容

［TSCAdm::getTSCServer］

TSC デーモンの初期化処理中に TSCAdm::getTSCServer を発行しました。

開発時の対策

［リトライ］

再度，TSCAdm::getTSCServer を発行してください。

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［TSC デーモンの開始待ち］

TSC デーモンが開始したあと，TSCAdm::getTSCServer の処理からやり直してください。

4012　CLNT_CONN_IN_START

障害内容

［TSCAdm::getTSCClient］

TSC デーモンの初期化処理中に TSCAdm::getTSCClient を発行しました。

開発時の対策

［リトライ］

再度，TSCAdm::getTSCClient を発行してください。

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［TSC デーモンの開始待ち］

TSC デーモンが開始したあと，TSCAdm::getTSCClient の処理からやり直してください。

4013　TSCD_IS_NOT_MY_HOST

障害内容

［TSCAdm::initClient］

［TSCAdm::initServer］

サーバアプリケーションまたはクライアントアプリケーションを管理元の TSC デーモンとは異なるホストで開始しました。

OTM のクライアントアプリケーションを開始するときに自コンピュータ内にない TSC デーモンの TSC ドメインおよび TSC 識別子を指定しました。

運用時の対策

［再開始］
管理元の TSC デーモンと同じホスト上でサーバアプリケーションを開始し直してください。

4014　OVER_ACPT_REGI

障害内容

［TSCRootAcceptor::registerAcceptor］
アクセプタの登録可能数の上限を超えました。

開発時の対策

TSC ユーザアクセプタの登録数を既定値以下にしてください。

4015　NOT_SUPPORTED

障害内容

機能をサポートしていません。
［TSCAdm.initClient］
OTM で，Java アプレットの場合の TSCAdm.initClient を発行しました。
［TSCAdm::getTSCClient］
シングルスレッドライブラリを使用して，way に TSCAdm::TSC_ADM_DIRECT を指定して TSCAdm::getTSCClient を発行しました。
［TSCAdm::initServer］
［TSCAdm::getTSCServer］
［TSCAdm::serverMainloop］
［TSCAdm::releaseTSCServer］
［TSCAdm::endServer］
［TSCAdm::shutdown］
シングルスレッドライブラリを使用しています。または OTM - Client でサーバアプリケーションのメソッドを発行しました。

開発時の対策

［リトライ（TSCAdm::getTSCClient)］
way に TSCAdm::TSC_ADM_REGULATOR を指定して再度，
TSCAdm::getTSCClient からやり直してください。
［接続経路の確認（TSCAdm::getTSCClient)］
接続経路が TSC レギュレータ経由になっているかどうかを確認してください。
［処理の終了（TSCAdm::getTSCClient 以外)］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［CLASSPATH の確認（TSCAdm::initClient)］
CLASSPATH で，次について参照するよう指定されているかどうかを確認してくだ

さい。

- UNIX を使用する場合
  " 格納ディレクトリ /tsccl.jar"，または " 格納ディレクトリ /tscj2cl.jar"
- Windows を使用する場合
  " 格納ディレクトリ ¥tsccl.jar"，または " 格納ディレクトリ ¥tscj2cl.jar"

［リンケージライブラリの確認］
［TSCAdm::initServer］
［TSCAdm::getTSCServer］
［TSCAdm::serverMainloop］
［TSCAdm::releaseTSCServer］
［TSCAdm::endServer］
［TSCAdm::shutdown］
上記のメソッドは OTM - Client またはシングルスレッドライブラリを使用する環境では使用できません。
次に示すライブラリを使用する形でリンケージされているかどうかを確認してください。

- UNIX を使用する場合
  libtscsv.sl または libtsccblsv.sl
- Windows を使用する場合
  tscsv.dll

［上記以外］
機能をサポートしていません。保守コード 1 ～ 4 を取得して，システム管理者に連絡してください。

4016　ACTIVATE_IN_START

障害内容

［TSCRootAcceptor::activate］
TSC デーモンの初期化処理中に TSCRootAcceptor::acitvate を発行しました。

開発時の対策

［リトライ］
再度，TSCRootAcceptor::activate を発行してください。
［処理の終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［TSC デーモンの開始待ち］
TSC デーモンが開始したあと，TSCAdm::getTSCServer の処理からやり直してください。

4018　DEACTIVATE_IN_END

障害内容

［TSCRootAcceptor::deactivate］

TSC デーモンの終了処理中に TSCRootAcceptor::deactivate を発行しました。

開発時の対策

［処理の終了］

TSCRootAcceptor::deactivate が受け付けられる前に，TSC デーモンが終了処理に入りました。OTM の処理を終了してください。

4020　CLNT_DISCONN_IN_END

障害内容

［TSCAdm::releaseTSCClient］

TSC デーモンの終了処理中に TSCAdm::releaseTSCClient を発行しました。

開発時の対策

［処理の終了］

TSCAdm::releaseTSCClient が受け付けられる前に，TSC デーモンが終了処理に入りました。OTM の処理を終了してください。

4022　ACTIVATE_WITH_DIFF_PROP

障害内容

［TSCRootAcceptor::activate］

すでに同じ TSC ルートアクセプタ登録名称で TSCRootAcceptor が登録されていて，該当する TSCRootAcceptor はすでに登録されている TSCRootAcceptor の提供できるサービスの種類（TSC サービス識別子の列）が違います。

開発時の対策

［属性の変更］

TSCRootAcceptor::registerAcceptor，または TSCRootAcceptor::cancelAcceptor によって，TSCRootAcceptor に登録されている TSCAcceptor を調整し，提供できるサービスの種類（TSC サービス識別子の列）を一致させてください。

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

同じ TSC ルートアクセプタ登録名称を利用するときは，TSCRootAcceptor の提供できるサービスの種類（TSC サービス識別子の列）を一致するようにしてください。

4023　CLNT_INIT_IN_END

障害内容

［TSCAdm::initClient］
TSC デーモンの終了処理中に TSCAdm::initClient を発行しました。

開発時の対策
［処理の終了］
TSCAdm::initClient が受け付けられる前に，TSC デーモンが終了処理に入りました。OTM の処理を終了してください。

4024　SERV_INIT_IN_END

障害内容
［TSCAdm::initServer］
TSC デーモンの終了処理中に TSCAdm::initServer を発行しました。

開発時の対策
［処理の終了］
TSCAdm::initServer が受け付けられる前に，TSC デーモンが終了処理に入りました。OTM の処理を終了してください。

4025　CLNT_INIT_IN_START

障害内容
［TSCAdm::initClient］
TSC デーモンの初期化処理中に TSCAdm::initClient を発行しました。

開発時の対策
［リトライ］
再度，TSCAdm::initClient を発行してください。
［処理の終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策
［TSC デーモンの開始待ち］
TSC デーモンが開始したあと，TSCAdm::initClient の処理からやり直してください。

4026　SERV_INIT_IN_START

障害内容
［TSCAdm::initServer］
TSC デーモンの初期化処理中に TSCAdm::initServer を発行しました。

開発時の対策
［リトライ］
再度，TSCAdm::initServer を発行してください。
［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［TSC デーモンの開始待ち］

TSC デーモンが開始したあと，TSCAdm::initServer の処理からやり直してください。

4027　NOT_ACCEPT_OBJECT

障害内容

［TSCRootAcceptor::activate］

TSC ユーザアクセプタから TSCObjectFactory の create を呼び出しましたが，戻り値である TSC ユーザオブジェクトと，TSC ユーザアクセプタのインタフェースが違います。

開発時の対策

［処理の終了］

TSC ユーザオブジェクトファクトリを TSC ユーザアクセプタのコンストラクタの引数に指定する場合，create の戻り値である TSC ユーザオブジェクトと，TSC ユーザアクセプタのインタフェースを一致させてください。

4028　CLNT_COMMAND_START

障害内容

［TSCAdm::initClient］

tscstartprc コマンドから開始されたアプリケーションプログラムから TSCAdm クラスの initClient メソッドを発行しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムの処理を終了してください。

運用時の対策

クライアントアプリケーションを直接，手動で開始してください。

4029　WATCH_IS_STARTED

障害内容

［TSCWatchTime::start］

時間監視中に，監視開始が要求されました。

［TSCWatchTime::reset］

時間監視中に，監視時間のリセットが要求されました。

開発時の対策

時間監視を中断してから，メソッドを発行してください。

4030　WATCH_IS_STOPPED

障害内容

［TSCWatchTime::stop］

時間監視終了または中断中に，監視中断が要求されました。

開発時の対策

時間監視を開始してから，監視中断を要求してください。

4031　SAME_APID_EXIST

障害内容

［TSCAdm::initServer］

指定されたアプリケーション識別子を持つサーバアプリケーションがすでに TSC デーモンに登録されています。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

重複しないアプリケーション識別子を再度指定して，サーバアプリケーションを開始してください。

4032　FILE_ACCESS_FAILURE

障害内容

［TSCAdm::initClient］

［TSCAdm::initServer］

アプリケーションプログラムの開始時にコマンドオプション引数 -TSCRetryReference に指定した接続先情報ファイルがない，または何らかの理由によって開くことができません。

［TSCORB コネクタの開始］

TSCORB コネクタの開始時にコマンドオプション引数 -TSCExceptConvertFile に指定した例外変換テーブルファイルがない，または何らかの理由によって開くことができません。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

［接続先情報ファイルの確認］

コマンドオプション引数 -TSCRetryReference に指定した接続先情報ファイルがあるかどうかと，読み込み権限があるかどうかを確認してください。

Java アプレットの場合，コマンドオプション引数 -TSCRetryReference に，URL でないローカルファイル，またはアプレットダウンロード元以外のファイルの URL を指定するときは，署名によって Java アプレットのセキュリティの制限を解除されているかどうかを確認してください。
［例外変換テーブルファイルの確認］
コマンドオプション引数 -TSCExceptConvertFile に指定した例外変換テーブルファイルがあるかどうかと，読み込み権限があるかどうかを確認してください。

4033　SESSION_IN_END

障害内容

［オブジェクトのユーザメソッド］
［TSCSessionProxy::_TSCStop］
セションが確立されていません。またはセション情報が削除されました。
原因を次に示します。
・セション確立先のサーバアプリケーションが異常終了しました。
・セション呼び出しインターバル監視時間を経過しました。
・セション情報を管理している TSC デーモンが異常終了しました。

開発時の対策

［TSCSessionProxy::_TSCStart からのリトライ（オブジェクトのユーザメソッド）］
TSCSessionProxy クラスの _TSCStop メソッドが発行されたか，またはセション呼び出しインターバル監視時間が過ぎたため，セションが解放された可能性があります。再度，セション呼び出しを行うには，TSCSessionProxy クラスの _Start メソッドの発行からやり直してください。

4034　SESSION_IN_CALL

障害内容

［オブジェクトのユーザメソッド］
［TSCSessionProxy::_TSCStop］
セション呼び出し中です。

開発時の対策

［セション呼び出しの確認］
セション呼び出しが完了したあとで，オブジェクトのユーザメソッド発行および TSCSessionProxy クラスの _TSCStop メソッドの処理からやり直してください。

4998　TPBROKER_NO_PERMISSION

障害内容

［オブジェクトのユーザメソッド］
クライアント側から TSC ユーザプロキシを使用して TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::NO_PERMISSION の例外通知を受けました。

付録 D.5　内容コード 5000 ～ 5999

5001　PROPERTIES_FAILURE

障害内容

コマンドオプション引数の解析中に障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。

5002　MSG_TYPE_FAILURE

障害内容

通信プロトコルの処理中に障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。

5003　MUTEX_FAILURE

障害内容

排他制御の処理で障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

場所コードで示される部分でメモリが不足している可能性が高いです。OTM のシステムを終了してください。その後，プロセス単位の最大ユーザメモリサイズを大きくし，再度，システムを開始してください。

5004　SIG_COND_FAILURE

障害内容

排他制御付きスレッド間通信の処理で障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

場所コードで示される部分でメモリが不足している可能性が高いです。OTM のシステムを終了してください。その後，プロセス単位の最大ユーザメモリサイズを大きくし，再度，システムを開始してください。

5005　EVENT_FAILURE

障害内容

スレッド間通信の処理で障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

場所コードで示される部分でメモリが不足している可能性が高いです。OTM のシステムを終了してください。その後，プロセス単位の最大ユーザメモリサイズを大きくし，再度，システムを開始してください。

5006　SH_MEM_FAILURE

障害内容

共有メモリの初期化処理で障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

TSCSPOOL 環境変数と TSC ドメイン名称の関係が正しいかどうかを確認してください。また，tscstart コマンドの -TSCServerCacheSize オプション引数の指定値が正しいかどうかを確認してください。

5007　THREAD_CREATE_FAILURE

障害内容

スレッド生成の処理中に障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

プロセス単位の最大スレッド数の上限を超えて，スレッドを生成しようとしました。プロセス単位のスレッド数の上限を大きくしてください。

5008　TSD_FAILURE

障害内容

スレッド固有メモリの初期化処理で障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

場所コードで示される部分でメモリが不足している可能性が高いです。OTM のシステムを終了してください。その後，プロセス単位の最大ユーザメモリサイズを大きくし，再度，システムを開始してください。

5009　CBL_ADAPTER_ERROR

障害内容

COBOL アダプタでの処理中に障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

場所コードで示される部分でメモリが不足している可能性が高いです。OTM のシステムを終了してください。その後，プロセス単位の最大ユーザメモリサイズを大きくし，再度，システムを開始してください。

5010　SYSTEM_TIME_FAILURE

障害内容

［TSCWatchTime::start］

［TSCWatchTime::stop］

［TSCWatchTime::reset］

システム時間の取得に失敗しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリ

ケーションプログラムを終了してください。

運用時の対策

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。

5998　TPBROKER_INTERNAL

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::INTERNAL の例外通知を受けました。

5999　PROGRAM_ERROR

障害内容

OTM のシステム中で，その他の障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。さらに，クライアントアプリケーションまたはサーバアプリケーションのプロセスを終了させてください。

運用時の対策

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。

付録 D.6　内容コード 6000 ～ 6999

6001　INVALID_STREAM_LEN

障害内容

［オブジェクトのユーザメソッド（Java）］

CDR マーシャルの処理中に障害が発生しました。ストリーム長に問題があります。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。再度，トランザクションフレームジェネレータを使用して生成し，アプリケーションプログラムをコンパイルおよびリンクしてください。

運用時の対策

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。

6002　INVALID_STREAM_VALUE

障害内容

［オブジェクトのユーザメソッド（Java）］

CDR マーシャルの処理中に障害が発生しました。ストリーム中の値に問題があります。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。再度，トランザクションフレームジェネレータを使用して生成し，アプリケーションプログラムをコンパイルおよびリンクしてください。

運用時の対策

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。

6003　MARSHAL_OTHERS

障害内容

［オブジェクトのユーザメソッド（Java）］

CDR マーシャルの処理中に，ストリーム長またはストリーム中の値以外の障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。再度，トランザクションフレームジェネレータを使用して生成し，アプリケーションプログラムをコンパイルおよびリンクしてください。

運用時の対策

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム

管理者に連絡してください。ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。

6004　REQ_MARSHAL_FAILURE

障害内容

［オブジェクトのユーザメソッド（C++）］

ユーザ要求データの CDR マーシャルの処理中に障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

さらに，クライアントアプリケーションが TSC ユーザオブジェクトのユーザメソッドを呼び出すときの in 引数，または inout 引数の内容に誤りがないかどうかを確認してください。

運用時の対策

クライアントアプリケーションが TSC ユーザオブジェクトのユーザメソッドを呼び出すときの in 引数，または inout 引数の内容に誤りがある可能性があります。引数の内容に誤りがないかどうかを確認してください。誤りがない場合は，内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。

6005　REQ_UNMARSHAL_FAILURE

障害内容

［オブジェクトのユーザメソッド（C++）］

ユーザ要求データの CDR アンマーシャルの処理中に障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

さらに，クライアントアプリケーションが TSC ユーザオブジェクトのユーザメソッドを呼び出すときの in 引数，または inout 引数の内容に誤りがないかどうかを確認してください。また，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンのユーザ定義 IDL インタフェースが一致しているかどうかを確認してください。

運用時の対策

クライアントアプリケーションが TSC ユーザオブジェクトのユーザメソッドを呼び出すときの in 引数，または inout 引数の内容に誤りがある可能性があります。または，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンの

ユーザ定義 IDL インタフェースが一致していない可能性があります。引数の内容に誤りがないかどうかを確認してください。誤りがない場合は，内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。

6006　REP_MARSHAL_FAILURE

障害内容

［オブジェクトのユーザメソッド（C++)］

ユーザ応答データの CDR マーシャルの処理中に障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

さらに，TSC ユーザオブジェクトのユーザメソッドがクライアントアプリケーションに戻す out 引数，inout 引数，または戻り値の内容に誤りがないかどうかを確認してください。

運用時の対策

TSC ユーザオブジェクトのユーザメソッドがクライアントアプリケーションに戻す out 引数，inout 引数，または戻り値の内容に誤りがある可能性があります。引数または戻り値の内容に誤りがないかどうかを確認してください。誤りがない場合は，内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。

6007　REP_UNMARSHAL_FAILURE

障害内容

［オブジェクトのユーザメソッド（C++)］

ユーザ応答データの CDR アンマーシャルの処理中に障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

さらに，TSC ユーザオブジェクトのユーザメソッドがクライアントアプリケーションに戻す out 引数，inout 引数，または戻り値の内容に誤りがないかどうかを確認してください。また，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンのユーザ定義 IDL インタフェースが一致しているかどうかを確認してください。

運用時の対策

TSC ユーザオブジェクトのユーザメソッドがクライアントアプリケーションに戻す out 引数，inout 引数，または戻り値の内容に誤りがある可能性があります。また

は，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンのユーザ定義 IDL インタフェースが一致していない可能性があります。引数または戻り値の内容に誤りがないかどうかを確認してください。誤りがない場合は，内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。

6008　UEXCEPT_MARSHAL_FAILURE

障害内容

［オブジェクトのユーザメソッド（C++)］

ユーザ例外データの CDR マーシャルの処理中に障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

さらに，TSC ユーザオブジェクトのユーザメソッドがクライアントアプリケーションに戻すユーザ例外の内容に誤りがないかどうかを確認してください。

運用時の対策

TSC ユーザオブジェクトのユーザメソッドがクライアントアプリケーションに戻すユーザ例外の内容に誤りがある可能性があります。ユーザ例外の内容に誤りがないかどうかを確認してください。誤りがない場合は，内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。

6009　UEXCEPT_UNMARSHAL_FAILURE

障害内容

［オブジェクトのユーザメソッド（C++)］

ユーザ例外データの CDR アンマーシャルの処理中に障害が発生しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

さらに，TSC ユーザオブジェクトのユーザメソッドがクライアントアプリケーションに戻すユーザ例外の内容に誤りがないかどうかを確認してください。また，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンのユーザ定義 IDL インタフェースが一致しているかどうかを確認してください。

運用時の対策

TSC ユーザオブジェクトのユーザメソッドがクライアントアプリケーションに戻すユーザ例外の内容に誤りがある可能性があります。または，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンのユーザ定義 IDL インタフェースが一致していない可能性があります。ユーザ例外の内容に誤りがないかど

うかを確認してください。誤りがない場合は，内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。

6010　MARSHAL_ERROR

障害内容

［オブジェクトのユーザメソッド（Java)］

CDR マーシャルに障害が発生しました。インタフェースの不一致です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。再度，トランザクションフレームジェネレータを使用して生成し，アプリケーションプログラムをコンパイルおよびリンクしてください。

運用時の対策

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，システム管理者に連絡してください。ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。

6998　TPBROKER_MARSHAL

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::MARSHAL の例外通知を受けました。

付録 D.7　内容コード 7000 ～ 7999

7002　INVALID_DEF_TIMEOUT

障害内容

［TSCAdm::initServer］

［TSCAdm::initClient］

アプリケーションプログラムの開始時に，コマンドオプション引数 -TSCTimeOut に指定したタイムアウト値が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリ

ケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCTimeOut に，規定の監視時間値（0 ～ 2147483647（秒））を指定してください。

7003　INVALID_DEF_RT_ACPT

障害内容

［TSCAdm::initServer］

コマンドオプション引数 -TSCRootAcceptor に指定した TSC ルートアクセプタ登録名称が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCRootAcceptor に，規定の TSC ルートアクセプタ登録名称（1 ～ 31 文字の英数字）を指定してください。

7004　INVALID_DEF_PARALLEL_COUNT

障害内容

［TSCAdm::initServer］

コマンドオプション引数 -TSCParallelCount に指定したパラレルカウント（常駐するスレッド数）が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCParallelCount に，規定のパラレルカウント値（1 ～ 127 の符号なし整数）を指定してください。

7005　LOAD_SHLIB_FAILURE

障害内容

［TSCAdm::initServer（Java）］

［TSCAdm::initClient（Java）］

共有ライブラリの読み込みに失敗しました。

開発時の対策

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

SHLIB_PATH 環境変数が正しく設定されているかどうかを確認してください。

7006　INVALID_DEF_TSCID

障害内容

［TSCAdm::initServer］

［TSCAdm::initClient］

コマンドオプション引数 -TSCID に指定した TSC 識別子が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

コマンドオプション引数 -TSCID に，規定の TSC 識別子（1 ～ 31 文字の英数字）を指定してください。

7007　INVALID_DEF_DOMAIN_NAME

障害内容

コマンドオプション引数 -TSCDomain に指定した TSC ドメイン名称が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

コマンドオプション引数 -TSCDomain に，規定の TSC ドメイン名称（1 ～ 31 文字の英数字）を指定してください。

7008　INVALID_DEF_WITH_SYSTEM

障害内容

［TSCAdm::initClient］

コマンドオプション引数 -TSCWithSystem に指定した値が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

コマンドオプション引数 -TSCWithSystem に，規定の値（0 または 1）を指定してください。

7009　INVALID_ENV_TSCDIR

障害内容

TSCDIR 環境変数または TSCSPOOL 環境変数が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

OTM がインストールされているディレクトリを TSCDIR 環境変数に指定してください。または TSCSPOOL ディレクトリがあるかどうか，アクセス権限があるかどうかを確認してください。

7010　MTRACE_FAILURE

障害内容

モジュールトレース関連のコマンドオプション引数に指定した値が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

モジュールトレース関連のコマンドオプション引数に，規定の値を指定してください。

7011　INVALID_DEF_NICE

障害内容

コマンドオプション引数 -TSCNice に指定した nice 値が不正です。そのため，モジュールトレースの初期化処理に失敗しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCNice に，規定の値（符号なし整数）を指定してください。

7012　INVALID_DEF_PRIORITY

障害内容

アプリケーションプログラムの開始時にコマンドオプション引数
-TSCRequestPriority に指定したプライオリティ値（優先順位）が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCRequestPriority に，規定のプライオリティ値（1 ～ 8 の符号なし整数）を指定してください。

7013　INVALID_DEF_ACPT

障害内容

サーバアプリケーションの開始時にコマンドオプション引数 -TSCAcceptor に指定した TSC アクセプタ名称が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCAcceptor に，規定の TSC アクセプタ名称（1 ～ 31 文字の英数字）を指定してください。

7014　INVALID_PRC_KIND

障害内容

［TSCDomain のコンストラクタ］

TSCAdm::initServer または TSCAdm::initClient を発行する前に TSCDomain を生成しました。

開発時の対策

TSCAdm::initServer または TSCAdm::initClient を発行したあとに TSCDomain を生成し，TSC デーモンと接続してください。

7015　INVALID_DEF_REQUEST_WAY

障害内容

アプリケーションプログラムの開始時にコマンドオプション引数 -TSCRequestWay に指定した接続経路の値が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

コマンドオプション引数 -TSCRequestWay に，規定の値（1 または 0）を指定して

ください。

7016　INVALID_DEF_CLIENT_MESSAGE_BUFFER_COUNT

障害内容

アプリケーションプログラムの開始時にコマンドオプション引数 -TSCClientMessageBufferCount に指定したクライアント通信用バッファ数が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

コマンドオプション引数 -TSCClientMessageBufferCount に，規定のクライアント通信用バッファ数（1 ～ 64 の符号なし整数）を指定してください。

7017　INVALID_DEF_APID

障害内容

［TSCAdm::initServer］

コマンドオプション引数 -TSCAPID に指定したアプリケーション識別子が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムの処理を終了してください。

運用時の対策

コマンドオプション引数 -TSCAPID に，規定のアプリケーション識別子（1 ～ 32 文字の英数字）を指定してください。

7018　INVALID_DEF_WATCH_TIME

障害内容

［TSCAdm::initServer］

サーバアプリケーションの開始時にコマンドオプション引数 -TSCWatchTime に指定した監視時間が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCWatchTime に，規定の監視時間値（0 ～ 2147483647

（秒））を指定してください。

7019　INVALID_DEF_WATCH_METHOD

障害内容

［TSCAdm::initServer］

サーバアプリケーションの開始時にコマンドオプション引数 -TSCWatchMethod に指定した監視時間が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCWatchMethod に，規定の監視時間値（0 ～ 2147483647（秒））を指定してください。

7020　INVALID_DEF_MY_HOST

障害内容

［TSCAdm::initClient］

［TSCAdm::initServer］

コマンドオプション引数 -TSCMyHost に指定したホスト名称または IP アドレスが不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCMyHost に，規定のホスト名称または IP アドレス（1 ～ 64 文字の文字列）を指定してください。

7021　INVALID_DEF_REBIND_TIMES

障害内容

［TSCAdm::initClient］

［TSCAdm::initServer］

アプリケーションプログラムの開始時に，コマンドオプション引数 -TSCRebindTimes に指定したリバインド回数が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCRebindTimes に，規定のリバインド回数（0 ～ 255 の符号なし整数）を指定してください。

7022　INVALID_DEF_REBIND_INTERVAL

障害内容

［TSCAdm::initClient］

［TSCAdm::initServer］

コマンドオプション引数 -TSCRebindInterval に指定したリバインド間隔が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCRebindInterval に，規定のリバインド間隔（0 ～ 65535（秒））を指定してください。

7024　INVALID_DEF_RETRY_REFERENCE

障害内容

［TSCAdm::initClient］

［TSCAdm::initServer］

アプリケーションプログラムの開始時にコマンドオプション引数 -TSCRetryReference に指定した接続先情報ファイルが不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCRetryReference に，規定の接続先情報ファイル名称を指定してください。

7025　INVALID_FILE_FORMAT

障害内容

［TSCAdm::initClient］

［TSCAdm::initServer］

アプリケーションプログラムの開始時にコマンドオプション引数 -TSCRetryReference に指定した接続先情報ファイルの形式が不正です。

［TSCORB コネクタの開始］

TSCORB コネクタの開始時にコマンドオプション引数 -TSCExceptConvertFile に指定した例外変換テーブルファイルの形式が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

［接続先情報ファイルの確認］

コマンドオプション引数 -TSCRetryReference に指定した接続先情報ファイルが，tscgetref コマンドで作成したファイルかどうかを確認してください。また，接続先情報ファイル作成時に，tscgetref コマンドにこのバージョンで使用できない形式を指定していないかどうかを確認してください。

また，接続先情報ファイルの内容を誤って変更していないかどうか，接続先情報ファイル名称にディレクトリを指定していないかどうかを確認してください。

［例外変換テーブルファイルの確認］

コマンドオプション引数 -TSCExceptConvertFile に指定した例外変換テーブルファイルの内容が誤っていないかどうかと，例外変換テーブルファイル名称にディレクトリを指定していないかどうかを確認してください。

7030　INVALID_DEF_RETRY_WAY

障害内容

［TSCAdm::initClient］

［TSCAdm::initServer］

アプリケーションプログラムの開始時にコマンドオプション引数 -TSCRetryWay に指定した値が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCRetryWay に，規定の値（4 けたの符号なし整数）を指定してください。

7031　ALREADY_SHUTDOWN

障害内容

［TSCAdm:: getTSCClient］

クライアントアプリケーションと，TSC デーモンまたは TSC レギュレータとの接続の過程で，CORBA::INITIALIZE の例外通知を受けました。

［オブジェクトのユーザメソッド］

オブジェクトのユーザメソッドを呼び出す過程で，CORBA::INITIALIZE の例外通

知を受けました。
［TSCAdm:: releaseTSCClient］
クライアントアプリケーションと，TSC デーモンまたは TSC レギュレータとの切断の過程で，CORBA::INITIALIZE の例外通知を受けました。

開発時の対策
［処理の終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策
CORBA::ORB_init() または ORB::shutdown() が発行されていないかを確認してください。

7034　INVALID_DEF_EXCEPT_CONVERT_FILE

障害内容
コマンドオプション引数 -TSCExceptConvertFile に指定した例外変換テーブルファイル名称が不正です。

開発時の対策
［処理の終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策
コマンドオプション引数 -TSCExceptConvertFile に，規定の例外変換テーブルファイル名称を指定してください。

7035　ENTRY_FAILURE

障害内容
[TSCAdm::initServer]
サーバアプリケーションのプロセス登録処理で障害が発生しました。

開発時の対策
[処理の終了]
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策
TSCAdm::initServer の発行までの時間が長過ぎる可能性があります。tscstartprc コマンドの -TSCStartTimeOut オプションの指定値を増やして，再度サーバアプリケーションを起動してください。

7036　INVALID_DEF_SESSION_INTERVAL

障害内容

コマンドオプション引数 -TSCSessionInterval に指定したセション呼び出しインターバル監視時間が不正です。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

コマンドオプション引数 -TSCSessionInterval に，規定のセション呼び出しインターバル監視時間（1 ～ 2147483647 の符号なし整数）を指定してください。

7040　INVALID_DEF_QUEUE_LENGTH

障害内容

tscstartprc コマンドまたはサーバアプリケーションの開始コマンドの -TSCQueueLength オプションで指定したスケジュール用キューの長さが不正です。

開発時の対策

tscstartprc コマンドまたはサーバアプリケーションの開始コマンドの -TSCQueueLength オプションで指定したスケジュール用キューの長さを見直し，規定のスケジュール用キューの長さを指定してください。

7998　TPBROKER_INITIALIZE

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::INITIALIZE の例外通知を受けました。

付録 D.8　内容コード 8000 ～ 8999

8001　NO_SUCH_INTERF

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトを呼び出しましたが，TSC ユーザプロキシと同じインタフェース名称の TSC ユーザオブジェクトが活性化されていません（active 状態ではありません）。そのため，該当する TSC ユーザオブジェクト呼び出しは成功しませんでした。

開発時の対策

［リトライ］

該当するインタフェース名称の TSC ユーザオブジェクトを active 状態に遷移させたあと，再度，TSC ユーザオブジェクトを呼び出してください。

［処理の終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，TSC ユーザオブジェクト呼び出しの処理を終了してください。

運用時の対策
［TSC ユーザオブジェクトの活性化（active 状態への遷移）］
該当するインタフェース名称の TSC ユーザオブジェクトを active 状態に遷移させたあと，再度，オブジェクトのユーザメソッドを呼び出す処理をしてください。
［環境設定の確認］
TSC ドメインマネジャの環境設定に誤りがないかどうかを確認し，対策したあと，再度，TSC ユーザオブジェクトを呼び出してください。次に示す環境設定に誤りがある可能性があります。
- OSAGENT_PORT 環境変数の指定値
- コマンドオプション引数 -TSCDomain の指定値
- コマンドオプション引数 -TSCPort の指定値
- コマンドオプション引数 -TSCSendInterval の指定値

8002　NO_SUCH_ACPT

障害内容
［オブジェクトのユーザメソッド］
クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトを呼び出しましたが，TSC ユーザプロキシと同じインタフェース名称で，かつ，同じ TSC アクセプタ名称の TSC ユーザオブジェクトが活性化されていません（active 状態ではありません）。そのため，該当する TSC ユーザオブジェクト呼び出しは成功しませんでした。

開発時の対策
［リトライ］
該当するインタフェース名称，TSC アクセプタ名称の TSC ユーザオブジェクトを active 状態に遷移させたあと，再度，TSC ユーザオブジェクトを呼び出してください。
［処理の終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，TSC ユーザオブジェクト呼び出しの処理を終了してください。

運用時の対策
該当するインタフェース名称，および TSC アクセプタ名称の TSC ユーザオブジェクトを active 状態に遷移させたあと，再度，オブジェクトのユーザメソッドを呼び出す処理を実行してください。

8998　TPBROKER_NO_IMPLEMENT

障害内容
［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::NO_IMPLEMENT の例外通知を受けました。

付録 D.9　内容コード 9000 ～ 9999

9001　NO_SUCH_OP_NAME

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトを呼び出しましたが，クライアント側から呼び出したオペレーションがサーバ側の TSC ユーザオブジェクトでサポートされていません。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，TSC ユーザオブジェクト呼び出しの処理を終了してください。

ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。再度，トランザクションフレームジェネレータを使用して生成し，アプリケーションプログラムをコンパイルおよびリンクしてください。

クライアント側から呼び出しオペレーションがサーバ側でサポートされていません。

運用時の対策

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得して，システム管理者に連絡してください。ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。再度，トランザクションフレームジェネレータを使用して生成し，アプリケーションプログラムをコンパイルおよびリンクしてください。

9998　TPBROKER_BAD_OPERATION

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::BAD_OPERATION の例外通知を受けました。

付録 D.10　内容コード 10000 ～ 10999

10001　OVER_MAX_CLNT

障害内容

［TSCAdm::getTSCClient］
TSC デーモンに直結するクライアントアプリケーションおよび TSC レギュレータの数が，TSC デーモンのコマンドオプション引数 -TSCClientConnectCount で指定した値を超過しました。

開発時の対策
［処理を終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。
［リトライ（ほかの TSC デーモンが開始済みの場合）］
ほかの TSC デーモンに対し，TSCAdm::getTSCClient の処理からやり直してください。

運用時の対策
［定義の変更］
TSC デーモンを終了後，コマンドオプション引数 -TSCClientConnectCount の値を増加させて，再度，開始してください。
［ほかの TSC デーモンの開始］
ほかの TSC デーモンに対し，TSCAdm::getTSCClient からやり直してください。

10002　OVER_MAX_SERV

障害内容
［TSCAdm::getTSCServer］
TSC デーモンに接続するサーバアプリケーションの数が，TSC デーモンのコマンドオプション引数 -TSCServerConnectCount で指定した値を超過しました。

開発時の対策
［処理を終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。
［リトライ（ほかの TSC デーモンが開始済みの場合）］
ほかの TSC デーモンに対し，TSCAdm::getTSCServer の処理からやり直してください。

運用時の対策
［定義の変更］
TSC デーモンを終了後，コマンドオプション引数 -TSCServerConnectCount の値を増加させて，再度，開始してください。
［ほかの TSC デーモンの開始］
ほかの TSC デーモンを開始し，TSCAdm::getTSCServer からやり直してください。

10005　OVER_ADM_MAX_CLNT

障害内容

［TSCAdm::initClient］
TSC デーモンに管理されるプロセス数が，TSC デーモンのコマンドオプション引数 -TSCEntryCount で指定した値を超過しました。

開発時の対策
［処理を終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策
［定義の変更］
TSC デーモンを終了後，コマンドオプション引数 -TSCEntryCount（省略している場合は -TSCClientConnectCount）の値を増加させて，再度，開始してください。
［ほかの TSC デーモンの開始］
ほかの TSC デーモンに管理されるように，コマンドオプション引数 -TSCDomain および -TSCID の値を変更してください。

10006　OVER_ADM_MAX_SERV

障害内容
［TSCAdm::initServer］
TSC デーモンに管理されるプロセス数が，TSC デーモンのコマンドオプション引数 -TSCEntryCount で指定した値を超過しました。

開発時の対策
［処理を終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策
［定義の変更］
TSC デーモンを終了後，コマンドオプション引数 -TSCEntryCount（省略している場合は -TSCServerConnectCount）の値を増加させて，再度，開始してください。
［ほかの TSC デーモンの開始］
ほかの TSC デーモンに管理されるように，コマンドオプション引数 -TSCDomain および -TSCID の値を変更してください。

10007　OVER_MAX_RT_ACPT_REGI

障害内容
［TSCRootAcceptor::activate］
TSC デーモンに登録する TSC ルートアクセプタ数が，TSC デーモンのコマンドオプション引数 -TSCRootAcceptorCount または -TSCRootAcceptorRegistCount に指定した値を超過しました。

開発時の対策

［処理を終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。
［リトライ（ほかの TSC デーモンが開始済みの場合）］
ほかの TSC デーモンに対し，TSCAdm::getTSCServer の処理からやり直してください。

運用時の対策
［定義の変更］
TSC デーモンを終了後，コマンドオプション引数 -TSCRootAcceptorCount，または -TSCRootAcceptorRegistCount の値を増加させて，再度，開始してください。
［ほかの TSC デーモンの開始］
ほかの TSC デーモンに対し，TSCAdm::getTSCServer からやり直してください。

10008　OVER_MAX_THIN_CLIENT

障害内容
［TSCAdm::getTSCClient］
TSC レギュレータ経由で TSC デーモンに接続する，クライアントアプリケーションの数が TSC レギュレータのコマンドオプション引数 -TSCClientConnectCount で指定した値を超過しました。または，クライアントアプリケーションの開始時に TSC レギュレータが動作していません。

開発時の対策
［処理を終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。
［リトライ（ほかの TSC デーモンが開始済みの場合）］
ほかの TSC デーモンに対し，TSCAdm::getTSCClient からやり直してください。

運用時の対策
［定義の変更］
TSC レギュレータを終了後，コマンドオプション引数 -TSCClientConnectCount の値を増加させて，再度，開始してください。または，クライアントアプリケーションの開始時までに TSC レギュレータを開始してください。
［ほかの TSC デーモンの開始］
ほかの TSC デーモンに対し，TSCAdm::getTSCClient の処理からやり直してください。

10009　OVER_MAX_DISPATCH_PARALLEL

障害内容
［TSCRootAcceptor::activate］
TSC デーモンに登録する TSC ルートアクセプタのパラレルカウント（常駐するス

レッド数）の和が，TSC デーモンのコマンドオプション引数
-TSCDispatchParallelCount で指定した値を超過しました。

開発時の対策

［処理を終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。
［リトライ（ほかの TSC デーモンが開始済みの場合）］
ほかの TSC デーモンに対し，TSCAdm:getTSCServer の処理からやり直してください。

運用時の対策

［定義の変更］
TSC デーモンを終了後，コマンドオプション引数 -TSCDispatchParallelCount の値を増加させて，再度，開始してください。
［ほかの TSC デーモンの開始］
ほかの TSC デーモンに対し，TSCAdm::getTSCServer からやり直してください。

10010　OVER_MAX_REQUEST_COUNT

障害内容

［オブジェクトのユーザメソッド］
クライアント側から呼び出したインタフェース名称と同じ TSC ユーザオブジェクトを持つ TSC ルートアクセプタの，スケジュール用キューの最大値を超えています。

開発時の対策

［リトライ］
再度，オブジェクトを呼び出してください。スケジュール用キューに空きができている可能性があります。
長時間連続して発生する場合は，該当するインタフェース名称の TSC ユーザオブジェクトを持つ TSC ルートアクセプタの常駐プロセス数，または TSC ルートアクセプタのプロセス数を増加してください。
［処理を終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。

運用時の対策

［定義の変更］
該当するインタフェース名称と同じ TSC ユーザオブジェクトを持つ TSC ルートアクセプタを増加するか，TSC ルートアクセプタのパラレルカウント数を増加させて，再度，TSC ルートアクセプタを開始してください。

10011　OVER_MAX_ORB_CLIENT

障害内容

［ORB クライアントからのリクエスト時］

TSCORB コネクタに接続する ORB クライアントアプリケーションの数が，TSCORB コネクタを開始時に tscstartgw コマンドに指定したコマンドオプション引数 -TSCClientConnectCount の値を超過しました。

開発時の対策

［ほかの TSCORB コネクタが開始済み］

ほかの TSCORB コネクタに対し，接続処理からやり直してください。

運用時の対策

［定義の変更］

TSCORB コネクタを終了後，コマンドオプション引数 -TSCClientConnectCount の値を増加させて，再度，TSCORB コネクタを開始してください。

［ほかの TSCORB コネクタの開始］

ほかの TSCORB コネクタに対し，接続処理からやり直してください。

10998　TPBROKER_NO_RESOURCES

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::NO_RESOURCES の例外通知を受けました。

付録 D.11　内容コード 11000 ～ 11999

11001　TIMED_OUT

障害内容

［オブジェクトのユーザメソッド］

オブジェクトのユーザメソッド呼び出しの応答が，監視時間を超えても戻りません。オブジェクトのユーザメソッドの呼び出しは完了している可能性があります。

11998　TPBROKER_NO_RESPONSE

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::NO_RESPONSE の例外通知を受けました。

付録 D.12　内容コード 12000 ～ 12999

12002　ALREADY_ACTIVE

障害内容

［TSCRootAcceptor::activate］

すでに TSCRootAcceptor の activate を発行していて，TSC ルートアクセプタは active 状態です。

開発時の対策

［TSCRootAcceptor の属性を変更する場合］

一度，TSCRootAcceptor の deactivate を発行して，non-active 状態に遷移させてください。その後，TSCRootAcceptor の属性を変更し，再度，activate を発行してください。TSCRootAcceptor は active 状態です。

12003　ALREADY_DEACTIVE

障害内容

［TSCRootAcceptor::deactivate］

TSCRootAcceptor は non-active 状態です。

開発時の対策

［処理の続行］

無視して，処理を続けてください。TSCRootAcceptor は non-active 状態です。

12004　CLNT_NOT_INITIALIZED

障害内容

［TSCAdm::getTSCClient］

［TSCAdm::endClient］

TSCAdm::initClient または TSCAdm::initServer が発行されていません。

開発時の対策

ユーザメソッドは，TSCAdm::initClient の発行から TSCAdm::endClient の発行までの間，または TSCAdm::initServer の発行から TSCAdm::endServer の発行までの間で使用してください。

12005　SERV_NOT_INITIALIZED

障害内容

［TSCAdm::getTSCServer］

［TSCAdm::serverMainloop］

［TSCAdm::releaseTSCServer］

［TSCAdm::endServer］

［TSCAdm::shutdown］

TSCAdm::initServer が発行されていません。

開発時の対策

［リトライ］

TSCAdm の initServer の処理からやり直してください。

12006　ALREADY_INITCLNT

障害内容

［TSCAdm::initServer］

TSCAdm::initClient の発行後は，TSCAdm::initServer を呼び出すことはできません。

［TSCAdm::initClient］

すでに TSCAdm::initClient を発行済みで，クライアントアプリケーションの初期化処理が完了しています。

開発時の対策

［処理の終了］

TSCAdm::initClient または TSCAdm::initServer の呼び出しは有効になりません。クライアントアプリケーションの処理を終了させてください。

12007　ALREADY_INITSERV

障害内容

［TSCAdm::initServer］

すでに TSCAdm::initServer を発行済みで，サーバアプリケーションの初期化処理が完了しています。

［TSCAdm::initClient］

TSCAdm::initServer の発行後は，TSCAdm::initClient を呼び出すことはできません。

開発時の対策

［処理の終了］

TSCAdm::initServer または TSCAdm::initClient の呼び出しは有効になりません。サーバアプリケーションの処理を終了させてください。TSCAdm::initServer は，プロセスで 1 回しか呼び出せません。

12008　ALREADY_SERV_ML

障害内容

［TSCAdm::serverMainloop］

すでに別のスレッドで TSCAdm::serverMainloop が発行済みです。

開発時の対策

［処理の続行］

TSCAdm::serverMainloop の呼び出しは有効になりません。

12013　ALREADY_SESSION_START

障害内容

［TSCSessionProxy::_TSCStart］

すでに TSCSessionProxy::_TSCStart が発行されています。

開発時の対策

［処理の続行］

無視して，処理を続けてください。セション呼び出し初期化処理は完了しています。

12014　SESSION_NOT_START

障害内容

［オブジェクトのユーザメソッド］

［TSCSessionProxy::_TSCStop］

TSCSessionProxy::_TSCStart が発行されていません。

開発時の対策

［リトライ］

TSCSessionProxy クラスの _TSCStart の処理からやり直してください。

12998　TPBROKER_BAD_INV_ORDER

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::BAD_INV_ORDER の例外通知を受けました。

付録 D.13　内容コード 13000 ～ 13999

13001　REBIND_FAILURE

障害内容

［オブジェクトのユーザメソッド］

［TSCSessionProxy::_TSCStart］

［TSCSessionProxy::_TSCStop］

アプリケーションプログラムの開始時に，コマンドオプション引数 -TSCRebindTimes で指定した回数分連続して，リバインド処理に失敗しました。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

［再接続］

再度，TSC ユーザオブジェクトのユーザメソッドを呼び出してください。

13002　ALL_CONN_FAILURE

障害内容

［TSCAdm::getTSCClient］

接続先情報ファイルに指定されたすべての接続に失敗しました。

開発時の対策

［リトライ（TSCAdm::getTSCClient）］

再度，TSCAdm::getTSCClient からやり直してください。

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，アプリケーションプログラムを終了してください。

運用時の対策

［最大コネクション数（ファイルディスクリプタ数）の確認］

プロセス単位で確立できる最大コネクション数，または取得できるファイルディスクリプタ数を超えて，コネクションを確立しようとした可能性が高いです。OTM のシステムを終了してください。その後，プロセス単位で確立できる最大コネクション数，または取得できるファイルディスクリプタ数の上限を変更し，再度，システムを開始してください。

［指定ポート番号の確認］

コマンドオプション引数 -TSCPort に指定したポート番号は，すでにほかのプロセスで使用されている可能性があります。コマンドオプション引数 -TSCPort に指定するポート番号をほかのプロセスが使用していないポート番号に変更し，再度，実行してください。

［接続先情報ファイルの確認］

接続先情報ファイルの内容が適切でない可能性があります。接続先情報ファイルの内容を修正し，再度実行してください。

13998　TPBROKER_TRANSIENT

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::TRANSIENT の例外通知を受けました。

付録 D.14　内容コード 14000 ～ 14999

14001　SERV_NO_SUCH_INTERF

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトを呼び出しましたが，TSC ユーザプロキシと同じインタフェース名称の TSC ユーザオブ

ジェクトがサーバアプリケーション上にありません。そのため，TSC ユーザオブジェクト呼び出しは成功しませんでした。

開発時の対策

［リトライ］

該当するインタフェース名称の TSC ユーザオブジェクトを活性化（active 状態に遷移）させたあと，再度，TSC ユーザオブジェクトを呼び出してください。

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，TSC ユーザオブジェクト呼び出しの処理を終了してください。

運用時の対策

該当するインタフェース名称の TSC ユーザオブジェクトを活性化（active 状態に遷移）させたあと，再度，オブジェクトのユーザメソッドを呼び出す処理を実行してください。

14002　SERV_NO_SUCH_ACPT

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトを呼び出しましたが，TSC ユーザプロキシと同じインタフェース名称で，かつ，同じ TSC アクセプタ名称の TSC ユーザオブジェクトがサーバアプリケーション上にありません。そのため，TSC ユーザオブジェクト呼び出しは成功しませんでした。

開発時の対策

［リトライ］

該当するインタフェース名称，TSC アクセプタ名称の TSC ユーザオブジェクトを活性化（active 状態に遷移）させたあと，再度，TSC ユーザオブジェクトを呼び出してください。

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，TSC ユーザオブジェクト呼び出しの処理を終了してください。

運用時の対策

該当するインタフェース名称，および TSC アクセプタ名称の TSC ユーザオブジェクトを活性化（active 状態に遷移）させたあと，再度，オブジェクトのユーザメソッドを呼び出す処理を実行してください。

14998　TPBROKER_OBJECT_NOT_EXIST

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::OBJECT_NOT_EXIST の例外通知を受けました。

付録 D.15　内容コード 15000 ～ 15999

15001　COMMON_EXCEPTION

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，TSCSystemException 以外の例外通知を受けました。

［ORB クライアントからのリクエスト］

ORB クライアントからのリクエストの処理中に，TSCORB コネクタ内で TSCSystemException および CORBA::SystemException 以外の例外が発生しました。

15002　INVALID_USER_EXCEPTION

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトを呼び出しユーザ例外通知を受けましたが，解釈できません。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，TSC ユーザオブジェクト呼び出しの処理を終了してください。

ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。再度，トランザクションフレームジェネレータを使用して生成し，アプリケーションプログラムをコンパイルおよびリンクしてください。または，サーバ側から通知されたユーザ例外がクライアント側でサポートされていません。

運用時の対策

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得して，システム管理者に連絡してください。ユーザ定義 IDL インタフェースから生成される，クライアント側の TSC ユーザプロキシとサーバ側の TSC ユーザスケルトンが異なる可能性があります。再度，トランザクションフレームジェネレータを使用して生成し，アプリケーションプログラムをコンパイルおよびリンクしてください。

15998　TPBROKER_UNKNOWN

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::UNKNOWN の例外通知を受けまし

た。

付録 D.16　内容コード 16000 ～ 16999

16001　FACTORY_CREATE_FAILURE

障害内容

［TSCRootAcceptor::activate］

TSCObjectFactory::create から TSCSystemException 以外の通知を受けました。

TSCRootAcceptor::activate は失敗しています。

開発時の対策

［リトライ］

TSCObjectFactory::create で例外を通知しない状態で，再度，

TSCRootAcceptor::activate を発行してください。

16002　THREAD_FACTORY_CREATE_FAILURE

障害内容

［TSCRootAcceptor::activate］

TSCThreadFactory::create から TSCSystemException 以外の通知を受けました。

TSCRootAcceptor::activate は失敗しています。

開発時の対策

［リトライ］

TSCThreadFactory::create で例外を通知しない状態で，再度，

TSCRootAcceptor::activate を発行してください。

16003　FACTORY_DESTROY_FAILURE

障害内容

［TSCRootAcceptor::deactivate］

TSCObjectFactory::destroy から TSCSystemException 以外の通知を受けました。

TSCRootAcceptor::deactivate は失敗しています。

開発時の対策

［処理の終了］

内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。完了状態が COMPLETED_MAYBE（0）の場合，

TSCRootAcceptor::deactivate の処理が完了していないため，アプリケーションプロセスが異常終了する可能性があります。

16004　THREAD_FACTORY_DESTROY_FAILURE

障害内容

［TSCRootAcceptor::deactivate］
TSCThreadFactory::destroy から TSCSystemException 以外の通知を受けました。TSCRootAcceptor::deactivate は失敗しています。

開発時の対策
［処理の終了］
内容コード，場所コード，完了状態，および保守コード 1 ～ 4 を取得し，OTM の処理を終了してください。完了状態が COMPLETED_MAYBE（0）の場合，TSCRootAcceptor::deactivate の処理が完了していないため，アプリケーションプロセスが異常終了する可能性があります。

16005　CREATED_OBJECT_IS_NULL

障害内容
［TSCRootAcceptor::activate］
TSCObjectFactory::create が null で返りました。TSCRootAcceptor::activate は失敗しています。

開発時の対策
［リトライ］
TSCObjectFactory::create で例外を通知しない状態で，再度，TSCRootAcceptor::activate を発行してください。

16006　CREATED_THREAD_OBJECT_IS_NULL

障害内容
［TSCRootAcceptor::activate］
TSCThreadFactory::create が null で返りました。TSCRootAcceptor::activate は失敗しています。

開発時の対策
［リトライ］
TSCThreadFactory::create で例外を通知しない状態で，再度，TSCRootAcceptor::activate を発行してください。
［確認］
TSCObjectFactory の状態を確認してください。

16998　TPBROKER_INV_OBJREF

障害内容
［オブジェクトのユーザメソッド］
クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::INV_OBJREF の例外通知を受けました。

付録 D.17　内容コード 17000 以降

17998　TPBROKER_IMP_LIMIT

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::IMPLIMIT の例外通知を受けました。

18998　TPBROKER_BAD_TYPECODE

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::BAD_TYPECODE の例外通知を受けました。

19998　TPBROKER_PERSIST_STORE

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::PERSIST_STORE の例外通知を受けました。

20998　TPBROKER_FREE_MEM

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::FREE_MEM の例外通知を受けました。

21998　TPBROKER_INV_IDENT

障害内容

［オブジェクトのユーザメソッド］

クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::INV_IDENT の例外通知を受けました。

22998　TPBROKER_INV_FLAG

障害内容

［オブジェクトのユーザメソッド］
クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::INV_FLAG の例外通知を受けました。

23998　TPBROKER_INTF_REPOS

障害内容
［オブジェクトのユーザメソッド］
クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::INTF_REPOS の例外通知を受けました。

24998　TPBROKER_BAD_CONTEXT

障害内容
［オブジェクトのユーザメソッド］
クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::BAD_CONTEXT の例外通知を受けました。

25998　TPBROKER_OBJ_ADAPTER

障害内容
［オブジェクトのユーザメソッド］
クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::OBJ_ADAPTER の例外通知を受けました。

26998　TPBROKER_DATA_CONVERSION

障害内容
［オブジェクトのユーザメソッド］
クライアント側から TSC ユーザプロキシを使用して，TSC ユーザオブジェクトのユーザメソッドを呼び出しましたが，CORBA::DATA_CONVERSION の例外通知を受けました。

索引

記号

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

W

あ

い

え

か

き

く

け

こ

さ

し

せ

と

な

は

ひ

ふ

ゆ

れ

ソフトウェアマニュアルのサービス ご案内

ソフトウェアマニュアルについて，３種類のサービスをご案内します。ご活用ください。

１．マニュアル情報ホームページ

ソフトウェアマニュアルの情報をインターネットで公開しております。

URL　http://www.hitachi.co.jp/soft/manual/

ホームページのメニューは次のとおりです。

■Web提供マニュアル一覧	インターネットで参照できるマニュアルの一覧を提供しています。（詳細は「２．インターネットからのマニュアル参照」を参照してください。）
■CD-ROMマニュアル情報	複数マニュアルを格納したCD-ROMマニュアルを提供しています。どの製品に対応したCD-ROMマニュアルがあるか，を参照できます。
■マニュアルに関するご意見・ご要望	マニュアルに関するご意見，ご要望をお寄せください。

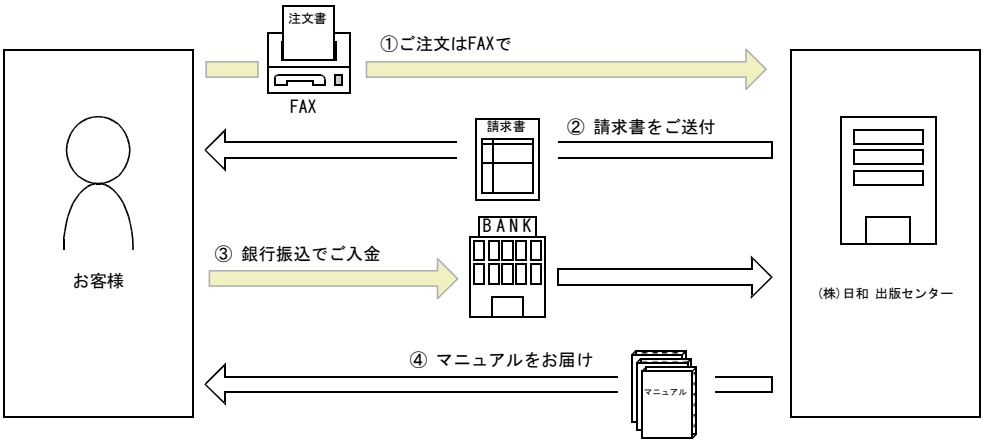
２．インターネットからのマニュアル参照（ソフトウェアサポートサービス）

ソフトウェアサポートサービスの契約をしていただくと，インターネットでマニュアルを参照できます。（本サービスの対象となる契約の種別，及び参照できるマニュアルは，マニュアル情報ホームページでご確認ください。参照できるマニュアルは，クライアント／サーバ系の日立オープンミドルウェア製品を中心に順次対象を拡大予定です。）

なお，ソフトウェアサポートサービスは，マニュアル参照だけでなく，対象製品に対するご質問への回答，問題解決支援，バージョン更新版の提供など，お客様のシステムの安定的な稼働のためのサービスをご提供しています。まだご契約いただいていない場合は，ぜひご契約いただくことをお勧めします。

３．マニュアルのご注文

裏面の注文書でご注文ください。



① マニュアル注文書に必要事項をご記入のうえ，FAX でご注文ください。
② ご注文いただいたマニュアルについて，請求書をお送りします。
③ 請求書の金額を指定銀行へ振り込んでください。なお，送料は弊社で負担します。
④ 入金確認後，7 日以内にお届けします。在庫切れの場合は，納期を別途ご案内いたします。

（株）日和 出版センター　行き
FAX 番号　０１２０－２１０－４５４（フリーダイヤル）

日立マニュアル注文書

ご注文日	年　　月　　日
送付先ご住所	〒
お客様名 （団体名，又は法人名など）	
お名前	
電話番号	（　　）
FAX 番号	（　　）

資料番号	マニュアル名	数量
合計		

マニュアルのご注文について，ご不明な点は
（株）日和 出版センター（☎０３－５２８１－５０５４）へお問い合わせください。