BEA Tuxedo

BEA Tuxedo CORBA
サーバ・アプリケーションの開発方法

BEA Tuxedo リリース 8.0
8.0 版
2001 年 10 月 31 日

BEA Tuxedo CORBA サーバ・アプリケーションの開発方法

Document Edition	Date	Software Version
8.0	2001 年 10 月 31 日	BEA Tuxedo 8.0

目次

2. BEA Tuxedo CORBA サーバ・アプリケーションの作成手順

3. Basic CORBA サーバ・アプリケーションの設計とインプリメント

4. マルチスレッド CORBA サーバ・アプリケーションの作成

8. BEA Tuxedo の CORBA サーバ・アプリケーションのスケーリング

索引

このマニュアルについて

このマニュアルでは、プログラマが BEA Tuxedo 製品の主な機能をインプリメントして、BEA Tuxedo ドメインで実行されるスケーラブルで高性能なサーバ・アプリケーションを設計およびインプリメントする方法について説明しています。「第 3 章 Basic CORBA サーバ・アプリケーションの設計とインプリメント」で使用されている例は、『BEA Tuxedo CORBA University サンプル・アプリケーション』で説明するサンプル・アプリケーションに基づいています。

このマニュアルでは、以下の内容について説明します。

- 「第 1 章 CORBA サーバ・アプリケーションの概念」では、BEA Tuxedo サーバ・アプリケーションの作成に関する基本的な概念を示し、BEA Tuxedo サーバ・アプリケーションをビルドするために作成する 2 つの主要なプログラミング・エンティティについて説明します。
- 「第 2 章 BEA Tuxedo CORBA サーバ・アプリケーションの作成手順」では、BEA Tuxedo サーバ・アプリケーションを作成する基本的な手順について説明します。
- 「第 3 章 Basic CORBA サーバ・アプリケーションの設計とインプリメント」では、CORBA サーバ・アプリケーションの設計およびインプリメントに関連する基本的な概念と手順について、Basic University サンプル・アプリケーションに基づいて説明します。
- 「第 4 章 マルチスレッド CORBA サーバ・アプリケーションの作成」では、マルチスレッド CORBA サーバ・アプリケーションを作成するための要件、目的、および概念について概説します。また、この章では、マルチスレッド CORBA サーバ・アプリケーションを開発およびビルドする手順について概説します。また、simpapp_mt サンプル・アプリケーションをビルドおよび実行する方法と、マルチスレッド CORBA サーバ・アプリケーションを管理する方法についても説明します。
- 「第 5 章 セキュリティと CORBA サーバ・アプリケーション」では、BEA Tuxedo ドメインにセキュリティ・モデルをインプリメントする際に、CORBA サーバ・アプリケーションが果たす役割について説明します。

- 「第 6 章 トランザクションの CORBA サーバ・アプリケーションへの統合」では、BEA Tuxedo システムによってサポートされている BEA Tuxedo ドメインのトランザクション、およびサーバ・アプリケーションにトランザクションをインプリメントする方法について説明します。
- 「第 7 章 BEA Tuxedo サービスの CORBA オブジェクトへのラッピング」では、BEA Tuxedo アプリケーションを CORBA サーバ・アプリケーションに統合する方法について説明します。
- 「第 8 章 BEA Tuxedo の CORBA サーバ・アプリケーションのスケーリング」では、複製されたサーバ・プロセスとサーバ・グループ、ファクトリ・ベース・ルーティング、およびオブジェクトの状態管理を含む主要なスケーラビリティ機能を BEA Tuxedo アプリケーションに組み込んで、スケーラビリティを向上させる方法について説明します。

対象読者

このマニュアルは、安全でスケーラブルなトランザクション・ベースのサーバ・アプリケーションを作成するプログラマを対象としています。BEA Tuxedo システム、CORBA、および C++ プログラミングに読者が精通していることを前提として書かれています。

e-docs Web サイト

BEA Tuxedo 製品のマニュアルは BEA 社の Web サイトで参照することができます。BEA ホーム・ページの [製品のドキュメント] をクリックするか、または http://edocs.beasys.co.jp/e-docs/index.html に直接アクセスしてください。

マニュアルの印刷方法

このマニュアルは、ご使用の Web ブラウザで一度に 1 ファイルずつ印刷できます。Web ブラウザの [ファイル] メニューにある [印刷] オプションを使用してください。

このマニュアルの PDF 版は、Web サイト上にあります。また、マニュアルの CD-ROM にも収められています。この PDF を Adobe Acrobat Reader で開くと、マニュアル全体または一部をブック形式で印刷できます。PDF 形式を利用するには、BEA Tuxedo マニュアルのホーム・ページにある [PDF 版] ボタンをクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader をお持ちでない場合は、Adobe 社の Web サイト (http://www.adobe.co.jp/) から無償でダウンロードできます。

関連情報

CORBA、BEA Tuxedo、分散オブジェクト・コンピューティング、トランザクション処理、C++ プログラミング、および Java プログラミングの詳細については、BEA Tuxedo オンライン・マニュアルの「Bibliography」を参照してください。

サポート情報

皆様の BEA Tuxedo マニュアルに対するフィードバックをお待ちしています。ご意見やご質問がありましたら、電子メールで **docsupport-jp@bea.com** までお送りください。お寄せいただきましたご意見は、BEA Tuxedo マニュアルの作成および改訂を担当する BEA 社のスタッフが直接検討いたします。

電子メール メッセージには、BEA Tuxedo 8.0 リリースのマニュアルを使用していることを明記してください。

BEA Tuxedo に関するご質問、または BEA Tuxedo のインストールや使用に際して問題が発生した場合は、www.bea.com の BEA WebSUPPORT を通して BEA カスタマ・サポートにお問い合わせください。カスタマ・サポートへの問い合わせ方法は、製品パッケージに同梱されているカスタマ・サポート・カードにも記載されています。

カスタマ・サポートへお問い合わせの際には、以下の情報をご用意ください。

- お客様のお名前、電子メール・アドレス、電話番号、Fax 番号
- お客様の会社名と会社の住所
- ご使用のマシンの機種と認証コード
- ご使用の製品名とバージョン
- 問題の説明と関連するエラー・メッセージの内容

表記上の規則

このマニュアルでは、以下の表記規則が使用されています。

規則	項目
太字	用語集に定義されている用語を示します。
Ctrl + Tab	2 つ以上のキーを同時に押す操作を示します。
イタリック体	強調またはマニュアルのタイトルを示します。

規則	項目

等幅テキスト	コード・サンプル、コマンドとオプション、データ構造とメンバ、データ型、ディレクトリ、およびファイル名と拡張子を示します。また、キーボードから入力するテキストも等幅テキストで表示します。
	例 :
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

太字の等幅テキスト	コード内の重要な語を示します。 例 : `void` **`commit`** `( )`
斜体の等幅テキスト	コード内の変数を示します。 例 : `String` *`expr`*
大文字のテキスト	デバイス名、環境変数、および論理演算子を示します。 例 : LPT1 SIGNON OR
{ }	構文の行で、選択肢の組み合わせを示します。かっこは入力しません。
[ ]	構文の行で、オプション項目を示します。かっこは入力しません。 例 : `buildobjclient [-v] [-o` *`name`* `] [-f` *`file-list`*`]...[-l` *`file-list`*`]...`
|	構文の行で、相互に排他的な選択肢の区切りとして使います。記号は入力しません。

規則	項目
. . .	コマンド・ラインで、以下のいずれかの場合を示します。 ■ コマンド・ラインで、同じ引数を繰り返し使用できることを示します。 ■ 文で、追加のオプション引数が省略されていることを示します。 ■ 追加のパラメータ、値、またはその他の情報を入力できることを示します。 記号は入力しません。 例 : `buildobjclient [-v] [-o name ] [-f file-list]...[-l file-list]...`
. . .	コード例または構文の行で、項目が省略されていることを示します。記号は入力しません。

1 CORBA サーバ・アプリケーションの概念

ここでは、次の内容について説明します。

- CORBA サーバ・アプリケーションをビルドするために作成するエンティティ
  - サーバ・アプリケーション用の CORBA オブジェクトのインプリメンテーション
  - Server オブジェクト
- CORBA サーバ・アプリケーションの開発プロセス
  - オブジェクト・リファレンスの生成
  - オブジェクト状態の管理
  - オブジェクトのデータの読み取りと書き込み
  - デザイン・パターンの使い方

このマニュアルの各章では、さまざまな BEA Tuxedo ソフトウェア機能を活用する CORBA サーバ・アプリケーションをビルドする手順および例について説明します。BEA Tuxedo CORBA サーバ・アプリケーションとその機能に関する基本情報については、『BEA Tuxedo CORBA アプリケーション入門』を参照してください。

CORBA サーバ・アプリケーションをビルドするために作成するエンティティ

CORBA サーバ・アプリケーションをビルドするには、次の 2 つのエンティティを作成します。

- サーバ・アプリケーションのビジネス・ロジックを実行する CORBA オブジェクトのインプリメンテーション
- サーバ・アプリケーションを初期化およびリリースし、クライアント要求を満たすために必要な CORBA オブジェクトをインスタンス化するオペレーションをインプリメントする Server オブジェクト

また、IDL コンパイラによって生成され、CORBA サーバ・アプリケーションに組み込む複数のファイルも使用します。これらのファイルのリストと説明については、「BEA Tuxedo CORBA サーバ・アプリケーションの作成手順」を参照してください。

以降の節では、これらのエンティティの基礎知識について説明します。これらのエンティティの生成の詳細については、「BEA Tuxedo CORBA サーバ・アプリケーションの作成手順」を参照してください。

サーバ・アプリケーション用の CORBA オブジェクトのインプリメンテーション

CORBA オブジェクトとは何か、およびそれらをどのように定義、インプリメント、インスタンス化、管理するかについて明確に理解しておくことは、CORBA サーバ・アプリケーションの設計者および作成者にとって不可欠です。

Object Management Group インターフェイス定義言語 (OMG IDL) で定義したインターフェイスの CORBA オブジェクトには、CORBA サーバ・アプリケーションのビジネス・ロジックとデータが含まれます。すべてのクライアント・アプリケーション要求では、CORBA オブジェクトのオペレーションが呼び出されます。インターフェイス用に定義されたオペレーションをインプリメントするコードを、オブジェクト・インプリメンテーションと言います。たとえば、C++ では、オブジェクト・インプリメンテーションは C++ クラスです。

ここでは、次の内容について説明します。

- OMG IDL インターフェイス定義で CORBA オブジェクトの呼び出し可能なオペレーションを確立する方法
- CORBA オブジェクトのオペレーションをインプリメントする方法
- クライアント・アプリケーションがサーバ・アプリケーションの CORBA オブジェクトをアクセスおよび操作する方法
- クライアント要求への応答としてコードとデータを持つ CORBA オブジェクトを実行時にインスタンス化する方法

OMG IDL インターフェイス定義で CORBA オブジェクトのオペレーションを確立する方法

CORBA オブジェクトのインターフェイスは、そのオブジェクトで実行可能なオペレーションを識別します。CORBA オブジェクトに固有の特性は、オブジェクトのインターフェイス定義がそのインプリメンテーションから独立していることです。インターフェイス定義により、インターフェイスのオペレーションをインプリメントする方法が確立されます。これには、オペレーションに受け渡し、オペレーションから返される有効なパラメータなどが含まれます。

インターフェイス定義は OMG IDL で表現され、アプリケーションのクライアント / サーバ間の取り決めを確立します。つまり、特定のインターフェイスについて、サーバ・アプリケーションは次のことを行います。

- そのインターフェイスで定義されたオペレーションをインプリメントします。
- 常に各オペレーションで定義されたパラメータを使用します。

サーバ・アプリケーションがどのようにオペレーションをインプリメントするかは、時間と共に変わる場合があります。この振る舞いは、サーバ・アプリケーションが定義されたインターフェイスをインプリメントし、定義されたパラメータを使用するという要件を満たす限り許容されます。このため、クライアント・スタブは常にサーバ・マシン上のオブジェクト・インプリメンテーションの信頼できるプロキシであり続けます。これは、CORBA アーキテクチャの主要なメリットの 1 つです。つまり、サーバ・アプリケーションがオブジェクトをインプリメントする方法を変更するときに、クライアント・アプリケーションを修正する必要がなく、クライアント・アプリケーションがその変更を認識する必要もないということです。

また、インターフェイス定義は、クライアント・スタブと、サーバ・アプリケーションのスケルトンの両方の内容を決定します。これらの 2 つのエンティティに ORB とポータブル・オブジェクト・アダプタ (POA) を組み合わせることにより、オブジェクトのオペレーションに対するクライアント要求を、その要求を満たすことができるサーバ・アプリケーションのコードにルーティングできます。

システム設計者がアプリケーションのビジネス・オブジェクトのインターフェイスを指定したら、プログラマはそのインターフェイスをインプリメントします。このマニュアルでは、その方法について説明します。

OMG IDL の詳細については、『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』を参照してください。

CORBA オブジェクトのオペレーションをインプリメントする方法

前述のとおり、インターフェイスで定義されたオペレーションをインプリメントするコードを、オブジェクト・インプリメンテーションと言います。C++ の場合、このコードはメソッドのセットで構成され、アプリケーションの OMG IDL ファイル内のインターフェイスで定義されたオペレーションごとに 1 つ存在します。アプリケーションのオブジェクト・インプリメンテーションのセットが記述されたファイルを、インプリメンテーション・ファイルと言います。BEA Tuxedo システムには、IDL コンパイラが用意されています。IDL コンパイラは、次の図に示すように、アプリケーションの OMG IDL ファイルをコンパイルして、インプリメンテーション・ファイルを含むいくつかのファイルを生成します。

生成されたインプリメンテーション・ファイルには、メソッド・テンプレート、メソッド宣言、オブジェクトのコンストラクタとデストラクタ、およびその他のデータが含まれています。これらを出発点として、アプリケーションのオブジェクト・インプリメンテーションを記述できます。たとえば C++ では、生成されたインプリメンテーション・ファイルには各インターフェイスのメソッドのシグニチャが含まれています。このファイルに各メソッドのビジネス・ロジックを入力し、そのファイルをサーバ・アプリケーションの実行可能ファイルをビルドするコマンドへの入力として提供します。

クライアント・アプリケーションがサーバ・アプリケーションの CORBA オブジェクトをアクセスおよび操作する方法

クライアント・アプリケーションは、サーバ・アプリケーションによって管理される CORBA オブジェクトのオブジェクト・リファレンスを介して、それらのオブジェクトをアクセスおよび操作します。クライアント・アプリケーションは、オブジェクト・リファレンス上のオペレーション (つまり要求) を呼び出します。これらの要求はメッセージとしてサーバ・アプリケーションに送信されます。サーバ・アプリケーションは、CORBA オブジェクトの適切なオペレーションを呼び出します。これらの要求がサーバ・アプリケーションに送信され、そのサーバ・アプリケーションで呼び出されるという事実は、クライアントからは完全に見えません。クライアント・アプリケーションは、単にクライアント・スタブで呼び出しを行っているように見えます。

クライアント・アプリケーションは、オブジェクト・リファレンスによってのみ CORBA オブジェクトを操作できます。設計上の主要な考慮事項の 1 つは、アプリケーションに適した方法でどのようにオブジェクト・リファレンスを作成し、それらを必要としているクライアント・アプリケーションにそれらを返すかです。

一般に、CORBA オブジェクトへのオブジェクト・リファレンスはファクトリによって BEA Tuxedo システムで作成されます。ファクトリは、別の CORBA オブジェクトへのオブジェクト・リファレンスをそのオペレーションの 1 つとして返す任意の CORBA オブジェクトです。アプリケーションのファクトリは、アプリケーション用に定義したほかの CORBA オブジェクトと同じようにインプリメントします。ファクトリを FactoryFinder に登録することによって、ファクトリを BEA Tuxedo ドメイン、およびその BEA Tuxedo ドメインに接続されているクライアントに認識させることができます。ファクトリの登録は、通常 Server オブジェクトによって実行されるオペレーションです。詳細については、「Server オブジェクト」を参照してください。ファクトリの設計の詳細については、「オブジェクト・リファレンスの生成」を参照してください。

オブジェクト・リファレンスの内容

クライアント・アプリケーションからは、オブジェクト・リファレンスはオペーク、つまりブラック・ボックスのように見えます。クライアント・アプリケーションは、その内容を知らなくてもオブジェクト・リファレンスを使用できます。ただし、オブジェクト・リファレンスには、BEA Tuxedo システムが特定のオブジェクト・インスタンス、およびそのオブジェクトに関連付けられている状態データを検索するために必要なすべての情報が格納されています。

オブジェクト・リファレンスには、以下の情報が格納されます。

- インターフェイス名

  これは、オブジェクトの OMG IDL インターフェイスのインターフェイス・リポジトリ ID です。

- オブジェクト ID (OID)

  OID は、リファレンスを適用するオブジェクトのインスタンスを一意に識別します。オブジェクトが外部ストレージ内にデータを持っている場合、OID には通常サーバ・マシンがそのオブジェクトのデータを検索するために使用できるキーも含まれます。

- グループ ID

  グループ ID は、クライアント・アプリケーションがオブジェクト・リファレンスを使用して要求を行ったときに、そのオブジェクト・リファレンスのルーティング先となるサーバ・グループを識別します。デフォルト以外のグループ ID の生成は、ファクトリ・ベース・ルーティングという BEA Tuxedo の主要機能の 1 つです。詳細については、「ファクトリ・ベース・ルーティング」で説明します。

注記　上のリストの 3 つの情報を組み合わせることにより、CORBA オブジェクトが一意に識別されます。2 つのグループに同じオブジェクト・インプリメンテーションが含まれている場合、特定のインターフェイスと OID を持つオブジェクトをその 2 つのグループで同時に活性化できます。特定のインターフェイス名と OID のオブジェクト・インスタンスをドメイン内でいつでも 1 つしか利用できないようにする必要がある場合は、ファクトリ・ベース・ルーティングを使用して特定の OID を持つオブジェクトが常に同じグループにルーティングされるようにするか、または特定のオブジェクト・インプリメンテーションが 1 つのグループだけに存在するようドメインをコンフィギュレーションします。これにより、複数のクライアントが特定のインターフェイス名と OID のオブジェクト・リファレンスを持つ場合、そのリファレンスは常に同じオブジェクト・インスタンスにルーティングされます。

ファクトリ・ベース・ルーティングの詳細については、「ファクトリ・ベース・ルーティング」を参照してください。

オブジェクト・リファレンスの存続期間

BEA Tuxedo ドメインで動作するサーバ・アプリケーションによって作成されるオブジェクト・リファレンスには、通常それを作成したサーバ・プロセスの存続期間より長い寿命が設定されます。BEA Tuxedo オブジェクト・リファレンスは、最初にそれらを作成したサーバ・プロセスが実行されているかどうかに関係なく、クライアント・アプリケーションで使用できます。このように、オブジェクト・リファレンスは特定のサーバ・プロセスに関連付けられてはいません。

`TP::create_active_object_reference()` オペレーションで作成されたオブジェクト・リファレンスは、それを作成したサーバ・プロセスが存続する間だけ有効です。詳細については、「状態を持つオブジェクトの事前活性化」を参照してください。

オブジェクト・インスタンスの受け渡し

ORB は、オブジェクト・インスタンスをオブジェクト・リファレンスとしてマーシャリングできません。たとえば、次のコードでファクトリ・リファレンスを受け渡すと、BEA Tuxedo システムで CORBA マーシャリング例外が発生します。

```
connection::setFactory(this);
```

オブジェクト・インスタンスを受け渡すには、次の例のように、プロキシ・オブジェクト・リファレンスを作成して、そのプロキシを代わりに受け渡します。

```
CORBA::Object myRef = TP::get_object_reference();
ResultSetFactory factoryRef = ResultSetFactoryHelper::_narrow(myRef);
connection::setFactoryRef(factoryRef);
```

CORBA オブジェクトの実行時のインスタンス化

サーバ・アプリケーションが、サーバ・マシンのメモリにマップされていないオブジェクト (活性化されていないオブジェクト) の要求を受信した場合、TP フレームワークは `Server::create_servant()` オペレーションを呼び出します。`Server::create_servant()` オペレーションは、作成する CORBA サーバ・アプリケーションのコンポーネントである Server オブジェクト (「サーバ・アプリケーション用の CORBA オブジェクトのインプリメンテーション」を参照) でインプリメントされます。

`Server::create_servant()` オペレーションにより、CORBA オブジェクト・インプリメンテーションのインスタンスがサーバ・マシンのメモリにマップされます。このオブジェクト・インプリメンテーションのインスタンスを、オブジェ

クトのサーバントと呼びます。正式には、サーバントとはアプリケーションの OMG IDL ファイルに定義されるインターフェイスをインプリメントする C++ クラスのインスタンスです。サーバントは、Server::create_servant() オペレーションに記述する C++ new 文によって生成されます。

オブジェクトのサーバントが作成されたら、TP フレームワークはそのサーバントの Tobj_ServantBase::activate_object() オペレーションを呼び出します。Tobj_ServantBase::activate_object() オペレーションは、すべてのオブジェクト・インプリメンテーション・クラスが継承する Tobj_ServantBase 基本クラスで定義される仮想オペレーションです。TP フレームワークは、Tobj_ServantBase::activate_object() オペレーションを呼び出してサーバントをオブジェクト ID (OID) に関連付けます。反対に、TP フレームワークが Tobj_ServantBase::deactivate_object() オペレーションを呼び出すと、サーバントと OID との関連付けが解除されます。

オブジェクトのデータがディスク上に存在し、CORBA オブジェクトを活性化するときにそのデータをメモリに読み取る場合、そのオブジェクトの Tobj_ServantBase::activate_object() オペレーションを定義およびインプリメントすることによってそのデータを読み取ることができます。Tobj_ServantBase::activate_object() オペレーションには、オブジェクトの永続的な状態をメモリに読み取るために必要な特定の読み取りオペレーションを格納できます。代わりに、インプリメンテーション・ファイルにコーディングしたオブジェクトの 1 つまたは複数のオペレーションによってオブジェクトのディスク・データをメモリに読み取ることもできます。詳細については、第 1 章の 17 ページ「オブジェクトのデータの読み取りと書き込み」を参照してください。Tobj_ServantBase::activate_object() オペレーションの呼び出しが完了したら、オブジェクトは活性化されていると見なされます。

こうしたオブジェクトのインプリメンテーションとデータの集合から、CORBA オブジェクトの実行時のアクティブ・インスタンスが構成されます。

サーバント・プール

サーバント・プールを使用すると、CORBA サーバ・アプリケーションはサーバントと特定の OID との関連付けが解除された後でも、サーバントをメモリに格納し続けることができます。プールされたサーバントで処理できるクライアント要求が到着すると、TP フレームワークは TP::create_servant オペレーションをバイパスし、プールされたサーバントとクライアント要求で提供された OID の間にリンクを作成します。

このため、サーバント・プールを使用すると、CORBA サーバ・アプリケーションは、サーバントによって処理できるオブジェクト要求が到着するたびにそのサーバントを再インスタンス化するコストを最小限に抑えることができます。サーバント・プールとその使い方の詳細については、「サーバント・プール」を参照してください。

注記　サーバント・プールは、WebLogic Enterprise リリース 4.2 で初めて導入されました。

Server オブジェクト

Server オブジェクトは、CORBA サーバ・アプリケーション用に作成するもう 1 つのプログラミング・コード・エンティティです。Server オブジェクトは、次のタスクを実行するオペレーションをインプリメントします。

- 基本的なサーバ・アプリケーション初期化オペレーションの実行。これには、サーバ・アプリケーションによって管理されるファクトリの登録や、サーバ・アプリケーションで必要なリソースの割り当てが含まれます。サーバ・アプリケーションがトランザクションに関与する場合、Server オブジェクトは XA リソース・マネージャをオープンするコードをインプリメントします。
- クライアント要求を満たすために必要な CORBA オブジェクトのインスタンス化。
- サーバ・アプリケーションが要求の処理を完了したときのサーバ・プロセスのシャットダウンおよびクリーンアップ手順の実行。たとえば、サーバ・アプリケーションがトランザクションに関与する場合、Server オブジェクトは XA リソース・マネージャをクローズするコードをインプリメントします。

Server オブジェクトは、一般のテキスト・エディタを使用してゼロから作成します。作成した Server オブジェクトは、サーバ・アプリケーションのビルド・コマンド、`buildobjserver` への入力として提供します。Server オブジェクトの作成については、「BEA Tuxedo CORBA サーバ・アプリケーションの作成手順」を参照してください。

CORBA サーバ・アプリケーションの開発プロセス

この節では、次のトピックに関する背景情報を提供します。この情報は、CORBA サーバ・アプリケーションの設計およびインプリメンテーション方法に大きな影響を与えます。

- オブジェクト・リファレンスの生成
- オブジェクト状態の管理
- オブジェクトのデータの読み取りと書き込み
- デザイン・パターンの使い方

次の章に進む前にこれらのトピックに目を通しておくことは必須ではありません。ただし、ここでこれらのトピックを提供するのは、すべての CORBA サーバ・アプリケーションの基本的な設計とインプリメンテーションの問題に広く適用されるからです。

オブジェクト・リファレンスの生成

CORBA サーバ・アプリケーションの最も基本的な機能の 1 つは、クライアント・アプリケーションに対して、そのビジネス・ロジックを実行するために必要なオブジェクトのオブジェクト・リファレンスを提供することです。通常、CORBA クライアント・アプリケーションは、自身が使用する初期 CORBA オブジェクトのオブジェクト・リファレンスを次の 2 つのソースから取得します。

- Bootstrap オブジェクト
- BEA Tuxedo ドメインで管理されるファクトリ

クライアント・アプリケーションは、Bootstrap オブジェクトを使用して BEA Tuxedo ドメイン内の特定のオブジェクト・セット (FactoryFinder オブジェクトや SecurityCurrent オブジェクトなど) の初期リファレンスを解決します。Bootstrap オブジェクトについては、『BEA Tuxedo CORBA アプリケーション入門』と『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』を参照してください。

ただし、ファクトリはユーザが設計、インプリメント、および登録します。ファクトリを使用すると、クライアント・アプリケーションは CORBA サーバ・アプリケーションのオブジェクト (特に初期サーバ・アプリケーション・オブジェクト) のリファレンスを取得できます。単純に言えば、ファクトリは別の CORBA オブジェクトのオブジェクト・リファレンスを返す任意の CORBA オブジェクトです。通常、クライアント・アプリケーションは、ファクトリのオペレーションを呼び出して特定のタイプの CORBA オブジェクトのオブジェクト・リファレンスを取得します。CORBA サーバ・アプリケーションを開発する際には、ファクトリの計画とインプリメンテーションを注意深く行うことが重要です。

クライアント・アプリケーションがサーバ・アプリケーションのファクトリを見つけるしくみ

クライアント・アプリケーションは、サーバ・アプリケーションによって管理されるファクトリを FactoryFinder を介して検索できます。通常、Server オブジェクトを開発する場合、サーバ・アプリケーションによって管理されるファクトリを FactoryFinder に登録するコードを組み込みます。この登録オペレーションを通じて、FactoryFinder はサーバ・アプリケーションのファクトリを追跡し、ファクトリを要求するクライアント・アプリケーションにそれらのオブジェクト・リファレンスを提供できます。オブジェクト・リファレンスを取得するときには、ファクトリを使用してそれらを FactoryFinder に登録することをお勧めします。この方法を使用すると、クライアント・アプリケーションは CORBA サーバ・アプリケーションのオブジェクトを簡単に見つけることができます。

アクティブ・オブジェクト・リファレンスの作成

アクティブ・オブジェクト・リファレンスは、サーバ・アプリケーションがオブジェクト・リファレンスを生成するための代わりの手段を提供する機能です。アクティブ・オブジェクト・リファレンスは、一般に前節で説明したファクトリによっては作成されません。アクティブ・オブジェクト・リファレンスは、状態を持つオブジェクトを事前活性化するための手段です。オブジェクトの状態については、次の節で詳しく説明します。

既存のオブジェクト・リファレンスに関連付けられているオブジェクトは、クライアント・アプリケーションがそのオブジェクトに対して呼び出しを行うまでインスタンス化されません。ただし、アクティブ・オブジェクト・リファレンスに関連付けられているオブジェクトは、そのアクティブ・オブジェクト・リファレンスの作成時に作成され、活性化されます。アクティブ・オブジェクト・リファ

レンスは、イテレータ・オブジェクトなどの特定の目的に役立ちます。アクティブ・オブジェクト・リファレンスの詳細については、「状態を持つオブジェクトの事前活性化」を参照してください。

注記 アクティブ・オブジェクト・リファレンスの機能は、WebLogic Enterprise バージョン 4.2 で最初に導入されました。

オブジェクト状態の管理

オブジェクト状態の管理は、大規模クライアント / サーバ・システムの重要な課題です。このようなシステムでは、スループットと応答時間の最適化が不可欠だからです。BEA Tuxedo ドメインで実行されるアプリケーションなど、高スループット・アプリケーションの大部分は、状態を持たない傾向にあります。つまり、こうしたシステムは、サービスまたはオペレーションが達成された後にメモリから状態情報をフラッシュします。

状態の管理は、CORBA ベースのサーバ・アプリケーションを記述するのに不可欠の要素です。一般に、これらのサーバ・アプリケーションの状態を、スケーラビリティと高性能を実現するような方法で管理するのは困難です。BEA Tuxedo ソフトウェアは、状態を管理すると同時に、スケーラビリティと高性能を保証するための簡単な方法を提供します。

CORBA サーバ・アプリケーションにスケーラビリティを組み込むことにより、サーバ・アプリケーションは、数百または数千のクライアント・アプリケーション、複数のマシン、複製サーバ・プロセス、およびそれに比例した多数のオブジェクトとそれらに対する呼び出しで構成される環境においても正常に機能します。

オブジェクト状態について

BEA Tuxedo ドメインでは、オブジェクト状態は、特にオブジェクトに対する複数のクライアント呼び出しにわたるそのオブジェクトのプロセス状態を表します。BEA Tuxedo ソフトウェアでは、状態を持つオブジェクトと状態を持たないオブジェクトに関する以下の定義を使用します。

オブジェクト	振る舞いの特性
状態を持たない	このオブジェクトは、自身のオペレーションの 1 つの呼び出しの間だけメモリにマップされ、その呼び出しが完了すると、非活性化されてそのプロセス状態がメモリからフラッシュされます。つまり、このオブジェクトの状態は、呼び出しの完了後はメモリに保持されません。
状態を持つ	このオブジェクトは、その呼び出しと呼び出しの間も活性化され、その状態はそれらの呼び出しを通して保持されます。状態は、次のような特定のイベントが発生するまでメモリに保持されます。 ■ オブジェクトが存在するサーバ・プロセスの停止またはシャットダウン ■ オブジェクトが参加しているトランザクションのコミットまたはロールバック ■ オブジェクトによる自身の `TP::deactivateEnable()` オペレーションの呼び出し これらのイベントについては、この節でそれぞれ詳しく説明します。

状態を持たないオブジェクトと状態を持つオブジェクトはどちらもデータを持ちますが、状態を持つオブジェクトは、それらのオペレーション呼び出しの間コンテキスト (状態) を保持するのに必要な非永続データをメモリ内に持つことができます。このため、こうした状態を持つオブジェクトの後続の呼び出しは、常に同じサーバントに送られます。反対に、状態を持たないオブジェクトの呼び出しは、BEA Tuxedo システムによって、そのオブジェクトを活性化できる任意の利用可能なサーバ・プロセスに送ることができます。

また、状態管理には、オブジェクトがアクティブであり続ける時間も含まれます。これは、サーバの性能とマシン・リソースの利用状態に重要な影響を与えます。活性化されたオブジェクトの存続期間は、オブジェクトのインターフェイスに割り当てるオブジェクトの活性化方針によって指定されます。オブジェクトの活性化方針については、次の節で説明します。

オブジェクトの状態は、クライアント・アプリケーションからは見えません。クライアント・アプリケーションは、分散オブジェクトとの既存の対話モデルをインプリメントします。クライアント・アプリケーションがオブジェクト・リファ

レンスを持っている限り、追加の要求に対してオブジェクトが常に利用可能であると見なされ、オブジェクトはクライアント・アプリケーションがそのオブジェクトと対話している間メモリ内に継続して保持されているように見えます。

アプリケーションの性能を最適化するには、作成するアプリケーションのオブジェクトがどのように状態を管理するのかについて注意深く計画する必要があります。オブジェクトは、非活性化される前に自身の状態を永続ストレージに保存する必要があります (該当する場合)。またオブジェクトは、活性化されるときに自身の状態を永続ストレージから復元する必要があります (該当する場合)。オブジェクト状態情報の読み取りと書き込みについては、「オブジェクトのデータの読み取りと書き込み」を参照してください。

注記 BEA Tuxedo リリース 8.0 では、性能強化としてパラレル・オブジェクトがサポートされています。この機能を利用すると、特定アプリケーションのすべてのビジネス・オブジェクトを状態を持たないオブジェクトとして指定できます。詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「TP フレームワーク」を参照してください。

オブジェクトの活性化方針

BEA Tuxedo ソフトウェアには、3 つのオブジェクトの活性化方針が用意されています。これらの方針をオブジェクトのインターフェイスに割り当てることで、オブジェクトがクライアント要求によって呼び出された後にメモリ内に保持される期間を指定できます。これらの方針によって、適用先のオブジェクトが一般に状態を持たないか、または状態を持つかが決定されます。

次の表に、3 つの方針とその説明を示します。

方針	説明
`Method`	オブジェクトは、自身のオペレーションの 1 つの呼び出しの間だけ活性化されます。つまり、オブジェクトは呼び出しの開始時に活性化され、呼び出しの終了時に非活性化されます。この活性化方針が適用されるオブジェクトを、メソッド・バウンド・オブジェクトと言います。 `method` 活性化方針は、状態を持たないオブジェクトに関連付けられます。この活性化方針はデフォルトです。

方針	説明
Transaction	オブジェクトは、自身のオペレーションが呼び出されたときに活性化されます。オブジェクトがトランザクションのスコープ内で活性化された場合、オブジェクトはそのトランザクションがコミットまたはロールバックされるまで活性化されます。オブジェクトがトランザクションのスコープの外で活性化された場合、その振る舞いはメソッド・バウンド・オブジェクトと同じです。この活性化方針が適用されるオブジェクトを、トランザクション・バウンド・オブジェクトと言います。 トランザクションのスコープ内のオブジェクトの振る舞い、およびこの方針を使用するための一般的なガイドラインについては、「第 6 章 トランザクションの CORBA サーバ・アプリケーションへの統合」を参照してください。 transaction 活性化方針は、期間が制限され、特定の環境に置かれている状態を持つオブジェクトに関連付けられます。
Process	オブジェクトは、自身のオペレーションが呼び出されたときに活性化され、次の環境下でのみ非活性化されます。 ■ このオブジェクトを管理するサーバ・プロセスがシャットダウンされた。 ■ このオブジェクトのオペレーションによって TP::deactivateEnable() オペレーションが呼び出され、その結果このオブジェクトが非活性化された。これは、アプリケーション制御の非活性化と呼ばれる BEA Tuxedo の主要機能の一部です。詳細については、第 1 章の 16 ページ「アプリケーション制御の非活性化」を参照してください。 この活性化方針が適用されるオブジェクトを、プロセス・バウンド・オブジェクトと言います。process 活性化方針は、状態を持つオブジェクトに関連付けられます。

オブジェクトの活性化方針を割り当てることにより、オブジェクトを非活性化させるイベントを指定できます。オブジェクトの活性化方針をオブジェクトのインターフェイスに割り当てる方法については、「ステップ 4: オブジェクトのメモリ内での振る舞い」を参照してください。

アプリケーション制御の非活性化

BEA Tuxedo ソフトウェアには、アプリケーション制御の非活性化という機能も用意されています。この機能を使用すると、アプリケーションは実行時にオブジェクトを非活性化できます。BEA Tuxedo ソフトウェアは、プロセス・バウンド・オブジェクトが自身に対して呼び出すことができる

`TP::deactivateEnable()` オペレーションを提供しています。

`TP::deactivateEnable()` オペレーションが呼び出されると、そのオペレーションが存在するオブジェクトは自身に対する現在のクライアント呼び出しの完了時に非活性化されます。オブジェクトは、自身に対してだけこのオペレーションを呼び出すことができます。呼び出しが行われれているオブジェクト以外のオブジェクトでこのオペレーションを呼び出すことはできません。

アプリケーション制御の非活性化機能は、オブジェクトに対する一定数のクライアント呼び出しの間だけそのオブジェクトをメモリに保持し、クライアント・アプリケーションからオブジェクトにそのオブジェクトの処理が終了したことを通知できるようにする場合に特に役立ちます。その時点で、オブジェクトは自身をメモリから消去します。

したがって、アプリケーション制御の非活性化を使用すると、オブジェクトはプロセス・バウンド・オブジェクトと同じようにメモリに保持されます。つまり、オブジェクトは自身に対するクライアント呼び出しによって活性化され、その初期クライアント呼び出しの完了後もメモリに保持されます。このオブジェクトは、それが存在するサーバ・プロセスをシャットダウンせずに非活性化できます。

アプリケーション制御の非活性化の代わりの方法は、トランザクションをスコープしてクライアント・アプリケーションとオブジェクト間の会話を管理することです。ただし、トランザクションは本質的にコストが高く、一般にその期間が不定の状態では適していません。

一般に、アプリケーション制御の非活性化または会話のトランザクションを選択する場合、それらにディスク書き込みオペレーションが含まれているかどうかを調べます。会話に読み取り専用オペレーションが含まれているか、またはメモリ内のみの状態の管理が含まれている場合、アプリケーション制御の非活性化の方が適しています。会話中または会話の最後にデータをディスクに書き込む場合は、トランザクションの方が適しています。

注記 アプリケーション制御の非活性化を使用して、クライアント・アプリケーションとサーバ・アプリケーションによって管理されるオブジェクト間の会話モデルをインプリメントする場合、オブジェクトは最終的に `TP::deactivateEnable()` オペレーションを呼び出す必要があります。呼び出さない場合、オブジェクトはメモリ内で永久的にアイドル状態に置かれます。これは、`TP::deactivateEnable()` オペレーションの呼び出し前にクライアント・アプリケーションがクラッシュした場合に危険を招くおそれがあります。一方、トランザクションはタイムアウト・メカニズムをインプリメントして、オブジェクトが永久的にアイドル状態に置かれることを防ぎます。これは、2 つの会話モデルのいずれかを選択する際のもう 1 つの考慮事項です。

アプリケーション制御の非活性化は、次の手順を使用してオブジェクトにインプリメントします。

1. インプリメンテーション・ファイルで、アプリケーション制御の非活性化を使用するインターフェイスのオペレーション内の適切な位置に `TP::deactivateEnable()` オペレーションの呼び出しを挿入します。
2. インプリメンテーション・コンフィギュレーション・ファイル (ICF ファイル) で、`process` 活性化方針を、`TP::deactivateEnable()` オペレーションを呼び出すオペレーションが含まれるインターフェイスに割り当てます。
3. 「ステップ 5: サーバ・アプリケーションのコンパイルとリンク」と「ステップ 6: サーバ・アプリケーションのデプロイ」で説明するとおり、アプリケーションをビルドおよびデプロイします。

オブジェクトのデータの読み取りと書き込み

サーバ・アプリケーションによって管理される CORBA オブジェクトの多くは、外部ストレージにデータを持っています。こうした外部に格納されているデータは、オブジェクトの永続状態と見なされます。オブジェクト状態管理が適切に機能するには、オブジェクト・インプリメンテーションの適切なポイントで永続状態を処理する必要があります。

オブジェクトの永続状態の読み取りと書き込みに関するクライアント / サーバ・アプリケーションのさまざまな要件のために、TP フレームワークはディスク上の永続オブジェクト状態を自動的に処理できません。一般に、オブジェクトの永続状態が 1 つまたは複数のクライアント呼び出しによって修正される場合、その

オブジェクトが非活性化される前に永続状態を保存する必要があります。また、オブジェクトが活性化されている間にどのようにオブジェクトのデータを格納または初期化するかを注意深く計画する必要があります。

以降の節では、オブジェクトの永続状態を処理するためのメカニズムについて説明し、特定の環境でオブジェクト状態を読み書きする方法についての一般的なヒントを示します。具体的には、次のトピックについて説明します。

- オブジェクトの永続状態を読み書きするためのメカニズム
- オブジェクトの活性化時の状態の読み取り
- オブジェクトの個々のオペレーションでの状態の読み取り
- 状態を持たないオブジェクトと永続状態
- 状態を持つオブジェクトと永続状態
- オブジェクトの非活性化の責任
- 不必要な I/O の回避

永続状態の読み書きの選択方法は、クライアント / サーバ・アプリケーションの要件、特にデータの構築方法によって異なります。一般に、最も重要なのは、特に XA リソース・マネージャが関与する場合に、ディスク・オペレーションの数を最小限に抑えることです。

オブジェクトの永続状態を読み書きするためのメカニズム

表 1-1 と 表 1-2 に、オブジェクトの永続状態を読み書きするためのメカニズムを示します。

表 1-1 オブジェクトの永続状態を読み取るためのメカニズム

メカニズム	説明
`Tobj_ServantBase::activate_object()`	オブジェクトのサーバントが作成されたら、TP フレームワークはそのサーバントの `Tobj_ServantBase::activate_object()` オペレーションを呼び出します。第 1 章の 7 ページ「CORBA オブジェクトの実行時のインスタンス化」で説明したとおり、このオペレーションはクライアント / サーバ・アプリケーション用に定義するすべての CORBA オブジェクトが継承する `Tobj_ServantBase` 基本クラスで定義されます。 オブジェクトの `Tobj_ServantBase::activate_object()` オペレーションを定義およびインプリメントしないことを選択することも可能です。この場合、TP フレームワークがオブジェクトを活性化したときに特定のオブジェクト状態の処理に関しては何も発生しません。ただし、このオペレーションを定義およびインプリメントする場合、オブジェクトの永続状態の一部または全部をメモリに読み取るコードをオペレーションに組み込むことができます。このため、`Tobj_ServantBase::activate_object()` オペレーションを使用すると、サーバ・アプリケーションには、オブジェクトの永続状態をメモリに読み取るための最初の機会が与えられます。 オブジェクトの OID にデータベース・キーが含まれている場合、`Tobj_ServantBase::activate_object()` オペレーションはそのキーを OID から抽出するための手段だけを提供します。 `Tobj_ServantBase::activate_object()` オペレーションのインプリメントについては、「ステップ 2: 各インターフェイスのオペレーションをインプリメントするメソッドの記述」を参照してください。 `Tobj_ServantBase::activate_object()` オペレーションのインプリメントの例については、「Basic CORBA サーバ・アプリケーションの設計とインプリメント」を参照してください。

表 1-1 オブジェクトの永続状態を読み取るためのメカニズム (続き)

メカニズム	説明
オブジェクトのオペレーション	オブジェクトで定義する個々のオペレーションの内部に、オブジェクトの永続状態を読み取るコードを組み込むことができます。

表 1-2 オブジェクトの永続状態を書き込むためのメカニズム

メカニズム	説明
`Tobj_ServantBase::deactivate_object()`	オブジェクトが TP フレームワークによって非活性化される場合、TP フレームワークはオブジェクト非活性化の最終ステップとしてこのオペレーションを呼び出します。`Tobj_ServantBase::activate_object()` オペレーションと同様、`Tobj_ServantBase::deactivate_object()` オペレーションも `Tobj_ServantBase` クラスで定義されます。オブジェクトの `deactivate_object()` オペレーションは、メモリからフラッシュするか、またはデータベースに書き込む特定のオブジェクト状態が存在する場合にオプションでインプリメントします。 `Tobj_ServantBase::deactivate_object()` オペレーションは、サーバ・アプリケーションに、オブジェクトの非活性化前に永続状態をディスクに書き込む最後の機会を提供します。 オブジェクトがメモリ内のデータを保持するか、任意の目的のためにメモリを割り当てる場合、`Tobj_ServantBase::deactivate_object()` オペレーションをインプリメントしてオブジェクトが最後にメモリからデータをフラッシュできるようにします。オブジェクトの非活性化前にメモリから状態をフラッシュすることは、メモリ・リークを回避するために不可欠です。

表 1-2 オブジェクトの永続状態を書き込むためのメカニズム (続き)

メカニズム	説明
オブジェクトのオペレーション	ディスクから永続状態を読み取る個々のオペレーションをオブジェクト上で持つことができるように、永続状態をディスクに書き込む個々のオペレーションを持つことができます。 一般に、メソッド・バウンド・オブジェクトとプロセス・バウンド・オブジェクトの場合、データベース書き込みオペレーションはこれらのオペレーションの中で実行し、`Tobj_ServantBase::deactivate_object()` オペレーションでは実行しません。 ただし、トランザクション・バウンド・オブジェクトの場合、`Tobj_ServantBase::deactivate_object()` オペレーションで永続状態を書き込むと、トランザクション・サーバ・アプリケーションにとって意味のあるオブジェクト管理上の効率化が実現されます。

注記 `Tobj_ServantBase::deactivate_object()` オペレーションを使用して永続状態をディスクに書き込む場合、ディスクへの書き込み中に発生するエラーはクライアント・アプリケーションにはレポートされません。このため、このオペレーションでデータをディスクに書き込むのは、オブジェクトがトランザクション・バウンドの場合 (`transaction` 活性化方針が割り当てられている場合) か、または TransactionCurrent オブジェクトを呼び出すことによってトランザクション内でディスク書き込みオペレーションをスコープする場合です。トランザクション内でディスクへの書き込み中に発生したエラーは、クライアント・アプリケーションにレポートできます。`Tobj_ServantBase::deactivate_object()` オペレーションを使用してオブジェクト状態をディスクに書き込む方法については、「Tobj_ServantBase::deactivate_object() での状態処理に関する注意」を参照してください。

オブジェクトの活性化時の状態の読み取り

オブジェクトの Tobj_ServantBase::activate_object() オペレーションを使用した永続状態の読み取りは、次の条件のいずれかが存在するときに適しています。

- オブジェクトのすべてのオペレーションでオブジェクト・データが常に使用または更新される。
- オブジェクトのすべてのデータを 1 つのオペレーションで読み取ることができる。

Tobj_ServantBase::activate_object() オペレーションを使用した永続状態の読み取りには、次のメリットがあります。

- オブジェクト・データを使用する各オペレーションでコードを複製する代わりに、データを読み取るコードを 1 度記述するだけで済みます。
- オブジェクトのデータを使用するオペレーションは、そのデータの読み取りを実行する必要がありません。このため、状態の初期化とは独立した方法でオペレーションを記述できます。

オブジェクトの個々のオペレーションでの状態の読み取り

活性化方針に関係なく、どのオブジェクトを使用する場合も、そのデータを必要とする各オペレーションで永続データを読み取ることができます。つまり、永続状態の読み取りを Tobj_ServantBase::activate_object() オペレーションの外部で処理できます。この方法は、次の場合に適しています。

- オブジェクト状態が、複数のオペレーションで読み取りまたは書き込みを行う必要がある個別のデータ・エレメントで構成されている。
- オブジェクトがその活性化時に必ずしも状態データを使用または更新しない。

一例として、顧客の投資ポートフォリオを表すオブジェクトを考えてみましょう。このオブジェクトには、各投資に対応する複数の個別レコードが含まれているとします。特定のオペレーションがポートフォリオの 1 つの投資だけに影響を与える場合、そのオペレーションを使用して 1 つのレコードを読み取る方が、オブジェクトを呼び出すたびに投資ポートフォリオ全体を自動的に読み取る汎用の Tobj_ServantBase::activate_object() オペレーションを使用するより効率的です。

状態を持たないオブジェクトと永続状態

状態を持たないオブジェクト、つまり、method 活性化方針で定義されたオブジェクトの場合、次のことを保証する必要があります。

- 要求によって必要とされる永続状態が、オペレーションのビジネス・ロジックが実行されるまでにメモリに読み込まれること。
- 永続状態に対する変更が、呼び出しの終了までに外部に書き込まれること。

TP フレームワークは、活性化時にオブジェクトの `Tobj_ServantBase::activate_object()` オペレーションを呼び出します。オブジェクトが自身のディスク上の永続状態へのキーを含む OID を持っている場合、`Tobj_ServantBase::activate_object()` オペレーションはそのキーを OID から抽出するための唯一の機会をオブジェクトに提供します。

状態を持たないオブジェクトがトランザクションに参加できるようにする場合、一般に、そのオブジェクトの個々のメソッドの中でそのオブジェクトが永続状態をディスクに書き込むようにします。ただし、状態を持たないオブジェクトが常にトランザクションに関与する (このオブジェクトが呼び出されたときにトランザクションが常にスコープされる) 場合、`Tobj_ServantBase::deactivate_object()` オペレーションでデータベース書き込みオペレーションを処理することを選択できます。これは、データベース書き込みオペレーションを正確にコミットまたはロールバックするための信頼できるメカニズムが XA リソース・マネージャに用意されているからです。

注記 オブジェクトがメソッド・バウンドの場合でも、2 つのサーバ・プロセスが同じディスク・データに同時にアクセスする可能性を考慮に入れる必要があります。このような場合、並行性管理テクニックを使用できます。その最も簡単な方法は、トランザクションです。トランザクションとトランザクション・オブジェクトの詳細については、「第 6 章 トランザクションの CORBA サーバ・アプリケーションへの統合」を参照してください。

サーバント・プールと状態を持たないオブジェクト

サーバント・プールは、状態を持たないオブジェクトに対して特に便利な機能です。CORBA サーバ・アプリケーションがサーバントをプールすると、クライアントがオブジェクトを呼び出すたびにそのオブジェクトをインスタンス化するコストを大幅に低減できます。「サーバント・プール」の節で説明したとおり、プールされたサーバントはそのクライアント呼び出しの完了後もメモリに保持されます。アプリケーションで特定のオブジェクトを繰り返し呼び出す可能性があ

る場合、サーバントをプールすると、各クライアントに対してオブジェクトのメソッドではなくデータだけをメモリに読み取り、メモリから書き込むだけで済みます。オブジェクトのメソッドをメモリに読み取るコストが高い場合、サーバント・プールによってそのコストを低減できます。

サーバント・プールをインプリメントする方法については、「サーバント・プール」を参照してください。

状態を持つオブジェクトと永続状態

状態を持つオブジェクトの場合、必要な場合にのみ永続状態を読み書きする必要があります。この場合、次の最適化が必要になる場合があります。

- プロセス・バウンド・オブジェクトの場合、オブジェクトが長期間にわたって大量のメモリを割り当てる状態を回避します。
- トランザクション・バウンド・オブジェクトの場合、トランザクションの結果がわかっているときには、`Tobj_ServantBase::deactivate_object()` オペレーションが呼び出されるまで永続状態の書き込みを延期できます。

一般に、トランザクション・バウンド・オブジェクトは、すべてのデータベース書き込みまたはロールバック・オペレーションを自動的に処理する XA リソース・マネージャに依存する必要があります。

注記　トランザクションに関与するオブジェクトの場合、XA リソース・マネージャによって管理されない外部ストレージへのオブジェクトのデータ書き込みはサポートされません。

オブジェクトとトランザクションの詳細については、トランザクションの CORBA サーバ・アプリケーションへの統合を参照してください。

サーバント・プールと状態を持つオブジェクト

サーバント・プールは、プロセス・バウンド・オブジェクトに対しては意味を持ちません。ただし、アプリケーション設計によっては、サーバント・プールによってトランザクション・バウンド・オブジェクトの性能が向上する場合があります。

オブジェクトの非活性化の責任

前節で説明したとおり、Tobj_ServantBase::deactivate_object() オペレーションは、オブジェクトの永続状態をディスクに書き込むための手段としてインプリメントします。また、このオペレーションは、保持されているオブジェクト・データをメモリからフラッシュして、オブジェクトのサーバントでそのオブジェクトの別のインスタンスを活性化できるようにするためにも使用されます。ただし、オブジェクトの Tobj_ServantBase::deactivate_object() オペレーションの呼び出しによってそのオブジェクトのデストラクタも呼び出されると見なしてはなりません。

不必要な I/O の回避

オブジェクトで不必要な I/O を行うことによってアプリケーションの効率を下げないよう注意する必要があります。注意が必要な状態は以下のとおりです。

- オブジェクトの多くのオペレーションがオブジェクト状態を使用せず、オブジェクト状態に影響を与えない場合、これらのオペレーションを呼び出すたびに状態を読み書きするのは非効率です。設計を修正して、これらのオブジェクトが状態を必要とするオペレーションでのみ状態を処理するようにします。この場合、オブジェクトの活性化でそのすべての永続状態が読み取られないようにすることができます。
- オブジェクト状態が複数のオペレーションで読み込まれるデータで構成されている場合、次のいずれかを行うことによって、オブジェクトの呼び出し時に必要なオペレーションだけを行うようにします。
  - すべてのオペレーションに共通の状態だけを Tobj_ServantBase::activate_object() オペレーションで読み取ります。それ以外の状態の読み取りは、それらを必要とするオペレーションのみで行うようにします。
  - 変更された状態だけを書き込みます。そのためには、活性化中に変更されたデータを示すフラグを管理するか、ビフォー・データ・イメージとアフター・データ・イメージを比較します。

    一般的な最適化の方法は、活性化時に dirtyState フラグを初期化し、オブジェクトの活性化中にこのフラグが変更された場合にだけ Tobj_ServantBase::deactivate_object() オペレーションでデータを書き込むことです。ただし、これはオブジェクトが常に同じサーバント・プロセスで活性化されると保証できる場合にだけ使用できます。

活性化のサンプル

オブジェクトが活性化されるときに行われる一連のアクティビティの例については、『BEA Tuxedo CORBA アプリケーション入門』を参照してください。

デザイン・パターンの使い方

優れた設計に基づいてアプリケーションのビジネス・ロジックを構築することは重要です。BEA Tuxedo ソフトウェアには、このニーズを満たすデザイン・パターンのセットが用意されています。デザイン・パターンは、特定の設計問題に対する構造化されたソリューションです。デザイン・パターンの価値は、再利用およびほかのデザイン問題への適用が可能な形式で表現できることにあります。

BEA Tuxedo デザイン・パターンは、エンタープライズ・クラスのアプリケーション設計の問題に対する構造化されたソリューションです。これらを使用すると、大規模クライアント / サーバ・アプリケーションを適切に設計できます。

ここで説明するデザイン・パターンは、CORBA クライアント・アプリケーションおよびサーバ・アプリケーションで優れた設計手法を採用するためのガイドです。これらのデザイン・パターンは CORBA クライアント・アプリケーションおよびサーバ・アプリケーションの設計の重要な部分です。このマニュアルの各章には、これらのデザイン・パターンを使用して University サンプル・アプリケーションをインプリメントする方法が示されています。

Process-Entity デザイン・パターン

Process-Entity デザイン・パターンは、大規模なエンタープライズ・クラスのクライアント / サーバ・アプリケーションに適用されます。このデザイン・パターンは、『Object-Oriented Design Patterns』(Gamma ほか著) では Flyweight パターンと呼ばれ、ほかの出版物ではモデル / ビュー / コントローラと呼ばれています。

このパターンでは、クライアント・アプリケーションは存続期間の長いプロセス・オブジェクトを作成し、このオブジェクトと対話して要求を行います。たとえば、University サンプル・アプリケーションでは、このオブジェクトはクライアント・アプリケーションに代わってコース参照オペレーションを処理する登録係です。コース自体はデータベース・エンティティで、クライアント・アプリケーションからは見えません。

Process-Entity デザイン・パターンのメリットは次のとおりです。

- 細粒度オブジェクトをインプリメントせずに細粒度オブジェクトのメリットを実現できます。代わりに CORBA `struct` データ型を使用してオブジェクトをシミュレートします。
- 1 つのオブジェクト (プロセス・オブジェクト) だけがメモリにマップされるので、マシンのリソース利用が最適化されます。対照的に、大規模なデプロイメントでは、各データベース・エンティティが独立したオブジェクト・インスタンスとしてメモリに活性化される場合、処理する必要があるオブジェクトの数によってマシンのリソースが急激に消費されてしまいます。
- これらはクライアント・アプリケーションからは見えないので、データベース・エンティティを CORBA オブジェクトとしてインプリメントする必要がありません。代わりに、エンティティはサーバ・プロセス内のローカル言語オブジェクトとしてインプリメントできます。これは、3 層設計の基本的な原則です。ただし、これは多くのビジネスの運用方法を正確にモデル化します (現実の大学における登録係など)。大学で登録係を務める 1 人の人間は、複数の学生のために大規模なコース・データベースを処理できます。このため、学生ごとに登録係は必要ありません。このように、プロセス・オブジェクトの状態は、エンティティ・オブジェクトの状態とは区別されます。

Process-Entity デザイン・パターンの適用例については、「Basic CORBA サーバ・アプリケーションの設計とインプリメント」を参照してください。
Process-Entity デザイン・パターンの詳細については、『CORBA 技術情報』を参照してください。

List-Enumerator デザイン・パターン

List-Enumerator デザイン・パターンも、大規模なエンタープライズ・クラスのクライアント / サーバ・アプリケーションに適用されます。List-Enumerator デザイン・パターンは、BEA Tuxedo の主要機能であるアプリケーション制御のオブジェクト活性化を利用して、複数のクライアント呼び出しの間メモリ内で保持および追跡されるデータのキャッシュを処理し、不要になったときにそのデータをメモリからフラッシュします。

List-Enumerator デザイン・パターンの適用例については、「Basic CORBA サーバ・アプリケーションの設計とインプリメント」を参照してください。

List-Enumerator デザインをインプリメントするのに特に役立つツール、オブジェクトの事前活性化については、「状態を持つオブジェクトの事前活性化」を参照してください。

2 BEA Tuxedo CORBA サーバ・アプリケーションの作成手順

この章では、CORBA サーバ・アプリケーションを作成するための手順について説明します。この章で説明する手順は、決定的なものではありません。サーバ・アプリケーションによっては、ほかの手順を行う必要があります。また、これらの手順のいくつかについては順番を変更することができます。ただし、これらの手順は、すべての CORBA サーバ・アプリケーションの開発プロセスに共通のものです。

ここでは、次の内容について説明します。

- CORBA サーバ・アプリケーション開発プロセスの概要
- 開発とデバッグのヒント
- サーバント・プール
- デレゲーション・ベースのインターフェイス・インプリメンテーション

この章では、最初に手順の要約と、このマニュアルで使用する開発ツールおよび開発コマンドのリストを示します。デプロイメント環境によっては、ほかのソフトウェア開発ツールも使用します。このため、この章で説明するツールとコマンドも決定的なものではありません。

この章では、BEA Tuxedo ソフトウェアに付属の Basic University サンプル・アプリケーションの中の例を使用します。Basic University サンプル・アプリケーションの詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。このマニュアルで使用するツールとコマンドの詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

マルチスレッド CORBA サーバ・アプリケーションの作成については、「マルチスレッド CORBA サーバ・アプリケーションの作成」を参照してください。

CORBA サーバ・アプリケーション開発プロセスの概要

サーバ・アプリケーションを作成するための基本的な手順は次のとおりです。

BEA Tuxedo ソフトウェアには、次の開発ツールと開発コマンドが用意されています。

ツール	説明
IDL コンパイラ	アプリケーションの OMG IDL ファイルをコンパイルします。
`genicf`	インプリメンテーション・コンフィギュレーション・ファイル (ICF ファイル) を生成します。このファイルを修正して、非デフォルト・オブジェクトの活性化方針とトランザクション方針を指定できます。
`buildobjserver`	CORBA サーバ・アプリケーションの実行可能イメージを作成できます。
`tmloadcf`	`TUXCONFIG` ファイルを作成します。これは、サーバ・アプリケーションのコンフィギュレーションを指定する CORBA ドメイン用のバイナリ・ファイルです。
`tmadmin`	特に、トランザクション・アクティビティのログを作成します。これは、いくつかのサンプル・アプリケーションで使用されます。

ステップ 1: サーバ・アプリケーション用の OMG IDL ファイルのコンパイル

BEA Tuxedo ドメインで実行されるアプリケーションのクライアントおよびサーバ部分の基本構造は、そのアプリケーションの OMG IDL ファイルに記述される文によって決定されます。アプリケーションの OMG IDL ファイルをコンパイルする場合、IDL コンパイラは、`idl` コマンドで指定するオプションに応じて、次の図に示すファイルの一部または全部を生成します。影付きのコンポーネントは、サーバ・アプリケーションを作成するために修正するファイルです。

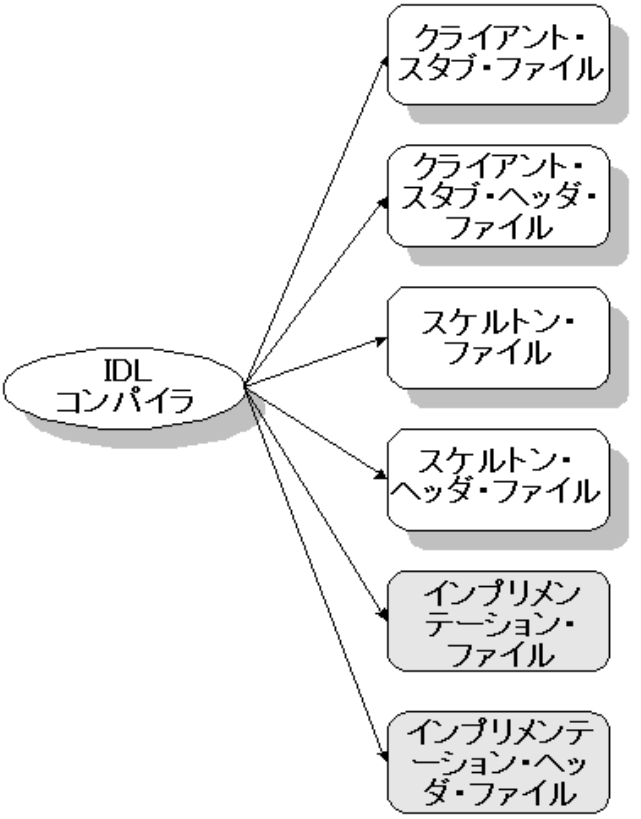


表 2-1 に、IDL コンパイラによって生成されるファイルを示します。

表 2-1 IDL コンパイラによって生成されるファイル

ファイル	デフォルト名	説明
クライアント・スタブ・ファイル	*application*_c.cpp	要求を送信するためのコードが格納されます。
クライアント・スタブ・ヘッダ・ファイル	*application*_c.h	OMG IDL ファイルで指定された各インターフェイスと各型に対応するクラス定義が格納されます。
スケルトン・ファイル	*application*_s.cpp	OMG IDL ファイルで指定された各インターフェイスに対応するスケルトンが格納されます。スケルトンは、実行時にクライアント要求をサーバ・アプリケーション内の適切なオペレーションにマップします。
スケルトン・ヘッダ・ファイル	*application*_s.h	スケルトン・クラス定義が格納されます。
インプリメンテーション・ファイル	*application*_i.cpp	OMG IDL ファイルで指定されたインターフェイスのオペレーションをインプリメントするメソッドのシグニチャが格納されます。
インプリメンテーション・ヘッダ・ファイル	*application*_i.h	OMG IDL ファイルで指定された各インターフェイスに対応する初期クラス定義が格納されます。

IDL コンパイラの使い方

表 2-1 に示したファイルを生成するには、次のコマンドを入力します。

```
idl [options] idl-filename [icf-filename]
```

idl コマンド構文のパラメータは次のとおりです。

- options は、IDL コンパイラへの 1 つまたは複数のコマンド行オプションを表します。コマンド行オプションについては、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。インプリメンテーション・ファイルを生成する場合は、-i オプションを指定する必要があります。

- *idl-filename* は、アプリケーションの OMG IDL ファイルの名前を表します。
- *icf-filename* は、アプリケーションのインプリメンテーション・コンフィギュレーション・ファイル (ICF ファイル) の名前を表すオプションのパラメータです。ICF ファイルは、オブジェクトの活性化方針を指定するか、または生成するスケルトンおよびインプリメンテーション・ファイルのインターフェイスの数を制限するために使用します。ICF ファイルの使い方については、「ステップ 4: オブジェクトのメモリ内での振る舞い」を参照してください。

注記 WebLogic Enterprise 5.1 では、プラグマの C++ IDL コンパイラ・インプリメンテーションが変更され、CORBA 2.3 機能がサポートされるようになりました。これにより、IDL ファイルが影響を受ける場合があります。CORBA 2.3 機能は、プラグマ接頭辞定義が影響を及す可能性があるスコープを変更します。プラグマはインクルードされる IDL ファイルに含まれる定義には影響を及さず、またインクルードされる IDL ファイル内で行われるプラグマ接頭辞定義はそれらのファイルの外部のオブジェクトに影響を及しません。

C++ IDL コンパイラは、プラグマ接頭辞の処理を訂正するよう変更されました。この変更はオブジェクトのリポジトリ ID に影響を与えるため、`_narrow` などのオペレーションでエラーが発生する場合があります。

このようなエラーを防ぐには、次のことを行います。

- IDL をリポジトリにリロードする場合、アプリケーションのクライアント・スタブとサーバ・スケルトンを再生成する必要があります。
- 任意のクライアント・スタブまたはサーバ・スケルトンを再生成する場合、アプリケーションのすべてのスタブとスケルトンを再生成し、IDL をインターフェイス・リポジトリにリロードする必要があります。

IDL コンパイラと `idl` コマンドの詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

スケルトンとインプリメンテーション・ファイルの生成

次のコマンド行は、OMG IDL ファイル `univb.idl` のクライアント・スタブ・ファイル、スケルトン・ファイル、初期インプリメンテーション・ファイル、スケルトン・ヘッダ・ファイル、およびインプリメンテーション・ヘッダ・ファイルを生成します。

```
idl -i univb.idl
```

`idl` コマンドの詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。BEA Tuxedo University サンプル・アプリケーション用のこれらのファイルを生成する方法については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。

注記 非デフォルトのオブジェクトの活性化方針またはトランザクション方針を指定する場合、または生成するスケルトン・ファイルとインプリメンテーション・ファイルのインターフェイスの数を制限する場合、インプリメンテーション・コンフィギュレーション・ファイル (ICF) を生成し、この ICF ファイルを IDL コンパイラに受け渡す必要があります。詳細については、ICF でオブジェクトの活性化方針とトランザクション方針を指定する方法を参照してください。

tie クラスの生成

IDL コンパイラには、インターフェイスの tie クラス・テンプレートを生成するために使用できる `-T` コマンド行オプションも用意されています。CORBA アプリケーションの tie クラスのインプリメントについては、「デレゲーション・ベースのインターフェイス・インプリメンテーション」を参照してください。

ステップ 2: 各インターフェイスのオペレーションをインプリメントするメソッドの記述

サーバ・アプリケーション・プログラマは、アプリケーションの OMG IDL ファイルで定義した各インターフェイスのオペレーションをインプリメントするメソッドを記述します。

インプリメンテーション・ファイルには以下のものが含まれます。

- OMG IDL ファイルで指定された各オペレーションのメソッド宣言
- アプリケーションのビジネス・ロジック、インクルード・ファイル、およびアプリケーションで使用するその他のデータ
- 各インターフェイス・インプリメンテーションのコンストラクタとデストラクタ (これらのインプリメントはオプション)
- オプションで、`Tobj_ServantBase::activate_object()` オペレーションと `Tobj_ServantBase::deactivate_object()` オペレーション

  `Tobj_ServantBase::activate_object()` オペレーションと `Tobj_ServantBase::deactivate_object()` オペレーションの中で、オブジェクトの活性化または非活性化に関連する特定の手順を実行するコードを記述します。これには、オブジェクトの永続状態のディスクからの読み取りとディスクへの書き込みがそれぞれ含まれます。これらのオペレーションをオブジェクトにインプリメントする場合、インプリメンテーション・ヘッダ・ファイルを編集して、これらのオペレーションを使用する各インプリメンテーションにそれらの定義を追加する必要があります。

IDL コンパイラによって生成されるインプリメンテーション・ファイル

サーバ・アプリケーションのインプリメンテーション・ファイルはすべて手動で作成できますが、インプリメンテーション・ファイルを記述する出発点として、IDL コンパイラが生成するインプリメンテーション・ファイルを使用することもできます。IDL コンパイラによって生成されるインプリメンテーション・ファイルには、アプリケーションのインターフェイス用に定義された各オペレーションをインプリメントするメソッドのシグニチャが含まれます。

通常、このインプリメンテーション・ファイルは、IDL コンパイラを呼び出すコマンドで `-i` オプションを指定することによって 1 度だけ作成します。アプリケーションのインターフェイスを繰り返し修正し、これらのインターフェイスのオペレーション (オペレーション・シグニチャを含む) を変更した場合は、必要なすべての変更をインプリメンテーション・ファイルに追加してこれらの変更を反映させる必要があります。

ファクトリのインプリメント

「クライアント・アプリケーションがサーバ・アプリケーションの CORBA オブジェクトをアクセスおよび操作する方法」で説明したように、クライアント・アプリケーションがサーバ・アプリケーションによって管理されるオブジェクトを簡単に検索できるようにするためには、ファクトリを作成する必要があります。ファクトリはインプリメントするほかの CORBA オブジェクトに似ていますが、FactoryFinder オブジェクトに登録する必要があります。ファクトリの登録については、「ファクトリを作成および登録するコードの記述」を参照してください。

ファクトリの主要な機能は、オブジェクト・リファレンスを作成することです。ファクトリは、`TP::create_object_reference()` オペレーションを呼び出すことによってこれを実行します。`TP::create_object_reference()` オペレーションでは、次の入力パラメータが必要となります。

- オブジェクトの OMG IDL インターフェイスのインターフェイス・リポジトリ ID
- 文字列形式のオブジェクト ID (OID)
- ルーティング基準 (オプション)

たとえば、Basic University サンプル・アプリケーションでは、RegistrarFactory インターフェイスは、次のように 1 つのオペレーションだけを指定します。

```
University::Registrar_ptr RegistrarFactory_i::find_registrar()
```

RegistrarFactory オブジェクトの find_registrar() オペレーションには、Registrar オブジェクトのリファレンスを作成するための TP::create_object_reference() オペレーションへの次の呼び出しが含まれています。

```
CORBA::Object_var v_reg_oref =
            TP::create_object_reference(
                University::_tc_Registrar->id(),
                object_id,
                CORBA::NVlist::_nil()
            );
```

このコード・サンプルでは、次のことに留意してください。

- 次のパラメータは、Registrar オブジェクトのインターフェイス・リポジトリ ID をタイプ・コードから抽出することによってその ID を指定します。

  ```
  University::_tc_Registrar->id()
  ```

- 次のパラメータは、ルーティング基準が使用されないことを指定します。この結果、Registrar オブジェクト用に作成されるオブジェクト・リファレンスは、それを作成する RegistrarFactory オブジェクトと同じグループにルーティングされます。

  ```
  CORBA::NVlist::_nil()
  ```

  オブジェクト・リファレンスのルーティング先のグループに影響を与えるルーティング基準の指定については、「BEA Tuxedo の CORBA サーバ・アプリケーションのスケーリング」を参照してください。

ステップ 3: Server オブジェクトの作成

Server オブジェクトのインプリメントは、ほかの言語オブジェクトのインプリメントとは異なります。Server オブジェクトのヘッダ・クラスは既に作成されており、Server オブジェクト・クラスは既にインスタンス化されています。Server オ

ブジェクトを作成するには、パッケージ済みの Server オブジェクト・クラスの特定のメソッド・セットをインプリメントします。この節では、インプリメントするこれらのメソッドについて説明します。

Server オブジェクトを作成するには、一般のテキスト・エディタで新しいファイルを作成し、次のオペレーションをインプリメントします。

オペレーション	説明
`Server::initialize();`	サーバ・アプリケーションの起動後、TP フレームワークはサーバ・アプリケーション初期化プロセスの最後のステップとしてこのオペレーションを呼び出します。このオペレーションでは、特定のサーバ・アプリケーションで必要な複数の初期化タスクを実行します。このオペレーションで提供するものについては、「サーバ・アプリケーションの初期化」を参照してください。
`Server::create_servant();`	既存のサーバントで処理できないクライアント要求が送信されると、TP フレームワークはこのオペレーションを呼び出して、活性化される CORBA オブジェクトの OMG IDL インターフェイスのインターフェイス・リポジトリ ID を受け渡します。このオペレーションで提供するものについては、「サーバントの作成」を参照してください。
`Server::release();`	TP フレームワークは、サーバ・アプリケーションをシャットダウンするときにこのオペレーションを呼び出します。このオペレーションには、サーバ・アプリケーションで管理されるオブジェクト・ファクトリの登録を削除し、ほかのシャットダウン・タスクを実行するためのコードが含まれます。このオペレーションで提供するものについては、「サーバ・アプリケーションのリリース」を参照してください。

どのサーバ・アプリケーションにも、Server オブジェクトのインスタンスは 1 つしか存在しません。サーバ・アプリケーションが複数の CORBA オブジェクト・インプリメンテーションを管理する場合、記述する `Server::initialize()`、`Server::create_servant()`、および `Server::release()` オペレーションにはこれらのインプリメンテーションすべてに適用するコードを組み込む必要があります。

これらのタスクの大部分のコードには、TP フレームワークとの対話が含まれます。以降の節では、これらの Server オブジェクト・オペレーションのそれぞれに対して必要なコードについて説明し、Basic University サンプル・アプリケーションのサンプル・コードを示します。

サーバ・アプリケーションの初期化

Server オブジェクトに実装する最初のオペレーションは、サーバ・アプリケーションを初期化するオペレーションです。このオペレーションは、BEA Tuxedo システムがサーバ・アプリケーションを起動するときに呼び出されます。TP フレームワークは、サーバ・アプリケーションの起動シーケンス中に Server オブジェクトの次のオペレーションを呼び出します。

```
CORBA::Boolean Server::initialize(int argc, char** argv)
```

BEA Tuxedo ドメインの UBBCONFIG ファイルの SERVERS セクションに指定する特定のサーバ・アプリケーション用の CLOPT パラメータは、Server::initialize() オペレーションに argc および argv として受け渡されます。サーバ・アプリケーションへの引数の受け渡しについては、『BEA Tuxedo アプリケーション実行時の管理』を参照してください。サーバ・アプリケーションに引数を受け渡す例については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。

Server::initialize() オペレーションの中には、該当する場合、次のことを行うコードを組み込みます。

- ファクトリの作成と登録
- マシン・リソースの割り当て
- サーバ・アプリケーションで必要なグローバル変数の初期化
- サーバ・アプリケーションによって使用されるデータベースのオープン
- XA リソース・マネージャのオープン

ファクトリを作成および登録するコードの記述

クライアント・アプリケーションがオブジェクトを簡単に検索できるようにするためのファクトリをサーバ・アプリケーションが管理する場合、そのファクトリを FactoryFinder オブジェクトに登録するコードを記述する必要があります。このコードは、通常サーバ・アプリケーション初期化プロセスの最後のステップとして呼び出されます。

サーバ・アプリケーションによって管理されるファクトリを登録するコードを記述するには、次のことを行います。

1. ファクトリのオブジェクト・リファレンスを作成します。

   このステップでは、「ファクトリのインプリメント」で説明したとおりオブジェクト・リファレンスを作成します。このステップでは、`TP::create_object_reference()` オペレーションの呼び出しを組み込み、OMG IDL インターフェイスのインターフェイス・リポジトリ ID を指定します。次の例では、`RegistrarFactory` ファクトリのオブジェクト・リファレンス (`s_v_fact_ref` 変数で表される) が作成されます。

   ```
   University::RegistrarFactory s_v_fact_ref =
        TP::create_object_reference(
            University::_tc_RegistrarFactory->id(),
            object_id,
            CORBA::NVList::_nil()
        );
   ```

2. BEA Tuxedo ドメインにファクトリを登録します。

   このステップでは、サーバ・アプリケーションによって管理される各ファクトリに対して次のオペレーションを呼び出します。

   ```
   TP::register_factory (CORBA::Object_ptr factory_or,
                         const char* factory_id);
   ```

   `TP::register_factory()` オペレーションは、サーバ・アプリケーションのファクトリを FactoryFinder オブジェクトに登録します。このオペレーションでは、次の入力パラメータが必要です。

   - 前述のステップ 1 で作成した、ファクトリのオブジェクト・リファレンス。
   - ファクトリ・オブジェクトのインターフェイス・タイプ・コードに基づく文字列識別子。これは、ファクトリの OMG IDL インターフェイスのインターフェイス・リポジトリ ID を識別するために使用されます。

次の例では、RegistrarFactory ファクトリが BEA Tuxedo ドメインに登録されます。

```
TP::register_factory(s_v_fact_ref.in(),
                     University::_tc_RegistrarFactory->id());
```

University::_tc_RegistrarFactory->id() パラメータに注目してください。これは、TP::create_object_reference() オペレーションで指定したパラメータと同じです。このパラメータは、オブジェクトの OMG IDL インターフェイスのインターフェイス・リポジトリ ID をそのタイプ・コードから抽出します。

サーバントの作成

サーバ・アプリケーション初期化プロセスが完了したら、サーバ・アプリケーションはクライアント要求を処理できる状態になります。CORBA オブジェクトのオペレーションに対する要求が到着し、それを処理するサーバントがメモリ内に存在しない場合、TP フレームワークは Server オブジェクトの次のオペレーションを呼び出します。

```
Tobj_Servant Server::create_servant(const char* interfaceName)
```

Server::create_servant() オペレーションには、クライアント要求によって必要とされるオブジェクトのサーバントをインスタンス化するコードが含まれます。たとえば、C++ では、このコードにはオブジェクトのインターフェイス・クラスの new 文が組み込まれます。

Server::create_servant() オペレーションでは、サーバントは OID に関連付けられません。サーバントと OID の関連付けは、TP フレームワークがそのサーバントの Tobj_ServantBase::activate_object() オペレーション (オブジェクトのインスタンス化を完了させるオペレーション) を呼び出したときに行われます。オブジェクトのコンストラクタで OID をオブジェクトに関連付けることはできません。同様に、サーバントと OID の関連付けの解除は、TP フレームワークがそのサーバントの deactivate_object() オペレーションを呼び出したときに行われます。

BEA Tuxedo システムでのこうしたサーバントの振る舞いにより、TP フレームワークは、オブジェクトの非活性化後、別のオブジェクトのインスタンス化のためにサーバントを使用できます。したがって、オブジェクトの Tobj_ServantBase::deactivate_object() オペレーションの呼び出しによってそのオブジェクトのデストラクタも呼び出されるとは考えないでください。

サーバ・アプリケーションでサーバント・プール機能を使用する場合、オブジェクトの Tobj_ServantBase::deactivate_object() オペレーションに TP::application_responsibility() オペレーションをインプリメントして、サーバントへのポインタを後で使用できるようにサーバント・プールに受け渡すことができます。サーバント・プールについては、「サーバント・プール」を参照してください。

Server::create_servant() オペレーションでは、入力引数が 1 つ必要です。この引数は、サーバントを作成するためのオブジェクトの OMG IDL インターフェイスのインターフェイス リポジトリ ID を含む文字列を指定します。

このオペレーション用に記述するコードには、サーバ・アプリケーションによって管理されるオブジェクトの OMG IDL インターフェイスのインターフェイス・リポジトリ ID を指定します。実行時に、Server::create_servant() オペレーションは、要求によって指定されたオブジェクトに対して必要なサーバントを返します。

次のコードは、Basic University サンプル・アプリケーションの University サーバ・アプリケーションの Server::create_servant() オペレーションをインプリメントします。

```
Tobj_Servant Server::create_servant(const char* intf_repos_id)
{
  if (!strcmp(intf_repos_id, University::_tc_RegistrarFactory->id())) {
      return new RegistrarFactory_i();
  }
  if (!strcmp(intf_repos_id, University::_tc_Registrar->id())) {
      return new Registrar_i();
  }
  if (!strcmp(intf_repos_id, University::_tc_CourseSynopsisEnumerator->id())) {
      return new CourseSynopsisEnumerator_i();
  }
  return 0; // 未知のインターフェイス
}
```

サーバ・アプリケーションのリリース

BEA Tuxedo システム管理者が tmshutdown コマンドを入力すると、TP フレームワークは、BEA Tuxedo ドメインで実行される各サーバ・アプリケーションの Server オブジェクトの次のオペレーションを呼び出します。

```
void Server::release()
```

Server::release() オペレーションでは、次のような、サーバ・アプリケーションに応じたアプリケーション固有のクリーンアップ・タスクを実行できます。

- サーバ・アプリケーションによって管理されるオブジェクト・ファクトリの登録の削除
- リソースの割り当て解除
- データベースのクローズ
- XA リソース・マネージャのクローズ

サーバ・アプリケーションがシャットダウン要求を受信すると、そのサーバ・アプリケーションはほかのリモート・オブジェクトから要求を受信することができなくなります。これは、管理者がサーバ・アプリケーションをシャットダウンする順番に影響を与えます。たとえば、あるサーバ・プロセスへの Server::release() オペレーションの呼び出しが別のサーバ・プロセスに含まれている場合、最初のサーバ・プロセスはシャットダウンしてはなりません。

サーバのシャットダウン中に、次の呼び出しを組み込んでサーバ・アプリケーションの各ファクトリの登録を削除できます。

```
TP::unregister_factory (CORBA::Object_ptr factory_or,
                        const char* factory_id)
```

TP::unregister_factory() オペレーションの呼び出しは、Server::release() インプリメンテーションの最初のアクションの 1 つである必要があります。TP::unregister_factory() オペレーションは、サーバ・アプリケーションのファクトリへの登録を削除します。このオペレーションでは、次の入力引数が必要です。

- ファクトリのオブジェクト・リファレンス
- ファクトリ・オブジェクトのインターフェイス・タイプ・コードに基づく文字列識別子。これは、ファクトリの OMG IDL インターフェイスのインターフェイス・リポジトリ ID を識別するために使用されます。

次の例では、Basic サンプル・アプリケーションで使用されている RegistrarFactory ファクトリへの登録が削除されます。

```
TP::unregister_factory(s_v_fact_ref.in(), UnivB::_tc_RegistrarFactory->id());
```

このコード例では、グローバル変数 s_v_fact_ref の使い方に注目してください。この変数は、RegistrarFactory オブジェクトを登録した Server::initialize() オペレーションで設定されたもので、ここで再び使用されます。

また、UnivB::_tc_RegistrarFactory->id() パラメータにも注目してください。これも、ファクトリの登録に使用されたインターフェイス名と同じです。

ステップ 4: オブジェクトのメモリ内での振る舞い

「オブジェクト状態の管理」で説明したように、オブジェクトの活性化方針とトランザクション方針を割り当て、オプションでアプリケーション制御の非活性化機能を使用することによって、オブジェクトを非活性化するイベントを指定します。

オブジェクトの活性化方針とトランザクション方針は ICF ファイルに指定し、アプリケーション制御の非活性化は TP::deactivateEnable() オペレーションを介して使用します。この節では、Basic University サンプル・アプリケーションを例として使用して、これらのメカニズムをインプリメントする方法について説明します。

ここでは、次のことについて説明します。

- ICF でオブジェクトの活性化方針とトランザクション方針を指定する方法
- アプリケーション制御の非活性化をインプリメントする方法

ICF でオブジェクトの活性化方針とトランザクション方針を指定する方法

BEA Tuxedo ソフトウェアは、「オブジェクトの活性化方針」で説明した次の活性化方針をサポートしています。

活性化方針	説明
`method`	オブジェクトは、自身のオペレーションの 1 つの呼び出しの間だけ活性化されます。
`transaction`	オブジェクトは、自身のオペレーションが呼び出されたときに活性化されます。オブジェクトがトランザクションのスコープ内で活性化された場合、オブジェクトはトランザクションがコミットまたはロールバックされるまでアクティブになります。
`process`	オブジェクトは、自身のオペレーションが呼び出されたときに活性化され、次のいずれかの状態が発生した場合にのみ非活性化されます。 ■ サーバ・アプリケーションが存在するプロセスがシャットダウンされた。 ■ オブジェクトが自身の `TP::deactivateEnable()` オペレーションを呼び出した。

BEA Tuxedo ソフトウェアは、「トランザクションの CORBA サーバ・アプリケーションへの統合」で説明する次のトランザクション方針もサポートしています。

トランザクション方針	説明
always	このオブジェクトのオペレーションが呼び出されると、この方針によって TP フレームワークはそのオブジェクトのトランザクションを開始します (活性化されているトランザクションが存在しない場合)。TP フレームワークがトランザクションを開始した場合、TP フレームワークは、オペレーションが正常に完了した場合はトランザクションをコミットし、例外が発生した場合はトランザクションをロールバックします。
optional	このオブジェクトのオペレーションが呼び出されると、この方針によって TP フレームワークはこのオブジェクトをトランザクションに組み込みます (トランザクションが活性化されている場合)。活性化されているトランザクションが存在しない場合、このオブジェクトに対して定義されている活性化方針に従ってこのオブジェクトの呼び出しが続行されます。 このトランザクション方針はデフォルトです。
never	TP フレームワークは、このオブジェクトがトランザクション中に呼び出された場合にエラー状態を生成します。
ignore	このオブジェクトが呼び出されたときにトランザクションが活性化されている場合、そのトランザクションはオペレーション呼び出しが完了するまで中断します。このトランザクション方針により、この方針が割り当てられているオブジェクトにトランザクションが伝達されるのを防ぐことができます。

これらの方針をアプリケーションのオブジェクトに割り当てるには、次の手順に従います。

1. 次の例のように、genicf コマンドを入力し、アプリケーションの OMG IDL ファイルを入力として指定して、ICF ファイルを生成します。

   ```
   # genicf university.idl
   ```

   このコマンドによって、university.icf というファイルが生成されます。

2. ICF ファイルを編集して、アプリケーションのインターフェイスごとに活性化方針を指定します。次の例に、Basic University サンプル・アプリケーション用に生成された ICF ファイルを示します。デフォルトのオブジェクトの活性化方針は `method` で、デフォルトのトランザクション活性化方針は `optional` であることに注意してください。

```
module POA_UniversityB
    {
    implementation CourseSynopsisEnumerator_i
        {
        activation_policy ( method );
        transaction_policy ( optional );
        implements ( UniversityB::CourseSynopsisEnumerator );
        };
    };
module POA_UniversityB
    {
    implementation Registrar_i
        {
        activation_policy ( method );
        transaction_policy ( optional );
        implements ( UniversityB::Registrar );
        };
    };
module POA_UniversityB
    {
    implementation RegistrarFactory_i
        {
        activation_policy ( method );
        transaction_policy ( optional );
        implements ( UniversityB::RegistrarFactory );
        };
    };
```

3. 生成するスケルトン・ファイルとインプリメンテーション・ファイルのインターフェイスの数を制限する場合、それらのインターフェイスをインプリメントするインプリメンテーション・ブロックを ICF ファイルから削除できます。前掲の ICF コードの例では、`RegistrarFactory` インターフェイスのスケルトン・ファイルとインプリメンテーション・ファイルが生成されないようにするには、次の行を削除します。

```
implementation RegistrarFactory_i
        {
        activation_policy ( method );
        transaction_policy ( optional );
        implements ( UniversityB::RegistrarFactory );
        };
```

4. ICF ファイルを IDL コンパイラに受け渡して、指定された方針に対応するスケルトン・ファイルとインプリメンテーション・ファイルを生成します。詳細については、「スケルトンとインプリメンテーション・ファイルの生成」を参照してください。

ステップ 5: サーバ・アプリケーションのコンパイルとリンク

Server オブジェクトとオブジェクト・インプリメンテーションのコードの記述が済んだら、サーバ・アプリケーションをコンパイルおよびリンクします。

CORBA サーバ・アプリケーションをコンパイルおよびリンクするには、`buildobjserver` コマンドを使用します。`buildobjserver` コマンドの形式は次のとおりです。

```
buildobjserver [-o servername] [options]
```

`buildobjserver` コマンド構文のパラメータは次のとおりです。

- `-o servername` は、このコマンドによって生成されるサーバ・アプリケーションの名前を表します。
- `options` は、`buildobjserver` コマンドへのコマンド行オプションを表します。

University サンプル・アプリケーションのコンパイルとリンクの詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。`buildobjserver` コマンドの詳細については、『BEA Tuxedo コマンド・リファレンス』を参照してください。

マルチスレッド CORBA サーバ・アプリケーションの設計とビルドには、特別な考慮事項が存在します。詳細については、「buildobjserver コマンドの使い方」を参照してください。

注記 IBM AIX 4.3.3 システム上で BEA Tuxedo ソフトウェアを実行する場合、`-brtl` コンパイラ・オプションを使用して CORBA アプリケーションを再コンパイルする必要があります。

ステップ 6: サーバ・アプリケーションのデプロイ

システム管理者は、この節で説明する手順を使用して CORBA サーバ・アプリケーションをデプロイします。University サンプル・アプリケーションのビルドとデプロイの詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。

サーバ・アプリケーションをデプロイするには、以下の手順に従います。

1. サーバ・アプリケーションの実行可能ファイルを、目的の BEA Tuxedo ドメインの一部であるマシンの適切なディレクトリに格納します。
2. 一般のテキスト・エディタを使用して、アプリケーションのコンフィギュレーション・ファイル (UBBCONFIG ファイル) を作成します。
3. CORBA サーバ・アプリケーションを起動するマシンで、次の環境変数を設定します。
   - TUXCONFIG。この変数は、UBBCONFIG ファイルの TUXCONFIG エントリと正確に一致する必要があります。この変数は、アプリケーションの UBBCONFIG ファイルの格納場所またはパスを表します。
   - APPDIR。この変数は、アプリの実行可能ファイルが存在するディレクトリを表します。
4. BEA Tuxedo ドメインで稼働しているすべてのマシン、または BEA Tuxedo ドメインに接続されているすべてのマシンで、TUXDIR 環境変数を設定します。この環境変数は、BEA Tuxedo ソフトウェアがインストールされている場所を指し示します。
5. 次のコマンドを入力して、TUXCONFIG ファイルを作成します。

   ```
   tmloadcf -y application-ubbconfig-file
   ```

   コマンド行引数の *application*-ubbconfig-*file* は、アプリケーションの UBBCONFIG ファイルの名前を表します。古い TUXCONFIG ファイルを削除してからこのコマンドを実行しなければならない場合があることに注意してください。

6. 次のコマンドを入力して、CORBA サーバ・アプリケーションを起動します。

   ```
   tmboot -y
   ```

   UBBCONFIG ファイルをリロードせずにサーバ・アプリケーションを再起動できます。

University サンプル・アプリケーションの詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。
CORBA アプリケーション用の UBBCONFIG ファイルの作成の詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

開発とデバッグのヒント

ここでは、次の内容について説明します。

- CORBA 例外とユーザ・ログの使い方
- コールバック・メソッドのエラー状態の検出
- OMG IDL インターフェイスのバージョンと修正に関する一般的な注意事項
- `Tobj_ServantBase::deactivate_object()` オペレーションでの状態処理に関する注意

CORBA 例外とユーザ・ログの使い方

ここでは、次の内容について説明します。

- 例外のクライアント・アプリケーション・ビュー
- 例外のサーバ・アプリケーション・ビュー

例外のクライアント・アプリケーション・ビュー

クライアント・アプリケーションが CORBA オブジェクトのオペレーションを呼び出した場合、その呼び出しの結果として例外が返される場合があります。クライアント・アプリケーションに返される有効な例外は次のとおりです。

- すべての CORBA 準拠の ORB によって認識される標準の CORBA 定義例外
- OMG IDL で定義され、クライアント・アプリケーションがそのスタブまたはインターフェイス・リポジトリを介して認識する例外

BEA Tuxedo システムは、これらの CORBA 定義の制限に違反することがないよう動作します。詳細については、「例外のサーバ・アプリケーション・ビュー」で説明します。

クライアント・アプリケーションが認識する例外セットは制限されているので、クライアント・アプリケーションは原因が不明な例外をキャッチする場合があります。BEA Tuxedo システムは、こうした例外を可能な限りユーザ・ログの説明メッセージで補足します。このメッセージは、エラー状態の検出とデバッグに役立ちます。これらのケースについては、次の節で説明します。

例外のサーバ・アプリケーション・ビュー

ここでは、次の内容について説明します。

- BEA Tuxedo システムによって生成され、アプリケーション・コードによってキャッチされる例外
- CORBA オブジェクトのオペレーションの呼び出し中にアプリケーション・コードによって生成された例外を BEA Tuxedo システムが処理する方法

BEA Tuxedo システムによって生成され、アプリケーション・コードによってキャッチされる例外

BEA Tuxedo システムは、TP オブジェクトのオペレーションが呼び出された場合、次の例外をアプリケーションに返す場合があります。

- CORBA 定義のシステム例外
- `TobjS_c.h` ファイルに定義される CORBA `UserExceptions`。このファイルに定義される例外の OMG IDL は次のとおりです。

```
interface TobjS {
          exception AlreadyRegistered { };
          exception ActivateObjectFailed { string reason; };
          exception ApplicationProblem { };
          exception CannotProceed { };
          exception CreateServantFailed { string reason; };
          exception DeactivateObjectFailed { string reason; };
          exception IllegalInterface { };
```

```
                exception IllegalOperation { };
                exception InitializeFailed { string reason; };
                exception InvalidDomain { };
                exception InvalidInterface { };
                exception InvalidName { };
                exception InvalidObject { };
                exception InvalidObjectId { };
                exception InvalidServant { };
                exception NilObject { string reason; };
                exception NoSuchElement { };
                exception NotFound { };
                exception OrbProblem { };
                exception OutOfMemory { };
                exception OverFlow { };
                exception RegistrarNotAvailable { };
                exception ReleaseFailed { string reason; };
                exception TpfProblem { };
                exception UnknownInterface { };
}
```

CORBA オブジェクトのオペレーションの呼び出し中にアプリケーション・コードによって生成された例外を BEA Tuxedo システムが処理する方法

サーバ・アプリケーションは、クライアント呼び出し中に次の場所で例外を生成する場合があります。

- `Server::create_servant`、`Tobj_ServantBase::activate_object()`、および `Tobj_ServantBase::deactivate_object()` コールバック・メソッド
- 呼び出されたオペレーションのインプリメンテーション・コード

サーバ・アプリケーションは次のタイプの例外を生成する可能性があります。

- CORBA 定義のシステム例外
- OMG IDL に定義される CORBA ユーザ定義例外
- `TobjS_c.h` ファイルに定義される CORBA ユーザ定義例外。次の例外は、BEA Tuxedo システムがユーザ・ログにメッセージを送るためにサーバ・アプリケーションで使用されます。このメッセージは、トラブルシューティングに役立ちます。

```
interface TobjS {
                exception ActivateObjectFailed { string reason; };
                exception CreateServantFailed { string reason; };
                exception DeactivateObjectFailed { string reason; };
                exception InitializeFailed { string reason; };
```

```
            exception ReleaseFailed { string reason; };
};
```

- その他の C++ 例外型

サーバ・アプリケーション・コードによって生成され、サーバ・アプリケーションによってキャッチされないすべての例外は、BEA Tuxedo システムによってキャッチされます。これらの例外がキャッチされると、次のいずれかの処理が発生します。

- 例外は変換されずにクライアント・アプリケーションに返されます。
- 例外は標準 CORBA 例外に変換され、クライアント・アプリケーションに返されます。
- 例外は標準 CORBA 例外に変換され、次のアクションが発生します。
  - 例外がクライアント・アプリケーションに返されます。
  - エラーに関する説明情報を含む 1 つまたは複数のメッセージがユーザ・ログに送られます。この説明情報は、サーバ・アプリケーション・コードか、または BEA Tuxedo システムから生成されます。

以下の節では、CORBA オブジェクトに対するクライアント呼び出し中にサーバ・アプリケーションによって生成される例外を BEA Tuxedo システムがどのように処理するかについて説明します。

`Server::create_servant()` オペレーションで生成された例外

例外が `Server::create_servant()` オペレーションで生成された場合、次の処理が行われます。

- `CORBA::OBJECT_NOT_EXIST` 例外がクライアント・アプリケーションに返されます。
- 生成された例外が `TobjS::CreateServantFailed` の場合、メッセージがユーザ・ログに送られます。例外のコンストラクタに reason 文字列が指定されている場合、その reason 文字列もメッセージの一部として書き込まれます。
- `Tobj_ServantBase::activate_object()` オペレーションと `Tobj_ServantBase::deactivate_object()` オペレーションは呼び出されません。クライアントによって要求されたオペレーションは呼び出されません。

Tobj_ServantBase::activate_object() オペレーションで生成された例外

例外が Tobj_ServantBase::activate_object() オペレーションで生成された場合、次の処理が行われます。

- CORBA::OBJECT_NOT_EXIST 例外がクライアント・アプリケーションに返されます。
- 生成された例外が TobjS::ActivateObjectFailed の場合、メッセージがユーザ・ログに送られます。例外のコンストラクタに reason 文字列が指定されている場合、その reason 文字列もメッセージの一部として書き込まれます。
- クライアントによって要求されたオペレーションと Tobj_ServantBase::deactivate_object() オペレーションは呼び出されません。

オペレーション・インプリメンテーションで生成された例外

BEA Tuxedo システムでは、オペレーション・インプリメンテーションは、CORBA システム例外、またはクライアント・アプリケーションによって認識され、OMG IDL に定義されるユーザ定義例外のいずれかをスローする必要があります。これらのタイプの例外がオペレーション・インプリメンテーションによってスローされた場合、BEA Tuxedo システムは次のいずれかの状態が存在しない限り、それらをクライアント・アプリケーションに返します。

- オブジェクトに always トランザクション方針が設定されており、BEA Tuxedo システムがオブジェクトの呼び出し時に自動的にトランザクションを開始した場合。この場合、トランザクションは BEA Tuxedo システムによって自動的にロールバックされます。クライアント・アプリケーションはこのトランザクションを認識しないので、BEA Tuxedo システムは、クライアントがトランザクションを開始した場合に生成される CORBA::TRANSACTION_ROLLEDBACK 例外ではなく CORBA::OBJ_ADAPTER CORBA システム例外を生成します。
- 例外が TobjS_c.h ファイルに定義されている場合。この場合、例外は CORBA::BAD_OPERATION 例外に変換され、クライアント・アプリケーションに返されます。また、次のメッセージがユーザ・ログに送られます。

  "WARN: Application didn't catch TobjS exception. TP Framework throwing CORBA::BAD_OPERATION."

例外が TobjS::IllegalOperation の場合、次の補足メッセージが書き込まれ、アプリケーションにコーディング・エラーが存在する可能性があることを開発者に警告します。

```
"WARN: Application called TP::deactivateEnable() illegally and
didn't catch TobjS exception."
```

これは、TP::deactivateEnable() オペレーションが、transaction 活性化方針が割り当てられているオブジェクトの内部で呼び出された場合に発生する可能性があります。アプリケーション制御の非活性化はトランザクション・バウンド・オブジェクトではサポートされません。

- BEA Tuxedo システムが、クライアント呼び出しに続いて内部システム例外を生成した場合。この場合、CORBA::INTERNAL 例外がクライアントに返されます。一般にこれは、オブジェクトが活性化されているプロセスに関する重大なシステム問題を示します。

CORBA 仕様で定義されているとおり、クライアントに送り返される応答には、オペレーション・インプリメンテーションからの結果値か、オペレーション・インプリメンテーションでスローされた例外のいずれかが含まれ、両方が含まれることはありません。最初のケース、つまり、応答ステータス値が NO_EXCEPTION– の場合、応答にはオペレーションの戻り値と任意の inout または out 引数値が含まれます。それ以外のケース、つまり、応答ステータス値が USER_EXCEPTION または SYSTEM_EXCEPTION の場合、応答には例外の符号化が含まれます。

Tobj_ServantBase::deactivate_object() オペレーションで生成された例外

例外が Tobj_ServantBase::deactivate_object() オペレーションで生成された場合、次の処理が発生します。

- 例外はクライアント・アプリケーションに返されません。
- 生成された例外が TobjS::DectivateObjectFailed の場合、メッセージがユーザ・ログに送られます。例外のコンストラクタに reason 文字列が指定されている場合、その reason 文字列もメッセージの一部として書き込まれます。
- TobjS::DeactivateObjectFailed 以外の例外のメッセージがユーザ・ログに送られます。このメッセージは、BEA Tuxedo システムによってキャッチされた例外の型を示します。

オブジェクト・インスタンスの受け渡し時に生成された CORBA マーシャリング例外

ORB は、オブジェクト・インスタンスをオブジェクト・リファレンスとしてマーシャリングできません。たとえば、次のコードでファクトリ・リファレンスを受け渡すと、BEA Tuxedo システムで CORBA マーシャリング例外が発生します。

```
connection::setFactory(this);
```

オブジェクト・インスタンスを受け渡すには、次の例のように、プロキシ・オブジェクト・リファレンスを作成して、そのプロキシを代わりに受け渡します。

```
CORBA::Object myRef = TP::get_object_reference();
ResultSetFactory factoryRef = ResultSetFactoryHelper::_narrow(myRef);
connection::setFactoryRef(factoryRef);
```

コールバック・メソッドのエラー状態の検出

BEA Tuxedo システムには、メッセージ文字列を指定できる定義済みの例外セットが用意されています。TP フレームワークは、アプリケーション・コードが次のコールバック・メソッドでエラーを取得した場合、これらのメッセージ文字列をユーザ・ログに書き込みます。

- `Tobj_ServantBase::activate_object()`
- `Tobj_ServantBase::deactivate_object()`
- `Server::create_servant()`
- `Server::initialize()`
- `Server::release()`

これらの例外は、例外の発生原因に関する明確な情報を送信するための便利なデバッグ支援機能として使用できます。TP フレームワークは、これらのメッセージをユーザ・ログにのみ書き込みます。これらのメッセージは、クライアント・アプリケーションには返されません。

これらのメッセージは、次の例外で指定します。これらの例外では、オプションで reason 文字列を指定できます。

例外	この例外を生成するコールバック・メソッド
ActivateObjectFailed	Tobj_ServantBase::activate_object()
DeactivateObjectFailed	Tobj_ServantBase::deactivate_object()
CreateServantFailed	Server::create_servant()
InitializeFailed	Server::initialize()
ReleaseFailed	Server::release()

メッセージ文字列をユーザ・ログに送るには、次の例のように、その文字列を例外に指定します。

```
throw CreateServantFailed("Unknown interface");
```

これらの例外をスローする場合、reason 文字列パラメータが指定されている必要があります。これらの例外の 1 つで reason 文字列を指定しない場合は、次の例のように、二重引用符を入力する必要があります。

```
throw ActivateObjectFailed("");
```

OMG IDL インターフェイスのバージョンと修正に関する一般的な注意事項

Server オブジェクトの Server::create_servant() オペレーションのインプリメンテーションは、そのインターフェイス ID に基づいてオブジェクトをインスタンス化します。このインターフェイス ID は、ファクトリが TP::create_object_reference() オペレーションを呼び出したときにファクトリに指定されるインターフェイス ID と同じである必要があります。インターフェイス ID が一致しない場合、通常 Server::create_servant() オペレーションで例外が発生するか、または NULL サーバントが返されます。この場合、BEA Tuxedo システムは、CORBA::OBJECT_NOT_EXIST 例外をクライアント・アプリケーションに返します。BEA Tuxedo システムは、TP::create_object_reference() オペレーションでインターフェイス ID の検証を行いません。

このような状態は、開発の過程で、インターフェイスの異なるバージョンが開発されるか、IDL ファイルに対して多くの変更が行われる場合に発生する可能性があります。OMG IDL にインターフェイス ID の文字列定数を指定し、これらの定数をファクトリと Server::create_servant() オペレーションで使用する場合でも、オブジェクト・インプリメンテーションとファクトリが異なる実行可能ファイルに存在する場合は、不一致が発生する可能性があります。多くの場合、この問題の診断は困難です。

こうした問題を避けるには、開発中に次の予防的プログラミング手法を検討する必要があります。このコードは、アプリケーションのデバッグ・バージョンにのみ記述する必要があります。実働バージョンでは受け入れられない性能の低下が発生する可能性があるからです。

- ファクトリが TP::create_object_reference() オペレーションを呼び出す直前に、インターフェイス・リポジトリをチェックして必要なインターフェイスが存在するかどうかを調べるコードを組み込みます。すべてのアプリケーション OMG IDL が最新のものになっており、インターフェイス・リポジトリにロードされていることを確認します。このチェックでインターフェイス ID が見つからない場合、不一致が存在すると見なすことができます。
- ファクトリでの TP::create_object_reference() オペレーションの呼び出しの後に、オブジェクトを ping するコード、つまり、オブジェクトの任意のオペレーション (通常何も行わないオペレーション) を呼び出すコードを組み込みます。この呼び出しが CORBA::OBJECT_NOT_EXIST 例外を生成する場合、インターフェイス ID の不一致が存在します。オブジェクトを ping すると、そのオブジェクトが活性化され、その活性化に関連するオーバーヘッドが発生することに注意してください。

Tobj_ServantBase::deactivate_object() での状態処理に関する注意

Tobj_ServantBase::deactivate_object() オペレーションは、オブジェクトの活性化境界に達したときに呼び出されます。このオペレーションのインプリメンテーションでは、オプションで永続状態をディスクに書き込むことができます。このオペレーションで生成された例外はクライアント・アプリケーションに返されないことを理解しておくことが重要です。クライアント・アプリケーションは、オブジェクトがトランザクションに参加していない限り、このオペレー

ションで生成されたエラー状態を認識しません。このため、このオペレーションで状態が正常に書き込まれたかどうかを知ることが重要である場合は、トランザクションを使用することをお勧めします。

Tobj_ServantBase::deactivate_object() オペレーションで状態を書き込むことを選択し、クライアント・アプリケーションがその書き込みオペレーションの結果を知る必要がある場合、次のことを行うことをお勧めします。

- オブジェクトの状態に影響を及す各オペレーションがトランザクションの中で呼び出され、非活性化がトランザクション境界の中で発生するようにします。そのためには、method 活性化方針か transaction 活性化方針のいずれかを使用します。また、TP::deactivateEnable() オペレーションがトランザクション境界の内部で呼び出される場合は、process 活性化方針を使用することもできます。
- オブジェクト状態の書き込み中にエラーが発生した場合、COSTransactions::Current::rollback_only() オペレーションを呼び出してトランザクションがロールバックされるようにします。これにより、クライアント・アプリケーションは次の例外を確実に受け取ることができます。
  - クライアント・アプリケーションがトランザクションを開始した場合、クライアント・アプリケーションは CORBA::TRANSACTION_ROLLEDBACK 例外を受け取ります。
  - BEA Tuxedo システムがトランザクションを開始した場合、クライアント・アプリケーションは CORBA::OBJ_ADAPTER 例外を受け取ります。

トランザクションを使用しない場合は、Tobj_ServantBase::deactivate_object() オペレーションを使用せずに、オブジェクトの個々のオペレーションのスコープの中でオブジェクト状態を書き込むことをお勧めします。エラーが発生した場合、オペレーションはクライアント・アプリケーションに返される例外を生成できます。

サーバント・プール

「サーバント・プールと状態を持たないオブジェクト」で説明したように、サーバント・プールを使用すると、メソッド・バウンド・オブジェクトとトランザクション・バウンド・オブジェクトのオブジェクト・インスタンス化のコストを削減できます。

サーバント・プールの機能

通常、オブジェクトの非活性化中 (TP フレームワークが Tobj_ServantBase::deactivate_object() オペレーションを呼び出したとき) に、TP フレームワークはオブジェクトのサーバントを削除します。ただし、サーバント・プールを使用する場合、TP フレームワークはオブジェクトの非活性化時にサーバントを削除しません。代わりに、サーバ・アプリケーションはプール内のサーバントへのポインタを保持します。それ以降、そのプール内のサーバントによって処理可能な要求がクライアントから送られてくると、サーバ・アプリケーションはそのサーバントを再利用して新しいオブジェクト ID を割り当てます。サーバントがプールから再利用される場合、TP フレームワークは新しいサーバントを作成しません。

サーバント・プールのインプリメント方法

サーバント・プールは、以下の手順でインプリメントします。

1. Server オブジェクトの Server::initialize() オペレーションに、サーバント・プールをセットアップするコードを記述します。サーバント・プールは 1 つまたは複数のサーバントへのポインタのセットで構成され、サーバント・プールのコードでは特定のクラスのサーバントをプールでどのくらい保持するのかを指定します。
2. プールされたサーバントの Tobj_ServantBase::deactive_object() オペレーションに、TP::application_responsibility() オペレーションをインプリメントします。TP::application_responsibility() オペレーションのインプリメンテーションには、TP フレームワークが Tobj_ServantBase::deactivate_object() オペレーションを呼び出したときにサーバント・プールにサーバントへのポインタを格納するコードを記述します。
3. Server オブジェクトの Server::create_servant() オペレーションのインプリメンテーションに、クライアント要求が到着したときに次のことを行うコードを記述します。
   a. プールをチェックして、その要求を満たすことができるサーバントが存在するかどうかを調べる。

b. サーバントが存在しない場合は、サーバントを作成してその `Tobj_ServantBase::activate_object()` オペレーションを呼び出す。

c. サーバントが存在する場合は、その `Tobj_ServantBase::activate_object()` オペレーションを呼び出して、クライアント要求に含まれているオブジェクト ID を割り当てる。

注記 このリリースでは、`TP::application_responsibility()` オペレーションのサポートが変更されています。詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

デレゲーション・ベースのインターフェイス・インプリメンテーション

BEA Tuxedo CORBA アプリケーションでオブジェクトをインプリメントする主要な方法は、継承とデレゲーションの 2 つです。オブジェクトが POA スケルトン・クラスから継承され、それによって CORBA オブジェクトとなった場合、そのオブジェクトは継承によってインプリメントされたと言われます。

ただし、POA スケルトン・クラスからの継承が困難または不可能な C++ オブジェクトを CORBA アプリケーションで使用したい場合もあります。たとえば、POA スケルトン・クラスから継承するために大幅な書き換えが必要な C++ オブジェクトなどです。こうした非 CORBA オブジェクトを CORBA アプリケーションで使用するには、そのオブジェクト用の tie クラスを作成します。tie クラスは、POA スケルトン・クラスから継承されます。また、tie クラスには 1 つまたは複数のオペレーションが含まれ、それらはインプリメンテーションのためにレガシー・クラスに委譲されます。これにより、レガシー・クラスはデレゲーションによって CORBA アプリケーションにインプリメントされます。

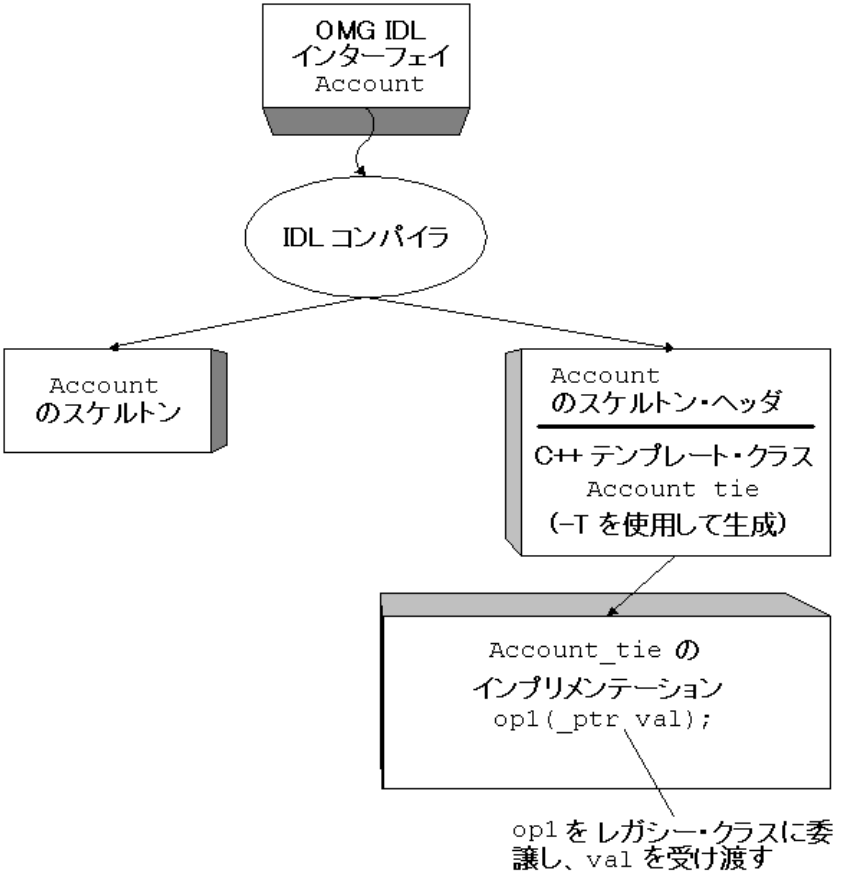
BEA Tuxedo システムの tie クラスについて

デレゲーション・ベースのインターフェイス・インプリメンテーションを作成するには、IDL コンパイラの -T コマンド行オプションを使用して、OMG IDL ファイルに定義されている各インターフェイスに対する tie クラス・テンプレートを生成します。

CORBA アプリケーションで tie クラスを使用する場合、Server オブジェクトに Server::create_servant() オペレーションをインプリメントする方法も変わります。以降の節では、BEA Tuxedo 製品で tie クラスを使用する方法についてさらに詳しく説明し、Server::create_servant() オペレーションをインプリメントしてこれらのクラスをインスタンス化する方法についても説明します。

BEA Tuxedo CORBA では、tie クラスはサーバントであり、したがって基本的にレガシー・クラスのラッパー・オブジェクトとして機能します。

次の図に、レガシー・オブジェクトのラッパーとして機能する、Account インターフェイスの継承の特性を示します。レガシー・オブジェクトには、オペレーション op1 のインプリメンテーションが含まれています。tie クラスは、op1 をレガシー・クラスに委譲します。



tie クラスは、クライアント・アプリケーションからは見えません。クライアント・アプリケーションには、tie クラスは自身が呼び出すオブジェクトの完全なインプリメンテーションのように見えます。tie クラスは、ユーザが提供するレガシー・クラスにすべてのオペレーションを委譲します。さらに、tie クラスには次のものが含まれます。

- コンストラクタとデストラクタのコード。このコードは、tie クラスとレガシー・クラスの起動およびシャットダウン手順を処理します。
- ハウスキーピング・コード。このコードは、アクセサなどのオペレーションをインプリメントします。

tie クラスを使用する条件

tie クラスは BEA Tuxedo CORBA に固有のものではなく、CORBA アプリケーションでデレゲーションをインプリメントするための唯一の方法でもありません。ただし、BEA Tuxedo CORBA の tie クラス用の便利な機能を使用すると、それらの tie クラスの基本的なコンストラクタ、デストラクタ、およびハウスキーピング・オペレーションに必要なコーディングの量を大幅に減らすことができます。

tie クラスは、次のいずれかの状況で使用することをお勧めします。

- POA スケルトン・クラスからの継承が困難または不可能なオブジェクトを CORBA アプリケーションにインプリメントする場合
- レガシー・クラスのインスタンスのすべての呼び出しを 1 つのサーバントから実現できる場合
- CORBA アプリケーションでレガシー・クラスを使用しており、そのインスタンスの存続期間をサーバント・クラスに結びつける場合
- 特定のサーバントの唯一の目的がデレゲーションであり、このため、そのサーバントのほぼすべてのコードがレガシー・オブジェクトの起動、シャットダウン、およびデレゲーションに使用される場合

次の場合、tie クラスの使用はお勧めしません。

- オブジェクト・インスタンスのオペレーションが複数のレガシー・オブジェクトのインスタンスに委譲される場合
- デレゲーションがオブジェクトの目的の一部に過ぎない場合

CORBA アプリケーションに tie クラスを作成する方法

BEA Tuxedo ドメインのアプリケーションに tie クラスを作成するには、次の手順に従います。

1. アプリケーションのオブジェクトと同じように、OMG IDL ファイルに tie クラスのインターフェイス定義を作成します。

2. -T オプションを使用して、OMG IDL ファイルをコンパイルします。

   IDL コンパイラは、C++ テンプレート・クラスを生成します。このクラスはスケルトンの名前を取り、その後ろに _tie という文字列が付加されます。IDL コンパイラは、このテンプレート・クラスをスケルトン・ヘッダ・ファイルに追加します。

   IDL コンパイラは、tie クラスのインプリメンテーション・ファイルを生成しないことに注意してください。このファイルは、次のステップで説明するように手動で作成する必要があります。

3. tie クラスのインプリメンテーション・ファイルを作成します。インプリメンテーション・ファイルには、そのオペレーションをレガシー・クラスに委譲するコードを記述します。

4. Server オブジェクトの Server::create_servant() オペレーションに、レガシー・オブジェクトをインスタンス化するコードを記述します。

   次の例では、tie クラス POA_Account_tie のサーバントが作成され、レガシー・クラス LegacyAccount がインスタンス化されます。

```
Account * Account_ptr = new LegacyAccount();
AccountFactoryServant = new POA_Account_tie<LegacyAccount> (Account_ptr)
```

注記 UNIX 用の Compaq C++ Tru64 コンパイラで tie クラスをコンパイルする場合、buildobjserver コマンドで使用される CFLAGS または CPPFLAGS 環境変数の定義に -noimplicit_include オプションを指定する必要があります。このオプションを指定すると、C++ コンパイラはサーバ・スケルトン・ファイル (_s.h) がインクルードされる場所にサーバ・スケルトン定義ファイル (_s.cpp) を自動的に組み込みません。これは、複数定義のシンボル・エラーを回避するために必要です。Tru64 C++ で tie クラスなどのクラス・テンプレートを使用する方法については、Compaq の出版物を参照してください。

3 Basic CORBA サーバ・アプリケーションの設計とインプリメント

この章では、Basic University サンプル・アプリケーションを例として使用して、CORBA サーバ・アプリケーションを設計およびインプリメントする方法について説明します。この章の内容は、インプリメントするアプリケーションの設計が完了し、OMG IDL に定義されていることを前提にしています。この章では、サーバ・アプリケーション向けの設計およびインプリメンテーションの選択肢に焦点を当てます。

ここでは、以下の内容について説明します。

- Basic University サンプル・アプリケーションのしくみ。このトピックでは、設計とインプリメンテーションの考慮事項について説明します。
- University サーバ・アプリケーションの設計に関する考慮事項。このトピックでは、次のトピックについて包括的に説明します。
  - オブジェクト・リファレンスの生成に関する設計上の考慮事項
  - オブジェクト状態の管理に関する設計上の考慮事項
  - 永続状態情報の処理に関する設計上の考慮事項
  - Basic サンプル・アプリケーションがデザイン・パターンを適用する方法
  - BEA Tuxedo システムに組み込まれた性能効率化
  - 状態を持つオブジェクトの事前活性化

Basic University サンプル・アプリケーションのしくみ

Basic University サンプル・アプリケーションを使用すると、学生は中央の University データベースからコース情報を参照できます。この Basic サンプル・アプリケーションでは、学生は次のことを行うことができます。

- 検索文字列を指定することで、データベースからコース概要を参照します。サーバ・アプリケーションは、検索文字列に含まれているタイトル、教授、または説明に一致するすべてのコースの概要を返します。クライアント・アプリケーションに返されるコース概要には、コースの番号とタイトルしか含まれません。
- 特定のコースに関する詳細情報を参照します。指定したコースの詳細情報には、コース概要のほかに次の情報が含まれます。
  - 受講料
  - 履修単位
  - クラス・スケジュール
  - 座席数
  - 登録学生数
  - 教授
  - 説明

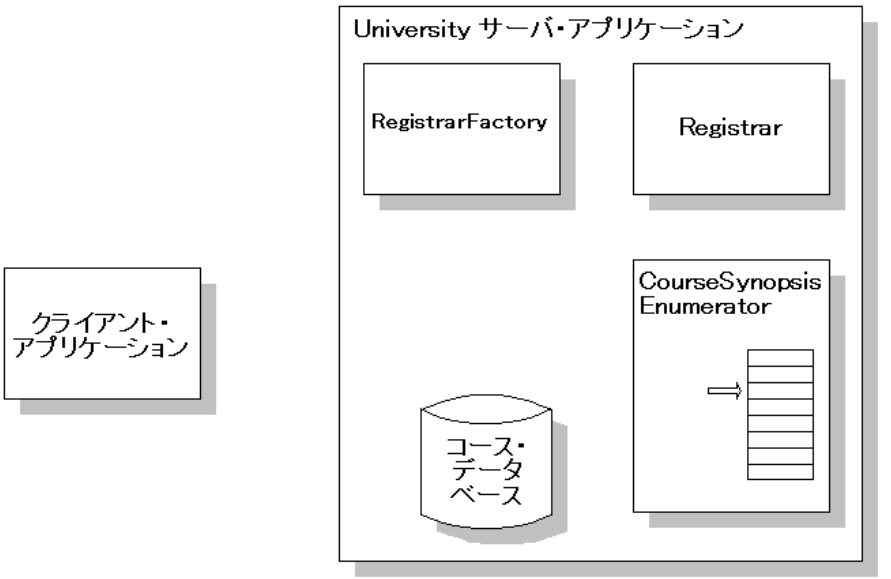
Basic University サンプル・アプリケーションの OMG IDL

Basic University サンプル・アプリケーションの OMG IDL ファイルには、次のインターフェイスが定義されます。

インターフェイス	説明	オペレーション
RegistrarFactory	Registrar オブジェクトのオブジェクト・リファレンスを作成します。	find_registrar()
Registrar	データベースからコース情報を取得します。	get_courses_synopsis() get_courses_details()
CourseSynopsisEnumerator	検索基準に一致するコースの概要をデータベースからフェッチして、それらをメモリに読み込みます。	get_next_n() destroy()

図 3-1 に、Basic University サンプル・アプリケーションを示します。

図 3-1　Basic University サンプル・アプリケーション



Basic University サンプル・アプリケーションで何が発生するかを説明するために、次のイベント・グループについて説明します。

- アプリケーションの起動 ― サーバ・アプリケーションが起動し、クライアント・アプリケーションが Registrar オブジェクトのオブジェクト・リファレンスを取得するときです。
- コース概要の参照 ― クライアント・アプリケーションが要求を送信してコース概要を参照するときです。
- コース詳細の参照 ― クライアント・アプリケーションが要求を送信して特定のコース・リストの詳細を参照するときです。

アプリケーションの起動

次に、Basic クライアント・アプリケーションおよびサーバ・アプリケーションが起動し、クライアント・アプリケーションが Registrar オブジェクトのオブジェクト・リファレンスを取得するときに行われる一般的な一連のイベントを示します。

1. Basic クライアント・アプリケーションおよびサーバ・アプリケーションが起動し、クライアント・アプリケーションは FactoryFinder から RegistrarFactory オブジェクトのオブジェクト・リファレンスを取得します。
2. RegistrarFactory オブジェクトのリファレンスを使用して、クライアント・アプリケーションは RegistrarFactory オブジェクトの find_registrar() オペレーションを呼び出します。
3. RegistrarFactory はメモリ内に存在しません (このオブジェクトに対する要求が以前にサーバ・プロセスに到着していないため)。このため、TP フレームワークは Server オブジェクトの Server::create_servant() オペレーションを呼び出して RegistrarFactory オブジェクトをインスタンス化します。
4. インスタンス化が済んだら、RegistrarFactory オブジェクトの find_registrar() オペレーションが呼び出されます。RegistrarFactory オブジェクトは、Registrar のオブジェクト・リファレンスを作成してクライアント・アプリケーションに返します。

コース概要の参照

次に、学生がコース概要のリストを参照するときに発生する一連のイベントを順番に示します。

1. Registrar オブジェクトのオブジェクト・リファレンスを使用して、クライアント・アプリケーションは get_courses_synopsis() オペレーションを呼び出し、次のものを指定します。
   - データベースからコース概要を検索するための検索文字列
   - 変数 number_to_get で表され、返される概要リストのサイズを指定する整数
2. Registrar はメモリ内に存在しません (このオブジェクトに対する要求が以前にサーバ・プロセスに到着していないため)。このため、TP フレームワークは Server オブジェクトにインプリメントされている Server::create_servant() オペレーションを呼び出します。これにより、Registrar オブジェクトがサーバ・マシンのメモリ内にインスタンス化されます。
3. Registrar オブジェクトはクライアント要求を受信し、CourseSynopsisEnumerator オブジェクトのオブジェクト・リファレンスを作成します。CourseSynopsisEnumerator オブジェクトは、データベースからコース概要をフェッチするために Registrar オブジェクトによって呼び出されます。

   CourseSynopsisEnumerator オブジェクトのオブジェクト・リファレンスを作成するために、Registrar オブジェクトは次のことを行います。

   a. CourseSynopsisEnumerator オブジェクトの一意な ID を生成します。
   b. CourseSynopsisEnumerator オブジェクトのオブジェクト ID を生成します。この ID は、前のステップで生成された一意の ID とクライアントによって指定された検索文字列を連結したものです。
   c. CourseSynopsisEnumerator オブジェクトのインターフェイス・リポジトリ ID をそのインターフェイス・タイプ・コードから取得します。
   d. TP::create_object_reference() オペレーションを呼び出します。このオペレーションにより、最初のクライアント要求で必要な CourseSynopsisEnumerator オブジェクトのオブジェクト・リファレンスが作成されます。

4. 前のステップで作成したオブジェクト・リファレンスを使用して、Registrar オブジェクトは CourseSynopsisEnumerator オブジェクトの get_next_n() オペレーションを呼び出して、リスト・サイズを受け渡します。ステップ 1 で説明したとおり、このリスト・サイズはパラメータ number_to_get によって表されます。
5. TP フレームワークは、Server オブジェクトの Server::create_servant() オペレーションを呼び出して、CourseSynopsisEnumerator オブジェクトをインスタンス化します。
6. TP フレームワークは、CourseSynopsisEnumerator オブジェクトの activate_object() オペレーションを呼び出します。このオペレーションは、次の 2 つのことを行います。
   - その OID から検索基準を抽出します。
   - その検索基準を使用して、一致するコース概要をデータベースからフェッチし、それらをサーバ・マシンのメモリに読み込みます。
7. CourseSynopsisEnumerator オブジェクトは、Registrar オブジェクトに次の情報を返します。
   - 戻り値の CourseSynopsisList に指定されるコース概要リスト。これは、最初のコース概要リストを含む sequence です。
   - 検索基準に一致するが、まだ返されていないコース概要の数。これは、number_remaining パラメータによって指定されます。
8. Registrar オブジェクトは、クライアント・アプリケーションに CourseSynopsisEnumerator オブジェクト・リファレンスを返します。また、そのオブジェクトから取得した次の情報も返します。
   - 最初のコース概要リスト
   - number_remaining 変数

   number_remaining 変数が 0 の場合、Registrar オブジェクトは CourseSynopsisEnumerator オブジェクトの destroy() オペレーションを呼び出し、クライアント・アプリケーションに nil リファレンスを返します。
9. クライアント・アプリケーションは、一致する次の概要リストを取得するための要求を直接 CourseSynopsisEnumerator オブジェクトに送信します。
10. CourseSynopsisEnumerator オブジェクトはそのクライアント要求を満たすと共に、更新した number_remaining 変数を返します。

11. クライアント・アプリケーションは、CourseSynopsisEnumerator オブジェクトとのやり取りを終了すると、CourseSynopsisEnumerator オブジェクトの destroy() オペレーションを呼び出します。この結果、CourseSynopsisEnumerator オブジェクトは TP::deactivateEnable() オペレーションを呼び出します。

12. TP フレームワークは、CourseSynopsisEnumerator オブジェクトの deactivate_object() オペレーションを呼び出します。この結果、CourseSynopsisEnumerator オブジェクトが保持しているコース概要のリストがサーバ・マシンのメモリから削除されます。これにより、CourseSynopsisEnumerator オブジェクトのサーバントを別のクライアント要求のために再利用できるようになります。

コース詳細の参照

次に、クライアント・アプリケーションがコース詳細を参照するときに行われる一連のイベントを示します。

1. 学生は、詳細を参照したいコースのコース番号を入力します。
2. クライアント・アプリケーションは、Registrar オブジェクトの get_course_details() オペレーションを呼び出し、コース番号のリストを受け渡します。
3. Registrar オブジェクトは、一致するコース番号をデータベースから検索し、指定された各コースの詳細を含んだリストを返します。このリストは、CourseDetailsList 変数に格納されます。これは、コースの詳細を含んだ一連の struct です。

University サーバ・アプリケーションの設計に関する考慮事項

Basic University サンプル・アプリケーションには、University サーバ・アプリケーションが含まれます。このサーバ・アプリケーションは、CORBA サーバ・アプリケーションの基本的な設計問題を処理します。この節では、以下のトピックについて説明します。

- オブジェクト・リファレンスの生成に関する設計上の考慮事項
- オブジェクト状態の管理に関する設計上の考慮事項
- 永続状態情報の処理に関する設計上の考慮事項
- Basic サンプル・アプリケーションがデザイン・パターンを適用する方法

この節では、さらに次の 2 つのトピックについても説明します。

- BEA Tuxedo システムに組み込まれた性能効率化
- 状態を持つオブジェクトの事前活性化

オブジェクト・リファレンスの生成に関する設計上の考慮事項

Basic クライアント・アプリケーションでは、University サーバ・アプリケーションによって管理される次のオブジェクトのリファレンスが必要です。

- `RegistrarFactory` オブジェクト
- `Registrar` オブジェクト
- `CourseSynopsisEnumerator` オブジェクト

次の表に、これらのリファレンスがどのように生成されて返されるかを示します。

オブジェクト	オブジェクト・リファレンスが生成されて返されるしくみ
RegistrarFactory	RegistrarFactory オブジェクトのオブジェクト・リファレンスは Server オブジェクトで生成されます。Server オブジェクトは、RegistrarFactory オブジェクトを FactoryFinder に登録します。クライアント・アプリケーションは、RegistrarFactory オブジェクトのリファレンスを FactoryFinder から取得します。 Basic University サーバ・アプリケーション・プロセスには、RegistrarFactory オブジェクトは 1 つしか存在しません。
Registrar	Registrar オブジェクトのオブジェクト・リファレンスは RegistrarFactory オブジェクトによって生成され、クライアント・アプリケーションが find_registrar() オペレーションを呼び出したときに返されます。Registrar オブジェクトに対して作成されるオブジェクト・リファレンスは常に同じです。このオブジェクト・リファレンスには一意な OID は含まれません。 Basic University サーバ・アプリケーション・プロセスには、Registrar オブジェクトは 1 つしか存在しません。

オブジェクト	オブジェクト・リファレンスが生成されて返されるしくみ
CourseSynopsisEnumerator	CourseSynopsisEnumerator オブジェクトのオブジェクト・リファレンスは、クライアント・アプリケーションが get_courses_synopsis() オペレーションを呼び出したときに Registrar オブジェクトによって作成されます。つまり、Registrar オブジェクトは CourseSynopsisEnumerator オブジェクトのファクトリです。CourseSynopsisEnumerator オブジェクトの設計と使い方については、この章で後述します。 Basic University サーバ・アプリケーション・プロセスには、任意の数の CourseSynopsisEnumerator オブジェクトが存在できます。

University サーバ・アプリケーションがオブジェクト・リファレンスを生成する方法について、以下のことに注意してください。

- Server オブジェクトは、RegistrarFactory オブジェクトを FactoryFinder に登録します。この方法により、クライアント・アプリケーションはサーバ・アプリケーション内の基本オブジェクトのオブジェクト・リファレンスを取得するために必要なファクトリを検索できます。
- Registrar オブジェクトのオブジェクト・リファレンスは、RegistrarFactory オブジェクトによって作成されます。これは、クライアント・アプリケーションにオブジェクト・リファレンスを返すためのきわめて一般的かつ基本的な方法を示します。つまり、クライアント・アプリケーションがビジネス・ロジックを実行するために必要な基本オブジェクトのリファレンスを作成して返すための専用ファクトリが存在するということです。
- CourseSynopsisEnumerator オブジェクトのオブジェクト・リファレンスは、登録されたファクトリの外部で作成されます。University サンプル・アプリケーションの場合、これは優れた設計です。これは、CourseSynopsisEnumerator オブジェクトの使い方のためです。つまり、その存在が特定のクライアント・アプリケーション・オペレーションに固有で

あるからです。CourseSynopsisEnumerator オブジェクトは、特定のリストと、ほかの問い合わせの結果とは関連性のない結果を返します。

- Registrar はそのオペレーションの 1 つで別のオブジェクトのオブジェクト・リファレンスを作成するので、Registrar オブジェクトはファクトリです。ただし、Registrar オブジェクトはファクトリとして FactoryFinder に登録されません。このため、クライアント・アプリケーションは Registrar オブジェクトのリファレンスを FactoryFinder から取得することはできません。

オブジェクト状態の管理に関する設計上の考慮事項

Basic サンプル・アプリケーションの 3 つのオブジェクトには、それぞれ独自の状態管理の要件が存在します。この節では、それぞれのオブジェクト状態管理の要件について説明します。

RegistrarFactory オブジェクト

RegistrarFactory オブジェクトは、特定のクライアント要求に一意なものである必要はありません。また、このオブジェクトをメモリに保持しておき、クライアント呼び出しごとにこのオブジェクトを活性化および非活性化する負担を回避することは合理的です。したがって、RegistrarFactory オブジェクトには process 活性化方針が割り当てられます。

Registrar オブジェクト

Basic サンプル・アプリケーションは、小規模な環境へのデプロイに適しています。Registrar オブジェクトは、RegistrarFactory オブジェクトに似た特性を備えています。つまり、このオブジェクトは特定のクライアント要求に一意なものである必要がありません。また、呼び出しごとにこのオブジェクトを活性化および非活性化する負担を回避することは合理的です。したがって、Basic サンプル・アプリケーションでは、Registrar オブジェクトには process 活性化方針が割り当てられます。

CourseSynopsisEnumerator オブジェクト

University サーバ・アプリケーションの基本的な設計上の問題は、大きすぎて 1 回の応答ではクライアント・アプリケーションに返すことができないコース概要リストをどのように処理するかです。したがって、この問題の解決策は、次のことが中心となります。

- クライアント・アプリケーションとコース概要を University データベースからフェッチできるオブジェクト間の会話を開始します。
- オブジェクトがクライアント・アプリケーションに最初の概要リストを返します。
- コース概要の残りをメモリに保持しておき、クライアント・アプリケーションが一度にそれらを取得できるようにします。
- 完了時にクライアント・アプリケーションが会話を終了することによって、マシンのリソースを解放します。

University サーバ・アプリケーションは、この解決策をインプリメントする `CourseSynopsisEnumerator` オブジェクトを備えています。このオブジェクトは最初に呼び出されたときに最初の概要リストを返しますが、その後もメモリ内コンテキストに保持されるので、クライアント・アプリケーションは後続の要求で概要の残りを取得できます。メモリ内コンテキストを保持するには、CourseSynopsisEnumerator オブジェクトは状態を持たなければなりません。つまり、このオブジェクトは複数のクライアント呼び出しにわたってメモリ内に常駐します。

クライアントが `CourseSynopsisEnumerator` オブジェクトとの会話を終了するときに、このオブジェクトにはメモリからフラッシュされるための手段が必要です。このため、`CourseSynopsisEnumerator` オブジェクトの適切な状態管理の方法は、`process` 活性化方針を割り当て、CORBA アプリケーション制御の非活性化機能をインプリメントすることです。

アプリケーション制御の非活性化は、そのオブジェクトの `destroy()` オペレーションによってインプリメントされます。

次のコード例に、`CourseSynopsisEnumerator` オブジェクトの `destroy()` オペレーションを示します。

```
void CourseSynopsisEnumerator_i::destroy()
{
        // クライアントが列挙子の「破棄」を呼び出すと、
        // このオブジェクトは「消滅」する必要がある。
        // そのためには、TP フレームワークに
```

```
        // このオブジェクトとの会話が完了したことを通知する
        TP::deactivateEnable();
    }
```

Basic University サンプル・アプリケーションの ICF ファイル

次のコード例に、Basic サンプル・アプリケーション用の ICF ファイルを示します。

```
module POA_UniversityB
{
    implementation CourseSynopsisEnumerator_i
    {
        activation_policy  ( process                                  );
        transaction_policy ( optional                                 );
        implements         ( UniversityB::CourseSynopsisEnumerator );
    };
    implementation Registrar_i
    {
        activation_policy  ( process                                  );
        transaction_policy ( optional                                 );
        implements         ( UniversityB::Registrar                   );
    };
    implementation RegistrarFactory_i
    {
        activation_policy  ( process                                  );
        transaction_policy ( optional                                 );
        implements         ( UniversityB::RegistrarFactory            );
    };
};
```

永続状態情報の処理に関する設計上の考慮事項

永続状態情報の処理とは、オブジェクトの活性化中または活性化後のある時点でディスクから永続状態情報を読み取り、必要な場合、非活性化中または非活性化後のある時点でそれを書き込むことです。Basic サンプル・アプリケーションでは、次の 2 つのオブジェクトが永続状態情報を処理します。

- `Registrar` オブジェクト
- `CourseSynopsisEnumerator` オブジェクト

以降の 2 つの節では、この 2 つのオブジェクトが永続状態情報を処理する方法に関する設計上の考慮事項を説明します。

Registrar オブジェクト

Registrar オブジェクトのオペレーションの 1 つによって、クライアント・アプリケーションにコース情報の詳細が返されます。一般的なシナリオでは、数多くのコース概要を参照した学生は、通常 2 つか 3 つのコースに関する詳細情報を一度に参照しようとします。

このシナリオを効率的にインプリメントするために、Registrar オブジェクトには get_course_details() オペレーションが定義されています。このオペレーションは、コース番号のリストを指定するパラメータを入力として受け付けます。次にこのオペレーションは、データベースからコースの詳細を検索し、その詳細をクライアント・アプリケーションに返します。このオペレーションがインプリメントされるオブジェクトはプロセス・バウンドなので、このオペレーションはその呼び出しの完了後に状態データをメモリに保持するのを避ける必要があります。

Registrar オブジェクトは、永続状態をメモリに保持しません。クライアント・アプリケーションが get_course_details() オペレーションを呼び出すと、このオブジェクトは関連するコース情報を University データベースからフェッチしてクライアントに送信します。このオブジェクトは、コース・データをメモリ内に保持しません。このオブジェクトの activate_object() オペレーションと deactivate_object() オペレーションでは永続状態は処理されません。

CourseSynopsisEnumerator オブジェクト

CourseSynopsisEnumerator オブジェクトは、自身が University データベースから検索するコース概要を処理します。状態処理に関する設計上の考慮事項は、ディスクから状態を読み取る方法です。このオブジェクトは、状態をディスクに書き込みません。

CourseSynopsisEnumerator オブジェクトの機能には、次の 3 つの重要な側面があります。これらの側面は、このオブジェクトがその永続状態を読み取る方法に影響を与えます。

- このオブジェクトの OID には、コース概要を取得するための最初のクライアント要求で指定される検索基準が含まれます。この検索基準は、データベー

スのキーとして機能します。つまり、このオブジェクトは OID に格納されている検索基準に基づいてデータベースから情報を抽出します。

- このオブジェクトのすべてのオペレーションは、そのオブジェクトがメモリに読み込むコース概要を使用します。
- このオブジェクトは、非活性化されるときにコース概要をメモリからフラッシュする必要があります。

この 3 つの側面を考えたとき、このオブジェクトでは次のことを行うのが合理的です。

- 活性化時に、そのオブジェクトの `activate_object()` オペレーションを介してその永続状態情報を読み取ります。
- 非活性化時に、`deactivate_object()` オペレーションを介してメモリからコース概要をフラッシュします。

したがって、`CourseSynopsisEnumerator` オブジェクトが活性化されるときに、そのオブジェクトの `activate_object()` オペレーションは次のことを行います。

1. その OID から検索基準を抽出します。
2. 検索基準に一致するコース概要をデータベースから検索します。

注記 オブジェクトの `Tobj_ServantBase::activate_object()` オペレーションまたは `Tobj_ServantBase::deactivate_object()` オペレーションをインプリメントする場合は、インプリメンテーション・ヘッダ・ファイル (つまり、*application*`_i.h` ファイル) を編集し、これらのオペレーションの定義をそのオブジェクトのインターフェイスのクラス定義テンプレートに追加してください。

University データベースの使い方

University サンプル・アプリケーションが University データベースを使用する方法について、次のことに注意してください。

- すべての University サンプル・アプリケーションは、University データベースにアクセスしてコース情報と学生情報を操作します。通常、これはインプリメンテーション・ファイルに記述するコードの大きい部分を占めます。University サンプルのインプリメンテーション・ファイルをシンプルにし、データベース・コードの代わりに CORBA 機能に集中できるようにするため

に、このサンプルではデータベースに対する読み書きを行うすべてのコードが 1 つのクラス・セットにラップされています。utils ディレクトリ内の samplesdb.h ファイルには、これらのクラスの定義が記述されています。これらのクラスは、University データベース内のコース・レコードと学生レコードを読み書きするために必要なすべての SQL 呼び出しを行います。

注記 Wrapper および Production サンプル・アプリケーションの BEA Tuxedo Teller アプリケーションは、University データベース内の口座情報に直接アクセスし、samplesdb.h ファイルを使用しません。

Basic サーバ・アプリケーションに組み込むファイルの詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。

- CourseSynopsisEnumerator オブジェクトは、データベース・カーソルを使用して University データベースから一致するコース概要を検索します。データベース・カーソルは複数のトランザクションにまたがることができないので、CourseSynopsisEnumerator オブジェクトの activate_object() オペレーションは一致するすべてのコース概要をメモリに読み取ります。カーソルはイテレータ・クラスによって管理されるため、CourseSynopsisEnumerator オブジェクトからは見えません。University サンプル・アプリケーションがトランザクションを使用する方法については、「トランザクションの CORBA サーバ・アプリケーションへの統合」を参照してください。

Basic サンプル・アプリケーションがデザイン・パターンを適用する方法

Basic サンプル・アプリケーションは、次のデザイン・パターンを使用します。

- Process-Entity
- List-Enumerator

この節では、この 2 つのデザイン・パターンが Basic サンプル・アプリケーションに適している理由と、Basic サンプル・アプリケーションがこれらのパターンをインプリメントする方法について説明します。

Process-Entity デザイン・パターン

第 1 章の 26 ページ「Process-Entity デザイン・パターン」で説明したように、このデザイン・パターンは、クライアント・アプリケーションによって必要とされるデータ・エンティティを処理する 1 つのプロセス・オブジェクトが存在する場合に適しています。データ・エンティティは、クライアント・アプリケーションではなくこのプロセス・オブジェクトによって操作される CORBA `struct` としてカプセル化されます。

Process-Entity デザイン・パターンを Basic サンプル・アプリケーションに適用すると、Basic サンプル・アプリケーションは細粒度オブジェクトのインプリメントを回避できます。たとえば、`Registrar` オブジェクトは、同じような多数のコース・オブジェクト・セットの有効な代替となります。単一の粗粒度 `Registrar` オブジェクトを管理する負荷は、何百、何千もの細粒度のコース・オブジェクトを管理する負荷に比べれば、それほど大きくはありません。

Process-Entity デザイン・パターンの詳細については、デザイン・パターンに関する技術情報を参照してください。

List-Enumerator デザイン・パターン

このデザイン・パターンは、大きすぎて 1 つの応答でクライアント・アプリケーションに返すことができないデータの内部リストをオブジェクトが生成した場合に適しています。このため、オブジェクトは 1 回の応答で最初のデータ・リストをクライアント・アプリケーションに返し、後続の応答で残りのデータを返すことができなければなりません。

また、List-Enumerator オブジェクトは、既に返されたデータの量を同時に追跡して、後続のリストを正確に返すことができるようにする必要があります。List-Enumerator オブジェクトは常に状態を持ち (クライアント呼び出しにまたがって活性化され、メモリに常駐する)、サーバ・アプリケーションはそれらが不要になったときにそれらを非活性化できます。

List-Enumerator デザイン・パターンは、`CourseSynopsisEnumerator` オブジェクトに最適です。このデザイン・パターンをインプリメントすると、次のメリットが得られます。

- University サーバ・アプリケーションは、大きいコース概要リストを、クライアント・アプリケーションが処理できる方法、つまり管理可能な分量で返すための方法を手に入れます。

- 各 CourseSynopsisEnumerator オブジェクトは一意で、その内容はこのオブジェクトが作成される原因となった要求によって決まります。また、各 CourseSynopsisEnumerator のオブジェクト ID も一意です。クライアントが Registrar オブジェクトの get_courses_synopsis() オペレーションを呼び出すと、Registrar オブジェクトは次のものを返します。
  - 最初の概要リスト
  - 残りの概要を返すことができる CourseSynopsisEnumerator オブジェクトのオブジェクト・リファレンス

  このため、後続のすべての呼び出しは適切な CourseSynopsisEnumerator オブジェクトに対して行われます。これは、サーバ・プロセスに CourseSynopsisEnumerator クラスのアクティブ・インスタンスが複数存在する場合に重要です。

  get_courses_synopsis() オペレーションは一意の CourseSynopsisEnumerator オブジェクト・リファレンスを返すので、クライアント要求どうしが衝突することはありません。つまり、クライアント要求が間違った CourseSynopsisEnumerator オブジェクトに誤って送られることはありません。

Registrar オブジェクトは get_courses_synopsis() オペレーションを備えていますが、データベース問い合わせと概要リストの知識はすべて CourseSynopsisEnumerator オブジェクトに埋め込まれます。この場合、Registrar オブジェクトは単にクライアントが次のデータを取得するための手段として機能します。

- 最初の概要リスト
- 残りの概要を返すことができる CourseSynopsisEnumerator オブジェクトのリファレンス

BEA Tuxedo システムに組み込まれた性能効率化

BEA Tuxedo システムは、同じサーバ・プロセス内の 2 つのオブジェクト間のデータ・マーシャリングを自動的に無効化することによって、性能の効率化を実現します。この効率化は、次の環境が存在するときに得られます。

- オブジェクト・リファレンスが、自身が作成されたサーバ・プロセスを含んでいるグループに送られる場合。

- そのサーバ・プロセス内のあるオブジェクトがそのオブジェクト・リファレンスを使用してオペレーションを呼び出し、それによって同じプロセスでオブジェクトがインスタンス化される場合

この例には、Registrar オブジェクトが CourseSynopsisEnumerator オブジェクトのオブジェクト・リファレンスを作成し、それによってそのオブジェクトがインスタンス化された場合などがあります。この 2 つのオブジェクト間の要求と応答では、データ・マーシャリングは行われません。

状態を持つオブジェクトの事前活性化

状態を持つオブジェクトの事前活性化機能を使用すると、クライアント・アプリケーションがオブジェクトを呼び出す前にそのオブジェクトを活性化できます。この機能は、University サンプルの CourseSynopsisEnumerator オブジェクトなどのイテレータ・オブジェクトを作成するときに特に役立ちます。

状態を持つオブジェクトの事前活性化では、TP::create_active_object_reference() オペレーションを使用します。通常、クライアントがあるオブジェクトに対する呼び出しを発行するまで、CORBA サーバ・アプリケーションでそのオブジェクトは作成されません。ただし、オブジェクトを事前活性化し、TP::create_active_object_reference() オペレーションを使用してそのオブジェクトのリファレンスをクライアントに受け渡すことによって、クライアント・アプリケーションは既に活性化され、状態を持っているオブジェクトを呼び出すことができます。

注記 状態を持つオブジェクトの事前活性化の機能は、WebLogic Enterprise バージョン 4.2 で最初に導入されました。

状態を持つオブジェクトを事前活性化する方法

事前活性化の機能を使用するためのプロセスは、サーバ・アプリケーションに次のコードを記述することです。

1. オブジェクトを作成するための C++ new 文の呼び出しを含めます。
2. オブジェクトの状態を設定します。
3. TP::create_active_object_reference() オペレーションを呼び出して、新しく作成されたオブジェクトのリファレンスを取得します。このオブジェクト・リファレンスは、クライアント・アプリケーションに返すことができます。

このように、事前活性化オブジェクトは、TP フレームワークがそのオブジェクトの Server::create_servant() オペレーションと Tobj_ServantBase::activate_object() オペレーションを呼び出さずに作成されます。

事前活性化オブジェクトの使い方に関する注意

事前活性化機能を使用するときには、以下のことに注意してください。

- 事前活性化オブジェクトには、process 活性化方針が割り当てられている必要があります。このため、これらのオブジェクトは、プロセスの終了時か、それらのオブジェクトの TP::deactivateEnable() オペレーションの呼び出しによってのみ非活性化できます。
- TP::create_active_object_reference() オペレーションによって作成されるオブジェクト・リファレンスは一時的なものです。これは、事前活性化オブジェクトは自身が作成されたプロセスの存続期間にわたって存在し、別のサーバ・プロセスで再び活性化してはならないからです。

  ある一時オブジェクト・リファレンスが作成されたプロセスのシャットダウン後にクライアント・アプリケーションがそのオブジェクト・リファレンスを呼び出した場合、TP フレームワークは次の例外を返します。

  ```
  CORBA::OBJECT_NOT_EXIST
  ```

- 事前活性化されるオブジェクトの場合、通常その状態はクラッシュが発生したときに回復できません。ただし、これは大きな問題ではありません。このようなオブジェクトは、一般に特定の一連のオペレーションのコンテキスト内で使用され、その後に削除されるからです。オブジェクトの状態は、それらのオペレーションの外部では意味を持ちません。

  サーバがクラッシュし、そのときに削除されたオブジェクトをクライアント・アプリケーションが呼び出そうとする状況を回避するには、事前活性化されるオブジェクトに対する Tobj_ServantBase::activate_object() オペレーションのインプリメンテーションに TobjS::ActivateObjectFailed 例外を追加します。クライアントがサーバ・クラッシュ後にこのようなオブジェクトを呼び出そうとし、TP フレームワークがそのオブジェクトの Tobj_ServantBase::activate_object() オペレーションを呼び出した場合、TP フレームワークはクライアント・アプリケーションに次の例外を返します。

  ```
  CORBA::OBJECT_NOT_EXIST
  ```

- 事前活性化はあまり使用しないでください。すべてのプロセス・バウンド・オブジェクトと同じように、事前活性化によって少ないリソースが事前に割り当てられるからです。

4　マルチスレッド CORBA サーバ・アプリケーションの作成

ここでは、次の内容について説明します。

概要

ここでは、次の内容について説明します。

- はじめに
- マルチスレッド CORBA サーバをサポートするためのメカニズム
- マルチスレッド・システムでのシングル・スレッド・サーバ・アプリケーションの実行

はじめに

複数の独立したスレッドを使用するアプリケーションを設計すると、アプリケーション内の並行性が実現され、全体的なスループットが向上します。複数のスレッドを使用すると、各スレッドが複数の独立したタスクを並列に処理する効率的なアプリケーションを構築できます。マルチスレッドは、次の場合に特に役立ちます。

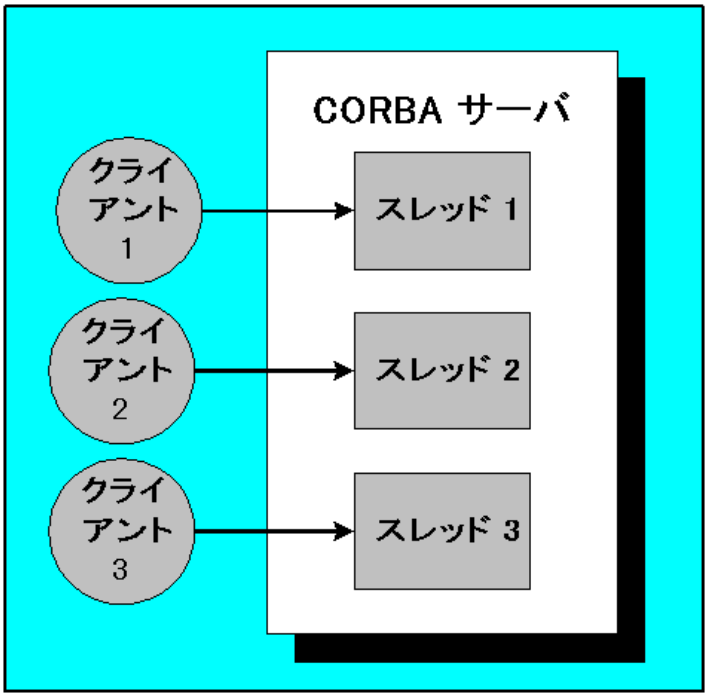
- ほかの処理に必ずしも依存しない一連の長いオペレーションが存在する。
- 共有されるデータの量が少なく、識別可能である。
- 並列に実行できる複数のアクティビティにタスクを分割できる。
- オブジェクトがリエントラントでなければならないときが存在する。

歴史的に見て、業界レベルのマルチスレッド・アプリケーションの設計とインプリメントは複雑なものでした。BEA Tuxedo が提供するサポートは、スレッドを CORBA サーバ環境の内部で管理することによって、こうした複雑さを単純化します。

BEA Tuxedo ソフトウェアは、次のマルチスレッド特性を備えたサーバ・アプリケーションをサポートします (図 4-1 を参照)。

- サーバ・オブジェクトのインスタンスが複数のクライアント要求を同時に処理できる。
- サーバ・オブジェクトが自身に対する再帰的呼び出しを行うことができる。
- サーバ・オブジェクトが独自のスレッドを作成および監視して、サーバント・メソッドの中で並列処理をインプリメントできる。

図 4-1　マルチスレッド CORBA サーバ・アプリケーション



通常、BEA Tuxedo ソフトウェアは、サーバ・アプリケーションの代わりにスレッドを作成および管理します。マルチスレッド・サーバ・アプリケーションをビルドする場合、TP フレームワークの使い方、サーバントのインプリメント方法、および独自のスレッドを作成するオブジェクトの設計方法が変わります。

BEA Tuxedo ソフトウェアを使用すると、要求ごとのスレッド・モデルか、オブジェクトごとのスレッド・モデルのいずれかをインプリメントできます。各モデルについては、「スレッド・モデル」で説明します。

要件、目標、概念

コンピュータ・オペレーションの中には、完了するのに長い時間を要するものがあります。マルチスレッド設計では、オペレーションの要求と完了の間の待機時間を飛躍的に短縮できます。これは、オペレーションが数多くの I/O オペレーションを実行する環境に当てはまります。たとえば、データベースにアクセスするとき、リモート・オブジェクトのオペレーションを呼び出すとき、マルチ・プ

ロセッサ・マシン上の CPU バウンドなどです。サーバ・プロセスでマルチスレッドをインプリメントすると、サーバが一定時間に処理できる要求の数が増加します。

マルチスレッド・サーバ・アプリケーションの主要な要件は、複数のクライアント要求を同時に処理することです。この種のサーバを開発する目的は次のとおりです。

- プログラム設計の単純化

  これは、既存のプログラミング抽象化を使用して複数のサーバ・タスクを独立して実行することにより達成されます。

- スループットの向上

  これは、マルチプロセッサ・ハードウェア・プラットフォームの並列処理のメリットを活用し、計算処理と通信をオーバーラップさせることによって達成されます。

- 知覚応答時間の向上

  独立したスレッドを異なるサーバ・タスクに関連付けることによって、クライアントは長時間にわたって互いをブロックすることがなくなります。

- リモート・プロシージャ・コールと会話のコーディングの単純化

  独立したスレッドを使用して異なるリモート・プロシージャ・コール (RPC) と会話とのやり取りを実現すると、一部のアプリケーションのコーディングが簡単になります。

- 複数アプリケーションへの同時アクセス

  レガシー・アプリケーションまたはレガシー・データベースを CORBA サーバ内でラップすると、インプリメンテーションは一度に複数のレガシー・アプリケーションとやり取りできます。

- 必要なサーバ数の削減

  1 つのサーバで複数のサービス・スレッドをディスパッチできるので、アプリケーションで必要なサーバの数を減らすことができます。

ただし、マルチスレッド設計にはコストがかかります。通常、マルチスレッド・サーバ・アプリケーションでは、シングル・スレッド・サーバより複雑な同期手法が必要になります。アプリケーション開発者は、スレッド・セーフなコードを記述する必要があります。また、スレッドを作成して要求を処理するオーバーヘッドが並列処理のメリットより大きくなる場合もあります。特定の同時実行モデルの実際の性能は、次の要因によって決まります。

- クライアントからの要求の特性

  要求の期間は長いですか、それとも短いですか。

- スレッドのインプリメント方法

  スレッドはオペレーティング・システム・カーネル、ユーザ領域のライブラリ、両方の組み合わせのいずれで管理されますか。

- オペレーティング・システムとネットワークのオーバーヘッド

  接続を繰り返しセットアップおよび切断することによって発生するオーバーヘッドはどのくらいですか。

- 高度なシステム・コンフィギュレーション要因

  複製、動的ロード・バランシング、またはほかの要因が性能に影響を与えますか。

スレッド・ライブラリは同時実行モデルを作成するためのメカニズムを提供しますが、そのメカニズムを適切に使用する方法を最終的に理解するのは開発者です。デザイン・パターンを学習することによって、アプリケーション開発者は微妙な差異を理解し、さまざまな状態に応じた設計を選択できるようになります。

スレッド・モデル

サーバ内の並行性を設計するために使用できるモデルはいくつか存在します。以降の節では、要求ごとのスレッド・モデル、オブジェクトごとのスレッド・モデル、スレッド・プール、および BEA Tuxedo ソフトウェアが各モデルをインプリメントする方法について説明します。特定のサーバは、要求ごとのスレッド・モデルかオブジェクトごとのスレッド・モデルのいずれかに対応するよう設計されます。

要求ごとのスレッド・モデル

このモデルでは、クライアントからの各要求は別々の制御スレッドで処理されます。このモデルは、サーバが一般に複数のクライアントから長期の要求を受信するときに役立ちます。短期の要求の場合、要求ごとに新しいスレッドを作成するオーバーヘッドが大きくなるので、あまり役立ちません。新しい要求が到着するたびに、BEA Tuxedo はその要求をスレッドに関連付けて、その要求を実行します。マルチスレッド・アプリケーション・サーバ・プロセスは一度に複数のスレッドをホストできるので、一度に複数のクライアント要求を同時に実行できま

す。BEA Tuxedo は、要求とスレッドの関連付けを制御します。このため、アプリケーションは BEA Tuxedo より強力な制御を必要としない限り、スレッドを明示的に作成する必要がありません。

要求ごとのスレッド・モデルでは、アプリケーション・サーバをスレッド・セーフにする必要があります。つまり、複数のサーバ・オブジェクト間で共有されるデータへのアクセスを制御するための同時実行メカニズムをインプリメントする必要があります。同時実行制御メカニズムを使用しなければならない場合、アプリケーション開発プロセスが複雑化します。さらに、多数のクライアントが同時に要求を行った場合、この設計ではオペレーティング・システム・リソースが大量に消費される可能性があります。

オブジェクトごとのスレッド・モデル

オブジェクトごとのスレッド・モデルでは、サーバ・プロセス内の活性化された各オブジェクトが常に 1 つのスレッドに関連付けられます。オブジェクトに対する要求ごとに、ディスパッチ・スレッドとオブジェクトとの関連付けが確立されます。同じオブジェクトに対する連続する要求は、別個のスレッドによって処理できます。特定のスレッドを複数のオブジェクトで共有できます。

スレッド・プール

スレッド・プールは、スレッドを管理するコストを削減するための手段です。起動時に、および必要に応じて、利用可能なスレッドのプールにスレッドが作成され、プールからスレッドが割り当てられ、プールにスレッドが戻されます。プール内のスレッドは、以後の要求の処理に必要となるまで待機します。スレッド・プールを使用すると、前述のスレッド・モデルをサポートできます。たとえば、要求ごとのスレッド・モデルで要求に対してスレッドを割り当て、メソッドの実行に使用して、プールに戻すことができます。

スレッドの割り当てと割り当て解除には、特に要求とオブジェクトの存続期間が短い場合、時間とコストがかかるおそれがあります。スレッド・プールは、スレッドを管理するコストを削減するための手段です。起動時に、または必要に応じて、BEA Tuxedo ソフトウェアは利用可能なスレッドのプールにスレッドを作成し、そのプールからスレッドを割り当てて、そのプールにスレッドを戻します。スレッドはプールに存在し、以後の要求の処理に必要となるまで待機します。

アプリケーション・サーバ・プロセス用の BEA Tuxedo スレッド・プールの初期サイズと最大サイズは、サーバ・コンフィギュレーション・ファイルの設定値によって制御されます。起動時に、最小プール・サイズが事前に割り当てられます。処理する要求が到着すると、BEA Tuxedo ソフトウェアはプールからスレッドを割り当ててその要求を処理します。要求の処理に利用可能なスレッドがプー

ルに存在せず、プールに空きがある場合、BEA Tuxedo ソフトウェアは新しいスレッドを作成してその要求を処理します。要求が到着したときに利用可能なスレッドがプールに存在せず、新しいスレッドを作成できない場合、その要求はスレッドが利用可能になるまでキューに格納されます。

スレッド・プールは、サーバ・スレッドのために消費されるシステム・リソースの量を制限したい場合に適しています。スレッド・プールを使用すると、同時要求の数がプール内のスレッドの数を上回るまでクライアント要求が同時に実行されます。

BEA Tuxedo スレッド・プールには、次の特性と振る舞いがあります。

- プールの最大サイズを BEA Tuxedo 管理機能として設定できます。アプリケーション自体に変更を加えずにプールのサイズを調節できます。
- BEA Tuxedo ソフトウェアは、必要に応じてプールからスレッドを割り当てます。スレッドは要求が処理される間使用され、その後プールに戻されます。
- スレッドは、複数の要求とオブジェクトを処理するために連続的に再利用できます。

リエントラント・サーバント

BEA Tuxedo ソフトウェアは、オブジェクトが自身のオペレーションを再帰的に呼び出すための機能を提供します。この機能を使用するときには、オブジェクトをインプリメントする方法に十分な注意を払う必要があります。アプリケーション・コードは、共有される状態データへのアクセスの制御に必要なオペレーティング・システム同時実行メカニズムを採用する必要があるからです。Process または Distribution Adapter デザイン・パターンをインプリメントするオブジェクトを使用するケースなど、オブジェクトの共有状態がほとんどまたはまったく存在せず、リエントラントをサポートするのが比較的簡単な場合があります。

また、BEA Tuxedo ソフトウェアを使用すると、活性化されたオブジェクトのメソッド呼び出しのリエントラントを有効化または禁止することができます。リエントラントはデフォルトによって無効化されています。活性化されたオブジェクトに対する要求が到着したときに、そのオブジェクトが異なるスレッドで別の要求を実行している場合、次の規則が適用されます。

- `_is_reentrant` メソッドが `TRUE` を返した場合、新しいスレッドがプールから割り当てられ、同じサーバント・インスタンスを使用して適切なメソッドに要求がディスパッチされます。複数のスレッドがオブジェクトと対話する

場合にオブジェクトの状態の整合性を保証するのは、サーバント・インプリメンテーション・コードの責任です。

- _is_reentrant メソッドが FALSE を返した場合、サーバントの新しいインスタンスが作成され、メソッドが新しいインスタンスにディスパッチされます。このインスタンスは自動的に削除されません。以後のリエントラント要求はいずれかのインスタンスにディスパッチされます。

注記　リエントラント・サーバント・メカニズムは、PER_REQUEST 同時実行手法が指定された状態でサーバが起動した場合にのみ利用可能です。

このメソッドの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

Current オブジェクト

マルチスレッド CORBA サーバ・アプリケーション環境の最も重要な属性の 1 つは、Current オブジェクトを適切に使用および管理できることです。このため、次のような振る舞いが保証されます。

- 個々のスレッドは、適切なトランザクション・コンテキストとセキュリティ・コンテキストの中で機能します。
- Current オブジェクトは、異なるスレッドからアクセスされたときに適切に振る舞います。

BEA Tuxedo 製品は、OMG 発行の ORB Portability 仕様によって定義されているマルチスレッド・モデルに準拠しています。この仕様は、OMG CORBA 仕様に組み込まれています。BEA Tuxedo 製品では、CORBA::Current から継承されたインターフェイスのオペレーションは、Current オブジェクトが取得されたスレッドに関連付けられている状態ではなく、オペレーションが呼び出されるスレッドに関連付けられている状態にアクセスできます。この振る舞いの理由は次の 2 つです。

- あるスレッドが別のスレッドの状態を操作するのを防ぐため
- メソッドごとにスレッド・コンテキスト内で新しい Current オブジェクトを取得してナロー変換する必要性をなくすため

マルチスレッド環境で使用する場合、次のオブジェクトの振る舞いは ORB Portability 仕様に合致します。

- CosTransactions::Current

- SecurityLevel1::Current
- SecurityLevel2::Current
- PortableServer::Current

たとえば、アプリケーションがあるスレッドから別のスレッドにトランザクションを受け渡す場合、そのアプリケーションは CosTransactions::Current オブジェクトを使用してはなりません。代わりに、アプリケーションはほかのスレッドに CosTransactions::Control オブジェクトを受け渡します。CosTransctions::Current オブジェクトを受け渡すと、受信側のスレッドはそのスレッドに関連付けられているトランザクション状態だけにアクセスできます。

マルチスレッド CORBA サーバをサポートするためのメカニズム

この節では、BEA Tuxedo CORBA に用意されているマルチスレッド・サーバ・アプリケーションをサポートする以下のツール、API、および管理機能の概要について説明します。

- コンテキスト・サービス
- TP フレームワークのクラスとメソッド
- build コマンドの機能
- 管理用ツール

コンテキスト・サービス

オブジェクト・インプリメンテーションに独自のスレッドを作成して管理することを選択できます。ほかのスレッドは、BEA Tuxedo CORBA ソフトウェアによって自動的に管理されます。BEA Tuxedo CORBA ソフトウェアは、作成および管理する各スレッドのコンテキスト情報を内部に保持します。この必須コンテキスト情報は、CORBA 要求の処理中に使用されます。BEA Tuxedo CORBA は、アプリケーションがいつ独自のスレッドを作成して削除するかに関する知識を持ちません。このため、プログラマはコンテキスト・サービス・メカニズムを使用して、BEA Tuxedo サービスの呼び出し前に独自のスレッドを適切に初期化し、スレッドの削除時に不要になったコンテキスト・リソースを解放できます。

次の ORB メソッド・セットは、スレッド管理の要件を満たします。これらのメソッドを、まとめて**コンテキスト・サービス**と呼びます。

- `ORB::get_ctx()`

  オブジェクトがスレッドを作成するときに、オブジェクトは ORB のこのオペレーションを呼び出して、オブジェクトがスレッドに受け渡すことができるシステム・コンテキスト情報を取得します。このオペレーションは、既にコンテキストを持っているスレッドから呼び出される必要があります。たとえば、メソッドがディスパッチされたスレッドはコンテキストを持っています。このオペレーションの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「`ORB::get_ctx()`」を参照してください。

- `ORB::set_ctx()`

  オブジェクトがスレッドを作成すると、そのスレッドは通常 `get_ctx` メソッドを呼び出したスレッドからコンテキスト情報を取得します。作成されたスレッドは、`ORB::set_ctx` を呼び出してそのスレッドが動作する必要があるシステム・コンテキストを設定するときに、そのコンテキストを使用します。このオペレーションの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「`ORB::set_ctx()`」を参照してください。

- `ORB::clear_ctx()`

  作成されたスレッドは、自己の仕事を完了するとこのメソッドを呼び出して自身とシステム・コンテキストとの関連付けを解除します。このオペレーションの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「`ORB::clear_ctx()`」を参照してください。

- `ORB::inform_thread_exit()`

  スレッドは、自己の仕事が完了するとこのメソッドを呼び出して、BEA Tuxedo システムにアプリケーション管理スレッドに関連付けられているリソースを解放できることを通知します。このオペレーションの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「`ORB::inform_thread_exit()`」を参照してください。

TP フレームワークのクラスとメソッド

BEA Tuxedo TP フレームワークの次のクラスとメソッドは、マルチスレッド・サーバ・アプリケーションをサポートします。

- `ServerBase` クラス

  `ServerBase` クラスのデフォルト・インプリメンテーションを上書きするために、アプリケーション開発者は `ServerBase` から継承されたクラスを作成できます。既にサポートされている `ServerBase` メソッドのほかにも、次のメソッドがマルチスレッド・サーバ・アプリケーションをサポートします。

  - `create_servant_with_id()`
  - `thread_initialize()`
  - `thread_release()`

  これらのメソッドを使用すると、アプリケーションのマルチスレッド特性を細かく制御できます。これらのメソッドの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「`ServerBase` インターフェイス」を参照してください。

- `Tobj_ServantBase` クラス

  このクラスは、マルチスレッド・サーバ・アプリケーションをサポートする次のメソッドを提供します。

  - `Tobj_ServantBase::_is_reentrant()`
  - `Tobj_ServantBase::_add_ref()`
  - `Tobj_ServantBase::_remove_ref()`

  これらのメソッドの詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「`Tobj_ServantBase` インターフェイス」を参照してください。

build コマンドの機能

`buildobjserver` コマンドと `buildobjclient` コマンドには、次のスレッド管理機能が組み込まれています。

- `buildobjserver` コマンドには、プラットフォーム固有のスレッド・ライブラリ・サポートが組み込まれています。このため、サーバ・アプリケーションは BEA Tuxedo ソフトウェアのマルチスレッド・サポートとの互換性を保持します。

buildobjserver コマンドには、マルチスレッド・サーバ・アプリケーションまたはシングル・スレッド・サーバ・アプリケーションをビルドするためのコマンド行オプションが組み込まれています。

- buildobjclient コマンドには、プラットフォーム固有のスレッド・ライブラリ・サポートが組み込まれています。このため、クライアント・アプリケーションは BEA Tuxedo ソフトウェアのマルチスレッド・サポートとの互換性を保持します。

管理用ツール

BEA Tuxedo システムは、アプリケーションを開発および実行するためのコンフィギュレーション・ファイルを使用します。通常、アプリケーション開発者がこれらのファイルを作成し、BEA Tuxedo システム管理者が必要に応じてその内容を修正し、アプリケーションとシステムの要件を満たします。

スレッドのサポートに関連する制御パラメータでは、次のことを指定します。

- サーバをシングル・スレッドにするか、マルチスレッドにするか
- オブジェクトのメソッドをディスパッチするためのスレッド・プールのサイズ

UBBCONFIG ファイルのスレッド・パラメータの詳細については、「UBBCONFIG ファイルの例」を参照してください。

マルチスレッド・システムでのシングル・スレッド・サーバ・アプリケーションの実行

BEA Tuxedo CORBA のスレッド・サポートのデフォルトの振る舞いは、シングル・スレッド・サーバ・サポート環境をエミュレートすることです。マルチスレッド環境でシングル・スレッド CORBA アプリケーションを実行する場合は、サーバ・アプリケーション・コードとコンフィギュレーション・ファイルを変更する必要がありません。ただし、既存のシングル・スレッド・アプリケーションを実行する前に、buildobjserver コマンドと buildobjclient コマンドを使用してそのアプリケーションを再ビルドする必要があります。サーバ・アプリケーションのマルチスレッドを有効化しなかった場合、そのアプリケーションはシングル・スレッド・サーバとして動作します。

マルチスレッド CORBA サーバ・アプリケーションの開発とビルド

ここでは、次の内容について説明します。

- buildobjserver コマンドの使い方
- buildobjclient コマンドの使い方
- 非リエントラント・サーバントの作成
- リエントラント・サーバントの作成
- マルチスレッド・システムでのシングル・スレッド・サーバ・アプリケーションの実行

buildobjserver コマンドの使い方

`buildobjserver` コマンドは、次の機能を通じて CORBA サーバ・アプリケーションをサポートします。

- プラットフォーム固有のスレッド・ライブラリ
- マルチスレッド・サポートの指定
- 代替サーバ・クラスの指定

プラットフォーム固有のスレッド・ライブラリ

`buildobjserver` コマンドによって生成されるサーバ・アプリケーションは、適切なプラットフォーム固有のコンパイラ設定を使用してコンパイルされ、適切なプラットフォーム固有のスレッド・サポート・ライブラリを使用してリンクされます。これにより、BEA Tuxedo ソフトウェアが提供する共有ライブラリとの互換性が保証されます。

マルチスレッド・サポートの指定

マルチスレッドをサポートする CORBA サーバ・アプリケーションを作成する場合、アプリケーションをビルドするときに buildobjserver コマンドで -t オプションを指定する必要があります。実行時に、BEA Tuxedo システムは、実行可能プログラムと、CORBA サーバ・アプリケーションのコンフィギュレーション・ファイル UBBCONFIG で選択されたスレッド・モデルとの互換性を検証します。UBBCONFIG ファイルでスレッド・モデルを設定する方法については、「UBBCONFIG ファイルの例」を参照してください。

注記　CORBA サーバ・アプリケーションのビルド時に -t を指定する場合、UBBCONFIG ファイルの MAXDISPATCHTHREADS パラメータを 1 より大きい値に設定する必要があります。それ以外の値を設定すると、CORBA サーバ・アプリケーションはシングル・スレッド・サーバとして動作することになります。

注記　マルチスレッド共同クライアント / サーバ・インプリメンテーションはサポートされていません。

コンフィギュレーション・ファイルに互換性のないスレッド・モデルを指定してシングル・スレッドの実行可能プログラムを起動しようとすると、次のイベントが発生します。

- BEA Tuxedo ソフトウェアは、ログ・ファイルに警告を記録します。
- サーバの実行可能プログラムがシングル・スレッド・サーバとして起動します。

代替サーバ・クラスの指定

ServerBase クラスから継承された独自の Server クラスを実装する場合、buildobjserver コマンドで -b オプションを使用してその代替 Server クラスを指定する必要があります。buildobjserver コマンドは、次の構文で -b オプションをサポートしています。

```
buildobjserver [-v] [-o outfile] [-f {firstfiles|@def-file}]
[-l {lastfiles|@def-file}] [-r rmname] [-b bootserverclass] [-t]
```

上の構文中、*bootserverclass* の値は、CORBA サーバ・アプリケーションの起動時に使用される C++ クラスを指定します。-b オプションを指定しない場合、BEA Tuxedo システムは Server. という名前でクラスのインスタンスを作成します。

-b オプションを指定する場合、Tuxedo システムは代替サーバ・クラスのメイン関数を作成し、プロジェクトは -b オプションで指定した *bootserverclass* の名前を持つヘッダ・ファイルを提供する必要があります。ヘッダ・ファイルには、代替 C++ クラスの定義が含まれます。代替 Server クラスは、ServerBase クラスから継承されなければなりません。

たとえば、コマンド行で -b AslanServer と指定した場合、アプリケーション・プロジェクトは AslanServer.h ファイルを提供する必要があります。AslanServer.h ファイルは、*bootserverclass*.h ファイルの例です。bootserverclass ファイルは、このコード例と同じようなロジックを提供します。

リスト 4-1 bootserverclass.h ファイルの例

```
// ファイル名 :AslanServer.h
#include <Server.h>
class AslanServer : public ServerBase {
    public:
     CORBA::Boolean initialize(int argc, char** argv);
     void release();
     Tobj_Servant create_servant(const char* interfaceName);
     Tobj_Servant create_servant_with_id(const char* interfaceName,
                                         const char* stroid);
     CORBA::Boolean thread_initialize(int argc, char** argv);
     void thread_release();
};
```

buildobjclient コマンドの使い方

buildobjclient コマンドを使用してクライアント・アプリケーションの実行可能プログラムを作成する場合、アプリケーションは適切なプラットフォーム固有のコンパイラ設定を使用してコンパイルされ、使用するオペレーティング・システム用の適切なスレッド・サポート・ライブラリを使用してリンクされます。これにより、BEA Tuxedo ソフトウェアが提供する共有ライブラリとクライアントの互換性が保証されます。

非リエントラント・サーバントの作成

BEA Tuxedo CORBA 環境で CORBA サーバ・アプリケーションを実行するには、buildobjserver コマンドでそのアプリケーションをビルドしておく必要があります。

buildobjserver -t オプションを使用すると、BEA Tuxedo システムに CORBA サーバ・アプリケーションがスレッド・セーフであることを通知できます。-t オプションは、アプリケーションが共有コンテキスト・データ、およびスレッド・セーフではないほかのプログラミング構造を使用しないことを示します。スレッド・セーフではないシングル・スレッド・アプリケーションをマルチスレッド環境で実行する場合、データが破損するおそれがあります。

アプリケーションのコンフィギュレーション・ファイルを更新してマルチスレッド・サポートを有効化したにもかかわらず、サーバ・インプリメンテーションがリエントラントをサポートできることをアプリケーション・コードに指定していない場合、次のことに注意してください。

- メソッドは BEA Tuxedo システムによって割り当てられた任意のスレッドで実行されます。
- サーバント・インプリメンテーション・コードは、オブジェクトがその状態に同時アクセスすることを必ずしも防ぎません。ただし、活性化されたサーバントは、一度に 1 つの実行スレッドに限定されます。
- メソッドが特定のスレッドで実行されることを保証できません。特定のスレッドに依存するか関連付けられるストレージは使用しないでください。
- サーバントの activate_object メソッドまたは deactivate_object メソッドが当初呼び出された要求と同じスレッドで実行されるとは仮定しないでください。
- 追加のアプリケーション管理スレッドをサーバント・メソッドの中に作成できます。オブジェクト・インプリメンテーションは、スレッドが適切に作成、処理、および破棄されることを保証する必要があります。
- アプリケーション管理スレッドには、ほかのオブジェクトの呼び出しを組み込むことができます。
- 同期化のためにシグナルを使用しないでください。シグナルとスレッドの混在はサポートされていません。

注記 プロセスを終了するための SIGKILL シグナルがサポートされています。シングル・スレッドまたはマルチスレッド・アプリケーションに対する SIGIO の使用は、BEA Tuxedo CORBA ではサポートされていません。

- 要求レベルのインターセプタは、そのメソッドによって使用される同じスレッドを通じて BEA Tuxedo CORBA によって呼び出されます。

リエントラント・サーバントの作成

マルチスレッド・リエントラント・サーバントは、次の手順で作成します。

- buildobjserver コマンドと -t オプションを使用して CORBA サーバ・アプリケーションをビルドし、そのアプリケーションの UBBCONFIG サーバ・コンフィギュレーション・ファイルを修正します。
- TobjServantBase::_is_reentrant メソッドを使用してリエントラントを有効化することによって、CORBA サーバ・アプリケーション・コードを更新します。
- UBBCONFIG ファイルに CONCURR_STRATEGY = PER_REQUEST 指定することによって、要求ごとのスレッド・モデルを使用してサーバを起動します。

マルチスレッド・リエントラント・サーバントを作成する場合、そのオブジェクトのインプリメンテーション・コードはそのオブジェクトの状態を保護して、複数のスレッドがそのオブジェクトと対話している間その整合性を保証する必要があります。

クライアント・アプリケーションに関する考慮事項

次に、BEA Tuxedo 環境で実行される CORBA クライアント・アプリケーションに関する考慮事項を挙げます。

- IIOP を使用したマルチスレッド CORBA クライアントがサポートされています。
- マルチスレッド・ネイティブ CORBA クライアントはサポートされていません。
- マルチ・スレッド CORBA クライアントは、単一 Bootstrap オブジェクトに制限されています。

- マルチスレッド CORBA クライアントは、ドメインへの単一ログオンに制限されています。
- スタブ・ベースの呼び出しを使用する CORBA クライアントがサポートされています。
- 動的起動インターフェイス (DII) を使用する CORBA クライアントはサポートされていません。

マルチスレッド simpapp サンプル・アプリケーションのビルドと実行

ここでは、次の内容について説明します。

simpapp マルチスレッド・サンプルについて

BEA Tuxedo ソフトウェアには、クライアント・プログラムと CORBA サーバ・プログラムで構成されるマルチスレッド CORBA サンプル・アプリケーションが用意されています。サーバは、クライアントからアルファベットの文字列を受信し、大文字と小文字の文字列を返します。simpapp_mt のマルチスレッド機能は、並列処理を提供します。この並列処理により、単一のサーバ・プロセスは、複数のオブジェクトまたは単一のオブジェクトに対する複数のクライアントの同時要求を処理できます。

注記　simpapp_mt サンプルのクライアント・アプリケーションは、マルチスレッド・クライアント・アプリケーションではありません。

サンプル・アプリケーションの機能

マルチスレッド・サーバの目的は、1 つまたは複数のクライアントからの要求を並列に処理することです。simpapp_mt サンプル・アプリケーションは、`buildobjserver -t` コマンド行オプションを使用し、`UBBCONFIG` ファイルを使用して同時実行手法を指定することによってマルチスレッド機能を例示する CORBA アプリケーションです。

simpapp_mt サンプルは、最初に `SimplePerObject` というサーバ・プロセスを作成し、次に `SimplePerRequest` というサーバ・プロセスを作成します。クライアントは、最初に `SimplePerRequest` サーバと通信し、次に `SimplePerObject` サーバと通信します。

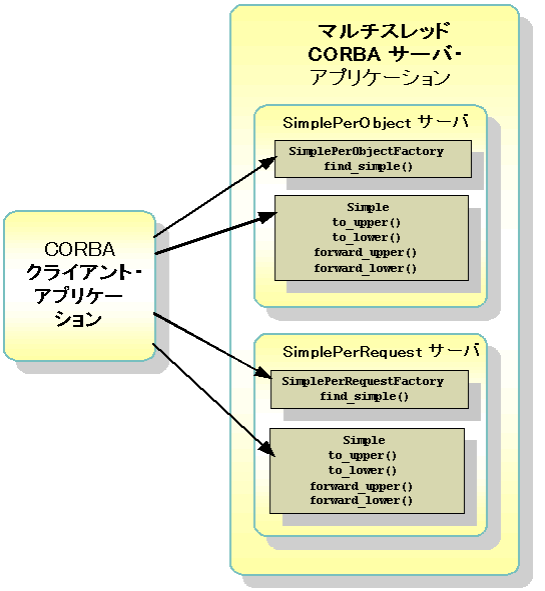
`SimplePerRequest` の要求ごとのスレッド・サーバ・インプリメンテーションは、スレッド初期化メソッドをインプリメントするユーザ定義のサーバ・クラスの使い方を例示します。`SimplePerRequest` サーバ・プロセスは、クライアントからの各要求を独立した制御スレッドで処理します。新しい要求が到着するたびに、その要求を処理するためにスレッド・プールからスレッドが割り当てられます。要求が処理され、応答が送信されると、そのスレッドはプールに戻されます。このモデルは、複数のクライアントからの長期の要求を処理するサーバに適しています。

simpapp_mt サンプル・アプリケーションは、次のメソッドを持つ CORBA オブジェクトのインプリメンテーションを提供します。

- `to_upper` メソッドは、クライアント・アプリケーションからの文字列を受け付けて大文字に変換します。
- `to_lower` メソッドは、クライアント・アプリケーションからの文字列を受け付けて小文字に変換します。
- `forward_upper` メソッドは、サーバの別のインスタンスへのアプリケーション管理スレッドを作成し、クライアントから受信した要求をその新しいサーバ・インスタンスに転送して文字列を大文字に変換します。
- `forward_lower` メソッドは、`Simple` オブジェクトの別のインスタンスを作成し、クライアントから受信した要求をその新しいインスタンスに転送して文字列を小文字に変換します。

図 4-2 に、オブジェクトごとのスレッド・モデルと要求ごとのスレッド・モデルの両方を使用する simpapp_mt サンプル・アプリケーションのオペレーションを示します。

図 4-2　simpapp_mt サンプル・アプリケーション



simpapp マルチスレッド・サンプル・アプリケーションの OMG IDL コード

この章で説明する simpapp マルチスレッド・サンプル・アプリケーションは、次の表に示す CORBA インターフェイスをインプリメントします。

インターフェイス	説明	アクション
`SimplePerRequestFactory`	`Simple` オブジェクトのオブジェクト・リファレンスを作成します。	`find_simple()`
`SimplePerObjectFactory`	`Simple` オブジェクトのオブジェクト・リファレンスを作成します。	`find_simple()`

インターフェイス	説明	アクション
Simple	文字列の大文字 / 小文字を変換します。	to_upper() to_lower() forward_upper() forward_lower()

リスト 4-2 に、simpapp_mt サンプル・アプリケーションの CORBA インターフェイスを定義した simple.idl ファイルの内容を示します。

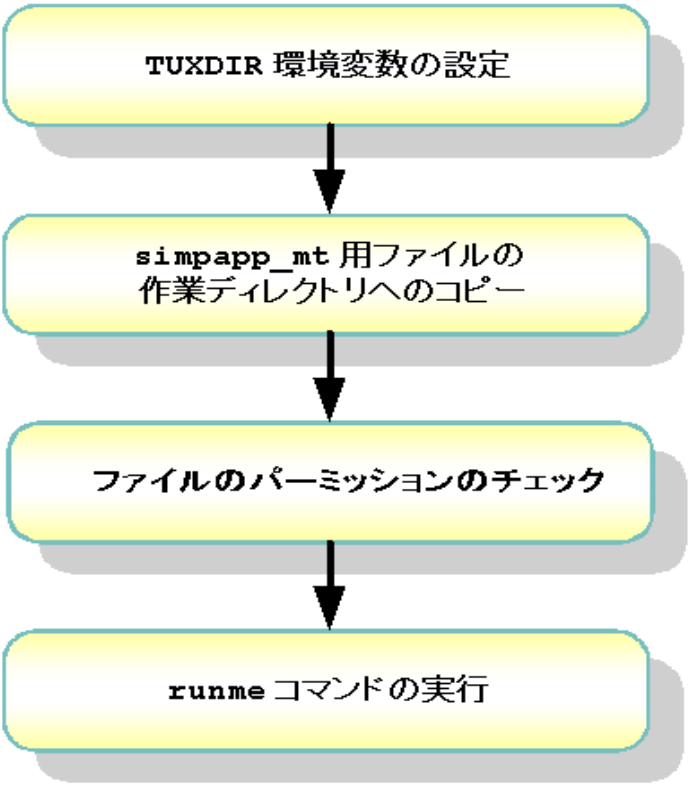
リスト 4-2 simpapp_mt サンプル・アプリケーション用の OMG IDL コード

```
#pragma prefix "beasys.com"

interface Simple
{
      // 文字列を小文字に変換 （新しい文字列を返す）
      string to_lower(in string val);

      // 文字列を大文字に変換 （置換）
      string to_upper(in string val);

      // ほかのサーバを使用して文字列を小文字に変換
      string forward_lower(in string val);

      // ほかのサーバを使用して文字列を大文字に変換
      string forward_upper(in string val);
};

interface SimplePerRequestFactory
{
      Simple find_simple();
};
interface SimplePerObjectFactory
{
      Simple find_simple();
};
```

サンプル・アプリケーションのビルドと実行の方法

この節では、simpapp_mt サンプル・アプリケーションをビルドおよび実行するプロセスをステップごとに説明します。次のフローチャートに、このプロセスの要約を示します。以降の節では、これらのタスクを実行する方法について説明します。

図 4-3　simpapp_mt のビルドおよび実行プロセス



TUXDIR 環境変数の設定

simpapp_mt サンプル・アプリケーションをビルドおよび実行する前に、TUXDIR 環境変数がシステムで設定されていることを確認してください。通常、この環境変数はインストール・プロセス中に設定されます。この環境変数に適切なディレクトリが定義されていることを確認する必要があります。

TUXDIR 環境変数は、BEA Tuxedo ソフトウェアがインストールされているディレクトリ・パスに設定されている必要があります。次に例を示します。

Windows

```
TUXDIR=D:\TUXDIR
```

UNIX

```
TUXDIR=/usr/local/TUXDIR
```

TUXDIR 環境変数の検証

アプリケーションを実行する前に、次の手順を実行してこの環境変数に正確な情報が定義されているかどうかを確認してください。

Windows

echo コマンドを実行して TUXDIR の設定を表示します。

```
prompt> echo %TUXDIR%
```

UNIX

1. プロンプトに対して ksh コマンドを実行して、Korn シェルを起動します。
2. printenv コマンドを実行して TUXDIR の設定を表示します。

   ```
   ksh prompt> printenv TUXDIR
   ```

環境変数の設定の変更

環境変数の設定を変更するには、次の手順を実行します。

Windows

set コマンドを実行して、TUXDIR の新しい値を設定します。

```
prompt> set TUXDIR=directorypath
```

UNIX

1. システム・プロンプトに対して ksh コマンドを実行して、Korn シェルを起動します。
2. ksh プロンプトに対して、export コマンドを入力して TUXDIR 環境変数の値を設定します。

```
ksh prompt> export TUXDIR=directorypath
```

サンプル・アプリケーションの作業ディレクトリの作成

注記　simpapp マルチスレッド・サンプルを実行するときにどのようなファイルが新しく作成されるかを確認できるよう、作業ディレクトリを使用することをお勧めします。runme コマンドを実行したら、インストール・ディレクトリ内のファイル・セットと作業ディレクトリ内のファイル・セットを比較してください。

simpapp マルチスレッド・サンプル・アプリケーションの必須ファイルは、次のディレクトリに格納されています。

Windows

```
%TUXDIR%\samples\corba\simpapp_mt
```

UNIX

```
$TUXDIR/samples/corba/simpapp_mt
```

simpapp マルチスレッド・ファイルをすべて格納する作業ディレクトリを作成します。

Windows

Windows エクスプローラを使用して simpapp_mt ディレクトリをコピーするか、次のようにコマンド・プロンプトを使用します。

1. simpapp_mt ファイルのコピー先となる作業ディレクトリを作成します。

```
> mkdir work_directory
```

2. simpapp_mt ファイルを作業ディレクトリにコピーします。

```
> copy %TUXDIR%\samples\corba\simpapp_mt\* work_directory
```

3. 作業ディレクトリに移動します。

```
cd work_directory
```

4. 作業ディレクトリ内のすべてのファイルを表示します。

```
prompt> dir

makefile.mk              simple_per_object_i.h
makefile.nt              simple_per_object_server.cpp
Readme.txt               simple_per_request_i.cpp
runme.cmd                simple_per_request_i.h
runme.ksh                simple_per_request_server.cpp
simple.idl               simple_per_request_server.h
simple_client.cpp        thread_macros.cpp
simple_per_object_i.cpp  thread_macros.h
```

UNIX

ユーザ・インターフェイス・ツールを使用して simpapp_mt ディレクトリのコピーを作成するか、次のようにコマンド・プロンプトを使用します。

1. simpapp_mt ファイルのコピー先となる作業ディレクトリを作成します。

```
> mkdir work_directory
```

2. すべての simpapp_mt ファイルを作業ディレクトリにコピーします。

```
> cp $TUXDIR/samples/corba/simpapp_mt/* work_directory
```

3. 作業ディレクトリに移動します。

```
cd work_directory
```

4. 作業ディレクトリ内のすべてのファイルを表示します。

```
$ ls

makefile.mk              simple_per_object_i.h
makefile.nt              simple_per_object_server.cpp
Readme.txt               simple_per_request_i.cpp
runme.cmd                simple_per_request_i.h
runme.ksh                simple_per_request_server.cpp
simple.idl               simple_per_request_server.h
simple_client.cpp        thread_macros.cpp
simple_per_object_i.cpp  thread_macros.h
```

表 4-1 に、アプリケーションをビルドおよび実行するための simpapp_mt ファイルとその説明を示します。

表 4-1 simpapp_mt ファイル

ファイル	説明
makefile.mk	(UNIX) simpapp_mt サンプル・アプリケーション用の Makefile。このファイルを使用してアプリケーションをビルドします。
makefile.nt	(Windows) simpapp_mt サンプル・アプリケーション用の Makefile。このファイルを使用してアプリケーションをビルドします。
Readme.txt	simpapp_mt サンプル・アプリケーションのビルドと実行に関する情報を提供する Readme ファイル。
runme.cmd	(Windows) simpapp_mt サンプル・アプリケーションをビルドおよび実行するためのコマンド・ファイル。
runme.ksh	(UNIX) simpapp_mt サンプル・アプリケーションをビルドおよび実行するための Korn シェル・スクリプト。
simple.idl	SimplePerRequestFactory、SimplePerObjectFactory、および Simple インターフェイスを宣言する Object Management Group (OMG) インターフェイス定義言語 (IDL) のコード。
simple_client.cpp	simpapp_mt サンプル・アプリケーション用の CORBA クライアント・プログラム・ソース・コード。

表 4-1 simpapp_mt ファイル (続き)

ファイル	説明
simple_per_object_i.cpp	サーバに組み込まれる Simple サーバントと SimplePerObjectFactory サーバントのインプリメンテーションが含まれるソース・コード。CORBA サーバは、オブジェクトごとのスレッド同時実行手法を使用して起動します。
simple_per_object_i.h	サーバに組み込まれる Simple サーバントと SimplePerObjectFactory サーバントを宣言するためのソース・コード。
simple_per_object_server.cpp	simpapp_mt サンプル・アプリケーション、オブジェクトごとのスレッド同時実行手法用の CORBA サーバ・プログラム・ソース・コード。UBBCONFIG ファイルに CONCURR_STRATEGY = PER_OBJECT を設定します。
simple_per_request_i.cpp	リエントラント・サーバに組み込まれる Simple サーバントと SimplePerRequestFactory サーバントのインプリメンテーションが含まれるソース・コード。リエントラント CORBA サーバは、要求ごとのスレッド同時実行手法を使用して起動します。
simple_per_request_i.h	サーバに組み込まれる Simple サーバントと SimplePerRequestFactory サーバントを宣言するためのソース・コード。
simple_per_request_server.cpp	simpapp_mt サンプル・アプリケーション、要求ごとのスレッド同時実行手法用の CORBA サーバ・プログラム・ソース・コード。UBBCONFIG ファイルに CONCURR_STRATEGY = PER_REQUEST を設定します。

表 4-1 simpapp_mt ファイル (続き)

ファイル	説明
simple_per_request_server.h	simpapp_mt サンプル・アプリケーションのユーザ定義 Server クラスに対して必要な宣言が含まれる bootserverclass.h ファイルの例。
thread_macros.cpp	simpapp_mt サンプル・アプリケーションをサポートするプラットフォーム非依存のスレッド・コンビニエンス・マクロ。
thread_macros.h	スレッド・コンビニエンス・マクロ用のすべてのクラスと変数を宣言するためのソース・コード・ファイル。

すべてのファイルのパーミッションのチェック

simpapp_mt サンプル・アプリケーションをビルドおよび実行するには、作業ディレクトリにコピーしたすべてのファイルに対するユーザ・パーミッションと読み取りパーミッションがなければなりません。パーミッションをチェックし、必要な場合はそれらを変更します。

注記　make ユーティリティがパスに存在することを確認してください。

Windows

```
> attrib -R /S *.*
```

UNIX

```
> /bin/ksh
> chmod u+r work_directory/*.*
```

runme コマンドの実行

この節では、アプリケーションの実行に必要な手順について説明します。次のように runme コマンドを入力します。

Windows

```
> cd work_directory
> ./runme
```

UNIX

```
> /bin/ksh
> cd work_directory
> ./runme.ksh
```

runme コマンドでは、以下の手順が自動的に実行されます。

1. TUXDIR 環境変数をチェックします。
2. このアプリケーションで使用される環境変数を設定します。
3. 適切な bin ディレクトリが PATH にあることを確認します。
4. このスクリプトが実行されるのが初めてではない場合、不要なファイルをディレクトリから削除します。
5. このスクリプトを実行することによって得られる結果を格納するためのディレクトリを作成します。
6. setenv.ksh ファイル (UNIX) または setenv.bat ファイル (Windows) を作成し、このサンプルをステップ・バイ・ステップ・モードでビルドおよび実行できるようにします。
7. このサンプル用の ubb コンフィギュレーション・ファイルを作成します。
8. クライアントのユーザ入力を格納するファイルを作成します。
9. クライアントからの期待される出力を格納するファイルを作成します。
10. サンプルをビルドします。
11. コンフィギュレーション・ファイルをロードします。
12. オブジェクトごとのスレッド・サーバを起動します。
13. 要求ごとのスレッド・サーバを起動します。

14. クライアントを実行し、出力を取得します。

15. その出力と期待された出力を比較します。

16. サーバ・アプリケーションをシャットダウンします。

17. サンプルの実行時に生成されるログを取得します。

18. 結果を保存します。

19. サンプルが正常に実行されたかどうかをユーザに通知します。

simpapp_mt サンプル・アプリケーションは、runme コマンドの実行中に次のメッセージを出力します。

```
Testing simpapp_mt
    cleaned up
    prepared
    built
    loaded ubb
    booted
    ran
    shutdown
    saved results
  PASSED
```

simpapp_mt サンプル・アプリケーションのすべての実行時出力は、作業ディレクトリ内の results ディレクトリに格納されます。実行時に作成された出力を見るには、次のファイルを参照します。

- log ― コンパイル、サーバ起動、またはサーバ・シャットダウンのエラー
- output ― クライアント・アプリケーションの出力と例外
- ULOG.*date* ― サーバ・アプリケーションのエラーと例外

表 4-2 と表 4-3 に、runme コマンドの実行によって作成されるファイルとその説明を示します。

表 4-2 作業ディレクトリに作成されるファイル

ファイル	説明
simple_c.cpp	simple.idl ファイルに対する idl コマンドによって作成されます。このモジュールには、Simple および SimplePerRequestFactory インターフェイス用のクライアント・スタブ機能が含まれます。

表 4-2 作業ディレクトリに作成されるファイル (続き)

ファイル	説明
simple_c.h	simple.idl ファイルに対する idl コマンドによって作成されます。このモジュールには、Simple および SimplePerRequestFactory インターフェイスの定義とプロトタイプが含まれます。
simple_s.cpp	simple.idl ファイルに対する idl コマンドによって作成されます。このモジュールには、Simple_i および SimplePerRequestFactory_i インプリメンテーション用のスケルトン機能が含まれます。
simple_s.h	simple.idl ファイルに対する idl コマンドによって作成されます。このモジュールには、Simple_i および SimplePerRequestFactory_i インターフェイスの定義とプロトタイプが含まれます。
simple_client	simple_c.cpp および simple_client.cpp ファイルに対する buildobjclient コマンドによって作成されます。
simple_per_object_server	simple_c.cpp、simple_s.cpp、simple_per_object_i.cpp、simple_per_object_server.cpp、および thread_macros.cpp ファイルに対する buildobjserver コマンドによって作成されます。
simple_per_request_server	simple_c.cpp、simple_s.cpp、simple_per_request_i.cpp、simple_per_request_server.cpp、および thread_macros.cpp ファイルに対する buildobjserver コマンドによって作成されます。

表 4-2 作業ディレクトリに作成されるファイル (続き)

ファイル	説明
results ディレクトリ	このスクリプトの実行結果を取得するために runme コマンドによって作成されます。
adm ディレクトリ	セキュリティ暗号化キー・データベース・ファイルを格納するために runme コマンドによって作成されます。

表 4-3 results ディレクトリに作成されるファイル

ファイル	説明
input	runme コマンドが C++ クライアント・アプリケーションに提供する入力を格納するために runme コマンドによって作成されます。
output	runme コマンドが C++ クライアント・アプリケーションを実行するときの出力を格納するために runme コマンドによって作成されます。
expected_output	runme コマンドが実行されるときに期待される出力を格納するために runme コマンドによって作成されます。この出力ファイルを比較して、テストに合格したかどうかを決定します。
log	runme コマンドによって生成される出力を格納するために runme コマンドによって作成されます。このコマンドが失敗した場合、このファイルと ULOG ファイルでエラーをチェックします。
setenv.cmd	(Windows) simpapp_mt サンプル・アプリケーションをステップ・バイ・ステップ・モードでビルドおよび実行するために必要な環境変数を設定するためのコマンド・ファイル。

表 4-3 results ディレクトリに作成されるファイル（続き）

ファイル	説明
setenv.ksh	(UNIX) simpapp_mt サンプル・アプリケーションをステップ・バイ・ステップ・モードでビルドおよび実行するために必要な環境変数を設定するためのコマンド・ファイル。
stderr	tmboot によって生成されるメッセージが含まれます。-noredirect サーバ・オプションが UBBCONFIG ファイルに指定されている場合、fprintf メソッドは出力をこのファイルに送信します。
stdout	tmboot によって生成されるメッセージが含まれます。-noredirect サーバ・オプションが UBBCONFIG ファイルに指定されている場合、fprintf メソッドは出力をこのファイルに送信します。
tmsysevt.dat	runme コマンドの tmboot コマンドによって生成されます。このファイルには、TMSYSEVT プロセスで使用されるフィルタ規則と通知規則が含まれます。
tuxconfig	コンフィギュレーション・ファイルのバイナリ・バージョン。
ubb	simpapp_mt サンプル・アプリケーション用の UBBCONFIG ファイル。
ULOG.*date*	実行時エラーを格納するための ULOG ファイル。

サンプル・アプリケーションのステップ・バイ・ステップ実行

この節では、simpapp_mt サンプル・アプリケーションをステップ・バイ・ステップ・モードで実行する方法について説明します。simpapp_mt をステップ・バイ・ステップ・モードで実行するには、あらかじめ runme コマンドを実行しておく必要があります。

次の手順に従って、simpapp_mt アプリケーションを実行します。

1. 環境変数を設定します。

   Windows

   ```
   > ..\results\setenv
   ```

   UNIX

   ```
   > ../results/setenv.ksh
   ```

2. `tmboot -y` を実行してアプリケーションを起動します。次のような情報が表示されます。

```
>tmboot -y
Booting all admin and server processes in /work_directory/results/tuxconfig

Booting admin processes ...

exec BBL -A : process id=212 ... Started.

Booting server processes ...

exec TMSYSEVT -A : process id=289 ... Started.
exec TMFFNAME -A -- -N -M : process id=297 ... Started.
exec TMFFNAME -A -- -N : process id=233 ... Started.
exec TMFFNAME -A -- -F : process id=265 ... Started.
exec simple_per_object_server -A : process id=116 ... Started.
exec simple_per_request_server -A : process id=127 ... Started.
exec ISL -A -- -n //MrBeaver:2468 : process id=270 ... Started.
7 processes started.
>
```

表 4-4 に、tmboot によって起動するサーバ・プロセスを示します。

表 4-4 tmboot によって起動するサーバ・プロセス

プロセス	説明
TMSYSEVT	システム EventBroker。
TMFFNAME	TMFFNAME サーバ・プロセス。 ■ Master NameManager ― -N オプションと -M オプションを指定したときに起動する TMFFNAME サーバ・プロセス。 ■ SLAVE NameManager ― -N オプションだけを指定したときに起動する TMFFNAME サーバ・プロセス。 ■ FactoryFinder オブジェクト ― -F オプションにこのオブジェクトが指定されているときに起動する TMFFNAME サーバ・プロセス。
simple_per_object_server	オブジェクトごとのスレッド・サーバとして起動します。
simple_per_request_server	リエントラントな要求ごとのスレッド・サーバとして起動します。
ISL	IIOP リスナ・プロセス。

3. クライアント・アプリケーションを実行します。

Windows

```
> .\simple_client
```

UNIX

```
> ./simple_client
```

クライアント・アプリケーションを実行すると、次のようなメッセージが表示されます。

リスト 4-3　simpapp_mt クライアントの実行時に表示されるメッセージ

```
Number of simultaneous requests to post (1-50)?
String to convert using thread-per-request server?
Sending 4 deferred forward_lower requests
forward_lower request #0
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #1
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #2
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #3
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
Sending 4 deferred forward_upper requests
forward_upper request #0 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
forward_upper request #1 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
forward_upper request #2 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
forward_upper request #3 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
String to convert using thread-per-object server?
Sending 4 deferred forward_lower requests
forward_lower request #0
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #1
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #2
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #3
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
Sending 4 deferred forward_upper requests
forward_upper request #0 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
forward_upper request #1 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
forward_upper request #2 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ

forward_upper request #3 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
```

サンプル・アプリケーションのシャットダウン

別のサンプル・アプリケーションを実行する前に、simpapp_mt サンプル・アプリケーションをシャットダウンして、不要なファイルを作業ディレクトリからすべて削除する必要があります。

1. アプリケーションを終了するには、tmshutdown -y コマンドを実行します。次のような情報が表示されます。

```
>tmshutdown -y
Shutting down all admin and server processes in /work_directory/results/tuxconfig

Shutting down server processes ...

Server Id = 5 Group Id = SYS_GRP Machine = SITE1:     shutdown succeeded.
Server Id = 2 Group Id = APP_GRP2 Machine = SITE1:    shutdown succeeded.
Server Id = 4 Group Id = SYS_GRP Machine = SITE1:     shutdown succeeded.
Server Id = 3 Group Id = SYS_GRP Machine = SITE1:     shutdown succeeded.
Server Id = 2 Group Id = SYS_GRP Machine = SITE1:     shutdown succeeded.
Server Id = 1 Group Id = SYS_GRP Machine = SITE1:     shutdown succeeded.

Shutting down admin processes ...

Server Id = 0 Group Id = SITE1 Machine = SITE1: shutdown succeeded.
7 processes stopped.
```

2. 作業ディレクトリを元の状態に戻します。

Windows

```
> ..\results\setenv
> make -f clean
```

UNIX

```
> ../results/setenv.ksh
> make -f makefile.mk clean
```

マルチスレッド CORBA サーバ・アプリケーションの管理

ここでは、次の内容について説明します。

スレッド・プール・サイズの指定

スレッド・プールの最小サイズと最大サイズを指定するための MAXDISPATCHTHREADS パラメータと　MINDISPATCHTHREADS パラメータは、UBBCONFIG ファイルの SERVERS セクションに存在します。これらのパラメータの指定例については、リスト 4-4 を参照してください。マルチスレッド CORBA アプリケーションは、これらの値を使用してスレッド・プールを作成および管理します。

MAXDISPATCHTHREADS

MAXDISPATCHTHREADS パラメータは、各サーバ・プロセスで生成される、同時に実行できるディスパッチ・スレッドの最大数です。このパラメータを指定する際には、以下のことに留意してください。

- MAXDISPATCHTHREADS の値によって、受信する要求を格納するために拡大できる最大サイズが決定されます。
- MAXDISPATCHTHREADS のデフォルト値は 1 です。1 より大きい値を指定した場合、特別なディスパッチ・スレッドが作成および使用されます。このディスパッチ・スレッドは、スレッド・プールの最大サイズを決定するスレッド数には含まれません。

  注記　MAXDISPATCHTHREADS に 1 より大きい値を指定し、CONCURR_STRATEGY スレッド・モデル・パラメータの値を指定しな

かった場合、アプリケーションのスレッド・モデルはデフォルトによってオブジェクトごとのスレッドに設定されます。CONCURR_STRATEGY スレッド・モデル・パラメータについては、「スレッド・モデルの指定」を参照してください。

- MAXDISPATCHTHREADS パラメータの値を 1 に設定した場合、CORBA サーバ・アプリケーションをシングル・スレッド・サーバとして構成する必要があります。

  注記 buildobjserver -t を指定してマルチスレッド CORBA サーバ・アプリケーションをビルドした場合、そのサーバはマルチスレッド・モードで動作できます。マルチスレッド CORBA サーバ・アプリケーションとして実行するには、UBBCONFIG ファイルの MAXDISPATCHTHREADS パラメータを 1 より大きい値に設定する必要があります。それ以外の値を設定した場合、サーバ・アプリケーションはシングル・スレッド・モードで動作します。

- MAXDISPATCHTHREADS パラメータに指定する値は、MINDISPATCHTHREADS パラメータの値より小さくしてはなりません。

- あるプロセスで作成可能なスレッドの最大数は、オペレーティング・システム・リソースによって制限されます。MAXDISPATCHTHREADS は、この制限からアプリケーションで必要なアプリケーション管理スレッドの数を差し引いた数より小さくする必要があります。

MAXDISPATCHTHREADS パラメータの値は、ほかのパラメータに影響を与えます。たとえば、MAXACCESSORS パラメータは BEA Tuxedo システムへの同時アクセスの数を制御し、各スレッドは 1 つのアクセサとしてカウントされます。マルチスレッド・サーバ・アプリケーションでは、各サーバが実行するシステム管理スレッドの数を考慮する必要があります。システム管理スレッドとは、アプリケーションによって開始および管理されるスレッドに対して、BEA Tuxedo ソフトウェアによって開始および管理されるスレッドです。内部的には、BEA Tuxedo は利用可能なシステム管理スレッドのプールを管理します。クライアント要求を受信すると、スレッド・プールから利用できるシステム管理スレッドがその要求を実行するようスケジューリングされます。要求が完了すると、システム管理スレッドは利用可能なスレッド・プールに戻されます。

たとえば、システムに 4 つのマルチスレッド・サーバが存在し、各サーバが 50 のシステム管理スレッドを実行するよう構成されている場合、これらのサーバのアクセサ要件は、次の計算によって得られるアクセサの合計です。

```
50 + 50 + 50 + 50 = 200 accessors
```

MINDISPATCHTHREADS

MINDISPATCHTHREADS パラメータは、サーバが最初に起動したときに開始されるサーバ・ディスパッチ・スレッドの数を指定するために使用します。このパラメータを指定する際には、次のことに留意してください。

- MINDISPATCHTHREADS の値によって、スレッド・プール内のスレッドの初期割り当てが決定されます。
- MAXDISPATCHTHREADS が 1 より大きい場合に作成された別個のディスパッチ・スレッドは、MINDISPATCHTHREADS 制限の一部としてカウントされません。
- MINDISPATCHTHREADS に指定する値は、MAXDISPATCHTHREADS に指定する値より大きくしてはなりません。
- MINDISPATCHTHREADS のデフォルト値は 0 です。

スレッド・モデルの指定

スレッド・モデルを指定するには、UBBCONFIG ファイルの SERVERS セクションに定義されている CONCURR_STRATEGY パラメータを設定します。

CONCURR_STRATEGY パラメータを使用して、マルチスレッド CORBA サーバ・アプリケーションが使用するスレッド・モデルを指定します。CONCURR_STRATEGY パラメータは、次のいずれかの値を受け付けます。

- CONCURR_STRATEGY = PER_REQUEST
- CONCURR_STRATEGY = PER_OBJECT

CONCURR_STRATEGY = PER_REQUEST を指定して要求ごとのスレッド・モデルを採用した場合、CORBA サーバ・アプリケーションの各呼び出しはスレッド・プール内の任意のスレッドに割り当てられます。

CONCURR_STRATEGY = PER_OBJECT を指定してオブジェクトごとのスレッド・モデルを採用した場合、活性化された各オブジェクトは常に 1 つのスレッドに関連付けられます。オブジェクトに対する要求ごとに、ディスパッチ・スレッドとオブジェクトとの関連付けが確立されます。

MAXDISPATCHTHREADS の値が 1 より大きく、CONCURR_STRATEGY の値を指定しなかった場合、スレッド・モデルは PER_OBJECT に設定されます。

スレッド・モデルの特性については、「スレッド・モデル」を参照してください。

活性化されたオブジェクトの数の指定

掲示板のアクティブ・オブジェクト・マップ表に格納されるマシンごとのオブジェクトの最大数を指定するには、MAXOBJECTS パラメータを使用します。この値は、コンフィギュレーション・ファイルの RESOURCES セクションか MACHINES セクションのいずれかで設定できます。RESOURCES セクションの MAXOBJECTS number は、システム全体にわたる設定です。システム全体にわたる設定をマシンごとに上書きするには、MACHINES セクションの MAXOBJECTS number を使用します。

システム全体にわたる設定の場合、次のように指定します。

```
*RESOURCES
        MAXOBJECTS number
```

特定のマシンのシステム全体にわたる設定を上書きするには、次のように指定します。

```
*MACHINES
        MAXOBJECTS = number
```

number の値は、オペレーティング・システムのリソースによってのみ制限されます。

UBBCONFIG ファイルの例

リスト 4-4 に、BEA Tuxedo Threads サンプル・アプリケーション用の UBBCONFIG ファイルを示します。スレッドに関連するパラメータは、**太字**で示してあります。

注記 MAXOBJECTS パラメータの値は、マルチスレッド・サーバのオペレーションに影響を与えます。ただし、このパラメータはマルチスレッド・サーバに固有のものではありません。この値は、シングル・スレッド・サーバのオペレーションにも影響を与えます。MAXOBJECTS の値を大きくすると、サーバで消費されるシステム・リソースが増加します。

リスト 4-4　Threads サンプル・アプリケーションの UBBCONFIG ファイル

```
*RESOURCES
     IPCKEY     55432
     DOMAINID   simpapp
     MAXOBJECTS 100
     MASTER     SITE1
     MODEL      SHM
     LDBAL      N

*MACHINES
     "sunstar"
     LMID         = SITE1
     APPDIR       = "/rusers1/lyon/samples/corba/simpapp_mt"
     TUXCONFIG    = "/rusers1/lyon/samples/corba/simpapp_mt/results/tuxconfig"
     TUXDIR       = "/usr/local/TUXDIR"
     MAXWSCLIENTS = 10
     MAXACCESSERS = 200


*GROUPS
     SYS_GRP
        LMID    = SITE1
        GRPNO   = 1
     APP_GRP1
        LMID    = SITE1
        GRPNO   = 2
     APP_GRP2
        LMID    = SITE1
        GRPNO   = 3

*SERVERS
     DEFAULT:
        RESTART = Y
        MAXGEN  = 5
     TMSYSEVT
        SRVGRP  = SYS_GRP
        SRVID   = 1
     TMFFNAME
        SRVGRP  = SYS_GRP
        SRVID   = 2
        CLOPT   = "-A -- -N -M"
     TMFFNAME
        SRVGRP  = SYS_GRP
        SRVID   = 3
        CLOPT   = "-A -- -N"
     TMFFNAME
        SRVGRP  = SYS_GRP
```

```
        SRVID   = 4
        CLOPT   = "-A -- -F"
     simple_per_object_server
        SRVGRP  = APP_GRP1
        SRVID   = 1
        MINDISPATCHTHREADS = 10
        MAXDISPATCHTHREADS = 100
        CONCURR_STRATEGY = PER_OBJECT
        RESTART = N
     simple_per_request_server
        SRVGRP  = APP_GRP2
        SRVID   = 2
        MINDISPATCHTHREADS = 10
        MAXDISPATCHTHREADS = 100
        CONCURR_STRATEGY = PER_REQUEST
        RESTART = N
     ISL
        SRVGRP  = SYS_GRP
        SRVID   = 5
     CLOPT   = "-A -- -n //sunbstar:2468 -d /dev/tcp"

*SERVICES
```

5　セキュリティと CORBA サーバ・アプリケーション

この章では、セキュリティと CORBA サーバ・アプリケーションについて、Security University サンプル・アプリケーションを例にして説明します。この Security サンプル・アプリケーションでインプリメントするセキュリティ・モデルでは、University サンプル・アプリケーションの学生ユーザはそのアプリケーションへのログイン・プロセスの一部として認証を受ける必要があります。

ここでは、次の内容について説明します。

- セキュリティと CORBA サーバ・アプリケーションの概要
- University サーバ・アプリケーションの設計上の考慮事項

セキュリティと CORBA サーバ・アプリケーションの概要

一般に、CORBA サーバ・アプリケーションはセキュリティにほとんど関係しません。BEA Tuxedo ドメインでのセキュリティはシステム管理者によって UBBCONFIG ファイルで指定され、ドメインへのログインや認証はクライアント・アプリケーションによって処理されます。BEA Tuxedo システムでサポートされているセキュリティ・モデルのいずれにも、BEA Tuxedo ドメインで稼働するサーバ・アプリケーションに関する要件はありません。

しかし、使用する CORBA アプリケーションのセキュリティ・モデルをインプリメントまたは改善する際に、オブジェクトを追加したり、既存のオブジェクトにオペレーションを追加したりすることになり、そうして追加されたオブジェクトやオペレーションがサーバ・アプリケーションの管理対象である場合もあります。

この章では、University サーバ・アプリケーションを拡張して学生という概念を追加する方法を説明します。この拡張はクライアント・アプリケーションに組み込まれて、クライアント・アプリケーションのユーザの識別、およびログインの手段となります。

クライアント・アプリケーションで BEA Tuxedo ドメインへの認証を受ける方法の詳細については、『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』を参照してください。BEA Tuxedo ドメインにセキュリティ・モデルをインプリメントする方法の詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

University サーバ・アプリケーションの設計上の考慮事項

Security University サンプル・アプリケーションの設計の要点は、クライアント・アプリケーションのユーザに対してログオンを求め、ログオンしないと何もできないようにすることです。このため、Security サンプル・アプリケーションでは、ユーザの概念を定義する必要があります。

アプリケーションにログオンするために、クライアント・アプリケーションは BEA Tuxedo ドメインのセキュリティ・サービスに以下の情報を提供する必要があります (このアプリケーションの学生ユーザは、ユーザ名とアプリケーション・パスワードのみを提供します)。

- クライアント名
- ユーザ名
- アプリケーション・パスワード

Security サンプル・アプリケーションは、`Registrar` オブジェクトに `get_student_details()` オペレーションを追加します。このオペレーションによって、クライアント・アプリケーションは、BEA Tuxedo ドメインにログオンした後で University データベースから学生の情報を取得することができます。

注記　`get_student_details()` オペレーションは、BEA Tuxedo ドメインのセキュリティ・モデルをインプリメントすることとは無関係です。このオペレーションを追加しても、Security サンプル・アプリケーションに補

足機能を追加するのみです。Security サンプル・アプリケーションに追加されるセキュリティ・モデルの詳細、およびクライアント・アプリケーションが Security サーバ・アプリケーションにログオンする方法の詳細については、『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』を参照してください。

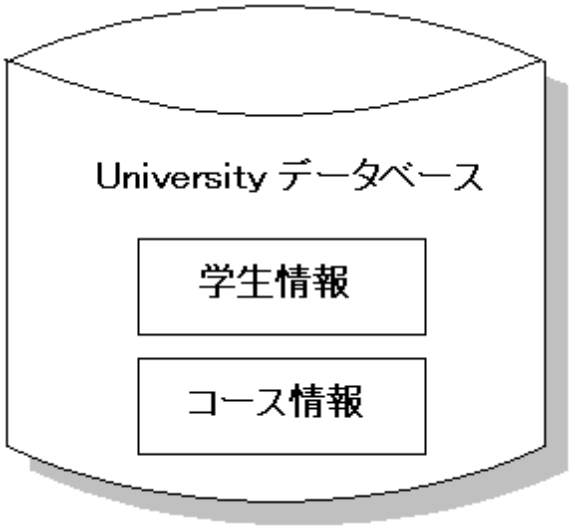
ここでは、次の内容について説明します。

- Security University サンプル・アプリケーションの動作
- クライアント・アプリケーションに学生の詳細情報を返す機能を設計する際の考慮事項

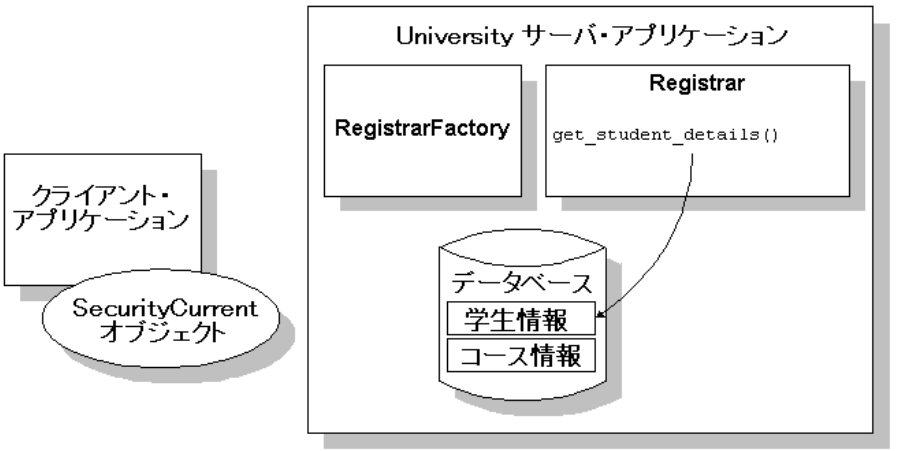
Security University サンプル・アプリケーションのしくみ

Security サンプル・アプリケーションをインプリメントするために、クライアント・アプリケーションではログオンについて学生エンド・ユーザとのやり取りが追加されます。このやり取りではクライアント・マシン上のローカルな SecurityCurrent オブジェクトを使用して PrincipalAuthenticator オブジェクトのオペレーションが呼び出され、BEA Tuxedo ドメインにアクセスするためのログイン手順の一部となります。ユーザ認証のプロセスの後、クライアント・アプリケーションは `Registrar` オブジェクトの `get_student_details()` を呼び出して、学生ユーザについての情報を取得します。

Security サンプル・アプリケーションで使用される University データベースは、コース情報に加えて学生の情報も含むように更新されて、次の図のようになります。



`get_student_details()` オペレーションはデータベースで学生情報の部分にアクセスして、クライアントのログオン・オペレーションに必要な学生情報を取得します。次の図は、Security サンプル・アプリケーションに関係する主要なオブジェクトを示します。



Security サンプル・アプリケーションの一般的な使用のシナリオには、以下のようなイベントのシーケンスが含まれることがあります。

1. クライアント・アプリケーションは、SecurityCurrent オブジェクトへのリファレンスを Bootstrap オブジェクトから取得します。
2. クライアント・アプリケーションは SecurityCurrent オブジェクトを呼び出して、BEA Tuxedo ドメインで要求されるセキュリティのレベルを判別します。
3. クライアント・アプリケーションは学生ユーザについて学生 ID および必要なパスワードを問い合わせます。
4. クライアント・アプリケーションは、学生についての情報を認証サービスから取得して、学生の認証を行います。
5. 認証の処理が正常に終了したら、クライアント・アプリケーションは BEA Tuxedo ドメインにログオンします。
6. クライアント・アプリケーションは `Registrar` オブジェクトの `get_student_details()` オペレーションを呼び出し、学生 ID を渡して、学生についての情報を取得します。
7. `Registrar` オブジェクトはデータベースをスキャンして、クライアント要求にある学生 ID と一致する学生情報を探します。
8. クライアント・アプリケーションの要求で指定されている学生 ID と、データベースにある学生情報の間に一致があれば、`Registrar` オブジェクトはクライアント・アプリケーションに `struct StudentDetails` を返します。学生によって入力された ID がデータベースにある情報と一致しない場合、`Registrar` オブジェクトはクライアント・アプリケーションに CORBA 例外を返します。
9. `Registrar` オブジェクトがクライアント・アプリケーションに `StudentDetails` を返した場合、クライアント・アプリケーションは学生ユーザ個人用の許可メッセージを表示します。

クライアント・アプリケーションに学生の詳細情報を返す機能を設計する際の考慮事項

クライアント・アプリケーションは、ユーザが University アプリケーションの使用を継続できるように、BEA Tuxedo システムにユーザを記録する方法を提供する必要があります。このためには、クライアント・アプリケーションでユーザを識別する手段が必要です。Security サンプル・アプリケーションでは、学生 ID で識別を行います。

Universtity サーバ・アプリケーションに必要な処理は、クライアント・アプリケーションがユーザ認証プロセスを完了できるように、学生 ID に基づいて学生のデータを返すことのみです。このため、Security サンプル・アプリケーションの OMG IDL によって、`Registrar` オブジェクトに `get_student_details()` オペレーションの定義が追加されています。University サーバ・アプリケーションの設計上の主な考慮事項は、既に説明したとおりオペレーション上のシナリオに基づいています。つまり、1 名の学生が一度に 1 つのクライアント・アプリケーションとやり取りできるようにすることです。このため、サーバ・アプリケーションは `get_student_details()` オペレーションのインプリメンテーションで大量のデータ処理を考慮する必要がありません。

`get_student_details()` オペレーションの OMG IDL 定義は次のとおりです。

```
struct StudentDetails
    {
        StudentId         student_id;
        string            name;
        CourseDetailsList registered_courses;
    };
```

6　トランザクションの CORBA サーバ・アプリケーションへの統合

この章では、CORBA サーバ・アプリケーションにトランザクションを統合する方法について、Transactions University サンプル・アプリケーションを例にして説明します。Transactions サンプル・アプリケーションは、学生がコースのセットを登録するプロセスをカプセル化します。Transactions サンプル・アプリケーションでは、CORBA サーバ・アプリケーションをトランザクションに統合するためのすべての方法ではなく、トランザクションの振る舞いの 2 つのモデルを示すことによって、アプリケーション一般、特にオブジェクトの永続状態に対するトランザクションの振る舞いの影響を示します。

ここでは、以下の内容について説明します。

- BEA Tuxedo システムでのトランザクションの概要
- CORBA サーバ・アプリケーションでのトランザクションの設計およびインプリメンテーション
- トランザクションの CORBA クライアントおよびサーバ・アプリケーションへの統合。この節では、以下の内容について説明します。
  - オブジェクトを自動的にトランザクションに関与させる方法
  - オブジェクトのトランザクションへの参加の有効化
  - トランザクションのスコープ指定中にオブジェクトが呼び出されるのを防ぐ方法
  - 進行中のトランザクションからオブジェクトを除外する方法
  - 方針の割り当て
  - XA リソース・マネージャのオープン
  - XA リソース・マネージャのクローズ
- トランザクションとオブジェクト状態の管理
- BEA Tuxedo のトランザクションの使用に関する注意事項

- ユーザ定義例外

また、この章ではユーザ定義例外についても説明します。Transactions サンプル・アプリケーションはユーザ定義例外を利用します。この例外は、クライアント・アプリケーションに返すことができ、クライアントが開始したトランザクションのロールバックを発生させることが可能です。

BEA Tuxedo システムでのトランザクションの概要

BEA Tuxedo システムでは、データベースのトランザクションが正確に完了すること、およびデータベースのトランザクションが高性能なトランザクションの **ACID プロパティ** (原子性、一貫性、独立性、および持続性) のすべてを備えることを保証する手段として、トランザクションが提供されています。つまり、永続ストレージに複数の書き込みオペレーションを実行する上での要件があるので、オペレーションの成功が保証されている必要があります。オペレーションが 1 つでも失敗すれば、一連のオペレーションの全体がロールバックされます。

一般に、トランザクションは次のリストで説明される状況に適しています。それぞれの状況は、BEA Tuxedo システムでサポートされているトランザクション・モデルをカプセル化しています。

- クライアント・アプリケーションは複数の異なるオブジェクトへの呼び出しを行う必要があり、このために 1 つまたは複数のデータベースへの書き込みオペレーションが関係します。呼び出しが 1 つでも失敗すれば、書き込まれているすべての状態 (メモリや、より一般的にはデータベースへの書き込み) をロールバックする必要があります。

  たとえば、旅行代行アプリケーションを検討してみます。クライアント・アプリケーションは、遠隔地への旅行の手配をする必要があります。たとえば、フランスのストラスブールから、オーストラリアのアリス・スプリングスまでの旅行などです。このような旅行では、複数のフライトを個別に予約する必要が生じます。クライアント・アプリケーションは、旅行の各行程を順に予約します。たとえば、ストラスブールからパリ、パリからニューヨーク、ニューヨークからロサンゼルスという順などです。しかし、いずれかのフライトが予約できないとき、クライアント・アプリケーションにはその時点で予約済みのほかのフライトをすべてキャンセルする方法が必要です。たとえば、クライアント・アプリケーションでロサンゼルスからホノルルへの

指定日のフライトを予約できなかったら、クライアント・アプリケーションはその時点までに済ませたフライト予約をキャンセルする必要があります。

- クライアントには、サーバ・アプリケーションによって管理されているオブジェクトと会話して、特定のオブジェクトのインスタンスに対する複数の呼び出しを実行する必要があります。この会話は、以下のような 1 つまたは複数の特徴があります。
  - 連続する呼び出しの各回の間または後には、データがメモリにキャッシュされるか、データベースに書き込まれます。
  - 会話の終了時には、データがデータベースに書き込まれます。
  - クライアントには、各呼び出し間のメモリ内のコンテキストを保持するオブジェクトが必要です。つまり、連続した呼び出しの各回では、会話の間にメモリに保持されているデータが使用されます。
  - 会話の終了時には、会話の間または終了時に発生したデータベース書き込みオペレーションのすべてをクライアントがキャンセルできる必要があります。

  たとえば、インターネット・ベースのオンライン・ショッピング・アプリケーションを検討してみます。クライアント・アプリケーションのユーザがオンライン・カタログを一覧して、複数の購入を選択します。ユーザは、購入したい商品をすべて選択したら、ボタンをクリックして購入を実行し、クレジット・カード情報を入力します。クレジット・カードの確認が失敗したら (たとえば、ユーザのクレジット・カード情報が無効だった場合)、ショッピング・アプリケーションでは、未決済の購入選択をすべてキャンセルしたり、会話で行われたすべての購入トランザクションをロールバックしたりできる必要があります。

- 1 つのオブジェクトに対する単一のクライアント呼び出しのスコープ内で、オブジェクトはデータベース内のデータに対して複数の編集を実行します。編集のいずれかが失敗した場合、オブジェクトはすべての編集をロールバックするメカニズムを必要とします。そして、この状況では、個々のデータベース編集は必ずしも CORBA 呼び出しである必要がありません。

  たとえば、銀行取引アプリケーションを検討してみます。クライアントは、窓口オブジェクトで振替のオペレーションを呼び出します。振替オペレーションは、銀行データベースに対して以下の呼び出しを実行するために窓口オブジェクトを必要とします。
  - ある口座に対する振替元メソッドの呼び出し
  - 別の口座に対する振替先メソッドの呼び出し

銀行データベースの振替先呼び出しに失敗した場合、銀行取引アプリケーションではそれまでの振替元呼び出しをロールバックする機能が必要です。

CORBA サーバ・アプリケーションでのトランザクションの設計およびインプリメンテーション

ここでは、CORBA サーバ・アプリケーションでのトランザクションを設計およびインプリメントする方法について、Transactions University サンプル・アプリケーションを例にして説明します。また、ここでは、Transactions サンプル・アプリケーションの動作と、トランザクションをインプリメントする際の設計上の考慮事項についても説明します。トランザクション全般の詳細については、「トランザクションの CORBA クライアントおよびサーバ・アプリケーションへの統合」を参照してください。

Transactions サンプル・アプリケーションは、学生がコースのセットを登録するプロセスを、トランザクションを利用してカプセル化します。このアプリケーションで使用されているトランザクション・モデルは、前の節で説明したように、会話モデルと、単一の呼び出しがデータベースに対して複数のオペレーションを個別に行うモデルを組み合わせたものです。

この Transactions サンプル・アプリケーションは、Security サンプル・アプリケーションを基に以下の機能を追加したものです。

- 学生は、登録したいコースのリストを提出できます。各コースは番号で示されます。
- リストの各コースについて、University サーバ・アプリケーションは次の項目を確認します。
  - コースが University データベースにあるかどうか。
  - 学生が既に登録されているかどうか。
  - 学生が履修できる単位の最大数を超えているかどうか。
- コースについて確認した結果に問題がなければ、University サーバ・アプリケーションは学生をコースに登録します。

- コースがデータベースに存在しない、または当該の学生が既にコースに登録済みであるといった理由でサーバ・アプリケーションがコースに学生を登録できない場合、サーバ・アプリケーションはクライアント・アプリケーションに対して、登録のプロセスに失敗したコースのリストを返します。このとき、クライアント・アプリケーションは、登録のプロセスが成功したコースについて学生の登録のトランザクションをコミットするか、トランザクション全体をロールバックするかを選択できます。
- 学生が登録可能なコース数の上限を超えたためにコース登録が失敗した場合、サーバ・アプリケーションはクライアント・アプリケーションに CORBA 例外を返し、コースの登録が失敗した理由を示す短いメッセージを提供します。サーバ・アプリケーションは、当該のトランザクションをロールバック・オンリーとしてマークしません。

Transactions サンプル・アプリケーションは、トランザクションをロールバックする 2 つの方法を示します。

- 致命的ではない場合。コースがデータベースに存在しないか、学生が既に登録されているためにコースの登録が失敗する場合は、それらのコースの数がサーバ・アプリケーションからクライアント・アプリケーションに返されます。トランザクションをロールバックするかどうかの決定は、クライアント・アプリケーションのユーザに委ねられます。Transaction クライアント・アプリケーションのコードでは、この場合はトランザクションが自動的にロールバックされます。
- 致命的な場合。登録できる単位の最大数を超えているためにコースの登録が失敗する場合は、サーバ・アプリケーションによって CORBA 例外が生成され、その例外がクライアントに返されます。この場合も、トランザクションをロールバックするかどうかは、クライアント・アプリケーションに依存します。

  したがって、Transactions サンプル・アプリケーションでも、ユーザ定義の CORBA 例外をインプリメントする方法が示されています。たとえば、生徒が登録可能なコースの最大数を超えているコースに登録しようとすると、サーバ・アプリケーションは、`TooManyCredits` 例外を返します。クライアント・アプリケーションは、この例外を受け取ると、自動的にトランザクションをロールバックします。

ここでは、次の内容について説明します。

- Transactions University サンプル・アプリケーションのしくみ
- Transactions University サンプル・アプリケーションで使用されるトランザクション・モデル

Transactions University サンプル・アプリケーションのしくみ

学生による登録のプロセスをインプリメントするために、Transactions サンプル アプリケーションでは以下のことが行われます。

- クライアント・アプリケーションは、TransactionCurrent オブジェクトへのリファレンスを Bootstrap オブジェクトから取得します。
- 学生が登録を希望するコースのリストを提出した時点で、クライアント・アプリケーションでは次のことが行われます。

  a. TransactionCurrent オブジェクトの `Current::begin()` オペレーションを呼び出して、トランザクションを開始します。

  b. `Registrar` オブジェクトの `register_for_courses()` オペレーションを呼び出して、コースのリストを渡します。

- `Registrar` オブジェクトの `register_for_courses()` オペレーションは、登録の要求を処理するためにループを実行し、リストの各コースについて次の処理を繰り返します。

  a. 既に学生が登録されているコースの数を調べます。

  b. 学生が登録済みであるコースを、コースのリストに追加します。

  `Registrar` オブジェクトは、発生する可能性のある次の問題を確認します。これらは、トランザクションのコミットを妨げる問題です。

  - 当該の学生は既にコースに登録済みである。
  - リスト内のコースが存在していない。
  - 学生が履修できる単位の最大数を超えている。

- アプリケーションの OMG IDL で定義されているとおりに、`register_for_courses()` オペレーションはクライアント・アプリケーションにパラメータ `NotRegisteredList` を返し、このパラメータには登録が失敗したコースのリストが含まれています。

NotRegisteredList の値が空であれば、クライアント・アプリケーションはトランザクションをコミットします。

NotRegisteredList の値にコースが 1 つでも含まれていたら、クライアント・アプリケーションは学生に対して、登録が成功したコースについて登録のプロセスを完了するかどうか問い合わせを表示します。ユーザが登録の完了を選択したら、クライアント・アプリケーションはトランザクションをコミットします。ユーザが登録のキャンセルを選択したら、クライアント・アプリケーションはトランザクションをロールバックします。

- 学生が登録の最大数を超えたためにコースの登録が失敗した場合、Registrar オブジェクトは TooManyCredits 例外をクライアント・アプリケーションに返し、クライアント・アプリケーションはトランザクションの全体をロールバックします。

Transactions University サンプル・アプリケーションで使用されるトランザクション・モデル

Transactions サンプル・アプリケーションの設計の要点は、コースの登録を一度に 1 つではなくグループにして処理することです。この設計は、Registrar オブジェクトへのリモート呼び出しの数を最小にすることに役立ちます。

この設計のインプリメンテーションで、Transactions サンプル・アプリケーションはトランザクションの使用の 1 モデルを示します。トランザクションの各モデルの詳細については、「BEA Tuxedo システムでのトランザクションの概要」を参照してください。このモデルは、以下のようになります。

- クライアントは、TransactionCurrent オブジェクト上で begin() オペレーションを呼び出してトランザクションを開始し、次に Registrar オブジェクト上で register_for_courses() オペレーションを呼び出します。

  Registrar オブジェクトは、登録可能なコースについて学生を登録してから、登録のプロセスに失敗したコースのリストを返します。クライアント・アプリケーションは、トランザクションをコミットするか、ロールバックするかを選択できます。トランザクションは、クライアントとサーバ・アプリケーションの間の会話をカプセル化します。

- register_for_courses() オペレーションは、University データベースについて複数回の確認を実行します。確認が 1 つでも失敗すれば、トランザクションをロールバックできます。

University サーバ・アプリケーションのオブジェクト状態の考慮事項

Transactions University サンプル・アプリケーションがトランザクションを扱うので、University サーバ・アプリケーションでは一般にオブジェクト状態への影響を検討する必要があり、特にロールバックの場合を考慮する必要があります。ロールバックが発生する場合、サーバ・アプリケーションは、影響を受けるすべてのオブジェクトの永続状態が適切な状態に復元されることを確認する必要があります。

`Registrar` オブジェクトがデータベースのトランザクションに使用されるので、このオブジェクトについてはトランザクションに関与させるのが正しい設計上の選択です。つまり、このオブジェクトのインターフェイスに `always` トランザクション方針を割り当てることです。このオブジェクトが呼び出されたときにトランザクションのスコープ指定がなされていない場合でも、BEA Tuxedo システムはトランザクションを自動的に開始します。

`Registrar` オブジェクトを自動的にトランザクション・モードになるようにすることで、このオブジェクトによって実行されるすべてのデータベース書き込みオペレーションは、常にトランザクションのスコープ内で行われることになり、クライアント・アプリケーションによってトランザクションが開始されたかどうかは関係がなくなります。サーバ・アプリケーションでは XA リソース・マネージャが使用され、オブジェクトはトランザクション内でデータベースへの書き込みを行うことが保証されているので、このオブジェクトにはロールバックやコミットを扱う必要が一切ありません。これは、XA リソース・マネージャがオブジェクトの代わりにこうしたデータベース・オペレーションを扱うためです。

しかし、`RegisterFactory` オブジェクトは、トランザクションの過程で使用されるデータを管理しないのでトランザクションから除外してもかまいません。このオブジェクトをトランザクションから除外することで、トランザクションに伴う処理のオーバーヘッドを最小にできます。

Regstrar オブジェクトについて定義されるオブジェクト方針

Registrar オブジェクトをトランザクションに関与するようにするために、ICF ファイルは Registrar インターフェイスに always トランザクション方針を指定します。このため、Transaction サンプル・アプリケーションでは、ICF ファイルによって Registrar インターフェイスに次のオブジェクト方針が指定されます。

活性化方針	トランザクション方針
process	always

RegstrarFactory オブジェクトについて定義されるオブジェクト方針

RegistrarFactory オブジェクトをトランザクションから除外するために、ICF ファイルは Registrar インターフェイスに ignore トランザクション方針を指定します。このため、Transaction サンプル・アプリケーションでは、ICF ファイルによって RegistrarFactory インターフェイスに次のオブジェクト方針が指定されます。

活性化方針	トランザクション方針
process	ignore

Transactions サンプル・アプリケーションでの XA リソース・マネージャの使用

Transactions サンプル・アプリケーションは、オブジェクト状態のデータを自動的に処理する Oracle トランザクション・マネージャ・サーバ (TMS) を使用します。XA リソース・マネージャを使用することで、サーバ・アプリケーションによって管理されている個々のオブジェクトがデータベースのデータを読み書きする方法について、次のような特定の要件が生じます。

- XA リソース・マネージャの一部 (たとえば、Oracle) では、すべてのデータベース・オペレーションがトランザクションのスコープ内であることが要求されます。これにより、CourseSynopsisEnumerator オブジェクトは、デー

タベースからの読み取りを行うので、トランザクション内にスコープ指定される必要があります。

- トランザクションがコミットまたはロールバックされるときは、コミットまたはロールバックに伴う永続状態が XA リソース・マネージャによって自動的に処理されます。つまり、トランザクションがコミットされる場合は、XA リソース・マネージャがすべてのデータベース更新を永続的にします。同様に、ロールバックが発生した場合は、XA リソース・マネージャが自動的にデータベースをトランザクション開始前の状態に復元します。

  XA リソース・マネージャのこの特徴によって、ロールバックが発生した場合のオブジェクト状態データの処理に関連する設計上の問題が大幅に簡単になります。トランザクション・オブジェクトはコミットおよびロールバックの処理をいつでも XA リソース・マネージャに委譲できるので、サーバ・アプリケーションをインプリメントする作業が大幅に簡略化されます。

Transactions サンプル・アプリケーションのコンフィギュレーションの要件

University サンプル・アプリケーションは、Oracle のトランザクション・マネージャ・サーバ (TMS) を使用します。Oracle のデータベースを使用するには、サーバ・アプリケーションを構築する際に、Oracle から提供された特定のファイルを含める必要があります。

Transactions サンプル・アプリケーションの構築、設定、および実行の詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。また、オンライン・マニュアルにも、各サンプル・アプリケーション用の `UBBCONFIG` ファイルと、ファイルの各エントリの説明が示されています。

トランザクションの CORBA クライアントおよびサーバ・アプリケーションへの統合

BEA Tuxedo システムは、以下のようにしてトランザクションをサポートします。

- クライアントまたはサーバ・アプリケーションは、TransactionCurrent オブジェクトへの呼び出しを使用してトランザクションを明示的に開始または終了できます。TransactionCurrent オブジェクトの詳細については、『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』および『BEA Tuxedo CORBA トランザクション』を参照してください。

- オブジェクトのインターフェイスにトランザクション方針を割り当てることができます。そうすることで、オブジェクトが呼び出されたときに、トランザクションが既に開始されていなければ、BEA Tuxedo システムがオブジェクトのために自動的にトランザクションを開始し、メソッド呼び出しが完了したときにトランザクションをコミットまたはロールバックすることができるようになります。トランザクションのコミットおよびロールバックに関する処理のすべてをリソース・マネージャに委譲する必要がある場合は、オブジェクトのトランザクション方針を XA リソース・マネージャおよびデータベースと組み合わせて使用します。

- トランザクションに関連するオブジェクトは、トランザクションのロールバックを強制することができます。つまり、トランザクションのスコープ内でオブジェクトが呼び出された後に、そのオブジェクトは TransactionCurrent オブジェクト上で `rollback_only()` オペレーションを呼び出して、トランザクションをロールバック・オンリーとしてマークすることができます。これにより、現在のトランザクションのコミットを防ぐことができます。オブジェクトがトランザクションにロールバックとマークする必要があるのは、ロールバックとマークしなければ、エントリ、通常はデータベースが、壊れたデータや不正確なデータで更新されてしまう可能性がある場合です。

- トランザクションに関係するオブジェクトは、最初に呼び出されたときからトランザクションのコミットまたはロールバックの準備が整うときまでメモリに保持されている可能性があります。コミット直前のトランザクションの場合は、リソース・マネージャによってトランザクションのコミットが準備される直前に、BEA Tuxedo システムによってこうしたオブジェクトがポーリングされます。ここでは、ポーリングとはオブジェクトの

Tobj_ServantBase::deactive_object() オペレーションを呼び出して reason 値を渡すことを意味します。

オブジェクトがポーリングされるとき、そのオブジェクトが TransactionCurrent オブジェクトの rollback_only() オペレーションを呼び出して現在のトランザクションのコミットを拒否する可能性があります。また、現在のトランザクションがロールバックされる場合、オブジェクトにはデータベースへの任意の書き込みを省略することができます。現在のトランザクションのコミットを拒否するオブジェクトがなければ、トランザクションはコミットされます。

以後の節では、必要なトランザクションの振る舞いをオブジェクトに指定するためにオブジェクトの活性化方針とトランザクション方針を使用する方法を説明します。これらの方針は、インターフェイスに適用されるので、このインターフェイスをインプリメントするすべてのオブジェクトでのオペレーションにも適用されるという点に注意してください。

注記　トランザクションに参加できるようにする必要があるオブジェクトがサーバ・アプリケーションによって管理されている場合、そのアプリケーションの Server オブジェクトは TP::open_xa_run() および TP::close_xa_rm() オペレーションを呼び出す必要があります。データベース接続の詳細については、「XA リソース・マネージャのオープン」を参照してください。

オブジェクトを自動的にトランザクションに関与させる方法

BEA Tuxedo システムでは always トランザクション方針が提供されています。この方針はオブジェクトのインターフェイスに定義することができ、オブジェクトが呼び出されて、トランザクションがまだスコープ指定されていないときに BEA Tuxedo システムでトランザクションを自動的に開始させることができます。オブジェクトの呼び出しが完了したら、BEA Tuxedo システムは自動的にトランザクションをコミットまたはロールバックします。この状況では、サーバ・アプリケーションまたはオブジェクト インプリメンテーションのどちらも、TransactionCurrent オブジェクトを呼び出す必要がありません。サーバ・アプリケーションの代わりに、BEA Tuxedo システムが自動的に TransactionCurrent オブジェクトを呼び出します。

オブジェクトのインターフェイスに always トランザクション方針を割り当てることが適切なのは、次の場合です。

- オブジェクトからデータベースへの書き込みがあり、このオブジェクトが呼び出されたときは常に、データベースのコミットまたはロールバックに関するすべての処理を XA リソース・マネージャに委譲する必要がある場合。
- 複数のオブジェクトへの呼び出しを包含する大規模なトランザクションにオブジェクトを含める機会をクライアント・アプリケーションに用意する必要があり、各オブジェクトの呼び出しについては、すべて成功するか、いずれか 1 つでも失敗したらロールバックする必要がある場合。

オブジェクトを自動的にトランザクションに関与させる必要がある場合は、当該のオブジェクトのインターフェイスに関する次の方針をインプリメンテーション・コンフィギュレーション・ファイル (ICF ファイル) に記述します。

活性化方針	トランザクション方針
process、method、または transaction	always

注記 データベース・カーソルは、複数のトランザクションをまたぐことができません。CORBA の University サンプル・アプリケーションにある CourseSynopsisEnumerator オブジェクトでは、一致するコース概要を University データベースで検索するためにデータベース・カーソルが使用されています。データベース・カーソルが複数のトランザクションをまたげないので、CourseSynopsisEnumerator オブジェクトの activate_object() オペレーションは、一致するコース概要をすべてメモリに読み込みます。カーソルはイテレータ・クラスによって管理されているので、CourseSynopsisEnumerator オブジェクトからは見えないことに注意してください。

オブジェクトのトランザクションへの参加の有効化

オブジェクトをトランザクションのスコープ内で呼び出せるようにする必要がある場合は、そのオブジェクトのインターフェイスに optional トランザクション方針を割り当てることができます。optional トランザクション方針が適したオブジェクトは、データベースへの書き込みオペレーションを一切しないものの、トランザクションの間も呼び出し可能にしておく必要があるオブジェクトです。

オブジェクトに optional トランザクション方針を適用するために、そのオブジェクトのインターフェイス用の ICF ファイルで次の方針を指定できます。

活性化方針	トランザクション方針
process、method、または transaction	optional

オブジェクトがデータベースへの書き込みオペレーションを実行するものである場合に、そのオブジェクトをトランザクションに参加させる必要があるならば、always トランザクション方針を割り当てる方が一般にはよりよい選択肢です。しかし、好みに応じて、optional 方針を使用し、TransactionCurrent オブジェクトへの呼び出し内にすべての書き込みオペレーションをカプセル化することも可能です。つまり、トランザクション内での TransactionCurrent オブジェクトのスコープが未指定であれば、データを書き込むオペレーションの内部で、トランザクションの開始時、およびコミットまたはロールバック時にそれぞれ TransactionCurrent オブジェクトを呼び出すことで write 文についてトランザクションのスコープ指定をすることができます。これにより、すべてのデータベース書き込みオペレーションがトランザクションで処理されるようにできます。また、性能も向上します。TransactionCurrent オブジェクトがトランザクションのスコープ内で呼び出されなければ、すべてのデータベース読み取りオペレーションがトランザクションの外部になるので、効率が高くなります。

注記 BEA Tuxedo システムで使用される XA リソース・マネージャの一部では、トランザクションに関与するすべてのオブジェクトについて、データベース書き込みオペレーションに加えて読み取りオペレーションについても、トランザクション内でスコープ指定する必要があります (しかし、それでも独自にトランザクションのスコープ指定はできます)。たとえば、BEA Tuxedo システムで Oracle TMS を使用する場合に、この要件が該当します。使用するオブジェクトに割り当てるトランザクション方針を選択する際は、使用する XA リソース・マネージャの要件を確認してください。

トランザクションのスコープ指定中にオブジェクトが呼び出されるのを防ぐ方法

多くの場合、オブジェクトをトランザクションから除外することが重要になります。除外すべきオブジェクトがトランザクションで呼び出されると、そのオブジェクトが例外を返すことで、トランザクションがロールバックされてしまう可能性があります。BEA Tuxedo システムには never トランザクション方針が用意されていて、これをオブジェクトのインターフェイスに割り当てれば、現在のトランザクションが一時停止中でも、特定のオブジェクトをトランザクションの処理中に呼び出されないようにできます。

このトランザクション方針は、ロールバックできないディスクに永続状態を書き込むオブジェクトに適しています。たとえば、XA リソース・マネージャに管理されていないディスクにデータを書き込むオブジェクトなどです。クライアント / サーバ・アプリケーションがこうした機能 (トランザクション方針) を備えることは、クライアント・アプリケーションの呼び出しでトランザクションがスコープ指定されることをクライアント・アプリケーション自身が知らないか、知ることができない場合に重要です。こうすれば、トランザクションにスコープ指定がされていて、never トランザクション方針が設定されたオブジェクトが呼び出されても、トランザクションのロールバックが可能です。

トランザクションのスコープ指定がされているときにオブジェクトの呼び出しを禁止するには、ICF ファイルで当該のオブジェクトのインターフェイスに次の方針を割り当てます。

活性化方針	トランザクション方針
process または method	never

進行中のトランザクションからオブジェクトを除外する方法

場合によっては、トランザクションの処理中にオブジェクトへの呼び出しを許可しながらも、そのオブジェクトをトランザクションの一部にすることは許可しないという方法が適していることがあります。こうしたオブジェクトがトランザクション中に呼び出されると、そのトランザクションは自動的に一時停止されます。オブジェクトへの呼び出しが完了した後で、トランザクションは自動的に再開されます。この目的のために、BEA Tuxedo システムには ignore トランザクション方針が用意されています。

ignore トランザクション方針は、通常はディスクにデータを書き込まないファクトリのようなオブジェクトに適しています。トランザクションからファクトリを除外することで、トランザクションの間もファクトリがほかのクライアント呼び出しから利用可能になります。さらに、この方針を使用すれば、オブジェクトをトランザクションで呼び出すオーバーヘッドが最小になるので、サーバ・アプリケーションの効率が高まります。

トランザクションがオブジェクトに伝達されることを禁止するには、ICF ファイルで当該のオブジェクトのインターフェイスに次の方針を割り当てます。

活性化方針	トランザクション方針
process または method	ignore

方針の割り当て

ICF ファイルを作成してオブジェクトに方針を指定する方法の詳細については、「ステップ 4: オブジェクトのメモリ内での振る舞い」を参照してください。

XA リソース・マネージャのオープン

オブジェクトのインターフェイスに always または optional トランザクション方針が割り当てられている場合は、Server オブジェクトの Server::initialize() オペレーションで TP::open_xa_rm() オペレーションを呼び出す必要があります。リソース・マネージャは、UBBCONFIG ファイルの GROUPS セクションにある OPENINFO パラメータで指定されている情報に基づいてオープンされます。Server::initialize() オペレーションのデフォルト・バージョンでは、リソース・マネージャが自動的にオープンされる点に注意してください。

ディスクにデータを書き込まず、トランザクションに参加するオブジェクトがある場合—そのオブジェクトは一般に optional トランザクション方針が割り当てられています — それでも、TP:open_xa_rm() オペレーションへの呼び出しを含める必要があります。こうした呼び出しでは、ヌル・リソース・マネージャを指定してください。

XA リソース・マネージャのクローズ

Server オブジェクトの Server::initialize() オペレーションで XA リソース・マネージャをオープンする場合は、Server::release() オペレーションに次の呼び出しを含める必要があります。

```
TP::close_xa_rm();
```

トランザクションとオブジェクト状態の管理

CORBA クライアントおよびサーバ・アプリケーションにトランザクションが必要な場合に、トランザクションとオブジェクト状態の管理を統合する方法は複数あります。一般に、BEA Tuxedo システムでは、オペレーション呼び出しの間のトランザクションについてスコープ指定を自動的に行うようにでき、このときにアプリケーションのロジックを変更したり、オブジェクトが永続状態をディスクに書き込む方法を変更したりする必要はありません。

ここでは、トランザクションとオブジェクト状態の管理に関する重要な項目の一部を説明します。

XA リソース・マネージャへのオブジェクト状態管理の委譲

XA リソース・マネージャを使用すると、たとえば CORBA University サンプル・アプリケーションで使用される Oracle のリソース・マネージャなどであれば、一般に、ロールバックでのオブジェクト状態データの処理に関連する設計上の問題が簡単になります。トランザクション・オブジェクトはコミットおよびロールバックの処理をいつでも XA リソース・マネージャに委譲できるので、このことによってサーバ・アプリケーションをインプリメントする作業が大幅に簡略化されます。トランザクションに関係するプロセス・バウンド・オブジェクトやメソッド・バウンド・オブジェクトでもトランザクション中にデータベースへ

書き込むことができ、トランザクションがロールバックされる際には、データベースへ書き込まれたすべてのデータを元に戻す処理をリソース・マネージャに任せることができます。

トランザクションの作業が完了してから、データベースへの書き込みが始まるまでの待機

transaction 活性化方針は、トランザクションの処理が完了するまでディスクに書き込みたくない、または書き込めないオブジェクト状態をメモリに保持するのに適した方法です。オブジェクトに transaction 活性化方針を割り当てると、そのオブジェクトは次のように振る舞います。

- トランザクションのスコープ内で最初に呼び出されたときにメモリに格納されます。
- トランザクションがコミットまたはロールバックされるまでメモリにとどまります。

トランザクションの作業が完了したら、BEA Tuxedo システムは各トランザクション・バウンド・オブジェクトの Tobj_ServantBase::deactivate_object() オペレーションを呼び出して、DR_TRANS_COMMITTING または DR_TRANS_ABORT のどちらかの reason コードを渡します。変数が DR_TRANS_COMITTING であれば、オブジェクトは自身のデータベース書き込みオペレーションを呼び出すことができます。変数が DR_TRANS_ABORT であれば、オブジェクトは自身のデータベース書き込みオペレーションを省略します。

オブジェクトに transaction 活性化方針を割り当てることは、以下の状況において適切です。

- トランザクションの作業が完了した時点で、オブジェクトの永続状態をディスクに書き込む必要がある。

  これにより、ロールバックの必要があるデータベース書き込みオペレーションが減少するので、性能が向上します。

- オブジェクトに、コミット直前のトランザクションのコミットを拒否する機能を提供する必要がある。

  BEA Tuxedo システムが reason 値として DR_TRANS_COMITTING を渡す場合、オブジェクトは、必要であれば TransactionCurrent オブジェクトの

rollback_only() オペレーションを呼び出すことができます。Tobj_ServantBase::deactivate_object() オペレーションの内部で rollback_only() オペレーションを呼び出した場合、その Tobj_ServantBase::deactivate_object() オペレーションが再度呼び出される点に注意してください。

- 単一のトランザクション中に複数回呼び出される可能性があるオブジェクトがあるが、そのトランザクション中に継続的に活性化と非活性化を繰り返すことのオーバーヘッドを回避する必要がある。

トランザクションのコミットを待機してからデータベースに書き込む機能をオブジェクトに付与するには、当該のオブジェクトのインターフェイスに関する次の方針を ICF ファイルに記述します。

活性化方針	トランザクション方針
transaction	always または optional

注記 トランザクション・バウンド・オブジェクトは、Tobj_ServantBase::deactivate_object() オペレーションの内部からトランザクションを開始したり、ほかのオブジェクトを呼び出したりすることができません。トランザクション・バウンド・オブジェクトが Tobj_ServantBase::deactivate_object() オペレーションの内部から実行できる唯一の有効な呼び出しは、データベースへの書き込みオペレーションのみです。

また、トランザクションに関与するオブジェクトがある場合、そのオブジェクトを管理する Server オブジェクトは、管理されるオブジェクトがディスクに一切のデータを書き込まない場合でも、XA リソース・マネージャをオープンするための呼び出し、およびクローズするための呼び出しをそれぞれ含んでいる必要があります。ディスクにデータを書き込まないトランザクション・オブジェクトについては、ヌル・リソース・マネージャを指定してください。XA リソース・マネージャをオープンする方法およびクローズする方法の詳細については、「XA リソース・マネージャのオープン」および「XA リソース・マネージャのクローズ」を参照してください。

BEA Tuxedo のトランザクションの使用に関する注意事項

CORBA クライアント / サーバ・アプリケーションにトランザクションを統合する際の注意事項は、以下のとおりです。

- 次のトランザクションは、BEA Tuxedo システムでは許可されません。
  - 入れ子になったトランザクション

    既存のトランザクションが既に活性化している場合に、新しいトランザクションを開始することはできません。既存のトランザクションを中断すれば、新しいトランザクションを開始できます。しかし、中断したトランザクションを後で再開できるのは、トランザクションを中断したオブジェクトだけです。
  - 再帰的トランザクション

    トランザクション・オブジェクトは、自身を呼び出すようなオブジェクトを呼び出せません。
- トランザクションを終了できるエンティティは、トランザクションを開始したオブジェクトのみです。厳密には、このオブジェクトはクライアント / アプリケーション、TP フレームワーク、またはオブジェクト (サーバ・アプリケーションによって管理されているもの) のいずれかです。トランザクションのスコープの内部で呼び出されたオブジェクトは、トランザクションを中断および再開できます。トランザクションが一時停止している間、オブジェクトはほかのトランザクションを開始および終了できます。しかし、1 つのオブジェクトの内部でのトランザクションを終了できるのは、その場所でトランザクションを開始したもののみです。
- オブジェクトは、一度に 1 つのトランザクションにのみ関与できます。BEA Tuxedo システムは並列トランザクションをサポートしていません。
- BEA Tuxedo システムは、現在トランザクションに関与しているオブジェクトへの要求をキューに入れません。トランザクションに関与していないクライアント・アプリケーションが、現在トランザクションに関与しているオブジェクト上にあるオペレーションの呼び出しを試みると、クライアント・アプリケーションに対して次のエラー・メッセージが出力されます。

  ```
  CORBA::OBJ_ADAPTER
  ```

トランザクション内にいるクライアントが、現在は別のトランザクションに関与しているオブジェクト上にあるオペレーションの呼び出しを試みると、クライアント・アプリケーションに対して次のエラー・メッセージが出力されます。

```
CORBA::INVALID_TRANSACTION
```

- トランザクション・バウンド・オブジェクトについては、すべての状態の処理を `Tobj_ServantBase::deactive_object()` オペレーションで行うことを検討してください。これにより、`Tobj_ServantBase::deactivate_object()` オペレーションが呼び出された時点でトランザクションの結果が分かるので、オブジェクトが自身の状態を適切に扱うことが簡単になります。
- 複数のオペレーションを持つものの、そのうちの少数のみがオブジェクトの永続状態に影響するメソッド・バウンド・オブジェクトの場合は、次の項目を検討する必要があります。
  - `optional` トランザクション方針を割り当てる。
  - TransactionCurrent オブジェクトを呼び出して、トランザクション内のすべての書き込みオペレーションにスコープ指定を行う。

  オブジェクトがトランザクションの外部で呼び出される場合、オブジェクトはデータを読み取るトランザクションにスコープを指定するためのオーバーヘッドを被りません。この方法で、オブジェクトがトランザクションの内部で呼び出されたかどうかとは無関係に、すべてのオブジェクトの書き込みオペレーションがトランザクションのように処理されます。
- トランザクションのロールバックは非同期です。このため、オブジェクトにとっては、トランザクションのコンテキストが依然として活性化されているときに呼び出される可能性もあります。こうしたオブジェクトの呼び出しを試みると、例外を受け取ります。
- クライアント・アプリケーションではなく BEA Tuxedo システムによって開始されたトランザクションに、`always` トランザクション方針が指定されているオブジェクトが関与している場合は、次のことに注意してください。

  オブジェクトのオペレーションの内部で例外が発生した場合、クライアント・アプリケーションは `OBJ_ADAPTER` 例外を受け取ります。この状況で、BEA Tuxedo システムは自動的にトランザクションをロールバックします。しかし、このクライアント・アプリケーションは、トランザクションが BEA Tuxedo ドメイン内でスコープ指定されたことをまったく認識しません。
- クライアント・アプリケーションがトランザクションを開始し、サーバ・アプリケーションがトランザクションにロールバックとマークして CORBA 例

外を返した場合、クライアント・アプリケーションはトランザクション・ロールバックの例外のみを受け取り、CORBA 例外は受け取りません。

注記　WebLogic Enterprise バージョン 4.2 ソフトウェアでは、この状況を回避する方法がありません。トランザクションを開始する前に、アプリケーションができるだけ周到にデータの検証を実行するようにしてください。

- `TP::deactivateEnable` メソッドを持つトランザクション・オブジェクトに関する次の制限に注意してください。

  トランザクション中に `TP::deactivateEnable` メソッドが呼び出された場合、オブジェクトはトランザクションの終了時に非活性化されます。しかし、`TP::deactivateEnable` メソッドが呼び出された時点とトランザクションがコミットされた時点の間に、オブジェクトで何らかのメソッドが呼び出された場合、そのオブジェクトは決して非活性化されません。

ユーザ定義例外

Transactions サンプル・アプリケーションには、ユーザ定義例外 `TooManyCredits` のインスタンスが含まれています。この例外は、クライアント・アプリケーションが学生をコースに登録しようとするときに、学生が登録可能なコースの最大数を超えてしまうと、サーバ・アプリケーションによってスローされます。クライアント・アプリケーションでこの例外がキャッチされると、クライアント・アプリケーションは学生をコースに登録するトランザクションをロールバックします。ここでは、CORBA クライアント / サーバ・アプリケーションでユーザ定義例外を定義およびインプリメントする方法について、`TooManyCredits` を例にして説明します。

CORBA クライアント / サーバ・アプリケーションにユーザ定義例外を含める作業には、次の手順が関与します。

1. 使用する OMG IDL ファイルで、例外を定義し、例外を使用できるオペレーションを指定します。
2. インプリメンテーション・ファイルで、例外をスローするコードを組み込みます。
3. クライアント・アプリケーションのソース・ファイルで、例外をキャッチおよび処理するコードを組み込みます。

次の節では、最初の 2 つの手順の説明と例を示します。

例外の定義

クライアント / サーバ・アプリケーションで使用する OMG IDL ファイルで、以下の作業をします。

1. 例外を定義し、例外と共に送信されるデータを定義します。たとえば TooManyCredits 例外は、学生が履修できる単位の最大数を表す short 型整数を渡すように定義されています。このため、TooManyCredits 例外の定義には、次の OMG IDL 文が含まれています。

   ```
   exception TooManyCredits
   {
       unsigned short maximum_credits;
   };
   ```

2. 例外をスローするオペレーションの定義に、この例外を含めます。次の例は、Registrar インターフェイスの register_for_courses() オペレーション用の OMG IDL 文を示します。

   ```
   NotRegisteredList register_for_courses(
       in StudentId        student,
       in CourseNumberList courses)
    raises (TooManyCredits);
   ```

例外のスロー

例外を使用するオペレーションのインプリメンテーションでは、次の例のように、例外をスローするコードを記述します。

```
if ( ... ) {
    UniversityZ::TooManyCredits e;
    e.maximum_credits = 18;
    throw e;
    }
```

7 BEA Tuxedo サービスの CORBA オブジェクトへのラッピング

この章では、Wrapper サンプル・アプリケーションを例にして、CORBA サーバ・アプリケーションによって管理されているオブジェクトの内部から BEA Tuxedo サービスを呼び出す方法の 1 つを概説します。

ここでは、次の内容について説明します。

- BEA Tuxedo サービスのラッピングの概要

  この節では、次の内容について説明します。

  - BEA Tuxedo サービスをラッピングするオブジェクトの設計
  - BEA Tuxedo サービス呼び出しをカプセル化するバッファの作成
  - BEA Tuxedo サービスとの間でメッセージを送信するオペレーションのインプリメンテーション
- Wrapper サンプル・アプリケーションの設計上の考慮事項

Wrapper サンプル・アプリケーションは、一連の課金用のオペレーションを BEA Tuxedo ATMI Teller アプリケーションに委譲します。このアプリケーションは、基本的な課金の手続きを実行する一連のサービスを含んでいます。この章で紹介するアプローチは、BEA Tuxedo アプリケーションを BEA Tuxedo ドメインに組み込むテクニックの一例です。

この章で示されている例では、CORBA オブジェクトのオペレーションと、アプリケーション内の特定のサービスへの呼び出しとの 1 対 1 の対応が示されます。つまり、BEA Tuxdedo サービスへの呼び出しが CORBA オブジェクトのオペレーションとしてラッピングされるとも言えます。これは、オブジェクトが処理機能を BEA Tuxedo アプリケーションに委譲することです。BEA Tuxedo サービスのセットを CORBA サーバ・アプリケーションで使用する必要があるときには、この章で説明されるテクニックを試みてください。

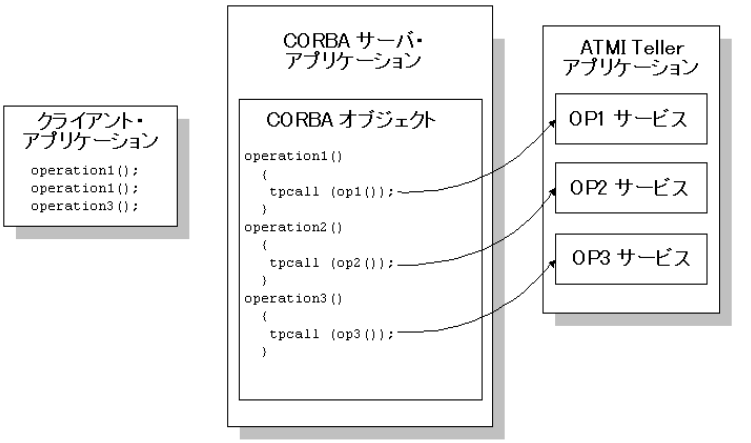
この章では、BEA Tuxedo ATMI アプリケーションの詳細は説明しません。BEA Tuxedo ATMI アプリケーションをビルドおよびコンフィギュレーションする方法、および動作の詳細については、BEA Tuxedo のオンライン・マニュアルに含まれている BEA Tuxedo ATIMI の情報を参照してください。

BEA Tuxedo サービスのラッピングの概要

この章で説明される、BEA Tuxedo サービスのセットをラッピングする処理には以下のステップがあります。

1. BEA Tuxedo システム向けの一連のタスクをオブジェクトのオペレーションとして構造化するオブジェクトを設計します。
2. BEA Tuxedo サービスによって使用されるメッセージ・バッファを作成します。このメッセージ・バッファは、BEA Tuxedo サービスとの間でメッセージの送信および受信に使用されます。このバッファは、アプリケーションのインプリメンテーション・ファイルでオブジェクトのコンストラクタに割り当てることができます。
3. オブジェクト上に、BEA Tuxedo サービスとの間でメッセージを送信および受信するオペレーションをインプリメントします。このステップには、BEA Tuxedo サービスを呼び出す方法のインプリメンテーションを選択することも含まれます。

次の図は、クライアント・アプリケーション、CORBA サーバ・アプリケーションに管理されている CORBA オブジェクト、および BEA Tuxedo ATMI アプリケーションの間の関係を簡単に示します。BEA Tuxedo ATMI アプリケーションは、CORBA オブジェクトから呼び出されるサービスをインプリメントします。



BEA Tuxedo サービスをラッピングするオブジェクトの設計

この章で説明する第一のステップは、BEA Tuxedo ATMI アプリケーションへの呼び出しをラッピングするオブジェクトの設計です。たとえば、Wrapper サンプル・アプリケーションの目的は、学生の登録プロセスに課金の機能を追加することで、これは既存の BEA Tuxedo ATMI Teller アプリケーションに課金用オペレーションのセットを委譲すれば実現できます。

Wrapper サンプル・アプリケーションで使用されている BEA Tuxedo ATMI Teller アプリケーションには、次のサービスが含まれています。

- `CURRBALANCE` ー 指定された口座の現在の残高を取得します。
- `CREDIT` ー 指定された金額 (ドル) を口座に振り込みます。
- `DEBIT` ー 指定された金額 (ドル) を口座から引き落とします。

これらのサービスをラッピングするために、Wrapper サンプル・アプリケーションには新しいインターフェイス `Teller` を定義する別個の OMG IDL ファイルが含まれています。このインターフェイスには、次のオペレーションがあります。

- `get_balance()`
- `credit()`

- debit()

Teller オブジェクトのこれらの各オペレーションは、BEA Tuxedo ATMI Teller アプリケーションにあるサービスへの呼び出しと 1 対 1 でマップされています。

Teller オブジェクトの一般的な使用のシナリオは、以下のようになります。

1. クライアント・アプリケーションによって Registrar オブジェクトの register_for_courses() オペレーションが呼び出されて、このときに学生 ID が要求されます。
2. 登録の処理の一部として、Registrar オブジェクトによって Teller オブジェクトの get_balance() オペレーションが呼び出されて、口座番号が渡されます。
3. Teller オブジェクトの get_balance() オペレーションによって、口座番号がメッセージ・バッファに格納され、このバッファが CURRBALANCE サービスの BEA Tuxedo ATMI Teller アプリケーションに送信されます。
4. BEA Tuxedo ATMI Teller アプリケーションによって、メッセージ・バッファが受信され、その内容が取り出されてから、CURRBALANCE サービスに対して適切な呼び出しが行われます。
5. CURRBALANCE サービスによって、口座の現在の残高が University データベースから取得されて、BEA Tuxedo ATMI Teller アプリケーションに渡されます。
6. BEA Tuxedo ATMI Teller アプリケーションによって、現在の残高がメッセージ・バッファに挿入され、Teller オブジェクトに返されます。
7. Teller オブジェクトによって、現在の残高の合計がメッセージ・バッファから取り出された上で、現在の残高が Registrar オブジェクトに返されます。

Teller オブジェクトおよび Wrapper サンプル・アプリケーションの設計の詳細については、「Wrapper サンプル・アプリケーションの設計上の考慮事項」を参照してください。

BEA Tuxedo サービス呼び出しをカプセル化するバッファの作成

この章で説明される次のステップは、オブジェクトと BEA Tuxedo サービスの間でメッセージの送信に使用されるバッファの作成です。さまざまな BEA Tuxedo ATMI アプリケーションで使用可能なバッファ型は複数あり、この章の例では FML バッファ型に基づいたバッファを使用します。BEA Tuxedo システムのバッファ型の詳細については、BEA Tuxedo の情報を参照してください。

使用するアプリケーション・インプリメンテーション・ファイルで、選択したバッファ型を割り当てる必要があります。割り当てるバッファは `Teller` オブジェクトの特定のインスタンスについて一意である必要がないので、オブジェクトのコンストラクタに割り当てることができます。この割り当てのオペレーションには、一般に、バッファ型の指定、BEA Tuxedo サービスへのプロシージャ・コールに適した任意のフラグの受け渡し、およびバッファ・サイズの指定などが含まれます。

また、使用するインプリメンテーションのヘッダ・ファイルには、バッファを表す変数の定義を追加する必要もあります。

次のコード例では Wrapper アプリケーションの `Teller` オブジェクトのコンストラクタに、BEA Tuxedo のバッファ `m_tuxbuf` が割り当てられています。

```
Teller_i::Teller_i() :
    m_tuxbuf((FBFR32*)tpalloc("FML32", "", 1000))
{
    if (m_tuxbuf == 0) {
        throw CORBA::INTERNAL();
    }
}
```

FML バッファを割り当てる行については、次のことに注意してください。

コード	説明
`tpalloc`	バッファを割り当てます。
`"FML32"`	FML バッファ型を指定します。
`""`	一般に、BEA Tuxedo サービスに渡されるフラグを記述する部分です。この例では、渡されるフラグはありません。

コード	説明
1000	バッファ・サイズをバイト単位で指定します。

また、オブジェクトのインプリメンテーション・ファイルは、Wrapper アプリケーションのインプリメンテーション・ファイルにある次の文のようにして、デストラクタでバッファの割り当てを解除する必要があります。

```
tpfree((char*)m_tuxbuf);
```

BEA Tuxedo サービスとの間でメッセージを送信するオペレーションのインプリメンテーション

次のステップは、BEA Tuxedo ATMI アプリケーションへの呼び出しをラッピングするオブジェクトでのオペレーションをインプリメントすることです。このステップでは、オブジェクトから BEA Tuxedo サービスを呼び出す方法のインプリメンテーションを選択します。Wrapper サンプル・アプリケーションでは、`tpcall` インプリメンテーションが使用されます。

BEA Tuxedo サービスをラッピングするオブジェクトでのオペレーションには、次のことをする文が一般に含まれます。

- BEA Tuxedo サービスへ送信するデータでメッセージ・バッファを満たします。
- BEA Tuxedo サービスを呼び出します。呼び出しには、次の引数が含まれます。
  a. 呼び出される BEA Tuxedo サービス
  b. BEA Tuxedo サービスへ送信されるメッセージ・バッファ
  c. BEA Tuxedo サービスから返されるメッセージ・バッファ
  d. BEA Tuxedo サービスの応答が格納されるバッファのサイズ
- BEA Tuxedo サービスからの応答を取り出します。
- 結果をクライアント・アプリケーションに返します。

次の例は、Wrapper アプリケーションの Teller オブジェクトでの get_balance() オペレーションのインプリメンテーションを示します。このオペレーションでは特定の口座の残高が取得され、BEA Tuxedo サービス CURRBALANCE が呼び出されます。

```
CORBA::Double Teller_i::get_balance(BillingW::AccountNumber account)
{
    // 「in」パラメータ （口座番号） をマーシャリングする
    Fchg32(m_tuxbuf, ACCOUNT_NO, 0, (char*)&account, 0);
    long size = Fsizeof32(tuxbuf);
    // CURRBALANCE Tuxedo サービスを呼び出す
    if (tpcall("CURRBALANCE", (char*)tuxbuf, 0,
                (char**)&tuxbuf, &size, 0) ) {
        throw CORBA::PERSIST_STORE();
    }
    // 「out」パラメータ（現在の残高）のマーシャリングを解除する
    CORBA::Double currbal;
    Fget32(m_tuxbuf, CURR_BALANCE, 0, (char*)&currbal, 0);
    return currbal;
}
```

次のコード例にある文は、メッセージ・バッファ m_tuxbu f を学生の口座番号で満たします。FML の詳細については、『BEA Tuxedo FML リファレンス』を参照してください。

```
Fchg32(m_tuxbuf, ACCOUNT_NO, 0, (char*)&account, 0);
```

次の文は、tpcall のインプリメンテーションを通じて CURRBALANCE BEA Tuxedo サービスを呼び出し、メッセージ・バッファを渡します。また、この文は、BEA Tuxedo サービスの応答が格納される場所も指定します。この例では、要求の送信元となったバッファと同じバッファです。

```
if (tpcall("CURRBALANCE", (char*)tuxbuf, 0,
                (char**)&tuxbuf, &size, 0) ) {
        throw CORBA::PERSIST_STORE();
        }
```

次の文は、返された BEA Tuxedo メッセージ・バッファから残高を取り出します。

```
Fget32(m_tuxbuf, CURR_BALANCE, 0, (char*)&currbal, 0);
```

get_balance() オペレーションの最後の行で、クライアント・アプリケーションへ結果が返されます。

```
return currbal;
```

制限事項

BEA Tuxedo ドメイン内に BEA Tuxedo サービスを組み込む方法については、次の制限事項に注意してください。

- オブジェクトのインプリメンテーションと BEA Tuxedo サービスを同一のサーバ・アプリケーション内で組み合わせることはできません。BEA Tuxedo サービスは、CORBA サーバ・アプリケーションと同一のドメインにある別個の BEA Tuxedo サーバ・アプリケーション内にのみ存在できます。
- BEA Tuxedo サービスを呼び出すオブジェクト内に `tpreturn()` または `tpforward()` の BEA Tuxedo インプリメンテーションを含めることはできません。

Wrapper サンプル・アプリケーションの設計上の考慮事項

Wrapper サンプル・アプリケーションの基本的な設計上の考慮事項は、この節で説明するシナリオに基づいています。学生がコースを登録するとき、`Registrar` オブジェクトが登録プロセスの一部として Teller オブジェクトの呼び出しを実行し、`Teller` オブジェクトは学生の口座にコースの課金をします。

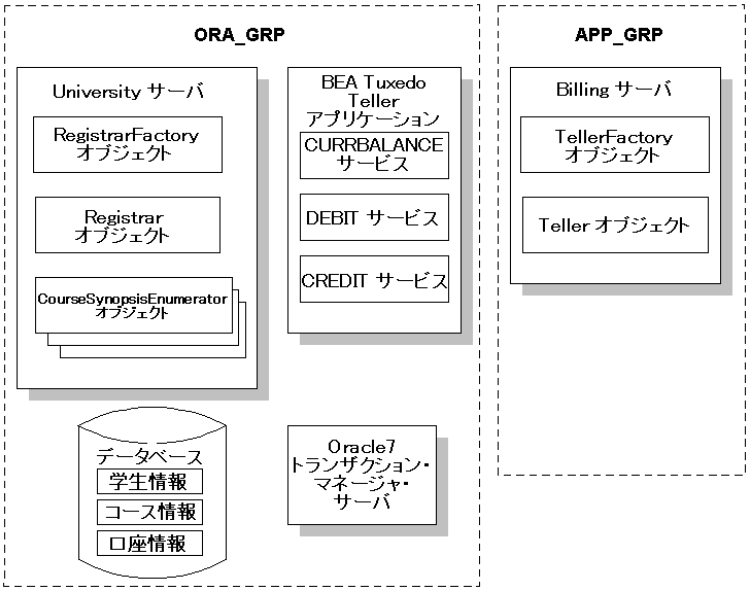
ここでは、Wrapper サンプル・アプリケーションの設計について説明し、Billing というサーバ・アプリケーションをコンフィギュレーションに追加します。このため、Wrapper サンプル・アプリケーションには次の 4 つのサーバ・アプリケーションがあります。

- University。`RegistrarFactory`、`Registrar`、および `CourseSynopsisEnumerator` オブジェクトがあります。
- Billing。`TellerFactory` および `Teller` オブジェクトがあります。
- BEA Tuxedo ATMI Teller アプリケーション。`CURRBALANCE`、`CREDIT`、および `DEBIT` サービスがあります。
- Oracle トランザクション・マネージャ・サーバ (TMS)。

さらに、Wrapper サンプル・アプリケーションの UBBCONFIG ファイルで、次のグループが指定されます。

- ORA_GRP。University サーバ・アプリケーション、BEA Tuxedo ATMI Teller アプリケーション、および Oracle TMS があります。これら 3 つのプロセスは、University データベースでのトランザクションに関係するので、データベースと共に同じグループに存在する必要があります。
- APP_GRP。Billing サーバ・アプリケーションがあります。Billing アプリケーションは University データベースとのやり取りをしないので、このアプリケーションが ORA_GRP に存在する必要はありません。

次の図に、Wrapper サンプル・アプリケーションでの BEA Tuxedo ドメインのコンフィギュレーションを示します。



BEA Tuxedo ATMI アプリケーションを University サンプル・アプリケーションに組み込むことは、Process-Entity デザイン・パターンを使用する観点からは合理的です。一般に BEA Tuxedo ATMI アプリケーションは Process-Entity デザイン・パターンをインプリメントしますが、このデザイン・パターンは University サンプル・アプリケーションでも使用されています。

University データベースが更新されて、各学生の口座情報が格納されている新しいテーブルが含まれます。このため、BEA Tuxedo ATMI Teller アプリケーションのサービスで課金データを処理する際には、University データベースを使用してトランザクションが実行されます。

Wrapper University サンプル・アプリケーションのしくみ

Wrapper サンプル・アプリケーションでの一般的な使用シナリオでは、以下のようなイベントのシーケンスを経ます。

1. 学生のログオン手続きの後で、クライアント・アプリケーションによって Registrar オブジェクトの get_student_details() オペレーションが呼び出されます。get_student_details() オペレーションのインプリメンテーションに含まれているコードによって、次の内容が取得されます。
   - 学生の口座番号。データベースにある学生のテーブルから。
   - 学生の残高。データベースの口座テーブルから。Teller オブジェクト上で get_balance() オペレーションを呼び出して取得されます。
2. 学生は、Basic サンプル・アプリケーションの場合と同様にコースを検索して、登録を希望するコースのリストを作成します。
3. クライアント・アプリケーションは、Transaction サンプル・アプリケーションのシナリオと同様に、Registrar オブジェクトに要求を送信して register_for_courses() オペレーションを呼び出します。この要求にコース番号と学生 ID のみが含められる点も同じです。
4. 学生ごとにコースのリストを登録する際に、register_for_courses() オペレーションは次の処理を呼び出します。
   - Teller オブジェクトの get_balance() オペレーション。学生の口座に滞納がないことを確認します。
   - Teller オブジェクトの debit() オペレーション。Billing サーバ・アプリケーションによって管理され、コースの課金を行います。
5. Teller オブジェクトの get_balance() および debit() オペレーションのそれぞれが、BEA Tuxedo ATMI Teller アプリケーションへの要求を送信します。この要求には、適切な呼び出しが格納された FML バッファがカプセル化されています。これには、BEA Tuxedo ATMI Teller アプリケーションの CURRBALANCE および DEBIT サービスそれぞれへの口座番号呼び出しも含まれます。

6. CURRBALANCE および DEBIT サービスは、それぞれ、適切なデータベース呼び出しを実行して、現在の残高を取得し、学生が登録したコースの課金を反映するように学生の口座から引き落とします。

   学生の口座に滞納があれば、Registrar オブジェクトはクライアント・アプリケーションに DelinquentAccount 例外を返します。この場合、クライアント・アプリケーションはトランザクションをロールバックします。

   debit() オペレーションが失敗した場合、Teller オブジェクトは TransactionCurrent オブジェクトの rollback_only() オペレーションを呼び出します。Teller および Registrar オブジェクトは同一のトランザクション内にスコープ指定されているため、このロールバックは、登録プロセスの全体に影響することで、データベースの不整合 (たとえば、学生がコースに登録したのに、学生の口座残高からコースの分が引き落とされていないなど) を防ぎます。
7. 例外が発生しなければ、Registrar オブジェクトによって学生は希望のコースに登録されます。

Billing サーバ・アプリケーションのインターフェイス定義

以下のインターフェイス定義が、Billing サーバ・アプリケーション用に定義されています。

- TellerFactory オブジェクト。その唯一のオペレーションは find_teller() です。find_teller() オペレーションは、University サーバの RegistrarFactory オブジェクトの find_registrar() オペレーションとまったく同じ動作をします。
- Teller オブジェクト。前述のように、このオブジェクトは次のオペレーションをインプリメントします。
  - debit()
  - credit()
  - current_balance()

  Registrar オブジェクトと同様に、Teller オブジェクトには状態データがなく、一意なオブジェクト ID (OID) もありません。

Wrapper サンプル・アプリケーションの設計上の追加考慮事項

次の追加考慮事項が、Wrapper サンプル・アプリケーションの設計に影響します。

- Registrar オブジェクトには、課金用オペレーションを扱う Teller オブジェクトへ要求を送信する方法が必要です。
- University サーバ・アプリケーションおよび BEA Tuxedo ATMI Teller アプリケーションは、同じデータベースにアクセスできる必要があります。このため、コース登録のトランザクションが適正に動作するには、2 つのサーバ・アプリケーションの両方が Oracle TMS および University データベースと同じサーバ・グループに属している必要があります。

こうした考慮事項の両方により、Wrapper サンプル・アプリケーションの UBBCONFIG ファイルが重要になります。以後の節では、これに関係する設計上の追加考慮事項を詳細に説明します。

Teller オブジェクトへの要求の送信

ここまでは、University サーバ・アプリケーションのすべてのオブジェクトは同じサーバ・プロセス内で定義されていました。そのため、1 つのオブジェクトがほかのオブジェクトに要求を送信する処理は明解でした。次のステップでは、Registrar オブジェクトおよび CouseSynopsisEnumerator オブジェクトを例にして示します。

1. Registrar オブジェクトが、CourseSynopsisEnumerator オブジェクトへのオブジェクト・リファレンスを作成します。
2. 新しく作成されたオブジェクト・リファレンスを使用して、Registrar オブジェクトは CourseSynopsisEnumerator オブジェクトへの要求を送信します。
3. CourseSynopsisEnumerator オブジェクトがメモリに存在しない場合は、TP フレームワークが Server オブジェクトの Server::create_servant() オペレーションを呼び出して、CourseSynopsisEnumerator オブジェクトをインスタンス化します。

しかし、2 つのサーバ・プロセスが実行中で、1 つのプロセスにあるオブジェクトが 2 つめのプロセスによって管理されているオブジェクトに要求を送信する必要がある場合、その手順はやや複雑です。たとえば、別のサーバ・プロセスにあ

るオブジェクトへのオブジェクト・リファレンスを取得するには、重要な前提があります。1 つは、2 番目のサーバ・プロセスの実行中に要求を行うことです。さらに、別のサーバ・プロセスにあるオブジェクト用のファクトリが利用可能である必要があります。

Wrapper サンプル・アプリケーションでは、これを解決するために以下のコンフィギュレーションおよび設計の要素を組み込んでいます。

- University サーバ・アプリケーションは、University Server オブジェクトの `Server::initialize()` オペレーションにある `TellerFactory` オブジェクトへのオブジェクト・リファレンスを取得します。次に、University サーバ・アプリケーションは `TellerFactory` へのオブジェクト・リファレンスをキャッシュします。これにより、`Registrar` オブジェクトが `TellerFactory` を必要とするたびに次の処理を実行せずに済むので、性能の最適化に貢献します。
  - Bootstrap オブジェクトの `resolve_initial_references()` オペレーションを呼び出して、FactoryFinder オブジェクトを取得します。
  - FactoryFinder オブジェクトの `find_one_factory_by_id()` オペレーションを呼び出して、`TellerFactory` オブジェクトへのリファレンスを取得します。
- Billing サーバ・プロセスが、University サーバ・プロセスよりも先に開始されます。続いて `Registrar` オブジェクトによって `TellerFactory` オブジェクトが呼び出されるとき、`Registrar` オブジェクトは `Server::initialize()` オペレーション (前項で説明したオペレーション) によって取得されたオブジェクト・リファレンスを使用します。`UBBCONFIG` ファイルは、各サーバ・プロセスの起動順に指定してください。
- コース登録のプロセス中に課金を処理するために、`Registrar` オブジェクトの `register_for_courses()` および `get_student_details()` オペレーションは、`Teller` オブジェクトのオペレーションを呼び出すコードを含むように変更されます。

例外の処理

Wrapper サンプル・アプリケーションは、学生による課金の総額が限度を超えた場合を処理できるように設計されています。学生が University で許される限度を超えてコースを登録しようとした場合、`Registrar` オブジェクトはユーザ定義の `DelinquentAccount` 例外を生成します。この例外がクライアント・アプリ

ケーションに返されると、クライアント・アプリケーションによってトランザクションがロールバックされます。ユーザ定義例外をインプリメントする方法の詳細については、「ユーザ定義例外」を参照してください。

Wrapper サンプル・アプリケーションのインターフェイスに対するトランザクション方針の設定

Wrapper サンプル・アプリケーションの性能に影響するもう 1 つの考慮事項は、アプリケーションのオブジェクトのインターフェイスに適したトランザクション方針を設定することです。`Registrar`、`CourseSynopsisEnumerator`、および `Teller` オブジェクトは、`always` トランザクション方針によってコンフィギュレーションされます。`RegistrarFactory` および `TellerFactory` オブジェクトは `ignore` トランザクション方針によってコンフィギュレーションされます。これらのオブジェクトには、トランザクションに含まれる必要がないので、トランザクションのコンテキストが伝達されないようになります。

University および Billing サーバ・アプリケーションのコンフィギュレーション

前述のように、Billing サーバ・アプリケーションは、University データベースおよび University アプリケーション、BEA Tuxedo ATMI Teller アプリケーション、および Oracle トランザクション・マネージャ・サーバ (TMS) アプリケーションを含むグループとは別のグループでコンフィギュレーションされます。

しかし、Billing サーバ・アプリケーションは学生をコースに登録するトランザクションに参加するので、Billing サーバ・アプリケーションは Server オブジェクトの `TP::open_xa_rm()` および `TP::close_xa_rm()` オペレーションへの呼び出しを含む必要があります。これは、任意のトランザクションに含まれるオブジェクトを管理するあらゆるサーバ・アプリケーションに対する要件です。そのようなオブジェクトがデータベースに対する読み取りまたは書き込みのオペレーションを一切実行しない場合は、次の場所にヌル・リソース・マネージャを指定することができます。

- `UBBCONFIG` ファイル内の適切なグループ定義
- サーバ・アプリケーションをビルドする際の `buidobjserver` コマンドの引数

Wrapper サンプル・アプリケーションの構築、設定、および実行の詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。

8 BEA Tuxedo の CORBA サーバ・アプリケーションのスケーリング

この章では、BEA Tuxedo システムの主要なスケーラビリティ機能を利用して CORBA サーバ・アプリケーションのスケーラビリティを高める方法について、Production University サンプル・アプリケーションを例にして説明します。Production サンプル・アプリケーションは、これらのスケーラビリティ機能を利用して以下の目標を達成します。

- 並列処理機能を追加して、BEA Tuxedo ドメインで複数のクライアント要求を同時に処理できるようにします。
- Production サンプル・アプリケーションのサーバ・アプリケーション上の処理負荷を複数のマシンに分散させます。

ここでは、次の内容について説明します。

- BEA Tuxedo システムで利用可能なスケーラビリティ機能の概要
- BEA Tuxedo サーバ・アプリケーションのスケーリング。この節では、次の内容について説明します。
  - サーバ・プロセスおよびサーバ・グループの複製
  - オブジェクト状態管理によるアプリケーションのスケーリング
  - ファクトリ・ベース・ルーティング
- Production サーバ・アプリケーションをさらにスケーリングする方法
- 状態を持たないオブジェクトまたは状態を持つオブジェクトの選択

BEA Tuxedo システムで利用可能なスケーラビリティ機能の概要

高度にスケーラブルなアプリケーションのサポートは、BEA Tuxedo システムの長所の 1 つです。多くのアプリケーションは、1 ～ 10 のサーバ・プロセス、および 10 ～ 100 のクライアント・アプリケーションが動作している環境で良好に機能します。しかし、ビジネス環境では、アプリケーションは次のレベルの処理をサポートする必要があります。

- 数百のサーバ・プロセス
- 数万のクライアント・アプリケーション
- 数百万のオブジェクト

このような要件を負ったアプリケーションをデプロイすると、リソースの不足および性能のボトルネックがすぐに表面化します。BEA Tuxedo システムではこうした大規模なデプロイメントが複数の方法でサポートされていて、この章ではそれらのうち次の 3 つについて説明します。

- サーバ・プロセスおよびサーバ・グループの複製
- オブジェクト状態管理
- ファクトリ・ベース・ルーティング

アプリケーションを高度にスケーラブルにするために BEA Tuxedo システムで提供されているほかの機能には、IIOP リスナ / ハンドラがあります。概要については『BEA Tuxedo CORBA アプリケーション入門』、詳細な説明については『BEA Tuxedo アプリケーションの設定』を参照してください。また、『BEA Tuxedo CORBA アプリケーションのスケーリング、分散、およびチューニング』も参照してください。

BEA Tuxedo サーバ・アプリケーションのスケーリング

ここでは、非常に大規模な処理機能を実現できるようにアプリケーションをスケーリングする方法について、Production サンプル・アプリケーションを例にして説明します。Production サンプル・アプリケーションの設計上の基本的な目標は、対応できるクライアント・アプリケーションの数を大幅に増やすために、次のように対処することです。

- 同じインターフェイスをインプリメントする複数のオブジェクト上のクライアント要求を 1 台のマシン上で並行処理します。
- 一部の学生の要求を 1 台のマシンに割り当て、残りの学生の要求は別のマシンに割り当てます。
- 処理負荷を分担するためのマシンを追加します。

こうした設計目標に対応するために、Production サンプル・アプリケーションでは次の処理をします。

- University、Billing、および BEA Tuxedo Teller アプリケーションのサーバ・プロセスを、それらがコンフィギュレーションされているグループ内で複製します。
- 上記のグループを追加のマシン上に複製します。
- サーバ・プロセスが同時に管理できるクライアント要求の数を増やすために、状態を持たないオブジェクト・モデルをインプリメントします。
- 次のオブジェクトに一意なオブジェクト ID (OID) を割り当てて、それらのオブジェクトを各自のグループで同時に複数回インスタンス化できるようにします。これにより、次のオブジェクトはプロセス単位ではなくクライアント・アプリケーション単位で利用可能になり、並行処理の機能に対応できます。
  - `RegistrarFactory`
  - `Registrar`
  - `TellerFactory`
  - `Teller`

- 一部の学生の要求を 1 台のマシンに割り当て、別の学生の要求をほかのマシンに割り当てるために、ファクトリ・ベース・ルーティングをインプリメントします。

注記 Production サンプル・アプリケーションを使いやすくするために、このアプリケーションは 1 台のマシン上で 1 つのデータベースを使用して実行できるように BEA Tuxedo ソフトウェア・キット上でコンフィギュレーションされています。しかし、この章で説明する例は、このアプリケーションを 2 台のマシン上で 2 つのデータベースを使用して実行する方法を示します。

Production サンプル・アプリケーションの設計では、複数台のマシン上で複数のデータベースを使用して実行するコンフィギュレーションが可能になるように設定されています。複数のマシンおよびデータベースを使用するようにコンフィギュレーションを変更する作業には、`UBBCONFIG` ファイルの変更とデータベースの分離が関係し、これらの詳細は「Production サーバ・アプリケーションをさらにスケーリングする方法」で説明されています。

以後の各節では、Production サンプル・アプリケーションのスケーラビリティの目標を達成するために、複製されたサーバ・プロセスおよびサーバ・グループの使用、オブジェクト状態管理、およびファクトリ・ベース・ルーティングについて説明します。次の節では、Production サンプル・アプリケーションにインプリメントされている OMG IDL の変更について説明します。

Production サンプル・アプリケーションの OMG IDL の変更

Production サンプル・アプリケーションの OMG IDL への変更は、`RegistrarFactory` オブジェクトの `find_registrar()` オペレーション、および `TellerFactory` オブジェクトの `find_teller()` オペレーションに限定されます。これら 2 つのオペレーションはそれぞれ、ファクトリ・ベース・ルーティングのインプリメンテーションに必要な学生 ID と口座番号を要求するように変更されます。Production サンプル・アプリケーションでのファクトリ・ベース・ルーティングのインプリメンテーションおよび使用については、「ファクトリ・ベース・ルーティング」を参照してください。

サーバ・プロセスおよびサーバ・グループの複製

BEA Tuxedo システムでは、サーバ・アプリケーションのコンフィギュレーションに多様な選択肢が用意されています。たとえば、以下のコンフィギュレーションが可能です。

- 1 台のマシンに、1 つのインターフェイスをインプリメントする 1 つのサーバ・プロセス。
- 1 台のマシンに、1 つのインターフェイスをインプリメントする複数のサーバ・プロセス。
- 1 台のマシンに、複数のインターフェイスをインプリメントする複数のサーバ・プロセス。ファクトリ・ベース・ルーティングの有無は任意。
- 複数台のマシンに複数のインターフェイスと複数のサーバ・プロセス。ファクトリ・ベース・ルーティングの有無は任意。

以上をまとめると、次のようになります。

- クライアント / サーバ・アプリケーションに並行処理の機能を追加するには、使用するサーバ・プロセスを複製します。
- デプロイメント環境にマシンを追加するには、グループを追加した上でファクトリ・ベース・ルーティングをインプリメントします。

次の節では、複製されたサーバ・プロセスおよびグループについて説明するほか、それらを BEA Tuxedo システムにコンフィギュレーションする方法についても説明します。

複製されたサーバ・プロセス

使用するアプリケーションでサーバ・プロセスを複製すると、以下の利点があります。

- サーバ・アプリケーションに着信する要求の負荷のバランシングを行う方法が得られます。BEA Tuxedo ドメインのサーバ・グループへの要求が着信すると、BEA Tuxedo システムはその要求をグループ内で負荷が最も低いと見なされるサーバへルーティングします。
- サーバ・アプリケーションの性能が改善されます。一度に 1 つのクライアント要求を処理できる 1 つのサーバ・プロセスを持つ代わりに、複数のクライアント要求を同時に処理できる複数のサーバ・プロセスを利用可能にできま

す。この方法が正常に機能するには、各オブジェクトが一意である必要があります。そのためには、サーバ・アプリケーションのファクトリで一意な OID を割り当ててください。

- サーバ・イメージの 1 つが停止した場合でも、有効なフェイルオーバー保護が得られます

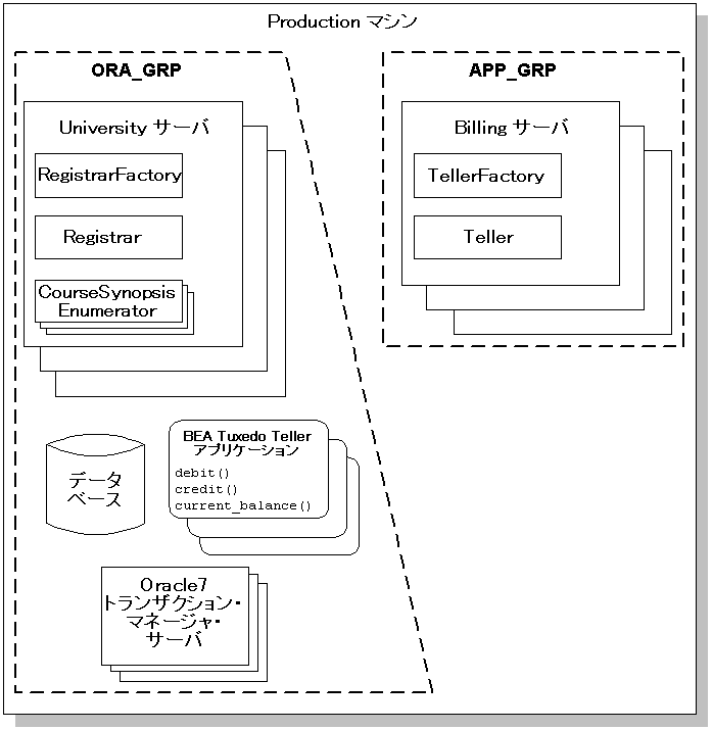
複製されたサーバ・プロセスの利点を完全に活用するには、使用するサーバ・アプリケーションによってインスタンス化される各オブジェクトのすべてが一意な ID を持つようにしてください。こうすれば、オブジェクト上でのクライアント呼び出しによって必要なオブジェクトがインスタンス化される処理が、利用可能な複数のサーバ・プロセスの境界内で実行されるので、既に活性化されているオブジェクトのためのキューに入れられずに済みます。

図 8-1 は、次のことを示します。

- University サーバ・アプリケーション、BEA Tuxedo Teller アプリケーション、および Oracle TMS の各サーバ・プロセスは、ORA_GRP グループ内で複製されます。
- Billing サーバ・プロセスは、APP_GRP グループ内で複製されます。

この図では、2 つのグループは 1 台のマシンで実行されているものとして示されます。

図 8-1 Production サンプルの複製されたサーバ・グループ



これらのグループのいずれかに要求が着信したとき、BEA Tuxedo ドメインには要求を処理できる利用可能なサーバ・プロセスが複数あり、BEA Tuxedo ドメインは最も負荷の少ないサーバ・プロセスを選択することができます。

図 8-1 では、次の点に注意してください。

- どの時点でも、任意のサーバ・プロセス内にある `RegistrarFactory`、`Registrar`、`TellerFactory`、または `Teller` オブジェクトのインスタンスは 1 つだけです。
- どの University サーバ・プロセスにも、任意の数の `CourseSynopsisEnumerator` オブジェクトが存在できます。

複製されたサーバ・グループ

サーバ・グループの概念は BEA Tuxedo システムに固有のもので、CORBA のインプリメンテーションに追加されます。サーバ・グループは、BEA Tuxedo システムのスケーラビリティ機能の重要な要素です。基本的に、デプロイメント・システムにマシンを追加する場合は、グループを追加する必要があります。

図 8-2 は、別のマシンに複製された Production サンプル・アプリケーションのグループを示します。このアプリケーションの UBBCONFIG ファイルに指定されているとおり、ORA_GRP2 と APP_GRP2 があります。

図 8-2　マシン間でのサーバ・グループの複製

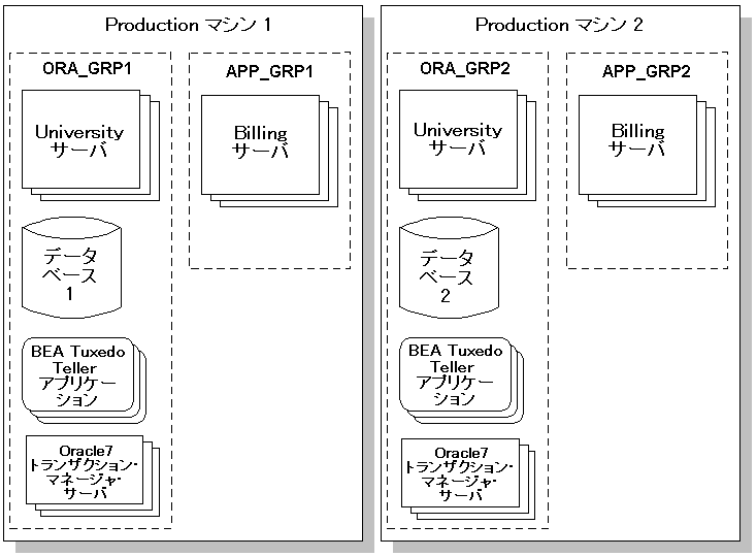


図 8-2 では、Production マシン 1 と 2 のグループの内容で異なるのはデータベースのみです。Production マシン 1 用のデータベースには、学生のサブセットについて、学生情報および口座情報が含まれています。Production マシン 2 用のデータベースには、学生の別のサブセットについて、学生情報および口座情報が含まれています。両方のデータベースにあるコース情報のテーブルは同じ内容です。データベースにある学生情報が、同じデータベース内の口座情報とは完全に無関連であることに注意してください。

サーバ・グループのコンフィギュレーションの方法、実行場所、および複製の方法は、UBBCONFIG ファイルに指定されています。サーバ・グループを複製するときは、次のことが可能です。

- 任意のアプリケーションまたはアプリケーションのセットに関する処理負荷を、別のマシンにも分散する方法があります。
- ファクトリ・ベース・ルーティングを使用して、任意のインターフェイスへの要求の 1 セットを 1 台のマシンに送信し、同じインターフェイスへの要求の別のセットは別のマシンへ送るということができます。

複数のサーバ・グループを持つと、次のような影響があります。

- クライアント要求が BEA Tuxedo ドメインに着信したとき、BEA Tuxedo システムはオブジェクト・リファレンスに指定されたグループ ID を確認します。
- BEA Tuxedo ドメインは、要求がルーティングされたグループ内で最も負荷の低い、要求を処理できるサーバ・プロセスに要求を送信します。

Production サンプル・アプリケーションがファクトリ・ベース・ルーティングを使用して複数台のマシンにアプリケーションの処理負荷を分散させる方法の詳細については、「ファクトリ・ベース・ルーティング」を参照してください。

複製されたサーバ・プロセスおよびグループのコンフィギュレーション

使用する BEA Tuxedo ドメインにある複製されたサーバ・プロセスおよびグループをコンフィギュレーションするには、以下の手順に従います。

1. アプリケーションの UBBCONFIG ファイルを、「ワードパッド」などのテキスト・エディタで開きます。
2. GROUPS セクションに、コンフィギュレーションするグループの名前を指定します。
3. SERVERS セクションに、複製するサーバ・プロセスに関する次の情報を入力します。
   - サーバ・アプリケーションの名前。
   - GROUP パラメータ。サーバ・プロセスが属するグループの名前を指定します。複数のグループに関係するサーバ・プロセスを複製する場合は、各グループに 1 つずつサーバ・プロセスを指定します。
   - SRVID パラメータ。数値識別子を指定して、サーバ・プロセスに一意な ID を割り当てます。

- MIN パラメータ。アプリケーションの起動時に開始されるサーバ・プロセスのインスタンスの数を指定します。
- MAX パラメータ。同時に実行できるサーバ・プロセスの最大数を指定します。

MIN および MAX パラメータは、指定されたオブジェクトへの要求をサーバ・アプリケーションでどの程度まで並行処理できるかを決定します。実行時には、必要に応じて、システム管理者がリソースのボトルネックを調べて追加のサーバ・プロセスを起動することができます。このアプリケーションは、システム管理者がスケーリングできるように設計されています。

次の例は、Production サンプル・アプリケーションの UBBCONFIG ファイルから、GROUPS および SERVERS セクションを示します。

```
*GROUPS
    APP_GRP1
        LMID      = SITE1
        GRPNO     = 2
        TMSNAME   = TMS
    APP_GRP2
        LMID      = SITE1
        GRPNO     = 3
        TMSNAME   = TMS
    ORA_GRP1
        LMID      = SITE1
        GRPNO     = 4
        OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
        CLOSEINFO = ""
        TMSNAME   = "TMS_ORA"
    ORA_GRP2
        LMID      = SITE1
        GRPNO     = 5
        OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
        CLOSEINFO = ""
        TMSNAME   = "TMS_ORA"


*SERVERS
    # デフォルトでは、各サーバのインスタンスを 2 個活性化
    # して、管理者による活性化については各サーバの
    # インスタンスを 5 個まで可能する
    DEFAULT:
        MIN     = 2
        MAX     = 5
    tellp_server
        SRVGRP  = ORA_GRP1
        SRVID   = 10
        RESTART = N
    tellp_server
        SRVGRP  = ORA_GRP2
        SRVID   = 10
```

```
        RESTART = N
    billp_server
        SRVGRP  = APP_GRP1
        SRVID   = 10
        RESTART = N
    billp_server
        SRVGRP  = APP_GRP2
        SRVID   = 10
        RESTART = N
    univp_server
        SRVGRP  = ORA_GRP1
        SRVID   = 20
        RESTART = N
univp_server
        SRVGRP  = ORA_GRP2
        SRVID   = 20
        RESTART = N
```

オブジェクト状態管理によるアプリケーションのスケーリング

「第 1 章 CORBA サーバ・アプリケーションの概念」で説明したように、オブジェクト状態管理は大規模なクライアント / サーバ・システムでは根本的に重要な検討事項です。それは、こうしたシステムでは最適なスループットおよび応答時間を実現することが重要であるためです。この節では、BEA Tuxedo サーバ・アプリケーションによって管理されるオブジェクトのスケーラビリティを高めるためにオブジェクト状態管理を使用する方法を、Production サンプル・アプリケーションの `Registrar` および `Teller` オブジェクトを例にして説明します。

次のテーブルは、使用する BEA Tuxedo アプリケーションのスケーラビリティを大きく高めるために、BEA Tuxedo システムでサポートされているオブジェクト状態管理モデルを使用する方法をまとめたものです。

状態モデル	スケーラビリティを達成するための使用法
メソッド・バウンド	メソッド・バウンド・オブジェクトは、クライアントによって呼び出されている間だけ、マシンのメモリへ格納されます。呼び出しが完了したら、オブジェクトは非活性化されて、オブジェクトの状態データはすべてメモリからフラッシュされます。 メソッド・バウンド・オブジェクトは、使用するアプリケーションに状態を持たないサーバ・モデルを作成する場合に使用できます。こうすれば、アプリケーションで数千のオブジェクトを管理できます。クライアント・アプリケーションのビューからは、すべてのオブジェクトがサービス要求に利用可能です。しかし、サーバ・アプリケーションがオブジェクトをメモリにマッピングするのはクライアントによってオブジェクトが呼び出されている間だけなので、任意の時点でメモリ内にあるオブジェクトのうち、サーバ・アプリケーションによって管理されているものは比較的少数にとどまります。 メソッド・バウンド・オブジェクトは、このマニュアルでは状態を持たないオブジェクトともいいます。
プロセス・バウンド	プロセス・バウンド・オブジェクトは、最初に呼び出された時点から、自身が実行されているサーバ・プロセスがシャットダウンされるまでメモリ内にとどまります。使用するアプリケーションで支障がなければ、大量の状態データを伴うプロセス・バウンド・オブジェクトは、複数のクライアント呼び出しに対応するためにメモリ内にとどまることができます。このようにすることで、システムのリソースは、クライアント呼び出しごとにオブジェクトの状態データを読み書きすることから解放されます。 プロセス・バウンド・オブジェクトは、このマニュアルでは状態を持つオブジェクトともいいます。トランザクション・バウンド・オブジェクトも状態を持つと見なされる点に注意してください。これは、トランザクションのスコープ内であれば、自身への呼び出しの間はメモリ内にとどまることが可能なためです。

スケーラビリティによる性能向上を達成するには、Production サーバ・アプリケーションの Registrar および Teller オブジェクトを、method 活性化方針を持つようにコンフィギュレーションする必要があります。これら 2 つのオブジェクトに method 活性化方針を割り当てた結果、振る舞いに次の変化がみられます。

- これらのオブジェクトが呼び出されるときは常に、適切なサーバ・グループで BEA Tuxedo ドメインによってインスタンス化されます。
- 呼び出しが完了した後で、BEA Tuxedo ドメインによってこれらのオブジェクトは非活性化されます。

Basic サンプル・アプリケーションから Wrapper サンプル・アプリケーションを通じて、Registrar オブジェクトはプロセス・バウンドです。このオブジェクトへのすべてのクライアント要求は、マシンのメモリにある同じオブジェクトのインスタンスへ送られます。Basic サンプル・アプリケーションの設計は、すべての小規模なデプロイメントに適しています。しかし、クライアント・アプリケーションの要求が増すにつれて、Registrar オブジェクト上のクライアント要求はキューに入れられるようになり、このため応答時間が遅くなります。

しかし、Registrar および Teller オブジェクトが状態を持たず、これらのオブジェクトを管理するサーバ・プロセスが複製されていれば、これらのオブジェクトはクライアント要求の並行処理に利用可能になります。これらのオブジェクトで同時に処理できるクライアント要求の数に対する唯一の制限は、これらのオブジェクトをインスタンス化できる利用可能なサーバ・プロセスの数です。そのため、こうした状態を持たないオブジェクトを使用すれば、マシンのリソースを効率的に利用でき、クライアントへの応答時間も短縮されます。

最も重要なのは、BEA Tuxedo システムが Registrar および Teller オブジェクトのコピーを、両オブジェクトごとに複製されたサーバ・プロセス内でインスタンス化できるようにするためには、これらのオブジェクトのコピーが一意である必要があるということです。こうしたオブジェクトの各インスタンスを一意にするには、これらのオブジェクトに関係するファクトリで一意なオブジェクト ID を割り当てる必要があります。このことも含めて、これら 2 つのオブジェクトに関する設計上の検討事項については、「Registrar および Teller オブジェクトに関する設計上の追加考慮事項」を参照してください。

ファクトリ・ベース・ルーティング

ファクトリ・ベース・ルーティングは、特定のサーバ・グループへクライアント要求を送信する方法を提供する強力な機能です。ファクトリ・ベース・ルーティングを使用すれば、アプリケーションの処理負荷を複数のマシンに分散できます。これは、指定されたオブジェクトがインスタンス化されるグループ、すなわちマシンを決定できるからです。

ファクトリ・ベース・ルーティングを使用すれば、BEA Tuxedo システムのさまざまなロード・バランシング機能およびスケーラビリティ機能を拡張することができます。Production サンプル・アプリケーションの場合、ファクトリ・ベース・ルーティングを使用すれば、学生のサブセットの 1 つを登録する要求を 1 台のマシンへ送信し、別のサブセットに関する要求は別のマシンへ送信することができます。使用するアプリケーションの処理能力を増やすためにマシンを追加しても、BEA Tuxedo システムではアプリケーションのファクトリ・ベース・ルーティングを簡単に変更してマシンの追加に対応できます。

ファクトリ・ベース・ルーティングの第一の利点は、デプロイメント環境の拡大に対応して、アプリケーション、特にインターフェイスの呼び出し機能をスケール・アップするための簡単な方法が提供されることです。アプリケーションのデプロイメント範囲を追加マシンにも広げることは、厳密には管理用の機能であり、アプリケーションの再コーディングや再ビルドは不要です。

クライアント / サーバ・アプリケーションにファクトリ・ベース・ルーティングをインプリメントする際に設計上最も重要な検討事項は、ルーティングのベースとなる値の選択です。以下の節では、ファクトリ・ベース・ルーティングの機能について、Production サンプル・アプリケーションを使用して説明します。Production サンプル・アプリケーションは、ファクトリ・ベース・ルーティングを次のように使用します。

- `Registrar` オブジェクトへのクライアント・アプリケーション要求は、学生 ID に基づいてルーティングされます。つまり、学生のサブセットの 1 つに関する要求が 1 つのグループに送られ、別のサブセットに関する要求は別のグループに送られます。
- `Teller` オブジェクトへの要求は、口座番号に基づいてルーティングされます。つまり、口座のサブセットの 1 つに関する要求が 1 つのグループに送られ、別のサブセットに関する要求は別のグループに送られます。

ファクトリ・ベース・ルーティングのしくみ

ファクトリでファクトリ・ベース・ルーティングをインプリメントする際には、ファクトリがオブジェクト・リファレンスを作成する方法が変更されます。すべてのオブジェクト・リファレンスはグループ ID を含み、デフォルトのグループ ID はオブジェクト・リファレンスを作成するファクトリと同じになります。しかし、ファクトリ・ベース・ルーティングを使用する場合、ファクトリはグループ ID を決定するルーティング基準も含めてオブジェクト・リファレンスを作成します。その後、クライアント・アプリケーションがこうしたオブジェクト・リファレンスを使用して呼び出しを送信すると、BEA Tuxedo システムによって、要求はオブジェクト・リファレンスに指定されたグループ ID へルーティングされます。ここでは、オブジェクト・リファレンスに関するグループ ID がどのようにして生成されるかを説明します。

ファクトリ・ベース・ルーティングをインプリメントするには、次の項目を調整する必要があります。

- UBBCONFIG ファイルの INTERFACES および ROUTING セクションのデータ。
- UBBCONFIG ファイルでコンフィギュレーションされているグループ、マシンおよびデータベース。
- ファクトリでルーティング基準が指定される方法。ファクトリのためのインターフェイス定義では、グループ ID の決定に使用されるルーティング基準を表すパラメータを指定する必要があります。

調整が必要なデータについて説明するために、以降の 2 つの節では、UBBCONFIG ファイルでのファクトリ・ベース・ルーティングのコンフィギュレーションと、ファクトリでのファクトリ・ベース・ルーティングのインプリメンテーションを説明します。

UBBCONFIG ファイルでのファクトリ・ベース・ルーティングのコンフィギュレーション

要求がルーティングされる各インターフェイスについて、UBBCONFIG ファイルに次の情報を記述する必要があります。

- ルーティング基準のデータの詳細。
- 各種の基準ごとに、特定のサーバ・グループにルーティングするための値。

ファクトリ・ベース・ルーティングをコンフィギュレーションするには、UBBCONFIG ファイルの INTERFACES および ROUTING セクションで次のデータを指定するほか、グループおよびマシンを識別する方法を指定する必要があります。

1. INTERFACES セクションには、ファクトリ・ベース・ルーティングを使用可能にする必要があるインターフェイスの名前をリストします。各インターフェイスについて、このセクションでインターフェースのルーティングの基準の種類を指定します。このセクションでは、次の例のように、FACTORYROUTING という識別子でルーティング基準を指定します。

   ```
   INTERFACES
       "IDL:beasys.com/UniversityP/Registrar:1.0"
           FACTORYROUTING = STU_ID
       "IDL:beasys.com/BillingP/Teller:1.0"
           FACTORYROUTING = ACT_NUM
   ```

   上の例では、Production サンプルでファクトリ・ベース・ルーティングが使用される 2 つのインターフェイスの完全修飾されたインターフェイス名を示します。FACTORYROUTING 識別子は、ルーティング値の名前として STU_ID および ACT_NUM をそれぞれ指定します。

2. ROUTING セクションでは、ルーティング値ごとに以下のデータを指定します。

   - TYPE パラメータ。ルーティングの種類を指定します。Production サンプルでは、ルーティングの種類はファクトリ・ベース・ルーティングです。このため、このパラメータは FACTORY と定義されます。
   - FIELD パラメータ。ファクトリがルーティング値に挿入する名前を指定します。Production サンプルでは、フィールドのパラメータはそれぞれ student_id および account_number です。
   - FIELDTYPE パラメータ。ルーティング値のデータ型を指定します。Production サンプルでは、student_id および account_number のフィールド・タイプは long です。
   - RANGES パラメータ。各グループにルーティングされる値を指定します。

   次の例は、Production サンプル・アプリケーションで使用される UBBCONFIG ファイルの ROUTING セクションを示します。

```
ROUTING
    STU_ID
        FIELD     = "student_id"
        TYPE      = FACTORY
```

```
    FIELDTYPE = LONG
    RANGES    = "100001-100005:ORA_GRP1,100006-100010:ORA_GRP2"
ACT_NUM
    FIELD     = "account_number"
    TYPE      = FACTORY
    FIELDTYPE = LONG
    RANGES    = "200010-200014:APP_GRP1,200015-200019:APP_GRP2"
```

上の例では、一方の範囲に入る ID を持った学生についての Registrar オブジェクト・リファレンスは一方のサーバ・グループにルーティングされ、もう一方の範囲に入る ID を持った学生についての Registrar オブジェクト・リファレンスは他方のグループにルーティングされるということが示されています。同様に、一方の範囲に入る口座についての Teller オブジェクト・リファレンスは一方のサーバ・グループへルーティングされ、もう一方の範囲に入る口座についての Teller オブジェクト・リファレンスは他方のグループにルーティングされています。

3. UBBCONFIG ファイルの ROUTING セクションにある RANGES 識別子で指定されているグループを識別およびコンフィギュレーションする必要があります。たとえば、Production サンプルでは 4 つのグループが指定されています。APP_GRP1、APP_GRP2、ORA_GRP1、および ORA_GRP2 です。これらのグループをコンフィギュレーションして、グループが実行されるマシンを識別する必要があります。

   次の例は、Production サンプルの UBBCONFIG ファイルの GROUPS セクションを示します。このセクションで、ORA_GRP1 および ORA_GRP2 グループがコンフィギュレーションされています。GROUPS セクションに列挙されている名前が、ROUTING セクションに指定されているグループ名とどのように一致するかに注目してください。この点が、ファクトリ・ベース・ルーティングの正常な動作に不可欠です。さらに、アプリケーションでグループをコンフィギュレーションする方法に関する何らかの変更も、ROUTING セクションに反映される必要があります。BEA Tuxedo ソフトウェアとともにパッケージされている Production サンプルは 1 台のマシンでだけ実行できるようにコンフィギュレーションされている点に注意してください。ただし、このアプリケーションを複数のマシンで実行できるようにコンフィギュレーションすることは簡単です。

```
*GROUPS
    APP_GRP1
        LMID      = SITE1
        GRPNO     = 2
        TMSNAME   = TMS
    APP_GRP2
        LMID      = SITE1
        GRPNO     = 3
        TMSNAME   = TMS
```

```
ORA_GRP1
    LMID      = SITE1
    GRPNO     = 4
    OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
    CLOSEINFO = ""
    TMSNAME   = "TMS_ORA"
ORA_GRP2
    LMID      = SITE1
    GRPNO     = 5
    OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
    CLOSEINFO = ""
    TMSNAME   = "TMS_ORA"
```

ファクトリでのファクトリ・ベース・ルーティングのインプリメンテーション

ファクトリは、TP::create_object_reference() オペレーションへの呼び出しがインプリメントされている方法によってファクトリ・ベース・ルーティングをインプリメントします。このオペレーションには、次の C++ バインディングがあります。

```
CORBA::Object_ptr  TP::create_object_reference (
                       const char* interfaceName,
                       const PortableServer::oid &stroid,
                       CORBA::NVlist_ptr criteria);
```

このオペレーションの 3 番目のパラメータ criteria は、ファクトリ・ベース・ルーティングで使用される名前付き値のリストを指定します。このため、ファクトリでファクトリ・ベース・ルーティングをインプリメントする機能は、NVlist をビルドすることにあります。

先に説明したように、Production サンプル・アプリケーションの RegistrarFactory オブジェクトは STU_ID という値を指定します。この値は、UBBCONFIG ファイルの次の項目と正確に一致する必要があります。

- INTERFACES セクションの FACTORYROUTING 識別子によって指定されるルーティング名、型、および使用できる値。
- ROUTING セクションに指定されるルーティング基準名、フィールド、およびフィールド型。

RegistrarFactory オブジェクトは、次のコードを使用して NVlist に学生 ID を挿入します。

```
// 学生 ID ( ルーティング基準になる )
// を CORBA NVList に挿入する
CORBA::NVList_var v_criteria;
```

```
TP::orb()->create_list(1, v_criteria.out());
CORBA::Any any;
any <<= (CORBA::Long)student;
v_criteria->add_value("student_id", any, 0);
```

RegistrarFactory オブジェクトは次のようにして TP::create_object_reference() オペレーションを呼び出し、前のコード例で作成された NVlist を渡します。

```
// registrar オブジェクト・リファレンスを、ルーティング基準を
// 使用して作成する
CORBA::Object_var v_reg_oref =
    TP::create_object_reference(
        UniversityP::_tc_Registrar->id(),
        object_id,
        v_criteria.in()
    );
```

また、Production サンプル・アプリケーションは、TellerFactory オブジェクトでのファクトリ・ベース・ルーティングも使用して、Teller オブジェクトが口座番号に基づいてどのグループでインスタンス化されるべきであるかを判別します。

注記 指定されたインターフェイスおよび OID を持つ 1 つのオブジェクトが 2 つの異なるグループで同時に活性化されることも、それら 2 つのグループに同じオブジェクト・インプリメンテーションがある場合には起こりえます。しかし、使用するファクトリで一意な OID が生成されていれば、こうした状況はほぼありません。指定されたインターフェイス名および OID の 1 つのオブジェクトがドメイン内で一度に 1 つだけ利用可能であることを保証する必要がある場合は、次のいずれかを行います。ファクトリ・ベース・ルーティングを使用して、特定の OID のオブジェクトが常に同じグループにルーティングされるようにするか、指定されるオブジェクト・インプリメンテーションが 1 つのグループのみに存在するようにドメインをコンフィグレーションする。そうすれば、指定されたインターフェイス名および OID を持つ 1 つのオブジェクトに複数のクライアントがオブジェクト・リファレンスをした場合でも、そのリファレンスは常に同じオブジェクト・インスタンスへルーティングされることになります。

オブジェクトの OID によるルーティングを利用可能にするには、TP::create_object_reference() オペレーションで OID をルーティング基準に指定してから、UBBCONFIG ファイルをそれに合わせて設定します。

実行時の処理

ファクトリでファクトリ・ベース・ルーティングをインプリメントするとき、BEA Tuxedo システムによってオブジェクト・リファレンスが生成されます。次の例は、ファクトリ・ベース・ルーティングがインプリメントされているときにクライアント・アプリケーションが Registrar オブジェクトへのオブジェクト・リファレンスを取得する方法を示します。

1. クライアント・アプリケーションが RegistrarFactory オブジェクトを呼び出して、Registrar オブジェクトへのリファレンスを要求します。要求には学生 ID が含まれます。
2. RegistrarFactory は、学生 ID を NVlist に挿入します。NVlist はルーティング基準として使用されます。
3. RegistrarFarctory は TP::create_object_reference() オペレーションを呼び出して、Registrar インターフェイス名、一意な OID、および NVlist を渡します。
4. BEA Tuxedo システムはルーティング・テーブルの内容を NVlist の値と比較して、グループ ID を判別します。
5. BEA Tuxedo システムは、オブジェクト・リファレンスにグループ ID を挿入します。

続いてクライアント・アプリケーションがオブジェクト・リファレンスを使用してオブジェクト呼び出しを実行した場合、BEA Tuxedo システムはその要求を、オブジェクト・リファレンスで指定されたグループへルーティングします。

注記 Process-Entity デザイン・パターンを使用する場合は、ファクトリ・ベース・ルーティングのインプリメンテーションに注意してください。オブジェクトは、グループのデータベースに含まれているエンティティだけを提供できます。

Registrar および Teller オブジェクトに関する設計上の追加考慮事項

Registrar および Teller オブジェクトの設計に影響する主な考慮事項には、以下のものがあります。

- Registrar および Teller オブジェクトが Production のデプロイメント環境、つまり、複数の複製されたサーバ・プロセスおよび複数のグループで適切に機能するようにする方法。University および Billing サーバ・プロセスが複製されている場合は、設計時にこれら 2 つのオブジェクトをインスタンス化する方法を考慮する必要があります。
- Production の BEA Tuxedo ドメインにある 2 つのサーバ・グループがそれぞれ別のデータベースを扱う場合に、特定の学生についての登録用のオペレーションおよび課金用のオペレーションに関するクライアント要求が適切なサーバ・グループへ送られるようにする方法。

これらの考慮事項の主要な意味は、これらのオブジェクトが次の項目を満たす必要があるということです。

- 一意なオブジェクト ID (OID) を持っている。
- メソッド・バウンドである。つまり、これらのオブジェクトには method 活性化方針が割り当てられている。

以降の節では、これらの検討事項およびその内容について詳しく説明します。

Registrar および Teller オブジェクトのインスタンス化

Production サンプル・アプリケーションの前に扱った University サーバ・アプリケーションでは、Registrar および Teller オブジェクトの実行時の振る舞いは簡単なものでした。

- 各オブジェクトはプロセス・バウンドであり、最初に呼び出された時点で活性化され、自身が実行されているサーバ・プロセスがシャットダウンされるまでメモリ内にとどまっていました。
- BEA Tuxedo ドメインを実行しているサーバ・グループは 1 つだけであり、グループにある University および Billing サーバ・プロセスも 1 つだけなので、すべてのクライアント要求は同じオブジェクトに割り当てられました。複数のクライアント要求が BEA Tuxedo ドメインに着信した場合でも、これ

らのオブジェクトは一度に 1 つのクライアント要求としてそれぞれ処理されました。

- サーバ・プロセス内には各オブジェクトのインスタンスが 1 つだけ存在するので、どのオブジェクトにも一意な OID は不要でした。各オブジェクトの OID は、インターフェイス・リポジトリ ID のみを指定しました。

しかし、University および Billing サーバ・プロセスが複製されると、BEA Tuxedo ドメインには `Registrar` および `Teller` オブジェクトの複数のインスタンスを区別する方法が必要になります。つまり、1 つのグループで 2 つの University サーバ・プロセスが実行中である場合、BEA Tuxedo ドメインには、いわば、1 番目の University サーバ・プロセスで実行中の `Registrar` オブジェクトと、2 番目の University サーバ・プロセスで実行中の `Registrar` オブジェクトを区別するための方法が必要になります。

これらのオブジェクトの複数のインスタンスを区別する機能を BEA Tuxedo ドメインに与える方法は、オブジェクトの各インスタンスを一意にすることです。

各 `Registrar` および `Teller` オブジェクトを一意にするには、これらのオブジェクトへのオブジェクト・リファレンスをファクトリで作成する際の方法を変更する必要があります。たとえば、Basic サンプル・アプリケーションの `RegistrarFactory` オブジェクトが `Registrar` オブジェクトへのオブジェクト・リファレンスを作成するとき、`TP::create_object_reference()` オペレーションが指定する OID には `registrar` という文字列が含まれているだけでした。しかし、Production サンプル・アプリケーションの場合、同じ `TP::create_object_reference()` オペレーションでは生成済みの一意な OID が使用されます。

`Registrar` および `Teller` オブジェクトのそれぞれに一意な OID を付与した結果、これらのオブジェクトの複数のインスタンスが、BEA Tuxedo ドメインの中で同時に実行できるようになりました。この特性は状態を持たないオブジェクト・モデルに典型的なもので、BEA Tuxedo ドメインのスケーラビリティを高めつつ、高い性能を提供する方法の例でもあります。

そして最後に、一意な `Registrar` および `Teller` オブジェクトはクライアント要求ごとにメモリに格納される必要があるため、これらのオブジェクトへの呼び出しが完了したときには、オブジェクトの状態がメモリ上でアイドル状態となって残らないように、オブジェクトを非活性化することが非常に重要になります。Production サーバ・アプリケーションでは、ICF ファイルでこれら 2 つのオブジェクトに `method` 活性化方針を割り当てることで、この問題を解決しています。

適切なサーバ・グループで学生の登録が発生するようにする方法

複製されたサーバ・グループを使用することでスケーラビリティについて得られる最大の利点は、複数のマシンに処理を分散できることです。しかし、University サンプル・アプリケーションのように、使用するアプリケーションがデータベースとやり取りを行う場合は、こうした複数のサーバ・グループがデータベースとのやり取りに与える悪影響を考慮することが不可欠です。

多くの場合、デプロイメント環境のマシン 1 台につき 1 つのデータベースが関連付けられています。使用するサーバ・アプリケーションが複数のマシンに分散されている場合は、データベースを設定する方法について検討する必要があります。

Production サンプル・アプリケーションでは、この章で説明したように、2 つのデータベースが使用されます。しかし、このアプリケーションでさらに多くのデータベースを使用するように、コンフィギュレーションを簡単に変更することができます。使用するデータベースの数は、システム管理者が決定してください。

Production サンプル・アプリケーションでは、学生情報と口座情報は 2 つのデータベースに分割されていますが、コース情報は同一です。2 つのデータベースで同一のコース情報を持つことは、問題になりません。なぜなら、コース登録の目的上、コース情報は読み取り専用であるからです。一方で、学生情報および口座情報については読み書きが行われます。複数のデータベースが学生および口座についても同一のデータを格納するとしたら (つまり、データベースが分割されていない場合)、アプリケーションでは、学生情報または口座情報が変更されるたびにデータベース全体にわたって学生および口座の情報の更新を同期する処理のオーバーヘッドに対処する必要が生じるでしょう。

Production サンプル・アプリケーションはファクトリ・ベース・ルーティングを使用して要求のセットの 1 つを 1 台のマシンに送信し、別のセットを別のマシンに送信します。先に説明したように、ファクトリ・ベース・ルーティングは、`Registrar` オブジェクトへのリファレンスが作成されている方法によって `RegistrarFactory` オブジェクトにインプリメントされています。

たとえば、`Registrar` オブジェクトへのリファレンスを取得するためにクライアント・アプリケーションが `RegistrarFactory` オブジェクトに要求を送信するとき、クライアント・アプリケーションは要求に学生 ID を含めます。以後、クライアント・アプリケーションは、特定の学生に関する `Registrar` オブジェクトの呼び出しを実行する際には、`RegistrarFactory` オブジェクトが返すオブ

ジェクト・リファレンスを使用する必要があります。これは、ファクトリによって返されるオブジェクト・リファレンスがグループ固有のものであるからです。このため、たとえば、続けてクライアント・アプリケーションが Registrar オブジェクトの get_student_details() オペレーションを呼び出したときに、学生のデータがあるデータベースと関連付けられているサーバ・グループでは Registrar オブジェクトが活性化されているということが、クライアント・アプリケーションに対して保証されるといったことも可能です。この機能を示すために、Production サンプル・アプリケーションにインプリメントされている次の実行シナリオを検討してみます。

1. クライアント・アプリケーションは、RegistrarFactory オブジェクトの find_registrar() オペレーションを呼び出します。この要求には学生 ID 1000003 が含まれます。
2. BEA Tuxedo ドメインは、クライアント要求を任意の RegistrarFactory オブジェクトにルーティングします。
3. RegistrarFactory オブジェクトは、UBBCONFIG ファイルのルーティング情報に基づいて、学生 ID を使用して ORA_GRP1 の Registrar オブジェクトへのオブジェクト・リファレンスを作成し、このオブジェクト・リファレンスをクライアント・アプリケーションに返します。
4. クライアント・アプリケーションは、Registrar オブジェクトの register_for_courses() オペレーションを呼び出します。
5. BEA Tuxedo ドメインは、クライアント要求を受信してから、それをオブジェクト・リファレンスで指定されているサーバ・グループへルーティングします。この場合、クライアント要求は Production マシン 1 にある ORA_GRP1 の University サーバ・プロセスに送信されます。
6. University サーバ・プロセスは、Registrar オブジェクトをインスタンス化してから、同オブジェクトにクライアント呼び出しを送信します。

以上のシナリオの RegistrarFactory オブジェクトがクライアント・アプリケーションに返すのは、ORA_GRP1 でだけインスタンス化できる Registrar オブジェクトへの一意なリファレンスです。ORA_GRP1 は Production マシン 1 で実行されていて、学生 ID が 100001 ～ 100005 の範囲の学生に関するデータが格納されているデータベースを利用します。このため、続けてクライアント・アプリケーションが学生に関する Registrar オブジェクトへの要求を送信したとき、Registrar オブジェクトは適切なデータベースとやり取りができます。

Teller オブジェクトが適切なサーバ・グループでインスタンス化されるようにする方法

Registrar オブジェクトで Teller オブジェクトが必要になると、Registrar オブジェクトは TellerFactory オブジェクトを呼び出します。このとき、University Server オブジェクトでキャッシュされている TellerFactory オブジェクト・リファレンスが使用されます。詳細については、「Teller オブジェクトへの要求の送信」を参照してください。

しかし、ファクトリ・ベース・ルーティングが TellerFactory オブジェクトで使用されているので、Registrar オブジェクトが学生の口座番号を渡すのは、Registrar オブジェクトが Teller オブジェクトへのリファレンスを要求するときになります。このようにして、TellerFactory オブジェクトは、正しいデータベースがあるグループに Teller オブジェクトへのリファレンスを作成します。

注記 Production サンプル・アプリケーションが正しく機能するには、システム管理者がサーバ・グループおよびデータベースのコンフィギュレーションを正しく行うことが必須です。特に、システム管理者は、ルーティング・テーブルで指定されているルーティング基準と、このルーティング基準を使用する要求のルーティング先となるデータベースとの間に、必ず一致が存在するようにしなければなりません。Production サンプルを例にすると、指定されたグループにあるデータベースは、そのグループにルーティングされた要求に正しく対応する学生情報および口座情報を含んでいる必要があります。

Production サーバ・アプリケーションをさらにスケーリングする方法

将来的には、Production サンプル・アプリケーションのシステム管理者が BEA Tuxedo ドメインの容量を増やす必要を感じる場合もあります。たとえば、University で学生数が大幅に増加した場合や、7 つのキャンパスがある大学の全体のコース登録処理を扱えるように Production アプリケーションをスケール・アップする場合などが考えられます。こうしたスケーリングを、アプリケーションの変更や再ビルドをせずに実現できます。

継続的に容量を追加するために、システム管理者には以下のツールが用意されています。

- Production サンプル・アプリケーションを複数のマシンにある複数のサーバ・グループに複製する場合。

  UBBCONFIG ファイルを変更する必要があります。サーバ・プロセスが実行されるグループを追加し、グループでどのサーバ・プロセスを実行するか指定してから、どのマシンで実行するかを指定してください。

- ファクトリ・ベース・ルーティングのテーブルを変更する場合。

  たとえば、この章で示したように 2 つのグループへルーティングする代わりに、システム管理者が UBBCONFIG ファイルでルーティング・ルールを変更することで、BEA Tuxedo ドメインに追加された新しいグループの間でアプリケーションの処理をさらに分離することが可能です。ルーティング・テーブルへのすべての変更は、UBBCONFIG ファイルでコンフィギュレーションされているサーバ・グループおよびマシンに対するすべての変更および追加と整合している必要があります。

注記 データベースを使用するアプリケーションの容量を増やす場合、特にファクトリ・ベース・ルーティングを使用しているときには、データベースの設定に与える影響を検討する必要があります。たとえば、Production サンプル・アプリケーションが 6 台のマシンに分散している場合、各マシン上のデータベースは、適切に設定されて、UBBCONFIG ファイルのルーティング・テーブルに合致している必要があります。

状態を持たないオブジェクトまたは状態を持つオブジェクトの選択

一般に、状態を持たないオブジェクトをインプリメントする際のコストと、状態を持つオブジェクトをインプリメントする際のコストを比較する必要があります。

オブジェクトをその永続状態で初期化するコストが高い場合 (これは、たとえば、そのオブジェクトのデータが大きな領域を占めている場合や、永続状態の場所が、それを活性化するサーバントから大幅に離れたディスク上である場合ですが)、たとえオブジェクトが会話の間アイドル状態であるとしても、オブジェクトが状態を持たないようにしておく方が合理的です。オブジェクトを活性化しておくコストがマシンのリソース使用の面で高くつく場合は、そうしたオブジェクトが状態を持たないようにするのが合理的です。

オブジェクト状態の管理をアプリケーションにとって効率的かつ適切な方法で行うことで、多数のオブジェクトを利用するクライアント・アプリケーションを同時に多数サポートするアプリケーションの機能を最高にすることができます。一般には、こうした目的のために、オブジェクトに対して `method` 活性化方針を割り当てます。こうすることによって、アイドル状態のオブジェクト・インスタンスが非活性化され、マシンのリソースをほかのオブジェクト・インスタンスに割り当てられるようになります。ただし、使用するアプリケーション固有の特性および要件は、アプリケーションごとに異なります。

注記 BEA Tuxedo リリース 8.0 では、性能の拡張として、パラレル・オブジェクトのサポートが提供されています。この機能を利用すると、特定アプリケーションのすべてのビジネス・オブジェクトを状態を持たないオブジェクトにすることができます。詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「TP フレームワーク」を参照してください。

状態を持たないオブジェクトが必要な場合

一般に状態を持たないオブジェクトは良好な性能とサーバ・リソースの最適な使用を実現しますが、これはオブジェクトがアイドル状態のときにサーバ・リソースが決して使用されないためです。状態を持たないオブジェクトは、一般に、サーバ・アプリケーションのインプリメンテーションに適した手法です。状態を持たないオブジェクトは、特に以下の状況に適しています。

- クライアント・アプリケーションは、オブジェクトへの呼び出しの間のユーザ入力を通常は待機する。
- クライアント要求はサーバ・アプリケーションが必要とするデータを通常はすべて含み、サーバはそのデータのみを使用してクライアント要求を処理できる。
- オブジェクトのアクセス・レートはたいへん高いが、特定のクライアント・アプリケーションからのアクセス・レートが低い場合。

オブジェクトを状態を持たないようにすることで、サーバ・アプリケーションのリソースがクライアント・アプリケーションからの入力の待機のために不定期に長時間拘束されないようにできます。

状態を持たないオブジェクト・モデルを使用するアプリケーションでは、次の特性に注意してください。

- 呼び出しについての情報および関連する情報は、サーバ・アプリケーションがクライアント要求の実行を終了した後は保守されません。
- 着信するクライアント要求は、最初に利用可能なサーバ・プロセスに送信されます。要求が終了した後は、アプリケーションの状態が消滅して、サーバ・アプリケーションは別のクライアント・アプリケーション要求で利用可能になります。
- オブジェクトの永続状態の情報は、サーバ・プロセスの外部に存在します。このオブジェクト上での呼び出しごとに、永続状態がメモリに読み込まれます。
- BEA Tuxedo ドメインは、クライアント・アプリケーションから 1 つのオブジェクトへの連続した要求を別のサーバ・プロセスへ割り当てることがあります。
- 状態を持たないオブジェクトを実行しているマシンの全体的なシステム・性能は、通常は改善されます。

状態を持つオブジェクトが必要な場合

状態を持つオブジェクトは、いったん活性化されると、特定のイベントが発生するまでメモリ内にとどまります。当該のイベントは、たとえば、オブジェクトが存在しているプロセスのシャットダウン、またはオブジェクトが活性化されているトランザクションの完了などです。

状態を持つオブジェクトは、一般に以下の状況に適しています。

- 1 つのオブジェクトが、多数のクライアント・アプリケーションによってたいへん頻繁に利用される場合。これには、ファクトリのように寿命が長く、よく知られたオブジェクトが該当します。サーバ・アプリケーションがこうしたオブジェクトを活性化させておくと、クライアント・アプリケーションアプリケーションは最短の応答時間でオブジェクトにアクセスできます。活性化されたオブジェクトは多くのクライアント・アプリケーションに共有されるので、この型のオブジェクトでメモリにあるものは比較的少数です。

  注記　プロセス・オブジェクトをトランザクションにどのように関与させるかについては、注意深く検討してください。トランザクションに関与しているすべてのオブジェクトは、ほかのクライアント・アプリケーションまたはオブジェクトから呼び出せません。プロセス・オブジェクトは多数のクライアント・アプリケーションによって利用されるので、トランザクションに対して頻繁に、または長期間関与すると、問題の原因になる可能性があります。

- トランザクションを完了するために 1 つのオブジェクト上でクライアント・アプリケーションがオペレーションを連続して呼び出す必要がある場合、および、こうした呼び出しの間にユーザからの入力を待機する際にクライアント・アプリケーションがアイドル状態でない場合。この場合、呼び出しの間にオブジェクトが非活性化されると、各呼び出しの間に状態の読み書きが発生するために、応答時間の遅延が発生します。こうした振る舞いは、トランザクションには適していません。応答時間を改善するには、サーバ・リソースの保持を犠牲にするという方法が可能です。

状態を持つオブジェクトの以下の振る舞いに注意してください。

- 状態の情報はサーバの呼び出しの間に保持されます。通常は、指定された時間にわたって、指定されたクライアント・アプリケーションに専用のサーバントが存在します。

- データはクライアント・アプリケーションとサーバ・アプリケーションの間で送信および受信されますが、サーバ・プロセスはメモリ内の付加的なコンテキストやアプリケーション状態の情報を保持します。

- 1 つまたは複数の状態を持つオブジェクトが大量のマシン・リソースを使用している場合、状態を持つオブジェクトと関連付けられていないタスクおよびプロセスを扱うときのサーバの性能は、状態を持たないサーバ・モデルの場合よりも悪化します。

  たとえば、1 つのオブジェクトがデータベースにロックを保持し、大量のデータをメモリにキャッシュしている場合、その状態を持つオブジェクトによって使用されているデータベースおよびメモリは、トランザクションが完了するまでほかのオブジェクトでは利用できません。

索引

D

E

F

I

K

L

M

N

O

P

R

S

T

U

W

X

あ

い

え

お

か

き

く

こ

さ

し

す

せ

ち

て

と

ぬ

ひ

ふ

へ

ま