HITACHI
Inspire the Next

OpenTP1 Version 7
分散トランザクション処理機能

TP1/Connector for .NET Framework 使用の手引

解説・手引・文法・操作書

3000-3-D69-40

前書き

■ 対象製品

R-15451-27 uCosminexus TP1/Connector for .NET Framework 07-50（適用 OS：Windows 7，Windows 8，Windows 8.1，Windows 10，Windows Server 2008 R2，Windows Server 2012，Windows Server 2012 R2，Windows Server 2016）

こ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「リリースノート」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、外国為替及び外国貿易法の規制並びに米国輸出管理規則など外国の輸出関連法規をご確認の上、必要な手続きをお取りください。

なお、不明な場合は、弊社担当営業にお問い合わせください。

■ 商標類

■ 発行

2018 年 10 月 3000-3-D69-40

■ 著作権

変更内容

変更内容（3000-3-D69-40） uCosminexus TP1/Connector for .NET Framework 07-50

追加・変更内容	変更個所
以下の適用 OS を追加した。 • Windows 7 • Windows 8 • Windows 8.1 • Windows 10 • Windows Server 2008 R2 • Windows Server 2012 • Windows Server 2012 R2 • Windows Server 2016	—
以下の適用 OS を削除した。 • Windows 2000 • Windows Server 2003 • Windows Server 2008 • Windows Vista • Windows XP	—
Visual Studio 2013 以降に対応した。 また，Visual Studio 2008 以前の記述を削除した。	—
製品の概要に関する記述を追加した。	1.
アプリケーションを開発する言語から J#および COBOL に関する記述を削除した。	1.1.1，2.6.1(3)(a)，4.8.2(1)，5.
Connector .NET を開発および使用する場合の前提ソフトウェアを変更した。	2.2.7(3)(a)，2.6.1(3)(a)，2.9.1(1)(a)，4.8.2(1)
Windows Vista および Windows Server 2008 の記述を削除した。	4.8.2(1)
IIS 7.0 以降の場合のサンプルプログラムの手順を追加した。	4.8.3(3)
次に示すバージョンの変更点を記載した。 • Connector .NET 07-50	付録 C.1

単なる誤字・脱字などはお断りなく訂正しました。

変更内容（3000-3-D69-30） uCosminexus TP1/Connector for .NET Framework 07-03

追加・変更内容
Windows Server 2008 に対応した。

追加・変更内容
.NET Framework v3.5 に対応した。
Visual Studio 2008 に対応した。 これに伴い，Visual Studio 2008 では，開発言語として J#を使用できない旨を記載した。
開発言語として，COBOL 言語に対応した。 これに伴い，次のクラスの説明に COBOL 言語の場合の説明を追記した。 • IndexedRecord • MessageBuffer • RpcInfo • TcpipConnection • TcpipInfo • TP1Connection • TP1ConnectionManager • TP1ConnectorError
バージョンアップ時の，API，定義，コマンドおよびデフォルト値の変更点を記載した。

変更内容（3000-3-D69-20） uCosminexus TP1/Connector for .NET Framework　07-02

追加・変更内容
MSDTC を利用するほかのリソースと OpenTP1 が使用するリソースとの間で分散トランザクション連携を実現する，MSDTC 連携機能を追加した。これに伴い，次のように変更した。 構成定義に次の要素を追加した。 • distributedTransaction 要素 • recoveryService 要素 構成定義の<log>要素の level 属性のログレベルに，次のメッセージ ID を追加した。 • KFCA32471-I • KFCA32472-I • KFCA32473-I • KFCA32474-I • KFCA32475-I • KFCA32476-I • KFCA32480-I • KFCA32495-I • KFCA32477-I • KFCA32478-I 次のコマンドを追加した。 • cnnnidgen • cnntrsls TP1ConnectorError クラスのフィールドに，DCCNNER_DISTRIBUTED_TRN を追加した。
リモート API 機能でスタティックコネクションスケジュールモードを使用する場合の注意事項を追加した。

追加・変更内容
WCF の統一化されたプログラミングモデルから OpenTP1 にサービスを要求できる，WCF 連携機能を追加した。これに伴い，次のクラスを追加した。 • TP1IntegrationBehavior • TP1IntegrationBinding • TP1RpcClient

変更内容（3000-3-D69-10） uCosminexus TP1/Connector for .NET Framework　07-01

追加・変更内容
あるコネクションが接続障害を検知したあとにほかのコネクションがその接続先に接続することを防止する，接続障害軽減機能を追加した。これに伴い，構成定義の<connection>要素に次の属性を追加した。 • failureInfoSharing 属性 • failureCheckInterval 属性 また，構成定義の<log>要素の level 属性のログレベル 2 に，次のメッセージ ID を追加した。 • KFCA32413-I • KFCA32414-I
Windows Vista を追加した。

はじめに

このマニュアルは，プログラムプロダクト R-15451-27 uCosminexus TP1/Connector for .NET Framework の機能と使い方について説明したものです。

本文中に記載されている製品のうち，このマニュアルの対象製品ではない製品については，OpenTP1 Version 7 対応製品の発行時期をご確認ください。

■ 対象読者

システム管理者，システム設計者，プログラマ，オペレータの方を対象としています。また，.NET Framework を利用したプログラミングの知識を持っていることを前提としています。

なお，このマニュアルの前提であるマニュアル「TP1/Client for .NET Framework 使用の手引」を必ずお読みください。

■ 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　概要

TP1/Connector for .NET Framework（Connector .NET）の概要について説明しています。

第 2 章　機能

TP1/Connector for .NET Framework の機能について説明しています。

第 3 章　構成定義

TP1/Connector for .NET Framework で使用する構成定義について説明しています。

第 4 章　UAP の作成と実行

TP1/Connector for .NET Framework で使用する UAP の作成方法とサンプルプログラムの使用方法について説明しています。

第 5 章　運用コマンド

TP1/Connector for .NET Framework で使用する運用コマンドについて説明しています。

第 6 章　クラスリファレンス

TP1/Connector for .NET Framework で利用するクラスについて説明しています。

第 7 章　障害運用

TP1/Connector for .NET Framework で障害が発生した場合の対処方法について説明しています。

付録 A　DCCM3 と接続する場合の注意事項

RPC やメッセージ送受信機能を使用して DCCM3 と接続する場合の注意事項について説明しています。

付録 B　Connector .NET で利用できるクラスのフィールド

TP1/Connector for .NET Framework で利用できるクラスについて説明しています。

付録 C　バージョンアップ時の変更点

各バージョンでの API，定義，およびコマンドの変更点について説明しています。

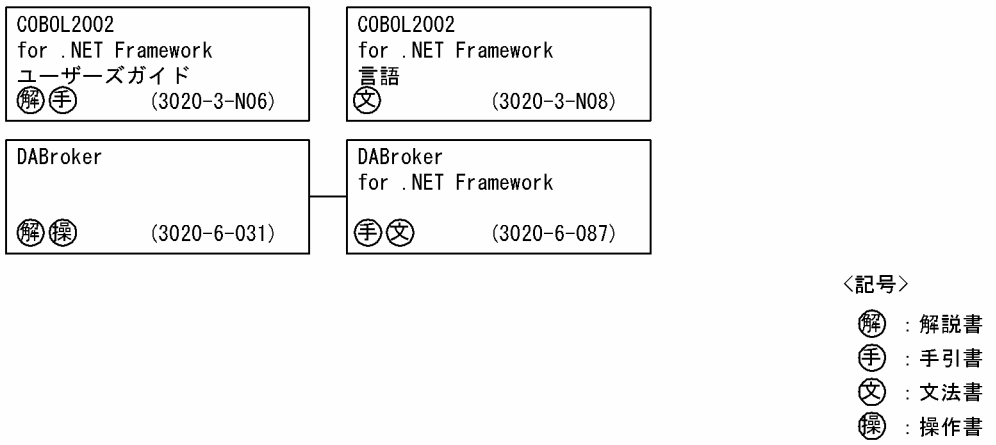
付録 D　用語解説

TP1/Connector for .NET Framework に関する用語について説明しています。

■ 関連マニュアル

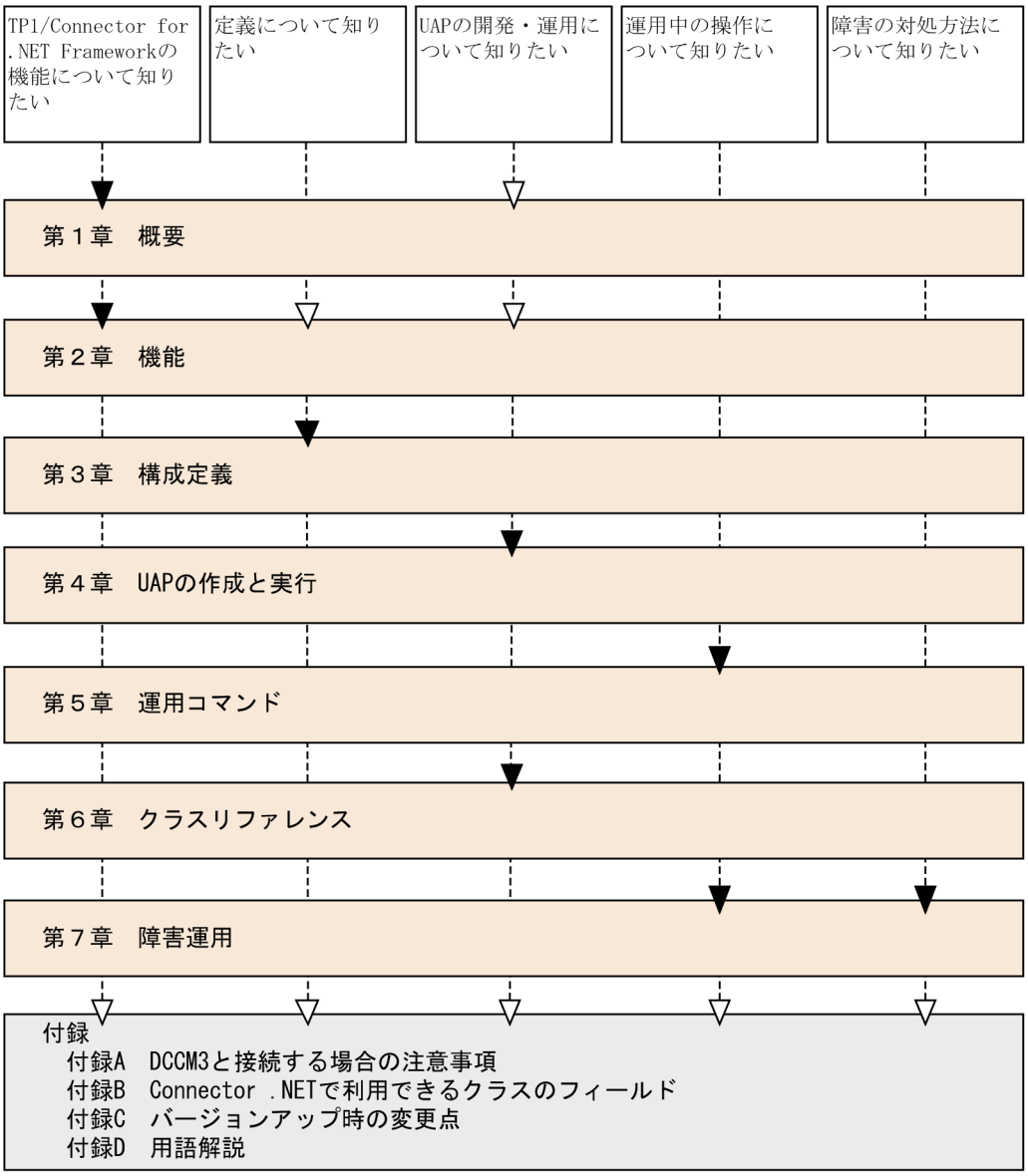
注※　このマニュアルおよびこのマニュアルが対象とする製品の発行時期についてはご確認ください。

●関連製品



■ 読書手順

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



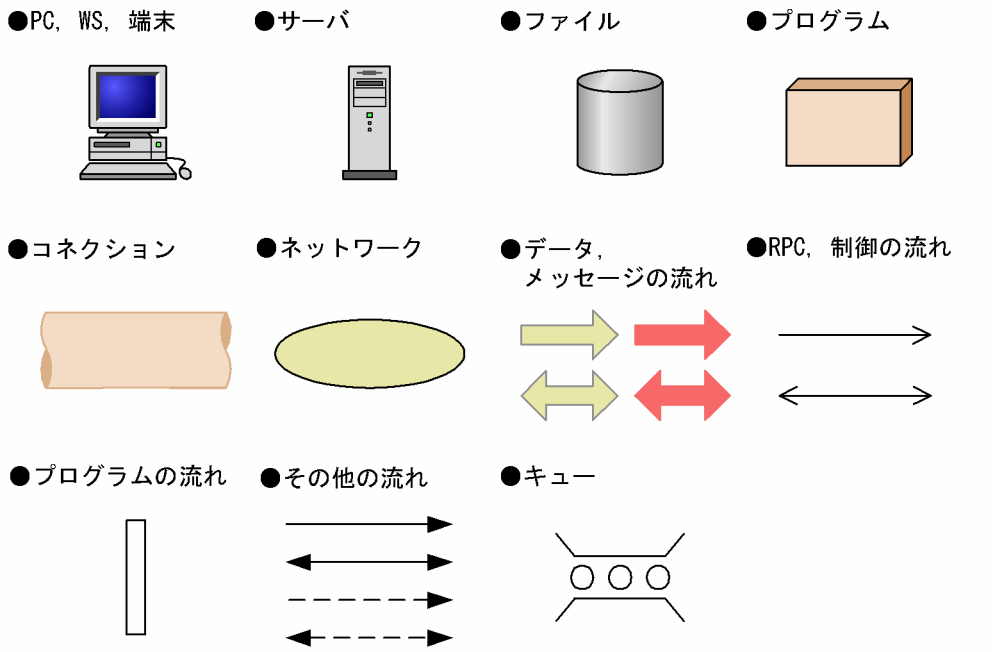
（凡例）

：　必ず読む項目　　：　必要に応じて読む項目

注　OpenTP1 for .NET Frameworkの概要については，
マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。

■ 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

(1) 文法記述記号

文法の記述形式について説明する記号です。

文法記述記号	意味
〔　　〕	この記号で囲まれている項目は省略できることを示します。 (例) spp2tsp 〔-n 名前空間名称〕 これは，「spp2tsp」と指定するか，または「spp2tsp -n 名前空間名称」と指定することを示します。 ただし，構成定義の説明では省略できる項目にこの記号は使用しません。
…	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) -R アセンブリ名称〔,アセンブリ名称〕… これは，-R オプションのアセンブリ名称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は選択できることを示します。 (例) watchTimeNotification="true | false" これは，watchTimeNotification 属性に true か false のどちらかを指定できることを示します。
{　　}	この記号で囲まれている複数の項目のうちから一つを選択できることを示します。 (例) -l {cs | vjs | vb} これは，cs，vjs，vb の三つのオプションのうち，どれか一つを指定することを示します。
____(下線)	この記号で示す項目は，該当オプション，要素，属性，またはコマンド引数を省略した場合の仮定値を示します。 (例) -E {big | little}

文法記述記号	意味
	これは，オプションの指定を省略した場合，big オプションを仮定することを示します。

(2) 属性表示記号

ユーザ指定値の範囲などを説明する記号です。

属性表示記号	意味
～	この記号のあとにユーザ指定値の属性を示します。
《　》	ユーザ指定値を省略した場合の仮定値を示します。
〈　〉	ユーザ指定値の構文要素記号を示します。
((　))	ユーザ指定値の指定範囲を示します。

(3) 構文要素記号

ユーザ指定値の内容を説明する記号です。

構文要素記号	意味
〈英字記号〉	アルファベット（A～Z，a～z）と#，@，¥
〈符号なし整数〉	数字（0～9）
〈識別子〉	先頭がアルファベット（A～Z，a～z）で始まる英数字列
〈記号名称〉	先頭が英字記号で始まる英数字記号列
〈文字列〉	任意の文字の並び
〈パス名〉	記号名称，円記号（¥），およびピリオド（.）の並び （ただし，パス名は使用している OS に依存します）
〈ファイル名〉	任意の文字の並び （ただし，ファイル名は使用している OS，およびアプリケーションに依存します）

■ このマニュアルでの表記

このマニュアルでは，製品の名称を省略して表記しています。製品の名称と，このマニュアルでの表記を次に示します。

製品名称	このマニュアルでの表記
Microsoft(R) .NET Framework	.NET Framework
Microsoft(R) Visual C#	C#
Microsoft(R) Visual Basic	Visual Basic

製品名称	このマニュアルでの表記	
Microsoft(R) Visual Basic .NET	VB.NET	
COBOL2002 for .NET Framework Developer	COBOL2002 for .NET Framework	
COBOL2002 for .NET Framework Server Suite		
COBOL2002 for .NET Framework Server Runtime		
uCosminexus TP1/Client for .NET Framework	TP1/Client for .NET Framework	Client .NET
uCosminexus TP1/Connector for .NET Framework	TP1/Connector for .NET Framework	Connector .NET
VOS1 DCCM3	DCCM3	
VOS3 XDM/DCCM3		
uCosminexus TP1/Extension for .NET Framework	TP1/Extension for .NET Framework	Extension .NET
Visual J#	J#	
MSDN(R)	MSDN	
uCosminexus TP1/NET/TCP/IP	TP1/NET/TCP/IP	
uCosminexus TP1/NET/TCP/IP(64)		
uCosminexus TP1/LiNK	TP1/LiNK	TP1/Server
uCosminexus TP1/Server Base	TP1/Server Base	
uCosminexus TP1/Server Base(64)		
Microsoft(R) Visual Studio(R) Community 2013	Visual Studio 2013	Visual Studio
Microsoft(R) Visual Studio(R) Professional 2013		
Microsoft(R) Visual Studio(R) Premium 2013		
Microsoft(R) Visual Studio(R) Ultimate 2013		
Microsoft(R) Visual Studio(R) Express 2013 for Windows Desktop	Visual Studio Express 2013 for Windows Desktop	
Microsoft(R) Visual Studio(R) Community 2015	Visual Studio 2015	
Microsoft(R) Visual Studio(R) Professional 2015		
Microsoft(R) Visual Studio(R) Enterprise 2015		
Microsoft(R) Visual Studio(R) Express 2015 for Windows Desktop	Visual Studio Express 2015 for Windows Desktop	

製品名称	このマニュアルでの表記		
Microsoft(R) Visual Studio(R) Community 2017	Visual Studio 2017		
Microsoft(R) Visual Studio(R) Professional 2017			
Microsoft(R) Visual Studio(R) Enterprise 2017			
Microsoft(R) Windows(R) Software Development Kit (v7.0) for Windows(R) 7 and .NET Framework 3.5 Service Pack 1	Windows SDK 7.0		Windows SDK
Microsoft(R) Windows(R) Software Development Kit (v7.1) for Windows(R) 7 and .NET Framework 4	Windows SDK 7.1		
Microsoft(R) Windows(R) 7 Enterprise	Windows 7	Windows 7	Windows
Microsoft(R) Windows(R) 7 Professional			
Microsoft(R) Windows(R) 7 Ultimate			
Microsoft(R) Windows(R) 7 Enterprise(x64)	Windows 7 x64 Edition		
Microsoft(R) Windows(R) 7 Professional(x64)			
Microsoft(R) Windows(R) 7 Ultimate(x64)			
Windows(R) 8 Enterprise	Windows 8	Windows 8	
Windows(R) 8 Pro			
Windows(R) 8 Enterprise(x64)	Windows 8 x64 Edition		
Windows(R) 8 Pro(x64)			
Windows(R) 8.1 Enterprise	Windows 8.1	Windows 8.1	
Windows(R) 8.1 Pro			
Windows(R) 8.1 Enterprise(x64)	Windows 8.1 x64 Edition		
Windows(R) 8.1 Pro (x64)			
Windows(R) 10 Pro	Windows 10	Windows 10	
Windows(R) 10 Enterprise			
Windows(R) 10 Pro (x64)	Windows 10 x64 Edition		
Windows(R) 10 Enterprise(x64)			
Microsoft(R) Windows Server(R) 2008 R2 Datacenter Edition	Windows Server 2008 R2		
Microsoft(R) Windows Server(R) 2008 R2 Enterprise Edition			
Microsoft(R) Windows Server(R) 2008 R2 Standard Edition			

製品名称	このマニュアルでの表記	
Microsoft(R) Windows Server(R) 2012 Datacenter	Windows Server 2012	
Microsoft(R) Windows Server(R) 2012 Standard		
Microsoft(R) Windows Server(R) 2012 R2 Datacenter	Windows Server 2012 R2	
Microsoft(R) Windows Server(R) 2012 R2 Standard		
Microsoft(R) Windows Server(R) 2016 Datacenter	Windows Server 2016	
Microsoft(R) Windows Server(R) 2016 Standard		

- TP1/Extension for .NET Framework，TP1/Client for .NET Framework，および TP1/Connector for .NET Framework を総称する場合は，OpenTP1 for .NET Framework と表記しています。

■ 略語一覧

このマニュアルで使用する英略語の一覧を次に示します。

英略語	英字での表記
ADO	ActiveX Data Object
AP	Application Program
API	Application Programming Interface
ASP	Active Server Pages
CLR	Common Language Runtime
CLS	Common Language Specification
CTS	Common Type System
CUP	Client User Program
CUP.NET	Client User Program for .NET Framework
DB	Data Base
DID	Distributed Identifier
DLL	Dynamic Linking Library
EBCDIK	Extended Binary Coded Decimal Interchange Kana Code
GAC	Global Assembly Cache
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol

英略語	英字での表記
IIS	Internet Information Services
MHP	Message Handling Program
MSDTC	Microsoft Distributed Transaction Coordinator
OS	Operating System
PC	Personal Computer
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
SPP	Service Providing Program
SPP.NET	Service Providing Program for .NET Framework
SUP	Service Using Program
SUP.NET	Service Using Program for .NET Framework
TCP/IP	Transmission Control Protocol/Internet Protocol
TSP	TP1 Service Proxy
UAC	User Account Control
UAP	User Application Program
UI	User Interface
WCF	Windows Communication Foundation
WS	Workstation
WSDL	Web Services Description Language
WS-I	Web Services Interoperability Organization
WWW	World Wide Web
XA	Extended Architecture
XML	eXtensible Markup Language

■ このマニュアルで使用している記号

注意事項

　間違いやすい点や注意しなければならない点などについて説明しています。

ポイント

　その説明の要点などについて説明しています。

参考

補足的な情報などについて説明しています。

■ KB（キロバイト）などの単位表記について

1KB（キロバイト），1MB（メガバイト），1GB（ギガバイト），1TB（テラバイト）はそれぞれ 1,024 バイト，$1{,}024^2$ バイト，$1{,}024^3$ バイト，$1{,}024^4$ バイトです。

目次

1 概要

この章では，TP1/Connector for .NET Framework（Connector .NET）の概要について説明します。Connector .NET は，OpenTP1 for .NET Framework を構成する 3 つの製品のうちの 1 つです。OpenTP1 for .NET Framework の概要については，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。なお，必要に応じて次のマニュアルも参照してください。

・「OpenTP1 解説」

・「TP1/LiNK 使用の手引」

1.1　前提条件

Connector .NET を使用する場合の前提条件については，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。

1.1.1　注意事項

前提ソフトウェアのサポート終了に伴い，次の言語のサポートを終了しました。マニュアルには次の言語の記述がありますが，サポートしておりません。

- J#
- COBOL 言語

2 機能

この章では，TP1/Connector for .NET Framework（Connector .NET）の機能について説明します。

2.1　リモートプロシジャコール（RPC）

Connector .NET のアプリケーションは，ほかの UAP とリモートプロシジャコール（RPC）を使用して通信できます。

2.1.1　RPC の形態による RPC インタフェースの使用可否

Connector .NET のアプリケーションからサービス要求をする場合の，RPC の形態による RPC インタフェースの使用可否を次の表に示します。

なお，RPC の形態および RPC インタフェースの詳細については，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。

表 2-1　RPC の形態による RPC インタフェースの使用可否

RPC の形態	RPC インタフェース		
	.NET インタフェース定義を使用した RPC	サービス定義（カスタムレコード）を使用した RPC	バイナリデータ（インデクスドレコード）を使用した RPC
同期応答型 RPC	○	○	○
非同期応答型 RPC	×	×	×
非応答型 RPC	○※	○	○
連鎖 RPC	○	○	○

（凡例）

○：使用できます。

×：使用できません。

注※

呼び出すメソッドの戻り値のデータ型が System.Void で，かつ引数が値渡しだけの場合に使用できます。それ以外の場合は RPC 要求時に例外が発生します。

2.1.2　RPC 送受信メッセージの最大長拡張機能

RPC 送受信メッセージの最大長拡張機能を使用すると，TP1Connection クラスの Execute メソッドで送受信できるユーザメッセージの最大長を 1～8 メガバイトで指定できます。ただし，RPC 送受信メッセージの最大長拡張機能を使用しない場合の RPC 送受信メッセージの最大長は 1 メガバイトです。この機能を使用する場合，option 要素の maxMessageSize 属性を指定してください。

この機能を使用する場合，次の点に注意してください。

- RPC 送受信メッセージの最大長拡張機能を使用する場合，次に示す製品でも同様の機能を使用できるように設定してください。

- TP1/Server
- Extension .NET
- Client .NET

設定方法については，マニュアル「OpenTP1 システム定義」，マニュアル「TP1/Extension for .NET Framework 使用の手引」，およびマニュアル「TP1/Client for .NET Framework 使用の手引」の RPC 送受信メッセージの最大長拡張機能の説明を参照してください。

- .NET インタフェース定義を使用する場合は，Connector .NET で設定した最大長の値以上の値を，TP1/Server や Extension .NET の RPC 送受信メッセージの最大長拡張機能に設定してください。

2.1.3 RPC データの XML マッピング機能

RPC データの XML マッピング機能を使用すると，入出力パラメタおよび戻り値を XML 文書（XmlDocument オブジェクト）（System.Xml.XmlDocument）として RPC が行えます。クライアントスタブが，XML 文書と.NET インタフェース定義のサービスメソッドの入出力パラメタおよび戻り値，またはサービス定義（カスタムレコード）との変換を行います。したがって，SPP.NET または SPP に送信するデータ，および SPP.NET または SPP から受信したデータを XML 文書として扱いたい場合，.NET Framework の DataSet オブジェクト（System.Data.DataSet）で利用したい場合などに便利です。

この機能は，Connector .NET 用のクライアントスタブでだけ使用できます。

詳細については，「4.1.1(4) .NET インタフェース定義から XML スキーマへのマッピング」，および「4.1.2(3) サービス定義から XML スキーマへのマッピング」を参照してください。

(1) RPC データの XML マッピング機能を使用した場合の RPC インタフェースの種類

OpenTP1 for .NET Framework の UAP から，サービス要求をする場合の RPC インタフェースには次の 3 種類があります。

- .NET インタフェース定義を使用した RPC
- サービス定義（カスタムレコード）を使用した RPC
- バイナリデータ（インデクスドレコード）を使用した RPC

上記の RPC インタフェースの概要については，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。

Connector .NET を使用すると，RPC データの XML マッピング機能を使用して，サービス要求をすることもできます。RPC データの XML マッピング機能を使用した場合，バイナリデータ（インデクスドレコード）を使用した RPC は実行できません。

ここでは，RPC データの XML マッピング機能を使用した RPC について説明します。

(a)　.NET インタフェース定義を使用した RPC

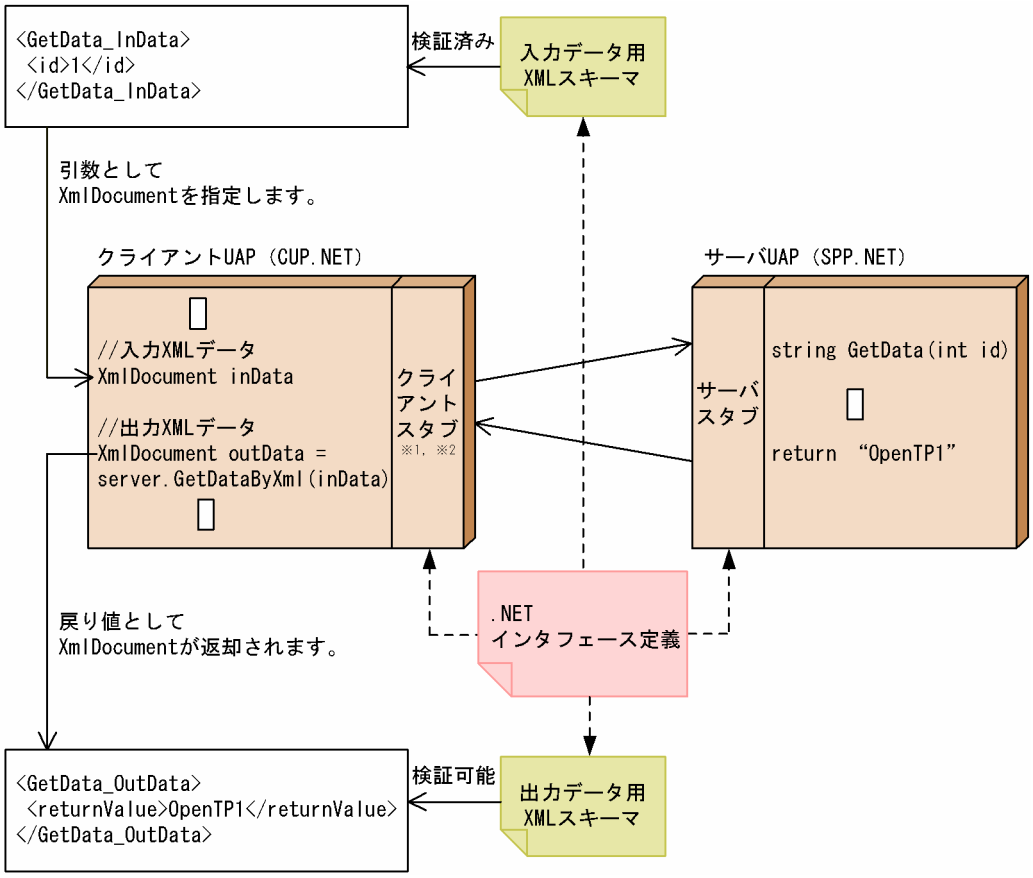
RPC データの XML マッピング機能を使用すると，入出力パラメタおよび戻り値を XML 文書（XmlDocument オブジェクト）（System.Xml.XmlDocument）として RPC が行えます。

クライアントスタブが XML 文書とサービスメソッドの入出力パラメタおよび戻り値との変換を行います。したがって，SPP.NET に送信するデータ，および SPP.NET から受信したデータを XML 文書として扱いたい場合，.NET Framework の DataSet オブジェクト（System.Data.DataSet）で利用したい場合などに便利です。

なお，RPC データの XML マッピング機能を使用して RPC を行った場合でも，SPP.NET が送受信するデータ形式には影響ありません。

RPC データの XML マッピング機能を使用した場合に，.NET インタフェース定義を使用した RPC の概要を次の図に示します。

図 2-1　.NET インタフェース定義を使用した RPC の概要（RPC データの XML マッピング機能を使用した場合）



注※1

サービス定義から生成されるクライアントスタブとは異なるものです。

注※2

RPC データの XML マッピング機能を使用したクライアントスタブは，Connector .NET でだけ生成および使用ができます。

入力データ用 XML スキーマに従った XML 文書を，サービスメソッドの引数として指定して RPC を行います。応答データがある場合，出力データ用 XML スキーマに従った XML 文書がサービスメソッドの戻り値として返されます。

(b) サービス定義（カスタムレコード）を使用した RPC

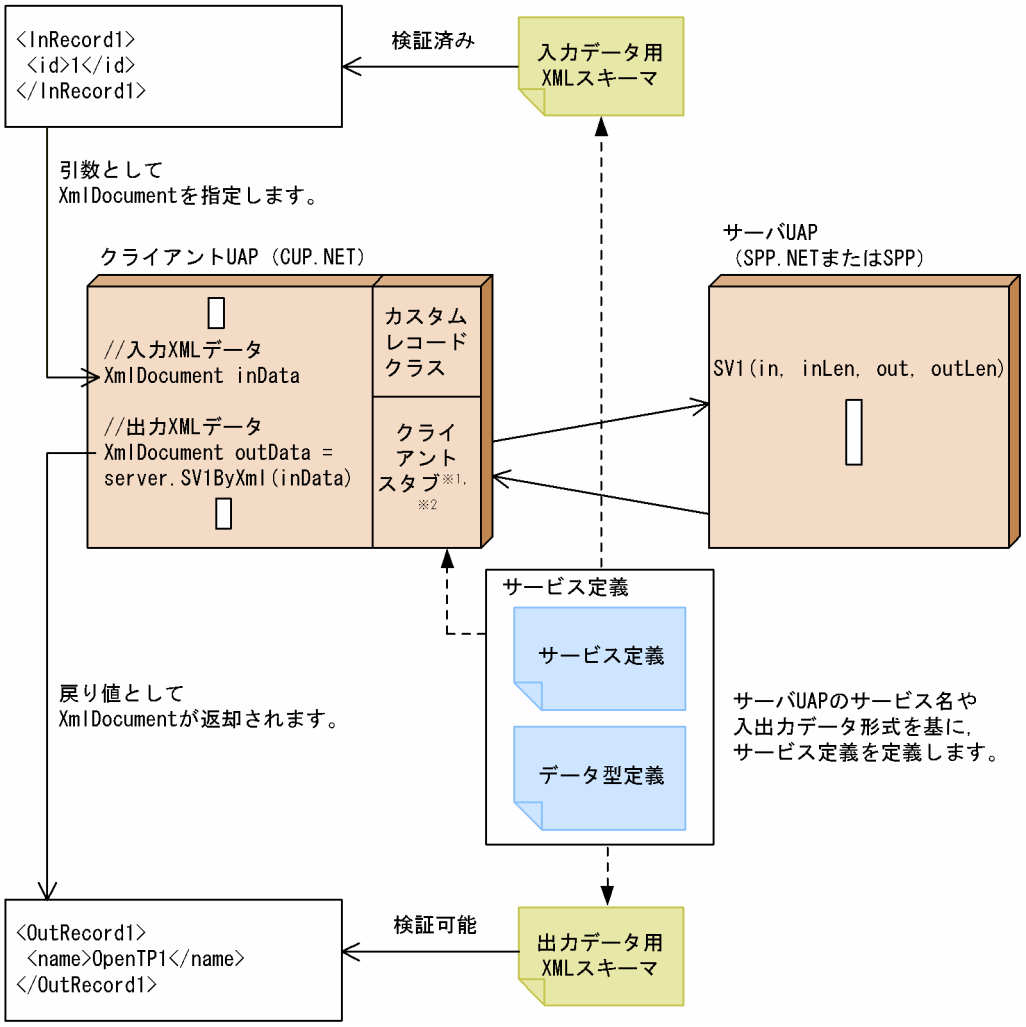
RPC データの XML マッピング機能を使用すると，入出力パラメタおよび戻り値を XML 文書（XmlDocument オブジェクト）（System.Xml.XmlDocument）として RPC が行えます。

クライアントスタブが XML 文書とカスタムレコードとの変換を行います。したがって，SPP.NET または SPP に送信するデータ，および SPP.NET または SPP から受信したデータを XML 文書として扱いたい場合，.NET Framework の DataSet オブジェクト（System.Data.DataSet）で利用したい場合などに便利です。

なお，RPC データの XML マッピング機能を使用して RPC を行った場合でも，SPP.NET または SPP が送受信するデータ形式には影響ありません。

RPC データの XML マッピング機能を使用した場合に，サービス定義（カスタムレコード）を使用した RPC の概要を次の図に示します。

図 2-2　サービス定義（カスタムレコード）を使用した RPC の概要（RPC データの XML マッピング機能を使用した場合）



注※1

.NET インタフェース定義から生成されるクライアントスタブとは異なるものです。

注※2

RPC データの XML マッピング機能を使用したクライアントスタブは，Connector .NET でだけ生成および使用ができます。

入力データ用 XML スキーマに従った XML 文書を，サービスメソッドの引数として指定して RPC を行います。応答データがある場合，出力データ用 XML スキーマに従った XML 文書がサービスメソッドの戻り値として返されます。

2.2　トランザクション制御機能

Connector .NET から OpenTP1 のトランザクションを制御できます。

2.2.1　トランザクションの種類

トランザクションには，次の 2 種類があります。

- ローカルトランザクション
  詳細は，「2.2.2　ローカルトランザクションの概要」を参照してください。
- 分散トランザクション
  詳細は，「2.2.7　分散トランザクションの概要」を参照してください。

2.2.2　ローカルトランザクションの概要

OpenTP1 が管理するトランザクションです。Connector .NET を使用するアプリケーション上で OpenTP1 以外のリソースマネジャへのアクセスをトランザクションに含めることはできません。

2.2.3　ローカルトランザクション制御機能を使用する場合の設定

トランザクション制御をする場合，Connector .NET に指定する Client .NET 構成定義およびサーバ側の TP1/Server Base の定義を次のように指定しておく必要があります。

(1)　Client .NET 構成定義

オートコネクトモードでリモート API 機能を使用するように指定します。

【指定例】

```
…
<tp1Server host="" />
<rpc use="rap" watchTime="0" />
<rapService port="10020" autoConnect="true"/>
…
```

(2)　TP1/Server Base のユーザサービス定義

サーバ側の TP1/Server Base のトランザクションとして実行する SPP は，ユーザサービス定義の atomic_update オペランドに Y を指定します。

2.2.4　ローカルトランザクションの開始と同期点取得

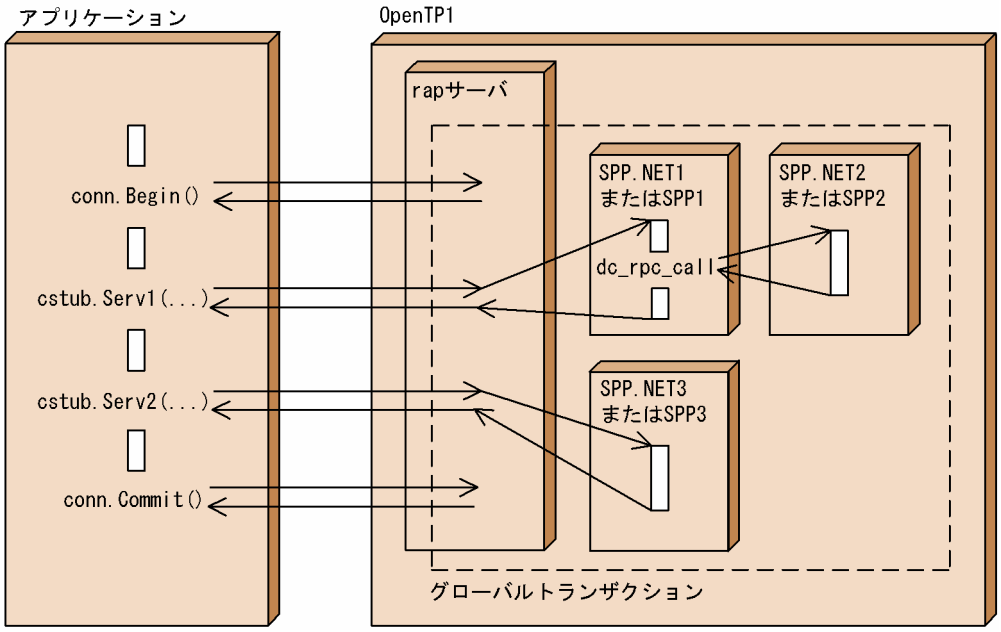
Connector .NET のアプリケーションから TP1Connection クラスの Begin メソッドを呼び出して，トランザクションを開始します。

Begin メソッドを呼び出してから，同期点取得（コミット）までが，グローバルトランザクションの範囲となります。Begin メソッドを呼び出したあと，そのグローバルトランザクションの中で新たな Begin メソッドは呼び出せません。

Connector .NET のアプリケーションから SPP.NET または SPP に対して RPC 要求を実行すると，アプリケーションは rap サーバでルートトランザクションブランチを生成し，呼び出した SPP.NET または SPP はトランザクションブランチとして実行されます。

トランザクションと RPC の関係を次の図に示します。

図 2-3　トランザクションと RPC の関係



なお，Begin メソッドの呼び出しから，RPC 要求，同期点取得の間は，同一の TP1Connection オブジェクトである必要があります。

2.2.5　ローカルトランザクションの同期点取得

(1)　コミット

トランザクションが正常終了したときの同期点取得（コミット）は，Connector .NET のアプリケーションから TP1Connection クラスの Commit メソッドを呼び出して要求します。TP1Connection クラスの Commit メソッドは，TP1Client クラスの Commit メソッドを呼び出します。

グローバルトランザクションは，すべてのトランザクションブランチが正常に終了したことで正常終了となります。

次の場合は，トランザクションはロールバックされます。

- TP1Connection クラスの Commit メソッドを呼び出さないでアプリケーションが終了したとき
- TP1Connection クラスの Commit メソッドを呼び出す前にアプリケーションが異常終了したとき

(2) ロールバック

(a) TP1/Server の処理でのエラーの場合

トランザクションでエラーが発生すると，TP1Connection クラスの Commit メソッドで例外が発生します。そのトランザクションは部分回復対象としてロールバックされます。グローバルトランザクション内のどれか一つのトランザクションブランチでエラーが発生した場合でも，グローバルトランザクション全体がロールバックの対象となります。

このとき TP1/Server は，トランザクションブランチをロールバック対象と見なして，部分回復処理をします。

(b) ロールバック要求メソッドを呼び出す場合

トランザクションを Connector .NET のアプリケーションの判断でロールバックしたいときは，Connector .NET のアプリケーションから TP1Connection クラスの Rollback メソッドを呼び出して要求します。TP1Connection クラスの Rollback メソッドは，TP1Client クラスの Rollback メソッドを呼び出します。

(3) トランザクションの処理時間について

トランザクションに関する次に示す時間を Client .NET 構成定義で指定できます。

- トランザクション同期点処理時の最大通信待ち時間
- トランザクションブランチ最大実行可能時間

詳細については，マニュアル「TP1/Client for .NET Framework 使用の手引」の構成定義についての記述を参照してください。

2.2.6 障害発生時のローカルトランザクションの同期点を検証する方法

Connector .NET のアプリケーションから開始したトランザクションで障害が発生した場合，そのトランザクションブランチがコミットされたかどうかを検証できます。トランザクションの同期点を検証するときは，構成定義の<log>要素の level 属性に 2 を設定しておきます。

Connector .NET ではトランザクション開始後，トランザクショングローバル識別子およびトランザクションブランチ識別子を Connector .NET ログファイルに出力します。

Connector .NET ログファイルのトランザクショングローバル識別子と，サーバ側のメッセージログファイルに出力されるトランザクションの結果を突き合わせることによって，アプリケーションから開始したトランザクションがコミットされたかどうか検証できます。

2.2.7　分散トランザクションの概要

Connector .NET は，Microsoft Distributed Transaction Coordinator（MSDTC）の分散トランザクションと OpenTP1 のトランザクションとの 2 相コミットによるトランザクション連携を行い，**MSDTC 連携機能**を実現します。MSDTC 連携機能を使用すると，MSDTC を利用するほかのリソースと OpenTP1 が使用するリソースとの間で分散トランザクション連携ができます。

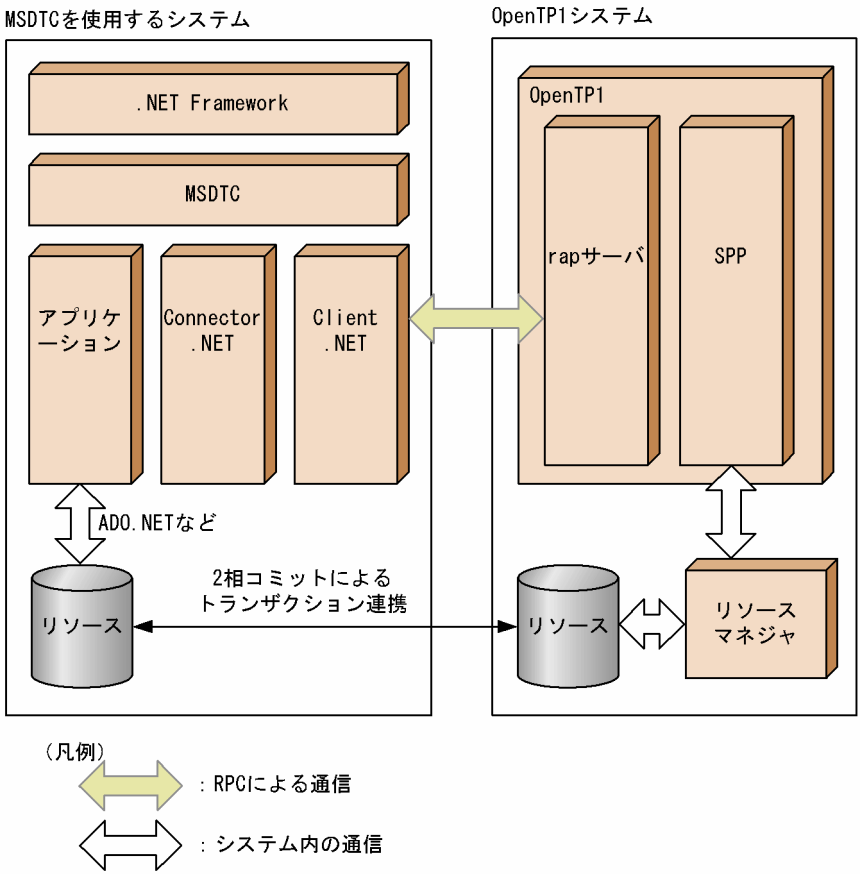
MSDTC 連携機能を使用したアプリケーションでは，.NET Framework が提供する System.Transactions 名前空間内の API を使用することで，OpenTP1 をリソースマネジャとして MSDTC の分散トランザクションに参加させることができます。また，アプリケーションや OpenTP1 で障害が発生した場合，Connector .NET が提供する**トランザクションリカバリサービス**がトランザクションを自動的に回復します。これによって，MSDTC の分散トランザクションで OpenTP1 と OpenTP1 以外のリソースとの整合性を保証します。トランザクションリカバリサービスの詳細については，「2.2.10　分散トランザクションのトランザクションリカバリサービス」を参照してください。

MSDTC 連携機能では，OpenTP1 が提供する XA リソースサービスを使用します。XA リソースサービスの詳細については，マニュアル「OpenTP1 解説」を参照してください。

(1)　MSDTC 連携機能の概要

MSDTC 連携機能の概要を次の図に示します。

図 2-4　MSDTC 連携機能の概要

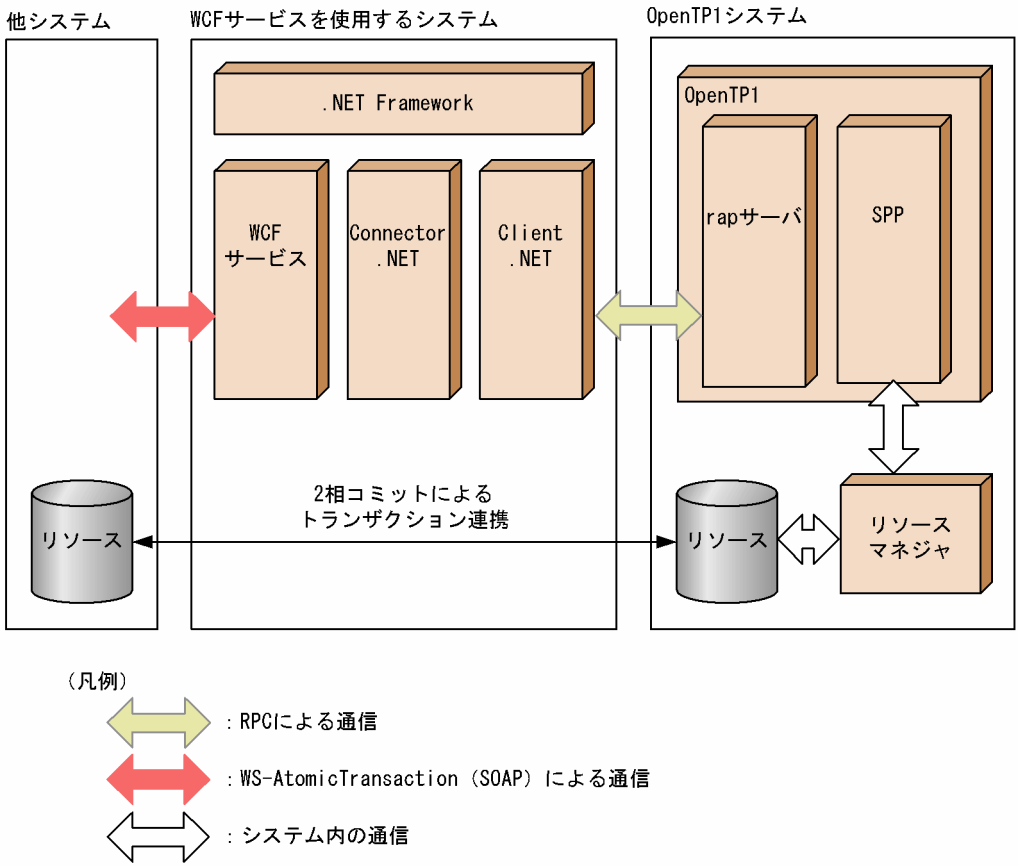


(2)　WCF 連携機能使用時の MSDTC 連携機能の概要

MSDTC 連携機能を WCF サービスで使用することで，他システムとの WS-AtomicTransaction によるトランザクション連携もできます。このとき，WCF サービスから OpenTP1 へのサービス要求には，WCF 連携機能を使用することもできます。WCF 連携機能の詳細については「2.9　WCF 連携機能」を参照してください。

WCF 連携機能使用時の MSDTC 連携機能の概要を次の図に示します。

図 2-5　WCF 連携機能使用時の MSDTC 連携機能の概要



(3)　前提条件

(a)　前提ソフトウェア

MSDTC 連携機能を使用する場合，次に示すソフトウェアが必要です。

- TP1/Client for .NET Framework 07-50 以降
- .NET Framework v3.5 Service Pack 1

また，トランザクション連携をする接続先の OpenTP1 は，次のどちらかである必要があります。

- TP1/Server Base 07-50 以降
- TP1/LiNK 07-51 以降

(4)　MSDTC 連携機能で使用できる RPC

MSDTC 連携機能で使用できる RPC を次の表に示します。

表 2-2　MSDTC 連携機能で使用できる RPC

分類	RPC の方式	RPC の使用可否	
		WCF 連携機能使用時	WCF 連携機能未使用時
RPC の種類	リモート API 機能を使用した RPC	○	○
	ネームサービスを使用した RPC	×	×
	スケジューラダイレクト機能を使用した RPC	×	×
コネクトモード	オートコネクトモード	○	○
	非オートコネクトモード	×	×
RPC の形態	同期応答型 RPC	○	○
	非同期応答型 RPC	×	×
	非応答型 RPC	×	○
	連鎖型 RPC	×	○
	トランザクションを引き継がない RPC	×	○

（凡例）

○：使用できます。

×：使用できません。

2.2.8　分散トランザクションの単一フェーズコミット最適化

MSDTC 連携機能では，トランザクションの**単一フェーズコミット最適化**を行うことができます。単一フェーズコミット最適化とは，トランザクションに参加するリソースが OpenTP1 のリソース一つだけである場合に，単一フェーズコミットを行う機能です。トランザクションに参加するリソースが複数ある場合は，自動的に 2 相コミットを行います。

単一フェーズコミット最適化を行う場合は，<distributedTransaction>要素の optimize1PC 属性に true を指定してください。false を指定した場合，トランザクションに対して常に 2 相コミットが行われます。

(1)　単一フェーズコミット最適化を使用する場合

単一フェーズコミット最適化を使用する場合で，トランザクションに参加するリソースが OpenTP1 のリソース一つだけであるときは，単一フェーズコミットが発行されるため，処理が簡略化されます。

ただし，次に示す内容を考慮して単一フェーズコミット最適化を使用してください。

- 単一フェーズコミット最適化を使用する場合に，OpenTP1 のリソースがトランザクションに参加した時点でほかのリソースが一つも参加していないと，MSDTC の DID が OpenTP1 に通知されません。

そのため，OpenTP1 が管理するトランザクションを強制決着する際に，MSDTC が管理するトランザクションと照合できないことがあります。

- 単一フェーズコミット最適化を使用する場合で，単一フェーズコミットを行う際に OpenTP1 に対して通信エラーが発生したときは，アプリケーションで System.Transactions.TransactionInDoubtException 例外が発生します。このときトランザクションがコミットまたはロールバックされたかは不明となります。

(2)　単一フェーズコミット最適化を使用しない場合

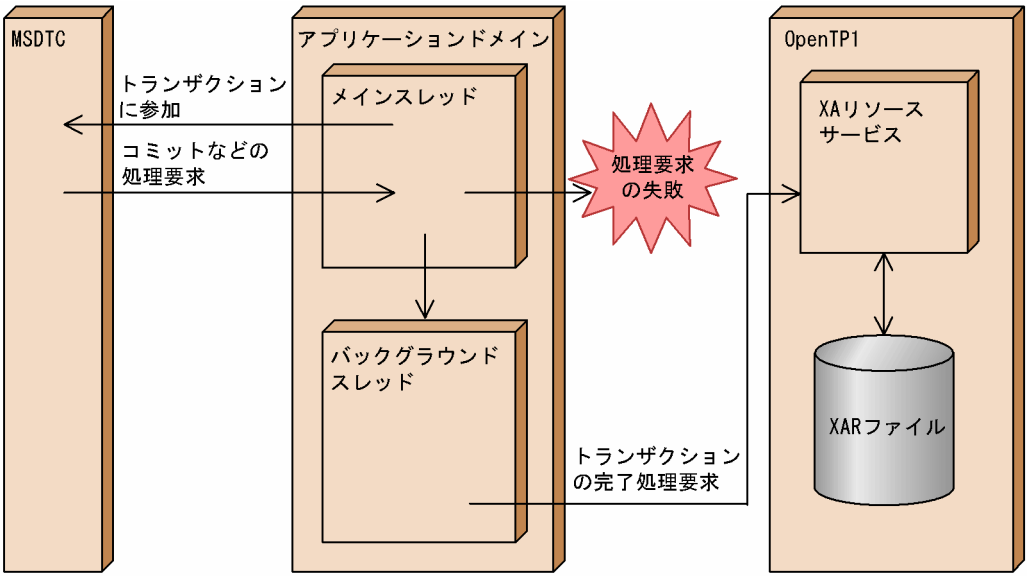
単一フェーズコミット最適化を使用しない場合は，MSDTC の DID が常に OpenTP1 に通知されます。このため，トランザクションの強制決着を行う場合などに，MSDTC と OpenTP1 のトランザクションのステータスを照合できます。

ただし，常に 2 フェーズコミットが行われるため，トランザクションに参加する OpenTP1 のリソースが一つであっても，単一フェーズコミット最適化を行う場合に比べて通信回数が増加します。

2.2.9　未決着トランザクションの完了処理

アプリケーションで未決着となったトランザクションの完了処理は，Connector .NET が生成するバックグラウンドスレッドが行います。このバックグラウンドスレッドは，アプリケーションドメインでトランザクションとして扱われる最初のサービス要求時に一度だけ生成されます。未決着トランザクションの完了処理の流れを次の図に示します。

図 2-6　未決着トランザクションの完了処理



アプリケーションからの処理要求（コミット準備，コミット，ロールバック処理）に失敗した場合，バックグラウンドスレッドはアプリケーションで失敗した処理要求を自動的に完了させます。バックグラウンドスレッドでトランザクションの完了処理に失敗した場合，バックグラウンドスレッドは一定間隔で再度

トランザクションの完了処理を試みます。この間隔は，<distributedTransaction>要素の recoverRetryInterval 属性で指定します。

OpenTP1 が障害中などの理由でトランザクションの完了処理に失敗した場合は，失敗するたびにログメッセージが出力されます。ログレベルを 3 に設定している場合は，トランザクションの完了処理を再試行するたびにコネクションの取得および解放に関するログメッセージが出力されます。そのため，MSDTC 連携機能を使用する場合は，アプリケーションの構成定義の<log>要素の fileSize 属性に十分大きな値を指定してください。

2.2.10　分散トランザクションのトランザクションリカバリサービス

トランザクションリカバリサービスとは，Connector .NET が提供する Windows サービスです。トランザクションリカバリサービスは，既定では停止の状態となっています。MSDTC 連携機能を使用する場合は，必ずトランザクションリカバリサービスを開始の状態にしてください。インストール時に Connector .NET のインストーラによって Windows サービスに登録されるトランザクションリカバリサービスの内容を次の表に示します。

表 2-3　トランザクションリカバリサービスの登録内容

タブ	項目	設定値
全般	サービス名	TP1ConnectorNETTRS
	表示名	TP1/Connector.NET Transaction Recovery Service
	説明	Recovers distributed transaction.
	実行ファイルのパス	"<Connector .NET のインストールディレクトリ>¥trs¥cnntrs.exe"
	スタートアップの種類	手動
ログオン	ログオン	ローカルシステムアカウント
	デスクトップとの対話をサービスに許可	（チェックなし）
回復	最初のエラー	何もしない
	次のエラー	何もしない
	その後のエラー	何もしない
	エラーカウントのリセット	0（日後に行う）
依存関係	－	依存関係なし

（凡例）

－：該当しません。

OS を起動する際にトランザクションリカバリサービスを自動的に起動させたい場合は，「スタートアップの種類」を変更してください。トランザクションリカバリサービスに障害が発生した場合にトランザクショ

ンリカバリサービスを自動的に再起動させたいときは，「最初のエラー」，「次のエラー」，「その後のエラー」および「エラーカウントのリセット」を，それぞれ必要に応じて変更してください。

「スタートアップの種類」，「最初のエラー」，「次のエラー」，「その後のエラー」および「エラーカウントのリセット」以外の項目は変更しないでください。変更した場合，トランザクションリカバリサービスが正常に動作しないことがあります。

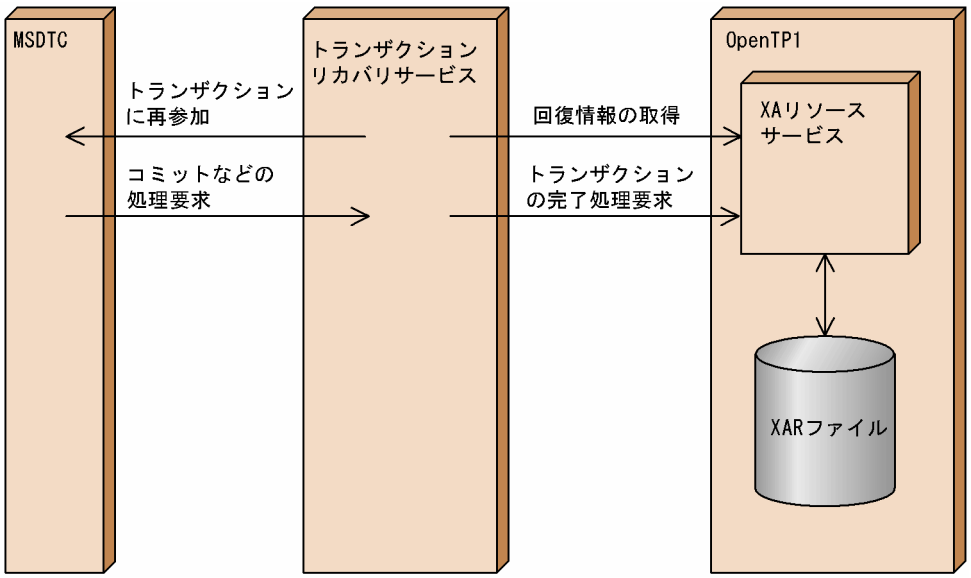
トランザクションリカバリサービスが提供する機能について説明します。

(1)　未決着トランザクションの回復

トランザクションリカバリサービスは，トランザクションリカバリサービスの開始時とアプリケーションドメイン終了の検知時に，未決着トランザクションを回復します。アプリケーションドメイン終了の検知については「(2)　アプリケーションドメインの監視」を参照してください。

未決着トランザクションの回復の流れを次の図に示します。

図 2-7　未決着トランザクションの回復の流れ



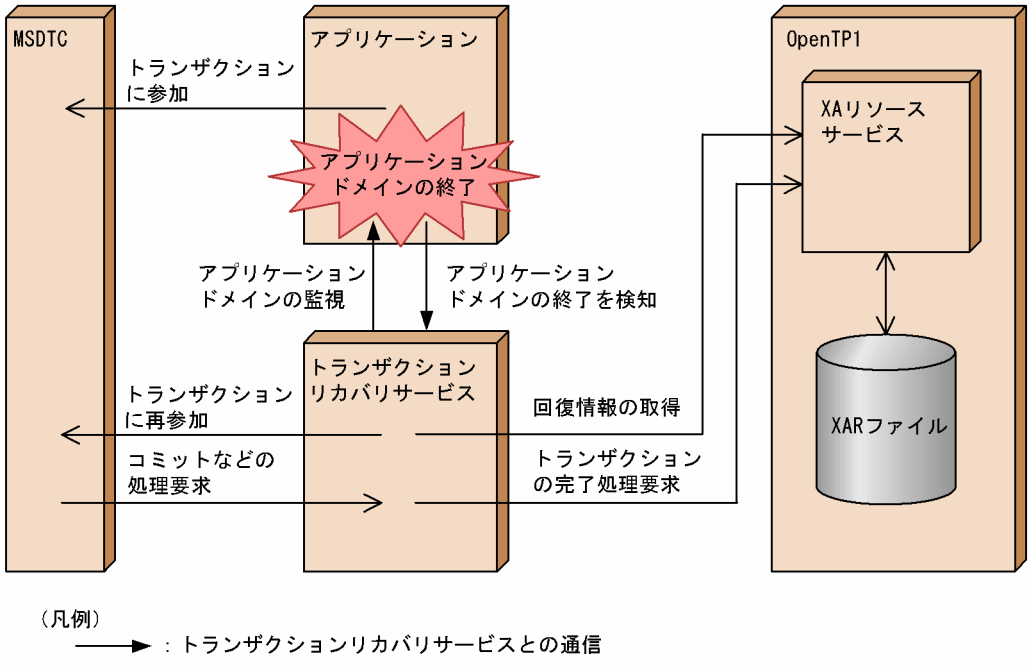
トランザクションリカバリサービスは，トランザクションリカバリサービスの開始時とアプリケーションドメイン終了の検知時に，OpenTP1 の XA リソースサービスから未決着トランザクションの回復情報を取得し，その回復情報を基に MSDTC に対してトランザクションの再参加を行います。そのあとで，アプリケーションで失敗した処理要求を MSDTC から受けたトランザクションリカバリサービスが，トランザクションの処理を完了させます。トランザクションリカバリサービスがトランザクションの完了に失敗した場合は，トランザクションリカバリサービスが再度トランザクションの回復を試みます。

OpenTP1 が未決着トランザクションの回復情報の取得やトランザクションの再参加に失敗した場合は，一定間隔で回復情報の取得を再度試みます。この間隔は，<recoveryService>要素の recoverRetryInterval 属性で指定します。

(2)　アプリケーションドメインの監視

トランザクションリカバリサービスは，MSDTC 連携機能を使用する Connector .NET のアプリケーションドメインを監視します。トランザクションリカバリサービスの監視下にあるアプリケーションドメインが終了した場合，終了したアプリケーションドメインで未決着となっていたトランザクションを回復します。アプリケーションドメインの監視の流れを次の図に示します。

図 2-8　アプリケーションドメインの監視の流れ



アプリケーションドメインの監視は，アプリケーションドメインでトランザクションとして扱われる最初のサービス要求時に自動的に開始されます。監視が開始されてからトランザクションリカバリサービスがダウンして，一定時間経過後にトランザクションリカバリサービスが復旧していた場合は，トランザクションとして扱われる最初のサービス要求時に再度監視が開始されます。アプリケーションドメインの監視は一定間隔で行われます。この間隔は，<recoveryService>要素の appDomainCheckInterval 属性で指定します。この間隔がアプリケーションダウンなどが発生してからトランザクションの回復処理が行われるまでの最大時間となります。

トランザクションリカバリサービスは，アプリケーションドメインの終了を検知すると，OpenTP1 の XA リソースサービスから未決着トランザクションの回復情報を取得し，その回復情報を基に MSDTC に対してトランザクションの再参加を行います。

コンソールアプリケーションなどでトランザクション範囲内の処理が終了し，コミットやロールバックの処理要求が完了する前にアプリケーションのプロセスが終了した場合は，トランザクションリカバリサービスがトランザクションのコミットやロールバックの処理を完了させます。

(3)　未決着トランザクションの確認処理

トランザクションリカバリサービスは，アプリケーションドメインの監視および未決着トランザクションの回復によってトランザクションの回復処理を完了したあとで，OpenTP1 に未決着トランザクションが存在しないかどうかを再確認します。これは，トランザクションリカバリサービスが回復処理を実行したタイミングで，OpenTP1 に処理中のトランザクションが存在した場合に OpenTP1 に未決着トランザクションが残ることを防止するためです。この確認で未決着トランザクションが存在した場合は，トランザクションの回復処理を行います。この確認処理の間隔は<recoveryService>要素の recoverCheckInterval 属性で，確認処理の回数は<recoveryService>要素の recoverCheckCount 属性で指定します。ただし，指定した回数の確認処理が終了する前に，新たに未決着トランザクションが存在する旨の通知を受けた場合やアプリケーションの終了を検知した場合は，確認処理を終了します。

2.2.11　MSDTC 連携機能を使用するための作業の流れ

MSDTC 連携機能を使用するために実施する作業の流れを次の図に示します。

図 2-9　MSDTC 連携機能を使用するために実施する作業の流れ

2.2.12　MSDTC 連携機能を使用する場合の TP1/Server の設定

MSDTC 連携機能でトランザクション連携をする場合，接続先（TP1/Server）の設定で XA リソースサービスを使用できるようしておく必要があります。

接続先（TP1/Server）で指定するトランザクションブランチの監視時間を指定する際は，アプリケーションで指定するトランザクションのタイムアウトを考慮してください。詳細については「2.2.13　分散トランザクションのアプリケーションの作成」を参照してください。

(1) TP1/Server Base の設定

TP1/Server Base での設定について説明します。

- ユーザサービス定義

  SPP.NET および SPP をトランザクションとして実行する場合は，atomic_update=Y が指定されていることを確認してください。

- rap リスナーサービス定義

  リモート API 機能の常設コネクションのスケジュール方法によって，次のように指定してください。

  - スタティックコネクションスケジュールモードを使用する場合

    rap サーバ数（rap_parallel_server オペランドで指定）が，次に示す値の合計値以上となるように指定してください。

    ・各アプリケーションの構成定義の<connection>要素の active 属性の指定値

    ・トランザクションリカバリサービスの構成定義の<connection>要素の active 属性の指定値

  - ダイナミックコネクションスケジュールモードを使用する場合

    rap_recovery_server オペランドにリカバリ要求用待機 rap サーバ数を指定してください。指定する値は，トランザクションリカバリサービスから回復処理要求を行うコネクションの数と合わせてください。

- トランザクションサービス定義

  XA リソースサービスの使用（xtrn_xar_use=Y）を指定してください。

- XA リソースサービス定義

  MSDTC 連携機能の使用（xar_msdtc_use=Y）を指定してください。また，xar_session_time オペランドにアイドル状態のトランザクションブランチの監視時間を指定してください。

定義の詳細については，マニュアル「OpenTP1 システム定義」を参照してください。

(2) TP1/LiNK の設定

TP1/LiNK での設定について説明します。

- ［SPP 環境設定］ダイアログボックスで［トランザクション属性(T)］チェックボックスをオンにしてください。
- リモート API 機能の常設コネクションのスケジュール方法によって，次のように指定してください。
  - スタティックコネクションスケジュールモードを使用する場合

    ［RAP サービス環境設定］ダイアログボックスの［RAP サービスのプロセス数］欄の［常駐(R)］が，次に示す値の合計値以上となるように指定してください。

    ・各アプリケーションの構成定義の<connection>要素の active 属性の指定値

    ・トランザクションリカバリサービスの構成定義の<connection>要素の active 属性の指定値

  - ダイナミックコネクションスケジュールモードを使用する場合

［RAP サービス詳細設定］ダイアログボックスの［その他］タブで［リカバリ要求用待機 rap サーバ数(R)］を指定してください。指定する値は，トランザクションリカバリサービスから回復処理要求を行うコネクションの数と合わせてください。

- ［システム環境設定］ウィンドウの［トランザクション機能］欄の［あり］チェックボックスをオンにしてください。
- ［XA リソースサービス環境設定］ダイアログボックスの［XA リソースサービスを使用する(X)］欄のチェックボックス，および［MSDTC 連携機能を使用する(M)］のチェックボックスをオンにしてください。また，［アイドル状態のトランザクションブランチの監視時間(T)］を指定してください。

定義の詳細については，マニュアル「TP1/LiNK 使用の手引」を参照してください。

2.2.13　分散トランザクションのアプリケーションの作成

分散トランザクションのアプリケーションの開発には，.NET Framework が提供する System.Transactions 名前空間内の API を使用します。

System.Transactions では，Transaction クラスに基づいた明示的なプログラミングモデルと，TransactionScope クラスを使用した暗黙的なプログラミングモデルが提供されています。それぞれのプログラミングモデルを使用した Connector .NET のアプリケーション開発例を次に示します。

明示的なプログラミングモデルを使用したアプリケーション開発例

```
TP1ConnectionManager tcm = new TP1ConnectionManager();

// コネクションの取得
TP1Connection tc = tcm.GetConnection();
               :

// タイムアウトを5分に設定
TimeSpan timeout = new TimeSpan(0,5,0);

using (CommittableTransaction ctrn =
          new CommittableTransaction(timeout))
{

// トランザクションの開始
Transaction.Current = ctrn;

// サービス要求の実行
bool ret = tc.Execute();

    try
    {

// トランザクションの完了
        ctrn.Commit();
    }
    catch (TransactionException exp)
    {
        TextBox1.Text = exp.ToString();
```

```
    }
}

// コネクションの解放
tc.Dispose();
```

暗黙的なプログラミングモデルを使用したアプリケーション開発例

```
TP1ConnectionManager tcm = new TP1ConnectionManager();

// コネクションの取得
TP1Connection tc = tcm.GetConnection();
               :

// タイムアウトを5分に設定
TimeSpan timeout = new TimeSpan(0,5,0);

// トランザクションの開始
using (TransactionScope ts =
new TransactionScope(TransactionScopeOption.Required, timeout))
{

// サービス要求の実行
bool ret = tc.Execute();

// トランザクションの完了
    ts.Complete();
}

// コネクションの解放
tc.Dispose();
```

(1) コネクションとトランザクションの関係

コネクションとトランザクションには，次に示す関係があります。

- トランザクション内でサービスを要求すると，そのコネクションが現在のスレッドコンテキストのトランザクションに参加します。
- 一つのコネクションを同時に複数のトランザクションに参加させることはできません。
- 複数のコネクションを一つのトランザクションに参加させることはできます。その場合，OpenTP1 が管理するトランザクションブランチは複数になります。
- トランザクションの操作が完了するまでコネクションは使用中のままとなります。
- トランザクション内で TP1Connection クラスの Dispose メソッドを呼び出した場合は，アプリケーションからコネクションオブジェクトを解放するだけとなります。
- 分散トランザクションに参加したコネクションでは，ローカルトランザクションを開始することはできません。
- 分散トランザクションに参加したコネクションで TP1Connection クラスの Begin メソッドを呼び出すと，TP1ConnectorException 例外が発生します。

- コネクションが分散トランザクションに参加していない場合は，アプリケーションのトランザクション範囲内に記述された TP1Connection クラスの Begin メソッドでもローカルトランザクションが開始されます。

暗黙的なプログラミングモデルを使用したアプリケーションの例とトランザクションの関係を次に示します。なお，明示的なプログラミングモデルの場合は，アプリケーションを次の表のように置き換えてください。

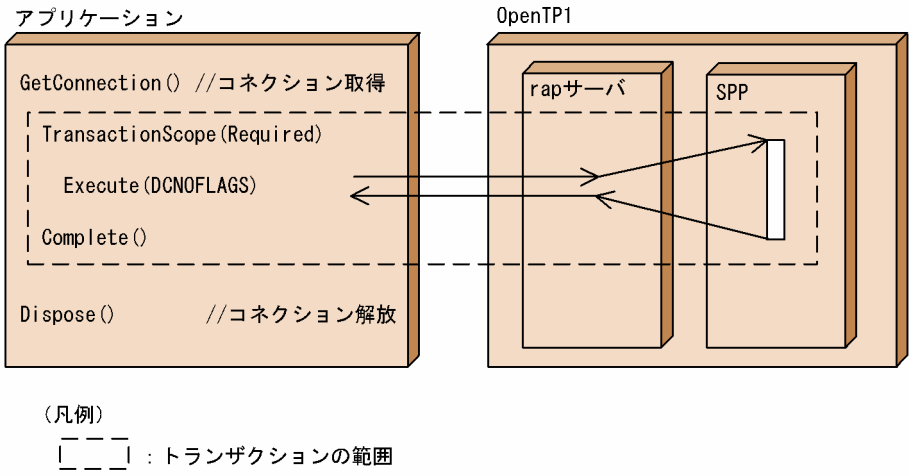
表 2-4　暗黙的なプログラミングモデルと明示的なプログラミングモデルの対応

暗黙的なプログラミングモデル	明示的なプログラミングモデル
TransactionScope(Required)	CommittableTransaction オブジェクトを Transaction.Current プロパティに設定
TransactionScope(RequiresNew)	CommittableTransaction オブジェクトを新規にインスタンス化して，Transaction.Current プロパティに設定
TransactionScope(Suppress)	Transaction.Current プロパティに null を設定
Complete()	Commit()または Rollback()の発行

(a)　RPC 実行時にトランザクションに参加する場合

RPC 実行時にトランザクションに参加する場合の，アプリケーションの例とトランザクションの関係を次の図に示します。

図 2-10　アプリケーションの例とトランザクションの関係（RPC 実行時にトランザクションに参加する場合）

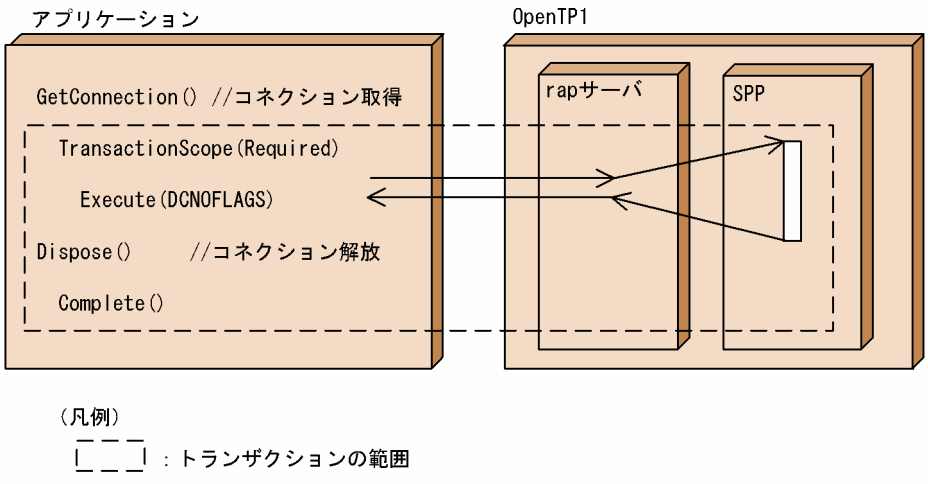


コネクションを取得してサービスの要求を実行する前後で，トランザクションに参加します。

(b)　トランザクションの終了前にコネクションを解放する場合

トランザクションの終了前にコネクションを解放する場合の，アプリケーションの例とトランザクションの関係を次の図に示します。

図 2-11　アプリケーションの例とトランザクションの関係（トランザクションの終了前にコネクションを解放する場合）

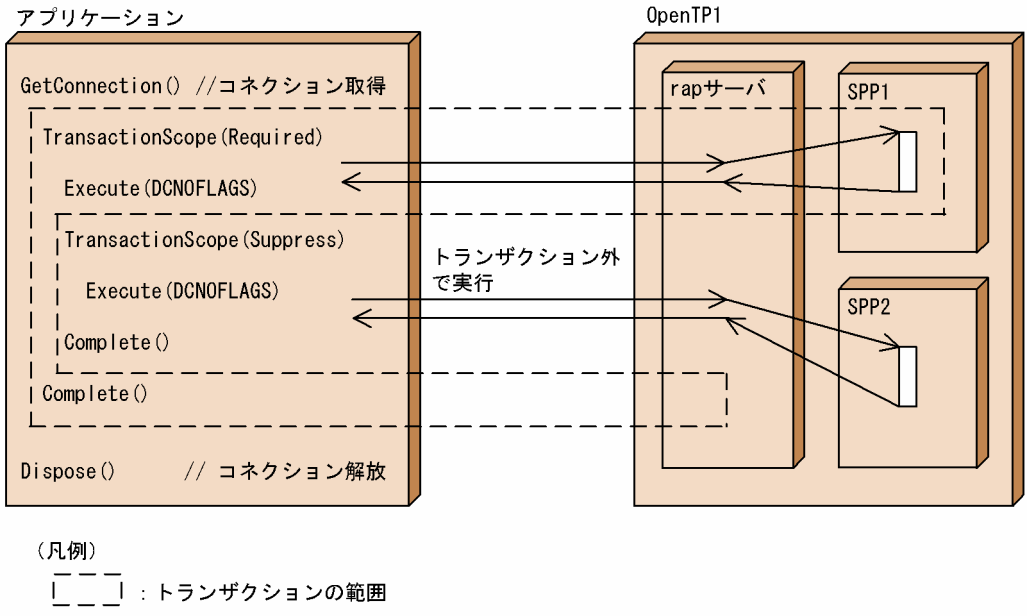


コネクションを解放してからトランザクションを終了します。トランザクションの開始後にコネクションを取得することもできます。

(c)　トランザクション内でトランザクションとしないサービスを要求する場合

トランザクション内でトランザクションとしないサービスを要求する場合の，アプリケーションの例とトランザクションの関係を次の図に示します。

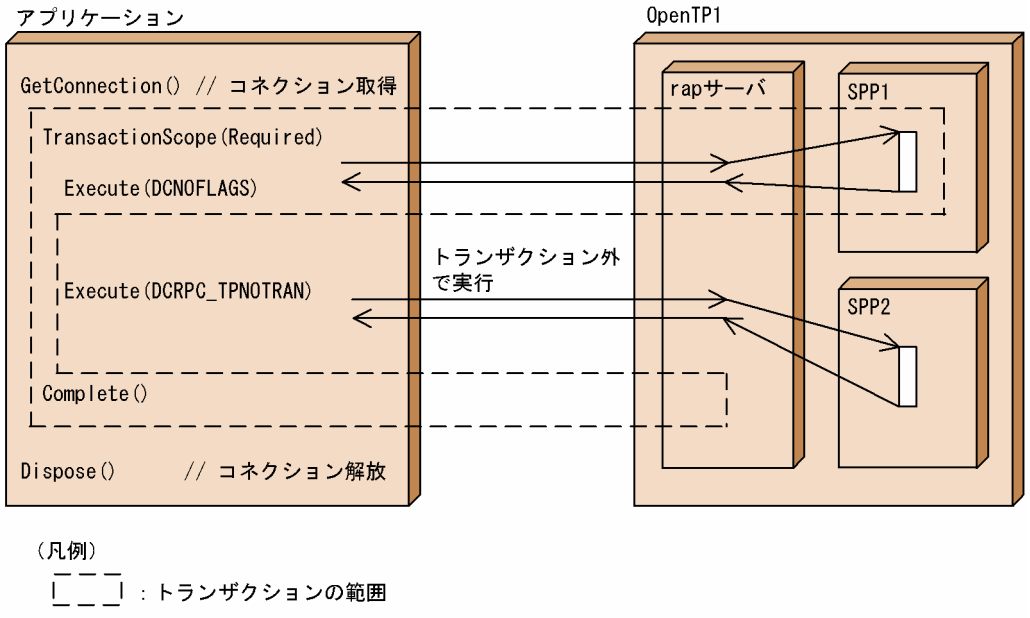
図 2-12　アプリケーションの例とトランザクションの関係（トランザクション内でトランザクションとしないサービスを要求する場合）



トランザクション内でトランザクションコンテキストの抑制を指定することで，サービスの要求をトランザクション外として扱うことができます。

また，トランザクション内で DCRPC_TPNOTRAN フラグを指定してサービスを要求した場合も同様の動作となります。トランザクション内で DCRPC_TPNOTRAN フラグを指定してサービスを要求した場合の，アプリケーションの例とトランザクションの関係を次の図に示します。

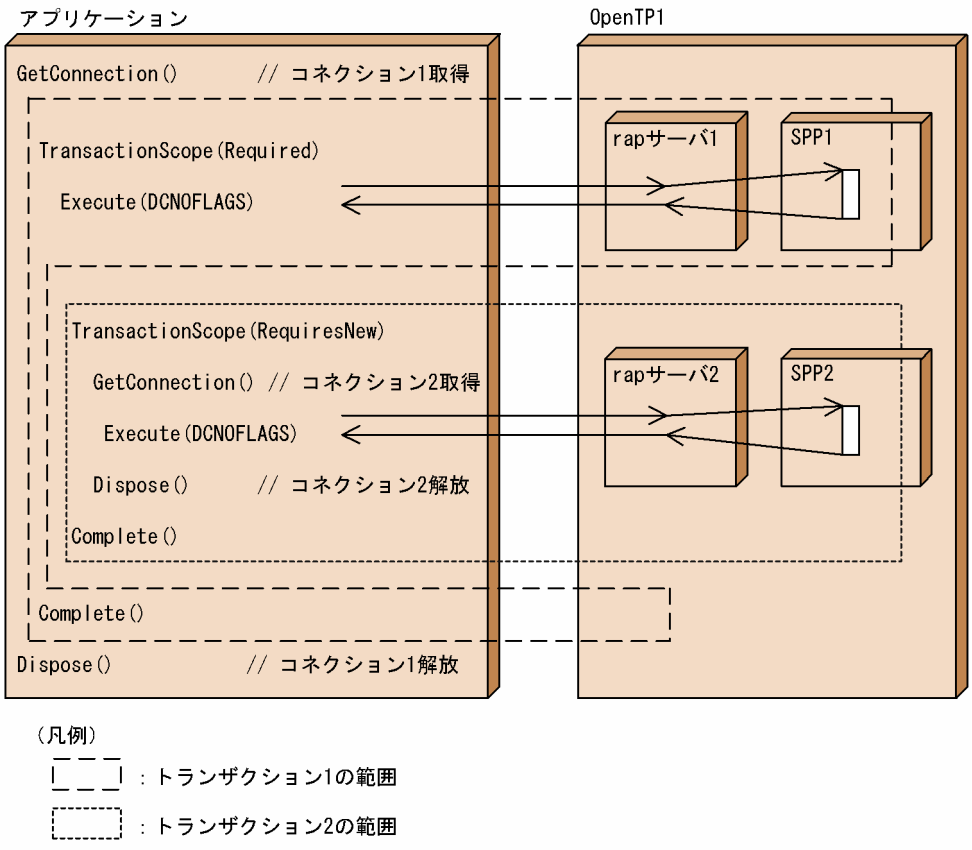
図 2-13　アプリケーションの例とトランザクションの関係（トランザクション内で DCRPC_TPNOTRAN フラグを指定してサービスを要求した場合）



(d)　トランザクション内で新しいトランザクションを作成する場合

トランザクション内で新しいトランザクションを作成する場合の，アプリケーションの例とトランザクションの関係を次の図に示します。

図 2-14　アプリケーションの例とトランザクションの関係（トランザクション内で新しいトランザクションを作成する場合）

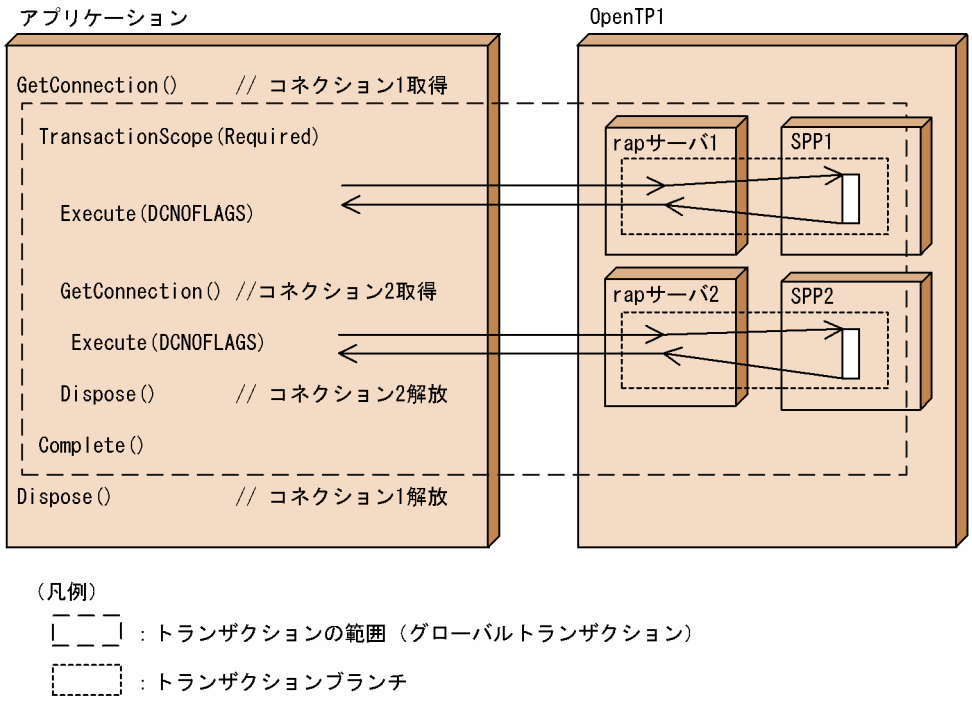


トランザクション内で新しいトランザクションを作成してサービスを要求する場合は，別に取得したコネクションを使用してください。一つのコネクションを同時に複数のトランザクションと関連づけた場合は，サービス要求時に TP1ConnectorException が返されます。

(e)　同一のトランザクション内で複数のコネクションを使用する場合

同一のトランザクション内で複数のコネクションを使用する場合の，アプリケーションの例とトランザクションの関係を次の図に示します。

図 2-15　アプリケーションの例とトランザクションの関係（同一のトランザクション内で複数のコネクションを使用する場合）

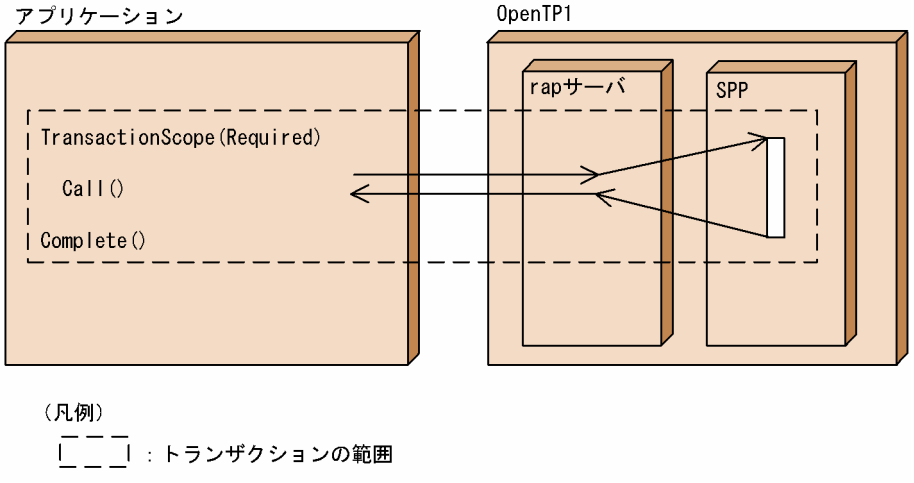


一つのトランザクション内で複数のコネクションを使用する場合は，複数のコネクションが同じトランザクションに参加します。

(f)　WCF 連携機能を使用した場合

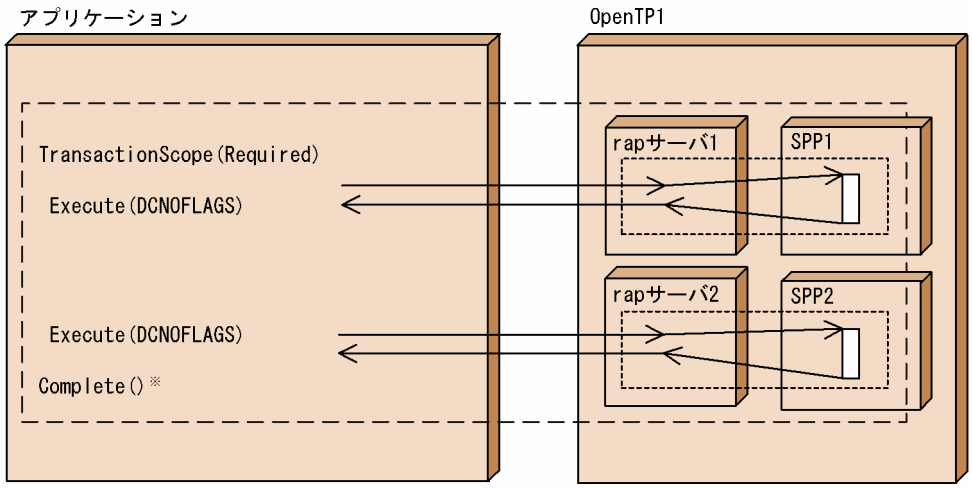
WCF 連携機能を使用した場合の，アプリケーションの例とトランザクションの関係を次の図に示します。

図 2-16　アプリケーションの例とトランザクションの関係（WCF 連携機能を使用した場合）



また，WCF 連携機能を使用した場合でグローバルトランザクションのときの，アプリケーションの例とトランザクションの関係を次の図に示します。

図 2-17　アプリケーションの例とトランザクションの関係（WCF 連携機能を使用した場合でグローバルトランザクションのとき）



（凡例）

：トランザクションの範囲（グローバルトランザクション）

：トランザクションブランチ

注※
Complete()が発行されるまでコネクションは使用中のままとなります。

WCF 連携機能を使用する場合，TP1RpcClient クラスの Call メソッドを発行すると，内部で GetConnection メソッド，Execute メソッド，および Dispose メソッドが発行されます。しかし，トランザクション内の処理が完了するまでコネクションは使用中のままとなります。そのため，一つのトランザクション内で複数回 TP1RpcClient クラスの Call メソッドを発行した場合は，複数のコネクションが生成されます。

注意事項

WCF 連携機能を使用する場合，最大同時使用コネクション数（Connector .NET 構成定義の<connection>要素の active 属性で指定）には，一つのトランザクション内で発行する TP1RpcClient クラスの Call メソッドの数以上を設定してください。

(2)　トランザクションのタイムアウトの設定

System.Transactions では，トランザクションのタイムアウトを設定できます。次のどちらかの方法でトランザクションのタイムアウトを設定します。

なお，設定するトランザクションのタイムアウトの値には条件があります。条件については，「(c)　設定するトランザクションのタイムアウトの条件」を参照してください。

(a)　アプリケーションから設定する場合

次に示す引数にタイムアウトを指定します。

明示的なプログラミングモデルを使用してアプリケーションを開発した場合

System.Transactions.CommittableTransaction クラスのコンストラクタの引数

暗黙的なプログラミングモデルを使用してアプリケーションを開発した場合

System.Transactions.TransactionScope クラスのコンストラクタの引数

(b) コントロールパネルから設定する場合

コントロールパネルから設定する「トランザクション タイムアウト」の値は，「(a) アプリケーションから設定する場合」で設定しなかった場合のデフォルト値となります。

1. [コントロールパネル] − [管理ツール] − [コンポーネント サービス] を選択します。

   [コンポーネント サービス] ダイアログを表示されます。

2. [コンソールルート] から，[コンピュータ] − [マイコンピュータ] を選択し，プロパティを開きます。

3. [オプション] タブの「トランザクション タイムアウト」の値を変更します。

(c) 設定するトランザクションのタイムアウトの条件

トランザクションのタイムアウトを設定する場合，CUP.NET および OpenTP1 で設定する時間監視（問い合わせ間隔，メッセージの送受信など）の値を考慮して設定する必要があります。次の条件をすべて満たす値を設定してください。

- トランザクションのタイムアウト（t0）＞問い合わせ間隔（t1）
- トランザクションのタイムアウト（t0）＞メッセージの送受信（t2）
- トランザクションのタイムアウト（t0）＞トランザクションブランチの処理（t3）
- トランザクションのタイムアウト（t0）＜トランザクションブランチの処理完了（t6）
- トランザクションのタイムアウト（t0）＜トランザクションブランチの処理完了（t7）

トランザクションのタイムアウトの値が上記の条件を満たしていない場合，トランザクションの実行中に OpenTP1 のサービス要求でタイムアウトが発生するおそれがあります。

トランザクションのタイムアウトの時間経過によるタイムアウトが発生した場合は，トランザクション内の処理が終了する前にロールバックが実行されることがあります。ロールバック完了後にトランザクション範囲内の TP1Connection.Execute メソッドなどが呼び出された場合，TP1ConnectorException 例外が発生します。

CUP.NET および OpenTP1 で設定する時間監視の監視区間を次の図に示します。また，時間監視の種類と設定方法を表 2-5 に示します。なお，図の t0～t8 は，表 2-5 の監視区間と対応しています。

図 2-18　時間監視の監視区間

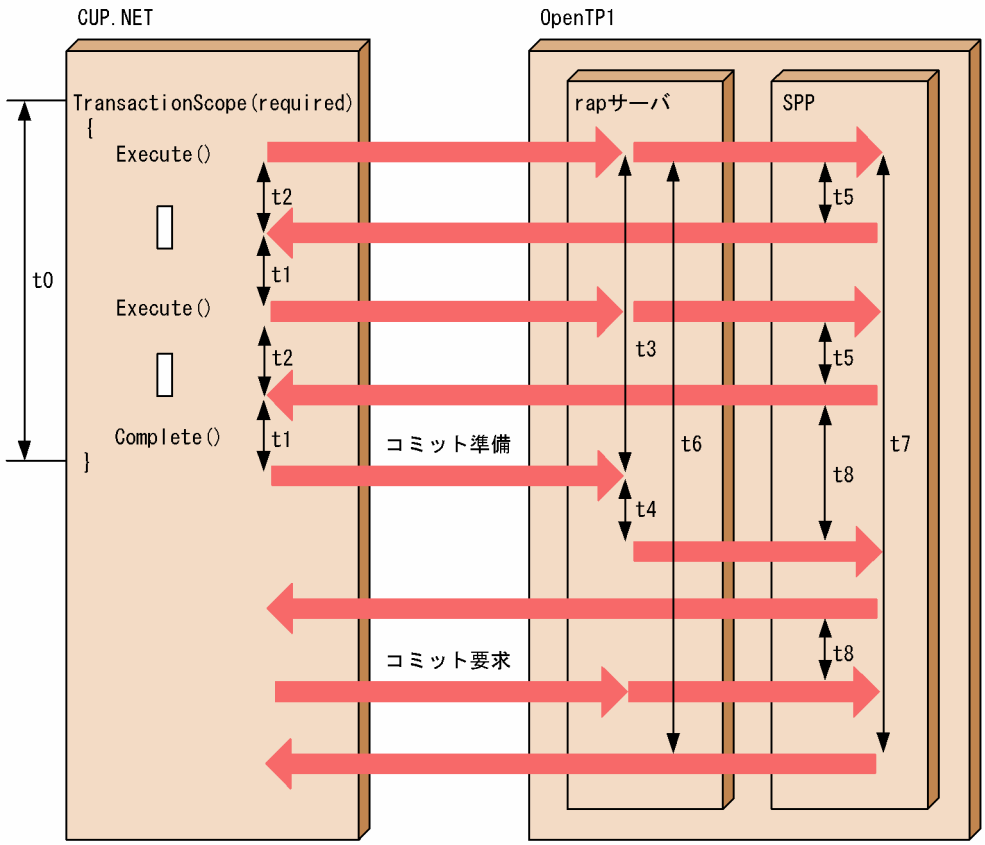


表 2-5　時間監視の種類と設定方法

監視区間	監視の種類	設定方法	
		CUP.NET での設定	OpenTP1 での設定
t0	トランザクションのタイムアウト	次のどちらかの方法で設定します。 **API で設定** TransactionScope クラスまたは CommittableTransaction クラスのコンストラクタの引数：TimeSpan オブジェクト **GUI で設定** ［コントロールパネル］から指定した［トランザクション タイムアウト］の値	−
t1	問い合わせ間隔	**Client .NET 構成定義** <rapService>要素の inquiredTime 属性	**rap リスナーサービス定義** rap_inquire_time オペランド
t2	メッセージの送受信	次のどちらかの方法で設定します。 **Client .NET 構成定義** <rpc>要素の watchTime 属性	**rap リスナーサービス定義** rap_watch_time オペランド

監視区間	監視の種類	設定方法	
		CUP.NET での設定	OpenTP1 での設定
		API で設定 RpcInfo クラスまたはクライアントスタブクラスの WatchTime プロパティ	
t3	トランザクションブランチの処理	Client .NET 構成定義 <xarTransaction>要素の expireTime 属性	rap リスナーサービス定義 trn_expiration_time オペランド
t4	アイドル状態のトランザクションブランチ	－	XA リソースサービス定義 xar_session_time オペランド
t5	トランザクションブランチの処理	－	ユーザサービス定義，またはトランザクションサービス定義 trn_expiration_time オペランド
	サービス関数の処理	－	ユーザサービス定義 service_expiration_time オペランド
t6	トランザクションブランチの処理完了	－	rap リスナーサービス定義，またはトランザクションサービス定義 trn_completion_limit_time オペランド
t7	トランザクションブランチの処理完了	－	ユーザサービス定義，またはトランザクションサービス定義 trn_completion_limit_time オペランド
t8	トランザクション同期点処理	－	ユーザサービス定義，またはトランザクションサービス定義 trn_watch_time オペランド

（凡例）

－：設定できません。

(3) アプリケーションを開発する場合の注意事項

MSDTC 連携機能を使用してアプリケーションを開発する場合は，次の点に注意してください。

- 構成定義の<distributedTransaction>要素の use 属性には true を指定してください。false を指定した場合，アプリケーションのトランザクション範囲内で OpenTP1 にサービスを要求してもトランザクションには含まれません。
- MSDTC 連携機能を使用するアプリケーションで，スケジューラダイレクト機能およびネームサービスを使用した RPC 要求をする場合は，必ずトランザクション範囲外で要求してください。トランザクション範囲内でスケジューラダイレクト機能およびネームサービスを使用した RPC 要求をした場合，TP1ConnectorException 例外が発生します。
- TCP/IP 通信機能を使用する場合は，トランザクション範囲内でのメッセージの送受信はできますが，トランザクションには含まれません。

2.2.14　ノード識別子の決定

アプリケーションを配置する OS 環境を一意に識別するためのノード識別子を決定します。ノード識別子を決定するためには，「管理者として実行」で起動したコマンドプロンプトで〈Connector .NET のインストールディレクトリ〉¥bin ディレクトリをパスに追加して，MSDTC 連携機能で使用するノード識別子生成コマンド（cnnnidgen）を実行します。運用コマンドの詳細については，「5.　運用コマンド」の「cnnnidgen（MSDTC 連携機能で使用するノード識別子生成コマンド）」を参照してください。

ノード識別子を決定したあと，MSDTC 連携機能を使用するアプリケーションおよびトランザクションリカバリサービスの構成定義を指定してトランザクションリカバリサービスを開始します。

2.2.15　MSDTC 連携機能を使用する場合の構成定義での指定

MSDTC 連携機能を使用する場合，アプリケーションとトランザクションリカバリサービスの両方の構成定義を指定する必要があります。構成定義の詳細については，「3.　構成定義」の「distributedTransaction」および「recoveryService」を参照してください。また，Client .NET 構成定義については，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。

(1)　アプリケーションの設定

アプリケーション構成ファイル（Web.config，<アプリケーションの実行ファイル名>.config など）に，次に示す定義を追加してください。

- MSDTC 連携機能使用の有無

  <distributedTransaction>要素の use 属性に true を指定してください。

- ノード識別子の指定

  <distributedTransaction>要素の nodeId 属性に指定するノード識別子は，Connector .NET で MSDTC 連携機能を使用している OS 環境を一意に識別するための ID です。

  <distributedTransaction>要素の nodeId 属性の指定値は，トランザクションリカバリサービスの構成定義の<recoveryService>要素の nodeId 属性と一致させる必要があります。この値が一致していない場合，トランザクション開始時にアプリケーションでエラーが発生します。すでにトランザクションリカバリサービスでノード識別子を指定している場合は，その値を使用して指定してください。ノード識別子を新規に作成する場合は，MSDTC 連携機能で使用するノード識別子生成コマンド（cnnnidgen）を実行してください。

- 単一フェーズコミット最適化使用の有無

  単一フェーズコミット最適化を行う場合は，<distributedTransaction>要素の optimize1PC 属性に true を指定してください。

- 未決着トランザクションの回復間隔の指定

  <distributedTransaction>要素の recoverRetryInterval 属性に，アプリケーションのバックグラウンドスレッドから未決着トランザクションの回復処理を行う間隔を指定してください。

- 窓口となる OpenTP1 のホスト名およびポート番号の指定

  Client .NET 構成定義の<tp1Server>要素の host 属性および port 属性に，窓口となる OpenTP1 のホスト名およびポート番号を指定してください。MSDTC 連携機能を使用するクライアントアプリケーションに指定できる OpenTP1 のホスト名およびポート番号は，プロファイル ID 一つにつき一つずつです。

- ログレベルの指定

  アプリケーションが，トランザクションリカバリサービスに未決着トランザクションが存在する旨を通知した場合などに情報を出力したいときは，<log>要素の level 属性に 2（ログレベル 2）を指定してください。トランザクションごとにコミットやロールバックが完了したなどの詳しい情報を出力したい場合は，<log>要素の level 属性に 3（ログレベル 3）を指定してください。

- 最大同時使用コネクション数

  WCF 連携機能を使用する場合は，<connection>要素の active 属性に，一つのトランザクション内で発行する TP1RpcClient クラスの Call メソッドの数以上を指定してください。

(2) トランザクションリカバリサービスの設定

〈Connector .NET のインストールディレクトリ〉¥trs 下に，トランザクションリカバリサービスのアプリケーション構成ファイル（cnntrs.exe.config）を作成してください。

Connector .NET では，トランザクションリカバリサービスの構成定義のサンプルを〈Connector .NET のインストールディレクトリ〉¥examples¥config 下に提供しています。必要に応じてコピーして使用してください。

- ノード識別子の指定

  <recoveryService>要素の nodeId 属性に指定するノード識別子は，Connector .NET で MSDTC 連携機能を使用している OS 環境を一意に識別するための ID です。

  ノード識別子を作成する場合は，MSDTC 連携機能で使用するノード識別子生成コマンド（cnnnidgen）を実行してください。

- 未決着トランザクションの回復間隔の指定

  トランザクションリカバリサービスから未決着トランザクションの回復処理を行う間隔を<recoveryService>要素の recoverRetryInterval 属性に指定してください。

- アプリケーションドメインの監視間隔の指定

  アプリケーションドメインの終了を監視する間隔を<recoveryService>要素の appDomainCheckInterval 属性に指定してください。

- 未決着トランザクションの確認間隔の指定

  トランザクション回復処理の完了後，未決着トランザクションが存在しないかを確認する間隔を<recoveryService>要素の recoverCheckInterval 属性に指定してください。

- 未決着トランザクションの確認回数の指定

  トランザクション回復処理の完了後，未決着トランザクションが存在しないかを確認する回数を<recoveryService>要素の recoverCheckCount 属性に指定してください。

- RMID 格納ディレクトリの指定

  トランザクションリカバリサービスが未決着トランザクションの回復処理に必要な情報を保存する RMID 格納ディレクトリのパスを，<recoveryService>要素の rmidStoragePath 属性に指定してください。なお，RMID 格納ディレクトリはあらかじめ作成しておいてください。

- プロファイル ID の指定

  トランザクションリカバリサービスの設定で Client .NET 構成定義の<tp1Server>要素の host 属性および port 属性に指定するプロファイル ID を，<recoveryService>要素の profiles 属性に指定してください。プロファイル ID は，必ずコンマ（,）で区切って指定してください。

- 窓口となる OpenTP1 のホスト名およびポート番号の指定

  接続する OpenTP1 が一つの場合，接続する OpenTP1 は<common>要素内に指定してください。この場合，トランザクションリカバリサービスの設定で<recoveryService>要素の profiles 属性を指定する必要はありません。

  接続する OpenTP1 が複数ある場合，MSDTC 連携機能を使用するアプリケーションが接続するすべての OpenTP1 のホスト名およびポート番号を，Client .NET 構成定義の<tp1Server>要素の host 属性および port 属性に指定してください。指定できる OpenTP1 のホスト名およびポート番号は，プロファイル ID 一つにつき一つずつです。なお，アプリケーションのプロファイル ID にはコンマ（,）を使用できますが，トランザクションリカバリサービスの構成定義に指定するプロファイル ID ではコンマ（,）は使用しないでください。

- コネクションに関する定義の指定

  - 最大同時使用コネクション数

    トランザクションリカバリサービスの設定の，窓口となる OpenTP1 のホスト名およびポート番号の指定で指定したプロファイル ID の個数を<connection>要素の active 属性に指定してください。

  - Connector .NET が管理するコネクションプールの最大数

    トランザクションリカバリサービスの設定の，窓口となる OpenTP1 のホスト名およびポート番号の指定で指定したプロファイル ID の個数を<connection>要素の pooled 属性に指定してください。

  - プロファイルが占有できるコネクションプールの最大数

    プロファイルが占有できるコネクションプールの最大数を，<occupation>要素の pooled 属性に指定してください。プロファイルが占有できるコネクションプールの最大数は，プロファイル ID 一つにつき一つずつ指定してください。

  特に必要がない場合は，これら以上のコネクション数を指定しないでください。これら以上のコネクション数を指定すると，トランザクションリカバリサービスから接続先 rap サーバへの常設コネクション数が増加することによって，アプリケーションで接続できる rap サーバの数が減少することがあります。

- ログレベルの指定

  トランザクションリカバリサービスの設定では，<log>要素の level 属性に 2（ログレベル 2）を指定してください。

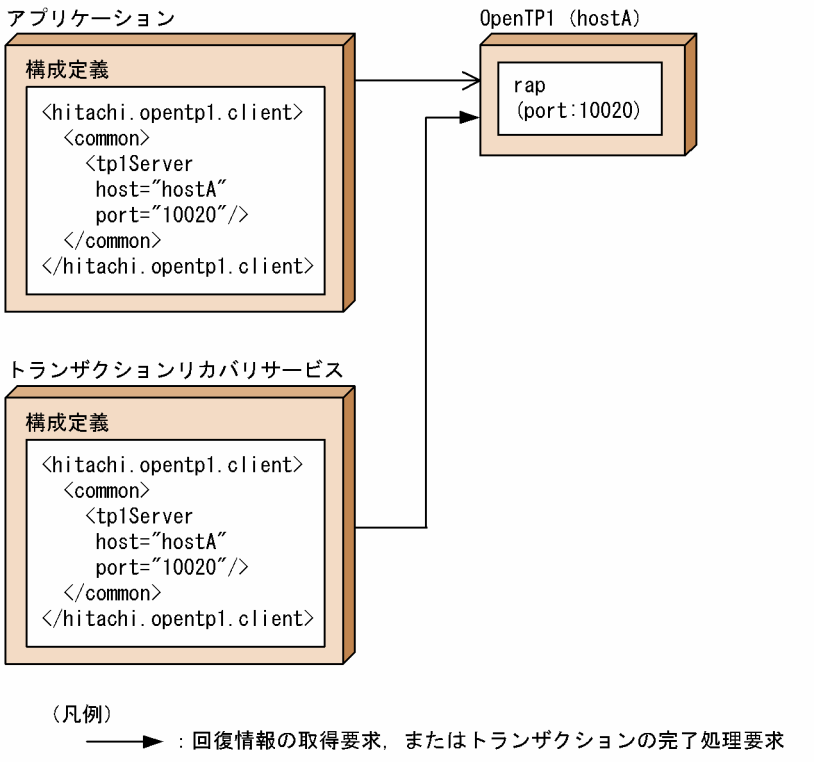
- Client .NET のエラートレースの取得

Client .NET 構成定義の<errTrace>要素を指定して，Client .NET のエラートレースを必ず取得してください。

(3) アプリケーションおよびトランザクションリカバリサービスの設定

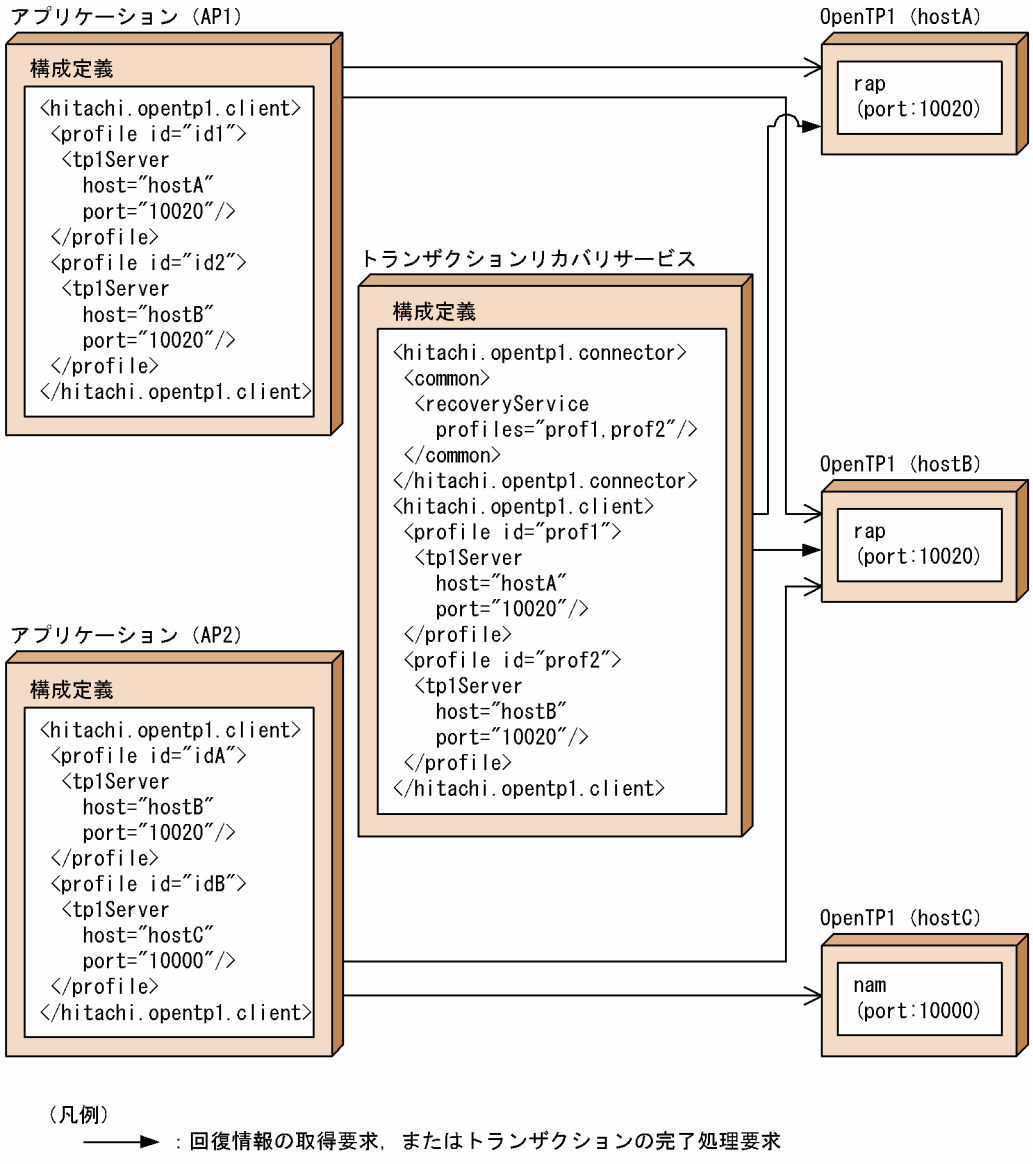
接続する OpenTP1 が一つの場合の設定例を次の図に示します。

図 2-19 アプリケーションおよびトランザクションリカバリサービスの設定例（接続する OpenTP1 が一つの場合）



接続する OpenTP1 が複数ある場合で，hostA および hostB で MSDTC 連携機能を使用する場合の設定例を次の図に示します。

図 2-20　アプリケーションおよびトランザクションリカバリサービスの設定例（接続する OpenTP1 が複数ある場合）



トランザクションリカバリサービスでは，プロファイル ID，および接続する OpenTP1（hostA および hostB）を指定します。このとき，hostC では MSDTC 連携機能を使用しないため，hostC のプロファイル ID の指定は不要です。

アプリケーションでは，プロファイル ID，および接続する OpenTP1（hostA，hostB，および hostC）を指定します。MSDTC 連携機能を使用する場合，接続する OpenTP1 はトランザクションリカバリサービスで指定した OpenTP1 を指定してください。AP1 では，接続する OpenTP1 に hostA および hostB を指定します。AP2 では，接続する OpenTP1 に hostB および hostC を指定します。

2.2.16　トランザクションリカバリサービスの開始

トランザクションリカバリサービスを開始します。トランザクションリカバリサービスのプロパティで「スタートアップの種類」が「自動」に設定されている場合，トランザクションリカバリサービスを開始する必要はありません。

トランザクションリカバリサービスを開始する方法について説明します。

［コントロールパネル］から開始する場合

［コントロールパネル］－［管理ツール］－［サービス］画面で「TP1/Connector.NET Transaction Recovery Service」をダブルクリックするとプロパティのウィンドウが表示されます。［全般］タブの［サービスの状態：］にある［開始(S)］ボタンをクリックしてください。［サービスの状態：開始］となったことを確認したら［OK］ボタンをクリックし，プロパティのウィンドウを閉じます。

コマンドプロンプトから開始する場合

「管理者として実行」で起動したコマンドプロンプトで「net start TP1ConnectorNETTRS」を実行します。

トランザクションリカバリサービスは，アプリケーションを実行する前に起動する必要があります。トランザクションリカバリサービスが起動していない状態でアプリケーションを実行すると，トランザクション内での最初のサービス要求時にエラーが発生します。

2.2.17　MSDTC 連携機能を使用している場合の障害発生時の運用

MSDTC 連携機能を使用している場合に障害が発生したときは，障害が回復されるとトランザクションリカバリサービスがすべてのトランザクションを自動的に回復します。しかし，次に示す条件を両方とも満たす場合は，MSDTC が管理するトランザクションと OpenTP1 が管理するトランザクションを強制決着させてください。

- コミットの準備が完了した状態で障害が発生して，長時間に渡って復旧できないなどの状態
- トランザクションを強制的にコミットまたはロールバックして MSDTC や OpenTP1 が使用するリソースを解放する必要がある状態

(1)　トランザクションを強制決着させる方法

MSDTC が管理するトランザクションと OpenTP1 が管理するトランザクションを強制決着させる方法について説明します。

(a)　MSDTC が管理するトランザクションを強制決着させる方法

dcomcnfg コマンドを実行して［コンポーネント サービス］を起動し，［コンポーネント サービス］－［コンピュータ］－［マイコンピュータ］－［分散トランザクション コーディネータ］－［トランザクションの一覧］画面で行います。詳細については，Microsoft 管理コンソールのヘルプを参照してください。

(b) OpenTP1 が管理するトランザクションを強制決着させる方法

TP1/Server Base のトランザクションを強制決着させる場合は，xarforce コマンドを使用します。詳細については，マニュアル「OpenTP1 運用と操作」を参照してください。

TP1/LiNK でトランザクションを強制決着させる場合は，［TP1/LiNK XA リソースサービス管理］ダイアログボックスで行います。詳細については，マニュアル「TP1/LiNK 使用の手引」を参照してください。

(2) トランザクションを強制決着させる際の運用方法

単一フェーズコミット最適化を使用していない場合は，次に示す ID を照合して，各トランザクションのステータスに応じてトランザクションを強制決着させてください。

- MSDTC が管理するトランザクションで表示される［トランザクションの一覧］画面の［作業 ID ユニット］に表示される ID
- TP1/Server Base の xarls コマンドで表示される DID，または TP1/LiNK の［TP1/LiNK XA リソースサービス管理］ダイアログボックスに表示される DID

次に示す場合は，あらかじめ決めておいた運用手順に従ってトランザクションを強制決着させてください。ただしこの場合，MSDTC が管理するトランザクションと OpenTP1 が管理するトランザクションの整合性は保証されないおそれがあります。

- MSDTC または OpenTP1 に障害が発生したことが原因となって ID の照合ができない場合
- 単一フェーズコミット最適化を使用している場合※

注※

TP1/Server Base の場合は xarls コマンドの実行，TP1/LiNK の場合は［TP1/LiNK XA リソースサービス管理］ダイアログボックスで，DID を表示できないことがあります。このため，ID を照合して，各トランザクションのステータスに応じてトランザクションを強制決着させることができません。

2.3 TCP/IP 通信機能

Connector .NET は，Client .NET を通して TCP/IP 通信機能を使用できます。TCP/IP 通信機能を使用すると，TCP/IP プロトコルによって OpenTP1 の MHP や OpenTP1 以外の他システムとメッセージの送受信ができます。

Connector .NET を使用した TCP/IP 通信では，コネクションプーリング機能およびバッファプーリング機能が利用できます。TCP/IP 通信の要求インタフェースには，インデクスドレコードを使用します。

Connector .NET を使用した TCP/IP 通信機能は，Connector .NET 側から OpenTP1 の MHP や他システムに対して TCP/IP の物理コネクションを確立する場合にだけ利用できます。OpenTP1 の MHP や他システムからのコネクション確立要求を受け付ける形態では利用できません。

2.3.1 通信形態

TCP/IP 通信機能を使用したメッセージの送受信には，次の 3 種類があります。

- メッセージの一方送信
- メッセージの同期送受信
- メッセージの一方受信

(1) メッセージの一方送信

OpenTP1 の MHP や他システムに対してデータを一方送信する形態です。

この通信形態を使用する場合は，TcpipInfo クラスの Flags プロパティに TCPIP_SEND フラグを指定してください。

(2) メッセージの同期送受信

OpenTP1 の MHP や他システムとデータを同期送受信する形態です。

この通信形態を使用する場合は，TcpipInfo クラスの Flags プロパティに TCPIP_SENDRECV フラグを指定してください。

(3) メッセージの一方受信

一方送信または同期送受信で確立した OpenTP1 の MHP や他システムとの物理コネクションを使用して，データを一方受信する形態です。受信用ソケットを開設する一方受信はサポートされません。

この通信形態を使用する場合は，TcpipInfo クラスの Flags プロパティに TCPIP_RECV フラグを指定していて，かつ事前に一方送信または同期送受信の通信形態で物理コネクションが確立されている必要があります。

2.3.2　TCP/IP 通信機能を使用する場合に関連する定義

TCP/IP 通信機能を使用する場合，Connector .NET のプロファイルに指定する Client .NET 構成定義で，<tcpip>要素を次のように指定しておく必要があります。

表 2-6　TCP/IP 通信機能を使用する場合の<tcpip>要素の指定値

属性	指定値
use 属性	true※
type 属性	sendrecv※
sendHost 属性	接続先ホスト名
sendPort 属性	接続先ポート番号
openPortAtRecv 属性	true※

注※

これらの値以外を指定した場合の動作は保証しません。

HostA の 20000 番ポートへ接続する場合の指定例を示します。

【指定例】

```
…
<tcpip use="true"
       type="sendrecv"
       sendHost="HostA"
       sendPort="20000"
       openPortAtRecv="true" />
…
```

なお，接続先が複数ある場合は，接続先数分のプロファイルが必要です。そのため，Client .NET 構成定義のプロファイルを複数作成して，それぞれのプロファイルを使用する Connector .NET 構成定義のプロファイルを作成してください。

2.3.3　物理コネクションの管理

物理コネクションは，次に示す規則で管理されます。

- TcpipConnection オブジェクト取得時に，コネクションプールから取得されないで新しくコネクションが生成された場合，その時点では物理コネクションは確立されません。
- TcpipConnection クラスの Execute メソッドが発行された時点で物理コネクションが確立されていない場合は，物理コネクションが確立されたあとに通信が行われ，通信後も物理コネクションは維持されます。
- TcpipConnection クラスの Disconnect メソッドが発行された時点で，接続先との物理コネクションが解放されます。

- TcpipConnection オブジェクトがコネクションプールに戻されるときは，Connector .NET 構成定義の<tcpip>要素で keepAlive 属性に指定した値によって，物理コネクションを維持するかどうかが決まります。
  - true を指定したとき
    物理コネクションを維持したまま，コネクションプールに戻されます。
  - false を指定したとき
    すべての接続先との物理コネクションが解放されたあとで，コネクションプールに戻されます。
- TcpipConnection クラスの Dispose メソッド発行時に，TcpipConnection オブジェクトがコネクションプールに戻されないで破棄される場合は，Connector .NET 構成定義の<tcpip>要素で keepAlive 属性に指定した値に関係なく，接続先との物理コネクションが解放されます。

2.3.4 TCP/IP 通信機能を使用する場合の注意事項

Connector .NET で TCP/IP 通信機能を使用する場合の注意事項を次に示します。なお，実際の通信は TP1Client オブジェクトが行うため，マニュアル「TP1/Client for .NET Framework 使用の手引」の TCP/IP 通信機能の注意事項もあわせて参照してください。

- TP1ConnectionManager クラスの GetTcpipConnection メソッドで TcpipConnection オブジェクトを取得した直後は，物理コネクションの状態が不明なため，一方受信での通信はしないでください。
- 物理コネクションが確立されていない状態や切断された状態で一方受信での通信をした場合は，予期しないで受信ソケットが開設されてしまいます。そのため，一方受信の通信で例外が発生した場合は，必ず TcpipConnection クラスの Disconnect メソッドを発行してください。
- 物理コネクションを維持した状態でコネクションをコネクションプールに戻す運用をする場合は，プーリングされたコネクション数分の物理コネクションが長時間維持されます。このような物理コネクションの解放は，接続先システム側で監視します。接続先システムが TP1/NET/TCP/IP の場合は，無通信状態監視機能を使用してください。無通信状態監視については，マニュアル「OpenTP1 プロトコル TP1/NET/TCP/IP 編」を参照してください。

  長時間維持された物理コネクションを Connector .NET 側で解放するためには，Connector .NET のアセンブリをロードしているアプリケーションドメインを解放する必要があります。このため，ASP.NET でサービスを利用している場合は，IIS を停止する，ワーカープロセスのリサイクル機能によって定期的にプロセスを停止するなどの運用が必要です。

  接続先システム側での監視および解放，ならびにワーカープロセスのリサイクルができない場合は，コネクションをコネクションプールに戻す前に必ず物理コネクションを解放する運用にしてください。
- コネクションプール，および Connector .NET の内部で保持される TP1Client オブジェクトは，TP1Connection オブジェクトと TcpipConnection オブジェクトが共有します。このため，RPC で使用する構成定義プロファイルと TCP/IP 通信で使用する構成定義プロファイルを，別々に定義してください。同じ構成定義プロファイルを使用した場合，TcpipConnection クラスの Disconnect メソッドの発行によって常設コネクションが解放されてしまうなど，相互の動作に影響する場合があります。

- コネクションプール数や最大同時接続数を見積もる場合は，RPC で利用する TP1Connection オブジェクトのコネクションプール数や最大同時接続数と，TCP/IP 通信で利用する TcpipConnection オブジェクトのコネクションプール数や最大同時接続数を合わせた値で見積もってください。RPC と TCP/IP 通信とで別々にコネクションプール占有数を制限したい場合は，それぞれの構成定義プロファイルで<occupation>要素の pooled 属性を指定してください。
- バッファプーリング機能を使用して TCP/IP 通信を行う場合，TcpipConnection クラスの Execute メソッドを発行して送受信できるデータ長は，1 回のメソッド発行につき最大 1 メガバイトになります。バッファプーリング機能を使用しない場合は，データ長に制限はありません。

2.4　コネクションプーリング機能

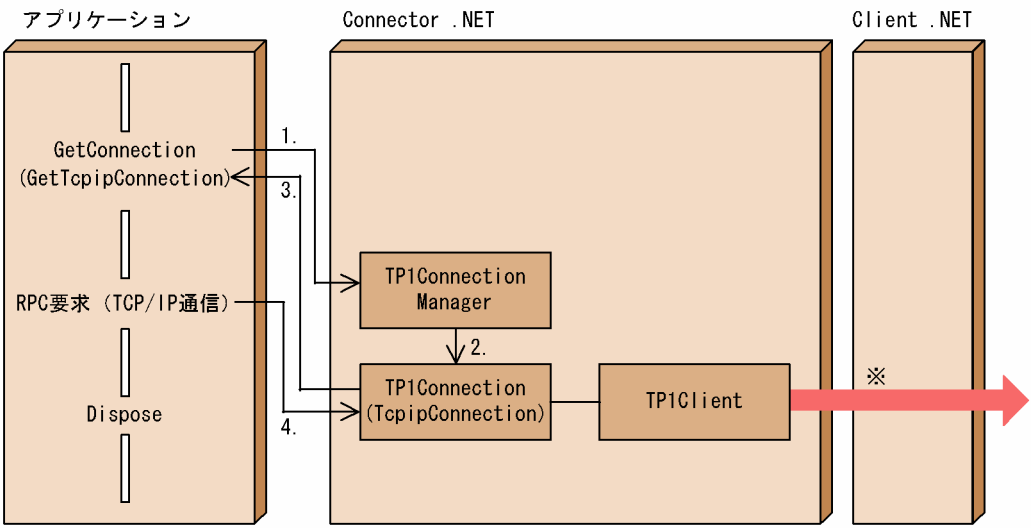
2.4.1　コネクションの生成とプーリング

Connector .NET は，Client .NET が提供する TP1Client クラスのインスタンスにハンドルを関連づけて管理します。TP1Client クラスのインスタンスに関連づけられたハンドルのことを，**コネクション**と呼びます。

(1)　コネクションの生成

コネクションの生成の概要を，次の図に示します。

図 2-21　コネクションの生成



注※

実際のコネクション制御は，RPC の通信形態や TCP/IP 通信時の構成定義によって異なります。

1. Connector .NET のアプリケーションから TP1ConnectionManager クラスの GetConnection メソッド（または GetTcpipConnection メソッド）が呼び出されると，TP1Client クラスのインスタンスが生成されます。

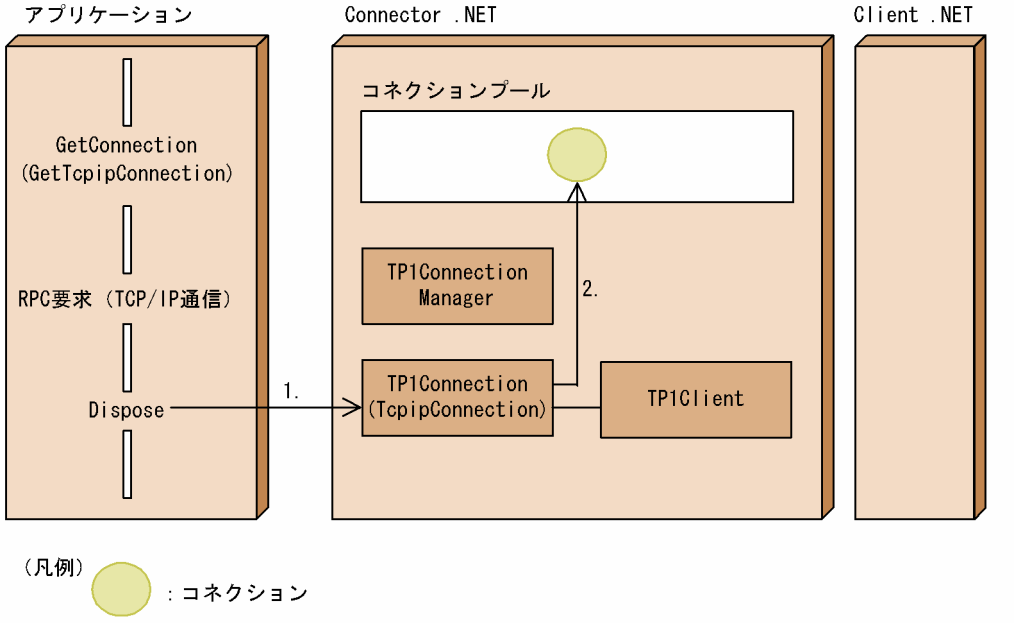
   このとき，アプリケーションのプロファイル ID に指定された Client .NET 構成定義に基づいて，TP1Client クラスの OpenRpc メソッドが呼び出され，RPC 環境が初期化されます。
2. TP1ConnectorManager クラスは，コネクション（TP1Connection オブジェクトまたは TcpipConnection オブジェクト）を生成します。
3. GetConnection メソッド（または GetTcpipConnection メソッド）にコネクションを返します。
4. アプリケーションは，コネクションに対して RPC 要求（または TCP/IP 通信）を実行します。

(2)　コネクションのプーリング

Connector .NET では，資源を効率的に使用するために，一度生成されたコネクションをプーリングできます。この領域を，**コネクションプール**と呼びます。コネクションプールは，最初にアプリケーションから GetConnection メソッド（または GetTcpipConnection メソッド）が呼び出されたときにアプリケーションドメインごとに生成され，コネクションが解放されると TP1Client クラスのインスタンスがプーリングされます。

コネクションプーリング機能の概要を，次の図に示します。

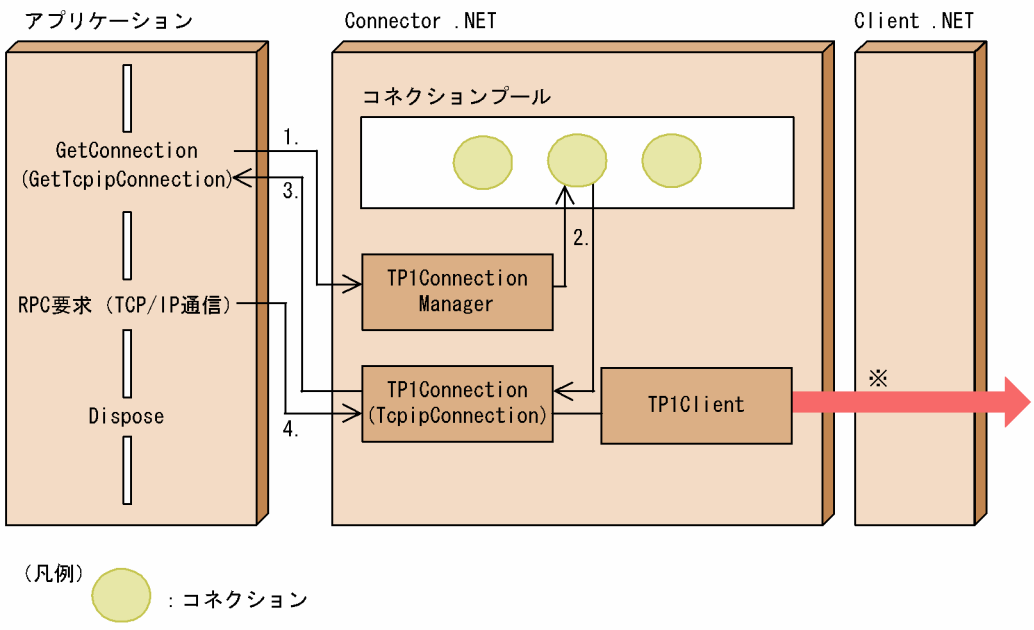
図 2-22　コネクションのプーリング



1. アプリケーションから TP1Connection クラスの Dispose メソッドが呼び出されると，使用していたコネクションが解放されます。
2. 使用していたコネクションは，コネクションプールにプーリングされます。

GetConnection メソッド（または GetTcpipConnection メソッド）が呼び出された時に，コネクションプール内に同じプロファイル ID（Client .NET 構成定義のプロファイル ID）のコネクションが存在する場合は，新たに TP1Client クラスのインスタンスを生成しないで，コネクションプール内のコネクションを取得します。

図 2-23　コネクションプールからのコネクションの取得



注※

実際のコネクション制御は，RPC の通信形態や TCP/IP 通信時の構成定義によって異なります。

1. アプリケーションから TP1ConnectionManager クラスの GetConnection メソッド（または GetTcpipConnection メソッド）が呼び出されます。
   このとき，アプリケーションのプロファイル ID で指定された Client .NET 構成定義に基づいて，TP1Client クラスの OpenRpc メソッドが呼び出され，RPC 環境が初期化されます。
2. コネクションプール内に同じプロファイル ID のコネクション（TP1Connection オブジェクトまたは TcpipConnection オブジェクト）が存在するため，TP1ConnectorManager クラスはコネクションプールからコネクションを取得します。
3. GetConnection メソッド（または GetTcpipConnection メソッド）にコネクションを返します。
4. アプリケーションは，コネクションに対して RPC 要求（または TCP/IP 通信）を実行します。

2.4.2　構成定義での指定

構成定義で，次の値を指定します。

- 最大同時使用コネクション数
- 最大コネクションプール数
- プロファイル単位の最大占有コネクション数

なお，ASP.NET では Web アプリケーションごとにアプリケーションドメインが生成されるため，最大コネクションプール数，最大同時使用コネクション数，およびプロファイル単位の最大占有コネクション数は，アプリケーションごとに見積もってください。

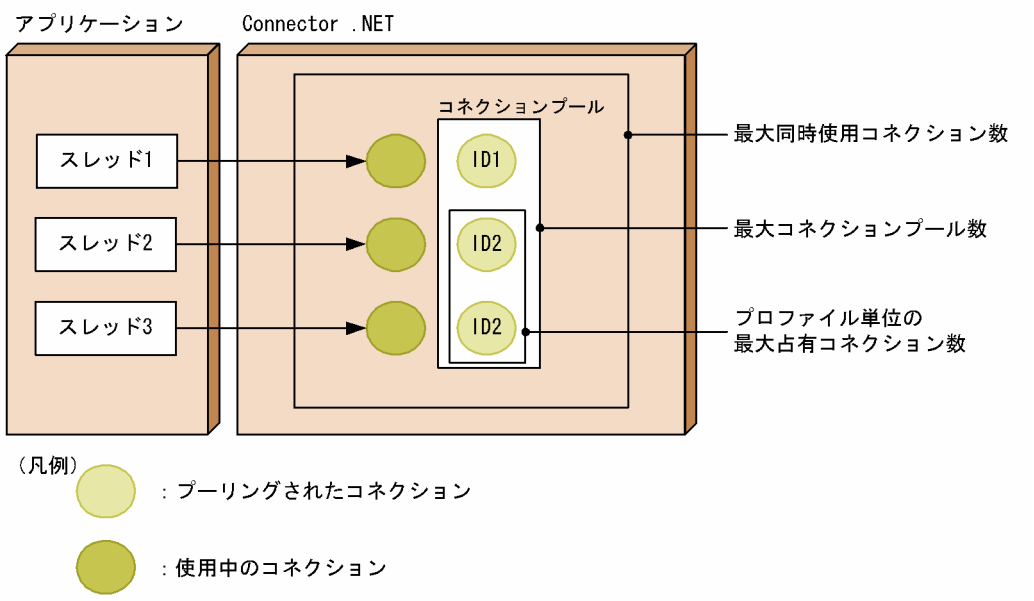
OpenTP1 のリモート API 機能でスタティックコネクションスケジュールモードを使用する場合，全ノードの全アプリケーションで設定する最大同時コネクション数（connection 要素の active 属性）の合計が rap サーバの数以下となるようにしてください。コネクションの合計数が rap サーバの数より多い場合は，サービス要求時に待ち状態となるときがあります。

リモート API 機能のダイナミックコネクションスケジュールモードを使用する場合は，最大同時コネクション数の指定には特に制限はありません。

リモート API 機能の詳細については，マニュアル「OpenTP1 解説」を参照してください。

構成定義で指定する項目を，次の図に示します。

図 2-24　構成定義の指定（コネクションプーリング機能）



コネクションの総数が最大同時使用コネクション数の指定値を超えた場合，TP1Connection クラスの Dispose メソッドが呼び出されて，ほかのコネクションが解放されるまでは，新たにコネクションを生成できません。また，最大コネクションプール数の指定値を超えてコネクションがプーリングされたときは，TP1Client クラスの CloseRpc メソッドが呼び出され，古い TP1Client クラスのインスタンスから破棄されます。

構成定義の詳細については，「3.　構成定義」の「connection」および「occupation」を参照してください。

2.4.3　コネクションプーリング機能を使用したアプリケーションの実行

コネクションプーリング機能を使用したアプリケーションの実行例を示します。なお，この実行例で使用されているコネクションのプロファイル ID は次のとおりです。

プロファイル ID1

窓口となる OpenTP1 のホスト名：hostA

窓口となる OpenTP1 のポート番号：10020

RPC の通信形態：rap サービス

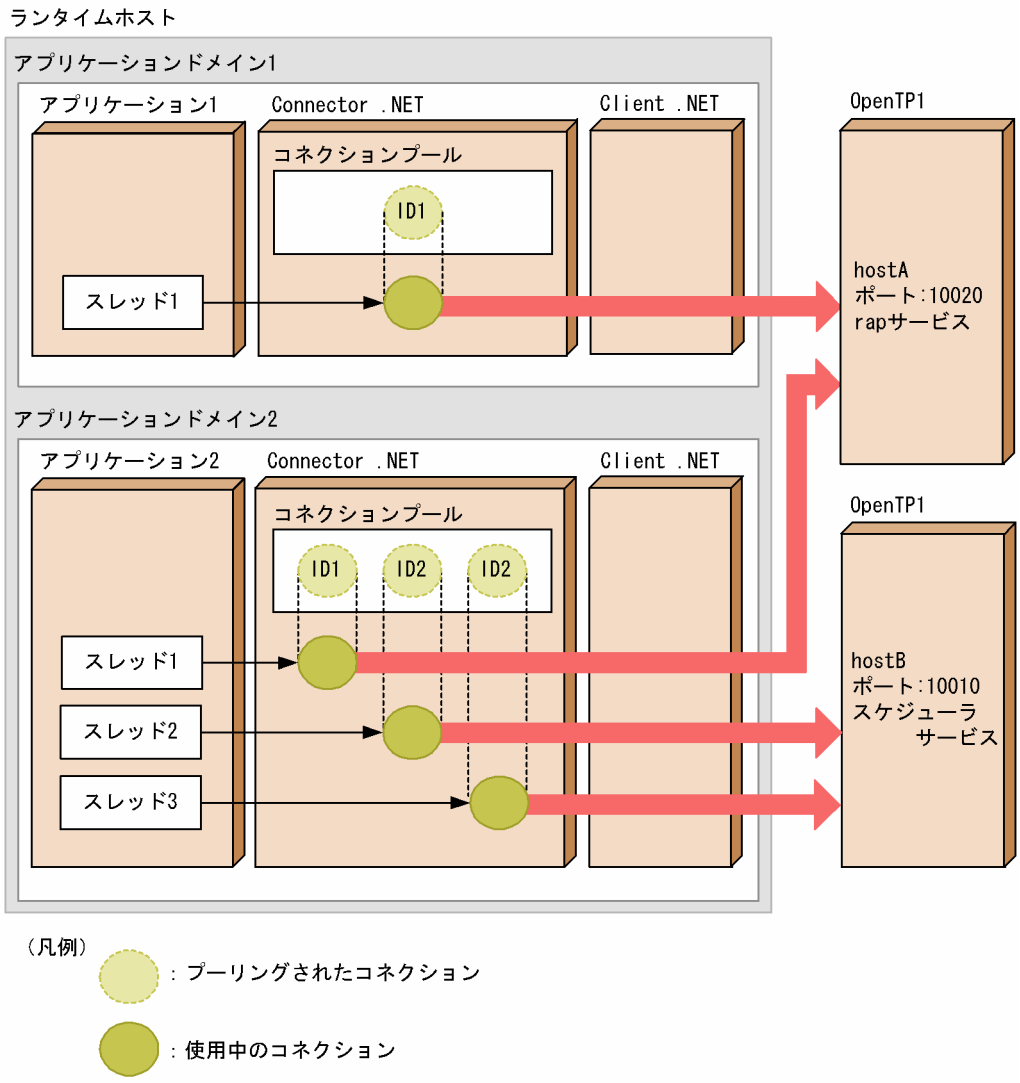
プロファイル ID2

窓口となる OpenTP1 のホスト名：hostB

窓口となる OpenTP1 のポート番号：10010

RPC の通信形態：スケジューラサービス

図 2-25　コネクションプーリング機能を使用したアプリケーションの実行例



注意事項

コネクションプールは必ずアプリケーションドメイン単位に生成されます。プロセス単位で生成されることはありません。

2.5 バッファプーリング機能

バッファプーリング機能は，RPC 要求および TCP/IP 通信に使用するメッセージを格納するバッファ（バイト配列）をプーリングする機能です。

RPC 要求や TCP/IP 通信を頻繁に実行するアプリケーションでは，サイズの大きなバッファの生成および破棄が繰り返されて，ガベージコレクションによる性能への影響が大きくなります。そのような場合に，バッファプーリング機能でバッファをプーリングすると，バッファの生成や破棄を少なくできます。これによってガベージコレクションの頻度が下がり，性能が向上します。

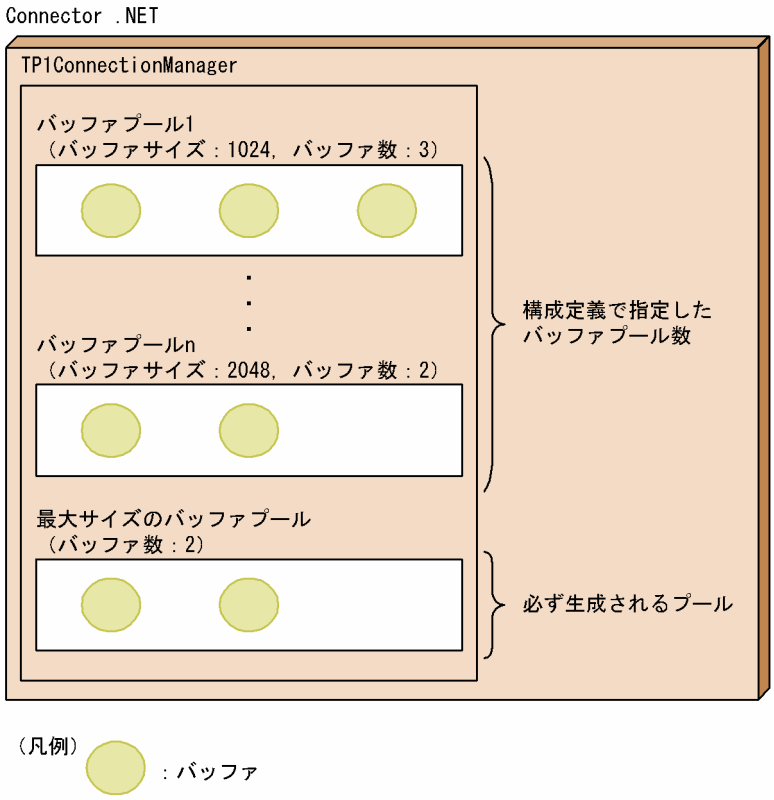
2.5.1 バッファプールとメッセージバッファ

(1) バッファプール

バッファプーリング機能を使用すると，生成されたバッファは，同じバッファサイズごとにプーリングされます。同じバッファサイズのバッファを管理するプールのことを，**バッファプール**と呼びます。なお，バッファプーリング機能を使用するとき，TP1ConnectionManager クラスは，最大サイズ（1048576 バイト）のバッファプールを必ず自動的に生成します。ただし，RPC 送受信メッセージの最大長拡張機能を使用する場合，最大サイズには<option>要素の maxMessageSize 属性で指定した値（×1048576 バイト）が設定されます。

バッファプール管理の構成例を，次の図に示します。

図 2-26　バッファプール管理の構成例



(2)　メッセージバッファ

RPC 要求および TCP/IP 通信にインデクスドレコードを使用するアプリケーションでは，バッファプールから取得したバッファを TP1ConnectionManager クラスが生成する MessageBuffer インスタンス（**メッセージバッファ**）に保持します。メッセージバッファは，インデクスドレコードのメッセージ用のレコード要素として使用します。

なお，RPC 要求にクライアントスタブを使用する場合は，Connector .NET の内部で TP1ConnectionManager クラスが生成するバッファプールを使用するため，アプリケーションが意識することなくバッファプーリング機能を利用できます。

2.5.2　構成定義での指定

構成定義で，次の値を指定します。なお，構成定義は RPC インタフェースごとに指定してください。

各プロファイルで異なるバッファプールを指定した場合は，バッファプールはプロファイルごとに生成されます。

- バッファプーリング機能の使用有無
  デフォルトでは，バッファプーリング機能は使用しません。
- プーリングされるバッファのバッファサイズと，格納されるバッファ数

バッファプールごとに設定します。RPC 要求を頻繁に実行するメッセージのメッセージ長やメッセージの使用頻度に合わせて指定することで，プーリングしてあるバッファを有効に使用できます。

- 最大サイズのバッファプールに格納されるバッファ数

ASP.NET では Web アプリケーションごとにアプリケーションドメインが生成されるため，プーリングされるバッファのバッファサイズと格納されるバッファ数，最大サイズのバッファプールに格納されるバッファ数はアプリケーションごとに見積もってください。

構成定義の詳細については，「3. 構成定義」の「buffer」，「largestBufferPool」，および「bufferPool」を参照してください。

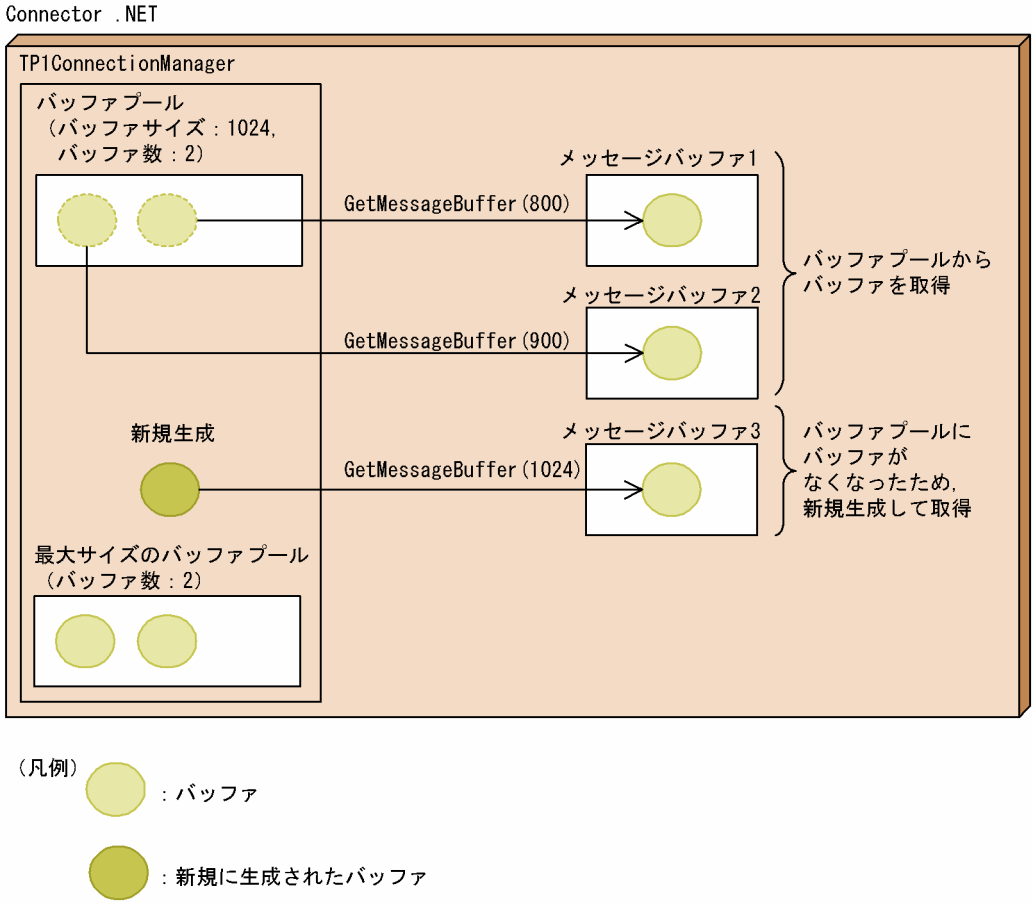
2.5.3 バッファの取得と解放

(1) バッファの取得

TP1ConnectionManager クラスは，GetMessageBuffer メソッドのパラメタに指定されたメッセージ長を保持できるメッセージバッファを生成します。

バッファ取得時のバッファプールの構造を，次の図に示します。

図 2-27 バッファ取得時のバッファプールの構造



メッセージバッファは，TP1ConnectionManager クラスがプール管理するバッファを内部に取得します。取得するバッファは，GetMessageBuffer メソッドのパラメタに指定されたメッセージ長を保持できる，最も小さいサイズのバッファです。

指定されたメッセージ長を保持できる，最も小さいサイズのバッファがバッファプール内に存在しない場合は，バッファを新規生成してメッセージバッファを生成します。この場合，バッファプールのバッファが解放されるまでは，GetMessageBuffer メソッドが呼び出されるたびにバッファを新規生成し，メッセージバッファを生成します。

注意事項

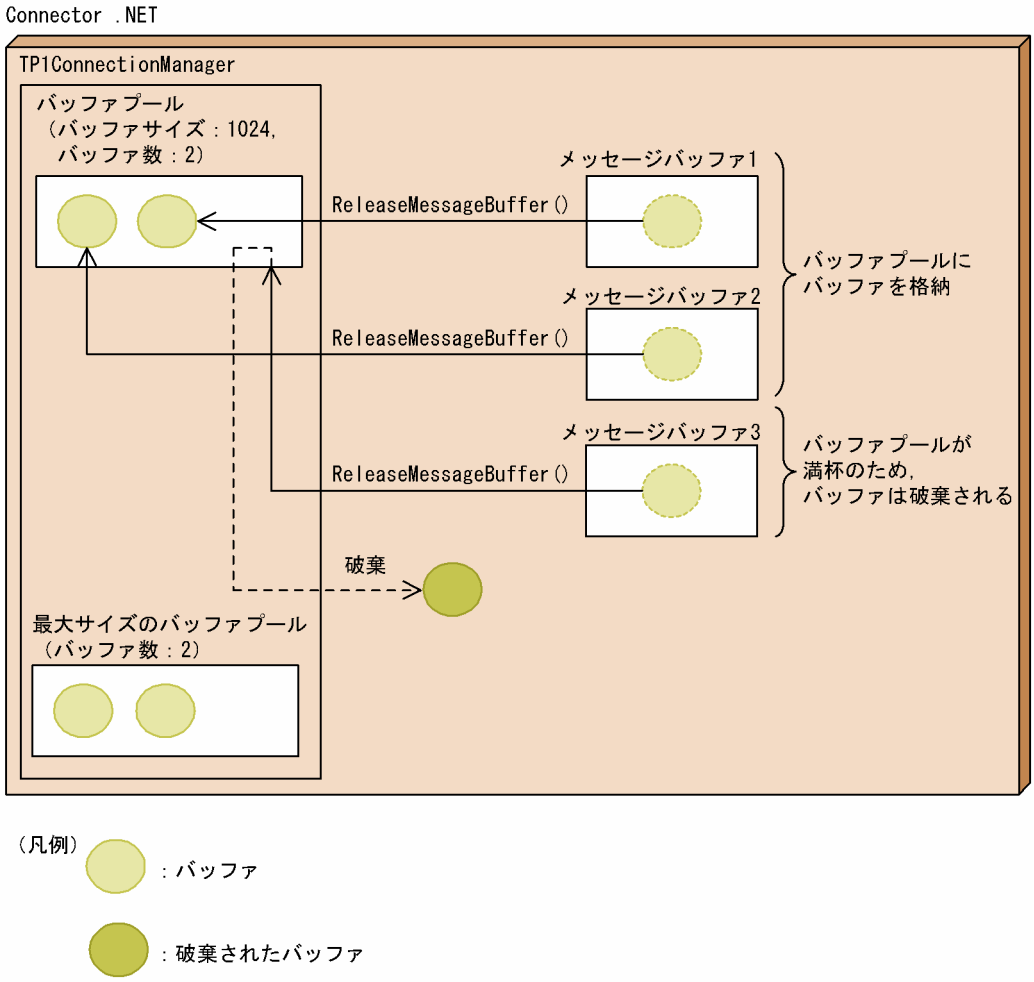
GetMessageBuffer メソッドに，構成定義で指定したバッファプールサイズの最大値を超えるサイズが指定された場合は，最大サイズ（1048576 バイト）のバッファプールからバッファを取得します。ただし，このメッセージバッファが最大サイズのバッファを取得するのは，処理を続行させるためです。アプリケーションが最大サイズのバッファを何度も使い続けると，性能およびリソースへの影響が大きくなります。そのため，構成定義で指定するバッファサイズの値は，アプリケーションの用途に合わせて見積もってください。

なお，RPC 送受信メッセージの最大長拡張機能を使用する場合は，GetMessageBuffer メソッドに構成定義で指定したバッファプールサイズの最大値を超えるサイズが指定されたときでも，<option>要素の maxMessageSize 属性で指定した値（×1048576 バイト）が設定されます。

(2) バッファの解放

バッファ解放時のバッファプールの構造を，次の図に示します。

図 2-28　バッファ解放時のバッファプールの構造



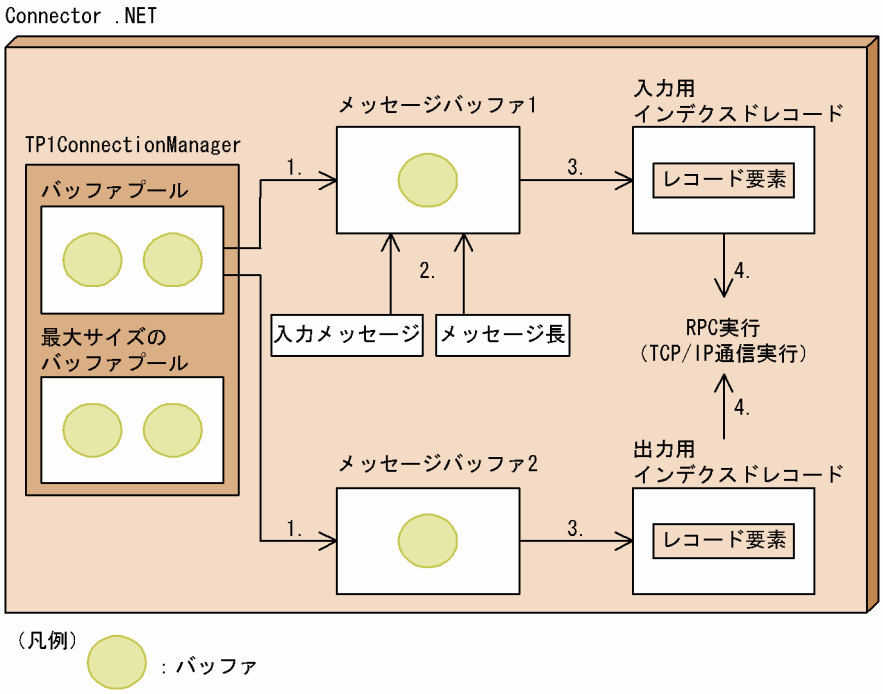
バッファを解放するときは，MessageBuffer クラスの ReleaseMessageBuffer メソッドを呼び出します。バッファを格納するバッファプールに空きがあれば，バッファはバッファプールに戻ります。バッファプールが満杯のときは，バッファはプーリングされないで破棄されます。

2.5.4　インデクスドレコード使用時のバッファの取得と解放

(1)　バッファの取得

RPC 要求および TCP/IP 通信にインデクスドレコードを使用する場合のバッファの取得について，次の図に示します。

図 2-29　インデクスドレコード使用時のバッファの取得



1. アプリケーションから TP1ConnectionManager クラスの GetMessageBuffer メソッドを呼び出して，メッセージバッファを生成します。

   GetMessageBuffer メソッドのパラメタには，メッセージバッファに取得するバッファに設定する入力メッセージまたは応答メッセージのメッセージ長を指定してください。

2. メッセージバッファが保持するバッファに，入力メッセージを設定します。次のどちらかの方法で設定してください。

   - MessageBuffer クラスの Append メソッドを呼び出して，入力メッセージ（バイト配列）を設定してください。
   - MessageBuffer クラスの Buffer プロパティで，バッファに直接入力メッセージを設定します。また，MessageLength プロパティでメッセージ長を設定してください。

3. IndexedRecord クラスの Add メソッドを呼び出し，インデクスドレコードのレコード要素としてメッセージバッファを設定してください。※

4. TP1Connection クラスまたは TcpipConnection クラスの Execute メソッドを呼び出し，RPC または TCP/IP 通信を実行します。

注※

入力用インデクスドレコードのレコード要素には，複数のメッセージバッファを設定することもできます。その場合は，メッセージ長の合計は 1 から 1048576 までの範囲で指定してください。ただし，RPC 送受信メッセージの最大長拡張機能を使用する場合，最大サイズには<option>要素の maxMessageSize 属性で指定した値（×1048576 バイト）が設定されます。

また，インデクスドレコードに設定したバッファのほかに，これらのメッセージ長の合計長のメッセージが格納できるバッファが RPC または TCP/IP 通信の実行時に Connector .NET の内部で使用され

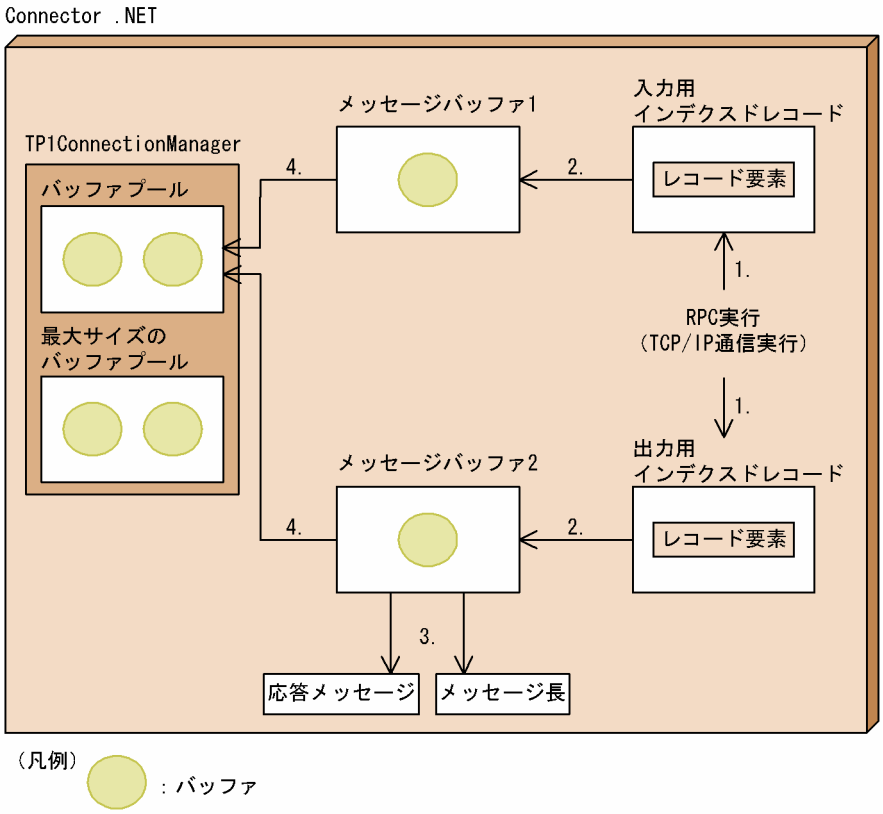
ます。そのため，バッファ数およびバッファサイズを見積もるときはこの分のバッファも考慮してください。

なお，出力用インデクスドレコードのレコード要素に設定できるメッセージバッファは一つだけです。

(2) バッファの解放

RPC 要求または TCP/IP 通信にインデクスドレコードを使用する場合のバッファの解放について，次の図に示します。

図 2-30 インデクスドレコード使用時のバッファの解放



1. RPC 要求または TCP/IP 通信が実行されると，TP1Connection クラスまたは TcpipConnection クラスの Execute メソッドは，インデクスドレコードに戻り値を返します。
2. アプリケーションが，インデクスドレコードからメッセージバッファを取り出します。
3. MessageBuffer クラスの Buffer プロパティで，応答メッセージを設定します。また，MessageLength プロパティでメッセージ長を取得してください。
4. MessageBuffer クラスの ReleaseMessageBuffer メソッドを呼び出すと，メッセージバッファに保持されているバッファがバッファプールに戻されます。

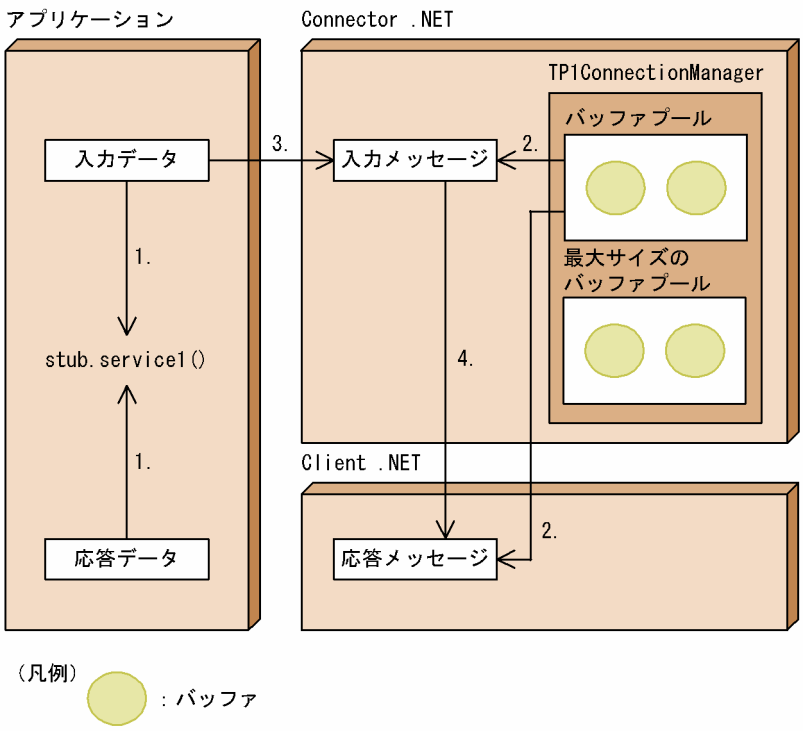
   ReleaseMessageBuffer メソッドを呼び出したあとは，メッセージバッファのバッファにはアクセスできません。

2.5.5　クライアントスタブ使用時のバッファの取得と解放

(1)　バッファの取得

RPC 要求にクライアントスタブを使用する場合のバッファの取得について，次の図に示します。

図 2-31　クライアントスタブ使用時のバッファの取得

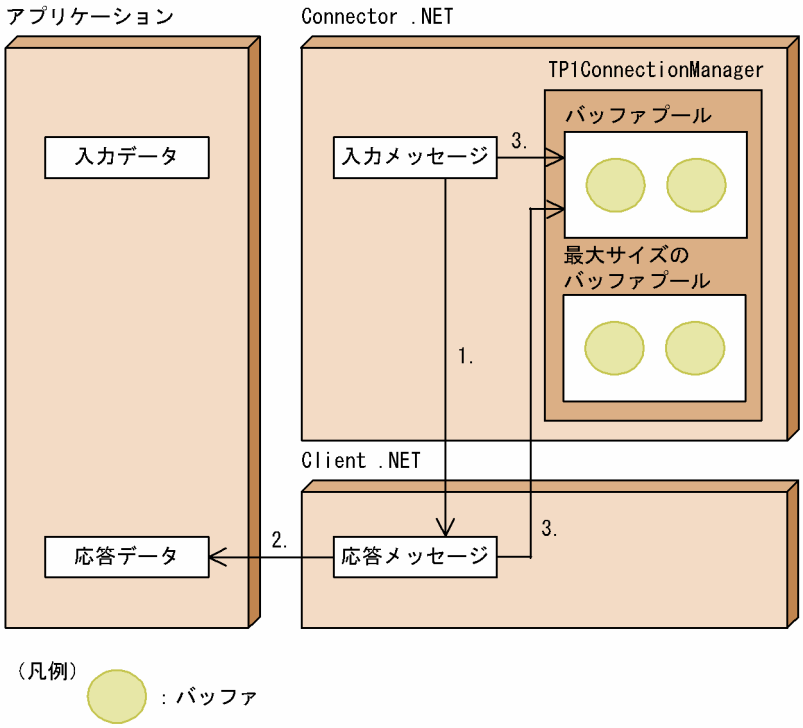


1. アプリケーションがクライアントスタブのサービスメソッドを呼び出すと，Connector .NET は入力メッセージ長を計算します。
2. Connector .NET は，バッファプールから入力メッセージ用のバッファを取得します。
3. 取得した入力メッセージ用のバッファに，カスタムレコードまたは入力パラメタの内容をコピーします。
4. Client .NET の Call メソッドを呼び出します。

(2)　バッファの解放

RPC 要求にクライアントスタブを使用する場合のバッファの解放について，次の図に示します。

図 2-32　クライアントスタブ使用時のバッファの解放

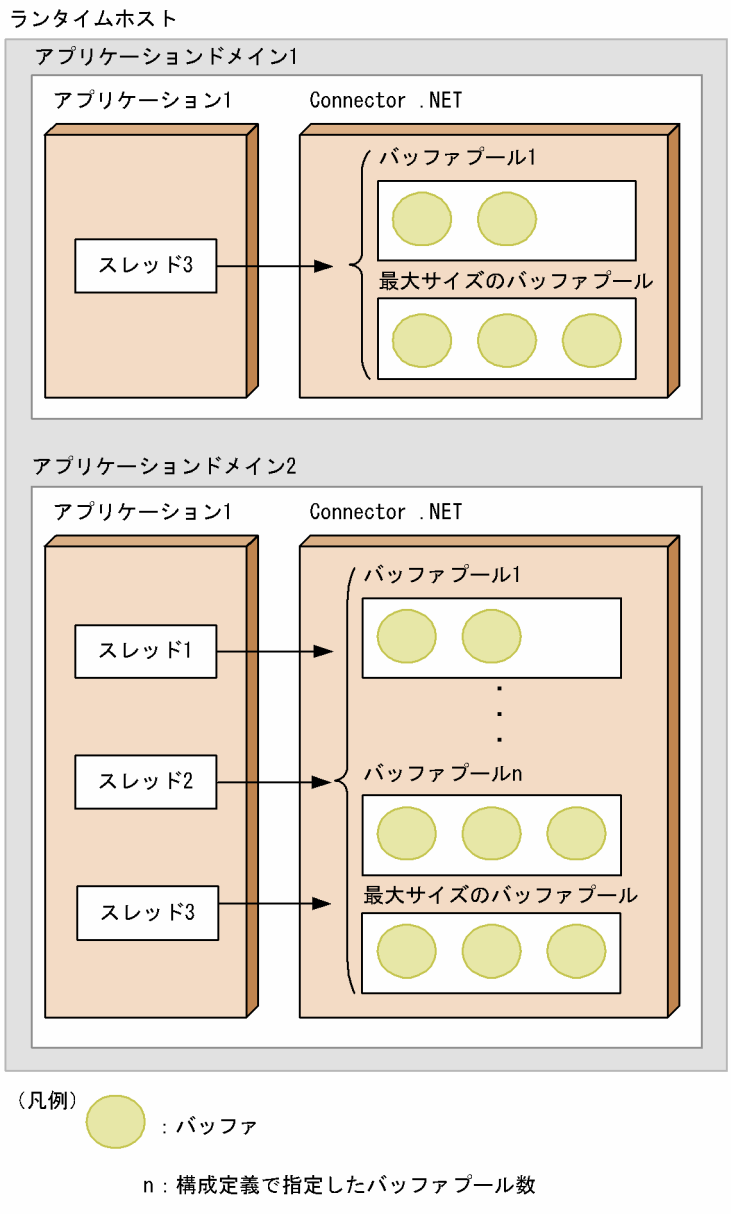


1. Client .NET の Call メソッドを呼び出すと，Client .NET はバッファプールから応答メッセージ用のバッファを取得して受信処理をします。
2. Connector .NET は，応答メッセージを出力用のカスタムレコードまたは出力パラメタにコピーします。
3. 応答メッセージが格納されたバッファを，バッファプールに戻します。

2.5.6　バッファプーリング機能を使用したアプリケーションの実行

バッファプーリング機能を使用した場合のアプリケーションの実行例を示します。

図 2-33　バッファプーリング機能を使用した場合のアプリケーションの実行例



注意事項

バッファプールは必ずアプリケーションドメイン単位に生成されます。プロセス単位で生成されることはありません。

2.5.7　バッファサイズの見積もり方法

RPC 要求にクライアントスタブを使用する場合，入出力パラメタやカスタムレコードクラスと，入出力メッセージの間の変換を Connector .NET が行うため，Connector .NET のアプリケーションは直接バッファ長を意識する必要はありません。しかし，バッファプーリング機能使用時のバッファサイズとプール

数を適切に設定するために，入力メッセージ長および応答メッセージ長の値を見積もっておく必要があります。

(1) .NET インタフェース定義から生成されたクライアントスタブを使用する場合

(a) データ型ごとの合計値を計算する

次に示す計算式を用いて，入力および出力で使用するデータ型ごとの合計値を計算してください。

入力メッセージ長＝メソッドの入力パラメタおよび参照パラメタの最大データ長の合計＋ 512

応答メッセージ長＝メソッドの出力パラメタ，参照パラメタ，および戻り値の最大データ長の合計＋ 512

計算式に代入する値を，次の表に示します。

表 2-7　計算式に代入する値（.NET インタフェース定義）

パラメタのデータ型	メッセージ上のサイズの最大値（バイト）
System.Byte	1
System.Int16	3
System.Int32	7
System.Int64	15
System.String	格納された文字数×2 ＋ 13
System.Byte[]	a ＋ 11
System.Int16[]	2×a ＋ 11
System.Int32[]	4×a ＋ 11
System.Int64[]	8×a ＋ 11
System.String[]	Σ（格納された文字数×2 ＋ 13）＋ 11
TP1 ユーザ構造体	各メンバの最大長の合計＋ 4
TP1 ユーザ構造体配列	Σ（各メンバの最大長の合計＋ 4）＋ 11

（凡例）

a：配列の要素数

注

スタブによって自動的に調整されるサイズを含みます。

(b) データトレースを使用する

Client .NET のトラブルシュート機能であるデータトレースを使用して，実際に送受信されたメッセージ長を確認できます。データトレースの詳細については，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。

(2)　サービス定義から生成されたカスタムレコードクラスを使用する場合

(a)　データ型ごとの合計値を計算する

次に示す計算式を用いて，入力および出力で使用するデータ型ごとの合計値を計算してください。

入力メッセージ長＝データ型定義の各メンバのデータ長の合計

応答メッセージ長＝データ型定義の各メンバのデータ長の合計

計算式に代入する値を，次の表に示します。

表 2-8　計算式に代入する値（サービス定義）

データ型定義のメンバのデータ型	メッセージ上のサイズ（バイト）
char	1
short	2
int	4
long	4
char[a]	1×a
char[a][b]	a×b
short[a]	2×a
int[a]	4×a
long[a]	4×a
byte[a]	1×a
struct	構造体の各メンバのサイズの合計
struct[a]	構造体の各メンバのサイズの合計×a

（凡例）

a，b：配列の要素数

注

可変長構造体配列の場合，配列の最大要素数で見積もるようにしてください。

(b)　カスタムレコードクラスのソースコードを確認する

カスタムレコードクラスのソースコードの private const フィールド_length を参照して，計算後のメッセージ上のサイズが確認できます。

(c)　クライアントスタブ生成コマンド（サービス定義用）を実行する

カスタムレコードクラスごとに必要なバッファサイズは，クライアントスタブ生成コマンド（サービス定義用）（spp2cstub）に-b オプションを指定して実行することで確認できます。運用コマンドの詳細につ

いては，「5.　運用コマンド」の「spp2cstub（クライアントスタブ生成コマンド（サービス定義用））」を参照してください。

2.6 TSP 自動生成機能

2.6.1 TSP 自動生成機能の概要

Connector .NET は，.NET インタフェース定義またはサービス定義から TP1 Service Proxy（TSP）を自動生成できます。

TSP は，コネクションを意識しないで OpenTP1 のサービスを利用できるプロキシクラスです。OpenTP1 のサービスを利用する操作を仮想化することで，Web サービスや Web アプリケーションなどから OpenTP1 のサービスが利用できるようになります。

また，OpenTP1 上のサービスを ASP.NET XML Web サービスとして公開するための ASP.NET XML Web サービスクラスを自動生成できます。これによって，SOAP プロトコルで OpenTP1 のサービスを利用できるようになります。

(1) 入力情報と運用コマンド

サーバ UAP の種別によって，TSP 自動生成機能を使用するために用いるインタフェース情報および運用コマンドが異なります。次の表に，サーバ UAP の種別ごとの入力情報と運用コマンドを示します。

表 2-9 TSP 自動生成機能を使用する際の入力情報と運用コマンド

サーバ UAP の種別	入力元のインタフェース情報	運用コマンド
.NET インタフェース定義を利用した SPP.NET	.NET インタフェース定義	TSP 生成コマンド（.NET インタフェース定義用）（if2tsp）
SPP，または.NET インタフェース定義を利用しない SPP.NET	サービス定義	TSP 生成コマンド（サービス定義用）（spp2tsp）

(2) 生成されるファイル

TSP 自動生成機能を使用すると，次のファイルが生成されます。

- TSP 実装クラス※1
- クライアントスタブ※1
- カスタムレコードクラス※1，※2
- 構成ファイル（App.config）
- ASP.NET XML Web サービスのソースプログラムファイル（拡張子は asmx）
- ASP.NET XML Web サービスのソースプログラムの分離コードファイル（拡張子は asmx.cs，asmx.vb，または asmx.jsl）※1

注※1

これらのファイルは，C#，J#，および Visual Basic の各プログラム言語で出力できます。

注※2

入力元のインタフェース情報がサービス定義の場合にだけ生成されます。

(3) TSP を実行するための環境および手順

(a) 実行環境

TSP 自動生成機能によって生成された TSP を実行するためには，次の環境が必要です。

- .NET Framework 3.5 Service Pack 1※
- Visual Studio

注※

ASP.NET の実行環境が構築されている必要があります。

(b) 実行手順

TSP を実行する手順を次に示します。

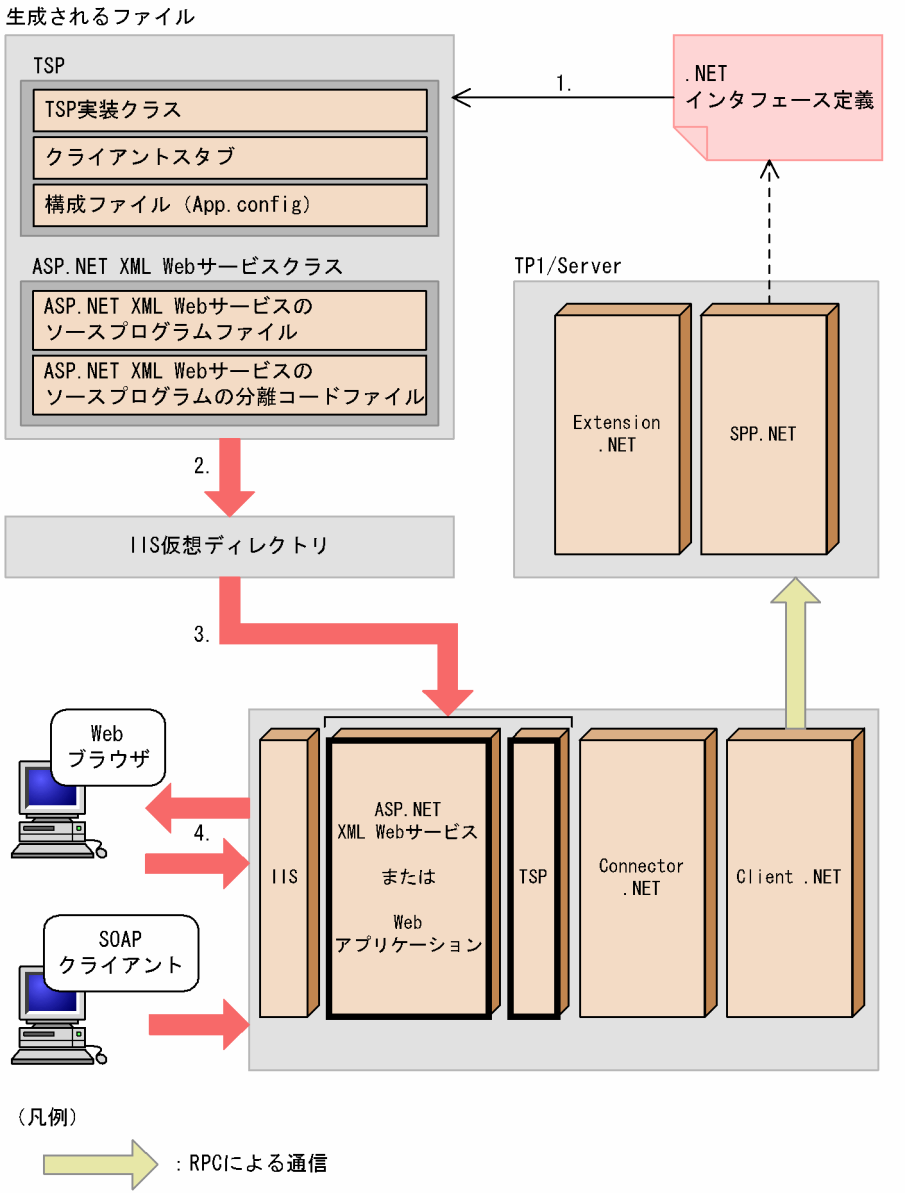
1. Visual Studio で，ASP.NET Web サービスのプロジェクトを作成します。
2. 生成された IIS 仮想ディレクトリに対応するディレクトリに，自動生成されたファイルをコピーし，プロジェクトに登録します。
3. プロジェクトをビルドします。

2.6.2 TSP 自動生成機能の運用

(1) サーバが.NET インタフェース定義を利用した SPP.NET の場合

サーバが.NET インタフェース定義を利用した SPP.NET の場合の，TSP 自動生成機能使用時の流れを次の図に示します。

図 2-34　TSP 自動生成機能使用時の流れ（サーバが.NET インタフェース定義を利用した SPP.NET の場合）



1. TSP 生成コマンド（.NET インタフェース定義用）（if2tsp）を実行し，TSP を生成します。オプションで，ASP.NET XML Web サービスクラスを同時に生成できます。
2. 生成されたファイルを，IIS 仮想ディレクトリに配置します。

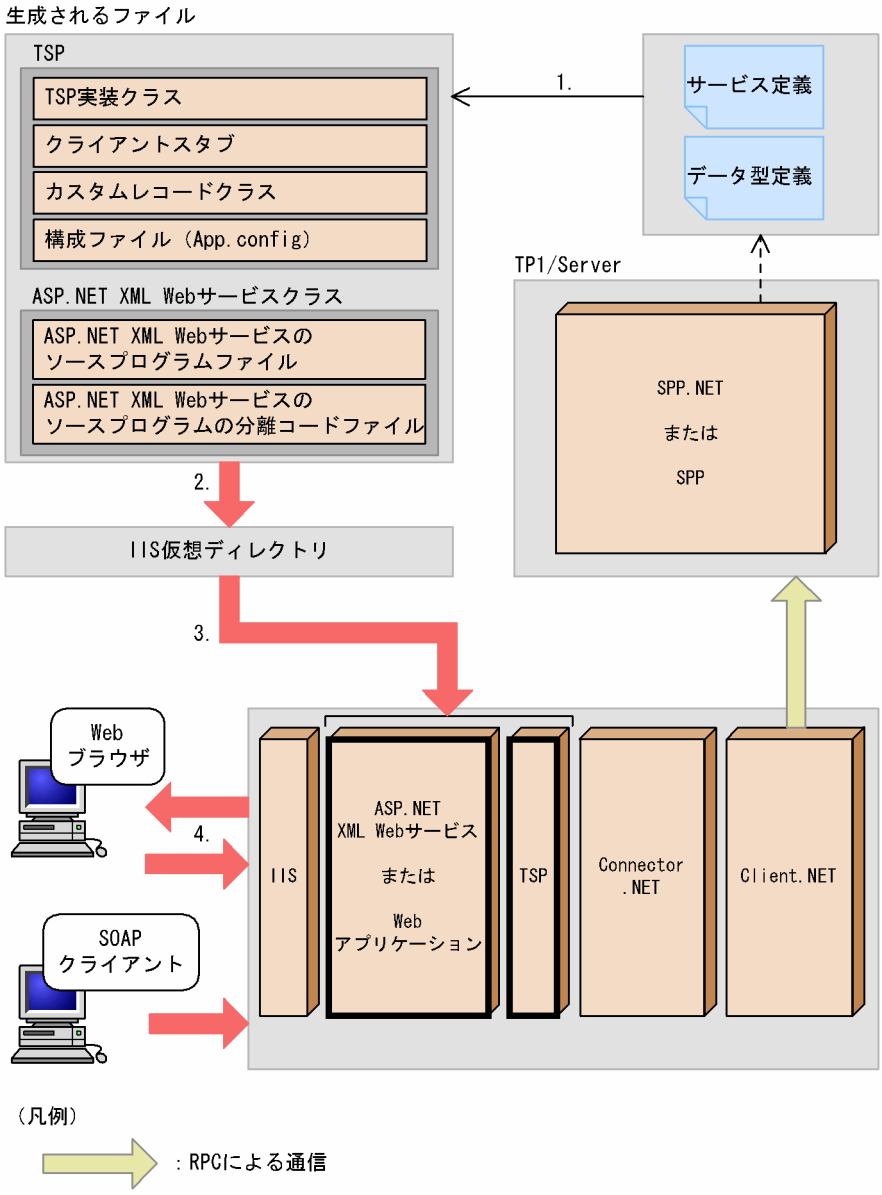
   詳細については，「2.6.1(3)　TSP を実行するための環境および手順」を参照してください。
3. ASP.NET Web サービスまたは Web アプリケーションの実行時に，IIS 仮想ディレクトリを読み込みます。
4. ASP.NET XML Web サービスまたは Web アプリケーションが利用できます。

   また，ASP.NET XML Web サービスとして公開された WSDL は，Web ブラウザから「〈公開された XML Web サービスの URL〉?wsdl」を指定してアクセスすることで参照できます。

(2)　サーバが SPP，または.NET インタフェース定義を利用しない SPP.NET の場合

サーバが SPP，または.NET インタフェース定義を利用しない SPP.NET の場合の，TSP 自動生成機能使用時の流れを次の図に示します。

図 2-35　TSP 自動生成機能使用時の流れ（サーバが SPP，または.NET インタフェース定義を利用しない SPP.NET の場合）



1. TSP 生成コマンド（サービス定義用）（spp2tsp）を実行し，TSP を生成します。オプションで，ASP.NET XML Web サービスクラスを同時に生成できます。
2. 生成されたファイルを，IIS 仮想ディレクトリに配置します。

   詳細については，「2.6.1(3)　TSP を実行するための環境および手順」を参照してください。

3. ASP.NET XML Web サービスまたは Web アプリケーションの実行時に，IIS 仮想ディレクトリを読み込みます。
4. ASP.NET XML Web サービスまたは Web アプリケーションが利用できます。
   また，ASP.NET XML Web サービスとして公開された WSDL は，Web ブラウザから「〈公開された XML Web サービスの URL〉?wsdl」を指定してアクセスすることで参照できます。

2.6.3 TSP の動作設定

(1) TSP 実装クラスの利用方法

TSP 実装クラスは，ASP.NET XML Web サービスおよび ASP.NET Web アプリケーションを含む，任意のアプリケーションから利用できます。

TSP の動作は，次の表に示す TSP 実装クラスのプロパティまたは ASP.NET 構成ファイルで変更します。自動生成された TSP 実装クラスのソースコードは変更しないでください。

表 2-10 TSP の実行時に指定できる項目

指定個所	指定できる項目
プロパティ	RPC 要求先のサービスグループ名
	Connector .NET 構成定義のプロファイル ID
	RPC 形態
	RPC 要求の最大応答監視時間
ASP.NET 構成ファイル	RPC 要求先のサービスグループ名
	Connector .NET 構成定義のプロファイル ID
	RPC 要求の最大応答監視時間（Web サービスメソッドごとに指定）

(2) TSP 実装クラスのプロパティの詳細

TSP 実装クラスのプロパティの詳細を，次の表に示します。

表 2-11 TSP 実装クラスのプロパティの詳細

名称	説明	データ型
ServiceGroup※	RPC 要求先のサービスグループ名を設定，および取得します。	System.String
ProfileID※	Connector .NET 構成定義のプロファイル ID を設定，および取得します。	System.String
RpcFlags	RPC 実行時のサービス呼び出し形態を設定，および取得します。RpcInfo クラスの DCNOFLAGS フィールド，または DCRPC_NOREPLY フィールドを設定します。	System.Int32

名称	説明	データ型
	それ以外の値が設定された場合は，次の例外が発生します。 `Hitachi.OpenTP1.Connector.TP1ConnectorException`	
WatchTime※	RPC の形態が同期応答型 RPC の場合に，SPP.NET にサービス要求を送ってからサービスの応答が返るまでの最大応答待ち時間（単位：秒）を設定および取得します。 -1，および 0～65535 までの整数が設定できます。-1 を指定した場合は，対応する Client .NET 構成定義の設定に従います。	－

（凡例）

－：該当しません。

注※

プロパティの設定値が不正な場合，TSP 実装クラスのサービスメソッドが呼び出された時点で TP1ConnectorException が発生します。

(3) ASP.NET 構成ファイルの詳細

(a) 構成定義名と設定値

ASP.NET 構成ファイルを変更して TSP の動作プロパティを設定するときは，<appSettings>要素の子要素に設定します。構成定義名と設定値を次の表に示します。

表 2-12 構成定義の設定（TSP）

構成定義名	指定内容	設定値※2 《デフォルト値》
〈TSP 実装クラス名〉.ServiceGroup※1	RPC 要求先のサービスグループ名を指定します。	31 文字以内の識別子 《TSP 生成コマンドで指定したサービスグループ名》
〈TSP 実装クラス名〉.ProfileId※1	Connector .NET 構成定義のプロファイル ID を指定します。	文字列 《空文字》
〈TSP 実装クラス名〉.〈サービス名〉.WatchTime※1	RPC 要求時の最大応答監視時間を，秒単位で指定します。対応するサービスの RPC 形態が同期応答型の場合にだけ有効です。 -1 を指定した場合は，対応する Client .NET 構成定義の設定に従います。	-1，0～65535 の整数（単位：秒） 《-1》

注※1

〈TSP 実装クラス名〉は，名前空間を含んだ完全限定名で指定します。

注※2

設定値が不正の場合は，デフォルト値が仮定されます。

(b) 構成定義の指定例

次の条件での構成定義の指定例を示します。

- TSP 実装クラス名：MyCompany.GYOUMU1WebService
- サービスグループ名：SVGRP1

【指定例】

```
<configuration>
…
 <appSettings>
   <add key="MyCompany.GYOUMU1WebService.ServiceGroup"
        value="SVGRP1" />
   <add key="MyCompany.GYOUMU1WebService.ProfileId"
        value="server1" />
   <add key="MyCompany.GYOUMU1WebService.PUTDATA.WatchTime"
        value="180" />
 </appSettings>
…
</configuration>
```

2.6.4　ASP.NET XML Web サービスの動作設定

(1)　ASP.NET XML Web サービスの設定

属性を修正したい場合やデータを加工したい場合には，ASP.NET XML Web サービスのソースプログラムの分離コードファイルを変更します。

また，ASP.NET XML Web サービスの動作は，生成時に指定する運用コマンドのオプションでも設定できます。指定できるオプションを次の表に示します。

表 2-13　ASP.NET XML Web サービスクラス生成時に運用コマンドのオプションで指定できる項目

オプション名	パラメタ	指定項目
-S	doc または rpc	SOAP Body 全体の書式
-x※	literal または encoded	SOAP パラメタの書式指定スタイル
-w	ユーザ指定	XML Web サービスの名前空間の URI
-N	ユーザ指定	SOAP 要求および SOAP 応答に関連づけられる名前空間の URI
-B	wsibp11 または none	XML Web サービスクラスが準拠していることを示す WS-I の仕様
-A	true または false	Web サービス記述言語での WS-I Basic Profile 準拠の表示の有無

注※

このオプションは，SOAP Body 全体の書式が Document の場合にだけ有効です。

(2) ASP.NET XML Web サービスクラスのソースコード例

指定したオプションは，次のように ASP.NET XML Web サービスクラスのソースコードに反映されます。

【ASP.NET XML Web サービスクラスのソースコード例】

```
//-wで指定した値
[WebService(Namespace="http://tempuri.org/")]

//-Sにdocを指定した場合
[SoapDocumentService(SoapBindingUse.Literal)]
public class sampleClassWebService : WebService {

//-Bにnoneを指定した場合
[WebServiceBinding(ConformsTo=WsiProfiles.None,

//-Aにtrueを指定した場合
EmitConformanceClaims=true)]

//-xおよび-Nで指定した値
[SoapDocumentMethod(RequestNamespace="http://tempuri.org/",
 ResponseNamespace="http://tempuri.org/")]
     [WebMethod()]
     public Record2 Service1(Record1 param0) {
         try {
             sampleClassProxy proxyClass =
               new sampleClassProxy();
             Record2 outRecord = new Record2();
             proxyClass.Service1(param0, outRecord);
             return outRecord;
         }
         catch (Hitachi.OpenTP1.TP1Exception tp1e) {
             System.Type t = tp1e.GetType();
             string className = t.FullName;
             string tp1eMessage = tp1e.Message;
             string errorCode =
               ("ErrorCode = " + tp1e.ErrorCode);
             string ErrorStr = (className + ": ");
             ErrorStr = (ErrorStr + tp1eMessage);
             ErrorStr = (ErrorStr + " ");
             ErrorStr = (ErrorStr + errorCode);
             throw new SoapException
               (ErrorStr, SoapException.ServerFaultCode, tp1e);
         }
     }
}
```

2.6.5 パラメタの対応づけ規則

.NET インタフェース定義，またはサービス定義およびデータ型定義で指定されたデータ型から，TSP 実装クラスへのパラメタの対応づけ規則は，各定義のメソッドの宣言と同じです。ただし，ASP.NET XML Web サービスクラスについては，次のようになります。

〈.NET インタフェース定義から生成した場合〉

.NET インタフェース定義の各メソッドの宣言が，そのまま ASP.NET XML Web サービスクラスの Web サービスメソッドの宣言と同じになります。

〈サービス定義から生成した場合〉

サービス定義の各サービスで指定した入出力パラメタがカスタムレコードクラスに対応づけられ，ASP.NET XML Web サービスの Web サービスメソッドのパラメタには入力用カスタムレコードクラスが，戻り値には出力用カスタムレコードクラスが宣言されます。

参考

ASP.NET XML Web サービスクラスのパラメタから WSDL への対応づけ規則については，.NET Framework のドキュメントを参照してください。なお，TSP 自動生成コマンドで指定した SOAP Body およびパラメタの書式指定によって，WSDL へのデータ型の対応づけ規則が変わります。

2.7　リソース監視機能

Connector .NET では，リソース（コネクションおよびバッファ）の使用状況を監視できます。この機能を利用して，コネクションプールおよびバッファプールのチューニングができます。

2.7.1　リソース不足を通知する警告メッセージの出力

構成定義で指定したコネクションおよびバッファに関するしきい値を超えた場合，警告メッセージが Connector .NET のログファイルに出力されます。これによって，コネクションおよびバッファの不足を事前に検知して，対処できます。Connector .NET のログファイルに警告メッセージを出力するには，<log>要素の level 属性に 1 以上を指定してください。

監視される対象を次に示します。

- 最大同時使用コネクション数（<connection>要素の active 属性で指定）
- 最大バッファプール内のバッファ数（<largestBufferPool>要素の maxCount 属性で指定）
- バッファプール内のバッファ数（<bufferPool>要素の maxCount 属性で指定）

上記の監視対象に対するしきい値は，<connection>要素，<largestBufferPool>要素，および<bufferPool>要素の threshold 属性で指定します。

なお，警告メッセージが出力された場合，上記の監視対象の不足が解消されるまで警告メッセージは出力されません。

この機能を使用する場合の構成定義の指定例を次に示します。この指定例では，最大同時使用コネクション数が 80%，最大バッファプール内のバッファ数が 100%，またはバッファプール内のバッファ数が 120%を超えた場合に，警告メッセージが出力されます。

【指定例】

```
…
<connection pooled=”10”
            active=”20”
            threshold=”80”
            timeout=”180”/>
<log destination=”c:¥temp¥connectorn” fileSize=”1048576”
     level=”1”/>
<buffer>
  <largestBufferPool maxCount=”5”
                     threshold=”100”/>
  <bufferPool size=”10240”
              maxCount=”10”
              threshold=”120”/>
</buffer>
…
```

構成定義の詳細については，「3. 構成定義」の「connection」，「log」，「largestBufferPool」，および「bufferPool」を参照してください。

2.7.2 パフォーマンスカウンタへのリソース使用状況の出力

コネクションおよびバッファの使用状況を Windows のパフォーマンスカウンタに出力することで，コネクションおよびバッファの使用状況を監視できます。パフォーマンスカウンタに出力されるコネクションおよびバッファの情報を次の表に示します。

表 2-14 パフォーマンスカウンタに出力されるコネクションおよびバッファの情報

項目		名称	説明
カテゴリ名称		TP1/Connector.NET Connection	Connector .NET のコネクションに関するカウンタ
カウンタ名称	プールされているコネクション数	Pooled Connections	同一アプリケーションで，現時点でプールされているコネクション数
	使用中のコネクション数	Active Connections	同一アプリケーションで，現時点で使用中のコネクションオブジェクトの数
	コネクションの取得回数	GetConnection Total	同一アプリケーションで発行された，TP1ConnectionManager クラスの GetConnection メソッドおよび GetTcpipConnection メソッドの合計発行回数（メソッドの実行結果の成否に関係なくカウントされた結果を表示）
	コネクションが解放された回数	Dispose Total	同一アプリケーションで発行された，TP1Connection クラスの Dispose メソッドおよび TcpipConnection クラスの Dispose メソッドの合計発行回数
	プールからコネクションを取得した回数	Use of Pooled Connections Total	同一アプリケーションで，コネクションプールからコネクションが取得された合計回数
	コネクションの生成回数	Created Connections	同一アプリケーションで，コネクションオブジェクトが新規に生成された回数
カテゴリ名称		TP1/Connector.NET Buffer	Connector .NET のバッファに関するカウンタ
カウンタ名称	プールされているバッファ数	Pooled Buffers	同一アプリケーションで，現時点でプールされているバッファ数（プロファイル ID およびバッファサイズごとに表示）

項目		名称	説明
	使用中のバッファ数	Active Buffers	同一アプリケーションで，現時点で使用中のバッファ数（プロファイル ID およびバッファサイズごとに表示）
	バッファの取得回数	GetMessageBuffer Total	同一アプリケーションで発行された，TP1ConnectionManager クラスの GetMessageBuffer メソッドの合計発行回数（メソッドの実行結果の成否に関係なくカウントされた結果を，プロファイル ID およびバッファサイズごとに表示）
	バッファが解放された回数	ReleaseBuffer Total	同一アプリケーションで発行された，MessageBuffer クラスの ReleaseMessageBuffer メソッドの合計発行回数（プロファイル ID およびバッファサイズごとに表示）
	プールからバッファを取得した回数	Use of Pooled Buffers Total	同一アプリケーションで，バッファプールからバッファを取得した合計回数（プロファイル ID およびバッファサイズごとに表示）
	バッファの生成回数	Created Buffers	同一アプリケーションで，新規に生成されたバッファの合計回数（プロファイル ID およびバッファサイズごとに表示）
インスタンス名称		• コネクション：アプリケーション名 • バッファ：アプリケーション名_プロファイル ID_バッファサイズ	現時点で起動中の Connector .NET を使用するアプリケーションの名称※

注※

コネクションおよびバッファに関するインスタンスは，次のメソッドが発行された時点で初期化され，アプリケーションを実行するプロセスが終了する時点で削除されます。

コネクションに関するインスタンス

- TP1ConnectionManager クラスの GetConnection メソッド
- TP1ConnectionManager クラスの GetTcpipConnection メソッド

バッファに関するインスタンス

- TP1ConnectionManager クラスの GetMessageBuffer メソッド

  web.config の変更などによってアプリケーションドメインだけが再構築された場合は，同じ名前のインスタンスで Active Connections，Pooled Connections，Active Buffers，および Pooled Buffers の値が 0 に初期化され，そのほかの値はプロセス終了まで継続して増加します。

パフォーマンスカウンタへのカテゴリおよびカウンタの登録・削除は，Connector .NET のインストールおよびアンインストール時に行われます。

パフォーマンスカウンタにコネクションおよびバッファの使用状況を出力するには，構成定義で<perfCounter>要素の use 属性に true を指定します。<perfCounter>要素は<common>要素の子要素として指定してください。構成定義での指定例を次に示します。

【指定例】

```
<hitachi.opentp1.connector>
 <common>
  …
  <perfCounter use="true"/>
  …
 </common>
</hitachi.opentp1.connector>
```

構成定義の詳細については，「3.　構成定義」の「perfCounter」を参照してください。

2.8　接続障害軽減機能

接続障害軽減機能は，あるコネクションが接続障害を検知したあとにほかのコネクションがその接続先に接続することを防止し，接続障害の発生頻度を軽減する機能です。

なお，この節では，コネクション確立時のすべての障害を「接続障害」，通信先となるホスト名または IP アドレス，およびポート番号の組み合わせを「接続先」と表記します。

2.8.1　接続障害軽減機能の概要

接続障害軽減機能を使用すると，コネクションが接続障害を検知した場合に，同一アプリケーションドメイン内のほかのコネクションに対しても接続先ごとの障害情報を共有します。これによって，ほかのコネクションが障害となった接続先に対して接続することを防止し，接続障害の発生頻度を軽減します。接続障害情報はアプリケーションドメイン単位で共有されます。

接続障害軽減機能を使用しない場合，障害を検知していないコネクションは障害が発生した接続先に対しても接続を試みます。そのため，同じ接続先を使用するすべてのコネクションは，稼働中の接続先に切り替えるまでに一度ずつ同じ障害を検知するため時間が掛かります。接続障害軽減機能を使用した場合，一度障害を検知した接続先への接続を防止するため，接続先の切り替えに掛かる時間を短縮できることがあります。なお，障害発生時の接続先の切り替えに必要な時間は，接続障害の種類，構成定義の内容，OS の設定などによって異なります。接続障害軽減機能は，特に接続先ホストのハードウェアや OS の障害時に効果があります。ただし，接続先の一時的な障害（サーバ側での listen キューのオーバーフローなど）も接続障害情報に登録されるため，TP1/Server が正常に稼働していても接続先の切り替えが行われることがあります。

障害を検知した接続先に対しては，一定時間経過ごとに接続を試みることができます。これを**復旧確認動作**といいます。

復旧確認動作は，障害となった接続先が定義されているコネクションのうち，一定時間経過後，最初にサービスを要求するコネクションが行います。復旧を確認し次第，サービスを要求するほかのコネクションを元の接続先に戻します。復旧確認動作を行うかどうかは，Connector .NET 構成定義の<connection>要素の failureCheckInterval 属性で指定します。

接続障害軽減機能は，RPC で通信する場合にだけ使用できます。TCP/IP 通信機能で通信する場合は使用できません。なお，この機能は次の RPC で使用できます。

- リモート API 機能を使用した RPC
- ネームサービスを使用した RPC
- スケジューラダイレクト機能を使用した RPC

2.8.2　構成定義での指定

接続障害軽減機能を使用するには，Connector .NET 構成定義の<connection>要素の failureInfoSharing 属性に true を，failureCheckInterval 属性に復旧確認間隔を指定します。復旧確認動作を行わない場合は，failureCheckInterval 属性に 0 を指定します。

構成定義の詳細については，「3.　構成定義」の「connection」を参照してください。

この機能を使用する場合は，接続障害情報への接続先の登録および削除に関する情報を Connector .NET のログに出力します。これらの情報をログに出力する場合は，ログレベル 2 以上を指定してください。ログレベルの指定については，「3.　構成定義」の「log」を参照してください。

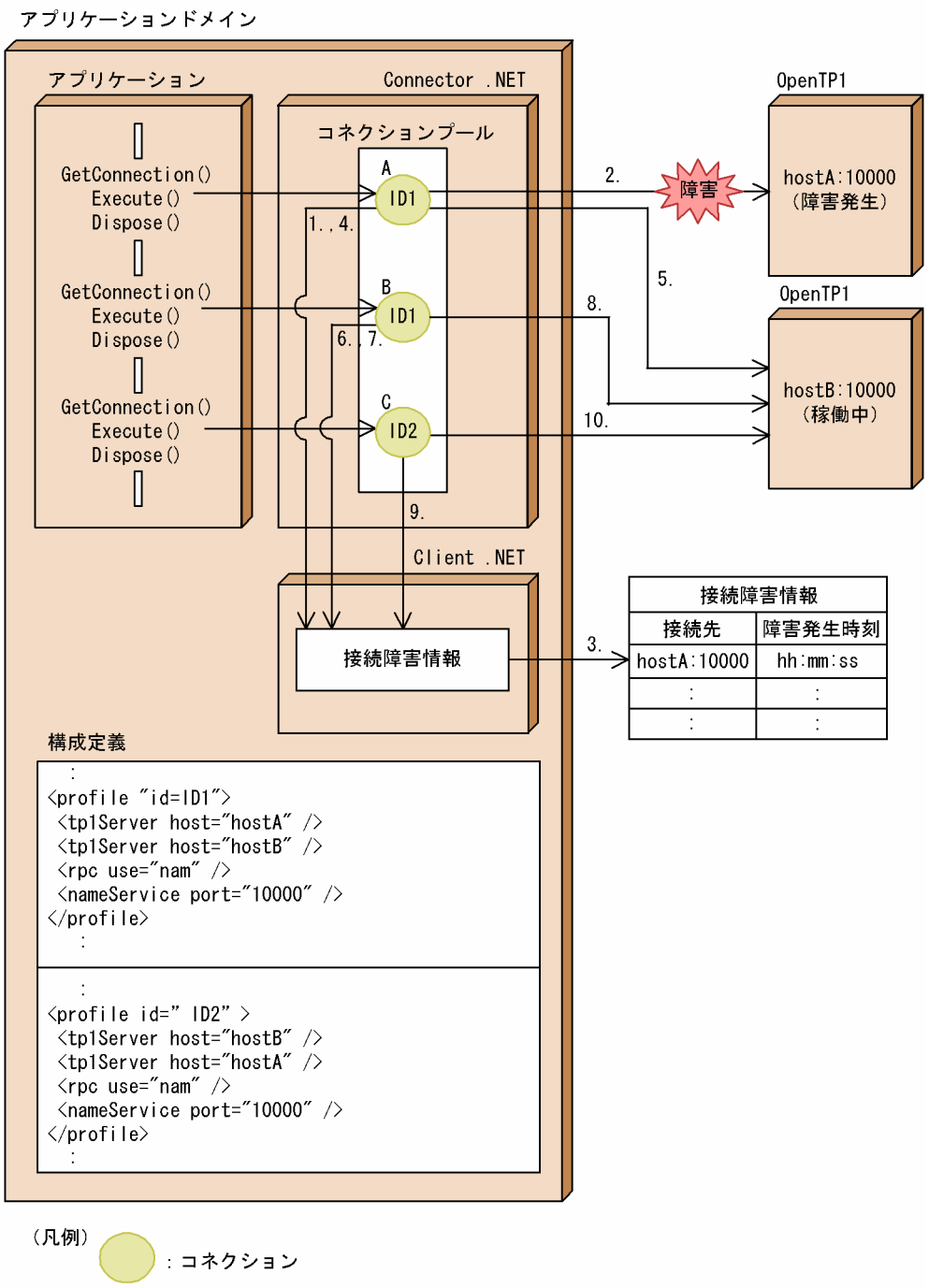
2.8.3　接続障害の検知と復旧確認

(1)　接続障害の検知の流れ

次の条件が定義された場合の接続障害の検知の流れを示します。

- ネームサービスを使用した RPC で行う。
- hostA で障害が発生している。
- コネクション A，B，C の順番に実行する。
- プロファイル ID1：hostA，hostB の順に接続先を選択する。
- プロファイル ID2：hostB，hostA の順に接続先を選択する。

図 2-36　接続障害の検知の流れ



1. コネクション A は，構成定義に指定された hostA:10000 に接続する前に，接続障害情報に hostA:10000 が登録されていないかどうかを確認します。
2. 接続障害情報に hostA:10000 が登録されていないため，接続を試みます。
   hostA:10000 で障害が発生したため，接続に失敗します。
3. 接続障害情報に hostA:10000 が登録されます。
4. コネクション A は，次の接続先として設定されている hostB:10000 に接続する前に，接続障害情報に hostB:10000 が登録されていないかどうかを確認します。
5. 接続障害情報に hostB:10000 が登録されていないため，接続を試みます。

接続に成功します。

6. コネクション B は，構成定義に指定された hostA:10000 に接続する前に，接続障害情報に hostA:10000 が登録されていないかどうかを確認します。

   接続障害情報に hostA:10000 が登録されているため，hostA:10000 には接続しません。

7. コネクション B は，次の接続先として設定されている hostB:10000 に接続する前に，接続障害情報に hostB:10000 が登録されていないかどうかを確認します。

8. 接続障害情報に hostB:10000 が登録されていないため，接続を試みます。

   接続に成功します。

9. コネクション C は，構成定義に指定された hostB:10000 に接続する前に，接続障害情報に hostB:10000 が登録されていないか確認します。

10. 接続障害情報に hostB:10000 が登録されていないため，接続を試みます。

    接続に成功します。

(2) 復旧確認動作の流れ

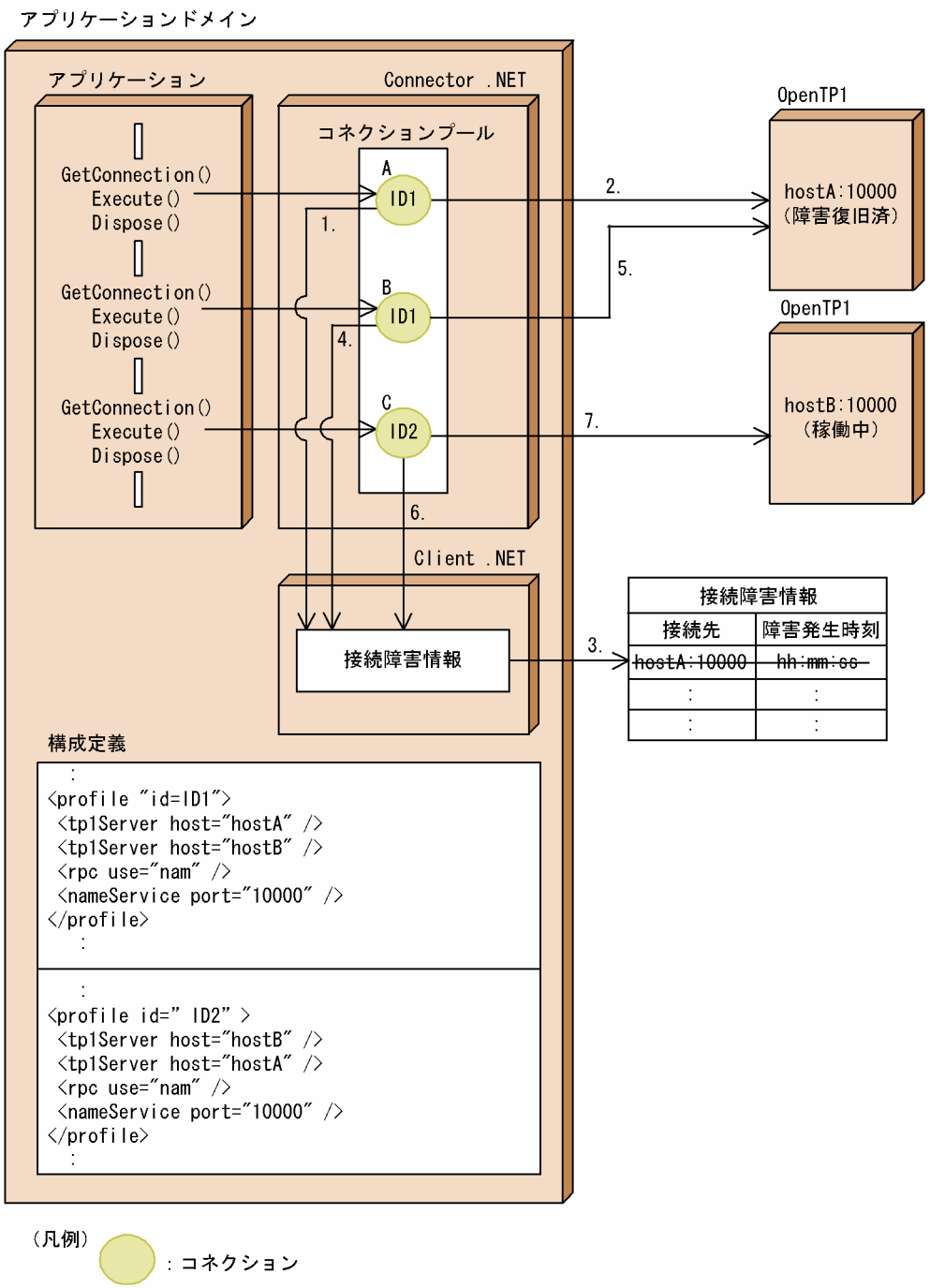
復旧確認動作の流れを，障害後に接続先が復旧している場合と復旧していない場合とに分けて説明します。

(a) 障害後に接続先が復旧している場合

次の条件が定義された場合の復旧確認動作の流れを示します。

- ネームサービスを使用した RPC で行う。
- hostA が障害発生後に復旧している。
- Connector .NET 構成定義の<connection>要素の failureCheckInterval 属性で指定した時間が経過している。
- コネクション A，B，C の順番に実行する。
- プロファイル ID1：hostA，hostB の順に接続先を選択する。
- プロファイル ID2：hostB，hostA の順に接続先を選択する。

図 2-37　復旧確認動作の流れ（障害後に接続先が復旧している場合）



1. コネクション A は，構成定義に指定された hostA:10000 に接続する前に，接続障害情報に hostA:10000 が登録されていないかどうかを確認します。
2. 接続障害情報に hostA:10000 が登録されているが，接続障害情報の障害発生時刻から一定時間が経過しているため，接続を試みます。

   接続に成功します。
3. 接続障害情報から hostA:10000 を削除します。
4. コネクション B は，構成定義に指定された hostA:10000 に接続する前に，接続障害情報に hostA:10000 が登録されていないかどうかを確認します。

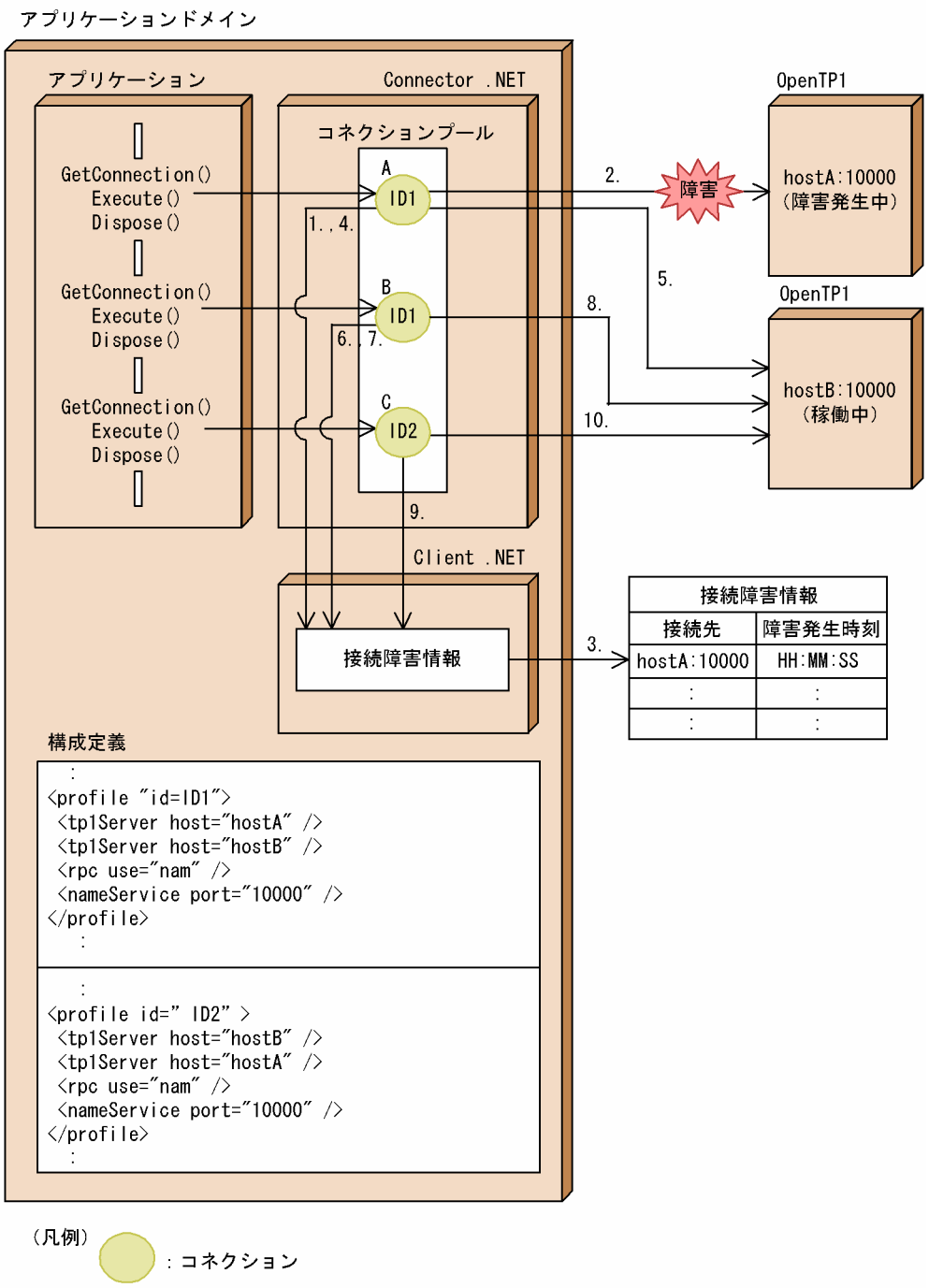
5. 接続障害情報に hostA:10000 が登録されていないため，接続を試みます。
   接続に成功します。
6. コネクション C は，構成定義に指定された hostB:10000 に接続する前に，接続障害情報に hostB:10000 が登録されていないかどうかを確認します。
7. 接続障害情報に hostB:10000 が登録されていないため，接続を試みます。
   接続に成功します。

(b) 障害後に接続先が復旧していない場合

次の条件が定義された場合の復旧確認動作の流れを示します。

- ネームサービスを使用した RPC で行う。
- hostA が障害発生後に復旧していない。
- Connector .NET 構成定義の<connection>要素の failureCheckInterval 属性で指定した時間が経過している。
- コネクション A，B，C の順番に実行する。
- プロファイル ID1：hostA，hostB の順に接続先を選択する。
- プロファイル ID2：hostB，hostA の順に接続先を選択する。

図 2-38　復旧確認動作の流れ（障害後に接続先が復旧していない場合）



1. コネクション A は，構成定義に指定された hostA:10000 に接続する前に，接続障害情報に hostA:10000 が登録されていないかどうかを確認します。
2. 接続障害情報に hostA:10000 が登録されているが，接続障害情報の障害発生時刻から一定時間が経過しているため，接続を試みます。

   hostA:10000 は障害発生中のため，接続に失敗します。
3. 接続障害情報の障害発生時刻を現在の時刻に更新します。
4. コネクション A は，次の接続先として設定されている hostB:10000 に接続する前に，接続障害情報に hostB:10000 が登録されていないかどうかを確認します。

5. 接続障害情報に hostB:10000 が登録されていないため，接続を試みます。
   接続に成功します。
6. コネクション B は，構成定義に指定された hostA:10000 に接続する前に，接続障害情報に hostA:10000 が登録されていないかどうかを確認します。
   接続障害情報に hostA:10000 が登録されているため，hostA:10000 には接続しません。
7. コネクション B は，次の接続先として設定されている hostB:10000 に接続する前に，接続障害情報に hostB:10000 が登録されていないかどうかを確認します。
8. 接続障害情報に hostB:10000 が登録されていないため，接続を試みます。
   接続に成功します。
9. コネクション C は，構成定義に指定された hostB:10000 に接続する前に，接続障害情報に hostB:10000 が登録されていないかどうかを確認します。
10. 接続障害情報に hostB:10000 が登録されていないため，接続を試みます。
    接続に成功します。

以降，hostA:10000 の復旧が確認されるまで，Connector .NET 構成定義の<connection>要素の failureCheckInterval 属性で指定した時間ごとに，コネクション A またはコネクション B が 1.～5.の動作をします。

2.8.4 接続障害軽減機能を使用する場合の注意事項

接続障害軽減機能を使用する場合の注意事項を次に示します。

- 接続障害軽減機能は接続先の TP1/Server が複数存在する場合に有効です。接続先が一つの場合にこの機能を使用しても効果はありません。
- ネームサービスを使用する場合，TP1/Server の設定で必ずスケジュールサービスのポート番号を指定してください。ポート番号を指定しない場合，不要な障害情報が残り，メモリリソースを圧迫するおそれがあります。
- 復旧確認動作時に接続先が障害から復旧していない場合，接続を試みるコネクションには再度接続障害が発生します。また，再度接続障害が発生した時刻から復旧確認間隔の経過後に次の復旧確認動作が実行されます。
- 復旧確認動作を行わない場合，接続先を切り替えたコネクションを障害発生前の接続先へと戻すためには，アプリケーションを再起動する必要があります。
- リモート API 機能を使用した RPC の場合，常設コネクションの確立時（コネクションの生成時や問い合わせ間隔の経過による rap サーバから常設コネクション解放後の再接続時など）にだけ，接続障害軽減機能が適用されます。常設コネクションの確立後，サーバ側で障害が発生した場合は，同じ接続先への常設コネクションを持つすべてのコネクションで，サービス要求時に通信障害が検知される場合があります。サーバ側の電源切断などサーバ側で障害が発生した場合，障害検知には RPC の最大応答待ち時間（Client .NET 構成定義の<rpc>要素の watchTime 属性で指定した値）が掛かります。

- 構成定義に指定されているすべての接続先が接続障害情報に登録されている場合，再度指定された接続先の順に接続を試みます。すべての接続先に対する接続に失敗した場合はエラーとなります。
- Client .NET 構成定義の<tp1Server>要素に同じ接続先を表すホスト名と IP アドレスを用いて複数定義している場合，それぞれ別の接続先として接続障害情報に登録されます。そのため，同じ接続先を表していても，それぞれの定義を使用したコネクションが一度ずつ接続障害を検知するまでは，接続障害情報として共有されません。
- 接続障害情報にはコネクションが接続障害を検知した時点で登録されます。そのため，一つのコネクションが接続障害を検知するまでの間にほかのコネクションが同じ接続先に対して接続を試みた場合，それぞれのコネクションで障害を検知するまでの時間が掛かります。
- 一時的な障害によって接続先が接続障害情報に登録されると，TP1/Server が正常に稼働している場合でも接続先の切り替えが行われ，コネクションの接続先に偏りが発生します。この場合，Connector .NET 構成定義の<connection>要素の failureCheckInterval 属性の値を小さくしたり，TP1/Server のシステム共通定義などで ipc_backlog_count の値を大きくしたりして，偏りの継続時間や発生頻度を軽減することができます。
- 接続障害情報に登録される情報の単位は，マシンや OpenTP1 のノードではありません。ホスト名と，rap リスナー，ネームサーバ，およびスケジューラのプロセスごとに設定されたポート番号の組み合わせで管理されます。接続障害情報に登録される接続先を次の表に示します。

表 2-15　接続障害情報に登録される接続先

接続先種別	接続障害情報に登録される接続先	
	登録数	ホスト名
rap リスナー	0～Client .NET 構成定義の<tp1Server>要素の数。	Client .NET 構成定義の<tp1Server>要素の host 属性に指定した文字列。
ネームサーバ	1～Client .NET 構成定義の<tp1Server>要素の数。 ただし，Client .NET 構成定義の<tp1Server>要素の数は，TP1/Server ごとに一つとなります。	Client .NET 構成定義の<tp1Server>要素の host 属性に指定した文字列。
スケジューラ（スケジューラダイレクト機能を使用した RPC）	1～Client .NET 構成定義の<tp1Server>要素の数。 マスタスケジューラ以外にマルチスケジューラのポートを指定できます。	Client .NET 構成定義の<tp1Server>要素の host 属性に指定した文字列。
スケジューラ（ネームサービスを使用した RPC）※	1～ネームサーバへの問い合わせで得られた接続先となるスケジューラの数。 ネームサーバへの問い合わせで得られた接続先となるスケジューラの数は，TP1/Server がマルチスケジューラ機能を使用しているか使用していないかで次のように異なります。 マルチスケジューラ機能を使用している場合 各 TP1/Server で，マスタスケジューラ一つに加えて，マルチスケジューラの数も含まれます。	ネームサーバから通知された IP アドレス。

接続先種別	接続障害情報に登録される接続先	
	登録数	ホスト名
	マルチスケジューラ機能を使用していない場合 各 TP1/Server で，マスタスケジューラの一つだけになります。	

注※

スケジューラダイレクト機能を使用した RPC で，Client .NET 構成定義の<tp1Server>要素の host 属性に IP アドレスを指定した場合だけ，同じ接続先として扱われます。

2.9 WCF 連携機能

WCF 連携機能を使用すると，Windows Communication Foundation（WCF）の統一化されたプログラミングモデルから OpenTP1 にサービスを要求できます。このため，WCF を採用するシステムや WCF をサポートした Microsoft の製品から OpenTP1 へのサービス要求が容易になります。

WCF とは，.NET Framework v3.0 および.NET Framework v3.5 でサポートされている，サービス指向のアプリケーション実行基盤です。WCF は.NET Remoting や XML Web サービスなどの通信方式を統一化したプログラミングモデルを提供しています。

2.9.1 WCF 連携機能の機能概要

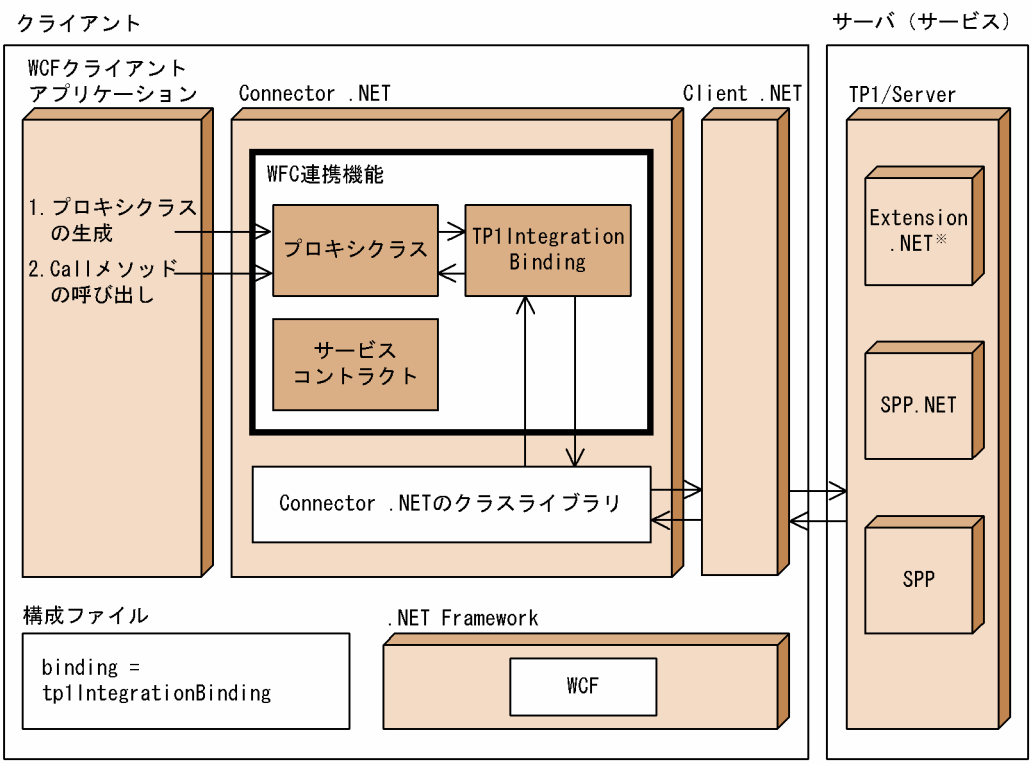
WCF では，クライアントとサービスとの間でどのように通信するかを Binding としてアプリケーション構成ファイルに定義することで，クライアントとサービス間の通信方式を決定します。WCF 連携機能では，OpenTP1 にサービスを要求できるように，WCF の Binding として TP1IntegrationBinding を提供します。

Binding に TP1IntegrationBinding を設定することで，WCF クライアントから OpenTP1 上の SPP および SPP.NET にサービスを要求できます。ただし，サービスを要求できる SPP.NET は，.NET インタフェース定義を使用していない SPP.NET だけです。

なお，WCF 連携機能は WCF クライアントから OpenTP1 にサービスを要求するための機能であり，WCF サービスの開発はできません。そのため，WCF を利用する際に必要なサービスコントラクトおよびそのプロキシクラスは，Connector .NET の TP1IntegrationBinding であらかじめ提供しています。

WCF 連携機能の概要を次の図に示します。

図 2-39　WCF 連携機能の概要



注※　使用しない場合もあります。

(1)　前提条件

(a)　前提ソフトウェア

WCF 連携機能を使用する場合，次に示すソフトウェアが必要です。

- .NET Framework v3.5 Service Pack 1

(2)　使用できるプログラム言語

WCF 連携機能を使用して UAP を開発する場合，次のプログラム言語が使用できます。

- C#
- Visual Basic

(3)　WCF 連携機能で使用できる Connector .NET の機能

WCF 連携機能を使用してサービスを要求する場合，Connector .NET の機能で使用できる機能と使用できない機能があります。

(a) 使用できる Connector .NET の機能

WCF 連携機能を使用する場合，Connector .NET の機能は Connector .NET 構成定義に定義することで使用できます。

WCF 連携機能で使用できる Connector .NET の機能と構成定義での定義を次の表に示します。

表 2-16 WCF 連携機能で使用できる Connector .NET の機能と構成定義での定義

Connector .NET の機能	使用可否	構成定義での定義
リモートプロシジャコール（RPC）	○	<client>要素
トランザクション制御機能（ローカルトランザクション）	×	−
トランザクション制御機能（分散トランザクション）	○	<distributedTransaction>要素
TCP/IP 通信機能	×	−
コネクションプーリング機能	○	<connection>要素
バッファプーリング機能	○	• <buffer>要素 • <largestBufferPool>要素 • <bufferPool>要素
TSP 自動生成機能	×	−
リソース監視機能	○	• <perfCounter>要素 • <buffer>要素の threshold 属性 • <largestBufferPool>の threshold 属性 • <bufferPool>要素の threshold 属性
接続障害軽減機能	○	<connection>要素の failureInfoSharing 属性および failureCheckInterval 属性

（凡例）

○：使用できます。

×：使用できません。

−：該当しません。

(b) サービスを要求できるサーバ UAP の種類

WCF 連携機能を使用してサービスを要求できるサーバ UAP の種類を次の表に示します。

表 2-17 サービスを要求できるサーバ UAP の種類

サーバ UAP の種類	使用可否
SPP	○
SPP.NET（.NET インタフェース定義を使用した場合）	×
SPP.NET（.NET インタフェース定義を使用しない場合）	○

サーバ UAP の種類	使用可否
MHP	×

（凡例）

○：使用できます。

×：使用できません。

(c)　サービスを要求する際の RPC 形態と種類

WCF 連携機能を使用してサービスを要求する際の RPC の形態と種類を次の表に示します。

表 2-18　サービスを要求する際の RPC の形態と種類

分類	RPC の形態と種類	使用可否
RPC の形態	同期応答型 RPC	○※1
	非同期応答型 RPC	×
	非応答型 RPC	×
	連鎖 RPC	×
RPC の種類※2	リモート API 機能を使用した RPC	○
	ネームサービスを使用した RPC	○
	スケジューラダイレクト機能を使用した RPC	○

（凡例）

○：使用できます。

×：使用できません。

注※1

プロキシクラスのメソッド（Call）を利用してサービスを要求します。

注※2

RPC の種類は，Client .NET 構成定義に定義することで自由に選択できます。

2.9.2　TP1IntegrationBinding のサービスコントラクトおよびプロキシクラス

TP1IntegrationBinding で使用するサービスコントラクトおよびそのプロキシクラスは，アセンブリ（Hitachi.OpenTP1.ServiceModel.TP1Integration.dll）で提供します。そのため，サービスコントラクトをコーディングし，プロキシクラスを生成する必要はありません。

サービスコントラクトおよびそのプロキシクラスは次の名称で提供しています。

サービスコントラクト

Hitachi.OpenTP1.ServiceModel.TP1Integration.ITP1Rpc

プロキシクラス

Hitachi.OpenTP1.ServiceModel.TP1Integration.TP1RpcClient

TP1IntegrationBinding で提供するサービスコントラクトの内容を次に示します。

(1) サービスコントラクト（C#の場合）

```
using System.ServiceModel;

namespace Hitachi.OpenTP1.ServiceModel.TP1Integration
{
   [ServiceContract(
     CallbackContract=null,
     ConfigurationName=
        "Hitachi.OpenTP1.ServiceModel.TP1Integration.ITP1Rpc",
     Name="ITP1Rpc",
     ProtectionLevel=System.Net.Security.ProtectionLevel.None,
     SessionMode=SessionMode.NotAllowed)]
   public interface ITP1Rpc
   {

      [OperationContract(
        AsyncPattern=false,
        IsOneWay=false,
        Name="Call",
        ProtectionLevel=System.Net.Security.ProtectionLevel.None,
      [TransactionFlow(TransactionFlowOption.Allowed)]
      void Call(string serviceName, byte[] inData, int inLength,
                ref byte[] outData, ref int outLength);

   }
}
```

(2) サービスコントラクト（Visual Basic の場合）

```
Imports System.ServiceModel

Namespace Hitachi.OpenTP1.ServiceModel.TP1Integration

    <ServiceContract( _
     CallbackContract:=Nothing, _
     ConfigurationName:= _
        "Hitachi.OpenTP1.ServiceModel.TP1Integration.ITP1Rpc", _
     Name:="ITP1Rpc", _
     ProtectionLevel:=Net.Security.ProtectionLevel.None, _
     SessionMode:=SessionMode.NotAllowed)> _
    Public Interface ITP1Rpc

        <OperationContract( _
         AsyncPattern:=False, _
         IsOneWay:=False, _
         Name:="Call", _
         ProtectionLevel:= _
            System.Net.Security.ProtectionLevel.None, _
```

```
        <TransactionFlow(TransactionFlowOption.Allowed)> _
        Sub [Call](ByVal service As String, _
            ByVal inData() As Byte, ByVal inLength As Integer, _
            ByRef outData() As Byte, ByRef outLength As Integer)
    End Interface

End Namespace
```

2.9.3　WCF 連携機能を使用した UAP の開発と実行

WCF 連携機能を使用した UAP の開発と実行には，Windows SDK を使用する方法と Visual Studio を使用する方法があります。

(1)　Windows SDK を使用する場合

Windows SDK を使用して，UAP を開発および実行する手順を次に示します。

1. TP1IntegrationBinding を使用した WCF のクライアントアプリケーションをコーディングします。
   コーディング方法については，「2.9.4　WCF クライアントアプリケーションのコーディング方法」を参照してください。

2. アプリケーション構成ファイルを作成します。
   WCF 連携機能を使用する場合のアプリケーション構成ファイルの設定方法については，「2.9.7　アプリケーション構成ファイルの設定方法」を参照してください。

3. アプリケーションをビルドします。
   WCF 連携機能を使用するには，次のアセンブリの参照が必要です。
   - System.ServiceModel.dll
   - Hitachi.OpenTP1.ServiceModel.TP1Integration.dll

   なお，Hitachi.OpenTP1.ServiceModel.TP1Integration.dll は，次のディレクトリに配置されています。
   〈Connector .NET のインストールディレクトリ〉¥bin

4. アプリケーションを実行します。

(2)　Visual Studio を使用する場合

Visual Studio を使用して，UAP を開発および実行する手順を次に示します。

1. WCF 連携機能を使用するプロジェクトを選択し，Visual Studio の［プロジェクト］メニューから［参照の追加］を選択します。

2. ［参照の追加］ダイアログから，コンポーネント名「TP1/Connector for .NET Framework TP1IntegrationBinding」および「System.ServiceModel.dll」選択し，追加します。

3. TP1IntegrationBinding を使用した WCF のクライアントアプリケーションをコーディングします。
   コーディング方法については，「2.9.4　WCF クライアントアプリケーションのコーディング方法」を参照してください。

4. アプリケーション構成ファイルを作成します。
   WCF 連携機能を使用する場合のアプリケーション構成ファイルの設定方法については，「2.9.7　アプリケーション構成ファイルの設定方法」を参照してください。

5. Visual Studio の［ビルド］メニューから，［ソリューションのビルド］を選択し，アプリケーションをビルドします。

6. Visual Studio の［デバッグ］メニューから，［デバッグ開始］を選択し，デバッグを実行します。

2.9.4　WCF クライアントアプリケーションのコーディング方法

WCF クライアントアプリケーションのコーディング方法には，アプリケーション構成ファイルに WCF を設定する方法と設定しない方法があります。

(1)　アプリケーション構成ファイルに WCF を設定する場合

アプリケーション構成ファイルに WCF を設定すると，設定した内容がプロキシオブジェクトによって取得され，接続先の OpenTP1 のサービスグループが決定します。

複数のサービスグループを呼び出す場合には，アプリケーション構成ファイルに接続先の OpenTP1 の設定を複数定義する必要があります。また，プロキシオブジェクトも呼び出すサービスグループの数だけ必要です。

(a)　コーディング手順

アプリケーション構成ファイルに WCF を設定する場合のコーディング手順を次に示します。

1. プロキシクラス（Hitachi.OpenTP1.ServiceModel.TP1Integration.TP1RpcClient）をインスタンス化します。
   アプリケーション構成ファイルに定義した内容を取得するため，引数には<endpoint>要素の name 属性の値を指定します。

2. 入力パラメタにデータを設定します。
   サービスの応答の領域には，C#の場合は null を，Visual Basic の場合は Nothing を設定します。

3. 必要に応じて，サービスを要求してからサービスの応答が返るまでの最大応答待ち時間を設定します。
   最大応答待ち時間については，「2.9.6　サービス要求の最大応答待ち時間の設定方法」を参照してください。

4. プロキシオブジェクトの Call メソッドを呼び出します。

5. サービスの応答を参照して，必要な処理をします。

(b) コーディング例

アプリケーション構成ファイルに次のように WCF 連携機能が設定されている場合のコーディング例を示します。

アプリケーション構成ファイル

```
<system.serviceModel>
  <client>
    <endpoint
      name="Gyoumu1"
      address="opentp1.rpc://MyProfile1/Gyoumu1"
      binding="tp1IntegrationBinding"
      contract="Hitachi.OpenTP1.ServiceModel.
                TP1Integration.ITP1Rpc"
      behaviorConfiguration="TP1BehaviorConfig">
    </endpoint>
    <endpoint
      name="Gyoumu2"
      address="opentp1.rpc://MyProfile1/Gyoumu2"
      binding="tp1IntegrationBinding"
      contract="Hitachi.OpenTP1.ServiceModel.
                TP1Integration.ITP1Rpc"
      behaviorConfiguration="TP1BehaviorConfig">
    </endpoint>
  </client>
   …
```

コーディング例（C#の場合）

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ServiceModel;
using Hitachi.OpenTP1.ServiceModel.TP1Integration;

namespace CSSample
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
       // (1) プロキシクラスのインスタンス化
       TP1RpcClient client = new TP1RpcClient("Gyoumu1");
       // (2) 入力パラメタにデータを設定
       int inLen = 24;
       int outLen = 32;
       byte[] inBuffer = new byte[inLen];
       byte[] outBuffer = null;
       // (3) サービス要求の最大応答待ち時間を設定
       client.Endpoint.Binding.SendTimeout = TimeSpan.MaxValue;
       // (4) プロキシオブジェクトのCallメソッドを呼び出す
       client.Call(
```

```
            "Service",
            inBuffer, inLen,
            ref outBuffer, ref outLen);
        // (5) サービスの応答を参照して必要な処理をする
      }
      catch (CommunicationException wcfExp)
      {
        // WCFに関連する例外
      }
      catch (Exception exp)
      {
        // その他の例外
      }
    }
  }
}
```

コーディング例（Visual Basic の場合）

```
Imports System
Imports System.Collections.Generic
Imports System.Text
Imports System.ServiceModel
Imports Hitachi.OpenTP1.ServiceModel.TP1Integration

Module VBSample

  Sub Main()
    Try
      ' (1) プロキシクラスのインスタンス化
      Dim client As TP1RpcClient = New TP1RpcClient("Gyoumu1")
      ' (2) 入力パラメタにデータを設定
      Dim inLen As Integer = 24
      Dim outLen As Integer = 32
      Dim inBuffer() As Byte = New Byte(inLen) {}
      Dim outBuffer() As Byte = Nothing
      ' (3) サービス要求の最大応答待ち時間を設定
      client.Endpoint.Binding.SendTimeout = TimeSpan.MaxValue
      ' (4) プロキシオブジェクトのCallメソッドを呼び出す
      client.Call("Service", _
             inBuffer, inLen, _
             outBuffer, outLen)
      ' (5) サービスの応答を参照して必要な処理をする
    Catch wcfExp As CommunicationException
      ' WCFに関連する例外
    Catch exp As Exception
      ' その他の例外
    End Try
  End Sub
End Module
```

(2) アプリケーション構成ファイルに WCF を設定しない場合

アプリケーション構成ファイルに WCF の設定をしない場合は，接続先の OepnTP1 の位置情報をオブジェクト（System.ServiceModel.EndpointAddress）に設定することで，呼び出す OpenTP1 のサービスグループを決定します。

複数のサービスグループを呼び出す場合には，位置情報を設定するオブジェクトが複数必要です。また，プロキシオブジェクトも呼び出すサービスグループの数だけ必要です。

なお，System で始まる名前空間に属するクラスの詳細については，.NET Framework のドキュメントを参照してください。

(a) コーディング手順

アプリケーション構成ファイルに WCF を設定しない場合のコーディング手順を次に示します。

1. System.ServiceModel.EndpointAddress クラスをインスタンス化します。
   コンストラクタの引数には，URI を指定します。URI の指定方法については，「2.9.5 接続先 OepnTP1 のサービスの位置情報の指定形式」を参照してください。
2. Hitachi.OpenTP1.ServiceModel.TP1Integration.TP1IntegrationBinding クラスをインスタンス化します。
3. プロキシクラス（Hitachi.OpenTP1.ServiceModel.TP1Integration.TP1RpcClient クラス）をインスタンス化します。
   コンストラクタの引数には，1.でインスタンス化した EndpointAddress オブジェクト，および 2.でインスタンス化した TP1IntegrationBinding オブジェクトを指定します。
4. Hitachi.OpenTP1.ServiceModel.TP1Integration.TP1IntegrationBehavior クラスをインスタンス化します。
5. プロキシオブジェクトの Endpoint.Behaviors プロパティから得られる System.Collections.Generic.KeyedByTypeCollection<System.ServiceModel.Description.IEndpointBehavior>オブジェクトの Add メソッドで，4.でインスタンス化した TP1IntegrationBehavior オブジェクトを追加します。
6. 入力パラメタにデータを設定します。
   サービスの応答の領域には，C#の場合は null を，Visual Basic の場合は Nothing を設定します。
7. 必要に応じて，サービスを要求してからサービスの応答が返るまでの最大応答待ち時間を設定します。
   最大応答待ち時間については，「2.9.6 サービス要求の最大応答待ち時間の設定方法」を参照してください。
8. プロキシオブジェクトの Call メソッドを呼び出します。
9. サービスの応答を参照して，必要な処理をします。

(b) コーディング例

アプリケーション構成ファイルに WCF の設定をしない場合のコーディング例を示します。

コーディング例（C#の場合）

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ServiceModel;
using Hitachi.OpenTP1.ServiceModel.TP1Integration;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
       // (1) EndpointAddressクラスのインスタンス化
       EndpointAddress address =
           new EndpointAddress(
               ”opentp1.rpc://MyProfile1/Gyoumu1”);
       // (2) TP1IntegrationBindingクラスのインスタンス化
       TP1IntegrationBinding binding =
           new TP1IntegrationBinding();
       // (3) プロキシクラスのインスタンス化
       TP1RpcClient client =
           new TP1RpcClient(binding, address);
       // (4) TP1IntegrationBehaviorクラスのインスタンス化
       TP1IntegrationBehavior behavior =
            new TP1IntegrationBehavior();
       // (5) KeyedByTypeCollectionオブジェクトの
       //     Addメソッドを呼び出す
       client.Endpoint.Behaviors.Add(behavior);
       // (6) 入力パラメタにデータを設定
       int inLen = 24;
       int outLen = 32;
       byte[] inBuffer = new byte[inLen];
       byte[] outBuffer = null;
       // (7) サービス要求の最大応答待ち時間を設定
       client.Endpoint.Binding.SendTimeout = TimeSpan.MaxValue;
       // (8) プロキシオブジェクトのCallメソッドを呼び出す
       client.Call(
           ”Service”,
           inBuffer, inLen,
           ref outBuffer, ref outLen);
       // (9) サービスの応答を参照して必要な処理をする
      }
      catch (CommunicationException wcfExp)
      {
        // WCFに関連する例外
      }
      catch (Exception exp)
      {
        // その他の例外
      }
    }
  }
}
```

コーディング例（Visual Basic の場合）

```
Imports System
Imports System.Collections.Generic
Imports System.Text
Imports System.ServiceModel
Imports Hitachi.OpenTP1.ServiceModel.TP1Integration

Module VBSample

  Sub Main()
    Try
      ’(1) EndpointAddressクラスのインスタンス化
      Dim address As EndpointAddress = _
          New EndpointAddress( _
              ”opentp1.rpc://MyProfile1/Gyoumu1”)
      ’(2) TP1IntegrationBindingクラスのインスタンス化
      Dim binding As TP1IntegrationBinding = _
          New TP1IntegrationBinding()
      ’(3) プロキシクラスのインスタンス化
      Dim client As TP1RpcClient = _
          New TP1RpcClient(binding, address)
      ’(4) TP1IntegrationBehaviorクラスのインスタンス化
      Dim behavior As TP1IntegrationBehavior = _
          New TP1IntegrationBehavior()
      ’(5) KeyedByTypeCollectionオブジェクトの
      ’     Addメソッドを呼び出す
      client.Endpoint.Behaviors.Add(behavior)
      ’(6) 入力パラメタにデータを設定
      Dim inLen As Integer = 24
      Dim outLen As Integer = 32
      Dim inBuffer() As Byte = New Byte(inLen) {}
      Dim outBuffer() As Byte = Nothing
      ’(7) サービス要求の最大応答待ち時間を設定
      client.Endpoint.Binding.SendTimeout = TimeSpan.MaxValue
      ’(8) プロキシオブジェクトのCallメソッドを呼び出す
      client.Call(”Service”, _
              inBuffer, inLen, _
              outBuffer, outLen)
      ’(9) サービスの応答を参照して必要な処理をする
    Catch wcfExp As CommunicationException
      ’WCFに関連する例外
    Catch exp As Exception
      ’その他の例外
    End Try
  End Sub
End Module
```

2.9.5　接続先 OepnTP1 のサービスの位置情報の指定形式

アプリケーション構成ファイルに WCF を設定しない場合は，WCF クライアントアプリケーションをコーディングするときに，接続先 OepnTP1 のサービスの位置情報を指定する必要があります。接続先

OepnTP1 のサービスの位置情報は，opentp1.rpc URI の形式で指定します。opentp1.rpc URI の形式を次に示します。

```
opentp1.rpc://プロファイルID※1/サービスグループ名※2/※3
```

注※1

WCF 連携機能を使用する場合に適用する Connector .NET 構成定義のプロファイルを文字列で指定します。

形式に指定するプロファイル ID は省略できます。省略した場合，デフォルトプロファイルの情報（<common>要素）が有効になります。

注※2

接続先の OpenTP1 のサービスグループ名を 1～31 文字の識別子（先頭がアルファベット（A～Z, a～z）で始まる英数字列）で指定します。

形式に指定するサービスグループ名は省略できません。

注※3

形式の最後に指定する「/」は省略できます。

指定例を次に示します。

【指定例 1】

条件

- プロファイル ID：GyoumuProfile1
- サービスグループ名：Gyoumu1

指定例

```
opentp1.rpc URI : opentp1.rpc://GyoumuProfile1/Gyoumu1
```

【指定例 2】

条件

- プロファイル ID：省略（<common>要素を使用する場合）
- サービスグループ名：Gyoumu2

指定例

```
opentp1.rpc URI : opentp1.rpc:///Gyoumu2
```

2.9.6　サービス要求の最大応答待ち時間の設定方法

WCF 連携機能では，サービスを要求してからサービスの応答が返るまでの最大応答待ち時間を設定できます。Hitachi.OpenTP1.ServiceModel.TP1Integration.TP1RpcClient クラスの次のどちらかのプロパティに，最大応答待ち時間を設定した System.TimeSpan オブジェクトを設定します。

- Endpoint.Binding.SendTimeout プロパティ
- InnerChannel.OperationTimeout プロパティ

上記のプロパティに最大応答待ち時間を設定しなかった場合，最大応答待ち時間は WCF のデフォルト値となります。プロパティの詳細については，.NET Framework のドキュメントを参照してください。

最大応答待ち時間として設定できる値を次の表に示します。

表 2-19　最大応答待ち時間として設定できる値

設定できる値	説明
TimeSpan.MaxValue	サービスの応答を受信するまで無限に待ちます。
TimeSpan(0,0,0)	Client.NET 構成定義の<rpc>要素の watchTime 属性で指定した値に従います。Client.NET 構成定義については，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。
TimeSpan(0,0,1)から TimeSpan(0,0,65535)※	設定した時間（秒）までサービスの応答を待ちます。1～65535（秒）まで指定できます。

注※

System.TimeSpan に設定できる時間は，秒だけでなく，日，時間，分，ミリ秒単位でも設定できますが，この値は秒単位で 1 から 65535 の範囲内になるように設定してください。

2.9.7　アプリケーション構成ファイルの設定方法

WCF 連携機能で TP1IntegrationBinding を使用する場合の WCF の設定方法について説明します。なお，WCF の設定方法の詳細については，.NET Framework のドキュメントを参照してください。また，Connector .NET 構成定義の詳細については，「3.　構成定義」を参照してください。

WCF 連携機能を使用する場合，呼び出すサービスおよびクライアントの設定を，アプリケーション構成ファイルの<system.serviceModel>要素に記述します。

形式

```
<system.serviceModel>
  <client>          ┐
    :               ├ (1)
  </client>         ┘
  <bindings>        ┐
    :               ├ (2)
  </bindings>       ┘
  <behaviors>       ┐
    :               ├ (3)
  </behaviors>      ┘
  <extensions>      ┐
    :               ├ (4)
  </extensions>     ┘
</system.serviceModel>
```

形式の(1)～(4)は，次に説明する番号に対応しています。

(1)　<client>要素に定義する内容

(a)　形式

```
<client>
  <endpoint name="<endpoint>要素の名称"
            address="呼び出したいOpenTP1のサービスの位置情報"
            binding="tp1IntegrationBinding"
            contract="WCF連携機能で使用するサービスコントラクト"
            behaviorConfiguration="<behavior>要素のname属性の値"
            bindingConfiguration="">
  </endpoint>
</client>
```

(b)　要素

■ endpoint

WCF のクライアント側の endpoint を設定する要素です。

この要素は省略できません。

(c)　属性

■ name

<endpoint>要素の名称を指定します。

■ address

呼び出したい OpenTP1 のサービスの位置情報を opentp1.rpc URI の形式で指定します。

opentp1.rcp URI の形式および指定例については，「2.9.5　接続先 OepnTP1 のサービスの位置情報の指定形式」を参照してください。

この属性は省略できません。

■ binding

「tp1IntegrationBinding」を指定します。

この属性は省略できません。また，大文字と小文字が区別されるため注意してください。

■ contract

WCF 連携機能で使用するサービスコントラクトを完全限定名で指定します。

この属性は省略できません。

■ behaviorConfiguration

<tp1Behavior>要素を含む<behavior>要素を指定します。この属性は，<behavior>要素の name 属性と同じ値を指定してください。

この属性は省略できません。

■ bindingConfiguration

WCF 連携機能では使用しません。この属性は省略するか，または空文字を指定してください。

(2) <bindings>要素に定義する内容

(a) 形式

```
<bindings>
  <tp1IntegrationBinding />
</bindings>
```

(b) 要素

■ tp1IntegrationBinding

tp1IntegrationBinding の Binding について設定する要素です。tp1IntegrationBinding を設定する場合，「<tp1IntegrationBinding />」と記述します。

この要素は省略できません。

この要素は，属性および子要素を持ちません。

(3) <behaviors>要素に定義する内容

(a) 形式

```
<behaviors>
  <endpointBehaviors>
    <behavior name="<behavior>要素を一意に識別するための名称">
      <tp1Behavior />
```

```
    </behavior>
  </endpointBehaviors>
</behaviors>
```

(b) 要素

■ endpointBehaviors

子要素として，<behavior>要素を定義します。

この要素は省略できません。

■ behavior

Binding の検証を設定する要素です。

この要素は省略できません。

子要素として，<tp1Behavior>要素を定義します。

■ tp1Behavior

tp1IntegrationBehavior について設定する要素です。tp1Behavior を設定する場合，「<tp1Behavior />」と記述します。

この要素は省略できません。

この要素は，属性および子要素を持ちません。

(c) 属性

■ name

<behavior>要素を一意に識別するための名称を指定します。この属性は，<endpoint>要素の behaviorConfiguration 属性と同じ値を指定してください。

この属性は省略できません。

(4) <extensions>要素に定義する内容

(a) 形式

```
<extensions>
  <behaviorExtensions>
    <add name="tp1Behavior"
         type="Hitachi.OpenTP1.ServiceModel.TP1Integration.
               TP1IntegrationBehaviorElement,
               Hitachi.OpenTP1.ServiceModel.TP1Integration,
               Version=7.0.0.0, Culture=neutral,
               PublicKeyToken=2440cf5f0d80c91c" />
  </behaviorExtensions>
```

```
  <bindingExtensions>
    <add name="tp1IntegrationBinding"
         type="Hitachi.OpenTP1.ServiceModel.TP1Integration.
               TP1IntegrationBindingCollectionElement,
               Hitachi.OpenTP1.ServiceModel.TP1Integration,
               Version=7.0.0.0, Culture=neutral,
               PublicKeyToken=2440cf5f0d80c91c"/>
  </bindingExtensions>
</extensions>
```

(b) 要素

■ extensions

binding や behavior を拡張するための設定をする要素です。

この要素は省略できません。

子要素として，<behaviorExtensions>要素および<bindingExtensions>要素を定義します。

■ behaviorExtensions

<tp1Behavior>要素をアプリケーション構成ファイルに定義できるように機能を拡張するための要素です。

この要素は省略できません。

子要素として，<add>要素を次のように定義します。

```
<behaviorExtensions>
  <add name="tp1Behavior"
       type="Hitachi.OpenTP1.ServiceModel.TP1Integration.
             TP1IntegrationBehaviorElement,
             Hitachi.OpenTP1.ServiceModel.TP1Integration,
             Version=7.0.0.0, Culture=neutral,
             PublicKeyToken=2440cf5f0d80c91c" />
</behaviorExtensions>
```

■ bindingExtensions

<tp1IntegrationBinding>要素をアプリケーション構成ファイルに定義できるように機能を拡張するための要素です。

この要素は省略できません。

子要素として，<add>要素を次のように定義します。

```
<bindingExtensions>
  <add name="tp1IntegrationBinding"
       type="Hitachi.OpenTP1.ServiceModel.TP1Integration.
             TP1IntegrationBindingCollectionElement,
             Hitachi.OpenTP1.ServiceModel.TP1Integration,
             Version=7.0.0.0, Culture=neutral,
```

```
            PublicKeyToken=2440cf5f0d80c91c”/>
</bindingExtensions>
```

(5)　アプリケーション構成ファイルの設定例

WCF 連携機能で TP1IntegrationBinding を使用する場合のアプリケーション構成ファイルの設定例を次に示します。WCF の設定以外の記述は，Connector .NET 構成定義です。Connector .NET 構成定義については，「3.　構成定義」を参照してください。

```
<configuration>
  <configSections>
    <section
      name=”hitachi.opentp1.connector”
      type=”Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,
            Culture=neutral,PublicKeyToken=2440cf5f0d80c91c,
            Custom=null”   />
    <section
      name=”hitachi.opentp1.client”
      type=”Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,
            Culture=neutral,PublicKeyToken=2440cf5f0d80c91c,
            Custom=null”   />
  </configSections>

  <!-- WCFの設定 -->

  <system.serviceModel>
    <client>
      <endpoint
        name=”Gyoumu1”
        address=”opentp1.rpc://MyProfile1/Gyoumu1”
        binding=”tp1IntegrationBinding”
        contract=”Hitachi.OpenTP1.ServiceModel
                  .TP1Integration.ITP1Rpc”
        behaviorConfiguration=”TP1BehaviorConfig”>
      </endpoint>
      <endpoint
        name=”Gyoumu2”
        address=”opentp1.rpc:///Gyoumu2”
        binding=”tp1IntegrationBinding”
        contract=”Hitachi.OpenTP1.ServiceModel
                  .TP1Integration.ITP1Rpc”
        behaviorConfiguration=”TP1BehaviorConfig”>
      </endpoint>
    </client>

    <bindings>
      <tp1IntegrationBinding />
    </bindings>

    <behaviors>
      <endpointBehaviors>
        <behavior name=”TP1BehaviorConfig”>
          <tp1Behavior />
```

```
      </behavior>
    </endpointBehaviors>
  </behaviors>

  <extensions>
    <behaviorExtensions>
      <add
        name="tp1Behavior"
        type="Hitachi.OpenTP1.ServiceModel.TP1Integration
              .TP1IntegrationBehaviorElement,
              Hitachi.OpenTP1.ServiceModel.TP1Integration,
              Version=7.0.0.0, Culture=neutral,
              PublicKeyToken=2440cf5f0d80c91c" />
    </behaviorExtensions>
    <bindingExtensions>
      <add
        name="tp1IntegrationBinding"
        type="Hitachi.OpenTP1.ServiceModel.TP1Integration
              .TP1IntegrationBindingCollectionElement,
              Hitachi.OpenTP1.ServiceModel.TP1Integration,
              Version=7.0.0.0, Culture=neutral,
              PublicKeyToken=2440cf5f0d80c91c"/>
    </bindingExtensions>
  </extensions>
</system.serviceModel>

<!-- Connector .NETの構成定義-->

<hitachi.opentp1.connector>
  <common>
  <client conf=""/>
    <connection pooled="20" active="100"
                threshold="80" watchtime="120"
                failureInfoSharing="true"
                failureCheckInterval="300">
      <occupation pooled="10"/>
    </connection>
    <log destination="c:¥temp¥connectorn"
         fileSize="1048576" level="1"/>
    <buffer pooling="true" create="false">
      <largestBufferPool maxCount="5" threshold="100" />
      <bufferPool size="102400" maxCount="50"
                  threshold="120" />
      <bufferPool size="10240" maxCount="50"
                  threshold="150" />
    </buffer>
    <perfCounter use="true" />
    <option maxMessageSize="8" />
  </common>
  <profile id="MyProfile1">
    <client conf="clinetProfile1"/>
    <connection>
      <occupation pooled="2"/>
    </connection>
  </profile>
</hitachi.opentp1.connector>

<hitachi.opentp1.client>
```

```
    <common>
      <tp1Server host="hostA" />
      <rpc use="nam" watchTime="0" />
      <nameService port="10000" />
    </common>
    <profile id="traceMode">
      <errTrace use="true" path="c:¥temp¥clientn"
                fileSize="100000" />
      <methodTrace use="true" path="c:¥temp¥clientn"
                   fileSize="100000" />
      <uapTrace use="true" path="c:¥temp¥clientn"
                fileSize="100000" />
      <dataTrace use="true" path="c:¥temp¥clientn"
                 fileSize="100000" />
    <profile id="clinetProfile1">
      <tp1Server host="hostA" />
      <rpc use="nam" watchTime="0" />
      <nameService port="10000" />
    </profile>
  </hitachi.opentp1.client>

</configuration>
```

2.9.8　WCF 連携機能のサンプルプログラムの使用方法

Connector .NET では，WCF 連携機能を使用するためのサンプルプログラムを提供しています。WCF 連携機能のサンプルプログラムについては，「4.8　サンプルプログラムの使用方法」を参照してください。

2.10　環境設定

Connector .NET のインストール後に必要な，環境設定やセキュリティポリシーの設定などについて説明します。

2.10.1　インストール後の環境設定

運用コマンドを使用する場合は，環境変数 PATH に〈インストールディレクトリ〉¥bin を追加してください。

Visual Studio で UAP を開発する場合は，必要に応じて次に示すアセンブリを UAP のプロジェクトの参照設定に追加してください。

- 〈インストールディレクトリ〉¥bin¥Hitachi.OpenTP1.Connector.dll
- 〈インストールディレクトリ〉¥bin¥Hitachi.OpenTP1.Client.dll
- 〈インストールディレクトリ〉¥bin¥Hitachi.OpenTP1.ServiceModel.TP1Integration.dll

なお，次のアセンブリはインストール時にグローバルアセンブリキャッシュ（GAC）に登録されます。

- 〈インストールディレクトリ〉¥bin¥Hitachi.OpenTP1.Connector.dll
- 〈インストールディレクトリ〉¥bin¥Hitachi.OpenTP1.ServiceModel.TP1Integration.dll

2.10.2　セキュリティポリシーの設定

Connector .NET をインストールおよび実行することで，インストールおよび実行した OS 上の.NET Framework のコードアクセスセキュリティポリシーを変更してしまうことはありません。

Connector .NET の機能を利用するためには，次に示すファイルに必要なアクセス許可が与えられていなければなりません。必要なアクセス許可が与えられていない場合は，コマンドやアプリケーションが実行できません。

(1)　.NET Framework のコードアクセスセキュリティポリシーの影響を受けるファイル（アセンブリ）

- 〈インストールディレクトリ〉¥bin¥if2cstub.exe
- 〈インストールディレクトリ〉¥bin¥spp2cstub.exe
- 〈インストールディレクトリ〉¥bin¥if2tsp.exe
- 〈インストールディレクトリ〉¥bin¥spp2tsp.exe
- 〈インストールディレクトリ〉¥bin¥cnnsetpf.exe

- 〈インストールディレクトリ〉¥bin¥cnnnidgen.exe
- 〈インストールディレクトリ〉¥bin¥cnntrsls.exe
- 〈インストールディレクトリ〉¥bin¥Hitachi.OpenTP1.Connector.dll
- 〈インストールディレクトリ〉¥bin¥Hitachi.OpenTP1.ServiceModel.TP1Integration.dll
- 〈インストールディレクトリ〉¥trs¥cnntrs.exe
- Connector .NET のアプリケーションのアセンブリ，および Connector .NET のアセンブリを参照するすべてのアセンブリ

(2) 必要なアクセス許可

「(1) .NET Framework のコードアクセスセキュリティポリシーの影響を受けるファイル（アセンブリ)」で示したすべてのファイル（アセンブリ）には，完全信頼を指定する必要があります。

2.10.3 メッセージの出力先

Connector .NET が提供する運用コマンドが出力するメッセージは，標準出力または標準エラー出力に出力されます。クラスライブラリが出力するメッセージは，Connector .NET ログファイルに出力されます。

Connector .NET ログファイルの出力先は，構成ファイルのプロパティで指定できます。また，TP1ConnectionManager クラスの LogWriter プロパティに独自の TextWriter を設定することによって，Connector .NET ログファイル以外に独自に出力先を設定できます。

この場合の出力形式はログファイルと同じになります。

3 構成定義

この章では，Connector .NET で使用する構成定義について説明します。

構成ファイルの形式

Connector .NET 構成定義は，アプリケーション構成ファイル，またはマシン構成ファイルにカスタム構成セクションとして記述します。

カスタム構成セクションの宣言

形式

```
<configSections>
  <section
    name="hitachi.opentp1.connector"
    type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
          Hitachi.OpenTP1.Client,Version=7.0.0.0,Culture=neutral,
          PublicKeyToken=2440cf5f0d80c91c,Custom=null"/>
  <section
    name="hitachi.opentp1.client"
    type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
          Hitachi.OpenTP1.Client,Version=7.0.0.0,Culture=neutral,
          PublicKeyToken=2440cf5f0d80c91c,Custom=null"/>
</configSections>
```

説明

カスタム構成セクション宣言の<section>要素は，<configSections>要素の子要素として，各アプリケーション構成ファイルに記述するか，マシン構成ファイルに記述します。マシン構成ファイルに記述した場合，各アプリケーション構成ファイルに記述する必要はありません。

カスタム構成セクション宣言は，必ず<hitachi.opentp1.connector>要素よりも前に記述してください。

構成定義の記述

形式

```
<hitachi.opentp1.connector>
  <common>
     〔〔<構成定義要素名〔〔属性名="属性値"〕…〕/>〕…〕
  </common>
   〔〔<profile id="プロファイルID">
     〔〔<構成定義要素名〔〔属性名="属性値"…〕…〕/>〕…〕
      </profile>〕…〕
</hitachi.opentp1.connector>
```

説明

<hitachi.opentp1.connector>要素の子要素として，<common>要素と<profile>要素が記述できます。<profile>要素は複数記述できます。<common>要素または<profile>要素の子要素として，各構成定義要素が記述できます。

<profile>要素が指定されていない，または指定された<profile>要素内に該当する構成定義要素が記述されていない場合は，<common>要素内に記述された構成定義要素が有効になります。この構成定義要素をデフォルトプロファイルと呼びます。

<common>要素内および<profile>要素内に，同じ構成定義要素が記述されている場合は，<profile>要素内の構成定義要素が有効になります。

<common>要素内または一つの<profile>要素内に，複数定義できない構成定義要素が複数定義されている場合は，最後に記述した構成定義要素が有効になります。

なお，必ず記述しなくてはならない構成定義要素が<common>要素内および<profile>要素内に記述されていない場合は，実行時にエラーが発生します。

TP1ConnectionManager クラスのコンストラクタで，引数にプロファイル ID を指定していない，または空文字列（""）を指定した場合は，<common>要素内に記述された構成定義要素が有効になります。TP1ConnectionManager クラスのコンストラクタで引数にプロファイル ID を指定した場合は，該当する<profile>要素内の構成定義要素が有効になります。

相手先のサーバ別，およびリソース制限別に<profile>要素を記述しておくと便利です。

外部ファイルでの構成定義の記述

形式

```
<hitachi.opentp1.connector import="絶対パスの外部XMLファイル名">
</hitachi.opentp1.connector>
```

外部ファイルの形式

```
<hitachi.opentp1.connector>
  <common>
     〔〔<構成定義要素名 〔〔属性名="属性値"〕…〕/>〕…〕
  </common>
   〔〔<profile id="プロファイルID">
     〔〔<構成定義要素名 〔〔属性名="属性値"…〕…〕/>〕…〕
      </profile>〕…〕
</hitachi.opentp1.connector>
```

説明

<hitachi.opentp1.connector>要素の import 属性を指定すると，外部ファイルで定義した構成定義を読み込ませることができます。複数のアプリケーションで同じ構成定義を使用したい場合に指定します。

外部ファイルは，カスタム構成セクションの<hitachi.opentp1.connector>要素をルート要素とする XML ファイルで指定します。また，import 属性には外部ファイルのパスを絶対パスで指定してください。

【外部ファイルの指定例】

c:¥MyApp¥tp1config.xml

注意事項

- 構成定義要素の属性値として指定する true および false には，大文字と小文字の区別はありません。
- 構成定義要素の属性に不正な値を指定した場合，デフォルト値が存在するものはデフォルト値が適用されます。

hitachi.opentp1.connector

形式

```
<hitachi.opentp1.connector>
  <common>
    <構成定義要素名 属性名="属性値"/>
  </common>
  <profile id="プロファイルID">
    <構成定義要素名 属性名="属性値"/>
  </profile>
</hitachi.opentp1.connector>
```

説明

Connector .NET 構成ファイルの構成セクション名です。

<hitachi.opentp1.connector>要素の子要素として，<common>要素と<profile>要素が指定できます。

<common>要素は省略できません。必ず<hitachi.opentp1.connector>要素内に<common>要素を一つだけ指定してください。

<profile>要素は省略できます。

common

形式

```
<common>
  <構成定義要素名 属性名=”属性値”/>
</common>
```

説明

構成定義要素を指定します。

この要素内では構成定義要素の指定を省略できます。

また，この要素内にだけ指定できる構成定義要素，属性名，および属性値もあります。

この要素は省略できません。

profile

形式

```
<profile id=”プロファイルID”>
  <構成定義要素名 属性名=”属性値”/>
</profile>
```

説明

構成定義要素を指定します。

この要素内では構成定義要素の指定を省略できます。

この要素内で指定していない構成定義要素，属性名，および属性値については，<common>要素内で指定した構成定義要素，属性名，および属性値が有効になります。

この要素は省略できます。

属性

●id="プロファイル ID"　～〈文字列〉

プロファイルを一意に識別するための ID を指定します。

<profile>要素を指定した場合，この属性は省略できません。

トランザクションリカバリサービスの構成定義ではこの属性の値にコンマ（,）を使用しないでください。

client

形式

```
<client conf=”Client .NET構成定義のプロファイルID”/>
```

説明

Client .NET 構成定義のプロファイルを指定します。

この要素は省略できます。省略した場合，Client .NET 構成定義の<common>要素内に記述された構成定義要素が有効になります。Client .NET 構成定義については，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。

属性

●conf="Client .NET 構成定義のプロファイル ID"　～〈文字列〉

Client .NET 構成定義のプロファイル ID を指定します。

この属性で指定する Client .NET 構成定義の<profile>要素内に，<tp1Server>要素が記述されていなければなりません。

また，リモート API 機能を使用した RPC を行う場合は，<rapService>要素の autoConnect 属性に true を指定してください。

この属性は省略できます。省略した場合，Client .NET 構成定義の<common>要素内に記述された構成定義要素が有効になります。

Client .NET 構成定義については，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。

記述例

```
<client conf=”server2”/>
```

connection

形式

```
<connection pooled="Connector .NETが管理する
                    コネクションプールの最大数"
            active="最大同時使用コネクション数"
            threshold="コネクション数監視時のしきい値"
            watchtime="コネクション取得要求最大応答待ち時間"
            failureInfoSharing="true|false"
            failureCheckInterval="復旧確認動作間隔"/>
```

説明

コネクションに関する指定をします。

この要素は省略できます。

属性

●pooled="Connector .NET が管理するコネクションプールの最大数"　～〈符号なし整数〉((0～4096))《64》

Connector .NET が管理するコネクションプールの最大数を指定します。

この属性は省略できます。

また，<profile>要素内にこの属性を指定する場合，値は指定できません。

●active="最大同時使用コネクション数"　～〈符号なし整数〉((1～4096))《64》

Connector .NET 使用時の最大同時使用コネクション数を指定します。コネクションプールの最大数より大きな値を指定できます。コネクションプールの最大数を超えた分のコネクションはプールされません。

この属性は省略できます。

また，<profile>要素内にこの属性を指定する場合，値は指定できません。

●threshold="コネクション数監視時のしきい値"　～〈符号なし整数〉((0～100))《0》（単位：%）

最大同時使用コネクション数に対する，使用中コネクション数の比率のしきい値を指定します。指定した値を超えた場合は，Connector .NET のログファイルに警告メッセージが出力されます。

この属性は省略できます。

また，0 を指定した場合は警告メッセージは出力されません。

●watchtime="コネクション取得要求最大応答待ち時間"　～〈符号なし整数〉((0～65535))《180》(単位：秒)

コネクション取得要求を行ってから実際にコネクションを取得するまでの最大待ち時間を指定します。指定時間を過ぎてもコネクションを取得ができない場合は，エラーが発生します。

この属性は省略できます。

0 を指定した場合は，コネクションを取得するまで待ち続けます。

●failureInfoSharing="true|false"　～《false》

接続障害軽減機能の使用するかどうかを指定します。

この属性は省略できます。

true：接続障害軽減機能を使用します。

false：接続障害軽減機能を使用しません。

●failureCheckInterval="復旧確認動作間隔"　～〈符号なし整数〉((0～2147483647))《300》(単位：秒)

接続障害が発生した接続先に対して，復旧確認動作を行う間隔を指定します。

この属性は省略できます。

0 を指定した場合，復旧確認動作は行いません。

記述例

```
<connection pooled=”20”
            active=”100”
            threshold=”80”
            watchtime=”120”
            failureInfoSharing=”true”
            failureCheckInterval=”300”/>
```

occupation

形式

```
<occupation pooled=”プロファイルが占有できる
                    コネクションプールの最大数”/>
```

説明

コネクションの占有に関する指定をします。

この要素は省略できます。

属性

●pooled="プロファイルが占有できるコネクションプールの最大数"　～〈符号なし整数〉((0～4096))

Connector .NET が管理するコネクションプールのうち，このプロファイルで占有できるコネクションプールの最大数を指定します。コネクションプールの最大数より大きな値は指定できません。

この属性は省略できます。省略した場合，この属性の値はコネクションプールの最大数と同じ値になります。

なお，この属性は<connection>要素を指定した場合にだけ有効です。その場合，<connection>要素内に指定してください。

記述例

```
<occupation pooled=”10”/>
```

tcpip

形式

```
<tcpip keepAlive=”true|false”/>
```

説明

コネクションに関する指定をします。

この要素は省略できます。

属性

●keepAlive="true | false"　～《true》

TCP/IP 通信機能を利用する場合に，TcpipConnection クラスの Dispose メソッドによってコネクションをコネクションプールに戻す際に，物理コネクションを解放するかどうかを指定します。

この属性は省略できます。

true：物理コネクションを解放しないで，そのままコネクションプールに戻します。

false：物理コネクションを解放してから，コネクションプールに戻します。

コネクションプールに戻されないで破棄される場合は，この指定に関係なく物理コネクションは解放されます。

記述例

```
<tcpip keepAlive=”true”/>
```

log

形式

```
<log destination="Connector .NETログファイル作成ディレクトリ"
     fileSize="Connector .NETログファイルサイズ"
     level="0|1|2|3"/>
```

説明

Connector .NET が用意しているログ出力機能に関する指定をします。

TP1ConnectionManager メソッドの LogWriter プロパティを設定した場合，この要素は無視されます。

この要素は省略できます。

また，この要素は<common>要素内でだけ有効です。

属性

●destination="Connector .NET ログファイル作成ディレクトリ"　～〈文字列〉

Connector .NET ログファイルを作成するディレクトリを絶対パス名で指定します。

指定したディレクトリに，ファイル名が tp1connector1.log および tp1connector2.log のファイルを作成し，ラウンドロビン方式で記録します。

指定するディレクトリは事前に作成し，アプリケーションごとに異なるディレクトリを指定してください。指定したディレクトリが存在しない，または書き込みができない場合，Connector .NET ログファイルは作成されません。

また，ASP.NET から利用している場合は，書き込み権限のあるディレクトリが限られていますので注意が必要です。

この属性は省略できます。この属性を省略した場合，Connector .NET ログファイルは作成されません。

●fileSize="Connector .NET ログファイルサイズ"　～〈符号なし整数〉((1048576～2147483637))《1048576》(単位：バイト)

取得する一つの Connector .NET ログファイルのサイズを指定します。

この属性は省略できます。

●level="0 | 1 | 2 | 3"　～《1》

ログレベルを指定します。

ログレベルには Error，Warning，Information，Detail の四つがあります。取得するログレベルを指定すると，指定したレベルに対応するメッセージインジケータのログだけを取得できます。

この属性は省略できます。

指定する数字によって，次のログレベルのメッセージが出力されます。

0：Error

1：Error，および Warning

2：Error，Warning，および Information

3：Error，Warning，Information，および Detail

ログレベルに 3 を指定した場合，ログレベル 2 の情報に加えて，リソース（コネクションおよびバッファ）の使用状況に関する情報が出力されます。ログレベル 2 およびログレベル 3 を指定した場合に出力されるメッセージを次に示します。メッセージの詳細については，マニュアル「OpenTP1 メッセージ」を参照してください。

ログレベル 2

- KFCA32400-I
- KFCA32401-I
- KFCA32409-I
- KFCA32410-I
- KFCA32412-I
- KFCA32413-I
- KFCA32414-I
- KFCA32415-I
- KFCA32471-I
- KFCA32472-I
- KFCA32473-I
- KFCA32474-I
- KFCA32475-I
- KFCA32476-I
- KFCA32480-I
- KFCA32495-I

ログレベル 3

- KFCA32402-I
- KFCA32403-I
- KFCA32404-I

- KFCA32405-I
- KFCA32406-I
- KFCA32407-I
- KFCA32477-I
- KFCA32478-I

記述例

```
<log destination=
       "C:¥Documents and Settings¥COMPUTERNAME¥ASPNET¥Log¥App1"
     fileSize="1048576"
     level="0"/>
```

buffer

形式

```
<buffer pooling=”true|false”/>
```

説明

バッファに関する指定をします。

この要素は省略できます。

属性

●pooling="true | false"　～《false》

バッファプーリング機能を使用する場合に指定します。

この属性は省略できます。

true：バッファプーリング機能を使用します。

false：バッファプーリング機能を使用しません。

記述例

```
<buffer pooling=”true”/>
```

largestBufferPool

形式

```
<largestBufferPool maxCount=”最大バッファプール内のバッファ数”
                   threshold=”バッファ数監視時のしきい値”/>
```

説明

最大バッファプールに関する指定をします。

バッファサイズの最大値は 1 メガバイトです。ただし，RPC 送受信メッセージの最大長拡張機能を使用する場合，<option>要素の maxMessageSize 属性で指定した値（×1048576 バイト）になります。

この要素は省略できます。

属性

●maxCount="最大バッファプール内のバッファ数"　　～〈符号なし整数〉((2～1024))《2》

最大バッファプール内のバッファ数を指定します。

この属性は省略できます。

なお，この属性は<buffer>要素の pooling 属性に true を指定した場合にだけ有効です。その場合，<buffer>要素内に指定してください。

●threshold="バッファ数監視時のしきい値"　　～〈符号なし整数〉((0～10000))《0》（単位：%）

maxCount 属性で指定したバッファ数に対する，使用中バッファ数の比率のしきい値を指定します。指定した値を超えた場合は，Connector .NET のログファイルに警告メッセージが出力されます。

この属性は省略できます。

100 以上を指定した場合は，maxCount 属性で設定した数を超えてバッファが生成されたことを意味します。0 を指定した場合は，Connector .NET のログファイルに警告メッセージは出力されません。

記述例

```
<largestBufferPool maxCount=”5”
                   threshold=”100”/>
```

bufferPool

形式

```
<bufferPool size="バッファサイズ"
            maxCount="バッファプール内のバッファ数"
            threshold="バッファ数監視時のしきい値"/>
```

説明

バッファプールに関する指定をします。

この要素は省略できます。

属性

●size="バッファサイズ"　～〈符号なし整数〉((1～1048575))《4096》(単位：バイト)

バッファプール内に作られるバッファのサイズを指定します。

同じバッファサイズを複数の<bufferPool>要素で指定した場合は，二つ目以降の要素の指定は無視されます。

この属性は省略できます。

また，この属性は<buffer>要素の pooling 属性に true を指定した場合にだけ有効です。その場合，<buffer>要素内に指定してください。

なお，指定できるバッファサイズの最大は 1048575 バイトですが，RPC 送受信メッセージの最大長拡張機能を使用する場合，指定した値から 1 を引いた値までバッファサイズ（最大で 8388607 バイト）を指定できます。

●maxCount="バッファプール内のバッファ数"　～〈符号なし整数〉((2～1024))《64》

バッファプール内のバッファ数を指定します。

この属性は省略できます。

なお，この属性は<buffer>要素の pooling 属性に true を指定した場合にだけ有効です。その場合，<buffer>要素内に指定してください。

●threshold="バッファ数監視時のしきい値"　～〈符号なし整数〉((0～10000))《0》(単位：%)

maxCount 属性で指定したバッファ数に対する，使用中バッファ数の比率のしきい値を指定します。指定した値を超えた場合は，Connector .NET のログファイルに警告メッセージが出力されます。

この属性は省略できます。

100 以上を指定した場合は，maxCount 属性で設定した数を超えてバッファが生成されたことを意味します。0 を指定した場合は，Connector .NET のログファイルに警告メッセージは出力されません。

記述例

```
<bufferPool size="102400"
            maxCount="50"
            threshold="120"/>
```

option

形式

```
<option maxMessageSize="RPCメッセージの最大長"/>
```

説明

RPC 送受信メッセージの最大長拡張機能に関する指定をします。

この要素は省略できます。

属性

●maxMessageSize="RPC メッセージの最大長"　～〈符号なし整数〉((1～8))《1》(単位：メガバイト)

TP1Connection クラスの Execute メソッドで送受信できるユーザメッセージの最大長を指定します。

この属性は省略できます。

この属性を指定する場合は，Client .NET 構成定義の<rpc>要素の maxMessageSize 属性もあわせて指定してください。なお，Connector .NET の maxMessageSize 属性で指定する値は，Client .NET の maxMessageSize 属性で指定する値以下にしてください。

記述例

```
<option maxMessageSize="8"/>
```

perfCounter

形式

```
<perfCounter use=”true|false”/>
```

説明

リソース監視機能に関する指定をします。

リソース（コネクションおよびバッファ）に関する情報を Windows のパフォーマンスカウンタに出力することで，それらの使用状況を監視できます。

この要素は省略できます。

属性

●use="true | false"　～《false》

コネクションおよびバッファの使用状況をパフォーマンスカウンタに出力するかどうかを指定します。

この属性は省略できます。

true：パフォーマンスカウンタに出力します。

false：パフォーマンスカウンタに出力しません。

記述例

```
<perfCounter use=”true”/>
```

distributedTransaction

形式

```
<distributedTransaction use="true | false"
                        optimize1PC="true | false"
                        nodeId
                          ="MSDTC連携機能を使用するOS環境を
                            一意に識別するためのノード識別子"
                        recoverRetryInterval
                          ="トランザクションの回復処理失敗時の
                            再試行間隔"/>
```

説明

MSDTC 連携機能の情報を設定します。

この要素はアプリケーション専用です。

この要素をトランザクションリカバリサービスの構成定義に記述した場合は無視されます。

属性

●use="true | false"　～《false》

MSDTC 連携機能を使用するかどうかを指定します。

この属性は省略できます。

true：MSDTC 連携機能を使用します。

false：MSDTC 連携機能を使用しません。

●optimize1PC="true | false"　～《false》

単一フェーズコミット最適化を行うかどうかを指定します。

この属性は省略できます。

true：単一フェーズコミット最適化を行います。

false：単一フェーズコミット最適化を行いません。

●nodeId="MSDTC 連携機能を使用する OS 環境を一意に識別するためのノード識別子"　～〈文字列〉

MSDTC 連携機能を使用する OS 環境を一意に識別するためのノード識別子を指定します。

MSDTC 連携機能で使用するノード識別子生成コマンド（cnnnidgen）で生成したノード識別子を文字列で指定します。

アルファベットの大文字と小文字は区別しません。

同一 OS 上で動作するトランザクションリカバリサービスの構成定義で指定する<recoveryService>要素の nodeId の値と必ず一致させてください。

この値が<recoveryService>要素の nodeId の値と一致しない，またはこの値を省略した場合は，アプリケーションのトランザクション内で最初のサービス要求時にエラーとなります。

●recoverRetryInterval="トランザクションの回復処理失敗時の再試行間隔"　～〈符号なし整数〉((1～65535))《10》（単位：秒）

アプリケーションのバックグラウンドスレッドがトランザクションの回復処理に失敗した場合の再試行間隔を指定します。

この属性は省略できます。

記述例

```
<distributedTransaction use=”true”
                        optimize1PC=”true”
                        nodeId=
                          ”18AF5B57-FED5-4522-9B0B-94FBDDDD3EA4”
                        recoverRetryInterval=”10”/>
```

recoveryService

形式

```
<recoveryService nodeId
                   ="MSDTC連携機能を使用するOS環境を
                     一意に識別するためのノード識別子"
                 appDomainCheckInterval
                   ="アプリケーションドメイン監視間隔"
                 recoverRetryInterval
                   ="トランザクションリカバリサービス
                     回復処理失敗時の再試行間隔"
                 recoverCheckInterval
                   ="未決着トランザクションの存在を確認する間隔"
                 recoverCheckCount
                   ="未決着トランザクションの存在を確認する回数"
                 profiles
                   ="トランザクションリカバリサービス構成定義の
                     プロファイルID"
                 rmidStoragePath
                   ="RMID格納ディレクトリ"/>
```

説明

MSDTC 連携機能の情報を設定します。

この要素はトランザクションリカバリサービス専用です。

この要素をアプリケーションの構成定義に記述した場合は無視されます。

属性

●nodeId="MSDTC 連携機能を使用する OS 環境を一意に識別するためのノード識別子"　～〈文字列〉

MSDTC 連携機能を使用する OS 環境を一意に識別するためのノード識別子です。

MSDTC 連携機能で使用するノード識別子生成コマンド（cnnnidgen）で作成したノード識別子を文字列で指定します。

アルファベットの大文字と小文字は区別しません。

この値を省略した場合，トランザクションリカバリサービスの起動に失敗します。

●appDomainCheckInterval="アプリケーションドメイン監視間隔"　～〈符号なし整数〉((1～65535))《10》（単位：秒）

アプリケーションドメイン監視間隔を指定します。

この属性は省略できます。

●recoverRetryInterval="トランザクションリカバリサービス回復処理失敗時の再試行間隔"　～〈符号なし整数〉((1～65535))《10》(単位：秒)

トランザクションリカバリサービスの回復処理に失敗した場合の再試行間隔を指定します。

この属性は省略できます。

●recoverCheckInterval="未決着トランザクションの存在を確認する間隔"　～〈符号なし整数〉((1～65535))《180》(単位：秒)

トランザクション回復処理の完了後，未決着トランザクションが存在しないかを確認する間隔を指定します。

この属性は省略できます。

●recoverCheckCount="未決着トランザクションの存在を確認する回数"　～〈符号なし整数〉((1～65535))《3》

トランザクション回復処理の完了後，未決着トランザクションが存在しないかを確認する回数を指定します。

0 を指定した場合，未決着トランザクションが存在しないか確認は行いません。

この属性は省略できます。

●profiles="トランザクションリカバリサービス構成定義のプロファイル ID"　～〈文字列〉

トランザクションリカバリサービスの構成定義で指定したプロファイル ID をすべてコンマ（,）区切りの文字列で指定します。

トランザクションリカバリサービスで指定する各プロファイル ID には，コンマ（,）を使用しないでください。

各プロファイルには，トランザクション回復情報を取得するための接続先となる<tp1Server>要素が指定されている必要があります。この値が指定されていない場合は<common>要素内に指定された<tp1Server>要素だけを参照します。

この値に次に示す値を指定した場合，トランザクションリカバリサービスの起動に失敗します。

- 空文字列（""）を指定した場合
- 同一名称のプロファイル ID を複数指定した場合
- 存在しないプロファイル ID を一つでも指定した場合
- 空文字列（""）のプロファイル ID を指定した場合

また，各プロファイル内の<tp1Server>要素が存在しない場合や誤りがある場合は，回復処理の実行時にエラーとなりますが処理は続行されます。

実行時のエラー情報は Client .NET のエラートレースを参照してください。

●rmidStoragePath="RMID 格納ディレクトリ"　～〈文字列〉

トランザクションリカバリサービスが未決着トランザクションの回復処理に必要な情報を保存する RMID 格納ディレクトリをフルパス名で指定します。

RMID 格納ディレクトリは，あらかじめ作成しておく必要があります。指定したディレクトリが存在しない場合，またはディレクトリへのアクセスに失敗した場合は，トランザクションリカバリサービスの起動に失敗します。

この属性は省略できません。

記述例

```
<recoveryService nodeId="18AF5B57-FED5-4522-9B0B-94FBDDDD3EA4"
                 appDomainCheckInterval="10"
                 recoverRetryInterval="10"
                 recoverCheckInterval="180"
                 recoverCheckCount="3"
                 profiles="tp1Server1,tp1Server2"
                 rmidStoragePath=
                   "C:¥Program Files¥Hitachi¥
                    TP1Connector for .NET Framework¥trs¥rmid"/>
```

定義例

MSDTC 連携機能を使用しない場合

```
<configuration>
  <configSections>
    <section
      name="hitachi.opentp1.connector"
      type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,
            Culture=neutral,PublicKeyToken=2440cf5f0d80c91c,
            Custom=null"/>
    <section
      name="hitachi.opentp1.client"
      type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,
            Culture=neutral,PublicKeyToken=2440cf5f0d80c91c,
            Custom=null"/>
  </configSections>

  <hitachi.opentp1.connector>
    <common>
      <client conf=""/>
      <connection pooled="20" active="100"
                  threshold="80" watchtime="120"
                  failureInfoSharing="true"
                  failureCheckInterval="300"/>
        <occupation pooled="10"/>
      </connection>
      <log destination="c:¥temp¥connectorn" fileSize="1048576"
           level="1"/>
      <buffer pooling="true">
        <largestBufferPool maxCount="5"
                           threshold="100"/>
        <bufferPool size="102400" maxCount="50" threshold="120"/>
        <bufferPool size="10240" maxCount="50" threshold="150"/>
      </buffer>
      <perfCounter use="true"/>
      <option maxMessageSize="8"/>
    </common>
    <profile id="gyoumu2">
      <client conf="server2"/>
      <connection>
        <occupation pooled="10"/>
      </connection>
      <buffer pooling="true">
        <largestBufferPool maxCount="2"/>
        <bufferPool size="102400" maxCount="10"/>
      </buffer>
    </profile>
    <profile id="gyoumu3">
      <client conf="server3"/>
      <connection>
        <tcpip keepAlive="true"/>
        <occupation pooled="2"/>
      </connection>
```

```
    </profile>
  </hitachi.opentp1.connector>

  <hitachi.opentp1.client>
    <common>
      <tp1Server host="hostA"/>
      <tp1Server host="hostB"/>
      <rpc use="nam" watchTime="0"/>
      <nameService port="10000"/>
    </common>
    <profile id="traceMode">
      <errTrace use="true" path="c:¥temp¥clientn"
                fileSize="100000"/>
      <methodTrace use="true" path="c:¥temp¥clientn"
                   fileSize="100000"/>
      <uapTrace use="true" path="c:¥temp¥clientn"
                fileSize="100000"/>
      <dataTrace use="true" path="c:¥temp¥clientn"
                 fileSize="100000"/>
    </profile>
    <profile id="server2">
      <tp1Server host="hostA"/>
      <rpc use="nam" watchTime="0"/>
      <nameService port="10000"/>
    </profile>
    <profile id="server3">
      <tcpip use="true"
             type="sendrecv"
             sendHost="hostB"
             sendPort="20000"
             openPortAtRecv="true"/>
    </profile>
  </hitachi.opentp1.client>

</configuration>

<configuration>
  <configSections>
    <section
      name="hitachi.opentp1.connector"
      type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,
            Culture=neutral,PublicKeyToken=2440cf5f0d80c91c,
            Custom=null"/>
    <section
      name="hitachi.opentp1.client"
      type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,
            Culture=neutral,PublicKeyToken=2440cf5f0d80c91c,
            Custom=null"/>
  </configSections>

  <hitachi.opentp1.connector>
    <common>
      <client conf=""/>
      <connection pooled="20" active="100"
                  threshold="80" watchtime="120"
                  failureInfoSharing="true"
```

```
                failureCheckInterval="300"/>
      <occupation pooled="10"/>
    </connection>
    <log destination="c:¥temp¥connectorn" fileSize="1048576"
         level="1"/>
    <buffer pooling="true" create="false">
      <largestBufferPool maxCount="5"
                         threshold="100"/>
      <bufferPool size="102400" maxCount="50" threshold="120"/>
      <bufferPool size="10240" maxCount="50" threshold="150"/>
    </buffer>
    <perfCounter use="true"/>
    <option maxMessageSize="8"/>
  </common>
  <profile id="gyoumu2">
    <client conf="server2"/>
    <connection>
      <occupation pooled="10"/>
    </connection>
    <buffer pooling="true">
      <largestBufferPool maxCount="2"/>
      <bufferPool size="102400" maxCount="10"/>
    </buffer>
  </profile>
  <profile id="gyoumu3">
    <client conf="server3"/>
    <connection>
      <tcpip keepAlive="true"/>
      <occupation pooled="2"/>
    </connection>
  </profile>
</hitachi.opentp1.connector>

<hitachi.opentp1.client>
  <common>
    <tp1Server host="hostA"/>
    <tp1Server host="hostB"/>
    <rpc use="nam" watchTime="0"/>
    <nameService port="10000"/>
  </common>
  <profile id="traceMode">
    <errTrace use="true" path="c:¥temp¥clientn"
              fileSize="100000"/>
    <methodTrace use="true" path="c:¥temp¥clientn"
              fileSize="100000"/>
    <uapTrace use="true" path="c:¥temp¥clientn"
              fileSize="100000"/>
    <dataTrace use="true" path="c:¥temp¥clientn"
              fileSize="100000"/>
  </profile>
  <profile id="server2">
    <tp1Server host="hostA"/>
    <rpc use="nam" watchTime="0"/>
    <nameService port="10000"/>
  </profile>
  <profile id="server3">
    <tcpip use="true"
           type="sendrecv"
```

```
            sendHost="hostB"
            sendPort="20000"
            openPortAtRecv="true"/>
    </profile>
  </hitachi.opentp1.client>

</configuration>
```

MSDTC 連携機能を使用した場合（アプリケーションの構成定義の設定例）

```
<configuration>
  <configSections>
    <section
      name="hitachi.opentp1.connector"
      type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,
            Culture=neutral,PublicKeyToken=2440cf5f0d80c91c,
            Custom=null"/>
    <section
      name="hitachi.opentp1.client"
      type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,
            Culture=neutral,PublicKeyToken=2440cf5f0d80c91c,
            Custom=null"/>
  </configSections>

  <hitachi.opentp1.connector>
    <common>
      <client conf=""/>
      <connection pooled="20" active="100"
                  threshold="80" watchtime="120"
                  failureInfoSharing="true"
                  failureCheckInterval="300"/>
        <connection pooled="10"/>
      </connection>
      <log destination="c:¥temp¥connectorn" fileSize="1048576"
           level="2"/>
      <buffer pooling="true" create="false">
        <largestBufferPool maxCount="5" threshold="100"/>
        <bufferPool size="102400" maxCount="50" threshold="120"/>
        <bufferPool size="10240" maxCount="50" threshold="150"/>
      </buffer>
      <perfCounter use="true"/>
      <option maxMessageSize="8"/>
      <distributedTransaction use="true"
        nodeId="18AF5B57-FED5-4522-9B0B-94FBDDDD3EA4"
        optimize1PC="false"
        recoverRetryInterval="10"/>
    </common>
  </hitachi.opentp1.connector>

  <hitachi.opentp1.client>
    <common>
      <tp1Server host="hostA"/>
      <rpc use="rap" watchTime="180"/>
      <rapService port="10020" autoConnect="true"/>
    </common>
```

```
    <profile id="traceMode">
      <errTrace use="true" path="c:¥temp¥clientn"
                fileSize="100000"/>
      <methodTrace use="true" path="c:¥temp¥clientn"
                fileSize="100000"/>
      <uapTrace use="true" path="c:¥temp¥clientn"
                fileSize="100000"/>
      <dataTrace use="true" path="c:¥temp¥clientn"
                fileSize="100000"/>
    </profile>
  </hitachi.opentp1.client>

</configuration>
```

MSDTC 連携機能を使用した場合（トランザクションリカバリサービスの構成定義の設定例 1）

```
<configuration>
  <configSections>
    <section
      name="hitachi.opentp1.connector"
      type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,
            Culture=neutral,PublicKeyToken=2440cf5f0d80c91c,
            Custom=null"/>
    <section
      name="hitachi.opentp1.client"
      type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,Culture=neutral,
            PublicKeyToken=2440cf5f0d80c91c,Custom=null"/>
  </configSections>

  <hitachi.opentp1.connector>
    <common>
      <client conf=""/>
      <connection pooled="1" active="1" watchtime="120"/>
      <log destination="c:¥temp¥connectorn¥trs" fileSize="1048576"
                level="2"/>
      <recoveryService nodeId=
                        "18AF5B57-FED5-4522-9B0B-94FBDDDD3EA4"
                        appDomainCheckInterval="10"
                        recoverRetryInterval="10"
                        recoverCheckInterval="180"
                        recoverCheckCount="3"/>
                        rmidStoragePath=
                        "C:¥Program Files¥Hitachi¥
                        TP1Connector for .NET Framework¥trs¥rmid"/>
    </common>
  </hitachi.opentp1.connector>

  <hitachi.opentp1.client>
    <common>
      <tp1Server host="hostA"/>
      <rpc use="rap" watchTime="180"/>
      <rapService port="10020" autoConnect="true"/>
      <errTrace use="true" path="c:¥temp¥clientn¥trs"
```

```
                fileSize="100000"/>
    </common>
  </hitachi.opentp1.client>
```

MSDTC 連携機能を使用した場合（トランザクションリカバリサービスの構成定義の設定例 2）

```
<configuration>
  <configSections>
    <section
      name="hitachi.opentp1.connector"
      type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,
      Culture=neutral,PublicKeyToken=2440cf5f0d80c91c,
      Custom=null"/>
    <section
      name="hitachi.opentp1.client"
      type="Hitachi.OpenTP1.Common.Util.ProfileSectionHandler,
            Hitachi.OpenTP1.Client,Version=7.0.0.0,
            Culture=neutral,
            PublicKeyToken=2440cf5f0d80c91c,Custom=null"/>
  </configSections>

  <hitachi.opentp1.connector>
    <common>
      <client conf=""/>
      <connection pooled="2" active="2" watchtime="120"/>
      <log destination="c:¥temp¥connectorn¥trs"
           fileSize="1048576" level="2"/>
      <recoveryService nodeId=
                       "18AF5B57-FED5-4522-9B0B-94FBDDDD3EA4"
                       appDomainCheckInterval="10"
                       recoverRetryInterval="10"
                       recoverCheckInterval="180"
                       recoverCheckCount="3"
                       rmidStoragePath=
                       "C:¥Program Files¥Hitachi¥
                       TP1Connector for .NET Framework¥trs¥rmid"/>
                       profiles="tp1Server1,tp1Server2"/>
    </common>
    <profile id="tp1Server1">
      <client conf="server1"/>
      <connection>
        <connection pooled="1"/>
      </connection>
    </profile>
    <profile id="tp1Server2">
      <client conf="server2"/>
      <connection>
        <connection pooled="1"/>
      </connection>
    </profile>
  </hitachi.opentp1.connector>

  <hitachi.opentp1.client>
    <common>
```

```
    <errTrace use="true" path="c:¥temp¥clientn¥trs"
              fileSize="100000"/>
  </common>
  <profile id="server1">
    <tp1Server host="hostA"/>
    <rpc use="rap" watchTime="180"/>
    <rapService port="10020" autoConnect="true"/>
  </profile>
  <profile id="server2">
    <tp1Server host="hostB"/>
    <rpc use="rap" watchTime="180"/>
    <rapService port="10020" autoConnect="true"/>
  </profile>
</hitachi.opentp1.client>
```

4 UAP の作成と実行

この章では，Connector .NET で使用する UAP の作成方法とサンプルプログラムの使用方法について説明します。

4.1 OpenTP1 for .NET Framework 環境での UAP 開発時に必要な定義

OpenTP1 for .NET Framework 環境での UAP 開発時に必要な定義について説明します。

4.1.1 .NET インタフェース定義

.NET インタフェース定義は，次の UAP から SPP.NET に対して.NET インタフェース定義を使用して RPC を実行する場合に定義します。

- SPP.NET
- SUP.NET
- CUP.NET（Connector .NET を利用して SPP.NET にサービスを要求するアプリケーションを含みます）

ここでは，.NET インタフェース定義の定義方法の詳細について説明します。.NET インタフェース定義を使用した RPC については，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。

(1) .NET インタフェース定義の定義方法

(a) .NET インタフェース定義に使用する言語

SPP.NET の.NET インタフェース定義は，SPP.NET を開発するプログラム言語で定義します。.NET インタフェース定義を定義する場合，次のプログラム言語が使用できます。COBOL 言語は使用できません。

- C#
- J#※
- Visual Basic

注※

Visual Studio 2013 以降では，J#を使用して開発できません。

各プログラム言語でのインタフェースの定義文法は，Visual Studio や.NET Framework SDK のドキュメントを参照してください。

なお，.NET インタフェース定義は，Visual Studio や.NET Framework SDK が提供する各プログラム言語のコンパイラで，.NET Framework および OpenTP1 for .NET Framework が提供するクラスライブラリだけを参照してコンパイルできる必要があります。

(b) .NET インタフェース定義の定義規則

SPP.NET のインタフェースを定義する場合，次の規則があります。これらの規則に従っていない場合，SPP.NET の.NET インタフェース定義として使用できません。

- 名前空間名，インタフェース名，メソッド名には，半角英数字および半角アンダスコア（_）が使用できます。なお，名前空間の区切り文字として半角ピリオド（.）が使用できます。
- メソッドの引数および戻り値に使用できるデータ型を次の表に示します。

表 4-1 メソッドの引数および戻り値で使用できるデータ型

共通型システム	プログラム言語		
	C#	J#	Visual Basic
System.Void※	void	void	−
System.Byte	byte	ubyte	Byte
System.Int16	short	short	Short
System.Int32	int	int	Integer
System.Int64	long	long	Long
System.String	string	String	String
System.Byte[]	byte[]	ubyte[]	Byte()
System.Int16[]	short[]	short[]	Short()
System.Int32[]	int[]	int[]	Integer()
System.Int64[]	long[]	long[]	Long()
System.String[]	string[]	String[]	String()
TP1 ユーザ構造体	−	−	−
TP1 ユーザ構造体配列	−	−	−

（凡例）

−：プログラム言語による差異はありません。

注※

System.Void はメソッドの戻り値にだけ指定できます。

- メソッドの引数に使用できるパラメタ属性を次の表に示します。

表 4-2 メソッドの引数に使用できるパラメタ属性

意味	プログラム言語		
	C#	J#※2	Visual Basic
値渡し	var（省略可）	なし	ByVal（省略可）
出力渡し	out	指定不可※1	指定不可※1

意味	プログラム言語		
	C#	J#※2	Visual Basic
参照渡し	ref	Holder クラス※2	ByRef

注※1

out 属性は C#固有の属性です。C#以外では指定できません。C#以外のプログラム言語で使用する場合は，代わりに参照渡しを使用してください。

注※2

J#ではパラメタ属性の代わりにデータ型で表現します。J#では値型の参照渡しができないため，OpenTP1 が提供する Holder クラスを使用してください。

- OpenTP1 が提供する Holder クラスを次の表に示します。

Holder クラス名	保持する値の型	
	J#	共通型システム
Hitachi.OpenTP1.UByteHolder	ubyte	System.Byte
Hitachi.OpenTP1.ShortHolder	short	System.Int16
Hitachi.OpenTP1.IntHolder	int	System.Int32
Hitachi.OpenTP1.LongHolder	long	System.Int64
Hitachi.OpenTP1.StringHolder	String	System.String
Hitachi.OpenTP1.UByteArrayHolder	ubyte[]	System.Byte[]
Hitachi.OpenTP1.ShortArrayHolder	short[]	System.Int16[]
Hitachi.OpenTP1.IntArrayHolder	int[]	System.Int32[]
Hitachi.OpenTP1.LongArrayHolder	long[]	System.Int64[]
Hitachi.OpenTP1.StringArrayHolder	String[]	System.String[]

注 1

これらの Holder クラスは J#でだけ使用できます。C#および Visual Basic では ref，ByRef などのパラメタ属性を使用してください。

注 2

これらの Holder クラスは，メソッドの引数としてだけ使用できます。メソッドの戻り値には使用できません。

- メソッド名の長さは 31 文字以内でなければなりません。なお，メソッド名は SPP.NET のサービス名になります。
- メソッド名は英字で始まる必要があります。
- メソッドのオーバロードはできません。
- 大文字，小文字だけが異なる同じ名称のメソッドを定義することはできません。

(2) TP1 ユーザ構造体

(a) TP1 ユーザ構造体とは

TP1 ユーザ構造体は，複数の値をまとめて保持できるクラスのことです。

(b) TP1 ユーザ構造体の定義規則

TP1 ユーザ構造体を定義する場合の規則を次に示します。

- TP1 ユーザ構造体として利用するクラスは，Hitachi.OpenTP1.TP1UserStruct クラスを継承します。
- デフォルトコンストラクタ（アクセス修飾子が public で，引数のないコンストラクタ）を持つ必要があります。
- set，get 両方のアクセサを持つ public プロパティが一つ以上必要です。
- 大文字，小文字だけが異なる同一名称のプロパティを定義することはできません。

注意事項

public プロパティ以外のすべてのメンバは RPC で送受信されるデータには含まれません。

(c) TP1 ユーザ構造体のプロパティとして利用できるデータ型

TP1 ユーザ構造体のプロパティとして利用できるデータ型，およびメソッドの引数として使用できるパラメタ属性を次の表に示します。

表 4-3 TP1 ユーザ構造体のプロパティとして利用できるデータ型

共通型システム	プログラム言語		
	C#	J#	Visual Basic
System.Byte	byte	ubyte	Byte
System.Int16	short	short	Short
System.Int32	int	int	Integer
System.Int64	long	long	Long
System.String	string	String	String
System.Byte[]	byte[]	ubyte[]	Byte()
System.Int16[]	short[]	short[]	Short()
System.Int32[]	int[]	int[]	Integer()
System.Int64[]	long[]	long[]	Long()
System.String[]	string[]	String[]	String()
TP1 ユーザ構造体	−	−	−

共通型システム	プログラム言語		
	C#	J#	Visual Basic
TP1 ユーザ構造体配列	－	－	－

（凡例）

－：プログラム言語による差異はありません。

(d)　TP1 ユーザ構造体の参照渡しの方法

TP1 ユーザ構造体の参照渡しをするには，パラメタ属性を使用します。ただし，J#の場合は参照渡しができないため，Hitachi.OpenTP1.ITP1UserStructHolder を実装する必要があります。実装上の規則を次に示します。

- プロパティとして利用できるデータ型は TP1 ユーザ構造体，または TP1 ユーザ構造体の配列に限られます。
- プロパティは set，get 両方のアクセサを持ち，アクセス修飾子が public である必要があります。
- プロパティは一つしか定義してはいけません。
- 実装クラスの名称規則を，次の表に示します。

プロパティのデータ型	実装クラスの名称
TP1 ユーザ構造体	TP1 ユーザ構造体クラス名称＋"Holder"
TP1 ユーザ構造体配列	TP1 ユーザ構造体クラス名称＋"ArrayHolder"

- 実装クラスの名前空間は，プロパティのデータ型と同じ名前空間である必要があります。

なお，プロパティ以外のすべてのメンバは RPC で送受信されるデータには含まれません。TP1 ユーザ構造体の参照渡しの使用例については，「(3)(b)　J#での定義例」を参照してください。

(3)　.NET インタフェース定義の定義例

(a)　C#での定義例

```
using System;
using Hitachi.OpenTP1;
namespace MyCompany
{

  public interface IGyoumuA
  {
    void Service1(string dataId, byte[] data);
    string[] Service2(string key);
    int Service3(int inCount, ref string[] ids);
    short Service4(MyStruct inStruct);
  }
  public class MyStruct : TP1UserStruct
  {
      private int id;
```

```
      private string name;

      public MyStruct(){}

      public int Id
      {
        set{
           id = value;
        }
        get{
           return id;
        }
      }

      public string Name
      {
         set{
            name = value;
         }
         get{
            return name;
         }
      }
  }
}
```

(b) J#での定義例

```
package MyCompany;
import System.*;
import Hitachi.OpenTP1.*;

public interface IGyoumuA
{
    void Service1(String dataId, ubyte[] data);
    String[] Service2(String key);
    int Service3(int inCount, StringArrayHolder ids);
    short Service4(MyStruct inStruct, MyStructHolder outStruct);
}

//TP1ユーザ構造体の参照渡しをするためのクラス
public class MyStructHolder implements ITP1UserStructHolder
{
   private MyStruct val;
   /** @property */
   public void set_Value(MyStruct value)
   {
          val = value;
   }
   /** @property */
   public MyStruct get_Value()
   {
          return val;
   }
}

//TP1ユーザ構造体
```

```
public class MyStruct extends TP1UserStruct
{
   private int id;
   private String name;

   public MyStruct()
   {
   }

   /** @property */
   public void set_Id(int value)
   {
          id = value;
   }
   /** @property */
   public int get_Id()
   {
          return id;
   }
   /** @property */
   public void set_Name(String value)
   {
          name = value;
   }
   /** @property */
   public String get_Name()
   {
          return name;
   }
}
```

(c) Visual Basic での定義例

```
Imports System
Imports Hitachi.OpenTP1
Namespace MyCompany
  Public Interface IGyoumuA
    Sub Service1(ByVal dataId As String, ByVal data() As Byte)
    Function Service2(ByVal key As String) As String()
    Function Service3(ByVal inCount As Integer, _
                      ByRef ids() As String) As Integer
    Function Service4(ByVal inStruct As MyStruct) As Short
  End Interface

  Public Class MyStruct
      Inherits TP1UserStruct
      Private idValue As Integer
      Private nameValue As String

      Public Sub New()
      End Sub

      Public Property Id() As Integer
          Set(ByVal value As Integer)
               idValue = value
          End Set
          Get
```

```
                Return idValue
            End Get
        End Property

        Public Property Name() As String
            Set(ByVal value As String)
                nameValue = value
            End Set
            Get
                Return nameValue
            End Get
        End Property
    End Class
End Namespace
```

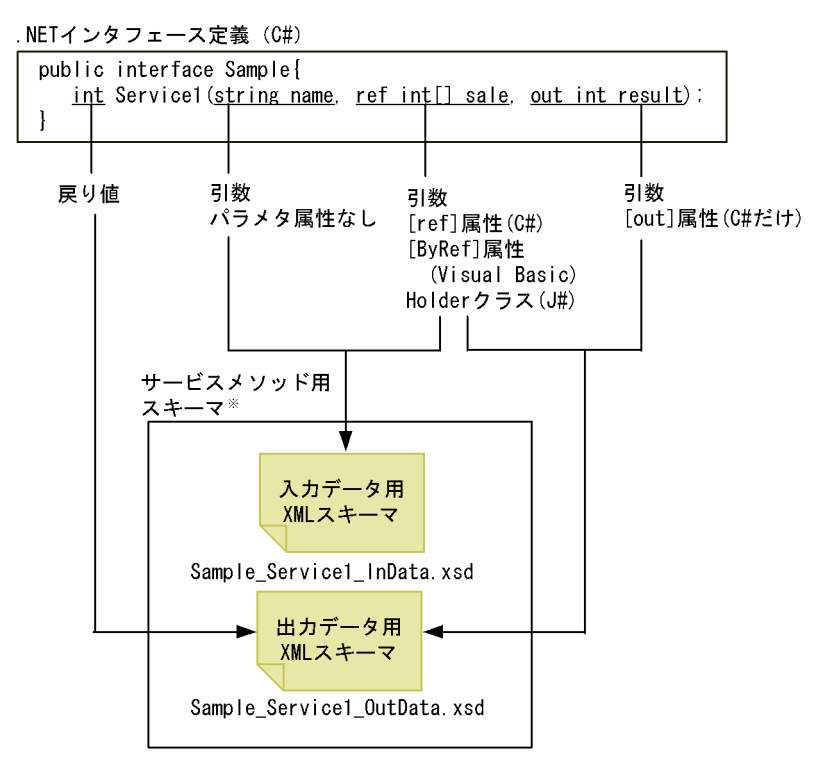
(4) .NET インタフェース定義から XML スキーマへのマッピング

クライアントスタブ生成コマンド（if2cstub）で-X オプションを指定して，.NET インタフェース定義からクライアントスタブを生成すると，引数および戻り値が XmlDocument クラス（System.Xml.XmlDocument）となるサービスメソッドがクライアントスタブに追加されます。サービスメソッドの引数および戻り値の XmlDocument クラスに指定できる XML の形式は，クライアントスタブと一緒に生成される入力データ用 XML スキーマおよび出力データ用 XML スキーマで定義されます。

.NET インタフェース定義のサービスメソッドの引数および戻り値は，次の図のように入力データ用 XML スキーマおよび出力データ用 XML スキーマに定義されます。入力データ用 XML スキーマのファイル名は，「〈.NET インタフェース定義のインタフェース名称〉_〈.NET インタフェース定義のサービスメソッド名称〉_InData.xsd」となり，出力データ用 XML スキーマのファイル名は，「〈.NET インタフェース定義のインタフェース名称〉_〈.NET インタフェース定義のサービスメソッド名称〉_OutData.xsd」となります。

クライアントスタブ生成コマンド（if2cstub）によって出力される XML スキーマは，XML スキーマ仕様「XML Schema Definition language（XSD）1.0」に従います。なお，名前空間 URI として「http://www.w3.org/2001/XMLSchema」が指定されます。

図 4-1　.NET インタフェース定義とサービスメソッド用スキーマの関係



注※

入力データ用 XML スキーマおよび出力データ用 XML スキーマに定義されたサービスメソッドごとに生成されます。

4.1.2　サービス定義

サービス定義は，次の UAP から SPP.NET，または SPP に対してサービス定義を使用して RPC を実行する場合に定義します。

- SPP.NET
- SUP.NET
- CUP.NET（Connector .NET を利用して SPP.NET，または SPP にサービスを要求するアプリケーションを含みます）

サービス定義は，サービス定義のファイルとサービス定義が参照するデータ型定義のファイルから構成されます。この節では，サービス定義の定義方法の詳細について説明します。サービス定義を使用した RPC については，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。

(1)　データ型定義ファイル

データ型定義とは，RPC でやり取りされる入力データや出力データの形式をメッセージ単位で定義するものです。

サービス定義から参照するデータ型定義ファイルは，次に示す形式で独立したファイルとして定義します。

(a) 形式

```
struct データ型定義名称 {
        データ型 メンバ名称[配列指定];
    〔〔データ型 メンバ名称[配列指定];〕…〕
};
 〔〔struct データ型定義名称 {
        データ型 メンバ名称[配列指定];
    〔〔データ型 メンバ名称[配列指定];〕…〕
};〕…〕
```

(b) 説明

●データ型定義名称　　～〈31 文字以内の識別子〉

データ型定義に付ける名称を指定します。

指定された名称がカスタムレコードクラスのクラス名となります。

●データ型

メンバ名称で示される変数に対するデータ型を指定します。

クライアントスタブ生成コマンド（spp2cstub）は，定義された各メンバのデータ型を次の表に示すように，.NET Framework のデータ型に対応づけてカスタムレコードクラスを生成します。

また，データの内容を次の表に示すように変換します。

表 4-4　データ型の変換規則と変換内容

データ型定義ファイルで定義されたデータ型	データ型の説明	カスタムレコードクラスで定義される.NET Framework のデータ型	データ変換の内容
char	文字列	System.String	char は.NET Framework の System.String に対応づけられます。カスタムレコードを入力データとして使用する場合，スタブ生成コマンドに指定したエンコード方式に従ってバイト配列に変換しデータを扱います。 バイト配列に変換した結果が 32 バイトで，データ型定義が char a[30]の場合，2 バイトは破棄されます。
int	32 ビット符号付き整数	System.Int32	int は.NET Framework の System.Int32 に対応づけられます。 カスタムレコードを入力データとして使用する場合，スタブ生成コマンドに指定されたエンディアンでバイト配列に変換して，RPC の要求メッセージに設定します。 カスタムレコードを出力データとして使用する場合，応答メッセージから 4 バイトをスタブ生成コマンドに指定されたエンディアンで読み込み，データにセットします。
short	16 ビット符号付き整数	System.Int16	short は.NET Framework の System.Int16 に対応づけられます。カスタムレコードを入力データとして使用する場

データ型定義ファイルで定義されたデータ型	データ型の説明	カスタムレコードクラスで定義される.NET Framework のデータ型	データ変換の内容
			合，指定されたエンディアンでバイト配列に変換して，RPC の要求メッセージに設定します。 カスタムレコードを出力データとして使用する場合，RPC の応答メッセージから 2 バイトをスタブ生成コマンドに指定されたエンディアンで読み込み，データにセットします。
long	32 ビット符号付き整数	System.Int32	long は.NET Framework の System.Int32 に対応づけられます。 カスタムレコードを入力データとして使用する場合，スタブ生成コマンドに指定されたエンディアンでバイト配列に変換して，RPC の要求メッセージに設定します。 カスタムレコードを出力データとして使用する場合，応答メッセージから 4 バイトをスタブ生成コマンドに指定されたエンディアンで読み込み，データにセットします。
byte	バイナリデータ	System.Byte	byte は.NET Framework の System.Byte にマッピングします。 一次元配列指定にだけ使用できます。
struct	構造体	class（インナークラス）	データ型定義で struct として定義されるデータ型のことを構造体と呼びます。 struct はインナークラスに対応づけられます。

データ型が int，long，または struct の場合は，先頭からのオフセットが 4 の整数倍でなければなりません。また，short の場合は，先頭からのオフセットが 2 の整数倍でなければなりません。

なお，スタブ生成コマンドでは，自動的にバウンダリ調整をしないため，先頭からのオフセットが正しい整数倍でない場合，エラーとなるので注意が必要です。

バウンダリ調整については，「(d)　バウンダリ調整」を参照してください。

データ型が struct の場合は，次に示すように定義します。

```
struct 構造体名称 {
        データ型 メンバ名称[配列指定];
    〔〔データ型 メンバ名称[配列指定];〕…〕
} メンバ名称[配列指定];
```

データ型定義で struct として定義されるメンバは，その構造体名称がそのままインナークラスのクラス名称になり，メンバ名称がインナークラスの型を持つプロパティ名称になります。

ここで指定するデータ型，メンバ名称，配列要素数はデータ型定義に従います。また，この構造体のデータ型定義先頭からのオフセットは，4 の整数倍でなければなりません。また，構造体自身のサイズも 4 の整数倍でなければなりません。

ただし，構造体を構成するメンバ（構造体の中に構造体がある場合は，そのメンバも含む）がすべて char または byte の場合は，任意のオフセットおよび任意のサイズを定義できます。

データ型定義で struct として定義される配列型は，可変長構造体配列として扱えます。可変長構造体配列については，「(e)　可変長構造体配列」を参照してください。

●構造体名称　　～〈31 文字以内の識別子〉

構造体に付ける名称を指定します。

構造体名称の先頭文字は半角英字とします。2 文字目以降は半角英数字および半角アンダスコア（_）が使用できます。

●メンバ名称　　～〈31 文字以内の識別子〉

メンバ名称を指定します。

メンバ名称の先頭文字は半角英字で指定してください。

●配列指定

配列要素数　　～〈符号なし整数〉((1～8388608))

メンバがデータ型の配列の場合，配列要素数を指定します。

データ型が int，short，long，または struct のときは，一次元配列が指定できます（配列指定なしも指定できます）。

データ型が byte のときは，一次元配列だけ指定できます。

データ型が char のときは，二次元配列まで指定できます。

一次元配列

```
[配列要素数]
```

二次元配列

```
[配列要素数][配列要素数]
```

(c)　データ型定義の定義例

```
struct in_data {
 long I_basho[3];
 long I_kakaku;
 long I_tokuchou;
};

struct out_data {
 char o_name[20];
 char o_basho[16];
 char o_tokuchou[20];
 long o_kakaku;
 char o_inf[80];
};

struct out_data2 {
 char o_name[20];
 char o_basho[16];
 char o_tokuchou[20];
 long o_kakaku;
 char o_inf[80][20];
};
```

```
struct put_data {
 int  o_num;
 struct data {
  char o_name[20];
  char o_basho[16];
  char o_tokuchou[20];
  long o_kakaku;
  char o_inf[80];
 } data_t[100];
};
```

(d) バウンダリ調整

バウンダリ調整とは，データ型定義の各変数を決められたバイト境界に配置することをいいます。バウンダリ調整は，コンパイラが自動的に行います。

例えば，次に示すような構造体を使用している場合，先頭からのオフセットを正しくするため，実際はOS，およびコンパイラによって，data と num の間に 1 バイトの補正が入ります。

構造体の定義例

```
struct s_data {
    char data[3];
    long num;
} s_data_t
```

バウンダリ調整後の定義例

```
struct s_data {
  char data[3];
    <1バイト>
    long num;
} s_data_t
```

この例の場合には，次のように修正して使用してください。

修正前

```
struct s_data {
    char data[3];
    long num;
} s_data_t;
```

修正後

```
struct s_data {
    char data[3];
    char wk;
    long num;
} s_data_t;
```

ポイント

OpenTP1 for .NET Framework 以外の OpenTP1 システムで使用していた構造体をデータ型定義として使用する場合などは，バウンダリ調整に注意してください。

(e)　可変長構造体配列

データ型が struct で，配列型の場合は，可変長構造体配列として扱えます。可変長構造体配列は次に示す形式で定義します。

可変長構造体配列の形式

```
int 配列要素数を示すメンバ名称
struct 構造体名称 {
                    データ型 メンバ名称[配列指定];
                    〔〔データ型 メンバ名称[配列指定];〕…〕
} メンバ名称[配列要素数を示すメンバ名称:配列の最大要素数];
```

可変長構造体配列を指定する場合は，可変長構造体配列の配列要素数を「データ型が int 型の配列要素数を示すメンバ名称:配列の最大要素数」という形式で記述します。構造体名称，データ型，メンバ名称および配列指定については，「(b)　説明」を参照してください。

可変長構造体配列の定義例

```
struct put_data {
                int o_num;
                struct data{
                            char o_name[20];
                            char o_basho[16];
                            char o_tokuchou[20];
                            long o_kakaku;
                            char o_inf[80];
                            } data_t[o_num:100];
                };
```

可変長構造体配列を使用する場合は，次の点に注意してください。

- 配列要素数を示すメンバ名称は可変長構造体配列より前に定義してください。
- 配列要素数を示すメンバ名称の値は配列の最大要素数に指定した値以下にしてください。配列要素数を示すメンバ名称の値が配列の最大要素数に指定した値を超える場合はエラーが発生します。
- 構造体の長さは配列の最大要素数に指定した値で計算されるので，配列の最大要素数を指定するときはデータ型定義の長さが 1 から 8388608 バイトの範囲になるように指定してください。ただし，RPC 送受信メッセージの最大長拡張機能を使用しない場合は，データ型定義の長さは 1 から 1048576 バイトの範囲になるように指定してください。
- 生成されたカスタムレコードを使用する場合，可変長構造体配列に対応するプロパティには，配列の最大要素数以内の構造体配列を指定してください。null または配列の最大要素数を超える構造体配列を指定した場合は，TP1MarshalException 例外が発生します。

(f) 注意事項

- データ型定義では任意の位置にコメントを記述できます。コメントは「/*」で始め，「*/」で終了します。なお，コメントのネストはできません。
- データ型定義の長さは 1 から 8388608 バイトの範囲になるように記述してください。ただし，RPC 送受信メッセージの最大長拡張機能を使用しない場合は，1 から 1048576 バイトの範囲になるように記述してください。
- データ型定義ファイルでは，1 文を複数行にわたって指定しないでください。
- データ型定義名称として dc および DC で始まる名称は使用しないでください。
- データ型定義名称，メンバ名称としてクライアントスタブおよびカスタムレコードクラスを生成するプログラム言語のキーワードは使用できません。
- 生成されたカスタムレコードを使用する場合，配列および固定長構造体配列に対応するプロパティには要素数分の配列数を指定してください。null または要素数分以外の配列数を指定した場合は，TP1MarshalException 例外が発生します。
- 一つのデータ型定義に同じ名前のメンバ名称を指定しないでください。

(2) サービス定義ファイル

サービス定義とは，サービスの入出力データに対応するデータ型定義を定義するものです。

サービス定義ファイルは，次に示す形式で独立したファイルとして定義します。

(a) 形式

```
#include ”データ型定義ファイル名”
 〔〔#include ”データ型定義ファイル名”〕
…〕

interface サービス定義名称 {
      サービス名称(入力データ型定義名, 出力データ型定義名);
   〔〔サービス名称(入力データ型定義名, 出力データ型定義名);〕…〕
}
```

(b) 説明

●データ型定義ファイル名　　～〈ファイル名〉

このサービス定義で参照するデータ型定義が定義されているファイル名を指定します。ファイル名は次のどちらかの形式で指定します。

- 拡張子を含んだファイル名だけを指定する。
- 相対パスまたは絶対パスを含んだファイル名で指定する。

なお，パス指定時の区切り文字には「/」または「¥」が使用できます。

●サービス定義名称　　～〈31 文字以内の識別子〉

このサービス定義に付けるサービス定義名称を指定します。

任意に名称を付けることができます。デフォルトではこの名称からクライアントスタブなどのクラス名が生成されます。

●サービス名称　　～〈31 文字以内の識別子〉

このサービス定義に含めるサービス名称を指定します。

対象となるユーザサーバが持つサービス名称を指定してください。

●入力データ型定義名　　～〈31 文字以内の識別子〉

各サービスの入力データに対応するデータ型定義名称を指定します。

入力データ型定義名は，このサービス定義ファイルの#include ディレクティブで指定したデータ型定義ファイルで定義されたデータ型定義名称でなければなりません。

●出力データ型定義名　　～〈31 文字以内の識別子〉

各サービスの出力データに対応するデータ型定義名称を指定します。

出力データ型定義名は，このサービス定義ファイルの#include ディレクティブで指定したデータ型定義ファイルで定義されたデータ型定義名称でなければなりません。

ただし，出力データがない場合は，出力データ型定義名には DC_NODATA を指定してください（非応答型 RPC の場合だけ使用できます）。

(c)　サービス定義の定義例

サービス定義の定義例 1（業務 1 のサービス定義）

```
#include ”mydata.h”
/* 業務1のサービス定義 */
interface GYOUMU1 {
    GETDATA1(in_data, out_data);
    GETDATA2(in_data, out_data2);
}
```

サービス定義の定義例 2（業務 2 のサービス定義）

```
#include ”datas/mydata.h”
/* 業務2のサービス定義 */
interface GYOUMU2 {
    GET_DATA1(in_data, out_data);
    PUT_DATA1(put_data, DC_NODATA);  /* 非応答型 */
}
```

データ型定義の定義例（mydata.h）

```
struct in_data {
 long I_basho[3];
 long I_kakaku;
};
struct out_data {
```

```
 char o_name[20];
 char o_basho[16];
 long o_kakaku;
 char o_inf[80];
};
struct out_data2 {
 char o_name[20];
 char o_basho[16];
 long o_kakaku;
 char o_inf[80][20];
};
struct put_data {
 int  o_num;
 struct data {
  char o_name[20];
  char o_basho[16];
  long o_kakaku;
  char o_inf[80];
 } data_t[100];
};
```

(d) 注意事項

- サービス定義では任意の位置にコメントを記述できます。コメントは「/*」で始め，「*/」で終了します。なお，コメントのネストはできません。
- コメント文中に「//」は使用できません。

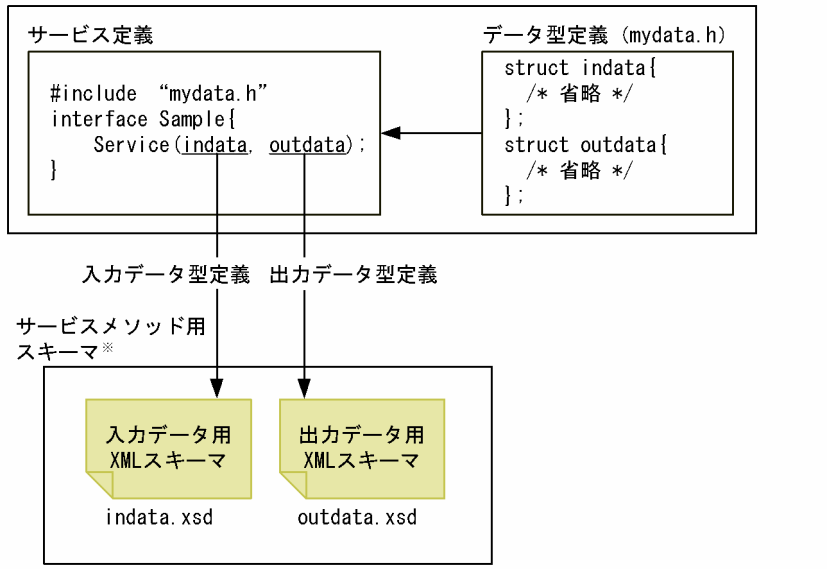
(3) サービス定義から XML スキーマへのマッピング

クライアントスタブ生成コマンド（spp2cstub）で-X オプションを指定して，サービス定義からクライアントスタブを生成すると，引数および戻り値が XmlDocument クラス（System.Xml.XmlDocument）となるサービスメソッドがクライアントスタブに追加されます。サービスメソッドの引数および戻り値の XmlDocument クラスに指定できる XML の形式は，クライアントスタブと一緒に生成される入力データ用 XML スキーマおよび出力データ用 XML スキーマで定義されます。

データ型定義の内容を基にして，次の図のように入力データ用 XML スキーマおよび出力データ用 XML スキーマに定義されます。入力データ用 XML スキーマのファイル名は，「〈対応する入力データ型定義名称〉.xsd」となり，出力データ用 XML スキーマのファイル名は，「〈対応する出力データ型定義名称〉.xsd」となります。

クライアントスタブ生成コマンド（spp2cstub）によって出力される XML スキーマは，XML スキーマ仕様「XML Schema Definition language（XSD）1.0」に従います。なお，名前空間 URI として「http://www.w3.org/2001/XMLSchema」が指定されます。

図 4-2　サービス定義，データ型定義，およびサービスメソッド用スキーマの関係



注※

サービス定義で指定した入力データ型定義，出力データ型定義ごとに，入力データ用 XML スキーマ，出力データ用 XML スキーマが生成されます。

4.2　.NET インタフェース定義を使用した SPP.NET の呼び出し方法

Connector .NET のアプリケーションから.NET インタフェース定義を使用した SPP.NET を呼び出す方法について説明します。

Extension .NET，Client .NET の各 UAP からの呼び出し方法については，それぞれマニュアル「TP1/Extension for .NET Framework 使用の手引」，マニュアル「TP1/Client for .NET Framework 使用の手引」の.NET インタフェース定義を使用した SPP.NET の呼び出し方法についての記述を参照してください。なお，次のプログラム言語が使用できます。COBOL 言語は使用できません。

- C#
- J#※
- Visual Basic

注※

Visual Studio 2013 以降では，J#を使用して開発できません。

4.2.1　クライアントスタブの生成

クライアントスタブ生成コマンド（if2cstub）を使用して，.NET インタフェース定義からクライアントスタブを生成します。

クライアントスタブは，クライアントスタブを利用するクライアントアプリケーション（Connector .NET のアプリケーション）を記述するプログラム言語で生成してください。

.NET インタフェース定義を記述したプログラム言語と生成するクライアントスタブのプログラム言語が異なる場合，クライアントスタブのメソッドの引数のパラメタ属性は次の表に従って対応づけられます。

表 4-5　クライアントスタブでのパラメタ属性の対応づけ

プログラム言語とメソッドの引数のパラメタ属性（.NET インタフェース定義）		メソッドの引数のパラメタ属性（クライアントスタブ）		
		C#	J#	Visual Basic
C#	なし	なし	なし	ByVal
	out	out	Holder クラス※1	ByRef※1
	ref	ref	Holder クラス	ByRef
J#	なし	なし	なし	ByVal
	Holder クラス	ref※2	Holder クラス	ByRef※2
Visual Basic	なし	なし	なし	ByVal
	ByVal	なし	なし	ByVal

プログラム言語とメソッドの引数のパラメタ属性（.NET インタフェース定義）		メソッドの引数のパラメタ属性（クライアントスタブ）		
		C#	J#	Visual Basic
	ByRef	ref	Holder クラス	ByRef

注※1

呼び出し元で値を設定しても，値はサーバに渡されません。

注※2

Holder クラスは，各 Holder クラスが保持する型の参照渡しに対応づけられます。

クライアントスタブ生成コマンド（if2cstub）で-X オプションを指定して，.NET インタフェース定義からクライアントスタブを生成すると，引数および戻り値が XmlDocument クラス（System.Xml.XmlDocument）となるサービスメソッドがクライアントスタブに追加されます。このサービスメソッドの名称は，「〈.NET インタフェース定義で指定されたサービスメソッド名称〉ByXml」となります。

サービスメソッドの引数および戻り値の XmlDocument クラスに指定できる XML の形式は，クライアントスタブと一緒に生成される入力データ用 XML スキーマおよび出力データ用 XML スキーマで定義されます。XML スキーマの詳細については，「4.1.1(4)　.NET インタフェース定義から XML スキーマへのマッピング」，および「4.1.2(3)　サービス定義から XML スキーマへのマッピング」を参照してください。

.NET インタフェース定義で指定されたサービスメソッドの引数および戻り値，入力データ用 XML スキーマおよび出力データ用 XML スキーマ，ならびにクライアントスタブに追加されたサービスメソッドの引数および戻り値の関係を次の表に示します。

表 4-6　.NET インタフェース定義のサービスメソッドの引数および戻り値，定義される XML スキーマ，ならびにクライアントスタブのサービスメソッドの引数および戻り値の関係

プログラム言語別の，サービスメソッドの引数のパラメタ属性および戻り値（.NET インタフェース定義）				定義される XML スキーマ	出力されるサービスメソッド（クライアントスタブ）	
					引数	戻り値
引数	in	C#	なし	入力データ用 XML スキーマ	XmlDocument	XmlDocument，またはなし※2
		J#	なし			
		Visual Basic	ByVal※1			
	out	C#	out	出力データ用 XML スキーマ	XmlDocument，またはなし※3	XmlDocument
		J#	×			
		Visual Basic	×			
	inout	C#	ref	入力データ用 XML スキーマ，出力データ用 XML スキーマ	XmlDocument	XmlDocument
		J#	Holder クラス			
		Visual Basic	ByRef			

プログラム言語別の，サービスメソッドの引数のパラメタ属性および戻り値（.NET インタフェース定義）			定義される XML スキーマ	出力されるサービスメソッド（クライアントスタブ）	
				引数	戻り値
戻り値	C#	−	出力データ用 XML スキーマ	XmlDocument，またはなし※3	XmlDocument
	J#	−			
	Visual Basic	−			

（凡例）

×：指定できません。

−：該当しません。

注※1

省略できます。

注※2

.NET インタフェース定義で指定されたサービスメソッドの引数が in 属性だけで，かつ戻り値がない場合，クライアントスタブに出力されるサービスメソッドの戻り値は，なしとなります。それ以外は，XmlDocument となります。

注※3

.NET インタフェース定義で指定されたサービスメソッドの引数に，in 属性または inout 属性がない場合，クライアントスタブに出力されるサービスメソッドの引数は，なしとなります。それ以外は，XmlDocument となります。

4.2.2　クライアントスタブの使用方法

クライアントスタブを使用してサービス要求をする手順を次に示します。

1. TP1ConnectionManager クラスのインスタンスを生成，または取り出します。

   プロパティを特定のプロファイルから読み込む場合は，コンストラクタの引数にプロファイル ID を指定します。インスタンスの生成はアプリケーションごとに行うことを推奨します。

2. TP1ConnectionManager クラスの GetConnection メソッドで TP1Connection オブジェクトを取得します。

   コネクションプールに使用できるコネクションがあった場合は，この時点でコネクションプールからコネクションを取り出した状態になります。

3. クライアントスタブを TP1Connection クラスのインスタンスごと，およびサービスグループごとにインスタンス化します。

4. 必要に応じてプロパティ（Flags，WatchTime）を設定します。

   クライアントスタブの初期値は Flags に RpcInfo.DCNOFLAGS，WatchTime に-1 が設定されています。Flags プロパティ，および WatchTime プロパティの詳細については，RpcInfo クラスの各プロパティの説明を参照してください。

5. 入力パラメタにデータを設定します。

XmlDocument を引数とするサービスメソッドの場合は，引数用の XmlDocument クラスをインスタンス化して，入力データ用 XML スキーマに従った XML 文書の内容を設定します。

6. サービスメソッドを呼び出します。

7. 戻り値や出力パラメタを参照して必要な処理をします。

   XmlDocument を戻り値とするサービスメソッドの場合は，出力データ用 XML スキーマに従って，返された XML 文書の内容を参照します。

8. TP1Connection クラスの Dispose メソッドを発行して，コネクションをコネクションプールに戻します。

   複数の TP1Connection オブジェクトを使う場合は，それぞれのインスタンスが不要になった時点で Close メソッドを発行します。

注意事項

ASP.NET Web アプリケーションで使用する場合，TP1Connection クラスの Dispose メソッドは，次の時点で必ず発行してください。

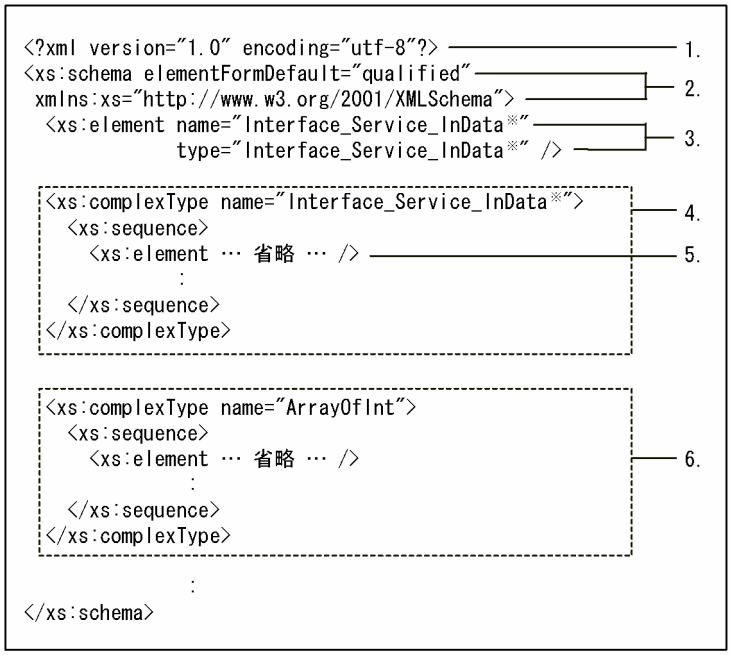
- 各 UI イベントハンドラの終了時
- Web サービスメソッドの終了時

Dispose メソッドを適切に発行しないと，メモリリークの原因となります。

また，TP1ConnectionManager クラスは ASP.NET ではアプリケーション状態，またはアプリケーションインスタンスに保持することを推奨します。各 UI イベントハンドラや Web サービスメソッドごとに生成と削除を繰り返した場合，性能劣化やメモリ使用効率低下の原因となります。

4.2.3 XML スキーマの定義方法

入力データ用 XML スキーマおよび出力データ用 XML スキーマの定義形式に示します。



注※

すべて同じ値が指定されます。

1. XML 宣言ノードの version 属性には 1.0，encoding 属性には utf-8 が指定されます。
2. schema 要素の elementFormDefault 属性には qualified，xmlns 属性には名前空間 URI として「http://www.w3.org/2001/XMLSchema」が指定されます。
3. ルート要素となる element 要素の name 属性には，入力データ用 XML スキーマの場合，「〈.NET インタフェース定義のインタフェース名称〉_〈.NET インタフェース定義のサービスメソッド名称〉_InData」，出力データ用 XML スキーマの場合，「〈.NET インタフェース定義のインタフェース名称〉_〈.NET インタフェース定義のサービスメソッド名称〉_OutData」が指定されます。type 属性には，name 属性と同じ値が指定されます。
4. 3.の type 属性が complexType 要素で定義されます。complexType 要素の name 属性には，3.の type 属性と同じ値が指定されます。
5. sequence 要素の子要素として element 要素が定義されます。element 要素は，.NET インタフェース定義のサービスメソッドの引数，TP1 ユーザ構造体のメンバ，および戻り値によって定義されます。.NET インタフェース定義のサービスメソッドの引数，TP1 ユーザ構造体のメンバ，および戻り値から定義される element 要素を表 4-7 に示します。
6. 5.の element 要素の type 属性が complexType 要素で定義されます。element 要素の type 属性が配列型，および TP1 ユーザ構造体の場合に定義される complexType 要素を表 4-8 に示します。

表 4-7　.NET インタフェース定義から定義される element 要素

.NET インタフェース定義のサービスメソッドの引数，TP1 ユーザ構造体のメンバ，および戻り値	入力データ用 XML スキーマおよび出力データ用 XML スキーマの element 要素
System.Byte	<xs:element minOccurs="1" maxOccurs="1" name="paramName※1" type="xs:unsignedByte"/>
System.Int16	<xs:element minOccurs="1" maxOccurs="1" name="paramName※1" type="xs:short"/>
System.Int32	<xs:element minOccurs="1" maxOccurs="1" name="paramName※1" type="xs:int"/>
System.Int64	<xs:element minOccurs="1" maxOccurs="1" name="paramName※1" type="xs:long"/>
System.String	<xs:element minOccurs="0" maxOccurs="1" name="paramName※1" type="xs:string"/>
System.Byte[]	<xs:element minOccurs="0" maxOccurs="1" name="paramName※1" type="xs:base64Binary"/>
System.Int16[]	<xs:element minOccurs="0" maxOccurs="1" name="paramName※1" type="ArrayOfShort※2"/>
System.Int32[]	<xs:element minOccurs="0" maxOccurs="1" name="paramName※1" type="ArrayOfInt※2"/>
System.Int64[]	<xs:element minOccurs="0" maxOccurs="1" name="paramName※1" type="ArrayOfLong※2"/>
System.String[]	<xs:element minOccurs="0" maxOccurs="1" name="paramName※1" type="ArrayOfString※2"/>
TP1 ユーザ構造体	xs:element minOccurs="0" maxOccurs="1" name="paramName※1" type="〈TP1 ユーザ構造体名〉※2"/>
TP1 ユーザ構造体配列	<xs:element minOccurs="0" maxOccurs="1" name="paramName※1" type="ArrayOf〈TP1 ユーザ構造体名〉※2"/>

注※1

クライアントスタブ生成コマンド（if2cstub）の-X オプションに normal を指定した場合，次の値が指定されます。

- 引数の場合：.NET インタフェース定義のサービスメソッドの引数の名称
- TP1 ユーザ構造体のメンバの場合：TP1 ユーザ構造体のメンバの名称
- 戻り値の場合：returnValue

この XML スキーマ例を「4.2.4(10)　XML スキーマ例（クライアントスタブ生成コマンド（if2cstub）の-X オプションに normal を指定した場合 1)」に示します。

また，-X オプションに dataset を指定した場合，次の値が指定されます。

- 引数の場合：〈complexType 要素の name 属性の値〉_〈.NET インタフェース定義のサービスメソッドの引数の名称〉
- TP1 ユーザ構造体のメンバの場合：〈complexType 要素の name 属性の値〉_〈TP1 ユーザ構造体のメンバの名称〉
- 戻り値の場合：returnValue

この XML スキーマ例を「4.2.4(11)　XML スキーマ例（クライアントスタブ生成コマンド（if2cstub）の-X オプションに dataset を指定した場合 1)」に示します。

注※2

クライアントスタブ生成コマンド（if2cstub）で，-X オプションに normal を指定した場合の XML スキーマ例を「4.2.4(12)　XML スキーマ例（クライアントスタブ生成コマンド（if2cstub）の-X オプションに normal を指定した場合 2)」に示します。

また，クライアントスタブ生成コマンド（if2cstub）の-X オプションに dataset を指定した場合で，二つ以上の element 要素の type 属性が同一の complexType 要素で定義されたとき，Connector .NET によって element 要素の数だけ complexType 要素が定義されます。element 要素の type 属性の値は，「〈complexType 要素の name 属性の値〉〈同一の complexType 要素を定義した type 属性が定義されている element 要素の個数の連番〉」となります。element 要素の個数の連番は，0 から始まり，1，2…と割り当てられます。この XML スキーマ例を「4.2.4(13)　XML スキーマ例（クライアントスタブ生成コマンド（if2cstub）の-X オプションに dataset を指定した場合 2)」に示します。

表 4-8　element 要素の type 属性が配列型，および TP1 ユーザ構造体の場合に定義される complexType 要素

element 要素の type 属性	入力データ用 XML スキーマおよび出力データ用 XML スキーマの complexType 要素
ArrayOfShort （+連番（1，2，…））	<xs:complexType name="ArrayOfShort"> <xs:sequence> <xs:element minOccurs="0" maxOccurs="unbounded" name="short※1" type="xs:short"/> </xs:sequence> </xs:complexType>
ArrayOfInt （+連番（1，2，…））	<xs:complexType name="ArrayOfInt"> <xs:sequence> <xs:element minOccurs="0" maxOccurs="unbounded" name="int※1" type="xs:int"/> </xs:sequence> </xs:complexType>
ArrayOfLong	<xs:complexType name="ArrayOfLong">

element 要素の type 属性	入力データ用 XML スキーマおよび出力データ用 XML スキーマの complexType 要素
(+連番（1，2，…))	<xs:sequence> <xs:element minOccurs="0" maxOccurs="unbounded" name="long※1" type="xs:long"/> </xs:sequence> </xs:complexType>
ArrayOfString (+連番（1，2，…))	<xs:complexType name="ArrayOfString"> <xs:sequence> <xs:element minOccurs="0" maxOccurs="unbounded" name="string※1" nillable="true" type="xs:string"/> </xs:sequence> </xs:complexType>
TP1 ユーザ構造体名 (+連番（1，2，…))	<xs:complexType name="(TP1 ユーザ構造体名)"> <xs:sequence> <xs:element … 省略 … />(TP1 ユーザ構造体のメンバの要素※2) : </xs:sequence> </xs:complexType>
ArrayOf(TP1 ユーザ構造体名) (+連番（1，2，…))	<xs:complexType name="ArrayOf(TP1 ユーザ構造体名)"> <xs:sequence> <xs:element minOccurs="0" maxOccurs="unbounded" name="(TP1 ユーザ構造体名)※1" nillable="true" type="(TP1 ユーザ構造体名)"/> </xs:sequence> </xs:complexType>

注※1

クライアントスタブ生成コマンド（if2cstub）の-X オプションに dataset を指定した場合，「〈complexType 要素の name 属性の値〉_〈「xs:」を除いた type 属性の値〉」が指定されます。この XML スキーマ例を「4.2.4(14)　XML スキーマ例（クライアントスタブ生成コマンド（if2cstub）の-X オプションに dataset を指定した場合 3)」に示します。

注※2

.NET インタフェース定義の TP1 ユーザ構造体のメンバに指定される値については，表 4-7 を参照してください。

4.2.4　.NET インタフェース定義から生成したクライアントスタブの使用例，および XML スキーマ例

クライアントスタブの使用例，および XML スキーマ例を次に示します。

この例で呼び出す SPP.NET のサービスメソッドの情報は次のとおりです。

- サービスグループ名：GRP1
- インタフェース名：MyCompany.IGyoumuA

- 呼び出すサービスメソッド名（サービス名）：Service3
- 構成ファイル：指定あり（プロファイル ID="TP1Host1"）

なお，コメント中の(1)，(2)などは「4.2.2　クライアントスタブの使用方法」の説明の番号に対応しています。

(1)　.NET インタフェース定義の定義例（C#の場合）

```
using System;

namespace MyCompany
{
  public interface IGyoumuA
  {
    void Service1(string dataId, byte[] data);
    string[] Service2(string key);
    int Service3(int inCount, ref string[] ids);
  }
}
```

(2)　入力データ用 XML スキーマ例（IGyoumuA_Service3_InData.xsd）

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="IGyoumuA_Service3_InData"
   type="IGyoumuA_Service3_InData" />
  <xs:complexType name="IGyoumuA_Service3_InData">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="inCount"
       type="xs:int" />
      <xs:element minOccurs="0" maxOccurs="1" name="ids"
       type="ArrayOfString" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ArrayOfString">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded"
       name="string" nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

(3)　出力データ用 XML スキーマ例（IGyoumuA_Service3_OutData.xsd）

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="IGyoumuA_Service3_OutData"
   type="IGyoumuA_Service3_OutData" />
  <xs:complexType name="IGyoumuA_Service3_OutData">
```

```
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="1" name="ids"
       type="ArrayOfString" />
      <xs:element minOccurs="1" maxOccurs="1" name="returnValue"
       type="xs:int" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ArrayOfString">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded"
       name="string" nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

(4) クライアントスタブの使用例（ASP.NET Web アプリケーション，C#の場合）

```
using System;
using Hitachi.OpenTP1;
using Hitachi.OpenTP1.Connector;

namespace MyCompany
{
  public class MyForm1 : System.Web.UI.Page
  {
    ...
    private void Button1_Click(object sender, System.EventArgs e)
    {
      TP1Connection tc = null;
      // このボタンがクリックされたら
      // OpenTP1<TP1Host1>のサービス要求を行う
      try {
      // グローバル変数のアプリケーション状態から
      // TP1ConnectionManagerを取得
        // (1) TP1ConnectionManagerクラスの生成
        TP1ConnectionManager tcm =
          (TP1ConnectionManager)this.Application["tcmTP1Host1"];
        // (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection();
        // (3) クライアントスタブの生成
        IGyoumuAStub server = new IGyoumuAStub(tc, "GRP1");
        // (4) RPC呼び出し形態
        server.Flags = RpcInfo.DCNOFLAGS;
        // (4) 最大応答待ち時間
        server.WatchTime = 180;
        // (5) 入力データの設定
        string[] ids = new string[]{"data1","data2","data3"};
        // (6) Service3を呼び出す
        int index = server.Service3(3, ref ids);
        // (7) 戻り値，出力パラメタの参照
        this.textBox1.Text = ids[index];
      } catch (TP1UserException exp) {
        // Service3()からユーザ例外がスローされた
      } catch (TP1RemoteException exp) {
```

```
        // Service3()で予期しない例外発生
      } catch (TP1ConnectorException exp) {
        // Connector .NETが検知したエラー
      } catch (TP1Exception exp) {
        // その他スタブなどが検知したエラー
      } catch (Exception exp) {
        // 予期しない例外
      }
      finally
      {
        // (8) コネクションをコネクションプールに戻す
        if (tc != null) tc.Dispose();
      }
    }
  }
}
```

(5) クライアントスタブの使用例（ASP.NET Web アプリケーション，J#の場合）

```
package MyCompany;
import System.*;
import Hitachi.OpenTP1.*;
import Hitachi.OpenTP1.Client.*;

public class MyForm1 extends System.Web.UI.Page
{
    ...
    private void Button1_Click(Object sender, System.EventArgs e)
    {
      TP1Connection tc = null;
      // このボタンがクリックされたら
      // OpenTP1<TP1Host1>のサービス要求を行う
      try {
      // グローバル変数のアプリケーション状態から
      // TP1ConnectionManagerを取得
        // (1) TP1ConnectionManagerクラスの生成
        TP1ConnectionManager tcm =
        (TP1ConnectionManager)this.get_Application().
                              Get("tcmTP1Host1");
        // (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection();
        // (3) クライアントスタブの生成
        IGyoumuAStub server = new IGyoumuAStub(tc, "GRP1");
        // (4) RPC呼び出し形態
        server.set_Flags(RpcInfo.DCNOFLAGS);
        // (4) 最大応答待ち時間
        server.set_WatchTime(180);
        // (5) 入力データの設定
        StringArrayHolder idsHolder = new StringArrayHolder();
        // (5) 入力データの設定
        String[] ids = new String[]{"data1","data2","data3"};
        // (6) Service3を呼び出す
        int index;
        // (6) Service3を呼び出す
```

```
        index  = server.Service3(3, idsHolder);
        // (7) 戻り値，出力パラメタの参照
        this.textBox1.set_Text(idsHolder.get_Value()[index]);
      } catch (TP1UserException exp) {
        // Service3()からユーザ例外がスローされた
      } catch (TP1RemoteException exp) {
        // Service3()で予期しない例外発生
      } catch (TP1ConnectorException exp) {
        // Connector .NETが検知したエラー
      } catch (TP1Exception exp) {
        // その他スタブなどが検知したエラー
      } catch (System.Exception exp) {
        // 予期しない例外
      }
      finally
      {
        // (8) コネクションをコネクションプールに戻す
        if (tc != null) tc.Dispose();
      }
    }
}
```

(6) クライアントスタブの使用例（ASP.NET Web アプリケーション，Visual Basic の場合）

```
Imports System
Imports Hitachi.OpenTP1
Imports Hitachi.OpenTP1.Connector

Namespace MyCompany
  Public Class MyForm1
      Inherits System.Web.UI.Page
    ...
    Private Sub Button1_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Button1.Click
      Dim tcm As TP1ConnectionManager
      Dim tc As TP1Connection
      Dim ret As Integer
      Dim ids() As String
      Dim index As Integer
      ' このボタンがクリックされたら
      ' OpenTP1<TP1Host1>のサービス要求を行う
      Try
      ' グローバル変数のアプリケーション状態から
      ' TP1ConnectionManagerを取得
        ' (1) TP1ConnectionManagerクラスの生成
        tcm = CType(Application("tcmTP1Host1"), _
                    TP1ConnectionManager)
        ' (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection()
        ' (3) クライアントスタブの生成
        Dim server As IGyoumuAStub = New IGyoumuAStub(tc, "GRP1")
        ' (4) RPC呼び出し形態
        server.Flags = RpcInfo.DCNOFLAGS
        ' (4) 最大応答待ち時間
```

```
        server.WatchTime = 180
        ’(5) 入力データの設定
        ids = New String(){”data1”,”data2”,”data3”}
        ’(6) Service3を呼び出す
        index = server.Service3(3, ids)
        ’(7) 戻り値，出力パラメタの参照
        textBox1.Text = ids(index)
      Catch exp As TP1UserException
        ’Service3()からユーザ例外がスローされた
      Catch exp As TP1RemoteException
        ’Service3()で予期しない例外発生
      Catch exp As TP1ConnectorException
        ’Connector .NETが検知したエラー
      Catch exp As TP1Exception
        ’その他スタブなどが検知したエラー
      Catch exp As Exception
        ’予期しない例外
      Finally
        If Not(tc Is Nothing) Then
          ’(8) コネクションをコネクションプールに戻す
          tc.Dispose()
        End If
      End Try
    End Sub
  End Class
End Namespace
```

(7) クライアントスタブの使用例（ASP.NET Web アプリケーション，C#，RPC データの XML マッピング機能を使用した場合）

```
using System;
using System.Xml;
using Hitachi.OpenTP1;
using Hitachi.OpenTP1.Connector;

namespace MyCompany
{
  public class MyForm1 : System.Web.UI.Page
  {
    ...
    private void Button1_Click(object sender, System.EventArgs e)
    {
      TP1Connection tc = null;
      // このボタンがクリックされたら
      // OpenTP1<TP1Host1>のサービス要求を行う
      try {
      // グローバル変数のアプリケーション状態から
      // TP1ConnectionManagerを取得
        // (1) TP1ConnectionManagerクラスの生成
        TP1ConnectionManager tcm =
          (TP1ConnectionManager)this.Application[”tcmTP1Host1”];
        // (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection();
        // (3) クライアントスタブの生成
        IGyoumuAStub server = new IGyoumuAStub(tc, ”GRP1”);
```

```
        // (4) RPC呼び出し形態
        server.Flags = RpcInfo.DCNOFLAGS;
        // (4) 最大応答待ち時間
        server.WatchTime = 180;
        // (5) 入力データの設定
        XmlDocument inDoc = GetInputData();
        // (5) 入力データの設定
        inDoc.SelectSingleNode("/IGyoumuA_Service3_InData/inCount"
                               ).InnerText = "3";
        // (6) Service3ByXmlを呼び出す
        XmlDocument outDoc = server.Service3ByXml(inDoc);
        // (7) 戻り値，出力パラメタの参照
        String index =
         outDoc.SelectSingleNode(
         "/IGyoumuA_Service3_OutData/returnValue").InnerText;
        // (7) 戻り値，出力パラメタの参照
        this.textBox1.Text =
         outDoc.SelectSingleNode(
         "/IGyoumuA_Service3_OutData/ids["+index+"]").InnerText;
      } catch (TP1UserException exp) {
        // Service3()からユーザ例外がスローされた
      } catch (TP1RemoteException exp) {
        // Service3()で予期しない例外発生
      } catch (TP1ConnectorException exp) {
        // Connector .NETが検知したエラー
      } catch (TP1Exception exp) {
        // その他スタブなどが検知したエラー
      } catch (Exception exp) {
        // 予期しない例外
      }
      finally
      {
        // (8) コネクションをコネクションプールに戻す
        if (tc != null) tc.Dispose();
      }
    }
  }
}
```

(8) クライアントスタブの使用例（ASP.NET Web アプリケーション，J#，RPC データの XML マッピング機能を使用した場合）

```
package MyCompany;
import System.*;
import System.Xml.*;
import Hitachi.OpenTP1.*;
import Hitachi.OpenTP1.Client.*;

public class MyForm1 extends System.Web.UI.Page
{
    ...
    private void Button1_Click(Object sender, System.EventArgs e)
    {
      TP1Connection tc = null;
      // このボタンがクリックされたら
```

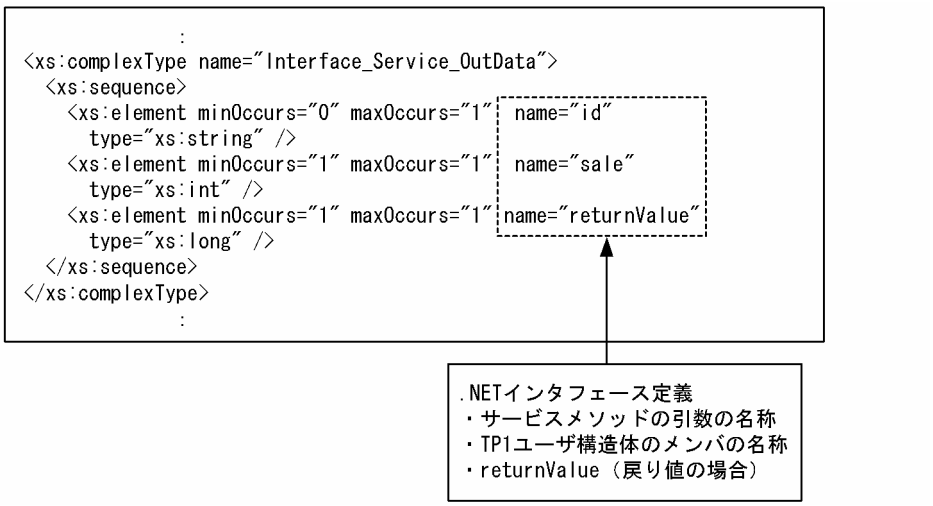
```
      // OpenTP1<TP1Host1>のサービス要求を行う
      try {
      // グローバル変数のアプリケーション状態から
      // TP1ConnectionManagerを取得
        // (1) TP1ConnectionManagerクラスの生成
        TP1ConnectionManager tcm =
         (TP1ConnectionManager)this.get_Application().
                              Get("tcmTP1Host1");
        // (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection();
        // (3) クライアントスタブの生成
        IGyoumuAStub server = new IGyoumuAStub(tc, "GRP1");
        // (4) RPC呼び出し形態
        server.set_Flags(RpcInfo.DCNOFLAGS);
        // (4) 最大応答待ち時間
        server.set_WatchTime(180);
        // (5) 入力データの設定
        XmlDocument inDoc = GetInputData();
        // (5) 入力データの設定
        inDoc.SelectSingleNode("/IGyoumuA_Service3_InData/inCount"
                              ).set_InnerText("3");
        // (6) Service3ByXmlを呼び出す
        XmlDocument outDoc = server.Service3ByXml(inDoc);
        // (7) 戻り値，出力パラメタの参照
        String index =
        outDoc.SelectSingleNode(
        "/IGyoumuA_Service3_OutData/returnValue").get_InnerText();
        // (7) 戻り値，出力パラメタの参照
        this.textBox1.set_Text(
        outDoc.SelectSingleNode(
        "/IGyoumuA_Service3_OutData/ids["+index+"]"
                               ).get_InnerText());
      } catch (TP1UserException exp) {
        // Service3()からユーザ例外がスローされた
      } catch (TP1RemoteException exp) {
        // Service3()で予期しない例外発生
      } catch (TP1ConnectorException exp) {
        // Connector .NETが検知したエラー
      } catch (TP1Exception exp) {
        // その他スタブなどが検知したエラー
      } catch (System.Exception exp) {
        // 予期しない例外
      }
      finally
      {
        // (8) コネクションをコネクションプールに戻す
        if (tc != null) tc.Dispose();
      }
    }
}
```

(9) クライアントスタブの使用例（ASP.NET Web アプリケーション，Visual Basic，RPC データの XML マッピング機能を使用した場合）

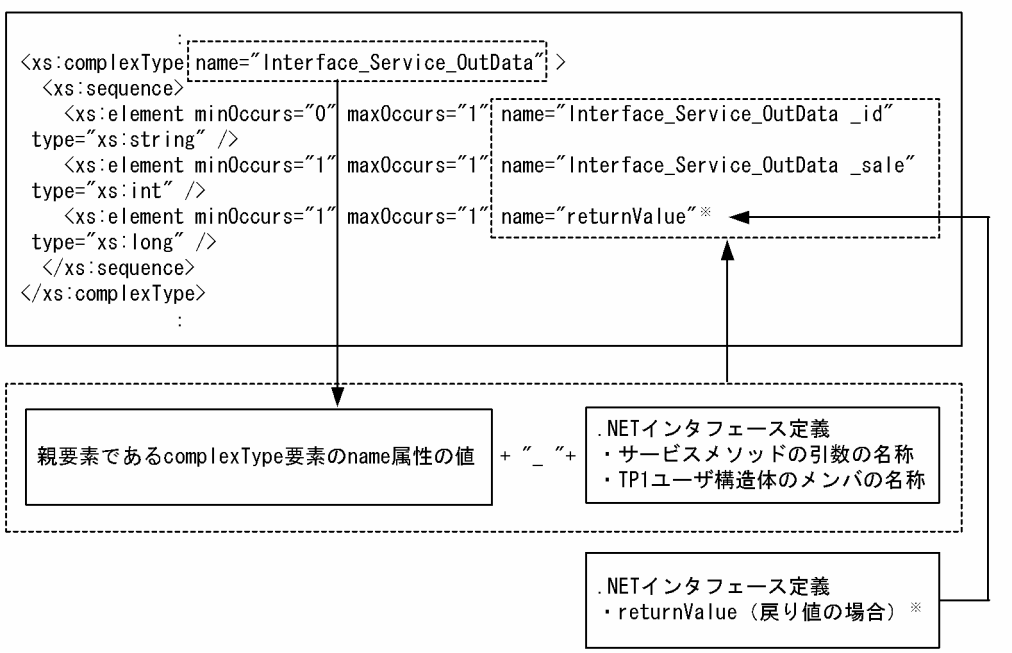
```
Imports System
Imports System.Xml
Imports Hitachi.OpenTP1
Imports Hitachi.OpenTP1.Connector

Namespace MyCompany
  Public Class MyForm1
      Inherits System.Web.UI.Page
    ...
    Private Sub Button1_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Button1.Click
      Dim tcm As TP1ConnectionManager
      Dim tc As TP1Connection
      Dim ret As Integer
      Dim ids() As String
      Dim index As String
      Dim inDoc, outDoc As XmlDocument
      ’ このボタンがクリックされたら
      ’ OpenTP1<TP1Host1>のサービス要求を行う
      Try
      ’ グローバル変数のアプリケーション状態から
      ’ TP1ConnectionManagerを取得
        ’ (1) TP1ConnectionManagerクラスの生成
        tcm = CType(Application(”tcmTP1Host1”), _
                    TP1ConnectionManager)
        ’ (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection()
        ’ (3) クライアントスタブの生成
        Dim server As IGyoumuAStub = New IGyoumuAStub(tc, ”GRP1”)
        ’ (4) RPC呼び出し形態
        server.Flags = RpcInfo.DCNOFLAGS
        ’ (4) 最大応答待ち時間
        server.WatchTime = 180
        ’ (5) 入力データの設定
        inDoc = GetInputData()
        ’ (5) 入力データの設定
        inDoc.SelectSingleNode( _
          ”/IGyoumuA_Service3_InData/inCount”).InnerText = ”3”
        ’ (6) Service3ByXmlを呼び出す
        outDoc = server.Service3ByXml(inDoc)
        ’ (7) 戻り値，出力パラメタの参照
        index = outDoc.SelectSingleNode( _
         ”/IGyoumuA_Service3_OutData/returnValue”).InnerText
        ’ (7) 戻り値，出力パラメタの参照
        this.textBox1.Text = outDoc.SelectSingleNode( _
          ”/IGyoumuA_Service3_OutData/ids[”+index+”]”).InnerText
      Catch exp As TP1UserException
        ’ Service3()からユーザ例外がスローされた
      Catch exp As TP1RemoteException
        ’ Service3()で予期しない例外発生
      Catch exp As TP1ConnectorException
        ’ Connector .NETが検知したエラー
      Catch exp As TP1Exception
```

```
        ’その他スタブなどが検知したエラー
      Catch exp As Exception
        ’予期しない例外
      Finally
        If Not(tc Is Nothing) Then
          ’(8) コネクションをコネクションプールに戻す
          tc.Dispose()
        End If
      End Try
    End Sub
  End Class
End Namespace
```

(10) XML スキーマ例（クライアントスタブ生成コマンド（if2cstub）の-X オプションに normal を指定した場合 1）



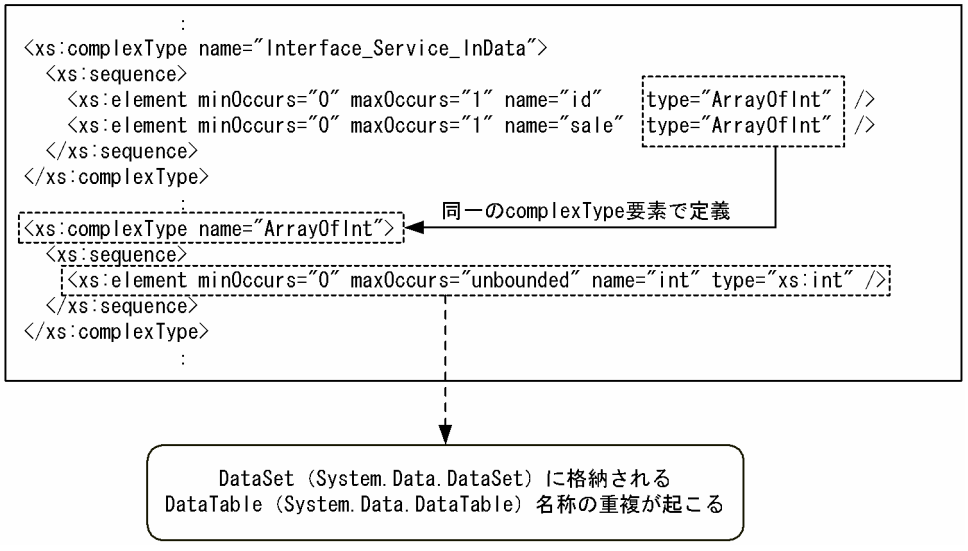
(11) XML スキーマ例（クライアントスタブ生成コマンド（if2cstub）の-X オプションに dataset を指定した場合 1）



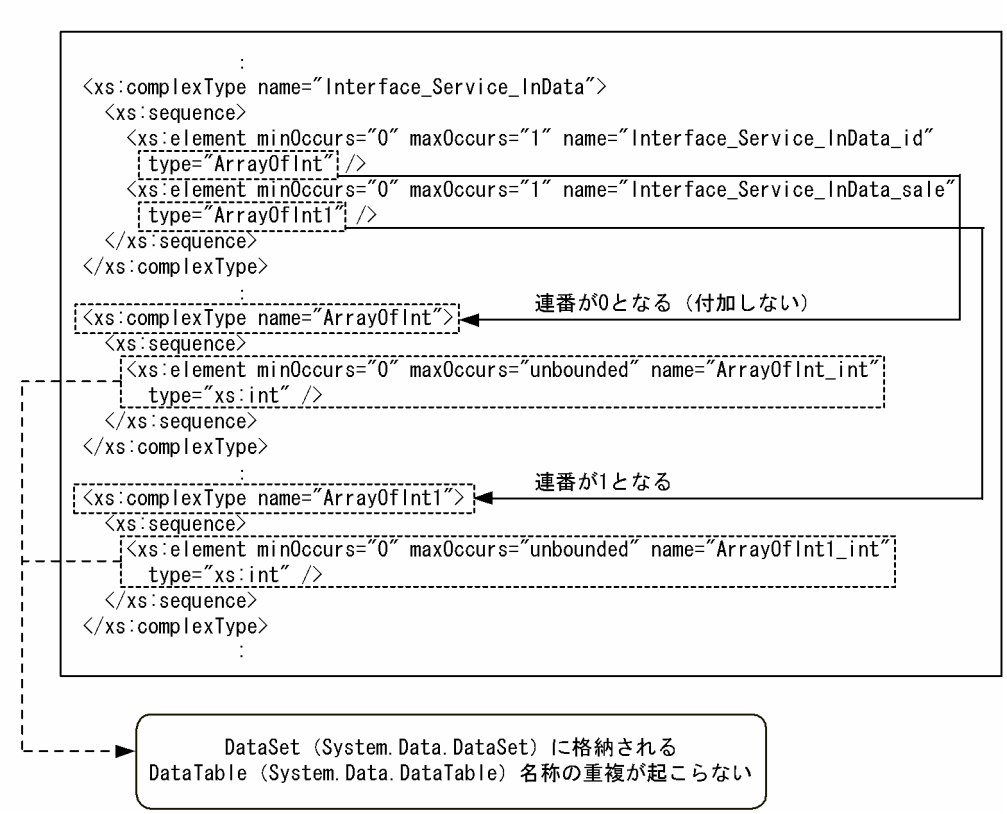
注※

親要素である complexType 要素の name 属性の値は付加しないで，returnValue だけが指定されます。

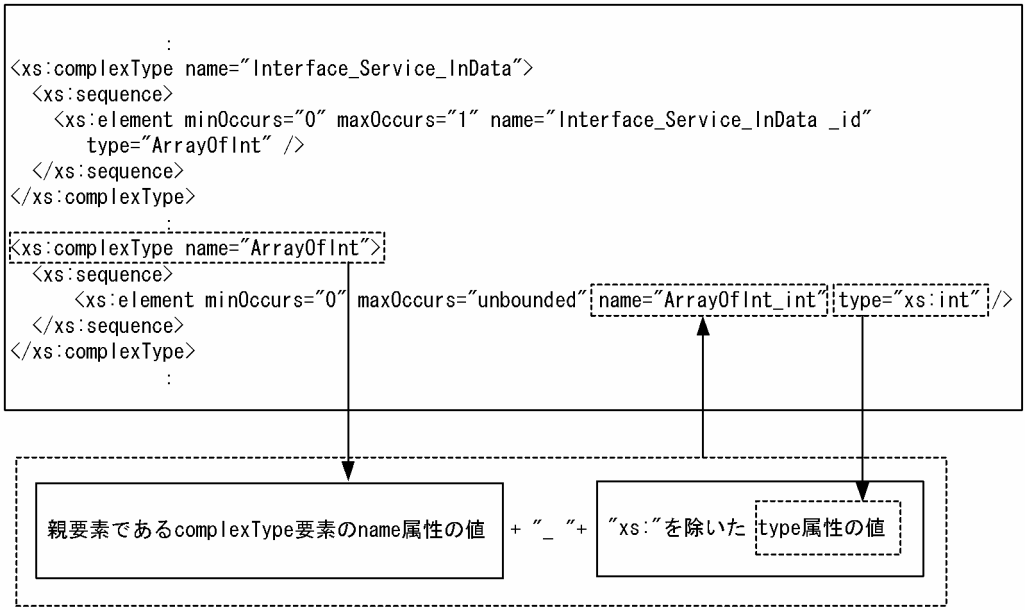
(12) XML スキーマ例（クライアントスタブ生成コマンド（if2cstub）の-X オプションに normal を指定した場合 2）



(13) XML スキーマ例（クライアントスタブ生成コマンド（if2cstub）の-X オプションに dataset を指定した場合 2）



(14) XML スキーマ例（クライアントスタブ生成コマンド（if2cstub）の-X オプションに dataset を指定した場合 3）



4.3　サービス定義を使用した SPP.NET または SPP の呼び出し方法

Connector .NET のアプリケーションからサービス定義を使用して SPP.NET または SPP を呼び出す方法について説明します。

Extension .NET，Client .NET の各 UAP からの呼び出し方法については，それぞれマニュアル「TP1/Extension for .NET Framework 使用の手引」，マニュアル「TP1/Client for .NET Framework 使用の手引」のサービス定義を使用した SPP.NET または SPP の呼び出し方法についての記述を参照してください。なお，次のプログラム言語が使用できます。COBOL 言語は使用できません。

- C#
- J#※
- Visual Basic

注※

Visual Studio 2013 以降では，J#を使用して開発できません。

4.3.1　クライアントスタブの生成

クライアントスタブ生成コマンド（spp2cstub）を使用して，サービス定義からクライアントスタブおよびカスタムレコードクラスを生成します。

クライアントスタブおよびカスタムレコードクラスは，クライアントスタブを利用するクライアントアプリケーション（Connector .NET のアプリケーション）を記述するプログラム言語で生成してください。

データ型定義で指定された各メンバは，カスタムレコードクラスの public プロパティとなります。

クライアントスタブ生成コマンド（spp2cstub）で-X オプションを指定して，サービス定義からクライアントスタブを生成すると，引数および戻り値が XmlDocument クラス（System.Xml.XmlDocument）となるサービスメソッドがクライアントスタブに追加されます。このサービスメソッドの名称は，「〈サービス定義で指定されたサービス名称〉ByXml」となります。

サービスメソッドの引数および戻り値の XmlDocument クラスに指定できる XML の形式は，クライアントスタブと一緒に生成される入力データ用 XML スキーマおよび出力データ用 XML スキーマで定義されます。XML スキーマの詳細については，「4.1.1(4)　.NET インタフェース定義から XML スキーマへのマッピング」，および「4.1.2(3)　サービス定義から XML スキーマへのマッピング」を参照してください。

サービス定義，入力データ用 XML スキーマおよび出力データ用 XML スキーマ，ならびにクライアントスタブに追加されたサービスメソッドの引数および戻り値の関係を次の表に示します。

表 4-9　サービス定義，定義される XML スキーマ，ならびにクライアントスタブのサービスメソッドの引数および戻り値の関係

サービス定義		定義される XML スキーマ	出力されるサービスメソッド（クライアントスタブ）	
			引数	戻り値
入力データ型定義		入力データ用 XML スキーマ	XmlDocument	－
出力データ型定義	あり	出力データ用 XML スキーマ	－	XmlDocument
	DC_NODATA	なし	－	なし

（凡例）

－：該当しません。

4.3.2　クライアントスタブの使用方法

クライアントスタブを使用してサービス要求をする手順を次に示します。

1. TP1ConnectionManager クラスのインスタンスを生成，または取り出します。

   プロパティを特定のプロファイルから読み込む場合はコンストラクタの引数にプロファイル ID を指定します。インスタンスの生成はアプリケーションごとに行うことを推奨します。

2. TP1ConnectionManager クラスの GetConnection メソッドで TP1Connection オブジェクトを取得します。

   コネクションプールに使用できるコネクションがあった場合は，この時点でコネクションプールからコネクションを取り出した状態になります。

3. クライアントスタブを TP1Connection クラスのインスタンスごと，およびサービスグループごとにインスタンス化します。

4. 必要に応じてプロパティ（Flags，WatchTime）を設定します。

   クライアントスタブの初期値は Flags に RpcInfo.DCNOFLAGS，WatchTime に-1 が設定されています。Flags プロパティ，および WatchTime プロパティの詳細については，RpcInfo クラスの各プロパティの説明を参照してください。

5. カスタムレコードを引数とするサービスメソッドの場合，入力用および出力用のカスタムレコードクラスをインスタンス化します。XmlDocument を引数とするサービスメソッドの場合，引数用の XmlDocument クラスをインスタンス化します。

6. カスタムレコードを引数とするサービスメソッドの場合，入力用カスタムレコードのプロパティに入力データを設定します。XmlDocument を引数とするサービスメソッドの場合，引数用の XmlDocument に入力データ用 XML スキーマに従った XML 文書の内容を設定します。

7. サービスメソッドを呼び出します。

8. カスタムレコードを引数とするサービスメソッドの場合，出力用カスタムレコードのプロパティから応答データを参照します。XmlDocument を戻り値とするサービスメソッドの場合，出力データ用 XML スキーマに従って，返された XML 文書の内容を参照します。

9. TP1Connection クラスの Dispose メソッドを発行して，コネクションをコネクションプールに戻します。

   複数の TP1Connection オブジェクトを使う場合は，それぞれのインスタンスが不要になった時点で Dispose メソッドを発行します。

注意事項

ASP.NET Web アプリケーションで使用する場合，TP1Connection クラスの Dispose メソッドを，次の時点で必ず発行してください。

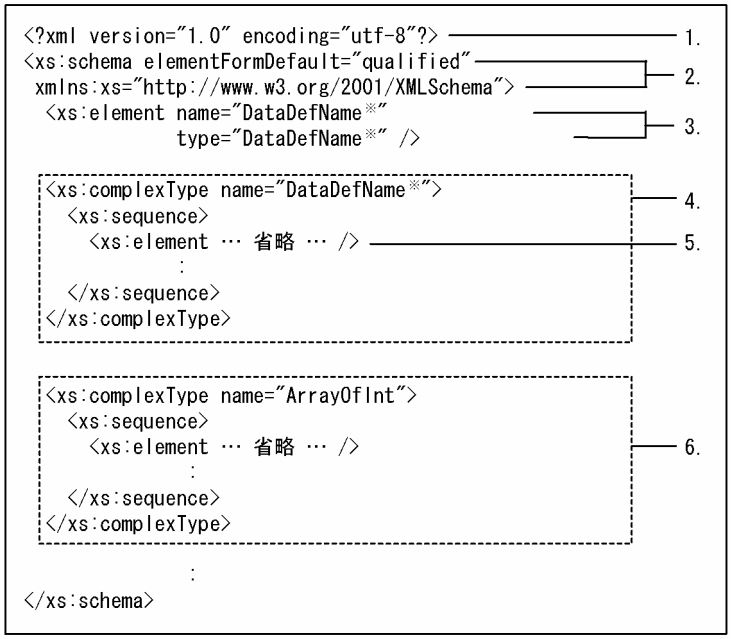
- 各 UI イベントハンドラの終了時
- Web サービスメソッドの終了時

Dispose メソッドを適切に発行しないと，メモリリークの原因となります。

また，TP1ConnectionManager クラスは，ASP.NET ではアプリケーション状態，またはアプリケーションインスタンスに保持することを推奨します。各 UI イベントハンドラや Web サービスメソッドごとに生成と削除を繰り返した場合，性能劣化やメモリ使用効率低下の原因となります。

4.3.3 XML スキーマの定義方法

入力データ用 XML スキーマおよび出力データ用 XML スキーマの定義形式を次に示します。



注※

すべて同じ値が指定されます。

1. XML 宣言ノードの version 属性には 1.0，encoding 属性には utf-8 が指定されます。
2. schema 要素の elementFormDefault 属性には qualified，xmlns 属性には名前空間 URI として「http://www.w3.org/2001/XMLSchema」が指定されます。
3. ルート要素となる element 要素の name 属性には，入力データ用 XML スキーマの場合，「入力データ型定義名称」または「出力データ型定義名称」が指定されます。type 属性には，name 属性と同じ値が指定されます。
4. 3.の type 属性が complexType 要素で定義されます。complexType 要素の name 属性には，3.の type 属性と同じ値が指定されます。
5. sequence 要素の子要素として element 要素が定義されます。element 要素は，入力データ型定義または出力データ型定義のメンバ，および構造体のメンバによって定義されます。データ型定義のメンバおよび構造体のメンバから定義される element 要素を表 4-10 に示します。
6. 5.の element 要素の type 属性が complexType 要素で定義されます。element 要素の type 属性が配列型，および構造体の場合に定義される complexType 要素を表 4-11 に示します。

表 4-10　データ型定義から定義される element 要素

データ型定義のメンバおよび構造体のメンバのデータ型	入力データ用 XML スキーマおよび出力データ用 XML スキーマの element 要素
char	<xs:element minOccurs="0" maxOccurs="1" name="memberName※1" type="xs:string"/>
int	<xs:element minOccurs="1" maxOccurs="1"

データ型定義のメンバおよび構造体のメンバのデータ型	入力データ用 XML スキーマおよび出力データ用 XML スキーマの element 要素
	name="memberName※1" type="xs:int"/>
short	<xs:element minOccurs="1" maxOccurs="1" name="memberName※1" type="xs:short"/>
long	<xs:element minOccurs="1" maxOccurs="1" name="memberName※1" type="xs:int"/>
struct	<xs:element minOccurs="0" maxOccurs="1" name="memberName※1" type="(構造体名称)"/>
char[]	<xs:element minOccurs="0" maxOccurs="1" name="memberName※1" type="xs:string"/>
int[]	<xs:element minOccurs="0" maxOccurs="1" name="memberName※1" type="ArrayOfInt※2"/>
short[]	<xs:element minOccurs="0" maxOccurs="1" name="memberName※1" type="ArrayOfShort※2"/>
long[]	<xs:element minOccurs="0" maxOccurs="1" name="memberName※1" type="ArrayOfInt※2"/>
struct[]	<xs:element minOccurs="0" maxOccurs="1" name="memberName※1" type="ArrayOf(構造体名称)※2"/>
char[][]	<xs:element minOccurs="0" maxOccurs="1" name="memberName※1" type="ArrayOfString※2"/>
byte[]	<xs:element minOccurs="0" maxOccurs="1" name="memberName※1" type="xs:base64Binary"/>

注※1

クライアントスタブ生成コマンド（spp2cstub）で-X オプションに normal を指定した場合，データ型定義のメンバの名称が指定されます。この XML スキーマ例を「4.3.4(11)　XML スキーマ例（クライアントスタブ生成コマンド（spp2cstub）の-X オプションに normal を指定した場合 1)」に示します。

また，-X オプションに dataset を指定した場合，〈complexType 要素の name 属性の値〉_〈データ型定義のメンバの名称〉が指定されます。この XML スキーマ例を「4.3.4(12)　XML スキーマ例（クライアントスタブ生成コマンド（spp2cstub）の-X オプションに dataset を指定した場合 1)」に示します。

注※2

クライアントスタブ生成コマンド（spp2cstub）で，-X オプションに normal を指定した場合の XML スキーマ例を「4.3.4(13)　XML スキーマ例（クライアントスタブ生成コマンド（spp2cstub）の-X オプションに normal を指定した場合 2)」に示します。

また，クライアントスタブ生成コマンド（spp2cstub）の-X オプションに dataset を指定した場合で，二つ以上の element 要素の type 属性が同一の complexType 要素で定義されたとき，Connector .NET によって element 要素の数だけ complexType 要素が定義されます。element 要素の type 属性の値は，「〈complexType 要素の name 属性の値〉〈同一の complexType 要素を定義した type 属性が定義されている element 要素の個数の連番〉」となります。element 要素の個数の連番は，0 から始まり，1，2…と割り当てられます。この XML スキーマ例を「4.3.4(14)　XML スキーマ例（クライアントスタブ生成コマンド（spp2cstub）の-X オプションに dataset を指定した場合 2)」に示します。

表 4-11　element 要素の type 属性が配列型，および構造体の場合に定義される complexType 要素

element 要素の type 属性	入力データ用 XML スキーマおよび出力データ用 XML スキーマの complexType 要素
ArrayOfInt （+連番（1，2，…））	<xs:complexType name="ArrayOfInt"> 　<xs:sequence> 　　<xs:element minOccurs="0" maxOccurs="unbounded" 　　name="int※1" type="xs:int"/> 　</xs:sequence> </xs:complexType>
ArrayOfShort （+連番（1，2，…））	<xs:complexType name="ArrayOfShort"> 　<xs:sequence> 　　<xs:element minOccurs="0" maxOccurs="unbounded" 　　name="short※1" type="xs:short"/> 　</xs:sequence> </xs:complexType>
ArrayOfString （+連番（1，2，…））	<xs:complexType name="ArrayOfString"> 　<xs:sequence> 　　<xs:element minOccurs="0" maxOccurs="unbounded" 　　name="string※1" nillable="true" type="xs:string"/> 　</xs:sequence> </xs:complexType>
構造体名 （+連番（1，2，…））	<xs:complexType name="(構造体名)"> 　<xs:sequence> 　　<xs:element … 省略 … /> (構造体のメンバの要素※2) 　　: 　</xs:sequence> </xs:complexType>
ArrayOf(構造体名) （+連番（1，2，…））	<xs:complexType name="ArrayOf(構造体名)"> 　<xs:sequence> 　　<xs:element minOccurs="0" maxOccurs="unbounded" 　　name="(構造体名)" nillable="true"type="(構造体名)"/> 　</xs:sequence> </xs:complexType>

注※1

クライアントスタブ生成コマンド（spp2cstub）の-X オプションに dataset を指定した場合，「〈complexType 要素の name 属性の値〉_〈「xs:」を除いた type 属性の値〉」が指定されます。この XML スキーマ例を「4.3.4(15)　XML スキーマ例（クライアントスタブ生成コマンド（spp2cstub）の-X オプションに dataset を指定した場合 3)」に示します。

注※2

データ型定義の構造体のメンバに指定される要素については，表 4-10 を参照してください。

4.3.4　サービス定義から生成したクライアントスタブの使用例，およびXMLスキーマ例

クライアントスタブの使用例，およびXMLスキーマ例を次に示します。

この例で呼び出すSPPのサービスの情報は次のとおりです。

- サービスグループ名：SVGRP1
- サービス定義名称：GYOUMU1
- 呼び出すサービス名（サービス名）：GETDATA1
- 入力用カスタムレコードクラス名：in_data
- 出力用カスタムレコードクラス名：out_data

なお，コメント中の(1)，(2)などは「4.3.2　クライアントスタブの使用方法」の説明の番号に対応しています。

(1)　サービス定義の定義例

(a)　サービス定義の定義例（業務1のサービス定義）

```
#include ”mydata.h”
/* 業務1のサービス定義 */
interface GYOUMU1 {
    GETDATA1(in_data, out_data);
    GETDATA2(in_data, out_data2);
}
```

(b)　データ型定義の定義例（mydata.h）

```
struct in_data {
        long i_basho[3];
        long i_kakaku;
};
struct out_data {
        char o_name[20];
        char o_basho[16];
        long o_kakaku;
        char o_inf[80];
};
struct out_data2 {
        int o_count;
        struct data {
                char o_name[20];
                char o_basho[16];
                long o_kakaku;
                char o_inf[80];
        } data_t[100];
};
```

(2) カスタムレコードクラスの出力例（C#の場合，in_data.cs）

```
using Hitachi.OpenTP1.Common;

namespace MyCompany
{
    public class in_data : RecordImpl
    {
        public in_data() : base("default")
        {
            …
        }
        public in_data(string recordName) : base(recordName)
        {
            …
        }
        …
        private int[] _I_basho;
        public int[] I_basho
        {
            get
            {
                return _I_basho;
            }
            set
            {
                _I_basho = value;
            }
        }

        private int _I_kakaku = 0;
        public int I_kakaku
        {
            get
            {
                return _I_kakaku;
            }
            set
            {
                _I_kakaku = value;
            }
        }
        …
    }
}
```

(3) 入力データ用 XML スキーマ例（in_data.xsd）

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="in_data" type="in_data" />
  <xs:complexType name="in_data">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="1" name="i_basho"
```

```
       type="ArrayOfInt" />
      <xs:element minOccurs="1" maxOccurs="1" name="i_kakaku"
       type="xs:int" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ArrayOfInt">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded"
       name="int" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

(4) 出力データ用 XML スキーマ例（out_data.xsd）

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="out_data" type="out_data" />
  <xs:complexType name="out_data">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="1" name="o_name"
       type="xs:string" />
      <xs:element minOccurs="0" maxOccurs="1" name="o_basho"
       type="xs:string" />
      <xs:element minOccurs="1" maxOccurs="1" name="o_kakaku"
       type="xs:int" />
      <xs:element minOccurs="0" maxOccurs="1" name="o_inf"
       type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

(5) クライアントスタブの使用例（ASP.NET Web アプリケーション，C#の場合）

```
using System;
using Hitachi.OpenTP1;
using Hitachi.OpenTP1.Connector;

namespace MyCompany
{
  public class MyForm1 : System.Web.UI.Page
  {
    ...
    private void Button1_Click(object sender, System.EventArgs e)
    {
      TP1Connection tc = null;
      // このボタンがクリックされたら
      // OpenTP1<TP1Host1>のサービス要求を行う
      try {
        // グローバル変数のアプリケーション状態から
        // TP1ConnectionManagerを取得
        // (1) TP1ConnectionManagerクラスの生成
```

```
        TP1ConnectionManager tcm =
          (TP1ConnectionManager)this.Application["tcmTP1Host1"];
        // (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection();
        // (3) クライアントスタブの生成
        GYOUMU1Stub server = new GYOUMU1Stub(tc, "SVGRP1");
        // (4) RPC呼び出し形態
        server.Flags = RpcInfo.DCNOFLAGS;
        // (4) 最大応答待ち時間
        server.WatchTime = 180;
        // (5) 入力用カスタムレコードの生成
        in_data inRecord = new in_data();
        // (5) 出力用カスタムレコードの生成
        out_data outRecord = new out_data();
        inRecord.i_basho[0] = 56;  // (6) 入力データの設定
        inRecord.i_basho[1] = 43;  // (6) 入力データの設定
        inRecord.i_basho[2] = 18;  // (6) 入力データの設定
        // (7) GETDATA1を呼び出す
        server.GETDATA1(inRecord, outRecord);
        // (8) 応答データの取り出し
        this.textBox1.Text = outRecord.o_name.Trim();
      } catch (TP1ConnectorException exp) {
        // Connector .NETが検知したエラー
      } catch (TP1Exception exp) {
        // その他スタブなどが検知したエラー
      } catch (Exception exp) {
        // 予期しない例外
      }
      finally
      {
        // (9) コネクションをコネクションプールに戻す
        if (tc != null) tc.Dispose();
      }
    }
  }
}
```

(6) クライアントスタブの使用例（ASP.NET Web アプリケーション，J#の場合）

```
package MyCompany;
import System;
import Hitachi.OpenTP1;
import Hitachi.OpenTP1.Client;

  public class MyForm1 extends System.Web.UI.Page
  {
    ...
    private void Button1_Click(Object sender, System.EventArgs e)
    {
      TP1Connection tc = null;
      // このボタンがクリックされたら
      // OpenTP1<TP1Host1>のサービス要求を行う
      try {
      // グローバル変数のアプリケーション状態から
```

```
      // TP1ConnectionManagerを取得
        // (1) TP1ConnectionManagerクラスの生成
        TP1ConnectionManager tcm =
        (TP1ConnectionManager)this.get_Application().
                                  Get("tcmTP1Host1");
        // (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection();
        // (3) クライアントスタブの生成
        GYOUMU1Stub server = new GYOUMU1Stub(tc, "SVGRP1");
        // (4) RPC呼び出し形態
        server.set_Flags(RpcInfo.DCNOFLAGS);
        // (4) 最大応答待ち時間
        server.set_WatchTime(180);
        // (5) 入力用カスタムレコードの生成
        in_data inRecord = new in_data();
        // (5) 出力用カスタムレコードの生成
        out_data outRecord = new out_data();
        inRecord.get_i_basho()[0] = 56;  // (6) 入力データの設定
        inRecord.get_i_basho()[1] = 43;  // (6) 入力データの設定
        inRecord.get_i_basho()[2] = 18;  // (6) 入力データの設定
        // (7) GETDATA1を呼び出す
        server.GETDATA1(inRecord, outRecord);
        // (8) 応答データの取り出し
        this.textBox1.set_Text(outRecord.get_o_name().Trim());
      } catch (TP1ConnectorException exp) {
        // Connector .NETが検知したエラー
      } catch (TP1Exception exp) {
        // その他スタブなどが検知したエラー
      } catch (System.Exception exp) {
        // 予期しない例外
      }
      finally
      {
        // (9) コネクションをコネクションプールに戻す
        if (tc != null) tc.Dispose();
      }
    }
}
```

(7) クライアントスタブの使用例（ASP.NET Web アプリケーション，Visual Basic の場合）

```
Imports System
Imports Hitachi.OpenTP1
Imports Hitachi.OpenTP1.Connector

Namespace MyCompany
  Public Class MyForm1
      Inherits System.Web.UI.Page
    ...
    Private Sub Button1_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles Button1.Click
      Dim tcm As TP1ConnectionManager
      Dim tc As TP1Connection
      Dim ret As Integer
```

```
      Dim ids(2) As String
      ' このボタンがクリックされたら
      ' OpenTP1<TP1Host1>のサービス要求を行う
      Try
      ' グローバル変数のアプリケーション状態から
      ' TP1ConnectionManagerを取得
        ' (1) TP1ConnectionManagerクラスの生成
        tcm = CType(Application("tcmTP1Host1"), _
                    TP1ConnectionManager)
        ' (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection()
        ' (3) クライアントスタブの生成
        Dim server As GYOUMU1Stub = New GYOUMU1Stub(tc, "SVGRP1")
        ' (4) RPC呼び出し形態
        server.Flags = RpcInfo.DCNOFLAGS
        ' (4) 最大応答待ち時間
        server.WatchTime = 180
        ' (5) 入力用カスタムレコードの生成
        Dim inRecord As in_data = New in_data
        ' (5) 出力用カスタムレコードの生成
        Dim outRecord As out_data = New out_data
        inRecord.i_basho(0) = 56  ' (6) 入力データの設定
        inRecord.i_basho(1) = 43  ' (6) 入力データの設定
        inRecord.i_basho(2) = 18  ' (6) 入力データの設定
        ' (7) GETDATA1を呼び出す
        server.GETDATA1(inRecord, outRecord)
        ' (8) 応答データの取り出し
        textBox1.Text = outRecord.o_name.Trim()
      Catch exp As TP1ConnectorException
        ' Connector .NETが検知したエラー
      Catch exp As TP1Exception
        ' その他スタブなどが検知したエラー
      Catch exp As Exception
        ' 予期しない例外
      Finally
        If Not(tc Is Nothing) Then
        ' (9) コネクションをコネクションプールに戻す
        tc.Dispose()
      End If
      End Try
    End Sub
  End Class
End Namespace
```

(8) クライアントスタブの使用例（ASP.NET Web アプリケーション，C#，RPC データの XML マッピング機能を使用した場合）

```
using System;
using System.Xml;
using Hitachi.OpenTP1;
using Hitachi.OpenTP1.Connector;

namespace MyCompany
{
  public class MyForm1 : System.Web.UI.Page
```

```
  {
    ...
    private void Button1_Click(object sender, System.EventArgs e)
    {
      TP1Connection tc = null;
      // このボタンがクリックされたら
      // OpenTP1<TP1Host1>のサービス要求を行う
      try {
        // グローバル変数のアプリケーション状態から
        // TP1ConnectionManagerを取得
        // (1) TP1ConnectionManagerクラスの生成
        TP1ConnectionManager tcm =
          (TP1ConnectionManager)this.Application["tcmTP1Host1"];
        // (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection();
        // (3) クライアントスタブの生成
        GYOUMU1Stub server = new GYOUMU1Stub(tc, "SVGRP1");
        // (4) RPC呼び出し形態
        server.Flags = RpcInfo.DCNOFLAGS;
        // (4) 最大応答待ち時間
        server.WatchTime = 180;
        // (5) 入力用カスタムレコードの生成
        XmlDocument inDoc = GetInputData();
        // (6) 入力データの設定
        inDoc.SelectSingleNode(
          "/in_data/i_basho[1]").InnerText = "56";
        // (6) 入力データの設定
        inDoc.SelectSingleNode(
          "/in_data/i_basho[2]").InnerText = "43";
        // (6) 入力データの設定
        inDoc.SelectSingleNode(
          "/in_data/i_basho[3]").InnerText = "18";
        // (7) GETDATA1ByXmlを呼び出す
        XmlDocument outDoc = server.GETDATA1ByXml(inDoc);
        // (8) 応答データの取り出し
        this.textBox1.Text =
          outDoc.SelectSingleNode("/out_data/o_name"
                                  ).InnerText.Trim();
      } catch (TP1ConnectorException exp) {
        // Connector .NETが検知したエラー
      } catch (TP1Exception exp) {
        // その他スタブなどが検知したエラー
      } catch (Exception exp) {
        // 予期しない例外
      }
      finally
      {
        // (9) コネクションをコネクションプールに戻す
        if (tc != null) tc.Dispose();
      }
    }
  }
}
```

(9) クライアントスタブの使用例（ASP.NET Web アプリケーション，J#，RPC データの XML マッピング機能を使用した場合）

```
package MyCompany;
import System.*;
import System.Xml.*;
import Hitachi.OpenTP1.*;
import Hitachi.OpenTP1.Connector.*;

  public class MyForm1 extends System.Web.UI.Page
  {
    ...
    private void Button1_Click(Object sender, System.EventArgs e)
    {
      TP1Connection tc = null;
      // このボタンがクリックされたら
      // OpenTP1<TP1Host1>のサービス要求を行う
      try {
      // グローバル変数のアプリケーション状態から
      // TP1ConnectionManagerを取得
        // (1) TP1ConnectionManagerクラスの生成
        TP1ConnectionManager tcm =
          (TP1ConnectionManager)this.get_Application().
                                  Get("tcmTP1Host1");
        // (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection();
        // (3) クライアントスタブの生成
        GYOUMU1Stub server = new GYOUMU1Stub(tc, "SVGRP1");
        // (4) RPC呼び出し形態
        server.set_Flags(RpcInfo.DCNOFLAGS);
        // (4) 最大応答待ち時間
        server.set_WatchTime(180);
        // (5) 入力用カスタムレコードの生成
        XmlDocument inDoc = GetInputData();
        // (6) 入力データの設定
        inDoc.SelectSingleNode(
          "/in_data/i_basho[1]").set_InnerText("56");
        // (6) 入力データの設定
        inDoc.SelectSingleNode(
          "/in_data/i_basho[2]").set_InnerText("43");
        // (6) 入力データの設定
        inDoc.SelectSingleNode(
          "/in_data/i_basho[3]").set_InnerText("18");
        // (7) GETDATA1ByXmlを呼び出す
        XmlDocument outDoc = server.GETDATA1ByXml(inDoc);
        // (8) 応答データの取り出し
        this.textBox1.set_Text(
        outDoc.SelectSingleNode("/out_data/o_name"
                                ).get_InnerText().Trim());
      } catch (TP1ConnectorException exp) {
        // Connector .NETが検知したエラー
      } catch (TP1Exception exp) {
        // その他スタブなどが検知したエラー
      } catch (System.Exception exp) {
        // 予期しない例外
      }
```

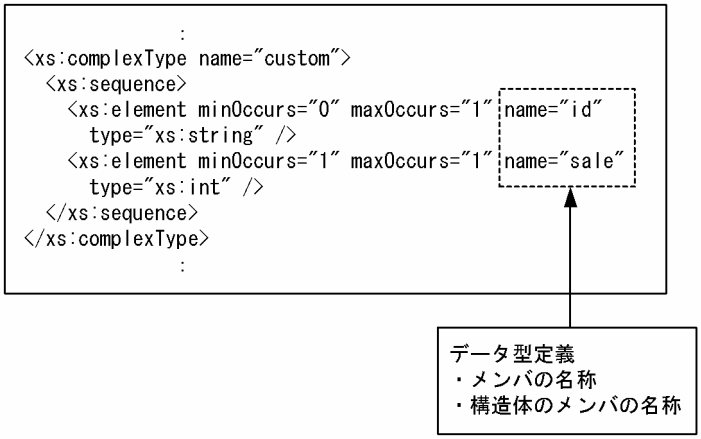
```
      finally
      {
        // (9) コネクションをコネクションプールに戻す
        if (tc != null) tc.Dispose();
      }
    }
}
```

(10) クライアントスタブの使用例（ASP.NET Web アプリケーション，Visual Basic，RPC データの XML マッピング機能を使用した場合）

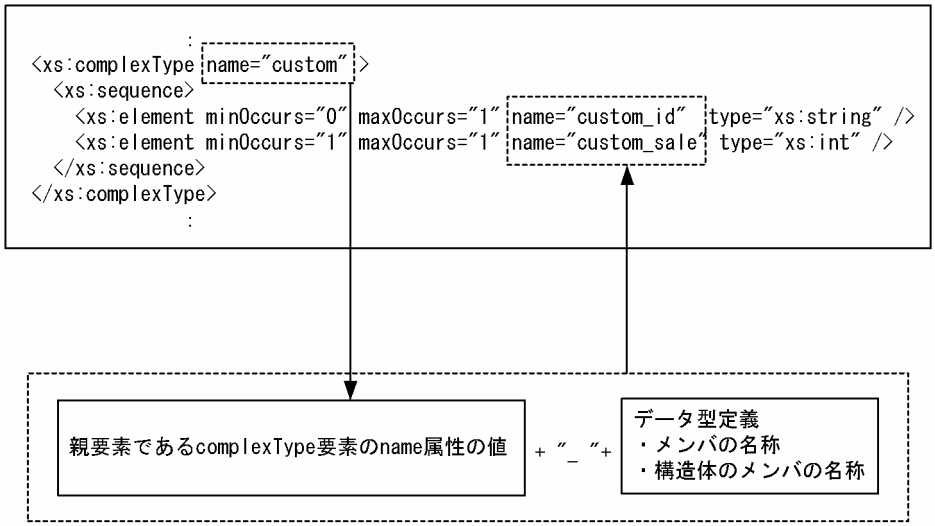
```
Imports System
Imports System.Xml
Imports Hitachi.OpenTP1
Imports Hitachi.OpenTP1.Connector

Namespace MyCompany
  Public Class MyForm1
      Inherits System.Web.UI.Page
    ...
    Private Sub Button1_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles Button1.Click
      Dim tcm As TP1ConnectionManager
      Dim tc As TP1Connection
      Dim ret As Integer
      Dim ids(2) As String
      Dim inDoc, outDoc  As XmlDocument
      ' このボタンがクリックされたら
      ' OpenTP1<TP1Host1>のサービス要求を行う
      Try
      ' グローバル変数のアプリケーション状態から
      ' TP1ConnectionManagerを取得
        ' (1) TP1ConnectionManagerクラスの生成
        tcm = CType(Application("tcmTP1Host1"), _
                    TP1ConnectionManager)
        ' (2) TP1Connectionオブジェクトの取得
        tc = tcm.GetConnection()
        ' (3) クライアントスタブの生成
        Dim server As GYOUMU1Stub = _
          New GYOUMU1Stub(tc, "SVGRP1")
        ' (4) RPC呼び出し形態
        server.Flags = RpcInfo.DCNOFLAGS
        ' (4) 最大応答待ち時間
        server.WatchTime = 180
        ' (5) 入力用カスタムレコードの生成
        inDoc = GetInputData()
        ' (6) 入力データの設定
        inDoc.SelectSingleNode( _
          "/in_data/i_basho[1]").InnerText = "56"
        ' (6) 入力データの設定
        inDoc.SelectSingleNode( _
          "/in_data/i_basho[2]").InnerText = "43"
        ' (6) 入力データの設定
        inDoc.SelectSingleNode( _
          "/in_data/i_basho[3]").InnerText = "18"
```

```
        ’(7) GETDATA1ByXmlを呼び出す
        outDoc = server.GETDATA1ByXml(inDoc)
        ’(8) 応答データの取り出し
        textBox1.Text = _
          outDoc.SelectSingleNode(”/out_data/o_name” _
                                  ).InnerText.Trim()
        Catch exp As TP1ConnectorException
        ’Connector .NETが検知したエラー
        Catch exp As TP1Exception
        ’その他スタブなどが検知したエラー
        Catch exp As Exception
        ’予期しない例外
      Finally
        If Not(tc Is Nothing) Then
        ’(9) コネクションをコネクションプールに戻す
        tc.Dispose()
      End If
      End Try
    End Sub
  End Class
End Namespace
```

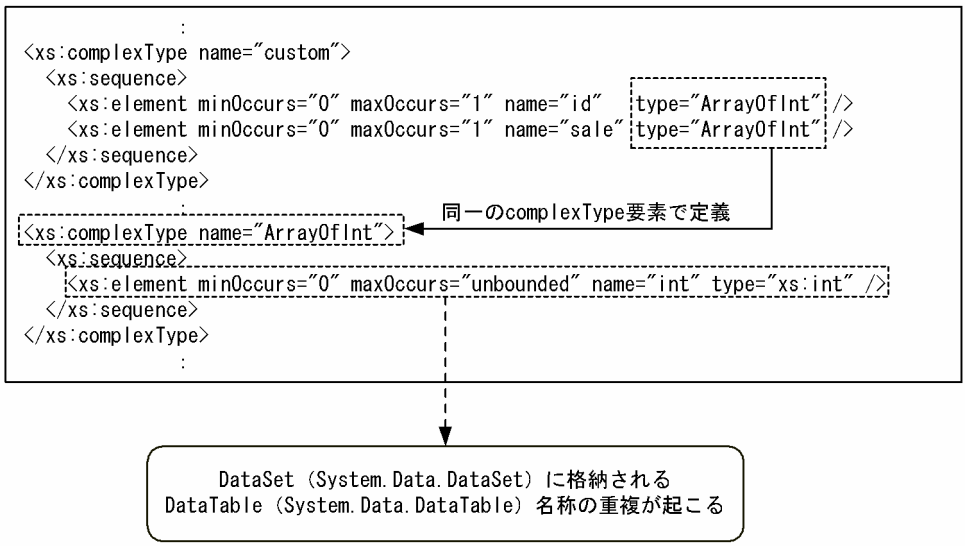
(11) XML スキーマ例（クライアントスタブ生成コマンド（spp2cstub）の-X オプションに normal を指定した場合 1）



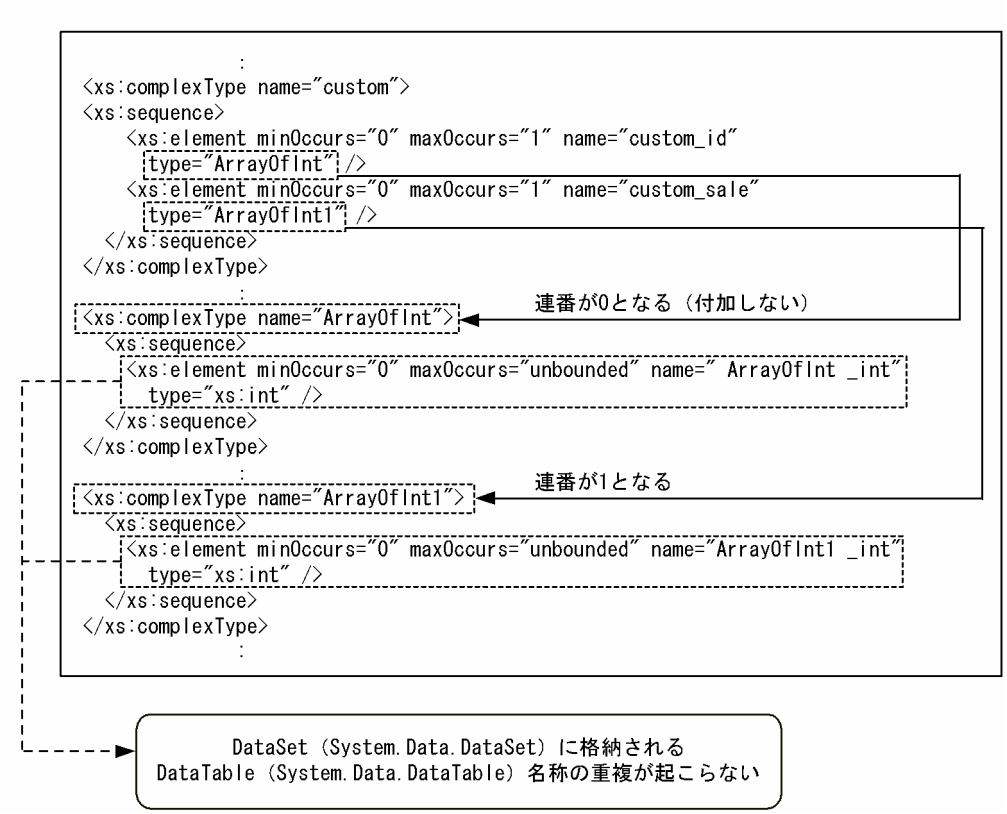
(12) XML スキーマ例（クライアントスタブ生成コマンド（spp2cstub）の-X オプションに dataset を指定した場合 1）



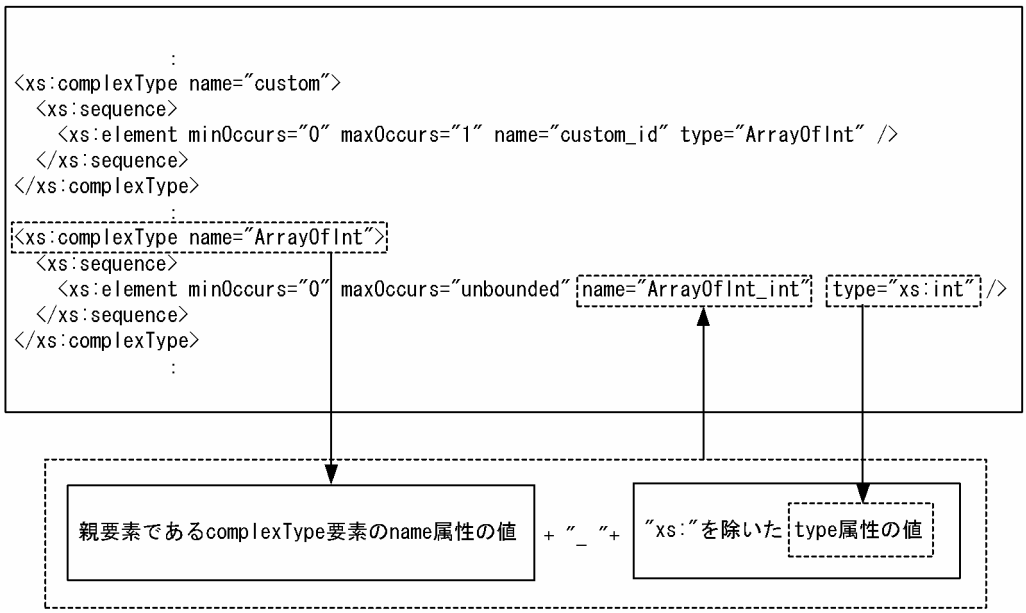
(13) XML スキーマ例（クライアントスタブ生成コマンド（spp2cstub）の-X オプションに normal を指定した場合 2）



(14) XML スキーマ例（クライアントスタブ生成コマンド（spp2cstub）の-X オプションに dataset を指定した場合 2）



(15) XML スキーマ例（クライアントスタブ生成コマンド（spp2cstub）の-X オプションに dataset を指定した場合 3）



4.4　インデクスドレコードを使用した SPP.NET または SPP の呼び出し方法

Connector .NET のアプリケーションからインデクスドレコードを使用して SPP.NET または SPP を呼び出す方法について説明します。なお，次のプログラム言語が使用できます。

- C#
- J#※
- Visual Basic
- COBOL 言語

注※

Visual Studio 2013 以降では，J#を使用して開発できません。

4.4.1　インデクスドレコードを使用する場合のメソッドの発行手順

インデクスドレコードを使用して RPC 要求をする場合のメソッド発行手順を次に示します。

1. TP1ConnectionManager クラスのインスタンスを生成または取り出します。
   プロパティを特定のプロファイルから読み込む場合は，コンストラクタの引数にプロファイル ID を指定します。インスタンスの生成はアプリケーションごとに行うことを推奨します。
2. TP1ConnectionManager クラスの GetConnection メソッドで TP1Connection オブジェクトを取得します。
   コネクションプールに使用できるコネクションがあった場合は，この時点でコネクションプールからコネクションを取り出した状態になります。
3. RpcInfo クラスのインスタンスを生成します。
4. RpcInfo オブジェクトにサービス名，サービスグループ名，タイムアウト値，RPC 形態フラグなどを設定します。
5. TP1ConnectionManager クラスの CreateIndexedRecord メソッドを発行して入力用インデクスドレコードを生成します。
6. 入力用インデクスドレコードに入力データを設定します。
7. TP1ConnectionManager クラスの CreateIndexedRecord メソッドを発行して出力用インデクスドレコードを生成します。
8. 出力データ用領域を確保して，出力用インデクスドレコードに設定します。
9. TP1Connection クラスの Execute メソッドを発行して RPC 要求をします（何度でも発行できます）。
   再度 RPC 要求をする場合は，4.～9.のどれかから行います。

各オブジェクトは複数回の RPC 要求で再利用できます。

10. 出力用インデクスドレコードからデータを取り出します。

11. TP1Connection クラスの Dispose メソッドを発行して，コネクションをコネクションプールに戻します。

注意事項

ASP.NET Web アプリケーションで使用する場合，TP1Connection クラスの Dispose メソッドは，次の時点で必ず発行してください。

- 各 UI イベントハンドラの終了時
- Web サービスメソッドの終了時

Dispose メソッドを適切に発行しないと，メモリリークの原因となります。

また，TP1ConnectionManager クラスは ASP.NET ではアプリケーション状態，またはアプリケーションインスタンスに保持することを推奨します。各 UI イベントハンドラや Web サービスメソッドごとに生成と削除を繰り返した場合，性能劣化やメモリ使用効率低下の原因となります。

4.4.2 インデクスドレコードを使用して RPC 要求をする場合のコーディング例

インデクスドレコードを使用して RPC 要求をする場合の，ASP.NET Web アプリケーションでの使用例を次に示します。

コメント中の(1)，(2)などは「4.4.1 インデクスドレコードを使用する場合のメソッドの発行手順」の説明の番号に対応しています。

(1) ASP.NET での初期化／終了コードの例 1（グローバル変数使用，Global.asax）

```
<script language="C#" runat="server">
     public void Application_OnStart() {
       this.Application["tcmServerA"] =
            new TP1ConnectionManager("ServerA");
     }
     public void Application_OnEnd() {
       this.Application["tcmServerA"] = null;
     }
</script>
```

(2) ASP.NET での初期化／終了コードの例 2（グローバル変数使用，Global.asax.cs）

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Web;
using System.Web.SessionState;

namespace webapp2
{
  public class Global : System.Web.HttpApplication
  {
    ...
    protected void Application_Start(Object sender, EventArgs e)
    {
      this.Application["tcmServerA"] =
           new TP1ConnectionManager("ServerA");
    }
    protected void Application_End(Object sender, EventArgs e)
    {
      this.Application["tcmServerA"] = null;
    }
  }
}
```

(3) ASP.NET での初期化／終了コードの例 3（ローカル変数使用，Global.asax）

```
<script language="C#" runat="server">
   protected TP1ConnectionManager tcm;
   public override void Init() {
      this.tcm = new TP1ConnectionManager("ServerA");
   }
   public override void Dispose() {
      this.tcm = null;
   }
</script>
```

(4) ASP.NET での初期化／終了コードの例 4（ローカル変数使用，Global.asax.cs）

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Web;
using System.Web.SessionState;

namespace webapp2
{
  public class Global : System.Web.HttpApplication
```

```
  {
    ...
    protected TP1ConnectionManager tcm;
    public override void Init() {
      this.tcm = new TP1ConnectionManager("ServerA");
    }
    public override void Dispose() {
      this.tcm = null;
    }
  }
}
```

(5) インデクスドレコードを使用して RPC 要求をする場合のコーディング例 (C#の場合)

```
using System;
using Hitachi.OpenTP1;
using Hitachi.OpenTP1.Connector;
using MyCompany;

namespace MyCompany
{
  public class MyForm1 : System.Web.UI.Page
  {
    ...
    private void Button1_Click(object sender, System.EventArgs e)
    {
      // このボタンがクリックされたら
      //  OpenTP1<ServerA>のサービス要求を行う
      // グローバル変数のアプリケーション状態から
      //  TP1ConnectionManagerを取得する
      // (1) TP1ConnectionManagerクラスの生成
      TP1ConnectionManager tcm =
         (TP1ConnectionManager)this.Application["tcmServerA"];
      // (2) TP1Connectionオブジェクトの取得
      TP1Connection tc = tcm.GetConnection();
      try {
        Encoding enc = Encoding.Default;
        // (3) RpcInfoクラスの生成
        RpcInfo ixSpec = new RpcInfo();
        // (4) サービスグループ名
        ixSpec.ServiceGroupName = "SVGRP1";
        // (4) サービス名
        ixSpec.ServiceName = "SERV1";
        // (4) RPC呼び出し形態
        ixSpec.Flags = RpcInfo.DCNOFLAGS;
        // (4) 最大応答待ち時間
        ixSpec.WatchTime = 180;
        // (5) 入力用インデクスドレコードの生成
        IndexedRecord input =
                      tcm.CreateIndexedRecord("in_record");
        // (6) 入力データの設定
        string indata = "Say Hello to OpenTP1!";
        // (6) 入力データの設定
        input.Add(enc.GetBytes(indata));
```

```
        // (7) 出力用インデクスドレコードの生成
        IndexedRecord output =
                    tcm.CreateIndexedRecord("out_record");
        // (8) 出力データ用領域の設定
        output.Add(new byte[this.textBox1.MaxLength*2]);
        // (9) RPCの実行
        bool ret = tc.Execute(ixSpec, input, output);
        // (10) データの取り出し
        byte[] outBuf = (byte[])output[0];
        // (10) データの取り出し
        this.textBox1.Text = enc.GetString(outBuf);
      } catch (TP1ConnectorException exp) {
        // Connector .NETが検知したエラー
      } catch (TP1Exception exp) {
        // Connector .NET (OpenTP1共通クラス) が検知したエラー
      } catch (Exception exp) {
        // 予期しない例外
      }
      finally
      {
        // (11) コネクションをコネクションプールに戻す
        if (tc != null) tc.Dispose();
      }
    }
  }
}
```

(6) インデクスドレコードを使用して RPC 要求をする場合のコーディング例 (COBOL 言語の場合)

```
>>PROPAGATE OFF
>>WEBSERVICE
>>WEBSERVICE-NAME 'MyWebService1'
>>WEBSERVICE-DESCRIPTION 'Sample of OpenTP1'
>>WEBSERVICE-NAMESPACE 'http://tempuri.org/'
>>END-WEBSERVICE
IDENTIFICATION DIVISION.
CLASS-ID. MyWebService1 AS 'MyCompany.MyWebService1'
 IS PUBLIC INHERITS WEBSERVICE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS WEBSERVICE AS 'System.Web.Services.WebService'.
  CLASS SYSTEMEXCEPTION AS 'System.Exception'.
  CLASS TP1EXCEPTION AS
     'Hitachi.OpenTP1.TP1Exception'.
  CLASS TP1CONNECTIONMANAGER AS
     'Hitachi.OpenTP1.Connector.TP1ConnectionManager'.
  CLASS TP1CONNECTION AS
     'Hitachi.OpenTP1.Connector.TP1Connection'.
  CLASS INDEXEDRECORD AS
     'Hitachi.OpenTP1.Connector.IndexedRecord'.
  CLASS RPCINFO AS 'Hitachi.OpenTP1.Connector.RpcInfo'.
  CLASS BYTE-ARRAY  AS 'System.Byte' IS ARRAY.
  CLASS OBJECT-CLS  AS 'System.Object'.
```

```
  CLASS CONVERT     AS 'System.Convert'.
  PROPERTY SERVICEGROUPNAME AS 'ServiceGroupName'.
  PROPERTY SERVICENAME      AS 'ServiceName'.
  PROPERTY FLAGS            AS 'Flags'.
  PROPERTY WATCHTIME        AS 'WatchTime'.
IDENTIFICATION DIVISION.
OBJECT.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  SERVICE-GROUP        PIC X(6) VALUE 'SVGRP1'.
01  SERVICE-NAME         PIC X(5) VALUE 'SERV1'.
PROCEDURE DIVISION.
>>WEBMETHOD
>>WEBMETHOD-DESCRIPTION 'SERVICE1'
>>WEBMETHOD-ENABLESESSION FALSE
>>WEBMETHOD-CACHEDURATION 0
>>WEBMETHOD-BUFFERRESPONSE TRUE
>>END-WEBMETHOD
IDENTIFICATION DIVISION.
METHOD-ID. SERVICE1 IS PUBLIC.
ENVIRONMENT DIVISION.
DATA DIVISION.
LOCAL-STORAGE SECTION.
01  IN-REC           PIC X(20) VALUE SPACE.
01  IN-REC-LEN       PIC S9(9) COMP-5 VALUE 20.
01  OUTDATA          PIC X(20) VALUE SPACE.
01  OUT-REC-LEN      PIC S9(9) COMP-5 VALUE 20.
01  TCNMGR    USAGE OBJECT REFERENCE TP1CONNECTIONMANAGER.
01  TCX       USAGE OBJECT REFERENCE TP1CONNECTION.
01  INRECORD  USAGE OBJECT REFERENCE INDEXEDRECORD.
01  OUTRECORD USAGE OBJECT REFERENCE INDEXEDRECORD.
01  RIF       USAGE OBJECT REFERENCE RPCINFO.
01  OBJECT-OBJ OBJECT REFERENCE OBJECT-CLS.
01  SERVICE-GROUP-STR    USAGE STRING.
01  SERVICE-NAME-STR     USAGE STRING.
01  IN-RECORD-NAME-STR   USAGE STRING.
01  OUT-RECORD-NAME-STR  USAGE STRING.
01  IN-RECORD-NAME       PIC X(10) VALUE 'in_record'.
01  OUT-RECORD-NAME      PIC X(10) VALUE 'out_record'.
01  INDATA-ARRAY   OBJECT REFERENCE BYTE-ARRAY.
01  OUTDATA-ARRAY  OBJECT REFERENCE BYTE-ARRAY.
01  RPC-FLAGS      USAGE BINARY-LONG VALUE 0.
01  WATCH-TIME     USAGE BINARY-LONG VALUE -1.
01  RST-INDEX      USAGE BINARY-LONG VALUE 0.
01  RESULT         USAGE LOGICAL.
01  OUTDATA-LEN    PIC S9(9) COMP-5.
01  TMP-LEN        USAGE BINARY-LONG.
01  OUTLEN-INDEX   USAGE BINARY-LONG VALUE 1.
LINKAGE SECTION.
01  SENDMSG  USAGE STRING.
01  RECVMSG  USAGE STRING.
PROCEDURE DIVISION USING BY VALUE SENDMSG RETURNING RECVMSG.
DECLARATIVES.
TP1-EXCEPTION SECTION.
  USE AFTER EO TP1EXCEPTION.
* Connector .NET（OpenTP1共通クラス）が検知したエラー
  RESUME METHOD-END.
```

```
ERROR-HANDLER1 SECTION.
  USE AFTER EO SYSTEMEXCEPTION.
*  予期しない例外
  RESUME METHOD-END.
END DECLARATIVES.
MAIN-PROC.
*    TP1ConnectionManagerのインスタンスを生成
     INVOKE TP1CONNECTIONMANAGER 'New' RETURNING TCNMGR.
*    TP1ConnectionManager.GetConnection()でTP1Connectionを取得します
     INVOKE TCNMGR 'GetConnection' RETURNING TCX.
*    TP1ConnectionManager.CreateIndexedRecord()を発行して
*    入力用インデクスドレコードを生成します
     SET IN-RECORD-NAME-STR TO
        FUNCTION ALPHANUMERIC-TO-STRING(IN-RECORD-NAME).
     INVOKE TCNMGR 'CreateIndexedRecord'
        USING BY VALUE IN-RECORD-NAME-STR
        RETURNING INRECORD.
*    TP1ConnectionManager.CreateIndexedRecord()を発行して
*    出力用インデクスドレコードを生成します
     SET OUT-RECORD-NAME-STR TO
        FUNCTION ALPHANUMERIC-TO-STRING(OUT-RECORD-NAME).
     INVOKE TCNMGR 'CreateIndexedRecord'
        USING BY VALUE OUT-RECORD-NAME-STR
        RETURNING OUTRECORD.
*    入力用インデクスドレコードに入力データを設定します
     INVOKE BYTE-ARRAY 'New' USING BY VALUE IN-REC-LEN
                            RETURNING INDATA-ARRAY.
     MOVE FUNCTION STRING-TO-ALPHANUMERIC(SENDMSG)
      TO IN-REC.
     CALL 'CBLXTOBYTEARRAY' USING IN-REC INDATA-ARRAY
                                  IN-REC-LEN.
     INVOKE INRECORD 'Add' USING BY VALUE
                           INDATA-ARRAY AS OBJECT-CLS
                         RETURNING RST-INDEX.
*    出力データ用領域を確保して出力用インデクスドレコードに設定します
     INVOKE BYTE-ARRAY 'New' USING BY VALUE OUT-REC-LEN
                     RETURNING OUTDATA-ARRAY.
     INVOKE OUTRECORD 'Add' USING BY VALUE
                            OUTDATA-ARRAY AS OBJECT-CLS
                          RETURNING RST-INDEX.
*    RpcInfoのインスタンスを生成します
     INVOKE RPCINFO 'New' RETURNING RIF.
*    RpcInfoオブジェクトにサービスグループ名を設定します
     SET SERVICE-GROUP-STR TO
        FUNCTION ALPHANUMERIC-TO-STRING(SERVICE-GROUP).
     SET SERVICEGROUPNAME OF RIF TO SERVICE-GROUP-STR.
*    RpcInfoオブジェクトにサービス名を設定します
     SET SERVICE-NAME-STR TO
        FUNCTION ALPHANUMERIC-TO-STRING(SERVICE-NAME).
     SET SERVICENAME OF RIF TO SERVICE-NAME-STR.
*    RpcInfoオブジェクトにRPC呼び出し形態を設定します
     MOVE RPC-FLAGS TO FLAGS OF RIF.
*    RpcInfoオブジェクトに最大応答待ち時間を設定します
     MOVE WATCH-TIME TO WATCHTIME OF RIF.
*    TP1Connection.Execute()を発行してRPC要求を行います
     INVOKE TCX 'Execute' USING BY VALUE RIF
                           BY VALUE INRECORD
                           BY VALUE OUTRECORD
```

```
                      RETURNING RESULT.
*    出力用インデクスドレコードからデータを取り出します
     INVOKE OUTRECORD 'get_Item' USING BY VALUE OUTLEN-INDEX
                            RETURNING OBJECT-OBJ.
     INVOKE CONVERT 'ToInt32' USING BY VALUE OBJECT-OBJ
                            RETURNING TMP-LEN.
     MOVE TMP-LEN TO OUTDATA-LEN.
     CALL 'CBLBYTEARRAYTOX' USING OUTDATA-ARRAY OUTDATA-LEN
                             OUTDATA.
     SET RECVMSG TO
        FUNCTION ALPHANUMERIC-TO-STRING(OUTDATA).
  METHOD-END.
*    TP1Connection.Dispose()を発行して，コネクションをコネクション
*    プールに戻します
     IF NOT TCX = NULL THEN
        INVOKE TCX 'Dispose'
     END-IF.
  PROG-END.
     EXIT METHOD.
  END METHOD SERVICE1.
*
  END OBJECT.
  END CLASS MyWebService1.
*
```

4.4.3 インデクスドレコードとバッファプーリング機能を使用する場合のメソッド発行手順

インデクスドレコードとバッファプーリング機能を使用して RPC 要求をする場合のメソッド発行手順を次に示します。

1. TP1ConnectionManager クラスのインスタンスを生成または取り出します。

   プロパティを特定のプロファイルから読み込む場合は，コンストラクタの引数にプロファイル ID を指定します。インスタンスの生成はアプリケーションごとに行うことを推奨します。

2. TP1ConnectionManager クラスの GetConnection メソッドで TP1Connection オブジェクトを取得します。

   コネクションプールに使用できるコネクションがあった場合は，この時点でコネクションプールからコネクションを取り出した状態になります。

3. RpcInfo クラスのインスタンスを生成します。

4. RpcInfo オブジェクトにサービス名，サービスグループ名，タイムアウト値，RPC 形態フラグなどを設定します。

5. TP1ConnectionManager クラスの CreateIndexedRecord メソッドを発行して，入力用インデクスドレコードを生成します。

6. TP1ConnectionManager クラスの GetMessageBuffer メソッドに入力データサイズを指定して発行して，入力データ用の MessageBuffer 実装オブジェクトを取得します。
   バッファプールに使用できるバッファがあった場合は，この時点でバッファプールからバッファを取り出した状態になります。

7. MessageBuffer 実装オブジェクトが保持しているバッファに入力データを設定します。

8. 入力用インデクスドレコードに入力データを設定したバッファを設定します。

9. TP1ConnectionManager クラスの CreateIndexedRecord メソッドを発行して，出力用インデクスドレコードを生成します。

10. TP1ConnectionManager クラスの GetMessageBuffer メソッドに出力データサイズを指定して発行し，出力データ用の MessageBuffer 実装オブジェクトを取得して，出力用インデクスドレコードに設定します。
    バッファプールに使用可能なバッファがあった場合は，この時点でバッファプールからバッファを取り出した状態になります。

11. TP1Connection クラスの Execute メソッドを発行して RPC 要求を行います（何度でも発行できます）。
    再度 RPC 要求を行う場合は，4.～11.のどれかから行います。
    各オブジェクトは複数回の RPC 要求で再利用できます。

12. 出力用 IndexedRecord からデータを取り出します。

13. 入力用 MessageBuffer 実装オブジェクトの ReleaseMessageBuffer メソッドを発行して，入力用メッセージバッファをバッファプールに戻します。

14. 出力用 MessageBuffer 実装オブジェクトの ReleaseMessageBuffer メソッドを発行して，出力用メッセージバッファをバッファプールに戻します。

15. TP1Connection クラスの Dispose メソッドを発行して，コネクションをコネクションプールに戻します。
    複数の TP1Connection オブジェクトを使う場合は，それぞれのインスタンスが不要になった時点で Dispose メソッドを発行します。

注意事項

ASP.NET Web アプリケーションで使用する場合，MessageBuffer クラスの ReleaseMessageBuffer メソッド，および TP1Connection クラスの Dispose メソッドを，次の時点で必ず発行してください。

- 各 UI イベントハンドラの終了時
- Web サービスメソッドの終了時

これらのメソッドを適切に発行しないと，メモリリークの原因となります。

また，TP1ConnectionManager クラスは，ASP.NET ではアプリケーション状態，またはアプリケーションインスタンスに保持することを推奨します。各 UI イベントハンドラや Web サービスメソッドごとに生成や削除を繰り返した場合，性能劣化やメモリ使用効率低下の原因となります。

4.4.4 インデクスドレコードとバッファプーリング機能を使用して RPC 要求をする場合のコーディング例

インデクスドレコードとバッファプーリング機能を使用して RPC 要求をする場合の，ASP.NET Web アプリケーションでの使用例を次に示します。

なお，コメント中の(1)，(2)などは「4.4.3 インデクスドレコードとバッファプーリング機能を使用する場合のメソッド発行手順」の説明の番号に対応しています。

(1) インデクスドレコードとバッファプーリング機能を使用して RPC 要求をする場合のコーディング例（C#の場合）

```
using System;
using System.Text;
using Hitachi.OpenTP1;
using Hitachi.OpenTP1.Connector;
using MyCompany;

namespace MyCompany
{
  public class MyForm1 : System.Web.UI.Page
  {
    …
    private void Button1_Click(object sender, System.EventArgs e)
    {
      TP1Connection tc = null;
      MessageBuffer inBuf = null;
      MessageBuffer outBuf = null;
      // このボタンがクリックされたら
      // OpenTP1<ServerA>のサービス要求を行う
      // グローバル変数のアプリケーション状態から
      // TP1ConnectionManagerを取得する
      // (1) TP1ConnectionManagerクラスの生成
      TP1ConnectionManager tcm =
         (TP1ConnectionManager)this.Application[”tcmServerA”];
      // (2) TP1Connectionオブジェクトの取得
      tc = tcm.GetConnection();
      try {
        Encoding enc = Encoding.Default;
        // (3) RpcInfoクラスの生成
        RpcInfo ixSpec = new RpcInfo();
        // (4) サービスグループ名
        ixSpec.ServiceGroupName = ”SVGRP1”;
        // (4) サービス名
        ixSpec.ServiceName = ”SERV1”;
```

```
        // (4) RPC呼び出し形態
        ixSpec.Flags = RpcInfo.DCNOFLAGS;
        // (4) 最大応答待ち時間
        ixSpec.WatchTime = 180;
        // (5) 入力用インデクスドレコードの生成
        IndexedRecord input =
                    tcm.CreateIndexedRecord("in_record");
        // (6)(7) 入力データ用メッセージバッファの取得,
        // および入力データの設定
        string indata = "Say Hello to OpenTP1!";
        // (6) 入力データ用メッセージバッファの取得
        int indataLen = enc.GetByteCount(indata);
        // (6) 入力データ用メッセージバッファの取得
        inBuf = tcm.GetMessageBuffer(indataLen);
        // (7) 入力データの設定
        inBuf.Append(enc.GetBytes(indata));
        // (8) 入力用インデクスドレコードの設定
        input.Add(inBuf);
        // (9) 出力用インデクスドレコードの生成
        IndexedRecord output =
                    tcm.CreateIndexedRecord("out_record");
        // (10) 出力データ用メッセージバッファの取得
        // および出力用インデクスドレコードの設定
        outBuf = tcm.GetMessageBuffer(this.textBox1.MaxLength*2);
        output.Add(outBuf);
        // (11) RPCの実行
        bool ret = tc.Execute(ixSpec, input, output);
        // (12) データの取り出し
        outBuf = (MessageBuffer)output[0];
        // (12) データの取り出し
        byte[] outdata = outBuf.Buffer;
        // (12) データの取り出し
        this.textBox1.Text = enc.GetString(outdata);
      } catch (TP1ConnectorException exp) {
        // Connector .NETが検知したエラー
      } catch (TP1Exception exp) {
        // Connector .NET (OpenTP1共通クラス) が検知したエラー
      } catch (Exception exp) {
        // 予期しない例外
      }
      finally
      {
        // (13) 入力用メッセージバッファをバッファプールに戻す
        if (inBuf != null) inBuf.ReleaseMessageBuffer();
        // (14) 出力用メッセージバッファをバッファプールに戻す
        if (outBuf != null) outBuf.ReleaseMessageBuffer();
        // (15) コネクションをコネクションプールに戻す
        if (tc != null) tc.Dispose();
       }
    }
  }
}
```

(2) インデクスドレコードとバッファプーリング機能を使用して RPC 要求をする場合のコーディング例（COBOL 言語の場合）

```
>>PROPAGATE OFF
>>WEBSERVICE
>>WEBSERVICE-NAME 'MyWebService1'
>>WEBSERVICE-DESCRIPTION 'Sample of OpenTP1'
>>WEBSERVICE-NAMESPACE 'http://tempuri.org/'
>>END-WEBSERVICE
IDENTIFICATION DIVISION.
CLASS-ID. MyWebService1 AS 'MyCompany.MyWebService1'
  IS PUBLIC INHERITS WEBSERVICE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS WEBSERVICE AS 'System.Web.Services.WebService'.
  CLASS SYSTEMEXCEPTION AS 'System.Exception'.
  CLASS TP1EXCEPTION AS
     'Hitachi.OpenTP1.TP1Exception'.
  CLASS TP1CONNECTIONMANAGER AS
     'Hitachi.OpenTP1.Connector.TP1ConnectionManager'.
  CLASS TP1CONNECTION AS
     'Hitachi.OpenTP1.Connector.TP1Connection'.
  CLASS INDEXEDRECORD AS
     'Hitachi.OpenTP1.Connector.IndexedRecord'.
  CLASS RPCINFO AS 'Hitachi.OpenTP1.Connector.RpcInfo'.
  CLASS MESSAGEBUFFER AS
     'Hitachi.OpenTP1.Connector.MessageBuffer'.
  CLASS BYTE-ARRAY  AS 'System.Byte' IS ARRAY.
  CLASS OBJECT-CLS  AS 'System.Object'.
  CLASS CONVERT     AS 'System.Convert'.
  PROPERTY SERVICEGROUPNAME AS 'ServiceGroupName'.
  PROPERTY SERVICENAME      AS 'ServiceName'.
  PROPERTY FLAGS            AS 'Flags'.
  PROPERTY WATCHTIME        AS 'WatchTime'.
  PROPERTY BUFFER           AS 'Buffer'.
  PROPERTY MESSAGELENGTH    AS 'MessageLength'.
IDENTIFICATION DIVISION.
OBJECT.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  SERVICE-GROUP        PIC X(6) VALUE 'SVGRP1'.
01  SERVICE-NAME         PIC X(5) VALUE 'SERV1'.
PROCEDURE DIVISION.
>>WEBMETHOD
>>WEBMETHOD-DESCRIPTION 'SERVICE1'
>>WEBMETHOD-ENABLESESSION FALSE
>>WEBMETHOD-CACHEDURATION 0
>>WEBMETHOD-BUFFERRESPONSE TRUE
>>END-WEBMETHOD
IDENTIFICATION DIVISION.
METHOD-ID. SERVICE1 IS PUBLIC.
ENVIRONMENT DIVISION.
DATA DIVISION.
LOCAL-STORAGE SECTION.
```

```
01  IN-REC           PIC X(20) VALUE SPACE.
01  IN-REC-LEN       PIC S9(9) COMP-5 VALUE 20.
01  OUTDATA          PIC X(20) VALUE SPACE.
01  OUT-REC-LEN      PIC S9(9) COMP-5 VALUE 20.
01  OUTDATA-LEN      PIC S9(9) COMP-5.
01  TCNMGR    USAGE OBJECT REFERENCE TP1CONNECTIONMANAGER.
01  TCX       USAGE OBJECT REFERENCE TP1CONNECTION.
01  INRECORD  USAGE OBJECT REFERENCE INDEXEDRECORD.
01  OUTRECORD USAGE OBJECT REFERENCE INDEXEDRECORD.
01  RIF       USAGE OBJECT REFERENCE RPCINFO.
01  INBUF     USAGE OBJECT REFERENCE MESSAGEBUFFER.
01  OUTBUF    USAGE OBJECT REFERENCE MESSAGEBUFFER.
01  OBJECT-OBJ OBJECT REFERENCE OBJECT-CLS.
01  SERVICE-GROUP-STR    USAGE STRING.
01  SERVICE-NAME-STR     USAGE STRING.
01  IN-RECORD-NAME-STR   USAGE STRING.
01  OUT-RECORD-NAME-STR  USAGE STRING.
01  IN-RECORD-NAME       PIC X(10) VALUE 'in_record'.
01  OUT-RECORD-NAME      PIC X(10) VALUE 'out_record'.
01  INDATA-ARRAY   OBJECT REFERENCE BYTE-ARRAY.
01  OUTDATA-ARRAY  OBJECT REFERENCE BYTE-ARRAY.
01  RPC-FLAGS      USAGE BINARY-LONG VALUE 0.
01  WATCH-TIME     USAGE BINARY-LONG VALUE -1.
01  RST-INDEX      USAGE BINARY-LONG VALUE 0.
01  RESULT         USAGE LOGICAL.
01  OUTBUF-INDEX   USAGE BINARY-LONG VALUE 0.
LINKAGE SECTION.
01  SENDMSG  USAGE STRING.
01  RECVMSG  USAGE STRING.
PROCEDURE DIVISION USING BY VALUE SENDMSG RETURNING RECVMSG.
DECLARATIVES.
TP1-EXCEPTION SECTION.
   USE AFTER EO TP1EXCEPTION.
*  Connector .NET（OpenTP1共通クラス）が検知したエラー
   RESUME METHOD-END.
ERROR-HANDLER1 SECTION.
   USE AFTER EO SYSTEMEXCEPTION.
*  予期しない例外
   RESUME METHOD-END.
 END DECLARATIVES.
 MAIN-PROC.
*    TP1ConnectionManagerのインスタンスを生成
     INVOKE TP1CONNECTIONMANAGER 'New' RETURNING TCNMGR.
*    TP1ConnectionManager.GetConnection()でTP1Connectionを取得します
     INVOKE TCNMGR 'GetConnection' RETURNING TCX.
*    RpcInfoのインスタンスを生成します
     INVOKE RPCINFO 'New' RETURNING RIF.
*    RpcInfoオブジェクトにサービスグループ名を設定します
     SET SERVICE-GROUP-STR TO
        FUNCTION ALPHANUMERIC-TO-STRING(SERVICE-GROUP).
     SET SERVICEGROUPNAME OF RIF TO SERVICE-GROUP-STR.
*    RpcInfoオブジェクトにサービス名を設定します
     SET SERVICE-NAME-STR TO
        FUNCTION ALPHANUMERIC-TO-STRING(SERVICE-NAME).
     SET SERVICENAME OF RIF TO SERVICE-NAME-STR.
*    RpcInfoオブジェクトにRPC呼び出し形態を設定します
     MOVE RPC-FLAGS TO FLAGS OF RIF.
*    RpcInfoオブジェクトに最大応答待ち時間を設定します
```

```
     MOVE WATCH-TIME TO WATCHTIME OF RIF.
*    TP1ConnectionManager.CreateIndexedRecord()を発行して
*    入力用インデクスドレコードを生成します
     SET IN-RECORD-NAME-STR TO
        FUNCTION ALPHANUMERIC-TO-STRING(IN-RECORD-NAME).
     INVOKE TCNMGR 'CreateIndexedRecord'
        USING BY VALUE IN-RECORD-NAME-STR
        RETURNING INRECORD.
*    TP1ConnectionManager.GetMessageBuffer（入力データサイズ）を
*    発行して入力データ用のMessageBuffer実装オブジェクトを取得します
     INVOKE TCNMGR 'GetMessageBuffer'
        USING BY VALUE IN-REC-LEN
        RETURNING INBUF.
     INVOKE BYTE-ARRAY 'New' USING BY VALUE IN-REC-LEN
                          RETURNING INDATA-ARRAY.
     MOVE FUNCTION STRING-TO-ALPHANUMERIC(SENDMSG)
      TO IN-REC.
     CALL 'CBLXTOBYTEARRAY' USING IN-REC INDATA-ARRAY
                                  IN-REC-LEN.
*    MessageBuffer実装オブジェクトが保持しているバッファに
*    入力データを設定します
     INVOKE INBUF 'Append' USING BY VALUE INDATA-ARRAY.
*    入力用インデクスドレコードに入力データを設定したバッファを設定します
     INVOKE INRECORD 'Add' USING BY VALUE INBUF
                        RETURNING RST-INDEX.
*    TP1ConnectionManager.CreateIndexedRecord () を発行して
*    出力用インデクスドレコードを生成します
     SET OUT-RECORD-NAME-STR TO
        FUNCTION ALPHANUMERIC-TO-STRING(OUT-RECORD-NAME).
     INVOKE TCNMGR 'CreateIndexedRecord'
        USING BY VALUE OUT-RECORD-NAME-STR
        RETURNING OUTRECORD.
*    TP1ConnectionManager.GetMessageBuffer（出力データサイズ）を
*    発行して出力データ用のMessageBuffer実装オブジェクトを取得し,
*    出力用インデクスドレコードに設定します
     INVOKE TCNMGR 'GetMessageBuffer'
         USING BY VALUE OUT-REC-LEN
         RETURNING OUTBUF.
     INVOKE OUTRECORD 'Add' USING BY VALUE OUTBUF
         RETURNING RST-INDEX.
*    TP1Connection.Execute()を発行してRPC要求を行います
     INVOKE TCX 'Execute' USING BY VALUE RIF
                            BY VALUE INRECORD
                            BY VALUE OUTRECORD
                       RETURNING RESULT.
*    出力用インデクスドレコードからデータを取り出します
     INVOKE OUTRECORD 'get_Item' USING BY VALUE OUTBUF-INDEX
                              RETURNING OBJECT-OBJ.
     SET OUTBUF TO OBJECT-OBJ AS MESSAGEBUFFER.
     SET OUTDATA-ARRAY TO BUFFER OF OUTBUF.
     MOVE MESSAGELENGTH OF OUTBUF TO OUTDATA-LEN.
     CALL 'CBLBYTEARRAYTOX' USING OUTDATA-ARRAY OUTDATA-LEN
                                  OUTDATA.
     SET RECVMSG TO FUNCTION ALPHANUMERIC-TO-STRING(OUTDATA).
 METHOD-END.
*    入力用MessageBuffer実装オブジェクトのReleaseMessageBuffer()を
*    発行して入力用メッセージバッファをバッファプールに戻します
     IF NOT INBUF = NULL THEN
```

```
        INVOKE INBUF 'ReleaseMessageBuffer'
     END-IF.
*    出力用MessageBuffer実装オブジェクトのReleaseMessageBuffer()を
*    発行して出力用MessageBufferをバッファプールに戻します
     IF NOT OUTBUF = NULL THEN
        INVOKE OUTBUF 'ReleaseMessageBuffer'
     END-IF.
*    TP1Connection.Dispose()を発行して，コネクションをコネクション
*    プールに戻します
     IF NOT TCX = NULL THEN
        INVOKE TCX 'Dispose'
     END-IF.
  PROG-END.
     EXIT METHOD.
 END METHOD SERVICE1.
*
 END OBJECT.
 END CLASS MyWebService1.
*
```

4.5　トランザクション制御機能の使用方法

OpenTP1 のトランザクション制御機能を使用する場合は，TP1Connection オブジェクトを使用します。RPC インタフェースに次のどのインタフェースを使用した場合でも使用できます。

- .NET インタフェース定義を使用した場合
- サービス定義を使用した場合
- インデクスドレコードを使用した場合

次の条件でトランザクション制御機能を使用した場合のコーディング例を示します。

- サービスグループ名：GRP1
- インタフェース名：MyCompany.IGyoumuB
- 呼び出すサービスメソッド名（サービス名）：
  PutData (DCRPC_CHAINED)
  Invoke (DCNOFLAGS)
- 構成ファイル：指定あり（プロファイル ID="TP1Host1"）
- クライアントスタブ：.NET インタフェース定義から生成
- アプリケーション形態：ASP.NET XML Web サービス
- RPC 形態：連鎖 RPC

.NET インタフェース定義の定義例（C#の場合）

```
using System;

namespace MyCompany
{
  public interface IGyoumuB
  {
    void PutData(string clientId, string[] key, string[] data);
    int Invoke(string clientId, ref string result);
  }
}
```

連鎖 RPC でトランザクション機能を使用する場合のコーディング例（ASP.NET XML Web サービス，C#の場合）

```
using System;
using Hitachi.OpenTP1;
using Hitachi.OpenTP1.Connector;
using MyCompany;

namespace MyCompany
{
  [WebService]
  public class MyWebService1
  {
```

```
...
[WebMethod]
private string Service(string[] key, string[] data)
{
  string clientId = "MyWS_Service_" + seqNo.ToString();
  string result = "";
  TP1Connection tc = null;
  bool trnFlag = false;
  try {
  // グローバル変数のアプリケーション状態から
  // TP1ConnectionManagerを取得
    TP1ConnectionManager tcm =
      (TP1ConnectionManager)this.Application["tcmTP1Host1"];
    tc = tcm.GetConnection();
    IGyoumuBStub server = new IGyoumuBStub(tc, "GRP1");
    // トランザクション開始
    tc.Begin();
    trnFlag = true;
    // 連鎖RPC
    server.Flags = RpcInfo.DCRPC_CHAINED;
    // 最大応答待ち時間(60秒)
    server.WatchTime = 60;
    // PutDataの呼び出し(サーバにデータの書き込み)
    server.PutData(clientId, key, data);
    // 連鎖RPCの終了
    server.Flags = RpcInfo.DCNOFLAGS;
    // 最大応答待ち時間(180秒)
    server.WatchTime = 180;
    // Invokeの呼び出し(処理要求)
    server.Invoke(clientId, ref result);
    // トランザクション終了
    tc.Commit();
    trnFlag = false;
    return result;
  } catch (TP1UserException exp) {
    // Service3()からユーザ例外がスローされた
  } catch (TP1RemoteException exp) {
    // Service3()で予期しない例外発生
  } catch (TP1ConnectorException exp) {
    // Connector .NETが検知したエラー
  } catch (TP1Exception exp) {
    // その他スタブなどが検知したエラー
  } catch (Exception exp) {
    // 予期しない例外
  }
  finally
  {
    try {
      // ロールバック
      if (trnFlag) tc.Rollback();
    }
    catch (Exception) {
      // 無視
    }
    // コネクションをコネクションプールに戻す
    if (tc != null) tc.Dispose();
  }
}
```

```
    }
}
```

4.6 TCP/IP 通信機能の使用方法

Connector .NET のアプリケーションから TCP/IP 通信機能を使用する方法について説明します。

4.6.1 TCP/IP 通信機能を使用する場合のメソッド発行手順

TCP/IP 通信機能を使用して，OpenTP1 の MHP や他システムと通信する場合のメソッド発行手順を次に示します。なお，TCP/IP 通信にはインデクスドレコードを使用します。

1. TP1ConnectionManager クラスのインスタンスを生成または取り出します。
   構成定義を特定のプロファイルから読み込む場合は，コンストラクタの引数にプロファイル ID を指定します。インスタンスの生成はアプリケーションごとに行うことを推奨します。TCP/IP 通信と RPC とを併用する場合は，RPC で使用するプロファイル ID とは異なるプロファイル ID で構成定義を設定することを推奨します。

2. TP1ConnectionManager クラスの GetTcpipConnection メソッドで TcpipConnection オブジェクトを取得します。
   コネクションプールに使用できるコネクションがあった場合は，この時点でコネクションプールからコネクションを取り出した状態になります。

3. TcpipInfo クラスのインスタンスを生成します。

4. TcpipInfo オブジェクトにタイムアウト値，通信形態を設定します。

5. TP1ConnectionManager クラスの CreateIndexedRecord メソッドを発行して入力用インデクスドレコードを生成します。

6. 入力用インデクスドレコードに入力データを設定します。

7. TP1ConnectionManager クラスの CreateIndexedRecord メソッドを発行して出力用インデクスドレコードを生成します。

8. 出力データ用領域を確保して，出力用インデクスドレコードに設定します。

9. TcpipConnection クラスの Execute メソッドを発行して TCP/IP 通信をします（何度でも発行できます）。
   再度 TCP/IP 通信をする場合は，4.～9.のどれかから行います。
   各オブジェクトは複数回の TCP/IP 通信で再利用できます。

10. 出力用インデクスドレコードからデータを取り出します。

11. TcpipConnection クラスの Dispose メソッドを発行して，コネクションをコネクションプールに戻します。

注意事項

ASP.NET Web アプリケーションで使用する場合，TcpipConnection クラスの Dispose メソッドは，次の時点で必ず発行してください。

- 各 UI イベントハンドラの終了時
- Web サービスメソッドの終了時

Dispose メソッドを適切に発行しないと，メモリリークの原因となります。

また，TP1ConnectionManager クラスは ASP.NET ではアプリケーション状態，またはアプリケーションインスタンスに保持することを推奨します。各 UI イベントハンドラや Web サービスメソッドごとに生成と削除を繰り返した場合，性能劣化やメモリ使用効率低下の原因となります。

4.6.2 TCP/IP 通信機能を使用する場合のコーディング例（C#の場合）

TCP/IP 通信機能を使用する場合の，ASP.NET Web アプリケーションでの使用例を次に示します。

コメント中の(1)，(2)などは「4.6.1 TCP/IP 通信機能を使用する場合のメソッド発行手順」の説明の番号に対応しています。

```
using System;
using Hitachi.OpenTP1;
using Hitachi.OpenTP1.Connector;
using MyCompany;

namespace MyCompany
{
  public class MyForm1 : System.Web.UI.Page
  {
    ...
    private void Button1_Click(object sender, System.EventArgs e)
    {
      // このボタンがクリックされたら
      //  OpenTP1<ServerA>のサービス要求を行う
      // グローバル変数のアプリケーション状態から
      //  TP1ConnectionManagerを取得する
      // (1) TP1ConnectionManagerクラスの生成
      TP1ConnectionManager tcm =
         (TP1ConnectionManager)this.Application["tcmServerA"];
      // (2) TcpipConnectionオブジェクトの取得
      tc = tcm.GetTcpipConnection();
      try {
        Encoding enc = Encoding.Default;
        // (3) TcpipInfoクラスの生成
        TcpipInfo info = new TcpipInfo();
        // (4) 通信形態
        info.Flags = TcpipInfo.TCPIP_SENDRECV;
        // (4) 最大応答待ち時間
```

```
        info.WatchTime = 180;
        // (5) 入力用インデクスドレコードの生成
        IndexedRecord input =
                    tcm.CreateIndexedRecord("in_record");
        // (6) 入力データの設定
        string indata = "Say Hello to OpenTP1!";
        // (6) 入力データの設定
        input.Add(enc.GetBytes(indata));
        // (7) 出力用インデクスドレコードの生成
        IndexedRecord output =
                    tcm.CreateIndexedRecord("out_record");
        // (8) 出力データ用領域の設定
        output.Add(new byte[this.textBox1.MaxLength*2]);
        // (9) TCP/IP通信の実行
        bool ret = tc.Execute(info, input, output);
        // (10) データの取り出し
        byte[] outBuf = (byte[])output[0];
        // (10) データの取り出し
        this.textBox1.Text = enc.GetString(outBuf);
      } catch (TP1ConnectorException exp) {
        // Connector .NETが検知したエラー
      } catch (TP1Exception exp) {
        // Connector .NET(OpenTP1共通クラス)が検知したエラー
      } catch (Exception exp) {
        // 予期しない例外
      }
      finally
      {
        // (11) コネクションをコネクションプールに戻す
        if (tc != null) tc.Dispose();
      }
    }
  }
}
```

4.6.3 TCP/IP 通信機能とバッファプーリング機能を使用する場合のメソッド発行手順

TCP/IP 通信機能とバッファプーリング機能を使用する場合のメソッド発行手順を次に示します。

1. TP1ConnectionManager クラスのインスタンスを生成または取り出します。

   構成定義を特定のプロファイルから読み込む場合は，コンストラクタの引数にプロファイル ID を指定します。インスタンスの生成はアプリケーションごとに行うことを推奨します。TCP/IP 通信と RPC とを併用する場合は，RPC で使用するプロファイル ID とは異なるプロファイル ID で構成定義を設定することを推奨します。

2. TP1ConnectionManager クラスの GetTcpipConnection メソッドで TcpipConnection オブジェクトを取得します。

   コネクションプールに使用できるコネクションがあった場合は，この時点でコネクションプールからコネクションを取り出した状態になります。

3. TcpipInfo クラスのインスタンスを生成します。

4. TcpipInfo オブジェクトにタイムアウト値，通信形態を設定します。

5. TP1ConnectionManager クラスの CreateIndexedRecord メソッドを発行して入力用インデクスドレコードを生成します。

6. TP1ConnectionManager クラスの GetMessageBuffer メソッドに入力データサイズを指定して発行して，入力データ用の MessageBuffer 実装オブジェクトを取得します。

   バッファプールに使用できるバッファがあった場合は，この時点でバッファプールからバッファを取り出した状態になります。

7. MessageBuffer 実装オブジェクトが保持しているバッファに入力データを設定します。

8. 入力用インデクスドレコードに入力データを設定したバッファを設定します。

9. TP1ConnectionManager クラスの CreateIndexedRecord メソッドを発行して，出力用インデクスドレコードを生成します。

10. TP1ConnectionManager クラスの GetMessageBuffer メソッドに出力データサイズを指定して発行し，出力データ用の MessageBuffer 実装オブジェクトを取得して，出力用インデクスドレコードに設定します。

    バッファプールに使用可能なバッファがあった場合は，この時点でバッファプールからバッファを取り出した状態になります。また，TCP/IP 通信では，出力データ用の MessageBuffer 実装オブジェクトの MessageLength にあらかじめ受信データサイズを設定しておきます。

11. TcpipConnection クラスの Execute メソッドを発行して TCP/IP 通信を行います（何度でも発行できます）。

    再度 TCP/IP 通信を行う場合は，4.～11.のどれかから行います。

    各オブジェクトは複数回の TCP/IP 通信で再利用できます。

12. 出力用インデクスドレコードからデータを取り出します。

13. 入力用 MessageBuffer 実装オブジェクトの ReleaseMessageBuffer メソッドを発行して，入力用メッセージバッファをバッファプールに戻します。

14. 出力用 MessageBuffer 実装オブジェクトの ReleaseMessageBuffer メソッドを発行して，出力用メッセージバッファをバッファプールに戻します。

15. TcpipConnection クラスの Dispose メソッドを発行して，コネクションをコネクションプールに戻します。

    複数の TcpipConnection オブジェクトを使う場合は，それぞれのインスタンスが不要になった時点で Dispose メソッドを発行します。

注 1

ASP.NET Web アプリケーションで使用する場合，MessageBuffer クラスの ReleaseMessageBuffer メソッド，および TcpipConnection クラスの Dispose メソッドを，次の時点で必ず発行してください。

- 各 UI イベントハンドラの終了時
- Web サービスメソッドの終了時

これらのメソッドを適切に発行しないと，メモリリークの原因となります。

また，TP1ConnectionManager クラスは，ASP.NET ではアプリケーション状態，またはアプリケーションインスタンスに保持することを推奨します。各 UI イベントハンドラや Web サービスメソッドごとに生成や削除を繰り返した場合，性能劣化やメモリ使用効率低下の原因となります。

注 2

出力データ用の MessageBuffer 実装オブジェクトの MessageLength プロパティにあらかじめ受信データサイズ設定しなかった場合は，バッファ長が受信データサイズと見なされます。バッファ長が実際の受信データサイズと異なる場合は，受信データにずれが生じたり，受信タイムアウトが発生したりすることがあります。

4.6.4 TCP/IP 通信機能とバッファプーリング機能を使用する場合のコーディング例（C#の場合）

TCP/IP 通信機能とバッファプーリング機能を使用する場合の，ASP.NET Web アプリケーションでの使用例を次に示します。

コメント中の(1)，(2)などは「4.6.3 TCP/IP 通信機能とバッファプーリング機能を使用する場合のメソッド発行手順」の説明の番号に対応しています。

```
using System;
using System.Text;
using Hitachi.OpenTP1;
using Hitachi.OpenTP1.Connector;
using MyCompany;

namespace MyCompany
{
  public class MyForm1 : System.Web.UI.Page
  {
    ...
    private void Button1_Click(object sender, System.EventArgs e)
    {
      TP1Connection tc = null;
      MessageBuffer inBuf = null;
      MessageBuffer outBuf = null;
      // このボタンがクリックされたら
      // OpenTP1<ServerA>のサービス要求を行う
      // グローバル変数のアプリケーション状態から
      // TP1ConnectionManagerを取得する
      // (1) TP1ConnectionManagerクラスの生成
```

```
TP1ConnectionManager tcm =
   (TP1ConnectionManager)this.Application["tcmServerA"];
// (2) TcpipConnectionオブジェクトの取得
tc = tcm.GetTcpipConnection();
try {
  Encoding enc = Encoding.Default;
  // (3) TcpipInfoクラスの生成
  TcpipInfo info = new TcpipInfo();
  // (4) 通信形態
  info.Flags = TcpipInfo.TCPIP_SENDRECV;
  // (4) 最大応答待ち時間
  info.WatchTime = 180;
  // (5) 入力用インデクスドレコードの生成
  IndexedRecord input =
               tcm.CreateIndexedRecord("in_record");
  // (6)(7) 入力データ用メッセージバッファの取得,
  // および入力データの設定
  string indata = "Say Hello to OpenTP1!";
  // (6) 入力データ用メッセージバッファの取得
  int indataLen = enc.GetByteCount(indata);
  // (6) 入力データ用メッセージバッファの取得
  inBuf = tcm.GetMessageBuffer(indataLen);
  // (7) 入力データの設定
  inBuf.Append(enc.GetBytes(indata));
  // (8) 入力用インデクスドレコードの設定
  input.Add(inBuf);
  // (9) 出力用インデクスドレコードの生成
  IndexedRecord output =
               tcm.CreateIndexedRecord("out_record");
  // (10) 出力データ用メッセージバッファの取得
  // および出力用インデクスドレコードの設定
  outBuf = tcm.GetMessageBuffer(this.textBox1.MaxLength*2);
  outBuf.MessageLength = 32;
  output.Add(outBuf);
  // (11) TCP/IP通信の実行
  bool ret = tc.Execute(info, input, output);
  // (12) データの取り出し
  outBuf = (MessageBuffer)output[0];
  // (12) データの取り出し
  byte[] outdata = outBuf.Buffer;
  // (12) データの取り出し
  this.textBox1.Text = enc.GetString(outdata);
} catch (TP1ConnectorException exp) {
  // Connector .NETが検知したエラー
} catch (TP1Exception exp) {
  // Connector .NET (OpenTP1共通クラス) が検知したエラー
} catch (Exception exp) {
  // 予期しない例外
}
finally
{
  // (13) 入力用メッセージバッファをバッファプールに戻す
  if (inBuf != null) inBuf.ReleaseMessageBuffer();
  // (14) 出力用メッセージバッファをバッファプールに戻す
  if (outBuf != null) outBuf.ReleaseMessageBuffer();
  // (15) コネクションをコネクションプールに戻す
  if (tc != null) tc.Dispose();
 }
```

```
        }
    }
}
```

4.7 UAP 作成時の注意事項

4.7.1 アプリケーションプログラムの実行環境と留意点

Connector .NET は，Connector .NET のアプリケーションから利用するクラスライブラリをアセンブリ（DLL 形式）として提供します。Connector .NET のアプリケーションは，その形態によってさまざまなランタイムホストで実行されます。

使用するランタイムホストごとに，アプリケーションドメインの構成，アセンブリの配置と検索方法，セキュリティポリシー，スレッド構成，およびアプリケーション構成ファイルの配置が異なりますので，アプリケーションプログラムを実行する前に，使用するランタイムホストの仕様を確認してください。

4.7.2 例外の捕捉とエラーの判定

Connector .NET のアプリケーションでは，Connector .NET が提供するクラスライブラリやスタブ，.NET Framework などから例外が発生する場合があります。これらの例外は適切に捕捉してください。

Connector .NET からは，次の例外が発生する可能性があります。

表 4-12 発生する可能性がある例外

名前空間	例外名	意味
Hitachi.OpenTP1	TP1Exception	共通の例外基底クラス
	TP1RemoteException	サーバで発生した例外の受信
	TP1UserException	ユーザが任意に使用
	TP1MarshalException	スタブでのデータ変換エラー
Hitachi.OpenTP1.Connector	TP1ConnectorException	クラスライブラリでのエラー検知
	TcnIllegalArgumentException（TP1ConnectorException のサブクラス）	メソッド呼び出し時の引数の不正
	TcnIllegalStateException（TP1ConnectorException のサブクラス）	メソッド呼び出し時の状態の不正
	TcnNotUsedException（TP1ConnectorException のサブクラス）	設定されていない機能の使用

(1) TP1Exception

OpenTP1 のすべての例外の基底クラスです。個別の例外を捕捉しないで，まとめて捕捉したい場合などにこの TP1Exception クラスですべての OpenTP1 の例外を捕捉できます。

(2) TP1RemoteException

クライアントスタブを使用して RPC 要求をした場合に，サーバ側で例外が発生したことを示す例外です。呼び出し元のクライアント側で発生した例外とサーバ側で発生した例外を区別できます。

(3) TP1UserException

クライアント，およびサーバを.NET インタフェース定義を使用した OpenTP1 for .NET Framework の UAP で作成した場合に，RPC 要求されたサービスメソッドからクライアントにこの例外を返すことができます。

この例外は UAP で任意に使用できます。クライアント側でも TP1RemoteException には変換されません。

なお，.NET インタフェース定義を使用しない場合は，この例外を使用しないでください。使用した場合，クライアント側に RPC 要求のエラーが通知されます。

(4) TP1MarshalException

クライアントスタブを使用して RPC 要求をした場合に，データ変換エラーが発生したことを示す例外です。クライアントとサーバでインタフェースの定義が一致していない場合や，サービス定義（データ型定義）に誤りがあった場合などに発生します。

(5) TP1ConnectorException

Connector .NET が提供するクラスライブラリの各クラスおよびメソッドでエラーを検知した場合に発生する例外です。この例外がそのまま発生することもありますが，この例外のサブクラスとして発生する例外もあります。

(6) TcnIllegalArgumentException

メソッド呼び出し時の引数が不正な場合に発生します。

(7) TcnIllegalStateException

不正な状態でメソッドを呼び出した場合に発生します。

(8) TcnNotUsedException

設定されていない機能を使用しようとした場合に発生します。

4.7.3 COBOL 言語で作成する場合の注意事項

COBOL 言語で UAP を作成する場合，次の点に注意してください。

- .NET インタフェース定義およびサービス定義は使用できません。
- リポジトリ段落で，Connector .NET が提供するクラスを参照するように指定してください。
- PUBLIC クラスとして宣言してください。
- AS 指定のないクラス名の場合，名前空間は「COBOL2002.DefaultNamespace」が仮定されます。AS 指定があるクラス名の場合，名前空間は仮定されません。

COBOL 言語のそのほかの注意事項については，マニュアル「COBOL2002 for .NET Framework ユーザーズガイド」を参照してください。

4.7.4 クラスライブラリを使用する場合の注意事項

クラスライブラリを使用する場合，次の点に注意してください。

- Hitachi.OpenTP1 で始まる名前空間は OpenTP1 が使用するため，OpenTP1 以外のアプリケーションでは使用できません。
- Connector .NET が提供するクラスライブラリは SPP.NET，SUP.NET 上では使用しないでください。これらの UAP 上では，Extension .NET が提供するクラスライブラリを使用してください。

4.8　サンプルプログラムの使用方法

Connector .NET のサンプルプログラムのディレクトリ構成，ビルド方法および実行手順を説明します。なお，以降の説明で，%DCCNNDIR%は Connector .NET のインストールディレクトリを示します。

4.8.1　ディレクトリ構成

Connector .NET のサンプルプログラムは，次の表に示すディレクトリに格納されています。

表 4–13　サンプルプログラムのディレクトリ構成

ディレクトリ	説明
%DCCNNDIR%¥examples	Connector .NET のサンプルプログラム格納ディレクトリ
%DCCNNDIR%¥examples¥C#	C#のサンプルプログラム格納ディレクトリ
%DCCNNDIR%¥examples¥J#	J#のサンプルプログラム格納ディレクトリ
%DCCNNDIR%¥examples¥VB.NET	Visual Basic のサンプルプログラム格納ディレクトリ
%DCCNNDIR%¥examples¥COBOL.NET	COBOL 言語のサンプルプログラム格納ディレクトリ
%DCCNNDIR%¥examples¥WSClient	ASP.NET XML Web サービスのサンプルプログラムアクセス用のクライアント（C#） （C#，J#および Visual Basic のサンプルプログラム用）
%DCCNNDIR%¥examples¥WSCltCB	ASP.NET XML Web サービスのサンプルプログラムアクセス用のクライアント（COBOL.NET） （COBOL 言語のサンプルプログラム用）
%DCCNNDIR%¥examples¥WCF	WCF 連携機能のサンプルプログラム格納ディレクトリ （C#および Visual Basic でだけ提供）

各言語のサンプルプログラム格納ディレクトリ下は，次の表に示す名称のディレクトリで構成されます。それぞれのディレクトリの名称と，格納されているサンプルプログラムの種類について，開発言語別に次の表に示します。

表 4–14　サンプルプログラムの種類（C#，J#，および Visual Basic の場合）

名称※	説明
xxWABIN	RPC 要求インタフェースとしてバイナリデータを使用した ASP.NET Web アプリケーションです。 このサンプルプログラムでは，クライアントスタブを使用しません。
xxWSCR	RPC 要求インタフェースとしてサービス定義から生成された TP1 Service Proxy とカスタムレコードを使用した ASP.NET XML Web サービスです。
xxWAIF	RPC 要求インタフェースとして.NET インタフェース定義から生成されたクライアントスタブを使用した ASP.NET Web アプリケーションです。

名称※	説明
xxWAXML	RPC 要求インタフェースとして.NET インタフェース定義から生成されたクライアントスタブを使用し，かつ RPC データの XML マッピング機能を使用した ASP.NET Web アプリケーションです。
xxWAWCF	WCF 連携機能を使用して OpenTP1 にサービスを要求する Windows アプリケーションです（C# および Visual Basic でだけ提供しています）。

注※

名称の先頭の xx は，言語ごとに次の文字に置き換えてください。

C#：CS，J#：VJ，Visual Basic：VB

表 4-15　サンプルプログラムの種類（COBOL 言語の場合）

名称	説明
CBWSBIN	RPC 要求インタフェースとしてバイナリデータを使用した ASP.NET Web サービスです。 このサンプルプログラムでは，クライアントスタブを使用しません。

4.8.2　サンプルプログラムのビルド方法

Connector .NET のサンプルプログラムをビルドする手順を説明します。

(1)　サンプルプログラムをビルドするための準備

1. 次のコマンドプロンプトを起動します。
   - C#および Visual Basic の場合

     Visual Studio が提供するコマンドプロンプト，または .NET Framework SDK が提供するコマンドプロンプト

2. 環境変数 DCCLNDIR に，TP1/Client for .NET Framework インストールディレクトリを設定します。

   【例】set DCCLNDIR=C:¥Program Files¥HITACHI¥TP1Client for .NET Framework

3. 環境変数 DCCNNDIR に，Connector .NET のインストールディレクトリを設定します。

   【例】set DCCNNDIR=C:¥Program Files¥HITACHI¥TP1Connector for .NET Framework

4. WCF 連携機能のサンプルプログラムをビルドする場合は，環境変数 WCFDIR に WCF のライブラリが格納されているディレクトリを設定します。

   【例】set WCFDIR=C:¥WINDOWS¥Microsoft.NET¥Framework¥v3.0¥Windows Communication Foundation

5. 環境変数 Path，LIBPATH に.NET Framework のパスを設定してください。ただし，WCF 連携機能のサンプルプログラムをビルドする場合は，環境変数 Path，LIBPATH に.NET Framework v2.0 のパスを設定してください。

   【例】.NET Framework のバージョンが v3.5 の場合

set Path=%SystemRoot%¥Microsoft.NET¥Framework¥v3.5;%Path%
set LIBPATH=%SystemRoot%¥Microsoft.NET¥Framework¥v3.5;%LIBPATH%
【例】.NET Framework のバージョンが v4 の場合
set Path=%SystemRoot%¥Microsoft.NET¥Framework¥v4.0.30319;%Path%
set LIBPATH=%SystemRoot%¥Microsoft.NET¥Framework¥v4.0.30319;%LIBPATH%

ユーザアカウント制御（UAC）で保護されたディレクトリ下※ではサンプルプログラムのビルドに失敗します。このため，ユーザアカウント制御（UAC）で保護されたディレクトリ下でサンプルプログラムをビルドする場合は，［管理者として実行］で起動したコマンドプロンプト上でビルドしてください。

注※
Connector .NET のサンプルプログラム格納ディレクトリ（<OS のインストールディレクトリ>:¥Program Files¥HITACHI¥TP1Connector for .NET Framework）もユーザアカウント制御（UAC）で保護されたディレクトリに該当します。

(2) サンプルプログラムをまとめてビルドする方法

サンプルプログラムはをまとめてビルドする方法は，どの開発言語のサンプルプログラムをビルドするかによって異なります。それぞれの場合に分けて方法を次に示します。

- C#，J#，および Visual Basic を使用した場合のサンプルプログラムをまとめてビルドする場合
  %DCCNNDIR%¥examples¥build_all.bat を実行します。
- すべての開発言語のサンプルプログラムをまとめてビルドする場合
  %DCCNNDIR%¥examples¥build_all2.bat を実行します。
  ただし，build_all.bat，および build_all2.bat を実行しても WCF 連携機能のサンプルプログラムはビルドされません。
- WCF 連携機能のサンプルプログラムをビルドする場合
  %DCCNNDIR%¥examples¥WCF¥build_wcf.bat を実行します。
- 言語別にサンプルプログラムをまとめてビルドする場合
  - C#の場合
    %DCCNNDIR%¥examples¥C#¥build_cs.bat を実行します。
  - J#の場合
    %DCCNNDIR%¥examples¥J#¥build_vjs.bat を実行します。
  - Visual Basic の場合
    %DCCNNDIR%¥examples¥VB.NET¥build_vb.bat を実行します。
  - COBOL 言語の場合
    %DCCNNDIR%¥examples¥COBOL.NET¥build_cbl.bat を実行します。

(3)　サンプルプログラムを個別にビルドする方法

各サンプルプログラム格納ディレクトリ下の build.bat を実行します。

【例】%DCCNNDIR%¥examples¥VB.NET¥SPPIF¥build.bat

4.8.3　サンプルプログラムの実行手順

Connector .NET のサンプルプログラムの実行手順を説明します。

(1)　SPP.NET のサンプルプログラムの起動

各サンプルプログラムを実行するためには，Extension .NET のサンプルプログラムのうち，各サンプルプログラムが利用する SPP.NET を起動します。各サンプルプログラムが利用する SPP.NET を次の表に示します。

SPP.NET の起動方法については，マニュアル「TP1/Extension for .NET Framework 使用の手引」を参照してください。

表 4–16　サンプルプログラムが利用する SPP.NET

サンプルプログラムの種類	利用する SPP.NET	ユーザサーバ名
C#¥CSWABIN	C#¥SPPBIN	CSSPPBIN
C#¥CSWSCR	C#¥SPPBIN	CSSPPBIN
C#¥CSWAIF	C#¥SPPIF	CSSPPIF
C#¥CSWAXML	C#¥SPPIF	CSSPPIF
J#¥VJWABIN	J#¥SPPBIN	VJSPPBIN
J#¥VJWSCR	J#¥SPPBIN	VJSPPBIN
J#¥VJWAIF	J#¥SPPIF	VJSPPIF
J#¥VJWAXML	J#¥SPPIF	VJSPPIF
VB.NET¥VBWABIN	VB.NET¥SPPBIN	VBSPPBIN
VB.NET¥VBWSCR	VB.NET¥SPPBIN	VBSPPBIN
VB.NET¥VBWAIF	VB.NET¥SPPIF	VBSPPIF
VB.NET¥VBWAXML	VB.NET¥SPPIF	VBSPPIF
COBOL.NET¥CBWSBIN	COBOL.NET¥SPPOBJ	CBSPPOBJ
WCF¥CSWINWCF	C#¥SPPBIN	CSSPPBIN
WCF¥VBWINWCF	VB.NET¥SPPBIN	VBSPPBIN

(2) 構成ファイルの変更

利用する OpenTP1 サーバ環境に応じて，構成ファイルの Client .NET 構成定義を変更します。例えば，Visual Basic の WAIF サンプルで，OpenTP1 のホスト名やネームサービスのポート番号などを変更する場合は，%DCCNNDIR%¥examples¥VB.NET¥VBWAIF¥web.config ファイルの tp1Server 要素，rpc 要素および nameService 要素を変更します。

(3) IIS での公開

ビルド済みのサンプルプログラムのディレクトリを指定して仮想ディレクトリを新規作成します。手順は次のとおりです。なお，WCF 連携機能のサンプルプログラムではこの作業は不要です。

1. 「コントロールパネル」の「管理ツール」から，「インターネットインフォメーションサービス」で IIS の管理画面を表示します。
2. 「ローカルコンピュータ」－「サイト」－「Default Web Site」を選択し，仮想ディレクトリの追加」を選択します。
3. 「エイリアス」にサンプルプログラム名称を，「物理パス」にサンプルプログラムのディレクトリを指定します。

   【例】Visual Basic の WAIF サンプルの場合

   エイリアス：VBWAIF

   物理パス：%DCCNNDIR%¥examples¥VB.NET¥VBWAIF
4. 作成した仮想ディレクトリの「アクセス許可の編集」を選択し，「プロパティ」－「セキュリティ」でアクセス許可を設定します。
5. 作成した仮想ディレクトリの「アプリケーションへの変換」を選択します。

(4) クライアントからのアクセス

クライアントを実行してアクセスします。

ASP.NET Web アプリケーションの場合

Web ブラウザ（Microsoft Internet Explorer）を起動して，次の URL を入力してアクセスします。

```
http://<アドレス>/<仮想ディレクトリ名>/WebForm<サンプル名>.aspx
```

【例】Visual Basic の WAIF サンプルの場合

```
http://localhost/VBWAIF/WebFormVBWAIF.aspx
```

ASP.NET XML Web サービスの場合

1. サンプルプログラムをビルドして起動します。

C#，J#，および Visual Basic の場合

WSClient サンプルプログラムを使用します。

【例】%DCCNNDIR%¥examples¥WSClient¥WSClient.exe

COBOL 言語の場合

WSCltCB サンプルプログラムを使用します。

【例】%DCCNNDIR%¥examples¥WSCltCB¥WSCltCB.exe

2. 次の URL を入力してアクセスします。

```
http://<アドレス>/<仮想ディレクトリ名>/CUSTOMSVWebService.asmx
```

【例】Visual Basic の WSCR サンプルの場合

http://localhost/VBWSCR/CUSTOMSVWebService.asmx

WCF 連携機能のサンプルプログラムの場合

そのまま WCF 連携機能のサンプルプログラムを起動します。

【例】Visual Basic の WINWCF サンプルの場合

```
%DCCNNDIR%¥examples¥WCF¥VBWINWCF¥VBWinWCF.exe
```

5 運用コマンド

この章では，Connector .NET で使用する運用コマンドについて説明します。

運用コマンドの種類

Connector .NET で使用できる運用コマンドを，次の表に示します。

表 5-1　運用コマンドの種類（Connector .NET）

運用コマンド	機能
if2cstub	C#，または Visual Basic で定義された.NET インタフェース定義を基に，クライアントスタブクラスのソースファイルを生成します。
if2tsp	C#，または Visual Basic で定義された.NET インタフェース定義を基に，TSP 実装クラス，XML Web サービスクラス，asmx ファイル，クライアントスタブ，および構成ファイルを生成します。
spp2cstub	指定されたサービス定義ファイル，およびそのサービス定義ファイルが参照するデータ型定義ファイルを基に，クライアントスタブクラスおよびカスタムレコードクラスのソースファイルを生成します。
spp2tsp	指定されたサービス定義ファイルおよびそのサービス定義ファイルが参照するデータ型定義ファイルを基に，TSP 実装クラス，XML Web サービスクラス，asmx ファイル，カスタムレコードクラス，クライアントスタブ，および構成ファイルを生成します。
cnnnidgen	Connector .NET の MSDTC 連携機能で使用するノード識別子を生成し，文字列で標準出力に表示します。
cnntrsls	トランザクションリカバリサービスの状態をアプリケーションドメインごとに一覧表示します。

if2cstub（クライアントスタブ生成コマンド（.NET インタフェース定義用））

形式

```
if2cstub {-t {svr|clt|con}
         〔-l {cs|vb}〕
         〔-s 生成ファイル拡張子〕
         〔-n 名前空間名称〕
         〔-o 出力先ディレクトリ〕
         〔-r スタブクラス名称〕
         〔-c {struct|nostruct}〕
         〔-X {normal|dataset}〕
         〔-m RPCメッセージの最大長〕
          -i .NETインタフェース定義ファイル名称
          インタフェース名称
         |-h}
```

機能

C#，または Visual Basic で定義された .NET インタフェース定義を基に，クライアントスタブクラスのソースファイル（以降，クライアントスタブソースファイルと呼びます）を生成します。

オプション

●-t {svr | clt | con}

生成するクライアントスタブの種類を指定します。

オプションの指定と生成されるクライアントスタブを次に示します。

オプションの指定	生成されるクライアントスタブ
svr	SPP.NET または SUP.NET（Extension .NET）用
clt	CUP.NET（Client .NET）用
con	CUP.NET（Connector .NET）用

●-l {cs | vb}

生成するクライアントスタブソースファイルのプログラム言語を指定します。

このオプションを省略した場合，入力元のソースファイルと同じプログラム言語で生成します。入力元のソースファイルのプログラム言語は，ファイルの拡張子を基に判断されます。

オプションの指定と生成されるクライアントスタブソースファイルのプログラム言語の関係を次に示します。

オプションの指定	生成されるクライアントスタブソースファイルのプログラム言語
cs	C#
vb	Visual Basic

このオプションを省略した場合に生成されるクライアントスタブソースファイルのプログラム言語を次に示します。

入力元のソースファイルの拡張子	生成されるクライアントスタブソースファイルのプログラム言語
cs	C#
vb	Visual Basic

●-s　生成ファイル拡張子　　～〈文字列〉

生成するクライアントスタブソースファイルの拡張子を指定します。

このオプションを省略した場合，生成されるクライアントスタブソースファイルのプログラム言語によって，拡張子は次のようになります。

生成されるクライアントスタブソースファイルのプログラム言語	生成されるクライアントスタブソースファイルの拡張子
C#	cs
Visual Basic	vb

●-n　名前空間名称　　～〈文字列〉

生成するクライアントスタブクラスの名前空間名称を指定します。

このオプションを省略した場合，入力元のインタフェースが属する名前空間と同じ名前空間でクライアントスタブクラスが生成されます。入力元のインタフェースが名前空間なしの場合，名前空間なしのクライアントスタブクラスが生成されます。

●-o　出力先ディレクトリ　　～〈パス名〉

生成するクライアントスタブソースファイルを出力するディレクトリを指定します。絶対パスまたは相対パスで指定してください。

このオプションを省略した場合，コマンド実行時のディレクトリに出力されます。なお，ファイル名はクライアントスタブごとに「〈名前空間を含まないスタブクラス名称〉.〈拡張子〉」で生成されます。

●-r　スタブクラス名称　　～〈文字列〉

生成するクライアントスタブのクラス名称を指定します。

このオプションを省略した場合，クラス名称は「〈インタフェース名称〉Stub」になります。

●-c　{struct | nostruct}　　～《struct》

.NET インタフェース定義のメソッドのパラメタまたは戻り値に TP1 ユーザ構造体を使用した場合に，クライアントスタブが利用するための TP1 ユーザ構造体クラスを出力するかどうかを指定します。

このオプションを省略した場合，TP1 ユーザ構造体クラスを出力します。.NET インタフェース定義に TP1 ユーザ構造体を指定していなかった場合，このオプションの指定は無視されます。

struct：TP1 ユーザ構造体クラスを出力します。

nostruct：TP1 ユーザ構造体クラスを出力しません。

●-X　{normal | dataset}

生成されるクライアントスタブに，引数および戻り値が XmlDocument クラス（System.Xml.XmlDocument）となるサービスメソッドが追加されます。

また，サービスメソッドに入力パラメタがある場合，入力データ用 XML スキーマファイルがサービスメソッドごとに出力されます。出力パラメタまたは戻り値がある場合，出力データ用 XML スキーマファイルがサービスメソッドごとに出力されます。

-t オプションに con を指定した場合だけ，このオプションの指定が有効になります。-t オプションに con 以外を指定した場合，このオプションの指定は無視されます。

このオプションを省略した場合，生成されるクライアントスタブに，引数および戻り値が XmlDocument クラスとなるサービスメソッドは追加されません。また，入力データ用 XML スキーマファイルおよび出力データ用 XML スキーマファイルは出力されません。

normal：引数および戻り値の XmlDocument オブジェクト（System.Xml.XmlDocument）を，.NET Framework の DataSet オブジェクト（System.Data.DataSet）と連携させない場合に指定します。normal を指定した場合，出力される入力データ用 XML スキーマファイルおよび出力データ用 XML スキーマファイルは，DataSet オブジェクトで利用できない場合があります。

dataset：引数および戻り値の XmlDocument オブジェクト（System.Xml.XmlDocument）を，.NET Framework の DataSet オブジェクト（System.Data.DataSet）と連携させて利用する場合に指定します。dataset を指定した場合，DataSet オブジェクトで利用できる入力データ用 XML スキーマファイルおよび出力データ用 XML スキーマファイルを出力します。また，その入力データ用 XML スキーマおよび出力データ用 XML スキーマに対応したクライアントスタブを出力します。

●-m　RPC メッセージの最大長　　～〈符号なし整数〉((1～8))《1》(単位：メガバイト)

サービスメソッドが指定する RPC メッセージの最大長を指定します。

このオプションに 1～8 以外を指定した場合は，エラーが発生します。なお，このオプションは，-t オプションに svr を指定したときだけ有効です。-t オプションの指定が svr 以外の場合は，-m オプションの指定は無視されます。

●-i　.NET インタフェース定義ファイル名称　　～〈ファイル名〉

入力元のインタフェースが定義されている.NET インタフェース定義ファイル名称を指定します。絶対パスまたは相対パスで指定してください。

.NET インタフェース定義を作成したプログラム言語は，このオプションで指定したファイルの拡張子によって次のように判断されます。

このオプションで指定した.NET インタフェース定義ファイルの拡張子	.NET インタフェース定義を作成したプログラム言語の判断結果
cs	C#
vb	Visual Basic

●-h

このコマンドの使用方法を標準出力に表示します。

このオプションを指定した場合，ほかのオプションおよびコマンド引数は指定できません。

コマンド引数

●インタフェース名称

生成したいクライアントスタブに対応する SPP.NET のインタフェース名称を完全限定名で指定します。インタフェース名称は一つだけ指定できます。

注意事項

- このコマンドの実行時にエラーが発生した場合は，対応するエラーメッセージが標準エラー出力に出力されます。
- このコマンドで生成したクライアントスタブソースファイルの内容は変更しないでください。
- 入力元となる.NET インタフェース定義ファイルは，使用する Windows のデフォルトコードページ（日本語版 Windows の場合は 932）で保存してください。Unicode など，ほかのコードページで保存したファイルは，このコマンドの入力元として使用できません。

if2tsp（TSP 生成コマンド（.NET インタフェース定義用））

形式

```
if2tsp { 〔-l {cs|vb}〕
         〔-s 生成ファイル拡張子〕
         〔-n 名前空間名称〕
         〔-o 出力先ディレクトリ〕
         〔-r クラス名称〕
         〔-c {struct|nostruct}〕
         〔-t soap〕
         〔-S {doc|rpc}〕
         〔-x {literal|encoded}〕
         〔-w XML Web サービスで使用するXML名前空間名称〕
         〔-N SOAPメッセージに関連づけられる名前空間名称〕
         〔-B {wsibp11|none}〕
         〔-A {true|false}〕
         〔-p Connector .NETが利用する構成定義のプロファイルID〕
          -i .NETインタフェース定義ファイル名称
          -g サービスグループ名
          インタフェース名称
         |-h}
```

機能

C#，または Visual Basic で定義された.NET インタフェース定義を基に，TSP 実装クラス，XML Web サービスクラス，asmx ファイル，クライアントスタブ，および構成ファイルを生成します。

オプション

●-l {cs | vb}

生成する TSP 実装クラス，XML Web サービスクラス，およびクライアントスタブクラスのソースファイルのプログラム言語を指定します。

このオプションを省略した場合，入力元のソースファイルと同じプログラム言語で生成します。入力元のソースファイルのプログラム言語は，ファイルの拡張子を基に判断されます。

オプションの指定と，生成される TSP 実装クラス，XML Web サービスクラス，およびクライアントスタブクラスのソースファイルのプログラム言語を次に示します。

オプションの指定	生成されるソースファイルのプログラム言語
cs	C#
vb	Visual Basic

このオプションを省略した場合に生成される TSP 実装クラス，XML Web サービスクラス，およびクライアントスタブクラスのソースファイルのプログラム言語を次に示します。

入力元ソースファイルの拡張子	生成されるソースファイルのプログラム言語
cs	C#
vb	Visual Basic

●-s　生成ファイル拡張子　　～〈文字列〉

生成する TSP 実装クラス，XML Web サービスクラス，およびクライアントスタブの拡張子を指定します。

このオプションを省略した場合，生成される TSP 実装クラス，XML Web サービスクラス，およびクライアントスタブクラスのソースファイルのプログラム言語によって，拡張子は次のようになります。

生成されるソースファイルのプログラム言語	生成されるソースファイルの拡張子
C#	cs
Visual Basic	vb

●-n　名前空間名称　　～〈文字列〉

生成する TSP 実装クラス，XML Web サービスクラス，およびクライアントスタブクラスの名前空間名称を指定します。

このオプションを省略した場合，入力元のインタフェースが属する名前空間と同じ名前空間で TSP 実装クラス，XML Web サービスクラス，およびクライアントスタブが生成されます。入力元のインタフェースが名前空間なしの場合，名前空間なしの TSP 実装クラス，XML Web サービスクラス，およびクライアントスタブが生成されます。

●-o　出力先ディレクトリ　　～〈パス名〉

生成する TSP 実装クラス，XML Web サービスクラス，クライアントスタブ，asmx ファイル，および構成ファイルを出力するディレクトリを指定します。絶対パスまたは相対パスで指定してください。

このオプションを省略した場合，コマンド実行時のディレクトリに出力されます。

●-r　クラス名称　　～〈文字列〉

生成する TSP 実装クラス，XML Web サービスクラス，およびクライアントスタブクラスのクラス名称，ならびに構成ファイルのファイル名称を指定します。

このオプションを指定した場合と省略した場合のクラス名称およびファイル名称は次のとおりです。

クラスおよびファイル	生成されるクラス名称およびファイル名称	
	オプションを指定した場合	オプションを省略した場合
TSP 実装クラス	〈クラス名称〉Proxy	〈インタフェース名称〉Proxy
XML Web サービスクラス	〈クラス名称〉WebService	〈インタフェース名称〉WebService
クライアントスタブクラス	〈クラス名称〉Stub	〈インタフェース名称〉Stub

クラスおよびファイル	生成されるクラス名称およびファイル名称	
	オプションを指定した場合	オプションを省略した場合
構成ファイル	〈TSP 実装クラス名称〉.config	〈インタフェース名称〉Proxy.config

なお，上記の表の〈クラス名称〉部分だけが，-r オプションで指定するクラス名称です。

●-c {struct | nostruct} ～《struct》

.NET インタフェース定義のメソッドのパラメタまたは戻り値に TP1 ユーザ構造体を使用した場合に，クライアントスタブが利用するための TP1 ユーザ構造体クラスを出力するかどうかを指定します。

このオプションを省略した場合，TP1 ユーザ構造体クラスを出力します。.NET インタフェース定義に TP1 ユーザ構造体を指定していなかった場合，このオプションの指定は無視されます。

struct：TP1 ユーザ構造体クラスを出力します。

nostruct：TP1 ユーザ構造体クラスを出力しません。

●-t soap

XML Web サービスクラスを生成します。

このオプションを省略した場合，XML Web サービスクラスは生成されません。

●-S {doc | rpc} ～《doc》

XML Web サービスメソッドとの間で送受信される SOAP メッセージの書式を指定します。このオプションは，-t オプションを指定した場合にだけ有効です。指定しなかった場合，-S オプションの指定は無視されます。

このオプションを省略した場合，doc が仮定されます。

オプションの指定と送受信される SOAP メッセージの書式を次に示します。

オプションの指定	送受信される SOAP メッセージの書式
doc	XML Web サービスメソッドとの間で送受信される SOAP メッセージの書式は，Document になります。
rpc	XML Web サービスメソッドとの間で送受信される SOAP メッセージの書式は，RPC になります。

●-x {literal | encoded} ～《literal》

XML Web サービスメソッドとの間で送受信される SOAP パラメタの書式指定スタイルを指定します。このオプションは，-t オプションを指定した場合かつ SOAP メッセージの書式が Document の場合にだけ有効です。-t オプションを指定しなかった場合，-x オプションの指定は無視されます。

-S オプションに rpc を指定した場合，-x オプションの指定は無視され，encoded が仮定されます。

このオプションを省略した場合，literal が仮定されます。

オプションの指定と送受信される SOAP パラメタの書式指定スタイルを次に示します。

オプションの指定	送受信される SOAP パラメタの書式指定スタイル
literal	XML Web サービスメソッドとの間で送受信される SOAP パラメタの書式指定スタイルは，Literal になります。
encoded	XML Web サービスメソッドとの間で送受信される SOAP パラメタの書式指定スタイルは，Encoded になります。

●-w　XML Web サービスで使用する XML 名前空間名称　　～〈文字列〉

XML Web サービスで使用する XML 名前空間名称を指定します。このオプションは，-t オプションを指定した場合にだけ有効です。指定しなかった場合，-w オプションの指定は無視されます。

このオプションを省略した場合の名前空間名称は，.NET Framework のデフォルト値になります。.NET Framework のデフォルト値については，.NET Framework のドキュメントを参照してください。

●-N　SOAP メッセージに関連づけられる名前空間名称　　～〈文字列〉

XML Web サービスメソッドに対する SOAP 要求，および SOAP 応答に関連づける名前空間を指定します。このオプションは，-t オプションを指定した場合にだけ有効です。指定しなかった場合，-N オプションの指定は無視されます。また，メソッドごとに名前空間名称を指定することはできません。

このオプションを省略した場合の名前空間名称は，.NET Framework のデフォルト値になります。.NET Framework のデフォルト値については，.NET Framework のドキュメントを参照してください。

●-B　{wsibp11 | none}

XML Web サービスクラスが，WS-I Basic Profile 1.1 に準拠するかどうかを宣言します。このオプションは-t オプションを指定し，かつ-S オプションに doc を指定した場合だけ有効です。これらのオプションを指定しなかった場合は，-B オプションの指定は無視されます。

wsibp11：XML Web サービスクラスが，WS-I Basic Profile 1.1 に準拠していることを宣言します。XML Web サービスクラスに，WebServiceBinding.ConformsTo プロパティの値として WsiProfiles.BasicProfile1_1 が付加されます。

none：XML Web サービスクラスが，WS-I Basic Profile 1.1 に準拠していることを宣言しません。XML Web サービスクラスに，WebServiceBinding.ConformsTo プロパティの値として WsiProfiles.None が付加されます。

●-A　{true | false}

XML Web サービスクラスの Web サービス記述言語（WSDL）ファイルで，WS-I Basic Profile 準拠の表示をするかどうかを指定します。このオプションは-B オプションを指定した場合だけ有効です。このオプションを省略した場合，WSDL ファイルで WS-I Basic Profile 準拠に関する表示はされません。

true：WSDL ファイルで，-B オプションで指定した WS-I Basic Profile 準拠の表示をします。XML Web サービスクラスに，WebServiceBinding.EmitConformanceClaims プロパティの値として true が付加されます。

false：WSDL ファイルで，-B オプションで指定した WS-I Basic Profile 準拠の表示をしません。XML Web サービスクラスに，WebServiceBinding.EmitConformanceClaims プロパティの値として false が付加されます。

●-p　Connector .NET が利用する構成定義のプロファイル ID　　～〈文字列〉

Connector .NET が利用する構成定義のプロファイル ID を指定します。

このオプションを省略した場合，デフォルトプロファイルの情報を利用して Connector .NET を使用します。

●-i　.NET インタフェース定義ファイル名称　　～〈ファイル名〉

入力元のインタフェースが定義されている.NET インタフェース定義ファイルを指定します。絶対パスまたは相対パスで指定してください。

.NET インタフェース定義を作成したプログラム言語は，このオプションで指定したファイルの拡張子によって次のように判断されます。

このオプションで指定した.NET インタフェース定義ファイルの拡張子	.NET インタフェース定義を作成したプログラム言語の判断結果
cs	C#
vb	Visual Basic

●-g　サービスグループ名　　～〈31 文字以内の識別子〉

呼び出したい SPP.NET のサービスグループ名を指定します。

このオプションの指定値を XML Web サービス生成後に変更したい場合は，サービスグループ名を構成ファイル（Web.config または machine.config）に指定します。詳細については，「2.6.3　TSP の動作設定」を参照してください。

●-h

このコマンドの使用方法を標準出力に表示します。

このオプションを指定した場合，ほかのオプションおよびコマンド引数は指定できません。

コマンド引数

●インタフェース名称

生成したい TSP に対応する SPP.NET のインタフェース名称を完全限定名で指定します。インタフェース名称は一つだけ指定できます。

注意事項

- このコマンドの実行時にエラーが発生した場合は，対応するエラーメッセージが標準エラー出力に出力されます。
- このコマンドで生成した TSP ソースファイルの内容を変更しないでください。
- 入力元となる.NET インタフェース定義ファイルは，使用する Windows のデフォルトコードページ（日本語版 Windows の場合は 932）で保存してください。Unicode など，ほかのコードページで保存したファイルは，このコマンドの入力元として使用できません。

spp2cstub（クライアントスタブ生成コマンド（サービス定義用））

形式

```
spp2cstub {-t {svr|clt|con}
          〔-l {cs|vb}〕
          〔-s 生成ファイル拡張子〕
          〔-n 名前空間名称〕
          〔-o 出力先ディレクトリ〕
          〔-r スタブクラス名称〕
          〔-R データ型定義名称:カスタムレコードクラス名称
               〔,データ型定義名称:カスタムレコードクラス名称〕…〕
          〔-F {space|null}〕
          〔-I エンコーディング名〕
          〔-O エンコーディング名〕
          〔-e {big|little}〕
          〔-E {big|little}〕
          〔-b〕
          〔-X {normal|dataset}〕
           -i サービス定義ファイル名称
          |-h}
```

機能

指定されたサービス定義ファイル，およびそのサービス定義ファイルが参照するデータ型定義ファイルを基に，クライアントスタブクラスおよびカスタムレコードクラスのソースファイルを生成します。

オプション

●-t {svr | clt | con}

生成するクライアントスタブの種類を指定します。

オプションの指定と生成されるクライアントスタブを次に示します。

オプションの指定	生成されるクライアントスタブ
svr	SPP.NET または SUP.NET（Extension .NET）用
clt	CUP.NET（Client .NET）用
con	CUP.NET（Connector .NET）用

●-l {cs | vb} 〜《cs》

生成するクライアントスタブクラスおよびカスタムレコードクラスのソースファイルのプログラム言語を指定します。クライアントスタブクラスとカスタムレコードクラスは同じプログラム言語で生成されます。

このオプションを省略した場合，cs が仮定されます。

オプションの指定と生成されるソースファイルのプログラム言語を次に示します。

オプションの指定	生成されるソースファイルのプログラム言語
cs	C#
vb	Visual Basic

●-s　生成ファイル拡張子　　～〈文字列〉

生成するクライアントスタブクラスおよびカスタムレコードクラスのソースファイルの拡張子を指定します。

このオプションを省略した場合，生成されるソースファイルのプログラム言語によって，拡張子は次のようになります。

生成されるソースファイルのプログラム言語	生成されるソースファイルの拡張子
C#	cs
Visual Basic	vb

●-n　名前空間名称　　～〈文字列〉

生成するクライアントスタブクラスおよびカスタムレコードクラスの名前空間名称を指定します。

このオプションを省略した場合，名前空間なしのクライアントスタブクラスおよびカスタムレコードクラスが生成されます。

●-o　出力先ディレクトリ　　～〈パス名〉

生成するクライアントスタブクラスおよびカスタムレコードクラスのソースファイルを出力するディレクトリを指定します。絶対パスまたは相対パスで指定してください。

このオプションを省略した場合，コマンド実行時のディレクトリに出力されます。なお，ファイル名はクライアントスタブクラスごとに「〈名前空間を含まないスタブクラス名称〉.〈拡張子〉」，カスタムレコードクラスごとに「〈名前空間を含まないカスタムレコードクラス名称〉.〈拡張子〉」で生成されます。

●-r　スタブクラス名称　　～〈文字列〉

生成するクライアントスタブのクラス名称を指定します。

このオプションを省略した場合，クラス名称は「〈サービス定義名称〉Stub」になります。なお，一つのサービス定義ファイルには，一つのサービス定義しか指定できません。

●-R　データ型定義名称:カスタムレコードクラス名称　　～〈文字列〉:〈31 文字以内の識別子〉

データ型定義ファイルで定義しているデータ型定義名称と，それを基にして生成するカスタムレコードのクラス名称を指定します。

このオプションを省略した場合，データ型定義名称がカスタムレコードのクラス名称に使用されます。

複数のデータ型定義からカスタムレコードを生成する場合は，データ型定義名称と生成されるカスタムレコードのクラス名称をコロン（:）で区切って指定します。ただし，データ型定義に存在しないデータ型定

義名称を指定した場合，存在しないデータ型定義名称は無視されます。存在するデータ型定義名称に対してだけカスタムレコードクラス名称の指定が有効になります。

●-F {space | null}　～《space》

入力レコードとなるカスタムレコードを引数として渡す場合，データ型定義で指定した文字列領域の余った領域に埋める文字を指定します。

このオプションを省略した場合，余った領域を半角スペースで埋めます。

space：半角スペースで埋めます。

null：ヌル文字で埋めます。

●-I　エンコーディング名　～〈文字列〉

TP1/Server への送信データを，エンコードするときに従うエンコード方式のエンコーディング名を指定します。指定できるエンコーディング名については，.NET Framework のドキュメントを参照してください。

このオプションを省略した場合，プラットフォームのデフォルトエンコーディング名になります。

サポートされているエンコード方式は，プラットフォームによって異なります。代表的なエンコーディング名は次のとおりです。

エンコーディング名	エンコード（Windows 上の表示名）	備考
euc-jp	日本語（EUC）	－
iso-8859-1	西ヨーロッパ言語（ISO）	－
shift_jis	日本語（シフト JIS）	MS932
unicodeFFFE	Unicode（Big-Endian）	－
us-ascii	US-ASCII	ISO646
utf-8	Unicode（UTF-8）	－
utf-16	Unicode	Little Endian

（凡例）

－：該当しません。

●-O　エンコーディング名　～〈文字列〉

TP1/Server から受け取った受信データを，デコードするときに従うエンコード方式のエンコーディング名を指定します。指定できるエンコーディング名については，.NET Framework のドキュメントを参照してください。

このオプションを省略した場合，プラットフォームのデフォルトエンコーディング名になります。

サポートされているエンコード方式は，プラットフォームによって異なります。代表的なエンコーディング名については，-I オプションの表を参照してください。

●-e {big | little} ～《big》

TP1/Server への送信データを，指定されたエンディアンに変換します。

このオプションを省略した場合，ビッグエンディアンに変換します。

big：ビッグエンディアンに変換します。

little：リトルエンディアンに変換します。

●-E {big | little} ～《big》

TP1/Server から受け取った受信データを，指定されたエンディアンであると仮定して変換します。

このオプションを省略した場合，ビッグエンディアンであると仮定して変換します。

big：ビッグエンディアンであると仮定して変換します。

little：リトルエンディアンであると仮定して変換します。

●-b

生成されたカスタムレコードクラスごとに必要なバッファサイズを標準出力に表示します。バッファプール機能を使用する場合で，バッファサイズの見積もりを行うときに使用します。

【表示例】

```
CustomRecordClassName : BufferSize[Bytes]
MyAppNS.Record1 : 448
MyAppNS.Record2 : 32
```

●-X {normal | dataset}

生成されるクライアントスタブに，引数および戻り値が XmlDocument クラス（System.Xml.XmlDocument）となるサービスメソッドが追加されます。

また，入力データ用 XML スキーマファイルがサービスごとに出力されます。出力データ型定義名称が DC_NODATA 以外の場合，出力データ用 XML スキーマファイルがサービスごとに出力されます。複数のサービスで同じデータ型定義名称を指定した場合，そのデータ型定義に対応する入力データ用 XML スキーマおよび出力データ用 XML スキーマは，一つだけ出力されます。

-t オプションに con を指定した場合だけ，このオプションの指定が有効になります。-t オプションに con 以外を指定した場合，このオプションの指定は無視されます。

このオプションを省略した場合，生成されるクライアントスタブに，引数および戻り値が XmlDocument クラスとなるサービスメソッドは追加されません。また，入力データ用 XML スキーマファイルおよび出力データ用 XML スキーマファイルは出力されません。

normal：引数および戻り値の XmlDocument オブジェクト（System.Xml.XmlDocument）を，.NET Framework の DataSet オブジェクト（System.Data.DataSet）と連携させない場合に指定します。normal を指定した場合，出力される入力データ用 XML スキーマファイルおよび出力データ用 XML スキーマファイルは，DataSet オブジェクトで利用できない場合があります。

dataset：引数および戻り値の XmlDocument オブジェクト（System.Xml.XmlDocument）を，.NET Framework の DataSet オブジェクト（System.Data.DataSet）と連携させて利用する場合に指定します。dataset を指定した場合，DataSet オブジェクトで利用できる入力データ用 XML スキーマファイルおよび出力データ用 XML スキーマファイルを出力します。また，その入力データ用 XML スキーマおよび出力データ用 XML スキーマに対応したクライアントスタブを出力します。

●-i　サービス定義ファイル名称　　～〈ファイル名〉

サービス定義が指定されているファイル名称を指定します。絶対パスまたは相対パスで指定してください。

●-h

このコマンドの使用方法を標準出力に表示します。

このオプションを指定した場合，ほかのオプションおよびコマンド引数は指定できません。

注意事項

- このコマンドの実行時にエラーが発生した場合は，対応するエラーメッセージが標準エラー出力に出力されます。
- このコマンドで生成したソースファイルの内容は変更しないでください。
- サービス定義ファイルの#include ディレクティブには，サービス定義ファイルの存在するディレクトリからの相対パスを指定します。#include ディレクティブに相対パスを指定していない場合は，データ型定義ファイルはサービス定義ファイルと同じディレクトリになければなりません。
- 入力元となるサービス定義ファイルおよびデータ型定義ファイルは，使用する Windows のデフォルトコードページ（日本語版 Windows の場合は 932）で保存してください。Unicode など，ほかのコードページで保存したファイルは，このコマンドの入力元として使用できません。

spp2tsp（TSP 生成コマンド（サービス定義用））

形式

```
spp2tsp { 〔-l {cs|vb}〕
          〔-s 生成ファイル拡張子〕
          〔-n 名前空間名称〕
          〔-o 出力先ディレクトリ〕
          〔-r クラス名称〕
          〔-t soap〕
          〔-S {doc|rpc}〕
          〔-x {literal|encoded}〕
          〔-w XML Web サービスで使用するXML名前空間名称〕
          〔-N SOAPメッセージに関連づけられる名前空間名称〕
          〔-B {wsibp11|none}〕
          〔-A {true|false}〕
          〔-p Connector .NETが利用する構成定義のプロファイルID〕
          〔-R データ型定義名称:カスタムレコードクラス名称
              〔,データ型定義名称:カスタムレコードクラス名称〕…〕
          〔-F {space|null}〕
          〔-I エンコーディング名〕
          〔-O エンコーディング名〕
          〔-e {big|little}〕
          〔-E {big|little}〕
          〔-b〕
          〔-d〕
           -g サービスグループ名
           -i サービス定義ファイル名称
          |-h}
```

機能

指定されたサービス定義ファイルおよびそのサービス定義ファイルが参照するデータ型定義ファイルを基に，TSP 実装クラス，XML Web サービスクラス，asmx ファイル，カスタムレコードクラス，クライアントスタブ，および構成ファイルを生成します。

オプション

●-l {cs | vb}　～《cs》

生成する TSP 実装クラス，XML Web サービスクラス，カスタムレコードクラス，およびクライアントスタブクラスのソースファイルのプログラム言語を指定します。

このオプションを省略した場合，cs が仮定されます。

オプションの指定と生成される TSP 実装クラス，XML Web サービスクラス，カスタムレコードクラス，およびクライアントスタブクラスのソースファイルのプログラム言語を次に示します。

オプションの指定	生成されるソースファイルのプログラム言語
cs	C#

オプションの指定	生成されるソースファイルのプログラム言語
vb	Visual Basic

●-s　生成ファイル拡張子　　～〈文字列〉

生成する TSP 実装クラス，XML Web サービスクラス，カスタムレコードクラス，およびクライアントスタブの拡張子を指定します。

このオプションを省略した場合，生成される TSP 実装クラス，XML Web サービスクラス，カスタムレコードクラス，およびクライアントスタブクラスのソースファイルのプログラム言語によって，拡張子は次のようになります。

生成されるソースファイルのプログラム言語	生成されるソースファイルの拡張子
C#	cs
Visual Basic	vb

●-n　名前空間名称　　～〈文字列〉

生成する TSP 実装クラス，XML Web サービスクラス，カスタムレコードクラス，およびクライアントスタブクラスの名前空間名称を指定します。

このオプションを省略した場合，名前空間なしの TSP 実装クラス，XML Web サービスクラス，カスタムレコードクラス，およびクライアントスタブが生成されます。

●-o　出力先ディレクトリ　　～〈パス名〉

生成する TSP 実装クラス，XML Web サービスクラス，asmx ファイル，カスタムレコードクラス，クライアントスタブ，および構成ファイルを出力するディレクトリを指定します。絶対パスまたは相対パスで指定してください。

このオプションを省略した場合，コマンド実行時のディレクトリに出力されます。

●-r　クラス名称　　～〈文字列〉

生成する TSP 実装クラス，XML Web サービスクラス，およびクライアントスタブクラスのクラス名称，ならびに構成ファイルのファイル名称を指定します。

このオプションを指定した場合と省略した場合のクラス名称およびファイル名称は次のとおりです。

クラスおよびファイル	生成されるクラス名称およびファイル名称	
	オプションを指定した場合	オプションを省略した場合
TSP 実装クラス	〈クラス名称〉Proxy	〈サービス定義名称〉Proxy
XML Web サービスクラス	〈クラス名称〉WebService	〈サービス定義名称〉WebService
クライアントスタブクラス	〈クラス名称〉Stub	〈サービス定義名称〉Stub

クラスおよびファイル	生成されるクラス名称およびファイル名称	
	オプションを指定した場合	オプションを省略した場合
構成ファイル	〈TSP 実装クラス名称〉.config	〈サービス定義名称〉Proxy.config

なお，上記の表の〈クラス名称〉部分だけが，-r オプションで指定するクラス名称です。

●-t soap

XML Web サービスクラスを生成します。

このオプションを省略した場合，XML Web サービスクラスは生成されません。

●-S {doc | rpc} ～《doc》

XML Web サービスメソッドとの間で送受信される SOAP メッセージの書式を指定します。このオプションは，-t オプションを指定した場合にだけ有効です。指定しなかった場合，-S オプションの指定は無視されます。

このオプションを省略した場合，doc が仮定されます。

オプションの指定と送受信される SOAP メッセージの書式を次に示します。

オプションの指定	送受信される SOAP メッセージの書式
doc	XML Web サービスメソッドとの間で送受信される SOAP メッセージの書式は，Document になります。
rpc	XML Web サービスメソッドとの間で送受信される SOAP メッセージの書式は，RPC になります。

●-x {literal | encoded} ～《literal》

XML Web サービスメソッドとの間で送受信される SOAP パラメタの書式指定スタイルを指定します。このオプションは，-t オプションを指定した場合かつ SOAP メッセージの書式が Document の場合にだけ有効です。-t オプションを指定しなかった場合，-x オプションの指定は無視されます。

-S オプションに rpc を指定した場合，-x オプションの指定は無視され，encoded が仮定されます。

このオプションを省略した場合，literal が仮定されます。

オプションの指定と送受信される SOAP パラメタの書式指定スタイルを次に示します。

オプションの指定	送受信される SOAP パラメタの書式指定スタイル
literal	XML Web サービスメソッドとの間で送受信される SOAP パラメタの書式指定スタイルは，Literal になります。
encoded	XML Web サービスメソッドとの間で送受信される SOAP パラメタの書式指定スタイルは，Encoded になります。

●-w　XML Web サービスで使用する XML 名前空間名称　　～〈文字列〉

XML Web サービスで使用する XML 名前空間名称を指定します。このオプションは，-t オプションを指定した場合にだけ有効です。指定しなかった場合，-w オプションの指定は無視されます。

このオプションを省略した場合の名前空間名称は，.NET Framework のデフォルト値になります。.NET Framework のデフォルト値については，.NET Framework のドキュメントを参照してください。

●-N　SOAP メッセージに関連づけられる名前空間名称　　～〈文字列〉

XML Web サービスメソッドに対する SOAP 要求，および SOAP 応答に関連づける名前空間を指定します。このオプションは，-t オプションを指定した場合にだけ有効です。指定しなかった場合，-N オプションの指定は無視されます。また，メソッドごとに名前空間名称を指定することはできません。

このオプションを省略した場合の名前空間名称は，.NET Framework のデフォルト値になります。.NET Framework のデフォルト値については，.NET Framework のドキュメントを参照してください。

●-B　{wsibp11 | none}

XML Web サービスクラスが，WS-I Basic Profile 1.1 に準拠するかどうかを宣言します。このオプションは-t オプションを指定し，かつ-S オプションに doc を指定した場合だけ有効です。これらのオプションを指定しなかった場合は，-B オプションの指定は無視されます。

wsibp11：XML Web サービスクラスが，WS-I Basic Profile 1.1 に準拠していることを宣言します。XML Web サービスクラスに，WebServiceBinding.ConformsTo プロパティの値として WsiProfiles.BasicProfile1_1 が付加されます。

none：XML Web サービスクラスが，WS-I Basic Profile 1.1 に準拠していることを宣言しません。XML Web サービスクラスに，WebServiceBinding.ConformsTo プロパティの値として WsiProfiles.None が付加されます。

●-A　{true | false}

XML Web サービスクラスの Web サービス記述言語（WSDL）ファイルで，WS-I Basic Profile 準拠の表示をするかどうかを指定します。このオプションは-B オプションを指定した場合だけ有効です。このオプションを省略した場合，WSDL ファイルで WS-I Basic Profile 準拠に関する表示はされません。

true：WSDL ファイルで，-B オプションで指定した WS-I Basic Profile 準拠の表示をします。XML Web サービスクラスに，WebServiceBinding.EmitConformanceClaims プロパティの値として true が付加されます。

false：WSDL ファイルで，-B オプションで指定した WS-I Basic Profile 準拠の表示をしません。XML Web サービスクラスに，WebServiceBinding.EmitConformanceClaims プロパティの値として false が付加されます。

●-p　Connector .NET が利用する構成定義のプロファイル ID　　～〈文字列〉

Connector .NET が使用する構成定義のプロファイル ID を指定します。

このオプションが省略された場合，デフォルトプロファイルの情報を利用して Connector .NET を使用します。

●-R　データ型定義名称:カスタムレコードクラス名称　　～〈文字列〉:〈31 文字以内の識別子〉

データ型定義ファイルで定義しているデータ型定義名称と，それを基にして生成するカスタムレコードのクラス名称を指定します。

このオプションを省略した場合，データ型定義名称がカスタムレコードのクラス名称に使用されます。

複数のデータ型定義からカスタムレコードを生成する場合は，データ型定義名称と生成されるカスタムレコードのクラス名称をコロン（:）で区切って指定します。ただし，データ型定義に存在しないデータ型定義名称を指定した場合，存在しないデータ型定義名称は無視されます。存在するデータ型定義名称に対してだけカスタムレコードクラス名称の指定が有効になります。

●-F　{space | null}　　～《space》

入力レコードとなるカスタムレコードを引数として渡す場合，データ型定義で指定した文字列領域の余った領域に埋める文字を指定します。

このオプションを省略した場合，余った領域を半角スペースで埋めます。

space：半角スペースで埋めます。

null：ヌル文字で埋めます。

●-I　エンコーディング名　　～〈文字列〉

TP1/Server への送信データをエンコードするときに従うエンコード方式のエンコーディング名を指定します。指定できるエンコーディング名については，.NET Framework のドキュメントを参照してください。

このオプションを省略した場合，プラットフォームのデフォルトエンコーディング名になります。

サポートされているエンコード方式は，プラットフォームによって異なります。代表的なエンコーディング名は次のとおりです。

エンコーディング名	エンコード（Windows 上の表示名）	備考
euc-jp	日本語（EUC）	−
iso-8859-1	西ヨーロッパ言語（ISO）	−
shift_jis	日本語（シフト JIS）	MS932
unicodeFFFE	Unicode（Big-Endian）	−
us-ascii	US-ASCII	ISO646
utf-8	Unicode（UTF-8）	−
utf-16	Unicode	Little Endian

（凡例）

－：該当しません。

●-O　エンコーディング名　　～〈文字列〉

TP1/Server から受け取った受信データをデコードするときに従うエンコード方式のエンコーディング名を指定します。指定できるエンコーディング名については，.NET Framework のドキュメントを参照してください。

このオプションを省略した場合，プラットフォームのデフォルトエンコーディング名になります。

プラットフォームによってサポートされているエンコード方式は異なります。代表的なエンコーディング名については，-I オプションの表を参照してください。

●-e　{big | little}　　～《big》

TP1/Server への送信データを，指定されたエンディアンに変換します。

このオプションを省略した場合，ビッグエンディアンに変換します。

big：ビッグエンディアンに変換します。

little：リトルエンディアンに変換します。

●-E　{big | little}　　～《big》

TP1/Server から受け取った受信データを，指定されたエンディアンであると仮定して変換します。

このオプションを省略した場合，ビッグエンディアンであると仮定して変換します。

big：ビッグエンディアンであると仮定して変換します。

little：リトルエンディアンであると仮定して変換します。

●-b

生成されたカスタムレコードクラスごとに必要なバッファサイズを標準出力に表示します。バッファプーリング機能を使用する場合で，バッファサイズの見積もりを行うときに使用します。

【表示例】

```
CustomRecordClassName : BufferSize[Bytes]
MyAppNS.Record1 : 448
MyAppNS.Record2 : 32
```

●-d

-F オプションで null を指定し，かつ-t オプションを指定した場合，XML Web サービスクラスで出力レコードの文字領域の末尾のヌル文字を削除します。文字領域の末尾の文字がヌル文字でなくなるまでヌル文字を削除します。

このオプションを指定した場合，XML Web サービスクラスでヌル文字を削除しません。-F オプションで null を指定し，かつ-t オプションを指定しない場合，このオプションの指定は無視されます。

●-g　サービスグループ名　　～〈31 文字以内の識別子〉

呼び出したい SPP.NET または SPP のサービスグループ名を指定します。

このオプションの指定値を XML Web サービス生成後に変更したい場合は，サービスグループ名を構成ファイル（Web.config または machine.config）に指定します。詳細については，「2.6.3　TSP の動作設定」を参照してください。

●-i　サービス定義ファイル名称　　～〈ファイル名〉

サービス定義が指定されているファイル名称を指定します。絶対パスまたは相対パスで指定してください。

●-h

このコマンドの使用方法を標準出力に表示します。

このオプションを指定した場合，ほかのオプションおよびコマンド引数は指定できません。

注意事項

- このコマンドの実行時にエラーが発生した場合は，対応するエラーメッセージが標準エラー出力に出力されます。
- このコマンドで生成したソースファイルの内容は変更しないでください。
- サービス定義ファイルの#include ディレクティブには，サービス定義ファイルの存在するディレクトリからの相対パスを指定します。#include ディレクティブに相対パスを指定していない場合は，データ型定義ファイルはサービス定義ファイルと同じディレクトリになければなりません。
- 入力元となるサービス定義ファイルおよびデータ型定義ファイルは，使用する Windows のデフォルトコードページ（日本語版 Windows の場合は 932）で保存してください。Unicode など，ほかのコードページで保存したファイルは，このコマンドの入力元として使用できません。
- データ型定義名称と構造体名称で同じ名称を使用したデータ型定義を利用したサービス定義から，XML Web サービスを生成した場合，XML 型名の重複によって，実行時にエラーとなります。そのため，データ型定義名称と構造体名称で同じ名称を使用しないようにしてください。一つのサービス定義が参照するデータ型定義が複数あり，その中に同じデータ型定義名称または構造体名称がある場合も，同様にエラーとなります。

cnnnidgen（MSDTC 連携機能で使用するノード識別子生成コマンド）

形式

```
cnnnidgen 〔-h〕
```

機能

Connector .NET の MSDTC 連携機能で使用するノード識別子を生成し，文字列で標準出力に表示します。

オプション

●オプションなしの場合

Connector .NET の MSDTC 連携機能で使用するノード識別子を生成し，文字列で標準出力に表示します。

●-h

このコマンドの使用方法を標準出力に表示します。

このオプションを指定した場合，ほかのオプションおよびコマンド引数は指定できません。

出力形式

●オプションなしの場合

```
aa....aa
```

- aa....aa：ノード識別子
  ノード識別子を文字列で表示します。

【表示例】

```
A78E26AE-3A1E-42fe-B0D0-8DFCD0783B55
```

cnntrsls（トランザクションリカバリサービスの状態表示コマンド）

形式

```
cnntrsls 〔-h〕
```

機能

トランザクションリカバリサービスの状態をアプリケーションドメインごとに一覧表示します。

オプション

●オプションなしの場合

トランザクションリカバリサービスの状態をアプリケーションドメインごとに一覧表示します。

●-h

このコマンドの使用方法を標準出力に表示します。

このオプションを指定した場合，ほかのオプションおよびコマンド引数は指定できません。

出力形式

●オプションなしの場合

```
started time  RMID      ApplicationDomain name
aa....aa      bb....bb  cc....cc
   :             :         :
```

- aa....aa：アプリケーションドメインの監視開始時刻

  トランザクションリカバリサービスがアプリケーションドメインの監視を開始した時刻を表示します。
  監視時刻の範囲は，西暦 0001 年 1 月 1 日の 00:00:00～9999 年 12 月 31 日の 23:59:59 です。
  例えば，西暦 0001 年 1 月 1 日の 00:00:00 の場合は次のように表示されます。

  ```
  0001/01/01 00:00:00
  ```

- bb....bb：RMID

  RMID を文字列で表示します。

- cc....cc：アプリケーションドメイン名

  監視対象のアプリケーションドメイン名を文字列で表示します。

【表示例】

```
started time        RMID                                  Application Domain Name
2007/04/01 12:34:56 A78E26AE-3A1E-42fe-B0D0-8DFCD0783B55 /LM/w3svc/1/ROOT/WebSite1-2-128190022519687500
```

6 クラスリファレンス

この章では，Connector .NET で利用できるクラスについて説明します。

Connector .NET で利用できるクラス

Connector .NET で利用できるクラスの一覧を次に示します。

表 6-1　Connector .NET で利用できるクラスの一覧

名前空間	クラス名	説明
Hitachi.OpenTP1.Connector	IndexedRecord	ArrayList 型のレコードクラスです。 TP1Connection クラスの Execute メソッドの入出力パラメタとして使用します。
	MessageBuffer	インデクスドレコードの入出力電文の電文保持領域として内部にバッファを持ちます。保持しているバッファは TP1ConnectionManager クラスが管理しているバッファプールから割り当てられたものです。
	RpcInfo	アプリケーションが OpenTP1 の SPP（SPP.NET）に対してリモートプロシジャコール（RPC）機能を実行するために必要な追加情報を TP1Connection クラスに渡すために使用します。
	TcnIllegalArgumentException	メソッド呼び出し時のパラメタに不正な引数を指定した場合にスローされる例外です。
	TcnIllegalStateException	不正または不適切なタイミングにメソッドが呼び出された場合にスローされる例外です。
	TcnNotUsedException	指定された機能が使用できないときにスローされる例外です。
	TcpipConnection	MHP または他システムと TCP/IP 通信を行う機能を提供します。
	TcpipInfo	アプリケーションが MHP または他システムに対して TCP/IP 通信機能を使用するために必要な追加情報を TcpipConnection に渡すために使用します。
	TP1Connection	OpenTP1 と対話する機能を提供します。
	TP1ConnectionManager	OpenTP1 との対話に必要なさまざまなオブジェクト（コネクション，レコードなど）を取得する機能を提供します。
	TP1ConnectorError	TP1ConnectorError の概要の説明です。
	TP1ConnectorException	Connector .NET で利用される Exception クラスのルートとなる Exception クラスです。
Hitachi.OpenTP1.ServiceModel.TP1Integration	TP1IntegrationBehavior	TP1IntegrationBinding の Endpoint の設定が正しいかどうかを検証する機能を提供します。
	TP1IntegrationBinding	OpenTP1 へのサービス要求を実現する Binding を提供します。

名前空間	クラス名	説明
	TP1RpcClient	WCF のクライアントから，TP1IntegrationBinding による OpenTP1 へのサービス要求を可能にする機能を提供する WCF のプロキシクラスです。

IndexedRecord

IndexedRecord の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- System.Collections.ArrayList
    +- Hitachi.OpenTP1.Connector.IndexedRecord
```

実装インタフェース

System.Collections.IList

System.Collections.ICollection

System.Collections.IEnumerable

System.ICloneable

Hitachi.OpenTP1.Common.IStreamable

Hitachi.OpenTP1.IRecord

説明

ArrayList 型のレコードクラスです。TP1Connection クラスの Execute メソッドの入出力パラメタとして使用します。

メソッドの一覧

名称	説明
Equals(System.Object)	このオブジェクトとほかのオブジェクトが等しいかどうかを示します。
GetHashCode()	IndexedRecord インスタンスのハッシュ値を返します。
GetRecordName()	レコード名称を取得します。
GetRecordShortDescription()	レコードの簡易説明を取得します。
SetRecordName(System.String)	レコード名称を設定します。
SetRecordShortDescription(System.String)	レコードの簡易説明を設定します。

メソッドの詳細

●Equals

説明

このオブジェクトとほかのオブジェクトが等しいかどうかを示します。

同一クラスで，かつデータが同じであるかどうかを比較します。

宣言

【C#の場合】

```
public virtual System.Boolean Equals(
  System.Object other
);
```

【Visual Basic の場合】

```
Public Overridable Function Equals( _
  ByVal other As System.Object _
) As System.Boolean
```

【J#の場合】

```
public System.Boolean Equals(
  System.Object other
);
```

【COBOL 言語の場合】

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS BASE AS  'System.Object' .
IDENTIFICATION DIVISION.
METHOD-ID. Equals PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 other USAGE IS OBJECT REFERENCE BASE.
01 RESULT USAGE IS LOGICAL.
PROCEDURE DIVISION USING BY VALUE other RETURNING RESULT.
END METHOD Equals.
```

パラメタ

other

比較対象の参照オブジェクト。

戻り値

other パラメタに指定されたオブジェクトとこのオブジェクトが等しい場合は true，等しくない場合は false が返ります。

例外

なし

●GetHashCode

説明

IndexedRecord インスタンスのハッシュ値を返します。

宣言

【C#の場合】

```
public virtual int GetHashCode(
);
```

【Visual Basic の場合】

```
Public Overridable Function GetHashCode( _
) As Integer
```

【J#の場合】

```
public int GetHashCode(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GetHashCode PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 RESULT USAGE IS BINARY-LONG.
PROCEDURE DIVISION RETURNING RESULT.
END METHOD GetHashCode.
```

パラメタ

なし

戻り値

このオブジェクトのハッシュ値を返します。

例外

なし

●GetRecordName

説明

レコード名称を取得します。

宣言

【C#の場合】

```
public virtual string GetRecordName(
);
```

【Visual Basic の場合】

```
Public Overridable Function GetRecordName( _
) As String
```

【J#の場合】

```
public System.String GetRecordName(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GetRecordName PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 RESULT USAGE IS STRING.
PROCEDURE DIVISION RETURNING RESULT.
END METHOD GetRecordName.
```

パラメタ

なし

戻り値

設定されているレコード名称が返ります。

例外

なし

●GetRecordShortDescription

説明

レコードの簡易説明を取得します。

宣言

【C#の場合】

```
public virtual string GetRecordShortDescription(
);
```

【Visual Basic の場合】

```
Public Overridable Function GetRecordShortDescription( _
) As String
```

【J#の場合】

```
public System.String GetRecordShortDescription(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GetRecordShortDescription PUBLIC.
```

```
DATA DIVISION.
LINKAGE SECTION.
01 RESULT USAGE IS STRING.
PROCEDURE DIVISION RETURNING RESULT.
END METHOD GetRecordShortDescription.
```

パラメタ

なし

戻り値

レコードに設定されている簡易説明を返します。

例外

なし

●SetRecordName

説明

レコード名称を設定します。

宣言

【C#の場合】

```
public virtual void SetRecordName(
  string recordName
);
```

【Visual Basic の場合】

```
Public Overridable Sub SetRecordName( _
  ByVal recordName As String _
)
```

【J#の場合】

```
public void SetRecordName(
  System.String recordName
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. SetRecordName PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 recordName USAGE IS STRING.
PROCEDURE DIVISION USING BY VALUE recordName.
END METHOD SetRecordName.
```

パラメタ

recordName

新たに設定するレコード名称。

戻り値

なし

例外

なし

●SetRecordShortDescription

説明

レコードの簡易説明を設定します。

宣言

【C#の場合】

```
public virtual void SetRecordShortDescription(
  string des
);
```

【Visual Basic の場合】

```
Public Overridable Sub SetRecordShortDescription( _
  ByVal des As String _
)
```

【J#の場合】

```
public void SetRecordShortDescription(
  System.String des
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. SetRecordShortDescription PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 des USAGE IS STRING.
PROCEDURE DIVISION USING BY VALUE des.
END METHOD SetRecordShortDescription.
```

パラメタ

des

新たに設定するレコードの簡易説明。

戻り値

なし

例外

なし

MessageBuffer

MessageBuffer の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- Hitachi.OpenTP1.Connector.MessageBuffer
```

説明

MessageBuffer クラスはインデクスドレコードの入出力電文の電文保持領域として内部にバッファを持ちます。保持しているバッファは TP1ConnectionManager クラスが管理しているバッファプールから割り当てられたものです。

プロパティの一覧

名称	説明
Buffer	メッセージバッファ内に保持しているバッファの参照を返します。
BufferSize	メッセージバッファに保持しているバッファのサイズを取得します。
MessageLength	メッセージバッファが保持しているメッセージ長を取得および設定します。

メソッドの一覧

名称	説明
Append(Hitachi.OpenTP1.Connector.MessageBuffer)	指定されたメッセージバッファのデータを，メッセージバッファが保持するバッファにコピーします。
Append(System.Byte[])	指定されたバイト配列のデータをメッセージバッファが保持するバッファにコピーします。
ReleaseMessageBuffer()	メッセージバッファ内に保持しているバッファを TP1ConnectionManager クラスが管理しているプールに返します。

プロパティの詳細

●Buffer

説明

メッセージバッファ内に保持しているバッファの参照を返します。

宣言

【C#の場合】

```
public virtual byte[] Buffer {get;}
```

【Visual Basic の場合】

```
Public Overridable ReadOnly Property Buffer As Byte()
```

【J#の場合】

```
public ubyte[] get_Buffer();
```

【COBOL 言語の場合】

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS BYTE-ARRAY AS  'System.Byte'  IS ARRAY.
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY Buffer IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE IS OBJECT REFERENCE BYTE-ARRAY.
PROCEDURE DIVISION RETURNING VAL.
END METHOD.
```

例外

なし

●BufferSize

説明

メッセージバッファに保持しているバッファのサイズを取得します。

宣言

【C#の場合】

```
public virtual int BufferSize {get;}
```

【Visual Basic の場合】

```
Public Overridable ReadOnly Property BufferSize As Integer
```

【J#の場合】

```
public int get_BufferSize();
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY BufferSize IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
```

```
01 VAL USAGE BINARY-LONG.
PROCEDURE DIVISION RETURNING VAL.
END METHOD.
```

例外

なし

●MessageLength

説明

メッセージバッファが保持しているメッセージ長を取得および設定します。

宣言

【C#の場合】

```
public virtual int MessageLength {get; set;}
```

【Visual Basic の場合】

```
Public Overridable Property MessageLength As Integer
```

【J#の場合】

```
public int get_MessageLength();
public void set_MessageLength(int);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY MessageLength IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE BINARY-LONG.
PROCEDURE DIVISION RETURNING VAL.
END METHOD.

IDENTIFICATION DIVISION.
METHOD-ID. SET PROPERTY MessageLength IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE BINARY-LONG.
PROCEDURE DIVISION USING BY VALUE VAL.
END METHOD.
```

例外

Hitachi.OpenTP1.Connector.TcnIllegalArgumentException

次のどれかの場合に発生します。

- バッファが null の場合
- 指定されたメッセージ長が 1 未満の場合
- バッファの長さより長い場合

メソッドの詳細

●Append

説明

指定されたメッセージバッファのデータを，メッセージバッファが保持するバッファにコピーします。

宣言

【C#の場合】

```
public virtual void Append(
  Hitachi.OpenTP1.Connector.MessageBuffer message
);
```

【Visual Basic の場合】

```
Public Overridable Sub Append( _
  ByVal message As _
  Hitachi.OpenTP1.Connector.MessageBuffer _
)
```

【J#の場合】

```
public void Append(
  Hitachi.OpenTP1.Connector.MessageBuffer message
);
```

【COBOL 言語の場合】

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS MESSAGEBUFFER AS  'Hitachi.OpenTP1.Connector.MessageBuffer'.
IDENTIFICATION DIVISION.
METHOD-ID. Append PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 message USAGE IS OBJECT REFERENCE MESSAGEBUFFER.
PROCEDURE DIVISION USING BY VALUE message.
END METHOD Append.
```

パラメタ

message

設定するメッセージ。

戻り値

なし

例外

Hitachi.OpenTP1.Connector.TcnIllegalArgumentException

電文が null です。

Hitachi.OpenTP1.Connector.TcnIllegalStateException

バッファが null の場合，または電文長がバッファのサイズを超える場合に発生します。

●Append

説明

指定されたバイト配列のデータをメッセージバッファが保持するバッファにコピーします。

宣言

【C#の場合】

```
public virtual void Append(
  byte[] message
);
```

【Visual Basic の場合】

```
Public Overridable Sub Append( _
  ByVal message() As Byte _
)
```

【J#の場合】

```
public void Append(
  ubyte[] message
);
```

【COBOL 言語の場合】

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS BYTE-ARRAY AS  'System.Byte'  IS ARRAY.
IDENTIFICATION DIVISION.
METHOD-ID. Append PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 message USAGE IS OBJECT REFERENCE BYTE-ARRAY.
PROCEDURE DIVISION USING BY VALUE message.
END METHOD Append.
```

パラメタ

message

設定するメッセージ。

戻り値

なし

例外

Hitachi.OpenTP1.Connector.TcnIllegalArgumentException

電文が null です。

Hitachi.OpenTP1.Connector.TcnIllegalStateException

バッファが null の場合，または電文長がバッファのサイズを超える場合に発生します。

●ReleaseMessageBuffer

説明

メッセージバッファ内に保持しているバッファを TP1ConnectionManager クラスが管理しているプールに返します。

宣言

【C#の場合】

```
public virtual void ReleaseMessageBuffer(
);
```

【Visual Basic の場合】

```
Public Overridable Sub ReleaseMessageBuffer( _
)
```

【J#の場合】

```
public void ReleaseMessageBuffer(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. ReleaseMessageBuffer PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
PROCEDURE DIVISION.
END METHOD ReleaseMessageBuffer.
```

パラメタ

なし

戻り値

なし

例外

なし

RpcInfo

RpcInfo の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- Hitachi.OpenTP1.Connector.RpcInfo
```

説明

アプリケーションが OpenTP1 の SPP（SPP.NET）に対してリモートプロシジャコール（RPC）機能を実行するために必要な追加情報を TP1Connection クラスに渡すために使用します。COBOL 言語からの定数値のフィールドを使用する場合は，「付録 B　Connector .NET で利用できるクラスのフィールド」を参照して，対応する値を指定してください。

コンストラクタの一覧

名称	説明
RpcInfo()	RpcInfo オブジェクトを生成します。

フィールドの一覧

名称	説明
DCNOFLAGS	RPC の形態として同期応答型 RPC を指定します。
DCRPC_CHAINED	RPC の形態として連鎖 RPC を指定します。
DCRPC_MAX_MESSAGE_SIZE	RPC 電文の最大長（1048576 バイト）です。
DCRPC_NOREPLY	RPC の形態として非応答型 RPC を指定します。
DCRPC_TPNOTRAN	トランザクションの処理からの RPC をトランザクションとしないサービス要求にできます。 RPC の形態を示す値と組み合わせて指定します。

プロパティの一覧

名称	説明
Flags	Flags プロパティは，RPC 実行時にサービス呼び出し形態を設定および取得します。
ServiceGroupName	ServiceGroupName プロパティは，RPC 実行時に呼び出すサービスグループ名を取得および設定します。
ServiceName	ServiceName プロパティは，RPC 実行時に呼び出すサービス名を取得および設定します。

名称	説明
WatchTime	WatchTime プロパティは，同期応答型 RPC の場合に SPP（SPP.NET）へサービス要求を送ってからサービスの応答が返るまでの最大応答待ち時間を設定および取得します。

コンストラクタの詳細

●RpcInfo

説明

RpcInfo オブジェクトを生成します。

宣言

【C#の場合】

```
public RpcInfo(
);
```

【Visual Basic の場合】

```
Public New( _
)
```

【J#の場合】

```
public RpcInfo(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
AUTO-METHOD. CONSTRUCTOR.
DATA DIVISION.
LINKAGE SECTION.
PROCEDURE DIVISION.
END AUTO-METHOD.
```

パラメタ

なし

例外

なし

フィールドの詳細

●DCNOFLAGS

説明

RPC の形態として同期応答型 RPC を指定します。

宣言

【C#の場合】

```
public const int DCNOFLAGS
```

【Visual Basic の場合】

```
Public Const DCNOFLAGS As Integer
```

【J#の場合】

```
public static final int DCNOFLAGS
```

●DCRPC_CHAINED

説明

RPC の形態として連鎖 RPC を指定します。

宣言

【C#の場合】

```
public const int DCRPC_CHAINED
```

【Visual Basic の場合】

```
Public Const DCRPC_CHAINED As Integer
```

【J#の場合】

```
public static final int DCRPC_CHAINED
```

●DCRPC_MAX_MESSAGE_SIZE

説明

RPC 電文の最大長（1048576 バイト）です。

宣言

【C#の場合】

```
public const int DCRPC_MAX_MESSAGE_SIZE
```

【Visual Basic の場合】

```
Public Const DCRPC_MAX_MESSAGE_SIZE As Integer
```

【J#の場合】

```
public static final int DCRPC_MAX_MESSAGE_SIZE
```

●DCRPC_NOREPLY

説明

RPC の形態として非応答型 RPC を指定します。

宣言

【C#の場合】

```
public const int DCRPC_NOREPLY
```

【Visual Basic の場合】

```
Public Const DCRPC_NOREPLY As Integer
```

【J#の場合】

```
public static final int DCRPC_NOREPLY
```

●DCRPC_TPNOTRAN

説明

トランザクションの処理からの RPC をトランザクションとしないサービス要求にできます。

RPC の形態を示す値と組み合わせて指定します。

宣言

【C#の場合】

```
public const int DCRPC_TPNOTRAN
```

【Visual Basic の場合】

```
Public Const DCRPC_TPNOTRAN As Integer
```

【J#の場合】

```
public static final int DCRPC_TPNOTRAN
```

プロパティの詳細

●Flags

説明

Flags プロパティは，RPC 実行時のサービス呼び出し形態を設定および取得します。

宣言

【C#の場合】

```
public virtual int Flags {get; set;}
```

【Visual Basic の場合】

```
Public Overridable Property Flags As Integer
```

【J#の場合】

```
public int get_Flags();
public void set_Flags(int);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY Flags IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE BINARY-LONG.
PROCEDURE DIVISION RETURNING VAL.
END METHOD.

IDENTIFICATION DIVISION.
METHOD-ID. SET PROPERTY Flags IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE BINARY-LONG.
PROCEDURE DIVISION USING BY VALUE VAL.
END METHOD.
```

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

指定された値が不正です。または，指定されたフラグの組み合わせが不正です。

●ServiceGroupName

説明

ServiceGroupName プロパティは，RPC 実行時に呼び出すサービスグループ名を取得および設定します。

宣言

【C#の場合】

```
public virtual string ServiceGroupName {get; set;}
```

【Visual Basic の場合】

```
Public Overridable Property ServiceGroupName As String
```

【J#の場合】

```
public System.String get_ServiceGroupName();
public void set_ServiceGroupName(System.String);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY ServiceGroupName IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE STRING.
PROCEDURE DIVISION RETURNING VAL.
END METHOD.

IDENTIFICATION DIVISION.
METHOD-ID. SET PROPERTY ServiceGroupName IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE STRING.
PROCEDURE DIVISION USING BY VALUE VAL.
END METHOD.
```

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

指定された引数が不正です。

値が null，""，またはサービスグループ名が 31 文字を超えています。

●ServiceName

説明

ServiceName プロパティは，RPC 実行時に呼び出すサービス名を取得および設定します。

宣言

【C#の場合】

```
public virtual string ServiceName {get; set;}
```

【Visual Basic の場合】

```
Public Overridable Property ServiceName As String
```

【J#の場合】

```
public System.String get_ServiceName();
public void set_ServiceName(System.String);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY ServiceName IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE STRING.
PROCEDURE DIVISION RETURNING VAL.
END METHOD.

IDENTIFICATION DIVISION.
METHOD-ID. SET PROPERTY ServiceName IS PUBLIC.
```

```
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE STRING.
PROCEDURE DIVISION USING BY VALUE VAL.
END METHOD.
```

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

指定された引数が不正です。

値が null，またはサービス名が 31 文字を超えています。

●WatchTime

説明

WatchTime プロパティは，同期応答型 RPC の場合に SPP（SPP.NET）へサービス要求を送ってからサービスの応答が返るまでの最大応答待ち時間を設定および取得します。

最大応答待ち時間は-1 から 65535（単位：秒）までの範囲で指定します。0 を指定した場合は，応答を受信するまで無限に待ちます。-1（初期値）を指定した場合は，対応する Client .NET 構成定義の設定に従います。

指定時間を過ぎても応答が返らない場合は，例外が発生します。

宣言

【C#の場合】

```
public virtual int WatchTime {get; set;}
```

【Visual Basic の場合】

```
Public Overridable Property WatchTime As Integer
```

【J#の場合】

```
public int get_WatchTime();
public void set_WatchTime(int);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY WatchTime IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE BINARY-LONG.
PROCEDURE DIVISION RETURNING VAL.
END METHOD.

IDENTIFICATION DIVISION.
METHOD-ID. SET PROPERTY WatchTime IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE BINARY-LONG.
```

```
PROCEDURE DIVISION USING BY VALUE VAL.
END METHOD.
```

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

指定された引数が不正です。

-1 未満か，または 65535 を超える値が指定されています。

TcnIllegalArgumentException

TcnIllegalArgumentException の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- System.Exception
    +- Hitachi.OpenTP1.TP1Exception
      +- Hitachi.OpenTP1.Connector.TP1ConnectorException
        +- Hitachi.OpenTP1.Connector.TcnIllegalArgumentException
```

実装インタフェース

System.Runtime.Serialization.ISerializable

説明

TcnIllegalArgumentException クラスは，TP1ConnectorException クラスを継承した Connector .NET 用の拡張クラスです。

TcnIllegalArgumentException クラスはメソッド呼び出し時のパラメタに不正な引数を指定した場合にスローされる例外です。

TcnIllegalStateException

TcnIllegalStateException の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- System.Exception
    +- Hitachi.OpenTP1.TP1Exception
      +- Hitachi.OpenTP1.Connector.TP1ConnectorException
        +- Hitachi.OpenTP1.Connector.TcnIllegalStateException
```

実装インタフェース

System.Runtime.Serialization.ISerializable

説明

TcnIllegalStateException クラスは，TP1ConnectorException クラスを継承した Connector .NET 用の拡張クラスです。

TcnIllegalArgumentException クラスは不正または不適切なタイミングにメソッドが呼び出された場合にスローされる例外です。

TcnNotUsedException

TcnNotUsedException の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- System.Exception
    +- Hitachi.OpenTP1.TP1Exception
      +- Hitachi.OpenTP1.Connector.TP1ConnectorException
        +- Hitachi.OpenTP1.Connector.TcnNotUsedException
```

実装インタフェース

System.Runtime.Serialization.ISerializable

説明

TcnNotUsedException クラスは，TP1ConnectorException クラスを継承した Connector .NET 用の拡張クラスです。

TcnNotUsedException クラスは，指定された機能が使用できないときにスローされる例外です。

TcpipConnection

TcpipConnection の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- Hitachi.OpenTP1.Connector.TcpipConnection
```

実装インタフェース

Hitachi.OpenTP1.Connector.ITP1Connection

System.IDisposable

説明

TcpipConnection クラスは，MHP または他システムと TCP/IP 通信を行う機能を提供します。

メソッドの一覧

名称	説明
Disconnect()	接続中の物理コネクションを解放します。
Dispose()	コネクションをコネクションプールに戻します。
Execute(Hitachi.OpenTP1.Connector.TcpipInfo , Hitachi.OpenTP1.IRecord, Hitachi.OpenTP1.IRecord)	TCP/IP 通信を実行します。

メソッドの詳細

●Disconnect

説明

接続中の物理コネクションを解放します。

宣言

【C#の場合】

```
public void Disconnect(
);
```

【Visual Basic の場合】

```
Public Sub Disconnect( _
)
```

【J#の場合】

```
public void Disconnect(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. Disconnect PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
PROCEDURE DIVISION.
END METHOD Disconnect.
```

パラメタ

なし

戻り値

なし

例外

なし

注意事項

接続中のすべての物理コネクションを解放します。

内部で保持している TP1Client オブジェクトに CloseRpc メソッドを発行します。

そのため，同じプロファイル ID で TP1Connection と併用している場合には，常設コネクションもクローズされます。

●Dispose

説明

コネクションをコネクションプールに戻します。

宣言

【C#の場合】

```
public sealed virtual void Dispose(
);
```

【Visual Basic の場合】

```
Public Overrides NotOverridable Overridable Sub Dispose( _
)
```

【J#の場合】

```
public final void Dispose(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. Dispose PUBLIC IS FINAL.
DATA DIVISION.
LINKAGE SECTION.
PROCEDURE DIVISION.
END METHOD Dispose.
```

パラメタ

なし

戻り値

なし

例外

なし

注意事項

コネクションをコネクションプールに戻します。

コネクションをコネクションプールに戻したあと，このオブジェクトに対してさまざまな処理を実行すると TP1ConnectorException がスローされます。

●Execute

説明

TCP/IP 通信を実行します。

宣言

【C#の場合】

```
public System.Boolean Execute(
  Hitachi.OpenTP1.Connector.TcpipInfo tcpipInfo,
  Hitachi.OpenTP1.IRecord input,
  Hitachi.OpenTP1.IRecord output
);
```

【Visual Basic の場合】

```
Public Function Execute( _
  ByVal tcpipInfo As Hitachi.OpenTP1.Connector.TcpipInfo, _
  ByVal input As Hitachi.OpenTP1.IRecord, _
  ByVal output As Hitachi.OpenTP1.IRecord _
) As System.Boolean
```

【J#の場合】

```
public System.Boolean Execute(
  Hitachi.OpenTP1.Connector.TcpipInfo tcpipInfo,
  Hitachi.OpenTP1.IRecord input,
  Hitachi.OpenTP1.IRecord output
);
```

【COBOL 言語の場合】

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS TCPIPINFO AS  'Hitachi.OpenTP1.Connector.TcpipInfo'.
  CLASS IRECORD AS  'Hitachi.OpenTP1.IRecord'.
IDENTIFICATION DIVISION.
METHOD-ID. Execute PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 tcpipInfo USAGE IS OBJECT REFERENCE TCPIPINFO.
01 input USAGE OBJECT REFERENCE IRECORD.
01 output USAGE OBJECT REFERENCE IRECORD.
01 output USAGE IS LOGICAL.
PROCEDURE DIVISION USING BY VALUE tcpipInfo BY VALUE input
                         BY VALUE output RETURNING output.
END METHOD Execute.
```

パラメタ

tcpipInfo

TCP/IP 通信を実行するために必要なさまざまな情報（通信形態，タイムアウト値など）を設定した TcpipInfo オブジェクトを設定します。

input

入力用の情報を格納した IRecord オブジェクトを設定します。

output

出力用の情報を格納した IRecord オブジェクトを設定します。

戻り値

TCP/IP 通信に成功した場合は true が返されます。

また，同期送受信または一方受信の場合は output に受信電文の内容を設定します。

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

次のどれかの場合に発生します。

- コネクションがすでに閉じられています。
- 指定された引数が不正です。
- TCP/IP 通信の実行に失敗しました。

注意事項

Execute メソッドは，input に指定された内容を入力電文として，また output に指定された内容を出力電文として TcpipInfo のプロパティ値に従って，TCP/IP 通信を実行します。

一方送信の場合，output の指定は無視されます。

一方受信の場合，input の指定は無視されます。

一方受信および同期送受信の場合，output に指定された IRecord オブジェクトの持つ長さが受信電文サイズと見なされます。受信電文サイズ分を受信するまで，Execute メソッドは呼び出せません。

TcpipInfo

TcpipInfo の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- Hitachi.OpenTP1.Connector.TcpipInfo
```

説明

アプリケーションが MHP または他システムに対して TCP/IP 通信機能を使用するために必要な追加情報を TcpipConnection に渡すために使用します。COBOL 言語からの定数値のフィールドを使用する場合は，「付録 B　Connector .NET で利用できるクラスのフィールド」を参照して，対応する値を指定してください。

コンストラクタの一覧

名称	説明
TcpipInfo()	TcpipInfo オブジェクトを生成します。

フィールドの一覧

名称	説明
TCPIP_RECV	TCP/IP 通信の形態として一方受信を指定します。
TCPIP_SEND	TCP/IP 通信の形態として一方送信を指定します。
TCPIP_SENDRECV	TCP/IP 通信の形態として同期送受信を指定します。

プロパティの一覧

名称	説明
Flags	Flags プロパティは，TCP/IP 通信実行時の通信形態を設定および取得します。
WatchTime	WatchTime プロパティは受信時の最大応答待ち時間を設定および取得します。

コンストラクタの詳細

●TcpipInfo

説明

TcpipInfo オブジェクトを生成します。

宣言

【C#の場合】

```
public TcpipInfo(
);
```

【Visual Basic の場合】

```
Public New( _
)
```

【J#の場合】

```
public TcpipInfo(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
AUTO-METHOD. CONSTRUCTOR.
DATA DIVISION.
LINKAGE SECTION.
PROCEDURE DIVISION.
END AUTO-METHOD.
```

パラメタ

なし

例外

なし

フィールドの詳細

●TCPIP_RECV

説明

TCP/IP 通信の形態として一方受信を指定します。

宣言

【C#の場合】

```
public const int TCPIP_RECV
```

【Visual Basic の場合】

```
Public Const TCPIP_RECV As Integer
```

【J#の場合】

```
public static final int TCPIP_RECV
```

●TCPIP_SEND

説明

TCP/IP 通信の形態として一方送信を指定します。

宣言

【C#の場合】

```
public const int TCPIP_SEND
```

【Visual Basic の場合】

```
Public Const TCPIP_SEND As Integer
```

【J#の場合】

```
public static final int TCPIP_SEND
```

●TCPIP_SENDRECV

説明

TCP/IP 通信の形態として同期送受信を指定します。

宣言

【C#の場合】

```
public const int TCPIP_SENDRECV
```

【Visual Basic の場合】

```
Public Const TCPIP_SENDRECV As Integer
```

【J#の場合】

```
public static final int TCPIP_SENDRECV
```

プロパティの詳細

●Flags

説明

Flags プロパティは，TCP/IP 通信実行時のサービス呼び出し形態を設定および取得します。

宣言

【C#の場合】

```
public virtual int Flags {get; set;}
```

【Visual Basic の場合】

```
Public Overridable Property Flags As Integer
```

【J#の場合】

```
public int get_Flags();
public void set_Flags(int);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY Flags IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE BINARY-LONG.
PROCEDURE DIVISION RETURNING VAL.
END METHOD.

IDENTIFICATION DIVISION.
METHOD-ID. SET PROPERTY Flags IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE BINARY-LONG.
PROCEDURE DIVISION USING BY VALUE VAL.
END METHOD.
```

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

指定された値が不正です。または，指定されたフラグの組み合わせが不正です。

●WatchTime

説明

WatchTime プロパティは受信時の最大応答待ち時間を設定および取得します。

-1 から 65535（単位：秒）までの範囲で指定します。0 を指定した場合は，指定された長さのメッセージを受信するまで無限に待ちます。-1（初期値）を指定した場合は 180 を適用します。

宣言

【C#の場合】

```
public virtual int WatchTime {get; set;}
```

【Visual Basic の場合】

```
Public Overridable Property WatchTime As Integer
```

【J#の場合】

```
public int get_WatchTime();
public void set_WatchTime(int);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY WatchTime IS PUBLIC.
DATA DIVISION.
```

```
LINKAGE SECTION.
01 VAL USAGE BINARY-LONG.
PROCEDURE DIVISION RETURNING VAL.
END METHOD.

IDENTIFICATION DIVISION.
METHOD-ID. SET PROPERTY WatchTime IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE BINARY-LONG.
PROCEDURE DIVISION USING BY VALUE VAL.
END METHOD.
```

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

指定された引数が不正です。

-1 未満か，または 65535 を超える値が指定されています。

TP1Connection

TP1Connection の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- Hitachi.OpenTP1.Connector.TP1Connection
```

実装インタフェース

Hitachi.OpenTP1.Connector.ITP1Connection

System.IDisposable

説明

TP1Connection クラスは OpenTP1 と対話する機能を提供します。

メソッドの一覧

名称	説明
Begin()	ローカルトランザクションの開始要求を行います。
Commit()	ローカルトランザクションのコミット要求を行います。
Dispose()	コネクションをコネクションプールに戻します。
Execute(Hitachi.OpenTP1.Connector.RpcInfo, Hitachi.OpenTP1.IRecord, Hitachi.OpenTP1.IRecord)	リモートプロシジャコール（RPC）機能を実行します。
Rollback()	ローカルトランザクションのロールバック要求を行います。

メソッドの詳細

●Begin

説明

ローカルトランザクションの開始要求を行います。

宣言

【C#の場合】

```
public void Begin(
);
```

【Visual Basic の場合】

```
Public Sub Begin( _
)
```

【J#の場合】

```
public void Begin(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. Begin PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
PROCEDURE DIVISION.
END METHOD Begin.
```

パラメタ

なし

戻り値

なし

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

ローカルトランザクションの開始に失敗しました。

または，コネクションがすでに閉じれらています。

●Commit

説明

ローカルトランザクションのコミット要求を行います。

宣言

【C#の場合】

```
public void Commit(
);
```

【Visual Basic の場合】

```
Public Sub Commit( _
)
```

【J#の場合】

```
public void Commit(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. Commit PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
PROCEDURE DIVISION.
END METHOD Commit.
```

パラメタ

なし

戻り値

なし

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

ローカルトランザクションのコミットに失敗しました。

または，コネクションがすでに閉じられています。

●Dispose

説明

コネクションをコネクションプールに戻します。

宣言

【C#の場合】

```
public sealed virtual void Dispose(
);
```

【Visual Basic の場合】

```
Public Overrides NotOverridable Overridable Sub Dispose( _
)
```

【J#の場合】

```
public final void Dispose(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. Dispose PUBLIC IS FINAL.
DATA DIVISION.
LINKAGE SECTION.
PROCEDURE DIVISION.
END METHOD Dispose.
```

パラメタ

なし

戻り値

なし

例外

なし

注意事項

コネクションをコネクションプールに戻します。

コネクションをコネクションプールに戻したあと，このオブジェクトに対してさまざまな処理を実行すると TP1ConnectorException がスローされます。

●Execute

説明

リモートプロシジャコール（RPC）機能を実行します。

宣言

【C#の場合】

```
public System.Boolean Execute(
  Hitachi.OpenTP1.Connector.RpcInfo rpcInfo,
  Hitachi.OpenTP1.IRecord input,
  Hitachi.OpenTP1.IRecord output
);
```

【Visual Basic の場合】

```
Public Function Execute( _
  ByVal rpcInfo As Hitachi.OpenTP1.Connector.RpcInfo, _
  ByVal input As Hitachi.OpenTP1.IRecord, _
  ByVal output As Hitachi.OpenTP1.IRecord _
) As System.Boolean
```

【J#の場合】

```
public System.Boolean Execute(
  Hitachi.OpenTP1.Connector.RpcInfo rpcInfo,
  Hitachi.OpenTP1.IRecord input,
  Hitachi.OpenTP1.IRecord output
);
```

【COBOL 言語の場合】

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS RPCINFO AS  'Hitachi.OpenTP1.Connector.RpcInfo'.
  CLASS IRECORD AS  'Hitachi.OpenTP1.IRecord'.
IDENTIFICATION DIVISION.
METHOD-ID. Execute PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 rpcInfo USAGE IS OBJECT REFERENCE RPCINFO.
01 input USAGE OBJECT REFERENCE IRECORD.
```

```
01 output USAGE OBJECT REFERENCE IRECORD.
01 RESULT USAGE IS LOGICAL.
PROCEDURE DIVISION USING BY VALUE rpcInfo BY VALUE input
                         BY VALUE output RETURNING RESULT.
END METHOD.
```

パラメタ

rpcInfo

リモートプロシジャコール（RPC）機能を実行するために必要なさまざまな情報（サービスグループ名，サービス名など）を設定した RpcInfo オブジェクトを設定します。

input

入力用の情報を格納した IRecord オブジェクトを設定します。

output

出力用の情報を格納した IRecord オブジェクトを設定します。

戻り値

リモートプロシジャコール（RPC）に成功した場合は true が返されます。

また，その場合は output に RPC 応答電文の内容を設定します。

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

次のどれかの場合に発生します。

- コネクションがすでに閉じられています。
- 指定された引数が不正です。
- リモートプロシジャコール（RPC）機能の実行に失敗しました。

注意事項

Execute メソッドは，input に指定された内容を入力電文として，また output に指定された内容を出力電文として rpcInfo のプロパティ値に従って，リモートプロシジャコール（RPC）機能を実行します。

●Rollback

説明

ローカルトランザクションのロールバック要求を行います。

宣言

【C#の場合】

```
public void Rollback(
);
```

【Visual Basic の場合】

```
Public Sub Rollback( _
)
```

【J#の場合】

```
public void Rollback(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. Rollback PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
PROCEDURE DIVISION.
END METHOD Rollback.
```

パラメタ

なし

戻り値

なし

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

ローカルトランザクションのロールバックに失敗しました。

または，コネクションがすでに閉じられています。

TP1ConnectionManager

TP1ConnectionManager の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- Hitachi.OpenTP1.Connector.TP1ConnectionManager
```

説明

TP1ConnectionManager クラスは，OpenTP1 との対話に必要なさまざまなオブジェクト（コネクション，レコードなど）を取得する機能を提供します。

コンストラクタの一覧

名称	説明
TP1ConnectionManager()	TP1ConnectionManager クラスのインスタンスを初期化します。
TP1ConnectionManager(System.String)	TP1ConnectionManager クラスのインスタンスを初期化します。

プロパティの一覧

名称	説明
LogWriter	ログの出力先になる TextWriter オブジェクトを設定および取得します。
ProfileID	このオブジェクトに設定されているプロファイル ID を取得します。

メソッドの一覧

名称	説明
CreateIndexedRecord(System.String)	リモートプロシジャコール（RPC）機能を実行する場合に，入力電文および出力電文を格納するための IndexedRecord オブジェクトを取得します。
GetConnection()	OpenTP1 にアクセスするための Connection オブジェクトを取得します。
GetMessageBuffer(System.Int32)	インデクスドレコードを使用してリモートプロシジャコール（RPC）機能を実行する場合に，利用するバッファをプールから取得します。
GetTcpipConnection()	TCP/IP 通信を実行するための TcpipConnection オブジェクトを取得します。

コンストラクタの詳細

●TP1ConnectionManager

説明

TP1ConnectionManager クラスのインスタンスを初期化します。

宣言

【C#の場合】

```
public TP1ConnectionManager(
);
```

【Visual Basic の場合】

```
Public New( _
)
```

【J#の場合】

```
public TP1ConnectionManager(
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
AUTO-METHOD. CONSTRUCTOR.
DATA DIVISION.
LINKAGE SECTION.
PROCEDURE DIVISION.
END AUTO-METHOD.
```

パラメタ

なし

例外

なし

注意事項

TP1ConnectionManager クラスのインスタンスを初期化します。

初期化時にこのオブジェクトから生成される各オブジェクトは構成ファイルに記述された<common>要素内の情報を利用します。

構成ファイルの<common>要素内に情報が存在しない場合はデフォルト値を利用します。

●TP1ConnectionManager

説明

TP1ConnectionManager クラスのインスタンスを初期化します。

宣言

【C#の場合】

```
public TP1ConnectionManager(
  string profileId
);
```

【Visual Basic の場合】

```
Public New( _
  ByVal profileId As String _
)
```

【J#の場合】

```
public TP1ConnectionManager(
  System.String profileId
);
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
AUTO-METHOD. CONSTRUCTOR.
DATA DIVISION.
LINKAGE SECTION.
01 profileId USAGE STRING.
PROCEDURE DIVISION USING BY VALUE profileId.
END AUTO-METHOD.
```

パラメタ

profileId

このオブジェクトに割り当てる構成ファイルのプロファイル ID を設定します。

""が設定された場合は<common>要素のプロファイル情報を使用します。

null が設定された場合は引数を持たないコンストラクタと同様の動作をします。

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

構成ファイルに存在しないプロファイル ID が設定されました。

注意事項

TP1ConnectionManager クラスのインスタンスを初期化します。

構成ファイルに記述されたプロファイルの<profile>要素内の情報を使用します。

プロファイル ID として""が設定された場合は<common>要素のプロファイル情報を使用します。

null が設定された場合は引数を持たないコンストラクタと同様の動作をします。

プロパティの詳細

●LogWriter

説明

ログの出力先になる TextWriter オブジェクトを設定および取得します。

宣言

【C#の場合】

```
public System.IO.TextWriter LogWriter {get; set;}
```

【Visual Basic の場合】

```
Public Property LogWriter As System.IO.TextWriter
```

【J#の場合】

```
public System.IO.TextWriter get_LogWriter();
public void set_LogWriter(System.IO.TextWriter);
```

【COBOL 言語の場合】

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS TEXTWRITER AS  'System.IO.TextWriter'.
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY LogWriter IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE IS OBJECT REFERENCE TEXTWRITER.
PROCEDURE DIVISION RETURNING VAL.
END METHOD.

IDENTIFICATION DIVISION.
METHOD-ID. SET PROPERTY LogWriter IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE IS OBJECT REFERENCE TEXTWRITER.
PROCEDURE DIVISION USING BY VALUE VAL.
END METHOD.
```

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

ログの出力先として null が設定されました。

注意事項

このプロパティを設定した場合，Connector .NET は設定された TextWriter オブジェクトに対してログ出力処理ごとには Flush 処理をしません。そのため，TextWriter オブジェクトの用途や TextWriter オブジェクトの種類に応じて，Flush 処理や Close 処理を適切に行ってください。

●ProfileID

説明

このオブジェクトに設定されているプロファイル ID を取得します。

宣言

【C#の場合】

```
public string ProfileID {get;}
```

【Visual Basic の場合】

```
Public ReadOnly Property ProfileID As String
```

【J#の場合】

```
public System.String get_ProfileID();
```

【COBOL 言語の場合】

```
IDENTIFICATION DIVISION.
METHOD-ID. GET PROPERTY ProfileID IS PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 VAL USAGE STRING.
PROCEDURE DIVISION RETURNING VAL.
END METHOD.
```

例外

なし

メソッドの詳細

●CreateIndexedRecord

説明

リモートプロシジャコール（RPC）機能を実行する場合に，入力電文および出力電文を格納するための IndexedRecord オブジェクトを取得します。

宣言

【C#の場合】

```
public Hitachi.OpenTP1.Connector.IndexedRecord
 CreateIndexedRecord(
  string recordName
);
```

【Visual Basic の場合】

```
Public Function CreateIndexedRecord( _
  ByVal recordName As String _
) As Hitachi.OpenTP1.Connector.IndexedRecord
```

【J#の場合】

```
public Hitachi.OpenTP1.Connector.IndexedRecord
 CreateIndexedRecord(
  System.String recordName
);
```

【COBOL 言語の場合】

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS INDEXEDRECORD AS  'Hitachi.OpenTP1.Connector.IndexedRecord'.
IDENTIFICATION DIVISION.
METHOD-ID. CreateIndexedRecord PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 recordName USAGE IS STRING.
01 RESULT USAGE IS OBJECT REFERENCE INDEXEDRECORD.
PROCEDURE DIVISION USING BY VALUE recordName RETURNING RESULT.
END METHOD CreateIndexedRecord.
```

パラメタ

recordName

IndexedRecord オブジェクトのレコード名を設定します。

戻り値

指定されたレコード名が設定された IndexedRecord オブジェクトを返します。

例外

なし

注意事項

リモートプロシジャコール（RPC）機能を実行する場合に，入力電文および出力電文を格納するための IndexedRecord オブジェクトを取得します。

●GetConnection

説明

OpenTP1 にアクセスするための Connection オブジェクトを取得します。

コネクションプールに存在している場合は，プールから取り出します。

存在しない場合は，新規に Connection オブジェクトを生成します。

宣言

【C#の場合】

```
public Hitachi.OpenTP1.Connector.TP1Connection
 GetConnection(
);
```

【Visual Basic の場合】

```
Public Function GetConnection( _
) As Hitachi.OpenTP1.Connector.TP1Connection
```

【J#の場合】

```
public Hitachi.OpenTP1.Connector.TP1Connection
 GetConnection(
);
```

【COBOL 言語の場合】

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS TP1CONNECTION AS  'Hitachi.OpenTP1.Connector.TP1Connection'.
IDENTIFICATION DIVISION.
METHOD-ID. GetConnection PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 RESULT USAGE IS OBJECT REFERENCE TP1CONNECTION.
PROCEDURE DIVISION RETURNING RESULT.
END METHOD GetConnection.
```

パラメタ

なし

戻り値

OpenTP1 と対話を行うためのコネクションが返されます。

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

コネクションの取得に失敗しました。

注意事項

取得した TP1Connection オブジェクトを利用して OpenTP1 との対話を行えます。

●GetMessageBuffer

説明

インデクスドレコードを使用してリモートプロシジャコール（RPC）機能を実行する場合に，利用するバッファをプールから取得します。

宣言

【C#の場合】

```
public Hitachi.OpenTP1.Connector.MessageBuffer
 GetMessageBuffer(
  int buffersize
);
```

【Visual Basic の場合】

```
Public Function GetMessageBuffer( _
  ByVal buffersize As Integer _
) As Hitachi.OpenTP1.Connector.MessageBuffer
```

【J#の場合】

```
public Hitachi.OpenTP1.Connector.MessageBuffer
 GetMessageBuffer(
  int buffersize
);
```

【COBOL 言語の場合】

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS MESSAGEBUFFER AS  'Hitachi.OpenTP1.Connector.MessageBuffer'.
IDENTIFICATION DIVISION.
METHOD-ID. GetMessageBuffer PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 buffersize USAGE IS BINARY-LONG.
01 RESULT USAGE IS OBJECT REFERENCE MESSAGEBUFFER.
PROCEDURE DIVISION USING BY VALUE buffersize RETURNING RESULT.
END METHOD GetMessageBuffer.
```

パラメタ

buffersize

利用するバッファのサイズを設定します。

戻り値

設定されたサイズより大きく，かつ最も近いサイズのバッファを返します。

例外

Hitachi.OpenTP1.Connector.TcnIllegalArgumentException

指定されたバッファ長が 0 以下であるか，または最大長を超えています。

Hitachi.OpenTP1.Connector.TcnNotUsedException

構成ファイルの設定でメッセージバッファの利用が無効です（<buffer>要素の pooling 属性が true ではありません）。

注意事項

インデクスドレコードを使用してリモートプロシジャコール（RPC）機能を実行する場合に，利用するバッファをプールから取得します。

バッファを取得する場合に，バッファのサイズを設定します。

設定されたサイズより大きく，かつ最も近いサイズのバッファを返します。

●GetTcpipConnection

説明

TCP/IP 通信を実行するための TcpipConnection オブジェクトを取得します。

コネクションプールに存在している場合はプールから取り出します。

存在しない場合は，新規に TcpipConnection オブジェクトを生成します。

宣言

【C#の場合】

```
public Hitachi.OpenTP1.Connector.TP1Connection
 GetTcpipConnection(
);
```

【Visual Basic の場合】

```
Public Function GetTcpipConnection( _
) As Hitachi.OpenTP1.Connector.TP1Connection
```

【J#の場合】

```
public Hitachi.OpenTP1.Connector.TP1Connection
 GetTcpipConnection(
);
```

【COBOL 言語の場合】

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS TCPIPCONNECTION AS  'Hitachi.OpenTP1.Connector. TcpipConnection'.
IDENTIFICATION DIVISION.
METHOD-ID. GetTcpipConnection PUBLIC.
DATA DIVISION.
LINKAGE SECTION.
01 RESULT USAGE IS OBJECT REFERENCE TCPIPCONNECTION.
PROCEDURE DIVISION RETURNING RESULT.
END METHOD GetTcpipConnection.
```

パラメタ

なし

戻り値

TCP/IP 通信を実行するためのコネクションが返されます。

例外

Hitachi.OpenTP1.Connector.TP1ConnectorException

コネクションの取得に失敗しました。

注意事項

取得した TcpipConnection オブジェクトを利用して TCP/IP 通信を実行できます。

TP1ConnectorError

TP1ConnectorError の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- Hitachi.OpenTP1.Connector.TP1ConnectorError
```

説明

TP1ConnectorError の概要の説明です。COBOL 言語からの定数値のフィールドを使用する場合は，「付録 B　Connector .NET で利用できるクラスのフィールド」を参照して，対応する値を指定してください。

フィールドの一覧

名称	説明
DCCNNER_BEGIN	Begin での例外です。
DCCNNER_BUF_STAT	バッファの状態が不正です。
DCCNNER_CALL	Call での例外です。
DCCNNER_CAST	クラスのキャストが不正です。
DCCNNER_COMMIT	Commit での例外です。
DCCNNER_CONF	構成ファイルが不正です。
DCCNNER_DISTRIBUTED_TRN	分散トランザクション処理の例外です。
DCCNNER_INVALID_ARGS	引数が不正です。
DCCNNER_INVALID_CONN	コネクションが不正です。
DCCNNER_INVALID_CUSTOM_RECORD	カスタムレコードが不正です。
DCCNNER_IOERR	入出力エラーです。
DCCNNER_NOT_USE_BUF	バッファプーリング機能を利用していない例外です。
DCCNNER_OBJECT_NOT_DEFINED	電文格納データが不正です。
DCCNNER_OPEN	Open での例外です。
DCCNNER_PROTO	プロトコルが不正です。
DCCNNER_RECORD_CREATE	カスタムレコード生成時に発生したエラーです。
DCCNNER_RECV	Receive での例外です。
DCCNNER_ROLLBACK	Rollback での例外です。

名称	説明
DCCNNER_SEND	Send での例外です。
DCCNNER_TRN	トランザクション処理の例外です。
DCCNNER_UNMATCH_LENGTH	データ長が不正です。

フィールドの詳細

●DCCNNER_BEGIN

説明

Begin での例外です。

宣言

【C#の場合】

```
public const int DCCNNER_BEGIN
```

【Visual Basic の場合】

```
Public Const DCCNNER_BEGIN As Integer
```

【J#の場合】

```
public static final int DCCNNER_BEGIN
```

●DCCNNER_BUF_STAT

説明

バッファの状態が不正です。

宣言

【C#の場合】

```
public const int DCCNNER_BUF_STAT
```

【Visual Basic の場合】

```
Public Const DCCNNER_BUF_STAT As Integer
```

【J#の場合】

```
public static final int DCCNNER_BUF_STAT
```

●DCCNNER_CALL

説明

Call での例外です。

宣言

【C#の場合】

```
public const int DCCNNER_CALL
```

【Visual Basic の場合】

```
Public Const DCCNNER_CALL As Integer
```

【J#の場合】

```
public static final int DCCNNER_CALL
```

●DCCNNER_CAST

説明

クラスのキャストが不正です。

宣言

【C#の場合】

```
public const int DCCNNER_CAST
```

【Visual Basic の場合】

```
Public Const DCCNNER_CAST As Integer
```

【J#の場合】

```
public static final int DCCNNER_CAST
```

●DCCNNER_COMMIT

説明

Commit での例外です。

宣言

【C#の場合】

```
public const int DCCNNER_COMMIT
```

【Visual Basic の場合】

```
Public Const DCCNNER_COMMIT As Integer
```

【J#の場合】

```
public static final int DCCNNER_COMMIT
```

●DCCNNER_CONF

説明

構成ファイルが不正です。

宣言

【C#の場合】

```
public const int DCCNNER_CONF
```

【Visual Basic の場合】

```
Public Const DCCNNER_CONF As Integer
```

【J#の場合】

```
public static final int DCCNNER_CONF
```

●DCCNNER_INVALID_ARGS

説明

引数が不正です。

宣言

【C#の場合】

```
public const int DCCNNER_INVALID_ARGS
```

【Visual Basic の場合】

```
Public Const DCCNNER_INVALID_ARGS As Integer
```

【J#の場合】

```
public static final int DCCNNER_INVALID_ARGS
```

●DCCNNER_INVALID_CONN

説明

コネクションが不正です。

宣言

【C#の場合】

```
public const int DCCNNER_INVALID_CONN
```

【Visual Basic の場合】

```
Public Const DCCNNER_INVALID_CONN As Integer
```

【J#の場合】

```
public static final int DCCNNER_INVALID_CONN
```

●DCCNNER_DISTRIBUTED_TRN

説明

分散トランザクション処理の例外です。

宣言

【C#の場合】

```
public const int DCCNNER_DISTRIBUTED_TRN
```

【Visual Basic の場合】

```
Public Const DCCNNER_DISTRIBUTED_TRN As Integer
```

【J#の場合】

```
public static final int DCCNNER_DISTRIBUTED_TRN
```

●DCCNNER_INVALID_CUSTOM_RECORD

説明

カスタムレコードが不正です。

宣言

【C#の場合】

```
public const int DCCNNER_INVALID_CUSTOM_RECORD
```

【Visual Basic の場合】

```
Public Const DCCNNER_INVALID_CUSTOM_RECORD As Integer
```

【J#の場合】

```
public static final int DCCNNER_INVALID_CUSTOM_RECORD
```

●DCCNNER_IOERR

説明

入出力エラーです。

宣言

【C#の場合】

```
public const int DCCNNER_IOERR
```

【Visual Basic の場合】

```
Public Const DCCNNER_IOERR As Integer
```

【J#の場合】

```
public static final int DCCNNER_IOERR
```

●DCCNNER_NOT_USE_BUF

説明

バッファプーリング機能を利用していない例外です。

宣言

【C#の場合】

```
public const int DCCNNER_NOT_USE_BUF
```

【Visual Basic の場合】

```
Public Const DCCNNER_NOT_USE_BUF As Integer
```

【J#の場合】

```
public static final int DCCNNER_NOT_USE_BUF
```

●DCCNNER_OBJECT_NOT_DEFINED

説明

電文格納データが不正です。

宣言

【C#の場合】

```
public const int DCCNNER_OBJECT_NOT_DEFINED
```

【Visual Basic の場合】

```
Public Const DCCNNER_OBJECT_NOT_DEFINED As Integer
```

【J#の場合】

```
public static final int DCCNNER_OBJECT_NOT_DEFINED
```

●DCCNNER_OPEN

説明

Open での例外です。

宣言

【C#の場合】

```
public const int DCCNNER_OPEN
```

【Visual Basic の場合】

```
Public Const DCCNNER_OPEN As Integer
```

【J#の場合】

```
public static final int DCCNNER_OPEN
```

●DCCNNER_PROTO

説明

プロトコルが不正です。

宣言

【C#の場合】

```
public const int DCCNNER_PROTO
```

【Visual Basic の場合】

```
Public Const DCCNNER_PROTO As Integer
```

【J#の場合】

```
public static final int DCCNNER_PROTO
```

●DCCNNER_RECORD_CREATE

説明

カスタムレコード生成時に発生したエラーです。

宣言

【C#の場合】

```
public const int DCCNNER_RECORD_CREATE
```

【Visual Basic の場合】

```
Public Const DCCNNER_RECORD_CREATE As Integer
```

【J#の場合】

```
public static final int DCCNNER_RECORD_CREATE
```

●DCCNNER_RECV

説明

Receive での例外です。

宣言

【C#の場合】

```
public const int DCCNNER_RECV
```

【Visual Basic の場合】

```
Public Const DCCNNER_RECV As Integer
```

【J#の場合】

```
public static final int DCCNNER_RECV
```

●DCCNNER_ROLLBACK

説明

Rollback での例外です。

宣言

【C#の場合】

```
public const int DCCNNER_ROLLBACK
```

【Visual Basic の場合】

```
Public Const DCCNNER_ROLLBACK As Integer
```

【J#の場合】

```
public static final int DCCNNER_ROLLBACK
```

●DCCNNER_SEND

説明

Send での例外です。

宣言

【C#の場合】

```
public const int DCCNNER_SEND
```

【Visual Basic の場合】

```
Public Const DCCNNER_SEND As Integer
```

【J#の場合】

```
public static final int DCCNNER_SEND
```

●DCCNNER_TRN

説明

トランザクション処理の例外です。

宣言

【C#の場合】

```
public const int DCCNNER_TRN
```

【Visual Basic の場合】

```
Public Const DCCNNER_TRN As Integer
```

【J#の場合】

```
public static final int DCCNNER_TRN
```

●DCCNNER_UNMATCH_LENGTH

説明

データ長が不正です。

宣言

【C#の場合】

```
public const int DCCNNER_UNMATCH_LENGTH
```

【Visual Basic の場合】

```
Public Const DCCNNER_UNMATCH_LENGTH As Integer
```

【J#の場合】

```
public static final int DCCNNER_UNMATCH_LENGTH
```

TP1ConnectorException

TP1ConnectorException の概要

名前空間

Hitachi.OpenTP1.Connector

継承関係

```
System.Object
  +- System.Exception
    +- Hitachi.OpenTP1.TP1Exception
      +- Hitachi.OpenTP1.Connector.TP1ConnectorException
```

実装インタフェース

System.Runtime.Serialization.ISerializable

説明

Connector .NET で利用される Exception クラスのルートとなる Exception クラスです。

TP1Exception を継承します。

TP1IntegrationBehavior

TP1IntegrationBehavior の概要

名前空間

Hitachi.OpenTP1.ServiceModel.TP1Integration

継承関係

```
System.Object
  +- Hitachi.OpenTP1.ServiceModel.TP1Integration.TP1IntegrationBehavior
```

実装インタフェース

System.ServiceModel.Description.IEndpointBehavior

説明

TP1IntegrationBinding の Endpoint の設定が正しいかどうかを検証する機能を提供します。

コンストラクタの一覧

名称	説明
TP1IntegrationBehavior()	TP1IntegrationBehavior オブジェクトを生成します。

コンストラクタの詳細

●TP1IntegrationBehavior

説明

TP1IntegrationBehavior オブジェクトを生成します。

宣言

【C#の場合】

```
public TP1IntegrationBehavior(
);
```

【Visual Basic の場合】

```
Public New( _
)
```

パラメタ

なし

例外

なし

TP1IntegrationBinding

TP1IntegrationBinding の概要

名前空間

Hitachi.OpenTP1.ServiceModel.TP1Integration

継承関係

```
System.Object
  +- System.ServiceModel.Channels.Binding
    +- Hitachi.OpenTP1.ServiceModel.TP1Integration.TP1IntegrationBinding
```

実装インタフェース

System.ServiceModel.IDefaultCommunicationTimeouts

説明

OpenTP1 へのサービス要求を実現する Binding を提供します。

コンストラクタの一覧

名称	説明
TP1IntegrationBinding()	TP1IntegrationBinding オブジェクトを生成します。

コンストラクタの詳細

●TP1IntegrationBinding

説明

TP1IntegrationBinding オブジェクトを生成します。

宣言

【C#の場合】

```
public TP1IntegrationBinding(
);
```

【Visual Basic の場合】

```
Public New( _
)
```

パラメタ

なし

例外

なし

TP1RpcClient

TP1RpcClient の概要

名前空間

Hitachi.OpenTP1.ServiceModel.TP1Integration

継承関係

```
System.Object
  +- System.ServiceModel.ClientBase
```

実装インタフェース：

```
System.ServiceModel.ICommunicationObject
System.IDisposable
Hitachi.OpenTP1.ServiceModel.TP1Integration.ITP1Rpc
```

説明

WCF のクライアントから，TP1IntegrationBinding による OpenTP1 へのサービス要求を可能にする機能を提供する WCF のプロキシクラスです。

コンストラクタの一覧

名称	説明
TP1RpcClient()	TP1RpcClient オブジェクトを生成します。
TP1RpcClient(System.String)	
TP1RpcClient(System.String, System.String)	
TP1RpcClient(System.String, System.ServiceModel.EndpointAddress)	
TP1RpcClient(System.ServiceModel.Channels.Binding, System.ServiceModel.EndpointAddress)	

メソッドの一覧

名称	説明
Call(System.String, System.Byte[], System.Int32, System.Byte[], System.Int32&)	TP1IntegrationBinding による OpenTP1 へのサービス要求を実行します。

コンストラクタの詳細

●TP1RpcClient

説明

TP1RpcClient オブジェクトを生成します。

宣言

【C#の場合】

```
public TP1RpcClient(
);
```

【Visual Basic の場合】

```
Public New( _
)
```

パラメタ

なし

例外

System.InvalidOperationException

既定<endpoint>要素が存在しないか，複数の<endpoint>要素が存在します。

または，アプリケーション構成ファイルが存在しません。

●TP1RpcClient

説明

TP1RpcClient オブジェクトを生成します。

宣言

【C#の場合】

```
public TP1RpcClient(string endpointConfigurationName
);
```

【Visual Basic の場合】

```
Public New(ByVal endpointConfigurationName As String _
)
```

パラメタ

endpointConfigurationName

アプリケーション構成ファイルの<endpoint>要素の名称

例外

System.ArgumentException

指定された<endpoint>要素の address 属性値が不正です。

System.ArgumentNullException

<endpoint>要素の名称に null が設定されました。

System.InvalidOperationException

指定された<endpoint>要素の名称が見つかりませんでした。

または，コントラクトが不正です。

●TP1RpcClient

説明

TP1RpcClient オブジェクトを生成します。

宣言

【C#の場合】

```
public TP1RpcClient(
string endpointConfigurationName,
string remoteAddress
);
```

【Visual Basic の場合】

```
Public New( _
ByVal endpointConfigurationName As String, _
ByVal remoteAddress As String _
)
```

パラメタ

endpointConfigurationName

アプリケーション構成ファイルの<endpoint>要素の名称

remoteAddress

接続先の OpenTP1 の URI

例外

System.ArgumentNullException

<endpoint>要素の名称に null が設定されました。

または，接続先の OpenTP1 の URI に null が設定されました。

System.InvalidOperationException

指定された<endpoint>要素の名称が見つかりませんでした。

または，コントラクトが不正です。

System.UriFormatException

指定された接続先の OpenTP1 の URI が不正です。

●TP1RpcClient

説明

TP1RpcClient オブジェクトを生成します。

宣言

【C#の場合】

```
public TP1RpcClient(
string endpointConfigurationName,
System.ServiceModel.EndpointAddress remoteAddress
);
```

【Visual Basic の場合】

```
Public New( _
ByVal endpointConfigurationName As String, _
ByVal remoteAddress As System.ServiceModel.EndpointAddress _
)
```

パラメタ

endpointConfigurationName

アプリケーション構成ファイルの<endpoint>要素の名称

remoteAddress

接続先の OpenTP1 の位置情報が設定された EndpointAddress オブジェクト

例外

System.ArgumentNullException

<endpoint>要素の名称に null が設定されました。

または，EndpointAddress に null が設定されました。

System.InvalidOperationException

指定された<endpoint>要素の名称が見つかりませんでした。

または，コントラクトが不正です。

●TP1RpcClient

説明

TP1RpcClient オブジェクトを生成します。

宣言

【C#の場合】

```
public TP1RpcClient(
System.ServiceModel.Channels.Binding binding,
```

```
System.ServiceModel.EndpointAddress remoteAddress
);
```

【Visual Basic の場合】

```
Public New( _
ByVal binding As System.ServiceModel.Channels.Binding, _
ByVal remoteAddress As System.ServiceModel.EndpointAddress _
)
```

パラメタ

binding

TP1IntegrationBinding オブジェクト

remoteAddress

接続先の OpenTP1 の位置情報が設定された EndpointAddress オブジェクト

例外

System.ArgumentNullException

TP1IntegrationBinding に null が設定されました。

または，EndpointAddress に null が設定されました。

メソッドの詳細

●Call

説明

OpenTP1 へのサービス要求を行います。RPC の形態は，同期応答型 RPC です。また，トランザクションには参加できません。

宣言

【C#の場合】

```
public sealed virtual void Call(
string service,
byte[] inData,
int inLength,
ref byte[] outData,
ref int outLength
);
```

【Visual Basic の場合】

```
Public Overrides NotOverridable Overridable Sub Call( _
ByVal service As String, _
ByVal inData() As Byte, _
ByVal inLength As Integer, _
ByRef outData() As Byte, _
ByRef outLength As Integer _
)
```

パラメタ

service

サービス名称

inData

サービスの入力パラメタ

inLength

サービスの入力パラメタ長を指定します。

1 から 1048576 までの範囲が指定できます。ただし，RPC 送受信メッセージの最大長拡張機能を使用した場合は，Connector .NET 構成定義の<option>要素の maxMessageSize 属性値×1048576 までの範囲が指定できます。

outData

サービスの応答の領域を指定します。

outLength

サービスの応答の長さを指定します。

1 から 1048576 までの範囲が指定できます。ただし，RPC 送受信メッセージの最大長拡張機能を使用した場合は，Connector .NET 構成定義の<option>要素の maxMessageSize 属性値×1048576 までの範囲が指定できます。

指定した値は，サービス側で使用する応答領域の長さとして使用されます。

サービス要求終了時，SPP.NET のサービスメソッド，または SPP のサービス関数で指定した応答の長さが outLength に設定されます。

戻り値

なし

例外

System.ServiceModel.CommunicationException

OpenTP1 へのサービス要求に失敗しました。

次のどちらかの場合に発生します。

- 指定された引数が不正です。
- リモートプロシジャコール（RPC）機能の実行に失敗しました。

System.ArgumentException

接続先の OpenTP1 の URI が不正です。

System.ArgumentNullException

接続先の OpenTP1 の情報を格納したオブジェクトが不正です。

System.InvalidOperationException

指定された<endpoint>要素の address 属性値が不正です。

注意事項

Call メソッドは，指定された引数，およびアプリケーション構成ファイル，または System.ServiceModel.EndpointAddress に設定する接続先の OpenTP1 の URI に従って，リモートプロシジャコール（RPC）機能を実行します。

7 障害運用

この章では，Connector .NET で障害が発生した場合の対処方法について説明します。

7.1　障害の種類と対処方法

Connector .NET で発生するおそれがある障害の種類とユーザが取る処置を次の表に示します。

表 7-1　障害の種類とユーザの取る処置

障害の種類	障害の内容	Connector .NET の処理	ユーザが取る処置
通信障害	・サーバ障害 ・ネットワーク障害 など	1. 障害発生時の情報を Connector .NET ログファイルに出力します。 2. 例外を UAP に通知します。	障害原因を調査して取り除くか，または障害回復を待ち，再度 UAP 処理を実行してください。 必要に応じて構成定義の内容，または TP1/Server 側の定義の内容を見直してください。
サーバアプリケーション障害※1	SPP.NET での例外発生（ユーザ例外）	TP1UserException を UAP に通知します。	SPP.NET の処理，入力データなどを見直してください。
	SPP.NET での例外発生（ユーザ例外以外）	TP1RemoteException を UAP に通知します。	SPP.NET の処理，実行環境などを見直してください。
データ変換障害	クライアントスタブでのデータ変換エラー	TP1MarshalException を UAP に通知します。	使用している.NET インタフェース定義の内容を見直してください。または，サービス定義およびデータ型定義の内容を見直してください。
クライアント環境障害	・ファイル障害 ・メモリ不足 など	1. 障害発生時の情報を Connector .NET ログファイルに出力します。※2 2. 例外を UAP に通知します。	障害の要因を取り除いてください。

注※1

サーバアプリケーションが.NET インタフェース定義を使用した SPP.NET の場合にだけ発生します。それ以外の場合は通信障害として通知されます。

注※2

Connector .NET ログファイルへの出力で障害が発生した場合は，ファイルへの出力のエラーを無視して処理を続行します。以降，ログは出力されません。

7.2　障害時に取得する情報

障害が発生した場合は，次に示す情報を取得してください。また，必要に応じて，取得した情報をシステム管理者に提供してください。

例外が発生した場合

　例外の情報を ToString メソッドで取得して，ファイルに出力するか画面に表示してください。ログファイルもあわせて取得してください。

コマンドがエラーとなった場合

　コマンドが出力したエラーメッセージ，コマンドの入力形式，入力ファイルなどを保存してください。

ログファイルを取得している場合

　ログファイルを保存してください。

付録

付録 A　DCCM3 と接続する場合の注意事項

Connector .NET では，RPC や TCP/IP 通信機能を使用して，VOS3 や VOS1 上の DCCM3 と接続できます。実際の通信は，Connector .NET の前提製品である Client .NET が行います。通信時の前提条件などについては，マニュアル「TP1/Client for .NET Framework 使用の手引」を参照してください。

Connector .NET を使用して DCCM3 と接続する場合の注意事項を次に示します。

RPC を使用する場合

- リモート API 機能を使用した RPC だけ使用できます。
- 連鎖 RPC は使用できません。
- .NET インタフェース定義を使用した RPC は使用できません。
- サービス定義を使用した RPC は，次の条件をすべて満たしている場合だけ使用できます。
  - データ型定義によってデータの形式をメッセージの単位に定義できる。
  - 文字列を含まないデータ，または.NET Framework がサポートしているエンコード方式でエンコードおよびデコードできる文字列のデータを送受信する。

    なお，.NET Framework がサポートしているエンコード方式については，.NET Framework のドキュメントを参照してください。
- トランザクション制御機能は使用できません。
- 文字列のデータを含むメッセージを送受信する場合，あらかじめメッセージ中の文字コードを通信先システムと決めた上で，必要に応じて UAP で文字コード変換を行ってください。.NET Framework 上では，Unicode（リトルエンディアン）が文字列データのメモリ上の内部表現として使用されます。
- マニュアル「TP1/Client for .NET Framework 使用の手引」の注意事項もあわせて参照してください。

TCP/IP 通信機能を使用する場合

- 文字列のデータを含むメッセージを送受信する場合，あらかじめメッセージ中の文字コードを通信先システムと決めた上で，必要に応じて UAP で文字コード変換を行ってください。.NET Framework 上では，Unicode（リトルエンディアン）が文字列データのメモリ上の内部表現として使用されます。
- マニュアル「TP1/Client for .NET Framework 使用の手引」の注意事項もあわせて参照してください。

付録 B　Connector .NET で利用できるクラスのフィールド

Connector .NET で利用できる各クラスのフィールドを次の表に示します。COBOL 言語から定数値のフィールドを使用する場合は，対応する値を指定してください。

表 B-1　Connector .NET で利用できるクラスのフィールド

クラス名	フィールド名	値
Hitachi.OpenTP1.Connector.RpcInfo	DCNOFLAGS	0
	DCRPC_CHAINED	4
	DCRPC_MAX_MESSAGE_SIZE	1048576
	DCRPC_NOREPLY	1
	DCRPC_TPNOTRAN	32
Hitachi.OpenTP1.Connector.TcpipInfo	TCPIP_RECV	256
	TCPIP_SEND	128
	TCPIP_SENDRECV	512
Hitachi.OpenTP1.Connector.TP1ConnectorError	DCCNNER_BEGIN	-7751
	DCCNNER_BUF_STAT	-7781
	DCCNNER_CALL	-7750
	DCCNNER_CAST	-7702
	DCCNNER_COMMIT	-7752
	DCCNNER_CONF	-7707
	DCCNNER_DISTRIBUTED_TRN	-7758
	DCCNNER_INVALID_ARGS	-7700
	DCCNNER_INVALID_CONN	-7701
	DCCNNER_INVALID_CUSTOM_RECORD	-7704
	DCCNNER_IOERR	-7703
	DCCNNER_NOT_USE_BUF	-7780
	DCCNNER_OBJECT_NOT_DEFINED	-7705
	DCCNNER_OPEN	-7754
	DCCNNER_PROTO	-7706
	DCCNNER_RECORD_CREATE	-7708
	DCCNNER_RECV	-7757
	DCCNNER_ROLLBACK	-7753

クラス名	フィールド名	値
	DCCNNER_SEND	-7756
	DCCNNER_TRN	-7755
	DCCNNER_UNMATCH_LENGTH	-7709

付録 C　バージョンアップ時の変更点

各バージョンでの変更点を次に示す分類ごとに示します。

- API，定義およびコマンドの追加と削除
- 動作の変更
- API，定義およびコマンドのデフォルト値の変更

付録 C.1　07-50 での変更点

Connector .NET 07-50 での API，定義およびコマンドの追加と削除を次の表に示します。

表 C–1　Connector .NET 07-50 での API，定義およびコマンドの追加と削除

種別	分類	内容
追加	API	なし
	定義	なし
	コマンド	なし
削除	API	なし
	定義	なし
	コマンド	なし
	その他	前提ソフトウェアのサポート終了に伴い，次の言語のサンプルプログラムを削除 • J# • COBOL 言語

Connector .NET 07-50 での動作の変更点を次の表に示します。

表 C–2　Connector .NET 07-50 での動作の変更点

分類	内容
API	なし
定義	なし
コマンド	if2cstub コマンド，if2tsp コマンド，spp2cstub コマンド，および spp2tsp コマンドに指定できるプログラム言語の種類から J#を削除
メッセージ	KFCA32271-I，KFCA32273-I，KFCA32274-I，KFCA32275-I • -l オプションの指定値から vjs を削除
その他	前提ソフトウェアのサポート終了に伴い，使用可能な開発言語から次の言語を除外 • J# • COBOL 言語

分類	内容
	適用 OS を Windows 7，および Windows Server 2008 R2 以降に変更
	Connector .NET を使用する場合の前提ソフトウェアを.NET Framework 3.5 Service Pack 1 に変更

Connector .NET 07-50 での API，定義およびコマンドのデフォルト値の変更はありません。

付録 C.2　07-03 での変更点

Connector .NET 07-03 での API，定義およびコマンドの追加と削除はありません。

Connector .NET 07-03 での動作の変更点を次の表に示します。

表 C–3　Connector .NET 07-03 での動作の変更点

分類	内容
API	なし
定義	なし
コマンド	なし
メッセージ	なし
その他	適用 OS に Windows Server 2008 を追加
	前提ソフトウェアに，Windows SDK 6.1，および Visual Studio 2008 を追加
	開発言語として COBOL 言語に対応

Connector .NET07-03 での API，定義およびコマンドのデフォルト値の変更はありません。

付録 C.3　07-02 での変更点

Connector .NET 07-02 での API，定義およびコマンドの追加と削除を次の表に示します。

表 C–4　Connector .NET 07-02 での API，定義およびコマンドの追加と削除

種別	分類	内容
追加	API	• TP1IntegrationBehavior クラス • TP1IntegrationBinding クラス • TP1RpcClient クラス
	定義	Connector .NET 構成定義 • <destributedTransaction>要素 • <recoveryService>要素

種別	分類	内容
	コマンド	• cnnnidgen コマンド • cnntrsls コマンド
削除	なし	

Connector .NET 07-02 での動作の変更点を次の表に示します。

表 C-5　Connector .NET07-02 での動作の変更点

分類	内容
API	なし
定義	なし
コマンド	なし
メッセージ	次のメッセージを追加 • KFCA32471-I • KFCA32472-I • KFCA32473-I • KFCA32474-I • KFCA32476-I • KFCA32477-I • KFCA32478-I • KFCA32479-E • KFCA32480-I • KFCA32481-I • KFCA32482-I • KFCA32483-E • KFCA32484-E • KFCA32485-E • KFCA32486-E • KFCA32487-E • KFCA32428-E • KFCA32429-E • KFCA32430-E • KFCA32495-I • KFCA32496-E • KFCA32497-W
その他	バッファサイズの見積もり方法で，データ型ごとの合計の計算式に代入できる値を追加 • char[a] • nt[a] • long[a]
	MSDTC 連携機能を追加

分類	内容
	WCF 連携機能を追加
	リモート API 機能でスタティックコネクションスケジュールモードを使用する場合の注意事項を追加
	TP1ConnectorError クラスに次のフィールドを追加 • DCCNNER_DISTRIBUTED_TRN

Connector .NET 07-02 での API，定義およびコマンドのデフォルト値の変更はありません。

付録 C.4　07-01 での変更点

Connector .NET 07-01 での API，定義およびコマンドの追加と削除を次の表に示します。

表 C–6　Connector .NET 07-01 での API，定義およびコマンドの追加と削除

種別	分類	内容
追加	API	なし
	定義	Connector .NET 構成定義 • <connection>要素の failureInfoSharing 属性 • <connection>要素の failureCheckInterval 属性
	コマンド	なし
削除	なし	

Connector .NET 07-01 での動作の変更点を次の表に示します。

表 C–7　Connector .NET 07-01 での動作の変更点

分類	内容
API	なし
定義	なし
コマンド	なし
メッセージ	次のメッセージを追加 • KFCA32413-I • KFCA32414-I • KFCA32415-I • KFCA32427-W
その他	適用 OS に Windows Vista を追加
	前提ソフトウェアに，.NET Framework v3.0，および Windows SDK 6.0 を追加
	接続障害軽減機能を追加

Connector .NET 07-01 での API，定義およびコマンドのデフォルト値の変更はありません。

付録 C.5　07-00 での変更点

Connector .NET 07-00 での API，定義およびコマンドの追加と削除を次の表に示します。

表 C–8　Connector .NET 07-00 での API，定義およびコマンドの追加と削除

種別	分類	内容
追加	API	なし
	定義	Connector .NET 構成定義 • <perfCounter>要素 • <option>要素 • <connection>要素の threshold 属性 • <connection>要素の watchtime 属性 • <log>要素のログレベル 3 の記述 • <common>要素の perfCounter 要素の説明，および子要素 option
	コマンド	if2cstub コマンドに次のオプションを追加 • -m オプション
		if2tsp，および spp2tsp コマンドに次のオプションを追加 • -B オプション • -A オプション
削除	API	TP1ConnectorError クラス • DCCNNER_BASE フィールド
	定義	なし
	コマンド	なし

Connector .NET 07-00 での動作の変更点を次の表に示します。

表 C–9　Connector .NET 07-00 での動作の変更点

分類	内容
API	なし
定義	Connector .NET 構成定義 • <bufferPool>要素の size 属性の範囲を 1～1048575 に変更
コマンド	なし
メッセージ	ログ出力メッセージの形式を変更
	次のメッセージを追加 • KFCA32424-W • KFCA32425-W

分類	内容
	• KFCA32426-W • KFCA32470-E • KFCA32471-I • KFCA32412-I
	次のメッセージを変更 • KFCA32271-E に-m オプションを追加 • KFCA32440-E • KFCA32441-E • KFCA32426-W • KFCA32275-I
その他	適用 OS に Windows Server 2003（64 ビット用）を追加
	リソース監視機能を追加
	セキュリティポリシーの設定 • cnnsepf.exe を追加
	データ型定義で次の項目を変更 • 配列指定の配列要素数の範囲を 1～8388608 に変更 • バイナリ型（byte）を追加
	RPC 送受信メッセージの最大長拡張機能を追加
	.NET インタフェース定義のメソッドの引数および戻り値で使用できるデータ型を追加 • TP1 ユーザ構造体列 • TP1 ユーザ構造体配列
	サービス定義を構成するデータ型定義ファイルに，定義するデータ型に byte[a]を追加
	XML スキーマ定義方法で，データ型定義から定義される element 要素に byte[]を追加

Connector .NET 07-00 での API，定義およびコマンドのデフォルト値の変更を次の表に示します。

表 C–10　Connector .NET 07-00 での API，定義およびコマンドのデフォルト値の変更

分類	内容
API	なし
定義	Connector .NET 構成定義 • <log>要素の level を 1 に変更
コマンド	なし
その他	なし

付録 D　用語解説

Connector .NET に関する用語について説明します。その他の用語については，必要に応じて次のマニュアルおよびドキュメントを参照してください。

- 「OpenTP1 解説」
- 「TP1/LiNK 使用の手引」
- .NET Framework のドキュメント

(記号)

.NET インタフェース定義

SPP.NET の RPC インタフェース情報を.NET Framework に対応したプログラム言語で定義したものです。

(英字)

ADO.NET

.NET Framework が提供するデータアクセスの技術セットです。

ASP.NET

.NET Framework が提供する Web アプリケーションや XML Web サービスを構築するための技術セットです。

CLR（Common Language Runtime）

→共通言語ランタイムの説明を参照してください。

CLS（Common Language Specification）

→共通言語仕様の説明を参照してください。

CTS（Common Type System）

→共通型システムの説明を参照してください。

CUP.NET（Client User Program for .NET Framework）

Client .NET を利用して，SPP.NET や SPP にサービスを要求するクライアントアプリケーションのことです。

さまざまなアプリケーション形態が可能であり，アプリケーション形態に応じたランタイムホストで実行されます。

GAC（Global Assembly Cache）

→グローバルアセンブリキャッシュの説明を参照してください。

SPP.NET（Service Providing Program for .NET Framework）

OpenTP1 for .NET Framework の UAP のうち，ファイルへのアクセスなどサーバの役割をするプログラムのことです。SPP.NET は，クライアント UAP から要求されたサービスを実行するサービスメソッドから構成されます。

OpenTP1 for .NET Framework が提供するランタイムホストで実行されます。

SPP.NET 実行コンテナ

Extension .NET が提供する SPP.NET 実行用のランタイムホストです。

SUP.NET（Service Using Program for .NET Framework）

OpenTP1 for .NET Framework の UAP のうち，SPP.NET または SPP に処理要求をするだけの，クライアントアプリケーションのことです。ほかの UAP にサービスを提供するためのメソッドは持ちません。

TSP（TP1 Service Proxy）

TSP は，コネクションを意識しないで OpenTP1 のサービスを利用できるプロキシクラスです。

XML Web サービス

XML，HTTP，SOAP などのインターネット標準技術を利用して，ネットワーク上に分散した異なるプラットフォーム上のアプリケーションを連携させる技術，または連携したアプリケーション（サービス）のことです。

（ア行）

アセンブリ

.NET Framework のアプリケーションの配置やバージョン管理などを行う場合の基本単位です。アセンブリは，リソースやファイルの論理的な集合として，一つの機能を構成します。

アプリケーションドメイン

共通言語ランタイムがアプリケーション間を分離するために使用する処理単位です。

同じアプリケーションの有効範囲内（アプリケーションスコープ）で作成されたオブジェクトの集合の境界を示します。

アプリケーションベースディレクトリ

アプリケーションドメインに読み込まれる DLL ファイルまたは EXE ファイルを格納するディレクトリのことです。

インデクスドレコード

Connector .NET でバイナリデータを使用した，RPC を発行する場合および TCP/IP 通信をする場合に使用するインタフェースのことです。

（カ行）

カスタムレコードクラス

データ型定義で定義したデータ型を.NET Framework のデータ型に対応づけて生成したデータ変換クラスのことです。

完全限定名

名前空間の名前から指定するオブジェクトの参照方法です。完全限定名を使用すると，使用するオブジェクトを特定できるため，名前の競合が発生しません。

共通型システム（CTS）

共通言語ランタイムでの型の宣言，使用，および管理方法を定義したものです。

共通言語仕様（CLS）

共通言語ランタイムによってサポートされる言語機能のサブセットです。

共通言語ランタイム（CLR）

.NET Framework に対応するプログラムが使用する共通の動作環境（ランタイム）です。

クライアントスタブ

→スタブの説明を参照してください。

グローバルアセンブリキャッシュ（GAC）

複数のアプリケーションで共有するアセンブリを格納するコードキャッシュのことです。

コネクション

Connector .NET は，Client .NET が提供する TP1Client クラスのインスタンスにハンドルを関連づけて管理します。TP1Client クラスのインスタンスに関連づけられたハンドルのことを，コネクションと呼びます。

コネクションプーリング機能

一度生成されたコネクションをプーリングする機能のことです。コネクションプーリング機能を使用すると，コネクションプール内のコネクションを再利用できるため，資源を効率的に使用できます。

コネクションプール

コネクションプーリング機能使用時に，生成されたコネクションをプーリングするための領域のことです。

（サ行）

サーバスタブ

→スタブの説明を参照してください。

サービス定義

SPP.NET や SPP のサービス名と入出力データの定義を対応づけて定義したものです。

サービス定義ファイルとデータ型定義ファイルから構成されます。

出力データ用 XML スキーマ

クライアントスタブのサービスメソッドの戻り値（出力データ）である XML 文書の構造が定義された XML スキーマのことです。

出力データ用 XML スキーマファイル

出力データ用 XML スキーマが記述されたファイルのことです。

スタブ

.NET インタフェース定義を使用する RPC や，サービス定義を使用する RPC の場合に，RPC で使用する入出力データの文字コード変換やエンディアンの識別などを自動的に行うプログラムのことです。

サーバで使用するスタブをサーバスタブ，クライアントで使用するスタブをクライアントスタブと呼びます。

スタブは運用コマンドで生成します。.NET インタフェース定義を使用する RPC の場合は，クライアントスタブとサーバスタブが生成されます。サービス定義を使用する RPC の場合は，クライアントスタブが生成されます。

（タ行）

データ型定義

RPC メッセージのユーザデータの形式を C 言語の構造体のような形式で定義したものです。

（ナ行）

名前空間

名前を一意に識別するための概念です。名前空間と名前を組み合わせることによって，オブジェクトなどを特定できます。

また，名前空間によってクラスなどを論理的にグループ化できます。

入力データ用 XML スキーマ

クライアントスタブのサービスメソッドの引数（入力データ）である XML 文書の構造が定義された XML スキーマのことです。

入力データ用 XML スキーマファイル

入力データ用 XML スキーマが記述されたファイルのことです。

（ハ行）

バッファプーリング機能

RPC 要求に使用するメッセージを格納するバッファ（バイト配列）をプーリングする機能です。

バッファプーリング機能でバッファをプーリングすると，バッファの生成や破棄が少なくなり性能が向上します。

バッファプール

バッファプーリング機能使用時に，生成されたバッファを同じバッファサイズのバッファごとに管理するプールのことです。

索引

E

F

G

I

K

L

M

N

O

P

R

S

T

U

W

X

あ

い

う

え

か

り

れ

ろ