CICS Transaction Server for z/OS
バージョン 4 リリース 2

CICS での Java アプリケーション

SA88-4321-01
(英文原典：SC34-7174-01)

CICS Transaction Server for z/OS
バージョン 4 リリース 2

CICS での Java アプリケーション

SA88-4321-01
(英文原典：SC34-7174-01)

お願い

本書および本書で紹介する製品をご使用になる前に、 461 ページの『特記事項』に記載されている情報をお読みください。

本書は、CICS Transaction Server for z/OS バージョン 4 リリース 2 (製品番号 5655-S97)、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： SC34-7174-01
CICS Transaction Server for z/OS
Version 4 Release 2
Java Applications in CICS

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第1刷 2011.9

目次

前書き

本書には、プログラムを作成するユーザーが バージョン 4 リリース 2 のサービスを使用するためのプログラミング・インターフェースが記述されています。

本書について

本書は、CICS® における Java アプリケーションとエンタープライズ Bean の作成と使用の方法について説明します。

本書の対象読者

本書は次の読者を対象にしています。

- CICS の経験がほとんどなく、Java プログラムの作成と実行に必要なだけの CICS の知識が必要な、経験のある Java アプリケーション・プログラマー。
- Java サポートに対する CICS 要件に関する知識が必要な、経験のある CICS ユーザーおよびシステム・プログラマー。

CICS Transaction Server for z/OS, バージョン 4 リリース 2 の変更点

このリリースに加えられた変更点に関する情報は、インフォメーション・センターの「*リリース・ガイド*」または以下の資料を参照してください。

- *CICS Transaction Server for z/OS リリース・ガイド*
- *CICS Transaction Server for z/OS V4.1 からのアップグレード*
- *CICS Transaction Server for z/OS V3.2 からのアップグレード*
- *CICS Transaction Server for z/OS V3.1 からのアップグレード*

リリース後に本文を技術的に変更した箇所は、その箇所の左側に縦線 (|) 引いて示しています。

第 1 章 CICS における Java サポート

CICS は、CICS 領域の制御下にある Java 仮想マシン (JVM) で Java エンタープライズ・アプリケーションを開発し、実行するためのツールおよびランタイム環境を提供します。Java アプリケーションは、CICS サービスおよび他の言語で作成されたアプリケーションと対話できます。

z/OS® 上の Java は、Java アプリケーションを実行するための包括的なサポートを提供します。CICS は、IBM® 64-bit SDK for z/OS, Java テクノロジー・エディション、バージョン 6.0.1 を使用します。 SDK には、Java API のフルセットおよび 1 組の開発ツールをサポートする Java ランタイム環境が含まれています。z/OS 上での Java の採用を促進するために、特定の System z® ハードウェアで特殊なプロセッサーが使用可能です。このプロセッサーは、IBM System z Application Assist Processor (zAAP) と呼ばれ、適格な Java ワークロードを実行するための追加プロセッサー容量を低コストで提供できます。CICS は、Java ワークロードでこの機能を活用できます。 http://www.ibm.com/servers/eserver/zseries/software/java/ で、z/OS プラットフォーム上の Java に関する詳細情報を見つけ、64 ビット・バージョンの SDK をダウンロードすることができます。

CICS は Eclipse ベースのツール、および Java アプリケーション用の 2 つのランタイム環境を提供します。

CICS Explorer™ SDK

CICS Explorer SDK は、Eclipse ベースの統合開発環境 (IDE) 用に自由にダウンロードできます。SDK は、OSGi Service Platform 仕様に従うアプリケーションの開発とデプロイをサポートします。OSGi Service Platform は、コンポーネント・モデルを使用してアプリケーションを開発し、それらのアプリケーションを OSGi バンドルとしてフレームワークにデプロイするためのメカニズムを提供します。*OSGi バンドル* は、アプリケーション・コンポーネントのデプロイメントの単位であり、バージョン管理情報、依存関係、およびアプリケーション・コードが入っています。OSGi の主な利点は、*OSGi サービス* と呼ばれる明確に定義されたインターフェースからのみアクセスされる再使用可能コンポーネントから、アプリケーションを作成できることです。また、Java アプリケーションのライフサイクルと依存関係をきめ細かく管理することもできます。

CICS Explorer SDK は、サポートされている任意のリリースの CICS 用に Java アプリケーションの開発をサポートします。 SDK には、CICS サービスにアクセスするためのクラス、および CICS 用のアプリケーション開発に取り掛かるためのサンプルから成る、Java CICS (JCICS) ライブラリーが組み込まれています。また、このツールを使用すると、既存の Java アプリケーションを OSGi に変換することもできます。

JVM サーバー

JVM サーバーは、CICS における Java アプリケーション用の戦略的なランタイム環境です。JVM サーバーは、単一の JVM でさまざまな Java アプリケーションからの複数の並行要求を処理できます。これにより、CICS 領

域で Java アプリケーションを実行するのに必要な JVM 数が減ります。JVM サーバーを使用するには、Java アプリケーションはスレッド・セーフであり、OSGi 仕様に従う必要があります。可能な場合はすべての Java アプリケーションにこのランタイム環境を使用してください。これは、CICS 領域で Java ワークロードを実行するための推奨方式であり、次の利点があります。

- 1 つの JVM サーバーで複数の Java アプリケーションを実行できるので、CICS 領域で JVM の実行と管理を行う操作が簡単になります。
- 適格な Java ワークロードを zAAP で実行し、トランザクションのコストを削減できます。
- 1 つの JVM サーバーで複数の異なるタイプの作業 (スレッド・セーフ Java プログラムや Web サービスを含む) を実行できます。
- JVM サーバーを再始動することなく、OSGi フレームワーク内のアプリケーションのライフサイクルを管理できます。
- CICS とその他のプラットフォームとの間で、OSGi を使用してパッケージされた Java アプリケーションをより容易に移植できます。

プールされた JVM

プールされた JVM は、各 Java プログラムが独自の JVM を使用するランタイム環境です。並行して実行される JVM プログラムは、相互に分離されます。Java プログラムが JVM を使用し終わると、JVM はそれ以降のプログラムで再利用できます。スレッド・セーフでない既存の Java アプリケーションにこのランタイム環境を使用してください。プールされた JVM は安定していますが、将来の CICS リリースで除去されます。可能な場合は、JVM サーバーで稼働するように既存の Java アプリケーションを変換してください。

OSGi Service Platform

OSGi Service Platform は、コンポーネント・モデルを使用してアプリケーションを開発し、それらのアプリケーションを OSGi フレームワークにデプロイするためのメカニズムを提供します。OSGi アーキテクチャーは、Java アプリケーションの作成と管理に有利な複数のレイヤーに分けられます。

OSGi フレームワークは、OSGi Service Platform 仕様の中核をなします。CICS は OSGi フレームワークの Equinox バージョン 3.6.1 実装環境を使用し、バージョン 4 の OSGi Service Platform 仕様をサポートします。 OSGi フレームワークは、JVM サーバーの開始時に初期化されます。Java アプリケーションに OSGi を使用すると、次の主な利点があります。

- Java アプリケーションの移植可能性が向上し、リエンジニアリングが容易になり、要件の変更への適応性が高まります。
- Plain Old Java Object (POJO) プログラミング・モデルに従って、動的なライフサイクルを持つ 1 組の OSGi バンドルとしてアプリケーションをデプロイすることを選択できます。
- アプリケーション・バンドルの依存関係とバージョンの管理が容易になります。

OSGi アーキテクチャーには、以下のレイヤーがあります。

- モジュール・レイヤー
- ライフサイクル・レイヤー
- サービス・レイヤー

モジュール・レイヤー

デプロイメントの単位は OSGi バンドルです。モジュール・レイヤーでは、OSGi フレームワークがバンドルのモジュラー・アスペクトを処理します。OSGi フレームワークがこの処理を実行できるようにするメタデータは、バンドルのマニフェスト・ファイルで提供されます。

OSGi の主な利点の 1 つは、クラス・ローダー・モデルで、これは、マニフェスト・ファイルのメタデータを使用します。OSGi にはグローバル・クラスパスはありません。バンドルが OSGi フレームワークにインストールされると、それらのメタデータはモジュール・レイヤーによって処理され、宣言された外部依存関係は、インストールされた他のモジュールによって宣言されたエクスポートおよびバージョン情報に照らして調整されます。OSGi フレームワークは、すべての依存関係を解明し、バンドルごとに独立した必須のクラスパスを計算します。この方法は、以下の要件が満たされることを確実にすることによって、プレーン Java クラス・ロードの欠点を解決します。

- 各バンドルが、明示的にエクスポートする Java パッケージのみを表示する。
- 各バンドルが、明示的にそのパッケージ依存関係を宣言する。
- パッケージを特定のバージョンでエクスポートし、特定のバージョンで、または特定の範囲のバージョンからインポートできる。
- パッケージの複数のバージョンが異なるクライアントに対して同時に使用可能である。

ライフサイクル・レイヤー

OSGi におけるバンドルのライフサイクル管理レイヤーは、JVM のライフサイクルとは無関係に、バンドルを動的にインストール、開始、停止、およびアンインストールすることを可能にします。ライフサイクル・レイヤーは、バンドルが、その依存関係がすべて解決される場合のみ開始することを確実にし、実行時の ClassNotFoundException 例外の発生を減らします。未解決の依存関係がある場合、OSGi フレームワークは問題を報告し、バンドルを開始しません。

各バンドルは、バンドル・マニフェストで識別されるバンドル・アクティベーター・クラスを提供できます。フレームワークはこのクラスにイベントの開始および停止を要求します。

サービス・レイヤー

OSGi のサービス・レイヤーは、本来、非永続サービス・レジストリー・コンポーネントを使用してサービス指向アーキテクチャーをサポートします。バンドルはサービス・レジストリーにサービスを公開し、他のバンドルがそれらのサービスをサービス・レジストリーからディスカバーできます。これらのサービスは、バンドル相互間のコラボレーションの 1 次的な手段です。OSGi サービスは、1 つ以上の Java インターフェース名でサービス・レジストリーに公開される Plain Old Java Object (POJO) であり、オプションのメタデータはカスタム・プロパティー (名前/値のペ

ア) として保管されます。ディスカバーする側のバンドルは、インターフェース名によってサービス・レジストリーでサービスを検索することができ、カスタム・プロパティーに基づいて検索されるサービスを潜在的にフィルターに掛けることができます。

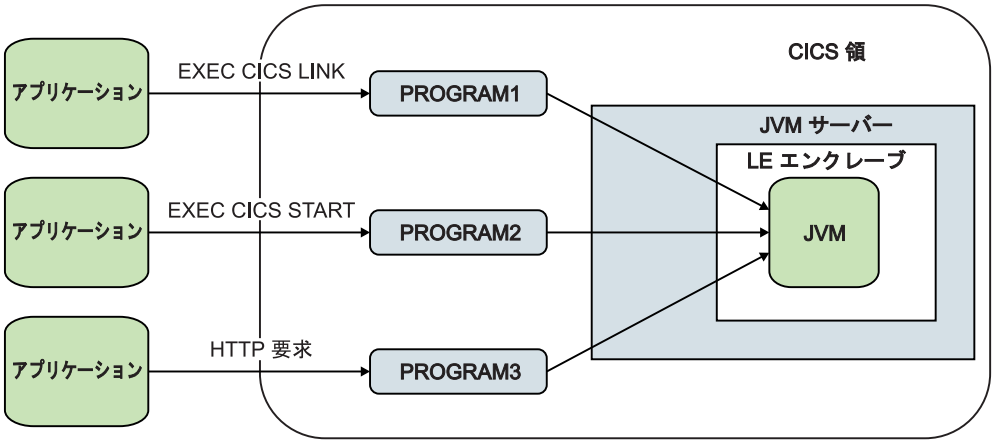
サービスは完全に動的であり、通常、それらのサービスを提供するバンドルと同じライフサイクルを持ちます。

JVM サーバー・ランタイム環境

JVM サーバー は、単一の 64 ビット JVM でさまざまな Java アプリケーションに対する複数の並行要求を処理できるランタイム環境です。JVM サーバーを使用して、OSGi フレームワークでスレッド・セーフ Java アプリケーションを実行し、Axis2 Web サービス・エンジンで Web サービス要求を処理することができます。

JVM サーバーは JVMSERVER リソースで表されます。JVMSERVER リソースを使用可能にすると、CICS は MVS™ にストレージを要求し、Language Environment® エンクレーブをセットアップし、そのエンクレーブで 64 ビット JVM を起動します。CICS は、JVMSERVER リソースで指定された JVM プロファイルを使用して、正しいオプションを持つ JVM を作成します。 このプロファイルで、Java アプリケーションから WebSphere® MQ にアクセスするためのネイティブ・ライブラリーを追加したり、JVM オプションを指定することができます。z/OS 上の Java は JVM メモリーとガーベッジ・コレクションを効率よく管理するので、プロファイルでこれらについてのオプションを設定する必要はありません。

JVM サーバーを使用する利点の 1 つは、同一 JVM で別々のアプリケーションに対して複数の要求を実行できることです。次の図では、3 つのアプリケーションが、別々のアクセス方式を使用して、CICS 領域内の 3 つの Java プログラムを同時に呼び出しています。各 Java プログラムは同じ JVM サーバーで実行されます。



Java アプリケーション

JVM サーバーで Java アプリケーションを実行するには、その Java アプリケーションがスレッド・セーフであり、1 つの CICS バンドル内の 1 つ以上の OSGi バンドルとしてパッケージされなければなりません。JVM サーバーは、OSGi バンドルおよび、OSGi サービスを実行できる OSGi フレームワークを実装します。OSGi

フレームワークは、サービスを登録し、バンドル相互間の依存関係とバージョンを管理します。OSGi は、フレームワーク内のすべてのクラスパス管理を処理するので、JVM サーバーの停止と再始動を行うことなく、Java アプリケーションを追加、更新、および除去できます。

OSGi を使用してパッケージされる Java アプリケーションのデプロイメントの単位は、CICS バンドルです。 CICS バンドルは、OSGi バンドルが入っている zFS のディレクトリーで使用可能でなければなりません。BUNDLE リソースは、アプリケーションを CICS に対して表現するもので、これを使用してアプリケーションのライフサイクルを管理することができます。CICS Explorer SDK は、CICS バンドル・プロジェクト内の OSGi バンドルを zFS にデプロイするためのサポートを提供します。

OSGi フレームワーク外部から Java アプリケーションにアクセスするには、PROGRAM リソースを使用して、アプリケーションが実行される JVM サーバー、および OSGi サービスの名前を識別してください。OSGi サービスは、CICS メイン・クラスを指します。

JVM サーバーにおける OSGi フレームワークの使用について詳しくは、26 ページの『OSGi に準拠する Java アプリケーション』を参照してください。

Web サービス

JVM サーバーを使用して、Web サービス・リクエスターおよびWeb サービス・プロバイダーのアプリケーションの SOAP 処理を実行できます。 Java ベースの SOAP エンジンである Axis2 をパイプラインが使用する場合、SOAP 処理は JVM サーバーで行われます。Web サービスに JVM サーバーを使用する利点は、作業を zAAP プロセッサーにオフロードできることです。

Web サービスに対する JVM サーバーの使用について詳しくは、22 ページの『Java Web サービス』を参照してください。

TP および T8 TCB

CICS はオープン・トランザクション環境 (OTE) を使用して JVM サーバーの作業を実行します。各タスクは、JVM サーバーのスレッドとして実行され、1 つの T8 TCB を使用して接続されます。JVM サーバーには、TP と呼ばれる親 TCB もあります。JVM サーバーが初期化され、システム・スレッドで実行されるときに、TP TCB が作成されます。システム・スレッドは、JVM サーバーの状態の照会、統計情報の収集、および JVM サーバーの停止のためのアクセスを提供します。

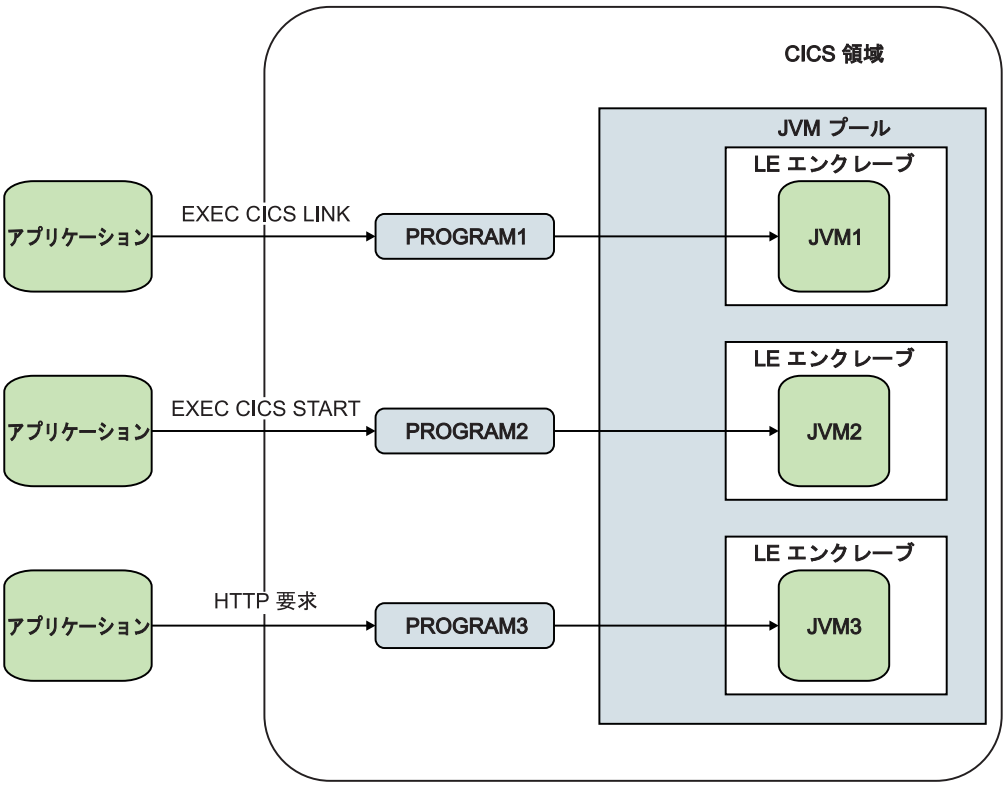
各タスクは、1 つの T8 TCB を使用して JVM のスレッドに接続されます。JVMSERVER リソースの THREADLIMIT 属性を設定すると、JVM サーバーから使用可能な T8 TCB の数を制御できます。JVM サーバー用に作成される T8 TCB は、仮想プールに存在し、同じ CICS 領域で実行される別の JVM サーバーで再利用できません。1 つの CICS 領域内に存在できる T8 TCB の最大数は、すべての JVM サーバーをあわせて 1024 です。また、ある 1 つの JVM サーバーでの最大数は 256 です。

プールされた JVM

プールされた JVM は、CICS タスクから一度に 1 つの要求のみを処理できる JVM です。これらの JVM のプールは、複数のタスクを同時に処理できます。つまり、Java ワークロードを処理するために多数の JVM が必要です。CICS はオープン・トランザクション環境 (OTE) を使用して、プールされた JVM を実行します。CICS が領域で作成できる JVM の数を管理することができます。

プールされた JVM は、1 つの Java プログラムのみを実行して、JVM が関係する各トランザクションが、JVM が関係する他のすべての並行トランザクションから分離されるようにします。したがって、Java プログラムを同時に処理するのに使用可能な複数の JVM が必要です。すべての新規 Java ワークロードには、JVM サーバー・ランタイム環境を使用してください。JVM サーバーでは、単一の JVM を使用して複数の Java プログラムを同時に実行できます。

次の図では、3 つのアプリケーションが、別々のアクセス方式を使用して、CICS 領域内の 3 つの Java プログラムを同時に呼び出しています。各要求は、別々の JVM とエンクレーブで実行されなければなりません。



JVM の再利用

Java プログラムが終了したら、プールされた JVM を別の Java プログラムに再割り当てすることができます。JVM プロファイルにより、JVM の特性、および再利用できるかどうかが決まります。JVM が再利用可能である場合、*継続 JVM* と呼ばれます。 JVM が再利用できない場合、*単独使用 JVM* と呼ばれます。プールされた JVM を使用する必要がある場合、継続 JVM を使用してパフォーマンスを改善してください。また、継続 JVM で共用クラス・キャッシュを使用して、ストレージ所要量を減らし、JVM の起動時間を改善することもできます。

継続 JVM は複数回再利用できます。次の Java プログラムまたはトランザクションで実行されるアプリケーション・コードは、前のプログラム呼び出しのアクションから自動的に分離されることはありません。すなわち、直列分離は自動ではありません。Java アプリケーション・プログラムが、望ましくない方法で継続 JVM の状態を変更したり、JVM を不要な状態のままにしたりしないことを確認する必要があります。

継続 JVM は、プログラム呼び出し間でストレージ・ヒープの内容を維持します。静的または動的状態は、継続 JVM のストレージ・ヒープに存続し、静止していないスレッドは、関連したストレージと一緒に存続します。JVM にロードされたすべてのアプリケーション・クラスは、そのまま保持されます。アプリケーションは、不要な項目のクリーンアップや必要な項目の保持を選択することができます。

Java プログラムの PROGRAM リソースにより、プログラムが使用する JVM の適切な実行キーと JVM プロファイルが決まります。 Java プログラムの要件に対応する、各種の JVM プロファイルを定義できます。

CICS は Java プログラムの実行要求を受け取ると、適切な JVM を作成するか、現在使用されていない既存の JVM を割り当てる必要があります。適切な JVM を作成するには、CICS は MVS にストレージを要求し、Language Environment エンクレーブをセットアップし、そのエンクレーブで JVM を起動します。CICS は、PROGRAM リソースで指定された JVM プロファイルを使用して、正しいクラスとオプションを持つ JVM を作成します。

JVM プール内の JVM の制限

プールされた JVM はそれぞれ、J8 および J9 のオープン TCB のプールから割り振られる MVS TCB 上で実行されます。オープン TCB のこのプールは JVM プールと呼ばれます。 JVM は、2 つの実行キー (ユーザー・キーまたは CICS キー) のいずれかに入れることができます。ユーザー・キーに入っている JVM は、J9 TCB 上で実行されます。CICS キーに入っている JVM は、J8 TCB 上で実行されます。統計はそれぞれのモードについて別々に収集されるので、各モードがどんな比率で JVM プールに入っているかを確認することができます。 JVM プロファイルと実行キーの間に依存関係はないため、2 つの JVM で同じプロファイルを使用しながら、異なる実行キーを使用することも可能です。

JVM のために作成できる TCB の総数は、MAXJVMTCBS システム初期設定パラメーターによって制限されます。このパラメーターは CICS 領域内の JVM プールに入れることのできる JVM の数の限度を定めます。

各 JVM はそれぞれの Language Environment エンクレーブ内で実行し、MVS ストレージを使用します。そのため、JVM で使用されるプロセッサー時間だけを考慮に入れて CICS 領域の **MAXJVMTCBS** 限度を選択するのではなく、以下の点も考慮に入れる必要があります。 **MAXJVMTCBS** 制限を非常に大きく設定した場合には、CICS は、使用可能な MVS ストレージ用に大量の JVM を作成しようとし、その結果 MVS ストレージは制限されます。

JVM プロファイル

JVM プロファイルは、JVM の特性を決定する Java ランチャー・オプションとシステム・プロパティーが入っているテキスト・ファイルです。任意の標準テキスト・エディターを使用して JVM プロファイルを編集することができます。

JVM プロファイルは、Java 用の CICS ランチャーで使用されるオプションをリストします。CICS に固有のオプションもあれば、JVM ランタイム環境に標準のオプションもあります。例えば、JVM プロファイルは、ストレージ・ヒープの初期サイズや拡張できる程度を制御します。また、プロファイルは、JVM によって作成されるメッセージやダンプ出力の宛先を定義することもできます。

JVM プロファイルは、クラスパスも指定します。クラスパスには、アプリケーションに必要なアプリケーション・クラスやリソースを JVM が検索するディレクトリーが入っています。

CICS が Java プログラムの実行要求を受け取ると、JVM プロファイルの名前が Java ランチャーに渡されます。 JVM プロファイルのオプションおよび JVM プロパティー・ファイル (指定されている場合) を使用して作成された JVM で、Java プログラムが実行されます。

CICS が使用する JVM プロファイルは、JVMPROFILEDIR システム初期設定パラメーターによって指定される z/OS UNIX システム・サービス・ディレクトリー内にあります。このディレクトリーには、CICS が JVM プロファイルを読み取るための適切な権限が必要です。

サンプル JVM プロファイル

CICS は、Java 環境の構成に役立つ 4 つのサンプル JVM プロファイルを提供します。これらのプロファイルは、CICS のインストール処理時にカスタマイズされます。これらのファイルは、CICS でデフォルトとして使用されるか、システム・プログラムに使用されます。

これらのサンプルをコピーし、独自のアプリケーションに合わせてカスタマイズすることができます。 CICS 提供のサンプル JVM プロファイルは、z/OS UNIX 上の `/usr/lpp/cicsts/cicsts42/JVMProfiles` ディレクトリーに置かれています。これらのサンプルをインストール・ディレクトリーから、**JVMPROFILEDIR** システム初期設定パラメーターで指定されたディレクトリーにコピーしてください。インストール場所にあるサンプル JVM プロファイルは、これらのファイルの変更を含む APAR を適用すると、上書きされます。変更内容が失われないように、必ず、サンプルを別の場所にコピーしてから、独自のアプリケーション・クラスの追加またはオプションの変更を行ってください。

サンプル JVM プロファイルには、Java のインストール・ディレクトリーの名前の可変部分にシンボル &JAVA_HOME が含まれています。 CICS のインストール時に、このシンボルは独自の値に置き換えられます。JVM の基本ライブラリー・パスと基本クラスパスは、JVM プロファイルには表示されませんが、これらのディレクトリーを使用して自動的に作成されます。&JAVA_HOME シンボルのデフォルト値は、`java/` です。

次の表は、各サンプル JVM プロファイルの主な特性をまとめています。

表 1. CICS 提供のサンプル JVM プロファイル

JVM プロファイル	特性
DFHJVMAX	DFHJVMAX プロファイルは、Axis2 JVM サーバー用に提供されたサンプル・プロファイルです。この JVM プロファイルは JVMSERVER リソースで指定されます。CICS は DFHJVMAX プロファイルを使用して、JVM サーバーを初期化します。独自のアプリケーションの PROGRAM リソースでこのプロファイルを指定しないでください。代わりに、PROGRAM リソースで JVMSERVER リソースの名前を指定してください。
DFHOSGI	DFHOSGI プロファイルは、OSGi JVM サーバー用に提供されたサンプル・プロファイルです。この JVM プロファイルは JVMSERVER リソースで指定されます。CICS は DFHJVMAX プロファイルを使用して、JVM サーバーを初期化します。独自のアプリケーションの PROGRAM リソースでこのプロファイルを指定しないでください。代わりに、PROGRAM リソースで JVMSERVER リソースの名前を指定してください。
DFHJVMPR	Java プログラムの PROGRAM リソースで JVM プロファイルが指定されない場合、DFHJVMPR プロファイルが、プールされた JVM のデフォルトです。DFHJVMPR プロファイルを使用して作成されたプールされた JVM は、このプロファイルが CLASSCACHE=YES を指定するため、共用クラス・キャッシュを使用します。
DFHJVMCD (CICS 用に予約済み)	CICS 提供のシステム・プログラムには、プールされた JVM 用に独自の JVM プロファイル DFHJVMCD があります。システム・プログラムは、デフォルトの JVM プロファイル DFHJVMPR に加えられた変更とは無関係です。特に、デフォルトの要求プロセッサー・プログラム DFJIIRP の PROGRAM リソースは、DFHJVMCD を指定します。DFHJVMCD プロファイルを使用して作成されたプールされた JVM は、このプロファイルが CLASSCACHE=NO を指定するため、共用クラス・キャッシュを使用しません。デフォルト値を変更できます。独自の Java アプリケーション用にセットアップした PROGRAM リソースでこのプロファイルを指定しないでください。ただし、CICS 領域に対して正しくセットアップされていることを確認する必要があります。 CICS は、DFHJVMCD を CICS 提供のシステム・プログラムに使用するのに加えて、共用クラス・キャッシュの初期化と終了にも使用します。

JVM の構造

CICS で実行される JVM は、JVM プロファイルで定義される 1 組のクラスとクラスパスを使用し、64 ビット・ストレージを使用します。各 JVM が実行される Language Environment エンクレーブを調整すると、MVS ストレージを最も効率よく使用することができます。

IBM 64-bit SDK for z/OS, Java テクノロジー・エディション のバージョン 6.0.1 について詳しくは、「*IBM 64-bit SDK for z/OS, Java Technology Edition, Version 6.0.1 SDK and Runtime Environment User Guide*」を参照してください。この資料は、www.ibm.com/servers/eserver/zseries/software/java/javaintr.html からダウンロードできます。

JVM におけるクラスおよびクラスパス

CICS で実行される JVM では、3 つのタイプのクラスとネイティブ・ライブラリーが使用されます。

- JVM で基本サービスを提供する z/OS JVM コード。これらのクラスは、システム・クラス および*標準拡張クラス* であり、総称して*原始クラス* と呼ばれます。
- JVM によって使用されるネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイル。z/OS UNIX では、これらのファイルには拡張子 `.so` があります。 JVM の実行には何らかのライブラリーが必要であり、アプリケーション・コードまたはサービスによって追加のネイティブ・ライブラリーをロードすることができます。例えば、追加のネイティブ・ライブラリーには、DB2® JDBC ドライバーを使用する DLL ファイルを含めることができます。
- JVM で実行されるアプリケーションの Java クラス。これらのクラスはアプリケーション・クラス と呼ばれます。このグループには、ユーザー作成アプリケーションの一部であるクラスが含まれます。また、CICS の標準 JVM セットアップに含まれていないリソース (JCICS インターフェース・クラス、JDBC、JNDI など) にアクセスするサービスを提供するために、IBM または他のベンダーによって提供されるクラスも含まれます。アプリケーション・クラスがロードされると、他のトランザクションで使用できるように、JVM が再利用されても保持されます。

JVM は、これらの各項目の目的を認識し、クラスまたはネイティブ・ライブラリーが JVM によってロードされる方法、および保管される場所を判別します。

JVM のクラスパスは、JVM プロファイルのオプションによって定義され、オプションとして、参照される JVM プロパティー・ファイルで定義されます。

クラスまたはネイティブ・ライブラリーを組み込むことができるクラスパスは次のとおりです。

- *ライブラリー・パス* は、JVM が使用するすべてのネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイル用です。これには、アプリケーション・コードまたはサービスによってロードされる JVM および追加のネイティブ・ライブラリーの実行に必要なファイルが含まれます。各 DLL ファイルの 1 つのコピーのみがロードされ、すべての JVM がそのコピーを共用しますが、各 JVM には、その DLL 用に静的データ域の独自のコピーがあります。

  JVM の基本ライブラリー・パスは、**USSHOME** システム初期設定パラメーターと JVM プロファイルの **JAVA_HOME** オプションで指定されたディレクトリーを使用して自動的に作成されます。この基本ライブラリー・パスは、JVM プロファイルでは表示されません。このライブラリー・パスには、CICS が使用する JVM とネイティブ・ライブラリーを実行するのに必要なすべての DLL ファイルが含まれています。 **LIBPATH_SUFFIX** オプションまたは **LIBPATH_PREFIX** オプションを使用して、このライブラリー・パスを拡張できます。 **LIBPATH_SUFFIX** は、ライブラリー・パスの終わりに、IBM 提供のライブラリーの後に項目を追加します。 **LIBPATH_PREFIX** は、項目を先頭に追加し、同じ名前である場合は IBM 提供のライブラリーの代わりにロードされます。問題判別の目的でそれを実行することが必要になる場合もあります。

ライブラリー・パスに組み込むすべての DLL ファイルを、LP64 オプションを使用してコンパイルし、リンクしてください。基本ライブラリー・パスで提供される DLL ファイルおよび DB2 JDBC ドライバーなどのサービスで使用される DLL ファイルは、LP64 オプションを指定して作成されます。

- *標準クラスパス*は、プールされた JVM、または OSGi 用に構成されていない JVM サーバーで実行されるすべてのアプリケーション・クラス用です。 Java `.class` および `.jar` ファイルはすべて、標準クラスパスに置かれます。JVM プロファイルの **CLASSPATH_SUFFIX** オプションまたは **CLASSPATH_PREFIX** オプションを使用して、標準クラスパスにクラスを追加することができます。

  また、CICS は、**USSHOME** システム初期設定パラメーターで指定されたディレクトリーの `/lib` サブディレクトリーを使用して、JVM の基本クラスパスを自動的に作成します。このクラスパスには、CICS と JVM によって用意されている JAR ファイルが入ります。それは JVM プロファイルでは見られません。

  OSGi をサポートするように構成されている JVM サーバーの場合、アプリケーション・クラスにクラスパスを設定してはなりません。 OSGi フレームワークは、アプリケーションを含む OSGi バンドル内の情報を使用することによって、アプリケーションのクラスパスを自動的に判別します。

システム・クラスと標準拡張クラス (原始クラス) は既に JVM のブート・クラスパスに含まれているので、これらをクラスパスに組み込む必要はありません。

JVM におけるストレージ・ヒープ

IBM 64-bit SDK for z/OS, Java テクノロジー・エディション バージョン 6.0.1 の JVM におけるランタイム・ストレージは、単一の 64 ビット・ストレージ・ヒープによって管理されます。

各 JVM のヒープは、JVM の Language Environment エンクレーブにある 64 ビット・ストレージから割り振られます。各ヒープのサイズは、JVM プロファイル内のオプションによって決まります。

単一のストレージ・ヒープはヒープ と呼ばれ、場合によってはガーベッジ・コレクション・ヒープ と呼ばれます。その初期ストレージ割り振りは、JVM プロファイルの **-Xms** オプションによって設定され、その最大サイズは **-Xmx** オプションによって設定されます。

ヒープのサイズを調整すると、JVM の最適なパフォーマンスを実現することができます。 184 ページの『JVM サーバー・ヒープとガーベッジ・コレクションの調整』および 193 ページの『プールされた JVM のヒープとガーベッジ・コレクションの調整』を参照してください。

JVM が構成される場所

JVM が必要な場合、JVM の CICS ランチャー・プログラムは MVS にストレージを要求し、Language Environment エンクレーブをセットアップし、その Language

Environment エンクレーブで JVM を起動します。並行して実行される JVM 間の分離を確実にするために、各 JVM は独自の Language Environment エンクレーブ内で構成されます。

Language Environment エンクレーブは、Language Environment 事前初期設定モジュール CELQPIPI を使用して作成され、JVM は z/OS UNIX プロセスとして実行されます。したがって、JVM は、CICS Language Environment サービスではなく、MVS Language Environment サービスを使用します。JVM に使用されるストレージは、MVS Language Environment サービスの呼び出しによって取得された MVS 64 ビット・ストレージです。このストレージは、CICS アドレス・スペース内に置かれますが、CICS 動的ストレージ域 (DSA) には含まれません。

JVM の Language Environment エンクレーブは、その JVM のストレージ要件に応じて拡張することができます。Language Environment エンクレーブに CICS が使用する Language Environment ランタイム・オプションは、Language Environment エンクレーブのヒープ・ストレージの初期サイズおよび追加増分を制御します。

Language Environment エンクレーブに CICS が使用するランタイム・オプションを調整することができます。その結果、CICS がそのエンクレーブのために要求するストレージ量は、JVM プロファイルで指定されたストレージ量と可能なかぎり近くなります。したがって、MVS ストレージを最も効率よく使用することができます。ストレージの調整について詳しくは、 198 ページの『JVM の Language Environment エンクレーブ・ストレージ』を参照してください。

JVM の実行キー

Java プログラムは、正しい実行キーで実行される JVM を使用する必要があります。プールされた JVM は、2 つの実行キー、すなわちユーザー・キーまたは CICS キーのいずれかで実行できます。JVM サーバーは CICS キーのみで実行されます。

JVM サーバーの実行キー

JVM サーバーは CICS キーのみで実行されます。JVM サーバーを使用するには、Java プログラムの PROGRAM リソースで、EXECKEY 属性が CICS に設定されなければなりません。 CICS は、T8 TCB を使用して JVM を実行し、CICS キーの MVS ストレージを取得します。

プールされた JVM の実行キー

Java プログラムの PROGRAM リソースで EXECKEY 属性を USER に設定する場合、CICS は、ユーザー・キーにあるプールされた JVM にそのプログラムを提供します。CICS は、J9 TCB を使用して JVM を実行し、ユーザー・キーの MVS ストレージを取得します。EXECKEY 属性を CICS に設定する場合、CICS は、CICS キーにある JVM にプログラムを提供します。CICS は、J8 TCB を使用して JVM を実行し、CICS キーの MVS ストレージを取得します。

ユーザー・キーでアプリケーションを実行すると、CICS ストレージ保護が拡張されるため、Java プログラムは、可能な場合、ユーザー・キーで実行されるプールされた JVM を使用します。ただし、Java プログラムが、**TASKDATAKEY**(CICS) を指定するトランザクションの一部である場合、そのプログラムは、CICS キーで実行される JVM を使用する必要があります。

Java PROGRAM リソースの EXECKEY 属性を変更する場合、他の変更を加える必要はありません。CICS は、同じ JVM プロファイルを使用して、両方の実行キーで JVM を作成することができます。単一の CICS タスクには、CICS キーで稼働する Java プログラム、およびユーザー・キーで稼働する Java プログラムを含めることができます。ただし、PROGRAM リソースで同じ実行キーと JVM プロファイルを指定しているプログラムでのみ、JVM を再利用できます。大部分の JVM が同じ実行キーで作成される場合、CICS では、新しい JVM を作成するのではなく、再利用される既存の JVM にプログラムを提供する可能性が高くなります。

JVM および z/OS 共用ライブラリー領域

共用ライブラリー領域は、アドレス・スペースがダイナミック・リンク・ライブラリー (DLL) ファイルを共用できるようにする z/OS 機能です。

この機能により、 CICS 領域は JVM に必要な DLL を共用できるようになり、各領域が DLL を個別にロードする必要はなくなります。これにより、MVS が使用する実際のストレージの量、および領域へのファイルのロード所要時間を大幅に削減できます。

共用ライブラリー領域用に予約されているストレージは、最初の JVM が領域で開始されるときにそれぞれの CICS 領域に割り振られます。割り振られるストレージの量は、z/OS の **SHRLIBRGNSIZE** パラメーターによって制御されます。共用ライブラリー領域に割り振られるストレージ量の調整について詳しくは、206 ページの『z/OS 共用ライブラリー領域の調整』を参照してください。

共用クラス・キャッシュ

IBM SDK for z/OS が JVM に提供するクラス共用機能では、複数の JVM が、既にロードされているクラス・ファイルの単一キャッシュを共用できます。CICS がプールされた JVM と JVM サーバーに対してこの機能をサポートする方法が異なります。

共用クラス・キャッシュには、共用クラス・キャッシュを使用する JVM で必要なすべてのクラスが含まれています。Java プログラムで必要なアプリケーション・クラスはすべて、JVM プロファイルの標準クラスパスに置かれ、すべてが共用クラス・キャッシュへのロード対象です。一部の例外的なシナリオでは、一部のクラスが共用クラス・キャッシュへのロード対象でない場合があります。

共用クラス・キャッシュは次の項目を保管しません。

- JVM プロファイルのライブラリー・パスで指定されるネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイル。各 DLL ファイルの単一コピーが、それを必要とするすべての JVM で使用されます。
- アプリケーションの作業データ (オブジェクトおよび変数)。作業データは個々の JVM に保管されます。
- ジャストインタイム (JIT) コンパイルによって作成されるコンパイル済みのクラス。コンパイル済みのクラスは、共用クラス・キャッシュではなく、個々の JVM に保管されます。これは、コンパイル・プロセスがワークロードによって異なる可能性があるからです。

アプリケーション・クラスまたは JAR ファイルを変更するか、JVM プロファイルのクラスパスに新規項目を追加し、該当する JVM を再始動する場合、共用クラス・キャッシュはその内容を自動的に更新します。共用クラス・キャッシュは、z/OS の IPL などの一部の状況を除いて、CICS のウォーム・スタートや緊急スタートがあっても持続します。したがって、そうした場合に CICS 領域で最初の JVM の開始コストがありません。

Java 6.0.1 では、同時に複数の共用クラス・キャッシュを使用可能にすることができます。CICS は、複数のクラス・キャッシュを管理するためのインターフェースを提供しませんが、JVM サーバーで複数のクラス・キャッシュを使用することができます。プールされた JVM は複数のクラス・キャッシュを使用できませんが、CICS は、プールされた JVM の領域内の単一クラス・キャッシュを管理するためのインターフェースを提供します。

JVM サーバーのクラス・キャッシュ

JVM サーバーでクラス・キャッシュを使用したい場合は、Java 6 によって直接提供されるサポートを使用できます。このサポートは、Class data sharing between JVMs で説明されています。JVM サーバーは、CICS で提供される、クラス・キャッシュのサポートを使用しません。例えば、SPI または CEMT コマンドを使用して JVM サーバーのクラス・キャッシュを使用可能または使用不可にすることはできません。

プールされた JVM のクラス・キャッシュ

共用クラス・キャッシュを使用するプールされた JVM は、共用クラス・キャッシュを使用しない JVM に比べて、より迅速に始動し、ストレージ要件は低くなります。また、プールされた JVM で共用クラス・キャッシュを使用する場合、クラス・ロードの全体的なコストも削減されます。クラス・キャッシュを共用する新規の JVM が初期設定されると、ファイル・システムからクラスを読み取る代わりに、プリインストール済みのクラスを使用します。システム内で処理される Java アプリケーション間の独立性を維持するために、クラス・キャッシュを共用する JVM は、その JVM で実行されるアプリケーションのすべての作業データ (オブジェクトおよび変数) を引き続き所有します。

CICS は、CICS 提供のサンプル・プロファイル DFHJVMCD を使用して、プールされた JVM の共用クラス・キャッシュの初期化と終了を行います。DFHJVMCD は、CICS 領域で使用するために常に使用可能であり、構成されなければなりませんが、共用クラス・キャッシュで使用するために追加の変更を加える必要はありません。

CICS は、各領域で 1 つのアクティブな共用クラス・キャッシュを管理するためのインターフェースを提供します。領域内に消去された古い共用クラス・キャッシュが含まれている可能性もあります。CICS コマンドを使用して共用クラス・キャッシュを管理し、その状況をモニターすることができます。

共用クラス・キャッシュには、`CICS_sharedcc_`*`APPLID_n`* という名前が付けられます。ここで、*APPLID* は CICS 領域のアプリケーション ID であり、*n* はゼロから始まる世代番号です。世代番号は、新しい共用クラス・キャッシュの名前を区別するために使用されます。

CICS は、共用クラス・キャッシュの管理機能に 1 つ以上の JM TCB (オープン TCB のタイプ) を使用します。JM TCB は、JVM プールの **MAXJVMTCBS** 制限の対象外です。

JVMCCSIZE システム初期設定パラメーターは、共用クラス・キャッシュの初期サイズを指定します。JVMCCSTART システム初期設定パラメーターは、CICS 領域の初期化時における共用クラス・キャッシュの開始動作を制御します。

第 2 章 Java の計画

企業で Java を使用する方法を計画する場合、このセクションの例は、CICS アプリケーションに使用可能なさまざまな戦略的オプションに関する手引きになります。

CICS では、次のようなさまざまな方法で Java を使用できます。

JCA を使用して外部 Java アプリケーションを CICS に接続する

Java EE Connector Architecture (JCA) を使用すると、CICS Transaction Gateway を使用することによって既存の CICS アプリケーションを外部 Java アプリケーションに接続できます。CICS ファミリーのこの製品は、JCA テクノロジーを実装するリソース・アダプターを使用することによって、アプリケーション・サーバー (WebSphere Application Server など) 内の Java アプリケーションを CICS に接続することをサポートします。

CICS アプリケーションは、サポートされている任意の高水準プログラミング言語で作成できます。

Java Web サービスを使用する

異機種混合環境でサービス・プロバイダーとサービス・リクエスターを処理するための Java Web サービスを作成して、HTTP または WebSphere MQ を介してインターネットに接続できます。Java Web サービスは JVM サーバーで実行され、SOAP 処理は Axis2 Web サービス・エンジンによって実行されます。 Axis2 で既存の Web サービスを処理することを選択できます。この場合、プロバイダーまたはリクエスターのアプリケーションは、サポートされている任意の高水準プログラミング言語 (Java を含む) で作成されます。また、標準の Java API を使用して、XML を処理したり、構造化データを操作したりする Java Web サービスを作成することもできます。

JVM サーバーで実行される Java ワークロードは、IBM System z Application Assist Processor (zAAP) での実行に適格です。

OSGi を使用して Java アプリケーションを作成する

OSGi Service Platform に準拠する、再使用可能なモジュラー Java アプリケーションを作成できます。これらのアプリケーションは、CICS と他のプラットフォーム間での移植が容易であり、OSGi は、依存関係とバージョンの管理に細分性をもたらします。

Java CICS (JCICS) API を使用すると、ファイルまたは一時記憶域キューからの読み取りなどの CICS サービスにアクセスするアプリケーションを作成できます。Java アプリケーションは、他の CICS アプリケーションにリンクし、DB2 および IMS™ 内のデータにアクセスすることができます。Java アプリケーションは、JVM サーバーまたはプールされた JVM で実行できます。Java アプリケーションを実行するための戦略的な環境は JVM サーバーです。したがって、すべての Java アプリケーションにこの環境を使用するように計画してください。JVM サーバーで実行される Java ワークロードは、zAAP での実行に適格です。

計画の一環として、Java ワークロードをルーティングし、それに応じて CICS 領域を拡大する方法も決定する必要があります。

CICS Transaction Gateway からの CICS アプリケーションへのアクセス

CICS Transaction Gateway は、Java クライアント・プログラムを既存の CICS アプリケーションに接続するためのリソース・アダプターを提供します。

CICS TG リソース・アダプターを使用すると、新しい Java アプリケーションで CICS アプリケーションを再利用できます。多くの場合、既存の Java または非 Java CICS アプリケーションを再利用すると、新しい Java アプリケーションをより迅速かつ確実に開発できます。通常、Java クライアント・アプリケーションはネットワーク・ベースで、CICS プログラムは COBOL などの言語で作成されます。

J2EE コネクター・アーキテクチャー (JCA)

Java 2 Platform Enterprise Edition (J2EE) コネクター・アーキテクチャーは、CICS などの異機種混合のエンタープライズ情報システム (EIS) に対して J2EE 準拠プラットフォームが接続するための標準的方法を定義します。 Java アプリケーションは、JCA によって定義されるオープン・スタンダードである Common Client Interface (CCI) を使用して、リソース・アダプターと対話します。

J2EE コネクター・アーキテクチャーを使用すると、 EIS ベンダーはその EIS に標準のリソース・アダプターを提供できます。リソース・アダプターは Java アプリケーションと EIS の中間層にあり、この Java アプリケーションを EIS に接続します。

CICS Transaction Gateway は、 Common Client Interface をサポートする J2EE CICS リソース・アダプターを提供して JCA を実装します。

外部 Java プログラムからの CICS プログラムへのアクセス

Java クライアント・アプリケーションは、以下のいずれかの方法を使用してネットワークから CICS TS プログラムを呼び出せます。

CICS Transaction Gateway API
CICS Transaction Gateway API は、とりわけ以下の機能を提供します。

外部呼び出しインターフェース
外部アプリケーションは、外部呼び出しインターフェース (ECI) を使用して、CICS 領域でプログラムを呼び出すことができます。これを行うには、CICS プログラムを、**EXEC CICS LINK** コマンドを使用して他の CICS プログラムから使用可能にする必要があります。 COMMAREA インターフェースを備えることができるか、または IPIC 接続が使用される場合は、プログラムはチャネルとコンテナーを使用してデータを転送できます。

ECI 要求によって呼び出される CICS プログラムは、分散プログラム・リンク (DPL) 要求の規則に従う必要があります。 DPL 要求については、「CICS アプリケーション・プログラミング」の『分散プログラム・リンク (DPL)』を参照してください。

外部表示インターフェース

外部アプリケーションは、外部表示インターフェース (EPI) を使用して、3270 ベースの CICS アプリケーション・プログラムを呼び出し、その出力を使用できます。クライアント・アプリケーションは、CICS 領域で仮想 IBM 3270 端末をインストールし、削除できます。EPI が使用する定義は、CICS によってリモート 3270 端末定義として処理されるので、自動トランザクション開始要求 (ATI) をサポートします。

外部セキュリティー・インターフェース

外部アプリケーションは、外部セキュリティー・インターフェース (ESI) を使用して、特定のセキュリティー機能を実行できます。例えば、アプリケーションは、CICS 外部セキュリティー・マネージャー (ESM) に保持されているユーザー ID に関する情報にアクセスし、サーバー接続用のデフォルトのセキュリティー資格情報を設定することができます。

ECI リソース・アダプター

ECI リソース・アダプターには外部呼び出しインターフェースへの高水準の CCI インターフェースが備えられています。アプリケーションは、このインターフェースを使用して、CICS アプリケーションにリンクし、データを COMMAREA またはコンテナーに渡すことができます。リソース・アダプターを J2EE アプリケーション・サーバー (WebSphere Application Server など) に配置できるので、 J2EE エンタープライズ・アプリケーションは CICS にアクセスできます。 JCA を使用すると、接続プール、セキュリティー、およびトランザクション・コンテキストはそのアプリケーションによってではなく J2EE アプリケーション・サーバーによって管理されます。

z/OS の場合、以下の 2 つの ECI リソース・アダプターが、CICS Transaction Gateway に提供されます。

- アダプター `cicseciXA.rar`。2 フェーズ・コミットをサポートします。
- アダプター `cicseci.rar`。単一フェーズ・コミットのみをサポートします。

ECI リソース・アダプターは、以下の追加機能もサポートします。

IPIC 接続のサポート

領域が CICS TS for z/OS バージョン 3.2 以降である場合、IPIC 接続を使用して TCP/IP 上で CICS にアクセスできます。 EXCI、APPC、および TCP/IP を介した ECI とは異なり、このタイプの接続は、コンテナーおよび SSL 認証をサポートしています。 CICS では、IPIC 接続は IPCONN リソースで表されます。

静的 IPCONN リソースを外部 Java クライアントにインストールすることはできません。これらの接続は常に自動的にインストールされます。「Customization Guide」の『Writing a program to control autoinstall of IPIC connections』を参照してください。

チャネルとコンテナー

チャネルとコンテナーにより、アプリケーションは、32 KB より大きい CICS 内のデータを転送することができます。チャネルとコンテナーの詳細については、「CICS アプリケーション・プログラミング」の『チャネルを使用した拡張プログラム間データ転送』 を参照してください。

Secure Sockets Layer (SSL) 認証

Secure Sockets Layer (SSL) 認証。 SSL は、CICS Transaction Gateway および CICS 間の IPIC 接続でサポートされます。SSL 認証の使用については、「RACF Security Guide」の『Configuring CICS to use SSL』を参照してください。

EPI リソース・アダプター

EPI リソース・アダプターでは、外部表示インターフェースに対する高水準の CCI インターフェースが提供されていて、CICS 領域で端末のインストールおよび 3270 ベース・トランザクションの実行に使用できます。自動トランザクション開始 (ATI) はサポートしていません。リソース・アダプターを J2EE アプリケーション・サーバーに配置できるので、 J2EE エンタープライズ・アプリケーションは CICS にアクセスできます。 JCA を使用すると、接続プール、セキュリティー、およびトランザクション・コンテキストはそのアプリケーションによってではなく J2EE アプリケーション・サーバーによって管理されます。

CICS リソース・アダプターの使用例

21 ページの図 1 に示されているシナリオは、*3 層構成* の例です。Java アプリケーションは、ECI リソース・アダプターを使用して CICS 領域内のプログラムにリンクします。クライアント・アプリケーションと CICS 領域間の接続は、中間システムを介して行われます。クライアント・アプリケーションは CICS Transaction Gateway と同じホスト上で実行されていないため、デーモンはクライアントを listen して通信します。 z/OS では、CICS Transaction Gateway は CICS に要求を渡すために外部 CICS インターフェース (EXCI) または IPIC ドライバーを使用します。これらの要求は、ECI 呼び出しとして CICS によって処理されます。

この図は、ECI リソース・アダプターを使用してサーバー・プログラムに接続する Java サーブレットも示します。この構成は *2 層構成* の例です。ここでは、クライアント・アプリケーションと CICS 領域間には、ECI アダプターを介した直接接続が存在します。サーブレットは CICS TG と同じホスト上で実行されているため、通信するためにローカル・プロトコルを使用します。

CICS Transaction Gateway on z/OS では外部呼び出しインターフェースはサポートされますが、外部表示インターフェースはサポートされていません。 ECI と ECI リソース・アダプターはサポートされますが、 EPI または EPI リソース・アダプターはサポートされていません。 Java クライアント・プログラムだけがサポートされます。 ECI 呼び出しは、EXCI または IPIC 接続を使用して CICS 領域に対して行えます。

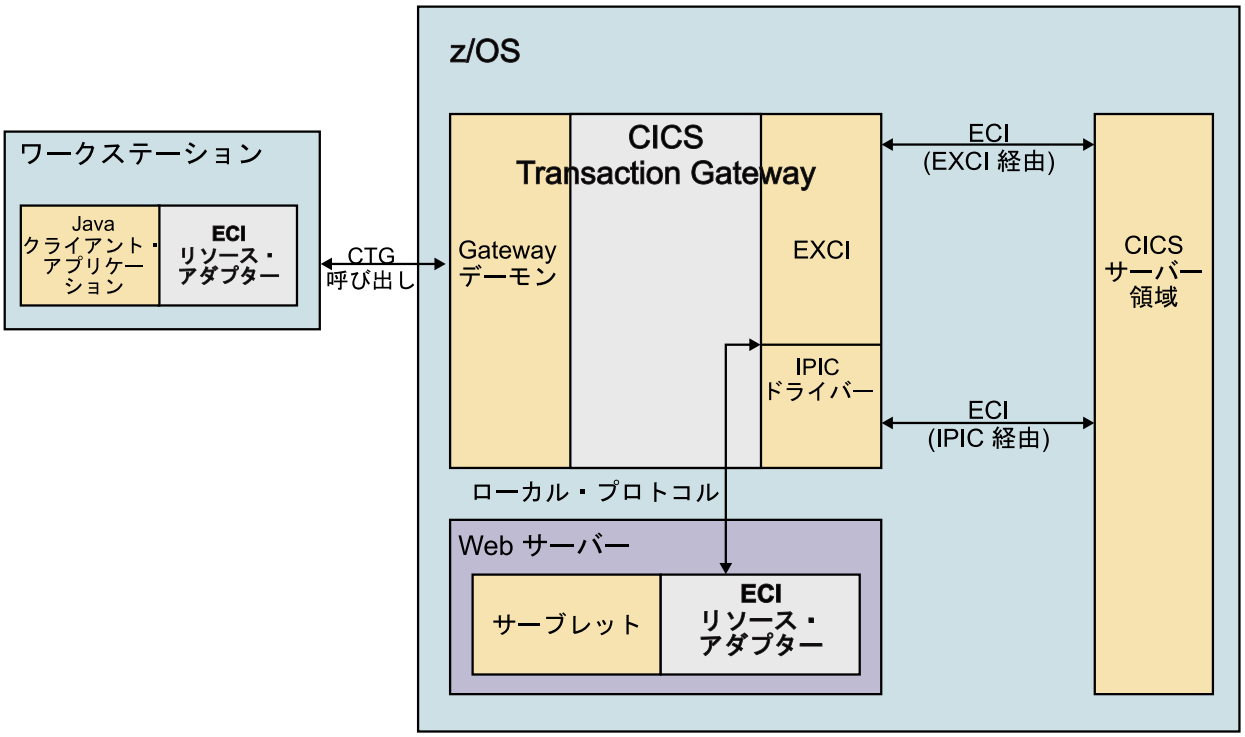


図 1. ECI リソース・アダプターを使用して CICS プログラムに接続する Java クライアント

22 ページの図 2 のようにすることもできます。この例では、CICS Transaction Gateway は Windows サーバーで実行されます。Windows および Linux 上の CICS Transaction Gateway は、ECI と ECI リソース・アダプター、および EPI と EPI リソース・アダプターの両方をサポートします。Java クライアントは、適切な COMMAREA またはコンテナーを使用する CICS プログラムに加えて、3270 ベースの CICS プログラムにもアクセスできます。

ECI 呼び出しは、APPC、TCP62、TCP/IP を介した ECI、または IPIC 接続上で CICS に対して行うことができます。 EPI 呼び出しは、APPC 接続でのみサポートされます。

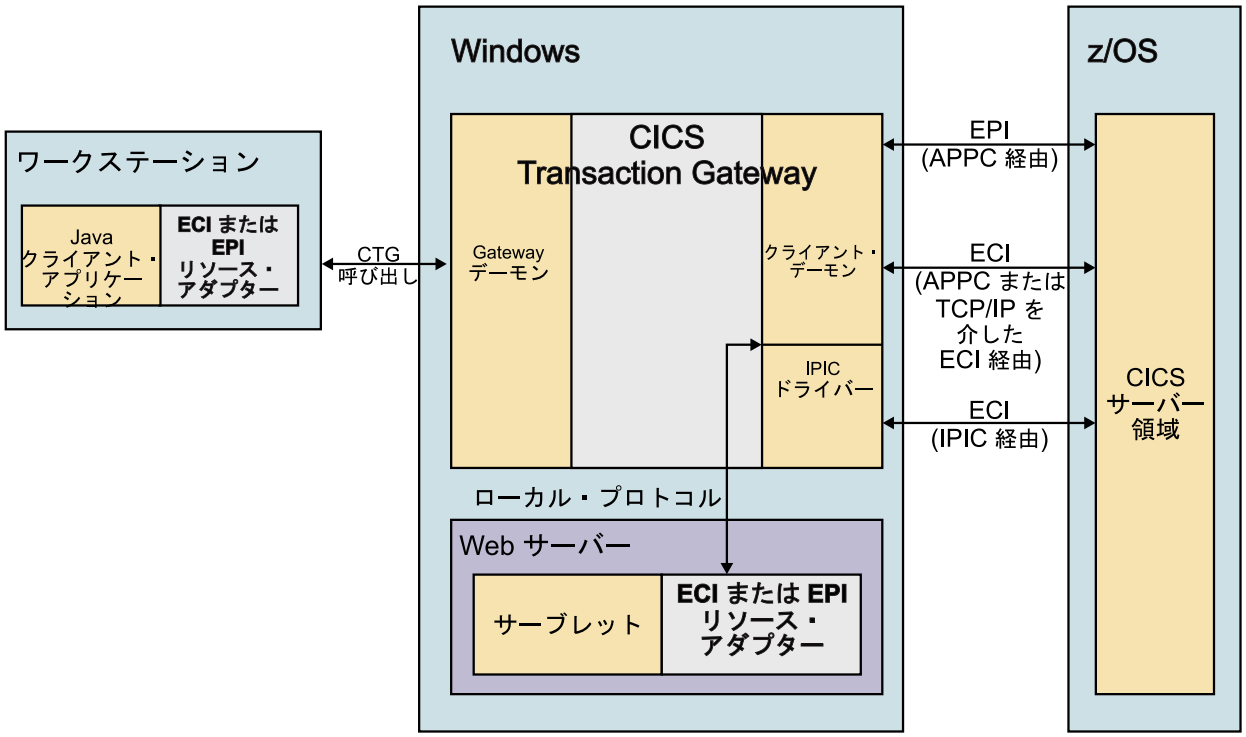


図 2. CICS 外部から CICS プログラムに接続する Java クライアント

この方法で CICS プログラムを使用するには、CICS アプリケーションの開発に関する知識が Java 開発者に求められます。

CICS Transaction Gateway

- Programming Guide
- Scenarios
- Programming Reference

IBM Redbooks®

- Developing Connector Applications for CICS
- Java Connectors for CICS Featuring the J2EE Connector Architecture

Java Web サービス

CICS には、Java Web サービスを実行するための Axis2 テクノロジーが組み込まれています。Axis2 は、Apache Foundation からのオープン・ソース Web サービス・エンジンであり、Java 環境で SOAP メッセージを処理するために CICS に提供されています。

Axis2 は、複数の Web サービス仕様をサポートする、Web サービス SOAP エンジンの Java ベースの実装です。また、Axis2 で実行できる Java アプリケーションの作成方法を記述するプログラミング・モデルも提供します。Axis2 は、Java 環境で Web サービスを処理するために CICS に提供されています。 Axis2 の応答時間

は、Java を使用しない同等なものよりもやや低速ですが、このタイプの Java ワークロードは、zAAP での実行に適格です。

JVM サーバーは、既存の Web サービスを変更することなく、Java ベースの SOAP パイプラインでインバウンドおよびアウトバウンド SOAP メッセージを処理するための Axis2 の実行をサポートします。ただし、Java アプリケーションから Web サービスを作成し、同じ JVM サーバーで実行することもできます。JVM サーバーの Axis2 リポジトリーにアプリケーションをデプロイすることによって、Java アプリケーションと SOAP 処理の両方が、zAAP での実行に適格になります。

次のいずれかの理由で Java Web サービスを使用できます。

- 他のプラットフォームで Axis2 Web サービスの経験があり、CICS で Web サービスを作成したい。
- 標準の Java API を使用して、Axis2 に統合される Java データ・バインディングを作成したい。
- CICS Web サービス・アシスタントで処理するのが困難な複雑な WSDL 文書がある。
- Web サービス・アプリケーションの処理を zAAP で実行したい。

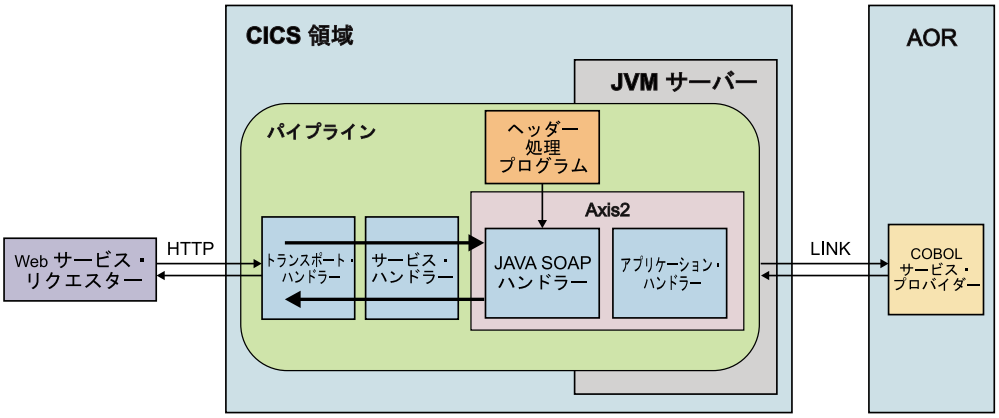
以下の例では、Web サービスで Java を使用する方法を説明しています。

JVM サーバーにおける SOAP メッセージの処理

Web サービス・パイプラインで発生する大部分の SOAP 処理は、SOAP ハンドラーおよびアプリケーション・ハンドラーによって実行されます。オプションとして、JVM サーバーでこの SOAP 処理を実行し、zAAP を使用して作業を実行できます。COBOL、C、C++、または PL/I で作成される Web サービス・アプリケーションを引き続き使用できます。

既存の Web サービスがある場合、JVM サーバーを使用するようにパイプラインの構成を更新できます。Web サービスに変更を加える必要はありません。パイプラインが SOAP ヘッダー処理プログラムを使用する場合、Axis2 プログラミング・モデルを使用することによって、Java でそのプログラムを再作成するのが最善の方法です。このヘッダー処理プログラムは、追加のデータ変換を行うことなく、Java オブジェクトを Axis2 と共用できます。例えば、COBOL のヘッダー処理プログラムがある場合、データが Java から COBOL に変換された後、再び戻す必要があります。これにより、SOAP 処理のパフォーマンスが低下する場合があります。

次の図に示されているシナリオは、Web サービス・プロバイダーである COBOL アプリケーションの例です。要求は、Java をサポートするように構成されているパイプラインで処理されます。SOAP ハンドラーおよびアプリケーション・ハンドラーは、Axis2 によって処理され、JVM サーバーで実行される Java プログラムです。アプリケーション・ハンドラーは、データを XML から COBOL に変換し、アプリケーションにリンクします。



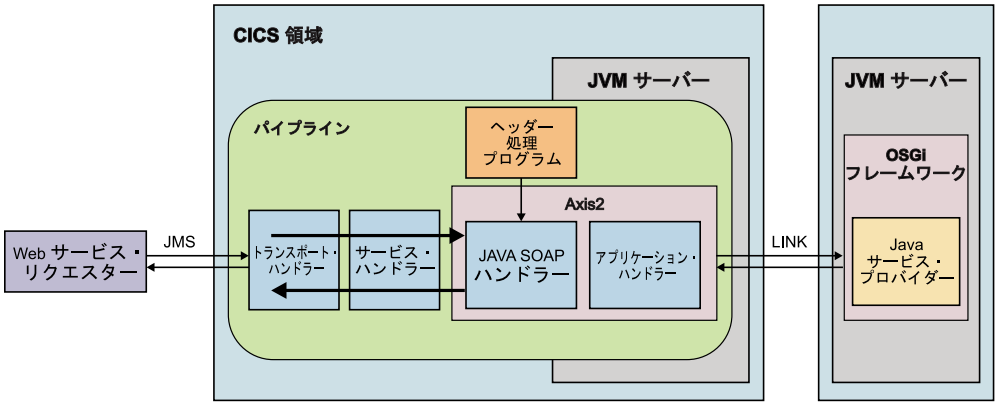
環境を計画する際には、JVM サーバーに 1 組の専用領域を使用するようにしてください。この例では、COBOL アプリケーションが実行されているアプリケーション専有領域 (AOR) は、JVM サーバーが実行される CICS 領域とは別です。ワークロード管理を使用すると、例えば **EXEC CICS LINK** ではアプリケーション・ハンドラーから、インバウンド要求では Web サービス・リクエスターからのように、ワークロードのバランスを取ることができます。

CICS Web サービス・アシスタントからの出力を使用する Java アプリケーションの作成

言語構造を解釈し、CICS Web サービス・アシスタントによって生成されるデータ・バインディングを使用する Java アプリケーションを作成できます。Web サービス・アシスタントは、WSDL から言語構造を作成し、言語構造から WSDL を作成することができます。また、このアシスタントは、SOAP 処理時に XML とターゲット言語間でデータを変換する方法を記述する Web サービス・バインディングも作成します。

アシスタントを使用して言語構造を生成する場合は、JZOS または J2C を使用して言語構造を処理して、Java クラスを生成できます。これらのツールを使用すると、Java 開発者は他の CICS アプリケーションと対話できます。この例では、これらのツールを使用して、CICS が XML からデータを変換した後にインバウンド SOAP メッセージを処理できる Java アプリケーションを作成することができます。詳しくは、55 ページの『Java からの構造化データとの対話』を参照してください。

次の図に示されているシナリオは、Web サービス・プロバイダーである Java アプリケーションの例です。SOAP 処理は、JVM サーバーで Axis2 によって処理されます。アプリケーション・ハンドラーがリンクする Java アプリケーションは、1 つ以上の OSGi バンドルとしてパッケージされ、デプロイされ、JVM サーバーで実行されます。



この方法の利点は、データ・バインディングが Web サービス・アシスタントによって生成されたため、CICS では Web サービスは WEBSERVICE リソースによって表されることです。CICS で統計、リソース管理、およびその他の機能を使用して、Web サービスを管理できます。欠点は、Java 開発者が、慣れていないプログラミング言語の言語構造を処理しなければならないことです。

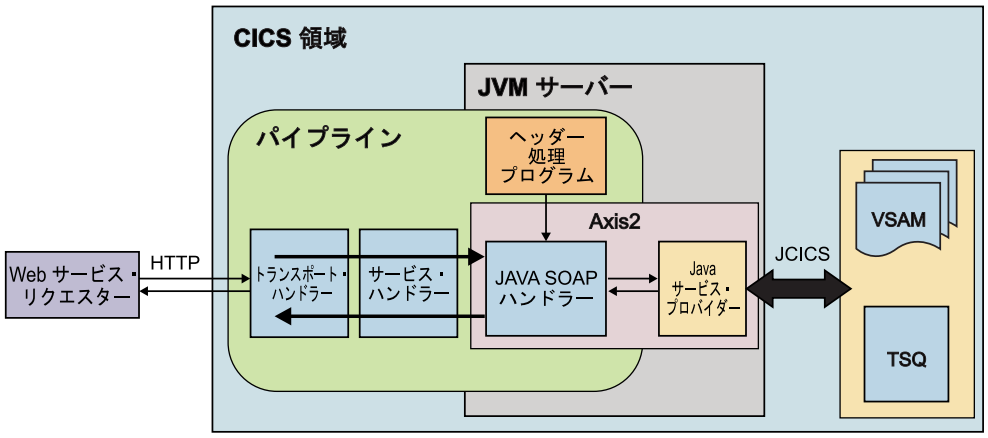
このタイプのアプリケーション用の環境を計画する場合は、アプリケーションを実行するために別個の JVM サーバーを使用してください。

- さまざまなワークロードに合わせて JVM サーバーをより効率よく管理し、調整することができます。
- インバウンド要求および **EXEC CICS LINK** でワークロード管理を使用して、ワークロードのバランスを取り、環境を拡大することができます。
- CICS における OSGi サポートを利用して、Java アプリケーションを管理できます。

Java データ・バインディングを使用する Java アプリケーションの作成

SOAP メッセージの XML を生成し、解析する Java アプリケーションを作成できます。Java 6 API は、XML を処理するために標準の Java ライブラリーを提供します。例えば、Java Architecture for XML Binding (JAXB) を使用して、Java データ・バインディングを作成し、Java API for XML Web Services (JAX-WS) ライブラリーを使用して、XML を生成し、解析することができます。これらのライブラリーを使用すると、アプリケーションは、SOAP パイプライン処理と同じ JVM サーバーの Axis2 で実行できます。

次の図に示されているシナリオは、Web サービス・プロバイダーであり、JVM サーバー内の Axis2 SOAP エンジンによって処理される Java アプリケーションの例です。



Java アプリケーションは Java データ・バインディングを使用し、Java SOAP ハンドラーと対話するので、アプリケーション・ハンドラーはありません。この例では、Web サービス・リクエスターは HTTP を使用して CICS 領域に接続しますが、JMS も使用することができます。Java アプリケーションは JCICS を使用して CICS サービス (この例では VSAM ファイルと一時記憶域キュー) にアクセスします。

この方法の利点は、Java 開発者が使い慣れたテクノロジーを使用してアプリケーションを作成することです。また、Java 開発者は、Web サービス・アシスタントが処理できない複雑な WSDL 文書を使用して、バインディングを作成できます。ただし、この方法には、次のようないくつかの制限があります。

- このタイプのアプリケーションには WS-Security を使用できません。したがって、セキュリティーを使用したい場合は、SSL を使用して接続を保護してください。
- パイプライン処理でユーザー ID のコンテキスト切り替えが行われません。要求でユーザー ID を変更するには、URIMAP リソースを使用してください。
- Web サービス・アシスタントからの Web サービス・バインディングを使用しないため、WEBSERVICE リソースがありません。
- アプリケーションが Web サービス・リクエスターである場合、パイプライン処理はバイパスされます。したがって、パイプラインで使用可能なサービス品質が得られません。

CICS 領域にワークロード管理を実装する場合は、このタイプのワークロードの経路指定方法を計画する必要があります。Java アプリケーションは SOAP 処理と同じ JVM サーバーで実行されるため、CICS は経路指定を行う機会を提供しません。ただし、経路指定が必要な場合は、JAX-WS アプリケーションで他のプログラムに対して分散プログラム・リンクを実装できます。

OSGi に準拠する Java アプリケーション

CICS には、JVM サーバーで OSGi 仕様に従う Java アプリケーションを実行するために、OSGi フレームワークの Equinox 実装環境が組み込まれています。

OSGi Service Platform 仕様は、 2 ページの『OSGi Service Platform』 に記述されているように、動的なモジュラー Java アプリケーションを実行し、管理するためのフレームワークを提供します。JVM サーバーのデフォルト構成には、OSGi フレームワークの Equinox 実装環境が組み込まれています。JVM サーバーの OSGi フレームワークにデプロイされる Java アプリケーションは、OSGi を使用する利点、および CICS におけるアプリケーションの実行に固有のサービス品質が得られます。

次のいずれかの理由で Java アプリケーションを使用できます。

- トランザクションのコストを削減するために、zAAP で実行できる Java ワークロードを作成したい。
- 他のプラットフォームで OSGi を使用する Java アプリケーションを作成した経験があり、CICS で Java アプリケーションを作成したい。
- アプリケーションおよびそれらのアプリケーションが実行される JVM の可用性に影響を与えることなく、独立して再使用し、更新できる 1 組のモジュラー・コンポーネントとして、Java アプリケーションを提供したい。

OSGi に準拠する Java アプリケーションを効率よく開発し、デプロイし、管理するには、CICS Explorer SDK と CICS Explorer を使用する必要があります。

- CICS Explorer SDK は、既存の Eclipse 統合開発環境 (IDE) を拡張して、Java 開発者が CICS における Java アプリケーションを作成し、デプロイするのに役立つツールとサポートを提供します。既存の Java アプリケーションを OSGi バンドルに変換するには、このツールを使用します。
- CICS Explorer は、OSGi バンドルと OSGi サービス、およびそれらが実行される JVM サーバーのビューをシステム管理者に提供する、Eclipse ベースのシステム管理者ツールです。Java アプリケーションの使用可能化と使用不可化、フレームワーク内の OSGi バンドルとサービスの状況の確認、および JVM サーバーのパフォーマンスに関する予備統計の取得に、このツールを使用します。

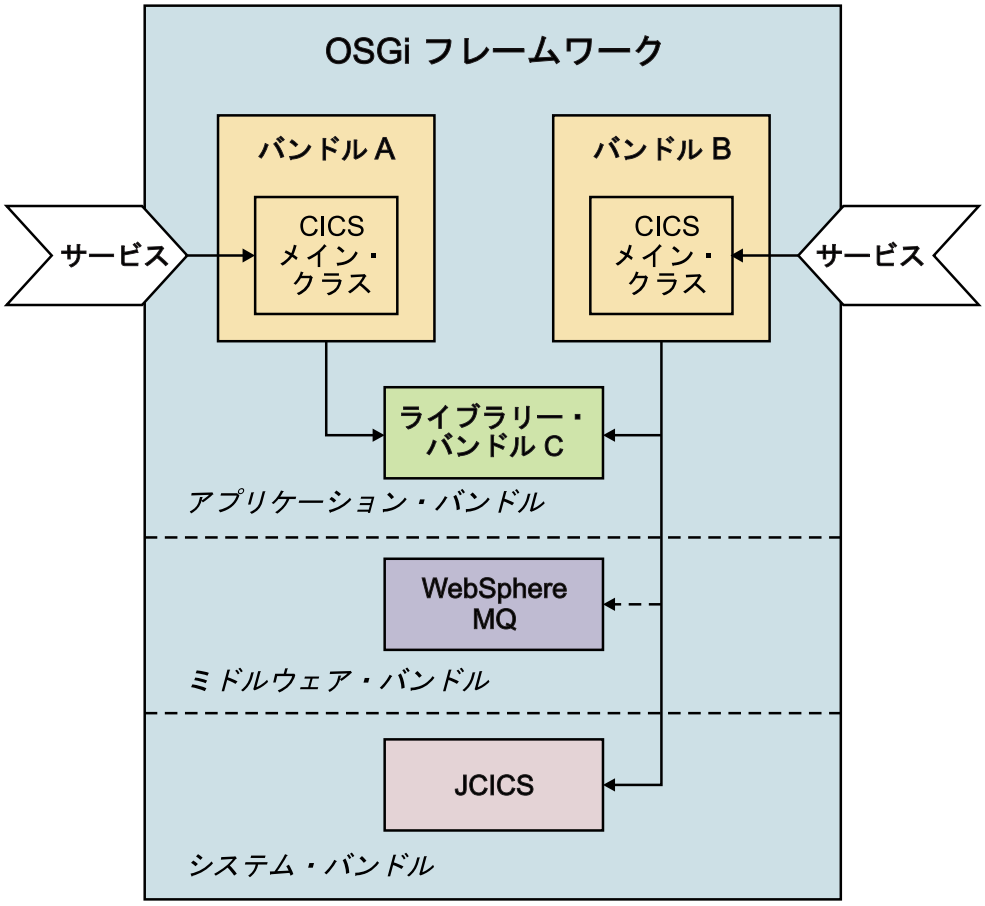
OSGi を使用する Java 開発者またはシステム管理者には、これらの自由に使用できるツールへのアクセス権限が必要です。

以下の例では、CICS において OSGi を使用する Java アプリケーションの実行方法を説明しています。

同一 JVM サーバー内の複数の Java アプリケーションの実行

JVM サーバーは、同一 JVM で複数の要求を同時に処理できます。したがって、同一アプリケーションを同時に複数回呼び出したり、同一 JVM サーバーで複数のアプリケーションを実行したりすることができます。

JVM サーバー間でアプリケーションを分割する方法を決定したら、OSGi モデルを使用してアプリケーションを 1 組の OSGi バンドルにコンポーネント化する方法を計画できます。また、アプリケーションにサービスを提供するために、フレームワークで必要な対応 OSGi バンドルを決定することも必要です。次の図に示されているように、OSGi フレームワークには、異なるタイプの OSGi バンドルを含むことができます。



アプリケーション・バンドル

アプリケーション・バンドルは、アプリケーション・コードを含む OSGi バンドルです。OSGi バンドルは、自己完結型であるか、フレームワーク内の他のバンドルに依存する場合があります。これらの依存関係はフレームワークによって管理され、未解決の依存関係がある OSGi バンドルはフレームワークで実行できません。CICS においてフレームワークの外部でアプリケーションにアクセスできるようにするために、OSGi バンドルは、CICS メイン・クラスを OSGi サービスとして宣言する必要があります。PROGRAM リソースが CICS メイン・クラスを指し示す場合、OSGi フレームワーク外部の他のアプリケーションは Java アプリケーションにアクセスできます。 1 つ以上のアプリケーションの共通ライブラリーを含む OSGi バンドルがある場合、Java 開発者は、CICS メイン・クラスを宣言しないことを決定することがあります。この OSGi バンドルは、フレームワーク内の他の OSGi バンドルからのみ使用可能です。

Java アプリケーションのデプロイメント単位は、CICS バンドルです。CICS バンドルは、任意の数の OSGi バンドルを含むことができ、1 つ以上の JVM サーバーにデプロイできます。JVM サーバーの管理とは無関係に、アプリケーション・バンドルを追加、更新、および除去することができます。

ミドルウェア・バンドル

ミドルウェア・バンドルは、WebSphere MQ への接続などのシステム・サービスを実装するためのクラスを含む OSGi バンドルです。もう 1 つの例は、ネイティブ・コードを含み、OSGi フレームワークに 1 回だけロードする必要がある OSGi バンドルです。ミドルウェア・バンドルは、クラスを使用するアプリケーションではなく、JVM サーバーのライフサイクルで管理されます。ミドルウェア・バンドルは、JVM サーバーの JVM プロファイルで指定され、JVM サーバーの開始時に CICS によってロードされます。

システム・バンドル

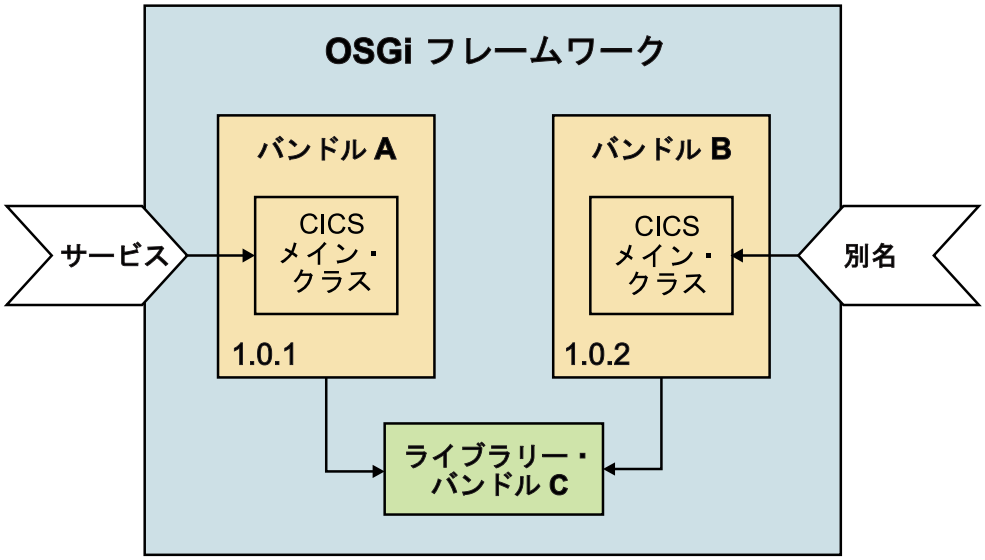
システム・バンドルは、アプリケーションに主なサービスを提供するために CICS と OSGi フレームワーク間の対話を管理する OSGi バンドルです。主な例は、CICS サービスとリソースにアクセスできるようにする JCICS OSGi バンドルです。

Java アプリケーションの管理をシンプルにするために、以下のベスト・プラクティスに従ってください。

- アプリケーションを構成する密結合 OSGi バンドルを同一 CICS バンドルにデプロイします。密結合バンドルは、OSGi サービスを使用することなく、相互にクラスを直接エクスポートします。これらの OSGi バンドルをまとめて 1 つの CICS バンドルにデプロイして、一緒に更新し、管理します。
- アプリケーション間に依存関係が生じないようにします。代わりに、共通ライブラリーを別個の OSGi バンドルで作成し、独自の CICS バンドルで管理します。アプリケーションとは別個にライブラリーを更新できます。
- バンドル間の依存関係を作成する場合、バージョンを使用して OSGi ベスト・プラクティスに従います。バージョンの範囲を使用すると、アプリケーションは、依存するバンドルへの両立可能な更新を許容できます。
- JVM サーバーの命名規則をセットアップし、システム・プログラマーと Java 開発者間でこの規則を合意します。

JVM サーバーにおける同一 Java アプリケーションの複数のバージョンの実行

OSGi フレームワークは、フレームワーク内で OSGi バンドルの複数バージョンの実行をサポートします。したがって、アプリケーションの使用可能状態を中断することなく、アプリケーションに段階的に更新を導入できます。ただし、フレームワーク内で同一 OSGi サービスの複数のバージョンを使用することはできません。OSGi バンドルの別々のバージョンに同じ CICS メイン・クラスがある場合、別名を使用して重複サービスをオーバーライドすることができます。別名は、CICS メイン・クラスの宣言で指定され、更新されたバージョンのバンドル用の OSGi サービスとして OSGi フレームワークで登録されます。別の PROGRAM リソースで別名を指定して、アプリケーションを使用可能にします。


OSGi フレームワーク
バンドル A
CICS
メイン・
クラス
1.0.1
バンドル B
CICS
メイン・
クラス
1.0.2
サービス
別名
ライブラリー・
バンドル C

第 3 章 CICS 用の Java アプリケーションの開発

CICS サービスを使用し、CICS 制御下で実行される Java アプリケーション・プログラムを作成することができます。CICS Explorer SDK を使用すると、JCICS クラス・ライブラリーを使用して CICS リソースにアクセスし、他の言語で作成されるプログラムと対話するアプリケーションを開発できます。また、さまざまなプロトコルやテクノロジー (Web サービスや CICS Transaction Gateway など) を使用して、Java プログラムに接続することもできます。

CICS は、Java アプリケーションをサポートするためのツールやランタイム環境を提供します。CICS Explorer SDK は Eclipse ベースのツールであり、CICS で Java アプリケーションを開発し、デプロイするためのサポートを提供します。CICS リソースやサービスにアクセスするアプリケーションを開発するための JCICS クラス・ライブラリーが含まれています。例えば、VSAM ファイル、一時データ・キュー、および一時記憶にアクセスできます。また、JCICS を使用して、他の言語 (COBOL や C など) で作成された CICS アプリケーションにリンクすることもできます。

CICS Explorer SDK は、その他の機能も提供します。例えば、OSGi 仕様に従うためのアプリケーションのパッケージや、CICS の特定のリリースでサポートされるクラスのみを使用することを確実にするためのターゲット環境の提供などです。また、CICS 用の Java アプリケーションの開発に慣れていない場合に役立つように、JCICS サンプルも含まれています。

CICS について必要な知識

CICS はトランザクション処理サブシステムであり、ユーザーが要求に応じてアプリケーションを実行するためのサービスを提供します。これは、他の複数のユーザーが同じファイルとプログラムを使用して同じアプリケーションを実行する要求を実行依頼するのと同時に行われます。CICS は、リソースの共用、データの保全性、実行の優先順位付けを管理する一方で、短い応答時間を維持します。

CICS アプリケーションは、製品オーダーの処理や会社の給与計算の準備などの業務を連携して実行する、関連したプログラムの集合です。CICS アプリケーションは、CICS 制御下で実行され、CICS サービスとインターフェースを使用してプログラムとファイルにアクセスします。

CICS アプリケーションを実行するには、*トランザクション* 要求を実行依頼します。 CICS では、トランザクションという用語に特別な意味があります。CICS での意味と、業界でより一般的に使用される意味との違いについては、32 ページの『CICS トランザクション』を参照してください。トランザクションの実行は、必要な機能を実装する 1 つ以上のアプリケーション・プログラムの実行で構成されます。

CICS 用の Java アプリケーションを開発するには、CICS プログラム、トランザクション、およびタスク間の関係を理解する必要があります。これらの用語は、CICS

資料全体で使用され、多くのプログラミング・コマンドで表示されます。また、ランタイム環境において CICS が Java アプリケーションを処理する方法も理解しておく必要があります。

CICS トランザクション

トランザクションは、単一の要求によって開始される 1 つの処理です。

要求は通常、ユーザーによって端末で行われます。ただし、Web ページから、リモート・ワークステーション・プログラムから、または別の CICS 領域のアプリケーションから行われる場合があります。もしくは、事前定義された時点に自動的にトリガーされる場合もあります。「インターネット・ガイド」の『概要: CICS および HTTP』 および External Interfaces Guide の外部インターフェースの概要 では、CICS トランザクションのさまざまな実行方法が記述されています。

単一トランザクションは、1 つ以上のアプリケーション・プログラムで構成されます。これらのアプリケーション・プログラムが実行されると、必要な処理を実行します。

ただし、CICS ではトランザクションという用語は、単一イベントと、同じタイプの他のすべてのトランザクションの両方を意味するのに使用されます。CICS に対し、それぞれのトランザクション・タイプを、TRANSACTION リソース定義を使用して記述します。この定義により、トランザクション・タイプに名前 (トランザクション ID、すなわち TRANSID) が指定され、実行される作業に関する複数の項目が CICS に指示されます。例えば、最初にどのプログラムを呼び出すか、トランザクションの実行全体でどの種類の認証が必要であるかなどです。

トランザクションを実行するには、その TRANSID を CICS に対して送信します。CICS は、TRANSACTION 定義に記録された情報を使用して正しい実行環境を確立し、最初のプログラムを開始します。

トランザクションという用語は、*リカバリー単位*、または CICS で*作業単位*と呼ばれるものを記述するために、IT 業界で広く使用されています。一般に、これはリカバリー可能な完全な論理オペレーションです。プログラムされたコマンド、またはシステム障害の発生によって、トランザクション全体をコミットまたはバックアウトすることができます。多くの場合、CICS トランザクションの有効範囲は単一の作業単位でもありますが、CICS 資料を読むときは、意味の違いを認識する必要があります。

CICS タスク

タスクは、トランザクションを実行する単一のインスタンスです。

CICS では、このタスク という用語に特別の意味があります。CICS はトランザクションの実行要求を受け取ると、トランザクション・タイプの実行のこの *1* つのインスタンス に関連した新規タスクを開始します。すなわち、CICS タスクは、通常は特定のユーザーのために、データの独自の専用セットを使用した、トランザクションの 1 つの実行です。また、タスクを*スレッド* と見なすこともできます。タスクは、優先順位と準備度にしたがって CICS によって*ディスパッチ* されます。トランザクションが完了すると、タスクは終了します。

CICS アプリケーション・プログラム

Java プログラムでは、CICS 用の Java クラス・ライブラリー (JCICS) を使用して CICS サービスにアクセスし、他の言語で作成されたアプリケーション・プログラムにリンクすることができます。

CICS アプリケーション・プログラムは COBOL、C、C++、Java、PL/I、またはアセンブラー言語で作成できます。大部分の処理ロジックは標準言語ステートメントで表されますが、CICS サービスを要求するには、アプリケーションは提供されたアプリケーション・プログラミング・インターフェースを使用します。
COBOL、C、C++、PL/I、またはアセンブラー・プログラムは、**EXEC CICS** アプリケーション・プログラミング・インターフェースまたは C++ クラス・ライブラリーを使用できます。Java プログラムは JCICS クラス・ライブラリーを使用します。
JCICS については、57 ページの『CICS 用 Java クラス・ライブラリー (JCICS)』で説明しています。

CICS サービス

Java プログラムは、JCICS プログラミング・インターフェースを介して、データ管理サービス、通信サービス、作業単位サービス、プログラム・サービス、および診断サービスの各 CICS サービスにアクセスできます。

CICS サービス・マネージャーの名称には、通常は「管理」または「制御」という語が含まれています (例えば、「端末管理」、「プログラム制御」など)。これらの用語は、CICS 資料で幅広く使用されています。

データ管理サービス

CICS が提供するデータ管理サービスは、次のとおりです。

- 仮想記憶アクセス方式 (VSAM) データ・セットにアクセスする際の、保全性のあるレコード・レベル共用。データのバックアウト (トランザクション障害またはシステム障害の場合)、または順方向リカバリー (メディア障害の場合) をサポートするために、CICS はアクティビティーをログに記録します。 CICS ファイル制御は、VSAM データを管理します。

  また、CICS は 2 つの専有ファイル構造も実装し、それらを操作するためのコマンドを提供します。

  一時記憶

  一時記憶 (TS) は、複数のトランザクションからデータを容易に使用可能にする手段です。データは、プログラムからの要求に応じて作成されるキューに保持されます。キューには順次にアクセスするか、または項目番号でアクセスすることができます。

  一時記憶域キューは、メインメモリーに常駐することも、ストレージ・デバイスに書き込むこともできます。

  一時記憶域キューは、名前付きのスクラッチパッドと見なすことができます。

  一時データ

  一時データ (TD) も複数のトランザクションから使用可能であり、キュー

に保持されます。ただし、TS キューとは異なり、TD キューは事前定義する必要があり、順次にしか読み取れません。各項目は、読み取られるとキューから除去されます。

一時データ・キューは、常にデータ・セットに書き込まれます。一時データ・キューは、特定数の項目が書き込まれると、特定トランザクションを開始するトリガーの役目をするように定義できます 例えば、起動したトランザクションによってそのキューを処理できます。

- データベース製品とのインターフェースを使用した、他のデータベース (DB2 を含む) 内のデータへのアクセス。

通信サービス

CICS は、SNA と TCP/IP プロトコルを使用して、さまざまな端末 (ディスプレイ、プリンター、およびワークステーション) へのアクセスを可能にするコマンドを備えています。CICS 端末管理により、SNA ネットワークおよび TCP/IP ネットワークを管理できます。

拡張プログラム間通信機能 (APPC) コマンドを使用して、SNA プロトコルを使用してリモート・システム内の他のプログラムを開始し、通信するプログラムを作成できます。CICS APPC は、ピアツーピア分散アプリケーション・モデルを実装します。

また、CICS は、共通オブジェクト・リクエスト・ブローカー・アーキテクチャー (CORBA) によって定義されるインバウンドおよびアウトバウンド IIOP プロトコルを実装するオブジェクト・リクエスト・ブローカー (ORB) も提供します。ORB は、Java ステートレス・オブジェクトおよびエンタープライズ Bean を実行する要求をサポートします。

次の CICS 専有通信サービスが提供されます。

機能シップ
リモート CICS 領域で既存のものとして定義されるリソース (ファイル、キュー、およびプログラム) にアクセスするプログラム要求は、自動的に CICS によって専有領域に転送されます。

分散プログラム・リンク (DPL)
リモート CICS 領域で既存のものとして定義されるプログラムに対するプログラム・リンク要求は、自動的に専有領域に転送されます。CICS は、分散アプリケーションの保全性を維持するためのコマンドを提供します。

非同期処理
CICS は、プログラムが同じ CICS 領域またはリモート CICS 領域内の別のトランザクションを開始し、オプションとしてデータをそのトランザクションに渡すことを可能にするコマンドを提供します。新しいトランザクションは、新しいタスク内で独立してスケジュールされます。この機能は、他のソフトウェア製品によって提供される *fork* 操作に似ています。

トランザクション・ルーティング
リモート CICS 領域で既存のものとして定義されるトランザクションを実行する要求は、自動的に専有領域に転送されます。ユーザーへの応答は、要求を受け取った領域に返されます。

作業単位サービス

CICS がトランザクションを実行する新しいタスクを作成すると、新しい作業単位 (UOW) が自動的に開始されます (したがって、BEGIN コマンドは必要ないため、CICS はこのコマンドを提供しません)。CICS トランザクションは常にトランザクション内で実行されます。

CICS は、実行されたリカバリー可能な作業をコミットまたはロールバックするために SYNCPOINT コマンドを提供します。同期点が完了すると、CICS は自動的に別の作業単位を開始します。SYNCPOINT コマンドを発行せずにプログラムを終了すると、CICS は暗黙的な同期点を取り、トランザクションをコミットしようとします。

コミットの有効範囲には、リカバリー可能として定義されたすべての CICS リソース、および CICS によって提供されたインターフェースを使用してインタレストを登録した他のすべてのリソース・マネージャーが含まれます。

Java Transaction Service (JTS) によって定義されたコマンドが提供するトランザクション・サービスを使用してエンタープライズ Bean を作成する場合、これらのコマンド (BEGIN を含む) は、CICS によってその作業単位サービスにマップされます。

プログラム・サービス

CICS は、プログラムが別のプログラムにリンクするか、制御を転送してから戻ることを可能にするコマンドを提供します。

診断サービス

CICS が提供するコマンドを使用して、プログラムをトレースし、ダンプを作成できます。

CICS における Java ランタイム環境

CICS は、Java アプリケーションを実行するための 2 つのランタイム環境を提供します。スレッド・セーフ・アプリケーションは JVM サーバーを使用できます。スレッド・セーフでないアプリケーションは、プールされた JVM を使用する必要があります。

JVM サーバー

JVM サーバーは、単一の JVM で複数のタスクを実行できるランタイム環境です。Java アプリケーションを実行する場合は、この環境が推奨されます。これは、Java タスクごとに必要な仮想ストレージを減らし、CICS が同時に複数のタスクを実行できるようにするからです。

CICS タスクは、同じ JVM サーバー・プロセス内のスレッドとして並列で実行されます。複数のアプリケーションを同時に実行するすべての CICS タスクによって JVM が共用されるだけでなく、すべての静的データと静的クラスも共用されます。したがって、CICS で JVM サーバーを使用するには、Java アプリケーションはスレッド・セーフでなければなりません。各スレッドは T8 TCB で実行され、JCICS API を使用して CICS サービスにアクセスできます。

新しいスレッドを開始したり、スレッドを開始するライブラリーを呼び出すためにアプリケーション・コードを作成できます。ただし、これらのスレッドは CICS サービスにアクセスできません。アプリケーションで spawn されたスレッドから CICS サービスにアクセスしようとすると、Java bm.exception が生じます。ご使用のアプリケーションでスレッドを作成したい場合は、そのアプリケーションを実行する CICS タスクの存続期間を超えてスレッドが実行されないようにしてください。システム・プログラマーが JVM サーバーを使用不可にする場合、CICS は、T8 TCB で実行中の現行のすべてのスレッドが JVM で終了するまで待ちます。ただし、アプリケーション自体によって作成されたスレッドは終了します。

静的データは JVM サーバーで実行中のすべてのスレッドで共用されるため、静的データを初期化し、JVM のシャットダウン時に正しい状態のままにしておくための OSGi バンドル・アクティベーター・クラスを作成できます。 JVM サーバーは、例えば、アプリケーションを追加したり、問題を修正したりするために、システム・プログラマーが使用不可にするまで実行されます。バンドル・アクティベーター・クラスを提供することによって、ご使用のアプリケーションに状態が正しく設定されていることを確認できます。 CICS にはタイムアウトがあります。このタイムアウトは、JVM サーバーの開始または停止を続行するまでにこれらのクラスが完了するのを待機する時間を指定します。始動クラスおよび終了クラスで JCICS を使用することはできません。

ご使用のアプリケーションで System.exit() メソッドを使用しないでください。このメソッドにより、JVM サーバーと CICS の両方がシャットダウンし、アプリケーションの状態と可用性に影響を与えます。

プールされた JVM

プールされた JVM は、Java アプリケーションに対して一度に 1 つの要求のみを処理できます。したがって、CICS 領域にはより多くの JVM が必要です。プールされた JVM は分離されるので、Java アプリケーションはスレッド・セーフである必要はありません。ただし、プールされた JVM は通常、おそらく別々のアプリケーションによって複数回再利用されます。そのため、トランザクションの分離とデータの状態を維持することが重要です。

JVM の開始に使用されるメイン・スレッドは、初期プロセス・スレッド (IPT) と呼ばれます。CICS は、任意の Java プログラムの public static main メソッドが、プールされた JVM の IPT で実行されるようにします。ご使用のアプリケーションでスレッドを作成したい場合は、それらのスレッドは CICS サービスにアクセスしようとしてはなりません。また、それらのスレッドを開始する CICS タスクの存続期間を超えて実行されてはなりません。IPT が CICS に制御を戻した後、ユーザー・スレッドが引き続き実行される場合、別のアプリケーションによって再利用されるとこれらのスレッドは JVM の分離を損ない、CICS が JVM を停止しようとすると問題が生じる可能性があります。

CICS Explorer SDK のインストール

CICS Explorer SDK は、IBM Web サイトから無料でダウンロードして、Eclipse 統合開発環境 (IDE) にインストールできます。

始める前に

ご使用のワークステーションに必須ソフトウェアがインストールされていなければなりません。オペレーティング・システムおよびソフトウェアのリストは、CICS Explorer Web サイトに記述されています。

このタスクについて

CICS Explorer SDK は、CICS Explorer の拡張機能を開発するための、Eclipse ベースのフレームワークです。また、任意のサポートされている CICS リリースで実行する Java アプリケーションの開発もサポートします。OSGi 仕様に従うための JCICS およびアプリケーションのパッケージをサポートします。

手順

1. CICS Explorer Web サイトに進みます。
2. 「**Download site**」リンクを選択し、IBM ID およびパスワードを入力します。
3. リストの中から「CICS Explorer」を選択し、「**Continue**」をクリックします。
4. ライセンスを読み、受諾します。
5. リストの中から「CICS Explorer SDK」を選択して、ワークステーション上のディレクトリーに圧縮ファイルをダウンロードします。
6. Eclipse IDE をオープンし、「**Help**」 > 「**Install new software**」をクリックします。
7. 「**Add**」をクリックします。「Add site」ダイアログ・ボックスで、「**Archive**」をクリックします。
8. ダウンロードされたファイルを参照し、「**Open**」をクリックします。
9. IBM CICS Explorer SDK の横のチェック・ボックスを選択し、「**Next**」をクリックします。
10. 使用許諾に同意し、「**Finish**」をクリックして CICS Explorer SDK をインストールします。

タスクの結果

CICS Explorer SDK は、Eclipse IDE にインストールされました。セキュリティー警告を受け入れ、IDE を再始動して、新規ソフトウェアを取り出してください。

次のタスク

CICS Explorer SDK によって提供される CICS サンプルを使用して、Java サポートを十分に理解することができます。詳しくは、『JCICS サンプルの概要』を参照してください。

JCICS サンプルの概要

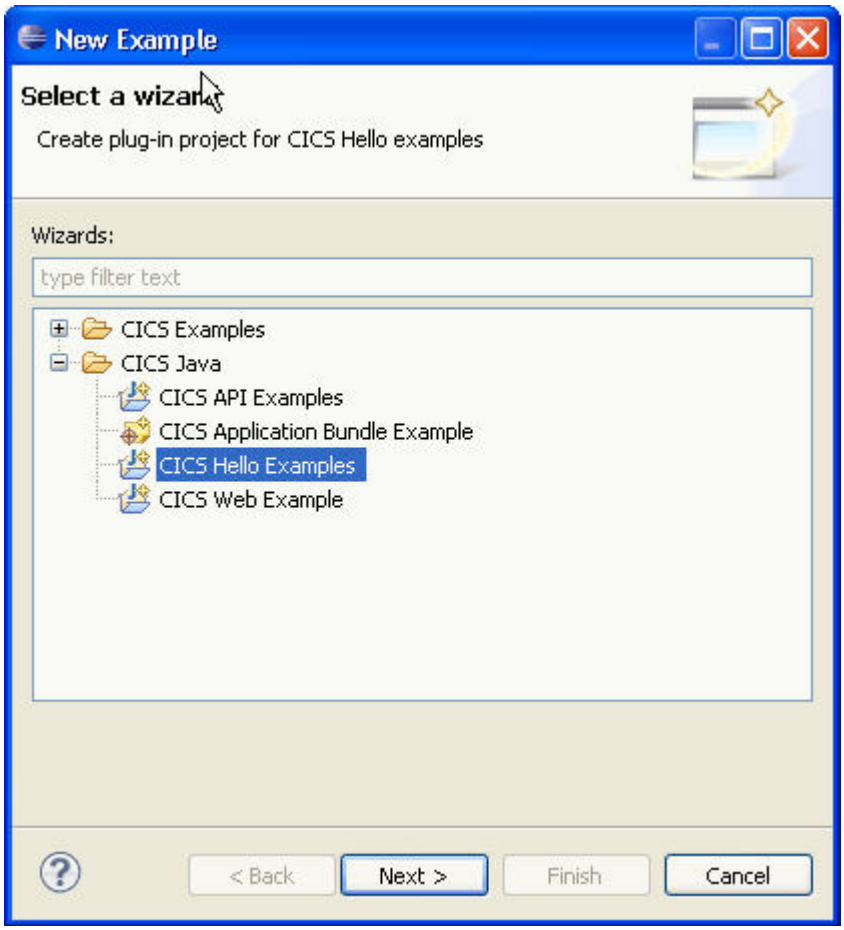
CICS Explorer SDK には、CICS 用の Java アプリケーションの開発を開始するのに役立つ JCICS サンプルが含まれています。

このタスクについて

JCICS サンプルは 1 組の OSGi バンドルとしてパッケージされます。このバンドルを Eclipse プラグイン・プロジェクトにインポートすると、Java ソース・コードを表示できます。また、コンテキスト・ヘルプを使用して、コードで使用されるメソッドの Javadoc の説明を検索することもできます。

手順

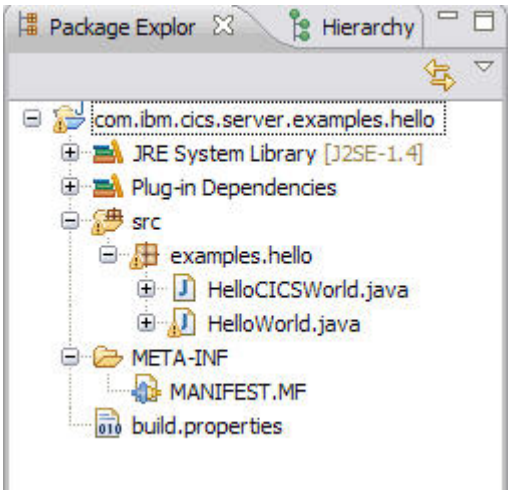
1. Eclipse IDE で、Java パースペクティブを開きます。
2. 新しいプラグイン・プロジェクト・サンプルを作成するために、次のいずれかの選択項目を使用して、「New Example」ウィザードを開きます。
   - Eclipse メニュー・バーで、「**File**」 > 「**New**」 > 「**Example**」をクリックします。
   - 「**New Wizard**」アイコンで下矢印をクリックし、「**Example**」をクリックします。
   - 「Project Explorer」ビューで右クリックし、「**New**」 > 「**Example**」をクリックします。
3. 「**CICS Java**」フォルダーで、「**CICS Hello Examples**」を選択し、「**Next**」をクリックします。



   - 「CICS API Examples」は、一時データ・キュー、一時記憶域キュー、およびチャネルと COMMAREA を Java プログラムで使用する方法を示します。

- 「CICS Application Bundle Example」は、CICS にデプロイするための CICS バンドルの作成方法を示します。
- 「CICS Hello Examples」は、CICS においてシンプルな Hello World テストを行う 2 とおりの方法を示します。
- 「CICS Web Example」は、クラスを使用して Web ブラウザーと対話する方法を示します。

4. 「**Project name**」フィールドに、新規プロジェクトの名前を入力します。 デフォルトで Eclipse が作成する名前は、ワークスペース内のサンプルのフォルダー位置の後に、サンプル名が続いたものです。例えば、Hello World サンプルのデフォルトのプロジェクト名は `com.ibm.cics.server.examples.hello` です。
5. 「**Finish**」をクリックします。 Eclipse は、JCICS Hello World サンプルを OSGi バンドルとして含むプラグイン・プロジェクトを作成します。
6. 「Package Explorer」ビューでプロジェクトを展開します。



- **Plug-in Dependencies** フォルダーには、OSGi バンドルの依存関係が入っています。この例では、バンドルは、JCICS を含む OSGi バンドルに依存します。この情報は、プロジェクトのマニフェストにも取り込まれます。
- **src** フォルダーには、サンプルの Java ソースが入っています。ソース・ファイルをブラウズして、使用されている JCICS クラスを確認し、コンテキスト・ヘルプを使用して特定のクラスを検索することができます。また、Javadoc ビューを開いて、選択された内容 (例えば、メソッドやクラス) について API の詳細を確認することもできます。
- **META-INF** フォルダーには、プロジェクトのマニフェストが入っています。このマニフェストには、OSGi バンドルを記述する OSGi ヘッダーが入っています。

7. 「New Example」ウィザードを使用して、CICS API サンプルおよび CICS Web サンプルのプラグイン・プロジェクトを作成します。 Java ソースを表示すると、JCICS クラスがプログラムおよび Web アプリケーションとの連携に使用される方法を理解できます。

タスクの結果

Eclipse で JCICS サンプル用に 3 つのプラグイン・プロジェクトを作成しました。これらのプロジェクトには、プラグイン依存関係やターゲット Java 環境を含む、OSGi バンドルのパッケージ情報が入っています。

次のタスク

CICS で Java アプリケーションを実行するには、CICS バンドル・プロジェクト内の Java アプリケーションを zFS にデプロイする必要があります。『JCICS サンプルのデプロイ』で説明されているように、JCICS サンプルを使用してデプロイメント・プロセスを試行できます。

JCICS サンプルのデプロイ

CICS Explorer SDK の CICS バンドル・サンプルを使用すると、JCICS サンプルを CICS 領域にデプロイできます。

始める前に

37 ページの『JCICS サンプルの概要』で説明されているとおりに、JCICS サンプル・プロジェクトを作成済みでなければなりません。

このタスクについて

CICS は、zFS から Java アプリケーションをロードし、実行します。したがって、zFS 内の適切なディレクトリーにコンパイル済みアプリケーションをデプロイする必要があります。 CICS Explorer 内で z/OS パースペクティブを使用して、zFS に適切なディレクトリーを作成できます。 CICS には、このディレクトリーへの読み取りおよび実行アクセス権限が必要です。

CICS Explorer SDK は、CICS バンドル・プロジェクト内の Java アプリケーションを zFS にデプロイするためのサポートを提供します。 CICS バンドル・プロジェクトは、単一の単位として論理的にデプロイされ、管理される 1 組の OSGi バンドルをまとめてグループ化します。CICS バンドル・プロジェクト・サンプルを使用すると、JCICS サンプルを CICS 領域にデプロイできます。

手順

1. Eclipse IDE で、Java パースペクティブを開きます。
2. 次のいずれかの選択項目を使用して、「New Example」ウィザードを開きます。
   - Eclipse メニュー・バーで、「**File**」 > 「**New**」 > 「**Example**」をクリックします。
   - 「**New Wizard**」アイコンで下矢印をクリックし、「**Example**」をクリックします。
   - 「Project Explorer」ビューで右クリックし、「**New**」 > 「**Example**」をクリックします。
3. 「**CICS Java**」フォルダーで、「**CICS Application Bundle Example**」を選択し、「**Next**」をクリックします。

4. 「**Project name**」フィールドに、新規プロジェクトの名前を入力します。 デフォルトで Eclipse が作成する名前は、ワークスペース内のサンプルのフォルダー位置の後に、サンプル名が続いたものです。例えば、CICS バンドルのデフォルトのプロジェクト名は `com.ibm.cics.server.examples` です。
5. 「**Finish**」をクリックします。 Eclipse は、マニフェストと 3 つのリソースを含む CICS バンドル・プロジェクトを作成します。これらのリソースは 3 つの OSGi バンドルを参照します。
6. `web.osgibundle` ファイルを開いてその内容を確認します。 このファイルは XML 形式であり、OSGi バンドルのシンボル名とバージョンを含みます。また、サンプル JVM サーバーの名前も入っています。JVM サーバーは、CICS における Java アプリケーション用のランタイム環境です。独自のアプリケーションを作成する場合は、このファイルでターゲット JVM サーバーの名前を提供する必要があります。
7. CICS バンドルを zFS にデプロイします。
   a. CICS バンドル・プロジェクトを右クリックして、「**Export to z/OS UNIX File System**」を選択します。
   b. 必要に応じて、FTP 資格情報を入力します。 以前に接続をセットアップしていなかった場合は、ターゲット・ホスト・マシンとの接続の作成が必要な場合があります。
   c. CICS バンドルをデプロイするディレクトリーまでブラウズして、「**Finish**」をクリックします。

   CICS バンドルは、指定のディレクトリーにデプロイされます。
8. CICS SM パースペクティブを開きます。CICSplex Explorer ビューで、JCICS サンプル・プログラムを実行する CICS 領域を選択します。
9. サンプル・グループ DFH$OSGI 内にある JVMSERVER リソース DFH$JVMS をインストールします。 このリソースは、OSGi フレームワークが入っている CICS 領域でサンプル JVM サーバーを作成します。このリソース名は、CICS バンドルのマニフェストで指定された JVM サーバーの名前と一致します。「**Operations**」 > 「**Java**」 > 「**JVM Servers**」をクリックすると、JVM サーバーの状況を確認できます。
10. 「**Definitions**」 > 「**Bundle Definitions**」をクリックして、「Bundle Definitions」ビューを開きます。 このビューは、CICS 領域のすべてのバンドル定義をリストします。
11. 「Resource Group Definitions」ビューで、提供された DFH$OSGI グループを選択します。 このビューが開いていない場合、「**Window**」 > 「**Show view**」を選択して、Eclipse パースペクティブでこのビューを開きます。 「Bundle Definitions」ビューがフィルターに掛けられて、DFH$OSGB リソース定義を表示します。
12. リソース定義を新規グループにコピーして属性を編集します。
    a. DFH$OSGB を右クリックして、「**Copy**」を選択します。
    b. 「Resource Group Definitions」ビューの任意の場所で右クリックして、「**Paste**」を選択します。
    c. 新規グループ名を入力し、「**OK**」をクリックします。

13. 新規グループで BUNDLE リソース定義を編集して、デプロイされた CICS バンドルの場所と一致するようにバンドル・ディレクトリーを変更します。
14. その定義を右クリックして、BUNDLE リソースをインストールします。「**Operations**」 > 「**Bundles**」をクリックすると、ENABLED 状態でインストールされている BUNDLE を確認できます。また、「**Operations**」 > 「**Java**」 > 「**OSGi Bundles**」をクリックして、OSGi バンドルのリストを確認することもできます。
15. JVM サーバーでこれらのサンプルを実行するために、CICS 領域に DFH$OSGI サンプル・グループをインストールします。 このグループには、すべてのサンプルのリソース定義が入っています。サンプルの BUNDLE および JVMSERVER リソースはインストールされません。同じ名前を持つリソースを既に作成しているからです。 グループをインストールすると、CICS は、サンプルの実行に必要なリソースを作成します。

タスクの結果

CICS バンドル・サンプルを正常に zFS にデプロイし、JCICS サンプルの実行に必要な CICS リソースを作成しました。

次のタスク

『JCICS サンプルの実行』で説明されているように、JCICS サンプルを実行できます。

JCICS サンプルの実行

CICS は、CICS で実行できる複数の JCICS サンプルを用意しています。Java アプリケーションを実行するための優先環境である JVM サーバーでこれらのサンプルを実行するか、プールされた JVM で実行することができます。

始める前に

JCICS サンプルは、CICS 領域が読み取りおよび実行アクセス権限を持つ zFS ディレクトリーにデプロイされなければなりません。

手順

1. CICS 領域が正しく構成されていることを確認します。
   - JVM サーバーでこれらのサンプルを実行したい場合は、DFH$OSGI グループが CICS 領域にインストールされなければなりません。特に DFH$JVMS リソースは、CICS 領域で有効でなければなりません。このリソースは、提供されたサンプル JVM サーバーであり、デフォルトの DFHOSGI プロファイルを使用します。 JCICS サンプル用の OSGi バンドルを含む BUNDLE リソースも有効でなければなりません。
   - プールされた JVM でこれらのサンプルを実行する場合は、DFH$JVM グループが CICS 領域にインストールされていることが必要です。デフォルトの JVM プロファイル DFHJVMPR を編集して、クラスパス `/usshome/samples/dfjcics` を `CLASSPATH_SUFFIX` オプションに追加してください。ここで、*usshome* は、**USSHOME** システム初期設定パラメーターの値です。

2. 該当する手順に従って各サンプルを実行します。

Hello World サンプルの実行

2 つの「Hello World」サンプル (HelloWorld および HelloCICSWorld) を実行できます。HelloWorld サンプルは、Java サービスのみを使用し、HelloCICSWorld サンプルは、JCICS TerminalPrincipalFacility クラスの使用を実証します。

始める前に

42 ページの『JCICS サンプルの実行』で説明されているとおりに、CICS 領域が構成されていることを確認します。

このタスクについて

プログラムは、サンプルの CICS トランザクションによって開始されます。これらのサンプルでは、次の Java クラスおよび CICS プログラムを使用します。

サンプル	トランザクション	Program	Java クラス
HelloWorld	JHE1	DFJ$JHE1	`HelloWorld`
		DFH$JSAM (C プログラム)	該当せず
HelloCICSWorld	JHE2	DFJ$JHE2	`HelloCICSWorld`

DFH$JSAM は標準の CICS プログラムであり、CICS 対応の任意の言語で作成できます。例えば、C コンパイラーがない場合、COBOL バージョンの DFH$JSAM を作成し、提供された C バージョンの代わりに使用することができます。または、JHE1 TRANSACTION 定義を変更して DFH$JSAM を完全にバイパスして、プログラム DFJ$JHE1 を実行することができます。ただし、定義を変更する場合、Java プログラムは端末に何も書き込みません。したがって、アプリケーションが正常に実行したことを示すものは、`stdout` ファイル内のメッセージのみです。

手順

- 端末で JHE1 トランザクションに入って、標準 Java アプリケーションを実行します。JHE1 から以下のメッセージが表示されます。 次のメッセージが端末に戻されます。

  ```
  SAMPLE *COMPLETED*, SEE STOUT
  ```

  次の項目が `stdout` ファイルに書き込まれます。

  ```
  Hello from a regular Java application
  ```

- 端末で JHE2 トランザクションに入って、JCICS アプリケーションを実行します。端末に JHE2 から以下のメッセージが表示されます。

  ```
  Hello from a Java CICS application
  ```

タスクの結果

これで Hello World サンプルを正常に実行しました。

次のタスク

他のサンプルを実行すると、CICS で Java プログラムから使用可能なさまざまなサービスを試用できます。

プログラム制御サンプルの実行

チャネルおよび COMMAREA のサンプルを実行すると、CICS がチャネルとコンテナーまたは COMMAREA を処理する方法を理解できます。プログラムはどちらかの方式を使用してデータを渡すことができますが、コンテナーは 32 KB に制限されません。

始める前に

42 ページの『JCICS サンプルの実行』で説明されているとおりに、CICS 領域が構成されていることを確認します。

このタスクについて

これらのサンプルは、JCICS `Program` クラスを使用して、チャネルとコンテナーまたは COMMAREA を別のプログラムに渡す方法を示します。COMMAREA サンプルは、Java コード内の ASCII 文字と、ネイティブ CICS プログラムで使用される等価な EBCDIC 文字との間で両方向に変換する方法も示します。

プログラムは、サンプルの CICS トランザクションによって開始されます。これらのサンプルでは、次の Java クラスおよび CICS プログラムを使用します。

サンプル	トランザクション	Program	Java クラス
Channel	JPC3	DFJ$JPC3	`ProgramControl .ClassThree`
		DFJ$JPC4	`ProgramControl .ClassFour`
		DFH$LCCC (C 言語)	該当せず
COMMAREA	JPC1	DFJ$JPC1	`ProgramControl .ClassOne`
		DFJ$JPC2	`ProgramControl .ClassTwo`
		DFH$LCCA (C 言語)	該当せず

DFH$LCCA および DFH$LCCC は標準の CICS プログラムであり、サポートされている任意の高水準言語で作成できます。 C コンパイラーがない場合、COBOL バージョンの DFH$LCCA および DFH$LCCC を作成し、提供された C バージョンの代わりに使用することができます。

手順

- チャネルのサンプルを実行するには、以下の手順を実行します。
  1. 端末で JPC3 トランザクションに入ります。 `Task.out` (通常は端末) に以下のメッセージが表示されます。

```
Entering ProgramControlClassThree.main()
About to link to C program
Leaving ProgramControlClassThree.main()
```

2. 画面をクリアしてください。 以下のメッセージが表示されます。

```
Entering ProgramControlClassFour.main()
ProgramControlClassFour invoked with Container "IntData         "
ProgramControlClassFour invoked with Container "StringData      "
ProgramControlClassFour invoked with Container "Response        "
Leaving ProgramControlClassFour.main()
```

コンテナーをリストするメッセージは、異なる順序で表示される場合があります。

CICS では次の処理が行われます。

1. トランザクションは、PROGRAM リソース DFJ$JPC3 で定義されるメイン Java クラスを実行します。Java プログラムは、2 つのコンテナーを持つ Channel オブジェクトを構成し、C プログラム DFH$LCCC にリンクします。
2. DFH$LCCC は、コンテナーを処理し、新しい応答コンテナーを作成して戻ります。
3. Java プログラムは、応答コンテナー内のデータを検査し、疑似会話型トランザクションの開始をスケジュールして、開始トランザクションに Channel オブジェクトを渡します。
4. 開始トランザクションは、PROGRAM リソース DFJ$JPC4 で定義される別の Java クラスを実行します。この Java プログラムは、`ContainerIterator` オブジェクトを使用して `Channel` をブラウズし、検出する各コンテナーの名前を表示します。

- COMMAREA のサンプルを実行するには、以下の手順を実行します。

1. JPC1 CICS トランザクションを入力して、この例を実行します。 `Task.out` (通常は端末) に以下のメッセージが表示されます。

```
Entering ProgramControlClassOne.main()
About to link to C program
Leaving ProgramControlClassOne.main()
```

2. 画面をクリアしてください。 以下のメッセージが表示されます。

```
Entering ProgramControlClassTwo.main()
data received correctly
Leaving ProgramControlClassTwo.main()
```

CICS では次の処理が行われます。

1. トランザクションは、PROGRAM リソース DFJ$JPC1 で定義されるメイン Java クラスを実行します。Java プログラムは、COMMAREA を構成し、C プログラム DFH$LCCA にリンクします。
2. この C プログラムは、COMMAREA を処理し、更新して、Java プログラムに戻ります。
3. Java プログラムは、COMMAREA 内のデータを検査し、疑似会話型トランザクションの開始をスケジュールして、開始トランザクションに COMMAREA 内の変更済みデータを渡します。
4. 開始トランザクションは、PROGRAM リソース DFJ$JPC2 で定義される別のメイン Java クラスを実行します。この Java プログラムは、COMMAREA を読み取り、再度検証します。

TDQ サンプルの実行

一時データ・キュー・サンプルを実行すると、Java プログラムが一時データと対話する方法を理解できます。プログラムは、順次キューに保管されている一時データの読み取りと書き込みを行うことができます。

始める前に

42 ページの『JCICS サンプルの実行』で説明されているとおりに、CICS 領域が構成されていることを確認します。

このタスクについて

このサンプルは、JCICS TDQ クラスの使用方法を示します。このサンプルでは、次の Java クラスおよびプログラムを使用します。

トランザクション	Program	Java クラス
JTD1	DFJ$JTD1	TDQ.ClassOne

手順

端末で JTD1 トランザクションに入って、このサンプルを実行します。 Task.out に以下のメッセージが表示されます。

```
Entering examples.TDQ.ClassOne.main()
Entering writeFixedData()
Leaving writeFixedData()
Entering writeFixedData()
Leaving writeFixedData()
Entering readFixedData()
Leaving readFixedData()
Entering readFixedDataConditional()
Leaving readFixedDataConditional()
Leaving examples.TDQ.ClassOne.main()
```

タスクの結果

CICS は次の処理を実行します。

1. トランザクションは、PROGRAM リソース DFJ$JTD1 で定義されるメイン Java クラスを実行します。
2. Java プログラムは、一時データ・キューにデータを書き込み、それを読み取ってから、キューを削除します。

TSQ サンプルの実行

一時記憶サンプルを実行すると、Java プログラムが一時記憶域キューと対話する方法を理解できます。一時記憶域キューは、任意の順序で読み取りと再読み取りが可能なデータ項目のキューです。このキューはタスクによって作成され、同じタスクまたは別のタスクにより削除されるまで持続します。

始める前に

42 ページの『JCICS サンプルの実行』で説明されているとおりに、CICS 領域が構成されていることを確認します。

このタスクについて

このサンプルは、JCICS TSQ クラスの使用方法、および他の Java プログラムと共用できるダイナミック・リンク・ライブラリー (DLL) としてクラスを構築する方法を示します。この例では、次の Java クラスおよびプログラムを使用します。

トランザクション	Program	Java クラス
JTS1	DFJ$JTS1	TSQ.ClassOne
	DFJ$JTSC	TSQ.Common

手順

JTS1 CICS トランザクションを入力して、この例を実行します。 Task.out に以下のメッセージが表示されます。

```
Entering TSQ.ClassOne.main()
Entering TSQ_Common.writeFixedData()
Leaving TSQ_Common.writeFixedData()
Entering TSQ_Common.serializeObject()
Leaving TSQ_Common.serializeObject()
Entering TSQ_Common.updateFixedData()
Leaving TSQ_Common.updateFixedData()
Entering TSQ_Common.writeConditionalFixedData()
Leaving TSQ_Common.writeConditionalFixedData()
Entering TSQ_Common.updateConditionalFixedData()
Leaving TSQ_Common.updateConditionalFixedData()
Entering TSQ_Common.readFixedData()
Leaving TSQ_Common.readFixedData()
Entering TSQ_Common.deserializeObject()
Leaving TSQ_Common.deserializeObject()
Entering TSQ_Common.readFixedConditionalData()
Number of items returned is 3
Leaving TSQ_Common.readFixedConditionalData()
Entering TSQ_Common.deleteQueue()
Leaving TSQ_Common.deleteQueue()
Leaving TSQ.ClassOne.main()
```

タスクの結果

CICS では次の処理が行われます。

1. トランザクションは、PROGRAM リソース DFJ$JTS1 で定義されるメイン Java クラスを実行します。Java プログラムは、PROGRAM リソース DFJ$JTSC で定義される別の共通 Java プログラムにリンクします。
2. 共通 Java プログラムは、補助一時記憶域キューに書き込み、そのキューを更新し、削除し、戻ります。

Web サンプルの実行

Web サンプルを実行すると、Java プログラムが CICS Web サポートを使用して Web ブラウザーと対話する方法を理解できます。

始める前に

42 ページの『JCICS サンプルの実行』で説明されているとおりに、CICS 領域が構成されていることを確認します。Web サンプルを実行する前に、「インターネット・ガイド」の『CICS Web サポートのコンポーネントの構成』 の説明に従ってく

ださい。サンプル・プログラム DFH$WB1A (アセンブラー) または DFH$WB1C (C) を使用して、CICS Web サポートが正しく構成されていることを確認します。

このタスクについて

このサンプルは、JCICS Web クラスおよび Document クラスの使用方法を示します。このサンプル・アプリケーションには、 Web ブラウザーからアクセスします。この例では、インバウンド・クライアント要求、HTTP ヘッダーおよびトランザクションの TCP/IP 特性に関する情報を取得します。この情報は、標準出力ストリーム System.out に書き込まれ、応答文書に挿入されます。文書に関する情報も取得され、System.out に書き込まれ、応答文書に挿入されます。その後、応答文書はクライアントに送信されます。

このサンプルでは、次の Java クラスおよびプログラムを使用します。

Program	Java クラス
DFJ$JWB1	Web.Sample1

手順

1. Web ブラウザーを開始し、絶対パス /CICS/CWBA/DFJ$JWB1 で CICS に接続する URL を入力します。 CICS は、次の応答文書を Web ブラウザーに戻します。

```
Web Sample1

Inbound Client Request Information:

Method: GET

Version: HTTP/1.1

Path: /cics/cwba/jcicxsa1

Request Type: HTTPYES

Query String: null

HTTP headers:

Value for HTTP header User-Agent is 'Mozilla/4.75 €en€ (WinNT; U)'

Browse of HTTP Headers started

Name: Host Value: winmvs2d.hursley.ibm.com:27361

Name: Connection Value: Keep-Alive, TE

Name: Accept Value: image/gif, image/x-xbitmap, image/jpeg,
      image/pjpeg, image/png,
      */*

Name: Accept-Encoding Value: gzip

Name: Accept-Language Value: en

Name: Accept-Charset Value: iso-8859-1,*,utf-8

Name: Cookie Value: PBC_NLSP=en_US

Name: TE Value: chunked
```

```
Name: Via Value: HTTP/1.0 sp15ce18.hursley.ibm.com (IBM-PROXY-WTE-US)

Name: User-Agent Value: Mozilla/4.75 €en€ (WinNT; U)

Browse of HTTP Headers completed

TCPIP Information:

Client Name: sp15ce18.hursley.ibm.com

Server Name: winmvs2d.hursley.ibm.com

Client Address: 9.20.136.28

ClientAddrNu: 9.20.136.28

Server Address: 9.20.101.8

ServerAddrNu: 9.20.101.8

Clientauth: NO

SSL: NO

TcpipService: HTTPNSSL

PortNumber: 27361

Document Information:

Doctoken: 33 92 112 0 0 0 0 1 64 64 64 64 64 64 64 64

Docsize: 2762
```

2. zFS における標準出力ストリームを確認します。 この例は、標準出力ストリーム System.out に情報メッセージを書き込み、標準出力ストリーム System.err にエラー・メッセージを書き込みます。 System.out 出力ストリームに書き込まれる出力の例は、次のとおりです。

```
Sample1 started
Method: GET (3)
Version: HTTP/1.1 (8)
Path: /cics/cwba/jcicxsa1 (19)
Request Type: HTTPYES
Value for HTTP header User-Agent is 'Mozilla/4.75  en  (WinNT; U)'
HTTP headers:
Name: Host (4)
Value: winmvs2d.hursley.ibm.com:27361 (30)
Name: Connection (10)
Value: Keep-Alive, TE (14)
Name: Accept (6)
Value: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */* (67)
Name: Accept-Encoding (15)
Value: gzip (4)
Name: Accept-Language (15)
Value: en (2)
Name: Accept-Charset (14)
Value: iso-8859-1,*,utf-8 (18)
Name: Cookie (6)
Value: PBC_NLSP=en_US (14)
Name: TE (2)
Value: chunked (7)
Name: Via (3)
Value: HTTP/1.0 sp15ce18.hursley.ibm.com (IBM-PROXY-WTE-US) (52)
Name: User-Agent (10)
Value: Mozilla/4.75  en  (WinNT; U) (28)
Client Name: sp15ce18.hursley.ibm.com (24)
```

```
Server Name: winmvs2d.hursley.ibm.com (24)
Client Address: 9.20.136.28 (11)
ClientAddrNu: 9.20.136.28
Server Address: 9.20.101.8 (10)
ServerAddrNu: 9.20.101.8
Clientauth: NO
SSL: NO
TcpipService: HTTPNSSL
PortNumber: 27361
Doctoken: Doctoken: 33 92 112 0 0 0 0 1 64 64 64 64 64 64 64 64
Docsize: 2762
Sample1 complete
```

CICS Explorer SDK を使用したアプリケーションの開発

CICS Explorer Software Development Kit (SDK) は、OSGi のサポートを含めて、CICS で Java アプリケーションを開発し、デプロイするための環境を提供します。

このタスクについて

SDK を使用すると、OSGi 仕様に準拠するように、新規アプリケーションの作成や既存 Java アプリケーションの再パッケージを行うことができます。 OSGi Service Platform は、コンポーネント・モデルを使用してアプリケーションを開発し、それらのアプリケーションを OSGi バンドルとしてフレームワークにデプロイするためのメカニズムを提供します。*OSGi バンドル* は、アプリケーションのデプロイメントの単位であり、バージョン管理情報、依存関係、およびアプリケーション・コードが入っています。OSGi の主な利点は、*OSGi サービス* と呼ばれる明確に定義されたインターフェースを介してのみアクセスされる再使用可能コンポーネントから、アプリケーションを作成できることです。また、Java アプリケーションのライフサイクルと依存関係をきめ細かく管理することもできます。 OSGi を使用したアプリケーションの開発については、OSGi Alliance Web サイトをご覧ください。

SDK を使用すると、任意のサポートされている CICS リリースで実行する Java アプリケーションを開発できます。異なるリリースの CICS は、別々のバージョンの Java をサポートします。また、JCICS API も、CICS の追加機能をサポートするように後のリリースで拡張されています。誤ったクラスを使用しないように、SDK はターゲット・プラットフォームをセットアップする機能を備えています。どのリリースの CICS 用に開発するかを定義すると、SDK は、使用できない Java クラスを自動的に非表示にします。

アプリケーションを開発し、デプロイするために以下の各ステップを実行する方法について詳しくは、SDK ヘルプで「*CICS Java Developer Guide*」を参照してください。

手順

1. Java 開発用のターゲット・プラットフォームをセットアップします。 ターゲット・プラットフォームでは、アプリケーション開発で CICS のターゲット・リリースに適切な Java クラスのみを使用することが確実になります。
2. Java アプリケーション開発用のプラグイン・プロジェクトを作成します。
3. ベスト・プラクティスを使用して Java アプリケーションを開発します。 CICS 用の Java アプリケーションの開発に慣れていない場合は、CICS Explorer SDK

で提供される JCICS サンプルを使用して開発に取り掛かることができます。Java アプリケーションで JCICS を使用するには、com.ibm.cics.server パッケージをインポートする必要があります。

4. CICS バンドル内の Java アプリケーションを zFS にデプロイします。 CICS バンドルは、1 つ以上の OSGi バンドルを含むことができ、CICS におけるアプリケーションのデプロイメント単位です。JVM サーバーで Java アプリケーションを実行する場合、アプリケーションをデプロイする JVMSERVER リソースの名前を認識しておく必要があります。

タスクの結果

CICS Explorer SDK を使用して CICS バンドル内のアプリケーションを正常に開発し、デプロイしました。

次のタスク

CICS BUNDLE リソースを作成して、JVM サーバーに OSGi バンドルをインストールしてください。CICS 領域でリソースを作成できない場合、システム・プログラマーが BUNDLE リソースを作成できます。バンドル・ディレクトリーが zFS 内で置かれている場所、およびターゲット JVM サーバーの名前をシステム・プログラマーに知らせる必要があります。詳しくは、97 ページの『JVM サーバーへの OSGi バンドルのインストール』を参照してください。

CICS Explorer SDK を使用したアプリケーションのマイグレーション

プールされた JVM で実行される既存のアプリケーションがあるときに、それらを JVM サーバーで実行したい場合、CICS Explorer SDK を使用して、アプリケーションを OSGi バンドルとして再パッケージすることができます。

このタスクについて

既存の Java アプリケーションを再パッケージするには、3 とおりの方式を使用できます。各方式は SDK ヘルプで詳しく説明されていますが、以下の手順で概要をまとめています。

手順

1. Java アプリケーションがスレッド・セーフであることを確認します。 JVM サーバーはマルチスレッド・ランタイム環境であるため、その環境で実行されるすべての Java アプリケーションがスレッド・セーフであることが重要です。
2. Java アプリケーションが System.exit() Java メソッドを使用しないことを確認します。 このメソッドが使用されると、JVM サーバーと CICS の両方がシャットダウンします。
3. Java アプリケーションを 1 つ以上の OSGi バンドルとしてパッケージします。アプリケーションをパッケージするには、次の 3 つの方式を使用できます。

   変換 Java アプリケーション用の Eclipse Java プロジェクトが既にある場合は、そのプロジェクトを OSGi プラグイン・プロジェクトに変換できます。この方法が推奨されるベスト・プラクティスです。OSGi バンドルは、プールされた JVM 環境と JVM サーバーで実行できます。

   注入 OSGi プラグイン・プロジェクトを作成し、既存の JAR ファイルの内容

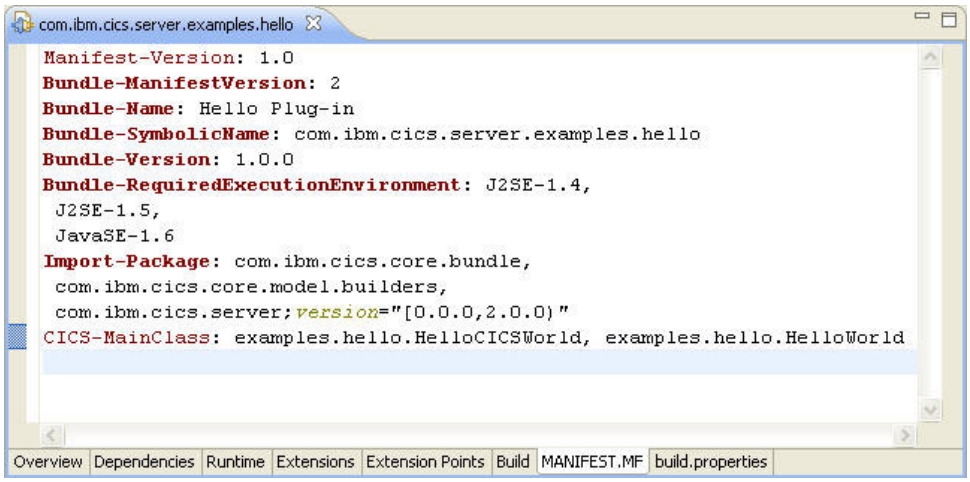
をインポートします。この方式が便利なのは、アプリケーションが既にスレッド・セーフであり、リファクタリングも再コンパイルも必要ない場合です。OSGi バンドルは、プールされた JVM 環境と JVM サーバーで実行できます。

ラッピング

OSGi プラグイン・プロジェクトを作成し、既存のバイナリー JAR ファイルをインポートします。この方式は、ライセンスの制限がある状態、またはバイナリー・ファイルを抽出できない状態で便利です。ただし、JAR ファイルを含む OSGi バンドルは、プールされた JVM 環境でサポートされません。

4. これらの方式ごとに、CICS-MainClass 宣言をプロジェクトのマニフェストに追加します。

   次の画面取りは、CICS Hello サンプルのマニフェスト・ファイル例を示しています。このアプリケーション例には 2 つのクラス (HelloCICSWorld と HelloWorld) が含まれ、これらは CICS-MainClass 宣言のマニフェスト・ファイルで宣言されます。アプリケーションで使用されるクラスごとに CICS-MainClass 宣言を追加する必要があります。



5. CICS バンドル内の OSGi バンドルを zFS にデプロイします。 CICS バンドルのプラグイン・リソース・ファイルでターゲット JVMSERVER リソースを指定する必要があります。

タスクの結果

1 つ以上の OSGi バンドルとしてパッケージされ、zFS で CICS バンドルとしてデプロイされるスレッド・セーフ・アプリケーションがあります。

次のタスク

146 ページの『JVM サーバーへのアプリケーションの移動』で説明されているように、システム・プログラマーは、JVM サーバーでアプリケーションを実行するのに必要なリソースを作成できます。

CICS で Java アプリケーションを開発するためのベスト・プラクティス

CICS で実行される Java アプリケーションの設計と開発を行う場合は、アプリケーションが、JVM を不要な状態のままにしたり、JVM の状態を望ましくない方法で変更したりしないようにしてください。 CICS サービスを使用して、JVM の状態を制御することができます。

JVM サーバーとプールされた JVM は別々の方法で作動しますが、開発される任意の Java アプリケーションは、両方のランタイム環境で正しく機能するように同じベスト・プラクティスに従うことができます。

JVM の状態の保護

ご使用のアプリケーションにより JVM の状態が変わる場合は、そのアプリケーションも元の状態にリセットされることを確実にしてください。例えば、アプリケーションはデフォルトのタイム・ゾーンをリセットし、このタイム・ゾーンに基づいて計算を行うことができます。同じ JVM を使用するその他のアプリケーションは、新しいデフォルトのタイム・ゾーンを使用しますが、このタイム・ゾーンが適切でない場合があります。

- アプリケーションがプールされた JVM で実行される場合、他のアプリケーションから分離されます。ただし、プールされた JVM は逐次再利用されます。あるアプリケーションによって加えられた変更は、後で同じ JVM で実行されるその他のアプリケーションに影響を与える可能性があります。
- アプリケーションが JVM サーバーで実行される場合、同じ JVM の別のスレッドでも実行中の他のアプリケーションから分離されません。JVM に対してアプリケーションが加える変更はすべて、他のアプリケーションに影響を与えます。

  ご使用のアプリケーションで System.exit() メソッドを使用しないでください。System.exit() メソッドを使用すると、JVM サーバーと CICS の両方がシャットダウンするため、アプリケーションの状態に影響を与える可能性があります。

JVM における静的状態の制御

JVM を不要な状態のままにしないでください。同じプールされた JVM を使用する後続のアプリケーションに状態が渡され、JVM サーバーでは、実行中のすべてのアプリケーション間で状態が共用されます。

変更可能なクラス・フィールドの状態に依存する場合、アプリケーションは独自の静的ストレージを再初期化しなければなりません。すべてのアプリケーション・クラスとシステム・クラスについて、静的変数の値は JVM 内で持続します。これには、アプリケーションによって明示的に使用されていなくてもアプリケーションに影響を与える可能性があるクラス、および静的イニシャライザーで使用されている値が含まれます。

大部分の場合、静的変数はストレージの再初期設定を避けるために使用され、静的変数が持続することを可能にすると、パフォーマンスを改善することができます。これらの変数の値のリセットがアプリケーションで必要な場合、アプリケーションは値自体をリセットする必要があります。アプリケーション設計の一部として意図的に含まれなかった、変更可能なクラス・フィールドと静的イニシャライザーの識別と除去を試みてください。

可能な場合は常に、クラス・フィールドを private および final として定義してください。native メソッドは final クラス・フィールドに書き込むことができ、private 以外のメソッドは、そのクラス・フィールドによって参照されるオブジェクトを取得でき、オブジェクトまたは配列の状態を変更できます。

情報をプログラム呼び出し間で持続させたい場合は、Java アプリケーションの設計の際に有利に状態を伝える機能を使用できます。静的状態と静的状態により参照されるオブジェクト・インスタンスは JVM 内で持続するので、同じ JVM の同じアプリケーションの将来の実行に役立つ可能性がある永続項目をアプリケーションが作成することは許容されます。

例えば、ある操作で DB2 情報を読み取って複合データ構造体を構成するとします。これは、コストのかかる操作であり、絶対必要な回数よりも多く繰り返したくありません。その複雑なデータ構造体をアプリケーションの静的ストレージに保管すれば、同じ JVM 内でそのアプリケーションを後で実行したときにもアクセスすることができるので、不要な初期化を避けることができます。オブジェクトが静的ストレージ、つまり静的クラス・フィールドにアンカーされる場合、決してガーベッジ・コレクションの候補になりません。

- JVM サーバーでは、システム・プログラマーが JVM サーバーを使用不可にするまで、すべてのアプリケーションの静的状態が持続します。OSGi バンドル・アクティベーター・クラスを提供すれば、JVM サーバーが再始動されてもオブジェクトの状態を維持することができます。これらのクラスには、JCICS 呼び出しを含むことはできません。
- プールされた JVM では、アプリケーションの後続の呼び出しが同じ JVM で実行されるという保証はありません。アプリケーションは、JVM で作成される永続項目の存在に依存してはなりません。アプリケーションは、不要な初期化を避けるために永続項目の有無を検査できますが、現在の JVM で検出されない場合は初期化する準備ができていなければなりません。

使用後の DB2 接続、ソケット、およびその他のタスク存続期間中のシステム・リソースのクローズ

アプリケーションがプールされた JVM で実行されている場合、そのアプリケーションは DB2 へのアクセス後に接続をクローズする必要があります。アプリケーションが接続をクローズしない場合、それ以降に同じアプリケーションを実行しても、接続をオープンできません。アプリケーションが JVM サーバーで実行される場合、別々のアプリケーションから DB2 に複数の接続を持つことが可能です。したがって、DB2 でタスクが終了したら、接続をクローズすることがベスト・プラクティスですが、必須ではありません。これは、タスクが完了すると接続が削除されるからです。

アプリケーションのスレッドを開始して、java.net パッケージを使用したソケットの管理を行う場合、そのアプリケーションが接続を管理し、クローズする必要があります。 java.net クラスを使用して作成されたソケットは、CICS ソケット・ドメインではなく、JVM のネイティブ・ソケット機能を使用します。CICS は、これらのソケットを使用して実行される通信の管理もモニターも行うことはできません。

アプリケーションによって使用されるその他のタスク存続期間中のシステム・リソースにも同じことが当てはまります。使用後に解放されなければなりません。

スレッド・セーフ問題があるかどうかについてのアプリケーションのテスト

常にスレッド・セーフ Java アプリケーションを作成してください。 CICS JVM アプリケーション分離ユーティリティーを使用して、Java アプリケーション内の静的変数の使用を監査することができます。このユーティリティーは、Java バイトコードを検査し、各クラスで使用される静的変数に関するレポートを作成します。この情報を使用して、ソース・コードの確認に役立てることができます。いずれの場合も、アプリケーションが静的変数を正しくリセットすることを確認してください。

Java アプリケーションが、所定の JVM で最初に使用されるときに正しく機能するものの、それ以降の使用時に正しく動作しない場合、この問題はおそらく、スレッド・セーフティーの問題が原因です。この場合は、問題判別作業の一環として CICS JVM アプリケーション分離ユーティリティーを使用して、問題の原因の特定に役立ててください。

Java からの構造化データとの対話

多くの場合、CICS Java プログラムは、当初は他のプログラミング言語用に設計されたデータと対話します。例えば、Java プログラムは、COBOL コピーブックで定義された COMMAREA を使用して COBOL プログラムにリンクしたり、C++ ヘッダー・ファイルを使用してデータが定義される VSAM ファイルからレコードを読み取ったりすることができます。インポーターを使用すると、これらの形式の構造化データと対話することができます。

JZOS および J2C を使用した、Java へのアプリケーション・データのインポート

CICS はコピーブック・インポーターをサポートするので、Java で他のプログラミング言語からの構造化データを使用することができます。サポートされるインポーターは、J2EE コネクター・アーキテクチャー (J2C) とも呼ばれる Java EE コネクター・アーキテクチャー (JCA) を使用して、JZOS ツールおよび Rational によって提供されます。

インポーターは、ソース・プログラムに含まれるデータ型をマップするので、アプリケーションはデータ構造内の個々のフィールドにアクセスできます。JZOS または Rational の J2C ツールを使用すると、データと対話して Java クラスを作成することができます。その結果、CICS で Java と他のプログラム間でデータを受け渡すことができます。

CICS は、以下のインポーターからの Java 成果物をサポートします。

- Rational® Application Developer (RAD) および Rational Developer for System z における J2C ツールからのデータ・バインディング Bean
- IBM JZOS Batch Toolkit for z/OS SDK からのレコード

IBM Redbook「Java Application Development for CICS」では、既存の COBOL アプリケーションを操作する、Heritage Trader アプリケーションと呼ばれるアプリケーション例を使用しています。以下のトピックに関する情報が提供されます。

- JZOS および J2C のインストール方法

- JCICS への COBOL アプリケーションのマイグレーション
- J2C 用の Java データ・バインディング・クラスの作成
- JZOS でのラッパー・クラスの生成
- JCICS API を使用した、Web、ファイル、および DB2 アクセスの実装例

J2C 要件

エンタープライズ・アプリケーションの作成に使用できる J2EE コネクター成果物を作成することができます。 RAD J2C ウィザードは、COBOL およびその他のアプリケーション・プログラム・データ構造にマップするクラスまたは一連のクラスの作成に役立ちます。

Rational J2C インポーターを使用するには、Windows または Linux ワークステーションに RAD が必要です。

JZOS 要件

IBM JZOS Batch Toolkit for z/OS SDK は、z/OS で Java バッチ機能を提供する 1 組のツールです。JZOS には、バッチ・ジョブまたは開始タスクとして Java アプリケーションを直接実行するためのランチャー、および Java アプリケーションから直接使用可能な従来の z/OS データと主なシステム・サービスにアクセスする 1 組の Java メソッドが含まれます。

JZOS は、COBOL コピーブックおよびアセンブラー DSECT からのレコード・クラスの自動生成をサポートします。

JZOS ダウンロードには、PDF 形式で「*JZOS COBOL Record Generator User's Guide*」および「*JZOS Assembler Record Generator User's Guide*」が入っています。

IBM Redbooks

Java Application Development for CICS

Java Connectors for CICS Featuring the J2EE Connector Architecture

Java Stand-alone Applications on z/OS Volume 2

J2C 情報

RAD: エンタープライズ情報システム (EIS) への接続

RAD: COBOL インポーターの概要

CICS Transaction Gateway Programming Guide

JZOS 情報

JZOS Java Launcher and Toolkit Overview

JZOS Batch Launcher and Toolkit Installation and User Guide

JCICS を使用した Java プログラミング

CICS Java クラス・ライブラリー (JCICS) を使用して CICS サービスにアクセスする Java アプリケーションを作成できます。 JCICS は、Java において、CICS でサポートされている他の言語 (COBOL など) に提供される **EXEC CICS** アプリケーション・プログラミング・インターフェースに相当するものです。

JCICS を使用すると、CICS リソースにアクセスする Java アプリケーションを作成し、他の言語で作成されたプログラムと統合することができます。 **EXEC CICS** API の大部分の機能がサポートされます。このライブラリーは、`com.ibm.cics.server.jar` ファイルで CICS および CICS Explorer SDK に提供されています。

CICS 用 Java クラス・ライブラリー (JCICS)

CICS 用の Java クラス・ライブラリー (JCICS) は、**EXEC CICS** API コマンドの大部分の機能をサポートします。

JCICS クラスは、クラス定義から生成される Javadoc で詳しく説明されています。Javadoc は「JCICS Class Reference」で入手可能です。

JavaBeans

JCICS の一部のクラスは JavaBeans として使用できます。つまり、Eclipse などのアプリケーション開発ツールでカスタマイズし、直列化し、JavaBeans API を使用して操作することができます。

以下の JavaBeans が JCICS で使用可能です。

- Program
- ESDS
- KSDS
- RRDS
- TDQ
- TSQ
- AttachInitiator
- EnterRequest

これらの Bean はイベントを定義しません。プロパティーとメソッドで構成されます。次の 3 つの方法のいずれかで実行時にインスタンス化することができます。

- クラス自体の new メソッドを呼び出す。この方法が優先されます。
- プロパティー値を手動で設定して、クラスの名前の Beans.instantiate() を呼び出す。
- プロパティー値を設計時に設定して、`.ser` ファイルの Beans.instantiate() を呼び出す。

最初の 2 つのオプションのどちらかを選択する場合、プロパティー値 (CICS リソースの名前を含めて) は、実行時に適切な set メソッドを呼び出すことによって設定されなければなりません。

ライブラリー構造

各 JCICS ライブラリー・コンポーネントは、4 つのカテゴリー (インターフェース、クラス、例外、エラー) のいずれかに分類されます。

インターフェース
一部のインターフェースは、1 組の定数を定義するために提供されます。例えば、TerminalSendBits インターフェースは、`java.util.BitSet` の構成に使用できる 1 組の定数を提供します。

クラス
指定されたクラスは、大部分の JCICS 機能を提供します。`API` クラスは抽象クラスであり、ABEND と例外を除いて、CICS API の一部に対応する、すべてのクラスに共通する初期化を提供します。例えば、`Task` クラスは、CICS タスクに対応する 1 組のメソッドと変数を提供します。

エラーと例外
Java 言語は、クラス `Throwable` のサブクラスとして例外とエラーの両方を定義します。JCICS は、`Error` のサブクラスとして `CicsError` を定義します。`CicsError` は、重大エラーに使用される他のすべての CICS エラー・クラスのスーパークラスです。

JCICS は、`Exception` のサブクラスとして `CicsException` を定義します。`CicsException` は、(CICS QIDERR 条件を表す、`InvalidQueueIdException` などの `CicsConditionException` クラスを含む) すべての CICS 例外クラスのスーパークラスです。

詳しくは、 62 ページの『エラー処理と異常終了』を参照してください。

CICS リソース

プログラムや一時記憶域キューなどの CICS リソースは、該当する Java クラスのインスタンスによって表され、リソースの名前などの各種プロパティーの値によって識別されます。

リソースは、CICS Explorer、CEDA トランザクション、または CICSPlex® SM BAS を使用して CICS に対して定義されなければなりません。CICS リソースについては、「*CICS Resource Definition Guide*」または「*CICSPlex System Manager 概念および計画*」マニュアルを参照してください。リモート・リソースを指し示すようにリソースをローカル側で定義することによって、暗黙的なリモート・アクセスを使用することが可能です。

データを渡すための引数

チャネルとコンテナーを使用するか、通信域 (COMMAREA) を使用して、プログラム間でデータを受け渡すことができます。

COMMAREA を使用する場合、一度に渡すデータは 32 KB に制限されます。チャネルとコンテナーを使用する場合、プログラム間で 32 KB より多く渡すことができます。COMMAREA またはチャネル、およびその他のすべてのパラメーターは、引数として該当するメソッドに渡されます。

メソッドの多くは多重定義されています。すなわち、バージョンが異なれば、取る引数の数か、引数のタイプのどちらかが異なります。引数がないか、最小限の必須

引数があるメソッドや、すべての引数があるメソッドがあります。例えば、Program クラスには、次のようにさまざまな link() メソッドが含まれています。

link()
このメソッドは、COMMAREA を使用してデータを受け渡したり、他のオプションを指定したりせず、単純なリンクを行います。

link(com.ibm.cics.server.CommAreaHolder)
このメソッドは、COMMAREA を使用してデータを受け渡すのみで、他のオプションを指定せず、単純なリンクを行います。

link(com.ibm.cics.server.CommAreaHolder, int)
このメソッドは、COMMAREA を使用してデータを受け渡し、DATALENGTH 値を使用して COMMAREA 内のデータの長さを指定することにより、分散リンクを行います。

link(com.ibm.record.IByteBuffer)
このメソッドは、VisualAge for Java に付属する Java Record Framework の IByteBuffer インターフェースを実装するオブジェクトを使用して、リンクを行います。

link(com.ibm.cics.server.Channel)
このメソッドは、チャネルを使用して 1 つ以上のコンテナー内のデータを受け渡すことによって、リンクを行います。

直列化可能クラス

直列化可能クラスは、受動/活動化のサイクル後も存続できる JCICS クラスです。

次に、直列化可能クラスのリストを示します。

- AddressResource
- AttachInitiator
- CommAreaHolder
- EnterRequest
- ESDS
- File
- KeyedFile
- KSDS
- NameResource
- Program
- RemotableResource
- Resource
- RRDS
- StartRequest
- SynchronizationResource
- SyncLevel
- TDQ
- TSQ

- TSQType

System.out および System.err

Java 関連の CICS タスクごとに、CICS は、標準出力および標準エラー・ストリームとして使用できる、2 つの Java `PrintWriters` クラスを自動的に作成します。標準出力と標準エラー・ストリームは、`out` と `err` と呼ばれる、`Task` クラス内のパブリック・フィールドです。

CICS タスクが端末 (この場合、端末は*基本機構*と呼ばれます) から駆動される場合、CICS は標準出力および標準エラー・ストリームをタスクの端末にマップします。

タスクに基本機構としての端末がない場合、標準出力および標準エラー・ストリームは System.out および System.err に送信されます。System.out および System.err は、CICS 一時データ・キュー CESO と CESE にそれぞれマップされます。CICS システム・プログラマーは、CICS のインストール時にこれらのキュー、および CICS メッセージに使用されるその他のキューを作成します。これらのメッセージ・キューは、DFH$TDWT サンプル・プログラムなどのユーティリティー・プログラムを使用してアクセスし、印刷または表示することができます。DFH$TDWT は CICSTS42.CICS.SDFHLOAD 内にあります。

スレッド

JVM 内の初期スレッドのみが JCICS API にアクセスできます。他のスレッドを作成できますが、すべての要求を初期スレッドから JCICS API に転送する必要があります。JVM サーバー環境では、複数の初期スレッドが、同じ JVM を使用して JCICS API にアクセスできます。

さらに、初期スレッド以外のすべてのスレッドが、次のいずれかの操作を行う前に終了していることを確認する必要があります。

- link()
- xctl()
- setNextTransaction(), setNextCOMMAREA()
- commit(), rollback()
- `AbendException` を戻す

JCICS サービスの解説

`EXEC CICS` API を使用して非 Java プログラムから使用可能なオプションとサービスの多くは、JCICS を使用して Java プログラムから使用可能です。

Java プログラムにおける CICS 例外処理

CICS で発生する問題に対処するために、CICS ABEND および例外は、Java 例外処理体系に組み込まれています。

通常の CICS ABEND はすべて、単一の Java 例外 `AbendException` にマップされます。一方、各 CICS 条件は別々の Java 例外にマップされます。これにより、Java の ABEND 処理モデルは他のプログラミング言語とほぼ同じになります。すなわち、すべての ABEND について単一のハンドラーに制御が与えれ、そのハンドラーは特定の ABEND に照会してから、処理内容を決定する必要があります。

条件を表す例外が CICS 自体によってキャッチされる場合、その例外は ABEND になります。

Java 例外処理は、他の言語の ABEND および条件処理に完全に組み込まれるので、ABEND は、言語に依存しない標準の方法で、Java プログラムと非 Java プログラム間に波及することができます。条件は、ABEND にマップされてから、その条件の原因になったかその条件を検出したプログラムを終了します。

ただし、他のプログラミング言語の ABEND 処理モデルには、複数の相違点があります。これらの相違点は、Java 例外処理体系の性質や、Java API の基礎となる一部のテクノロジーの実装に起因します。

- 他のプログラミング言語では処理できない ABEND を、Java プログラムでキャッチできます。これらの ABEND は通常、同期点処理時に発生します。これらの ABEND が Java アプリケーションを中断しないようにするために、チェックなし例外の拡張にマップされます。したがって、これらの ABEND の宣言もキャッチも必要ありません。
- プログラム終了などの複数の内部 CICS イベントも、Java 例外にマップされるので、Java アプリケーションでキャッチできます。また、正常な状態を中断しないように、これらのイベントはチェックなし例外の拡張にマップされ、キャッチも宣言も必要ありません。

例外の 3 つのクラス階層が CICS に関連しています。

1. CicsError。java.lang.Error を拡張し、AbendError および UnknownCicsError のベースです。
2. CicsRuntimeException。java.lang.RuntimeException を拡張する一方で、以下によって拡張されます。

   AbendException
   通常の CICS ABEND を表します。

   EndOfProgramException
   リンク先プログラムが通常どおりに終了したことを示します。

   TransferOfControlException
   CICS XCTL コマンドに相当する xctl() メソッドをプログラムが使用したことを示します。
3. CicsException。java.lang.Exception を拡張し、サブクラスを持ちます。

   CicsConditionException.
   すべての CICS 条件の基本クラス。

CICS エラー処理コマンド:

上記のように、CICS 条件処理は Java 例外体系に組み込まれています。相当する「**EXEC CICS**」コマンドが Java でサポートされる方法を以下で説明します。

HANDLE ABEND
任意の CICS 対応言語でプログラムによって生成される ABEND を処理するには、catch 節に AbendException を表示して、Java try-catch ステートメントを使用します。

HANDLE CONDITION
PGMIDERR などの特定の条件を処理するには、適切な例外の名前を指定する catch 節を使用します。この場合は、InvalidProgramException です。または、すべての CICS 条件がキャッチされる場合は、CicsConditionException という名前を指定する catch 節を使用します。

IGNORE CONDITION
このコマンドは、Java アプリケーションでは適切ではありません。

POP HANDLE および PUSH HANDLE
これらのコマンドは、Java アプリケーションでは適切ではありません。CICS ABEND および条件を表すのに使用される Java 例外は、スコープ内の任意の catch ブロックによってキャッチされます。

CICS 条件:

Java の条件処理モデルは、他の CICS プログラミング言語とは異なります。

COBOL では、条件ごとに条件処理ラベルを定義できます。その条件が CICS コマンドの処理中に発生する場合、制御はラベルに転送されます。

C および C++ では、条件に例外処理ラベルを定義することはできません。条件を検出するために、各 CICS コマンドの後に、EIB 内の RESP フィールドが検査されなければなりません。

Java では、CICS コマンドによって戻される条件はすべて、Java 例外にマップされます。すべての CICS コマンドを try-catch ブロックに組み込んで、条件ごとに特定の処理を行うか、特定の例外が適切でない場合は、単一の null catch 節を持つことができます。または、条件を波及させて、より大きいスコープで catch 節によって処理できるようにすることができます。

CICS 条件と Java 例外間の関係については、80 ページの『JCICS 例外マッピング』を参照してください。

エラー処理と異常終了

Java プログラムから ABEND を開始するには、Task.abend() または Task.forceAbend() メソッドのいずれかを呼び出す必要があります。

メソッド	JCICS クラス	EXEC CICS コマンド
abend()、forceAbend()	タスク	ABEND

ABEND
Java プログラムから ABEND を開始するには、Task.abend() メソッドのいずれかを呼び出します。これにより、アベンド条件が CICS で設定され、AbendException がスローされます。AbendException が上位レベルのアプリケーション・オブジェクト内でキャッチされないか、呼び出し側プログラムに登録されている ABEND ハンドラー (ある場合) によって処理される場合、CICS はトランザクションを終了し、ロールバックします。

各種 abend() メソッドは次のとおりです。

- abend(String *abcode*)。ABEND コード *abcode* で ABEND が生じます。

- abend(String *abcode*, boolean *dump*)。ABEND コード *abcode* で ABEND が生じます。 **dump** パラメーターが false である場合、ダンプは取られません。
- abend()。ABEND コードもダンプもない ABEND が生じます。

ABEND CANCEL

処理できない ABEND を開始するには、Task.forceAbend() メソッドのいずれかを呼び出します。上記のように、これにより、AbendCancelException がスローされ、Java プログラムでキャッチされます。これを行う場合は、**ABEND_CANCEL** 処理を完了するために例外を再スローする必要があります。その結果、制御が CICS に戻ると、CICS はトランザクションを終了し、ロールバックします。通知目的の AbendCancelException のキャッチのみを行ってから、再スローしてください。

各種 forceAbend() メソッドは次のとおりです。

- forceAbend(String *abcode*)。ABEND コード *abcode* で **ABEND CANCEL** が生じます。
- forceAbend(String *abcode*, boolean *dump*)。ABEND コード *abcode* で **ABEND CANCEL** が生じます。**dump** パラメーターが false である場合、ダンプは取られません。
- forceAbend()。ABEND コードもダンプもない **ABEND CANCEL** が生じます。

APPC マップ式会話

APPC 非マップ式会話は、JCICS API からサポートされません。

APPC マップ式会話:

メソッド	JCICS クラス	EXEC CICS コマンド
initiate()	AttachInitiator	ALLOCATE、CONNECT PROCESS
converse()	Conversation	CONVERSE
get*() メソッド	Conversation	EXTRACT ATTRIBUTES
get*() メソッド	Conversation	EXTRACT PROCESS
free()	Conversation	FREE
issueAbend()	Conversation	ISSUE ABEND
issueConfirmation()	Conversation	ISSUE CONFIRMATION
issueError()	Conversation	ISSUE ERROR
issuePrepare()	Conversation	ISSUE PREPARE
issueSignal()	Conversation	ISSUE SIGNAL
receive()	Conversation	RECEIVE
send()	Conversation	SEND
flush()	Conversation	WAIT CONVID

基本マッピング・サポート (BMS)

基本マッピング・サポート (BMS) は、CICS プログラムと端末装置間のアプリケーション・プログラミング・インターフェースです。JCICS は、BMS アプリケーション・プログラミング・インターフェースの一部をサポートします。

メソッド	JCICS クラス	EXEC CICS コマンド
sendControl()	TerminalPrincipalFacility	SEND CONTROL
sendText()	TerminalPrincipalFacility	SEND Text
	サポートされない	SEND MAP、RECEIVE MAP

チャネルとコンテナー

コンテナー は、プログラム間で情報を渡すための、データの名前付きブロックです。コンテナーは、チャネル と呼ばれる集合にグループ化されます。CICS エンタープライズ Bean の作成時に、チャネルとコンテナーに関連した JCICS コマンドを使用できます。ただし、CICS は、IIOP 要求ストリームを介したチャネルの送信をサポートしません。

チャネルとコンテナーの概要、および非 Java アプリケーションでのチャネル使用の手引きについては、「CICS アプリケーション・プログラミング」の『チャネルを使用した拡張プログラム間データ転送』を参照してください。

Java プログラムが既存の CICS アプリケーション・データにアクセスできるようにするツールについては、「CICS での Java アプリケーション」の『Java からの構造化データとの対話』を参照してください。

注: CICS は、IIOP 要求ストリームを介したチャネルの送信をサポートしません。例えば、リモート領域のエンタープライズ Bean にチャネルを渡すことはできません。

表 2 では、チャネルとコンテナーに対する JCICS サポートを実装するクラスとメソッドをリストしています。

表 2. チャネルとコンテナーに対する JCICS サポート

メソッド	JCICS クラス	EXEC CICS コマンド
containerIterator()	Channel	STARTBROWSE CONTAINER
createContainer()	Channel	
deleteContainer()	Channel	DELETE CONTAINER CHANNEL
getContainer()	Channel	
getName()	Channel	
delete()	Container (コンテナー)	DELETE CONTAINER CHANNEL
get()、getLength()	Container (コンテナー)	GET CONTAINER CHANNEL [NODATA]
getName()	Container (コンテナー)	
put()	Container (コンテナー)	PUT CONTAINER CHANNEL
getOwner()	ContainerIterator	
hasNext()	ContainerIterator	
next()	ContainerIterator	GETNEXT CONTAINER BROWSETOKEN
remove()	ContainerIterator	
link()	Program	LINK
xctl()	Program	XCTL

表 2. チャネルとコンテナーに対する JCICS サポート (続き)

メソッド	JCICS クラス	EXEC CICS コマンド
setNextChannel()	TerminalPrincipalFacility	RETURN CHANNEL
issue()	StartRequest	START CHANNEL
createChannel()	タスク	
getCurrentChannel()	タスク	ASSIGN CHANNEL
containerIterator()	タスク	STARTBROWSE CONTAINER

CICS 条件 CHANNELERR の結果、ChannelErrorException がスローされます。 CONTAINERERR CICS 条件の結果、ContainerErrorException になります。 CCSIDERR CICS 条件の結果、CCSIDErrorException になります。

JCICS におけるチャネルとコンテナーの作成:

チャネルを作成するには、Task クラスの createChannel() メソッドを使用します。

例えば、次のようになります。

```
Task t=Task.getTask();
Channel custData = t.createChannel("Customer_Data");
```

createChannel メソッドに提供されるストリングは、Channel オブジェクトが CICS に認識されている名前です (この名前は、CICS 命名規則に準拠するために、16 文字までスペースで埋め込まれます)。

チャネルに新しいコンテナーを作成するには、Channel の createContainer() メソッドを使用します。例えば、次のようになります。

```
Container custRec = custData.createContainer("Customer_Record");
```

createContainer() メソッドに提供されるストリングは、Container オブジェクトが CICS に認識されている名前です (この名前は、CICS 命名規則に準拠するために、必要に応じて 16 文字までスペースで埋め込まれます)。同じ名前のコンテナーがこのチャネルに既に存在する場合、ContainerErrorException がスローされます。

コンテナーへのデータの書き込み:

Container オブジェクトにデータを書き込むには、Container.put() メソッドを使用します。

Container オブジェクトにデータを書き込むには、Container.put() メソッドを使用します。データはバイト配列またはストリングとしてコンテナーに追加できます。例えば、次のようになります。

```
String custNo = "00054321";
byte[] custRecIn = custNo.getBytes();
custRec.put(custRecIn);
```

または

```
custRec.put("00054321");
```

別のプログラムまたはタスクへのチャネルの受け渡し:

プログラム・リンクまたはプログラム制御転送 (XCTL) 呼び出しでチャネルを受け渡すには、Program クラスの link() および xctl() メソッドをそれぞれ使用します。

```
programX.link(custData);

programY.xctl(custData);
```

プログラム戻り呼び出しで次のチャネルを設定するには、TerminalPrincipalFacility クラスの setNextChannel() メソッドを使用します。

```
terminalPF.setNextChannel(custData);
```

START 要求でチャネルを渡すには、StartRequest クラスの issue メソッドを使用します。

```
startrequest.issue(custData);
```

現行チャネルの受け取り:

プログラムが現行チャネルを明示的に受け取る必要はありません。ただし、プログラムは現行チャネルを現行タスクから取得することができます。

プログラムが現行タスクから現行チャネルを取得する場合、タスクはコンテナーを名前で取り出すことができます。

```
Task t = Task.getTask();
Channel custData = t.getCurrentChannel();
if (custData != null) {
    Container custRec = custData.getContainer("Customer_Record");
} else {
    System.out.println("There is no Current Channel");
}
```

コンテナーからのデータの取得:

コンテナー内のデータをバイト配列に読み取るには、Container.get() メソッドを使用します。

```
byte[] custInfo = custRec.get();
```

現行チャネルのブラウズ:

チャネルが渡される JCICS プログラムは、そのチャネルを明示的に受け取ることなく、すべての Container オブジェクトにアクセスできます。

これを行うには、ContainerIterator オブジェクトを使用します (ContainerIterator クラスは java.util.Iterator インターフェースを実装します)。Task オブジェクトが現行タスクからインスタンス化される場合、その containerIterator() メソッドは、現行チャネルの Iterator を戻し、現行チャネルがない場合は null を戻します。例えば、次のようになります。

```
Task t = Task.getTask();
ContainerIterator ci = t.containerIterator();
While (ci.hasNext()) {
    Container custData = ci.next();
    // Process the container...
}
```

JCICS の例:

この例は、PAYR という名前の COBOL サーバー・プログラムを呼び出す、Payroll と呼ばれる Java クラスの抜粋を示しています。Payroll クラスは、JCICS com.ibm.cics.server.Channel および com.ibm.cics.server.Container クラスを使用して、非 Java クライアント・プログラムが **EXEC CICS** コマンドを使用して実行するものと同じものを実行します。

```
import com.ibm.cics.server.*;
public class Payroll {
     ...
     Task t=Task.getTask();

     // create the payroll_2004 channel
     Channel payroll_2004 = t.createChannel("payroll-2004");

     // create the employee container
     Container employee = payroll_2004.createContainer("employee");

     // put the employee name into the container
     employee.put("John Doe");

     // create the wage container
     Container wage = payroll_2004.createContainer("wage");

     // put the wage into the container
     wage.put("2000");

     // Link to the PAYROLL program, passing the payroll_2004 channel
     Program p = new Program();
     p.setName("PAYR");
     p.link(payroll_2004);

     // Get the status container which has been returned
     Container status = payroll_2004.getContainer("status");

     // Get the status information
     byte[] payrollStatus = status.get();
     ...
}
```

図 3. JCICS com.ibm.cics.server.Channel および com.ibm.cics.server.Container クラスを使用して、COBOL サーバー・プログラムにチャネルを渡す Java クラス

診断サービス

JCICS アプリケーション・プログラミング・インターフェースは、以下の CICS トレースおよびダンプ・コマンドをサポートします。

メソッド	JCICS クラス	EXEC CICS コマンド
	サポートされない	DUMP
enterTrace()	EnterRequest	ENTER
enableTrace()、disableTrace()	Region、Task	TRACE

文書サービス

このセクションでは、DOCUMENT アプリケーション・プログラミング・インターフェースにおけるコマンドの JCICS サポートについて説明します。

Document クラスは **EXEC CICS DOCUMENT** API にマップされます。DocumentLocation クラスのコンストラクターは、**EXEC CICS DOCUMENT** API の AT および TO キーワードにマップされます。SymbolList クラスの setter と getter は、**EXEC CICS DOCUMENT** API の SYMBOLLIST、LENGTH、DELIMITER、および UNESCAPE キーワードにマップされます。

メソッド	JCICS クラス	EXEC CICS コマンド
create*()	Document	DOCUMENT CREATE
append*()	Document	DOCUMENT INSERT
insert*()	Document	DOCUMENT INSERT
addSymbol()	Document	DOCUMENT SET
setSymbolList()	Document	DOCUMENT SET
retrieve*()	Document	DOCUMENT RETRIEVE
get*()	Document	DOCUMENT

環境サービス

CICS 環境サービスは、アプリケーション・プログラムに関連する CICS データ域、パラメーター、およびリソース属性にアクセスできるようにします。

JCICS サポートに相当する **EXEC CICS** コマンドとオプションは次のとおりです。

- ADDRESS
- ASSIGN
- INQUIRE SYSTEM
- INQUIRE TASK
- INQUIRE TERMINAL/NETNAME

ADDRESS:

ADDRESS API コマンド・オプションには次のサポートが提供されます。

EXEC CICS ADDRESS コマンドの詳細については、「CICS アプリケーション・プログラミング」の『ADDRESS』を参照してください。

ACEE アクセス制御環境エレメント (ACEE) は、CICS ユーザーがサインオンするときに外部セキュリティー・マネージャーによって作成されます。このオプションは JCICS ではサポートされません。

COMMAREA
COMMAREA には、コマンドで渡されるユーザー・データが入っています。COMMAREA ポインターは、**CommAreaHolder** 引数によって、リンクされたプログラムに自動的に渡されます。詳しくは、58 ページの『データを渡すための引数』を参照してください。

CWA 共通作業域 (CWA) には、タスク間で共用可能なグローバル・ユーザー・データが入っています。

EIB には、最後に実行された CICS コマンドに関する情報が入っています。EIB 値へのアクセスは、該当するオブジェクトのメソッドによって提供されます。その例を次に示します。

eibtrnid
Task クラスの getTransactionName() メソッドによって戻されます。

eibaid TerminalPrincipalFacility クラスの getAIDbyte() メソッドによって戻されます。

eibcposn
Cursor クラスの getRow() および getColumn() メソッドによって戻されます。

TCTUA
端末管理テーブル・ユーザー域 (TCTUA) には、CICS トランザクションを起動する端末 (基本機構) に関連したユーザー・データが入っています。この領域は、アプリケーション・プログラム間で情報を渡すのに使用されますが、関係するアプリケーション・プログラムに同じ端末が関連付けられている場合のみです。TCTUA の内容は、TerminalPrincipalFacility クラスの getTCTUA() メソッドを使用して取得できます。

TWA トランザクション作業域 (TWA) には、CICS タスクに関連したユーザー・データが入っています。この領域は、アプリケーション・プログラム間で情報を渡すのに使用されますが、同じタスク内にある場合のみです。TWA のコピーは、Task クラスの getTWA() メソッドを使用して取得できます。

ASSIGN:

ASSIGN API コマンド・オプションには次のサポートが提供されます。

このコマンドについて詳しくは、「CICS アプリケーション・プログラミング」の『ASSIGN』を参照してください。

メソッド	JCICS クラス
getABCODE()	AbendException
getAPPLID()	領域
getCurrentChannel()	タスク
getCWA()	領域
getName()	TerminalPrincipalFacility または ConversationPrincipalFacility
getFCI()	タスク
getNetName()	TerminalPrincipalFacility または ConversationPrincipalFacility
getPrinSysid()	TerminalPrincipalFacility または ConversationPrincipalFacility
getProgramName()	タスク
getQNAME()	タスク
getSTARTCODE()	タスク
getSysid()	領域
getTCTUA()	TerminalPrincipalFacility
getTERMCODE()	TerminalPrincipalFacility
getTWA()	タスク

メソッド	JCICS クラス
getUSERID(), Task.getUSERID()	Task、TerminalPrincipalFacility または ConversationPrincipalFacility

その他の ASSIGN オプションはサポートされません。

INQUIRE SYSTEM:

INQUIRE SYSTEM SPI オプションに対するサポートが提供されます。

メソッド	JCICS クラス
getAPPLID()	領域
getSYSID()	領域

その他の **INQUIRE SYSTEM** オプションはサポートされません。

INQUIRE TASK:

INQUIRE TASK API コマンド・オプションには次のサポートが提供されます。

メソッド	JCICS クラス
getSTARTCODE()	タスク
getTransactionName()	タスク
getUSERID()	タスク

FACILITY
タスクの基本機構で getName() メソッドを呼び出すことによって、タスクの基本機構の名前を見つけることができます。基本機構は、現行の Task オブジェクトで getPrincipalFacility() メソッドを呼び出すことによって見つけることができます。

FACILITYTYPE
Java instanceof 演算子を使用して、戻されたオブジェクト参照のクラスを確認することによって、機構のタイプを判別できます。

その他の INQUIRE TASK オプションはサポートされません。

INQUIRE TERMINAL および INQUIRE NETNAME:

INQUIRE TERMINAL および **INQUIRE NETNAME** SPI オプションには次のサポートが提供されます。

メソッド	JCICS クラス
getUSERID()	Terminal、ConversationalPrincipalFacility
Terminal.getUser()	Terminal、ConversationalPrincipalFacility

現行の Task オブジェクトで、またはタスクの基本機構を表すオブジェクトで getUSERID() メソッドを呼び出すことでも、USERID 値を見つけることができます。

その他の **INQUIRE TERMINAL** または **INQUIRE NETNAME** オプションはサポートされません。

ファイル・サービス

JCICS は、CICS ファイルおよび索引のタイプごとに **EXEC CICS** API コマンドにマップされるクラスおよびメソッドを提供します。

Java プログラムが既存の CICS アプリケーション・データにアクセスできるようにするツールについては、「CICS での Java アプリケーション」の『Java からの構造化データとの対話』を参照してください。

CICS は、次のタイプのファイルをサポートします。

- キー順データ・セット (KSDS)
- 入力順データ・セット (ESDS)
- 相対レコード・データ・セット (RRDS)

KSDS および ESDS ファイルには、代替 (または 2 次) 索引を備えることができます。CICS は、2 次索引から RRDS ファイルへのアクセスをサポートしません。2 次索引は、それ自体が別々の KSDS ファイルである、つまり別々の FD 項目を持つ場合と同じように CICS によって扱われます。

KSDS、ESDS (1 次索引)、および ESDS (2 次索引) ファイルのアクセスにはいくつかの相違点があります。すなわち、常に共通インターフェースを使用できるとは限りません。

レコードは、あらゆるタイプのファイルで読み取り、更新、削除、およびブラウズを行うことができます。ただし、ESDS ファイルからはレコードを削除できません。

データ・セットについて詳しくは、VSAM data sets: KSDS, ESDS, RRDS を参照してください。

データを読み取る Java コマンドは、**EXEC CICS** コマンドの SET オプションに相当するもののみをサポートします。戻されるデータは、CICS ストレージから Java オブジェクトに自動的にコピーされます。

ファイル制御に関連する Java インターフェースには、5 つのカテゴリーがあります。

File 他のファイル・クラスのスーパークラス。すべてのファイル・クラスに共通のメソッドが含まれます。

KeyedFile
1 次索引を使用してアクセスされる KSDS ファイル、2 次索引を使用してアクセスされる KSDS ファイル、および 2 次索引を使用してアクセスされる ESDS ファイルに共通のインターフェースが含まれます。

KSDS KSDS ファイルに固有のインターフェースが含まれます。

ESDS 相対バイト・アドレス (RBA、1 次索引) または拡張相対バイト・アドレス

(XRBA) からアクセスされる ESDS ファイルに固有のインターフェースが含まれます。RBA ではなく、XRBA を使用するには、setXRBA(true) メソッドを発行します。

RRDS 相対レコード番号 (RRN、1 次索引) からアクセスされる RRDS ファイルに固有のインターフェースが含まれます。

ファイルごとに、作動可能な 2 つのオブジェクト、つまり、File オブジェクトと FileBrowse オブジェクトがあります。 File オブジェクトはファイル自体を表し、次の API オペレーションを実行するメソッドで使用できます。

- DELETE
- READ
- REWRITE
- UNLOCK
- WRITE
- STARTBR

File オブジェクトは、必要なファイル・クラスを明示的に開始するユーザー・アプリケーションによって作成されます。FileBrowse オブジェクトは、ファイル上のブラウズ・オペレーションを表します。特定のファイルに対して常に複数のアクティブ・ブラウズが可能であり、各ブラウズは REQID で区別されます。メソッドは、次の API オペレーションを実行するために FileBrowse オブジェクトに対してインスタンス化することができます。

- ENDBR
- READNEXT
- READPREV
- RESETBR

FileBrowse オブジェクトは、ユーザー・アプリケーションによって明示的にインスタンス化されません。STARTBR オペレーションを実行するメソッドによって作成され、ユーザー・クラスに戻されます。

以下の表では、JCICS クラスとメソッドが、CICS ファイルおよび索引のタイプごとに **EXEC CICS** API コマンドにどのようにマップされるかを示しています。これらの表では、JCICS クラスとメソッドは `class.method()` の形式で示されます。例えば、KeyedFile.read() は、KeyedFile クラスの read() メソッドを参照します。

最初の表では、キー付きファイルのクラスとメソッドを示しています。

表 3. キー付きファイルのクラスとメソッド

KSDS 1 次または 2 次索引のクラスとメソッド	**ESDS 2 次索引のクラスとメソッド**	**CICS File API コマンド**
KeyedFile.read()	KeyedFile.read()	**READ**
KeyedFile.readForUpdate()	KeyedFile.readForUpdate()	**READ UPDATE**
KeyedFile.readGeneric()	KeyedFile.readGeneric()	**READ GENERIC**
KeyedFile.rewrite()	KeyedFile.rewrite()	**REWRITE**
KSDS.write()	KSDS.write()	**WRITE**

表 3. キー付きファイルのクラスとメソッド (続き)

KSDS 1 次または 2 次索引のクラスとメソッド	ESDS 2 次索引のクラスとメソッド	CICS File API コマンド
KSDS.delete()		**DELETE**
KSDS.deleteGeneric()		**DELETE GENERIC**
File.unlock()	File.unlock()	**UNLOCK**
KeyedFile.startBrowse()	KeyedFile.startBrowse()	**START BROWSE**
KeyedFile.startGenericBrowse()	KeyedFile.startGenericBrowse()	**START BROWSE GENERIC**
KeyedFileBrowse.next()	KeyedFileBrowse.next()	**READNEXT**
KeyedFileBrowse.previous()	KeyedFileBrowse.previous()	**READPREV**
KeyedFileBrowse.reset()	KeyedFileBrowse.reset()	**RESET BROWSE**
FileBrowse.end()	FileBrowse.end()	**END BROWSE**

次の表は、キー付きでないファイルのクラスとメソッドを示しています。ESDS と RRDS は 1 次索引によってアクセスされます。

ESDS 1 次索引のクラスとメソッド	RRDS 1 次索引のクラスとメソッド	CICS File API コマンド
ESDS.read()	RRDS.read()	**READ**
ESDS.readForUpdate()	RRDS.readForUpdate()	**READ UPDATE**
ESDS.rewrite()	RRDS.rewrite()	**REWRITE**
ESDS.write()	RRDS.write()	**WRITE**
	RRDS.delete()	**DELETE**
File.unlock()	File.unlock()	**UNLOCK**
ESDS.startBrowse()	RRDS.startBrowse()	**START BROWSE**
ESDS_Browse.next()	RRDS_Browse.next()	**READNEXT**
ESDS_Browse.previous()	RRDS_Browse.previous()	**READPREV**
ESDS_Browse.reset()	RRDS_Browse.reset()	**RESET BROWSE**
FileBrowse.end()	FileBrowse.end()	**END BROWSE**
ESDS.setXRBA()		

ファイルに書き込まれるデータは、Java バイト配列でなければなりません。

データはファイルから RecordHolder オブジェクトに読み取られます。ストレージは CICS によって提供され、プログラムの終わりに自動的に解放されます。

File メソッドには **KEYLENGTH** 値を指定する必要はありません。使用される長さは、渡されるキーの実際の長さです。FileBrowse オブジェクトが作成されると、startBrowse メソッドで指定されたキーの長さが入っています。この長さは、そのオブジェクトに対する以降のブラウズ要求で CICS に渡されます。

ブラウズ・オペレーションに **REQID** を指定する必要はありません。各ブラウズ・オブジェクトには、固有の REQID が含まれ、そのブラウズ・オブジェクトに対する以降のすべてのブラウズ要求に自動的に使用されます。

HTTP サービスおよび TCP/IP サービス

HttpHeader、NameValueData、および FormField クラスの getter は、HTTP ヘッダー、名前と値のペア、および該当する API コマンドのフォーム・フィールド値を戻します。

メソッド	JCICS クラス	EXEC CICS コマンド
get*()	CertificateInfo	EXTRACT CERTIFICATE / EXTRACT TCPIP
get*()	HttpRequest	EXTRACT WEB
getHeader()	HttpRequest	WEB READ HTTPHEADER
getFormField()	HttpRequest	WEB READ FORMFIELD
getContent()	HttpRequest	WEB RECEIVE
getQueryParm()	HttpRequest	WEB READ QUERYPARM
startBrowseHeader()	HttpRequest	WEB STARTBROWSE HTTPHEADER
getNextHeader()	HttpRequest	WEB READNEXT HTTPHEADER
endBrowseHeader()	HttpRequest	WEB ENDBROWSE HTTPHEADER
startBrowseFormField()	HttpRequest	WEB STARTBROWSE FORMFIELD
getNextFormField()	HttpRequest	WEB READNEXT FORMFIELD
endBrowseFormField()	HttpRequest	WEB ENDBROWSE FORMFIELD
startBrowseQueryParm()	HttpRequest	WEB STARTBROWSE QUERYPARM
getNextQueryParm()	HttpRequest	WEB READNEXT QUERYPARM
endBrowseQueryParm()	HttpRequest	WEB ENDBROWSE QUERYPARM
writeHeader()	HttpResponse	WEB WRITE
getDocument()	HttpResponse	WEB RETRIEVE
getCurrentDocument()	HttpResponse	WEB RETRIEVE
sendDocument()	HttpResponse	WEB SEND

注: HttpRequest オブジェクトを取得するには、メソッド get HttpRequestInstance() を使用してください。

CICS Web サポートによって処理される各着信 HTTP 要求には、HTTP ヘッダーが含まれています。要求で POST HTTP verb を使用する場合、文書データも含まれます。CICS Web サポートによって生成される各応答 HTTP 要求には、HTTP ヘッダーと文書データが含まれています。

これを処理するために、JCICS は次の Web および TCP/IP サービスを提供します。

HTTP ヘッダー
HttpRequest クラスを使用して HTTP ヘッダーを調べることができます。GET モードの HTTP では、クライアントが HTTP フォームに入力し、送信ボタンを選択した場合、照会ストリングが送信されます。

SSL CICS Web サポートは、TcpipRequest クラスを提供します。これは、要求を送信したクライアントに関する詳細情報と、SSL サポートに関する基本情

報を取得するために、HttpRequest によって拡張されます。SSL 証明書が提供される場合、CertificateInfo クラスを使用して詳細に調べることができます。

文書 文書がサーバーに公開される (HTTP POST) 場合、CICS 文書として提供されます。HttpRequest クラスの getDocument() メソッドを呼び出すことによって、この文書にアクセスできます。既存文書の処理の詳細については、67 ページの『文書サービス』を参照してください。

要求から生じる HTTP クライアント Web コンテンツを提供するために、サーバー・プログラマーは、Document Services API を使用して CICS 文書を作成し、sendDocument() メソッドを呼び出す必要があります。

CICS Web サポートについて詳しくは、「インターネット・ガイド」の『インターネットの概要』を参照してください。 JCICS Web クラスについて詳しくは、「*JCICS Class Reference*」を参照してください。

プログラム・サービス

JCICS は、CICS プログラム制御コマンド LINK、RETURN、XCTL、および SUSPEND をサポートします。

Java プログラムが既存の CICS アプリケーション・データにアクセスできるようにするツールについては、「CICS での Java アプリケーション」の『Java からの構造化データとの対話』を参照してください。

表 4 は、CICS プログラム制御コマンドにマップされるメソッドと JCICS クラスをリストしています。

表 4. メソッド、JCICS クラスおよび CICS コマンド間の関係

メソッド	JCICS クラス	EXEC CICS コマンド
link()	Program	LINK
setNextTransaction()、setNextCOMMAREA()、setNextChannel()	TerminalPrincipalFacility	RETURN
xctl()	Program	XCTL
	サポートされない	SUSPEND

LINK および XCTL

link() および xctl() メソッドを使用して、CICS に対して定義される別のプログラムに制御を移動することができます。ターゲット・プログラムは、CICS でサポートされる任意の言語にすることができます。

xctl() メソッドを使用する場合、正常に完了した場合であっても、TransferOfControlException が発行側プログラムにスローされます。

RETURN

このコマンドの疑似会話型の側面のみがサポートされます。Return に対する CICS 呼び出しを行う必要はありません。アプリケーションは通常どおり終了できます。疑似会話型機能は、TerminalPrincipalFacility クラスのメソッドでサポートされます。setNextTransaction() は、RETURN の TRANSID オプションの使用に相当します。setNextCOMMAREA() は、COMMAREA オプションの使用に相当します。一方、setNextChannel() は、CHANNEL オプションの使用に相当しま

す。これらのメソッドは、プログラムの実行中にいつでも呼び出すことができ、プログラムが終了するときに有効になります。

注: 指定される COMMAREA の長さは、CICS の LENGTH 値として使用されます。COMMAREA が任意の 2 つの CICS サーバー (製品/バージョン/リリースの任意の組み合わせ) 間で渡される場合、この値は 32,500 バイトを超えてはなりません。この制限により、32,500 バイトの COMMAREA とヘッダー用のスペースが可能になります。

スケジューリング・サービス

JCICS は、CICS スケジューリング・サービスをサポートします。これにより、タスク用に保管されたデータを取り出し、インターバル制御要求を取り消し、指定された時間にタスクを開始することができます。

メソッド	JCICS クラス	EXEC CICS コマンド
cancel()	StartRequest	CANCEL
retrieve()	タスク	RETRIEVE
issue()	StartRequest	START

Task.retrieve() メソッドによって取り出される内容を定義するには、java.util.BitSet オブジェクトを使用します。com.ibm.cics.server.RetrieveBits クラスは、BitSet オブジェクトで設定できる次のビットを定義します。

- RetrieveBits.DATA
- RetrieveBits.RTRANSID
- RetrieveBits.RTERMID
- RetrieveBits.QUEUE

これらは、**EXEC CICS** RETRIEVE コマンドのオプションに対応します。

Task.retrieve() メソッドは、RetrieveBits の設定に応じて、単一の呼び出しで最大 4 つの情報を取り出します。DATA、RTRANSID、RTERMID および QUEUE データは、RetrievedData オブジェクトに置かれ、このオブジェクトは RetrievedDataHolder オブジェクトに保持されます。次の例では、データと transid を取り出します。

```
BitSet bs = new BitSet();
bs.set(RetrieveBits.DATA, true);
bs.set(RetrieveBits.RTRANSID, true);
RetrievedDataHolder rdh = new RetrievedDataHolder();
t.retrieve(bs, rdh);
byte[] inData = rdh.value.data;
String transid = rdh.value.transId;
```

直列化サービス

JCICS は、タスクによるリソースの使用をスケジュールできる CICS 直列化サービスをサポートします。

メソッド	JCICS クラス	EXEC CICS コマンド
dequeue()	SynchronizationResource	DEQ
enqueue()、tryEnqueue()	SynchronizationResource	ENQ

ストレージ・サービス

CICS サービスを使用した明示的なストレージ管理 (**EXEC CICS** GETMAIN など) はサポートされません。標準の Java ストレージ管理機能で、タスク専用ストレージのニーズを十分に満たすことができます。

タスク間のデータの共用は、CICS リソースを使用して行われなければなりません。

名前は一般的に、Java ストリングまたはバイト配列として表されます。これらが必要な長さであることを確認する必要があります。

一時記憶域キュー・サービス

JCICS は、CICS 一時記憶コマンド DELETEQ TS、READQ TS、および WRITEQ TS をサポートします。

JCICS メソッドと EXEC CICS コマンド間の対話

Java プログラムが既存の CICS アプリケーション・データにアクセスできるようにするツールについては、「CICS での Java アプリケーション」の『Java からの構造化データとの対話』を参照してください。

表 5 は、CICS 一時記憶コマンドにマップされるメソッドと JCICS クラスをリストしています。

表 5. メソッド、JCICS クラスおよび CICS コマンド間の関係

メソッド	JCICS クラス	EXEC CICS コマンド
delete()	TSQ	DELETEQ TS
readItem()、readNextItem()	TSQ	READQ TS
writeItem()、rewriteItem() writeItemConditional() rewriteItemConditional()	TSQ	WRITEQ TS

DELETEQ TS

TSQ クラスの delete() メソッドを使用して一時記憶域キュー (TSQ) を削除することができます。

READQ TS

CICS INTO オプションは Java プログラムではサポートされません。TSQ クラスの readItem() および readNextItem() メソッドを使用して、TSQ から特定の項目を読み取ることができます。これらのメソッドは、引数の 1 つとして ItemHolder オブジェクトを取ります。これには、バイト配列で読み取られるデータが含まれます。このバイト配列のストレージは、CICS によって作成され、プログラムの終わりにガーベッジ・コレクションされます。

WRITEQ TS

Java バイト配列で一時記憶域キューに書き込まれるデータを提供する必要があります。NOSPACE 条件が検出される場合、writeItem() および rewriteItem() メソッドは中断し、データをキューに書き込むためのスペースが使用可能になるまで待機します。writeItemConditional() および rewriteItemConditional() メソッドは、NOSPACE 条件の場合に中断しませんが、この条件を NoSpaceException としてアプリケーションに即時に戻します。

端末サービス

JCICS は、以下の CICS 端末サービス・コマンドをサポートします。

メソッド	JCICS クラス	EXEC CICS コマンド
converse()	TerminalPrincipalFacility	CONVERSE
	サポートされない	HANDLE AID
receive()	TerminaPrincipalFacility	RECEIVE
send()	TerminaPrincipalFacility	SEND
	サポートされない	WAIT TERMINAL

タスクに基本機構として端末がある場合、CICS は、標準出力と標準エラー・ストリームとして使用できる 2 つの Java PrintWriter を自動的に作成します。これらは、タスクの端末にマップされます。out と err と呼ばれる 2 つのストリームは、Task オブジェクトのパブリック・ファイルであり、System.out や System.err と同様に使用できます。

端末に送られるデータは、Java バイト配列で提供されなければなりません。データは、端末から DataHolder オブジェクトに読み取られます。 CICS は戻されたデータのストレージを提供し、そのストレージは、プログラムが終了すると割り振り解除されます。

一時データ・キュー・サービス

JCICS は、CICS 一時データ・コマンド DELETEQ TD、READQ TD、および WRITEQ TD をサポートします。INTO オプションを除くすべてのオプションがサポートされます。

JCICS メソッドと EXEC CICS コマンド間の対話

Java プログラムが既存の CICS アプリケーション・データにアクセスできるようにするツールについては、「CICS での Java アプリケーション」の『Java からの構造化データとの対話』を参照してください。

表 6 は、CICS 一時データ・コマンドにマップされるメソッドと JCICS クラスをリストしています。

表 6. メソッド、JCICS クラスおよび CICS コマンド間の関係

メソッド	JCICS クラス	EXEC CICS コマンド
delete()	TDQ	DELETEQ TD
readData()、readDataConditional()	TDQ	READQ TD
writeData()	TDQ	WRITEQ TD

DELETEQ TD

TDQ クラスの delete() メソッドを使用して一時データ・キュー (TDQ) を削除することができます。

READQ TD

CICS INTO オプションは Java プログラムではサポートされません。TDQ クラスの readData() または readDataConditional() メソッドを使用して TDQ から読

み取ることができます。これらのメソッドは、バイト配列で読み取られるデータが入っている DataHolder オブジェクトのインスタンスをパラメーターとして取ります。このバイト配列のストレージは、CICS によって作成され、プログラムの終わりにガーベッジ・コレクションされます。

readDataConditional() メソッドは、CICS NOSUSPEND ロジックを駆動します。QBUSY 条件が検出されると、QueueBusyException として即時にアプリケーションに戻されます。

readData() メソッドは、別のタスクによって使用中のレコードにアクセスしようとするときに、コミットされたレコードがそれ以上ない場合は中断します。

WRITEQ TD

Java バイト配列で TDQ に書き込まれるデータを提供する必要があります。

作業単位 (UOW) サービス

JCICS は、CICS SYNCPOINT サービスをサポートします。

表 7. UOW サービスの JCICS と EXEC CICS コマンド間の関係

メソッド	JCICS クラス	EXEC CICS コマンド
commit()、rollback()	タスク	SYNCPOINT

Web サービス

JCICS は、アプリケーション内で Web サービスを操作するために使用できるすべての API コマンドをサポートします。

メソッド	JCICS クラス	EXEC CICS コマンド
invoke()	WebService	INVOKE WEBSERVICE
create()	SoapFault	SOAPFAULT CREATE
addFaultString()	SoapFault	SOAPFAULT ADD FAULTSTRING
addSubCode()	SoapFault	SOAPFAULT ADD SUBCODESTR
delete()	SoapFault	SOAPFAULT DELETE
create()	WSAEpr	WSAEPR CREATE
delete()	WSAContext	WSACONTEXT DELETE
set*()	WSAContext	WSACONTEXT BUILD
get*()	WSAContext	WSACONTEXT GET

次の例に、JCICS を使用して Web サービス要求を作成する方法を示します。

```
Channel requesterChannel = Task.getTask().createChannel("TestRequester");
                    Container appData = requesterChannel.createContainer("DFHWS-DATA");
                    byte[] exampleData = "ExampleData".getBytes();
                    appData.put(exampleData);

                    WebService requester = new WebService();
                    requester.setName("MyWebservice");
                    requester.invoke(requesterChannel, "myOperationName");

                    byte[] response = appData.get();
```

Web サービス要求の中で送受信されるアプリケーション・データを処理する際に、JZOS などのツールを使用して、構造化データを処理するためのクラスを自動的に生成できます。詳しくは、55 ページの『Java からの構造化データとの対話』を参照してください。また、Java を使用して XML の生成とコンシュームを直接行うこともできます。

JCICS 例外マッピング

Java では、CICS コマンドによって戻される条件は、Java 例外にマップされます。

表 8. Java 例外マッピング

CICS 条件	Java 例外	CICS 条件	Java 例外
ALLOCERR	AllocationErrorException	CBIDERR	InvalidControlBlockIdException
CCSIDERR	CCSIDErrorException	CHANNELERR	ChannelErrorException
CONTAINERERR	ContainerErrorException	DISABLED	FileDisabledException
DSIDERR	FileNotFoundException	DSSTAT	DestinationStatusChangeException
DUPKEY	DuplicateKeyException	DUPREC	DuplicateRecordException
END	EndException	ENDDATA	EndOfDataException
ENDFILE	EndOfFileException	ENDINPT	EndOfInputIndicatorException
ENQBUSY	ResourceUnavailableException	ENVDEFERR	InvalidRetrieveOptionException
EOC	EndOfChainIndicatorException	EODS	EndOfDataSetIndicatorException
EOF	EndOfFileIndicatorException	ERROR	ErrorException
EXPIRED	TimeExpiredException	FILENOTFOUND	FileNotFoundException
FUNCERR	FunctionErrorException	IGREQID	InvalidREQIDPrefixException
IGREQCD	InvalidDirectionException	ILLOGIC	LogicException
INBFMH	InboundFMHException	INVERRTERM	InvalidErrorTerminalException
INVEXITREQ	InvalidExitRequestException	INVLDC	InvalidLDCException
INVMPSZ	InvalidMapSizeException	INVPARTNSET	InvalidPartitionSetException
INVPARTN	InvalidPartitionException	INVREQ	InvalidRequestException
INVTSREQ	InvalidTSRequestException	IOERR	IOErrorException
ISCINVREQ	ISCInvalidRequestException	ITEMERR	ItemErrorException
JIDERR	InvalidJournalIdException	LENGERR	LengthErrorException
MAPERROR	MapErrorException	MAPFAIL	MapFailureException
NAMEERROR	NameErrorException	NODEIDERR	InvalidNodeIdException
NOJBUFSP	NoJournalBufferSpaceException	NONVAL	NotValidException
NOPASSBKRD	NoPassbookReadException	NOPASSBKWR	NoPassbookWriteException
NOSPACE	NoSpaceException	NOSPOOL	NoSpoolException
NOSTART	StartFailedException	NOSTG	NoStorageException
NOTALLOC	NotAllocatedException	NOTAUTH	NotAuthorisedException
NOTFND	RecordNotFoundException	NOTOPEN	NotOpenException
OPENERR	DumpOpenErrorException	OVERFLOW	MapPageOverflowException
PARTNFAIL	PartitionFailureException	PGMIDERR	InvalidProgramIdException
QBUSY	QueueBusyException	QIDERR	InvalidQueueIdException
QZERO	QueueZeroException	RDATT	ReadAttentionException

表 8. Java 例外マッピング (続き)

CICS 条件	Java 例外	CICS 条件	Java 例外
RETPAGE	ReturnedPageException	ROLLEDBACK	RolledBackException
RTEFAIL	RouteFailedException	RTESOME	RoutePartiallyFailedException
SELNERR	DestinationSelectionErrorException	SESSBUSY	SessionBusyException
SESSIONERR	SessionErrorException	SIGNAL	InboundSignalException
SPOLBUSY	SpoolBusyException	SPOLERR	SpoolErrorException
STRELERR	STRELERRException	SUPPRESSED	SuppressedException
SYMBOLERR	SymbolErrorException	SYSBUSY	SystemBusyException
SYSIDERR	InvalidSystemIdException	TASKIDERR	InvalidTaskIdException
TCIDERR	TCIDERRException	TEMPLATERR	TemplateErrorException
TERMERR	TerminalException	TERMIDERR	InvalidTerminalIdException
TOKENERR	TokenErrorException		
TRANSIDERR	InvalidTransactionIdException	TSIOERR	TSIOErrorException
UNEXPIN	UnexpectedInformationException	USERIDERR	InvalidUserIdException
WRBRK	WriteBreakException	WRONGSTAT	WrongStatusException

注: CICS コマンド WEB RECEIVE が、受信されたデータが非 HTTP メッセージ (TYPE=HTTPNO の設定による) であることを示す場合、`NonHttpDataException` が `getContent()` によってスローされます。

JCICS の使用

通常の Java クラスのように、JCICS ライブラリーからクラスを使用します。アプリケーションは必要なタイプの参照を宣言し、クラスの新しいインスタンスが `new` 演算子を使用して作成されます。

このタスクについて

基礎の CICS リソースの名前を指定するために、setName メソッドを使用して CICS リソースに名前を付けます。リソースを作成した後、標準の Java 構成体を使用してオブジェクトを操作することができます。宣言されたオブジェクトのメソッドを通常の方法で呼び出すことができます。クラスごとにサポートされるメソッドの詳細は、提供される Javadoc で入手可能です。

CICS Java プログラムでファイナライザーを使用しないでください。ファイナライザーを推奨しない理由については、Java Diagnostics Guide を参照してください。

System.exit() 呼び出しを発行することによって CICS Java プログラムを終了しないでください。 Java アプリケーションが CICS で実行される場合、Java ラッパーと呼ばれる別の Java プログラムを使用して public static void main() メソッドが呼び出されます。ラッパーを使用する場合、CICS は、Java アプリケーションの環境を初期化し、さらに重要なことに、アプリケーションの存続中に使用されるすべてのプロセスをクリーンアップします。 JVM を終了すると、クリーン戻りコードが 0 であっても、このクリーンアップ・プロセスは実行できず、データの不整合が生じる可能性があります。アプリケーションが JVM サーバーで実行中に System.exit() を使用すると、その JVM サーバーが終了し、CICS が即時に静止します。

手順

1. main メソッドを書き込みます。 CICS は、PROGRAM リソースの JVMCLASS 属性で指定されるクラスで、main(CommAreaHolder) の署名を使用するメソッドに制御を渡そうとします。このメソッドが見つからない場合、CICS は、main(String[]) メソッドを呼び出そうとします。
2. JCICS を使用してオブジェクトを作成するために、以下の手順を実行します。
   a. 参照を宣言します。
      ```
      TSQ tsq;
      ```
   b. new 演算子を使用してオブジェクトを作成します。
      ```
      tsq = new TSQ()
      ```
   c. setName メソッドを使用して、オブジェクトに名前を指定します。
      ```
      tsq.setName("JCICSTSQ");
      ```
3. オブジェクトを使用して、CICS と対話します。

例

この例では、TSQ オブジェクトを作成し、作成したばかりの一時記憶域キュー・オブジェクトで delete メソッドを呼び出し、キューが空の場合はスローされた例外をキャッチする方法を示しています。

```
// Define a package name for the program
package unit_test;

// Import the JCICS package
import com.ibm.cics.server.*;

// Declare a class for a CICS application
public class JCICSTSQ
{

    // The main method is called when the application runs
    public static void main(CommAreaHolder cah)
    {

         try
         {
             // Create and name a Temporary Storage queue object
             TSQ tsq = new TSQ();
             tsq.setName("JCICSTSQ");

             // Delete the queue if it exists
             try
             {
                   tsq.delete();
             }
             catch(InvalidQueueIdException e)
             {
                  // Absorb QIDERR
                  System.out.println("QIDERR ignored!");
             }

             // Write an item to the queue
             String transaction = Task.getTask().getTransactionName();
             String message = "Transaction name is - " + transaction;
             tsq.writeItem(message.getBytes());

         }
         catch(Throwable t)
         {
             System.out.println("Unexpected Throwable: " + t.toString());
```

```
        }

        // Return from the application
        return;
    }
}
```

Java の制約事項

Java アプリケーションを開発する場合は、アプリケーションが CICS で実行される場合の問題を回避するために、特定の制約事項に従う必要があります。

CICS で実行される Java アプリケーションは、次の制約事項に従います。

- ご使用の Java アプリケーションでは System.exit() メソッドを使用できません。このメソッドを使用すると、アプリケーションは異常終了します。JVM サーバーと CICS もシャットダウンします。
- OSGi バンドルのアクティベーター・クラスの JCICS API 呼び出しを使用できません。
- バンドル・アクティベーターの start および stop メソッドは、妥当な時間内に戻らなければなりません。

Java アプリケーションからのデータへのアクセス

DB2 および VSAM のデータのアクセスと更新を行うことができる Java アプリケーションを作成できます。または、他の言語のプログラムにリンクして、DB2、VSAM、および IMS にアクセスすることができます。

CICS のデータにアクセスするための Java アプリケーションを作成する際に、次のいずれかの手法を使用できます。CICS リカバリー・マネージャーがデータ保全性を維持します。

リレーショナル・データへのアクセス

次のいずれかの方法を使用して、DB2 のリレーショナル・データにアクセスするための Java アプリケーションを作成できます。

- 構造化照会言語 (SQL) コマンドを使用してデータにアクセスするプログラムにリンクする場合は、**JCICS LINK** コマンド、または CCI Connector for CICS TS。
- 適切なドライバーが使用可能な場合は、Java Data Base Connectivity (JDBC) または Structured Query Language for Java (SQLJ) 呼び出しを使用して、データに直接アクセスします。DB2 に適切な JDBC ドライバーが使用可能です。 JDBC および SQLJ アプリケーション・プログラミング・インターフェースの使用について詳しくは、「DB2 Guide」の『Using JDBC and SQLJ to access DB2 data from Java programs』を参照してください。
- 基礎のアクセス機構として JDBC または SQLJ を使用する JavaBeans。このような JavaBeans を開発するには、適切な Java 統合開発環境 (IDE) を使用できます。
- エンティティー Bean。CICS は、CICS で実行されるエンティティー Bean をサポートするのではなく、他の EJB サーバーで実行されるエンティティー Bean

へのアクセスをサポートします。例えば、CICS エンタープライズ Bean は、WebSphere Application Server で実行中のエンティティー Bean を使用して、z/OS 上の DB2 にアクセスできます。

DL/I データへのアクセス

IMS の DL/I データにアクセスするには、Java アプリケーションで **JCICS LINK** コマンドを使用して、EXEC DLI コマンドを発行してデータにアクセスする中間プログラムにリンクする必要があります。

VSAM データへのアクセス

VSAM データにアクセスするには、Java アプリケーションで次のいずれかの方法を使用できます。

- VSAM に直接アクセスする場合は、JCICS ファイル制御クラス。
- CICS ファイル制御コマンドを発行してデータにアクセスするプログラムにリンクする場合は、**JCICS LINK** コマンドまたは CCI Connector for CICS TS。

CICS における Java アプリケーションからの接続性

CICS 環境内の Java プログラムは、TCP/IP ソケットをオープンし、外部プロセスと通信することができます。Java プログラムをゲートウェイとして使用すると、他の言語の CICS プログラムからは使用できない可能性がある他のエンタープライズ・アプリケーションに接続することができます。例えば、リモート・サーブレットまたはデータベースと通信する Java プログラムを作成できます。

この接続が CICS に統合されて、分散トランザクションや ID 伝搬などのエンタープライズ・サービス品質を提供する場合があります。また、CICS によって提供される分散トランザクションやその他のサービスなしに接続を使用できる場合もあります。必要な接続のタイプによっては、CICS で本来はサポートされないエンタープライズ・アプリケーションとの接続を可能にするサード・パーティー・ベンダー製品が使用できる場合があります。

一般に、CICS 環境における JVM の機能は、バッチ・モード JVM とほぼ同じです。バッチ・モード JVM は、CICS 環境の外部ではスタンドアロン・プロセスとして実行され、通常は、UNIX システム・サービスのコマンド行から、または JCL ジョブで開始されます。バッチ・モード JVM で作動可能な大部分のアプリケーションは、同じ範囲で CICS における JVM でも実行できます。例えば、サード・パーティーの JDBC ドライバーを使用して IBM 以外のデータベースと通信するバッチ・モード Java アプリケーションを作成する場合、同じアプリケーションがおそらく、CICS における JVM で作動します。ベンダー提供のコード (IBM 以外の JDBC ドライバーなど) を CICS における JVM で使用したい場合は、ベンダーに問い合わせて、そのコードが CICS における JVM で実行されることをサポートするかどうかを判別してください。

一部のバッチ・モード・アプリケーションは、CICS における JVM でホスティングされる場合は動作が異なる可能性があります。これは、CICS が JVM を再利用する方法のためです。静的変数に保管されるデータはすべて、JVM の各使用の間で持続されます。 CICS における Java アプリケーションの動作について詳しくは、35 ページの『CICS における Java ランタイム環境』を参照してください。

CICS 環境における JVM で実行されるバッチ・モード・アプリケーションは、通常、CICS の機能を利用しません。例えば、CICS の Java プログラムが、サード・パーティーの JDBC ドライバーを使用して IBM 以外のデータベース内のレコードを更新する場合、CICS はこのアクティビティーを認識せず、現行の CICS トランザクションに更新を組み込もうとしません。

第 4 章 Java サポートのセットアップ

CICS 領域で Java をサポートするための基本的なセットアップ・タスクを実行します。

始める前に

CICS に必要な Java コンポーネントは、製品のインストール時にセットアップされます。「インストール・ガイド」の『Java コンポーネントのインストール検査』の情報を使用して、Java コンポーネントが正しくインストールされていることを確認する必要があります。

手順

1. JVMPROFILEDIR システム初期設定パラメーターを、CICS 領域で使用される JVM プロファイルを保管する先の z/OS UNIX 内の適切なディレクトリーに設定します。 詳しくは、『JVM プロファイルのロケーションの設定』を参照してください。
2. Java アプリケーションを実行できる十分なメモリーが CICS 領域にあることを確実にします。 詳しくは、88 ページの『Java のメモリー制限の設定』を参照してください。
3. JVM の作成に必要な JVM プロファイル、ディレクトリー、およびファイルを含めて、z/OS UNIX に保持されているリソースにアクセスする許可を CICS 領域に付与します。 詳しくは、89 ページの『z/OS UNIX ディレクトリーおよびファイルに対するアクセス権の CICS 領域への付与』を参照してください。

タスクの結果

これで、Java をサポートする CICS 領域をセットアップしました。

次のタスク

既存の Java アプリケーションをアップグレードする場合は、Upgrading の手引きに従ってください。 JVM サーバーまたはプールされた JVM を作成して Java ワークロードを実行するには、93 ページの『第 5 章 JVM を使用するアプリケーションの有効化』を参照してください。

JVM プロファイルのロケーションの設定

CICS は、**JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーから、JVM プロファイルをロードします。**JVMPROFILEDIR** パラメーターの値を新規ロケーションに変更し、提供されたサンプル JVM プロファイルをこのディレクトリーにコピーする必要があります。その結果、それらのプロファイルを使用してインストールを検証できます。

始める前に

USSHOME システム初期設定パラメーターは、z/OS UNIX に CICS ファイル用のルート・ディレクトリーを指定する必要があります。

このタスクについて

CICS 提供のサンプル JVM プロファイルは、CICS インストール処理時にシステムに合わせてカスタマイズされるので、これらのプロファイルを即時に使用してインストールを検証できます。独自の Java アプリケーション用にこれらのファイルのコピーをカスタマイズすることができます。

JVM プロファイルでの使用に適している設定は、CICS のリリースごとに異なる可能性があるので、問題判別を容易にするために、すべてのプロファイルのベースとして CICS 提供のサンプルを使用してください。アップグレード情報を確認して、JVM プロファイルの新規オプションまたは変更されたオプションを見つけてください。

手順

1. JVMPROFILEDIR システム初期設定パラメーターを、CICS 領域で使用される JVM プロファイルを保管する先の z/OS UNIX のロケーションに設定します。指定する値の長さは 240 文字までにすることができます。

   JVMPROFILEDIR システム初期設定パラメーターに指定された設定は、`/usr/lpp/cicsts/cicsts42/JVMProfiles` です。これは、サンプル JVM プロファイルのインストール場所です。このディレクトリーは、カスタマイズされた JVM プロファイルを安全に保管できる場所ではありません。プログラムの保守時にサンプル JVM プロファイルが上書きされると、変更内容が失われる危険があるためです。したがって、必ず **JVMPROFILEDIR** を変更して、JVM プロファイルを保管できる別の z/OS UNIX ディレクトリーを指定する必要があります。JVM プロファイルをカスタマイズする必要があるユーザーに該当する許可を付与できるディレクトリーを選択してください。

2. CICS 提供のサンプル JVM プロファイル DFHJVMPR、DFHJVMAX、DFHOSGI、および DFHJVMCD を、インストール・ロケーションから z/OS UNIX ディレクトリーにコピーします。DFHJVMCD プロファイルは、厳密にはサンプル JVM プロファイルではありませんが、内部の CICS Java トランザクションに必要であり、共用クラス・キャッシュの管理にも必要です。

   CICS をインストールすると、CICS 提供のサンプル JVM プロファイルは、`/usr/lpp/cicsts/cicsts42/JVMProfiles` ディレクトリーに置かれます。`/usr/lpp/cicsts/cicsts42` ディレクトリーは、z/OS UNIX で CICS ファイル用のインストール・ディレクトリーです。このディレクトリーは、DFHISTAR インストール・ジョブの **USSDIR** パラメーターで指定されます。

Java のメモリー制限の設定

Java アプリケーションには、他の言語で作成されたプログラムよりも多くのメモリーが必要です。Java アプリケーションを実行するために使用できる、十分なストレージとメモリーが CICS および Java にあることを確認する必要があります。

このタスクについて

Java は、16 MB 境界より下のストレージ、31 ビット・ストレージ、および 64 ビット・ストレージを使用します。JVM ヒープに必要なストレージは、EDSA ではなく MVS 内の CICS 領域ストレージから取得されます。

手順

1. z/OS **MEMLIMIT** パラメーターが適切な値に設定されていることを確認します。このパラメーターは、CICS アドレス・スペースが使用できる 64 ビット・ストレージの量を制限します。CICS は 64 ビット・バージョンの Java を使用するので、**MEMLIMIT** が、64 ビット・ストレージを使用するこの機能やその他の CICS 機能に十分な大きい値に設定されていることを確認する必要があります。

   次のトピックを参照してください。
   - 192 ページの『プールされた JVM のストレージ所要量の計算』
   - 183 ページの『JVM サーバーのストレージ所要量の計算』
   - 「パフォーマンス・ガイド」の『MEMLIMIT の見積もり、確認、および設定』
2. 開始ジョブ・ストリームの **REGION** パラメーターに、Java の実行に十分な大きさの値が指定されていることを確認します。 それぞれの JVM に、アプリケーション (ジャストインタイム・コンパイル・コードを含む) を実行するための 16 MB 境界より下のストレージ、およびパラメーターを CICS に渡すための作業用ストレージが必要です。

z/OS UNIX ディレクトリーおよびファイルに対するアクセス権の CICS 領域への付与

CICS では、z/OS UNIX 内のディレクトリーとファイルへのアクセスが必要です。各 CICS 領域には、インストール時に z/OS UNIX ユーザー ID (UID) が割り当てられます。これらの領域は、z/OS UNIX グループ ID (GID) を割り当てられた RACF グループに接続されます。この UID および GID を使用して、CICS 領域が z/OS UNIX 内のディレクトリーおよびファイルにアクセスする権限を付与します。

始める前に

自分が z/OS UNIX のスーパーユーザーであるか、ディレクトリーおよびファイルの所有者であることを確認してください。ディレクトリーおよびファイルの所有者は、最初は、製品をインストールしたシステム・プログラマーの UID として設定されます。ディレクトリーおよびファイルの所有者は、インストール時に GID が割り当てられた RACF グループに接続されなければなりません。所有者は、この RACF グループをデフォルト・グループ (DLFTGRP) として指定するか、補足グループの 1 つとして接続することができます。

このタスクについて

z/OS UNIX システム・サービスでは、個々の CICS 領域が単一の UNIX ユーザーとして扱われます。 z/OS UNIX ディレクトリーおよびファイルにアクセスするユーザー権限をさまざまな方法で付与することができます。例えば、CICS 領域の接続先の RACF グループに対して、ディレクトリーまたはファイルに関する適切なグル

ープ権限を付与できます。このオプションは、実稼働環境に最適な場合があり、以下の手順で説明しています。

手順

1. CICS 領域からのアクセスが必要となる z/OS UNIX 内のディレクトリーおよびファイルを識別します。

デフォルト・ディレクトリー	権限	説明
/usr/lpp/java/J6.0.1_64/bin	読み取りおよび実行	IBM 64-bit SDK for z/OS, Java テクノロジー・エディション ディレクトリー
/usr/lpp/java/J6.0.1_64/bin/j9vm	読み取りおよび実行	IBM 64-bit SDK for z/OS, Java テクノロジー・エディション ディレクトリー
/usr/lpp/cicsts/cicsts42	読み取りおよび実行	z/OS UNIX 上の CICS ファイル用のインストール・ディレクトリー。このディレクトリー内のファイルには、サンプル・プロファイルと CICS 提供の JAR ファイルが含まれます。
/u/*CICS region userid*	読み取り、書き込み、および実行	CICS 領域の作業ディレクトリー。このディレクトリーには、JVM からの入力、出力、およびメッセージが入っています。
/usr/lpp/cicsts/cicsts42/JVMProfiles/	読み取りおよび実行	**JVMPROFILEDIR** システム初期設定パラメーターで指定された、CICS 領域の JVM プロファイルが入っているディレクトリー。

2. ディレクトリーおよびファイルをリストして、権限を表示します。 最初のディレクトリーに移動し、次の UNIX コマンドを発行します。

   ```
   ls -la
   ```

   現行ディレクトリーが CICSHT## のホーム・ディレクトリーであるときに、このコマンドが z/OS UNIX システム・サービスのシェル環境で発行されると、次の例のようなリストが表示される場合があります。

   ```
   /u/cicsht##:>ls -la
   total 256
   drwxr-xr-x   2 CICSHT## CICSTS42     8192 Mar 15  2008 .
   drwx------   4 CICSHT## CICSTS42     8192 Jul  4 16:14 ..
   -rw-------   1 CICSHT## CICSTS42     2976 Dec  5  2010 Snap0001.trc
   -rw-r--r--   1 CICSHT## CICSTS42     1626 Jul 16 11:15 dfhjvmerr
   -rw-r--r--   1 CICSHT## CICSTS42        0 Mar 15  2010 dfhjvmin
   -rw-r--r--   1 CICSHT## CICSTS42      458 Oct  9 14:28 dfhjvmout
   /u/cicsht##:>
   ```

3. グループ権限を使用してアクセス権限を付与する場合は、各ディレクトリーおよびファイルのグループ権限が、CICS がリソースに対して必要とするアクセス・レベルを認可するものであることを確認します。 r、w、x、および - という文字を使用して 3 組の権限が指定されます。これらの文字は、読み取り、書き込み、実行、およびなしを表し、コマンド行の左側の列の 2 文字目から表示されます。最初の組は所有者権限、2 番目の組はグループ権限、3 番目の組はその他の権限です。 前述の例では、所有者には dfhjvmerr、dfhjvmin、および

dfhjvmout に対する読み取りおよび書き込み権限がありますが、グループおよびその他すべてには、読み取り権限しかありません。

4. 特定のリソースのグループ権限を変更したい場合は、 UNIX コマンド chmod を使用します。 次の例では、指定されたディレクトリーおよびそのサブディレクトリーとファイルのグループ権限を、読み取り、書き込み、および実行に設定します。 -R は、すべてのサブディレクトリーおよびファイルに権限を再帰的に適用します。

   ```
   chmod -R g=rwx directory
   ```

   次の例では、指定されたファイルのグループ権限を読み取りおよび実行に設定します。

   ```
   chmod g+rx filename
   ```

   次の例では、指定された 2 つのファイルでグループに対する書き込み権限をオフにします。

   ```
   chmod g-w filename filename
   ```

   上記のすべての例で、g はグループ権限を指定します。その他に訂正したい権限がある場合、 u はユーザー (所有者) 権限を指定し、o はその他の権限を指定します。

5. CICS 領域から z/OS UNIX にアクセスするために選択された RACF グループに対して、リソースごとにグループ権限を割り当てます。個々のディレクトリーとそのサブディレクトリー、およびそこに含まれるファイルに対して、グループ権限を割り当てる必要があります。 次の UNIX コマンドを入力します。

   ```
   chgrp -R GID directory
   ```

   GID は RACF グループの GID の数値であり、*directory* は、CICS 領域の権限を付与する対象となるディレクトリーの絶対パスです。 例えば、/usr/lpp/cicsts/cicsts42 ディレクトリーにグループ権限を割り当てるには、次のコマンドを使用します。

   ```
   chgrp -R GID /usr/lpp/cicsts/cicsts42
   ```

   CICS 領域のユーザー ID は RACF グループに接続されるため、CICS 領域は、これらすべてのディレクトリーおよびファイルに関する適切な権限を持つことになります。

タスクの結果

これで、CICS には、Java アプリケーションを実行するために z/OS UNIX 内のディレクトリーとファイルにアクセスする適切な権限があることが確実になりました。

ファイルの移動または新規ファイルの作成などにより、セットアップ中の CICS 機能を変更した場合は、新規のファイルまたは移動したファイルに関する CICS 領域のアクセス権限が保持されるように、この手順を忘れずに繰り返してください。

次のタスク

サンプル・プログラムおよびプロファイルを使用して、Java サポートが正しくセットアップされたことを確認します。

第 5 章 JVM を使用するアプリケーションの有効化

非 Java アプリケーションの場合と同様に、 CICS では JVM で Java プログラムを実行するために必要なリソースを定義する必要があります。また、アプリケーションのクラスを検出する場所も定義する必要があります。

PROGRAM リソースを作成することによって、プールされた JVM または JVM サーバーで標準 Java アプリケーションを実行できます。アプリケーションがスレッド・セーフでない場合のみ、プールされた JVM を使用してください。それ以外の場合は、Java アプリケーションに JVM サーバーを使用します。

CORBA ステートレス・オブジェクトおよびエンタープライズ Bean には独自の PROGRAM リソースはありませんが、要求プロセッサー・プログラムで指定されたプロファイルを使用します。CORBA ステートレス・オブジェクトおよびエンタープライズ Bean は、プールされた JVM でのみ実行できます。

JVM サーバーのセットアップ

JVM サーバーで Java アプリケーションまたは Axis2 を実行するには、CICS リソースをセットアップし、JVM にオプションを渡す JVM プロファイルを作成する必要があります。

このタスクについて

JVM サーバーは、単一の JVM でさまざまな Java アプリケーションに対する複数の並行要求を処理できます。JVMSERVER リソースは、CICS における JVM サーバーを表します。このリソースは、JVM のオプション、Language Environment エンクレーブに値を提供するプログラム、およびスレッドの限度を指定する JVM プロファイルを定義します。JVM サーバーは、異なるタイプのワークロードを実行できます。

- OSGi バンドルとしてパッケージされるアプリケーションを実行するように、JVM サーバーを構成できます。
- Axis2 SOAP エンジンを使用して Web サービスの SOAP 処理を実行するように、JVM サーバーを構成できます。

JVM サーバーは両方のタイプのワークロードを実行することはできないので、ワークロードのタイプごとに JVM プロファイルが提供されます。これらのプロファイルに加える変更はすべて、そのプロファイルを使用するすべての JVM サーバーに適用されます。DFHOSGI JVM プロファイルには、JVM サーバーで OSGi フレームワークを実行するためのオプションが含まれています。 DFHOSGI プロファイルをカスタマイズする場合は、変更内容が、JVM サーバーを使用するすべての Java アプリケーションに適切であることを確認してください。DFHJVMAX JVM プロファイルには、JVM サーバーで Axis2 を実行するためのオプションが含まれています。

手順

1. JVM サーバー用の JVM プロファイルを作成します。 提供された該当するプロファイル DFHJVMAX または DFHOSGI をインストール・ディレクトリーから、**JVMPROFILEDIR** システム初期設定パラメーターで指定されたディレクトリーにコピーすることができます。コピーするプロファイルには追加の変更は必要ありませんが、ご使用の環境に合わせてオプションを編集できます。プロファイルの名前を変更する場合、その長さは 1 文字から 8 文字でなければなりません。

   ヒント: CICS Explorer で z/OS パースペクティブを使用すると、ディレクトリー間でプロファイルをコピーできます。

2. オプション: JVM プロファイルを開き、必要に応じてオプションを編集します。JVM サーバーではオプションのサブセットのみがサポートされているため、111 ページの『JVM プロファイル: オプションおよびサンプル』のオプションのリストをガイドとして使用してください。各パラメーターまたはプロパティーは別々の行で指定され、パラメーターまたはプロパティーの値は、行の終わりで区切られます。114 ページの『JVM プロファイルのコーディング規則』のコーディング規則に従ってください。

   DFHOSGI プロファイルでクラスパス・オプションを指定しないでください。OSGi フレームワークにより、アプリケーションごとのクラスが置かれる場所が決まります。次の変更を加えることができます。

   a. JVM サーバーで必要なネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイルを含む任意のディレクトリーを指定するには、LIBPATH_SUFFIX オプションを使用します。 IBM またはベンダーによって提供されるミドルウェアおよびツールでは、DLL ファイルをライブラリー・パスに追加する必要がある場合があります。例えば、DB2 JDBC ドライバーを使用するには、DLL ファイルが必要です。

   b. OSGi の場合のみ、OSGI_BUNDLES オプションを使用して、OSGi フレームワークで実行したいミドルウェア・バンドルを指定します。 ミドルウェア・バンドルは、WebSphere MQ への接続などのシステム・サービスを実装するためのクラスを含む、OSGi バンドルの 1 つのタイプです。

   c. OSGi の場合のみ、OSGI_FRAMEWORK_TIMEOUT オプションを使用して、OSGi フレームワークの初期設定またはシャットダウンがタイムアウトになるまでに CICS が待機する秒数を指定します。デフォルト値は 60 秒です。 フレームワークで、指定された時間より長くかかる場合、JVM サーバーは正しく初期化またはシャットダウンできません。

   d. JVM からのメッセージ、トレース、および出力の宛先を変更します。dfhjvmtrc、stdin、stdout、stderr ファイルおよび Java メモリー・ダンプの名前と場所を変更できます。出力のインターリーブを避けるには、&JVMSERVER; シンボルを使用して、これらのファイルを各 JVM サーバーに固有のものにしてください。

3. 変更内容を JVM プロファイルに保管します。 JVM プロファイルは、EBCDIC で保管されなければなりません。

4. JVM サーバー用の JVMSERVER リソースを作成します。

   a. 作成した JVM プロファイルの名前を指定します。

b. JVM サーバーのスレッド限度を指定します。 必要なスレッド数は、JVM サーバーで実行したいワークロードによって異なります。始めにデフォルト値を受け入れてから、環境を調整できます。1 つの JVM サーバーで最大 256 個のスレッドを設定できます。

c. オプション: DFHAXRO とは異なるエンクレーブ用の Language Environment オプションを使用する場合、それらを提供するプログラムを指定します。 CICS は、DFHAXRO プログラムで既にコンパイルされている一連のデフォルト値を提供します。必要に応じて独自のオプションを提供して、エンクレーブを調整することができます。詳しくは、 201 ページの『DFHAXRO を使用した JVM サーバーのエンクレーブの変更』を参照してください。

タスクの結果

JVMSERVER リソースを使用可能にすると、CICS は Language Environment エンクレーブを作成し、JVM プロファイルから JVM サーバーにオプションを渡します。プロファイル内のオプションに応じて、JVM サーバーは OSGi フレームワークまたは Axis2 を実行するように構成されます。

- JVM サーバーが OSGi をサポートする場合、JVM が始動し、OSGi フレームワークはすべての OSGi ミドルウェア・バンドルを解決します。
- JVM サーバーが Axis2 をサポートする場合、JVM が始動し、Axis2 JAR ファイルをロードします。

JVM サーバーが始動を正常に完了すると、JVMSERVER リソースが ENABLED 状態でインストールされます。

エラーが発生する場合、例えば、CICS が JVM プロファイルの検出も読み取りもできない場合、JVM サーバーは初期化できません。 JVMSERVER リソースは DISABLED 状態でインストールされ、CICS はシステム・ログにエラー・メッセージを発行します。

次のタスク

OSGi をサポートするように構成されている JVM サーバーの場合、 97 ページの『JVM サーバーへの OSGi バンドルのインストール』で説明されているとおりに、フレームワークに OSGi バンドルをインストールできます。Axis2 をサポートするように構成されている JVM サーバーの場合、「*CICS Web サービス・ガイド*」で説明されているとおりに、JVM サーバーで Web サービス要求を実行するように CICS を構成できます。

DB2 用の JVM サーバーのセットアップ

JVM サーバー内で実行されている Java アプリケーションから DB2 にアクセスする場合は、JVM プロファイルにオプションを追加する必要があります。

始める前に

DB2 と組み合わせて JVM サーバーを使用するには、IBM Data Server Driver for JDBC and SQLJ の最新バージョンが必要です。必要な APAR については、http://www-01.ibm.com/support/docview.wss?uid=swg27020857 に記載されているシステム要件を参照してください。

このタスクについて

CICS は、DB2 ドライバーを操作するためのミドルウェア OSGi バンドル com.ibm.cics.db2.jcc.jar を提供しています。アプリケーションが DB2 にアクセスできるように、OSGi フレームワークにこのミドルウェア・バンドルをインストールする必要があります。

手順

1. 該当する JVM サーバーの JVM サーバー・プロファイルを開きます。 CICS Explorer 内で z/OS パースペクティブを使用して、JVM プロファイルを開いて編集し、保管できます。
2. LIBPATH_SUFFIX オプションに、該当する DB2 ドライバーの lib ディレクトリーの場所を追加します。
3. プロファイルに JVM システム・プロパティー **-Dcom.ibm.cics.db2.jcc.jdbc.home** を追加し、DB2 ドライバーの場所を指定します。
4. OSGI_BUNDLES オプションに com.ibm.cics.db2.jcc.jar ミドルウェア・バンドルを追加します。
5. 変更内容を保管します。
6. 既存の JVM サーバーを更新している場合は、JVMSERVER リソースを使用不可にしてから使用可能にします。そうでなければ、JVMSERVER リソースを作成します。 CICS は OSGi フレームワークを始動し、ミドルウェア・バンドルをインストールします。

例

次に、必要なオプションを指定した JVM プロファイルの例を抜粋して示します。

```
#*********************************************************************
#
#                          Required parameters
#                          -------------------
#
# When using a JVM server, the set of CICS options that are supported
JAVA_HOME=/usr/lpp/java/J6.0.1_64
WORK_DIR=.
LIBPATH_SUFFIX=/usr/lpp/db2910/lib
...
#*********************************************************************
#
#                     JVM server specific parameters
#                     ------------------------------
#
OSGI_BUNDLES=/usr/lpp/cicsts42/lib/com.ibm.cics.db2.jcc.jar
#
OSGI_FRAMEWORK_TIMEOUT=60
#
#*********************************************************************
#
#                               JVM options
#                               -----------
# The following option sets the Garbage collection Policy.
#
-Xgcpolicy:gencon
#
#*********************************************************************
#
```

```
#                 Setting user JVM system properties
#                 ----------------------------------
#
# -Dcom.ibm.cics.some.property=some_value
#Djdbc.drivers=com.ibm.db2.jcc.DB2Driver
-Dcom.ibm.cics.db2.jcc.jdbc.home=/usr/lpp/db2910
#
#*********************************************************************
#
#              Unix System Services Environment Variables
#              ------------------------------------------
#
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,SYSDUMP),ONINTERRUPT(NONE)"
#
#
```

JVM サーバーへの OSGi バンドルのインストール

JVM サーバーに Java アプリケーションをデプロイするには、ターゲット JVM サーバーの OSGi フレームワークにそのアプリケーションの OSGi バンドルをインストールする必要があります。

始める前に

アプリケーションの OSGi バンドルが入っている CICS バンドルは、zFS にデプロイされなければなりません。ターゲット JVM サーバーは、CICS 領域で稼働していなければなりません。

このタスクについて

CICS バンドルには、1 つ以上の OSGi バンドルおよびサービスを含むことができます。CICS バンドルはデプロイメントの単位であるため、すべての OSGi バンドルとサービスは、BUNDLE リソースの一部として一緒に管理されます。また、OSGi フレームワークは、依存関係とバージョン管理方式の管理を含めて、OSGi バンドルおよびサービスのライフサイクルを管理します。

ベスト・プラクティスとしては、Java アプリケーションを構成するすべての OSGi バンドルが、同じ CICS バンドルにデプロイされていることを確認してください。この方式を使用すると、BUNDLE リソースを使用することによってアプリケーションを単一エンティティーとして管理できます。OSGi バンドル相互間に依存関係がある場合は、それらを同じ CICS バンドルにデプロイしてください。CICS BUNDLE リソースをインストールすると、CICS によって OSGi バンドル相互間の依存関係がすべて解決されます。

共通コードのライブラリーを含む OSGi に依存する場合、ベスト・プラクティスは、そのライブラリー用に別個の CICS バンドルを作成することです。この場合、そのライブラリーを含む CICS BUNDLE リソースを最初にインストールすることが重要です。Java アプリケーションをインストールしてから、その Java アプリケーションが依存する CICS バンドルをインストールすると、OSGi フレームワークは、その Java アプリケーションの依存関係を解決できません。

手順

1. CICS バンドル内のターゲット JVM サーバーを調べて、その名前を持つ JVM サーバーが CICS 領域に存在することを確認します。

2. zFS 内のバンドルのディレクトリーを指定する BUNDLE リソースを作成します。
   a. CICS Explorer メニュー・バーの「**Definitions**」 > 「**Bundle Definitions**」をクリックして、「Bundles Definitions」ビューを開きます。
   b. そのビュー内の任意の場所を右クリックし、「**New**」をクリックして「New Bundle Definition」ウィザードを開きます。 そのウィザードのフィールドに、BUNDLE リソースの詳細を入力してください。
   c. BUNDLE リソースをインストールします。 リソースを Enabled 状態または Disabled 状態のどちらかでインストールできます。
      - DISABLED 状態でリソースをインストールすると、CICS は OSGi バンドルをフレームワークにインストールし、依存関係を解決しますが、バンドルを開始しようとしません。
      - ENABLED 状態でリソースをインストールすると、CICS は OSGi バンドルをインストールし、依存関係を解決し、OSGi バンドルを開始します。OSGi バンドルに遅延バンドルのアクティベーターが含まれている場合、OSGi フレームワークは、別の OSGi バンドルによって最初に呼び出されるまでそのバンドルを開始しようとしません。
3. オプション: BUNDLE リソースがまだ ENABLED 状態でない場合、そのリソースを使用可能にして、フレームワークで OSGi バンドルを開始します。
4. CICS Explorer メニュー・バーの「**Operations**」 > 「**Bundles**」をクリックして、「Bundles」ビューを開きます。BUNDLE リソースの状態を確認します。
   - BUNDLE リソースが ENABLED 状態である場合、CICS はバンドル内のすべてのリソースを正常にインストールできました。
   - BUNDLE リソースが DISABLED 状態である場合、CICS はバンドル内の 1 つ以上のリソースをインストールできませんでした。

   BUNDLE リソースが ENABLED 状態でインストールできなかった場合、BUNDLE リソースのバンドル・パーツを確認してください。いずれかのバンドル・パーツが UNUSABLE 状態である場合、CICS は OSGi バンドルを作成できませんでした。通常、この状態は、zFS で CICS バンドルに問題があることを示します。その BUNDLE リソースを破棄し、問題を修正してから、BUNDLE リソースを再度インストールする必要があります。
5. CICS Explorer メニュー・バーの「**Operations**」 > 「**Java**」 > 「**OSGi Bundles**」をクリックして、「OSGI Bundles」ビューを開きます。OSGi フレームワークにインストールされた OSGi バンドルおよびサービスの状態を確認します。 次の表では、状態をまとめています。

BUNDLE	BUNDLEPART	OSGIBUNDLE	OSGISERVICE
ENABLED	ENABLING	INSTALLED	該当せず
	ENABLED	STARTING	該当せず
		ACTIVE	ACTIVE
			INACTIVE
DISABLED	DISABLING	STOPPING	該当せず
	DISABLED	RESOLVED	該当せず
	UNUSABLE	該当せず	該当せず

- OSGi バンドルが STARTING 状態である場合、バンドル・アクティベーターが呼び出されましたが、まだ戻っていません。OSGi バンドルに遅延活動化ポリシーがある場合、OSGi フレームワークで呼び出されるまで、そのバンドルはこの状態のままです。
- OSGi バンドルと OSGi サービスがアクティブである場合、Java アプリケーションは作動可能です。
- OSGi サービスが非アクティブである場合、CICS は、その名前を持つ OSGi サービスが OSGi フレームワークに既に存在することを検出しました。

タスクの結果

BUNDLE が使用可能になり、OSGi バンドルが OSGi フレームワークに正常にインストールされ、すべての OSGi サービスがアクティブです。OSGi バンドルおよびサービスは、フレームワーク内の他のバンドルから使用可能です。

次のタスク

OSGi フレームワークの外部にある他の CICS アプリケーションから Java アプリケーションを使用可能にすることができます。

JVM サーバー内の Java アプリケーションの呼び出し

JVM サーバーで実行している Java アプリケーションを別の CICS アプリケーションから呼び出すには、OSGi フレームワークでアクティブな OSGi サービスを使用する必要があります。

このタスクについて

OSGi サービスは明確に定義されたインターフェースであり、OSGi バンドル用に OSGi フレームワークに登録されています。他の OSGi バンドルやリモート・アプリケーションは、OSGi サービスを使用して、OSGi バンドルにパッケージされているアプリケーション・コードを呼び出します。OSGi バンドルには、複数の OSGi サービスがある場合があります。

OSGi フレームワークは、同じフレームワークにインストールされている OSGi バンドル用のサービスの呼び出しを管理します。OSGi フレームワークの外部にある CICS アプリケーションから、Java アプリケーションを呼び出すには、OSGi バンドル用の適切な OSGi サービスを使用してください。

手順

1. OSGi フレームワークで使用したい、アクティブな OSGi サービスのシンボル名を判別します。 CICS Explorer で「**Operations**」 > 「**Java**」 > 「**OSGi Services**」をクリックして、アクティブな OSGi サービスをリストします。
2. 他の CICS アプリケーションに対して OSGi サービスを表す PROGRAM リソースを作成します。
   a. JVM 属性で、YES を指定して、プログラムが Java プログラムであることを示します。
   b. JVMCLASS 属性で、OSGi サービスのシンボル名を指定します。 この値は大/小文字の区別があります。

c. JVMSERVER 属性で、OSGi サービスが実行される JVMSERVER リソースの名前を指定します。

3. Java アプリケーションをさまざまな方法で呼び出すことができます。
   - トランザクション ID を指定する 3270 または **EXEC CICS START** 要求を使用します。OSGi サービスの PROGRAM リソースを定義する TRANSACTION リソースを作成します。
   - **EXEC CICS LINK** 要求、ECI 呼び出し、または EXCI 呼び出しを使用します。要求をコーディングする際に、OSGi サービスの PROGRAM リソースの名前を指定します。
   - プログラム・リスト・テーブル (PLT) のエントリーを使用します。OSGi サービスの PROGRAM リソースの名前を指定します。

タスクの結果

他の CICS アプリケーションから OSGi バンドルを使用できるようにする PROGRAM リソースを作成しました。OSGi サービスが呼び出されると、CICS は、ターゲット JVM サーバーで要求を実行します。OSGi サービスがアクティブとして登録されている場合、Java プログラムは正常に実行されます。OSGi サービスが登録されていないか、非アクティブである場合、呼び出し側プログラムにエラーが戻されます。

Java セキュリティー・マネージャーの有効化

デフォルトで、Java アプリケーションでは、Java API に要求されるアクティビティーにセキュリティーの制限がありません。Java セキュリティーを使用して、安全でない可能性があるアクションを Java アプリケーションが実行しないようにするために、そのアプリケーションが実行される JVM に対してセキュリティー・マネージャーを有効にすることができます。

このタスクについて

セキュリティー・マネージャーは、コード・ソースに割り当てられる 1 組の権限 (システム・アクセス権) であるセキュリティー・ポリシーを実施します。 Java プラットフォームにはデフォルトのポリシー・ファイルが用意されています。ただし、Java セキュリティーがアクティブであるときに、Java アプリケーションを CICS で正常に実行できるようにするには、アプリケーションの実行に必要な権限を CICS に付与する追加のポリシー・ファイルを指定する必要があります。

この追加のポリシー・ファイルは、セキュリティー・マネージャーが有効になっている JVM の種類ごとに指定する必要があります。CICS 提供のいくつかの例を使用して、ユーザー独自のポリシーを作成できます。

手順

- JVM サーバーの OSGi フレームワークで実行されるアプリケーションの場合は、次の手順で行います。
  1. CICS Explorer SDK でプラグイン・プロジェクトを作成し、提供されている OSGi セキュリティー・エージェントの例を選択します。 この例により、セキュリティー・プロファイルを含むプロジェクト内に、`com.ibm.cics.server.examples.security` という名前の OSGi ミドルウェ

ア・バンドルが作成されます。このプロファイルは、インストール先のフレームワーク内にあるすべての OSGi バンドルに適用されます。

2. プロジェクト内で、セキュリティー・ポリシーの権限を編集するために、example.permissions ファイルを選択します。 このファイルには、アプリケーションが System.exit() メソッドを使用していないことを確認するための検査など、JVM サーバー内でのアプリケーションの実行に固有の権限が含まれています。
3. zFS 内の適切なディレクトリーに OSGi バンドルをデプロイします。 CICS には、このディレクトリーへの読み取りおよび実行アクセス権限が必要です。
4. Java ランチャーにすべての権限を与えるポリシー・ファイルを作成します。プラグイン・プロジェクトには、ポリシーの例 all.policy が提供されています。この例はミドルウェア・バンドルには含まれていませんが、zFS 内の適切なディレクトリーにコピーできます。 このポリシー・ファイルには、次の権限が含まれています。

   ```
   grant {
   permission java.security.AllPermission;
   };
   ```

5. JVM サーバーの JVM プロファイルを編集して、他のすべてのバンドルより前になるように、OSGI_BUNDLES オプションに OSGi バンドルを追加します。

   ```
   OSGI_BUNDLES=/u/bundles/com.ibm.cics.server.examples.security_1.0.0.jar,
   /usr/lpp/cicsts42/lib/com.ibm.cics.db2.jcc.jar
   ```

6. JVM プロファイルに次の Java 環境変数を追加して、OSGi フレームワーク内でセキュリティーを有効にします。

   ```
   org.osgi.framework.security=osgi
   ```

7. セキュリティー・ポリシーを指定するために、JVM プロファイルに次の Java セキュリティー・システム・プロパティーを追加します。

   ```
   -Djava.security.policy=/u/policies/all.policy
   ```

8. 変更内容を保管し、JVMSERVER リソースを使用可能に設定して、JVM サーバーにミドルウェア・バンドルをインストールします。

- プールされた JVM 内で実行されるアプリケーションの場合は、dfhjejbpl.policy ファイルを使用してセキュリティー・ポリシーを実装します。

1. /usr/lpp/java/J6.0.1_64/lib/security/ でアプリケーション用のポリシー・ファイルを作成します。ここで、java/J6.0.1_64 は IBM 64-bit SDK for z/OS, Java テクノロジー・エディション の場所です。

   セキュリティー・マネージャーは、このディレクトリーで提供されるデフォルトのポリシー・ファイル java.policy を常に使用します。アプリケーションが JDBC または SQLJ を使用する必要がある場合は、JDBC ドライバーに権限を付与するポリシー・ファイルを作成します。Java セキュリティーでは JDBC 2.0 ドライバーを使用する必要があります。
2. **-Djava.security.manager** システム・プロパティーを JVM プロファイルに追加して、セキュリティー・マネージャーを有効にします。 次のいずれかの形式を使用します。
   - -Djava.security.manager=default
   - -Djava.security.manager=""
   - -Djava.security.manager=

3. **-Djava.security.policy** システム・プロパティーを JVM プロファイルに追加して、ポリシー・ファイルを指定します。 セキュリティー・マネージャーは、デフォルトのセキュリティー・ポリシーに加えて、このプロパティーで設定されるすべてのポリシーを使用します。

タスクの結果

Java アプリケーションが呼び出されると、JVM は、クラスのコード・ソースを判別し、セキュリティー・ポリシーを調べてから、適切な権限をクラスに付与します。

プールされた JVM のセットアップ

プールされた JVM 環境は、Enterprise Java Beans、CORBA、およびスレッド・セーフでない Java アプリケーションの実行に必要です。提供された JVM プロファイルおよび CICS リソースをセットアップする必要があります。オプションとして、Hello World サンプルを実行して、ご使用の環境が正しくセットアップされていることを確認することができます。

このタスクについて

手順

1. 提供されたサンプル DFHJVMPR および DFHJVMCD を、それらのインストール場所から、**JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーにコピーします。 提供されたプロファイルのコピーを使用すると、保守の適用時にプロファイルが更新されても、変更内容が失われません。
   - DFHJVMPR は、プールされた JVM 用に提供されているプロファイルです。
   - DFHJVMCD は、システム・プログラムと共用クラス・キャッシュ用に提供されているプロファイルです。
2. JVM プロファイルをカスタマイズして、JVM の開始時に JVM を構成するオプションを編集します。 例えば、使用可能なストレージの量を変更し、セキュリティー設定を適用します。これらのオプションについては、111 ページの『JVM プロファイル: オプションおよびサンプル』を参照してください。
3. オプション: JCICS HelloWorld サンプルを使用して、プールされた JVM 環境のセットアップを確認します。
4. CICS リソースおよび JVM プロファイルを作成して、アプリケーション、Enterprise Java Bean、または CORBA が、プールされた JVM を使用できるようにします。 カスタマイズされたデフォルト・プロファイル DFHJVMPR を使用するか、独自のプロファイルを作成できます。

タスクの結果

環境が構成され、プールされた JVM で Java アプリケーションを実行するための CICS リソースが作成されました。

DFHJVMCD のカスタマイズ

JVM プロファイル DFHJVMCD は、CICS 提供のシステム・プログラム、特に CICS 提供の CIRP 要求プロセッサー・トランザクションで使用されるデフォルト

の要求プロセッサー・プログラム DFJIIRP で使用するために予約されています。また、CICS は、DFHJVMCD を使用して、プールされた JVM の共用クラス・キャッシュの初期化と終了も行います。

始める前に

DFHJVMCD は、CICS 領域に対して正しくセットアップされなければなりませんが、カスタマイズするのは必要な場合のみです。DFHJVMCD には、JVM プロパティー・ファイルを関連付けることができますが、これはオプションです。

インストール場所のオリジナル・ファイルではなく、**JVMPROFILEDIR** システム初期設定パラメーターで指定した z/OS UNIX ディレクトリーで DFHJVMCD のコピーを使用していることを確認します。

このタスクについて

変更できるオプションは、DFHJVMCD のテキストで示されます。ファイルにこれ以外の変更を加えないでください。

DFHJVMCD で変更可能なオプション、およびそれらのオプションを変更する目的について詳しくは、117 ページの『CICS 環境における JVM のオプション』および 127 ページの『JVM システム・プロパティー』を参照してください。

手順

1. 標準テキスト・エディターで DFHJVMCD を開きます。
2. CICS 領域に共用クラス・キャッシュがある場合は、DFHJVMCD プロファイルを使用して作成されたすべてのプールされた JVM が、その共用クラス・キャッシュを使用することを指定できます。CLASSCACHE オプションを CLASSCACHE=YES に変更します。 デフォルトの CLASSCACHE=NO は、スタンドアロン JVM であることを意味します。
3. JAVA_HOME オプションの値が、z/OS UNIX 上の IBM 64-bit SDK for z/OS, Java テクノロジー・エディション のインストール・ディレクトリーと一致しない場合は、この値を変更します。
4. DFHJVMCD プロファイルで JVM によって使用される z/OS UNIX 上の作業ディレクトリーを変更するには、WORK_DIR オプションを変更して適切なディレクトリーを指定します。
5. `stderr`、`stdin`、および `stdout` に使用される z/OS UNIX ファイルの名前を変更するには、STDERR、STDIN、および STDOUT オプションを変更します。
6. 出力リダイレクト・クラスを使用して、JVM からの出力およびメッセージを代行受信し、リダイレクトするには、USEROUTPUTCLASS オプションを使用してそのクラスの名前を指定します。 実稼働環境ではこのオプションを使用しないでください。
7. アプリケーションのニーズに対応するように、DFHJVMCD プロファイルで JVM のヒープ・サイズを調整するには、**-Xms** または **-Xmx** オプションを変更します。

8. JDBC を使用するアプリケーションがある場合は、サンプル・プロファイル DFHJVMPR で指定される適切な DB2 ライブラリーおよびファイルを、DFHJVMCD の LIBPATH_SUFFIX および CLASSPATH_SUFFIX オプションに追加します。
9. ご使用のシステムで必要な場合、Java セキュリティー・ポリシー・メカニズム (**-Djava.security.policy** システム・プロパティー) を有効にします。
10. 変更内容をプロファイルに保管します。 カスタマイズされた DFHJVMCD のコピーが、**JVMPROFILEDIR** システム初期設定パラメーターで指定した z/OS UNIX ディレクトリーにあることを確認します。

次のタスク

独自のアプリケーション用にセットアップした PROGRAM リソースで DFHJVMCD を指定しないでください。アプリケーションで使用するために、他の CICS 提供のサンプル JVM プロファイル DFHJVMPR のコピーに、同じようなカスタマイズの変更を加えることができます。

DFHJVMPR のカスタマイズ

DFHJVMPR は、プールされた JVM 用のデフォルトの JVM プロファイルです。このプロファイルに加える変更はすべて、PROGRAM リソースが別の JVM プロファイルを指定しないすべてのプールされた JVM に適用されます。

始める前に

始める前に、インストール場所にあるオリジナルのファイルではなく、提供されたサンプル JVM プロファイルのコピーを使用していることを確認してください。

このタスクについて

DFHJVMPR プロファイルをカスタマイズする場合は、変更内容が、そのプロファイルを使用するすべての Java アプリケーションに適切であることを確認してください。追加したい Java アプリケーションのオプションが、他のアプリケーションに適用されない場合は、別の名前を使用して、DFHJVMPR に基づく JVM プロファイルを作成してください。

手順

1. 標準テキスト・エディターで JVM プロファイルを開き、オプションを編集します。 111 ページの『JVM プロファイル: オプションおよびサンプル』にあるオプションのリストを使用してください。各パラメーターまたはプロパティーは別々の行で指定され、パラメーターまたはプロパティーの値は、行の終わりで区切られます。 114 ページの『JVM プロファイルのコーディング規則』のコーディング規則に従ってください。

   プールされた JVM の以下の主なオプションを変更することができます。

   - セキュリティー・マネージャーとポリシーをプロファイルに追加して、Java セキュリティーを有効にします。Java セキュリティー・ポリシー・メカニズムは、Java アプリケーションが安全でないアクションを実行しないように保護します。**-Djava.security.manager** および **-Djava.security.policy** システ

ム・プロパティーを使用して、プロファイルにセキュリティーを追加できます。詳しくは、 100 ページの『Java セキュリティー・マネージャーの有効化』を参照してください。

- アプリケーションに使用可能なストレージの量を調整するには、JVM プロファイルの **-Xmx** オプションを変更します。このオプションは、JVM 内のヒープのサイズを変更します。デフォルト値は 16 MB ですが、大きい Java アプリケーションがある場合は、この値を増やすことができます。
- JVM プロファイルの IDLE_TIMEOUT オプションを使用して、JVM のタイムアウトしきい値を変更します。デフォルトは、非アクティブな JVM が 30 分後に CICS による自動終了の対象になることです。未使用の JVM をより長い期間使用可能にしておきたい場合は、タイムアウトしきい値を最大 7 日まで指定するか、JVM がタイムアウトにならないように設定することができます。
- JVM からのメッセージおよび出力の宛先を変更します。 stdin、stdout、stderr ファイルおよび Java ダンプの名前と場所を変更し、シンボルを使用してこれらのファイルを各 JVM に固有にすることができます。アプリケーション開発時に、JVM プロファイルの USEROUTPUTCLASS オプションを使用して、JVM 内部からのメッセージおよび Java アプリケーションからの出力をリダイレクトすることができます。 213 ページの『JVM stdout、stderr およびダンプ出力の場所の制御』では、可能な変更内容について詳しく説明しています。

2. オプション: Java アプリケーションに、JDBC を使用した DB2 データへのアクセスが必要な場合、**-Djdbc.drivers** システム・プロパティーを使用します。 詳しくは、「DB2 Guide」の『Using JDBC and SQLJ to access DB2 data from Java programs』を参照してください。
3. CICS 領域に **JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーに、カスタマイズされた JVM プロファイルを保管します。 CICS は、このディレクトリーから JVM プロファイルをロードします。
4. z/OS UNIX で JVM プロファイルおよびそれが入っているディレクトリーに対する読み取りおよび書き込みアクセス権限が CICS にあることを確認します。
5. DFHJVMPR プロファイルを使用する JVM が実行中である場合、そのプロファイルの JVM プールを段階的に停止します。 DFHJVMPR を使用している既存の JVM はすべて、いったん停止されてから、開始されます。新しい JVM は、最新バージョンの JVM プロファイルを使用します。

タスクの結果

プールされた JVM のサンプル JVM プロファイルをカスタマイズし、z/OS UNIX ディレクトリー内のプロファイルに対する適切なアクセス権限が CICS にあることが確実になりました。

次のタスク

カスタマイズされた JVM プロファイルを使用し、アプリケーションのクラスをクラスパスに追加するように、アプリケーションをセットアップできます。 109 ページの『プールされた JVM を使用するアプリケーションの有効化』を参照してください。

独自の JVM プロファイルの作成

カスタマイズされたサンプル・プロファイルを更新しないようにしたい場合、または特定のアプリケーションに対して、別のファイル名を持つ JVM プロファイルを作成することができます。

始める前に

CICS 領域に **JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーに、提供されたサンプル・プロファイルのコピーが必要です。CICS には、このディレクトリーへの読み取りおよび実行アクセス権限が必要です。

このタスクについて

管理を最小限に抑えるために、必ず、該当するサンプル・プロファイルをコピーし、名前を変更してください。Java アプリケーションが JVM サーバーで実行される場合、変更のベースとして DFHOSGI プロファイルを使用してください。 Java アプリケーションがプールされた JVM で実行される場合、変更のベースとして DFHJVMPR プロファイルを使用してください。

手順

1. 該当するサンプル JVM プロファイルをコピーし、名前を変更します。 DFH で始まる名前を JVM プロファイルに付けないでください。これらの文字は CICS が使用するために予約されているからです。 JVM プロファイルの名前には、大/小文字の区別があります。プロファイルの命名について詳しくは、 8 ページの『JVM プロファイル』を参照してください。
2. 111 ページの『JVM プロファイル: オプションおよびサンプル』のオプションのリストを使用して、標準テキスト・エディターで JVM プロファイルを編集します。 JVM サーバーに固有のオプションもあれば、プールされた JVM に固有のオプションもあります。

   各パラメーターまたはプロパティーは別々の行で指定され、パラメーターまたはプロパティーの値は、行の終わりで区切られます。 114 ページの『JVM プロファイルのコーディング規則』のコーディング規則に従ってください。
3. Java セキュリティーを有効にしたい場合は、セキュリティー・オプションを指定し、1 つ以上のセキュリティー・ポリシー・ファイルをセットアップして、JVM のセキュリティー・プロパティーを定義します。 詳しくは、 100 ページの『Java セキュリティー・マネージャーの有効化』を参照してください。
4. CICS 領域に **JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーに、JVM プロファイルを保管します。 CICS は、このディレクトリーから JVM プロファイルをロードします。
5. z/OS UNIX で JVM プロファイルに対する読み取りアクセス権限が CICS にあることを確認します。 89 ページの『z/OS UNIX ディレクトリーおよびファイルに対するアクセス権の CICS 領域への付与』を参照してください。

タスクの結果

作成した JVM プロファイルは、命名規則に準拠し、ご使用の Java アプリケーションと目的のランタイム環境に適切なオプションを含んでいます。

次のタスク

CICS リソースの作成を含めて、JVM プロファイルを使用するようにアプリケーションをセットアップします。 93 ページの『JVM サーバーのセットアップ』で説明されているとおりに JVM サーバーをセットアップするか、 109 ページの『プールされた JVM を使用するアプリケーションの有効化』で説明されているとおりにプールされた JVM をセットアップすることができます。

プログラム例によるプールされた JVM のセットアップの確認

「Hello World」および「Hello CICS World」プログラム例をセットアップし、実行して、プールされた JVM 環境が CICS 領域で正しくセットアップされていることを確認します。

始める前に

プログラム例を実行する前に、 87 ページの『第 4 章 Java サポートのセットアップ』で説明されている他のセットアップ・タスクが完了していることを確認します。

このタスクについて

アプリケーション開発者が CICS における Java アプリケーションの開発に取り掛かるのに役立つように、Java の例が CICS Explorer SDK に用意されています。また、Java ソース・ファイルおよびビルド・ファイルも CICS インストール時に z/OS UNIX で提供されており、プールされた JVM 環境が正しくセットアップされていることを確認するためにプログラム例を実行する場合に使用できます。

提供されたプログラムをセットアップし、実行するには、z/OS UNIX で環境変数を定義する必要があります。**export** コマンドを使用して、z/OS UNIX のプロファイルで変数を定義できます。または、z/OS UNIX にログインするときに手動で **export** コマンドを入力することもできます。

手順

1. PATH は、z/OS UNIX システム・サービスの検索パスです。PATH 環境変数を次のように定義します。

   ```
   /usr/lpp/java/J6.0.1_64/bin
   ```

   パスは、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション が z/OS UNIX 上にインストールされている場所を指定します。次のように、**export** コマンドを使用してパスを追加することができます。

   ```
   export PATH=/usr/lpp/java/J6.0.1_64/bin:$PATH
   ```

2. CICS_HOME は、z/OS UNIX システム・サービスで CICS Transaction Server for z/OS ファイルのインストール・ディレクトリーです。次のように、CICS_HOME 環境変数を定義します。

   ```
   /usr/lpp/cicsts/cicspath
   ```

   cicspath の値は、CICS TS をインストールしたときに **USSDIR** インストール・パラメーターによって定義されます。cicsts42 がデフォルトです。次のように、export コマンドを使用してディレクトリー接頭部を設定できます。

```
export CICS_HOME=/usr/lpp/cicsts/cicsts42
```

$CICS_HOME/samples/dfjcics ディレクトリーには、Make ファイルが入っています。

$CICS_HOME/samples/dfjcics/examples ディレクトリーには Java ソースが入っています。

3. JAVA_HOME は、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション サブディレクトリーへのパスを指定します。次のように、JAVA_HOME 環境変数を定義します。

```
/usr/lpp/java/java_location/
```

java_location は、z/OS UNIX 上で IBM 64-bit SDK for z/OS, Java テクノロジー・エディション がインストールされている場所です。デフォルト値は java/J6.0.1_64/ です。

4. Java の例をビルドします。
   a. samples/dfjcics ディレクトリー内で、make jvm と入力してすべての例をビルドします。 Make ファイルは javac を呼び出し、出力ファイルを *$CICS_HOME*/samples/dfjcics/examples/*sample_name* z/OS UNIX ディレクトリーに保管します。ここで、sample_name はプログラム例の名前です。
   b. SDFHSAMP 内にある提供された C プログラムをコンパイルし、変換します。
      - DFH$LCCA
      - DFH$JSAM
      - DFH$LCCC

      これらのプログラムは、プログラム制御のプログラム例と、「Hello World」Java プログラム例の 1 つによってリンクされます。DFH$LCCA および DFH$JSAM は標準の CICS プログラムであり、CICS によってサポートされている任意の言語で作成できます。 C コンパイラーがない場合、提供されたプログラムの COBOL バージョンを作成し、提供された C バージョンの代わりに使用することができます。
   c. プログラムを DFHRPL またはダイナミック・ライブラリー連結にリンクします。
5. デフォルト JVM プロファイル DFHJVMPR 内の CLASSPATH_SUFFIX オプションに、ストリング /usr/lpp/cicsts/cicsts42/samples/dfjcics を追加します。/usr/lpp/cicsts/cicsts42 は、**USSHOME** システム初期設定パラメーターの値です。
6. 43 ページの『Hello World サンプルの実行』で説明されている手順のとおりに行って、Hello World プログラム例を実行します。

タスクの結果

Hello World プログラム例が正常に実行されました。

次のタスク

プールされた JVM を使用した Java アプリケーションの実行を可能にすることができます。

プールされた JVM を使用するアプリケーションの有効化

Java アプリケーションがプールされた JVM を使用できるようにするには、JVM で Java プログラムを実行するための CICS リソースをセットアップする必要があります。また、アプリケーションのクラスを検出する場所も定義する必要があります。

このタスクについて

単一 JVM でスレッド・セーフでないアプリケーションを実行する場合は、プールされた JVM を使用します。

手順

1. Java アプリケーションに該当する JVM プロファイルを選択または作成します。提供されたサンプル DFHJVMPR をコピーすることができます。 JVM プロファイルはすべて、**JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーに置かれます。
2. JVM プロファイルを編集して、アプリケーションで必要なクラスとライブラリーを追加します。 任意の標準テキスト・エディターを使用できます。パス間の分離文字としてコロンを使用してください。改行を組み込むには、円記号とブランク (\ ) を使用します。
   a. 標準クラスパスにアプリケーション・クラスを置きます。標準クラスパスは **CLASSPATH_SUFFIX** オプションによって定義されます。 クラス自体の名前や、JVM プロファイル内のパッケージの名前を指定しないでください。JVM プロファイル内のオプションが、クラスへのパスを指定します。
   b. クラスがパッケージ内にない場合、すべてのサブディレクトリーをクラスパスに組み込みます。
   c. クラスまたはパッケージが JAR ファイル内にある場合、JAR ファイルの名前をクラスパスに組み込みます。

   JVM プロファイル内のクラスパスやその他の項目をコーディングする規則について詳しくは、114 ページの『JVM プロファイルのコーディング規則』を参照してください。
3. アプリケーションで必要なすべてのネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイルを、JVM プロファイル内の **LIBPATH_SUFFIX** オプションに追加します。

   IBM またはベンダーによって提供されるミドルウェアおよびツールでは、DLL ファイルをライブラリー・パスに追加する必要がある場合があります。例えば、DB2 JDBC ドライバーを使用するには、DLL ファイルが必要です。また、作成したクラスにネイティブ・コードが関連付けられている場合もあります。
4. **JVMPROFILEDIR** システム初期設定パラメーターによって指定されたディレクトリーに、JVM プロファイルを保管します。
5. PROGRAM リソースを作成し、該当する Java 属性を設定します。 属性に値を入力する際には、JVM クラスおよび JVM プロファイルに正しい大/小文字を使用するようにしてください。
   a. EXECKEY 属性で、Java プログラムの実行キーを指定します。 この属性のデフォルト値は USER であり、ストレージ保護を改善するので大部分の Java

プログラムに適しています。ただし、プログラムが、TASKDATAKEY(CICS) を指定するトランザクションの一部である場合、プログラムは CICS キーで実行されなければなりません。

b. JVM 属性で、YES を指定して、プログラムが Java プログラムであることを示します。

c. JVMCLASS 属性で、Java プログラム内のメイン・クラスの名前を指定します。 プログラムがパッケージとして構築された場合、完全修飾名を指定してください。これは、パッケージ名で修飾された Java クラス名であり、ピリオド (.) を分離文字として使用します。

例えば、example.HelloWorld パッケージには、HelloCICSWorld クラスが含まれます。この場合、完全修飾クラス名は example.HelloWorld.HelloCICSWorld です。プログラムがパッケージとして構築されていない場合、修飾子なしでクラス名を指定してください。

d. JVMPROFILE 属性で、アプリケーション・クラスを組み込むように編集したプロファイルの名前を指定します。

タスクの結果

Java アプリケーションの実行をサポートするために、JVM プロファイルと PROGRAM リソースが CICS 領域で使用可能になります。アプリケーションが Java プログラムの実行を要求する場合、さまざまな方法で要求を行うことがあります。

- トランザクション ID を指定する 3270 または **EXEC CICS START** 要求。
- **EXEC CICS LINK** 要求、または Java プログラムの名前を直接指定する ECI または EXCI 呼び出し。
- プログラム・リスト・テーブル (PLT) のエントリー。

EXEC CICS LINK 要求または ECI もしくは EXCI 呼び出しの場合、およびプログラム・リスト・テーブルのエントリーの場合、CICS には PROGRAM リソースの名前が直接提供されます。しかし、3270 または START 要求の場合、CICS が、トランザクション ID を使用して PROGRAM リソースを判別します。

次のタスク

JVM プロファイルで、JVM が共用クラス・キャッシュを使用することが指定される場合、Java アプリケーションが実行されるには、クラス・キャッシュが開始されるか、自動開始が使用可能であることを確認する必要があります。 163 ページの『共用クラス・キャッシュの開始』では、共用クラス・キャッシュを開始するか、自動開始を使用可能にする方法を説明しています。

JVM を使用する CORBA またはエンタープライズ Bean アプリケーションの有効化

CORBA アプリケーションまたはエンタープライズ Bean がプールされた JVM を使用できるようにするには、そのアプリケーションの JVM プロファイルとクラスパスを更新する必要があります。

このタスクについて

CORBA ステートレス・オブジェクトおよびエンタープライズ Bean には独自の PROGRAM リソースはありません。エンタープライズ Bean または CORBA ステートレス・オブジェクトに対するメソッド要求には JVM が関係します。これは、処理を行う要求プロセッサーが JVM 内で実行されるためです。要求プロセッサーとは、メソッドを処理するコンテナーの呼び出しを含めて、IIOP 要求の実行を管理するプログラムです。 CICS はメソッド要求を受け取ると、REQUESTMODEL リソースと比較し、要求に最も適合するものを検出し、この要求モデルのトランザクション ID を使用して PROGRAM リソース定義を判別します。

場合によっては、既に JVM が割り当てられている既存の要求プロセッサー・トランザクションを使用して IIOP 要求が処理されることもあります。新規の要求プロセッサー・トランザクションが必要になった場合、 CICS は一致する要求モデルのトランザクション ID のみを確認します。

手順

1. CORBA ステートレス・オブジェクトまたはエンタープライズ Bean を処理する要求プロセッサー・プログラムの JVM プロファイルを識別します。 JVM プロファイルは、要求プロセッサー・プログラムの PROGRAM リソースで指定されます。デフォルトの要求プロセッサー・プログラムは DFHJIIRP であり、このプログラムのデフォルト JVM プロファイルは DFHJVMCD です。
2. CORBA ステートレス・オブジェクトの場合のみ、要求プロセッサー・プログラムの JVM プロファイルで、CLASSPATH_SUFFIX オプションにアプリケーションの JAR ファイルを追加します。 クラスパスで指定するパス間の分離文字としてコロンを使用してください。改行を組み込むには、円記号とブランク (\ ) を使用します。

   エンタープライズ Bean のデプロイ済み JAR (DJAR) ファイルをクラスパスに追加する必要はありません。
3. JAR ファイルに含まれていないクラス (ユーティリティー用のクラスなど) がエンタープライズ Bean または CORBA アプリケーションで使用されている場合は、要求プロセッサー・プログラムの JVM で使用されるクラスパスにこれらのクラスを含めます。

タスクの結果

Java アプリケーションの実行をサポートするために、JVM プロファイルとクラスパスが CICS 領域で使用可能になります。

JVM プロファイル: オプションおよびサンプル

CICS には、CICS 環境で使用される IBM JVM 用の一連のオプションが入っているサンプルの JVM プロファイルが備えられています。こうしたオプションの中には CICS 環境固有のものもあり、他の環境の JVM には使用できません。その他のオプションは、すべての環境の IBM JVM で使用可能な標準または非標準の Java オプションです。

JVM プロファイル内に任意の JVM オプションまたはシステム・プロパティーを指定して、それらを JVM に渡すことができます。JVM サーバーとプールされた JVM の JVM プロファイルは異なるので、1 つのタイプの Java 環境のみのオプションを指定できます。

JVM プロパティー・ファイルでシステム・プロパティーを設定できます。ただし、JVM プロパティー・ファイルは、プールされた JVM に対してのみサポートされ、サンプル・プロパティー・ファイルは CICS に提供されません。

JVM のすべてのオプションおよびシステム・プロパティーの中心となるリポジトリーはありません。使用できる情報源は次のとおりです。

- IBM 64-bit SDK for z/OS, Java テクノロジー・エディション, バージョン 6 の資料。
- http://www-03.ibm.com/systems/z/os/zos/tools/java/products/sdk601_64.html で入手可能な「*IBM SDK Java Technology Edition Version 6 Supplement*」。この資料には、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション、バージョン 6.0.1 に固有の情報が記載されています。
- Java Diagnostics Guide。このガイドには、JVM トレースおよび問題判別に使用するシステム・プロパティーが記載されています。

Java クラス・ライブラリーには他のシステム・プロパティーも含まれており、アプリケーションに独自のシステム・プロパティーがある場合もあります。 IBM Java 資料が 1 次的な情報源であり、CICS 資料は 2 次的な情報源です。

要約表である表 9 は、サンプル JVM プロファイルで使用されるオプション、および JVM サーバーとプールされた JVM に適用されるオプションをリストしています。この表には、CICS 資料で記述されているタスクの実行に使用できる、いくつかの追加オプションも含まれています。この表は、サンプル JVM プロファイルでデフォルトが指定されていない場合の、オプションごとのデフォルトを示しています。

表 9. CICS 環境における JVM 用の JVM オプション参照表

オプション	デフォルト	JVM サーバー	プールされた JVM	コメント
JVM タイプ				
CLASSCACHE	NO	Not supported	Supported	YES では、JVM が共用クラス・キャッシュを使用し、NO では、使用しません。
REUSE	YES	Not supported	Supported	YES は継続 JVM になり、NO は単一使用 JVM になります
ディレクトリー				
JAVA_HOME	None	Supported	Supported	必須。サンプル・プロファイルにこのディレクトリーが含まれています。
WORK_DIR	/tmp	Supported	Supported	
パス				

表 9. CICS 環境における JVM 用の JVM オプション参照表 (続き)

オプション	デフォルト	JVM サーバー	プールされた JVM	コメント
CLASSPATH_PREFIX	None	Not supported	Supported	
CLASSPATH_SUFFIX	None	Not supported	Supported	
LIBPATH_PREFIX	None	Supported	Supported	
LIBPATH_SUFFIX	None	Supported	Supported	
OSGI_BUNDLES	None	Supported	Not supported	JVM サーバーでミドルウェア・バンドルを使用したい場合に設定されます
タイムアウトしきい値				
IDLE_TIMEOUT	30 分	Not supported	Supported	継続のプールされた JVM のみに適用されます。
OSGI_FRAMEWORK_TIMEOUT	60 秒	Supported	Not supported	
JVM 用の追加設定と機能				
JVMPROPS	None	Not supported	Supported	JVM プロパティー・ファイルを使用する場合のみ設定されます。
INVOKE_DFHJVMAT	NO	Not supported	Supported	単一使用のプールされた JVM のみに適用されます。
ストレージ・ヒープ・サイズ				
-Xms		Supported	Supported	**-Xms** のデフォルト値については、Default settings for the JVM の参照情報を参照してください。
-Xmx		Supported	Supported	**-Xmx** のデフォルト値については、Default settings for the JVM の参照情報を参照してください。
ガーベッジ・コレクションしきい値				
GC_HEAP_THRESHOLD	85%	Not supported	Supported	継続のプールされた JVM のみに適用されます。
JVM からの出力				
JVMTRACE	`dfhjvmtrc`	Supported	Not supported	
LEHEAPSTATS	NO	Not supported	Supported	
STDERR	`dfhjvmerr`	Supported	Supported	
STDIN	`dfhjvmin`	Supported	Supported	
STDOUT	`dfhjvmout`	Supported	Supported	
USEROUTPUTCLASS	None	Supported	Supported	開発環境のみで設定されます。

表 9. CICS 環境における JVM 用の JVM オプション参照表 (続き)

オプション	デフォルト	JVM サーバー	プールされた JVM	コメント
問題判別およびアプリケーション・デバッグ				
JAVA_DUMP_OPTS	YES	Supported	Supported	
-Xdebug	NO	Supported	Supported	
PRINT_JVM_OPTIONS	NO	Supported	Supported	YES に設定するのは一時的のみ

z/OS UNIX システム・サービス環境変数

JVM の構成に使用される JVM オプションとシステム・プロパティーに加えて、z/OS UNIX システム・サービス環境変数も JVM プロファイルで指定できます。JVM オプションまたはシステム・プロパティーとして認識されない、JVM プロファイル内の名前と値のペアはすべて、z/OS UNIX システム・サービス環境変数として扱われ、エクスポートされます。 JVM プロファイルで指定される z/OS UNIX システム・サービス環境変数は、そのプロファイルで作成される JVM にのみ適用されます。

サンプル JVM プロファイルの `JAVA_DUMP_OPTS` オプションと `JAVA_DUMP_TDUMP_PATTERN` オプションは、z/OS UNIX システム・サービス環境変数です。もう 1 つの例は `TZ` 環境変数で、JVM 用の時間帯を変更する際に指定できます。

z/OS UNIX システム・サービス環境変数は、JVM プロファイルのみで指定できます。

JVM プロファイルのコーディング規則

任意の標準テキスト・エディターを使用して JVM プロファイルを編集することができます。JVM プロファイルをコーディングする際には、以下の規則に従ってください。

- JVM プロファイルの名前の長さは最大 8 文字にすることができます。JVM プロパティー・ファイルの名前は任意の長さにすることができますが、使いやすいように、一般に、そのプロパティー・ファイルを参照する JVM プロファイルの名前に似た短い名前です。
- JVM プロファイルまたは JVM プロパティー・ファイルの名前は、z/OS UNIX システム・サービス内のファイルに有効な任意の名前にすることができます。DFH で始まる名前を使用しないでください。これらの文字は CICS が使用するために予約されているからです。
- JVM プロファイルと JVM プロパティー・ファイルは UNIX ファイルであるため、大文字小文字が重要です。 CICS で名前を指定する場合は、z/OS UNIX ファイル名にあるのと同じ大文字と小文字の組み合わせを使用して名前を入力する必要があります。
- JVM プロファイルでディレクトリーの値を指定する場合は、引用符を使用しないでください。
- CEDA パネルは、端末の UCTRAN 設定にかかわらず、JVMPROFILE フィールドでの大文字小文字混合入力を受け入れます。ただし、コマンド行から CEDA

を使用する場合、または別の CICS トランザクションを使用する場合は、大文字小文字混合で JVM プロファイルの名前を入力する必要があります。使用する端末が、大文字変換が抑止された状態で正しく構成されていることを確実にしてください。提供される CEOT トランザクションを使用して、現行セッションに対してのみ、独自の端末の英大文字変換状況 (UCTRAN) を変更することができます。

JVM オプションまたはシステム・プロパティーをコーディングする際には、以下の規則に従ってください。

大/小文字の区別

パラメーター・キーワードおよびオペランドはすべて、大/小文字の区別があり、 117 ページの『CICS 環境における JVM のオプション』および 127 ページの『JVM システム・プロパティー』に示されているとおり正確に指定する必要があります。

クラスパス分離文字

CLASSPATH_SUFFIX などのクラスパス・オプションで指定するディレクトリー・パスを分離するには、: (コロン) 文字を使用してください。

継続

JVM オプションまたはシステム・プロパティーの場合、値は、テキスト・ファイル内の行の終わりで区切られます。入力または編集しようとする値が、エディターのウィンドウには長すぎる場合は、スクロールしないですむように改行することができます。次の行に継続するには、次の例のように、現在行の終わりに円記号 (\) 文字とブランクの継続文字を付けます。

```
CLASSPATH_SUFFIX=/u/example/pathToJarOrZipFile/jarfile.jar:\
/u/example/pathToRootDirectoryForClasses
```

コメント

コメントを追加するか、オプションを削除する代わりにコメント化するには、コメントの各行の先頭に # 記号を付けます。ファイルが JVM ランチャーによって読み取られるときに、コメント行は無視されます。

ブランク行も無視されます。オプション間、またはオプションのグループ間の分離文字としてブランク行を使用できます。

文字のエスケープ・シーケンス

プロパティー・エレメント・ストリングでは、表 10 に示されているエスケープ・シーケンスをコーディングできます。

表 10. エスケープ・シーケンス

エスケープ・シーケンス	文字の値
\b	バックスペース
\t	水平タブ
\n	改行
\r	復帰
\"	二重引用符
\'	単一引用符
\\	円記号 (\)

表 10. エスケープ・シーケンス (続き)

エスケープ・シーケンス	文字の値
\xxx	8 進値 xxx に相当する文字。ここで、xxx は 000 から 377 の値です。
\uxxxx	xxxx をエンコードする Unicode 文字。ここで、xxxx は 1 桁から 4 桁の 16 進数字です (詳しくは、注を参照してください)。

注: Unicode \u エスケープは、他のエスケープ・タイプとは異なります。Unicode エスケープ・シーケンスは、 115 ページの表 10 で記述されている他のエスケープ・シーケンスより前に処理されます。Unicode エスケープは、非 Unicode システムで表示できない文字を表す代替方法です。ただし、文字エスケープは特殊文字を表すことができますが、それらの文字は通常どおりに解釈されません。

オプションの複数インスタンス

JVM プロファイルでは各オプションを一回しか使用できません。同じオプションの複数のインスタンスが JVM プロファイルに含まれている場合、最後に検出されるオプションの値が使用され、それより前の値は無視されます。

ストレージ・サイズ

JVM プロファイルでストレージ関連のオプションを指定する場合、ストレージ・サイズを 1024 バイトの倍数で指定してください。文字 K は KB を、文字 M は MB を、および文字 G は GB をそれぞれ表します。例えば、ヒープの初期サイズとして 6 291 456 バイトを指定するには、以下のいずれかの方法で **-Xms** をコーディングします。

```
-Xms6144K
-Xms6M
```

JVM プロファイル・オプションの検証

CICS は、JVM プロファイルで指定されたかぎとなるオプションに関して JVM の開始時に必ず幾つかの検査を実行します。これらの検査は、JVM セットアップにおける問題の早期検出を可能にします。

CICS は、以下の JVM プロファイル・オプションに関して検査を行います。

CLASSPATH_PREFIX、CLASSPATH_SUFFIX

JVM サーバー・プロファイルの場合、CICS は、これらのオプションがプロファイルに存在しないことを検査します。プロファイルでどちらかのオプションが指定される場合、JVM サーバーの OSGi フレームワークは開始できません。JVMSERVER リソースを使用可能にすることはできず、CICS は DFHSJ0210 エラー・メッセージを発行します。

JAVA_HOME

CICS は、このディレクトリーの以下の点を検査します。

- ディレクトリーが z/OS UNIX に存在すること。
- CICS が、ディレクトリーにアクセスするために少なくとも読み取り権限を持っていること。

- JDK_INSTALL_OK ファイルがディレクトリー内にあること。これは、この場所でIBM 64-bit SDK for z/OS, Java テクノロジー・エディション 6.0.1 ファイルのインストールが完了したことを示します。
- JDK_INSTALL_OK ファイルの Java リリース番号が、CICS によってサポートされているバージョンであること。

何らかの問題が見つかると、CICS はエラー・メッセージを発行して JVM は開始されません。

LIBPATH、CICS_HOME、CLASSPATH、TMPREFIX、および TMSUFFIX は、使用すべきでないクラスパス・オプションです。
プールされた JVM プロファイルにこれらのオプションが 1 つ以上ある場合、JVM の開始時に警告メッセージが出されます。JVM プロファイルでこれらのオプションを使用しないでください。メッセージには、代わりに使用する正しいオプションが表示されます。

OSGI_BUNDLES
JVM サーバー・プロファイルの場合、CICS は、指定された JAR ファイルが OSGi バンドルであることを検査します。また、CICS は、ミドルウェア・バンドルが正しく区切られ、正しい分離文字があることも検査します。

CICS 環境における JVM のオプション

JVM プロファイル内のオプションは JVM を始動するために、CICS、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション、または z/OS UNIX システム・サービスによって使用されます。

オプションを指定する場合、 114 ページの『JVM プロファイルのコーディング規則』で説明されているコーディング規則に従っていることを確認してください。オプションの形式は、変わる場合があります。

- JVM プロファイル内の一部のオプションは、キーワードと値が = 記号によって区切られる形式 (例: `LEHEAPSTATS=NO`) を取ります。
- 一部のオプションでは、= 記号がなく、オプションの直後に値が続きます (例: `-Xms16M`)。
- ハイフン (-) で始まるオプションはすべて、Java 標準オプションまたは Java 非標準オプションのどちらかであり、CICS によって構文解析されることなく JVM に渡されます。
- JVM プロファイルで任意の z/OS UNIX システム・サービス環境変数を指定することができます。 JVM オプションまたはシステム・プロパティーとして認識されない、JVM プロファイル内の名前と値のペアはすべて、z/OS UNIX システム・サービス環境変数として扱われ、エクスポートされます。
- -D で始まる JVM システム・プロパティーを、JVM プロファイルで指定できます。 127 ページの『JVM システム・プロパティー』には、そうしたプロパティーだけが取り上げられています。

-Xmso、**-Xiss**、および **-Xss** JVM オプションおよびすべてのデフォルト値については、Default settings for the JVM の参照情報を参照してください。

プロファイルのシンボル

以下のシンボルは、サンプル JVM プロファイルで例示されているように、JVM プロファイルのオプションの値で使用できます。

&APPLID;
このシンボルを使用すると、実行時に CICS 領域のアプリケーション ID に置換されます。この方法ですべての領域に同じプロファイルまたはプロパティー・ファイルを使用でき、同時に領域固有の作業ディレクトリーまたは出力宛先を用いることができます。APPLID は常に大文字です。WORK_DIR、STDOUT、STDERR、および JAVA_DUMP_TDUMP_PATTERN オプションでこのシンボルを使用できます。

&DATE;
このシンボルを使用すると、シンボルは実行時に *Dyymmdd* 形式の現在日付で置き換えられます。WORK_DIR、STDOUT、STDERR、および JAVA_DUMP_TDUMP_PATTERN オプションを含めて、JVM からの任意のタイプの出力に &DATE; シンボルを指定できます。

&JVM_NUM;
このシンボルを使用すると、プールされた JVM の固有の番号に実行時に置換されます。 JVM ごとに固有の出力またはダンプ・ファイルを作成する場合に、このシンボルを使用してください。WORK_DIR、STDOUT、STDERR、および JAVA_DUMP_TDUMP_PATTERN オプションでこのシンボルを使用できます。TDUMP の MVS データ・セット命名標準に準拠するために、CICS が JVM 番号を変更する場合があります。

このシンボルは JVM サーバーには適用されません。

&JVMSERVER;
このシンボルを使用すると、実行時に JVMSERVER リソースの名前に置換されます。JVM サーバーごとに固有の出力またはダンプ・ファイルを作成する場合に、このシンボルを使用してください。

このシンボルは、プールされた JVM には適用されません。

&TIME;
このシンボルを使用すると、シンボルは実行時に *Thhmmss* 形式の現在時刻で置き換えられます。WORK_DIR、STDOUT、STDERR、および JAVA_DUMP_TDUMP_PATTERN オプションを含めて、JVM からの任意のタイプの出力に &TIME; シンボルを指定できます。

JVM オプションのリスト

以下のオプションのリストでは、明示的に指定される場合を除いて、すべてのオプションが JVM サーバーとプールされた JVM の両方に適用されます。オプションに指定されるデフォルト値は、そのオプションが JVM プロファイルで指定されない場合に CICS が使用する値です。一部またはすべてのサンプル JVM プロファイルで、デフォルト値とは異なる値が指定される場合があります。

CLASSCACHE={YES,NO}
この JVM で共用クラス・キャッシュを使用するかどうかを指定します。デフォルト値は NO です。

このオプションは JVM サーバーには適用されません。

CLASSPATH_PREFIX, CLASSPATH_SUFFIX=*class_pathnames*

標準クラスパスは、プールされた JVM がアプリケーション・クラスとリソースを検索するディレクトリー・パス、JAR ファイル、および圧縮ファイルを指定します。各行の最後で \ (バックスラッシュ) を使用すると行が継続されるので、行を分けて複数のエントリーを指定することができます。

CLASSPATH_PREFIX は、クラスパス・エントリーを標準クラスパスの先頭に追加し、CLASSPATH_SUFFIX は標準クラスパスの末尾に追加します。

SDK のバージョン 6.0.1 では、すべてのアプリケーション・クラスが標準クラスパスに置かれます。プールされた JVM の場合、すべてのアプリケーション・クラスが共用クラス・キャッシュへのロード対象です。

CLASSPATH_PREFIX オプションは注意して使用してください。CLASSPATH_PREFIX のクラスは、CICS と Java ランタイムによって提供される、同じ名前のクラスより優先されるため、誤ったクラスがロードされる可能性があります。

CICS は、**USSHOME** システム初期設定パラメーターと JVM プロファイルの JAVA_HOME オプションで指定されたディレクトリーの /lib サブディレクトリーを使用して、JVM の基本クラスパスを作成します。この基本クラスパスには、CICS と JVM によって用意されている JAR ファイルが入ります。それは JVM プロファイルでは見られません。 JVM プロファイルのクラスパスでこれらのファイルを再度指定することはありません。

JVM サーバーのプロファイルでいずれかのオプションを設定すると、OSGi フレームワークが開始されません。

DISPLAY_JAVA_VERSION=

このオプションを YES に設定すると、アプリケーションによって JVM が始動される際、CICS はメッセージ DFHSJ0901 を MSGUSER ログに必ず書き込みます。このメッセージには、使用している IBM Software Developer Kit for z/OS、Java Technology Edition のバージョンとビルドが表示されます。

GC_HEAP_THRESHOLD=

JVM のヒープのヒープ使用率限界を指定します。 ヒープのアクティブな部分のストレージでこのパーセンテージが使用されると、CICS はガーベッジ・コレクションをスケジュールします。 CICS は Java プログラムが実行されるたびにヒープ使用率をチェックします。使用率の限界に達すると、現在使用中の JVM が終了した直後にガーベッジ・コレクション・トランザクション CJGC を JVM で実行するようスケジュールされます。

デフォルトのヒープ使用率限界は 85 (85%) です。最小は 50 です。 CICS にガーベッジ・コレクションをスケジュールさせる場合、99 が最大です。ヒープ使用率限界を 100 に指定すると、CICS はガーベッジ・コレクションをスケジュールせず、アプリケーション実行時の割り振り失敗の結果としてガーベッジ・コレクションが生じます。

このオプションは、JVM サーバーにも、単独使用のプールされた JVM にも適用されません。

IDLE_TIMEOUT={30|*number***}**

この JVM プロファイルで使用するプールされた JVM 用のタイムアウトしきい値 (分単位) を指定します。プールされた JVM が指定の時間、非アクティブである場合、自動終了の対象になります。 CICS が次回アイドル JVM を検査す

る際、JVM が依然として非アクティブな場合には、JVM とその J8 または J9 TCB は破棄される可能性があります。 CICS は、タイムアウトになったすべての JVM を直ちに停止するわけではありません。時間をかけて徐々に終了します。

デフォルトのタイムアウトしきい値は 30 分で、最大は 10,080 分 (7 日) です。また、タイムアウトしきい値はゼロに指定することもできます。その結果、そのプロファイルを使用する JVM は非アクティブになっても自動停止されません。タイムアウトしきい値がゼロの JVM が停止する可能性があるのは、スチールまたはミスマッチで選択される場合、または MVS ストレージが制限状態になる場合です。受け入れられない値を指定すると、CICS は代わりにデフォルトを使用します。

このオプションは、JVM サーバーにも、単独使用のプールされた JVM にも適用されません。

INVOKE_DFHJVMAT={NO|YES}
JVM を作成する前に、ユーザーが置き換え可能なモジュール DFHJVMAT が呼び出されるかどうかを指定します。 DFHJVMAT は単独使用のプールされた JVM にのみ、つまりオプション REUSE=NO が JVM プロファイルで指定されている場合にのみ使用できます。

このオプションは、JVM サーバーにも、継続のプールされた JVM にも適用されません。

JAVA_DUMP_OPTS=
z/OS UNIX システム・サービス環境変数。 JVM での異常終了の診断情報を取得するために使用される 1 組の Java ダンプ・オプションを指定します。 Dump agent environment variables で Java ダンプ・オプションに関する情報を参照してください。

JAVA_DUMP_TDUMP_PATTERN=
JVM からのトランザクション・ダンプ (TDUMP) に使用されるファイル名を指定する z/OS UNIX システム・サービス環境変数。 Java TDUMP は、JVM 異常終了の場合にデータ・セット宛先に書き込まれます。 JVM ごとに固有のダンプ・ファイル名を作成するには、提供されたサンプル JVM プロファイルに示されているように、この値で &APPLID; (CICS 領域アプリケーション ID) シンボルと &JVM_NUM; (固有の JVM 番号) シンボルを使用できます。

&JVM_NUM シンボルをここで使用する場合、MVS データ・セット命名標準に準拠するために、CICS が JVM 番号を変更する可能性があります。この番号は 8 桁の 16 進値としてフォーマットされます。先頭文字が数字である場合、変更する必要があります。0 は G に変更され、1 は H に変更され、以下同様に続き、最後に 9 が P に変更されます。

JAVA_HOME=/usr/lpp/java/J6.0.1_64/
z/OS UNIX で IBM 64-bit SDK for z/OS, Java テクノロジー・エディション のインストール場所を指定します。この場所には、Java サポートに必要なサブディレクトリーと JAR ファイルが入っています。

提供されたサンプル JVM プロファイルには、DFHISTAR CICS インストール・ジョブで **JAVADIR** パラメーターによって生成されたパスが含まれています。**JAVADIR** パラメーターのデフォルトは、`java/J6.0.1_64/` であり、これは、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション のデフォルトのイ

ンストール場所です。この値では、JVM プロファイルの JAVA_HOME 設定値は /usr/lpp/java/J6.0.1_64/ になります。

JAVA_PIPELINE={YES,NO}
必要な JAR ファイルをクラスパスに追加して、JVM サーバーが、Java ベースの SOAP パイプラインで処理する Web サービスをサポートできるようにします。デフォルト値は NO です。この値をYES に設定すると、JVM サーバーは、OSGi ではなく、Axis2 をサポートするように構成されます。

このオプションは、プールされた JVM には適用されません。

JVMPROPS=*path/file_name*
オプションの JVM プロパティー・ファイルのパスと名前を指定します。このファイルは、この JVM のシステム・プロパティーを入れるのに使用できる z/OS UNIX ファイルです。JVM プロパティー・ファイルで指定可能な内容について詳しくは、127 ページの『JVM システム・プロパティー』を参照してください。

このオプションは JVM サーバーには適用されません。

JVMTRACE={*applid.jvmserver.*dfhjvmtrc|*file_name*}
JVM サーバーの開始と終了時に Java トレースが書き込まれる先の z/OS UNIX ファイルの名前を指定します。値が指定されない場合、トレースは *applid.jvmserver.*dfhjvmtrc ファイルに書き込まれます。 CICS は、&APPLID; および &JVMSERVER; シンボルを使用して、JVM サーバーごとに固有の出力ファイルを自動的に作成します。このファイルは、WORK_DIR オプションで指定されるディレクトリーに作成されます。

このオプションは、プールされた JVM には適用されません。

LEHEAPSTATS={YES|NO}
JVM が使用する Language Environment ヒープ・ストレージ量に関する統計を収集するかどうかを指定します。デフォルト値は NO です。統計は、JVM プロファイル統計の「Peak Language Environment heap storage used」フィールドとして報告されます。こうした統計を収集すると JVM のパフォーマンスに影響を与えるので、LEHEAPSTATS=YES を指定するのは、JVM のヒープ・サイズを調整しているときだけにしてください。詳しくは、を参照してください。実稼働環境では、LEHEAPSTATS=NO を指定してください。

このオプションは JVM サーバーには適用されません。

LIBPATH_PREFIX, LIBPATH_SUFFIX=*pathnames*
JVM が使用するネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイルで、z/OS UNIX で拡張子 .so を持つものを検索するためのディレクトリー・パスを指定します。これには、アプリケーション・コードまたはサービスによってロードされる JVM および追加のネイティブ・ライブラリーを実行するのに必要なファイルが含まれます。

JVM の基本ライブラリー・パスは、**USSHOME** システム初期設定パラメーターと JVM プロファイルの JAVA_HOME オプションで指定されたディレクトリーを使用して自動的に作成されます。この基本ライブラリー・パスは、JVM プロファイルでは表示されません。このライブラリー・パスには、CICS が使用する JVM とネイティブ・ライブラリーを実行するのに必要なすべての DLL ファイルが含まれています。

LIBPATH_SUFFIX オプションを使用すると、このライブラリー・パスを拡張できます。このオプションはディレクトリーを、ライブラリー・パスの最後、基本ライブラリー・パスの後に追加します。このオプションは、アプリケーションで、または CICS の標準 JVM セットアップに含まれていないサービスで使用する追加のネイティブ・ライブラリーが含まれるディレクトリーを指定するのに使用します。例えば、追加のネイティブ・ライブラリーには、DB2 JDBC ドライバーを使用するのに必要な DLL ファイルを含めることができます。

LIBPATH_PREFIX オプションは、ライブラリー・パスの先頭で、基本ライブラリー・パスの前にディレクトリーを追加します。このオプションは注意して使用してください。指定されたディレクトリー内の DLL ファイルが、基本ライブラリー・パス上の DLL ファイルと同じ名前である場合は、提供されたファイルの代わりにロードされます。

最適なパフォーマンスを得るには、アプリケーションで使用するためにライブラリー・パス上に含めるすべての DLL ファイルをコンパイルし、XPLink オプションを指定してリンクしてください。基本ライブラリー・パスで提供される DLL ファイルおよび DB2 JDBC ドライバーなどのサービスで使用される DLL ファイルは、XPLink オプションを指定して作成されます。

OSGI_BUNDLES=*pathnames*
JVM サーバーの OSGi フレームワークで有効なミドルウェア・バンドルのディレクトリー・パスを指定します。これらの OSGi バンドルには、WebSphere MQ への接続などのシステム機能をフレームワークに実装するためのクラスが含まれています。複数の OSGi バンドルを指定する場合は、コンマを使用してそれらのバンドルを分離してください。

このオプションは、プールされた JVM には適用されません。

OSGI_FRAMEWORK_TIMEOUT=60|*number*
OSGi フレームワークの初期設定またはシャットダウンがタイムアウトになるまでに CICS が待機する秒数を指定します。1 秒から 60000 秒までの値を設定できます。デフォルト値は 60 秒です。OSGi フレームワークが開始するのに、指定された秒数よりも長くかかる場合、JVM サーバーは初期設定できず、DFHSJ0215 メッセージが CICS によって発行されます。 zFS の JVM サーバー・ログ・ファイルにもエラー・メッセージが書き込まれます。OSGi フレームワークがシャットダウンするのに、指定された秒数よりも長くかかる場合、JVM サーバーは正常にシャットダウンできません。

このオプションは、プールされた JVM には適用されません。

PRINT_JVM_OPTIONS={YES|NO}
このオプションを YES に設定すると、JVM が始動される際、その時に JVM に渡されるすべてのオプションは必ず SYSPRINT に印刷されます。JVM がプロファイル内でこのオプションを指定して開始されるたびに、出力が生成されます。このオプションを使用すると、JVM プロファイルでは見えない、特定の JVM プロファイルのクラスパス (CICS によって作成される基本ライブラリー・パスと基本クラスパスを含む) の内容を確認できます。

REUSE={YES|NO}
プールされた JVM が再使用可能かどうかを指定します。

- REUSE=YES (デフォルト) は、Java アプリケーションが何度も再利用できる JVM が作成されます。このタイプのプールされた JVM は、継続 JVM と呼ばれます。
- REUSE=NO は再利用されない JVM を作成します。その場合、単一の Java プログラムによって実行された後に破棄されます。このタイプのプールされた JVM は、単独使用 JVM と呼ばれます。

このオプションは JVM サーバーには適用されません。

STDERR={dfhjvmerr|*file_name*} [-generate]
stderr に使用される z/OS UNIX ファイルの名前を指定します。このファイルが存在しない場合、WORK_DIR オプションで指定されたディレクトリーに作成されます。ファイルが存在する場合には、そのファイルの末尾に出力が追加されます。 JVM が停止するときに、stderr ファイルが空で、そのファイルが特定の JVM 用に作成されたものであれば、そのファイルは削除されます。それ以外の場合には、ファイルは保持されます。

- プールされた JVM の場合、デフォルト名は dfhjvmerr です。固定のファイル名の場合、複数の JVM からの出力がその指定されたファイルに追加され、出力はインターリーブされます。 JVM ごとに固有の出力ファイルを作成するには、サンプル JVM プロファイルで示されているように &JVM_NUM; シンボルと &APPLID; シンボルをファイル名で使用するか、**-generate** オプションを指定してください。**-generate** オプションは、固有の JVM 番号、CICS 領域のアプリケーション ID、および追加の識別情報をファイル名に追加します。 **-generate** の前にはブランクを 1 つ付ける必要があります。
- JVM サーバーの場合、ファイル名は *applid.jvmserver.*dfhjvmerr です。CICS は、&APPLID; および &JVMSERVER; シンボルを使用して、JVM サーバーごとに固有の出力ファイルを自動的に作成します。

JVM プロファイルで USEROUTPUTCLASS オプションを指定すると、このオプションで指定された Java クラスが、代わりに System.err 要求を処理します。提供されたサンプル・クラス com.ibm.cics.samples.SJMergedStream を使用する場合と同様に、USEROUTPUTCLASS オプションで指定されたクラスが目的の宛先にデータを書き込めない場合、STDERR オプションで指定された z/OS UNIX ファイルが引き続き使用される可能性があります。またこのファイルは、USEROUTPUTCLASS オプションで指定されたクラスによって、他の何らかの理由のために出力がそのファイルに送信される場合にも使用できます。

STDIN={dfhjvmin|*file_name*}
stdin に使用される z/OS UNIX ファイルの名前を指定します。このファイルが存在しない場合、WORK_DIR オプションで指定されたディレクトリーに作成されます。

STDOUT={dfhjvmout|*file_name*} [-generate]
stdout ファイルへの出力に使用される z/OS UNIX ファイルの名前を指定します。このファイルが存在しない場合、WORK_DIR オプションで指定されたディレクトリーに作成されます。ファイルが存在する場合には、そのファイルの末尾に出力が追加されます。 JVM が停止するときに、stdout ファイルが空で、そのファイルが特定の JVM 用に生成されたものであれば、そのファイルは削除されます。それ以外の場合には、ファイルは保持されます。

- プールされた JVM の場合、デフォルト名は dfhjvmout です。固定のファイル名の場合、複数の JVM からの出力がその指定されたファイルに追加され、出力はインターリーブされます。 JVM ごとに固有の出力ファイルを作成するには、サンプル JVM プロファイルで示されているように &JVM_NUM; シンボルと &APPLID; シンボルをファイル名で使用するか、**-generate** オプションを指定してください。
- JVM サーバーの場合、ファイル名は *applid.jvmserver.*dfhjvmout です。CICS は、&APPLID; および &JVMSERVER; シンボルを使用して、JVM サーバーごとに固有の出力ファイルを自動的に作成します。

JVM プロファイルで USEROUTPUTCLASS オプションを指定すると、このオプションで指定された Java クラスが、代わりに System.out 要求を処理します。USEROUTPUTCLASS オプションで指定されたクラスが目的の宛先にデータを書き込めない場合、STDOUT オプションで指定された z/OS UNIX ファイルが引き続き使用される可能性があります。例えば、サンプル・クラス com.ibm.cics.samples.SJMergedStream を使用する場合です。またこのファイルは、USEROUTPUTCLASS オプションで指定されたクラスによって、他の何らかの理由のために出力がそのファイルに送信される場合にも使用できます。

USEROUTPUTCLASS={*classname***}**

JVM からの出力および JVM 内部からのメッセージを代行受信する Java クラスの完全修飾名を指定します。この Java クラスを使用して、ご使用の JVM からの出力およびメッセージをリダイレクトし、出力レコードにタイム・スタンプとヘッダーを追加できます。 Java クラスがデータを目的の宛先に書き込めない場合、STDOUT および STDERR オプションで指定されたファイルが引き続き使用される場合があります。

USEROUTPUTCLASS オプションを指定すると、JVM のパフォーマンスに悪影響が出ます。実稼働環境で最高のパフォーマンスを発揮するには、このオプションは使用しないでください。ただし、このオプションは、JVM 出力を識別可能な宛先に送信できるので、同じ CICS 領域を使用するアプリケーション開発者に便利な場合があります。

このクラスおよび提供されたサンプルについて詳しくは、 213 ページの『JVM stdout、stderr およびダンプ出力の場所の制御』を参照してください。

WORK_DIR={.|*directory_name***}**

CICS 領域が Java 関連の活動に使用する、z/OS UNIX 上にある作業ディレクトリーを指定します。 CICS JVM インターフェースは、stdin、stdout、および stderr ファイルの作成時にこのディレクトリーを使用します。提供された JVM プロファイルではピリオド (.) が定義されます。これは、CICS 領域ユーザー ID のホーム・ディレクトリー (つまり、z/OS UNIX ディレクトリー /u/*CICS region userid*) が、作業ディレクトリーとして使用されることを示します。このディレクトリーは、CICS インストール時に作成されます。 CICS 領域ユーザー ID にこのホーム・ディレクトリーがない場合、または WORK_DIR が省略されると、/tmp が z/OS UNIX ディレクトリー名として使用されます。

相対作業サブディレクトリー

プールされた JVM だけの場合、ピリオドの後にサブディレクトリー名を指定すると、この z/OS UNIX ディレクトリーに相対サブディレクトリーが作成されるので、そこで出力ファイルを保持できます。例えば、次のように指定するとします。

```
WORK_DIR=./javaoutput
```

その CICS 領域内のすべての JVM からの出力ファイルは、CICS 領域ユーザー ID のホーム・ディレクトリー内にあるサブディレクトリー javaoutput に作成されます。

絶対作業ディレクトリー

プールされた JVM と JVM サーバーの場合、作業ディレクトリーへの絶対パスを指定できます。ホーム・ディレクトリーを Java 関連の活動の作業ディレクトリーとして使用したくない場合や、CICS 領域が同じ z/OS ユーザー ID (UID) を共用しているため同一のホーム・ディレクトリーを持っている場合には、CICS 領域ごとに異なる作業ディレクトリーを作成できます。このために、&APPLID; シンボルを使用するディレクトリー名を指定します。CICS がこのシンボルを実際の CICS 領域アプリケーション ID に置換します。したがって、すべての CICS 領域で同じ JVM プロファイルのセットを共用する場合でも、領域ごとに固有の作業ディレクトリーを持つことができます。例えば、次のように指定するとします。

```
WORK_DIR=/u/&APPLID;/javaoutput
```

その JVM プロファイルを使用する各 CICS 領域には、独自の作業ディレクトリーがあります。z/OS UNIX 上で適切なすべてのディレクトリーを作成し、それらのディレクトリーへの読み取り、書き込み、および実行アクセス権限を CICS 領域に付与したことを確認してください。

作業ディレクトリーに固定名を指定することも可能ですが、その場合もやはり z/OS UNIX 上で適切なディレクトリーを作成し、CICS 領域に適切なアクセス権限を付与したことを確認します。作業ディレクトリーに固定名を使用する場合は、JVM プロファイルを共用する CICS 領域内のすべての JVM からの出力ファイルが、そのディレクトリーに作成されます。ご使用の出力ファイルにも固定名を使用した場合には、こうした CICS 領域内のすべての JVM からの出力は同じ z/OS UNIX ファイルに追加されてしまいます。同じファイルに追加されないようにするには、適切な JVM プロファイル・オプションを指定した &JVM_NUM; シンボルと &APPLID; シンボルを使用して、各 CICS 領域内の JVM ごとに固有な出力およびダンプ・ファイルを生成します。

USSHOME システム初期設定パラメーターで定義された CICS ファイルのホーム・ディレクトリーである、z/OS UNIX 上の CICS ディレクトリーで、作業ディレクトリーを定義しないでください。

また USEROUTPUTCLASS オプションを使用すると、JVM からの stderr および stdout 出力を代行受信、リダイレクト、およびフォーマット設定する Java クラスを指定できます。場合によっては、出力リダイレクト用に提供されたサンプル・クラスでは、WORK_DIR によって指定されたディレクトリーを使用します。

-generate

z/OS UNIX 上の stdout (JVM 出力) ファイルと stderr (JVM エラー・メッセージ) ファイルを固有に識別するために、このオプションを指定します。このオ

プションは、STDOUT オプションと STDERR オプションでファイル名の後に指定されなければなりません。例えば、STDOUT=dfhjvmout -generate を指定できます。

-generate オプションは、STDOUT オプションまたは STDERR オプションに指定したファイル名に、固有の JVM 番号 (&JVM_NUM; シンボルと同様)、CICS 領域アプリケーション ID (&APPLID; シンボルと同様)、および追加の修飾子を付加します。

例えば、**-generate** オプションを使用して作成される標準的な stdout ファイルには、次の名前があります。

```
dfhjvmout.IYK2ZIK1.0067240142.06004165342.txt
```

ここで、

- dfhjvmout は、ファイル名の固定部分です。
- IYK2ZIK1 は、CICS 領域のアプリケーション ID です。
- 0067240142 は、固有の JVM 番号です。
- 06004165342 は、JVM が作成された時を示すタイム・スタンプです。
- .txt はファイル接尾部です。

-generate オプションを使用すると、&APPLID; シンボルと &JVM_NUM; シンボルはファイル名では必要とされません。**-generate** によってそうした情報が自動的に提供されるためです。

-generate オプションには JVM 番号が含まれているので、結果の出力ファイルは、JVM に対して固有であり、**INQUIRE JVM** コマンドで識別される JVM 番号と一致します。このオプションには CICS 領域アプリケーション ID が含まれているため、複数の CICS 領域全体でも固有です。

このオプションは JVM サーバーには適用されません。

-Xdebug

JVM でデバッグ・サポートを使用可能にするかどうかを指定します。

詳細については、 223 ページの『Java アプリケーションのデバッグ』を参照してください。Oracle Technology Network Java Web サイトで入手可能な Java Platform Debugger Architecture (JPDA) に関する情報も参照してください。

プールされた JVM のデバッグ・セッションを正常に終了するには、デバッグ・サポートが使用可能な場合には REUSE=NO を指定します。

-Xms

ヒープの初期サイズを指定します。ストレージ・サイズは 1024 バイトの倍数で指定します。文字 K は KB を、文字 M は MB を、および文字 G は GB をそれぞれ表します。例えば、ヒープの初期サイズとして 6,291,456 バイトを指定するには、以下のいずれかの方法で **-Xms** をコーディングします。

```
-Xms6144K
-Xms6M
```

size を KB 数または MB 数として指定します。デフォルト値については、Default settings for the JVM を参照してください。

-Xmx
ヒープの最大サイズを指定します。なお、この固定ストレージ量は、JVM 初期設定時に JVM によって割り振られます。

size を KB 数または MB 数として指定します。

-Xshareclasses
共用クラス・キャッシュでクラス・データ共用を使用可能にする場合に、このオプションを指定します。JVM は既存のキャッシュに接続するか、キャッシュが存在しない場合は作成します。**-Xshareclasses** オプションにサブオプションを追加すると、複数のキャッシュを持つことができ、正しいキャッシュを指定することができます。詳しくは、Class data sharing between JVMs を参照してください。

このオプションは、プールされた JVM には適用されません。

JVM システム・プロパティー

システム・プロパティーには、JVM およびその環境を構成するための情報が含まれています。一部のシステム・プロパティーは、CICS 環境における JVM にとって特に重要です。

JVM システム・プロパティーは JVM プロファイルで指定します。プールされた JVM の場合、これらのオプションを JVM プロパティー・ファイルで設定して、さまざまなプロファイル間で同じオプションを共用することもできます。このファイルを参照するには、各 JVM プロファイルで JVMPROPS オプションを使用します。CICS は、JVM プロファイルまたは JVM プロパティー・ファイル内のすべてのシステム・プロパティーを、変更しないまま JVM に渡します。

JVM プロパティー・ファイルを使用する場合は、セキュリティー・ポリシー・ファイルなどの重要な JVM 構成オプションの定義に使用される JVM プロパティー・ファイルについては、更新権限をシステム管理者のみに制限して、ファイルのセキュリティーを確保してください。

JVM は、ここに記載されているものよりもはるかに広範囲のシステム・プロパティーをサポートできます。 111 ページの『JVM プロファイル: オプションおよびサンプル』では、システム・プロパティーに関して推奨される情報源をリストしています。

以下のリストには、一連の関連するシステム・プロパティーが含まれ、CICS 環境における使用方法を説明しています。 **-Dcom.ibm.cics** で始まるシステム・プロパティーは、CICS 環境における IBM JVM に固有のものです。 **-Dcom.ibm** (**.cics** なし) または **-Djava** で始まるシステム・プロパティーは、より広く使用されます。114 ページの『JVM プロファイルのコーディング規則』で説明されているコーディング規則に従って、各システム・プロパティーを指定してください。

-Dcom.ibm.cics.datasource.path=
DB2 にアクセスする CICS 内の Java アプリケーション用に JDBC 接続を生成するためにデプロイした、CICS 互換 DataSource の名前とサブコンテキストを指定します。詳しくは、「DB2 Guide」の『Acquiring a connection using the DataSource interface』を参照してください。

-Dcom.ibm.cics.ejs.nameserver=

JNDI 参照に使用するネーム・サーバーの URL および TCP/IP ポート番号を指定します。

- LDAP ネーム・サーバーの場合は、次のように指定します。

  ```
  -Dcom.ibm.cics.ejs.nameserver=ldap://myldserv.example.com:389
  ```

  myldserv.example.com はネーム・サーバーの URL であり、389 は、ネーム・サーバーが listen するように構成されるポート番号です。LDAP 管理者が適切な URL とポート番号を提供できます。

- 標準の COS Naming Directory Server の場合は、次のように指定します。

  ```
  -Dcom.ibm.cics.ejs.nameserver=iiop://mycsserv.example.com:900
  ```

  組織内の該当する管理者が適切な名前とポート番号を提供できます。

  WebSphere Application Server で提供される COS Naming Directory Server を使用する場合は、次のように指定します。

```
-Dcom.ibm.cics.ejs.nameserver=iiop://mycsserv.example.com:2809/domain/legacyRoot
```

  WebSphere Application Server では以下の条件が適用されます。

  - COS Naming Directory Server が使用するデフォルトの TCP/IP ポートは 2809 です。
  - CICS オブジェクトは、domain/legacyRoot と呼ばれる WebSphere 命名構造で特別に構成された場所にパブリッシュされなければなりません。CICS は、オブジェクトを、CORBASERVER 定義の JNDIPREFIX オプションで定義されたコンテキストにパブリッシュします。ここで、JNDI 接頭部は相対パスです。ネーム・スペースのルート・ノードからの /domain/legacyRoot パスを指定しない場合、CICS はオブジェクトをルート・ノード自体に対する相対的な JNDI 接頭部の位置にパブリッシュしようとします。 WebSphere Application Server で提供される COS Naming Directory Server を使用する場合、この試みは失敗します。

COS ネーミング・サービスを使用するときに、それを **-Djava.naming.provider.url** で指定することを選択した場合、ここで再度指定しないでください。

-Dcom.ibm.cics.ejs.loadjndiproperties=

一連の CICS 領域全体で共通の JNDI ネーム・サーバー構成プロパティーを含む、jndi.properties と呼ばれるファイルをセットアップします。デフォルトで、CICS は jndi.properties ファイルの位置を確認しようとしません。CICS にこの JVM の jndi.properties をロードさせるには、次のシステム・プロパティーを組み込んでください。

```
-Dcom.ibm.cics.ejs.loadjndiproperties=true
```

同じネーム・サーバー設定を共用させたいすべての領域について、関連するすべての JVM プロファイルで、jndi.properties ファイルを含むディレクトリーを JVM プロファイルの標準クラスパスに置きます。

-Dcom.ibm.cics.iiop.CSIv2Enabled=true

Common Secure Interoperability バージョン 2 (CSIv2) プロトコルに対する CICS サポートの ID アサーションを有効にします。このサポートをアクティブ

にするには、CICS 領域で使用されるすべての JVM プロファイルまたは JVM プロパティー・ファイルでこのシステム・プロパティーを指定します。このサポートが必須であるのは、WebSphere Application Server for z/OS から送信される IIOP メッセージに対するアサーション ID 認証を CICS CorbaServer がサポートしなければならない場合です。

-Dcom.ibm.cics.soap.validation.local.CCSID=
CICS WEBSERVICE リソースの検証が有効である場合、SOAP メッセージの検証時に使用するローカル・コード・ページを指定します。ローカル CCSID を指定しない場合、SOAP メッセージの検証時に、ご使用のシステムのデフォルトの USS コード・ページが取られます。

-Dcom.ibm.websphere.naming.jndicache.cacheobject={populated |none}
JNDI キャッシュをオンまたはオフにします。JNDI キャッシュは、JNDI 検索の結果をローカル・ストレージに保管します。その結果、おそらく別々のタスクで、アプリケーションが同じ検索を 2 回行う場合、結果は既に使用可能になっています。このキャッシュには次の特性があります。

- JVM 固有です。つまり、JVM ごとに別々のキャッシュがあります。
- IBM JNDI ネーム・サーバーのみを処理します。
- オブジェクト参照のみを保管し、DataSources などの他のものを保管しません。

populated
JNDI キャッシュがアクティブです。

none JNDI キャッシュは使用されません。

-Dcom.ibm.websphere.naming.jndicache.maxcachelife={20 *vmins***}**
JNDI キャッシュの「存続時間」を分数で指定します。この時間を超えた後でキャッシュにアクセスされる場合、キャッシュ全体からその内容がフラッシュされます。

-Dcom.ibm.websphere.naming.jndicache.cacheobject プロパティーも参照してください。

-Dcom.ibm.ws.naming.ldap.containerdn=
LDAP ネーム・サーバーのコンテナー識別名を指定します。例えば、次のようになります。

```
-Dcom.ibm.ws.naming.ldap.containerdn=ibm-wsnTree=t1,o=WASNaming,c=us
```

LDAP 管理者が、ご使用のシステムに適切な値を提供できます。

コンテナー識別名は、システム・ネーム・スペースのルートです。

COS ネーミング・サービスを指定する場合、このプロパティーは不要です。

-Dcom.ibm.ws.naming.ldap.noderootrdn=
LDAP ネーム・サーバーのノード・ルート相対識別名を指定します。例えば、次のようになります。

```
-Dcom.ibm.ws.naming.ldap.noderootrdn=ibm-wsnName=legacyroot,
ibm-wsnName=PLEX2,ibm-wsnName=domainRoots
```

LDAP 管理者が、適切な値を提供できます。

COS ネーミング・サービスを指定する場合、このプロパティーは不要です。

-Dfile.encoding=
エンコード方式を指定します。

-Djava.naming.security.authentication=
ネーミング操作に使用するセキュリティー認証のタイプを指定します。LDAP ネーム・サーバーを使用している場合、このプロパティーが必要なことがあります。

CICS には、LDAP ネーム・スペースへの書き込みアクセス権限が必要です。LDAP サービスが安全にセットアップされている場合、認証、資格情報、およびプリンシパルの 3 つのプロパティーが必要です。任意のユーザーが書き込めるように LDAP サービスがセットアップされている場合、これらの 3 つのプロパティーは不要です。 LDAP 管理者が、これらのプロパティーを JVM プロファイルまたはオプションの JVM プロパティー・ファイルに組み込む必要があるかどうかを指示する場合があります。

このプロパティーに対して CICS でサポートされる値は **simple** のみです。**-Djava.naming.security.authentication=simple** を指定すると、LDAP ネーム・サーバーがセキュア・モードで実行されることを示します。

重要:

このプロパティーを指定する場合は、**-Djava.naming.security.principal** および **-Djava.naming.security.credentials** も指定する必要があります。

これらのプロパティーは、CICS でセキュア LDAP サービスにアクセスする際に必要となるユーザー ID とパスワードを指定するため、これらのシステム・プロパティーを含むすべてのファイルへのアクセスを制御するファイル権限に特に注意してください。

-Djava.naming.security.credentials=
java.naming.security.principal で説明されている **principal** が LDAP ネーム・サーバーにアクセスするのに必要なパスワードを指定します。

-Djava.naming.security.authentication=simple を指定した場合、このプロパティーが必要です。指定する値は、LDAP 管理者から提供されます。例えば、次のとおりです。

```
-Djava.naming.security.credentials=secret
```

.

-Djava.naming.security.principal=
LDAP ネーム・サーバーへのアクセスに必要な **principal** を指定します。

-Djava.naming.security.authentication=simple を指定した場合、このプロパティーが必要です。指定する値は、LDAP 管理者から提供されます。例えば、**-Djava.naming.security.principal=cn=CICSUser,c=uk** です。

-Djava.security.manager={default| "" | |*other_security_manager***}**
JVM に使用可能にする Java セキュリティー・マネージャーを指定します。デフォルトの Java セキュリティー・マネージャーを使用可能にするには、次のいずれかの形式でこのシステム・プロパティーを組み込みます。

- `-Djava.security.manager=default`

- `-Djava.security.manager=""`
- `-Djava.security.manager=`

これらのステートメントはすべて、デフォルトのセキュリティー・マネージャーを使用可能にします。 **-Djava.security.manager** システム・プロパティーを JVM プロファイルに組み込まない場合、JVM は Java セキュリティーが無効な状態で実行されます。JVM の Java セキュリティーを使用不可にするには、このシステム・プロパティーをコメント化します。

-Djava.security.policy=
セキュリティー・マネージャーが JVM のセキュリティー・ポリシーの判別に使用する、追加のポリシー・ファイルの場所を記述します。デフォルトのポリシー・ファイルは、`/usr/lpp/java/J6.0.1_64/lib/security/java.policy` で JVM に提供されます。ここで、`java/J6.0.1_64` サブディレクトリー名は、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション をインストールする際のデフォルト値です。デフォルトのセキュリティー・マネージャーは常にこのデフォルト・ポリシー・ファイルを使用して、JVM のセキュリティー・ポリシーを判別します。**-Djava.security.policy** システム・プロパティーを使用すると、デフォルト・ポリシー・ファイルの他に、セキュリティー・マネージャーが考慮に入れる追加のポリシー・ファイルを指定することができます。

Java セキュリティーがアクティブであるときに、CICS Java アプリケーションが正常に実行できるようにするには、少なくとも、アプリケーションの実行に必要な権限を CICS に付与する追加のポリシー・ファイルを指定してください。

Java セキュリティーの有効化については、100 ページの『Java セキュリティー・マネージャーの有効化』を参照してください。

-Djdbc.drivers=
1 つ以上の 64 ビット JDBC ドライバーを指定します。ドライバー名をシステム・プロパティーとして設定するのは、Java アプリケーション自体が **Class.forName("driver_name");** コマンドを使用してドライバーをロードするのに代わる方法です。リスト内の各ドライバー名を : (コロン) で区切ってください。

DB2 提供の JDBC ドライバーを指定するには、次のシステム・プロパティーを設定します。

```
-Djdbc.drivers=com.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver
```

この共通名は、DB2 Universal JDBC Driver を含めて、DB2 によって提供されるすべてのレベルの JDBC ドライバーに有効です。

64 ビット・バージョンの JDBC ドライバーを使用する必要があります。JDBC について詳しくは、「DB2 Guide」の『Using JDBC and SQLJ to access DB2 data from Java programs』を参照してください。

DFHJVMAX、JVM サーバー用の JVM プロファイル

JVM プロファイル DFHJVMAX は、Axis2 JVM サーバーによって使用される CICS 提供の JVM プロファイルです。CICS 領域に対して DFHJVMAX が正しくセットアップされていることを確認してください。

DFHJVMAX JVM プロファイルの JVM オプション

```
##############################################################################
# JVM profile: DFHJVMAX                                                      #
#                                                                            #
#   This sample CICS JVM profile is for an Axis2 JVM server.                 #
#                                                                            #
##############################################################################
#
#                          Symbol Substitution
#                          -------------------
#
# The following substitutions are supported:
#   &APPLID;     => The applid of the CICS region.
#   &JVMSERVER;  => The name of the JVMSERVER resource.
#   &DATE;       => Date the JVMSERVER is enabled.  Dyymmdd
#   &TIME;       => Time the JVMSERVER is enabled.  Thhmmss
#
# Using substitutions means that you can use the same profile
# for multiple regions and still have unique working directories
# and output destinations for each region.
#
# With this substitution
#     ENV_VAR=myvar.&APPLID;.&JVMSERVER;.data
# becomes
#     ENV_VAR=myvar.ABCDEF.JSERVER1.data
# for a JVMSERVER resource with the name JSERVER1 in a CICS region with
# applid ABCDEF.
#
#*****************************************************************************
#
#                          Required parameters
#                          -------------------
#
# The set of supported CICS options for JVM servers
# differs from those used with JVM pools.
#
# JAVA_HOME specifies the location of the Java directory.
#
JAVA_HOME=/usr/lpp///&JAVA_HOME///
#
# Set the current working directory. If this environment variable is
# set, a change to the specified directory is issued before the JVM
# is initialized, and the STDIN, STDOUT and STDERR streams are
# allocated to this directory.
#
# If you do not specify this option, the current working directory is
# left unchanged and the STDIN, STDOUT and STDERR streams are allocated
# to the /tmp directory.
#
WORK_DIR=.
#
# Specify any directories that contain DLLs required at run time. For
# example, to use the IBM DB2 Driver for JDBC and SQLJ, add the
# directory containing the native DLLs to the LIBPATH_SUFFIX option.
# See the "Application Programming Guide and Reference for Java" relevant
# to the level of DB2 being used.
#
#LIBPATH_SUFFIX=
#
#*****************************************************************************
#
#                    JVM server specific parameters
#                    ------------------------------
#
# Use the JAVA_PIPELINE option to configure the JVM Server
# to support Java-based SOAP pipelines.
```

```
#
JAVA_PIPELINE=YES
#
#*********************************************************************
#
#                          Output redirection
#                          ------------------
#
# STDOUT, STDERR, STDIN, and JVMTRACE are allocated with file names
# beginning with &APPLID;.&JVMSERVER;. You can specify different file
# names using the STDOUT, STDERR, STDIN, and JVMTRACE environment
# variables.
#
# The default file name for JVMTRACE is dfhjvmtrc.
# To send the output to somewhere other than a file, specify a user
# output redirection class. CICS provides a sample that demonstrates
# this capability. JVMTRACE cannot be redirected.
#
#USEROUTPUTCLASS=com.ibm.cics.samples.SJMergedStream
#
#*********************************************************************
#
#                              JVM options
#                              -----------
#
# See "IBM SDK for z/OS platforms, Java Technology Edition, SDK Guide"
# or "IBM Developer Kit and Runtime Environment, Java Technology
# Edition, Diagnostics Guide" for information on all JVM options.
#
# JVM options which print output and then exit must not be specified
# because they will cause the creation of the JVM to fail. These
# options include: -version, -help, -?, -assert and -X.
#
# Use the following options to tune the JVM.
# -Xms    Initial Java heap size, for example -Xms64M
# -Xmx    Maximum Java heap size, for example -Xmx512M
# -Xmso   Initial stack size for native threads (default -Xmso256KB)
# -Xiss   Initial stack size for Java threads (default -Xiss128KB)
# -Xss    Maximum stack size for Java threads (default -Xss256KB)
#
#
# Omit these values from the profile to accept the JVM defaults,
# unless you have performed workload analysis and can provide
# tuned values from a stable workload.
#
# The -Xgcthreads option sets the maximum number of helper threads
# allowed for garbage collection. If you do not specify this option,
# the default is set to (the number of CPUs - 1).
#
# -Xgcthreads4
#
# The following option sets the Garbage collection Policy.
#
-Xgcpolicy:gencon
#
#*********************************************************************
#
#                Setting user JVM system properties
#                ----------------------------------
#
# Specify JVM system properties for a JVM server if required.
# Properties are key name and value pairs that
# contain basic information about the JVM and its environment. They are
# always prefixed with -D. 例えば、次のようになります。
#
# -Dcom.ibm.cics.some.property=some_value
#
```

```
#*********************************************************************
#
#              Unix System Services Environment Variables
#              ------------------------------------------
#
# Java Dump Options. See "IBM Developer Kit and Runtime Environment,
# Java Technology Edition Diagnostics Guide" or "IBM SDK for z/OS
# platforms, Java Technology Edition, SDK Guide" for information on all
# Java runtime options.
#
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,SYSDUMP),ONINTERRUPT(NONE)"
#
# Specify where JVM dumps are written to
#
#JAVA_DUMP_TDUMP_PATTERN=DUMP.&APPLID;.&JVMSERVER;.&DATE;.&TIME;
#
# Specify the local time zone
#
#TZ=CET-1CEST,M3.5.0,M10.5.0
#
```

DFHOSGI、JVM サーバー用の JVM プロファイル

JVM プロファイル DFHOSGI は、OSGi JVM サーバーによって使用される CICS 提供の JVM プロファイルです。CICS 領域に対して DFHOSGI が正しくセットアップされていることを確認してください。

DFHOSGI JVM プロファイルの JVM オプション

```
#*********************************************************************
# JVM profile: DFHOSGI                                               #
#                                                                    #
#   This sample CICS JVM profile is for a JVM server.                #
#                                                                    #
#*********************************************************************
#
#                         Symbol Substitution
#                         -------------------
#
# The following substitutions are supported:
#   &APPLID;     => The applid of the CICS region.
#   &JVMSERVER;  => The name of the JVMSERVER resource.
#   &DATE;       => Date the JVMSERVER is enabled.  Dyymmdd
#   &TIME;       => Time the JVMSERVER is enabled.  Thhmmss
#
# Using substitutions means that you can use the same profile
# for multiple regions and still have unique working directories
# and output destinations for each region.
#
# With this substitution
#     ENV_VAR=myvar.&APPLID;.&JVMSERVER;.data
# becomes
#     ENV_VAR=myvar.ABCDEF.JSERVER1.data
# for a JVMSERVER resource with the name JSERVER1 in a CICS region with
# applid ABCDEF.
#
# Note: The continuation character for use with JVMProfiles is '\'.
#*********************************************************************
#
#                         Required parameters
#                         -------------------
#
# The set of supported CICS options for JVM servers
# differs from those used with JVM pools.
#
# JAVA_HOME specifies the location of the Java directory.
```

```
#
JAVA_HOME=/usr/lpp///&JAVA_HOME///
#
# Set the current working directory. If this environment variable is
# set, a change to the specified directory is issued before the JVM
# is initialized, and the STDIN, STDOUT and STDERR streams are
# allocated to this directory.
#
# If you do not specify this option, the current working directory is
# left unchanged and the STDIN, STDOUT and STDERR streams are allocated
# to the /tmp directory.
#
WORK_DIR=.

# Specify any directories that contain DLLs required at run time. For
# example, to use the IBM DB2 Driver for JDBC and SQLJ, add the
# directory containing the native DLLs to the LIBPATH_SUFFIX option.
# See the "Application Programming Guide and Reference for Java" relevant
# to the level of DB2 being used.
#
#LIBPATH_SUFFIX=
#
#*********************************************************************
#
#                    JVM server specific parameters
#                    ------------------------------
#
# Use the OSGI_BUNDLES option to specify a list of middleware
# bundles that are installed and activated in the OSGi framework
# when the JVM is initialized.
# The list of bundles must be comma separated. The continuation
# character is '\'.
#
#OSGI_BUNDLES=/u/example/pathToBundleDirectory/B1.jar,\
#             /u/example/pathToBundleDirectory/B2.jar
#
# This option is used to specify, in seconds, how long the OSGi
# framework initialization, termination, and middleware bundles
# initialization are allowed to run before being timed out.
# The specified value must be in the range 1-60000. If it falls
# outside of this range then it will default to 60. If the
# initialization exceeds the limit, the JVMserver fails to initialize.
#
#OSGI_FRAMEWORK_TIMEOUT=60
#
#*********************************************************************
#
#                         Output redirection
#                         ------------------
#
# STDOUT, STDERR, STDIN, and JVMTRACE are allocated with file names
# beginning with &APPLID;.&JVMSERVER;. You can specify different file
# names using the STDOUT, STDERR, STDIN, and JVMTRACE environment
# variables.
#
# The default file name for JVMTRACE is dfhjvmtrc.
# To send the output to somewhere other than a file, specify a user
# output redirection class. CICS provides a sample that demonstrates
# this capability. JVMTRACE cannot be redirected.
#
#USEROUTPUTCLASS=com.ibm.cics.samples.SJMergedStream
#
#*********************************************************************
#
#                              JVM options
#                              -----------
#
```

```
# See "IBM SDK for z/OS platforms, Java Technology Edition, SDK Guide"
# or "IBM Developer Kit and Runtime Environment, Java Technology
# Edition, Diagnostics Guide" for information on all JVM options.
#
# JVM options which print output and then exit must not be specified
# because they will cause the creation of the JVM to fail. These
# options include: -version, -help, -?, -assert and -X.
#
# Use the following options to tune the JVM.
# -Xms    Initial Java heap size, for example -Xms64M
# -Xmx    Maximum Java heap size, for example -Xmx512M
# -Xmso   Initial stack size for native threads (default -Xmso256KB)
# -Xiss   Initial stack size for Java threads (default -Xiss128KB)
# -Xss    Maximum stack size for Java threads (default -Xss256KB)
#
# Omit these values from the profile to accept the JVM defaults,
# unless you have performed workload analysis and can provide
# tuned values from a stable workload.
#
# The -Xgcthreads option sets the maximum number of helper threads
# allowed for garbage collection. If you do not specify this option,
# the default is set to (the number of CPUs - 1).
#
# -Xgcthreads4
#
# The following option sets the Garbage collection Policy.
#
-Xgcpolicy:gencon
#
#*********************************************************************
#
#                 Setting user JVM system properties
#                 ----------------------------------
#
# Specify JVM system properties for a JVM server if required.
# Properties are key name and value pairs that
# contain basic information about the JVM and its environment. They are
# always prefixed with -D. 例えば、次のようになります。
#
# -Dcom.ibm.cics.some.property=some_value
#
#*********************************************************************
#
#              Unix System Services Environment Variables
#              ------------------------------------------
#
# Java Dump Options. See "IBM Developer Kit and Runtime Environment,
# Java Technology Edition Diagnostics Guide" or "IBM SDK for z/OS
# platforms, Java Technology Edition, SDK Guide" for information on all
# Java runtime options.
#
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,SYSDUMP),ONINTERRUPT(NONE)"
#
# Specify where JVM dumps are written to
#
#JAVA_DUMP_TDUMP_PATTERN=DUMP.&APPLID;.&JVMSERVER;.&DATE;.&TIME;
#
# Specify the local time zone
#
#TZ=CET-1CEST,M3.5.0,M10.5.0
#
```

DFHJVMPR、プールされた JVM の JVM プロファイル

JVM プロファイル DFHJVMPR は、共用クラス・キャッシュを使用するプールされた JVM のサンプル JVM プロファイルです。Java プログラムの PROGRAM リソースで JVM プロファイルまたは JVM サーバーを指定しない場合のデフォルトとして、このファイルが使用されます。

DFHJVMPR JVM プロファイルの JVM オプション

```
############################################################
# JVMProfile: DFHJVMPR
############################################################
#
# This is a sample CICS JVM Profile for JVMs that use the
# Shared Class Cache. This profile is the default profile
# for use with all CICS PROGRAMs defined with JVM(YES)
# unless specified otherwise.
#
######
#
# Symbol Substitution:
#
# If you use any of the following variable symbols in any of
# the variables below, they will be replaced with appropriate
# values. The variable symbols may be specified in upper or
# lower case.
#
#  Symbol       Replacement value
#  ------       -----------------
#
# &APPLID;      The APPLID of the CICS region
# &JVM_NUM;     The Unix Systems Services Process ID (pid)
#               of the JVM. This is guaranteed to be unique
# &DATE;        The current date in the format Dyymmdd
# &TIME;        The current time in the format Thhmmss
#
# With this substitution, for example
#     STDERR=dfhjvmerr.&APPLID;.&JVM_NUM;.&DATE;.&TIME;
# becomes
#     STDERR=dfhjvmerr.ABCDEF.0084214386.D081220.T135323
#
######
#
# ********* CICS-specific parameters ***********
#
JAVA_HOME=/usr/lpp/java/J6.0.1_64
WORK_DIR=.
REUSE=YES
CLASSCACHE=YES
#
# A JVM Properties file can optionally be used by supplying its
# full path and file name on the JVMPROPS option.
# See "Java Applications in CICS" for more information on JVM
# Properties Files.
#
# JVMPROPS=/u/example/pathToProperties/myJVMProps.data
#
STDIN=dfhjvmin
STDOUT=dfhjvmout
STDERR=dfhjvmerr
#
DISPLAY_JAVA_VERSION=NO
# Percentage of heap full which will trigger a scheduled GC
GC_HEAP_THRESHOLD=85
# Timeout value in minutes after which a JVM and its TCB become
```

```
# eligible for termination
IDLE_TIMEOUT=30
#
# Specify any directories containing DLLs needed at runtime.
# For example, to use the IBM DB2 Driver for JDBC and SQLJ,
# add the directory containing the native DLLs to the
# LIBPATH_SUFFIX. See the DB2 Application and Programming
# Guide for Java relevant to the level of DB2 being used.
#
#LIBPATH_PREFIX=
#LIBPATH_SUFFIX=
#
# Specify any directories containing application Java classes
# and jar files. (Uncomment the lines below if needed)
#
#CLASSPATH_SUFFIX=/u/example/pathToJarOrZipFile/jarfile.jar:\
#                 /u/example/pathToRootDirectoryForClasses
#
# Uncomment the line below to use the specified output redirection
# class.
#
#USEROUTPUTCLASS=com.ibm.cics.samples.SJMergedStream
#
######
#
# ********* Unix System Services Environment Variables ***********
#
# Java Dump Options. See "IBM Developer Kit and Runtime Environment,
# Java Technology Edition, Diagnostics Guide" for information on all
# Java runtime options.
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,CEEDUMP,SYSDUMP),ONINTERRUPT(NONE)"
#
# Specify where JVM dumps should be written to
#JAVA_DUMP_TDUMP_PATTERN=DUMP.JVM.TDUMP.&APPLID;.&JVM_NUM;.LATEST
#
# Specify the local timezone
#TZ=CET-1CEST,M3.5.0,M10.5.0
#
######
#
# ********* JVM options **************
#
-Xms16M
-Xmx16M
-Xmso128K
-Xiss64K
-Xss256K
```

DFHJVMCD、CICS 提供のシステム・プログラム用に予約された JVM プロファイル

JVM プロファイル DFHJVMCD は CICS 提供の JVM プロファイルであり、CICS 提供のシステム・プログラム、特に CICS 提供の CIRP 要求プロセッサー・トランザクションで使用されるデフォルトの要求プロセッサー・プログラム DFJIIRP で使用するために予約されています。また、CICS は、DFHJVMCD を使用して共用クラス・キャッシュの初期化と停止も行います。DFHJVMCD が CICS 領域に対して正しくセットアップされていることを確認してください。ただし、カスタマイズするのは必要な場合のみです。

102 ページの『DFHJVMCD のカスタマイズ』には、この JVM プロファイルでオプションをカスタマイズする方法が記載されています。

DFHJVMCD JVM プロファイルの JVM オプション

```
############################################################
# JVMProfile: DFHJVMCD
############################################################
#
# This is the CICS JVM Profile for use by CICS programs.
# It must have a valid value for JAVA_HOME.
# It must always be available in the directory specified by
# the JVMPROFILEDIR SIT parameter.
#
######
#
# Symbol Substitution:
#
# If you use any of the following variable symbols in any of
# the variables below, they will be replaced with appropriate
# values. The variable symbols may be specified in upper or
# lower case.
#
#  Symbol       Replacement value
#  ------       -----------------
#
# &APPLID;      The APPLID of the CICS region
# &JVM_NUM;     The Unix Systems Services Process ID (pid)
#               of the JVM. This is guaranteed to be unique
# &DATE;        The current date in the format Dyymmdd
# &TIME;        The current time in the format Thhmmss
#
# With this substitution, for example
#     STDERR=dfhjvmerr.&APPLID;.&JVM_NUM;.&DATE;.&TIME;
# becomes
#     STDERR=dfhjvmerr.ABCDEF.0084214386.D081220.T135323
#
######
#
# ********* CICS-specific parameters ***********
#
JAVA_HOME=/usr/lpp/java/J6.0.1_64
WORK_DIR=.
REUSE=YES
CLASSCACHE=NO
#
STDIN=dfhjvmin
STDOUT=dfhjvmout
STDERR=dfhjvmerr
######
#
# ********* Unix System Services Environment Variables ***********
#
# Java Dump Options. See "IBM Developer Kit and Runtime Environment,
# Java Technology Edition, Diagnostics Guide" for information on all
# Java runtime options.
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,CEEDUMP,SYSDUMP),ONINTERRUPT(NONE)"
#
# Specify where JVM dumps should be written to
#JAVA_DUMP_TDUMP_PATTERN=DUMP.JVM.TDUMP.&APPLID;.%JVM_NUM;.LATEST
#
# Specify the local timezone
#TZ=CET-1CEST,M3.5.0,M10.5.0
#
######
#
# ********* JVM options **************
#
-Xms16M
```

```
-Xmx16M
-Xmso128K
-Xiss64K
-Xss256K
```

第 6 章 Java アプリケーションの管理

Java アプリケーションを使用可能にした後、CICS 領域をモニターして、アプリケーションの動作を理解することができます。また、JVM および Language Environment エンクレーブを調整して、アプリケーションのパフォーマンスを最適化することもできます。

このタスクについて

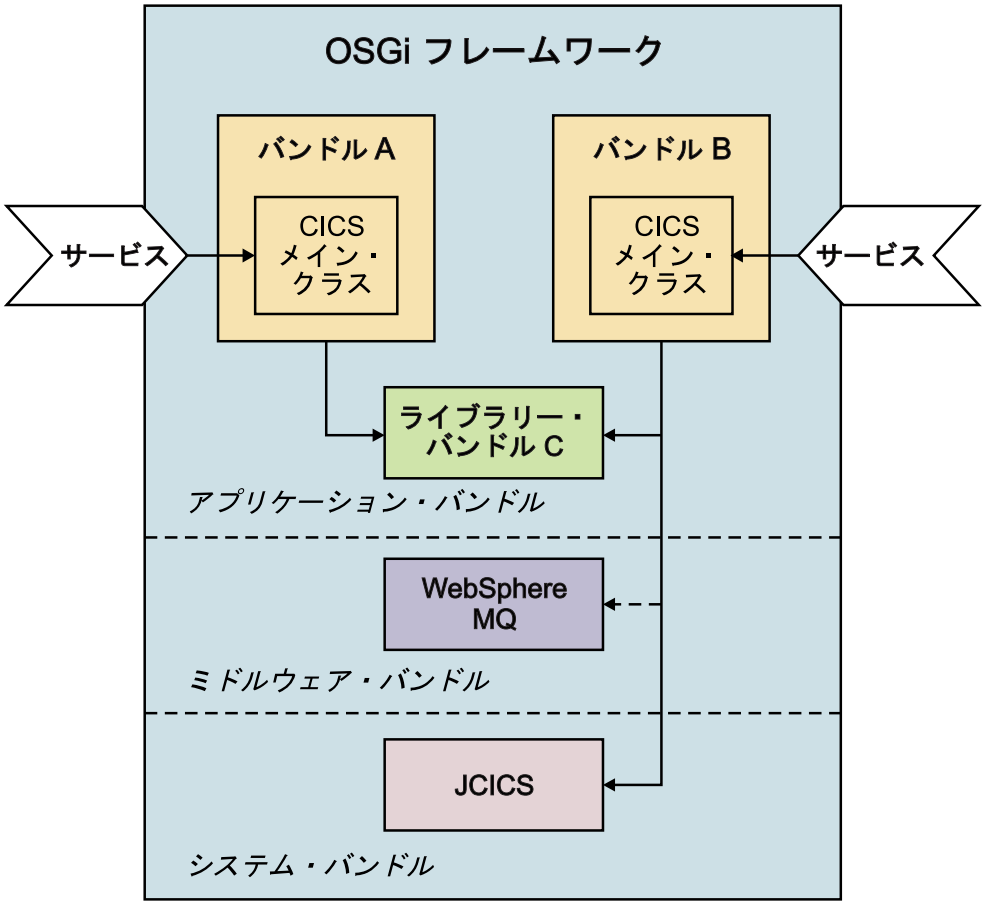
統計とモニターを使用して、CICS 領域における Java アプリケーションの動作に関する情報を収集することができます。特に、JVM の動作を確認することができます。情報を収集した後、JVM または Language Environment エンクレーブに変更を加えて、パフォーマンスを改善することができます。また、アプリケーションを無効にしたり、CICS 領域間で移動したりして、Java ワークロードをより効率よくバランス調整することもできます。

JVM サーバーにおける OSGi バンドルの更新

OSGi フレームワーク内の OSGi バンドルを更新するプロセスは、バンドルのタイプおよびその依存関係によって異なります。JVM サーバーを再始動することなく、アプリケーションの OSGi バンドルを更新できます。ただし、ミドルウェア・バンドルを更新するには、JVM サーバーの再始動が必要です。

このタスクについて

標準的な JVM サーバーでは、次の図に示されているように、OSGi フレームワークに OSGi バンドルの混合が入っています。



バンドル A とバンドル B は別々の Java アプリケーションであり、別々の CICS バンドルの OSGi バンドルとしてパッケージされています。両方のアプリケーションは、バンドル C でパッケージされている共通ライブラリーに依存しています。バンドル C は別個に管理され、更新されます。さらに、バンドル B は、WebSphere MQ ミドルウェア・バンドルと JCICS システム・バンドルに依存しています。

バンドル A と B の両方は、フレームワーク内の他のバンドルに影響を与えることなく、独立して更新できます。ただし、バンドル C を更新すると、それに依存する両方のバンドルに影響を与える可能性があります。バンドル C のどのエクスポートされたパッケージも、OSGI フレームワーク内のメモリーにとどまります。このため、バンドル C での変更を取り上げるには、バンドル A と B もこのフレームワークで更新する必要があります。

ミドルウェア・バンドルは、フレームワーク・サービスを含み、JVM サーバーのライフサイクルで管理されます。例えば、フレームワークに 1 回ロードしたいネイティブ・コードがあったり、WebSphere MQ などの別の製品にアクセスするためにドライバーを追加したりすることができます。

システム・バンドルは、OSGi フレームワークとの対話を管理するために CICS によって提供されます。これらのバンドルは、製品の一部として IBM によって保守されます。システム・バンドルの例は `com.ibm.cics.server.jar` ファイルです。こ

のファイルは、CICS サービスにアクセスするための大部分の JCICS API を提供します。

OSGi バンドルの更新

Java 開発者が更新されたバージョンの CICS バンドルを提供する場合、CICS バンドルを完全に置き換えるか、または新しいバージョンを段階的に導入してから、旧バージョンを除去することができます。

始める前に

新しいバージョンの OSGi バンドルを含む、更新された CICS バンドルが、zFS に存在しなければなりません。

このタスクについて

新しいバージョンを段階的に導入し、フレームワークで同時に両方のバンドルを実行させるには、OSGi サービスに別名が指定されなければなりません。別名が指定されない場合、このサービスはフレームワーク内で非アクティブとしてリストされます。既に実行中のサービスの重複と見なされるからです。

手順

- 既存の OSGi バンドルを置き換えるには、以下の手順を実行します。
  1. 更新したい CICS バンドルの BUNDLE リソースを使用不可にし、破棄します。 その CICS バンドルの一部である OSGi バンドルとサービスが、OSGi フレームワークから除去されます。
  2. オプション: 更新された CICS バンドルが別のディレクトリーにデプロイされる場合は、BUNDLE リソース定義を編集します。
  3. BUNDLE リソース定義をインストールして、変更された OSGi バンドルを取得します。 CICS バンドル内の OSGi バンドルとサービスが、OSGi フレームワークにインストールされます。
  4. CICS Explorer の「**Operations**」 > 「**Java**」ビューで、OSGi バンドルとサービスの状況を確認します。
- 既存のデプロイ済みバンドルと同時に、フレームワーク内で新しいバージョンの OSGi バンドルを作成するには、以下の手順を実行します。
  1. BUNDLE リソースを作成して、変更された CICS バンドルを取得します。 CICS バンドル内の OSGi バンドルとサービスが、OSGi フレームワークにインストールされます。 バンドル・マニフェストで別名が指定されている場合を除き、OSGi サービスは非アクティブ状態になります。
  2. CICS Explorer の「**Operations**」 > 「**Java**」ビューで、OSGi バンドルとサービスの状況を確認します。 2 つのバージョンの OSGi バンドルが OSGi バンドル・ビューにリストされます。別名が指定されている場合を除き、バンドルの OSGi サービスは非アクティブ状態になります。別名が指定される場合、両方の OSGi サービスがアクティブになります。
  3. 旧バージョンの OSGi バンドルを指し示す BUNDLE リソースを使用不可にします。 CICS は、そのバンドルに関連した OSGi サービスを除去し、OSGi バンドルを解決済み状態に設定します。 その結果、変更された OSGi バンドルの OSGi サービスは、非アクティブ状態からアクティブ状態に移ります。

4. OSGi サービスの別名がある場合、PROGRAM リソースでその別名を指定すると、JVM サーバーの外部から、更新されたアプリケーションを呼び出すことができます。

タスクの結果

OSGi バンドルのシンボル・バージョンが増え、Java コードが更新されたことを示します。更新された OSGi バンドルは、OSGi フレームワークで使用可能であり、JVM サーバーの外部から呼び出すことができます。

共通ライブラリーを含むバンドルの更新

他の OSGi バンドルで使用するための共通ライブラリーを含む OSGi バンドルは、特定の順序で更新される必要があります。

始める前に

新しいバージョンの OSGi バンドルを含む、更新された CICS バンドルが、zFS に存在しなければなりません。共通ライブラリーを別個の CICS バンドルで管理することがベスト・プラクティスです。その結果、これらのライブラリーのライフサイクルを、それらに依存するアプリケーションとは別個に管理できます。

このタスクについて

OSGi バンドルは、別の OSGi バンドルへの依存関係でサポートされているバージョンの範囲を指定するのが標準的です。範囲を使用すると、フレームワーク内で両立可能な変更を行う際の柔軟性が向上します。共通ライブラリーを含むバンドルを更新する場合、OSGi バンドルのバージョン番号が増えます。ただし、実行中のアプリケーションは、依存関係に対応するバージョンのバンドルを既に使用しています。最新バージョンのライブラリーを取得するには、アプリケーションの OSGi バンドルをリフレッシュする必要があります。したがって、特定のアプリケーションはさまざまなバージョンのライブラリーを使用するように更新し、その他のアプリケーションは旧バージョンで実行するままにすることが可能です。

共通ライブラリーを含む OSGi バンドルを更新する場合、その CICS バンドルを完全に置き換えることができます。ただし、クラスがライブラリーにロードされていない場合、従属バンドルがエラーを受け取る可能性があります。新しいバージョンのライブラリーを段階的に導入し、オリジナルのバージョンと一緒にフレームワークで実行することができます。OSGi バンドルにさまざまなバージョン番号がある限り、OSGi フレームワークは両方のバンドルを並行して実行できます。

手順

1. 新規バージョンの OSGi バンドルを指し示す BUNDLE リソースを作成します。 CICS は、OSGi フレームワークで新規バージョンの OSGi バンドルを作成します。既存の OSGi バンドルは、引き続き前のバージョンのライブラリーを使用します。
2. CICS Explorer で「OSGi Bundles」ビューを確認します。 リストには、フレームワーク内で異なるバージョンで実行中の同じ OSGi バンドル・シンボル名の 2 つの項目を示されます。

3. 従属 Java アプリケーションで新規バージョンのライブラリーを取得するには、以下の手順を実行します。
   a. Java アプリケーションの BUNDLE リソースを使用不可にし、破棄します。別の方法として、アプリケーションの可用性を維持するために、Java 開発者に、OSGi バンドルのバージョン情報を更新し、新規バージョンの CICS バンドルをデプロイしてもらうことができます。
   b. BUNDLE リソースをインストールします。 OSGi バンドルがフレームワークにロードされると、最新バージョンの共通ライブラリーを取得します。
4. CICS Explorer の「Bundles」ビューで BUNDLE リソースの状況を確認します。

タスクの結果

共通ライブラリーを含む OSGi バンドルを更新し、最新バージョンのライブラリーを使用するように Java アプリケーションを更新しました。

OSGi ミドルウェア・バンドルの更新

OSGi フレームワークで実行中のミドルウェア・バンドルを更新したい場合は、JVM サーバーを停止してから、再始動する必要があります。

このタスクについて

OSGi ミドルウェア・バンドルは、JVM サーバーの初期化中に OSGi フレームワークにインストールされます。例えば、パッチを適用したり、新しいバージョンを使用したりするために、ミドルウェア・バンドルを更新したい場合、変更されたバンドルを取得するために、JVM サーバーを停止してから、再始動する必要があります。

手順

1. 新しいバージョンのミドルウェア・バンドルが、CICS が読み取りおよび実行アクセス権限を持つ、zFS 上のディレクトリーにあることを確認します。CICS には、ファイルへの読み取りアクセス権限も必要です。
2. zFS ディレクトリー名またはファイル名が、JVM プロファイルで指定された値と異なる場合、JVM サーバーの JVM プロファイルで OSGI_BUNDLES オプションを編集します。 JVM プロファイルは、**JVMPROFILEDIR** システム初期設定パラメーターによって指定された zFS ディレクトリー内にあります。
3. JVMSERVER リソースを使用不可にして、JVM サーバーをシャットダウンします。 JVMSERVER を使用不可にすると、その JVM サーバーにインストールされている OSGi バンドルを含むすべての BUNDLE リソースも使用不可になります。
4. JVMSERVER リソースが、更新された JVM プロファイルを使用して JVM サーバーを開始できるようにします。 JVM サーバーが開始し、新しいバージョンのミドルウェア・バンドルを OSGi フレームワークにインストールします。また、CICS は、使用不可になった BUNDLE リソースを使用可能にし、更新されたフレームワークに OSGi バンドルとサービスをインストールします。

タスクの結果

OSGi フレームワークには、更新されたミドルウェア・バンドル、および JVM サーバーをシャットダウンする前にインストールされていた、Java アプリケーション用の OSGi バンドルとサービスが入っています。

JVM サーバーからの OSGi バンドルの除去

JVM サーバーから OSGi バンドルを除去したい場合は、CICS Explorer を使用して BUNDLE リソースを使用不可にするか破棄します。

このタスクについて

BUNDLE リソースは、CICS バンドルで定義される OSGi バンドルと OSGi サービスの集合に対するライフサイクル管理を行います。OSGi フレームワークから OSGi バンドルを除去しても、インストールされている他の OSGi バンドルやサービスの状態に自動的に影響を与えることはありません。別のバンドルの前提条件であるバンドルを除去しても、そのバンドルを明示的にリフレッシュするまで従属バンドルの状態は変わりません。

手順

1. 「**Operations**」 > 「**Java**」 > 「**OSGi Bundles**」をクリックして、OSGi バンドルが入っている BUNDLE リソースを検出します。
2. 「**Operations**」 > 「**Bundles**」をクリックして、BUNDLE リソースを使用不可にします。 CICS は、CICS バンドルで定義されている各リソースを使用不可にします。OSGi バンドルとサービスについては、CICS は JVM サーバーの OSGi フレームワークに要求を送信して OSGi サービスの登録を解除し、OSGi バンドルを解決済み状態にします。すべての未完了トランザクションが完了しますが、CICS アプリケーションから OSGi サービスへの新しいリンクはすべて、エラーを戻します。
3. BUNDLE リソースを破棄します。 CICS は OSGi フレームワークに要求を送信して、JVM サーバーから OSGi バンドルを除去します。

タスクの結果

OSGi バンドルとサービスを OSGi フレームワークから除去しました。

次のタスク

OSGi フレームワークに存在しなくなった OSGi サービスを指す PROGRAM リソースがある場合、それらの PROGRAM リソースを使用不可にし、破棄する必要がある可能性があります。

JVM サーバーへのアプリケーションの移動

プールされた JVM で Java アプリケーションを実行している場合、JVM サーバーで実行するようにそれらのアプリケーションを移動することができます。JVM サーバーは、同じ JVM 内で Java アプリケーションに対する複数の要求を処理できるので、同じワークロードの実行に必要な JVM 数を減らすことができます。

始める前に

アプリケーションがスレッド・セーフであり、1 つ以上の OSGi バンドルとしてパッケージされていることを確実にしてください。これらの OSGi バンドルは、1 つの CICS バンドルで zFS にデプロイされ、正しいターゲット JVMSERVER リソースを指定する必要があります。

Java 開発者は、51 ページの『CICS Explorer SDK を使用したアプリケーションのマイグレーション』で説明されているとおり、CICS Explorer SDK を使用すると、OSGi を使用して Java アプリケーションを再パッケージすることができます。

このタスクについて

既存の JVM サーバーを使用するか、アプリケーション用に JVM サーバーを作成することができます。スレッド限度と使用量が既に高い JVM サーバーにアプリケーションを移動しないでください。その JVM サーバーでロッキングの競合が生じる可能性があるからです。

手順

1. JVM サーバーを作成または更新します。
   - JVM サーバーを作成することを決定する場合は、93 ページの『JVM サーバーのセットアップ』を参照してください。プールされた JVM の JVM プロファイルの設定の多くは、JVM サーバーに適用されません。プールされた JVM プロファイルから DFHOSGI プロファイルにコピーできるオプションは、`LIBPATH_SUFFIX` オプションのみです。
   - 既存の JVM サーバーを使用する場合は、JVMSERVER リソースの THREADLIMIT 属性を増やして追加アプリケーションを処理するか、JVM サーバー・プロファイルのオプションを更新する必要がある可能性があります。JVMSERVER リソースまたは JVM プロファイルに対するすべての変更では、JVM サーバーを再始動して変更内容を取得する必要があります。
2. zFS でデプロイされたバンドルを指す BUNDLE リソースを作成します。BUNDLE リソースをインストールすると、CICS は、JVM サーバーの OSGi フレームワークに OSGi バンドルをロードします。OSGi フレームワークは、OSGi バンドルを解決し、OSGi サービスを登録します。 CICS Explorer を使用して、BUNDLE リソースが使用可能であることを確認してください。また、「OSGi Bundles」ビューおよび「OSGi Services」ビューを使用して、OSGi バンドルとサービスの状態を確認することもできます。
3. アプリケーションの PROGRAM リソースを更新します。
   a. EXECKEY 属性が CICS に設定されていることを確認します。 すべての JVM サーバーの作業は CICS キーで実行されます。
   b. JVM プロファイル名を除去し、JVMSERVER リソースの名前を入力します。
   c. JVMCLASS 属性が、Java アプリケーションの OSGi サービスと一致することを確認します。
   d. アプリケーションの PROGRAM リソースを再インストールします。

   PROGRAM リソースは、OSGi サービスを使用して、OSGi バンドルを JVM サーバー外部の他の CICS アプリケーションから使用可能にします。

タスクの結果

Java アプリケーションが呼び出されると、JVM サーバーで実行されます。

次のタスク

CICS Explorer の JVM サーバー・ビュー、および CICS 統計を使用して、JVM サーバーをモニターすることができます。パフォーマンスが最適でない場合は、スレッド限度を調整してください。

JVM サーバーのスレッド限度の管理

JVM サーバーでは、Java アプリケーションの実行に使用できるスレッド数が制限されています。また、各スレッドが 1 つの T8 TCB を使用するため、CICS 領域にもスレッド数の制限があります。CICS 統計を使用してスレッド限度を調整すると、領域内の JVM サーバー数と、各 JVM サーバーで実行されるアプリケーションのパフォーマンスとのバランスを取ることができます。

このタスクについて

各 JVM サーバーには、Java アプリケーションを実行するために最大 256 個のスレッドがあります。CICS 領域には最大 1024 個のスレッドを備えることができます。CICS 領域で多数の JVM サーバーが実行している場合、どの JVM サーバーにも最大値を設定できるわけではありません。すなわち、4 つの JVM サーバーで最大値を設定すると、CICS 領域で他の JVMSERVER リソースを使用可能にすることができません。各 JVM サーバーのスレッド限度を調整して、CICS 領域内の JVM サーバー数と、Java アプリケーションのパフォーマンスとのバランスを取ることができます。

スレッド限度は JVMSERVER リソースで設定されるので、初期値を設定し、CICS 統計を使用して、Java ワークロードのテスト時にスレッド数を調整してください。

手順

1. JVMSERVER リソースを使用可能にし、Java アプリケーションのワークロードを実行します。
2. 適切な統計間隔を使用して JVMSERVER リソース統計を収集します。 CICS Explorer で「**Operations**」 > 「**Java**」 > 「**JVM Servers**」ビューを使用するか、DFH0STAT 統計プログラムを使用することができます。
3. タスクでスレッドを待機した回数と時間を確認します。 「JVMSERVER thread limit waits」フィールドと「JVMSERVER thread limit wait time」フィールドに、この情報が含まれています。
   - これらのフィールドの値が高く、多数のタスクが JVMTHRD 待機で中断する場合、JVM サーバーには使用可能な十分なスレッドがありません。スレッド数を増やすと、プロセッサーの使用量が増える可能性があるので、十分な MVS リソースが使用可能であることを確認してください。
   - これらのフィールドの値が低く、ピーク時のタスク数が使用可能な最大スレッド数より少ない場合、スレッド限度を減らして、他の JVM サーバー用にスレッドを解放することができます。

4. MVS リソースが使用可能かどうかを確認するために、ディスパッチャー TCB プール統計と TCB モード統計を使用して、CICS 領域全体での T8 TCB の使用量を判断します。 JVM サーバーの各スレッドは 1 つの T8 TCB を使用し、1 つの領域内で 1024 個に制限されます。T8 TCB は複数の JVM サーバー間で共用できませんが、すべての TCB は 1 つの THRD TCB プール内にあります。待機している TCB 数とプロセッサー使用量が少ない場合、十分な MVS リソースが使用可能であることを示します。
5. JVM サーバーで実行できるスレッド数を調整するには、THREADLIMIT 属性を変更し、JVMSERVER リソースを再度使用可能にします。
6. Java アプリケーションのワークロードを再度実行し、統計を使用して、待機しているタスクの数が減ったことを確認します。

次のタスク

JVM サーバーのパフォーマンスを調整するには、181 ページの『JVM サーバーのパフォーマンスの改善』を参照してください。

CICS リスタート後の OSGi バンドル・リカバリー

OSGi バンドルを含む CICS 領域をリスタートすると、CICS は BUNDLE リソースをリカバリーし、OSGi バンドルを JVM サーバーのフレームワークにインストールします。

CICS バンドルにパッケージされる OSGi バンドルは、CSD に保管されません。BUNDLE リソース自体はカタログに保管されるため、CICS 領域のリスタート後、BUNDLE リソースが復元されると OSGi バンドルは動的に再作成されます。

CICS のコールド・リスタート、ウォーム・リスタート、または緊急リスタート後、JVM サーバーは、BUNDLE リソースのリカバリーとは非同期に開始されます。CICS リスタートで OSGi バンドルを正常に復元するには、JVM サーバーが完全に使用可能でなければなりません。したがって、BUNDLE リソースは CICS 開始の最後の段階時にリカバリーされますが、OSGi バンドルがインストールされるのは、JVM サーバーがその開始を完了したときのみです。

BUNDLE リソースとそれに含まれている OSGi バンドルは正しい順にインストールされ、CICS バンドルと OSGi バンドルの両方の間の依存関係がフレームワークで確実に解決されるようになります。CICS が OSGi バンドルをインストールできない場合、BUNDLE リソースは Disabled 状態でインストールされます。IBM CICS Explorer を使用すると、BUNDLE リソース、OSGi バンドル、および OSGi サービスの状態を表示できます。

プールされた JVM における Java アプリケーションの更新

プールされた JVM で実行される Java アプリケーションを変更する場合、変更されたリソースをロードするために、それらのアプリケーションを実行する JVM の停止と再始動が必要です。また、クラスパスでリソースまたはファイルに変更を加える場合も、プールされた JVM の停止と再始動が必要です。

始める前に

更新された Java アプリケーションをコンパイルし、パッケージして、z/OS UNIX ファイル・システムにデプロイする必要があります。

このタスクについて

JVM のクラスパスに Java アプリケーション・クラスを追加するか、ファイルの名前を変更することができます。JVM の実行中に、JVM は、JVM プロファイルの変更を認識しません。したがって、アプリケーションへの変更を取得するために、JVM の停止と再始動が必要です。

手順

1. オプション: アプリケーションの JVM プロファイルを編集して、新規クラスまたは変更されたクラスをクラスパスに追加します。 クラスまたは JAR ファイルの内容を変更したものの、同じ名前を保持する場合は、この手順を実行する必要はありません。
2. JVM を再始動して、アプリケーションの変更を取得します。 変更されたファイルをリストする JVM プロファイルごとに JVM プールを段階的に停止します。このアプリケーションを実行しない他の JVM は引き続き実行することができます。段階的に停止したプロファイルを使用する JVM を要求が待機している場合、CICS は新しい JVM を開始します。 変更されたクラスを JVM がロードすると、共用クラス・キャッシュが自動的に更新されるので、JVM を再始動する必要はありません。

タスクの結果

CICS は、更新されたバージョンの JVM プロファイルを使用してプールされた JVM を作成し、新しいクラスまたは変更されたクラスをロードします。

JVM stdout および stderr 出力をリダイレクトするための Java クラスの作成

JVM プロファイルの USEROUTPUTCLASS オプションを使用して、JVM からの stdout および stderr 出力を代行受信する Java クラスを指定します。このクラスを更新すると、適当なタイム・スタンプとレコード・ヘッダーを指定し、出力をリダイレクトすることができます。

CICS は、この目的に使用できるサンプル Java クラス com.ibm.cics.samples.SJMergedStream および com.ibm.cics.samples.SJTaskStream を用意しています。/usr/lpp/cicsts/cicsts42/samples/com.ibm.cics.samples ディレクトリーに、これらの両方のクラスのサンプル・ソースが提供されています。/usr/lpp/cicsts/cicsts42 ディレクトリーは、z/OS UNIX 上の CICS ファイル用のインストール・ディレクトリーです。このディレクトリーは、DFHISTAR インストール・ジョブの **USSDIR** パラメーターで指定されます。また、これらのサンプル・クラスは、クラス・ファイル com.ibm.cics.samples.jar として出荷時に付属しています。このクラス・ファイルは、/usr/lpp/cicsts/cicsts42/lib ディレクトリーにあります。これらのクラスを変更するか、サンプルに基づいて独自のクラスを作成することができます。

213 ページの『JVM stdout、stderr およびダンプ出力の場所の制御』には、以下に関する情報が記載されています。

- JVM からの出力のタイプで、USEROUTPUTCLASS オプションによって指定されるクラスによって代行受信されるものと、されないもの。使用するクラスは、代行受信する可能性があるすべてのタイプの出力を処理できなければなりません。
- 提供されたサンプル・クラスの動作。com.ibm.cics.samples.SJMergedStream クラスは、JVM 出力用とエラー・メッセージ用にマージされた 2 つのログ・ファイルを作成します。アプリケーション ID、日付、時刻、トランザクション ID、タスク番号、およびプログラム名を含む各レコードにヘッダーが付けられます。これらのログ・ファイルは、一時データ・キューが使用可能な場合は、その一時データ・キューを使用して作成されます。一時データ・キューが使用不可であるか、Java アプリケーションで使用できない場合は、z/OS UNIX ファイルを使用して作成されます。com.ibm.cics.samples.SJTaskStream クラスは、単一のタスクからの出力を z/OS UNIX ファイルに送信し、タイム・スタンプとヘッダーを追加して、単一のタスクに固有の出力ストリームを提供します。

プールされた JVM が、変更または作成された出力リダイレクト・クラスを使用するには、そのクラスが、JVM プロファイルまたはプロパティー・ファイルの該当するクラスパスのディレクトリーに存在しなければなりません。サンプル出力リダイレクト・クラスの JAR ファイルを含むディレクトリーは、該当するクラスパスに自動的に組み込まれます。そのため、明示的に JVM プロファイルで指定する必要はありません。独自のクラスを提供する場合は、標準クラスパスにディレクトリーを追加する必要があります。

JVM サーバーが出力リダイレクト・クラスを使用するには、出力リダイレクト・クラスを含む OSGi バンドルを作成する必要があります。バンドル・アクティベーターがクラスのインスタンスをフレームワーク内のサービスとして登録し、プロパティー com.ibm.cics.server.outputredirectionplugin.name=*class_name* を確実に設定しておく必要があります。定数 com.ibm.cics.server.Constants.CICS_USER_OUTPUT_CLASSNAME_PROPERTY を使用して、プロパティー名を取得できます。次のコードの抜粋は、バンドル・アクティベーターでサービスを登録する方法を示しています。

```
Properties serviceProperties = new Properties();
    serviceProperties.put(Constants.CICS_USER_OUTPUT_CLASSNAME_PROPERTY, MyOwnStreamPlugin.class.getName());
    context.registerService(OutputRedirectionPlugin.class.getName(), new MyOwnStreamPlugin(), serviceProperties);
```

OSGi バンドルを JVM プロファイルの OSGI_BUNDLES オプションに追加するか、または最初のタスクの実行時にバンドルがフレームワークに確実にインストールされるようにすることができます。どちらの方法を使用しても、USEROUTPUTCLASS オプションでクラスを指定する必要があります。

独自のクラスを作成することを決定した場合は、以下に関する知識が必要です。

- OutputRedirectionPlugin インターフェース
- 出力の予想される宛先
- 出力リダイレクト・エラーと内部エラーの処理

出力リダイレクト・インターフェース

CICS は、com.ibm.cics.server.jar に com.ibm.cics.server.OutputRedirectionPlugin と呼ばれるインターフェースを備えています。このインターフェースは、JVM からの stdout および stderr 出力を代行受信するクラスによって実装できます。提供のサンプルはこのインターフェースを実装します。

以下のサンプル・クラスが用意されています。

- このインターフェースを実装するスーパークラス com.ibm.cics.samples.SJStream。
- JVM プロファイルで指定されるクラスである、サブクラス com.ibm.cics.samples.SJMergedStream および com.ibm.cics.samples.SJTaskStream。

サンプル・クラスのように、ご使用のクラスがインターフェース OutputRedirectionPlugin を直接実装するか、このインターフェースを実装するクラスを拡張することを確認してください。スーパークラス com.ibm.cics.samples.SJStream から継承するか、同じインターフェースを持つクラス構造を実装することができます。どちらのメソッドを使用する場合でも、クラスは java.io.OutputStream を拡張する必要があります。

initRedirect() メソッドは、1 つ以上の出力リダイレクト・クラスで使用される 1 組のパラメーターを受け取ります。次のコードはインターフェースを示しています。

```
package com.ibm.cics.server;

import java.io.*;

public interface OutputRedirectionPlugin {

  public boolean initRedirect( String inDest,
                               PrintStream inPS,
                               String inApplid,
                               String inProgramName,
                               Integer inTaskNumber,
                               String inTransid
                              );
  }
```

スーパークラス com.ibm.cics.samples.SJStream には、com.ibm.cics.samples.SJMergedStream および com.ibm.cics.samples.SJTaskStream の共通コンポーネントが含まれています。これに含まれている initRedirect() メソッドは「false」を戻します。これは、このメソッドがサブクラス内の別のメソッドによってオーバーライドされる場合を除いて、出力のリダイレクトを事実上使用不可にします。これは、writeRecord() メソッドを実装しません。このようなメソッドは、出力リダイレクト・プロセスを制御するために任意のサブクラスによって提供されなければなりません。独自のクラス構造でこのメソッドを使用することができます。出力リダイレクトの初期化も、initRedirect() メソッドではなく、コンストラクターを使用して実行できます。

inPS パラメーターには、JVM のオリジナルの System.out 印刷ストリームまたはオリジナルの System.err 印刷ストリームのどちらかが入っています。基礎となるこれらのロギング宛先のどちらにでもロギングを書き込むことができます。これらの印刷ストリームのどちらかで close() メソッドを呼び出してはなりません。印刷ストリームは完全にクローズされたままになり、将来使用できないからです。

出力の予想される宛先

CICS 提供のサンプル・クラスは、JVM からの出力を、CICS 領域に固有のディレクトリーに送信します。このディレクトリー名は、その CICS 領域に関連したアプリケーション ID を使用して作成されます。独自のクラスを作成する場合、必要に応じて、複数の CICS 領域から、同じ z/OS UNIX ディレクトリーまたはファイルに出力を送信することができます。

例えば、単一のファイルを作成して、複数の異なる CICS 領域で実行される特定アプリケーションに関連した出力をそのファイルに含めることができます。

初期プロセス・スレッド (IPT) 以外のスレッドで実行される Java アプリケーションは、CICS 要求を行うことができません。これらのアプリケーションの場合、JVM からの出力は、USEROUTPUTCLASS に指定されたクラスによって代行受信されますが、CICS 機能 (一時データ・キューなど) を使用してリダイレクトすることはできません。提供されたサンプル・クラスの場合と同様に、これらのアプリケーションからの出力を z/OS UNIX ファイルに送信することができます。IPT で実行される Java アプリケーションの場合は、一時データ・キューなどの CICS 機能を使用して出力をリダイレクトすることができます。

出力リダイレクト・エラーと内部エラーの処理

ご使用のクラスで、CICS 機能を使用して出力をリダイレクトする場合、それらのクラスには、これらの機能を使用する際のエラーを処理するために適切な例外処理が含まれていなければなりません。

例えば、一時データ・キュー CSJO および CSJE に書き込もうとするときに、これらのキューに CICS 提供の定義を使用する場合、次の例外が TDQ.writeData によってスローされます。

- IOErrorException
- LengthErrorException
- NoSpaceException
- NotOpenException

ご使用のクラスが出力を z/OS UNIX ファイルに送信する場合、それらのクラスには、z/OS UNIX への書き込み時に発生するエラーを処理する適切な例外処理が含まれていなければなりません。これらのエラーで最も一般的な原因は、セキュリティー例外です。

クラスを USEROUTPUTCLASS オプションで指定する JVM で実行される Java プログラムには、クラスでスローされる可能性がある例外を処理する適切な例外処理が含まれていなければなりません。CICS 提供のサンプル・クラスは、例外を内部で処理します。これには、Try/Catch ブロックを使用してすべてのスロー可能な例外をキャッチしてから、問題を報告する 1 つ以上のメッセージを書き込みます。出力メッセージのリダイレクト中にエラーが検出されると、これらのエラー・メッセージは `System.err` に書き込まれ、リダイレクトに使用可能にします。しかし、エラー・メッセージのリダイレクト中にエラーが検出される場合、この問題を報告するメッセージは、要求を処理する JVM で使用される JVM プロファイルの STDERR オプションによって指定されるファイルに書き込まれます。このようにサンプル・クラスはすべてのエラーをトラップするため、これは、呼び出し側プログラムが出

力リダイレクト・クラスによってスローされる例外を処理する必要がないことを意味します。このメソッドを使用すると、呼び出し側プログラムへの変更を避けることができます。クラスで発行されるエラー・メッセージを、失敗した宛先にリダイレクトしようとして、出力リダイレクト・クラスをループに入れないように注意してください。

プールされた JVM の管理

CICS は、JVM の作成や再利用を含めて、プールされた JVM を管理するための複数のタスクを実行します。プールされた JVM と共用クラス・キャッシュをモニターし、Java 環境を調整してパフォーマンスを最適化することができます。

JVM プール内の JVM の開始と停止を行ったり、CICS 領域内のプールを使用不可にしたりすることができます。また、例えば、非アクティブな JVM をプールから除去するまでに CICS が待機する時間を決定するタイムアウトしきい値の変更などのオプションを、JVM プロファイルで調整することもできます。

CICS は、CICS 領域におけるプールされた JVM の動作に関する統計とモニターの情報を提供します。この情報を使用して、パフォーマンスを最適化するための Java 環境の調整に役立てることができます。最適なパフォーマンスの実現について詳しくは、175 ページの『第 7 章 Java パフォーマンスの改善』を参照してください。

プールされた JVM を CICS がアプリケーションに割り振る方法

アプリケーションが、プールされた JVM を使用して実行される Java プログラムを実行する場合、CICS はまず最初に、JVM プールで再利用可能な適切な JVM を検出しようとします。正しい JVM プロファイルおよび実行キーを持つ適切な JVM が使用不可である場合、CICS は、可能な場合は新しい JVM を作成するか、またはその選択メカニズムを使用して代替方法を決定します。

JVM が、Java プログラムの PROGRAM リソースで指定される JVM プロファイルおよび実行キー (USER または CICS) を使用して作成された場合、アプリケーションは、使用可能なプールされた JVM を再利用することができます。適切な JVM が使用可能である場合、CICS はその JVM を要求に割り当てます。

正しい JVM プロファイルと実行キーを持つ適切な JVM が使用不可であり、**MAXJVMTCBS** システム初期設定パラメーターで設定された限界にまだ到達しておらず、MVS ストレージが厳しく制限されていない場合、CICS は Java プログラム用に新しい JVM を作成します。新しい JVM には、そのプログラムに適切なプロファイルと実行キーがあります。

CICS が適切な JVM を検出できないときに、**MAXJVMTCBS** 限界に達したためか、または MVS ストレージが厳しく制限されているので、**MAXJVMTCBS** 限界に達した場合と同じように CICS が動作するために、新しい JVM を作成できない場合、CICS は、アプリケーションに JVM を提供する最適な方法を決定する必要があります。これには、CICS 領域におけるさまざまなタイプの JVM の必要性と比較して、アプリケーションにとっての JVM の必要性を評価する必要があります。CICS は、次のいずれかの方法で、JVM に対するアプリケーションの要求に対応することができます。

- 要求に対する正しい実行キーを持つものの、プロファイルが正しくないフリー JVM を選び、その JVM を破棄し、正しいプロファイルを使用して旧 JVM の TCB で JVM を再作成します。これはミスマッチ と呼ばれます。
- 実行キーが正しくないフリー JVM とその TCB を破棄し、正しい実行キーがある JVM と TCB で置き換えます。この状態はスチール またはスチーリング と呼ばれます。これは、TCB が、ある TCB モード (J8 または J9) から別の TCB モードに「スチール」されたためです。

ミスマッチとスチールはどちらもコストが高いため、これらの方法のいずれかを取る前に、CICS は利点があるかどうかを決定しようとします。 CICS 領域におけるさまざまなタイプの JVM の必要性という観点から見ると、適切な JVM が使用可能になるまでアプリケーションを待機させ、より多くの利点がある要求のためにフリー JVM を保持しておく方が、CICS の全体的なシステム・パフォーマンスにとって経済性が高い場合があります。 CICS には、この決定を行うための選択メカニズムがあります。

図 4 は、このプロセスの様子を示しています。

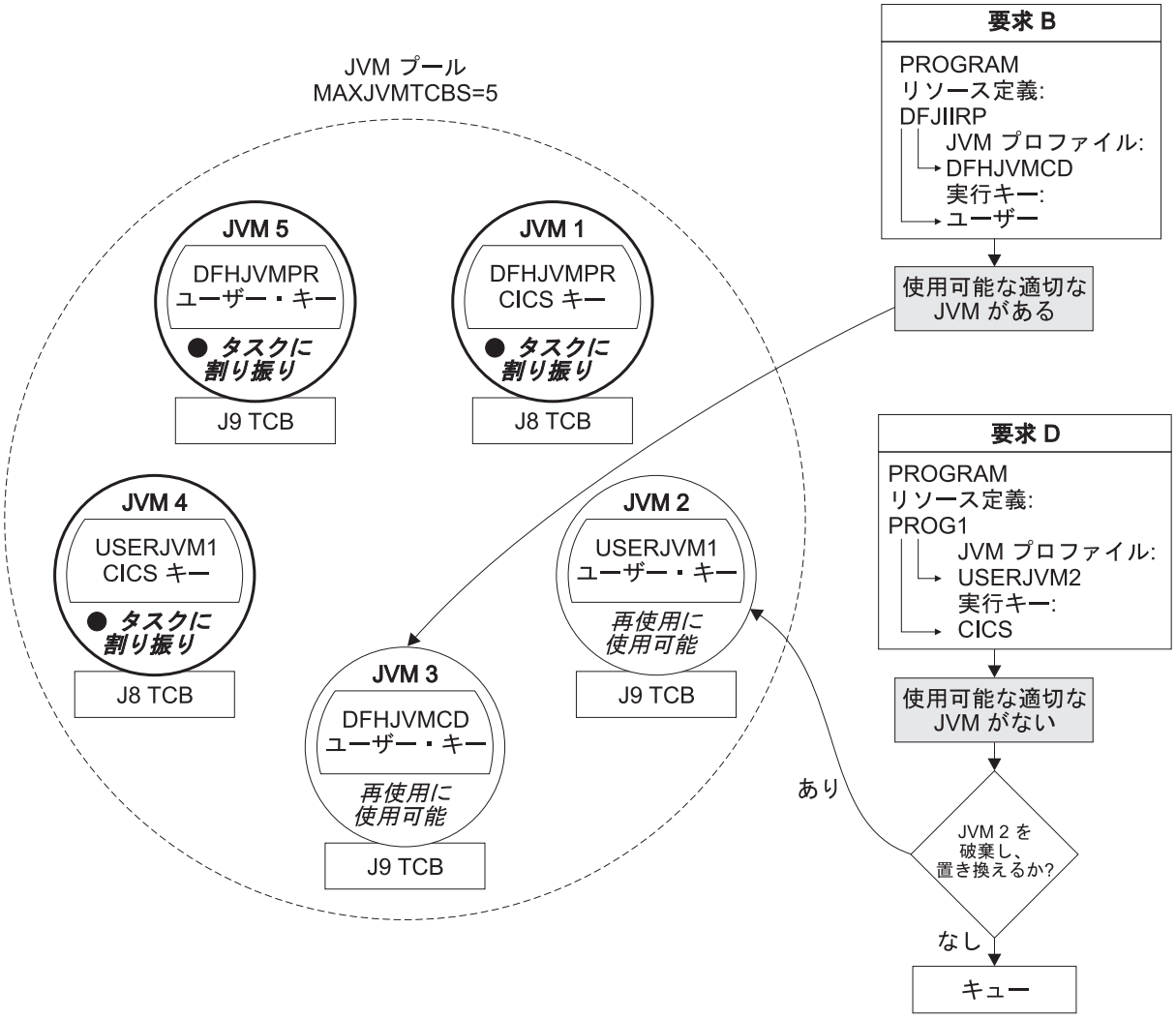


図 4. JVM に対する要求の処理: 例

要求 B は、デフォルトの要求プロセッサー・プログラム DFJIIRP に PROGRAM リソース定義を指定します。この定義は、JVM プロファイル DFHJVMCD と実行キー USER を指定します。CICS は JVM プールを調べて、JVM 3 に、この要求と一致する正しい JVM プロファイルと実行キーがあり、JVM 3 が再利用可能であることを検出します。CICS は、要求 B に JVM 3 を割り当てます。

要求 D は、PROG1 に PROGRAM リソース定義を指定します。この定義は、JVM プロファイル USERJVM2 と実行キー CICS を指定します。 CICS は JVM プールを調べます。フリー JVM である JVM 2 がありますが、JVM 2 のプロファイルと実行キーは、要求 D には正しくありません。**MAXJVMTCBS** 限界に達したので、CICS は、要求 D に対して新しい JVM を作成できません。したがって、CICS は選択メカニズムを使用して、JVM 2 とその TCB を破棄して、要求 D に合致する JVM と TCB で置き換えるかどうか、または要求 D を待機させ、さらに利点がある要求のために JVM 2 を保持しておくかどうかを決定する必要があります。要求 D を待機させる場合、要求 D は、JVM を待機する他の要求と一緒にキューに入れられます。

CICS は、次の 2 つの段階で JVM をアプリケーションに割り当てる決定を行います。

- JVM に対する着信要求を処理するための 1 組のアクションを取ります。
- JVM を待機する要求のキューがある場合、別の 1 組のアクションを取ります。

CICS が JVM に対する着信要求を処理する方法

JVM に対する着信要求を処理するために、CICS は以下に要約されているアクションを取ります。

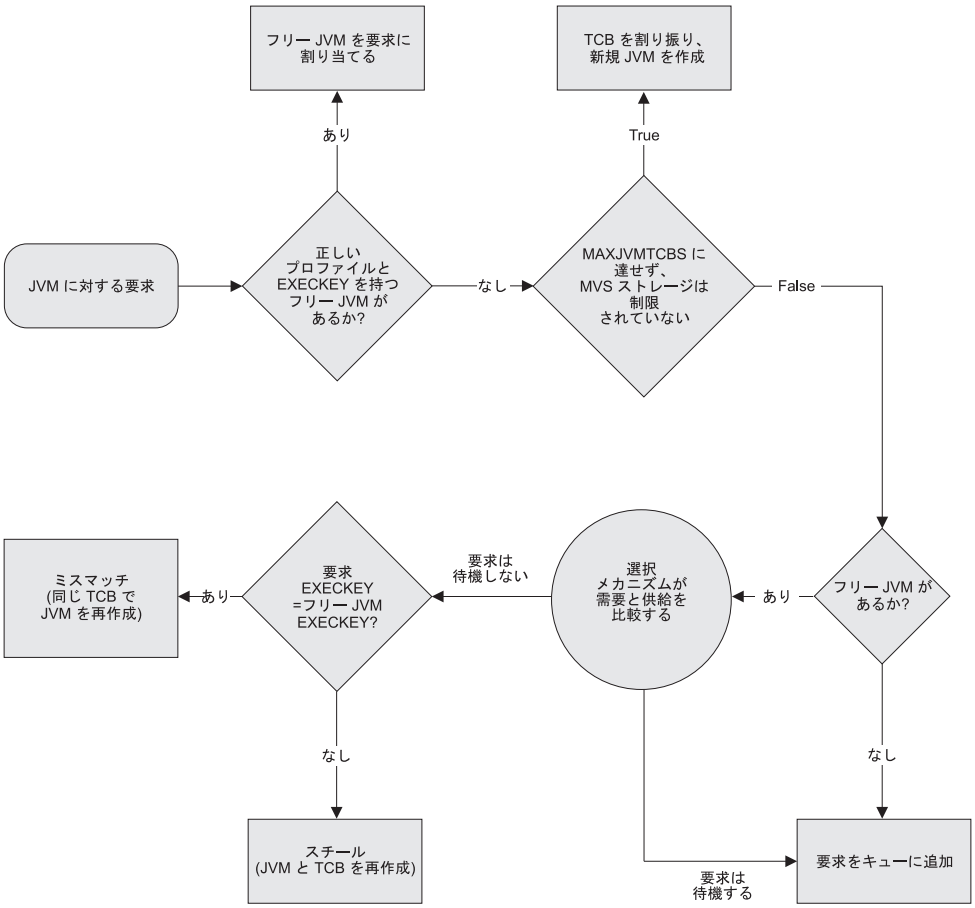


図 5. JVM に対する着信要求の処理

1. CICS が JVM に対する要求を受け取るときに、正しいプロファイルと実行キーのある JVM がフリーである場合、CICS はその JVM を着信要求に割り当てます。
2. 以下のいずれかが当てはまるときに、CICS が JVM に対する要求を受け取るとします。
   - フリー JVM がない
   - フリー JVM があるが、プロファイルと実行キーが要求に適切でない

   このときに、(**MAXJVMTCBS** 限界に達せず、MVS ストレージが厳しく制限されていないため) CICS はより多くの JVM を作成**できる**場合、TCB が割り振られ、その要求に対する新しい JVM が作成されます。
3. フリー JVM があるものの、プロファイルと実行キーが正しくないときに CICS が要求を受け取り、(**MAXJVMTCBS** 限界に達したか、MVS ストレージが厳しく制限されているため) CICS がそれ以上の JVM を作成**できない**場合、選択メカニズムが使用されます。選択メカニズムにより、要求が適切な JVM を待機するかどうか、またはフリー JVM のいずれかを受け取るかどうかが決まります。
   a. 要求がフリー JVM のいずれかを受け取る場合、ミスマッチかスチールのいずれかになり、JVM とおそらく TCB の再初期化が必要です。したがって、適切な場合は、選択メカニズムによりこの方法は避けられます。要求がフリ

ー JVM のいずれかを受け取ることが選択メカニズムにより決定される場合、CICS は、要求で指定される実行キーが、その JVM の実行キーと一致するかどうかを検査します。実行キーが一致しない場合、JVM とその TCB は破棄され、再初期化されます (スチール)。実行キーが一致するときに、JVM プロファイルのみが正しくない場合、JVM は同じ TCB で再初期化されます (ミスマッチ)。

b. フリー JVM のいずれかを受け取るのではなく、要求が待機することが選択メカニズムにより決定される場合、要求は、適切な JVM がフリーになるのを待機するためにキューに入れられます。

4. フリー JVM がないときに CICS が要求を受け取り、(**MAXJVMTCBS** 限界に達したか、MVS ストレージが厳しく制限されているため) CICS がそれ以上の JVM を作成**できない**場合、要求は、JVM がフリーになるのを待機するためにキューに入れられます。

JVM を待機する要求のキューを CICS が処理する方法

JVM を待機する要求のキューが CICS にある場合、CICS は以下のアクションを取ります。

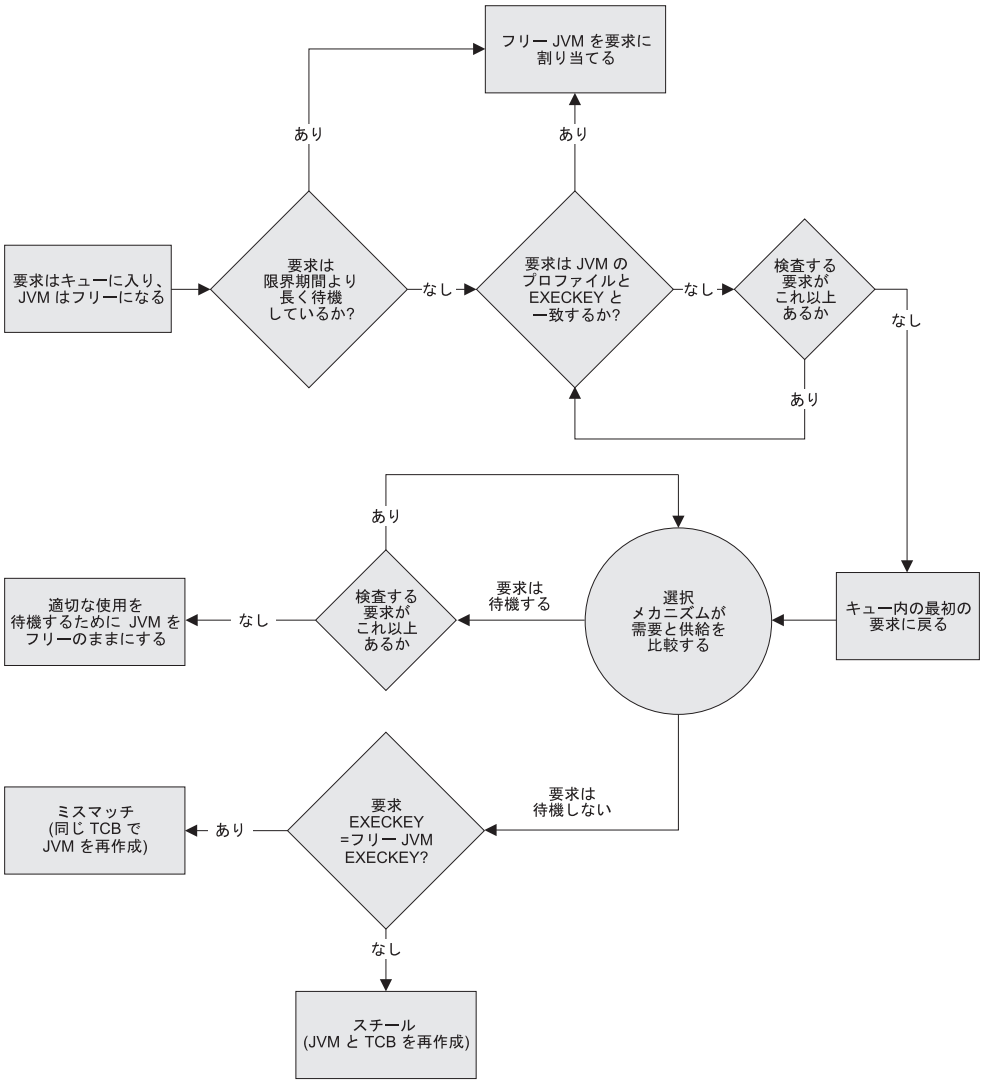


図 6. JVM を待機する要求のキューの処理

1. JVM がフリーになるのを待機する要求が、(CICS が決定する) 限界期間より長く待機していた場合、JVM のプロファイルと実行キーがどのようなものであっても、CICS は次に使用可能な JVM をその要求に提供します。これは、JVM がフリーでないためにキューに入れられた要求と、フリー JVM のプロファイルまたは実行キーが正しくないためにキューに入れられた要求の両方に適用されます。ミスマッチかスチールのいずれかになり、JVM とおそらく TCB は再初期化される可能性がありますが (要求がキュー内にあり、次のフリー JVM にたまたま正しいプロファイルと実行キーがある場合を除く)、要求はこれ以上待機しないので、このアクションを取る価値があります。
2. 複数の要求がキューに入っていて、1 つの JVM がフリーになるにもかかわらず、限界期間より長く待機していた要求がない場合、CICS はキュー全体をスキャンして、そのプロファイルと実行キーを持つ JVM を必要とする要求で、待機時間が最長のものを検出します。CICS は、正しいプロファイルと実行キーを指

定する、待機時間が最長の要求に、フリー JVM を提供します。したがって、この状態では、JVM の再初期化は不要であり、ミスマッチまたはスチールは避けられます。

3. フリー JVM のプロファイルおよび実行キーと一致する要求を CICS が検出できない場合、CICS は、再度キュー全体をスキャンし、選択メカニズムを使用して、フリー JVM の破棄と再初期化が有利である要求を探し、その要求が必要とするプロファイルと実行キーを持つ JVM としてそのフリー JVM を再初期化します。ミスマッチまたはスチールが発生しますが、相応な要求に対して発生することが選択メカニズムにより確実になります。
4. CICS が、フリー JVM の破棄と再初期化が有利である要求をキュー内で検出しない場合、その JVM はより適切な使用を待機するためにフリーのままになります。例えば、フリー JVM のプロファイルと実行キーを持つ JVM を必要とする要求を CICS が受け取る場合があります。またはキュー内の最初の要求が限界期間より長く待機しているので、フリー JVM が提供される場合があります。または、フリー JVM の破棄と再初期化が有利である要求を CICS が受け取る場合があります。

選択メカニズム

選択メカニズムが使用されるのは、着信要求がより適切な JVM を待機すべきかどうかを CICS が認識する必要がある場合、またはフリー JVM と一致しない要求のキューが CICS にあり、それらの要求のいずれかが、その JVM の取得、破棄および再初期化にふさわしいかどうかを CICS が認識する必要がある場合です。

これらの状態で、このメカニズムは、CICS 領域におけるさまざまなタイプの JVM の必要性について全体像を検討します。以下を調べることによって、各プロファイルと実行キーを持つ JVM の需要と供給を比較します。

- JVM の各タイプに対する最近の要求に関する履歴データ (需要)。
- プール内の各タイプの JVM の数、およびタスクでこれらの JVM が保持される時間 (供給)。

選択メカニズムでは、このデータを使用して、所定の要求が正しいプロファイルと実行キーを持つ JVM を待機する必要があるかどうか、またはフリー JVM が提供される必要があるかどうかを解明します。これと同じ方法が、JVM がフリーになるのをキュー内で待機する要求、またはフリー JVM があるにもかかわらず、プロファイルまたは実行キーが正しくない場合に行われる要求に有効です。どちらの場合も、要求が必要とするタイプの JVM (すなわち、そのプロファイルと実行キーを持つ JVM) に対する需要が、一般的に供給より*低い* ので、そのタイプの JVM としてフリー JVM を破棄し、再作成する価値がないことをデータが示す場合に、要求が待機させられます。選択メカニズムが要求のキューを調べる際には、その要求が必要とするタイプの JVM に対する需要が、一般的に供給より*高い* ことをデータが示す要求に到達するまで、キューを調べ続けます。この要求について、選択メカニズムは、そのタイプの JVM が CICS 領域で必要なため、そのタイプの JVM としてフリー JVM を破棄し、再作成する価値があることを決定し、フリー JVM をその要求に割り当てます。フリー JVM のプロファイルが正しくないものの、実行キーは正しい場合、これはミスマッチであり、JVM が再初期化されます。フリー JVM の実行キーが正しくない場合、これはスチールであり、TCB と JVM の両方が破棄され、再作成されます。したがって、JVM の再初期化と (必要に応じて)

TCB の再作成のオーバーヘッドが生じましたが、選択メカニズムにより、新しい JVM と TCB が、将来使用される可能性があるタイプのものであることが確実になりました。

特定の状況では、JVM に対して異常に多数の要求が、限界期間より長く待機している場合があります。例えば、システム・ダンプが取られたために、すべての処理が遅延する場合にこの状態が生じる可能性があります。この場合、要求が限界期間より長く待機していたときに通常行われるように、突き合わせを中止して、待機中の各要求に次に使用可能な JVM を提供するのではなく、CICS は、JVM プールの限界期間値を一時的に増やします。これにより、待機中の要求の突き合わせを実行することができ、異常なオーバーヘッドが避けられます。この状態が終わった後、CICS は限界期間値を減らして元に戻します。

手動による JVM の始動と終了および JVM プールの無効化

CICS は、アプリケーションの要求に応じて JVM を始動し、ワークロードが JVM を必要としない場合は、使用可能な JVM の数を自動的に削減します。さらに、CICS コマンドを使用して、JVM プールを制御することもできます。 JVM を始動および終了し、JVM プールを一時的に無効にすることができます。この手動による制御を使用すれば、JVM プロファイルへの変更をインプリメントしたり、JVM プールでのアクティビティーを中断することができます。また、これを使用して、アプリケーションの要求に先立って JVM を作成することもできます。

04 JVM プール内の容量を均衡の取れたレベルに保ち、アプリケーションからの要求を
04 満たすために、CICS は、通常、JVM の始動および終了を管理します。CICS は、
04 特にピーク要求時の複雑なワークロードのパフォーマンスを最適化する必要がある
04 場合に、プール内の JVM の数およびタイプを管理するための高度なメカニズムを
04 備えています。

04 特定の状況では、JVM を手動で始動または終了したい場合があります。

04 • CICS の実行中に JVM プロファイルまたは JVM プロパティー・ファイルを変
04 更する (クラスパスへの新規のクラスまたは JAR ファイルの追加を含む) 場合
04 は、JVM を更新する必要があります。
04 • Java ワークロードが通常のワークロードで予測可能であり、限られた数の各種の
04 JVM プロファイルが関係する場合、アプリケーションからの要求に先立って
04 JVM を始動し、必要になったらすぐに使用できるようにしておくことができま
04 す。

CICS コマンドを使用した JVM の始動

JVM を手動で始動するには、**EXEC CICS** または **CEMT PERFORM JVMPOOL** コマンドを使用します。始動する JVM の数と、その際に使用する JVM プロファイルおよび実行キーを指定する必要があります。

指定した数は、JVM プールに既に存在する JVM の数に追加されますが、CICS 領域用の **MAXJVMTCBS** の限度を超えてはなりません。**EXEC CICS** または **CEMT INQUIRE DISPATCHER** コマンドを発行することによって、これをチェックできます。**MAXJVMTCBS** は限度を示し、ACTJVMTCBS は現存する JVM の数を示します。

CICS は、すべての JVM を一度に始動するのではなく、短時間の間に開始をスケジュールします。それぞれの JVM は、始動すると同時にアプリケーションによって使用できます。 JVM がアプリケーションによって使用されない場合は、他のアイドル状態の JVM と同様に、JVM プロファイルで指定したタイムアウトしきい値で自動的に終了する対象になります。

JVM プロファイルへの変更をインプリメントするために JVM を終了したばかりであり、CICS 領域のアプリケーション・アクティビティーが低下している場合は、**PERFORM JVMPOOL** コマンドを使用して、変更を適用したタイプの JVM を始動できます。これによって、アプリケーションの要求を待たずに、変更したプロファイルによって JVM を始動でき、クラスパスで指定したクラスがロードできることを確認できます。

CICS 領域の Java ワークロードが通常のワークロードで予測可能な場合、要求に応じて CICS に始動させるのではなく、手動の始動機能を使用して、アプリケーションのニーズを見越した JVM プールを作成したい場合があります。この方法を使用すれば、ワークロードが増加している期間中のアプリケーションの遅延時間を削減できます。

タイムアウトしきい値 (デフォルトは 30 分) を構成し、ニーズに先行して JVM を始動することによって、要件に対して使用可能な十分な容量を常に備えた JVM プールを構築できます。例えば、ピークのワークロードを処理するのに十分な数の JVM を始動し、24 時間のアイドル状態の後に初めて自動終了の対象となるようにタイムアウトしきい値を設定することができます。 (CICS 領域の開始時に適切な数の JVM を始動するタスクをセットアップしたい場合があります。) このような JVM プールを使用すれば、1 日のうちで何度かワークロードが減少しても、CICS が JVM を自動的に終了することはありません。それらは、システムが拡張された期間の間にアイドル状態であったか、ワークロードが長期間にわたって減少した場合にのみ終了されます。

特定の JVM プロファイルを持つ JVM を手動で始動した場合、これらの JVM は、CICS によって始動された JVM と同様にミスマッチまたはスチーリングの対象となります。ミスマッチとスチーリングによって、JVM プロファイルまたはユーザー・キーが変更されるので、最初に JVM を始動したアプリケーションは、JVM を使用できなくなります。また、ミスマッチとスチーリングによって JVM が再始動されるので、あらかじめ JVM を始動する場合に得られたメリットが得られなくなる可能性があります。ミスマッチおよびスチーリングの可能性によって、CICS 領域内の各種の JVM プロファイルの数が増えるため、JVM プールを手動で構築したい場合は、アプリケーションが 1 つまたはごく少数の JVM プロファイルを使用する場合に、最大限のメリットが得られる可能性があります。

JVM の終了

JVM を終了するには、CEMT または **EXEC CICS PERFORM JVMPOOL** コマンドを使用します。 JVM プール内のすべての JVM を終了することも、JVM プロファイルを指定して、そのプロファイルを持つ JVM のみを終了することもできます。

変更内容を JVM プロファイルにインプリメントするか、新規のアプリケーション・クラスを追加するには、JVM を終了する必要があります。 標準クラスパス上の既存のクラスを変更する場合は、JVM を終了する必要はありません。スタンドア

ロン JVM の場合は、標準クラスパスの選択をお勧めしますが、再設定可能な JVM から継続 JVM へ移行中の場合は、引き続き、スタンドアロン JVM 内の共用可能アプリケーション・クラスパス上にクラスを持つことができます。

PERFORM JVMPOOL コマンドは、共用クラス・キャッシュを終了しません。 クラスが変更されるか、新規クラスが追加されると、共用クラス・キャッシュは自動的に更新されるので、この状態で共用クラス・キャッシュを終了する必要はありません。

アプリケーションの中断を最小化するには、JVM プロファイル、関連する JVM プロパティー・ファイル、またはそれを使用するアプリケーションに変更を加えた JVM プロファイルのみを終了するように試みます。 JVM プールのサブセットの終了は、JVM プール全体の終了より効率的です。必ず、変更によって影響を受ける JVM をすべて終了するようにしてください。例えば、変更した共用 Java クラスは、複数の JVM プロファイルのクラスパス上にリストされる可能性があります。特定の特殊な環境では、アプリケーション・クラスが複数のプロファイルを持つ JVM によって使用される場合がありますが、これは JVM プロファイルからは明白ではない可能性があります。これは、例えば、カスタム・クラス・ローダーを使用する場合、リフレクションを通してクラスをインスタンス化する場合、または他のエンタープライズ Bean を呼び出すエンタープライズ Bean がある場合に問題となる可能性があります。複数のプロファイルを持つ JVM によってアプリケーション・クラスが使用されているかどうかが確かでない場合は、安全策を取って JVM プール全体を終了することを選択したいかもしれません。

CICS は、JVM のタイプごとにアプリケーションから要求を受け取ると同時に新規 JVM を始動します。必要であれば、**PERFORM JVMPOOL** コマンドを使用して手動で JVM を始動することができます。 JVM プロファイルを変更した場合は、新規の JVM は変更されたオプションを使用します。 Java アプリケーションを変更した場合は、新規の JVM は新規のクラスまたは変更されたクラスをロードします。

JVM プールの無効化

JVM プールのアクティビティーをすべて中断するには、**EXEC CICS** または **CEMT SET JVMPOOL** コマンドを使用して、状況を DISABLED に設定します。この状態では、JVM プールは新規の要求を処理できません。

JVM プールを使用不可にすると、その中の JVM は保存されますが、JVM プールを再び使用可能にするまでは、新規の Java プログラムがそれらを使用することはできません。既に JVM を使用している Java プログラムは、完了まで実行することができます。 JVM プールを再度有効にするには、**EXEC CICS** または **CEMT SET JVMPOOL** コマンドを使用して、状況を ENABLED に設定します。

共用クラス・キャッシュの開始

デフォルトでは、共用クラス・キャッシュの使用を求めるプロファイルを持つプールされた JVM で Java アプリケーションを実行する要求を CICS が受け取るとすぐに、共用クラス・キャッシュが自動的に開始します。任意の時点で共用クラス・キャッシュを停止した後で、再開始したい場合は、自動開始を使用可能にするか、CICS コマンドを使用することができます。

このタスクについて

JVMCCSTART システム初期設定パラメーターは、共用クラス・キャッシュの通常の開始動作を制御します。デフォルト設定は AUTO です。AUTO では、プールされた JVM で共用クラス・キャッシュが要求されると直ちに、共用クラス・キャッシュが開始します。ウォーム・スタートまたは緊急スタート後に CICS 領域がシャットダウンするときに、共用クラス・キャッシュがアクティブである場合、z/OS の IPL などの一部の状況を除いて、共用クラス・キャッシュは通常、持続されます。

手順

1. **JVMCCSTART** システム初期設定パラメーターの値が AUTO または YES に設定されていることを確認します。 クラス・キャッシュは、最初のプールされた JVM で要求されるときに開始します。
2. CICS が実行中に共用クラス・キャッシュを再開始するには、次のいずれかの方法を使用します。
   - 共用クラス・キャッシュを即時に再開始するには、**CEMT PERFORM CLASSCACHE START** コマンド (または同等の **EXEC CICS** コマンド) を使用します。自動開始を使用可能にしたい場合は、このコマンドで AUTOSTARTST オプションを使用します。共用クラス・キャッシュのサイズを変更したい場合は、このコマンドで CACHESIZE オプションを使用することができます。
   - JVM で共用クラス・キャッシュが要求されるときに開始するように共用クラス・キャッシュを設定するには、**CEMT SET CLASSCACHE AUTOSTARTST** コマンド (または同等の **EXEC CICS** コマンド) を使用して、CICS の実行中の自動開始を使用可能にします。

タスクの結果

共用クラス・キャッシュを要求するプールされた JVM で Java アプリケーションを実行する要求を CICS が受け取るときに、共用クラス・キャッシュは再開始されます。それ以降の CICS のウォーム・スタートまたは緊急スタートでは、自動開始のこの設定が使用されます。ただし、開始時のオーバーライドとして **JVMCCSTART** システム初期設定パラメーターを指定した場合を除きます。

共用クラス・キャッシュのサイズの調整

共用クラス・キャッシュが開始すると、キャッシュ内のストレージ量は固定されます。デフォルトのサイズは 24 MB です。共用クラス・キャッシュ内のストレージが満杯になると、共用クラス・キャッシュに既に存在しているクラスは引き続き使用できますが、新しいクラスを追加することはできません。この状態では、共用クラス・キャッシュのサイズを増やす必要があります。

このタスクについて

CICS が提供するコマンドとパラメーターは、プールされた JVM の共用クラス・キャッシュのサイズを制御するのに役立ちます。JVM サーバーでクラス・キャッシュを使用する場合は、これらのコマンドを使用できません。Java によって提供されるサポートを使用する必要があります。Java 共用クラス・ユーティリティーについて詳しくは、Java Diagnostics Guide を参照してください。

共用クラス・キャッシュのサイズは、共用クラス・キャッシュを使用するすべてのプールされた JVM に対して標準クラスパスで指定されている、アプリケーションのすべてのクラスを含むのに十分なサイズでなければなりません。共用クラス・キャッシュは、共用可能なアプリケーション・クラスと共用不能なアプリケーション・クラスとを区別せず、JIT コンパイル・コードを含みません。

手順

1. ご使用のアプリケーション・クラスに必要なストレージを見積もるか、またはより良い結果を得るために、テスト環境でアプリケーションを実行して、共用クラス・キャッシュで必要な合計スペースを特定します。
   a. 共用クラス・キャッシュを使用して、テスト環境で各アプリケーションを繰り返し実行します。
   b. アプリケーションの実行中に、共用クラス・キャッシュ内のフリー・スペース量をモニターします。 CACHESIZE および CACHEFREE オプションを指定した **INQUIRE CLASSCACHE** コマンドを使用して、共用クラス・キャッシュのサイズ、および共用クラス・キャッシュ内のフリー・ストレージ量に関するレポートを作成します。

      z/OS UNIX システム・サービス・シェルで次のコマンドを実行すると、共用クラス・キャッシュの追加統計を取得することができます。

      ```
      java -Xshareclasses:name=CICS_sharedcc_APPLID_n,printStats
      ```

      ここで、*APPLID* は CICS システムの z/OS Communications Server アプリケーション ID、*n* は共用クラス・キャッシュの現行の世代番号です。
   c. フリー・スペース量が安定するまで、アプリケーションを実行します。
   d. 共用クラス・キャッシュを使用するアプリケーションごとにこのプロセスを繰り返します。
   e. 各アプリケーションで使用されるストレージ量を加算し、将来のアプリケーションの変更に対処するために適切な安全マージンを加えます。

   この合計により、共用クラス・キャッシュのおおよそのサイズが分かります。
2. **PERFORM CLASSCACHE RELOAD** コマンドを使用して、新しい共用クラス・キャッシュを作成します。このコマンドで CACHESIZE オプションを使用すると、新しい共用クラス・キャッシュのサイズを指定できます。 このコマンドにより、共用クラス・キャッシュを使用するプールされた JVM への悪影響が最小限に抑えられます。
3. オプション: JVMCCSIZE システム初期設定パラメーターの値を変更します。 このパラメーターは、共用クラス・キャッシュの初期サイズを指定し、CICS のコールド・リスタートや初期リスタートで使用されます。

タスクの結果

CICS が実行中に共用クラス・キャッシュに新しいサイズを指定すると、それ以降の CICS ウォーム・リスタートや緊急リスタートで新しい値が使用されます。CICS の初期リスタートまたはコールド・リスタートでは、**JVMCCSIZE** システム初期設定パラメーターからの値が使用されます。

共用クラス・キャッシュの終了

CICS を使用して、プールされた JVM で使用される共用クラス・キャッシュを終了させ、再開始しないようにすることができます。また、共用クラス・キャッシュを使用しているすべてのプールされた JVM も終了させることができます。

このタスクについて

共用クラス・キャッシュを終了させるときに、自動開始が有効である場合、プールされた JVM が共用クラス・キャッシュの使用を要求すると直ちに、新しい共用クラス・キャッシュが作成されます。共用クラス・キャッシュを再開始することなく終了させたい場合は、自動開始を使用不可にする必要があります。

共用クラス・キャッシュを終了させるときに、共用クラス・キャッシュが再開始しない場合、共用クラス・キャッシュを使用するプールされた JVM は実行できません。

CICS の実行中に共用クラス・キャッシュの自動開始状況を変更すると、それ以降の CICS ウォーム・リスタートでは、最新の設定が使用されます。CICS 領域が INITIAL または COLD として開始するか、**JVMCCSTART** システム初期設定パラメーターが開始時のオーバーライドとして指定される場合、そのシステム初期設定パラメーターからの設定値が使用されます。

手順

1. 共用クラス・キャッシュでの自動開始の状況を確認します。 CICS Explorer の JVM クラス・キャッシュ・オペレーションのビューまたは **INQUIRE CLASSCACHE** コマンドを使用することができます。
2. 共用クラス・キャッシュを終了するときに再開始させたくない場合は、自動開始を使用不可にします。 共用クラス・キャッシュの自動開始を使用不可にするには、次の 3 とおりの方法があります。
   - コマンドを入力して共用クラス・キャッシュを終了させる前に、**SET CLASSCACHE AUTOSTARTST** コマンドを使用して自動開始を使用不可にします。
   - **PERFORM CLASSCACHE** コマンドを入力して共用クラス・キャッシュを終了する場合は、AUTOSTARTST オプションを使用して自動開始を使用不可にします。
   - 次回の CICS 実行の自動開始を使用不可にするために、**JVMCCSTART** システム初期設定パラメーターを NO に設定します。この設定では、CICS の初期スタートまたはコールド・スタートで自動開始が常に妨げられます。領域がシャットダウンするときに共用クラス・キャッシュがアクティブである場合、**JVMCCSTART** をオーバーライドとして指定する場合であっても、共用クラス・キャッシュはウォーム・スタートまたは緊急スタート後も持続します。
3. 共用クラス・キャッシュ、および共用クラス・キャッシュを使用するすべてのプールされた JVM を終了させます。 JVM クラス・キャッシュ・オペレーションのビューまたは **PERFORM CLASSCACHE** コマンドを使用することができます。JVM をパージまたは強制パージするか、削除前に JVM に現行の Java プログラムの実行を終了させることができます。 共用クラス・キャッシュを使用しない JVM は、このコマンドの影響を受けません。
4. 共用クラス・キャッシュを再開始したくないときに、共用クラス・キャッシュを使用するプールされた JVM がアクティブなままである時間が長すぎる場合、

PERFORM CLASSCACHE コマンドを繰り返して、プールされた JVM を使用するタスクのパージを試みます。 共用クラス・キャッシュの自動開始が使用不可である場合のみ、このコマンドを繰り返してください。このコマンドは、最新の共用クラス・キャッシュと、引き続き JVM が使用する、領域内の任意の旧共用クラス・キャッシュの両方で作動します。自動開始が使用可能であるときに、このコマンドを繰り返して共用クラス・キャッシュを終了させる場合、このコマンドにより、自動開始機能によって開始された新しい共用クラス・キャッシュが終了する可能性があります。

タスクの結果

プールされた JVM のクラス・キャッシュが正常に終了します。

共用クラス・キャッシュのモニター

CICS コマンドを使用して、プールされた JVM の共用クラス・キャッシュおよびプール内の JVM ごとの共用クラス・キャッシュの状況を報告することができます。

手順

- プールされた JVM の共用クラス・キャッシュの状況を報告するために、**CEMT INQUIRE CLASSCACHE** コマンド (または同等の **EXEC CICS** コマンド) を使用します。 このコマンドは、共用クラスが初期化中であるか (STARTING)、使用する準備ができたか (STARTED)、再ロード中であるか (RELOADING)、またはアクティブでないか (STOPPED) を示します。 このコマンドは、自動開始の状況、共用クラス・キャッシュのサイズ、およびキャッシュ内のフリー・スペース量などの情報も示します。 また、このコマンドは、CICS 領域内で段階的に停止中のすべての旧共用クラス・キャッシュも報告します。
- JVM プール内の JVM の状況を報告するために、**CEMT INQUIRE JVM** コマンド (または同等の **EXEC CICS** コマンド) を使用します。 このコマンドは、指定された JVM またはプール内の各 JVM に関する情報を知らせ、JVM が割り振られているタスク、実行キーが USER であるか、CICS であるか、および共用クラス・キャッシュを使用するかどうかを示します。

JVM プールのモニター

CEMT INQUIRE JVMPOOL コマンド (または同等の **EXEC CICS** コマンド) を使用して、JVM プールに関する情報を検出することができます。

このコマンドは、以下の内容を示します。

- プール内の JVM 数。
- 削除のマークが付けられたものの、引き続きタスクで使用されている JVM の数。
- JVM プールが使用可能であるか、使用不可であるか (すなわち、新しい要求を処理できるかどうか)。
- どのトレース・オプションがプール内の JVM に適用されるか (このオプションは、**EXEC CICS** バージョンのコマンドのみで使用可能です)。

JVM プール内の JVM のモニター

EXEC CICS INQUIRE JVM コマンドまたは **CEMT INQUIRE JVM** コマンドを使用して、JVM プール内の各 JVM の状況を特定し、報告することができます。また、CICS 統計を使用して、JVM プール内のアクティビティーをモニターすることもできます。

EXEC CICS INQUIRE JVM コマンドを使用すると、特定の JVM について調べるか、JVM プール内のすべての JVM をブラウズすることができます。CEMT INQUIRE JVM コマンドを使用すると、JVM プール内のすべての JVM をリストするか、指定された状態のすべての JVM を調べることができます。

これらのコマンドは、以下の内容を示します。

- プール内の JVM の JVM プロファイルと実行キー。
- プール内のどの JVM が共用クラス・キャッシュを使用するか。
- 各 JVM の経過時間。
- JVM が割り振られているタスク、および JVM がタスクに割り振られた時間。
- **CEMT SET JVMPOOL PHASEOUT**、**PURGE** または **FORCEPURGE** コマンド、もしくは **CEMT PERFORM CLASSCACHE PHASEOUT**、**PURGE** または **FORCEPURGE** コマンド (または同等の **EXEC CICS** コマンド) の結果として段階的に停止される JVM。

また、CICS 統計を使用して、JVM プール内のアクティビティーをモニターすることもできます。これらの統計を収集するには、適切なオプションを指定して **EXEC CICS COLLECT STATISTICS** コマンドまたは **CEMT PERFORM STATISTICS** コマンドを使用してください。JVM プール統計 (JVMPOOL オプション)、TCB モード統計 (DISPATCHER オプション)、JVM プロファイル統計 (JVMPROFILE オプション)、および JVM プログラム統計 (JVMPROGRAM オプション) が役に立ちます。これらの統計は、特に次の内容を示すことができます。

- JVM プール内にある、特定 TCB モードで特定プロファイルの JVM の数 (JVM プロファイル統計から)。
- 特定 TCB モードで特定プロファイルの JVM に対する要求の数 (JVM プロファイル統計から)。
- 要求と一致する実行キーとプロファイルを持つ JVM が使用可能でなかったために、JVM に対する要求が待機しなければならなかった回数 (JVM プールの TCB プール統計から)。これには、最終的に適切な JVM が割り当てられた要求と、これ以上要求を待機させるのではなく、ミスマッチ JVM またはスチール JVM を割り当てることを CICS が決定した要求の両方が含まれます。また、この数字には直列化の待機、つまり、必要なロックを取得するための待機に費やされた時間も含まれる場合があります。
- これらの要求が待機に費やした時間 (JVM プールの TCB プール統計から)。
- JVM に対する要求に、正しくないプロファイルまたは正しくない実行キーを持つ JVM が割り当てられた回数 (JVM プロファイル統計から)。ミスマッチとスチールの発生は、JVM プロファイル別に分類されるので、特定のプロファイルが過剰なスチール・アクティビティーの原因となるかどうかを確認できます。

プールされた JVM のプロファイル使用量のモニター

ブラウズ・モードで **EXEC CICS INQUIRE JVMPROFILE** コマンドを使用すると、CICS 領域の開始以降に、どの JVM プロファイルがプールされた JVM に使用されたかを調べることができます。また、JVM プロファイルの CICS 統計を収集することもできます。

INQUIRE JVMPROFILE は、CICS 領域の存続期間中に使用された JVM プロファイルを検出します。このコマンドは、PROGRAM リソースで使用されるとおりの各 JVM プロファイル名、およびその JVM プロファイルの z/OS UNIX ファイルの絶対パス名を戻します。また、このコマンドは、そのプロファイルを持つ JVM が共用クラス・キャッシュを使用するかどうかも示します。

EXEC CICS COLLECT STATISTICS コマンドまたは **CEMT PERFORM STATISTICS** コマンドを使用して、JVM プロファイルの統計を収集することができます。どちらのコマンドにも、JVMPROFILE オプションを指定してください。この統計は JVM プロファイルと実行キー別に分類され、特に以下のものを示します。

- このプロファイルの JVM に対してアプリケーションが行った要求の数
- JVM プール内にあった、このプロファイルの JVM の総数、現在の数およびピーク数
- CICS でストレージが不足したために破棄された、このプロファイルのプールされた JVM の数。
- このプロファイルの JVM による、およびこのプロファイルの JVM からの TCB スチールの発生率
- このプロファイルの JVM で使用された Language Environment ヒープ・ストレージと JVM ヒープ・ストレージ

「パフォーマンス・ガイド」の『JVM サーバーおよびプールされた JVM の統計』には、JVM 統計に関する詳細情報があり、これらの統計の完全なリストとレポートを検出する方法が記載されています。

プールされた JVM 内のプログラムのモニター

EXEC CICS COLLECT STATISTICS コマンドまたは **CEMT PERFORM STATISTICS** コマンドを使用すると、プールされた JVM で実行される Java プログラムに関する統計を収集することができます。どちらのコマンドにも、JVMPROGRAM オプションを指定してください。

COLLECT コマンドまたは **PERFORM STATISTICS PROGRAM** コマンドが発行される場合、CICS はこれらのプログラムの統計を収集しません。これは、JVM プログラムが CICS によってロードされないからです。

プログラムごとに、統計は次の詳細情報を示します。

- PROGRAM リソースの JVMPROFILE 属性で指定されている、プログラムに必要な JVM プロファイル
- PROGRAM リソースの EXECKEY 属性で指定されている、プログラムに必要な実行キー (CICS キーまたはユーザー・キー)
- PROGRAM リソースの JVMCLASS 属性で指定されている、プログラムにおけるメイン・クラス

- プログラムが使用された回数

JVM 統計の詳細については、「パフォーマンス・ガイド」の『JVM サーバーおよびプールされた JVM の統計』を参照してください。

DFHJVMAT を使用して JVM プロファイルのオプションを変更する

DFHJVMAT は、単独使用のプールされた JVM に対して JVM プロファイルで指定されているオプションのオーバーライドに使用できるユーザー置換可能プログラムです。通常、JVM プロファイルには、JVM を必要に応じて構成するだけの十分な柔軟性があります。JVM プロファイルでオプションを指定しても実現できない方法で JVM を調整する必要がある場合にのみ、DFHJVMAT を使用してください。

DFHJVMAT を使用して、CICS PROGRAM リソースで JVMCLASS 属性をオーバーライドすることもできます。この属性は、JVM で実行する Java プログラムのメイン・クラスを指定します。 PROGRAM リソースを使用する場合、JVMCLASS 属性の限度は 255 文字ですが、DFHJVMAT を使用すると 255 文字より長いクラス名を指定することができます。

DFHJVMAT を呼び出すには、オーバーライドする JVM プロファイルで INVOKE_DFHJVMAT=YES をオプションとして指定します。

重要

DFHJVMAT を呼び出すことができるのは、単独使用のプールされた JVM、つまりオプション REUSE=NO を指定する JVM プロファイルを持つ JVM の場合のみです。単独使用 JVM を使用する場合、その JVM を使用するタスクが終了しても、CICS がその JVM を別のタスクに再使用可能にすることはありません。

継続のプールされた JVM または JVM サーバーに対して DFHJVMAT を呼び出すことはできません。どちらかのタイプの JVM に INVOKE_DFHJVMAT=YES を指定しても、INVOKE_DFHJVMAT=YES は無視され、DFHJVMAT が呼び出されることはありません。

JVM プロファイルで指定された値は、z/OS UNIX システム・サービスの環境変数として DFHJVMAT で使用できます。この値は、JVM を作成する前に変更することができます。

注: STDERR および STDOUT パラメーターの値は、タスク固有の名前を生成するために CICS が解釈できるものであり、解釈する前に DFHJVMAT に渡されます。

DFHJVMAT は C/C++ の `getenv` および `setenv` 関数を使用して、JVM プロファイル内のオプションに対応する環境変数を変更します。例えば、以下のコマンドを使用して、CLASSPATH_SUFFIX 環境変数を指定値に置換することができます。

```
setenv(ecp_suffix, cp_suffix_val,1)
```

ここで、

```
char *ecp_suffix = "CLASSPATH_SUFFIX";
char *cp_suffix_val ="/u/jtest1/Java/test:.";
```

setenv 関数は、CICS PROGRAM リソースには影響せず、JVM の存続期間中のみ有効です。

CICS 提供の DFHJVMAT は、

- 各変数ごとに getenv 要求を発行します。
- printf を宛先 stdout に発行して、各変数の設定を記録します。
- **stdout** および **stderr** に提供される名前に対して setenv コマンドを使用して、CICS タスクごとに固有の出力ファイルおよびエラー・ファイルを作成する方法を示すサンプル・コードを (コメント内に) 含んでいます。

独自のプログラムを作成して、提供されるバージョンに基づいて JVM プロファイルのオプションを調整する場合、その名前は DFHJVMAT でなければならず、プログラムは C で作成される必要があります。DFHJVMAT で EXEC CICS コマンドを使用できますが、これらのコマンドは処理時間を増やす可能性があります。このモジュールの複数の呼び出しは並行して実行される可能性があるため、DFHJVMAT はスレッド・セーフ標準に合わせて作成され、その PROGRAM リソース内に CONCURRENCY(THREADSAFE) と共に定義されなければなりません。

DFHJVMAT から使用可能な JVM プロファイル・オプション

このトピックでリストされている JVM プロファイル・オプションは、DFHJVMAT から使用可能です。これらのオプションは、指定された JVM プロファイルから読み取られ、Language Environment サービスを使用して環境変数として作成されます。

ほとんどの場合、これらのオプションの詳細な説明は 111 ページの『JVM プロファイル: オプションおよびサンプル』にあります。DFHJVMAT を使用してこれらのオプションのいずれかを変更する場合は、事前にそのオプションの詳細な説明をお読みください。

一部のオプションは、CICS 資料に記載されなくなりました。これらのオプションに関する情報は、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション 用の資料やその他の Java 資料で見つけることができます。

注:

1. 通知専用として明示的に指定される場合を除いて、これらの変数の値を再設定することができます。
2. すべての環境変数および値は、大/小文字の区別があり、示されているとおりに設定されなければなりません。
3. CICS は、有効なオプションとして認識しないすべての値を無視します。
4. X で始まるオプションの場合は、以下のとおりです。
   - これらのオプションの一部は、CICS 資料に記載されなくなりました。ただし、引き続き有効なオプションであり、DFHJVMAT から使用可能です。
   - これらのオプションは、JVM プロファイルで指定される場合、先頭が -X でなければなりません。ただし、DFHJVMAT で使用される環境変数にはハイフンが含まれないので、引き続き先頭は X です。

表 11. *DFHJVMAT から使用可能な JVM プロファイル・オプション*

オプション	指定内容
CLASSPATH_PREFIX、CLASSPATH_SUFFIX	標準クラスパスの接頭部と接尾部
INVOKE_DFHJVMAT	通知専用
JAVA_DUMP_OPTS	JVM での異常終了の診断を取得するための、Java ダンプ・オプションのセット
JAVA_HOME	IBM 64-bit SDK for z/OS, Java テクノロジー・エディション サブディレクトリーおよび JAR ファイルへのパス
JVMPROPS	JVM プロパティー・ファイルのパスと名前
LIBPATH_PREFIX、LIBPATH_SUFFIX	ライブラリー・パスの接頭部と接尾部
STDERR	JVM からの **stderr** 出力用の z/OS UNIX ファイルの名前
STDIN	**stdin** 用の z/OS UNIX ファイルの名前
STDOUT	JVM からの **stdout** 出力用の z/OS UNIX ファイルの名前
USEROUTPUTCLASS	JVM からの **stdout** および **stderr** 出力を代行受信し、リダイレクトする Java クラスの名前
VERBOSE	JVM からの情報メッセージのレベル
WORK_DIR	z/OS UNIX 上の CICS 領域用の作業ディレクトリー
`Xcheck`	追加の検査を実行します
`Xdebug`	デバッグ・サポートを使用可能にします
`Xmaxe`、`Xmaxf`、`Xmine`、`Xminf`	ヒープの最大および最小のヒープ拡張サイズ、およびフリー・ヒープ・パーセンテージ・サイズ
`Xms`	ヒープの初期サイズ
`Xmx`	ヒープの最大サイズ
`Xnoagent`	以前の `sun.tools.debug` エージェントを使用不可にします (`Xdebug` が指定されている場合)
`Xnoclassgc`	クラスのガーベッジ・コレクションを使用不可にします
`Xoss`	任意のスレッドの最大 Java スタック・サイズ
`Xrs`	JVM によるオペレーティング・システム・シグナルの使用を減らします
`Xrun`*`dllname`*	指定されたダイナミック・リンク・ライブラリー (DLL) をロードし、指定されたオプションをその DLL に渡します
`Xss`	新しい Java スレッドごとのスタックのサイズ
`Xverify`	ロードされたクラスで実行される検証のレベル

次のように、標準の JVM プロファイルにない 2 つの追加フィールドが DFHJVMAT に渡されます。

CICS_PROGRAM
実行される、Java クラスに関連した CICS プログラムの名前 (1 文字から 8 文

字) を指定します。この名前は実行時に設定されます。通知のためのみに DFHJVMAT に渡され、変更できません。変更はすべて、CICS で無視されます。

CICS_PROGRAM_CLASS
CICS ユーザー・アプリケーション・クラス名を指定し、プログラム・リソース定義から取得されます。これは、CICS PROGRAM リソース定義の JVMCLASS 属性で定義されます。 DFHJVMAT を使用したこの属性のオーバーライドに代わる方法として、JVM に制御が渡される前に、**SET PROGRAM** コマンドを使用して PROGRAM リソースの JVMCLASS 属性を変更することができます。PROGRAM リソースを使用する場合、JVMCLASS 属性の限度は 255 文字ですが、DFHJVMAT を使用すると 255 文字より長いクラス名を指定することができます。

第 7 章 Java パフォーマンスの改善

Java アプリケーションと、Java アプリケーションが実行される JVM の両方のパフォーマンスを改善する方法がいくつかあります。

このタスクについて

どんなに CICS の調整が適切であっても、アプリケーションが効率よく作成されていないと、適切に作成されたアプリケーションと比べれば常にパフォーマンスが劣ります。例えば、生成するガーベッジを減らすようにアプリケーションを変更すれば、ガーベッジ・コレクションのコストを大きく節約することができます。生成されるガーベッジが少なくなればなるほど、ガーベッジ・コレクションに要する時間も短くなります。パフォーマンスを改善するために、Java 環境を調整するだけでなく、ご使用の Java アプリケーションが効率よく作成されていることを必ず確認してください。

手順

1. Java ワークロードのパフォーマンス目標を決定します。 最も一般的な目標には、プロセッサー使用率やアプリケーション応答時間の最小化があります。目標を決定した後、それに応じて Java 環境を調整することができます。
2. Java アプリケーションを分析して、その Java アプリケーションが効率よく動作し、過剰なガーベッジを生成していないことを確認します。 IBM が提供するツールは、Java アプリケーションを分析して、特定の方法およびアプリケーション全体の効率とパフォーマンスを改善するのに役立ちます。
3. JVM サーバーまたはプールされた JVM を調整します。 統計と IBM ツールを使用すれば、ストレージの設定、ガーベッジ・コレクション、タスクの待機などの情報を分析して、JVM のパフォーマンスを調整することができます。
4. JVM が実行される Language Environment エンクレーブを調整します。 JVM が使用する MVS ストレージは、MVS Language Environment サービスの呼び出しによって取得されます。Language Environment のランタイム・オプションを変更して、MVS によって割り振られるストレージを調整することができます。
5. オプション: z/OS 共用ライブラリー領域を使用して、異なる CICS 領域内の JVM 間で DLL を共用する場合、ストレージの設定を調整できます。

Java ワークロードにおけるパフォーマンス・ゴールの判別

所定のアプリケーション・ワークロードの全体的なパフォーマンスが最良のものになるように CICS JVM を調整するには、いくつかの異なる要因が関係しています。Java ワークロードに必要なパフォーマンス特性を決定する必要があります。これらの特性を設定すると、変更が必要なパラメーターとその変更方法を判別することができます。

Java ワークロードの最も一般的なパフォーマンス目標は、以下のとおりです。

最小限の総プロセッサー使用率
この目標は、使用可能なプロセッサー・リソースを最も効率的に使用するこ

とに重点を置いています。この目標を達成するようにワークロードを調整すると、ワークロード全体のプロセッサーの合計使用率は最小化されますが、個々のタスクはプロセッサーを著しく消費する可能性があります。総プロセッサー使用率が最小限になるよう調整するには、JVM に大きいストレージ・ヒープ・サイズを指定して、ガーベッジ・コレクション数を最小化することが必要です。

アプリケーションの最小応答時間
この目標は、アプリケーション・タスクをできるだけ素早く呼び出し元に戻すことに重点を置いています。達成しなければならないサービス・レベル・アグリーメントが存在する場合、この目標は特に重要である可能性があります。この目標を達成するようにワークロードを調整すると、アプリケーションは一貫して素早く応答します。ただし、ガーベッジ・コレクションのためにプロセッサー使用率が高くなることがあります。アプリケーションの最小応答時間のための調整には、ヒープ・サイズを小さくしておくことと、おそらく gencon ガーベッジ・コレクション・ポリシーを使用することが必要です。

最小限の JVM ストレージ・ヒープ・サイズ
この目標は、JVM によって使用されるストレージの量を削減することに重点を置いています。 JVM は 64 ビット・ストレージを使用するので、1 つの CICS 領域で多数のプールされた JVM と JVM サーバーを実行することが可能です。プールされた JVM が使用するストレージ・ヒープが小さくなれば、CICS 領域でより多くの JVM を実行することが可能です。ただし、この目標を選択するとプロセッサー・コストが増大する可能性があります。ストレージ・ヒープ・サイズを最小化するように JVM を調整すると、ガーベッジ・コレクション・イベントの頻度は非常に高くなります。

その他の要因が、アプリケーションの応答時間に影響を与える場合があります。その中でも最も重要なのが、Just In Time (JIT) コンパイラーです。 JIT コンパイラーは、実行時にアプリケーション・コードを動的に最適化し、多くの利点を提供しますが、これを行うには一定量のプロセッサー・リソースが必要になります。

IBM Health Center を使用した Java アプリケーションの分析

Java アプリケーションのパフォーマンスを改善するには、IBM Health Center を使用してそのアプリケーションを分析することができます。このツールは、アプリケーションのパフォーマンスと効率の改善に役立つ推奨事項を提供します。

このタスクについて

IBM Health Center は、IBM Support Assistant Workbench で使用できます。これらの無料のツールは、Getting Started guide に記載されているとおりに IBM からダウンロードできます。JVM でアプリケーションを単独で実行してみてください。JVM サーバーで混合ワークロードを実行している場合は、特定アプリケーションの分析がさらに難しくなることがあります。

手順

1. 必要な接続オプションを JVM サーバーの JVM プロファイルに追加します。 IBM Health Center の資料に、ツールから JVM に接続するために追加する必要があるオプションが記述されています。
2. IBM Health Center を開始し、実行中の JVM に接続します。 IBM Health Center は JVM のアクティビティーをリアルタイムで報告するので、IBM Health Center が JVM をモニターするまでしばらく待ってください。
3. 「**Profiling**」リンクを選択して、アプリケーションのプロファイルを作成します。 さまざまなメソッドで費やす時間を確認できます。使用率が最大のメソッドを確認して、潜在的な問題を探してください。

   ヒント: 「**Analysis and Recommendations**」タブでは、最適化の候補になりうる特定のメソッドを識別できます。
4. 「**Locking**」リンクを選択して、アプリケーションにロッキングの競合がないか確認します。 Java ワークロードが、使用可能なすべてのプロセッサーを使用できるとは限らない場合、ロッキングが原因である可能性があります。アプリケーションでのロッキングにより、実行できる並列スレッドの量が減る可能性があります。
5. 「**Garbage Collection**」リンクを選択して、ヒープ使用量とガーベッジ・コレクションを確認します。 「**Garbage Collection**」タブでは、ヒープの使用量、およびガーベッジ・コレクションを実行するために JVM が一時停止する頻度が表示されます。
   a. ガーベッジ・コレクションで費やされた時間の割合を確認します。 この情報は「Summary」セクションに表示されます。ガーベッジ・コレクションで費やされた時間が 2% を超える場合、ガーベッジ・コレクションの調整が必要な可能性があります。
   b. ガーベッジ・コレクションの一時停止時間を確認します。 一時停止時間が 10 ミリ秒を超える場合、ガーベッジ・コレクションがアプリケーションの応答時間に影響を与えている可能性があります。
   c. ガーベッジ・コレクションの比率をトランザクション数で除算して、各トランザクションが生成するおおよそのガーベッジ量を確認します。 ガーベッジの量がアプリケーションにとって多いと思われる場合は、さらにアプリケーションを調査する必要がある可能性があります。

次のタスク

アプリケーションを分析した後、Java ワークロードに合わせて Java 環境を調整することができます。

ガーベッジ・コレクションおよびヒープ拡張

ガーベッジ・コレクションおよびヒープ拡張は、JVM の操作において重要な部分を占めています。JVM におけるガーベッジ・コレクションの頻度は、JVM で実行するアプリケーションによって作成される不要情報、つまりオブジェクトの量に影響されます。

割り振り失敗

JVM がストレージ・ヒープでスペース不足になり、それ以上のオブジェクトを割り振ることができない (割り振り失敗) 場合には、ガーベッジ・コレクションがトリガーされます。ガーベッジ・コレクターは、アプリケーションによって参照されなくなったストレージ・ヒープのオブジェクトをクリーンアップし、スペースの一部を解放します。ガーベッジ・コレクションは、ガーベッジ・コレクション・サイクルの期間に JVM で実行中の他のすべての処理を停止するため、ガーベッジ・コレクションに費やされる時間は、アプリケーションの実行には使用されていない時間ということになります。「Java Diagnostics Guide」には、JVM のガーベッジ・コレクション処理の詳しい説明があります。

割り振り失敗によってガーベッジ・コレクションが起動しても、十分なスペースが解放されなかった場合、ガーベッジ・コレクターはストレージ・ヒープを拡張します。ガーベッジ・コレクターは、ヒープ拡張時に、ヒープ用に予約されるストレージの最大量 (`-Xmx` オプションで指定された量) からストレージを取り、それをヒープのアクティブな部分 (`-Xms` オプションで指定されたサイズで始まる) に追加します。開始時に、`-Xmx` オプションで指定されたストレージの最大量が既に JVM に割り振られているため、ヒープ拡張によって、JVM に必要なストレージの量が増えることはありません。 `-Xms` オプションの値が、アプリケーションのヒープのアクティブな部分に十分なストレージを提供する場合、ガーベッジ・コレクターがヒープ拡張を実行する必要はありません。

JVM の存続時間中のある時点で、ガーベッジ・コレクターはストレージ・ヒープの拡張を停止します。これは、ヒープが、ガーベッジ・コレクションの頻度および処理によって解放されるスペースの量に関して、ガーベッジ・コレクターが適切であると判断する状態に達したためです。ガーベッジ・コレクターは割り振り失敗を除去するわけではありません。このため、ガーベッジ・コレクターがストレージ・ヒープの拡張を停止した後も、割り振り失敗により一部のガーベッジ・コレクションを起動することができます。パフォーマンス目標に応じて、ガーベッジ・コレクションの頻度が高すぎないかどうか考慮することができます。

ガーベッジ・コレクションのオプション

ガーベッジ・コレクションにはさまざまなポリシーを使用でき、これらのポリシーは、アプリケーションとシステム全体のスループットと、ガーベッジ・コレクションが原因の一時停止時間とのトレードオフを行います。ガーベッジ・コレクションは `-Xgcpolicy` オプションによって制御されます。

`-Xgcpolicy:optthruput`
このポリシーは、高いスループットをアプリケーションで実現しますが、その代わりに、ガーベッジ・コレクションの処理時に一時停止が生じることがあります。

`-Xgcpolicy:gencon`
このポリシーは、ガーベッジ・コレクションの一時停止で費やされる時間を最小限に抑えるのに役立ちます。このガーベッジ・コレクション・ポリシーは JVM サーバーで使用してください。 JVMSERVER リソースを調べると、JVM サーバーで使用されているポリシーを確認することができます。JVM サーバー統計にあるフィールドは、ガーベッジ・コレクションのメジ

ャー・イベントとマイナー・イベントの数、およびガーベッジ・コレクションで費やされたプロセッサー時間を示します。

JVM プロファイルを更新することによって、ガーベッジ・コレクション・ポリシーを変更できます。すべてのガーベッジ・コレクション・オプションについて詳しくは、Specifying garbage collection policy を参照してください。

例 1: ガーベッジの生成量が少ないアプリケーション

図 7 は、optthruput ガーベッジ・コレクション・ポリシーのさまざまな段階で、プールされた JVM におけるストレージ・ヒープを示しています。ストレージ・ヒープ用に予約されるストレージの最大量は -Xmx オプションによって決定されます。-Xms オプションによって決定されるストレージ・ヒープのアクティブな部分は陰影で表示されます。

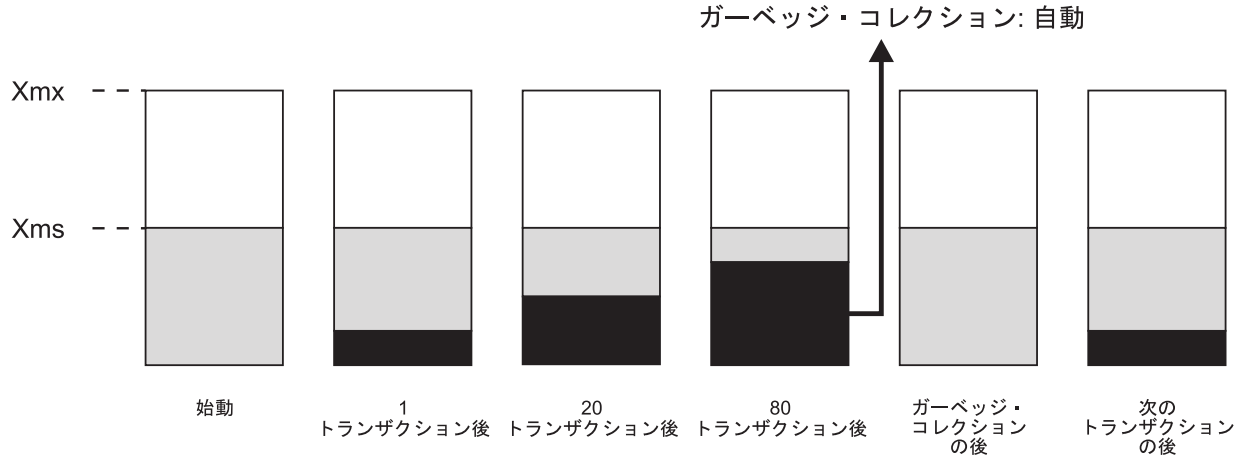


図 7. ガーベッジの量が少ない場合の、プールされた JVM のストレージ・ヒープ

開始時に、-Xms は -Xmx の半分の値に設定されています (提供されている JVM プロファイルのデフォルト設定と同じ)。ヒープ使用率の限度 (GC_HEAP_THRESHOLD オプション) はデフォルトで 85% に設定されています。

JVM で実行する最初のアプリケーションは、ヒープのアクティブな部分にある少量のストレージを使用します。使用されるストレージは、黒で示されます。トランザクションが完了すると、アプリケーションによって使用されるオブジェクトは参照されなくなるため、ガーベッジ・コレクションの対象になります。ガーベッジ・コレクションが発生するまで、それらのオブジェクトはストレージ・ヒープに残ります。

JVM を使用するトランザクションが 20 回行われた後、ヒープのアクティブな部分を占めるストレージの量は増加しています。各トランザクションは少量のストレージを使用しましたが、ガーベッジ・コレクションはまだ行われていません。

トランザクションが 80 回行われた後、ストレージの 85% がヒープのアクティブな部分を占めたため、ヒープ使用率の限度である 85% に達しました。限度に達したトランザクションの直後に、CICS はガーベッジ・コレクションを開始します。ガ

ーベッジ・コレクションが行われた後、最初の 80 のトランザクションによって使用されたすべてのオブジェクトはガーベッジ・コレクションされたため、ストレージ・ヒープのアクティブな部分は現時点で空になっています。JVM で実行する次のアプリケーションは、再び少量のストレージを使用し、こうしてサイクルが再開します。

この例では、割り振り失敗およびヒープ拡張は発生しません。これは、-Xms オプションの値が、アプリケーション用のヒープのアクティブな部分に十分なストレージが存在するように設定されているためです。ヒープ使用率の限度に達したときに行われるのは、CICS によって要求されるガーベッジ・コレクションのみです。ただし、このワークロードが一定の場合は、-Xmx オプションの値は必要以上に高くなります。ストレージ・ヒープ用に予約されるすべてのストレージは割り振られましたが、その半分は使用されておらず、無駄になっています。

例 2: ガーベッジの生成量が多いマルチスレッド・アプリケーション

図 8 は、gencon ガーベッジ・コレクション・ポリシーを使用する場合の、さまざまな段階での JVM サーバー内のストレージ・ヒープを示しています。プールされた JVM とは異なり、JVM サーバーは 1 つのアプリケーションに対する要求を複数同時に実行できます。このため、アプリケーションが生成するガーベッジの量が多くなる可能性があります。JVM サーバー内では、ガーベッジ・コレクションは JVM によって自動的に処理されます。

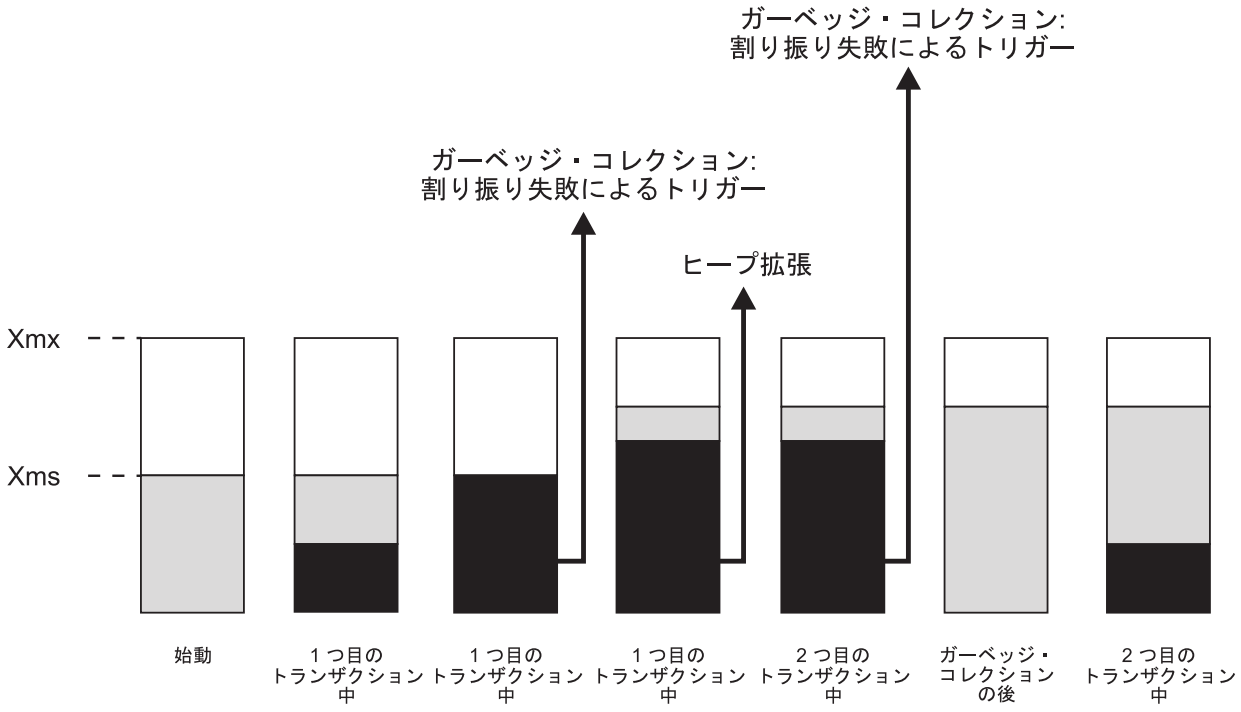


図 8. ガーベッジの量が多い場合の JVM サーバーのストレージ・ヒープ

最初の 20 件のトランザクションが行われる過程で、ストレージ・ヒープのアクティブな部分が満杯になり始めます。トランザクション数が 80 に達した後、ヒープ

は満杯になり、割り振り失敗が発生します。これにより、JVM 内で小規模なガーベッジ・コレクションが起動されます。ガーベッジ・コレクションにより、存続時間の短いオブジェクトはクリーンアップされます。ただし、アプリケーション要求がまだ実行中であるため、オブジェクトの一部は引き続き参照されており、ガーベッジ・コレクションに適格ではありません。

トランザクション数が 100 に達した後、ガーベッジ・コレクターは、現在必要なすべてのオブジェクトのための十分なスペースを見つけられないため、ストレージ・ヒープを拡張します。ストレージ・ヒープ用に予約されるストレージの最大量 (`-Xmx` オプションで指定された量) から、一部のストレージがヒープのアクティブな部分に追加されます。アプリケーションはオブジェクトの生成を継続しますが、ヒープ拡張によって十分なスペースが作成されたため、現行トランザクションを完了することができます。

トランザクション数が 110 に達した後、ストレージ・ヒープはほとんど占有された状態になります。再び割り振り失敗が発生したため、大規模なガーベッジ・コレクションが起動されます。前のトランザクションによって使用されていた、存続時間の長いオブジェクトの多くが JVM によってクリーンアップされます。さらに 20 件のトランザクションが行われた後、ヒープは再び満杯になり始めます。

`gencon` ポリシーを使用すると、大規模なガーベッジ・コレクションが実行される前に、ヒープ・サイズを管理するために多数の小規模なガーベッジ・コレクションが行われることがあります。JVM サーバー統計を使用して、ガーベッジ・コレクションが行われた回数、ヒープの占有率などの情報を調べることができます。

JVM サーバーのパフォーマンスの改善

JVM サーバーで実行されているアプリケーションのパフォーマンスを改善するには、ガーベッジ・コレクションやヒープ・サイズを始めとする、環境のさまざまな部分を調整することができます。

このタスクについて

CICS が提供する JVM サーバーに関する統計レポートには、タスクでスレッドを待機する時間、ヒープ・サイズ、ガーベッジ・コレクションの頻度、およびプロセッサー使用率の詳細が記載されています。また、JVM のモニターと分析を直接行って JVM サーバーを調整し、問題診断に役立つ、追加の IBM ツールを使用することもできます。統計を使用すると、JVM が効率よく動作していること、特にヒープ・サイズが適切で、ガーベッジ・コレクションが最適化されていることを確認することができます。

手順

1. JVM サーバーで使用されているプロセッサー時間の量を確認します。 ディスパッチャー統計は、T8 TCB が使用しているプロセッサー時間の量を示すことができます。JVM サーバー統計は、JVM がガーベッジ・コレクションに費やす時間、およびガーベッジ・コレクションの回数を示します。JVM ガーベッジ・コレクションは、アプリケーションの応答時間とプロセッサー使用率に悪影響を与える可能性があります。

2. JVM サーバーが必要とするヒープ・サイズの調整に使用できる十分なストレージがあることを確実にします。
3. JVM におけるガーベッジ・コレクションとヒープを調整します。 ヒープが小さいと、ガーベッジ・コレクションの頻度が非常に高くなる可能性があります。一方、ヒープが大きすぎると、MVS ストレージの使用効率が悪くなるおそれがあります。 IBM Health Center を使用すると、ガーベッジ・コレクションの視覚化と調整を行い、それに応じてヒープを調整することができます。

次のタスク

メモリー使用量とヒープ・サイズをより詳しく分析するには、IBM Support Assistant の Memory Analyzer ツールを使用すると、Java プロセスのシステム・ダンプまたはヒープ・ダンプのスナップショットを使用して Java ヒープ・メモリーを分析することができます。

JVM サーバーによるプロセッサー使用量の確認

CICS モニター機能を使用すると、JVM サーバーで実行中のトランザクションで使用されるプロセッサー時間をモニターすることができます。 JVM サーバー内のすべてのスレッドは T8 TCB で実行されます。

このタスクについて

DFH$MOLS ユーティリティーを使用すると、SMF レコードを印刷したり、CICS Performance Analyzer などのツールを使用して SMF レコードを分析したりすることができます。

手順

1. CICS 領域でモニターをオンにして、パフォーマンス・クラスのモニター・データを収集します。
2. パフォーマンス・データ・グループ DFHTASK を確認します。 特に、次のフィールドを調べることができます。

フィールド ID	フィールド名	説明
283	MAXTTDLY	CICS 領域が使用可能なスレッドの限界に達したため、ユーザー・タスクが T8 TCB を取得するために待っている間に経過した時間。スレッドの限度は CICS 領域ごとに 1024 で、各 JVM サーバーは最大で 256 のスレッドを持つことができます。
400	T8CPUT	ユーザー・タスクが、CICS T8 モードの TCB 上の CICS ディスパッチャー・ドメインによってディスパッチされている間のプロセッサー時間。T8 TCB に 1 つのスレッドが割り振られると、処理が完了するまでそのスレッドに対して同じ TCB が関連付けられた状態が続きます。
401	JVMTHDWT	CICS システムが CICS 領域内の JVM サーバーに関するスレッドの限界に達したため、ユーザー・タスクが JVM サーバー・スレッドを取得するために待っている間に経過した時間。

3. プロセッサー使用量を改善するために、可能ならば、トレースの使用を削減または除去します。
   a. 実稼働環境では、CICS マスター・システムのトレース・フラグをオフに設定して、CICS 領域を稼働させることを検討してください。 このフラグをオンに設定すると、Java プログラムの実行によってプロセッサーの消費が著しく増加します。 SYSTR=OFF で CICS を初期化するか、CETR トランザクションを使用することによって、フラグをオフに設定することができます。
   b. 特殊なトランザクションのみの JVM トレースをアクティブにしていることを確認します。 JVM トレースは、短時間で大量の出力を生成することがあり、プロセッサー・コストを増加させます。JVM トレースの制御について詳しくは、210 ページの『Java の診断』を参照してください。
4. 実稼働環境で JVM プロファイルでの USEROUTPUTCLASS オプションを使用しないでください。 このオプションを指定すると、JVM のパフォーマンスに悪影響が出ます。 USEROUTPUTCLASS オプションによって、同一の CICS 領域を使用する開発者は、JVM 出力を分離し、適切な宛先に送信することができます。ただし、このためには、追加クラス・インスタンスの作成および呼び出しが必要です。

JVM サーバーのストレージ所要量の計算

CICS 領域内の JVM サーバー数を増やすには、十分なストレージが CICS に使用可能であることを確実にする必要があります。

このタスクについて

JVM は、16 MB 境界より下のストレージ、31 ビット・ストレージ、および 64 ビット・ストレージを使用します。CICS 領域で実行される JVM 数に関係なく、JVM サーバーの実行には一回限りのストレージ・コストが発生します。各 JVM サーバーとその Language Environment エンクレーブにも、一定量の 31 ビットおよび 64 ビット・ストレージが必要です。JVM ヒープ・サイズは JVM によって管理され、CICS はデフォルト値を使用します。環境を調整する一環として必要な場合は、ヒープ・サイズを調整できます。

JVM ヒープに必要なストレージは、CICS 領域ストレージ (EDSA ストレージではなく MVS ストレージ) から作成されます。JVM ヒープが大きければ、CICS 領域に存在する JVM の数は減り、それらの JVM をサポートするための領域サイズは増加します。ただし、ヒープ・サイズの設定が小さすぎると、ガーベッジ・コレクションが必要以上に行われるため、パフォーマンスに影響します。 Java ワークロードで最良のパフォーマンスを得るために、JVM ストレージ・オプションを調整できます。JVM ストレージ・オプションは、Java アプリケーションのプロセッサー使用率、ストレージ使用量、およびタスクの応答時間の判別に役立ちます。

手順

1. サンプル統計プログラム DFH0STAT を使用して、2 GB 境界より下で使用可能なフリー・ストレージの量を判別します。ストレージ・レポートには、31 ビット・ストレージで 16 MB 境界より下で割り振られるユーザー・ストレージの量が記載されています。
   - 実行中の JVM サーバーがない場合、CICS アドレス・スペース内のフリー・ストレージの合計量から、z/OS 共用ライブラリー領域用に予約されているス

トレージを減算します。ストレージは、MVS で **SHRLIBRGNSIZE** パラメーターによって制御され、最初の JVM が領域で開始されるときに一回割り振られます。

- 実行中の JVM サーバーがある場合は、フリー・ストレージの合計量から、**SHRLIBRGNSIZE** パラメーターの値を減算します。実行中の各 JVM は、16 MB 境界より下で 12 KB のストレージを使用します。JVM ごとの Language Environment エンクレーブは、ヒープおよびライブラリー・ヒープに 31 ビット・ストレージを使用します。割り振られる 31 ビット・ストレージの量は、DFHOSGI の HEAP64 オプションと LIBHEAP64 オプションで設定されます。また、現在使用可能なストレージ量を算出するには、フリー・ストレージの合計量から、これらの値を減算することも必要です。

31 ビット・ストレージの設定を変更したい場合は、**SHRLIBRGNSIZE** パラメーターと Language Environment オプションを調整することができます。206 ページの『z/OS 共用ライブラリー領域の調整』および 201 ページの『DFHAXRO を使用した JVM サーバーのエンクレーブの変更』を参照してください。

2. 追加の JVM サーバーごとに必要な 64 ビット・ストレージの量を計算します。JVM サーバーに必要な 64 ビット・ストレージ量を計算するには、次のストレージ所要量を合計することができます。
   - **-Xmx** 値。このパラメーターのデフォルト値は JVM によって設定されるので、Java インフォメーション・センターの資料を確認してください。
   - DFHAXRO の HEAP64 オプションによって割り振られる 64 ビット・ストレージの値。
   - DFHAXRO の LIBHEAP64 オプションによって割り振られる 64 ビット・ストレージの値。
   - DFHAXRO の STACK64 オプションによって割り振られる 64 ビット・ストレージの値。JVM サーバーで使用できるスレッド数をこの値に乗算します。使用できるスレッド数を計算するには、JVMSERVER リソースの THREADLIMIT 属性値を、**-Xgcthreads** パラメーターの値に加算します。この Java オプションは、JVM 内のガーベッジ・コレクション・ヘルパー・スレッド数を制御します。
3. **MEMLIMIT** 値を調べて、追加の JVM サーバーの実行に使用できる十分な 64 ビット・ストレージがあるかどうかを判別します。64 ビット・ストレージを使用する他の CICS 機能を考慮する必要があります。

   z/OS **MEMLIMIT** パラメーターは、CICS 領域の 64 ビット (2 GB 境界より上) ストレージの量を制限します。 64 ビット・ストレージを使用する CICS 機能、およびこのパラメーターの確認と調整の方法については、「パフォーマンス・ガイド」の『MEMLIMIT の見積もり、確認、および設定』 を参照してください。

JVM サーバー・ヒープとガーベッジ・コレクションの調整

JVM サーバーのガーベッジ・コレクションは、JVM によって自動的に処理されます。ガーベッジ・コレクション処理とヒープ・サイズを調整して、アプリケーションの応答時間とプロセッサー使用率が最適であることを確認できます。

このタスクについて

ガーベッジ・コレクション処理は、アプリケーションの応答時間とプロセッサー使用率に影響を与えます。ガーベッジ・コレクションは、JVM におけるすべての作業を一時的に停止するので、アプリケーションの応答時間に影響を与える場合があります。小さいヒープ・サイズを設定する場合、メモリーを節約できますが、ガーベッジ・コレクションの頻度が高くなり、ガーベッジ・コレクションで費やされるプロセッサー時間が増える可能性があります。設定するヒープ・サイズが大きすぎると、JVM は、MVS ストレージの使用効率が悪くなるので、データ・キャッシュ・ミスやページングまでもが生じる可能性があります。CICS が提供する統計を使用すると、JVM サーバーを分析することができます。また、有利なデータ分析と調整オプションの推奨を行う IBM Health Center を使用することもできます。

手順

1. 適切な間隔で JVM サーバー統計とディスパッチャー統計を収集します。 JVM サーバー統計は、メジャー・ガーベッジ・コレクションとマイナー・ガーベッジ・コレクションの数、およびプロセッサーがガーベッジ・コレクションで費やした時間を示すことができます。ディスパッチャー統計は、CICS 領域全体で T8 TCB のプロセッサー使用率に関する情報を示すことができます。
2. T8 TCB のディスパッチャー TCB モード統計を使用して、JVM サーバー・スレッドで費やされたプロセッサー時間を確認します。 「Accum CPU Time / TCB」フィールドは、この TCB モードで接続中であるか、または接続されていたすべての TCB に要したプロセッサー時間の累積を示します。「TCB attaches」フィールドは、統計間隔で使用された T8 TCB の数を示します。これらの数値を使用して、各 T8 TCB が使用したおおよそのプロセッサー時間を算出してください。
3. JVM サーバー統計を使用して、ガーベッジ・コレクションで費やされた時間のパーセンテージを検出します。 統計間隔の時間を、ガーベッジ・コレクションで費やされた経過時間で除算してください。ガーベッジ・コレクションにおけるプロセッサー使用率を 2% 未満にすることを目標にします。パーセンテージがこれより高い場合、ガーベッジ・コレクションの頻度が低くなるようにヒープのサイズを大きくすることができます。
4. 「Current heap size」フィールドを「GC heap occupancy」フィールドと比較して、ヒープで使用されているライブ・データの量を確認します。 ヒープが大きい場合は、ガーベッジ・コレクションの後であっても、ガーベッジ・コレクションを実行するための一時停止時間が長くなります。

   optthruput ポリシーで使用する単一ヒープは、頻度を下げるものの、一時停止時間が長くなる可能性があります。gencon ポリシーでは、ヒープを 2 つの部分に分割するので、JVM は、マイナー・ガーベッジ・コレクションをナーサリーで実行し、メジャー・ガーベッジ・コレクションを全ヒープで実行します。gencon ポリシーは、ガーベッジ・コレクションの一時停止で費やされる時間を最小限に抑えるのに役立ちます。

   アプリケーションの応答時間を改善したい場合は、gencon ポリシーを使用します。

   a. JVMSERVER リソースの統計を調べて、JVM でどのポリシーを使用しているかを確認します。

b. JVM が optthruput を使用している場合は、JVM プロファイルを編集して、**-Xgcpolicy** オプションを追加します。 `-Xgcpolicy:gencon` を指定してください。

c. JVMSERVER リソースをいったん使用不可にしてから、使用可能にして、JVM プロファイルの変更を取得します。 使用可能になった JVMSERVER リソースを調べて、変更内容が適用されていることを確認できます。

5. 解放されたヒープの値を、その間隔内で実行されたトランザクション数で除算して、トランザクションごとに収集されている不要情報の量を確認します。 T8 TCB のディスパッチャー統計を調べると、実行されたトランザクション数を確認できます。JVM サーバー内の各スレッドは 1 つの T8 TCB を使用します。

6. オプション: より詳細な分析を実行するには、JVM プロファイルに **-verbose:gc** オプションを追加します。 JVM は、XML のガーベッジ・コレクション・メッセージを、JVM プロファイルの STDERR オプションで指定されるファイルに書き込みます。これらのメッセージの例と説明については、verbose:gclogging を参照してください。

ヒント: Memory Analyzer ツールのファイルを使用すると、より詳細な分析を実行することができます。

タスクの結果

調整結果は、ユーザーの Java ワークロード、CICS および IBM SDK for z/OS の保守レベル、およびその他の要因によって異なります。ストレージとガーベッジ・コレクションの設定、および JVM の調整の可能性について詳しくは、「Java Diagnostics Guide」を参照してください。

シスプレックス内の JVM サーバー開始の調整

シスプレックス内の複数の CICS 領域全体で複数の JVM サーバーを同時に開始する際に問題がある場合、環境を調整することによってパフォーマンスを改善できます。

このタスクについて

JVM サーバーが開始すると、`/usr/lpp/cicsts/cicsts42/lib` ディレクトリーに 1 組のライブラリーをロードします。複数の JVM サーバーを同時に開始すると、各 JVM が必要なライブラリーをロードするのにしばらく時間がかかる場合があります。一部の JVM サーバーがタイムアウトになるか、開始に長時間かかる可能性があります。開始時間を短縮するために、環境を調整することができます。

手順

1. zFS を読み取り専用モードでマウントして、シスプレックス内のさまざまな JVM サーバーからライブラリーにアクセスするのにかかる時間を改善します。
2. シスプレックス内の別の LPAR に zFS をマウントして、ライブラリーのローカル・コピーを CICS 領域に提供します。
3. JVM サーバー用の共用クラス・キャッシュを作成して、ライブラリーを 1 回ロードします。 共用クラス・キャッシュを使用するには、**-Xshareclasses** オプションを各 JVM サーバーの JVM プロファイルに追加します。このオプションについて詳しくは、Class data sharing between JVMs を参照してください。

4. OSGi フレームワークのタイムアウト値を増やします。 DFHOSGI JVM プロファイルに入っている OSGI_FRAMEWORK_TIMEOUT オプションは、JVM サーバーが開始し、シャットダウンするのを CICS が待機する時間量を指定します。この時間を超えると、JVM サーバーは正しく初期化またはシャットダウンできません。デフォルト値は 60 秒であるので、ご使用の環境にもっと適切な秒数にこの値を増やしてください。

パフォーマンスに関する JVM プールの管理

JVM プールの設定を調整すると、プールされた JVM の使用中にプロセッサー時間が無駄になっていないことを確認することによって、トランザクションの応答時間を短縮できる場合があります。また、Java ワークロードを実行している各 CICS 領域が、その領域サイズに最適な数の JVM を含んでいるので、ストレージおよびプロセッサー時間を最大限活用していることも確認できます。

このタスクについて

1 つの CICS 領域がサポート可能なプールされた JVM の数は、主に以下の要因によって制御されます。

- JVM が使用するプロセッサー時間。
- JVM が必要とする MVS ストレージの量。
- CICS 領域の使用で利用可能な MVS ストレージおよびプロセッサー時間の量。

必要なレベルのトランザクション・スループットをサポートするために必要な、プールされた JVM の数を見積もるには、以下の数式を使用します。

```
ETR x 応答時間 = JVM の数
```

ここで、

- ETR は、望まれるトランザクション・スループットのレベルです。
- 応答時間は、JVM でのトランザクションの実行にかかる時間です。

以下の手順は、JVM プールを調整する推奨プロセスです。

手順

1. JVM を獲得するためにトランザクションが待機している時間を検出します。 CICS ディスパッチャー TCB プール統計の統計フィールド「Total Max TCB Pool Limit delay time」を確認してください。このフィールドは、JVM プールが **MAXJVMTCBS** 制限に達したときに、トランザクションが JVM を獲得するために待機した時間の長さを示しています。 また、パフォーマンス・データ・グループ DFHTASK の CICS モニター・データ・フィールド MAXJTDLY (フィールド ID 277) を使用して、個々のトランザクションが JVM を獲得するために待機した時間量を確認することもできます。
   a. 遅延時間が短いと思われる場合には、JVM プールの **MAXJVMTCBS** 制限に達したのが頻繁でない可能性があります。 CICS ディスパッチャー TCB プール統計の「Times at Max TCB Pool Limit」統計フィールドは、その制限に達した回数を示します。この状態では、トランザクションの遅延時間を大幅に増加させることなく、**MAXJVMTCBS** 制限を下げることが可能な場合があります。
   b. 遅延時間が長いと思われる場合には、CICS ディスパッチャー TCB プール統計の「Total Attaches delayed by Max TCB Pool Limit」統計フィールドで除

算して、それぞれのトランザクションがどのくらいの長さ待機するかを確認します。要約 TCB プール統計の「Average Max TCB Pool Limit delay time」フィールドにこの情報があります。 JVM プールが通常は **MAXJVMTCBS** 限界にある場合、トランザクションは JVM を獲得するために、少なくとも短時間待機することがよくあります。各トランザクションの遅延時間が過度に長いと思われる場合のみ、**MAXJVMTCBS** 制限を引き上げてください。

2. JVM を獲得するために待機しているトランザクションの遅延時間が過度に長いことがわかった場合、QR TCB 使用率のレベルを確認します。 Java プログラムによる CICS サービスの呼び出し (一時データ・キューにアクセスする JCICS クラスの使用など) では、QR TCB への切り替えが必要です。QR TCB が一度高いレベルの使用率に到達すると、それ以上の JVM を追加しても、CICS システムのスループットは向上しません。QR TCB 使用率のレベルは、CICS ディスパッチャー TCB モード統計の QR モードに関する「Accum CPU Time / TCB」統計フィールドを調べることによって確認できます。
3. プールされた JVM が独自の J8 および J9 TCB 上で使用するプロセッサー時間を調べます。すべての不要なプロセッサー使用を停止したことを確認してください。 詳しくは、『プールされた JVM によるプロセッサー使用量の確認』を参照してください。
4. プールされた JVM の数を増やしたい場合は、単一の JVM をサポートするために必要なストレージの量と、CICS 領域に使用可能な (または使用可能にできる) ストレージ・スペースの量とを比較して、CICS 領域がサポートできる JVM の最大数を計算します。 詳しくは、192 ページの『プールされた JVM のストレージ所要量の計算』を参照してください。
5. プロセッサーの使用量およびストレージの可用性について判明したことを考慮して、CICS 領域に最適な**MAXJVMTCBS** 制限を設定します。 CEMT SET DISPATCHER コマンドを使用して、CICS を再始動せずに、**MAXJVMTCBS** の設定を変更できます。
6. メッセージ DFHSJ0203 を受け取り、Language Environment から戻りコード 12 を受け取る場合、JVM のストレージ設定を調べます。JVM は 64 ビット・ストレージを使用します。CICS 領域の 64 ビット・ストレージの限界は、z/OS **MEMLIMIT** パラメーターによって制御されます。ストレージ設定または **MAXJVMTCBS** 制限、もしくはその両方を調整して、CICS 領域内で JVM が使用しているストレージの量を減らすことができます。 詳しくは、196 ページの『MVS ストレージ制約の処理』を参照してください。
7. JVM プール内でミスマッチおよびスチーリングが過度に発生している場合は、いくつかの手法を使ってこの数を減らすことができます。 詳しくは、196 ページの『過度のミスマッチおよびスチールの取り決め』を参照してください。
8. Java ワークロードが通常の予測可能なものであり、限られた数の異なる JVM プロファイルしか含まれていない場合、アプリケーションからの要求がある前に手動で JVM を開始できます。 この手法によって、ワークロードが増えたときに、期間中のアプリケーションの遅延時間が減少します。詳しくは、手動による JVM の始動と終了および JVM プールの無効化を参照してください。

プールされた JVM によるプロセッサー使用量の確認

CICS モニター機能を使用すると、JVM プログラムを呼び出すトランザクションによって使用されるプロセッサー時間をモニターすることができます。これには、プ

ールされた JVM によって J8 または J9 TCB で使用されるプロセッサー時間の量が含まれます。CICS モニター機能には、JVM で使用された経過時間、および JVM プログラムが発行した JCICS API 要求数も含まれています。

このタスクについて

JVM を調整する際の最初のステップは、それらが不必要なプロセッサー時間を使っていないことを確認することです。DFH$MOLS ユーティリティーを使用すると、SMF レコードを印刷したり、CICS Performance Analyzer などのツールを使用して SMF レコードを分析したりすることができます。

手順

1. CICS 領域でモニターをオンにして、パフォーマンス・クラスのモニター・データを収集します。
2. パフォーマンス・データ・グループ DFHTASK および DFHCICS を確認します。 特に、次のフィールドを調べることができます。

表 12. JVM 関連のモニター・データ・フィールド

グループ	フィールド ID	フィールド名	説明
DFHTASK	253	JVMTIME	ユーザー・タスクが JVM で費やした総経過時間。この時間は、JVM 初期化時間、Java アプリケーション実行時間、および JVM クリーンアップ時間で構成されています。フィールド JVMITIME および JVMRTIME は、初期化およびクリーンアップ時間をそれぞれ示しています。
DFHTASK	254	JVMSUSP	JVM で実行しているときに、ユーザー・タスクが CICS ディスパッチャーによって中断されている間の経過時間。
DFHTASK	260	J8CPUT	ユーザー・タスクが CICS J8 モード TCB (CICS キーの JVM に対して使用された) で CICS ディスパッチャー・ドメインによってディスパッチされたプロセッサー時間。フィールド JVMTIME は、JVM で使用した実際の経過時間を示しています。
DFHTASK	267	J9CPUT	ユーザー・タスクが CICS J9 モード TCB (ユーザー・キーの JVM に対して使用された) で CICS ディスパッチャー・ドメインによってディスパッチされたプロセッサー時間。フィールド JVMTIME は、JVM で使用した実際の経過時間を示しています。
DFHTASK	273	JVMITIME	JVM 環境の初期化に費やした経過時間。CICS 領域で初期化される最初の JVM は、そのタイプとは関係なく、その領域で初期化されるそれ以降の JVM よりも初期化時間が長くなります。これは、最初に必要なセットアップが原因です。

表 12. JVM 関連のモニター・データ・フィールド (続き)

グループ	フィールド ID	フィールド名	説明
DFHTASK	275	JVMRTIME	Java プログラムによる使用の後、JVM をクリーンアップするのに費やされた経過時間これには、CICS によってスケジュールされたガーベッジ・コレクションは含まれません。これは別のトランザクション (CJGC) にて実行されます。
DFHTASK	277	MAXJTDLY	CICS システムが、**MAXJVMTCBS** システム初期設定パラメーターによって設定された制限に達したために、ユーザー・タスクが CICS JVM TCB (J8 または J9 モード) を取得するために待機している間の経過時間。
DFHCICS	025	CFCAPICT	CICS OO 基礎クラス要求の数。ユーザー・タスクが発行した CICS (JCICS) クラスの Java API を含む。

3. プロセッサー使用量を改善するために、可能ならば、トレースの使用を削減または除去します。
   a. 実稼働環境では、CICS マスター・システムのトレース・フラグをオフに設定して、CICS 領域を稼働させることを検討してください。 このフラグをオンに設定すると、Java プログラムの実行によってプロセッサーの消費が著しく増加します。 SYSTR=OFF で CICS を初期化するか、CETR トランザクションを使用することによって、フラグをオフに設定することができます。
   b. JVM トレースは、特別なトランザクションに対してのみ活動化させるようにしてください。 JVM トレースは、短時間で大量の出力を生成することがあり、プロセッサー・コストを増加させます。JVM トレースの制御方法については 210 ページの『Java の診断』で説明しています。
4. 実稼働環境で JVM プロファイルでの USEROUTPUTCLASS オプションを使用しないでください。 このオプションを指定すると、JVM のパフォーマンスに悪影響が出ます。 USEROUTPUTCLASS オプションによって、同一の CICS 領域を使用する開発者は、JVM 出力を分離し、適切な宛先に送信することができます。ただし、このためには、追加クラス・インスタンスの作成および呼び出しが必要です。実稼働環境で最高のパフォーマンスを発揮するには、このオプションは使用しないでください。アプリケーション開発時に使用するために残しておいてください。 CICS 提供の JVM プロファイルでは、USEROUTPUTCLASS オプションは指定しません。
5. CICS 領域で異なるタイプのプールされた JVM を調べます。特に、単一使用 JVM があるかどうかチェックします。 単一使用 JVM は、継続 JVM に比べて大量のプロセッサー時間を使用します。アプリケーションがスレッド・セーフであれば、JVM サーバーで実行できます。スレッド・セーフでない場合は、継続のプールされた JVM で実行するためにアプリケーションを移動してください。

タイプ別のプールされた JVM がプロセッサー使用量に与える影響

プールされた JVM のタイプは継続または単独使用であり、オプションとして、継続 JVM は共用クラス・キャッシュを使用できます。どちらのタイプのプールされた JVM を選択するかによって、プロセッサー使用量に大きな影響を与える可能性があります。

継続的 JVM と単独使用 JVM

単独使用 JVM は、継続的 JVM と比較して、プロセッサー使用およびトランザクション・スループットのパフォーマンスが低くなります。新しい JVM は、プログラム呼び出しごとに初期化され、使用後には破棄されるため、プロセッサー使用コストが非常に高くなります。

1 つの単独使用 JVM を初期化するために必要な時間は、共用クラス・キャッシュを使用しない継続的 JVM のための時間より少し短くなりますが、共用クラス・キャッシュを使用する継続的 JVM のための時間より長くなります。ただし、この初期化は単独使用 JVM でプログラムが実行されるたびに行われるので、累積初期化時間およびトランザクションごとのプロセッサー時間が非常に長くなります。

実稼働環境で Java アプリケーションを実行するには、単独使用 JVM を使用しないでください。それらが役立つのは最初から単独使用 JVM で実行するように設計された Java アプリケーションだけで、再利用が意図された JVM での実行には適していません。単独使用 JVM で実行している Java プログラムがあれば、パフォーマンスを改善するための最初の処置は、これらの Java プログラムを再設計して、継続 Java で実行できるようにすることです。

以下の状況では、継続 JVM は定期的に再初期化が必要で、初期化コストが発生する可能性があります。

- CICS 領域内に異なる JVM プロファイルを使用する複数の混合した Java アプリケーションがあり、ミスマッチおよびスチールが生じる。
- Java ワークロードに変動があり、ワークロードが少ない期間、一部の JVM が長時間未使用のままであるためタイムアウトになる。

これらの状態が CICS 領域内であまりにも頻繁に生じる場合、その影響を回避または最小化するための方法を使用できます。

共用クラス・キャッシュを使用する JVM

共用クラス・キャッシュを使用する JVM の初期化時間は、かなり短くなります。これは、それらの JVM が、共用クラス・キャッシュで使用可能なプリロード済みクラスを使用するからです。

それぞれのトランザクションで使用されるプロセッサー時間に関しては、共用クラス・キャッシュを使用する継続的 JVM 内のほうがパフォーマンスが良いアプリケーションと、共用クラス・キャッシュを使用しない継続的 JVM 内のほうがパフォーマンスが良いアプリケーションがあります。トランザクションごとのプロセッサー時間が最優先の考慮事項であり、初期化時間よりも重要である場合、アプリケーションを両方のタイプの JVM でテストしてください。 JVM が定期的に再初期化される必要がある場合、評価の際には、共用クラス・キャッシュを使用する JVM の短い初期化時間を考慮してください。

プールされた JVM のストレージ所要量の計算

領域内のプールされた JVM 数を増やすには、十分なストレージが CICS に使用可能であることを確実にする必要があります。

このタスクについて

JVM は、16 MB 境界より下のストレージ、31 ビット・ストレージ、および 64 ビット・ストレージを使用します。CICS 領域で実行される JVM 数に関係なく、プールされた JVM の実行には一回限りのストレージ・コストが発生します。各プールされた JVM とその Language Environment エンクレーブにも、一定量の 31 ビットおよび 64 ビット・ストレージが必要です。JVM ヒープ・サイズのデフォルト値は、以下のとおりです。

表 13. ヒープ・サイズの JVM プロファイル・オプション

説明	JVM プロファイルのオプション	プールされた JVM プロファイルの値
ヒープ: 初期のストレージ割り振り	-Xms	16 MB
ヒープ: 最大サイズ	-Xmx	16 MB

JVM ヒープに必要なストレージは、CICS 領域の 64 ビット・ストレージ (2 GB 境界より上) から作成されます。JVM ヒープがこれより大きければ、CICS 領域に存在できる JVM の数は減り、それらの JVM をサポートするのに必要な **MEMLIMIT** サイズは増加します。ただし、ヒープ・サイズの設定が小さすぎると、ガーベッジ・コレクションが必要以上に行われるため、パフォーマンスに影響します。 Java ワークロードで最良のパフォーマンスを得るには、JVM ストレージ・オプションを調整する必要があります。JVM ストレージ・オプションは、Java アプリケーションのプロセッサー使用率、ストレージ使用量、およびタスクの応答時間の判別に役立ちます。

手順

1. サンプル統計プログラム DFH0STAT を使用して、2 GB 境界より下で使用可能なフリー・ストレージの量を判別します。ストレージ・レポートには、31 ビット・ストレージで 16 MB 境界より下で割り振られるユーザー・ストレージの量が記載されています。
   - 実行中の JVM がない場合、CICS アドレス・スペース内のフリー・ストレージの合計量から、z/OS 共用ライブラリー領域用に予約されているストレージを減算します。ストレージは、MVS で **SHRLIBRGNSIZE** パラメーターによって制御され、最初のプールされた JVM が領域で開始されるときに一回割り振られます。
   - 実行中のプールされた JVM がある場合は、フリー・ストレージの合計量から、**SHRLIBRGNSIZE** パラメーターの値を減算します。実行中の各 JVM は、16 MB 境界より下で 12 KB のストレージを使用します。JVM ごとの Language Environment エンクレーブは、ヒープおよびライブラリー・ヒープに 31 ビット・ストレージを使用します。割り振られる 31 ビット・ストレージの量は、

DFHJVMRO の HEAP64 オプションと LIBHEAP64 オプションで設定されます。また、現在使用可能なストレージ量を算出するには、これらの値を減算することも必要です。

31 ビット・ストレージの設定を変更したい場合は、**SHRLIBRGNSIZE** パラメーターと Language Environment オプションを調整することができます。 206 ページの『z/OS 共用ライブラリー領域の調整』および 205 ページの『DFHJVMRO を使用した、プールされた JVM のエンクレーブの変更』を参照してください。

2. 追加のプールされた JVM ごとに必要な 64 ビット・ストレージの量を計算します。 プールされた JVM に必要な 64 ビット・ストレージ量を計算するには、次のストレージ所要量を合計することができます。
   - JVM プロファイルの -Xmx 値。
   - DFHJVMRO の HEAP64 オプションで指定される 64 ビット・ストレージの量。
   - DFHJVMRO の LIBHEAP64 オプションで指定される 64 ビット・ストレージの量。
   - DFHJVMRO の STACK64 オプションで指定される 64 ビット・ストレージの量。各プールされた JVM で使用されるシステム・スレッドおよびアプリケーション・スレッドを組み込むには、この値に 5 を乗算してください。

   異なるヒープ・サイズを指定する複数の JVM プロファイルをアプリケーションが使用する場合、平均最大ヒープを見積もることができます。
   a. JVM プロファイルごとのアクティビティーのレベルおよび最大ヒープを報告する、JVM プロファイルの統計を収集します。
   b. 「Total number of requests for this profile」フィールドは、サンプリング期間中にアプリケーションによって各タイプの JVM が要求された回数を示しています。この回数は、通常は JVM プール内に存在する各タイプの JVM の比率を反映する場合があります。それぞれの JVM プロファイルごとの要求の合計数を、そのプロファイルで計算したストレージ要件で乗算します。
   c. すべての JVM プロファイルの結果を加算し、次に、この数値をサンプリング期間中の JVM に対する要求の合計数で除算します。
3. **MEMLIMIT** 値を調べて、追加のプールされた JVM の実行に使用できる十分な 64 ビット・ストレージがあるかどうかを判別します。64 ビット・ストレージを使用する他の CICS 機能を考慮する必要があります。

   z/OS **MEMLIMIT** パラメーターは、CICS 領域の 64 ビット (2 GB 境界より上) ストレージの量を制限します。 64 ビット・ストレージを使用する CICS 機能、およびこのパラメーターの確認と調整の方法については、「パフォーマンス・ガイド」の『MEMLIMIT の見積もり、確認、および設定』 を参照してください。

プールされた JVM のヒープとガーベッジ・コレクションの調整

プールされた JVM の JVM プロファイルで指定するガーベッジ・コレクション・オプションは、Java アプリケーションのパフォーマンスに大きな影響を与える場合があります。JVM のガーベッジ・コレクション処理からの出力を使用して、これらの設定を調整できます。

このタスクについて

割り振り失敗で起動するほかに、ガーベッジ・コレクションは CICS によって開始することも可能です。ストレージ・ヒープのアクティブな部分でのヒープ使用率が、指定された限度に達すると、CICS は System.gc() 呼び出しを使用してガーベッジ・コレクションを開始します。デフォルトは 85% ですが、これは、ストレージ・ヒープのアクティブな部分にあるストレージの 85% が使用されると、CICS がガーベッジ・コレクションをスケジュールに入れることを意味します。CICS は Java プログラムが実行されるたびにヒープ使用率をチェックします。使用率の限界に達すると、現在使用中の JVM が終了するとすぐに、ガーベッジ・コレクション・トランザクション CJGC を JVM で実行するようスケジュールされます。ただし、これらのガーベッジ・コレクション間で割り振り失敗が依然として発生する可能性があります。それは、ヒープ使用率が制限以下の状態で Java プログラムの実行が開始され、ヒープのアクティブな部分にある残りのストレージをすべて使用してしまい、より多くのストレージを必要とする場合です。

CICS によってスケジュールされたガーベッジ・コレクションは、個別のシステム・トランザクション CJGC として実行されます。ただし、割り振り失敗が原因で生じるガーベッジ・コレクションは、アプリケーションが JVM で実行されている間に行われます。アプリケーションの実行中にガーベッジ・コレクションが行われると、アプリケーションに遅延が発生し、ユーザー・トランザクションの CICS 統計にカウントされます。

手順

1. 調整したいプールされた JVM の JVM プロファイルを指定します。
2. JVM プロファイルを編集します。
   a. パフォーマンス目標に応じて、GC_HEAP_THRESHOLD オプションを設定し、ガーベッジ・コレクションが割り振り失敗によってではなく CICS によって開始されるようにすることによって、タスク応答時間を最小化することも可能です。CICS によってガーベッジ・コレクションを開始しない場合、GC_HEAP_THRESHOLD を 100 に設定できます。すべてのガーベッジ・コレクションは、アプリケーション実行時の割り振り失敗の結果として生じます。
   b. オプション -verbose:gc を指定します。 JVM は、JVM プロファイルの STDERR オプションで指定されるファイル (デフォルト名は dfhjvmerr) に、ガーベッジ・コレクションのメッセージを出力します。このファイルは、JVM プロファイルの WORK_DIR オプションで指定される z/OS UNIX ディレクトリーにあります。可能な場合には、このファイルのすべての既存メッセージを消去します (ファイルを削除すると再作成されます)。
   c. JVM の通常の振る舞いを現在のヒープ設定を使用して検査する場合には、最大と最小のヒープ・サイズ値を編集しないでください。この JVM プロファイルにより適したヒープ設定を決定しようとする場合、JVM プロファイルに以下の値を指定します。

```
-Xmx100M
-Xms1M
```

-Xmx 値が大きいため、ヒープは必要なサイズまで拡張することができます。-Xms 値は小さいため、ヒープは、必要なサイズよりも小さいサイズから開始して、Java ワークロードを実行するために必要とされる最小サイズまで拡張します。

3. **MAXJVMTCBS** システム初期設定パラメーターを 1 に設定します。この設定は、ユーザーの CICS が、**CEMT SET DISPATCHER MAXJVMTCBS** コマンドを使用して実行されている間に行うことができます。 CICS 提供のサンプル JVM プロファイルのデフォルト設定を使用した場合、CICS 領域内のすべての JVM からの出力は、同じファイルに送信されます。したがって、この場合は、ただ 1 つの JVM を持つことによって、ガーベッジ・コレクターの振る舞いの分析が簡単になります。代わりに、JVM プロファイルで STDERR オプションを変更して、JVM ごとに個別の出力ファイルを指定することもできます。
4. **CEMT INQUIRE JVM** コマンドを使用して、JVM プールの内容を表示します。いずれかの JVM が表示された場合には、**CEMT PERFORM JVMPOOL** コマンドを使用して JVM プールをパージします。 これにより、調整しようとしているプロファイルを使用する JVM は、-verbose:gc オプションおよび指定した新規ヒープ設定を使用して再作成されます。
5. TPNS (Teleprocessing Network Simulator) またはその他のネットワーク・シミュレーターを使用して、調整するプロファイルを使用した JVM の、通常のワークロード、または意図するワークロードを発生させるトランザクションを多数実行します。 ガイドとして、ほとんどの JIT コンパイルを確実に呼び出すために、すべての単一トランザクションは、約 1000 回実行する必要があります。ただし、所定の JVM に対して、トランザクションがこれほどの回数行われないことがわかっている場合には、代わりにそのトランザクションを最大予想回数だけ実行します。
6. 分析のために、ガーベッジ・コレクションからの出力を含んでいるファイルの位置を確認します。 出力は XML 形式です。ヒープ拡張を伴う割り振り失敗や、System.gc() 呼び出しによって起動されるガーベッジ・コレクションからの出力を含めて、出力の例と説明については、verbose:gclogging を参照してください。

   ヒント: Memory Analyzer ツールのファイルを使用すると、さらに詳細な分析を実行することができます。

   ガーベッジ・コレクションの出力は、以下の情報を示します。
   - ヒープ使用率のしきい値に達したときに CICS によって開始されたガーベッジ・コレクションの発生回数
   - 割り振り失敗によって生じたガーベッジ・コレクションの発生回数
   - ストレージ・ヒープ内のフリー・スペースの大きさ (バイト数およびパーセンテージ)
   - ガーベッジ・コレクションの前と後のフリー・スペースの大きさ
   - 各ガーベッジ・コレクションに要した時間 (ミリ秒)
   - ヒープ拡張の発生回数
   - ストレージ・ヒープが拡張した大きさ、および新しいヒープのサイズ (バイト数)

   System.gc() 呼び出しで起動されたガーベッジ・コレクションに要した合計時間は、出力に 2 回表示されます。1 回は排他的 VM アクセス権限を得るのに必要

な時間が除外されたもの、もう 1 回はその時間を含むものです。これらの合計時間のどちらに基づいてチューニングを行うこともできますが、同じ方を一貫して使用するようにし、両方を一緒に加算しないように注意してください。

タスクの結果

調整結果は、ユーザーの Java ワークロード、CICS および IBM SDK for z/OS の保守レベル、およびその他の要因によって異なります。ストレージとガーベッジ・コレクションの設定、および JVM の調整の可能性について詳しくは、「Java Diagnostics Guide」を参照してください。

MVS ストレージ制約の処理

CICS が使用可能な MVS ストレージ用に作成しようとするプールされた JVM が多すぎる場合、その結果として MVS ストレージの制約が生じる可能性があります。プールされた JVM は開始できず、CICS はメッセージ DFHSJ0203 を発行し、Language Environment は戻りコード 12 で失敗します。

このタスクについて

JVM は 64 ビット・ストレージを使用します。CICS 領域の 64 ビット (2 GB 境界より上) ストレージの量は、z/OS **MEMLIMIT** パラメーターによって制限されます。 CICS の実行中はこのパラメーターの値を変更できません。CICS 領域の次回の開始後に新しい値を指定できます。したがって、**MEMLIMIT** パラメーターを変更する前に、JVM プロファイルで他の設定を調整しておくことができます。

手順

1. JVM プロファイルのストレージ・ヒープが高く設定されすぎていないことを確認します。特に、ストレージ・ヒープの最大サイズを定義する `-Xmx` オプションを確認します。 JVM プロファイルの統計を収集するときに、CICS 領域で使用中のそれぞれの JVM プロファイルの `-Xmx` オプションが表示されます。 193 ページの『プールされた JVM のヒープとガーベッジ・コレクションの調整』では、これらの設定を変更する方法について示します。
2. 特定の JVM プロファイルでピーク使用量が高くなっているという問題が発生しているかどうか検査します。 この問題がある場合には、『過度のミスマッチおよびスチールの取り決め』に記述されている技法を使用して、そのプロファイルを使用して JVM を要求するトランザクションの数を制限することを検討します。これを行うには、その JVM プロファイルを要求する JVM プログラムを実行するトランザクションを同じトランザクション・クラス (TRANCLASS) に定義し、そのトランザクション・クラスを制限します。
3. CICS 領域で実行したい JVM の数を計算し、**MEMLIMIT** パラメーターの値を調整します。 192 ページの『プールされた JVM のストレージ所要量の計算』では、これを行う方法を示します。また、これに応じて MAXJVMTCBS 制限も調整する必要があります。

過度のミスマッチおよびスチールの取り決め

CICS は、プールされた JVM をアプリケーションに割り当て、ミスマッチおよびスチールを回避するよう試みることに意味がある場合には、いつでもそうします。しかし、JVM プールに適切な JVM がなく、スペースもない場合、CICS はミスマッ

チまたはスチールによりアプリケーション要求に対応することがあります。 CICS 統計を使用して、JVM プール内のミスマッチおよびスチールの発生が予想よりも多いかどうかを確認できます。必要に応じて、介入するのに使えるいくつかの手法があります。

このタスクについて

アプリケーションが JVM を要求すると、CICS はまず最初に、JVM プールで再使用可能な適切な JVM を検出しようとします。正しい JVM プロファイルおよび実行キーを持つ JVM が使用不可で、その JVM プールの MAXJVMTCBS 制限にまだ到達していない場合、CICS はそのアプリケーションに対して新規 JVM を作成できます。

JVM プールに適切な JVM がなく、スペースもない場合、CICS は、使用可能な JVM を破棄し、有効なプロファイルと実行キーを持つ JVM を再初期化します。JVM が破棄されて再初期化されるものの、TCB は保持されて再使用される場合、このプロセスはミスマッチ と呼ばれます。JVM および TCB が共に破棄されて置換される場合、スチール と呼ばれます。ミスマッチまたはスチールを許可する前に、CICS は、その選択メカニズムを使用して、許可する利点があるかどうかを判断します。

手順

1. JVM プール内のミスマッチおよびスチールの発生を評価します。 CICS ディスパッチャー TCB モード統計の TCB モード J8 および J9 に関する統計フィールド「TCB Mismatches」および「TCB Steals」は、JVM プール内のミスマッチおよびスチールの全体的な発生を示しています。JVM プロファイルの CICS 統計の「Number of times this profile stole a TCB」フィールドは、それぞれの JVM プロファイルごとのミスマッチおよびスチールの両方を組み合わせた発生回数を示しています。
2. CICS が JVM プール内に保持するそれぞれの JVM プロファイルの JVM の数は、指定できません。そのプロファイルを使用する JVM を要求するトランザクション数を制限することによって、特定の JVM プロファイルを使用する JVM の数を間接的に制限できます。
   a. その JVM プロファイルの必要な JVM プログラムを実行するトランザクションを同じトランザクション・クラス (TRANCLASS) に定義します。
   b. MAXACTIVE 値を TRANCLASS に割り当てます。

   この値は、その JVM プロファイルを要求する JVM プログラムの同時実行数を制限し、これにより、どの時点でも JVM プールに存在するその JVM プロファイルを使用する JVM の最大数を制限します。
3. 代わりに、CICS 領域が使用する異なる JVM プロファイルの数の削減を試行することができます。 JVM のタイプの数が少ないほど、アプリケーション要求に既存の JVM が一致する可能性が高くなるので、ミスマッチおよびスチールの発生回数は減少します。
   a. すべての JVM プロファイルおよび関連する JVM プロパティー・ファイルが、異なるオプションを指定していることを確認します。
   b. 単一の JVM プロファイルを作成するために異なる JVM プロファイルの互換性のあるオプションを結合することができるかどうかを調査します。 例え

ば、同様のオプションを含んでいる 2 つのほとんど使用されない JVM プロファイルがあり、1 つがより大きなストレージ・ヒープ・サイズを指定している場合、より大きなストレージ・ヒープ・サイズを指定している方の単一の JVM プロファイルにこれらのプロファイルを結合することができます。いくつかのアプリケーションがより大きい JVM を使用する可能性がありますが、ミスマッチとスチールの発生回数が減る方が大きなメリットがあります。

JVM の Language Environment エンクレーブ・ストレージ

JVM は、Language Environment 事前初期設定モジュール CELQPIPI を使用して作成される Language Environment エンクレーブで、z/OS UNIX システム・サービス・プロセスとして実行されます。このエンクレーブのランタイム・オプションを変更して、MVS によって割り振られるストレージを調整することができます。

JVM は、CICS Language Environment サービスではなく、MVS Language Environment サービスを使用します。この結果、JVM によって取得されたすべてのストレージは、 MVS Language Environment サービスの呼び出しを通じて取得された MVS ストレージになります。このストレージは、CICS アドレス・スペース内に常駐しています。ただし、CICS 動的ストレージ域 (DSA) には、含まれていません。CICS で実行されるすべての JVM は、64 ビット・ストレージを使用します。

それぞれの JVM の Language Environment エンクレーブは、JVM ストレージ・ヒープだけでなく、それぞれの JVM のストレージの基本量も含まれている必要があります。この基本ストレージ・コストは、JVM の構造で使用される Language Environment エンクレーブのストレージ量を表しています。 JVM の合計サイズを計算する場合には、基本ストレージ・コストをストレージ・ヒープが使用するストレージに追加する必要があります。

Language Environment ランタイム・オプションは、DFHAXRO および DFHJVMRO で設定されます。DFHAXRO は、JVM サーバーのオプションを提供し、DFHJVMRO は、プールされた JVM のオプションを提供します。これらのプログラムによって JVM エンクレーブに提供されるデフォルト値を表 14 に示しています。

表 14. CICS で JVM エンクレーブに使用される Language Environment のランタイム・オプション

Language Environment のランタイム・オプション	**JVM サーバーの値**	**プールされた JVM の値**
ヒープ・ストレージ	`HEAP64(100M,4M,KEEP,4M,512K, KEEP,1K,1K,KEEP)`	`HEAP64(8M,2M,KEEP,512K,512K, KEEP,1K,1K,KEEP)`
ライブラリー・ヒープ・ストレージ	`LIBHEAP64(3M,3M)`	`LIBHEAP64(1M,1M,FREE,16K,4K, FREE,4K,2K,FREE)`
ストレージ内のどこにでも常駐可能なライブラリー・ルーチン・スタック・フレーム	`STACK64(1M,1M,32M)`	`STACK64(1M,1M,32M)`
マルチスレッド・アプリケーションに対するオプションのユーザー・ヒープ・ストレージ管理	`HEAPPOOLS64(ALIGN)`	該当せず

表 14. CICS で JVM エンクレーブに使用される Language Environment のランタイム・オプション (続き)

Language Environment のランタイム・オプション	JVM サーバーの値	プールされた JVM の値
マルチスレッド・アプリケーションに対するオプションのヒープ・ストレージ管理	HEAPPOOLS(ALIGN)	該当せず
ストレージ不足の状態のために予約されたストレージ量、および割り振り時と解放時のストレージの初期内容	STORAGE(NONE,NONE,NONE)	STORAGE(NONE,NONE,NONE,0K)

Language Environment ランタイム・オプションについては、「*z/OS Language Environment カスタマイズ*」を参照してください。

Language Environment ランタイム・オプションをオーバーライドすることができます。

JVM サーバーのオプション

JVM サーバーの場合、 201 ページの『DFHAXRO を使用した JVM サーバーのエンクレーブの変更』で説明されているサンプル・プログラム DFHAXRO を変更し、再コンパイルします。このプログラムは JVMSERVER リソースで設定されるので、必要に応じて個々の JVM サーバーの Language Environment エンクレーブに別々のオプションを使用できます。Language Environment エンクレーブのヒープ・ストレージの初期サイズおよび増分を制御する Language Environment のデフォルトのストレージ設定値により、 MVS ストレージが非効率的に使用されることがあります。CICS が提供するストレージ設定の方が、より効果的です。また、JVM のストレージの使用状況に合わせて、これらの設定値を変更することもできます。多数のセグメント割り振りと解放を避けるようにヒープ・サイズが設定されていることを確認してください。

プールされた JVM のオプション

プールされた JVM の場合、 205 ページの『DFHJVMRO を使用した、プールされた JVM のエンクレーブの変更』で説明されている DFHJVMRO ユーザー置換可能モジュールを使用します。

MVS ストレージの使用を改善するには、DFHJVMRO を使用して、Language Environment エンクレーブのヒープ・ストレージ量の初期割り振りを、プールされた JVM で実行される Java アプリケーションが使用するストレージに近い値に設定して、これを初期ヒープ・サイズとして使用します。DFHJVMRO を使用して行う設定は、CICS 領域のすべての JVM に適用されます。このため、JVM のプロファイルが異なれば、ストレージ・ヒープ・サイズと基本ストレージ・コストも異なることを考慮してください。

Language Environment エンクレーブ内で 1 つの JVM が必要とするストレージの量によっては、**REGION** および **MEMLIMIT** サイズの制限に使用するインストール・システム出口 IEALIMIT または IEFUSI の変更が必要になることがあります。すべての Java プログラム要求のルート先として、Java 所有領域 (JOR) を使用する方法が考

えられます。このような領域で Java ワークロードのみを実行することにより、必要な CICS DSA ストレージの量を最小化し、最大量の MVS ストレージを JVM に割り振り可能にします。

JVM サーバーの Language Environment ストレージ必要量の識別

ストレージ・レポートを生成することによって、JVM サーバーの Language Environment エンクレーブ・ヒープ・ストレージの初期割り振りに適切な値を識別できます。ストレージ・レポートを生成するとプロセッサー・コストが増えるので、実稼働環境では適切な時間に実行してください。

このタスクについて

DFHAXRO の HEAP64 ランタイム・オプションは、JVM サーバーの Language Environment エンクレーブのヒープ・サイズを制御します。このオプションには、64 ビット・ストレージと 31 ビット・ストレージの設定が含まれています。必要に応じて、DFHAXRO の代わりに独自のプログラムを使用できます。このプログラムは JVMSERVER リソースで指定する必要があります。

手順

1. DFHJVMRO に RPTO(ON) および RPTS(ON) オプションを設定します。 これらのオプションは、DFHAXRO の指定されたソースでコメント化されています。これらのオプションを指定すると、Language Environment は、ストレージ・オプションについて報告し、実際に使用されているストレージを示すストレージ・レポートを作成します。
2. JVMSERVER リソースを使用不可にします。 JVM サーバーがシャットダウンし、Language Environment エンクレーブが除去されます。
3. JVMSERVER リソースを使用可能にします。 CICS は、DFHAXRO の Language Environment ランタイム・オプションを使用して、JVM サーバーのエンクレーブを作成します。JVM も開始します。
4. JVM サーバーで Java ワークロードを実行して、Language Environment エンクレーブで使用されるストレージに関するデータを収集します。
5. DFHAXRO から RPTO(ON) オプションと RPTS(ON) オプションを除去します。
6. JVMSERVER リソースを使用不可にして、ストレージ・レポートを生成します。 このストレージ・レポートには、初期の Language Environment エンクレーブ・ヒープ・ストレージの提案が含まれています。64 ビット・ユーザー・ヒープ統計の「Suggested initial size」項目に推奨値が含まれ、JVM サーバーで使用された Language Environment エンクレーブ・ヒープ・ストレージの合計量と等しくなります。

タスクの結果

ストレージ・レポートは、z/OS UNIX の stderr ファイルに保管されます。ディレクトリーは、JVM プロファイルで JVM の出力をリダイレクトしたかどうかによって異なります。リダイレクトが存在しない場合、このファイルは、JVM の作業ディレクトリーに保管されます。プロファイルで WORK_DIR に値が設定されていない場合、このファイルは /tmp ディレクトリーに保管されます。

ストレージ・レポートの情報を使用して、DFHAXRO で初期の Language Environment エンクレーブ・ヒープ・ストレージに適切な値を選択してください。Language Environment はヒープ・ストレージを追加できますが、初期割り振りで指定された不要なストレージを除去できません。割り振りや解放が行われたセグメントの数が確実に最小限になるように、十分なストレージを割り振ってください。

また、この手法を使用して、LIBHEAP64 および STACK64 ランタイム・オプションに初期サイズを設定し、値を増やすこともできます。

例

次の例は、Language Environment からのストレージ・レポートです。

```
64bit User HEAP statistics:
      Initial size:                                          50M
      Increment size:                                         4M
      Total heap storage used:                          91977408
      Suggested initial size:                                88M
      Successful Get Heap requests:                        2439
      Successful Free Heap requests:                       1619
      Number of segments allocated:                           1
      Number of segments freed:                               0
31bit User HEAP statistics:
      Initial size:                                      524288
      Increment size:                                    524288
      Total heap storage used (sugg. initial size):     8440784
      Successful Get Heap requests:                        1965
      Successful Free Heap requests:                       1904
      Number of segments allocated:                           2
      Number of segments freed:                               0
```

この例の Language Environment エンクレーブ・ヒープ・ストレージの値に基づいて、DFHAXRO でヒープ・ストレージに次の値を設定できます。

```
HEAP64(88M,4M,KEEP,10M,512K,KEEP,1K,1K,KEEP)
```

DFHAXRO を使用した JVM サーバーのエンクレーブの変更

DFHAXRO は、JVM サーバーが実行される Language Environment エンクレーブにデフォルトの実行時オプション・セットを提供するサンプル・プログラムです。例えば、JVM ヒープとスタックにストレージ割り振りパラメーターを定義します。CICS の場合、DFHAXRO で提供されるストレージ設定値は、Language Environment のデフォルトのストレージ設定値よりも適切です。

このタスクについて

このサンプル・プログラムを更新して Language Environment エンクレーブを調整するか、このサンプルに基づいて独自のプログラムを作成することができます。このプログラムは JVMSERVER リソースで定義され、JVM サーバー用に作成される Language Environment エンクレーブの CELQPIPI 事前初期設定段階時に呼び出されます。

このプログラムはアセンブラー言語で作成する必要があり、CICS 変換プログラムで変換してはなりません。オプションは文字ストリングとして指定され、2 バイトのストリングの長さと、それに続く実行時オプションで構成されます。すべての Language Environment 実行時オプションの最大長は 255 バイトです。したがって、

各オプションの省略バージョンを使用し、変更内容を全体で 200 バイト未満に制限してください。

手順

1. DFHAXRO プログラムを新規ロケーションにコピーして、実行時オプションを編集します。 CICS 領域に保守が適用される場合、プログラムの変更を反映したい場合があります。DFHAXRO のソースは、CICSTS42.CICS.SDFHSAMP ライブラリーにあります。
2. 各オプションの省略形を使用して、実行時オプションを編集します。 「*z/OS Language Environment プログラミング・ガイド*」に、Language Environment 実行時オプションに関する詳細情報が記載されています。
   - CICS はいくつかのオプションをこのリストに追加するため、高速処理のためにはオプションのリストのサイズを最小限に保つ必要があります。
   - HEAP64 オプションを使用して、初期のヒープ割り振りを指定します。
   - ALL31 オプション、POSIX オプション、および XPLINK オプションは、CICS によって強制的にオンにされます。 ABTERMENC オプションは、CICS によって (ABEND) に設定され、TRAP オプションは (ON,NOSPIE) に設定されます。
   - RPTO および RPTS オプションによって生成される出力は、CESE 一時データ・キューに書き込まれます。
   - 出力を生成するオプションがそれを行うのは、各 JVM の終了時です。生成後に CESE に送信される出力のボリュームを考慮に入れてください。
3. DFHASMVS プロシージャーを使用して、プログラムをコンパイルします。

タスクの結果

JVMSERVER リソースを使用可能にすると、CICS は、DFHAXRO プログラムで指定された実行時オプションを使用して Language Environment エンクレーブを作成します。CICS は、Language Environment に渡す前に実行時オプションの長さをチェックします。この長さが 255 バイトより長い場合、CICS は JVM サーバーの開始を試みず、CSMT にエラー・メッセージを書き込みます。指定された値は、CICS によってチェックされずに Language Environment に渡されます。

JVM 統計を使用した Language Environment 必要量の識別

CICS 統計を使用して、ユーザーの JVM が使用する Language Environment エンクレーブ・ヒープ・ストレージの量を確認できます。JVM プロファイル統計の「Peak Language Environment heap storage used (使用されているピーク Language Environment ヒープ・ストレージ)」フィールドは、指定された実行キーおよびプロファイルを使用する JVM によって使用された Language Environment エンクレーブ・ヒープ・ストレージのピーク (または最高水準点) 量を示しています。この統計の収集は、JVM のパフォーマンスに影響を与えます。このため、この処理は実稼働環境では適切な時間に実行してください。

手順

1. **EXEC CICS INQUIRE JVMPROFILE** コマンドを使用して、CICS 領域で使用中のそれぞれの JVM プロファイルを識別します。 (このコマンドに等価な CEMT はありません。)

2. 識別したそれぞれの JVM プロファイルにオプション LEHEAPSTATS=YES を指定します。
3. 統計リセット時刻の付近で (前または直後)、CEMT SET JVMPOOL PHASEOUT コマンド (または等価の EXEC CICS コマンド) を使用して JVM をパージします。 これにより、次の統計間隔で収集される統計が、JVM のストレージ使用量をより正確に反映したものになります。また、これによってユーザーの JVM は、LEHEAPSTATS=YES オプションを使用して再作成されます。
4. ユーザーの JVM を使用するトランザクションの代表的なサンプルを実行します。
5. EXEC CICS COLLECT STATISTICS JVMPROFILE または CEMT PERFORM STATISTICS JVMPROFILE コマンドを使用して、JVM プロファイル統計を収集するか、または統計インターバル中に収集された JVM プロファイル統計を確認します。
6. ユーザーの JVM プロファイルからオプション LEHEAPSTATS=YES を除去するか、NO (デフォルトです) に変更します。
7. CEMT SET JVMPOOL PHASEOUT コマンドを使用して JVM をパージして、このオプションがオプション LEHEAPSTATS=NO を使用して再作成されていることを確認します。
8. それぞれの JVM プロファイルごとに、JVM プロファイル統計の「Peak Language Environment heap storage used (使用されているピーク Language Environment ヒープ・ストレージ)」フィールドを検査します。

タスクの結果

「Peak Language Environment heap storage used (使用されているピーク Language Environment ヒープ・ストレージ)」フィールドの値を使用して、DFHJVMRO での初期ヒープ・サイズとして設定します。使用されているストレージのピーク量が JVM プロファイル間で変化している場合には、それぞれの JVM プロファイルの相対使用量に基づいて、適切な値を選択します。ほとんどの JVM が使用するストレージに近い値を選択するようにしてください。Language Environment はヒープ・ストレージを追加できますが、初期割り振りで指定された不要なストレージを除去できません。

Language Environment ストレージの識別には DFHJVMRO の使用が必要

DFHJVMRO で追加のランタイム・オプションを設定すると、Language Environment エンクレーブ・ヒープ・ストレージ量の初期割り振りに適切な値を識別できます。これらのオプションはプロセッサー・コストを増やすので、実稼働環境では適切な時間に使用してください。

このタスクについて

DFHJVMRO の `HEAP64` ランタイム・オプションは、Language Environment エンクレーブのヒープ・サイズを制御します。このオプションには、64 ビット・ストレージと 31 ビット・ストレージの設定が含まれています。

DFHJVMRO は、それぞれのストレージ・レポートが適用される JVM プロファイルを識別できません。このため、一度にただ 1 つの JVM プロファイルに対してこ

の手順を使用します。その JVM プロファイルのみを要求するトランザクションを使用していることを確認してください。

手順

1. DFHJVMRO に RPTO(ON) および RPTS(ON) オプションを設定します。 これらのオプションは、DFHJVMRO の指定されたソースでコメント化されています。これらのオプションを指定すると、Language Environment は、ストレージ・オプションについて報告し、実際に使用されているストレージを示すストレージ・レポートを作成します。
2. JVM プール内のすべての JVM をパージして、それら JVM が RPTO(ON) および RPTS(ON) オプションを使用して再作成されていることを確認します。 CICS Explorer で「**Operations**」 > 「**Java**」 > 「**JVM pools**」ビューを使用するか、**CEMT SET JVMPOOL PHASEOUT** コマンドを使用できます。
3. 検査対象の JVM プロファイルで、プールされた JVM を使用するトランザクションの代表的なサンプルを実行します。 プログラムの JVM プロファイルは、PROGRAM リソースで指定されます。
4. DFHJVMRO から RPTO(ON) および RPTS(ON) オプションを除去します。
5. JVM をパージします。 ストレージ・レポートは、それぞれの JVM の終了時に書き込まれます。このストレージ・レポートには、初期の Language Environment エンクレーブ・ヒープ・ストレージの提案が含まれています。「Total heap storage used (sugg. initial size)」項目に推奨値が含まれ、JVM で使用された Language Environment エンクレーブ・ヒープ・ストレージの合計量と等しくなります。
6. 使用されているストレージの量のすべての変動を検査するために、一連のすべてのストレージ・レポートを調べます。

タスクの結果

DFHJVMRO で初期の Language Environment エンクレーブ・ヒープ・ストレージに適切な値を選択します。ほとんどの JVM が使用するストレージに近い値を選択するようにしてください。Language Environment はヒープ・ストレージを追加できますが、初期割り振りで指定された不要なストレージを除去できません。

また、この手法を使用して、LIBHEAP64 および STACK64 ランタイム・オプションに初期サイズを設定し、値を増やすこともできます。

例

次の例は、Language Environment からのストレージ・レポートです。

```
64bit User HEAP statistics:
   Initial size:                                    8M
   Increment size:                                  2M
   Total heap storage used:                  30573536
   Suggested initial size:                        30M
   Successful Get Heap requests:               24395
   Successful Free Heap requests:              12416
   Number of segments allocated:                   7
   Number of segments freed:                       0

31bit User HEAP statistics:
   Initial size:                              524228
   Increment size:                            524228
```

```
Total heap storage used (sugg. initial size):      1099824
Successful Get Heap requests:                          599
Successful Free Heap requests:                         567
Number of segments allocated:                            2
Number of segments freed:                                0
```

この例の Language Environment エンクレーブ・ヒープ・ストレージの値に基づいて、DFHJVMRO でヒープ・ストレージに次の値を設定できます。

```
HEAP64(30M,2M,KEEP,1099824,512K,KEEP,1K,1K,KEEP)
```

DFHJVMRO を使用した、プールされた JVM のエンクレーブの変更

DFHJVMRO は、プールされた JVM が実行される Language Environment エンクレーブの作成に使用される実行時オプションを指定します。このモジュールは、ヒープおよびスタック用のストレージ割り振りパラメーター、ならびにほかのオプションをいくつか定義します。CICS の場合、DFHJVMRO で提供されるストレージ設定値は、Language Environment のデフォルトのストレージ設定値よりも適切です。

このタスクについて

DFHJVMRO はユーザー置換可能モジュール (URM) であり、プールされた JVM ごとに Language Environment エンクレーブの CELQPIPI 事前初期設定段階時に呼び出されます。以下の状態では、提供されるバージョンのプログラムを変更することができます。

- RPTO および RPTS オプションを使用して、JVM 用のストレージ・オプションのセットおよび使用される実際のストレージに関するレポートを取得する。
- 提供の設定と異なるストレージ・ヒープ値をエンクレーブに設定する。Java ヒープは、エンクレーブ・ヒープとは別個に割り振られます。
- IBM サービス・チームからの要求がある場合には、他のオプションを設定して診断情報を取得してください。

このプログラムはアセンブラー言語で作成する必要があり、CICS 変換プログラムで変換してはなりません。オプションは文字ストリングとして指定され、2 バイトのストリングの長さと、それに続く実行時オプションで構成されます。すべての Language Environment 実行時オプションの最大長は 255 バイトです。したがって、各オプションの省略バージョンを使用し、変更内容を全体で 200 バイト未満に制限してください。

手順

1. DFHJVMRO プログラムを新規ロケーションにコピーして、実行時オプションを編集します。 CICS 領域に保守が適用される場合、プログラムの変更を反映したい場合があります。DFHJVMRO のソースは、CICSTS42.CICS.SDFHSAMP ライブラリーにあります。
2. 各オプションの省略形を使用して、実行時オプションを編集します。DFHJVMRO のソース・コードには、これらのオプションの設定方法の例と一緒にコメントが記載されています。「*z/OS Language Environment プログラミング・ガイド*」に、Language Environment 実行時オプションに関する詳細情報が記載されています。

- CICS はいくつかのオプションをこのリストに追加するため、高速処理のためにはオプションのリストのサイズを最小限に保つ必要があります。
- HEAP64 オプションを使用して、初期のヒープ割り振りを指定します。
- XPLINK オプションは CICS によって強制的にオンになり、したがって ALL31 オプションは Language Environment によって強制的にオンになります。POSIX オプションは、AMODE(64) オプションがあるため、デフォルトで ON に設定されます。ABTERMENC はデフォルトで ABEND に設定され、TRAP オプションは CICS によって (ON,NOSPIE) に設定されます。
- RPTO および RPTS オプションによって生成される出力は、CESE 一時データ・キューに書き込まれます。
- 出力を生成するオプションがそれを行うのは、各 JVM の終了時です。生成後に CESE に送信される出力のボリュームを考慮に入れてください。

3. オプションの長さが 200 文字を超えないことを確認します。 最大長は 255 文字ですが、CICS がいくつかのオプションを自動的に追加します。
4. DFHASMVS プロシージャーを使用して、プログラムをコンパイルします。

タスクの結果

CICS が Java プログラムの実行要求を受け取ると、CICS は、DFHJVMRO プログラムで指定された実行時オプションを使用して、プールされた JVM の Language Environment エンクレーブを作成します。CICS は、Language Environment に渡す前に実行時オプションの長さをチェックします。この長さが 255 バイトより長い場合、CICS はプールされた JVM の開始を試みず、CSMT にエラー・メッセージを書き込みます。指定された値は、CICS によってチェックされずに Language Environment に渡されます。

z/OS 共用ライブラリー領域の調整

共用ライブラリー領域は、 z/OS の機能であり、アドレス・スペース間のダイナミック・リンク・ライブラリー (DLL) ファイルの共用を可能にします。この機能により、 CICS 領域は JVM に必要な DLL を共用できるようになり、各領域が DLL を個別にロードする必要はなくなります。これにより、MVS が使用する実際のストレージの量、および領域へのファイルのロード所要時間を大幅に削減できます。

共用ライブラリー領域用に予約されているストレージは、最初の JVM が領域で開始されるときにそれぞれの CICS 領域に割り振られます。割り振られるストレージの量は、z/OS の **SHRLIBRGNSIZE** パラメーターによって制御されます。このパラメーターは、SYS1.PARMLIB の BPXPRMxx メンバー内にあります。最小値は 16 MB で、z/OS のデフォルトは 64 MB です。必要なスペースの量を調査することによって、共用ライブラリー領域用に割り振られるストレージの量を調整できます。CICS 以外の他のアプリケーションが、その共用ライブラリー領域を使用しており、それに応じて **SHRLIBRGNSIZE** パラメーターを調整している可能性があることに注意してください。

共用ライブラリー領域用に割り振られるストレージの量を削減する場合は、最初に共用ライブラリー領域に無駄なスペースがないことを確認します。z/OS システムで通常の作業負荷をかけ、コマンド **D OMVS,L** を発行してライブラリー統計を表示します。共用ライブラリー領域に未使用のスペースがある場合は、**SHRLIBRGNSIZE** の

設定を減らしてこのスペースを除去できます。CICS が、共用ライブラリー領域の唯一のユーザーの場合は、**SHRLIBRGNSIZE** を最小の 16 MB まで削減できます。これは、JVM が必要とする DLL が、約 10 MB の領域しか使用しないためです。

共用ライブラリー領域内のすべてのスペースが使用されている状態でも、CICS 領域におけるこのストレージ割り振りを削減したい場合は、以下の 3 つの手順が検討できます。

1. 共用ライブラリー領域サイズをファイルで必要なストレージの量より小さく設定することは可能です。共用ライブラリー領域がフルの場合には、代わりにファイルは専用ストレージにロードされ、共用機能の利点は得られません。この方法を選択した場合は、より重要なアプリケーションを先に開始して、共用ライブラリー領域を確実に利用できるようにする必要があります。共用ライブラリー領域内のスペースのほとんどが、重要ではないアプリケーションによって使用されている場合は、この方法が最も適切です。
2. 共用ライブラリー領域内に配置される DLL は、拡張属性 +l によって識別されます。一部のファイルからこの属性を除去し、これらのファイルが共用ライブラリー領域に配置されることを防止することで、共用ライブラリー領域に必要なストレージ量を削減できます。この方法を選択する場合は、共用される頻度が少ないファイルを選択してください。また、拡張子 .so 付きのファイルは選択しないようにしてください。拡張子 .so が付いたファイルは、共用ライブラリー領域内に配置されない場合、ユーザー共用ライブラリーを通じて共用されますが、共用ライブラリー領域を使用した場合と比較して、この共用機能は非効率です。共用ライブラリー領域内のスペースのほとんどが、拡張子 .so が付いていない大きなファイルによって使用されている場合は、この方法が最適です。
3. CICS JVM に関連するすべてのファイルから拡張属性 +l を除去すると、 CICS 領域では共用ライブラリー領域が完全に使用されなくなり、 CICS 領域内で共用ライブラリー領域用のストレージが割り振られることもありません。この方法を選択した場合は、共用ライブラリー領域の共用機能を利用することはできません。 z/OS システム上の他のアプリケーションが大きな共用ライブラリー領域を必要としているが、 CICS 領域内でこのストレージ量を割り振りたくない場合は、この方法が最適です。

いずれかのファイルから拡張属性 +l を除去した場合は、これらのファイルを新規バージョンで置き換える際に (例えば、ソフトウェア・アップグレードの場合)、新バージョンのファイルにこの属性が設定されていないことを忘れずに確認してください。

z/OS UNIX の共用ライブラリーについて詳しくは、http://www.ibm.com/servers/eserver/zseries/zos/unix/perform/sharelib.html の z/OS UNIX System Services Web サイトを参照してください。

第 8 章 Java アプリケーションのトラブルシューティング

Java アプリケーションに問題がある場合は、CICS と JVM で提供される診断機能を使用して、問題の原因を調べることができます。

このタスクについて

CICS は、Java に関連した問題の診断に役立ついくつかの統計、メッセージ、およびトレースを提供します。Java に付属の診断ツールとインターフェースは、JVM で何が発生しているかについて CICS よりも詳しい情報を提供できます。これは、CICS が JVM 内のアクティビティーの多くを認識しないからです。

JVM の分析をリアルタイムとオフラインで実行する無料のツール (例えば、JConsole や IBM Health Center) を使用できます。詳しくは、「*Java Diagnostics Guide*」の Using diagnostic tools を参照してください。

手順

1. JVM サーバーもプールされた JVM も開始できない場合は、Java インストールのセットアップが正しいことを確認します。 CICS メッセージおよび JVM の `stderr` ファイルにあるエラーを使用して、問題の原因を判別してください。
   a. 正しいバージョンの Java SDK がインストールされていること、および CICS が z/OS UNIX 内の Java SDK にアクセスできることを確認します。 CICS は IBM 64-bit SDK for z/OS, Java テクノロジー・エディション バージョン 6.0.1 をサポートします。
   b. **USSHOME** システム初期設定パラメーターが CICS 領域で設定されていることを確認します。 このパラメーターは、z/OS UNIX 上のファイルのホームを指定します。
   c. **JVMPROFILEDIR** システム初期設定パラメーターが CICS 領域で正しく設定されていることを確認します。 このパラメーターは、z/OS UNIX 上の JVM プロファイルの場所を指定します。
   d. JVM プロファイルが入っている z/OS UNIX ディレクトリーへの読み取りアクセス権限と実行アクセス権限が CICS 領域にあることを確認します。
   e. JVM の作業ディレクトリーへの書き込みアクセス権限が CICS 領域にあることを確認します。 このディレクトリーは、JVM プロファイルの `WORK_DIR` オプションで指定されます。
   f. JVM プロファイルの `JAVA_HOME` オプションが、Java SDK が入っているディレクトリーを指し示していることを確認します。
   g. SDFJAUTH が CICS 開始 JCL の STEPLIB 連結にあることを確認します。
   h. WebSphere MQ または DB2 の DLL ファイルを使用している場合は、これらのファイルの 64 ビット・バージョンが CICS から使用可能であることを確認します。
   i. Language Environment エンクレーブを構成するために DFHAXRO または DFHJVMRO を変更した場合は、実行時オプションが 200 バイトを超えないこと、およびそれらのオプションが有効であることを確認します。 CICS

は、指定されるオプションを検証せずに、Language Environment に渡します。 Language Environment からのエラー・メッセージがないかどうか、SYSOUT を確認してください。

2. セットアップが正しい場合は、診断情報を収集して、アプリケーションと JVM に何が起きているかを調べます。
   a. JVM プロファイルに PRINT_JVM_OPTIONS=YES を追加します。 このオプションを指定すると、クラスパスの内容を始めとして、開始時に JVM に渡されるすべてのオプションが SYSPRINT に出力されます。 JVM がプロファイル内でこのオプションを指定して開始されるたびに、情報が生成されます。
   b. JVM からの情報およびエラー・メッセージがないか、dfhjvmout ファイルと dfhjvmerr ファイルを調べます。 これらのファイルは、JVM プロファイルの WORK_DIR オプションで指定されるディレクトリーにあります。JVM プロファイルで STDOUT オプションと STDERR オプションが変更された場合、これらのファイルの名前が異なる可能性があります。
3. アプリケーションが障害を起こしたか、アプリケーションのパフォーマンスが低下する場合は、JPDA デバッガーを使用してアプリケーションをデバッグします。
4. メモリー不足エラーが表示される場合、64 ビット・ストレージが不十分であるか、アプリケーションにメモリー・リークがあるか、またはヒープ・サイズが非常に小さい可能性があります。
   a. CICS 統計またはツール (IBM Health Center など) を使用して、JVM をモニターします。アプリケーションにメモリー・リークがある場合、ガーベッジ・コレクション後に残っているライブ・データの量が、ヒープが使い果たされるまで時間と共に徐々に増えます。 JVM サーバー統計は、最後のガーベッジ・コレクション後のヒープのサイズ、およびヒープの最大サイズとピーク・サイズを報告します。
   b. Language Environment のストレージ・レポート機能を実行して、使用可能なストレージが十分にあるかどうかを確認します。 198 ページの『JVM の Language Environment エンクレーブ・ストレージ』を参照してください。

次のタスク

問題の原因を確定できない場合は、IBM サポートにお問い合わせください。 Java の問題を報告するための MustGather にリストされているとおりに、必要な情報を確実に提供してください。

Java の診断

通常の CICS 診断情報源の多くには、Java アプリケーションに適用される情報が含まれています。CICS 提供の情報に加えて、問題判別に使用できる、JVM 固有の複数のインターフェースがあります。

Java の CICS 診断ツール

CICS には、実行中の Java アプリケーションに関して収集できる統計とモニター・データがあります。エラーが発生すると、トランザクションは異常終了し、メッセージが該当するログに書き込まれます。JVM (SJ) ドメインに適用される異常終了と

メッセージのリストについては、Messages and Codes Vol 2 の CICS Messages and Codes 概要 を参照してください。 Java に関連したメッセージの形式は DFHSJxxxx です。

また、トレースをオンにして、追加の診断情報を生成することもできます。 JVM ドメインのトレース・ポイントは、Trace Entries の JVM ドメインのトレース・ポイント にリストされています。

初期化後に最初の JVM が CICS 領域で開始すると、CICS は、メッセージ DFHSJ0207 を発行して、使用されている Java のバージョンを表示します。

Java SDK が提供する診断ツールとインターフェースは、JVM で何が発生しているかについてより詳細な情報を提供します。 JVM からのメッセージと診断情報は、JVM の stderr ログ・ファイルに書き込まれます。Java の問題を検出した場合は、必ずこのファイルを調べてください。例えば、CICS がメッセージを発行して、JVM が異常終了したことを示す場合、stderr ログ・ファイルが第一の診断情報源です。213 ページの『JVM stdout、stderr およびダンプ出力の場所の制御』では、JVM からの出力の場所を制御する方法、および JVM 内部からのメッセージならびに JVM で実行中の Java アプリケーションからの出力のリダイレクト方法を示しています。

CICS 用の Java アプリケーションを開発する際には、CICS におけるスレッド・セーフティーとトランザクション分離の要件を考慮することが重要です。 Java アプリケーションが、最初に使用されるときに正しく機能するものの、それ以降の使用時に正しく動作しない場合、この問題はおそらく、分離の問題が原因です。この場合は、問題判別作業の一環として CICS JVM アプリケーション分離ユーティリティーを使用して、問題の原因の特定に役立ててください。

OSGi 診断ファイル

OSGi フレームワークは、JVM サーバーにおける OSGi バンドルおよびサービスの問題のトラブルシューティングに使用できる診断ファイルを zFS で作成します。

OSGi キャッシュ

OSGi キャッシュは、JVM サーバーの *$WORK_DIR*/*applid*/*jvmserver*/configuration/org.eclipse.osgi ディレクトリーにあります。*$WORK_DIR* は JVM サーバーの作業ディレクトリー、*applid* は CICS APPLID、*jvmserver* は JVMSERVER リソースの名前です。OSGi キャッシュには、フレームワークのメタデータ、およびフレームワークの実行に必要なその他の情報が入っています。JVM サーバーが始動するときに、キャッシュが置き換えられます。

OSGi ログ

OSGi フレームワークでエラーが発生すると、OSGi ログが、JVM サーバーの *$WORK_DIR*/*applid*/*jvmserver*/configuration/ ディレクトリーに作成されます。ファイル拡張子は .log です。 OSGi フレームワークは、サイズが 1000 KB に達するまでログ・ファイルに書き込み続けます。その後、OSGi フレームワークは、さらにエラー・メッセージを書き出すために別のログ・ファイルを作成します。そのディレクトリーに最大 10 個のログ・ファイルを保持できます。10 個目のログ・ファイルが満杯になると、OSGi フレームワークは最も古いログ・ファイルを上書きします。

JVM 診断ツール

CICS 資料には、一部の Java 診断ツールとインターフェースに関する情報が記載されています。

- 219 ページの『JVM サーバーのトレースの活動化と管理』では、CETR トランザクションによって提供されるコンポーネント・トレースを使用して、JVM サーバーと JVM サーバー内で実行されるタスクのライフサイクルをトレースする方法を説明しています。JVM サーバーは補助トレースも GTF トレースも使用しません。代わりに、JVM サーバーごとに固有に名前が指定される zFS 上のファイルにトレースが書き込まれます。
- 220 ページの『プールされた JVM 用のトレースの定義および活動化』では、CICS に付属のインターフェースを使用して、プールされた JVM の内部トレース機能を使用する方法を説明しています。内部トレース機能は、JVM 内の入り口点、出口点、およびイベント・ポイントの詳細なトレースを行うことができます。この情報は、CICS トレースとして出力されます。
- 223 ページの『Java アプリケーションのデバッグ』 では、リモート・デバッガーを使用して、JVM で実行されている Java アプリケーションのアプリケーション・コードをステップスルーする方法について説明しています。また、CICS は CICS Java ミドルウェアで 1 組の代行受信ポイント (すなわち「プラグイン」) も提供します。これにより、デバッグ、ロギング、またはその他の目的に、アプリケーション Java コードの実行直前と直後に追加の Java プログラムを挿入できます。詳しくは、 224 ページの『CICS JVM プラグイン・メカニズム』を参照してください。

JVM にはより多くの診断ツールとインターフェースが使用できます。JVM の問題判別に使用できる追加の機能については、Java Diagnostics Guide を参照してください。次の機能が、役に立つ診断情報を提供します。

- CICS が提供するインターフェースを使用することなく、JVM の内部トレース機能を直接使用できます。内部トレース機能の制御と、各種宛先への JVM トレース情報の出力に使用できるシステム・プロパティーに関する情報は、「*Diagnostics Guide*」に記載されています。これらのシステム・プロパティーを使用すると、JVM 内の任意のメソッドまたはクラスからトレースを出力し、メソッド呼び出しで任意のパラメーターと戻りの型の値を検出することができます。
- JVM にメモリー・リークが検出される場合、JVM にヒープ・ダンプを要求できます。ヒープ・ダンプは、JVM のヒープ内にあるすべてのライブ・オブジェクト (引き続き使用中のオブジェクト) のダンプを生成します。また、IBM Health Center や Memory Analyzer ツールを使用して、メモリー・リークを分析することもできます。これらのツールはどちらも、IBM Support Assistant で入手可能です。Java ツールについて詳しくは、IBM Monitoring and Diagnostics Tools for Java を参照してください。
- IBM 64-bit SDK for z/OS, Java テクノロジー・エディション に付属の HPROF プロファイラーは、JVM で実行されるアプリケーションのパフォーマンス情報を提供します。したがって、プログラムのどの部分が最大のメモリーまたはプロセッサー時間を使用しているかが分かります。
- JVM は、モニター、プロファイル作成、および RAS (信頼性・可用性・保守性) 用のインターフェースを提供します。

CICS 環境に固有ではなく、IBM JVM に使用可能なすべてのインターフェース、オプション、またはシステム・プロパティーでは、IBM JVM 資料を第一の情報源として使用してください。

JVM stdout、stderr およびダンプ出力の場所の制御

JVM で実行中の Java アプリケーションからの出力は、通常、その JVM の JVM プロファイルにある STDOUT オプションと STDERR オプションで指定される z/OS UNIX ファイルに書き込まれます。JAVADUMP ファイルは、z/OS UNIX 上の JVM の作業ディレクトリーに書き込まれ、より詳細な Java TDUMP は、JAVA_DUMP_TDUMP_PATTERN オプションで指定されるファイルに書き込まれます。これらのファイル名の大部分は実行時にカスタマイズすると、それらのファイルを生成した JVM を一意的に識別できます。また、アプリケーション開発時に、Java クラスを使用して、JVM からの出力および JVM 内部からのメッセージをリダイレクトすることもできます。

CICS JVM の標準のセットアップでは、JVM プロファイルの STDOUT オプションで指定されたファイルは、System.out 要求に使用されます。STDERR オプションで指定されたファイルは、System.err 要求に使用されます。出力ファイルは、JVM プロファイルの WORK_DIR オプションで指定された作業ディレクトリーにある z/OS UNIX ファイルです。

stdout ファイルと stderr ファイルに固定のファイル名を指定できます。ただし、固定のファイル名を使用する場合、その JVM プロファイルを使用して作成されたすべての JVM からの出力が、同じファイルに追加され、異なる JVM からの出力はレコード・ヘッダーなしにインターリーブされます。これは、問題判別には役立ちません。

より良い方法は、stdout ファイルと stderr ファイルに可変ファイル名を指定することです。可変ファイル名を指定する場合、CICS 領域の存続期間中、これらのファイルを個々の JVM ごとに固有にすることができます。また、追加の識別情報を組み込むこともできます。

- 固有の JVM 番号は、JVM を CICS 領域内の他の JVM と区別します。CICS で使用される JVM 番号は、z/OS UNIX 環境で JVM の識別に使用される番号 (JVM のプロセス ID (PID) と呼ばれます) と同じです。この番号は、&JVM_NUM; シンボルまたは **-generate** オプションを使用して、ファイル名の一部として指定できます。
- &APPLID; シンボルまたは **-generate** オプションを使用することによって、ファイル名に CICS 領域アプリケーション ID を組み込むことができます。
- **-generate** オプションを使用して、ファイル名にタイム・スタンプを組み込むことができます。

ファイル名のその他の識別情報には、&DATE; シンボルと &TIME; シンボルがあります。

&DATE; は、形式 Dyymmdd の現在日付で置き換えられます。

&TIME; は、形式 Thhmmss の現在時刻で置き換えられます。

JVM から出力された JAVADUMP ファイルの場所は、JVM プロファイルの WORK_DIR オプションで指定された、z/OS UNIX 上の作業ディレクトリーです。JAVADUMP ファイルは、それらの名前にあるタイム・スタンプで一意的に識別されます。これらのファイルの名前をカスタマイズすることはできません。

JVM から出力された TDUMP は、JVM のアドレス・スペースを始めとする詳細なダンプ出力を含み、データ・セット宛先に書き込まれます。宛先の名前は、JVM プロファイルの JAVA_DUMP_TDUMP_PATTERN オプションで指定されます。CICS 提供のサンプル JVM プロファイルで示されているように、この値で &APPLID;、&DATE;、&JVM_NUM;、および &TIME; シンボルを使用すると、個々の JVM に固有の名前にすることができます。このため、MVS データ・セット命名標準に準拠するために、CICS が JVM 番号を変更しなければならない可能性があることに注意してください。

JVM は、JAVADUMP または TDUMP を生成すると、情報をその stderr ファイルに書き込みます。Java Diagnostics Guide には、JAVADUMP ファイルと TDUMP ファイルの内容に関する詳細情報が記載されています。

アプリケーション開発時に、JVM プロファイルの USEROUTPUTCLASS オプションを使用すると、JVM からの出力と JVM 内部からのメッセージを代行受信し、リダイレクトする Java クラスを指定できます。出力レコードにタイム・スタンプとヘッダーを追加し、JVM で実行中の個々のトランザクションからの出力を識別することができます。CICS が提供するサンプル・クラスが、これらのタスクが実行します。このオプションを指定すると、JVM のパフォーマンスに悪影響が出るため、実稼働環境では使用しないでください。

JVM stdout および stderr 出力のリダイレクト

アプリケーション開発時に、開発者は、USEROUTPUTCLASS オプションを使用して、CICS 領域で開発者固有の JVM stdout および stderr 出力を分離し、開発者が選択した識別可能な宛先に送信できます。Javaクラスを使用して出力をリダイレクトし、出力レコードにタイム・スタンプとヘッダーを追加できます。ただし、この方法ではダンプ出力を代行受信することはできません。

USEROUTPUTCLASS オプションを指定すると、JVM のパフォーマンスに悪影響が出ます。実稼働環境で最高のパフォーマンスを発揮するには、このオプションは使用しないでください。

アプリケーションまたはシステム・コードのどちらかによって System.out() または System.err() に書き込まれた出力は、出力リダイレクト・クラスによってリダイレクトできます。JVM プロファイルの STDOUT オプションと STDERR オプションで指定された z/OS UNIX ファイルは、JVM で発行されるいくつかのメッセージに引き続き使用されます。または、USEROUTPUTCLASS オプションで指定されたクラスが目的の宛先にデータを書き込めない場合にも使用されます。したがって、これらのファイルに適切なファイル名を指定する必要があります。

USEROUTPUTCLASS オプションを使用するには、適当な Java クラスの名前を指定して、JVM プロファイルで USEROUTPUTCLASS=[java class] を指定してください このクラスは java.io.OutputStream を拡張します。提供されたサンプル JVM プロファイルには、コメント化されたオプション

USEROUTPUTCLASS=com.ibm.cics.samples.SJMergedStream が含まれています。このオプションは、提供されたサンプル・クラスを指定します。com.ibm.cics.samples.SJMergedStream クラスを使用して JVM からの出力をそのプロファイルで処理するには、このオプションをアンコメントしてください。また、CICS は代わりのサンプル Java クラス com.ibm.cics.samples.SJTaskStream も提供します。

提供されたユーザー出力クラスのソースはサンプルとして提供されているので、必要に応じてそれらのクラスを変更するか、サンプルに基づいて独自のクラスを作成することができます。

プールされた JVM の場合、使用しているクラスが、JVM プロファイルの該当するクラスパスのディレクトリーに存在していなければなりません。提供されたサンプル・クラスは、該当するクラスパスに自動的に組み込まれます。そのため、JVM プロファイルで指定する必要はありません。独自の出力リダイレクト・クラスを提供する場合は、USEROUTPUTCLASS オプションを指定した JVM プロファイルで、CLASSPATH_SUFFIX オプションを使用して標準クラスパスにディレクトリーを追加してください。

JVM サーバーの場合、クラスパスを指定する必要はありません。ただし、OSGi フレームワークで出力リダイレクト・クラスを実行するには、そのクラスを OSGi バンドルとしてパッケージする必要があります。詳しくは、150 ページの『JVM stdout および stderr 出力をリダイレクトするための Java クラスの作成』を参照してください。

CICS 提供のサンプル・クラス com.ibm.cics.samples.SJMergedStream および com.ibm.cics.samples.SJTaskStream

初期プロセス・スレッド (IPT) で実行され、CICS 要求を行うことができる Java アプリケーションの場合、代行受信された JVM からの出力は一時データ・キューに書き込まれ、タイム・スタンプ、タスク ID とトランザクション ID、およびプログラム名を追加できます。これにより、複数の JVM からの出力が入っている、マージされたログ・ファイルを作成できます。このログ・ファイルを使用すると、JVM のアクティビティーを CICS のアクティビティーと相互に関連付けることができます。CICS 提供のサンプル・クラス com.ibm.cics.samples.SJMergedStream は、このようにマージされたログ・ファイルを作成するためにセットアップされます。

com.ibm.cics.samples.SJMergedStream クラスは、JVM からの出力を一時データ・キュー CSJO (stdout 出力の場合) および CSJE (stderr 出力および内部メッセージの場合) に送信します。これらの一時データ・キューはグループ DFHDCTG で提供され、CSSL に間接的に送信されますが、必要に応じて再定義することができます。

特に、JVM によって発行されるメッセージの長さが変わる可能性があり、CSSL キューの最大レコード長 (133 バイト) が、表示される一部のメッセージを収容するのに十分でない場合があることに注意してください。この状態が起きる場合、サンプルの出力リダイレクト・クラスはエラー・メッセージを発行し、そのメッセージの本文が影響を受ける可能性があります。

133 バイトより長いメッセージを JVM から受信することが分かる場合は、CSJO と CSJE を別々の一時データ・キューとして再定義する必要があります。それらのキューを区画外宛先にして、キューのレコード長を増やしてください。そのキューを物理データ・セットまたはシステム出力データ・セットに割り振ることができます。この場合、システム出力データ・セットの方が便利な場合があります。これは、出力を表示するためにキューをクローズする必要がないからです。一時データ・キューの定義方法については 「Resource Definition Guide」の『TDQUEUE リソース』 で説明しています。 CSJO と CSJE を再定義する場合は、グループ DFHDCTG で定義される一時データ・キューと同じように、コールド・スタート時にできるだけ早くそれらのキューがインストールされるようにしてください。

一時データ・キュー CSJO および CSJE にアクセスできない場合、出力は、z/OS UNIX ファイル */work_dir/applid*/stdout/CSJO および */work_dir/applid*/stderr/CSJE に書き込まれます。ここで、*work_dir* は JVM プロファイルの WORK_DIR オプションで指定されたディレクトリー、*applid* は CICS 領域に関連したアプリケーション ID です。これらのファイルが使用不可である場合、出力は、JVM プロファイルの STDOUT および STDERR オプションで指定された z/OS UNIX ファイルに書き込まれます。

このクラスは、出力をリダイレクトするほかに、日付、時刻、アプリケーション ID、TRANSID、タスク番号およびプログラム名を含むヘッダーを各レコードに追加します。その結果、JVM 出力用とエラー・メッセージ用に 2 つのマージされたログ・ファイルが作成されます。これらのログ・ファイルでは、出力とメッセージの送信元を容易に特定できます。

初期プロセス・スレッド (IPT) 以外のスレッドで実行され、CICS 要求を行うことができない Java アプリケーションの場合、JVM からの出力は、CICS 機能を使用してリダイレクトできません。com.ibm.cics.samples.SJMergedStream クラスは、引き続き出力を代行受信し、各レコードにヘッダーを追加します。この出力は次に、上記で説明されているように z/OS UNIX ファイル */work_dir/applid*/stdout/CSJO および */work_dir/applid*/stderr/CSJE に書き込まれます。これらのファイルが使用不可である場合は、JVM プロファイルの STDOUT および STDERR オプションで指定された z/OS UNIX ファイルに書き込まれます。

JVM 出力用にマージされたログ・ファイルを作成する代わりに、単一のタスクからの出力を z/OS UNIX ファイルに送信し、タイム・スタンプとヘッダーを追加して、単一のタスクに固有の出力ストリームを提供することができます。CICS 提供のサンプル・クラス com.ibm.cics.samples.SJTaskStream は、これを行うためにセットアップされます。このクラスは、タスクごとの出力を 2 つの z/OS UNIX ファイルに送信します。1 つは stdout 出力用であり、もう 1 つは stderr 出力用です。これらのファイルには、タスク番号を使用して固有の名前が付けられます (YYYYMMDD.task.*tasknumber* の形式で)。これらの z/OS UNIX ファイルは、stdout 出力の場合は stdout ディレクトリーに保管され、stderr 出力の場合は stderr ディレクトリーに保管されます。このプロセスは、IPT で実行される Java アプリケーションと、その他のスレッドで実行される Java アプリケーションの両方で同じです。

提供されたサンプル出力リダイレクト・クラスでエラーが検出される場合、このエラーを報告するエラー・メッセージが 1 つ以上発行されます。出力メッセージの処理中にエラーが発生した場合、エラー・メッセージは System.err に送信され、リダ

イレクトの対象になります。しかし、エラー・メッセージの処理中にエラーが発生した場合は、JVM プロファイルの STDERR オプションで指定されたファイルに、新しいエラー・メッセージが送信されます。これにより、Java クラスの再帰的ループが避けられます。これらのクラスは、呼び出し側の Java プログラムに例外を戻しません。

これらのクラスは、/usr/lpp/cicsts/cicsts42/lib ディレクトリーにあるファイル com.ibm.cics.samples.jar に出荷時に入っています。ここで、/usr/lpp/cicsts/cicsts42 は、z/OS UNIX 上の CICS ファイルのインストール・ディレクトリーです。これらのクラスのソースもサンプルとして提供されているので、必要に応じてそれらのクラスを変更するか、サンプルに基づいて独自のクラスを作成することができます。詳しくは、150 ページの『JVM stdout および stderr 出力をリダイレクトするための Java クラスの作成』を参照してください。

Java ダンプ・オプションの制御

JVM プロファイルの JAVA_DUMP_OPTS オプションは、JVM の Java ダンプ・オプションを指定します。

このオプションを使用すると、優先 Java ダンプ・オプションを設定できます。

Java ダンプ・オプションに関する情報は、Java Diagnostics Guide に記載されています。

JVM サーバーの OSGi ログ・ファイルの管理

OSGi フレームワークは、JVM サーバーの作業ディレクトリーにある 1 組のログ・ファイルにエラーを書き込みます。ご使用の環境にデフォルトが適切でない場合は、JVM サーバーごとにログ・ファイルの数とサイズを管理できます。

このタスクについて

OSGi フレームワークは、zFS の *$WORK_DIR*/*applid*/*jvmserver*/configuration ディレクトリーにあるログ・ファイルにエラーを書き込みます。ここで、*$WORK_DIR* は JVM サーバーの作業ディレクトリー、*applid* は CICS APPLID、*jvmserver* は JVMSERVER リソースの名前です。OSGi フレームワークは、サイズが 1000 KB に達するまでログ・ファイルに書き込み続けます。その後、OSGi フレームワークは、さらにエラー・メッセージを書き出すために別のログ・ファイルを作成します。そのディレクトリーに最大 10 個のログ・ファイルを保持できます。10 個目のログ・ファイルが満杯になると、OSGi フレームワークは最も古いログ・ファイルに上書きします。したがって、各 JVM サーバーでは、zFS のログ・ファイルに最大 10,000 KB のストレージを割り振ることができます。

ファイル数とストレージ使用量を増減するために、OSGi フレームワークで使用されるログ・ファイルの数とサイズを変更するオプションを JVM プロファイルに追加できます。

手順

- ログ・ファイルの最大数を変更するには、**eclipse.log.backup.max** パラメーターを JVM プロファイルに追加します。

- 各ログ・ファイルの最大サイズを変更するには、**eclipse.log.size.max** パラメーターを JVM プロファイルに追加します。

例

次の例は、2 つのパラメーターが指定された JVM プロファイルを示しています。この例では、OSGi フレームワークは最大 5 つのログ・ファイルを使用でき、各ログ・ファイルの最大サイズは 500 KB です。

```
#Parameters to control the number and size of OSGi logs
#
eclipse.log.backup.max=5
eclipse.log.size.max=500
#
#
```

JVM の CICS コンポーネント・トレース

Java によって作成されるトレースに加えて、CICS は、SJ (JVM) および AP ドメインで、 トレース・レベル 0、1、および 2 において、いくつかの標準トレース・ポイントを提供します。これらのトレース・ポイントは、CICS が JVM サーバーおよびプールされた JVM のセットアップと管理を行う際に取るアクションをトレースします。

CETR トランザクションを使用して、SJ および AP ドメイン・トレース・ポイントをレベル 0、1 および 2 で活動化できます。SJ ドメインのすべての標準トレース・ポイントの詳細については、Trace Entries の JVM ドメインのトレース・ポイントを参照してください。

JVM サーバーの SJ および AP コンポーネント・トレース

JVM サーバーの SJ コンポーネントは、JVM サーバーの開始とシャットダウンをトレースします。JVM サーバーのライフサイクル操作は、内部トレース・テーブルにトレースされます。JVM ランチャー、JVM、および OSGi フレームワークのライフサイクル操作は、zFS のファイルにトレースされます。さらに、AP コンポーネントは、JVM サーバーで実行中のトランザクションを同じトレース・ファイルにトレースします。例えば、OSGi フレームワークのイベントは、次のようにトレース・ファイルに書き込まれます。

- トレース・レベル 0 で、OSGi フレームワークはエラーをトレース・ファイルに書き出します。
- トレース・レベル 1 で、OSGi フレームワークは情報、警告、およびエラーをトレース・ファイルに書き出します。
- トレース・レベル 2 で、OSGi フレームワークはデバッグ、情報、警告、およびエラーをトレース・ファイルに書き出します。

AP コンポーネント・トレースをオンに切り替えると、OSGi フレームワークに着信する次の要求がトレースされます。

プールされた JVM の SJ コンポーネント・トレース

プールされた JVM の SJ ドメイン・トレースは、プールされた JVM の開始と管理に関連した CICS アクションをトレースします。

- トレース・レベル 0 で、CICS は異常なイベントとエラーをトレースします。オフに切り替えることができない CICS 例外トレースとは異なり、JVM レベル 0 のトレースは通常はオフに切り替えられます。
- トレース・レベル 1 および 2 で、さらに深いレベルの JVM トレースを取得できます。JVM トレース・ポイントのレベルはレベル 9 まで達し、コンポーネントの詳細を提供します。あるレベルで内部 JVM トレースを活動化すると、そのレベルより上のすべてのレベルのトレースも使用可能になります。例えば、トランザクションのレベル 1 トレースを活動化すると、そのトランザクションのレベル 0 トレースも受け取ります。

さらに、プールされた JVM の内部トレース機能を制御するために追加のトレース・レベルも使用できます。2 よりも上のレベルを選択するには、**JVMxxxxTRACE** システム初期設定パラメーターを変更してください。例えば、レベル 5 トレースを **JVMLEVEL5TRACE** として指定できます。複数のトレース・ポイント・レベルを使用するより複雑な仕様を JVM トレース用に作成したい場合、または仕様でトレース・ポイント・レベルをまったく使用したくない場合は、**JVMUSERTRACE** パラメーターを使用して、独自のトレース・オプション・ストリングを作成してください。

JVM サーバーのトレースの活動化と管理

SJ および AP コンポーネントのトレースをオンにすると、JVM サーバーのトレースを活動化できます。少量のトレースは内部トレース・テーブルに書き込まれますが、大部分のトレースは、JVM サーバーごとに zFS の固有ファイルに書き込まれます。このファイルはラップしないので、そのサイズを zFS で管理する必要があります。

このタスクについて

JVM サーバーのトレースは、補助トレースも GTF トレースも使用しません。代わりに、JVM サーバーごとに固有に名前が指定される zFS 内のファイルにトレースが書き込まれます。デフォルトのファイル名の形式は *applid.jvmserver.*dfhjvmtrc です。このファイルは、JVMSERVER リソースを使用可能にするときに JVM の作業ディレクトリーに CICS によって作成されます。JVM プロファイルでトレース・ファイルの名前と場所を変更できます。JVM サーバーの実行時にトレース・ファイルを削除または名前変更すると、CICS はそのファイルを再作成せず、トレースが別の宛先に書き込まれることもありません。

手順

1. CETR トランザクションを使用して、JVM サーバーのトレースを活動化します。 JVM サーバーのトレースを作成するには、2 つのコンポーネントを使用できます。
   - JVM サーバーの開始と停止のために CICS が取るアクションをトレースするには、SJ コンポーネントを選択します。CICS は、JVM サーバーの開始時に zFS のトレース・ファイルに書き込み、JVM サーバーのシャットダウン時には内部トレース・テーブルに書き込みます。
   - JVM サーバーの内部で実行されるトランザクションをトレースするには、AP コンポーネントを選択します。このオプションを選択すると、CICS は zFS のトレース・ファイルに書き込みます。

2. SJ および AP コンポーネントのトレース・レベルを設定します。
   - SJ および AP のレベル 0 は、例外のみのトレースを作成します。例えば、JVM サーバーの初期化時のエラーや、OSGi フレームワーク内の問題です。
   - SJ および AP のレベル 1 は、追加のトレース情報を作成します。例えば、OSGi フレームワーク内の警告メッセージや情報メッセージです。
   - SJ および AP のレベル 2 は、デバッグ・トレース情報を作成します。これは、JVM サーバー処理に関するより詳細な情報を提供します。

   CICS は zFS のトレース・ファイルにトレースを書き出します。
3. *applid.jvmserver.*dfhjvmtrc ファイルのトレースの結果を表示します。 各トレース項目には、日時のタイム・スタンプがあります。JVMTRACE プロファイル・オプションを使用して、このトレース・ファイルの名前と場所を変更できます。
4. ファイルのサイズを管理するために、古い項目を削除できます。 JVMSERVER リソースを使用不可にする場合、ファイルを削除するか、または情報を別々に保持したい場合はファイルの名前を変更することができます。JVMSERVER リソースを使用可能にすると、CICS は、トレース・ファイルが既に存在する場合はそのファイルにトレース・エントリーを追加し、トレース・ファイルがない場合は、zFS にファイルを作成します。

プールされた JVM 用のトレースの定義および活動化

プールされた Java 仮想マシン (JVM) は、独自のトレース・ポイントを作成します。プールされた JVM の内部トレース機能は、CICS によって提供されるインターフェースを使用して制御できます。 CICS 環境におけるプールされた JVM のトレース・ポイントは、CICS トレースとして出力されます。

このタスクについて

SJ ドメインは、トレース・レベル 29 から 32 を使用して JVM の内部トレース機能を制御します。これらのレベルは、プールされた JVM のレベル 0 トレース、プールされた JVM のレベル 1 トレース、プールされた JVM のレベル 2 トレース、およびプールされた JVM のユーザー・トレースの CICS オプションに対応します。

CICS で提供されるデフォルトの JVM トレース・オプションは、JVM のレベル 0、レベル 1 およびレベル 2 のトレース・ポイント・レベルにマップされます。JVM ユーザー・トレース・オプションは、さらに深いレベルのトレースまたは複雑なトレース・オプションの指定に使用できます。

JVM トレース・オプションは、「フリー・フォーム」の 240 文字フィールドを使用して定義されます。以下の一部またはすべてのパラメーターを指定できます。

- トレース・レベル。
- JVM サブコンポーネント (CICS ドメインのような機能領域) であるコンポーネント。
- トレース・ポイントのタイプまたはグループ。
- トレース・ポイント ID。

選択されたトレース・レベルで特定の項目を組み込むか、除外したい場合は、JVM レベル 0 トレース、JVM レベル 1 トレース、および JVM レベル 2 トレースの CICS 指定にさらにパラメーターを追加できます。より深いレベルのトレースまたは複雑なトレース・オプションを指定したい場合は、JVM ユーザー・トレース・オプションを使用して、適当なパラメーターを含むトレース・オプション・ストリングを作成してください。トレース・ポイント・レベルの指定は、トレース・ポイント ID によって明示的に指定されるトレース・ポイントには適用されないことに注意してください。つまり、任意のレベルで分類されるトレース・ポイント ID を、CICS のいずれかの JVM トレース・オプションに追加できます。トレース・オプションによって使用可能になるトレース・ポイント・レベルに関係なく、トレース・ポイント ID は、そのオプションがアクティブになるときに提供されます。

Java Diagnostics Guide における Java アプリケーションおよび JVM のトレースに関する情報では、可能なトレース・レベル、コンポーネント、トレース・ポイント・タイプ、およびトレース・ポイント・グループをリストしています。これらのトレース・パラメーターは、使用している IBM 64-bit SDK for z/OS, Java テクノロジー・エディション のバージョンによって異なります。また、バージョンの存続期間中に変更される場合もあるので、該当するバージョンの「*Diagnostics Guide*」で最新情報を確認する必要があります。

IBM 64-bit SDK for z/OS, Java テクノロジー・エディション で提供されるトレース・フォーマット・ファイルは、各 JVM トレース・ポイントをその ID と一緒にリストします。バージョン 6.0.1 では、このファイルは `J9TraceFormat.dat` と呼ばれます。このファイルを使用すると、個別の JVM トレース・ポイントを識別できます。このファイルは予告なく変更される場合があります。バージョン番号はファイルの最初の行として含まれ、ファイルが変更されると更新されます。このファイルは、Java のインストール・ディレクトリーで見つけることができます。

JVM トレースによって大量の出力が作成される可能性があるので、JVM トレースをすべてのトランザクション用にグローバルにオンにするのではなく、通常は、特殊なトランザクション用に活動化する必要があります。トランザクションに対してトレース・オプションをアクティブにすると、CICS は、そのトランザクションが JVM の使用を開始する時点でトレース・オプションを JVM に渡します。CICS SJ ドメイン・レベル 2 のトレース・ポイント SJ 052E は、JVM に渡されたオプション・ストリングを示します。トレース・オプションは、トランザクションが JVM を使用する期間のみ適用されます。

手順

- CICS 領域内のすべての JVM にデフォルトの JVM トレース・オプションを設定するには、CICS システム初期設定パラメーター **JVMLEVEL0TRACE**、**JVMLEVEL1TRACE**、**JVMLEVEL2TRACE**、および **JVMUSERTRACE** を使用できます。 これらのパラメーターは CICS の開始時のみに提供できます。DFHSIT マクロで定義することはできません。CETR トランザクションを使用すると、これらのオプションを表示し、変更することができます。これらのパラメーターは、JVM トレースを活動化しません。デフォルトの JVM トレース・オプションを設定するだけです。
- CICS の実行中に JVM トレース・オプションを定義または変更するには、以下のいずれかの方式を使用します。

1. CETR トランザクションで「JVM Trace Options」画面を使用します。 トレース・オプション・ストリングを指定し、各トレース・レベルが適用されるのが標準トレースであるか、特殊トレースであるか、その両方であるかを指定できます。詳しくは、CICS Supplied Transactions の CETR - トレース制御を参照してください。
2. **EXEC CICS INQUIRE JVMPOOL** および **EXEC CICS SET JVMPOOL** コマンドを使用します。 **INQUIRE JVMPOOL** コマンドは、JVM プールに対して設定した JVM トレース・オプションを表示し、**SET JVMPOOL** コマンドはそれらのオプションを変更します。JVM トレース・オプションは、CEMT でこれらのコマンドに相当するコマンドでは使用できません。

- JVM トレースを活動化するために、次のいずれかの方式を使用します。必ず、特殊なトランザクションのみに JVM トレースを活動化してください。
  1. CETR トランザクションの「Transaction and Terminal Trace」画面を使用して、適切なトランザクションに対して特殊トレース (または必要に応じて、標準トレース) をオンに切り替えます。 詳しくは、CICS Supplied Transactions の CETR - トレース制御を参照してください。
  2. CICS システム初期設定パラメーター **SPCTRSJ** または **STNTRSJ** を使用して、開始時に JVM トレースを活動化します。 **SPCTRSJ** は特殊トレースに適用され、**STNTRSJ** は標準トレースに適用されます。**STNTRSJ** システム初期設定パラメーターではなく、**SPCTRSJ** システム初期設定パラメーターを使用してください。レベル番号 29 から 32 を指定して、必要な JVM トレースのレベルを活動化します。これらのパラメーターは CICS の開始時のみに提供できます。DFHSIT マクロで定義することはできません。
  3. **EXEC CICS SET TRACETYPE** コマンドを使用して、SJ コンポーネントにトレース・レベル 29 から 32 を設定します。 `STANDARD` オプションではなく、`SPECIAL` オプションを使用してください。

タスクの結果

JVM トレースを活動化すると、結果が SJ (JVM) ドメインの CICS トレース・ポイントとして表示されます。生成される各 JVM トレース・ポイントは、CICS トレース・ポイントのインスタンスとして表示されます。

- SJ 4D02 は、フォーマット済みの JVM トレース情報用に使用されるトレース・ポイントです。
- SJ 4D01 は、CICS によってフォーマットできない JVM トレース・ポイントに使用されます。このトレース・ポイントが頻繁に表示される場合、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション で提供されるトレース・フォーマット・ファイルが SDK インストールの `/lib/` サブディレクトリーに存在することを確認してください。バージョン 6.0.1 では、このファイルは `J9TraceFormat.dat` と呼ばれます。CICS には、JVM トレース・ポイントをフォーマットするためにこのファイルが必要です。

JVM トレース機能が失敗すると、CICS はトレース・ポイント SJ 4D00 を発行します。

次のタスク

Java Diagnostics Guide には、JVM トレースおよび JVM の問題判別に関する詳細情報が記載されています。

CICS に付属のインターフェースの他に、JVM の内部トレース機能を直接使用することもできます。JVM システム・プロパティーは、CICS 環境で JVM のトレース・オプションを設定し、アクティブにする有効な方法です。「*Diagnostics Guide*」には、内部トレース機能の制御に使用できるシステム・プロパティーに関する詳細情報が記載されています。

Java アプリケーションのデバッグ

CICS における JVM は、Java 2 Platform で提供される標準デバッグ・メカニズムである Java Platform Debugger Architecture (JPDA) をサポートします。このアーキテクチャーは、リモート・デバッガーと JVM との接続を可能にする 1 組の API を提供します。

このタスクについて

JDPA をサポートする任意のツールを使用して、CICS で実行される Java アプリケーションをデバッグすることができます。例えば、z/OS 上の Java SDK に付属の Java Debugger (JDB) を使用できます。JPDA リモート・デバッガーを接続するには、JVM プロファイルでいくつかのオプションを設定する必要があります。

CLASSCACHE=YES を指定する JVM プロファイルまたは DFHJVMCD プロファイルにはデバッグを使用可能にしないでください。

手順

1. **-Xdebug** オプションを JVM プロファイルに追加して、JVM をデバッグ・モードで開始します。 JVM プロファイルが複数のアプリケーションで共用される場合、別の JVM プロファイルをデバッグに使用できます。
2. オプション: **-Xrunjdwp** オプションを追加して、デバッガーと JVM 間の接続の詳細を指定します。 このオプションを設定して JVM サーバーをデバッグする場合、suspend=n を指定してください。このオプションでは、領域内のコマンドまたは処理を完了する前に JVM にデバッガーを接続するのを CICS は待機しません。

   デバッガーの接続要件や機能が異なる可能性があるので、デバッガーに付属の資料を参照してください。
3. デバッガーを JVM に接続します。 接続中にエラーが発生する (例えば、正しくない TCP/IP ホストやポート値) 場合、JVM 標準出力ストリームと標準エラー・ストリームにメッセージが書き込まれます。
4. デバッガーを使用して、JVM の初期状態を確認します。例えば、開始したスレッドの ID やロードされたシステム・クラスを確認してください。 JVM は実行を中断します。Java アプリケーションはまだ開始していません。
5. 完全 Java クラス名とソース・コード行番号を指定して、Java アプリケーションの適切なポイントでブレークポイントを設定します。アプリケーション・クラスは通常、この時点でロードされていないため、デバッガーは、クラスがロードさ

れるまでこのブレークポイントの活動化が延期されることを示します。 JVM を CICS ミドルウェア・コードを使用してアプリケーションのブレークポイントまで実行させてください。このポイントで JVM は再度実行を中断します。

6. ロードされたクラスと変数を調べ、追加のブレークポイントを設定して、必要に応じてコードをステップスルーします。
7. デバッグ・セッションを終了します。アプリケーションを最後まで実行させることができます。その時点でデバッガーと CICS JVM 間の接続はクローズします。一部のデバッガーは、JVM の強制終了をサポートします。その結果、異常終了し、CICS システム・コンソールにエラー・メッセージが表示されます。

CICS JVM プラグイン・メカニズム

JVM の標準 JPDA デバッグ・インターフェースに加えて、CICS Java ミドルウェアには CICS 提供の 1 組の代行受信ポイントがあります。これは、開発者がアプリケーションをデバッグするために役立つことがあります。これらの代行受信ポイント (すなわちプラグイン) を使用して、アプリケーション Java コードの実行直前と直後に追加の Java プログラムを挿入できます。

アプリケーションに関する情報 (例えば、クラス名やメソッド名) が、プラグイン・プログラムに提供されます。また、プラグイン・プログラムは JCICS API を使用して、アプリケーションに関する情報を取得することもできます。これらの代行受信ポイントを標準 JPDA インターフェースと一緒に使用すると、追加の CICS 固有のデバッグ機能を提供できます。また、CICS におけるユーザー出口点とほぼ同じように、デバッグ以外の目的にも使用できます。

次の 3 つの Java 出口点があります。

- CICS EJB コンテナー・プラグイン。EJB メソッドが呼び出される直前および直後に呼び出されるメソッドを提供します。
- CICS CORBA プラグイン。CORBA メソッドが呼び出される前後に呼び出されるメソッドを提供します。
- CICS Java ラッパー・プラグイン。Java プログラムが呼び出される直前および直後に呼び出されるメソッドを提供します。

デバッグ・プラグインは、プールされた JVM で使用できます。プラグイン・プログラムを使用して Java アプリケーションをデバッグする場合は、アプリケーションが使用する JVM の標準クラスパス上にあるクラスを指定する必要があります。標準クラスパスは、JVM プロファイルの `CLASSPATH_SUFFIX` オプションで指定されます。詳しくは、10 ページの『JVM におけるクラスおよびクラスパス』を参照してください。プラグイン・プログラムのクラスは、通常のアプリケーションのクラスと同じ方法で追加できます。

プログラミング・インターフェースは次の 2 つの Java インターフェースで構成されます。

- **DebugControl** (フルネーム: `com.ibm.cics.server.debug.DebugControl`) は、ユーザー提供の実装環境に対して可能なメソッド呼び出しを定義します。
- **Plugin** (フルネーム: `com.ibm.cics.server.debug.Plugin`) は、プラグイン実装環境の登録用の汎用インターフェースを提供します。

これらのインターフェースは、com.ibm.cics.server.jar で提供され、Javadoc で文書化されます (詳しくは、 57 ページの『CICS 用 Java クラス・ライブラリー (JCICS)』を参照してください)。

図 9 のコード・フラグメントは、DebugControl インターフェースの実装例を示しています。

```
public interface DebugControl
{
    // called before an application object method or program main is invoked
    public void startDebug(java.lang.String className,java.lang.String methodName);

    // called after an application object method or program main is invoked
    public void stopDebug(java.lang.String className,java.lang.String methodName);

    // called before an application object is deleted
    public void exitDebug();

}
public interface Plugin
{
    // initaliser, called when plugin is registered
    public void init();
}
```

図 9. DebugControl および Plugin インターフェースの定義

226 ページの図 10 のコード・フラグメントは、DebugControl インターフェースと Plugin インターフェースの実装例を示しています。

```
import com.ibm.cics.server.debug.*;

public class SampleCICSDebugPlugin
    implements Plugin, DebugControl
{
    // Implementation of the plugin initialiser
    public void init()
    {
        // This method is called when the CICS Java middleware loads and
        // registers the plugin. It can be used to perform any initialisation
        // required for the debug control implementation.
    }

    // Implementations of the debug control methods
    public void startDebug(java.lang.String className,java.lang.String methodName)
    {
        // This method is called immediately before the application method is
        // invoked. It can be used to start operation of a debugging tool. JCICS
        // calls such as Task.getTask can be used here to obtain further
        // information about the application.
    }

    public void stopDebug(java.lang.String className,java.lang.String methodName)
    {
        // This method is called immediately after the application method is
        // invoked. It can be used to suspend operation of a debugging tool.
    }

    public void exitDebug()
    {
        // This method is called immediately before an application object is
        // deleted. It can be used to terminate operation of a debugging tool.
    }
}
```

図 10. DebugControl および Plugin インターフェースの実装例

デバッグ・プラグインの実装環境をアクティブにするには、JVM の JVM プロパティー・ファイルで次のシステム・プロパティーを 1 つ以上設定してください。

EJB コンテナー・デバッグ・プラグイン

次のシステム・プロパティーを設定すると、提供されたプラグインは、EJB コンテナーの初期化時に CICS EJB サーバー・レイヤーで Java コードによって登録されます。

```
-Dcom.ibm.cics.server.debug.EJBPlugin=<fully qualified classname,
   for example com.ibm.cics.server.debug.SampleCICSDebugPlugin>
```

CORBA デバッグ・プラグイン

次のシステム・プロパティーを設定すると、提供されたプラグインは、ORB の初期化時に CICS ORB で Java コードによって登録されます。

```
-Dcom.ibm.cics.server.debug.CORBAPlugin=<fully qualified classname,
   for example com.ibm.cics.server.debug.SampleCICSDebugPlugin>
```

CICS Java デバッグ・プラグイン

次のシステム・プロパティーを設定すると、提供されたプラグインは、Java プログラムの実行時に JCICS ラッパーで追加 Java コードによって登録されます。

```
-Dcom.ibm.cics.server.debug.WrapperPlugin=<fully qualified classname,
   for example com.ibm.cics.server.debug.SampleCICSDebugPlugin>
```

Java アプリケーションが実行されるときに、複数のプラグイン・インターフェースを起動できます。例えば、3 つのすべてのインターフェースについてプラグイン実装環境が登録され、エンタープライズ Bean メソッドが実行される場合、JCICS ラッパー、CORBA、および EJB プラグインが順に起動されます。

第 9 章 安定した Java テクノロジー

CORBA、IIOP、およびエンタープライズ Bean は、CICS で安定した Java テクノロジーです。新しいアプリケーションの開発にこれらのテクノロジーを使用しないでください。

ステートレス CORBA オブジェクト

クライアントの観点から見ると、CICS ORB を使用して呼び出されるステートレス CORBA オブジェクトは、単にメソッドの集合 (つまり、ステートレス・オブジェクト) にすぎません。

各リモート・メソッドは、既存の CICS プログラムへの 1 つ以上の CICS API 呼び出し (プログラム・リンク呼び出しを含む) を行うことができる 1 つのロジックを表します。CICS ステートレス CORBA オブジェクトは CICS JVM で実行されます。リモート・メソッドの終わりに、オブジェクトの状態は使用できなくなります。

CICS の継続 JVM で実行されるすべての Java プログラムと同様に、CORBA オブジェクトによって生じるすべての静的状態は、後のタスクでそれ以降に検索するために JVM 内で持続します。ただし、CORBA クライアントと CICS JVM 間には親和性がないため、同じソケットを使用する 2 つの後続 CORBA 要求が同一 JVM で (または同じ CICS 領域であっても) 処理される保証はありません。つまり、以前に初期化された静的状態が使用可能かどうかに依存することはできません。

したがって、どのリモート・メソッドにも、作業を完了できるようにするために、パラメーター・リストで十分な情報が渡されなければなりません。実装クラスの検出に使用されるオブジェクト参照 (オブジェクト・タイプを除く) を介してサーバー ORB に渡される情報はありません。ただし、オブジェクトのメソッドは、呼び出し間でアプリケーション管理のデータ・ストレージに状態を保管する場合があります。保管された状態を取り出せるように、十分な情報がパラメーターとして後続メソッドに確実に渡されるようにする必要があります。

CORBA オブジェクトは、同じ CorbaServer または異なる CorbaServer で実行中のエンタープライズ Bean への呼び出しを始めとして、アウトバウンド IIOP 呼び出しを行うことができます。CORBA オブジェクトは、リモート IIOP メソッドのパラメーターとして、それ自体への参照を渡すことさえできます。これは、**コールバック参照**と呼ばれます。ただし、ターゲット・オブジェクトがコールバック参照を使用して、最初の CORBA オブジェクトを呼び出す場合、この新しい要求は新しい JVM で処理されます。したがって、元の JVM からどの状態にもアクセスできません。

メソッドの呼び出しは、オブジェクト・トランザクション・サービス (OTS) **分散トランザクション**に関与する場合があります。クライアントが **OTS トランザクション**の有効範囲内で IIOP アプリケーションを呼び出すと、その OTS トランザクションに関する情報が、IIOP 呼び出しで追加パラメーターとして流れます。ターゲッ

トのステートレス CORBA オブジェクトが CosTransactions::TransactionalObject を実装すると、そのオブジェクトはトランザクションとして扱われます。

ステートレス CORBA オブジェクトの開発

ステートレス CORBA オブジェクトは、IIOP プロトコルを使用してクライアント・アプリケーションと通信する Java サーバー・アプリケーションです。リモート・メソッドの連続するクライアント呼び出し間でオブジェクト属性の状態は維持されません。状態は、各リモート・メソッド呼び出しの開始時に初期化され、明示的なパラメーターで参照されます。

注: リモート・メソッド とは、リモート・クライアントから呼び出し可能なメソッドを意味します。つまり、リモート・クライアントからアクセスできない内部メソッドではなく、オブジェクトの (おそらく複数の) リモート・インターフェースのいずれかの一部として公開されるか、またはオブジェクトに対して IDL で宣言されるパブリック・メソッドです。

サーバー・プログラミング・モデルでは、各メソッドは 1 つのサブルーチンです。渡されるパラメーターを使用すると、既存の任意のデータベースまたはアプリケーションから一時的な状態を確立したり、ビジネス・ロジックを実行したり、既存のデータベースまたはアプリケーションにデータを保管したり、サブルーチンが戻るときに結果を戻したり、例外をスローしたりすることができます。ステートレス CORBA オブジェクトのリモート・メソッド (つまり、リモート・クライアントによって呼び出されるリモート・メソッド) は、ローカル側で相互に呼び出したり、またはオブジェクトの一時状態を失うことなく、非リモート・メソッドを呼び出したりすることができます。一時状態は、クライアントが開始したリモート・メソッド要求の終わりに、すなわちクライアントの要求に対する応答が送信されたときのみに破棄されます。

次の 2 とおりの方法のどちらかを使用して、ステートレス CORBA アプリケーションを開発できます。

1. 標準的な CORBA 開発スタイルを使用します。このスタイルでは、アプリケーション・インターフェースがインターフェース定義言語 (IDL) で定義されてから、アプリケーションがそのインターフェースに合わせてコーディングされます。この方法については、以下のセクションで説明しています。
2. 標準的な Java 開発スタイルを使用します。このスタイルでは、Java リモート・メソッド呼び出し (RMI) アプリケーションが開発され、オプションとして後で IDL が生成されます。この方法は RMI-IIOP と呼ばれます。これについては、239 ページの『RMI-IIOP ステートレス CORBA アプリケーションの開発』 で説明しています。

最初の (CORBA スタイルの) 方法を使用してステートレス CORBA オブジェクトを開発するには、以下のステップを実行する必要があります。

1. インターフェース定義言語 (IDL) を使用して、オブジェクトのインターフェースとオペレーションを定義します。
2. IDL に対して IDL-to-Java コンパイラー (IDLJ) を実行して、オブジェクトのスタブ・クラスとスケルトン・クラスを生成します。
3. 生成されたスタブ・クラスを使用してサーバーに呼び出しを行うクライアント・アプリケーションを作成します。

4. 生成された基本スケルトン・クラスを拡張するサーバー・アプリケーション (ステートレス CORBA オブジェクト) を作成します。
5. クライアント・アプリケーションとサーバー・アプリケーションをコンパイルし、パッケージします。
6. サーバーの CICS リソースを定義し、アプリケーションが使用する JVM の JVM プロファイルで標準クラスパスにサーバー・アプリケーションの JAR ファイルを追加します。

2 番目の (Java スタイルの) 方法を使用してステートレス CORBA オブジェクトを開発するには、以下のステップを実行する必要があります。

1. サーバー・アプリケーション (ステートレス CORBA オブジェクト) のリモート・インターフェースを作成します。
2. このリモート・インターフェースを使用してサーバーに呼び出しを行うクライアント・アプリケーションを作成します。
3. リモート・インターフェースを実装するサーバー・アプリケーションを作成します。
4. クライアント・アプリケーションとサーバー・アプリケーションをコンパイルします。
5. リモート・インターフェースとサーバー・アプリケーションに対して Java RMI コンパイラー (RMIC) を実行して、オブジェクトのスタブ・クラスとタイ・クラスを生成します。
6. クライアント・アプリケーションとサーバー・アプリケーションをパッケージします。
7. サーバーの CICS リソースを定義し、アプリケーションが使用する JVM の JVM プロファイルで標準クラスパスにサーバー・アプリケーションの JAR ファイルを追加します。
8. オプションとして、非 Java CORBA クライアントで使用するためにアプリケーションの IDL を作成します。

この 2 つの方法にはそれぞれ、利点と欠点があります。主な相違点の 1 つは、CORBA 方法では、ステートレス CORBA オブジェクトは、生成された基本クラスを拡張しなければならないことです。Java が単一の継承階層のみをサポートすると仮定すると、これは、ステートレス CORBA オブジェクトに適当なクラスを拡張させることができないことを意味します。RMI-IIOP 方法では、ステートレス CORBA オブジェクトに適当な継承階層を使用できます。これは、オブジェクトが特定のインターフェースを実装するだけで済むからです。

CORBA インターフェース名とオペレーション名は、対応する Java 実装環境にマップされます。CICS Java クラス (JCICS) を使用して CICS サービスにアクセスするサーバー実装環境を作成できます。JCICS クラスの詳細については、*JCICS Class Reference* を参照してください。それらのクラスを使用したサーバー・アプリケーションの開発方法については、57 ページの『JCICS を使用した Java プログラミング』を参照してください。

JCICS クラスは、クラス定義から生成される JAVADOC html で詳しく説明されています。これは、CICS インフォメーション・センターの *JCICS Class Reference* で入手できます。

相互運用オブジェクト参照 (IOR) の取得

実行時にサーバー・オブジェクトの位置を確認するには、そのサーバー・オブジェクトへの参照がクライアント・アプリケーションで必要です。

この参照は**相互運用オブジェクト参照 (IOR)** と呼ばれます。 IOR は、クライアント ORB が IOR をデコードしてリモート・サーバー・オブジェクトの位置を確認できるように、特定の方法でエンコードされたテキスト・ストリングです。これには、以下を可能にする十分な情報が入っています。

- 要求を正しいサーバーに送信できるようにする情報 (ホスト、ポート番号)
- オブジェクトを検出または作成できるようにする情報 (クラス名、インスタンス・データ)

IOR はサーバー・メソッドによって戻される場合がありますが、初期 IOR の作成にはファクトリー・クラスが必要です。CICS は、この目的に CORBA ライフサイクル・サービス (CosLifeCycle) の GenericFactory クラスを使用します。クライアント・アプリケーションはこの GenericFactory を使用して、実行時に必要なステートレス CORBA オブジェクトごとに IOR を作成できます。ただし、GenericFactory 自体がステートレス CORBA オブジェクトであるため、クライアント・アプリケーションは、ターゲット・オブジェクトの IOR を作成する前に*それ自体の* IOR が必要です。

GenericFactory クラスのストリング変換された IOR を公開するには、PERFORM CORBASERVER PUBLISH コマンドを使用してください。すると、GenericFactory IOR が作成され、シェルフ (CorbaServer に関連した z/OS UNIX ディレクトリー) に保管され、ネーム・サーバーに公開されます。GenericFactory IOR をクライアント・アプリケーションが使用すると、この CorbaServer (およびこの CorbaServer のみ) について存在する任意のステートレス CORBA オブジェクト用の IOR を作成できます。この IOR は、`genfac.ior` という名前で公開されます。クライアントが実行時に GenericFactory IOR の位置をどのように確認するかは、アプリケーション体系により決まります。IOR は、JNDI ネーム・スペース内の既知の場所から取り出されたり、ローカル側でクライアント・マシンに保持されたり、その他のプロセスでアクセスされたりする場合があります。

IOR を公開するには、**CEMT PERFORM CORBASERVER** コマンドを使用するか、CICS アプリケーションから **EXEC CICS PERFORM CORBASERVER** コマンドを発行することができます。

`genfac.ior` ファイルは、CORBASERVER のシェルフ・ディレクトリーに書き込まれます。

/shelf/applid/corbaserver/

ここで、

shelf CORBASERVER リソース定義で指定された SHELF ディレクトリー名です。デフォルトで `/var/cicsts/` になります。

applid CICS 領域に関連したアプリケーション ID です。

corbaserver
CORBASERVER リソース名です。

FTP を使用して、シェルフからクライアント・ワークステーションに (ASCII モードで) IOR をダウンロードできます。または、クライアントが JNDI インターフェースを使用して、ネーム・サーバーから IOR を取得することもできます。

オブジェクトにステートレスの性質があるため、クライアントでクラスの複数のインスタンスを作成してもあまり意味がありません。クライアントがオブジェクトのインスタンス (例えば、bankaccountfacilitator) を作成した後、同じオブジェクトを使用して、Mr X のアカウントと Mr Y のアカウントの両方にアクセスできます。アカウント番号は、各メソッドの入力パラメーターです。

注: この例では、オブジェクトに bankaccountfacilitator を呼び出したので、任意のアカウントでアクションを実行できます。オブジェクトに bankaccount を呼び出した場合、インスタンスは常に Mr X のアカウントを表したことを示します。

インターフェース定義言語 (IDL) の作成

CORBA 開発スタイルを使用してステートレス CORBA オブジェクト・アプリケーションを作成する場合、サーバーの実装環境がサポートするインターフェースの定義を含む OMG IDL ファイルを作成する必要があります。

注: このセクションでは、CORBA 開発スタイルを使用してステートレス CORBA オブジェクト・アプリケーションを作成しようとしている (RMI-IIOP 方法ではなく、230 ページの『ステートレス CORBA オブジェクトの開発』の方法 1) ことを前提としています。RMI-IIOP 方法は、239 ページの『RMI-IIOP ステートレス CORBA アプリケーションの開発』で説明されています。

OMG IDL ファイルは、クライアントが要求を行うのに使用でき、かつサーバーが所定オブジェクトの実装環境に提供しなければならない、データ型、オペレーション、およびオブジェクトを記述します。

IDL の作成については、OMG Web サイト (http://www.omg.org/) から取得可能な OMG 資料「*Common Object Broker: Architecture and Specification*」を参照してください。

IDL 定義は IDL-to-Java コンパイラー (場合によっては「パーサー」または「ジェネレーター」と呼ばれます) で処理します。サーバー・サイドのスケルトンおよびヘルパー・クラスを生成するには、サーバー環境で提供されたコンパイラーを使用する必要があります。また、クライアント・サイドのスタブ (場合によっては「プロキシー」と呼ばれます) およびヘルパー・クラスを生成するには、クライアント環境で提供されるコンパイラーを使用する必要があります。CICS で使用するのに適したスケルトン・クラスは、任意の IBM Java 2 SDK に付属の IDLJ コンパイラーを使用して作成できます。IBM 以外の IDLJ コンパイラーを使用する場合、その結果作成されるスケルトン・クラスが、CICS での使用に適している場合と適していない場合があります。疑わしい場合は、CICS で使用される、z/OS に提供されている Java SDK に付属の IDLJ コンパイラーを使用できます。

IBM IDL コンパイラー (IDLJ) によって作成されるスタブ・クラスまたはプロキシー・クラスは、任意の IBM ORB での使用に適しています。別のベンダーからのクライアント・サイド ORB を使用する場合は、その ORB に付属の IDL コンパイラーを使用してください。あるベンダーの ORB 用に生成されたスタブ・クラスを、

別のベンダーの ORB と一緒に使用する場合、結果は不定です。スタブが機能する場合もあれば、機能しない場合もあります。

プロキシーとスケルトンは、ORB がメソッド起動を配布するのに必要な、オブジェクト固有の情報を提供します。

図 11 は、同じ IDL ファイルを使用して、クライアントとサーバーで使用される別々のクラスを生成する方法を示しています。

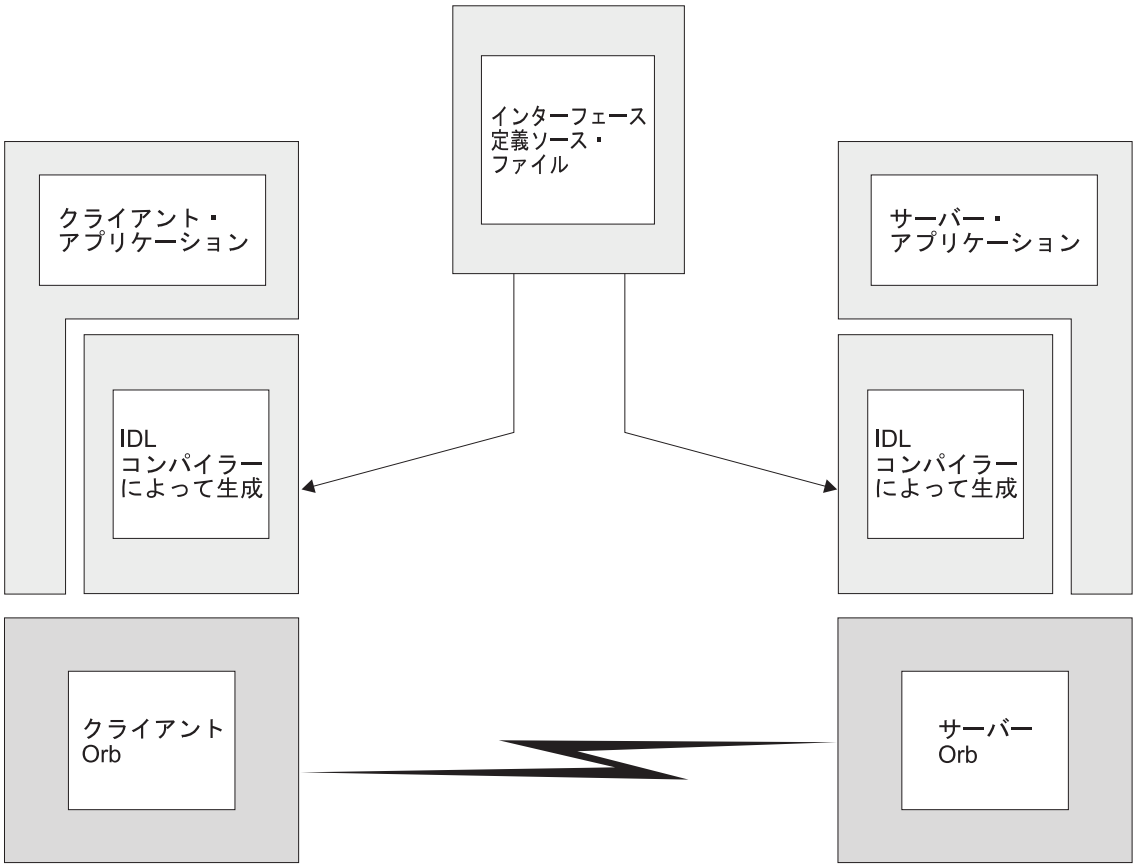


図 11. IDL と生成されたコード

IIOP サーバー・プログラムの開発

このサーバー・プログラムは、Java をサポートする任意のプラットフォームで開発できます。例えば、NT ワークステーション、AIX®、または z/OS の UNIX システム・サービス環境です。

このタスクについて

注: このセクションでは、CORBA 開発スタイルを使用してステートレス CORBA オブジェクト・アプリケーションを作成しようとしている (RMI-IIOP 方法ではなく、 230 ページの『ステートレス CORBA オブジェクトの開発』の方法 1) ことを前提としています。RMI-IIOP 方法は、 239 ページの『RMI-IIOP ステートレス CORBA アプリケーションの開発』で説明されています。

以下のステップが必要です。

手順

1. アプリケーションを形成するインターフェースおよびオペレーションの IDL 定義を作成します。
2. Java 2 SDK に含まれている IDL コンパイラー **idlj** コマンドを使用して、CORBA スケルトン・クラスとヘルパー・クラスを生成するように、IDL ファイルをコンパイルします。

   注:

   a. これを行うには、IBM 提供の IDL-to-Java コンパイラーを使用する必要があります。Sun バージョンの Java 2 SDK に付属の IDL-to-Java コンパイラーは、IBM ORB との 100% の互換性がない場合があります。

   b. **idlj** コマンドは、Java ランタイム環境 (JRE) の一部として提供されていません。これが機能するには、ご使用のマシンに完全な SDK がインストールされている必要があります。

   IDL コンパイラーは次のように起動することができます。

   ```
   idlj [options] <idl file>
   ```

   ここで、<idl file> は、IDL 定義が入っているファイルの名前です。[options] は、以下のオプションの任意の組み合わせです。これらのオプションは任意の順序で指定できます。 <idl file> は必須であり、最後に指定されなければなりません。少なくとも **-f** を指定してください。

   例えば、次のようになります。

   ```
   idlj -v -fall myidl.idl
   ```

   また、CICS 互換の実装環境が確実に生成されるように、**-oldImplBase** オプションも指定する必要があります。このオプションを使用しない場合、生成される実装環境では、CICS でサポートされない Portable Object Adapter (POA) が使用されます。例えば、次のようになります。

   ```
   idlj -v -fall -oldImplBase myidl.idl
   ```

   -d<symbol>
   IDL ファイル内の次の行に相当します。 #define <symbol>

   -emitAll
   #included ファイルで検出されるタイプを含めて、すべてのタイプを出力します。

   -f<side>
   出力するバインディングを定義します。<side> は次のいずれかです。

   client CICS には適用されません。

   server 通常の使用に十分なクラスを生成しません。

   all すべてのバインディングを出力します。

   serverTIE
   CICS ではサポートされません。

   allTIE CICS ではサポートされません。

このオプションを指定しなかった場合、**-fclient** を指定したと見なされます。大部分の場合は、**-fall** を使用してください。

-i<include path>
別のディレクトリーを追加します。デフォルトでは、組み込みファイルがないかどうか、現行ディレクトリーがスキャンされます。

-keep 生成しようとするファイルが既に存在する場合は、上書きしないでください。デフォルトで上書きされます。

-oldImplBase
このオプションは必須です。このオプションを省略すると、IDLJ が、Portable Object Adapter (POA) を使用するコードを生成します。POA は CICS ではサポートされません。

-pkgPrefix <t> <pkg>
タイプまたはモジュール <t> が検出されるたびに、生成されるすべてのファイルの <pkg> に置かれていることを確認してください。<t> は、完全修飾 Java スタイル名です。

-v 冗長モード。

3. サーバー実装環境を Java コードで作成します。 IDL コンパイラーは、*interfacename***ImplBase** と呼ばれる抽象クラスを生成します。ご使用のプログラムでこれを拡張する必要があります。このタイプのオブジェクトが Generic Factory によって作成される場合、実装クラスは *_interfacename*Impl と呼ばれなければなりません。この命名規則を使用しないと、GenericFactory は、CORBA オブジェクトへの参照を作成できません。例えば、次のようになります。

```
public class _BankAccountImpl extends _BankAccountImplBase
```

実装クラスは、JCICS API を利用して、従来の CICS サービスと対話することができます。

4. javac コンパイラーまたはそれと同等なコンパイラー (例えば、VisualAge® for Java) を使用して、プログラムおよびステップ 2 からの出力をコンパイルします。 JVM プロファイルの CLASSPATH_SUFFIX オプションを使用して、出力ファイルの場所が CICS 標準クラスパスの終わりに追加されていることを確認してください。

例

この例は、内容の照会と更新が行われる銀行口座を記述しています。この例には、「ステートレス」の制約事項に対応するために、BankAccount のインスタンスを識別するパラメーターがあります。次の IDL はインターフェースとオペレーションを定義します。

```
module bank {

// this interface is used to manage the bank accounts
interface BankAccount {
  exception ACCOUNT_ERROR { long errcode; string message;};

  // query methods
  long querybalance(in long acnum) raises (ACCOUNT_ERROR);
  string queryname(in long acnum) raises (ACCOUNT_ERROR);
  string queryaddress(in long acnum) raises (ACCOUNT_ERROR);

  // setter methods
```

```
    void setbalance(in long acnum, in long balance) raises (ACCOUNT_ERROR);
    void setaddress(in long acnum, in string address) raises (ACCOUNT_ERROR);
};
};
```

上記の IDL のサーバー実装環境は、このタイプのオブジェクトが GenericFactory によって作成される場合は `_BankAccountImpl` と呼ばれなければなりません。また、IDL コンパイラーによって生成される `_BankAccountImplBase` を拡張しなければなりません。これは、Java パッケージ `bank` に含まれています。この実装環境の詳細は、以下で配布されているステートレス CORBA BankAccount サンプル・アプリケーションを参照してください。

```
/usr/lpp/cicsts/<username>/samples/dfjcorb
```

ここで、*username* は、CICS のインストール時に選択できる名前です。デフォルトでは `cicsts42` になります。

この例を使用するには、次のリソースが必要です。

- CICS で、所定のポートで listen するために定義され、インストールされている TCPIPSERVICE リソース。この TCPIPSERVICE は、次のとおりです。
  - IIOP プロトコルを使用するように定義されなければなりません。
  - 要求を受信するために「オープン」状態でなければなりません。
- TCPIPSERVICE で IIOP 要求を処理するように定義されている CORBASERVER リソース。

オプションとして、所定の TRANSID で要求を強制的に処理するために、REQUESTMODEL 定義の追加を選択できます。

IIOP クライアント・プログラムの開発

生成されたスタブ・クラスを使用してサーバーに呼び出しを行うクライアント・アプリケーションを作成します。

このタスクについて

注: このセクションでは、CORBA 開発スタイルを使用してステートレス CORBA オブジェクト・アプリケーションを作成しようとしている (RMI-IIOP 方法ではなく、230 ページの『ステートレス CORBA オブジェクトの開発』の方法 1) ことを前提としています。RMI-IIOP 方法は、239 ページの『RMI-IIOP ステートレス CORBA アプリケーションの開発』で説明されています。

手順

1. (サーバー・アプリケーションの作成に使用したのと同じ IDL ファイルを使用して) クライアント・システムに適した IDL-to-Java コンパイラーで IDL ファイルを処理します。
2. CorbaServer のシェルフ・ディレクトリーから `genfac.ior` を (ASCII モードで) ダウンロードして、GenericFactory へのストリング変換されたオブジェクト参照を取得します。genfac.ior は、CORBASERVER リソースの公開時にこのシェルフ・ディレクトリーで作成されました。 別の方法として、**EXEC CICS PERFORM CORBASERVER PUBLISH** コマンドまたは **CEMT PERFORM CORBASERVER PUBLISH** コマンドを発行する場合、CorbaServer の Generic Factory IOR がネーム・スペース

に公開されるので、JNDI を使用できます。JNDI を使用する計画の場合は、ネーム・サーバーを定義する必要があります。 430 ページの『ネーム・サーバーの定義』を参照してください。IOR は、GenericFactory という名前で、CORBASERVER リソース定義の JNDI 接頭部で識別されるコンテキストにバインドされます。例えば、パス名は次のとおりです。

```
/jndiprefix/GenericFactory
```

「*CICS Resource Definition Guide*」および「*CICS Supplied Transactions*」マニュアルを参照してください。

3. サーバーへの呼び出しを含む、クライアント・プログラムを作成します。初期オブジェクト参照を取得するには、『クライアントの例』に示されているように GenericFactory を使用します。
4. javac または同等のコンパイラーを使用して、クライアント・プログラムおよびステップ 1 からの出力をコンパイルします。

タスクの結果

クライアントの例

次の例では、クライアント・プログラムが GenericFactory サービスを使用して **account** オブジェクトを作成する方法を示しています。クライアントはまず最初に、GenericFactory のプロキシーを作成する必要があります。

CORBA CosLifeCycle および CosNaming モジュールの一部に対する Java バインディングが必要です。クライアント ORB によって提供されない場合、OMG Web サイト (www.omg.org) から入手可能な CORBA サービス IDL から、クライアント ORB の IDL-to-Java コンパイラーを使用して作成できます。別の方法として、以下で提供されている、プリコンパイル済み Java バージョンの IDL を使用できます。

```
/usr/lpp/cicsts/<cicsts42>/lib/omgcos.jar
```

ここで、*cicsts42* は、CICSTS のインストール時に定義した USSDIR インストール・パラメーターに選択した値です。

JAR ファイルはバイナリー・モードでダウンロードされ、クライアントの CLASSPATH 環境項目で使用可能にならなければなりません。

次の例、および提供されたサンプルでは、org.omg.CosNaming および org.omg.CosLifeCycle としてインポートできるバインディングが必要です。

アカウント・オブジェクトを作成するには、クライアントはまず最初に、GenericFactory のプロキシーを作成する必要があります。次の例では、GenericFactory へのストリング変換された参照が、クライアントから使用可能なファイルに存在し、getFactoryIOR() メソッドによって戻されることを前提としています。

```
import java.io.*;
import org.omg.CORBA.*;
import org.omg.CosLifeCycle.*;
import org.omg.CosNaming.*;
public class bankLineModeClient{

//The following method reads the ior from a file and returns it in the string
 String factoryIOR = getFactoryIOR();
```

```
// Turn the stringified reference into the proxy
 org.omg.CORBA.Object genFacRef = orb.string_to_object(factoryIOR);
// narrow to correct interface
 GenericFactory fact = GenericFactoryHelper.narrow(genFacRef);
```

クライアントには Generic Factory があるので、これを使用して account オブジェクトを作成できます。

```
// The Generic factory needs a key, which is a sequence of namecomponents
 NameComponent nc = new NameComponent("bank::BankAccount","object interface");
 NameComponent key[] = {nc};
//The  Generic factory also requires criteria (which it ignores)
 NVP mycriteria[] = {};

//Now create the object
 org.omg.CORBA.Object objRef = fact.create_object(key, mycriteria);
// and narrow to correct interface
BankAccount acctRef = BankAccountHelper.narrow(objRef);
```

クライアントにはオブジェクトがあるので、そのオブジェクトを使用できます。

```
int ac1 = 1234; // Tony's account
int ac2 = 3456; // Lou's account
String name;
String address;
int balance;


try {
  name=acctRef.queryname(ac1);
  System.out.println("a/c num:"+ac1+" name:"+name);
}
catch (exception e) {
  System.err.println("query error");
}
```

注: NVP (Name Value Pair) は、Generic Factory インターフェースに対して CORBA IDL で定義されるデータ型です。

RMI-IIOP ステートレス CORBA アプリケーションの開発

RMI-IIOP 開発スタイルを使用して、ステートレス CORBA オブジェクト・アプリケーションを作成することができます。

これは、前のセクションで説明された CORBA 開発方法ではなく、230 ページの『ステートレス CORBA オブジェクトの開発』の方法 2 で定義された開発スタイルです。

RMI-IIOP 方法には、標準の Java リモート・メソッド呼び出し (RMI) アプリケーションを開発すること、およびこのアプリケーションをデプロイして、そのトランスポート・プロトコルとして IIOP を使用することが含まれます。これは、エンタープライズ Bean で取られる方法です。

注: このセクションでは、特に、RMI-IIOP を使用してステートレス CORBA アプリケーションを開発する方法を説明しています。エンタープライズ Bean は、他のツールを使用してデプロイされます。エンタープライズ Bean のデプロイについては、351 ページの『エンタープライズ Bean のデプロイ』を参照してください。

RMI-IIOP を使用する場合、IDL を使用してインターフェースを定義する必要はありませんが、必要に応じて、後でオプションとして IDL を生成できます。その代わりに、最初に少なくとも 1 つのリモート・インターフェースを定義します。ここでは、「リモート・インターフェース」とは、java.rmi.Remote を拡張する任意の Java インターフェースを意味します。これは、エンタープライズ Bean の「リモート・インターフェース」と同じものではありません。ここで定義した用語を使用すると、エンタープライズ Bean のリモート・インターフェースとそのホーム・インターフェースの両方が「リモート・インターフェース」と見なされます。どちらも、最終的に java.rmi.Remote を拡張するからです。

このリモート・インターフェースは、Java RMI の規則に従うようにコーディングされなければなりません。リモート・インターフェースの例は次のとおりです。

```
package hello;
public interface HelloWorldRMI extends java.rmi.Remote
{
   public String sayHello(String msgFromClient) throws java.rmi.RemoteException;
}
```

上記のインターフェースが定義する、sayHello と呼ばれる単一のメソッドは、パラメーターとして String を取り、String を戻します。このインターフェースのすべてのメソッドは、java.rmi.RemoteException をスローするように定義されなければなりません。

次に、このインターフェースのサーバー・サイドの実装環境を提供する必要があります。例は次のとおりです。

```
package hello;
public class _HelloWorldRMIImpl implements HelloWorldRMI
{
 public String sayHello(String msgFromClient)
 { return "Hello: You said: " + msgFromClient;}
}
```

実装クラスは、既に作成されているインターフェースを実装します。実装クラスに使用される命名規則は _<*interface name*>Impl です。この命名規則が必要なのは、CORBA CosLifeCycle Generic Factory 方法を使用してサーバー・オブジェクトを位置指定する場合です。この命名規則を使用しないと、Generic Factory は、ステートレス CORBA オブジェクトのインスタンスを構成できません。

従来の IDL ベースの開発プロセスよりも勝る RMI-IIOP の利点の 1 つは、基本クラスの拡張が強制されないことです。これは、必要に応じて独自の継承階層の使用を選択できることを意味します。また、単一のサーバー・オブジェクトで複数のリモート・インターフェースを実装することもできます。

上記のクラスはどちらも、javac コンパイラーまたはそれと同等なコンパイラーを使用してコンパイルする必要があります。

次に、このステートレス CORBA オブジェクトにサーバー・サイドの Tie ファイルを作成します。これは、RMI コンパイラー (RMIC) を使用して行われます。IBM Java 2 SDK に付属の RMI コンパイラーを使用する必要があります。Java 2 SDK に付属のバージョンの RMIC を使用する場合、生成される Tie ファイルが CICS ORB を処理することは保証されません。

使用するコマンドは次のとおりです。

```
rmic -iiop hello._HelloWorldRMIImpl
```

RMIC は、サーバー・サイドの実装クラスに対して実行されていることに注意してください。

次に、クライアント・サイドのスタブ・クラスが必要です。これも、RMI コンパイラーを使用して作成されます。クライアント ORB に適した RMI コンパイラーを使用するようにしてください。使用するコマンドは次のとおりです。

```
rmic -iiop hello.HelloWorldRMI
```

RMIC は、リモート・インターフェース・クラスに対して実行されていることに注意してください。

これが完了した後、次のクラスが使用可能でなければなりません。

```
hello\HelloWorldRMI.class              - the remote interface
hello\_HelloWorldRMIImpl.class         - the stateless CORBA object
hello\__HelloWorldRMIImpl_Tie.class    - the RMI-IIOP server side Tie file
hello\_HelloWorldRMI_Stub.class        - the RMI-IIOP client side Stub file
```

次に、クライアント・アプリケーションを作成します。このクライアント・アプリケーションは、CORBA 開発に対する IDL ベースの方法 (237 ページの『IIOP クライアント・プログラムの開発』を参照) を使用して開発されたクライアント・アプリケーションに非常によく似ています。これまでのように、CORBA CosLifeCycle Generic Factory を使用してステートレス CORBA オブジェクトへの参照を見つける必要があります。RMI-IIOP クライアント・アプリケーション例の一部は次のとおりです。

```
ORB orb = ORB.init((String[]) null, (java.util.Properties) null);

// The following method reads the generic factory IOR from a file and returns
// it in the string
String factoryIOR = getFactoryIOR();

// Turn the stringified reference into the proxy
org.omg.CORBA.Object genFacRef = orb.string_to_object(factoryIOR);

// narrow to correct interface
GenericFactory fact = GenericFactoryHelper.narrow(genFacRef);

// The Generic factory needs a key, which is a sequence of namecomponents
NameComponent nc = new NameComponent("hello::HelloWorldRMI","object interface");

//Now create the object
org.omg.CORBA.Object objRef=fact.create_object(new NameComponent[]{nc},
                                               new NVP[] {});

// and narrow to correct interface using the RMI-IIOP narrow operation
HelloWorldRMI remote = (HelloWorldRMI) javax.rmi.PortableRemoteObject.narrow
                       (objRef, HelloWorldRMI.class);

// Invoke the remote method
System.out.println("Received from Server: "+remote.sayHello("Hi!")+"\n");}
```

IDL ベースのクライアント・アプリケーションと同様に、CosLifeCycle クラスを検出するには、ワークステーションおよびクライアント・マシン上の CICS lib z/OS UNIX ディレクトリーからの `omgcos.jar` ファイルが必要です。

残りの手順は、サーバー・サイト・アプリケーションとクライアント・サイド・アプリケーションを JAR ファイルにパッケージし、サーバー・サイド JAR ファイルを標準クラスパスに追加することです。

非 Java ベースの CORBA クライアント・アプリケーションでの使用に適した IDL を RMI-IIOP リモート・インターフェース用に生成したい場合は、次のコマンドを使用してください。

```
rmic -idl hello.HelloWorldRMI
```

スタンドアロン CICS CORBA クライアント・アプリケーション

CICS CORBA サポートは、主として IIOP サーバー・サイド・オブジェクト、すなわちエンタープライズ Bean とステートレス CORBA オブジェクトのサポートを中心にしています。これらのサーバー・サイド・コンポーネントは CICS EJB/CORBA サーバーで実行され、CorbaServer 実行環境では CORBASERVER リソースに相当します。これらのコンポーネントは CICS EJB/CORBA サーバーで実行されるので、豊富な ORB 機能セットを利用できます。

このセクションでは、「スタンドアロン *CICS CORBA* クライアント・アプリケーション」という用語は、次に当てはまる CICS アプリケーションを指します。

1. CORBA クライアント・アプリケーションである
2. JVM=YES が指定されている PROGRAM 定義を使用して、CICS に対して標準 Java アプリケーションとして定義されている
3. new 演算子を使用して ORB インスタンスを作成する
4. CICS CorbaServer 実行環境で実行されない

スタンドアロン CICS CORBA クライアント・アプリケーションは CICS EJB/CORBA サーバーで実行されないため、サーバー・サイド・コンポーネントと同じ品質の CORBA サポートを利用できません。これらのクライアント・アプリケーションから使用可能な ORB は、「JCICS ORB」と呼ばれることがあるクライアント専用 ORB です。この ORB は、ソケットでインバウンド接続を listen することができません。したがって、この ORB によって公開される IOR をサポートできません。同様に、CICS CORBA クライアント・アプリケーションは分散 OTS トランザクションを開始 (または関与) することができません。また、CICS CORBA クライアント・アプリケーションは、アサーション ID 認証にも関与できません。

これらの制限は、CICS サーバー ORB 環境には適用されません。CICS EJB/CORBA サーバー内の任意のサーバー・オブジェクトは、OTS トランザクションに関与するアウトバウンド・クライアント IIOP 呼び出しを行うことができます。ただし、これらのアウトバウンド呼び出しの実行に使用される ORB インスタンスが、現行の CICS EJB/CORBA サーバー ORB である場合です。新しい ORB インスタンスが new 演算子を使用してサーバー・オブジェクトによって作成される場合、CICS はこの新しい ORB を使用して既存のトランザクション・コンテキストを自動的に伝搬することはできません。IIOP サーバー・オブジェクトは、次の静的メソッド呼び出しを使用して、現行のサーバー ORB インスタンスへのハンドルをプログラマチックに取得できます。

```
com.ibm.cics.iiop.ORBFactory.getORB()
```

CORBA の相互運用性

CORBA アーキテクチャーを CICS に実装すると、エンタープライズ Bean を含めて、CORBA ORB と CICS サービスに基づくアプリケーション間のリンクが提供されます。

他の CICS 領域 (CICS TS 1.3 以降からのバックレベル CICS 領域を含む) 上のオブジェクト、WebSphere Application Server、およびサード・パーティー J2EE アプリケーション・サーバーや ORB と、CICS によってホスティングされるエンタープライズ Bean を相互運用させることができます。エンタープライズ Bean は、純粋な CORBA クライアントから使用可能であり、(おそらく、別のプログラミング言語で実装され、別のプラットフォームでホスティングされている) リモート CORBA オブジェクトに対してクライアントの役目をすることができます。

CICS ORB は、Java で作成されたクライアント・アプリケーションとサーバー・アプリケーションのみのホスティングに使用できます。ただし、他のプログラミング言語で作成されたクライアントとサーバーにサービスを提供するリモート ORB と相互運用するのに使用できます。

非 Java CORBA クライアントの使用

プログラミング言語が異なれば、ORB への別の言語バインディングが必要です。

これには ORB 間のレベルの相互運用性が必要であり、これを考慮に入れなければなりません。CORBA アーキテクチャーは、C++、Java、 COBOL、Ada、PL/I、Smalltalk などの複数の言語の言語バインディングを定義します。一部のプログラミング言語の言語バインディングは、すべての IDL および IIOP の機能をサポートするとは限らないことに注意してください。特に、valuetype は、C++ および Java 言語バインディングのみに対して定義されています。CORBA がエンタープライズ Bean にアクセスするには valuetype が必要であるため、現在、CORBA インターフェースを通じて大部分のエンタープライズ Bean にアクセスできるのは、C++ および Java アプリケーションのみです。

エンタープライズ Bean への CORBA クライアントの書き込み

Java 以外のクライアント・プログラミング言語 (C++ など) では、多くの場合、CORBA アーキテクチャーが、エンタープライズ Bean にアクセスするための唯一の有効なオプションです。

このタスクについて

Java 以外のクライアント・プログラミング言語 (C++ など) では、多くの場合、CORBA アーキテクチャーが、エンタープライズ Bean にアクセスするための唯一の有効なオプションです。エンタープライズ Bean は、次のように CORBA プログラミング・モデルを通じて CORBA クライアントから利用可能です。

- エンタープライズ Bean を作成します。
- -IDL オプションを指定した RMI コンパイラーを使用して、エンタープライズ Bean の IDL を生成します (これは、オブジェクトの生成に IDL が使用される標準的な CORBA モデルの逆です)。

  データ型および戻りの型として CORBA プリミティブのみを使用する場合、非 Java クライアントから Bean にアクセスする方が簡単です。

- クライアント環境に適した IDL コンパイラーを使用して、IDL をコンパイルしてクライアント・サイドのスタブを生成します。
- 生成されたスタブを使用して、クライアントを書き込みます。
- エンタープライズ Bean の IOR をクライアント・アプリケーションから使用可能にします。この IOR には、任意の CORBA ORB がエンタープライズ Bean を位置指定するのに十分な情報が入っています。

パラメーターおよび戻りの型として CORBA プリミティブのみを使用するようにセッション Bean がコーディングされている場合であっても、例外タイプが引き続き CORBA valuetype として戻されます。CORBA クライアント ORB が valuetype をサポートしない場合、不明の例外を処理するように強制されます。

注: エンタープライズ Bean に対して Java CORBA クライアントを使用することはお勧めしません。代わりに RMI-IIOP を使用してください。

CORBA クライアントとしてのエンタープライズ Bean

エンタープライズ Bean は、ORB を含む高度なランタイム環境で動作する Java オブジェクトです。

エンタープライズ Bean が、(RMI-IIOP を使用せずに) リモート CORBA オブジェクトへのアウトバウンド IIOP 呼び出しを行う場合、アプリケーションが既存の ORB インスタンスを利用することを強くお勧めします。エンタープライズ Bean が new 演算子を使用して新しい ORB インスタンスを作成する場合、CICS は、Bean が実行されている既存のトランザクションおよびセキュリティー・コンテキストを、この新しい ORB のメソッド要求にまで伝搬することはできません。

エンタープライズ Bean 内から現行 ORB へのハンドルを取得する必要がある場合は、次の静的メソッド呼び出しを使用できます。

```
com.ibm.cics.iiop.ORBFactory.getORB()
```

コード・セット

CICS が GIOP char/wchar および string/wstring データ型を受け入れることができるのは、それらのデータ型が次のいずれかのコード・ページを使用してエンコードされる場合のみです。

- UCS2 — 標準 Java コード・セット (Unicode)
- UTF-8

IIOP サンプルの使用

以下のサンプル・アプリケーションは、IIOP アプリケーション (ステートレス CORBA オブジェクト) および CICS Java プログラミング・サポート (JCICS) の使用を示しています。

HelloWorld サンプル

このサンプルは、IIOP コンポーネントの簡単なテストを行います。このクライアント・プログラムは以下のことを行います。

- genfac.ior ファイルを読み取って、Generic Factory への参照を取得します
- Generic Factory を使用して HelloWorld オブジェクトを作成します

- メソッド sayHello を呼び出して、サーバーにグリーティングを送信し (Hello from HelloWorldClient)、応答としてサーバーからグリーティングを受信します (Hello from CICS TS)

アプリケーションの設計は、コード内のコメントで記述されています。

BankAccount サンプル

このサンプルは、次の主な部分で構成されます。

1. BMS、および C で書かれた **EXEC CICS** API を使用する従来の CICS アプリケーション。このアプリケーションは次の 2 つのトランザクションで構成されます。

   BNKI 複数の銀行口座に関する情報を使用してファイルを初期化します。これらの口座には、23 から 30 までの番号があります。

   BNKQ 口座内の情報を照会します。口座の信用調査を行う CICS プログラム DFH$IICC もあります。
2. 銀行口座オブジェクトを定義する IDL インターフェースの実装。この実装は Java で作成され、ステートレス CORBA オブジェクトとして実行されます。この実装は、銀行口座ファイルを使用して銀行口座情報にアクセスし、DFH$IICC 信用調査プログラムを使用して信用等級を取得します。
3. 銀行口座オブジェクトに関する情報を表示する、Java で作成された CORBA クライアント・アプリケーション。

アプリケーションの設計は、コード内のコメントで記述されています。

IIOP サンプル環境のセットアップ

提供されたサンプルを使用して、CICS における IIOP 環境をセットアップできます。

始める前に

IIOP サーバーまたはクライアントとして CICS を構成するには、z/OS UNIX および IBM 64-bit SDK for z/OS, Java テクノロジー・エディション にアクセスする権限を持つ CICS 領域が必要です。Language Environment が構成され、アクティブでなければなりません。

手順

1. IIOP をサポートする CICS 領域の始動ジョブ・ストリームで、JCL パラメーター **REGION** を定義します。 値を少なくとも 1000M に設定してください。
2. IIOP をサポートする CICS 領域の始動ジョブで、次のシステム初期設定パラメーターを定義します。

   ```
   EDSALIM=500M
   MAXJVMTCBS=number
   TCPIP=YES
   ```

   EDSALIM パラメーターに最小値 500M を設定します。**MAXJVMTCBS** パラメーターの適切な値を算出するには、187 ページの『パフォーマンスに関する JVM プールの管理』を参照してください。
3. IIOP をサポートする CICS 領域の始動ジョブ・ストリームに DD ステートメントを追加して、次のファイルを作成します。

DFHEJDIR

要求ストリーム・ディレクトリーを含む、リカバリー可能な共用ファイル。このファイルは、VSAM ファイルの場合とカップリング・ファシリティー・データ・テーブルの場合があります。CICS では、このファイルを SDFHINST ライブラリーの DFHDEFDS メンバーに作成するためのサンプル JCL が提供されています。

DFHEJOS

CORBASERVER リソースのインストール時に CICS で使用される、リカバリー不能共用ファイル。このファイルは、不動態化されているステートフル・セッション Bean の保管にも使用されます。DFHEJOS は、VSAM ファイルの場合とカップリング・ファシリティー・データ・テーブルの場合があります。CICS では、このファイルを SDFHINST ライブラリーの DFHDEFDS メンバーに作成するためのサンプル JCL が提供されています。

これらのファイルのサンプル・ローカル VSAM データ・セット定義は、CICS 提供の RDO グループ DFHEJVS で提供されます。これらのデータ・セットは、UPDATE アクセスに対して RACF® で許可されなければなりません。を参照してください。「*CICS RACF Security Guide*」の Authorizing access to CICS data sets。

4. z/OS UNIX でシェルフ・ディレクトリーを作成し、CICS 領域ユーザー ID に、そのディレクトリーへの全アクセス権限を付与します。詳しくは、Giving CICS regions access to z/OS UNIX System Services を参照してください。
5. 適切な JVM プロファイルを選択し、102 ページの『プールされた JVM のセットアップ』で説明されているとおりに、CICS がこのプロファイルを確実に検出できるようにします。
6. JAVA_HOME の値が、サーバー・サイド・アプリケーションの JVM プロファイルで正しく定義されていることを確認します。 JAVA_HOME は、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション のインストール・ディレクトリーを定義します。この SDK のバージョン 6.0.1 のデフォルトは、`/usr/lpp/java/J6.0.1_64/` です。
7. 以下のファイルが、JVM プロファイルで適切なクラスパスに確実に追加されるようにします。
   - CICS インストール時に z/OS UNIX システム・サービス・ファイル・システムで以下のディレクトリーに保管されている、サンプルの Java ソースおよび Make ファイル。
     - `$CICS_HOME/samples/dfjcorb/HelloWorld`
     - `$CICS_HOME/samples/dfjcorb/BankAccount`
   - サーバー・サイド・アプリケーションのクラスをコンパイルした場所。

   詳しくは、10 ページの『JVM におけるクラスおよびクラスパス』 を参照してください。
8. CICS 提供のリソース定義グループ DFHIIOP および DFH$IIOP がインストールされていることを確認します。 提供されたグループ DFH$IIOP には、次の定義が入っています。

- TCP/IP リスナー領域に必要なリソース定義 (サンプル・プログラムを実行するのと同じ領域である場合もあります)。
  - SSL TCPIPSERVICE 定義
  - NOSSL TCPIPSERVICE 定義
- HelloWorld サンプルに必要なリソース定義。
  - IIHE TRANSACTION 定義
  - DFJIIRH REQUESTMODEL 定義
  - IIOP CORBASERVER 定義
- BankAccount サンプルに必要なリソース定義。
  - DFH$IIBI PROGRAM 定義
  - DFH$IIBQ PROGRAM 定義
  - DFH$IICC PROGRAM 定義
  - BANKINQ MAPSET 定義
  - BNKI TRANSACTION 定義
  - BNKQ TRANSACTION 定義
  - BNKS TRANSACTION 定義
  - BANKACCT FILE 定義
  - DFJIIRB REQUESTMODEL 定義
  - IIOP CORBASERVER 定義

TCPIPSERVICE および IIOP CORBASERVER 定義は、デフォルトのポート番号 683 および 684 を指します。使用可能なポート番号への変更が必要な場合があります。また、IIOP 定義では、CICSHOST を CorbaServer のホストとして参照します。これを独自のホスト名に変更する必要があります。TCPIPSERVICE resources と CORBASERVER resources を参照してください。

9. 以下の CICS C 言語プログラムおよびマップ・セットの変換とコンパイルを行い、CICS DFHRPL 連結のライブラリーに入れます。 CICS インストール時に SDFHSAMP に保管されます。コンパイルの順序が重要です。DFH$IIBI と DFH$IICC はどちらも独立してコンパイルできますが、DFH$IIBQ をコンパイルする前に BMS マップ・セット DFH$IIMA をコンパイルしておく必要があります。 CICS アプリケーション・プログラムの変換、コンパイル、およびリンクについては、「*CICS アプリケーション・プログラミング・ガイド*」を参照してください。

   DFH$IIMA ファイルには、2 つのマップを持つ 1 つのマップ・セット BANKINQ が入っています。マップ・セット BANKINQ をコンパイルし、リンクしてください。

   BMS マップのコンパイルとリンクについては、「*CICS アプリケーション・プログラミング・ガイド*」の Installing map sets and partition sets を参照してください。

   DFH$IIBI
   BANKACCT ファイルを初期化する C プログラム。BNKI トランザクションによって実行されます。

   DFH$IIBQ
   BANKACCT で保持される口座を照会する C プログラム。

DFH$IICC
信用調査を実行する C プログラム。これは DFH$IIBQ によって呼び出されます。

DFH$IIMA
BMS マップ・セット BANKINQ。

注: 本書で説明されているサンプル・プログラムとファイルの名前で、ドル記号 ($) が国の通貨記号として使用され、EBCDIC コード・ポイント X'5B' が割り当てられるものと見なされます。一部の国では、別の通貨記号 (例えば、ポンド記号 (£) や円記号 (¥)) が、同じ EBCDIC コード・ポイントに割り当てられます。これらの国では、ドル記号ではなく、該当する通貨記号を使用してください。

10. IIOP HelloWorld クライアントをコンパイルするには、CosLifeCycle および CosNaming ランタイム・クラスが必要です。 クライアント ORB 環境でこれらのサービスがあらかじめ用意されていない場合、出荷時に `$CICS_HOME/lib` ディレクトリーにある `omgcos.jar` ファイルを使用できます。または、オリジナルの OMG 提供 IDL からこれらのクラスを作成することも選択できます。この場合、関連する IDL ファイルのコピーが `$CICS_HOME/samples/dfjcorb` で入手可能です。純粋な IDL を実行可能コードに変えるプロセスは、ORB に依存します。しかし、JVM に付属の ORB を使用する場合、おそらく次のコマンドが機能します。

```
idlj -pkgprefix CosNaming org.omg -pkgprefix CosLifeCycle org.omg -fall CosLifeCycle.idl
idlj -pkgprefix CosNaming org.omg -pkgprefix CosLifeCycle org.omg -fall CosNaming.idl
javac org\omg\CosLifeCycle\*.java org\omg\CosNaming\NamingContextPackage\*.java
      org\omg\CosNaming\*.java
```

CICS ステートレス CORBA クライアント・アプリケーションをビルドしようとするときに、CLASSPATH 環境変数でこれらのクラスが使用可能であることを確認する必要があります。

11. サーバーの Generic Factory へのオブジェクト参照を含む `genfac.ior` ファイルを取得し、現行ディレクトリーに入れます。 `genfac.ior` ファイルは、インストールされたサンプル IIOP CORBASERVER リソースに対して **PERFORM CORBASERVER PUBLISH** コマンドを発行するときに作成されます。このファイルは、CORBASERVER のシェルフ・ディレクトリーに書き込まれます。

    `/var/cicsts/applid/IIOP`

    ここで、*applid* は、CICS 領域に関連したアプリケーション ID です。

    CORBASERVER 定義を公開するには、**CEMT PERFORM CORBASERVER** コマンドを使用するか、CICS アプリケーションから発行される **EXEC CICS PERFORM CORBASERVER** コマンドを使用できます。

    FTP を使用して、シェルフからクライアント・ワークステーションに (ASCII モードで) IOR をダウンロードできます。

タスクの結果

これで、サンプルを使用して IIOP 環境をセットアップしました。

IIOP HelloWorld サンプルの実行

このセクションでは、HelloWorld サンプル・アプリケーションを実行するために必要な作業について説明します。

以下のトピックについて説明します。

- 『サーバー・サイド HelloWorld アプリケーションのビルド』
- 『クライアント・サイド HelloWorld アプリケーションのビルド』
- 250 ページの『HelloWorld サンプル・アプリケーションの実行』

サーバー・サイド HelloWorld アプリケーションのビルド:

$CICS_HOME/samples/dfjcorb/HelloWorld/server 内の Make ファイルは、サーバー・サイド・アプリケーションに必要なすべてのものを作成します。

JVM プロファイル DFHJVMCD の CLASSPATH_SUFFIX オプションを使用して、$CICS_HOME/samples/dfjcorb/HelloWorld/server が標準クラスパスに追加されなければなりません。

プログラムをビルドするには、$CICS_HOME/samples/dfjcorb/HelloWorld/server から make コマンドを実行します。このコマンドは HelloWorld オブジェクトを作成します。

クライアント・サイド HelloWorld アプリケーションのビルド:

$CICS_HOME/samples/dfjcorb/HelloWorld/client には、このアプリケーションの CORBA クライアント部分が入っています。Java クライアント・アプリケーションのソースは HelloWorldClient.java と呼ばれます。このアプリケーションは、任意の CORBA 準拠の ORB で実行されなければなりません。

このタスクについて

Java クライアント・アプリケーションをビルドするには、以下のステップが必要です。

1. 次のファイルをクライアント・ワークステーションに (ASCII モードで) ダウンロードします。
   - .../dfjcorb/HelloWorld/HelloWorld.idl
   - .../dfjcorb/HelloWorld/client/HelloWorldClient.java
2. クライアント ORB の IDL-to-Java コンパイラーを使用して、提供されている IDL をコンパイルして、このサンプル・アプリケーションで必要な Java クライアント・サイド・スタブを作成します。これらのスタブは、hello と呼ばれるサブディレクトリーで作成されます。クライアント・アプリケーション HelloWorldClient.java をこのサブディレクトリーに移動します。
3. クライアント・アプリケーションをコンパイルして、前のステップで作成された Java クラスが、CLASSPATH 環境変数から使用可能であることを確実にします。現行ディレクトリーからクライアント・アプリケーションをコンパイルするには、次のように入力します。

   ```
   javac hello\HelloWorldClient.java
   ```

また、CosLifeCycle および CosNaming ランタイム・クラスも必要です。クライアント ORB 環境でこれらのサービスがあらかじめ用意されていない場合、z/OS UNIX の $CICS_HOME/lib ディレクトリーに出荷時にある omgcos.jar ファイルを使用できます。または、オリジナルの OMG 提供 IDL からこれらのクラスを作成することも選択できます。この場合、関連する IDL ファイルのコピーが $CICS.HOME/samples/dfjcorb/ で入手可能です。

純粋な IDL を実行可能コードに変えるプロセスは、ORB に依存します。しかし、JVM に付属の ORB を使用する場合、おそらく次のコマンドが機能します。

```
idlj -pkgprefix CosNaming org.omg -pkgprefix CosLifeCycle org.omg -fall CosLifeCycle.idl
idlj -pkgprefix CosNaming org.omg -pkgprefix CosLifeCycle org.omg -fall CosNaming.idl
javac org\omg\CosLifeCycle\*.java
              org\omg\CosNaming\NamingContextPackage\*.java
              org\omg\CosNaming\*.java
```

CICS ステートレス CORBA クライアント・アプリケーションをビルドしようとするときに、これらのクラスがクラスパスで使用可能でなければなりません。

HelloWorld サンプル・アプリケーションの実行:

このクライアント・アプリケーションを実行するには、次のコマンドを使用する必要があります。

このタスクについて

```
java hello.HelloWorldClient
```

IIOP BankAccount サンプルの実行

このセクションでは、BankAccount サンプル・アプリケーションを実行するために必要な作業について説明します。

以下のトピックについて説明します。

- 『サーバー・サイド BankAccount アプリケーションのビルド』
- 251 ページの『クライアント・サイド BankAccount アプリケーションのビルド』
- 251 ページの『BankAccount サンプル・アプリケーションの実行』

VSAM ファイルの作成:

次の IDCAMS パラメーターを使用して銀行口座データを保持するための VSAM ファイルを定義する必要があります。

```
DEFINE    CLUSTER (                        -
                NAME (CICS610.BANKACCT )     -
                CYLINDERS(01)                -
                REUSE                        -
                KEYS(4 0)                    -
                RECORDSIZE(168 168))
```

サーバー・サイド BankAccount アプリケーションのビルド:

$CICS_HOME/samples/dfjcorb/BankAccount/server 内の Make ファイルは、サーバー・サイド・アプリケーションの CORBA 部分に必要なすべてのものを作成します。

JVM プロファイル DFHJVMCD の CLASSPATH_SUFFIX オプションを使用して、$CICS_HOME/samples/dfjcorb/BankAccount/server を標準クラスパスに追加します。

Java サーバー・プログラムをビルドするには、$CICS_HOME/samples/dfjcorb/BankAccount/server から make コマンドを実行します。

クライアント・サイド BankAccount アプリケーションのビルド:

$CICS_HOME/samples/dfjcorb/BankAccount/javaclient には、このアプリケーションの CORBA クライアント部分が入っています。Java クライアント・アプリケーションのソースは bankLineModeClient.java と呼ばれます。このアプリケーションは、任意の CORBA 準拠の ORB で実行できます。

このタスクについて

Java クライアント・アプリケーションをビルドするには、以下のステップが必要です。

1. 次のファイルをクライアント・ワークステーションに (ASCII モードで) ダウンロードします。
   - .../dfjcorb/BankAccount/BankAccount.idl
   - .../dfjcorb/BankAccount/javaclient/bankLineModeClient.java
2. クライアント ORB の IDL-to-Java コンパイラーを使用して、提供されている IDL をコンパイルして、このサンプル・アプリケーションで必要な Java クライアント・サイド・スタブを作成します。IDL をコンパイルしてサブディレクトリー bank を作成した後、Java ファイルをこのサブディレクトリーに移動します。次のコマンドを使用して、現行ディレクトリーからプログラムをコンパイルします。

   ```
   javac bank\bankLineModeClient.java
   ```
3. 前のステップで作成された Java クラスが、CLASSPATH 環境変数から使用可能であることを確実にします。

   また、CosLifeCycle および CosNaming ランタイム・クラスも必要です。クライアント ORB 環境でこれらのサービスがあらかじめ用意されていない場合、 249 ページの『クライアント・サイド HelloWorld アプリケーションのビルド』と同じ方法で取得できます。

BankAccount サンプル・アプリケーションの実行:

このサンプル・アプリケーションを実行するには、以下のステップを実行する必要があります。

このタスクについて

手順

1. BNKI CICS トランザクションを実行して、データをアカウント・ファイルにロードします。
2. 次のコマンドを使用してクライアント・アプリケーションを実行します。

   ```
   java  bank.bankLineModeClient
   ```

エンタープライズ Bean の使用

このセクションでは、CICS におけるエンタープライズ Bean の開発と使用に必要な知識について説明します。

- 『エンタープライズ Bean とは』
- 280 ページの『EJB サーバーのセットアップ』
- 301 ページの『EJB IVP の使用』
- 306 ページの『サンプル EJB アプリケーションの実行』
- 336 ページの『エンタープライズ Bean の作成』
- 351 ページの『エンタープライズ Bean のデプロイ』
- 357 ページの『実動領域でのエンタープライズ Bean の更新』
- 369 ページの『CCI Connector for CICS TS』
- 387 ページの『CICS エンタープライズ Bean の問題の処理』
- 395 ページの『エンタープライズ Bean のセキュリティーの管理』
- 409 ページの『エンタープライズ Bean での CICSPlex SM』

エンタープライズ Bean とは

CICS は、Enterprise JavaBeans(EJB) アーキテクチャーをサポートします。

EJB アーキテクチャーの詳しい説明が必要な場合は、http://www.oracle.com/technetwork/java/index.html を参照してください。

このセクションでは、次のトピックを取り上げています。

- 『エンタープライズ Bean』
- 253 ページの『JavaBeans および Enterprise JavaBeans』
- 255 ページの『EJB サーバー — 概要』
- 256 ページの『EJB コンテナー — 概要』
- 257 ページの『エンタープライズ Bean — ホーム・インターフェースとコンポーネント・インターフェース』
- 258 ページの『エンタープライズ Bean — デプロイメント記述子』
- 259 ページの『エンタープライズ Bean のタイプ』
- 262 ページの『エンタープライズ Bean — トランザクションの管理』
- 264 ページの『エンタープライズ Bean — セキュリティーの概要』
- 265 ページの『エンタープライズ Bean — ユーザー・タスク』
- 267 ページの『エンタープライズ Bean のデプロイの概要』
- 270 ページの『EJB サーバーとしての CICS の構成の概要』
- 278 ページの『エンタープライズ Bean — クライアントが Bean でできること』
- 279 ページの『エンタープライズ Bean — Bean ができること』

エンタープライズ Bean

「*Enterprise JavaBeans Specification, Version 1.1*」は、エンタープライズ *Bean* と呼ばれる再使用可能な Java サーバー・コンポーネントの開発モデルを定義します。これらのコンポーネントは、その仕様で定義されるサービスとインターフェースを提供する任意のアプリケーション・サーバーで使用できます。

CICS は *EJB* サーバー として構成できます。CICS が提供するランタイム環境では、EJB サービスに対する要求は、既存または拡張 CICS サービスにマップされます。

過去に投資された CICS アプリケーションとデータへのアクセス権限を Java クライアントに付与するエンタープライズ Bean を作成できます。例えば、以下に当てはまるエンタープライズ Bean を作成できます。

- JCICS クラスを使用して CICS リソースにアクセスします。 JCICS クラスを使用するエンタープライズ Bean は、非 CICS 環境に移植できません。
- JCICS を使用して、COBOL などのプロシージャー型言語で作成された既存の CICS プログラムにリンクします。

図 12 は、CICS EJB アプリケーション・サーバーがその環境と対話する様子を簡単な形式で示しています。ワークステーションで開発されたエンタープライズ Bean が、デプロイメント と呼ばれるプロセスで EJB サーバーにインストールされる様子を示しています。エンタープライズ Bean はサーバーにインストールされた後、クライアント・プログラムの要求により Java 仮想マシン (JVM) で実行されます。

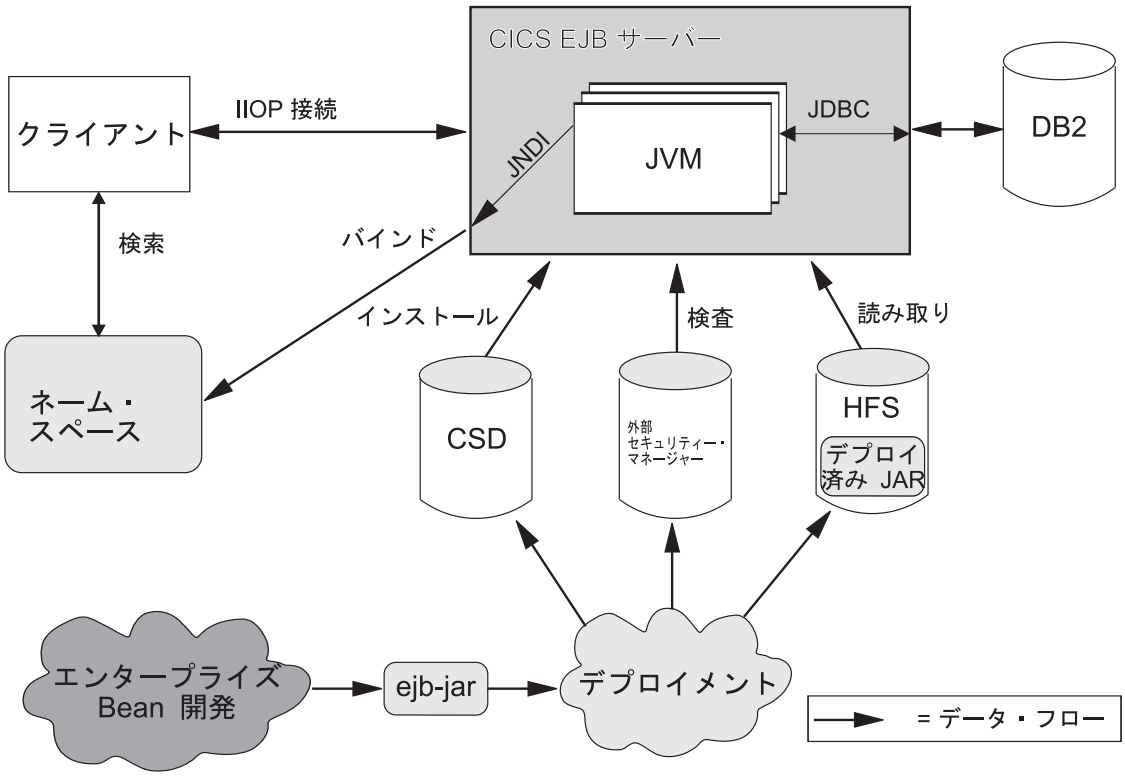


図 12. CICS EJB アプリケーション・サーバー

JavaBeans および Enterprise JavaBeans

JavaBeans と Enterprise JavaBeans は、Java 言語のコンポーネント・アーキテクチャーです。

コンポーネント:

コンポーネントは、再使用可能なソフトウェア・ビルディング・ブロックです。事前に作成されたカプセル化アプリケーション・コードであり、他のコンポーネントや手書きコードと組み合わせてカスタムビルト・アプリケーションを迅速に作成できます。

アプリケーション開発者は、ソース・コードにアクセスする必要なく、コンポーネントを利用できます。コンポーネントは、1 組の外部プロパティー値を使用してアプリケーション固有の要件を満たすようにカスタマイズできます。例えば、ボタン・コンポーネントには、そのボタンに表示されるキャプションを指定するプロパティーがあります。アカウント管理コンポーネントには、アカウント・データベースの場所を指定するプロパティーがあります。

コンポーネントは、**コンテナー**と呼ばれる構成体内で実行されます。コンテナーは、(特に) コンポーネントを実行するオペレーティング・システム・プロセスを提供します。

コンポーネント・モデルは、コンポーネントがそのコンテナーや他のコンポーネントと対話するのに使用するインターフェースを定義します。コンポーネントの開発者は、さまざまな内部メソッドやプロパティーを使用してコンポーネントをコード化できますが、他のコンポーネントと一緒に使用できるようにするには、コンポーネント・モデルで定義されたインターフェースを実装する必要があります。また、これらのインターフェースは、rapid application development (RAD) ツール (WebSphere Studio Application Developer など) にコンポーネントをロードできるようにします。

JavaBeans:

JavaBean は、通常はデスクトップまたはクライアント・アプリケーションで使用することを目的とした、Java で作成された内蔵タイプの再使用可能なソフトウェア・コンポーネントです。

一般に、デスクトップ JavaBeans はビジュアル・エレメントを備え、何らかのタイプのビジュアル・コンテナー (フォーム、パネル、Web ページなど) 内で実行されます。その例は、単純なボタンから、フル機能を備えたソフトウェア CD プレイヤーまでにわたります。

Bean 開発者は、WebSphere Studio Application Developer などのビジュアル・ツールを使用して、JavaBeans を作成できます。アプリケーション開発者はこのようなツールを使用して、JavaBeans をまとめて 1 つのより大きいアプリケーションに「接続」し、個々の Bean のプロパティーを設定することができます。

Enterprise JavaBeans:

Enterprise JavaBeans アーキテクチャーはサーバー・コンポーネント をサポートします。サーバー・コンポーネントは、CICS などのアプリケーション・サーバーで実行されるアプリケーション・コンポーネントです。デスクトップ・コンポーネントとは異なり、サーバー・コンポーネントにはビジュアル・エレメントがなく、それらのコンポーネントが実行されるコンテナーは可視ではありません。

Enterprise JavaBeans 仕様にしたがって作成されたサーバー・コンポーネントは、**エンタープライズ Bean** と呼ばれます。これらのコンポーネントは、EJB 準拠のどのアプリケーション・サーバー間でも移植可能です。

サーバー・コンポーネントを便利に利用するには、サーバー・コンポーネントに、アプリケーション・サーバーのインフラストラクチャー・サービス (分散通信サービス、ネーミング・サービスやディレクトリー・サービス、トランザクション管理サービス、データ・アクセス・サービスやパーシスタンス・サービス、リソース共用サービスなど) へのアクセスが必要です。アプリケーション・サーバーが異なれば、別のテクノロジーを使用してこれらのインフラストラクチャー・サービスを実装します。ただし、EJB 準拠のアプリケーション・サーバーは、エンタープライズ Bean が標準インターフェースを使用してこれらのサービスにアクセスできるようにして、Bean に代わってそれらのサービスの多くを管理します。

Bean 開発者は、WebSphere Studio Application Developer などのビジュアル・ツールを使用して、エンタープライズ Bean を作成できます。アプリケーション開発者は、エンタープライズ Bean へのメソッド呼び出しを、デスクトップ JavaBeans、Web サーブレット、および手書きコードと組み合わせて、クライアント/サーバー・アプリケーションを作成できます。

EJB サーバー — 概要

EJB 準拠のアプリケーション・サーバーは *EJB サーバー* と呼ばれます。

EJB サーバーは、CICS、Web サーバー、データベース、またはその他の何らかのタイプのサーバーなどのトランザクション処理モニターにすることができます。 271 ページの『論理サーバー: シスプレックス内のエンタープライズ Bean』で説明されているように、CICS EJB サーバーは、複数の CICS 領域で構成される場合があることに注意してください。

EJB サーバーは、エンタープライズ Bean コンポーネントをサポートするための 1 組の標準サービスを提供します。これらのサービスには次のものがあります。

- エンタープライズ Bean が通信に使用する Java リモート・メソッド呼び出し (RMI) インターフェースのサポート。RMI には、2 つのトランスポート・プロトコル・オプションがあります。すなわち、Java-to-Java 相互運用のための JRMP と、CORBA オブジェクト・リクエスト・ブローカー (ORB) を使用して仲介される、言語間相互運用のための IIOP です (CICS ORB の説明については、416 ページの『オブジェクト・リクエスト・ブローカー (ORB)』を参照してください)。

  CICS Transaction Server for z/OS, バージョン 4 リリース 2 は、RMI over IIOP (RMI-IIOP) をサポートしますが、JRMP をサポートしません。(JRMP は、非 Java コンポーネントとの相互運用に使用できない専有プロトコルです。 CICS は JRMP を介した分散トランザクションをサポートしません。)
- エンタープライズ Bean に管理サービスを提供する、**EJB コンテナー**と呼ばれるコンテナー。
- Java Transaction API (JTA) の javax.transaction.UserTransaction インターフェースを実装する分散トランザクション管理サービス。javax.transaction.UserTransaction インターフェースは、独自のトランザクションを管理するセッション Bean で使用されます。
- セキュリティー・サービス。

- Java Naming and Directory Interface (JNDI) のサポート。 JNDI API は、Java アプリケーションにディレクトリーとネーミング機能を提供します。これにより、クライアントはエンタープライズ Bean の位置を確認することができます。
- Java Data Base Connectivity (JDBC) インターフェースのサポート。

EJB コンテナー — 概要

デスクトップ JavaBeans は通常、フォームや Web ページなどのビジュアル・コンテナー内で実行されますが、エンタープライズ Bean は、アプリケーション・サーバーによって提供されるコンテナー内で実行されます。

EJB コンテナーは、実行時にエンタープライズ Bean インスタンスの作成と管理を行い、その EJB コンテナーで実行される各エンタープライズ Bean で必要なサービスを提供します。

EJB コンテナーは、ライフサイクル、状態管理、セキュリティー、トランザクション管理を始めとする、複数の暗黙サービスをサポートします。

ライフサイクル
個々のエンタープライズ Bean は、プロセス割り振り、スレッド管理、オブジェクトの活動化、オブジェクトの不動態化を明示的に管理する必要はありません。 EJB コンテナーが、エンタープライズ Bean の代わりにオブジェクトのライフサイクルを自動的に管理します。

状態管理
個々のエンタープライズ Bean は、メソッド呼び出し間でオブジェクト状態の明示的な保管も復元も行う必要はありません。 EJB コンテナーが、エンタープライズ Bean の代わりにオブジェクト状態を自動的に管理します。

セキュリティー
個々のエンタープライズ Bean は、ユーザーの認証も、許可レベルの検査も明示的に行う必要はありません。 EJB コンテナーが、エンタープライズ Bean の代わりにすべてのセキュリティー検査を自動的に行うことができます。

トランザクション管理
個々のエンタープライズ Bean は、分散トランザクションに関与するためのトランザクション区分コードを指定する必要はありません。 EJB コンテナーが、エンタープライズ Bean の代わりにトランザクションの開始、登録、コミットメント、およびロールバックを自動的に管理できます。

実行環境:

エンタープライズ Bean を EJB サーバーにデプロイする前に、実行環境を構成する必要があります。

CICS では、これは CORBASERVER リソース定義をインストールすることによって行われます。CORBASERVER は、エンタープライズ Bean および CORBA ステートレス・オブジェクトの実行環境を定義します。便宜上、CORBASERVER 定義によって定義される実行環境を **CorbaServer** と呼びます。

以下のことに留意してください。

- CICS EJB サーバーには、複数の CorbaServer が含まれる場合があります。
- 複数のエンタープライズ Bean を同一の CorbaServer にデプロイできます。

- 特定のエンタープライズ Bean を同一の CICS EJB サーバーに複数回デプロイできますが、同一の CorbaServer に複数回デプロイすることはできません。(つまり、特定のエンタープライズ Bean を同一の CICS EJB サーバーに複数回インストールするには、別々の CorbaServer 実行環境にインストールする必要があります。これを行う 1 つの理由は、異なるデプロイメント・プロパティーで Bean を使用可能にするためです。258 ページの『エンタープライズ Bean — デプロイメント記述子』を参照してください。) 各デプロイメントの結果、異なるホーム・オブジェクトが作成されます (『エンタープライズ Bean — ホーム・インターフェースとコンポーネント・インターフェース』を参照してください)。

エンタープライズ Bean — ホーム・インターフェースとコンポーネント・インターフェース

クライアント・アプリケーションは、エンタープライズ Bean と直接対話しません。

代わりに、クライアントは、デプロイメント・ツールによって生成されるクラスからコンテナーによって作成される 2 つの中間オブジェクトを使用して、エンタープライズ Bean と対話します。それらのクラスの 1 つは EJB **ホーム・インターフェース**を実装し、もう 1 つは EJB **コンポーネント・インターフェース**を実装します。クライアントがこれらの中間オブジェクトを使用してオペレーションを呼び出すと、コンテナーは各メソッド呼び出しを代行受信し、管理サービスを挿入します。

ホーム・インターフェースとコンポーネント・インターフェースは、Java RMI リモート・オブジェクトとして実装されます。これにより、ORB はそれらを分散オブジェクトとしてサポートできます。

ホーム・インターフェース

ホーム・インターフェースは、クライアントが必要なエンタープライズ Bean を識別するメカニズムです。このインターフェースを使用して、クライアントはエンタープライズ Bean の作成と除去、および (CICS でサポートされないエンティティー Bean の場合) エンタープライズ Bean の既存のインスタンスを検出できます。*「クライアント」は、ネットワーク・ワークステーションで実行されるプログラムでない可能性もあることに注意してください。例えば、Web サーバーで実行されるサーブレット、ローカル EJB サーバー上または別の EJB サーバー上のエンタープライズ Bean、プログラム、またはオブジェクトである場合があります。*

Bean が EJB サーバーにデプロイされると、コンテナーは、リモート側からアクセス可能なネーム・スペースにホーム・インターフェースを登録します。Java Naming and Directory Interface (JNDI) API を使用すると、このネーム・スペースへのアクセス権限を持つ任意のクライアントは、名前でホーム・インターフェースの位置を確認することができます。(正確には、クライアントは、ホーム・インターフェースを実装するオブジェクトの位置を名前で確認します。ホーム・インターフェースは EJBHome インターフェースを拡張します。)

コンポーネント・インターフェース

コンポーネント・インターフェースを使用すると、クライアントはエンタープライズ Bean のビジネス・メソッドにアクセスできます。クライアントからのす

べてのビジネス・メソッド呼び出しを代行受信し、Bean がデプロイされたときに指定されたトランザクション、状態管理、パーシスタンス、およびセキュリティーのどのサービスでも挿入します。

クライアントがエンタープライズ Bean のインスタンスを作成するか、検出すると、コンテナーはコンポーネント・インターフェース・オブジェクトを (インスタンスごとに 1 つ) 戻します。(正確には、コンテナーは、コンポーネント・インターフェースを実装するクラスのインスタンスへの参照を戻します。コンポーネント・インターフェースは EJBObject インターフェースを拡張します。)

エンタープライズ Bean ― デプロイメント記述子

エンタープライズ Bean のライフサイクル、トランザクション管理、セキュリティー、およびパーシスタンスを制御する規則は、**デプロイメント記述子**と呼ばれる関連した XML 文書で定義されます。

267 ページの『エンタープライズ Bean のデプロイの概要』を参照してください。

再使用可能なコンポーネントは、1 組の外部プロパティー値を使用してカスタマイズできる場合があります。その結果、ソース・コードを変更することなく、特定アプリケーションの要件を満たすように変更できます。エンタープライズ Bean の開発者は、アプリケーション開発者が Bean をカスタマイズできるようにする 1 組の**環境プロパティー**を (デプロイメント記述子内で) 提供できます。例えば、データベースの場所の指定や、デフォルトの各国語の指定にプロパティーを使用できます。実行時に環境オブジェクトが作成されます。この環境オブジェクトには、アプリケーション・アセンブリー・プロセスまたは Bean デプロイメント・プロセス時に設定されたカスタマイズ済みのプロパティー値が入っています。

EJB サーバー: 要約

このトピックでは、前のトピックで説明された EJB サーバーに関する情報をまとめています。

次の図は、CICS EJB サーバー内のエンタープライズ Bean オブジェクトを示しています。EJB コンテナーは、それに含まれているエンタープライズ Bean を管理し、サービスを提供します。Bean がデプロイされると、デプロイメント・ツールにより、EJB ホーム・インターフェース・クラスとコンポーネント・インターフェース・クラスが生成されます。

ホーム・インターフェースは、JNDI からアクセス可能であり、Bean のライフサイクル・サービスを実装します。クライアントはこのインターフェースを使用して、エンタープライズ Bean を作成し、除去し、(CICS で直接サポートされないエンティティー Bean の場合) エンタープライズ Bean のインスタンスを検出します。

コンテナーは、Bean のインスタンスごとに EJB コンポーネント・インターフェース・オブジェクトを作成します。コンポーネント・インターフェースは、Bean 内のビジネス・メソッドにアクセスできるようにします。クライアントからのすべてのビジネス・メソッド呼び出しを代行受信し、Bean のデプロイメント記述子の設定に基づいて、Bean のトランザクション、状態管理、パーシスタンス、およびセキュリティー・サービスを実装します。

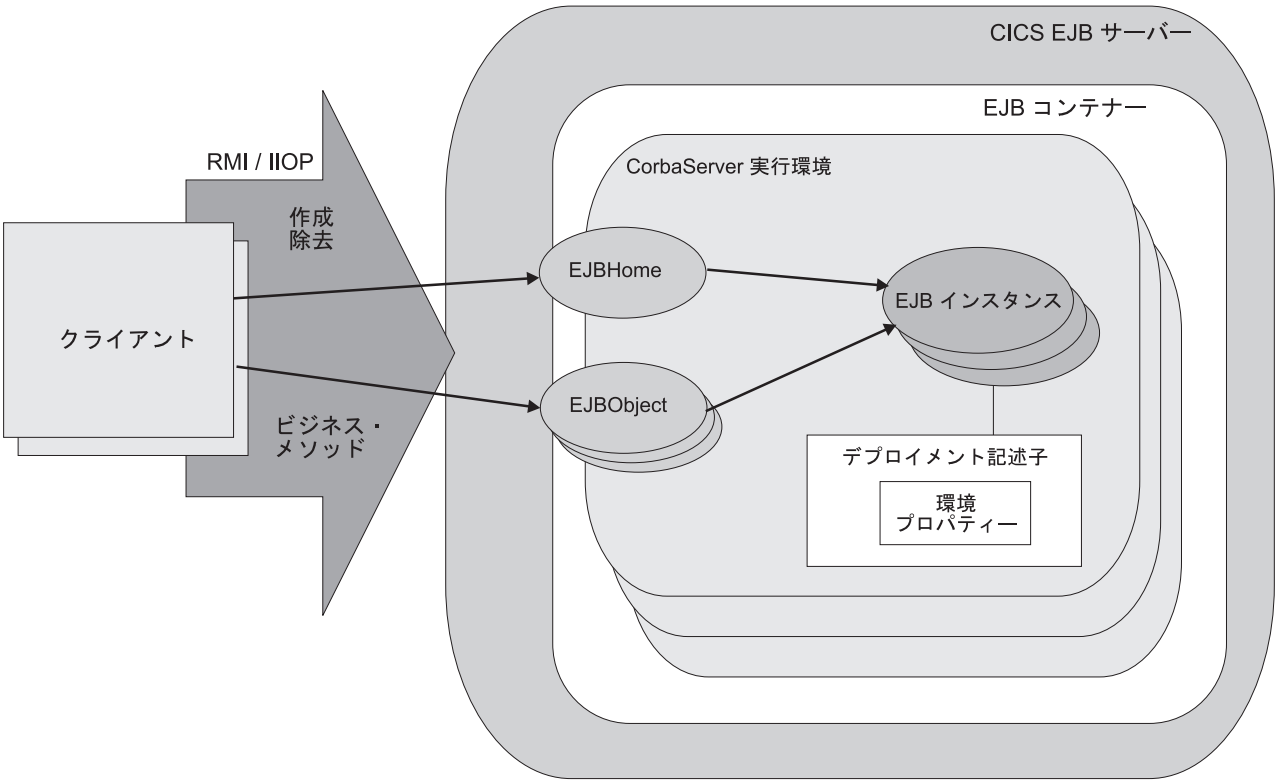


図 13. CICS EJB サーバー内のエンタープライズ Bean オブジェクト

エンタープライズ Bean のタイプ

このセクションでは、2 つのタイプのエンタープライズ Bean (**セッション Bean** と **エンティティー Bean**) について説明しています。

セッション Bean: セッション Bean は次の特徴があります。

- クライアントによって作成され、そのクライアントとの単一の会話またはセッションを表します。
- 通常、クライアントとの会話の存続時間中のみ持続します。この意味で、疑似会話型トランザクションに例えられます。

  Bean 開発者がセッションの存続時間を超えて情報を保管することを選択する場合、Bean クラス・メソッドに直接、パーシスタンス・オペレーション (例えば、JDBC または SQL 呼び出し) を実装する必要があります。
- 通常、クライアントに代わってビジネス・データでオペレーションを実行します (例えば、データベースのアクセスや計算の実行)。
- トランザクションである場合とない場合があります。トランザクションである場合、独自のオブジェクト・トランザクション・サービス (OTS) トランザクションを管理したり、コンテナー管理 OTS トランザクションを使用したりすることができます。OTS トランザクションと CICS 作業単位との関係については、262 ページの『エンタープライズ Bean — トランザクションの管理』を参照してください。

- リカバリー不能です。EJB サーバーが異常終了すると、破棄される可能性があります。
- 2 つのフレーバー (**ステートフル**と**ステートレス**) があります。

ステートフル・セッション Bean:

ステートフル・セッション Bean には、*クライアント固有の* 会話型状態があります。この状態は複数のメソッドやトランザクション間で維持されます。例えば、「ショッピング・カート」オブジェクトは、ユーザーが購入するために選択した品目のリストを維持します。

独自のトランザクションを管理するステートフル・セッション Bean は、あるメソッドで OTS トランザクションを開始し、後続のメソッドでコミットまたはロールバックすることができます。

ステートレス・セッション Bean:

ステートレス・セッション Bean には、クライアント固有の非一時状態がありません (また、その他の種類の非一時状態もありません)。例えば、「株価情報」オブジェクトは現在の株価を戻すことができます。

独自のトランザクションを管理し、トランザクションを開始するステートレス・セッション Bean は、開始したのと同じメソッドでそのトランザクションをコミット (またはロールバック) する必要があります。

エンティティー Bean:

CICS は、エンティティー Bean を直接サポートしません。つまり、エンティティー Bean は CICS EJB サーバーで実行できません。ただし、CICS EJB サーバーで実行中のセッション Bean またはプログラムは、非 CICS EJB サーバーで実行中のエンティティー Bean のクライアントになることができます。

重要

エンティティー Bean は次の特徴があります。

- 通常、顧客オーダーなどのビジネス・データのオブジェクト表示です。一般に、このデータは次のようになります。
  - データベースなどの永続データ・ストアで維持されます。
  - クライアント・インスタンスの存続時間を超えて持続する必要があります。したがって、エンティティー Bean は、セッション Bean に比べると相対的に存続時間が長くなります。
- 同時に複数のクライアントがオブジェクトにアクセスできます。これが可能であるのは、エンティティー Bean の各インスタンスが **1 次キー**によって識別されるからです。1 次キーは、ホーム・インターフェースを介した各インスタンスの検出に使用できます。
- 独自のパーシスタンスを管理するか (**Bean 管理パーシスタンス**)、そのコンテナーにタスクを委任する (**コンテナー管理パーシスタンス**) ことができます。

Bean が独自のパーシスタンスを管理する場合、Bean 開発者は、Bean に直接、パーシスタンス・オペレーション (例えば、JDBC または SQL 呼び出し) を実装する必要があります。

エンティティー Bean がパーシスタンスをコンテナーに委任する場合、コンテナーはパーシスタント状態を透過的に管理します。Bean 開発者は、Bean 内でパーシスタンス・オペレーションをコード化する必要はありません。

- トランザクションである場合とない場合があります。トランザクションである場合、すべてのトランザクション機能は、EJB コンテナーとサーバーによって暗黙的に実行されます。 Bean コード内にトランザクション区分ステートメントはありません。セッション Bean とは異なり、エンティティー Bean は独自の OTS トランザクションの管理を許可されていません。 262 ページの『エンタープライズ Bean — トランザクションの管理』を参照してください。
- リカバリー可能です。サーバーの異常終了後も存続します。

セッション Bean とエンティティー Bean の比較:

ここでは、エンティティー Bean とセッション Bean との相違点についてまとめています。

表 15. セッション Bean とエンティティー Bean の比較

セッション Bean	エンティティー Bean
クライアントとの単一の会話を表します。 通常、ビジネス・データに対して実行される 1 つ以上のアクションをカプセル化します。	通常、パーシスタント・ビジネス・データをカプセル化します (例えば、データベース内の行)。
相対的に存続時間が短くなります。	相対的に存続時間が長くなります。
単一のクライアントによって作成され、使用されます。	複数のクライアントによって共用できます。
1 次キーがありません。	1 次キーがあります。1 次キーにより、1 つのインスタンスを検出し、複数のクライアントで共用できます。
通常、クライアントとの会話の存続時間中のみ持続します。(ただし、情報を保管することを選択できます。)	クライアント・インスタンスの存続時間を超えて持続します。パーシスタンスはコンテナー管理か、Bean 管理にすることができます。
リカバリー不能です。EJB サーバーに障害が起こると、破棄される可能性があります。	リカバリー可能です。EJB サーバーの障害後も存続します。
ステートフル (つまり、クライアント固有の状態がある)、またはステートレス (非一時状態がない) の場合があります。	通常、ステートフルです。

表 15. セッション *Bean* とエンティティー *Bean* の比較 (続き)

セッション Bean	エンティティー Bean
トランザクションである場合とない場合があります。トランザクションである場合、独自の OTS トランザクションを管理したり、コンテナー管理トランザクションを使用したりすることができます。 独自のトランザクションを管理するステートフル・セッション Bean は、あるメソッドで OTS トランザクションを開始し、後続のメソッドでコミットまたはロールバックすることができます。 独自のトランザクションを管理し、OTS トランザクションを開始するステートレス・セッション Bean は、開始したのと同じメソッドでそのトランザクションをコミット (またはロールバック) する必要があります。 トランザクションのステートフル・セッション Bean の状態は、トランザクションのロールバックで自動的にロールバックされることはありません。場合によっては、Bean はセッション同期を使用して同期点に対応することができます。	トランザクションである場合とない場合があります。コンテナー管理トランザクション・モデルを使用する必要があります。 トランザクションである場合、その状態は、トランザクションのロールバックで自動的にロールバックされます。
再入可能ではありません。	再入可能の場合があります。

エンタープライズ Bean — トランザクションの管理

クライアントは、Java Transaction Service (JTS) または CORBA オブジェクト・トランザクション・サービス (OTS) の実装環境を使用して、ACID トランザクションを開始、コミット、およびロールバックすることができます。

これらの ACID トランザクション[1]は、CICS 分散作業単位に似ています。**OTS トランザクション**という用語は、これらのトランザクションを、CICS トランザクション定義 (4 文字のトランザクション ID を持つトランザクション) や CICS トランザクション・インスタンス (大まかに「タスク」と呼ばれる場合があります) から区別するために使用します。

クライアントが OTS トランザクションの有効範囲内でエンタープライズ Bean を呼び出すと、そのトランザクションに関する情報が、IIOP「サービス・コンテキスト」内の EJB サーバーに流れます。これは、メソッド要求の追加 (非表示) パラメーターのようなものです。EJB サーバーは、トランザクションへの関与が必要な場合にこの情報を使用します。エンタープライズ Bean のメソッドがクライアントの OTS トランザクション (ある場合) で実行する必要があるかどうかは、Bean のデプロイメント記述子で指定された**トランザクション属性**の設定で決まります。このメ

1. トランザクション処理の原子性、一貫性、独立性および耐久性。Jim Gray and Andreas Reuter、「*Transaction Processing: Concepts and Techniques*」1993。

ソッドは、クライアントの OTS トランザクションで、またはメソッドの存続期間中に作成される別個の OTS トランザクションで、または OTS トランザクションなしで実行することができます。

エンティティー Bean は**コンテナー管理 OTS トランザクション**を使用する必要があります。すべてのトランザクション機能は、EJB コンテナーとサーバーによって暗黙的に実行されます。 Bean コード内にトランザクション区分ステートメントはありません。

セッション Bean は、コンテナー管理 OTS トランザクションか **Bean 管理 OTS トランザクション**のどちらでも使用できます。Bean 管理トランザクションを使用するセッション Bean は、javax.transaction.UserTransaction インターフェースのメソッドを使用してトランザクションを区分します。独自のトランザクションを管理するステートフル・セッション Bean は、あるメソッドで OTS トランザクションを開始し、後続のメソッドでコミットまたはロールバックすることができます。独自のトランザクションを管理し、OTS トランザクションを開始するステートレス・セッション Bean は、同じメソッドでそのトランザクションをコミット (またはロールバック) する必要があります。

実行時に EJB コンテナーは、Bean のデプロイメント記述子で指定されたトランザクション属性の設定にしたがって、トランザクション・サービスを実装します。トランザクション属性の予想される設定は次のとおりです。

Mandatory
Bean が常に呼び出し元の OTS トランザクションのコンテキスト内で実行されなければならないことを示します。呼び出し元が Bean を呼び出すときに呼び出し元にトランザクションがない場合、コンテナーは javax.transaction.TransactionRequiredException 例外をスローし、要求は失敗します。

Never
Bean が OTS トランザクションのコンテキスト内で呼び出されてはならないことを示します。呼び出し元が Bean を呼び出すときに呼び出し元に OTS トランザクションがある場合、コンテナーは java.rmi.RemoteException 例外をスローし、要求は失敗します。

NotSupported
Bean が OTS トランザクションのコンテキスト内で実行できないことを示します。呼び出し元が Bean を呼び出すときに呼び出し元に OTS トランザクションがある場合、コンテナーはメソッド呼び出しの間、トランザクションを中断します。メソッドが完了したら、中断されたトランザクションを再開します。クライアントの中断状態のトランザクション・コンテキストは、メソッドから呼び出されるリソース・マネージャーにも、エンタープライズ Bean オブジェクトにも渡されません。

Required
Bean が OTS トランザクションのコンテキスト内で実行されなければならないことを示します。呼び出し元が Bean を呼び出すときに呼び出し元に OTS トランザクションがある場合、メソッドは呼び出し元のトランザクションに参加します。呼び出し元に OTS トランザクションがない場合、コンテナーはメソッドの新しい OTS トランザクションを開始します。

RequiresNew
Bean が新しい OTS トランザクションのコンテキスト内で実行されなければならないことを示します。コンテナーは常にメソッドの新しい OTS トランザクションを開始します。呼び出し元が Bean を呼び出すときに呼び出し元に OTS トランザクションがある場合、コンテナーはメソッド呼び出しの間、呼び出し元のトランザクションを中断します。クライアントの中断状態のトランザクション・コンテキストは、メソッドから呼び出されるリソース・マネージャーにも、エンタープライズ Bean オブジェクトにも渡されません。

Supports
トランザクション・コンテキストの有無にかかわらず、Bean が実行できることを示します。呼び出し元が Bean を呼び出すときに呼び出し元に OTS トランザクションがある場合、メソッドは呼び出し元のトランザクションに参加します。呼び出し元に OTS トランザクションがない場合、メソッドは OTS トランザクションなしに実行されます。

注: エンタープライズ Bean メソッドは常に、CICS タスクで CICS 作業単位の下で実行されます。エンタープライズ Bean メソッドが OTS トランザクションなしで実行される場合であっても、そのメソッドがリカバリー可能リソースに対して行う更新はすべて、CICS タスクの正常終了時のみにコミットされ、ロールバックの必要がある場合はバックアウトされます。

メソッドのトランザクション属性の設定により、そのメソッドが実行される CICS タスクが、その作業単位をより幅広い分散 OTS トランザクションの一部にするかどうかが決まります。

CICS はエンタープライズ Bean メソッドの実行を新規タスクの開始として処理するため、単一の CICS タスクに複数のエンタープライズ Bean を含めることはできません。複数のエンタープライズ Bean が含まれるアプリケーションを作成することはできますが、そのアプリケーションは単一の CICS タスクとしては動作しません。

エンタープライズ Bean — セキュリティーの概要

EJB セキュリティーは、認証、アクセス制御、および Java 2 セキュリティー・ポリシー・メカニズムに関係しています。

認証:

EJB クライアントの認証では、TCP/IP Secure Sockets Layer (SSL) プロトコルを使用します。

SSL を使用するように CICS を構成する方法については、「*CICS RACF Security Guide*」の Support for security protocols を参照してください。

アクセス制御:

エンタープライズ Bean メソッドへのアクセスは、セキュリティー役割という概念に基づいて行われます。CICS トランザクション・セキュリティーとリソース・セキュリティーを EJB リソースで使用できます。

セキュリティー役割:

エンタープライズ Bean メソッドへのアクセスは、**セキュリティー役割**という概念に基づいて行われます。セキュリティー役割は、ユーザーが正常にアプリケーションを使用するのに必要な権限の観点からみたときの、アプリケーションのユーザーのタイプを表します。

特定のエンタープライズ Bean または Bean の特定メソッドの実行が許可される役割は、Bean のデプロイメント記述子で指定され、個々のユーザーへのセキュリティー役割のマッピングは、外部セキュリティー・マネージャーで行われます。

セキュリティー役割について詳しくは、400 ページの『セキュリティー役割』を参照してください。

CICS トランザクション・セキュリティーとリソース・セキュリティー:

CICS トランザクション・セキュリティーとリソース・セキュリティーを EJB リソースで使用できます。

CICS トランザクション・セキュリティーは、エンタープライズ Bean メソッドに関連した CICS トランザクション、つまり EJB REQUESTMODEL 定義で指定されたトランザクションに適用されます。

CICS リソース・セキュリティーは、エンタープライズ Bean によって (例えば、JCICS を使用して) アクセスされる CICS リソースに適用されます。

Java セキュリティー・マネージャー:

エンタープライズ Bean コンテナー環境のセキュリティーは、Java セキュリティー・ポリシー・メカニズムによって保護され、CICS セキュリティーとは無関係です。このセキュリティー・ポリシー・メカニズムは、Java セキュリティー・モデルを構成するコンポーネントの 1 つです。

セキュリティー・ポリシー・メカニズムは、エンタープライズ Bean によって発行できない、Java 機能に関する制限を EJB 仕様で実施するのに使用されます。CICS は、この動作を実施するポリシー・ファイルを提供します。

Java セキュリティー・ポリシー・メカニズムがアクティブな状態でエンタープライズ Bean から JDBC または SQLJ を使用するには、DB2 で提供される JDBC 2.0 ドライバーを使用する必要があります。DB2 が提供する IBM Data Server Driver for JDBC and SQLJ は、Java セキュリティーをサポートせず、このメカニズムを使用不可にしない限り、セキュリティー障害で失敗に終わります。

エンタープライズ Bean — ユーザー・タスク

エンタープライズ Bean を使用するアプリケーションの開発とデプロイメントに関係する役割は、Bean プロバイダー、アプリケーション・アセンブラー、デプロイヤー、およびシステム管理者です。

注: 小規模な組織では、1 人の個人が、これらの役割の複数を担当する場合があります。

Bean プロバイダー:

Bean プロバイダーは、一般にビジネス・タスクまたはビジネス・エンティティーを実装する、再使用可能なエンタープライズ Bean を開発します。

Bean プロバイダーの出力は、1 つ以上のエンタープライズ Bean を含む **ejb-jar ファイル**です。Bean プロバイダーは次のものを担当します。

- エンタープライズ Bean のビジネス・メソッドを実装する Java クラス。
- Bean のコンポーネント・インターフェースとホーム・インターフェースの定義。
- Bean のデプロイメント記述子。

  このデプロイメント記述子は、エンタープライズ Bean の構造情報 (例えば、エンタープライズ Bean クラスの名前) を含み、Bean のすべての外部依存関係 (例えば、エンタープライズ Bean が使用するリソース・マネージャーの名前とタイプ) を宣言します。

アプリケーション・アセンブラー:

アプリケーション・アセンブラーは、エンタープライズ Bean を使用するアプリケーションを作成します。エンタープライズ Bean と手書きのクライアント・コードを 1 つのクライアント/サーバー・アプリケーションに結合します。エンタープライズ Bean のコンポーネントとホーム・インターフェースによって提供される機能を十分に理解している必要がありますが、エンタープライズ Bean の実装についての知識は必要ありません。

アプリケーション・アセンブラーへの入力データは、Bean プロバイダーによって生成される 1 つ以上の ejb-jar ファイルです。アプリケーション・アセンブラーの出力は、アプリケーション・アセンブリーの方法とカスタマイズされた環境の設定と共に、エンタープライズ Bean を含む 1 つ以上の ejb-jar ファイルです。アプリケーション・アセンブラーは、アプリケーション・アセンブリーの方法、セキュリティー役割、および環境値をデプロイメント記述子に挿入しました。

また、アプリケーション・アセンブラーは、アプリケーションのアセンブル時にエンタープライズ Bean を他のタイプのアプリケーション・コンポーネント (例えば、JavaBeans) と結合します。

通常、アプリケーション・アセンブリー手順は、エンタープライズ Bean のデプロイメントの前に行われます。ただし、エンタープライズ Bean の全部または一部のデプロイメント後に、アセンブリーが実行される場合があります。

デプロイヤー:

デプロイヤーは、アプリケーション・アセンブラーによって生成される 1 つ以上の ejb-jar ファイルを取得し、その ejb-jar ファイルに含まれているエンタープライズ Bean を、EJB サーバーの特定の CorbaServer にデプロイします。

デプロイヤーは、以下のことを行う必要があります。

- Bean プロバイダーによって宣言されるすべての外部依存関係を解決します。例えば、エンタープライズ Bean で使用されるすべてのリソース・マネージャー接続ファクトリーが、稼働環境に存在することを確実にし、それらをデプロイメント記述子で宣言されるリソース・マネージャー接続ファクトリー参照にバインドする必要があります。

- アプリケーション・アセンブラーによって定義されたアプリケーション・アセンブリー方法に従います。例えば、デプロイヤーは、アプリケーション・アセンブラーによって定義されるセキュリティー役割を、CICS ユーザー・グループおよび外部セキュリティー・マネージャー・プロファイルにマップすることを担当します。

デプロイメント・プロセスは半自動で行われます。デプロイヤーは、役割を実行するために**デプロイメント・ツール**を使用します。デプロイメント・ツールは CICS によって提供されます。

デプロイヤーの出力は、ターゲット稼働環境用にカスタマイズされ、1 つ以上の CorbaServer にデプロイされたエンタープライズ Bean です。

システム管理者:

システム管理者は、論理 EJB サーバーを構成する CICS 領域を、ネットワーク接続と共に構成し、管理することを担当します。また、デプロイされた EJB アプリケーションの実行時の適切な状態の監督も担当します。

エンタープライズ Bean のデプロイの概要

デスクトップ Java Bean は、ワークステーションで開発され、インストールされ、実行されます。サーバー上で実行されるエンタープライズ Bean には、Bean を実行時環境用に準備し、EJB サーバーにインストールするために、**デプロイメント**という追加のステージが必要です。

エンタープライズ Bean は Bean プロバイダーによって生成され、アプリケーション・アセンブラーによってカスタマイズされます。アプリケーション・アセンブラーは、ejb-jar ファイルをカスタマイズするために、 Assembly Toolkit (ATK) などのツール (「*CICS Operations and Utilities Guide*」の The enterprise bean deployment tool, ATK を参照) を使用できます。デプロイヤーに渡されるカスタマイズ済みの ejb-jar ファイルには、次のものが入っています。

- 1 つ以上のエンタープライズ Bean の Java クラス。
- XML で書かれた単一のデプロイメント記述子。これは、各エンタープライズ Bean の次のような特性を記述します。
  - トランザクション属性
  - 環境プロパティー
  - セキュリティー・レベル
  - アプリケーション・アセンブリー情報

また、リソース定義要件などの、CICS に固有の情報も必要です。

デプロイメント・プロセスの概要は次のとおりです。

1. エンタープライズ Bean デプロイメント・ツール ATK などの、デプロイメント・ツール。このツールを使用して、ejb-jar ファイルをデプロイメントに適切なデプロイ可能 JAR ファイルに変換します。変換されたファイルには、ejb-jar ファイルからの XML デプロイメント記述子とエンタープライズ Bean クラスに加えて、EJB コンテナーをサポートして生成される追加クラスが入っています。変換されたファイルは、デプロイ済み JAR ファイルとして z/OS UNIX ファイル・システムに保管されます。

CorbaServer のデプロイ済み JAR ファイル・ディレクトリー (CORBASERVER 定義の DJARDIR オプションで指定された) に、デプロイ済み JAR ファイルを保管します。デプロイ済み JAR ファイル・ディレクトリーは、「**ピックアップ**」**ディレクトリー**とも呼ばれます。CICS はピックアップ・ディレクトリーをスキャンすると、そこで検出した新規または更新済みの各デプロイ済み JAR ファイルの定義を自動的に作成し、インストールします。CICS は、以下のいずれかの方法でピックアップ・ディレクトリーをスキャンします。

- CORBASERVER 定義がインストールされる場合は、自動的にスキャンします
- 明示的な **EXEC CICS** または **CEMT PERFORM CORBASERVER SCAN** コマンドによって指示された場合。
- **PERFORM CORBASERVER SCAN** コマンドをユーザーに代わって発行する、エンタープライズ Bean のリソース・マネージャー (エンタープライズ Bean の RM とも呼ばれます) によって指示される場合。(エンタープライズ Bean のリソース・マネージャーについては、「*CICS Operations and Utilities Guide*」の The Resource Manager for Enterprise Beans を参照してください。)

2. 以下のものに CICS リソース定義が必要です。
   - CorbaServer 実行環境 (CORBASERVER) (同じ CORBASERVER 定義が論理 EJB サーバーの各 CICS AOR にインストールされます)。
   - TCP/IP サービス (IIOP 用)。1 つ以上の TCPIPSERVICE 定義が、論理 EJB サーバーの各 CICS 領域にインストールされます。
   - クライアント IIOP 要求を CICS TRANSID に関連付ける (したがって、セキュリティー、優先順位、モニターなどを含む実行特性のセットに Bean メソッドを関連付ける) ための要求モデル。要求モデルが必要なのは、デフォルトの TRANSID である CIRP が不適切な場合のみです(例えば、IIOP ワークロードをトランザクション ID によって分けることができます)。

     注: CREA CICS 提供トランザクションを使用すると、CorbaServer 内の特定の Bean および Bean メソッドに関連したトランザクション ID を表示できます。トランザクション ID を変更し、変更を適用し、新しい REQUESTMODEL 定義に対する変更を保管できます。
   - デプロイ済み JAR ファイル (DJAR)。それぞれに、デプロイ済み JAR ファイルの z/OS UNIX ファイル名が含まれています。デプロイ済み JAR ファイルを CorbaServer の「ピックアップ」ディレクトリーに保管する場合、CorbaServer がインストールされるときに (または以降のスキャンが行われるときに) DJAR 定義は自動的に作成され、インストールされます。

   注: 272 ページの『論理 EJB サーバーのセットアップ』に、これらの RDO 定義の詳細情報が記載されています。
3. セキュリティー定義が外部セキュリティー・マネージャーに追加されます。これらの定義は、どの役割が特定の Bean およびメソッドを実行できるか、およびどのユーザー ID が各役割に関連付けられるかを指定します。
4. リソース定義が CICS にインストールされます。DJAR 定義をインストールすると、CICS は以下を行います。

- デプロイ済み JAR ファイル (およびそれに入っているクラス) を z/OS UNIX 上の「シェルフ」ディレクトリーにコピーします。**シェルフ・ディレクトリー** は、インストールされたデプロイ済み JAR ファイルのコピーを CICS が保持する場所です。
- デプロイ済み JAR をシェルフから読み取り、その XML デプロイメント記述子を解析し、それに含まれている情報を保管します。

注: デプロイ済み JAR ファイルを CorbaServer の「ピックアップ」ディレクトリーに保管する場合、CorbaServer がインストールされるときに (または以降のスキャンが行われるときに) DJAR 定義は自動的にインストールされます。

5. デプロイされた各 Bean のホーム・インターフェース・クラスへの参照が、外部ネーム・スペースで公開されます。このネーム・スペースは、JNDI を通じてクライアントからアクセス可能です。

   CORBASERVER 定義で AUTOPUBLISH(YES) を指定する場合、DJAR 定義が正常に CorbaServer にインストールされるときに、デプロイ済み JAR ファイルの内容が自動的にネーム・スペースに公開されます。別の方法として、**PERFORM CORBASERVER PUBLISH** または **PERFORM DJAR PUBLISH** コマンドを発行することもできます。

図 14 は、デプロイメント・プロセスを示しています。

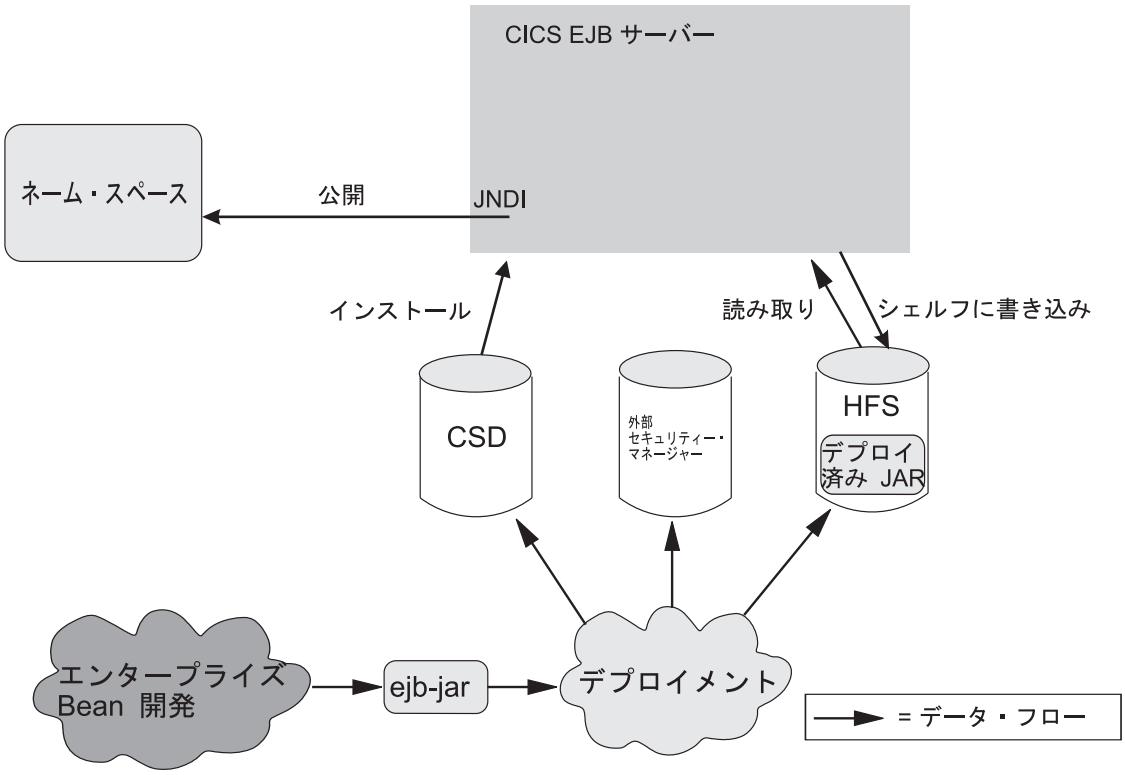


図 14. CICS EJB サーバーへのエンタープライズ Bean のデプロイ: Bean クラスが入っている ejb-jar ファイルでコード生成を実行するために、デプロイメント・ツールが使用されます。変換されたファイルは、デプロイ済み JAR ファイルとして z/OS UNIX に保管されます。デプロイ済み JAR ファイルの RDO 定義が作成され、TCP/IP サービス、要求モデル、および CorbaServer 実行環境の他の定義と共に CICS にインストールされます。セキュリティー定義が外部セキュリティー・マネージャーに作成されます。

EJB サーバーとしての CICS の構成の概要

CICS EJB サーバーには、次の基本コンポーネントが含まれています。

リスナー

リスナーのジョブは、着信 TCP/IP 接続要求を listen する (および応答する) ことです。IIOP リスナーは、特定の TCP/IP ポートで listen し、IIOP **要求受信側**を接続して各接続を処理するように、TCPIPSERVICE リソースによって構成されます。

IIOP 接続がクライアント・プログラムと特定の要求受信側間で確立された後、その接続を介したクライアント・プログラムからの後続の要求はすべて、同じ要求受信側に流れます。

要求受信側

要求受信側は、構造化された IIOP データを分析します。**要求ストリーム**を通じて**要求プロセッサー**に着信要求を渡します。これは内部の CICS ルーティング・メカニズムです。要求のオブジェクト・キーにより、要求が送信される先が新規要求プロセッサーであるか、既存の要求プロセッサーであるかが決まります。

要求が新規要求プロセッサーに送信されなければならない場合、CICS トランザクション ID は、REQUESTMODEL リソースで定義されるテンプレートと要求データとを比較して決定されます (一致する REQUESTMODEL リソースを検出できない場合、デフォルトのトランザクションである CIRP が使用されます)。TRANSID は、要求プロセッサーで使用される実行パラメーターを定義します。

要求プロセッサー

要求プロセッサーは、IIOP 要求の実行を管理するトランザクション・インスタンスです。これは、以下を行います。

- 要求で識別されるオブジェクトの位置を確認します
- エンタープライズ Bean 要求の場合、Bean メソッドを処理するためのコンテナーを呼び出します
- ステートレス CORBA オブジェクトの要求の場合、通常、ORB が要求自体を処理します (ただし、トランザクション・サービスも関係する場合があります)

リスナー、要求受信側、および要求プロセッサーに関する総合的な情報については、420 ページの『IIOP 要求フロー』を参照してください。

271 ページの図 15 は、CICS 論理 EJB サーバーを示しています。この例では、リスナー領域と AOR は別々のグループ内にあり、接続の最適化を使用して、リスナー領域間でクライアント接続のバランスが取られ、分散ルーティングを使用して、AOR 間で OTS トランザクションのバランスが取られます。

この論理サーバーは、1 組の複製されたリスナー領域と 1 組の複製された AOR で構成されます。この例では、動的 DNS 登録を通じた接続の最適化を使用して、リスナー領域間でクライアント接続のバランスを取ります。分散ルーティングを使用して、AOR 間で OTS トランザクションのバランスを取ります。

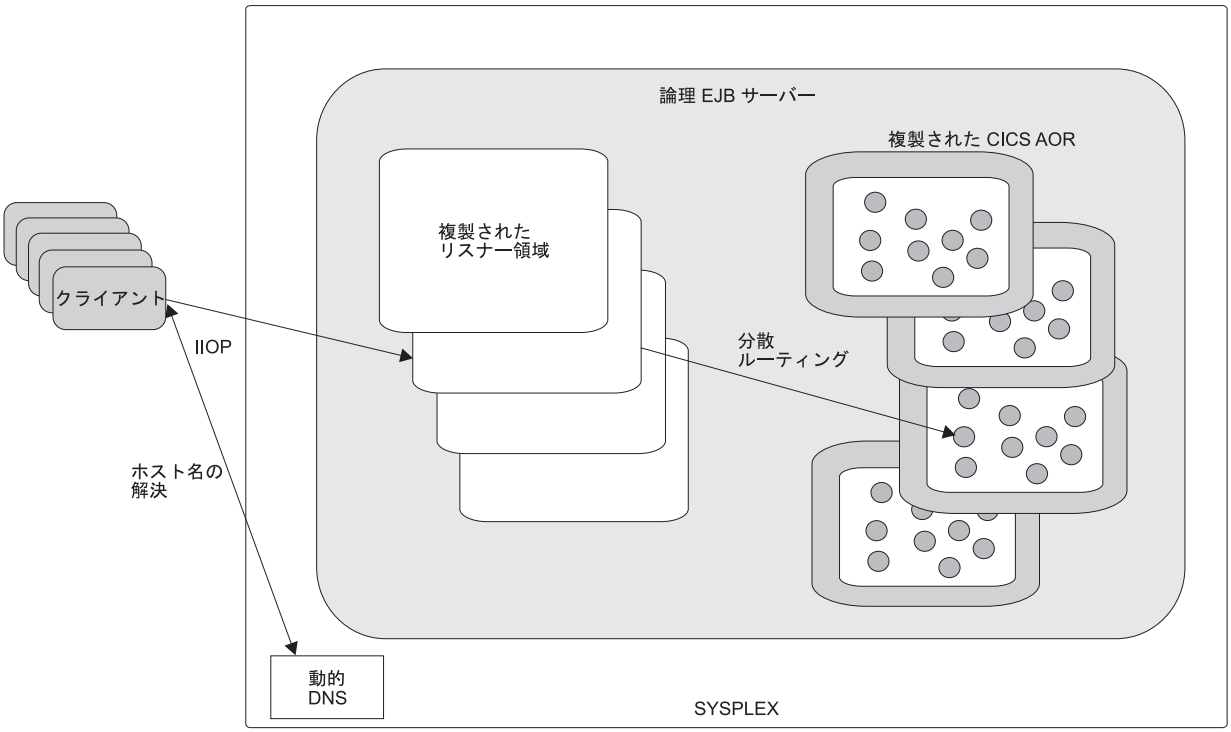


図 15. CICS 論理 EJB サーバー

論理サーバー: シスプレックス内のエンタープライズ Bean:

単一の CICS 領域に 1 つの CICS EJB サーバーを実装できます。

ただし、シスプレックス内では、おそらく、複数の領域で構成されるサーバーの作成が必要になります。複数の領域を使用すると、単一領域の障害の重大度が低くなり、ワークロード・ルーティングの使用が可能になります。CICS 論理 EJB サーバーは、単一の EJB サーバーのように動作するように構成された 1 つ以上の CICS 領域から成ります。

通常、CICS 論理 EJB サーバーは次のコンポーネントから成ります。

- 着信 IIOP 要求を listen するために同一の TCPIPSERVICE 定義で定義される、1 組の複製されたリスナー領域。
- 1 組の複製されたアプリケーション専有領域 (AOR)。各 AOR は、同じように定義された CORBA サーバー内の同じ 1 組のエンタープライズ Bean クラスをサポートします。

注: リスナー領域と AOR は別々にしておくか、またはリスナー AOR に結合することができます。

シスプレックス内のワークロード・ルーティング:

リスナー領域間でクライアント接続を送信し、AOR 間で OTS トランザクションをルーティングすることによって、ワークロード・ルーティングは 2 つのレベルで実装されます。

1. リスナー領域間でクライアント接続をルーティングするには、次のいずれかの方式を使用できます。

- 動的ドメイン・ネーム・システム (DNS) 登録を通じた接続の最適化
- IP ルーティング
- 接続の最適化と IP ルーティングの組み合わせ

動的 DNS 登録を通じた接続の最適化では、例えば、複数の CICS 領域を開始して、(仮想 IP アドレスを使用して) 同じポートで IIOP 要求を listen します。各クライアント IIOP 接続要求には、汎用ホスト名とポート番号が含まれています。各接続要求の汎用ホスト名は、MVS DNS およびワークロード管理 (WLM) サービスによって実 IP アドレスに解決されます。

2. AOR 間で OTS トランザクションをルーティングするには、次のどちらかを使用できます。
   - CICSPlex SM
   - CICS 分散ルーティング・プログラム、DFHDSRP のカスタマイズ・バージョン。

   重要

   分散ルーティング・プログラムを使用する場合、AOR 間で OTS トランザクションの動的ルーティングについて説明するのが好都合です。しかし、厳密に言うと、動的にルーティングされるのは、エンタープライズ Bean と CORBA ステートレス・オブジェクトのメソッド*要求* です。メソッド要求の動的なルーティングと、OTS トランザクションの動的なルーティングとの間には対応関係があります。CICS は、*新規* OTS トランザクションで実行されるメソッドに対する要求にはルーティング・プログラムを起動しますが、*既存の* OTS トランザクションで実行されるメソッドに対する要求には起動しません。これらを、既存の OTS トランザクションが実行される AOR に自動的に送信します。ただし、*OTS トランザクションなし* で実行されるメソッドに対する要求も動的にルーティングできるため、この対応関係は正確ではありません。

   新規 OTS トランザクションと既存の OTS トランザクションとの相違点を理解することが重要です。

   a. *新規* OTS トランザクションは、現行のメソッド呼び出しの前にターゲット論理サーバーがまだ参加していない OTS トランザクションです。必ずしも、メソッド呼び出しの直前に開始された OTS トランザクションである必要はありません。
   b. *既存の* OTS トランザクションは、現行のメソッド呼び出しの前にターゲット論理サーバーが既に参加している OTS トランザクションです。しばらく前に開始された OTS トランザクションではありません。

   例えば、クライアントが OTS トランザクションを開始し、なんらかの作業を行ってから、エンタープライズ Bean でメソッドを呼び出す場合、CICS EJB サーバーに関する限り、これは新規 OTS トランザクションです。これは、このサーバーが以前にこのトランザクションの有効範囲内で呼び出されたことがないからです。クライアントが、OTS トランザクションをコミットする前に、同じターゲット・オブジェクトに対して 2 番目および 3 番目のメソッド呼び出しを行う場合、これらの 2 番目と 3 番目の呼び出しは、既存の OTS トランザクションの有効範囲内で行われます。

論理 EJB サーバーのセットアップ:

エンタープライズ Bean をサポートするように CICS 論理 EJB サーバーをセットアップするには、複数のステップを実行する必要があります。

論理 EJB サーバーをセットアップする前に、論理 EJB サーバー内の領域 (リスナーと AOR の両方) が同じ CICS レベルであることを確認してください。

エンタープライズ Bean をサポートするように CICS 論理 EJB サーバーをセットアップするには、以下のステップを実行します。

1. 1 組の複製された CICS Transaction Server for z/OS, バージョン 4 リリース 2 リスナー領域を作成します。各リスナー領域では、**IIOPLISTENER** システム初期設定パラメーターが YES に設定されていなければなりません。
2. 1 組の複製された CICS Transaction Server for z/OS, バージョン 4 リリース 2 AOR を作成します。各 AOR は以下の基準を満たす必要があります。
   - JNDI を使用するように構成されている
   - 他の AOR と同じ JNDI 初期コンテキストを使用する
   - MRO (ISC ではない) を使用してすべてのリスナー領域に接続されている
   - **IIOPLISTENER** システム初期設定パラメーターを NO に設定して構成されている
3. z/OS UNIX でシェルフ・ルート・ディレクトリーを作成します。例えば、`/var/cicsts/` と呼ばれるディレクトリーを作成できます。これを行うには、CICS によって使用されるディレクトリー・パスへの書き込み権限を持つ z/OS UNIX ユーザー ID が必要です。シェルフ・ディレクトリーを作成したら、そのディレクトリーへの全アクセス権限 (読み取り、書き込み、および実行アクセス権限) を AOR のユーザー ID に付与する必要があります。
4. デプロイ済み JAR ファイル (ピックアップ) ディレクトリーを z/OS UNIX で作成します。例えば、`/var/cicsts/pickup` と呼ばれるディレクトリーを作成できます。AOR には、少なくともそのディレクトリーへの読み取りアクセス権限が必要です。

   AOR に複数の CorbaServer ランタイム環境が含まれる場合は、次のとおりです。
   - CorbaServer ごとに別個のピックアップ・ディレクトリーを作成する必要があります。
   - 各 CorbaServer でサポートされるオブジェクトに、別々のトランザクション ID のセットを割り当ててください。つまり、AOR 内の各 CorbaServer は、別々のトランザクション ID のセットをサポートします。トランザクション ID を Bean メソッドに割り当てるために、REQUESTMODEL 定義を使用します。ステップ 5 を参照してください。
5. 次のリソース定義を作成します。論理サーバー内のすべての領域で共用される CSD でリソース定義を作成するか、それらの領域で使用されるすべての CSD にコピーするか、またはすべての領域に適用される CICSPlex SM Resource Description に追加することができます。オプションとして、以下で説明されているように、CICS スキャン・メカニズム、エンタープライズ Bean のリソース・マネージャー、および CICS 提供のトランザクション CREA を使用して、これらの定義の一部を作成できます。
   - TCPIPSERVICE。

- PROTOCOL オプションで、IIOP を指定します。
- SSL オプションで、NO を指定します。
- AUTHENTICATE オプションで、NO を指定します。この指定を使用すると、このポート上のサービスは、非認証インバウンド IIOP 要求を受け入れます。

- いくつかの REQUESTMODEL 定義。単一領域 EJB サーバーでは、この定義が必要なのは、デフォルトの TRANSID である CIRP が不適切な場合のみです。 しかし、複数領域論理サーバーでは、複数の AOR 全体でメソッドをルーティングしたい場合にこの定義が必要です。CIRP の TRANSACTION 定義は、DYNAMIC(NO) を指定します。例えば、IIOP ワークロードをトランザクション ID によって分けたい場合も、定義が必要です。

  注:

  a. 各 REQUESTMODEL 定義の BEANNAME 属性は、z/OS UNIX 上でデプロイ済み JAR ファイルのデプロイメント記述子にあるエンタープライズ Bean の名前と「一致する」(パターン・マッチングの意味で) 必要があります。CORBASERVER 属性の値は、逐語的またはパターン・マッチングの意味のどちらかで、CORBASERVER 定義の CorbaServer の名前と一致する必要があります。

  b. REQUESTMODEL で指定された TRANSID のトランザクション定義を、CIRP のトランザクション定義からコピーしてください。DYNAMIC 属性を YES に設定してください。他のどの属性でも変更できますが、プログラム名は、acJVMClass が com.ibm.cics.iiop.RequestProcessor である JVM プログラムの名前でなければなりません。

  c. CorbaServer が作動可能である場合、CREA CICS 提供トランザクションを使用すると、CorbaServer 内の特定の Bean および Bean メソッドに関連したトランザクション ID を表示できます。トランザクション ID を変更し、変更を適用し、新しい REQUESTMODEL 定義に対する変更を保管できます。

- CORBASERVER 定義。

  CORBASERVER 定義の HOST オプションの値は、TCPIPSERVICE 定義の HOST または IPADDRESS オプションの値と一致しなければなりません。ただし、TCPIPSERVICE が DNSGROUP の値を指定する場合、CORBASERVER 定義の HOST オプションは、一致する汎用ホスト名を指定しなければなりません。

  UNAUTH オプションで、TCPIPSERVICE 定義の名前を指定します。CorbaServer へのすべてのインバウンド要求が認証されることを予定している場合であっても、CorbaServer を定義するときに常に UNAUTH 属性に値を指定する必要があります。TCPIPSERVICE からのポート番号は、この論理サーバーからエクスポートされる相互運用オブジェクト参照 (IOR) の構成に使用されるため、この値が必要です。CLIENTCERT または SSLUNAUTH オプションの一方または両方に他の TCPIPSERVICE 定義の名前を指定すると、リスナー領域に他のポートでさまざまなタイプの認証済みインバウンド IIOP 要求を listen させることができます。詳しくは、「Resource Definition

Guide」の『CORBASERVER リソース』および「Resource Definition Guide」の『TCPIPSERVICE リソース』を参照してください。

SHELF オプションで、ステップ 3 で作成した z/OS UNIX シェルフ・ディレクトリーの完全修飾名を指定します。CORBASERVER 定義は、論理サーバー内のすべての AOR にインストールされるので、この「ハイレベル」シェルフ・ディレクトリーはすべての AOR で共用されます。各 AOR は、シェルフ・ディレクトリーの下に独自のサブディレクトリーと、その下に CorbaServer のサブディレクトリーを自動的に作成します。

DJARDIR オプションで、ステップ 4 で作成した z/OS UNIX デプロイ済み JAR ファイル・ディレクトリー (ピックアップ・ディレクトリー) の完全修飾名を指定します。ピックアップ・ディレクトリー (または AOR に複数の CorbaServer が含まれている場合は、複数のピックアップ・ディレクトリー) は、シェルフ・ディレクトリーのように、論理サーバー内のすべての AOR で共用されます。各 AOR では、CORBASERVER 定義がインストールされると、CICS は CorbaServer ピックアップ・ディレクトリーをスキャンし、そこで検出したすべてのデプロイ済み JAR ファイルをインストールします。それらのファイルをシェルフ・サブディレクトリーにコピーし、それらのファイルの DJAR 定義を動的に作成し、インストールします。

AUTOPUBLISH(YES) を指定して、DJAR 定義が正常にインストールされるときに CICS が Bean をネーム・スペースに自動的に公開します。

STATUS オプションで、Enabled を指定します。

- CICS で必要な次のファイルの FILE 定義。

  EJB ディレクトリー DFHEJDIR
  要求ストリーム・ディレクトリーが入っているファイルです。これは、論理 EJB サーバー内のすべての領域 (リスナーおよび AOR) によって共用されなければなりません。要求ストリームは、エンタープライズ Bean と CORBA ステートレス・オブジェクトに対するメソッド要求の分散ルーティングで使用されます。DFHEJDIR をリカバリー可能として定義する必要があります。

  EJB オブジェクト・ストア DFHEJOS
  不動態化されているステートフル・セッション Bean のファイルです。論理 EJB サーバー内のすべての AOR で共用されなければなりません。リカバリー不能として定義する必要があります。

  複数の領域全体で DFHEJDIR および DFHEJOS を共用するには、例えば、次のいずれかの方法を使用できます。

  – ファイル専有領域 (FOR) 内のリモート・ファイルとして定義する
  – カップリング・ファシリティー・データ・テーブルとして定義する
  – VSAM RLS を使用する

  サンプル FILE 定義は次のグループに入っています。

  – DFHEJDIR および DFHEJOS の場合、CICS 提供 RDO グループ DFHEJVS に入っています

– DFHEJDIR および DFHEJOS の場合、CICS 提供 RDO グループ DFHEJCF に入っています

DFHEJDIR および DFHEJOS のサンプル VSAM RLS FILE 定義は、CICS 提供 RDO グループ DFHEJVR に入っています。DFHEJVS、DFHEJCF、および DFHEJVR は、デフォルトの CICS 開始グループ・リスト DFHLIST に含まれません。

注: これらのステップでは、論理サーバーに 1 つの CorbaServer しかないことを前提としています。別の CorbaServer を作成するには、2 番目の CORBASERVER 定義と別の TCPIPSERVICE 定義を作成してください。

6. DFHEJDIR および DFHEJOS に基礎の VSAM データ・セットを定義します。CICS では、SDFHINST ライブラリーの DFHDEFDS メンバーに役立つサンプル JCL が提供されています。
7. Assembly Toolkit (ATK) などのデプロイメント・ツールを使用して、1 つ以上の ejb-jar ファイルを使用し、それらでコード生成を実行して、z/OS UNIX でデプロイ済み JAR ファイルを作成します。デプロイ済み JAR ファイルを CorbaServer のピックアップ・ディレクトリーに保管します。
8. すべての CICS 領域を開始します。各リスナー領域で、CSD からインストールされる定義は次のとおりです。
   - TCPIPSERVICE 定義
   - REQUESTMODEL 定義
   - DFHEJDIR のファイル定義

   各 AOR で、CSD からインストールされる定義は次のとおりです。
   - TCPIPSERVICE 定義。
   - REQUESTMODEL 定義。

     ローカル・オブジェクトに対するアウトバウンド要求用に AOR 内の REQUESTMODEL 定義を必要としています。CORBA ステートレス・オブジェクトまたはエンタープライズ Bean が別のオブジェクトを呼び出した場合、そのオブジェクトがローカル AOR 内で使用可能であれば、CICS はリスナー領域に要求を送信しません。その代わりに、CICS は現行タスク内で呼び出されたメソッドを実行するか (「緊密ループバック」)、ローカル AOR 内で別の要求プロセッサーを開始します (「通常ループバック」)。通常ループバックを使用する場合は、新規の要求プロセッサー・タスクで、最初のオブジェクトの呼び出しに使用されたものと同じ REQUESTMODEL を使用することをお勧めします。そうしないと、予測不能な結果が発生する可能性があります。CORBA ステートレス・オブジェクトまたはエンタープライズ Bean がアウトバウンド・コールを行わない場合、AOR 内の REQUESTMODEL は厳密には必要とされません。
   - CORBASERVER 定義。
   - DFHEJDIR および DFHEJOS のファイル定義。

   デプロイ済み JAR ファイルを共用ピックアップ・ディレクトリーに書き込む場合、CorbaServer がインストールされるときに、または以降のスキャンが行われるときに、DJAR 定義は自動的に AOR で作成され、インストールされます。

他の z/OS UNIX ディレクトリーに置くデプロイ済み JAR ファイルに対してのみ、静的 (CSD にインストールされた) DJAR 定義を作成します。

9. 各 AOR では、CORBASERVER 定義がインストールされると、CICS はピックアップ・ディレクトリーをスキャンし、そこで検出したすべてのデプロイ済み JAR ファイルをインストールします。それらのファイルをシェルフ・ディレクトリーにコピーし、それらの DJAR 定義を動的に作成し、インストールします。

   デプロイ済み Jis をインストールさせることができます。これを行う場合、**CORBASERVER PERFORM SCAN** コマンドを発行することによって、CICS に別のスキャンを実行させることができます。このコマンドは、**EXEC CICS**、CEMT マスター端末トランザクション、またはエンタープライズ Bean の Web ベースのリソース・マネージャー (エンタープライズ Bean の RM とも呼ばれます) を使用して発行してください。

10. CORBASERVER 定義で AUTOPUBLISH(YES) を指定したため、DJAR 定義が正常にインストールされると、エンタープライズ Bean のホームが自動的に JNDI ネーム・スペースにバインドされます。

    AUTOPUBLISH(NO) を指定する場合、少なくとも 1 つの AOR で **PERFORM CORBASERVER(CorbaServer_name) PUBLISH** コマンドを発行する必要があります。このコマンドは、**EXEC CICS**、CEMT マスター端末トランザクション、エンタープライズ Bean の RM を使用して、または CICSPlex SM WUI ビューから、発行する必要があります。

11. リスナー領域の DSRTPGM システム初期設定パラメーターに、使用する分散ルーティング・プログラムの名前を指定します。CICSPlex SM を使用している場合は、 CICSPlex SM ルーティング・プログラムの名前、EYU9XLOP を指定します。それ以外の場合はカスタマイズ済みルーティング・プログラムの名前を指定します。DSRTPGM システム初期設定パラメーターについては、「System Definition Guide」の『DSRTPGM システム初期設定パラメーター』を参照してください。

278 ページの図 16 は、CICS 論理 EJB サーバーの定義に必要な RDO 定義を示しています。この図は、リスナー領域で必要な定義、AOR で必要な定義、および両方で必要な定義を示しています。

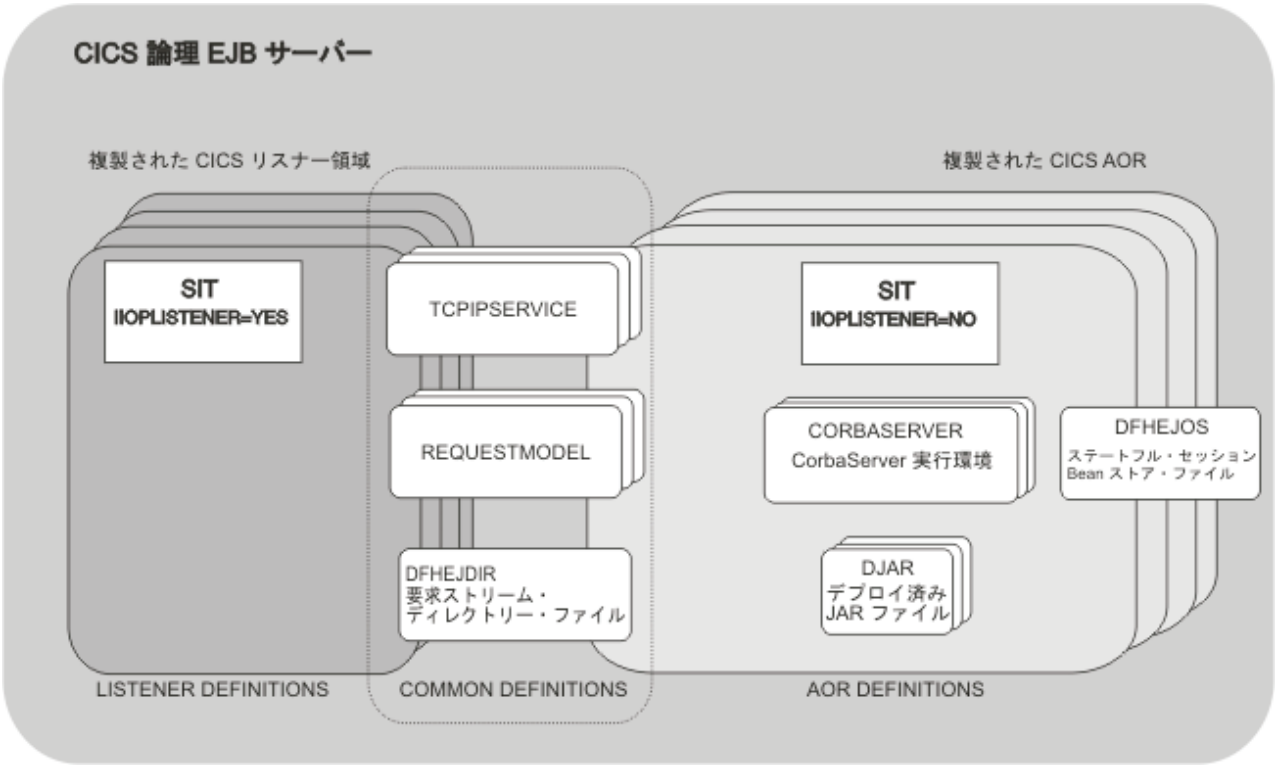


図 16. CICS 論理 EJB サーバー内のリソース定義

エンタープライズ Bean — クライアントが Bean でできること

このセクションでは、クライアント・プログラムがエンタープライズ Bean をどのように使用するかを示すコード・フラグメント例を示しています。

Bean のホームへの参照を取得する:

クライアントは、Bean を使用して何らかの処理を実行するには、Bean のホーム・インターフェースへの参照を取得する必要があります。

これを行うために、クライアントは JNDI を介して既知の名前を検索します。

```
// Obtain a JNDI initial context
Context initContext = new InitialContext();

// Look up the home interface of the bean
Object accountBeanHome = initContext.lookup("JNDI_prefix/AccountBean");
// where:
// 'JNDI_prefix/' is the JNDI prefix on the CORBASERVER definition
// 'AccountBean' is the name of the bean in the XML deployment descriptor

// Convert to the correct type
AccountHome accountHome = (AccountHome)
            PortableRemoteObject.narrow(accountBeanHome,AccountHome.class);
```

ホーム・インターフェースを使用する:

クライアントは、Bean のホーム・インターフェースを使用して、Bean の新規インスタンスを作成し、Bean のインスタンスを削除することができます。

```
// Create two bean instances
Account anAccount = accountHome.create();
Account anotherAccount = accountHome.create("12345");

// Remove a bean instance
accountHome.remove("12345");
```

コンポーネント・インターフェースを使用する:

クライアントは、Bean のコンポーネント・インターフェースを使用して、Bean のメソッドを呼び出し、Bean を削除することができます。

```
// Use the bean
anAccount.deposit(1000000);
// Remove it
anAccount.remove();
```

エンタープライズ Bean — Bean ができること

エンタープライズ Bean は、ライフサイクル管理やセキュリティーなどの多くのサービスを利用できます。これらのサービスは、デプロイメント記述子の設定に基づいて、EJB コンテナーによって暗黙的に提供されます。

これにより、Bean プロバイダーは、これらのサービスを気にせずに Bean のビジネス・ロジックに専念することができます。このセクションでは、Bean が実行できる項目を説明します。

JNDI エントリーを検索する

Bean は JNDI 呼び出しを使用して次のものを検索できます。

- リソースへの参照
- 環境変数
- 他の Bean への参照

リソース・マネージャーにアクセスする

Bean は次のことが可能です。

- リソース・マネージャーへの接続を取得します
- リソース・マネージャーのリソースを使用します
- 接続をクローズします

CICS プログラムにリンクする

Bean は、JCICS または CCI Connector for CICS TS を使用して CICS プログラムにリンクできます。このプログラムは、CICS でサポートされる任意の言語で作成でき、ローカルまたはリモートにすることができます。 Bean プロバイダーは、CCI Connector for CICS TS を使用して、既存の (非 Java) CICS プログラムの機能を利用する Bean を作成できます。

CCI Connector for CICS TS については、 369 ページの『CCI Connector for CICS TS』で説明されています。

ファイルにアクセスする

Bean は JCICS を使用して、ファイルの読み取りとファイルへの書き込みを行うことができます。

他の Bean を呼び出す

Bean は次のことが可能です。

- 他の Bean オブジェクトのホーム・インターフェースとコンポーネント・インターフェースへの参照を取得します
- 別の Bean オブジェクトのメソッドを呼び出します
- 別の Bean オブジェクトから呼び出されます

Bean は、別の Bean オブジェクトのクライアント、別の Bean オブジェクトのサーバー、またはその両方の役目をすることができます。

CICS はエンタープライズ Bean の実行を新規タスクの開始として処理するため、単一の CICS タスク (トランザクションの 1 つのインスタンス) に複数のエンタープライズ Bean を含めることができないことに注意してください。複数のエンタープライズ Bean が含まれるアプリケーションを作成することはできますが、そのアプリケーションは単一の CICS タスクとしては動作しません。

トランザクションを管理する

オプションとして、セッション Bean は、コンテナー管理トランザクションを使用するのではなく、独自の OTS トランザクションを管理することができます。別の方法として、呼び出し元でトランザクションを管理することもできます。

EJB サーバーのセットアップ

この章では、EJB サーバーのセットアップとテストの方法を説明します。

単一領域 EJB サーバーのセットアップ

このセクションでは、単一領域 CICS EJB サーバーのセットアップ方法を説明します。リスナー領域と AOR の両方が単一領域になります。

この最小構成を基本として使用し、 290 ページの『複数領域 EJB サーバーのセットアップ』の説明に従って、複数領域 CICS EJB サーバーを開発できます。

重要

- 誤解のないように、次の前提条件を明確にしておきます。
  1. 基本的な、カスタマイズされていない、CICS Transaction Server for z/OS, バージョン 4 リリース 2 領域から開始します。
  2. EJB サーバー内の CorbaServer 実行環境は 1 つだけになります。
- 最初の EJB サーバーを作成するときには、デフォルトの JVM プロファイル DFHJVMCD を使用することをお勧めします。最初の EJB サーバーを稼働状態にした後に、JVM プロファイルのカスタマイズが必要になることも考えられます。この方法については、 287 ページの『EJB IVP の実行後 — オプションのステップ』 で説明しています。
- このセクションでは、エンタープライズ Bean のデプロイ方法については説明していません。デプロイメントは、 EJB サーバーのセットアップ後に実行される別個のプロセスです。これについては、 351 ページの『エンタープライズ Bean のデプロイ』 で説明しています。
- このセクションの残りの部分は 2 つの部分に分かれています。

– 『EJB IVP を実行する前に』 では、CICS が EJB サーバーとして正しく構成されていること、およびネーム・サーバーが正しくセットアップされていることを確認する EJB インストール検査プログラムを実行するまでの手順を解説します。

  注: デフォルトでは、EJB IVP は Java 1.3 以上で提供される軽量 tnameserv COS ネーム・サーバーを使用します。このため、 IVP の実行前にエンタープライズ品質のネーム・サーバーをセットアップする必要はありません。ただし、「実動用」ネーム・サーバーをセットアップした後で、IVP を使用してサーバーをテストすることもできます。

– 287 ページの『EJB IVP の実行後 — オプションのステップ』 では、EJB サーバーのカスタマイズに使用可能なオプションの手順をいくつか説明しています。

EJB IVP を実行する前に:

このセクションの手順を使用すると、CICS が EJB サーバーとして正しく構成されていることを確認する EJB インストール検査プログラムを実行できます。

このセクションの手順を使用すると、CICS が EJB サーバーとして正しく構成されていることを確認する EJB インストール検査プログラムを実行できます。以下でのアクションが必要です。

1. 使用するネーム・サーバーのタイプに応じて、z/OS または Windows NT
2. z/OS UNIX
3. CICS

z/OS または Windows NT で必要なアクション:

EJB IVP を実行するには、Java Naming and Directory Interface (JNDI) バージョン 1.2 をサポートするネーム・サーバーが必要です。デフォルトでは、IVP は Java 1.3 以上で提供される軽量 tnameserv COS ネーム・サーバーを使用します。

ローカル・ホストで tnameserv を開始するには、z/OS UNIX システム・サービスまたは Windows NT コマンド・プロンプトで次のコマンドを入力します。

```
tnameserv -ORBInitialPort 2809
```

これにより、ネーム・サーバーは TCP/IP ポート 2809 で接続を listen します。このポートが既にシステムで使用中である場合、別のポートで再試行するように求められます。

注: ファイアウォール・ソフトウェアを実行する場合、デフォルトでファイアウォールが、指定されたポートをブロックする可能性があります。ご使用のファイアウォール・ポリシーで、CICS および任意の EJB クライアント・アプリケーションがネーム・サーバーと通信できることを確実にする必要があります。

エンタープライズ品質のネーム・サーバーの選択とセットアップについては、 431 ページの『JNDI 参照の使用可能化』を参照してください。

z/OS UNIX で必要なアクション:

このセクションのタスクを実行するには、CICS によって使用されるディレクトリー・パスへの書き込み権限を持つ z/OS UNIX ユーザー ID が必要です。

このタスクについて

まだ存在していない場合は、z/OS UNIX で以下のディレクトリーを作成します (既に CICS を IIOP サーバーとして構成している場合は、これらのディレクトリーのいくつかが既に存在しています)。z/OS UNIX の名前には大/小文字の区別があることに注意してください。

1. CICS 作業ディレクトリー。各 CICS 領域には作業ディレクトリーが必要です。この名前は、JVM プロファイルの WORK_DIR パラメーターで指定されます。領域の実行に使用される USERID がこのディレクトリーの読み取りと書き込みが可能であるように、ディレクトリー権限を設定する必要があります。詳しくは、Giving CICS regions access to z/OS UNIX System Services を参照してください。
2. シェルフ・ルート・ディレクトリー。シェルフ・ディレクトリーには、任意の名前を指定できます。ただし、/var ディレクトリーの下で作成することをお勧めします。例えば、/var/cicsts/ と呼ばれる z/OS UNIX ディレクトリーを作成できます。シェルフ・ディレクトリーを作成したら、そのディレクトリーへの全アクセス権限 (読み取り、書き込み、および実行) を CICS 領域のユーザー ID に付与する必要があります。この方法については、Giving CICS regions access to z/OS UNIX System Services を参照してください。
3. デプロイ済み JAR ファイル・ディレクトリー (ピックアップ・ディレクトリーとも呼ばれます)。ピックアップ・ディレクトリーには、任意の名前を指定できます。ただし、/var ディレクトリーの下で作成することをお勧めします。例えば、/var/cicsts/pickup と呼ばれる z/OS UNIX ディレクトリーを作成できます。 CICS 領域のユーザー ID に、少なくともそのディレクトリーへの読み取りアクセス権限を付与する必要があります。

   注:

   a. 複数の CorbaServer 実行環境を EJB サーバーにインストールした場合、環境ごとに別々のピックアップ・ディレクトリーを作成する必要があります。

   b. (デプロイ済み JAR ファイルをピックアップ・ディレクトリーからインストールするために) 実動領域でスキャン・メカニズムを使用する場合、セキュリティーの影響に注意してください。特に、DJAR 定義の CICS コマンド・セキュリティーが迂回される可能性です。これを防ぐために、z/OS UNIX のデプロイ済み JAR ファイル・ディレクトリーへの書き込みアクセス権限が付与されるユーザー ID を、DJAR および CORBASERVER 定義の作成と更新のための RACF 権限が付与されたユーザー ID に制限することをお勧めします。

CICS で必要なアクション:

CORBA ステートレス・オブジェクトへのメソッド呼び出しをサポートするために、既に CICS を IIOP サーバーとして構成している場合は、以下のステップの一部を既に完了していることに注意してください。

このタスクについて

1. IBM 64-bit SDK for z/OS, Java テクノロジー・エディション をインストールします。http://www.ibm.com/servers/eserver/zseries/software/java/ でこの製品をダウンロードし、それに関する詳細情報を見つけることができます。
2. IIOP 呼び出しをサポートするように CICS をセットアップします (CICS は、CORBA ステートレス・オブジェクトとエンタープライズ Bean の両方に対するクライアント・メソッド要求をサポートするために同じ RMI-over-IIOP プロトコルを使用します)。この方法については、445 ページの『IIOP 用の CICS のセットアップ』 で説明しています。

   445 ページの『IIOP 用の CICS のセットアップ』を読む場合は、以下のことに注意してください。

   - 単一領域 EJB サーバーはリスナー/AOR の結合であるため、IIOPLISTENER システム初期設定パラメーターで「YES」を指定する必要があります。
   - CICS は、**JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーから、JVM プロファイルをロードします。この値は、CICS 領域で使用される JVM プロファイルが入っているディレクトリーを指定していることを確認してください。
   - 単一領域サーバーを複数領域サーバーのベースとして使用したい場合は、要求ストリーム・ディレクトリー・ファイル DFHEJDIR と EJB オブジェクト・ストア・ファイル DFHEJOS が、複数の領域間で共用できることを確実にしてください。このため、次のいずれかの方法で定義することをお勧めします。
     - ファイル専有領域 (FOR) 内のリモート・ファイルとして定義する
     - カップリング・ファシリティー・データ・テーブルとして定義する
     - VSAM RLS を使用して定義する
   - PROGRAM 定義は、エンタープライズ Bean 自体には不要です。必要な PROGRAM 定義は、要求受信側および要求プロセッサー・プログラムに対する定義のみです。REQUESTMODEL 定義のデフォルト CIRP トランザクションで指定されている、デフォルトの要求プロセッサー・プログラムは DFJIIRP です。要求受信側トランザクションとプログラムである CIRR および DFHIIRRS と同じように、CIRP および DFJIIRP は、提供されたリソース定義グループ DFHIIOP で定義されます。DFHIIOP は、デフォルトの CICS 開始グループ・リストに含まれています。

     デフォルトの DFHJVMCD 以外の JVM プロファイルを使用する場合、要求プロセッサー・プログラムの PROGRAM 定義の JVMPROFILE オプションでプロファイルの名前を指定する必要があります。 (インストールされた PROGRAM 定義で指定された JVM プロファイルから変更するには、CEMT SET PROGRAM JVMPROFILE コマンドを使用することが可能です。ただし、独自の JVM プロファイルを作成する場合は、デフォルトの定義を変更するのではなく、要求プロセッサー・プログラム用に新しい TRANSACTION および PROGRAM 定義を作成することをお勧めします。)
   - CICS がデプロイ済み JAR ファイルの公開に使用するプロファイルを含めて、CORBA アプリケーションまたはエンタープライズ Bean で使用されるプロファイルで、**-Dcom.ibm.cics.ejs.nameserver** システム・プロパティーにネーム・サーバーの場所を指定する必要があります。

ネーム・サーバーの場所の定義について詳しくは、 127 ページの『JVM システム・プロパティー』を参照してください。

- この段階で REQUESTMODEL 定義または DJAR 定義をインストールする必要はありません。理由は次のとおりです。
  - EJB IVP および EJB サンプル・アプリケーションは、デフォルトの REQUESTMODEL トランザクション ID である CIRP を使用します。
  - REQUESTMODEL 定義を最も簡単に作成するには、CICS にエンタープライズ Bean をデプロイした後に、CREA トランザクションを使用します。デプロイメントは、EJB サーバーのセットアップ後に実行される別個のプロセスです。これについては、 351 ページの『エンタープライズ Bean のデプロイ』 で説明しています。
  - DJAR 定義は、通常、デプロイメント時に CICS スキャン・メカニズムによって作成され、インストールされます。

3. 次の CICS リソース定義を作成します。
   - TCPIPSERVICE
   - CORBASERVER

   CICS 提供のサンプル・グループ DFH$EJB には、EJB IVP の実行に適切な TCPIPSERVICE および CORBASERVER 定義が含まれています。独自の環境に適合するように、これらのリソース定義のいくつかの属性を変更する必要があります。これを行うには、CEDA トランザクションまたは DFHCSDUP ユーティリティーを使用します。

   a. このサンプル・グループを独自に選択したグループにコピーします。例えば、次のとおりです。

      ```
      CEDA COPY GROUP(DFH$EJB) TO(mygroup)
      ```

   b. mygroup グループを表示し、以下の属性を適宜変更します。
      - TCPIPSERVICE リソース定義で、必要に応じて PORTNUMBER をご使用のシステム上の適切な TCP/IP ポートに変更します。指定するポート番号は、ネットワーク管理者によって許可されなければなりません。

        注:

        1) 提供された TCPIPSERVICE 定義では、以下の点に注意してください。
           - PROTOCOL オプションは IIOP を指定します。これは、エンタープライズ Bean および CORBA ステートレス・オブジェクトへのメソッド呼び出しに必要なプロトコルです。
           - SSL オプションは NO を指定します。
           - AUTHENTICATE オプションはデフォルトで NO になります。つまり、このポート上のサービスは、非認証インバウンド IIOP 要求を受け入れます。

        2) 290 ページの『複数領域 EJB サーバーのセットアップ』で説明されているように、単一領域サーバーを複数領域サーバーのベースとして使用したい場合は、DNSGROUP オプションに値を指定する必要があります。これにより、複数領域サーバーでは、リスナー領域間でクライアント接続のバランスを取るために、動的 DNS 登録を通じた接続の最適化を使用できます。

3) TCPIPSERVICE 定義の参照情報については、「*CICS Resource Definition Guide*」を参照してください。

- CORBASERVER リソース定義で以下のことを行います。

1) 281 ページの『z/OS UNIX で必要なアクション』のステップ 2 で作成した z/OS UNIX シェルフ・ディレクトリーの完全修飾名を指定するように、SHELF オプションを変更します。

注: 複数領域 EJB サーバーでは、CORBASERVER 定義が論理サーバー内のすべての AOR にインストールされるので、この「ハイレベル」シェルフ・ディレクトリーはすべての AOR で共用されます。各 AOR は、シェルフ・ディレクトリーの下に独自のサブディレクトリーと、その下に CorbaServer のサブディレクトリーを自動的に作成します。

2) 281 ページの『z/OS UNIX で必要なアクション』のステップ 3 で作成した z/OS UNIX デプロイ済み JAR ファイル・ディレクトリー (ピックアップ・ディレクトリー) の完全修飾名を指定するように、DJARDIR オプションを変更します。

注: 複数領域 EJB サーバーでは、ピックアップ・ディレクトリー (または AOR に複数の CorbaServer が含まれている場合は、複数のピックアップ・ディレクトリー) が、シェルフ・ディレクトリーのように、論理サーバー内のすべての AOR で共用されます。

3) HOST を TCP/IP ホスト名に設定します。

注:

1) 提供された CORBASERVER 定義では、以下の点に注意してください。
   - UNAUTH オプションは、TCPIPSERVICE 定義の名前を指定します。

     CorbaServer へのすべてのインバウンド要求が認証されることを予定している場合であっても、CorbaServer を定義するときに常に UNAUTH 属性に値を指定する必要があります。これは、TCPIPSERVICE からのポート番号が、この論理サーバーからエクスポートされる相互運用オブジェクト参照 (IOR) の構成に使用されるためです。CLIENTCERT または SSLUNAUTH オプションの一方または両方に他の TCPIPSERVICE 定義の名前を指定すると、リスナー領域に他のポートでさまざまなタイプの認証済みインバウンド IIOP 要求を listen させることができます。詳しくは、「*CICS Resource Definition Guide*」を参照してください。
   - AUTOPUBLISH オプションは YES を指定します。これにより、DJAR 定義が正常にインストールされるときに CICS が Bean をネーム・スペースに自動的に公開します。
   - STATUS オプションは Enabled を指定します。
2) CORBASERVER 定義の HOST オプションの値は、関連した TCPIPSERVICE リソースの HOST または IPADDRESS オプションの値との互換性がなければなりません。複数領域サーバーでは、リスナー領域間でクライアント接続のバランスを取るのに動的 DNS 登録が使用

される場合、HOST オプションの値は、TCPIPSERVICE 定義の DNSGROUP オプションで指定された汎用ホスト名と一致する必要があります。

3) CORBASERVER 定義の参照情報については、「*CICS Resource Definition Guide*」を参照してください。

c. mygroup グループをインストールして、これらの定義を CICS が認識するようにします。

CORBASERVER 定義がインストールされる場合は、CICS は次のことを行います。

1) DJARDIR オプションで指定されたピックアップ・ディレクトリーをスキャンします
2) ピックアップ・ディレクトリーで検出したすべてのデプロイ済み JAR ファイルをシェルフ・ディレクトリーにコピーします
3) ピックアップ・ディレクトリーで検出したデプロイ済み JAR ファイル (ある場合) の DJAR 定義を動的に作成し、インストールします
4) CORBASERVER 定義が AUTOPUBLISH(YES) を指定するため、DJAR に含まれているすべてのエンタープライズ Bean を JNDI ネーム・スペースに公開します

d. TCPIPSERVICE の状況を OPEN に設定します。

```
CEMT SET TCPIPSERVICE(EJBTCP1) OPEN
```

CICS コンソールで、特に次のようなメッセージが表示されるはずです。

```
DFHEJ0701 CorbaServer EJB1 has been created.
DFHEJ5024 Scan commencing for CorbaServer EJB1, directory being scanned is
          DJARDIR_name.
DFHEJ5025 Scan completed for CorbaServer EJB1, 0 DJars created, 0 DJars
          updated.
DFHEJ1520 CorbaServer EJB1 is now accessible.
DFHSO0107 TCPIPSERVICE EJBTCP1 has been opened on port port_number at IP
          address xxx.xxx.xxx.xxx
```

ここで、

- **DJARDIR_name** は、CorbaServer のデプロイ済み JAR ファイル (「ピックアップ」) ディレクトリーの名前です。
- **port_number** は、CorbaServer で使用される TCP/IP ポートの番号です。
- **xxx.xxx.xxx.xxx** は、CorbaServer の IP アドレスです。

4. JNDI を使用するように CICS をセットアップします。CICS で実行される Java コードが JNDI API 呼び出しを発行できるようにし、CICS がエンタープライズ Bean のホーム・インターフェースへの参照を公開できるようにするには、ネーム・サーバーの場所を指定する必要があります (LDAP ネーム・サーバーの場合、指定が必要な追加情報があります)。ネーム・サーバーの URL とポート番号を **-Dcom.ibm.cics.ejs.nameserver** システム・プロパティーで指定します。

例えば、tnameserv という、Java 1.3 以降で提供される軽量 COS Naming Directory Server を使用するには、次のように指定します。

```
-Dcom.ibm.cics.ejs.nameserver=iiop://tnameserv.yourcompany.com:2809
```

ここで、tnameserv.yourcompany.com は、tnameserv ネーム・サーバーを開始したホストのアドレスです。2809 は、選択したポートです。

エンタープライズ品質の LDAP サーバーを使用する場合は、次のように指定できます。

```
-Dcom.ibm.cics.ejs.nameserver=ldap://demojndi.yourcompany.com:389
```

その他に必要なプロパティー、および LDAP ネーム・サーバーのセットアップ方法については、432 ページの『LDAP サーバーのセットアップ』を参照してください。

標準の COS Naming Directory Server を使用する場合は、次のように指定できます。

```
-Dcom.ibm.cics.ejs.nameserver=iiop://demojndi.yourcompany.com:900
```

WebSphere Application Server バージョン 5 以降で提供される COS Naming Directory Server を使用する場合は、次のように指定します。

```
-Dcom.ibm.cics.ejs.nameserver=iiop://demojndi.yourcompany.com:2809/domain/legacyRoot
```

重要: ネーム・サーバーの場所の定義について詳しくは、127 ページの『JVM システム・プロパティー』で **-Dcom.ibm.cics.ejs.nameserver** プロパティーの説明を参照してください。

デフォルトの要求プロセッサー・プログラムの JVM プロファイルは DFHJVMCD です。このセクションの上記のステップを実行した場合、使用するプロファイル (複数の場合あり) は、**JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリー内になければなりません。

重要: ここでは、単一の CorbaServer 実行環境を含む単一領域 EJB サーバーをセットアップする方法を説明しました。複数のアプリケーションをサポートする実動領域で、それぞれ独自のエンタープライズ Bean のセットを使用する場合は、複数の CorbaServer が必要になることが考えられます。実動領域の保守を容易に行えるように、357 ページの『実動領域でのエンタープライズ Bean の更新』 のガイドラインに従って、 Bean を CorbaServer およびトランザクション ID へ割り振る必要があります。

上記のステップを完了した後、必要に応じて、CICS が EJB サーバーとして正しく構成されていることを確認する EJB インストール検査プログラムを実行できます。EJB IVP の詳細については、301 ページの『EJB IVP の使用』を参照してください。 または、IVP を実行する前に、次のセクションに進むこともできます。

EJB IVP の実行後 — オプションのステップ:

オプションとして、完全な EJB サーバーのセットアップを完了するには、いずれかのサンプル JVM プロファイルをカスタマイズするか、またはデフォルトの JVM プロファイル DFHJVMCD を使用するのではなく、エンタープライズ Bean で使用するための独自の JVM プロファイルを作成することができます。

このタスクについて

DFHJVMCD は、内部 CICS プログラムに使用されるため、限定された方法でしかカスタマイズできません。しかし、他の JVM プロファイルは必要に応じてカスタマイズできます。

102 ページの『プールされた JVM のセットアップ』では、JVM プロファイルの選択とカスタマイズの方法、または必要に応じて、提供されたサンプル・プロファイルのいずれかに基づいた独自の JVM プロファイルの作成方法を示しています。そのセクションの手順に従って、JVM プロファイルのカスタマイズまたは作成を行ってください。

JVM プロファイルをカスタマイズまたは作成したら、そのプロファイルをエンタープライズ Bean で使用するために、以下の手順を実行します。

1. 要求プロセッサー・プログラムの PROGRAM 定義の JVMPROFILE オプションで JVM プロファイルの名前を指定します (デフォルトの要求プロセッサー・プログラム DFJIIRP に提供された PROGRAM リソースは、デフォルト・プロファイル DFHJVMCD を指定します)。

   デフォルトの定義を変更するのではなく、447 ページの『CICS リソースの定義』で説明されているように、要求プロセッサー・プログラム用に独自の TRANSACTION および PROGRAM 定義を作成する必要があります。新しいプロファイルで実行される Bean メソッドに REQUESTMODEL 定義で TRANSACTION の名前を指定してください。
2. **JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーにプロファイルを置きます。

重要: CICS がデプロイ済み JAR ファイルの公開に使用するプロファイルを含めて、CORBA アプリケーションまたはエンタープライズ Bean で使用されるすべての JVM プロファイルまたはオプションのプロパティー・ファイルで、**-Dcom.ibm.cics.ejs.nameserver** システム・プロパティーにネーム・サーバーの場所を指定する必要があります。ネーム・サーバーの場所の定義について詳しくは、127 ページの『JVM システム・プロパティー』を参照してください。

EJB サーバーのテスト

このセクションでは、単一領域 CICS EJB サーバーが正しく構成されていることを確認する方法を説明します。

EJB IVP の実行:

ネーム・サーバーの構成を含めて、CICS EJB 構成を最も簡単にテストする方法は、CICS に付属の EJB インストール検査プログラム (IVP) を実行することです。

IVP は次のもので構成されます。

- z/OS 上の UNIX システム・サービス (USS) で実行されるライン・モード・クライアント・プログラム
- CICS EJB サーバーで実行されるエンタープライズ Bean

IVP を実行するには、281 ページの『EJB IVP を実行する前に』のすべてのステップを完了しておく必要があります。287 ページの『EJB IVP の実行後 — オプショ

ンのステップ』のステップは完了していても、完了していなくてもかまいません。IVP を正常に実行すると、外部プログラムが CICS EJB サーバー上のエンタープライズ Bean を起動できることが確認されます。

EJB IVP の詳細については、 301 ページの『EJB IVP の使用』を参照してください。

EJB「Hello World」サンプルの使用:

「Hello World」は、HTML フォーム、Web サーバーで実行される Java サーブレットと Java Server Pages、および CICS エンタープライズ Bean から成る、シンプルなアプリケーションです。

このアプリケーションはユーザーに入力を要求し、エンタープライズ Bean を使用してユーザーの入力を標準メッセージに付加してから、結果のストリングを表示します。

EJB「Hello World」サンプルを実行するには、 281 ページの『EJB IVP を実行する前に』のすべてのステップを完了しておく必要があります。 287 ページの『EJB IVP の実行後 — オプションのステップ』のステップは完了していても、完了していなくてもかまいません。

EJB「Hello World」アプリケーションの詳細、およびそのインストール方法については、 306 ページの『EJB「Hello World」サンプル・アプリケーション』を参照してください。

EJB Bank Account サンプルの使用:

Hello World サンプルを正常に実行した後、さらに意欲的に試行することができます。

EJB Bank Account サンプルは、エンタープライズ Bean を使用して、CICS 制御情報を Web ユーザーが利用できるようにする方法を示します。データ表からカスタマー情報を抜き出し、ユーザーに戻します。

このサンプルは、HTML フォーム、Web サーバーで実行される Java サーブレットと Java Server Pages、CICS エンタープライズ Bean、2 つの CICS COBOL サーバー・プログラム、およびいくつかの DB2 データ表から成ります。エンタープライズ Bean は CCI Connector for CICS TS を使用して、DB2 データ表にアクセスする CICS サーバー・プログラムにリンクします。

EJB Bank Account サンプルを実行するには、 281 ページの『EJB IVP を実行する前に』のすべてのステップを完了しておく必要があります。 287 ページの『EJB IVP の実行後 — オプションのステップ』のステップは完了していても、完了していなくてもかまいません。

EJB Bank Account アプリケーションの詳細、およびそのインストール方法については、 316 ページの『EJB Bank Account サンプル・アプリケーション』を参照してください。

独自のエンタープライズ Bean の使用:

サンプル・アプリケーションを実行し、CICS EJB サーバーが正しく動作していることを確認した後、おそらく、独自のエンタープライズ Bean を CICS にデプロイすることになります。

この方法の詳細については、351 ページの『エンタープライズ Bean のデプロイ』を参照してください。

複数領域 EJB サーバーのセットアップ

このセクションでは、複数のリスナー領域および複数の AOR から構成される CICS 論理 EJB サーバーのセットアップ方法を説明します。

始める前に

複数領域 EJB サーバーをセットアップするには、280 ページの『単一領域 EJB サーバーのセットアップ』で説明されているとおりに単一領域 EJB サーバーを既に作成していなければなりません。

このタスクについて

複数領域 EJB サーバー内のすべての領域 (リスナーと AOR の両方) が、同じ CICS レベルであることを確認してください。

手順

1. 単一領域サーバーの CICS を複製し、リスナー領域のセットを作成します。 複製されたすべての領域は、単一領域サーバーの CICS システム定義ファイル (CSD) を共用します。オプションとして、必要でなければ、リスナー領域の以下のリソース定義を破棄できます。
   - CORBASERVER
   - DJAR
   - DFHEJOS

   IIOPLISTENER システム初期設定パラメーターの値は YES のままで設定してください。

   注: CICSPlex SM を使用する場合は、すべてのリスナー領域を含む CICS グループ (CICSGRP) を定義できます。この方法には、リソースを個々の領域ではなくグループに関連付けることができるという利点があります (リソース記述を使用します)。グループで領域が追加または削除された場合は、その領域で自動的にリソースが追加または削除されます。
2. 単一領域サーバーの CICS を複製し、AOR のセットを作成します。 (複製されたすべての領域は、単一領域サーバーの CSD を共用します。)

   すべての AOR で同じ JNDI 初期コンテキストを使用する必要があります。

   AOR はリスナー領域ではないため、**IIOPLISTENER** システム初期設定パラメーターの値を「NO」に変更してください。

   注: CICSPlex SM を使用する場合は、すべての AOR を含む CICS グループ (CICSGRP) を定義できます。グループで領域が追加または削除された場合は、その領域で自動的にリソースが追加または削除されます。

293 ページの図 17は、リスナー領域で必要な定義、 AOR で必要な定義、および両方で必要な定義を示しています。

3. MRO (ISC ではない) を使用して、各 AOR をすべてのリスナー領域に接続します。 CICS 領域間で MRO 接続を定義する方法については、「*CICS 相互通信ガイド*」を参照してください。

   CICSPlex SM を使用する場合は、1 つの AOR からすべてのリスナー領域への SYSLINK を定義することにより、必要な CONNECTION および SESSION 定義の数 (および保守コスト) を大幅に削減できます (CICSPlex SM ではリスナーから AOR への相互接続が自動的に作成されます)。他の AOR からの接続のモデルとして SYSLINK を使用してください。

4. EJB サーバー内のすべての領域で EJB ディレクトリー・ファイル DFHEJDIR が共用されていることを確認します。 単一領域 EJB サーバーの DFHEJDIR を推奨された方法で定義してある場合は (リモート・ファイルとして、カップリング・ファシリティー・データ・テーブルとして、または VSAM RLS を使用するものとして)、複数領域サーバーの複製された領域で自動的にファイルが共用されるはずです。

   注: DFHEJDIR ファイルを所有する CICS 領域が、これにアクセスするその他の領域、特に AOR よりも前に開始されるようにしてください。そうしないと、その他の AOR に CORBASERVER および DJAR 定義をインストールする試みは失敗し、メッセージ DFHEJ0736 が発行されます。

5. EJB サーバー内のすべての AOR で EJB オブジェクト保管ファイル DFHEJOS が共用されていることを確認します。 単一領域 EJB サーバーで DFHEJOS を推奨された方法で定義してある場合は、複数領域サーバーの複製されたすべての領域で自動的にファイルが共用されるはずです。 (オプションとして、必要でなければ、リスナー領域から DFHEJOS の定義を削除できます。)

6. リスナー領域間でクライアント接続のバランスを取るために、動的 DNS 登録を通じて接続の最適化を適用します。 このセットアップ方法については、 424 ページの『ドメイン・ネーム・システム (DNS) による接続の最適化』 で説明しています。

7. AOR 間で動的にルーティングされるようにエンタープライズ Bean のメソッド要求を調整します。 次のいずれかの方法を使用できます。

   a. CICSPlex SM。CICSPlex SM を使用してエンタープライズ Bean のメソッド要求をルーティングする方法については、 409 ページの『エンタープライズ Bean での CICSPlex SM』 を参照してください。

   b. CICS 分散ルーティング・プログラム、DFHDSRP のカスタマイズ・バージョン。エンタープライズ Bean および CORBA ステートレス・オブジェクトに対するメソッド要求をルーティングするための分散ルーティング・プログラムの作成方法については、「*CICS Customization Guide*」を参照してください。

   リスナー領域の DSRTPGM システム初期設定パラメーターに、使用する分散ルーティング・プログラムの名前を指定します。CICSPlex SM を使用している場合は、 CICSPlex SM ルーティング・プログラムの名前、EYU9XLOP を指定します。それ以外の場合はカスタマイズ済みルーティング・プログラムの名前を指

定します。DSRTPGM システム初期設定パラメーターについては、「System Definition Guide」の『DSRTPGM システム初期設定パラメーター』を参照してください。

要確認:

a. エンタープライズ Bean のメソッド要求を動的にルーティングするには、REQUESTMODEL 定義に指定されているトランザクションの TRANSACTION 定義で DYNAMIC(YES) を指定する必要があります。REQUESTMODEL 定義に指定されているデフォルト・トランザクション、CIRP は DYNAMIC(NO) として定義されています。CIRP の TRANSACTION 定義のコピーを作成し、DYNAMIC 設定値を変更し、この定義を新しい名前で保管することをお勧めします。その後、REQUESTMODEL 定義に新しいトランザクションを指定します。 (REQUESTMODEL 定義を作成する最も簡単な方法は、CICS でエンタープライズ Bean をデプロイした後に、 CREA トランザクションを使用する方法です。)

b. DTRTRAN システム初期設定パラメーターで指定された「共通の」トランザクション定義 (端末から開始されたトランザクション・ルーティング要求で TRANSACTION 定義が検出されない場合に使用される) がエンタープライズ Bean のメソッド要求に関連付けられることはありません。リスナー領域の中に、要求と一致する REQUESTMODEL 定義がない場合、要求は CIRP トランザクション (DYNAMIC(NO) が指定された) 下で実行されます。

c. 293 ページの図 17では、ローカル・オブジェクトに対するアウトバウンド要求用に AOR 内の REQUESTMODEL 定義を必要としています。 CORBA ステートレス・オブジェクトまたはエンタープライズ Bean が別のオブジェクトを呼び出した場合、そのオブジェクトがローカル AOR 内で使用可能であれば、CICS はリスナー領域に要求を送信しません。その代わりに、CICS は現行タスク内で呼び出されたメソッドを実行するか (「緊密ループバック」)、ローカル AOR 内で別の要求プロセッサーを開始します (「通常ループバック」)。通常ループバックを使用する場合は、新規の要求プロセッサー・タスクでも、最初のオブジェクトの呼び出しに使用されたものと同じ REQUESTMODEL が使用されることが望まれます。そうでないと、予測不能な結果が発生する可能性があります。CORBA ステートレス・オブジェクトまたはエンタープライズ Bean がアウトバウンド・コールを行わない場合、AOR 内の REQUESTMODEL は厳密には必要とされません。

タスクの結果

これらのステップは、各領域に単一の CorbaServer 実行環境を含む、複数領域 EJB サーバーをセットアップする方法を説明しています。複数のアプリケーションをサポートする実動領域で、それぞれ独自のエンタープライズ Bean のセットを使用する場合は、複数の CorbaServer が必要になることが考えられます。実動領域の保守を容易に行えるように、 357 ページの『実動領域でのエンタープライズ Bean の更新』 のガイドラインに従って、 Bean を CorbaServer およびトランザクション ID へ割り振ります。

この図は、リスナー領域で必要な定義、 AOR で必要な定義、および両方で必要な定義を示しています。

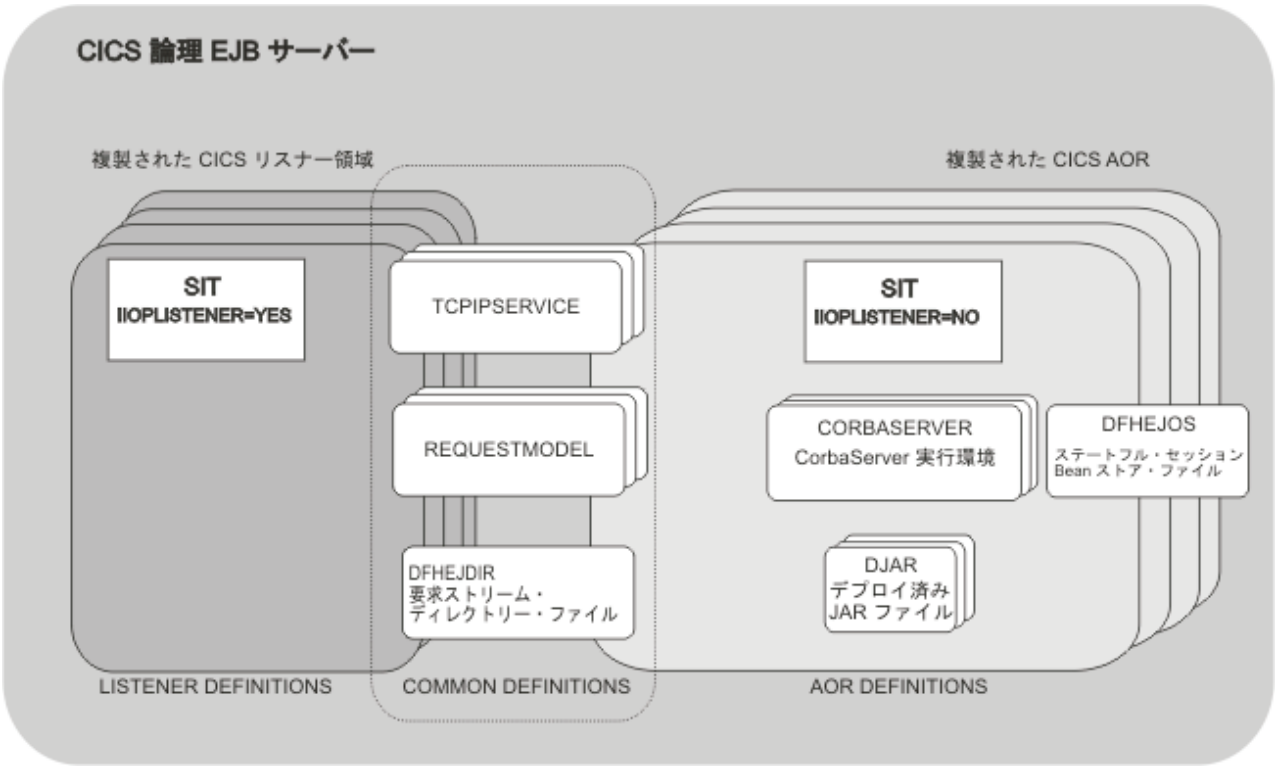


図 17. 複数領域 CICS EJB サーバー内のリソース定義

CICS Transaction Server for z/OS, バージョン 4 リリース 2 への EJB サーバーのアップグレード

このセクションでは、バックレベルの EJB サーバーを CICS TS for z/OS, バージョン 4.2 にアップグレードする方法を説明します。

単一領域 CICS EJB/CORBA サーバーのアップグレード:

単一領域 CICS EJB/CORBA サーバーを CICS Transaction Server for z/OS, バージョン 4 リリース 2 にアップグレードするには、以下のステップを実行します。

手順

1. ワークロードを静止状態にします。
2. 領域をシャットダウンします。
3. アップグレード元のリリースに設定されているアップグレード情報で説明されている標準アップグレード手順に従って、領域を CICS Transaction Server for z/OS, バージョン 4 リリース 2 にアップグレードします。
4. 299 ページの『アップグレードのヒント』を確認します。ここでは、異なるリリースの CICS 間で EJB/CORBA サポートの変更点の一部を説明しています。 また、 280 ページの『単一領域 EJB サーバーのセットアップ』を参照することもできます。ここでは、CICS TS for z/OS, バージョン 4.2 で単一領域 EJB サーバーをセットアップする方法を詳しく説明しています。
5. 領域を再始動します。

6. **PERFORM CORBASERVER**(*CorbaServer_name*) PUBLISH コマンドを実行することによって、サーバーが処理するすべてのエンタープライズ Bean およびステートレス CORBA オブジェクトに対して相互運用オブジェクト参照 (IOR) をリパブリッシュします。 このコマンドは、**EXEC CICS**、CEMT、エンタープライズ Bean のリソース・マネージャーを使用して、または CICSPlex SM WUI ビューから、実行できます。領域内の CorbaServer ごとに個別のコマンドを実行することに注意してください。

複数領域 CICS EJB/CORBA サーバーのアップグレード:

複数領域 CICS EJB/CORBA サーバーを CICS Transaction Server for z/OS, バージョン 4 リリース 2 にアップグレードするには、次のいずれかの方法を使用できます。

このタスクについて

1. **サーバーをシャットダウンし、全領域をアップグレードしてからサーバーを再始動する。**

   この方法は、次の点を除いて 293 ページの『単一領域 CICS EJB/CORBA サーバーのアップグレード』に記載されている方法に大変似ています。

   a. サーバーを再始動する前に全領域を CICS Transaction Server for z/OS, バージョン 4 リリース 2 にアップグレードしなければなりません。この場合も、アップグレード元のリリースに設定されているアップグレード情報で説明されている標準アップグレード手順に従ってください。
   b. 290 ページの『複数領域 EJB サーバーのセットアップ』を参照する必要があります。ここでは、CICS TS for z/OS, バージョン 4.2 で複数領域 EJB サーバーをセットアップする方法を詳しく説明しています。
   c. エンタープライズ Bean およびステートレス CORBA オブジェクトの IOR をリパブリッシュするには、少なくとも 1 つの AOR で PERFORM CORBASERVER(*CorbaServer_name*) PUBLISH コマンドを実行します。 AOR 内の CorbaServer ごとに個別のコマンドを実行することに注意してください。

   この方法の利点は、解決策 2 および 3 に比べ、比較的簡単であることです。主な欠点としては、サーバーのアプリケーションがアップグレード処理の間、使用不可になることです。

2. **別個の CICS TS for z/OS, バージョン 4.2 論理サーバーを作成し、古いバックレベル・サーバーから新規のサーバーにアプリケーションを徐々に移動する。**

   この方法の利点は次のとおりです。

   a. アップグレード処理の間中、アプリケーションが使用可能です。
   b. 最低限の CICS TS for z/OS, バージョン 4.2 サーバー (おそらく、2 つの領域 (リスナー 1 つと AOR 1 つのみの構成)) で開始できます。移動するアプリケーションが増えると、CICS TS for z/OS, バージョン 4.2 サーバーを拡張し、同時にバックレベル・サーバー内の領域数を削減することができます。その結果、リソースが節約されます。
   c. これはおそらく解決策 3 より実施が簡単です。

新規の CICS TS for z/OS, バージョン 4.2 複数領域 EJB サーバーをセットアップするには、 280 ページの『単一領域 EJB サーバーのセットアップ』および 290 ページの『複数領域 EJB サーバーのセットアップ』のすべてのステップを実行してください。

3. **「ローリング・アップグレード」を実行する。**

   「ローリング・アップグレード」では、サーバーを稼働させたまま、一度に 1 つの領域をこれまでの CICS レベルから現行レベルにアップグレードします。

   この方法の利点は次のとおりです。

   a. アップグレード処理の間中、アプリケーションが使用可能です。
   b. 解決策 2 と異なり、どのステージでも追加の CICS 領域をセットアップする必要はありません。

   この方法については、『「ローリング・アップグレード」の実行』に詳細が記載されています。

「ローリング・アップグレード」の実行:

このセクションで説明している混合レベルの運用では、同一の論理サーバー内の異なる CICS 領域は異なるレベルの CICS にあり、ローリング・アップグレードに対してのみ使用されることを目的としています。

重要

これは、一部のインターオペラビリティー・シナリオで障害のリスクを増加させるため、永続的に使用するべきではありません。通常の推奨される運用方式は、論理サーバー内の全領域を同レベルの CICS および Java にする方式です。

このセクションでは、複数領域 CICS EJB/CORBA サーバーを CICS Transaction Server for z/OS, バージョン 4 リリース 2 に「ローリング・アップグレード」する方法について説明します。このプロセスは以下のステップから成ります。

1. 論理サーバーが「ローリング・アップグレード」の基準を満たしていることの確認。『要件』を参照してください。
2. 296 ページの『準備ステップ』
3. 296 ページの『リスナー領域のアップグレード』
4. 297 ページの『AOR のアップグレード』
5. 299 ページの『タイディアップ』

要件:

サーバーは、個別のリスナー およびアプリケーション専有領域で構成されていなければなりません。これはアップグレード・プロセスでは、アプリケーション専有領域 (AOR) の更新の前に全リスナー領域が更新される必要があるためです。

要求受信側と要求プロセッサーの両方の役目をする複合リスナー/AOR を実行している場合は、これは行えません。さらに、すべてのリスナーのアップグレードが完了しないうちにいずれかの AOR をアップグレードすると、要求を受け取るリスナー領域の CICS バージョンによっては、マイグレーション期間中、IIOP クライアント・アプリケーションに一時的な障害が発生する可能性があります。

準備ステップ:
このタスクについて

1. 299 ページの『アップグレードのヒント』を確認してください。
2. CICS TS **2.2** からアップグレードする場合は、すべての CICS TS 2.2 領域に APAR PQ 79565 がインストールされていることを確認します。この APAR が CICS TS 2.2 診断を改善し、CICS TS for z/OS, バージョン 4.2 ワークロードが CICS TS 2.2 領域に到達します。これにより CICS TS 2.2 要求プロセッサー (AOR) も CICS TS for z/OS, バージョン 4.2 要求受信側 (リスナー) から作業を受信できます。
3. すべての CORBASERVER 定義の AUTOPUBLISH オプションを NO に設定します。 IOR を JNDI ネーム・スペースに自動公開するように CorbaServer を設定すると、アップグレード・プロセスが中断する可能性があります。
4. ご使用の論理サーバーの AOR 間でエンタープライズ Bean および CORBA ステートレス・オブジェクトに対するメソッド要求の平衡を取るために分散ルーティング・プログラムを使用している場合は、ご使用のルーティング・プログラムを DYRLEVEL パラメーターを使用するようにカスタマイズしてください。DYRLEVEL は、アップグレードの援助機能です。これには、ターゲット AOR がルーティングされたターゲットを正常に処理するために必要なレベルの CICS が含まれています。(これは、要求を正常に処理するために必要な CICS の**特定の**レベルである (最小のレベルではない) ことに注意してください。) 混合レベルの論理サーバーでは、経路選択 (または経路選択エラー) のためにルーティング・プログラムが起動されると、ルーティング・プログラムは DYRLEVEL 値を使用して、要求をバックレベルの AOR にルーティングするか、CICS TS for z/OS, バージョン 4.2 AOR にルーティングするかを判別できます。

   DYRLEVEL の使用方法の詳細と、分散ルーティング・プログラム作成に関する正確な情報については、「*CICS Customization Guide*」を参照してください。

   EJB サーバーのすべての 領域 (リスナーと AOR の両方) にお客様がカスタマイズしたプログラムをインストールしてください。

   メソッド要求のワークロード・バランシングに CICSPlex SM をご使用の場合、このステップはスキップできます。CICS Transaction Server for z/OS, バージョン 4 リリース 2 で提供された CICSPlex SM ルーティング・プログラムが DYRLEVEL フィールドをチェックし、それに応じて経路要求をチェックします。

リスナー領域のアップグレード:

リスナー領域をアップグレードするには、以下のステップを実行します。

このタスクについて

1. リスナー領域を静止させてから停止します。
2. アップグレード元のリリースに設定されているアップグレード情報で説明されている標準アップグレード手順に従って、この単一リスナー領域を CICS Transaction Server for z/OS, バージョン 4 リリース 2 にアップグレードします。

   重要:

a. CSD を CICS TS **2.2** から CICS TS for z/OS, バージョン 4.2 レベルにアップグレードするときに、アップグレードされる領域以外にいずれかの CICS TS 2.2 領域でその CSD が共用されている場合、DFHCOMPA リソース・グループ (CICS TS for z/OS, バージョン 4.2 で提供されている) をこれらの領域の開始グループ・リストに組み込んでください。DFHCOMPA は、DFJIIRP の定義、デフォルトの要求プロセッサー・プログラムを提供する互換性グループで、CICS TS for z/OS, バージョン 4.2 CSD を共用しているときに CICS TS 2.2 領域で使用できます。

   CICS TS for z/OS, バージョン 4.2 では、DFJIIRP が使用する JVM プロファイルが DFHJVMCD であるため、このステップが必要になります。CICS TS 2.2 では、DFHJVMPR です。

b. この段階では、CICS TS for z/OS, バージョン 4.2 固有の新しいオプションをリソース定義上で使用可能にしないでください。これらのオプションは、バックレベルの AOR では認識されないためです。論理サーバー全体 (リスナー領域と AOR の両方) がアップグレードされるまで、これら新機能を使用するのは待つ必要があります。

CICS TS for z/OS, バージョン 4.2 でのリスナー領域のセットアップに関する正確な情報については、429 ページの『IIOP 用の CICS の構成』を参照してください。

3. リスナーを稼働状態に戻します。これでこの領域の CICS バージョンは最新になりましたが、バックレベルの論理サーバーの一部として引き続き動作機能させることもできます。
4. 論理サーバーの全リスナー領域に対して、ステップ 1 から 3 を繰り返します。

AOR のアップグレード:

エンタープライズ Bean の AOR をアップグレードするには、以下のステップを実行します。

このタスクについて

1. AOR を静止させてから停止します。
2. アップグレード元のリリースに設定されているアップグレード情報で説明されている標準アップグレード手順に従って、この単一 AOR を CICS Transaction Server for z/OS, バージョン 4 リリース 2 にアップグレードします。

   CICS TS 2.2 からアップグレードする場合、この手順の一部には、CorbaServer が使用している JVM プロファイルの更新が含まれます。299 ページの『アップグレードのヒント』に記載されているように、CICS TS 2.3 で導入された JVM プロファイルおよびプロパティー・ファイルの変更点に注意してください。

   重要:

   a. CSD を CICS TS **2.2** から CICS TS for z/OS, バージョン 4.2 レベルにアップグレードするときに、アップグレードされる領域以外にいずれかの CICS TS 2.2 領域でその CSD が共用されている場合、DFHCOMPA リソース・グループ (CICS TS for z/OS, バージョン 4.2 で提供されている) をこれらの領域の開始グループ・リストに組み込んでください。

b. この段階では、CICS TS for z/OS, バージョン 4.2 固有の新しいオプションをリソース定義上で使用可能にしないでください。

3. AOR を稼働状態に戻します。
4. この AOR とリスナー領域で、すべての TCPIPSERVICE をオープン状態にします。
5. CEMT PERFORM DJAR PUBLISH コマンドを使用して、1 つ以上の CICS TS for z/OS, バージョン 4.2 形式のエンタープライズ Bean の IOR をリパブリッシュします。CorbaServer ごとに 1 つ以上のデプロイ済み JAR ファイルを選択してリパブリッシュします。リパブリッシュするデプロイ済み JAR ファイルを選択するときには、以下に注意してください。
   - 単一領域で全作業負荷を処理できる DJAR を選択するようにします。
   - 可能なかぎり、1 つのアプリケーションで使用するすべての Bean を同時にアップグレードしなければなりません。例えば、Bean A が Bean B を呼び出すと分かっている場合は、2 つの Bean を一緒にアップグレードしなければなりません。それが不可能な場合は、Bean A を最初にアップグレードしなければなりません。

     これは、CICS TS 2.2 からアップグレードしようとするときに、Bean を同じ CorbaServer 内ではあるものの、異なるレベルの CICS にある別々の AOR にインストールする場合には特に重要です。これは、両方のオブジェクトが同一の CorbaServer にある場合、CICS TS 2.2 領域は CICS TS for z/OS, バージョン 4.2 領域のオブジェクトの JNDI 検索を行うことができないためです。例えば、CICS TS 2.2 AOR 内の CorbaServer EJB1 にある Bean A は CICS TS for z/OS, バージョン 4.2 AOR 内の CorbaServer EJB1 にある Bean B を検索できません。

     注: A および B が異なる CorbaServers、または同じレベルの CICS にある AOR にインストールされている場合、A および B は個別にアップグレードできます。

   選択した DJAR を、バックレベル AOR が使用しているのと同じ場所で JNDI ネーム・スペースにリパブリッシュします。

   この時点で次のようになります。
   - この AOR は、ワークロードを受け入れる準備できました。
   - 論理サーバーは、バックレベル AOR のプールと CICS TS for z/OS, バージョン 4.2 AOR のプール (現在 1 つの領域のみが入っている) を含みます。
   - ネーム・スペースでリパブリッシュされた Bean の IOR を検索するクライアントはすべて CICS TS for z/OS, バージョン 4.2 形式の新規 IOR を取得します。ユーザーがカスタマイズしたルーティング・プログラムか CICSPlex SM が、このような要求を CICS TS for z/OS, バージョン 4.2 AOR に送信します。
   - リパブリッシュされている Bean に対して、キャッシュに格納され、失効している IOR を持つクライアントは、この Bean を引き続き使用できます。カスタマイズされたルーティング・プログラムまたは CICSPlex SM は、このような古い形式の要求をバックレベル AOR のいずれかに送信します。

注: 多くのアプリケーション・サーバーが JNDI 検索結果をローカルでキャッシュに入れてパフォーマンスを向上させるため、新規の IOR が使用される前にこれらのキャッシュがパージされる必要があることが分かります。ある期間にわたって、リパブリッシュ済みのエンタープライズ Bean に対する要求は、バックレベル AOR のプールから、CICS TS for z/OS, バージョン 4.2 AOR のプールに徐々に移行します。

6. 全論理サーバーの AOR に対して、ステップ 1 から 5 を繰り返します。各 AOR がアップグレードされるに従い、次のようになります。
   - さまざまなエンタープライズ Bean のセットがリパブリッシュされ、徐々により多くの Bean が CICS TS for z/OS, バージョン 4.2 領域のプールでサポートされるようになります。
   - CICS TS for z/OS, バージョン 4.2 プールにはより多くの AOR があるため、リパブリッシュするデプロイ済み JAR ファイルを選択する際、全ワークロードを単一領域で処理できるものを選ぶことはますます重要ではなくなります。

   最終的には、すべての AOR が CICS TS for z/OS, バージョン 4.2 を実行して、ワークロードを 100% 処理するようになります。

タイディアップ:

ローリング・アップグレードを完了するには、以下の最終タスクを実行する必要があります。

このタスクについて

手順

1. 必要な場合には、CORBASERVER 定義の AUTOPUBLISH オプションを YES にリセットします。
2. 使用するすべての CICS TS for z/OS, バージョン 4.2 固有リソース定義オプションを使用可能にします。

タスクの結果

アップグレードのヒント:

このセクションでは、EJB サーバーを CICS TS for z/OS, バージョン 4.2 にアップグレードする際に知っておくべきこととして、一般的なヒントを簡単にリストしています。

これらの全変更内容については、「87 ページの『第 4 章 Java サポートのセットアップ』」に詳細な説明があります。

1. JVM プロファイルは、**JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーに保管されます。
2. CorbaServer で使用されるデフォルト JVM プロファイルは DFHJVMCD です。
3. 「ローリング・アップグレード」プロセス時は、CICS TS for z/OS, バージョン 4.2 固有の新しい属性をリソース定義で使用可能にしないでください。論理サーバー全体 (リスナー領域と AOR の両方) がアップグレードされるまで、これら新機能を使用するのは待つ必要があります。

4. CICS TS for z/OS, バージョン 4.2 の AOR からは、旧リリースの CICS から以前にパブリッシュされたデプロイ済み JAR ファイルを、先に撤回することなく、リパブリッシュできます。 Bean の IOR は、新規リリースの形式に更新されます。**ただし、逆を行うことはできません。**旧リリースの CICS からは、CICS TS for z/OS, バージョン 4.2 の AOR から以前にパブリッシュされたデプロイ済み JAR ファイルをリパブリッシュする前に、まず撤回しておく必要があります。さらに、旧 CICS リリースでは CICS TS for z/OS, バージョン 4.2 IOR の形式が認識されないため、*CICS TS for z/OS, バージョン 4.2 の AOR から撤回する必要があります。*

   どのような理由でも、1 つ以上の AOR のアップグレードをバックアウトする必要がある場合は、このことに注意してください。CICS TS for z/OS, バージョン 4.2 の AOR からパブリッシュされたエンタープライズ Bean の IOR を、旧レベルの CICS に戻す (その結果、バックレベル AOR にもう一度ルーティングできるように) 必要がある場合、以下の手順を実行する必要があります。

   a. デプロイ済み JAR ファイルを CICS TS for z/OS, バージョン 4.2 AOR から撤回する
   b. デプロイ済み JAR ファイルをバックレベル AOR から公開する

   最初に Bean を撤回せずにリパブリッシュをするか、間違ったレベルの CICS から Bean を撤回しようとすると、結果として `InvalidUserKeyException: Bad version number` 例外になります。

潜在的な問題:

1. EJB サーバーを CICS TS for z/OS, バージョン 4.2 にアップグレードした後、一部のクライアントには、キャッシュに格納され、失効している IOR で、以前のサーバーを指す IOR が存在する場合があります。これは、一部のアプリケーション・サーバーは JNDI 検索結果をローカルでキャッシュに入れてパフォーマンスを向上させるためです。新規の IOR が使用される前に、これらのキャッシュがパージされる必要があることが分かります。
2. CICS TS for z/OS, バージョン 4.2 を含めて、CICS TS 2.3 以降は GIOP 1.2 をサポートするのに対して、CICS TS 2.2 は GIOP 1.1 しかサポートしません。CICS TS 2.2 領域で GIOP 1.2 のメッセージを受信した場合、メッセージはリジェクトされます。CICS がサポートする GIOP の最高バージョンは CICS がパブリッシュする IOR に格納されるため、通常の状態ではこのようなことは起こりません。クライアントが、所定のサーバーが GIOP 1.1 しかサポートしないことを認識している場合、そのサーバーと通信する際により新しいものを使用しようとすることはありません。これは、CICS TS for z/OS, バージョン 4.2 が GIOP メッセージを CICS TS 2.2 に送信できることを意味します。

   クライアントが CICS TS for z/OS, バージョン 4.2 (または CICS TS 3.1 または CICS TS 2.3) と通信していると認識しているにもかかわらず、メッセージが CICS TS 2.2 領域にルーティングされている場合にのみ問題が発生します。これは CICS TS 2.2 および CICS TS for z/OS, バージョン 4.2 領域が、同一論理サーバー内の兄弟要求プロセッサー (AOR) としてセットアップされている場合のみ発生します。(これは、CICS では混合レベルの論理サーバーが推奨されない理由の 1 つです。) 「ローリング・アップグレード」中、論理サーバーは当然混合レベルの要求プロセッサーを含みます。しかし、 295 ページの『「ローリン

グ・アップグレード」の実行』 のステップに従った場合には、問題 (CICS TS 2.2 領域で GIOP 1.2 のメッセージが受信される) は発生しません。

3. CICS TS for z/OS, バージョン 4.2 を含めて、CICS TS 2.3 以降は、CICS TS 2.2 とは異なる形式の IOR を使用します。CICS TS for z/OS, バージョン 4.2 を対象とした GIOP 1.1 メッセージが CICS TS 2.2 領域にルーティングされた場合、CICS TS 2.2 領域は不明の IOR 形式が使用されていることが原因で要求をリジェクトします。EJB/CORBA サーバー内の全領域が同レベルの CICS および Java にある場合、このエラーが発生することはありません。

   「ローリング・アップグレード」中、論理サーバーは当然混合レベルの領域を含みます。しかし、 295 ページの『「ローリング・アップグレード」の実行』のステップに従った場合、この問題は発生しません。

EJB IVP の使用

EJB インストール検査プログラム (IVP) は、CICS インストーラーが CICS EJB 環境の検査に使用できる小規模アプリケーションです。

EJB IVP は、Web サーバーを使用する必要がないクライアント・プログラムを使用します。IVP は次のもので構成されます。

- z/OS 上の UNIX システム・サービスで実行されるライン・モード・クライアント・プログラム
- CICS EJB サーバーで実行されるステートレス・セッション・エンタープライズ Bean

IVP は次のものをテストします。

- CICS JVM (その再使用可能性を含む)。
- オプションとして、エンタープライズ・レベルの「実」ネーム・サーバー (デフォルトで、IVP は Java で提供される軽量 `tnameserv` COS ネーム・サーバーを使用します)。
- EJB サーバーが基本的なエンタープライズ Bean を実行できるかどうか。
- z/OS UNIX の設定 (ファイル・アクセス権限を含む)。

構成された後、クライアントは以下の操作を実行します。

1. JNDI 検索を実行して、JNDI ネーム・スペース内の特定エンタープライズ Bean への公開された参照を検出します
2. CICS 内でエンタープライズ Bean の新しいインスタンスを作成します
3. Bean インスタンスでリモート・メソッドを呼び出します

EJB IVP の前提条件

EJB IVP を実行する前に、次のリソースが必要です。

- UNIX システム・サービスのユーザー ID およびファイル・エディター。
- CICS EJB サーバー。このセットアップ方法については、 280 ページの『単一領域 EJB サーバーのセットアップ』で説明しています。
- Java Naming and Directory Interface (JNDI) バージョン 1.2 以降をサポートするネーム・サーバー。エンタープライズ品質のネーム・サーバーをセットアップする方法については、 431 ページの『JNDI 参照の使用可能化』で説明しています。または、Java で提供される軽量 `tnameserv` COS ネーム・サーバーを使用できます。

注:

1. 上記の前提条件では、単一領域 CICS EJB サーバーをテストすることを前提としています。
2. IVP を実行するには、 281 ページの『EJB IVP を実行する前に』のステップを完了しておく必要があります。
3. 開始する前に、ご使用の TSO セッションのストレージ・サイズが 6000 KB 以上であることを確認してください。ストレージ・サイズを増やすには、標準の TSO ログオン画面で SIZE フィールドの値を変更してください。

EJB IVP のインストール

EJB をインストールするには、z/OS UNIX および CICS をセットアップする必要があります。z/OS UNIX システム・サービスで、クライアントを構成する必要があります。

EJB IVP 用の z/OS UNIX のセットアップ:

IVP では、EJB「Hello World」サンプル・アプリケーションと同じ CICS エンタープライズ Bean を使用します。

このサンプルについては、 306 ページの『EJB「Hello World」サンプル・アプリケーション』で説明しています。したがって、z/OS UNIX で、`HelloWorldEJB.jar` デプロイ済み JAR ファイルを EJB サンプル・ディレクトリーから、 281 ページの『EJB IVP を実行する前に』で作成したデプロイ済み JAR ファイル (「ピックアップ」) ディレクトリーにコピーする必要があります。

注: エンタープライズ Bean のソース・コードと実行可能コードはどちらも、`HelloWorldEJB.jar` ファイル内にあります。

サンプル・ディレクトリーは `/usr/lpp/cicsts/cicsts42/samples/ejb/helloworld` です。ここで、`/usr/lpp/cicsts/cicsts42` は、z/OS UNIX 上の CICS ファイルのインストール・ディレクトリーです。

z/OS UNIX の名前には大/小文字の区別があることに注意してください。

CICS のセットアップ:

EJB IVP を実行する前に、以下の CICS セットアップ・タスクを実行する必要があります。

このタスクについて

1. EJB 役割ベースのセキュリティーが CICS 領域でアクティブである場合、IVP の実行前にオフにしておく必要があります。すなわち、SEC と XEJB の両方のシステム初期設定パラメーターで現在、「YES」が指定されている場合、XEJB を「NO」に設定して、CICS を再始動する必要があります。
2. CICS 提供のサンプル・リソース・グループ DFH$EJB には、IVP の実行に適切な TCPIPSERVICE および CORBASERVER 定義が含まれています。独自の環境に適合するようにこれらのリソース定義のいくつかの属性を変更し、変更された定義を CICS にインストールする必要があります。EJB サーバーのセットアッ

プ・タスクの一環として、既にこれを行っているはずです。まだ行っていない場合は、 282 ページの『CICS で必要なアクション』にあるステップバイステップの手順に従ってください。

3. CEMT PERFORM CORBASERVER(EJB1) SCAN コマンドを実行します。

   CICS は次のことを行います。

   a. CorbaServer 定義の DJARDIR オプションで指定されたピックアップ・ディレクトリーをスキャンします
   b. ピックアップ・ディレクトリーで検出した HelloWorldEJB.jar デプロイ済み JAR ファイルを、シェルフ・ディレクトリーにコピーします
   c. HelloWorldEJB.jar の DJAR 定義を動的に作成し、インストールします。
   d. CORBASERVER 定義が AUTOPUBLISH(YES) を指定するため、HelloWorldEJB.jar に含まれているエンタープライズ Bean を JNDI ネーム・スペースに公開します

4. CorbaServer のセットアップ中にまだ行っていない場合は、TCPIPSERVICE の状況を OPEN に設定します。

   ```
   CEMT SET TCPIPSERVICE(EJBTCP1) OPEN
   ```

   CICS コンソールで、特に次のようなメッセージが表示されるはずです。

   ```
   DFHEJ5024 Scan commencing for CorbaServer EJB1, directory being scanned is
             DJARDIR_name.
   DFHEJ5030 New DJar HelloWorldEJB is being created during a scan against
             CorbaServer EJB1.
   DFHEJ0901 DJar HelloWorldEJB within CorbaServer EJB1 has been created.
   DFHEJ5025 Scan completed for CorbaServer EJB1, 1 DJars created, 0 DJars updated.
   DFHEJ5032 DJar HelloWorldEJB is having its contents automatically published to
             the namespace.
   DFHEJ5009 Published bean HelloWorld to JNDI server
             iiop://nameserver.location.company.com:2809 at location samples.
   DFHEJ1540 DJar HelloWorldEJB and the Beans it contains are now accessible.
   ```

   ここで、

   - **DJARDIR_name** は、CorbaServer のデプロイ済み JAR ファイル (「ピックアップ」) ディレクトリーの名前です。
   - **iiop://nameserver.location.company.com:2809** は、ネーム・サーバーの URL とポート番号です。この例では、COS ネーム・サーバーが使用されます。

クライアントの構成:

クライアント・アプリケーションのソース・コードは、HelloWorldCLI.jar ファイルに入っています。

このタスクについて

z/OS UNIX システム・サービスで、以下の手順を実行する必要があります。

1. runEJBIVP スクリプトを作業ディレクトリーにコピーします。オリジナルの runEJBIVP スクリプトが、IVP サンプルと一緒に次のディレクトリーに配置されます。

   ```
   /usr/lpp/cicsts/cicsts42/samples/ejb/helloworld
   ```

ここで、cicsts42 は、z/OS UNIX 上の CICS ファイル用のインストール・ディレクトリーです。

2. 次のように runEJBIVP スクリプトのコピーを編集します。クライアントが JNDI ネーム・スペース内で公開されたエンタープライズ Bean の位置を確認できるようにするために、これが必要です (標準的なクライアントには、CICS JVM プロファイルへのアクセス権限がありません)。

   a. スクリプト内のコメントで示されているとおりに、JAVA_HOME 変数を IBM SDK 6.0.1 インストール・ディレクトリーに変更します。変更される行は次のとおりです。

      ```
      JAVA_HOME=/usr/lpp/<Java SDK java installation directory>/J6.0.1_64
      ```

   b. スクリプト内のコメントで示されているとおりに、CICS_HOME 変数を z/OS UNIX 上の CICS ファイル用のインストール・ディレクトリーに変更します。変更される行は次のとおりです。

      ```
      CICS_HOME=/usr/lpp/cicsts/<CICS installation directory>
      ```

   c. スクリプト内のコメントで示されているとおりに、JNDI_PROVIDER_URL 変数をネーム・サーバーの URL とポート番号に変更します。変更される行は次のとおりです。

      ```
      JNDI_PROVIDER_URL=iiop://nameserver.location.company.com:2809
      ```

      上記の行では、Java 1.3 以降で提供される軽量 COS Naming Directory Server である tnameserv などの COS ネーム・サーバーを使用していること、およびそのネーム・サーバーがポート 2809 で listen するように構成されていることを前提としています。

      例えば、ポート 900 で listen するように構成された COS ネーム・サーバーを使用している場合は、次のように指定できます。

      ```
      JNDI_PROVIDER_URL=iiop://nameserver.location.company.com:900
      ```

      myworkstation.acme.com という名前のワークステーションで、ポート 2809 で listen するように構成された tnameserv ネーム・サーバーを使用している場合、次のように指定します。

      ```
      JNDI_PROVIDER_URL=iiop://myworkstation.acme.com:2809
      ```

      tnameserv プログラムを開始するには、ワークステーションのコマンド・プロンプトで次のコマンドを入力します。

      ```
      tnameserv -ORBInitialPort 2809
      ```

      ポート 2809 で listen するように構成された、WebSphere Application Server バージョン 5 以降で提供される COS Naming Directory Server を使用する場合は、次のように指定します。

```
JNDI_PROVIDER_URL=iiop://nameserver.location.company.com:2809/domain/legacyRoot
```

      LDAP ネーム・サーバーを使用する場合、プロトコルは、iiop ではなく、ldap でなければなりません。ポート番号は 389 です。例えば、次のとおりです。

      ```
      JNDI_PROVIDER_URL=ldap://nameserver.location.company.com:389
      ```

d. LDAP ネーム・サーバーを使用する場合、スクリプト内のコメントで示されているとおりに、LDAP_CONTAINERDN および LDAP_NODEROOTDN 変数を変更します。

   COS ネーム・サーバーを使用する場合、これらのプロパティーは無視されます。

e. 必要に応じて、スクリプト内のコメントで示されているとおりに、INITIAL_CONTEXT_FACTORY 変数を変更します。通常、このプロパティーをデフォルトのままにすることができます。ただし、一部の JNDI サービス・プロバイダーには、デフォルトの初期コンテキスト・ファクトリーを使用してアクセスできません。例えば、JNDI プロバイダーとして WebSphere Application Server を使用している場合、この変数を `com.ibm.websphere.naming.WsnInitialContextFactory` に設定する必要があります。

f. 推奨された方法で CorbaServer をセットアップし、IVP をインストールした場合、CORBASERVER_JNDI_PREFIX および BEAN_NAME 変数は既に正しい値に設定されています。スクリプト内のコメントを参照してください。

EJB IVP の実行

EJB インストール検査プログラムを実行するには、以下のステップを実行する必要があります。

このタスクについて

手順

1. ネーム・サーバーが実行していることを確認します。
   a. ローカル・ホストで `tnameserv` を開始するには、z/OS UNIX システム・サービスまたは Windows コマンド・プロンプトで次のコマンドを入力します。

      ```
      tnameserv -ORBInitialPort 2809
      ```

      これにより、`tnameserv` は TCP/IP ポート 2809 で接続を listen します。

2. `./runEJBIVP` を入力して、z/OS UNIX システム・サービスの作業ディレクトリーから IVP クライアント・プログラムを実行します。 z/OS UNIX システム・サービス端末で、次のようなメッセージが表示されるはずです。

   ```
   CICS EJB IVP: Querying the Java SDK level
   java version "1.6.0"
   Java(TM) SE Runtime Environment (build pmz6460_26-20110218_01)
   IBM J9 VM (build 2.6, JRE 1.6.0 z/OS s390x-64 20110217_75924
   (JIT enabled, AOT enabled)
   J9VM - R26_Java626_GA_20110217_1713_B75924
   JIT - r11_20110215_18645
   GC - R26_Java626_GA_20110217_1713_B75924
   J9CL - 20110217_75924)
   JCL - 20110207_01
   CICS EJB IVP: Starting the EJB client program
   HelloWorld client program started
   Performing JNDI lookup using CosNaming
   Testing the following location: samples/HelloWorld
   Located home interface for HelloWorld bean
   You said: Hello from CICS EJB IVP client
   HelloWorld client program ended
   CICS EJB IVP: Completed successfully
   ```

注:

a. この例では、COS ネーム・サーバーが使用されました。LDAP ネーム・サーバーを使用する場合、同じようなメッセージが生成されます。

b. `javax.naming.CommunicationException` を取得する場合、その原因は、MVS ホスト名が `tcpip.data` ファイルで正しくないためである可能性があります。 MVS システムのエントリーを `/etc/hosts` ファイルに追加して、この問題を修正できる場合があります。手引きが必要な場合は、MVS のマニュアルを参照してください。

JVM stdout ファイルに次のメッセージが表示されます。

```
CICS EJB hello world sample called with string: Hello from CICS EJB IVP client
```

3. IVP を実行した後、次のステップを実行する必要があります。

a. `mygroup` で作成したリソース定義を破棄します。

b. IVP の実行前に EJB 役割ベースのセキュリティーをオフにした場合は、オンに戻します。これを行うには、**XEJB** システム初期設定パラメーターを「YES」に設定して CICS を再始動します。

サンプル EJB アプリケーションの実行

サンプル EJB アプリケーションには CICS EJB サーバーが必要です。

重要

サンプルをインストールしようとする前に、280 ページの『EJB サーバーのセットアップ』で説明されているとおりに CICS を構成する必要があります。

CICS は次のサンプル EJB アプリケーションを提供します。

EJB インストール検査プログラム (IVP)
CICS EJB 環境とネーム・サーバーのテストに使用できるシンプルなアプリケーション。Web サーバーは不要です。 301 ページの『EJB IVP の使用』を参照してください。

EJB「Hello World」サンプル
CICS、ネーム・サーバー、および Web サーバーを含めて、EJB 環境のテストに使用できるシンプルなアプリケーション。『EJB「Hello World」サンプル・アプリケーション』を参照してください。

EJB Bank Account サンプル
エンタープライズ Bean を使用して、既存の CICS 制御情報を Web ユーザーが利用できるようにする方法を示す、より複雑なアプリケーション。 316 ページの『EJB Bank Account サンプル・アプリケーション』を参照してください。

EJB「Hello World」サンプル・アプリケーション

「Hello World」は、CICS、ネーム・サーバー、および Web サーバーを含めて、EJB 環境のテストに使用できるシンプルなアプリケーションです。

EJB「Hello World」サンプルが行うこと:

このサンプル・アプリケーションは、入力を要求し、その入力を標準メッセージに付加し、結果のストリングを表示します。

このサンプルは次のもので構成されます。

- HTML フォーム。
- J2EE 準拠の Web アプリケーション・サーバーで実行される Java サーブレットおよび JavaServer Pages (JSP)。
- CICS EJB サーバーで実行されるエンタープライズ Bean。

このサンプルの仕組みは次のとおりです。

1. ユーザーは Web ブラウザーからこのアプリケーションを開始します。フォームが表示されます。
2. そのフォームで、語句を入力するようにユーザーに求められます。ユーザーが SUBMIT ボタンを押すと、サーブレットが起動されます。
3. サーブレットは以下のことを行います。
   a. JNDI ネーム・スペース内のエンタープライズ Bean への参照を検索します
   b. CICS 内でエンタープライズ Bean の新しいリモート・インスタンスを作成します
   c. Bean インスタンスでメソッドを呼び出し、ユーザーが入力した語句を入力として渡します
4. エンタープライズ Bean はユーザーの語句をストリング「You said」に付加し、結果をサーブレットに戻します。
5. サーブレットは JavaServer Page を使用して、ユーザーの Web ブラウザーに結果を表示します。

308 ページの図 18 は、このサンプル・アプリケーションのコンポーネントを示しています。このサンプルの主要素は、Java サーブレットとエンタープライズ Bean です。この例では、サーブレットは Windows サーバー上の Web アプリケーション・サーバーで実行されます。COS ネーム・サーバーが使用されます。その他の構成も可能です。例えば、LDAP ネーム・サーバーを使用できます。または、COS ネーム・サーバーが、サーブレットと同じアプリケーション・サーバーでホスティングされないことも可能です。

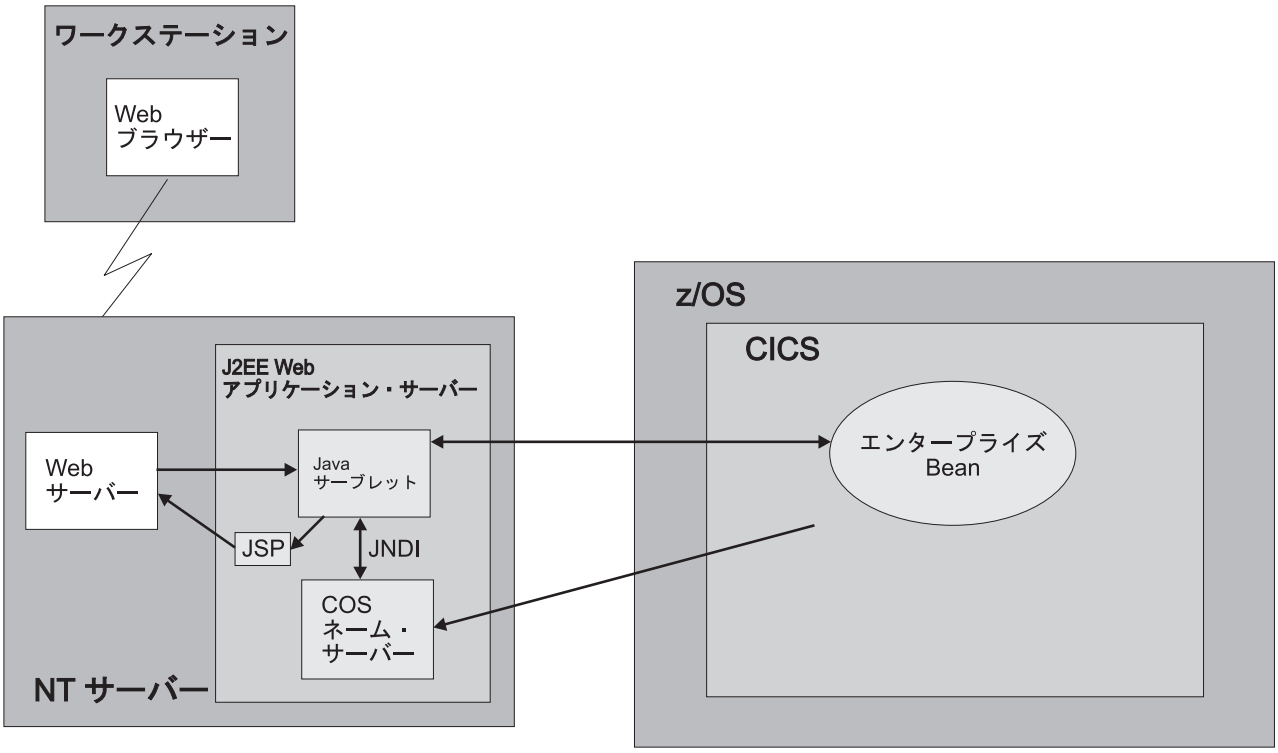


図 18. EJB「Hello World」サンプル・アプリケーションの概要

EJB「Hello World」サンプルの前提条件:

EJB「Hello World」サンプルを実行するには、次のリソースが必要です。

- CICS EJB サーバー。このセットアップ方法については、 280 ページの『EJB サーバーのセットアップ』で説明しています。
- J2EE バージョン 1.2.1 以降をサポートする Web アプリケーション・サーバー。WebSphere Application Server を使用する場合、このサンプルには WebSphere Application Server バージョン 4 以降が必要です。
- Java Naming and Directory Interface (JNDI) バージョン 1.2 以降をサポートするネーム・サーバー。このセットアップ方法については、 281 ページの『z/OS または Windows NT で必要なアクション』で説明しています。

EJB「Hello World」サンプルに提供されているコンポーネント:

EJB「Hello World」サンプルには、以下のファイルが提供されています。

表 16. EJB「Hello World」サンプルに提供されているコンポーネント

Filename (ファイル名)	タイプ	デフォルトの場所	コメント
CICSHelloWorld.ear	EAR ファイル	z/OS UNIX サンプル・ディレクトリー。注を参照。	サンプル・アプリケーションの Web コンポーネント。Java サーブレット・クラスとソース・ファイル、HTML と JSP。
DFH$EJB	リソース定義グループ	CSD	このサンプル・アプリケーションに必要な CICS リソース定義が入っています。
HelloWorldCLI.jar	JAR ファイル	z/OS UNIX サンプル・ディレクトリー。注を参照。	サーブレットで必要なクライアント EJB スタブ。

表 16. EJB「Hello World」サンプルに提供されているコンポーネント (続き)

Filename (ファイル名)	タイプ	デフォルトの場所	コメント
HelloWorldEJB.jar	デプロイ済み JAR ファイル	z/OS UNIX サンプル・ディレクトリー。注を参照。	Java クラス、ソース・ファイル、デプロイメント記述子に加えて、CICS エンタープライズ Bean をサポートするクラス。ソース・コードの変更が必要な場合を除いて、解凍する必要はありません。
readme.txt	テキスト・ファイル	z/OS UNIX サンプル・ディレクトリー。注を参照。	何を組み込めるか: 1. WebSphere Application Server に EJB「Hello World」サンプルの Web コンポーネントをインストールするためのステップバイステップの説明。 2. ヒントおよびデバッグ情報。
注: デフォルトの z/OS UNIX サンプル・ディレクトリーは、次のとおりです。 /usr/lpp/cicsts/cicsts42/samples/ejb/helloworld ここで、/usr/lpp/cicsts/cicsts42 は、z/OS UNIX 上の CICS ファイル用のインストール・ディレクトリーです。			

EJB「Hello World」サンプルのインストール:

EJB「Hello World」サンプルをインストールするには、次のリソースのセットアップが必要です。

1. z/OS UNIX。 EJB IVP を以前に実行した場合は、既にこのアクションを実行済みです。
2. CICSEJB IVP を以前に実行した場合は、既にこれらのアクションを実行済みです。
3. Web アプリケーション・サーバー。

EJB「Hello World」サンプル用の z/OS UNIX のセットアップ:

必要に応じて、z/OS UNIX で、HelloWorldEJB.jar デプロイ済み JAR ファイルを EJB サンプル・ディレクトリーから、CorbaServer のデプロイ済み JAR ファイル (「ピックアップ」) ディレクトリーにコピーしてください。

注:

1. これを実行する必要があるのは、EJB IVP の実行中に HelloWorldEJB.jar デプロイ済み JAR ファイルをまだインストールしていない場合のみです。
2. デプロイ済み JAR ファイル・ディレクトリーは、 281 ページの『EJB IVP を実行する前に』で作成したディレクトリーであり、CORBASERVER 定義の DJARDIR オプションで指定されます。
3. サンプル・ディレクトリーは /usr/lpp/cicsts/cicsts42/samples/ejb/helloworld です。ここで、/usr/lpp/cicsts/cicsts42 は、z/OS UNIX 上の CICS ファイルのインストール・ディレクトリーです。
4. z/OS UNIX の名前には大/小文字の区別があることに注意してください。
5. HelloWorldEJB.jar ファイルには、エンタープライズ Bean のソース・コードと実行可能コードの両方が入っています。

CICS のセットアップ:

このタスクについて

1. EJB 役割ベースのセキュリティーが CICS 領域でアクティブである場合、EJB「Hello World」サンプルの実行前にオフにしておく必要があります。すなわち、SEC と XEJB の両方のシステム初期設定パラメーターで現在、「YES」が指定されている場合、XEJB を「NO」に設定して、CICS を再始動する必要があります。
2. CICS 提供のサンプル・グループ DFH$EJB には、EJB「HelloWorld」サンプルの実行に適切な TCPIPSERVICE および CORBASERVER 定義が含まれています。独自の環境に適合するようにこれらのリソース定義のいくつかの属性を変更し、変更された定義を CICS にインストールする必要があります。EJB サーバーのセットアップ・タスクの一環として、既にこれを行っているはずです。まだ行っていない場合は、282 ページの『CICS で必要なアクション』にあるステップバイステップの手順に従ってください。

   注: DFH$EJB グループには、インストールに必要ないため REQUESTMODEL 定義が含まれていません。このサンプルはデフォルトのトランザクション ID である CIRP を使用します。

   a. 必要に応じて、`CEMT PERFORM CORBASERVER(EJB1) SCAN` コマンドを実行します (これを実行する必要があるのは、EJB IVP の実行中に `HelloWorldEJB.jar` デプロイ済み JAR ファイルをまだインストールしていない場合のみです)。CICS は次のことを行います。
      1) ピックアップ・ディレクトリーをスキャンします
      2) ピックアップ・ディレクトリーで検出した `HelloWorldEJB.jar` デプロイ済み JAR ファイルを、シェルフ・ディレクトリーにコピーします
      3) `HelloWorldEJB.jar` の DJAR 定義を動的に作成し、インストールします。
      4) CORBASERVER 定義が AUTOPUBLISH(YES) を指定するため、`HelloWorldEJB.jar` に含まれているエンタープライズ Bean を JNDI ネーム・スペースに公開します
3. まだ行っていない場合は、TCPIPSERVICE の状況を OPEN に設定します。

   ```
   CEMT SET TCPIPSERVICE(EJBTCP1) OPEN
   ```

   `CEMT PERFORM CORBASERVER(EJB1) SCAN` コマンドを実行した場合、CICS コンソールで、特に次のようなメッセージが表示されるはずです。

   ```
   DFHEJ5024 Scan commencing for CorbaServer EJB1, directory being scanned is
             DJARDIR_name.
   DFHEJ5030 New DJar HelloWorldEJB is being created during a scan against
             CorbaServer EJB1.
   DFHEJ0901 DJar HelloWorldEJB within CorbaServer EJB1 has been created.
   DFHEJ5025 Scan completed for CorbaServer EJB1, 1 DJars created, 0 DJars updated.
   DFHEJ5032 DJar HelloWorldEJB is having its contents automatically published to
             the namespace.
   DFHEJ5009 Published bean HelloWorld to JNDI server
             iiop://nameserver.location.company.com:900 at location samples.
   DFHEJ1540 DJar HelloWorldEJB and the Beans it contains are now accessible.
   ```

   ここで、

   - **DJARDIR_name** は、CorbaServer のデプロイ済み JAR ファイル (「ピックアップ」) ディレクトリーの名前です。

- **iiop://nameserver.location.company.com:900** は、ネーム・サーバーの URL とポート番号です。この例では、COS ネーム・サーバーが使用されます。

***Web* アプリケーション・サーバーのセットアップ:**

Web アプリケーション・サーバーで、EJB「Hello World」サンプル・アプリケーションの Web コンポーネントをインストールする必要があります。

このタスクについて

z/OS UNIX EJB サンプル・ディレクトリーから、次のものが必要です。

- `CICSHelloWorld.ear`。このサンプルの Web コンポーネント、およびサーブレットと JSP のソース・コードが入っている、J2EE エンタープライズ・アーカイブ (EAR) ファイル。
- `readme.txt`。次のものが入っているテキスト・ファイル。
  1. WebSphere Application Server にこのサンプルの Web コンポーネントをインストールするためのステップバイステップの説明。
  2. ヒントおよびデバッグ情報。

注: デフォルトのサンプル・ディレクトリーは次のとおりです。

```
/usr/lpp/cicsts/cicsts42/samples/ejb/helloworld
```

ここで、`/usr/lpp/cicsts/cicsts42` は、z/OS UNIX 上の CICS ファイル用のインストール・ディレクトリーです。

重要: このセクションの残りの部分には、このサンプルの Web コンポーネントを J2EE 準拠の Web アプリケーション・サーバー (WebSphere である場合とない場合があります) にインストールするための一般的な説明が記載されています。これは、経験のあるユーザーに適切な内容です。ご使用の Web アプリケーション・サーバーが WebSphere Application Server バージョン 4 以降であり、かつ ユーザーがその製品に慣れていない場合は、`readme.txt` ファイルにある WebSphere 固有の詳細な説明に従うことをお勧めします。

1. アプリケーションのインストールについてベンダーのガイドラインに従って、(`CICSHelloWorld.ear` に入っている) EJB「Hello World」サンプルの Web コンポーネントを、J2EE Web アプリケーション・サーバーにインストールします。WebSphere Application Sever では、例えば、これには管理コンソールを使用した次の手順が含まれています。
   a. 新規アプリケーションのインストール
   b. 更新された Web サーバー・プラグインの生成
   c. 構成の保管

   注: `CICSHelloWorld.ear` には、EJB「Hello World」サンプルのデフォルト構成が含まれています。このサンプルを実行するには、構成情報の編集も追加も不要です。
2. Web アプリケーション・サーバーの標準手順を使用してアプリケーションを開始します。

EJB「Hello World」サンプルのテスト:

アプリケーションをテストするには、以下のステップを実行する必要があります。

このタスクについて

1. 次のものがすべて実行していることを確実にします。
   - Web サーバー
   - Web アプリケーション・サーバーとサンプル・アプリケーション
   - ネーム・サーバー
   - CICS 領域
2. Web ブラウザーを開始し、「cicshello」を後に付けた Web サーバーの URL にアクセスします。例えば、次のとおりです。

   ```
   http://myServer.ibm.com/cicshello
   ```

   313 ページの図 19 に示されている開始画面が表示されます。

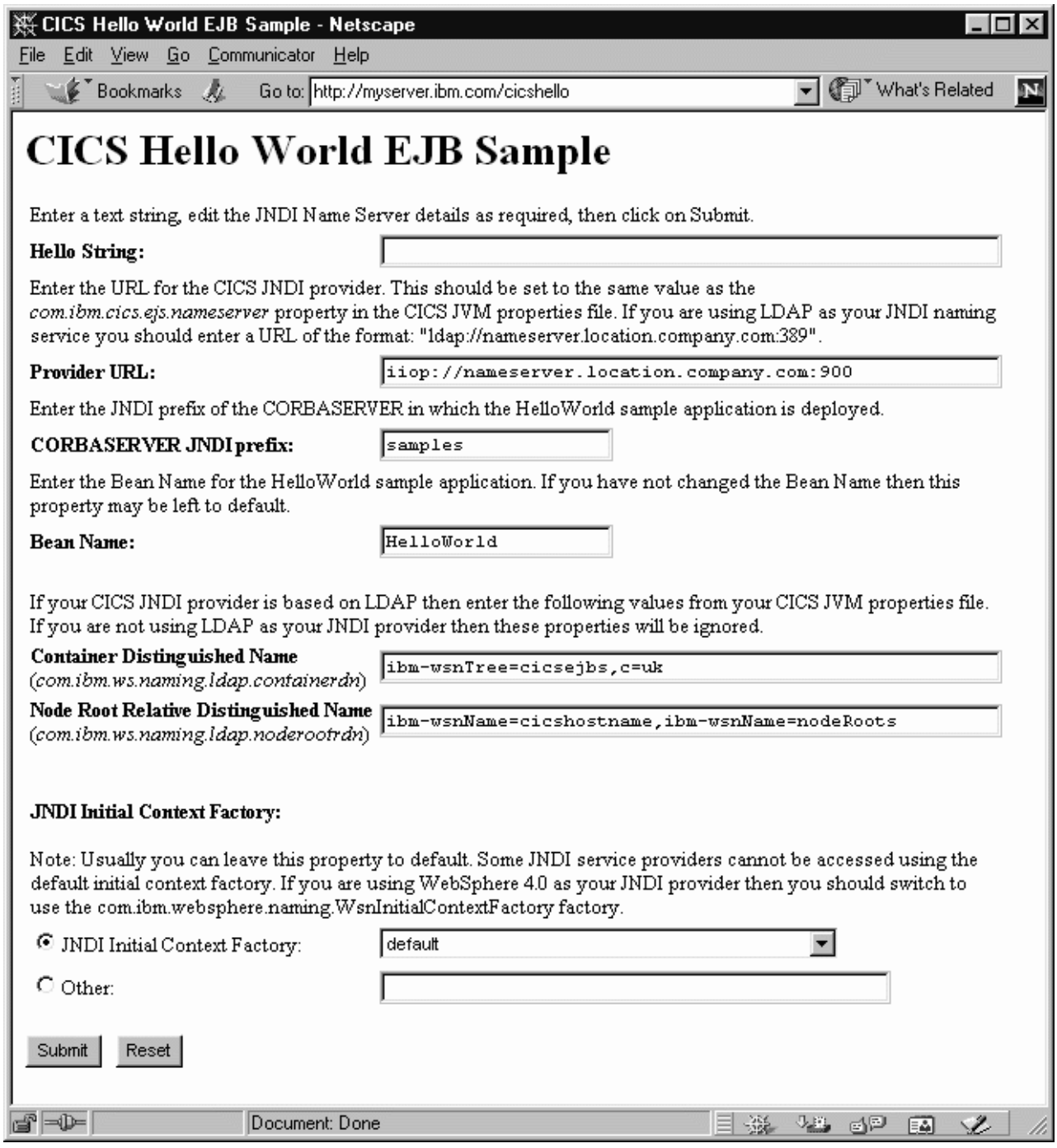


図 19. EJB「Hello World」サンプル・アプリケーションの開始画面

3. 「Hello String:」フィールドに語句を入力します。
4. 「Provider URL:」、「CORBASERVER JNDI prefix:」、「Bean Name:」、「Container Distinguished Name:」、「Node Root Relative Distinguished Name:」、および「JNDI Initial Context Factory:」の各フィールドに、ご使用のシステムに有効な値が入っていることを確認します。そうでない場合は、次のように上書きします。

   Provider URL:
   エンタープライズ Bean が公開されるネーム・サーバーの URL とポート番

号を入力します (これらは、JVM プロパティー・ファイルの **-Dcom.ibm.cics.ejs.nameserver** プロパティーで指定されます)。例えば、次のとおりです。

- URL myldapns.ibm.com およびポート番号 389 を指定する LDAP ネーム・サーバーを使用する場合は、 「ldap://myldapns.ibm.com:389」を指定します。
- URL mycosns.ibm.com およびポート番号 900 を指定する標準の COS ネーム・サーバーを使用する場合は、「iiop://mycosns.ibm.com:900」を指定します。
- URL mycosns.ibm.com およびポート番号 2809 を指定する、WebSphere Application Server バージョン 5 以降で提供される COS Naming Directory Server を使用する場合は、次のように指定します。

  ```
  -Dcom.ibm.cics.ejs.nameserver=iiop://mycosns.ibm.com:2809/domain/legacyRoot
  ```

ネーム・サーバーの場所の指定方法について詳しくは、 127 ページの『JVM システム・プロパティー』で **-Dcom.ibm.cics.ejs.nameserver** プロパティーの説明を参照してください。

CORBASERVER JNDI prefix:
CorbaServer の JNDI 接頭部を入力します。DFH$EJB で提供される CORBASERVER 定義を使用する場合は、デフォルト値「samples」を変更する必要はありません。

Bean name:
提供されている HelloWorldEJB.jar ファイルのデプロイメント記述子で定義されているとおりに、このサンプルで使用されるエンタープライズ Bean の名前を入力します。*Bean の名前を変更した場合を除いて、デフォルト値「HelloWorld」を変更する必要はありません。*

Container Distinguished Name:
LDAP ネーム・サーバーを使用する場合、LDAP 管理者によって指定されているとおりに、LDAP システム・ネーム・スペース・ルートの識別名を入力します(LDAP システム・ネーム・スペース・ルートの識別名は、JVM プロパティー・ファイルの **-Dcom.ibm.ws.naming.ldap.containerdn** プロパティーで指定されます)。 *COS ネーム・サーバーを使用する場合、このフィールドの値は無視されます。*

Node Root Relative Distinguished Name:
LDAP ネーム・サーバーを使用する場合、LDAP 管理者によって指定されているとおりに、LDAP ノード・ルートの識別名を入力します(LDAP ノード・ルートの識別名は、JVM プロパティー・ファイルの **-Dcom.ibm.ws.naming.ldap.noderootrdn** プロパティーで指定されます)。 *COS ネーム・サーバーを使用する場合、このフィールドの値は無視されます。*

JNDI Initial Context Factory:
該当する JNDI 初期コンテキスト・ファクトリーをドロップダウン・リストから選択します。ご使用の Web アプリケーション・サーバーが WebSphere である場合、使用するファクトリーは、以下のものによって異なります。
- 使用している WebSphere のバージョン

- WebSphere の場所。すなわち、Windows NT などの分散プラットフォーム上であるか、z/OS などのホスト・プラットフォーム上であるか。
- 使用しているネーム・サーバーのタイプ (COS ネーム・サーバーまたは LDAP)

表 17 は、ご使用の Web アプリケーション・サーバーが WebSphere である場合に指定する、正しい初期コンテキスト・ファクトリーを示しています。

表 17. WebSphere のバージョンと場所およびネーム・サーバーのタイプに応じた、初期コンテキスト・ファクトリーの設定

WebSphere のバージョン	**Web アプリケーション・サーバーの場所**	**ネーム・サーバーのタイプ**	**使用する初期コンテキスト・ファクトリー**
3.5	分散	COS	`com.ibm.ejs.ns.jndi.CNInitialContextFactory`
3.5	分散	LDAP	`com.ibm.jndi.LDAPCtxFactory`
3.5	z/OS	COS	`com.sun.jndi.cosnaming.CNCtxFactory`
3.5	z/OS	LDAP	`com.sun.jndi.ldap.LdapCtxFactory`
4 以降	分散	COS または LDAP	`com.ibm.websphere.naming.WsnInitialContextFactory`
4 以降	z/OS	COS	`com.sun.jndi.cosnaming.CNCtxFactory`
4 以降	z/OS	LDAP	`com.sun.jndi.ldap.LdapCtxFactory`

ご使用の Web アプリケーション・サーバーが WebSphere でない場合は、適切な値をドロップダウン・リストから選択してください。

注: このドロップダウン・リストには、複数の初期コンテキスト・ファクトリー・クラスに加えて、「デフォルト」のリスト項目が入っています。このサンプル・アプリケーションは、デフォルト・リスト項目の値を次のように割り当てます。

a. Java クラスパスで `com.ibm.websphere.naming.WsnInitialContextFactory` クラスが検出される場合、このサンプルはそれをデフォルト項目にします。このクラスは、`com.ibm.ejs.ns.jndi.CNInitialContextFactory` と `com.ibm.jndi.LDAPCtxFactory` の両方をラップする「ラッパー」クラスです。このサンプルは、「**Provider URL**」フィールドで指定されたネーム・サーバーのタイプを調べることによって、使用する正しい基本クラスを判別します。指定されたプロトコルが「iiop」である場合、このサンプルは `com.ibm.ejs.ns.jndi.CNInitialContextFactory` を使用します。「ldap」である場合、このサンプルは `com.ibm.jndi.LDAPCtxFactory` を使用します。

b. `com.ibm.websphere.naming.WsnInitialContextFactory` クラスが Java クラスパスで検出されない 場合、このサンプルは、「**Provider URL**」フィールドで指定されたネーム・サーバーのタイプを調べることによって、使用する正しいクラスを判別します。指定されたプロトコルが「iiop」である場合、このサンプルは `com.ibm.ejs.ns.jndi.CNInitialContextFactory` を使用します。「ldap」である場合、このサンプルは `com.ibm.jndi.LDAPCtxFactory` を使用します。

ドロップダウン・リスト内の値がご使用のシステムに有効でない場合、「Other」ラジオ・ボタンを選択し、下部のテキスト・フィールドに正しい値を入力してください。

5. SUBMIT ボタンを押します。これにより、サーブレットが起動し、アプリケーションが実行されます。

   アプリケーションが正しく構成され、入力値が有効である場合、HelloWorldResults JSP は、Web ブラウザーに「You said *your phrase*」というメッセージを表示します (ここで、*your phrase* は、ステップ 3 で入力した語句です)。

   アプリケーションが正しく構成されないか、1 つ以上の入力値が無効である場合、HelloWorldError JSP は、Web ブラウザーにエラー・メッセージを表示します。readme.txt ファイルには、障害が起きたアプリケーションのデバッグに役立つヒントが記載されています。

EJB Bank Account サンプル・アプリケーション

EJB Bank Account サンプルは、エンタープライズ Bean および DB2 を使用して、既存の CICS 制御情報を Web ユーザーが利用できるようにする方法を示します。

EJB「Bank Account」サンプルが行うこと:

このサンプル・アプリケーションは、データ表からカスタマー情報を抜き出し、ユーザーに戻します。

このサンプルは次のもので構成されます。

- HTML フォーム。
- J2EE 準拠の Web アプリケーション・サーバーで実行される Java サーブレットおよび JavaServer Pages。
- CICS EJB サーバーで実行されるエンタープライズ Bean。
- カスタマー情報を含む 2 つの DB2 データ表。一方のデータ表には、現在残高などの口座情報が入っています。もう一方のデータ表には、名前と住所の詳細が入っています。
- COBOL で作成された 2 つの CICS サーバー・プログラム。DFH0ACTD プログラムは、アカウント・データ表から情報を取り出します。DFH0CSTD プログラムは、名前と住所のデータ表から情報を取り出します。

このサンプルの仕組みは次のとおりです。

1. ユーザーは Web ブラウザーからこのアプリケーションを開始します。フォームが表示されます。
2. フォームはカスタマー番号をユーザーに要求します。ユーザーがカスタマー番号を入力し、SUBMIT ボタンを押すと、サーブレットが起動されます。
3. サーブレットは以下のことを行います。
   a. JNDI ネーム・スペース内のエンタープライズ Bean への参照を検索します
   b. CICS 内でエンタープライズ Bean の新しいリモート・インスタンスを作成します
   c. Bean インスタンスでメソッドを呼び出し、ユーザーが入力したカスタマー番号を入力として渡します

4. エンタープライズ Bean は、CCI Connector for CICS TS の Common Connector Interface (CCI) を使用して、CICS COBOL サーバー・プログラムにリンクし、カスタマー番号を渡します。

   CCI Connector for CICS TS については、 369 ページの『CCI Connector for CICS TS』で説明されています。
5. サーバー・プログラムは、指定された番号をこのカスタマーの DB2 レコードのキーとして使用します。DB2 データ表からカスタマーの詳細を取り出し、口座番号、残高、および住所をエンタープライズ Bean に戻します。
6. エンタープライズ Bean は、カスタマーの詳細をサーブレットに戻します。サーブレットは、JavaServer Page を使用してユーザーの Web ブラウザーにそれらの詳細を表示します。カスタマー番号が無効な場合、Web ブラウザーはエラー・ページを表示します。

318 ページの図 20 は、このサンプル・アプリケーションのコンポーネントを示しています。このサンプルの主要素は、Java サーブレット、エンタープライズ Bean、2 つの CICS サーバー・プログラム、および 2 つの DB2 データ表です。このサンプルは、データ表からカスタマーの詳細を抜き出し、ユーザーに戻します。この例では、サーブレットは Windows サーバー上の Web アプリケーション・サーバーで実行されます。LDAP ネーム・サーバーが使用されます。その他の構成も可能です。例えば、COS ネーム・サーバーを使用できます。

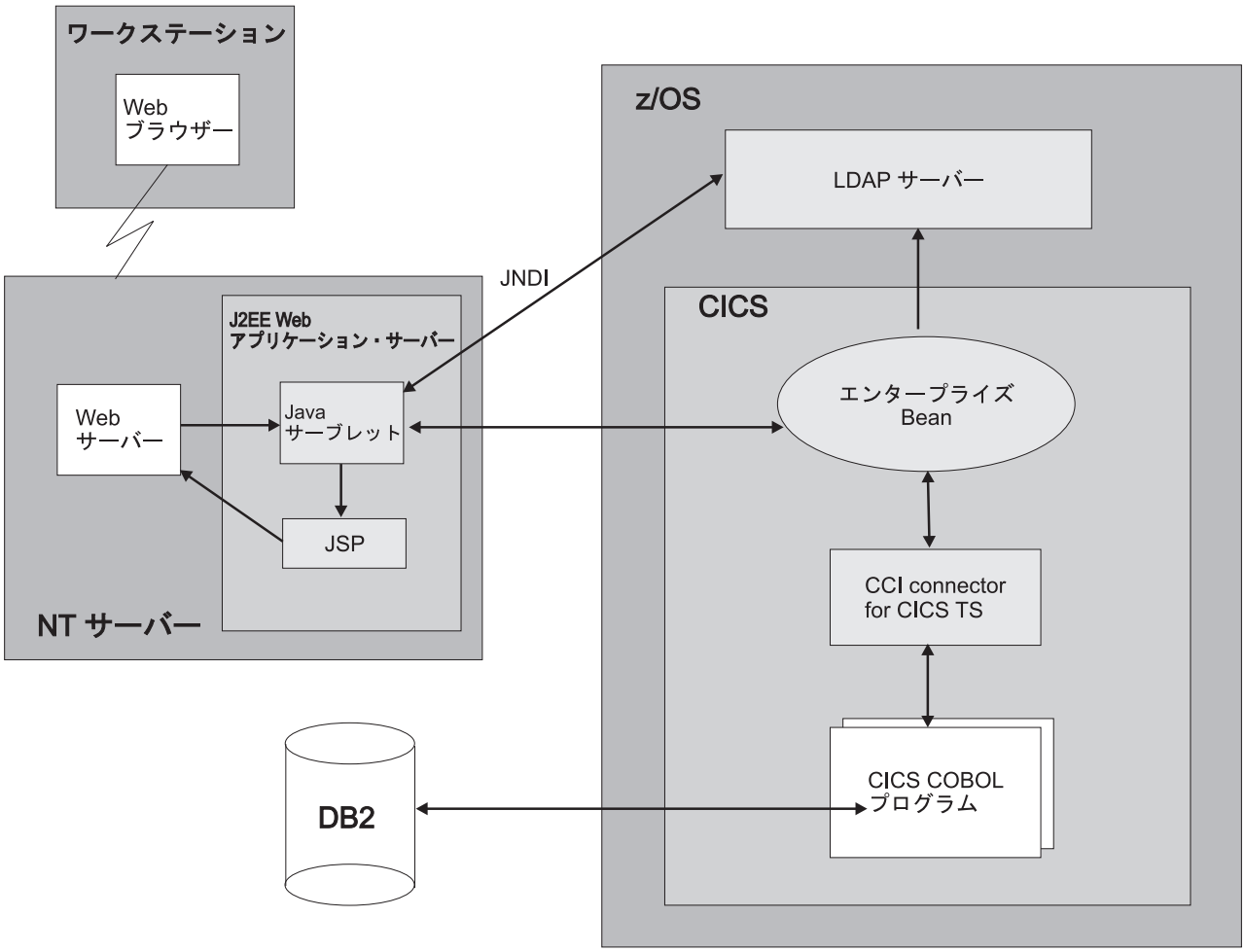


図 20. EJB Bank Account サンプル・アプリケーションの概要

EJB Bank Account サンプルの前提条件:

EJB Bank Account サンプルを実行するには、次のリソースが必要です。

- CICS EJB サーバー。このセットアップ方法については、 280 ページの『EJB サーバーのセットアップ』で説明しています。
- DB2 バージョン 7 以降。
- J2EE バージョン 1.2.1 以降をサポートする Web アプリケーション・サーバー。WebSphere Application Server を使用する場合、このサンプルには WebSphere Application Server バージョン 4 以降が必要です。
- JNDI Version 1.2 以降をサポートするネーム・サーバー。このセットアップ方法については、 281 ページの『z/OS または Windows NT で必要なアクション』で説明しています。

EJB Bank Account サンプルに提供されているコンポーネント:

EJB Bank Account サンプルには、以下のファイルが提供されています。

表 18. EJB Bank Account サンプルに提供されているコンポーネント

Filename (ファイル名)	**タイプ**	**デフォルトの場所**	**コメント**
DFH$EDB2	テキスト・デック	SDFHSAMP	このサンプルで使用される DB2 データ表を定義し、データ表にデータを取り込む、DB2 データ定義言語 (DDL) ステートメント。
DFH$ESQL	テキスト・デック	SDFHSAMP	DB2 データ表を COBOL サーバー・プログラムにバインドするための DB2 データ操作言語 (DML) ステートメント。
DFH$EJB2	リソース定義グループ	CSD	このサンプル・アプリケーションに必要な CICS リソース定義が入っています。
DFH0ACTD	COBOL ソース・コード	SDFHSAMP	DFH0ACTD サーバー・プログラムのソース・コード。
DFH0CSTD	COBOL ソース・コード	SDFHSAMP	DFH0CSTD サーバー・プログラムのソース・コード。
DFHEBURM	ユーザーが置き換え可能なサンプル・プログラム	SDFHSAMP	このサンプルの実行に使用されるユーザー ID を変更します。
CicsSample.ear	EAR ファイル	z/OS UNIX サンプル・ディレクトリー。注を参照。	サンプル・アプリケーションの Web コンポーネント。Java サーブレット・クラスとソース・ファイル、HTML と JSP。
readme.txt	テキスト・ファイル	z/OS UNIX サンプル・ディレクトリー。注を参照。	何を組み込めるか: 1. WebSphere Application Server にこの EJB サンプルの Web コンポーネントをインストールするためのステップバイステップの説明。 2. ヒントおよびデバッグ情報。
SampleCLI.jar	JAR ファイル	z/OS UNIX サンプル・ディレクトリー。注を参照。	サーブレットで必要なクライアント EJB スタブ。
SampleEJB.jar	デプロイ済み JAR ファイル	z/OS UNIX サンプル・ディレクトリー。注を参照。	Java クラス、ソース・ファイル、デプロイメント記述子に加えて、CICS エンタープライズ Bean をサポートするクラス。ソース・コードの変更が必要な場合を除いて、解凍する必要はありません。

表 18. EJB Bank Account サンプルに提供されているコンポーネント (続き)

Filename (ファイル名)	タイプ	デフォルトの場所	コメント
注: デフォルトの z/OS UNIX サンプル・ディレクトリーは、次のとおりです。 /usr/lpp/cicsts/cicsts42/samples/ejb/bankaccount ここで、cicsts42 は、z/OS UNIX 上の CICS ファイル用のインストール・ディレクトリーです。			

EJB Bank Account サンプルのセキュリティー:

機密保護機能のある環境で Bank Account サンプルを実行することをお勧めします。ただし、インストール・プロセスを簡単にするために、最初はそうしないことを選択できます。

機密保護機能のある環境を即時に活動化したくない場合は、XEJB システム初期設定パラメーターを「NO」に設定し、このセクションの残りの部分をスキップしてください。機密保護機能のある環境を後日活動化するには、このセクションの残りの手順を実行してください。

このサンプルのセキュリティーを実装するには、複数の方法があります。例えば、次のいずれかの選択肢を使用できます。

- すべてのユーザーが、デフォルトのユーザー ID でこのサンプルを実行できるようにします。
- すべてのユーザーが、IIOP 用のセキュリティー出口プログラムで指定されたユーザー ID でこのサンプルを実行できるようにします。
- SSL サーバー・サイド証明書を使用して、Web 層と CICS 間で送信されるデータを暗号化して、すべてのユーザーがデフォルトのユーザー ID でセキュア・トランスポートを介してこのサンプルを実行できるようにします。
- SSL サーバー・サイド証明書を使用して、Web 層と CICS 間で送信されるデータを暗号化して、すべてのユーザーが、IIOP 用のセキュリティー出口プログラムで指定されるユーザー ID でセキュア・トランスポートを介してこのサンプルを実行できるようにします。
- SSL クライアント認証を使用して、Web 層のアプリケーション・サーバーを CICS に対して自動的に認証して、Web 層のアプリケーション・サーバーに割り当てられたユーザー ID で、すべてのユーザーがセキュア・トランスポートを介してこのサンプルを実行できるようにします。
- アサーション ID 認証を使用して、WebSphere Application Server for z/OS で実行中の Web 層のクライアント・アプリケーションが、セキュア・トランスポートを介して既存のユーザー ID を CICS に伝搬できるようにします。

注:

1. デフォルトで、Bank Account アプリケーションではユーザーが Web 層で認証される必要はありません。アプリケーション・サーバーの指示に従って、Web コンテナー内で認証をアクティブにすることを選択できます。 Web 層で認証を行う場合、セキュリティー原則は CICS に伝搬されないので、CICS セキュリティーの観点では無効です。ただし、Web 層での早期認証を使用すると、「保護

ドメイン」を作成できます。保護ドメインでは、非認証ユーザーによる CICS エンタープライズ Bean でのビジネス・メソッドの呼び出しを許可しないようにすることを CICS は Web 層に任せます。

2. SSL 暗号化または認証を使用するには、SSL を完全にサポートする J2EE 準拠の Web アプリケーション・サーバーが必要です。詳しくは、ベンダーの資料を参照してください。
3. SSL 認証について詳しくは、「*CICS RACF Security Guide*」の SSL authentication を参照してください。

どちらの認証方式を選択しても、(特に) 以下の手順を行う必要があります。

1. CICS におけるエンタープライズ Bean のデプロイメント記述子で許可情報を提供します。この許可情報は次のもので構成されます。

   「セキュリティー役割」エレメント
   所定のアクションの実行または所定のリソースの使用を行うことができるユーザーのクラスを識別します。

   「メソッド・アクセス権」エレメント
   指定されたセキュリティー役割のメンバーが使用する権限を持つ、エンタープライズ Bean の特定のメソッドを識別します。
2. CICS 外部セキュリティー・マネージャー (ESM) を更新して、指定されたセキュリティー役割を複数の実ユーザー ID にマップします。セキュリティーを実装するための以下のステップバイステップの手順では、選択した ESM が RACF であることを前提としています。別の ESM を使用する場合、詳細については ESM ベンダーにお問い合わせください。

Bank Account サンプルの役割ベース・セキュリティーの実装:

Assembly Toolkit (ATK。Application Server Toolkit (ASTK) のコンポーネント) を使用して、Bank Account サンプルの役割ベースのセキュリティーを実装できます。

このタスクについて

このツールは、WebSphere Application Server バージョン 5.1 以降の一部として出荷時に付属しています。ATK のグラフィカル・ユーザー・インターフェースを使用して、(特に) エンタープライズ Bean のデプロイメント記述子の内容を編集できます。

始める前に、ワークステーションに ATK がインストールされていることを確認してください。インストール後、このツールは、Windows の「スタート」メニューに追加されるアイコンから起動できます。

ATK は、役割ベースのセキュリティーを実装する最初の段階に使用されます。これには、エンタープライズ Bean のデプロイメント記述子の編集が含まれます。その段階を完了したら、役割ベース・セキュリティーの 2 番目の実装段階の指示に従います。これには、他のソフトウェアの構成が含まれます。

段階 1. ATK を使用したデプロイメント記述子の編集:

この時点で、ATK を理解するために、JAR ファイルの内容を参照できます。

1. SampleEJB.jar ファイルを z/OS UNIX サンプル・ディレクトリーからワークステーションにコピーします。これを行うには、バイナリー・モードで FTP を使用するか、他の適当な方法を使用できます。z/OS UNIX サンプル・ディレクトリーは /usr/lpp/cicsts/cicsts42/samples/ejb/bankaccount です。ATK に対して、dfjcci.jar ファイルに同じ処理を実行することも必要です。このファイルは /usr/lpp/cicsts/cicsts42/lib ディレクトリーにあります。ユーザーがその JAR ファイルを編集する必要はありませんが、ATK が、編集後に EJB bank account サンプルの JAR ファイルを正しく再作成するのに必要です。
2. JAR ファイルを EJB プロジェクトとして ATK にインポートします。
   a. ATK を開始し、「**Window**」>「**Open Perspective**」>「**J2EE**」を選択して J2EE パースペクティブに進みます。
   b. 「**File**」メニューから「**Import**」オプションを選択します。インポートのソースとして「**EJB JAR file**」を選択します。「**Browse**」を選択して、SampleEJB.jar ファイルを見つけます。プロジェクトに適切な名前を入力します。「**Next**」を選択して、すべてのエンタープライズ Bean のインポートを選択します (これがデフォルトです)。「**Finish**」を選択して EJB プロジェクトを作成します。
   c. プロジェクトが作成されると、「Tasks」リストにいくつかのエラーが表示されるはずです。これらのエラーを訂正するには、dfjcci.jar ファイルを EJB プロジェクトのビルド・パスに追加する必要があります。左側のナビゲーション・ペインで (J2EE 階層ビューを使用して)、「EJB Modules」項目を展開して EJB プロジェクトを表示します。プロジェクト名を右クリックして、「**Properties**」を選択します。「**Java Build Path**」を選択します。「**Libraries**」タブに進み、「**Add External JARs**」ボタンを選択します。dfjcci.jar ファイルまでナビゲートし、「**Open**」を選択します。「**OK**」を選択します。ATK が EJB プロジェクトを再ビルドし、エラーが消えます。

   EJB デプロイメント記述子について詳しくは、258 ページの『エンタープライズ Bean — デプロイメント記述子』を参照してください。
3. セキュリティー役割をデプロイメント記述子に追加します。ATK の左側のナビゲーション・ペインで (J2EE 階層ビューを使用して)、「EJB Modules」項目を展開して EJB プロジェクトを表示します。プロジェクト名をダブルクリックして、プロジェクトをオープンします。ペインの下部にある「**Assembly Descriptor**」タブを選択します。「**Security Roles**」の下で、「Add」ボタンを選択して新規セキュリティー役割を追加します。

   他のアプリケーションで使用するためにセキュリティー役割を組織で既にセットアップ済みの場合は、既存の役割を再利用できます。そうする場合は、指定されたフィールドに使用する役割の名前を指定します。再利用したい既存のセキュリティー役割がない場合は、「All_users」などの新しい役割名を入力します。また、将来の記憶の保持になるように、役割の説明をオプションで指定することもできます。「**Finish**」を選択してメインウィンドウに戻ります。

   注: ESM に対して既に定義されている既存のセキュリティー役割を再利用する場合は、JAR ファイルのデプロイメント記述子から Display Name エレメントを除去する必要があります。このエレメントは CICS で使用され、実行時にセキュリティー検査を行うときに、すべてのセキュリティー役割名の前に付けられるアプリケーション名を指定します。したがって、企業全体ではなく、アプリケーシ

ョン・レベルを対象にしたセキュリティー役割をサポートします。ATK では、ペインの下部にある「**Overview**」タブを選択して、このエレメントを除去できます。「Display Name」フィールド内のテキストを選択し、削除してください。

4. メソッド・アクセス権を定義し、セキュリティー役割に関連付けます。ATK で、「**Assembly Descriptor**」タブを再度選択します。「**Method Permissions**」の下で、「**Add**」ボタンを選択します。ウィザードに、定義したセキュリティー役割のリストが表示されます。Bank Account サンプルの場合、同じセキュリティー役割ですべてのメソッドを実行するのが適切です。メソッド・アクセス権に関連付けたいセキュリティー役割を選択し、「**Next**」を選択します。CICSSample Bean を選択し、「**Next**」を選択します。CICSSample のボックスにチェック・マークを付けて、Bean のすべてのメソッド要素を選択します。「**Finish**」を選択します。前の画面に戻ります。
5. 「**File**」メニューから「**Save**」オプションを選択して、更新されたデプロイメント記述子を保管します。
6. ワークステーションでプロジェクトを ATK から JAR ファイルにエクスポートします。これを行うには、「**File**」メニューから「**Export**」オプションを選択します。エクスポートの宛先として「**EJB JAR file**」を選択し、「**Next**」を選択します。EJB プロジェクトをドロップダウン・リストから選択します。「**Browse**」を選択し、宛先として使用する SampleEJB.jar ファイルを見つけます (これは、このファイルのオリジナル・バージョンを上書きします。ワークステーションで、オリジナル・バージョンのファイルのバックアップを別の名前で保持できます)。「**Export source files**」のチェック・ボックスを選択して、ソース・ファイルを JAR ファイルと一緒に保持します。「**Finish**」を選択します。ATK を終了します。
7. 更新された SampleEJB.jar ファイルを z/OS UNIX にコピーして戻します。バイナリー・モードの FTP か、適切なファイル転送プロセスを使用できます。SampleEJB.jar ファイルを CorbaServer のピックアップ・ディレクトリーに保管します。

段階 2. その他のセキュリティー設定の構成:

エンタープライズ Bean のデプロイメント記述子で定義されたセキュリティー役割に関連付けるために選択する CICS ユーザー ID (複数の場合あり) は、このセクションの始めで選択したセキュリティー実装環境に応じて選択されなければなりません。

1. SEC と XEJB の両方の CICS システム初期設定パラメーターが「YES」に指定されていることを確認します (どちらかで「NO」が指定される場合、EJB 役割ベースのセキュリティーはオフになります)。
2. ご使用のシステムで既にセットアップされた既存のセキュリティー役割を再利用した場合、このステップをスキップできます。このステップでは、RACF を更新して、EJB セキュリティー役割を 1 組の CICS ユーザー ID に関連付けます。

   注: ESM が RACF でない場合は、このステップの実行方法について、ESM ベンダーにアドバイスを求める必要があります。

   例えば、次のとおりです。

- すべての匿名ユーザーがサンプルを実行できるようにしたい場合 (SSL を使用するかどうかに関係なく)、CICSUSER デフォルト・ユーザー ID をセキュリティー役割に関連付ける必要があります。
- IIOP 用のセキュリティー出口プログラムで選択されるユーザー ID (複数の場合あり) でこのサンプルを実行したい場合 (SSL を使用するかどうかに関係なく)、そのユーザー ID (複数の場合あり) をセキュリティー役割に関連付ける必要があります。
- 完全な SSL クライアント認証を使用したい場合は、Web 層アプリケーション・サーバーの証明書のユーザー ID をセキュリティー役割に関連付ける必要があります。

EJB セキュリティー役割と CICS ユーザー ID 間の必要なマッピングをセットアップするには、以下の手順を実行します。

a. 更新された `SampleEJB.jar` ファイルに対して、RACF EJBROLE 生成ユーティリティーを実行します (RACF EJBROLE 生成ユーティリティーは Java プログラムであり、デプロイメント記述子からセキュリティー役割情報を抜き出し、RACF に対してセキュリティー役割を定義する REXX プログラムを生成します。生成ユーティリティーの使用法については、407 ページの『RACF EJBROLE 生成ユーティリティーの使用』を参照してください)。
b. RACF EJBROLE 生成ユーティリティーによって生成される REXX プログラムを実行するように、RACF 管理者に依頼します。

3. IIOP 用のセキュリティー出口プログラムを使用して、サンプルの実行に使用されるユーザー ID を (デフォルトの CICS ユーザー ID から別の適当な ID に) 変更したくない場合、このステップをスキップできます。

   CICS が提供するサンプル・セキュリティー出口プログラム DFHEBURM は、Bank Account サンプルの実行に使用されるユーザー ID を、デフォルトの CICS ユーザー ID から「SAMPLE」に変更します。このバージョンのユーザー置換可能プログラムを使用するか、ニーズに合わせてそれを変更することができます。IIOP 用のカスタマイズされたセキュリティー出口プログラムが既にある場合は、ご使用のバージョンを更新して同じような機能を実行できます。

   サンプルの実行に使用される TCPIPSERVICE 定義の URM オプションで、セキュリティー出口プログラムの名前を指定する必要があります。

   IIOP 用のセキュリティー出口プログラムについて詳しくは、454 ページの『IIOP ユーザー置換可能セキュリティー・プログラムの使用』を参照してください。

   IIOP 用のセキュリティー出口プログラムの作成については、「*CICS Customization Guide*」を参照してください。また、コメントやヒントが含まれている、提供されたサンプル・プログラムのソースも調べてください。

   ユーザー置換可能プログラムのコンパイルとインストールについては、「*CICS Customization Guide*」の Assembling and link-editing user-replaceable programs を参照してください。

   TCPIPSERVICE 定義のコーディングについては、「*CICS Resource Definition Guide*」を参照してください。

4. SSL 暗号化または認証を使用する場合は、以下の手順を実行する必要があります。
   - SSL を使用するように J2EE 準拠 Web アプリケーション・サーバーを構成します。詳しくは、ご使用の Web サーバーの資料を参照してください。
   - サーバー証明書を使用できるように用意します。
   - サンプルの実行に使用される、CORBASERVER および TCPIPSERVICE リソースの定義を変更します。具体的には、以下のとおりです。
     – SSL クライアント・サイド認証を使用する場合、CORBASERVER 定義の CLIENTCERT オプションは、SSL クライアント認証でインバウンド IIOP 要求に使用されるポートを定義する TCPIPSERVICE の名前を指定する必要があります。また、Web アプリケーション・サーバーの SSL 証明書は次のとおりでなければなりません。
       - RACF において、CICS で信頼される証明書のリストに含まれなければなりません。
       - RACF ユーザー ID にマップされなければなりません。
     – SSL サーバー・サイド認証を使用する場合、CORBASERVER 定義の SSLUNAUTH オプションは、SSL (ただし、クライアント認証なし) でインバウンド IIOP 要求に使用されるポートを定義する TCPIPSERVICE の名前を指定する必要があります。

     CORBASERVER リソース定義および TCPIPSERVICE リソース定義のコーディングについては、「*CICS Resource Definition Guide*」を参照してください。
   - 暗号化、認証、および ID 伝搬にアサーション ID 認証を使用する場合は、次の手順を実行する必要があります。
     – ユーザーを認証するように WebSphere Application Server for z/OS を構成します。
     – WebSphere Application Server for z/OS バージョン 6.1 以降を使用して適切な認証プロトコルを使用可能にする場合、CICS 領域で使用されるすべての JVM プロパティー・ファイルでシステム・プロパティー **-Dcom.ibm.cics.iiop.CSIv2Enabled=true** を指定します (この機能をサポートするには、WebSphere Application Server for z/OS のリリース 6.1.0.13 以降が必要です)。
     – WebSphere で SSL クライアント認証を使用可能にします。
     – サーバー SSL 証明書を CICS で使用できるように用意します。
     – WebSphere Application Server に関連したサーバー証明書を、CICS で信頼される RACF の証明書リストに組み込みます。さらに、RACF 証明書に関連したユーザー ID には、他のユーザーの ID を表明する権限が付与されなければなりません。
     – サンプルの実行に使用される、CORBASERVER および TCPIPSERVICE リソースの定義を変更します。CORBASERVER 定義の ASSERTED オプションは、アサーション ID 認証でインバウンド IIOP 要求に使用されるポートを定義する TCPIPSERVICE の名前を指定する必要があります。

EJB Bank Account サンプルのインストール:

EJB Bank Account サンプルのインストールには、以下に対するアクションが必要です。

1. z/OS (DB2 および CICS)
2. Web アプリケーション・サーバー

z/OS のセットアップ:

z/OS 上で EJB サンプルをインストールするには、次の手順で行います。

このタスクについて

1. 組織で通常行われている手順を使用して、CICS COBOL DB2 サーバー・プログラムをコンパイルし、リンク・エディットします。SDFHSAMP ライブラリーの DFH0ACTD および DFH0CSTD メンバーには、サーバー・プログラムのソース・コードが入っています。

   CICS DD DFHRPL 連結に含まれているアプリケーション・ロード・ライブラリーに、ロード・モジュールを保管します
2. このサンプルで使用される DB2 データ表を定義し、それらの表にデータを取り込みます。DFH$EDB2 テキスト・デックには、必要な DB2 DDL ステートメントおよび提供されたデータが入っています。

   DFH$EDB2 を使用する前に、システムを適合させるために次の行を変更する必要があります。

   ```
   CREATE STOGROUP EBSAMPSG VOLUMES(SYSDA,SYSDB) VCAT DSNxxxxx;
   ```

   DSN*xxxxx* を、ユーザー定義 VSAM データ・セットに対する高位の統合カタログ機能 (ICF) カタログ ID の名前に変更します。

   必要な権限: データベース、ストレージ・グループ、表スペース、表、およびインデックスを作成するための DB2 権限。
3. DB2 表を COBOL サーバー・プログラムにバインドします。DFH$ESQL テキスト・デックには、必要な DB2 DML ステートメントが入っています。

   必要な権限: このデータベースに対して BIND を実行するための DB2 権限。

   注:

   a. このステップは、サーバー・プログラム内の SQL ステートメントを DB2 に静的にバインドします。その結果、それらのステートメントを実行時に動的にバインドする必要がないため、実行時のパフォーマンスが改善されます。

   b. 後でいずれかのサーバー・プログラムを再コンパイルし、そのプログラムが DB2 にアクセスする必要がある場合は、再コンパイルのたびに次の手順を使用してください。

      1) COBOL サーバー・プログラムに DB2 表を再バインドします。
      2) CICS 領域で次の CICS コマンドを実行して、CICS でサーバー・プログラムのコピーをリフレッシュします。

         ```
         CEMT SET PROG(program_name) NEW
         ```

例えば、DFH0CSTD プログラムを変更し、それを再コンパイルする場合は、CEMT SET PROG(DFH0CSTD) NEW を使用します (DFH0CSTD は、DFH$EJB2 リソース定義グループで CICS 領域に対して定義されます。ステップ 5 を参照してください)。

4. 組織で通常行われている手順 (例えば、SPUFI) を使用して、DB2 プランにアクセスする権限を CICS ユーザー ID に付与します。DB2 プランにアクセスする権限の付与については、「*CICS DB2 Guide*」のプランに対するユーザーのアクセスの制御を参照してください。
5. このサンプルで使用されるプログラムと DB2 接続を CICS に対して定義します。CICS 提供のサンプル・グループ DFH$EJB2 には、EJB「Bank Account」サンプルのリソース定義が入っています。独自の環境に適合するように、これらのリソース定義のいくつかの属性を変更する必要があります。これを行うには、CEDA トランザクションまたは DFHCSDUP ユーティリティーを使用します。
   a. このサンプル・グループを独自に選択したグループにコピーします。例えば、次のとおりです。

      ```
      CEDA COPY GROUP(DFH$EJB2) TO(mygroup)
      ```

   b. mygroup グループを表示し、示されているとおりに以下の定義の属性を変更します。
      - DB2CONN 定義で、DB2ID の値を、サンプルで使用される DB2 表を作成した DB2 サブシステムの ID に変更します。
      - PROGRAM 定義を変更する必要はありません。
   c. 不要な定義を mygroup グループから破棄します。

      DB2CONN および PROGRAM 定義に加えて、DFH$EJB2 には CORBASERVER および TCPIPSERVICE 定義も含まれます。ただし、これらは参照専用です。このサンプル・プログラムをインストールしようとする*前*に、280 ページの『EJB サーバーのセットアップ』で説明されているとおりに EJB サーバーをセットアップすることを強くお勧めします。このようにすれば、DFH$EJB2 内の CORBASERVER 定義および TCPIPSERVICE 定義は必要ありません。これは、リソース・グループ DFH$EJB で提供された定義に基づいて、独自の定義を既に作成済みであるからです。それらを mygroup グループから破棄します。

      DFH$EJB2 の CORBASERVER および TCPIPSERVICE 定義を使用することを決定する 場合、282 ページの『CICS で必要なアクション』で説明されているとおりにそれらの定義を変更する必要があります。

      CICS 領域でプログラムの自動インストールが使用される場合、PROGRAM 定義は必要ありません。それらを mygroup グループから破棄します。

      注: REQUESTMODEL 定義は、インストールの必要がないため、提供されていません。このサンプルはデフォルトのトランザクション ID である CIRP を使用します。
   d. 変更されたリソース定義が入っているリソース・グループを、CICS CSD および CICS 開始グループ・リストに追加します。CICS システム定義ユーテ

ィリティー・プログラム DFHCSDUP を使用できます。「*CICS Resource Definition Guide*」のシステム定義ファイル・ユーティリティー・プログラム (DFHCSDUP) を参照してください。

必要な権限: リソース定義を CICS 領域にインストールするための RACF 権限。

6. セキュリティーのセットアップ中に行わなかった場合は、提供された SampleEJB.jar デプロイ済み JAR ファイルを、CorbaServer のピックアップ・ディレクトリーに置きます。
7. ネーム・サーバーが開始していることを確認します。CICS が開始していない場合は、直ちに開始します。
8. CICS 領域コンソールで次のコマンドを発行します。

   ```
   CEMT PERFORM CORBASERVER(corbaserver_name) SCAN
   ```

   CICS はピックアップ・ディレクトリーをスキャンし、SampleEJB.jar デプロイ済み JAR ファイルをシェルフ・ディレクトリーにコピーし、そのファイルの DJAR 定義を作成し、インストールします。

   注: ステップ 7 で CICS を開始しなければならなかった場合は、CICS が開始時にピックアップ・ディレクトリーをスキャンしているため、このステップは不要です。

   必要な権限: DJAR を作成し、CORBASERVER へのアクセスを更新するための RACF 権限。
9. エンタープライズ Bean を JNDI ネーム・スペースに公開します。CORBASERVER 定義に AUTOPUBLISH(YES) が指定されている場合、SampleEJB.jar デプロイ済み JAR ファイルがインストールされたときにこれは自動的に行われています。CORBASERVER 定義が AUTOPUBLISH(NO) を指定する場合、CICS 領域コンソールで次のコマンドを発行します。

   ```
   CEMT PERFORM DJAR(SampleEJB) PUBLISH
   ```

   必要な権限: DJAR を更新するための RACF 権限。
10. CICSConnectionFactoryPublish サンプル・プログラムを使用して、CCI Connector for CICS TS で使用するための ConnectionFactory オブジェクトを作成し、それをネーム・サーバーに公開します。CICSConnectionFactoryPublish プログラムの使用法については、379 ページの『接続ファクトリーの管理および獲得のためのサンプル・ユーティリティー・プログラムの使用』を参照してください。
11. CICS システム・コンソールで次のコマンドを発行して、DB2 接続状況が CONNECTED であることを確実にします。

    ```
    CEMT SET DB2CONN CONNECTED
    ```

Web アプリケーション・サーバーのセットアップ:

Web アプリケーション・サーバーで、EJB Bank Account サンプル・アプリケーションの Web コンポーネントをインストールする必要があります。

このタスクについて

z/OS UNIX EJB サンプル・ディレクトリーから、次のものが必要です。

- CicsSample.ear。このサンプルの Web コンポーネントが入っている J2EE エンタープライズ・アーカイブ (EAR) ファイル。
- readme.txt。次のものが入っているテキスト・ファイル。
  1. WebSphere Application Server にこのサンプルの Web コンポーネントをインストールするためのステップバイステップの説明。
  2. ヒントおよびデバッグ情報。

注: デフォルトのサンプル・ディレクトリーは次のとおりです。

```
/usr/lpp/cicsts/cicsts42/samples/ejb/bankaccount
```

ここで、/usr/lpp/cicsts/cicsts42 は、z/OS UNIX 上の CICS ファイル用のインストール・ディレクトリーです。

重要: このセクションの残りの部分には、このサンプルの Web コンポーネントを J2EE 準拠の Web アプリケーション・サーバー (WebSphere である場合とない場合があります) にインストールするための一般的な説明が記載されています。これは、経験のあるユーザーに適切な内容です。ご使用の Web アプリケーション・サーバーが WebSphere Application Server バージョン 4 以降であり、かつ ユーザーがその製品に慣れていない場合は、readme.txt ファイルにある WebSphere 固有の詳細な説明に従うことをお勧めします。

手順

1. アプリケーションのインストールについてベンダーのガイドラインに従って、(CicsSample.ear に入っている) EJB Bank Account サンプルの Web コンポーネントを、J2EE Web アプリケーション・サーバーにインストールします。WebSphere Application Sever では、例えば、これには管理コンソールを使用した次の手順が含まれています。
   a. 新規アプリケーションのインストール
   b. 更新された Web サーバー・プラグインの生成
   c. 構成の保管

   注: CicsSample.ear には、EJB Bank Account サンプルのデフォルト構成が含まれています。このサンプルを実行するには、構成情報の編集も追加も不要です。
2. Web アプリケーション・サーバーの標準手順を使用してアプリケーションを開始します。

タスクの結果

EJB Bank Account サンプルのテスト:

アプリケーションをテストするには、以下のステップを実行する必要があります。

このタスクについて

1. 次のものがすべて実行していることを確実にします。
   - Web サーバー
   - Web アプリケーション・サーバーとサンプル・アプリケーション

- ネーム・サーバー
- CICS 領域
- DB2 サブシステム

2. Web ブラウザーを開始し、「cicssample」を後に付けた Web サーバーの URL にアクセスします。例えば、次のとおりです。

   ```
   http://myServer.ibm.com/cicssample
   ```

   331 ページの図 21 に示されている開始画面が表示されます。

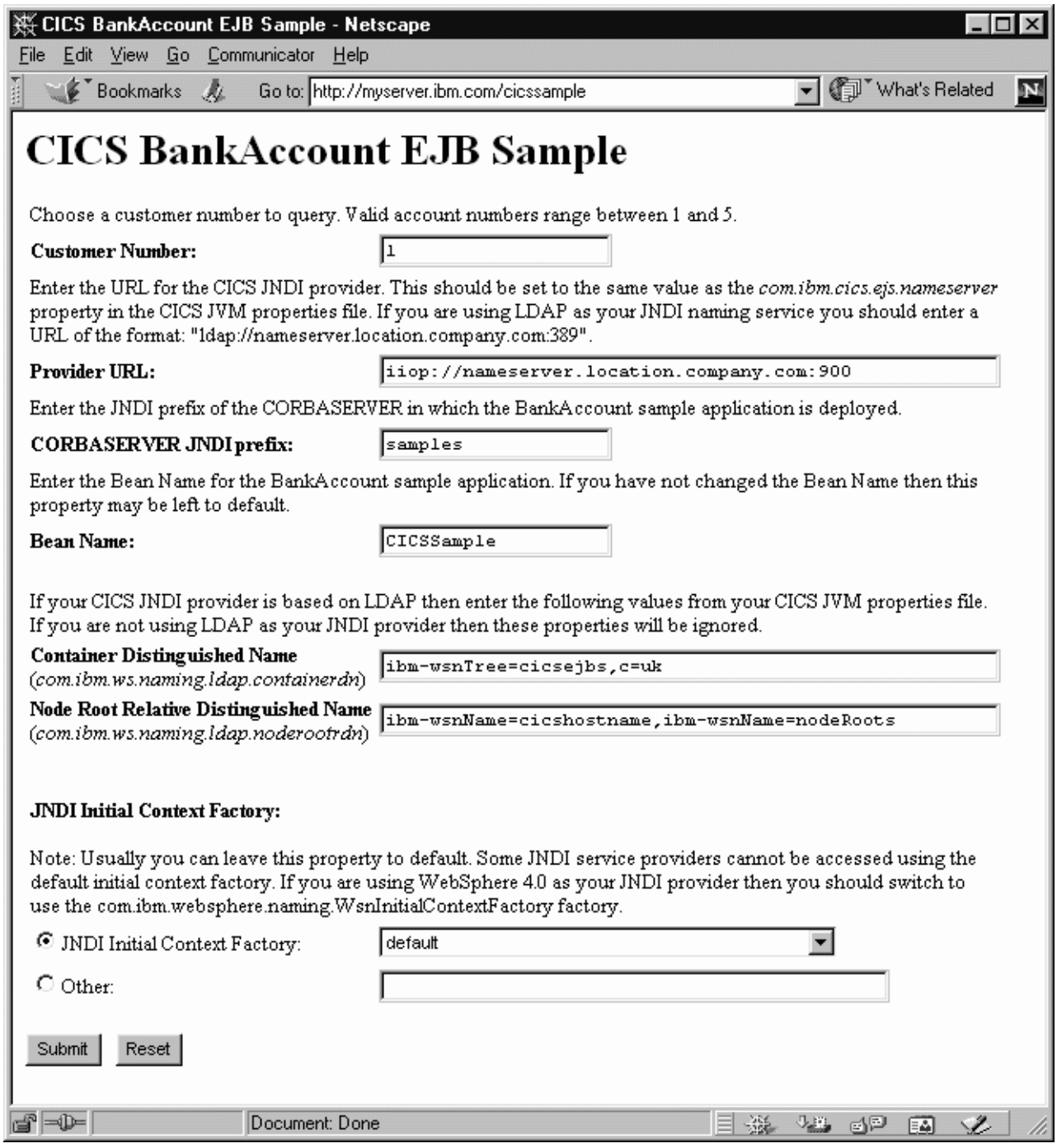


図 21. EJB Bank Account サンプル・アプリケーションの開始画面

3. カスタマー番号を入力します (提供された DB2 データを使用すると、有効なカスタマー番号は 1 から 5 です)。
4. 「Provider URL:」、「CORBASERVER JNDI prefix:」、「Bean Name:」、「Container Distinguished Name:」、「Node Root Relative Distinguished Name:」、および「JNDI Initial Context Factory:」の各フィールドに、ご使用のシステムに有効な値が入っていることを確認します。そうでない場合は、次のように上書きします。

Provider URL:
エンタープライズ Bean が公開されるネーム・サーバーの URL とポート番号を入力します (これらは、JVM プロパティー・ファイルの **-Dcom.ibm.cics.ejs.nameserver** プロパティーで指定されます)。例えば、次のとおりです。

- URL mycosns.ibm.com およびポート番号 900 を指定する COS ネーム・サーバーを使用する場合は、「iiop://mycosns.ibm.com:900」を指定します。
- URL myldapns.ibm.com およびポート番号 389 を指定する LDAP ネーム・サーバーを使用する場合は、 「ldap://myldapns.ibm.com:389」を指定します。
- URL mycosns.ibm.com およびポート番号 2809 を指定する、WebSphere Application Server バージョン 5 以降で提供される COS Naming Directory Server を使用する場合は、次のように指定します。

  ```
  -Dcom.ibm.cics.ejs.nameserver=iiop://mycosns.ibm.com:2809/domain/legacyRoot
  ```

ネーム・サーバーの場所の指定方法について詳しくは、127 ページの『JVM システム・プロパティー』で **-Dcom.ibm.cics.ejs.nameserver** プロパティーの説明を参照してください。

CORBASERVER JNDI prefix:
CorbaServer の JNDI 接頭部を入力します。DFH$EJB で提供される CORBASERVER 定義を使用する場合は、デフォルト値「samples」を変更する必要はありません。

Bean name:
提供されている SampleEJB.jar ファイルのデプロイメント記述子で定義されているとおりに、このサンプルで使用されるエンタープライズ Bean の名前を入力します。*Bean の名前を変更した場合を除いて、デフォルト値「CICSSample」を変更する必要はありません。*

Container Distinguished Name:
LDAP ネーム・サーバーを使用する場合、LDAP 管理者によって指定されているとおりに、LDAP システム・ネーム・スペース・ルートの識別名を入力します(LDAP システム・ネーム・スペース・ルートの識別名は、**-Dcom.ibm.ws.naming.ldap.containerdn** システム・プロパティーで指定されます)。 *COS ネーム・サーバーを使用する場合、このフィールドの値は無視されます。*

Node Root Relative Distinguished Name:
LDAP ネーム・サーバーを使用する場合、LDAP 管理者によって指定されているとおりに、LDAP ノード・ルートの識別名を入力します(LDAP ノード・ルートの識別名は、**-Dcom.ibm.ws.naming.ldap.noderootrdn** プロパティーで指定されます)。 *COS ネーム・サーバーを使用する場合、このフィールドの値は無視されます。*

JNDI Initial Context Factory:
該当する JNDI 初期コンテキスト・ファクトリーをドロップダウン・リストから選択します。ご使用の Web アプリケーション・サーバーが WebSphere である場合、使用するファクトリーは、以下のものによって異なります。

- 使用している WebSphere のバージョン

- WebSphere の場所。すなわち、Windows NT などの分散プラットフォーム上であるか、z/OS などのホスト・プラットフォーム上であるか。
- 使用しているネーム・サーバーのタイプ (COS ネーム・サーバーまたは LDAP)

表 19 は、ご使用の Web アプリケーション・サーバーが WebSphere である場合に指定する、正しい初期コンテキスト・ファクトリーを示しています。

表 19. WebSphere のバージョンと場所およびネーム・サーバーのタイプに応じた、初期コンテキスト・ファクトリーの設定

WebSphere のバージョン	**Web アプリケーション・サーバーの場所**	**ネーム・サーバーのタイプ**	**使用する初期コンテキスト・ファクトリー**
3.5	分散	COS	`com.ibm.ejs.ns.jndi.CNInitialContextFactory`
3.5	分散	LDAP	`com.ibm.jndi.LDAPCtxFactory`
3.5	z/OS	COS	`com.sun.jndi.cosnaming.CNCtxFactory`
3.5	z/OS	LDAP	`com.sun.jndi.ldap.LdapCtxFactory`
4 以降	分散	COS または LDAP	`com.ibm.websphere.naming.WsnInitialContextFactory`
4 以降	z/OS	COS	`com.sun.jndi.cosnaming.CNCtxFactory`
4 以降	z/OS	LDAP	`com.sun.jndi.ldap.LdapCtxFactory`

ご使用の Web アプリケーション・サーバーが WebSphere でない場合は、適切な値をドロップダウン・リストから選択してください。

注: このドロップダウン・リストには、複数の初期コンテキスト・ファクトリー・クラスに加えて、「デフォルト」のリスト項目が入っています。このサンプル・アプリケーションは、デフォルト・リスト項目の値を次のように割り当てます。

a. Java クラスパスで `com.ibm.websphere.naming.WsnInitialContextFactory` クラスが検出される場合、このサンプルはそれをデフォルト項目にします。このクラスは、`com.ibm.ejs.ns.jndi.CNInitialContextFactory` と `com.ibm.jndi.LDAPCtxFactory` の両方をラップする「ラッパー」クラスです。このサンプルは、「**Provider URL**」フィールドで指定されたネーム・サーバーのタイプを調べることによって、使用する正しい基本クラスを判別します。指定されたプロトコルが「iiop」である場合、このサンプルは `com.ibm.ejs.ns.jndi.CNInitialContextFactory` を使用します。「ldap」である場合、このサンプルは `com.ibm.jndi.LDAPCtxFactory` を使用します。

b. `com.ibm.websphere.naming.WsnInitialContextFactory` クラスが Java クラスパスで検出されない 場合、このサンプルは、「**Provider URL**」フィールドで指定されたネーム・サーバーのタイプを調べることによって、使用する正しいクラスを判別します。指定されたプロトコルが「iiop」である場合、このサンプルは `com.ibm.ejs.ns.jndi.CNInitialContextFactory` を使用します。「ldap」である場合、このサンプルは `com.ibm.jndi.LDAPCtxFactory` を使用します。

ドロップダウン・リスト内の値がご使用のシステムに有効でない場合、「Other」ラジオ・ボタンを選択し、下部のテキスト・フィールドに正しい値を入力してください。

5. SUBMIT ボタンを押します。これにより、サーブレットが起動し、アプリケーションが実行されます。

   アプリケーションが正しく構成され、入力値が有効である場合、SampleResults JSP は、Web ブラウザーにカスタマーの詳細を表示します。 図 22 は、正常な照会の結果を示しています。

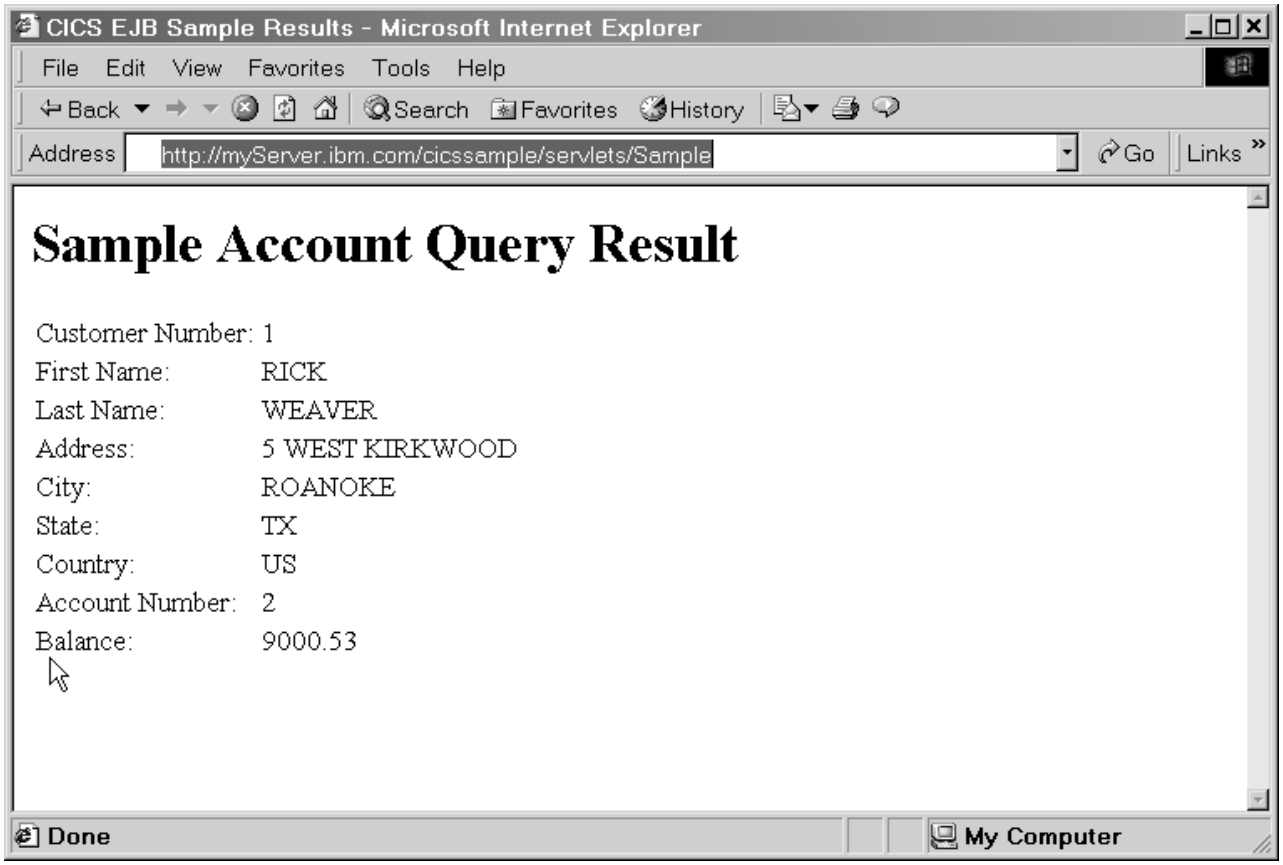


図 22. EJB Bank Account サンプル・アプリケーションの結果画面

アプリケーションが正しく構成されないか、1 つ以上の入力値が無効である場合、SampleError JSP は、Web ブラウザーにエラー・メッセージを表示します。readme.txt ファイルには、障害が起きたアプリケーションのデバッグに役立つヒントが記載されています。

分散トランザクションに関する注記:

分散トランザクションをサポートするために、複数のプロトコルが存在します。

CICS Enterprise Java 環境は、CORBA オブジェクト・トランザクション・サービス (OTS) プロトコルのみをサポートします。しかし、いくつかの J2EE 準拠 Web アプリケーション・サーバー (WebSphere など) はこのプロトコルを使用しないか、デフォルトでこのプロトコルを使用しません。純粋な OTS 分散トランザクション

を使用するように、WebSphere を構成できます。OTS を使用するように WebSphere をセットアップする方法について詳しくは、Bank Account サンプルに付属の readme.txt ファイルを参照してください。

Web アプリケーション・サーバー上のオブジェクトが、既存のトランザクション・コンテキストの有効範囲内で CICS エンタープライズ Bean を呼び出す場合、CORBA OTS を使用するように Web アプリケーション・サーバーをセットアップする必要があります。

分散トランザクションを使用するためのサンプルの変更:

この方法を試行すると、ご使用の J2EE Web アプリケーション・サーバーに CICS との完全な互換性があるかどうかをテストできます。

このタスクについて

デフォルトで、EJB Bank Account サンプルは、分散トランザクションを使用するように構成されていません。ただし、これを変更できます。SampleServlet サーブレットには、クライアント区分トランザクションをオンにするためのサンプル・コード (コメント化されています) が入っています (SampleServlet.java ソース・ファイルは CicsSample.ear ファイル内にあります)。

クライアント区分トランザクションをオンにするには、次の手順を実行します。
1. SampleServlet.java でトランザクション関連のコードをアンコメントします。
2. SampleServlet サーブレットを再コンパイルします。
3. このサーブレットの更新されたコピーを Web アプリケーション・サーバーにインストールします。

クライアント区分トランザクションを使用するようにサンプルをセットアップしたものの、ご使用の J2EE Web アプリケーション・サーバーが純粋な OTS トランザクションをサポートしない (または使用するように構成されていない) 場合、このサンプルを実行すると、CICS は org.omg.CORBA.INVALID_TRANSACTION 例外をスローします。これは、トランザクション・コンテキストが送信されたものの、CICS がそれを使用できなかったからです。

エンタープライズ Bean のトランザクション属性の変更:

エンタープライズ Bean のトランザクション属性を「Supports」から「Mandatory」に (デプロイメント記述子で) 変更することもできます。

これを行う場合、CICS が Bean のリモート・メソッドの起動を許可するのは、既存の OTS トランザクション・コンテキストが、呼び出しでクライアントの環境から渡される場合のみです。

一方、エンタープライズ Bean のトランザクション属性を「Supports」に設定したままにすると、CICS は、メソッド起動をクライアントのトランザクション・コンテキストにバインドします (このようなコンテキストが存在する場合)。そうしないと、メソッドはアトミック・トランザクションで実行され、他の Bean の呼び出し時に新しいトランザクション・コンテキストを伝搬しません。

トランザクション属性を変更するには、「*CICS Operations and Utilities Guide*」で説明されている Assembly Toolkit (ATK) を使用できます。トランザクション属性を変更したら、その変更を有効にするために以下の手順を実行する必要があります。

1. 更新された `SampleEJB.jar` ファイルをピックアップ・ディレクトリーに保管します (前のバージョンを上書きします)。
2. `CEMT CORBASERVER(corbaserver_name) PERFORM SCAN` コマンドを発行します。

トランザクション属性を「Mandatory」に設定したものの、クライアント区分トランザクションを使用するようにサーブレットを更新しない場合、このサンプルを実行すると、CICS は `javax.transaction.TransactionRequiredException` をスローします。これは、トランザクション・コンテキストが送信されていないからです。

データ変換に関する注記:

テキスト・データを表すために、Java プログラムは常に Unicode 文字セットを使用しますが、CICS TS プログラムは EBCDIC を使用します。

Java プログラムまたはエンタープライズ Bean が CICS TS サーバー・プログラムを呼び出す場合、サーバー・プログラムの通信領域にある任意のテキスト値は、入力時に Unicode から EBCDIC に、出力時に EBCDIC から Unicode に変換されなければなりません。EJB Bank Account サンプルのエンタープライズ Bean は、このデータ変換を自動的に処理する CCI Connector for CICS TS を使用します。378 ページの『データ変換と CCI Connector for CICS TS』を参照してください。

注: COBOL プログラム DFH0CSTD によって戻されるテキスト・データのみが、EBCDIC から Unicode に変換されます (サーバー・プログラム DFH0ACTD にも、DFH0CSTD への入力時にも変換は不要です。通信領域にテキスト値がないからです)。

エンタープライズ Bean の作成

セッション Bean を作成できます。これらの Bean で使用されるインターフェースは、CICS サービスとリソースにマップされます。Bean は他の任意の EJB 準拠サーバーに移植可能です。

セッション Bean は、「*Enterprise JavaBeans Specification, Version 1.1*」で定義されるインターフェースを使用します。仕様をダウンロードするには、Oracle Technology Network Java Web サイトに進み、「*Enterprise JavaBeans specification*」を検索して仕様の Web ページを見つけてください。

また、JCICS クラスを使用して CICS サービスやリソースに直接アクセスするセッション Bean を作成することもできます。これらの Bean は、他の CICS EJB サーバーのみに移植可能です。

CICS はエンティティー Bean をサポートしません。すなわち、CICS EJB サーバーでエンティティー Bean を実行できません (CICS EJB サーバーで実行中のセッション Bean またはプログラムは、非 CICS EJB サーバーで実行中のエンティティー Bean と通信できます)。

「*Enterprise JavaBeans Specification, Version 1.1*」をサポートする任意の統合開発環境 (IDE) を使用するワークステーションで Bean を作成できます。

CICS 用の新規 Java エンタープライズ Bean およびプログラムを開発する場合、SDK 5.0 レベルで Java 2 をサポートするアプリケーション開発環境を使用する必要があります。以下を**行わないでください**。

- CICS でサポートされるよりも新しいバージョンの Java SDK のみでサポートされる API 呼び出しを使用すること。
- CICS でサポートされるより後のバージョンの「*Enterprise JavaBeans Specification*」のみでサポートされる機能を使用すること (現在、CICS は「*Enterprise JavaBeans Specification, Version 1.1*」をサポートしています)。

EJB 1.0 仕様に合わせて開発されたすべてのエンタープライズ Bean は、提供される開発ツールを使用して、EJB 1.1 仕様レベルにマイグレーションする必要があります。351 ページの『CICS システムでのエンタープライズ Bean 用のデプロイメント・ツール』を参照してください。

『セッション Bean のコーディング』は、IDE を使用せずにセッション Bean を作成するのに必要なステップの例を示しています。

CCI Connector for CICS TS を使用すると、既存の CICS プログラムを利用するエンタープライズ Bean を作成できます。CCI Connector for CICS TS、およびその使用法については、369 ページの『CCI Connector for CICS TS』を参照してください。

Bean の実行の準備

エンタープライズ Bean をインストールし、実行のための準備をするプロセスは、**デプロイメント**と呼ばれます。

CICS は、ホスト CICS 環境へのエンタープライズ Bean のデプロイメントを管理するために、ワークステーション・ベースのツールを提供します。

デプロイメント・ツールのワークステーションおよび WebSphere コンポーネントは、1 組の InstallShield パッケージとして提供されます。これらのパッケージを z/OS システムからダウンロードするか、ターゲット・ワークステーション上で提供された CD から実行することができます。

デプロイメント・プロセスの説明については、351 ページの『エンタープライズ Bean のデプロイ』を参照してください。ツールの使用については、352 ページの『エンタープライズ Bean 用の CICS デプロイメント・ツールの使用』を参照してください。

セッション Bean のコーディング

このセクションでは、非常にシンプルなセッション Bean のコーディング方法を説明します。

このセクションのステップを完了したら、デプロイメントの準備ができた JAR ファイルが作成されます。デプロイメント・プロセスと使用可能なツールについては、351 ページの『エンタープライズ Bean のデプロイ』を参照してください。

ここに示されている Bean 例は、カジノのルーレット盤をシミュレートしています。ルーレット盤は、2 つのステートフル・フィールドを含むステートフル・セッション Bean です。最初のフィールドは、ルーレット盤の現行の番号です。2 番目

のフィールドは、ギャンブラーが持つ賭け金のクレジット金額です。クライアントはルーレット盤を作成し、オプションとして賭け金の額を指定します (金額が指定されない場合、デフォルトで 100 ドルに設定されます)。クライアントは、出てくる色に賭けることができます。ルーレット盤が回転し、勝ったかどうかを呼び出し元に知らせます。次に、クライアントは勝ち金額を受け取るか、賭けを続行することができます。

コーディングが必要な次の 3 つの要素があります。

1. 『ホーム・インターフェースのコーディング』.
2. 『リモート・インターフェースのコーディング』.
3. 339 ページの『Bean 実装のコーディング』.

次に、プログラムのコンパイルとパッケージが必要です。

1. 341 ページの『コードのコンパイル』
2. 341 ページの『コードのパッケージ化』

ホーム・インターフェースのコーディング:

Bean のホーム・インターフェースは javax.ejb.EJBHome インターフェースを拡張します。このインターフェースは、クライアント・プログラムが Bean インスタンスを作成するために呼び出すことができる、1 つ以上の create メソッドを定義します。

ステートレス・セッション Bean には、パラメーターを取らない正確に 1 つの create メソッドが必要です。ステートフル・セッション Bean は、さまざまな組み合わせのパラメーターを取る異なるバリアントで create メソッドを多重定義する場合があります。RouletteWheel Bean はステートフル・セッション Bean です。ルーレット盤インスタンスの作成時にそのインスタンスで持つクレジット金額を指定できるように、create を多重定義します。

```
package casino;

     public interface RouletteWheelHome extends javax.ejb.EJBHome {

       public RouletteWheel create()
         throws javax.ejb.CreateException, javax.ejb.EJBException;

       public RouletteWheel create(int dollars)
         throws javax.ejb.CreateException, javax.ejb.EJBException;
     }
```

リモート・インターフェースのコーディング:

Bean のリモート・インターフェースは javax.ejb.SessionBean インターフェースを拡張します。リモート・インターフェースは、クライアント・プログラムが個々の Bean インスタンスで呼び出すことができる実際のビジネス・メソッドを定義します。

```
package casino;

     public interface RouletteWheel extends javax.ejb.EJBObject {

       // Place a bet on either "red" or "black" of the given amount,
       // the return value indicates to the caller whether the bet was
       // successful or not.
```

```
  public String bet(String bet,int amount) throws javax.ejb.EJBException;

  // Check the current status of the wheel.
  public String getCurrentStatus() throws javax.ejb.EJBException;

  // Collect winnings from the wheel (if any!)
  public int collectWinnings() throws javax.ejb.EJBException;

}
```

Bean 実装のコーディング:

このクラスは、Bean リモート・インターフェースで定義されるビジネス・メソッドを実装します。

また、SessionBean で抽象であると宣言される標準メソッドを定義します。したがって、Bean 実装が完了するには、これらのメソッドが実装されなければなりません。最後に、ホーム・インターフェースで create メソッドを多重定義したので、同じセットのパラメーターを受け入れる、Bean 実装内で一致する ejbCreate メソッドを提供する必要があります。これは、Bean 実装クラスが、Bean コードを書き込む唯一の場所であるからです。 338 ページの『ホーム・インターフェースのコーディング』で定義したホーム・インターフェースの実装は、ツールによって生成されます。したがって、多重定義された create メソッドを実装する必要がある場合は、ここでそれを行う必要があります。

```
package casino;

    import java.util.Random;
    import javax.ejb.*;

    public class RouletteWheelBean implements SessionBean {

      // Necessary code to fulfill SessionBean interface definition.

      private SessionContext ctx = null;

      public void ejbActivate() throws javax.ejb.EJBException {}
      public void ejbPassivate() throws javax.ejb.EJBException {}
      public void ejbRemove() throws javax.ejb.EJBException {}
      public SessionContext getSessionContext() { return ctx; }
      public void setSessionContext(SessionContext ctx) throws
        javax.ejb.EJBException { this.ctx = ctx;
      }

      ////////////////////////////
      // The bean state information
      private int wheelValue;

      private int currentCredit;

      ////////////////////
      // Our create methods

      public void ejbCreate() throws javax.ejb.EJBException, CreateException {
        currentCredit = 100;
        wheelValue = ((int)System.currentTimeMillis())%37;
      }

      public void ejbCreate(int credit) throws javax.ejb.EJBException,
        CreateException { currentCredit = credit;
        wheelValue = ((int)System.currentTimeMillis())%37;
      }
```

```
////////////////////////////////////////////////////////////////////////
// Implementations of the remote methods the client may call on an instance

//
// Place a bet, either "red" or "black" for the specified amount.
// Then simulate the wheel spinning and construct a response string
// indicating the outcome to the caller.
//
public String bet(String color,int amount) throws javax.ejb.EJBException {

  if (!color.equalsIgnoreCase("red") && !color.equalsIgnoreCase("black"))
    return new String("You can only bet on red or black");

  if (amount > currentCredit)
    return new String("You only have $"+currentCredit+" !");

  // Use the current wheel value as the random number seed
  Random randomizer = new Random((long)wheelValue);

  // Spin the wheel
  wheelValue = Math.abs(randomizer.nextInt()) % 37;

  // Construct a reply
  StringBuffer result =
    new StringBuffer("Number: "+wheelValue+" Color: "+color(wheelValue)+"\n");

  // Did the caller win?
  if (color(wheelValue).equalsIgnoreCase(color)) {
    currentCredit+=(amount*2);
    result.append("Well Done! You won $");
    result.append((amount*2));
  } else {
    currentCredit -= amount;
    result.append("Bad Luck! You lost $");
    result.append(amount);
  }
  result.append(", you now have $");
  result.append(currentCredit);
  return result.toString();
}


//
// Return the current status of this roulette wheel instance.
// The number and color
// it is currently on and the amount of credit the client still has to gamble.
//
public String getCurrentStatus() throws javax.ejb.EJBException {
  return new String("Number:"+wheelValue+" Color:"+color(wheelValue)+"
  You have $"+currentCredit);
}


//
// Allow the client to collect his winnings, then zero the credit so
// they cannot collect twice!
//
public int collectWinnings()throws javax.ejb.EJBException {
  int winnings = currentCredit;
  currentCredit = 0;
  return winnings;
}


//
// Convert a number on the wheel into a color
//
```

```
  private String color(int value) {
    if (value == 0) return "Green";
    if (value % 2 == 0) return "Black";
    return "Red";
  }

}
```

コードのコンパイル:

基本 SDK の他に必要なものは、javax.ejb インターフェースを含む JAR ファイルのみです。

これは、Java インストールの standard/ejb/1_1 ディレクトリーにある ejb11.jar として使用可能です。CLASSPATH に ejb11.jar を追加する場合、記述されるクラスとインターフェースをコンパイルできなければなりません。

コードのパッケージ化:

コンパイルされたクラスは、デプロイメントの準備ができた JAR ファイルにパッケージされなければなりません。

クラス・ファイルがサブディレクトリー casino にあることを前提とすると、次の jar コマンドを使用できます。

```
jar -cvf casino.jar casino\*.class
```

クライアント・プログラムの作成

クライアント・プログラムは、エンタープライズ Bean を呼び出す任意のプログラムです。

クライアント・プログラムは次のいずれかです。

1. 同じ CICS で実行される別のエンタープライズ Bean、JavaBean、Java プログラム、またはオブジェクト
2. 別の CICS で実行されるエンタープライズ Bean、JavaBean、Java プログラム、またはオブジェクト
3. 非 CICS システムまたはワークステーションで実行されるエンタープライズ Bean、JavaBean、Java プログラム、またはオブジェクト

クライアントは、CICS サーバー環境と共用する JNDI ネーム・スペースを使用することによって、呼び出したいエンタープライズ Bean の Bean ホームへの参照を取得します。

ネーム・スペース内のオブジェクト参照の作成:

オブジェクト参照を作成するには、CICS 領域にインストールされている Bean を公開する必要があります。

このタスクについて

これを行うには次の 2 とおりの方法があります。

1. サーバー CICS システムで PERFORM DJAR(XXXX) PUBLISH を発行します。これを行うには、次のいずれかの方法を使用できます。
   - CEMT

- CICSPlex SM
- CICS アプリケーション

指定された DJAR からインストールされる Bean ごとに、オブジェクト参照がネーミング・ディレクトリー・サーバーに公開されます。ネーム・サーバーの使用については、 430 ページの『ネーム・サーバーの定義』を参照してください。

2. 複数の DJAR を単一の CORBASERVER にインストールした場合、PERFORM CORBASERVER(XXXX) PUBLISH コマンドを使用して、その CORBASERVER の下に現在インストールされているすべての Bean を公開することができます。Bean のオブジェクト参照が表示されるネーム・スペース内のサブコンテキストは、DJAR がインストールされた CORBASERVER のリソース定義で定義される JNDI 接頭部で決まります。

撤回が暗黙的に行われることは決してありません。Bean を「非公開にする」推奨方法は、PERFORM DJAR(XXXX)/CORBASERVER(XXXX) RETRACT を発行することです。DJAR または CORBASERVER が破棄される場合、Bean オブジェクト参照は引き続きネーム・スペースに存在しますが、実際の Bean が CICS 内に存在しなくなるので、それらの参照をクライアントは使用できません。DJAR を再インストールし、それらの参照を撤回することが可能です。

JNDI を使用した Bean 参照の取得:

Java Naming and Directory Interface (JNDI) は、Java プログラムにネーミングとディレクトリー機能を提供する、Java プログラミング言語で指定されたアプリケーション・プログラミング・インターフェース (API) を定義します。

また、各種のディレクトリーおよびネーミング・サービス・ドライバーを接続できるようにするサービス・プロバイダー・インターフェース (SPI) も定義します。図 23 は、JNDI API を使用して Java アプリケーションとインターフェースを取り、JNDI SPI を介して各種ネーム・サーバーとインターフェースを取る Naming Manager を示すことによってこの仕組みを図示しています。

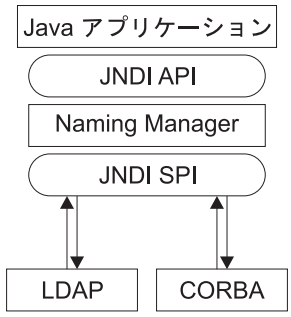


図 23. JNDI 構造

JNDI インターフェースは Web サイト (http://www.oracle.com/technetwork/java/index.html) で説明されています。

サーバー・システムの管理者によってエンタープライズ Bean がネーム・サーバーに登録された後、クライアント・アプリケーションは JNDI インターフェースを使用して、そのホーム・インターフェースを見つけることができます。

Java Naming and Directory Interface (JNDI) バージョン 1.2 をサポートする適切なネーム・サーバーをセットアップし、その場所を CICS に対して定義します。詳しくは、432 ページの『LDAP サーバーのセットアップ』および 443 ページの『COS Naming Directory Server のセットアップ』を参照してください。必要な JVM プロパティーについて詳しくは、127 ページの『JVM システム・プロパティー』を参照してください。

LDAP を使用するためのクライアント・プログラムの作成:

CICS Transaction Server は LDAP をサポートします。クライアント・プログラムが CICS 領域から公開された Bean ホームを検出できるようにするには、クライアント・プログラムに何らかの変更を加える必要がある場合があります。

LDAP クライアントは、WebSphere コンテキスト・ファクトリー、または Java によって提供される LDAP コンテキスト・ファクトリーのどちらかを使用する必要があります。WebSphere コンテキスト・ファクトリーを使用する利点は、システム・ネーム・スペース (すなわち、CICS が Bean ホームを公開する LDAP サーバー上の構造化されたネーム・スペース) を自動的に認識することです。ただし、このコンテキスト・ファクトリーには、複数の依存関係があるので、最も軽量なクライアントではありません。Java によって提供されるコンテキスト・ファクトリーは、基本 IBM Developer Kit for the Java Platform を別として、依存関係はありません。したがって、非常に軽量です。しかし、システム・ネーム・スペースを認識しないので、プログラマチックにネゴシエーションする必要があります。ただし、これに役立つように、CICS によって提供されるいくつかのユーティリティー・メソッドがあります。

これらの選択肢は、以下の例で最もよく実証されています。

***WebSphere* コンテキスト・ファクトリー:**

WebSphere コンテキスト・ファクトリーを使用して HelloWorld Bean のホームを見つけるクライアント・ソース・コードの例は、次のとおりです。

```
import org.omg.CORBA.ORB;
import java.io.*;
import javax.naming.*;
import examples.helloworld.*;
import java.util.*;

public class WASNamingClient {
   public static void main(String[] argv) {
      try {

// Set the necessary properties
      Properties prop = new Properties();

// These four are *fixed* values, you never need to change them.

  prop.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.ibm.websphere.naming.WsnInitialContextFactory");

  prop.put("com.ibm.websphere.naming.namespaceroot","bootstraphostroot");
  prop.put("com.ibm.ws.naming.ldap.config","local");
  prop.put("com.ibm.ws.naming.implementation","WsnLdap");

// These two depend on your server settings and should match your CICS region settings

  prop.put("com.ibm.ws.naming.ldap.containerdn","ibm-wsnTree=WASNaming,c=us");
```

```
        prop.put("com.ibm.ws.naming.ldap.noderootrdn",
    "ibm-wsnName=legacyroot,ibm-wsnName=PLEX2,ibm-wsnName=domainRoots");

    // Finally, instead of com.ibm.cics.ejs.nameserver,
    // set com.ibm.ws.naming.ldap.masterurl to your destination LDAP server

           prop.put("com.ibm.ws.naming.ldap.masterurl","ldap://wibble.example.com:389");

           InitialContext ctx = new InitialContext(prop);
           org.omg.CORBA.Object obj =
                (org.omg.CORBA.Object)ctx.lookup("samples/HelloWorld");

           HelloWorldHome hhome =
                (HelloWorldHome)javax.rmi.PortableRemoteObject.narrow
                (obj,HelloWorldHome.class);

           System.out.println("HelloWorldHome successfully found!");
           HelloWorld hello = hhome.create();
           System.out.println(hello.sayHello());

     } catch (Exception e) {
       System.err.println("Exception while looking up and calling the HelloWorld bean:");
       e.printStackTrace();
      }
    }
}
```

コメントに記されているように、最初の 4 つのプロパティーは固定です。残りの 3 つは、CICS 領域の設定と一致します (ただし、**-Dcom.ibm.cics.ejs.nameserver** プロパティーが com.ibm.ws.naming.ldap.masterurl になっています)。しかし、WebSphere コンテキスト・ファクトリーは、WebSphere のコンポーネントに依存しています。したがって、コマンド行からこれを実行するには、ご使用の環境を適切にセットアップするためのスクリプトを実行する必要があります。

スクリプト DFHWAS4Setup.bat は、CICS で提供されるコマンド行です。このスクリプトは、CICS がインストールされる z/OS UNIX 領域内の utils サブディレクトリーからダウンロードできます。これは、WebSphere がインストールされているシステムで実行しなければなりません。WebSphere がインストールされている場所 (例えば、c:\WebSphere\AppServer) を指すように設定されている環境変数 WAS_HOME を利用するからです。このスクリプトが実行されると、CLASSPATH をさらに拡張して、エンタープライズ Bean に必要なクライアント・サイド・コードを組み込む必要があります。上記の例の場合、これは HelloWorld.jar です。上記のコードをコンパイルし、実行することができます (このコード例では、JNDI 接頭部が *samples* である CorbaServer でホームが公開されることを前提としています)。

CICS では、**-Dcom.ibm.cics.ejs.nameserver =** <*hostname*> を設定しますが、このクライアント・プログラムでは **com.ibm.ws.naming.ldap.masterurl =** <*hostname*> を設定します。CICS は前者を認識し、WebSphere は後者を認識します。

Java に付属の LDAP コンテキスト・ファクトリー:

IBM Developer Kit for the Java Platform 構成の観点から見ると、Java に付属の LDAP コンテキスト・ファクトリーを使用する方がずっと簡単です。LDAP コンテキスト・ファクトリーは、IBM Developer Kit for the Java Platform ベースで提供され、それ以外の依存関係がないからです。

しかし、このコンテキスト・ファクトリーは、WebSphere 用に構成されている任意の LDAP サーバーに存在するネーム・スペース構造を認識しないため、クライアント・アプリケーション・プログラマーにとっては要件が厳しくなる場合があります。CICS は、この複雑さが増すことを緩和するネーム・スペース・ヘルパー機能を備えています。com.ibm.cics.portable.CICSNameSpaceHelper クラスが CICSEJBClient.jar で提供されています。この JAR ファイルは、CICS がインストールされる z/OS UNIX 領域内の utils サブディレクトリーで入手可能です。

このクラスを使用する例は次のとおりです。

```
 import org.omg.CORBA.ORB;
 import java.io.*;
 import examples.helloworld.*;
 import javax.naming.*;
 import javax.naming.directory.*;
 import java.util.*;
 import com.ibm.cics.portable.CICSNameSpaceHelper;

 public class SUNNamingClient {

 public static void main(String[] argv) {

    try {
       Hashtable env = new Hashtable();

// Set up the first two obvious properties, the LDAP factory and LDAP server supplied with Java
       env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
       env.put(Context.PROVIDER_URL,   "ldap://wibble.example.com:389");
// These two settings match the values from the CICS system
       env.put("com.ibm.ws.naming.ldap.containerdn",  "ibm-wsnTree=WASNaming,c=us");
       env.put("com.ibm.ws.naming.ldap.noderootrdn",
             "ibm-wsnName=legacyroot,ibm-wsnName=PLEX2,ibm-wsnName=domainRoots");

// Use the LDAPSNSLookup helper method to negotiate the WebSphere System Name
// Space on wibble.example.com and locate our HelloWorld bean.  "samples"
// is the JNDI prefix on the CICS CorbaServer that published the HelloWorld Bean.
       org.omg.CORBA.Object obj =
       CICSNameSpaceHelper.LDAPSNSLookup(env,"samples/HelloWorld");

       HelloWorldHome hhome =
              (HelloWorldHome)javax.rmi.PortableRemoteObject.narrow
              (obj,HelloWorldHome.class);

       System.out.println("HelloWorld home successfully found!");
       Hello hello = hhome.create();
         System.out.println(hello.sayHello());
 } catch (Exception e) {
   System.err.println("Exception while looking up and calling the HelloWorld bean:");
  e.printStackTrace();
 }
  }
}
```

WebSphere コンテキスト・ファクトリーの例で使用される masterurl プロパティーではなく、providerURL プロパティーを認識する、Java に付属の LDAP コードを使用します。

ヘルパー・クラス CICSNameSpaceHelper は、他のコンテキスト・ファクトリーとも連携する場合があります。LDAPSNSLookup に渡される名前の構文は JNDI 構文 a/b/c/d であることに注意してください。

COS Naming を使用するためのクライアント・プログラムの作成: 以下の例は、337 ページの『セッション Bean のコーディング』で作成された RouletteWheel Bean と連携するクライアント・プログラム Gambler.java を示しています。Bean 参照が COS Naming ネーム・スペースから得られる場合、クライアントがその参照を使用する前に実行しておく必要がある複数のオペレーションがあります。これらのオペレーションは、ユーティリティー・クラス EJBUtils で収集されるため、大部分のクライアント・プログラムで同じです。このユーティリティー・クラスは、クライアント・プログラム Gambler で使用されます。

EJBUtils.java:

ユーティリティー・クラス EJBUtils の実装は次のとおりです。

```
import javax.naming.*;
import java.util.Hashtable;

class EJBUtils {

  public static Object jndi_lookup(String name, Class resultClass) {

    // Set up environment for creating initial context
    Hashtable env = new Hashtable(11);

    // Define the nameserver - see note 1 below
    env.put(Context.PROVIDER_URL,
      "iiop://wibble.example.com:900");

    // Define the initial context factory -see note 2
    env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.sun.jndi.cosnaming.CNCtxFactory");

    try {

      // Create the initial context
      Context ctx = new InitialContext(env);

      // Lookup the object
      Object tempObject = ctx.lookup(name);

      // Narrow that to the requested class
      return javax.rmi.PortableRemoteObject.narrow(tempObject,resultClass);

    } catch (NamingException ne) {
      System.err.println("EJBUtils.jndi_lookup() failed:");
      ne.printStackTrace();
    }
    return null;
  }

}
```

注:

1. ここでは、Bean の検索に使用されるネーム・サーバーを「iiop://wibble.example.com:900」として定義します。この値は、ご使用のネーム・サーバーの名前であり、CICS JVM プロパティー・ファイルで定義された **-Djava.naming.provider.url** と一致する必要があります。その結果、クライアントは、CICS によって公開されたのと同じネーム・サーバーで Bean を検索します。ネーム・サーバーの使用については、430 ページの『ネーム・サーバーの定義』を参照してください。

2. ここでは、クライアント環境の初期コンテキスト・ファクトリーを定義します。クライアント環境で必要な値に設定する必要があります。この例は、IBM SDK に付属の ORB を使用する場合に設定される値を示しています。クライアントが、CICS Transaction Server for z/OS バージョン 2 で実行される Java アプリケーションまたはエンタープライズ Bean である場合、ここで初期コンテキスト・ファクトリーを指定するのではなく、デフォルトで `com.ibm.websphere.naming.wsnInitialContextFactory` に設定されるままにする必要があります。

Gambler.java:

クライアント・プログラム例 `Gambler.java` の実装は次のとおりです。

```
import org.omg.CORBA.ORB;
import java.io.*;
import casino.*;

public class Gambler {

  public static void main(String[] argv) {

    try {

      System.out.println("Gambler\n");

      System.out.println("Looking up RouletteWheel home");
      RouletteWheelHome wheelHome =
        (RouletteWheelHome)
        EJBUtils.jndi_lookup("cics/ejbs/RouletteWheel",
                             RouletteWheelHome.class);
        //
        // See Note 1.
        //
      System.out.println("Creating a new roulette wheel");
      RouletteWheel wheel = wheelHome.create();

      System.out.println("");
      System.out.println("Gambling $50 on red !");
      System.out.println(wheel.bet("red",50));

      System.out.println("");
      System.out.println("Gambling $20 on black !");
      System.out.println(wheel.bet("black",20));

      System.out.println("");
      System.out.println("Gambling $20 on red !");
      System.out.println(wheel.bet("red",20));

      System.out.println("");
      System.out.print("Collecting winnings:$");
      System.out.println(wheel.collectWinnings());


      System.out.println("");
      System.out.print("Removing the roulette wheel");
      wheel.remove();

    } catch (Exception e) {
      System.err.println("Error while gambling:");
      e.printStackTrace();
    }
```

```
    }
  }
```

注:

1. クライアント・プログラム Gambler.java は、ネーム・スペース内の「cics/ejbs」で RouletteWheel を検索します。これは、RouletteWheel Bean をインストールした先の CICS 内の CORBASERVER に、JNDI 接頭部 cics/ejbs が必要であることを意味します。インストールされ、公開された後、RouletteWheel は、クライアント・プログラムからアクセス可能になります。
2. このクライアント・プログラムの最後に `remove` 呼び出しがあります。Roulette Wheel Bean はステートフルであり、CICS はすべてのインスタンスの状態を管理します。その Bean インスタンスで操作を完了するときに `remove` が呼び出される場合を除いて、CICS は引き続き状態を保管します。Bean のタイムアウトは、CORBASERVER リソース定義の SESSBEANTIME パラメーターを使用して制御できます。これは、そのインスタンスを使用するための要求が着信しない場合にインスタンスの状態を管理しなければならない時間を CICS に対して指定し、一種のガーベッジ・コレクションを実装します。ただし、このタイプのガーベッジ・コレクションに依存しないように、インスタンスの処理が終了したときに `remove` を呼び出すことを、プログラミングの観点からお勧めします。

クライアント・プログラムの使用:

クライアント・プログラムをコンパイルする場合、CICS Jar Development Tool で以前に正常に処理したデプロイ済み JAR ファイル、および Java インストールの `standard/ejb/1_1` ディレクトリー内の `ejb11.jar` で入手可能な、EJB 1.1 サポート用の javax.ejb インターフェースを組み込むように、クラスパスを慎重に設定する必要があります。

コンパイルした後、次のコマンドでクライアントを実行します。

```
java Gambler
```

Web アプリケーション・サーバーとのトランザクションの相互運用性:

分散トランザクションをサポートするために、複数のプロトコルが存在します。CICS Enterprise Java 環境は、標準の CORBA オブジェクト・トランザクション・サービス (OTS) プロトコルのみをサポートします。しかし、いくつかの J2EE 準拠 Web アプリケーション・サーバー (WebSphere バージョン 4 など) はこのプロトコルを使用しないか、デフォルトでこのプロトコルを使用しません。

*Web アプリケーション・サーバー上のオブジェクトが、既存のトランザクション・コンテキストの有効範囲内で CICS エンタープライズ Bean を呼び出す場合、CORBA OTS を使用するように Web アプリケーション・サーバーをセットアップする必要があります。*これが可能でない場合、Web アプリケーション・サーバーは CICS エンタープライズ Java サポートとの完全な互換性がありません (EJB Bank Account サンプル・アプリケーションを使用して、Web アプリケーション・サーバーが CICS エンタープライズ Java サポートとの完全な互換性があるかどうかをテストする方法については、334 ページの『分散トランザクションに関する注記』を参照してください)。

ご使用の Web アプリケーション・サーバーが WebSphere Application Server バージョン 4 である場合は、デフォルトで標準の CORBA OTS を使用しませんが、使用するようにできることに注意してください。既存のトランザクション・コンテキストの有効範囲内で CICS エンタープライズ Bean を呼び出す WebSphere オブジェクトがある場合、CORBA OTS を使用するように WebSphere をセットアップする必要があります。バージョン 5 以降の WebSphere Application Server のバージョンは、この問題の影響を受けません。

WebSphere Application Server が CORBA OTS を使用するように強制するには、次の手順を実行します。

1. WebSphere 管理コンソールで、「JVM settings」タブを選択します。
2. 「System Properties」セクションで次のように入力します。

   ```
   com.ibm.ejs.jts.ControlSet.interoperabilityOnly=true
   com.ibm.ejs.jts.ControlSet.nativeOnly=false
   ```

   変更内容を保管します。
3. アプリケーション・サーバーを再始動します。

EJB Handle、HomeHandle および EJBMetaData の処理

Enterprise JavaBeans 仕様は、セッション Bean がリモート・インターフェースで定義されたメソッドだけでなく、いくつかの追加メソッドをサポートする方法を記述しています。

- EJBHome インターフェースで定義されるメソッドがあります。これらのメソッドは、以下が必要なクライアントから呼び出し可能です。
  - ホームへの「保管可能な」参照 (ホーム・ハンドル) を取得する、または
  - Bean タイプの EJBMetaData を取得する。

  .
- EJBObject インターフェースで定義されるメソッドがあります。これらのメソッドは、以下が必要なクライアントから呼び出し可能です。
  - EJB のホームを取得する、または
  - オブジェクト自体への「保管可能な」参照 (ハンドル) を取得する。

ハンドルの目的は、ハンドルが直列化可能であり、Bean インスタンスについてハンドルが取得された後、おそらくフラット・ファイルに対して直列化できることです。後でプログラムがその同じインスタンスに対して呼び出しを行う場合、プログラムはハンドルを非直列化し、再度メソッドの呼び出しを開始できます。ハンドルとメタデータ・クラスの実装は、製品固有です。

CICS では、3 つのインターフェース HomeHandle、Handle および EJBMetaData の実装は次のとおりです。

- com.ibm.cics.portable.CICSSessionHomeHandle
- com.ibm.cics.portable.CICSSessionHandle
- com.ibm.cics.portable.CICSEJBMetaData

これらの実装は、CICS がインストールされる z/OS UNIX 領域内の `utils` サブディレクトリーからダウンロード可能な、`CICSEJBClient.jar` JAR ファイルに含まれています。この JAR ファイルは、上記で説明されている特殊なメソッドを呼び出す

任意のクライアント・プログラムの CLASSPATH に含まれていなければなりません。これは、そのクライアント・プログラムがサーバーから戻されるオブジェクトのタイプを確実に認識するためです。例えば、その CLASSPATH に CICSEJBClient.jar が含まれていない場合、エンタープライズ Bean の **getEJBMetaData** 関数を呼び出すクライアント・プログラムは、次のいずれかが戻されることがあります。

1. 例外
2. Null

戻される正確な値は、クライアントのオブジェクト・リクエスト・ブローカー (ORB) の実装によって異なります。

エンタープライズ Bean での EDF の使用

EDF を使用してエンタープライズ Bean をテストするには、次のタスクを実行する必要があります。

このタスクについて

- グループ DFHIIOP で提供される DFJIIRP の PROGRAM リソース定義で、CEDF パラメーターを YES に設定します。
- TRANCLASS(DFHEDFTC) で MAXACTIVE を 1 に設定します。
- トランザクションがトラップされる端末で CEDX (*transid*) を入力して、EDF をアクティブにします。transid は、デフォルトの transid CIRP か、RequestModel 定義で指定されるトランザクションのいずれかです。
- Bean を開始します。

Bean 間の通信:

ご使用の Bean が、同じ AOR 内において同じトランザクション ID で Bean 間通信を使用する場合、MAXACTIVE を 1 に設定すると、通信が機能しません。

これは、2 番目のトランザクションの実行が、実行されるスロットを待機して中断され、オリジナルの Bean が「タイムアウト」条件を検出するからです。これを回避するには、次のいずれかのアクションを取ります。

- REQUESTMODEL を使用して、Bean ごとに固有のトランザクション ID を指定します。
- すべての create メソッドが CIRP (デフォルトのトランザクション ID) を使用できるようにし、REQUESTMODEL を使用してビジネス・メソッドのセットごとに固有のトランザクション ID を定義します。

注: Bean が要求プロセッサー内で実行されるときに、その Bean によって行われるリモート・メソッド呼び出しに現行の要求プロセッサーで対応できない場合、CICS は requestmodel のみを使用します (したがって、新しいトランザクション ID で新しい CICS トランザクションを開始します)。次の場合、現行の要求プロセッサーでローカル側でメソッド呼び出しに対応できません。

- 呼び出されるメソッドのトランザクション属性に、異なるトランザクション・コンテキストが必要である場合
- 呼び出される Bean が異なる CorbaServer 内にある場合

エンタープライズ Bean のデプロイ

このセクションでは、エンタープライズ Bean を CICS EJB サーバーにデプロイするプロセスについてより詳しく説明します。

デプロイメントの概念は、267 ページの『エンタープライズ Bean のデプロイの概要』で紹介されています。

EJB 仕様で使用される「デプロイメント」という用語は、1 つ以上の JAR ファイル内のエンタープライズ Bean が、特定の稼働環境 (この場合、CICS EJB サーバー) で使用できるようにする、一連のタスクを示します。

CICS システムでのエンタープライズ Bean 用のデプロイメント・ツール

CICS は、エンタープライズ Bean を CICS EJB サーバーにデプロイするのを支援する 3 つのツールを提供します。

Assembly Toolkit (ATK)

Assembly Toolkit (ATK) は、ランタイム環境用に準備ができている JAR ファイルを作成するために、CICS を含めて複数の IBM EJB サーバーで使用される汎用ツールです。Assembly Toolkit for Windows は、WebSphere Application Server に付属しています。

ATK の使用について詳しくは、「*CICS Operations and Utilities Guide*」を参照してください。

エンタープライズ Bean 用のリソース・マネージャー

エンタープライズ Bean 用のリソース・マネージャーは Web ベースのツールであり、エンタープライズ Bean の使用をサポートするために CICS にインストールされるリソース (CORBASERVER および DJAR) で所定のオペレーションを実行するのに使用できます。また、EJB 関連の問題診断にもこのツールを使用できます。このツールは、DJAR 定義に関連したエラーを表示する機能を備え、デプロイ済み JAR ファイル内の Bean がネーミング・サービスに公開されたかどうかを示すからです。

エンタープライズ Bean 用のリソース・マネージャーの詳細については、「*CICS Operations and Utilities Guide*」を参照してください。

CREA トランザクション

CREA は CICS 提供のトランザクションであり、インストールされたデプロイ済み JAR ファイルで Bean の REQUESTMODEL リソース定義を作成するのに使用できます。 CREA は、**EXEC CICS CREATE** コマンドを使用することによって、実行中の CICS システムに定義をインストールするか、定義を CSD に書き込むことができます。CREC は、CREA の読み取り専用バージョンです。変更を加える機能を与えることなく、検査機能を提供します。

CREA および CREC の詳細については、「*CICS Supplied Transactions*」の CREA - create REQUESTMODELs for enterprise beans を参照してください。

3270 端末にアクセスする必要なく、CREA および CREC を使用できます。詳しくは、「*CICS インターネット・ガイド*」の CICS から Web への接続を参照してください。

エンタープライズ Bean 用の CICS デプロイメント・ツールの使用

Bean を作成し、CICS にデプロイするには、アプリケーション開発者は、(後の段階で CICS システム・プログラマーと連携して) 複数のステップを実行する必要があります。

このタスクについて

Bean を作成し、デプロイ可能にする

Bean を作成し、JAR ファイルにパッケージします。Bean は、適当なツールを使用して作成し、テストすることができます。

注: JAR ファイルには、1 つ以上のエンタープライズ Bean 用の Java クラスが含まれる場合があります。通常、CICS EJB サーバーで使用される JAR ファイルには、複数のエンタープライズ Bean が含まれます。
Bean が JAR ファイルにパッケージされた後、ATK を使用してデプロイ可能にします。ATK の簡単な概要と詳細情報の参照については、「*CICS Operations and Utilities Guide*」の The enterprise bean deployment tool, ATK を参照してください。

z/OS UNIX ピックアップ・ディレクトリーに保管する

デプロイ可能な JAR ファイルのコピーを、Bean を実行したい CorbaServer の z/OS UNIX ピックアップ・ディレクトリーに保管します。これを行うには、FTP、NFS、または SMB を使用できます。z/OS UNIX ディレクトリーをワークステーションにマウントできる場合、このプロセスは前の JAR ファイル作成プロセスに統合できます。

ピックアップ・ディレクトリーをスキャンする

CEMT またはエンタープライズ Bean 用のリソース・マネージャーのどちらかを使用して、ピックアップ・ディレクトリーのスキャンを開始します (エンタープライズ Bean 用のリソース・マネージャーについては、「*CICS Operations and Utilities Guide*」の The Resource Manager for Enterprise Beans を参照してください)。 CICS は、デプロイ済み JAR ファイルの DJAR 定義を作成し、ピックアップ・ディレクトリーにインストールします。

ピックアップ・ディレクトリーがスキャンされた後、新しい DJAR 定義の状態を表示して、デプロイ済み JAR ファイルがすぐに使用できるかどうかを判別できます。

デプロイ済み JAR ファイルがすぐに使用できない場合、システム・プログラマーが関与する必要なく、アプリケーション開発者がエラーの原因を判別し、大部分の場合、訂正できます。

公開する

デプロイ済み JAR ファイル内の各 Bean のホーム・インターフェースへの参照を、外部ネーム・スペースに公開します。このネーム・スペースは、JNDI を通じてクライアントからアクセス可能です。

CORBASERVER 定義で AUTOPUBLISH(YES) を指定する場合、DJAR 定義が正常に CorbaServer にインストールされるときに、デプロイ済み JAR ファイル内の Bean が自動的にネーム・スペースに公開されます。別の方法として、PERFORM CORBASERVER PUBLISH または PERFORM DJAR PUBLISH コマンドを発行することもできます。

エンタープライズ Bean 用のリソース・マネージャー (「*CICS Operations and Utilities Guide*」の The Resource Manager for Enterprise Beans を参照) は、「自動公開」機能がオンであるか、オフであるかを示します。

追加のクラスがクラスパスにあることを確認する

エンタープライズ Bean については、デプロイ済み JAR ファイルを JVM プロファイルまたは JVM プロパティー・ファイル内のクラスパスに追加する必要がありません。これらのファイルに含まれているクラスの読み込みについては、 CICS が DJAR 定義を通じて管理します。ただし、例えばユーティリティー用のクラスなど、デプロイ済み JAR ファイルに含まれて**いない**クラスがエンタープライズ Bean で使用されている場合は、要求プロセッサー・プログラム用の JVM によって使用されるクラスパスにこれらのクラスを含める必要があります。93 ページの『第 5 章 JVM を使用するアプリケーションの有効化』では、これを行う方法を示します。

単体テスト

デプロイ済み JAR ファイル内の Bean がネーミング・サーバーに公開された後、アプリケーション・プログラマーは、CICS 環境内でそれらの単体テストを行うことができます。

システム・テスト

Bean がシステム・テストに対応可能になると、アプリケーション・プログラマーはシステム・プログラマーと連携して、REQUESTMODEL 定義が必要かどうかを検討します。CICS 提供のトランザクション CREA を使用して、REQUESTMODEL 定義を生成します (CREA については、「*CICS Supplied Transactions*」マニュアルの CREA - create REQUESTMODELs for enterprise beans を参照してください)。

Bean および Bean メソッドをアプリケーションから特定できます。システム・プログラマーは、最適な REQUESTMODEL 定義のセットを生成させることによって、Bean メソッドをトランザクション ID に関連付けることができます。例えば、ワークロード管理のためや、効果的なモニターと統計情報の収集のために、異なる Bean を別々のトランザクション ID で実行するのが便利です。

実稼働環境にインストールする

システム・テストから実稼働環境に移動するには、次の手順を実行します。

1. ATK を使用して、各 JAR ファイルのデプロイメント記述子で設定されたリソースと参照のコンテナー・バインディングが、実稼働環境に適切であることを確認します。
2. ピックアップ・ディレクトリーを識別するために、実動領域の CORBASERVER 定義で DJARDIR パラメーターを設定した場合は、次の手順を実行します。
   a. デプロイ可能な JAR ファイルを CorbaServer のピックアップ・ディレクトリーに保管します。

b. CORBASERVER 定義をインストールします。
c. 適切な DJAR 定義が作成されます。

3. 設定していない場合は、次の手順を実行します。
   a. デプロイ可能な JAR ファイルを、実動領域で使用する予定の z/OS UNIX ディレクトリーに保管します。
   b. 実動 CORBASERVER 定義をインストールします。
   c. ご使用のシステムで通常使用するどのプロセスでも使用して、テスト領域にあったものと同等の DJAR 定義を作成し、インストールします。
4. 実動領域の CORBASERVER 定義で AUTOPUBLISH(YES) パラメーターを設定した場合は、次のとおりです。
   a. DJAR 定義が正常に CorbaServer にインストールされるときに、デプロイ済み JAR ファイル内の Bean が自動的にネーム・スペースに公開されます。
5. 設定していない場合は、次の手順を実行します。
   a. CEMT PERFORM CORBASERVER PUBLISH または CEMT PERFORM DJAR PUBLISH を使用して、実動に使用する JNDI サーバーに Bean を公開します。
6. ご使用のシステムで通常使用するプロセスを使用して、REQUESTMODEL 定義をテスト領域の CSD から実動 CSD に移転します。
7. エンタープライズ Bean 用のデプロイ済み JAR ファイルに含まれていない追加のクラス (ユーティリティー用のクラスなど) が、標準クラスパスに存在することを確実にします。

注: 実動領域でエンタープライズ Bean を更新したい場合は、357 ページの『実動領域でのエンタープライズ Bean の更新』を参照してください。

エンタープライズ Bean 用調整

CICS システムでエンタープライズ Bean を使用している場合には、この調整情報は、以下の事項に役立ちます。

このタスクについて

- エンタープライズ Bean の過度の使用は、EJB Object Store DFHEJOS のサイズを増加させる必要があることを意味しています。 355 ページの『Stateful エンタープライズ Bean の予想される使用量に関する DFHEJOS のカスタマイズ』では、増加させる方法について説明します。
- クライアント制御 OTS (オブジェクト・トランザクション・サービス) トランザクションを使用すると、JVM の要件に影響がでます。 355 ページの『クライアント制御 OTS (オブジェクト・トランザクション・サービス) トランザクションで呼び出されるエンタープライズ Bean』では、確認すべき事項について説明します。

- 単一のエンタープライズ Bean メソッドによる複数の要求プロセッサーの使用では、デッドロックが発生する可能性があります。 『複数の要求プロセッサーを必要とするエンタープライズ Bean メソッド』では、この可能性を除去する方法について説明します。

Stateful エンタープライズ Bean の予想される使用量に関する DFHEJOS のカスタマイズ

EJB オブジェクト・ストア DFHEJOS は、不動態化されている Stateful Session Bean を保管するために使用するファイルです。このファイルは、VSAM ファイルの場合とカップリング・ファシリティー・データ・テーブルの場合があります。CICS では、このファイルを SDFHINST ライブラリーの DFHDEFDS メンバーに作成するためのサンプル JCL が提供されています。

CICS 提供の DFHEJOS の設定は、ストレージの無駄を最小化するために、最大サイズ 8K の少数のオブジェクト (不動態化された Bean) のストレージ用に設計されています。Stateful エンタープライズ Bean を極度に使用することが予測される場合には、このデータ・セットのスペース割り振りおよびレコード・サイズを増加させます。

「*CICS System Definition Guide*」の『Defining the EJB data sets』では、DFHEJOS の作成方法およびレコード・サイズに適切な設定を計算する手順を説明しています。

クライアント制御 OTS (オブジェクト・トランザクション・サービス) トランザクションで呼び出されるエンタープライズ Bean

クライアント制御 OTS (オブジェクト・トランザクション・サービス) トランザクションを使用すると、JVM 要件に影響がでます。

CICS 内の標準的なエンタープライズ Bean の作業負荷は、CICS 内の IIOP リスナー・タスクによって受信される GIOP 要求など、着信 IIOP メッセージで始まります。要求は、要求受信側タスクに渡されます。このタスクは、GIOP メッセージを検査し、そのメッセージの処理を要求プロセッサー・タスクに渡します。最後に、要求プロセッサー・タスクの完了時に、応答は、要求受信側タスクによって要求したクライアントに返されます。

GIOP 要求が、クライアント制御 OTS トランザクションを形成する場合には、要求プロセッサーおよび要求受信局タスクは、OTS トランザクションがコミットされ、ロールバックされるまで終了しません。要求プロセッサーが JVM で稼働中のため、その JVM は OTS トランザクションが終了するまで、他のタスクは使用できません。この状態が頻繁に発生する場合には、JVM プール内の JVM の数を増加させて、着信要求の待ち時間が長くなるのを避ける必要があります。

複数の要求プロセッサーを必要とするエンタープライズ Bean メソッド

このタスクについて

エンタープライズ Bean メソッドの単独の実行で、複数の要求プロセッサーが必要とされている場合には、アプリケーションにデッドロックの問題が発生することがあります。(メソッドは、そのメソッドが、異なる要求プロセッサーで実行する必要

がある 1 つ以上のメソッド (通常はリモート) を呼び出す場合には、「複数の要求プロセッサーが必要」ということが言えます。) デッドロックは、これ以上 JVM が許可されない場合に、JVM を強制的に待機させられているメソッドを満足させるために必要なすべての要求プロセッサーで発生する可能性があります。これは、以下の 2 つが原因で発生します。

1. 単純なケースとしては、CICS (MAXJVMTCBS) で並行して存在することが許可されている最大 JVM 数が、そのメソッド要求をサービスするために必要な要求プロセッサー数よりも小さい場合。
2. 複雑な場合:
   - CICS は、多重要求を同時に処理しています。
   - すべての要求は、他の JVM を待っています。
   - すべての許可された JVM は、現在使用中です。

単純な事例を回避するのは簡単です。複雑な事例を回避するのはより困難です。要求プロセッサー・インスタンスの少なくとも 1 つのメソッドの要件を満足させるには、常に十分なフリー JVM があることを確認することが必要です。

Bean メソッドが使用可能な同時 JVM の最大数は、要求プロセッサー・トランザクションに適用される TRANCLASS 定義の MAXACTIVE 属性によって設定されます。CICS が使用可能な同時 JVM の最大数は、MAXJVMTCBS システム初期設定パラメーターによって設定されます。

複数の要求プロセッサーを使用する Bean メソッドによって発生するデッドロックの可能性を排除するには、以下を行います。

1. アプリケーション要件と整合性がある限り、それぞれのメソッドが必要とする要求プロセッサーの数を最小値にし、可能であれば 1 にするようにしてください。すべてのメソッドの要件をすべてのアプリケーションにおいて 1 つの要求プロセッサーに削減できる場合には、それ以上削減する必要はありません。
2. すべてのメソッドの要件を 1 つの要求プロセッサーに削減することができない場合には、どれが「ワーストケース」なのかを発見します。すなわち、要件を満足させるためにほとんどの要求プロセッサーを必要とする Bean メソッドです。
3. 新規 TRANCLASS 定義を作成します。このトランザクション・クラスは、複数の要求プロセッサーを必要とする Bean メソッドが稼働する要求プロセッサー・トランザクションに適用します。
4. TRANCLASS 定義で、次の数式を使用して MAXACTIVE の値を設定します。

   ```
   MAXACTIVE <= ((MAXJVMTCBS - n) / (n - 1)) + 1
   ```

   ここで、n は、ユーザーの「ワーストケース」のメソッドが必要とする要求プロセッサーの最大数です。

   この計算の結果が小数値の場合は、一番近い整数に切り下げます。
5. 新規 TRANSACTION および REQUESTMODEL 定義を以下のように作成します。
   a. 複数の要求プロセッサーを必要とする Bean メソッドが稼働する要求プロセッサー・トランザクションに対して、新規に TRANSACTION 定義を作成します。(これを行うために一番簡単な方法は、デフォルトの CIRP 要求プロセ

ッサー・トランザクションの定義をコピーして、そのコピーを変更することです。)TRANCLASS オプションで、新規トランザクション・クラスの名前を指定します。

b. 1 つ以上の REQUESTMODEL 定義を作成します。これらの定義の間では、新規 REQUESTMODEL 定義は、複数の要求プロセッサーを要求する Bean メソッド用に受信するすべての要求をカバーする必要があります。REQUESTMODEL 定義の TRANSID オプションで、新規トランザクション名を指定します。

実動領域でのエンタープライズ Bean の更新

このセクションでは、実動領域でエンタープライズ Bean を更新するための最良の手順を検討します。以下のトピックが含まれます。

- 『問題』
- 359 ページの『可能な解決方法』

問題

実行中の CICS 実動領域でエンタープライズ Bean を更新すると同時に、CICS をリサイクルせずに、現行のワークフローへの中断を最小限にするには、どのようにしますか。

現行のワークフローを中断することなく、実行中の EJB サーバーに新しいエンタープライズ Bean を導入するのは、ごく簡単です。次のいずれかの方法を実行できます。

1. CICS スキャン・メカニズムを使用します。つまり、新しい Bean が入っているデプロイ済み JAR ファイルを、CorbaServer のデプロイ済み JAR ファイル (「ピックアップ」) ディレクトリーに配置し、**PERFORM CORBASERVER SCAN** コマンドを発行します。論理 EJB サーバー内のすべての AOR で繰り返してください。CORBASERVER 定義が AUTOPUBLISH(NO) を指定する場合、いずれかの AOR で、**PERFORM DJAR PUBLISH** コマンドを発行します。

   注: 実動領域でスキャン・メカニズムを使用する場合、セキュリティーの影響に注意してください。特に、DJAR 定義の CICS コマンド・セキュリティーが迂回される可能性です。これを防ぐために、z/OS UNIX のデプロイ済み JAR ファイル・ディレクトリーへの書き込みアクセス権限が付与されるユーザー ID を、DJAR および CORBASERVER 定義の作成と更新のための RACF 権限が付与されたユーザー ID に制限することをお勧めします。

2. **EXEC CICS CREATE DJAR** コマンドを使用して、新しい Bean が入っているデプロイ済み JAR ファイルの定義をインストールします。論理 EJB サーバー内のすべての AOR で繰り返してください。いずれかの AOR で、**PERFORM DJAR PUBLISH** コマンドを発行します。

残念ながら、未完了トランザクションに予測不能な影響があるため、アクティブな EJB サーバーではこれらのメソッドを使用して Bean を*更新* できません。連続するメソッド呼び出しでどのバージョンの Bean (旧または新規) を使用するかを制御する方法はありません(タイミングが異なるため、この問題は、複数領域 EJB サーバーで悪化する可能性が高くなります)。

代替アプローチは、CICS を静止させ、シャットダウンしてから、更新された DJAR 定義を指定して再始動することです。これはテスト環境では受け入れ可能ですが、実動領域には魅力的な解決方法ではありません。359 ページの図 24 を検討してみましょう。 CorbaServer COR2 で bean5 と bean6 を更新することを想像してみてください。CICS をクローズした場合、bean5 と bean6 がシャットダウン時に使用不可になるだけでなく、CorbaServer COR1 内のすべての Bean も使用不可になります。

EJB サーバーに複数の AOR が含まれているときに、複数の AOR 間で要求のバランスを取るためにワークロード管理が使用されている場合は、どうなりますか。パフォーマンスに与える影響を最小限に抑えて、各 AOR を順にシャットダウンし、アップグレードできないですか。残念ながらできません。この理由は次のとおりです。

- アップグレード・プロセス時に、AOR が異なれば、異なるバージョンの Bean があります。新しいバージョンの Bean が旧バージョンとの完全が互換性がなかった場合、これにより、予測不能な影響が生じます。 (新しいバージョンの Bean に旧バージョンとの完全な互換性があるためには、特に、これらの 2 つのバージョンのホーム・インターフェースとコンポーネント・インターフェースが同一でなければなりません。また、任意のステートフル・セッション Bean の状態が保持されなければなりません。)
- 1 つの AOR であってもそれをシャットダウンすると、必然的に EJB サーバーのパフォーマンスがある程度低下します。(アップグレードが重要であれば、これは受け入れ可能な場合があります。パフォーマンスの低下を補うために、おそらく、EJB サーバーに余分な AOR を追加できます。)

このセクションの残りの部分では、実動領域でエンタープライズ Bean を更新するために CICS EJB サーバーで実行する必要があることについて説明します。クライアント・サイドでも変更が必要な可能性があることに注意してください。特に、更新のために、エンタープライズ Bean のホーム・インターフェースまたはコンポーネント・インターフェースが変わる場合、クライアント・アプリケーションが更新された Bean を使用する前に、新しいインターフェースを使用するために再書き込みされなければなりません。

次の図は、クライアントが CorbaServer COR1 および COR2 で Bean メソッドを呼び出している様子を示しています。Bean の保守と可用性の要件に基づいて、CorbaServer 間で Bean を分割できます。

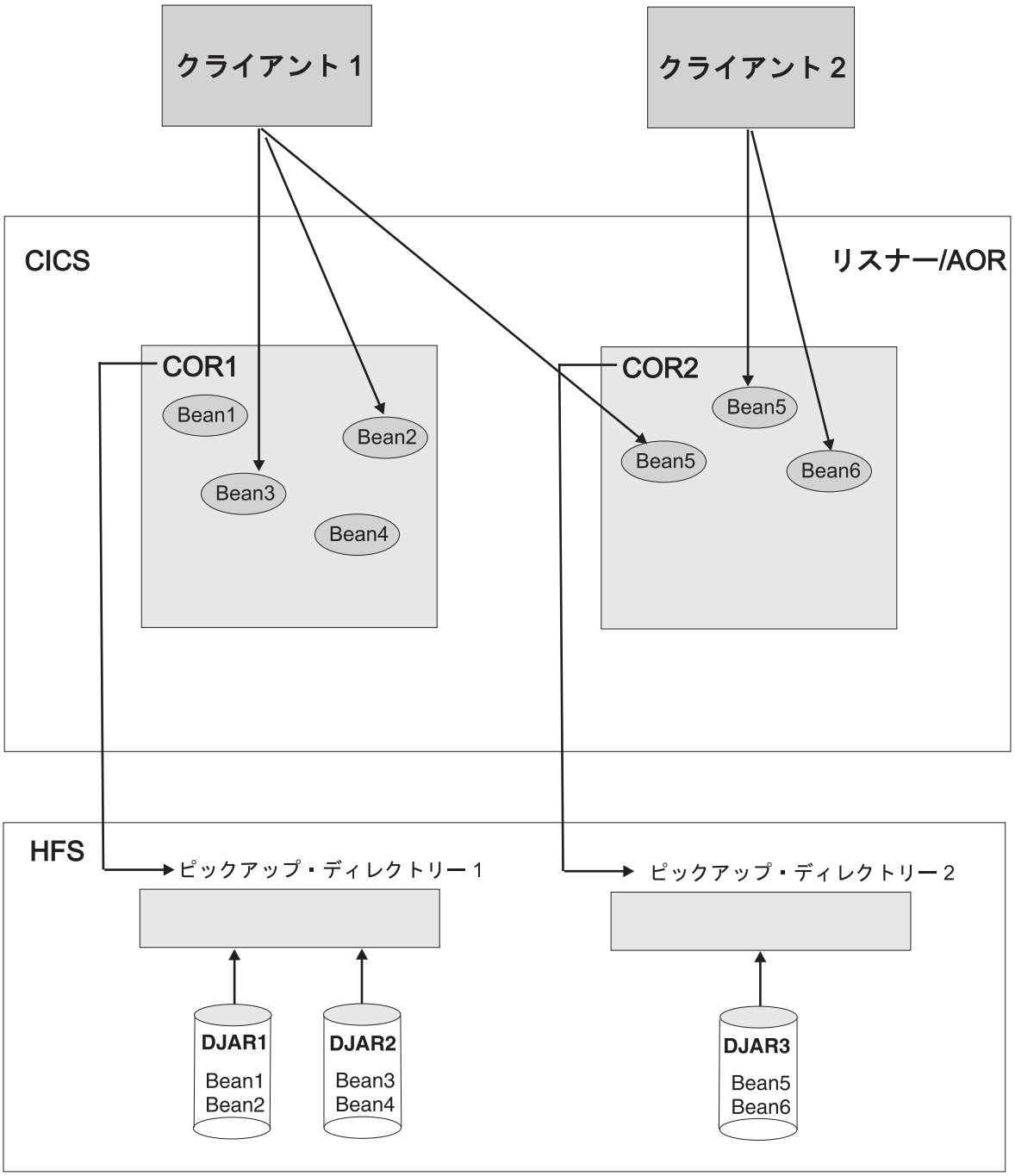


図 24. CICS EJB 実動領域

実動領域で Bean を更新する最適な方法についての問題の推奨解決方法については、『可能な解決方法』を参照してください。

可能な解決方法

実動領域で Bean を更新する最適な方法についての問題の推奨解決方法は、次のとおりです。提示される解決方法は、ご使用の EJB サーバーが単一のリスナー/AOR で構成されているか、複数のリスナーと AOR で構成されているかによって異なります。

一般に、次の場合は、アップグレードの解決方法の方が実装が簡単です。

1. Bean の機能に基づくだけでなく、保守と可用性の要件にも基づいて、CorbaServer 間でエンタープライズ Bean を分割する場合。つまり、異なる保守と可用性の要件を持つ Bean のセットは、異なる CorbaServer にインストールされなければなりません。
2. Bean の機能に基づくだけでなく、保守と可用性の要件にも基づいて、CICS トランザクション ID をエンタープライズ Bean メソッドに割り振る場合。つまり、保守を容易にするために、異なる保守と可用性の要件を持つ Bean のセットは、異なる CICS トランザクション ID で実行されなければなりません。

   重要:

   a. 複数領域 EJB サーバーでは、AOR に複数の CorbaServer が含まれている場合は、各 CorbaServer でサポートされるオブジェクトに異なるトランザクション ID のセットを割り当てるよう強くお勧めします。つまり、AOR 内の各 CorbaServer は異なるトランザクション ID のセットをサポートしていなければなりません。

   b. これにより、分散ルーティング・プログラムが、使用不可になった CorbaServer を迂回すると同時に、他の使用可能な CorbaServer を領域内で使用可能なままにしておくのが簡単になります。分散ルーティング・プログラムをコーディングして、使用不可になっている CorbaServer を処理する方法の詳細については、「*CICS Customization Guide*」を参照してください。

   注: Bean メソッドが実行される CICS トランザクションは、そのメソッドと一致する REQUESTMODEL 定義で指定されます。CREA CICS 提供トランザクションを使用すると、以下を行うことができます。
   - 特定の Bean および Bean メソッドに関連したトランザクション ID を表示します
   - トランザクション ID を変更し、変更を適用し、新しい REQUESTMODEL 定義に対する変更を保管します

単一リスナー/AOR の解決方法:

単一のリスナー/AOR から構成される EJB サーバーには、以下の解決方法が有効です。

359 ページの図 24 で、CorbaServer `COR2` の `bean5` と `bean6` を更新することを前提としましょう。`DJAR3.jar` は、更新される Bean が入っているデプロイ済み JAR ファイルです。以下が必要です。

1. CorbaServer `COR1` とその Bean が、アップグレード・プロセス全体で使用可能なままであること。
2. 可能な場合、CorbaServer `COR2` 内の Bean に対するアップグレードがシームレスであること。つまり、`bean5` または `bean6` の新規インスタンスを作成できない時間があってはなりません (または、少なくとも可能な限り最小の時間でなければなりません)。

解決方法 1:

このタスクについて

この解決方法の利点は、相対的に実装が簡単であることです。欠点は、シームレスでないことです。つまり、bean5 または bean6 の新規インスタンスを作成できない期間があります (その間、旧バージョンの bean5 および bean6 のインスタンスが破棄または不動態化されています)。

1. **EXEC CICS SET CORBASERVER**(COR2) ENABLESTATUS(DISABLED) コマンドまたは **CEMT SET CORBASERVER**(COR2) DISABLED コマンドを発行します。クライアントに Bean のホーム・インターフェースへの参照があるかどうかにかかわらず、bean5 または bean6 の新規インスタンスを作成しようとしても失敗します。

   通常、bean5 および bean6 のインスタンスで現在実行中のメソッドは、完了まで続行します。

   OTS トランザクションに参加していない bean5 または bean6 のインスタンスは、現在実行中のメソッドの終わりに破棄または不動態化されます (現在実行中のメソッドがない場合、すべてのインスタンスが既に破棄または不動態化されています)。

   注: ステートレス・セッション Bean は破棄されます。ステートフル セッション Bean は不動態化されます。

   OTS トランザクションに参加している bean5 または bean6 のインスタンスは、その OTS トランザクションの終わりまで破棄も不動態化もされません。通常、(OTS トランザクションの有効範囲内で) このインスタンスに対する今後のメソッド呼び出しは成功します。OTS トランザクションの終わりに、このインスタンスは破棄または不動態化されます。
2. **EXEC CICS** または **CEMT INQUIRE CORBASERVER**(COR2) ENABLESTATUS コマンドを発行して、bean5 および bean6 のすべてのインスタンスがいつ破棄または不動態化されたかを確認します。DISABLED の状況は、すべての Bean インスタンスが破棄または不動態化されたことを示します。
3. bean5 および bean6 のすべてのインスタンスが破棄または不動態化された場合、CICS スキャン・メカニズムまたは静的 DJAR 定義のどちらかを使用して、DJAR3.jar デプロイ済み JAR ファイルの更新されたバージョンをインストールします (スキャン・メカニズムを使用して静的 DJAR 定義を更新することはできません)。

   次のどちらかです。

   a. DJAR3.jar デプロイ済み JAR ファイルの新規バージョンを CorbaServer COR2 のピックアップ・ディレクトリーに入れます。

   b. **PERFORM CORBASERVER**(COR2) SCAN コマンドを発行します。CICS は COR2 のピックアップ・ディレクトリーをスキャンし、DJAR3.jar の新規定義をインストールし、bean5 および bean6 の新規バージョンを COR2 のシェルフ・ディレクトリーにコピーします。

   または

   a. **EXEC CICS** または **CEMT DISCARD DJAR** (DJAR3) コマンドを発行して、DJAR3.jar の現行定義を CICS から除去します。

b. **CEDA INSTALL DJAR**(DJAR3) または **EXEC CICS CREATE DJAR**(DJAR3) CORBASERVER(COR2) HFSFILE(new_version_of_DJAR3.jar_on_HFS) コマンドを発行します。CICS は、DJAR3.jar の新規定義をインストールし、bean5 および bean6 の新規バージョンを COR2 のシェルフ・ディレクトリーにコピーします。

注:

a. Bean のホーム・インターフェースまたはコンポーネント・インターフェースが前のバージョン以降に変更された場合であっても、bean5 および bean6 の更新されたバージョンをネーム・スペースにリパブリッシュする必要はありません。

b. bean5 または bean6 のホーム・インターフェースまたはコンポーネント・インターフェースが、前のバージョン以降に変更された場合、変更された Bean を使用する前に、新しい署名を使用するようにクライアント・アプリケーションが更新されなければなりません。

c. ステートフル・セッション Bean を更新する場合、正確な変更内容に応じて、直列化された状態の構造を変更できます。これが起きる場合、オブジェクト・ストア内の Bean の不動態化されたインスタンスをすべて無効にします。これが起きる場合、無効にした Bean を使用しようとしても、例外が生じます。この可能性に対処するように、クライアント・アプリケーションをコーディングする必要があります。

4. **CEMT SET CORBASERVER**(COR2) ENABLED コマンドを発行します。*この時点から、すべての新規処理では、更新されたバージョンの bean5 および bean6* が使用されます。

解決方法 2:

この解決方法には、CICSPlex System Manager が必要です。リスナー/AOR 上のすべての CICS アプリケーションは、複数の領域にまたがる複製に適切でなければなりません。

このタスクについて

このソリューションの利点は、解決方法 1 とは異なり、相対的にシームレスであることです。つまり、bean5 または bean6 の新規インスタンスを作成できない期間が、最悪の場合でもごくわずかです。欠点は、解決方法 1 よりも実装が複雑であることです。

1. CICSPlex SM を使用して、以下を行います。

a. 単一のリスナー/AOR を複製します。

b. すべての新規ワークロードをその複製に送信します。つまり、オリジナルの AOR を静止させ、複製をアクティブにします。この方法については、「*CICSPlex System Manager Managing Workloads*」マニュアルの Balancing an enterprise bean workload を参照してください。

COR1 であるか、COR2 であるかにかかわらず、新規 OTS トランザクションで実行されるか、OTS トランザクションなしで実行される Bean メソッドに対するすべての要求は、複製に転送されます。

(COR1 であるか、COR2 であるかにかかわらず) 既存の OTS トランザクションで実行される Bean メソッドに対する要求は、オリジナルの領域に転送されます。

注:

1) 「*新規* OTS トランザクション」とは、Bean の参加が、すべての新規処理が複製に送信された*後で* 開始する OTS トランザクションを意味します。
2) 「*既存の* OTS トランザクション」とは、Bean の参加が、すべての新規処理が複製に送信される*前に* 開始した OTS トランザクションを意味します。

オリジナルの領域では、次のとおりです。

- OTS トランザクションに参加していないエンタープライズ Bean のインスタンスは、現在実行中のメソッドの終わりに破棄または不動態化されます (現在実行中のメソッドがない場合、すべてのインスタンスが既に破棄または不動態化されています)。
- OTS トランザクションに参加している エンタープライズ Bean のインスタンスは、その OTS トランザクションの終わりまで破棄も不動態化もされません。通常、(OTS トランザクションの有効範囲内で) このインスタンスに対する今後のメソッド呼び出しは成功します。OTS トランザクションの終わりに、このインスタンスは破棄または不動態化されます。

2. オリジナルの領域で以下を行います。
   a. `bean1` から `bean6` のすべてのインスタンスがいつ破棄または不動態化されたかを確認します。
      1) `bean1` から `bean6` に関連した CICS トランザクション ID がまだ分からない場合は、CREC トランザクションを使用してこの情報を表示します。
      2) **`INQUIRE TASK`** コマンドを使用して、これらのトランザクションのインスタンスが実行中であるかどうかを確認します。
   b. `bean1` から `bean6` のすべてのインスタンスが破棄または不動態化された場合、CICS スキャン・メカニズムまたは静的 DJAR 定義のどちらかを使用して、`DJAR3.jar` デプロイ済み JAR ファイルの更新されたバージョンをインストールします (スキャン・メカニズムを使用して静的 DJAR 定義を更新することはできません)。

      次のどちらかです。

      1) `DJAR3.jar` デプロイ済み JAR ファイルの新規バージョンを CorbaServer `COR2` のピックアップ・ディレクトリーに入れます。
      2) **`PERFORM CORBASERVER`**(`COR2`) SCAN コマンドを発行します。CICS は `COR2` のピックアップ・ディレクトリーをスキャンし、`DJAR3.jar` の定義を更新し、`bean5` および `bean6` の新規バージョンを `COR2` のシェルフ・ディレクトリーにコピーします。

      または

      1) **`CEMT DISCARD DJAR`**(`DJAR3`) コマンドを発行して、`DJAR3.jar` の旧定義を削除します。

2) **CEDA INSTALL DJAR**(DJAR3) または **EXEC CICS CREATE DJAR**(DJAR3) CORBASERVER(COR2) HFSFILE(new_version_of_DJAR3.jar_on_HFS) コマンドを発行します。CICS は、DJAR3.jar の新規定義をインストールし、bean5 および bean6 の新規バージョンを COR2 のシェルフ・ディレクトリーにコピーします。

注:

1) Bean のホーム・インターフェースまたはコンポーネント・インターフェースが前のバージョン以降に変更された場合であっても、bean5 および bean6 の更新されたバージョンをネーム・スペースにリパブリッシュする必要はありません。
2) bean5 または bean6 のホーム・インターフェースまたはコンポーネント・インターフェースが、前のバージョン以降に変更された場合、変更された Bean を使用する前に、新しい署名を使用するようにクライアント・アプリケーションが更新されなければなりません。
3) ステートフル・セッション Bean を更新する場合、正確な変更内容に応じて、直列化された状態の構造を変更できます。これが起きる場合、オブジェクト・ストア内の Bean の不動態化されたインスタンスをすべて無効にします。これが起きる場合、無効にした Bean を使用しようとしても、例外が生じます。この可能性に対処するように、クライアント・アプリケーションをコーディングする必要があります。

3. CICSPlex SM を使用して、すべての新規ワークロードをオリジナルの領域に送信します。つまり、複製を静止させ、オリジナルの領域をアクティブにします。

   COR1 であるか、COR2 であるかにかかわらず、新規 OTS トランザクションで実行されるか、OTS トランザクションなしで実行される Bean メソッドに対するすべての要求は、オリジナルの領域に転送されます。*この時点から、すべての新規処理では、更新されたバージョンの bean5 および bean6* が使用されます。(COR1 であるか、COR2 であるかにかかわらず) 既存の OTS トランザクションで実行される Bean メソッドに対する要求は、引き続き複製に転送されます。

   注:

   a. 「*新規* OTS トランザクション」とは、Bean の参加が、すべての新規処理がオリジナルの領域にリダイレクトされた*後で* 開始する OTS トランザクションを意味します。
   b. 「*既存の* OTS トランザクション」とは、Bean の参加が、すべての新規処理がオリジナルの領域にリダイレクトされる*前に* 開始した OTS トランザクションを意味します。

   上記で説明したように、最終的に、複製上のエンタープライズ Bean のすべてのインスタンスは、破棄または不動態化されます。

4. 複製領域で、**INQUIRE TASK** コマンドを使用して、bean1 から bean6 のすべてのインスタンスがいつ破棄または不動態化されたかを確認します。これが起きる場合、複製領域を破棄できます。

複数領域 EJB サーバーの解決方法:

1 つ以上のリスナー領域および複数の同一の AOR から構成される EJB サーバーには、以下の解決方法が有効です。

ご使用の EJB サーバーが 3 つの同一のリスナー領域および 5 つの同一の AOR から構成されることを前提とします。各 AOR は、 359 ページの図 24 に示されている領域の複製です (ただし、リスナー/AOR ではなく、AOR であることを除きます)。すべての AOR は同じピックアップ・ディレクトリーを共用し、COR1 および COR2 という名前の同一の CorbaServer に、同じセットのエンタープライズ Bean がそれぞれデプロイされます。

論理 CorbaServer COR2 で bean5 および bean6 を更新するものとします。DJAR3.jar は、更新される Bean が入っているデプロイ済み JAR ファイルです。

以下が必要です。

1. 論理 CorbaServer COR1 とその Bean が、アップグレード・プロセス全体で使用可能なままであること。
2. 可能な場合、論理 CorbaServer COR2 内の Bean に対するアップグレードがシームレスであること。つまり、bean5 または bean6 の新規インスタンスを作成できない時間があってはなりません (または、少なくとも可能な限り最小の時間でなければなりません)。

解決方法 1:

この解決方法は、単一領域用の解決方法 1 を発展させたものです。

このタスクについて

この利点は、相対的に実装が容易であることです。欠点は、シームレスでないことです。つまり、bean5 または bean6 の新規インスタンスを作成できない期間があります (その間、旧バージョンの bean5 および bean6 のインスタンスが破棄または不動態化されています)。

1. 各 AOR で、**EXEC CICS SET CORBASERVER**(COR2) ENABLESTATUS(DISABLED) または **CEMT SET CORBASERVER(**COR2) DISABLED コマンドを発行します。すべての AOR で、次のとおりです。
   - クライアントに Bean のホーム・インターフェースへの参照があるかどうかにかかわらず、bean5 または bean6 の新規インスタンスを作成しようとしても失敗します。
   - 通常、bean5 および bean6 のインスタンスで現在実行中のメソッドは、完了まで続行します。
   - OTS トランザクションに参加していない bean5 または bean6 のインスタンスは、現在実行中のメソッドの終わりに破棄または不動態化されます (現在実行中のメソッドがない場合、すべてのインスタンスが既に破棄または不動態化されています)。
   - OTS トランザクションに参加している bean5 または bean6 のインスタンスは、その OTS トランザクションの終わりまで破棄も不動態化もされません。通常、(OTS トランザクションの有効範囲内で) このインスタンスに対する今後のメソッド呼び出しは成功します。OTS トランザクションの終わりに、このインスタンスは破棄または不動態化されます。
2. 各 AOR で、**EXEC CICS** または **CEMT INQUIRE CORBASERVER**(COR2) ENABLESTATUS コマンドを発行して、bean5 および bean6 のすべてのインス

タンスがいつ破棄または不動態化されたかを確認します。DISABLED の状況は、すべての Bean インスタンスが破棄または不動態化されたことを示します。

3. すべての AOR 上の bean5 および bean6 のすべてのインスタンスが破棄または不動態化された場合、CICS スキャン・メカニズムまたは静的 DJAR 定義のどちらかを使用して、DJAR3.jar デプロイ済み JAR ファイルの更新されたバージョンをインストールします (スキャン・メカニズムを使用して静的 DJAR 定義を更新することはできません)。

   次のどちらかです。

   a. DJAR3.jar デプロイ済み JAR ファイルの新規バージョンを CorbaServer COR2 のピックアップ・ディレクトリー (すべての AOR で共用されます) に入れます。

   b. 各 AOR で、**PERFORM CORBASERVER**(COR2) SCAN コマンドを発行します。AOR は COR2 のピックアップ・ディレクトリーをスキャンし、DJAR3.jar の新規定義をインストールし、bean5 および bean6 の新規バージョンを COR2 のシェルフ・ディレクトリーにコピーします。

   または、各 AOR で以下を行います。

   a. **EXEC CICS** または **CEMT DISCARD DJAR** (DJAR3) コマンドを発行して、DJAR3.jar の現行定義を CICS から除去します。

   b. **CEDA INSTALL DJAR**(DJAR3) または **EXEC CICS CREATE DJAR**(DJAR3) CORBASERVER(COR2) HFSFILE(new_version_of_DJAR3.jar_on_HFS) コマンドを発行します。CICS は、DJAR3.jar の新規定義をインストールし、bean5 および bean6 の新規バージョンを COR2 のシェルフ・ディレクトリーにコピーします。

   注:

   a. Bean のホーム・インターフェースまたはコンポーネント・インターフェースが前のバージョン以降に変更された場合であっても、bean5 および bean6 の更新されたバージョンをネーム・スペースにリパブリッシュする必要はありません。

   b. bean5 または bean6 のホーム・インターフェースまたはコンポーネント・インターフェースが、前のバージョン以降に変更された場合、変更された Bean を使用する前に、新しい署名を使用するようにクライアント・アプリケーションが更新されなければなりません。

   c. ステートフル・セッション Bean を更新する場合、正確な変更内容に応じて、直列化された状態の構造を変更できます。これが起きる場合、オブジェクト・ストア内の Bean の不動態化されたインスタンスをすべて無効にします。これが起きる場合、無効にした Bean を使用しようとしても、例外が生じます。この可能性に対処するように、クライアント・アプリケーションをコーディングする必要があります。

4. 各 AOR で、**CEMT SET CORBASERVER**(COR2) ENABLED コマンドを発行します。*この時点から、すべての新規処理では、更新されたバージョンの bean5 および bean6 が使用されます。*

解決方法 2:

このタスクについて

この解決方法には、CICSPlex System Manager が必要です。これは、単一領域用の解決方法 2 を発展させたものです。その利点は、相対的にシームレスであることです。つまり、bean5 または bean6 の新規インスタンスを作成できない期間が、最悪の場合でもごくわずかです。その欠点は、解決方法 1 よりも実装が複雑であることです。

1. CICSPlex SM を使用して、以下を行います。
   a. すべての AOR で複製を作成します。
   b. すべての新規ワークロードを複製に送信します。つまり、オリジナル AOR を静止させ、複製をアクティブにします。この方法については、「*CICSPlex System Manager Managing Workloads*」マニュアルの Balancing an enterprise bean workload を参照してください。

      COR1 であるか、COR2 であるかにかかわらず、新規 OTS トランザクションで実行されるか、OTS トランザクションなしで実行される Bean メソッドに対する各要求は、複製のどちらかに転送されます。

      (COR1 であるか、COR2 であるかにかかわらず) 既存の OTS トランザクションで実行される Bean メソッドに対する各要求は、該当するオリジナル AOR に転送されます。

      注:

      1) 「*新規* OTS トランザクション」とは、Bean の参加が、すべての新規処理が複製に送信された*後で* 開始する OTS トランザクションを意味します。
      2) 「*既存の* OTS トランザクション」とは、Bean の参加が、すべての新規処理が複製に送信される*前に* 開始した OTS トランザクションを意味します。
      3) 「*該当する* オリジナル AOR」とは、OTS トランザクション用の要求プロセッサーが入っているオリジナルの AOR を意味します。
2. 各オリジナル AOR で以下を行います。

   bean1 から bean6 のすべてのインスタンスがいつ破棄または不動態化されたかを確認します。

   a. bean1 から bean6 に関連した CICS トランザクション ID がまだ分からない場合は、CREC トランザクションを使用してこの情報を表示します。
   b. **INQUIRE TASK** コマンドを使用して、これらのトランザクションのインスタンスが実行中であるかどうかを確認します。
3. すべてのオリジナル AOR 上の bean1 から bean6 のすべてのインスタンスが破棄または不動態化された場合、CICS スキャン・メカニズムまたは静的 DJAR 定義のどちらかを使用して、DJAR3.jar デプロイ済み JAR ファイルの更新されたバージョンをインストールします (スキャン・メカニズムを使用して静的 DJAR 定義を更新することはできません)。

   次のどちらかです。

a. DJAR3.jar デプロイ済み JAR ファイルの新規バージョンを COR2 のピックアップ・ディレクトリー (すべてのオリジナル AOR で共用されます) に入れます。

b. 各オリジナル AOR で、**PERFORM CORBASERVER**(COR2) SCAN コマンドを発行します。AOR は COR2 のピックアップ・ディレクトリーをスキャンし、DJAR3.jar の定義を更新し、bean5 および bean6 の新規バージョンを COR2 のシェルフ・ディレクトリーにコピーします。

または

a. 各オリジナル AOR で、**CEMT DISCARD DJAR**(DJAR3) コマンドを発行して、DJAR3.jar の旧定義を削除します。

b. 各オリジナル AOR で、**CEDA INSTALL DJAR**(DJAR3) または **EXEC CICS CREATE DJAR**(DJAR3) CORBASERVER(COR2) HFSFILE (new_version_of_DJAR3.jar_on_HFS) コマンドを発行します。CICS は、DJAR3.jar の新規定義をインストールし、bean5 および bean6 の新規バージョンを COR2 のシェルフ・ディレクトリーにコピーします。

注:

a. Bean のホーム・インターフェースまたはコンポーネント・インターフェースが前のバージョン以降に変更された場合であっても、bean5 および bean6 の更新されたバージョンをネーム・スペースにリパブリッシュする必要はありません。

b. bean5 または bean6 のホーム・インターフェースまたはコンポーネント・インターフェースが、前のバージョン以降に変更された場合、変更された Bean を使用する前に、新しい署名を使用するようにクライアント・アプリケーションが更新されなければなりません。

c. ステートフル・セッション Bean を更新する場合、正確な変更内容に応じて、直列化された状態の構造を変更できます。これが起きる場合、オブジェクト・ストア内の Bean の不動態化されたインスタンスをすべて無効にします。これが起きる場合、無効にした Bean を使用しようとしても、例外が生じます。この可能性に対処するように、クライアント・アプリケーションをコーディングする必要があります。

4. CICSPlex SM を使用して、すべての新規ワークロードをオリジナル AOR に送信します。つまり、複製を静止させ、オリジナル AOR をアクティブにします。

COR1 であるか、COR2 であるかにかかわらず、新規 OTS トランザクションで実行されるか、OTS トランザクションなしで実行される Bean メソッドに対するすべての要求は、オリジナル AOR に転送されます。*この時点から、すべての新規処理では、更新されたバージョンの bean5 および bean6 が使用されます。*(COR1 であるか、COR2 であるかにかかわらず) 既存の OTS トランザクションで実行される Bean メソッドに対する要求は、引き続き複製に転送されます。

注:

a. 「*新規* OTS トランザクション」とは、Bean の参加が、すべての新規処理がオリジナル AOR にリダイレクトされた後で 開始する OTS トランザクションを意味します。

b. 「*既存の* OTS トランザクション」とは、Bean の参加が、すべての新規処理がオリジナル AOR にリダイレクトされる*前に* 開始した OTS トランザクションを意味します。

最終的に、複製上のエンタープライズ Bean のすべてのインスタンスは、破棄または不動態化されます。

5. 各複製で、**INQUIRE TASK** コマンドを使用して、bean1 から bean6 のすべてのインスタンスがいつ破棄または不動態化されたかを確認します。これが起きる場合、複製を破棄できます。

その他の可能な解決方法: 360 ページの『単一リスナー/AOR の解決方法』および 364 ページの『複数領域 EJB サーバーの解決方法』で説明された解決方法のみが可能であるわけではありません。例えば、別のアプローチは次のとおりです。

1. 更新される Bean に関連した要求プロセッサーに非デフォルト TRANID を使用します (つまり、上記で推奨された方法で、CorbaServer とトランザクション ID によってエンタープライズ Bean を分けます)。
2. 要求プロセッサーのトランザクションを使用不可にするか、トランザクションをトランザクション・クラスに入れ、TCLASS 限界をゼロに減らします。
3. Bean のすべてのインスタンスが破棄または不動態化された場合、その他の解決方法で説明された方法のいずれかで、デプロイ済み JAR ファイルの更新されたバージョンをインストールします。

CCI Connector for CICS TS

CCI Connector for CICS TS は、既存の CICS プログラムを利用する Enterprise JavaBean (EJB) サーバー・コンポーネントの作成に役立ちます。

CCI Connector for CICS TS の概要

CCI Connector for CICS TS は、既存の CICS プログラムを利用する Enterprise JavaBean (EJB) サーバー・コンポーネントの作成に役立ちます。

背景 — コネクター:

多くの場合、既存の (非 Java) CICS プログラムの能力を活かすと、新しい Java アプリケーションを迅速かつ確実に開発できます。

CICS コネクターは、Java クライアント・アプリケーションが CICS アプリケーションを起動できるようにするソフトウェア・コンポーネントです。通常、CICS コネクターを使用する Java クライアント・プログラムは、サーブレットです。

複数のリリースで、CICS は、*CICS の外部で実行される* (例えば、Windows、UNIX、またはネイティブ z/OS で実行される) Java クライアント・プログラムが、CICS サーバー上の指定されたプログラムに接続できるようにする CICS コネクターをサポートしてきました。CCI Connector for CICS TS は、*CICS Transaction Server for z/OS で実行される* Java プログラムまたはエンタープライズ Bean が、CICS サーバー・プログラムにリンクできるようにします。

CCI Connector for CICS TS は、J2EE Connector Architecture Specification、バージョン 1.0 で定義される業界標準の **Common Client Interface (CCI)** を実装します。

注: CICS TS for z/OS バージョン 2.1 で導入された CICS Connector for CICS TS は、サポートされなくなりました。CCI Connector for CICS TS とは異なり、CICS Connector for CICS TS は、非標準の IBM 専有クライアント・インターフェースを実装しました。CICS Connector for CICS TS を使用する既存のアプリケーションを、代わりに CCI Connector for CICS TS を使用するようにアップグレードする場合の助言については、386 ページの『CICS Connector for CICS TS から CCI Connector for CICS TS へのアップグレード』を参照してください。

Common Client Interface:

このセクションでは、Common Client Interface (CCI) の概要を説明します。Common Client Interface は、J2EE コネクター・アーキテクチャーの一部です。

このインターフェースの確実な情報については、J2EE Connector Architecture Specification を参照してください。この仕様をダウンロードするには、Oracle Technology Network Java Web サイトに進み、「*J2EE Connector architecture*」を検索してください。

CCI が提供する標準インターフェースは、開発者が、汎用プログラミング・スタイルを使用して、特定のリソース・アダプターを通じて任意の数のエンタープライズ情報システム (EIS) と通信できるようにします。CCI は、Java Database Connectivity (JDBC) で使用されるクライアント・インターフェースに基づいて厳密にモデル化され、*Connection* や *Interaction* の使用が似ています。

CCI 内には、2 つの特殊なタイプのクラスがあります。便宜のためにフレームワーク・クラスと*入出力* クラスと呼びます。

フレームワーク・クラス:

フレームワーク・クラスは、CICS などの EIS との接続を要求し、EIS でコマンドを実行して、入力を渡し、出力を取り出すのに使用されます。

フレームワーク・クラスは次のとおりです。

ConnectionFactory
ConnectionFactory オブジェクトは、Java コンポーネントが特定の EIS との通信に使用できる接続の作成に使用されます。ConnectionFactory の属性は、接続を作成する対象の EIS を指定します。ConnectionFactory は、Connection オブジェクト用のファクトリーです。

Connection
Connection オブジェクトは、特定サーバーとの固有の接続を識別します。Interaction オブジェクト用のファクトリーです。

Interaction
Interaction オブジェクトの execute メソッドを使用すると、サーバーとの対話を進めることができます。CICS TS では、execute メソッドは 3 つの引数を取ります。すなわち、対話のタイプを指定する 1 つの InteractionSpec オブジェクトと、入力データと出力データを伝送する 2 つの Record オブジェクトです。

J2EE コンポーネントは、フレームワーク・クラスを使用して、EIS との接続を獲得し、データを送信および受信します。最初に、J2EE コンポーネントは、アクセスで

きる特定の EIS (例えば、CICS) の ConnectionFactory オブジェクトを取得します (このコンポーネントは、ConnectionFactory をプログラマチックに作成するか、さらに可能性が高いのは、JNDI ネーム・スペース内でそれを検索することができます)。ConnectionFactory を使用して、Connection オブジェクトを取得します。次に、**Connection** オブジェクトを使用して、1 つ以上の Interaction オブジェクトを作成します。これらの Interaction オブジェクトを通じて EIS でコマンドを実行します。

図 25 は、EIS に接続し、コマンドを実行するのに使用される CCI フレームワーク・クラスを示しています。

```
ConnectionFactory cf = <Lookup from JNDI namespace>
Connection conn = cf.getConnection();
Interaction int = conn.createInteraction();
int.execute(<Input output data>);
int.close();
conn.close();
```

図 25. CCI フレームワーク・クラスを使用した EIS への接続とコマンドの実行

入出力クラス:

フレームワーク・クラスを使用すると、J2EE リソース・アダプターを使用して EIS にアクセスする一般的な方法が得られます。

しかし、各 EIS には異なる入出力のニーズがあるため、CCI インターフェースは、J2EE コンポーネントが EIS 固有の情報を J2EE リソース・アダプターに渡す方法を備えています。この目的に、J2EE コンポーネントは次のタイプのオブジェクトを使用します。

- ConnectionSpec オブジェクト
- InteractionSpec オブジェクト
- Record オブジェクト

ConnectionSpec

ConnectionSpec オブジェクトは、サーバーとの対話で使用されるセキュリティー属性 (ユーザー ID やパスワードなど) の指定に使用できます。

注: CICS は、ConnectionSpec オブジェクトで指定されるセキュリティー設定を無視します。コネクターに適切なセキュリティー・コンテキストを既に確立しているからです。

CCI Connector for CICS TS の ConnectionSpec クラスは、ECIConnectionSpec と呼ばれます。

InteractionSpec

InteractionSpec オブジェクトは、サーバーとの対話に必要な必須の属性 (例えば、ターゲット・プログラムの名前) を保持します。これは、特定の対話が実行されるときに **Interaction.execute()** メソッド呼び出しで必須引数として渡されます。

CCI Connector for CICS TS の InteractionSpec クラスは、ECIInteractionSpec と呼ばれます。

Record

Record オブジェクトは、ターゲット・プログラムと交換されるデータを保

持する Bean です。CICS 通信域 (COMMAREA) と等価なものと見なすことができます。データは、Record で定義されたインターフェースを使用してアクセス可能です。

図 26 は、EIS に接続し、EIS 固有の入出力パラメーターを渡し、コマンドを実行するために一緒に使用される CCI フレームワーク・クラスと入出力クラスを示しています。

```
ConnectionFactory cf = <Lookup from JNDI namespace>
ECIConnectionSpec cs = new ECIConnectionSpec();
cs.setXXX();     //Set any connection specific properties

Connection conn = cf.getConnection(cs);
Interaction int = conn.createInteraction();
ECIInteractionSpec is = new ECIInteractionSpec();
is.setXXX();    //Set any interaction specific properties

RecordImpl in = new RecordImpl();
RecordImpl out = new RecordImpl();
int.execute(is,in,out);
int.close();
conn.close();
```

図 26. EIS との完全な CCI 対話

CCI Connector for CICS TS:

CICS Transaction Gateway には、CICS 用の外部呼び出しインターフェース (ECI) リソース・アダプターが含まれています。

ECI リソース・アダプターが提供する標準 CCI インターフェースは、J2EE コンポーネントが、サーバーとの間で両方向に情報を渡すためのデータ域 (COMMAREA) を使用して、CICS サーバー・プログラムを呼び出すことができるようにします。通常、これらの J2EE コンポーネントはサーブレットまたはエンタープライズ Bean です。いかなる場合でも、CICS の外部で実行されます。

CICS TS には、CCI Connector for CICS TS が含まれています。これが提供する標準 CCI インターフェースは、*CICS* 内で 実行される Java プログラムとコンポーネント (例えば、エンタープライズ Bean) が CICS サーバー・プログラムを呼び出すことができるようにします。

CICS TS で実行される Java プログラムまたはエンタープライズ Bean は、 CCI Connector for CICS TS を使用して、適切な CICS サーバー・プログラムにリンクできます。CICS サーバー・プログラムは、以下のとおりです。

- CICS でサポートされる任意の言語で作成できます。
- 適切な通信域 (COMMAREA) を使用しなければなりません。
- 端末入出力を行ってはなりません。
- 通常、別個のバックエンド CICS Transaction Server for z/OS 領域で実行されますが、オプションとして、Java プログラムまたは Bean と同じ CICS 領域で実行することもできます。

コネクターは JCICS `Program.link()` 呼び出しを使用して、バックエンド・サーバー・プログラムにアクセスします。リンクおよび分散プログラム・リンク (DPL) 呼び出しがサポートされます。このシナリオが 373 ページの図 27 に示されていま

す。この例では、Java クライアント・アプリケーションまたはサーブレットは RMI-IIOP を使用して、CICS EJB サーバーでエンタープライズ Bean のインスタンスを作成します。このエンタープライズ Bean は、CCI Connector for CICS TS を使用して、バックエンド CICS Transaction Server for z/OS 領域のサーバー・プログラムにリンクします。

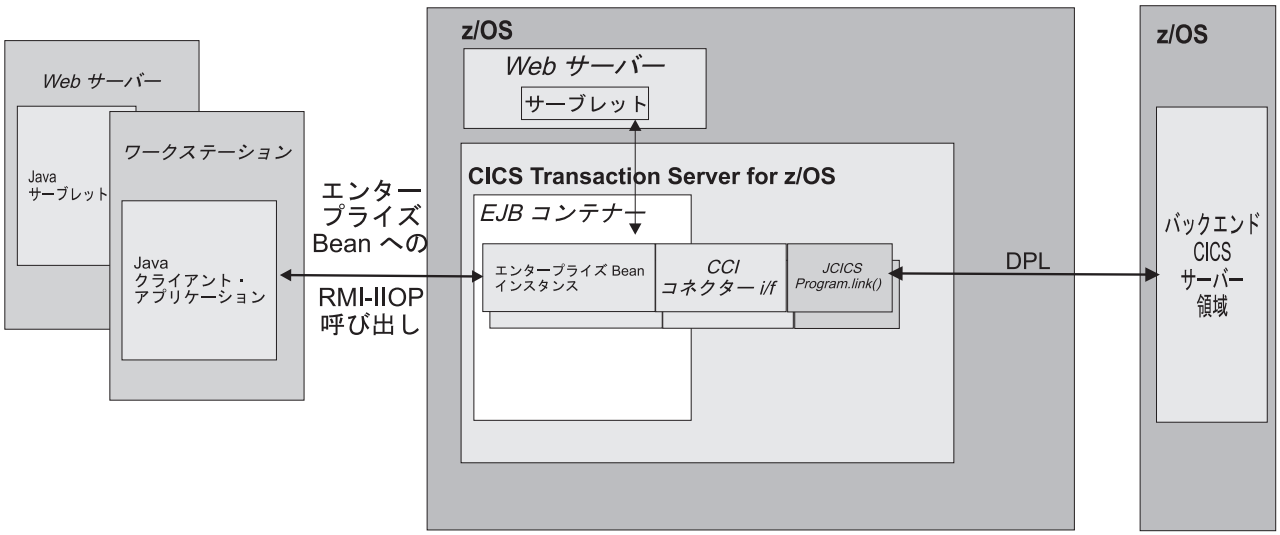


図 27. CICS エンタープライズ Bean は CCI Connector for CICS TS を使用して、CICS サーバー・プログラムに接続します :

Java クライアント・アプリケーションまたはサーブレットは RMI-IIOP を使用して、CICS EJB コンテナーに存在するエンタープライズ Bean のインスタンスを作成します。このエンタープライズ Bean は、CCI Connector for CICS TS を使用して、バックエンド CICS TS for z/OS 領域のサーバー・プログラムにリンクします。

CCI Connector for CICS TS を使用するエンタープライズ Bean を作成するために、Java プログラマーには CICS の妥当な知識が必要です (ただし、JCICS を使用する場合よりも必要性はいくらか少ない)。しかし、作成されるエンタープライズ Bean は、CICS の知識がほとんどない Java プログラマーが使用できます。

CCI Connector for CICS TS は、CICS 内で実行するために高度に最適化されます。JCICS `Program.link()` 呼び出しよりもむしろ、これを使用すると、オーバーヘッドがほとんど生じません。

CCI Connector for CICS TS の利点:

CCI Connector for CICS TS を使用して、既存の CICS プログラムを利用する強力なサーバー・コンポーネントを作成するには、複数の利点があります。

1. このコネクターを使用する CICS エンタープライズ Bean は、次のとおりです。
   - 通常、CICS の知識がほとんどない、Java クライアント・アプリケーションのプログラマーが、CICS の機能をアプリケーションに追加できるようにします。
   - 複数のプラットフォームで実行中の Java クライアント・アプリケーションおよびサーブレットによって呼び出すことができます。Bean (および Bean を通じて CICS サーバー・プログラム) を呼び出すのに使用されるクライアント・

コードは、すべての Java プラットフォームで同一です。したがって、例えば、クライアントは、WebSphere で実行されるエンタープライズ Bean であったり、Web サーバーで実行されるサーブレットであったり、ワークステーションのスタンドアロン・アプリケーションであったりすることができます。

- 正しく作成される場合、Common Client Interface をサポートするすべての EJB サーバー間で、ほとんど変更なしに移植可能でなければなりません。

2. Common Client Interface は非専有標準であるので、サーバー・プログラムを呼び出す CCI コードは、ほとんど変更なく、大部分の Java 対応プラットフォームとの間で双方向に移植可能でなければなりません。
3. CCI Connector for CICS TS は CICS 内で実行されるので、コネクターと CICS との間にはネットワーク・フローは必要ありません。したがって、このコネクターのパフォーマンスは、ECI リソース・アダプターを使用して CICS 外から CICS プログラムにアクセスする CCI コネクターのパフォーマンスよりもすぐれています。
4. CICS セッション Bean からこのコネクターを使用すると、シンプルな 2 層デプロイメント・モデル (クライアント → CICS TS) になります。
5. ECI リソース・アダプターを使用するように作成されたプログラムは、CCI Connector for CICS TS を使用するように容易に適応できます。したがって、以前に CICS の外部から CICS サーバー・プログラムにアクセスしたクライアント・プログラムは、CICS 内で実行するためにマイグレーションできます。

   注: ECI リソース・アダプターを使用するように作成されたプログラムを、CCI Connector for CICS TS を使用するように移植すると、CICS Transaction Gateway クラスではなく、`dfjcci.jar` JAR ファイルで CICS TS 提供のクラスを使用するように、そのプログラムを再コンパイルする必要があります。
6. CCI Connector for CICS TS は、Java 2 セキュリティー・ポリシー・メカニズムをサポートします。

サンプル・アプリケーション:

CICS が提供する 2 つのサンプル・アプリケーションは、CICS Java プログラムまたはエンタープライズ Bean が、CCI Connector for CICS TS を使用して CICS サーバー・プログラムを呼び出す方法を示しています。

1. CCI Connector サンプル。これは、CCI API を直接コーディングする方法を示す、相対的にシンプルなアプリケーションです。

   CCI Connector サンプルは、次の方法を示します。
   a. 以前に公開された接続ファクトリーを JNDI ネーム・スペースで検索する方法
   b. CCI Connector for CICS TS を使用して、CICS サーバー・プログラムを呼び出す方法

   CCI Connector サンプルについては、383 ページの『CCI Connector サンプル・アプリケーション』で説明しています。
2. EJB Bank Account サンプル。これはより複雑なサンプルであり、エンタープライズ Bean および DB2 を使用して、CICS 制御情報を Web ユーザーが利用できるようにする方法を示します。このサンプルは、 CCI Connector for CICS TS

を使用してバックエンド CICS COBOL プログラムにリンクする CICS エンタープライズ Bean を実装します。COBOL プログラムは、DB2 データ表から情報を取り出します。

EJB Bank Account サンプルについては、 316 ページの『EJB Bank Account サンプル・アプリケーション』で説明しています。

また、CICS は、次の方法を示す 2 つのサンプル・ユーティリティー・プログラムも提供します。

1. JNDI ネームスペースへの接続ファクトリーの公開 (`CICSConnectionFactoryPublish` サンプル)。これについては、 381 ページの『CICSConnectionFactoryPublish を使用した接続ファクトリーの公開』 で説明しています。
2. 前に公開された接続ファクトリーの JNDI ネームスペースからの撤回 (`CICSConnectionFactoryRetract` サンプル)。これについては、 382 ページの『CICSConnectionFactoryRetract を使用した接続ファクトリーの撤回』 で説明しています。

CCI Connector for CICS TS の使用

CCI Connector for CICS TS を使用する CICS Java コンポーネントは、次の 2 とおりの方法でプログラミングできます。

このタスクについて

1. コネクターによる Common Client Interface の実装に直接プログラミングします。このアプローチで最良のパフォーマンスが生じます。
2. コネクターの Common Client Interface をプログラミングするためにビジュアル・インターフェースとハイレベルの構成体を提供する rapid application development (RAD) ツールを使用します。

どちらの方法を選択しても、CICS TS で実行中の Java コンポーネントから CCI Connector for CICS TS を使用する方法を理解する必要があります。

CICS エンタープライズ Bean がバックエンド CICS プログラムへのリンクに使用する必要があるロジックが、 372 ページの図 26 に示されています。つまり、

1. CICS 提供のサンプル・プログラム `CICSConnectionFactoryPublish` を使用して、CCI Connector for CICS TS で使用するのに適した ConnectionFactory オブジェクトを、ローカル CICS 領域で使用される JNDI ネーム・スペースに公開します。(379 ページの『接続ファクトリーの管理および獲得のためのサンプル・ユーティリティー・プログラムの使用』を参照。)
2. ConnectionFactory オブジェクトを宣言し、それを JNDI 検索を使用して CICS 接続ファクトリーに設定します。
3. ECIConnectionSpec オブジェクトを作成します。必要に応じてそのプロパティーを設定します。

   注: このステップは、完全性を確保するために含まれています。ただし、ECIConnectionSpec オブジェクトで指定されるユーザー ID またはパスワードはすべて、CICS によって無視されます。

4. ConnectionFactory を使用して、Connection オブジェクトを作成します。このオブジェクトは、CICS との単一接続を表します。
5. Interaction オブジェクトを Connection オブジェクトから作成します。
6. ECIInteractionSpec オブジェクトを作成します。ターゲット・プログラムの名前や対話のモード (同期または非同期) を始めとする、そのプロパティーを設定します (CICS TS の場合、同期モードのみがサポートされます)。
7. ターゲット・プログラムの入力通信域と出力通信域を表す、2 つの Record オブジェクトを作成します。
8. Interaction オブジェクトの execute メソッドを実行して、ECIInteractionSpec、および入力と出力の Record オブジェクトを引数として渡します。
9. ターゲット・プログラムによって戻されるデータを出力 Record オブジェクトから取り出します。
10. Interaction オブジェクトの close メソッドを実行します。
11. Connection オブジェクトの close メソッドを実行します。

注: リンクされるプログラムを所有する CICS サーバー領域を指定するには、サーバー・プログラムのローカル PROGRAM リソースを使用してください。サーバー・プログラムの場所 (ローカルまたはリモート)、およびリモートである場合は、動的ルーティングが行われるかどうかを指定してください。

重要: CCI Connector アーキテクチャー API の Javadoc を使用して、CCI アプリケーションのコーディングに役立てください。これは、CCI 実装で使用される例外などの情報も提供します。 CICS 固有の ECIConnectionSpec および ECIInteractionSpec クラス用の Javadoc は、CICS インフォメーション・センターの「*CCI Connector for CICS TS: Class Reference*」にあります。

使用するクラス:

`javax.resource.cci` パッケージ内の標準 CCI クラスか、`com.ibm.connector2.cics` パッケージ内の CCI Connector for CICS TS で提供される CICS 固有のクラスのどちらを使用しますか。

フレームワーク・クラス:

CCI Connector for CICS TS は、**ECIConnectionFactory**、**ECIConnection**、および **ECIInteraction** と呼ばれるフレームワーク・クラスの実装を提供します。

ただし、CICS 固有の実装ではなく、標準の **ConnectionFactory**、**Connection**、および **Interaction** クラスを使用する必要があります。これらのクラスのプログラミングについては、「*CICS Transaction Gateway: Programming Guide*」を参照してください。参照情報については、**ConnectionFactory**、**Connection**、および **Interaction** クラスのソース・コードから生成される Sun Javadoc を参照してください。

「*CICS Transaction Gateway: Programming Guide*」のすべての情報が、CCI Connector for CICS TS に適用されるとは限らないことに注意してください。**ConnectionFactory** クラス (および CICS 提供の **ECIManagedConnectionFactory** クラス) の次のプロパティーは、CICS TS によって無視されます。

- clientSecurity
- connectionURL (CICS TS では、これは常に `local:` です)

- password
- portNumber
- serverName
- serverSecurity
- userName

上記のいずれかのプロパティーに値を指定しても無効です。

入出力クラス:

CCI Connector for CICS TS は、入出力クラスの実装を提供します。標準の ConnectionSpec および InteractionSpec クラスではなく、これらの CICS 固有のクラス (ECIConnectionSpec および ECIInteractionSpec) を使用してください。

CICS 固有のクラスのプログラミングについては、「*CICS Transaction Gateway: Programming Guide*」を参照してください。参照情報については、「*CCI Connector for CICS TS: Class Reference*」内の ECIConnectionSpec および ECIInteractionSpec クラスから生成された CICS Javadoc を参照してください。CCI Connector for CICS TS に適用される特別な考慮事項を、以下にリストしています。

注: 「CICS TS でサポートされない」と記述されているプロパティーまたは値を指定すると、例外が生じます。「CICS TS で無視される」と記述されているプロパティーまたは値を指定しても、無効です。

ECIConnectionSpec
このクラスは、J2EE コンポーネントが、接続ファクトリーに定義されたものとは異なるセキュリティー資格情報を渡すことができるようにします。プロパティーには以下のものがあります。

Password
UserName で指定されたユーザー ID のパスワード。CICS TS で無視されます。

UserName
CICS へのアクセスに使用されるユーザー ID。CICS TS で無視されます。

ECIInteractionSpec
このクラスは、CICS との対話に必要な対話関連の属性 (例: ターゲット・プログラムの名前や対話のモード (同期または非同期)) をすべて保持します。各 Interaction.execute() メソッド呼び出しの必須パラメーターです。そのプロパティーは次のとおりです。

InteractionVerb
CICS への呼び出しのモード (同期または非同期)。CCI Connector for CICS TS は、次のもののみをサポートします。

SYNC_SEND_RECEIVE
同期呼び出し。これは、CICS プログラムへのリンクに使用されます。

FunctionName
CICS で実行されるプログラムの名前。CCI Connector for CICS TS では、FunctionName を指定する必要があります。

注: FunctionName は、ローカル・プログラムまたはリモート・プログラムのどちらでも参照できます。ローカル領域の PROGRAM 定義は、サーバー・プログラムの場所 (ローカルまたはリモート)、およびリモートである場合は、動的ルーティングが行われるかどうかを指定します。

ExecuteTimeout
CICS との対話のタイムアウト値。

0 タイムアウトなし。これがデフォルト値であり、CICS TS でサポートされる唯一の値です。

正整数 ミリ秒単位の時間の長さ。CICS TS で無視されます。

CommareaLength
入力レコード内で CICS に渡される通信域 (COMMAREA) の長さ。これが指定されない場合、CCI Connector for CICS TS で使用されるデフォルトは、入力レコード・データの長さです。

ReplyLength
CICS から戻すデータの量。戻される大きい COMMAREA の少量のみがエンタープライズ Bean または Java コンポーネントで必要な場合、この設定を使用してネットワーク帯域幅を減らすことができます。指定されない場合、デフォルトでは、COMMAREA 内のすべてのデータを受け取ります。

注: ReplyLength を設定しないようにお勧めします。CCI Connector for CICS TS は常にローカル・モードで実行される (つまり、このコネクターを呼び出すエンタープライズ Bean または Java コンポーネントは、コネクター自体と同じ CICS 領域で実行される) ので、考慮が必要なネットワーク・フローがなく、したがって応答全体より少なく受け取る必要はありません。

Record
入力と出力の場合、CCI Connector for CICS TS は、javax.resource.cci.Streamable インターフェースを実装する Record クラスのみをサポートします。これにより、このコネクターは、CICS COMMAREA を構成するバイトのストリームを、ECIInteraction の **execute()** メソッドに提供される Record オブジェクトとの間で両方向に直接読み書きすることができます。

javax.resource.cci.Streamable インターフェースを使用して、入力レコードを作成し、出力レコードからバイト配列を取り出す方法について詳しくは、「*CICS Transaction Gateway: Programming Guide*」を参照してください。

データ変換と CCI Connector for CICS TS

テキスト・データを表すために、Java プログラムは常に Unicode 文字セットを使用しますが、CICS TS プログラムは EBCDIC を使用します。

Java プログラムまたはエンタープライズ Bean が CICS TS サーバー・プログラムを呼び出す場合、サーバー・プログラムの通信領域にある任意のテキスト値は、入力時に Unicode から EBCDIC に、出力時に EBCDIC から Unicode に変換されなければなりません。ただし、CCI Connector for CICS TS は、このデータ変換を自動的に処理します。Unicode との間の両方向の変換の場合、このコネクターで発行される JCICS Program.link() 呼び出しは、代替コーディング・システムとして、実

行環境のコーディング・システムを使用します。このコネクターは z/OS で実行されるので、代替コーディング・システムは EBCDIC です。

注: デフォルトで、コネクターの Interaction.execute() メソッドに渡される Record オブジェクトは、コネクターの実行環境で使用される EBCDIC コード・ページを使用します。

CCI Connector for CICS TS のインストール

CCI アプリケーションのコンパイル

CCI Connector for CICS TS を使用するアプリケーションをコンパイルするには、以下の CICS 提供 JAR ファイルを Java クラスパスに組み込む必要があります。

connector.jar
すべての CCI アプリケーションで必要な CCI API

dfjcci.jar
CICS TS での CCI API の実装

CICS をインストールするときに、connector.jar は %JAVA_HOME%/standard/jca z/OS UNIX ディレクトリーにインストールされます (ここで、%JAVA_HOME% は、DFHISTAR CICS インストール・ジョブの JAVADIR パラメーターの値です)。dfjcci.jar は、/usr/lpp/cicsts/cicsts42/lib ディレクトリーにインストールされます (ここで、cicsts42 は、DFHISTAR インストール・ジョブの USSDIR パラメーターの値です)。

接続ファクトリーの管理および獲得のためのサンプル・ユーティリティー・プログラムの使用

このタスクについて

CICS は、次の方法を示す 3 つのサンプル・プログラムを提供しています。

1. JNDI ネームスペースへの接続ファクトリーの公開 (CICSConnectionFactoryPublish サンプル)。このサンプルを使用して、CCI Connector for CICS TS で使用するのに適した **ConnectionFactory** オブジェクトを作成し、それをローカル CICS 領域で使用される JNDI ネームスペースに公開することができます。これにより、CICS 上で実行されるエンタープライズ Bean または Java プログラムが、 JNDI 検索を実行して接続ファクトリーへの参照を取得できるようになります。

   このサンプルについては、 381 ページの『CICSConnectionFactoryPublish を使用した接続ファクトリーの公開』 で説明しています。
2. 前に公開された接続ファクトリーの JNDI ネームスペースからの撤回 (CICSConnectionFactoryRetract サンプル)。このサンプルについては、 382 ページの『CICSConnectionFactoryRetract を使用した接続ファクトリーの撤回』 で説明しています。
3. JNDI ネーム・スペースでの接続ファクトリーの検索 (CCI Connector サンプル・アプリケーション)。このサンプルでは、CCI Connector for CICS TS を使用して CICS サーバー・プログラムを呼び出す方法も示しています。これについては、 383 ページの『CCI Connector サンプル・アプリケーション』 で説明しています。

CICSConnectionFactoryPublish および CICSConnectionFactoryRetract サンプルを使用することで、接続ファクトリーの作成、パブリッシュ、および管理を、これを使用するアプリケーションとは切り離して実行できます。

サンプル・プログラムを使用するには、適切に構成されたネーム・サーバーが必要です。ネーム・サーバーの構成が必要な場合は、 431 ページの『JNDI 参照の使用可能化』 および 431 ページの『JNDI ネーム・サーバーの場所の指定』 を参照してください。

publish および retract サンプル・プログラムのインストール:

このセクションでは、CICSConnectionFactoryPublish および CICSConnectionFactoryRetract プログラムをインストールする方法について説明します。

このタスクについて

CCI Connector アプリケーションのインストール方法については、 385 ページの『CCI Connector サンプルのインストール』で説明しています。

CICS 提供 JAR ファイル CICSCCISamples.jar には、サンプル・プログラムのオブジェクト (.class) ファイルが入っています。CICS は、CICSCCISamples.jar を /usr/lpp/cicsts/cicsts42/samples/cci ディレクトリーにインストールします (ここで、/usr/lpp/cicsts/cicsts42 は、z/OS UNIX 上の CICS ファイルのインストール・ディレクトリーです)。また、プログラムのソース (.java) ファイルも /usr/lpp/cicsts/cicsts42/samples/cci ディレクトリーにインストールされます。

CICSConnectionFactoryPublish および CICSConnectionFactoryRetract プログラムをインストールするには、次の手順を実行します。

手順

1. プログラムが入っている JAR ファイル /usr/lpp/cicsts/cicsts42/samples/cci/CICSCCISamples.jar を、それらのプログラムが使用する JVM プロファイルの CLASSPATH_SUFFIX ステートメントに追加します。提供されるサンプル・プログラムは、CICS 提供のサンプル JVM プロファイル DFHJVMPR を使用します。これは、プログラムのリソース定義で JVM プロファイルが指定されない場合のデフォルトです。CICS は、DFHJVMPR を /usr/lpp/cicsts/cicsts42/JVMProfiles ディレクトリーにインストールします。
2. 編集されたバージョンの DFHJVMPR を、**JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーに入れます (デフォルトの CICS インストールでは、**JVMPROFILEDIR** は /usr/lpp/cicsts/cicsts42/JVMProfiles を指定します)。
3. CEDA を使用して、トランザクション CCPB および CCRT をグループ DFH$CCI からインストールします。
4. CEDA を使用して、プログラム DFJ$CCPB および DFJ$CCRT をグループ DFH$CCI からインストールします。

   注: CICS 領域でプログラムの自動インストールが使用される場合、この最後の手順は不要です。

タスクの結果

CICSConnectionFactoryPublish を使用した接続ファクトリーの公開:

CICSConnectionFactoryPublish プログラムは、以下のタスクを実行します。

1. CICS 領域の初期 JNDI コンテキストを取得します。
2. ConnectionFactory subContext がコンテキスト構造に存在するかどうかを確認します。
3. ConnectionFactory subContext が存在しない場合は、それを作成します。
4. ConnectionFactory/CICSConnectionFactory 接続ファクトリーがネーム・サーバーにまだ公開 (バインド) されていない場合は、それを公開します。

提供されたバージョンの CICSConnectionFactoryPublish プログラムで設定される、接続ファクトリーのデフォルト名は、CICSConnectionFactory です。接続ファクトリーが公開される JNDI subContext のデフォルト名は ConnectionFactory です。 CICSConnectionFactoryPublish プログラムのソース・コードを編集すると、以下のものを変更できます。

- 接続ファクトリーの名前。
- JNDI subContext。
- リンク先サーバー・プログラムがリモートである場合、リモート領域でそのプログラムが実行されるミラー・トランザクションの名前。ただし、ミラー・プログラムは、サーバー・プログラムのローカル PROGRAM 定義で指定することをお勧めします。

変更を加える方法については、ソース・コード内のコメントを参照してください。

接続ファクトリーの名前、または subContext の名前を変更する場合は、必ず、3 つのすべてのサンプル・プログラムで同じ変更を加えてください。

プログラムの実行:

CCI Connector for CICS TS で使用するのに適した ConnectionFactory を CICS JNDI ネーム・サーバーに公開 (バインド) するには、トランザクション CCPB を実行します。

CICSConnectionFactoryPublish プログラムを変更した場合を除いて、ConnectionFactory は、名前が CICSConnectionFactory に指定され、JNDI サーバーのネーム・スペース内の subContext ConnectionFactory に公開されます。

次のメッセージが画面に表示されます。

```
ccpb - ConnectionFactory published to JNDI successfully.
```

注: 同じ名前と subContext を持つ ConnectionFactory が、JNDI サーバーに既に公開されている (かつ、撤回されていない) 場合、別のメッセージが表示されます。

```
ccpb - The ConnectionFactory is already published to JNDI.
```

接続ファクトリーが正常に公開されることを前提とすれば、次の出力が **stdout** に送信されます。

```
*************************************************************************************************
**** CICSConnectionFactoryPublish: Started
**** CICSConnectionFactoryPublish: Binding ConnectionFactory ConnectionFactory/CICSConnectionFactory
**** CICSConnectionFactoryPublish: ConnectionFactory bound to JNDI
**** CICSConnectionFactoryPublish: Ended
*************************************************************************************************
```

図 28. デフォルトの名前と subContext を持つ ConnectionFactory を公開するためのトランザクション CCPB からの Stdout 出力

CICSConnectionFactoryPublish を PLTPI プログラムとして実行したり、PLTPI プログラムからリンクしたりすることはお勧めしません。これは、JVM が使用不可である場合、CICS の起動時間が長くなるからです。

接続ファクトリーの検索:

このコード例は、CICS で使用される JNDI ネーム・スペース内で以前に公開された接続ファクトリーを検索する方法を示しています。

```
// Declare a ConnectionFactory object
ConnectionFactory cf = null;

try{
    // Get the initial JNDI context
    javax.naming.Context ic = new javax.naming.InitialContext();

    // Do the lookup, casting the returned CICSConnectionFactory to type
    // ConnectionFactory
    cf = (ConnectionFactory)ic.lookup("ConnectionFactory/CICSConnectionFactory");

    // Use the connection factory to create a connection to CICS
    Connection eciConn = (Connection)cf.getConnection();
}
catch (Exception e){
    // Lookup failed, or specified connection factory has not been published
    // Exception processing
}
```

これは CCI Connector アプリケーションで具体的に説明されています。383 ページの『CCI Connector サンプル・アプリケーション』を参照してください。

CICSConnectionFactoryRetract を使用した接続ファクトリーの撤回:

公開した接続ファクトリーを撤回 (アンバインド) するには、トランザクション CCRT を実行します。CICSConnectionFactoryRetract プログラムを変更した場合を除いて、撤回される ConnectionFactory は、JNDI サーバーのネーム・スペースにある subContext ConnectionFactory 内の CICSConnectionFactory です。

次のメッセージが画面に表示されます。

```
ccrt - ConnectionFactory retracted from JNDI successfully.
```

注: CICSConnectionFactoryRetract プログラムで指定される ConnectionFactory が JNDI サーバーに存在しない (例えば、既に撤回されている) 場合、別のメッセージが表示されます。

```
ccrt - unable to locate ConnectionFactory on JNDI.
```

接続ファクトリーが正常に撤回されることを前提とすれば、次の出力が **stdout** に送信されます。

```
********************************************************************************
**** CICSConnectionFactoryRetract: Started
**** CICSConnectionFactoryRetract: Unbinding ConnectionFactory/CICSConnectionFactory
**** CICSConnectionFactoryRetract: ConnectionFactory/CICSConnectionFactory unbound
**** CICSConnectionFactoryRetract: Ended
********************************************************************************
```

図 29. デフォルトの名前と subContext を持つ接続ファクトリーを撤回するためのトランザクション CCRT からの Stdout 出力

`CICSConnectionFactoryRetract` を PLTSD プログラムとして実行したり、PLTSD プログラムからリンクしたりすることはお勧めしません。これは、CICS のシャットダウン時間が長くなるからです。

CCI Connector サンプル・アプリケーション

CCI Connector サンプルは、CCI API を直接コーディングする方法を示す、相対的にシンプルなアプリケーションです。

次の方法を具体的に示します。

1. 以前に公開された接続ファクトリーを JNDI ネーム・スペースで検索する方法
2. CCI Connector for CICS TS を使用して、CICS サーバー・プログラムを呼び出す方法

このサンプルは次のもので構成されます。

- CICS Java プログラム
- javax.resource.cci.Streamable インターフェースの使用を示すカスタム Record
- CICS COBOL サーバー・プログラム

このサンプルの仕組みは次のとおりです。

1. ユーザーは、CICS 端末から CCCI トランザクションを実行することによって、アプリケーションを開始します。
2. CICS Java プログラム `CICSCCISample` (DFJ$CCIC) が開始します。この Java プログラムは以下のことを行います。
   a. 一連のランダムで未ソートの 10 進数を入力するようにユーザーに要求します。
   b. ネーム・サーバーの JNDI 検索を行って、CICS 接続ファクトリーを取得します。
   c. 接続ファクトリーがネーム・サーバーに公開されていない場合は、接続ファクトリーをプログラマチックに作成します。
   d. 接続ファクトリーを使用して、CICS との接続を作成します。
   e. Interaction オブジェクトを **Connection** オブジェクトから作成し、ECIInteractionSpec オブジェクトを使用して対話のプロパティー (ターゲット・プログラムの名前を含めて) を設定します。
   f. Interaction.execute メソッドを使用して、COBOL プログラム DFH$0CCIS にリンクして、ユーザーの一連の未ソートの数字に加えて ECIInteractionSpec オブジェクトを、入力として (カスタム Record オブジェクト内で) 渡します。

3. COBOL プログラムは数字を昇順にソートし、ソートされたシーケンスを出力 COMMAREA で戻します。
4. Java プログラムは、COBOL プログラムの出力を出力 **Record** オブジェクトから取り出し、ソートされたリストをユーザーの端末に表示します。

図 30 は、このサンプル・アプリケーションのコンポーネントを示しています。

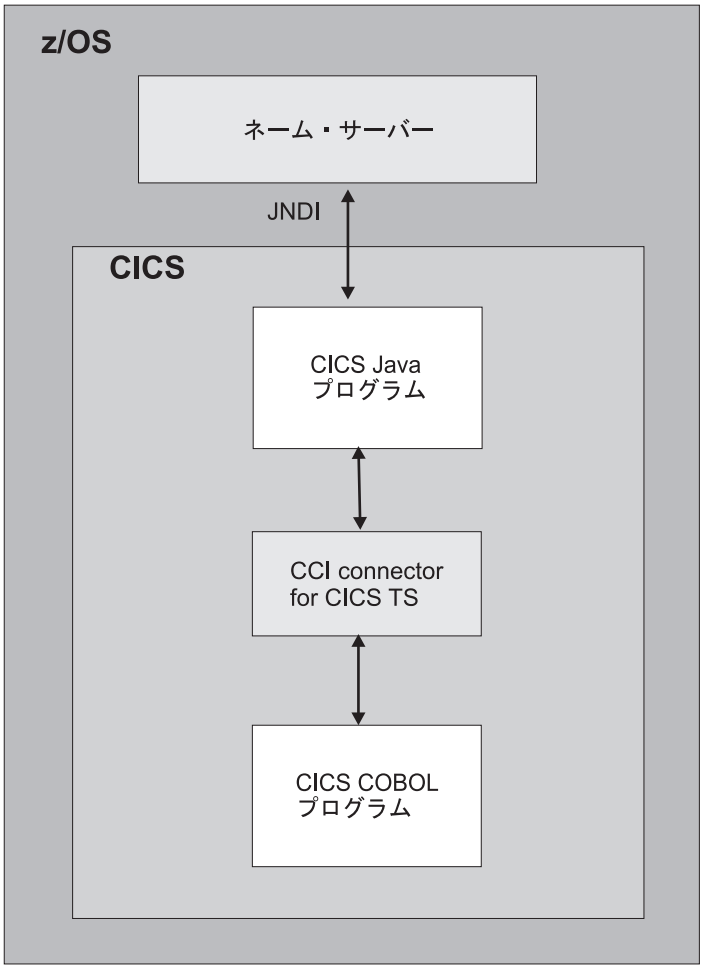


図 30. CCI Connector サンプル・アプリケーションの概要： このサンプルの主要素は、CICS Java プログラムと CICS COBOL サーバー・プログラムです。Java プログラムは CCI Connector for CICS TS を使用して、COBOL サーバー・プログラムにリンクします。 CICS 接続ファクトリーは、COS ネーム・サーバーか LDAP ネーム・サーバーのどちらかに公開できます。

CCI Connector サンプルの要件:

JNDI 検索を実行することによって、CCI Connector サンプルが CICS 接続ファクトリーを取得できるようにするには、Java Naming and Directory Interface (JNDI)、バージョン 1.2 以降をサポートするネーム・サーバーが必要です。

このセットアップ方法については、 281 ページの『z/OS または Windows NT で必要なアクション』で説明しています。COS ネーム・サーバーか LDAP サーバーのどちらかを使用できます。

ただし、このサンプルがネーム・サーバーに接続できないか、CICS 接続ファクトリーがネーム・サーバーに公開されていない場合、このサンプルは接続ファクトリーをプログラマチックに作成します。したがって、厳密に言えば、ネーム・サーバーは、このサンプルを実行するための要件ではありません。

CCI Connector サンプルのインストール:
このタスクについて

手順

1. CICSConnectionFactoryPublish および CICSConnectionFactoryRetract サンプルの実行時にまだ行っていない場合は、サンプル・プログラム /usr/lpp/cicsts/cicsts42/samples/cci/CICSCCISamples.jar が入っている JAR ファイルの位置を確認します。ここで、/usr/lpp/cicsts/cicsts42 は、z/OS UNIX 上の CICS ファイルのインストール・ディレクトリーです。この JAR ファイルを、それらのプログラムが使用する JVM プロファイルの CLASSPATH_SUFFIX ステートメントに追加します。 提供されるサンプル・プログラムは、CICS 提供のサンプル JVM プロファイル DFHJVMPR を使用します。これは、プログラムのリソース定義で JVM プロファイルが指定されない場合のデフォルトです。

   CICS は、DFHJVMPR を /usr/lpp/cicsts/cicsts42/JVMProfiles ディレクトリーにインストールします。

   編集されたバージョンの DFHJVMPR を、**JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーに置きます。

2. connector.jar および dfjcci.jar ファイルが標準クラスパスにあることを確認します。

   注: 379 ページの『CCI アプリケーションのコンパイル』で説明されているとおり、CICS をインストールするときに、connector.jar は %JAVA_HOME%/standard/jca ディレクトリーにインストールされ、dfjcci.jar は /usr/lpp/cicsts/cicsts42/lib ディレクトリーにインストールされます。/usr/lpp/cicsts/cicsts42/lib ディレクトリーは、CICS によって作成された基本クラスパスにあります。これは、JVM プロファイルには表示されません。したがって、このディレクトリーは常に組み込まれています。

3. ネーム・サーバーが実行していることを確認します。
4. CICSConnectionFactoryPublish プログラムを使用して、CCI Connector for CICS TS で使用するための ConnectionFactory オブジェクトを作成し、それをネーム・サーバーに公開します。 381 ページの『CICSConnectionFactoryPublish を使用した接続ファクトリーの公開』を参照してください。
5. CEDA を使用して、トランザクション CCCI をグループ DFH$CCI からインストールします。
6. CEDA を使用して、CICS Java および COBOL プログラムの定義をインストールします。 プログラム DFJ$CCIC および DFH0CCIS をグループ DFH$CCI からインストールします。

   注: CICS 領域でプログラムの自動インストールが使用される場合、この手順は不要です。

サンプルのテスト:
このタスクについて

CCI Connector サンプルをテストするには、次の手順を実行します。

1. トランザクション CCCI を CICS 端末で開始します。
2. サンプルから、いくつかの数字を入力するように求められます。5 個以上の 10 進数をスペースで区切って入力して、Enter キーを押します (各数字は、5 桁以下でなければなりません。これらの数字はサイズ順に並べる必要はありません)。
3. このサンプルは、ソートされた数字のリストを画面および **stdout** に書き込みます。例えば、数字 54、3、77、55、および 19 を入力した場合、画面には次のように表示されます。

```
CCCI - CCI sample transaction starting.

A Connection object has been instantiated.

An Interaction object has been instantiated.

Enter a series of numbers:  54 3 77 55 19

An InteractionSpec object has been instantiated.

Connecting to program DFH0CCIS by invoking execute() on Interaction object.

Commarea sent:    54    3   77   55   19*

Commarea returned:     3   19   54   55   77*

CCCI - CCI sample transaction finished.
```

問題判別

CCI Connector for CICS TS メッセージおよび CICS トレースを使用して問題を診断できます。

CCI Connector for CICS TS メッセージ:

CCI Connector for CICS TS に関連した CICS メッセージは、「*CICS Messages and Codes Vol 1*」マニュアルで説明されています。

CCI Connector for CICS TS のトレース:

このコネクターに関連した CICS トレース・ポイントは、EJ 0600 から EJ 06FF の範囲内にあります。

これについては、Trace Entries のトレース項目の概要を参照してください。

コネクターからの CICS トレース情報の出力を制御するには、通常どおりに CICS トレース制御を使用してください。

CICS Connector for CICS TS から CCI Connector for CICS TS へのアップグレード

CICS Connector for CICS TS を使用する既存のアプリケーションがある場合は、代わりに CCI Connector for CICS TS を使用するようにそれらのアプリケーションをアップグレードする必要があります。

表 20 では、CICS Connector for CICS TS または CCI Connector for CICS TS のいずれかを使用する CICS Java コンポーネントのアップグレードの選択肢をまとめ、事例ごとの推奨解決方法を示しています。

表 20. CICS CCF または CCI コネクターを使用する CICS Java コンポーネントの推奨アップグレード・パス

現行プログラムで使用されているコネクター	現行プログラムで使用されているコネクター・インターフェース	CICS TS 4.2 の状況	推奨されるアップグレード戦略
CICS Connector for CICS TS	CICS Transaction Gateway API (**ECIRequest**)	Not supported	CICS Transaction Gateway API はサポートされなくなりました。 CCI Connector for CICS TS を使用するようにリエンジニアリングしてください。コネクターを直接プログラミングするか、そのコネクターをサポートする rapid application development (RAD) ツールを使用してコネクターをプログラミングします。
CICS Connector for CICS TS	直接プログラミングされるか、または VAJ Enterprise Access Builder もしくはそれに類似するものを使用してプログラミングされる CCF	Not supported	CCF は CCI で置き換えられます。CCI Connector for CICS TS を使用するようにリエンジニアリングします。これは、CICS Connector for CICS TS よりもパフォーマンスがすぐれ、業界標準のインターフェースを使用します。コネクターを直接プログラミングするか、そのコネクターをサポートする RAD ツールを使用してコネクターをプログラミングします。 **注:** VAJ Enterprise Access Builder を使用して CCI Connector for CICS TS をプログラミングすることは可能ですが、VAJ/EAB がサポートされなくなったためこれはお勧めしません。
CCI Connector for CICS TS	直接プログラミングされる CCI	Supported	CCI を無制限に使用できます。CCI を直接プログラミングすると、最良のパフォーマンスが得られます。
CCI Connector for CICS TS	VAJ Enterprise Access Builder もしくはそれに類似するものを使用してプログラミングされる CCI	Supported	VAJ/EAB を引き続き使用するには、アプリケーションに変更を加える必要があります。

CICS エンタープライズ Bean の問題の処理

このセクションには、CICS エンタープライズ Bean サポートのセットアップと使用の問題の処理に関する情報が記載されています。

CICS の問題判別と診断の一般情報については、Problem Determination の問題判別の概要を参照してください。

- 388 ページの『CICS エンタープライズ Bean のセットアップの問題』
- 389 ページの『EJB サーバーの実行時診断の使用』
- 390 ページの『EJB クライアントの実行時診断の使用』
- 393 ページの『RMI-IIOP のクラス・バージョンの問題』
- 394 ページの『EJB トレースと保守性コマンドの使用』

CICS エンタープライズ Bean のセットアップの問題

CICS EJB サーバーのセットアップに問題がある場合、その問題は基本的な CICS Java セットアップに関連する可能性があります。Java HelloWorld サンプルを実行してみてください。これも失敗する場合、他の問題ではなく、JVM のセットアップの問題を示しています。

複数の要求プロセッサーを必要とするメソッド:

エンタープライズ Bean メソッドの単独の実行で、複数の要求プロセッサーが必要とされている場合には、アプリケーションにデッドロックの問題が発生することがあります。

このタスクについて

(メソッドは、そのメソッドが、異なる要求プロセッサーで実行する必要がある 1 つ以上のメソッド (通常はリモート) を呼び出す場合には、「複数の要求プロセッサーが必要」ということが言えます。) デッドロックは、これ以上 JVM が許可されない場合に、JVM を強制的に待機させられているメソッドを満足させるために必要なすべての要求プロセッサーで発生する可能性があります。これは、以下の 2 つが原因で発生します。

1. 単純なケースとしては、CICS (**MAXJVMTCBS**) で並行して存在することが許可されている最大 JVM 数が、そのメソッド要求をサービスするために必要な要求プロセッサー数よりも小さい場合。
2. 複雑な場合:
   - CICS は、多重要求を同時に処理しています。
   - すべての要求は、他の JVM を待っています。
   - すべての許可された JVM は、現在使用中です。

単純な事例を回避するのは簡単です。複雑な事例を回避するのはより困難です。要求プロセッサー・インスタンスの少なくとも 1 つのメソッドの要件を満足させるには、常に十分なフリー JVM があることを確認することが必要です。

Bean メソッドが使用可能な同時 JVM の最大数は、要求プロセッサー・トランザクションに対する TRANCLASS 定義の MAXACTIVE 属性によって設定されます。CICS が使用可能な同時 JVM の最大数は、**MAXJVMTCBS** システム初期設定パラメーターによって設定されます。

複数の要求プロセッサーを使用する Bean メソッドによって発生するデッドロックの可能性を排除するには、以下を行います。

1. アプリケーション要件と整合性がある限り、それぞれのメソッドが必要とする要求プロセッサーの数を最小値にし、可能であれば 1 にするようにしてください。すべてのメソッドの要件をすべてのアプリケーションにおいて 1 つの要求プロセッサーに削減できる場合には、それ以上削減する必要はありません。
2. すべてのメソッドの要件を 1 つの要求プロセッサーに削減することができない場合には、どれが「ワーストケース」なのかを発見します。すなわち、要件を満足させるためにほとんどの要求プロセッサーを必要とする Bean メソッドです。
3. 新規 TRANCLASS 定義を作成します。このトランザクション・クラスは、複数の要求プロセッサーを必要とする Bean メソッドが稼働する要求プロセッサー・トランザクションに適用します。

4. TRANCLASS 定義で、次の数式を使用して MAXACTIVE の値を設定します。

   ```
   MAXACTIVE <= ((MAXJVMTCBS - n) / (n - 1)) + 1
   ```

   ここで、n は、ユーザーの「ワーストケース」のメソッドが必要とする要求プロセッサーの最大数です。

   この計算の結果が小数値の場合は、一番近い整数に切り下げます。
5. 新規 TRANSACTION および REQUESTMODEL 定義を以下のように作成します。
   a. 複数の要求プロセッサーを必要とする Bean メソッドが稼働する要求プロセッサー・トランザクションに対して、新規に TRANSACTION 定義を作成します。(これを行うために一番簡単な方法は、デフォルトの CIRP 要求プロセッサー・トランザクションの定義をコピーして、そのコピーを変更することです。)TRANCLASS オプションで、新規トランザクション・クラスの名前を指定します。
   b. 1 つ以上の REQUESTMODEL 定義を作成します。これらの定義の間では、新規 REQUESTMODEL 定義は、複数の要求プロセッサーを要求する Bean メソッド用に受信するすべての要求をカバーする必要があります。REQUESTMODEL 定義の TRANSID オプションで、新規トランザクション名を指定します。

EJB サーバーの実行時診断の使用

EJB サーバーは、問題の診断と解決に役立つ実行時診断を行います。これらの診断には、エラー・メッセージ、JVM トレース、および Java Platform Debugger Architecture (JPDA) が含まれます。

CICS エンタープライズ Bean のエラーとメッセージ:

CICS からのエラー・メッセージを探す場所のリストは、次のとおりです。

Enterprise Java ドメイン (DFHEJnnnn) メッセージ
CICS は、Enterprise Java ドメインから多数の情報メッセージ、警告メッセージおよびエラー・メッセージを発行します。これらの大部分は、CEJL および CJRM 一時データ・キューに転送され、残りはコンソールに送信されます。完全なリストについては、「*CICS Messages and Codes Vol 1*」マニュアルを参照してください。

CICS JVM (DFHSJnnnn) メッセージ
これは、CICS JVM によって発行されるメッセージです。大部分は一時データ・キュー CSMT に転送されます。完全なリストについては、「*CICS Messages and Codes Vol 1*」マニュアルを参照してください。

CICS Development Deployment Tool (DFHADnnnn) メッセージ
これは、このツールで発行され、SYSPRINT メッセージとして CICS に転送されるメッセージです。完全なリストについては、「*CICS Messages and Codes Vol 1*」マニュアルを参照してください。

CICS 異常終了コード

- AJMA から AJM9 は CICS JVM によって発行されます
- AJ01 から AJ99 は Java 環境のセットアップ・クラス・ラッパーによって発行されます。

リストについては、「*CICS Messages and Codes Vol 1*」マニュアルを参照してください。

JVM トレース:

Java 仮想マシン (JVM) には、独自の内部トレース機能があります。 JVM トレースは、JVM における問題の診断に役立ちます。JVM トレースは大量の出力を作成する可能性があるため、すべてのトランザクションに対してグローバルにオンにするのではなく、特定のトランザクションに対して活動化するようにしてください。

220 ページの『プールされた JVM 用のトレースの定義および活動化』では、プールされた JVM トレースを活動化し、JVM トレース・オプションを変更するためのさまざまな方法について説明しています。

JVM トレースを活動化すると、生成される各 JVM トレース・ポイントは、SJ ドメインの CICS トレース・ポイントのインスタンスとして表示されます。

JVM トレース・オプションに加えて、 CICS トレース・レベル 0、1 および 2 の、SJ ドメインの標準トレース・ポイントを使用すると、CICS が JVM と共用クラス・キャッシュのセットアップおよび管理について行うアクションをトレースできます。

Java platform debugger architecture (JPDA):

CICS における JVM は、Java 2 Platform で提供される標準デバッグ・メカニズムである Java Platform Debugger Architecture (JPDA) をサポートします。

CICS における JVM は、Java 2 Platform で提供される標準デバッグ・メカニズムである Java Platform Debugger Architecture (JPDA) をサポートします。このアーキテクチャーは、リモート・デバッガーと JVM との接続を可能にする 1 組の API を提供します。さまざまなサード・パーティー・デバッガーが使用可能です。これらのデバッガーは、JPDA を活用し、エンタープライズ Bean、CORBA オブジェクトまたは CICS Java プログラムを実行する JVM に接続し、デバッグするのに使用できます。通常、デバッガーが提供するグラフィカル・ユーザー・インターフェースは、ワークステーションで実行されます。これを使用すると、アプリケーションのフローをたどって、ブレークポイントの設定やアプリケーションのソース・コードのステップスルーとともに、変数の値の検査を行うことができます。

CICS JVM でのデバッガーのセットアップと使用については、 223 ページの『Java アプリケーションのデバッグ』を参照してください。

JPDA および JPDA 準拠アプリケーションに関する情報を見つけるには、Oracle Technology Network Java Web サイトに進み、*Java Platform Debugger Architecture* を検索して JPDA ホーム・ページを見つけてください。

EJB クライアントの実行時診断の使用

クライアントによって発行される大部分のエラー・メッセージは、問題が CICS にある場合はあまり役に立ちません。しかし、場合によっては、クライアントから役立つ情報を入手でき、それが明白な出発点になります。

さらに役に立つクライアント例外のいくつかは次のとおりです。

NoClassDefFoundException および ClassNotFoundException

クライアントがこれらのいずれかを発行すると、おそらく、クライアント・サイドのクラスパスで何かが欠落しているか、壊れています。この例外は、どのクラスが欠落しているかを適切に示し、これからクラスパスに追加する JAR ファイルを導き出すことができる場合があります。j2ee.jar、およびクラスパス内で完全にデプロイされた jar が必要であることに注意してください。CICS はおそらく、これらの問題に役立つ追加情報を出しそうにありません。

NoClassDefFoundError:javax/ejb/HomeHandle

これは、クライアント・アプリケーションに、クラスパスで使用可能な EJB 1.1 レベルのクラスがないことを示しています。j2ee.jar が使用可能であることを確認してください。

ObjectNotFoundException

この例外は、セッション Bean がタイムアウトになったか、複数の並行トランザクションでセッション Bean を使用しようとしたことを示す場合があります。

RemoteException

これは、サーバー・アプリケーションの問題を示し、多くの場合、詳細情報を提供するネストされた例外が入っています。該当するものを以下に示します。

NoClassDefFoundError

これは、サーバー・サイド上で欠落した JAR ファイルを指します。追加情報がないか、CICS システム・コンソールおよび JVM 標準エラー・ファイルと出力ファイルを調べてください。

CORBA.INTERNAL

これは、JVM の外部でサーバー・サイド・アプリケーション (例えば、エンタープライズ Bean によって呼び出された COBOL プログラム) の障害を示しています。詳細情報がないか、CICS システム・コンソールを調べてください。

CORBA 例外:

これらの例外は、役に立つ情報を提供する場合があります。

完了状況は、次の 3 つの値のいずれかになります。

- **No** は、サーバーが明らかに、呼び出されたメソッドの実行を正常に完了しなかったことを意味します。
- **Yes** は、サーバー上で呼び出されたオペレーションが完了したことを意味します。
- **Maybe** は、サーバー上でオペレーションが完了したかどうかをクライアントが判別できないことを意味します。

完了状況が **Yes** である場合、サーバー上で実行するものをクライアントが検出したことを確信できます (ただし、JNDI/IOR が正しくない場合、正しいエンタープライズ Bean でなかったか、予想された CICS 領域上ではなかった可能性があります)。通常、メソッド呼び出しが失敗した理由に関する、さらに役に立つ情報が CICS 出力で見つかります。

クライアントが受信するより一般的な CORBA 例外のいくつかは、次のとおりです。

org.omg.CORBA.COMM_FAILURE
これは、次のいずれかの状態で生じる可能性があります。

- JNDI ネーム・サーバーが実行していない (JNDI 検索中である場合)
- エンタープライズ Bean が JNDI ネーム・サーバーに公開されていない
- CICS 領域がダウンしている
- TCPIPSERVICE がインストールされていないか、(CICS 上のメソッド起動に) オープンである

org.omg.CORBA.INTERNAL
この原因は、通常、サーバー・サイド・アプリケーションの異常終了または障害です。詳細情報がないか、CICS コンソールを調べてください。

org.omg.CORBA.INVALID_TRANSACTION
これが生じる可能性があるのは、Web アプリケーション・サーバーと CICS 間でトランザクションの相互運用性の問題があるからです。

分散トランザクションをサポートするために、複数のプロトコルが存在します。CICS Enterprise Java 環境は、標準の CORBA オブジェクト・トランザクション・サービス (OTS) プロトコルのみをサポートします。しかし、いくつかの J2EE 準拠 Web アプリケーション・サーバー (WebSphere バージョン 4 など) はこのプロトコルを使用しないか、デフォルトでこのプロトコルを使用しません。(バージョン 5 以降の WebSphere Application Server のバージョンは、この問題の影響を受けません。)

*Web アプリケーション・サーバー上のオブジェクトが、既存のトランザクション・コンテキストの有効範囲内で CICS エンタープライズ Bean を呼び出す場合、CORBA OTS を使用するように Web アプリケーション・サーバーをセットアップする必要があります。*これが可能でない場合、Web アプリケーション・サーバーは CICS エンタープライズ Java サポートとの完全な互換性がありません (EJB Bank Account サンプル・アプリケーションを使用して、Web アプリケーション・サーバーが CICS エンタープライズ Java サポートとの完全な互換性があるかどうかをテストする方法については、334 ページの『分散トランザクションに関する注記』を参照してください)。

WebSphere Application Server が CORBA OTS を使用するように強制するには、次の手順を実行します。

1. WebSphere 管理コンソールで、「JVM settings」タブを選択します。
2. 「System Properties」セクションで次のように入力します。

   ```
   com.ibm.ejs.jts.ControlSet.interoperabilityOnly=true
   com.ibm.ejs.jts.ControlSet.nativeOnly=false
   ```

   変更内容を保管します。
3. アプリケーション・サーバーを再始動します。

org.omg.CORBA.OBJECT_NOT_EXIST
これが生じる可能性があるのは、クライアントが JNDI ネーム・サーバーで Bean への参照を検出するものの、その Bean が CICS にインストールされなくなった場合です。

org.omg.CORBA.UNKNOWN

コード内のエラーや CICS 内のエラーを含めて、この例外には複数の理由があります。この問題の原因のヒントについては、CICS 出力を参照してください。

多くのインスタンスで、CORBA 例外には、問題判別に役立つ CICS 固有のマイナー・コードが含まれています。CICS は現在、次のマイナー・コードを使用します。

表 21. CICS 固有の CORBA マイナー・コード

コード	問題を検出する CICS コンポーネント
1229111296	CICS IIOP 要求受信側
1229111297	CICS II ドメイン内の他のどこか
1229111298	CICS OT ドメインの ORB コンポーネント
1229111299	CICS OT ドメインの JTS コンポーネント
1229111300	CICS OT ドメインの CSI コンポーネント
1229111301	CICS EJ ドメインの CSI コンポーネント

クライアントが、いずれかの CICS マイナー・コードを含む CORBA 例外を受け取る場合は、エラーに関する追加情報がないか、CICS メッセージ・ログを調べる必要があります。

RMI-IIOP のクラス・バージョンの問題

Remote Method Invocation over IIOP (RMI-IIOP) は、エンタープライズ Bean と CORBA ステートレス・オブジェクトの両方によって、CICS で使用される通信プロトコルです。したがって、このセクションの情報は、エンタープライズ Bean と CORBA ステートレス・オブジェクトの両方に適用されます。

Java RMI は、object-by-value プロトコルです。これは、Java オブジェクトがメソッド呼び出しでパラメーターとして使用されるときはいつでも、ワイヤー上で送信されるものがオブジェクト状態であることを意味します。これと同じことが、戻りの型と例外にも当てはまります。この状態は、「直列化された」Java オブジェクトです。この状態は、リモート JVM でオリジナル・オブジェクトの新しいコピーを作成するために、リモート JVM によって非直列化できます。直列化された状態には、特に、状態が表すクラスのバージョンを示すバージョン番号が含まれます。直列化オブジェクトがリモート JVM によって非直列化されるには、同じバージョンのクラス・ファイルが IIOP 接続の両側に存在することが必要です。リモート JVM がオブジェクト状態を認識できない場合、おそらく、次の例外がスローされます。

```
java.rmi.MarshalException:unable to read from underlying bridge
```

(この例外は、その他の理由でもスローされる場合があります。)

Java でクラスを作成する場合、独自のカスタマイズされた直列化メカニズムを提供することが可能です。このメカニズムを使用すると、Java のデフォルト直列化プロセスを利用するのではなく、クラスのバージョン管理を明示的に処理できます。さらに、カスタム直列化メカニズムを提供する場合、デフォルトのメカニズムよりも大幅にパフォーマンス改善を実現できます。カスタム直列化を利用したい場合、オブジェクトは `java.io.Externalizable` インターフェースを実装しなければなりません。

多くの場合、直列化が必要なオブジェクトは、標準 Java クラス・ライブラリーからのクラスのインスタンスです。これらは通常、あるバージョンの Java から次のバージョンに変わりませんが、変わると、上記で説明した種類の問題が生じる可能性があります。これらの問題を最小化するために、CICS が使用するのと同じバージョンの Java をパートナー・マシンで使用することをお勧めします。例えば、Java 1.3.1 と Java 1.4 間で、`java.lang.Throwable` クラスは大幅に変更されました。このクラスは、Java におけるすべての例外のスーパータイプであり、したがって Java 1.4.1 以降で直列化された多くの例外を、旧バージョンの Java で非直列化できません。

クラスにおけるバージョン変更の問題を回避するために、多くの ORB で使用されるメカニズムが CORBA にあります。残念ながら、そのメカニズムは CICS では完全に機能するわけではありません。CICS ではパートナー ORB と JVM との間の親和性が必要であるからです。CICS における同一 CORBA オブジェクトへの複数の RMI-IIOP 呼び出しは、おそらく、別々の JVM で処理されます。つまり、親和性はサポートされず、クラスのバージョン管理の問題を回避するためのメカニズムは CICS では機能しません。CICS アプリケーションがこの問題を検出するのは、直列化オブジェクトをリモート JVM に送信する場合のみです。リモート JVM が直列化オブジェクトを CICS に送信する場合、CICS は、標準の CORBA メカニズムを使用して、バージョンの非互換性に対処することができます。

この種類の問題を検出するときに、パートナー・プラットフォームで使用中の Java のバージョンを変更できない場合、バージョン管理の問題を生じないデータ型を使用するようにアプリケーションを変更することをお勧めします。

EJB トレースと保守性コマンドの使用

要求の停止または失敗を診断しようとする場合、またはアクティビティーをモニターするか、またはおそらくアカウンティングのために単一の要求に関連したすべてのトランザクションを一意的に識別できなければならない場合、EJB 要求のトレースが必要になることがあります。

EJB 論理サーバーが複数の CICS 領域から成る場合、要求の停止または失敗を診断しようとするときの主な問題は、以下のものを判別しなければならないことです。

- 要求が発生した領域 (要求受信側)
- 要求が転送された先のターゲット (CICS 領域またはその他のサーバー)

システム・プログラミング・インターフェース (SPI) コマンド **INQUIRE WORKREQUEST** および **SET WORKREQUEST** を使用すると、以下のことが可能です。

- 単一の要求に関連したトランザクションを判別する
- 単一の要求に関連したすべてのトランザクションを相互に関連付ける
- 選択された作業要求をパージする

各要求は次のものを示します。

- ローカル・タスク番号とトランザクション ID
- 要求のタイプ。第一にサポートされるタイプは IIOP です。
- フィルターなどとしてコマンドに入力できる固有の (印刷可能な) ストリング
  - Worktype
  - ClientIPAddress

– ターゲット SNA (z/OS Communications Server) のアプリケーション ID または TCPIP アドレス

これらのコマンドについて詳しくは、「*CICS System Programming Reference*」および「*CICS Supplied Transactions*」マニュアルを参照してください。

INQUIRE および SET WORKREQUEST コマンドは、IIOP タスクのみに使用可能です。

RequestReceiver に関連した WorkRequest は含まれません。それらは軽量であり、この情報はすべて RequestProcessor で入手可能です。RequestReceiver は、インスタンスごとに複数の要求を処理でき、要求が完了するずっと前にシステムを離れた可能性があります。

CPSM WUI を使用して論理サーバーに問い合わせる場合、単一の画面にサーバー内のすべての WorkRequest が表示されます。

CEMT INQ TASK リストからタスクをパージするのと同じような方法で、これらのコマンドを使用して RequestProcessor をパージできます。

エンタープライズ Bean のセキュリティーの管理

エンタープライズ Bean では、セキュリティー・メカニズムとして Java2 セキュリティー、Secure Sockets Layer (SSL) セキュリティー、MRO セキュリティー、およびセキュリティー役割を使用できます。

これらのセキュリティーの任意の組み合わせを実装できます。

Java セキュリティー
この形式のセキュリティー管理は、Java 仮想マシン (JVM) によって実装され、JVM 制御の下で実行される任意の Java プログラムで使用できます。このタイプのセキュリティー管理のセットアップ方法については、100 ページの『Java セキュリティー・マネージャーの有効化』を参照してください。

Secure Sockets Layer (SSL) セキュリティー
Secure Sockets Layer (SSL) は、TCP/IP を使用して通信するクライアントとサーバー間でプライバシーと認証を提供するセキュリティー・プロトコルです。SSL について詳しくは、「RACF Security Guide」の『Support for security protocols』を参照してください。

MRO セキュリティー
要求受信側が、要求に関連付けられる CICS USERID を設定した後、アプリケーション専有領域 (AOR) に転送されなければならない場合があります。ルーティング・メカニズムが複数領域操作 (MRO) 接続を使用する場合、ユーザー ID の送信は MRO セキュリティー規則に従います。Link security with MRO を参照してください。

セキュリティー役割
セキュリティー役割は、ユーザーが正常にアプリケーションを使用するのに必要な権限の観点からみたときの、アプリケーションのユーザーのタイプを表します。400 ページの『セキュリティー役割』を参照してください。

CICS 提供のエンタープライズ Bean ポリシー・ファイル

CICS 提供のエンタープライズ Bean ポリシー・ファイル `dfjejbpl.policy` は、Java セキュリティー・ポリシー・メカニズムに基づいています。

Java セキュリティー・ポリシー・メカニズムは、「*Enterprise JavaBeans Specification, Version 1.1*」で説明されています。サンプル・ポリシー・ファイルが、図 31 に示されています。

Java において、セキュリティー・ポリシーは、権限をコード・ソースにマップする保護ドメインの形で定義されます。保護ドメインには、1 組の関連した権限を持つコード・ソースが入っています。

CICS 提供のエンタープライズ Bean ポリシー・ファイルは、次のことを行う 2 つの保護ドメインを定義します。

1. CICS エンタープライズ Bean の Container コード・ソースに実行のための必要な権限を付与します。図 31 の「grant codeBase」ブロックを参照してください。
2. *Enterprise JavaBeans* 仕様のバージョン 1 で概要が示されている権限のみを任意のコード・ソースに付与します。図 31 で、デフォルトの「grant」ブロックを参照してください。
   - だれでも印刷ジョブ要求を開始できるようにします。
   - 任意の TCP/IP ポートでアウトバウンド接続を可能にします。
   - すべてのシステム・プロパティーを読み取れるようにします。

エンタープライズ Bean から JDBC または SQLJ を使用したい場合は、CICS 提供のエンタープライズ Bean ポリシー・ファイルを修正して、JDBC ドライバーに権限を付与することに注意してください。詳しくは、「DB2 Guide」の『Using JDBC and SQLJ to access DB2 data from Java programs』を参照してください。

```
// permissions granted to CICS enterprise beans Container codesource protection
//domain
   grant codeBase "file:usr/lpp/cicsts/cicsts42//-" {
     permission java.security.AllPermission;
   };

// default EJB 1.1 permissions granted to all protection domains
   grant {
     // allows anyone to initiate a print job request
     permission java.lang.RuntimePermission "queuePrintJob";

     // allows outbound connection on any TCP/IP ports
     permission java.net.SocketPermission "*:0-65535", "connect";

     // allows anyone to read properties
     permission java.util.PropertyPermission "*", "read";
   };
```

図 31. CICS エンタープライズ Bean セキュリティー・ポリシーの例

エンタープライズ Bean セキュリティーの使用

EJB 1.1 仕様は、以下のセキュリティー API を定義して、エンタープライズ Bean が呼び出し元のセキュリティー詳細に基づいてアプリケーションの決定を行うことを可能にします。

java.security.Principal getCallerPrincipal()
このメソッドは、現行の Bean メソッドをだれが呼び出したかを判別するのに使用されます。getCallerPrincipal メソッドは、CICS で完全にサポートされています。現行呼び出し元の ID の判別方法の詳細は、399 ページの『識別名の派生』に示されています。

boolean isCallerInRole(String SecurityRoleReference)
このメソッドは、現行の呼び出し元に、メソッド呼び出しで指定されるセキュリティー役割参照にリンクされるセキュリティー役割が割り当てられているかどうかのテストに使用されます。

次の推奨されない EJB 1.0 セキュリティー API が使用される場合、CICS は、(EJB 1.1 仕様に準拠する) 実行時例外をスローします。

- java.security.Identity getCallerIdentity()
- boolean isCallerInRole(java.security.Identity role)

注: Enterprise JavaBeans(EJB) 1.0 仕様に合わせて作成されたエンタープライズ Bean は、提供された開発ツールを使用して、Enterprise JavaBeans 1.1 仕様レベルにアップグレードする必要があります。

- デプロイメント・ツールについては、351 ページの『CICS システムでのエンタープライズ Bean 用のデプロイメント・ツール』を参照してください。
- エンタープライズ Bean の作成については、336 ページの『エンタープライズ Bean の作成』を参照してください。

エンタープライズ Bean のファイル・アクセス権限の定義:

CICS でエンタープライズ Bean を正常に実行するには、CICS 領域のユーザー ID に、エンタープライズ・ロジックで使用されるファイルへのアクセスを許可する必要があります。

実装されるセキュリティーのレベルに関係なく、これらのファイル権限は、エンタープライズ Bean の実行に必要です。「*CICS Transaction Server for z/OS インストール・ガイド*」も参照してください。

エンタープライズ Bean で使用される z/OS UNIX ファイルへのアクセス:

以下のファイル権限が、エンタープライズ Bean の実行に必要です。

表 22. CICS エンタープライズ Bean に必要なファイル・アクセス権限

ファイル/ディレクトリー構造	最小限の権限	コメント
CORBASERVER シェルフ・ディレクトリー (例: /var/cicsts/)	読み取り、書き込みおよび実行	シェルフには、CORBASERVER および DJAR インストール時にアクセスされ、各 CICS は固有のサブディレクトリーを作成する必要があります (注 1 を参照)。

表 22. *CICS エンタープライズ Bean に必要なファイル・アクセス権限 (続き)*

ファイル/ディレクトリー構造	最小限の権限	コメント
/usr/lpp/cicsts/cicsts42 ディレクトリー構造とクラス	読み取りおよび実行	CICS 提供の Java コードが入っています (注 2 を参照)。
/usr/lpp/java/J6.0.1_64/bin および /usr/lpp/java/J6.0.1_64/bin/classic ディレクトリー	読み取りおよび実行	IBM JVM コードが入っています (注 3 を参照)。
CICS 作業ディレクトリー	読み取り、書き込みおよび実行	stdin ファイルの作成に使用されます (注 4 を参照)
デプロイ済み JAR ファイル	Read	デプロイメント・プロセスによって DJAR インストール時に使用されます。
セキュリティー・ポリシー・ファイル (必要な場合)	Read (読み取り)	**-Djava.security.policy** プロパティーが JVM システム・プロパティー・ファイルで指定される場合に必要です。
システム・プロパティー・ファイル	Read (読み取り)	JVM を作成する場合のオプション (注 5 を参照)。

注:

1. /var/cicsts/ は、CORBASERVER リソース定義を定義するときのデフォルト SHELF ディレクトリー名です。各 CICS 領域は、リソース定義のインストール時にこのシェルフで固有のサブディレクトリーを作成します。
2. *cicsts42* は、CICS TS のインストール時に定義した USSDIR インストール・パラメーターに選択した値です。
3. java/J6.0.1_64 は、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション のインストール場所です。
4. CICS 作業ディレクトリーは、JVM プロファイルの WORK_DIR パラメーターで定義されます。
5. オプションのシステム・プロパティー・ディレクトリーとファイル名は、JVM プロファイルの JVMPROPS オプションで指定されます。

ファイル所有権と権限は、**chmod** および **chown** コマンドを使用して定義できます。詳しくは、「*z/OS UNIX System Services Command Reference*」を参照してください。

エンタープライズ Bean で使用されるデータ・セットへのアクセス:

CORBASERVER を CICS 領域にインストールする前に、次の 2 つのデータ・セットが UPDATE アクセス権限を使用して作成され、CICS に対して定義され、インストールされなければなりません。これらのファイルは、VSAM データ・セットまたはカップリング・ファシリティー・データ・テーブルの場合があります。

399 ページの図 32 は、必要な許可でデータ・セットにアクセスするための RACF コマンドの例を示しています。

注: これらのファイルは内部で CICS によって使用されるので、それらのファイルへのリソース・レベルのセキュリティー・アクセス権限をユーザーに付与する必要はありません。これにより、VSAM アプリケーションがこれらのファイル内のデータにアクセスしません。

DFHEJDIR

このデータ・セットには、CICS IIOP サーバーを構成するリスナー領域と AOR で共用される要求ストリーム・ディレクトリーが入っています。ファイルはリカバリー可能でなければなりません。

DFHEJOS

DFHEJOS は、不動態化されたステートフル・セッション Bean が入っているデータ・セットです。CICS IIOP サーバーを構成するすべての AOR で共用されます。このファイルはリカバリー可能である必要はありません。

```
ADDSD  'CICSTS42.CICS.CICS.DFHEJDIR' NOTIFY(cics_sys_admin_id) UACC(NONE)
PERMIT 'CICSTS42.CICS.CICS.DFHEJDIR' ID(cics_id1,...,cics_group1,..,cics_groupn)
       ACCESS(UPDATE)
ADDSD  'CICSTS42.CICS.CICS.DFHEJOS'  NOTIFY(cics_sys_admin_id) UACC(NONE)
PERMIT 'CICSTS42.CICS.CICS.DFHEJOS'  ID(cics_id1,...,cics_group1,..,cics_groupn)
       ACCESS(UPDATE)
```

図 32. CICS データ・セットへのアクセスを許可するのに使用される RACF コマンドの例

CICS データ・セットへのアクセスの許可について詳しくは、「*CICS RACF Security Guide*」の Authorizing access to CICS data sets を参照してください。

識別名の派生:

エンタープライズ Bean は、*Principal* オブジェクトを使用して、エンド・ユーザーすなわちクライアントを識別できます。

getCallerPrincipal メソッドは、クライアントを表す Principal オブジェクトを戻します。その Principal オブジェクトには、クライアントに関する情報を戻すために呼び出すことができるメソッドが入っています。特に、Principal オブジェクトの getName メソッドは、クライアントの「識別名」が入っている String を戻します。識別名すなわち DN は、相対識別名すなわち RDN と呼ばれる一連のキーワードと値のペアであり、X.500 勧告 (標準 ISO/IEC 9594) に含まれます。識別名のストリング表記は、RFC2253、*LDAP V3: UTF-8 String Representation of Distinguished Names* によって推奨されています。

注: CICS Transaction Server for z/OS, バージョン 4 リリース 2 は、ステートフル・セッション Bean インスタンスが、それを作成したのと同じプリンシパルによってのみ使用されることを検証しません。したがって、プリンシパルのユーザー ID と識別名は、Bean インスタンスが再活動化された後で異なる場合があります。

Bean のクライアントが、Secure Sockets Layer プロトコルを使用するクライアント証明書を使用して識別され、認証された場合、識別名は常にその証明書から取得されます。しかし、Bean のクライアントが証明書を提供しなかった場合、識別名は、DFHEJDNX ユーザー置換可能モジュールを呼び出すことによって取得されます。DFHEJDNX モジュールへの入力は、CORBASERVER 定義の CERTIFICATE オプションでラベルが指定されるサーバー証明書から得られる表題、組織単位、組織、市区町村、都道府県、および国です。さらに、Bean を実行するユーザーのユーザー

ID とそのユーザー ID に関連した共通名です。ただし、SEC=NO が指定される場合、CICS 領域のユーザー ID が使用されます。共通名は、そのユーザーのユーザー名を大/小文字混合のストリングに変換することによって得られます。証明書ラベルは、KEYRING システム初期設定パラメーターで識別される鍵リング内の証明書を指定します。CERTIFICATE オプションが省略される場合、鍵リング内のデフォルトの証明書から情報が得られます。KEYRING パラメーターが省略される場合、証明書情報は DFHEJDNX に渡されず、共通名 RDN のみが使用可能です。

CICS 提供バージョンの DFHEJDNX は、CORBASERVER 証明書とユーザー名から得られる入力を受け入れ、次のスタイルでそれらの入力を識別名にフォーマットします。

T=CICS EJB Container,CN=Louise Peters,OU=CICS/390 Development, O=IBM,L=Hursley,ST=Hampshire,C=GB

DFHEJDNX の CICS 提供サンプルは、次のように SDFHSAMP ライブラリー *CICSTS42.CICS.CICS.SDFHSAMP* に置かれます。

- DFHEJDN1 (アセンブラー言語の場合)
- DFHEJDN2 (C 言語の場合)

セキュリティー役割

エンタープライズ Bean メソッドへのアクセスは、セキュリティー役割という概念に基づいて行われます。セキュリティー役割は、ユーザーが正常にアプリケーションを使用するのに必要な権限の観点からみたときの、アプリケーションのユーザーのタイプを表します。

例えば、給与計算アプリケーションでは次のとおりです。

- `manager` 役割は、アプリケーションのすべての部分の使用を許可されるユーザーを表すことができます。
- `team_leader` 役割は、アプリケーションの管理機能の使用を許可されるユーザーを表すことができます。
- `data_entry` 役割は、アプリケーションのデータ入力機能の使用を許可されるユーザーを表すことができます。

アプリケーションのセキュリティー役割は、アプリケーション・アセンブラーによって定義され、Bean のデプロイメント記述子で指定されます。詳しくは、405 ページの『デプロイメント記述子内のセキュリティー役割』を参照してください。

また、Bean メソッドの実行が許可されるセキュリティー役割も、アプリケーション・アセンブラーによって Bean のデプロイメント記述子で指定されます。この例では、従業員が毎週勤務した時間数を更新するメソッドは、`data_entry` 役割に割り当てられます。一方、給与計算から従業員を削除するメソッドは、`team_leader` 役割に割り当てられます。

異なるアプリケーションまたは異なるシステムで同じような名前を持つセキュリティー役割を区別するには、CICS システムに Bean がデプロイされるときに、Bean のデプロイメント記述子で指定されるセキュリティー役割に、1 部構成または 2 部構成の修飾子を指定できます。例えば、次のとおりです。

- 修飾子のないセキュリティー役割:

team_leader

- 1 つの修飾子があるセキュリティー役割:

  payroll.team_leader

- 2 つの修飾子があるセキュリティー役割:

  test.payroll.team_leader

修飾子を持つセキュリティー役割は、**デプロイ済みセキュリティー役割**と呼ばれます。詳細については、『デプロイ済みセキュリティー役割』を参照してください。

個々のユーザーへのセキュリティー役割のマッピングは、外部セキュリティー・マネージャーで行われます。このマッピングは必ずしも 1 対 1 である必要はありません。例えば、複数のユーザーを data_entry 役割に割り当てることができ、team_leader 役割と data_entry 役割の両方に一部のユーザーを割り当てることができます。詳細については、407 ページの『セキュリティー役割の実装』を参照してください。

デプロイメント記述子内のセキュリティー役割と表示名には、任意の ASCII または Unicode 文字を含むことができます。RACF で使用される名前の場合はそうではありません。この名前は EBCDIC コード・ページ 037 の文字に制限されます。さらに、RACF コマンドで使用される場合、一部の文字 (例えば、アスタリスク (*)) には特別な意味があります。したがって、CICS がそのコンポーネントからデプロイ済みセキュリティー役割を作成する場合、文字によっては別の文字で置き換えられたり、エスケープ・シーケンスで置き換えられるものがあります。詳しくは、403 ページの『デプロイ済みセキュリティー役割の文字置換』を参照してください。

デプロイ済みセキュリティー役割:

Bean のデプロイメント記述子で指定されるセキュリティー役割と個々のユーザーとの間の直接のマッピングでは、Bean メソッドへのアクセスを十分に制御しない場合があります。

次に例を示します。

- 異なるサプライヤーによって提供される 2 つのアプリケーションが、セキュリティー役割に似た名前を使用する場合があります。企業では、各アプリケーションのユーザーが異なります。
- 1 つの Bean を複数のアプリケーションで使用する場合があります。ユーザーは、一方のアプリケーションで特定のメソッドを使用する資格が与えられ、もう一方のアプリケーションでは与えられません。
- 1 つのアプリケーションがテスト・システムと実動システムにデプロイされる場合があります。テスト部門のメンバーは、テスト・システム内のすべての Bean メソッドの使用が許可されますが、実動システムでは許可されません。

こうした事例で必要な程度の制御を提供するために、アプリケーション・レベルとシステム・レベルでセキュリティー役割を修飾することができます。修飾子を持つセキュリティー役割は、**デプロイ済みセキュリティー役割**と呼ばれます。両方のレベルで修飾される役割名の例は次のとおりです。

test.payroll.team_leader

- payroll は、アプリケーション・レベルでセキュリティー役割を修飾します。給与計算アプリケーションの team_leader 役割と他のアプリケーションの team_leader 役割との区別に使用されます。
- test は、システム・レベルでセキュリティー役割を修飾します。テスト・システムの payroll.team_leader 役割と他のシステムの payroll.team_leader 役割との区別に使用されます。

アプリケーション・レベルでは、デプロイメント記述子で指定される場合、セキュリティー役割は**表示名**によって修飾されます。表示名が指定されない場合、セキュリティー役割はアプリケーション・レベルで修飾されません。アプリケーション・レベルの修飾子が使用される場合、ピリオド (.) が区切り文字として使用されます。修飾子が使用されない場合、区切り文字はありません。

システム・レベルでは、オプションとしてセキュリティー役割が、EJBROLEPRFX システム初期設定パラメーターで指定される接頭部で修飾されます。EJBROLEPRFX が指定されない場合、セキュリティー役割はシステム・レベルで修飾されません。システム・レベルの修飾子が使用される場合、ピリオド (.) が区切り文字として使用されます。修飾子が使用されない場合、区切り文字はありません。

次の例は、Bean のデプロイメント記述子で定義されるセキュリティー役割の修飾方法を示しています。

- Bean には、3 つのセキュリティー役割 manager、team_leader、および data_entry が含まれます。
- この Bean は、表示名 payroll を持つ給与計算アプリケーションで使用されます。また、Bean は、表示名のないテスト・アプリケーションにも含まれます。
- 給与計算アプリケーションは 2 つの実動システムで使用されます。最初の実動システムは接頭部を指定しませんが、2 つ目の実動システムは接頭部 executive を指定します。
- テスト・アプリケーションは、接頭部 test1 を持つテスト・システムで使用されます。

2 つのレベルの修飾が、デプロイメント記述子で指定されるセキュリティー役割に適用される場合、デプロイ済みセキュリティー役割は次のとおりです。

```
payroll.manager        executive.payroll.manager        test1.manager
payroll.team_leader    executive.payroll.team_leader    test1.team_leader
payroll.data_entry     executive.payroll.data_entry     test1.data_entry
```

企業のセキュリティーのニーズに対応するために、これらのデプロイ済み役割のそれぞれを、個々のユーザー (またはユーザーのグループ) にマップできます。

セキュリティー役割がアプリケーション・レベルまたはシステム・レベルで修飾されない場合、デプロイ済みセキュリティー役割は、デプロイメント記述子で定義されるセキュリティー役割と同じです。例えば、上記の例の Bean が、表示名のないアプリケーションで使用されるときに、そのアプリケーションが、EJBROLEPRFX を指定しないシステムで使用される場合、デプロイ済みセキュリティー役割は次のとおりです。

```
manager
team_leader
data_entry
```

セキュリティー役割に対するサポートの有効化と無効化:

デフォルトで、セキュリティー役割に対する CICS サポートは有効です。

XEJB システム初期設定パラメーターを使用すると、セキュリティー役割に対するサポートを無効化 (または明示的に有効化) することができます。このサポートを無効にする場合は、次のようになります。

- CICS は、メソッドの許可検査を実行しません。すべてのユーザーが、すべての Bean メソッドを使用できます。
- isCallerInRole() メソッドは、すべてのユーザーについて `true` を戻します。

セキュリティー役割参照:

アプリケーション内で、isCallerInRole() メソッドを使用すると、アプリケーションのユーザーが所定の役割に対して定義されているかどうかを判別できます。

このメソッドは、セキュリティー役割ではなく、**セキュリティー役割参照**を引数として取ります。Bean でコーディングされるセキュリティー役割参照は、Bean プロバイダーによって定義され、Bean のデプロイメント記述子で宣言されます。

詳しくは、 405 ページの『デプロイメント記述子内のセキュリティー役割』を参照してください。

各セキュリティー役割参照は、アプリケーション・アセンブラーによってセキュリティー役割にリンクされます。このリンクは、Bean のデプロイメント記述子で宣言されます。例えば、Bean のコード内で使用される administrator のセキュリティー役割参照は、デプロイメント記述子で、team_leader 役割にリンクされます。

詳しくは、 405 ページの『デプロイメント記述子内のセキュリティー役割』を参照してください。

デプロイ済みセキュリティー役割の文字置換:

デプロイメント記述子内のセキュリティー役割と表示名には、任意の ASCII または Unicode 文字を含むことができます。

デプロイ済みセキュリティー役割で使用できる文字セットは、さらに制限されます。

- RACF で使用されるプロファイル名は、EBCDIC コード・ページ 037 の文字に制限されます。
- RACF コマンドで使用される場合、一部の文字 (例えば、アスタリスク (*)) には特別な意味があり、プロファイル名で使用できません。

セキュリティー役割および表示名内の Unicode 文字をデプロイ済みセキュリティー役割で直接使用できない場合、それらの文字は、 404 ページの表 23 に示されているエスケープ・シーケンスで置き換えられます。 置換が生じるのは次の場合です。

- EJBROLE 生成ユーティリティー (`dfhreg`) が、デプロイメント記述子を処理して RACF コマンドを生成する場合
- CICS がセキュリティー役割を RACF ユーザー ID にマップする場合

表 23. セキュリティー役割で使用されるエスケープ・シーケンス

文字	説明	ASCII/Unicode	EBCDIC コード・ページ 037	エスケープ・シーケンス
デプロイ済みセキュリティー役割で使用できない等価の EBCDIC 値を持つ ASCII 値と Unicode 値は、次のように 3 文字のエスケープ・シーケンスで置き換えられます。				
	ブランク	X'20'	X'40'	¢
¢	セント	X'A2'	X'4A'	\A2
\	円記号	X'5C'	X'E0'	\5C
*	アスタリスク	X'2A'	X'5C'	\2A
&	アンパーサンド	X'26'	X'50'	\26
%	パーセント	X'25'	X'6C'	\25
,	コンマ	X'2C'	X'6B'	\2C
(	左括弧	X'28'	X'4D'	\28
)	右括弧	X'29'	X'5D'	\29
;	セミコロン	X'3B'	X'5E'	\3B
EBCDIC コード・ページ 037 に等価のものがない Unicode 値は、Unicode エスケープ・シーケンスで置き換えられます。Unicode 表記 X'yyyy' を持つ文字は、\uyyyy で置き換えられます。例えば、次のとおりです。				
€	ユーロ記号	X'20AC'	サポートされない	\u20AC
	ひらがなの「き」	X'304D'	サポートされない	\u304D
α	アルファ	X'03B1'	サポートされない	\u03B1

文字の置き換え方法を示す 2 つの例を以下に挙げます。

例 1

- EJBROLEPRFX には値 `test` があります。
- デプロイメント記述子内の表示名の値は `year.end.processing` です。
- デプロイメント記述子内のセキュリティー役割の値は `auditor 1` です。

この例では、デプロイ済みセキュリティー役割が作成されるときに、次のようになります。

1. 各スペースは ¢ で置き換えられます。
2. デプロイ済みセキュリティー役割は、EJBROLEPRFX 値、表示名、およびセキュリティー役割から成ります。ピリオドが区切り文字として使用されます。

その結果生成されるデプロイ済みセキュリティー役割は次のとおりです。

```
test.year.end.processing.auditor¢1
```

例 2

- EJBROLEPRFX には値 `test` があります。
- デプロイメント記述子内の表示名の値は *αβ32* です。Unicode エンコードは X'03B1 03B2 0033 0034' です。

- デプロイメント記述子内のセキュリティー役割の値は auditor 1 です。

この例では、デプロイ済みセキュリティー役割が作成されるときに、次のようになります。

1. EBCDIC コード・ページ 037 に等価のものがある各 Unicode 文字は、それに応じて置き換えられます。表示名では、X'0033 0034' が 34 で置き換えられます。
2. EBCDIC コード・ページ 037 に等価のものが**ない** 各 Unicode 文字は、対応するエスケープ・シーケンスで置き換えられます。表示名では、X'03B1 03B2' が \u03B1\u03B2 で置き換えられます。
3. 各スペースは ¢ で置き換えられます。
4. デプロイ済みセキュリティー役割は、EJBROLEPRFX 値、表示名、およびセキュリティー役割から成ります。ピリオドが区切り文字として使用されます。

その結果生成されるデプロイ済みセキュリティー役割は次のとおりです。

```
test.\u03B1\u03B234.auditor¢1
```

デプロイメント記述子内のセキュリティー役割:

ここでは、デプロイメント記述子のフラグメントを示します。次のセキュリティー役割情報が含まれます。

- 1 payroll という表示名。
- 2 team_leader 役割にリンクされるセキュリティー役割参照 administrator。
- 3 team_leader というセキュリティー役割。
- 4 team_leader 役割で定義されるユーザーが create() メソッドを呼び出すことを許可するメソッド・アクセス権。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN"
"http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
  <ejb-jar id="ejb-jar_ID">
    <display-name>payroll</display-name>         1
      <enterprise-beans>
        <session id="Session_1">
          .
          .
          <security-role-ref id="SecurityRoleRef_1">
            <role-name>administrator</role-name> 2
            <role-link>team_leader</role-link>
          </security-role-ref>
          .
          .
        </session>
      </enterprise-beans>
      <assembly-descriptor id="AssemblyDescriptor_1">
        <security-role id="SecurityRole_1">
          <role-name>team_leader</role-name>     3
        </security-role>
        .
        .
        <method-permission id="MethodPermission_1">
          <description>team_leader:+:</description>
          <role-name>team_leader</role-name>     4
          <method id="MethodElement_01">
            <ejb-name>Managed</ejb-name>
            <method-intf>Home</method-intf>
            <method-name>create</method-name>
            <method-params>
            </method-params>
          </method>
           .
           .
        </method-permission>
         .
         .
      </assembly-descriptor>
       .
       .
     </ejb-jar>
```

図 33. セキュリティー役割を含むデプロイメント記述子の例

このデプロイメント記述子を持つアプリケーションが、次のシステム初期設定パラメーターを持つ CICS システムで使用される場合は、以下のとおりです。

```
SEC=YES
XEJB=YES
EJBROLEPRFX='test'
```

- デプロイ済みセキュリティー役割 `test.payroll.team_leader` が、RACF に対して定義されなければなりません。
- そのデプロイ済みセキュリティー役割への READ アクセス権限を持つユーザーは、create() メソッドを呼び出すことが許可されます。
- `isCallerInRole('administrator')` は、デプロイ済みセキュリティー役割 `test.payroll.team_leader` で定義されるユーザーについて `true` を戻し、それ以外のユーザーについて `false` を戻します。

デプロイメント記述子の内容について詳しくは、「*Enterprise JavaBeans Specification, Version 1.1*」を参照してください。

デプロイメント記述子の内容を表示するには、Assembly Toolkit (ATK) を使用できます。ATK について詳しくは、「*CICS Operations and Utilities Guide*」の The enterprise bean deployment tool, ATK を参照してください。

セキュリティー役割の実装

エンタープライズ Bean メソッドへのアクセスは、**セキュリティー役割**という概念に基づいて行われます。

このタスクについて

これについては、400 ページの『セキュリティー役割』を参照してください。

CICS エンタープライズ Bean 環境でセキュリティー役割の使用を実装するには、以下を行う必要があります。

1. アプリケーションのデプロイメント記述子で定義されるセキュリティー役割を判別します。
2. アプリケーションのデプロイメント記述子内のセキュリティー役割に関連した表示名を判別します。表示名は、アプリケーション・レベルでセキュリティー役割を修飾します。
3. システム・レベルでセキュリティー役割名を修飾する必要があるかどうか、および必要がある場合は、アプリケーションが実行される各システムで使用する接頭部の値を決定します。
4. ステップ 1 から 3 で収集された情報を使用して、各システム内のアプリケーションで使用されるデプロイ済みセキュリティー役割の名前を判別します。セキュリティー役割と表示名の文字で、EBCDIC コード・ページ 37 に直接等価なものがない文字 (およびその他のいくつかの文字) は、デプロイ済みセキュリティー役割の作成時に別の文字またはエスケープ・シーケンスで置き換えられなければなりません。詳しくは、403 ページの『デプロイ済みセキュリティー役割の文字置換』を参照してください。
5. ステップ 1 から 3 で収集された情報を使用して、デプロイ済みセキュリティー役割の RACF プロファイルを定義します。詳しくは、408 ページの『RACF に対するセキュリティー役割の定義』 を参照してください。
6. 個々のユーザーまたはユーザーのグループを RACF 内の各デプロイ済みセキュリティー役割に関連付けます。詳しくは、408 ページの『RACF に対するセキュリティー役割の定義』 を参照してください。
7. 次のシステム初期設定パラメーターを指定します。
   - `SEC=YES`
   - `XEJB=YES`。これはデフォルト値であるため、明示的に指定する必要はありません。
8. デプロイ済みセキュリティー役割にシステム・レベルの修飾子が含まれているシステムの場合 (ステップ 3 を参照)、EJBROLEPRFXEJBROLEPRFX システム初期設定パラメーターを指定します。

RACF EJBROLE 生成ユーティリティーの使用:

RACF EJBROLE 生成ユーティリティー dfhreg は Java アプリケーション・プログラムであり、デプロイメント記述子からセキュリティー役割情報を抜き出し、RACF に対してセキュリティー役割を定義するのに使用できる REXX プログラムを生成します。

dfhreg が生成する REXX プログラムには、セキュリティー役割を GEJBROLE クラスでプロファイルのメンバーとして定義する RACF コマンドが入っています。REXX プログラムを実行する前に、これを修正して、定義されるプロファイルの名前を変更してください。

z/OS UNIX 用の dfhreg 呼び出しスクリプト (dfhreg) および Windows 用の dfhreg 呼び出しスクリプト (dfhreg.bat) は、$CICS_HOME/lib/security ディレクトリーにあります。 dfhreg の実装 (dfhreg.jar) もこのディレクトリーにあります。dfhreg の実行に必要なその他の JAR ファイル (dfjcsi.jar、dfjejbdd.jar、および dfjorb.jar) は、$CICS_HOME/lib ディレクトリーにあります。$CICS_HOME は、CICS の USS コンポーネントをインストールした z/OS UNIX ディレクトリーです。

dfhreg は、Java をサポートする任意のプラットフォームで実行できます。ただし、結果の REXX プログラムは、セキュリティー役割を定義する z/OS システム上の RACF データベースに対して実行する必要があります。dfhreg を実行する場合は、次の要件を満たす必要があります。

1. クラスパスに次の JAR ファイルが入っていなければなりません。

   ```
   dfhreg.jar
   dfjcsi.jar
   dfjejbdd.jar
   dfjorb.jar
   ```

2. サポートされているバージョンの Java 2 SDK を使用しなければなりません。

ユーティリティーが生成する REXX プログラムは、そのユーティリティーを実行するプラットフォームのコード・ページ内にあります。ASCII コード・ページを使用するプラットフォームでこのユーティリティーを実行する場合は、REXX プログラムを、ターゲット z/OS システムで使用される EBCDIC コード・ページに変換する必要があります。

RACF に対するセキュリティー役割の定義:

RACF では、デプロイ済みセキュリティー役割は一般リソースとして管理されます。デプロイ済みセキュリティー役割を定義するには、適切なアクセス・リストを使用して、GEJBROLE または EJBROLE リソース・クラスでプロファイルを定義してください。

例えば、次のコマンドを使用して、デプロイ済みセキュリティー役割 deployed_security_role_1 および deployed_securityrole_2 を、GEJBROLE クラスの securityrole_group プロファイルのメンバーとして定義し、user1 および user2 に READ アクセス権限を与えます。

```
RDEFINE GEJBROLE securityrole_group UACC(NONE)
                ADDMEM(deployed_security_role_1, deployed_securityrole_2, ...)
                NOTIFY(sys_admin_userid)
PERMIT securityrole_group CLASS(GEJBROLE) ID(user1, user2) ACCESS(READ)
```

または、次のコマンドを使用して、EJBROLE クラスでデプロイ済みセキュリティー役割を定義し、各デプロイ済みセキュリティー役割への READ アクセス権限をユーザーに与えます。

```
RDEFINE EJBROLE (deployed_security_role1, deployed_security_role2, ...) UACC(NONE)
                NOTIFY(sys_admin_userid)
PERMIT deployed_security_role1 CLASS(EJBROLE) ID(user1, user2) ACCESS(READ)
PERMIT deployed_security_role2 CLASS(EJBROLE) ID(user1, user2) ACCESS(READ)
```

注:

1. 指定するセキュリティー役割はデプロイ済みセキュリティー役割であり、デプロイメント記述子で定義される非修飾のセキュリティー役割ではありません。
2. Bean メソッドを実行するか、isCallerInRole() メソッドから `true` 応答を受け取るには、ユーザーには READ アクセス権限が必要です。

エンタープライズ Bean での CICSPlex SM

エンタープライズ Bean の管理は、CICSplex 全体のレベルで行うことができます。

エンタープライズ Bean に対する CICSPlex SM サポート

エンタープライズ Bean の管理は、CICSPlex SM の Operator および API サービスを使用して、CICSplex 全体のレベルで行うことができます。

Enterprise JavaBeans のサポートのために CICSPlex SM が提供する機能には、次のものがあります。

- CorbaServer および DJAR 定義のオブジェクト管理
- インストールされた CorbaServer および DJAR インスタンスのオブジェクト管理
- エンタープライズ Bean 実行の動的管理

これらの機能を扱う CICSPlex SM 領域は次のとおりです。

- アプリケーション・プログラミング・インターフェース (API)。**EXEC CPSM** インターフェースを使用してエンタープライズ Bean オブジェクトの定義、照会、および管理を可能にします。詳しくは、「*CICSPlex System Manager Application Programming Guide*」を参照してください。
- Web ユーザー・インターフェース。Web ブラウザーを使用してエンタープライズ Bean オブジェクトの照会と管理を可能にします。Web ユーザー・インターフェースについては、「*CICSPlex System Manager Web User Interface Guide*」を参照してください。

エンタープライズ Bean に対する CICSPlex SM 定義サポート

ビジネス・アプリケーション・サービス (BAS) は、CICS リソースの定義とインストールに関係する CPSM コンポーネントです。

BAS について詳しくは、「*CICSPlex System Manager Managing Business Applications*」マニュアルを参照してください。Enterprise JavaBeans に固有の BAS オブジェクトは次のとおりです。

- EJCODEF — エンタープライズ Bean CorbaServer 定義
- EJDJDEF — エンタープライズ Bean CICS デプロイ済み JAR ファイル定義

CorbaServer 定義オブジェクト (EJCODEF) により、CEDA バージョンとまったく同じ CorbaServer 特性を指定できます。EJCODEF については、「*CICSPlex System Manager Managing Business Applications*」マニュアルの Defining CorbaServers using BAS で説明しています。

CICS デプロイ済み JAR ファイル定義オブジェクト (EJDJDEF) により、CEDA バージョンとまったく同じ DJAR 特性を指定できます。EJDJDEF については、「*CICSPlex System Manager Managing Business Applications*」マニュアルの Defining a CICS-deployed JAR file using BAS で説明しています。

これらのリソースは、標準の BAS 機能に完全に統合され、自動的に、またはユーザーの必要に応じて随時に管理し、インストールすることができます。

これらの 2 つのオブジェクト・タイプに加えて、エンタープライズ Bean のオペレーションに関連したその他の BAS オブジェクトがあります。

- TCPDEF—TCPIPSERVICE 定義
- RQMDEF—REQUESTMODEL 定義
- TRANDEF—CICS TRANSACTION 定義
- PROGDEF—PROGRAM 定義

クライアントからのエンタープライズ Bean 実行要求は、TCP/IP ポートを通じて CICS リスナー領域に到達します。BAS を使用する場合、このポートの番号が、これらの呼び出しに応答することが期待されるすべてのリスナー領域にインストールされる、TCPDEF オブジェクトを通じて指定されなければなりません。TCPDEF の内容は、CEDA TCPIPSERVICE 定義に指定されたものをミラーリングする必要があります。詳しくは、443 ページの『IIOP 用の TCP/IP のセットアップ』を参照してください。

特定のエンタープライズ Bean に対する実行要求が認識され、一般的なエンタープライズ Bean 実行の場合とは異なる方法で管理されることをユーザーが要求する場合、要求モデルを使用して、ユーザー指定のトランザクション・コードに関連付けることができます。CICSPlex SM 内で、要求モデルは、RQMDEF オブジェクトを通じて定義され、このような要求が代行受信を必要とするすべてのリスナー領域にインストールされなければなりません。エンタープライズ Bean の複雑さによっては、さらに要求モデルを関連した AOR にインストールすることが必要な場合があります。これらの RQMDEF の内容は、CEDA REQUESTMODEL 定義に指定されたものをミラーリングする必要があります。詳しくは、455 ページの『CICS TRANSID の取得』を参照してください。

分散エンタープライズ Bean 処理環境では、特定の CICS 領域が、IIOP 実行要求を受け取るリスナーの役目をし、その他の領域は、必要なエンタープライズ Bean の実行用に実際の EJB 環境を提供する AOR の役目をすることが期待されます。CICSPlex SM TRANDEF オブジェクトは、ここで利用する特にパワフルなツールです。単一の BAS リソース割り当て (RASGNDEF) を使用して、単一のトランザクション定義オブジェクトがリスナー領域に動的にインストールでき、かつ AOR に静的にインストールできるからです。これは、「*CICSPlex System Manager Managing Business Applications*」マニュアルの Resource assignments で説明されています。

BAS 論理スコープの考慮事項

BAS を使用してユーザー・ビジネス・アプリケーション・スイートを定義し、インストールする利点の 1 つは、ユーザーがオブジェクト・ビューの有効範囲を、インストールされたアプリケーション・インスタンスに関連するリソースに限定できることです。

例えば、ビジネス・アプリケーションが特定の 1 組のファイル、トランザクション、およびプログラムで構成される場合、LOCTRAN、LOCFILE および PROGRAM ビューを、それらがインストールされる領域上で一致するオブジェクトのみのインスタンスまで切り分けられます。この制限されたオブジェクト・ビューを可能にする機能は、「論理スコープ」と呼ばれます。CorbaServer および DJAR オブジェクトは、他の従来の BAS 定義とまったく同じように論理スコープに加わることができます。

注: エンタープライズ Bean は、このようなものとして CICS に対して定義されません。CICS 領域へのインストール後に、関連した DJAR が使用可能になると、CICS に対して識別されます。したがって、エンタープライズ Bean は、DJAR の関連を通じて論理スコープを「採用」することができます。ただし、エンタープライズ Bean 仕様により、別々のアプリケーションのエンタープライズ Bean を単一の DJAR にインストールできます。このプラクティスに従っても、論理スコープ・プロセスが、インストールされたエンタープライズ Bean と該当するビジネス・アプリケーション名とを区別することは不可能です。したがって、ユーザーが BAS 論理スコープを活用して、エンタープライズ Bean オブジェクトの CICSPlex ビューを補強したい場合は、別個の DJAR を使用して、スコープ対象のビジネス・アプリケーションにとって別個のエンタープライズ Bean を含む必要があります。

エンタープライズ Bean コンポーネントのマイグレーション

CICSPlex SM が提供するツール・セットは、ユーザーが RDO (オンライン・リソース定義) オブジェクトを CICS CSD から CICSPlex SM データ・リポジトリーにマイグレーションするのに役立ちます。

このツール・セットは、CICS オフライン CSD ユーティリティー・プログラム用の出口プログラム、およびそれを実行するためのサンプル JCL サンプルで構成されます。「*CICSPlex System Manager Managing Business Applications*」マニュアルの Extracting records from the CSD を参照してください。

この CICSPlex SM 出口は、CSD 内の CORBASERVER および DJAR 定義を認識し、CICSPlex SM BatchRep プロセスを通じた入力用に、適切な BAS CREATE EJCODEF および CREATE EJDJDEF ステートメントを生成します。リソース識別の通常の選択規則がすべて、これらの EJB リソース・タイプに適用できます。

エンタープライズ Bean に対する CICSPlex SM 照会サポート

インストールされた CorbaServer インスタンスと DJAR インスタンスは、409 ページの『エンタープライズ Bean に対する CICSPlex SM サポート』で説明されている 3 つのインターフェースのいずれかを使用して、CICSPlex SM によって管理できます。 CICS CEMT および CEOT トランザクションを通じて提供されるすべての対話式オペレーター・サービスの機能は、Web ユーザー・インターフェース (WUI) を介して CICSPlex SM に複製されます。いずれの場合でも、CICSPlex SM によってマップされるインストール済み CICS オブジェクトは次のとおりです。

- EJCOSE — CorbaServer インスタンス
- EJDJAR — CICS デプロイ済み JAR ファイル・インスタンス

さらに、これらのオブジェクトを通じて任意の実行可能なエンタープライズ Bean をリストできます。

- EJCOBEAN — CorbaServer に直接関連した Enterprise JavaBeans
- EJDJBEAN — DJAR に直接関連したエンタープライズ Bean

これらのオブジェクトはどちらも、エンタープライズ Bean 構造を記述します。一方は CorbaServer 名をキーにし、もう一方は DJAR ID をキーにしています。どちらの場合も、照会に使用可能なエンタープライズ Bean の内容は、CorbaServer 名、DJAR 名、およびエンタープライズ Bean 名のみであり、最大長は 240 文字です。Enterprise JavaBean 仕様は、エンタープライズ Bean 名をさらに長くすることができると定めていますが、CICS 実装では 240 バイトに制限されます。標準 CICS 照会が提供するものの他に、CICSPlex SM 照会が提供する追加の詳細情報は、所定の DJAR または CorbaServer で使用可能な Bean 数です。新しいセットのエンタープライズ Bean が DJAR を介して特定の CorbaServer にデプロイされると、エンタープライズ Bean 数により、該当するエンタープライズ Bean が使用可能かどうかについて即時に確認できます。この値は、DJAR インストール・プロセスを通じて受け入れられるエンタープライズ Bean 数に応じて増加します。

CPSM を通じて照会可能なその他の Enterprise Java 関連 CICS オブジェクトは、次のとおりです。

- TCPIPS - TCPIPSERVICE インスタンス
- RQMODEL — REQUESTMODEL インスタンス
- LOCTRAN — ローカル・トランザクション・インスタンス
- UOWORK — 作業単位インスタンス
- UOWLINK — 作業単位リンク (UOWLINK) インスタンス
- PROGRAM — プログラム・インスタンス

これらのすべてのオブジェクトには、エンタープライズ Bean の管理と実行に関連する属性が含まれています。

エンタープライズ Bean オブジェクトに使用可能な照会のタイプ

CICSPlex SM を使用して EJB オブジェクトの状態を照会するには、複数の方法があります。

CICSPlex SM アプリケーション・プログラミング・インターフェース
使用可能な CICSPlex SM API コマンドを使用して EJB オブジェクトを照会するには、「*CICSPlex System Manager Application Programming Reference*」を参照してください。また、「*CICSPlex System Manager Resource Tables Reference Vol 1*」で、各 CICSPlex SM オブジェクトに対して許可される属性とアクションの詳細も参照してください。

CICSPlex SM Web ユーザー・インターフェース
WUI を使用して EJB オブジェクトを照会するには、「*CICSPlex System Manager Web User Interface Guide*」を参照してください。

Web ユーザー・インターフェースには、1 組のメニューとパネルで構成されるスターター・セットがあります。このスターター・セットには、1 組の Enterprise Java コンポーネント・ビューが含まれています。

CICSPlex SM を使用した EJB ワークロードの管理

標準の CICSPlex SM コンポーネント機能の 1 つは、MRO 環境で CICS トランザクションのバランスを取り、分離する機能です。これはワークロード管理 (WLM) と呼ばれます。

この機能は、EJB ワークロードの管理に適しています。この場合、エンタープライズ Bean は、分散または論理 CorbaServer 環境で実行されます。最もシンプル構成では、CICSPlex SM は、ユーザー定義で設定されるパフォーマンス目標や安定性アルゴリズムに応じて、一連のアプリケーション専有領域 (AOR) 間でエンタープライズ Bean 実行ワークロードのバランスを取ることができます。これらの機能が実装されるのは、CICSPlex SM 提供の分散ルーティング出口プログラム (EYU9XLOP) が、参加するリスナーおよび AOR のシステム初期設定パラメーターで DSRTPGM パラメーターとして指定される場合です (「*CICSPlex System Manager Managing Workloads*」の Balancing an enterprise bean workload を参照してください)。

エンタープライズ Bean 実行に適した AOR を選択するために CICSPlex SM が使用するアルゴリズムは、本製品の方向付け以降に確立され、調整されてきました。しかし、ユーザーは、必要に応じて、独自のルーティング・アルゴリズム・プログラムを作成し、提供された CICSPlex SM バージョン (EYU9WRAM) を置き換えることを選択できます。

ワークロード・ルーティング:

CICSPlex SM のワークロード・ルーティングは、事前に決定された選択基準にしたがって、エンタープライズ Bean の実行をホスティングするのに最適な AOR を選択する機能を提供します。

AOR 選択プロセスは、可能なルーティング・ターゲットとして指定される領域上ですべての同時実行アクティビティーを評価し、実行ワークロードや照会時の領域の安定性の観点から最適な領域を選択します。これは、逐次実行される Bean のターゲットの有効範囲で使用可能なすべてのものからの AOR 循環選択と同じではありません。新規トランザクション (エンタープライズ Bean) を実行しようとしている時点における WLM の有効範囲内のすべてのアクティブ・トランザクションの評価、およびオブジェクト実行をホスティングするための、ロードが最も少ない、すなわち最も安定した領域の選択です。すべての Enterprise Java Bean スループットに対する基本的なワークロード・ルーティングの実装には、次の前提条件があります。

- 必要な TCP/IP 定義が、指定されたリスナー領域にインストールされていること
- DSRTPGM=EYU9XLOP が、すべてのリスナーおよび AOR で SIT パラメーターとして指定されていること
- MASPLTWAIT(YES) が、すべてのリスナー領域で EYUPARM として組み込まれていること
- 要求プロセッサー・トランザクション (デフォルトのトランザクションは CIRP) が、リスナー領域に対して動的に定義され、AOR に対して静的に定義されていること

- 実行可能な EJB 環境を確立するために、必要な CorbaServer および DJAR 定義が (BAS または CEDA のどちらかを通じて) インストールされていること
- エンタープライズ Bean がデプロイ済みであり、サービス中であること

これらの基準が満たされた場合、ワークロード仕様オブジェクト (WLMSPEC) を定義し、ターゲットの有効範囲として AOR を指定します。その後、ワークロードに加わるすべてのリスナーおよび AOR で WLMSPEC オブジェクトをインストールできます。WLMSPEC がインストールされると、それに含まれるすべての領域では、再始動された後に EJB ワークロードが転送されます。エンタープライズ Bean のワークロード・ルーティングの詳細な例については、「*CICSPlex System Manager Managing Workloads*」マニュアルの Balancing an enterprise bean workload を参照してください。

ワークロード分離:

ワークロード分離は、事前に指定された選択基準を満たすトランザクションが、特定のターゲット有効範囲に送られるようにするワークロード管理 (WLM) 機能です。

分離されたワークロード項目のターゲット有効範囲としては、単一のアプリケーション専有領域 (AOR) から、多数の CICS 領域を含む大きい AOR グループまで、さまざまな範囲が存在する可能性があります。 AOR グループがターゲットである場合、そのグループに対して定義されている領域の中から最適な領域を選択するために、ルーティング・アルゴリズムが適用されます。分離されたエンタープライズ Bean を含むワークロードを実装するには、最初に、413 ページの『ワークロード・ルーティング』で説明されている前提条件のワークロード・ルーティングを確立する必要があります。その構成は、次の追加コンポーネントで補強される必要があります。

- 分離が必要なエンタープライズ Bean ごとに複製される CIRP トランザクション (新規名への既存定義のシンプルなコピー)
- 分離される各エンタープライズ Bean を、複製された CIRP トランザクションの 1 つに関連付ける要求モデル。

これにより、CICS および EJB 環境が確立され、エンタープライズ Bean の分離が可能になります。次に、分離を実装するために、WLM 定義が作成される必要があります。これには、複製された CIRP トランザクションを対象オブジェクトとして識別し、それらを、一連の WLM 定義を通じて必要なターゲット有効範囲に関連付ける必要があります。これらの WLM 定義は、中間の WLM グループを介して WLM 仕様全体に関連付けられなければなりません。次にこの仕様は、ワークロードに参加するすべてのリスナーおよび AOR を含む CICS グループに追加されなければなりません。エンタープライズ Bean のワークロード分離の詳細な例については、「*CICSPlex System Manager Managing Workloads*」マニュアルの Separating enterprise beans in a workload を参照してください。

エンタープライズ Bean に対する CICSPlex SM リソース・モニター

CICSPlex SM モニターは、一連の CICS システム内で指定されたリソース・インスタンスについて、ユーザー定義の間隔でパフォーマンス関連データの収集を可能にします。

現在、特定の EJB オブジェクト (CorbaServer および DJAR) についてパフォーマンス関連データは記録されていません。しかし、IIOP 要求受信側トランザクションおよび要求プロセッサー・トランザクションのパフォーマンス・データは、通常どおりに使用可能です。したがって、エンタープライズ Bean の実行パフォーマンスは、関連したトランザクション・コードを通じてモニターできます (「*CICSPlex System Manager Monitor Views Reference*」を参照してください)。414 ページの『ワークロード分離』で説明されているエンタープライズ Bean ワークロード分離の場合と同じように、ユーザーには、モニターする必要がある Bean ごとに要求モデルと CIRP 複製が必要です。ただし、CICSPlex SM モニターは BAS 論理スコープに統合されないので、モニター・ビューのスコープは、トランザクション定義をインストールした BAS リソース記述ではなく、モニター対象の領域を含む物理 CICS グループに設定されなければなりません。モニター機能の概要については、「*CICSPlex System Manager 概念および計画*」マニュアルの Collecting statistics using CICSPlex SM monitoring を参照してください。モニター機能の詳細は、「*CICSPlex System Manager Managing Resource Usage*」マニュアルの Preparing to monitor resources に記載されています。

エンタープライズ Bean の CICSPlex SM リアルタイム分析の考慮事項

CICSPlex SM のリアルタイム分析 (RTA) 機能は、ユーザーが関心を示した条件の自動および外部通知を行います。

リアルタイム分析は、次の複数のサブコンポーネントに分割できます。

- System Availability Monitoring (SAM) - 使用可能であることが計画された時間中に CICS 領域をモニターし、アクティブであると予想される領域から応答が受信されないときに通知を生成します。
- MAS Resource Monitoring (MRM) - 照会可能なすべての CICS リソースの状態をモニターし、その状態が事前決定された基準と異なるときに通知を生成します。
- 分析点モニター (APM) - MRM の機能を複製します。ただし、特定の CICS 領域ではなく、CICSplex レベルで状態を分析する点を除きます。APM は、複製された AOR を使用する環境で特に役立ちます。この場合、領域は同一であり、一般的な問題の警報を出すには 1 つの通知で十分です。

明らかに、SAM は、指定されたリスナーであるか、AOR であるかにかかわらず、CICS 領域が使用可能かどうかを報告するための便利な機能です。分散環境でエンタープライズ Bean を実行する場合、MRM は、APM の領域ベースの機能よりも、CorbaServer と DJAR の状態をモニターするのに便利です。ただし、RTA 内でエンタープライズ Bean オブジェクト自体 (EJCOBEAN および EJDJBEAN) をモニターできないことに注意してください。エンタープライズ Bean の照会は、対応する CorbaServer 名または DJAR 名のみをキーにして行うことができます。エンタープライズ Bean 名のみで特定の照会を行うことはできません。RTA 機能の概要については、「*CICSPlex System Manager 概念および計画*」マニュアルの Exception reporting using real-time analysis (RTA) を参照してください。RTA 機能の詳細は、「*CICSPlex System Manager Managing Resource Usage*」マニュアルの Preparing to perform real-time analysis に記載されています。

CICS および IIOP

このセクションでは、分散 IIOP アプリケーションをサポートするように CICS を構成するのに必要な知識について説明します。

- 『CICS における IIOP サポート』
- 420 ページの『IIOP 要求フロー』
- 429 ページの『IIOP 用の CICS の構成』
- 451 ページの『IIOP 要求の処理』

CICS における IIOP サポート

Internet Inter-ORB Protocol (IIOP) は、分散アプリケーションのフォーマットとプロトコルを定義する General Inter-ORB Protocol (GIOP) の TCP/IP ベースの実装です。

これは、共通オブジェクト・リクエスト・ブローカー・アーキテクチャー (CORBA) の一部です。IIOP の相互運用性を実装するには、クライアント・システムとサーバー・システムの両方に、CORBA オブジェクト・リクエスト・ブローカー (ORB) が必要です。

共通オブジェクト・リクエスト・ブローカー・アーキテクチャー (CORBA) は、分散アプリケーション用の標準オブジェクト指向アーキテクチャーの仕様です。オブジェクト管理グループ (OMG) と呼ばれる、500 社を超える情報技術組織のコンソーシアムによって定義されました。CORBA「*Architecture and Specification*」文書は Web サイト (http://www.omg.org/) でご覧ください。

CICS は、CORBA 2.3 によって定義された ORB および IIOP サポートを提供します。

オブジェクト・リクエスト・ブローカー (ORB)

CORBA は、システム内のクライアントとサーバー間の要求を処理するために、**ブローカー**すなわち仲介を使用します。ブローカーは、クライアントの要求を満たすのに最適なサーバーを選択し、サーバーの**実装**から、クライアントが認識する**インターフェース**を分離します。

ORB と呼ばれるブローカーは、クライアントのメソッド呼び出しを代行受信し、要求を実装できるオブジェクトの検出、それらへのパラメーターの引き渡し、メソッドの呼び出し、および結果の戻しを担当します。クライアントは、オブジェクトがどこに置かれているか、そのプログラミング言語、オペレーティング・システム、またはオブジェクトのインターフェースの一部でない他のシステム・アスペクトを認識する必要はありません。

この方法で、ORB は、異機種分散環境内のさまざまなマシン上にあるアプリケーション間の相互運用性を提供し、複数のオブジェクト・システムを相互接続します。

CICS ORB は、次のレベルの機能を実装します。

- CORBA バージョン 2.3 に対するサポート。*ただし、以下を除きます。*
  - ステートフル CORBA オブジェクト (ステートレス CORBA オブジェクトのみがサポートされます)。

注: この規則の例外は、ステートフル・セッション Bean のみです。これはサポートされます。
  - 動的起動インターフェース (DII)。
  - Dynamic Skeleton Interface (DSI)。
  - GIOP 1.1 フラグメント。
  - Portable Object Adapter (POA)。
  - 双方向 GIOP。
- IIOP 1.2 (GIOP 1.2 フラグメントを含む) に対するサポート。
- インバウンドおよびアウトバウンド IIOP 要求の両方のサポート。IIOP アプリケーションはクライアントとサーバーの両方の役目をすることができます。
- **トランザクション・オブジェクト**に対するサポート。CICS メソッド呼び出しは、オブジェクト・トランザクション・サービス (OTS) 分散トランザクションに関与する場合があります。クライアントが OTS トランザクションの有効範囲内で IIOP アプリケーションを呼び出すと、そのトランザクションに関する情報が、IIOP 呼び出しの追加パラメーターとして流れます。クライアント ORB が OTS Transaction Service Context を送信し、ターゲットのステートレス CORBA オブジェクトが `CosTransactions::TransactionalObject` を実装すると、そのオブジェクトはトランザクションとして扱われます。

  注: OTS トランザクション は、CICS トランザクション・インスタンスまたはリソース定義ではなく、分散作業単位です。 CICS トランザクションについては、32 ページの『CICS トランザクション』を参照してください。

ORB 機能は、以下のものによって CICS に実装されます。

- CICS ソケット・ドメイン・リスナー
- CICS IIOP 要求受信側
- CICS IIOP 要求プロセッサー

CICS IIOP アプリケーション・モデル

IIOP アプリケーションは、TCP/IP ネットワークで実行されるクライアント/サーバー・オブジェクト指向プログラムです。

CICS は、次のタイプの IIOP アプリケーションをサポートします。

ステートレス CORBA オブジェクト

ステートレス CORBA オブジェクトは、IIOP プロトコルを使用してクライアント・アプリケーションと通信する Java サーバー・アプリケーションです。メソッドの連続する呼び出し間でオブジェクト属性の状態は維持されません。状態は、各メソッド呼び出しの開始時に初期化され、明示的なパラメーターで参照されます。

ステートレス CORBA オブジェクトは、クライアントからのインバウンド要求を受け取ることができ、アウトバウンド IIOP 要求を行うこともできます。

CICS ステートレス CORBA オブジェクトは CICS JVM で実行されます。

CICS ステートレス CORBA オブジェクトについての詳細は、229 ページの『ステートレス CORBA オブジェクト』を参照してください。

エンタープライズ Bean

エンタープライズ Bean は移植可能な Java サーバー・アプリケーションであ

り、「*Enterprise JavaBeans Specification, Version 1.1*」によって定義されるインターフェースを使用します。 CICS は、これらのインターフェースを基礎の CICS サービスにマップすることによって実装しました。

エンタープライズ Bean は、Java リモート・メソッド呼び出し (RMI) インターフェースを使用して通信します。CICS は、CORBA オブジェクト・リクエスト・ブローカー (ORB) によって仲介される RMI over IIOP をサポートします。

エンタープライズ Bean は、**CCI Connector for CICS TS** を使用して他の CICS プログラムにリンクすることができます。また、JCICS クラス・ライブラリーを使用して CICS サービスまたはプログラムに直接アクセスするエンタープライズ Bean を作成することもできますが、これらのサーバー・アプリケーションは、CICS 以外のプラットフォームに移植できません。

エンタープライズ Bean は、プールされた JVM で実行されます。

エンタープライズ Bean について詳しくは、 252 ページの『エンタープライズ Bean とは』を参照してください。

一般的な CORBA 用語

この資料では以下の用語が使用されます。

CORBA
共通オブジェクト・リクエスト・ブローカー・アーキテクチャー (Common Object Request Broker Architecture)。オブジェクト指向の分散コンピューティング用のアーキテクチャーと仕様。

GIOP General Inter-Orb Protocol。CORBA のデータ表記仕様と相互運用性のプロトコル。さまざまな ORB の通信方法を定義しますが、使用するトランスポート・プロトコルを定義しません。

IDL インターフェース定義言語 (Interface Definition Language)。CORBA で使用される定義言語。オブジェクトで実行できるオペレーションを含めて、ある種のオブジェクトの特性と動作を記述します。

IIOP Internet Inter-Orb Protocol。TCP/IP トランスポート層を介して GIOP メッセージを送信する方法を定義します。IIOP は、TCP/IP を介した GIOP です。

Interface (インターフェース)
オブジェクトで実行できるオペレーションを含めて、ある種のオブジェクトの特性と動作を記述します。これは、Java **クラス**にマップされます。CORBA 用語では、クライアント要求は、サーバー・オブジェクトを定義するインターフェースを IDL で指定します。

IOR 相互運用オブジェクト参照 (Interoperable Object Reference)。リモート CORBA オブジェクトへの「ストリング変換された」参照。これは、サーバー ORB によって公開されます。クライアント・アプリケーションは、実行時に IOR にアクセスできなければなりません。クライアント ORB は、IOR をデコンストラクションして、(特に) リモート ORB とオブジェクトの場所、リモート ORB によってサポートされる GIOP の最大バージョン、およびリモート ORB によってサポートされるすべての関連 CORBA サービスを判別できます。

モジュール
インターフェースを含む IDL パッケージ構造。これは、Java **パッケージ**にマップされます。

OMG オブジェクト管理グループ (Object Management Group)。CORBA アーキテクチャーを定義したソフトウェア組織のコンソーシアム。

Operation (操作)
オブジェクトで実行できるアクション。これは、Java メソッドにマップされます。CORBA 用語では、クライアントは、サーバー・オブジェクトのメソッドにマップされるオペレーション (IDL で定義された) を要求します。

ORB オブジェクト・リクエスト・ブローカー (Object Request Broker)。クライアント・アプリケーションとサーバー・アプリケーション間の仲介の役目をする CORBA システム・コンポーネント。クライアントとサーバーの両方のプラットフォームで ORB が必要です。各 ORB は特定の環境に合わせて調整されますが、共通の CORBA プロトコルと IDL をサポートします。

RMI-IIOP
Remote Method Invocation (RMI) over IIOP 仕様およびプロトコル。この仕様は、CORBA プロトコルを使用して、Java 固有の RMI アプリケーション・アーキテクチャーを相互運用させる方法を定義します。これは、エンタープライズ Bean で使用される通信プロトコルです。

スケルトン
サーバーの IDL コンパイラーによって生成されるコードの断片。メッセージを (サーバーに対して) ローカル・オブジェクト上のメソッド呼び出しに解析するために、サーバー ORB によって使用されます。

スタブまたはプロキシー
クライアントの IDL または RMI コンパイラーによって生成されるコードの断片。リモート・オブジェクトでメソッドを呼び出すためにクライアント・アプリケーションで使用されます。スタブ・クラスは、クライアント ORB でメソッドを呼び出します。クライアント ORB はリモート・メソッド要求をサーバー ORB に送信します。スタブ・クラスは、併用される特定のクライアント ORB 用に生成されなければなりません。さまざまなベンダー製のクライアント ORB を使用する場合は、正しいクライアント ORB で提供されたツールを使用して生成されたクライアント・サイド・スタブを使用していることを確認する必要があります。

タイ RMI コンパイラーによって生成されるコードの断片。メッセージを (サーバーに対して) ローカル・オブジェクト上のメソッド呼び出しに解析するために、サーバー ORB によって使用されます。

IIOP 要求フロー

次の図は、着信要求の実行フローを示しています。

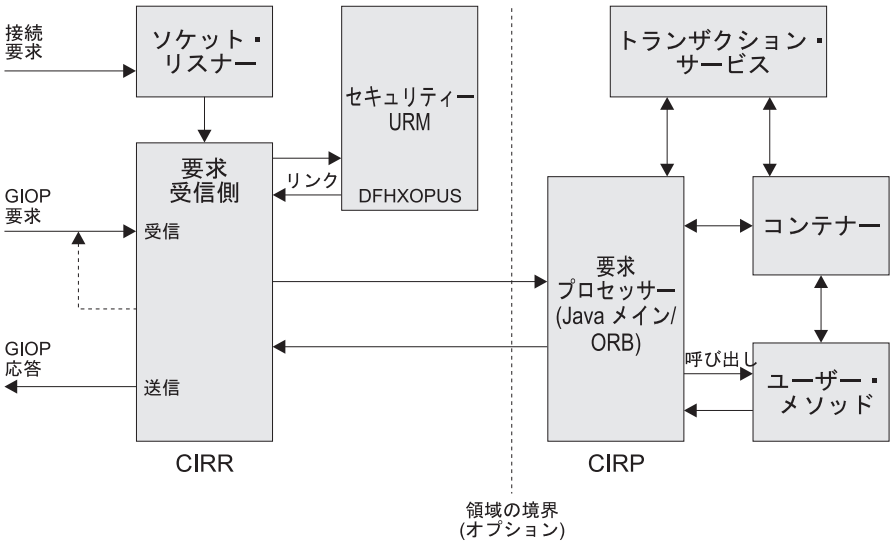


図 34. IIOP 要求の実行フロー

TCP/IP リスナー

CICS TCP/IP リスナーは、指定されたポートでインバウンド要求をモニターします。ユーザーは、TCPIPSERVICE リソースを定義し、インストールすることによって、IIOP ポートを指定し、リスナーを構成します。

リスナーは、着信要求を受け取り、そのポートに対して TCPIPSERVICE 定義で指定されたトランザクションを開始します。IIOP サービスの場合、このトランザクション・リソース定義では、プログラム属性が DFHIIRRS (要求受信側プログラム) に設定されなければなりません。デフォルトのトランザクション名は CIRR です。

要求受信側

要求受信側は、着信要求を取り出し、GIOP フォーマットのメッセージ・ストリームの内容を調べます。以下の GIOP メッセージ・タイプを受け取ることができ、次のように処理されます。

Request

- CICS USERID は、Secure Sockets Layer (SSL) パラメーターから、または TCPIPSERVICE リソース定義で指定される CICS ユーザー置換可能プログラムを呼び出すことによって判別されます。CICS USERID は、要求プロセッサーによる要求の許可に使用されます。
- CICS TRANSID は、インストールされた REQUESTMODEL リソース定義との比較によって、メッセージの内容から判別されます。CICS TRANSID は、要求を処理するために新しい要求プロセッサー・インスタンスが作成される場合に使用される、実行パラメーターを定義します。
- 要求は、関連した**要求ストリーム**を使用して要求プロセッサーに渡されます。これは内部 CICS ルーティング・メカニズムです。要求のオブジェ

クト・キー、または任意のトランザクション・サービス・コンテキストにより、要求を既存のプロセッサーに送信する必要があるかどうかが決まります。

注: この状況におけるトランザクション は、**オブジェクト・トランザクション・サービス** (OTS) 仕様を使用して定義され、管理される作業単位を意味します。

要求処理ロジックは、ディレクトリーを使用して、IIOP 要求が (関連した要求ストリームを使用して) 既存の要求プロセッサー・インスタンスに転送されるかどうかを決定します。ディレクトリー DFHEJDIR は、要求ストリーム (および要求プロセッサー・インスタンス) を OTS トランザクション、および独自のトランザクションを管理するステートフル・セッション Bean のオブジェクト・キーに関連付けます。DFHEJDIR はリカバリー可能な CICS ファイルです。

- 着信 GIOP 1.1 フラグメントは、GIOP MessageError メッセージで拒否されます。

LocateRequest
オペレーションまたはパラメーターがない要求を見つけます。これらの要求は、要求プロセッサーの新規インスタンスに渡されます。

CancelRequest
取り消し要求は、指定された保留中の Request または LocateRequest メッセージへの応答をクライアントが期待しなくなったことをサーバーに通知します。これは通知専用メッセージであり、応答は必要ありません。フラグメント処理時に受信された取り消し要求は、進行中の要求を終了させます。その他の取り消し要求はすべて無視されます。

MessageError
メッセージ・エラーは、要求受信側が送信した応答をクライアントが認識しなかったことを示します。このエラーは、診断のために記録され、CloseConnection メッセージが接続を終了するために送信されます。

Fragments
フラグメントは、Request または Reply の継続です。これには、GIOP メッセージ・ヘッダーの後にデータが続いたものが入っています。着信 GIOP 1.1 フラグメントは、GIOP MessageError メッセージで拒否されます。

要求受信側から要求プロセッサーへのリンケージは、CICS 動的ルーティング・サービスを活用して、CICSplex 内のロード・バランシングを提供できます。

CIRR 要求受信側は、実行する作業がなくなったら終了します。(すなわち、CIRR が終了するのは、TCPIPSERVICE から読み取る未処理の GIOP 要求がなく、以前の要求から送信する未処理の応答がない場合です。CIRR タスクが終了した後で TCPIPSERVICE の追加ワークロードが着信する場合、新しい CIRR タスクが開始されます。)

要求プロセッサー
要求プロセッサーは IIOP 要求の実行を管理します。これは、以下を行います。

- 要求で識別されるオブジェクトの位置を確認します

- エンタープライズ Bean 要求の場合、Bean メソッドを処理するためのコンテナーを呼び出します
- ステートレス CORBA オブジェクトの要求の場合、要求自体を処理します (ただし、トランザクション・サービスも関係する場合があります)

各 IIOP 要求を処理する要求プロセッサー・インスタンスは、CORBASERVER リソース定義によって構成されます。

シスプレックス内の IIOP

単一の CICS 領域に 1 つの CICS CORBA サーバーを実装できます。ただし、シスプレックス内では、複数の領域で構成されるサーバーを作成する必要が生じる場合があります。

複数の領域を使用すれば、単一領域の障害の重大度が低くなり、ワークロード・ルーティングを使用できるようになります。CICS 論理サーバーは、単一のサーバーと同様に動作するように構成された 1 つ以上の CICS 領域からなります。

通常、CICS 論理サーバーは次の要素からなります。

- 着信 IIOP 要求を listen するために同一の TCPIPSERVICE リソース定義で定義される、1 組の複製されたリスナー領域。
- 1 組の複製されたアプリケーション専有領域 (AOR)。各 AOR は、同じ定義を持つ CORBA サーバー内の、同じ 1 組の IIOP アプリケーションまたはエンタープライズ Bean クラスをサポートします。同一 OTS (オブジェクト・トランザクション・サービス) トランザクションに対する複数のメソッドが、同一 AOR に送られます。各 AOR には、対応するリスナー領域のものと一致する TCPIPSERVICE 定義が必要です。

リスナー領域と AOR は分離することも、結合してリスナー AOR にすることもできます。次のシステム初期設定パラメーターを指定する必要があります。

IIOPLISTENER=YES
: リスナー領域、または結合リスナー AOR でこの値を指定します。 YES がデフォルト値です。

IIOPLISTENER=NO
: リスナー領域ではない AOR でこの値を指定します。

IIOP 要求のワークロード・ルーティング

リスナー領域間でクライアント接続をルーティングするために、IP ルーティング、またはドメイン・ネーム・システム (DNS) 登録を使用することによる接続最適化のどちらかを使用できます。1 組の複製されたアプリケーション専有領域 (AOR) 全体でオブジェクト・トランザクション・サービス (OTS) トランザクションをルーティングするには、分散ルーティングを使用します。分散ルーティングを実装するには、CICSPlex SM、またはカスタマイズされたバージョンの CICS 分散ルーティング・プログラム DFHDSRP のどちらかを使用できます。

ドメイン・ネーム・システム (DNS) による接続の最適化
: 接続の最適化とは、sysplex ドメイン内の IP 接続のバランスを DNS によって調整するための手法です。DNS では、同じポートで (仮想 IP アドレスを使用して) IIOP 要求を listen するために、複数の CICS システムが開始され、MVS ワークロード・マネージャー (WLM) に登録されます。各クライアント IIOP

要求には、汎用ホスト名とポート番号が含まれています。このホスト名は、DNS および WLM サービスによって IP アドレスに解決されます。

WLM を使用した接続最適化については、「*z/OS Communications Server: IP 構成ガイド*」で説明されています。

分散ルーティング

分散ルーティングは、1 組の CICS AOR 間でエンタープライズ Bean および CORBA ステートレス・オブジェクトに対するメソッド呼び出しをルーティングするために使用されます。ワークロード・マネージャー (CICSPlex SM またはユーザー作成の分散ルーティング・プログラム) によって、ターゲットの動的選択が行われます。これは、ロードが最も少ない、すなわち最も効率の高いアプリケーション領域を選択します。CICS は、新規 OTS トランザクションで実行されるか、OTS トランザクションなしで実行されるメソッド要求に対してワークロード・マネージャーを呼び出します。ただし、既存の OTS トランザクションで実行されるメソッド要求には呼び出しません。これらの要求は、既存の OTS トランザクションが実行される AOR に自動的に送信されます。カスタマイズされた分散ルーティング・プログラムの作成については、「*CICS Customization Guide*」の Writing a distributed routing program を参照してください。CICSPlex SM ワークロード管理については、「*CICSPlex System Manager Managing Workloads*」マニュアルの Workload management and dynamic routing を参照してください。

次の図は、CICS 論理サーバーを示しています。この例では、リスナー領域と AOR は別々のグループ内にあり、接続の最適化を使用して、リスナー領域とのクライアント接続のバランスが取られ、分散ルーティングを使用して、AOR 間で OTS トランザクションがルーティングされます。

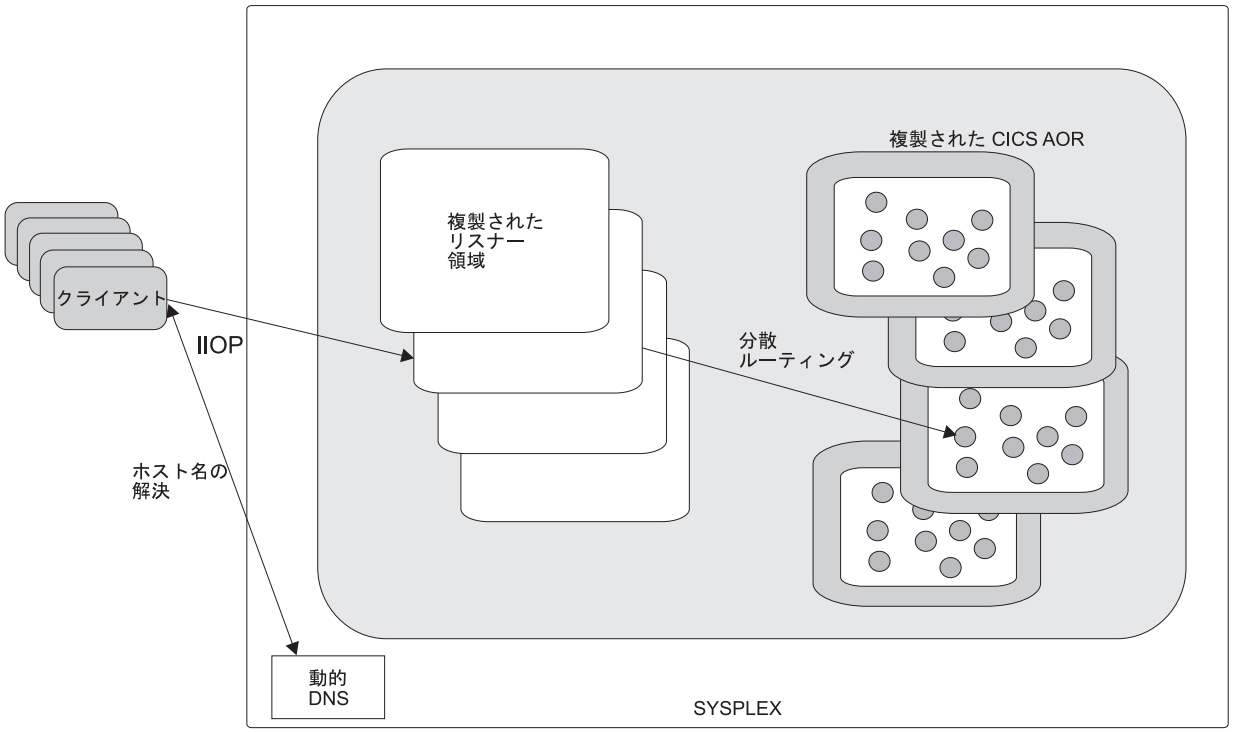


図 35. CICS 論理サーバー： この例では、論理サーバーは、1 組の複製された「リスナー」領域と 1 組の複製された AOR で構成されます。動的 DNS 登録を通じた接続の最適化を使用して、クライアント接続をリスナー領域にルーティングします。分散ルーティングを使用して、AOR 間で OTS トランザクションのバランスを取ります。

ドメイン・ネーム・システム (DNS) による接続の最適化

接続の最適化とは、sysplex ドメイン内の IP 接続とワークロードのバランスを DNS によって調整するための手法です。

DNS の用語でいえば、sysplex は、DNS のネーム・スペースに追加するサブドメインになります。接続の最適化機能は、「DNS のホスト名」の概念をサーバー・アプリケーションまたはホストのクラスターまたはグループに拡張したものです。同じグループに組み込まれている各サーバー・アプリケーションは、同等のサービスを提供するものと見なされます。接続の最適化機能は、負荷に基づく順序付けによって、それぞれのクラスターについてどのアドレスを返すかを決定します。

接続最適化の登録:

サーバー・アプリケーションは、MVS ワークロード・マネージャー (WLM) に登録します。WLM は、sysplex 内のサーバー・リソースの可用性を定量化します。

sysplex 内のすべてのホストでは、WLM をゴール・モードで構成する必要があります。TCP/IP スタックも WLM に登録できます。そうすれば、開始 IP アドレスについての情報を提供できるようになります。スタックが登録をサポートしていない場合は、静的な定義を使用できます。登録時に、サーバー・アプリケーションでは、以下の情報を用意します。

グループ名
sysplex 内の同等のサーバー・アプリケーションを組み込んだクラスターの名前

です。この名前は、sysplex ドメイン内でクライアント・アプリケーションがサーバー・アプリケーションにアクセスするときに使用することになります。CICS は、WLM に登録するグループ名として、TCPIPSERVICE リソース定義の DNSGROUP パラメーターを使用します。

サーバー名
サーバー・アプリケーション・インスタンスの名前です。サーバー名は、同じグループ名を共用するすべてのサーバーの間で固有の名前でなければなりません。1 つのサーバー・アプリケーション・インスタンスを複数のグループに割り当てることもできます。CICS は、APPLID システム初期設定パラメーターで指定されている領域の特定の APPLID を使用して、WLM に登録します。

ホスト名
サーバー・アプリケーションを実行する TCP/IP スタックのホスト名です。CICS は始動時に、TCP/IP 関数 *gethostbyaddr* を呼び出して、自身が実行されているマシンのホスト名を確認し、そのホスト名を登録のために WLM に渡します。

ネーム解決の例:

この例は、4 つの CICS 領域を組み込んだ CICSplex を示しています。この場合、それぞれの CICS 領域は、sysplex 内の別々のマシンで稼働します。

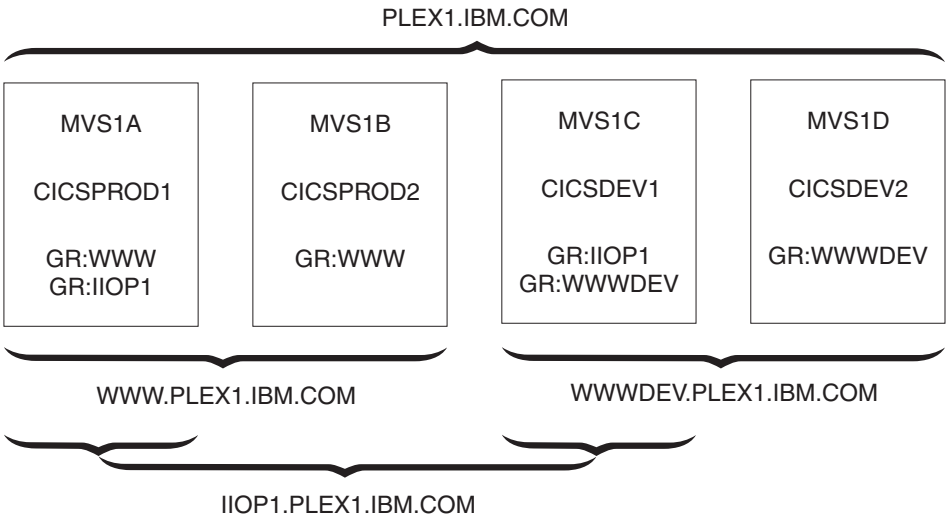


図 36. DNS による接続の最適化を使用した CICSplex

各 MVS システムの名前は、MVS1A、MVS1B、MVS1C および MVS1D です。各 CICS 領域の APPLID は、CICSPROD1、CICSPROD2、CICSDEV1 および CICSDEV2 です。

この sysplex は、PLEX1 という名前を持つように DNS に対して定義され、各 MVS マシンには 1 つの IP アドレスがあります。この図からわかるとおり、クライアント・マシンが CICS 領域にアクセスするときに使用できる名前は、各 CICS にインストールされている以下のリソース定義に基づいています。

- マシン MVS1A で実行される領域 CICSPROD1 には、2 つの TCPIPSERVICE リソースがあります。1 つのリソースは group_name を WWW に指定し、もう 1 つのリソースは group_name を IIOP1 に指定します。
- マシン MVS1B で実行される領域 CICSPROD2 には、1 つの TCPIPSERVICE リソースがあり、group_name を WWW に指定します。
- マシン MVS1C で実行される領域 CICSDEV1 には、2 つの TCPIPSERVICE リソースがあります。1 つのリソースは group_name を IIOP1 に指定し、もう 1 つのリソースは group_name を WWWDEV に指定します。
- マシン MVS1D で実行される領域 CICSDEV2 には、1 つの TCPIPSERVICE リソースがあり、group_name を WWWDEV に指定します。

クライアントは次の名前にアクセスできます。

- PLEX1.IBM.COM は、sysplex 内の任意のマシンの IP アドレスを戻します。
- WWW.PLEX1.IBM.COM は、アドレス MVS1A または MVS1B のどちらかを戻します。
- IIOP1.PLEX1.IBM.COM は、アドレス MVS1A または MVS1C のどちらかを戻します。
- WWWDEV.PLEX1.IBM.COM は、アドレス MVS1C または MVS1D のどちらかを戻します。

グループ内の個々の CICS 領域のアドレス指定にも、APPLID (またはサーバー名) を使用できます。例えば、CICSPROD1.WWW.PLEX1.IBM.COM からは、MVS1A のアドレスが返されます。このアドレスは MVS1A.PLEX1.IBM.COM と等価ですが、この場合、WWW グループに含まれているのは CICSPROD1 だけなので、クライアントは、CICSPROD1 サーバーが稼働しているマシンを認識する必要はありません。

これらの名前は、CICS 領域を WLM に登録した時点で動的に有効になるので、CICS 領域や MVS マシンを追加したからといって、管理作業が増えるわけではありません。汎用のホスト名 (WWWDEV.PLEX1.IBM.COM など) を使用すれば、特定の CICS 領域や MVS ホストからクライアント・アプリケーションを分離できるので、可用性と拡張容易性が向上します。

DNS 接続最適化に対するリソース定義:

これらの TCPIPSERVICE オプションは、DNS 接続最適化を使用する TCP/IP ポートに定義されなければなりません。

DNSGROUP
ワークロード・マネージャーに対する IWMSRSRG レジスター呼び出しに渡すロケーション・パラメーターを指定します。長さが 18 文字以内の値を指定できます。末尾ブランクは無視されます。

このパラメーターは、OS/390 TCP/IP DNS 資料では group_name と呼ばれます。これは、sysplex 内の同等のサーバー・アプリケーションを組み込んだクラスターの総称名です。これは、CICS TCPIPSERVICE にアクセスするためにクライアントが使用するシスプレックス・ドメイン内の名前でもあります。

複数の TCPIPSERVICE が同じグループ名を指定できます。

このグループ名を指定した最初のサービスがオープンされたときに、WLM への登録呼び出しが行われます。同じグループ名を指定した以降のサービスでは、追加の登録呼び出しは行われません。

以下で説明しているように、登録解除アクションは、GRPCRITICAL 属性によって決まります。マスター端末 (CEMT) または **EXEC CICS** のコマンド **SET TCPIPSERVICE DNSSTATUS DEREGISTERED** を発行するか、または等価の CICSPlex SM コマンドを使用して、グループから明示的に CICS を登録解除することもできます。

GRPCRITICAL
サービスを DNS グループの重要なメンバーとしてマークします。その結果、このサービスをクローズしたり、このサービスで障害が発生すると、WLM に対してこのグループ名の登録解除呼び出しが行われます。

デフォルトは NO であり、同じ名前の 2 つ以上のサービスが別個に失敗することが可能であり、グループに対して CICS は登録されたままになります。グループ内の最後のサービスがクローズされたときに、まだ WLM に対する登録解除呼び出しが明示的に行われていない場合にのみ、登録解除呼び出しが行われます。

同じグループ名を持つ複数のサービスに、異なる grpcritical 設定を指定できます。GRPCRITICAL(NO) を指定したサービスは、登録解除を引き起こさずにクローズまたは失敗することが可能です。GRPCRITICAL(YES) を指定したサービスがクローズまたは失敗した場合、グループは WLM から登録解除されます。

IIOP 要求 (エンタープライズ Bean に対する要求を含む) の DNS 接続最適化を実装する場合は、以下の CORBASERVER オプションを定義する必要があります。

- CORBASERVER 定義の HOSTNAME オプションは、総称ホスト名を指定する必要があります。この汎用ホスト名は、TCPIPSERVICE 定義の DNSGROUP 値に、MVS のネーム・サーバーによって管理されているドメイン名またはサブドメイン名を接尾部として追加した名前になります。そのドメイン名は、TCP/IP 管理者が設定します。例えば、上記の例では、WWW.PLEX1.IBM.COM を使用して CICSPROD1 および CICSPROD2 に転送できます。
- 総称ホスト名を持つ CORBASERVER (またはその中の DJAR) を、ネーム・サーバーに公開する必要があります。

ネーム・サーバーは、総称ホスト名を検索し、解決できるように構成されなければなりません。

ドメイン・ネーム・システム (DNS) の問題の回避:
重要

ネーム・サーバーを使用する際の問題を回避するために、以下の点に注意する必要があります。

- 動的名の検索をキャッシュに入れてはなりません。ネーム・サーバーの検索結果をキャッシュに入れるクライアントを使用する場合、正しい IP アドレスを引き続き処理することが確実になりません。この結果、クライアントは、他のサーバーによって以前に実行された役割を引き継いだ別のサーバー領域のアドレスを取得するのではなく、クローズされたサーバー領域を引き続き呼び出そうとする可能性があります。

- 使用されるネーム・サーバーのストレスのために、問題が生じる可能性があります。成功する検索もあれば、NameNotFoundException で失敗する検索もあります。

  同時検索数が増えるときに、おそらくクライアントまたは Bean がキャッシングなしに繰り返し検索を行うときに、これらのネーム・サーバーのいずれかの「ブリップ」を検出する可能性が高くなります。検討できる手段は次のとおりです。

  – ネーム・サーバーを実行するためにより大きい容量のマシンをインストールします。
  – この可能性を認識し、このエラーが検出されると再試行するように、アプリケーションをコーディングします。
  – 最もよく使用されるアドレスが /etc/hosts ファイルに組み込まれるように、MVS システムをセットアップします。これにより、これらの名前のネーム・サーバー検索が迂回され、このファイルでコーディングされるアドレスが使用されます。
  – IP アドレスを名前で指定するのではなく、番号で指定します(ただし、この解決方法は、実稼働環境ではお勧めしません)。

IIOP ユーザー置換可能セキュリティー・プログラム

これは、オプションの識別メカニズムです。

これは、認証メカニズムではなく、CICS USERID を指定する方法です。これを使用するには、IIOP ポート用の TCPIPSERVICE 定義の URM オプションで、セキュリティー・プログラムの名前を指定する必要があります。これを行う場合、セキュリティー・プログラムは IIOP 要求プロセッサーによって呼び出されます。

呼び出し後、セキュリティー・プログラムは、システム初期設定パラメーター DFLTUSER によって定義される値 (デフォルトで CICSUSER に設定されます) があらかじめ提供されます。ただし、これをオーバーライドできます。IIOP 要求を要求プロセッサーに転送する前に、CICS は、要求受信側トランザクションが、セキュリティー・プログラムによって生成される USERID に代わって作業を開始できることを、RACF で確認します。

USERID を提供する独自のプログラムを作成するか、サンプル・セキュリティー・プログラム DFHXOPUS を使用できます。454 ページの『IIOP ユーザー置換可能セキュリティー・プログラムの使用』を参照してください。

CONNECTION 認証

クライアント USERID がリスナー領域から AOR に送信されるのは、ATTACHSEC(IDENTIFY) が AOR の CONNECTION 定義で指定される場合のみです。

詳しくは、「*CICS RACF Security Guide*」の Link security with MRO を参照してください。

IIOP ユーザーは SEC=YES および ATTACHSEC(IDENTIFY) を指定するようにお勧めします。

IIOP 用の CICS の構成

エンタープライズ Bean を含めて、IIOP ベースのすべてのアプリケーションを実行するには、CICS を CORBA 参加プログラムとして構成する必要があります。

Java を実行するための要件に加えて、次のソフトウェアも必要な場合があります。

- Java Naming and Directory Interface (JNDI) バージョン 1.2。
- IBM Data Server Driver for JDBC and SQLJ の拡張機能を持った DB2。

以下のステップを実行してください。

- 『IIOP 用のホスト・システムのセットアップ』
- 443 ページの『IIOP 用の TCP/IP のセットアップ』
- 445 ページの『IIOP 用の CICS のセットアップ』

以下のいずれかのステップの実行が必要になる場合もあります。

- 432 ページの『LDAP サーバーのセットアップ』
- 443 ページの『COS Naming Directory Server のセットアップ』

432 ページの『LDAP サーバーのセットアップ』を選択した場合は、438 ページの『LDAP ネーム・スペース構造』もお読みください。

IIOP 用のホスト・システムのセットアップ

IIOP をサポートするには、以下のシステム・タスクを実行してください。

このタスクについて

手順

1. CICS 領域が z/OS UNIX システム・サービスにアクセスできるようにします。このタスクの一環として、次のことを行います。
   a. JVM の作成に必要な z/OS UNIX ディレクトリーおよびファイルに CICS がアクセスできるようにします。
   b. JVM からの入力、出力、およびメッセージに指定した z/OS UNIX 作業ディレクトリーを作成し、CICS がアクセスできるようにします。
2. 102 ページの『プールされた JVM のセットアップ』. このタスクの間、次のことを行います。
   a. CICS が JVM プロファイルおよび関連したすべての JVM プロパティー・ファイルの位置を確認できるようにします。
   b. CORBA ステートレス・オブジェクトおよびエンタープライズ Bean に適切な JVM プロファイルを選択します。
   c. 必要に応じて、CICS 領域の要件に合うように JVM プロファイルをカスタマイズします (CICS を CORBA サーバーとしてセットアップしている間に、何らかの情報を追加する必要があります)。

   102 ページの『プールされた JVM のセットアップ』を読む場合は、*CORBA ステートレス・オブジェクトとエンタープライズ Bean* について、以下のことに注意してください。

   - 使用される JVM プロファイルは、**要求プロセッサー**・プログラムの PROGRAM 定義で指定されたプロファイルです。

- すべての CICS Java プログラムについて、JVM プロパティー・ファイルを使用する場合は、それが JVM プロファイルで指定されなければなりません。
- デフォルトの要求プロセッサー・プログラムの PROGRAM 定義で指定されるデフォルトの JVM プロファイルは、DFHJVMCD です。
- CORBA ステートレス・オブジェクトおよびエンタープライズ Bean 要求でデフォルトの JVM プロファイルを使用する計画の場合は、102 ページの『プールされた JVM のセットアップ』で説明しているように、DFHJVMCD を見つけ、CICS 領域に合わせてプロファイルをカスタマイズするだけで済みます。

  カスタマイズされた JVM プロファイルを使用する計画の場合であっても、CICS 領域のセットアップに合わせるために必要な変更を DFHJVMCD に加える必要があります。これは、DFHJVMCD が CICS によって内部で使用されるとともに、デフォルトの要求プロセッサー・プログラムに使用されるからです。

3. 『シェルフ・ディレクトリーの定義』. デプロイ済み JAR ファイルにはシェルフ・ディレクトリーが使用されます。
4. 『ネーム・サーバーの定義』. このステップが必要なのは、その手順で説明されている目的にネーム・サーバーを定義する必要がある場合のみです。

シェルフ・ディレクトリーの定義:

どの CORBASERVER 定義でも、z/OS UNIX 上のシェルフ・ディレクトリーの名前を指定する必要があります。

DJAR 定義がインストールされると、CICS はデプロイ済み JAR ファイルをシェルフ・ルート・ディレクトリーのサブディレクトリーにコピーします (また、PERFORM CORBASERVER PUBLISH コマンドが発行されるときに、CorbaServer の IOR がそのサブディレクトリーに書き込まれます)。

シェルフ・ディレクトリーには、任意の名前を指定できます。ただし、/var ディレクトリーの下で作成することをお勧めします。例えば、/var/cicsts/ と呼ばれる z/OS UNIX ディレクトリーを作成できます。シェルフ・ディレクトリーを作成したら、そのディレクトリーへの全アクセス権限 (読み取り、書き込み、および実行) を CICS 領域のユーザー ID に付与する必要があります。詳しくは、Giving CICS regions access to z/OS UNIX System Services を参照してください。

ネーム・サーバーの定義:

次の 2 つの目的でネーム・サーバーの定義が必要な場合があります。

1. ドメイン・ネーム・システムの接続最適化を使用する場合、MVS ワークロード・マネージャーが使用するように構成されるのと同じ、z/OS 上のネーム・サーバーと対話するように、リスナー領域が構成される必要があります。

   CICS 開始 JCL の SYSTCPD DD ステートメントをリスナー領域に提供することによって、TCP/IP で使用されるネーム・サーバーを定義できます。「*CICS Transaction Server for z/OS インストール・ガイド*」マニュアルの Enabling TCP/IP in a CICS region を参照してください。
2. クライアント・アプリケーションは、ネーム・サーバーで登録されたオブジェクト参照を使用して、IIOP サーバー・アプリケーションを見つけることができま

す。例えば、Java クライアントは JNDI インターフェースを使用して、エンタープライズ Bean のホーム・インターフェースのインスタンスなどのサーバー・アプリケーション・オブジェクトへの参照を取得できます。オブジェクト参照を CICS からネーム・サーバーで登録するには、PERFORM CORBASERVER PUBLISH コマンドまたは PERFORM DJAR PUBLISH コマンドを発行します。

JNDI 参照の使用可能化:

アプリケーションが JNDI インターフェースを使用して参照を取得できるようにするには、Java Naming and Directory Interface (JNDI) V 1.2 をサポートするネーム・サーバーをセットアップしてください。

次のいずれかの方法を使用できます。

Lightweight Directory Access Protocol (LDAP) サーバー
z/OS で LDAP ネーム・サーバーを使用する場合は、CICS および WebSphere からのエンタープライズ Bean は、共用ネーム・スペースでより簡単に相互運用できます。 432 ページの『LDAP サーバーのセットアップ』を参照してください。

Corba Object Services (COS) ネーミング・ディレクトリー・サービス
COS ネーム・サーバーは外部マシンで実行されます。

JNDI バージョン 1.2 をサポートする業界標準の任意の COS ネーム・サービスを使用できます。 443 ページの『COS Naming Directory Server のセットアップ』を参照してください。

JNDI ネーム・サーバーの場所の指定:

CICS で実行される Java コードが JNDI API 呼び出しを発行できるようにし、CICS がエンタープライズ Bean のホーム・インターフェースまたはステートレス CORBA オブジェクトの IOR への参照を公開できるようにするには、ネーム・サーバーの場所を定義する必要があります

このタスクについて

-Dcom.ibm.cics.ejs.nameserver システム・プロパティーを使用して、ネーム・サーバーの Web アドレス (URL) および TCP/IP ポート番号を指定します。 127 ページの『JVM システム・プロパティー』に詳細情報が記載されています。

重要:

1. CORBA ステートレス・オブジェクトまたはエンタープライズ Bean で使用されるすべての JVM プロファイルまたはオプションのプロパティー・ファイルで、**-Dcom.ibm.cics.ejs.nameserver** システム・プロパティーにネーム・サーバーの場所を指定する必要があります。
2. 特に、必ず、DFHJVMCD JVM プロファイルでネーム・サーバーの場所を指定してください。DFHJVMCD プロファイルは、デフォルトの要求プロセッサー・プログラムや、デプロイ済み JAR ファイルの公開と撤回に CICS が使用するプログラムを含めて、CICS 定義のプログラムで使用されます。
3. また、CORBA ステートレス・オブジェクトまたはエンタープライズ Bean に使用することを選択する他の JVM プロファイルで、ネーム・サーバーの場所を指定することも必要です。これらは、CICS 提供のサンプル JVM プロファイルま

たは独自の JVM プロファイルの場合があります。CORBA ステートレス・オブジェクトおよびエンタープライズ Bean の場合、JVM プロファイルは、要求プロセッサー・プログラムの PROGRAM リソース定義で指定されます。

4. ネーム・サーバーの場所の定義について詳しくは、 127 ページの『JVM システム・プロパティー』を参照してください。

LDAP サーバーのセットアップ

WebSphere 用に構成されている既存の LDAP サーバーを使用するか、新しい LDAP サーバーを構成してください。

WebSphere 用に構成されている既存の LDAP サーバーを使用する場合:

CICS で使用するために選択したネーム・サーバーが、WebSphere Application Server for z/OS 用に既に構成されている場合、CICS がそれを使用できるようにするのに必要な構成はおそらくほとんどありません。

CICS における EJB サポートの適切なオペレーションには、選択された LDAP ネーム・スペースが WebSphere System Namespace を使用して構成されていることが必要です。CICS の公開メカニズムと撤回メカニズムはどちらも、System Namespace 構造内で作動しようとします。ただし、EJB メソッドに入った後、または通常の Java トランザクションを CICS で実行する場合、システム・ネーム・スペースをサポートするかどうかに関係なく、任意の LDAP ネーム・スペースと通信できます。

WebSphere System Namespace を使用して構成されていない LDAP サーバーを使用する場合は、CICS に提供される WebSphere コンテキスト・ファクトリーではなく、IBM Developer Kit for the Java Platform 5.0 ベースに提供される LDAP サービスなどの代わりのディレクトリー・サービスを使用してください。LDAP ファクトリーの使用について詳しくは、 344 ページの『Java に付属の LDAP コンテキスト・ファクトリー』を参照してください。

LDAP サーバーに存在する WebSphere ネーミング構造 (438 ページの『LDAP ネーム・スペース構造』を参照) を理解すると、ユーザー自身や LDAP 管理者が、CICS 領域が認識する必要がある 6 つの主なプロパティーに適切な値を判断するのが容易になります。これについては、 127 ページの『JVM システム・プロパティー』を参照してください。LDAP ネーム・スペースが安全な方法でセットアップされる場合に必要なのは、3 つのセキュリティー・プロパティーのみです。一部の LDAP サーバーでは、すべてのユーザーが書き込みアクセス権限を持っていれば、プリンシパル・プロパティーも資格情報プロパティーも CICS 領域に設定する必要がないのは事実です。

WebSphere によってネーム・スペースで配置される構造がニーズに適している場合、追加構成は不要です。

nameserver、containerdn および noderootrdn の値を入手するには、System Namespace の構造を理解し、選択した LDAP サーバーに配置された構造を監視します。このセクションの最後の部分では、既存のネーム・スペースをブラウズする場合にプロパティー値を判別する方法について説明します。

追加構成の理由:

サーバーが既に WebSphere Application Server for z/OS 用に構成されている場合であっても、次のいずれかの理由で LDAP サーバー構成の続行が必要な場合があります。

1. 導入される CICS 領域を処理するために、セキュリティー構成に変更が必要であるため。LDAP 構造とセキュリティー問題について詳しくは、 438 ページの『LDAP ネーム・スペース構造』および 440 ページの『セキュリティーの考慮事項』を参照してください。
2. CICS が、WebSphere とは別個の*ドメイン* で実行する必要があるため。新しい別個のドメインを作成する場合、WebSphere Application Server for z/OS および CICS は、互いのエンタープライズ Bean を容易に見つけることができません。ただし、新しいドメインを作成する予定である場合、実行する必要がある構成ステップは、ステップ 4.「legacyRoot ノードを作成します」とステップ 5.「CICS 領域レベルでセキュリティーを適用します」のみです。
3. CICS が、LDAP サーバー上のまったく異なるシステム・ネーム・スペース構造で実行する必要があるため。つまり、CICS には、サーバー上の既存のネーム・スペース・ルート・ロケーション以外のどこかを指す `containerdn` が必要です。この場合、ステップ 2.「新しい接尾部を追加します」から構成手順を開始します。この場合、異なるコンテナー設定を使用する CICS と WebSphere Application Server for z/OS システムは、相互のエンタープライズ Bean を見つけることはできません。

新規 LDAP サーバーの構成:

WebSphere Application Server for z/OS 用に構成されている既存の LDAP サーバーがない場合は、以下のステップを実行して新規 LDAP サーバーを構成してください。

このタスクについて

1. WebSphere ネーミング・スキーマをインストールします。
2. 新しい接尾部を追加します。
3. システム・ネーム・スペース・ルート・ノード (containerdn) を作成します。
4. ネーム・スペース・ルート・ノード (noderootrdn) の下に legacyRoot ノードを作成します
5. オプションとして、CICS 領域レベルでセキュリティー手段を適用します。

これらのステップの多くを実行するために、おそらく、*ルート・レベル*で新規項目を作成するために LDAP サーバーで適切な権限がある LDAP プリンシパルへのアクセス権限が必要になります。

これらのステップを完了したら、JVM プロパティー・ファイルで必要なシステム・プロパティーの値を判別して、CICS が LDAP サーバーで動作できるようにし、これらのシステム・プロパティーを関係するすべての JVM プロパティー・ファイルに追加することができます。

以下の例のステップでは、JVM プロパティー・ファイルのシステム・プロパティーに以下の値を指定して LDAP サーバーを構成できます。

```
-Dcom.ibm.cics.ejs.nameserver=ldap://wibble.example.com:389
-Dcom.ibm.ws.naming.ldap.containerdn=ibm-wsnTree=t1,o=WASNaming,c=US
-Dcom.ibm.ws.naming.ldap.noderootrdn=ibm-wsnName=legacyRoot,ibm-wsnName=PLEX2,
```

```
      ibm-wsnName=domainRoots
-Djava.naming.security.authentication=simple
-Djava.naming.security.principal=cn=CICSSystems,c=US
-Djava.naming.security.credentials=secret
```

CICS 提供のサンプル JVM プロパティー・ファイルではシステム・プロパティーの例に、同じような値が与えられます。

例:

この例で使用される構成ファイルには注記があります。これは、特定のニーズに合わせたこのプロパティー・セットの調整方法を示します。

変更する可能性が最も高いのは、*noderootrdn* です。おそらく、ノードのグループとして、PLEX2 以外のドメインがあります。この値は、ステップ 4.「legacyRoot ノードを作成します」でシステムに入力されます。

この例では、プリンシパル「cn=admin」が LDAP サーバーに存在し、パスワード「adminpwd」を持っていること、およびこのプリンシパルが LDAP サーバー上のすべてのオペレーションを実行する権限を持つことを前提としています。

1. WebSphere ネーミング・スキーマをインストールします。

   構成される LDAP サーバーに既に WebSphere ネーミング・スキーマがある場合、このステップをスキップできます。WebSphere 用に構成されている LDAP ネーム・サーバーには、既にこのスキーマがあります。

   それ以外の LDAP サーバーである場合は、WebSphere ネーミング・スキーマをインストールします。このスキーマは、 z/OS UNIX 上の /usr/lpp/cicsts/cicsts42/utils/namespace/WebSphereNamingSchema.ldif として CICS に付属しています。

   注: WebSphereNamingSchema.ldif ファイルでは、RFC2256.ldif および RFC2713.ldif が最初にロードされる必要があります。これは、**ibm-wsnEntry** オブジェクト・クラスの定義が、**javaClassName** 属性タイプを参照するからです。z/OS で LDAP サーバーを使用する場合、LDAP サーバーのセットアップ時には、これらの前提条件 LDAP ファイルはデフォルトでロードされません。

   z/OS 上の LDAP サーバーは、スキーマ項目が適用されるバックエンド・ストアに、それらのスキーマ項目を保管する必要があります。これを行うには、各スキーマ項目の dn に接尾部を追加します。提供される WebSphereNamingSchema.ldif ファイルは、スキーマ項目に接尾部を指定しないので、ユーザーが追加する必要があります。例えば、バックエンド・ストアの接尾部が「c=US」である場合、ldif ファイル内の「dn:cn=schema」のすべてのインスタンスを「dn:cn=schema,c=US」に変更する必要があります。

   次の **ldapmodify** コマンドを使用して、スキーマをネーム・サーバーに適用します。

   ```
   ldapmodify -h <hostname>
              -p <portnumber>
              -D <authorized_principal>
              -w <authorized_principal_password>
              -f WebSphereNamingSchema.ldif
   ```

ここで、hostname と portnumber は、LDAP サーバーのものであり、許可されたプリンシパルは、ネーム・サーバー上で項目を書き込むための十分な権限を持つユーザーの識別名です。

ldapmodify コマンドは、選択された LDAP サーバーに使用可能でなければなりません。使用可能でない場合は、LDAP サーバー資料を参照して、新しいスキーマ (ldif 形式) のインストール方法を判別してください。

具体的な例は次のとおりです。

```
ldapmodify -h wibble.example.com
           -p 389
           -D cn=admin
           -w adminpwd
           -f WebSphereNamingSchema.ldif
```

2. 新しい接尾部を追加します。

   ネーム・スペース内に新しい階層を作成するには、新しい基本識別名接尾部を作成する必要があります。この構成例では、接尾部は *c=US* であり、新しい階層は *ibm-wsnTree=t1,o=WASNaming,c=US* になります。接尾部を追加する手順は、LDAP プロバイダー間で異なります。選択したプロバイダーについてこれを行う方法は、ご使用の LDAP 資料を参照してください。一例として、Windows 32 で z/OS Communications Server インストールに接尾部を追加する手順は次のとおりです。

   - http://[hostname]/ldap と入力して、Web ブラウザーで LDAP Administration インターフェースを開始します。ここで、hostname は、LDAP ディレクトリーがインストールされているマシンのホスト名です。「Administration logon」ウィンドウが表示されます。
   - 管理者ユーザー ID (例えば、cn=root 形式で) およびパスワードを入力します。
   - LDAP サーバーが実行していることを確認します。
   - 左側のナビゲーション・ペインで、「Settings」フォルダーをクリックしてから、「Suffixes」をクリックします。
   - 接尾部として使用される Base DN の名前を入力し (この例では、「c=US」)、「Update」をクリックします。
   - Base DN 接尾部が追加された後、LDAP サーバーをいったん停止してから、再始動します。

   これで、接尾部は LDAP システムに存在します。

   z/OS システムで、slapd.conf ファイルを更新して、新しい接尾部をシステムに取り込んでから、ネーム・サーバーを再始動してください。slapd.conf に追加する行は次のとおりです。

   ```
   suffix "c=US"
   ```

3. システム・ネーム・スペース・ルート・ノード (containerdn) を作成します。

   システム・ネーム・スペースのルート (containerdn と呼ばれるノード) を作成する ldif ファイルは、utils/namespace/dfhsns.ldif で CICS に付属しています。このファイルには、ご使用の環境に合わせて調整する方法を説明するコメントが含まれています。変更なしに使用される場合、ibm-

wsnTree=t1,o=wasnaming,c=US という containerdn を作成し、LDAP ネーム・スペースに 2 つの CICS ユーザーも作成します。最初の CICS ユーザーには、cn=CICSSystems,c=US という識別名があり、2 番目の CICS ユーザーは cn=CICSUser,c=US です。

2 つのユーザー ID が定義されます。これらのユーザー ID の使用方法を理解するには、440 ページの『セキュリティーの考慮事項』を参照してください。

ldapmodify コマンドが、選択した LDAP サーバーに使用可能でなければなりません。使用可能でない場合は、ご使用の LDAP サーバーの資料を参照して、システム・ネーム・スペースのルートの作成方法を判別してください。

LDIF ファイルは、次のように LDAP サーバーに適用できます。

```
ldapmodify-h <hostname>
        -p <portnumber>
        -D <authorized_principal>
        -w <authorized_principal_password>
        -f dfhsns.ldif
```

ここで、hostname と portnumber は、LDAP サーバーのものであり、許可されたプリンシパルは、項目を書き込むネーム・サーバー上で十分な権限を持つユーザーの識別名です。

具体的な例は次のとおりです。

```
ldapmodify-h wibble.example.com
        -p 389
        -D cn=admin
        -w adminpwd
        -f dfhsns.ldif
```

4. ネーム・スペース・ルート・ノード (noderootrdn) の下に legacyRoot ノードを作成します。

   ネーム・スペース内の legacyRoot ノードは、新しい InitialContext を作成するために呼び出されるときに、CICS がそれ自体を配置するように通常構成される場所です。このステップの場合、スクリプト DFHBuildSNS は、ディレクトリー utils/namespace で CICS に付属しています。

   構文は次のとおりです。

```
DFHBuildSNS   -ldapserver  <server_url>
             [-node        <node within the domain>]
              -domain      <domain_name>
              -containerdn <Root of the namespace>
              -principal   <principal authorised to write to the namespace>
              -credentials <password for that principal>
             [-force]
```

   例えば、次のとおりです。

```
DFHBuildSNS    -ldapserver ldap://wibble.example.com:389
               -domain PLEX2
               -containerdn ibm-wsnTree=t1,o=WASNaming,c=US
               -principal cn=admin
               -credentials adminpwd
```

   (*-force* オプションは *-node* フラグと一緒にしか使用されませんが、どちらも CICS 環境では使用されません。)

5. オプションとして、 441 ページの『CICS 領域レベルでのセキュリティー』で説明されている追加の手段を適用します。

このスクリプトを実行した後、JVM プロパティー・ファイルで必要なシステム・プロパティーの値を判別できます。それらの値を関連するすべての JVM プロパティー・ファイルに追加できます。

LDAP システム・プロパティーの値の判別:

これらのシステム・プロパティーは、JNDI に対する LDAP ネーム・スペースの使用に関連しています。

127 ページの『JVM システム・プロパティー』には、これらのシステム・プロパティーのそれぞれが詳しく記載されています。

- この LDAP ネーム・スペースをセットアップしたばかりの場合、使用した値が分かります。これらのうちのいくつかは、CICS プロパティーの設定に必要な値です。
- 既存のシステム・ネーム・スペースを使用または再使用する場合は、これらのプロパティーの適切な値を LDAP 管理者に要求してください。
- LDAP 管理者にアクセスできないか、値が使用不可である場合は、以下の情報を使用して、ネーム・スペースをブラウズすることによって判別できる場合があります。

  ネーム・スペースをブラウズすることによってセキュリティー・プリンシパルまたは資格情報を検出する可能性はほとんどありません。

-Dcom.ibm.cics.ejs.nameserver
: 構成される LDAP サーバーの URL です。 433 ページの『新規 LDAP サーバーの構成』の例では、*ldap://wibble.example.com:389* です。

-Dcom.ibm.ws.naming.ldap.containerdn
: `dfhsns.ldif` ファイルで指定された値です。 ldif ファイルを調整しない場合、デフォルトは *ibm-wsnTree=t1,o=WASNaming,c=US* です。既存のネーム・スペースをブラウズすることによってこの値を探す場合は、ibm-wsnTree タイプのノードを探してください。このノードへのパスが、containerdn の値である可能性があります。

-Dcom.ibm.ws.naming.ldap.noderootrdn
: DFHBuildSNS 呼び出しで指定したドメインから判別できます。この例では、noderootrdn は *ibm-wsnName=legacyRoot,ibm-wsnName=PLEX2,ibm-wsnName=domainRoots* です。既存のネーム・スペースをブラウズすることによってこの値を探す場合は、選択された containerdn から legacyRoot 項目までのパスを探します。

-Djava.naming.security.authentication
: バインド (または書き込み) を行うために CICS がそれ自体を LDAP に対して認証する必要がある場合は、*simple* に設定されます。提供されたスクリプトのデフォルトを使用すると、認証が必要です。これは、`dfhsns.ldif` スクリプトが ANYBODY グループのデフォルトの書き込みアクセス権限を除去し、作成した新しいプリンシパル `cn=CICSUser,c=US` に書き込みアクセス

権限を付与したからです。CICS が書き込むためにそれ自体を LDAP に対して認証する必要がない場合は、このシステム・プロパティーに値を設定しないでください。

重要: このシステム・プロパティーを指定する場合は、**-Djava.naming.security.principal** および **-Djava.naming.security.credentials** も指定する必要があります。これらのシステム・プロパティーは、CICS でセキュア LDAP サービスにアクセスする際に必要となるユーザー ID とパスワードを保持するため、これらのシステム・プロパティーを含むファイルに対して、ご使用のシステムで有効なアクセス制御に特に注意する必要があります。更新権限をシステム管理者のみに制限して、ファイルのセキュリティーを確保する必要があります。

-Djava.naming.security.principal
ネーム・スペースにバインドする権限を持つプリンシパルです。セキュリティーが実際の問題ではない場合、ネーム・スペース全体への書き込みアクセス権限を持つシステム・プリンシパルを選択できます。ただし、少なくとも `dfhsns.ldif` で指定される *cn=CICSUser,c=US* 識別名を使用するようにお勧めします。これは、その ID は、LDAP ネーム・スペースの特定の領域 (containerdn 以下) にしか書き込めないからです。

より厳重なセキュリティーが必要な場合は、プリンシパルを *cn=CICSSystems,c=US* にすることができます。この ID を使用する場合は、追加の LDAP 構成を実行する必要があります。CICS LDAP セキュリティー構成について詳しくは、440 ページの『セキュリティーの考慮事項』を参照してください。

-Djava.naming.security.credentials
プリンシパルのパスワードです。`dfhsns.ldif` を調整しなかった場合のデフォルトは *secret* です。

これらのシステム・プロパティーの値を判別したら、CORBA アプリケーションまたはエンタープライズ Bean で使用されるすべての JVM プロファイルまたはオプションの JVM プロパティー・ファイルでそれらの値を指定します。

特に、DFHJVMCD JVM プロファイルまたは参照されるプロパティー・ファイルで指定してください。DFHJVMCD プロファイルは、デフォルトの要求プロセッサー・プログラムや、デプロイ済み JAR ファイルの公開と撤回に CICS が使用するプログラムを含めて、CICS 定義のプログラムで使用されます。

また、これらのシステム・プロパティーは、CORBA ステートレス・オブジェクトまたはエンタープライズ Bean に使用することを選択した他の JVM プロファイルで参照される、JVM プロファイルまたはプロパティー・ファイルでも指定する必要があります。これらのプロファイルは、CICS 提供のサンプル JVM プロファイルまたは独自の JVM プロファイルの場合があります。CORBA ステートレス・オブジェクトおよびエンタープライズ Bean の場合、JVM プロファイルは、要求プロセッサー・プログラムの PROGRAM リソース定義で指定されます。

LDAP ネーム・スペース構造

WebSphere Application Server Version 4 for z/OS で使用される LDAP ネーム・スペース構造は、CICS 環境で使用するのに便利な構造です。

注: WebSphere Application Server バージョン 5 以降は、デフォルトで COS ネーム・サーバーを使用し、WebSphere Application Server バージョン 4 との後方互換性に対してのみ LDAP をサポートします。

LDAP ネーム・スペース構造には、WebSphere で使用される 2 つの重要なノード (コンテナー・ルートとレガシー・ルート) があります。

コンテナー・ルート:

コンテナー・ルートは、ibm-wsnTree タイプのノードです。デフォルトで、これは *ibm-wsnTree=t1, o=wasnaming, c=us* と呼ばれます。ただし、これは、WebSphere に付属の *bboldif.cb* ファイルを変更することによってカスタマイズ可能です。

レガシー・ルート:

レガシー・ルートは、コンテナー・ルートの下にある、*ibm-wsnName* タイプのノードです。

この標準的な名前は *ibm-wsnName=legacyRoot,ibm-wsnName=PLEX2,ibm-wsnName=domainRoots,ibm-wsnTree=t1,o=WASNaming,c=us* です。legacyRoot および domainRoots という名前は固定されています。唯一の変数はミドルネーム (この例では PLEX2) です。

複数の legacyRoot ノードが可能であり、各ノードには別々の名前があります。これらはそれぞれ「ドメイン」です。WebSphere Application Server for z/OS 構成では、ドメインは sysplex にマップされます。WebSphere Application Server for z/OS がインストールされるときに、カスタマイズ・ダイアログに sysplex 名が入力されるときに構成されます。

ドメイン:

ドメインには複数のサーバーが入っています。

WebSphere Application Server for z/OS では、各サーバーには、legacyRoot の下にノードがあります。例えば、BBOSV1 と呼ばれるサーバーには、レガシー・ルートを基準にした名前 *ibm-wsnName=BBOSV1,ibm-wsnName=PLEX2* があります。公開されるオブジェクトはこのノードの下にあります。

WebSphere と同じ LDAP サーバーを使用するように CICS が構成される場合、各 CICS CorbaServer には、legacyRoot のすぐ下にノードがあります。したがって、CorbaServer に CICS1 という JNDI 接頭部がある場合、レガシー・ルートを基準にしたノード *ibm-wsnName=CICS1* があり、CICS はこのノードの下にある CorbaServer のオブジェクトを公開します。WebSphere Application Server for z/OS または上記のように構成された CICS で、新しい InitialContext が作成される場合、その InitialContext は legacyRoot ノードに基づいています。これにより、CICS 内のエンタープライズ Bean が、WebSphere によって公開されるオブジェクトを検索したり、WebSphere 内のエンタープライズ Bean またはサーブレットが、CICS によって公開されるオブジェクトを検索したりするのが容易になります。

注: CICS 領域の初期 JNDI コンテキスト (通常、legacyRoot ノード) の下にある任意の JNDI サブコンテキストは、一時的である場合があります。CICS に初期コンテキスト・ノードへの書き込みアクセス権限がある場合、これが当てはまります。

CorbaServer の JNDI サブコンテキストは、CORBASERVER 定義の JNDIPREFIX オプションで指定されます。エンタープライズ Bean が CorbaServer から公開される場合、CICS はサブコンテキストを作成します (必要な書き込み権限があり、サブコンテキストがネーム・スペース構造にまだ存在しない場合)。ただし、CorbaServer 内のすべてのエンタープライズ Bean が撤回されると、CICS はネーム・スペース構造からサブコンテキストを削除する場合があります。複数の CorbaServer が接頭部階層の一部を共用する場合、CICS は、それらのいずれかが引き続き使用中のコンテキストを決して除去しません。しかし、接頭部内のコンテキストが空である場合は、初期コンテキストまでさかのぼって除去されます。

サブコンテキスト階層の最上位ノードが削除されないようにしたい場合は、初期コンテキスト・ノードへの書き込みアクセス権限を CICS に与えないでください (つまり、サブコンテキストの最上位ノードを手動で作成する必要があります)。サブコンテキスト階層の複数の上位を保護したい場合は、下位のみへの書き込み権限を CICS に与えます (つまり、サブコンテキストの上位ノードを手動で作成する必要があります)。詳しくは、441 ページの『CICS 領域レベルでのセキュリティー』を参照してください。

分散プラットフォーム用の WebSphere Application Server のバージョンには、同じようなドメインの概念がありますが、その概念は sysplex に関連していません。

Nodes (ノード):

もう 1 つの概念、つまり***ノード*** という概念があります。ドメインは複数のノードを表します。ドメイン名ではなく、ノード名を知っていれば、ドメインにまでナビゲートすることができます。したがって、ノードは一種のドメインの別名です。

ノードは、分散プラットフォーム用の WebSphere Application Server のバージョンで使用されますが、WebSphere Application Server for z/OS では使用されません。CICS では使用されません。しかし、CICS で使用するために新しい LDAP サーバーをセットアップする場合、ノードをサポートする構造の一部が作成されます。WebSphere Application Server for z/OS はノードを使用しないため、nodename は DFHBuildSNS ユーティリティーに対するオプション・パラメーターであり、このユーティリティーは CICS の下でシステム・ネーム・スペースを作成します。

セキュリティーの考慮事項: CICS が LDAP に書き込むためにそれ自体を LDAP に対して認証する必要があることを指定した場合、JVM プロファイルでシステム・プロパティー **-Djava.naming.security.authentication=***simple* をコーディングすることによって、次のどちらかを選択できます。

- 441 ページの『containerdn レベルでのセキュリティー』、または
- 441 ページの『CICS 領域レベルでのセキュリティー』.

この決定に役立つように、LDAP ネーム・スペースの一部のごく簡単なビューを441 ページの図 37 に示しています。

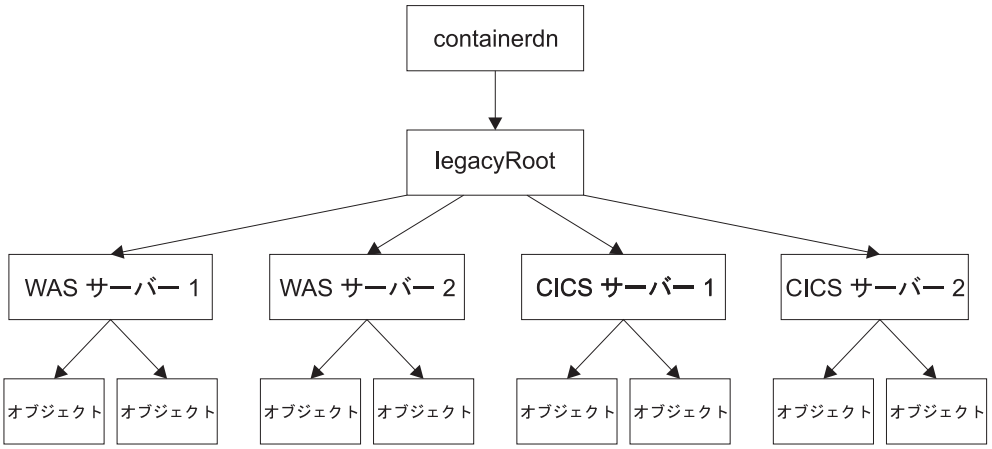


図 37. LDAP ネーム・スペースの一部の簡単なビュー

containerdn レベルでセキュリティーを使用する場合、CICS には、containerdn およびその下にあるすべてのノードへの書き込みアクセス権限があります。これにより、CICS、または JNDI インターフェースを使用する CICS アプリケーションが、WebSphere Application Server for z/OS に属するノードを含めて、これらのすべてのノードに書き込むことができます。CICS 領域レベルでセキュリティーを使用する場合、CICS および CICS アプリケーションは、ツリー内の特定の CICS ノードにしか書き込みできません。

containerdn レベルでのセキュリティー: containerdn レベルでセキュリティーを使用するには、*dfhsns.ldif* ファイルによって作成される CICS 管理プリンシパル (cn=CICSUser,c=us) を使用します (ステップ 3 「システム・ネーム・スペース・ルート・ノードを作成します」を参照してください)。containerdn ノードへのアクセス権限を作成時にこのプリンシパルに付与します。このユーザー ID とパスワードが、JVM プロパティー・ファイルのシステム・プロパティー **-Djava.naming.security.principal** および **-Djava.naming.security.credentials** に表示されていることを確認してください。

CICS 領域レベルでのセキュリティー:

containerdn ノードへのアクセス権限を作成時にこのプリンシパルに付与します。このユーザー ID とパスワードが、JVM プロパティー・ファイルのシステム・プロパティー **-Djava.naming.security.principal** および **-Djava.naming.security.credentials** に表示されていることを確認してください。

CICS 領域レベルでセキュリティーを使用するには、*dfhsns.ldif* ファイルによって作成される CICS 実行時プリンシパル (cn=CICSSystems,c=US) を使用します。ステップ 3 「システム・ネーム・スペース・ルート・ノードを作成します」を参照してください。これには、いくつかの追加ステップが必要です。このユーザー ID とパスワードが、JVM プロパティー・ファイルのシステム・プロパティー **-Djava.naming.security.principal** および **-Djava.naming.security.credentials** に表示されていることを確認してください。さらに、CICS に legacyRoot への書き込みアクセス権限がないので、CICS は、独自のノード (図 37 で CICS サーバー 1 と呼ばれる) を作成できません。したがって手動で作成してから、このノードへの書き込みアクセス権限を CICS 実行時プリンシパル (cn=CICSSystems,c=US) に付与する必要があります。これについて以下で説明しています。

この方法で CICS 領域を構成してから、新しいサブコンテキストを使用するには、次の手順を実行します。

- *cicsabcd* と呼ぶ適切なサブコンテキストを選択します。
- CICS システムで使用するために legacyRoot の下にそのサブコンテキストを作成します (『サブコンテキストの作成』を参照)。
- CICS 実行時プリンシパルがそのサブコンテキストに確実に書き込めるようにします。
- 領域で使用中の JVM プロパティー・ファイルでシステム・プロパティー **-Djava.naming.security.principal** および **-Djava.naming.security.credentials** を使用して、CICS 実行時プリンシパルおよび資格情報を指定します。
- CICS 領域で作成された任意の CORBASERVER 定義に、*cicsabcd* から始まる JNDIPREFIX 属性があることを確実にします。これは、公開する参照が、legacyRoot の下にある新しいサブコンテキスト *cicsabcd* の下で 公開されることを意味します。

これでセキュリティー構成が完了しました。LDAP ネーム・スペースをブラウズするユーザーは、legacyRoot の下でこのコンテキスト *cicsabcd* を見つけ、それを CORBASERVER 定義に関連付けることができます。

サブコンテキストの作成: LDAP ネーム・スペース内の legacyRoot の下でサブコンテキスト *cicsabcd* を作成し、それに適切なアクセス制御リスト (ACL) を設定するには、*utils/namespace/dfhNewCICSSubcontext.ldif* で CICS に提供される LDIF ファイルを使用してください。

- LDIF ファイルには、必要なステップ、および特定の LDAP システム・ネーム・スペース構成に対する変更を必要とする可能性がある値を説明するコメントが入っています。
- LDIF ファイルは、ldapadd コマンドを使用して LDAP サーバーに適用できます。

```
Ldapadd -h wibble.example.com
        -p 389
        -D cn=CICSUser,c=us
        -w CICSUserpwd
        -f dfhNewCICSSubcontext.ldif
```

  ここで、`CICSUserpwd` は、`CICSuser` がセットアップされたときに設定された `CICSuser` のパスワードです。

  このコマンドは、legacyRoot ノードに書き込むことができるプリンシパル (および資格情報) を指定して実行する必要があります。ここで使用している例では、この目的用に作成された *cn=CICSUser,c=US id* です。

- LDIF ファイルの中で変更が必要な最も重要な行は、作成されるノードの識別名です。LDAP システム・ネーム・スペースが、CICS に付属のすべてのデフォルト・スクリプトを使用して構成されたことを前提とすると、識別名は次のとおりです。

```
ibm-wsnName=cicsabcd,ibm-wsnName=legacyRoot,ibm-wsnName=PLEX2,
ibm-wsnName=domainRoots,ibm-wsnTree=t1,o=wasnaming,c=US
```

- LDIF の残りの部分は、新しいノードに対して適宜、アクセス制御リストを設定します。

- この LDIF ファイル内のコメントが重要です。これらのコメントは、検討が必要な可能性があるその他の項目について説明しています。例えば、どのプリンシパルに現在システム・ネーム・スペースへの書き込みアクセス権限があるかによって、ご使用のシステムで適切な追加の ACL 項目がある可能性があります。
- LDIF が適用された後、新しいノードが legacyRoot の下の LDAP サーバーに存在し、CICS 実行時プリンシパルが書き込みアクセス権限を持つようにアクセス制御リストが設定されます。

その他の考慮事項: 以下の項目を考慮する必要がある場合があります。

- 異なる領域に別々の複数の CICS 実行時プリンシパルを作成して、各プリンシパルに付与されるアクセス権限の有効範囲を減らすことができます。
- 既存のシステム・ネーム・スペース内でこのプロセスを使用する場合、使用中の他のプリンシパル (および資格情報) がある可能性があります。*dfhNewCICSSubcontext* によって作成される新しいサブコンテキストへの書き込みアクセス権限をそれらに付与する必要があります。 dfhNewCICSSubContext LDIF ファイル内のコメントは、これが該当するかどうかを確認する方法、および ldapadd を実行する前に LDIF ファイルを適宜調整する方法について説明しています。

COS Naming Directory Server のセットアップ

COS Naming Directory Server をセットアップする最も便利な方法は、外部の Windows マシンで実行中の IBM WebSphere Application Server を使用することです。

このタスクについて

COS Naming Directory Server をセットアップする最も便利な方法は、外部の Windows マシンで実行中の IBM WebSphere Application Server を使用することです。それに付属のインストール手順に従ってください。

IIOP 用の TCP/IP のセットアップ

IIOP 要求を受け入れて送信する TCP/IP リスナーとして CICS 領域を構成するには、CICS で以下の定義を行う必要があります。

このタスクについて

1. このリスナーが必要なすべての CICS 領域の CICS 開始ジョブ・ストリームで、次のシステム初期設定パラメーターを設定します。
   - **IIOPLISTENER** を **YES** に設定します
   - **TCPIP** を **YES** に設定します
2. 以下の項目を指定して、リスナーがモニターするすべてのポートに対してリスナー領域で TCPIPSERVICE リソース定義を定義し、インストールします。
   - PROTOCOL(IIOP)
   - CICS が着信 IIOP 要求を listen するポートまたは IP アドレス。

     注: SSL 接続が失敗する場合、一部のクライアントは関連した非 SSL ポートで再試行しようとします。CICS TS はこのポートを SSL port–1 に定義します。このポート (SSL port–1) が他の目的に定義されていないことを確認します。ウェルノウン IIOP ポートは 683 (非 SSL) および 684 (SSL) です。

- 要求の着信時に開始する CICS トランザクション。IIOP サービスの場合、これは、CICS IIOP 要求受信側 CIRR に設定されなければなりません。
- 使用する Secure Sockets Layer (SSL) 認証のレベル。
- DNS 接続最適化が使用される場合は、DNSGROUP 名。426 ページの『DNS 接続最適化に対するリソース定義』を参照してください。
- セキュリティーまたはワークロード管理のためにこの要求を CICS USERID に関連付けるために呼び出される、ユーザー置換可能プログラムの名前。これが省略される場合、ユーザー置換可能プログラムは呼び出されません。ユーザー置換可能プログラムのサンプル DFHXOPUS が提供されています。454 ページの『IIOP ユーザー置換可能セキュリティー・プログラムの使用』を参照してください。

例えば、次のとおりです。

```
DEFINE TCPIPSERVICE(IIOPNSSL) GROUP(DFH$IIOP)
     DESCRIPTION(IIOP TCPIPSERVICE with no SSL support)
     URM(DFHXOPUS)          BACKLOG(10)            PORTNUMBER(683)
     TRANSACTION(CIRR)      SSL(NO)
     STATUS(CLOSED)         PROTOCOL(IIOP)
```

重要: 複数領域サーバーでは、TCPIPSERVICE 定義が論理サーバーのすべての領域 (リスナーと AOR の両方) にインストールされなければなりません。リスナー領域では、IIOPLISTENER システム初期設定パラメーターが「YES」に設定されなければなりません。AOR では、「NO」に設定されなければなりません。リスナー/AOR の結合では、「YES」に設定されなければなりません。

TCPIPSERVICE リソース定義の完全な構文については、「Resource Definition Guide」の『TCPIPSERVICE リソース』を参照してください。

DNS による接続の最適化の使用:

IIOP で DNS による接続の最適化を使用するには、IIOP TCPIPSERVICE リソース定義で DNSGROUP 名を定義する必要があります。

同じ TCPIPSERVICE を提供し、同じ DNSGROUP 名を持つすべての CICS 領域は、同じサービスを必要とするクライアント要求の候補として、同じ *group-name* で MVS ワークロード管理 (WLM) に登録されます。また、この登録には、TCP/IP 機能 **gethostbyaddr** によって取得される領域のホスト名、および APPLID システム初期設定パラメーターによって指定されるとおりに領域の特定の APPLID から CICS が取得する固有のサーバー名 も含まれます。

MVS ワークロード・マネージャーが使用するように構成されるのと同じ、z/OS 上の DNS ネーム・サーバーと対話するように、リスナー領域が構成される必要があります。 CICS 開始 JCL の SYSTCPD DD ステートメントを提供することによって、TCP/IP で使用されるネーム・サーバーを定義できます。「*CICS Transaction Server for z/OS インストール・ガイド*」 Enabling TCP/IP in a CICS region を参照してください。

注:

1. クライアントと CICS サーバーの両方が、同じ TCP/IP ネーム・サーバーを使用する必要があります。

2. ネーム・サーバーは、逆検索を実行できなければなりません。つまり、サーバーの IP アドレスを完全なホスト名に変換できなければなりません。

IIOP 用の CICS のセットアップ

IIOP をサポートするには、CICS 開始ジョブ・ストリームを定義し、いくつかの CICS リソースを定義し、インストールする必要があります。

CICS 開始ジョブ・ストリームの定義:

IIOP をサポートする CICS 領域の始動ジョブ・ストリームで、パラメーターを定義する必要があります。

JCL パラメーター

REGION
最小 1000M をお勧めします。

CICS システム初期設定パラメーター

EDSALIM
最小 500M をお勧めします。

IIOPLISTENER

- CICS 領域が IIOP リスナー領域であるか、またはリスナーとアプリケーション専有領域 (AOR) の結合である場合は、IIOPLISTENER=YES を指定します。
- CICS 領域が IIOP アプリケーション専有領域である場合は、IIOPLISTENER=NO を指定します。PROTOCOL(IIOP) を指定する、領域にインストールされる TCPIPSERVICE 定義はオープンできません。

JVMPROFILEDIR
アプリケーションに使用する JVM プロファイルが入っている z/OS UNIX ディレクトリーに設定されます。この方法については 87 ページの『JVM プロファイルのロケーションの設定』 で説明しています。

KEYRING
RACF に対して登録されている証明書で Secure Sockets Layer (SSL) 認証を使用する場合は、必須です。

MAXJVMTCBS
CICS 領域がサポートできる JVM 数を指定します。 187 ページの『パフォーマンスに関する JVM プールの管理』 に、**MAXJVMTCBS** システム初期設定パラメーターの適切な設定値を算出する方法が記述されています。

TCPIP YES に設定されます。

CICS データ・セットの DD ステートメント
サンプルのローカル VSAM データ・セット定義は、CICS 提供の RDO グループ DFHEJVS で提供されます。これらのデータ・セットは、UPDATE アクセスに対して RACF で許可されなければなりません。「*CICS RACF Security Guide*」の Authorizing access to CICS data sets を参照してください。

DFHEJDIR

要求ストリーム・ディレクトリーを含む、リカバリー可能な共用ファイル。このファイルは、VSAM ファイルの場合とカップリング・ファシリティー・データ・テーブルの場合があります。CICS では、このファイルを SDFHINST ライブラリーの DFHDEFDS メンバーに作成するためのサンプル JCL が提供されています。

注: 大部分の場合、提供された JCL の RECORDSIZE パラメーターには変更は不要です。ただし、論理 EJB/CORBA サーバーに 40 を超える CorbaServer をインストールしようとする場合は、『DFHEJDIR および DFHEJOS の RECORDSIZE の指定』 を参照してください。

DFHEJOS

CorbaServer がインストールされるときに、不動態化されたステートフル・セッション Bean を保管するために CICS で使用される、リカバリー不能共用ファイル。このファイルは、VSAM ファイルの場合とカップリング・ファシリティー・データ・テーブルの場合があります。CICS では、このファイルを SDFHINST ライブラリーの DFHDEFDS メンバーに作成するためのサンプル JCL が提供されています。

注: 大部分の場合、提供された JCL の RECORDSIZE パラメーターには変更は不要です。ただし、論理 EJB/CORBA サーバーに 40 を超える CorbaServer をインストールしようとする場合は、『DFHEJDIR および DFHEJOS の RECORDSIZE の指定』 を参照してください。

DFHEJDIR および DFHEJOS の RECORDSIZE の指定:

CICS EJB/CORBA 論理サーバーに対して定義できる CorbaServer の最大数は、要求ストリーム・ディレクトリー・ファイル DFHEJDIR および EJB オブジェクト・ストア・ファイル DFHEJOS の RECORDSIZE 値で制御されます。

DFHEJDIR に提供される JCL および FILE 定義の RECORDSIZE 属性は、RECORDSIZE を 1017 バイトに指定します。DFHEJOS に提供される JCL および FILE 定義の RECORDSIZE 属性は、RECORDSIZE を 8185 バイトに指定します。通常、これらの値には変更は不要です。論理 EJB/CORBA サーバーに 40 を超える CorbaServer をインストールしようとする場合のみ、これらの値を変更する必要があります。

DFHEJDIR と DFHEJOS の両方には制御レコードが含まれています。この制御レコードは、24 バイト・ヘッダー、および CorbaServer 制御フィールド (各 24 バイト長) の反復グループで構成されます。DFHEJDIR のデフォルト長 1017 は、論理サーバーを 41 の CorbaServer に事実上制限します。つまり、(1 + 41) * 24 = 1008 バイトです。これより多くの CorbaServer を論理サーバーにインストールする必要がある場合は、次のように DFHEJDIR に必要な RECORDSIZE を計算してください。

1. 必要な CorbaServer 数に 24 を乗算します。
2. 制御レコード・ヘッダーの 24 バイトを加算します。これにより、絶対最小レコード・サイズが得られます。
3. 最終の値を次の 512 の倍数に切り上げて、最小制御インターバル・サイズを算出します。

4. 7 を減算して、RECORDSIZE パラメーターの値を取得します。

DFHEJOS の RECORDSIZE 値を DFHEJDIR の RECORDSIZE 値より大きくしてください。長さが短すぎると、Bean を不動態化するとき衝突が生じます (提供された定義では、DFHEJOS の RECORDSIZE が DFHEJDIR の RECORDSIZE のほぼ 8 倍になります)。

注: DFHEJDIR および DFHEJOS のサンプル JCL は、SDFHINST ライブラリーの DFHDEFDS メンバーにあります。DFHEJDIR および DFHEJOS のサンプルの FILE リソース定義は、DFHEJVS RDO グループ内にあります。サンプルのカップリング・ファシリティー FILE 定義は DFHEJCF グループにあり、サンプルの VSAM RLS FILE 定義は DFHEJVR グループにあります。

CICS リソースの定義:

エンタープライズ Bean に必要な CICS リソースを作成する必要があります。

FILE

CICS で必要な次のファイルに FILE リソース定義を提供し、インストールします。

「EJB ディレクトリー」、DFHEJDIR
要求ストリーム・ディレクトリーが入っているファイル。このディレクトリーは、エンタープライズ Bean と CORBA ステートレス・オブジェクトの両方に対するメソッド要求のルーティングで使用されます。DFHEJDIR をリカバリー可能として定義する必要があります。

「EJB オブジェクト・ストア」、DFHEJOS
不動態化されているステートフル・セッション Bean のファイル。CorbaServer のインストール時にも使用されます。リカバリー不能として定義する必要があります。

単一領域 CICS EJB/CORBA サーバーでは、DFHEJDIR および DFHEJOS をローカル・ファイルとして定義することが受け入れられます。ただし、複数領域 CICS EJB/CORBA サーバーでは次のとおりです。

- DFHEJDIR は、サーバー内のすべての領域 (リスナーおよび AOR) で共用されなければなりません。
- DFHEJOS は、サーバー内のすべての AOR で共用されなければなりません。

DFHEJDIR および DFHEJOS を複数の領域間で共用できるようにするには、次のいずれかの方法で定義できます。

- ファイル専有領域 (FOR) 内のリモート・ファイルとして定義する
- カップリング・ファシリティー・データ・テーブルとして定義する
- VSAM RLS を使用して定義する

CICS 提供 RDO グループ DFHEJVS には、DFHEJDIR および DFHEJOS のサンプル FILE 定義があります。CICS 提供 RDO グループ DFHEJCF には、DFHEJDIR および DFHEJOS のサンプル・カップリング・ファシリティー FILE 定義があります。CICS 提供 RDO グループ DFHEJVR には、DFHEJDIR および DFHEJOS のサンプル VSAM RLS FILE 定義があります。
(DFHEJVS、DFHEJCF、および DFHEJVR は、デフォルトの CICS 開始グループ・リスト DFHLIST に含まれません。)

注: 大部分の場合、提供された FILE 定義の RECORDSIZE 属性の値には変更は不要です。ただし、論理 EJB/CORBA サーバーに 40 を超える CorbaServer をインストールしようとする場合は、 446 ページの『DFHEJDIR および DFHEJOS の RECORDSIZE の指定』 を参照してください。

FILE 定義に関する参照情報については、FILE resources を参照してください。

TRANSACTION および PROGRAM

CORBA ステートレス・オブジェクトおよびエンタープライズ Bean には、独自の PROGRAM リソース定義がありません。CORBA ステートレス・オブジェクトまたはエンタープライズ Bean に関連した PROGRAM リソース定義は、要求プロセッサー・プログラムのものです。

CICS 提供の要求受信側プログラムおよび要求プロセッサー・プログラムに必要なデフォルトの TRANSACTION および PROGRAM 定義は、デフォルトの CICS 開始グループ・リスト DFHLIST に含まれているリソース・グループ DFHIIOP にあります。

通常、要求受信側のデフォルトの TRANSACTION および PROGRAM 定義 (それぞれ、CIRR および DFHIIRRS) を置き換える必要はありません。 DFHIIOP における CIRR の定義は次のとおりです。

```
DEFINE TRANSACTION(CIRR)      GROUP(DFHIIOP)
       PROGRAM(DFHIIRRS)      TWASIZE(0)
       PROFILE(DFHCICST)      STATUS(ENABLED)
       TASKDATALOC(ANY)       TASKDATAKEY(USER)
       RUNAWAY(SYSTEM)        SHUTDOWN(ENABLED)
       PRIORITY(1)            TRANCLASS(DFHTCL00)
       DTIMOUT(NO)            TPURGE(NO)
       SPURGE(YES)            ISOLATE(NO)
       RESSEC(NO)             CMDSEC(NO)
       RESTART(NO)
       DESCRIPTION(Default CICS IIOP Request Receiver transaction)
```

要求プロセッサー・プログラムに独自の TRANSACTION および PROGRAM 定義を作成する 1 つの理由は、デフォルト以外の JVM プロファイルを指定することです。使用される JVM プロファイルの名前は、要求プロセッサー・プログラムの PROGRAM 定義の JVMPROFILE オプションで指定されます。要求プロセッサーのデフォルトの PROGRAM 定義 (DFHIIOP 内の DFJIIRP) は、JVM プロファイル DFHJVMCD を指定します。DFHIIOP における DFJIIRP の定義は次のとおりです。

```
DEFINE PROGRAM(DFJIIRP)       GROUP(DFHIIOP)
     DESCRIPTION(CICS IIOP Request Processor)
     JVM(YES)
     JVMCLASS(com.ibm.cics.iiop.RequestProcessor)
     JVMPROFILE(DFHJVMCD)
     LANGUAGE(LE370)
     RELOAD(NO)
     EXECKEY(USER)
     RESIDENT(NO)
     USAGE(NORMAL)
     USELPACOPY(NO)
     STATUS(ENABLED)
     CEDF(NO)
     DATALOCATION(ANY)
     DYNAMIC(NO)
```

注: デバッグのために、CEDF 属性を YES に設定できます。 350 ページの『エンタープライズ Bean での EDF の使用』を参照してください。

要求プロセッサーに独自の PROGRAM 定義を作成する場合、任意の名前をその定義に指定できますが、JVMCLASS パラメーターは **com.ibm.cics.iiop.RequestProcessor** に設定されなければなりません。要求プロセッサーが使用する別の JVM プロファイルを選択し、JVMPROFILE オプションで JVM プロファイルの名前を指定してください。CICS は、/usr/lpp/cicsts/cicsts42/JVMProfiles z/OS UNIX ディレクトリーにサンプル JVM プロファイルを提供します。ここで、/usr/lpp/cicsts/cicsts42 は、z/OS UNIX 上の CICS ファイルのインストール・ディレクトリーです。 102 ページの『プールされた JVM のセットアップ』は、JVM プロファイルを見つけて、選択し、カスタマイズする方法を示しています。

TCPIPSERVICE

IIOP 要求を受け取り、IIOP *要求受信側* を呼び出すように CICS リスナーを構成するために、TCPIPSERVICE リソース定義を提供し、インストールします。TCPIPSERVICE リソース定義は、ロード・バランシングとセキュリティーのオプションも指定します。 443 ページの『IIOP 用の TCP/IP のセットアップ』を参照してください。

CICS は、リソース・グループ DFH$EJB で、EJB インストール検査プログラム (IVP) と EJB「Hello World」サンプル・アプリケーションで使用するための TCPIPSERVICE 定義を提供します。 CICS EJB サーバーをセットアップする場合は、 282 ページの『CICS で必要なアクション』にある、この定義の構成方法についてステップバイステップの例に従うようにお勧めします。

CORBASERVER

CORBASERVER リソース定義を提供し、インストールします。CORBASERVER をインストールする前に、DFHEJDIR ファイルが定義され、インストールされ、使用可能でなければならないことに注意してください。

CICS は、リソース・グループ DFH$EJB で、EJB IVP プログラムと EJB「Hello World」サンプル・アプリケーションで使用するための CORBASERVER 定義を提供します。 CICS EJB サーバーをセットアップする場合は、 282 ページの『CICS で必要なアクション』にある、この定義の構成方法についてステップバイステップの例に従うようにお勧めします。

REQUESTMODEL

要求受信側 が着信要求を CICS トランザクションと突き合わせて、要求を処理するために新しい要求プロセッサー・インスタンスが作成される場合に使用される、実行パラメーターを定義できるようにするために、REQUESTMODEL リソース定義を提供し、インストールします。REQUESTMODEL 定義のデフォルトの TRANSID は CIRP であり、これは、デフォルトの要求プロセッサー・プログラム DFJIIRP を指定します。独自の TRANSACTION 定義を使用することを選択する場合は、それを定義し、インストールする必要があります。JVMCLASS パラメーターを **com.ibm.cics.iiop.RequestProcessor** に設定して、PROGRAM 定義を指定する必要があります。 455 ページの『CICS TRANSID の取得』を参照してください。

注:

1. REQUESTMODEL 定義を提供する必要があるのは、デフォルトの TRANSID である CIRP が不適切な場合、または IIOP ワークロードをトランザクション ID によって分けたい (例えば、モニターのために) 場合のみです。

2. CIRP の TRANSACTION 定義は、DYNAMIC(NO) を指定します。エンタープライズ Bean および CORBA ステートレス・オブジェクトに対するメソッド要求の動的ルーティングを使用したい場合は、DYNAMIC(YES) を指定する 1 つ以上の TRANSACTION 定義を提供し、REQUESTMODEL 定義でそれらの定義を指定する必要があります。
3. CorbaServer が作動可能になった後、CREA CICS 提供トランザクションを使用すると、CorbaServer 内の特定のエンタープライズ Bean および Bean メソッドに関連したトランザクション ID を表示できます。トランザクション ID を変更し、変更を適用し、新しい REQUESTMODEL 定義に対する変更を保管できます。これは、手動で REQUESTMODEL 定義を作成するよりも簡単な方法です。
4. 複数領域 CICS 論理サーバーでは、AOR とともにリスナー領域にも REQUESTMODEL 定義をインストールすることをお勧めします。 451 ページの図 38 を参照してください。ローカル・オブジェクトに対するアウトバウンド要求用に AOR 内の REQUESTMODEL 定義を必要としています。CORBA ステートレス・オブジェクトまたはエンタープライズ Bean が別のオブジェクトを呼び出した場合、そのオブジェクトがローカル AOR 内で使用可能であれば、CICS はリスナー領域に要求を送信しません。その代わりに、CICS は現行タスク内で呼び出されたメソッドを実行するか (「緊密ループバック」)、ローカル AOR 内で別の要求プロセッサーを開始します (「通常ループバック」)。通常ループバックを使用する場合は、新規の要求プロセッサー・タスクでも、最初のオブジェクトの呼び出しに使用されたものと同じ REQUESTMODEL が使用されることが望まれます。そうでないと、予測不能な結果が発生する可能性があります。CORBA ステートレス・オブジェクトまたはエンタープライズ Bean がアウトバウンド・コールを行わない場合、AOR 内の REQUESTMODEL は厳密には必要とされません。

DJAR

任意のエンタープライズ Bean の DJAR リソース定義を提供し、インストールします。

注: DJAR 定義は、通常、CICS スキャン・メカニズムによって作成され、インストールされます (「Resource Definition Guide」の『DJAR リソース』を参照)。

451 ページの図 38 は、CICS 論理サーバーの定義に必要な RDO 定義を示しています。この図は、リスナー領域で必要な定義、AOR で必要な定義、および両方で必要な定義を示しています。

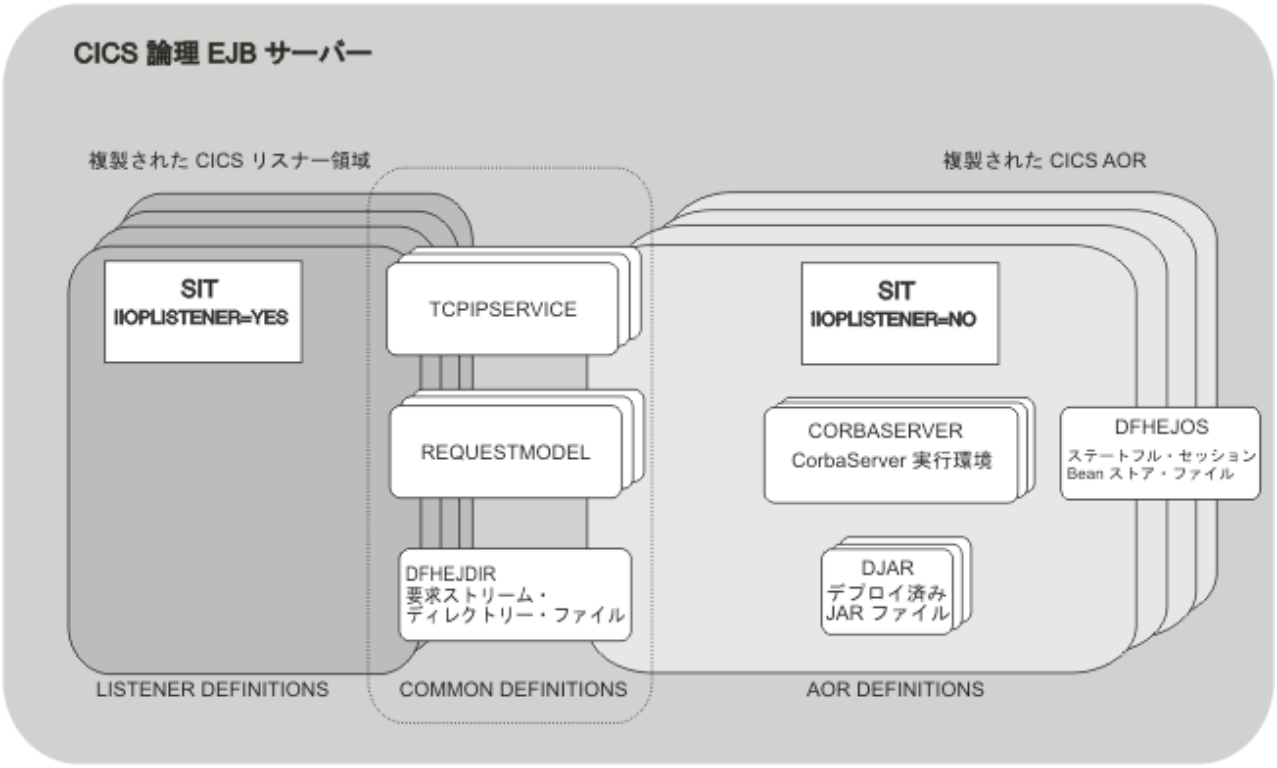


図 38. CICS 論理サーバー内のリソース定義： この図は、リスナー領域で必要な定義、AOR で必要な定義、および両方で必要な定義を示しています。

IIOP 要求の処理

CICS 要求受信側は、ターゲット・メソッドを呼び出すために IIOP 要求プロセッサーに制御を渡す前に、要求の CICS 実行パラメーターを設定する CICS USERID および TRANSID を派生させます。

CICS ユーザー ID の取得

IIOP 要求の場合、次の方法でユーザーを認証し、識別することができます。

このタスクについて

1. Secure Sockets Layer (SSL) クライアント認証を使用します。詳しくは、「*CICS RACF Security Guide*」を参照してください。
2. SSL 認証がユーザー ID を提供しない場合、IIOP ユーザー置換可能セキュリティー・プログラムを使用して提供できます。ポート用の TCPIPSERVICE 定義の URM 属性で、IIOP セキュリティー・プログラムの名前を指定してください。詳しくは、454 ページの『IIOP ユーザー置換可能セキュリティー・プログラムの使用』を参照してください。
3. 上記のどちらのメカニズムもユーザー ID を提供しない場合は、デフォルト・ユーザー ID が使用されます。

TCPIPSERVICE 定義でセキュリティー・プログラムの名前を指定するときに、その PROGRAM リソース定義を省略すると、CICS はリソース定義を作成しようとしま

す (自動インストール)。これが失敗するか、セキュリティー・プログラムが USERID を戻さない場合、CICS は SSL クライアント証明書 (ある場合) に関連したユーザー ID を使用します。これらからユーザー ID を提供しない場合は、デフォルト・ユーザー ID が使用されます。

以下の通信域がユーザー置換可能プログラムに渡されます。この構造は、OMG Web サイト (http://www.omg.org/library) から取得可能な「*The Common Object Request Broker: Architecture and Specification*」で定義される IIOP メッセージのフォーマットに基づいています。

オフセット 16 進	タイプ	長さ	名前
(0)	STRUCTURE	80	sXOPUS
(0)	CHARACTER	4	standard_header
(4)	FULLWORD	4	pIIOPData
(8)	FULLWORD	4	lIIOPData
(C)	FULLWORD	4	pRequestBody
(10)	FULLWORD	4	lRequestBody
(14)	CHARACTER	4	corbaserver
(18)	FULLWORD	4	pBeanName
(1C)	FULLWORD	4	lBeanName
(20)	FULLWORD	4	BeanInterfaceType
(24)	FULLWORD	4	pModule
(28)	FULLWORD	4	lModule
(2C)	FULLWORD	4	pInterface
(30)	FULLWORD	4	lInterface
(34)	FULLWORD	4	pOperation
(38)	FULLWORD	4	lOperation
(3C)	CHARACTER	8	userid
(44)	FULLWORD	4	transid
(48)	FULLWORD	4	flag_bytes
(4C)	FULLWORD	4	return_code
(50)	FULLWORD	4	reason_code

standard_header
以下の形式の標準ヘッダーが含まれます。

機能　X'00' に設定される 1 バイト・フィールド

ドメイン

II を含む 2 文字フィールド

*　1 文字の予約フィールド

pIIOPData
変換されていない IIOP バッファーの最初のメガバイトのアドレスが含まれます。

lIIOPData
変換されていない IIOP バッファーの長さが入っています。

pRequestbody
着信 IIOP 要求のアドレスが入っています。

lRequestbody
着信 IIOP 要求の長さが入っています。

corbaserver
この要求に関連した CorbaServer の名前が入っています。

pBeanName
EBCDIC の Bean 名を指すポインターが入っています。

lBeanName
Bean 名の長さが入っています。

BeanInterfaceType
列挙された値が入っています。 X'00' はホーム、X'01' はリモートを示します。

pModule
EBCDIC のモジュール名を指すポインターが入っています。

lModule
モジュール名の長さが入っています。

pInterface
EBCDIC のインターフェース名を指すポインターが入っています。

lInterface
インターフェース名の長さが入っています。

pOperation
EBCDIC のオペレーション名を指すポインターが入っています。

lOperation
オペレーションの長さが入っています。

userid
入出力ユーザー ID が入っています。出力ユーザー ID の長さはちょうど 8 文字である必要があります。それが 8 文字よりも短い場合、ブランクで埋め込む必要があります。

transid
入力 TRANSID が入っています。

Flag_bytes
以下の標識が入っています。

littleEndian
バイト・オーダーを示す 1 バイト・フィールド。**1** は TRUE、**0** は FALSE を表します。

sslClientUserid
TCPIPSERVICE 定義で SSLTYPE CLIENTAUTH が指定されている場合の、USERID の導出を示す 1 バイト・フィールド。ここで、以下のようになります。

0 DFLTUSER からの USERID のセット

1 SSL CERTIFICATE からの USERID

* 2 バイトの予約フィールド

return_code
戻りコードが入っています。

reason_code
理由コードが入っています。

USERID が戻される場合、RETNCODE は RCUSRID (X'01') に設定されます。ユーザー置換可能プログラムは、その他のすべてのフィールドを未変更で戻す必要があります。そうしないと予測不能な結果が生じます。

ユーザー置換可能プログラムのインストールについては、「*CICS Customization Guide*」のユーザー置換可能プログラムによるカスタマイズを参照してください。

IIOP ユーザー置換可能セキュリティー・プログラムの使用:

オプションとして、IIOP セキュリティー・プログラムを提供して、着信 IIOP 要求のエレメントを調べ、USERID を生成することができます。

TCPIPSERVICE リソース定義の URM 属性でセキュリティー・プログラムの名前を指定し、それに PROGRAM リソース定義も提供する必要があります。
TCPIPSERVICE で URM に値を指定しないと、プログラムは呼び出されません。

IIOP セキュリティー・プログラムが呼び出されるのは、CICS が SSL クライアント認証を使用してユーザー ID を取得できない場合のみです。詳しくは、「*CICS RACF Security Guide*」の SSL authentication を参照してください。

サンプル IIOP セキュリティー・プログラム DFHXOPUS が提供されています。

セキュリティー・プログラムは、DB2 にアクセスするためのタスク関連のユーザー出口などの CICS サービス、および要求の本体にエンコードされたアプリケーション・パラメーターを使用することができます。

DFHXOPUS の使用:

CICS 提供のサンプル・ユーザー置換可能プログラム DFHXOPUS は、クライアント証明書 (ある場合) に関連した RACF USERID を受け入れます。

証明書に関連した RACF USERID がない場合は、次のとおりです。

- SSL(CLIENTAUTH) の場合、DFHXOPUS は、クライアント証明書から取り出される COMMONNAME の最初の 8 文字を使用します。

- SSL(YES) または SSL(NO) の場合、DFHXOPUS は、IIOP プリンシパル (ある場合) の最初の 8 文字を使用します。

  注: General Inter-ORB Protocol (GIOP) のバージョン 1.2 以降は、要求ヘッダー内の IIOP プリンシパル・フィールドをサポートしません。したがって、DFHXOPUS は、要求が GIOP 1.1 以前のフォーマットである場合、IIOP プリンシパルから派生されるユーザー ID を戻すだけです。

USERID がこれらの手順を使用して検出されなかった場合、DFHXOPUS は CICS システム初期設定パラメーター DFLTUSERDFLTUSER で指定された USERID を戻します。

セキュリティー出口プログラムは、通信域の `userid` フィールドでユーザー ID を戻します。ユーザー ID が 8 文字未満である場合、出口プログラムはフィールドにブランクを埋め込みます。ユーザー ID が戻されるので、`return_code` フィールドは RCUSRID (X'01') に設定されます。

独自のセキュリティー出口プログラムを作成する場合、`userid` および `return_code` 以外のすべてのフィールドを未変更で戻す必要があります。そうしないと予測不能な結果が生じる可能性があります。

CICS TRANSID の取得

着信 GIOP 要求を CICS トランザクション ID に関連付けるには、REQUESTMODEL リソース定義を提供し、インストールする必要があります。

デフォルト以外のトランザクション ID で実行されるすべての可能な要求に対して、REQUESTMODEL リソースを提供する必要があります。実行時に、CICS が GIOP 要求を受信すると、要求内のフィールドを、REQUESTMODEL で事前定義された値と比較して、最も正確に要求と一致する REQUESTMODEL を検出します。選択された REQUESTMODEL は、要求の処理に使用される TRANSID 名を提供します。一致が見つからない場合、デフォルトの TRANSID (CIRP) が使用されます。REQUESTMODEL は、エンタープライズ Bean またはステートレス CORBA オブジェクト、もしくはその両方で使用できます。次のものを指定します。

- ステートレス CORBA オブジェクトに対する要求と突き合わせる CORBA MODULE および INTERFACE パターン。
- エンタープライズ Bean を突き合わせるための Bean 名。
- 次のものと突き合わせる OPERATION パターン。
  - エンタープライズ Bean メソッド名
  - CORBA ステートレス・オブジェクト・メソッド名
  - IDL オペレーション (CORBA ステートレス・オブジェクトのみ)

  注: OPERATION フィールドは、457 ページの『OPERATION フィールドのネーム・マングリング』で説明されている Java-to-IDL ネーム・マングリング規則に従います。
- 突き合わせ要求が受信されるときに開始される CICS トランザクション。デフォルトは CIRP です。これはデフォルトの DFJIIRP プログラムを指定します。独自のトランザクション定義を使用することを選択する場合、CIRP に基づいてその定義を作成し、TRANSACTION リソース定義を提供します。この定義では、JVMCLASS パラメーターを **com.ibm.cics.iiop.RequestProcessor** に設定して定義

される CICS プログラムの名前に、PROGRAM パラメーターが設定されます。次のデフォルト・リソース定義が DFHIIOP グループで CICS によって提供されます。

```
DEFINE TRANSACTION(CIRP)    GROUP(DFHIIOP)
     PROGRAM(DFJIIRP)       TWASIZE(0)
     PROFILE(DFHCICST)      STATUS(ENABLED)
     TASKDATALOC(ANY)       TASKDATAKEY(USER)
     RUNAWAY(SYSTEM)        SHUTDOWN(ENABLED)
     PRIORITY(1)            TRANCLASS(DFHTCL00)
     DTIMOUT(NO)            TPURGE(NO)
     SPURGE(YES)            ISOLATE(YES)
     RESSEC(YES)            CMDSEC(YES)
     RESTART(NO)
     DESCRIPTION(Default CICS IIOP Request Processor transaction)


DEFINE PROGRAM(DFJIIRP)       GROUP(DFHIIOP)
     DESCRIPTION(CICS IIOP Request Processor)
     JVM(YES)
     JVMCLASS(com.ibm.cics.iiop.RequestProcessor)
     JVMPROFILE(DFHJVMCD)
     LANGUAGE(LE370)        RELOAD(NO)             EXECKEY(USER)
     RESIDENT(NO)           USAGE(NORMAL)          USELPACOPY(NO)
     STATUS(ENABLED)        CEDF(NO)               DATALOCATION(ANY)
     DYNAMIC(NO)
```

要求が AOR に転送される場合は、458 ページの『動的ルーティング』を参照してください。

- 要求を処理する CorbaServer の名前。

REQUESTMODEL リソース定義の詳細については、「*CICS Resource Definition Guide*」を参照してください。

注: エンタープライズ Bean の REQUESTMODEL 定義を作成するプロセスを簡単にするには、CREA CICS 提供トランザクションを使用してください。

パターン・マッチング:

すべての要求は、CORBASERVER および TYPE について、インストールされた REQUESTMODEL 値と比較されます。

TYPE 値 CORBA は、ステートレス CORBA オブジェクトに対する要求を示します。TYPE 値 EJB は、エンタープライズ Bean に対する要求を示します。TYPE 値 GENERIC は、どちらのタイプの要求でも示すことができます。その後、追加のマッチングが TYPE 値に基づいて実行されます。

ステートレス CORBA オブジェクト

ステートレス CORBA オブジェクト (TYPE=CORBA または GENERIC) の場合、マッチング・プロセスは、IIOP メッセージ内に含まれている **MODULE** 名、**INTERFACE** および **OPERATION** フィールドを、インストールされている各 REQUESTMODEL で定義されるパターンと比較します。これは、最も近い一致が見つかるまで行われます。INTERFACE、MODULE、および OPERATION を汎用パターンとして定義できます。パターン・マッチングの規則を以下にまとめています。

- 二重コロンは、コンポーネントの分離文字として使用されます。各コンポーネントの長さは、1 文字から 16 文字の間でなければなりません。
- 汎用パターンは、ゼロまたは 1 個以上の文字と、その後に続く * で構成されます。

複数の異なる汎用パターンが特定のストリングと一致する場合、最長の汎用パターンが最も明確な一致になります。

エンタープライズ Bean
エンタープライズ Bean の場合、マッチング・プロセスは、IIOP メッセージ内の BEANNAME、OPERATION、および INTFACETYPE フィールドを、インストールされた各 REQUESTMODEL で定義されるフィールドと比較します。

OPERATION フィールドのネーム・マングリング:

REQUESTMODEL 定義の OPERATION フィールドは、この要求モデルによってマッチングされるリモート・メソッドの名前を提供するのに使用されます。

実行時に受信される GIOP 要求には、要求モデルの OPERATION フィールドと比較されるオペレーション・フィールドが含まれています。ただし、オペレーション・フィールドの値は、必ずしも、ステートレス CORBA オブジェクトまたはエンタープライズ Bean で使用されるメソッド名と同じであるとは限りません。RMI-IIOP が使用される (エンタープライズ Bean では常に行われ、ステートレス CORBA オブジェクトでは行われる場合があります) 場合、メソッド名は、「マングリング」と呼ばれるプロセスにかけられます。これは、メソッド名を、IIOP を使用する送信に適切な標準のフォームに変更するプロセスです。このマングリングされたメソッド名は、オリジナルのメソッド名と同じではない可能性があります。REQUESTMODEL 内のオペレーション・フィールドは、マングリングされたバージョンのメソッド名 (または、それと一致する、ワイルドカード文字を使用するパターン) を提供する必要があります。

CICS 提供の CREA トランザクションを使用すると、このネーム・マングリングの問題を自動的に処理する、エンタープライズ Bean の REQUESTMODEL 定義を作成できます。

このマングリングとデマングリングの知識は、RMI コンパイラー (RMIC) を使用して生成されるアプリケーションのスタブ・クラスとタイ・クラスにコンパイルされます。

マングリングについて詳しくは、 458 ページの『Java のネーム・マングリング』を参照してください。

REQUESTMODEL の例:

ステートレス CORBA オブジェクトの REQUESTMODEL の例は、次のとおりです。

```
DEFINE REQUESTMODEL(DFJ$IIRH)  GROUP(DFH$IIOP)
       CORBASERVER(IIOP)
       TYPE(Corba)
       MODULE(hello)
       INTERFACE(HelloWorld)
```

```
OPERATION(*)
TRANSID(IIHE)
DESCRIPTION(Hello world java server sample)
```

動的ルーティング:

メソッド起動が別の領域 (AOR) に転送される場合、REQUESTMODEL で指定された TRANSID を、(DYNAMIC パラメーターを使用して) リスナー領域で動的にルーティング可能として定義する必要があります。提供されるデフォルトの TRANSACTION 定義 CIRP を使用する場合、それを変更する必要があります。

Java のネーム・マングリング

ネーム・マングリングは、特定のプログラミング言語で有効な名前を、CORBA インターフェース定義言語 (IDL) で有効な名前にマッピングするプロセスを示す用語です。このセクションでは、Java 名にマングリングが必要な理由、名前がマングリングされる方法、およびマングリングが CICS システムに与える影響について説明します。

Java 名にマングリングが必要な理由:

Java クライアント・プログラムは、Java リモート・メソッド呼び出し (RMI) を使用してサーバー内のメソッドを起動します。

RMI は、クライアントとサーバー間の 2 つの通信プロトコルのどちらかを使用します。

Java Remote Method Protocol (JRMP)
RMI は、クライアント・アプリケーションとサーバー・アプリケーションの両方が Java で作成されるときに JRMP を使用します。CICS は JRMP を使用しません。

Internet Inter-ORB Protocol (IIOP)
RMI は、クライアント・アプリケーションとサーバー・アプリケーションが異なる言語で作成される場合の環境で使用します。IIOP が通信プロトコルとして使用される場合、Java クライアント・アプリケーションは RMI を使用して、別の言語 (例えば、C++) でサーバー・プログラムを起動するとともに、リモート Java プログラムを起動することができます。

IIOP は、インターフェース定義言語 (IDL) を使用して、言語に依存しない方法でオブジェクト間のインターフェースを指定します。Java クライアントがリモート・メソッド呼び出しを行う場合、Java メソッド名およびその引数は、IIOP を使用してサーバーに送信するために等価の IDL に変換されます。マングリングが必要になる可能性があるのは、この時点です。Java 名と IDL 名の規則には数々の相違点があるからです。これらの相違点の一部は次のとおりです。

- Java 名には大/小文字の区別があり、IDL 名にはありません
- Java は多重定義されたメソッドをサポートしますが、IDL はサポートしません
- Java 名には Unicode 文字を含むことができますが、IDL 名はできません
- 一部の有効な Java 名は IDL キーワードと衝突する場合があります
- Java 名は先行する下線で始まることができますが、IDL 名はできません

このような場合には、IDL で許可されない Java 名、または許可されるもののあいまいになる可能性がある Java 名は、受け入れられるフォームにマングルされます。

Java 名のマングリング方法:

Java メソッド呼び出しが IDL 名にマップされる場合の規則は、単純ではなく、環境に依存します。

一例は次のとおりです。

> Java リモート・インターフェースにはメソッド save、Save および SAVE があります。これらの名前は Java では別々ですが、IDL 名には大/小文字の区別がないため、IDL はこれらの名前を区別できません。したがって、これらの名前を別々のものにするために、マングルされます。マングルされた名前は save_、Save_0 および SAVE_0_1_2_3 です。ただし、Java リモート・インターフェースに 1 つのメソッド (save) だけがある場合、あいまいになる可能性がないので、名前はマングルされません。

この例は、次の 2 つの重要な原則を示しています。

- 他のどのメソッドが存在するかを認識することなく、所定メソッドのマングルされた名前を判別することはできません。
- メソッドの追加または除去を行うと、他のメソッドのマングルされた名前に影響を与える可能性があります。

その他にマングリングが必要な事例は、処理が異なります。 Java と IDL 間のマッピングについて詳しくは、オブジェクト管理グループ (OMG) (`http://www.omg.org`) によって公開される「*Java Language to IDL Mapping*」を参照してください。

マングリングが CICS に与える影響:

CICS 内の IIOP に対するサポートには、マングリング規則を実装するコードが含まれていますが、CICS システムを構成し、使用する方法には目に見える影響はほとんどありません。

マングリングが行われることを認識する必要がある状態は、次の 2 つだけです。

REQUESTMODEL を定義する場合
: REQUESTMODEL リソース定義は、インバウンド IIOP 要求を CICS トランザクションにマップします。 Java リモート・メソッド呼び出しによって開始されたインバウンド要求が受信されると、REQUESTMODEL の OPERATION 属性が、そのインバウンド要求内のマングルされた名前と比較されて、REQUESTMODEL が要求と一致するかどうかを判別します。マングリングが行われる可能性がある場合、REQUESTMODEL の OPERATION 属性でメソッド名を指定するのではなく、代わりに汎用オペレーションを指定してください。

Java プログラムのデバッグ・プロファイルを作成する場合
: デバッグ・プロファイルは、デバッガーの制御下で実行されるプログラム・インスタンスを指定します。Java リモート・メソッド呼び出しによって開始されたインバウンド要求が受信されると、デバッグ・プロファイルのメソッド・フィールドが、そのインバウンド要求内のマングルされた名前と比較されて、プロファイルが要求と一致するかどうかを判別します。マングリン

グが行われる可能性がある場合、デバッグ・プロファイルでメソッド名を指定するのではなく、代わりに汎用メソッドを指定してください。

注意: 理論上は、各メソッドに対応するマングルされた名前を推定することは可能ですが、それは簡単な作業ではなく、お勧めしません。これを行うには、マングリング規則について、およびアプリケーションで使用されるすべてのメソッド名について詳細な知識が必要です。また、アプリケーションに小さな変更を加えるだけで、マングルされた名前が変わるリスクもあります。

IIOP 診断の処理

IIOP を介して起動されるリモート・メソッドが失敗すると、クライアント・コードは CORBA 例外を受け取ります。これには、すべてのエンタープライズ Bean 例外が含まれます。

CORBA 例外は、CORBA Web サイト (`http://www.omg.org`) で取得可能な CORBA 資料で定義されます。

多くのインスタンスで、この例外には、問題判別に役立つ CICS 固有のマイナー・コードが含まれています。CICS は現在、次のマイナー・コードを使用します。

表 24. CICS 固有の CORBA マイナー・コード

コード	問題を検出する CICS コンポーネント
1229111296	CICS IIOP 要求受信側
1229111297	CICS II ドメイン内の他のどこか
1229111298	CICS OT ドメインの ORB コンポーネント
1229111299	CICS OT ドメインの JTS コンポーネント
1229111300	CICS OT ドメインの CSI コンポーネント
1229111301	CICS EJ ドメインの CSI コンポーネント

クライアントが、いずれかの CICS マイナー・コードを含む CORBA 例外を受け取る場合は、エラーに関する追加情報がないか、CICS メッセージ・ログを調べる必要があります。

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒242-8502
神奈川県大和市下鶴間1623番14号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。

IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

本書には、技術的に正確でない記述や誤植がある場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。 IBM United Kingdom Laboratories, MP151, Hursley Park, Winchester, Hampshire, England, SO21 2JN 本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

商標

IBM、IBM ロゴおよび ibm.com は、世界の多くの国で登録された International Business Machines Corp. の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、 http://www.ibm.com/legal/copytrade.shtml をご覧ください。

Java およびすべての Java 関連の商標およびロゴは Oracle やその関連会社の米国およびその他の国における商標または登録商標です。

Linux は、Linus Torvalds の米国およびその他の国における登録商標です。

Microsoft、Windows、Windows NT および Windows ロゴは、Microsoft Corporation の米国およびその他の国における商標です。

UNIX は The Open Group の米国およびその他の国における登録商標です。

参考文献

CICS Transaction Server for z/OS の CICS ブック

一般

CICS Transaction Server for z/OS Program Directory、GI13-0565
CICS Transaction Server for z/OS リリース・ガイド、GA88-4308
CICS Transaction Server for z/OS CICS TS V3.1 からのアップグレード、GA88-4310
CICS Transaction Server for z/OS CICS TS V3.2 からのアップグレード、GA88-4311
CICS Transaction Server for z/OS CICS TS V4.1 からのアップグレード、GA88-4312
CICS Transaction Server for z/OS インストール・ガイド、GA88-4309

CICS へのアクセス

CICS インターネット・ガイド、SA88-4317
CICS Web サービス・ガイド、SA88-4315

管理

CICS System Definition Guide、SC34-7185
CICS Customization Guide、SC34-7161
CICS Resource Definition Guide、SC34-7181
CICS Operations and Utilities Guide、SC34-7213
CICS RACF Security Guide、SC34-7179
CICS Supplied Transactions、SC34-7184

プログラミング

CICS アプリケーション・プログラミング・ガイド、SA88-4313
CICS アプリケーション・プログラミング・リファレンス、SA88-4314
CICS System Programming Reference、SC34-7186
CICS Front End Programming Interface User's Guide、SC34-7169
CICS C++ OO Class Libraries、SC34-7162
CICS Distributed Transaction Programming Guide、SC34-7167
CICS Business Transaction Services、SC34-7160
CICS での Java アプリケーション、SA88-4321

診断

CICS Problem Determination Guide、GC34-7178
CICS パフォーマンス・ガイド、SA88-4318
CICS Messages and Codes Vol 1、GC34-7175
CICS Messages and Codes Vol 2、GC34-7176
CICS Diagnosis Reference、GC34-7166
CICS Recovery and Restart Guide、SC34-7180
CICS Data Areas、GC34-7163
CICS Trace Entries、SC34-7187

CICS Debugging Tools Interfaces Reference、GC34-7165

通信

CICS 相互通信ガイド、SA88-4316
CICS External Interfaces Guide、SC34-7168

データベース

CICS DB2 Guide、SC34-7164
CICS IMS Database Control Guide、SC34-7170
CICS Shared Data Tables Guide、SC34-7182

CICS Transaction Server for z/OS の CICSPlex SM ブック

一般

CICSPlex SM 概念および計画、SA88-4319
CICSPlex SM Web User Interface Guide、SC34-7214

管理

CICSPlex SM Administration、SC34-7193
CICSPlex SM Operations Views Reference、SC34-7202
CICSPlex SM Monitor Views Reference、SC34-7200
CICSPlex SM Managing Workloads、SC34-7199
CICSPlex SM Managing Resource Usage、SC34-7198
CICSPlex SM Managing Business Applications、SC34-7197

プログラミング

CICSPlex SM Application Programming Guide、SC34-7194
CICSPlex SM Application Programming Reference、SC34-7195

診断

CICSPlex SM Resource Tables Reference Vol 1、SC34-7204
CICSPlex SM Resource Tables Reference Vol 2、SC34-7205
CICSPlex SM Messages and Codes、GC34-7201
CICSPlex SM Problem Determination、GC34-7203

他の CICS 資料

以下の資料には CICS に関する詳しい情報が含まれますが、これらの資料は CICS Transaction Server for z/OS, バージョン 4 リリース 2 の一部としては提供されません。

Designing and Programming CICS Applications、SR23-9692
CICS Application Migration Aid Guide、SC33-0768
CICS ファミリー: API の構成、SC88-7261
CICS ファミリー クライアント・サーバー プログラミングの手引き、SC88-7429
CICS Family: Interproduct Communication、SC34-6853
CICS Family: Communicating from CICS on System/390、 SC34-6854
CICS Transaction Gateway (OS/390 版) 管理の手引き、SD88-7246

CICS Family: General Information、GC33-0155

CICS 4.1 Sample Applications Guide、SC33-1173

CICS/ESA 3.3 XRF Guide、SC33-0661

他の IBM 資料

以下の資料には、関連する IBM 製品に関する情報が記載されています。

IBM Developer Kit and Runtime Environment, Java 2 Technology Edition Diagnostics Guide、SC34-6358

Persistent Reusable Java Virtual Machine User's Guide、SC34-6201

アクセシビリティー

アクセシビリティー機能は、運動障害または視覚障害など身体に障害を持つユーザーがソフトウェア・プロダクトを快適に使用できるようにサポートします。

CICS システムのセットアップ、実行、および保守に必要なほとんどの作業は、以下のいずれかの方法で行うことができます。

- CICS にログオンした 3270 エミュレーターを使用する
- TSO にログオンした 3270 エミュレーターを使用する
- 3270 エミュレーターを MVS システム・コンソールとして使用する

IBM パーソナル・コミュニケーションズは、身体障害のある方々のためのアクセシビリティー機能を持つ 3270 エミュレーションを提供します。 CICS システムで必要なアクセシビリティー機能を提供するためにこの製品を使用することができます。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

[サ行]

[タ行]

[ハ行]

[マ行]

[ヤ行]

[ラ行]

[ワ行]

[数字]

A

B

C

D

E

G

I

J

L

M

O

P

R

S

T

U

W

Z

[特殊文字]

SA88-4321-01

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21