

Oracle9iAS Containers for J2EE における クラス・ロード

オラクル・ホワイト・ペーパー
2003 年 4 月

Oracle9iAS Containers for J2EE におけるクラス・ロード

よくある問題	3
クラス・ロードの基本	3
クラス・ローダーの名前空間	4
クラス・ローダーのリンク	5
クラスの検索	6
J2SE 環境におけるクラス・ロード	9
クラス・ローダーに関するその他の考慮点	10
依存関係の宣言	10
依存関係の解決に関する問題	10
セキュリティ	11
J2EE におけるクラス・ロード	11
モジュール間の依存関係	12
アプリケーション間の依存関係	13
Web モジュール内の検索順序	14
ORACLE9IAS CONTAINERS FOR J2EE (OC4J) における	
クラス・ロード	14
アプリケーション間の共有	17
構成オプションのまとめ	18
J2EE でよくあるクラス・ロード問題	19
不十分な参照可能性	20
過剰な参照可能性	20
参照可能性の混乱	21
J2EE のクラス・ロードのベスト・プラクティス	22
結論	23

Oracle9iAS Containers for J2EE におけるクラス・ロード

よくある問題

ClassNotFoundException

NoClassDefFoundError

ClassCastException

見覚えがありますか。このような例外のデバッグに、何時間も費やすケースは少なくないはずです。Java 2 および J2EE 仕様で追加された変更により、クラスのロードは、環境変数 CLASSPATH の定義ほどシンプルではなくなりました。Oracle9iAS Containers for J2EE (Oracle9iAS に付属の OC4J) などの J2EE アプリケーション・サーバーは、クラスの場所を指定する様々なメカニズムで CLASSPATH を補足します。このように複雑さが増したことにより、診断が難しく複雑なクラス・ロード・エラーが発生します。

本書では、標準 J2SE 環境とより複雑な J2EE 環境 (Oracle9iAS Containers for J2EE など) の両方でクラス・ロードがどのように行われるかを詳しく説明します。核となるクラス・ロード概念のいくつかを理解することにより、問題を即座に診断できるようになり、複雑なクラス・ロード・エラーを解決できるようになります。

クラス・ロードの基本

基本から始めましょう。「クラス・ロード」という用語は、あるクラス名のバイトを検索してそれを Java の *Class* インスタンスに変換するプロセスを指します。Java Virtual Machine (JVM) 内のすべての *java.lang.Class* インスタンスは、JVM 仕様で定義されているクラス・ファイル形式で構成されるバイト配列として発生します。

クラス・ロードは、起動プロセス中は JVM によって実行され、その後は *java.lang.ClassLoader* クラスのサブクラスによって実行されます。これらのクラス・ローダーが行う抽象化により、Java の最も強力な機能の 1 つであるコンパイル時での不明なクラスからのコードの選択、ロードおよび実行機能が提供されます。この動的ロード機能は、Java でよく宣伝される「モバイル・コード」機能の基礎であり、Java 言語のスケラビリティを大幅に高めます。

たとえば *Class.forName()* メソッドを使用すると、開発者は完全修飾クラス名文字列を単に指定するだけで *java.lang.Class* インスタンスを作成できます。

すなわち、クラス・ローダーはクラスをロードする *java.lang.ClassLoader* クラスのサブクラスになります。そして各クラス・ローダーが 1 つ以上のコードソースを処理します。コードソースとは、JVM がクラスを検索するルートとなる場所です。コードソースは、バイナリ・クラス・ファイルや最初にコンパイルさ

れる必要がある Java ソース、さらには動的に生成されたクラスの物理ストレージを表すよう定義することができます。

例としては、ほとんどの開発者が精通している CLASSPATH 環境変数があります。この CLASSPATH 内の各要素がディレクトリ、zip ファイルまたは jar ファイルといったコードソースになります。クラス・ローダーはそれぞれのコードソースをクラス・パッケージ名とともに使用して、クラスを検索する場所を定義します。たとえば、`c:\classes` などのディレクトリ・コードソースとクラス名 `com.acme.Dynamite` を与えられると、クラス・ローダーは `c:\classes\com\acme\Dynamite.class` への単純な変換を通じてフル・パスを作成できます。

Java アプリケーションには、数多くの異なるメカニズムを使用してクラスをロードする様々なクラス・ローダーが多数含まれています。前述の CLASSPATH の例に加えて、クラス・ローダーはクラスを次の場所から取得するよう設計することができます。

- データベースから。この場合、構成には特定のデータベース内の正しい表を指すために必要な全データなどが含まれます。
- 独自の通信プロトコルを実行しているリモート・サーバーから。この場合、構成には DNS 名、ポート、その他のネットワーク情報などが含まれます。
- ファイル・システムから（ただし、プロパティ・ファイルに指定されている特別な検索順序に従う）。
- XML ファイル内に定義されているソースから。
- コンパイルされる必要があるソース・コード（*.java）ファイルから。

クラス・ローダーの名前空間

各クラス・ローダー（または *ClassLoader* サブクラスのインスタンス）は、個々の一意の名前空間を定義することに注意してください。たとえば *N* という名前のクラスに対し、あるクラス・ローダー内に存在できるクラス *N* のインスタンスは 1 つのみです。ただし、同じクラスを 2 つの異なるクラス・ローダーでロードすることができます。この場合、JVM は、各クラスがそれをロードした *ClassLoader* インスタンスによりさらに修飾されることを前提とします。実際に、クラスのフル・ネームはその完全修飾クラス名にそのクラス・ローダーを足したものになります。

重複クラスのロードは、些細でも面倒なバグの原因となることがあります。たとえば、2 つの異なるクラス・ローダーが `com.acme.Dynamite` をロードすると、それぞれが独自の静的データを個々に持つ 2 つの *Class* インスタンスが発生します。オブジェクトを一方のインスタンスから他方にキャストしようとすると、*ClassCastException* が発生します¹。例として、EJB と Web の両方のモジュールを含むアプリケーションがあり、EJB モジュールの *EJBObject* インタフェース (*Cart*) が Web モジュールの *war* ファイルで複製されると仮定します。Web モジュールの

¹ 型が等しいかどうかをテストする場合、JVM はソースの継承階層 `==` (識別) 比較を使用して各 *Class* インスタンスを必要な型の *Class* インスタンスと比較します。

サーブレットが JNDI を使用して EJB を検索すると、Cart インタフェースへのキャスト時に `ClassCastException` が発生します。

この例外が発生するのは、他の全ローダーとは異なり、Web モジュール・ローダーはまずローカルに検索を行う必要があるためです。この場合、ローダーはローカルにパッケージングされた Cart インタフェースを取得し、JNDI により返される EJB は EJB ローダーの Cart インタフェース取得します。(この動作の理由は「Web モジュール内の検索順序」で説明します。)

クラス・ローダーのリンク

クラスを検索が行われると、その検索の実行に特定の順序でクラス・ローダーが使用されるよう、各クラス・ローダーがリンクされています。それぞれのローダーには親クラス・ローダーが関連付けられます。この関係を使用して、クラス・ローダー階層は単純な連鎖から複雑なマルチブランチ・ツリーまで、幅広いツリー構造を表すことができます。このようなツリーの重要な特性として、1 方向、すなわち親を介して上方向にのみ検索が可能ながあげられます。

この階層の結果、名前空間がネストされ、ツリー内のクラス・ローダーの位置によりクラスの参照可能性が決定されます。すなわち、ツリーの一番下のノードはそれぞれの親のコンテンツを参照できますが、親はそれぞれの子ノードのコンテンツを参照できません。

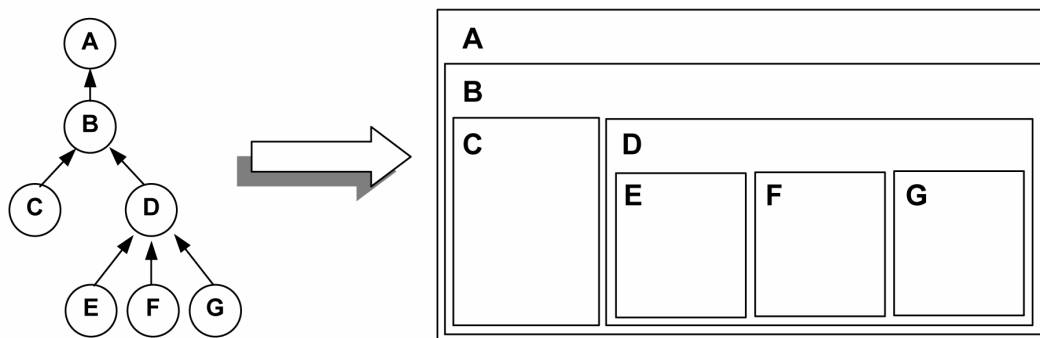


図 1: クラス・ローダー・ツリー内のクラスの参照可能性

この関係を図 1 に示します。右側の各名前空間 (ボックス) は、それ自体と、それを含む全名前空間のコンテンツを参照できます。ツリーの最上位近くにあるクラスは、下のクラスより参照可能性が高くなります。より多くのクラスがそのクラスを参照できるためです。

このパーティション化スキームの制限的性質により、問題が発生する場合があります。例として、実装コードを (`Class.forName()`メソッドを使用して) 動的にロードし、実装がサードパーティ・コードにより追加されるフレームワークを考えてみます。ツリー内では、サードパーティ・コードがフレームワーク自体よりも下に存在することがあります。この場合、フレームワークは、どのようにその範囲を広げてこのコードへの参照可能性を得るのでしょうか。

その答は次のとおりです。フレームワークは正しい参照可能性を持つクラス・ローダーを明示的に検索し、それを使用して実装をロードする必要があります。Java 2はこの状況を処理する単純で重要なメカニズムを提供します。Javaの各 Threadは、コンテキスト ClassLoader として知られる単一 ClassLoader インスタンス（メソッド `Thread.currentThread().getContextClassLoader()` をコールすることにより取得可能）を格納できます。このインスタンスは、任意の参照可能性を提供するよう環境で設定できます。（デフォルトでは、コンテキスト・ローダーは `ClassLoader.getSystemClassLoader()` をコールした結果に設定されず。）

クラスの検索

クラス・ローダー間の関係について説明しました。次は JVM による実際の検索方法を説明します。クラス・ローダーは、単純な委譲モデルを使用してクラスを検索します。各 ClassLoacer インスタンスには、関連付けられている親クラスと、前にロードされたクラスに対するキャッシュが存在します。クラスをロードするようコールされると、クラス・ローダーはまずそのキャッシュ内を検索し、クラスが検索されない場合、その親に委譲します。親がクラスを返すことができない場合、ローダーはクラスを自身で検索します。このプロセスがクラス・ローダー・ツリーの最上位レベルまで繰り返されます。

検索プロセスの最初の手順は、*初期クラス・ローダー*を検索することです。初期クラス・ローダーとは、前述の検索実行時に使用された最初のローダーを指します。初期ローダーは、実際のローダーである必要はないことに注意してください（たとえば、親ローダーがクラスを最初に返す場合もあります）。初期ローダーの選択は、重要な第1手順であり、次に説明するとおり、*暗黙的*または*明示的*に行われます。

*暗黙的*の選択が圧倒的に多く実行され、次の状況で発生します。

- *オブジェクトのインスタンス化*。クラス A が `new` 演算子を起動してクラス B のインスタンスを作成する際、JVM は A のクラス・ローダーを初期クラス・ローダーとして選択します。
- *依存関係の解決*。クラス A が B に依存している場合、JVM は B のロードが必要な場合に A のクラス・ローダーを選択します。このプロセスは再帰的なため、B が C に依存している場合、JVM は C をロードする際に B のクラス・ローダーを選択します。各クラスには異なるクラス・ローダーが含まれている場合があることに注意してください。

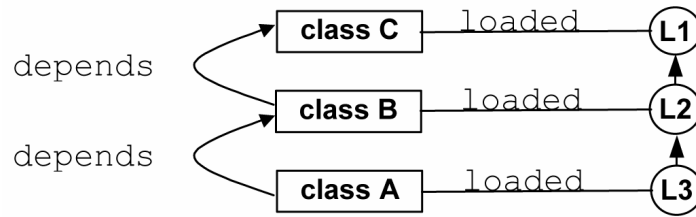


図 2: 初期クラス・ローダーの選択

- **動的ロード。** `Class.forName()` メソッドはオーバーロードされ、`ClassLoader` パラメータを取るバージョンと取らないバージョンがあります。ClassLoader が渡されない場合、JVM は非 NULL の `ClassLoader` (ブートストラップ・ローダー以外、詳細は次項を参照) を使用してコール・スタックで最初のオブジェクトを検索します。オブジェクトが検索されない場合は、`ClassLoader.getSystemClassLoader()` が使用されます。
- **オブジェクトのデシリアライズ。** `ObjectInputStream.resolveClass()` メソッドは `Class.forName()` と同様にコール・スタックを検索します。

明示的選択は暗黙的選択ほど一般的ではありませんが、アプリケーション・コードが次のメソッドへのコールからの結果を使用する場合に実行されます。

- `Class` インスタンス上の `getClassLoader()`
- `ClassLoader.getSystemClassLoader()`
- `Thread.currentThread().getContextClassLoader()`
- クラス・ローダーを取得するアプリケーション固有のメソッド

これによって参照可能性が決定されるため、いずれの場合にも、初期ローダーの選択は非常に重要です。たとえば、ターゲット・クラスまたはその依存関係の 1 つが、連鎖の下位または他のブランチ内のローダーに対してのみ参照可能な場合、必然的に例外が発生します。

`loadClass()` のデフォルト実装は、次の標準検索アルゴリズムをエンコードします (図 3 を参照)²。

² このメソッドを上書きしてアルゴリズムを変更することは可能ですが、そのような操作は Java 2 の世界では敬遠されます。動作が予測不可能になり、診断が難しくなるためです。

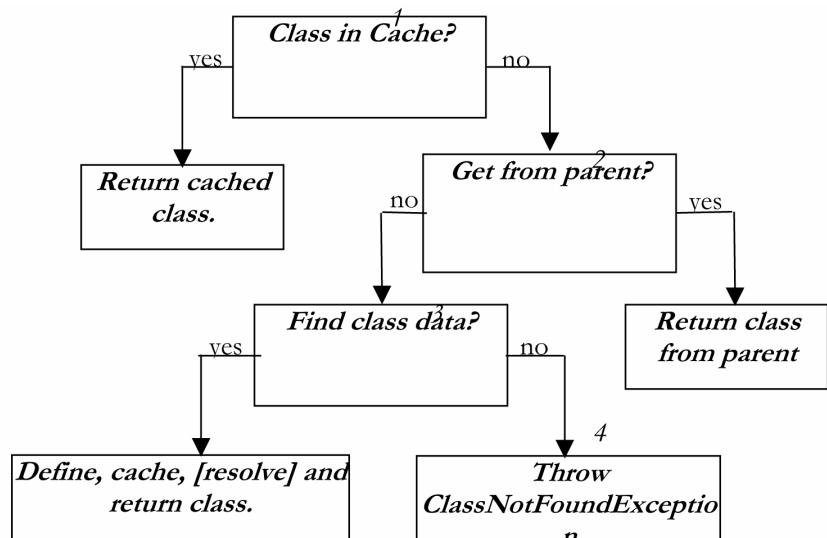


図 3: クラス・ロードのデフォルト・アルゴリズム

1. findLoadedClass(name)をコールしてクラスが以前にロードおよびキャッシュされたかを調べます。そのような形跡がない場合は、手順 2 に進みます。
2. 親が非 NULL の場合は、parent.loadClass(name)をコールします。(親クラスがこれと同様のアルゴリズムを実行します。) 親が NULL の場合は、findBootstrapClass(name)をコールします。いずれの場合でも、クラスが存在しない場合は手順 3 に進みます。
3. findClass(name)をコールします。このメソッドは各 ClassLoader サブクラスに実装する必要があり、クラスのバイトを検索して defineClass()をコールする必要があります。次に、defineClass()がバイトを Class インスタンスに変換してキャッシュを更新します。クラスがこの手順でも検索されない場合は手順 4 に進みます。
4. ClassNotFoundException が発生します。

クラスが手順 2 または 3 でロードされた場合、リンク・プロセスを再帰的な方法で完了する resolveClass()への最終コール(任意)が行われます。これは、loadClass()への‘resolveClass’パラメータの設定により制御されます。デフォルトではこのパラメータは False のため、レイジー・ロード(この後の「依存関係の解決に関する問題」を参照)と呼ばれる機能が使用可能になります。

この検索順序では、クラスが同一連鎖内の複数のクラス・ローダー間で複製される場合、参照可能性が高いほうのクラス・ローダー(ツリー内で上位のクラス・ローダー)が常に選択されます。このため通常は、連鎖内で上位のクラスを置換または上書きすることは不可能です。通常、これは利点となります。一部のオブジェクトのみのスーパークラスとして使用される、複製の異なる java.lang.Object クラスを追加することにより引き起こされる混乱を想像してみてください。

別途指定のないかぎり、ClassLoader 内の検索順序は、コードソースの指定順序に基づき決定されます。

J2SE 環境におけるクラス・ロード

JVM (JDK 1.2 以上の全バージョン) は起動時に、次の 3 種類のローダーで構成される初期クラス・ローダー階層を形成します。

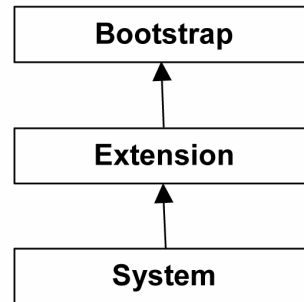


図 4: 標準 J2SE クラス・ローダー連鎖

ブートストラップ (Bootstrap) ローダー (起動ローダーとも呼ばれる) は、コア Java クラス (`rt.jar`、`l18n.jar` など) をロードします。Sun の JVM では、`-Xbootclasspath` オプションまたは `sun.boot.class.path` システム・プロパティを使用して追加クラスを指定できます。このローダーの特長は、実際には `java.lang.ClassLoader` のサブクラスではなく、JVM 自体によって実装されるという点です。

拡張 (Extension) ローダーは、JRE の拡張ディレクトリ (`jre/lib/ext` または `java.ext.dirs` システム・プロパティで指定されるディレクトリ) 内の `jar` からクラスをロードします。このローダーは、コア Java クラスの範囲外の新機能を導入する標準メカニズムを提供します。このインスタンス上の `getParent()` をコールすると、ブートストラップ・ローダーは実際の `ClassLoader` インスタンスではないため、常に `NULL` が返されます。デフォルトの拡張ディレクトリは、同一 JRE から起動された全 JVM において共通で、このディレクトリに置かれた `jar` は、プロセス間で参照できます。

システム (System) ローダー (アプリケーション・ローダーとも呼ばれる) は、JVM の起動時にコマンドラインや `java.class.path` システム・プロパティに記載のディレクトリおよび `jar` からクラスをロードします。このローダーは、スタティック・メソッド `ClassLoader.getSystemClassLoader()` で必ず検索できます。別途指定のないかぎり、ユーザーがインスタンス化するすべての `ClassLoader` は、このローダーを親として持ちます。

JDK 1.4 は、承認済標準 (CORBA など、Java Community Process では定義されていないが JRE では提供される標準) をオーバーライドするメカニズムを追加しています。³

この機能により、JRE で提供される標準にかわり、これらの標準の新バージョンを使用できます。この機能をサポートするメカニズムは、JVM/ブートストラップ・ローダー内で実装されます。

³ <http://java.sun.com/j2se/1.4/docs/guide/standards/index.html> を参照。

クラス・ローダーに関するその他の考慮点

J2EE および OC4J 環境へと進む前に、標準 J2SE 環境に関連するその他のクラス・ロード問題として考慮の必要な点がいくつかあります。

依存関係の宣言

JDK 1.2 より前は、1 つの jar ファイル内のクラスと、その他のファイル内のクラス間の依存関係を表現するメカニズムは存在しませんでした。ユーザーは、どの jar を含めるかを把握し、それをクラス・パスに記述することを求められていましたが、これはエラーの原因となりました。

例として、グラフを作成するサードパーティ・ライブラリ (graphs.jar) のクラスを使用する必要があるアプリケーションを考えてみます。さらに、このライブラリは他のライブラリ (compuserve.gif.jar) 内の下位レベルのイメージ・フォーマット・コードに対する依存関係があると仮定します。この例では、コードはクラス・パス内の graphs.jar のみで正しくコンパイルしますが、実行時には失敗します。compuserve-gif.jar がクラス・パス内に存在しないためです。この依存関係がサードパーティ・ライブラリにより明示的に記述されない場合、開発者は試行錯誤を繰り返して他方の jar をクラス・パスに追加することになります。

Java 2 は、jar ファイルが他の jar に対する自身の依存関係を宣言できるようにする重要な新しい汎用構成メカニズムを追加しました。このメカニズムは、バンドル済オプション・パッケージ⁴と呼ばれます。また、新しいマニフェスト属性である Class-Path も追加されました。この値はディレクトリや jar ファイルのパス (囲み jar の場所に相対的である必要があるパス) をスペースで区切ったリストになります。jar ファイルを受け入れる ClassLoader は、この機能をサポートすることを求められます。

この機能の一般的かつ重要な使用方法の 1 つは、`-jar <jarfile>` JVM オプションによるアプリケーションの実行を可能にすることです。このマニフェスト属性は、jar が独自のクラス・パスを定義できるようにするため、ユーザーがそれを `-cp` または `-classpath` オプションで宣言する必要がなくなります。

JDK 1.3 は、jar が特定バージョンの jar に対する依存関係を宣言できるようにする、異なるメカニズムを導入しました (主に Applet 用)。このメカニズムは、インストール済オプション・パッケージと呼ばれます。これらのメカニズムは基本 JDK 機能に対する拡張機能であり、通常は `jre/lib/ext` ディレクトリにインストールされます。Java Plugin 環境で実行される Applet に対して、このメカニズムには、jar がまだ存在していない場合にそれをフェッチおよびインストールする元となる URL を提供する機能が含まれます。

依存関係の解決に関する問題

クラス・ロードは JVM 自体による方法 (new 演算子を使用したオブジェクトのインスタンス化など) とアプリケーション・コードによる明示的な方法

(`Class.forName()` のコールなど) の両方で起動することができます。

ただし、ある特定の状況に対しては特別な注意が必要です。それは、依存関係の

⁴ この名前は、これより後のバージョンのメカニズムに由来します。
<http://java.sun.com/products/jdk/1.2/docs/guide/extensions/index.html> を参照してください。
1.2 以降のバージョンに関しては、1.2 を 1.3 または 1.4 に変更してください。

解決です。

クラスは、それが依存する他のクラスへの参照を含んでいて、そのクラスの一部はまだロードされていない場合もあります。このような参照を解決するプロセス（「リンク」と呼ばれる）は、クラスのロード中に実行されますが、一部の参照はメソッド実行中に最初に使用されるまで遅延されます。このレイジー・ロードにより起動時間が短縮されますが、予期せぬエラーにつながることもあります。

セキュリティ

Java 2 のセキュリティは、特定の権限を特定のコードに付与または削除できるという概念に基づいていて、細分性の単位は URL 形式のコードソースになります。ClassLoader は、class インスタンスとそのコードソース間を関連付ける上で重要な役割を果たし、このため、セキュリティにおいても重大な役割を担います。

この問題に関する詳しい説明は本書の範囲外です。ただし、次のメソッドをコールすることにより、あらゆるオブジェクトに対してコードソース URL を取得できます。

```
obj.getClass().getProtectionDomain().getCodeSource().getLocation();
```

J2EE におけるクラス・ロード

J2EE 開発環境は複雑であり、そのためにクラス・ロード動作が大きな影響を受けます。特に J2EE 仕様は、アプリケーションを柔軟に定義するため、クラス・ロードの概念に重要な要素が追加されます。

アプリケーションの種類は、コンポーネント・ベース、個別共存および再起動可能なものがあります。次に、それぞれの特長について説明します。

コンポーネント・ベース。アプリケーションは 1 つの巨大な塊ではなく、事前定義済みのパッケージング (JAR、WAR および RAR ファイルを使用) / デプロイ・ディレクトリ構造およびアンブレラ・パッケージング構造 (EAR ファイル) を持つコンポーネント (EJB、サーブレット、JSP、リソース・アダプタなど) の集合です。このような構造はそれぞれ ClassLoader を持っています。

EAR ファイル内では、META-INF/application.xml ファイルにそれぞれの内部モジュールへの相対パスが含まれています。各モジュールは、パッケージングにより、自体が事前定義済コードソースであるか、または事前定義済コードソースを含んでいます。

- JAR ファイル (EJB およびアプリケーション・クライアント・モジュール)。コードソースには、次のものが含まれます。
 1. jar ファイル自体。
 2. META-INF/manifest.mf ファイルが存在する場合は、その内部の全 ClassPath エントリ。
 3. 2 の全 jar 内の全マニフェスト ClassPath エントリ (再帰的)。
- WAR ファイル (Web モジュール)。コードソースには、次のものが含まれます。

4. WEB-INF/classes ディレクトリ。
 5. WEB-INF/lib ディレクトリが存在する場合は、その内部の全 jar ファイル。
 6. META-INF/manifest.mf ファイルが存在する場合は、その内部の全 ClassPath エントリ。
 7. 2 および 3 の全 jar 内の全マニフェスト ClassPath エントリ(再帰的)。
- RAR (リソース・アダプタ・モジュール)。コードソースには次のものが含まれます。
 8. 全 jar ファイル (全レベル)。
 9. 1 の全 jar 内の全マニフェスト ClassPath エントリ (再帰的)。

EAR ファイル自体のルートにあるマニフェスト・ファイル(および ClassPath エントリ)は、すべて無視されることに注意してください。

アプリケーション・サーバー・ベンダーは、追加の(ただし非標準の)コードソースを取り入れる拡張機能を自由に追加できます。(Oracle OC4J の拡張機能は次項で説明します。)

個別共存。 多数のアプリケーションが 1 つのアプリケーション・サーバー内に隣り合わせで存在することは可能ですが、それぞれのアプリケーションは独自の名前空間内に個々に存在する必要があります。これは実際には、各アプリケーションが(1 つ以上の)個々の ClassLoader インスタンスでロードされる必要があることを意味します。

再起動可能。 アプリケーションは、サーバーを停止せずに再起動できます。この機能をサポートするサーバー(OC4J など)では、あるアプリケーションに対して使用された ClassLoader は破棄され、再インスタンス化される場合があります⁵。この動作により、同一クラス/異なるローダーという固有のシナリオが生まれ、些細なクラス・ロード問題につながる可能性があります。

モジュール間の依存関係

J2EE には、サーブレットおよび JSP が EJB をコールでき、これら 3 つすべてがリソース・アダプタを使用できるという固有の機能があります。アプリケーション内の各 Web モジュールは相互に分離される必要があります。EJB 1.x では、クライアント要素(ホーム/リモート・インタフェースおよびその依存関係)のみ参照できる必要がありました。ただし、現在の EJB 2.x では、実装クラスも参照できる必要があるという要件がローカル・インタフェースにより追加されました。

J2EE サーバーは、各モジュールがこの要件を満たす正しい参照可能性を持つようにアプリケーションのクラス・ローダーを配置する必要があります。複数の Web モジュール、EJB およびリソース・アダプタを含む EAR ファイルの典型的なクラス・ローダー構造は次のとおりです。

⁵ JDK1.2 以降では、クラスおよびクラス・ローダーに対するガベージ・コレクションが可能なため、元のアプリケーション・クラスへの参照がなくなると、ガベージ・コレクションが行われます。

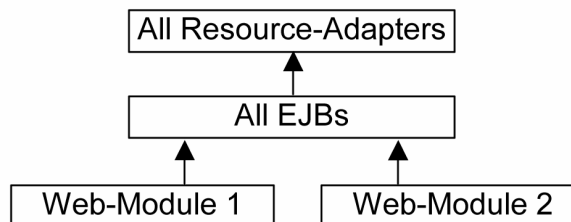


図 5: J2EE アプリケーション内の複数モジュール間におけるクラス・ローダー構造

前述のとおり、各ローダーはそれぞれのパッケージに定義されている全コードソースを含みます⁶。この構造により、各 Web モジュールにロードされるクラスが確実に分離される一方で、EJB およびリソース・アダプタ・クラスが各モジュールを参照できるようになります。

アプリケーション間の依存関係

一般に、アプリケーション間の依存関係は最小化する必要があり、J2EE 1.3 では依存関係を満たす明示的なメカニズムは提供していません。このメカニズムが必要な場合は、共有コードを参照可能にする対象に応じて、2 つの範囲を考慮に入れます。

- **全アプリケーション。** このためには、ライブラリを JRE 拡張ディレクトリに入れてサーバーの起動時に `-classpath` 引数でこれらのライブラリを含めるか、ベンダー固有のメカニズムを使用します。このレベルでコードを変更するには、サーバーの再起動が必要な場合があります。
- **一部のアプリケーション。** このためには、各アプリケーション内のライブラリを複製するか、ベンダー固有の構成オプションを使用します。ここで重要なことは、共有クラスのオブジェクトをアプリケーション全体で参照可能にする必要があるかどうかです。必要がある場合には、ライブラリを複製しても効果がありません。クラスが複製され、前述のとおり、JVM はこれらのクラスを異なるクラスと見なすためです。

一部のアプリケーションに参照可能性を持たせることは理想的ではありません。複製によりディスクとメモリーの両方のフットプリントが増え、バージョン問題にもつながる可能性があるためです。ベンダー固有のメカニズムを利用すると移植性が損なわれます。コンテナ・レベルでのデプロイはいつでも実行できますが、ライブラリに対する参照可能性が過剰に高くなり、アプリケーションの再起動が制限される場合もあります。

⁶ 一部のベンダーは、J2EE 1.3 仕様のあいまいさを理由に、WAR マニフェスト `Class-Path` に定義されているコードソースをアプリケーションの構造内の最上位ローダーにエクスポートするという選択をしています。OC4J では、それは行っていません。この問題は J2EE 1.4 仕様で解決される予定です。OC4J R9.0.3 では、`orion-web.xml` の次のタグを使用して WAR ファイルでマニフェスト `Class-Path` を使用できます。

```
<web-app-class-loader include-war-manifestclass-path="true" />
```

これ以上に優れた解決策は、アプリケーション間で次のコードの共有を可能にすることです。この方法は J2EE 1.4 で提供される予定です。

1. 標準化されているコード
2. 複製を不要にするコード
3. 任意のアプリケーション・サブセットのみに参照可能なコード
4. アプリケーションの再起動を適切にサポートするコード

Web モジュール内の検索順序

Servlet 2.3 仕様では、Web モジュール内の検索順序を WEB-INF/classes/が最初で、次に WEB-INF/lib/と規定しています。さらに、標準のクラス・ローダー検索順序に特異性を加えています。

「また、WAR 内にパッケージングされているクラスおよびリソースが、コンテナ全体のライブラリ JAR にあるクラスおよびリソースより優先してロードされるようにアプリケーション・クラス・ローダーを実装することが推奨される」

すなわち、親連鎖を検索するよりも、Web アプリケーション・ローダーは最初にローカルに検索を行う必要があるという意味です。これにより、WAR 内にパッケージングされているクラスがローダー連鎖の上位にインストールされているコードを上書きできるようになります。

ただし、これは推奨であるため、全ベンダーがこれを実装するとはかぎらないことに注意してください。

また、これは Web アプリケーションのみに適用されるものであり、たとえこの機能が有効であっても、他の J2EE コンポーネントには適用されないことにも注意してください。この問題も、J2EE 1.4 ではより総合的に対処される予定です。

OC4J R9.0.3 からは、orion-web.xml の次のタグを使用して、この機能を Web アプリケーションごとに実行できます。

```
<web-app-class-loader search-local-classesfirst="true"/>
```

ORACLE9iAS CONTAINERS FOR J2EE (OC4J) における クラス・ロード

Oracle9iAS Containers for J2EE (OC4J) は、すべてのアプリケーション・サーバーと同様に、J2EE の要件をサポートするよう J2SE のクラス・ロード機能を拡張する必要があります。ここでは、OC4J におけるクラス・ロードの詳細を検討します。

すべてのアプリケーション・サーバーと同様に、OC4J は J2EE 仕様に含まれない様々な構成ファイルを使用します。これらのファイルの一部では、OC4J の他の機能の制御に加えて、クラス・ロードに関連する 2 つのオプションが使用可能であり、この両方のオプションにより追加コードソースの指定が可能になります。ファイルが XML 形式のため、クラス・ロード・オプションは XML タグの形式を取ります。

- `<classpath path="...">`: ディレクトリ、jar または zip ファイルをセミコロンの区切ったリスト。相対と絶対の両方のパスを使用できます。

- `<library path="...">`: `<classpath>`と同様ですが、ディレクトリには特別な処理が行われ、パス属性に記載されるディレクトリに含まれているあらゆる jar または zip ファイルも含まれます。ディレクトリ検索は再帰的ではありません。

これらのタグをサポートする構成ファイルには次の 2 種類があります。

- **サーバー全体。**この種類のファイルは、通常 config ディレクトリに存在し、全アプリケーションに影響を与えます。
 1. *application.xml*⁷。 `<library>`タグをサポートします。ここに記載のコードソースは、全アプリケーションが参照できます。
 2. *global-web-application.xml*。 `<classpath>`タグをサポートします。ここに記載のコードソースは、全 Web アプリケーションに追加され、その層より上では参照できません。各 Web アプリケーションは、同じパスを使用してこれらのコードソースを参照しますが、それぞれの Web アプリケーションは、独自のクラス・ローダーを持っているため、このタグを使用するとクラスの複製が発生します。
- **アプリケーション固有。**この種類のファイルは、EAR および WAR ファイル内の標準デプロイメント・ディスクリプタ・ファイルとともに存在し、その機能を拡張します。
 3. *orion-application.xml*。 *application.xml* を拡張し、 `<library>`タグをサポートします。ここに記載のコードソースは、アプリケーション内の全モジュールが参照できます。
 4. *orion-web.xml*。 *web.xml* を拡張し、 `<classpath>`タグをサポートします。ここに記載のコードソースは、Web アプリケーションのみが参照できます。

⁷ このファイルは、同名の標準 EAR デプロイメント・ディスクリプタとの混同を避けるため、*global-application.xml* という名前がより適切でした。

OC4J インスタンスの典型的なクラス・ローダー・ツリーは次のとおりです。

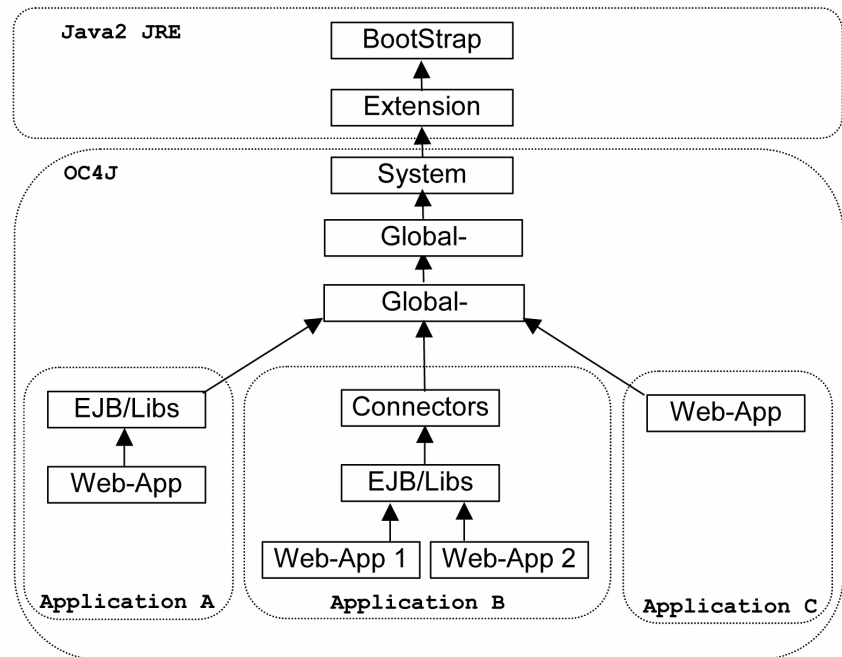


図 6: 典型的な OC4J クラス・ローダー・ツリー

次に、それぞれの OC4J ローダーについて簡単に説明します。

システム (System) ローダーとは、標準 J2SE システム・ローダーのことで、J2EE API および OC4J の全クラスを含むよう構成されています。

グローバル・コネクタ (Global-connectors) ローダーは、OC4J の oc4j-connectors.xml ファイルで参照される RAR ファイルの全コードソースを含んでいます。

グローバル・アプリケーション (Global-application) ローダーは、OC4J の application.xml ファイル内のあらゆる<library>タグの全コードソースを含んでいます。デフォルトでは、OC4J インストール・ディレクトリ内の home/lib ディレクトリがここに記載されるため、このディレクトリ内にあるすべての jar が追加されることに注意してください。

コネクタ (Connectors) ローダーは、あらゆるアプリケーション RAR ファイルの全コードソースを含んでいます。

EJB/ライブラリ (EJB/Libraries) ローダーは、application.xml 内のあらゆる<ejb>タグ、および orion-application.xml ファイル内のあらゆる<library>タグの全コードソースを含んでいます。

Web アプリケーション (Web-application) ローダーは、あらゆる WAR ファイル、orion-web.xml のあらゆる<classpath>タグ、および global-web-application.xml ファイルのあらゆる<library>タグの全コードソースを含んでいます。

ツリーは、構成ファイルのコンテンツと各アプリケーションのコンテンツに基づいて構成されます。たとえば、図のアプリケーション C は、単一の Web モジュールのみを含んでおり、orion-application.xml ファイル内の<library>タグを指定していないため、Web アプリケーション・ローダーが 1 つのみ存在します。同様に、グローバル application.xml ファイルに<library>タグが存在しない場合、グローバル・アプリケーション・ローダーは作成されず、システム・ローダーが共通の親アプリケーションになります。

アプリケーション間の共有

前述したとおり、OC4J は一部のアプリケーションにおける参照可能性をサポートするメカニズムを提供します。このメカニズムでは、2 つ以上のアプリケーション間で複製なしにクラスを共有する必要があります。このメカニズムは、シンプルかつ柔軟で、あらゆるアプリケーションが他のアプリケーションをその親であると宣言できます。次に OC4J は、親アプリケーションの ClassLoader の 1 つを、子アプリケーション内の最上位ローダーの親として配置します。

たとえば、モジュールを含まずに共有 jar のみを含むアプリケーションを作成して、その jar を orion-application.xml ファイルの<library>タグに記載すると、次の配置を作成できます。

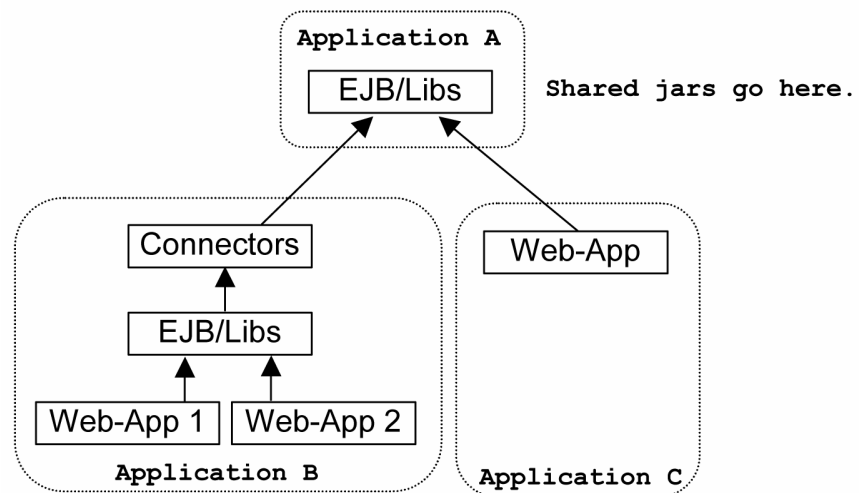


図 7: 共有 jar を含む親アプリケーション

親の宣言は server.xml 内の<application>タグで行われます。図の例では、これは次のように表されます。

```

<application name="A" path="a.ear" />
<application name="B" path="b.ear" parent="A" />
<application name="C" path="c.ear" parent="A" />

```

構成オプションのまとめ

本書全体を通じて、OC4J アプリケーションの開発者が使用可能な多数の構成オプションを説明してきました。このオプションは、基本的な J2SE オプションから汎用 J2EE オプション、そして OC4J 固有のオプションまで、非常に多岐にわたります。次の表は、これらの全オプションを簡単に参照できるようまとめたものです。

クラス・ロード	構成オプション	タイプ
ブートストラップ	コマンドライン: -Xbootclasspath (Sun JVM のみ) システム・プロパティ: sun.boot.class.path (Sun JVM のみ) 前述の jar の META-INF/manifest.mf の Class-Path	JVM JVM JRE
拡張	システム・プロパティ: java.ext.dir。デフォルト: \$JAVA_HOME/jre/lib/ext 前述の jar の META-INF/manifest.mf の Class-Path	JRE JRE
システム	コマンドライン: -classpath コマンドライン: -cp コマンドライン: -jar システム・プロパティ: java.class.path 前述の jar の META-INF/manifest.mf の Class-Path	JRE JRE JRE JRE JRE
グローバル・アプリケーション	グローバル application.xml 内の<library>タグ 前述の jar の META-INF/manifest.mf の Class-Path	OC4J JRE
アプリケーション間	server.xml 内の<application>タグの親属性	OC4J
コネクタ	RAR ファイル: ルートの全 jar 前述の jar の META-INF/manifest.mf の Class-Path	J2EE JRE
EJB/ライブラリ	application.xml の<ejb>タグ orion-application.xml の<library>タグ 前述の jar の META-INF/manifest.mf の Class-Path	J2EE OC4J JRE
Web アプリケーション	WAR ファイル: META-INF/manifest.mf の Class-Path WAR ファイル: META-INF/classes WAR ファイル: META-INF/lib orion-web.xml の<classpath>タグ global-web-application.xml の<library>タグ 前述の jar の META-INF/manifest.mf の Class-Path	J2EE J2EE J2EE OC4J OC4J JRE

網掛け部分のオプションは EAR ファイル内ですべて使用可能です。

構成オプションとしての環境変数(CLASSPATH など)に対するサポートは、JVM および OS によって異なり、そのために移植が不可能であることに注意してください。

J2EE でよくあるクラス・ロード問題

次に、開発者が最もよく直面するクラス・ロード問題について説明します。J2EE のほとんどのクラス・ロード問題は、[参照可能性](#)に関連しており、少数の一連の例外の 1 つとして表面化します。

不十分な参照可能性

この状況は、必要なクラスが現在のスコープ内から参照できない場合に発生します。この問題は、次の例外として表面化することがあります。

- **ClassNotFoundException**。この例外は、クラスを明示的にロードするいずれかのメソッド（`Class.forName()`、`ClassLoader.loadClass()`など）による動的ロード中に発生します。「クラス・ローダーのリンク」で説明したフレームワーク・シナリオが、この例外の典型的な例です。
- **NoClassDefFoundException**。この例外は、コードが `new` 演算子を使用してオブジェクトをインスタンス化しようとした場合、または以前にロードされたクラスの依存関係が解決できない場合に発生します。後者のケースは、レイジー・ロードや隠された依存関係により不明瞭になることがあります。次の状況を考えてみます。
 1. コードによりクラス `Foo` のインスタンスを作成します。
 2. `Foo.doIt()`メソッドのコードに隠れて、`Foo` はクラス `XYZ` をインスタンス化して使用します。
 3. クラス `XYZ` は、`Foo` またはコードとは異なるパッケージおよび異なる `jar` に存在します。
 4. `XYZ` に対する依存関係が隠されていたため、その `jar` はアプリケーションに含まれていません。
 5. レイジー・ロードが原因で、`XYZ` は手順 1 でロードされません。
 6. コードが `Foo.doIt()`を起動し、`XYZ` に対して `NoClassDefFoundException` が発生します。これは予想外の展開のようです。

この状況は一般的であり、依存関係の連鎖が非常に長いことに起因して発生します。このため、原因と結果がさらに切り離されます。

依存関係連鎖が長い場合、複数の名前空間（`ClassLoader`）にまたがる可能性も高くなるため、参照可能性エラーの発生率も高くなります。

過剰な参照可能性

この問題は、クラスが複製される際に発生し、次のように表面化します。

依存 `jar` を各アプリケーションにコピーすることにより管理されるアプリケーション間の依存関係。

アプリケーションの再起動。アプリケーションが再起動される際、コンテナは新しい `ClassLoader` を作成し、その `ClassLoader` を使用してアプリケーションのクラスを再起動します。ただし、このシナリオでは、元のクラスがまだアクセス可能な場合があります。通常、重複クラスの存在は問題にはなりません（フットプリントを除く）。例外が発生するためには、ある名前空間（クラス・ローダー）で作成されたインスタンスが他の名前空間に渡される必要があります。これは、両方の名前空間で参照できる次のような任意の格納機能により可能になります。

ClassCastException。この例外の原因は通常、シンプルかつ明快です。ただし、同一クラス/異なるローダーといった状況では、ソースとターゲットの型で同じクラス名を持っているという事実により、予想外かつ複雑な問題が発生することがあります。一般に、クラスの複製は次の2つのシナリオで発生します。

- **アプリケーション間の依存関係**。依存 jar を各アプリケーションにコピーすることにより管理されるアプリケーション・モジュール間の依存関係。
- **アプリケーションの再起動**。アプリケーションが再起動される際、コンテナは新しいClassLoaderを作成し、そのClassLoaderを使用してアプリケーションのクラスを再起動します。ただし、このシナリオでは、元のクラスがまだアクセス可能な場合があります。

通常、重複クラスの存在は問題にはなりません(フットプリントを除く)。例外が発生するためには、ある名前空間(クラス・ローダー)で作成されたインスタスが他の名前空間に渡される必要があります。これは、両方の名前空間で参照できる次のような任意の格納機能により可能になります。

静的フィールド

グローバル・コレクション

JNDI

通常、後者のケースは問題にはなりません。コピーはシリアライズを通じて作成されるため、同じインスタスが複数の名前空間の間で共有されません。デシリアライズ・プロセスによって作成されるコピーは、ローカルな名前空間のクラスを使用します。

一部のJNDI実装では、ローカルにバインドおよび取得されるオブジェクトをシリアライズしません。このように、同じJVM内の複数のアプリケーションがオブジェクトを格納および取得する場合に、そのインスタスが共有され、重複クラスの問題が発生する可能性があります。

参照可能性の混乱

この問題は、深刻なエラーがある場合に発生します。ただし、このような問題はほとんど起こらず、すべて次のエラーとして分類されます。

IncompatibleClassChangeError。このエラーは、スーパークラスまたはインタフェースが変更されたことを示します。

ClassCircularityError。このエラーは、現在のクラスがそれ自体のスーパークラスまたはスーパーインタフェースの1つとして存在していることを示します。

UnsupportedClassVersionError。このエラーは通常、実行中のJDKバージョンよりも新しいバージョンでコンパイルされたクラスをロードしている場合に発生します。

VerifyError。このエラーは、クラス内のコードがJVMによる制約のいずれかに違反している場合に発生します。

ClassFormatError。このエラーは、クラスのファイル形式が無効であることを示します（一般に破損が原因）。

J2EE のクラス・ロードのベスト・プラクティス

ここまでの説明で、多くの知識を吸収してきました。最後に、前述の多数の概念から導き出される指針を記載しておきます。

依存関係を宣言する。 依存関係を明示してください。アプリケーションを他の環境に移動すると、隠された依存関係や不明な依存関係が残されます。

依存関係をグループ化する。 すべての依存関係が同一レベル以上で参照できるようにしてください。ライブラリの移動が必要な場合は、すべての依存関係が移動後も参照できるようにします。

参照可能性を最小化する。 依存関係ライブラリは、すべての依存関係を満たす最小の参照可能性レベルに配置してください。たとえば、単一 Web アプリケーションでのみ使用されるライブラリは、WAR ファイルの WEBINF/lib ディレクトリに含みます。

ライブラリを共有する。 ライブラリの複製はできるかぎり避けてください。一部のアプリケーション間でクラスを共有するには、「親」属性を使用します。また、全アプリケーション間でクラスを共有するには、グローバル application.xml ファイルの <library> タグを使用します。

構成を移植可能にしておく。 一般に、できるかぎり移植可能な構成を作成してください。構成オプションは次の順序で指定します。

1. 標準 J2EE オプション
2. EAR ファイル内で表現可能なオプション
3. サーバー・レベルのオプション
4. J2SE 拡張オプション

正しいローダーを使用する。

`Class.forName()` をコールする場合は、次のメソッドで返されるローダーを必ず明示的に渡してください。

```
Thread.currentThread().getContextClassLoader()
```

プロパティ・ファイルをロードする場合は、次のメソッドを使用します。

```
Thread.currentThread().getContextClassLoader().getResourceAsStream()
```

結論

本書の説明に対し、読者が「なるほど」と思う瞬間が何回かあったこと、およびクラス・ロードについての疑問の一部が多少でも解明されたことを期待します。

次は、本書で得た知識を実際の J2EE アプリケーションで活用してください。



Oracle9iAS Containers for J2EE におけるクラス・ロード

2003 年 4 月

著者: Bryan Atsatt、Debu Panda

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

海外からのお問合せ窓口:
電話: +1.650.506.7000
ファックス: +1.650.506.7200
www.oracle.com

この文書はあくまでも参考資料であり、掲載されている情報は予告なしに変更されることがあります。万一、誤植などにお気づきの場合は、オラクル社までお知らせください。オラクル社は本書の内容に関していかなる保証もしません。また、本書の内容に関連したいかなる損害についても責任を負いかねます。

オラクル社は、インターネット上での活動を強化するソフトウェアを提供します。

Oracle はオラクル社の登録商標です。
このガイドで使用されているさまざまな製品名およびサービス名には、オラクル社の商標が含まれています。その他のすべての製品名およびサービス名は、各社の商標です。

Copyright © 2003 Oracle Corporation
All rights reserved.