
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

3000-3-909

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-E151 TPBroker Object Transaction Monitor 05-00

P-1M64-E251 TPBroker Object Transaction Monitor - Client 05-00

P-1M64-E551 TPBroker Object Transaction Monitor - Connector for Object Request Broker 02-00

・適用 OS : HP-UX 11i , 11.0

P-1B64-E151 TPBroker Object Transaction Monitor 05-00

P-1B64-E251 TPBroker Object Transaction Monitor - Client 05-00

P-1B64-E551 TPBroker Object Transaction Monitor - Connector for Object Request Broker 02-00

・適用 OS : HP-UX 11i V2.0 (IPF)

P-1J64-E151 TPBroker Object Transaction Monitor 05-00

P-1J64-E251 TPBroker Object Transaction Monitor - Client 05-00

P-1J64-E551 TPBroker Object Transaction Monitor - Connector for Object Request Broker 02-00

・適用 OS : Red Hat Enterprise Linux AS 3.0

P-9S64-E151 TPBroker Object Transaction Monitor 05-00

P-9S64-E251 TPBroker Object Transaction Monitor - Client 05-00

P-9S64-E551 TPBroker Object Transaction Monitor - Connector for Object Request Broker 02-00

・適用 OS : Red Hat Enterprise Linux AS 3.0 (IPF)

P-9V64-E151 TPBroker Object Transaction Monitor 05-00

P-9V64-E251 TPBroker Object Transaction Monitor - Client 05-00

P-9V64-E551 TPBroker Object Transaction Monitor - Connector for Object Request Broker 02-00

・適用 OS : Solaris

P-9D64-E151 TPBroker Object Transaction Monitor 05-00

P-9D64-E251 TPBroker Object Transaction Monitor - Client 05-00

P-9D64-E551 TPBroker Object Transaction Monitor - Connector for Object Request Broker 02-00

・適用 OS : Windows Server 2003 , Windows XP , Windows 2000

P-2464-E154 TPBroker Object Transaction Monitor 05-00

P-2464-E254 TPBroker Object Transaction Monitor - Client 05-00

P-2464-E554 TPBroker Object Transaction Monitor - Connector for Object Request Broker 02-00

・適用 OS : Windows Server 2003 (IPF)

P-2864-E154 TPBroker Object Transaction Monitor 05-00

P-2864-E254 TPBroker Object Transaction Monitor - Client 05-00

P-2864-E554 TPBroker Object Transaction Monitor - Connector for Object Request Broker 02-00

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

AIX は、米国における米国 International Business Machines Corp. の登録商標です。
Borland のブランド名および製品名はすべて、米国 Borland Software Corporation の米国およびその他の国における商標または登録商標です。
CORBA は、Object Management Group が提唱する分散処理環境アーキテクチャの名称です。
HP-UX は、米国 Hewlett-Packard Company のオペレーティングシステムの名称です。
Itanium は、アメリカ合衆国および他の国におけるインテル コーポレーションまたはその子会社の登録商標です。
Java 及びすべての Java 関連の商標及びロゴは、米国及びその他の国における米国 Sun Microsystems, Inc. の商標または登録商標です。
Linux は、Linus Torvalds の米国およびその他の国における登録商標あるいは商標です。
Microsoft は、米国およびその他の国における米国 Microsoft Corp. の登録商標です。
Red Hat は、米国およびその他の国で Red Hat, Inc. の登録商標若しくは商標です。
Solaris は、米国 Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。
UNIX は、X/Open Company Limited が独占的にライセンスしている米国ならびに他の国における登録商標です。
Windows は、米国およびその他の国における米国 Microsoft Corp. の登録商標です。
Windows Server は、米国およびその他の国における米国 Microsoft Corp. の商標です。

プログラムプロダクト「P-9D64-E151, P-9D64-E251, P-9D64-E551」には、米国 Sun Microsystems, Inc. が著作権を有している部分が含まれています。

プログラムプロダクト「P-9D64-E151, P-9D64-E251, P-9D64-E551」には、UNIX System Laboratories, Inc. が著作権を有している部分が含まれています。

発行

2004年9月(第1版) 3000-3-909

著作権

All Rights Reserved. Copyright (C) 2004, Hitachi, Ltd.

はじめに

このマニュアルは、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 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。

OTM - Connector for ORB は、TPBroker OTM 05-00 ではサポートされていません。サポート時期をご確認の上、ご使用ください。

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 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。

はじめに

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。

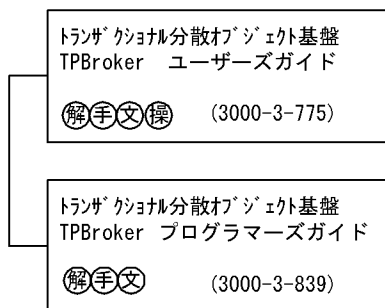
付録 D 内容コード一覧

例外で使用する内容コードについて説明しています。

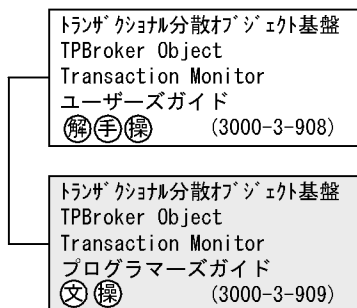
関連マニュアル

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。

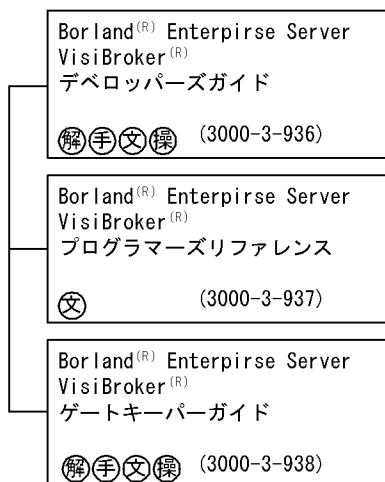
・ TPBroker Version 5



・ TPBroker OTM Version 5



・ VisiBroker Version 5

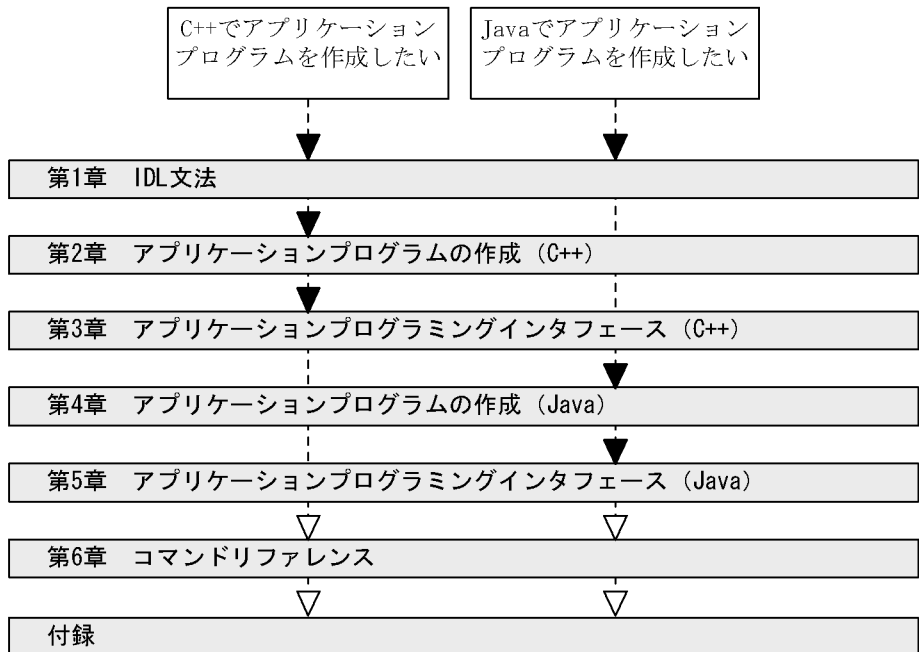


<記号>

- 解 : 解説書
- 手 : 手引書
- 文 : 文法書
- 操 : 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。



(凡例)  : 必ず読む項目  : 必要に応じて読む項目

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。

製品名称	略称	
AIX 5L	AIX	
HP-UX 11.0	HP-UX	
HP-UX 11i		
HP-UX 11i V2.0 (IPF)		
Itanium(R) Processor Family	IPF	
Java(TM)	Java	
Red Hat Enterprise Linux AS 3.0	Linux	
Red Hat Enterprise Linux AS 3.0 (IPF)		
TPBroker for C++	TPBroker	
TPBroker for Java		
TPBroker Object Transaction Monitor	OTM	TPBroker OTM
TPBroker Object Transaction Monitor - Client	OTM - Client	
TPBroker Object Transaction Monitor - Connector for Object Request Broker	OTM - Connector for ORB	

はじめに

製品名称	略称
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 Server(TM) 2003, Standard Edition	Windows Server 2003
Microsoft(R) Windows Server(TM) 2003, Enterprise Edition	
64 ビットバージョン Microsoft(R) Windows Server(TM) 2003, Enterprise Edition (IPF)	Windows Server 2003 (IPF)
Microsoft(R) Windows(R) XP Professional Operating System	Windows XP

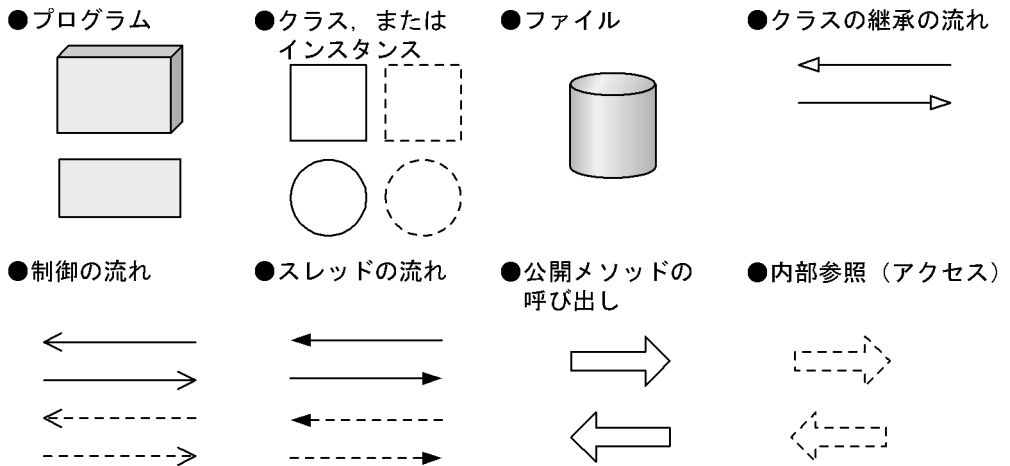
- Windows 2000 , Windows Server 2003 , Windows Server 2003 (IPF) および Windows XP で機能差がない場合 , Windows と表記しています。
- AIX , HP-UX , Linux および Solaris を合わせて UNIX と表記しています。

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	<u>C</u> ommon <u>O</u> bject <u>R</u> equest <u>B</u> roker <u>A</u> rchitecture
DLL	<u>D</u> ynamic <u>L</u> inking <u>L</u> ibrary
IDL	<u>I</u> nterface <u>D</u> efinition <u>L</u> anguage
ORB	<u>O</u> bject <u>R</u> equest <u>B</u> roker
OS	<u>O</u> perating <u>S</u> ystem
QoS	<u>Q</u> uality of <u>S</u> ervice
TSC	<u>T</u> PBroker <u>S</u> chedule <u>C</u> ontrol
URL	<u>U</u> niform <u>R</u> esource <u>L</u> ocator

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。



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 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) <code>tscdmstart [-TSCForceStart]</code> これは, <code>tscdmstart</code> と指定するか, または <code>tscdmstart -TSCForceStart</code> 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) <code>-I ディレクトリ [: ディレクトリ ...]</code> これは, <code>-I</code> 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) <code>-format 1 2</code> これは, <code>-format</code> 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 <code>_</code> (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)

はじめに

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）

目次

1	IDL 文法	1
1.1	IDL 定義一覧	2
1.2	IDL データ型	4
1.3	IDL 文法の注意事項	6
2	アプリケーションプログラムの作成 (C++)	7
2.1	アプリケーションプログラムの作成手順 (C++)	8
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++)	11
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++)	12
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++)	17
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++)	27
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++)	29
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++)	30
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++)	35
2.4	セッション呼び出しをするアプリケーションプログラム (C++)	45
2.4.1	セッ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++)	45
2.4.2	セッション呼び出しをするサーバアプリケーションの例 (C++)	52
2.5	TSCContext を利用するアプリケーションプログラム (C++)	53
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++)	53
2.5.2	TSCContext を利用するサーバアプリケーションの例 (C++)	58
2.6	TSCThread を利用するアプリケーションプログラム (C++)	61
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++)	61
2.6.2	TSCThread を利用するサーバアプリケーションの例 (C++)	61
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++)	73
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++)	74
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++)	79
2.8	TSCWatchTime を利用するアプリケーションプログラム (C++)	89
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++)	89
2.8.2	TSCWatchTime を利用するサーバアプリケーションの例 (C++)	89
3	アプリケーションプログラミングインタフェース (C++)	93
	クラスの一覧 (C++)	94

クラス関連図 (C++)	97
公開メソッド呼び出しと内部参照 (C++)	100
ABC_TSCacpt (C++)	102
ABC_TSCfactimpl (C++)	104
ABC_TSCimpl (C++)	105
ABC_TSCprxy (C++)	106
ABC_TSCsk (C++)	108
ABC_TSCspxy (C++)	110
TSCAcceptor (C++)	112
TSCAdm (C++)	117
TSCClient (C++)	125
TSCContext (C++)	128
TSCDomain (C++)	131
TSCObject (C++)	133
TSCObjectFactory (C++)	137
TSCProxyObject (C++)	140
TSCRootAcceptor (C++)	146
TSCServer (C++)	155
TSCSessionProxy (C++)	158
TSCSystemException (C++)	165
TSCSystemException の派生クラス (C++)	175
TSCThread (C++)	186
TSCThreadFactory (C++)	187
TSCWatchTime (C++)	190

4

アプリケーションプログラムの作成 (Java)	193
4.1 アプリケーションプログラムの作成手順 (Java)	194
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)	197
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)	198
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)	202
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)	209
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)	212
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)	213
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)	217
4.4 セッション呼び出しをするアプリケーションプログラム (Java)	225

4.4.1	セッ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)	225
4.4.2	セッション呼び出しをするサーバアプリケーションの例 (Java)	230
4.5	TSCContext を利用するアプリケーションプログラム (Java)	231
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)	231
4.5.2	TSCContext を利用するサーバアプリケーションの例 (Java)	235
4.6	TSCThread を利用するアプリケーションプログラム (Java)	237
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)	237
4.6.2	TSCThread を利用するサーバアプリケーションの例 (Java)	237
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)	246
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)	247
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)	251
4.8	TSCWatchTime を利用するアプリケーションプログラム (Java)	259
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)	259
4.8.2	TSCWatchTime を利用するサーバアプリケーションの例 (Java)	259

5

アプリケーションプログラミングインタフェース (Java)	263
クラスの一覧 (Java)	264
クラス関連図 (Java)	266
公開メソッド呼び出しと内部参照 (Java)	269
ABC_TSCacpt (Java)	271
ABC_TSCfactimpl (Java)	273
ABC_TSCimpl (Java)	274
ABC_TSCprxy (Java)	275
ABC_TSCsk (Java)	277
ABC_TSCspxy (Java)	278
TSCAcceptor (Java)	280
TSCAdm (Java)	285
TSCClient (Java)	294
TSCContext (Java)	297
TSCDomain (Java)	299
TSCObject (Java)	301
TSCObjectFactory (Java)	304
TSCProxyObject (Java)	306
TSCRootAcceptor (Java)	312
TSCServer (Java)	320

TSCSessionProxy (Java)	323
TSCSystemException (Java)	330
TSCSystemException の派生クラス (Java)	340
TSCThread (Java)	349
TSCThreadFactory (Java)	350
TSCWatchTime (Java)	352

6

コマンドリファレンス	355
コマンドの一覧	356
tscidl2cpp (トランザクションフレームの出力 (C++))	357
tscidl2j (トランザクションフレームの出力 (Java))	364

付録

付録 A エラーコード一覧	367
付録 B 場所コード一覧	368
付録 C 完了状態一覧	370
付録 D 内容コード一覧	371
付録 D.1 内容コード 1000 ~ 1999	372
付録 D.2 内容コード 2000 ~ 2999	378
付録 D.3 内容コード 3000 ~ 3999	384
付録 D.4 内容コード 4000 ~ 4999	395
付録 D.5 内容コード 5000 ~ 5999	399
付録 D.6 内容コード 6000 ~ 6999	404
付録 D.7 内容コード 7000 ~ 7999	414
付録 D.8 内容コード 8000 ~ 8999	415
付録 D.9 内容コード 9000 ~ 9999	416
付録 D.10 内容コード 10000 ~ 10999	421
付録 D.11 内容コード 11000 ~ 11999	421
付録 D.12 内容コード 12000 ~ 12999	424
付録 D.13 内容コード 13000 ~ 13999	425
付録 D.14 内容コード 14000 ~ 14999	427
付録 D.15 内容コード 15000 ~ 15999	428
付録 D.16 内容コード 16000 ~ 16999	430
付録 D.17 内容コード 17000 以降	

索引

図目次

図 2-1	アプリケーションプログラムを作成する手順 (C++)	8
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++)	27
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++)	28
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++)	98
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++)	99
図 3-3	システム提供クラスのインスタンス関連図 (C++)	101
図 4-1	アプリケーションプログラムを作成する手順 (Java)	194
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)	210
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)	211
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)	267
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)	268
図 5-3	システム提供クラスのインスタンス関連図 (Java)	270

表目次

表 1-1	IDL 定義の使用の可否	2
表 1-2	#include 文で指定される IDL 定義の使用の可否	2
表 1-3	operation 定義の使用の可否	3
表 1-4	IDL データ型の使用の可否	4
表 1-5	Primitive 型データの使用の可否	5
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++)	11
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++)	29
表 2-3	セッ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++)	45
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++)	73
表 3-1	クラス一覧 (C++)	94
表 3-2	インスタンス数 (C++)	100
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++)	123
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++)	123
表 3-5	エラーコードの一覧 (C++)	165
表 3-6	内容コードの分類 (C++)	166
表 3-7	場所コードの一覧 (C++)	167
表 3-8	完了状態の一覧 (C++)	167
表 3-9	OTM のシステム例外 (C++)	175
表 3-10	OTM の内容コードの分類 (C++)	176
表 3-11	OTM の場所コード (C++)	177
表 3-12	OTM の完了状態 (C++)	177
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)	197
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)	212
表 4-3	セッ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)	225
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)	247
表 5-1	クラス一覧 (Java)	264

表 5-2	インスタンス数 (Java)	269
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)	292
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)	292
表 5-5	エラーコードの一覧 (Java)	330
表 5-6	内容コードの分類 (Java)	331
表 5-7	場所コードの一覧 (Java)	332
表 5-8	完了状態の一覧 (Java)	332
表 5-9	OTM のシステム例外 (Java)	340
表 5-10	OTM の内容コード (Java)	341
表 5-11	OTM の場所コード (Java)	342
表 5-12	OTM の完了状態 (Java)	342
表 6-1	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	356
表 A-1	エラーコード一覧	368
表 B-1	場所コード一覧	370
表 C-1	完了状態一覧	371

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

1.2 IDL データ型

1.3 IDL 文法の注意事項

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	
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	
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	(セッ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++ および Java)
Primitive	1
struct	
sequence	(最大長指定は使用できません)
array	
union	2
enum	
interface	× (データ型として使用できません)
any	(Primitive 型または sequence<octet> 型だけ使用できます)
typedef	(array 型または 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)
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	x	
wchar		
wstring		(最大長指定は使用できません)

(凡例)

: 使用できます。

x : 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。

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++)

 - 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++)

 - 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++)

 - 2.4 セッション呼び出しをするアプリケーションプログラム (C++)

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

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

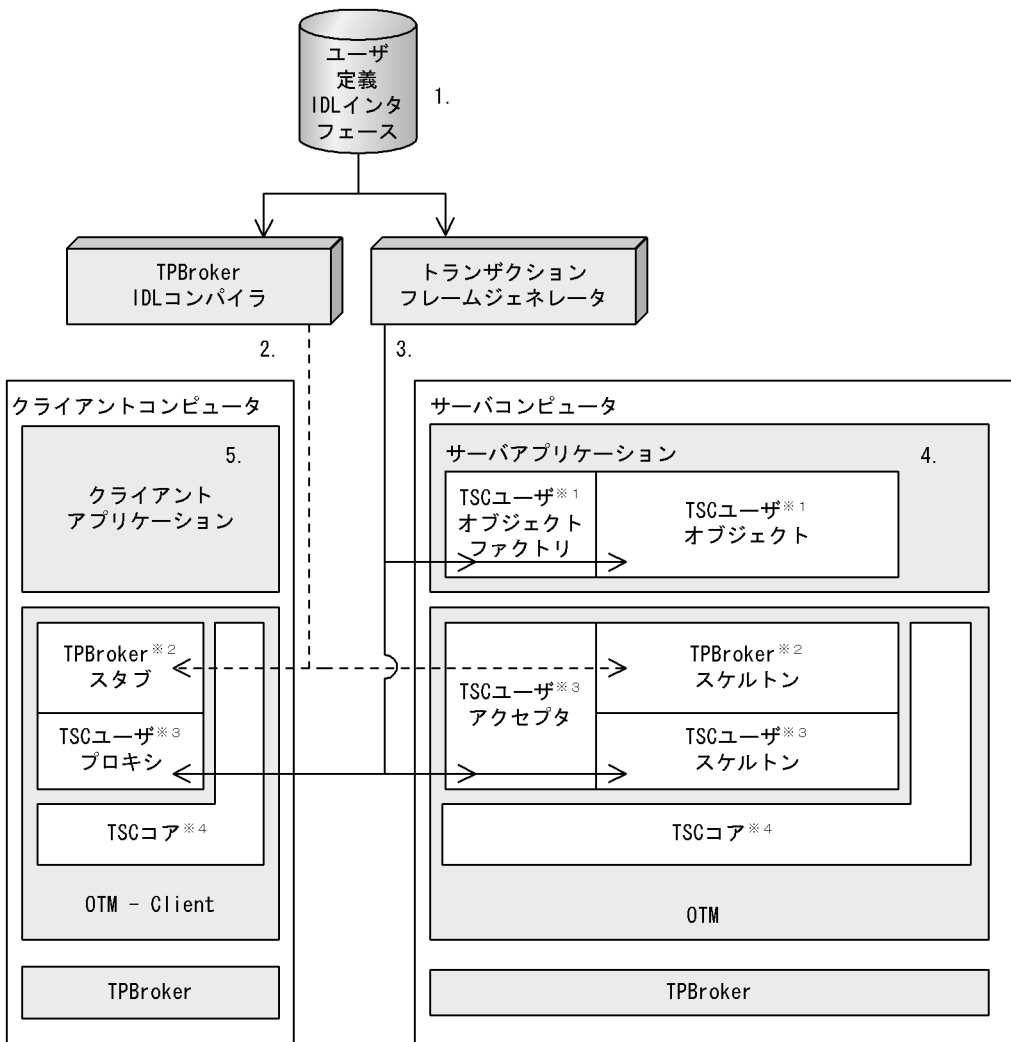
 - 2.7 ユーザ例外通知を利用するアプリケーションプログラム (C++)

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

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

2. アプリケーションプログラムの作成 (C++)

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	<ul style="list-style-type: none"> • ABCfile_TSC_c.hh • ABCfile_TSC_c.cc 	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • ABCfile_TSC_s.hh • ABCfile_TSC_s.cc 	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • ABCfile_TSC_t.hh • ABCfile_TSC_t.cc 	<ul style="list-style-type: none"> • 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>  
using namespace std;  
  
#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);
}

```

2. アプリケーションプログラムの作成 (C++)

```
}

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)

```

2. アプリケーションプログラムの作成 (C++)

```
    {
        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>
using namespace std;

#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_var res_data;

try
{
    // サーバのメソッドの呼び出し
    abc->call(req_data, res_data.out());
}
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:

```

2. アプリケーションプログラムの作成 (C++)

```
// 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 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 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)
    m_counter++;
    printf("Call method in ABC_TSCimpl¥n");
}

```

2. アプリケーションプログラムの作成 (C++)

(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>  
using namespace std;  
  
#include <tscadm.h>  
#include <tscobject.h>  
#include <tsceexcept.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の初期化处理
//////////

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;

```

2. アプリケーションプログラムの作成 (C++)

```
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の平行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
    {

```

2. アプリケーションプログラムの作成 (C++)

```
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
    {
        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
    {
        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;

```

2. アプリケーションプログラムの作成 (C++)

```
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++)

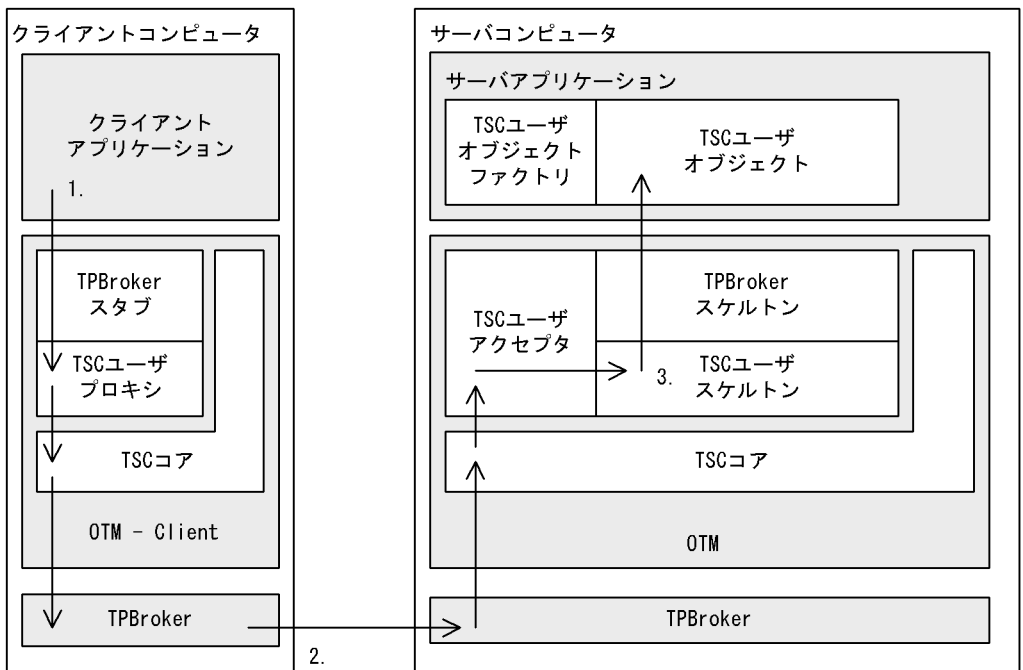
同期型呼び出しをするアプリケーションプログラムを実行した場合の処理シーケンスを示します。

(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++)

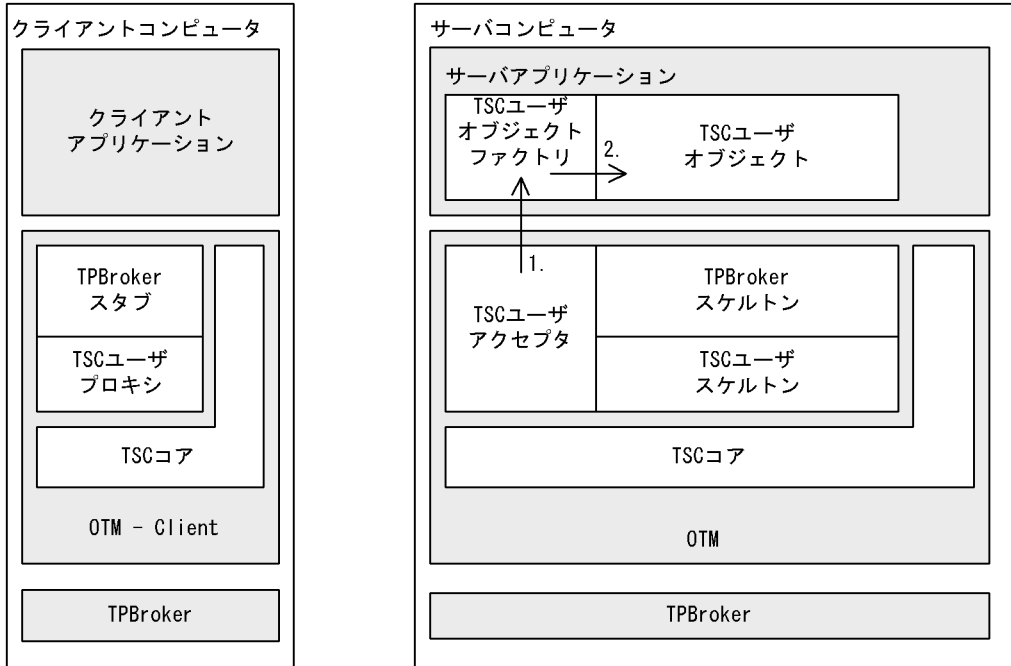
サーバアプリケーションを開始すると、サービスの登録処理が実行されます。TSCRootAcceptor の activate メソッドを呼び出すと、ABC_TSCfactimpl に

2. アプリケーションプログラムの作成 (C++)

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	<ul style="list-style-type: none"> • XYZfile_TSC_c.hh • XYZfile_TSC_c.cc 	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • XYZfile_TSC_s.hh • XYZfile_TSC_s.cc 	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • XYZfile_TSC_t.hh • XYZfile_TSC_t.cc 	<ul style="list-style-type: none"> • 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>
using namespace std;

#include <corba.h>

#include <tscadm.h>
#include <tscproxy.h>
#include <tsceexcept.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);
}

```

2. アプリケーションプログラムの作成 (C++)

```
}

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)

```

2. アプリケーションプログラムの作成 (C++)

```
    {
        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>
using namespace std;

#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);

```

2. アプリケーションプログラムの作成 (C++)

```
};

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

2. アプリケーションプログラムの作成 (C++)

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>
using namespace std;

#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();
    }
}

```

2. アプリケーションプログラムの作成 (C++)

```
    }
    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の平行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

```

2. アプリケーションプログラムの作成 (C++)

```
//////////
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
    {
        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
    {
        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(),

```

2. アプリケーションプログラムの作成 (C++)

```
        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 セッション呼び出しをするアプリケーションプログラム (C++)

セッ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 セッ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	<ul style="list-style-type: none"> • ABCfile_TSC_p.hh • ABCfile_TSC_p.cc 	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • ABCfile_TSC_s.hh • ABCfile_TSC_s.cc 	<ul style="list-style-type: none"> • ABC_TSCsk (TSC ユーザスケルトン) • ABC_TSCaccept (TSC ユーザアクセプタ)
雛形クラス	<ul style="list-style-type: none"> • ABCfile_TSC_t.hh • ABCfile_TSC_t.cc 	<ul style="list-style-type: none"> • 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 セッ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++)

セッ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

2. アプリケーションプログラムの作成 (C++)

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>
using namespace std;

#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)

```

2. アプリケーションプログラムの作成 (C++)

```
{
    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());
}

```

2. アプリケーションプログラムの作成 (C++)

```
        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>
using namespace std;

#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, セッションの開始
    //////////

    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_var 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.out());
    }
}
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, セッションの停止
//////////

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. アプリケーションプログラムの作成 (C++)

2.4.2 セッ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>
using namespace std;

#include <corba.h>

#include <tscadm.h>
```

2. アプリケーションプログラムの作成 (C++)

```
#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
{

```

2. アプリケーションプログラムの作成 (C++)

```
    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);

```

2. アプリケーションプログラムの作成 (C++)

```
};
```

(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 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 <tscontext.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()

```

2. アプリケーションプログラムの作成 (C++)

```
{
    // 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 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"
//
#ifdef _ABCfile_TSC_T_HDR
#define _ABCfile_TSC_T_HDR

#include <tscobject.h>

#include "ABCfile_TSC_s.hh"

class ABC_TSCfactimpl : public TSCObjectFactory
```

2. アプリケーションプログラムの作成 (C++)

```
{
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 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 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。

```

2. アプリケーションプログラムの作成 (C++)

```
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 <tsceexcept.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 <tsceexcept.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

2. アプリケーションプログラムの作成 (C++)

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>
using namespace std;

#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の初期化处理
    //////////

    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の初期化处理
    //////////

    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)
    {

```

2. アプリケーションプログラムの作成 (C++)

```
        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の平行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);
}

```

2. アプリケーションプログラムの作成 (C++)

```
//////////
// 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
    {
        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
    {
        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

```

2. アプリケーションプログラムの作成 (C++)

```
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	<ul style="list-style-type: none"> • EEEfile_TSC_c.hh • EEEfile_TSC_c.cc 	<ul style="list-style-type: none"> • EEE_TSCprxy (TSC ユーザプロキシ)

2. アプリケーションプログラムの作成 (C++)

分類	ファイル名	生成するクラス名 (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	<ul style="list-style-type: none">• EEEfile_TSC_s.hh• EEEfile_TSC_s.cc	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• EEEfile_TSC_t.hh• EEEfile_TSC_t.cc	<ul style="list-style-type: none">• 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>  
using namespace std;  
  
#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(),

```

2. アプリケーションプログラムの作成 (C++)

```
        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)
{

```

2. アプリケーションプログラムの作成 (C++)

```
// 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>
using namespace std;

#include <corba.h>

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

```

2. アプリケーションプログラムの作成 (C++)

```
#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. アプリケーションプログラムの作成 (C++)

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>
using namespace std;

#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の初期化处理
//////////

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

```

2. アプリケーションプログラムの作成 (C++)

```
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の平行カウンタ指定もできます。
    // 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;
    }
}

```

2. アプリケーションプログラムの作成 (C++)

```
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
    {
        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
    {
        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;

//////////

```

2. アプリケーションプログラムの作成 (C++)

```
// 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"
//
#ifdef _ABCfile_TSC_T_HDR
#define _ABCfile_TSC_T_HDR

#include <tscobject.h>

#include "ABCfile_TSC_s.hh"

class ABC_TSCfactimpl : public TSCObjectFactory
{
public:
```

2. アプリケーションプログラムの作成 (C++)

```
// 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 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 <tsceexcept.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;
    }
}

```

2. アプリケーションプログラムの作成 (C++)

```
}  
  
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 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++)

クラス関連関 (C++)

公開メソッド呼び出しと内部参照 (C++)

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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• TSCSystemException

分類	クラス
	<ul style="list-style-type: none"> • TSCSystemException の派生クラス • TSCBadContextException • TSCBadInvOrderException • TSCBadOperationException • TSCBadParamException • TSCBadTypecodeException • TSCCodesetIncompatibleException • TSCCommFailureException • TSCDataConversionException • TSCFreeMemException • TSCImpLimitException • TSCInitializeException • TSCInternalException • TSCIntfReposException • TSCInvalidTransactionException • TSCInvFlagException • TSCInvIdentException • TSCInvObjrefException • TSCInvPolicyException • TSCMarshalException • TSCNoImplementException • TSCNoMemoryException • TSCNoPermissionException • TSCNoResourcesException • TSCNoResponseException • TSCObjAdapterException • TSCObjectNotExistException • TSCPersistStoreException • TSCRebindException • TSCTimeoutException • TSCTransactionRequiredException • TSCTransactionRolledbackException • 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)	<ul style="list-style-type: none"> • ABC_TSCapt (TSCAcceptor) • ABC_TSCprxy (TSCProxyObject) • ABC_TSCsk (TSCObject) • ABC_TSCspxy (TSCSessionProxy)
雛形クラス (基底クラス)	<ul style="list-style-type: none"> • ABC_TSCfactimpl (ABC_TSCfacts) • 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;
```

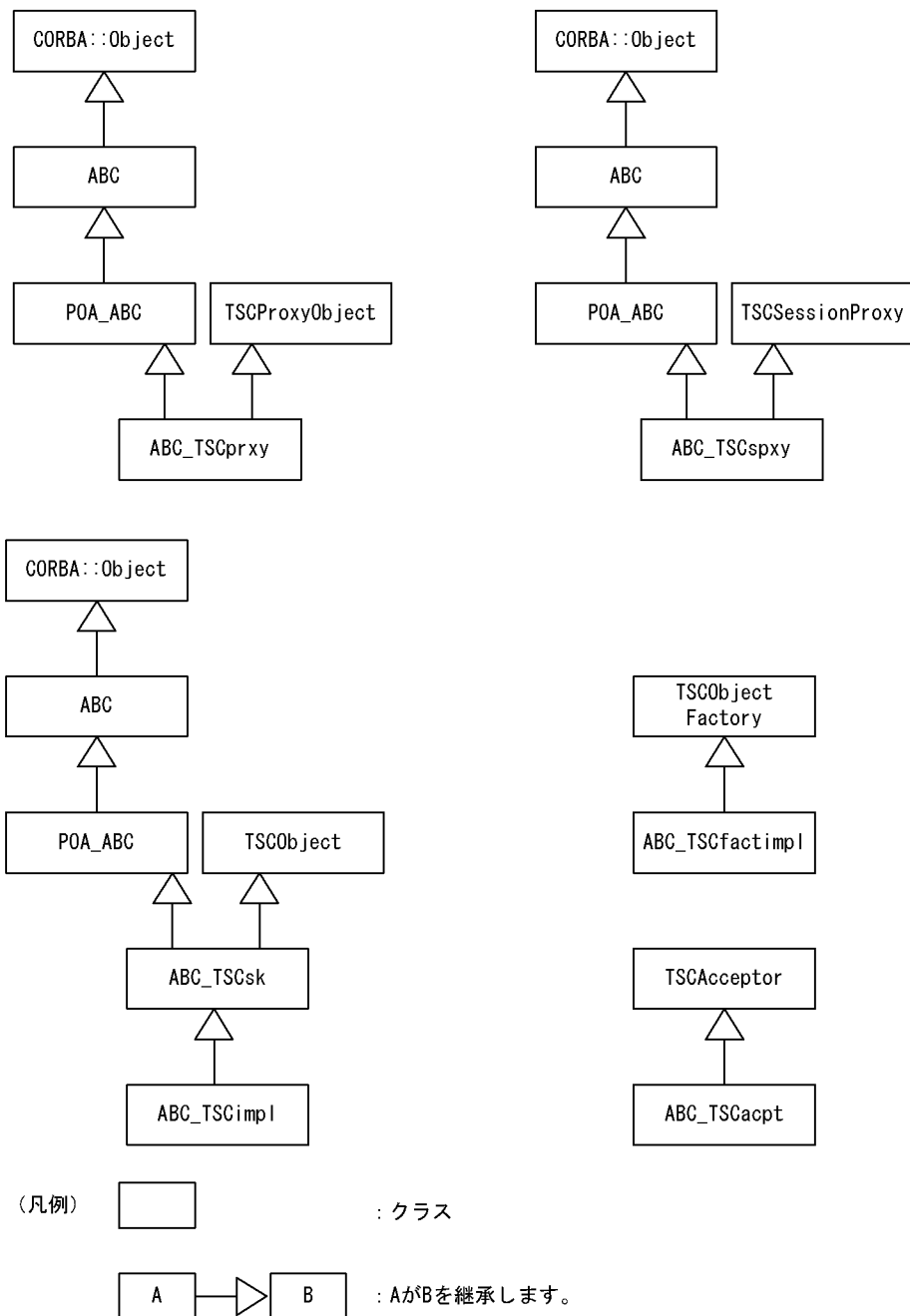
クラスの一覧 (C++)

```
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++)

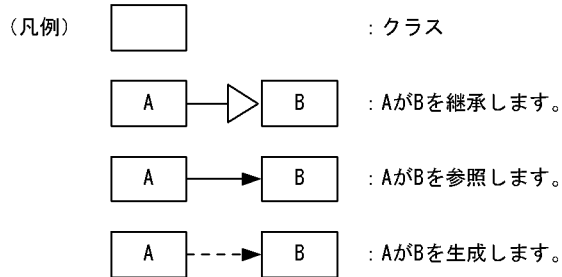
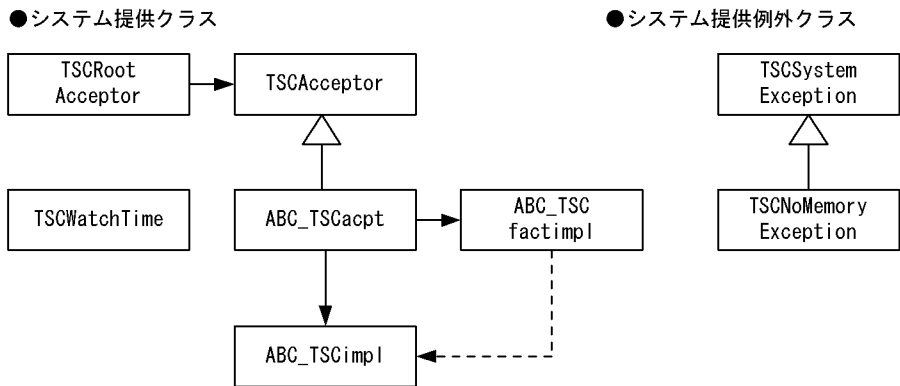
システム提供クラスから派生するユーザ定義 IDL インタフェース依存クラスの関連を次の図に示します。

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)

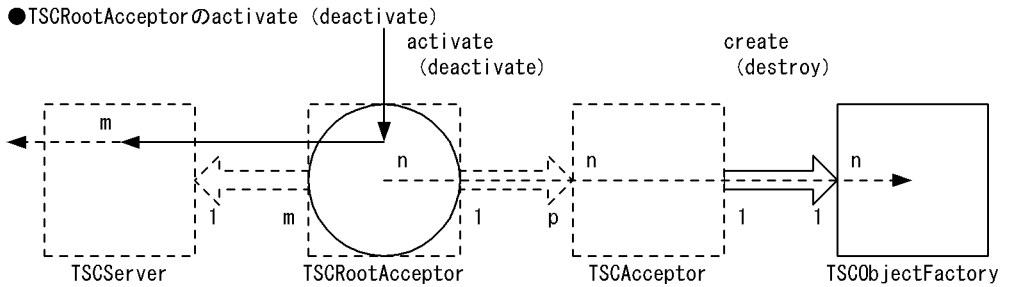
OTM のシステム提供クラスのインスタンスが、ほかのインスタンスの非公開メソッドを内部的にアクセスすることです。またはそのインスタンスを内部的に参照することです。

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。

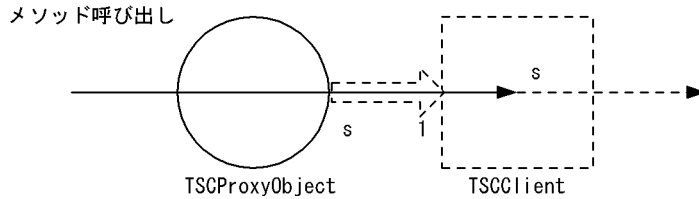
表 3-2 インスタンス数 (C++)

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)

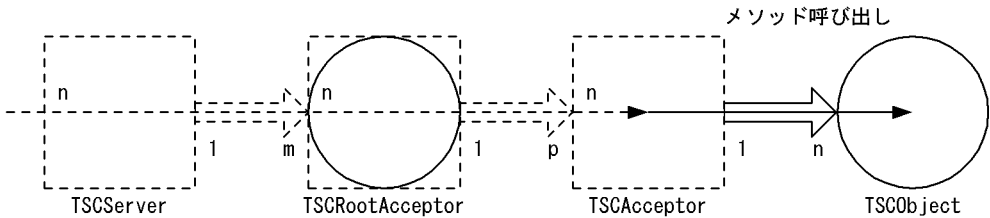
図 3-3 システム提供クラスのインスタンス関連図 (C++)



●OTMのオブジェクト呼び出し (クライアント)



●OTMのオブジェクト呼び出し (サーバ)



(凡例)

○ : 公開メソッドが呼び出される
マルチスレッドセーフではない
インスタンス

⊙ : 公開メソッドが呼び出される際,
マルチスレッドセーフではないが,
内部参照 (アクセス) される際,
マルチスレッドセーフなインスタンス

□ : 公開メソッドが呼び出される
マルチスレッドセーフな
インスタンス

⊡ : 内部参照 (アクセス) される
マルチスレッドセーフな
インスタンス

⇒ : 公開メソッドの呼び出し
x y

⊡ : 内部参照 (アクセス)
x y

1:1 単数のインスタンスから
単数のインスタンスへの呼び出し
1:n 単数のインスタンスから
複数のインスタンスへの呼び出し
n:1 複数のインスタンスから
単数のインスタンスへの呼び出し

1:1 単数のインスタンスから
単数のインスタンスへの内部参照
1:n 単数のインスタンスから
複数のインスタンスへの内部参照
n:1 複数のインスタンスから
単数のインスタンスへの内部参照

→ : シングルスレッド

n ---→ : マルチスレッド (並行度n)

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 のサーバントクラスである POA_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 のサーバントクラスである POA_ABC も継承します。また、ABC_TSCprxy の派生クラスも同様に、TPBroker のサーバントクラスを継承します。

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 POA_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 のサーバントクラスである POA_ABC も継承します。また、ABC_TSCsk の派生クラスも同様に、TPBroker のサーバントクラスを継承します。

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 POA_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、セッ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 のサーバントクラスである POA_ABC も継承します。また、ABC_TSCspxy の派生クラスも同様に、TPBroker のサーバントクラスを継承します。

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 POA_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せます。

```
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せます。

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せます。

```
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せます。

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せます。

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当てられている TSCThread を取得できます。

TSCObject の呼び出し時の TSC アクセプタ名称の取得

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 `_TSCAcceptorName` を呼び出すことによって、サービス提供メソッドの呼び出し元である TSCAcceptor の TSC アクセプタ名称を取得できます。

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();

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

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。

```
const char* _TSCAcceptorName()
```

項目	型・意味
戻り値	TSC アクセプタ名称

TSCObject を保持している TSCAcceptor の TSC アクセプタ名称を取得します。

戻り値の 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。

メソッド	複数のスレッド上からの同時呼び出し
_TSCAcceptorName	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entrant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entrantなコードを記述する必要があります。

なお、コネクション切断などの障害時には、ユーザオブジェクトを生成したスレッドとは別のスレッドで destroy() を発行することがあります。

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 を呼び出せます。

"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 を呼び出せます。

"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
_TSPriority(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。

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();

};

```

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。

activate() や deactivate() に失敗した場合、TSCRootAcceptor 上でユーザオブジェクトの生成や削除を実施していることがあります。TSCRootAcceptor を削除する場合は、deactivate() が正常終了したあとで実施してください。

なお、このメソッドを複数のスレッドから同時に呼び出せます。

```
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。

activate() や deactivate() に失敗した場合，TSCRootAcceptor 上でユーザオブジェクトの生成や削除を実施していることがあります。TSCRootAcceptor を削除する場合は，deactivate() が正常終了したあとで実施してください。

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せません。

```
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せません。

```
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せます。

```
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。

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()	×

(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。

メソッド	内部参照
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
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せません。

```
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せません。

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	できません。
getTSCID	できません。

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、セッ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() メソッドでセッションを確立します。

セッ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ッ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ッ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

セッションを開始します。

セッ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、セッ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

セッションを解放します。

セッ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	セッション呼び出しインターバル監視時間 (秒)

セッ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	セッ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	

セッ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。

- セッション呼び出し中の場合
セッ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
- セッ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、セッ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 _TSCPriorty()`

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 _TSCPriorty(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

セッション呼び出し機能使用中のメソッド呼び出し規則

セッ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	セッ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
TRANSACTION_REQUIRED	TSCTransactionRequiredException
TRANSACTION_ROLLEDBACK	TSCTransactionRolledbackException
INVALID_TRANSACTION	TSCInvalidTransactionException
INV_POLICY	TSCInvPolicyException
CODESET_INCOMPATIBLE	TSCCodesetIncompatibleException
REBIND	TSCRebindException
TIMEOUT	TSCTimeoutException

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 ~ 40000

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;
static const TSCInt TRANSACTION_REQUIRED = 27;
static const TSCInt TRANSACTION_ROLLEDBACK = 28;
static const TSCInt INVALID_TRANSACTION = 29;
static const TSCInt INV_POLICY = 30;
static const TSCInt CODESET_INCOMPATIBLE = 31;
static const TSCInt REBIND = 32;
static const TSCInt TIMEOUT = 33;

//
//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 INVALID_FLAG = 1022;
static const TSCInt INVALID_WATCH_TIME = 1027;
static const TSCInt INVALID_RETRY_REQUIREMENT = 1029;
static const TSCInt INVALID_SESSION_INTERVAL = 1038;

//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;
static const TSCInt RECEIVE_CONN_DOWN = 3023;

//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_REGISTERED = 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;

```

TSCSystemException (C++)

```
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 SYSTEM_TIME_FAILURE = 5010;
static const TSCInt CONN_INFO_FAILURE = 5011;
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;

//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_CLIEN = 10008;
static const TSCInt OVER_MAX_DISPATCH_PARALLEL = 10009;
static const TSCInt OVER_MAX_REQUEST_COUNT = 10010;
static const TSCInt OVER_MAX_ORB_CLIEN = 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;
static const TSCInt ALREADY_DISCONN = 12019;

//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;
static const TSCInt TPBROKER_TRANSACTION_REQUIRED = 27998;
static const TSCInt TPBROKER_TRANSACTION_ROLLEDBACK = 28998;
static const TSCInt TPBROKER_INVALID_TRANSACTION = 29998;
static const TSCInt TPBROKER_INV_POLICY = 30998;
static const TSCInt TPBROKER_CODESET_INCOMPATIBLE = 31998;
static const TSCInt TPBROKER_REBIND = 32998;
static const TSCInt TPBROKER_TIMEOUT = 33998;

//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 <tsceexcept.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。
TSCCodesetIncompatibleException	コードセット間の通信に失敗しました。
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。
TSCInvalidTransactionException	トランザクションコンテキストが不正です。
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。
TSCInvPolicyException	呼び出しに使用したポリシーが不正です。
TSCMarshalException	パラメタ、および結果のマーシャル処理中に障害が発生しました。
TSCNoImplementException	オペレーションが実装できません。

例外名	説明
TSCNoMemoryException	動的メモリ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てしました。
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	リクエストへの応答はまだできません。
TSCObjAdapterException	オブジェクトアダプタが障害を検出しました。
TSCObjectNotExistException	オブジェクトが使用可能状態ではありません。
TSCPersistStoreException	パーシステントストレージ障害が発生しました。
TSCRebindException	設定された QoS ポリシーと IOR が矛盾しています。
TSCTimeoutException	リクエストでタイムアウトが発生しました。
TSCTransactionRequiredException	null 以外のトランザクションコンテキストが必要です。
TSCTransactionRolledbackException	トランザクションがロールバックしました。
TSCTransientException	トランジェント障害が発生しました。
TSCUnknownException	java.lang.Exception 型が発生した場合、Java 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 ~ 40000

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 TSCCodesetIncompatibleException
    : public TSCSystemException
{
public:
    TSCCodesetIncompatibleException(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 TSCInvalidTransactionException
    : public TSCSystemException
{
public:
    TSCInvalidTransactionException(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 TSCInvPolicyException
    : public TSCSystemException
{
public:
    TSCInvPolicyException(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 TSCRebindException
  : public TSCSystemException
{
public:
    TSCRebindException(TSCInt _detail_code,
                      TSCInt _place_code,
                      TSCInt _completion_status,
                      TSCInt _maintenance_code1,
                      TSCInt _maintenance_code2,
                      TSCInt _maintenance_code3,
                      TSCInt _maintenance_code4);
};

class TSCTimeoutException
  : public TSCSystemException
{
public:

```

```

        TSCTimeoutException(TSCInt _detail_code,
                            TSCInt _place_code,
                            TSCInt _completion_status,
                            TSCInt _maintenance_code1,
                            TSCInt _maintenance_code2,
                            TSCInt _maintenance_code3,
                            TSCInt _maintenance_code4);
};

class TSCTransactionRequiredException
    : public TSCSystemException
{
public:
    TSCTransactionRequiredException(TSCInt _detail_code,
                                    TSCInt _place_code,
                                    TSCInt _completion_status,
                                    TSCInt _maintenance_code1,
                                    TSCInt _maintenance_code2,
                                    TSCInt _maintenance_code3,
                                    TSCInt _maintenance_code4);
};

class TSCTransactionRolledbackException
    : public TSCSystemException
{
public:
    TSCTransactionRolledbackException(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,

```

```
};
```

インクルードファイル

```
#include <tsexcept.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。スレッドに割り当て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。

なお、コネクション切断などの障害時には、TSC ユーザスレッドを生成したスレッドとは別のスレッドで destroy() を発行することがあります。

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)

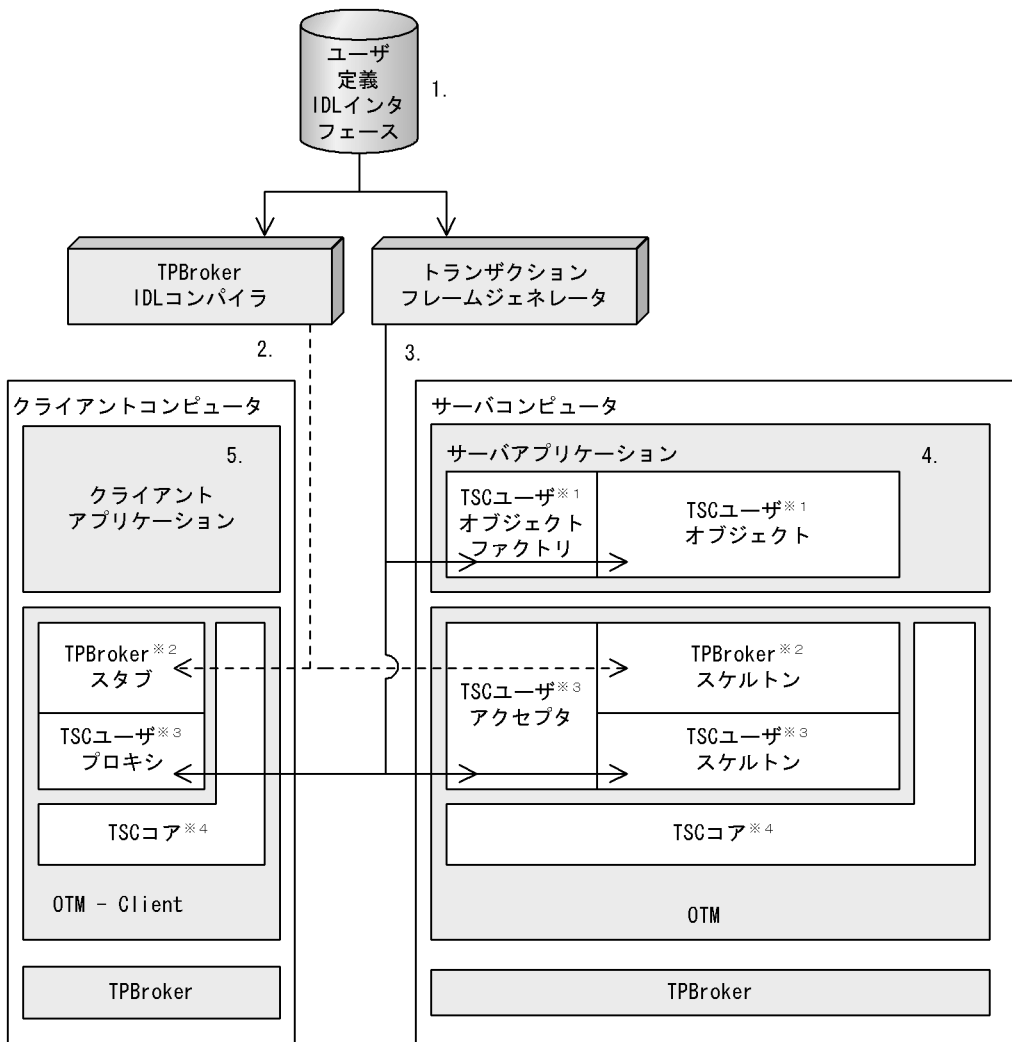
この章では、Java でアプリケーションプログラムを作成する方法について説明します。

-
- 4.1 アプリケーションプログラムの作成手順 (Java)
 - 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)
 - 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)
 - 4.4 セッション呼び出しをするアプリケーションプログラム (Java)
 - 4.5 TSCContext を利用するアプリケーションプログラム (Java)
 - 4.6 TSCThread を利用するアプリケーションプログラム (Java)
 - 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)
 - 4.8 TSCWatchTime を利用するアプリケーションプログラム (Java)
-

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。
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

4. アプリケーションプログラムの作成 (Java)

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。

- _ABCStub.java
- ABC.java
- ABCHelper.java
- ABCHolder.java
- ABCOperations.java
- ABCPOA.java
- ABCPOATie.java
- OctetSeqHelper.java
- OctetSeqHolder.java

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)

4. アプリケーションプログラムの作成 (Java)

分類	生成するクラス名 (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	<ul style="list-style-type: none">• ABC_TSCsk (TSC ユーザスケルトン)• ABC_TSCaccept (TSC ユーザアクセプタ)
雛形クラス	<ul style="list-style-type: none">• 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();
    }

```

4. アプリケーションプログラムの作成 (Java)

```
    }
    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)
    {

        //////////

```

4. アプリケーションプログラムの作成 (Java)

```
//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();
    };
};

```

4. アプリケーションプログラムの作成 (Java)

```
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
    {

```

4. アプリケーションプログラムの作成 (Java)

```
        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の平行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

```

4. アプリケーションプログラムの作成 (Java)

```
//////////
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。

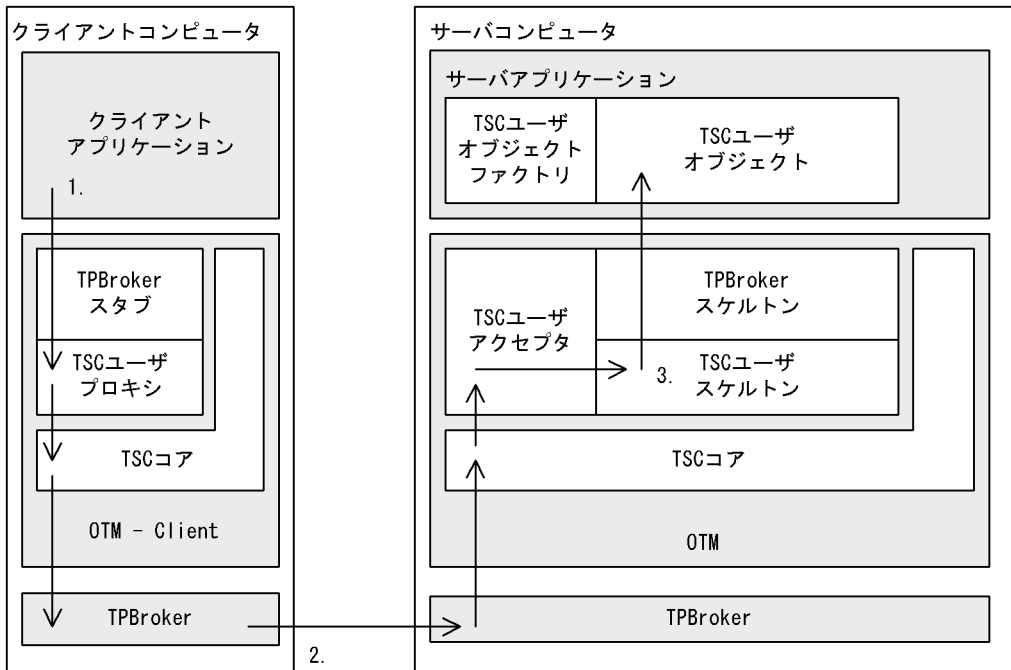
(1) クライアントアプリケーションの開始 (Java)

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。

4. アプリケーションプログラムの作成 (Java)

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)



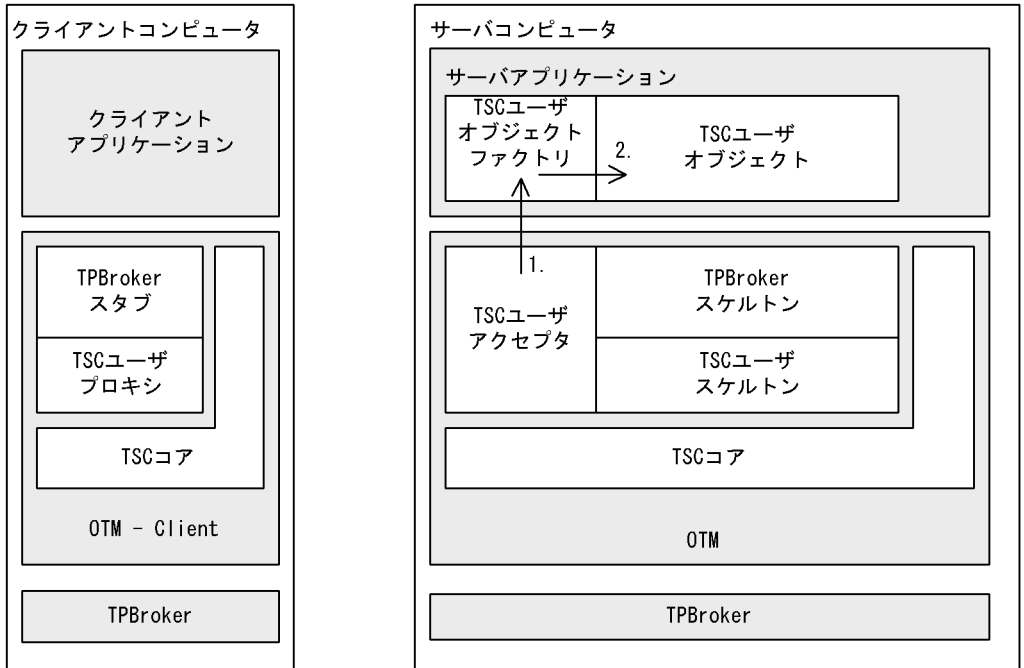
1. TSC ユーザプロキシのメソッドが呼び出されます。
2. スケジューリングおよび TPBroker による通信が実行されます。
3. TSC ユーザオブジェクトのメソッドが呼び出されます。

(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。

- _XYZStub.java
- XYZ.java
- XYZHelper.java
- XYZHolder.java
- XYZOperations.java
- XYZPOA.java
- XYZPOATie.java

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
```

4. アプリケーションプログラムの作成 (Java)

```
{
    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)
{

```

4. アプリケーションプログラムの作成 (Java)

```
// 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)
    }
}

```

4. アプリケーションプログラムの作成 (Java)

```
        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);
        }
    }
}
```

4. アプリケーションプログラムの作成 (Java)

```
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);
}

```

4. アプリケーションプログラムの作成 (Java)

```
// TSCRootAcceptorの平行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);
    }
}

```

4. アプリケーションプログラムの作成 (Java)

```
    }  
  
    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 セッション呼び出しをするアプリケーションプログラム (Java)

セッ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 セッ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 セッ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)

セッ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 の初期化处理
2. TPBroker OTM の初期化处理
3. TSC デモンへの接続
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 デモンへの接続解放

4. アプリケーションプログラムの作成 (Java)

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);
}

```

4. アプリケーションプログラムの作成 (Java)

```
//////////
// 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, セッションの開始
        //////////
        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)

```

4. アプリケーションプログラムの作成 (Java)

```
{
    //例外処理
    System.out.println(tsc_se);
    try{
        my_proxy._TSCStop();
    }
    catch(TSCSystemException se)
    {}
    throw tsc_se;
}

////////
// 3, セッションの停止
////////
try
{
    my_proxy._TSCStop();
}
catch(TSCSystemException tsc_se)
{
    //例外処理
    System.out.println(tsc_se);
    throw tsc_se;
}
}
}
```

4.4.2 セッ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)
    {
        //////////
    }
}
```

4. アプリケーションプログラムの作成 (Java)

```
// 1, TPBrokerの初期化处理
//////////
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の初期化处理
//////////
// 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

```

4. アプリケーションプログラムの作成 (Java)

```
{
    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
    }
}

```

4. アプリケーションプログラムの作成 (Java)

```
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
```

4. アプリケーションプログラムの作成 (Java)

```
{
    // 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

4. アプリケーションプログラムの作成 (Java)

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

```

4. アプリケーションプログラムの作成 (Java)

```
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の平行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)
    {

```

4. アプリケーションプログラムの作成 (Java)

```
        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
- _st_ EEE.java
- _tie_ EEE.java
- EEEPOA.java
- EEEPOATie.java
- _EEESTub.java

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_TSCaopt (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;
```

4. アプリケーションプログラムの作成 (Java)

```
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)

```

4. アプリケーションプログラムの作成 (Java)

```
        {
            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()
    {

```

4. アプリケーションプログラムの作成 (Java)

```
// 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);
        }
    }
}

```

4. アプリケーションプログラムの作成 (Java)

```
        System.exit(1);
    }

    ////////////
    // 2, TPBroker OTMの初期化处理
    ////////////
    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);
}

```

4. アプリケーションプログラムの作成 (Java)

```
}  
  
try  
{  
    // TSCRootAcceptorへの登録  
    my_rt_acpt.registerAcceptor(my_acpt);  
  
    // TSCRootAcceptorの平行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();
    }
}

```

4. アプリケーションプログラムの作成 (Java)

```
        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()
    {
```

4. アプリケーションプログラムの作成 (Java)

```
// 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)

クラス関連図 (Java)

公開メソッド呼び出しと内部参照 (Java)

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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• TSCSystemException

分類	クラス	
	<ul style="list-style-type: none"> • TSCSystemException の派生クラス 	<ul style="list-style-type: none"> • TSCBadContextException • TSCBadInvOrderException • TSCBadOperationException • TSCBadParamException • TSCBadTypecodeException • TSCCodesetIncompatibleException • TSCCommFailureException • TSCDataConversionException • TSCFreeMemException • TSCImpLimitException • TSCInitializeException • TSCInternalException • TSCIntfReposException • TSCInvalidTransactionException • TSCInvFlagException • TSCInvIdentException • TSCInvObjrefException • TSCInvPolicyException • TSCMarshalException • TSCNoImplementException • TSCNoMemoryException • TSCNoPermissionException • TSCNoResourcesException • TSCNoResponseException • TSCObjAdapterException • TSCObjectNotExistException • TSCPersistStoreException • TSCRebindException • TSCTimeoutException • TSCTransactionRequiredException • TSCTransactionRolledbackException • 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)	<ul style="list-style-type: none"> • ABC_TSCacpt (TSCAcceptor) • ABC_TSCprxy (TSCProxyObject) • ABC_TSCsk (TSCObject) • ABC_TSCspxy (TSCSessionProxy) 	
雑形クラス (基底クラス)	<ul style="list-style-type: none"> • ABC_TSCfactimpl (ABC_TSCfactsk) • ABC_TSCimpl (ABC_TSCsk) 	

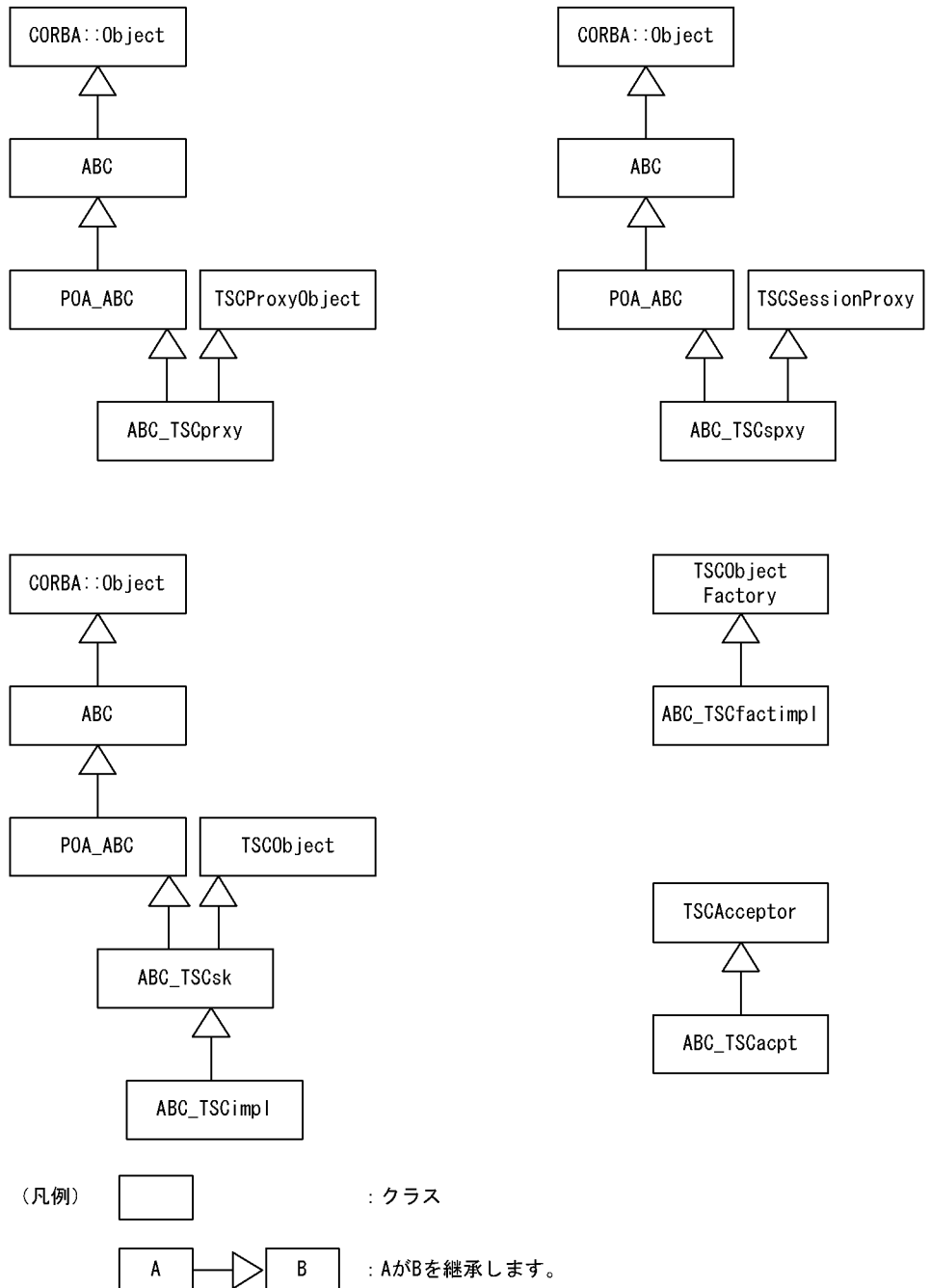
基本データ型 (Java)

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。

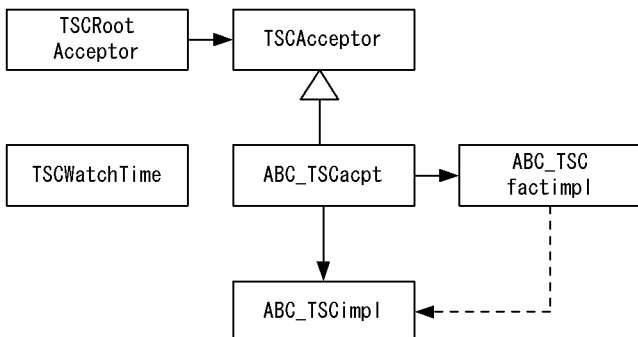
図 5-1 システム提供クラスから派生するユーザ定義 IDL インタフェース依存クラス (Java)



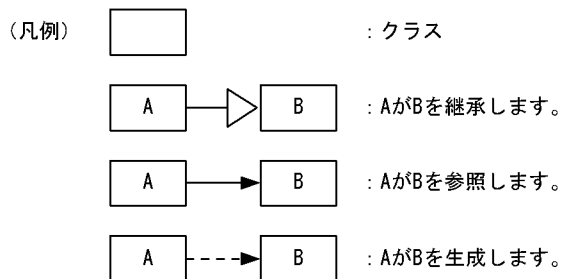
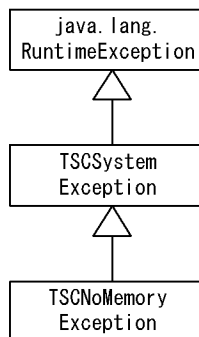
サーバアプリケーションを作成する際に使用するシステム提供クラスおよびシステム提供例外クラスの関連を次の図に示します。

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)

●システム提供クラス



●システム提供例外クラス



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)

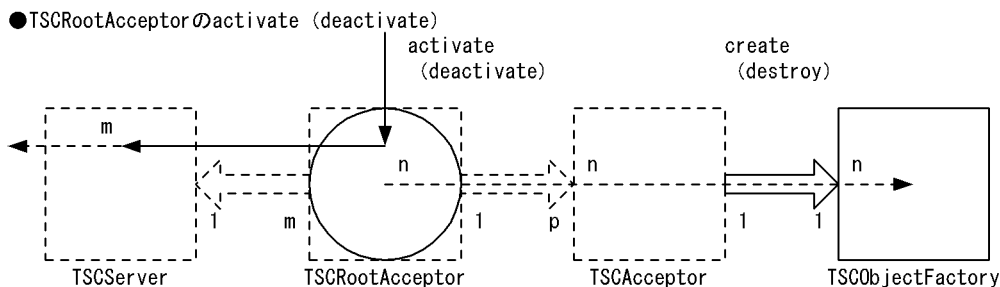
OTM のシステム提供クラスのインスタンスが、ほかのインスタンスの非公開メソッドを内部的にアクセスすることです。またはそのインスタンスを内部的に参照することです。

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。

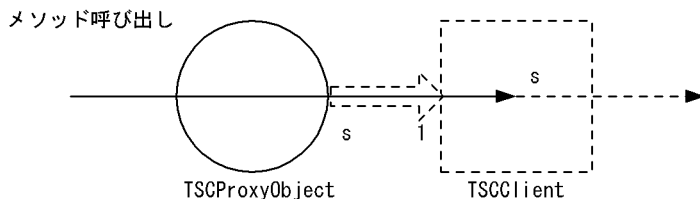
表 5-2 インスタンス数 (Java)

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)

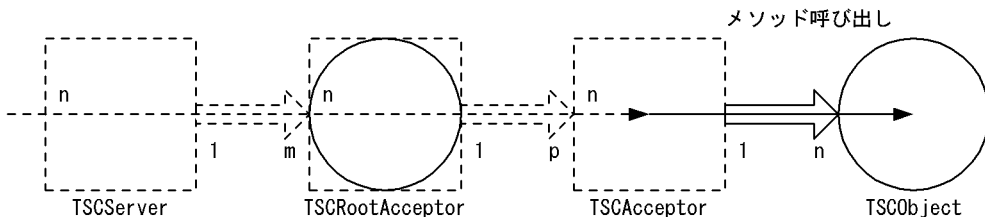
図 5-3 システム提供クラスのインスタンス関連図 (Java)



●OTMのオブジェクト呼び出し (クライアント)



●OTMのオブジェクト呼び出し (サーバ)



(凡例)

○ : 公開メソッドが呼び出される
マルチスレッドセーフではない
インスタンス

□ : 公開メソッドが呼び出される
マルチスレッドセーフな
インスタンス

⇒ : 公開メソッドの呼び出し
x y

1:1 単数のインスタンスから
単数のインスタンスへの呼び出し
1:n 単数のインスタンスから
複数のインスタンスへの呼び出し
n:1 複数のインスタンスから
単数のインスタンスへの呼び出し

→ : シングルスレッド

○ (dashed) : 公開メソッドが呼び出される際,
マルチスレッドセーフではないが,
内部参照 (アクセス) される際,
マルチスレッドセーフなインスタンス

□ (dashed) : 内部参照 (アクセス) される
マルチスレッドセーフな
インスタンス

⇒ (dashed) : 内部参照 (アクセス)
x y

1:1 単数のインスタンスから
単数のインスタンスへの内部参照
1:n 単数のインスタンスから
複数のインスタンスへの内部参照
n:1 複数のインスタンスから
単数のインスタンスへの内部参照

n → (dashed) : マルチスレッド (並行度n)

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 tsc_object_fact);
    public ABC_TSCacpt (TSCObjectFactory tsc_object_fact,
                        String tsc_acpt_name);
}
```

コンストラクタ

```
public ABC_TSCacpt(TSCObjectFactory tsc_object_fact)
```

項目	型・意味	
引数	TSCObjectFactory 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 tsc_object_fact,
                  String tsc_acpt_name)
```

項目	型・意味	
引数	TSCObjectFactory tsc_object_fact	TSCObjectFactory オブジェクト
	String 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

ABC_TSCacpt (Java)

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。

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 のサーバントクラスである POA_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 のサーバントクラスである POA_ABC も継承します。また、ABC_TSCprxy の派生クラスも同様に、TPBroker のサーバントクラスを継承します。

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 POA_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 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 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 のサーバントクラスである POA_ABC も継承します。また、ABC_TSCsk の派生クラスも同様に、TPBroker のサーバントクラスを継承します。

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 POA_ABC implements virtual TSCObject
{
    public abstract ... 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 abstract ... 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、セッ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 のサーバントクラスである POA_ABC も継承します。また、ABC_TSCspxy の派生クラスも同様に、TPBroker のサーバントクラスを継承します。

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 POA_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 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 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せます。

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せます。

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	できません。
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。tscstartpre 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、tscstartpre 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せます。

```
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せます。

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と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せます。

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当てられている TSCThread を取得できます。

TSCObject の呼び出し時の TSC アクセプタ名称の取得

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 `_TSCAcceptorName` を呼び出すことによって、サービス提供メソッドの呼び出し元である TSCAcceptor の TSC アクセプタ名称を取得できます。

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();

    //TSCアクセプタ名称
    public String _TSCAcceptorName();
};
```

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一内容の 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。

```
public String _TSCAcceptorName()
```

項目	型・意味
戻り値	TSC アクセプタ名称

TSCObject を保持している TSCAcceptor の TSC アクセプタ名称を取得します。

戻り値の 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
_TSCAcceptorName	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entrant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entrantなコードを記述する必要があります。

なお、コネクション切断などの障害時には、ユーザオブジェクトを生成したスレッドとは別のスレッドで destroy() を発行することがあります。

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 を呼び出せます。

"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 を呼び出せます。

"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 _TSCPriorty()
```

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 _TSCPriorty(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。

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();
};

```

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せます。

```

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せます。

```
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	
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せません。

```
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。

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()	×

(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
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
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せます。

```
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せます。

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
<code>getTSCDomainName</code>	できます。
<code>getTSCID</code>	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
<code>getTSCDomainName</code>	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、セッ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() メソッドでセッションを確立します。

セッ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ッ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ッ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

セッションを開始します。

セッ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、セッ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

セッションを解放します。

セッ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	セッション呼び出しインターバル監視時間(秒)

セッ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	セッ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	

セッ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。

- セッション呼び出し中の場合
セッ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
- セッ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

セッション呼び出し機能使用中のメソッド呼び出し規則

セッ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	セッ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
TRANSACTION_REQUIRED	TSCTransactionRequiredException
TRANSACTION_ROLLEDBACK	TSCTransactionRolledbackException
INVALID_TRANSACTION	TSCInvalidTransactionException
INV_POLICY	TSCInvPolicyException
CODESET_INCOMPATIBLE	TSCCodesetIncompatibleException
REBIND	TSCRebindException
TIMEOUT	TSCTimeoutException

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 ~ 40000

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;
public static final int TRANSACTION_REQUIRED = 27;
public static final int TRANSACTION_ROLLEDBACK = 28;
public static final int INVALID_TRANSACTION = 29;
public static final int INV_POLICY = 30;
public static final int CODESET_INCOMPATIBLE = 31;
public static final int REBIND = 32;
public static final int TIMEOUT = 33;

//
//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 INVALID_FLAG = 1022;
public static final int INVALID_WATCH_TIME = 1027;
public static final int INVALID_RETRY_REQUIREMENT = 1029;
public static final int INVALID_SESSION_INTERVAL = 1038;

//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;
public static final int RECEIVE_CONN_DOWN = 3023;

//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_REGISTERED = 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 SYSTEM_TIME_FAILURE = 5010;
public static final int CONN_INFO_FAILURE = 5011;
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;

//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;

```

TSCSystemException (Java)

```
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;
public static final int ALREADY_DISCONN = 12019;

//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;
public static final int TPBROKER_TRANSACTION_REQUIRED = 27998;
public static final int TPBROKER_TRANSACTION_ROLLEDBACK = 28998;
public static final int TPBROKER_INVALID_TRANSACTION = 29998;
public static final int TPBROKER_INV_POLICY = 30998;
public static final int TPBROKER_CODESET_INCOMPATIBLE = 31998;
public static final int TPBROKER_REBIND = 32998;
public static final int TPBROKER_TIMEOUT = 33998;

//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。

メソッド	複数のスレッド上からの同時呼び出し
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。
TSCCodesetIncompatibleException	コードセット間の通信に失敗しました。
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。
TSCInvalidTransactionException	トランザクションコンテキストが不正です。
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。
TSCInvPolicyException	呼び出しに使用したポリシーが不正です。
TSCMarshalException	パラメタ、および結果のマーシャル処理中に障害が発生しました。
TSCNoImplementException	オペレーションが実装できません。

例外名	説明
TSCNoMemoryException	動的メモリ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てしました。
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	リクエストへの応答はまだできません。
TSCObjAdapterException	オブジェクトアダプタが障害を検出しました。
TSCObjectNotExistException	オブジェクトが使用可能状態ではありません。
TSCPersistStoreException	パーシステントストレージ障害が発生しました。
TSCRebindException	設定された QoS ポリシーと IOR が矛盾しています。
TSCTimeoutException	リクエストでタイムアウトが発生しました。
TSCTransactionRequiredException	null 以外のトランザクションコンテキストが必要です。
TSCTransactionRolledbackException	トランザクションがロールバックしました。
TSCTransientException	トランジェント障害が発生しました。
TSCUnknownException	java.lang.Exception 型が発生した場合、Java 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 ~ 40000

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
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 TSCCodesetIncompatibleException extends
TSCSystemException
{
    public TSCCodesetIncompatibleException(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 TSCInvalidTransactionException extends
TSCSystemException
{
    public TSCInvalidTransactionException(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 TSCInvPolicyException extends TSCSystemException
{
    public TSCInvPolicyException(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);  
};
```

```

        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 TSCRebindException extends TSCSystemException
{
    public TSCRebindException(int _detail_code,
        int _place_code,
        int _completion_status,
        int _maintenance_code1,
        int _maintenance_code2,
        int _maintenance_code3,
        int _maintenance_code4);
};

```

```

public class TSCTimeoutException extends TSCSystemException
{
    public TSCTimeoutException(int _detail_code,
                               int _place_code,
                               int _completion_status,
                               int _maintenance_code1,
                               int _maintenance_code2,
                               int _maintenance_code3,
                               int _maintenance_code4);
};

public class TSCTransactionRequiredException extends
TSCSystemException
{
    public TSCTransactionRequiredException(int _detail_code,
                                           int _place_code,
                                           int _completion_status,
                                           int _maintenance_code1,
                                           int _maintenance_code2,
                                           int _maintenance_code3,
                                           int _maintenance_code4);
};

public class TSCTransactionRolledbackException extends
TSCSystemException
{
    public TSCTransactionRolledbackException(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。スレッドに割り当て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

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。

なお、コネクション切断などの障害時には、ユーザオブジェクトを生成したスレッドとは別のスレッドで destroy() を発行することがあります。

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

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 6-1 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

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] [-TSCno_proxy]
  [-TSCno_skel] [-TSCidl2cppfix フラグ]
  [-type_code_info] [-namespace] [-TSCcompat]
  [[-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け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け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け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》

セッション用 TSC ユーザプロキシ部分を出力するファイル名称に付け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け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け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け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け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

セッション用 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。

-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 > 《0011》

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 0011 が指定された場合、該当する領域を解放します。

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 "0" 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、領域を TSC ユーザスケルトンで解放しません。

上位 4 けた目

上位 4 けた目に "0" 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を TSC ユーザプロキシで解放しません。

両方とも解放しないようにする場合は、"0000" を指定してください。

-type_code_info

IDL に定義されたデータ型に対する TypeCode 情報を、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生成します。生成された TypeCode 情報を利用すると、CORBA::Any 型を容易に扱えます。

-idl2cpp オプションを指定しない場合は、-type_code_info オプションは無視されます。

-namespace

module を namespace として実装します。namespace として出力される module 名称は、class として出力された名称のように拡張されないで、IDL に定義した名称のままになります。このコマンドに -idl2cpp オプションを指定しないで idl2cpp コマンドを実

行する場合は、idl2cpp コマンドにも `-namespace` オプションを指定してください。
`-namespace` オプションを指定しない場合は、コンパイルエラーが発生するおそれがあります。

`-TSCcompat`

out 属性の引数として T*& 型を使用します。TPBroker OTM の Version 3 以前のバージョンで `-template` オプションを使用して作成した雛形クラスを、そのまま使用できるようにする互換オプションです。

`-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 <code>KFOT70008-I</code> が出力された場合、および <code>-h</code> 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。 <code>-idl2cpp</code> 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。

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ことがあります。

`-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ける文字列、サーバ部分のファイルに付ける文字列、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。

`tscidl2cpp` コマンドを実行すると、TPBroker が作成した "IDL 名称 .rollback" というファイルが、ホームディレクトリ下に残る場合があります。内容が異なる同名の IDL ファイルを編集した場合に影響が生じるおそれがあるので、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、`tscidl2cpp` 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、`tscidl2cpp 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。

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] 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。

-template

雛形クラスの Java ソースを出力します。

-TSCspxy

セッション用 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。

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 <code>-h</code> 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。 <code>-idl2java</code> オプションを指定した場合は、 <code>idl2java</code> 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。

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ことがあります。

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。

tscidl2j コマンドを実行すると、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に TPBroker が作成した "IDL 名称 .rollback" というファイルが残る場合があります。内容が異なる同名の IDL ファイルを編集した場合に影響が生じるおそれがあるので、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 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。

付録

付録A エラーコード一覧

付録B 場所コード一覧

付録C 完了状態一覧

付録D 内容コード一覧

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。

エラーコード一覧を次の表に示します。

表 A-1 エラーコード一覧

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てしました。
5	INTERNAL	ORB 内部エラーが発生しました。
6	MARSHAL	パラメタ、および結果のマーシャル処理中に障害が発生しました。
7	INITIALIZE	ORB 初期化障害が発生しました。
8	NO_IMPLEMENT	オペレーションが実装できません。
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	リクエストへの応答はまだできません。
12	BAD_INV_ORDER	ルーチン呼び出しの順番が不正です。
13	TRANSIENT	トランジェント障害が発生しました。
14	OBJECT_NOT_EXIST	オブジェクトが使用可能状態ではありません。
15	UNKNOWN	java.lang.Exception 型が発生した場合、Java 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。
25	OBJ_ADAPTER	オブジェクトアダプタが障害を検出しました。

値	エラーコード	障害内容
26	DATA_CONVERSION	データ変換に失敗しました。
27	TRANSACTION_REQUIRED	null 以外のトランザクションコンテキストが必要です。
28	TRANSACTION_ROLLEDBACK	トランザクションがロールバックしました。
29	INVALID_TRANSACTION	トランザクションコンテキストが不正です。
30	INV_POLICY	呼び出しに使用したポリシーが不正です。
31	CODESET_INCOMPATIBLE	コードセット間の通信に失敗しました。
32	REBIND	設定された QoS ポリシーと IOR が矛盾しています。
33	TIMEOUT	リクエストでタイムアウトが発生しました。

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。

場所コード一覧を次の表に示します。

表 B-1 場所コード一覧

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

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。

完了状態一覧を次の表に示します。

表 C-1 完了状態一覧

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。

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おそれがあります。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。

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。

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應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]
引数に指定したセッ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、規定のセッ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。

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おそれがあります。

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。

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，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おそれがあります。

3023 RECEIVE_CONN_DOWN

障害内容

TSC デーモンからクライアントアプリケーションへの接続、または TSC デーモンから TSC レギュレータへの接続が切断されていることを検知しました。

開発時の対策

[TSC デーモンへの接続からのリトライ]

再度、TSCAdm::getTSCClient の処理からやり直し、コネクションを再接続してください。

[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。

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]

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。

4010 ACPT_NOT_REGISTERED

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はない 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 /tscj2cl.jar"
- Windows を使用する場合
" 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 libtscblsv.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::activate 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

セッ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けません。

[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けません。

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]

セッションが確立されていません。またはセッション情報が削除されました。

原因を次に示します。

- ・セッ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。
- ・セッション呼び出しインターバル監視時間を経過しました。

- ・セッ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、またはセッション呼び出しインターバル監視時間が過ぎたため、セッションが解放された可能性があります。再度、セッション呼び出すには、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]
セッション呼び出し中です。

開発時の対策

[セッション呼び出しの確認]
セッ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ていました。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。

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。

5011 CONN_INFO_FAILURE

障害内容

コネクション識別子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、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。

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

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)]

共用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どう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。tscstartpre 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` に指定したセッ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、規定のセッ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。

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 ドメインマネー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 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平行カウント (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平行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。セッ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。

12019 ALREADY_DISCONN

障害内容

[オブジェクトのユーザメソッド]

切断済みのコネクション，または削除中の TSCClient に対してリクエストを送信しようとした。

開発時の対策

[TSC デーモンへの接続からのリトライ]

再度，TSCAdm::getTSCClient の処理からやり直し，コネクションを再接続してください。

[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。

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おそれがあります。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下さい。

[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下さい。

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下さい。

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下さい。

[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下さい。

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下さい。

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おそれがあります。

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おそれがあります。

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。

27998 TPBROKER_TRANSACTION_REQUIRED

障害内容

[CORBA メソッド]

CORBA::TRANSACTION_REQUIRED の例外通知を受けました。

28998 TPBROKER_TRANSACTION_ROLLEDBACK

障害内容

[CORBA メソッド]

CORBA::TRANSACTION_ROLLEDBACK の例外通知を受けました。

29998 TPBROKER_INVALID_TRANSACTION

障害内容

[CORBA メソッド]

CORBA::INVALID_TRANSACTION の例外通知を受けました。

30998 TPBROKER_INV_POLICY

障害内容

[CORBA メソッド]

CORBA::INV_POLICY の例外通知を受けました。

31998 TPBROKER_CODESET_INCOMPATIBLE

障害内容

[CORBA メソッド]

CORBA::CODESET_INCOMPATIBLE の例外通知を受けました。

32998 TPBROKER_REBIND

障害内容

[CORBA メソッド]

CORBA::REBIND の例外通知を受けました。

33998 TPBROKER_TIMEOUT

障害内容

[CORBA メソッド]

CORBA::TIMEOUT の例外通知を受けました。

索引

記号

`_TSCAcceptorName`
135, 142, 161, 308, 325
`_TSCAcceptorName0` 303
`_TSCContext` 134, 144, 302, 310
`_TSCInterfaceName`
134, 142, 161, 302, 308, 325
`_TSCPRIORITY` 162, 309, 326
`_TSCSessionInterval` 160, 325
`_TSCStart` 159, 324
`_TSCStop` 160, 324
`_TSCThread` 135, 303
`_TSCTimeout` 142, 161, 308, 326

A

`ABC_TSCacpt (C++)` 102
`ABC_TSCacpt (Java)` 271
`ABC_TSCfactimpl (C++)` 104
`ABC_TSCfactimpl (Java)` 273
`ABC_TSCimpl (C++)` 105
`ABC_TSCimpl (Java)` 274
`ABC_TSCprxy (C++)` 106
`ABC_TSCprxy (Java)` 275
`ABC_TSCsk (C++)` 108
`ABC_TSCsk (Java)` 277
`ABC_TSCspxy (C++)` 110
`ABC_TSCspxy (Java)` 278
`ACPT_IS_NULL` 373
`ACPT_NOT_REGISTERED` 387
`activate` 151, 317
`ACTIVATE_IN_END` 385
`ACTIVATE_IN_START` 390
`ACTIVATE_WITH_DIFF_PROP` 391
`ALL_CONN_FAILURE` 424
`ALREADY_ACTIVE` 421
`ALREADY_DEACTIVE` 421
`ALREADY_DISCONNECT` 423
`ALREADY_INITCLNT` 422
`ALREADY_INITSERV` 422

`ALREADY_SERV_ML` 423
`ALREADY_SESSION_START` 423
`ALREADY_SHUTDOWN` 412

B

`BAD_CONTEXT` 166, 331
`BAD_INV_ORDER` 165, 330
`BAD_OPERATION` 165, 330
`BAD_PARAM` 165, 330
`BAD_TYPECODE` 166, 331
`BASIC_CONN_FAILURE` 380

C

`CALL_IN_END` 386
`CALL_IN_HOLD` 384
`cancelAcceptor` 150, 316
`CLNT_COMMAND_START` 393
`CLNT_CONN_IN_END` 386
`CLNT_CONN_IN_START` 388
`CLNT_DISCONNECT_IN_END` 390
`CLNT_INIT_IN_END` 391
`CLNT_INIT_IN_START` 392
`CLNT_IS_NULL` 375
`CLNT_NOT_INITIALIZED` 422
`CODESET_INCOMPATIBLE` 166, 331
`COMM_FAILURE` 165, 330
`COMMON_EXCEPTION` 427
`COMPLETED_MAYBE`
167, 177, 332, 342, 371
`COMPLETED_NO`
167, 177, 332, 342, 371
`COMPLETED_YES`
167, 177, 332, 342, 371
`CONN_FAILURE` 381
`CONN_INFO_FAILURE` 398
`CREATED_OBJECT_IS_NULL` 429
`CREATED_THREAD_OBJECT_IS_NULL`
429

D

DATA_CONVERSION 166, 331
 deactivate 152, 318
 DEACTIVATE_FAILURE 383
 DEACTIVATE_IN_END 390
 DIFF_THREAD_CALL 386
 DOMAIN_IS_NULL 375

E

endClient 120, 289
 endServer 119, 288
 ENTRY_FAILURE 413
 EVENT_FAILURE 396

F

FACTORY_CREATE_FAILURE 428
 FACTORY_DESTROY_FAILURE 428
 FILE_ACCESS_FAILURE 394
 FREE_MEM 166, 331

G

get_status 123, 291
 getAcceptorName 115, 283
 getCompletionStatus 173, 338
 getDetailCode 172, 337
 getErrorCode 172, 337
 getInterfaceName 115, 283
 getMaintenanceCode1 173, 338
 getMaintenanceCode2 173, 338
 getMaintenanceCode3 173, 338
 getMaintenanceCode4 173, 338
 getParallelCount 151, 316
 getPlaceCode 173, 337
 getTSCClient 120, 121, 289, 290
 getTSCDomainName 126, 156, 321
 getTSCID 126, 156, 321
 getTSCServer 122, 290
 getUserData 129, 298
 getUserDataLength 129
 getUserDataRelease 129

I

IDL コンパイラが生成するクラス
 11, 29, 73, 197, 212, 246
 IDL 定義一覧 2
 IDL データ型 4
 IDL 文法の注意事項 6
 IMP_LIMIT 166, 331
 INCOMPATIBLE_PROTOCOL 383
 initClient 118, 286
 INITIALIZE 165, 330
 initServer 117, 285
 INTERNAL 165, 330
 INTF_REPOS 166, 331
 INV_FLAG 166, 331
 INV_IDENT 166, 331
 INV_OBJREF 165, 330
 INV_POLICY 166, 331
 INVALID_ACPT_NAME 372
 INVALID_ACPT_REGID 373
 INVALID_DEF_ACPT 407
 INVALID_DEF_APIID 409
 INVALID_DEF_CLIENT_MESSAGE_BUFFER_COUNT 408
 INVALID_DEF_DOMAIN_NAME 406
 INVALID_DEF_EXCEPT_CONVERT_FILE 413
 INVALID_DEF_MY_HOST 410
 INVALID_DEF_NICE 407
 INVALID_DEF_PARALLEL_COUNT 405
 INVALID_DEF_PRIORITY 407
 INVALID_DEF_REBIND_INTERVAL 410
 INVALID_DEF_REBIND_TIMES 410
 INVALID_DEF_REQUEST_WAY 408
 INVALID_DEF_RETRY_REFERENCE 411
 INVALID_DEF_RETRY_WAY 412
 INVALID_DEF_RT_ACPT 404
 INVALID_DEF_SESSION_INTERVAL 413
 INVALID_DEF_TIMEOUT 404
 INVALID_DEF_TSCID 405
 INVALID_DEF_WATCH_METHOD 409
 INVALID_DEF_WATCH_TIME 409
 INVALID_DEF_WITH_SYSTEM 406
 INVALID_DOMAIN_NAME 374

INVALID_ENV_TSCDIR 406
 INVALID_FILE_FORMAT 411
 INVALID_FLAG 376
 INVALID_OP_PARAM 374
 INVALID_PARALLEL_COUNT 372
 INVALID_PRC_KIND 408
 INVALID_PRIORITY 376
 INVALID_REQUEST_WAY 376
 INVALID_RETRY_REQUIREMENT 377
 INVALID_RT_ACPT_NAME 372
 INVALID_SESSION_INTERVAL 377
 INVALID_STREAM_LEN 399
 INVALID_STREAM_VALUE 399
 INVALID_TIMEOUT 372
 INVALID_TRANSACTION 166, 331
 INVALID_TSCID 373
 INVALID_USER_EXCEPTION 427
 INVALID_WATCH_TIME 377

L

LOAD_SHLIB_FAILURE 405

M

MARSHAL 165, 330
 MARSHAL_ERROR 403
 MARSHAL_OTHERS 400
 MEM_ALLOC_FAILURE 378
 MSG_TYPE_FAILUR 396
 MTRACE_FAILURE 407
 MUTEX_FAILURE 396

N

NO_IMPLEMENT 165, 330
 NO_MEMORY 165, 330
 NO_PERMISSION 165, 330
 NO_RESOURCES 165, 330
 NO_RESPONSE 165, 330
 NO_SUCH_ACPT 414
 NO_SUCH_INTERF 414
 NO_SUCH_OP_NAME 415
 NOT_ACCEPT_OBJECT 392
 NOT_IGNORE_PROTOCOL 383

NOT_SUPPORTED 388

O

OBJ_ADAPTER 166, 331
 OBJ_FACT_IS_NULL 373
 OBJECT_NOT_EXIST 165, 330
 ORB_IS_NULL 375
 OVER_ACPT_REGI 388
 OVER_ADM_MAX_CLNT 417
 OVER_ADM_MAX_SERV 417
 OVER_MAX_CLNT 416
 OVER_MAX_DISPATCH_PARALLEL 419
 OVER_MAX_ORB_CLIENT 420
 OVER_MAX_REQUEST_COUNT 419
 OVER_MAX_RT_ACPT_REGI 418
 OVER_MAX_SERV 417
 OVER_MAX_THIN_CLIENT 418

P

PERSIST_STORE 166, 331
 PLACE_CODE_CLNT
 167, 177, 332, 342, 370
 PLACE_CODE_CLNT_REG
 167, 177, 332, 342, 370
 PLACE_CODE_DAEMON
 167, 177, 332, 342, 370
 PLACE_CODE_ORBGW
 167, 177, 332, 342, 370
 PLACE_CODE_SERV
 167, 177, 332, 342, 370
 PLACE_CODE_SKELTON
 167, 177, 332, 342, 370
 PLACE_CODE_STUB
 167, 177, 332, 342, 370
 PLACE_CODE_USER_AP
 167, 177, 332, 342, 370
 PROGRAM_ERROR 398
 PROPERTIES_FAILURE 395

R

REBIND 166, 331
 REBIND_FAILURE 424

RECEIVE_CONN_DOWN 384
 registerAcceptor 150, 316
 releaseTSCClient 122, 291
 releaseTSCServer 122, 291
 REP_MARSHAL_FAILURE 401
 REP_UNMARSHAL_FAILURE 402
 REQ_MARSHAL_FAILURE 400
 REQ_UNMARSHAL_FAILURE 401
 reset 191, 353
 RT_ACPT_IS_ACTIVE 385

S

SAME_APID_EXIST 393
 SEND_CLNT_FAILURE 378
 SEND_SERV_FAILURE 379
 SEND_THIN_CLNT_FAILURE 379
 SEND_TSCD_FAILURE 380
 SERV_CONN_IN_END 385
 SERV_CONN_IN_START 387
 SERV_INIT_IN_END 391
 SERV_INIT_IN_START 392
 SERV_IS_NULL 375
 SERV_NO_SUCH_ACPT 426
 SERV_NO_SUCH_INTERF 425
 SERV_NOT_INITIALIZED 422
 serverMainloop 119, 288
 SESSION_IN_CALL 395
 SESSION_IN_END 394
 SESSION_NOT_START 423
 setParallelCount 151, 316
 setUserData 128, 297
 SH_MEM_FAILURE 397
 shutdown 119
 SIG_COND_FAILURE 396
 start 191, 353
 stop 191, 353
 SYSTEM_TIME_FAILURE 398

T

THREAD_CREATE_FAILURE 397
 THREAD_FACT_IS_NULL 376

THREAD_FACTORY_CREATE_FAILURE
 428
 THREAD_FACTORY_DESTROY_FAILURE
 E 428
 TIMED_OUT 421
 TIMEOUT 166, 331
 TPBROKER_BAD_CONTEXT 431
 TPBROKER_BAD_INV_ORDER 424
 TPBROKER_BAD_OPERATION 416
 TPBROKER_BAD_PARAM 377
 TPBROKER_BAD_TYPECODE 430
 TPBROKER_CODESET_INCOMPATIBLE
 432
 TPBROKER_COMM_FAILURE 384
 TPBROKER_DATA_CONVERSION 431
 TPBROKER_FREE_MEM 430
 TPBROKER_IMP_LIMIT 430
 TPBROKER_INITIALIZE 414
 TPBROKER_INTERNAL 398
 TPBROKER_INTF_REPOS 431
 TPBROKER_INV_FLAG 430
 TPBROKER_INV_IDENT 430
 TPBROKER_INV_OBJREF 429
 TPBROKER_INV_POLICY 432
 TPBROKER_INVALID_TRANSACTION
 432
 TPBROKER_MARSHAL 404
 TPBROKER_NO_IMPLEMENT 415
 TPBROKER_NO_MEMORY 378
 TPBROKER_NO_PERMISSION 395
 TPBROKER_NO_RESOURCES 420
 TPBROKER_NO_RESPONSE 421
 TPBROKER_OBJ_ADAPTER 431
 TPBROKER_OBJECT_NOT_EXIST 426
 TPBROKER_PERSIST_STORE 430
 TPBROKER_REBIND 432
 TPBROKER_TIMEOUT 432
 TPBROKER_TRANSACTION_REQUIRED
 431
 TPBROKER_TRANSACTION_ROLLEDBA
 CK 432
 TPBROKER_TRANSIENT 425
 TPBROKER_UNKNOWN 427

- TPBroker クラス 9, 195
- TPBroker スケルトン 10, 196
- TPBroker スタブ 9, 195
- TRANSACTION_REQUIRED 166, 331
- TRANSACTION_ROLLEDBACK 166, 331
- TRANSIENT 165, 330
- TSCAcceptor (C++) 112
- TSCAcceptor (Java) 280
- TSCAcceptor の生成と削除 115
- TSCAcceptor または派生クラスの生成 283
- TSCAdm (C++) 117
- TSCAdm (Java) 285
- TSCAdm.Active 292
- TSCAdm.Dead 292
- TSCAdm.Dying 292
- TSCAdm.Living 292
- TSCAdm::TSC_ADM_PRC_ACTIVE 123
- TSCAdm::TSC_ADM_PRC_DEAD 123
- TSCAdm::TSC_ADM_PRC_DYING 123
- TSCAdm::TSC_ADM_PRC_LIVING 123
- TSCBadContextException 175, 340
- TSCBadInvOrderException 175, 340
- TSCBadOperationException 175, 340
- TSCBadParamException 175, 340
- TSCBadTypecodeException 175, 340
- TSCClient (C++) 125
- TSCClient (Java) 294
- TSCClient の取得 125, 294
- TSCClient の取得と解放 126, 295
- TSCClient のリファレンスを取得
120, 121, 289, 290
- TSCCodesetIncompatibleException
175, 340
- TSCCommFailureException 175, 340
- TSCContext (C++) 128
- TSCContext (Java) 297
- TSCContext の取得 133, 301
- TSCContext の生成 298
- TSCContext の生成と削除 129
- TSCContext を利用するアプリケーションブ
ログラム (C++) 53
- TSCContext を利用するアプリケーションブ
ログラム (Java) 231
- TSCContext を利用するクライアントアプリ
ケーションの例 (C++) 53
- TSCContext を利用するクライアントアプリ
ケーションの例 (Java) 231
- TSCContext を利用するサーバアプリケー
ションの例 (C++) 58
- TSCContext を利用するサーバアプリケー
ションの例 (Java) 235
- TSCD_IS_NOT_MY_HOST 388
- TSCDataConversionException 175, 340
- TSCDomain (C++) 131
- TSCDomain (Java) 299
- TSCFreeMemException 175, 340
- tsclid2cpp 357
- tsclid2j 364
- TSCImpLimitException 175, 340
- TSCInitializeException 175, 340
- TSCInternalException 175, 340
- TSCIntfReposException 175, 340
- TSCInvalidTransactionException 175, 340
- TSCInvFlagException 175, 340
- TSCInvIdentException 175, 340
- TSCInvObjrefException 175, 340
- TSCInvPolicyException 175, 340
- TSCMarshalException 175, 340
- TSCNoImplementException 175, 340
- TSCNoMemoryException 176, 341
- TSCNoPermissionException 176, 341
- TSCNoResourcesException 176, 341
- TSCNoResponseException 176, 341
- TSCObjAdapterException 176, 341
- TSCObject (C++) 133
- TSCObject (Java) 301
- TSCObjectFactory (C++) 137
- TSCObjectFactory (Java) 304
- TSCObjectFactory の派生クラスの生成と削
除 138
- TSCObjectFactory または派生クラスの生成
305
- TSCObjectNotExistException 176, 341
- TSCObject の管理 112, 137, 280, 304
- TSCObject の派生クラスのインスタンスの生
成 303

- TSCObject の派生クラスの生成と削除 135
- TSCPersistStoreException 176, 341
- TSCPRIORITY 143
- TSCProxyObject (C++) 140
- TSCProxyObject (Java) 306
- TSCProxyObject または派生クラスの生成 310
- TSCRebindException 176, 341
- TSCRootAcceptor (C++) 146
- TSCRootAcceptor (Java) 312
- TSCRootAcceptor の生成 318
- TSCRootAcceptor の生成と削除 152
- TSCServer (C++) 155
- TSCServer (Java) 320
- TSCServer と接続 155, 320
- TSCServer の取得と解放 156, 321
- TSCSessionProxy (C++) 158
- TSCSessionProxy (Java) 323
- TSCSystemException (C++) 165
- TSCSystemException (Java) 330
- TSCSystemException の派生クラス (C++) 175
- TSCSystemException の派生クラス (Java) 340
- TSCThread (C++) 186
- TSCThread (Java) 349
- TSCThreadFactory (C++) 187
- TSCThreadFactory (Java) 350
- TSCThreadFactory の派生クラスの生成 351
- TSCThreadFactory の派生クラスの生成と削除 188
- TSCThread の管理 187, 350
- TSCThread の取得 133, 301
- TSCThread を利用するアプリケーションプログラム (C++) 61
- TSCThread を利用するアプリケーションプログラム (Java) 237
- TSCThread を利用するクライアントアプリケーションの例 (C++) 61
- TSCThread を利用するクライアントアプリケーションの例 (Java) 237
- TSCThread を利用するサーバアプリケーションの例 (C++) 61
- TSCThread を利用するサーバアプリケーションの例 (Java) 237
- TSCTimeoutException 176, 341
- TSCTransactionRequiredException 176, 341
- TSCTransactionRolledbackException 176, 341
- TSCTransientException 176, 341
- TSCUnknownException 176, 341
- TSCWatchTime (C++) 190
- TSCWatchTime (Java) 352
- TSCWatchTime を利用するアプリケーションプログラム (C++) 89
- TSCWatchTime を利用するアプリケーションプログラム (Java) 259
- TSCWatchTime を利用するクライアントアプリケーションの例 (C++) 89
- TSCWatchTime を利用するクライアントアプリケーションの例 (Java) 259
- TSCWatchTime を利用するサーバアプリケーションの例 (C++) 89
- TSCWatchTime を利用するサーバアプリケーションの例 (Java) 259
- TSC アクセプタ名称 113, 140, 281, 306
- TSC アクセプタ名称の取得 134, 302
- TSC サービス識別子 113, 140, 281, 306
- TSC デモンに直結してリクエスト 120, 289
- TSC ユーザアクセプタ 9, 195
- TSC ユーザアクセプタの実装クラス 102, 271
- TSC ユーザオブジェクト 9, 195
- TSC ユーザオブジェクトの実装クラス 105, 274
- TSC ユーザオブジェクトファクトリ 9, 195
- TSC ユーザオブジェクトファクトリの実装クラス 104, 273
- TSC ユーザスケルトン 10, 196
- TSC ユーザスケルトンの実装クラス 108, 277
- TSC ユーザスレッドの管理 148, 314

TSC ユーザプロキシ 9, 195
 TSC ユーザプロキシの実装クラス
 106, 110, 275, 278
 TSC ルートアクセプタ状態 146, 312
 TSC ルートアクセプタ登録名称 147, 313
 TSC レギュレータを經由してリクエスト
 120, 289
 TSD_FAILURE 397

U

UEXCEPT_MARSHAL_FAILURE 402
 UEXCEPT_UNMARSHAL_FAILURE 403
 UNKNOWN 165, 330

W

WATCH_IS_STARTED 393
 WATCH_IS_STOPPED 393

あ

アプリケーションプログラムの作成手順
 (C++) 8
 アプリケーションプログラムの作成手順
 (Java) 194

い

インタフェース名称
 113, 133, 140, 281, 301, 306

え

エラーコード 165, 330
 エラーコード一覧 368

か

完了状態 167, 177, 332, 342
 完了状態一覧 371

き

基本データ型 (C++) 95
 基本データ型 (Java) 265

く

クライアントアプリケーションの終了
 120, 289
 クライアントアプリケーションの初期化
 118, 287
 クラス関連図 (C++) 97
 クラス関連図 (Java) 266
 クラスの一覧 (C++) 94
 クラスの一覧 (Java) 264

こ

公開メソッド呼び出し 100, 269
 公開メソッド呼び出しと内部参照 (C++)
 100
 公開メソッド呼び出しと内部参照 (Java)
 269
 コマンドの一覧 356

さ

サーバアプリケーションの終了 119, 288
 サーバアプリケーションの初期化 118, 286

し

システム提供クラス 9, 195
 受信待ち状態 119, 288

せ

セッション呼び出しをするアプリケーションプ
 ログラム (C++) 45
 セッション呼び出しをするアプリケーションプ
 ログラム (Java) 225
 セッ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++) 45
 セッ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) 225
 接続経路 120, 289

と

同期型呼び出しをするアプリケーションプロ
 グラム (C++) 11

同期型呼び出しをするアプリケーションプログラム (Java) 197
 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++) 27
 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) 209
 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++) 12
 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) 198
 同期型呼び出しをするサーバアプリケーションの例 (C++) 17
 同期型呼び出しをするサーバアプリケーションの例 (Java) 202
 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
 11, 29, 45, 73, 197, 212, 225, 246
 トランザクションフレームの出力 (C++) 357
 トランザクションフレームの出力 (Java) 364

な

内部参照 (アクセス) 100, 269
 内容コード 166, 176, 331, 341
 内容コード一覧 372

は

場所コード 167, 176, 332, 341
 場所コード一覧 370
 パラレルカウント 146, 312

ひ

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++) 29
 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) 212
 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++) 30
 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) 213

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++) 35
 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) 217
 雛形クラス 8, 194

ふ

プロセスステータス 123, 291

ゆ

ユーザ定義 IDL インタフェース 2
 ユーザ定義 IDL インタフェース依存クラス 9, 195
 ユーザ定義 IDL インタフェースの例 11, 29, 73, 197, 212, 246
 ユーザデータの取得 128, 297
 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++) 73
 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) 246
 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++) 74
 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) 247
 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++) 79
 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) 251

れ

例外クラス 175, 340

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。

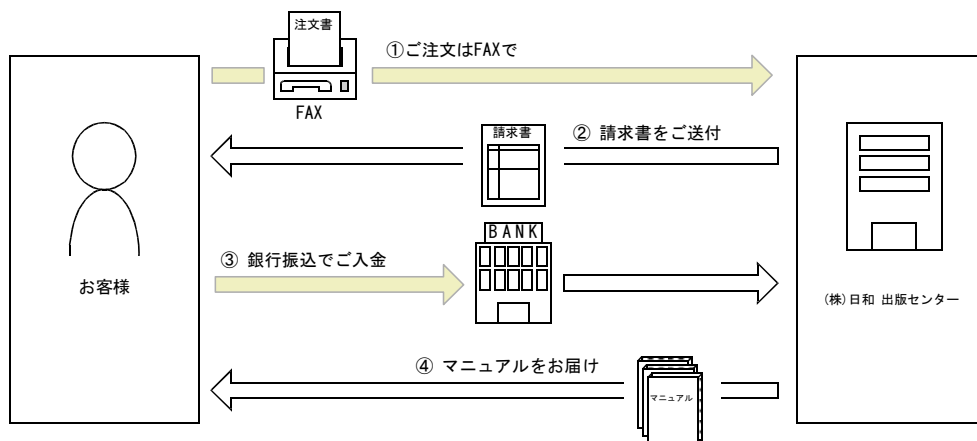
2. インターネットからのマニュアル参照(ソフトウェアサポートサービス)

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。