Oracle Internet File System

開発者リファレンス

リリース 9.0.2

2002 年 8 月

部品番号 : J06110-01

ORACLE®

Oracle Internet File System 開発者リファレンス , リリース 9.0.2

部品番号 : J06110-01

原本名：Oracle Internet File System Developer Reference, Release 9.0.2

原本部品番号：A95996-01

原本著者：Alison Stokes, Dennis Dawson

原本協力者：Tom Grant, Francine Hyman, Joyce Peng, David Pitfield.

目次

2 Oracle 9*i*FS の詳細

3 Java API の概要

4 Oracle Internet File System ドキュメントの作成

5 コンテンツ・タイプと属性の拡張

6 任意のメタデータと動作の適用

7 属性の妥当性チェック

8 検索アプリケーションの構築

9 カスタム・パーサーの作成

10 XML および Oracle Internet File System

11 カスタム・レンダラの作成

12 Web インタフェースのカスタマイズ

13 カスタム・サーバーの作成

14 バージョニングの実装

15 セキュリティ

16 セッションおよびトランザクションの管理

17 コンテンツ・タイプの動作のカスタマイズ

18 プログラムによる電子メールの送受信

A　エラー・メッセージ

索引

はじめに

このマニュアルでは、Oracle Internet File System（Oracle 9*i*FS）の動作をカスタマイズしたり、開発プラットフォームとして Oracle 9*i*FS を使用して、新規アプリケーションを作成する方法について説明します。

対象読者

このマニュアルは、XML と Java を使用して独自のファイル・システム・アプリケーションを作成するアプリケーション開発者を対象としています。

Oracle 9*i*FS は、単純なプレゼンテーション層の設計や、ファイル・システムのアクションと外部データベース・スキーマとの複雑な相互作用など、広範囲な開発の可能性をもたらします。どのようなカスタマイズ作業の場合にもフレームワーク全体を完全に把握する必要があるとはかぎらず、Oracle 9*i*FS Java API 全体はごくわずかな部分を占めるにすぎません。

構成

開発者ガイドは、次の章と付録で構成されています。

第 1 章「Oracle Internet File System SDK スタート・ガイド」

Oracle 9*i*FS Software Development Kit（SDK）に基づいてアプリケーションを作成するための要件の概要を説明します。

第 2 章「Oracle 9*i*FS の詳細」

システム要素の概要、主なデータベース機能、開発者が Oracle 9*i*FS のアーキテクチャを理解する上でポイントとなる拡張性について説明します。

第 3 章「Java API の概要」

Oracle 9*i*FS Java API の概要を説明します。この章は、各クラスに固有の情報を調べる場合のリファレンスとして使用できます。

第 4 章「Oracle Internet File System ドキュメントの作成」

用意されている Document クラスをプログラムで処理して、汎用 Document オブジェクトを作成する方法について説明します。また、カスタマイズの概念を構築するための基礎についても説明します。

第 5 章「コンテンツ・タイプと属性の拡張」

Oracle 9*i*FS のコンテンツ・タイプの階層を拡張し、独自の型を持つ情報を管理する方法について説明します。

第 6 章「任意のメタデータと動作の適用」

コンテンツ・タイプに任意のメタデータと動作を適用する方法について説明します。

第 7 章「属性の妥当性チェック」

Oracle 9*i*FS で属性を検証する 3 つの方法、つまり、値のデフォルト、値のドメインおよびクラスのドメインの使用方法について説明します。

第 8 章「検索アプリケーションの構築」

Oracle 9*i*FS のセレクタおよび検索機能を使用して検索アプリケーションを構築する方法について説明します。

第 9 章「カスタム・パーサーの作成」

Oracle 9*i*FS でのカスタム・パーサーの作成と使用について説明します。

第 10 章「XML および Oracle Internet File System」

XML データを処理できるように Oracle 9*i*FS に組み込まれている特定のサポートについて説明します。

第 11 章「カスタム・レンダラの作成」

カスタム・レンダラの作成について説明します。

第 12 章「Web インタフェースのカスタマイズ」

ユーザーが HTML ブラウザを使用してカスタム・ドキュメント・タイプの属性を表示し、変更できるように、カスタム・ユーザー・インタフェースを作成する方法について説明します。

第 13 章「カスタム・サーバーの作成」

カスタム・プロトコル・サーバーおよびエージェントの作成、テストおよび配置プロセスについて説明します。

第 14 章「バージョニングの実装」
Oracle 9*i*FS によるバージョニングの実装方法と、カスタム・アプリケーションにバージョニングを実装する方法について説明します。

第 15 章「セキュリティ」
Oracle 9*i*FS で情報を保護する方法について説明します。

第 16 章「セッションおよびトランザクションの管理」
セッション管理の詳細について説明します。

第 17 章「コンテンツ・タイプの動作のカスタマイズ」
コンテンツ・タイプの動作をカスタマイズする方法について説明します。

第 18 章「プログラムによる電子メールの送受信」
カスタム・アプリケーションで JavaMail API と Oracle 9*i*FS Java API を使用して、Oracle 9*i*FS で電子メールをプログラムにより送受信する方法について説明します。

付録 A「エラー・メッセージ」
アプリケーションの開発中に発生する代表的なエラー・メッセージについて説明します。

関連文書

- 『Oracle Internet File System リリース・ノート』
- 『Oracle Internet File System インストレーションおよび構成ガイド』
- 『Oracle Internet File System セットアップおよび管理ガイド』
- 『Oracle9*i* データベース管理者ガイド』

表記規則

このマニュアルでは、次の表記規則を使用しています。

表記	説明
. . .	例における垂直の省略記号は、例に関係のない情報が省略されていることを示します。
. . .	文またはコマンド構文における水平の省略記号は、例に関係のない文やコマンドの一部が省略されていることを示します。
太字	本文中の太字は、本文または用語集（あるいはその両方）で定義されている用語を示します。
< >	山カッコは、ユーザーが指定する変数を示します。
[ ]	大カッコは、内側のオプション句から 1 つ選択するか、またはまったく選択しなくてもよいことを示します。

1

Oracle Internet File System SDK スタート・ガイド

この章では、Oracle Internet File System（Oracle 9*i*FS）Software Development Kit（SDK）に基づいてアプリケーションを記述するための要件の概要を説明します。様々なアプリケーション・コンポーネントやタスクに固有の情報に進む前に、まずこの章をお読みください。また、第 2 章「Oracle 9iFS の詳細」も、開発者の観点から Oracle 9*i*FS のアーキテクチャの概要を詳しく説明しているため重要です。さらに、第 3 章「Java API の概要」にも必ず目を通してください。Oracle 9*i*FS アプリケーションの中心となる Java API の概要が記載されています。

この章の内容は、次のとおりです。

- 開発スキル
- 開発時のロール
- Oracle 9iFS の開発ツール
- Oracle 9iFS SDK を選択する理由
- Java API
- 基本的な開発タスクと高度な開発タスク
- Oracle 9iFS アプリケーションの配置と管理
- Oracle 9iFS の開発環境
- Oracle 9iFS サンプル・コード
- 上級 Java プログラマのためのクイック・スタート

開発スキル

Oracle 9*i*FS アプリケーションを記述するには、次のスキルが必要です。

必要なスキル

1. **Oracle9*i* データベースのインストール、構成および管理：** ファイル・システムのリポジトリである Oracle9i データベースをインストールして管理できる必要があります。ただし、必要なのは基本的なデータベース管理スキルのみです。
2. **Oracle 9*i*FS のインストール、構成および管理：** Oracle 9*i*FS を Oracle9*i* データベースの一部としてインストールするか、Oracle9*i* Application Server の一部としてインストールするかに関係なく、Oracle 9*i*FS コンポーネントもインストールして管理できる必要があります。Oracle 9*i*FS をデータベースと同じマシンにインストールする場合も、中間層マシンにインストールする場合も、多数のアプリケーション・コンポーネントを管理することになります。特に、開発プロセス中にトラブルシューティングを行う場合には、中間の Oracle 9*i*FS の管理スキルが必要です。
3. **Java でのプログラミング：** ほとんどのアプリケーション・コンポーネントには、ある程度の Java のスキルが必要です。

オプションのスキル

1. **XML 構成ファイルの記述：** アプリケーション・ユーザーの作成やパーサーの登録など、多数のアプリケーション開発タスク用に、XML 構成ファイルを記述できます。この種の構成ファイルを編集するには、Oracle 9*i*FS システム構成に関する XML 構文の知識が必要です。
2. **サーブレットまたは JavaServer Pages の開発：** カスタムの Web ユーザー・インタフェースを記述するには、Java と HTML のスキルが必要です。Oracle 9*i*FS API のコールには Java を使用し、Web ユーザー・インタフェースを形成するページの設計には HTML を使用します。
3. **SQL を使用したデータベースの内容の問合せ：** カスタム・ビューを作成して Oracle 9*i*FS スキーマの内容を問い合せる場合は、Java と SQL のスキルが必要です。
4. **SQL を使用した DDL および DML 操作の実行：** 独自のデータ定義（DDL）、データ操作（DML）またはトランザクション処理（TPL）操作を Java コードに埋め込む場合は、Java と SQL のスキルが必要です。この機能の詳細は、第 16 章「セッションおよびトランザクションの管理」を参照してください。

開発時のロール

前述したすべてのスキルを開発チームの全員がマスターする必要はありません。開発チームの規模が大きくなるにつれて、様々なチーム・メンバーの能力がそれぞれ異なる開発ロールに特化することになります。一般的な特化の例を次に示します。

- **アーキテクチャ設計者：** アーキテクチャ設計者は、アプリケーション・コンポーネントのリスト作成、相互作用、アプリケーション・ロジックおよびユーザー・インタフェース全体の要件の決定を受け持ちます。
- **Web ユーザー・インタフェース設計者：** ユーザー・インタフェース設計者は、Web ユーザー・インタフェースの設計のみを受け持ちます。ユーザー・インタフェースの動作のダミーとなる静的ページを記述する以上のことは行いません。ユーザー・インタフェース設計者は、ユーザー・インタフェースの元になる機能を記述するために、Java でのプログラミングを必要とするようになってきています。
- **Java 開発者：** Java 開発者は、Web ユーザー・インタフェースの元になる Bean の実装やエージェントの記述など、Java 開発タスクを受け持ちます。

このような作業区分は、開発者が習熟する必要のある Oracle 9*i*FS SDK のコンポーネントを決定する上で重要です。アーキテクチャ設計者には SDK とその機能に関する広範囲な知識が必要ですが、Web ユーザー・インタフェース設計者に必要な知識は、HTTP プロトコル・サーバーの動作、サーブレットおよび JSP コードの配置場所、開発者が記述する Bean 内で Java API をコールする方法のみです。

Oracle 9*i*FS の開発ツール

Oracle 9*i*FS 用のアプリケーションを記述するには、次のツールを用意することをお薦めします。

- **Java IDE:** Java コードはすべての Oracle 9*i*FS アプリケーションに含まれているため、Java IDE が必要です。Oracle JDeveloper は Oracle 9*i*FS に理想的な IDE です。JDeveloper9*i* には 9*i*FS 用ライブラリが添付されない可能性大のため、互換性のある JDK バージョンが使用可能な Java IDE を使用してください。
- **XML エディタ：** XML 構成ファイルを記述する場合や、XML ファイルを処理するアプリケーションを記述する場合は、XML エディタも重要なツールとなります。Oracle JDeveloper には、必要な機能（この場合はビルトイン XML エディタ）も組み込まれています。
- **テキスト・エディタ：** Oracle 9*i*FS が外部（つまり、Oracle 9*i*FS ですべてのファイル・システムの内容が格納されるデータベース・スキーマではなく、Oracle 9*i*FS が実行中のファイル・システム）に格納するシステム構成ファイルを変更するために、テキスト・エディタが必要になる場合があります。
- **Oracle Enterprise Manager（OEM）に対する Oracle 9*i*FS の拡張機能：** Oracle 9*i*FS Manager コンポーネントと Oracle 9*i*FS ダッシュボードは、どちらもアプリケーションの開発とテストに役立つツールです。Oracle 9*i*FS Manager は、新規サブクラスを作成

して結果をテストするなど、増分開発に特に役立ちます。また、Oracle 9*i*FS Manager を使用すると、エージェントのような多数のアプリケーション・コンポーネントを監視し、カスタム・オブジェクトのようにファイル・システムに定義され、登録されているすべてのオブジェクトを完全に表示できます。

- **Web オーサリング・ツール：** カスタム Web ユーザー・インタフェースを設計する場合は、Macromedia Dreamweaver などの Web オーサリング・ツールがあれば開発がさらに簡単になります。
- **コマンドライン・ユーティリティ：** この種のユーティリティを使用すると、コマンドライン・インタフェースを通じて個々のコマンドを実行したり、複数のコマンドを実行するセットアップ・スクリプトやクリーンアップ・スクリプトを実行できます。

Oracle 9*i*FS SDK を選択する理由

この開発プラットフォームをアプリケーションの基礎として使用するには、いくつかの理由があります。

1. **ビルトインのコンテンツ管理コンポーネント：** バージョニング、コンテンツ・ベースの検索、拡張可能なメタデータ、チェックイン / チェックアウト、ロックなど、様々なアプリケーションで使用されるコンテンツ管理機能の多くが、Oracle 9*i*FS に再使用可能コンポーネントとして組み込まれており、このような機能の記述とテストの所要時間が短縮されます。
2. **データベース記憶域：** Oracle 9*i*FS ではファイル・システムのリポジトリとして Oracle9*i* データベースが使用されるため、Oracle 9*i*FS のユーザーであれば、TB 規模のコンテンツや数千のユーザーへの拡張性など、データベースが持っている機能を使用できます。また、データベースにコンテンツが格納されるため、Oracle 9*i*FS とアプリケーションでは Oracle Text などのデータベース機能を使用できます。Oracle Text は、Oracle9*i* データベースのテキスト索引付けおよび検索エンジンです。
3. **サーバー側のロジック：** ビジネス・ルールやアプリケーション・ロジックを施行するには、現在、他のファイル・システムには組み込まれていないサーバー側のインテリジェンスを作成する必要があります。このようなサーバー側のロジックを他のファイル・システムに実装するのは不可能でないとしても困難ですが、Oracle 9*i*FS に実装するのは比較的簡単です。
4. **同じコンテンツおよびサーバー側アプリケーション・コンポーネントへのビルトイン・プロトコル・アクセス：** Oracle 9*i*FS には、AFP、SMB、HTTP、WebDAV、FTP、NFS および IMAP4 など、複数のプロトコル・サーバーが組み込まれており、いずれも同じファイルおよびフォルダへのアクセスを提供します。サーバーへのアクセスに使用するプロトコルに関係なく、ファイル・システムの操作（挿入、削除、更新など）には同じアプリケーション・ロジックが適用されます。
5. **コンテンツの解析とレンダリング：** Oracle 9*i*FS の SDK には、汎用的な解析およびレンダリング・フレームワークが組み込まれています。

6. **XML 機能：** XML は構造化データ形式として普及しているため、Oracle 9*i*FS にもデフォルトの XML パーサーおよびレンダラと、DTD 検証などの XML 機能が組み込まれています。

7. **異種ファイル・データ：** Oracle 9*i*FS は、通常は別の種類のリポジトリに格納されるコンテンツ（XML や電子メールなど）を含め、あらゆるタイプのファイルを格納するように設計されています。

8. **ファイルとリレーショナル・データ：** データベースはファイル・システムのリポジトリであるため、Oracle 9*i*FS により、この 2 つの領域を結ぶアプリケーションを記述すると同時に、両方のタイプのコンテンツを同じサーバーに格納することで、開発と管理の両方を簡素化できます。

9. **標準ベースの SDK:** Oracle 9*i*FS SDK では、Java と XML、HTML または SQL を使用するのみでよいため、これらの言語によるプログラミングですでに習得しているスキルを活用できます。

10. **エンド・ユーザーに対する完全な透過性：** 最後に、ファイル・システムの体裁と動作は、従来のファイル・サーバーと同じです。したがって、すべてのユーザー・アプリケーションは Oracle 9*i*FS で自動的に機能し、どのネットワーク・プロトコル経由でサーバーに接続する場合も、特別なスキルを習得する必要はありません。

カスタマイズとアプリケーション

Oracle 9*i*FS SDK を使用する開発では、特定のインスタンスにおけるファイル・システムの動作やデータ表示や、カスタム・ユーザー・インタフェース、アプリケーション・ロジックおよび他のコンポーネントの独自セットを持つ完全なアプリケーションなど、あらゆる範囲のカスタマイズに対処できます。つまり、目的に応じて、単一の開発プロジェクトをごく小規模なものにすることも、きわめて大規模なものにすることもできます。

ファイル・サーバーとしての Oracle 9*i*FS に、カスタマイズを施すことができます。状況（削除操作の動作の変更など）によって、ファイル・システムの動作を変更したり（第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照）、ユーザーに対するデータの表示方法を（ある種の XML コンテンツ用のカスタム・レンダラの実装などにより）変更できます。ただし、この種の開発プロジェクトのポイントは、ファイル・サーバーを置き換えることではなく、変更を加えることです。

これに対して、多くの大規模アプリケーションは、ファイル・サーバー・アプリケーションを必要としません。この種のアプリケーションで必要なのは一部のプロトコル・サーバー（通常は、Web アプリケーション・アクセス用の HTTP と、コンテンツの大量アップロードおよびダウンロード用の FTP）のみです。この種のアプリケーションの作成者は、プロトコル・サーバーの範囲を超えた機能を簡単に開発できるという理由から、ファイル・サーバーよりも Oracle 9*i*FS SDK を必要とします。

このマニュアルは、2 種類の開発者を対象としています。カスタマイズに関心を持っている開発者と、Oracle 9*i*FS プロトコル・サーバーと付属の Web インタフェースを使用するかどうかに関係なく、アプリケーションを作成する開発者です。

Oracle 9*i*FS のカスタマイズとアプリケーションの例

Oracle 9*i*FS SDK の主なメリットの 1 つは、同じサーバー上で複数のアプリケーションを簡単に開発して管理できることです。複数の従来型ファイル・サーバーから単一の Oracle 9*i*FS サーバーに統合することにより、同一のコンテンツを複数のアプリケーションとカスタマイズのターゲットとして使用できます。たとえば、Oracle 9*i*FS 用に記述されたデジタル資産管理システムで管理されていたファイルを、同じファイル・サーバーに配置された Web コンテンツ管理システム経由で Web サイト用にステージングできます。

ここでは、作動中の Oracle 9*i*FS SDK の様々なコンポーネントを具体的に示すために（また、独自プロジェクトの有効範囲の設定の参考として）、Oracle 9*i*FS を使用して作成できるアプリケーションの例をいくつか示します。開発者がこのようなアプリケーションについて行う、実装上の決定例も示します。実際には、このような決定ができるのみでなく、Oracle 9*i*FS SDK では、このような実装上の決定を柔軟に行えることが、このプラットフォームの大きなメリットの 1 つです。

ユーザーに対する変更の通知

ユーザーが Oracle 9*i*FS の特定のディレクトリでドキュメントを共有する必要があるとします。また、ディレクトリのコンテンツが変更された場合（他のユーザーがファイルを追加、削除またはアップデートした場合）に、電子メールでその通知を受け取る必要があるとします。Oracle 9*i*FS にはエージェント・フレームワークが組み込まれているため、Oracle 9*i*FS でそのようなアプリケーションを作成するのは比較的簡単です。

この機能を開発するには、このエージェント・フレームワークを利用して、ディレクトリのコンテンツのリストを管理し、コンテンツに変更があったかどうかを定期的にチェックし、変更が検出された場合は電子メール・リストのメンバーに通知する新規エージェントを追加します。エージェントは、Oracle 9*i*FS SMTP サーバーを使用して、どのメール・サーバーにも電子メールを送信できます。

この通知エージェントの開発と配置の手順は、次のとおりです。

1. Oracle 9*i*FS Manager を使用して、Oracle 9*i*FS ユーザーを作成します（Oracle 9*i*FS にユーザーが存在しない場合）。Oracle 9*i*FS の共有ディレクトリにアクセスする必要があり、Oracle 9*i*FS のユーザー・アカウントを所有しているユーザーは、guest アカウント権限のみで共有ディレクトリにアクセスできます。
2. このユーザーがまだ Oracle 9*i*FS ユーザーとして定義されていない場合は、最初に Oracle 9*i*FS Manager（Oracle Enterprise Manager の管理インタフェースへの拡張機能）または XML 構成ファイルを通じて、そのユーザーを作成する必要があります。実際には、電子メールを受け取るユーザーは Oracle 9*i*FS ユーザーでなくてもかまいません。
3. エージェントを記述します。このエージェントは、Oracle 9*i*FS Java API をコールする Java アプリケーションです。この場合、エージェントは API をコールして、ディレクトリのコンテンツを検討し、現行のコンテンツをオリジナルのコンテンツを示すファイルと比較します。
4. エージェントを配置します。

5. エージェントを登録します。
6. SMTP プロセスが実行中であることを確認します。
7. 削除できないように、ディレクトリに対するセキュリティを変更します。

コンテンツ管理システム

企業が社内 Web サイトを実装することを考えているとします。ただし、高コストでメンテナンスの困難な Web コンテンツ管理システムによるオーバーヘッドは望んでいません。かわりに、全員が共有するファイル・サーバーの最上位に簡単なアプリケーションを構築します。また、この企業は、将来的に他の種類のコンテンツ管理アプリケーションを計画しているため、このプロジェクトを慎重に見守ろうと考えています。

コンテンツ管理システムのポイントは、Oracle 9*i*FS でコンテンツを作成し、ステージングすることです。アプリケーションは、単にすべてのサイトのコンテンツの Access Control List（ACL）を Published に変更し、読取り可能にすることで、Web サイトを公開できます。これに対して、一部の Web サイト管理者は、Web サイトを Oracle 9*i*FS でステージングしてから、そのコンテンツをアクティブな Web サーバーにコピーする方法を望む場合があります。

開発チームと Web サイト管理者がどのアプローチを選択した場合も、このアプリケーションの実装方法はほぼ同じです。

1. Web サイトのコンテンツのステータスを追跡するために必要なカテゴリを定義します。
2. コンテンツのステージングに使用するディレクトリを作成します。
3. 各ユーザーの Home ディレクトリに Review サブディレクトリを作成します。
4. 検討用のコンテンツをユーザーの Home ディレクトリにリンクするエージェントを記述します。
5. Web ユーザー・インタフェースの元になる Bean を記述します。
6. Web ユーザー・インタフェースを設計します。
7. HTML ページを JavaServer Pages（JSP）またはサーブレットに変換します。

XML ベースの保険料請求システム

保険会社が、すべての保険料請求を XML ファイルとして標準化するように決定したとします。各請求は業界標準の Document Type Definition（DTD）に準拠しているため、すべての要素は細部まで理解できます。この XML への切替えで重要なことは、ビジネス・インテリジェンスのために XML をマイニングできることです。ただし、会社が使用している分析ツールは、XML 用ではなくリレーショナル・データ用に記述されています。Oracle 9*i*FS には、解析を通じてファイル・ベースの XML コンテンツをリレーショナル・データに変換する方法が用意されています。

この保険料請求システムを実装する手順は、次のとおりです。

1. DTD に基づいてサブクラスを作成します。このサブクラスには、解析対象となる XML ファイルの各要素に対応するドキュメント属性が含まれます。
2. XML コンテンツを DTD に基づいて検証するかどうかを決定します。DTD の検証を有効化する方法の詳細は、第 10 章「XML および Oracle Internet File System」を参照してください。
3. このサブクラスを XML パーサーに対応付けます。
4. Web 上で XML を表示できるように JSP を設計します。
5. JSP コールの元になる Bean を記述します。
6. このサブクラスをメイン JSP に対応付けます。
7. 解析した XML コンテンツに基づいてカスタム・ビューを作成します。このビューにより、すべてのドキュメントと共通属性 ODM_DOCUMENT の格納に使用される実表のコンテンツと、解析済み属性の格納用に Oracle 9*i*FS で作成された表が結合されます。

このアプリケーションのもう 1 つの特長は、簡単に想像できます。たとえば、各請求の総額を調べ、特定の金額を超える請求について調査人にアラートを出すエージェントを記述するとします。XML コンテンツは、Web アプリケーション経由でアクセスするユーザーのために、より判読しやすい形式で表示できます。

ドキュメント・リポジトリ

ドキュメント・リポジトリに必要な機能は、Oracle 9*i*FS がデフォルトで備えている機能の他にごくわずかです。サンプル・ドキュメント・リポジトリには、カテゴリ（BookInformation、JournalInformation など）が必要であるとします。ユーザーがこれらのカテゴリの属性の設定を入力しなければ、他のユーザーはアップロードしたファイルを表示できません。コンテンツに基づく検索とカテゴリ検索を組み合せた独自の検索 Web ページを設計する必要があります。他の Web ページでは、カテゴリをブラウズし、特定の設定（西部のすべての書籍など）に該当するファイルをすべて表示できます。

このプロジェクトに必要な手順は、次のとおりです。

1. ドキュメント・リポジトリに使用するカテゴリを定義します。
2. ファイルをアップロードしたユーザーが必要な属性設定を入力するまでは、ファイルの ACL を Private（つまり、そのユーザーしか参照できないよう）に設定するエージェントを開発します。これと同じエージェントが、ACL を Published に設定し、誰でも読み取れるようにします。
3. コンテンツとカテゴリの検索に必要なカスタム Web ユーザー・インタフェースを構築します。

削除操作の動作の変更

多数の組織で、ドキュメント保持要件や他のビジネス・ルールにより、ユーザーが実際にファイルを削除するのを禁止するように要求しています。取引、会計または法的処理の全面復元が会計検査で必要となるような場合、これらを確実に復元できるようにするため、各ファイルのすべてのバージョンを保持しなければならない場合もあります。一方、管理者またはマネージャは、システムからパージされる前にドキュメントを最初に検討する必要がありますが、ドキュメントを永続的に保存する必要があるとはかぎりません。

この機能、つまり、ファイル・システムでの削除操作の動作を効率的に変更することが、標準的なファイル・システムには適さないことは明らかです。ただし、Oracle 9*i*FS では、単一のオーバーライドのみで削除操作を変更できます。

Java API

Oracle 9*i*FS SDK のうち、習得可能で最も重要な側面は、Java API です。Oracle 9*i*FS は Java で記述されているため、このファイル・システムの元になるオブジェクト階層は開発者に公開されています。ドキュメント、フォルダ、ユーザー、グループ、属性など、ファイル・システムを形成するすべてのオブジェクトは、Java オブジェクトとして存在します。このように標準的なクラスには、Java コードでコールできるメソッドが含まれます。

Java コードを記述する前に、第 3 章の内容を検討してください。Oracle 9*i*FS SDK を使用した開発の成否は、個々の Java オブジェクトのみでなく、オブジェクトの相互関係を理解しているかどうかによって決まります。たとえば、Oracle 9*i*FS では、ユーザーという用語は、実際には DirectoryUser オブジェクトと他の Java オブジェクトで構成されます。Java API の概要は、第 3 章「Java API の概要」を参照してください。

基本的な開発タスクと高度な開発タスク

開発タスクは、基本タスクと高度なタスクに分類することができます。基本タスクは、サブクラス化など、Oracle 9*i*FS アプリケーションを記述するときに一般的に実行するタスクです。通常、基本タスクは技術的な専門知識をあまり必要としません。これに対して、高度なタスクは限定的であり、技術的な専門知識を必要とします。

表 1-1 に、Oracle 9*i*FS の基本な開発タスクと高度な開発タスクの概要を示します。

表 1-1 Oracle 9*i*FS の基本的な開発タスクと高度な開発タスクの比較

基本	高度
サブクラス化	カスタム・レンダラの記述
サーブレットベースの Web ユーザー・インタフェースの記述	オーバーライドの開発
JSP ベースの Web ユーザー・インタフェースの記述	Oracle 9*i*FS のトランザクション・コンテキスト内での SQL または PL/SQL の実行
API を使用した検索の実行	Oracle 9*i*FS の表のカスタム・ビューの作成
カスタム属性の適用と移入	
エージェントの開発	
XML の操作	
カスタム・パーサーの記述	

Oracle 9*i*FS の開発環境

Oracle 9*i*FS アーキテクチャの概要は、第 2 章「Oracle 9iFS の詳細」を参照してください。この項では、Oracle 9*i*FS アプリケーションのコンポーネント・タイプと、各タイプをいつ使用するかについて説明します。

Oracle 9*i*FS アプリケーションのコンポーネント

Oracle 9*i*FS アプリケーションは、次の一部またはすべてのコンポーネントで構成できます。各コンポーネントの記述と配置に必要なスキルを、アプリケーション・コンポーネントごとに示します。

- **標準クラス：** 新規属性を追加すると、Oracle 9*i*FS を定義する標準クラス（`PublicObject`、`DirectoryUser` など）を拡張できます。
- **カスタム・クラス：** 通常は、標準クラスを拡張するのではなく、その標準クラスのサブクラスを作成します。
- **パーサー：** パーサーは、ドキュメントを挿入または更新する前に処理を実行します。たとえば、パーサーの一般的な用途は、ファイルのコンテンツを抽出し、Oracle 9*i*FS の属性として外部化することです。
- **レンダラ：** レンダラは、ファイルまたはフォルダ要求のストリームを変更し、エンド・ユーザーへの表示方法を効率的に変更します。
- **エージェント：** エージェントは、イベントに応答したり、応答を必要とする特定のシステム条件（新規ファイルがフォルダに表示されているかどうかなど）の有無を定期的にチェックできるプロセスです。

- **オーバーライド：** オーバーライドは、標準クラスのメソッド（フォルダに対する `AddItem()` など）がコール時に実行する処理を効率的に変更します。
- **カスタム Web ユーザー・インタフェース：** カスタム Web ユーザー・インタフェースは、Oracle 9*i*FS に用意されている Web ユーザー・インタフェースを拡張または置換できます。サーブレットまたは JavaServer Pages（JSP）を使用して、カスタム Web ユーザー・インタフェースを記述できます。

使用するコンポーネント

特定のタスクを実行するときに使用する必要のあるコンポーネントは、前述のリストを見ただけでは判断できない場合があります。表 1-2 に、一般的な開発目標を達成するために必要なコンポーネントの概要を示します。

表 1-2 開発目標の達成に必要なコンポーネント

タスク	コンポーネント
ファイルまたはフォルダへの任意のメタデータの適用	カテゴリ
特定のファイル形式に対する同一メタデータの適用	サブクラス、値のデフォルト
イベントへの応答	エージェント、オーバーライド
ファイルにエンコードされたメタデータの抽出	サブクラス、パーサー
パーサーと特定のファイル形式との対応付け	サブクラス、パーサー
Oracle 9*i*FS でのドキュメントの表示方法の変更	サブクラス、レンダラ
カスタム Web ユーザー・インタフェースの構築	サーブレット、JSP
カスタム Web ユーザー・インタフェースと特定の型のコンテンツとの対応付け	サブクラス、JSP
ファイル・システム操作の動作の変更	オーバーライド
非同期タスクの実行	エージェント
Oracle 9*i*FS 操作内での SQL コマンドの実行	オーバーライド

開発プロジェクトによっては、前述のコンポーネントの一部のみを使用する場合もあります。

コンポーネントと使用ツール

前述のように、開発タスクを実行する場合は、カスタマイズの実行方法として Java コードの記述、XML 構成ファイルの作成または Oracle 9*i*FS Manager の使用の間で頻繁に切り替えることができます。ただし、この 3 つのツールすべてを、どの開発タスクにも使用できるわけではありません。多くの場合、選択肢は 3 つのツールのうち 1 つまたは 2 つに限定されます。

表 1-3 に、これらの各アプリケーション・コンポーネントの開発に使用可能なツールを示します。

表 1-3 アプリケーション・コンポーネント開発ツール

タスク	Java	XML	9*i*FS Manager
サブクラス			
サブクラスの作成	○	○	○
サブクラスへのカスタム属性の追加	○	○	○
サブクラスへのカスタム・メソッドの追加	○		
汎用カスタム属性			
カテゴリの作成	○		○
オブジェクトへのカテゴリの適用	○		
オブジェクト内の属性値の設定	○	○	
クラス固有のカスタム属性			
サブクラスへの属性の追加	○	○	○
標準クラスのすべてのインスタンスへの属性の追加	○		○
パーサー			
パーサーの作成	○		
パーサーの登録	○	○	○
パーサーとサブクラスの対応付け	○	○	○
レンダラ			
レンダラの作成	○		
レンダラの登録	○	○	○
レンダラとサブクラスの対応付け	○	○	○
サーブレット			

サーブレットの作成	○		
JavaServer Pages			
JavaServer Pages（JSP）の作成	○		
JSP とサブクラスの対応付け	○	○	○
エージェント			
エージェントの作成	○		
エージェントの登録			○
オーバーライド			
オーバーライドの作成	○		
カスタム・ビュー			
カスタム・ビューの作成	○		

Oracle 9*i*FS アプリケーションの配置と管理

アプリケーション・コンポーネントの開発後は、それを配置する必要があります。表 1-4 に、これらのコンポーネントの格納場所を示します。

表 1-4 アプリケーション・コンポーネントの配置場所

コンポーネント	配置場所	必須ディレクトリ	推奨ディレクトリ
JavaBeans	ファイル・システム		/custom_classes
パーサー	ファイル・システム		/custom_classes
レンダラ	ファイル・システム		/custom_classes
JSP	Oracle 9*i*FS	/ifs/jsp-bin	
サーブレット	ファイル・システム		/custom_classes
ADM、DEF ファイル	ファイル・システム	/settings	
XML 構成ファイル	Oracle 9*i*FS	なし	なし
エージェント	ファイル・システム		/custom_classes
オーバーライド	ファイル・システム		/custom_classes

表 1-4 で、配置場所の欄がファイル・システムとなっている場合は、アプリケーション・コンポーネントを Oracle 9*i*FS が配置されているのと同じファイル・システムに配置すること

を意味します。必須ディレクトリと推奨ディレクトリは、Oracle 9*i*FS がインストールされているディレクトリのサブディレクトリです。これに対して、配置場所の欄が Oracle 9*i*FS となっている場合は、アプリケーション・コンポーネントを Oracle 9*i*FS に（FTP 経由で XML ファイルをアップロードするなどの方法で）挿入して配置することを意味します。

前述のように、`/custom_classes` ディレクトリは Java アプリケーション・コンポーネントの格納用に設計されています（これらのコンポーネントは別のディレクトリに配置することもできますが、その場合は、そのディレクトリを含めるために Oracle 9*i*FS インスタンスの CLASSPATH 設定の変更が必要になることがあります。CLASSPATH 設定の詳細は、表 1-6「環境変数の要件」を参照してください）。

カスタマイズ後の再起動

アプリケーション・コンポーネントを配置するには、Oracle 9*i*FS サーバー・プロセスの再起動が必要になる場合があります。Oracle 9*i*FS では複数のキャッシュがメンテナンスされますが、これらのコンポーネントを配置するときに、キャッシュが即時にはリフレッシュされないことがあります。また、システム設定の変更結果は、プロトコル・サーバーを再起動するまで適用されないことがあります。Oracle 9*i*FS Java プロセス（リポジトリ、プロトコル・サーバーおよびエージェント）の起動手順と終了手順は、『Oracle Internet File System セットアップおよび管理ガイド』の第 2 章「Oracle 9*i*FS ドメインの管理」を参照してください。

表 1-5 に、アプリケーション・コンポーネントの配置後に Oracle 9*i*FS プロセスを再起動する必要があるかどうかを示します。

表 1-5 コンポーネントの配置後に再起動する必要があるプロセス

コンポーネント	再起動するプロセス
標準クラスへの拡張機能	すべて
サブクラス	すべて
JSP	なし
サーブレット	HTTP
パーサー	すべて
レンダラ	すべて
エージェント	なし
オーバーライド	すべて
プロトコル・サーバー設定	影響を受けるプロトコル・サーバー

システム構成ファイル

Oracle 9*i*FS では、次の 2 種類のシステム構成ファイルを使用します。

1. **外部ファイル・システムに格納されているファイル：** プロトコル・サーバーおよび他の Oracle 9*i*FS アプリケーション・コンポーネントがどのように動作するのかを決定する構成設定は、Oracle 9*i*FS がインストールされるディレクトリのサブディレクトリ `/settings` に、プレーン・テキスト・ファイルとして保存されます。これらのファイルで Oracle 9*i*FS のオブジェクト階層がカスタマイズされることはありません。

2. **Oracle 9*i*FS に挿入される XML ファイル：** この種のファイルは、Oracle 9*i*FS の XML パーサーで認識される特殊な構文に従う必要があります。通常、この種のファイルは、フォルダに対する属性の設定や新規ユーザーの作成など、Oracle 9*i*FS SDK 内でオブジェクトの作成または変更を行います。

Oracle 9*i*FS では、Oracle 9*i*FS のアプリケーション・コンポーネントがインストールされているファイル・システムに格納される構成情報が、Oracle Internet File System の以前のバージョンに比べて少なくなっていることに注意してください。Oracle 9*i*FS の構成方法の詳細は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

システムのメンテナンス

Oracle 9*i*FS のカスタマイズおよびアプリケーションでは、同じタイプの管理と運用が必要になります。これには、重要なビジネス・アプリケーションで必要とされる障害時リカバリ計画が含まれます。これは、ファイル・システムのコンテンツとアプリケーション・コンポーネントの両方をバックアップする必要があることを意味し、そうすることで、必要時にシステムを再構築することができます。次の 2 つの事項を考慮します。

- **ファイル・システムのコンテンツ：** Oracle 9*i*FS をバックアップすることで、すべてのコンテンツ（ファイルとフォルダ）に加え、それらを定義するユーザー、グループ、ACL および他の各要素をリストアできます。これには、Oracle 9*i*FS インポート / エクスポート・ユーティリティを使用します。『Oracle Internet File System セットアップおよび管理ガイド』の第 8 章「データの移行、バックアップおよびリストア」を参照してください。

- **アプリケーション・コンポーネント：** エージェントやパーサーなど、複数のアプリケーション・コンポーネントが、他の Oracle 9*i*FS コンポーネントとともにファイル・システムに格納されるため、これらのファイルとデータベースの内容のバックアップを作成する必要があります。

アプリケーション・コンポーネントを単一の Oracle 9*i*FS インスタンスで再作成する場合、または同じアプリケーションを複数の中間層マシンにインストールする場合は、アプリケーション・コンポーネントのインストール・スクリプトを記述することをお薦めします。スクリプトでは、Oracle 9*i*FS を実行中のファイル・システムにコンポーネントを配置できます。また、FTP などの標準プロトコルと Oracle 9*i*FS 独自のプロトコルである CUP を通じて、コンポーネントを Oracle 9*i*FS に挿入することもできます。

コマンドライン・ユーティリティ・プロトコル・サーバー（CUP）

Oracle 9*i*FS には、管理者および開発者用に設計された特別なプロトコル・サーバー CUP が組み込まれています。CUP を通じて、拡張属性の表示など、FTP や他のプロトコル・サーバーに付属しない多数の Oracle 9*i*FS コマンドを呼び出すことができます。CUP は、1 回限りのプロトタイプ化タスクを実行するときや、アプリケーション・コンポーネントの配置または更新スクリプトを実行するときに役立ちます。

CUP の実行方法は、『Oracle Internet File System セットアップおよび管理ガイド』の付録 A「コマンドライン・ユーティリティ・リファレンス」を参照してください。

Oracle 9*i*FS サンプル・コード

サンプル・コードはこの『Oracle Internet File System 開発者リファレンス』に記載されていますが、Oracle 9*i*FS インスタンスの `/root/ifs/examples/devdoc` フォルダにもインストールされています。このフォルダは、任意の Oracle 9*i*FS プロトコル・サーバーからアクセス可能です。このマニュアルに記載されているサンプル・コードへのフルパスは、次のとおりです。

```
<ORACLE_HOME>¥9ifs¥samplecode¥oracle¥ifs¥examples¥devdoc
```

この『Oracle Internet File System 開発者リファレンス』で提供されるサンプル・コードの他にも、Oracle 9*i*FS 製品には、製品開発者によって記述された多くのサンプル・コードが含まれています。詳細は、「上級 Java プログラマのためのクイック・スタート」または次のファイルを直接参照してください。

```
<ORACLE_HOME>9ifs¥samplecode¥oracle¥ifs¥examples¥api¥readme.htm
```

その他のサンプル・コード、技術的概要およびチュートリアルは、OTN-J（Oracle Technology Network Japan）から入手できます。

`http://otn.oracle.co.jp/`

スタート・ガイド

サンプル・コードを使用するために必要な手順は、次のとおりです。

1. Oracle 9*i*FS のインストールと構成を行います。詳細は、Oracle Internet File System のインストレーションおよび構成ガイドを参照してください。
2. `http://java.sun.com/j2se/` から、Java SDK（Software Development Kit。JDK ともいう）を開発ワークステーションにダウンロードおよびインストールします。または、Oracle 9*i*FS がインストールおよび構成されているマシン上で開発を行う場合、JDK のダウンロードおよびインストールは、Oracle 9*i*FS マシンに対して行います（Solaris では、ほとんどの場合 JDK はすでにインストールおよび構成されています）。

- オプションとして、Oracle JDeveloper などの Java 統合開発環境をインストールおよび構成することもできます。

3. 表 1-6 に示す CLASSPATH および PATH 環境変数を設定します。設定例は、「CLASSPATH、PATH および LD_LIBRARY_PATH 変数の例（UNIX）」または「CLASSPATH および PATH 変数の例（Windows NT/2000）」を参照してください。

表 1-6 環境変数の要件

環境変数	設定に含める要素	説明
CLASSPATH	<ORACLE_HOME>/9iFS/settings	Oracle 9*i*FS インスタンスの構成ファイル（*.properties）を格納するディレクトリ。
	<ORACLE_HOME>/9iFS/lib/repos.jar	Oracle 9*i*FS のコア・ソフトウェアおよび API セットを含む Java パッケージ。
	<ORACLE_HOME>/9iFS /lib/utils.jar	
	<ORACLE_HOME>/9iFS /lib/adk.jar	
	<ORACLE_HOME>/9iFS /lib/email.jar	
	<ORACLE_HOME>/lib/xmlparserv2.jar	Oracle9*i* Database Server Java XDK（XML Developer's Kit）。
	<ORACLE_HOME>/jdbc/lib/classes13.zip	Oracle データベースと通信するために Oracle 9*i*FS によって使用される JDBC ドライバ。このドライバは JDK のバージョンに固有であるため、JDK 1.2 を使用している場合、CLASSPATH には classes13.zip ではなく classes12.zip を含める必要があります。
	<ORACLE_HOME>/jsp/lib/ojsp.jar	Oracle JavaServer Pages Engine。
	<ORACLE_HOME>/lib/servlet.jar	第 12 章「Web インタフェースのカスタマイズ」で説明されているサンプル・アプリケーションで必要。
LD_LIBRARY_PATH	<ORACLE_HOME>/lib	Solaris での Oracle ライブラリの場所。

表 1-6 環境変数の要件（続く）

環境変数	設定に含める要素	説明
PATH	<ORACLE_HOME>¥bin	Windows NT または Windows 2000 での Oracle ライブラリの場所。
	< マウント・ポイントまたはドライブ :>/jdk1.3/bin または $JAVA_HOME（JAVA_HOME 環境変数を設定している場合）	JDK の場所。
JAVA_HOME	< マウント・ポイントまたはドライブ :>/jdk1.3/bin	JDK が常駐するディレクトリ。

CLASSPATH、PATH および LD_LIBRARY_PATH 変数の例（UNIX）

ここでは、Oracle 9*i*FS 用に構成された開発用 Solaris マシンの `.profile` の例を示します。

```
umask 022
ORACLE_BASE=/data1/home/oracle/product; export ORACLE_BASE
ORACLE_HOME=/data1/home/oracle/product; export ORACLE_HOME
LD_LIBRARY_PATH=$ORACLE_HOME/lib:/lib:/usr/lib; export LD_LIBRARY_PATH
ORACLE_SID=ifs9kw; export ORACLE_SID
ORACLE_TERM=vt100;export ORACLE_TERM
DISPLAY=localhost:0;export DISPLAY
PATH=$ORACLE_HOME/bin:/usr/java1.3/bin:/usr/sbin:/usr/bin:
/usr/dt/bin:/usr/openwin/bin:/bin:/usr/ucb:$PATH; export PATH
JAVA_HOME=/usr/java1.3/bin:.:; export JAVA_HOME
CLASSPATH=$ORACLE_HOME/9ifs/settings:$ORACLE_HOME/9ifs/lib/repos.jar:$ORACLE_
HOME/9ifs/lib/utils.jar:$ORACLE_HOME/9ifs/lib/adk.jar:$ORACLE_
HOME/9ifs/lib/email.jar:$ORACLE_HOME/lib/xmlparserv2.jar:$ORACLE_
HOME/jdbc/lib/classes13.zip:$ORACLE_HOME/jsp/lib/ojsp.jar:$ORACLE_
HOME/lib/servlet.jar:$ORACLE_HOME/9ifs/samplecode:$JAVA_HOME/src.jar:$JAVA_
HOME/bin:.:; export CLASSPATH
```

CLASSPATH および PATH 変数の例（Windows NT/2000）

ここでは、Oracle 9*i*FS 用に構成された開発用 Windows 2000 マシンの CLASSPATH および PATH 設定の例を示します。これらの環境変数は、コントロール パネルの「システム」→「詳細」タブ→「環境変数」ボタンで確認できます。

CLASSPATH:

```
d:¥oracle¥ora90¥9ifs¥lib;d:¥oracle¥ora90¥9ifs¥settings;d:¥oracle¥ora90¥9ifs¥lib¥repo
s.jar;d:¥oracle¥ora90¥9ifs¥lib¥utils.jar;d:¥oracle¥ora90¥9ifs¥lib¥adk.jar;d:¥oracle¥
ora90¥9ifs¥lib¥email.jar;d:¥oracle¥ora90¥LIB¥xmlparserv2.jar;d:¥oracle¥ora90¥jdbc¥li
b¥classes13.zip;d:¥oracle¥ora90¥jsp¥lib¥ojsp.jar;d:¥oracle¥ora90¥9ifs¥samplecode;
```

PATH:

```
D:¥jdk1.3¥bin;D:¥oracle¥ora90¥bin;
```

この CLASSPATH の例では、ORACLE_HOME ディレクトリは、ORACLE_HOME 環境変数を参照するのではなく、CLASSPATH 環境変数にハード・コードされています。必要に応じて ORACLE_HOME 環境変数を作成することもできます。この場合は、フル・パス名ではなく、変数を使用してディレクトリ構造に接頭辞をつけます。

ただし、新たに Oracle ホームを作成する時点で、ORACLE_HOME 変数が Windows プラットフォームのパス情報と競合することを、Oracle Universal Installer によって警告されます。（Windows NT の場合、ORACLE_HOME のパスは、Oracle Universal Installer でインストールされるクライアント・アプリケーションの Oracle Home Selector によって動的に修正されます）。

上級 Java プログラマのためのクイック・スタート

上級 Java プログラマで、すぐにでも作業を始めたいユーザーは、`<ORACLE_HOME>9ifs/samplecode/oracle/ifs/examples` の `/api`、`/email` および `/servers` サブディレクトリに格納されているサンプル・コードを出発点として使用できます。各ディレクトリ内に、サンプル・コードの実行に関する手順が記載された `readme.htm` ファイルがあります。

ここでは、`/api` サブディレクトリにあるサンプル・コードの使用を開始するための初期設定で必要な手順を示します。これらは、Oracle 9*i*FS インスタンスがあるマシンとは別のマシン上での開発手順です。Oracle 9*i*FS がインストールされたマシン上で作業している場合は、手順 2 を省略してください。

1. Oracle 9*i*FS を起動します（まだ動作していない場合）。
2. 開発マシン上で次の手順を行います。
   a. 開発マシン上に、Oracle9*i* Client ソフトウェアをインストールおよび構成します。
   b. 開発マシン上に、Oracle 9*i*FS ソフトウェアを管理専用インストールでインストールおよび構成します（あるいは、既存の Oracle 9*i*FS ドメインに追加ノードとして

Oracle 9*i*FS ソフトウェアをインストールおよび構成することができます。ただし、管理専用インストールの方がリソースを節約できます)。

管理専用インストール

– 次の場所にある `IfsDefault.properties` ファイルを変更します。

```
<ORACLE_HOME>/9iFS/settings/oracle/ifs/server/properties
```

Oracle 9*i*FS インスタンスを格納している Oracle データベース・サービスの名前が含まれるようにします。

```
IFS.SERVICE.JDBC.DatabaseUrl=jdbc:oracle:oci8:@<service_name>
```

– apiuser の接続文字列にこのサービス名が含まれるように `setup.sql` スクリプトを変更します。

```
connect apiuser/apiuser@<service_name>
```

3. 表 1-6 で説明されているとおりに、環境変数 CLASSPATH、PATH および LD_LIBRARY_PATH（LD_LIBRARY_PATH は、Solaris に固有。Windows にはありません）を変更します。
4. JDK コンパイラが含まれるように PATH を変更します（表 1-6 を参照)。
5. `/samplecode/oracle/ifs/examples/api` のすべてのサブディレクトリにあるソース・ファイルをコンパイルします。次に例を示します。

```
cd <ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/api/utils
javac *.java
cd <ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/api/server
javac *.java
<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/api
javac *.java
```

6. `setup.sql` スクリプトを実行し、コード・サンプルを実行するために必要なデータベース・オブジェクトを作成します。

```
SQL>connect ifssys/<pswd> as sysdba
Connected.
SQL>@setup.sql
```

7. 手順 5 のコンパイルで生成された GlobalSetup.class ファイルを実行します。

```
java oracle.ifs.exmaples.api.GlobalSetup system manager9ifs
   IfsDefault <pswd>
```

GlobalSetup を実行している間、コンソールに出力画面が表示されます。GlobalSetup は、サービスの開始、セッションの確立、複数のサンプル Oracle 9*i*FS ユーザーの作成、

複数のサンプル・ドキュメントとサンプル・フォルダの作成、およびその他の基本タスクを行います。

出力画面およびソース・コード（GlobalSetup.java）を確認すると、Oracle 9*i*FS の基本操作を理解できます（サービスの作成、セッションの確立およびロギングを実行するメソッドは、親クラス BaseSample.java によって提供されていることがわかります）。

8. Oracle Text データを再索引付けします（一部のサンプルにのみ必要。詳細は、readme.htm ファイルを参照）。Oracle Text データを再索引付けすると、サンプル・データだけでなく Oracle 9*i*FS インスタンス内のすべての内容が再索引付けされます。そのため、データベース内のデータ量によってはこの作業に時間がかかる場合があります。

```
<ORACLE_HOME>¥9ifs¥samplecode¥oracle¥ifs¥examples¥api>sqlplus /nolog
SQL>connect ifssys/<pswd>
Connected.
SQL>exec ctx_ddl.sync_index('ifs_text');

PL/SQL procedure successfully completed.
SQL>
```

これで、/api ディレクトリとサブディレクトリ内のすべてのサンプル・コードを使用して作業できます。詳細は、<ORACLE_HOME>9ifs/samplecode/oracle/ifs/examples/api サブディレクトリ内の readme.htm を参照してください。

2

Oracle 9*i*FS の詳細

この章では、Oracle 9*i*FS のシステム要素、主要なデータベース機能および拡張性のポイントについて説明します。この章は、開発者がインストール時の製品の基本的なカスタマイズ計画を作成し、既存の要素を修正したり新規要素を作成して機能を追加できるように、Oracle 9*i*FS のアーキテクチャについて理解を深めることを目的としています。この章の内容は、次のとおりです。

- オブジェクトの管理
- ファイル・システムにもたらされるデータベース機能
- アーキテクチャの概要
- コンテンツ管理用の開発プラットフォームとしての Oracle 9iFS
- 拡張のポイント

オブジェクトの管理

Oracle Internet File System（Oracle 9*i*FS）で実行される基本的なタスクは、オブジェクトを管理することです。オブジェクトには、次のように多数の形式があります。

- ドキュメントとフォルダ。
- ファイルのメタデータ： 名前、説明および変更日などです。
- ユーザー情報： 名前を持つユーザー、グループおよび認証設定などです。
- ACL: ファイルを表示および変更できるユーザーを制御します。
- スキーマ要素： オブジェクト・クラスなどです。
- システム要素： 他のオブジェクトの動作を制御する非表示の値です。

「管理」という用語には、ファイルの追加、変更、削除、バージョニングおよび編成などのアクティビティが含まれます。

多数のオブジェクト管理方法

インストール時の Oracle 9*i*FS には、オブジェクト操作用に多数のグラフィカル・インタフェースが用意されています。エンド・ユーザーは、基礎となるデータベース・スキーマの知識がなくても、Oracle 9*i*FS のコンテンツ管理、検索、バージョニングおよびセキュリティ機能を使用できます。ユーザーは他のファイル・サーバーの場合と同様に Oracle 9*i*FS に接続し、そこに格納されているファイルを他の共有ネットワーク記憶デバイスの場合と同様に操作します。

Java 実装

Oracle 9*i*FS は、JDBC のクラスを使用して Oracle9*i* データベースと相互に作用する Pure Java アプリケーションです。ファイルとフォルダは、Java クラス・オブジェクトの一時インスタンスを使用して表示され、データベースにレコードとして永続的に格納されます。ファイルの格納と取出しの複雑さは、Oracle 9*i*FS API により透過的、かつ効率的に処理されます。

多機能の XML フレームワーク

Oracle 9*i*FS では、新しい XML 標準の多数の機能がサポートされます。XML ファイルは、データの格納や転送に使用できるのみでなく、特定のデータ格納ニーズに合せて Oracle 9*i*FS を構成し、カスタマイズするためのメカニズムとしても使用できます。

ファイル・システムにもたらされるデータベース機能

Oracle 9*i*FS オブジェクトの Java 表現はプレゼンテーション層で操作できますが、基礎となるレコードは Oracle9*i* データベースにリレーショナル・レコードとして格納されます。これにより、標準的なファイル・システムにはない高機能と柔軟性が得られます。

セキュリティ

Oracle 9*i*FS のセキュリティ・モデルは、オブジェクトごとに構成可能なアクセス制御機能を持っています。ACL では、各ファイルに対して検出（フォルダ階層を検索または移動してファイルを探す機能）、読込み、書込みまたは削除の各アクセス権を持っているユーザーまたはグループを、ドキュメントごとに指定できます。

検索

Oracle 9*i*FS では、ユーザーは属性（名前、説明、変更日または開発者が定義したカスタム属性など）に基づいてドキュメントを検索できます。また、Oracle 9*i*FS では Oracle Text の索引付け機能が公開されます。これにより、ドキュメントの内容（マルチメディア・ファイルのテキスト要素など）の検索を、標準的なファイル・システムの全文コンテキスト検索機能の数千倍の速度で実行できます。

ファイル共有

Oracle 9*i*FS には、標準的なファイル・システムより優れたコンテンツ管理機能とコラボレーション機能があります。ファイルはバージョニングできます。つまり、ユーザーがドキュメントをチェックアウトし、変更し、チェックインするたびに、新バージョンが作成されます。また、ドキュメントの反復をさかのぼり、前の内容にアクセスできます。

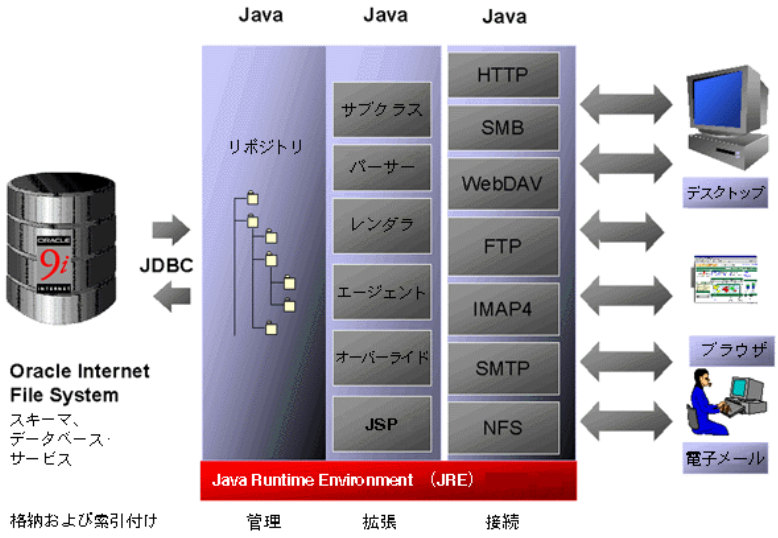
ファイルには多数のプロトコルを通じてアクセスでき、世界中のユーザーから使用できます。集中的な格納により、セキュリティやバックアップなど、データのメンテナンスに関連する管理タスクが簡素化されます。

同じファイルを複数のファイルとして使用できます。ユーザーは、使用中のファイルを個人用として最も適切なディレクトリ構造に置くことができます。Oracle 9*i*FS には、ファイルのコピーが 1 つのみ格納され、1 箇所でも更新されると、すべての場所で更新されます。あるディレクトリからファイルが削除されても、他のすべてのディレクトリには引き続き表示されます。ファイルが Oracle 9*i*FS リポジトリから削除されるのは、そのファイルがすべての親フォルダから削除された場合のみです。

アーキテクチャの概要

この項では、Oracle 9*i*FS のコンポーネントの概要を説明します（図 2-1 を参照）。

図 2-1 Oracle Internet File System のアーキテクチャ



プロトコル・サーバー

Oracle 9*i*FS のプロトコル・サーバーは、Oracle 9*i*FS の Java API を使用して記述されており、クライアントは HTTP、Windows Explorer、NFS、FTP、Macintosh DAVE および電子メール・クライアントを使用して、リポジトリ内のオブジェクトにアクセスできます。ファイルとフォルダは、各プロトコルに適した方法でエンド・ユーザーに表示されます。

リポジトリ

機能的には、リポジトリは、データベースの行や表の内容をファイル、フォルダ、ユーザーおよびグループなどのオブジェクトに変換するタスクを実行する単一メカニズムです。開発者の観点から見ると、リポジトリは実際には 2 つの部分で構成されています。一方はデータベースへのアクセスに使用される Java API で、他方は Oracle9*i* データベースのインスタンス内の IFSSYS スキーマです。

リポジトリの Java クラスでは、次のタスクが実行されます。

- オブジェクトの格納。各オブジェクトの内容とオブジェクトの属性が格納され、特定の型を持つオブジェクトに特別なルールが適用されます。
- アクセス制御。指定のオブジェクトが操作される前に、ユーザーが適切なアクセス権を持っているかどうかが確認されます。
- コンテンツ管理。バージョニング、チェックアウトおよびチェックインの機能があり、オブジェクトの索引付けを実行する必要があるかどうかが決定されます。
- インフラストラクチャの操作。非公開のクラスにより、ファイル・システムのアクションがデータベース操作に変換されます。データベース・セッションはリポジトリによりプールされ、必要に応じて割り当てられます。

Oracle 9*i*FS のスキーマ

スキーマは、特定のユーザーに属している表とフィールドのコレクションです。Oracle 9*i*FS のスキーマは `IFSSYS` で、Oracle 9*i*FS で使用されるすべてのオブジェクトが含まれています。ただし、ユーザー資格証明は `IFSSYS$CM` に格納されます。

Oracle 9*i*FS に対する開発の場合、データベースをこのレベルで直接取り扱う必要はありません。`IFSSYS` スキーマには、Oracle 9*i*FS で管理されるすべての永続オブジェクトが格納されます。

表

Oracle 9*i*FS のどの ClassObject にも、`ODM_<DatabaseObjectName>` という表があります。`<DatabaseObjectName>` は、ClassObject の DatabaseObjectName 属性の値です。ほとんどの場合、DatabaseObjectName の名前は ClassObject と同じですが、名前は 24 文字以内である必要があります。

列

各表の 1 列目は、その行の一意の ID 番号です。この値には NULL を指定できません。1 列に各属性が格納されます。属性は、データベースでサポートされるデータ型であれば、どのような型でもかまいません。データベースのデータ型から Java プログラミング言語と互換性のあるデータ型への、値の変換が必要になる場合があります。このため、属性値は AttributeValue オブジェクトを使用して Java からデータベースに渡されます。この種のオブジェクトは、どのようなデータ型でも受け入れるようにオーバーロードされ、データベースに対する挿入や抽出のための変換を実行します。

Oracle 9*i*FS では、どのようなデータ型でも同じ型の値の配列として格納する操作がサポートされます。このような場合、Oracle 9*i*FS では配列内の項目数がオブジェクト表に値として格納されます。配列値そのものは、その型の配列値を保持する別の表に格納されます。たとえば、カスタム配列属性 HOLIDAYS が Document クラスのサブクラスに追加されている場合、列 HOLIDAYS には特定レコードの日付値の数が格納されます。値そのものは、`ODM_AVDATEARRAY` 表に別個に格納され、一意のドキュメント ID により親レコードにリンクされます。

Oracle 9*i*FS に格納される永続オブジェクト

Oracle 9*i*FS には、3 つの型の永続オブジェクトが格納されます。他のオブジェクトは、それぞれこの 3 つの型のサブクラスです。表 2-1 に、Oracle 9*i*FS に格納される 3 つの型の永続オブジェクトを示します。

表 2-1 永続オブジェクトの型

データベース・オブジェクト	説明
ODM_PUBLICOBJECT	ユーザーが検出して操作できるすべて。
ODM_SYSTEMOBJECT	Format や Relationship など、PublicObject のメタデータ。
ODM_SCHEMAOBJECT	Attribute や ValueDomain など、PublicObject に関する情報。

オブジェクトの継承

Oracle 9*i*FS のすべてのオブジェクトは、3 つの主要な型のサブクラスです。たとえば、PublicObject の Document サブクラスは、PublicObject のすべての属性を継承する他に、2 つの独自属性が追加されます。Document クラスのサブクラスを作成すると、作成後のクラスは PublicObject および Document のすべての属性を継承します。このため、アプリケーションに必要な特定の属性と動作のみを追加して、洗練されたドキュメント格納モデルをすばやく構成できます。

どのプロトコル（HTTP や FTP など）から Oracle 9*i*FS にファイルが置かれる場合も、そのファイルの属性は内容とは別個に格納されます。ドキュメントの内容が Oracle Text を使用して索引付けされている場合、その内容は表 `ODMM_CONTENTSTORE` に格納されます。索引が付いていない場合は、`ODMM_NONINDEXEDSTORE` に格納されます。

ビュー

`ODM_` 表ごとに、多数のビューが作成されます。最も役立つのは `ODMV_<DatabaseObjectName>` で、親表から最も役立つ列の結合が表示されます。たとえば、ビュー `ODMV_PROPERTYBUNDLE` には、表 `ODM_PROPERTYBUNDLE`、`ODM_APPLICATIONOBJECT` および `ODM_PUBLICOBJECT` からの列が含まれます。

SQL*Plus を使用すると、IFSSYS スキーマを活用し、Java API 経由で行った変更結果を調べることができます。ただし、SQL コマンドを使用して Oracle 9*i*FS リポジトリで値を直接変更することは絶対にしないでください。この操作を行うとスキーマが破損し、どの程度のデータ量が失われるかは予測できません。

コンテンツ管理用の開発プラットフォームとしての Oracle 9*i*FS

Oracle 9*i*FS の Java API により、開発者は拡張ファイル・システムとコンテンツ管理機能を使用できるようになりました。開発プロジェクトの例を次に示します。

- ファイルとフォルダの新しいタイプの定義、既存のクラスのサブクラス化、新規の属性および動作の追加
- XML ベース・アプリケーションの作成
- Oracle 9*i*FS に挿入されたドキュメントから構造化データを抽出するカスタム・パーサーの作成
- Oracle 9*i*FS に格納されたファイルをオリジナル形式またはまったく異なるファイル・タイプとして再構成する、カスタム・レンダラの作成
- 特定フォルダへのファイルの格納などのイベントが発生した場合の、電子メール通知の生成

拡張のポイント

Oracle 9*i*FS の事実上あらゆる拡張機能が、カスタマイズ用に公開されています。次のエントリ・ポイントが使用可能です。

- サブクラス
- Tie クラス
- パーサー
- レンダラ
- エージェント
- オーバーライド
- サーブレットおよび JavaServer Pages

サブクラス

Oracle 9*i*FS の動作を変更する場合に最も簡単な方法は、既存のクラスを拡張してカスタム属性を追加することです。Document クラス、Folder クラスまたは独自のカスタム・サブクラスを拡張できます。作成したサブクラスは、PublicObject クラスまでの親の属性をすべて継承します。サブクラス化には、Java API、XML 構成ファイルまたは Oracle 9*i*FS Manager ツールを使用できます。

Tie クラス

Java API では、本来のクラスとその親の間に Tie クラスが存在します。Tie クラスは、親の動作を継承してサブクラスに渡す空のクラスです。たとえば、PublicObject クラスと Document クラスの間には、TieDocument というクラスがあります。TieDocument 自体には、メソッドはありません。

Tie クラスの目的は、本来のクラス自体を変更せずにその動作を変更できるようにすることです。たとえば、TieDocument では、Document クラスとそのすべてのサブクラスが継承する動作を追加できます。

パーサー

パーサーは、Oracle 9*i*FS でドキュメントが挿入または更新されるときにコールされるプログラムです。パーサーはファイルから構造化データを抽出し、ファイルの属性として格納します。予測可能な構造を持つドキュメントであれば、どのような種類のドキュメント用のカスタム・パーサーでも記述できます。属性は、名前 / 値のペアとして、または HTML タグや XML タグで識別できます。属性は、予測可能な位置（先頭、末尾またはファイルの先頭から *n* 文字目）に特定の値が常に挿入されるファイルから取り込むことができます。情報は識別されますが、パーサー・クラスは InputStream を仲介してデータを抽出し、開発者が選択したデータに対する操作を実行し、ファイル格納ジョブを完了できるように Oracle 9*i*FS リポジトリに制御を戻します。また、ParserCallback メカニズムもあります。このメカニズムは、開発者により識別されたプログラムをコールし、ファイルのコピーを特定のフォルダに格納するなどの後処理を実行します。

レンダラ

レンダラは、Oracle 9*i*FS リポジトリから Document オブジェクトを取り出し、開発者が選択した形式でクライアントに配信します。レンダラがパーサーと正反対の機能を持つというわけではありません。レンダラはパーサーとは逆のプロセスを実行し、リポジトリから項目を取り出してファイルとして再構成しますが、ファイルを元の形式で再格納する必要があるとはかぎりません。たとえば、Document のサブクラスに対して複数のレンダラを定義し、要求側クライアントのタイプに基づいて特定のレンダラをコールできます。

サーバー

サーバーは、プロトコル・サーバーおよびエージェントという 2 つのカテゴリに分類できます。プロトコル・サーバーは、Oracle 9*i*FS への接続に使用される様々なプロトコルについて接続を管理します。エージェントは、イベントにより起動するプログラムで、Oracle 9*i*FS リポジトリが変更されるか、事前定義の期間が経過すると起動します。Oracle 9*i*FS リポジトリ内でイベントが発生した場合に、特定のタスクまたはタスク・セットを実行するように、エージェントを定義できます。

オーバーライド

一部のクラスには、クライアント側ではなくサーバー側で実行される特殊なメソッドが用意されており、その目的は Oracle 9*i*FS のデフォルト動作をオーバーライドできる場所を提供することです。インストール時には、この種のメソッドは空です。この種のポイントには、Oracle 9*i*FS リポジトリに格納される情報の整合性の維持に必要なデフォルト動作をオーバーライドする危険を犯さずに、独自の処理を追加できます。

ほとんどのクラスには、メソッド `extendedPreInsert()`、`extendedPreUpdate()` および `extendedPreFree()` が含まれています。この 3 つのメソッドは、それぞれ Oracle 9*i*FS オブジェクトの作成、更新または削除の直前にコールされます。

`S_PublicObject` クラスには、オーバーライド `extendedPostInsert()`、`extendedPostUpdate()` および `extendedPostFree()` も含まれており、対応するアクションが発生した直後に実行される動作を追加できます。

`S_Folder` クラスには、独自の特殊なオーバーライド `extendedPreAddItem()`、`extendedPostAddItem()`、`extendedPreRemoveItem()`、`extendedPostRemoveItem()` があります。

サーブレットおよび JavaServer Pages

サーブレットは、HTTP サーバー内でモジュールとして実行され、動的コンテンツを生成する Java プログラムです。サーブレットは Sun Java Servlet API を使用して記述されますが、どのようなプラットフォームや Web サーバーでも実行できます。実質的には、サーブレットは、クライアント要求に応答して HTML コードを生成する Java プログラムです。サーブレットには、各要求の処理に必要なオーバーヘッドを最小限に抑える高パフォーマンス、シングル・プロセス、マルチスレッドのアーキテクチャが使用されます。

Oracle 9*i*FS では、HttpServlet クラスに基づくカスタム・サーブレットを使用して、リポジトリへの HTTP プロトコル・アクセスと、Distributed Authoring and Versioning（DAV）のある程度のサポートを提供します。このサーブレットは、Apache 駆動の Oracle HTTP Server で動作します。

JavaServer Pages（JSP）では、サーブレットの作成アプローチが異なります。JSP は、HTML コードが埋め込まれた Java プログラムではなく、Java コードが埋め込まれた HTML ページです。JSP では、ユーザー・インタフェースの作成タスクを、情報を提供するビジネス・ロジックの開発タスクから分離できます。ほとんどの場合、JSP はサーバー側に JavaBeans を持ち、ページのデータ要素用のアクセッサ・メソッドとミューテータ・メソッド（getter および setter）を提供します。サーブレットと JSP の詳細は、第 12 章「Web インタフェースのカスタマイズ」を参照してください。

3

Java API の概要

Oracle Internet File System には、広範囲なカスタマイズやアプリケーションに適用できるように設計された多数のメカニズムが用意されています。その 1 つが Java API であり、Oracle 9*i*FS で情報を操作するための強力なオブジェクト指向のインタフェースを提供します。この項の内容は、次のとおりです。

- Oracle 9iFS Java API の概要
- ファンクション別 API
- リポジトリへの接続
- 情報の管理
- コンテンツ・タイプの定義
- コンテンツ・タイプの動作の拡張
- コンテンツ・タイプのインスタンス化
- 情報の検索
- 情報の処理

Oracle 9*i*FS Java API の概要

Java API では Oracle 9*i*FS サーバーの機能が全面的に公開されるため、そのコンテンツ管理機能をアプリケーションで活用できます。また、Java で実装されるため、JavaServer Pages、J2EE および JavaBeans など、インターネット・アプリケーションの構築に適した標準ベースのテクノロジを使用できます。

ただし、Java API を初めて使用する場合、Javadoc は始点として適切でない場合があります。Oracle 9*i*FS Java API には、200 以上のクラスが含まれています。Document や Folder のようなクラスの他に、SystemObject クラス、SchemaObject クラス、Tie クラス、Server-side クラスおよび Definition クラスなど、なじみの薄いクラスも多数あります。

この章は、開発プラットフォームの 1 側面の概要を示すものと考えてください。この章を全体として読むと、API の概要として役立ちます。また、目標が決まった後では、クラスに関して特定情報を見つけるためのリファレンスとして役立ちます。

ドキュメントとユーザーを Oracle 9*i*FS でモデル化する方法と、このプラットフォームでの作業に使用できるすべてのツールについては、第 1 章「Oracle Internet File System SDK スタート・ガイド」を参照してください。

API パッケージ

Java API は、Java パッケージのセットとして編成されています。各パッケージを使用すると、API を簡単にブラウズして操作できます。パッケージでは、Java API のクラスがアプリケーションでの使用方法に従って編成されています。表 3-1 に、Java API のパッケージを示します。

表 3-1 Oracle 9*i*FS Java API のパッケージ

パッケージ名	注意	関連項目
`oracle.ifs.adk.filesystem`	このパッケージは、ファイルとフォルダの操作用に簡素化されたインタフェースを提供します。このパッケージ内の `IfsFileSystem` クラスでは上位レベルのファンクションが実行され、各ファンクションで他のパッケージに対して複数の下位レベル・コールを実行できます。 このパッケージでは、セキュリティと全文検索を除き、ある程度のバージョニングやファイル・ロックなど、単純なファイルおよびフォルダ操作が公開されます。	第 5 章「コンテンツ・タイプと属性の拡張」
`oracle.ifs.adk.mail`	このクラスでは、電子メール・メッセージ機能が実装されます。	第 18 章「プログラムによる電子メールの送受信」

表 3-1 Oracle 9*i*FS Java API のパッケージ（続く）

パッケージ名	注意	関連項目
`oracle.ifs.adk.security`	このパッケージには、セキュアな Web アクセスを管理するためのインタフェースが 1 つ含まれています。	第 15 章「セキュリティ」
`oracle.ifs.adk.user`	`UserManager` クラスでは、ユーザーの作成と管理が行われます。	第 15 章「セキュリティ」
`oracle.ifs.beans`	このパッケージには、Oracle 9*i*FS のコンテンツ管理機能を活用するカスタム・アプリケーションを構築できるように、プライマリ・インタフェースが用意されています。この種のクラスは、Bean 側 Java クラスとも呼ばれます。 このパッケージ内のクラスを使用してリポジトリ・セッションを確立し、Oracle 9*i*FS オブジェクトに対する検出、作成、変更および削除アクションを実行します。	第 8 章「検索アプリケーションの構築」、第 5 章「コンテンツ・タイプと属性の拡張」
`oracle.ifs.beans.parsers`	このパッケージには、カスタム・パーサーの作成時に必要なインタフェースとクラスが含まれています。	第 9 章「カスタム・パーサーの作成」、第 10 章「XML および Oracle Internet File System」
`oracle.ifs.beans.resources`	このパッケージのクラスでは、リポジトリ用のエラー・コードとローカライズ済みの文字列が処理されます。	付録 A「エラー・メッセージ」
`oracle.ifs.common`	このパッケージには、ユーザー・セッション、コレクション、イベント通知、キャッシュ、ローカライゼーションおよび Oracle Text のテーマへのアクセスの管理に必要なクラスが含まれています。	付録 A「エラー・メッセージ」
`oracle.ifs.management.domain`	このパッケージには、サーバーの構築に使用する 2 つのインタフェースと 1 つのクラスが含まれています。Servers（旧称は Agents）は、Oracle 9*i*FS に対してバックグラウンドで実行されるタスクを自動化するために使用されます。 このパッケージは、使用できないパッケージ `oracle.ifs.agents.common` のかわりに使用されます。	第 13 章「カスタム・サーバーの作成」
`oracle.ifs.search`	このパッケージ内のクラスは、`oracle.ifs.beans.Search` クラスとともに複合問合せの構成と実行に使用されます。	第 8 章「検索アプリケーションの構築」

表 3-1 Oracle 9*i*FS Java API のパッケージ（続く）

パッケージ名	注意	関連項目
`oracle.ifs.server`	このパッケージ内のクラスは、サーバー側 Java クラスとも呼ばれます。各クラスでは、Oracle 9*i*FS の動作が実装されます。`oracle.ifs.beans` パッケージは Oracle 9*i*FS の機能へのアクセスに使用されますが、これらのクラスにはその機能を実装するコードが含まれています。サーバー側 Java クラスをオーバーライドすると、挿入、更新および削除など、特定のタスクについて Oracle 9*i*FS サーバーでの実行方法を変更できます。	第 17 章「コンテンツ・タイプの動作のカスタマイズ」
`oracle.ifs.server.renderers`	このパッケージには、カスタム・レンダラの作成と、Oracle 9*i*FS の XML レンダリング機能の使用のための、3 つのレンダラ・クラスと 1 つのインタフェースが含まれています。	第 11 章「カスタム・レンダラの作成」、第 10 章「XML および Oracle Internet File System」
`oracle.ifs.server.sql`	これらのクラスでは、Oracle 9*i*FS スキーマ内でカスタム表に対して SQL 文を実行できます。	第 16 章「セッションおよびトランザクションの管理」

ファンクション別 API

カスタム・アプリケーションの設計および構築プロセスを開始すると、API のクラスの機能別ビューが必要になる場合があります。この項では、API をタスク順に説明します。

リポジトリへの接続

Oracle 9*i*FS では、`oracle.ifs.beans` パッケージのトップレベルに 2 つのクラス `LibraryService` および `LibrarySession` が用意されており、リポジトリへの接続用のプライマリ・インタフェースとして機能します。

この 2 つのクラスは、`ChallengeResponseCredential`、`CleartextCredential`、`HTTPDigestCredential`、`TokenCredential` および `ConnectOptions` など、Oracle 9*i*FS への接続に必要なユーザー名、パスワードおよび他の情報の構成に使用される `oracle.ifs.common` パッケージ内の複数のクラスでサポートされます。

情報の管理

Java API のクラスのほとんどは、リポジトリに格納される各種の情報の管理に使用されます。これらのクラスは、`oracle.ifs.beans` パッケージのトップレベル・クラス `LibraryObject` からの派生クラスです。このクラスは、リポジトリ内の基本情報単位を表します。`LibrarySession` クラスは、これらのクラスのインスタンスを作成するファクトリとして機能します。

コンテンツ・タイプの定義

Oracle 9*i*FS には柔軟な情報モデルが用意されており、特別なコンテンツ・タイプを定義し、型ごとにルールと動作を作成できます。カスタム・コンテンツ・タイプを定義することは、すべてのカスタマイズの基礎のため、複数のテクニックが用意されています。Oracle Enterprise Manager や、XML ファイル、Java API を使用できます。API には、`oracle.ifs.beans` パッケージの `SchemaObject` クラスから派生したクラス・セットが含まれており、コンテンツ・タイプの構造、そのタイプの情報の格納に使用する表、コンテンツ・タイプの動作を実装する Java クラスを定義できます。

コンテンツ・タイプの動作の拡張

Oracle 9*i*FS では、コンテンツ・タイプの動作を実装する Java クラスを拡張し、その動作をカスタマイズできます。Oracle 9*i*FS Java API には `Tie` クラスが含まれており、このクラスにはクラス階層のどこででも Java クラスを拡張する整然とした方法が用意されています。ネーミング規則 `Tie<classname>` により、各コンテンツ・タイプの Java クラス名が保たれ、カスタマイズ内容を簡単に実装でき、他のユーザーは実装後に簡単に理解できます。

コンテンツ・タイプのインスタンス化

各コンテンツ・タイプには対応する `Definition` クラスがあり、このクラスは `oracle.ifs.beans` パッケージ内の `LibraryObjectDefinition` クラスの派生クラスです。`Definition` クラスは、新規コンテンツ・タイプ・インスタンスの作成に使用されます。たとえば、`Document` のインスタンスを作成するには、`DocumentDefinition` オブジェクトを作成し、それを `LibrarySession` のメソッドに渡してドキュメントを作成します。作成したドキュメントは、永続オブジェクトとして Oracle 9*i*FS リポジトリに格納されます。ネーミング規則 `<classname>Definition` により、対応するコンテンツ・タイプの Java クラス名が保たれます。

情報の検索

Oracle 9*i*FS では、`oracle.ifs.beans` パッケージのトップレベルに、リポジトリを検索して検索結果を操作するための 4 つのプライマリ・クラス `Selector`、`Search`、`SearchResultObject` および `FolderPathResolver` が用意されています。

- `Selector` クラスは、リポジトリ内の情報に関する単純問合せの実行用です。
- `Search` クラスには、`oracle.ifs.search` パッケージ内の添付クラスで構成される複合問合せを実行するための確実なインタフェースが用意されています。
- `SearchResultObject` クラスでは、`Search` から返された情報が操作されます。
- `FolderPathResolver` クラスは、ファイル・システム・パス経由でリポジトリ内で情報を検索するために使用されます。

情報の処理

サーバー側の情報は、3 つのクラス・セットで処理されます。`oracle.ifs.beans.parsers` パッケージには、リポジトリに格納される前に情報を自動的に解析するためのクラスが含まれています。`oracle.ifs.server.renderers` パッケージには、リポジトリから取り出される情報を自動的にレンダリングするためのクラスが含まれています。`oracle.ifs.management.domain` パッケージには、期限切れになった情報の削除など、Oracle 9*i*FS サーバーに対する操作の実行タスクを自動化するサーバーを構築するためのクラスが含まれています。

リポジトリへの接続

Oracle 9*i*FS には、リポジトリへのアプリケーション接続を管理するためのクラス・セットが用意されています。リポジトリへの接続に使用されるプライマリ・クラスは、`LibraryService` および `LibrarySession` の 2 つです。この 2 つのクラスは、`oracle.ifs.beans` パッケージのトップレベルにあります。どちらも、接続の認証に必要な情報の保持に使用される `oracle.ifs.common` パッケージ内の他の少数のクラスによりサポートされます。

LibraryService

`LibraryService` クラスは、Oracle 9*i*FS データベース・スキーマへの接続の確立に使用されます。LibraryService では、Oracle 9*i*FS 用の複数のクライアント・セッションが管理されます。また、クライアント・セッションの開始と停止、休止セッションのタイムアウト、セッションのトレース、イベントの管理およびセッション間での共有情報のキャッシュに使用されます。LibraryService は、Oracle 9*i*FS サーバー上の Oracle ソフトウェア・ホーム・ディレクトリにあるサービス・プロパティ・ファイルを使用して構成されます。複数の LibraryService を実行し、各種のクライアント・セッションを管理するように個別に構成できます。

サービス・プロパティ・ファイルを使用してサービスを構成する方法の詳細は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

表 3-2 に、LibraryService で最も使用頻度の高いメソッドを示します。

表 3-2 LibraryService のメソッド

findService() startService()	既存または新規の LibraryService を名前で取得します。
connect()	Oracle 9*i*FS への接続を確立します。ユーザーの資格証明と他の接続情報を保持する ConnectOptions のインスタンスを受け取ります。
getTokenCredential() invalidateTokenCredential()	以前に作成されたトークン資格証明を操作します。

表 3-2 LibraryService のメソッド（続く）

dispose()	LibraryService を処分します。

LibrarySession

LibrarySession クラスは、Oracle 9*i*FS の各クライアント・セッションの表示に使用されます。ユーザーがクライアント経由で Oracle 9*i*FS にログインするたびに、LibrarySession が作成され、ログイン中のユーザーと Oracle 9*i*FS との対話方法が管理されます。ユーザーがログインしている間は 'session' と呼ばれます。LibrarySession では、ユーザーのセッションの状態が追跡されます。また、このクラスは Oracle 9*i*FS で行われるすべての操作のプライマリ・ファクトリです。たとえば、LibrarySession は、コンテンツ・タイプのインスタンスの作成、削除および更新に使用されます。また、Search クラスの構成と実行にも使用されます。

表 3-3 に、LibrarySession で最も使用頻度の高いメソッドを示します。

表 3-3 LibrarySession のメソッド

abortTransaction() beginTransaction() completeTransaction() disconnect()	トランザクションの状態を管理します。
createPublicObject() createSchemaObject() createSystemObject() createView()	新規の永続オブジェクトを作成します。

表 3-3 LibrarySession のメソッド（続く）

getClassAccessControlListCollection() getClassDomainCollection() getClassObjectCollection() getDirectoryUserCollection() getExtendedPermissionCollection() getExtendedUserProfileCollection() getFormatCollection() getFormatExtensionCollection() getPermissionBundleCollection() getPolicyCollection() getSharedAccessControlListCollection() getSystemAccessControlListCollection() getValueDefaultCollection() getValueDomainCollection()	オブジェクトの特定クラスの Collection を取得します。
getDefaultFolderRelationshipClassname() getDefaultFolderSortSpecification() getFolderPathDelimiter() getRootFolder() getWorldDirectoryGroup() setDefaultFolderRelationshipClassname() setDefaultFolderSortSpecification() setFolderPathDelimiter()	システムのデフォルトを決定し、操作します。
getDirectoryObject() getPublicObject() getSchemaObject() getSystemObject()	オブジェクトを Identifier 別に検索します。

表 3-3 LibrarySession のメソッド（続く）

getId() getSchemaVersionNumber() getSchemaVersionString() getServerName() getServerProperty() getServerVersionNumber() getServerVersionString() getServiceId() getVersionNumber() getVersionString()	サーバー構成およびバージョン情報を決定します。
grantAdministration() isAdministrationMode() setAdministrationMode()	セッションの管理モードを取得および設定します。
getObjectsLockedForSession() unlockForSession() getUser() impersonateUser() isConnected()	接続ユーザーに関する情報を取得します。
getLocalizer()	ローカライゼーション情報を取得します。
deregisterClassEventHandler() deregisterEventHandler() invokeServerMethod() registerClassEventHandler() registerEventHandler()	イベントを管理します。

共通クラス

LibraryService による接続の確立時には、接続先となる Oracle 9*i*FS サーバーなどの接続オプション・セットとユーザーの資格証明を渡す必要があります。`oracle.ifs.common` パッケージの `ConnectOptions` クラスには、Locale、Service name および Service password など、`LibraryService` に渡される接続情報が保持されます。クラス `ChallengeResponseCredential`、`CleartextCredential`、`HttpDigestCredential` および `TokenCredential` には、Oracle 9*i*FS へのユーザー接続の認証に使用される各種のユーザー資格証明が保持されます。

表 3-4 に、`ConnectOptions` で最も使用頻度の高いメソッドを示します。

表 3-4 ConnectOptions のメソッド

getLocale() setLocale()	作成されるセッションのロケールを決定します。
getServiceName() setServiceName()	Oracle 9*i*FS への接続の確立に使用する LibraryService を決定します。
getServicePassword() setServicePassword()	LibraryService で Oracle 9*i*FS データベース・スキーマに接続するために必要なパスワードを決定します（ifssys データベース・ユーザー・パスワード）。

表 3-5 に、`ClearTextCredential` で最も使用頻度の高いメソッドを示します。

表 3-5 ClearTextCredential のメソッド

getName() setName()	ユーザー名を決定します。
getPassword() setPassword()	ユーザーのパスワードを決定します。パスワードは暗号化されずに渡されます。

表 3-6 に、`ChallengeResponseCredential` で最も使用頻度の高いメソッドを示します。

表 3-6 ChallengeResponseCredential のメソッド

getName() setName()	ユーザー名を決定します。
getChallenge() setChallenge()	要求を決定します。
getResponse() setResponse()	応答を決定します。

表 3-6 ChallengeResponseCredential のメソッド（続く）

TYPE_LMSESSIONKEY TYPE_NTSESSIONKEY getType() setType()	応答のタイプを決定します。

表 3-7 に、`HTTPDigestCredential` で最も使用頻度の高いメソッドを示します。

表 3-7 HTTPDigestCredential のメソッド

getName() setName()	ユーザー名を決定します。
getChallenge() setChallenge()	要求を決定します。
getResponse() setResponse()	応答を決定します。
HTTPDIGESTCREDENTIAL_RFC2069 HTTPDIGESTCREDENTIAL_RFC2617 getType() setType()	HTTPDigestCredential のタイプを決定します。
getMethod() setMethod()	メソッドを決定します。
getCnonce() setCnonce()	nonce を決定します。
getNc() setNc()	nc を決定します。
getRealm() setRealm()	レルムを決定します。
getQop() setQop()	qop（quality of protection）を決定します。
getUri() setUri()	uri を決定します。

表 3-8 に、TokenCredential で最も使用頻度の高いメソッドを示します。

表 3-8 TokenCredential のメソッド

getName() setName()	ユーザー名を決定します。
getToken() setToken() getTokenLength() setTokenLength()	ユーザーの認証に使用されるトークンを決定します。
getTimeoutPeriod() setTimeoutPeriod() getAllowedAuthenticationCount() setAllowedAuthenticationCount()	ユーザーが認証を試行できるタイムアウト期間と回数を決定します。
getParameters() setParameters()	追加のパラメータを提供します。

情報の管理

Oracle 9*i*FS では、事実上あらゆるタイプの情報を管理できます。ドキュメント、フォルダ、ユーザーおよびグループを管理するように事前に構成されていますが、医療記録、保険料請求および設計仕様など、ビジネスに特有のカスタム情報タイプを追加できます。

Oracle 9*i*FS では、各種の情報は、定義済みの属性と動作のセットを持つコンテンツ・タイプとして定義されます。属性は、Name、Owner、Size および Format など、情報のタイプの記述に使用されます。動作では、フォルダ内のアイテムのリスト表示やドキュメントの内容のフェッチなど、情報に対して実行できる操作が定義されます。

各コンテンツ・タイプは、次の 2 つの部分で構成されます。

- そのタイプの情報のデータが格納される表のセット
- 情報の動作を実装する Java クラスのセット

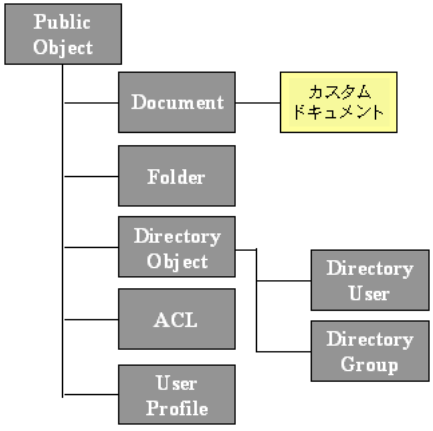
たとえば、Document コンテンツ・タイプは、Oracle 9*i*FS で管理されるドキュメントのデータを格納する表のセットで構成されます。これらの表では、ドキュメントのファイルの内容は LOB 列に格納され、Name、Description、Format および Size などの属性はスカラー列に格納されます。Java クラスのセットである `Document.class` および `S_Document.class` は、ドキュメントの内容の更新やサマリー作成など、Oracle 9*i*FS でのドキュメントの動作の実装に使用されます。この 2 つの動作は、それぞれ Java クラスのメソッド `setContent()` および `getSummary()` により実装されます。

Oracle 9*i*FS リポジトリ内では、情報はコンテンツ・タイプの表にデータ行として格納されます。RDBMS の機能を活用し、情報の格納と取出しを最適化するために、データをデータベース内の複数の表間で正規化できます。Oracle 9*i*FS を使用してリポジトリ内の情報を操作する場合、情報は API によりコンテンツ・タイプの Java クラスのインスタンスとして表示されます。情報を Java クラスのインスタンスとして表示すると、アプリケーションではオブジェクト指向の方法で情報を管理できます。たとえば、ドキュメントの属性やファイルの内容がデータベース内で複数の表にまたがってどのように格納されているかを知っている必要はありません。かわりに、ドキュメント全体を単一の API コールでフェッチし、1 つのオブジェクトとして操作できます。

コンテンツ・タイプ階層

図 3-1「コンテンツ・タイプ階層」のように、Oracle 9*i*FS ではコンテンツ・タイプが階層形式で編成されるため、類似するコンテンツ・タイプを簡単に定義できます。コンテンツ・タイプは、階層内で上位にあるコンテンツ・タイプから属性と動作を継承します。たとえば、Document および Folder は、Owner、Description、LockState のような特定の属性と、setOwner、getDescription、lock のような動作を共有します。共有の属性と動作を簡単に定義できるように、Document と Folder は PublicObject から派生します。PublicObject では、ユーザーが操作するすべての情報に共通する属性と動作が定義されます。PublicObject から派生するすべてのコンテンツ・タイプは、その属性と動作を継承します。該当する場合は、各コンテンツ・タイプで継承する特定の属性と動作をオーバーライドできます。Oracle 9*i*FS サーバーは Java で実装されるため、コンテンツ・タイプ階層は Java 言語用に定義された継承ルールに従います。

図 3-1 コンテンツ・タイプ階層



Oracle 9*i*FS のコンテンツ・タイプ階層は、全面的に拡張可能です。この階層を拡張し、ビジネス固有の情報タイプの管理に使用するカスタム・コンテンツ・タイプを組み込むことがで

きます。カスタム・コンテンツ・タイプを作成する場合は、まず情報の属性と動作を定義します。次に、カスタム・コンテンツ・タイプを拡張するコンテンツ・タイプを選択します。別のコンテンツ・タイプを拡張すると、カスタム・コンテンツ・タイプは Oracle 9*i*FS にすでに組み込まれている一部の属性と動作を継承できます。たとえば、書籍を管理するカスタム・コンテンツ・タイプを作成するには、Document を拡張し、ファイルの内容を管理するための属性と動作を継承させることができます。その後で、さらに属性とメソッドを追加し、継承したメソッドをオーバーライドして、Oracle 9*i*FS による書籍の管理方法をカスタマイズできます。

Oracle 9*i*FS Java API には、コンテンツ・タイプの定義、インスタンス化および管理に使用するクラスが含まれています。API クラス階層は、コンテンツ・タイプ階層とパラレルになっています。

クラス階層の最上位は `LibraryObject` です。このクラスでは、Oracle 9*i*FS で管理されるすべての情報に共通の属性と動作が定義されます。たとえば、Oracle 9*i*FS で管理されるすべての情報は、オブジェクトを一意に識別する長い番号である属性 ID と、オブジェクトの記述ラベルを提供する文字列を持ちます。LibraryObject では、Java メソッド `free()` を使用してリポジトリから情報を削除するなど、共通の動作も定義されます。このクラス階層で、LibraryObject の下位には次の 2 つの分岐があります。

- **PublicObject クラス**では、Document や Folder など、アプリケーションのエンド・ユーザーが操作できる、リポジトリ内の永続情報が管理されます。PublicObject には AccessControlList があり、特定のユーザーおよびグループによる情報の操作方法が決定されます。通常、PublicObject は、適切なアクセス権限を持つユーザーがカテゴリに分類し、バージョニングし、フォルダに格納し、相互に関連付けることができます。Oracle 9*i*FS では、オブジェクトの作成日時、作成者および最終変更時刻が追跡されます。
- **SystemObject** クラスは、PublicObject を管理するためのサポート用オブジェクトであり、アプリケーションのエンド・ユーザーは直接操作できません。たとえば、Format は SystemObject の派生クラスで、各種ファイル形式に関する情報の管理に使用されます。SystemObject を直接操作できるのは、管理権限を持っているユーザーのみです。

PublicObject クラスと SystemObject クラスは、次のようにその機能に従ってグループ化できます。

- 各種の情報の管理
- 特別なメタデータと動作の管理
- 情報のバージョニング
- 情報へのアクセスの制御
- ディレクトリの管理

各種の情報の管理

Oracle 9*i*FS Java API には、次の 3 つの基本的なコンテンツ・タイプを管理するための PublicObject および SystemObject クラスのセットが含まれています。

- **Document:** 内容が構造化されていない情報用です。
- **Folder:** コンテナ性特質を持つオブジェクト用です。
- **ApplicationObject:** 構造化データを構成する情報用です。

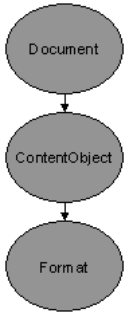
カスタム・コンテンツ・タイプを作成する場合、通常はコンテンツ・タイプの物理的な構造と動作に従って、この 3 つのタイプのうち 1 つを拡張するように選択します。カスタム・コンテンツ・タイプに、構造化されていない内容を管理するためのメソッドが必要な場合は、Document を拡張します。また、アイテムの追加、アイテムの削除および内容のリスト表示など、コンテナシップ動作が必要な場合は、Folder を拡張します。カスタム・コンテンツ・タイプに構造化されていない内容がなく、コンテナ性特質もないが、PublicObject として処理する必要がある場合は、ApplicationObject を拡張するように選択できます。

Document

Document は、構造化されていない内容を持つ情報です。図 3-2「ドキュメント・オブジェクト・モデル」に、Document の定義に使用する 3 つのクラスを示します。

- **Document:** アプリケーションのエンド・ユーザーがドキュメントの操作に使用する、プライマリ・パブリック・インタフェースです。
- **ContentObject:** ドキュメントの内容が格納され、操作されます。
- **Format:** MS Word、PDF、テキストなど、各種の内容形式が登録されます。

図 3-2 ドキュメント・オブジェクト・モデル



Document クラスは、Name、Description、Owner、Creator、CreateDate、LastModifier、LastModifyDate など、Oracle 9*i*FS で管理される情報の各部の基本属性を持ちます。また、

ドキュメントの構造化されていない内容を操作するための属性 ContentObject も持っています。ドキュメントのうち構造化されていない内容は、実際には ContentObject 属性に格納されません。かわりに、Document クラスの ContentObject 属性は、ドキュメントの内容の保持と管理に使用される ContentObject クラスのインスタンスを参照します。

ContentObject クラスは、Format、Language および Media など、構造化されていない内容に固有の属性を持ちます。また、`getContentStream()` および `getFilteredContent()` など、内容を操作するためのメソッドも持っています。

ドキュメントの内容を別個のオブジェクトとして正規化すると、ドキュメントの内容をさらに柔軟に操作できます。

- 第 1 に、様々なコンテンツ・タイプのドキュメントを管理するアプリケーションを構築できます。たとえば、ドキュメントの一部の内容は、従来どおり用紙を使用している場合があります。ContentObject を拡張すると、キャビネットやファイルの索引番号など、用紙ファイルの位置を追跡する属性を持たせることができます。内容を別個のオブジェクトとして操作すると、内容の物理的な性質や位置に関係なく、すべてのドキュメントを Document のインスタンスとして管理できます。
- 第 2 に、ライフ・サイクル中に内容が物理的に変化するドキュメントを管理できます。たとえば、用紙ファイルの場合、後でファイルをスキャンして電子的な形式で取り込み、電子コピーを Oracle 9*i*FS に格納する必要が生じることがあります。ドキュメントの内容は別個のオブジェクトとして管理されるため、電子コピーを ContentObject のインスタンスとして簡単にインポートし、それを参照するように Document の ContentObject 属性を更新できます。内容は、電子ファイルに適した新しい属性と動作を持つことになりますが、ID、Name および Description など、Document の属性は影響を受けません。

ドキュメントの内容の形式は、もう 1 つのクラスである Format でも管理されます。Format は、MimeType や Extension のように内容形式に関する情報を管理するための属性と、形式が Binary か ASCII かを決定する `isBinary()` のように、形式を操作するメソッドを持っています。Format のインスタンスは、認識される各内容形式を表すように作成されます。Oracle 9*i*FS のインストール時には、複数の Format インスタンスが自動的に作成されます。ContentObject クラスの Format 属性は、ドキュメントの内容の形式を表す Format インスタンスを参照します。

Document は、PublicObject の派生クラスです。したがって、アプリケーションのエンド・ユーザーが操作できる属性と動作を持っています。ContentObject は、SystemObject の派生クラスです。管理権限を持たないエンド・ユーザーは、このクラスを直接操作できません。Document には、エンド・ユーザーがドキュメントの ContentObject を操作できるようにインタフェースが用意されています。Format も、SystemObject の派生クラスです。Format のインスタンスを作成したり変更できるのは、管理権限を持っているユーザーのみです。

表 3-9 に、Document で最も使用頻度の高い属性とメソッドを示します。

表 3-9 Document の属性およびメソッド

Name getName() setName()	ドキュメント名を決定します。
Description getDescription() setDescription()	ドキュメントの説明を提供します。
Owner getOwner() setOwner()	ドキュメントを所有するユーザーを決定します。ドキュメントの所有者には、そのドキュメントへのすべてのアクセス権が暗黙的に付与されます。Owner 属性は DirectoryUser を参照します。
ACL getAcl() setAcl() getDefaultAccessLevel() getEffectiveAccessLevel() grantAccess() revokeAccess() revokeAllAccess()	ドキュメントの AccessControlList を決定します。ACL 属性は AccessControlList を参照します。
Creator	ドキュメントを最初に作成したユーザーを決定します。Creator 属性は DirectoryUser を参照します。Creator 属性は Oracle 9*i*FS システムにより設定されます。
CreateDate	ドキュメントの作成日時が格納されます。この属性は Oracle 9*i*FS システムにより設定されます。
LastModifier	ドキュメントを最後に変更したユーザーを決定します。LastModifier 属性は DirectoryUser を参照します。LastModifier 属性は Oracle 9*i*FS システムにより設定されます。
LastModifyDate	ドキュメントの最終更新日時が格納されます。この属性は Oracle 9*i*FS システムにより設定されます。
ExpirationDate getExpirationDate() setExpirationDate()	ドキュメントの有効期限を決定します。アプリケーション開発者は、この日付に達した時点でドキュメントを期限切れにする方法を管理するエージェントを作成できます。

表 3-9 Document の属性およびメソッド（続く）

ContentObject generateFileName() generateThemes() getCharacterSet() getContentReader() getContentObject() getContentSize() getContentStream() getFilteredContent() getFormat() getLanguage() getSummary() getThemes() setCharacterSet() setContent() setFormat() setLanguage()	ドキュメントの構造化されていない内容を操作します。ContentObject 属性は、ドキュメントの内容を表す ContentObject を参照します。getContentObject() メソッドを使用すると、ContentObject のインスタンスを取得できます。 ContentObject 属性には、ContentObject インスタンスの一意識別子が値として保持されます。getContentObject() メソッドは、この識別子を使用して、オブジェクトとして操作できる ContentObject インスタンスを取得します。setContent() メソッドは新規の ContentObject インスタンスを作成し、新規インスタンスの識別子を使用して ContentObject 属性の値を設定します。 **注意：** バージョニングされていないドキュメントが更新されるたびに、setContent() メソッドは新規の ContentObject を作成します。ドキュメントをバージョニングする方法は、第 14 章「バージョニングの実装」を参照してください。 Document には、getFormat()、getContentSize() および getContentReader() など、Document インスタンスから ContentObject インスタンスを直接操作するための便利なメソッドが用意されています。

表 3-10 に、ContentObject で最も使用頻度の高い属性とメソッドを示します。

表 3-10 ContentObject の属性およびメソッド

CharacterSet getCharacterSet() setCharacterSet()	ドキュメントの内容のキャラクタ・セットを決定します。
ContentSize getContentSize()	ドキュメントの内容のサイズを決定します。
Format getFormat()	内容形式を表す Format インスタンスを決定します。
Language getLanguage()	ドキュメントの内容の言語を決定します。

表 3-10 ContentObject の属性およびメソッド（続く）

Media	ドキュメントの内容が格納されているメディアを参照します。現行のリリースの Oracle Internet File System には、2 つの記憶メディア BlobMedia および IndexedBlobMedia が用意されています。
ReadOnly isReadOnly() setReadOnly()	ドキュメントの内容が読取り専用かどうかを決定します。
Content filterContent() generateSummary() generateThemes() getContentReader() getContentStream() getFilteredContent() getSummary() getThemes()	ドキュメントの内容を操作します。 Content 属性は、記憶メディア内で内容を一意に識別するために使用される内容の ID を参照します。また、Content および Media 属性は、データベース内で内容の格納場所の検索に使用されます。 getContentReader() および getContentStream() メソッドは、Reader または Stream でデータベースからドキュメントの内容を取り出すために使用されます。 filterContent() メソッドでは、Oracle Text を使用して内容が ASCII または HTML になるようにフィルタリングされます。generateSummary() および generateThemes() メソッドでは、Oracle Text の言語分析機能を使用して、ドキュメントの内容のテーマ・プロファイルまたはサマリーが生成されます。これらのタスクの実行後は、getFilteredContent()、getSummary() および getThemes() メソッドを使用して出力を取得できます。

表 3-11 に、Format で最も使用頻度の高い属性とメソッドを示します。

表 3-11 Format の属性およびメソッド

Binary isBinary() setBinary()	形式が Binary であるか ASCII であるかを決定します。
Extension getExtension() setExtension()	この形式に対応付けられているファイル拡張子を決定します。
MimeType getMimeType() setMimeType()	形式の MIME タイプを決定します。

表 3-11 Format の属性およびメソッド（続く）

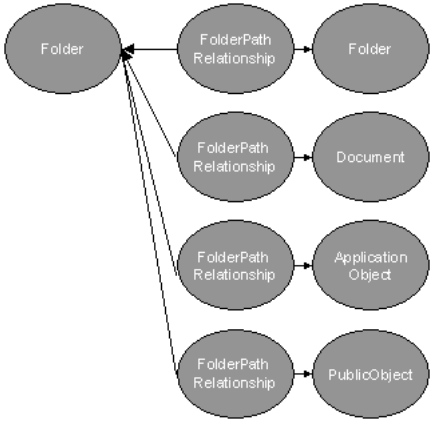
Name getName() setName()	Format オブジェクトの記述的な Name またはラベルを提供します。

Folder

Folder は、他のオブジェクトを含むオブジェクトです。Folder は Oracle 9*i*FS クライアント内で従来のファイル・システムのフォルダと同様に動作し、リポジトリの編成に使用されます。図 3-3「フォルダ・オブジェクト・モデル」のように、Folder は主として次の 2 つのコンポーネントで管理されます。

- **Folder:** 実際のフォルダを表します。
- **FolderPathRelationship:** フォルダとそこに含まれるアイテム間のリンクを表します。

図 3-3 フォルダ・オブジェクト・モデル



Folder は PublicObject クラスを拡張するため、Name、Description、Owner、Creator、CreateDate、LastModifier および LastModifyDate などの基本属性を継承します。また、`addItem()`、`removeItem()`、`getItemCount()` および `getItems()` など、フォルダの内容を管理するためのメソッドを持っています。

フォルダとそこに含まれるアイテムの関係は、FolderPathRelationship で表されます。FolderPathRelationship は、2 つの属性を使用して関係を表します。その一方はフォルダを参照する LeftObject で、他方はフォルダに含まれるオブジェクトを参照する RightObject です。FolderPathRelationship の LeftObject および RightObject 属性は、Document、Folder および ApplicationObject など、PublicObject を拡張するすべてのクラスのインスタンスを参照できます。また、FolderPathRelationship には、パス指向の関係を簡単に操作できるよう

にする属性と動作も含まれています。たとえば、FolderPathRelationship には、フォルダ内のアイテムが一意の名前を持つことを保証し、不正なファイル名やフォルダ名を禁止し、フォルダの相対パスに基づいてアイテムを簡単に検索できるようにするためのロジックが組み込まれています。

Folder は PublicObject を拡張するため、非管理ユーザーがフォルダを操作するためのプライマリ・パブリック・インタフェースです。FolderPathRelationship は SystemObject の派生クラスであり、非管理ユーザーは直接操作できません。Folder には、FolderRelationship のインスタンスを作成、削除および取得できるように、`addItem()`、`removeItem()` および `getItems()` などのメソッドが用意されています。

表 3-12 に、Folder で最も使用頻度の高い属性とメソッドを示します。

表 3-12 Folder の属性およびメソッド

Name getName() setName()	フォルダ名を決定します。
Description getDescription() setDescription()	フォルダの説明を提供します。
Owner getOwner() setOwner()	フォルダを所有するユーザーを決定します。フォルダの所有者には、そのフォルダへのすべてのアクセス権が暗黙的に付与されます。Owner 属性は DirectoryUser を参照します。
ACL getAcl() setAcl() getDefaultAccessLevel() getEffectiveAccessLevel() grantAccess() revokeAccess() revokeAllAccess()	フォルダの AccessControlList を決定します。 ACL 属性は AccessControlList を参照します。
Creator	フォルダを最初に作成したユーザーを決定します。 Creator 属性は DirectoryUser を参照します。 Creator 属性は Oracle 9*i*FS システムにより設定されます。
CreateDate	フォルダの作成日時が格納されます。この属性は Oracle 9*i*FS システムにより設定されます。

表 3-12 Folder の属性およびメソッド（続く）

LastModifier	フォルダを最後に変更したユーザーを決定します。LastModifier 属性は DirectoryUser を参照します。LastModifier 属性は Oracle 9*i*FS システムにより設定されます。
LastModifyDate	フォルダの最終更新日時が格納されます。この属性は Oracle 9*i*FS システムにより設定されます。
ExpirationDate getExpirationDate() setExpirationDate()	フォルダの有効期限を決定します。アプリケーション開発者は、この日付に達した時点でフォルダを期限切れにする方法を管理するエージェントを作成できます。
addItem() removeItem() removeItems() getItems() getItemCount() hasSubfolders() getSubfolderCount() moveItems()	フォルダに含まれるアイテムをカウント、追加、削除およびリスト表示します。
getSortSpecification() setSortSpecification()	アイテムをリスト表示するときのソート方法を決定します。アイテムのソート条件には、PublicObject のどの属性でも使用できます。
checkExistenceOfPublicObjectByPath() findPublicObjectByPath()	フォルダ・パスに基づいてアイテムを検索します。

表 3-13 に、FolderRelationship で最も使用頻度の高い属性とメソッドを示します。

表 3-13 Folder Relationship の属性およびメソッド

LeftObject getLeftObject()	関係の左辺として表されるフォルダを決定します。
RightObject getRightObject()	関係の右辺として表されるフォルダに含まれているアイテムを決定します。

ApplicationObject

ApplicationObject は、PublicObject の特質を持つ必要はあるが Document または Folder の特質は持たない拡張コンテンツ・タイプを定義するための始点として機能します。ApplicationObject は、PublicObject のすべての属性とメソッドを継承しますが、特殊な用途を持つ拡張属性やメソッドは含まれていません。

たとえば、顧客関連を管理するアプリケーションを実装する場合は、Primary Contact、Telephone Number および Address など、顧客に関する属性を持つ Customer コンテンツ・タイプを作成できます。顧客は構造化されていない内容を持たないため、Document を拡張することはありません。また、顧客はコンテナシップ動作を持たないため、Folder を拡張することもありません。したがって、ApplicationObject はこのコンテンツ・タイプの基礎となります。ApplicationObject を拡張することで、Customer コンテンツ・タイプは PublicObject の動作と属性を継承するため、エンド・ユーザーが顧客情報にアクセスして操作する方法を制御できます。

表 3-14 に、ApplicationObject で最も使用頻度の高い属性とメソッドを示します。

表 3-14 ApplicationObject の属性およびメソッド

Name getName() setName()	ApplicationObject の名前を決定します。
Description getDescription() setDescription()	ApplicationObject の説明を提供します。
Owner getOwner() setOwner()	ApplicationObject を所有するユーザーを決定します。ApplicationObject の所有者には、その ApplicationObject へのすべてのアクセス権が暗黙的に付与されます。Owner 属性は DirectoryUser を参照します。
ACL getAcl() setAcl() getDefaultAccessLevel() getEffectiveAccessLevel() grantAccess() revokeAccess() revokeAllAccess()	ApplicationObject の AccessControlList を決定します。ACL 属性は AccessControlList を参照します。

表 3-14 ApplicationObject の属性およびメソッド（続く）

Creator	ApplicationObject を最初に作成したユーザーを決定します。Creator 属性は DirectoryUser を参照します。Creator 属性は Oracle 9*i*FS システムにより設定されます。
CreateDate	ApplicationObject の作成日時が格納されます。この属性は Oracle 9*i*FS システムにより設定されます。
LastModifier	ApplicationObject を最後に変更したユーザーを決定するために使用されます。LastModifier 属性は DirectoryUser を参照します。LastModifier 属性は Oracle 9*i*FS システムにより設定されます。
LastModifyDate	ApplicationObject の最終更新日時が格納されます。この属性は Oracle 9*i*FS システムにより設定されます。
ExpirationDate getExpirationDate() setExpirationDate()	ApplicationObject の有効期限を決定します。アプリケーション開発者は、この日付に達した時点で ApplicationObject を期限切れにする方法を管理するエージェントを作成できます。

特別なメタデータと動作の管理

前項で説明したクラスは、様々な物理的情報タイプを管理するためのものです。たとえば、Document では、構造化されていない内容を持つ情報が管理されます。Folder では、コンテナシップ動作を持つ情報が管理されます。

Oracle 9*i*FS には、様々な物理タイプの情報に特別なメタデータを適用するためのクラスも用意されています。

- **Category** では、様々な物理タイプの情報がビジネス・カテゴリ別に編成され、そのカテゴリに関連する特別な属性が追加されます。
- **PropertyBundle** には、情報と名前 / 値のペアのハッシュ表が格納されます。
- **PolicyPropertyBundle** では、情報の処理方法に関するポリシーが適用されます。
- **Relationship** では、2 つの情報が関連付けられます。

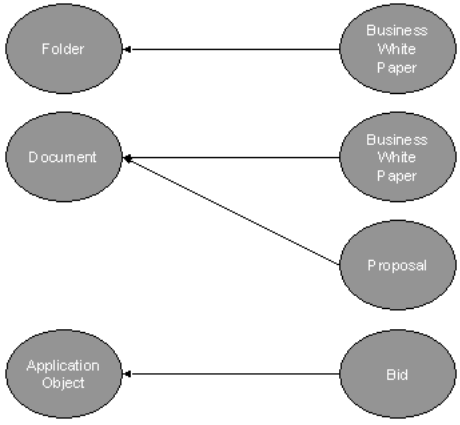
これらのクラスを使用して、ドキュメント、フォルダおよび他のパブリック・オブジェクトのコンテキストが記述されます。たとえば、ビジネスにおける情報の使用方法や、リポジトリ内の他の情報との関連が記述されます。特別なメタデータを使用すると、リポジトリ内の情報を検索できます。また、オーバーライドとサーバーによる情報の管理方法に関するビジネス・ルールの基礎としても使用できます。

Category

Oracle 9*i*FS には、情報を 1 つ以上のビジネス・カテゴリ別に編成できるように、Category クラスが用意されています。カテゴリに割り当てることで、情報に特別な属性と動作を追加できます。カテゴリは別個のクラスとして管理されるため、同じビジネス・メタデータを様々な物理タイプの情報に適用できます。

図 3-4「カテゴリ・オブジェクト・モデル」のように、Category を拡張して、Business White Paper、Proposal、Draft for Legislature および Bill のカテゴリを作成できます。これにより、各カテゴリに関連する付加的な属性を指定できます。たとえば、Business White Paper には属性 Product Name および Marketing Campaign を指定し、Request for Proposals には属性 Project および Cost を指定できます。

図 3-4 カテゴリ・オブジェクト・モデル



Document のインスタンスとして格納されている Microsoft Word ファイルは、その Document インスタンス用に Proposal クラスのインスタンスを作成し、Proposal カテゴリに分類できます。Proposal インスタンスは、属性 AssociatedPublicObject でカテゴリに分類されている Document を参照します。また、Proposal インスタンスは、Proposal としての Document の使用に関連する特別な属性ごとに値を持ちます。提案されたプロジェクトの開始後は、そのプロジェクトの製品に関する Business White Paper に同じ Document を使用できます。Document に Business White Paper クラスのインスタンスを適用して、その新規使用に関連する属性を追加できます。

また、Category は、情報がそのカテゴリに分類される場合にのみ適用されるカスタム動作を持ちます。たとえば、Bill クラスは、Document がすでに Draft for Legislature カテゴリに分類されていないかぎり、Bill にならないようにするメソッドを持つことができます。Bill メソッドは、Draft for Legislature インスタンスの Digital Signature 属性を検査して、Draft が Bill になるように認可されていることを確認できます。

同じ Category を、Document、Folder および ApplicationObject など、すべての物理タイプの情報に適用できます。たとえば、Business White Paper を構成する一連の Adobe FrameMaker ファイルを含むフォルダがあるとします。Folder 用に Business White Paper クラスのインスタンスを作成し、Business White Paper としての Folder の使用に関する属性を格納できます。後で FrameMaker のブックが単一の PDF ファイルとして発行されると、その PDF ドキュメントを含む Document もカテゴリ Business White Paper として分類されます。

Category クラスは階層形式で編成されるため、カスタム属性および動作を簡単に定義できます。たとえば、属性 DateGenerated および DataSource を持つ Category を拡張する Report クラスを作成できます。これにより、Report クラスを拡張し、属性 Quarter および Fiscal Year を持つ Financial Report と、属性 Product Number および Component を持つ Bug Report を作成できます。

PublicObject の検索条件として、その Category のメタデータを使用できます。そのためには、Search を作成し、Category に関する条件を指定する AttributeQualifications と、Category を PublicObject と結合する JoinQualification を含めます。

注意： アプリケーションでの Category の使用方法の詳細は、第 6 章「任意のメタデータと動作の適用」を参照してください。

表 3-15 に、Category で最も使用頻度の高い属性とメソッドを示します。

表 3-15 Category の属性およびメソッド

Name getName() setName()	Category 名を決定します。
Description getDescription() setDescription()	Category の説明を提供します。
Owner getOwner()	Category を所有するユーザーを決定します。ドキュメントの所有者には、そのフォルダへのすべてのアクセス権が暗黙的に付与されます。Owner 属性は DirectoryUser を参照します。
ACL getAcl() getDefaultAccessLevel() getEffectiveAccessLevel()	Category の AccessControlList を決定します。ACL 属性は AccessControlList を参照します。

表 3-15 Category の属性およびメソッド（続く）

Creator	Category を最初に作成したユーザーを決定します。Creator 属性は DirectoryUser を参照します。Creator 属性は Oracle 9*i*FS システムにより設定されます。
CreateDate	Category の作成日時が格納されます。この属性は Oracle 9*i*FS システムにより設定されます。
LastModifier	Category を最後に変更したユーザーを決定します。LastModifier 属性は DirectoryUser を参照します。LastModifier 属性は Oracle 9*i*FS システムにより設定されます。
LastModifyDate	Category の最終更新日時が格納されます。この属性は Oracle 9*i*FS システムにより設定されます。
ExpirationDate getExpirationDate() setExpirationDate()	Category の有効期限を決定します。アプリケーション開発者は、この日付に達した時点で Category を期限切れにする方法を管理するエージェントを作成できます。
AssociatedPublicObject getAssociatedPublicObject()	Category インスタンスが適用される PublicObject インスタンスを決定します。

表 3-16 に、Category を操作する場合に PublicObject で最も使用頻度の高いメソッドを示します。

表 3-16 PublicObject のメソッド

addCategory()	PublicObject のインスタンスに新規の Category インスタンスを適用します。
getCategories()	PublicObject に適用される 1 つまたはすべての Category インスタンスを取得します。

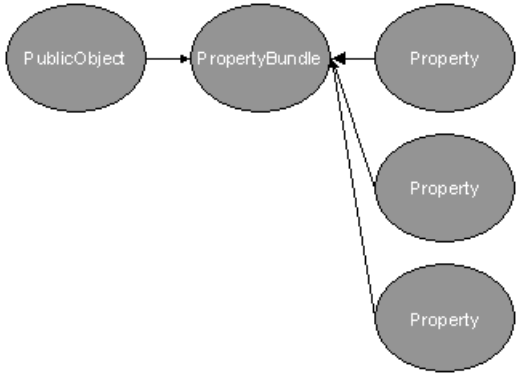
PropertyBundle

PublicObject の各インスタンスは PropertyBundle を持つことができます。PropertyBundle は、PublicObject インスタンスに関するアドホックなメタデータを、ハッシュ表のような名前 / 値のペア形式で格納するために使用されます。たとえば、Document の Name 属性の変換済みの値のリストを {english/Guide, spanish/Guida} として格納できます。PublicObject インスタンスは、メタデータを含む PropertyBundle インスタンスを参照する属性 PropertyBundle を持ちます。

図 3-5「PropertyBundle オブジェクト・モデル」のように、PropertyBundle 内の名前 / 値のペアは、それぞれ Property クラスのインスタンスとして表されます。Property インスタン

スは、属性 Bundle 経由で属している PropertyBundle を参照します。Property の値は、スカラー・データ型、オブジェクト・データ型、スカラー型とオブジェクト型の配列など、Oracle 9*i*FS で管理される型であれば、どのようなデータ型でもかまいません。

図 3-5 PropertyBundle オブジェクト・モデル



Search を作成し、PublicObject の Property に関する条件を指定する PropertyQualifications を含めると、PropertyBundle に含まれているメタデータに基づいて PublicObject を検索できます。

アプリケーションでの PropertyBundle の使用方法の詳細は、第 6 章「任意のメタデータと動作の適用」を参照してください。

表 3-17 に、PropertyBundle を操作する場合に PublicObject で最も使用頻度の高い属性とメソッドを示します。

表 3-17 PublicObject の属性およびメソッド

PropertyBundle getPropertyBundle() setPropertyBundle()	PublicObject の PropertyBundle を決定します。
putProperty()	PublicObject の PropertyBundle に新規 Property を追加します。
removeProperty() removeAllProperties()	PublicObject の PropertyBundle から 1 つまたはすべての Property を削除します。

表 3-18 に、PropertyBundle で最も使用頻度の高いメソッドを示します。

表 3-18 PropertyBundle のメソッド

getProperties() getPropertyValue() getPropertyValueByUpperCaseName()	PropertyBundle から Property またはその値をフェッチします。
putPropertyValue()	PropertyBundle に新規 Property を追加します。
removePropertyValue() removeAllPropertyValues()	PropertyBundle から 1 つ以上の Property を削除します。

表 3-19 に、Property で最も使用頻度の高い属性とメソッドを示します。

表 3-19 Property の属性およびメソッド

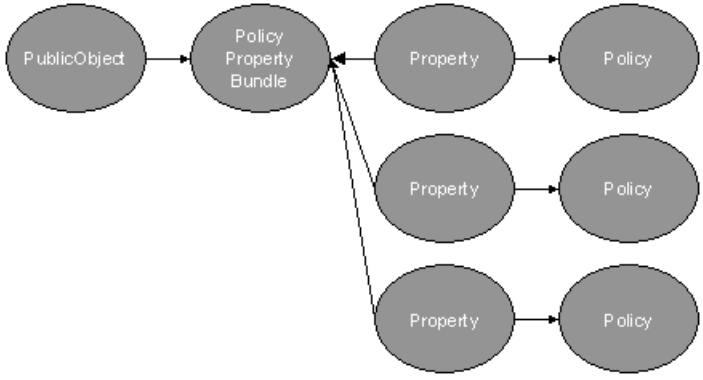
Bundle getBundle()	Property が属する PropertyBundle をフェッチします。
BooleanValue BooleanValues DateValue DateValues StringValue StringValues LongValue LongValues DoubleValue DoubleValues IntegerValue IntegerValues DirectoryObjectValue DirectoryObjectValues PublicObjectValue PublicObjectValues SystemObjectValue SystemObjectValues SchemaObjectValue SchemaObjectValues getValue() getDataType() setValue()	Property の値がその値のデータ型に従って格納されます。 getDateType() メソッドは値のデータ型を判断します。 getValue() および setValue() メソッドは Property の値を判断し、その値のデータ型に該当する属性を使用します。

PolicyPropertyBundle

PublicObject の各インスタンスは PolicyPropertyBundle を持つことができます。PolicyPropertyBundle は、特定の操作がコールされた場合の、PublicObject の処理方法に関するポリシー・セットを格納するために使用されます。

図 3-6「PolicyPropertyBundle オブジェクト・モデル」のように、PublicObject は属性 PolicyBundle を経由して PolicyPropertyBundle を参照します。PolicyPropertyBundle 内の各ポリシーは、Policy クラスのインスタンスとして表されます。Policy は、そのポリシーが適用される操作、ポリシーの動作を実装する JavaBeans の完全修飾名およびポリシーを識別する列挙キーの属性を持ちます。Policy は、PolicyPropertyBundle に Property の値として格納されます。Property インスタンスは、属性 Bundle 経由で属している PolicyPropertyBundle を参照します。

図 3-6 PolicyPropertyBundle オブジェクト・モデル



たとえば、Oracle 9*i*FS では、PolicyPropertyBundle を使用してレンダラの動作が実装されます。レンダラは、様々なコンテンツ・タイプのデータを各種の形式やレイアウトに変換するために使用されます。各レンダラは、Policy クラスのインスタンスに登録されます。Policy の Operation 属性では、情報をレンダリングするアプリケーションによりコールされる操作が識別されます。Policy の ImplementationName 属性では、レンダラの Java クラスが指定されます。レンダラの場合、ImplementationEnum 属性は使用されません。複数のレンダラが、PolicyPropertyBundle 内の Policy をまとめることで 1 つのコンテンツ・タイプに対応付けられています。

Oracle 9*i*FS では、コンテンツ・タイプのインスタンスに対する操作を実行するときに、まず PolicyPropertyBundle がチェックされ、その操作の処理に関するポリシーがあるかどうかが判断されます。たとえば、FTP クライアントがドキュメントの内容を表示する場合、Oracle 9*i*FS サーバーでは、最初に Document の PolicyPropertyBundle で操作 FtpRenderer に関する Policy の有無がチェックされます。PolicyPropertyBundle にはこの操作の Policy が含まれており、その ImplementationName 属性で ContentRenderer のパス

`oracle.ifs.server.renderers.ContentRenderer` が指定されています。このパスを使用して、FTP クライアントで表示できる方法で Document の内容がレンダリングされます。

アプリケーションでの PolicyPropertyBundle の使用方法の詳細は、第 6 章「任意のメタデータと動作の適用」を参照してください。

表 3-20 に、PolicyPropertyBundle を操作する場合に PublicObject の最も使用頻度の高い属性とメソッドを示します。

表 3-20 PublicObject の属性およびメソッド

PolicyBundle getPolicyBundle() setPropertyBundle()	PublicObject の PolicyPropertyBundle を決定します。
putPolicy()	PublicObject の PolicyPropertyBundle に新規 Policy を追加します。
removePolicy() removeAllPolicies()	PublicObject の PolicyPropertyBundle から 1 つまたはすべての Policy を削除します。

表 3-21 に、PropertyBundle で最も使用頻度の高いメソッドを示します。

表 3-21 PropertyBundle の属性およびメソッド

getProperties() getPropertyValue() getPropertyValueByUpperCaseName()	PropertyBundle から Property またはその値をフェッチします。
putPropertyValue()	PropertyBundle に新規 Property を追加します。
removePropertyValue() removeAllPropertyValues()	PropertyBundle から 1 つ以上の Property を削除します。

表 3-22 に、Property で最も使用頻度の高いメソッドを示します。

表 3-22 Property の属性およびメソッド

Bundle getBundle()	Property が属する PropertyBundle をフェッチします。
SystemObjectValue getValue() getDataType() setValue()	Policy により SystemObject が拡張されるため、SystemObjectValue 属性には Policy が Property の値として格納されます。 getValue() および setValue() メソッドでは、Policy が Property の値として判断されます。

表 3-23 に、Policy で最も使用頻度の高いメソッドを示します。

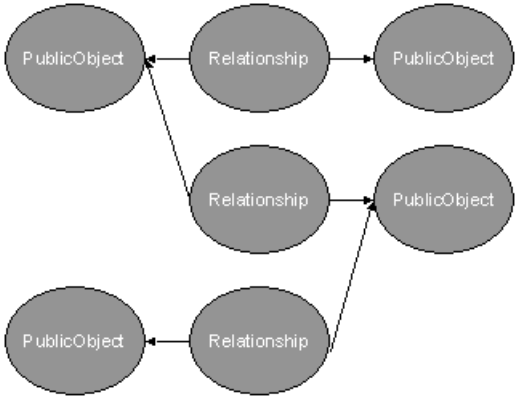
表 3-23 Policy の属性およびメソッド

Operation getOperation() setOperation()	ポリシーが適用される操作を決定します。
ImplementationName getImplementationName() setImplementationName()	Policy の動作を実装する JavaBeans の完全修飾パスを決定します。
ImplementationEnum getImplementationEnum() setImplementationEnum()	PolicyPropertyBundle の Policy について実装の列挙値を決定します。

Relationship

Oracle 9*i*FS には、PublicObject を相互に対応付けることができるように、もう 1 つのクラスである Relationship が用意されています。図 3-7「Relationship オブジェクト・モデル」のように、Relationship は RightObject および LeftObject という 2 つの属性を持ちます。この 2 つの属性は、それぞれ関係の左辺または右辺で PublicObject を参照します。

図 3-7 Relationship オブジェクト・モデル



PublicObject と中間的な Relationship との対応関係を示すモデルを作成する方法は、多対多の関係を管理する場合に適しています。Relationship を使用すると、関係の両側で PublicObject を効率的に取得できます。Relationship を使用すると、特定の PublicObject を RightObject として参照するすべての PublicObject、または特定の PublicObject を LeftObject として参照するすべての PublicObject を取得できます。どちらの場合も、RightObject 属性と LeftObject 属性の一意キー索引により問合せが最適化されます。

Relationship には、特定の関係の両側から PublicObject を取得できるように、基本的なメソッド `getRightObject()` および `getLeftObject()` が用意されています。また、PublicObject には、PublicObject のすべての関係および関連 PublicObject をすばやく取得できるように、便利なメソッドが用意されています。たとえば、`getLeftwardRelationshipObjects()` メソッドは、それに関連するすべての PublicObject を LeftObject として取得します。Relationship にも、管理者が関連情報の属性や内容に基づいて情報を検索できるように、強化された手段が用意されています。たとえば、管理者は、Joe Smith が所有するドキュメントを含むすべてのフォルダを検索できます。検索アプリケーションでは、関連 PublicObject に関係する SearchQualification および関連 PublicObject を Relationship と結合する JoinQualification を含む Search が構成されます。

Search、SearchQualification および JoinQualification については、第 8 章「検索アプリケーションの構築」を参照してください。

また、Relationship を使用すると、関係のコンテキスト内でのみ適用可能な、PublicObject の特別な属性と動作を格納できます。Relationship クラスを拡張し、付加的な属性と動作を持つカスタム・タイプの関係を作成できます。たとえば、FolderPathRelationship クラスは Relationship を拡張して、同じフォルダ内のアイテムが一意の名前を持つことを保証するロジックを実装します。カスタムの Relationship クラスを作成し、レターと添付ファイル、調査ドキュメントと参考資料、ワープロで作成されたドキュメントと HTML、PDF および XML 表示、または複合ドキュメントとコンポーネントとの対応付けを管理できます。

アプリケーションでの Relationship の使用方法の詳細は、第 6 章「任意のメタデータと動作の適用」を参照してください。

表 3-24 に、Relationship で最も使用頻度の高い属性とメソッドを示します。

表 3-24 Relationship の属性およびメソッド

LeftObject getLeftObject()	関係の左辺として表されるフォルダを決定します。
RightObject getRightObject()	関係の右辺として表されるフォルダに含まれているアイテムを決定します。

表 3-25 に、Relationship を操作する場合の PublicObject のメソッドを示します。

表 3-25 PublicObject のメソッド

addRelationship()	この PublicObject と別の PublicObject との Relationship を作成します。このメソッドは、この PublicObject を Relationship の LeftObject にする必要があるものとみなします。
getLeftwardRelationshipObjects() getRightwardRelationshipObjects()	この PublicObject に RightObject または LeftObject として関係するすべての PublicObject を取得します。
getLeftwardRelationships() getRightwardRelationships()	この PublicObject が RightObject または LeftObject となるすべての Relationship インスタンスを取得します。
removeRelationship()	この PublicObject と別の PublicObject との Relationship を削除します。

情報のバージョニング

Oracle 9*i*FS には、PublicObject のバージョニング用のクラス・セットが用意されています。バージョニングとは、PublicObject に対するすべての変更を、そのライフ・サイクルを通じて追跡する手段です。たとえば、組織は変更者、変更日時および変更順序を記録して、ドキュメントの内容や属性に対する変更をすべて追跡する必要があるとします。その場合、組織は、ドキュメントの各バージョンのバックアップを、特定の時点でのドキュメントの体裁記録として保存できます。

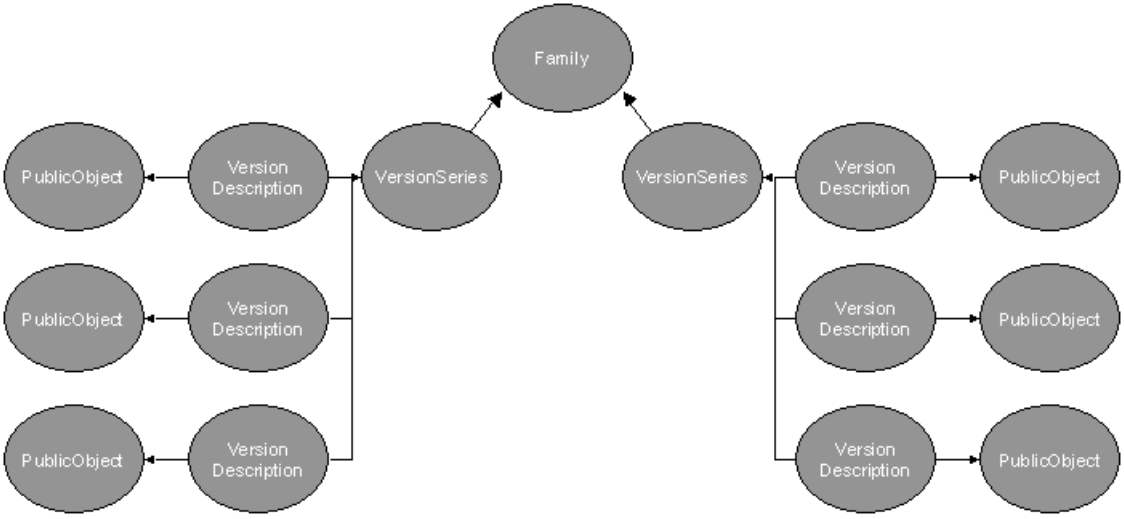
Oracle 9*i*FS には、情報を作成して発行するための多様なプロセスをサポートできるように、柔軟なバージョニング・モデルが用意されています。Oracle 9*i*FS では、どのような PublicObject でもバージョニングできます。他のクラスのオブジェクトをバージョニングする計画がある場合は、必要なディスク領域とパフォーマンスを考慮することが重要です。たとえば、フォルダをバージョニングするには、フォルダの属性に対して行われる変更ごとに、Folder の個別インスタンスを格納できます。また、フォルダの特定バージョンを LeftObject として参照する FolderPathRelationship の個別インスタンスを作成し、フォルダの各バージョンに含まれるアイテムを追跡する方法もあります。ただし、Folder および FolderPathRelationship の個別インスタンスのデータをデータベースに格納するために必要になる、追加のディスク領域の量を検討する必要があります。また、フォルダやそこに含まれるアイテムを検索するときに、追加のインスタンスがアプリケーションのパフォーマンスにどの程度影響するかも考慮する必要があります。

バージョニングの詳細は、第 14 章「バージョニングの実装」を参照してください。

図 3-8「バージョニング・オブジェクト・モデル」のように、Oracle 9*i*FS では、PublicObject をバージョニングするために Family、VersionSeries および VersionDescription という 3 つのクラスが使用されます。

- **Family:** Family クラスは、PublicObject のライフ・サイクル全体を表します。このクラスでは、PublicObject に関して作成されたバージョンがすべて収集されます。
- **VersionSeries:** Family では、バージョンが VersionSeries を使用してグループ化されます。VersionSeries クラスでは、PublicObject に対する一連の変更が順番に表されます。VersionSeries 内の各バージョンは、特定時点での PublicObject の状態を表します。VersionSeries 内のバージョンの順序は、PublicObject に対して行われた変更の順序を表します。
- **VersionDescription:** VersionSeries 内では、PublicObject の各バージョンは VersionDescription で表されます。VersionDescription では、順序番号（1、2、3 など）のように、シリーズ内のバージョンのコンテキストが記述されます。VersionDescription の各インスタンスは、PublicObject のインスタンスを参照します。

図 3-8 バージョニング・オブジェクト・モデル



Oracle 9*i*FS クライアントおよびプロトコルでは、ドキュメントについてのみシリアル・バージョニング・モデルが実装されます。シリアル・バージョニング・モデルでは、ドキュメントの Family にのみ VersionSeries が 1 つ含まれ、その VersionSeries にドキュメントのすべてのバージョンが含まれます。したがって、各バージョンは相互に順番になっている必要があります。

Oracle 9*i*FS のバージョニング・モデルには、パラレル・バージョニングや分岐バージョニングなど、より複雑なバージョニングのパラダイムを持つカスタム・アプリケーションをサポートする機能があります。

ほとんどの PublicObject のインスタンスはバージョニングできます。PublicObject の Family に複数の VersionSeries を含めて、パラレルに行われた変更の個別の順序や、相互に分岐またはマージする変更の順序を表すことができます。

VersionSeries の各 VersionDescription では様々なクラスのオブジェクトを参照できるため、クラス間での PublicObject の状態の変化を表すことができます。たとえば、最初のバージョンは、ブックを構成する FrameMaker ファイルのセットを含む Folder のインスタンスであるとします。第 2 のバージョンの場合、FrameMaker のブックは Document のインスタンスとして格納される PDF ファイルに変換されている場合があります。

VersionDescription では、PublicObject の同一インスタンスも参照できます。たとえば、ドキュメントに発行前 VersionSeries と発行済み VersionSeries があるとします。発行前 VersionSeries の第 3 のバージョンでは、発行済み VersionSeries の最初のバージョンを表している場合があります。この場合、第 3 の発行前バージョンと最初の発行済みバージョンを表す VersionDescription では、Document の同一インスタンスを参照できます。

表 3-26 に、Family で最も使用頻度の高い属性とメソッドを示します。

表 3-26 Family の属性およびメソッド

PrimaryVersionSeries getPrimaryVersionSeries() setPrimaryVersionSeries()	PublicObject のプライマリ・バージョン・シリーズを指定してアクセスします。 Family には複数の VersionSeries を含めることができるため、Family は PrimaryVersionSeries 属性を持ちます。この属性は、VersionSeries の 1 つを PublicObject についてデフォルトでアクセスされる VersionSeries として指定するために使用されます。
getResolvedPublicObject()	有効ドキュメントの現行の状態を表す PublicObject のバージョンにアクセスします。たとえば、ユーザーがバージョニング対象のドキュメントをダブルクリックし、その内容を表示すると、Oracle 9*i*FS では DefaultVersionDescription 属性で指定されたバージョンが戻されます。この属性が指定されていない場合、Oracle 9*i*FS では、バージョニング対象ドキュメントの現行の状態を表すプライマリ・バージョン・シリーズの最終バージョンが戻されます。
DefaultVersionDescription getDefaultVersionDescription() setDefaultVersionDescription()	カスタム・バージョニング・アプリケーションは、これらの属性とメソッドを使用して、プライマリ・バージョン・シリーズの最終バージョンとは異なる PublicObject のバージョンを、デフォルト・バージョンとして明示的に設定できます。

表 3-27 に、VersionSeries で最も使用頻度の高い属性とメソッドを示します。

表 3-27 VersionSeries の属性およびメソッド

getFirstVersionDescription()	バージョン・シリーズの最初のバージョンにアクセスします。
LastVersionDescription getLastVersionDescription()	バージョン・シリーズの最終バージョンを格納し、アクセスします。
getVersionDescriptions() getVersionDescriptions(int index) getNextVersionDescription() getPreviousVersionDescription()	バージョン・シリーズ内の、すべてのバージョンの配列、特定バージョンおよび次または前のバージョンにアクセスします。
DefaultVersionDescription getDefaultVersionDescription() setDefaultVersionDescription()	カスタム・バージョニング・アプリケーションは、これらの属性とメソッドを使用して、プライマリ・バージョン・シリーズの最終バージョンとは異なる PublicObject のバージョンを、デフォルト・バージョンとして明示的に設定できます。
Reservor ReservationDate ReservationComments reserveNext() unReserve() isReserved() isReservedByCurrentUser() getReservor() getReservationDate() getReservationComment()	特定のユーザーが変更を行っている間は、他のユーザーがシリーズの新バージョンを作成しないように、VersionSeries を予約し、予約を解除します。Oracle 9*i*FS では、バージョン・シリーズが予約された時期、予約したユーザーおよびユーザー指定のコメントが追跡されます。
WorkPath getWorkPath()	ユーザーが PublicObject に変更を加える場合に、バージョニング・アプリケーションはそのユーザーに対して、ドキュメントをオーサリング・クライアントのローカル・オペレーティング・システムにコピーするように許可できます。WorkPath 属性を使用すると、エクスポートされたコピーへのパスを追跡できます。 **注意：** Oracle 9*i*FS クライアントおよびプロトコルでは、この機能を提供するために WorkPath 属性が使用されることはありません。

表 3-27 VersionSeries の属性およびメソッド

PendingPublicObject getPendingPublicObject() setPendingPublicObject()	バージョン・シリーズを予約したユーザーは、変更結果を新バージョンとしてチェックインする前に一時的に格納できます。変更は、PublicObject の別インスタンスとして一時的に格納され、PendingPublicObject 属性により参照されます。
newVersion()	バージョン・シリーズの新バージョンを作成するために使用されます。
VersionLimit setVersionLimit()	Oracle 9*i*FS によりバージョン・シリーズ内で保持されるバージョン数の制限を設定するために使用されます。カスタム・アプリケーションでは、この属性を使用してバージョン・パージ機能を実装できます。

表 3-28 に、VersionDescription で最も使用頻度の高い属性とメソッドを示します。

表 3-28 VersionDescription の属性およびメソッド

VersionSeries VersionNumber RevisionComment getVersionSeries() getVersionNumber() getRevisionComment()	このバージョンが属するバージョン・シリーズ、バージョン・シリーズ内でこのバージョンに割り当てられた順序番号、バージョンの作成時にユーザーにより指定されたコメントを追跡します。
VersionLabel getVersionLabel() setVersionLabel()	VersionLabel 属性には、バージョン・シリーズのバージョンのカスタム・ラベルが格納されます。バージョニング・アプリケーションでは、組織の改訂採番方式に準拠するバージョン・ラベルを格納できます。[1]
isLatestVersionDescription()	バージョンがバージョン・シリーズの最終バージョンかどうかを判断します。
PublicObject getPublicObject()	各バージョンは、最終的に PublicObject のインスタンスとして格納されます。VersionDescription の PublicObject 属性は、そのバージョンを構成する PublicObject インスタンスを参照します。getPublicObject() メソッドは、PublicObject を取得するために使用されます。

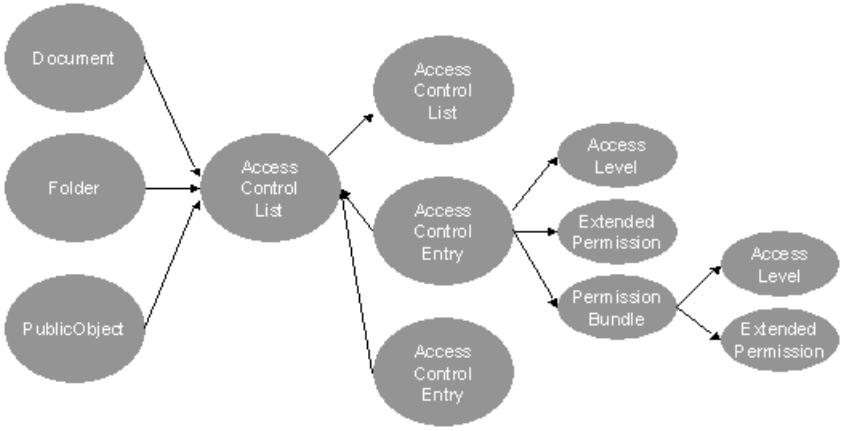
[1] たとえば、Oracle 9*i*FS ではバージョンに順序を示す整数（1、2、3）が自動的に割り当てられますが、組織ではバージョンにラベル（ドラフト A、ドラフト B、ドラフト C、リリース 1.0 など）が必要な場合があります。

情報へのアクセスの制御

図 3-9「ACL オブジェクト・モデル」のように、Oracle 9*i*FS では、情報へのアクセスを制御するために主として `AccessControlList`、`ClassAccessControlList` および `SystemAccessControlList` という 3 つのクラスが使用されます。この 3 つは、PublicObject の派生クラスです。したがって、いずれも、エンド・ユーザーが情報へのアクセスを制御するために使用するプライマリ・インタフェースです。この 3 つのクラスは、他の 4 つのクラス、つまり AccessControlEntry、AccessLevel、PermissionBundle および ExtendedPermission によりサポートされます。

セキュリティの詳細は、第 15 章「セキュリティ」を参照してください。

図 3-9 ACL オブジェクト・モデル



AccessControlList

AccessControlList は、リポジトリ内の情報に対する特定の権限が付与されているユーザーとグループのリストを表します。AccessControlList の単一インスタンスを 1 つ以上のドキュメント、フォルダまたは任意の PublicObject に適用できます。この方法で、AccessControlList はリポジトリ内の情報についてセキュリティを定義して適用するための一元的な場所として機能します。

PublicObject の各インスタンスは、そのインスタンスに AccessControlList を割り当てるために使用される属性 ACL を持ちます。Oracle 9*i*FS では、ユーザーに PublicObject インスタンスに対するアクションの実行を許可する前に、まず ACL 属性で参照される AccessControlList がチェックされ、そのユーザーにそのアクションを実行する権限が付与されているかどうかが判断されます。

AccessControlList は PublicObject の派生クラスのため、同じく ACL 属性を持っています。この属性は、AccessControlList 内での権限の付与、取消しおよび AccessControlList の削除ができるユーザーとグループを制御するために使用されます。

表 3-29 に、AccessControlList で最も使用頻度の高い属性とメソッドを示します。

表 3-29 AccessControlList の属性およびメソッド

Shared isShared()	AccessControlList を複数の PublicObject 間で共有できるかどうかを決定します。
checkEffectiveAccess() checkGrantedAccess() getEffectiveAccessLevel() getGrantedAccessLevel()	AccessControlList でユーザーまたはグループに付与されているアクセス権を判断します。
grantAccess() removeAccessControlEntry() revokeAccess() revokeAllAccess() updateAccessControlEntry()	ユーザーが AccessControlList でアクセス権を付与または取り消すことができるように、便利なインタフェースを提供します。
isComposite() getComponentAcls() setComponentAcls() getCompositeAcls()	複合 ACL の取得および複合 ACL を構成するコンポーネント ACL の設定と取得のためのメソッド。boolean 型の `isComposite()` は、ターゲット ACL が複合 ACL（1 つ以上のコンポーネント ACL を含む ACL）である場合、true を戻します。
ACL	この AccessControlList での権限の付与と取消し、またはこの AccessControlList の削除ができるユーザーとグループを制御する、別の AccessControlList を参照します。

AccessControlEntry

AccessControlEntry は、AccessControlList でユーザーまたはグループに付与された権限の各セットを表すために使用されます。AccessControlEntry は、ユーザーまたはグループ、権限、および権限が付与されるか取り消されるかを決定するための属性とメソッドを持ちます。AccessControlEntry の各インスタンスは、属性 ACL を経由して、属している AccessControlList を参照します。

表 3-30 に、AccessControlEntry で最も使用頻度の高い属性とメソッドを示します。

表 3-30 AccessControlEntry の属性およびメソッド

ACL getACL()	この AccessControlEntry が属している AccessControlList を判断します。
Grantee getGrantee()	権限が付与または取り消されるユーザーまたはグループを判断します。
AccessLevel getDistinctAccessLevel() getMergedAccessLevel()	付与または取り消される権限を判断します。
Granted	権限が付与されるか取り消されるかを示します。
SortSequence getSortSequence()	この AccessControlEntry の、AccessControlList でのソート順序を決定します。
PermissionBundles getPermissionBundles()	AccessControlEntry に設定されている PermissionBundle を判断します。
ExtendedPermissions getExtendedPermissions()	この AccessControlEntry で付与または取り消された拡張権限を判断します。

AccessLevel

リポジトリ内の情報について付与または取消しが可能な各権限は、AccessLevel クラスで表されます。AccessLevel は、様々な権限を指定するために使用される定数のセットを持ちます。権限には、情報の存在を検出するための権限、クラスのインスタンスを作成する権限、属性や内容を設定する権限、情報を削除する権限などがあります。AccessLevel には広範囲な権限リストが用意されており、情報へのアクセス方法を厳密に制御できます。AccessLevel クラスの定数で表される権限を次に示します。

All	Selector Access	Create	Set Content
None	Get Content	Delete	Set Default Version
Discover	Grant	Set Attribute	Set Policy
Remove Member	Add Member	Remove Item	Add Item
Add Relationship	Remove Relationship	Lock	Unlock
Add Version	Remove Version	Add VersionSeries	Remove Version Series

表 3-31 に、AccessLevel で最も使用頻度の高い属性とメソッドを示します。

表 3-31 AccessLevel の属性およびメソッド

add() equal() subtract()	この AccessLevel について、別の AccessLevel に存在する権限を追加または削除します。
clearAllPermissions() clearAllStandardPermissions() disableAllPermissions() disableAllStandardPermissions() enableAllPermissions() enableAllStandardPermissions()	AccessLevel での権限を消去、有効または無効にします。
getAllDefinedStandardPermissions() getAllDefinedStandardPermissionNames() enableEnabledStandardPermissions() getEnabledStandardPermissionNames() isStandardPermissionEnabled() isSufficientEnabled()	AccessLevel で有効化され、定義されている権限を判断します。

ExtendedPermission

ExtendedPermission では、カスタムのアクセス権が定義されます。たとえば、法的契約書の管理アプリケーションをサポートするために、ドキュメントにデジタル署名を添付できるかどうかについて、カスタム権限を定義できます。このカスタム権限を定義するには、一意の名前（'Sign' など）でカスタム権限を識別する、ExtendedPermission クラスのインスタンスを作成します。

その後は、AccessLevel を使用して、AccessControlEntry 内で ExtendedPermission を付与または取り消すことができます。ExtendedPermission を保持する AccessLevel は、標準的な権限と同様に簡単に構成できます。Java API の AccessLevel クラスには、ExtendedPermission を操作するためのメソッドのセットが用意されています。AccessLevel は、AccessControlList の AccessControlEntry 内でユーザーに対して付与または取り消すことができます。

表 3-32 に、ExtendedPermission の操作に使用される AccessLevel のメソッドを示します。

表 3-32 ExtendedPermission に関連する AccessLevel のメソッド

enableExtendedPermission()	AccessLevel 内で ExtendedPermission を設定するために使用されます。
disableExtendedPermission()	AccessLevel から ExtendedPermission を削除するために使用されます。
getExtendedPermissions()	AccessLevel に保持されているすべての ExtendedPermission を取得するために使用されます。
isExtendedPermissionEnabled()	ExtendedPermission が AccessLevel に保持されているかどうかをチェックするために使用されます。

ExtendedPermission が AccessControlList に適用された後は、ExtendedPermission を使用してユーザーによる PublicObject の操作方法を制御するように、カスタム・アプリケーションを作成できます。アプリケーションでは、ユーザーが PublicObject に対する ExtendedPermission を付与されているかどうかをチェックし、検出結果に基づいて操作の実行を許可または禁止するオーバーライドを実装できます。

表 3-33 に、ExtendedPermission の操作に使用される S_PublicObject のメソッドを示します。

表 3-33 ExtendedPermission に関連する S_PublicObject のメソッド

checkAccess()	S_PublicObject のユーザーに ExtendedPermission が付与されているかどうかを判断します。
extendedPostInsert() extendedPreInsert() extendedPostUpdate() extendedPreUpdate() extendedPostDelete() extendedPreDelete()	ユーザーが実行できる操作を ExtendedPermission に基づいて制御する、カスタム・ビジネス・ルールを実装します。

PermissionBundle

AccessLevel で定義される権限よりも広範囲にアクセスを管理するには、PermissionBundle を使用して AccessLevel をグループ化できます。PermissionBundle は Oracle 9*i*FS に永続的に格納されており、ユーザーに付与できる上位レベルのセキュリティ設定を表すために使用できます。たとえば、ユーザーにフォルダの変更権限を簡単に付与できるように、Discover、Add Item および Remove Item の権限を Modify Folder Permission Bundle にグループ化できます。

表 3-34 に、PermissionBundle で最も使用頻度の高い属性とメソッドを示します。

表 3-34 PermissionBundle の属性およびメソッド

AccessLevel getAccessLevel() setAccessLevel()	PermissionBundle での AccessLevel を判断します。

ClassAccessControlList

ClassAccessControlList は、AccessControlList を拡張します。したがって、AccessControlEntry、AccessLevel、ExtendedPermission および PermissionBundle も使用されます。ClassAccessControlList は、ユーザーによるコンテンツ・タイプの操作方法を管理するように特化されています。管理者は、ClassAccessControlList を使用して、コンテンツ・タイプをインスタンス化したり削除できるユーザーやグループを制御します。ClassAccessControlList は、Oracle 9*i*FS で一意の名前を持つように拡張 UniqueName 属性を持っています。

SystemAccessControlList

SystemAccessControlList も、AccessControlList を拡張します。

SystemAccessControlList は、UniqueName 属性を持つシステム単位の AccessControlList を表すために使用されます。このクラスを作成できるのは、管理ユーザーのみです。Oracle 9*i*FS には、インストール時に SystemAccessControlList の次の 4 つのインスタンスが含まれています。

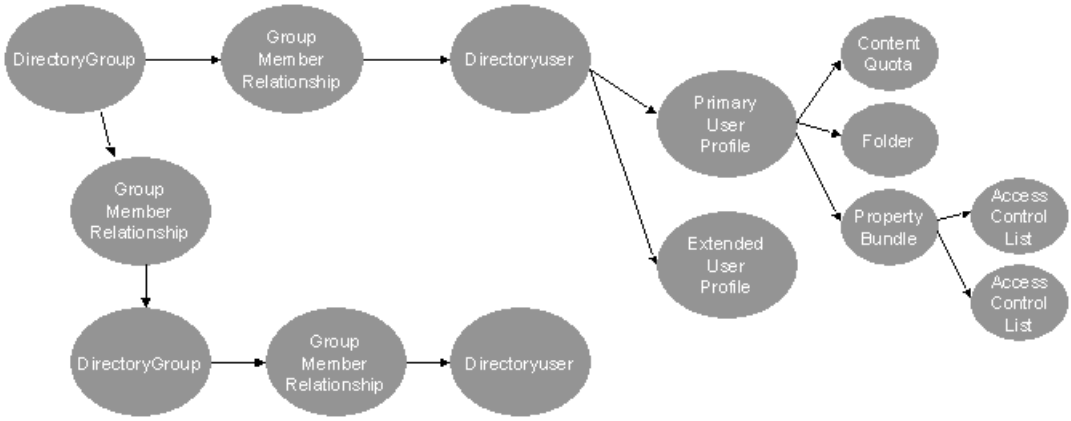
- Private
- Protected
- Public
- Published

SystemAccessControlList は、Oracle 9*i*FS で一意の名前を持つように拡張 UniqueName 属性を持っています。

ディレクトリの管理

図 3-10「ディレクトリ・オブジェクト・モデル」のように、Oracle 9*i*FS にはユーザーおよびグループ情報を管理するためのクラスのセットが用意されています。DirectoryUser および DirectoryGroup は PublicObject の派生クラスで、Oracle 9*i*FS のリポジトリ内の情報を操作するユーザーとグループを表すためのプライマリ・インタフェースとして機能します。この 2 つのクラスは、リポジトリでの操作に関する各ユーザーの設定項目と、ユーザーをグループに関連付ける方法を表すための、少数の SystemObject クラスでサポートされます。

図 3-10 ディレクトリ・オブジェクト・モデル



アプリケーションでのディレクトリの管理方法は、第 15 章「セキュリティ」を参照してください。

DirectoryUser

Oracle 9*i*FS にログインできる各ユーザーは、DirectoryUser クラスのインスタンスで表されます。

DirectoryUser は、ユーザーの管理権限の割当て、ユーザーへの一意名の割当て、ユーザー認証に使用する資格証明マネージャの指定など、ユーザーを管理するための属性とメソッドを持っています。

表 3-35 に、DirectoryUser で最も使用頻度の高い属性とメソッドを示します。

表 3-35 DirectoryUser の属性およびメソッド

AdminEnabled isAdminEnabled() setAdminEnabled()	ユーザーが Oracle 9*i*FS システムでの管理権限を持っているかどうかを判断します。
CredentialManager getCredentialManager() setCredentialManager()	ユーザーが Oracle 9*i*FS システムにログインするときに、ユーザーの資格証明の認証に使用される資格証明マネージャを判断します。
DistinguishedName UniqueName getDistinguishedName() setDistinguishedName()	システム内でユーザーを一意に識別します。

PrimaryUserProfile、ExtendedUserProfile および ContentQuota

Oracle 9*i*FS には、Oracle 9*i*FS リポジトリ内での操作に関するユーザーの設定項目を管理できるように、SystemObject クラスのセットが用意されています。

- **PrimaryUserProfile** は、ユーザーのホーム・フォルダやコンテンツ・クオータなど、Oracle 9*i*FS でデフォルトで定義されている設定項目を管理するための属性とメソッドを持ちます。
- **ExtendedUserProfile** には、カスタム設定項目を管理するための基本コンテンツ・タイプが用意されています。
- **ContentQuota** は、ユーザーまたはグループが Oracle 9*i*FS リポジトリに格納できるコンテンツの量を管理するために使用されます。

表 3-36 に、PrimaryUserProfile で最も使用頻度の高い属性とメソッドを示します。

表 3-36 PrimaryUserProfile の属性およびメソッド

ContentQuota getContentQuota() setContentQuota()	リポジトリへのコンテンツ格納に関するユーザーのクオータを判断します。 ContentQuota 属性は、ユーザーのクオータを表す ContentQuota のインスタンスを参照します。

表 3-36 PrimaryUserProfile の属性およびメソッド（続く）

DefaultACLs getDefaultACLs() setDefaultACLs()	ユーザーがリポジトリ内で作成する情報にデフォルトで適用される AccessControlList を判断します。 DefaultACLs 属性は、デフォルトの AccessControlList を含む PropertyBundle を参照します。
HomeFolder getHomeFolder() setHomeFolder()	ユーザーの個人用記憶領域として機能するフォルダを判断します。ユーザーが Oracle 9*i*FS クライアントにログインすると、デフォルトでホーム・フォルダに置かれます。 HomeFolder 属性は、ユーザーのホーム・フォルダとして機能する Folder のインスタンスを参照します。

表 3-37 に、ExtendedUserProfile で最も使用頻度の高い属性とメソッドを示します。

表 3-37 ExtendedUserProfile の属性およびメソッド

`Application`
`getApplication()`
`setApplication()`
	ExtendedUserProfile が適用されるカスタム・アプリケーションを示します。同じ Oracle 9*i*FS システム用に複数のカスタム・アプリケーションを構築できます。各ユーザーは、各アプリケーションでの操作用に異なるプロファイルを持つことができます。

表 3-38 に、ContentQuota で最も使用頻度の高い属性とメソッドを示します。

表 3-38 ContentQuota の属性およびメソッド

AllocatedStorage getAllocatedStorage() setAllocatedStorage()	ユーザー用の記憶領域を割り当てます。
ConsumedStorage calculateConsumedStorage() getConsumedStorage()	ユーザーがクオータのうち消費した領域の量を判断します。
AssociatedPublicObject getAssociatedPublicObject()	ContentQuota が適用される PublicObject を指定します。
Enabled isEnabled() setEnabled()	ユーザーのコンテンツ・クオータが有効かどうかを判断します。管理者は、ユーザーのコンテンツ・クオータを無効化して、リポジトリ内で無制限の記憶領域を与えることができます。

DirectoryGroup および GroupMemberRelationship

ユーザーはグループ単位で編成できます。1 つのグループに必要な数のユーザーを含めることができます。また、グループには他のグループを含めることもできます。これにより、グループを階層形式に含めて並べ替えることができます。

共通の要素を持つユーザーを表すグループを作成できます。たとえば、ユーザーが持つ様々なロールを表すグループ（Manager、Developer、Writer など）を作成する方法があります。また、会社の部門やプロジェクトを表すグループ（Marketing、Sales、Development など）を作成する方法もあります。

グループは、多数のユーザー間で情報へのアクセス権を一貫した方法で付与する場合に役立ちます。たとえば、AccessControlEntry の Grantee として Manager グループを指定する AccessControlList を作成し、すべてのマネージャにリポジトリ内のドキュメントへの読取りアクセス権を付与できます。

グループは、Oracle 9*i*FS では `DirectoryGroup` および `GroupMemberRelationship` という 2 つのクラスで表されます。`DirectoryGroup` は PublicObject の派生クラスのため、グループのメンバーシップを管理するためのプライマリ・インタフェースとして機能します。`GroupMemberRelationship` は SystemObject の派生クラスであり、ユーザーとグループ間の関係を表します。GroupMemberRelationship のインスタンスは、各ユーザーまたはグループのメンバーシップを表すように作成されます。

`DirectoryGroup` を拡張してグループを分類し、そのタイプのグループ専用の特別な属性と動作を追加できます。たとえば、`DirectoryGroup` を拡張する Division グループを作成し、属性 Vice President および Mission Statement を含めることができます。また、`GroupMemberRelationship` を拡張し、Team Role など、Division グループの各ユーザーのメンバーシップに関する特別なメタデータを管理できます。

表 3-39 に、DirectoryGroup で最も使用頻度の高い属性とメソッドを示します。

表 3-39 DirectoryGroup の属性およびメソッド

addMember() addMembers()	グループのメンバーとしてユーザーまたはグループ、あるいはその両方を追加します。
getAllMembers() getAllUserMembers() getDirectMembers() isMember()	グループのメンバーであるユーザーまたはグループ、あるいはその両方を判断します。直接のメンバーは、このグループに含まれる他のグループに属しているユーザーやグループではなく、このグループに直接関連しているユーザーまたはグループです。
removeMember() removeMembers()	グループからユーザーまたはグループ、あるいはその両方を削除します。

表 3-40 に、GroupMemberRelationship で最も使用頻度の高い属性とメソッドを示します。

表 3-40 GroupMemberRelationship の属性およびメソッド

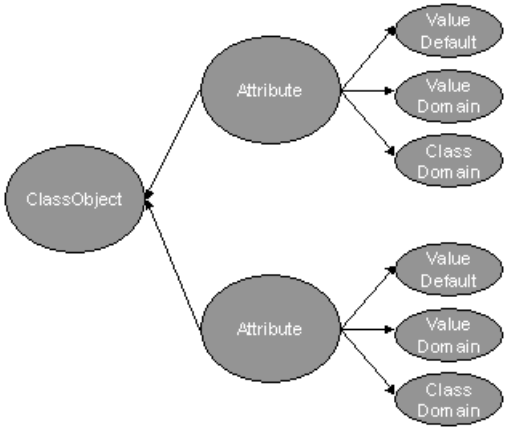
LeftObject getLeftObject()	LeftObject 属性は DirectoryGroup を参照します。
RightObject getRightObject()	RightObject 属性は、DirectoryGroup に属しているユーザーまたはグループを参照します。

コンテンツ・タイプの定義

Oracle 9*i*FS では、管理対象となるすべてのコンテンツ・タイプのレジストリをメンテナンスするために、もう 1 組のクラスが使用されます。この種のクラスは、`oracle.ifs.beans` パッケージ内の SchemaObject クラスの派生クラスです。各クラスの機能は、コンテンツ・タイプの属性、コンテンツ・タイプの動作を実装する Java クラス、コンテンツ・タイプのインスタンスをデータベースに格納する方法など、各コンテンツ・タイプに関するメタデータを記録することです。

図 3-11「SchemaObject モデル」のように、このレジストリの作成に使用されるクラスは、ClassObject、Attribute、AttributeDefault、AttributeDomain、ClassDomain などです。

図 3-11 SchemaObject モデル



Oracle 9*i*FS 内の各コンテンツ・タイプは、ClassObject クラスのインスタンスで表されます。コンテンツ・タイプの属性は、それぞれ Attribute クラスのインスタンスで表されます。

ValueDefault、ValueDomain および ClassDomain クラスは、コンテンツ・タイプの属性に対する制約を定義するために使用されます。

ClassObject

ClassObject は、Oracle 9*i*FS 内のコンテンツ・タイプを登録するための属性と動作を持っています。ClassObject では、コンテンツ・タイプの名前と記述、コンテンツ・タイプのインスタンスが格納される表、コンテンツ・タイプの動作の実装に使用される Java クラスが記録されます。

表 3-41 に、ClassObject で最も使用頻度の高い属性とメソッドを示します。

表 3-41 ClassObject の属性およびメソッド

Name getName() setName()	コンテンツ・タイプの名前（Document、Folder など）を決定します。コンテンツ・タイプを表す ClassObject インスタンスの作成後は、その Name は変更できません。
DatabaseObjectName getDatabaseObjectName()	コンテンツ・タイプのインスタンスの格納に使用されるデータベース内の表を決定します。コンテンツ・タイプを表す ClassObject インスタンスの作成後は、DatabaseObjectName は変更できません。
BeanClasspath getBeanClasspath() setBeanClasspath() ServerClasspath getServerClasspath() setServerClasspath() SelectorClasspath getSelectorClasspath() setSelectorClasspath()	コンテンツ・タイプの動作の実装に使用される Java クラスを決定します。新規コンテンツ・タイプのパスを指定していない場合は、そのスーパークラスから動作が取得されます。
Abstract Final Partitioned isAbstract() isFinal() isPartitioned()	コンテンツ・タイプをインスタンス化できる（isAbstract() = false）かどうかと、サブクラス化できる（isFinal() = false）かどうかを決定します。Partitioned 属性により、コンテンツ・タイプのインスタンスを格納する表がパーティション化されるかどうかが決定されます。
isVersionable()	コンテンツ・タイプのインスタンスをバージョニングできるかどうかを決定します。

表 3-41 ClassObject の属性およびメソッド（続く）

ClassACL getClassACL() setClassACL()	ユーザーとグループがコンテンツ・タイプをインスタンス化できるかどうかを制御する、ClassObject の ClassAccessControlList を提供します。
SuperClass getSuperClass() getDirectSubclasses() getSubclasses() isSubclassOf()	クラス階層を記録し、横断します。
addAttribute() getEffectiveClassAttributes() getExtendedClassAttributes() removeAttribute()	コンテンツ・タイプの属性を追加、削除およびリスト表示します。属性とコンテンツ・タイプ間の関係は、ClassObject インスタンスには記録されませんが、Attribute インスタンスの Class 属性で記録されます。

Attribute

Attribute は、コンテンツ・タイプの属性を登録するための属性とメソッドを持ちます。Attribute には、属性の名前、データ型、スケールおよび長さなどのメタデータが記録されます。また、属性値の格納に使用される列の名前も記録されます。

表 3-42 に、Attribute で最も使用頻度の高い属性とメソッドを示します。

表 3-42 Attribute の属性およびメソッド

Name getName() setName()	属性名（Owner、CreateDate など）を決定します。コンテンツ・タイプの属性を表す Attribute インスタンスの作成後は、その Name は変更できません。
DatabaseObjectName getDatabaseObjectName()	コンテンツ・タイプのインスタンスの格納に使用されるデータベース内の列を決定します。この列は、属性の ClassObject に指定された表に置かれます。コンテンツ・タイプを表す Attribute インスタンスの作成後は、DatabaseObjectName は変更できません。
Class getDefiningClass()	属性が属しているコンテンツ・タイプの ClassObject（Document、Folder など）を決定します。

表 3-42 Attribute の属性およびメソッド（続く）

Datatype DataLength DataScale getDataType() getDataTypeLabel() getDataLength() getDataScale()	属性のデータ型、長さおよびスケールを決定します。属性には、INTEGER、STRING および DATE のようなスカラー・データ型や、PublicObject、SystemObject および DirectoryObject のような Oracle 9*i*FS のオブジェクト・データ型、またはこれらの型の配列を使用できます。
ValueDefault getAttributeDefault() setAttributeDefault() ValueDomain ValueDomainValidated getValueDomain() setValueDomain() isValueDomainValidated() ClassDomain getClassDomain() setClassDomain() Indexed isIndexed() Required isRequired() Unique isUnique() Updatable isUpdatable()	属性値に制約を適用します。ValueDefault、ValueDomain および ClassDomain の各制約については、それぞれ「ValueDefault」、「ValueDomain」および「ClassDomain」を参照してください。
ReferentialIntegrityRule RIRULE_CLEAR RIRULE_RESTRICT getReferentialIntegrityRule() setReferentialIntegrityRule()	属性の値がオブジェクトを参照する場合に適用される、参照整合性ルールを決定します。 RIRULE_CLEAR 制約は、参照先オブジェクトが削除された場合に、属性の値を NULL に設定する必要があることを示すために使用されます。RIRULE_RESTRICT 制約は、この属性で参照されている間は、Oracle 9*i*FS でオブジェクトの削除を禁止する必要があることを示すために使用されます。

ValueDefault

属性値のデフォルトは、コンテンツ・タイプのインスタンスの作成時に自動的に設定できます。属性のデフォルト値を設定すると、エンド・ユーザーは各属性の値を手動で入力せずに、新規インスタンスを簡単に作成できます。属性が通常は特定の値に設定されるのであれば、その値を Oracle 9*i*FS で自動的に設定できます。

デフォルト値は、ValueDefault のインスタンスで表されます。ValueDefault は、値がどのようなデータ型の場合にも Attibute のデフォルト値を設定する汎用的な方法を提供します。ValueDefault では、PropertyBundle 内の値が保持されます。ValueDefault は、Oracle 9*i*FS システムでデフォルト値を一意に識別して取得するための属性およびメソッドを持っています。

表 3-43 に、ValueDefault で最も使用頻度の高い属性とメソッドを示します。

表 3-43 ValueDefault の属性およびメソッド

UniqueName	ValueDefault の一意名を指定します。
ValueDefaultPropertyBundle getValueDefaultPropertyBundle() setValueDefaultPropertyBundle()	値を保持する PropertyBundle を決定します。
getPropertyValue()	PropertyBundle から値を取得する便利な方法を提供します。

ValueDomain

属性値は、列挙値のドメインまたは値のドメイン範囲に制限できます。ドメイン範囲では、値が指定した境界値より小さい、以下、より大きい、以上、両端の値を除く境界値の範囲内および境界値の範囲内になるように要求できます。属性値に制約を適用してドメインに限定すると、エンド・ユーザーに確実に有効値を入力させることができます。たとえば、PublicationDate 属性の値を 2000 年以後になるように制限したり、Publisher 属性の値を {'ACME Publishing', 'GreenTree Publishing', 'Atlantis Publishing'} のいずれか 1 つに制限できます。

属性値に適用するドメインは、Oracle 9*i*FS に ValueDomain のインスタンスとして登録されます。ドメイン範囲または列挙値は、PropertyBundle に保持されます。ValueDomain の定数セットは、値に適用するドメインのタイプを示すために使用されます。

表 3-44 に、ValueDomain で最も使用頻度の高い属性とメソッドを示します。

表 3-44 ValueDomain の属性およびメソッド

UniqueName	ValueDomain の一意名を指定します。
ValueDomainPropertyBundle getValueDomainPropertyBundle() setValueDomainPropertyBundle()	列挙値のドメイン、つまり有効な範囲を保持する PropertyBundle を決定します。
VALUEDOMAINTYPE_ENUMERATED VALUEDOMAINTYPE_EXCLUSIVE_MAXIMUM VALUEDOMAINTYPE_EXCLUSIVE_MINIMUM VALUEDOMAINTYPE_EXCLUSIVE_RANGE VALUEDOMAINTYPE_MAXIMUM VALUEDOMAINTYPE_MINIMUM VALUEDOMAINTYPE_RANGE toValueDomainTypeLabel() valueDomainTypeResourceBundleKey()	値ドメインのタイプを決定します。

ClassDomain

属性値がオブジェクト（PublicObject、System Object、SchemaObject、DirectoryObject など）またはその配列を参照する場合は、値を 1 つ以上のコンテンツ・タイプに限定できます。たとえば、属性のデータ型が PublicObject の場合は、値が Document のインスタンスになるように要求できます。また、値を Document のサブクラスのインスタンスにできるかどうかも指定できます。

この種の制約は、Oracle 9*i*FS に ClassDomain のインスタンスとして登録されます。ClassDomain を使用すると、オブジェクトの相互関係の管理を属性経由でさらに厳密に制御できます。特に、複合ドキュメント管理アプリケーションを構築する場合に役立ちます。たとえば、属性 Ordered_Products を持つコンテンツ・タイプ PurchaseOrder を作成する場合に、その属性が Product コンテンツ・タイプのインスタンスを参照する必要があるとします。Ordered_Products 属性のデータ型を PublicObject、ClassDomain を Product として指定できます。

表 3-45 に、ClassDomain で最も使用頻度の高い属性およびメソッドを示します。

表 3-45 ClassDomain の属性およびメソッド

UniqueName	ClassDomain の一意名を指定します。
Classes getClasses() setClasses()	ドメインのコンテンツ・タイプを決定します。各コンテンツ・タイプは、ClassObject クラスのインスタンスの Classes 属性で表されます。
DomainType CLASSDOMAINTYPE_ENUMERATED CLASSDOMAINTYPE_ENUMERATED_AND_SUBCLASSES getDomainType() setDomainType()	ClassDomain を列挙されたコンテンツ・タイプのみに限定する必要があるか、または派生するコンテンツ・タイプを含むことができるかを決定します。

コンテンツ・タイプの動作の拡張

「情報の管理」で説明したように、Oracle 9*i*FS で管理される各コンテンツ・タイプは、データを格納する表のセットと、コンテンツ・タイプの動作を実装する Java クラスのセットで構成されています。各コンテンツ・タイプは、次の 3 つの Java クラスを持つことができます。

- **Bean 側 Java クラス：** Bean 側 Java クラスは、コンテンツ・タイプを操作するためのプライマリ・インタフェースを提供します。この種のクラスは、`oracle.ifs.beans` パッケージにあります。
- **サーバー側 Java クラス：** サーバー側 Java クラスには、コンテンツ・タイプの動作を実装するコードが含まれています。この種のクラスは、`oracle.ifs.server` パッケージにあります。サーバー側 Java クラスのネーミング規則 `S_<classname>` により、この種のクラスは Bean 側クラスにパラレルになります。
- **Selector Java クラス：** Selector Java クラスは、Oracle 9*i*FS 内部でオブジェクトをインスタンス化するために使用されます。カスタム・アプリケーションの構築時には、この Java クラスのカスタマイズ作業は行いません。

新規に作成したコンテンツ・タイプは、それを拡張するコンテンツ・タイプの動作を自動的に持つことになります。たとえば、Document を拡張する新規コンテンツ・タイプ SimpleImage を作成すると、`oracle.ifs.beans.Document` クラスと `oracle.ifs.server.S_Document` クラスで実装される動作を持ちます。コンテンツ・タイプの動作をカスタマイズする必要がある場合は、そのカスタム Java クラスを作成するのみで済みます。

Oracle 9*i*FS には、Java クラスを拡張してカスタム動作を実装できるように便利なインタフェースを提供する、Tie クラスというクラス・セットが用意されています。Bean 側とサーバー側の Java クラスには、それぞれ対応する Tie クラスがあります。たとえば、Document Java クラスは対応するクラス `TieDocument` を持ち、`S_Document` は対応するクラス `S_TieDocument` を持ちます。Tie クラスは、それに対応する Bean 側またはサーバー側 Java クラスを拡張します。

コンテンツ・タイプの動作を拡張する必要がある場合は、Tie クラスを拡張または置き換えるカスタム Java クラスを作成します。Tie クラスは Bean 側とサーバー側の Java クラスを拡張するため、カスタム Java クラスは Oracle 9*i*FS の各クラスで実装される動作を継承します。

コンテンツ・タイプのインスタンス化

Oracle 9*i*FS には、コンテンツ・タイプのインスタンス作成に使用するクラス・セット、Definition クラスも用意されています。

Oracle 9*i*FS で管理される各コンテンツ・タイプは、Definition クラスを持ちます。Definition クラスのネーミング規則 `<classname>Definition` により、このクラスはコンテンツ・タイプの Bean 側 Java クラスとパラレルになります。すべての Definition クラスは `oracle.ifs.beans` パッケージにあり、トップレベルのクラス `LibraryObjectDefinition` を拡張します。

Definition クラスは、情報データを一時的に保持する構造をアセンブルするために使用されます。この構造を使用して Bean 側 Java クラスがインスタンス化され、情報がリポジトリに永続的に格納されます。

また、類似する複数のインスタンスを作成する場合に、カスタム・アプリケーションに大幅な柔軟性をもたらします。たとえば、同じ Owner および Format を持つ複数の Document を作成する必要があれば、Owner および Format の値を保持する Definition を作成し、それをテンプレートとして使用して、各 Document を作成できます。

さらに、Definition を使用すると複合コンテンツ・タイプを簡単にインスタンス化できます。複数のコンポーネントで構成されるコンテンツ・タイプを作成する場合は、コンポーネントごとに Definition を作成してから、コンテンツ・タイプを単一コールでインスタンス化できます。たとえば、バージョニング対象のドキュメントは Document、VersionDescription、VersionSeries および Family という 4 つのコンポーネントで構成されています。コンポーネントごとに Definition を作成してから、Oracle 9*i*FS API の単一コールで 4 つの永続オブジェクトすべてを作成できます。

情報の検索

Oracle 9*i*FS Java API には、リポジトリ内のあらゆるタイプの情報を検索して取り出せるように、Java クラスのセットが用意されています。この種のクラスを使用すると、次のように取り出す情報に関する条件を指定できます。

- **コンテンツ・タイプ / クラス条件 :** 取り出す必要のある情報のタイプ（Document、Folder、DirectoryObject、カスタム・コンテンツ・タイプなど）を指定できます。
- **属性条件 :** 名前、所有者または変更日など、情報に関する属性に基づいて検索するか、owner = 'user1' や modification date > January 1, 2001 のように属性条件を結合してブール構造体を指定できます。
- **内容条件 :** ドキュメントの内容に基づいて検索できるように、Oracle 9*i*FS ではデータベースの Oracle Text 機能を使用してドキュメントの内容に索引を付けます。ドキュメントの検索時には、ドキュメントに含まれる語句（トークン）を指定する方法、ブール（AND、OR、ACCUMULATE、MINUS など）を使用して内容条件を結合する方法、トークン間の近接性（'XML' near 'Internet Applications' など）を指定する方法があります。Oracle Text の言語分析機能を使用すると、ドキュメントを件名（'about XML' など）に基づいて検索し、シソーラス、Soundex（発音の類似）、またはファジー演算子およびワイルドカードを使用して検索対象を拡張し、関連または類似するトークンを含むドキュメントを検索できます。Oracle Text 機能のすべての内容検索機能は、Oracle 9*i*FS 検索の構成時に使用できます。

  Oracle Text の詳細は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。
- **フォルダ制限 :** 検索対象を限定し、特定のフォルダまたは特定のフォルダ分岐にある情報を取り出すことができます。フォルダは、リポジトリ内のオブジェクトを編成する場合に役立ちます。フォルダ制限による検索では、検索対象をフォルダ階層の特定の分岐にあるオブジェクトに限定できます。
- **関連情報 :** 関連情報に関する条件に基づいて情報を検索できます。たとえば、ドキュメントの検索時に、そのドキュメントを参照するフォルダの属性を条件として指定できます。
- **ソート条件 :** 検索により取り出された情報のどの属性順にでも、検索結果をソートできます。また、内容条件を指定した場合は、検索結果のドキュメントを、そのドキュメントの内容と条件との関連性に基づいてソートできます（この関連性は、ドキュメントのスコアと呼ばれます）。

Oracle 9*i*FS には、リポジトリの検索と検索結果の操作用に、次の 2 つのインタフェースが用意されています。

- **Selector** は、単純問合せを実行するための使用しやすいインタフェースです。
- **Search** は、複合問合せを作成するための確実なインタフェースです。

Selector

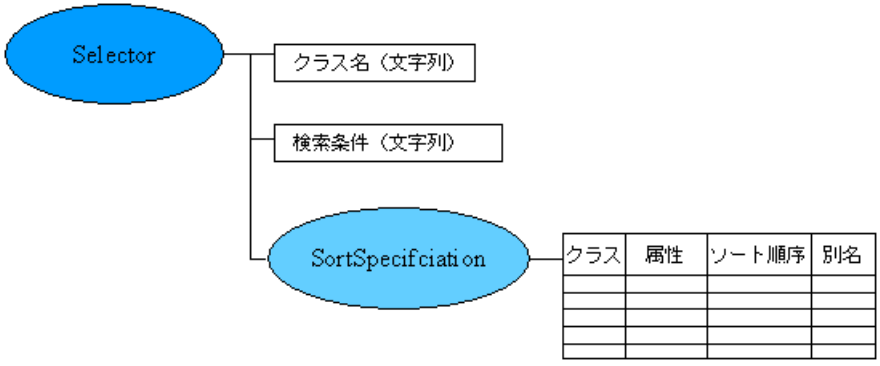
Selector を使用すると、リポジトリ内で単純問合せを実行できます。属性に関する静的条件を指定して単一クラスの情報を検索する必要があれば、Selector クラスのインスタンスを作成して問合せを作成し、実行できます。

例 3-1 単純問合せ

```
select all Documents whose Owner is 'user1' and whose Name ends with '.html'
```

図 3-12「Selector オブジェクト・モデル」のように、Selector は、3 つのコンポーネント、つまり、クラスを指定する文字列、検索条件を指定する文字列およびソート条件を指定する SortSpecification オブジェクトで構成されています。

図 3-12 Selector オブジェクト・モデル



Selector は、次のルールに基づいて動作します。

- **単一のコンテンツ・タイプ / クラス：** Selector では、1 種類の情報を検索できます。複数のコンテンツ・タイプを選択したり、結合を構成して関連コンテンツ・タイプに関する条件を指定することはできません。条件は、Oracle 9*i*FS で Selector から構成される SQL SELECT 文の FROM 句となります。
- **再帰的なコンテンツ・タイプ / クラス：** Selector では、指定したコンテンツ・タイプ / クラスから派生するすべてのコンテンツ・タイプを再帰的に検索できます。
- **自由形式の条件：** Selector では、単一文字列で検索条件を指定できます。この文字列は、Oracle 9*i*FS で Selector から構成される SQL SELECT 文の WHERE 句となります。
- **遅延バインド変数なし：** Selector では、遅延バインド変数を使用して検索条件を動的に構成することはできません。検索条件は、単一文字列で表現可能である必要があります。

- **ソート条件：** Selector では、検索結果を対象となったコンテンツ・タイプの属性順にソートできます。このソート条件は、Oracle 9*i*FS で Selector から構成される SQL SELECT 文の ORDER BY 句となります。

表 3-46 に、Selector クラスで最も使用頻度の高い属性とメソッドを示します。

表 3-46 Selector の属性およびメソッド

getSearchClassname() setSearchClassname()	問合せ対象となるクラスを決定します。
isRecursiveSearch()	Selector で派生クラス間を再帰的に問い合せる必要があるかどうかを決定します。
getSearchSelection() setSearchSelection()	問合せ条件を決定します。
getSortSpecification() setSortSpecification()	問合せ結果のソート順序を決定します。
getItems() getItems(int index)	問合せを実行して LibraryObject の配列を戻します。
getItemCount() openItems() closeItems() resetItems() nextItem()	問合せ結果を操作します。 getItemCount() は、結果の配列に含まれる項目数を決定します。 openItems() は、Selector 上でカーソルをオープンします。 nextItem() は、カーソルを使用して結果の配列内で次の項目をフェッチします。 resetItems() は、結果の消去に使用されます。 closeItems() は、Selector をクローズします。

Oracle 9*i*FS Java API には、Selector 用のソート条件を構成するための SortSpecification クラスも用意されています。SortSpecification は、Selector 上で `setSortSpecification()` メソッドを使用して設定されます。

表 3-47 に、SortSpecification クラスで最も使用頻度の高い属性とメソッドを示します。

表 3-47 SortSpecification の属性およびメソッド

addSortQualifiers()	SortSpecification に属性の配列とソート修飾子の配列を追加します。
addSortQualifier()	SortSpecification に、単一の属性、ソート順序およびオプションで属性のクラスと別名を追加します。
getSortAttributes()	SortSpecification に現在含まれているすべての属性の配列を戻します。
getSortOrders()	SortSpecification に現在含まれているすべてのソート順序の配列を戻します。
getDefaultClass() setDefaultClass()	SortSpecification にある属性のデフォルト・クラスを決定します。
getDefaultAlias() setDefaultAlias()	SortSpecification にある属性のデフォルト別名を決定します。

Search

Search インタフェースは、リポジトリ内で複合問合せを構成して実行できるように、より確実なインタフェースを提供します。Search では、属性、内容、フォルダ、他の相互関係および非定型情報に関する動的条件に基づいて、複数クラスの情報を問合せできます。

例 3-2 複合問合せ

```
"find all Documents and Folders which are in my Home Folder, whose ModifiedDate is
less than <VariableDate>, and whose Name contains 'XML' or whose Content contains
'XML' "
```

Search インタフェースでは、Java 構造体を使用して、オブジェクト指向の方法で複合問合せをアセンブルできます。図 3-13「Search オブジェクト・モデル」に、FROM 句、WHERE 句、WHERE 句の各 EXPRESSION および ORDER BY 句など、リポジトリに対する問合せ用の SQL SELECT 文の各コンポーネントを Java オブジェクトとして表す方法を示します。これらの各種コンポーネントは、問合せの実行に使用される Search オブジェクトに渡す Java 構造体または仕様部として動的にアセンブルできます。Oracle 9*i*FS では、問合せの実行時に SQL SELECT 文が自動的に生成され、リポジトリに対する問合せが実行されます。その後、Oracle 9*i*FS では問合せ結果が情報オブジェクトの配列として戻され、それを読み取って操作できます。

図 3-13 Search オブジェクト・モデル

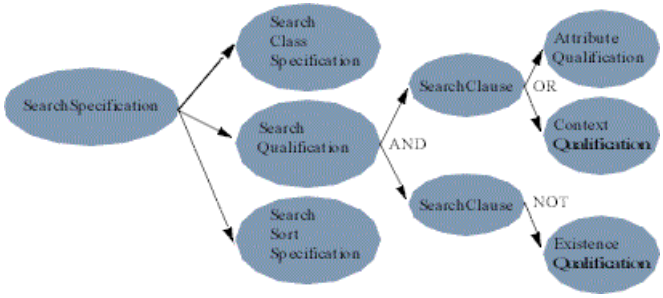


表 3-48 に、SearchSpecification のコンポーネントと SQL 問合せのコンポーネントとのマッピングを示します。

表 3-48 SQL と Java の比較

`SELECT * FROM <list>`	SearchClassSpecification
`WHERE`	SearchQualification
`(`	SearchClause
`<expression>`	AttributeQualficiation、FolderQualficiation、ContextQualficiation、ExistenceQualification、FreeFormQualification または PropertyQualification
`<join>`	JoinQualfication
`(`	SearchClause
`<expression>`	AttributeQualficiation、FolderQualficiation、ContextQualficiation、ExistenceQualification、FreeFormQualification または PropertyQualification
`<join>`	JoinQualification
`<expression>`	AttributeQualficiation、FolderQualficiation、ContextQualficiation、ExistenceQualification、FreeFormQualification または PropertyQualification
`ORDER BY <list>`	SearchSortQualification

このように、Oracle 9*i*FS の Search インタフェースでは、複合問合せをオブジェクト指向の方法でプログラムによりアセンブルできます。Java 構造を使用して問合せを定義すると、各

コンポーネントを柔軟に操作し、問合せを動的に変更し、リポジトリに対して複数回実行できます。また、Oracle 9*i*FS の基礎となるスキーマの複雑さを把握する必要もなくなります。

Oracle 9*i*FS の Search インタフェースには、検索の各種コンポーネントを構成し、問合せを実行し、検索結果を操作できるように、Java クラスのセットが用意されています。

- **SearchSpecification** は、検索の構造体全体を保持します。
- **SearchClassSpecification** は、検索の FROM 句に含まれるクラスを保持します。
- **SearchQualification** は、検索の WHERE 句を構成するすべての条件をアセンブルするための抽象クラスです。このクラスは、AttributeQualification、FolderRestrictQualification、ContextQualification、ExistenceQualification、FreeFormQualification、PropertyQualification および SearchClause により拡張されます。
- **AttributeQualification** は、属性条件に関係する WHERE 句の式を構成するために使用されます。
- **FolderRestrictQualification** は、フォルダ条件に関係する WHERE 句の式を構成するために使用されます。
- **ContextQualification** は、内容条件に関係する WHERE 句の式を構成するために使用されます。
- **ExistenceQualification** は、WHERE 句に IN 文を構成するために使用されます。
- **PropertyQualification** は、プロパティ条件に関係する WHERE 句の式を構成するために使用されます。
- **FreeFormQualification** は、WHERE 句に非定型式を含む文字列を指定するために使用されます。
- **JoinQualification** は、WHERE 句に結合を構成するために使用されます。
- **SearchClause** は、検索の WHERE 句で条件式を結合するために使用されます。
- **SearchSortSpecification** は、検索の ORDER BY 句の条件を保持します。
- **Search** は、検索を実行して結果を取り出すために使用されます。
- **SearchResultsObject** は、検索結果の操作に使用されます。

SearchSpecification

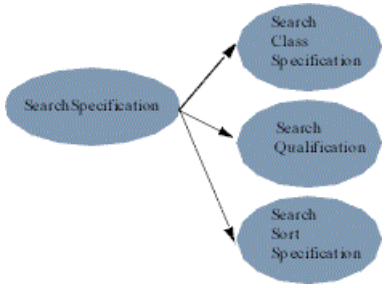
Oracle 9*i*FS で Search を実行するには、最初に SearchSpecification を構成します。SearchSpecification には、問合せを実行するまで、Search のすべてのコンポーネントが一時的に保持されます。

図 3-14「SearchSpecification のコンポーネント」のように、SearchSpecification は次の 3 つのコンポーネントから構成されています。

- SearchClassQualification

- SearchQualification
- SearchSortQualification

図 3-14 SearchSpecification のコンポーネント



SearchSpecifications には、AttributeSearchSpecification および ContextSearchSpecification という 2 つのタイプがあります。Search をアセンブルするときには、検索条件に応じて AttributeSearchSpecification または ContextSearchSpecification を構成します。

AttributeSearchSpecification: ドキュメントの内容に基づく条件を指定せずに Search を構成する場合に使用します。

Oracle 9*i*FS では、条件の性質に従って検索が自動的に最適化されます。同じ Search（Owner = "user1" の Document など）でも、構成に ContextSearchSpecification を使用するよりは、AttributeSearchSpecification を使用する方がパフォーマンスが向上します。

表 3-49 に、AttributeSearchSpecification クラスで最も使用頻度の高いメソッドを示します。

表 3-49 AttributeSearchSpecification のメソッド

getClassSpecification() setClassSpecification()	検索対象となる情報のクラスを指定する SearchClassSpecification を設定します。
getSearchQualification() setSearchQualification()	検索対象となる情報のクラスを指定する SearchQualification を設定します。
getSearchSortSpecification() setSearchSortSpecification()	検索結果のソート順序を指定する SearchSortSpecification を設定します。

ContextSearchSpecification: 内容条件（Content に 'XML' を含む Document の選択など）を含む Search を構成する場合に使用します。ContextSearchSpecification の SearchQualification コンポーネントには、1 つ以上の ContextQualification が必要です。

Oracle 9*i*FS では、ContextSearchSpecification に基づく Search の実行時に、リポジトリにドキュメントの内容を格納する表が自動的に組み込まれ、SQL 文中に対応する contains() 句が構成されます。

ContextSearchSpecification は AttributeSearchSpecification を拡張するため、表 3-49 に定義したメソッドと同じメソッドを持ちます。

表 3-50 に、ContextSearchSpecification クラスに用意されているその他のメソッドを示します。

表 3-50 ContextSearchSpecification のメソッド

setContextClassname() getContextClassname()	ContextQualification に指定した条件に基づいて検索されるクラスを指定します。デフォルトでは、Oracle 9*i*FS ではすべての Document の内容が ContentObject のインスタンスとして格納されます。ContentObject は属性 Content を持つクラスであり、この属性にはドキュメントの実際の内容が格納されます。

SearchClassSpecification

SearchClassSpecification は、SQL SELECT 文の 'FROM' 句を構成します。ただし、Oracle 9*i*FS は情報をオブジェクト指向の方法で操作するため、SelectClassSpecification オブジェクトでは、SQL のように表のリストを指定するのではなく、問合せ対象となるクラスのリストを指定します。

Oracle 9*i*FS は、検索結果として、情報オブジェクトの属性の配列ではなく、これらのオブジェクトのリストを戻します。各情報オブジェクトは、SearchClassSpecification に指定されたクラスのインスタンスである必要があります。これにより、アプリケーションでは、各情報オブジェクトを構成するすべてのデータ（属性、内容など）にアクセスしたり、各オブジェクトに対して操作（チェックイン、チェックアウトなど）を実行できます。このような理由から、SQL SELECT 文のように選択リストを指定する必要はありません。Oracle 9*i*FS の検索の場合、選択リストは常に 'SELECT *' です。

表 3-51 に、SearchClassSpecification クラスで最も使用頻度の高いメソッドを示します。

表 3-51 SearchClassSpecification のメソッド

addSearchClass() addSearchClasses() getSearchClassnames()	内容条件が適用される特定の情報クラスを決定します。そのクラスの別名指定にも使用されます。

表 3-51 SearchClassSpecification のメソッド（続く）

getSearchClassAliases()	SearchClassSpecification 内のクラスの別名のリストを取得します。
getRecursiveBehavior()	検索をクラスのサブクラス間で再帰的に適用する必要があるかどうかを決定します。
addResultClass() getResultClassnames()	検索で戻される必要のある情報のクラスを決定します。検索で戻される情報のクラスは、Classes 属性でも指定する必要があります。

SearchQualification

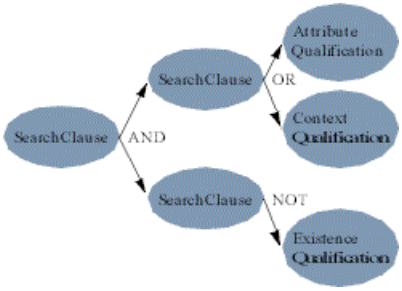
SearchQualification は、検索条件を保持する抽象クラスです。Oracle 9*i*FS では、SQL SELECT 文の 'WHERE' 句を構成するために使用されます。

SearchQualification では、owner = 'user1' や contains ('XML') など、様々な条件に基づいて検索条件を指定できます。条件は、SearchQualification 内で条件を表すオブジェクトのグループをアセンブルして定義します。

図 3-15「SearchQualification のアセンブリ」のように、'WHERE' 句は SearchQualification のサブクラスの複数のインスタンスをネストすることで構成されます。

- AttributeQualification
- FolderRestrictQualification
- ContextQualification
- ExistenceQualification
- PropertyQualification
- FreeFormQualification
- JoinQualification
- SearchClause

図 3-15 SearchQualification のアセンブリ



AttributeQualification

AttributeQualification は、属性値に関する条件を表します。属性は、Oracle 9*i*FS でサポートされるデータ型であれば、どのような型でもかまいません。

例 3-3 属性ベースの問合せ

```
"find all documents where the owner is jdoe"
```

表 3-52 に、AttributeQualification クラスで最も使用頻度の高い定数とメソッドを示します。

表 3-52 AttributeQualification の定数とメソッド

getAttributeName() getAttributeClassname() setAttribute()	条件の基礎となる Attribute を決定します。
getOperatorType() setOperatorType() EQUAL GREATER_THAN GREATER_THAN_EQUAL IS_NOT_N ULL IS_NULL LESS_THAN LESS_THAN_EQUAL LIKE NOT_EQUAL	Attribute を指定の値と比較する演算子を決定します。 定数は、条件中の比較演算子を表します。現在、Oracle 9*i*FS でサポートされている演算子は、文字列に対する 'IS_NULL'、'IS_NOT_NULL' および 'LIKE' のみです。
getValue() setValue()	Attribute の比較対象となる値を決定します。 **注意：** 演算子 LIKE を使用すると、有効な SQL ワイルドカード（% や _ など）を値に使用できます。
getDateComparisonLevel() setDateComparisonLevel() DATE_COMP_DAY DATE_COMP_HOUR DATE_COMP_MIN DATE_COMP_MONTH DATE_COMP_SEC DATE_COMP_YEAR	DATE 型の Attribute の場合、これらのメソッドと定数は、Oracle 9*i*FS で属性と日付値を比較するレベルを指定するために使用されます。 たとえば、AttributeQualification を使用すると、'CreateDate' Attribute を、'March 21, 2001 08:30 am'（'CreateDate < SYSDATE' など）に設定されているシステム生成の日付 'SYSDATE' と比較できます。ただし、2001 年 3 月 20 日以前、つまり 2001 年 3 月 20 日 23:59 p.m. 以前に作成されたすべての情報を取り出す必要がある場合があります。AttributeQualification で 2001 年 3 月 21 日 7:00 a.m. に作成された情報が戻されるのを防ぐには、日付の比較レベルを DATE_COMP_DAY に設定します。

表 3-52 AttributeQualification の定数とメソッド（続く）

isCaseIgnored() setCaseIgnore()	条件に大 / 小文字区別が必要かどうかを決定します。 たとえば、IgnoreCase を 'true' に設定すると、条件 NAME = 'XML' では NAME が 'xml' の情報は戻されません。

FolderRestrictQualification

FolderRestrictQualification は、検索対象を指定のフォルダまたはフォルダの分岐に限定する条件を表します。

例 3-4 フォルダ制限による問合せ

```
"find all documents in my home folder"
```

表 3-53 に、FolderRestrictQualification クラスで最も使用頻度の高いメソッドを示します。

表 3-53 FolderRestrictQualification のメソッド

getStartFolder() setStartingFolder()	検索の限定対象となるフォルダを決定します。フォルダ階層の複数レベルの分岐にまたがって検索する場合、StartFolder は分岐の最上位ノードです。
isMultiLevel() setMultiLevel()	検索で StartFolder 内のサブフォルダを横断するかどうかを決定します。FALSE の場合、検索は StartFolder のみに限定され、StartFolder 内のサブフォルダは横断しません。TRUE の場合、検索はすべてのサブフォルダを横断します。
getSearchClassname() setSearchClassname()	フォルダ内で検索する情報のクラスを決定します。Search クラスは、FolderRestrictQualification とともに SearchSpecification に自動的に含まれます。

ContextQualification

ContextQualification は、ドキュメントの内容に関する条件を表します。

例 3-5 内容ベースの問合せ

```
"find all documents containing the word 'XML' "
```

表 3-54 に、ContextQualification クラスで最も使用頻度の高い定数とメソッドを示します。

表 3-54 ContextQualification の定数とメソッド

getName() setName()	ContextQualification の Name を決定するために使用されます。名前を使用して、Oracle Text でソート条件として生成される関連性スコアを組み込むことができます。
ORDER_PREFIX	Oracle Text で生成される関連性スコアを SearchSortSpecification に組み込む場合に使用します。定数は ContextQualification の Name に接頭辞として使用され、Oracle 9*i*FS により ORDER BY 句に正しい SQL 構文を生成するために使用されます。
getQuery() setQuery()	この 2 つのメソッドは、Oracle Text に内容検索の条件として渡されるテキスト問合せ文字列を決定するために使用されます。問合せ文字列は、Oracle 9*i*FS から CONTAINS() SQL ファンクションのテキスト文字列引数にそのまま渡され、このファンクションにより Oracle Text がコールされ、内容の問合せが実行されます。問合せ文字列は Oracle 9*i*FS で変更されないため、Oracle Text で要求される構文に従って、必要な数のトークン、式演算子および結合演算子を含めることができます。[1]

[1] Oracle Text に対するテキスト問合せ文字列の構文規則の詳細は、『Oracle Text リファレンス』を参照してください。

ExistenceQualification

ExistenceQualification は、属性値が可能な値のリストの 1 つと一致するという条件、または属性値が別のオブジェクトの属性の値と一致するという条件を表します。

例 3-6 存在の問合せ

```
"find all reports where the report number is one of the following: 3a, 3b, 6a"

"find all documents where the name of a document matches the name of a folder"
```

表 3-55 に、ExistenceQualification クラスで最も使用頻度の高いメソッドを示します。

表 3-55 ExistenceQualification のメソッド

getLeftAttributeName() getLeftAttributeClassname() setLeftAttribute()	条件の左辺で、値が条件の右辺に指定された値の 1 つと一致する必要のある Attribute を決定します。

表 3-55 ExistenceQualification のメソッド（続く）

getRightAttributeName() getRightAttributeClassname() setRightAttribute()	Attribute を経由して条件の右辺の値のリストを決定します。
getRightAttributeValue() isRightAttributeValue() setRightAttributeValue()	AttributeValue[ ] 配列を経由して条件の右辺の値のリストを決定します。

PropertyQualification

PropertyQualification は、情報の PropertyBundle 内のプロパティに関する条件を表します。

例 3-7 プロパティ・ベースの問合せ

```
" find all documents where the italian_title_translation property = 'Vino di Italia'
"
```

Oracle 9*i*FS 内の情報に関するメタデータは、属性として格納するのみでなく、PropertyBundle に格納することもできます。PropertyBundle は、それぞれが名前 / 値のペアで構成される複数の Property のまとまりです。PropertyBundle は、ハッシュ表のように最も簡単に操作できる非定型のメタデータを格納するために使用されます。

表 3-56 に、PropertyQualification クラスで最も使用頻度の高いメソッドを示します。

表 3-56 PropertyQualification のメソッド

getPropertyName() setPropertyName()	検索対象となる Property の名前を決定します。
getOperatorType() setOperatorType()	Property と指定の Value との比較に使用する演算子を決定します。このメソッドは、AttributeQualification に定義された定数（EQUAL など）を受け取って、条件内で比較演算子を表します。現在、Oracle 9*i*FS でサポートされている演算子は、文字列に対する 'IS_NULL'、'IS_NOT_NULL' および 'LIKE' のみです。
getValue() setValue()	Attribute の比較対象となる値を決定します。演算子 LIKE を使用すると、有効な SQL ワイルドカード（% や _ など）を値に使用できます。

表 3-56 PropertyQualification のメソッド（続く）

getDateComparisonLevel() setDateComparisonLevel()	DATE 型の Property の場合、これらのメソッドは、Oracle 9*i*FS で属性と日付値を比較するレベルを指定するために使用されます。このメソッドは、AttributeQualification に定義された定数（DATE_COMP_DAY など）を受け取って、条件内で様々な比較レベルを表します。 たとえば、PropertyQualification を使用すると、'PublicationDate' Property を、'March 21, 2001 08:30 am'（'PublicationDate < SYSDATE' など）に設定されているシステム生成の日付 'SYSDATE' と比較できます。ただし、2001 年 3 月 20 日以前、つまり 2001 年 3 月 20 日 23:59 p.m. 以前に作成されたすべての情報を取り出す必要がある場合があります。AttributeQualification で 2001 年 3 月 21 日 7:00 a.m. に作成された情報が戻されるのを防ぐには、日付の比較レベルを DATE_COMP_DAY に設定します。
isCaseIgnored() setCaseIgnored()	条件に大 / 小文字区別が必要かどうかを決定します。 たとえば、IgnoreCase を 'true' に設定すると、条件 NAME = 'XML' では NAME が 'xml' の情報は戻されません。
isLateBound() getLateBoundDataType() setLateBoundDataType()	Property の値が遅延バインド変数で与えられるように指定します。
setClassname() getClassname()	PropertyBundle を持つ情報のクラスを決定します。Search クラスは、PropertyQualification とともに SearchSpecification に自動的に含まれます。

FreeFormQualification

FreeFormQualification を使用すると、SQL 構文の自由形式の文字列を通じて他のあらゆるタイプの条件を組み込むことができます。FreeFormQualification で指定した SQL 構文は、Oracle 9*i*FS により SearchSpecification から生成される SQL SELECT 文に挿入され、解析解除されます。

表 3-57 に、FreeFormQualification クラスで最も使用頻度の高いメソッドを示します。

表 3-57 FreeFormQualification のメソッド

getSQLExpression() setSQLExpression()	SQL 式を含む String を決定します。

JoinQualification

検索条件に複数の条件を指定する場合は、JoinQualification および SearchClause を使用して条件の結合方法を制御できます。JoinQualification は、関連するオブジェクト情報に関する条件を含める場合の結合の構成に使用します。

例 3-8 結合の問合せ

```
"find all documents where owner's name is 'wtalman' "
```

この検索では、ドキュメントのメタデータをドキュメントの所有者に関するメタデータと結合するように要求しています。SQL では、これが次のように表されます。

```
SELECT *
FROM Document d, DirectoryUser u
WHERE d.owner = u.id AND u.name = 'wtalman'
```

この場合は、JoinQualification を使用して、ドキュメント情報をユーザー情報と結合する方法（d.owner = u.id など）を指定できます。次に、AttributeQualification を作成して、ユーザーのメタデータに関する条件（u.name = 'wtalman' など）を指定します。

表 3-58 に、JoinQualification クラスで最も使用頻度の高いメソッドを示します。

表 3-58 JoinQualification のメソッド

getLeftAttributeName() getRightAttributeName() setLeftAttribute() setRightAttribute()	結合条件の左辺と右辺を決定します。
getLeftAttributeClassname() getRightAttributeClassname()	結合条件に指定したどちらかの Attribute のクラスを取得します。

SearchClause

SearchClause は、演算子を使用して条件をより複雑な条件に結合する、SQL WHERE 文の句を表します。

演算子と優先順位の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

ブール条件

SearchClause は、2 つの条件を演算子 AND および OR で結合してブール条件にするために使用されます。

例 3-9 ブール検索句

```
"find all documents where name = 'User Guide' AND modification_date > 'January 01,
2001' "
```

所有者と変更日の属性に関する条件を表す 2 つの AttributeQualification を作成した後に、SearchClause を作成して、結合に使用した演算子で 2 つの AttributeQualification をアセンブルします。

否定条件

SearchClause は、演算子 NOT を使用して単一条件に関する否定条件を指定する場合にも使用されます。

例 3-10 NOT 検索句

```
"find all documents which are NOT in my home folder "
```

my home folder の FolderRestrictQualification を作成した後に、SearchClause を作成して条件の前に NOT 演算子を付けます。

SearchClause を相互にネストすると、3 つ以上の条件を結合できます。

例 3-11 ネストした検索句

```
"find all documents where name = 'User Guide' OR description = 'User Guide' AND
modification_date > 'January 01, 2001'
```

それぞれ条件を表す 3 つの AttributeQualification を作成した後に、SearchClause を使用して 2 つの AttributeQualification を結合します。2 番目の SearchClause は、3 番目の AttributeQualification を最初の SearchClause と結合するために使用されます。この方法で SearchClause を使用して、SearchQualification 内ですべての条件（AttributeQualification、FolderRestrictQualification など）をアセンブルします。

表 3-59 に、SearchClause クラスで最も使用頻度の高い定数とメソッドを示します。

表 3-59 SearchClause の定数とメソッド

getLeftSearchQualification() setLeftSearchQualification()	SearchClause の左辺の条件を決定します。

表 3-59 SearchClause の定数とメソッド（続く）

getOperatorType() setOperatorType() AND OR NOT	SearchClause で条件の結合に使用する演算子を決定します。 これらの定数は、句の演算子を表すために使用します。
getRightSearchQualification() setRightSearchQualification()	SearchClause の右辺の条件を決定します。

SearchSortSpecification

SearchSortQualification は、検索結果のソート順序を指定します。Oracle 9*i*FS では、SQL SELECT 文の 'ORDER BY' 句を構成するために使用されます。

表 3-60 に、SearchSortQualification クラスで最も使用頻度の高い定数とメソッドを示します。

表 3-60 SearchSortQualification の定数とメソッド

add()	SearchSortSpecification に 1 つ以上のソート条件を追加します。
getClassnames()	検索結果の順序付けに使用される属性のクラスを決定します。
getAttributesNames()	検索結果の順序付けに使用される属性を決定します。
getOrders() ASCENDING DESCENDING	属性に基づき、結果の順序として昇順または降順を決定します。 これらの定数は、SearchSortSpecification に指定された順序を表すために使用します。

Search

Oracle 9*i*FS の検索オブジェクト・モデルには、問合せの実行用にもう 1 種類のオブジェクトである Search が用意されています。SearchSpecification は、アセンブル後に新規 Search の作成に使用されます。次に、Search が次の操作に使用されます。

- 検索条件の遅延バインド変数を渡します。
- 検索に使用する言語を指定します。
- 問合せを実行します。

- 結果を取得します。

Search を実行すると、Oracle 9*i*FS では SQL SELECT 文が構成され、データベース・スキーマに問い合せる複雑さが処理されます。これにより、情報検索をオブジェクト指向の方法で構成でき、データベース内の様々な表に情報がどのように格納されるかを知る必要はありません。SQL SELECT 文の一部を明示的に指定するには、FreeFormQualification オブジェクトを使用します。

表 3-61 に、Search クラスで最も使用頻度の高いメソッドを示します。

表 3-61 Search の属性およびメソッド

getSearchSpecification() setSearchSpecification()	Search の問合せ条件を保持する SearchSpecification を決定します。
open()	遅延バインド変数を渡し、問合せを実行し、検索結果へのカーソルをオープンします。
close()	検索のカーソルをクローズします。
next() getItemCount()	検索結果を取得します。

SearchResultsObject

Search を実行すると、Oracle 9*i*FS は検索条件と一致する個々の情報について、ヒットした結果の配列を戻します。ヒットした結果は、SearchSpecification に指定したソート条件に従って配列内でソートされます。また、それぞれ配列内で SearchResultObject で表されます。

SearchResultObject を使用すると、ヒットした結果がある実際の情報オブジェクトを取り出すことができます。この情報オブジェクトは LibraryObject として表されます。LibraryObject は、リポジトリに格納される情報の各部を表すために使用されます。戻される LibraryObject は、SearchClassQualification に指定されたコンテンツ・タイプの 1 つである必要があります。各 LibraryObject は、情報のコンテンツ・タイプに関連する属性と内容で構成されます。

SearchResultObject から LibraryObject を取り出すと、検索アプリケーションでは情報の属性と内容を表示して操作でき、リポジトリから情報を取り出すために第 2 のコールを行う必要はありません。ただし、検索アプリケーションで関連情報（親フォルダの属性など）の表示または操作が必要な場合は、第 2 の操作を実行し、リポジトリから関連情報を取り出す必要があります。

表 3-62 に、SearchResultObject クラスで最も使用頻度の高いメソッドを示します。

表 3-62 SearchResultObject のメソッド

getLibraryObject()	検索でヒットした結果、つまり SearchResultObject がある LibraryObject を戻します。

情報の処理

Oracle 9*i*FS には、リポジトリ内の情報についてサーバー側の処理を自動化できるように、3 つのクラス・パッケージが用意されています。

- `oracle.ifs.beans.parsers` パッケージ
- `oracle.ifs.server.renderers` パッケージ
- `oracle.ifs.management.domain` パッケージ

パーサー

`oracle.ifs.beans.parsers` パッケージには、Oracle 9*i*FS にインポートされる情報を前処理するパーサーの構築用に、インタフェース・セットが用意されています。また、解析機能を提供するインタフェースを実装するクラス・セットも組み込まれています。このインタフェースを実装するか、このパッケージ内のクラスを拡張して、カスタム・パーサーを構築できます。

たとえば、IfsSimpleXmlParser クラスは、Oracle 9*i*FS に含まれている XML ファイルの解析用パーサーを構成します。IfsSimpleXmlParser は、XML ファイルの入力ストリームを受け取って解析し、そのコンポーネントの DOM ツリーを生成してから、XML ファイルのスキーマに対応するコンテンツ・タイプのインスタンスを作成します。

パーサーの実装手順は、第 9 章「カスタム・パーサーの作成」を参照してください。

レンダラ

`oracle.ifs.server.renderers` パッケージには、パーサーとは逆に、Oracle 9*i*FS からエクスポートされる情報を前処理するためのインタフェースとクラスが用意されています。カスタム・レンダラを構築するには、そのインタフェースを実装する方法と、クラスを拡張する方法があります。

たとえば、SimpleXmlRenderer クラスでは、Oracle 9*i*FS に XML レンダリング機能を提供する Renderer インタフェースが実装されます。SimpleXmlRenderer は、コンテンツ・タイプのインスタンスを受け取って XML ファイルに変換します。

レンダラの実装手順は、第 11 章「カスタム・レンダラの作成」を参照してください。

サーバー

`oracle.ifs.management.domain` パッケージには、サーバーを構築するためのインタフェースとクラスのセットが用意されています。サーバーにより、Oracle 9*i*FS で実行されるタスクが自動化されます。サーバーでは、これらのタスクを特定の時刻に開始するか、Oracle 9*i*FS で特定のイベントが発生した時点で開始できます。

このパッケージ内の抽象クラス IfsServer は、Oracle 9*i*FS のすべてのサーバーの基本クラスとして機能します。IfsServer クラスは Server インタフェースを実装するため、このクラスを拡張するサーバーは Oracle 9*i*FS Server Manager で管理できます。

サーバーの実装手順は、第 13 章「カスタム・サーバーの作成」を参照してください。

4

Oracle Internet File System ドキュメントの作成

この章では、用意されている Document クラスをプログラムで処理して、汎用 Document オブジェクトを作成する方法について説明します。標準 Document のインスタンスの作成プロセスを理解すると、それに基づいてカスタマイズの概念を構築できます。

この章の内容は、次のとおりです。

- Oracle 9iFS リポジトリでのファイルの格納方法の概要
- Document オブジェクトの作成手順の概要
- Document オブジェクトの作成
- フォルダへのドキュメントの追加
- アプリケーション全体の検討

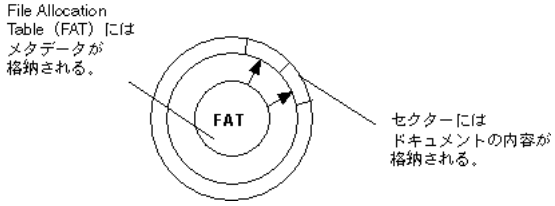
Oracle 9*i*FS リポジトリでのファイルの格納方法の概要

この項では、Oracle9*i* データベースでファイル情報を編成して格納する方法の概念の概要を説明します。詳細の一部は簡略化されています。

ファイル・システムへのファイルの格納方法の概要

標準的なファイル・システムでは、ファイルは 2 つの部分に格納されます。ここでは、一般的に普及している特定の実装、つまり、FAT とも呼ばれる DOS ファイル・システムについて説明します。メタデータ、つまりファイル属性は、File Allocation Table（FAT）に格納されます。メタデータには、ファイル名、パス、作成日および変更日などがあります。ドキュメントの内容は、セクターと呼ばれるディスクの論理区画に格納されます。FAT では、ファイルのうち、再構成時にドキュメントの本体を構成するセクターの位置が追跡されます。図 4-1 に、ローカル・ハード・ディスクに格納されているファイルを簡略化したものを示します。

図 4-1 ハード・ディスクに格納されているドキュメント



Oracle 9*i*FS リポジトリでのファイルの格納方法の概要

Oracle 9*i*FS では、メタデータとドキュメントの内容が別個に格納されます。メタデータは Oracle 9*i*FS スキーマの複数の表に格納され、内容は表 `ODMM_CONTENTSTORE` にラージ・オブジェクト（LOB）として格納されます。ドキュメントの一意 ID 番号は、複数の表に格納されている情報を結合して Document オブジェクトを構成するために使用される外部キーです。図 4-2 に、Oracle 9*i*FS リポジトリでのドキュメントの格納方法を示す概略図を示します。

図 4-2 Oracle 9*i***FS リポジトリに格納されているドキュメント**

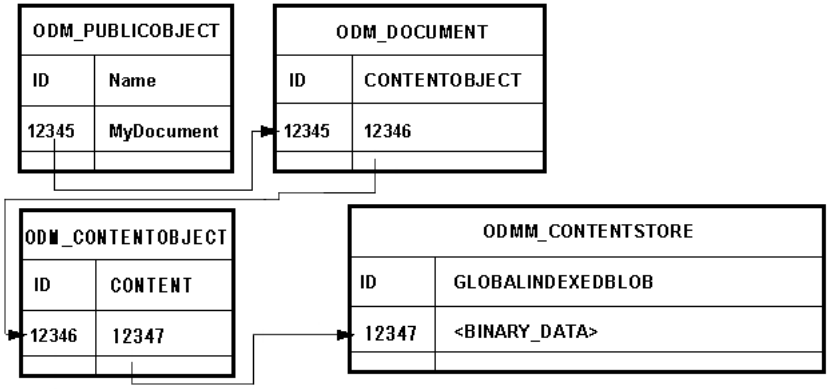


表 4-1 に、Document のメタデータの格納に使用されるスキーマの表を示します。

表 4-1 Document のメタデータの格納に使用される表

表	説明
ODM_ PUBLICOBJECT	この表には、フォルダに表示でき、エンド・ユーザーが直接変更できるファイルに共通する基本情報が格納されます。名前、最終変更日、作成者、所有者などの属性が格納されます。
ODM_ DOCUMENT	PublicObject の Document サブクラスのみが、2 つの追加属性を必要とします。一方は ReadByOwner で、ファイルが読み取られたことを示す論理フラグです（主として電子メール・ドキュメントに使用されます）。他方は ContentObject で、表 ODMM_CONTENTSTORE に格納されるドキュメント本体の ID 番号です。
ODM_ CONTENTOBJECT	この表には、ドキュメントの形式と言語に関する情報が格納されています。
ODMM_ CONTENTSTORE	この表には、ドキュメントの形式と言語に関する情報が格納され、ドキュメント本体はバイナリ・ラージ・オブジェクト（Binary Large Object: BLOB）として格納されます。（実際には、内容の格納に使用される表は 2 つあり、一方は索引付きデータ用の ODMM_CONTENTSTORE で、他方は索引なしのデータ用の ODMM_NONINDEXEDSTORE です。）

表 `ODM_PUBLICOBJECT` および `ODM_DOCUMENT` は実際に FAT の役割を果たし、表 `ODM_CONTENTOBJECT` はドキュメントに関する情報、表 `ODMM_CONTENTSTORE` はドキュメントの本体の格納に使用されます。

この方法でドキュメントを格納すると、ドキュメントの内容に索引を付けて、リレーショナル・データベースのフィールドとして問合せできます。索引付きデータがドキュメントとして取り扱われる時点（ファイルが Oracle 9*i*FS からローカル・ディスクにコピーされる場合など）で、メタデータがローカル・ファイル・システムに送られ、ドキュメントのコンテンツ・オブジェクトがバイト・ストリームとして送られて、ドキュメントが再構成されます。

この処理はすべてエンド・ユーザーに対して透過的であり、データ操作によるオーバーヘッドの対部分はアプリケーション開発者のために Oracle 9*i*FS で処理されます。この概要は、システムでのドキュメント作成プロセスを理解しやすいように、画面の裏側における情報の処理方法の概要を示すことを意図しています。

Document オブジェクトの作成手順の概要

Oracle 9*i*FS でのドキュメントの格納方法を把握したところで、Oracle 9*i*FS リポジトリにドキュメントをプログラムでロードする方法について説明します。Oracle 9*i*FS でドキュメントを作成する手順は、実際には次の 4 つに分かれています。

1. Oracle 9*i*FS サーバーへの接続を確立します。
2. 属性と内容を持つドキュメントのランタイム・バージョンである DocumentDefinition を作成します。
3. DocumentDefinition に基づいて、リポジトリに永続的な Document オブジェクトを作成します。
4. 新しい Document をフォルダに入れます。

リポジトリへの接続

Oracle 9*i*FS リポジトリ内で操作を行う前に、アプリケーションで Oracle 9*i*FS サーバーとの接続、LibrarySession を確立する必要があります。最初のステップは、実際には接続先サーバーである LibraryService を表すオブジェクトをインスタンス化することです。次に、LibraryService に有効な資格証明のセットを渡すと、LibraryService から新しい LibrarySession が戻されます。LibrarySession は、現行の作業セッションに関する有効な情報にアクセスするための便利なクラスで、Oracle 9*i*FS リポジトリ内で情報を変更するほとんどのコマンドでパラメータとして使用されます。

例 4-1 Oracle 9*i*FS サーバーへの接続の確立

次に示すのは、Oracle 9*i*FS アプリケーションの骨格部分です。このプログラムでは、接続が確立されてから切断されます。このクラスのコンストラクタは、4 つの引数からなる配列を受け入れます。

```
package oracle.ifs.examples.devdoc.connection;
import oracle.ifs.beans.DirectoryUser;
import oracle.ifs.beans.LibraryService;
import oracle.ifs.beans.LibrarySession;
```

```
import oracle.ifs.common.CleartextCredential;
import oracle.ifs.common.IfsException;

public class SimpleConnection
{
  public static void main(String[] args) throws IfsException
  {
    try {
      // Connect to the 9iFS server. The arguments (args) are
      // IfsService, IfsServicePassword, UserName, UserPassword
      // The default values are ifsDefault, ifssys, system, manager9ifs
      System.out.println("Connecting...");

      // Use the IfsService and IfsServicePassword to start a LibraryService
      LibraryService service = LibraryService.startService(args[0], args[1]);

      // Create a CleartextCredential object, which encapsulates the
      // username and password.
      CleartextCredential cred = new CleartextCredential(args[2], args[3]);

      // Start a LibrarySession by passing the credentials to the service
      LibrarySession ifsSession = service.connect(cred, null);

      /* This is the place where you add code that actually does something.
       * To show that we're connected, we'll query the repository for
       * the name of the user who has logged in.
      */

      DirectoryUser du = ifsSession.getDirectoryUser();
      System.out.println("Current user: " + du.getName());

      // Disconnect from the 9iFS LibrarySession.
      ifsSession.disconnect();
      System.out.println("Disconnected.");
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

Document オブジェクトの作成

Oracle 9*i*FS では、オブジェクトの作成ステップが 2 段階に分かれています。最初のステップは、作成するオブジェクトの属性を定義するオブジェクト定義を作成することです。2 番目のステップでは、サーバー上でオブジェクトをインスタンス化します。これにより、Oracle 9*i*FS リポジトリでは、個々のトランザクションで各属性を設定するのではなく、単一トランザクションの結果としてオブジェクトを作成できます。

DocumentDefinition オブジェクトの作成

DocumentDefinition クラスのコンストラクタは、パラメータとして現行の LibrarySession のみを必要とします。

たとえば、次のコード行は、前述の例で作成された LibrarySession session を使用して、newDocDef という新規の DocumentDefinition をインスタンス化しています。

```
DocumentDefinition newDocDef = new DocumentDefinition(session);
```

DocumentDefinition の設定

DocumentDefinition の作成後のステップは、その属性を移入することです。ただし、Java で使用されるデータ型には、Oracle9*i* データベースで使用されるデータ型との互換性が常にあるとはかぎらないため、軽度な変換の問題があります。Oracle 9*i*FS のオブジェクトが他のオブジェクトの属性として使用される場合があります。たとえば、Document の所有者は DirectoryUser オブジェクトの ID で識別されます。情報をフォーマットして Oracle 9*i*FS リポジトリの特殊なニーズを処理するために、すべての属性はまず AttributeValue オブジェクトとしてインスタンス化されます。AttributeValue オブジェクトはハンドラで、Java の値を Oracle 9*i*FS リポジトリに挿入できるようにフォーマットし、リポジトリに格納されている値を Java プログラムに戻す前にフォーマットします。

AttributeValue クラスには、各データ型を Oracle 9*i*FS に格納できる単一値または配列値として受け入れるように、オーバーロードされているコンストラクタがあります。Java アプリケーションから DocumentDefinition に値を挿入するには、最初にそれを AttributeValue としてインスタンス化してから、setAttribute() メソッドの引数として使用します。

AttributeValue オブジェクトに使用可能なデータ型の詳細は、AttributeValue クラスの Javadoc を参照してください。

この例で設定するのは、ファイル名とその内容のみです。Name 属性には独自のメソッド setName() があり、値の設定に使用することができます。これは、そのファイルが Oracle 9*i*FS フォルダにリストされる時に表示されるファイル名です。

Document オブジェクトの内容の設定は、異なる方法で処理されます。内容はメタデータではないため、属性データ格納用の表には格納されません。`DocumentDefinition()` クラスには、内容を指定できるように複数のメソッドが用意されています。最も単純なケースは、String を `setContent()` メソッドに直接渡すことですが、より一般的な方法は `setContentPath()` メソッドを使用することです。このメソッドは、新規 Document オブ

ジェクトが内容を取り込むローカル・ディスク上のファイルへのパスを、引数として取ります。

例 4-2 DocumentDefinition での値の設定

次のコード・フラグメントに、DocumentDefinition の作成プロセス、つまり、新規 DocumentDefinition をインスタンス化し、setName および setContent メソッドを使用して、Document の Name および Content 属性をそれぞれ設定するプロセスを示します。

```
// Instantiate an empty DocumentDefinition object.
DocumentDefinition newDocDef = new DocumentDefinition(ifsSession);

// Set the Name attribute for the new document.
newDocDef.setName("HelloWorld.txt");

// Set the content of the document.
newDocDef.setContent("Hello, world.");
```

Document の作成

DocumentDefinition の作成後は、`LibrarySession.createPublicObject()` メソッドを使用して、Document オブジェクト自体を単一トランザクションで作成できます。すべての PublicObject は、このメソッドを使用して作成されます。このメソッドに DocumentDefinition を渡すと、Oracle 9*i*FS API では作成対象となる特定タイプの PublicObject が Document であることが認識されます。

例 4-3 永続的な Document オブジェクトの作成

次のコード行では、例 4-2 で作成した DocumentDefinition に基づいて永続ドキュメントが作成されます。作成されたドキュメントは、プログラムで参照してさらに処理できるように、ランタイムの Document オブジェクトとしてインスタンス化されます。

```
Document doc = (Document) ifsSession.createPublicObject(newDocDef);
```

フォルダへのドキュメントの追加

デフォルトでは、PublicObject とそのサブクラスの新規インスタンスは、フォルダなしで作成されます。ユーザーがディレクトリ・インタフェースを使用して Document に移動できるように、新規ドキュメントは作成時にフォルダに入れる必要があります。

ファイル・システムでのドキュメントとフォルダの関係の表現の概要

標準的なファイル・システムでは、ドキュメントのパスは FAT 内でその定義に埋め込まれています。たとえば、ファイル `myDoc.txt` がディレクトリ `myFolder` に格納されている場合、FAT 内のエントリは次のようになります。

```
¥myFolder¥myDoc.txt
```

ドキュメントの特定のインスタンスを参照するには、埋め込まれたパスを経由する必要があります。ドキュメントの複数のコピーを他のディレクトリに置くこともできますが、それは別個のドキュメントであり、オリジナルが更新されてもコピーは更新されません。

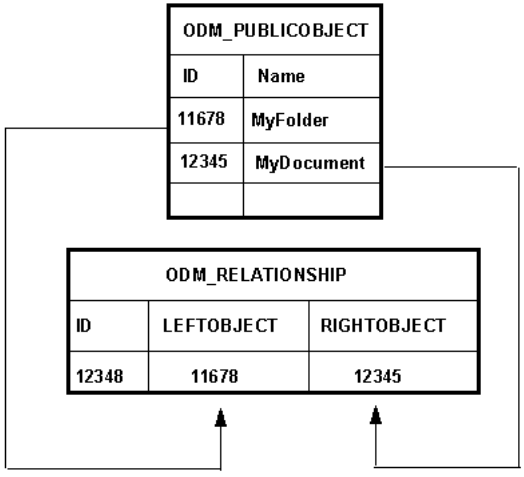
例えば Microsoft Windows では、ドキュメントへのショートカットを作成できますが、ショートカット自体はファイルの位置を指すためにのみ使用されるドキュメントです。ファイルが削除されてもショートカットは残りますが、何も指さなくなります。

Oracle 9*i*FS でのドキュメントとフォルダの関係の表現の概要

Oracle 9*i*FS では、フォルダは PublicObject です。実際には、Folder サブクラスは、PublicObject クラスが提供する属性のみを継承します。

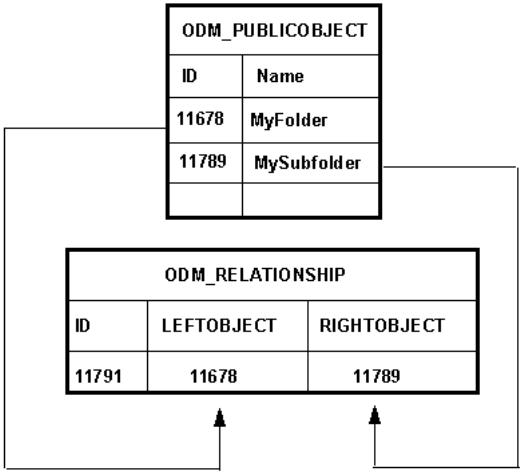
ドキュメントとサブフォルダは、Relationship オブジェクト（より限定的には、Relationship クラスから必要な属性を継承する FolderPathRelationship）を作成することで、Folder オブジェクトに対応付けられます。Folder ID は Relationship レコードに LEFTOBJECT として格納され、Document ID は RIGHTOBJECT として格納されます。

図 4-3 フォルダ / ドキュメントの関係



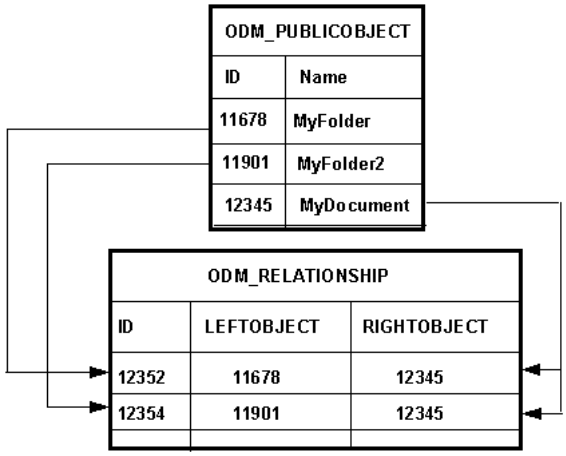
Folder は、同じ方法で他の Folder オブジェクトと対応付けられます。親の ID は LEFTOBJECT として格納され、子の ID は RIGHTOBJECT 列に格納されます。

図 4-4 フォルダ / サブフォルダの関係



これにより、エンド・ユーザーは大きな柔軟性が得られます。同じドキュメントを複数のフォルダに表示できますが、Oracle 9*i*FS に格納されるのは 1 回のみです。すべてのフォルダは同じオブジェクトを指しているため、1 箇所でドキュメントが更新されると、すべての箇所で更新されます。親フォルダの 1 つからファイルが削除されても、システムには残り、他のすべての親フォルダには表示されます。ドキュメントが Oracle 9*i*FS リポジトリから削除されるのは、そのドキュメントの全インスタンスがすべての親フォルダから削除された場合のみです。

図 4-5 複数の親フォルダを持つドキュメント



Document と Folder の対応付け

Document を Folder に対応付けるには、永続的な Folder オブジェクトをランタイム Java オブジェクトとしてインスタンス化し、`Folder.addItem()` メソッドを使用して Folder に Document オブジェクトを追加します。

例 4-4 フォルダへのドキュメントの追加

次のコード・フラグメントでは、例 4-3 で作成した Document オブジェクトをユーザーの home フォルダに追加しています。

```
DirectoryUser thisUser = ifsSession.getDirectoryUser();
PrimaryUserProfile userProfile =
  ifsSession.getPrimaryUserProfile(thisUser);
Folder homeFolder = userProfile.getHomeFolder();
homeFolder.addItem(doc);
```

アプリケーション全体の検討

この章のサンプル・コードでは、Oracle 9*i*FS リポジトリで Document オブジェクトを作成するための各ステップを示しました。例 4-5「HelloWorld.java」に、アプリケーション全体を示します。

例 4-5 HelloWorld.java

```
package oracle.ifs.examples.devdoc.helloworld;
// Class used to represent the current user at runtime
import oracle.ifs.beans.DirectoryUser;

// Class used to instantiate the Document object at runtime
import oracle.ifs.beans.Document;

//Class used to define the attributes of the new Document object
import oracle.ifs.beans.DocumentDefinition;

//Class used to represent a Folder object at runtime
import oracle.ifs.beans.Folder;

// Classes used to create the connection to the Oracle 9iFS server
import oracle.ifs.beans.LibraryService;
import oracle.ifs.beans.LibrarySession;

// Class used to access information about the current user; in this
//  program, it's used to access the user's home folder.
import oracle.ifs.beans.PrimaryUserProfile;

// Class used to create the persistent Document object in the repository
import oracle.ifs.beans.PublicObject;

// Class used to hold the user's authentication information
import oracle.ifs.common.CleartextCredential;

// Class used to trap and report 9iFS specific exceptions
import oracle.ifs.common.IfsException;

/*
 * HelloWorld class. This class demonstrates the steps needed to
 * connect to an instance of Oracle9iFS, create a new document,
 * insert the document in a folder, and disconnect from the Oracle
 * 9iFS server.
 */

public class HelloWorld
{
```

```
/* Connect to the Oracle 9iFS server. The arguments are IfsService,
 * IfsServicePassword, UserName, UserPassword.
 * Default syntax to run the program is:
 * java oracle.ifs.examples.devdoc.helloworld.HelloWorld ifsDefault
 *   ifssys system manager9ifs
 */
public static void main(String[] args) throws IfsException
{
  /* As a rule, you should always use VerboseMessages in your 9iFS classes
   * because it will result in better error message output to facilitate
   * debugging. Set the value to true at the start of each of your programs.
   */
  IfsException.setVerboseMessage(true);
  try
  {
    // Start the LibraryService.
    LibraryService service = LibraryService.startService(args[0], args[1]);

    // Create a CleartextCredential object that encapsulates the user's
    //   log in information.
    CleartextCredential cred = new CleartextCredential(args[2], args[3]);

    // Connect (create a LibrarySession) using the Service and
    //   CleartextCredential objects.
    LibrarySession ifsSession = service.connect(cred, null);

    // Instantiate an empty DocumentDefinition object.
    DocumentDefinition newDocDef = new DocumentDefinition(ifsSession);

    // Set the Name attribute for the new document.
    newDocDef.setName("HelloWorld.txt");

    // Set the content of the document.
    newDocDef.setContent("Hello, world.");

    // Create the document object in the Oracle 9iFS repository.
    Document doc = (Document) ifsSession.createPublicObject(newDocDef);

    // Begin the foldering process. First, instantiate a DirectoryUser
    //  object that represents the current user.
    DirectoryUser thisUser = ifsSession.getDirectoryUser();

    // Get the PrimaryUserProfile information for the current user.
    PrimaryUserProfile userProfile =
      ifsSession.getPrimaryUserProfile(thisUser);

    // Instantiate a runtime representation of the user's home folder.
```

```
      Folder homeFolder = userProfile.getHomeFolder();

      // Insert the new document to the user's home folder.
      homeFolder.addItem(doc);

      // Close the connection to Oracle 9iFS.
      ifsSession.disconnect();

    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

HelloWorld.java アプリケーションの実行

HelloWorld.java アプリケーションを実行する手順は、次のとおりです。

1. 第 1 章「Oracle Internet File System SDK スタート・ガイド」の説明に従って、開発環境のセットアップを完了したことを確認します。
2. `HelloWorld.java` をコンパイルします。
3. コンパイル後のパッケージ構造 `/oracle/ifs/examples/devdoc/helloworld/HelloWorld.class` を `$IFS_HOME/custom_classes` ディレクトリに置きます。
4. コマンドラインから次のコマンドを入力し、このクラスを実行します。

   ```
   java oracle.ifs.examples.devdoc.helloWorld.HelloWorld
   <serviceName> <schemaPassword> <userName> <userPassword>
   ```

   serviceName のデフォルトは IfsDefault で、schemaPassword は ifssys です。userName に `system`、userPassword に `manager9ifs` を使用すると、system ユーザーのホーム・フォルダに新規ドキュメントを作成できます。
5. ドキュメント `HelloWorld.txt` が、手順 4 で userName および userPassword 引数で識別した Oracle 9*i*FS ユーザーのホーム・フォルダに表示されます。

5

コンテンツ・タイプと属性の拡張

この章では、Oracle 9*i*FS のコンテンツ・タイプの階層を拡張し、独自の型を持つ情報を管理する方法について説明します。この章の内容は、次のとおりです。

- コンテンツ・タイプと属性の拡張の概要
- カスタム・コンテンツ・タイプおよび属性の実装
- サンプル・コード

コンテンツ・タイプと属性の拡張の概要

Oracle Internet File System（Oracle 9*i*FS）でコンテンツ管理アプリケーションを構築する場合に実行する最も一般的なタスクは、独自の型を持つ情報を管理するように Oracle9*i* を構成することです。Oracle 9*i*FS は、ドキュメント、フォルダ、ユーザーおよびグループなど、一般的なタイプの情報を管理するように事前に構成されています。Oracle 9*i*FS SDK には、Oracle 9*i*FS で管理される各タイプの情報の特性を定義するコンテンツ・タイプ階層が含まれています。このコンテンツ・タイプ階層を簡単に拡張し、医療記録、保険料請求および設計仕様など、他のタイプの情報について、Oracle 9*i*FS での管理方法をカスタマイズできます。

注意： コンテンツ・タイプ階層の詳細と、SDK を使用してリポジトリ内の情報を管理する方法は、第 3 章「Java API の概要」の「情報の管理」を参照してください。

カスタム・コンテンツ・タイプを実装する前に、少し時間をかけてコンテンツ・タイプ階層を設計することが重要です。予想できる各種情報の管理方法について計画を立ててください。各コンテンツ・タイプの特性と、コンテンツ・タイプを階層形式で編成する方法を定義します。この作業を行うと、より一貫して管理しやすいコンテンツ・タイプ階層を実装できます。この作業を省いた場合は、計画外の情報のタイプに対応するために、後でコンテンツ・タイプ階層を変更するのが困難であることがわかります。属性を追加したり削除してコンテンツ・タイプ階層を変更することもできますが、現在、Oracle 9*i*FS では、階層内のコンテンツ・タイプの再編成（変更など）や、属性のデータ型または制約の変更は許されていません。この項では、コンテンツ・タイプ階層の設計方法について説明します。

カスタム・コンテンツ・タイプの特性の定義

Oracle 9*i*FS のコンテンツ・タイプをカスタマイズする最初のステップは、管理する情報の特性を定義することです。ビジネスで Oracle 9*i*FS に格納して管理する必要のある情報について、例をすべて収集します。次に、共通する特性に従ってソートし、コンテンツ・タイプとしてグループ化します。次の特性を考慮してください。

- **属性：** そのコンテンツ・タイプのすべてのインスタンスの構造の一部として格納されるメタデータです。たとえば、Image コンテンツ・タイプには、幅、高さおよび色相など、イメージの色または空間的な側面を記述する属性を持たせることができます。
- **動作：** そのタイプの情報の操作に必要なメソッドです。たとえば、Image コンテンツ・タイプには、`generateThumbnail()` など、イメージを操作する特殊なメソッドが必要な場合があります。
- **任意のメタデータ：** 様々なタイプの情報の特定インスタンスに任意に適用できるメタデータです。たとえば、ドキュメント、フォルダおよびイメージを、ビジネスで生成する製品との関連性に従ってカテゴリ別に分類できます。また、ドキュメント、イメージおよびフォルダを FTP クライアントに表示する方法についてポリシーを定義できます。

- **ファイル拡張子：** 特定のコンテンツ・タイプのインスタンスとして格納する必要があるファイルの拡張子です。たとえば、すべての .jpg、.bmp および .gif ファイルを、Image コンテンツ・タイプのインスタンスとして格納するように指定できます。
- **パーサー：** 特定のコンテンツ・タイプのインスタンスを作成するためのファイルの解析に使用されるパーサーです。たとえば、すべての .jpg ファイルを最初にカスタム・パーサーに渡して、Image コンテンツ・タイプの属性値を抽出するように指定できます。
- **JSP:** Oracle 9*i*FS の Web ユーザー・インタフェースで特定のコンテンツ・タイプのインスタンスを表示するための JavaServer Pages です。
- **レンダラ：** 様々なクライアントやプロトコルから表示するときの、情報の形式とレイアウトを決定するためのレンダラです。たとえば、Engineering Specification コンテンツ・タイプを、SMB からアクセスされた場合は固有の形式で表示し、Web からアクセスされた場合は HTML で表示するように構成できます。

コンテンツ・タイプ階層の定義

カスタム・コンテンツ・タイプを定義する第 2 のステップは、各コンテンツ・タイプがコンテンツ・タイプ階層に適合する場合を評価することです。

最初に、共通する特性に従ってカスタム・コンテンツ・タイプを階層形式で編成します。一部のコンテンツ・タイプが、より広範囲なコンテンツ・タイプに含まれる情報のサブセットを表していることに気づく場合があります。広範囲なコンテンツ・タイプについて一度定義した共通の特性は、階層内でそのタイプから派生するコンテンツ・タイプが継承できます。

次に、カスタム・コンテンツ・タイプが、インストール時の Oracle 9*i*FS に含まれているコンテンツ・タイプ階層にどのように適合するかを査定します。カスタムコンテンツ・タイプに対して定義する特性の多くは、すでに Document または Folder コンテンツ・タイプが持っている場合があります。たとえば、イメージのカスタム・コンテンツ・タイプには、Format や Size などの属性と `getContent()` や `setContent()` などのメソッドが必要ですが、これらは Document コンテンツ・タイプが持っています。

次に、本来のコンテンツ・タイプを拡張するか、修正して、トップレベルのコンテンツ・タイプが持つ追加の特性を実装するかを決定します。Image コンテンツ・タイプの場合、イメージはドキュメントのサブセットを表すため、Document コンテンツ・タイプを拡張する必要があります。Width、Height および Color Depth などの属性は、Oracle 9*i*FS に格納されているすべてのドキュメントではなく、イメージに適用する必要があります。また、すべてのドキュメントに属性と動作を追加する必要が生じる場合もあります。たとえば、ドキュメント内でウィルスをスキャンするアプリケーションの場合は、正常であるか感染しているかを示すために、すべてのドキュメントに属性 InfectionStatus を追加する必要があります。

カスタム・コンテンツ・タイプおよび属性の実装

コンテンツ・タイプと、コンテンツ・タイプ階層への編成方法を定義した後に、各コンテンツ・タイプに対応付ける必要のあるファイルを認識し、それを解析してカスタム属性を格納し、カスタム動作を実装し、様々なクライアントやプロトコルから表示されるときにレンダリングするように、Oracle 9*i*FS を構成できます。

Oracle 9*i*FS のカスタマイズは、XML の基礎知識があれば開始できます。この項では、次の基本的な開発タスクの実行方法について説明します。

- 新規コンテンツ・タイプの作成
- 既存のコンテンツ・タイプへの属性の追加
- 既存の属性の変更
- コンテンツ・タイプからの属性の削除
- ファイル拡張子とコンテンツ・タイプの対応付け
- カスタム・コンテンツ・タイプの削除

前述の特性に加えて、各カスタム・コンテンツ・タイプは、それを拡張する本来のコンテンツ・タイプから動作、JSP の対応付けおよびレンダラの対応付けを継承します。したがって、プログラミング・スキルがあまりなくても、Oracle 9*i*FS が本来持っているすべてのコンテンツ管理機能を活用して、独自の型を持つ情報を管理できます。

より高度なプログラミング・スキルを持っている場合は、動作方法、解析方法および表示方法など、コンテンツ・タイプの他の特性をカスタマイズできます。表 5-1「高度なコンテンツ・タイプ特性のカスタマイズ」に、コンテンツ・タイプの他の特性をカスタマイズするための高度な開発タスクを示します。この種の開発タスクについては後述します。このような他の特性の実装手順は、対応する章を参照してください。

表 5-1 高度なコンテンツ・タイプ特性のカスタマイズ

特性	開発タスク	章
動作	コンテンツ・タイプの動作を実装する Java クラスのメソッドを追加し、オーバーライドして、コンテンツ・タイプの動作をカスタマイズします。	第 17 章「コンテンツ・タイプの動作のカスタマイズ」
任意のメタデータと動作	Category、PropertyBundle、PolicyPropertyBundle および Relationship を使用して、各種コンテンツ・タイプとコンテンツ・タイプ・インスタンスに任意のメタデータと動作を適用します。	第 6 章「任意のメタデータと動作の適用」

表 5-1 高度なコンテンツ・タイプ特性のカスタマイズ（続く）

特性	開発タスク	章
パーサー	属性値が自動的に抽出され、カスタム・コンテンツ・タイプがインスタンス化されるように、カスタム・パーサーをファイル拡張子に対応付けます。	第 9 章「カスタム・パーサーの作成」
JSP	Oracle 9*i*FS の Web ユーザー・インタフェースにコンテンツ・タイプ・インスタンスが自動的に表示されるように、カスタム JavaServer Pages をコンテンツ・タイプに対応付けます。	第 12 章「Web インタフェースのカスタマイズ」
レンダラ	アプリケーションが内容を特定の表示クライアントに適した様々な形式やレイアウトに動的に変換できるように、1 つ以上のカスタム・レンダラをコンテンツ・タイプに対応付けます。	第 11 章「カスタム・レンダラの作成」

新規コンテンツ・タイプの作成

新規コンテンツ・タイプを実装するには、そのコンテンツ・タイプを表す SchemaObject のセットを作成します。第 3 章「Java API の概要」で前述したように、各コンテンツ・タイプは 1 つの ClassObject と Attribute のセットで表されます。ClassObject は、コンテンツ・タイプのインスタンスのデータが格納されるデータベース表や、コンテンツ・タイプの動作の実装に使用される Java クラスなど、コンテンツ・タイプに関する情報を保持します。コンテンツ・タイプの各拡張属性は、Attribute のインスタンスで表されます。Attribute インスタンスは、属性のデータ型、長さ、スケール、コンテンツ・タイプのインスタンスの属性値を保持するデータベース表のフィールドなど、属性に関するメタデータを保持します。また、Attribute インスタンスは、属性値に制約を適用するために ValueDefault、ValueDomain、ClassDomain のうちのどれを使用するかを登録します。

注意： ClassObject および Attribute で定義されるコンテンツ・タイプのメタデータの詳細は、第 3 章「Java API の概要」または Oracle 9*i*FS Javadoc を参照してください。ValueDefaults、ValueDomain および ClassDomain の実装手順は、第 7 章「属性の妥当性チェック」を参照してください。

コンテンツ・タイプを表す ClassObject を作成する場合は、少なくともコンテンツ・タイプの Name と SuperClass（階層内の派生元となるコンテンツ・タイプなど）を指定する必要があります。拡張コンテンツ・タイプ属性を表す Attribute インスタンスを作成する場合は、その属性の Name および Datatype を指定するのみで済みます。他のすべてのメタデータは、Oracle 9*i*FS によりデフォルトで設定されます。デフォルトをオーバーライドする場合は、

ClassObject インスタンスと Attribute インスタンスの値を同じ方法で明示的に設定できます。

新規コンテンツ・タイプを作成するには、管理権限が必要です。コンテンツ・タイプには、そのインスタンスのデータを格納するデータベース表が対応付けられているため、新規コンテンツ・タイプを作成するにはデータベース内で DDL 操作が必要になります。Oracle 9*i*FS のスキーマに対する DDL 変更を許可されているのは、管理者のみです。

実装方法

Oracle 9*i*FS SDK では、新規コンテンツ・タイプを表す SchemaObject を複数の方法で作成できます。

- **XML:** Oracle 9*i*FS へのインポート時にコンテンツ・タイプを自動的に構成する XML 構成ファイルを作成します。
- **Java:** Oracle 9*i*FS Java API 経由でコンテンツ・タイプをプログラムで構成します。
- **Oracle 9*i*FS Manager:** Graphical User Interface（GUI）を使用してコンテンツ・タイプを構成します。

以降は、XML または Java を使用して新規コンテンツ・タイプを実装する方法を、例を挙げて説明します。

注意： Oracle 9*i*FS Manager を使用して新規コンテンツ・タイプを作成する手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

XML 構成ファイルの使用

XML 構成ファイルを使用すると、新規コンテンツ・タイプを実装でき、プログラミング・スキルを必要としません。単純な XML ファイルを作成し、コンテンツ・タイプについて必要な情報を含めることができます。この XML ファイルを Oracle 9*i*FS にインポートすると、コンテンツ・タイプを表す ClassObject および Attribute インスタンスが自動的に作成されます。XML 構成ファイルのインポートに使用するユーザー・アカウントには、管理権限が必要です。

SMB 経由で XML 構成ファイルをインポートする場合は、コンテンツ・タイプの構成時にエラーが発生すると、Oracle 9*i*FS によりエラー・ログが生成されます。このログには XML 構成ファイルを示す名前が付けられ、XML 構成ファイルのインポート先ディレクトリに置かれます。Oracle 9*i*FS の XML ロギング機能により、コンテンツ・タイプが正常に構成されたかどうかを即時に判断できます。

XML 構成ファイルを使用すると、Oracle 9*i*FS に対して行ったカスタマイズ内容を、後で、または複数のシステム間で再生できます。構成ファイルを配置するときにエラーが発生した場合は、ファイルを簡単に変更して再配置できます。また、XML 構成ファイル・セットを

使用すると、同じコンテンツ・タイプ階層を持つ複数の Oracle 9*i*FS システムをセットアップできます。

例 5-1「XML を使用した新規コンテンツ・タイプの構成」に、新規コンテンツ・タイプを構成する XML 構成ファイルを示します。コンテンツ・タイプ名は Image です。Image コンテンツ・タイプは Document を拡張します。Image コンテンツ・タイプは最終的なものであるため、他のコンテンツ・タイプでは拡張できません。また、Width、Height および Artists という 3 つの属性を持っています。

例 5-1 XML を使用した新規コンテンツ・タイプの構成

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <NAME>Image</NAME>
    <SUPERCLASS RefType = "Name">Document</SUPERCLASS>
    <DESCRIPTION>Image files</DESCRIPTION>
    <ABSTRACT>false</ABSTRACT>
    <FINAL>true</FINAL>
    <CLASSACL RefType = "Name">CustomClassACL</CLASSACL>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>Width</NAME>
            <DATATYPE>Integer</DATATYPE>
        </ATTRIBUTE>
        <ATTRIBUTE>
            <NAME>Height</NAME>
            <DATATYPE>Integer</DATATYPE>
        </ATTRIBUTE>
        <ATTRIBUTE>
            <NAME>Artists</NAME>
            <DATATYPE>DirectoryObject_Array</DATATYPE>
            <CLASSDOMAIN RefType = "Name">DirectoryUser...</CLASSDOMAIN>
       </ATTRIBUTE>
    </ATTRIBUTES>
</CLASSOBJECT>
```

XML 構成ファイルの各要素を使用して、コンテンツ・タイプの表示に使用される ClassObject および Attribute インスタンスの属性の値が指定されます。<CLASSDOMAIN> 要素の値など、Oracle 9*i*FS 内の既存のオブジェクトを参照する属性値の場合は、RefType 属性を使用して要素の値によるオブジェクトの参照方法を指定します。

表 5-2「コンテンツ・タイプを表す XML 構成ファイルの要素」に、コンテンツ・タイプ構成ファイルの各 XML 要素を示します。この表は、Java クラスとその要素に対応する Oracle 9*i*FS Java API の属性、各要素の使用方法および要素の有効な XML 属性を示しています。

表 5-2 コンテンツ・タイプを表す XML 構成ファイルの要素

要素	クラス情報	属性	使用方法
<CLASSOBJECT>	ClassObject クラス		どのようなコンテンツ・タイプの場合も、XML 構成ファイルのルート要素は <CLASSOBJECT> である必要があります。この要素は、IfsSimpleXmlParser により ClassObject クラスのインスタンスの作成に使用されます。 <CLASSOBJECT> 要素の副要素は、それぞれ ClassObject クラスの属性に直接対応します。それぞれの副要素のタグ名は、ClassObject 属性の名前と一致する必要があります。(ClassObject 属性の詳細リストは、Oracle 9*i*FS Javadoc を参照してください)。
<NAME>	ClassObject クラスの Name 属性		XML 構成ファイルでは、<NAME> 要素の値として新規 ClassObject の名前を指定する必要があります。
<SUPERCLASS>	ClassObject クラスの SuperClass 属性	RefType	新規 ClassObject のスーパークラスは、<SUPERCLASS> 要素で指定します。要素の RefType 属性を使用してスーパークラスを名前で参照するのが、最も便利な方法です。
<DESCRIPTION>	ClassObject クラスの Description 属性		新規 ClassObject の記述を <DESCRIPTION> 要素で指定できます。
<BEANCLASSPATH>	ClassObject クラスの BeansClassPath 属性		<BEANSCLASSPATH> 要素を使用して、この ClassObject のインスタンスを SDK で表示して管理するための、Bean 側 Java クラスの完全修飾クラス名を指定します。この要素の詳細は、第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照してください。
<SERVERCLASSPATH>	ClassObject クラスの ServerClassPath 属性		<SERVERCLASSPATH> 要素を使用して、この ClassObject のインスタンスを表示して管理するための、サーバー側 Java クラスの完全修飾クラス名を指定します。この要素の詳細は、第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照してください。

表 5-2 コンテンツ・タイプを表す XML 構成ファイルの要素（続く）

要素	クラス情報	属性	使用方法
<DATABASEOBJECTNAME>	ClassObject クラスの DatabaseObjectName 属性		ClassObject のインスタンス・データの格納に使用するデータベース・オブジェクトのカスタム・ベース名を指定できます。指定しない場合、デフォルトでデータベース・オブジェクトのベース名は ClassObject の名前になります。 ClassObject の Name が 25 文字以上の場合は、DatabaseObjectName の値を指定する必要があります。
<FINAL>	ClassObject クラスの Final 属性		<FINAL> 要素を 'true' または 'false' に設定して、このコンテンツ・タイプを他のコンテンツ・タイプで拡張できるかどうかを指定できます。
<ABSTRACT>	ClassObject クラスの Abstract 属性		<ABSTRACT> 要素を 'false' または 'true' に設定して、コンテンツ・タイプをインスタンス化できるか、または派生コンテンツ・タイプの共通特性を定義する抽象クラスとしてのみ機能するかを指定できます。
<PARTITIONED>	ClassObject クラスの Partitioned 属性		<PARTITIONED> 要素を 'true' または 'false' に設定して、ClassObject のデータベース表がパーティション化されるかどうかを指定できます。
<CLASSACL>	ClassObject クラスの ClassACL 属性		新規クラスの ClassAccessControlList では、クラスをインスタンス化できるユーザーやグループを定義します。新規クラスの ClassACL を定義するには、XML 構成ファイルで <CLASSOBJECT> の <CLASSACL> 要素を定義します。この要素を定義しない場合は、デフォルトですべてのユーザーがクラスをインスタンス化できます。 ClassACL の構成方法の詳細は、第 15 章「セキュリティ」を参照してください。

表 5-2 コンテンツ・タイプを表す XML 構成ファイルの要素（続く）

要素	クラス情報	属性	使用方法
<ATTRIBUTES>	ClassObject クラスの属性との直接の対応関係はなし		新規 ClassObject の属性は、<ATTRIBUTES> 配列に埋め込まれます。この配列の各要素は属性の宣言です。 <ATTRIBUTES> 要素は、ClassObject クラスの属性とは直接対応しません。ClassObject とその Attribute オブジェクトとの関係は、Attribute クラスの Class 属性により追跡されます。 XML 構成ファイルを処理するときに、IfsSimpleXMLParser では <ATTRIBUTES> 配列内で宣言されている属性ごとに Attribute クラスのインスタンスが作成されます。
<ATTRIBUTE>	Attribute クラスが SchemaObject を拡張		新規コンテンツ・タイプの属性は、それぞれ <ATTRIBUTE> 要素で宣言されます。 <ATTRIBUTE> 要素では、データ型や制約など、属性に関するメタデータが定義されます。
<NAME>	Attribute クラスの Name 属性		各 <ATTRIBUTE> 宣言では、<NAME> 要素で属性の名前が指定されます。
<DATATYPE>	Attribute クラスの DataType 属性		属性のデータ型は、<DATATYPE> 要素内で指定します。データ型には、STRING、INTEGER、LONG、DOUBLE、BOOLEAN、DATE、ObjectReference、PublicObject、DirectoryObject、SystemObject、SchemaObject、String_Array、Integer_Array、Long_Array、Double_Array、Boolean_Array、Date_Array、ObjectReference_Array、PublicObject_Array、DirectoryObject_Array、SystemObject_Array、SchemaObject_Array のいずれかを指定できます。
<DATALENGTH>	Attribute クラスの DataLength 属性		String 属性の値の最大長です。
<UNIQUE>	Attribute クラスの Unique 属性		<UNIQUE> 要素を使用して、ClassObject の各インスタンスがこの属性に必ず一意の値を持つように（NULL でない場合）指定できます。

表 5-2 コンテンツ・タイプを表す XML 構成ファイルの要素（続く）

要素	クラス情報	属性	使用方法
<REQUIRED>	Attribute クラスの Required 属性		<REQUIRED> 要素を使用して、ClassObject の各インスタンスがこの属性に必ず NULL 以外の値を持つように指定できます。
<INDEXED>	Attribute クラスの Indexed 属性		<INDEXED> 要素を使用して、この ClassObject のデータを格納するデータベース表に、この属性用の索引を作成する必要があるかどうかを指定できます。
<SETTABLE>	Attribute クラスの Settable 属性		<SETTABLE> 要素を使用して、ClassObject の新規インスタンスを作成するときに、リポジトリ SDK を使用してこの属性を設定できるかどうかを指定できます。設定できない属性は、サーバー拡張コードによる「システム設定」のみです。
<UPDATABLE>	Attribute クラスの Updatable 属性		<UPDATABLE> 要素を使用して、既存の ClassObject インスタンスについて、リポジトリ SDK を使用してこの属性を更新できるかどうかを指定できます。更新できない属性の値は、サーバー拡張コードでのみ設定できます。また、更新できない属性は、「システム設定」属性とも呼ばれます。
<DATABASEOBJECTNAME>	Attribute クラスの DatabaseObjectName		ClassObject について、この属性の値をデータベース表に格納するために使用する列のカスタム・ベース名を指定できます。 指定しない場合は、デフォルトでデータベース・オブジェクトのベース名はこの属性の名前になります。 Attribute の Name が 31 文字以上の場合、または SQL の予約語（'table' など）と同じ場合は、DatabaseObjectName を指定する必要があります。

表 5-2 コンテンツ・タイプを表す XML 構成ファイルの要素（続く）

要素	クラス情報	属性	使用方法
<CLASSDOMAIN>	Attribute クラスの ClassDomain 属性		<CLASSDOMAIN> 要素を使用して、この属性に適用される ClassDomain を指定できます。 ClassDomain を使用できるのは、オブジェクト・データ型（つまり、PublicObject、SystemObject、SchemaObject、DirectoryObject、PublicObject_Array、SystemObject_Array、SchemaObject_Array、DirectoryObject_Array）を持つ属性の場合のみです。ClassDomain は、属性の値を指定されたクラスのインスタンスに制限します。ClassDomain の構成手順は、第 7 章「属性の妥当性チェック」を参照してください。
<VALUEDOMAIN>	Attribute クラスの ValueDomain 属性		<VALUEDOMAIN> 要素を使用して、この属性に適用される ValueDomain を指定できます。 ValueDomain は、Attribute の値を可能な値の列挙リストまたは範囲に制限します。ValueDomain の構成手順は、第 7 章「属性の妥当性チェック」を参照してください。
<VALUEDEFAULT>	Attribute クラスの ValueDefault 属性		<VALUEDEFAULT> 要素を使用して、この属性のデフォルト値を指定できます。ValueDefault の構成手順は、第 7 章「属性の妥当性チェック」を参照してください。
<REFERENTIALINTEGRITYRULE>	Attribute クラスの ReferentialIntegrity Rule 属性		<REFERENTIALINTEGRITYRULE> 要素を使用して、この属性の参照整合性動作を指定できます。考えられる値は 0（ゼロ）と 1 で、それぞれ値定数 RIRULE_CLEAR および RIRULE_RESTRICT の整数表現です。

Java API の使用

Oracle 9*i*FS Java API を使用して、新規コンテンツ・タイプをプログラムで構成できます。

Java は、カスタム・アプリケーション用の Java インストール・ユーティリティを提供する必要がある場合に、コンテンツ・タイプの構成に役立ちます。たとえば、独立系ソフトウェア・ベンダー（ISV）が、Oracle 9*i*FS のコンテンツ・タイプ階層を拡張して自社のテクノロジを統合するとします。ISV は、顧客が事前にインストールされた Oracle Internet File System に自社ソフトウェアとカスタム・コンテンツ・タイプをインストールできるように、インストール・ユーティリティを組み込む必要があります。また、別の例として、管理者がコンテンツ・タイプ階層を拡張できるように、JSP ベースのユーザー・インタフェースを構

築できます。JSP の JavaBeans では、Oracle 9*i*FS Java API をコールして ClassObject および Attribute を動的に構成できます。

Java を使用してコンテンツ・タイプを構成すると、構成プロセスを厳密に制御できます。たとえば、属性 Products を持つカスタム・コンテンツ・タイプ PurchaseOrder を作成し、この属性に ClassDomain で制約を適用し、別のカスタム・コンテンツ・タイプ Product のインスタンスに限定するとします。両方のコンテンツ・タイプおよび ClassDomain を 1 回の操作で作成し、三者間の参照整合性を保証できます。

Java API を使用してコンテンツ・タイプを構成するには、Java プログラムで次の手順を実行します。

1. 管理モードを有効化します。
2. コンテンツ・タイプの ClassObjectDefinition を構成します。
3. ClassObjectDefinition の SuperClass 属性に拡張されるコンテンツ・タイプを指定します。
4. コンテンツ・タイプの各属性の AttributeDefinition を構成します。
5. ClassObjectDefinition を LibrarySession に渡して、Oracle 9*i*FS 内のコンテンツ・タイプを表す ClassObject および Attribute インスタンスを作成します。

例 5-2 Java を使用した新規コンテンツ・タイプの構成

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. 新規 ClassObject の値を保持する ClassObjectDefinition を作成します。Name および Description 属性を設定します。

```
ClassObjectDefinition codef = new ClassObjectDefinition(session);
codef.setAttributeByUpperCaseName(ClassObject.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("IMAGE"));
codef.setAttributeByUpperCaseName(ClassObject.DESCRIPTION_ATTRIBUTE,
       AttributeValue.newAttributeValue("Image files"));
```

3. 拡張するコンテンツ・タイプを指定します。

```
ClassObject superClass = session.getClassObjectByName("DOCUMENT");
codef.setSuperclass(superClass);
```

4. 新規コンテンツ・タイプの各拡張属性を表す AttributeDefinition を作成します。AttributeDefinition を ClassObjectDefinition に追加します。

```
AttributeDefinition adef1 = new AttributeDefinition(session);
adef1.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("Height"));
adef1.setAttributeByUpperCaseName(Attribute.DESCRIPTION_ATTRIBUTE,
```

```
     AttributeValue.newAttributeValue("Image Height"));
adef1.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
     AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_INTEGER));
adef1.setAttributeByUpperCaseName(Attribute.REQUIRED_ATTRIBUTE,
             AttributeValue.newAttributeValue(false));
codef.addAttributeDefinition(adef1);

AttributeDefinition adef2 = new AttributeDefinition(session);
adef2.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("Width"));
adef2.setAttributeByUpperCaseName(Attribute.DESCRIPTION_ATTRIBUTE,
     AttributeValue.newAttributeValue("Image Width"));
adef2.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
     AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_INTEGER));
adef2.setAttributeByUpperCaseName(Attribute.REQUIRED_ATTRIBUTE,
             AttributeValue.newAttributeValue(false));
codef.addAttributeDefinition(adef2);
```

5. ClassObjectDefinition を LibrarySession に渡して、コンテンツ・タイプの ClassObject および Attribute インスタンスを作成します。

```
ClassObject classObj = (ClassObject) session.createSchemaObject(codef);
```

既存のコンテンツ・タイプへの属性の追加

Oracle 9*i*FS では、作成したコンテンツ・タイプの属性を追加または変更することもできます。

定義したカスタム・コンテンツ・タイプの 1 つについて、さらに属性が必要であることが後で判明することがあります。また、Oracle 9*i*FS に付属の Document および Folder コンテンツ・タイプに、カスタム属性を追加する必要が生じることもあります。コンテンツ・タイプには、いつでも属性を追加できます。

既存のコンテンツ・タイプに属性を追加する場合、新規属性を必須属性にすることはできません。必須属性は、その属性値に NULL を使用できないことを示します。この場合、Oracle 9*i*FS では、そのコンテンツ・タイプの既存のインスタンスについて、その属性の値を NULL に設定できなくなります。

「新規コンテンツ・タイプの作成」で説明したように、Oracle 9*i*FS では、すべてのコンテンツ・タイプは ClassObject のインスタンスで表され、その各属性は Attribute のインスタンスで表されます。追加の属性を表す新規 Attribute インスタンスを作成して、既存のコンテンツ・タイプの ClassObject インスタンスに対応付けることができます。

実装方法

Oracle 9*i*FS SDK では、次の 2 つの方法で既存のコンテンツ・タイプに属性を追加できます。

- **Java:** Oracle 9*i*FS Java API を使用してコンテンツ・タイプをプログラムで構成します。
- **Oracle 9*i*FS Manager:** GUI を使用してコンテンツ・タイプを構成します。

注意： Oracle 9*i*FS Manager の使用手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

Java API を使用してコンテンツ・タイプに属性を追加するには、Java プログラムで次の手順を実行します。

1. セッションの管理モードを有効化します。
2. コンテンツ・タイプを表す ClassObject を取得します。
3. 新規属性の AttributeDefinition を作成します。
4. AttributeDefinition を ClassObject の `addAttribute()` メソッドに渡して新規 Attribute インスタンスを作成し、ClassObject インスタンスに対応付けます。

例 5-3 Java を使用したコンテンツ・タイプへの属性の追加

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. ClassObject を取得します。

```
ClassObject co = session.getClassObjectByName("IMAGE");
```

3. AttributeDefinition を作成します。

```
AttributeDefinition attdef = new AttributeDefinition(session);
attdef.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("Artists"));
attdef.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(
            Attribute.ATTRIBUTEDATATYPE_DIRECTORYOBJECT_ARRAY));
```

4. Attribute を ClassObject に追加します。

```
co.addAttribute(attdef);
```

既存の属性の変更

Oracle 9*i*FS では、XML または Java を使用して、コンテンツ・タイプの既存の属性を変更できます。たとえば、属性の参照整合性ルールを指定するのを忘れる場合があります。また、新規の ValueDefault、ValueDomain または ClassDomain を作成し、それを Oracle 9*i*FS の既存の複数の属性に適用することが必要となる場合があります。

属性を変更できるのは、変更結果がすでに作成済みのコンテンツ・タイプのインスタンスと矛盾しない場合のみです。また、属性の変更を、その属性の値の格納に使用される基礎となるデータベース列に反映させることはできません。したがって、Oracle 9*i*FS で属性を作成した後は、その Name、DatabaseObjectName または Datatype は変更できません。

XML を使用した属性の更新

XML を使用して ClassObject を更新できるのと同様に、属性を更新する XML 構成ファイルを作成できます。この場合、<ATTRIBUTE> 要素には、更新対象の属性を参照するための <UPDATE> 副要素が含まれます。<ATTRIBUTE> 指定には更新後の参照整合性ルールを表す副要素が含まれますが、属性について変化しないメタデータ（データ型、長さなど）の副要素を含める必要はありません。

注意： Attribute を名前で参照する場合は、値を大文字にする必要があります（<UPDATE RefType = "Name">ARTISTS</UPDATE> など）。

例 5-4 に、XML を使用してコンテンツ・タイプの属性を更新する方法を示します。XML 構成ファイルでは、Image コンテンツ・タイプの Artists 属性を更新し、イメージのアーティストとして指定された場合に、DirectoryUsers の削除を制限する参照整合性ルールを適用します。

例 5-4 XML を使用したコンテンツ・タイプの属性の更新

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <UPDATE RefType = "Name">Image</UPDATE>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <UPDATE RefType = "Name">ARTISTS</UPDATE>
            <REFERENTIALINTEGRITYRULE>1</REFERENTIALINTEGRITYRULE>
        </ATTRIBUTE>
    </ATTRIBUTES>
</CLASSOBJECT>
```

Java を使用した属性の更新

Oracle 9*i*FS Java API を使用して、属性をプログラムで変更できます。この方法が役立つのは、管理者にコンテンツ・タイプの構成変更用のカスタム・ユーザー・インタフェースを提供する必要がある場合です。たとえば、ユーザーが新規の ValueDomain、ClassDomain および ValueDefault を作成してコンテンツ・タイプの属性に適用できるように、JSP ベースのクライアントを作成できます。

Java API を使用してコンテンツ・タイプの属性を更新するには、Java プログラムで次の手順を実行します。

1. セッションの管理モードを有効化します。
2. コンテンツ・タイプの属性を表す Attribute インスタンスを取得します。
3. Attribute の更新に適したメソッド（つまり、`setClassDomain()`、`setValueDomain()`、`setValueDefault()`、`setReferentialIntegrityRule()`）をコールします。

属性の更新に使用するメソッドの詳細リストは、Oracle 9*i*FS Javadoc を参照してください。

例 5-5 Java を使用したコンテンツ・タイプの属性の更新

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. コンテンツ・タイプを表す ClassObject を取得します。

```
ClassObject co = session.getClassObjectByName("IMAGE");
```

3. Attribute インスタンスを取得します。

```
Attribute att = co.getEffectiveClassAttributes("ARTISTS");
```

4. 適切なメソッドをコールして Attribute を更新します。

```
att.setReferentialIntegrityRule(Attribute.RIRULE_RESTRICT);
```

コンテンツ・タイプからの属性の削除

作成後のコンテンツ・タイプから属性を削除することもできます。削除できるのは、拡張属性のみです。Oracle 9*i*FS のコンテンツ・タイプの属性（Document など）は、どの実装方法でも削除できません。

実装方法

Oracle 9*i*FS SDK でサポートされている次の 2 つのプログラミング方法を使用して、属性を削除できます。

- **Java:** Oracle 9*i*FS Java API を使用して属性をプログラムで削除します。

- **Oracle 9*i*FS Manager:** GUI を使用して属性を削除します。

このリリースの Oracle Internet File System では、属性の削除方法としての XML はサポートされません。コンテンツ・タイプのインスタンスが存在する場合、属性の削除には Oracle 9*i*FS Manager を使用できません。その場合は、Java API を使用して、属性の削除がコンテンツ・タイプのインスタンスに与える影響を処理します。

注意： Oracle 9*i*FS Manager の使用手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

コンテンツ・タイプから属性を削除するには、Java プログラムで次の手順を実行します。

1. 管理モードを有効化します。
2. 属性を削除するコンテンツ・タイプを表す ClassObject インスタンスを取得します。
3. ClassObject の `getAttributeFromLabel()` メソッドをコールして、削除する Attribute インスタンスを取得します。
4. Attribute を ClassObject の `removeAttribute()` メソッドに渡して、その属性を削除します。

例 5-6 Java を使用したコンテンツ・タイプからの属性の削除

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. コンテンツ・タイプを表す ClassObject を取得します。

```
ClassObject co = session.getClassObjectByName("IMAGE");
```

3. ClassObject から削除する Attribute を取得します。

```
Attribute att = co.getEffectiveClassAttributes("ARTISTS");
```

4. Attribute を ClassObject から削除します。

```
co.removeAttribute(att);
```

ファイル拡張子とコンテンツ・タイプの対応付け

デフォルトでは、すべてのファイルは Oracle 9*i*FS により Document コンテンツ・タイプのインスタンスとして格納されます。ただし、特定のファイル拡張子を持つファイルをカスタム・コンテンツ・タイプのインスタンスとしてインポートするように、Oracle 9*i*FS を構成できます。たとえば、Width、Height および Color Depth などのカスタム属性を格納したり、`generateThumbnail()` などのカスタム動作を適用できるように、.jpg、.bmp および .gif

ファイルを Image コンテンツ・タイプのインスタンスとして格納できます。エンド・ユーザーは標準的なプロトコルとデスクトップ・クライアントを使用して .jpg、.bmp および .gif ファイルを操作する間に、Oracle 9*i*FS ではそれを Image コンテンツ・タイプのインスタンスとして透過的に格納できます。これにより、ユーザーは実装されているその他の特性を活用できます。

ClassSelectionParser

Oracle 9*i*FS のこの機能は、ClassSelectionParser とともに実装されます。ClassSelectionParser は、特定のファイル拡張子に対応付けられているコンテンツ・タイプを判断する特殊なパーサーです。

ファイルがリポジトリにインポートされると、Oracle 9*i*FS では、まずファイルの拡張子が、そのファイルの前処理に使用する必要のあるパーサーと対応付けられているかどうかが判断されます。ファイル拡張子とパーサー間のマッピングは、`ParserLookupByFileExtension` という ValueDefault 経由で Oracle 9*i*FS によりアクセスされる PropertyBundle に格納されています。PropertyBundle は、登録対象となるファイル拡張子ごとに 1 つずつ、複数の Property オブジェクトからなる配列で構成されます。各 Property には、マッピングが名前 / 値のペアとして格納されます。

- Name には、ファイル拡張子（.gif など）が格納されます。
- **Value** には、パーサーの完全修飾パス（`oracle.ifs.beans.parsers.ClassSelectionParser` など）が格納されます。

ファイルの拡張子が ClassSelectionParser に対応付けられている場合は、Oracle 9*i*FS ではそのパーサーを使用して、ファイルでインスタンス化する必要のあるコンテンツ・タイプが判断されます。ファイル拡張子とコンテンツ・タイプとのマッピングは、`IFS.PARSER.ObjectTypeLookupByFileExtension` という ValueDefault 経由でアクセスされる PropertyBundle に格納されます。PropertyBundle は、各マッピングを名前 / 値のペアとして格納する Property の配列で構成されます。

- Name には、ファイル拡張子（.gif など）が格納されます。
- **Value** にはコンテンツ・タイプ名（Image など）が格納されます。

コンテンツ・タイプの確認後に、ClassSelectionParser はファイルをそのコンテンツ・タイプのインスタンスとして格納します。インスタンスの作成時に、ClassSelectionParser はそのコンテンツ・タイプのすべての必須属性の値を指定できる必要があります。必須属性がない場合、ClassSelectionParser はコンテンツ・タイプのインスタンスを作成します。このインスタンスにはファイルの内容が格納され、Document クラスから継承したシステム設定属性（Owner、CreateDate、Modification Date など）を自動的に提供します。ただし、カスタム属性が必須で、ValueDefault を持っていない場合、ClassSelectionParser はその属性の値を与えることができず、コンテンツ・タイプを正常にインスタンス化できません。その場合は、カスタム・パーサーを実装し、コンテンツ・タイプをインスタンス化する前に、ファイルから必須属性の値を抽出する必要があります。

注意： カスタム・パーサーの実装手順は、第 9 章「カスタム・パーサーの作成」を参照してください。

デフォルト・マッピング

明示的なマッピングを持たないすべてのファイル拡張子について、デフォルトのコンテンツ・タイプを定義できます。たとえば、jpg、bmp および gif 拡張子は Image コンテンツ・タイプに明示的にマップされていますが、他のすべての拡張子はコンテンツ・タイプ Document ではなく ProjectFile にマップする必要があります。このマッピングを実装するには、デフォルトのコンテンツ・タイプをアスタリスクとマップします。`ParserLookupByFileExtension` の PropertyBundle には、Name = '*' および Value = 'oracle.ifs.beans.parsers.ClassSelectionParser' の Property が含まれ、`IFS.PARSER.ObjectTypeLookupByFileExtension` の PropertyBundle には、Name = '*' および Value = 'ProjectFile' の Property が含まれています。ClassSelectionParser により使用クラスが判断されるときには、特定のマッピングがアスタリスクより優先されます。

実装方法

Oracle 9*i*FS SDK には、ファイル拡張子をコンテンツ・タイプと対応付けるために、様々なメソッドが用意されています。

- **XML:** Oracle 9*i*FS へのインポート時にマッピングを自動的に作成するように、XML ファイルを作成します。
- **Java:** Oracle 9*i*FS Java API を使用して、ファイル拡張子をコンテンツ・タイプとプログラムでマップします。
- **Oracle 9*i*FS Manager:** GUI を使用してマッピングを作成します。

以降は、XML または Java を使用して属性を変更する方法を、例を挙げて説明します。

注意： Oracle 9*i*FS Manager の使用手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

XML を使用したファイル拡張子のマップ

XML 構成ファイルを使用すると、簡単で再生可能な方法でファイル拡張子をコンテンツ・タイプとマップできます。次の 3 つの例に、XML を次の操作に使用する方法を示します。

- 特定のファイル拡張子とコンテンツ・タイプとのマップ
- 他のすべてのファイル拡張子のデフォルト・マッピングの作成

- ファイル拡張子とコンテンツ・タイプ間のマッピングの削除

例 5-7 XML を使用した特定のファイル拡張子のマップ

a. ClassSelectionParser を .gif 拡張子と対応付けます。

```
<?xml version='1.0' standalone='yes'?>
<OBJECTLIST>
 <PROPERTYBUNDLE>
  <UPDATE RefType='ValueDefault'>ParserLookupByFileExtension</UPDATE>
   <PROPERTIES>
    <PROPERTY Action = 'add'>
     <NAME> gif </NAME>
      <VALUE DataType = 'String'>
         oracle.ifs.beans.parsers.ClassSelectionParser
      </VALUE>
   </PROPERTY>
  </PROPERTIES>
 </PROPERTYBUNDLE>
```

b. .gif 拡張子を Image クラスと対応付けます。

```
 <PROPERTYBUNDLE>
  <UPDATE RefType='ValueDefault'>
IFS.PARSER.ObjectTypeLookupByFileExtension</UPDATE>
   <PROPERTIES>
    <PROPERTY Action = 'add'>
     <NAME>gif</NAME>
     <VALUE DataType='String'>Image</VALUE>
    </PROPERTY>
   </PROPERTIES>
  </PROPERTYBUNDLE>
</OBJECTLIST>
```

例 5-8 XML を使用したデフォルト・マッピングの作成

a. ParserLookupByFileExtension にデフォルトの ClassSelectionParser エントリを追加します。

```
<?xml version = '1.0' standalone = 'yes'?>
<OBJECTLIST>
 <PROPERTYBUNDLE>
  <UPDATE RefType='ValueDefault'>ParserLookupByFileExtension</UPDATE>
   <PROPERTIES>
    <PROPERTY Action = 'add'>
     <NAME>*</NAME>
     <VALUE DataType = 'String'>
        oracle.ifs.beans.parsers.ClassSelectionParser
```

```
      </VALUE>
    </PROPERTY>
   </PROPERTIES>
  </PROPERTYBUNDLE>
```

b. すべてのファイルを Document クラスと対応付けます。

```
 <PROPERTYBUNDLE>
  <UPDATE RefType = 'ValueDefault'>
IFS.PARSER.ObjectTypeLookupByFileExtension</UPDATE>
   <PROPERTIES>
    <PROPERTY Action = 'add'>
     <NAME>*</NAME>
      <VALUE DataType='String'>Document</VALUE>
    </PROPERTY>
   </PROPERTIES>
  </PROPERTYBUNDLE>
</OBJECTLIST>
```

例 5-9 XML を使用したファイル拡張子のマッピングの削除

a. ClassSelectionParser と .gif 拡張子との対応付けを削除します。

```
<?xml version = '1.0' standalone = 'yes'?>
<OBJECTLIST>
 <PROPERTYBUNDLE>
   <UPDATE RefType = 'ValueDefault'> ParserLookupByFileExtension</UPDATE>
   <PROPERTIES>
    <PROPERTY Action = 'remove'>
     <NAME>gif</NAME>
     <VALUE datatype = 'String'>
        oracle.ifs.beans.parsers.ClassSelectionParser
     </VALUE>
    </PROPERTY>
   </PROPERTIES>
  </PROPERTYBUNDLE>
```

b. .gif 拡張子と Image クラスとの対応付けを削除します。

```
 <PROPERTYBUNDLE>
   <UPDATE RefType = 'ValueDefault'>
      IFS.PARSER.ObjectTypeLookupByFileExtension
   </UPDATE>
   <PROPERTIES>
    <PROPERTY Action = 'remove'>
    <NAME>gif</NAME>
     <VALUE datatype='String'>Image</VALUE>
    </PROPERTY>
```

```
    </PROPERTIES>
   </PROPERTYBUNDLE>
</OBJECTLIST>
```

Java を使用したファイル拡張子のマップ

Oracle 9*i*FS Java API を使用して、ファイル拡張子とコンテンツ・タイプとのマッピングをプログラムで作成し、削除できます。

マッピングを作成するには、Java プログラムで次の手順を実行します。

1. `ParserLookupByFileExtension` という ValueDefault を選択し、その PropertyBundle を取得します。
2. PropertyBundle の `putPropertyValue()` メソッドをコールして、ファイル拡張子を ClassSelectionParser とマップします。
3. `IFS.PARSER.ObjectTypeLookupByFileExtension` という ValueDefault を選択し、その PropertyBundle を取得します。
4. PropertyBundle の `putPropertyValue()` メソッドをコールして、ファイル拡張子をコンテンツ・タイプとマップします。

例 5-10 Java を使用したファイル拡張子のマッピングの作成

1. ValueDefault を介して ParserLookupByFileExtension の PropertyBundle を取得します。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd =
     (ValueDefault) vdColl.getItems("ParserLookupByFileExtension");
AttributeValue av = vd.getPropertyValue();
PropertyBundle pb = (PropertyBundle) av.getPublicObject(session);
```

2. 'gif' 拡張子を ClassSelectionParser とマップする新規 Property を追加します。

```
pb.putPropertyValue("gif", AttributeValue.newAttributeValue(
                            "oracle.ifs.beans.parsers.ClassSelectionParser"));
```

3. ValueDefault を経由して、IFS.PARSER.ObjectTypeLookupByFileExtension の PropertyBundle を取得します。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd = (ValueDefault)
     vdColl.getItems("IFS.PARSER.ObjectTypeLookupByFileExtension");
AttributeValue av = vd.getPropertyValue();
PropertyBundle pb = (PropertyBundle) av.getPublicObject(session);
```

4. 'gif' 拡張子を 'Image' コンテンツ・タイプとマップする新規 Property を追加します。

```
pb.putPropertyValue("gif", AttributeValue.newAttributeValue("IMAGE"));
```

マッピングを削除するには、Java プログラムで次の手順を実行します。

1. `ParserLookupByFileExtension` という ValueDefault を選択し、その PropertyBundle を取得します。
2. PropertyBundle の `removePropertyValue()` メソッドをコールして、ClassSelectionParser からファイル拡張子との対応付けを解除します。
3. `IFS.PARSER.ObjectTypeLookupByFileExtension` という ValueDefault を選択し、その PropertyBundle を取得します。
4. PropertyBundle の `removePropertyValue()` メソッドをコールして、ファイル拡張子とコンテンツ・タイプとの対応付けを解除します。

例 5-11 Java を使用したファイル拡張子のマッピングの削除

1. ValueDefault を介して ParserLookupByFileExtension の PropertyBundle を取得します。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd =
     (ValueDefault) vdColl.getItems("ParserLookupByFileExtension");
AttributeValue av = vd.getPropertyValue();
PropertyBundle pb = (PropertyBundle) av.getPublicObject(session);
```

2. 'gif' 拡張子を ClassSelectionParser とマップする Property を削除します。

```
pb.removePropertyValue("gif");
```

3. ValueDefault を経由して、IFS.PARSER.ObjectTypeLookupByFileExtension の PropertyBundle を取得します。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd = (ValueDefault)
     vdColl.getItems("IFS.PARSER.ObjectTypeLookupByFileExtension");
AttributeValue av = vd.getPropertyValue();
PropertyBundle pb = (PropertyBundle) av.getPublicObject(session);
```

4. 'gif' 拡張子を 'Image' コンテンツ・タイプとマップする Property を削除します。

```
pb.removePropertyValue("gif");
```

カスタム・コンテンツ・タイプの削除

後になってコンテンツ・タイプの削除が必要になる場合があります。コンテンツ・タイプのインスタンスが存在しているかぎり、そのコンテンツ・タイプは削除できません。また、コンテンツ・タイプ階層内で他のコンテンツ・タイプが派生しているコンテンツ・タイプは、削除できません。

実装方法

Oracle 9*i*FS SDK でサポートされている次の 2 つのプログラミング方法を使用して、コンテンツ・タイプを削除できます。

- **Java:** Oracle 9*i*FS Java API を経由してコンテンツ・タイプをプログラムで削除します。
- **Oracle 9*i*FS Manager:** GUI を使用してコンテンツ・タイプを削除します。

このリリースの Oracle Internet File System では、コンテンツ・タイプの削除方法としての XML はサポートされていません。コンテンツ・タイプのインスタンスが存在する場合、そのコンテンツ・タイプの削除には Oracle 9*i*FS Manager を使用できません。このような場合は、Java API を使用してこれらのインスタンスを処理してから、コンテンツ・タイプを削除します。

注意： Oracle 9*i*FS Manager の使用手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

コンテンツ・タイプを削除するには、Java プログラムで次の手順を実行します。

1. 管理モードを有効化します。
2. 削除するコンテンツ・タイプを表す ClassObject インスタンスを取得します。
3. `free()` メソッドをコールしてコンテンツ・タイプを削除します。

例 5-12 Java を使用したコンテンツ・タイプの削除

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. コンテンツ・タイプを表す ClassObject を取得します。

```
ClassObject co = session.getClassObjectByName("IMAGE");
```

3. `free()` メソッドをコールして ClassObject を削除します。

```
co.free();
```

サンプル・コード

Oracle 9*i*FS は、この章の例について実行可能なサンプル・コード・ファイルとともにインストールされます。サンプル・コード・ファイルは、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/devdoc/subclassing` ディレクトリにあります。表 5-3 に、サンプル・コード・ファイルとそれに対応する例を示します。

表 5-3 例とサンプル・コード・ファイル

例	サンプル・コード・ファイル
例 5-1「XML を使用した新規コンテンツ・タイプの構成」	`CreateContentType.xml`
例 5-2「Java を使用した新規コンテンツ・タイプの構成」	`CreateContentType.java`
例 5-3「Java を使用したコンテンツ・タイプへの属性の追加」	`AddContentTypeAttributes.java`
例 5-4「XML を使用したコンテンツ・タイプの属性の更新」	`UpdateContentTypeAttributes.xml`
例 5-5「Java を使用したコンテンツ・タイプの属性の更新」	`UpdateContentTypeAttributes.java`
例 5-6「Java を使用したコンテンツ・タイプからの属性の削除」	`DeleteContentTypeAttributes.java`
例 5-7「XML を使用した特定のファイル拡張子のマップ」	`RegisterContentTypeExtensions.xml`
例 5-8「XML を使用したデフォルト・マッピングの作成」	`RegisterDefaultExtension.xml`
例 5-9「XML を使用したファイル拡張子のマッピングの削除」	`DeregisterContentTypeExtensions.xml`
例 5-10「Java を使用したファイル拡張子のマッピングの作成」	`RegisterContentTypeExtensions.java`
例 5-11「Java を使用したファイル拡張子のマッピングの削除」	`DeregisterContentTypeExtensions.java`
例 5-12「Java を使用したコンテンツ・タイプの削除」	`DeleteContentType.java`

Java サンプル・コード・ファイルの実行

Java サンプル・コード・ファイルを実行する手順は、次のとおりです。

1. Oracle 9*i*FS ホスト・マシン上で、CLASSPATH に Java API の JAR ファイルが含まれているかどうかを確認します。
2. JDK 1.2.2 で、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/devdoc/subclassing` ディレクトリにあるサンプル・コード・ファイルをコンパイルします。

例 5-13 Solaris でのサンプル・コード・ファイルのコンパイル

```
> cd $ORACLE_HOME/9ifs/samplecode/oracle/ifs/examples/devdoc/subclassing
> javac CreateContentType.java
```

3. ファイルのコメントをチェックし、このファイルの前に他のファイルの実行が予期されているかどうかを調べます。
4. クラスを実行して引数を指定します。引数の入力値が表示されます。すべての引数に値が必要です。値を指定しない場合は、一部またはすべての引数にデフォルトが設定されます。クラスにより結果が画面に出力されます。

例 5-14 Solaris でのサンプル・コード・ファイルの実行

```
> java oracle.ifs.examples.devdoc.subclassing.CreateContentType

  Running with arguments :
    user = system
    password = manager9ifs
    service = IfsDefault
    schemapassword = ifssys
  -----------Results----------
ClassObject Created : Image
```

XML サンプル・コード・ファイルの実行

XML 構成ファイルを実行する手順は、次のとおりです。

1. XML 構成ファイルの解析をサポートする Oracle 9*i*FS クライアントを開きます。
2. XML 構成ファイルにより実行されるタスクに該当する管理権限を持つユーザーでログインします。
3. XML 構成ファイルをインポートします。
4. 結果を Oracle 9*i*FS Manager で表示します。

6

任意のメタデータと動作の適用

この章では、コンテンツ・タイプに任意のメタデータと動作を適用する方法について説明します。この章の内容は、次のとおりです。

任意のメタデータと動作の適用の概要

Oracle 9*i*FS をビジネスに合せてカスタマイズする最初のステップは、通常は Oracle 9*i*FS で格納して管理する情報の物理タイプをすべて定義することです。インストール時のままの Oracle 9*i*FS では、ドキュメントやフォルダなど、標準的なタイプ情報が管理されます。Oracle 9*i*FS には、情報の物理構造（Content、Format、Size など）を定義する組込みコンテンツ・タイプ（Document および Folder など）と、情報を操作するためのインタフェース（`getContent()` および `listItems()` など）が用意されています。これらのコンテンツ・タイプを拡張し、Image や Book など、他の物理タイプの情報を定義できます。カスタム・タイプ階層を拡張して様々な物理タイプの情報の属性と動作を組み込む方法は、第 5 章「コンテンツ・タイプと属性の拡張」および第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照してください。

ただし、ビジネスのコンテンツ・タイプを定義するときに、情報に適用できる任意のメタデータと動作の管理も必要であることが判明する場合があります。メタデータと動作は、各種物理タイプの情報のうちの選択したインスタンスに適用しても適用しなくてもかまわないという点で任意です。たとえば、様々なコンテンツ・タイプの特定インスタンスを、ビジネスでの使用方法に従ってサブジェクト・カタログのような共通のメタデータ構造にカテゴリとして分類できます。また、各種コンテンツ・タイプやその特定のインスタンスに対して、特定の操作を実行する方法に関するビジネス・ルールを定義することもできます。

Oracle 9*i*FS には、情報に任意のメタデータと動作を適用できるように、次のメカニズムが用意されています。

- **Category:** 特定のドキュメント、フォルダおよび他のタイプの情報をカテゴリに分類し、そのカテゴリに関連する情報に追加の属性を適用します。
- **PropertyBundle:** 各種のコンテンツ・タイプやその特定インスタンスに適用できるデータのハッシュ表を作成します。
- **PolicyPropertyBundle:** 各種のコンテンツ・タイプやその特定インスタンスについて、操作の実行方法を定義するビジネス・ルール・セットを作成します。
- **Relationship:** 情報の様々な部分の間に多対多の対応付けを作成します。

この章では、Category、PropertyBundle、PolicyPropertyBundle および Relationship を作成して適用する手順について説明します。

Category

Category を使用すると、Oracle 9*i*FS に格納されている情報をビジネスでの使用方法に従って編成できます。たとえば、ドキュメント、フォルダおよびイメージを、使用するプロジェクトに従って編成できます。カテゴリに割り当てることで、そのカテゴリに関連する特別な属性と動作を追加できます。たとえば、Project カテゴリに属するドキュメントとフォルダに、属性 Project Name および Project Record Number を追加できます。

Oracle 9*i*FS には、カテゴリのスキーマを作成して情報に適用できるように、Category というコンテンツ・タイプが用意されています。Category は、拡張して様々なタイプのカテゴリ

（Project など）とそのカテゴリに関連する属性（Project Name、Project Record Number など）を定義できるコンテンツ・タイプです。

ドキュメント、フォルダおよびすべての PublicObject は、0 個または 1 個以上のカテゴリと対応付けることができます。PublicObject は、拡張 Category コンテンツ・タイプのインスタンスと対応付けることでカテゴリに分類できます。Category インスタンスには、そのカテゴリに属することで PublicObject に関連する特別な属性の値（Project Name = "Content Management Research Project"、Project Record Number = "AR1098a" など）が格納されます。Category インスタンスでは、属性 AssociatedPublicObject に PublicObject へのポインタも格納されます。

PublicObject（ドキュメント、フォルダ、イメージなど）は、同じカテゴリ・タイプに対応付けることができるため、カテゴリは様々な物理タイプの情報を編成するための共通スキーマを提供します。このように、カテゴリは様々なコンテンツ・タイプ間で属性を相互に継承する手段を提供します。

Category コンテンツ・タイプは PublicObject を拡張するため、Category インスタンスはフォルダに入れることができます。これが役立つのは、カスタム・ユーザー・インタフェースの構築中に、ドキュメントをフォルダごとに異なる方法で表示する必要がある場合です。たとえば、プロジェクトに関連するすべてのドキュメントを含む Project フォルダを作成できます。Document インスタンスをフォルダに入れて Name 順にリスト表示するのではなく、Project カテゴリ・インスタンスをフォルダに入れて Project Record Number 属性順にリスト表示できます。これにより、ユーザーは Category インスタンスで `getAssociatedPublicObject()` メソッドをコールして Document インスタンスにアクセスできます。

注意： このリリースの Oracle 9*i*FS の組込みのクライアントでは、Category をフォルダに入れる操作はサポートされません。Catetory をフォルダに入れる計画がある場合は、ユーザーがそれを表示できるようにカスタム・ユーザー・インタフェースを実装する必要があります。

Oracle 9*i*FS では、Category インスタンスはバージョニングされません。ドキュメントのカテゴリ・メタデータがバージョンごとに異なる場合は、各ドキュメントのバージョンを新規の Category インスタンスに対応付けることができます。ただし、バージョン情報はカテゴリではなくドキュメントでメンテナンスされます。

Category インスタンスへのアクセス権は、対応付けられた PublicObject に適用される AccessControlList により決定されます。PublicObject に対する `getAttribute()` が付与されているユーザーは、カテゴリの属性にもアクセスできます。

対応付けられている PublicObject が解放されると、Category は自動的に削除されます。この方法で Category インスタンスが削除される場合、通知は生成されません。

この項では、次の方法について説明します。

- 新規 Category タイプの定義
- PublicObject のカテゴリ分類
- カテゴリに基づく情報の検索

新規 Category タイプの定義

新しいタイプの Category を定義するには、カスタム・コンテンツ・タイプを定義する場合と同じ手順で操作する必要があります。他のコンテンツ・タイプの場合と同様に、Category タイプを表す SchemaObject インスタンスのセットを作成します。また、Java クラスを実装して、カテゴリ・タイプのカスタム動作を定義することもできます。ただし、新規カテゴリ・タイプを定義するためにカスタム Java クラスを実装する必要はありません。その後、SchemaObject を編集または削除して、カテゴリ・タイプを変更したり削除できます。このような開発タスクは、Java API、XML ファイルまたは Oracle 9*i*FS Manager を使用して実行できます。

注意： カスタム・コンテンツ・タイプを作成、変更および削除する手順の詳細は、第 5 章「コンテンツ・タイプと属性の拡張」および第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照してください。すでにこの 2 つの章を読んでいる場合は、この項で Java と XML を使用してカテゴリ・タイプを作成、変更および削除する方法をすばやく確認できます。

カテゴリ・タイプの作成

新しいタイプのカテゴリを作成するには、Category を拡張する新規コンテンツ・タイプを作成し、そのカテゴリに関連する属性を追加します。たとえば、Category を拡張する新規カテゴリ Project Record を作成し、拡張属性 Project Name および Project Record Number を含めることができます。

新規カテゴリ・タイプを定義する手順は、次のとおりです。

- 管理権限を持っているかどうかを確認します。XML 構成ファイルを使用する場合は、そのファイルを管理者としてインポートします。Java API を使用する場合は、Oracle 9*i*FS でのセッション用に管理モードを有効化します。
- カテゴリ・タイプを表す ClassObject のインスタンスを作成します。
- カテゴリ・タイプに関連する各属性を表す Attribute のインスタンスを作成します。
- カテゴリ・タイプ用のカスタム Java クラスを作成し、カスタム・メソッドを定義します。

注意： カテゴリ・タイプの Java クラスを拡張する方法は、第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照してください。

例 6-1 XML を使用したカテゴリ・タイプの作成

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <NAME>ProjectRecord</NAME>
    <SUPERCLASS RefType = "Name">Category</SUPERCLASS>
    <DESCRIPTION>Public Objects that pertain to a Project.</DESCRIPTION>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>ProjectName</NAME>
            <DATATYPE>String</DATATYPE>
        </ATTRIBUTE>
        <ATTRIBUTE>
            <NAME>ProjectRecordNumber</NAME>
            <DATATYPE>String</DATATYPE>
        </ATTRIBUTE>
    </ATTRIBUTES>
</CLASSOBJECT>
```

注意： カテゴリ・タイプの作成に使用する XML ファイルの各要素の詳細は、第 5 章「コンテンツ・タイプと属性の拡張」を参照してください。

例 6-2 Java を使用したカテゴリ・タイプの作成

1. 管理モードを有効化します。

   ```
   session.setAdministrationMode(true);
   ```

2. カテゴリ・タイプを表す ClassObject の値を保持する ClassObjectDefinition を作成します。

   ```
   ClassObjectDefinition codef = new ClassObjectDefinition(session);
   codef.setAttributeByUpperCaseName(ClassObject.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("ProjectRecord"));
   ```

3. 新規カテゴリ・タイプで Category コンテンツ・タイプを拡張するように指定します。

   ```
   ClassObject catco = session.getClassObjectByName("CATEGORY");
   codef.setSuperclass(catco);
   ```

4. カテゴリ・タイプに関連する属性ごとに AttributeDefinition を作成します。AttributeDefinition を ClassObjectDefinition に追加します。

```
AttributeDefinition attdef1 = new AttributeDefinition(session);
attdef1.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("ProjectName"));
attdef1.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_STRING));

codef.addAttributeDefinition(attdef1);

AttributeDefinition attdef2 = new AttributeDefinition(session);
attdef2.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("ProjectRecordNumber"));
attdef2.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_STRING));

codef.addAttributeDefinition(attdef2);
```

5. 新規カテゴリ・タイプの ClassObject および Attribute を作成します。

```
ClassObject mycatco = (ClassObject) session.createSchemaObject(codef);
```

カテゴリ・タイプの変更

作成後のカテゴリ・タイプの属性を追加したり削除できます。

注意： カテゴリ・タイプの属性の追加や削除には、XML を使用できません。この操作には、Oracle 9*i*FS Manager または Java API を使用する必要があります。Oracle 9*i*FS Manager で属性を削除する手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

例 6-3 Java を使用したカテゴリの属性の追加、削除および更新

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. カテゴリ・タイプを表す ClassObject を取得します。

```
ClassObject catco = session.getClassObjectByName("PROJECTRECORD");
```

3. Attribute を ClassObject に追加します。

```
AttributeDefinition attdef1 = new AttributeDefinition(session);
attdef1.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("ProjectPriority"));
attdef1.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_INTEGER));
catco.addAttribute(attdef1);
```

4. Attribute を ClassObject から削除します。

```
Attribute att2 = catco.getEffectiveClassAttributes("PROJECTRECORDNUMBER");
catco.removeAttribute(att2);
```

5. ClassObject の Attribute を更新します。

```
Attribute att3 = catco.getEffectiveClassAttributes("PROJECTPRIORITY");
Collection vdColl = session.getValueDomainCollection();
ValueDomain vd = (ValueDomain) vdColl.getItems("Projects");
att3.setValueDomain(vd);
```

カテゴリ・タイプの削除

Java API または Oracle 9*i*FS Manager を使用して、カテゴリ・タイプを削除できます。Category のインスタンスが存在する場合は、Category タイプを削除する前に、そのインスタンスを削除する必要があります。

注意： Oracle 9*i*FS Manager でカテゴリ・タイプの ClassObject を削除する手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

例 6-4 Java を使用したカテゴリの削除

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. カテゴリ・タイプを表す ClassObject を取得します。

```
ClassObject catco = session.getClassObjectByName("PROJECTRECORD");
```

3. `free()` メソッドをコールして ClassObject を削除します。

```
catco.free();
```

PublicObject のカテゴリ分類

新規 Category タイプの作成後に、それを使用して Oracle 9*i*FS 内のドキュメント、フォルダまたはすべての PublicObject をカテゴリに分類できます。この項では、次のタスクについて説明します。

- Category インスタンスの作成
- Category インスタンスの更新
- Category インスタンスの削除

Category インスタンスの作成

PublicObject をカテゴリに分類するには、カテゴリ・タイプの新規インスタンスを作成し、それを PublicObject に対応付けます。Category インスタンスは、新規 PublicObject の作成時に適用するか、Oracle 9*i*FS の既存の PublicObject に適用できます。

Java を使用した PublicObject のカテゴリ分類： Java API で PublicObject をカテゴリに分類する手順は、次のとおりです。

1. 新規 Category インスタンスの値を保持する CategoryDefinition を作成します。
2. 新規 Category インスタンスのタイプを指定します。
3. Category の属性値を設定します。
4. PublicObject を Oracle 9*i*FS での作成時にカテゴリに分類する場合は、PublicObjectDefinition の `addCategoryDefinition()` メソッドに CategoryDefinition を渡します。PublicObjectDefinition は、PublicObject および Category の作成に使用されます。
5. Oracle 9*i*FS の既存の PublicObject をカテゴリに分類する場合は、PublicObject の `addCategory()` メソッドに CategoryDefinition を渡します。

例 6-5 Java を使用した PublicObject のカテゴリ分類

1. 新規 CategoryDefinition を作成します。

```
CategoryDefinition cdef = new CategoryDefinition(session);
```

2. 新規 Category インスタンスのタイプを指定します。

```
ClassObject co = session.getClassObjectByName("PROJECTRECORD");
cdef.setClassObject(co);
```

3. Category の属性値を設定します。

```
cdef.setAttributeByUpperCaseName(Category.NAME_ATTRIBUTE,
         AttributeValue.newAttributeValue(
           "Content Management Marketing Analysis Report"));
cdef.setAttributeByUpperCaseName("PROJECTNAME",
```

```
        AttributeValue.newAttributeValue("Content Management Research Project"));
    cdef.setAttributeByUpperCaseName("PROJECTRECORDNUMBER",
        AttributeValue.newAttributeValue("AR1098a"));
```

4. PublicObjectDefinition に CategoryDefinition を追加して、新規 PublicObject をカテゴリに分類します。

```
FolderDefinition fdef = new FolderDefinition(session);
fdef.setAttributeByUpperCaseName(Folder.NAME_ATTRIBUTE,
    AttributeValue.newAttributeValue("Collateral"));
fdef.addCategoryDefinition(cdef);
Folder f = (Folder) session.createPublicObject(fdef);
```

5. または、CategoryDefinition を使用して既存の PublicObject をカテゴリに分類します。

```
f.addCategory(cdef);
```

XML を使用した PublicObject のカテゴリ分類： XML を使用して新規 PublicObject をカテゴリに分類することもできます。XML ファイルを Oracle 9*i*FS にインポートすると、IfsSimpleXmlParser によりファイルが自動的に解析され、PublicObject のカテゴリ・メタデータを保持できるようにカテゴリ・タイプのインスタンスが生成されます。PublicObject のカテゴリ分類に使用する XML ファイルの主な側面は、次のとおりです。

- XML 構成ファイルのルート要素で、新規 PublicObject を指定する必要があります。
- ルート要素のスキーマには、新規 PublicObject の Category の指定に使用する特殊タグ <CATEGORIES> を含めることができます。
- 各 Category は、<CATEGORIES> 要素の副要素として指定できます。副要素名は、カテゴリ・タイプ名に対応させる必要があります。
- Category 要素には、各カテゴリ属性の副要素が含まれます。これらの要素名は、属性名に対応させる必要があります。

注意： 既存の PublicObject のカテゴリ分類には、XML は使用できません。

例 6-6 XML を使用した新規 PublicObject のカテゴリ分類

```
<?xml version="1.0" standalone="yes"?>
  <FOLDER>
    <NAME>CMProject</NAME>
    <FOLDERPATH>/home/guest</FOLDERPATH>
    <CATEGORIES>
      <PROJECTRECORD>
        <NAME>Content Management Market Analysis</NAME>
```

```
      <PROJECTNAME>Content Management Research Project</PROJECTNAME>
    </PROJECTRECORD>
    <!-- You can have multiple Category elements here -->
  </CATEGORIES>
</FOLDER>
```

注意： XML 構成ファイルの構文規則の詳細は、第 10 章「XML および Oracle Internet File System」を参照してください。

Category インスタンスの更新

PublicObject をカテゴリに分類した後に、Category インスタンスにより適用される属性の値の更新が必要になる場合があります。PublicObject のカテゴリの属性の更新には、XML 構成ファイルを使用する方法と、Java API を使用する方法があります。

XML を使用した Category インスタンスの更新： Category インスタンスを更新するには、次の要件を満たす XML ファイルが必要です。

- ルート要素が Category タイプの Name に対応していること。
- 変更対象となる Category インスタンスを <UPDATE> 副要素で参照していること。
- 変更対象となるすべての属性の副要素を含んでいること。これらの要素の値には、更新後の属性値が保持されます。

例 6-7 XML を使用した Category インスタンスの属性の更新

```
<?xml version="1.0" standalone="yes"?>
<PROJECTRECORD>
    <UPDATE RefType = "Name">Content Management Market Analysis</UPDATE>
    <PROJECTNAME>XML Research Project</PROJECTNAME>
</PROJECTRECORD>
```

Java を使用した Category インスタンスの更新： Category インスタンスの属性の値を更新するには、Oracle 9*i*FS Java API を使用することもできます。拡張 Category タイプは、それぞれ Category クラスからメソッド `getAttribute()` および `setAttribute()` を継承します。この 2 つのメソッドは、あらゆる属性値を操作する汎用的な方法を提供します。

また、カスタム・カテゴリ・タイプの拡張属性の値を取得および設定する便利なメソッドを提供するために、カスタム Java クラスを実装することもできます。カスタム Java クラスの実装手順は、第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照してください。

例 6-8 Java を使用した Category インスタンスの更新

1. アプリケーションですでに Category インスタンスを取得しているとします。

```
Category cat = ......
```

2. ProjectRecord インスタンスの ProjectPriority 属性を更新します。

```
cat.setAttribute("PROJECTPRIORITY",
                AttributeValue.newAttributeValue(new Integer(3)));
```

Category インスタンスの削除

PublicObject のカテゴリの属性値を保持する Category インスタンスを削除すると、PublicObject をカテゴリから削除できます。PublicObject を削除すると、そのすべてのカテゴリが Oracle 9*i*FS により自動的に削除されます。

Category インスタンスは、他のオブジェクトの場合と同様に、Oracle 9*i*FS Java API で `free()` メソッドをコールして削除できます。Oracle 9*i*FS では、カテゴリから PublicObject を削除する手段としての XML はサポートされず、Oracle 9*i*FS Manager には Category インスタンスを削除するためのユーザー・インタフェースは用意されていません。

例 6-9 Java を使用した Category インスタンスの削除

1. アプリケーションですでに Category インスタンスを取得しているとします。

```
Selector s = new Selector(session);
s.setSearchClassname("PROJECTRECORD");

LibraryObject[] cats = s.getItems();
Category c;
```

2. Category インスタンスを削除します。

```
int i;
int count = (cats == null) ? 0 : cats.length;

for (i = 0; i < count; i++)
{
   c = (Category) cats[i];
   c.free();
}
```

カテゴリに基づく情報の検索

ドキュメント、フォルダおよび他のタイプの情報をカテゴリに分類した後で、カテゴリとカテゴリ属性に基づいて情報を検索できます。

たとえば、Content Management Research Project のプロジェクト・レコードであるドキュメント、フォルダおよび他のタイプの情報をすべて検索できます。そのためには、Project Record カテゴリに分類され、カテゴリ属性 Project Name が Content Management Research Project に設定されている PublicObject をすべて検索します。

Oracle 9*i*FS には、情報とそのカテゴリ間の対応付けを簡単に横断できるように API も用意されています。たとえば、PublicObject インスタンスの場合は、対応付けられているすべてのカテゴリをフェッチできます。逆に、Category インスタンスの場合は、対応付けられている PublicObject をフェッチできます。

カテゴリに基づく PublicObject の検索

Oracle 9*i*FS Java API で Selector または Search を構成すると、カテゴリ・メタデータに基づいて PublicObject を検索するアプリケーションを構築できます。問合せを構成するために、Search または Selector は Category の AssociatedPublicObject 属性と PublicObject の ID 属性との間の参照に基づいて、Category および PublicObject コンテンツ・タイプを結合します。これで、Search または Selector は、Category または PublicObject、あるいは両方の属性について条件を追加できます。

注意： Selector と Search の作成手順の詳細は、第 8 章「検索アプリケーションの構築」を参照してください。

例 6-10 Selector を使用した PublicObject の特定カテゴリの検索

1. アプリケーションですでに PublicObject を取得しているとします。

```
PublicObject po = .....
```

2. Selector を構成します。

```
Selector s = new Selector(session);
s.setSearchClassname("PROJECTRECORD");
s.setSearchSelection("ASSOCIATEDPUBLICOBJECT = " + po.getId() +
                     " AND PROJECTNAME = 'XML Research Project'");
```

3. 問合せを実行し、PublicObject の各カテゴリを取得します。

```
LibraryObject[] cats = s.getItems();

int i;
int count = (cats == null) ? 0 : cats.length;
for (i = 0; i < count; i++)
{
   c = (Category) cats[i];
}
```

例 6-11 Search を使用した共通カテゴリ属性に基づく全 PublicObject の検索

1. 問合せ条件を保持する SearchSpecification を構成します。次の例には内容条件が含まれていないため、AttributeSearchSpecification を構成します。

```
AttributeSearchSpecification asp = new AttributeSearchSpecification();
```

2. SearchClassSpecification を構成し、AttributeSearchSpecification で設定します。

```
String[] searchClasses = {"PUBLICOBJECT", "PROJECTRECORD"};
SearchClassSpecification scp = new SearchClassSpecification(searchClasses);
scp.addResultClass("PUBLICOBJECT");

asp.setSearchClassSpecification(scp);
```

3. Category タイプ ProjectRecord を PublicObject コンテンツ・タイプと結合する JoinQualification を構成します。

```
JoinQualification jq = new JoinQualification();
jq.setLeftAttribute("PUBLICOBJECT", null);
jq.setRightAttribute("PROJECTRECORD", "ASSOCIATEDPUBLICOBJECT");
```

4. AttributeQualification を構成し、カテゴリの ProjectName 属性は必ず Content Management Research Project であるという条件を指定します。

```
AttributeQualification aq = new AttributeQualification();
aq.setAttribute("PROJECTRECORD", "PROJECTNAME");
aq.setOperatorType(AttributeQualification.EQUAL);
aq.setValue("Content Management Research Project");
```

5. SearchClause を構成し、Join および Attribute 修飾を結合し、両方の条件が必ず満たされるように要求します。これを使用して SearchQualification を設定します。

```
SearchClause sc = new SearchClause(aq, jq, SearchClause.AND);
asp.setSearchQualification(sc);
```

6. Search を実行し、結果をループして条件を満たす LibraryObject を取り出します。戻される LibraryObject は、SearchClassSpecification に指定したコンテンツ・タイプのものです。

```
Search srch = new Search(session, asp);
srch.open();

int i;
LibraryObject lo;
SearchResultObject sro = srch.next();
String lName;

while (sro != null)
{
```

```
      lo = sro.getLibraryObject();
      lName = lo.getName();

    try
    {
       sro = srch.next();
     }
     catch (IfsException e)
     {
       if (e.getErrorCode() == 22000)
       {
         break;
      }
      else
      {
          e.printStackTrace();
      }
    }
  }
  srch.close();
```

PublicObject のカテゴリのフェッチ

Oracle 9*i*FS Java API には、PublicObject のカテゴリと Category インスタンスに対応付けられている PublicObject を簡単に取得できるようにメソッドが用意されています。

PublicObject クラスにはメソッド `getCategories()` が含まれており、このメソッドは PublicObject に対応付けられている全 Category インスタンスを戻します。Category インスタンスの取得後は、そのカテゴリで PublicObject に適用される特別な属性を取り出すことができます。

例 6-12 PublicObject のカテゴリのフェッチ

1. アプリケーションですでに PublicObject を取得しているとします。

   ```
   PublicObject po = .....
   ```

2. PublicObject の Category インスタンスをフェッチします。

   ```
   Category[] cats = po.getCategories();
   ```

3. Category インスタンスごとに、カテゴリ・タイプを判断してカテゴリ属性をフェッチします。

   ```
   AttributeValue av; // variable to hold value of a category attribute
   Attribute[] catatts; // variable to hold the Attributes on the Category
   Category c; // variable to hold each Category instance
   ClassObject catco; // variable to hold the ClassObject
                      // that represents the category type
   ```

```
int i, j, icount, jcount;
String catName, attName; // variables to hold the names of
                         //the category and attributes

icount = (cats == null) ? 0 : cats.length;
for(i=0; i < icount; i++)
{
   c = cats[i];
   catco = c.getClassObject();
   catName = catco.getName();
   catatts = catco.getExtendedClassAttributes();
   jcount = (catatts == null) ? 0 : catatts.length;
   for (j=0; j < jcount; j++)
   {
       attName = catatts[j].getName();
       av = c.getAttribute(attName);
    }
 }
```

Category クラスには `getAssociatedPublicObject()` というメソッドが用意されており、カテゴリ・メタデータが適用される PublicObject インスタンスを戻します。PublicObject インスタンスの取得後は、その PublicObject が持つ他のすべての属性を取り出し、メソッドをコールして PublicObject を操作できます。

例 6-13 Category インスタンスからの対応付けられている PublicObject のフェッチ

1. アプリケーションですでに Category インスタンスを取得しているとします。

```
Category cat = ......
```

2. Category に対応付けられている PublicObject をフェッチします。

```
PublicObject po = cat.getAssociatedPublicObject();
```

3. PublicObject のコンテンツ・タイプと属性をフェッチします。

```
ClassObject co = po.getClassObject();
Attribute[] atts = co.getEffectiveClassAttributes();
String coName = co.getName();

AttributeValue av; //variable to hold value of a PublicObject's attribute
int i, icount;
String attName;

icount = (atts == null) ? 0 : atts.length;
for (i = 0; i < icount; i++)
{
   attName = atts[i].getName();
```

```
        av = po.getAttribute(attname);

    }
```

PropertyBundle

PropertyBundle は、Oracle 9*i*FS でPropertyのセットを永続的に格納する場合に役立ちます。

カテゴリに比べると、PropertyBundle は比較的緩やかな構造を持っています。PropertyBundle は、それぞれが名前 / 値のペアからなる Property のセットで構成されています。PropertyBundle は Oracle 9*i*FS に独立して存在でき、0 個または 1 個以上の PublicObject に対応付けられています。PropertyBundle の方がデータの格納方法としては柔軟ですが、カテゴリのように構造化されている構造体の方が操作が簡単です。

PropertyBundle の使用

PropertyBundle を使用すると、システム単位のタスク、1 つ以上のコンテンツ・タイプまたは特定のインスタンスの処理用に、非定型のメタデータを格納できます。

- **システム単位の PropertyBundle:** PropertyBundle は Oracle 9*i*FS に独立して存在でき、アプリケーションで処理ロジックの決定に使用できます。たとえば、Oracle 9*i*FS では、ファイル拡張子とコンテンツ・タイプ間のマッピングは PropertyBundle に格納されます。Oracle 9*i*FS では、この PropertyBundle を使用して、ファイルを特定のコンテンツ・タイプのインスタンスとして格納するかどうかが判断されます。
- **コンテンツ・タイプの PropertyBundle:** PropertyBundle をコンテンツ・タイプの ClassObject に対応付けて、コンテンツ・タイプとそのすべてのインスタンスの処理に使用できます。ClassObject は PropertyBundle 属性を持ち、この属性はコンテンツ・タイプの処理に使用される PropertyBundle を参照します。
- **インスタンスの PropertyBundle:** PropertyBundle を使用すると、ドキュメントのローカライズ名のリスト（English/Guide、Spanish/Guida など）のように、コンテンツ・タイプの特定インスタンスに関する非定型のメタデータを格納できます。各 PublicObject は属性 PropertyBundle を持ち、この属性は非定型のメタデータを保持する PropertyBundle を参照します。

PropertyBundle は Oracle 9*i*FS に独立して存在でき、通常はアプリケーションの処理に使用されるため、PropertyBundle を ValueDefault に対応付けると役立ちます。PropertyBundle を ValueDefault に対応付けると、文字列、つまり ValueDefault の UniqueName で一意に参照できます。ValueDefault などの SchemaObject とは異なり、PropertyBundle は UniqueName 属性を持ちません。PropertyBundle は一意 ID を持ちますが、ID 属性には PropertyBundle を参照する便利な方法が用意されていません。ValueDefault を使用すると、PropertyBundle を簡単に参照できます。たとえば、XML 構成ファイルでは、ValueDefault の UniqueName を、PropertyBundle を参照する要素の値として指定できます（<Update RefType = "ValueDefault">ContentTypeLookupByFileExtension</Update> など）。また、Oracle 9*i*FS Java API には、ValueDefault 経由でオブジェクトを取得できるように特

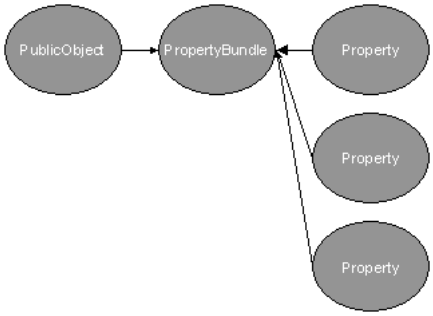
殊なメソッドも用意されています（`LibrarySession.getValueDefaultCollection()` など）。

注意： ValueDefault の詳細は、第 7 章「属性の妥当性チェック」を参照してください。

PropertyBundle の構造

PropertyBundle は、名前 / 値のペアの配列で構成されています。各ペアは、Name 属性と Value 属性を持つ Property オブジェクトで表されます。Value 属性は、Oracle 9*i*FS でサポートされているデータ型であれば、どのような型でもかまいません（STRING、INTEGER、PublicObject、DirectoryObject_Array など）。Property オブジェクトは、それが属している PropertyBundle を参照する第 3 の属性 Bundle を持ちます。

図 6-1 PropertyBundle オブジェクト・モデル



Oracle 9*i*FS のすべてのオブジェクトは、PropertyBundle 属性を持っています。PublicObject インスタンスは、継承した属性を使用して、特定のコンテンツ・タイプのインスタンスの処理に使用する PropertyBundle を対応付けることができます。ClassObject インスタンスは、継承した属性を使用して、ClassObject が表すコンテンツ・タイプの全インスタンスの処理に使用する PropertyBundle を対応付けることができます。

PropertyBundle の定義

Oracle 9*i*FS SDK では、PropertyBundle を定義する方法として次の 2 つがサポートされています。

- **XML:** Oracle 9*i*FS へのインポート時に PropertyBundle を自動的に定義する XML ファイルを作成します。

- **Java:** Oracle 9*i*FS Java API 経由で PropertyBundle をプログラムで定義します。

注意： 現在、Oracle 9*i*FS Manager には PropertyBundle を定義するためのインタフェースは用意されていません。ただし、PropertyBundle 情報を検索して表示することはできます。Oracle 9*i*FS Manager の検索インタフェースの使用方法は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

XML を使用した PropertyBundle の作成および変更

XML ファイルを使用して、Oracle 9*i*FS で PropertyBundle を定義できます。XML ファイルを Oracle 9*i*FS にインポートすると、IfsSimpleXmlParser では自動的にそのファイルが使用され、PropertyBundle インスタンスが作成または変更されます。Oracle 9*i*FS では、XML を使用した PropertyBundle の削除はサポートされていません。

PropertyBundle を文字列で一意に参照できるように、PropertyBundle ごとに ValueDefault を作成することをお薦めします。次の例に、XML の <OBJECTLIST> タグを使用して XML ファイルに PropertyBundle と ValueDefault を作成する方法と、ValueDefault を使用して XML ファイルで PropertyBundle を更新する方法を示します。

例 6-14 XML を使用した PropertyBundle の作成

```
<?xml version='1.0' standalone='yes'?>
<OBJECTLIST>

 <PROPERTYBUNDLE>
   <NAME>Title Translation PropertyBundle</NAME>
   <PROPERTIES>
     <PROPERTY>
       <NAME>English</NAME>
       <VALUE>Guide</VALUE>
     </PROPERTY>
     <PROPERTY>
       <NAME>Italian</NAME>
       <VALUE>Guida</VALUE>
     </PROPERTY>
   </PROPERTIES>
 </PROPERTYBUNDLE>

 <VALUEDEFAULT>
   <NAME>TranslatedTitleValueDefault</NAME>
   <VDVALUE Datatype="PublicObject" ClassName = "PropertyBundle" RefType = "Name">
     Title Translation PropertyBundle
   </VDVALUE>
```

```
 </VALUEDEFAULT>

</OBJECTLIST>
```

例 6-15 XML を使用した PropertyBundle の変更

```
<?xml version='1.0' standalone='yes'?>
 <PROPERTYBUNDLE>
  <UPDATE RefType='ValueDefault'>TranslatedTitleValueDefault</UPDATE>
   <PROPERTIES>
    <PROPERTY Action = 'add'>
     <NAME> Spanish </NAME>
      <VALUE DataType = 'String'>
          Guida
      </VALUE>
   </PROPERTY>
   <PROPERTY Action = 'Remove'>
      <NAME> Italian </NAME>
   </PROPERTY>
   <PROPERTY Action = 'Add'>
     <NAME> English </NAME>
      <VALUE DataType = 'String'>
          Manual
      </VALUE>
   </PROPERTY>
  </PROPERTIES>
 </PROPERTYBUNDLE>
```

XML 構成ファイルの各要素を使用して、PropertyBundle および Property インスタンスの属性の値を指定します。

注意： XML 構成ファイルの構文規則の詳細は、第 10 章「XML および Oracle Internet File System」を参照してください。

表 6-1 に、PropertyBundle 構成ファイルの各 XML 要素を示します。この表は、Java クラスとその要素に対応する Oracle 9*i*FS Java API の属性、各要素の使用方法および要素の有効な XML 属性を示しています。

表 6-1 PropertyBundle の XML 構成ファイルの要素

要素	クラス情報	属性	使用方法
<PROPERTYBUNDLE>	PropertyBundle が PublicObject のサブクラス ApplicationObject を拡張		PropertyBundle 構成ファイルのルート要素。PropertyBundle クラスの名前に対応する必要があります。
<PROPERTIES>	PropertyBundle クラスの属性との直接の対応関係はなし		PropertyBundle の Property の宣言は、<PROPERTIES> 要素で囲む必要があります。
<PROPERTY>	Property が SystemObject を拡張	Action	各ポリシーは <PROPERTY> 要素で参照されます。
<NAME>	LibraryObject スーパークラスから継承される、Property クラスの Name 属性		<NAME> 要素は、まとまりの各プロパティの名前を指定するために使用されます。
<VALUE>	Property クラスの Value 属性	データ型 RefType	<VALUE> 要素は、Property の値を保持するために使用されます。<VALUE> 要素の Datatype 属性は、Property クラスの DataType 属性に対応します。 <VALUE> 要素の RefType 属性を使用すると、既存のオブジェクト（PublicObject、SystemObject、SchemaObject、DirectoryObject）を参照できます。

Java を使用した PropertyBundle の作成、変更および削除

Oracle 9*i*FS Java API には、PropertyBundle の操作用に、より確実なインタフェースが用意されています。この Java API には、PropertyBundle に格納された非定型データを操作するためのメソッドを持つ、クラス PropertyBundle および Property のセットが含まれています。次の例に、Java API で PropertyBundle を作成、変更および削除する方法を示します。

PropertyBundle を文字列で一意に参照できるように、PropertyBundle ごとに ValueDefault を作成することをお薦めします。次の例では、新規 PropertyBundle 用の ValueDefault を作成する方法と、ValueDefault を使用して PropertyBundle にアクセスする方法も示します。

例 6-16 Java を使用した PropertyBundle の作成

1. 値を一時的に保持するように PropertyBundleDefinition を構成します。

```
PropertyBundleDefinition pbdef = new PropertyBundleDefinition(session);
```

```
pbdef.setAttribute(PropertyBundle.NAME_ATTRIBUTE,
   AttributeValue.newAttributeValue("Title Translation PropertyBundle"));
```

2. PropertyBundleDefinition の addPropertyValue() メソッドを使用して、新規 Property を定義します。

```
pbdef.addPropertyValue("English", AttributeValue.newAttributeValue("Guide"));
```

3. 新規 Property を追加するように PropertyDefinition を構成します。

```
PropertyDefinition pd = new PropertyDefinition(session);
pd.setName("Italian");
pd.setValue(AttributeValue.newAttributeValue("Guida"));
pbdef.addPropertyDefinition(pd);
```

4. PropertyBundleDefinition を LibrarySession に渡して、新規の PropertyBundle および Property インスタンスを作成します。

```
PropertyBundle pb = (PropertyBundle) session.createPublicObject(pbdef);
```

5. ValueDefault に PropertyBundle を保持するように ValueDefaultPropertyBundle を作成します。

```
ValueDefaultPropertyBundleDefinition vdpbd =
    new ValueDefaultPropertyBundleDefinition(session);
vdpbd.setAttributeByUpperCaseName(ValueDefaultPropertyBundle.NAME_ATTRIBUTE,
                            AttributeValue.newAttributeValue(
                                 "TranslatedTitleValueDefaultPropertyBundle"));
vdpbd.setValue(pb);

ValueDefaultPropertyBundle vdpb = (ValueDefaultPropertyBundle)
session.createPublicObject(vdpbd);
```

6. PropertyBundle を文字列で参照できるように ValueDefault を作成します。ValueDefault は SchemaObject を拡張するため、ValueDefault を作成するには管理モードを有効化する必要があります。

```
session.setAdministrationMode(true);

ValueDefaultDefinition vdd = new ValueDefaultDefinition(session);
vdd.setAttributeByUpperCaseName(ValueDefault.NAME_ATTRIBUTE,
                           AttributeValue.newAttributeValue(
                                "TranslatedTitleValueDefault"));

ValueDefault vd = (ValueDefault) session.createSchemaObject(vdd);
vd.setValueDefaultPropertyBundle(vdpb);
```

例 6-17 Java を使用した PropertyBundle の変更

1. 変更する PropertyBundle を取得します。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd = (ValueDefault) vdColl.getItems("TranslatedTitleValueDefault");
AttributeValue av = vd.getPropertyValue();
PropertyBundle pb = (PropertyBundle) av.getPublicObject(session);
```

2. PropertyBundle の新規 Property を追加します。

```
pb.putPropertyValue("Spanish", AttributeValue.newAttributeValue("Guida"));
```

3. PropertyBundle から新規 Property を削除します。

```
pb.removePropertyValue("Italian");
```

4. PropertyBundle の Property を変更します。

```
pb.putPropertyValue("English",
                    AttributeValue.newAttributeValue("Manual"));
```

例 6-18 Java を使用した PropertyBundle の削除

1. 変更する PropertyBundle を取得します。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd = (ValueDefault) vdColl.getItems("TranslatedTitleValueDefault");
AttributeValue av = vd.getPropertyValue();
PropertyBundle pb = (PropertyBundle) av.getPublicObject(session);
```

2. ValueDefault を削除します。管理モードを有効化している必要があります。

```
session.setAdministrationMode(true);
vd.free();
```

3. PropertyBundle を削除します。

```
pb.free();
```

コンテンツ・タイプとインスタンスへの PropertyBundle の適用

PropertyBundle は Oracle 9*i*FS に独立して存在し、アプリケーションで使用される他の PublicObject に対応付ける必要はありません。ただし、PropertyBundle を 1 つ以上のコンテンツ・タイプまたはその特定インスタンスに対応付けることができます。

Oracle 9*i*FS では、各コンテンツ・タイプとインスタンスは PropertyBundle 属性を持っています。この属性で、特別なメタデータを適用する PropertyBundle を参照できます。PropertyBundle をコンテンツ・タイプに対応付けるには、そのコンテンツ・タイプを表す ClassObject インスタンスの PropertyBundle 属性の値を設定します。PropertyBundle をド

キュメントに対応付けるには、PropertyBundle を参照する Document インスタンスの PropertyBundle 属性の値を設定します。PropertyBundle とコンテンツ・タイプまたはインスタンスとの対応付けを解除するには、PropertyBundle 属性を NULL に設定します。

Oracle 9*i*FS では、XML 構成ファイルまたは Java API を使用して、PropertyBundle をコンテンツ・タイプおよびインスタンスに適用できます。以降にその方法を示します。

XML を使用した PropertyBundle の適用

XML 構成ファイルを使用すると、次のようにいつでも PropertyBundle をコンテンツ・タイプおよびインスタンスに適用できます。

- 新規 PropertyBundle の作成時
- 新規コンテンツ・タイプまたはインスタンスの作成時
- PropertyBundle、コンテンツ・タイプまたはインスタンスが存在する場合

例 6-19 XML を使用した新規コンテンツ・タイプへの新規 PropertyBundle の適用

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <NAME>Image</NAME>
    <SUPERCLASS RefType = "Name">Document</SUPERCLASS>
    <PROPERTYBUNDLE>
      <NAME>Title Translation PropertyBundle</NAME>
      <PROPERTIES>
        <PROPERTY>
          <NAME>English</NAME>
          <VALUE>Guide</VALUE>
        </PROPERTY>
        <PROPERTY>
          <NAME>Italian</NAME>
          <VALUE>Guida</VALUE>
        </PROPERTY>
      </PROPERTIES>
    </PROPERTYBUNDLE>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>Artists</NAME>
            <DATATYPE>Integer</DATATYPE>
        </ATTRIBUTE>
     </ATTRIBUTES>
</CLASSOBJECT>
```

例 6-20 XML を使用した新規コンテンツ・タイプへの既存 PropertyBundle の適用

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
```

```
    <NAME>Image</NAME>
    <SUPERCLASS RefType = "Name">Document</SUPERCLASS>
    <PROPERTYBUNDLE RefType = "ValueDefault">
      TranslatedTitleValueDefault
    </PROPERTYBUNDLE>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>Artists</NAME>
            <DATATYPE>Integer</DATATYPE>
        </ATTRIBUTE>
     </ATTRIBUTES>
</CLASSOBJECT>
```

例 6-21 XML を使用した既存コンテンツ・タイプへの既存 PropertyBundle の適用

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <UPDATE RefType = "Name">IMAGE</UPDATE>
    <PROPERTYBUNDLE RefType = "ValueDefault">
        TranslatedTitleValueDefault
    </PROPERTYBUNDLE>
</CLASSOBJECT>
```

例 6-22 XML を使用した新規フォルダへの既存 PropertyBundle の適用

```
<?xml version="1.0" standalone="yes"?>
<FOLDER>
    <NAME>Collateral</NAME>
    <PROPERTYBUNDLE RefType = "ValueDefault">
      TranslatedTitleValueDefault
    </PROPERTYBUNDLE>
</FOLDER>
```

例 6-23 XML を使用した既存フォルダへの既存 PropertyBundle の適用

```
<?xml version="1.0" standalone="yes"?>
<FOLDER>
    <UPDATE RefType = "Path">/home/guest/Collateral</UPDATE>
    <PROPERTYBUNDLE RefType = "ValueDefault">
      TranslatedTitleValueDefault
    </PROPERTYBUNDLE>
</FOLDER>
```

Java を使用したコンテンツ・タイプおよびインスタンスへの PropertyBundle の適用

Java API を使用すると、コンテンツ・タイプとインスタンスに対する PropertyBundle の適用方法を厳密に制御できます。たとえば、インスタンスが特定の条件を満たしている場合に、PropertyBundle をプログラムで適用できます。また、一方を作成できない場合はどちらも作成されないように、PolicyPropertyBundle および PublicObject を単一トランザクションで作成することもできます。

注意： Oracle 9*i*FS でトランザクションを管理する手順は、第 16 章「セッションおよびトランザクションの管理」を参照してください。

例 6-24 Java を使用した新規コンテンツ・タイプへの新規 PropertyBundle の適用

1. PropertyBundle を作成します。

```
PropertyBundleDefinition pbDef = new PropertyBundleDefinition(session);
pbDef.setAttributeByUpperCaseName(PropertyBundle.NAME_ATTRIBUTE,
      AttributeValue.newAttributeValue("Title Translation PropertyBundle"));
pbDef.addPropertyValue("English", AttributeValue.newAttributeValue("Guide"));
pbDef.addPropertyValue("Italian", AttributeValue.newAttributeValue("Guida"));

PropertyBundle pb = (PropertyBundle) session.createPublicObject(pbDef);
```

2. ClassObjectDefinition を作成します。管理モードを有効化している必要があります。

```
session.setAdministrationMode(true);

ClassObjectDefinition coDef = new ClassObjectDefinition(session);
coDef.setAttributeByUpperCaseName(ClassObject.NAME_ATTRIBUTE,
       AttributeValue.newAttributeValue("IMAGE"));
ClassObject superClass = session.getClassObjectByName("DOCUMENT");
coDef.setSuperclass(superClass);
```

3. ClassObjectDefinition の PropertyBundle 属性を、新規 PropertyBundle を参照するように設定します。

```
coDef.setAttributeByUpperCaseName(ClassObject.PROPERTYBUNDLE_ATTRIBUTE,
        AttributeValue.newAttributeValue(pb));

ClassObject classObj = (ClassObject) session.createSchemaObject(coDef);
```

例 6-25 Java を使用した既存コンテンツ・タイプへの PropertyBundle の適用

1. ValueDefault 経由で PropertyBundle をフェッチします。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd = (ValueDefault) vdColl.getItems("TranslatedTitleValueDefault");
AttributeValue av = vd.getPropertyValue();
PropertyBundle pb = (PropertyBundle) av.getPublicObject(session);
```

2. ClassObject をフェッチします。

```
ClassObject co = session.getClassObjectByName("IMAGE");
```

3. PropertyBundle 属性を、PropertyBundle を参照するように設定します。管理モードを有効化している必要があります。

```
session.setAdministrationMode(true);

co.setAttribute(ClassObject.PROPERTYBUNDLE_ATTRIBUTE,
AttributeValue.newAttributeValue(pb));
```

例 6-26 Java を使用した新規フォルダへの PropertyBundle の適用

1. ValueDefault 経由で PropertyBundle をフェッチします。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd = (ValueDefault) vdColl.getItems("TranslatedTitleValueDefault");
AttributeValue av = vd.getPropertyValue();
PropertyBundle pb = (PropertyBundle) av.getPublicObject(session);

FolderDefinition fd = new FolderDefinition(session);
fd.setAttributeByUpperCaseName(Folder.NAME_ATTRIBUTE,
     AttributeValue.newAttributeValue("Collateral"));
fd.setAttributeByUpperCaseName(Folder.PROPERTYBUNDLE_ATTRIBUTE,
     AttributeValue.newAttributeValue(pb));

Folder f = (Folder) session.createPublicObject(fd);
```

PropertyBundle に基づく情報の検索

Oracle 9*i*FS Java API を使用すると、PropertyBundle の条件に基づいて情報を検索できます。たとえば、ファイル拡張子 'ppt' に関連するすべての Property を検索するとします。Oracle 9*i*FS Java API で Selector を構成し、データに基づいて Property を問い合せるアプリケーションを構築できます。

また、値が Guide の English という Property 持つすべての PublicObject をフェッチするとします。Oracle 9*i*FS Java API を使用すると、PropertyBundle に基づいて PublicObject に問合せできます。問合せを構成するには、Search で Property および PublicObject コンテン

ツ・タイプを結合する PropertyQualification を含め、Property に関する追加の条件を含めます。

注意： Selector と Search の作成手順の詳細は、第 8 章「検索アプリケーションの構築」を参照してください。

例 6-27 Selector を使用した Property のフェッチ

1. Property を選択するには、管理モードを有効化する必要があります。

```
session.setAdministrationMode(true);
```

2. Selector を構成します。

```
Selector s = new Selector(session);
s.setSearchClassname("PROPERTY");
s.setSearchSelection("NAME = 'ppt'");
```

3. 問合せを実行して各 Property を取得します。

```
LibraryObject[] props = s.getItems();

Property p;
int count = (props == null) ? 0 : props.length;
for (i = 0; i < count; i++)
{
   p = (Property) props[i];
}
```

例 6-28 Search を使用した共通プロパティ・データに基づく全 PublicObject の検索

1. 問合せ条件を保持する SearchSpecification を構成します。次の例には内容条件が含まれていないため、AttributeSearchSpecification を構成します。

```
AttributeSearchSpecification asp = new AttributeSearchSpecification();
```

2. SearchClassSpecification を構成し、AttributeSearchSpecification で設定します。

```
String[] searchClasses = {"PUBLICOBJECT"};
SearchClassSpecification scp = new SearchClassSpecification(searchClasses);
scp.addResultClass("PUBLICOBJECT");

asp.setSearchClassSpecification(scp);
```

3. Property 条件を指定するように PropertyQualification を構成します。

```
PropertyQualification pq = new PropertyQualification();
```

```
pq.setClassname("PUBLICOBJECT");
pq.setPropertyName("English");
pq.setOperatorType(AttributeQualification.EQUAL);
pq.setValue(AttributeValue.newAttributeValue("Guide"));
pq.setCaseIgnored(true);

asp.setSearchQualification(pq);
```

4. Search を実行し、結果をループして条件を満たす LibraryObject を取り出します。戻される LibraryObject は、SearchClassSpecification に指定したコンテンツ・タイプのものです。

```
Search srch = new Search(session, asp);
srch.open();

SearchResultObject sro = srch.next();
LibraryObject lo;

while (sro != null)
{
   lo = sro.getLibraryObject();
   try
   {
      sro = srch.next();
   }
   catch ( IfsException e)
   {
     if (e.getErrorCode() == 22000)
     {
       break;
     }
     else
     {
       e.printStackTrace();
     }
   }
}
srch.close();
```

PolicyPropertyBundle

PolicyPropertyBundle には、アプリケーションによる特定の操作の実行方法について、ポリシーまたはビジネス・ルールを定義する手段が用意されています。ビジネス・ルールのロジックは、カスタム Java クラスで実装されます。ポリシーは、特定の操作がコールされた時点でビジネス・ルールを実行する必要があることを示すように定義されます。ポリシー・セットをまとめて、1 つ以上のコンテンツ・タイプまたはインスタンスに対応付け、各オブジェクトに対する操作の実行方法を示すすべてのポリシーを保持できます。その後、アプリケーションで操作の実行を試みるときには、PolicyPropertyBundle を参照し、その操作のポリシーが定義されているかどうかを判断できます。ポリシーが定義されている場合、アプリケーションでは、ビジネス・ルールを実装する Java クラスを取得し、そのクラスを使用して操作を実行できます。

たとえば、Oracle 9*i*FS では、PolicyPropertyBundle を使用してレンダリング・フレームワークを実装しています。レンダリング・フレームワークには、開発者が Oracle 9*i*FS により自動的にコールされるレンダラを簡単にプラグインし、様々なプロトコルからフェッチされた時点でコンテンツ・タイプのインスタンスを表示できるような手段が用意されています。たとえば、開発者は、FTP 経由でフェッチされた場合は XML 形式で、SMB 経由でフェッチされた場合は VCard 形式で VCard インスタンスを表示するように、Oracle 9*i*FS を構成できます。このレンダラを構成するには、開発者は VCard コンテンツ・タイプの PolicyPropertyBundle にポリシーを作成します。Policy により、開発者が使用する必要のある XML レンダラの完全修飾名（'MyApplication.Renderers.MyXMLRenderer' など）に、操作 'FtpContentRenderer' が対応付けられます。それ以降は、このコンテンツ・タイプのインスタンスがフェッチされるときに、各プロトコル・サーバーは特定の操作名をコールするようにプログラムされます。Oracle 9*i*FS サーバーは、インスタンスのコンテンツ・タイプに対応付けられている PolicyPropertyBundle を参照して、指定の操作にレンダラが対応付けられているかどうかを判断します。対応付けられている場合は、Oracle 9*i*FS はその完全修飾名を取り出し、インスタンスを処理対象のレンダラに渡して、結果をプロトコル・サーバーに渡します。

PolicyPropertyBundle の使用

PolicyPropertyBundle を使用すると、システム単位のタスク、コンテンツ・タイプの全インスタンスに対する操作または特定インスタンスに対する操作の実行に使用するポリシーを格納できます。

- **システム単位の PolicyPropertyBundle:** PolicyPropertyBundle は、Oracle 9*i*FS に独立して存在できます。アプリケーションでは、PolicyPropertyBundle を参照し、特定のコンテンツ・タイプやインスタンスに限定されていない操作を実行できます。
- **コンテンツ・タイプの PolicyPropertyBundle:** PolicyPropertyBundle をコンテンツ・タイプの ClassObject に対応付けて、そのコンテンツ・タイプの全インスタンスに対する標準的なポリシーを定義できます。ClassObject は PolicyBundle 属性を持ち、この属性はコンテンツ・タイプに対する操作の実行に使用される PolicyPropertyBundle を参照します。

- **インスタンスの PolicyPropertyBundle:** PolicyPropertyBundle を使用すると、コンテンツ・タイプの特定インスタンスに対する操作を実行するためのポリシーを格納できます。各 PublicObject は属性 PolicyBundle を持ち、この属性はそのポリシーを保持する PolicyPropertyBundle を参照します。

PolicyPropertyBundle は Oracle 9*i*FS に独立して存在でき、通常はアプリケーションの処理に使用されるため、PolicyPropertyBundle を ValueDefault に対応付けると役立ちます。PolicyPropertyBundle を ValueDefault に対応付けると、文字列、つまり ValueDefault の UniqueName で一意に参照できます。たとえば、XML 構成ファイルでは、ValueDefault の UniqueName を、PolicyPropertyBundle を参照する要素の値として指定できます（<Update RefType = "ValueDefault">MyPolicyPropertyBundleValueDefault</Update> など）。また、Oracle 9*i*FS Java API には、ValueDefault 経由でオブジェクトを取得できるように特殊なメソッドも用意されています（`LibrarySession.getValueDefaultCollection()` など）。

注意： 現在、Oracle 9*i*FS Manager には PropertyBundle を定義するためのインタフェースは用意されていません。ただし、PropertyBundle 情報を検索して表示することはできます。Oracle 9*i*FS Manager の検索インタフェースの使用方法は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

PolicyPropertyBundle の構造

Oracle 9*i*FS では、操作の各ポリシーは Policy コンテンツ・タイプのインスタンスとして定義されます。Policy は、そのポリシーが適用される操作、ポリシーの動作を実装する Java Bean の完全修飾名およびポリシーを識別する列挙キーを格納するための属性を持ちます。

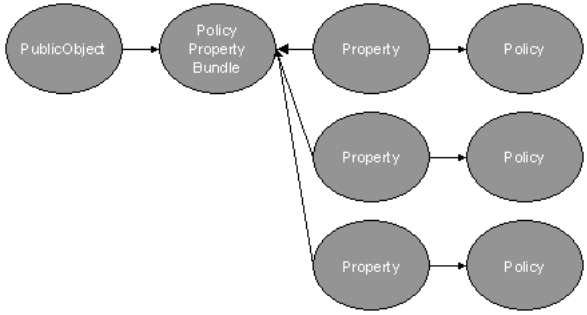
表 6-2 ポリシーの属性

属性	使用方法
Name	この Policy の名前。Oracle 9*i*FS の各永続オブジェクトには名前が付いています。
Operation	この Policy が適用される Oracle 9*i*FS サーバー操作の名前。
ImplementationName	Java クラスの完全修飾名。
ImplementationNum	Policy を識別する列挙キー。（この属性は、レンダラのプラグイン時には使用されません。）

Policy は、PolicyPropertyBundle インスタンスによりまとめられます。PolicyPropertyBundle コンテンツ・タイプは、PropertyBundle コンテンツ・タイプを拡張し

ます。したがって、PropertyBundle と同じ構造を持っています。ただし、この場合、PropertyBundle の Property の値は Policy を参照します。

図 6-2 PolicyPropertyBundle オブジェクト・モデル



注意： PropertyBundle の構造の詳細は、「PropertyBundle」を参照してください。

Oracle 9*i*FS のトップレベルのコンテンツ・タイプ LibraryObject は、PolicyBundle 属性を持っています。この属性は、Oracle 9*i*FS のすべてのコンテンツ・タイプに継承されます。ClassObject インスタンスは、継承した属性を使用して、ClassObject が表すコンテンツ・タイプの全インスタンスの処理に使用する PolicyPropertyBundle を対応付けることができます。PublicObject インスタンスは、継承した属性を使用して、特定のインスタンスの操作に対するポリシーの定義に使用する PolicyPropertyBundle を対応付けることができます。

PolicyPropertyBundle の定義

カスタム・アプリケーションによる操作の実行方法に関するポリシーを実装するには、次を定義する必要があります。

1. **操作名：** アプリケーションで実行対象となる特定タスクの参照に使用する操作を定義します。
2. **ポリシーの実装クラス：** 操作に対するポリシーの動作を実装する Java クラスを作成します。
3. **PolicyPropertyBundle:** カスタム操作を、ポリシーを実装する Java クラスに対応付ける PolicyPropertyBundle を作成します。

4. **カスタム・アプリケーション・ロジック：** PolicyPropertyBundle を使用してタスクの実行時に非定型動作を適用する、特別なロジックをカスタム・アプリケーションに組み込むことができます。アプリケーション・クライアントでは、実行する操作を指定できます。アプリケーション・サーバーでは、操作を受け取るときに、Java クラスを参照し、そのクラスを使用して操作を実行します。

この項では、Java クラスをカスタム操作に対応付ける PolicyPropertyBundle を定義する手順について説明します。ポリシーの動作やカスタム・アプリケーション・ロジックを実装する Java クラスは、アプリケーションに固有のため、その構築方法については説明しません。Oracle 9*i*FS では、この種の Java クラスの作成に特別な要件は適用されません。

Oracle 9*i*FS SDK では、PolicyPropertyBundle を複数の方法で定義できます。

- **XML:** Oracle 9*i*FS へのインポート時に PolicyPropertyBundle を自動的に定義する XML ファイルを作成します。
- **Java:** Oracle 9*i*FS Java API 経由で PolicyPropertyBundle をプログラムで定義します。

注意： 現在、Oracle 9*i*FS Manager には PolicyPropertyBundle を定義するためのインタフェースは用意されていません。ただし、PolicyPropertyBundle 情報を検索して表示することはできます。

Oracle 9*i*FS Manager の検索インタフェースの使用方法は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

XML を使用した PolicyPropertyBundle の作成および変更

XML ファイルを使用して、Oracle 9*i*FS で PolicyPropertyBundle を定義できます。XML ファイルを Oracle 9*i*FS にインポートすると、IfsSimpleXmlParser では自動的にそのファイルが使用され、PolicyPropertyBundle インスタンスが作成または変更されます。Oracle 9*i*FS では、XML を使用した PolicyPropertyBundle の削除はサポートされていません。

PolicyPropertyBundle を文字列で一意に参照できるように、PolicyPropertyBundle ごとに ValueDefault を作成することをお薦めします。次の例に、XML の <OBJECTLIST> タグを使用して XML ファイルに PolicyPropertyBundle と ValueDefault を作成する方法と、ValueDefault を使用して XML ファイルで PolicyPropertyBundle を更新する方法を示します。

例 6-29 XML を使用した PolicyPropertyBundle の作成

```
<?xml version='1.0' standalone='yes'?>
<OBJECTLIST>
 <POLICYPROPERTYBUNDLE>
   <NAME>Delivery Policies PolicyPropertyBundle</NAME>
   <POLICIES>
```

```
      <PROPERTY>
        <NAME>Fax</NAME>
        <VALUE Datatype="SystemObject" ClassName="Policy">
            <NAME>FaxPolicy</NAME>
            <OPERATION>SendByFax</OPERATION>
            <IMPLEMENTATIONNAME>
               MyCompany.MyApp.DeliveryServers.FaxServer
            </IMPLEMENTATIONNAME>
            <IMPLEMENTATIONENUM>1</IMPLEMENTATIONENUM>
        </VALUE>
      </PROPERTY>
      <PROPERTY>
        <NAME>Print</NAME>
        <VALUE Datatype="SystemObject" ClassName="Policy">
            <NAME>PrintPolicy</NAME>
            <OPERATION>SendByPrinter</OPERATION>
            <IMPLEMENTATIONNAME>
               MyCompany.MyApp.DeliveryServers.PrintServer
            </IMPLEMENTATIONNAME>
            <IMPLEMENTATIONENUM>2</IMPLEMENTATIONENUM>
        </VALUE>
      </PROPERTY>
    </POLICIES>
  </POLICYPROPERTYBUNDLE>

  <VALUEDEFAULT>
    <NAME>DeliveryPoliciesValueDefault</NAME>
    <VDVALUE Datatype="PublicObject" ClassName = "PolicyPropertyBundle"
      RefType = "Name">
         Delivery Policies PolicyPropertyBundle
    </VDVALUE>
  </VALUEDEFAULT>
</OBJECTLIST>
```

例 6-30 XML を使用した PolicyPropertyBundle への Policy の追加

```
<?xml version='1.0' standalone='yes'?>
 <POLICYPROPERTYBUNDLE>
  <UPDATE RefType='ValueDefault'>DeliveryPoliciesValueDefault</UPDATE>
   <POLICIES>
    <PROPERTY Action = 'add'>
     <NAME>Mail</NAME>
      <VALUE Datatype = 'SystemObject' ClassName="Policy">
            <NAME>MailPolicy</NAME>
            <OPERATION>SendByMail</OPERATION>
            <IMPLEMENTATIONNAME>
               MyCompany.MyApp.DeliveryServers.MailServer
```

```
          </IMPLEMENTATIONNAME>
          <IMPLEMENTATIONENUM>2</IMPLEMENTATIONENUM>
      </VALUE>
   </PROPERTY>
  </POLICIES>
 </POLICYPROPERTYBUNDLE>
```

例 6-31 XML を使用した PolicyPropertyBundle からの Policy の削除

```
<?xml version='1.0' standalone='yes'?>
 <POLICYPROPERTYBUNDLE>
   <UPDATE RefType='ValueDefault'>DeliveryPoliciesValueDefault</UPDATE>
   <POLICIES>
    <PROPERTY Action = 'remove'>
      <NAME>Fax</NAME>
    </PROPERTY>
  </POLICIES>
 </POLICYPROPERTYBUNDLE>
```

例 6-32 XML を使用した Policy の変更

```
<?xml version='1.0' standalone='yes'?>
 <POLICY>
    <UPDATE RefType = "Name">MailPolicy</UPDATE>
    <IMPLEMENTATIONENUM>1</IMPLEMENTATIONENUM>
 </POLICY>
```

XML 構成ファイルの各要素を使用して、PolicyPropertyBundle および Policy インスタンスの属性の値を指定します。

表 6-3 に、PolicyPropertyBundle 構成ファイルの各 XML 要素を示します。この表は、Java クラスとその要素に対応する Oracle 9*i*FS Java API の属性、各要素の使用方法および要素の有効な XML 属性を示しています。

表 6-3 PolicyPropertyBundle の XML 構成ファイルの要素

要素	クラス情報	属性	使用方法
<POLICYPROPERTYBUNDLE>	PolicyPropertyBundle は PublicObject のサブクラス ApplicationObject を拡張		PolicyPropertyBundle 構成ファイルのルート要素。PolicyPropertyBundle クラスの名前に対応する必要があります。
<POLICIES>	PolicyPropertyBundle クラスの属性との直接の対応関係はなし		PolicyPropertyBundle の Property の宣言は、<POLICIES> 要素で囲む必要があります。
<PROPERTY>	Property が SystemObject を拡張	Action	各ポリシーは <PROPERTY> 要素で参照されます。
<NAME>	LibraryObject スーパークラスから継承される、Property クラスの Name 属性		<NAME> 要素は、まとまりの各プロパティの名前を指定するために使用されます。
<VALUE>	Property クラスの Value 属性	データ型 RefType	<VALUE> 要素は、Property が参照する Policy を保持するために使用されます。 <VALUE> 要素の Datatype 属性は、Property クラスの DataType 属性に対応します。PolicyPropertyBundle の場合、Value は SystemObject を拡張する Policy を参照するため、Datatype は常に SystemObject です。 <VALUE> 要素の RefType 属性を使用すると、既存の Policy を参照できます。
<POLICY>	Policy が SystemObject を拡張		<POLICY> 要素は Policy の宣言を保持します。
<NAME>	LibraryObject スーパークラスから継承される、Policy クラスの Name 属性		この Policy の名前。

表 6-3 PolicyPropertyBundle の XML 構成ファイルの要素（続く）

要素	クラス情報	属性	使用方法
<OPERATION>	Policy クラスの Operation 属性		<OPERATION> 要素は、ポリシーが定義される操作を指定するために使用されます。
<IMPLEMENTATIONNAME>	Policy クラスの ImplementationName 属性		ポリシーの動作を実装する Java クラスの完全修飾名。
<IMPLEMENTATIONNUM>	Policy クラスの ImplementationNum 属性		Policy を列挙する整数。

Java を使用した PolicyPropertyBundle の作成、変更および削除

Oracle 9*i*FS Java API を使用して PolicyPropertyBundle を定義することもできます。この Java API には、各ポリシーに定義されている操作とポリシー実装クラスを操作するためのメソッドが用意されています。

例 6-33 Java を使用した PolicyPropertyBundle の作成

1. Policy を作成するには、管理モードを有効化する必要があります。

```
session.setAdministrationMode(true);
```

2. PolicyPropertyBundle 内で各 Policy を構成します。

```
PolicyDefinition poldef = new PolicyDefinition(session);
poldef.setAttributeByUpperCaseName(Policy.NAME_ATTRIBUTE,
     AttributeValue.newAttributeValue("FaxPolicy"));
poldef.setAttributeByUpperCaseName(Policy.OPERATION_ATTRIBUTE,
     AttributeValue.newAttributeValue("SendByFax"));
poldef.setAttributeByUpperCaseName(Policy.IMPLEMENTATIONNAME_ATTRIBUTE,
     AttributeValue.newAttributeValue(
        "MyCompany.MyApp.DeliveryServers.FaxServer"));
Policy pol1 = (Policy) session.createSystemObject(poldef);

poldef.setAttributeByUpperCaseName(Policy.NAME_ATTRIBUTE,
     AttributeValue.newAttributeValue("PrintPolicy"));
poldef.setAttributeByUpperCaseName(Policy.OPERATION_ATTRIBUTE,
     AttributeValue.newAttributeValue("SendByPrinter"));
poldef.setAttributeByUpperCaseName(Policy.IMPLEMENTATIONNAME_ATTRIBUTE,
     AttributeValue.newAttributeValue(
        "MyCompany.MyApp.DeliveryServers.PrintServer"));
Policy pol2 = (Policy) session.createSystemObject(poldef);
```

3. 値を一時的に保持するように PolicyPropertyBundleDefinition を構成します。

```
PolicyPropertyBundleDefinition ppbdef =
```

```
    new PolicyPropertyBundleDefinition(session);
ppbdef.setAttributeByUpperCaseName(PolicyPropertyBundle.NAME_ATTRIBUTE,
    AttributeValue.newAttributeValue("Delivery Policies PolicyPropertyBundle"));
ppbdef.addPropertyValue("Fax", AttributeValue.newAttributeValue(pol1));
ppbdef.addPropertyValue("Print", AttributeValue.newAttributeValue(pol2));
```

4. PolicyPropertyBundleDefinition を LibrarySession に渡して、新規の PolicyPropertyBundle および Policy インスタンスを作成します。

```
PolicyPropertyBundle ppb =
    (PolicyPropertyBundle) session.createPublicObject(ppbdef);
```

5. ValueDefault に PolicyPropertyBundle を保持するように ValueDefaultPropertyBundle を作成します。

```
ValueDefaultPropertyBundleDefinition vdpbd =
    new ValueDefaultPropertyBundleDefinition(session);
vdpbd.setAttributeByUpperCaseName(ValueDefaultPropertyBundle.NAME_ATTRIBUTE,
    AttributeValue.newAttributeValue(
        "DeliveryPoliciesValueDefaultPropertyBundle"));
vdpbd.setValue(ppb);

ValueDefaultPropertyBundle vdpb =
    (ValueDefaultPropertyBundle) session.createPublicObject(vdpbd);
```

6. PolicyPropertyBundle を文字列で参照できるように ValueDefault を作成します。

```
ValueDefaultDefinition vdd = new ValueDefaultDefinition(session);
vdd.setAttributeByUpperCaseName(ValueDefault.NAME_ATTRIBUTE,
    AttributeValue.newAttributeValue(
       "DeliveryPoliciesValueDefault"));

ValueDefault vd = (ValueDefault) session.createSchemaObject(vdd);
vd.setValueDefaultPropertyBundle(vdpb);
```

例 6-34 Java を使用した PolicyPropertyBundle の変更

1. 変更する PolicyPropertyBundle を取得します。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd = (ValueDefault)vdColl.getItems("DeliveryPoliciesValueDefault");
AttributeValue av = vd.getPropertyValue();
PolicyPropertyBundle ppb = (PolicyPropertyBundle) av.getPublicObject(session);
```

2. PolicyPropertyBundle に新規 Policy を追加します。

```
PolicyDefinition poldef = new PolicyDefinition(session);
poldef.setAttributeByUpperCaseName(Policy.NAME_ATTRIBUTE,
```

```
     AttributeValue.newAttributeValue("MailPolicy"));
poldef.setAttributeByUpperCaseName(Policy.OPERATION_ATTRIBUTE,
     AttributeValue.newAttributeValue("SendByMail"));
poldef.setAttributeByUpperCaseName(Policy.IMPLEMENTATIONNAME_ATTRIBUTE,
     AttributeValue.newAttributeValue(
        "MyCompany.MyApp.DeliveryServers.MailServer"));
Policy pol = session.createSystemObject(poldef);

ppb.putPropertyValue("Spanish", AttributeValue.newAttributeValue(pol));
```

3. PropertyBundle から新規 Property を削除します。

```
ppb.removePropertyValue("Fax");
```

4. PropertyBundle の Policy を変更します。

```
AttributeValue av = ppb.getPropertyValue("Mail");
Policy pol = (Policy) av.getSystemObject(session);
pol.setImplementationEnum(1);
```

例 6-35 Java を使用した PolicyPropertyBundle の削除

1. ValueDefault から削除する PolicyPropertyBundle を取得します。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd =
      (ValueDefault) vdColl.getItems("DeliveryPoliciesValueDefault");
ValueDefaultPropertyBundle vdpb = vd.getValueDefaultPropertyBundle();
AttributeValue av = vd.getPropertyValue();
PolicyPropertyBundle ppb = (PolicyPropertyBundle) av.getPublicObject(session);
```

2. PolicyPropertyBundle の Policy を取得します。

```
Property[] props = ppb.getProperties();
int count = (props == null) ? 0 : props.length;
```

3. 最初に Policy を削除します。Policy を削除するには、管理モードを有効化する必要があります。

```
session.setAdministrationMode(true);
int i;
Policy p;
for (i = 0; i < count; i++)
{
  p = (Policy) props[i].getValue().getSystemObject(session);
  p.free();
}
```

4. PolicyPropertyBundle を削除します。

```
ppb.free();
```

5. ValueDefault と ValueDefaultPropertyBundle を削除します。

```
vd.free();
vdpb.free();
```

コンテンツ・タイプとインスタンスへの PolicyPropertyBundle の適用

PolicyPropertyBundle は Oracle 9*i*FS に独立して存在し、アプリケーションで使用される他の PublicObject に対応付ける必要はありません。ただし、PolicyPropertyBundle を 1 つ以上のコンテンツ・タイプまたはその特定インスタンスに対応付けて、そのオブジェクトに対して限定的に実行される操作に関するカスタム・ポリシーを定義できます。

Oracle 9*i*FS では、各コンテンツ・タイプおよびインスタンスは PolicyBundle 属性を持ち、この属性で PolicyPropertyBundle を参照できます。PolicyPropertyBundle をコンテンツ・タイプに対応付けるには、そのコンテンツ・タイプを表す ClassObject インスタンスの PolicyBundle 属性の値を設定します。PolicyPropertyBundle をドキュメントに対応付けるには、PolicyPropertyBundle を参照する Document インスタンスの PolicyBundle 属性の値を設定します。PolicyPropertyBundle とコンテンツ・タイプまたはインスタンスとの対応付けを解除するには、PolicyBundle 属性を NULL に設定します。

Oracle 9*i*FS では、XML 構成ファイルまたは Java API を使用して、PolicyPropertyBundle をコンテンツ・タイプおよびインスタンスに適用できます。

XML を使用した PolicyPropertyBundle の適用

XML 構成ファイルを使用すると、次のようにいつでも PolicyPropertyBundle をコンテンツ・タイプおよびインスタンスに適用できます。

- 新規 PolicyPropertyBundle の作成時
- 新規コンテンツ・タイプまたはインスタンスの作成時
- PolicyPropertyBundle、コンテンツ・タイプまたはインスタンスが存在する場合

例 6-36 XML を使用した新規コンテンツ・タイプへの新規 PolicyPropertyBundle の適用

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <NAME>Image</NAME>
    <SUPERCLASS RefType = "Name">Document</SUPERCLASS>
    <POLICYBUNDLE>
     <NAME>Delivery Policies PolicyPropertyBundle</NAME>
     <POLICIES>
      <PROPERTY>
       <NAME>Fax</NAME>
```

```
        <VALUE Datatype="SystemObject" ClassName="Policy">
            <NAME>FaxPolicy</NAME>
            <OPERATION>SendByFax</OPERATION>
            <IMPLEMENTATIONNAME>
               MyCompany.MyApp.DeliveryServers.FaxServer
            </IMPLEMENTATIONNAME>
            <IMPLEMENTATIONENUM>1</IMPLEMENTATIONENUM>
        </VALUE>
       </PROPERTY>
       <PROPERTY>
        <NAME>Print</NAME>
        <VALUE Datatype="SystemObject" ClassName="Policy">
            <NAME>PrintPolicy</NAME>
            <OPERATION>SendByPrinter</OPERATION>
            <IMPLEMENTATIONNAME>
               MyCompany.MyApp.DeliveryServers.PrintServer
            </IMPLEMENTATIONNAME>
            <IMPLEMENTATIONENUM>2</IMPLEMENTATIONENUM>
        </VALUE>
       </PROPERTY>
      </POLICIES>
    </POLICYBUNDLE>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>Artists</NAME>
            <DATATYPE>DirectoryObject_Array</DATATYPE>
        </ATTRIBUTE>
      </ATTRIBUTES>
</CLASSOBJECT>
```

例 6-37 XML を使用した新規コンテンツ・タイプへの既存 PropertyBundle の適用

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <NAME>Image</NAME>
    <SUPERCLASS RefType = "Name">Document</SUPERCLASS>
    <POLICYBUNDLE RefType = "ValueDefault">
         DeliveryPoliciesValueDefault
    </POLICYBUNDLE>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>Artists</NAME>
            <DATATYPE>DirectoryObject_Array</DATATYPE>
        </ATTRIBUTE>
      </ATTRIBUTES>
</CLASSOBJECT>
```

例 6-38 XML を使用した既存コンテンツ・タイプへの既存 PropertyBundle の適用

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <UPDATE RefType = "Name">IMAGE</UPDATE>
    <POLICYBUNDLE RefType = "ValueDefault">
      DeliveryPoliciesValueDefault
    </POLICYBUNDLE>
</CLASSOBJECT>
```

Java を使用したコンテンツ・タイプおよびインスタンスへの PolicyPropertyBundle の適用

Java API を使用すると、コンテンツ・タイプとインスタンスに対する PolicyPropertyBundle の適用方法を厳密に制御できます。たとえば、インスタンスが特定の条件を満たしている場合に、PolicyPropertyBundle をプログラムで適用できます。また、一方を作成できない場合はどちらも作成されないように、PolicyPropertyBundle および PublicObject を単一トランザクションで作成することもできます。

注意： Oracle 9*i*FS でトランザクションを管理する手順は、第 16 章「セッションおよびトランザクションの管理」を参照してください。

例 6-39 Java を使用した新規コンテンツ・タイプへの新規 PolicyPropertyBundle の適用

1. PolicyPropertyBundle 内で各 Policy を構成します。

```
PolicyDefinition poldef = new PolicyDefinition(session);
poldef.setAttributeByUpperCaseName(Policy.NAME_ATTRIBUTE,
       AttributeValue.newAttributeValue("FaxPolicy"));
poldef.setAttributeByUpperCaseName(Policy.OPERATION_ATTRIBUTE,
       AttributeValue.newAttributeValue("SendByFax"));
poldef.setAttributeByUpperCaseName(Policy.IMPLEMENTATIONNAME_ATTRIBUTE,
       AttributeValue.newAttributeValue(
         "MyCompany.MyApp.DeliveryServers.FaxServer"));
Policy pol1 = (Policy) session.createSystemObject(poldef);

poldef.setAttributeByUpperCaseName(Policy.NAME_ATTRIBUTE,
       AttributeValue.newAttributeValue("PrintPolicy"));
poldef.setAttributeByUpperCaseName(Policy.OPERATION_ATTRIBUTE,
       AttributeValue.newAttributeValue("SendByPrinter"));
poldef.setAttributeByUpperCaseName(Policy.IMPLEMENTATIONNAME_ATTRIBUTE,
       AttributeValue.newAttributeValue(
         "MyCompany.MyApp.DeliveryServers.PrintServer"));
Policy pol2 = (Policy) session.createSystemObject(poldef);
```

2. PolicyPropertyBundle を作成します。

```
PolicyPropertyBundleDefinition ppbdef =
       new PolicyPropertyBundleDefinition(session);
ppbdef.setAttributeByUpperCaseName(PolicyPropertyBundle.NAME_ATTRIBUTE,
       AttributeValue.newAttributeValue(
         "Delivery Policies PolicyPropertyBundle"));
ppbdef.addPropertyValue("Fax", AttributeValue.newAttributeValue(pol1));
ppbdef.addPropertyValue("Print", AttributeValue.newAttributeValue(pol2));
```

3. PolicyPropertyBundleDefinition を LibrarySession に渡して、新規の PolicyPropertyBundle および Policy インスタンスを作成します。

```
PolicyPropertyBundle ppb =
      (PolicyPropertyBundle) session.createPublicObject(ppbdef);
```

4. ClassObjectDefinition を作成します。

```
ClassObjectDefinition codef = new ClassObjectDefinition(session);
codef.setAttributeByUpperCaseName(ClassObject.NAME_ATTRIBUTE,
      AttributeValue.newAttributeValue("IMAGE"));
ClassObject superClass = session.getClassObjectByName("DOCUMENT");
codef.setSuperclass(superClass);
```

5. ClassObjectDefinition の PropertyBundle 属性を、新規 PropertyBundle を参照するように設定します。

```
codef.setAttributeByUpperCaseName(ClassObject.POLICYBUNDLE_ATTRIBUTE,
      AttributeValue.newAttributeValue(ppb));
ClassObject classObj = (ClassObject) session.createSchemaObject(codef);
```

例 6-40 Java を使用した既存コンテンツ・タイプへの PolicyPropertyBundle の適用

1. ValueDefault 経由で PolicyPropertyBundle をフェッチします。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd =
       (ValueDefault) vdColl.getItems("DeliveryPoliciesValueDefault");
AttributeValue av = vd.getPropertyValue();
PolicyPropertyBundle ppb = (PolicyPropertyBundle) av.getPublicObject(session);
```

2. ClassObject をフェッチします。

```
ClassObject co = session.getClassObjectByName("IMAGE");
```

3. PropertyBundle 属性を、PropertyBundle を参照するように設定します。管理モードを有効化している必要があります。

```
session.setAdministrationMode(true);
co.setAttribute(ClassObject.POLICYBUNDLE_ATTRIBUTE,
```

```
    AttributeValue.newAttributeValue(ppb));
```

例 6-41 Java を使用した新規フォルダへの PolicyPropertyBundle の適用

1. ValueDefault 経由で PropertyBundle をフェッチします。

```
Collection vdColl = session.getValueDefaultCollection();
ValueDefault vd =
     (ValueDefault) vdColl.getItems("DeliveryPoliciesValueDefault");
AttributeValue av = vd.getPropertyValue();
PolicyPropertyBundle ppb = (PolicyPropertyBundle) av.getPublicObject(session);

FolderDefinition fd = new FolderDefinition(session);
fd.setAttributeByUpperCaseName(Folder.NAME_ATTRIBUTE,
     AttributeValue.newAttributeValue("Collateral"));
fd.setAttributeByUpperCaseName(Folder.POLICYBUNDLE_ATTRIBUTE,
     AttributeValue.newAttributeValue(ppb));

Folder f = (Folder) session.createPublicObject(fd);
```

Relationship

Oracle 9*i*FS では、ドキュメント、フォルダおよび他のタイプの情報を管理できるのみでなく、それぞれの間の対応付けも管理できます。Oracle 9*i*FS には、オブジェクトを関連付けるための 3 つの方法が用意されています。

- **オブジェクト・タイプ属性：** PublicObject の属性の値で、他の PublicObject またはその配列を参照できます。属性のデータ型は、PublicObject、SystemObject、SchemaObject、PublicObject_Array、SystemObject_Array または SchemaObject_Array のうち、どれでもかまいません。この方法は、1 対 1 または 1 対多の対応付けに適しています。PublicObject は ID で索引付けされるため、このモデルを使用すると左から右へと効率的に横断できます。また、属性に索引を付けて、右から左への対応付けを効率的に横断することもできます。

注意： Attribute の操作を定義する方法の詳細は、第 5 章「コンテンツ・タイプと属性の拡張」を参照してください。

- **PropertyBundle:** 同様に、PublicObject の PropertyBundle 内の Property の値で、別の PublicObject またはその配列を参照することもできます。PropertyBundle は、比較的静的なオブジェクト・グループを定義して、複数の PublicObject に対応付ける場合に適しています。PropertyBundle は、横断と操作が他の方法ほど簡単ではありません。

注意： PropertyBundle の操作方法の詳細は、「PropertyBundle」を参照してください。

- **Relationship:** Oracle 9*i*FS には Relationship コンテンツ・タイプが含まれており、多対多の対応付けをモデル化するときに使用されます。Relationship は、2 つの PublicObject 間の単方向（つまり左から右へ）の対応付けを定義します。1 つの PublicObject を、関係の RightObject または LeftObject として複数の PublicObject に対応付けることができます。Relationship により、対応付けの一方向への横断が簡単で効率的になります。Relationship コンテンツ・タイプを拡張し、関係のコンテキスト内で PublicObject に関連する属性と動作を含めることができます。

Relationship を使用すると、複合ドキュメントのように複合タイプの情報をモデル化できます。複合ドキュメントは複数のコンポーネントで構成され、各コンポーネントは複数の複合ドキュメントで再使用できます。各種の複合ドキュメントで、OLE、Hurls および独自の定義ファイル（つまり `FrameMaker .bk` ファイル）など、コンポーネント間の関係の管理に、それぞれ異なる方法を使用できます。各種複合ドキュメントの管理に必要な属性とメソッドを持つ、カスタム・タイプの関係を定義できます。

新しい Relationship タイプの定義

新しい Relationship タイプを定義するには、カスタム・コンテンツ・タイプを定義する場合と同じ手順で操作する必要があります。他のコンテンツ・タイプの場合と同様に、Relationship タイプを表す SchemaObject インスタンスのセットを作成します。また、Java クラスを実装して、Relationship タイプのカスタム動作を定義することもできます。ただし、新規 Relationship タイプを定義するためにカスタム Java クラスを実装する必要はありません。その後、SchemaObject を編集または削除して、Relationship タイプを変更したり削除できます。このような開発タスクは、Java API、XML ファイルまたは Oracle 9*i*FS Manager を使用して実行できます。

注意： カスタム・コンテンツ・タイプを作成、変更および削除する手順の詳細は、第 5 章「コンテンツ・タイプと属性の拡張」および第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照してください。すでにこの 2 つの章を読んでいる場合は、この項で Java と XML を使用して Relationship タイプを作成、変更および削除する方法をすばやく確認できます。

Relationship タイプの作成

新しい Relationship タイプを作成するには、Relationship を拡張する新規コンテンツ・タイプを作成し、そのタイプの対応付けの管理に必要な属性とメソッドを追加します。たとえば、Relationship を拡張する新しい関係 BookRelationship を作成し、拡張属性 ComponentLabel を含めることができます。

新規 Relationship タイプを定義する手順は、次のとおりです。

- 管理権限を持っているかどうかを確認します。XML 構成ファイルを使用する場合は、そのファイルを管理者としてインポートします。Java API を使用する場合は、Oracle 9*i*FS でのセッション用に管理モードを有効化します。
- Relationship タイプを表す ClassObject のインスタンスを作成します。
- Relationship タイプに関連する各属性を表す Attribute のインスタンスを作成します。
- Relationship タイプ用のカスタム Java クラスを作成し、カスタム・メソッドを定義します。

注意： Relationship タイプの Java クラスを拡張する方法は、第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照してください。

例 6-42 XML を使用した Relationship タイプの作成

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <NAME>BookRelationship</NAME>
    <SUPERCLASS RefType = "Name">Relationship</SUPERCLASS>
    <DESCRIPTION>Relationships between a Book and its components.</DESCRIPTION>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>ComponentLabel</NAME>
            <DATATYPE>String</DATATYPE>
        </ATTRIBUTE>
    </ATTRIBUTES>
</CLASSOBJECT>
```

注意： Relationship タイプの作成に使用する XML ファイルの各要素の詳細は、第 5 章「コンテンツ・タイプと属性の拡張」を参照してください。

例 6-43 Java を使用した Relationship タイプの作成

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. Relationship タイプを表す ClassObject の値を保持する ClassObjectDefinition を作成します。

```
ClassObjectDefinition codef = new ClassObjectDefinition(session);
codef.setAttributeByUpperCaseName(ClassObject.NAME_ATTRIBUTE,
     AttributeValue.newAttributeValue("BookRelationship"));
```

3. 新規 Relationship タイプで Relationship コンテンツ・タイプを拡張するように指定します。

```
ClassObject catco = session.getClassObjectByName("RELATIONSHIP");
codef.setSuperclass(catco);
```

4. Relationship タイプに関連する属性ごとに AttributeDefinition を作成します。AttributeDefinition を ClassObjectDefinition に追加します。

```
AttributeDefinition attdef1 = new AttributeDefinition(session);
attdef1.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("ComponentLabel"));
attdef1.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_STRING));
codef.addAttributeDefinition(attdef1);
```

```
AttributeDefinition attdef2 = new AttributeDefinition(session);
attdef2.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("FullPath"));
attdef2.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_STRING));
codef.addAttributeDefinition(attdef2);
```

5. 新規 Relationship タイプの ClassObject および Attribute を作成します。

```
ClassObject co = (ClassObject) session.createSchemaObject(codef);
```

Relationship タイプの変更

作成後の Relationship タイプの属性を追加したり削除できます。

注意： Relationship タイプの属性の追加や削除には、XML を使用できません。この操作には、Oracle 9*i*FS Manager または Java API を使用する必要があります。

Oracle 9*i*FS Manager で属性を削除する手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

例 6-44 Java を使用した Relationship タイプの属性の追加、削除および更新

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. Relationship タイプを表す ClassObject を取得します。

```
ClassObject relco = session.getClassObjectByName("BOOKRELATIONSHIP");
```

3. Attribute を ClassObject に追加します。

```
AttributeDefinition attdef1 = new AttributeDefinition(session);
attdef1.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("RelativePath"));
attdef1.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_STRING));
relco.addAttribute(attdef1);
```

4. Attribute を ClassObject から削除します。

```
Attribute att2 = catco.getEffectiveClassAttributes("FULLPATH");
relco.removeAttribute(att2);
```

5. ClassObject の Attribute を更新します。

```
Attribute att3 = catco.getEffectiveClassAttributes("COMPONENTLABEL");
Collection vdColl = session.getValueDomainCollection();
ValueDomain vd = vdColl.getItems("ComponentLabelValueDomain");
att3.setValueDomain(vd);
```

Relationship タイプの削除

Java API または Oracle 9*i*FS Manager を使用して、Relationship タイプを削除できます。Relationship タイプのインスタンスが存在する場合は、まずインスタンスを削除する必要があります。

注意： Oracle 9*i*FS Manager で Relationship タイプの ClassObject を削除する手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

例 6-45 Java を使用した Relationship タイプの削除

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. Relationship タイプを表す ClassObject を取得します。

```
ClassObject relco = session.getClassObjectByName("BOOKRELATIONSHIP");
```

3. `free()` メソッドをコールして ClassObject を削除します。

```
relco.free();
```

オブジェクトの関係付け

新規 Relationship タイプの作成後に、それを使用して 2 つの PublicObject を関係付けることができます。この項では、次のタスクについて説明します。

- PublicObject 間の関係の作成
- 関係の更新
- PublicObject の関係付けの解除

PublicObject 間の関係の作成

2 つの PublicObject を関係付けるには、Relationship タイプの新規インスタンスを作成します。

Java を使用した PublicObject の関係付け

Java API で 2 つの PublicObject を関係付ける手順は、次のとおりです。

1. 新規 Relationship インスタンスの値を保持する RelationshipDefinition を作成します。
2. 新しい関係のタイプを指定します。
3. 関係の Left_Object および Right_Object となる PublicObject を指定します。
4. 拡張関係属性の値を設定します。

例 6-46 Java を使用した PublicObject の関係付け

1. 関係付ける PublicObject をすでに取得しているとします。

```
PublicObject po1 = .....
PublicObject po2 = .....
```

2. 新規の RelationshipDefinition を作成します。

```
RelationshipDefinition rdef = new RelationshipDefinition(session);
```

3. 新規 Relationship インスタンスのタイプを指定します。

```
ClassObject co = session.getClassObjectByName("BOOKRELATIONSHIP");
rdef.setClassObject(co);
```

4. 関係の左辺と右辺の PublicObject を指定します。

```
rdef.setAttributeByUpperCaseName(Relationship.LEFTOBJECT_ATTRIBUTE,
    AttributeValue.newAttributeValue(po1));
rdef.setAttributeByUpperCaseName(Relationship.RIGHTOBJECT_ATTRIBUTE,
    AttributeValue.newAttributeValue(po2));
```

5. 拡張関係属性の値を設定します。

```
rdef.setAttributeByUpperCaseName("COMPONENTLABEL",
         AttributeValue.newAttributeValue("Chapter 1"));
```

6. RelationshipDefinition を使用して、PublicObject 間の新しい関係を作成します。

```
Relationship r = (Relationship) session.createSystemObject(rdef);
```

XML を使用した PublicObject の関係付け

管理者は、XML を使用して PublicObject を関係付けることもできます。XML ファイルを Oracle 9*i*FS にインポートすると、IfsSimpleXmlParser により自動的にファイルが解析され、Relationship タイプのインスタンスが生成されます。PublicObject の関係付けに使用する XML ファイルの主な側面は、次のとおりです。

- XML 構成ファイルのルート要素は、Relationship タイプの名前に対応する必要があります。
- それぞれの副要素は、その Relationship タイプに属する各 Attribute の名前に対応する必要があります。
- LeftObject および RightObject 属性の副要素では、関係付ける PublicObject を参照する必要があります。
- XML 構成ファイルは管理者がインポートする必要があります。エンド・ユーザーが XML を使用して関係を作成することはできません。

例 6-47 XML を使用した PublicObject の関係付け

```
<?xml version="1.0" standalone="yes"?>
<BOOKRELATIONSHIP>
    <LEFTOBJECT RefType = "Path">/home/guest/Manuals/UserGuide.bk</LEFTOBJECT>
    <RIGHTOBJECT RefType = "Path">
        /home/guest/Manuals/UserAdmin.fm
    </RIGHTOBJECT>
    <COMPONENTLABEL>Chapter 1 : User Administration</COMPONENTLABEL>
</BOOKRELATIONSHIP>
```

XML ファイルを使用して新規 PublicObject を作成する場合は、同じ XML ファイル内で PublicObject を関係付けることができます。そのためには、新規 PublicObject のデータを含む要素を、XML ファイルの先頭で指定する必要があります。これにより、関係データを含む要素では、前の要素で作成される PublicObject インスタンスを参照できます。Oracle 9*i*FS では、最初に PublicObject、次に関係インスタンスというように、各要素のインスタンスが順番に作成されます。このため、関係インスタンスを表す要素では、PublicObject をすでに存在するかのように参照できます。

例 6-48 XML を使用した新規 PublicObject の関係付け

```
<?xml version="1.0" standalone="yes"?>
<OBJECTLIST>
  <FEATURE>
    <NAME>Oracle 9iFS</NAME>
    <FOLDERPATH>/public/features</FOLDERPATH>
  </FEATURE>

  <PRODUCT>
    <NAME>Oracle9i Database Server</NAME>
    <FOLDERPATH>/public/products</FOLDERPATH>
  </PRODUCT>

  <PRODUCT>
    <NAME>Oracle9i Application Server</NAME>
    <FOLDERPATH>/public/products</FOLDERPATH>
```

```
  </PRODUCT>

  <PRODUCTFEATURERELATIONSHIP>
    <LEFTOBJECT RefType = "Path">/public/products/Oracle9i Application Server
    </LEFTOBJECT>
    <RIGHTOBJECT RefType = "Path">/public/features/Oracle 9iFS
    </RIGHTOBJECT>
    <PRICE>No additional charge.</PRICE>
  </PRODUCTFEATURERELATIONSHIP>

  <PRODUCTFEATURERELATIONSHIP>
    <LEFTOBJECT RefType = "Path">/public/products/Oracle9i Database Server
    </LEFTOBJECT>
    <RIGHTOBJECT RefType = "Path">/public/features/Oracle 9iFS
    </RIGHTOBJECT>
    <PRICE>No additional charge.</PRICE>
  </PRODUCTFEATURERELATIONSHIP>

</OBJECTLIST>
```

関係の更新

PublicObject を関係付けた後に、ユーザーがオブジェクト間の関係を変更する必要が生じる場合があります。

「PublicObject 間の関係の作成」および「PublicObject の関係付けの解除」のように、ユーザーが新規の関係を作成し、古い関係を削除して、PublicObject 間の関係を変更できるように、カスタム・アプリケーションを構築できます。たとえば、Book がある章のバージョン 1 に関係付けられていて、その章が改訂される場合、ユーザーはバージョン 1 から Book の関係付けを解除して、その章のバージョン 2 に関係付けることができます。

カスタム・アプリケーションでは、管理者がオブジェクト間の既存の関係を直接変更できるようにすることもできます。この項では、管理者がオブジェクト間の関係を直接更新する方法について説明します。

XML を使用した関係の更新： Relationship インスタンスを更新するには、次の要件を満たす XML 構成ファイルが必要です。

- ルート要素が Relationship タイプの Name に対応していること。
- 変更対象となる Relationship インスタンスを <UPDATE> 副要素で参照していること。
- 変更対象となるすべての属性の副要素を含んでいること。これらの要素の値には、更新後の属性値が保持されます。
- 管理者がインポートすること。

例 6-49 XML を使用した関係の更新

```
<?xml version="1.0" standalone="yes"?>
<BOOKRELATIONSHIP>
    <UPDATE RefType = "ComponentLabel">Chapter 1 : User Administration</UPDATE>
    <LEFTOBJECT RefType = "Path">/home/guest/Manuals/AdminGuide.bk</LEFTOBJECT>
</BOOKRELATIONSHIP>
```

Java を使用した関係の更新： Oracle 9*i*FS Java API を使用して関係を更新することもできます。それぞれの拡張 Relationship タイプは、メソッド `getLeftObject()`、`getRightObject()` および `getSortSequence()` を継承します。各メソッドを使用して、PublicObject 間の関係を操作できます。また、継承されるメソッド `getAttribute()` および `setAttribute()` には、拡張属性を操作する汎用的な方法が用意されています。

カスタム Relationship タイプの拡張属性の値を取得および設定するコンビニエンス・メソッドを提供するために、カスタム Java クラスを実装することもできます。

注意： カスタム Java クラスの実装手順は、第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照してください。

例 6-50 Java を使用した Relationship の属性の更新

1. 管理モードを有効化します。

   ```
   session.setAdministrationMode(true);
   ```

2. アプリケーションですでに Relationship インスタンスを取得しているとします。

   ```
   Relationship r = ......
   ```

3. BookRelationship インスタンスの SortSequence 属性を更新します。

   ```
   r.setAttribute("COMPONENTLABEL",
       AttributeValue.newAttributeValue("Chapter 2 : User Administration"));
   ```

PublicObject の関係付けの解除

PublicObject 間の関係を削除するには、両者をリンクしている Relationship インスタンスを削除します。

他のオブジェクトの場合と同様に、Oracle 9*i*FS Java API で PublicObject の `removeRelationship()` メソッドをコールして、PublicObject の関係を削除できます。このメソッドをコールすると、Oracle 9*i*FS により Relationship インスタンスが削除されます。Oracle 9*i*FS では、関係を削除する手段としての XML はサポートされず、Oracle 9*i*FS Manager には Relationship インスタンスを削除するためのユーザー・インタフェースは用意されていません。

例 6-51 Java を使用した Relationship インスタンスの削除

1. アプリケーションですでに Relationship および PublicObject を取得しているとします。

```
Relationship r = ......
PublicObject po = .....
```

2. Relationship インスタンスを削除します。

```
po.removeRelationship(r);
```

関係に基づく PublicObject の検索

Oracle 9*i*FS では、情報間の関係を横断できます。特定の PublicObject に関係付けられている情報をすべてフェッチできます。また、選択対象を限定し、PublicObject と特定タイプの関係を持つ情報をすべて検索することもできます。たとえば、特定のドキュメントについて、そのドキュメントを含むすべてのフォルダをフェッチするとします。このとき、ドキュメントが RightObject となる FolderRelationship で、LeftObject となっているすべての Folder を検索します。

また、関連する PublicObject に関するメタデータに基づいて、情報を検索することもできます。実際には、対応付ける Relationship オブジェクトに基づいて、PublicObject のメタデータの結合を構成する検索を構成します。たとえば、2001 年 6 月 1 日に作成されたすべてのフォルダにある全ドキュメントを検索できます。そのためには、CreateDate 属性が *June 1, 2001* に設定されている Folder を LeftObject に持つ FolderRelationship で、RightObject となっているすべての PublicObject を検索します。

関係の横断

Oracle 9*i*FS Java API を使用すると、PublicObject 間の関係を簡単に横断できます。PublicObject クラスには、`getLeftwardRelationshipObjects()` や `getRightwardRelationshipObjects()` などのメソッドが含まれており、オブジェクトに関係付けられたすべての情報をすばやくフェッチできます。

例 6-52 PublicObject の関係のフェッチ

1. アプリケーションですでに PublicObject を取得しているとします。

```
PublicObject po = .....
```

2. PublicObject を LeftObject または RightObject として参照する、すべての Relationship インスタンスをフェッチします。

```
Relationship[] rightwardRels =
       po.getRightwardRelationships("BOOKRELATIONSHIP");
Relationship[] leftwardRels =
       po.getLeftwardRelationships("BOOKRELATIONSHIP");
```

3. この PublicObject に LeftObject または RightObject として関係付けられている PublicObject を直接フェッチします。

```
PublicObject[] rightwardObjs =
       po.getRightwardRelationshipObjects("BOOKRELATIONSHIP");
PublicObject[] leftwardObjs =
       po.getLeftwardRelationshipObjects("BOOKRELATIONSHIP");
```

4. 関連する各 PublicObject ごとに、その PublicObject の名前を取得します。

```
AttributeValue av; // variable to hold value of a relationship attribute
int i;
PublicObject relPO;
String poName;

int icount = (rightwardObjs == null) ? 0 : rightwardObjs.length;
for(i = 0; i < icount; i++)
{
   relPO = rightwardObjs[i];
   poName = relPO.getName();
}
```

Relationship クラスには、メソッド `getLeftObject()` および `getRightObject()` も用意されており、特定の Relationship インスタンスから関連 PublicObject インスタンスを取得できます。PublicObject の取得後に、その PublicObject が持つ他のすべての属性を取得して、PublicObject を操作するメソッド（`getContent()`、`setContent()` など）をコールできます。

例 6-53 Relationship インスタンスからの PublicObject のフェッチ

1. アプリケーションですでに Relationship インスタンスを取得しているとします。

```
Relationship r = ......
```

2. 関係から Product をフェッチします。

```
PublicObject p = r.getLeftObject();
```

3. 関係から Feature をフェッチします。

```
PublicObject f =  r.getRightObject();
```

4. Product に関するメタデータをフェッチします。

```
ClassObject co = p.getClassObject();
String coName = co.getName();
Attribute[] atts = co.getEffectiveClassAttributes();

AttributeValue av;
```

```
int i, icount;
String attName;

icount = (atts == null) ? 0 : atts.length;
for (i = 0; i < icount; i++)
{
   attName = atts[i].getName();
   av = p.getAttribute(attName);
}
```

関係に基づく PublicObject の検索

Oracle 9*i*FS Java API では、Selector と Search を使用して、さらに複雑な問合せを構成することもできます。Selector を使用すると、特定タイプの関係に関連するすべての PublicObject をフェッチできます。Search を使用すると、関連する PublicObject に関する条件に基づいて PublicObject に問合せできます。例 6-54 および例 6-55 に、関係メタデータに基づく検索方法を示します。

注意： Selector と Search の作成手順の詳細は、第 8 章「検索アプリケーションの構築」を参照してください。

例 6-54 では、特定のドキュメントを含むすべての Book を選択しています。そのためには、そのドキュメントを RightObject として参照するすべての BookRelationship を選択するように、Selector を構成します。この例では、BookRelationship をループし、LeftObject として参照されている PublicObject（つまり Book）を取り出しています。

例 6-54 Selector を使用した、特定タイプの関係に関する全 PublicObject の検索

1. アプリケーションですでに Document インスタンスを取得しているとします。

```
Document doc = ...
```

2. このドキュメントを含むすべての Book を取得するように Selector を構成します。

```
Selector s = new Selector(session);
s.setSearchClassname("BOOKRELATIONSHIP");
s.setSearchSelection("RIGHTOBJECT = " + doc.getId());
```

3. 問合せを実行し、PublicObject を RightObject として参照する各 BookRelationship インスタンスを取得します。

```
LibraryObject[] rels = s.getItems();

int count = (rels == null) ? 0 : rels.length;
for (i = 0; i < count; i++)
```

```
Relationship br;
PublicObject po;
{
   br = (Relationship) rels[i];
   po = br.getLeftObject();
}
```

例 6-55 では、名前に 'Work-in-progress' を含む PublicObject があるすべての Book を問い合せています。この例では、'Work-in-progress' を含む名前を持つ PublicObject を RightObject として参照するすべての BookRelationship を選択するように、検索を構成しています。この例では、選択された BookRelationship をループし、LeftObject として参照されている PublicObject（つまり Book）を取り出しています。

例 6-55 Search を使用した、関連 PublicObject の属性に基づく全 PublicObject の検索

1. 問合せ条件を保持する SearchSpecification を構成します。次の例には内容条件が含まれていないため、AttributeSearchSpecification を構成します。

```
AttributeSearchSpecification asp = new AttributeSearchSpecification();
```

2. SearchClassSpecification を構成し、AttributeSearchSpecification で設定します。

```
String[] searchClasses = {"PUBLICOBJECT", "BOOKRELATIONSHIP"};
SearchClassSpecification scp = new SearchClassSpecification(searchClasses);
scp.addResultClass("BOOKRELATIONSHIP");

asp.setSearchClassSpecification(scp);
```

3. BookRelationship で RightObject として参照される PublicObject を結合するように、JoinQualification を構成します。

```
JoinQualification jq = new JoinQualification();
jq.setLeftAttribute("PUBLICOBJECT", null);
jq.setRightAttribute("BOOKRELATIONSHIP", "RIGHTOBJECT");
```

4. PublicObject 名に Work-in-progress を含むように指定する AttributeQualification を構成します。

```
AttributeQualification aq = new AttributeQualification();
aq.setAttribute("PUBLICOBJECT", "NAME");
aq.setOperatorType(AttributeQualification.LIKE);
aq.setValue("%Work-in-progress%");
```

5. SearchClause を構成し、Join および Attribute 修飾を結合し、両方の条件が必ず満たされるように要求します。これを使用して SearchQualification を設定します。

```
SearchClause sc = new SearchClause(aq, jq, SearchClause.AND);
asp.setSearchQualification(sc);
```

6. Search を実行し、結果をループして条件を満たす LibraryObject を取り出します。戻される LibraryObject は、SearchClassSpecification に指定したコンテンツ・タイプのものです。

```
Search srch = new Search(session, asp);
srch.open();

int i;
LibraryObject lo;
SearchResultObject sro = srch.next();
PublicObject po;
Relationship r;
String poName;

while (sro != null)
{
    r = (Relationship) sro.getLibraryObject();
    po = r.getLeftObject();
    poName = po.getName();

    try
    {
       sro = srch.next();
    }
    catch (IfsException e)
    {
      if (e.getErrorCode() == 22000)
      {
         break;
      }
      else
      {
          e.printStackTrace();
       }
     }
}

srch.close();
```

サンプル・コード

Oracle 9*i*FS には、Java API の操作の出発点として使用できるように、2 組のサンプル・コード・ファイルが用意されています。

- ドキュメント・サンプル・ファイル
- API サンプル・コード

ドキュメント・サンプル・ファイル

Oracle 9*i*FS は、この章の例について実行可能なサンプル・コード・ファイルとともにインストールされます。各サンプル・コード・ファイルは、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/devdoc/arbitrarymetadata` ディレクトリにあります。これらのサンプル・コード・ファイルは、この章の例をテストする場合に役立ちます。ただし、適切な順序で実行しないと、2 回以上実行した場合にネーミングの競合が生じることがあります。

表 6-4 に、サンプル・コード・ファイルとそれに対応する例を示します。

表 6-4 例とサンプル・コード・ファイル

例	サンプル・コード・ファイル
例 6-1「XML を使用したカテゴリ・タイプの作成」	`CreateCategoryType.xml`
例 6-2「Java を使用したカテゴリ・タイプの作成」	`CreateCategoryType.java`
例 6-3「Java を使用したカテゴリの属性の追加、削除および更新」	`UpdateCategoryType.java`
例 6-4「Java を使用したカテゴリの削除」	`DeleteCategoryType.java`
例 6-5「Java を使用した PublicObject のカテゴリ分類」	`CategorizePublicObject.java`
例 6-6「XML を使用した新規 PublicObject のカテゴリ分類」	`CategorizeNewPublicObject.xml`
例 6-7「XML を使用した Category インスタンスの属性の更新」	`UpdateCategories.xml`
例 6-8「Java を使用した Category インスタンスの更新」	`UpdateCategories.java`
例 6-9「Java を使用した Category インスタンスの削除」	`DeleteCategories.java`
例 6-10「Selector を使用した PublicObject の特定カテゴリの検索」	`SelectPOCategories.java`
例 6-11「Search を使用した共通カテゴリ属性に基づく全 PublicObject の検索」	`SearchPOByCategory.java`
例 6-12「PublicObject のカテゴリのフェッチ」	`FetchPOCategories.java`
例 6-13「Category インスタンスからの対応付けられている PublicObject のフェッチ」	`FetchPOFromCategory.java`

表 6-4 例とサンプル・コード・ファイル（続く）

例	サンプル・コード・ファイル
例 6-14「XML を使用した PropertyBundle の作成」	CreatePB.xml
例 6-15「XML を使用した PropertyBundle の変更」	UpdatePB.xml
例 6-16「Java を使用した PropertyBundle の作成」	CreatePB.java
例 6-17「Java を使用した PropertyBundle の変更」	UpdatePB.java
例 6-18「Java を使用した PropertyBundle の削除」	DeletePB.java DeleteOrphanedPBs.java DeleteOrphanedPBValueDefaults.java DeleteOrphanedPBValueDefaultPBs.java
例 6-19「XML を使用した新規コンテンツ・タイプへの新規 PropertyBundle の適用」	ApplyNewPBtoNewCT.xml
例 6-20「XML を使用した新規コンテンツ・タイプへの既存 PropertyBundle の適用」	ApplyExistingPBtoNewCT.xml
例 6-21「XML を使用した既存コンテンツ・タイプへの既存 PropertyBundle の適用」	ApplyExistingPBtoExistingCT.xml
例 6-22「XML を使用した新規フォルダへの既存 PropertyBundle の適用」	ApplyExistingPBtoNewPO.xml
例 6-23「XML を使用した既存フォルダへの既存 PropertyBundle の適用」	ApplyExistingPBtoExistingPO.xml
例 6-24「Java を使用した新規コンテンツ・タイプへの新規 PropertyBundle の適用」	ApplyNewPBtoNewCT.java
例 6-25「Java を使用した既存コンテンツ・タイプへの PropertyBundle の適用」	ApplyExistingPBtoExistingCT.java
例 6-26「Java を使用した新規フォルダへの PropertyBundle の適用」	ApplyExistingPBtoNewPO.java
例 6-27「Selector を使用した Property のフェッチ」	SelectProperties.java
例 6-28「Search を使用した共通プロパティ・データに基づく全 PublicObject の検索」	SearchPOByPB.java

表 6-4 例とサンプル・コード・ファイル（続く）

例	サンプル・コード・ファイル
例 6-29「XML を使用した PolicyPropertyBundle の作成」	CreatePPB.xml
例 6-30「XML を使用した PolicyPropertyBundle への Policy の追加」	AddPolicytoPPB.xml
例 6-31「XML を使用した PolicyPropertyBundle からの Policy の削除」	RemovePolicyfromPPB.xml
例 6-32「XML を使用した Policy の変更」	UpdatePolicy.xml
例 6-33「Java を使用した PolicyPropertyBundle の作成」	CreatePPB.java
例 6-34「Java を使用した PolicyPropertyBundle の変更」	UpdatePPB.java
例 6-35「Java を使用した PolicyPropertyBundle の削除」	DeletePPB.java DeleteOrphanedPPBValueDefaults.java DeleteOrphanedPPBValueDefaultPBs.java DeleteOrphanedPPBs.java DeleteOrphanedPolicies.java
例 6-36「XML を使用した新規コンテンツ・タイプへの新規 PolicyPropertyBundle の適用」	ApplyNewPPBtoNewCT.xml
例 6-37「XML を使用した新規コンテンツ・タイプへの既存 PropertyBundle の適用」	ApplyExistingPPBtoNewCT.xml
例 6-38「XML を使用した既存コンテンツ・タイプへの既存 PropertyBundle の適用」	ApplyExistingPPBtoExistingCT.xml
例 6-39「Java を使用した新規コンテンツ・タイプへの新規 PolicyPropertyBundle の適用」	ApplyNewPPBtoNewCT.java
例 6-40「Java を使用した既存コンテンツ・タイプへの PolicyPropertyBundle の適用」	ApplyExistingPPBtoExistingCT.java
例 6-41「Java を使用した新規フォルダへの PolicyPropertyBundle の適用」	ApplyExistingPPBtoNewPO.java
例 6-42「XML を使用した Relationship タイプの作成」	CreateRelationshipType.xml

表 6-4 例とサンプル・コード・ファイル（続く）

例	サンプル・コード・ファイル
例 6-43「Java を使用した Relationship タイプの作成」	`CreateRelationshipType.java`
例 6-44「Java を使用した Relationship タイプの属性の追加、削除および更新」	`UpdateRelationshipType.java`
例 6-45「Java を使用した Relationship タイプの削除」	`DeleteRelationshipType.java`
例 6-46「Java を使用した PublicObject の関係付け」	`RelatePublicObjects.java`
例 6-47「XML を使用した PublicObject の関係付け」	`RelateExistingPublicObjects.xml`
例 6-48「XML を使用した新規 PublicObject の関係付け」	`RelateNewPublicObjects.xml`
例 6-49「XML を使用した関係の更新」	`UpdateRelationship.xml`
例 6-50「Java を使用した Relationship の属性の更新」	`UpdateRelationships.java`
例 6-51「Java を使用した Relationship インスタンスの削除」	`DeleteRelationships.java`
例 6-52「PublicObject の関係のフェッチ」	`FetchPORelationships.java`
例 6-53「Relationship インスタンスからの PublicObject のフェッチ」	`FetchPOFromRelationship.java`
例 6-54「Selector を使用した、特定タイプの関係に関する全 PublicObject の検索」	`SelectPOByRelationshipType.java`
例 6-55「Search を使用した、関連 PublicObject の属性に基づく全 PublicObject の検索」	`SearchPOByRelatedPO.java`

Java サンプル・コード・ファイルの実行

Java サンプル・コード・ファイルを実行する手順は、次のとおりです。

1. Oracle 9*i*FS ホスト・マシン上で、CLASSPATH に Java API の JAR ファイルが含まれているかどうかを確認します。
2. JDK 1.2.2 で、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/devdoc/arbitrarymetadata` ディレクトリにあるサンプル・コード・ファイルをコンパイルします。

例 6-56 Solaris でのサンプル・コード・ファイルのコンパイル

```
> cd $ORACLE_HOME/9ifs/samplecode/oracle/ifs/examples/devdoc/arbitrarymetadata
> javac CreatePPB.java
```

3. ファイルのコメントをチェックし、このファイルの前に他のファイルの実行が予期されているかどうかを調べます。
4. クラスを実行して引数を指定します。引数の入力値が表示されます。すべての引数に値が必要です。値を指定しない場合は、一部またはすべての引数にデフォルトが設定されます。クラスにより結果が画面に出力されます。

例 6-57 Solaris でのサンプル・コード・ファイルの実行

```
> java oracle.ifs.examples.devdoc.arbitrarymetadata.CreatePPB

  Running with arguments :
    user = system
    password = manager9ifs
    service = IfsDefault
    schemapassword = ifssys
  -----------Results----------
Policy created : FaxPolicy
 for operation : SendByFax
 with implementation : MyCompany.MyApp.DeliveryServers.FaxServer
Policy created : PrintPolicy
 for operation : SendByPrinter
 with implementation : MyCompany.MyApp.DeliveryServers.PrintServer
PolicyPropertyBundle created : Delivery Policies PolicyPropertyBundle
ValueDefaultPropertyBundle created : DeliveryPoliciesValueDefaultPropertyBundle
Value Default created : DeliveryPoliciesValueDefault
```

XML サンプル・コード・ファイルの実行

XML 構成ファイルを実行する手順は、次のとおりです。

1. XML 構成ファイルの解析をサポートする Oracle 9*i*FS クライアントを開きます。
2. XML 構成ファイルにより実行されるタスクに該当する管理権限を持つユーザーでログインします。
3. XML 構成ファイルをインポートします。
4. 結果を Oracle 9*i*FS Manager で表示します。

API サンプル・コード

Oracle 9*i*FS は、例のサンプル・コード・ファイルのみでなく、Java API の操作開始の出発点として役立つ高度なサンプル・コードとともにインストールされます。この API サンプル・コードは、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/api` ディレクトリにあります。この API サンプル・コードは何度実行しても名前の競合が生じることはないため、ドキュメント・サンプル・コード・ファイルより操作が簡単です。

表 6-5 に、この章に関連する API サンプル・コードを示します。

表 6-5 API サンプル・コード

クラス	使用方法
`CategorySample.java`	Category を使用して Document を作成します。
`PropertyBundleSample.java`	PropertyBundle を作成します。PropertyBundle 内で Property を追加、更新および削除します。

7

属性の妥当性チェック

この章の内容は、次のとおりです。

- 属性の妥当性チェックの概要 Oracle 9iFS
- ValueDefault の使用
- ClassDomain の使用
- ValueDomain の操作

属性の妥当性チェックの概要 Oracle 9*i*FS

Oracle 9*i*FS には、属性の妥当チェック用に次の 3 つのメカニズムが用意されています。

- **ValueDefault:** ClassObject の作成時に属性の初期値を設定します。
- **ClassDomain:** ClassObject を取る属性に可能な値を特定のクラスに限定するために使用されます。
- **ValueDomain:** 属性に可能な値を特定のリストに限定するために使用されます。

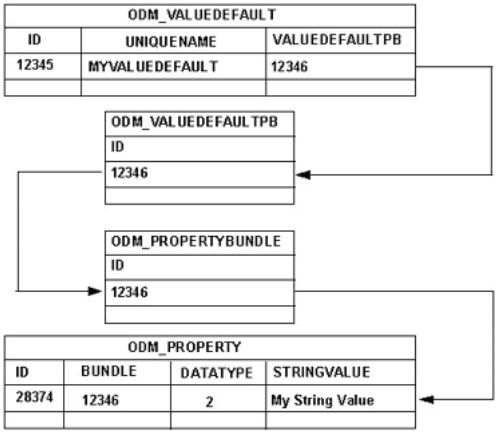
属性の妥当性チェックが設定されている場合は、エンド・ユーザーが属性を無効な値に設定すると、例外が発生します。開発者は、例外の結果として発生する動作を判断し、それをユーザー・インタフェース経由で適切に処理する必要があります。

ValueDefault の使用

ValueDefault を使用して、ClassObject の作成時に属性の初期値を特定の値に設定します。エンド・ユーザーは、ValueDefault を明示的にオーバーライドできます。ユーザーが値を指定していない場合、ValueDefault が使用されます。

ValueDefault は単一の値のみを持つ PropertyBundle として格納されます（ValueDefault が配列値の場合、デフォルト値として識別される配列は 1 つのみです）。PropertyBundle は、一意の整数値で識別されます。図 7-1 に、ValueDefault オブジェクトを表 ODM_PROPERTY に格納されている値に関係付けるための複合関係を示します。

図 7-1 Oracle 9*i*FS のスキーマに ValueDefault が格納される方法



ValueDomain 表は、VALUEDEFAULTPB 属性により ODM_ValueDefaultPB 表にリンクされています。ODM_VALUEDEFAULTPB は、同じ ID 値で ODM_PROPERTYBUNDLE 表にリ

ンクされています。最後に、ODM_PROPERTYBUNDLE 表は、同じ ID を使用して ODM_PROPERTY 表の BUNDLE 列にリンクされています。これにより、STRINGVALUE と ValueDefault オブジェクト間の対応付けが完成します。

XML 構成ファイルを使用して ValueDomain を作成すると、この複雑さのほとんどは自動的に処理されます。

XML 構成ファイルを使用した値デフォルトの作成

例 7-1「ValueDefault の XML 構成ファイル ApproverVDefault.xml」の XML 構成ファイルでは、カスタム・クラス ExpenseReport の ValueDefault である ApproverVDefault を作成しています。この XML 構成ファイルのルート要素では、作成対象のオブジェクトが ValueDefault として識別されます。VDValue タグは、Datatype パラメータを使用してデータ型 STRING を持つデフォルトを識別し、使用する文字列値を識別します。

例 7-1 ValueDefault の XML 構成ファイル ApproverVDefault.xml

```
<?xml version='1.0' standalone= 'yes'?>
<ValueDefault>
    <Name> ApproverVDefault </Name>
    <VDValue Datatype="String">Chris Stevens</VDValue>
</ValueDefault>
```

ValueDefault オブジェクトの作成後は、例 7-1 で作成した ValueDefault を使用する ClassObject を作成できます。例 7-2「ClassObject の XML 構成ファイル ExpenseReport.xml」に、ExpenseReport カスタム・クラスを作成する XML 構成ファイルを示します。ExpenseReport クラスは、新規 ExpenseReport インスタンスの作成時に、ApproverVDefault を使用して Approver 属性を移入します。

例 7-2 ClassObject の XML 構成ファイル ExpenseReport.xml

```
<?xml version = '1.0' standalone = 'yes'?>
<ClassObject>
 <Name>ExpenseReport</Name>
 <Superclass RefType= 'name'>Document</Superclass>
 <Attributes>
  <Attribute>
   <Name>Approver</Name>
   <DataType>String</DataType>
   <DataLength>20</DataLength>
   <ValueDefault RefType='name'>ApproverVDefault
   </ValueDefault>
  </Attribute>
 </Attributes>
</ClassObject>
```

ValueDefault の例のテスト

ValueDefault の例は、次の例を使用してテストできます。例 7-3「ExpenseReport101.xml」では、例 7-1 で定義したデフォルト値を使用する ExpenseReport クラスのインスタンスを作成します。例 7-4「ExpenseReport102.xml」では、Approver 属性に独自エントリ Joyce Doe を提供してデフォルト値をオーバーライドする ExpenseReport クラスのインスタンスを作成します。

例 7-3 ExpenseReport101.xml

```
<EXPENSEREPORT>
 <Name>ExpenseReport101</Name>
</EXPENSEREPORT>
```

例 7-4 ExpenseReport102.xml

```
<EXPENSEREPORT>
 <Name>ExpenseReport102</Name>
 <Approver>Joyce Doe</Approver>
</EXPENSEREPORT>
```

ExpenseReport サンプル・ファイルをテストする手順は、次のとおりです。

1. 使用したいプロトコルを通じてファイル `ApproverVDefault.xml` をアップロードします。このファイルでは、ValueDefault が定義されています。Web ユーザー・インタフェースを使用している場合は、このファイルと残りのファイルを Oracle 9*i*FS にアップロードするときに、「Parse File on Upload」ボックスがオンになっていることを確認してください。
2. ファイル `ExpenseReport.xml` を Oracle 9*i*FS にアップロードします。これにより、カスタム・サブクラス ExpenseReport が定義されます。
3. ファイル `ExpenseReport101.xml` をアップロードします。この XML ファイルには、Approver 属性の定義が含まれていないため、デフォルトが使用されます。
4. ファイル `ExpenseReport102.xml` をアップロードします。この XML ファイルには、Approver 属性の定義が含まれています。ValueDefault ではなく、このファイルに指定されている明示的な値が使用されます。

ClassDomain の使用

ClassDomain は、属性の値を特定の Oracle 9*i*FS クラスまたはクラス・セットに限定します。

クラス・ドメインの作成

引き続き ExpenseReport の例で、別のクラスがより大きいレポートの詳細行として ExpenseReport オブジェクトを使用するとします。このとき、タイプが PublicObject の属性をサマリー・レポートに追加できますが、ExpenseReport クラスの PublicObject に限定する必要があります。例 7-5「ExpenseReportClassDomain.xml の XML 構成ファイル」に、ExpenseReport クラスに限定された ClassDomain を作成する XML 構成ファイルを示します。

表 7-1「ClassDomain クラスの属性」に、この XML 構成ファイルで設定する必要のある属性を示します。

表 7-1 ClassDomain クラスの属性

名前	データ型	説明
Name	STRING	ClassDomain オブジェクト間で一意である必要があります。
Classes	SchemaObject[]	ClassDomain 内のクラスを含む配列です。
DomainType	Int	ClassDomain に指定のクラスのサブクラスを含める必要がある場合は、1 に設定します。 ClassDomain に指定のクラスのサブクラスを含めないようにする必要がある場合は、0（ゼロ）に設定します。 この 2 つの値は、ClassDomain クラスの静的最終変数 CLASSDOMAINTYPE_ENUMERATED_AND_SUBCLASSES (1) および CLASSDOMAINTYPE_ENUMERATED (0) として表されます。

ExpenseReportClassDomain を作成するために設定する必要のある値は、次のとおりです。

- Name: ExpenseReportClassDomain。
- Classes: ExpenseReport。
- DomainType: CLASSDOMAINTYPE_ENUMERATED_AND_SUBCLASSES。ただし、XML ファイルでは静的変数を使用できないため、値は整数 1 として入力されます。

例 7-5「ExpenseReportClassDomain.xml の XML 構成ファイル」に、結果的な XML ファイルを示します。

例 7-5 ExpenseReportClassDomain.xml の XML 構成ファイル

```
<?xml version = '1.0' standalone = 'yes'?>
<ClassDomain>
 <Name>ExpenseReportClassDomain</Name>
 <DomainType>1</DomainType>
 <Classes>
  <ArrayElement RefType='name'>ExpenseReport</ArrayElement>
 </Classes>
</ClassDomain>
```

ExpenseReportClassDomain の作成後に、ExpenseDetailLine クラスを作成し、ExpenseReportClassDomain オブジェクトを使用して、その単一属性 RelatedExpenseReport への入力の妥当性をチェックできます。

例 7-6 XML 構成ファイル ExpenseDetailLine.xml

```
<?xml version = '1.0' standalone = 'yes'?>
<ClassObject>
 <Name>ExpenseDetailLine</Name>
 <Superclass Reftype='name'>Document</Superclass>
 <Attributes>
  <Attribute>
   <Name>RelatedExpenseReport</Name>
   <DataType>PublicObject</DataType>
   <ClassDomain RefType='name'>ExpenseReportClassDomain</ClassDomain>
  </Attribute>
 </Attributes>
</ClassObject>
```

ClassDomain の例のテスト

前述までの例のように、ClassDomain の作成を完了すると、サンプル・ファイルを使用して結果をテストできます。

例 7-7「GoodExpenseDetailLine.xml」では、例 7-3「ExpenseReport101.xml」で作成した ExpenseReport101 ExpenseReport インスタンスを属性 RelatedExpenseReport の値として使用し、ExpenseDetailLine クラスの有効なインスタンスを作成しています。

例 7-7 GoodExpenseDetailLine.xml

```
<EXPENSEDETAILLINE>
 <Name>GoodExpenseDetailLine</Name>
```

```
 <RelatedExpenseReport RefType='name'>ExpenseReport101
  </RelatedExpenseReport>
</EXPENSEDETAILLINE>
```

例 7-8「ExpenseDetailLine101.xml」では、ExpenseDetailLine101 という名前で ExpenseDetailLine クラスのインスタンスを作成しています。これは、ExpenseReport クラスに属していないオブジェクトを示すためのダミー・オブジェクトです。

例 7-8 ExpenseDetailLine101.xml

```
<EXPENSEDETAILLINE>
 <Name>ExpenseDetailLine101</Name>
</EXPENSEDETAILLINE>
```

例 7-9「BadExpenseDetailLine.xml」では、ExpenseDetailLine のインスタンスの作成を試みていますが、RelatedExpenseReport 属性では、ClassDomain に必要な ExpenseReport クラスのインスタンスではなく、例 7-8「ExpenseDetailLine101.xml」で作成した ExpenseDetailLine クラスのインスタンスを識別しているため、作成試行は失敗します。

例 7-9 BadExpenseDetailLine.xml

```
<EXPENSEDETAILLINE>
 <Name>BadExpenseDetailLine</Name>
 <RelatedExpenseReport RefType='name'>ExpenseDetailLine101
 </RelatedExpenseReport>
</EXPENSEDETAILLINE>
```

ClassDomain サンプル・ファイルをテストする手順は、次のとおりです。

1. 使用したいプロトコルを使用して ExpenseReportClassDomain.xml をアップロードします。これらの例のアップロードに Web ユーザー・インタフェースを使用している場合は、「Parse File on Upload」ボックスがオンになっているかどうかを確認してください。このファイルでは、ExpenseReportClassDomain が作成されます。
2. `ExpenseDetailLine.xml` をアップロードします。このファイルでは、ClassDomain が適用される属性 RelatedExpenseReport を持つカスタム・クラス ExpenseDetailLine が作成されます。
3. `GoodExpenseDetail.xml` をアップロードします。このファイルは、RelatedExpenseReport 属性の値が有効な ExpenseReport ClassObject を使用して設定されているため成功します。

   ClassDomain が予期したとおり動作するかどうかを確認するには、Web ユーザー・インタフェースを使用して属性のリストを表示します。

   - Web インタフェースで「GoodExpenseDetailLine」の左にあるチェックボックスにチェックをし、「Edit」アイコンをクリックして「Properties」を選択します。

- 「Properties」ウィンドウで、RELATEDEXPENSEREPORT 属性の値が ExpenseReport クラスの ExpenseReport101 になっていることに注意してください。これは、ExpenseReportClassDomain に準拠しています。

4. `ExpenseDetailLine101.xml` をアップロードします。このファイルでは、ExpenseDetailLine クラスのインスタンスが作成されます。
5. `BadExpenseDetailLine.xml` をアップロードします。このファイルの RelatedExpenseReport の値は ExpenseDetailLine101 オブジェクトに設定されており、ExpenseReport クラスの有効なインスタンスではありません。Oracle 9*i*FS では、このインスタンスは作成されません。

ValueDomain の操作

ValueDomain を属性に適用し、その属性の値が許容値セットの範囲内になるように保証できます。この Expense Report の使用例のコンテキストで、例 7-10 に ValueDomain を使用して Approver 属性の値を制限する方法を示します。

ValueDomain は、例 7-10 のように、XML で ValueDomain の値に `VDomainValue` タグを使用して作成できます。このタグには、次の必須パラメータがあります。

- DATATYPE: Oracle 9*i*FS のデータ型です。
- DOMAINTYPE: `Enumerated`、`exclusive_maximum`、`exclusive_minimum`、`exclusive_range`、`maximum`、`minimum` および `range` です。

手順の概要

ValueDomain の動作を示すために、この項では XML を使用して複数のオブジェクトを作成し、XML ファイルで行った割当ての確認に使用する Java クラスを示します。

特に、この項では表 7-2 に示すファイルを使用します。

1. `ValueDomain1.xml` をアップロードします。このファイルでは、`TestVDomain.java` クラスを使用してテストできる ValueDomain が作成されます。
2. `ApproverVDomain.java` をコンパイルし、生成されたクラスを `<ORACLE_HOME>/9ifs/custom_classes/oracle/ifs/examples/devdoc/validate/` ディレクトリに置きます。
3. コマンドラインから次のコマンドを入力し、このクラスを実行します。
   - `java oracle.ifs.examples.devdoc.validate.ApproverVDomain <userName> <passWord> <serviceName> <schemaPassWord>`
4. `TestVDomain.java` をコンパイルし、生成されたクラスを `<ORACLE_HOME>/9ifs/custom_classes/oracle/ifs/examples/devdoc/validate` ディレクトリに置きます。

5. コマンドラインから次のコマンドを入力し、このクラスをテストします。
   - `java oracle.ifs.examples.devdoc.validate.TestVDomain <userName> <passWord> <serviceName> <schemaPassWord>`
6. 定義した ValueDomain を持つ属性を作成または更新すると、Oracle 9*i*FS では、ValueDomainPropertyBundle に定義されている値と比較して妥当性がチェックされます。属性値が有効でなければ、Oracle 9*i*FS で例外が発生します。

表 7-2 ValueDomain のサンプル・ファイル

`ValueDomain1.xml`	ValueDomain を作成し、DomainType および DataType を設定し、配列要素を宣言し、UniqueName を設定します。
`ValueDomainIntArray.xml`	整数配列のデモとなる ValueDomain を作成します。
`UPDValueDomain.xml`	既存の ValueDomain の ValueDomain および DataType を更新します。
`ApproverVDomain.java`	ExpenseReport の Approver 属性について、値を Bob Alva、John Smith または Chris Stevens に制限して ValueDomain を作成します。
`TestVDomain.java`	一方には有効な Approver 値、他方には無効な Approver 値を指定して、ExpenseReport の 2 つのインスタンスを作成します。2 番目のオブジェクトで例外が発生します。

XML を使用した ValueDomain の作成

ValueDomain は、例 7-10 のような XML ファイルをリポジトリに挿入することで作成できます。

例 7-10 ValueDomain1.xml

```
<?xml version = '1.0' standalone = 'yes'?>
<VALUEDOMAIN>
  <Name>ApproverValueDomain</Name>
  <Active>true</Active>
  <Description> my value_domain to set the Approver attribute </Description>
  <VDomainValue Domaintype="enumerated" Datatype="string_array">
      <ArrayElement>Bob Alva</ArrayElement>
      <ArrayElement>John Smith</ArrayElement>
      <ArrayElement>Chris Stevens</ArrayElement>
   </VDomainValue>
  <UniqueName>ApproverValueDomain</UniqueName>
</VALUEDOMAIN>
```

例 7-10 では、DomainType は "enumerated" に設定されています。これは、明示的に指定された値のみが有効であることを意味します。Datatype は "string_array" に設定されており、これは ValueDomain が String オブジェクトの配列で構成されていることを意味します。

DomainType と DataType は同じオブジェクトに適用されるため、両者は対応している必要があります。たとえば、整数の範囲が等しくなるように ValueDomain を割り当てるには、例 7-11 のように、DataType を `integer_array` に、DomainType を `range` にする必要があります。

例 7-11 ValueDomainIntArray.xml

```
<?xml version = '1.0' standalone = 'yes'?>
<VALUEDOMAIN>
  <Name>codes_range</Name>
  <Active>true</Active>
  <Description> my value_domain to set value ranges  </Description>
  <VDomainValue datatype="integer_array" domaintype="range">
         <MinValue>10</MinValue>
         <MaxValue>20</MaxValue>
  </VDomainValue>
  <UniqueName>codes_range</UniqueName>
</VALUEDOMAIN>
```

XML を使用した ValueDomain の更新

例 7-12 のように、ValueDomain は XML を使用して更新できます。この場合、`codes_max` という既存の ValueDomain の DomainType は `maximum` に設定され、DataType には `integer` が割り当てられています。

例 7-12 UPDValueDomain.xml

```
<?xml version="1.0" standalone="yes"?>
<ValueDomain>
   <update reftype="name"> codes_max</update>
   <VDomainValue domaintype="maximum" datatype="integer">
          100
   </VDomainValue>
</ValueDomain>
```

Java を使用した ValueDomain の作成

例 7-13 の ApproverVDomain.java クラスは、Oracle 9*i*FS Java API を使用してこれらの妥当性チェック・オブジェクトを使用する方法を示しています。このクラスでは、ExpenseReport の Approver 属性について、値を Bob Alva、John Smith または Chris

Stevens に制限して ValueDomain が作成されます。このプログラムでは、例 7-10 の XML 構成ファイルで作成されたのと同じ ValueDomain が作成されます。

例 7-13 ApproverVDomain.java

```
package oracle.ifs.examples.devdoc.validate;

import oracle.ifs.beans.Attribute;
import oracle.ifs.beans.ClassObject;
import oracle.ifs.beans.LibraryService;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.PublicObject;
import oracle.ifs.beans.SchemaObject;
import oracle.ifs.beans.ValueDomain;
import oracle.ifs.beans.ValueDomainDefinition;
import oracle.ifs.beans.ValueDomainPropertyBundle;
import oracle.ifs.beans.ValueDomainPropertyBundleDefinition;
import oracle.ifs.common.AttributeValue;
import oracle.ifs.common.CleartextCredential;
import oracle.ifs.common.Collection;
import oracle.ifs.common.IfsException;

/**
 * ApproverVDomain class
 * This class creates a valuedomain for the APPROVER attribute of
 * class EXPENSEREPORT. The valuedomain limits the value of the APPROVER
 * attribute to Bob Alva, John Smith, or Chris Stevens.
 **/

public class ApproverVDomain
{

  /**
   * Creates the valuedoamin and applies it to the APPROVER attribute
   * of class EXPENSEREPORT.
   *
   * @param args the command-line arguments for connecting to iFS:
   *  userName, passWord, serviceName, schemaPassWord
   */
  public static void main(String[] args)
  {
    ApproverVDomain vDomain = new ApproverVDomain();
    vDomain.CreateVDomain(args[0], args[1], args[2], args[3]);
  }


  /**
```

```
   * Creates the valuedomain and applies it to the APPROVER attribute
   * of class EXPENSEREPORT.
   *
   * @param usedfor connecting to iFS:
   *        userName, passWord, serviceName, schemaPassWord
   */
  public void CreateVDomain(String userName, String passWord, String serviceName,
  String schemaPassWord)
  {
    try
    {
      //Connects to the IFS.
      LibraryService service=LibraryService.startService(serviceName, schemaPassWord);
      CleartextCredential cred=new CleartextCredential(userName, passWord);
      LibrarySession session=service.connect(cred, null);
      session.setAdministrationMode(true);
      System.out.println("Successfully connected.");

      String s = "STRING_ENUMERATED";

      //Creates the ValueDomainPropertyBundle
      ValueDomainPropertyBundleDefinition vdpbd = new
ValueDomainPropertyBundleDefinition(session);
      vdpbd.setEnumeratedValues(new String[] { "Bob Alva", "John Smith", "Chris
Stevens" });
      ValueDomainPropertyBundle vdpb =
(ValueDomainPropertyBundle)session.createPublicObject(vdpbd);

      //Creates the ValueDomain
      ValueDomainDefinition vdd = new ValueDomainDefinition();
      vdd.setName("ApproverValueDomain");
      vdd.setAttribute("VALUEDOMAINPROPERTYBUNDLE",
       AttributeValue.newAttributeValue(vdpb));
      ValueDomain vd = (ValueDomain) session.createSchemaObject(vdd);

      //Gets class object collection, then gets the specific EXPENSEREPORT class
object.
      Collection classObjectCollection = session.getClassObjectCollection();
      ClassObject expenseReportClassObject = (ClassObject)
classObjectCollection.getItems("EXPENSEREPORT");

      //Gets the APPROVER attribute for the class object.
      Attribute attribute =
expenseReportClassObject.getEffectiveClassAttributes("APPROVER");

      //Modifies the attribute for the class object.
      //Applies the value doamin to this class object.
```

```
      attribute.setValueDomain(vd);
      attribute.setValueDomainValidated(true);
      System.out.println("ValueDomain applied.");
    }
    catch ( IfsException ifex )
    {
      System.err.println( "ERROR OCCURRED: " + ifex.toString() );
      ifex.printStackTrace( System.err );
    }
  }
}
```

Java を使用した ValueDomain のテスト

例 7-14 の TestVDomain クラスを使用すると、例 7-10 または例 7-13 で作成した ValueDomain が予期したとおりに機能しているかどうかを検査できます。

例 7-14 TestVDomain.java

```
package oracle.ifs.examples.devdoc.validate;

import oracle.ifs.beans.ClassObject;
import oracle.ifs.beans.Document;
import oracle.ifs.beans.DocumentDefinition;
import oracle.ifs.beans.LibraryService;
import oracle.ifs.beans.LibrarySession;

import oracle.ifs.common.AttributeValue;
import oracle.ifs.common.CleartextCredential;
import oracle.ifs.common.Collection;
import oracle.ifs.common.IfsException;

/**
 * TestVDomain class.
 * This class creates two instances of the EXPENSEREPORT class.
 * One instance has a value in the APPROVER attribute that complies
 * with the value domain, the other instance has one that violates
 * the value domain.
 *
 * An exception is generated when you try to create an instance
 * that violates the value domain.
 **/


public class TestVDomain
{
```

```
public static void main(String[] args)
{
  try
  {
    //Shows detailed error message while the exception is thrown
    IfsException.setVerboseMessage(true);

    //Connects to iFS

    LibraryService service=LibraryService.startService(args[2], args[3]);
    CleartextCredential cred=new CleartextCredential(args[0], args[1]);
    LibrarySession session=service.connect(cred, null);
    System.out.println("Successfully connected");

   //Gets class object collection, then gets the specific EXPENSEREPORT class
   //object.
   Collection classObjectCollection = session.getClassObjectCollection();
   ClassObject expenseReportClassObject = (ClassObject)
     classObjectCollection.getItems("EXPENSEREPORT");


   //Creates an EXPENSEREPORT instance that complies with the value domain

   System.out.println("Attempting to create an EXPENSEREPORT instance that" +
     " complies with the value domain");
   DocumentDefinition docdef = new DocumentDefinition(session);
   docdef.setName("GoodVDomainExpenseReport");
   docdef.setClassObject(expenseReportClassObject);
   docdef.setAttribute("APPROVER",
     AttributeValue.newAttributeValue("John   Smith"));
   Document doc = (Document)session.createPublicObject(docdef);
   AttributeValue av = doc.getAttribute("APPROVER");
   System.out.println("Created an expense report with Approver as " +
     av.getString(session));


  //Creates an EXPENSEREPORT instance that violates the value domain

  System.out.println("Attempting to create an EXPENSEREPORT instance that " +
    "violates the value domain");
  docdef = new DocumentDefinition(session);
  docdef.setName("BadVDomainExpenseReport");
  docdef.setClassObject(expenseReportClassObject);
  docdef.setAttribute("APPROVER", AttributeValue.newAttributeValue("foo"));
  doc = (Document)session.createPublicObject(docdef);
  av = doc.getAttribute("APPROVER");
  System.out.println("Created an expense report with Approver as " +
```

```
      av.getString(session));

  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
 }
}
```

8

検索アプリケーションの構築

この章では、Oracle 9*i*FS の Selector および Search 機能を使用して検索アプリケーションを構築する方法について説明します。

検索機能の概要

検索とは、Oracle 9*i*FS に格納されている情報を見つける方法です。検索では、情報に関する条件に基づいてリポジトリから情報が動的に取り出されます。たとえば、Name に文字 *XML* を含むすべてのドキュメントを検索するとします。検索では、その条件と実行時に一致していたドキュメントが戻されます。その後、条件と一致する新規ドキュメントを作成して再び検索を実行すると、そのドキュメントも自動的に取り出されます。

Oracle 9*i*FS では、オブジェクト指向の方法で情報を格納、検索して操作できます。ドキュメント、フォルダ、ユーザーおよびグループなど、すべての情報はオブジェクト、つまり Oracle 9*i*FS のクラス（Document、Folder、DirectoryUser および DirectoryGroup など）のインスタンスとして定義されます。Oracle 9*i*FS Java API では、情報がデータベースに格納される方法の複雑さを知らなくても、これらのオブジェクトを操作できます。たとえば、ドキュメントをフェッチして変更し、様々な表にまたがる複数の行としてではなく単一オブジェクトとしてリポジトリに保存できます。同様に、Oracle 9*i*FS ではオブジェクト指向の方法で情報を問い合せて、問合せの結果を操作できます。情報が格納されている表を結合する複合 SQL 問合せを構成するのではなく、クラス、属性、内容および他のオブジェクトとの関係に関する条件に基づいて、オブジェクトを問合せできます。

- **クラス条件：** 取り出す必要のあるオブジェクトのクラス（Documents、Folders、DirectoryObjects、カスタム・コンテンツ・タイプなど）を指定できます。また、問合せを拡張し、そのクラスのサブクラスを含めることもできます。
- **属性条件：** 名前、所有者または変更日など、オブジェクトの属性に基づいて問合せできます。属性条件を結合して論理的な構造体にすることができます（owner = 'wtalman' および modification date > January 1, 2001 など）。
- **内容条件：** ドキュメントの内容に基づいて問合せできます。Oracle 9*i*FS では Oracle Text 機能を使用してドキュメントの内容に索引付けするため、ドキュメントに含まれる語句（トークン）別にドキュメントを問合せできます。論理演算子を使用して内容条件のセットを結合し、トークン間の近接性（'XML' near 'Internet Applications' など）に基づいてドキュメントを問合せできます。また、問合せを拡張し、指定したトークンと発音やスペルが似ていたり、関連付けられているトークンを含めることもできます。
- **フォルダ制限：** 問合せを制限し、特定のフォルダまたは特定のフォルダ分岐にあるオブジェクトを取り出すことができます。フォルダは、リポジトリにある情報を編成するカタログを作成する場合や、ユーザーが情報を共有できるように作業領域を作成する場合に役立ちます。フォルダ制限による検索では、問合せ対象をカタログ内の特定のサブジェクトや特定の作業領域に限定できます。
- **関係条件：** 関連オブジェクトに関する条件に基づいてオブジェクトを問合せできます。たとえば、ドキュメントの問合せ時に、そのドキュメントを参照するフォルダの属性を条件として指定できます。
- **ソート条件：** 取り出されたオブジェクトのどの属性順にでも、問合せ結果をソートできます。また、内容条件を指定した場合は、問合せ結果のドキュメントを、そのドキュメントの内容と条件との関連性、つまり、内容検索の実行時に Oracle Text により生成されるスコアに基づいてソートできます。

Oracle 9*i*FS Java API では、リレーショナル・データベースに対する検索と、オブジェクト指向のプログラミング環境の使用しやすさという、2 つのメリットを活用できます。API では、Java を使用してオブジェクト指向の方法で問合せを作成し、結果を操作できます。問合せの実行時に、Oracle 9*i*FS は自動的に問合せの Java 構造体を SQL SELECT 文に変換し、Oracle 9*i*FS スキーマに対して問合せを実行します。したがって、ドキュメント、フォルダおよび他の情報を Java オブジェクトとして問い合せて操作でき、基礎となるデータベース表に属性と内容がどのように格納されるかを知っている必要はありません。

Oracle 9*i*FS Java API には、Oracle 9*i*FS のオブジェクトを問合せできるように、次の 2 つのプライマリ・インタフェースが用意されています。

- **Selector** は、リポジトリで単純問合せを実行するための使用しやすいインタフェースです。Selector インタフェースの説明は、「Selector の操作」を参照してください。
- **Search** は、複合問合せを作成して実行するための確実なインタフェースです。Search インタフェースの説明は、「Search の操作」を参照してください。

Selector の操作

Selector を使用すると、リポジトリ内で単純問合せを実行できます。静的条件を指定して単一クラスの情報を検索する必要がある場合は、Selector が最も便利で効率的です。

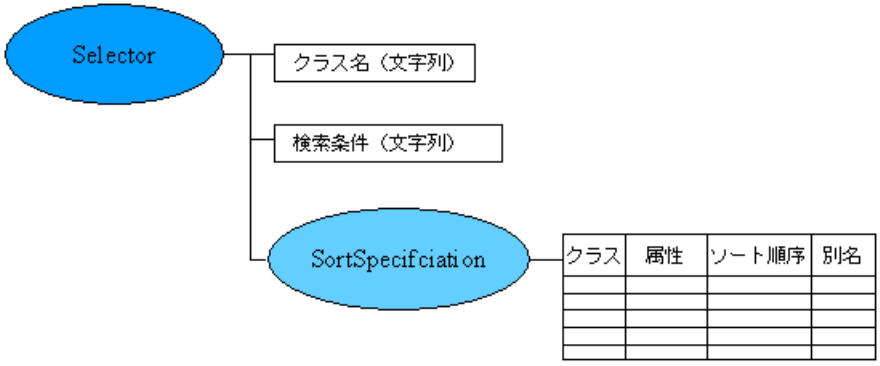
例 8-1 単純問合せ

```
select all Documents whose Owner is 'WTalman' and whose ModifiedDate is greater than
'January 01, 2001'
```

Selector オブジェクト・モデル

Selector は、3 つのコンポーネント、つまり、クラスを指定する文字列、検索条件を指定する文字列および結果のソート方法を指定する SortSpecification オブジェクトで構成されています。図 8-1 に、Selector の構造体を示します。

図 8-1 Selector オブジェクト・モデル



Selector は、次のルールに基づいて動作します。

- **単一クラス：** Selector では、単一クラスのオブジェクトを検索できます。複数クラスの情報を選択したり、結合を構成して関連クラスに関する条件を指定することはできません。クラス条件は、Oracle 9*i*FS で Selector から構成される SQL SELECT 文の FROM 句となります。
- **再帰的クラス：** Selector では、指定したクラスのすべてのサブクラスで再帰的に検索できます。
- **自由形式の SQL 式：** Selector では、単一文字列で検索条件を指定できます。この文字列は、Oracle 9*i*FS で Selector から構成される SQL SELECT 文の WHERE 句となります。
- **遅延バインド変数なし：** Selector では、遅延バインド変数を使用して検索条件を動的に構成することはできません。検索条件は、単一文字列で表現可能である必要があります。
- **ソート条件：** Selector では、検索結果を対象となったクラスの属性順にソートできます。ソート条件は、Oracle 9*i*FS で Selector から構成される SQL SELECT 文の ORDER BY 句となります。

Selector API

Oracle 9*i*FS Java API には、`oracle.ifs.beans` パッケージに Selector 構成用の 2 つのクラスが用意されています。

- Selector
- SortSpecification

Selector

Selector クラスは、検索の構成と実行に使用します。表 8-1 に、`Selector` クラスの主要メソッドを示します。

表 8-1 Selector の主要メソッド

メソッド	用途
getSearchClassname() setSearchClassname()	検索対象のクラスを決定します。
isRecursiveSearch()	Selector でサブクラスにまたがって再帰的に検索する必要があるかどうかを決定します。
getSearchSelection() setSearchSelection()	検索条件を決定します。
getSortSpecification() setSortSpecification()	ソート条件を決定します。
getItems() getItems(int index)	検索を実行して LibraryObject の配列を戻します。
getItemCount() openItems() closeItems() resetItems() nextItem()	検索結果を操作します。 ■ getItemCount() は、結果の配列に含まれる項目数を決定します。 ■ openItems() は、Selector 上でカーソルをオープンします。 ■ nextItem() は、カーソルを使用して結果の配列内で次の項目をフェッチします。 ■ resetItems() は、結果の消去に使用されます。 ■ closeItems() は、Selector をクローズします。

SortSpecification

`SortSpecification` クラスは、`setSortSpecification()` メソッドを使用して Selector で設定されたソート条件の構成に使用されます。表 8-2 に、`SortSpecification` クラスの主要メソッドを示します。

表 8-2 SortSpecification の主要メソッド

メソッド	用途
addSortQualifiers()	結果のソートに使用する属性とソート順序の配列を追加します。
addSortQualifier()	SortSpecification に、単一の属性、ソート順序およびオプションで属性のクラスと別名を追加します。
getSortAttributes()	SortSpecification に現在含まれているすべての属性の配列を戻します。
getSortOrders()	SortSpecification に現在含まれているすべてのソート順序の配列を戻します。
getDefaultClass() setDefaultClass()	SortSpecification にある属性のデフォルト・クラスを決定します。
getDefaultAlias() setDefaultAlias()	SortSpecification にある属性のデフォルト別名を決定します。

Selector の構成と実行

Selector を構成して、実行し、問合せ結果を取り出す手順は、次のとおりです。

1. Selector を構成します。
2. クラス名を指定します。
3. このクラスのサブクラスを含めるかどうかを指定します。
4. 問合せ条件を指定します。
5. SortSpecification を構成して Selector 上で設定し、ソート条件を指定します。
6. 問合せを実行して問合せ結果を取り出します。

Selector の構成

最初の手順は、`Selector` クラスをインスタンス化することです。

```
Selector selector = new Selector();
```

注意： このクラスのインスタンスは永続的ではありません。同じ Selector を複数セッションで使用するには、保存できる SelectorObject を使用することを考慮してください。このクラスの詳細は、`oracle.ifs.beans.SelectorObject.` の Javadoc を参照してください。

クラス名の指定

FOLDER や DOCUMENT など、問合せで選択する必要のあるクラスの名前を指定します。指定したクラスは、Oracle 9*i*FS で生成される SQL FROM 句に挿入されます。次の例では、Selector は `Attribute` クラスのインスタンスを問い合せます。

```
selector.setSearchClassname("ATTRIBUTE");
```

サブクラスを含めるかどうかの指定

Selector で指定したクラスのサブクラス間で再帰的に問い合せる必要があるかどうかを指定します。

```
selector.setRecursiveSearch(false);
```

問合せ条件の指定

SQL の WHERE 句になる文字列（ワード WHERE は除く）を構成して問合せ条件を指定し、文字列を Selector の `setSearchSelection()` メソッドに渡します。次の例では、Selector は ATTRIBUTE クラスのうち、DATATYPE が Boolean のすべてのインスタンスを検索します。

```
s = "DATATYPE = " + Attribute.ATTRIBUTEDATATYPE_BOOLEAN;
selector.setSearchSelection(s);
```

ソート条件の指定

問合せ結果のソート順序条件を保持するように SortSpecification を構成します。SortSpecification で指定する属性は、Selector に指定したクラスの属性と一致する必要があります。

1. メソッド `new SortSpecification()` がコールされ、SortSpecification がインスタンス化されます。SortSpecification オブジェクトには、すべてのソート条件が保持されます。
2. SortSpecification のメソッド `addSortQualifiers()` がコールされ、SortSpecification に属性とソート順序の配列が作成されます。次の例では、Selector は検索結果を ATTRIBUTE クラスの NAME および REQUIRED 属性順にソートします。

3. Selector の `setSortSpecification()` メソッドがコールされ、検索にソート条件が追加されます。

```
ss = new SortSpecification();
ss.addSortQualifiers(new String[] {"NAME", "REQUIRED"}, new boolean[] {false,
true});
selector.setSortSpecification(ss);
```

問合せの実行と問合せ結果の取出し

Selector を実行して問合せ結果を取り出すには、次の 2 つの方法があります。

1. 配列の使用
2. カーソルの使用

配列の使用 ： `getItems()` メソッドをコールして Selector を実行し、問合せ結果を配列として操作できます。その後は、配列をループして各問合せの結果を取り出すことができます。

```
LibraryObject[] results = selector.getItems();
LibraryObject lo;

int count = (results == null) ? 0 : results.length;
for (int i = 0; i < count; i++)
{
    lo = results[i];
}
```

`getItems()` メソッドで索引の位置を指定し、配列から特定項目をフェッチできます。

```
LibraryObject lo = selector.getItems(i);
```

カーソルの使用 ： 検索で多数の結果が戻されることが予想されるが、すべてを一度に表示しない場合は、カーソルを使用できます。

1. 最初に、`resetItems()` メソッドをコールしてカーソルをリセットします。
2. Selector で `openItems()` メソッドをコールし、検索のカーソルをオープンします。
3. `getItemCount()` メソッドを使用すると、問合せから戻される結果の数を指定できます。
4. Selector で `nextItem()` メソッドをコールして、個々の項目へと順に切り替えます。
5. 終了後に、Selector で `closeItems()` メソッドをコールして、カーソル・ベースのアクセスの後に検索をクローズします。

```
selector.resetItems();
selector.openItems();
```

```
int count = selector.getItemCount();
LibraryObject lo;

for (int i = 0; i < count; i++)
{
  lo = selector.nextItem();
}

selector.closeItems();
```

サンプル・コード

例 8-2 に、次の SQL 問合せのように、Oracle 9*i*FS 内の Attribute のうち、Boolean データ型の Attribute をすべて問い合せて結果をソートするように、Selector を構成して実行します。

```
SELECT * FROM ATTRIBUTE WHERE DATATYPE = BOOLEAN
ORDER BY NAME DESC, REQUIRED
```

例 8-2 Selector の構成と実行

1. Selector を構成します。

```
Selector selector = new Selector(session);
```

2. クラス名を指定します。

```
selector.setSearchClassname("ATTRIBUTE");
```

3. サブクラスを含めるかどうかを指定します。

```
selector.setRecursiveSearch(false);
```

4. 問合せ条件を指定します。

```
String s = "DATATYPE = " +
  Attribute.ATTRIBUTEDATATYPE_BOOLEAN;
selector.setSearchSelection(s);
```

5. 問合せ結果のソート順序を指定するように SortSpecification を構成します。

```
SortSpecification ss = new SortSpecification();
ss.addSortQualifiers(new String[] {"NAME", "REQUIRED"},
                     new boolean[] {false, true});
selector.setSortSpecification(ss);
```

6. 問合せを実行して問合せ結果を取り出します。

```
LibraryObject[] attributes = selector.getItems();
```

7. 結果をループします。

```
Attribute att;
int count = (attributes == null) ? 0 : attributes.length;
for (int i = 0; i < count; i++)
{
  att = (Attribute) attributes[i];
}
```

8. 索引を使用して、結果に含まれる最初と最後のオブジェクトをフェッチします。

```
att = (Attribute) selector.getItems(0);
att = (Attribute) selector.getItems(count - 1);
```

9. カーソルを使用して結果をフェッチします。

```
selector.resetItems();
selector.openItems();
count = selector.getItemCount();

for (int i = 0; i < count; i++)
{
   attribute = (Attribute) selector.nextItem();
}
```

Search の操作

単純な問合せ条件を指定して単一クラスの情報を問い合せる場合は、Selector を使用すれば十分です。ただし、セレクタでは、複数のクラスの結合、内容条件と関係条件の指定またはバインド変数の使用を伴う問合せは構成できません。

Search には、リポジトリ内で複合問合せを構成して実行できるように、より確実なインタフェースが用意されています。Search では、属性、内容、フォルダ、他の関係および非定型メタデータに関する条件に基づいて、複数クラスの情報を問合せできます。検索のコンポーネントを動的にアセンブルし、バインド変数で渡すことができます。また、後で検索の実行に使用できるように、検索条件を永続的に保存することもできます。

例 8-3 複合問合せ

```
find all Documents and Folders that are in my Home Folder, whose ModifiedDate is
less than <VariableDate>, and whose Name contains 'XML' or whose Content contains
'XML'
```

Search オブジェクト・モデル

Oracle 9*i*FS の Search オブジェクト・モデルでは、次の 4 つのプライマリ Java オブジェクトを使用して問合せを構成して実行し、検索結果を操作し、検索条件を保存しています。

- SearchSpecification
- Search
- SearchResultObject
- SearchObject

SearchSpecification

Search オブジェクト・モデルでは、SearchSpecification を使用してオブジェクト指向の方法で複合問合せをアセンブルできます。SearchSpecification は、問合せのコンポーネントを動的にアセンブルするために使用される Java 構造体です。図 8-2「Search オブジェクト・モデル」に、FROM 句、WHERE 句、WHERE 句の各 EXPRESSION および ORDER BY 句など、リポジトリに対する問合せ用の SQL SELECT 文の各コンポーネントを Java オブジェクトとして表す方法を示します。この各種コンポーネントは SearchSpecification にアセンブルされてから、問合せの実行に使用される Search オブジェクトに渡されます。Oracle 9*i*FS では、問合せの実行時に SQL SELECT 文が自動的に生成され、リポジトリに対する問合せが実行されます。その後、Oracle 9*i*FS では問合せ結果がオブジェクトの配列として戻され、それを読み取って操作できます。

図 8-2 Search オブジェクト・モデル

表 8-3 に、SearchSpecification のコンポーネントと SQL 問合せのコンポーネントとのマッピングを示します。

表 8-3 SQL と Java の比較

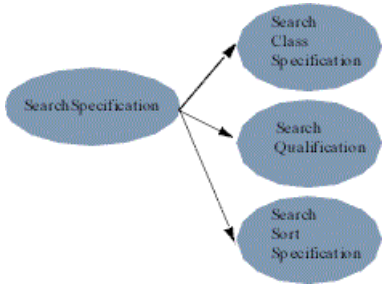
SELECT <result> FROM <list>	SearchClassSpecification
WHERE	SearchQualification
(	SesarchClause
<expression>	AttributeQualification、FolderRestrictQualification、ContextQualification、ExistenceQualification、FreeFormQualification または PropertyQualification
<join>	JoinQualification
(	SearchClause
<expression>	AttributeQualification、FolderRestrictQualification、ContextQualification、ExistenceQualification、FreeFormQualification または PropertyQualification
<join>	JoinQualification
<expression>	AttributeQualification、FolderRestrictQualification、ContextQualification、ExistenceQualification、FreeFormQualification または PropertyQualification
ORDER BY <list>	SearchSortSpecification

このように、Oracle 9*i*FS の Search インタフェースでは、複合問合せをオブジェクト指向の方法でプログラムによりアセンブルできます。Java 構造を使用して問合せを定義すると、各コンポーネントを柔軟に操作し、問合せを動的に変更し、リポジトリに対して複数回実行できます。また、Oracle 9*i*FS の基礎となるスキーマの複雑さを知る必要もなくなります。

Oracle 9*i*FS では、次の Java オブジェクトを使用して SearchSpecification のコンポーネントが構成されます。

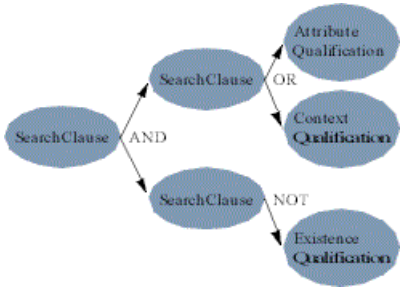
- **SearchSpecification** は、検索の構造体全体を表します。SearchSpecification には、AttributeSearchSpecification および ContextSearchSpecification という 2 つのタイプがあります。図 8-3 のように、SearchSpecification は、SearchClassSpecification、SearchQualification および SearchSortSpecification という 3 つのサブコンポーネントから構成されています。

図 8-3 SearchSpecification のコンポーネント



- **SearchClassSpecification** は、検索の 'SELECT' 句および 'FROM' 句に含まれるクラスを表します。Oracle 9*i*FS は情報をオブジェクト指向の方法で操作するため、SelectClassSpecification では、SQL のように表やフィールドのリストを指定するのではなく、問合せ対象となるクラスのリストを指定します。SearchClassSpecification は、結果クラスおよび検索クラスという 2 つのクラス・セットを保持します。結果クラスは 'SELECT' 句（SELECT Document など）の構成に使用されます。検索から戻されるオブジェクトは、結果クラスのインスタンスとなります。検索クラスは 'FROM' 句（FROM Document、ContentObject など）の構成に使用されます。検索条件は、検索クラスに関連している必要があります。
- **SearchQualification** は検索条件を表します。Oracle 9*i*FS では、SQL SELECT 文の 'WHERE' 句を構成するために使用されます。SearchQualification では、owner = 'WTalman' や contains('XML') など、様々な条件に基づいて検索条件を指定できます。図 8-4 のように、条件は、SearchQualification 内で条件を表すオブジェクトのグループをアセンブルして定義します。AttributeQualification、FolderRestrictQualification、ContextQualification、ExistenceQualification、FreeFormQualification、PropertyQualification および JoinQualification は、様々なタイプの条件を表すために使用されます。SearchQualification 内の条件は、SearchClause オブジェクトを使用して結合されます。

図 8-4 SearchQualification のアセンブリ



- **AttributeQualification** は、属性の値に関する条件（"find all documents whose owner is 'WTalman'" など）を表します。属性は、Oracle 9*i*FS でサポートされるデータ型であれば、どのような型でもかまいません。
- **FolderRestrictQualification** は、検索対象を指定のフォルダまたはその分岐に限定する条件（"find all documents in my home folder" など）を表します。
- **ContextQualification** は、ドキュメントのテキストに関する条件（"find all documents containing the word 'XML' " など）を表します。ContextQualification は、SQL SELECT 文の WHERE 句で CONTAINS() 式を構成するために使用されます。
- **ExistenceQualification** は、属性の値が可能な値のリストの 1 つと一致するという条件（"find all reports where the report number is one of the following: 3a, 3b, 6a, 7c" など）、または、属性の値が別のオブジェクトの属性の値と一致するという条件（"find all documents where the name of a document matches the name of a folder" など）を表します。ExistenceQualification は、SQL SELECT 文の WHERE 句で IN 式を構成するために使用されます。
- **PropertyQualification** は、PropertyBundle 内のプロパティに関する条件（"find all documents where the italian_title_translation property = 'Vino di Italia' " など）を表します。Oracle 9*i*FS では、オブジェクトに関するメタデータを Attribute として格納できるのみでなく、非定型メタデータの名前 / 値のペアで構成される PropertyBundle に格納することもできます。PropertyQualification は、オブジェクトの Property に関する条件を指定するために使用されます。Oracle 9*i*FS では、PropertyQualification は SQL 式である exists (select list of properties where name = <Name> and type = <Type> and value <oper> <Value>) のように変換されます。
- **FreeFormQualification** は、WHERE 句に非定型 SQL 式を含む文字列を表します。FreeFormQualification を使用すると、SQL 構文の自由形式の文字列を通じて他のあらゆるタイプの条件を組み込むことができます。指定した SQL 構文は、Oracle 9*i*FS により生成される SQL SELECT 文に挿入され、逆解析されます。
- **JoinQualification** は、SearchSpecification に関連オブジェクトに関する条件が含まれている場合の WHERE 句内の結合を表します。たとえば、問合せ "find all documents

where owner's e-mail address is 'roadrunner@acme.com'" では、ドキュメントのメタデータを所有者に関するメタデータと結合する必要があります。JoinQualification を使用すると、ドキュメントのメタデータをユーザーのメタデータと結合する方法（"select * from Document d, DirectoryUser u WHERE d.owner = u.id" など）を指定できます。

- **SearchClause** は、SQL SELECT 文の 'WHERE' 句内の条件を結合するために使用されます。SearchClauses では、条件を表す 2 つのオブジェクトを論理演算子で結合できます（"find all documents where name = 'User Guide' AND modification_date > 'January 01, 2001' " など）。また、NOT 演算子を使用して単一条件に関する否定条件を指定することもできます（"find all documents that are NOT in my home folder " など）。SearchClauses では、他の SearchClauses を結合して 3 つ以上の条件を 1 つの SearchQualification にネストできます。

注意： 演算子と優先順位の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

- **SearchSortSpecification** は、検索結果のソート順序を表します。Oracle 9*i*FS では、SQL SELECT 文の ORDER BY 句を構成するために使用されます。

Search

Oracle 9*i*FS の Search オブジェクト・モデルには、問合せの実行用にもう 1 種類のオブジェクトである Search が用意されています。SearchSpecification は、アセンブル後に新規検索の作成に使用されます。次に、Search が次の操作に使用されます。

1. 検索条件の遅延バインド変数を渡します。
2. 検索に使用する言語を指定します。
3. 問合せを実行します。
4. 結果を取得します。

SearchResultObject

検索の実行時に、Oracle 9*i*FS は検索条件と一致したオブジェクトごとに、ヒットした結果の配列を戻します。ヒットした結果は、SearchSpecification に指定したソート条件に従って配列内でソートされます。また、それぞれ配列内で SearchResultObject で表されます。SearchResultObject を使用すると、ヒットした結果がある実際のオブジェクトを取り出すことができます。

戻されるオブジェクトは、SearchClassSpecification で指定した結果クラスのインスタンスです。各オブジェクトは、オブジェクトのクラスに関連する属性と内容で構成されます。

SearchResultObject からオブジェクトを取り出すと、検索アプリケーションでは、そのオブジェクトの属性と内容を表示して操作でき、リポジトリからオブジェクトを取り出すために

第 2 のコールを行う必要はありません。ただし、検索アプリケーションで関連オブジェクト（親フォルダなど）の表示または操作が必要な場合は、第 2 の操作を実行し、リポジトリから関連オブジェクトを取り出す必要があります。

SearchObject

SearchSpecification は、Oracle 9*i*FS に SearchObject として永続的に保存できます。SearchObject には検索条件が格納されるため、ユーザーはいつでもその検索を再構成して実行できます。検索条件のみが保存されるため、検索ではその検索条件を現在満たしているオブジェクトが動的に戻されます。

SearchObject は、ユーザーがリポジトリ内の情報にアクセスするための強力な手段を提供します。たとえば、特定のトピックへの関連性を示す属性と内容に関する条件を格納し、そのトピックに関連する情報を検索するように SearchObject を定義できます。調査担当は、SearchObject を使用すると、そのトピックに関する情報を簡単に検索できます。また、SearchObject を使用すると、リポジトリを共通の属性、内容または関係を持つオブジェクト・セットに動的に編成する仮想フォルダを実装できます。リポジトリ内で新規オブジェクトが作成されると、そのオブジェクトは関連する仮想フォルダ内で自動的に参照されます。このように、SearchObject は強力なリポジトリ編成方法を提供します。ユーザーがオブジェクトを手作業でフォルダに格納する必要はありません。

SearchObject クラスは PublicObject を拡張するため、ユーザーは AccessControlList を適用して SearchObject を共有できます。AccessControlList は、ユーザーとグループが SearchObject にアクセスできる方法を指定します。また、SearchObject は、ユーザーがアクセスしやすいようにフォルダに入れることもできます。

PublicObject に AccessControlList を適用する方法の詳細は、第 15 章「セキュリティ」を参照してください。

注意： このリリースの Oracle 9*i*FS のクライアントでは、SearchObject へのアクセスはサポートされません。ユーザーが SearchObject を操作できるように、カスタム・ユーザー・インタフェースを構築する必要があります。

Search API

Oracle 9*i*FS Java API には、カスタム・アプリケーションで検索を構成して実行し、検索結果を操作できるように、Java クラスのセットが用意されています。`oracle.ifs.search` パッケージには、検索構成用の次の Java クラスが含まれています。

- SearchSpecification
- AttributeSearchSpecification
- ContextSearchSpecification

- SearchClassSpecification
- SearchQualification
- AttributeQualification
- ContextQualification
- FolderRestrictQualification
- ExistenceQualification
- PropertyQualification
- FreeFormQualification
- JoinQualification
- SearchClause
- SearchSortSpecification

`oracle.ifs.beans` パッケージには、検索の実行、検索結果の取出しおよび検索条件の保存用に、次の 3 つの Java クラスが含まれています。

- Search
- SearchResultObject
- SearchObject

SearchSpecification

`SearchSpecification` は、検索のすべてのコンポーネントをアセンブルするための抽象クラスです。このクラスは、`AttributeSearchSpecification` および `ContextSearchSpecification` により拡張されます。`SearchSpecification` クラスをインスタンス化するかわりに、検索条件に応じて、そのサブクラスのどちらかをインスタンス化します。

注意： `SearchSpecification` を使用すると、リポジトリ内のオブジェクトのカスタム SQL ビューを作成できます。このビューの作成手順は、『Oracle9*i* データベース管理者ガイド』を参照してください。

AttributeSearchSpecification

`AttributeSearchSpecification` クラスは、ドキュメントの内容に関する条件を含まない検索の構成に使用されます。

Oracle 9*i*FS では、条件の性質に従って検索が自動的に最適化されます。内容条件（"find all documents whose owner is 'user1'" など）を含まない検索の場合は、

ContextSearchSpecification で構成するよりも、AttributeSearchSpecification で構成する方が効率的になります。

表 8-4 に、AttributeSearchSpecification クラスで最も使用頻度の高いメソッドを示します。

表 8-4 AttributeSearchSpecification のメソッド

getSearchClassSpecification() setSearchClassSpecification()	検索対象となる情報のクラスを指定する SearchClassSpecification を設定します。
getSearchQualification() setSearchQualification()	検索対象となる情報のクラスを指定する SearchQualification を設定します。
getSearchSortSpecification setSearchSortSpecification()	検索結果のソート順序を指定する SearchSortSpecification を設定します。

ContextSearchSpecification

ContextSearchSpecification クラスは、内容条件（"find all documents whose content contains 'XML'" など）を含む検索を構成する場合に使用されます。ContextSearchSpecification の SearchQualification コンポーネントには、1 つ以上の ContextQualification が必要です。

注意： Context は Oracle Text の旧称で、Oracle 9*i*FS によりリポジトリ内のドキュメントの内容に索引付けするためにオプションで使用されるデータベース機能です。

ContextSearchSpecification は AttributeSearchSpecification を拡張するため、表 8-4 に定義したのと同じメソッドを持ちます。表 8-5 に、ContextSearchSpecification クラスに用意されているその他のメソッドを示します。

表 8-5 ContextSearchSpecification のメソッド

setContextClassname() getContextClassname()	ContextQualification に指定した条件に基づいて検索されるクラスを指定します。デフォルトでは、Oracle 9*i*FS ではすべての Document の内容が ContentObject のインスタンスとして格納されます。ContentObject は属性 Content を持つクラスであり、この属性にはドキュメントの実際の内容が格納されます。

SearchClassSpecification

SearchClassSpecification クラスは、検索対象となるクラスの指定に使用されます。また、検索を指定のクラスのサブクラス間で再帰的に実行する必要があるかどうかも指定できます。検索に複数クラスの情報を結合する条件が含まれている場合は、SearchClassSpecification を使用すると戻されるクラスを指定できます。表 8-6 に、SearchClassSpecification クラスで最も使用頻度の高いメソッドを示します。

表 8-6 SearchClassSpecification のメソッド

addSearchClass() addSearchClasses() getSearchClassnames()	内容条件が適用される特定の情報クラスを決定します。そのクラスの別名指定にも使用されます。
getSearchClassAliases()	SearchClassSpecification 内のクラスの別名のリストを取得します。
getRecursiveBehavior()	検索をクラスのサブクラス間で再帰的に適用する必要があるかどうかを決定します。
addResultClass() getResultClassnames()	検索で戻される必要のある情報のクラスを決定します。検索で戻される情報のクラスは、Classes 属性でも指定する必要があります。

SearchQualification

SearchQualification は、検索条件をアセンブルするための抽象クラスです。AttributeQualification、FolderRestrictQualification、ContextQualification、ExistenceQualification、FreeFormQualification、PropertyQualification、JoinQualification および SearchClause は、このクラスを拡張し、各種条件の構成に使用されます。すべての条件が最終的に単一の SearchQualification に結合され、SearchSpecification の setSearchQualification() メソッドに渡されます。

AttributeQualification

AttributeQualification クラスは、属性値に関する条件の構成に使用されます。

表 8-7 に、AttributeQualification クラスで最も使用頻度の高い定数とメソッドを示します。

表 8-7 AttributeQualification の定数とメソッド

getAttributeName() getAttributeClassname() setAttribute()	条件の基礎となる属性を決定します。
getOperatorType() setOperatorType() EQUAL GREATER_THAN GREATER_THAN_EQUAL IS_NOT_N ULL IS_NULL LESS_THAN LESS_THAN_EQUAL LIKE NOT_EQUAL	属性を指定の値と比較する演算子を決定します。 定数は、条件中の比較演算子を表します。現在、Oracle 9*i*FS でサポートされている演算子は、文字列に対する 'IS_NULL'、'IS_NOT_NULL' および 'LIKE' のみです。
getValue() setValue()	属性の比較対象となる値を決定します。 **注意：** 演算子 LIKE を使用すると、有効な SQL ワイルドカード（% や _ など）を値に使用できます。
getDateComparisonLevel() setDateComparisonLevel() DATE_COMP_DAY DATE_COMP_HOUR DATE_COMP_MIN DATE_COMP_MONTH DATE_COMP_SEC DATE_COMP_YEAR	DATE 型の属性の場合、これらのメソッドと定数では Oracle 9*i*FS で属性と日付値を比較するレベルを指定します。 たとえば、AttributeQualification を使用すると、'CreateDate' Attribute を、'March 21, 2001 08:30 am'（'CreateDate < SYSDATE' など）に設定されているシステム生成の日付 'SYSDATE' と比較できます。ただし、2001 年 3 月 20 日以前、つまり 2001 年 3 月 20 日 23:59 p.m. 以前に作成されたすべての情報を取り出す必要がある場合があります。AttributeQualification で 2001 年 3 月 21 日 7:00 a.m. に作成された情報が戻されるのを防ぐには、日付の比較レベルを DATE_COMP_DAY に設定します。
isCaseIgnored() setCaseIgnored()	条件に大 / 小文字区別が必要かどうかを決定します。 たとえば、IgnoreCase を 'true' に設定すると、条件 NAME = 'XML' では NAME が 'xml' の情報は戻されません。

ContextQualification

ContextQualification クラスは、ドキュメントの内容に関する条件の構成に使用されます。

表 8-8 に、ContextQualification クラスで最も使用頻度の高い定数とメソッドを示します。

表 8-8 ContextQualification の定数とメソッド

getName() setName()	ContextQualification の Name を決定します。名前を使用して、Oracle Text でソート条件として生成される関連性スコアを組み込むことができます。
ORDER_PREFIX	Oracle Text で生成される関連性スコアを SearchSortSpecification に組み込む場合に使用します。定数は ContextQualification の Name に連結され、Oracle 9*i*FS により ORDER BY 句に正しい SQL 構文を生成するために使用されます。
getQuery() setQuery()	Oracle Text に内容検索の条件として渡されるテキスト問合せ文字列を決定します。問合せ文字列は、Oracle 9*i*FS から CONTAINS() SQL ファンクションのテキスト文字列引数にそのまま渡され、このファンクションにより Oracle Text がコールされ、内容の問合せが実行されます。問合せ文字列は Oracle 9*i*FS で変更されないため、Oracle Text で要求される構文に従って、必要な数のトークン、式演算子および結合演算子を含めることができます。

注意： Oracle Text に対するテキスト問合せ文字列の構文規則の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

FolderRestrictQualification

FolderRestrictQualification は、検索対象を指定のフォルダまたはその分岐に限定する条件を構成するために使用されます。

表 8-9 に、FolderRestrictQualification クラスで最も使用頻度の高いメソッドを示します。

表 8-9 FolderRestrictQualification のメソッド

getStartingFolder() setStartFolder()	検索の限定対象となるフォルダを決定します。フォルダ階層の複数レベルの分岐にまたがって検索する場合、StartFolder は分岐の最上位ノードです。

表 8-9 FolderRestrictQualification のメソッド（続く）

isMultiLevel() setMultiLevel()	検索で StartFolder のサブフォルダを横断するかどうかを決定します。
getSearchClassname() setSearchClassname()	フォルダ内で検索する情報のクラスを決定します。Search クラスは、FolderRestrictQualification とともに SearchSpecification に自動的に含まれます。

ExistenceQualification

`ExistenceQualification` は、属性値が可能な値のリストの 1 つと一致するという条件、または属性値が別のオブジェクトの属性の値と一致するという条件を構成するために使用されます。

表 8-10 に、`ExistenceQualification` クラスで最も使用頻度の高いメソッドを示します。

表 8-10 ExistenceQualification のメソッド

getLeftAttributeName() getLeftAttributeClassname() setLeftAttribute()	条件の左辺で、値が条件の右辺に指定された値の 1 つと一致する必要のある属性を決定します。
getRightAttributeName() getRightAttributeClassname() setRightAttribute()	属性を経由して条件の右辺の値のリストを決定します。
getRightAttributeValue() isRightAttributeValue() setRightAttributeValue()	`AttributeValue[]` 配列を経由して条件の右辺の値のリストを決定します。

PropertyQualification

`PropertyQualification` は、オブジェクトの PropertyBundle 内のプロパティに関する条件を表します。

表 8-11 に、`PropertyQualification` クラスで最も使用頻度の高いメソッドを示します。

表 8-11 PropertyQualification のメソッド

getPropertyName() setPropertyName()	検索対象となるプロパティの名前を決定します。

表 8-11 PropertyQualification のメソッド（続く）

getOperatorType() setOperatorType()	プロパティを指定の値と比較するために使用する演算子を決定します。このメソッドは、AttributeQualification に定義された定数（EQUAL など）を受け取って、条件内で比較演算子を表します。現在、Oracle 9*i*FS でサポートされている演算子は、文字列に対する 'IS_NULL'、'IS_NOT_NULL' および 'LIKE' のみです。
getValue() setValue()	属性の比較対象となる値を決定します。演算子 LIKE を使用すると、有効な SQL ワイルドカード（% や _ など）を値に使用できます。
getDateComparisonLevel() setDateComparisonLevel()	DATE 型のプロパティの場合、これらのメソッドでは Oracle 9*i*FS で属性と日付値を比較するレベルを指定します。このメソッドは、AttributeQualification に定義された定数（DATE_COMP_DAY など）を受け取って、条件内で様々な比較レベルを表します。 たとえば、PropertyQualification を使用すると、'PublicationDate' プロパティを、'March 21, 2001 08:30 am'（'PublicationDate < SYSDATE' など）に設定されているシステム生成の日付 'SYSDATE' と比較できます。ただし、2001 年 3 月 20 日以前、つまり 2001 年 3 月 20 日 23:59 p.m. 以前に作成されたすべての情報を取り出す必要がある場合があります。AttributeQualification で 2001 年 3 月 21 日 7:00 a.m. に作成された情報が戻されるのを防ぐには、日付の比較レベルを DATE_COMP_DAY に設定します。
isCaseIgnored() setCaseIgnored()	条件に大 / 小文字区別が必要かどうかを決定します。 たとえば、IgnoreCase を 'true' に設定すると、条件 NAME = 'XML' では NAME が 'xml' の情報は戻されません。
isLateBound() getLateBoundDataType() setLateBoundDataType()	プロパティの値が遅延バインド変数で与えられるように指定します。
setClassname() getClassname()	PropertyBundle を持つ情報のクラスを決定します。Search クラスは、PropertyQualification とともに SearchSpecification に自動的に含まれます。

FreeFormQualification

FreeFormQualification クラスは、SQL 構文の自由形式の文字列を通じて他のタイプの条件を組み込むために使用されます。

表 8-12 に、FreeFormQualification クラスで最も使用頻度の高いメソッドを示します。

表 8-12 FreeFormQualification のメソッド

getSQLExpression() setSQLExpression()	SQL 式を含む String を決定します。

JoinQualification

JoinQualification は、関連するオブジェクトに関する条件を含める場合の結合の構成に使用します。

表 8-13 に、JoinQualification クラスで最も使用頻度の高いメソッドを示します。

表 8-13 JoinQualification のメソッド

getLeftAttributeName() getRightAttributeName() setLeftAttribute() setRightAttribute()	結合条件の左辺と右辺を決定します。
getLeftAttributeClassname() getRightAttributeClassname()	結合条件に指定したどちらかの属性のクラスを取得します。

SearchClause

SearchClause は、SearchQualification のコンポーネントをアセンブルするために使用されます。SearchClause は、論理演算子を使用して条件を結合します。SearchClause は他の SearchClauses を結合できるため、複数の SearchClauses を単一の SearchQualification にネストできます。

表 8-14 に、SearchClause クラスで最も使用頻度の高い定数とメソッドを示します。

表 8-14 SearchClause の定数とメソッド

getLeftSearchQualification() setLeftSearchQualification()	SearchClause の左辺の条件を決定します。

表 8-14 SearchClause の定数とメソッド（続く）

getOperatorType() setOperatorType() AND OR NOT	SearchClause で条件の結合に使用する演算子を決定します。 これらの定数は、句の演算子を表すために使用します。
getRightSearchQualification() setRightSearchQualification()	SearchClause の右辺の条件を決定します。

SearchSortSpecification

`SearchSortSpecification` は、検索結果のソート順序を構成します。

表 8-15 に、`SearchSortSpecification` クラスで最も使用頻度の高い定数とメソッドを示します。

表 8-15 SearchSortQualification の定数とメソッド

add()	SearchSortSpecification に 1 つ以上のソート条件を追加します。
getClassnames()	検索結果の順序付けに使用される属性のクラスを決定します。
getAttributesNames()	検索結果の順序付けに使用される属性を決定します。
getOrders() ASCENDING DESCENDING	属性に基づき、結果の順序として昇順または降順を決定します。 これらの定数は、SearchSortSpecification に指定された順序を表すために使用します。

Search

`Search` クラスは、問合せを実行して結果を取り出すために使用されます。SearchSpecification が構成されると、Search の `setSearchSpecification()` メソッドに渡されます。その後、`open()` メソッドがコールされてバインド変数が渡され、検索が実行されます。この時点で、Oracle 9*i*FS は SearchSpecification から SQL SELECT 文をコンパイルし、Oracle 9*i*FS スキーマに対して問合せを実行します。

表 8-16 に、`Search` クラスで最も使用頻度の高いメソッドを示します。

表 8-16 Search の属性およびメソッド

getSearchSpecification() setSearchSpecification()	Search の問合せ条件を保持する SearchSpecification を決定します。
open()	遅延バインド変数を渡し、問合せを実行し、検索結果へのカーソルをオープンします。
close()	検索のカーソルをクローズします。
dispose()	検索のカーソルをクローズし、そのリソースを完全に解放します。dispose 後の Search は reopened() できません。
next() getItemCount()	検索結果を取得します。

SearchResultObject

`SearchResultObject` クラスは、検索結果の操作に使用されます。`next()` メソッドは、単一の検索結果を表す SearchResultObject を戻します。SearchResultObject は、検索条件を満たすオブジェクトの取出しに使用されます。

表 8-17 に、`SearchResultObject` クラスで最も使用頻度の高いメソッドを示します。

表 8-17 SearchResultObject のメソッド

getLibraryObject()	SearchResultObject など、検索でヒットした結果がある LibraryObject を戻します。
getScore()	Oracle Text により生成されたスコアを戻します。このスコアは、ContextQualification に指定された内容条件に対するドキュメントの関連性を示します。

SearchObject

`SearchObject` クラスは、検索条件をリポジトリに永続的に保存するために使用されます。SearchObject には SearchSpecification が格納されるため、それを使用して検索を再構成し、複数回実行できます。

表 8-18 に、`SearchObject` クラスで最も使用頻度の高いメソッドを示します。

表 8-18 SearchObject のメソッド

getSearch()	SearchObject により格納された SearchSpecification から新規の Search を構成します。

表 8-18 SearchObject のメソッド（続く）

getSearchSpecification() setSearchSpecification()	SearchObject に格納された検索条件を表す SearchSpecification を決定します。

Search の構成と実行

Search を構成し、実行して保存するには、アプリケーションで次のタスクを実行します。

1. SearchSpecification の構成
2. SearchClassSpecification の構成
3. 条件ごとの SearchQualification の構成
4. 単一の SearchQualification への条件の結合
5. SearchSortSpecification の構成
6. Search の構成と実行
7. SearchResultObject のフェッチ
8. SearchObject としての SearchSpecification の保存

SearchSpecification の構成

検索アプリケーションの構築の最初のステップは、SearchSpecification を構成することです。問合せに Document 内容条件が含まれている場合は、ContextSearchSpecification を構成します。それ以外の場合は、AttributeSearchSpecification を構成します。

例 8-4 AttributeSearchSpecification の構成

```
AttributeSearchSpecification asp = new AttributeSearchSpecification();
```

例 8-5 ContextSearchSpecification の構成

```
ContextSearchSpecification csp = new ContextSearchSpecification();
```

SearchClassSpecification の構成

検索して検索結果に含める情報のクラスを指定するには、SearchClassSpecification を構成して、SearchSpecification の `setClassSpecification()` メソッドに渡します。

例 8-6 SearchClassSpecification の構成

```
// Following code constructs a SearchClassSpecification
// to represent the SELECT and FROM clauses in a query.
//
```

```
//   e.g.,  SELECT Document, Category
//          FROM Document, Category, Folder
```

1. クラス名、別名および再帰動作の配列を作成します。

```
String [] classNames = new String[] {"Document", "Category", "Folder"};
String [] aliasNames = new String[] {"d", "c", "f"};
Boolean [] delBvrs = new Boolean [] {false, false, false};
Boolean [] recBvrs = new Boolean [] {false, true, false};
```

2. SearchClassSpecification を構成してクラス名に渡します。

```
SearchClassSpecification scp =
     new SearchClassSpecification(classNames, aliasNames, delBvrs, recBvrs);
```

3. 検索で戻される情報のクラスを指定します。

```
scp.addResultClass("Document");
scp.addResultClass("Category");
```

4. SearchSpecification に SearchClassSpecification コンポーネントを追加します。

```
asp.setSearchClassSpecification(scp);
```

条件ごとの SearchQualification の構成

問合せ条件の条件ごとに、該当する `SearchQualification` サブクラスのインスタンスを構成します。

- AttributeQualification
- ContextQualification
- FolderRestrictQualification
- JoinQualification
- ExistenceQualification
- PropertyQualification
- FreeFormQualification

AttributeQualification ： 属性値に基づく条件を構成するには、`AttributeQualification` を作成します。AttributeQualification を構成するには、条件の 3 つの主コンポーネント、つまり属性、比較演算子および値を指定します。

例 8-7 文字列値による AttributeQualification の構成

1. AttributeQualification を構成します。

```
AttributeQualification aq1 = new AttributeQualification();
```

2. 条件の 3 つの主コンポーネントを指定します。

```
aq1.setAttribute(PublicObject.NAME_ATTRIBUTE);
aq1.setOperator(AttributeQualification.LIKE);
aq1.setValue("Oracle 9iFS");
```

3. 値の大 / 小文字区別を無視するように指定します。

```
aq1.setCaseIgnored(true);
```

例 8-8 遅延バインド変数による AttributeQualification の構成

1. AttributeQualification を構成します。

```
AttributeQualification aq1 = new AttributeQualification();
```

2. 条件の 3 つの主コンポーネントを指定します。LATE_BIND_OPER 定数で値を設定します。

```
aq1.setAttribute(PublicObject.NAME_ATTRIBUTE);
aq1.setOperator(AttributeQualification.EQUAL);
aq1.setValue(SearchQualification.LATE_BIND_OPER);
```

例 8-9 日付値による AttributeQualification の構成

1. AttributeQualification を構成します。

```
AttributeQualification aq1 = new AttributeQualification();
```

2. 条件の属性と演算子を指定します。

```
aq1.setAttribute(PublicObject.CREATEDATE_ATTRIBUTE);
aq1.setOperator(AttributeQualification.LESS_THAN);
```

3. Date 値を与える AttributeValue を作成します。

```
Date today = new Date();
AttributeValue av = AttributeValue.newAttributeValue(today);
aq1.setValue(av, session);
```

4. Oracle 9*i*FS で属性と日付値を比較するレベルを指定します。

```
aq1.setDateComparisionLevel(AttributeQualification.DATE_COMP_DAY);
```

例 8-10 オブジェクト値による AttributeQualification の構成

1. AttributeQualification を構成します。

```
AttributeQualification aq1 = new AttributeQualification();
```

2. 条件の属性と演算子を指定します。

```
aq1.setAttribute(PublicObject.ACL_ATTRIBUTE);
aq1.setOperator(AttributeQualification.EQUAL);
```

3. Date 値を与える AttributeValue を作成します。

```
Collection aclColl = session.getSystemAccessControlListCollection();
SystemAccessControlList publicAcl =
           (SystemAccessControlList) aclColl.getItems("PUBLIC");
AttributeValue av = AttributeValue.newAttributeValue(publicAcl);
aq1.setValue(av);
```

FolderRestrictQualification ： 検索対象をフォルダが参照するオブジェクトに限定するという条件を指定するには、`FolderRestrictQualification` を構成します。FolderRestrictQualification を使用すると、特定のフォルダ内、またはフォルダ階層の分岐にまたがって検索できます。

例 8-11 FolderRestrictQualification の構成

1. FolderRestrictQualification を構成します。

```
FolderRestrictQualification frq1 = new FolderRestrictQualification();
```

2. 検索の限定対象となるフォルダを指定します。

```
Folder startFolder = session.getPrimaryUserProfile().getHomeFolder();
frq1.setStartFolder(startFolder);
```

3. 検索でフォルダ分岐内のサブフォルダを横断するように指定します。

```
frq1.setMultiLevel(true);
```

4. フォルダから選択される情報のクラスを指定します。

```
frq1.setSearchClassname(Document.CLASS_NAME);
```

例 8-12 遅延バインド変数による FolderRestrictQualification の構成

1. FolderRestrictQualification を構成します。

```
FolderRestrictQualification frq1 = new FolderRestrictQualification();
```

2. 検索の限定対象となるフォルダを指定します。

```
frq1.setStartFolder(SearchQualification.LATE_BIND_OPER);
```

3. フォルダから選択される情報のクラスを指定します。

```
frq1.setSearchClassName(PublicObject.CLASS_NAME);
```

JoinQualification ： 関連オブジェクトに基づいて検索条件を結合するには、`JoinQualification` を構成します。JoinQualification は、等価かどうかがテストされる 2 つの属性で構成されます。JoinQualification を構成する場合は、比較の左辺と右辺について属性とそのクラスを指定します（Document.CreateDate = Folder.CreateDate など）。オブジェクト・データ型を持つ属性に基づいて 2 つのクラスを結合する場合、結合の片側では属性が保持するオブジェクトのクラスのみを指定します（Document.Owner = DirectoryUser など）。Oracle 9*i*FS はオブジェクト指向のシステムのため、オブジェクト・データ型を持つ属性は、オブジェクトの特定の属性を参照するのではなく、そのオブジェクトを参照します。

例 8-13 JoinQualification の構成

1. JoinQualification を構成します。

```
JoinQualification jq1 = new JoinQualification();
```

2. 結合の左辺のクラスと属性を指定します。

```
jq1.setLeftAttribute(Document.CLASS_NAME,PublicObject.OWNER_ATTRIBUTE);
```

3. 結合の右辺のクラスと属性を指定します。属性については 'null' が渡されるため注意してください。これは、暗黙的に ID での結合を示します。

```
jq1.setRightAttribute(DirectoryUser.CLASS_NAME, null);
```

ContextQualification ： ドキュメントの内容、つまり Document クラスとそのサブクラスのインスタンスに関する条件を指定するには、`ContextQualification` を構成します。ContextQualification は、2 つの主コンポーネント、つまりテキスト問合せ文字列と一意名から構成されます。

検索の実行時に、Oracle 9*i*FS は Oracle Text を使用してドキュメントの内容の索引にまたがって検索します。SQL の生成時に、内容条件は CONTAINS 句の形式を取ります（CONTAINS(<index>, <text query string>, <order prefix>) > 0 など）。<index> パラメータは、インストール時に生成された索引に基づいて Oracle 9*i*FS により決定されます。<text query string> は、`setQuery()` メソッドに指定した値からそのまま渡されます。したがって、`setQuery()` メソッドには、テキスト問合せ文字列に関する Oracle Text の構文規則に従った値を指定する必要があります。<order prefix> パラメータは、`setName()` メソッドに指定した値から Oracle 9*i*FS により構成されます。

Oracle Text は、内容条件に基づいてドキュメントを検索するときに、条件との関連性に従ってドキュメントをランク付けします。各ドキュメントには 0 ～ 100 のスコアが与えられ、100 は最も関連性の高いドキュメントを表します。SearchSortSpecification に ContextQualification の Name を含めると、検索結果をスコアに基づいてソートできます。

`ContextQualification` を使用して内容条件を指定する手順は、次のとおりです。

1. ContextSearchSpecification を構成します。
2. 内容条件を保持する ContextQualification を構成します。

3. SearchClassSpecification に、ContentObject クラスまたは ContentObject のサブクラスを含めます。
4. ContentObject クラスを、情報に関する他の条件（AttributeQualifications、FolderRestrictQualifications など）に関連する Document クラスまたは Document サブクラスと結合するように、JoinQualification を構成します。
5. SearchClassSpecification のうち、Oracle Text で検索する内容を含むクラスを指定します。
6. オプションで、結果を Oracle Text から戻される関連性スコア順にソートします。

例 8-14 ContextQualification の構成

1. ContextQualification を構成します。

```
ContextQualification cq1 = new ContextQualification();
```

2. Oracle Text に CONTAINS() ファンクションの <text query string> パラメータとして渡すテキスト問合せ文字列を指定します。

```
cq.setQuery("XML");
```

3. ContextQualification の Name を指定します。Name は、Oracle 9*i*FS で CONTAINS() ファンクションの <order prefix> パラメータを指定するために使用されます。

```
String contextClauseName = "CQ1";
cq.setName(contextClauseName);
```

4. SearchSortSpecification の構成時に、ContextQualification の Name を指定して、Oracle Text により CONTAINS() ファンクションから生成されたスコア順に結果をソートできます。

```
SearchSortSpecification sortSpec;
sortSpec.add(Document.CLASS_NAME,
             ContextQualification.ORDER_PREFIX + "." + ctxClauseName, true);
```

5. 内容は ContentObject として格納されるため、SearchClassSpecification に Document クラスと ContentObject クラスの両方を追加してから、この 2 つのクラスを結合する必要があります。

```
String [] classNames = new String[] {"Document", "ContentObject"};
SearchClassSpecification scp = new SearchClassSpecification(classNames);
scp.addResultClass("Document");

JoinQualification jq1 = new JoinQualification();
jq1.setLeftAttribute(Document.CLASS_NAME, Document.CONTENTOBJECT_ATTRIBUTE);
jq1.setRightAttribute(ContentObject.CLASS_NAME, null);
```

ExistenceQualification ： Attribute が可能な値のリストの 1 つと一致するという条件を含めるには、`ExistenceQualification` を構成します。`ExistenceQualification` を構成するときに、比較の左辺に属性、右辺に値リストを指定します。属性は、どのようなスカラー属性でもかまいません。現在、Oracle 9*i*FS では、比較の左辺として配列型の属性を指定することはサポートされていません。値リストは、スカラー型または配列型の別の属性か、`AttributeValue[]` 配列として指定できます。

例 8-15 文字列値の ExistenceQualification の構成

```
// Suppose you want to look for, all documents
// whose name appears in STRINGVALUE attribute of any PROPERTY
// Generates query - DOCUMENT.Name in (select STRINGVALUE from PROPERTY)
// This shows scalar attributes on both LHS and RHS.
```

1. ExistenceQualification を構成します。

   ```
   ExistenceQualification eq1 = new ExistenceQualification();
   ```

2. 値が値リストに存在する必要のある属性を指定します。

   ```
   eq1.setLeftAttribute(Document.CLASS_NAME, PublicObject.NAME_ATTRIBUTE);
   ```

3. ユーザー指定の値のリストを指定します。

   ```
   eq1.setRightAttribute(Property.CLASS_NAME, Property.STRINGVALUE_ATTRIBUTE);
   ```

例 8-16 配列値の ExistenceQualification の構成

```
// Suppose you want to look for, all documents whose name appears in
// STRINGVALUES array type attribute in a property.
// Generates query - DOCUMENT.Name in (select STRINGVALUES[]
// from PROPERTY)
// This flattens out the array attribute on the right hand side
// and generates a concatenated list of all values in
// STRINGVALUES attribute in each row.
```

1. ExistenceQualification を構成します。

   ```
   ExistenceQualification eq1 = new ExistenceQualification();
   ```

2. 値が値リストに存在する必要のある属性を指定します。

   ```
   eq1.setLeftAttribute(Document.CLASS_NAME, PublicObject.NAME_ATTRIBUTE);
   ```

3. ユーザー指定の値のリストを指定します。

   ```
   eq1.setRightAttribute(Property.CLASS_NAME, Property.STRINGVALUES_ATTRIBUTE);
   ```

例 8-17 ユーザー指定値の ExistenceQualification の構成

```
// Suppose you want to look for documents whose name is
// in {"FOO", "BAR", "NOTES"},
// then you use the EQ in the following manner.
```

1. ExistenceQualification を構成します。

```
ExistenceQualification eq1 = new ExistenceQualification();
```

2. 値が値リストに存在する必要のある属性を指定します。

```
eq1.setLeftAttribute(Document.CLASS_NAME, PublicObject.NAME_ATTRIBUTE);
```

3. ユーザー指定の値のリストを指定します。

```
AttributeValue [] avArray = new AttributeValue[3];
avArray[0] = AttributeValue.newAttributeValue("FOO");
avArray[1] = AttributeValue.newAttributeValue("BAR");
avArray[2] = AttributeValue.newAttributeValue("NOTES");

eq1.setRightAttributeValue(avArray);
```

PropertyQualification ： オブジェクトの PropertyBundle 内のプロパティに関する条件を指定するには、`PropertyQualification` を構成します。PropertyQualification は、プロパティ名、プロパティ値および比較演算子で構成されます。Oracle 9*i*FS で値をデータ型に関係なくプロパティの Value 属性と自動的に比較できるように、値は `AttributeValue` のインスタンスとして指定する必要があります。また、`PropertyQualification` を構成すると、PropertyBundle を持つオブジェクトのクラス用にもう 1 つの `Search` クラスが自動的に追加されます。

例 8-18 文字列値の PropertyQualification の構成

1. PropertyQualification を構成します。

```
PropertyQualification pq = new PropertyQualification();
```

2. Property が存在する情報のクラスを指定します。

```
pq.setClassname(Document.CLASS_NAME);
```

3. Property の Name、Value および比較演算子を指定します。また、Oracle 9*i*FS で値の大 / 小文字の区別を無視する必要があるかどうかを指定します。

```
pq.setPropertyName("color");
pq.setOperatorType(AttributeQualification.EQUAL);
pq.setValue(AttributeValue.newAttributeValue("RED"));
pq.setCaseIgnored(ignoreCase);
```

例 8-19 日付値の PropertyQualification の構成

1. PropertyQualification を構成します。

```
PropertyQualification pq = new PropertyQualification();
```

2. プロパティが存在する情報のクラスを指定します。

```
pq.setClassname(Document.CLASS_NAME);
```

3. プロパティの Name、Value および比較演算子を指定します。また、Oracle 9*i*FS で日付値を比較する必要のあるレベルを指定します。

```
pq.setPropertyName("expire-date");
pq.setOperatorType(AttributeQualification.EQUAL);
pq.setValue(
       AttributeValue.newAttributeValue(
           new GregorianCalendar(1995, 0, 15).getTime()));
pq.setDateComparisonLevel(AttributeQualification.DATE_COMP_MONTH);

          // Using DOCUMENT(or any object type) type attribute value
          // This looks for a particular document in a property
          // containing DOCUMENT objects
```

例 8-20 オブジェクト値の PropertyQualification の構成

1. PropertyQualification を構成します。

```
PropertyQualification pq = new PropertyQualification();
```

2. プロパティが存在する情報のクラスを指定します。

```
pq.setClassname(Document.CLASS_NAME);
```

3. プロパティの Name、Value および比較演算子を指定します。また、Oracle 9*i*FS で日付値を比較する必要のあるレベルを指定します。

```
pq.setPropertyName("related-docs");
pq.setOperatorType(AttributeQualification.EQUAL);
pq.setValue(AttributeValue.newAttributeValue(new Document[] {d}));
```

例 8-21 遅延バインド変数による PropertyQualification の構成

1. PropertyQualification を構成します。

```
PropertyQualification pq = new PropertyQualification();
```

2. プロパティが存在する情報のクラスを指定します。

```
pq.setClassname(Document.CLASS_NAME);
```

3. プロパティの Name、Value および比較演算子を指定します。

```
AttributeValue srchDate =
      AttributeValue.newAttributeValue(
         new GregorianCalendar(1995, 0, 1).getTime());
pq.setPropertyName("expire-date");
pq.setOperatorType(AttributeQualification.EQUAL);
pq.setLateBoundDataType(srchDate.getDataType());
```

FreeFormQualification ： Oracle 9*i*FS により生成される WHERE 句に非定型 SQL 式を含めるには、`FreeFormQualification` を構成します。`FreeFormQualification` クラスには、高度な開発者が Oracle 9*i*FS では構成できない検索条件を指定できるように、フックが用意されています。たとえば、カスタム検索アプリケーションで、Oracle 9*i*FS では自動的にフル・テキスト索引付けができない属性（Description など）の Oracle Text 索引に基づいて、検索を作成する必要があるとします。そのためには、アプリケーションで、その属性に対する CONTAINS() ファンクションを構成する `FreeFormQualification` を構成できます。また、アプリケーションでは、Oracle 9*i*FS API で自動的に生成されない他の SQL ファンクション（SUM など）を使用する必要がある場合もあります。

`FreeFormQualification` に指定した SQL 式の妥当性チェックは、Oracle 9*i*FS では行われません。また、FreeFormQualification では、遅延バインドはサポートされません。したがって、SQL 文字列には ? を使用できません。

FreeFormQualification を使用するには、Oracle 9*i*FS スキーマを十分に理解している必要があります。現行のセッションのユーザーが管理者で、セッションの管理モードがオンになっている場合（`ifsSession.setAdministrationMode(true)` など）、検索アプリケーションでは Oracle 9*i*FS スキーマの保護表へのアクセスのみが可能です。

`FreeFormQualification` により生成された SQL は、Oracle 9*i*FS により生成された WHERE 句に挿入されます。したがって、FreeFormQualification は、DDL 文または DML 文の実行や、完全な SQL SELECT 文の構成には使用できません。

例 8-22 FreeFormQualification の構成

```
// Create a FreeFormQualification to find CONTENTOBJECTS
// that are greater than the average content size.
```

1. FreeFormQualification を構成します。

```
FreeFormQualification ffq= new FreeFormQualification();
```

2. SQL 文字列を構成して FreeFormQualification に渡します。

```
String sqlExpression = "co.contentsize > " +
                       "(select AVG(contentsize) " +
                       "from odm_contentobject c, odm_document d " +
                       "where c.id = d.contentobject)");
ffq.setSqlExpression(sqlExpression);
```

単一の SearchQualification への条件の結合

検索に複数の条件が含まれる場合は、それを単一の SearchQualification に結合します。各条件を対応する `SearchQualification` サブクラスのインスタンスとして構成した後に、SearchClause を構成し、条件を結合してネストした条件ツリーにします。結果的な SearchClause を `setSearchQualification()` メソッドに渡して、SearchSpecification の検索条件を設定します。

SearchClause を使用すると、2 つの条件をブール演算子（AND や OR など）で結合できます。この場合、SearchClause は左辺の条件、右辺の条件およびブール演算子で構成されます。

また、SearchClause では、演算子 NOT を使用して否定文を構成できます。この場合、SearchClause は演算子と右辺の条件で構成されます。

SearchClause を別の SearchClause の左辺または右辺、あるいはその両方として渡すとネストできます（<condition> OR (<condition> AND <condition>) など）。

例 8-23 SearchClause の構成

```
/*
 * SearchClauses are used to construct the following WHERE statement:
 * WHERE (Document.contentobject = ContentObject
 * AND ContentObject.Format = Format)
 * AND (Document.name = ? AND Format.extension = 'html')
 * OR (Document.owner = ? AND Format.name = 'MS Word')
 */
```

1. Document.ContentObject = ContentObject の JoinQualification を構成します。

```
JoinQualification jq1 = new JoinQualification();
jq1.setLeftAttribute("DOCUMENT", "CONTENTOBJECT");
jq1.setRightAttribute("CONTENTOBJECT");
```

2. ContentObject.Format = Format の JoinQualification を構成します。

```
JoinQualification jq2 = new JoinQualification();
jq2.setLeftAttribute("CONTENTOBJECT", "FORMAT");
jq2.setRightAttribute("FORMAT");
```

3. DOCUMENT.NAME および FORMAT.XTENSION の AttributeQualification を構成します。

```
AttributeQualification aq1 = new AttributeQualification();
aq1.setAttribute("DOCUMENT", "NAME");
aq1.setOperatorType("=");
aq1.setValue("?");

AttributeQualification aq2 = new AttributeQualification()
```

```
aq2.setAttribute("FORMAT", "EXTENSION");
aq2.setOperatorType("=");
aq2.setValue("html");
```

4. DOCUMENT.OWNER および FORMAT.NAME の AttributeQualification を構成します。

```
AttributeQualification aq3 = new AttributeQualification();
aq3.setAttribute("DOCUMENT", "OWNER");
aq3.setOperatorType("=");
aq3.setValue("?");

AttributeQualification aq4 = new AttributeQualification();
aq4.setAttribute("FORMAT", "NAME");
aq4.setOperatorType("=");
aq4.setValue("MS Word");
```

5. コンストラクタを使用して各部を SearchClause に結合し、WHERE 句全体を構成します。

```
SearchClause sc1 = new SearchClause(aq1, aq2, SearchClause.AND);
SearchClause sc2 = new SearchClause(aq3, aq4, SearchClause.AND);
SearchClause sc3 = new SearchClause(jq1, jq2, SearchClause.AND);
SearchClause sc4 = new SearchClause(sc1, sc2, SearchClause.OR);
SearchClause scComplete = new SearchClause(sc3, sc4, SearchClause.AND);
```

また、SearchClause クラスには、より厳密な制御により句を構築して取り出すためのメソッドも用意されています。この種のメソッドは、アプリケーションで SearchClause のコンポーネントを再利用したり操作する場合に役立ちます。

例 8-24 SearchClause の構成

1. Document.ContentObject = ContentObject の JoinQualification を作成します。

```
JoinQualification jq1 = new JoinQualification();
jq1.setLeftAttribute("DOCUMENT", "CONTENTOBJECT");
jq1.setRightAttribute("CONTENTOBJECT");
```

2. ContentObject.Format = Format の JoinQualification を作成します。

```
JoinQualification jq2 = new JoinQualification();
jq2.setLeftAttribute("CONTENTOBJECT", "FORMAT");
jq2.setRightAttribute("FORMAT");
```

3. DOCUMENT.NAME および FORMAT.EXTENSION の AttributeQualification を設定します。

```
AttributeQualification aq1 = new AttributeQualification();
aq1.setAttribute("DOCUMENT", "NAME");
aq1.setOperatorType("=");
```

```
aq1.setValue("?");

AttributeQualification aq2 = new AttributeQualification()
aq2.setAttribute("FORMAT", "EXTENSION");
aq2.setOperatorType("=");
aq2.setValue("html");
```

4. clone() メソッドを使用し、同じ Attribute を持つ aq2 に基づいて新規の AttributeQualification を簡単に作成します。

```
AttributeQualification aq3 = aq2.clone();
aq3.setValue("txt");
```

5. 各メソッドで各部を SearchClause に明示的に結合し、WHERE 句全体を構成します。

```
SearchClause sc1 = new SearchClause();
sc1.setLeftSearchQualification(aq1);
sc1.setRightSearchQualification(aq2);
sc1.setOperatorType(SearchClause.AND);
```

6. clone() メソッドを使用し、同じ左辺の SearchQualification を持つ sc1 に基づいて別の SearchClause を作成します。

```
SearchClause sc2 = sc1.clone();
sc2.setRightSearchQualification(aq3);
sc2.setOperatorType(SearchClause.AND);
```

7. コンストラクタを使用して、残りの各部を完全な SearchClause に結合します。

```
SearchClause sc3 = new SearchClause(jq1, jq2, SearchClause.AND);
SearchClause sc4 = new SearchClause(sc1, sc2, SearchClause.OR);
SearchClause scComplete = new SearchClause(sc3, sc4, SearchClause.AND);
```

SearchSortSpecification の構成

検索結果のソート順序を指定するには、SearchSortSpecification を構成します。`SearchSortSpecification` クラスには、結果の順序付けに使用する属性とそのクラスを指定するためのメソッドが用意されています。また、このクラスには、結果を ASCENDING 順と DESCENDING 順のどちらでリスト表示するかを指定するための、定数とメソッドも用意されています。

SearchSortSpecification を構成する最も簡単な方法は、次のコンストラクタを使用し、ソート条件セットを指定して初期化することです。

例 8-25 初期化時の SearchSortSpecification の構成

```
SearchSortSpecification ss = new SearchSortSpecification(
       new String[] {"CATEGORY", "FOLDER"},
       new String[] {"NAME", "NAME"},
```

```
new boolean[] {SearchSortSpecification.ASCENDING,
               SearchSortSpecification.ASCENDING});
```

また、SearchSortSpecification クラスには、より厳密な制御によりソート条件を構築して取り出すためのメソッドも用意されています。この種のメソッドは、アプリケーションで SearchSortSpecification のコンポーネントを再利用したり操作する場合に役立ちます。

例 8-26 Add メソッドを使用した SearchSortSpecification の構成

```
SearchSortSpecification ss = new SearchSortSpecification();
ss.add("CATEGORY", "NAME", SearchSortSpecification.ASCENDING);
ss.add("FOLDER", "NAME", SearchSortSpecification.ASCENDING);
```

Search の構成と実行

問合せ条件を SearchSpecification にアセンブルした後に、Search クラスで問合せを実行します。Search クラスには、バインド変数を渡して検索を実行するためのメソッドが用意されています。また、検索結果を横断するためのメソッドもあります。

例 8-27 Search の構成と実行

1. 問合せ条件を保持する AttributeSearchSpecification を構成します。

   ```
   AttributeSearchSpecification asp = new AttributeSearchSpecification();
   ```

2. SearchClassSpecification を構成します。

   ```
   SearchClassSpecification scp = new SearchClassSpecification("DOCUMENT');
   asp.setSearchClassnames(scp);
   ```

3. DOCUMENT.OWNER の AttributeQualification を構成します。

   ```
   AttributeQualification aq3 = new AttributeQualification();
   aq.setAttribute("DOCUMENT", "OWNER");
   aq.setOperatorType("=");
   aq.setValue("?");
   asp.setSearchQualification(aq);
   ```

4. Search オブジェクトを構成します。

   ```
   Search srch = new Search(session, asp);
   ```

5. バインド変数を保持する配列を構成します。

   ```
   AttributeValue[] bindValues = new AttributeValue[1];
   DirectoryUser dsUser = session.getDirectoryUser();
   bindValues[0] = AttributeValue.newAttributeValue(dsUser);
   ```

6. 検索でカーソルをオープンし、バインド変数を渡します。open() メソッドにより問合せが実行されます。

```
srch.open(bindValues);
```

7. 検索のカーソルをクローズします。

```
srch.close();
```

SearchResultObject のフェッチ

検索結果をフェッチするには、open() メソッドを使用して検索のカーソルをオープンし、open() メソッドを使用して各結果オブジェクトをフェッチします。next() メソッドは、単一の検索ヒットを表す SearchResultObject を戻します。その後、SearchResultObject の getLibraryObject() メソッドをコールして、検索条件と一致してヒットした情報の各部を取り出すことができます。

例 8-28 SearchResultObject のフェッチ

1. 検索でカーソルをオープンし、バインド変数を渡します。open() メソッドにより問合せが実行されます。

```
srch.open(bindValues);
```

2. 各検索結果のヒットを増分フェッチし、LibraryObject の名前を出力します。

```
int i;
SearchResultObject sro = srch.next();
LibraryObject lo;

while (sro != null)
{
  lo = sro.getLibraryObject();
  try
  {
     sro = srch.next();
  }
  catch (IfsException e)
  {
    if (e.getErrorCode() == 22000)
    {
       break;
    }
    else
    {
       e.printStackTrace();
    }
  }
```

```
}
```

3. 検索のカーソルをクローズします。

```
srch.close();
```

SearchObject としての SearchSpecification の保存

検索条件をリポジトリに永続的に保存するには、SearchSpecification から SearchObject を構成します。SearchObject を構成する手順は、次のとおりです。

1. SearchObjectDefinition を構成します。
2. SearchObjectDefinition の SearchSpecification を設定します。
3. SearchObjectDefinition を LibrarySession に渡して SearchObject を構成します。

例 8-29 SearchObject としての SearchSpecification の保存

1. アプリケーションですでに SearchSpecification を構成しているとします。

```
SearchSpecification ss = ....
```

2. SearchObjectDefinition を構成します。

```
SearchObjectDefinition sod = new SearchObjectDefinition(session);
sod.setAttributeByUpperCaseName("NAME",
   AttributeValue.newAttributeValue("XML Research"));

Collection aclColl = session.getSystemAccessControlListCollection();
AccessControlList acl = (AccessControlList) aclColl.getItems("Public");
sod.setAttributeByUpperCaseName("ACL"
   AttributeValue.newAttributeValue(acl));

Folder folder =
  session.getUser().getPrimaryUserProfile().getHomeFolder();
sod.setAddToFolderOption(folder);
```

3. SearchObjectDefinition の SearchSpecification を設定します。

```
sod.setSearchSpecification(ss);
```

4. SearchObjectDefinition を LibrarySession に渡して SearchObject を構成します。

```
SearchObject so = session.createPublicObject(sod);
```

サンプル・コード

Oracle 9*i*FS は、Java API の操作開始の出発点として役立つサンプル・コードとともにインストールされます。この API サンプル・コードは、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/devdoc/api` ディレクトリにあります。

表 8-19 に、この章に関連する API サンプル・コードを示します。

表 8-19 API サンプル・コード

クラス	使用方法
`SearchSample.java`	属性検索を作成して実行します。SearchObject を作成します。検索により生成された SQL を表示します。
`SelectorSample.java`	Selector 検索を作成して実行します。
`TextSearchSample.java`	内容ベースの検索を作成して実行します。
`ViewsSample.java`	検索からビューを作成します。

9

カスタム・パーサーの作成

この章では、Oracle 9*i*FS でのカスタム・パーサーの作成と使用について説明します。この章の内容は、次のとおりです。

- パーサーの概要
- パーサー・アプリケーションの記述
- サンプル・コード： カスタム・パーサー
- ClassSelectionParser の使用

パーサーの概要

パーサーは、情報の文字列を受け入れて構成要素に分割するプログラムです。Oracle 9*i*FS では、パーサーは、リポジトリへの挿入中にファイルから構造化されたデータ要素を抽出する Java クラスです。ドキュメントの場合、パーサーは次の操作を実行します。

- ファイルの内容を InputStream または Reader オブジェクトとして取り込みます。
- 文字入力を処理して属性を抽出します。
- 抽出した属性やファイルの内容に基づいて、Document や Folder など、1 つ以上のデータベース・オブジェクトを生成します。

パーサーがデータ・ストリームをまったく操作しない場合があり、その場合は Oracle 9*i*FS に挿入されるドキュメントのプリプロセッサとみなすことができます。この種の例に、「ClassSelectionParser の使用」で説明する ClassSelectionParser があります。

ほとんどの場合、構造化データを Oracle 9*i*FS に挿入する最も便利な方法は、XML ファイルと組込みの XML 解析フレームワークを使用して、ドキュメントからデータ要素を抽出することです。この方法については、第 10 章「XML および Oracle Internet File System」を参照してください。

カスタム・パーサーの使用

`.doc` または `.xls` など、非 XML 文書を解析する場合や、カスタム・タイプを定義している場合は、このようなドキュメントからデータベース・オブジェクトを作成するカスタム・パーサーを記述する必要があります。カスタム・パーサーを作成するには、既存の Oracle 9*i*FS パーサーをサブクラス化する方法と、`oracle.ifs.beans.parsers.Parser` インタフェースを実装するカスタム・クラスを作成する方法があります。

Parser クラスでは、1 つ以上のオブジェクトが作成されます。ほとんどの場合、Parser クラスは次のオブジェクトの作成に使用されます。

- Document
- Folder
- Document と Folder の組合せ

パーサーでは、作成されるオブジェクトのタイプは、渡された InputStream または Reader オブジェクトに基づいて決定されます。InputStream または Reader で複数タイプのオブジェクトが記述されている場合、パーサーは次のどちらかを実行できます。

- 完了直後に各オブジェクトを作成します。
- ストリームの終わりに達した時点ですべてのオブジェクトを作成します。

パーサー・アプリケーションの概要

パーサー・アプリケーションには、次の 4 つのコンポーネントが含まれています。

- パーサーをコールするアプリケーション（パーサー・アプリケーション）
- パーサー自体
- ParserCallback（オプション）
- パーサー登録メカニズム ParserLookupByFileExtension PropertyBundle

表 9-1 に、各コンポーネントを示します。

表 9-1 パーサーのコンポーネント

コンポーネント	説明 / サンプル
Application	アプリケーションは、必要なパーサーのインスタンスを作成し、ドキュメント表現（必須）、ParserCallback オブジェクト名（オプション）および Options オブジェクト（オプション）を指定してパーサーをコールします。
Parser	パーサーは、解析済みオブジェクトの作成に必要なカスタム・コードを実行し、解析済みオブジェクトをリポジトリに格納します。
ParserCallback	アプリケーションは、オプションで ParserCallback オブジェクトを指定できます。ParserCallback オブジェクトの preOperation() または postOperation() メソッドは、解析操作の前、後または前後に実行される処理を追加指定します。
ParserLookupBy FileExtension PropertyBundle	Oracle 9*i*FS は、このドキュメント・クラスのパーサーの名前を ParserLookupByFileExtension PropertyBundle 内で検索します。

パーサー・アプリケーションの記述

パーサー・アプリケーションを記述するタスクは、次のとおりです。

1. パーサー・クラスの記述
2. パーサーの配置
3. パーサーのコール（パーサー・アプリケーション内）
4. ParserCallback の記述（オプション）

パーサー・クラスの記述

パーサーの目的は、ファイル内のデータ要素を識別し、各要素を使用してデータベース・オブジェクトに属性を移入することです。

カスタム・パーサーの作成時に、次の 2 つのアプローチから選択できます。

- 既存の Oracle 9*i*FS パーサーの 1 つがアプリケーションの要件と部分的に一致している場合は、既存のパーサーをサブクラス化できます。（IfsSimpleXmlParser は、`oracle.ifs.beans.parsers.Parser` インタフェースを実装します。）
- 既存の Oracle 9*i*FS パーサーを始点として使用できない場合は、インタフェース `oracle.ifs.beans.parsers.Parser` を実装するカスタム・クラスを作成する必要があります。

どちらのアプローチを選択した場合も、カスタム・パーサーを記述するには、Parser インタフェースを直接または間接的に実装することになります。Parser インタフェースには、次の 2 種類の入力を受け入れるようにオーバーロードされたメソッド `parse()` が含まれています。

- InputStream オブジェクト
- Reader オブジェクト

`parse()` メソッドがコールされると、パーサー自体のコードのバランスによって各行が検査され、その内容がパーサーにより作成されるオブジェクトの該当する属性に置かれます。`parse()` の構文と引数については後述します。

カスタム・パーサーを記述するには、次の 2 つのメソッドを記述する必要があります。

- コンストラクタ
- parse() メソッド

コンストラクタの記述

各パーサーは、パーサー用の標準コンストラクタを実装する必要があります。標準コンストラクタは、表 9-2 のようにパラメータを 1 つ取ります。

表 9-2 コンストラクタのパラメータ

パラメータ	データ型	説明
`session`	LibrarySession	現行のユーザーの LibrarySession

例 9-1 コンストラクタ

```
public SimplestParser(LibrarySession session) throws IfsException
```

parse() メソッドの記述

表 9-3 に、parse() メソッドのパラメータを示します。

表 9-3 parse() メソッドのパラメータ

パラメータ	データ型	説明
`stream`	InputStream	パーサーが読み込む InputStream。InputStream は、オーディオ・データやビデオ・データなど、文字ベースでないデータに使用します。
`reader`	Reader	パーサーが読み込む Reader。文字ベースのデータには、Reader を使用する必要があります。
`callback`	ParserCallback	オプション・パラメータ。NULL でもかまいません。値を指定すると、ParserCallback オブジェクトには解析前、解析後または解析前後に実装する処理を指定するメソッドが組み込まれます。
`options`	Hashtable	オプション・パラメータ。NULL でもかまいません。値を指定すると、Options パラメータにより、オプションの名前 / 値のペアのセットを通じてパーサーの動作がさらに厳密に制御されます。通常は、文字コードの指定に使用します。

parse() メソッドのサンプル・コードは、この章の「サンプル・コード： カスタム・パーサー」を参照してください。

パーサーの配置

プロトコル・サーバーと他の標準 Oracle 9*i*FS コンポーネントがカスタム・パーサーにアクセスできるように、パーサーのクラスを含むフォルダ・ツリーを Oracle 9*i*FS の CLASSPATH に指定する必要があります。Oracle 9*i*FS には、そのための特別なディレクトリがあります。このディレクトリは `custom_classes` で、Oracle 9*i*FS サーバー・ソフトウェアが使用する CLASSPATH 環境変数に定義されています。

パーサーを配置する手順は、次のとおりです。

1. パーサーをコンパイルして `.class` ファイルを作成します。
2. 生成された `.class` ファイルを含むフォルダ・ツリーを、Oracle 9*i*FS がインストールされているサーバーのディレクトリ `$ORACLE_HOME/ifs/custom_classes` に置きます。

注意： コンパイル済みの Java コードは、Oracle 9*i*FS リポジトリではなくサーバー固有のファイル・システムにコピーする必要があります。

パーサーの登録

パーサーを登録する目的は、特定のパーサーに特定のファイル拡張子をマップすることです。このマッピングの作成後は、指定の拡張子を持つファイルが Oracle 9*i*FS クライアントまたはプロトコルによりアップロードされると、そのファイルはリポジトリに格納される前にカスタム・パーサーに渡されます。パーサーを登録するには、次の 2 つの方法があります。

機能	メリット / デメリット
Oracle 9*i*FS Manager	操作が簡単で使用しやすい方がよい場合は、Oracle 9*i*FS Manager を使用します。Oracle 9*i*FS Manager を使用して登録できるのは、Oracle 9*i*FS Manager 機能と同じ Oracle 9*i*FS インスタンスに存在するパーサーのみです。
XML	XML を使用するのは、スクリプトを使用してパーサーを登録する場合や、パーサーを別の Oracle 9*i*FS インスタンスに配置する必要がある場合です。

登録済みの各パーサーは、次の 2 つの属性を持ちます。

属性	データ型	説明	例
Extension	STRING	ファイル拡張子。	cus
ClassName	STRING	パーサーの完全修飾クラス名。	ifs.demo.simplestparser.parser.SimplestParser

ファイル拡張子とパーサー間のマッピングを格納するための基礎となるメカニズムは、「ParserLookupByFileExtension」と呼ばれる PropertyBundle オブジェクトです。PropertyBundle は、Property オブジェクトの配列として格納される名前 / 値のペアのリストです。この PropertyBundle 内の各 Property オブジェクトには、ファイル拡張子とパーサー間のマッピングが Name/Value のペアとして格納されます。

- Property の Name 属性には、cus など、Extension 属性の値が格納されます。
- Property の Value 属性には、ifs.demo.simplestparser.parser.SimplestParser など、パーサーの ClassName が格納されます。

Oracle 9*i*FS Manager を使用したパーサーの登録

Oracle 9*i*FS Manager を使用してパーサーを登録する手順は、次のとおりです。

1. Oracle 9*i*FS Manager の「Object」メニューから「Register」を選択します。
2. 「Select Object Type」ウィンドウで「Parser Lookup」を選択します。
3. 「Parser Lookup Registry」ウィンドウで「Add」を選択します。
4. 「Parser Lookup Entry」ウィンドウで、属性のテキスト・ボックスに値を入力します。
5. 「OK」をクリックします。

XML を使用したパーサーの登録

XML を使用してパーサーを登録するには、パーサーのファイル拡張子とクラス名を指定して、ParserLookupByFileExtension PropertyBundle に新規 Property オブジェクトを追加する XML ファイルを記述します。

```
<?xml version="1.0" standalone="yes"?>
<!--SimplestParser.xml-->
<PROPERTYBUNDLE>
   <UPDATE RefType="valuedefault">ParserLookupByFileExtension</UPDATE>
   <PROPERTIES>
      <PROPERTY ACTION="add">
         <NAME>cus</NAME>
```

```
          <VALUE DataType="String">
            oracle.ifs.examples.devdoc.parser.SimplestParser
          </VALUE>
        </PROPERTY>
    </PROPERTIES>
</PROPERTYBUNDLE>
```

パーサーのコール

SimplestParser の例（「XML を使用したパーサーの登録」）では、適切な拡張子（.cus）が付いたドキュメントが Oracle 9*i*FS に挿入されると、プロトコル・サーバーは自動的にパーサーをコールします。

アプリケーション・プログラムが内容をリポジトリに挿入するときに、アプリケーションは適切なパーサー、つまり標準 Oracle 9*i*FS パーサーまたはカスタム・パーサーをコールする必要があります。

ドキュメントを解析するには、アプリケーションで次の操作が必要です。

- 適切なパーサーのインスタンス化
- データ・ストリームに対する parse() メソッドのコール

ParserCallback の記述

カスタム・アプリケーションでは、パーサーをコールするときに、オプションで ParserCallback オブジェクトを渡すことができます。ParserCallback により、アプリケーションはデータ・ストリームが処理される前または後に処理を追加できます。

ParserCallback インタフェースは、アプリケーションがパーサーと相互に作用できるように、次の 3 つのメソッドを指定します。

- preOperation() メソッド
- postOperation() メソッド
- signalException() メソッド

preOperation() メソッド

アプリケーションでは、preOperation() を使用して、パーサーでリポジトリの更新に使用される前に LibraryObjectDefinition を次のように変更できます。

- 既存の LibraryObjectDefinition を使用するには、Definition をそのまま戻します。
- 解析対象のオブジェクトを変更するには、異なる Definition を構成するか、戻す前の Definition を変更します。
- パーサーによるリポジトリの更新を禁止するには、Definition 値の NULL を戻します。

パラメータ名	データ型	説明
lo	LibraryObject	解析操作により更新されるオブジェクト。操作により新規オブジェクトが作成される場合、値は NULL です。
def	LibraryObjectDefinition	オブジェクト lo の更新に使用される LibraryObjectDefinition。

例 9-2 preOperation()

```
public LibraryObjectDefinition preOperation (LibraryObject lo,
                                             LibraryObjectDefinition def)
             throws IfsException
```

postOperation() メソッド

アプリケーションは postOperation() を使用して、パーサーにより作成または更新されたリポジトリ・オブジェクトにアクセスします。

パラメータ名	データ型	説明
lo	LibraryObject	解析操作により作成または更新された LibraryObject。

例 9-3 postOperation()

```
public void postOperation (LibraryObject lo)
             throws IfsException
```

signalException() メソッド

アプリケーションは signalException() を実装して、解析中に発生する例外をインターセプトできます。次の 2 つのオプションがあります。

- 例外を発生させます。この場合、解析は終了します。
- 単に値を戻します。この場合、解析は続行されます。

パラメータ名	データ型	説明
e	IfsException	潜在的な例外。

例 9-4 signalException()

```
public void signalException(IfsException e)
            throws IfsException
```

例 9-5 ParserCallback の実装

例 9-5 に、ParserCallback インタフェースの簡単な実装例を示します。

- preOperation() メソッドを使用して、オブジェクトの解析前にフォルダを検索します。
- postOperation() メソッドを使用して、解析済みのオブジェクトをフォルダに追加します。

```
/*---FolderParsedObject.java---*/
package oracle.ifs.examples.devdoc.parser;

import oracle.ifs.beans.Folder;
import oracle.ifs.beans.LibraryObject;
import oracle.ifs.beans.LibraryObjectDefinition;
import oracle.ifs.beans.PublicObject;
import oracle.ifs.beans.parsers.ParserCallback;

import oracle.ifs.common.IfsException;

public class FolderParsedObject implements ParserCallback {
    private Folder m_TargetFolder;

    public FolderParsedObject(Folder f) {
        m_TargetFolder = f;
    }

    public oracle.ifs.beans.LibraryObjectDefinition preOperation(LibraryObject
parm1, LibraryObjectDefinition parm2) throws oracle.ifs.common.IfsException
    {
        return parm2;
    }

    public void postOperation(LibraryObject newObject) throws
oracle.ifs.common.IfsException
    {
        m_TargetFolder.addItem((PublicObject) newObject);
    }

    public void signalException(IfsException e) throws IfsException
    {
        throw e;
    }

}
```

サンプル・コード： カスタム・パーサー

この SimplestParser は、HTML ドキュメントの <TITLE> タグで囲まれたテキストを抽出し、その情報をカスタム・フィールドに格納します。そのためには、ファイル拡張子 .cus を指定して、属性 TITLE を持つ CUSTOM という、Document のサブクラスをサーバーに登録する必要があります。この例ではカスタム・ファイル拡張子を使用していますが、これは必須ではありません。このパーサーの登録時にファイル拡張子 .htm および .html を指定し、パーサーにすべての HTML ドキュメントを処理させることもできます。

この例は簡素化されており、バージョニングされるドキュメントは考慮されていません。また、ローカル・キャラクタ・セットに関する問題も考慮されていません。

```
package oracle.ifs.examples.devdoc.parser;

// These classes provide the building blocks for a 9iFS document.

import oracle.ifs.beans.Attribute;
import oracle.ifs.beans.Document;
import oracle.ifs.beans.DocumentDefinition;
import oracle.ifs.beans.Format;
import oracle.ifs.beans.LibraryObject;
import oracle.ifs.common.Collection;
import oracle.ifs.common.AttributeValue;

// These classes are used to instantiate a folder object to store the document.

import oracle.ifs.beans.Folder;
import oracle.ifs.beans.FolderPathResolver;

// These classes are used to obtain information about the user at runtime.

import oracle.ifs.beans.DirectoryUser;
import oracle.ifs.beans.PrimaryUserProfile;
import oracle.ifs.beans.LibrarySession;

// These classes are the base classes for creating a parser.

import oracle.ifs.beans.parsers.Parser;
import oracle.ifs.beans.parsers.ParserCallback;
import java.util.Hashtable;

// These are standard Java objects used to process the document content.

import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import java.io.BufferedReader;
```

```
// This class is used to report exceptions in iFS methods.

import oracle.ifs.common.IfsException;

public class SimplestParser implements Parser
{
  private String title;
  private LibrarySession m_librarySession;
  private Document newDoc;
  private Folder currentFolder;
  private Folder homeFolder;

// The constructor argument captures the current library session, which is
// used to pass information about the user and environment at runtime.

  public SimplestParser(LibrarySession session) throws IfsException
  {
    m_librarySession = session;
  }

/* This parser is called by the host protocol at runtime, passing a Reader
 * object with the contents of the document being parsed. The callback is
 * an optional argument that enables the parser to respond to the calling
 * method. The Hashtable is used to store three key parameters:
 * CURRENT_PATH_OPTION: the current working directory.
 * CURRENT_NAME_OPTION: the name of the file being parsed.
 * UPDATE_OBJECT_OPTION: indicates if the document being parsed is
 *                       replacing an object that already exists.
 */
  public LibraryObject parse(Reader htmlStream, ParserCallback callback,
        Hashtable options) throws IfsException
  {
  try
  {

/*  Instantiate a FolderPathResolver, then pass it the CURRENT_PATH_OPTION as
 *  a string. Set the currentFolder variable to the path where the document
 *  to be parsed was inserted.
 */

    FolderPathResolver fpr = new FolderPathResolver(m_librarySession);
    fpr.setRelativePath(options.get(Parser.CURRENT_PATH_OPTION).toString());
    currentFolder = fpr.getCurrentDirectory();

/*  Instantiate the string variable documentContent.
 *  Instantiate a BufferedReader object named dataStream and populate it
```

```
 *  with the document content passed in to the method.
 */

    String documentContent = "";
    BufferedReader dataStream =
                          new BufferedReader(htmlStream);

// Read the buffered data into the documentContent variable one line at a time.

    for (String line = dataStream.readLine();line != null;
                          line = dataStream.readLine())
    {
      documentContent = documentContent + line + "¥n";
    }

//  Send the resulting string to the parseTitle method to extract the title.

    String docTitle = parseTitle(documentContent);

//  Instantiate a DocumentDefinition object.

    DocumentDefinition docDef = new DocumentDefinition(m_librarySession);

//  Instantiate a Collection object and populate it with the list of
//  format extensions. Set the format in the document definition.

    Collection allFormats = m_librarySession.getFormatExtensionCollection();
    docDef.setFormat((Format) allFormats.getItems("cus"));

//  The Classname is the name of the subclass we've defined (CUSTOM).

    docDef.setClassname("Custom");

//  Set the Name attribute in the document definition to the variable passed
//  to the parser in the options Hashtable.

    docDef.setAttribute("NAME", AttributeValue.newAttributeValue
                    (options.get(Parser.CURRENT_NAME_OPTION)));

//  Set the custom attribute "TITLE" to the docTitle variable returned by the
//  parseTitle method.

    docDef.setAttribute("TITLE", AttributeValue.newAttributeValue(docTitle));

//  Set the content of the document to the String documentContent.

    docDef.setContent(documentContent);
```

```
/*  Check to see if the UPDATE_OBJECT_OPTION variable is set. If so, update
 *  the document (update). If not, create a new document (addItem).
*/
    if(options.get(UPDATE_OBJECT_OPTION) != null)
    {
      Document currentDoc = (Document) currentFolder.findPublicObjectByPath
                      (docDef.getAttribute("NAME").toString());
      currentDoc.update(docDef);
    }
    else
    {

      //  Instantiate a new Document using the DocumentDefinition just defined.

      Document newDoc = (Document) m_librarySession.createPublicObject(docDef);
      currentFolder.addItem(newDoc);
    }
  }

// Catch any exceptions. Set VerboseMessage to true to get a more complete
// report of the methods that threw the exception.

  catch (IfsException ifsExceptionCaught)
  {
    ifsExceptionCaught.setVerboseMessage(true);
    ifsExceptionCaught.printStackTrace();
  }
  catch (Exception exceptionCaught)
  {
    exceptionCaught.printStackTrace();
  }
  return newDoc;
  }

/* parse method called when the protocol sends the file content as an
 * InputStream. This method converts the InputStream to a BufferedReader
 * and forwards it to the first parse method (keeps code concise).
*/

  public LibraryObject parse(InputStream htmlStream, ParserCallback callback,
                             Hashtable options)
  {

// Convert the InputStream htmlStream to the BufferedReader named redirect.

    BufferedReader redirect =
```

```
        new BufferedReader(new InputStreamReader(htmlStream));

// Send the resulting BufferedReader to the first parse method.

    try {
      Document newDoc = (Document) parse(redirect,callback,options);
    }

// Catch and report (in verbose mode) any exceptions.

    catch (IfsException ifsExceptionCaught)
    {
      ifsExceptionCaught.setVerboseMessage(true);
      ifsExceptionCaught.printStackTrace();
    }
    catch (Exception exceptionCaught)
    {
      exceptionCaught.printStackTrace();
    }
    return newDoc;
  }

/*  This is the actual custom parsing routine. It searches the text String
 *  for the tag <TITLE>, starts at the 7th character (the length of the
 *  <TITLE> tag and extracts a substring of all the information through
 *  the last character before the </TITLE> tag.
 */

  private String parseTitle (String parseString){
    try
    {
      title = parseString.substring((parseString.indexOf("<TITLE>")+ 7),
                  parseString.indexOf("</TITLE>"));
    }
    catch (Exception e)
    {
      title = "Untitled";
      e.printStackTrace();
    }
    return title;
  }
}
```

ClassSelectionParser の使用

ClassSelectionParser は、PropertyBundle を使用して、ドキュメントが Oracle 9*i*FS に挿入される時点で使用する適切な ClassObject を識別します。このパーサーは固有のもので、ドキュメントの内容を処理するのではなく、そのメタデータ（名前）を処理します。ClassSelectionParser を使用するには、次のタスクを実行する必要があります。

1. クラス定義の作成
2. パーサーの登録
3. クラスの登録

クラス定義の作成

ClassSelectionParser での最初のステップは、カスタムのクラス定義を作成することです。この例では、XML 構成ファイルを使用してプレゼンテーション・スライド用のカスタム・クラスを定義し、新規タイプ Presentation のすべてのファイルに追加する新規属性 NumberOfSlides を記述しています。

```
<?xml version = '1.0' standalone = 'yes'?>
<!--Presentation.xml-->
<ClassObject>
  <Name>Presentation</Name>
   <Superclass RefType='name'> Document </Superclass>
  <Description>Custom Class for Presentations</Description>
  <Attributes>
    <Attribute>
      <Name>NumberOfSlides</Name>
      <DataType>INTEGER</DataType>
    </Attribute>
  </Attributes>
</ClassObject>
```

パーサーの登録

カスタム ClassObject の作成後に、ParserLookupByFileExtension の ValueDefault PropertyBundle 内でファイル拡張子（この例では `.ppt`）を `ClassSelectionParser` に対応付けます。パーサーを登録するには、Oracle 9*i*FS Manager または XML を使用できます。

```
<?xml version = '1.0' standalone = 'yes'?>
<!--RegisterPPTParser.xml-->
<PropertyBundle>
  <update reftype='valuedefault'>  ParserLookupByFileExtension </update>
  <Properties>
    <Property action = 'add'>
      <Name> ppt </Name>
```

```
      <Value datatype='String'>
          oracle.ifs.beans.parsers.ClassSelectionParser
      </Value>
    </Property>
  </Properties>
</PropertyBundle>
```

クラスの登録

パーサーの登録後に、PropertyBundle である IFS.PARSER.ObjectTypeLookupByFileExtension にエントリを追加して、カスタム・クラスを登録する必要があります。この場合、登録プロセスでは、ファイル拡張子（ppt）をカスタム・クラス（Presentation）に対応付けています。

クラスを登録すると、ClassSelectionParser のコールに必要なプロセスが完了したことになります。このステップを省略すると、ClassSelectionParser に対応付けられたクラスはデフォルトで Document に設定され、解析は発生しません。

```
<?xml version = '1.0' standalone = 'yes'?>
<!--RegisterPPTObjectType.xml-->
<PropertyBundle>
  <update reftype='valuedefault'>  IFS.PARSER.ObjectTypeLookupByFileExtension
</update>
  <Properties>
    <Property action = 'add'>
      <Name> ppt </Name>
      <Value datatype='String'>Presentation</Value>
    </Property>
  </Properties>
</PropertyBundle>
```

ファイル拡張子と対応するファイル・タイプの登録後は、定義した ClassObject のインスタンスの後に、その拡張子を持つドキュメントが作成されます。

10

XML および Oracle Internet File System

XML は、データ転送方法およびシステム構成メカニズムとして、Oracle 9*i*FS で広範囲に使用されます。この章では、XML データを処理できるように Oracle 9*i*FS に組み込まれている特定のサポートについて説明します。

この章の内容は、次のとおりです。

- XML の概要
- Oracle 9iFS での XML データ・ファイルの使用
- XML ファイルを使用した Oracle 9iFS の構成

XML の概要

XML（eXtensible Markup Language）は、情報システム間で標準的なデータ転送方法を提供する情報記述仕様であり、2 つのシステムのデータ・スキーマが正確に一致している必要はありません。XML ファイルは自己定義型です。新規データ・フィールドは、任意に名前を付けた 1 組のタグでデータ要素を囲んで定義できます。たとえば、データ要素 MyXMLDataElementTag を持つ MyXMLRootTag 型の XML 文書を作成する場合、完全に正当な XML 構文は次のようになります。

```
<?xml version="1.0" standalone="yes"?>
<MyXmlRootTag>
<MyXmlDataElementTag>bar</MyXmlDataElementTag>
</MyXmlRootTag>
```

前後の空白は省略されるため、次のデータ要素は、

```
<MyXmlDataElement>   data     </MyXmlDataElement>
```

次のデータ要素と同じです。

```
<MyXmlDataElement>data</MyXmlDataElement>
```

タグには大 / 小文字区別がないため、タグ <MYXMLDATAELEMENT> および <MyXMLDataElement> は同じです。ただし、各要素の開始タグと終了タグは正確に一致する必要があります。たとえば、次の構文を使用するとします。

```
<MyXmlDataElement>data</MYXMLDATAELEMENT>
```

この場合、Oracle 9*i*FS では例外が発生します。

Oracle 9*i*FS では、Oracle9*i* データベースの基礎となる XML 処理機能と、XML ファイルを使用して様々なタスクを実行するための追加機能が公開されています。

XML は新たな標準です。Oracle 9*i*FS で使用されている実装は、変化がないと思われる中心的な想定に基づいて基本レベルで保たれています。

インターネット・ファイル・システムにおける XML の様々な役割の概要

Oracle 9*i*FS では、後述のように、XML ファイルに少なくとも 5 つの使用方法があります。ここで説明する永続的な XML 文書とは、Oracle 9*i*FS のフォルダ階層に XML 文書として表示されるものを指します。一時的な XML 文書は、トランザクションを実行するためにアップロードされるドキュメントです。そのトランザクションの結果、別のタイプの永続的なドキュメントが作成される場合や、Oracle 9*i*FS の構成が変更される場合がありますが、XML 文書自体が Oracle 9*i*FS にドキュメントとして格納されることはありません。

Oracle 9*i*FS に格納される永続的な XML 文書

最も基本的なレベルでは、XML 文書は他のタイプのファイルと同様に Oracle 9*i*FS に格納できます。ファイルをアップロードし、XML エディタで作成または変更し、あらゆる XML アプリケーションで使用できます。

Oracle 9*i*FS に格納される解析済み属性を持つ永続的な XML 文書

永続的な XML 文書のデータ要素を解析し、情報の一部またはすべてをドキュメントのカスタム属性として格納するように選択できます。解析済みの要素は、ドキュメントの「プロパティ」ダイアログ・ボックスに（Windows と Web インタフェースの両方で）表示され、そのカスタム属性を検索できます。Oracle 9*i*FS で情報を表示、変更またはレンダリングする方法を決定するときには、複数のオプションから選択できます。

カスタム・タイプの永続的ドキュメントを作成するための一時的な着信データ・ドキュメント

一部のカスタム・アプリケーションでは、XML ファイルではなくカスタム・ドキュメント・タイプを作成する必要があります。XML ファイルを使用すると、XML タグでファイルの属性を定義して、ドキュメント・インスタンスを作成する必要のあるドキュメント・サブクラスを識別できます。ファイルを Oracle 9*i*FS にアップロードすると、ファイルが解析され、選択したサブクラスのドキュメントが作成されます。ドキュメントの作成に使用された XML ファイルは Oracle 9*i*FS には表示されず、ドキュメントのランタイム・インスタンスはファイルの作成後に破棄されます。

発信 XML データ・ファイルとして表示される永続的なドキュメント

Oracle 9*i*FS に格納されているドキュメントは、作成方法に関係なく XML 文書としてレンダリングできます。レンダリングされる情報量を制御するには、組込みの構成設定を使用する方法と、カスタム・レンダラを作成する方法があります。レンダリングされた XML 文書は、ディスクや別のファイル・サーバーに転送したり、別のデータ・ストアへの入力として使用できます。

Oracle 9*i*FS 構成用の一時的な着信データ転送メカニズム

XML ファイルを使用して、新規ドキュメント・タイプ、プロパティのまとまり、ACL または他の Oracle 9*i*FS オブジェクトの作成など、様々な管理タスクを実行できます。これらのファイルは、Oracle 9*i*FS スキーマの変更や、Oracle 9*i*FS で使用する永続的メタデータの作成または変更に使用されます。このファイルのランタイム・インスタンスは変更の完了時に破棄されますが、変更内容は Oracle 9*i*FS に永続的な値として格納されます。

Oracle 9*i*FS での XML データ・ファイルの使用

XML をデータ転送メカニズムとして操作する場合は、Oracle 9*i*FS をほとんどカスタマイズせずに、複数のオプションを使用できます。

- 永続的な XML 文書の格納
- 解析済みデータ要素を持つ永続的な XML 文書の格納
- ラウンドトリップ XML 文書の格納

永続的な XML 文書の格納

XML 文書は、他のタイプのファイルと同様に Oracle 9*i*FS に格納できます。必要なプロトコル経由で XML ファイルをアップロードし、XML エディタで変更し、あらゆる XML アプリケーションで使用できます。

Oracle 9*i*FS オブジェクト用に予約済みルート・タグの 1 つと一致するタグが、XML 文書のルート・タグとして使用されていない場合は、XML 文書を格納するためのカスタマイズは不要です。XML 文書に予約済みのルート・タグが使用されている場合は、データ要素タグが Oracle 9*i*FS で使用されるタグと競合しないように、XML の名前空間を使用する必要があります。予約済みのルート・タグは、Oracle 9*i*FS クラス階層内のアイテムの名前（Document、Folder、PublicObject など）と、2 つの特殊タグ SimpleUser および ObjectList、作成したカスタム・サブクラスと一致するタグなどです。名前空間の詳細は、「XML 名前空間の概要」を参照してください。

解析が使用禁止になっている場合は（ルート・タグが Oracle 9*i*FS の要素と一致していても）、リテラルの XML 文書も格納できます。CUP、FTP および Web インタフェースなどのプロトコルでは、解析を使用禁止にできます。

例 10-1 リテラルの XML 文書

次の内容を含むドキュメント example.xml は、解析されずに XML 文書として格納されます。

```
<?xml version="1.0" standalone="yes"?>
<MyUnrecognizedTag>
<Name>A Name</Name>
<Description>A Description</Description>
</MyUnrecognizedTag>
```

Name および Description は多数の Oracle 9*i*FS オブジェクトに共通のタグですが、ルート・タグが認識されないため、この XML 文書は解析されません。

DTD の妥当性チェック

Oracle 9*i*FS のパーサー・フレームワークでは、Document Type Definition（DTD）の妥当性チェックを実行できます。DTD では、XML 文書の構造が記述され、ドキュメントが従う必要のある検証規則が指定されます。必須要素の欠落など、これらの規則で予期される構造に従っていないドキュメントは無効です。DTD 情報は、ドキュメント自体に埋め込む方法と外部ファイルに格納する方法があります。

Oracle DOM パーサーは、ファイルを解析し、その内容を DTD 検証規則と比較して、ファイルを有効または無効として宣言できます。デフォルトでは、パーサー・フレームワークでは XML ファイルの妥当性チェックは行われません。妥当性チェックを行うという決定はシステム全体に適用されるため、DTD の妥当性チェックを有効化する場合は、Oracle 9*i*FS Manager を通じてシステム・プロパティを設定する必要があります。

DTD 妥当性チェックを有効化する手順は、次のとおりです。

1. 「Object」メニューから「Register」を選択します。
2. 「Select Object Type」ダイアログ・ボックスで「Parser Lookup」を選択し、「Register」をクリックします。
3. 特定のパーサー・クラス名に対応付けるドキュメント拡張子を追加します。
4. 表示される「Parser Lookup Registry」ウィンドウで、「DTD validation by default」オプションをオンにします（オフになっている場合）。
5. 「OK」をクリックしてダイアログ・ボックスを閉じ、このシステム単位の変更を適用します。

解析中に、Oracle 9*i*FS により（`Parse()` メソッドにオプションとして渡された）「Validate」オプションがチェックされます。オプションが渡されていない場合、Oracle 9*i*FS はシステム・プロパティをチェックし、それに応じてオプションを設定します。これは、解析前フェーズで発生します。後の解析時には、IfsSimpleXmlParser は妥当性チェック・オプションの値を完全に無視します。ただし、LiteralDocumentParser は「Validate」オプションを調べて、ファイルの妥当性をチェックするかどうかを決定します。カスタム・パーサーでは「Validate」オプションを使用できます。その値をチェックして `IfsXmlParser.createDOM()` メソッドに渡すか、開発者が選択した方法で処理できます。

XML パーサーでファイルが無効であることが検出されると、Oracle 9*i*FS でエラーが生成されます。このエラーは、パーサー・コールバック・メカニズムや、Oracle 9*i*FS アプリケーションの他の部分でトラップできます。また、ファイルのアップロードに使用したプロトコル・サーバーにも、エラー・メッセージが表示されます。

ファイルが有効でない場合は、Oracle 9*i*FS には格納されません。ユーザーは、そのファイルのエラーを訂正して再度アップロードする必要があります。

解析済みデータ要素を持つ永続的な XML 文書の格納

Oracle 9*i*FS はファイル・システムですが、その中心は Oracle9*i* データベースです。Oracle 9*i*FS 固有の機能は、ファイルをファイル・インスタンスおよびリレーショナル・データの両方として処理できることです。オプションで、XML 文書自体は変更せずに、そのドキュメントから選択したデータ要素を抽出するカスタム・パーサーを作成できます。ドキュメントは XML アプリケーションでも表示して編集できますが、その解析済み要素に基づいて情報を検索することもできます。

この機能を活用するには、Document タグ、または Document クラスのカスタム・サブクラスを表すタグを使用して、XML 文書を作成します。認識されたタグはファイルの属性として格納され、認識されないタグで囲まれたデータは無視されます。

例 10-2 解析済みデータ要素を持つ XML 文書

次の XML 文書は、Oracle 9*i*FS への挿入時にデータ要素がファイルの属性として解析されています。

```
<?xml version="1.0" standalone="yes"?>
<Document>
  <Name>A Name</Name>
  <Description>A Description</Description>
</Document>
```

ただし、このドキュメントの内容は、次のように XML ファイルの内容になるため注意してください。これらの属性を検索し、ドキュメントを XML として表示できます。テキストの内容を持つドキュメント（HTML ファイルなど）を作成するには、Content 属性タグを使用します。

例 10-3 Content 属性を持つ XML 文書

次の XML 文書の場合は、Oracle 9*i*FS への挿入時にデータ要素がファイルの属性として解析され、Content タグで囲まれた情報がドキュメントの内容として格納されます。

```
<?xml version="1.0" standalone="yes"?>
<Document>
 <Name>A Name</Name>
 <Description>A Description</Description>
 <Content>Here is the actual content of the document.</Content>
</Document>
```

このドキュメントをアップロードすると、このファイルを Web インタフェースで開いている場合は、「Here is the actual content of the document.」と表示されます。

ラウンドトリップ XML 文書の格納

ラウンドトリップ XML は Oracle 9*i*FS の新機能で、XML ファイルを永続的なドキュメントとして格納してから、元どおり正確に表示できます。これは、コメント・タグを使用して情報を格納したり、未定義のタグを使用して Oracle 9*i*FS で解析されない情報を格納する、一部の顧客に固有の要件を満たすための機能です。

デフォルトでは、Document またはそのサブクラスを処理する XML ファイルを Oracle 9*i*FS にアップロードすると、ドキュメント自体がファイルの Content 属性に格納されます。ドキュメントを表示すると、オリジナルの XML ファイルは、すべてのコメント・タグと Oracle 9*i*FS で未定義のタグを含めて挿入時のままの状態で出力されます。

ラウンドトリップ XML を使用する場合は、次の 2 つの考慮事項が重要になります。

1. ドキュメントは Oracle 9*i*FS から挿入時のままの状態で表示されます。ファイルの変更内容が Oracle 9*i*FS に格納されていても、ドキュメントの属性と Content 属性内の情報の同期状態を保つカスタム・インタフェースを記述しないと、表示されたファイルには反映されません。
2. ドキュメント全体が、Oracle 9*i*FS の標準 Content 属性に格納されます。つまり、ラウンドトリップ・ドキュメントでは、独自の Content タグを定義できません。Content タグを定義すると、そのタグで囲まれたテキストはドキュメントの内容として格納されます。

XML を使用したカスタム・サブクラスのドキュメントのインスタンス作成

XML を使用して組込みオブジェクトをインスタンス化するのと同じ方法で、カスタム・サブクラスのインスタンスを作成できます。

カスタム・サブクラスのドキュメントをインスタンス化する XML 文書のインスタンスを作成する手順は、次のとおりです。

1. ルート要素としてカスタム・サブクラスを使用し、XML ファイルを作成します。
2. すべての必須属性のエントリと、必要な数のオプション属性のエントリを含めます。
3. ファイルを Oracle 9*i*FS にアップロードしてインスタンスを作成します。使用可能 / 必須の属性は、定義したカスタム・サブクラス（必須の値など）とその親クラス（Document、PublicObject など）に応じて異なります。

この方法で XML を使用すると、XML ファイル自体が Oracle 9*i*FS に格納されることはありません。格納されるのは、XML ファイルで定義したドキュメントのみです。

例 10-4 XML を使用したカスタム・サブクラスのドキュメントのインスタンス作成

次の XML ファイルは、Oracle 9*i*FS への挿入時に、定義済みフィールド Author および Title を持つ Shortstory カスタム・ドキュメント・タイプのインスタンスを作成します。Name 属性は必須で、親クラス Document から継承されています。このドキュメントを Oracle 9*i*FS にアップロードすると、ファイルは `example.shortstory` という名前でロードされたディ

レクトリに表示されます。XML ファイル自体は、このディレクトリに表示されません。ドキュメントの内容は、テキスト・エディタで開くと「This is a very short story.」というテキストになります。

```
<?xml version="1.0" standalone="yes"?>
<Shortstory>
 <Name>example.shortstory</Name>
 <Author>A Name</Author>
 <Title>A Description</Title>
 <Content>This is a very short story.</Content>
</Shortstory>
```

XML データ・ファイル内の日付値の書式設定

日付値は、XML データ・ファイル内で特別な処理を必要とします。XML は自己定義言語であり、データ・ファイルには格納されたデータ用のビルトイン・サポートが必要です。日付値の場合は、属性タグに書式パラメータが必要です。これにより、XML ファイルをどのようなデータ・ストアでも使用でき、データ・ストアではロード時に着信データのフォーマットを判断できます。

XML は、`java.text.SimpleDateFormat` クラスと同じフォーマット文字列を使用します。表 10-1 に、より便利なフォーマット値を示します。

表 10-1　Oracle 9*i*FS の XML 文書に使用される日付書式文字列

文字	意味
M	月
d	日
y	年
h	時間
m	分
s	秒
a	AM/PM（省略した場合、XML では 24 時間書式を想定）

表 10-2 に、日付値 2001 年 2 月 18 日午後 7 時 15 分を表す様々な書式を示します。

表 10-2　日付値 2001 年 2 月 18 日午後 7 時 15 分の書式のバリエーション

フォーマット文字列	日付値の文字列
M-d-yy	2-18-01
MM/dd/yy	02/18/01
MMM d, yyyy	Feb 18, 2001
M/d/yyyy	2/18/2001
M-d-yy hh:mm	2-18-01 19:15
yy-MM-dd hh:mm a	02-18-01 07:15 p.m.

Date フォーマット文字列の詳細は、`java.text.SimpleDateFormat` の Javadoc を参照してください。

XML ファイルに日付属性を入力するための書式を定義するには、属性タグにフォーマット引数を直接入力します。構文を次に示します。

```
<DateAttributeName
  format='formatString'>date_value</DateAttributeName>
```

DateAttributeName はカスタムまたは組込みの日付属性の名前で、formatString は date_value の入力に使用される書式と一致する箇所に記述されたコードを使用するフォーマット・テンプレートです。

例 10-5 日付値を含む XML ファイル

次の XML ファイルは、Oracle 9*i*FS への挿入時に、以前に定義した Shortstory カスタム・ドキュメント・タイプのインスタンスを作成し、定義済みフィールド Author、Title および Pubdate（Publication Date）を移入します。

```
<?xml version="1.0" standalone="yes"?>
<Shortstory>
 <Author>A Name</Author>
 <Title>A Description</Title>
 <Pubdate format='MM-dd-yy'>02-18-01</Pubdate>
</Shortstory>
```

配列属性の操作

ドキュメント属性が値の配列で構成されている場合があります。配列属性を移入するには、ArrayElement タグを使用します。

たとえば、ブール配列 OpenDays に、顧客の営業曜日を示す 7 つの TRUE/FALSE 値が含まれているとします。ブール値は、文字列 true および false を使用して設定されます。

```
.
.
.
<OpenDays>
 <ArrayElement>false</ArrayElement>
 <ArrayElement>true</ArrayElement>
 <ArrayElement>true</ArrayElement>
 <ArrayElement>true</ArrayElement>
 <ArrayElement>true</ArrayElement>
 <ArrayElement>true</ArrayElement>
 <ArrayElement>false</ArrayElement>
</OpenDays>
.
.
.
```

配列内で Date 値を設定する場合、各要素には単一値を設定する場合と同様に format パラメータが必要です。

たとえば、年の第 1 四半期の PayDays の配列を次のように入力できます。

```
.
.
.
<PayDays>
 <ArrayElement format="yy-MM-dd">01-01-15</ArrayElement>
 <ArrayElement format="yy-MM-dd">01-01-31</ArrayElement>
 <ArrayElement format="yy-MM-dd">01-02-15</ArrayElement>
 <ArrayElement format="yy-MM-dd">01-02-28<ArrayElement>
 <ArrayElement format="yy-MM-dd">01-03-15</ArrayElement>
 <ArrayElement format="yy-MM-dd">01-03-31</ArrayElement>
</PayDays>
.
.
.
```

属性としての Oracle 9*i*FS オブジェクトの使用

オプションで、Oracle 9*i*FS の Object 型の参照をファイルの属性として格納できます。たとえば、マネージャに Oracle 9*i*FS に送信されたドキュメントの編集管理を割り当てる場合は、そのマネージャの DirectoryUser オブジェクトを属性として使用できます。オブジェクトの単一値と配列の両方を格納できます。

Oracle 9*i*FS のオブジェクトを属性として使用するには、その参照型（RefType）を属性タグ内でパラメータとして識別する必要があります。属性の RefType には、表 10-3 のように 4 つの書式のうち 1 つを使用できます。

表 10-3 属性の RefType

RefType の値	使用方法	例

ID	この要素では、参照先オブジェクトのオブジェクト ID を値として指定します。IfsSimpleXmlParser は、オブジェクトをその ID 番号に基づいてリポジトリから選択します。	`<CLASSOBJECT>`
		`<Name>TypeDef</Name>`
		`<Superclass RefType='ID'>`
		`263`
		`</Superclass>`
		`<Description>`
		`XML demo test class`
		`</Description>`
		`</CLASSOBJECT>`

Path	PublicObject である埋込みオブジェクトを Path で参照できます。要素の値として指定した文字列は、パスとして解析されます。パスは、相対パスでも絶対パスでもかまいません。相対パスを機能させるには、IfsXmlParser をコールするアプリケーションで options ハッシュ表に Path オプションを設定する必要があります。	`<PublicObject>`
		`<Name> Test Object </Name>`
		`<Document RefType='Path'>`
		`/docs/test01.doc`
		`</Document>`
		`</PublicObject>`
		これは絶対パスです（/ で始まります）。パスのセパレータは、セッション固有の構成パラメータに依存するため注意してください。

表 10-3 属性の RefType（続く）

ValueDefault	埋込みオブジェクトの位置を、そのオブジェクトを参照する ValueDefault オブジェクト経由で指定できます。	`<PropertyBundle>`
		`<Update RefType="valuedefault">`
		`ParserLookupByFileExtension`
		`</Update>`
		`<Properties>`
		`<Property action = "add">`
		`<Name>xls</Name>`
		`<Value DataType="String">`
		`MyApplication.Parsers.XLSParser`
		`</Value>`
		`</Property>`
		`</Properties>`
		`</PropertyBundle >`

Search	この型の参照を使用すると埋込みオブジェクトを検索できます。埋込みオブジェクトが使用される属性値には、単一クラス（または単一クラスの派生クラス）に限定する ClassDomain、またはオブジェクトのクラスを指定する別の属性ラベル（classname）が必要です。RefType 属性の値には、検索対象となる属性の名前を指定する必要があります。タグの値には、一致させる値を指定する必要があります。検索で正確に一致する値が戻されない場合は、例外が発生します。	`<PublicObject>`
		`<Name> Test Object </Name>`
		`<Owner RefType='DistinguishedName'>`
		`jdoe@us.oracle.com`
		`</Owner>`
		`</PublicObject>`
		Owner 属性の ClassDomain では DirectoryUser オブジェクトに限定されているため、Search 参照は Classname 属性を設定せずに実行できます。

XML 名前空間の操作

名前空間は、あるコンテキストで使用される XML タグ名を、別のコンテキストで使用される同じ名前から分離して区別する方法です。たとえば、企業内では、すでに一連の XML データ・ファイルにルート・タグ Document を使用している場合があります。これらのファイルは、Oracle 9*i*FS の組込みの Document クラスと競合する可能性があります。企業内で、すべての XML ファイルを個別に変更するのではなく、新しい名前空間を設定し、定義済みの名前空間内で独自ファイルの特殊処理を提供できます。

XML 名前空間の概要

名前空間は、Universal Resource Indicator（URI）、通常は Uniform Resource Locator（URL）である一意名で識別されます。その用途は、World Wide Web のドメイン名に似てい

ます。Web のドメインを使用すると、インターネット上で一意の存在を定義できます。たとえば、Foo Corporation がドメイン名 `http://www.foo.com` を使用しているとします。世界中のインターネット・ブラウザで `http://www.foo.com/index.html` を指すと、World Wide Web に index.html という名前のページが文字どおり数百万あっても、一意のドメイン名に基づいて特定のページが識別されます。

XML 名前空間も同じ機能を持っています。名前空間用に、World Wide Web 上の他の名前空間と一致する可能性のない一意の文字列を選択します。そのため、名前空間のルートには、World Wide Web の登録済みドメイン名を使用するのが最善の方法です。会社が登録済みドメイン名を持っていない場合は、World Wide Web 上の他のエンティティ用に選択される可能性のない名前空間文字列を選択する必要があります。

たとえば、Oracle 9*i*FS のデフォルトの名前空間は `http://xmlns.oracle.com/ifs` です。これは、オラクル社が全製品用の一意の名前空間を定義するためにのみ使用しているドメイン名です。/ifs 名前空間に対して定義されているタグは、オラクル社の他の名前空間とも、World Wide Web 上の他のエンティティの名前空間とも競合しません。

URL がドキュメントを指すとは限らないため注意する必要があります。URL は、World Wide Web 全体で一意な識別子を作成する便利な方法です。

XmlParserLookup PropertyBundle の概要

Oracle 9*i*FS のプロパティのまとまりは、関連する名前 / 値のペアからなるクラスタです。Oracle 9*i*FS は、Property リストで名前を検索し、それに対応付けられた値を取り出して、システムのメタデータを検索できます。これにより、システムをカスタマイズし、Oracle 9*i*FS では引き続き必要な値の名前と場所を予測できます。

XmlParserLookup プロパティのまとまりを使用して、着信 XML 文書の処理に使用するパーサーを識別します。XmlParserLookup プロパティのまとまりには、表 10-4 のように 4 つの標準値があります。

表 10-4 XMLParserLookup プロパティのまとまりの要素

プロパティ名	プロパティ値	説明
IfsDefaultNamespace	http://xmlns.oracle.com/ifs	これは、別の名前空間を指定しないすべての XML ファイルで使用されるデフォルトの名前空間です。この値は、デフォルトで IfsSimpleXmlParser ではなくカスタム・パーサーが使用されるように、代替名前空間を指す値に変更できます。
http://xmlns.oracle.com/ifs	oracle.ifs.beans.parsers.IfsSimpleXmlParser	これは、Oracle 9*i*FS に付属するデフォルトの XML パーサーのパッケージのフルネームです。

表 10-4 XMLParserLookup プロパティのまとまりの要素（続く）

LiteralDocumentParser	oracle.ifs.beans.parsers. LiteralDocumentParser	これは、Oracle 9*i*FS のクラス階層の一部、Document（またはサブクラス化できる他のクラス）のカスタム・サブクラス、SimpleUser または ObjectList のいずれかとして認識されるルート要素を持たない XML ファイルに使用されるパーサーです。
IfsDefaultDTDValidation	FALSE	XML ファイルの妥当性を DTD ファイルと対照してチェックするかどうかを決定するブール値。
IfsRoundTripEnabled	TRUE	XML ファイルの内容をファイルの Content オブジェクトとして格納するかどうかを決定する論理値。

いずれにせよ、IfsSimpleXmlParser を使用せずに独自のカスタム・パーサーを使用する場合は、プロパティのまとまりを更新し、デフォルトの Oracle 9*i*FS 名前空間に対応付けられているプロパティ値を、独自パーサーのパッケージ・フルネームで置き換えることができます。ほとんどのドキュメントには IfsSimpleXmlParser を使用し、特定の名前空間のドキュメントには独自パーサーを使用する場合は、新規の名前空間を作成する必要があります。カスタム動作を実装する最善の方法は、Oracle 9*i*FS の名前空間はそのままにして、会社用の名前空間を定義してから、独自の名前空間を指すように IfsDefaultNamespace プロパティの値を変更することです。

名前空間の作成

名前空間を作成するには、XmlParserLookup プロパティのまとまりに新規プロパティを追加します。プロパティ名は名前空間文字列で、プロパティ値はカスタム・パーサーのパッケージ・フルネームです。

たとえば、`http://www.foo.com/myIfsNamespace` をカスタム FooParser に対応付ける場合、エントリは次のようになります。

プロパティ名	プロパティ値
http://www.foo.com/myIfsNamespace	foo.ifs.custom.parser.FooParser

独自の解析要件を処理できるように、必要な数だけ名前空間を作成できます。たとえば、顧客ごとに別個の名前空間を作成し、個々のニーズに合せて調整済みのパーサーを対応付けることができます。

以前のバージョンの Oracle 9*i*FS では、カスタム XML パーサーを作成して着信 XML ファイルの処理に使用できました。下位互換性を保つために、CustomXMLParser プロパティは引き続き有効です。ファイルに名前空間が対応付けられていても、その名前空間にパーサーが割り当てられていない場合は、CustomXmlParser が使用されます。この場合のエントリは次のようになります。

プロパティ名	プロパティ値
CustomXMLParser	foo.ifs.custom.parser.FooParser

XML 構成ファイルを使用して名前空間を作成できます。FooParser 用の名前空間を作成するには、次の XML 構成ファイルを使用します。

例 10-6 XML 構成ファイルを使用した XML 名前空間の作成

```
<?xml version="1.0" standalone="yes"?>
 <PropertyBundle>
  <Update RefType="valuedefault">
   XmlParserLookup
  </Update>
  <Properties>
   <Property action='add'>
    <Name>http://www.foo.com/myIfsNamespace</Name>
    <Value DataType="String">
     foo.ifs.custom.parser.FooParser
    </Value>
   </Property>
  </Properties>
</PropertyBundle>
```

詳細は、「XML ファイルを使用した Oracle 9iFS の構成」を参照してください。

名前空間の使用

Oracle 9*i*FS のインスタンス用に名前空間を定義した後に、その名前空間を使用する XML ファイルを作成できます。この名前空間は、XML ファイルのルート・タグのパラメータです。構文を次に示します。

```
<RootTag xmlns="namespaceString">
```

RootTag はドキュメントのインスタンス化に使用されるサブクラスで、namespaceString は Oracle 9*i*FS サーバー上の定義済み名前空間です。

例として、例 10-7 に前述のように定義された Foo Company の名前空間を使用する XML ファイルを示します。

例 10-7 カスタム名前空間を使用する XML ファイル

```
<?xml version="1.0" standalone="yes"?>
 <Document xmlns="http://www.foo.com/myifsNamespace">
 <Name>FooName</Name>
 <Description>This is a document for  www.foo.com</Description>
</Document>
```

この XML 文書を Oracle 9*i*FS に挿入すると、パーサー `foo.ifs.custom.parser.FooParser` を使用してデータ値が抽出されます。

XML データ・ファイルのレンダリング

XML ファイルは、Oracle 9*i*FS への情報転送のみでなく、Oracle 9*i*FS からの情報転送にも使用されます。オブジェクトは、作成方法に関係なく Oracle 9*i*FS から XML ファイルとしてレンダリングできます。

Oracle 9*i*FS に格納されたファイルには、大量のメタデータが格納されています。そのほとんどの情報は、Oracle 9*i*FS のコンテキスト内でのみ役立ちます（ドキュメントに割り当てられた ACL など）。XML ファイルをレンダリングするときに、レンダリングされる属性と詳細レベルを制御できます。

ClassObject のレンダリング・オプションの概要

Oracle 9*i*FS の各 ClassObject には、PropertyBundle オブジェクトが対応付けられています。PropertyBundle 名は、「Class PropertyBundle for <classname>」です。このプロパティのまとまりで次の 3 つのプロパティを設定し、ClassObject ごとにレンダリングする属性を決定できます。

- `IFS.RENDERER.RenderedAttributes`
- `IFS.RENDERER.InheritSuperClassPolicy`
- `IFS.RENDERER.InlineRendering`

IFS.RENDERER.RenderedAttributes: この属性は、ClassObject についてレンダリングされるすべての属性の文字配列リストです。プロパティが定義されていない場合は、クラスの拡張属性がレンダリングされ、それ以上の継承属性のレンダリングは `IFS.RENDERER.InheritSuperClassPolicy` により制御されます。

デフォルトでは、3 つの基本クラス（PublicObject、SchemaObject および SystemObject）に属性リストのセットが定義されています。PublicObject の場合、リストは Name および Description で構成されます。SchemaObject の場合は、Name 属性のみがリストされます。SystemObject の場合、Document の値は NULL に設定されます。このため、最も有効な情報をレンダリングできます（たとえば、Document の拡張属性は、Oracle 9*i*FS のコンテキス

ト外では意味を持ちません)。これらの設定は変更できますが、システム固有の多数のプロパティを通じて操作せずに、ほとんどの顧客が必要とする情報を提供できるように試みられています。

IFS.RENDERER.InheritSuperClassPolicy: これはブール値です。TRUE の場合は、親クラスの属性がそのクラスの `IFS.RENDERER` 設定に従ってレンダリングされます。親の `IFS.RENDERER.InheritSuperClassPolicy` も TRUE に設定されている場合は、階層の上位にある親クラスの属性もレンダリングされます。

FALSE の場合は ClassObject の属性のみがレンダリングされます。ClassObject のサブクラスの `IFS.RENDERER.InheritSuperClassPolicy` が TRUE に設定されている場合は、このポリシーが FALSE に設定されている ClassObject が検出されるまで、階層の上位へと各属性がレンダリングされます。検出された ClassObject とその子オブジェクトの属性のみがレンダリングされます。

`IFS.RENDERER.InheritSuperClassPolicy` の値が明示的に設定されていない場合、デフォルトは TRUE です。

IFS.RENDERER.InlineRendering: このプロパティには、3 つの値 Name、ID または Deep のいずれか 1 つを指定できます。このプロパティは、ClassObject が別のクラスの属性値としてインラインでレンダリングされるときに、レンダリングされる ClassObject の属性を定義するために使用します。

たとえば、属性 Manager を持つカスタム・ドキュメントを作成するとします。この属性は、発行前にファイルを検討するマネージャを表します。Manager の DataType を DirectoryUser に設定できます。情報のうち、カスタム・ドキュメントのデータ表に実際に表示される部分は、Manager として割り当てられた DirectoryUser の ID 番号のみです。大きい順序値が意味を持つのは Oracle 9*i*FS コンテキスト内のみのため、最も役立つ情報部分が XML ファイルにエクスポートされない場合があります。
`IFS.RENDERER.InlineRendering` 値を使用すると、他の ClassObject で属性として使用される ClassObject について、3 種類のインライン出力から選択できます。

Name を選択すると、埋込み ClassObject から Name 属性のみがレンダリングされます。

ID を選択すると、埋込み ClassObject から ID 属性のみがレンダリングされます。

Deep はディープ・レンダリングを指します。これは、埋込み ClassObject に対して定義された属性セット全体がレンダリングされることを意味します。この場合、埋込み ClassObject の IFS.RENDERER.RenderedAttributes 設定が考慮され、そこで指定された属性のみがレンダリングされます。

IFS.RENDERER.InlineRendering が明示的に設定されていない場合、デフォルト動作では埋込み ClassObject の Name のレンダリングが試行されます。Name が NULL の場合、Oracle 9*i*FS では ClassObject の ID がレンダリングされます。

ClassObject の PropertyBundle でのレンダリング・オプションの設定

Oracle 9*i*FS で PropertyBundle の値を設定する場合と同様に、ClassObject のレンダリング・オプションを設定します。

例 10-8 に、カスタム・サブクラス *Shortstory* の PropertyBundle の作成に使用できる XML ファイルを示します。このサブクラスは、親クラスの表示動作を継承するのではなく、特定の属性が含まれており、他のクラスの要素としてレンダリングされるときにはディープ・レンダリングを使用します（つまり、RenderedAttributes プロパティに指定されたすべての属性がレンダリングされます）。

例 10-8 XML を使用した ClassObject の PropertyBundle でのレンダリング・オプションの設定

```
<ClassObject>
  <Update RefType="name">
   Shortstory
  </Update>
  <PropertyBundle>
    <Name> Class Property Bundle for Shortstory </Name>
    <Properties>
      <Property>
        <Name>IFS.RENDERER.RenderedAttributes</Name>
        <Value DataType="String_array">
          <ArrayElement> Title </ArrayElement>
          <ArrayElement> Author </ArrayElement>
        </Value>
      </Property>
      <Property>
        <Name>IFS.RENDERER.InheritSuperClassPolicy</Name>
        <Value DataType="Boolean">false</Value>
      </Property>
      <Property>
        <Name>IFS.RENDERER.InlineRendering</Name>
        <Value> Deep </Value>
      </Property>
    </Properties>
  </PropertyBundle>
</ClassObject>
```

詳細は、「XML ファイルを使用した Oracle 9iFS の構成」を参照してください。

特殊処理を必要とするクラスの操作

デフォルトでは、特定クラスのデータ要素は常にレンダリングされます。このような特殊ケースは、全詳細を抑止できるようにそれぞれ独自の Boolean プロパティを持ちます。たとえば、デフォルトでは、ACL がレンダリングされるときには、すべての Access Control Entry（ACE）がレンダリングされます。ただし、ACL の全詳細ではなく ACL 名を表示す

るのみでよい場合は、特殊プロパティ IFS.RENDERER.ACCESSCONTROLLIST.ShowAces を FALSE に設定できます。

これらの特殊ケースには、継承はありません。たとえば、PropertyBundle クラスをサブクラス化した場合に、このプロパティを表示しないようにするには、そのプロパティをクラス PropertyBundle 内で設定する必要があります。プロパティを指定しない場合のデフォルトは TRUE です。

表 10-5 に、特殊処理を必要とするクラスのプロパティを示します。これらのプロパティはすべてブール値で、デフォルトでは *True* に設定されます。

表 10-5 特殊処理を必要とするプロパティ

プロパティ	説明
IFS.RENDERER.ACCESSCONTROLLIST. ShowAces	Access Control List オブジェクトの場合に指定します。TRUE の場合は、Access Control Entry のリスト全体がレンダリングに含まれます。
IFS.RENDERER.PUBLICOBJECT.ShowCategories	PublicObject の場合に指定します。TRUE の場合は、オブジェクトに割り当てられた Category のリスト全体がレンダリングに含まれます。
IFS.RENDERER.CLASSOBJECT.ShowAttributes	ClassObject の場合に指定します。TRUE の場合は、ClassObject の属性リスト全体がレンダリングされます。
IFS.RENDERER.ACCESSCONTROLENTRY. ShowAccessLevels	Access Control Entry の場合にのみ指定します。TRUE の場合は、各 Access Control Entry の個々のアクセス・レベルがレンダリングに含まれます。
IFS.RENDERER.PROPERTYBUNDLE. ShowProperties	TRUE の場合は、PropertyBundle のプロパティ・リスト全体がレンダリングされます。

ファイルの内容も特殊ケースです。CONTENTOBJECT 属性がプロパティ `IFS.RENDERER.RenderedAttributes` の属性リストに含まれている場合は、インラインでレンダリングされます。

XML ファイルを使用した Oracle 9*i*FS の構成

ここまでは、ファイルの内容と属性の操作を重点的に説明しました。XML を使用するとデータ要素を識別でき、IfsSimpleXmlParser ではその値に基づいてドキュメントを作成して移入できます。ファイルは Oracle9*i* データベースの Oracle 9*i*FS スキーマに関連レコードとして格納されます。

ただし、Oracle 9*i*FS では、スキーマ自体が自己記述型です。スキーマの要素は、表のレコードに SchemaObject として格納されます。認識されるデータ要素を持つドキュメントをアップロードして PublicObject を作成できるのと同様に、認識されるデータ要素を持つ XML 文書をアップロードして SchemaObject を作成できます。

アップロード・プロセスは、一時的な XML ファイルをアップロードして永続的なカスタム・ドキュメントを作成するという点では似ています。この場合は、SchemaObject を記述する一時的な XML ファイルをアップロードします。そのパラメータが解析され、SchemaObject が作成されます。XML ファイル自体がシステムに格納されることはありません。変更後は、XML ファイル自体のランタイム・インスタンスは破棄されます。

Oracle 9*i*FS のクラス階層の多くのオブジェクトは、XML 構成ファイルを使用して作成できます。XML 構成ファイルの使用例を示し、特定のオブジェクト型を作成する場合の付加的な考慮点について説明します。

XML を使用した Document クラスのサブクラスの作成

Document クラスのサブクラスを作成するには、新規 ClassObject を作成します。必須フィールドは、作成する ClassObject とそのスーパークラスの名前のみです。たとえば、Folder クラスには、独自の属性はありません。必要な属性は、すべてスーパークラスである PublicObject から継承します。ただし、Folder には、Folder オブジェクトの特殊処理を行えるように、独自のサブクラスが必要です。

ただし、サブクラス化を行うのは、主にカスタム属性を追加するためです。属性定義は、Attributes 表に ClassObject 定義とは別個に格納されます。このため、サブクラスの基本定義には影響を与えずに、ClassObject 定義の属性を追加したり削除できます。Oracle 9*i*FS は、ClassObject および Attribute 表のエントリを使用して、カスタム・サブクラスのドキュメントのインスタンスを格納する新規の表 ODM_<CustomSubclassName> を自動的に構成します。

例 10-9 Document クラスのサブクラスの作成

ここでは、カスタム Document タイプ Shortstory を作成する XML コード全体のリストを示し、続いて備考を示します。このファイルは、任意の名前で保存できます（.xml 拡張子が付いている場合）。このファイルを Oracle 9*i*FS に置くと、情報が解析され、オブジェクトが作成され、この XML ファイルのランタイム・インスタンスがメモリーから削除されます。

```
<?xml version="1.0" standalone="yes"?>
<ClassObject>

  <Name> Shortstory </Name>
```

```
  <Superclass RefType='name'>
    Document
  </Superclass>

  <Attributes>

    <Attribute>
      <Name> Title </Name>
      <DataType> String </DataType>
      <DataLength> 1000 </DataLength>
    </Attribute>

    <Attribute>
      <Name> Author </Name>
      <DataType> String </DataType>
      <DataLength> 50 </DataLength>
    </Attribute>

    <Attribute>
      <Name> PubDate </Name>
      <DataType> Date </DataType>
    </Attribute>

  </Attributes>
</ClassObject>
```

例に関する備考

- カスタム・サブクラスのルート・タグは <ClassObject> です。ClassObject 表には、Oracle 9*i*FS のクラス階層とカスタム・サブクラス内のオブジェクトに関するスキーマ情報が格納されます。
- ClassObject 名は Shortstory です。
- その親クラス、つまりスーパークラスは Document です。このクラスは Document クラスの属性を継承し、Document のスーパークラス PublicObject の属性を継承します。
- 次のタグ <Attributes> では、一度に複数の属性定義を入力できます。属性は Attributes 表に格納され、新規サブクラスの ID 番号でカスタム・サブクラスにリンクされます。
- 各属性は、Name および DataType の指定で定義されます。最初の 2 つの属性は String 属性で、追加のパラメータ DataLength が必要です。Date フィールドは LONG INTEGER 型の値として格納されるため、長さの引数は不要です。

これは、カスタム・サブクラスの定義ファイルを示す最小限の例です。

XML 構成ファイルを使用したプロパティのまとまりの作成

プロパティのまとまりを定義するには、<PropertyBundle> 要素タグを使用します。<PropertyBundle> 要素には、Oracle 9*i*FS クラス階層の PropertyBundle クラスの各属性の要素を含めることができます。

名前 / 値のペア、つまりプロパティのリストは、<Properties> 要素内で指定します。<Properties> 要素は、PropertyBundle 内の各 Property オブジェクトを定義する <Property> 要素を囲むために使用します。

各 <Property> 要素には、<Name> および <Value> 要素が含まれます。<Value> 要素は、Property オブジェクトの値（String、String_Array、PublicObject_Array など）の DataType を指定する属性 DataType を持ちます。<Value> 要素は、Property クラスの属性とは直接対応しません。<Value> 要素の値は、その DataType に応じて Property 属性の 1 つに格納されます。

例 10-10 PropertyBundle オブジェクトを定義する XML ファイル

```
<PropertyBundle>
    <Name>pb_test</Name>
    <CreateDate format="dd-MMM-yy">28-Oct-99</CreateDate>
    <Properties>
      <Property>
        <Name> Proptest1 </Name>
        <Value Datatype="String"> foo1 </Value>
      </Property>
      <Property>
        <Name> Proptest2 </Name>
        <Value Datatype="Date" format="dd-MMM-yyyy">
          28-Oct-2000
        </Value>
      </Property>
      <Property>
        <Name> Proptest3 </Name>
        <Value Datatype="Integer_Array">
          <ArrayElement>1</ArrayElement>
          <ArrayElement>934</ArrayElement>
          <ArrayElement>123</ArrayElement>
        </Value>
      </Property>
      <Property>
        <Name> Proptest4 </Name>
        <Value Datatype="SystemObject" Classname="Policy"
          RefType="name">
            SmbXmlRenderer
        </Value>
      </Property>
    </Properties>
```

```
</ PropertyBundle>
```

各 <Property> 要素では、Action 属性を使用して、PropertyBundle のプロパティを追加するか削除するかを指定できます。Action 属性の値は、Add および Remove です。PropertyBundle を作成する場合、Remove 属性を指定しても何も行われません。Action 属性を指定しない場合は、Add とみなされます。

既存の Property を置換するには、同じ名前で別のプロパティ値を追加します。同じ <PropertyBundle> 要素に同じ Property を追加して削除すると、<Property> 要素が宣言された順序に関係なく、削除指示により追加指示がオーバーライドされるため注意してください。

例 10-11 Action 属性を使用する XML ファイル

```
<PropertyBundle>
  <Update RefType='Id'>1234</Update>
  <Properties>
    <Property action='add'>
      <Name> Proptest1 </Name>
      <Value Datatype="String"> foo1 </Value>
    </Property>
    <Property action='remove'>
      <Name> Proptest2 </Name>
    </Property>
  </Properties>
</PropertyBundle>
```

単一の XML 構成ファイルを使用した複数オブジェクトの一括作成

Oracle 9*i*FS スキーマ言語では、複数のオブジェクトを作成する XML 構成ファイルの定義機能がサポートされます。XML 構成ファイルに複数のオブジェクトを含めるには、<ObjectList> タグで要素宣言をラップする必要があります。

複数オブジェクトを作成する場合の考慮点は、すべてのオブジェクトのコードを正しく入力する必要があることです。XML コードに誤りがあると、IfsSimpleXmlParser はそのオブジェクトでファイルの解析を停止し、オブジェクトは何も作成されません。ただし、ClassObject と DirectoryUser は例外です。この 2 つのオブジェクト型は明示的にコミットされます。この型のオブジェクトの作成時にエラーがあった場合は、XML ファイルを変更し、作成済みのオブジェクトの定義を削除する必要があります。

例 10-12 単一の XML ファイルを使用した複数オブジェクトの作成

次の XML 構成ファイルは、home フォルダに 2 つの Folder オブジェクト foo および goo を作成してから、Document オブジェクト Test document を作成して /home/goo ディレクトリに格納します。

```
<?xml version = '1.0' standalone = 'yes'?>
<ObjectList>
  <Folder>
    <Name> foo </Name>
    <FolderPath> /home </FolderPath>
  </Folder>
  <Folder>
    <Name> goo </Name>
    <FolderPath> /home </FolderPath>
  </Folder>
  <Document>
    <Name> Test document  </Name>
    <FolderPath> /home/goo </FolderPath>
    <Content> This is the content </Content>
  </Document>
</ObjectList>
```

11

カスタム・レンダラの作成

この章では、カスタム・レンダラの作成について重点的に説明します。この章の内容は、次のとおりです。

- レンダラの概要
- レンダラ・アプリケーションの記述

レンダラの概要

レンダラは、リポジトリに格納されているオブジェクトを取り出して、その内容を特定のフォーマットで出力します。レンダラによる情報出力は入力時のドキュメントと同じにできますが、同じでなくてもかまいません。レンダラの用途は、次のとおりです。

- ファイルの再構成とオリジナル・フォーマットでの表示
- ファイルの再構成と異なるファイル・フォーマットでの表示
- フィルタリングによる特定のファイル・コンポーネントのみの表示
- ファイル・コンポーネントからの値の計算

オリジナル・ドキュメント内の情報が解析され、Oracle 9*i*FS にオブジェクトとして格納されると、そのオブジェクトを様々なフォーマットとレイアウトでレンダリングできます。

すべての LibraryObject をレンダリングできます。たとえば、XML レンダラをコールして次を表示するアプリケーションを記述できます。

- オブジェクトがフォルダ、ACL または内容を含まない LibraryObject の場合、レンダリングされる出力は属性のみで構成されます。
- オブジェクトに内容が含まれている場合（つまり、Document オブジェクトまたはそのサブクラスの場合）は、レンダリングされる出力を属性のみ、内容のみまたはその両方で構成できます。

通常は、どのような LibraryObject でもレンダリングできますが、カスタム・レンダラや特定の種類のオブジェクトをレンダリングする目的で記述される傾向があります。たとえば、IfsSimpleXmlRenderer はどのような LibraryObject でもレンダリングできますが、PurchaseOrder レンダラがレンダリングできるのは PurchaseOrder オブジェクトのみで、処理できないオブジェクトを表示するように要求されると例外が発生します。

Document オブジェクトは、最も一般的にレンダリングされるオブジェクトです。

リポジトリ・オブジェクトを作成しないレンダラ

レンダラからの出力は読取り専用で、永続的ではありません。レンダラは、リポジトリやローカルに格納されているデータ・ファイルに、Document オブジェクトを自動的に作成することはありません。アプリケーションで、レンダリングされた出力を後で使用できるようにする必要がある場合は、それは Document オブジェクトを作成するか、データ・ファイルをローカルに保存するためのレンダリング後のステップになります。

IfsSimpleXmlRenderer

Oracle 9*i*FS には、本来、PublicObject を XML 文書としてレンダリングするための標準レンダラが組み込まれています。詳細は、第 10 章「XML および Oracle Internet File System」を参照してください。

レンダリングの動作

Oracle 9*i*FS のレンダリング・フレームワークでは、開発者は次のことができます。

- レンダラ・インタフェースを実装するクラスを記述して、カスタム・レンダラを作成します。
- カスタム・レンダラをリポジトリに登録します。
- LibraryObject から継承されるメソッドを使用してレンダラをコールします。これらのメソッドを使用して、デフォルト・レンダラとカスタム・レンダラをコールできます。

サーバー側操作であるレンダリング

Oracle 9*i*FS の Java クラス階層では、各 Oracle 9*i*FS オブジェクトに次の 2 つの表現があります。

- **Bean 側表現：** Document などのオブジェクト名で認識されます。
- **サーバー側表現：** S_Document など、S_ で始まるオブジェクト名で認識されます。

レンダリングはサーバー側操作のため、S_ クラスを使用する必要があります。

カスタム・レンダラの使用

カスタム・レンダラには、次のように様々な用途があります。

- アプリケーションに必要なフォーマットでリポジトリ・オブジェクトをレンダリングします。たとえば、Windows のアドレス帳アプリケーションでは、VCard を必要な VCard フォーマットでレンダリングする必要があります。
- ある MIME タイプから別の MIME タイプにドキュメントを変換します（たとえば、イメージ /JPEG からイメージ /GIF に変換します）。
- リポジトリ内の情報の計算と操作に基づいて仮想ドキュメントを作成します。
- 複数のソースからのデータを組み合せて複合ドキュメントを構成します。

レンダラ・アプリケーションの概要

カスタム・レンダラを開発するには、カスタム・アプリケーションからカスタム・レンダラへの情報の流れを理解する必要があります。

ステップ	領域	説明 / サンプル
1.	クライアント	アプリケーションが、該当する renderAs*Xxx*() メソッドをコールしてオブジェクトをレンダリングするように要求します。 例 : `Document.renderAsStream(rendererType, rendererName, options)`
2.	サーバー	サーバーが、`rendererType` および `rendererName` 引数を使用して、コールするレンダラを決定します。サーバーはレンダラをコールし、レンダリングするオブジェクトのサーバー側表現と、クライアントから提供されたオプションを渡します。 例 : `Renderer.renderAsStream(S_Document, options)`
3.	レンダラ	レンダラがドキュメントの表現とオプションを受け取ります。次に、オブジェクトをクライアントから要求されたフォーマットでレンダリングするために必要なカスタム・コードを実行します。

レンダラ・アプリケーションの記述

カスタム・レンダラ・アプリケーションを記述する手順は、次のとおりです。

1. レンダラ・クラスの記述

   既存の Oracle 9*i*FS レンダラの 1 つがアプリケーションのニーズと部分的に一致している場合は、そのレンダラを始点としてサブクラス化します。このようなレンダラが存在しない場合は、`oracle.ifs.server.renderers.Renderer` インタフェースを実装し、カスタム・クラスを作成する必要があります。

2. レンダラの配置

   レンダラを配置するには、レンダラをコンパイルして `.class` ファイルを作成し、作成された `.class` ファイルを含むフォルダ・ツリーを、Oracle 9*i*FS がインストールされているサーバーのディレクトリ `$ORACLE_HOME/ifs/custom_classes` に置く必要があります。

3. レンダラの登録

   レンダラを登録し、オブジェクト・クラスを特定のレンダラに接続します。レンダラを登録するには、Oracle 9*i*FS Manager、XML スクリプトまたは Java API を使用できます。

4. レンダラのコール

   レンダラをコールするとカスタム・レンダラがコールされ、該当する場合はスタイルシートが連結されて、結果がクライアントにレンダリングされます。

レンダラ・クラスの記述

カスタム・レンダラの作成時に、次の 2 つのアプローチから選択できます。

- 既存の Oracle 9*i*FS レンダラの 1 つがアプリケーションの要件と部分的に一致している場合は、既存のレンダラをサブクラス化できます。(既存の各 Oracle 9*i*FS レンダラは、`oracle.ifs.server.renderers.Renderer` インタフェースを実装します。)
- 既存の Oracle 9*i*FS レンダラを始点として使用できない場合は、`oracle.ifs.server.renderers.Renderer` インタフェースを実装し、新規のカスタム・クラスを作成する必要があります。

レンダラ・インタフェースの実装

どちらのアプローチを選択した場合も、カスタム・レンダラを記述するには、新しいクラスを作成するか、または既存レンダラをサブクラス化することによって、Renderer インタフェースを実装することになります。Renderer インタフェースでは、次の 2 つのメソッドが定義されます。

- renderAsStream()
- renderAsReader()

この 2 つの renderAs*Xxxx*() メソッドを使用すると、Oracle 9*i*FS クライアントまたはカスタム・アプリケーションでは、ドキュメントを必要なフォーマットでレンダリングできます。

各メソッドの構文と引数については後述します。

カスタム・レンダラを記述する手順は、次のとおりです。

- コンストラクタを記述します。
- renderAs*Xxxx*() メソッドを記述します。

コンストラクタの記述

各レンダラは、レンダラ用の標準コンストラクタを実装する必要があります。標準コンストラクタは、表 11-1 のようにパラメータを 1 つ取ります。

表 11-1 標準コンストラクタのパラメータ

パラメータ	データ型	説明
`session`	S_LibrarySession	現行のユーザーの LibrarySession のサーバー側表現。

例 11-1 コンストラクタの記述

```
public AirportDynamicRenderer(S_LibrarySession session) throws IfsException
 {
   m_IfsSession = ifs;
 }
```

renderAs*Xxxx*() メソッドの記述

表 11-2 に、次の 2 つの renderAs*Xxxx* ( ) メソッドのパラメータを示します。

- renderAsStream()
- renderAsReader()

表 11-2 renderAsStream および renderAsReader メソッドのパラメータ

パラメータ	データ型	説明
`lo`	S_LibraryObject	レンダリングするオブジェクト。
`options`	Hashtable	オプション・パラメータ。NULL でもかまいません。値を指定すると、options パラメータにより、オプションの名前 / 値のペアのセットを通じてレンダラの動作がさらに厳密に制御されます。通常は、文字コードの指定に使用します。

例： renderAsXxxx() メソッドの記述

この例では、必要な内容を含む文字列を作成し、StringBufferInputStream クラスを使用して、このメソッドが戻すオブジェクトである InputStream に変換します。

この例全体の構造は、次のとおりです。

1. renderAsStream() メソッドが renderAsString() をコールします。

   指定された LibraryObject を InputStream として renderAsStream() メソッドがレンダリングします。

2. renderAsString() メソッドが renderAirport() をコールします。
3. renderAirport() メソッドが、空港について必要な表現を含む String を作成します。

例 11-2 renderAsStream() メソッド

```
public InputStream renderAsStream(S_LibraryObject lo,
                                  Hashtable options)
    throws IfsException
  {
    InputStream in = null;
    String stream = renderAsString(lo, options);
    in = new ByteArrayInputStream(stream.getBytes());
    return in;
  }
```

例 11-3 renderAsString() メソッド

renderAsString() メソッドは renderAirport メソッドをコールし、結果を文字列として戻します。

```
public String renderAsString(S_LibraryObject lo,
                             Hashtable options)
    throws IfsException
  {
    String documentBody = null;
    documentBody = renderAirport(lo, options);
    return documentBody;
  }
```

サンプル・コード： renderAirport() メソッド

このメソッドの動作は、次のとおりです。

1. Oracle 9*i*FS のデフォルトの IfsSimpleXmlRenderer を使用し、Oracle 9*i*FS オブジェクトを XML としてレンダリングします。
2. パラメータ・オプションを通じて渡された XSL（スタイルシート）を取得します。
3. ステップ 1 からの XML 文書の XSL 変換を行います。
4. 生成された結果を戻します。

例 11-4 renderAirport() メソッド

```
public String renderAirport(S_LibraryObject lo,
                            Hashtable options)
  {
```

```
String resultOutput = "";
String xmlDoc = "";

DOMParser parser = new DOMParser();

try
{
  //Retrieve the result from the IfsSimpleXmlRenderer
  //This call is referencing the IfsSimpleXmlRenderer, as defined
  //in the file AirportDefinitionPolicyBundle.xml.
  Reader reader = lo.renderAsReader("RenderXmlAirportDefinition",
                                    "AirportDefinitionXmlRenderer",
                                    null);
  BufferedReader r = new BufferedReader(reader);
  for (String nextLine = r.readLine(); nextLine != null; nextLine =
                                                        r.readLine())
    xmlDoc += nextLine;

  //Turn the XML String into an XML Document
  XMLDocument xml = ParseDocument(xmlDoc, parser);

  XMLDocument xsl = null;
  //Retrieves the XSL Style Sheet in a String format
  //from the Hashtable parameter (named options) passed in to the Renderer.
  String xslContent = (String)options.get("xsl");

  if (xslContent != null && xslContent.length() > 0 &&
            !xslContent.toUpperCase().equals("NONE"))
  {
    try
    {
      //Turn XSL String into an XML Document
      xsl = ParseDocument(xslContent, parser);
    }
    catch (Exception e)
    {
      System.err.println("XSL : " + e.toString());
    }
  }

  if (xsl != null)
  {
    //Do the XSL Transformation.
    resultOutput = ProcessXML(xml, xsl, parser);
  }
  else
```

```
      {
        resultOutput = xmlDoc;
      }
    }
    catch (IfsException e)
    {
      resultOutput += ("<errorInRenderer type=¥"IFS¥">" + e.toString() +
                                                "</errorInRenderer>");
    }
    catch (IOException e)
    {
      resultOutput += ("<errorInRenderer type=¥"IO¥">" + e.toString() +
                                                "</errorInRenderer>");
    }
    //Return the result
    return resultOutput;
  }
```

Javadoc の参照先

レンダラの詳細は、Oracle 9*i*FS Javadoc の表 11-3 に示すクラスを参照してください。

表 11-3 Javadoc でのレンダラに関する参照先

クラス	用途
`oracle.ifs.server.renderers.Renderer`	すべてのレンダラについて実装する必要のあるインタフェース。2 つの主要メソッド renderAsStream() および renderAsReader() が含まれています。
`oracle.ifs.server.renderers.XmlRenderer`	カスタム XML レンダラを作成するための基本クラス。 XmlRenderer は、XML Document オブジェクトを InputStream または Reader に変換します。 S_LibraryObject を XML Document 表現に変換する拡張 XmlRenderer。
`oracle.ifs.server.renderers.IfsSimpleXmlRenderer`	XmlRenderer に基づく単純な XML レンダラのサンプル。

レンダラの配置

プロトコル・サーバーと他の標準 Oracle 9*i*FS コンポーネントがカスタム・レンダラにアクセスできるように、レンダラのクラスを含むフォルダ・ツリーを Oracle 9*i*FS の CLASSPATH に指定する必要があります。Oracle 9*i*FS には、そのための特別なディレクトリがあります。このディレクトリは `custom_classes` で、Oracle 9*i*FS サーバー・ソフトウェアが使用する CLASSPATH 環境変数に指定されています。

レンダラを配置する手順は、次のとおりです。

1. レンダラをコンパイルして `.class` ファイルを作成します。
2. 生成された `.class` ファイルを含むフォルダ・ツリーを、Oracle 9*i*FS がインストールされているサーバーのディレクトリ `$ORACLE_HOME/ifs/custom_classes` に置きます。

注意： コンパイル済みの Java コードは、Oracle 9*i*FS リポジトリではなくサーバー固有のファイル・システムにコピーする必要があります。

レンダラの登録

レンダラを登録するプロセスでは、オブジェクトのクラスと特定のレンダラ間の接続をマッピングします。この種のマッピングを格納するための基礎となるメカニズムは、PolicyPropertyBundle オブジェクトです。

PolicyPropertyBundle を使用したレンダラの登録

PolicyPropertyBundle は、特定のタイプの PropertyBundle オブジェクトです。通常、PropertyBundle は名前 / 値のペアの格納に使用されます。*Policy*PropertyBundle の場合、PropertyBundle の各 Property には Policy が格納されます。各 Policy には、特定のプロトコルに関するクラスとレンダラのマッピングが含まれます。

Oracle 9*i*FS の各クラスには、PolicyPropertyBundle が対応付けられています。クラスのオブジェクトがリポジトリから取り出されると、Oracle 9*i*FS は対応付けられている PolicyPropertyBundle をチェックし、そのオブジェクトを要求したプロトコルに基づいて、オブジェクトの表示に使用するレンダラを決定します。

Oracle 9*i*FS には複数のプロトコル（FTP、HTTP、SMB など）が含まれているため、プロトコルごとに特定の Policy を登録する必要があります。つまり、HTTP、FTP および SMB について、Policy を 1 つずつ登録します。

Property を使用して Policy を格納する場合の操作は、次のとおりです。

- 最初に、Property オブジェクトの Name 属性を指定します。*Property* の Name 属性には、Policy の Operation 属性の値を指定する必要があります。

- 次に、Property オブジェクトの Value 属性を指定します。Property の Value 属性は、Policy オブジェクトの Object ID を保持します。

各 Policy オブジェクトには、表 11-4 の属性が格納されます。

表 11-4 Policy オブジェクトの属性

属性	データ型	説明	例
Name	STRING	このカスタム・レンダラの名前。一意の名前を 1 ワードで空白を含めずに指定する必要があります。	POSmbRenderer
Operation	STRING	Property オブジェクトの名前。Operation では、PolicyBundle のハッシュ表のキーを指定するため、ハッシュ表で使用されている名前と正確に一致する必要があります。	SmbRenderer
Implementation Name	STRING	パッケージ名で始まるカスタム・レンダラの完全修飾クラス名。	ifs.demo.po.renderer.PoRenderer

登録方法

レンダラを登録するプロセスでは、オブジェクト・クラスを特定のレンダラに接続します。レンダラの登録には、表 11-5 に示すどの機能でも使用できます。

表 11-5 レンダラの登録方法

機能	メリット / デメリット
Oracle 9*i*FS Manager	操作が簡単で使用しやすい方がよい場合は、Oracle 9*i*FS Manager を使用します。Oracle 9*i*FS Manager を使用して登録できるのは、Oracle 9*i*FS Manager 機能と同じ Oracle 9*i*FS インスタンスに存在するレンダラのみです。
XML	XML を使用するのは、スクリプトを使用してレンダラを登録する場合や、レンダラを別の Oracle 9*i*FS インスタンスに配置する必要がある場合です。

表 11-5 レンダラの登録方法（続く）

機能	メリット / デメリット
Java	Java は次のような特殊ケースに使用します。 ■ クラスではなく特定のオブジェクトに使用するレンダラを登録する必要がある場合。 ■ 同じオブジェクト用に複数のレンダラを指定する必要がある場合。たとえば、1 つのドキュメントを、SMB アプリケーション用に 1 つ、HTTP アプリケーション用に 1 つ、合せて 2 つのレンダラでレンダリングする場合があります。

どの機能を使用する場合も、レンダラを登録する手順は次のとおりです。

- Policy オブジェクトを作成します。
- Policy オブジェクトの属性を設定します。
- Policy オブジェクト（またはそれを含む PolicyPropertyBundle）を、特定のクラスまたは複数のクラスに対応付けます。

Policy オブジェクトの属性の詳細は、「PolicyPropertyBundle を使用したレンダラの登録」を参照してください。

Oracle 9*i*FS Manager を使用したレンダラの登録 ： Oracle 9*i*FS Manager を使用してレンダラを登録する手順は、次のとおりです。

1. Oracle 9*i*FS Manager の「Object」メニューから「Register」を選択します。
2. 「Select Object Type」ウィンドウで「Renderer Lookup」を選択します。
3. 「Renderer Lookup Registry」ウィンドウで「Add」を選択します。
4. 「Renderer Lookup Entry」ウィンドウで、属性のテキスト・ボックスに値を入力します。
5. 「Class Association」リストから、このレンダラに対応付けるクラスを選択します。
6. 「OK」をクリックします。

XML を使用したレンダラの登録 ： XML を使用してレンダラを登録するには、特定のクラス（AirportDefinition など）とそれに対応付けられた PolicyPropertyBundle（AirportDefinitionPolicyBundle など）の間のマッピングを作成し、PolicyPropertyBundle を更新する XML ファイルを記述します。

例 11-5 XML を使用したレンダラの登録

```
<?xml version = '1.0' standalone = 'yes'?>
<CLASSOBJECT>
```

```
        <update reftype= 'name'>AirportDefinition</update>
        <policybundle reftype='name' classname='PolicyPropertyBundle'>
              AirportDefinitionPolicyBundle
        </policybundle>
</CLASSOBJECT>
```

注意：

- プロパティ名は、プロトコルに基づいて事前に定義されています。たとえば、SMB は SmbRenderer を検索し、FTP は FtpRenderer を検索します。
- `custom_classes` ディレクトリに対して完全修飾（パッケージ + クラス）名で指定する必要があります。

通常、PolicyPropertyBundle を削除する必要はなく、それに対応付けられているポリシーのリストを変更できます。XML を通じて、このまとまりに対応付けられているポリシーを追加、削除および更新できます。

PolicyPropertyBundle（PPB）は、ポリシーのコレクションです。各ポリシー名は、Oracle 9*i*FS のインスタンス間で一意である必要があります。通常、次の場合には PPB の作成時に重複値を取得します。

- 問題の名前を持つ PPB がすでに存在する場合
- 問題の名前を持つポリシーがすでに存在する場合

サンプル・コード： AirportDefinitionPolicyBundle

次のサンプル・コードでは、Property オブジェクトのコレクションである PolicyPropertyBundle オブジェクトを作成しています。Policy オブジェクトの属性の詳細は、「PolicyPropertyBundle を使用したレンダラの登録」を参照してください。

例 11-6 AiportDefinitionPolicyBundle

```
<?xml version="1.0" standalone="yes"?>
<POLICYPROPERTYBUNDLE>
  <NAME> AirportDefinitionPolicyBundle </NAME>
  <PROPERTIES>
    <PROPERTY>
      <NAME> RenderXmlAirportDefinition </NAME>
      <VALUE Datatype='SystemObject' Classname='Policy' >
        <NAME> AirportDefinitionXmlRenderer </NAME>
        <IMPLEMENTATIONNAME>
          oracle.ifs.server.renderers.IfsSimpleXmlRenderer
```

```
        </IMPLEMENTATIONNAME>
        <OPERATION> RenderXmlAirportDefinition </OPERATION>
      </VALUE>
    </PROPERTY>
    <PROPERTY>
      <NAME> CompleteDynamicRenderer </NAME>
      <VALUE Datatype='SystemObject' Classname='Policy' >
        <NAME> AirportDefinitionCompleteRenderer </NAME>
        <IMPLEMENTATIONNAME>
          ifs.sampleapps.OlivAirlines.AirportDynamicRenderer
        </IMPLEMENTATIONNAME>
        <OPERATION> CompleteDynamicRenderer </OPERATION>
      </VALUE>
    </PROPERTY>
  </PROPERTIES>
</POLICYPROPERTYBUNDLE>
```

レンダラのコール

アプリケーションで Oracle 9*i*FS の標準レンダラとカスタム・レンダラのどちらをコールする場合も、プロセスは同じです。アプリケーションは、oracle.ifs.beans.LibraryObject に定義されている適切な renderAs*Xxxx*() メソッドを使用して、レンダラをコールします。レンダラ・アプリケーションは、必要な出力のタイプに適したメソッドをコールする必要があります。

- InputStream オブジェクト（一連のバイト）
- Reader オブジェクト（文字列）

LibraryObject クラスから継承されるメソッドの使用

レンダラを使用するには、レンダリングする必要のあるオブジェクトの LibraryObject クラスから継承される、次のメソッドの 1 つをコールします。

```
public java.io.InputStream renderAsStream
                     (String rendererType,
                      String rendererName,
                      Hashtable options)
    throws IfsException

public java.io.Reader renderAsReader
                     (String rendererType,
                      String rendererName,
                      Hashtable options)
    throws IfsException
```

renderAsXxxx() メソッドのパラメータ

表 11-6 に、renderAs*Xxxx*() メソッドのパラメータを示します。これらのパラメータは、起動するレンダラを決定し、ターゲット・レンダラにオプションを渡すために使用されます。

表 11-6 renderAsXxxx() のパラメータ

パラメータ	データ型	説明	例
`rendererType`	STRING	Policy の Operation 属性の値。一意ではありません。	`SmbRenderer`
`rendererName`	STRING	Policy オブジェクトの名前。 一意である必要があります。	`VcardSmbRenderer`
`options`	Hashtable	使用する特定のレンダラごとに値が含まれます。	レンダラ固有のオプション。この Hashtable の値は、シリアライズ可能である必要があります。

`rendererType` 引数と `rendererName` 引数の違いは、次のとおりです。

- `rendererType` 引数は、Policy の Operation 属性の名前を指定する String 値です。
- `rendererName` 引数は、`rendererType` で指定された Operation を含む Policy の Name 属性を指定する String 値です。

`rendererType` および `rendererName` 引数により、使用するレンダラが決定されます。この決定は、次のように行われます。

- `rendererName` が NULL でない場合
  - `rendererType` で指定された Operation の*カスタム* Policy オブジェクト `rendererName` が取得されます。このカスタム Policy オブジェクトの ImplementationName 属性には、レンダラの完全修飾クラス名が含まれます。
- `rendererName` が NULL の場合
  - `rendererType` で指定された Operation の、この LibraryObject の*デフォルト* Policy オブジェクトが取得されます。このデフォルト Policy オブジェクトの ImplementationName 属性には、レンダラの完全修飾クラス名が含まれます。

レンダラの選択 ： `renderAs()` メソッドには、次の 2 つの使用方法があります。

- 明示的な指定： アプリケーションで特定のレンダラを明示的に指定します。
- デフォルト選択： リポジトリによりデフォルト・レンダラが選択されます。

表 11-7 に、それぞれの使用方法に対応するメソッドのシグネチャを示します。

表 11-7 メソッドのシグネチャ

使用方法	メソッドのシグネチャ
明示的	`renderAsStream (String rendererType,String rendererName, Hashtable options)` `renderAsReader (String rendererType,String rendererName, Hashtable options)`
デフォルト	`renderAsStream (String rendererType, null, Hashtable options)` `renderAsReader (String rendererType, null, Hashtable options)`

例： レンダラの明示的な選択 ： 特定のレンダラを選択するには、`rendererName` 属性を指定します。たとえば、アプリケーションでは、次の Stream 入力の場合に SimpleTextRenderer を明示的に指定できます。

```
renderAsStream("RenderAsText", "SimpleTextRenderer", myOptions)
```

`options` 引数は、指定のレンダラに文字コードなどの追加情報を渡します。使用可能なオプション、その設定、各オプション / 設定のペアの意味は、レンダラ固有です。次の例で、SimpleTextRenderer は `myOptions` Hashtable を使用して追加情報を取得します。使用可能なオプションの詳細は、各標準レンダラの Javadoc を参照してください。

例： デフォルト・レンダラの受入れ ： Oracle 9*i*FS により指定されたデフォルト・レンダラを受け入れるには、`rendererName` 属性を `null` に置き換えます。アプリケーションでは、この Document オブジェクトに次の 2 つのデフォルト・レンダラを使用できるものとみなして、リポジトリにデフォルト・レンダラを選択させることができます。

- デフォルトのテキスト・レンダラ
- デフォルトの XML レンダラ

```
renderAsStream("RenderAsText", null, myOptions)
renderAsStream("RenderAsXML",null, myOptions)
```

例： サーブレットを使用したカスタム・レンダラのコール

例 11-7 では、カスタム・レンダラをコールし、要求側クライアントに基づいて適切な XSL スタイルシートを渡し、結果をクライアントにレンダリングしています。

例 11-7 レンダラのコール

```
package ifs.sampleapps.OlivAirlines;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.Reader;
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Hashtable;

import oracle.ifs.common.IfsException;

import oracle.ifs.beans.Document;
import oracle.ifs.beans.LibraryObject;
import oracle.ifs.beans.LibraryService;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.Selector;

import oracle.ifs.search.AttributeQualification;
import oracle.ifs.search.AttributeSearchSpecification;
import oracle.ifs.search.SearchClassSpecification;
import oracle.ifs.search.SearchSortSpecification;

import oracle.ifs.beans.Search;


public class iFSAirportServlet extends HttpServlet
{

  private static final boolean DEBUG = false;
  private static final int ERRORCODE = 22000;

 /**
  * Constructs the iFSAirportServlet.
  */
  public iFSAirportServlet()
  {
  }

 /**
  * The servlet container calls the init method exactly once
```

```
 * after instantiating the servlet. The init method must
 * complete successfully before the servlet can receive any requests.
 *
 * @param      ServletConfig config
 * @exception  An exception a servlet throws when it encounters difficulty.
 * @pub
 */

 public void init(ServletConfig config)
 throws ServletException
 {
   super.init(config);
 }


/**
 * Called by the servlet container to allow the servlet to
 * respond to a request.
 * Calls the printContent method, which does the actual work.
 *
 * @param req  the ServletRequest object that contains the client's request.
 * @param res  the ServletResponse object that contains the servlet's respond.
 *
 * @exception  ServletException if an exception occurs that interferes with
 *                              the servlet's normal operation
 * @exception  java.io.IOException if an input or output exception occurs
 * @pub
 */
 public void service(HttpServletRequest request,
                     HttpServletResponse response)
 throws ServletException,
        IOException
 {
   // Retrieve Servlet's output stream
   PrintWriter out = new PrintWriter(response.getOutputStream());
   try
   {
     printContent(request, response, out);
   }
   catch (IfsException e)
   {
     out.println("<IfsException>" + e.toString() + "</IfsException>");
   }
   out.close();
 }

/**
```

```
* Calls the custom renderer and passes in the appropriate XSL
* style sheet to the custom renderer based on which client is
* making a request. Outputs the rendered result to the client.
*
* @param request  the ServletRequest object that contains the client's request.
* @param resonse  the ServletResponse object that contains the servlet's respond.
* @param out  for output
* @exception  IfsException if operation fails.
* @exception  IOException if an input or output exception occurs.
* @pub
*/
private void printContent(HttpServletRequest request,
                          HttpServletResponse response,
                          PrintWriter out) throws IOException,
                                                  IfsException
{
  // Retrieve User-Agent to know which kind of client is making the request.
  String userAgent = request.getHeader("User-Agent");

  LibraryService service = new LibraryService();

  String userName = request.getParameter("userName");
  String passWord = request.getParameter("passWord");
  String serviceName = request.getParameter("serviceName");
  LibrarySession  ifs = service.connect(userName, passWord, serviceName);
  // Finds the iFS object we are doing to render with a Selector.
  Selector mySelector = new Selector(ifs);
  // Select the Airportdefinition class based on its attribute: AIRPORTCODE.
  mySelector.setSearchClassname("AIRPORTDEFINITION");

  // The airport code is passed in as a parameter provide along with the URL.
  String code  = request.getParameter("code");
  mySelector.setSearchSelection("AIRPORTCODE = '" + code + "'");
  for (int i=0; i<mySelector.getItemCount(); i++)
  {
    LibraryObject lo = mySelector.getItems(i);
    String contentType = "";

    Hashtable h = new Hashtable();
    // Check if a given string ("HANDHTTP" here) matchs any substring
    // of the User-Agent parameter passed in, case insensative.
    // If it matches, the corresponding style sheet is read from iFS,
    // and put in an Hashtable, to be passed in to the renderer.
    if (userAgent.toUpperCase().indexOf("HANDHTTP") > -1)
    {
       h.put("xsl", getStyleSheetContent(ifs, "apHTML.xsl"));
       contentType = "text/html";
```

```
      }
      else if (userAgent.toUpperCase().indexOf("MOZILLA") > -1)
      {
         h.put("xsl", getStyleSheetContent(ifs, "apHTML.xsl"));
         contentType = "text/html";
      }
      else if (userAgent.toUpperCase().indexOf("UP") > -1)
      {
         h.put("xsl", getStyleSheetContent(ifs, "apWAP.xsl"));
         contentType = "text/x-wap.wml";
      }
      else if (userAgent.toUpperCase().indexOf("NOKIA") > -1)
      {
         h.put("xsl", getStyleSheetContent(ifs, "apWAP.xsl"));
         contentType = "text/x-wap.wml";
      }
      else if (userAgent.toUpperCase().indexOf("MOTOROLA") > -1)
      {
         h.put("xsl", getStyleSheetContent(ifs, "apVox.xsl"));
         contentType = "text/html";
      }
      else
      {
         h.put("xsl", "none");
      }

      response.setContentType(contentType);
     // Calls the custom renderer. Pass in the Hashtable
     // The custom renderer is registered in the
     // AirportDefinitionPolicyBundle.xml file.
    Reader reader = lo.renderAsReader("CompleteDynamicRenderer",
"AirportDefinitionCompleteRenderer", h);
     // Reads results from the Reader, and prints to the Servlet output.
     printAirport(reader, out);
    } //end for loop
  }


 /**
 * This method seaches and gets the content of an iFS document, the XSL style
 * sheet that will be passed to the custom renderer, based on its file name.
 **/
 private static String getStyleSheetContent(LibrarySession ifs, String xslName)
   throws IfsException
  {
    String retString = "";
```

```
    String className[] = {"DOCUMENT"};
    SearchClassSpecification scs = new SearchClassSpecification(className);
    scs.addResultClass("DOCUMENT");
    AttributeQualification aq1 = new AttributeQualification();
    aq1.setAttribute("DOCUMENT", "NAME");
    aq1.setOperatorType(AttributeQualification.LIKE);
    aq1.setValue(xslName);
    SearchSortSpecification ss = new SearchSortSpecification();
    ss.add("NAME" , SearchSortSpecification.ASCENDING);
    AttributeSearchSpecification ass = new AttributeSearchSpecification();
    ass.setSearchClassSpecification(scs);
    ass.setSearchQualification(aq1);
    ass.setSearchSortSpecification(ss);
    Search srch = new Search(ifs, ass);
    srch.open();
    try
    {  while (true)
       {
          LibraryObject lo = srch.next().getLibraryObject();
          Document d = (Document)lo;
          InputStream is = d.getContentStream();
          BufferedReader br = new BufferedReader(new InputStreamReader(is));
          for (String nextLine = br.readLine(); nextLine != null; nextLine =
br.readLine())
          {
            retString += nextLine;
          }
          br.close();
        }
    }
    catch (IfsException e)
    {
      if (e.getErrorCode() == ERRORCODE)
      {    [Is this set of braces needed????}
       }
      else
      {
        throw e;
      }
    }
    catch (IOException ioe)
    {
       System.err.println("IOException reading XSL : " + ioe.toString());
    }
    srch.close();
    return retString;
  }
```

```
/**
 * Prints out the renderer output to the client.
 *
 * @param reader   the renderer output passed in
 * @param out      for output
 */
 public static void printAirport(Reader reader,
                                 PrintWriter out)
 throws IOException
 {
    // Dumps the reader on the output.
    BufferedReader r = new BufferedReader(reader);
    for (String nextLine = r.readLine(); nextLine != null; nextLine = r.readLine())
    {
       out.println(nextLine);
    }
  }


}
```

カスタム・レンダラからの出力

サーブレットを実行する手順は、次のとおりです。

1. Web ブラウザを開きます。
2. 「Location」ウィンドウに次のコマンドを入力します。

   Http://*machineName*:*portNumber*/XSLRenderer?code=LAX&userName=*yourUserName*&passWord=*yourPassWord*&serviceName=*yourServiceName*

   各項目の意味は次のとおりです。

   machineName は、Oracle 9*i*FS を実行中のサーバーの名前です。

   portNumber は、Oracle HTTP Server を実行中のポートです。ポート番号を取得するには、「http://machineName:9090」と入力します。

   yourUserName は、この例で作成したユーザーです。

   yourPassWord は、このユーザーのパスワードです。

   serviceName は、Oracle 9*i*FS のサービス名です。サービス名のデフォルトは、ServerManager です。

ブラウザに LAX および Los Angeles が表示されます。このコードは SEA または SFO に変更できます。次に例を示します。

```
http://bsmith-sun:80/XSLRenderer?code=LAX&userName=gking&passWord=ifs&serviceNam
e=ServerManager
```

様々なデバイスからサーブレットへのアクセス

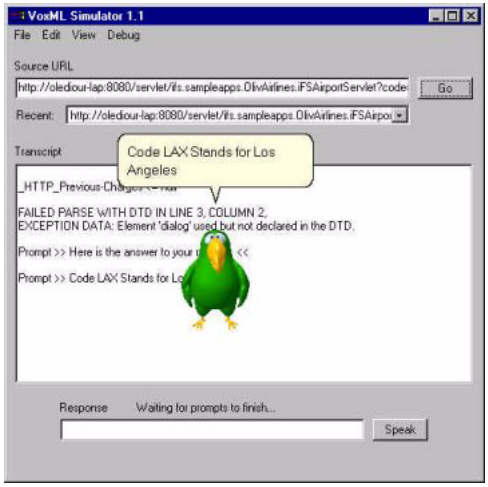
前の手順で使用したコマンドを様々なデバイスに入力し、出力がどうなるかを確認してください。

図 11-1 デスクトップの HTML ブラウザに表示される AirportDefinition

図 11-2 携帯電話に表示される AirportDefinition

図 11-3 パーソナル・デジタル・アシスタントに表示される AirportDefinition

図 11-4　音声シミュレータに表示される AirportDefinition



12

Web インタフェースのカスタマイズ

Oracle 9*i*FS でのサーブレットまたは JSP の使用方法は、Web 開発での使用方法の場合とほとんど同様です。この章では、Oracle 9*i*FS Java API をコールする拡張サーブレットおよび JSP の例について説明します。

この章の内容は、次のとおりです。

- サーブレットまたは JavaServer Pages（JSP）の使用
- サーブレットを使用したカスタム Web インタフェースの作成
- JSP を使用したカスタム Web インタフェースの作成

サーブレットまたは JavaServer Pages（JSP）の使用

Oracle Internet File System（Oracle 9*i*FS）は、Apache 駆動の Oracle HTTP Server に付属しています。ユーザーが HTML ブラウザを使用してカスタム・ドキュメント・タイプの属性を表示し、変更できるように、カスタム・ユーザー・インタフェースを作成できます。この方が、ユーザーに対して、XML を使用したり別のドキュメント・タイプから情報を解析して既存ドキュメントを変更するように要求するよりも効率的です。

サーブレットまたは JavaServer Pages を使用すると、カスタム情報を含む Web ページを動的に生成できます。

サーブレットの概要

サーブレットは、HTML ページを動的に作成する Java クラスです。サーブレットを登録し、実行時に使用する一意名を与えます。サーブレットを名前でコールし、表示して編集する Oracle 9*i*FS ドキュメントのパスをパラメータとして渡します。サーブレットには、アクセッサ・メソッドおよびミューテータ・メソッドが埋め込まれているか、または別個の JavaBeans を使用してファイル属性を取得します（この 2 つのメソッドは、getter メソッドおよび setter メソッドとも呼ばれます）。サーブレットでは、Java メソッドで計算された値の妥当性チェック、フォーマットおよび生成ができます。サーブレットは必要なデータ要素を生成した後に、`java.io.PrintWriter` オブジェクトを作成します。このオブジェクトには、情報の表示に使用する HTML ドキュメントが書き込まれます。サーバーは、生成された HTML ページをクライアント・ブラウザに戻します。

JSP の概要

JSP は、Java コードが埋め込まれている HTML ページです。（JSP は実行時にコンパイルされてサーブレットになるため、実際には同じことです。）

JSP では、HTML を拡張する特殊なタグが使用されます。タグにより、Java 変数とコード・フラグメントがカプセル化され、Java の処理をページの生成に使用する HTML コードと混合できます。Java コードを JSP 自体に置くこともできますが、通常は JavaBeans を添付する方が簡単です。

Oracle 9*i*FS では、JSP をファイル拡張子に割り当てることができます。その拡張子を持つファイルをユーザーが開くと、ドキュメント自体ではなく JSP がコールされます。JSP は、ドキュメントの内容、メタデータまたはその組合せを表示できます。

サーブレットと JSP の選択

サーブレットと JSP のどちらを選択するかは、個人の作業環境に大きく左右されます。

サーブレットを作成するには、Java コードと HTML コードの記述方法を知る必要があります。JSP を作成するには、Java、HTML および特殊な JSP 構文の知識が必要です。

全般に、Java デバッグ・ツールの方が、JSP に現在使用可能なデバッグ・ツールよりも洗練されています。JSP は動的にコンパイルされるため、通常は、プログラムの問題は配置後ま

で検出されません。一般に、デバッグに必要な時間と労力は、設計時よりも実行時の方がはるかに大きいとみなされています。

ただし、JSP は動的にコンパイルされるため、Oracle 9*i*FS の実行中に配置できます。サーブレットの場合は、変更するたびに Oracle 9*i*FS サーバーを再起動する必要があります。

理論上、JSP を使用するとチームが動的 Web ページで共同作業を行うことができます。たとえば、ユーザー・インタフェースの専門家は、ページのコントロールのレイアウトに専念でき、Java プログラマは別個にビジネス・ロジックを実装します。ただし、実際には、ページにビジネス・ロジックを追加すると、Java 開発者がページの所有者となり、コラボレーションによる利便性が乏しくなり、通常それ以降のページ更新はプログラマが処理します。

Oracle 9*i*FS では、JSP には明らかなメリットが 1 つあります。それは、JSP を登録すると、その使用は特定のファイル拡張子に自動的に対応付けられることです。サーブレットは、この方法では登録できません。ただし、JSP をファイル拡張子に登録した場合は、JSP 内でドキュメントの内容を表示するための手段を用意する必要があります。ドキュメントの内容は、Web インタフェースを使用して固有のフォーマットで表示できなくなります。

カスタム・サブクラス *Shortstory* の作成

Oracle 9*i*FS 用のカスタム Web インタフェースの作成について 2 つのアプローチを比較するために、この章の例では最初にサーブレット、次に JSP を使用して、同じ単純なアプリケーションを実装するプロセスを説明します。これらのサンプル Web インタフェースの実装をテストするには、カスタム・クラス Shortstory とサンプル・ドキュメントを作成する必要があります。例 12-1 および例 12-2 に、このカスタム・サブクラスを作成する XML 構成ファイルと Shortstory ドキュメントのインスタンスを作成する XML データ・ファイルを示します。Shortstory サブクラスは、カスタム属性 TITLE、AUTHOR および PUBDATE で Document クラスを拡張します。

例 12-1 Shortstory サブクラスの XML 構成ファイル

```
<?xml version="1.0" standalone="yes"?>
<classobject>
  <name> SHORTSTORY </name>
  <superclass RefType='name'>
    Document
  </superclass>
  <Description>A simple document type used to store short stories
    as plain text files, with custom attributes for AUTHOR, TITLE
    and PUBDATE (publication date).</Description>
  <attributes>
    <attribute>
      <name> TITLE </name>
      <DataType> STRING </DataType>
      <DataLength> 1000 </DataLength>
      <indexed> TRUE </indexed>
    </attribute>
```

```
    <attribute>
      <name> AUTHOR </name>
      <DataType> STRING </DataType>
      <DataLength> 50 </DataLength>
      <indexed> TRUE </indexed>
    </attribute>
    <attribute>
      <name> PUBDATE </name>
      <DataType> DATE </DataType>
      <indexed> TRUE </indexed>
   </attribute>
  </attributes>
</classobject>
```

例 12-2 Shortstory サブクラスのインスタンスを作成する XML データ・ファイル

```
<?xml version="1.0" standalone="yes"?>
<SHORTSTORY>
  <NAME> test.shortstory </NAME>
  <PUBDATE format='yyyy/MM/dd'>2002/04/21</PUBDATE>
  <AUTHOR>Ann Otherpoe</AUTHOR>
  <TITLE>The Visitor</TITLE>
  <CONTENT>"Welcome to my home, Lester" said Sheldon. Then he remembered Lester had
died in a bizarre salad dressing accident. As Sheldon fainted, he heard the sinister
sound of oil separating from vinegar.</CONTENT>
</SHORTSTORY>
```

サーブレットを使用したカスタム Web インタフェースの作成

この項では、Shortstory カスタム・ドキュメント・サブクラスからの属性 Title、Author および Pubdate を表示するサーブレットの作成プロセスについて説明します。この例では簡素化のために、サーブレットはターゲット・ドキュメントへのパスを URL に埋め込むことでコールされていますが、本番システムでは、通常はサーブレットを別個の HTML ページからコールします。

サーブレットのタスクの概要

Oracle 9*i*FS を操作するサーブレットは、次の基本タスクを実行する必要があります。

1. Oracle 9*i*FS に接続します。
2. 既存のドキュメントから値を取得し、それを HTML ページに埋め込んで戻します。
3. HTML ページに加えられた変更に基づいて値を設定し、変更後の値を HTML ページに表示します。

後述の「PropertySettingServlet.class の詳細コード」のサーブレットの例とこの項のコード・フラグメントは、HttpServlet クラスの 2 つの標準メソッド doGet() および doPost() を活用して前述のタスクを実行する方法を示しています。doGet() および doPost() の意図は、サーブレットに値を渡す 2 つの方法を示すことです。doGet() 要求のパラメータは URL に埋め込まれていますが、doPost() 要求では非表示の変数を通じてパラメータが送られます。このような特定の動作は、この 2 つのメソッドを使用する目標ではありません。目標は、doGet() メソッドを使用して、Oracle 9*i*FS に格納されているドキュメントから初期値を取得することです。doPost() メソッドは、ドキュメントの更新と変更内容の表示に使用されます。doGet() メソッドは、ドキュメントを最初に表示するときに使用されます。サーブレットが初期値を取得した後は、doPost() メソッドを使用して変更内容を格納し、表示するのみです。

この項のコード・フラグメントは、サーブレット・クラスでの相対位置ではなくフロー別に編成されています。サーブレットの完全なコメント付きコードは、「PropertySettingServlet.class の詳細コード」を参照してください。

PropertySettingServlet クラスの作成

サーブレットは、最初にサポート用クラスをインポートし、カスタム属性とセッション情報（ドキュメント・パスなど）を格納するグローバル変数をインスタンス化します。

PropertySettingServlet では、次のグローバル変数が使用されています。

```
private String m_path;
private String m_author = "";
private String m_title = "";
private String m_pubdate = "";
private Folder m_folder;
private PublicObject m_po;
private SimpleDateFormat m_dateFormatter =
    new SimpleDateFormat("MM-dd-yy");
```

この変数のうち最も重要なのは m_dateFormatter で、String 値と Date オブジェクト間の変換時に使用される日付書式が格納されます。テンプレート MM-dd-yy は、日付文字列が 2 桁の月、2 桁の日および 2 桁の年として入力されることを示します。詳細は、java.text.SimpleDateFormat クラスの Javadoc を参照してください。

doGet() メソッドの作成

doGet() メソッドは、2 つの引数 HttpServletRequest および HttpServletResponse を取ります。HttpServletRequest 引数は、クライアントからの着信パラメータをすべてカプセル化します。HttpServletResponse 引数は、元のクライアントに渡される発信パラメータをすべてカプセル化します。

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
```

以降の説明では、HttpServletRequest 引数を要求と呼びます。

Oracle 9*i*FS への接続

要求を通じて渡されるパラメータの 1 つは、値 IfsHttpLogin です。この値を使用すると、Oracle 9*i*FS へのログインに使用した既存のセッションに基づいて、新規 LibrarySession をインスタンス化できます。LibrarySession は、現行のディレクトリやユーザーのログイン情報など、ユーザーによる Oracle 9*i*FS への現行の接続に関して役立つ情報をカプセル化します。

```
IfsHttpLogin ifsLogin =
    (IfsHttpLogin) request.getSession(true).getValue("IfsHttpLogin");

LibrarySession ifsSession = ifsLogin.getSession();
```

Oracle 9*i*FS に格納された値の取出し

次の try/catch ブロックは、Oracle 9*i*FS から選択したドキュメントのカスタム属性にアクセスするために使用されます。現行のセッションに基づいてルート・フォルダを検出し、フォルダ・オブジェクトとしてインスタンス化できます。要求からは、URL で渡されたパス引数にアクセスできます。Oracle 9*i*FS は、ルート・フォルダを起点としてパスをたどり、ドキュメントを識別して PublicObject をインスタンス化できます。PublicObject（m_po）は、Oracle 9*i*FS に格納されている Document のランタイム・インスタンスです。ドキュメントのランタイム・バージョンを使用すると、アクセッサ・メソッドをコールしてカスタム属性 Title、Author および Pubdate を取得できます。

```
try
 {
  m_folder = ifsSession.getRootFolder();
  m_path = (String)request.getParameter("path");
  m_po = m_folder.findPublicObjectByPath(m_path);
  m_title = getTitleFromPO(m_po, ifsSession);
  m_author = getAuthorFromPO(m_po, ifsSession);
  m_pubdate = getPubdateFromPO(m_po, ifsSession);
 }
 catch (Exception e )
 {
   e.printStackTrace();
 }
```

次のコード・フラグメントは、選択したドキュメントからカスタム属性 Title を取得するためのアクセッサ・メソッド（getter）です。このメソッドが Title 文字列を取得するには、PublicObject と現行の LibrarySession が必要です。新規の文字列変数 title が、値を戻すために使用されます。現在の値は AttributeValue としてアクセスされます。AttributeValue が存在しない場合（av==null）、このメソッドは空の文字列を戻します。AttributeValue が存在する場合は、title 変数がその String 値に設定されます。String 値が NULL の場合は、title 値が空の文字列に設定されます。title 変数は、空の String として、または PublicObject からの属性を表す String 値として戻されます。

```
private String getTitleFromPO(PublicObject po, LibrarySession ifsSession)
  {
```

```
    String title = "";
    try
    {
      AttributeValue av = po.getAttributeByUpperCaseName("TITLE");
      title = (av == null) ? "" : av.getString(ifsSession);
      if (title == null) title = "";
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
    return title;
  }
```

アクセッサ・メソッド getAuthorFromPO および getPubdateFromPO の動作も同様です。

ページの戻り

PublicObject からの変数値を設定すると、サーブレットは HTML ページを書き出して値を表示する準備が完了したことになります。次の行では、応答の MIME タイプを HTML に設定し、応答用の PrintWriter オブジェクトをインスタンス化しています。

```
    response.setContentType("text/html");
    PrintWriter out = new PrintWriter (response.getOutputStream());
    out.println("<html>");
    out.println("<body bgcolor=WHITE>");
    out.println("<p>");
```

例 12-3 の 1 行目から HTML フォームが始まります。サーブレットは、このフォームに属性を表示するコントロールを作成します。このフォームの再描画に使用されるメソッドは POST で、これはフォームの送信時に doPost() メソッドが使用されることを意味します。このフォームのアクションは shortstory サーブレットをコールすることで、shortstory はサーブレットを Oracle HTTP Server に登録するときに使用した名前です。

例 12-3 HTML 出力のフォーマット

```
out.println("<p>Author: <INPUT TYPE=TEXT NAME=author VALUE=¥"" +
m_author + "¥">");
out.println("<p>");
out.println("Title: <INPUT TYPE=TEXT NAME=title VALUE=¥"" + m_title +
"¥">");
out.println("<p>");
out.println("Publication Date: <INPUT TYPE= TEXT NAME=pubdate VALUE=¥""
+ m_pubdate + "¥">");
out.println("<p>");
```

例 12-4 の各行では、変数値を表示するコントロールを作成しています。それぞれ、編集可能フィールドにテキスト値を表示する Text Input コントロールです。

例 12-4 Set Display 変数

```
out.println("<p>Author: <INPUT TYPE=TEXT NAME=author VALUE=¥"" +
  m_author + "¥">");
out.println("<p>");
out.println("Title: <INPUT TYPE=TEXT NAME=title VALUE=¥"" + m_title +
  "¥">");
out.println("<p>");
out.println("Publication Date: <INPUT TYPE= TEXT NAME=pubdate VALUE=¥""
  + m_pubdate + "¥">");
out.println("<p>");
```

ドキュメントへのパスは、例 12-5 のように Hidden Input コントロールで設定されます。非表示フィールドには、表示されない値が格納されます。値は、フォームの送信時に要求オブジェクトに組み込まれます。

例 12-5 非表示フィールドを使用したドキュメントの PATH の送信

```
out.println("<INPUT TYPE=HIDDEN NAME=path VALUE=¥"" + m_path + "¥">");
  out.println("<p>");
```

最後のコントロールは、Submit Input コントロールです。このコントロールは、ラベル Save が付いたプッシュ・ボタンとして表示されます。このボタンをクリックすると、編集済みの値が要求の変数を通じて doPost() メソッドに送られます。

```
out.println("<INPUT TYPE=SUBMIT VALUE=¥"Save¥">");
```

例 12-6 のコードに、フォームを完了して PrintWriter オブジェクトを閉じる方法を示します。これにより、HTML ページがクライアント・ブラウザに送信され、そこでページが表示されます。

例 12-6 フォームの終わり

```
// Complete the form.

 out.println("</FORM>");
 out.close();
}
```

doPost() メソッドの作成

doPost() メソッドは、ユーザーが HTML ページの「Save」ボタン（Submit Input コントロール）をクリックした時点でコールされます。属性値は、要求の引数を通じて渡されます。doPost() メソッドは、Oracle 9*i*FS から値を取得するかわりに、値を更新（設定）して表示します。それ以外は、doPost() および doGet() メソッドは実質的には同じです。

要求からの値の取得

doPost() での最初のタスクは、グローバル変数を要求経由で渡された値に設定することです。この場合も、サーブレットはユーザーの現行の作業セッションに基づいて LibrarySession を作成します。

```
IfsHttpLogin ifsLogin =
    (IfsHttpLogin) request.getSession(true).getValue("IfsHttpLogin");
LibrarySession ifsSession = ifsLogin.getSession();
```

各属性変数は、要求を通じて渡された値に設定されます。すべての変数は、String として渡されます。Pubdate は文字列として表示できますが、Oracle 9*i*FS に格納するには java.util.Date オブジェクトに変換する必要があります。サーブレットは、doPost() メソッドに String 表現と Date オブジェクト表現の両方を使用します。

```
m_path = request.getParameter("path");
m_author = request.getParameter("author");
m_title = request.getParameter("title");
m_pubdate = request.getParameter("pubdate");
Date pubDate = m_dateFormatter.parse(m_pubdate);

m_folder = ifsSession.getRootFolder();
m_po = m_folder.findPublicObjectByPath(m_path);
```

変更後の値の格納

次のコード・ブロックでは、変数値を PublicObject の属性として Oracle 9*i*FS に格納しています。PublicObject が存在するかどうかをテストした後に、各変数を使用して AttributeValue オブジェクトがインスタンス化されます。それぞれの値は、setAttribute() メソッドを使用して Oracle 9*i*FS に格納されます。

```
if (m_po != null)
{
  AttributeValue m_avAuthor =
   AttributeValue.newAttributeValue(m_author);

  m_po.setAttribute("AUTHOR",m_avAuthor);

  AttributeValue m_avTitle =
   AttributeValue.newAttributeValue(m_title);

  m_po.setAttribute("TITLE",m_avTitle);
```

```
    AttributeValue m_avPubdate =
     AttributeValue.newAttributeValue(pubDate);

    m_po.setAttribute("PUBDATE",m_avPubdate);
  }
```

取得された変数値を持つページの戻り

最後に、doPost() メソッドは HTML ページをクライアント・ブラウザに戻します。doPost() メソッドのこの部分は、doGet() メソッドと同じです。

```
response.setContentType("text/html");
 PrintWriter out = new PrintWriter (response.getOutputStream());
 out.println("<html>");
 out.println  ("<body bgcolor=WHITE>");
  out.println("<FORM NAME=f METHOD=POST ACTION=¥"shortstory¥">");
  out.println("Author: <INPUT TYPE=TEXT NAME=¥"author¥" VALUE=¥"" +
  m_author + "¥">");
 out.println("<p>");
  out.println("<p>Title:<INPUT TYPE=TEXT NAME=¥"title¥" VALUE=¥"" +
  m_title + "¥">");
 out.println("<p>");
  out.println("<p>Pubdate:<INPUT TYPE=TEXT NAME=¥"pubdate¥" VALUE=¥""
  + m_pubdate + "¥">");
 out.println("<p>");
  out.println("<INPUT TYPE=HIDDEN NAME=path VALUE=¥"" + m_path + "¥">");
 out.println("<p>");
 out.println("<INPUT TYPE=SUBMIT VALUE=¥"Save¥">");
 out.println("</FORM>");
 out.close();
}
```

サーブレットのコンパイル

第 1 章「Oracle Internet File System SDK スタート・ガイド」の「スタート・ガイド」の指示に従って操作したかどうかを確認してください。Java IDE を使用するか、コマンドラインから javac を使用してサーブレットをコンパイルします。

サーブレットの配置

Oracle 9*i*FS サーバーとして機能しているのと同じコンピュータ上でアプリケーションを開発できる場合は、クラスを <IFS_HOME>/custom_classes ディレクトリに直接コンパイルできます。それ以外の場合は、クラスとパッケージのフォルダ階層（oracle/ifs/examples/devdoc/webui/PropertySettingServlet.class）を、サーバー固有のファイル・システムの <IFS_HOME>/custom_classes ディレクトリにコピーします。

サーブレットを Oracle HTTP Server に登録するには、Oracle 9*i*FS 用の Oracle HTTP Server 構成ファイル（`ifs.properties`）に 1 行を追加します。

1. ファイル `<ORACLE_HOME>/Apache/JServ/servlets/ifs.properties` のディレクトリに移動します。
2. テキスト・エディタで `ifs.properties` を開きます。
3. このファイルの末尾に次の 1 行を追加します。構文は `servlet.<servletname>.code=<package.class>` です。<servletname> はサーブレットのコールに使用する一意の記述名で、<package.class> はサーブレット・クラスで終わるパッケージ・フルネームです。パッケージ名は、CLASSPATH のどこかで始まるディレクトリ階層と対応している必要があります。

   ```
   servlet.shortstory.code=oracle.ifs.examples.devdoc.webui.PropertySettingServlet
   ```

4. ファイルを保存します。

サーブレットの実行

サーブレットを配置後に実行する手順を次に示します。この手順は、Shortstory カスタム・ドキュメント・タイプを作成し、`test.shortstory` ドキュメントをホーム・ディレクトリにアップロードしていることを前提としています。

1. Oracle 9*i*FS Web インタフェースにログインします。
2. ファイル `test.shortstory` をクリックします。ファイルのテキストの内容が新規ブラウザ・ウィンドウに表示されます。
3. ウィンドウを閉じます。
4. Oracle 9*i*FS のビューア・ページで、Web ブラウザのナビゲーション・バーに次の URL を入力します。

   ```
   http://<iFSHost>/ifs/shortstory?path=home/<UserName>/sample.shortstory
   ```

   <iFSHost> はホスト・サーバー名で、<UserName> は Oracle 9*i*FS ユーザー名です。疑問符（?）は、その後に続くのがコール時にサーブレットに渡されるパラメータであることを示します。
5. サーブレットでは、ファイルとそのカスタム属性が表示されます。属性を変更し、「Save」ボタンをクリックして変更内容を格納できます。

Web インタフェースで「Edit」->「Properties」を選択すると、入力後に変更結果を表示できます。

PropertySettingServlet.class の詳細コード

PropertySettingServlet クラスの詳細コードとコメントを次に示します。

```
package oracle.ifs.examples.devdoc.webui;

import java.io.IOException;
// Reports exceptions to classes in java.io.*;

import java.io.PrintWriter;
// Used to write out the HTML page to be used by the client browser.

import java.text.SimpleDateFormat;
// Used to convert String values to Java Date objects, and vice-versa.

import java.util.Date;
// Used to store values as Java Date objects.

import javax.servlet.ServletConfig;
// Stores configuration parameters passed between the server and the servlet
// when the servlet is initialized.

import javax.servlet.ServletException;
// Reports exceptions to classes in javax.servlet.*

import javax.servlet.http.HttpServlet;
// Used to instantiate HttpServlet objects.

import javax.servlet.http.HttpServletRequest;
// Encapsulates the parameters sent to a servlet when it is invoked.

import javax.servlet.http.HttpServletResponse;
// Encapsulates the text or binary object sent in response to a servlet
// request.

import oracle.ifs.adk.security.IfsHttpLogin;
// Utility class that provides access to the user's log-in information for
// re-use by the servlet.

import oracle.ifs.beans.Folder;
// Used to instantiate an Oracle 9iFS Folder object.

import oracle.ifs.beans.LibrarySession;
// Used to instantiate a LibrarySession object, which encapsulates useful
// information about the user's current work session.

import oracle.ifs.beans.PublicObject;
// Used to instantiate the selected Shortstory document. Shortstory is a
```

```
// subclass of Document, which is a subclass of PublicObject. Any item that
// can be stored in an Oracle 9iFS folder is a PublicObject.

import oracle.ifs.common.AttributeValue;
// Used as an interpolator to convert primitive values to values suitable
// for storage in the Oracle 9iFS repository.

/* PropertySettingServlet
 *
 * The purpose of this servlet is to display and update the custom
 * attributes in a Shortstory file.
 *
 * This servlet demonstrates how to set values in a custom class without
 * creating supporting JavaBeans. We make the same calls, but we do it with
 * methods in this class rather than separate beans. The methods are:
 * getAuthorFromPO(), getTitleFromPO, and getPubdateFromPO().
 *
 * In general, the JavaBeans approach is preferred, because it makes your
 * application more flexible and your code reusable. However, a JavaBean is
 * not required, particularly in situations where the code has a very
 * specific purpose.
 *
 * Copyright (c) 2001 Oracle Corporation. All rights reserved.
 */

public class PropertySettingServlet extends HttpServlet
{

  // Initialize global variables
  private String m_path;
  private String m_author = "";
  private String m_title = "";
  private String m_pubdate = "";
  private Folder m_folder;
  private PublicObject m_po;

  // Create a SimpleDateFormat template used to convert strings with a two-
  // digit month, two-digit day, and two-digit year to a Java Date object.

  private SimpleDateFormat m_dateFormatter =
    new SimpleDateFormat("MM-dd-yy");

  // The init method is used by the HttpServer at runtime to pass
  // configuration parameters. The configuration parameters are set
  // automatically by Oracle 9iFS.

  public void init(ServletConfig config) throws ServletException
```

```
{
    super.init(config);
  }

  // The doGet() method access the values of an existing Shortstory
  // document stored in Oracle 9iFS and displays the current attribute
  // values. The parameters are the standard HttpServletRequest and
  // HttpServletResponse objects.

  public void doGet(HttpServletRequest request, HttpServletResponse
    response)
    throws ServletException, IOException
  {

    // Instantiate an IfsHttpLogin object with the user's log in
    // information.

    IfsHttpLogin ifsLogin =
      (IfsHttpLogin) request.getSession(true).getValue("IfsHttpLogin");

    // Create a new LibrarySession based on the user's current session.

    LibrarySession ifsSession = ifsLogin.getSession();

    try
    {

      // Set the m_folder variable to the root folder on this Oracle 9iFS
      // instance.
      m_folder = ifsSession.getRootFolder();

      // Set the m_path variable to the "path" parameter embedded in the
      // URL that called the servlet.
      m_path = (String)request.getParameter("path");

      // Set the m_po variable to the PublicObject identified by the path,
      // starting from the root folder.
      m_po = m_folder.findPublicObjectByPath(m_path);

      // Set the m_author variable to the Author attribute of the
      // PublicObject.
      m_author = getAuthorFromPO(m_po, ifsSession);

      // Set the m_title variable to the Title attribute of the
      // PublicObject.
      m_title = getTitleFromPO(m_po, ifsSession);
```

```
  // Set the Pubdate variable to the Pubdate attribute of the
  // PublicObject.
  m_pubdate = getPubdateFromPO(m_po, ifsSession);
}
catch (Exception e )
{
  e.printStackTrace();
}
// Begin writing the HTML page. Start by setting the MIME type to HTML
response.setContentType("text/html");

// Create a new PrintWriter object that represents the output stream of
// the response object.
PrintWriter out = new PrintWriter (response.getOutputStream());

// Begin the HTML page.
out.println("<html>");

// Set its background color to white.
out.println("<body bgcolor=WHITE>");
out.println("<P>");

// Create a Form object on the HTML page. When submitted, call the
// doPost()method of the PropertySettingServlet. The servlet must be
// registered on the Oracle HTTP server with the name "shortstory."
out.println("<FORM NAME=f METHOD=POST ACTION=¥"shortstory¥">");

// Create an HTML Text Input control named 'author', and populate it
// with the m_author variable. Enclose the value in quotation marks
// to allow for space characters (%20).
out.println("<P>Author: <INPUT TYPE=TEXT NAME=author VALUE=¥"" +
  m_author + "¥">");
out.println("<P>");

// Create an HTML Text Input control named 'title', and populate it
// with the m_title variable.
out.println("Title: <INPUT TYPE=TEXT NAME=title VALUE=¥"" +
  m_title + "¥">");
out.println("<P>");

// Create an HTML Text Input control named 'pubdate', and populate it
// with the m_pubdate variable.
out.println("Publication Date: <INPUT TYPE=TEXT NAME=pubdate VALUE=¥""
  + m_pubdate + "¥">");
out.println("<P>");

// Store the path to the selected PublicObject so that it can be passed
```

```
    // to the servlet when the form is submitted.
    out.println("<INPUT TYPE=HIDDEN NAME=path VALUE=¥"" + m_path + "¥">");

    // Create a Submit button.
    out.println("<P><INPUT TYPE=SUBMIT VALUE=¥"Save¥">");

    // Complete the form.

    out.println("</FORM>");

    // Return the page to the client browser.
    out.close();

  }

  // The doPost() method receives modified attribute values via the
  // HTTPServletRequest and updates the selected Shortstory document.

  public void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
{
    try
    {

      // Instantiate an IfsHttpLogin object with the user's log in
      // information.
      IfsHttpLogin ifsLogin =
        (IfsHttpLogin) request.getSession(true).getValue("IfsHttpLogin");

      // Create a LibrarySession based on the user's current work session.
      LibrarySession ifsSession = ifsLogin.getSession();

      // Set the global variables to the values passed in through the
      // HttpServletRequest argument.
      m_path = request.getParameter("path");
      m_author = request.getParameter("author");
      m_title = request.getParameter("title");
      m_pubdate = request.getParameter("pubdate");

      // Parse the m_pubdate String variable and create a Java Date object.
      Date pubDate = m_dateFormatter.parse(m_pubdate);

      // Get the root folder of the Oracle 9iFS instance.
      m_folder = ifsSession.getRootFolder();

      // Locate the selected PublicObject based on its path, starting at
```

```
  // the root folder.
  m_po = m_folder.findPublicObjectByPath(m_path);

  // If the PublicObject exists...
  if (m_po != null)
  {
    // Create an AttributeValue object for the Author String.
    AttributeValue m_avAuthor =
      AttributeValue.newAttributeValue(m_author);

    // Set the Author attribute of the PublicObject to m_avAuthor.
    m_po.setAttribute("AUTHOR",m_avAuthor);

    // Create an AttributeValue object for the Title String.
    AttributeValue m_avTitle =
      AttributeValue.newAttributeValue(m_title);

    // Set the Title attribute of the PublicObject to m_avTitle.
    m_po.setAttribute("TITLE",m_avTitle);

    // Create an AttributeValue for the Pubdate Date object.
    AttributeValue m_avPubdate =
      AttributeValue.newAttributeValue(pubDate);

    // Set the Pubdate attribute of the PublicObject to m_avPubdate.
    m_po.setAttribute("PUBDATE",m_avPubdate);
  }
}
catch (Exception e)
{
  System.out.println("Error trapped in doPost() method.");
  e.printStackTrace();
}

// Create the HTML page. Start by setting the MIME type to HTML.
response.setContentType("text/html");

// Create a new PrintWriter object that represents the output stream of
// the response object.
PrintWriter out = new PrintWriter (response.getOutputStream());

// Begin the HTML page.
out.println("<html>");

// Set its background color to white.
out.println ("<body bgcolor=WHITE>");
```

```
  // Create a Form object on the HTML page. When submitted, call the
  // doPost()method of the PropertySettingServlet. The servlet must be
  // registered on the Oracle HTTP server with the name "shortstory."
  out.println("<FORM NAME=f METHOD=POST ACTION=¥"shortstory¥">");

  // Create an HTML Text Input control named 'author', and populate it
  // with the m_author variable.
  out.println("<P>Author: <INPUT TYPE=TEXT NAME=author VALUE=¥"" +
    m_author + "¥">");
  out.println("<P>");

  // Create an HTML Text Input control named 'title', and populate it
  // with the m_title variable.
  out.println("Title: <INPUT TYPE=TEXT NAME=title VALUE=¥"" + m_title +
    "¥">");
  out.println("<P>");

  // Create an HTML Text Input control named 'pubdate', and populate it
  // with the m_pubdate variable.
  out.println("Publication Date: <INPUT TYPE=TEXT NAME=pubdate VALUE=¥""
    + m_pubdate + "¥">");
  out.println("<P>");

  // Store the path to the selected PublicObject so that it can be passed
  // to the servlet when the form is submitted.
  out.println("<INPUT TYPE=HIDDEN NAME=path VALUE=¥"" + m_path + "¥">");

  // Create a Submit button.
  out.println("<P><INPUT TYPE=SUBMIT VALUE=¥"Save¥">");

  // Complete the form.

  out.println("</FORM>");

  // Return the page to the client browser.
  out.close();
}

// getAuthorFromPO is the accessor method used to get the Author
// attribute from an existing Shortstory document, based on the selected
// PublicObject.
private String getAuthorFromPO(PublicObject po,
  LibrarySession ifsSession)
{
  // Create a string that will be used to return the value.
  String author = "";
  try
```

```
  {
    // Get the Author attribute.
    AttributeValue av = po.getAttributeByUpperCaseName("AUTHOR");

    // If it is null (i.e., the attribute doesn't exist), set the author
    // variable to an empty string. If it exists, get the AttributeValue
    // as a String value.
    author = (av == null) ? "" : av.getString(ifsSession);

    // If the attribute exists, but the value is null, set the author
    // variable to an empty string.
    if (author == null) author = "";
 }
  catch (Exception e)
  {
    e.printStackTrace();
  }

  // Return the author String variable.
  return author;

}

// getTitleFromPO is the accessor method used to get the Title attribute
// from an existing Shortstory document, based on the selected
// PublicObject.
private String getTitleFromPO(PublicObject po, LibrarySession ifsSession)
{
 // Create a string that will be used to return the value.
 String title = "";
  try
  {
    // Get the Title attribute.
    AttributeValue av = po.getAttributeByUpperCaseName("TITLE");

    // If it is null (i.e., the attribute doesn't exist), set the title
    // variable to an empty string. If it exists, get the AttributeValue
    // as a String value.
    title = (av == null) ? "" : av.getString(ifsSession);

    // If the attribute exists, but the value is null, set the title
    // variable to an empty string.
    if (title == null) title = "";
  }
  catch (Exception e)
  {
    e.printStackTrace();
```

```
    }

    // Return the title String variable.
    return title;
  }

  // getPubdateFromPO is the accessor method used to get the Pubdate
  // attribute from an existing Shortstory document, based on the selected
  // PublicObject.
  private String getPubdateFromPO(PublicObject po,
    LibrarySession ifsSession)
  {
    // Create a variable that will represent the pubdate as a String
    // object.
    String pubDateString = "";

    try
    {
      // Get the Pubdate attribute.
      AttributeValue av = po.getAttributeByUpperCaseName("PUBDATE");

      // If it is null (i.e., the attribute doesn't exist), set the pubdate
      // variable to an empty string. If it exists, get the AttributeValue
      // as a String value.
      pubDateString = (av == null) ? "" :
        m_dateFormatter.format(av.getDate(ifsSession));
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }

    // Return the pubDateString variable.
    return pubDateString;
  }

  //Get Servlet information

  public String getServletInfo()
  {
    return "PropertySettingServlet, used to set custom attributes in "
      + "files of the Document subclass SHORTSTORY.";
  }

}
```

JSP を使用したカスタム Web インタフェースの作成

この項では、Shortstory ドキュメントの属性の表示と編集に使用できる、JSP とサポート用 JavaBeans の作成プロセスについて説明します。

JSP のコンポーネント

JSP は、Java プログラミングが埋め込まれている HTML ファイルです。JSP 仕様では、Java コードを JSP ファイルの HTML および Java スクリプト・コード・セクションから分離するための標準タグ・セットが定義されています。

JSP は、サポート用 JavaBeans とともに動作するように設計されています。JSP 構文には、JavaBeans をインスタンス化できるように設計された特別なメソッドがあります。必ずしも必要ではありませんが、サポート用 JavaBeans を使用すると、Java を操作する場合に常に目標となっている再使用可能なコードを記述できます。

JSP 構文の詳細は、`java.sun.com` で Java 開発者の Web サイトを参照してください。

開発プロセス

JSP を作成するアプローチの 1 つは、Java スクリプト・コントロールが正常に表示されて動作するように、最初にフォームを HTML ページとして作成することです。次に、使用する JavaBeans メソッドのコールを追加します。最後に、必要なコールをサポートする JavaBeans を記述します。この方法は逆のように見えますが、通常は配信側から始め、JavaBeans の構築に戻って作業を進める方が簡単です。

Shortstory.htm の作成

これは、JSP の基礎となる HTML です。この時点では何の機能もありませんが、Oracle 9*i*FS からアクセスされる属性のプレースホルダを表示するコントロールを持ったフォームが描画されます。

例 12-7 Shortstory.htm

```
<HEAD>
  <TITLE>Shortstory Viewer</TITLE>
</HEAD>
<BODY BGCOLOR=WHITE>
  <H1>Shortstory</H1>
  <FORM NAME="ShortstoryForm">
    <P>
    Title: <INPUT TYPE=TEXT SIZE=100 NAME="title" VALUE= "The Title">
    <P>
    Author: <INPUT TYPE=TEXT NAME="author" VALUE= "The Author">
    <P>
    Pubdate: <INPUT TYPE=TEXT NAME="pubdate" VALUE= "The Publication Date">
```

```
    <P>
    Content:<P>
    <TEXTAREA ROWS=20 COLS=60>The whole Story.</TEXTAREA>
    <P>
    <INPUT TYPE=SUBMIT VALUE="Save">
  </FORM>
</BODY>
</HTML>
```

このファイルを保存して、HTML ブラウザで開くことができます。編集可能フィールドにはデフォルト値が表示されます。「Save」ボタンをクリックすると、この Web ページが再描画され、渡された引数がナビゲーション・バーの URL に表示されます。表示されているページは JSP ではなく、引数のハンドラを記述していないため、この時点では当然パラメータは無視されます。

Shortstory.htm から Shortstory.jsp への変換

この時点で、ページには適切なコントロールがあるため、元に戻って、Oracle 9*i*FS のコールをサポートする JavaBeans に値を渡すロジックを追加できます。コメント付きのページに続いて詳細コードのリストを示します。

例 12-8 Shortstory.jsp のコメント付きコード

```
<HTML>
<HEAD>
```

各行は、Java の IMPORT 文と等価の JSP 構文です。JSP では、ページで使用される Java コールのサポートに必要なクラスを、すべて宣言する必要があります。これらの文は ShortstoryBean をインポートします。ShortstoryBean は、格納された Shortstory ドキュメントから値を取得するための、ほとんどのアクセッサ・メソッドのサポートに使用されます。InputStream、BufferedInputStream、Document および PublicObject の各クラスは、選択した Shortstory ドキュメントのメタデータではなく内容に直接アクセスするために使用されます。

```
<%@ page import="oracle.ifs.examples.devdoc.webui.ShortstoryBean" %>
<%@ page import="java.io.InputStream" %>
<%@ page import="java.io.BufferedInputStream" %>
<%@ page import="oracle.ifs.beans.Document" %>
<%@ page import="oracle.ifs.beans.PublicObject" %>
Standard HTML tags complete the header and begin the HTML page.

<TITLE>
Shortstory Viewer
</TITLE>

</HEAD>
<BODY BGCOLOR=WHITE>
```

```
<H1>Shortstory</H1>
```

jsp:useBean タグは、ページ用にインポートされる Bean のインスタンス化に使用されます。scope は、このページの表示中にのみ Bean が使用されることを示します。「Save」をクリックしてページを送信すると、現行の Bean が破棄され、新規 Bean がインスタンス化されて、変更後の情報が表示されます。

```
<jsp:useBean id = "sb" scope = "page"
  class="oracle.ifs.examples.devdoc.webui.ShortstoryBean">
```

<% [code] %> タグは、Pure Java コードを囲みます。このタグでコードを囲むと、JSP に必要なだけ Java ロジックを直接挿入できます。

```
<%
```

次の文は、JSP からの要求および応答パラメータを渡して、ShortstoryBean（sb）インスタンスを初期化します。JSP 仕様では、サポート用 JavaBeans に空のコンストラクタが必要です。init() メソッドは、パラメータを渡して Oracle 9*i*FS への接続を作成するために使用されます。

```
sb.init(request, response);
```

次のコードは、ダミー・パラメータである action にアクセスしています。このコードは、選択したページについてフォームが初めて表示されたかどうかをテストするために使用されています。これが初めてのフォーム描画である場合、action は NULL になり、フォームはカスタム属性値の設定を試みず、リポジトリから取得します。action に値がある場合は、フォームが再描画され、フィールドの値がパラメータとして渡され、フォームはパラメータに基づいて属性を設定して表示できます。

```
String reaction = request.getParameter("action");
Initialize the variables as empty strings.

  String the_value = "";
  String title = "";
  String author = "";
  String pubdate = "";
```

reaction に値がある場合は、JSP の初めての描画ではありません。

```
if (reaction != null && reaction.equals("redraw")) {
```

フィールドの値は、要求のパラメータから抽出できます。

```
     title = request.getParameter("title");
     author = request.getParameter("author");
     pubdate = request.getParameter("pubdate");
```

渡されたパラメータに基づいて、JSP はファイル属性をその値に設定します。

```
    sb.setTitle(title);
    sb.setAuthor(author);
    sb.setPubdate(pubdate);

}
```

それ以外の場合、JSP の初めての描画です。

```
else
{
```

JSP は、Oracle 9*i*FS リポジトリに格納されているドキュメントから現在の値を要求します。これらのコールはそれぞれ、サポート用 JavaBeans 内に対応するアクセッサ・メソッド（getter）を必要とします。

```
    title = sb.getTitle();
    author = sb.getAuthor();
    pubdate = sb.getPubdate();

  }
```

JSP とサーブレット間の違いの 1 つは、JSP は Web インタフェースから Shortstory 型の全ドキュメントを直接表示するように登録でき、サーバーに渡される URL にドキュメントのパスを埋め込む必要がないことです。この方法でページを登録する場合は、JSP 自体の中でドキュメントの内容を表示するメソッドを考慮する必要があります。この場合は、JavaBeans からアクセッサ・メソッドをコールするのではなく、ドキュメントの内容へのアクセスに使用するコードを JSP に組み込みます。

```
  String theStory = "";
  InputStream is = null;
  PublicObject m_po = sb.getPo();
  try {
    Document d = (Document) m_po;
    is = d.getContentStream();

    BufferedInputStream bis = new BufferedInputStream(is);
    boolean eof = false;
    while (!eof)
    {
      int c = bis.read();
      if (c == -1)
      {
        eof = true;
      }
```

```
        else
        {
          theStory += (char) c;
        }
      }
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
%>
```

これで、このフォームの Java 処理は完了です。次のタスクは、Java コードで設定した属性を使用してフォームを描画することです。

まず、属性値の表示と変更に使用するフォームを作成します。ユーザーがフォームを送信すると、フォーム ACTION でそれ自体がコールし、変更後の値を使用して再描画されます。

```
<FORM NAME="ShortstoryForm" ACTION= "/shortstory.jsp">
```

次の各行では、INPUT コントロールをテキスト・フィールドとして作成しています。各フィールドには、Oracle 9*i*FS に格納された値から設定された変数（フォームの初回描画時）、または要求のパラメータを通じて渡されたパラメータから設定された変数（フォームの 2 度目以降の描画時）が移入されます。Java コードで設定された変数は、JSP 式のタグ（<%= %>）で囲まれ、その値が HTML ページに表示されます。

```
<P>
Title: <INPUT TYPE=TEXT SIZE=100 NAME="title" VALUE= "<%= title %>">
<P>
Author: <INPUT TYPE=TEXT NAME="author" VALUE= "<%= author %>">
<P>
Pubdate: <INPUT TYPE=TEXT NAME="pubdate" VALUE= "<%= pubdate %>">
<P>
```

ドキュメントの内容は、スクロール・テキスト・フィールドに表示されます。この情報は読取り専用です。

```
Content (Read Only):<P>
<TEXTAREA ROWS=20 COLS=60>
<%= theStory %>
</TEXTAREA>
```

JSP は、ファイルへのパスを獲得して非表示フィールドに格納する必要があります。次のコードは、JSP でドキュメントの格納場所が認識されるように、フォームが再描画されるときにパスを JSP に渡します。

```
<INPUT TYPE=HIDDEN NAME="path" VALUE = <%= sb.getPath()%>>
```

この指示により、フォームが 1 回以上表示されたことがあることを示すために、ダミーのアクション・パラメータが移入されます。以降の再描画では、カスタム属性が Oracle 9*i*FS リポジトリ内の値ではなくフォーム上の値に基づいて更新され、表示されます。

```
<INPUT TYPE=HIDDEN NAME="action" VALUE="redraw">
A Submit button (Input control) completes the form.

<P>
<INPUT TYPE=SUBMIT VALUE="Save">
</FORM>
```

次のタグは、<jsp:useBean> タグの適用範囲の終わりを示します。useBean タグ・セットでは、その変数を使用するコード・フラグメント全体を囲む必要があります。

```
</jsp:useBean>
```

標準 HTML タグは、JSP を完了させます。

```
</BODY>
</HTML>
```

例 12-9 に、作成した JSP の詳細コードをコメントなしで示します。

例 12-9 Shortstory.jsp のコード

```
<HTML>
<HEAD>
<%@ page import="oracle.ifs.examples.devdoc.webui.ShortstoryBean" %>
<%@ page import="java.io.InputStream" %>
<%@ page import="java.io.BufferedInputStream" %>
<%@ page import="oracle.ifs.beans.Document" %>
<%@ page import="oracle.ifs.beans.PublicObject" %>
<TITLE>
Shortstory Viewer
</TITLE>
</HEAD>
<BODY BGCOLOR=WHITE>
<H1>Shortstory</H1>
<jsp:useBean id = "sb" scope = "page"
   class="oracle.ifs.examples.devdoc.webui.ShortstoryBean">
<%
   sb.init(request, response);
   String reaction = request.getParameter("action");
   String the_value = "";
   String title = "";
   String author = "";
   String pubdate = "";
   if (reaction != null && reaction.equals("redraw")) {
```

```
      title = request.getParameter("title");
      author = request.getParameter("author");
      pubdate = request.getParameter("pubdate");
      sb.setTitle(title);
      sb.setAuthor(author);
      sb.setPubdate(pubdate);
    }
    else
    {
      title = sb.getTitle();
      author = sb.getAuthor();
      pubdate = sb.getPubdate();
    }
    String theStory = "";
    InputStream is = null;
    PublicObject m_po = sb.getPo();
    try {
      Document d = (Document) m_po;
         BufferedInputStream bis = new BufferedInputStream(is);
      boolean eof = false;
      while (!eof)
      {
        int c = bis.read();
        if (c == -1)
        {
          eof = true;
        }
        else
        {
          theStory += (char) c;
        }
      }
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
%>
<FORM NAME="ShortstoryForm" ACTION= "./shortstory.jsp">
<P>
Title: <INPUT TYPE=TEXT SIZE=100 NAME="title" VALUE= "<%= title %>">
<P>
Author: <INPUT TYPE=TEXT NAME="author" VALUE= "<%= author %>">
<P>
Pubdate: <INPUT TYPE=TEXT NAME="pubdate" VALUE= "<%= pubdate %>">
<P>
Content (Read Only):<P>
```

```
<TEXTAREA ROWS=20 COLS=60>
<%= theStory %>
</TEXTAREA>
<INPUT TYPE=HIDDEN NAME="path" VALUE = <%= sb.getPath()%>>
<INPUT TYPE=HIDDEN NAME="action" VALUE="redraw">
<P>
<INPUT TYPE=SUBMIT VALUE="Save">
</FORM>
</jsp:useBean>
</BODY>
</HTML>
```

次のステップは、Shortstory サブクラスのカスタム属性にアクセスして変更するサポート用メソッドを使用して、JavaBeans を作成することです。

サポート用 JavaBeans の作成

`Shortstory.jsp` のサポートに使用する JavaBeans では、次の基本タスクを実行する必要があります。

1. Oracle 9*i*FS への接続を確立します。
2. Oracle 9*i*FS の Shortstory サブクラスのインスタンスから格納されている値を取得するアクセッサ・メソッド（getter）を提供します。
3. JSP に表示される値の変更内容を格納するミューテータ・メソッド（setter）を提供します。

例 12-10 に、`ShortstoryBean.java` の詳細コードをコメント付きで示します。

例 12-10 ShortstoryBean.java

Shortstory サブクラスに対応付けられているパッケージの Java クラスは、oracle.ifs.examples.devdoc.webui パッケージに格納されています。

```
package oracle.ifs.examples.devdoc.webui;
```

SimpleDateFormat は、特定の日付書式と一致するテキスト文字列を Java の Date オブジェクトに変換し、Date オブジェクトを Java から書式付きテキスト文字列に変換するために使用されます。

```
import java.text.SimpleDateFormat;
import java.util.Date;
```

次のクラスは JSP に必須であり、重要な要求および応答オブジェクトがすべて用意されており、クライアントとサーバー間での情報の送受信に使用されます。

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

このクラスは、この JavaBeans では使用されませんが、JavaBeans は IfsHttpLogin Interface を拡張し、実装方法の 1 つにはクラス HttpSessionBindingEvent が必要です。

```
import javax.servlet.http.HttpSessionBindingEvent;
```

これは、多数の共通するファイル I/O タスクをアクセスしやすいようにカプセル化して公開する便利なクラスです。

```
import oracle.ifs.adk.filesystem.IfsFileSystem;
```

次の各クラスには、既存の Oracle 9*i*FS LibrarySession を使用してサーバーにログインする便利な方法を提供します。

```
import oracle.ifs.adk.http.HttpUtils;
import oracle.ifs.adk.security.IfsHttpLogin;
```

Document クラスは、Shortstory サブクラスの親です。このクラスは、実行時に選択されたドキュメントをインスタンス化し、そのドキュメントの内容にアクセスするために使用されます。

```
import oracle.ifs.beans.Document;
```

次のクラスは使用されませんが、IfsHttpLogin インタフェースを実装するためにコードでオーバーライドする必要のあるメソッドの 1 つに必要です。

```
import oracle.ifs.beans.FolderPathResolver;
```

次の各クラスは、クライアントからサーバーへの接続を実行時にインスタンス化するために使用されます。

```
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.LibraryService;
```

PublicObject は、Oracle 9*i*FS でフォルダに格納できるアイテムです。これは、実行時に選択されたドキュメントを表すために使用されます。

```
import oracle.ifs.beans.PublicObject;
```

AttributeValue クラスは、Java の値とデータベースへの格納に適した値との間の変換を処理するハンドラです。

```
import oracle.ifs.common.AttributeValue;
```

IfsException は、Oracle 9*i*FS エラーを獲得してレポートするために使用されます。

```
import oracle.ifs.common.IfsException;
```

JavaBeans は、IfsHttpLogin を実装することで Oracle 9*i*FS サーバーへの接続に必要なメソッドを継承します。これは、サーバーへの接続方法の 1 つにすぎません。

```
public class ShortstoryBean implements IfsHttpLogin
{
```

最初に、次のコードは Oracle 9*i*FS への接続に使用するグローバル変数を初期化します。

```
  private LibrarySession m_session;
  private FolderPathResolver m_resolver;
  private IfsFileSystem m_ifs;
```

次の行は、選択されたドキュメントへの完全修飾パスを格納するためのグローバル変数を初期化します。

```
  private String m_path;
```

m_po グローバル変数は、実行時に選択されたドキュメント（PublicObject）を表すために使用されます。

```
  private PublicObject m_po;
```

次の文は、Oracle 9*i*FS との間でやりとりされる日付値のフォーマットに使用される SimpleDateFormat オブジェクトを作成します。大文字 M が重要で、これは月を表し、小文字の m は分を表すため注意してください。

```
private SimpleDateFormat m_dateFormatter =
    new SimpleDateFormat("MM-dd-yy");
```

JavaServer Pages 仕様では、引数を取らないコンストラクタが必要です。

```
 // constructor

  public void ShortstoryBean()
    throws IfsException
  {
  }
```

init メソッドは、他の場合にはコンストラクタで処理されるファンクションを実行します。これは、Oracle 9*i*FS の作業セッションの作成と、選択された Document の検索に使用されます。

```
  public void init(HttpServletRequest request,
    HttpServletResponse response)
   {
```

次の行は、Oracle 9*i*FS のすべての Java コードに含める必要があります。Oracle 9*i*FS のエラー・メッセージはネストされ、特定のクラスでエラー条件が発生すると、そのクラスよりクラス階層内で上位のクラスに最も役立つエラー・メッセージが発生します。冗長メッセージ機能をオンにすると、すべての例外セットがレポートされ、より意味のあるフィードバックを使用してデバッグできます。

```
IfsException.setVerboseMessage(true);
```

次のコマンドは、JSP をコールしたセッションから取り込まれたユーザーのログイン情報を含むオブジェクトを作成します。

```
try {
     IfsHttpLogin ifsLogin =
       (IfsHttpLogin) request.getSession(true).getValue("IfsHttpLogin");
```

このログイン情報に基づいて、次の文がこの JSP トランザクション用の新規セッションを作成します。

```
// Create an object representing the current work session.

   m_session = ifsLogin.getSession();

   m_ifs = new IfsFileSystem(m_session);
```

次は、JSP の要求パラメータから選択されたドキュメントへのパスを取得するためにのみ使用される、HttpUtils のコールです。

```
   m_path = HttpUtils.getIfsPathFromJSPRedirect(request);
```

このパスに基づいて、次の文は選択されたドキュメントのランタイム・バージョンを表す PublicObject をインスタンス化します。この PublicObject はメモリーにのみ常駐し、トランザクションの存続期間中にのみ存在します。これは、ドキュメントの永続的な値にアクセスして更新するために使用されます。この値は、Oracle 9*i*FS リポジトリに格納されています。

```
   m_po = m_ifs.findPublicObjectByPath(m_path);

   } catch (Exception e)
   {
     e.printStackTrace();
   }
 }
```

次のメソッドを使用すると、ユーザーの現行の LibrarySession にアクセスできます。

```
/**
 * Return the login's session
 *
 * @pub
 */
public LibrarySession getSession()
{
  return m_session;
}
```

getPath() メソッドは、ドキュメントの内容にアクセスするために JSP 自体からコールされます。

```
public String getPath()
{
  return m_path;
}
```

次の各メソッドは、IfsHttpLogin クラスを実装するために必要ですが、このアプリケーションでは使用していません。

```
/**
 * Return the login's path resolver
 *
    */
public FolderPathResolver getResolver()
{
  return m_resolver;
}

/**
 * Return the IfsFileSystem API object
 *
 */
public IfsFileSystem getIfsFileSystem()
{
  return m_ifs;
}

/**
 * Implementation of valueBound
 *
 */
public void valueBound(HttpSessionBindingEvent e)
{
}

/**
 * Implementation of valueUnbound
 *
 */
public void valueUnbound(HttpSessionBindingEvent e)
{
  try
  {
    m_session.disconnect();
  }
  catch (IfsException ie)
```

```
    {
    }
  }
```

次に、Shortstory サブクラスのドキュメントのカスタム属性を取得して変更するために使用する、アクセッサ・メソッド（getter）とミューテータ・メソッド（setter）を示します。

getTitle() メソッドは、IfsFileSystem ヘルパー・クラスに用意されている getAttribute() メソッドを使用します。1 回のコールで、現行の PublicObject の Title 属性を取得して文字列に変換します。このため、最初に AttributeValue を取得してから文字列値に変更するという中間ステップが不要になります。

```
public String getTitle(){
    String title = "";
    try
    {
      title =  m_ifs.getAttribute(m_po,"Title").toString();
    } catch (Exception e)
    {
      e.printStackTrace();
    }
    return title;
  }
```

getAuthor() の動作は getTitle() メソッドと同じです。

```
  public String getAuthor(){
    String author = "";
    try
    {
      author =  m_ifs.getAttribute(m_po,"AUTHOR").toString();
    } catch (Exception e)
    {
      e.printStackTrace();
    }
    return author;
  }
```

pubDate は、Date オブジェクトとして格納されます。このメソッドは、Date オブジェクトを JavaServer Pages に渡す前に、文字列に変換する必要があります。JSP は、String 値のみをパラメータとして送受信します。

```
  public String getPubdate()
  {
    Date pubDate = null;
    String pubDateString = "";
```

最初に、AtributeValue オブジェクトを作成して PUBDATE 属性を移入します。

```
    try
    {
      AttributeValue av = m_ifs.getAttribute(m_po,"PUBDATE");
```

pubDate 変数には、AttributeValue から値が与えられます。

```
      pubDate = av.getDate(m_session);
```

次は、値が NULL でないことを保証するための三項引数です。値が存在する場合、前に作成された SimpleDateFormat オブジェクトは日付値に MM-dd-yy 書式を適用して文字列に変換します。

```
      pubDateString = (av == null) ? "" : m_dateFormatter.format(pubDate);

    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
    return pubDateString;
  }
```

次はミューテータ・メソッド（setter）です。この場合は、String を AttributeValue に変換するメソッドに文字列変数を渡しています。AttributeValue は、PublicObject に値を設定するために使用されます。

```
  public void setTitle(String sv)
  {
    try
    {
      AttributeValue sv_av = AttributeValue.newAttributeValue(sv);
      m_po.setAttribute("TITLE", sv_av);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  public void setAuthor(String sv) {
    try
    {
      AttributeValue sv_av = AttributeValue.newAttributeValue(sv);
      m_po.setAttribute("AUTHOR", sv_av);
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
```

ここでは、SimpleDateFormat オブジェクトは、parse() メソッドを使用して日付の String 表現を Java の Date オブジェクトに変換します。

```
  public void setPubdate(String sv) {
    try
    {
      Date pubdate = m_dateFormatter.parse(sv);

      AttributeValue sv_av = AttributeValue.newAttributeValue(pubdate);
      m_po.setAttribute("PUBDATE", sv_av);
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

JavaServer Pages の配置

これで JavaServer Pages（`shortstory.jsp`）と JavaBeans（`ShortstoryBean.java`）が作成されたため、次のステップはそれぞれを Oracle 9*i*FS にアップロードし、.shortstory ファイル拡張子が付いたファイルの表示に使用されるように JSP を登録することです。

Oracle 9*i*FS への JSP のアップロード

サーブレットはサーバー固有のファイル・システムに配置されますが、JSP は Oracle 9*i*FS 階層内のディレクトリに配置する必要があります。そのために `/ifs/jsp-bin` ディレクトリが用意されています。

JSP をアップロードする手順は、次のとおりです。

1. 管理権限を持つユーザーで、必要なプロトコルを通じて Oracle 9*i*FS にログインします。
2. `/ifs/jsp-bin` ディレクトリに移動します。
3. `shortstory.jsp` を `/ifs/jsp-bin` ディレクトリにコピーします。

JavaBeans のアップロード

サポート用 JavaBeans を、Oracle 9*i*FS のインスタンス用の IFS_CLASSPATH に指定されている位置に配置する必要があります。

そのために、`$ORACLE_HOME/9ifs/custom_classes` ディレクトリが用意されています。開発環境が Oracle 9*i*FS インスタンスと同じマシンにある場合は、クラスをそのフォルダに直接コンパイルできます。

ShortstoryBean.class をアップロードする手順は、次のとおりです。

1. `ShortstoryBean.java` をコンパイルします。
2. 必要な場合は、コンパイル後のディレクトリ構造全体（`/oracle/ifs/examples/devdoc/webui`）を、Oracle 9*i*FS インスタンス固有のファイル・システムの `$ORACLE_HOME/9ifs/custom_classes` ディレクトリにコピーします。

   結果的なパスは次のようになります。

   ```
   $ORACLE_HOME/9ifs/custom_classes/oracle/ifs/examples/devdoc/webui/
   ShortstoryBean.class
   ```

JSP の登録

最後のステップは、JSP を Oracle 9*i*FS に登録することです。そのためには、XML 構成ファイルを使用する方法と、Oracle 9*i*FS Manager アプリケーションを使用する方法があります。

Oracle 9*i*FS Manager アプリケーションを使用して JSP を登録する手順は、次のとおりです。

1. Oracle 9*i*FS Manager アプリケーションを起動して、管理者でログインします。
2. 「Object」->「Register...」を選択します。
3. 「Select Object Type」ダイアログ・ボックスで、「Java Server Page (JSP) Lookup...」を選択します。
4. 「Register」をクリックします。
5. 「Java Server Page (JSP) Lookup Registry」ウィンドウで「Add」をクリックします。
6. 「Classname」ポップアップ・メニューから「Shortstory」を選択します。
7. 「Mimetype」ポップアップ・メニューから「*/*」を選択します。
8. JSP へのフルパス名を次のように入力します。

   ```
   /ifs/jsp-bin/shortstory.jsp
   ```

9. 「OK」をクリックして「Java Server Page Lookup Entry」ウィンドウを閉じます。
10. 「OK」をクリックして「Java Server Page (JSP) Registry」ウィンドウを閉じます。

XML 構成ファイルを使用して JSP を登録する手順は、次のとおりです。

1. 次の XML 構成ファイルを作成します。

```
<?xml version="1.0" standalone="yes"?>
<!--RegisterJSP.xml-->
<PROPERTYBUNDLE>
   <UPDATE RefType="valuedefault">JspLookup</UPDATE>
   <PROPERTIES>
      <PROPERTY ACTION="add">
         <NAME>SHORTSTORY.*/*</NAME>
         <VALUE DataType="String">/ifs/jsp-bin/shortstory.jsp</VALUE>
      </PROPERTY>
   </PROPERTIES>
</PROPERTYBUNDLE>
```

2. このファイルに拡張子 .xml を付けて保存します（推奨名は `RegisterJSP.xml` です）。
3. 必要なプロトコルを通じて管理ユーザーでログインします。
4. ファイルを Oracle 9*i*FS ディレクトリ階層の必要な位置にアップロードします。Web インタフェースを使用している場合は、「Parse File on Upload」チェックボックスをオンにしてください。

JavaServer Pages のテスト

これで JSP アプリケーションが完成しました。JSP をテストするには、Web インタフェースでサンプル・ファイル `sample.shortstory` を作成したディレクトリに移動します。ファイルをクリックして開きます。ファイルが、プレーン・テキスト・ファイルとしてではなく JSP に表示されます。値を変更し、変更結果を保存できます。

13

カスタム・サーバーの作成

Oracle 9*i*FS では、付属のプロトコル・サーバーやエージェントと同様に管理できるカスタム・プロトコル・サーバーおよびエージェントを構築できます。この章では、カスタム・プロトコル・サーバーおよびエージェントの作成、テストおよび配置プロセスについて説明します。

この章の内容は、次のとおりです。

サーバーの概要

Oracle 9*i*FS サーバーは、Oracle 9*i*FS アプリケーション開発 API を使用し、Oracle 9*i*FS ノードで動作する Java アプリケーションです。

すべての Oracle 9*i*FS アプリケーションがサーバーであるとはかぎりません。アプリケーションがサーバーとして機能するには、この章で説明するように、規定されたパターンに従う必要があります。このパターンに従って、ノード上で実行でき、Oracle 9*i*FS 管理ツールを使用して監視および管理できるアプリケーションを作成します。これには、次のようなメリットがあります。

- 管理者は、Oracle 9*i*FS に付属のサーバーに使用するのと同じツールでカスタム・サーバーを監視し、管理できます。新しいユーザー・インタフェースについて学習する必要はなく、一箇所から管理できます。
- サーバー管理用に独自のユーザー・インタフェースを作成する必要がなく、アプリケーションの開発期間が短縮されます。
- サーバーはノード上で実行され、Oracle 9*i*FS に付属のものなど、他のサーバーと同じ Oracle 9*i*FS サービスを使用できます。このため、本番環境でのプロセス数、ノードを実行するコンピュータのメモリー要件およびデータベース接続数が減少します。
- Oracle には、カスタム・サーバーを簡単に開発できるようにツール・セットが用意されており、アプリケーションの開発期間がさらに短縮されます。この章では、このようなツールについて説明します。

ただし、すべての Oracle 9*i*FS アプリケーションをサーバーにする必要はありません。サーバー・パターンに適しているとは言えないアプリケーションの例を次に示します。

- コマンドライン引数やパラメータ・ファイルで指定されたアクションを実行してから終了する、コマンドライン・ユーティリティ。このように実行時間の短いアプリケーションをノードで実行するのは不便です。
- Java AWT または Swing ライブラリを使用して Graphical User Interface（GUI）を表示するアプリケーション。Oracle 9*i*FS ノードは中間層プロセスとして動作するため、GUI は不要です。

プロトコル・サーバーおよびエージェント

サーバーをプロトコル・サーバーまたはエージェントとして分類すると便利です。

- プロトコル・サーバーは、別のプロセス（通常は別のコンピュータ）で動作するクライアント・アプリケーションと、相互のプロトコルに基づいて対話します。たとえば、Oracle 9*i*FS FTP サーバーは、RFC 959 で指定された FTP プロトコルを使用して FTP クライアントと対話します。代表的なプロトコル・サーバーは、TCP/IP サーバー・ソケットをオープンし、クライアントからの接続を受け入れ、クライアント要求に対する応答時にプロトコルで定義された操作を実行します。

- エージェントは無人で実行され、規定のなんらかの操作を定期的に、またはどこかで発生したイベントに応答して実行します。たとえば、Oracle 9*i*FS Garbage Collection Agent は事前定義済みの間隔で起動し、Oracle 9*i*FS リポジトリに対して一連のクリーンアップ・タスクを実行します。Oracle 9*i*FS Quota Agent は定期的に、およびイベントへの応答時に操作を実行します。ユーザーが Oracle 9*i*FS で内容を作成、変更または削除すると生成されるイベントに応答して、各ユーザーが使用した記憶領域の見積りを更新します。このような見積りを常に正確に行うために、ユーザーの実際の記憶領域を定期的に再計算します。

プロトコル・サーバーおよびエージェントは異なる機能を持っていますが、カスタム・プロトコル・サーバーおよびカスタム・エージェントの作成プロセスはほぼ同じです。この章では、それぞれの例を挙げて説明します。

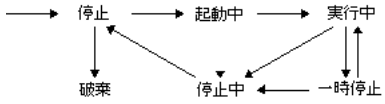
サーバーの状態

Oracle 9*i*FS サーバーはシンプル・ステート・マシンです。サーバーの状態は常に次の 6 つのいずれかです。

- **停止：** サーバーがノードにロードされていますが、現在は実行されていません。これはサーバーの初期状態です。停止状態のサーバーは、何も操作を実行しません。たとえば、停止状態のプロトコル・サーバーは、クライアント接続を持ちません。
- **起動中：** サーバーは実行開始を要求されましたが、まだ実行中にはなっていません。
- **実行中：** サーバーは実行中です。これがサーバーの通常の操作状態です。たとえば、実行中のプロトコル・サーバーは、既存のクライアント接続をサービスして新規接続を受け入れます。
- **一時停止：** サーバーは実行中でしたが一時停止されています。一時停止中のサーバーの動作は、サーバー定義です。一般に、一時停止中のサーバーは、すべての通常操作の実行を停止するか、通常操作のサブセットのみを実行します。たとえば、一時停止中のプロトコル・サーバーは、既存のクライアント接続を引き続きサービスできますが、新規接続は受け入れません。
- **停止中：** サーバーは実行停止を要求されましたが、まだ完全には停止していません。
- **破棄：** サーバーはノードからアンロードされています。これはサーバーの終了状態です。

図 13-1「サーバーの状態遷移」に、可能な状態遷移を示します。破線はエラー条件を示しています。

図 13-1 サーバーの状態遷移



サーバー管理

サーバーには、oracle.ifs.management.domain.ServerInterface インタフェースが実装されます。このインタフェースは、6 つの可能なサーバー状態を列挙し、Oracle 9*i*FS ノードでサーバーの管理に使用するメソッドのセットを宣言します。この種の ServerInterface メソッドのいくつかは、サーバーの状態遷移を要求します。

- start(): サーバーに対して起動を要求し、サーバーの状態を停止から起動中に変更し、起動中から実行中に変更するように要求します。
- stop(): サーバーに対して停止を要求し、サーバーの状態を実行中または一時停止から停止中に変更し、停止中から停止に変更します。
- suspend(): サーバーに対して操作の一時停止を要求し、サーバーの状態を実行中から一時停止に変更します。
- resume(): サーバーに対して操作の再開を要求し、サーバーの状態を一時停止から実行中に変更します。
- dispose(): サーバーに対してそれ自体をノードからアンロードするように要求し、サーバーの状態を停止から破棄に変更します。

カスタム・サーバーの作成

この項では、カスタム・プロトコル・サーバーまたはエージェントの作成方法について説明します。最初にサーバー用の新規 Java クラスを作成する方法を説明してから、プロトコル・サーバーおよびエージェントに通常必要な機能を提供する方法を説明します。

サーバー・クラスの作成

Oracle 9*i*FS サーバーは、管理 ServerInterface を実装する oracle.ifs.management.domain.Server のサブクラスです。新規サーバーを作成するには、Server の具体的なサブクラスを記述できます。ただし、より簡単なアプローチは、それ自体が Server のサブクラスである oracle.ifs.management.domain.IfsServer をサブクラス化することです。IfsServer には、Server により宣言される多数の抽象メソッドのデフォルト実装が用意されており、記述する必要のあるコードの量が減少します。

これ以降は、新規プロトコル・サーバーまたはエージェントを作成するために IfsServer をサブクラス化するものと想定します。Server の詳細は、そのクラスの Javadoc を参照してください。

例 13-1 FingerServer

FingerServer は、Finger 形式のプロトコル（RFC 1288 に類似）を提供する単純なプロトコル・サーバーです。そのソース・コードは、カスタム・プロトコル・サーバーの構築方法を示す例として Oracle 9*i*FS とともに配布されます。

FingerServer のメイン・クラス oracle.ifs.examples.servers.FingerServer は、IfsServer を拡張します。

```
public class FingerServer
    extends IfsServer
```

サーバーは、IfsException の発生を宣言する、引数を取らないコンストラクタを提供する必要があります。

```
public FingerServer()
    throws IfsException
```

通常、サーバーのコンストラクタが実行する操作はごくわずかです。コンストラクタが正常に終了した場合、サーバーは起動していないか、または完全に初期化されていません。

サーバーおよびスレッド・セーフティ

プロトコル・サーバーおよびエージェントには、通常は複数の同時スレッドがあります。たとえば、プロトコル・サーバーは次のスレッドを持つ場合があります。

- 新規クライアント接続の待機および作成： たとえば、スレッドは java.net.ServerSocket の accept() メソッドでブロックすることがあります。
- 既存接続のサービス： たとえば、スレッドはソケットの入力ストリームでクライアントの要求を待機し、その要求を処理し、ソケットの出力ストリームで応答を送信し、クライアントからの次の要求を待機することがあります。
- サーバーの状態の監視および管理。

エージェントは次のスレッドを持つ場合があります。

- 定期的なタスクの実行。
- エージェントに重要なイベントの受信。
- エージェントの状態の監視および管理。

このようなスレッドに加えて、サーバーは、start() および stop() など、Oracle 9*i*FS 管理ツールが非同期で行う ServerInterface メソッドのコールに応答する必要があります。

サーバーの操作の実行中に、そのスレッドが Oracle 9*i*FS セッションまたはサーバーが保持する他の状態を共同で操作できます。サーバーが正常に動作するには、これがスレッド・セーフな方法で行われる必要があります。

`IfsServer` には、スレッド・セーフ・サーバーの開発作業を簡素化するツールが用意されています。このため、複数のスレッドで動作する timer および `ServerInterface` メソッドは、サーバー別の要求キューに要求をエンキューできます。同様に、着信イベントもサーバー別のイベント・キューにエンキューできます。このような要求とイベントは、後でサーバーの単一の管理スレッドでデキューして処理できます。このような要求とイベントを単一スレッドに委譲することで、処理のレース条件が防止されます。

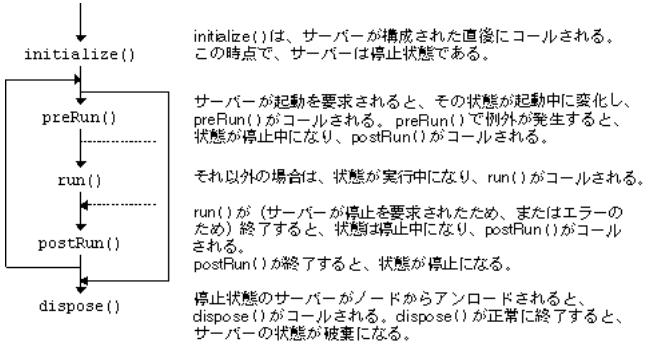
サーバーのライフサイクル

サーバーのライフサイクル中に、ノードにより次の 5 つのメソッドがコールされます。

- initialize()
- preRun()
- run()
- postRun()
- dispose()

図 13-2「サーバーのライフサイクル」に、ノードがこの 5 つのライフサイクル・メソッドをコールする順序と、各メソッドとサーバーの状態の関係を示します。

図 13-2 サーバーのライフサイクル



プロトコル・サーバーまたはエージェントでは、この 5 つのメソッドを実装して処理を実行できます。`run()` メソッドは必ず実装する必要があります。他の 4 つのメソッドのデフォルト実装は、`IfsServer` により提供されます。

initialize() メソッドは、ノードへのロード時にサーバーが構成された直後に、1 度のみコールされます。サーバーが起動するたびに preRun() がコールされ、サーバー固有の初期化を実行します。preRun() が正常に終了すると、run() がコールされてサーバーの操作を実行します。run() メソッドは、サーバーが停止を要求されるか、停止せずにはリカバリできないエラーが発生するまで続行されます。run() が終了すると、postRun() がコールされてクリーンアップを実行します。postRun() が正常に終了すると、サーバーが停止します。サーバーは、再起動するかノードからアンロードされるまで、停止状態になっています(再起動時には、初回と同様に preRun()、run() および postRun() が順番にコールされます)。サーバーがノードからアンロードされると、dispose() がコールされて最終的なクリーンアップを実行します。

IfsServer サブクラスの場合、run() はサーバーの管理スレッドを実行し、他のスレッドでキューに入れられた要求とイベントを処理する必要があります。そのためには、例 13-2「管理スレッドでの run メソッドの実装」のように run() を実装します。

例 13-2 管理スレッドでの run メソッドの実装

```
public void run()
{
    //
    // The run method should return when stopRequested returns
    // true, or when an exception is thrown from which the
    // IfsServer cannot recover.  This code is boilerplate.
    //

    while (!stopRequested())
    {
        try
        {
            handleRequests();
            processEvents();
            waitServer();
        }
        catch (Throwable t)
        {
            //
            // The run method is not allowed to throw an unchecked
            // exception, so if something goes wrong, just log it.
            // Notice that we catch Throwable, not Exception, so
            // that if an Error occurs, it gets logged.
            //

            log(LEVEL_LOW, t);
        }
    }
}
```

FingerServer の run() メソッドは、このテンプレートと同じです。

サーバーの実行環境

ノードは、サーバーのロード時にサーバーに次の情報を提供します。

- サーバー名
- サーバー構成パラメータ
- サーバーの操作対象となるサービスの名前

IfsServer には、サーバーがこの情報や他の情報にアクセスできるように、次のメソッドが用意されています。

- getName() はサーバー名を戻します。
- getParameterTable() は、サーバー構成パラメータを構成する名前 / 値のペアを戻します。
- getServiceName() はサーバーの操作対象となるサービスの名前を戻し、getService() は LibraryService 自体を戻します。
- getStatus() は、サーバーの状態を表す整数値を戻します。状態値は、ServerInterface に定数で列挙されます。静的 toStatusLabel メソッドは、整数の状態値をローカライズされた文字列に変換します。
- log メソッドでは、文字列と Throwable をノードのログ・ファイルに記録できます。これらのメソッドは、メッセージのログ・レベルを指定する整数の引数を取ります。有効なログ・レベルは、Server に定数で列挙されます。ノードのログ・レベルがメッセージのログ・レベルより低い場合、メッセージはログに記録されません。

セッション管理

サーバーでは、通常の方法で Oracle 9*i*FS セッションを作成して使用できます。ただし、サーバーにより独自のサービスが起動されることはありません。かわりに、getService() および getServiceName() メソッドで指定されたサービスを使用する必要があります。適切に構成されたノードでは、このサービスがサーバーのロード前に起動します。例 13-3 に、このサービスに対するセッションの作成方法を示します。

例 13-3 既存のサービスに対するセッションの作成

```
LibraryService service = getService();

CleartextCredential credential =
    new CleartextCredential(username, password);

// The application name is displayed in Oracle 9iFS Manager.
// Setting it is optional, but allows administrators to see
```

```
// which servers created which sessions.
String serverName = getName();
ConnectOptions options = new ConnectOptions();
options.setApplicationName(serverName);

LibrarySession session = service.connect(credential, options);
```

サーバーは、停止時に、作成したセッションがすべて切断されたことを保証する必要があります。

システム・セッション

サーバーは、通常、システム・セッションを使用して特定の操作を実行します。システム・セッションは、サーバーが起動時に、ユーザーによる介入が行われる前に作成するセッションです。IfsServer には、デフォルト・セッションと呼ばれる単一のシステム・セッションを作成して管理するためのメソッドが用意されています。

- connectSession() は、サーバーのデフォルト・セッションを作成して戻します。サーバーに（このメソッドが以前にコールされているために）すでにデフォルト・セッションがある場合は、そのセッションが戻されます。
- disconnectSession() は、サーバーのデフォルト・セッションを切断します。
- getSession() はサーバーのデフォルト・セッションを戻し、デフォルト・セッションがない場合は NULL を戻します。
- checkSession() はサーバーのデフォルト・セッションを戻し、追加のチェックを実行して、セッションが引き続き接続状態で有効であるかどうかを確認します。
- デフォルト・セッションの作成時に、IfsServer は次の 5 つのサーバー構成パラメータを使用します。
  - IFS.SERVER.SESSION.User: ユーザー名。これは必須です。
  - IFS.SERVER.SESSION.ApplicationName: ConnectOptions アプリケーション名。オプションです。デフォルトでサーバー名に設定されます。
  - IFS.SERVER.SESSION.LOCALE.Language: ConnectOptions のロケール言語。オプションです。
  - IFS.SERVER.SESSION.LOCALE.Country: ConnectOptions のロケールの国。オプションです。
  - IFS.SERVER.SESSION.LOCALE.Variant: ConnectOptions のロケール可変。オプションです。

ゲスト・セッション

プロトコル・サーバーでは、通常、ユーザー認証を必要としない無名アクセス、つまりゲスト・アクセスが許可されます。IfsServer では、サーバーはパスワードなしのゲスト・セッションを作成できます。そのためには、getCredential(String) メソッドを使用して資格証明を取得します。例 13-4 に、このテクニックを示します。

例 13-4 ユーザー名 Guest 用のセッションの作成

```
LibraryService service = getService();
Credential credential = getCredential("guest");

// The application name is displayed in Oracle 9iFS Manager.
// Setting it is optional, but allows administrators to see
// which servers created which sessions.
String serverName = getName();
ConnectOptions options = new ConnectOptions();
options.setApplicationName(serverName);

LibrarySession session = service.connect(credential, options);
```

指定された Oracle 9*i*FS ユーザー用の有効な資格証明を作成できるため、サーバーでは getCredential(String) メソッドを慎重に使用する必要があります。

要求の処理

サーバーの起動と停止

サーバーの起動時には、ノードにより preRun() メソッドがコールされます。このメソッドをオーバーライドして、サーバーに適した初期化を実行できます。たとえば、このメソッドはシステム・セッションを作成し、プロトコル・サーバーのサーバー・ソケットをオープンできます。preRun() が例外を発生させずに正常に終了すると、run() が自動的にコールされ、サーバーに対して通常の操作を実行するように指示します。

run() メソッドは、サーバーが停止を要求されるか、リカバリできないエラーが発生するまで続行する必要があります。前述の「サーバーのライフサイクル」にコードを示した標準的な run() ループは、stopRequested() をコールしてサーバーの停止が必要かどうかをチェックします。

サーバーが停止を要求されると、handleStopRequest() がコールされます。デフォルトでは、このメソッドは単に後続の stopRequested() のコールで TRUE を戻させます。handleStopRequest() をオーバーライドすると、停止要求を条件付きで禁止できます。

run() が終了すると、正常に終了したか例外が発生したかに関係なく、postRun() がコールされます。また、preRun() で例外が発生した場合も、postRun() メソッドがコールされます。この postRun() メソッドをオーバーライドし、クリーンアップ・アクティビ

ティを実行できます。たとえば、このメソッドはサーバーにより作成されたセッションを切断し、オープンされたソケットをクローズし、起動されたスレッドを停止する必要があります。postRun() が正常に終了すると、サーバーは完全に停止し、実行時に取得したシステム・リソースを解放する必要があります。

サーバーの一時停止と再開

サーバーは、オプションで一時停止 / 再開機能をサポートできます。メソッド supportsSuspendResume() は、サーバーを一時停止できるかどうかを示します。このメソッドはデフォルトで FALSE を戻します。このメソッドをオーバーライドして、一時停止 / 再開機能を有効化できます。

例 13-5 オーバーライドを使用した一時停止 / 再開の有効化

```
public boolean supportsSuspendResume()
    throws IfsException
{
    //
    // We call super, since it performs some error checking on
    // our behalf, such as making sure this IfsServer has not
    // been disposed.
    //

    super.supportsSuspendResume();

    return true;
}
```

管理者がサーバーを一時停止または再開すると、handleSuspendRequest() および handleResumeRequest() がコールされます。一時停止中のサーバーの動作は、サーバー定義です。この 2 つのメソッドをオーバーライドすると、サーバーに適した動作を実装できます。

サーバーの破棄

サーバーがそれ自体のアンロードをノードから要求されると、dispose() メソッドがコールされます。このメソッドをオーバーライドすると、最終的なクリーンアップ・タスクを実行したり、破棄要求を禁止できます。廃棄要求を禁止する場合を除き、必ず super.dispose() をコールしてください。

タイマーの使用

IfsServer には、サーバーが定期的操作のトリガーに使用できるようにタイマーが用意されています。タイマーの動作は、次の 3 つのサーバー構成パラメータで制御されます。

- IFS.SERVER.TIMER.ActivationPeriod: タイマーの有効期限です。たとえば、4h（4 時間ごと）、90m（90 分ごと）、60s（60 秒ごと）、3500（3500 ミリ秒ごと）などです。
- IFS.SERVER.TIMER.InitialTimeOfDay: タイマーの最初の有効期限です。たとえば、3:30:30（3:30 a.m.）、20:00:00（8:00 p.m.）などです。
- IFS.SERVER.TIMER.InitialDelay: 最初のタイマーが期限切れになるまでの時間です。たとえば、4h（4 時間ごと）、90m（90 分ごと）、60s（60 秒ごと）、3500（3500 ミリ秒ごと）などです。

サンプル・タイマー構成を次に示します。

- 10 分間隔の場合
  ```
  IFS.SERVER.TIMER.ActivationPeriod=10m
  ```
- 1 時間後から 10 分間隔の場合
  ```
  IFS.SERVER.TIMER.ActivationPeriod=10m
  IFS.SERVER.TIMER.InitialDelay=1h
  ```
- 3:30 a.m. から 10 分間隔の場合
  ```
  IFS.SERVER.TIMER.ActivationPeriod=10m
  IFS.SERVER.TIMER.InitialTimeOfDay=3:30:00
  ```
- 3:30 a.m. から 1 日に 1 度の場合
  ```
  IFS.SERVER.TIMER.ActivationPeriod=24h
  IFS.SERVER.TIMER.InitialTimeOfDay=3:30:00
  ```

IfsServer には、次のようにタイマー制御メソッドのセットが用意されています。

- startTimer() はサーバーのタイマーを起動します。
- stopTimer() はタイマーを停止します。
- isTimerActive() は、タイマーが起動されたかどうかを取得します。
- getLastTimerExpiration() はタイマーが前回期限切れになった時刻を取得します。
- getNextTimerExpiration() はタイマーが次回に期限切れになる時刻を取得します。

タイマーが期限切れになるたびに、handleTimerExpired() メソッドがコールされます。このメソッドをオーバーライドして、定期的な操作を実行します。

イベントへの応答

Oracle 9*i*FS リポジトリから LIBRARYOBJECT が作成、更新または削除されると、そのオブジェクトに関するイベントが転送されます。セッションでは、独自操作と他のセッションの操作により生成されたイベントを受信するように登録できます。

イベントは、`oracle.ifs.common.IfsEvent` のインスタンスです。そのメソッドは、次のとおりです。

- `getId()` は、イベントを転送した LIBRARYOBJECT の ID を戻します。
- `getClassId()` は、LIBRARYOBJECT の classobject の ID を戻します。
- `getSessionId()` は、イベントを転送したセッションの ID を戻します。
- `getEventType()` および `getEventSubtype()` は、LIBRARYOBJECT が作成されたか、更新されたか、削除されたか、特別な方法で影響を受けたかを示す値を取得します。イベントのタイプとサブタイプは、`IfsEvent` に定数で列挙されます。

サーバー、特にエージェントは、通常はイベント・ドリブンです。イベントを受信するために、エージェントのシステム・セッション（またはエージェントが保持する他のセッション）は、`LibrarySession` の `registerEventHandler` および `registerClassEventHandler` メソッドをコールして、重要なイベントを登録します。

- `registerEventHandler(LibraryObject, IfsEventHandler)` メソッドは、単一の LIBRARYOBJECT でのイベントを登録します。
- `registerClassEventHandler(ClassObject, boolean, IfsEventHandler)` メソッドは、指定の classobject（および、オプションでその classobject のサブクラス）のすべてのインスタンスでのイベントを登録します。

`IfsEventHandler` 引数はコールバック・オブジェクトを指定します。セッションは登録したイベントを受信すると、このオブジェクトの `handleEvent(IfsEvent)` をコールします。サーバーの場合、最も簡単なアプローチは、サーバー自体が `IfsEventHandler` インタフェースを実装することです。

```
public class MyAgent
    extends IfsServer
    implements IfsEventHandler
```

通常、サーバーは `preRun()` メソッドでイベントを登録します。`preRun()` で `registerEventHandler` または `registerClassEventHandler` をコールするたびに、それに対応して `postRun()` でも `deregisterEventHandler` または `deregisterClassEventHandler` をコールする必要があります。

例 13-6 preRun() メソッドでのイベントの登録

```
public void preRun()
    throws Exception
{
```

```
    LibrarySession session = connectSession();

    // Register for events on the root folder.
    Folder rootFolder = session.getRootFolder();
    session.registerEventHandler(rootFolder, this);

    // Register for events on all Documents (and subs).
    Collection c = session.getClassObjectCollection();
    ClassObject co = (ClassObject)c.getItems("DOCUMENT");
    session.registerEventHandler(co, true, this);
}

public void postRun()
{
    try
    {
        LibrarySession session = getSession();

        // Deregister for events on the root folder.
        Folder rootFolder = session.getRootFolder();
        session.deregisterEventHandler(rootFolder, this);

        // Deregister for events on all Documents (and subs).
        Collection c = session.getClassObjectCollection();
        ClassObject co = (ClassObject)c.getItems("DOCUMENT");
        session.deregisterEventHandler(co, true, this);

        disconnectSession();
    }
    catch (Exception e)
    {
        log(LEVEL_LOW, e);
    }
}
```

スレッド・セーフな方法でイベントを処理するために、サーバーの `handleEvent(IfsEvent)` メソッドはイベント処理をサーバーの管理スレッドに委譲する必要があります。そのためには、`handleEvent(IfsEvent)` に `queueEvent(IfsEvent)` をコールさせます。これにより、`processEvent(IfsEvent)` が管理スレッド内でコールされます。

例 13-7 スレッド・セーフな方法によるイベントの処理

```
public void handleEvent(IfsEvent event)
    throws IfsException
{
    // Ignore "FREE" events (for example).
```

```
    // But process all other event types.

    if (event.getEventType() != IfsEvent.EVENTTYPE_FREE)
    {
        queueEvent(event);
    }
}

public void processEvent(IfsEvent event)
    throws Exception
{
    // This is where the server actually handles the event.
    // (This executes in the server's administration thread.)

    // For example, log the event.
    log(LEVEL_MEDIUM, "Event received: " + event.getId());
}
```

優先順位の管理

サーバーでは、その優先順位を管理できます。サーバーの優先順位は、1 ～ 10 の範囲内で指定できます。最上位の優先順位を持つサーバーは、優先順位の低いサーバーに優先して実行されます。

`supportsPriority()` メソッドは、サーバーで優先順位の管理がサポートされるかどうかを示します。デフォルトでは、このメソッドは TRUE を戻します。このメソッドをオーバーライドして、優先順位の管理を無効化できます。

管理者がサーバーの優先順位を設定すると、`handlePriorityChangeRequest()` がコールされます。このメソッドをオーバーライドして、サーバーにより作成されるすべてのスレッドの優先順位を設定するなど、優先順位の変更タスクを実行します。サーバーの新規優先順位を取得するには、`getPriority()` をコールします。

サービス構成パラメータ

サーバーがノードにロードされると、`getParameterTable()` をコールして取得できるサービス構成パラメータのセットが与えられます。この種のサービス構成パラメータは、サーバーのライフサイクル中は変化しません。

`IfsServer` は、次のサービス構成パラメータを使用します。

- `IFS.SERVER.Class`: サーバーの完全修飾クラス名。これは必須です。
- `IFS.SERVER.SESSION.User`: サーバーのデフォルト・セッションのユーザー。デフォルト・セッションを使用する場合は必須です。

- IFS.SERVER.SESSION.ApplicationName: デフォルト・セッションの ConnectOptions アプリケーション名。オプションです。デフォルトでサーバー名に設定されます。
- IFS.SERVER.SESSION.LOCALE.Language、IFS.SERVER.SESSION.LOCALE.Country および IFS.SERVER.SESSION.LOCALE.Variant: 言語、国および可変を表す Java 定義コードとしての、デフォルト・セッションの ConnectOptions ロケール。オプションです。指定していない場合、デフォルト・セッションの ConnectOptions ロケールは設定されません。
- IFS.SERVER.TIMER.ActivationPeriod: タイマーの期限切れ間隔。タイマーを使用する場合は必須です。たとえば、2500（2.5 秒ごと）、5s（5 秒ごと）、10m（10 分ごと）、2h（2 時間ごと）などです。
- IFS.SERVER.TIMER.InitialDelay: 最初のタイマーが期限切れになるまでの遅延。オプションです。このパラメータと IFS.TIMER.InitialTimeOfDay の値を指定していない場合は、デフォルトで IFS.SERVER.TIMER.ActivationPeriod の値に設定されます。たとえば、2500（2.5 秒ごと）、5s（5 秒ごと）、10m（10 分ごと）、2h（2 時間ごと）などです。
- IFS.SERVER.TIMER.InitialTimeOfDay: 最初のタイマー・イベントの時刻。オプションです。IFS.SERVER.TIMER.InitialDelay に指定した値をオーバーライドします。たとえば、3:30:00（3:30 a.m.）、20:00:00（8:00 p.m.）などです。
- IFS.SERVER.TIMER.HourSuffix、IFS.SERVER.TIMER.MinuteSuffix, および IFS.SERVER.TIMER.SecondSuffix: 時間単位の接尾辞。オプションです。デフォルトでそれぞれ H、M および S に設定されます。
- IFS.SERVER.TIMER.TimeFormat: 時刻書式。オプションです。デフォルトで HH:mm:ss に設定されます。

サーバーは、独自のサービス構成パラメータを導入することもできます。

動的プロパティ

サーバーは、オプションで名前付きの動的プロパティを 1 つ以上発行し、その値をサーバーの実行時に変更できます。管理者は、Oracle 9*i*FS 管理ツールを使用してサーバーの動的プロパティの値を監視し、必要に応じて変更できます。

IfsServer には、動的プロパティを管理するメソッドのセットが用意されています。

- getProperties() は、サーバーのすべての動的プロパティを AttributeValue 配列として戻します。
- getProperty(String) は、指定のプロパティの値を取得します。
- setProperty(AttributeValue) および setProperty(String, AttributeValue) は、指定のプロパティの値を設定します。

- `isPropertyReadonly(String)` は、管理者が Oracle 9*i*FS 管理ツールを使用して指定のプロパティを変更できるかどうかを示す値を戻します。デフォルトでは、すべての動的プロパティは読取り専用です。このメソッドをオーバーライドして、動的プロパティを設定可能にします。
- `handlePropertyChangeRequest(AttributeValue)` は、動的プロパティの変更時にコールされます。このメソッドをオーバーライドし、プロパティ変更に応答してタスクを実行します。デフォルトでは、このメソッドは単に要求されたプロパティの値を変更します。

カスタム・サーバーのテスト

カスタム・サーバーは、Oracle 9*i*FS ノードで実行するのみでなく、スタンドアロンの Java アプリケーションとしても実行できます。スタンドアロン・モードは、カスタム・サーバーの予備テストを行うには便利ですが、本番システムには使用しないでください。

通常、サーバーはサーバー構成パラメータをノードから取得します。ただし、スタンドアロン・モードの場合、サーバーはこの情報を名前 / 値のペアのファイルから取得します。

カスタム・サーバーをスタンドアロン・モードで実行するには、`IfsServer` サブクラスに Java の `static main(String[])` メソッドを追加する必要があります。このメソッドは、アプリケーションのエントリ・ポイントを提供するのみでなく、通常はノードで処理される次のような複数のタスクを実行する必要があります。

- サーバー構成パラメータを含む `oracle.ifs.common.ParameterTable` の作成
- サービスの起動
- サーバーのインスタンスの構成
- サーバーの初期化
- サーバーの起動

例 13-8 に、FingerServer の `main` メソッドを示します。この例を独自サーバーのテンプレートとして使用できます。

例 13-8 FingerServer の main メソッド

```
public static void main(String[] args)
    throws Exception
{
    //
    // If something goes wrong we want verbose exception messages.
    //

    IfsException.setVerboseMessage(true);

    //
```

```
    // Construct a parameter table from command-line arguments.
    //

    ParameterTable pt = new ParameterTable(args, "parameterfile");

    //
    // Extract some additional parameters required for standalone
    // operation.
    //

    String serviceName =
        pt.getString("IFS.SERVER.PROTOCOL.FINGER.TEST.Service");

    String schemaPassword = pt.getString(
        "IFS.SERVER.PROTOCOL.FINGER.TEST.SchemaPassword");

    //
    // Start a service against which to run a FingerServer.
    //

    LibraryService.startService(serviceName, schemaPassword);

    //
    // Construct, initialize, and start a FingerServer.
    //

    FingerServer server = new FingerServer();

    server.initialize("Finger", serviceName, schemaPassword,
        pt, null, LEVEL_HIGH);

    server.start();
}
```

main メソッドを使用すると、サーバーをスタンドアロン・モードで起動できます。たとえば、FingerServer をスタンドアロン・モードで起動するには、次のコマンドラインを入力します。

```
java -mx64M oracle.ifs.examples.servers.FingerServer
  parameterfile=FingerServer.def
  IFS.SERVER.PROTOCOL.FINGER.TEST.Service=IfsDefault
  IFS.SERVER.PROTOCOL.FINGER.TEST.SchemaPassword=ifssys
```

カスタム・サーバーの配置

カスタム・サーバーを配置するには、次のタスクを実行する必要があります。

- **サーバー構成の作成：** Oracle 9*i*FS Manager を使用してサーバーのサーバー構成パラメータを定義し、サーバー構成を作成します。パラメータ `IFS.SERVER.Class` を含めてください。このパラメータの値は、サーバーの完全修飾クラス名です。
- **ノード構成の更新：** サーバーを自動的にロードしてノード上で起動する場合は、Oracle 9*i*FS Manager を使用して、そのノードの構成を更新します。次の情報を指定して、このノードに新規サーバーを追加します。
  - サーバー名。
  - サーバーの構成パラメータを提供するサーバー構成オブジェクト。
  - サーバーがアクティブかどうか。アクティブでないサーバーは、ノードにより自動的にはロードされません。
  - サーバーの操作対象となるサービスの名前。
  - サーバーの優先順位。
  - サーバーがロード後に自動的に起動するかどうか。

カスタム HTTP サーバー

Apache Web サーバー内で実行するカスタム Oracle 9*i*FS サーバーを作成して配置するプロセスは、この章で説明したものと同じです。ただし、HTTP サーバーの場合は、他にもいくつか考慮点があります。

- カスタム・サーバーを使用するカスタム HTTP サーブレットを作成する場合は、サーバー側のメソッドをコールできるように、HTTP サーブレットでサーバーの参照を取得するためのメカニズムが必要です。アプローチの 1 つは、サーバーの単一インスタンスを戻す `public static` メソッドをサーバーのクラスに用意することです。（このメソッドの実装では、サーバーの `initialize` メソッドにより設定されたクラス変数を使用できます。）
- サーブレットの `init` メソッドでは、`oracle.ifs.management.domain.HttpNodeUtilities` の静的 `createNode(HttpServlet)` メソッドをコールして、Apache Web サーバーの JavaVM 内で Oracle 9*i*FS ノードを構成する必要があります。（Oracle 9*i*FS に付属の `DavIfsServlet` を引き続き Apache Web サーバーにロードする場合は、このコールは不要です。`createNode` メソッドは、ノードが構成済みの場合は無効です。）

例

Oracle 9*i*FS には、カスタム・サーバーの 2 つの例が用意されています。

- **FingerServer:** `oracle.ifs.examples.servers.FingerServer` は、Finger 形式のプロトコル（RFC 1288 に類似）を提供する単純なプロトコル・サーバーです。
- **LoggingAgent:** `oracle.ifs.examples.servers.LoggingAgent` は、単純なイベントベース・エージェントです。

サーバーごとに、コメント付きの完全なソース・コードとサーバー構成ファイル（スタンドアロン・テスト用）があります。

詳細情報

カスタム・サーバーの詳細は、次を参照してください。

- `oracle.ifs.management.domain` パッケージの Javadoc
- カスタム・サーバーの例（FingerServer および LoggingAgent）

14

バージョニングの実装

この章では、Oracle Internet File System によるバージョニングの実装方法と、カスタム・アプリケーションにバージョニングを実装する方法について説明します。

この章の内容は、次のとおりです。

バージョニングの概要

バージョニングは、オブジェクトに対する変更内容を、そのライフサイクルを通じて系統的に保存することです。たとえば、バージョニングにより、変更者、変更日時および変更順序を記録して、ドキュメントの内容や属性に対する変更をすべて追跡できます。

ドキュメントの各バージョンのバックアップを、特定の時点でのドキュメントの体裁のレコードとして保存できます。バージョニングされていないドキュメントでは、変更内容がオリジナルの内容に上書きされます。バージョニングを実装するドキュメントでは、変更内容が新規バージョンとして作成され、オリジナルの内容も残ります。

また、バージョニングにより、オブジェクトの変更方法に関するビジネス・ルールを施行して、オーサリングおよび公開プロセスを管理できます。

Oracle 9*i*FS のバージョニングはシリアル・モデルです。チェックイン / チェックアウト機能を使用すると、一度に 1 人の作成者のみがドキュメントを編集できることを保証できます。作成者は、変更中に他の作成者が更新できないように、ドキュメントをチェックアウトできます。作成者がドキュメントを編集するときに、新バージョンが完成するまでは変更内容を一時的に格納できます。完成した時点で、作成者はドキュメントの新バージョンを他の作成者が使用できるようにチェックインします。チェックイン / チェックアウト機能により、各作成者が適切なバージョンのドキュメントを操作し、複数の作成者が相互の作業を上書きしないことが保証されます。

バージョニング機能をアプリケーションで使用するには、ユーザー要件とシステム・リソースのバランスを慎重に調整する必要があります。ドキュメントは、コンテンツ管理システムでバージョニング対象となる最も一般的なオブジェクトです。ドキュメントの新規バージョンを作成すると、ミッション・クリティカルな情報の改訂履歴を残せますが、その反面、記憶領域を消費し、システム・パフォーマンスにも影響を及ぼすことがあります。ドキュメントのバージョニングの必要性とパフォーマンスに及ぼす全体的な影響のバランスを慎重に検討する必要があります。

バージョニング・オブジェクト・モデル

Oracle 9*i*FS には、情報を作成して発行するための多様なプロセスをサポートできるように、柔軟なバージョニング・モデルが用意されています。ほとんどの PublicObject はバージョニングできます。PublicObject は、ドキュメントやフォルダなど、Oracle 9*i*FS ユーザーがアクセスできるオブジェクトです。この種のオブジェクトは、Oracle 9*i*FS のクラス階層内で PublicObject から派生します。デフォルトでは、バージョニング可能に設定されるのは Document クラスのみです。他のクラスのオブジェクトをバージョニングする計画がある場合は、必要なディスク領域とパフォーマンスを考慮してください。この章では、具体例を示すためにドキュメントをバージョニングする方法を重点的に説明します。

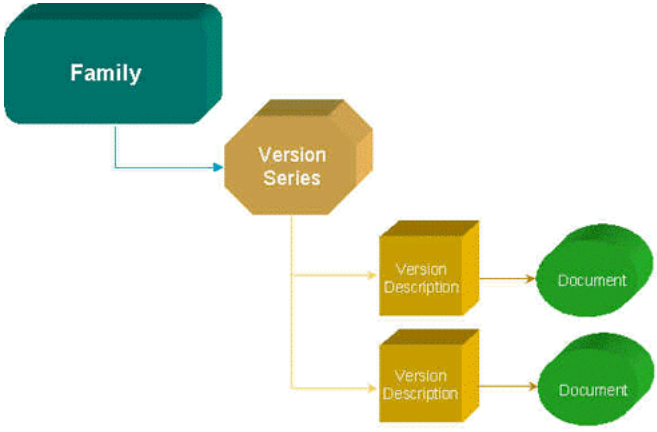
注意： ClassObject クラスの `isVersionable()` メソッドを使用すると、特定クラスのインスタンスをバージョニングできるかどうかを判断できます。

Oracle Internet File System の 4 つの主要なバージョニング・クラスは、Family、VersionSeries、VersionDescription および Document です。

リポジトリ内でバージョニングされないドキュメントは、Document クラスのインスタンス、または Document を拡張するカスタム・クラスとして表されます。インスタンスは、Name、Owner および CreationDate など、ドキュメントを記述する構造化された情報を持ちます。この種の情報は Document クラスの属性として表されます。また、ドキュメントは構造化されていない内容を持ち、この内容は Document クラスの ContentObject 属性で参照される ContentObject のインスタンスとして表されます。

ドキュメントがバージョニングされる場合、Oracle 9*i*FS では 3 つの型のオブジェクトが追加作成され、ドキュメントのライフサイクル中に作成されるバージョンがすべて管理されます。この 3 つの型は、図 14-1 のように Family、VersionSeries および VersionDescription です。

図 14-1 オブジェクトのバージョニング



- **Family:** Document がバージョニングされる場合、Family オブジェクトはドキュメントを表し、そのドキュメントについて作成されたすべてのバージョンが収集されます。
- **VersionSeries:** Family はバージョンを VersionSeries にグループ化します。VersionSeries は、ドキュメントに対して順番に行われた変更のシリーズを表します。VersionSeries 内の各バージョンは、特定時点でのドキュメントの状態を表します。VersionSeries 内のバージョンの順序は、ドキュメントに対して行われた変更の順序を表します。
- **VersionDescription:** VersionSeries 内で、ドキュメントの各バージョンは VersionDescription で表されます。VersionDescription は、順序番号（1、2、3 など）のように、シリーズ内のバージョンのコンテキストの記述です。各 VersionDescription

は、ドキュメントの特定バージョンの属性と内容を含む Document のインスタンスを参照します。

デフォルトでは、Oracle 9*i*FS のクライアントおよびプロトコルは、シリアル・バージョニング・モデルを実装します。このモデルでは、ドキュメントの Family に 1 つの VersionSeries が含まれ、その VersionSeries にドキュメントの全バージョンが含まれます。バージョンは順番に、連続して作成されます。

VersionSeries の各 VersionDescription では様々なクラスのオブジェクトを参照できるため、クラス間でのドキュメントの状態の変化を表すことができます。たとえば、ドキュメントのバージョン 1 が RequirementsDocument（Document のサブクラス）で、これがバージョン 2 では DesignSpecification に変更される場合があります。

VersionDescription では、Document の同一インスタンスも参照できます。たとえば、ドキュメントに発行前 VersionSeries と発行済み VersionSeries があるとします。発行前 VersionSeries の第 3 のバージョンでは、発行済み VersionSeries の最初のバージョンを表している場合があります。この場合、第 3 の発行前バージョンと最初の発行済みバージョンを表す VersionDescription では、Document の同一インスタンスを参照できます。

バージョニングのクラス

Oracle 9*i*FS には、バージョニング機能を実装するカスタム・アプリケーションを構築できるように、Java API が用意されています。表 14-1「Document の属性およびメソッド」、表 14-2「Family の属性およびメソッド」、表 14-3「VersionSeries の属性およびメソッド」および表 14-4「VersionDescription の属性およびメソッド」に、Oracle 9*i*FS Java API の各バージョニング・クラスの主要な属性とメソッドを示します。各クラスの詳細は、Javadoc を参照してください。

Document クラス

最終的に、ドキュメントの全バージョンは Document クラスのインスタンスとして格納されます。ユーザーがドキュメントを変更するときに、アプリケーションでは次のアクションを使用可能にできます。

1. **ドキュメントの現行のバージョンの上書き：** 変更内容を上書きするには、アプリケーションでそのバージョンの VersionDescription で参照される Document インスタンスを変更します。
2. **ユーザーによる変更の一時的な格納：** 変更内容を一時的に格納するには、アプリケーションで Document の新規インスタンスを作成し、それを VersionSeries の PendingPublicObject 属性で参照します。
3. **ドキュメントの新バージョンの作成：** 新バージョンを作成するには、アプリケーションで、変更内容を含む新規 Document インスタンスを参照する VersionSeries に、新規 VersionDescription を作成します。変更が一時的に格納されている場合は、新規 Document インスタンスを作成するかわりに、PendingPublicObject 属性で参照される Document インスタンスを使用できます。

Document クラスには、ドキュメントに対する変更を追跡できるように、主要属性のセットがあります。表 14-1「Document の属性およびメソッド」のように、この種の属性とメソッドのほとんどはスーパークラス PublicObject から継承され、他の型のオブジェクトのバージョニングに使用できます。また、Document クラスには、ドキュメントの内容を操作するための属性とメソッドも用意されています。

ユーザーがドキュメントを変更する場合に、アプリケーションではユーザーに対して、ドキュメントの新バージョンの作成ではなく、ドキュメントの属性値や内容の置換を許可できます。この機能をサポートするために、PublicObject は、ドキュメントの作成日、作成者、最終変更日および変更者を追跡するための属性を持っています。PublicObject クラスも Family 属性を持ち、バージョニング対象ドキュメントの Family を簡単に検索できます。

表 14-1 Document の属性およびメソッド

属性 / メソッド	用途
Creator getCreator()	ドキュメント（または PublicObject）のこのインスタンスを最初に作成した DirectoryUser を格納し、取得します。
CreateDate getCreateDate()	ドキュメントが最初に作成された日付を格納し、取得します。
Modifier getLastModifer()	ドキュメントのこのインスタンスを最後に変更した DirectoryUser を取得します。オプションで、ユーザーはドキュメントのインスタンスをバージョニングせずに変更できます。たとえば、バージョニング対象外のドキュメントの内容を置換したり、ドキュメントの最初のバージョンの属性値を更新できます。
LastModifiedDate getLastModifuDate()	ドキュメントのこのインスタンスが最後に変更された日付を取得します。
Family getFamily()	有効ドキュメントを表す Family オブジェクトを取得します。ドキュメントは、1 つの Family にのみ属することができます。
isVersioned()	ドキュメントがバージョニング対象かどうかを判断します。
isVersionable()	クラスがバージョニング可能かどうかを判断します。
lock() unlock()	ドキュメントのロックを取得し、解放します。
ContentObject getContentObject() getContentReader() getContentStream() setContent()	ドキュメントの内容を格納、取得および更新します。

Family クラス

ドキュメントがバージョニング対象の場合、Oracle 9*i*FS は Family クラスのインスタンスを作成して、ライフサイクル中にドキュメントに対して作成される全バージョンを管理します。バージョニング対象ドキュメントは、1 つの Family にのみ属することができます。表 14-2「Family の属性およびメソッド」に、Family の管理に重要な属性とメソッドを示します。

表 14-2 Family の属性およびメソッド

属性 / メソッド	用途
PrimaryVersionSeries getPrimaryVersionSeries() setPrimaryVersionSeries()	ドキュメントのプライマリ・バージョン・シリーズを指定してアクセスします。Family に含めることができる VersionSeries は 1 つのみで、これが PrimaryVersionSeries になります。今後のリリースでは、PrimaryVersionSeries を使用して、Family で複数の VersionSeries がサポートされる予定です。
getResolvedPublicObject()	有効ドキュメントの現行の状態を表すドキュメントのバージョンにアクセスします。たとえば、ユーザーがバージョニング対象のドキュメントをダブルクリックし、その内容を表示すると、Oracle 9*i*FS では DefaultVersionDescription 属性で指定されたバージョンが戻されます。この属性が指定されていない場合、Oracle 9*i*FS では、バージョニング対象ドキュメントの現行の状態を表すプライマリ・バージョン・シリーズの最終バージョンが戻されます。
DefaultVersionDescription getDefaultVersionDescription() setDefaultVersionDescription()	カスタム・バージョニング・アプリケーションは、これらの属性とメソッドを使用して、プライマリ・バージョン・シリーズの最終バージョンとは異なるドキュメントのバージョンを、デフォルト・バージョンとして明示的に設定できます。

`getResolvedPublicObject()`、`DefaultVersionDescription` および `PrimaryVersionSeries` を使用してバージョニング対象ドキュメントの現行の状態を解決する方法の詳細は、「バージョニング対象ドキュメントの解決」を参照してください。

VersionSeries クラス

VersionSeries は、ドキュメントに対して順番に行われた一連の変更（バージョン）を表します。表 14-3「VersionSeries の属性およびメソッド」に、この種の変更を管理するための属性とメソッドを示します。

表 14-3 VersionSeries の属性およびメソッド

属性 / メソッド	用途
getFirstVersionDescription()	バージョン・シリーズの最初のバージョンにアクセスします。
LastVersionDescription getLastVersionDescription()	バージョン・シリーズの最後のバージョンを格納し、アクセスするために使用します。
getVersionDescriptions() getVersionDescriptions(int index) getNextVersionDescription() getPreviousVersionDescription()	バージョン・シリーズ内の、すべてのバージョンの配列、特定バージョンおよび次または前のバージョンにアクセスします。
DefaultVersionDescription getDefaultVersionDescription() setDefaultVersionDescription()	カスタム・バージョニング・アプリケーションは、これらの属性とメソッドを使用して、プライマリ・バージョン・シリーズの最終バージョンとは異なるドキュメントのバージョンを、デフォルト・バージョンとして明示的に設定できます。
Reservor ReservationDate ReservationComments reserveNext() unReserve() isReserved() isReservedByCurrentUser() getReservor() getReservationDate() getReservationComment()	特定のユーザーが変更を行っている間は、他のユーザーがシリーズの新バージョンを作成しないように、VersionSeries を予約し、予約を解除します。Oracle 9*i*FS では、バージョン・シリーズが予約された時期、予約したユーザーおよびユーザー指定のコメントが追跡されます。
WorkPath getWorkPath()	ユーザーがドキュメントに変更を加える場合に、バージョニング・アプリケーションはそのユーザーに対して、ドキュメントをオーサリング・クライアントのローカル・オペレーティング・システムにコピーするように許可できます。WorkPath 属性を使用すると、エクスポートされたコピーへのパスを追跡できます。 **注意：** Oracle 9*i*FS クライアントおよびプロトコルでは、この機能を提供するために WorkPath 属性が使用されることはありません。

表 14-3 VersionSeries の属性およびメソッド（続く）

属性 / メソッド	用途
PendingPublicObject getPendingPublicObject() setPendingPublicObject()	バージョン・シリーズを予約したユーザーは、変更結果を新バージョンとしてチェックインする前に一時的に格納できます。変更は、PublicObject の別インスタンスとして一時的に格納され、PendingPublicObject 属性により参照されます。
newVersion()	バージョン・シリーズの新バージョンを作成します。
VersionLimit setVersionLimit()	バージョン・シリーズに保存されるバージョン数。

これらのメソッドと属性を使用してドキュメントをバージョニングする方法の詳細は、「バージョニング対象ドキュメントの設定」を参照してください。

VersionDescription クラス

VersionDescription では、バージョン・シリーズのコンテキスト内でドキュメントのバージョンを記述します。表 14-4「VersionDescription の属性およびメソッド」に、ドキュメントの特定バージョンを操作するための属性とメソッドを示します。

表 14-4 VersionDescription の属性およびメソッド

属性 / メソッド	用途
VersionSeries VersionNumber RevisionComment getVersionSeries() getVersionNumber() getRevisionComment()	このバージョンが属するバージョン・シリーズ、バージョン・シリーズ内でこのバージョンに割り当てられた順序番号、バージョンの作成時にユーザーにより指定されたコメントを追跡します。
VersionLabel getVersionLabel() setVersionLabel() 6	VersionLabel 属性には、バージョン・シリーズのバージョンのカスタム・ラベルが格納されます。バージョニング・アプリケーションでは、組織の改訂採番方式に準拠するバージョン・ラベルを格納できます。たとえば、Oracle 9*i*FS ではバージョンに順序を示す整数（1、2、3）が自動的に割り当てられますが、組織ではバージョンにラベル（ドラフト A、ドラフト B、ドラフト C、リリース 1.0）が必要な場合があります。
isLatestVersionDescription()	バージョンがバージョン・シリーズの最終バージョンかどうかを判断します。

表 14-4 VersionDescription の属性およびメソッド（続く）

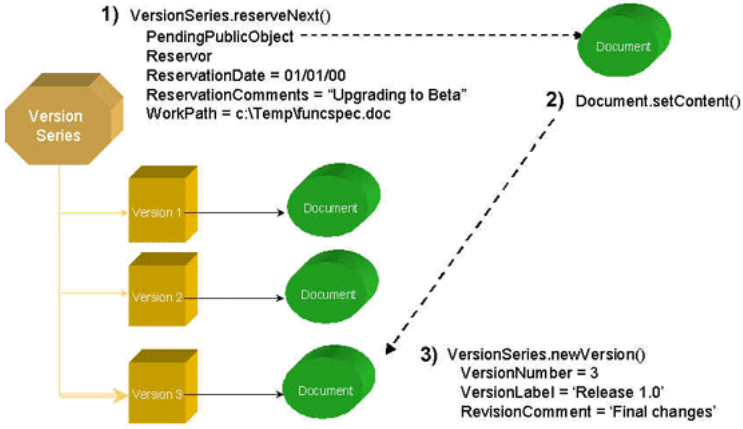
属性 / メソッド	用途
PublicObject getPublicObject()	ドキュメントの各バージョンは、最終的に Document クラスのインスタンスとして格納されます。VersionDescription の PublicObject 属性は、そのバージョンを構成するドキュメント・インスタンスを参照します。getPublicObject() メソッドは、ドキュメント、つまり PublicObject を取得するために使用されます。

バージョニング・アプリケーションの実装

Oracle 9*i*FS のバージョニング・クラスを使用すると、カスタム・アプリケーションにバージョニング機能を持たせることができます。たとえば、バージョニング・アプリケーションでは、次の 3 手順で新バージョンを作成できます。

1. ドキュメントをチェックアウトします。
2. ドキュメントを編集し、変更結果を一時的な保留領域に保存します。
3. 変更結果をドキュメントの新バージョンとしてチェックインします。

図 14-2 新バージョンの作成手順



この機能を実装するには、バージョニング・アプリケーションで次の API コールを使用します。

1. アプリケーションで、ドキュメントの VersionSeries オブジェクトの `reserveNext()` メソッドをコールして、他のユーザーが同時に変更を行わないようにします。アプリ

ケーションで VersionSeries の Reservor、ReservationDate、ReservationComments および WorkPath 属性を設定し、ユーザーに予約者、予約理由および予約日に関する情報を提供します。

2. ユーザーがドキュメントを編集すると、アプリケーションは Document のインスタンスをもう 1 つ作成し、それを VersionSeries の PendingPublicObject 属性で参照します。アプリケーションは Document インスタンスの `setContent()` メソッドをコールし、ユーザーによる反復を一時的に格納してから、ドキュメントの新バージョンを作成します。

3. 変更内容がドキュメントの新バージョンとして格納される場合、アプリケーションは `newVersion()` メソッドをコールして新規の VersionDescription を作成し、新バージョンの VersionLabel、RevisionComments および PublicObject の値を設定します。新規 VersionDescription の PublicObject 属性を、ユーザーの変更を VersionSeries の PendingPublicObject として格納するために使用された Document インスタンスを参照するように設定します。

この項では、Oracle 9*i*FS Java API を使用し、次の基本的なバージョニング・タスクを実行して、ドキュメントのシリアル・バージョニング機能を持つカスタム・アプリケーションを構築する方法について説明します。

- バージョニング対象ドキュメントの設定
- ドキュメントのチェックアウト
- 保留中の変更の保持
- 新バージョンのチェックイン
- ドキュメントのチェックアウトの取消し
- フォルダへのバージョニング対象ドキュメントの格納
- バージョニング対象ドキュメントの解決
- ドキュメント履歴の表示
- 特定のドキュメント・バージョンの削除
- バージョニング対象ドキュメントの削除

バージョニング対象ドキュメントの設定

デフォルトでは、ドキュメントは最初に Oracle 9*i*FS で作成された時点では、バージョニング対象ではありません。ユーザーがドキュメントをバージョニングできるように、アプリケーションでは最初にバージョニング対象ドキュメントのインフラストラクチャ・オブジェクト Family、VersionSeries および VersionDescription を作成する必要があります。

ドキュメントをバージョニング対象にする手順は、次のとおりです。

1. ドキュメントをフェッチします。

2. （オプション）FamilyDefinition を作成します。
3. （オプション）VersionSeriesDefinition を作成し、Family に属するように指定します。
4. VersionDescriptionDefinition を作成し、VersionSeries に属してドキュメントを参照するように指定します。
5. VersionDescriptionDefinition を LibrarySession に渡して、ドキュメントの Family、VersionSeries および VersionDescription オブジェクトを作成し、VersionDescription を戻します。
6. VersionDescription オブジェクトから、バージョニング対象ドキュメントの他のコンポーネントにアクセスできます。

バージョニング対象ドキュメントの 4 つのコンポーネント（DocumentDefinition、VersionDescriptionDefinition、VersionSeriesDefintion、FamilyDefinition など）すべてについて、定義を明示的に作成する必要はありません。かわりに、バージョニング対象ドキュメントを参照する VersionDescriptionDefinition を使用できます。VersionDescriptionDefinition が LibrarySession に渡されると、Oracle 9*i*FS はバージョニング対象ドキュメントの 4 つのコンポーネントすべてを自動的に作成します。Oracle 9*i*FS では、このうち 2 つの定義オブジェクトから他のコンポーネントを作成できます。例 14-1 に、すべてのコンポーネントを明示的に定義して、ドキュメントをバージョニング対象にする方法を示します。VersionDescriptionDefinition を使用してバージョニング対象ドキュメントのコンポーネントを暗黙的に作成する方法は、例 14-2 を参照してください。

SecuringPublicObject 属性を使用すると、バージョニング対象ドキュメントのすべてのコンポーネントに対するアクセスを、1 箇所で制御できます。あるコンポーネントを他のコンポーネントの SecuringPublicObject 属性で参照し、そのコンポーネントの AccessControlList をバージョニング対象ドキュメントのすべてのコンポーネントに適用できます。SecuringPublicObject 属性を設定すると、Oracle 9*i*FS は自動的に参照先オブジェクトの AccessControlList を使用して、参照しているオブジェクトへのアクセスを制御します。既存のドキュメントをバージョニング対象にすると、Oracle 9*i*FS は自動的に Document、VersionDescription および VersionSeries の SecuringPublicObject 属性を、ドキュメントの Family を参照するように設定します。新規のバージョニング対象ドキュメントを作成する場合は、例 14-2 のように SecuringPublicObject 属性を明示的に設定する必要があります。

注意： SecuringPublicObject 属性の使用方法の詳細は、第 15 章「セキュリティ」を参照してください。

例 14-1 バージョニング対象ドキュメントの設定

1. アプリケーションがドキュメントを取得しているとします。

```
Document doc = ...
```

2. オプションで FamilyDefinition を作成します。

```
FamilyDefinition fd = new FamilyDefinition ( session );
fd.setAttribute( PublicObject.NAME_ATTRIBUTE ,
                 AttributeValue.newAttributeValue ( po.getName() ) );
AccessControlList acl = doc.getAcl();
fd.setAttribute(PublicObject.ACL_ATTRIBUTE,
                AttributeValue.newAttributeValue(acl));
```

3. オプションで VersionSeriesDefinition を作成します。

```
VersionSeriesDefinition vsd = new VersionSeriesDefinition(session);
vsd.setFamilyDefinition(fd);
```

4. VersionDescriptionDefinition を作成します。

```
VersionDescriptionDefinition vdd =
      new VersionDescriptionDefinition(session);
vdd.setPublicObject(doc);
vdd.setVersionSeriesDefinition(vsd);
vdd.setOwnerBasedOnPublicObjectOption ( true );
```

5. VersionDescription を作成します。

```
VersionDescription vd =
     (VersionDescription) session.createPublicObject(vdd);
```

6. VersionDescription から VersionSeries および Family にアクセスします。

```
VersionSeries vs = vd.getVersionSeries();
Family family = vs.getFamily();
```

例 14-2 新規バージョニング対象ドキュメントの作成

1. DocumentDefinition を作成します。

```
DocumentDefinition dd = new DocumentDefinition(session);
dd.setAttribute(Document.NAME_ATTRIBUTE,
   AttributeValue.newAttributeValue(docname));

Collection formatColl = session.getFormatCollection();
Format fmt = (Format) formatColl.getItems("Text");
dd.setFormat(fmt);

dd.setContent("A new versioned document.");

Folder hf = session.getUser().getPrimaryUserProfile().getHomeFolder();
dd.setAddToFolderOption(hf);
```

2. VersionDescriptionDefinition を作成します。

```
VersionDescriptionDefinition vdd = new VersionDescriptionDefinition(session);
vdd.setAttribute(VersionDescription.NAME_ATTRIBUTE,
    AttributeValue.newAttributeValue("Versioning1 VersionDescription"));
vdd.setAttribute("VERSIONLABEL",
    AttributeValue.newAttributeValue("Version 1.0"));
vdd.setPublicObjectDefinition(dd);
```

3. VersionDescriptionDefinition を LibrarySession に渡して、Document、VersionDescription、VersionSeries および Family を暗黙的に作成し、VersionDescription を戻します。

```
VersionDescription vd = (VersionDescription)session.createPublicObject(vdd);
```

4. VersionDescription を使用して他のコンポーネントをフェッチします。

```
Family f = vd.getFamily();
VersionSeries vs = vd.getVersionSeries();
Document d = (Document) vd.getResolvedPublicObject();
```

5. VersionDescription および VersionSeries の SecuringPublicObject を、ドキュメントの Family になるように設定します。

```
d.setSecuringPublicObject(f);
vd.setSecuringPublicObject(f);
vs.setSecuringPublicObject(f);
```

ドキュメントのチェックアウト

ドキュメントをバージョニング対象にした後に、エンド・ユーザーに対してドキュメントのチェックアウトを許可できます。ユーザーがドキュメントをチェックアウトすると、他のユーザーはそのドキュメントの新バージョンを作成できなくなります。

ドキュメントをチェックアウトする手順は、次のとおりです。

1. ドキュメントをフェッチします。
2. ドキュメントがバージョニング対象であることを確認します。
3. 新バージョンを作成する VersionSeries をフェッチします。
4. VersionSeries がまだ予約されていないことと、ドキュメントの Family がロックされていないことを確認します。
5. `reserveNext()` メソッドをコールして VersionSeries を予約します。`reserveNext()` メソッドは、他のユーザーがそのシリーズの新バージョンを作成できないように VersionSeries を予約します。

例 14-3 ドキュメントのチェックアウト

1. アプリケーションがすでにドキュメントを取得しているとします。

```
Document doc ...
```

2. ドキュメントがバージョニング対象であることを確認します。

```
if (doc.isVersioned())
{
```

3. 新バージョンが含まれる VersionSeries をフェッチします。

```
Family family = doc.getFamily();
VersionSeries vs = family.getPrimaryVersionSeries();
```

4. VersionSeries がまだ予約されていないことと、ドキュメントの Family がロックされていないことを確認します。

```
if (!vs.isReserved() && !family.isLocked())
{
```

5. reserveNext() メソッドをコールして VersionSeries を予約します。reserveNext() メソッドは、他のユーザーがそのシリーズの新バージョンを作成できないように VersionSeries を予約します。

```
     vs.reserveNext(null,
                   "File Reserved By " +
                    session.getDirectoryUser().getDistinguishedName());
    }
}
```

保留中の変更の保持

`reserveNext()` メソッドは、コールされてドキュメントをチェックアウトするときに、そのドキュメントの最終バージョンに基づいて新規 Document インスタンスを作成してから、VersionSeries の PendingPublicObject 属性を Document インスタンスに設定します。PendingPublicObject は、チェックアウト中にドキュメントに対して行われたすべての変更内容の一時的な保留場所として機能します。ドキュメントの反復は、それぞれ Oracle 9*i*FS に Document インスタンスとして記録され、VersionSeries の PendingPublicObject 属性で参照されます。

保留中の変更を保持するために、アプリケーションは PendingPublicObject 属性で参照されるドキュメントを次の手順でフェッチし、更新できます。

1. ドキュメントをフェッチします。
2. ドキュメントがバージョニング対象であることを確認し、VersionSeries をフェッチします。

3. VersionSeries が現行のユーザーにより予約済みであることを確認し、PendingPublicObject をフェッチします。
4. 変更内容を PendingPublicObject に保持するように、DocumentDefinition を作成します。
5. 変更内容で PendingPublicObject を更新します。

例 14-4 保留中の変更の保持

1. アプリケーションですでにドキュメントを取得しているとします。

```
Document doc .....
```

2. ドキュメントがバージョニング対象であることを確認し、VersionSeries をフェッチします。

```
if (doc.isVersioned())
{
  Family.family = doc.getFamily();
  VersionSeries vs = family.getPrimaryVersionSeries();
```

3. ドキュメントが現行のユーザーによりチェックアウトされていることを確認し、PendingPublicObject をフェッチします。

```
    if (vs.isReservedByCurrentUser() )
    {
        PublicObject ppo = vs.getPendingPublicObject();
```

4. 変更内容を保持する DocumentDefinition を作成します。

```
        DocumentDefinition dd = (DocumentDefinition) ppo.getDefinition();
        dd.setContent("Changed content.");
```

5. 変更内容で PendingPublicObject を更新します。

```
        ppo.update(dd);
    }
}
```

新バージョンのチェックイン

作成者がドキュメントを変更し終わった後に、変更内容をドキュメントの新バージョンとしてチェックインできます。ドキュメントの新バージョンをチェックインする手順は、次のとおりです。

1. ドキュメントをフェッチします。
2. ドキュメントがバージョニング対象であることを確認し、VersionSeries をフェッチします。

3. VersionSeries が現行のユーザーにより予約済みであることを確認し、PendingPublicObject をフェッチします。
4. PendingPublicObject が NULL でないことを確認し、新バージョンの VersionDescriptionDefinition を作成します。
5. `newVersion()` メソッドをコールしてドキュメントの新バージョンを作成します。

例 14-5 新バージョンのチェックイン

1. アプリケーションですでにドキュメントを取得しているとします。

```
Document doc .....
```

2. ドキュメントがバージョニング対象であることを確認し、VersionSeries をフェッチします。

```
if (doc.isVersioned())
{
    Family family = doc.getFamily();
    VersionSeries vs = family.getPrimaryVersionSeries();
```

3. ドキュメントが現行のユーザーによりチェックアウトされていることを確認し、PendingPublicObject をフェッチします。

```
    if (vs.isReservedByCurrentUser() )
    {
       PublicObject ppo = vs.getPendingPublicObject();
```

4. PendingPublicObject が NULL でないことを確認し、新バージョンの VersionDescriptionDefinition を作成します。

```
       if (ppo != null)
       {
          vdd = new VersionDescriptionDefinition( session );
          vdd.setAttribute(vd.REVISIONCOMMENT_ATTRIBUTE,
              AttributeValue.newAttributeValue(
               "Updated document to incorporate review comments."));
```

5. ドキュメントの新バージョンを作成します。

```
          vd = vs.newVersion(vdd);
       }
    }
}
```

ドキュメントのチェックアウトの取消し

ドキュメントのチェックアウト中に、いつでもチェックアウトを取り消すことができます。チェックアウトを取り消すと、PendingPublicObject に一時的に保持されている変更内容が破棄され、他のユーザーがドキュメントをチェックアウトできるように VersionSeries が解放されます。

ドキュメントのチェックアウトを取り消す手順は、次のとおりです。

1. ドキュメントをフェッチします。
2. ドキュメントがバージョニング対象であることを確認し、VersionSeries をフェッチします。
3. VersionSeries が現行のユーザーにより予約されていることを確認します。
4. VersionSeries の `unReserve()` メソッドをコールします。

例 14-6 ドキュメントのチェックアウトの取消し

1. アプリケーションですでにドキュメントを取得しているとします。

```
Document doc .....
```

2. ドキュメントがバージョニング対象であることを確認し、VersionSeries をフェッチします。

```
if (doc.isVersioned())
{
    Family family = doc.getFamily();
    VersionSeries vs = family.getPrimaryVersionSeries();
```

3. ドキュメントが現行のユーザーによりチェックアウトされていることを確認します。

```
    if (vs.isReservedByCurrentUser() )
    {
```

4. VersionSeries の unReserve() メソッドをコールしてチェックアウトを取り消します。

```
      vs.unReserve();
    }
}
```

ドキュメント履歴の表示

ドキュメントがバージョニングされた後は、そのドキュメントに対して行われたすべての変更の履歴を取得し、保存されているすべてのドキュメント・バージョンにアクセスできます。ドキュメントの履歴を表示する手順は、次のとおりです。

1. ドキュメントをフェッチします。

2. ドキュメントがバージョニング対象であることを確認し、VersionSeries をフェッチします。
3. VersionSeries 内の VersionDescription をフェッチして履歴情報を選択します。

例 14-7 ドキュメント履歴の表示

1. アプリケーションですでにドキュメントを取得しているとします。

```
Document doc .....
```

2. ドキュメントがバージョニング対象であることを確認し、VersionSeries をフェッチします。

```
if (doc.isVersioned())
{
    Family family = doc.getFamily();
    VersionSeries vs = family.getPrimaryVersionSeries();
```

3. VersionSeries 内の VersionDescription をフェッチして履歴情報を選択します。

```
    VersionDescription[] versions = vs.getVersionDescriptions();

    Document docVersion;
    Date lastModifyDate;
    Long size;
    PublicObject version;
    String docVersionName, format, lastModifier, revisionComment;
    VersionDescription vd;

    for (int i = 0; i < versions.length; i++)
    {
      vd = versions[i];
      version = vd.getPublicObject();
      versionName = version.getName();
      lastModifier = version.getLastModifier().getName();
      lastModifyDate = version.getLastModifyDate();
      revisionComment = vd.getRevisionComment();

      if (docVersion instanceof Document)
      {
        docVersion = (Document) version;
        format = docVersion.getFormat().getName();
        size = docVersion.getContentSize();
      }
    }
}
```

特定のドキュメント・バージョンの削除

作成された各バージョンはリポジトリに保存されるため、ドキュメントをバージョニングすると、システムの記憶領域とパフォーマンスに影響します。エンド・ユーザーに、不要なバージョンを削除してリポジトリ内の情報量を減らすように許可できます。

ドキュメント・バージョンを削除する手順は、次のとおりです。

1. 削除するバージョンの VersionDescription をフェッチします。
2. オプションで、削除前に VersionDescription が最新バージョンでないことを確認します。
3. VersionDescription を削除します。

例 14-8 特定のドキュメント・バージョンの削除

1. アプリケーションが削除するバージョンの VersionDescription をすでに取得しているとします。

```
VersionDescription vd .....
```

2. オプションで、VersionDescription が VersionSeries 内の最新バージョンでないことを確認します。

```
if (!vd.isLatestVersionDescription())
{
```

3. VersionDescription を削除します。

```
    vd.free();
}
```

バージョニング対象ドキュメントの削除

バージョン対象ドキュメントを削除するのは、バージョニング対象外ドキュメントの場合と同様に簡単です。ドキュメントの Family で free() メソッドがコールされると、Oracle 9*i*FS は自動的にそのバージョニング・コンポーネントを削除します。これには、VersionSeries、VersionDescriptions およびそのドキュメントのバージョンを表す Document の全インスタンスが含まれます。

例 14-9 バージョニング対象ドキュメントの削除

1. アプリケーションですでにドキュメントの Family を取得しているとします。

```
Document doc .....
Family family = doc.getFamily();
```

2. Family がロックされていないことを確認します。

```
if (!family.isLocked())
{
```

3. Family を削除します。

```
    family.free();
}
```

フォルダへのバージョニング対象ドキュメントの格納

フォルダを使用すると、ドキュメントへのアクセスを編成し、リポジトリを通じてブラウズできます。バージョニング対象外のドキュメントをフォルダに格納すると、フォルダはそのドキュメント、つまり Document クラスのインスタンスを直接参照します。ただし、ドキュメントがバージョニング対象の場合、フォルダはそのドキュメントのすべてのコンポーネント、つまり Family、VersionSeries または VersionDescription を参照できます。これは、様々な目的でドキュメントのライフサイクル中の特定時点に簡単にアクセスできる方法を、ユーザーに提供する場合に役立ちます。

たとえば、Release 1.0 Documentation Set フォルダが、Oracle 9*i*FS 1.0 でリリースされたドキュメントを含むように作成されているとします。このフォルダは、そのリリースで配信されたドキュメントの特定バージョンである Version 3 を参照します。そのためには、Version 3 を表す VersionDescription をフォルダに格納します。

注意： このリリースの Oracle 9*i*FS クライアントでは、VersionSeries および VersionDescription のフォルダ格納はサポートされていません。これらのオブジェクトをフォルダに格納するカスタム・アプリケーションを構築する場合は、デフォルトの Oracle 9*i*FS クライアントのかわりに、フォルダに格納されたオブジェクトにアクセスするカスタム・ユーザー・インタフェースを構築する必要があります。

例 14-10 フォルダへの特定のドキュメント・バージョンの格納

1. フォルダに格納するドキュメント・バージョンの VersionDescription を取得します。

```
VersionDescription vd = ...
```

2. フォルダを取得します。

```
Folder f = .....
```

3. Folder に VersionDescription を追加します。

```
f.addItem(vd);
```

Future Documentation フォルダを作成し、Oracle 9*i*FS の次のリリース用に開発中のドキュメント・セットを格納できます。この場合、このフォルダは Work-in-Progress VersionSeries を参照します。作成者は、ここからドキュメントにアクセスし、Work-in-Progress VersionSeries の最新バージョンを取得できます。

例 14-11 フォルダへの特定の VersionSeries の格納

1. フォルダを取得します。

```
Folder f = ....
```

2. フォルダに格納する VersionSeries を取得します。

```
Document doc = ....
Family family = doc.getFamily();
VersionSeries[] vss = family.getVersionSeries();

int i;
String vsName;
VersionSeries vs;

for (i = 0; i < vss.length; i++)
{
  vs = vss[i];
  vsName = vs.getName();

  if (vsName equals ("Work-in-Progress"))
  {
```

3. Folder に VersionSeries を追加します。

```
    f.addItem(vs);
    break;
  }
}
```

ユーザーがドキュメントの最新の発行済みバージョンに簡単にアクセスできるように、Current Documentation フォルダを作成できます。そのためには、ドキュメントの Family をフォルダに格納します。一般のコンシューマは、ドキュメントのプライマリ・バージョン・シリーズのデフォルト・バージョン Published VersionSeries として指定されているドキュメント・バージョンにアクセスできます。

例 14-12 フォルダへのドキュメントの Family の格納

1. フォルダを取得します。

```
Folder f = ....
```

2. ドキュメントの Family を取得します。

```
Document doc = ....
Family family = doc.getFamily();
```

3. Folder に Family を追加します。

```
f.addItem(f);
```

ドキュメントをバージョニング対象にすると、ドキュメントを含むすべてのフォルダを、そのドキュメントの Family を参照するように更新できます。ドキュメントの Family を参照すると、エンド・ユーザーはドキュメントの最初のバージョンではなく最新バージョンを取得することが保証されます。

例 14-13 ドキュメントをバージョニング対象にする場合のフォルダの更新

1. アプリケーションがすでにドキュメントを取得しているとします。

```
Document doc = ....
```

2. ドキュメントをバージョニング対象にします。

```
FamilyDefinition fd = new FamilyDefinition ( session );
fd.setAttribute( PublicObject.NAME_ATTRIBUTE ,
                 AttributeValue.newAttributeValue ( po.getName() ) );
AccessControlList acl = doc.getAcl();
fd.setAttribute(PublicObject.ACL_ATTRIBUTE,
                AttributeValue.newAttributeValue(acl));

VersionSeriesDefinition vsd = new VersionSeriesDefinition(session);
vsd.setFamilyDefinition(fd);

VersionDescriptionDefinition vdd =
      new VersionDescriptionDefinition(session);
vdd.setPublicObject(doc);
vdd.setVersionSeriesDefinition(vsd);
vdd.setOwnerBasedOnPublicObjectOption ( true );
```

3. トランザクションを開始します。

```
Transaction transaction = session.beginTransaction();
```

4. try ブロックを開始して操作をテストし、例外をキャッチします。

```
try
{
   VersionDescription vd =
     (VersionDescription) session.createPublicObject(vdd);
```

5. ドキュメントの Family を取得します。

```
VersionSeries vs = vd.getVersionSeries();
Family family = vs.getFamily();
```

6. ドキュメントを参照するすべてのフォルダをフェッチします。

```
Folder fldr;
Folder[] fldrs =
   doc.getFolderReferences();
```

7. 各フォルダを、Document インスタンスのかわりにドキュメントの Family を参照するように更新します。

```
   for(i=0; i < fldrs.length; i++)
   {
      fldr = fldrs[i];
      fldr.removeItem(doc);
      fldr.addItem(family);
   }

   session.completeTransaction(transaction);
}
catch (IfsException e)
{
  session.abortTransaction(transaction);
  throw e;
}
```

注意：

- Oracle 9*i*FS で VersionDescription、VersionSeries または Family からアクセスされたときに戻すドキュメントのバージョンを解決する方法は、「バージョニング対象ドキュメントの解決」を参照してください。
- 例 14-13 では、ドキュメントがバージョニング対象のときに、すべてのフォルダが更新されることを保証するようにトランザクションを使用しています。Oracle 9*i*FS のトランザクション管理機能の使用方法は、第 16 章「セッションおよびトランザクションの管理」を参照してください。

バージョニング対象ドキュメントの解決

ユーザーがバージョニング対象ドキュメントの表示を試みる場合に、現行の状態を表すのはどのバージョンであるかを考えます。Oracle 9*i*FS のバージョニング・モデルでは、ドキュメントの現行の状態の解決方法を制御できます。また、アプリケーションでは、ドキュメント

の状態がバージョニング・モデルがアクセス中のノードに応じて異なる方法で解決されるように、バージョニング対象ドキュメントを構成できます。

Oracle 9*i*FS には、特定バージョン（VersionDescription）、バージョンの特定シリーズ（VersionSeries）または「有効ドキュメント」（Family）の現行の状態を表すドキュメントを自動的に解決できるように、`getResolvedPublicObject()` メソッドが用意されています。アプリケーションでは、Family および VersionSeries の特定の属性を使用して、ドキュメントの状態の解決方法を制御できます。

`getResolvedPublicObject()` メソッドは、バージョニング対象ドキュメントの各コンポーネントの解決に使用される場合に、次のロジックを使用します。

- **VersionDescription:** VersionDescription オブジェクトの `getResolvedPublicObject()` メソッドがコールされると、Oracle 9*i*FS は VersionDescription の PublicObject 属性で参照されるドキュメントを戻します。
- **VersionSeries:** `getResolvedPublicObject()` メソッドが VersionSeries でコールされると、Oracle 9*i*FS は VersionSeries の DefaultVersionDescription 属性で参照される VersionDescription について、解決済みの PublicObject を戻します。DefaultVersionDescription 属性は、バージョニング・アプリケーションで明示的に設定する必要があります。この属性が設定されていない場合、Oracle 9*i*FS は VersionSeries の LastVersionDescription 属性で参照される VersionDescription について、解決済みの PublicObject を戻します。LastVersionDescription 属性は Oracle 9*i*FS によりメンテナンスされ、VersionSeries 内の最後の VersionDescription を参照します。明示的には設定できません。
- **Family:** `getResolvedPublicObject()` メソッドがドキュメントの Family でコールされると、Oracle 9*i*FS は Family の DefaultVersionDescription 属性で参照される VersionDescription について、解決済みの PublicObject を戻します。DefaultVersionDescription 属性は、バージョニング・アプリケーションで明示的に設定する必要があります。この属性が設定されていない場合、Oracle 9*i*FS は Family の PrimaryVersionSeries について解決済み PublicObject を戻します。

例 14-14 に、バージョニング対象ドキュメントの各コンポーネントで `getResolvedPublicObject()` メソッドを使用する方法を示します。

例 14-14 バージョニング対象ドキュメントの PublicObject の解決

1. アプリケーションがすでにドキュメントを取得しているとします。

```
Document doc = ....
```

2. Document インスタンスの getResolvedPublicObject() メソッドをコールすると、それ自体のみが戻されます。

```
PublicObject rpo = doc.getResolvedPublicObject();
```

3. ドキュメントの Family の getResolvedPublicObject() メソッドをコールすると、Family の最後の DefaultVersionDescription 属性で参照される Document インスタンス、PrimaryVersionSeries の DefaultVersionDescription または PrimaryVersionSeries 内の最後の VersionDescription が戻されます。

```
Family family = doc.getFamily();
PublicObject rpo = family.getResolvedPublicObject();
```

4. VersionSeries の getResolvedPublicObject() メソッドをコールすると、VersionSeries の DefaultVersionDescription が参照する Document インスタンスまたは VersionSeries 内の LastVersionDescription が戻されます。

```
VersionSeries vs = doc.getVersionSeries();
PublicObject rpo = vs.getResolvedPublicObject();
```

5. VersionDescription の getResolvedPublicObject() メソッドをコールすると、VersionDescription で参照される Document インスタンスが戻されます。

```
VersionDescription vd = vs.getLastVersionDescription();
PublicObject rpo = vd.getResolvedPublicObject();
```

サンプル・コード

Oracle 9*i*FS は、Java API の操作開始の出発点として役立つサンプル・コードとともにインストールされます。この API サンプル・コードは、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/api` ディレクトリにあります。

表 14-5 に、この章に関連する API サンプル・コードを示します。

表 14-5 API サンプル・コード

クラス	使用方法
`VersioningSample.java`	ドキュメントをバージョニングします。

15

セキュリティ

この章では、Oracle 9*i*FS で情報を保護する方法について説明します。この章の内容は、次のとおりです。

- セキュリティの概要
- リポジトリへのアクセスの管理
- PublicObject へのアクセスの管理
- ClassObject へのアクセスの管理
- オブジェクトへの暗黙的なアクセス権の付与
- サンプル・コード

セキュリティの概要

どのようなコンテンツ管理アプリケーションの場合も、根本的な要件は情報へのアクセス方法を保護することです。Oracle 9*i*FS には確実なインタフェース・セットが用意されており、アプリケーションで情報の保護についてファイングレイン・アクセス・コントロールを行い、エンド・ユーザーが情報にアクセスできるような広範囲のセキュリティ・レベルを定義できます。

Oracle 9*i*FS では、セキュリティが次の 2 つのレベルで管理されます。

- リポジトリ・レベルのセキュリティ
- オブジェクト・レベルのセキュリティ

リポジトリ・レベルのセキュリティ

ユーザーが Oracle 9*i*FS に格納されている情報へのアクセスを試みると、Oracle 9*i*FS は最初にそのユーザーがリポジトリへのアクセス権を持っているかどうかを判断します。Oracle 9*i*FS では、リポジトリにアクセスできるユーザー全員のディレクトリがメンテナンスされます。ユーザーに対してリポジトリへのアクセスを許可する前に、Oracle 9*i*FS はそのユーザーの資格証明を認証します。認証を受けたユーザーは、情報について定義されたオブジェクト・レベルのセキュリティ制約の範囲内で、リポジトリ内の情報をブラウズ、検索および操作できます。

Oracle 9*i*FS では、各ユーザーによるリポジトリの操作方法も制御されます。また、ユーザーの識別名、ホーム・フォルダ、コンテンツ・クオータおよびデフォルト ACL など、ユーザーの作業環境のプロファイルがメンテナンスされます。このプロファイルを拡張し、アプリケーションに固有の他の作業環境を含めることができます。

Oracle 9*i*FS を使用すると、ユーザー・グループを定義して作成し、ユーザーによる情報へのアクセスを簡単に管理できます。グループにより、共通の要素を持つユーザーの集合が編成されます。グループに他のグループを含めて、ユーザーの包含階層を作成できます。Oracle 9*i*FS を拡張し、カスタム・タイプのグループを定義して、そのタイプのグループに関連する属性と動作を持たせることができます。ユーザー・グループを定義して作成した後は、ユーザーをグループに割り当てることで、一度に複数のユーザーに権限とコンテンツ・クオータを割り当てることができます。

オブジェクト・レベルのセキュリティ

Oracle 9*i*FS では、ユーザーがリポジトリ内の各オブジェクトにアクセスする方法を制御できます。オブジェクトへのアクセスを制御するには、次の 3 つの方法があります。

- **PublicObject へのアクセスの管理：** PublicObject に対するアクセス権を定義して、特定のドキュメント、フォルダおよび他のタイプの情報を検出、読込み、変更および削除できるユーザーを制御できます。
- **ClassObject へのアクセスの管理：** ClassObject に対するアクセス権を定義して、特定タイプの情報を作成したり検索できるユーザーを制御できます。

- **オブジェクトへの暗黙的なアクセス権の付与：** ユーザーをオブジェクトの所有者に指定して、オブジェクトへのすべてのアクセス権を暗黙的に付与したり、ユーザーをシステム管理者にして、リポジトリ内の全情報へのすべてのアクセス権を暗黙的に付与することもできます。

PublicObject へのアクセスの管理

PublicObject に対するアクセス権は、AccessControlList で管理します。ユーザーが PublicObject へのアクセスを試みるたびに、Oracle 9*i*FS はまず対応付けられている AccessControlList をチェックして、そのユーザーに適切な権限が付与されているかどうかを確認します。

AccessControlList は、PublicObject へのアクセス権が付与または取り消されているユーザーおよびグループのリストです。ユーザーまたはグループのアクセス権を付与または取り消すたびに、AccessControlList にエントリが追加されます。Oracle 9*i*FS では、AccessControlList 内で作成された順に AccessControlEntry が集計され、PublicObject へのユーザーの有効アクセスが解決されます。各 AccessControlList は Oracle 9*i*FS 内で独立して定義され、1 つ以上の PublicObject に適用されます。このため、PublicObject について共通のセキュリティ・レベルを一元的に管理できます。

いずれのユーザーも、AccessControlList を作成できます。AccessControlList の作成時に、ユーザーはそれを PublicObject に適用して AccessControlList のエントリを管理できる他のユーザーを決定できます。これにより、ユーザーは各自の情報へのアクセス管理に使用する個人用 AccessControlList を作成したり、AccessControlList を共有にして他のユーザーと共同で作業する情報へのアクセスを管理できます。また、管理者は特殊なタイプの AccessControlList である SystemAccessControlList を作成し、Oracle 9*i*FS ユーザーはリポジトリ内の情報に適用できるシステム単位のセキュリティ・レベルを定義できます。

AccessControlList を使用すると、Discover、GetContent、SetAttribute、SetContent、および Delete など、Oracle 9*i*FS により定義される標準的な権限を付与または取り消すことができますが、Oracle 9*i*FS ではカスタム権限も定義できます。拡張権限を定義して、カスタム・アプリケーションの動作へのアクセスを管理できます。また、ユーザーが PublicObject のセキュリティを簡単に管理できるように、権限をグループ化して、より広範囲なアクセス・レベルを定義することも可能です。

ClassObject へのアクセスの管理

第 5 章「コンテンツ・タイプと属性の拡張」で説明したように、各コンテンツ・タイプは ClassObject により定義されます。ClassObject の ClassAccessControlList を定義して、コンテンツ・タイプのインスタンスを作成したり検索できるユーザーおよびグループを制御できます。

暗黙的なアクセス制御

ユーザーには、オブジェクト所有者またはシステム管理者になることで、リポジトリ内の情報へのすべてのアクセス権が暗黙的に付与されます。Oracle 9*i*FS 内でオブジェクト所有者として指定されたユーザーには、そのオブジェクトへのすべてのアクセス権が付与されます。

管理者として指定されたユーザーには、リポジトリ内の全情報へのすべてのアクセス権が付与されます。アクセス権は Oracle 9*i*FS により暗黙的に付与されるため、オブジェクトの AccessControlList により適用されるのではなく、それにかわって適用されます。

リポジトリへのアクセスの管理

この項では、Oracle 9*i*FS でディレクトリを使用してリポジトリにアクセスする方法について説明します。この項の内容は、次のとおりです。

- ディレクトリの機能
- ディレクトリ構造
- DirectoryUser および PrimaryUserProfile の作成、変更および削除
- ExtendedUserProfile クラスの定義
- DirectoryUser への ExtendedUserProfile の適用
- DirectoryGroup の作成、変更および削除
- カスタム・グループ・クラスの定義
- カスタム・グループの作成、変更および削除
- DirectoryUser の認証

ディレクトリの機能

Oracle 9*i*FS では、リポジトリにアクセスできるユーザー全員のディレクトリがメンテナンスされます。ディレクトリには、ユーザーの識別名、管理者かどうか、認証方法、リポジトリでの作業方法など、ユーザーに関する情報が格納されます。また、Oracle 9*i*FS のディレクトリではグループが管理されます。グループを使用すると、ユーザーを階層形式で編成し、分類し、複数のユーザーが共通して保持するアクセス権限を割り当てることができます。

ユーザーの認証

ユーザーに対してリポジトリへのアクセスを許可する前に、Oracle 9*i*FS はそのユーザーの資格証明を認証します。最初に、ユーザーは識別名によりディレクトリ内で識別されます。次に、ユーザーのパスワードが資格証明マネージャと対照して検証されます。各ユーザー・アカウントは、そのユーザーのパスワードの認証に使用される資格証明マネージャに対応付けられます。このリリースの Oracle 9*i*FS には、ユーザー全員の認証に使用されるビルトインの資格証明マネージャが用意されています。今後のリリースでは、外部資格証明マネージャがサポートされます。

ユーザー・プロファイルの作成

Oracle 9*i*FS 内の各ユーザー・アカウントは、そのユーザーのリポジトリでの作業に関する設定項目が格納される 1 つ以上のプロファイルに対応付けられています。Oracle 9*i*FS では、プ

ライマリ・ユーザー・プロファイルに格納される標準設定項目セットが定義されます。この標準設定項目には、ユーザーのホーム・フォルダ（ユーザーの個人用作業領域）、コンテンツ・クオータ（ユーザーに割り当てられた記憶領域の量）およびデフォルト ACL（ユーザーが作成するオブジェクトにデフォルトで適用される AccessControlList）が含まれています。

Oracle 9*i*FS では、拡張ユーザー・プロファイルを作成して、ユーザーに関するカスタム設定項目を格納できます。たとえば、カスタム Web アプリケーションでユーザーのリポジトリ・ビューを個別設定する設定項目を格納できます。

ユーザーのグループ化

Oracle 9*i*FS では、ユーザーを階層形式でグループとして編成できます。グループは、共通の要素を持つユーザーの集合を表します。たとえば、ユーザーが持つ様々なロールを表すグループ（Manager、Developer、Writer など）を作成する方法があります。また、会社の部門やプロジェクトを表すグループ（Marketing、Sales、Development など）を作成する方法もあります。グループに他のグループを含めて、ユーザーの包含階層を作成できます。たとえば、Oracle グループに Server Technologies グループを含め、Server Technologies グループには Oracle 9*i*FS、Oracle Text および Oracle *inter*Media グループを含めることができます。

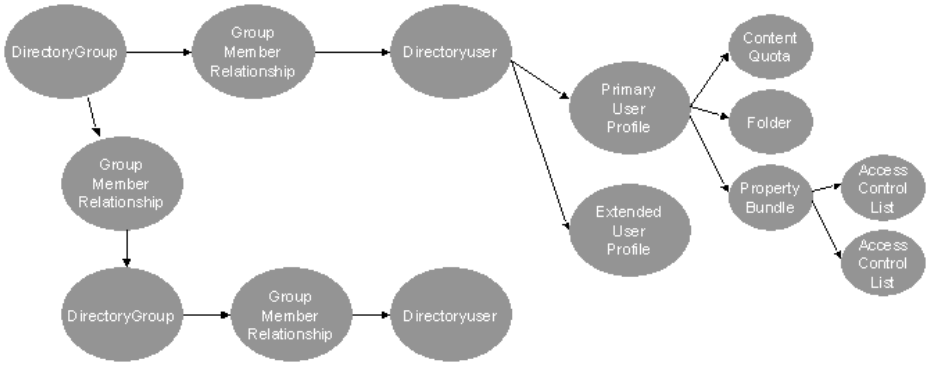
グループにより、ユーザーの集合に関する情報にセキュリティを簡単に施行できます。グループには、リポジトリ内のオブジェクトへのアクセス権を付与できます。たとえば、AccessControlEntry の Grantee として Manager グループを指定する AccessControlList を作成し、すべてのマネージャにリポジトリ内のドキュメントへの読取りアクセス権を付与できます。これにより、アクセス権をユーザーごとに定義するのではなく、そのグループのユーザー全員に対して一度にすばやく定義できます。グループのユーザーを追加または削除すると、そのグループの現行のメンバーにアクセス権が動的に付与されます。

Oracle 9*i*FS では、特別な属性と動作を持つカスタム・グループ・タイプを定義できます。たとえば、属性 VicePresident および MissionStatement を持つ Organization グループ・タイプを作成できます。グループ・タイプは、属性と動作を継承するように階層形式で定義できます。たとえば、属性 Address および MainPhoneNumber を持つグループ・タイプ Company を作成してから、この 2 つの属性を継承するサブグループ・タイプ Partner を作成し、属性 ContractNumber を追加できます。

ディレクトリ構造

Oracle 9*i*FS には、ユーザーとグループの管理用にクラス・セットが用意されています。クラス `DirectoryUser` および `DirectoryGroup` は PublicObject の派生クラスで、Oracle 9*i*FS 内の情報を操作するユーザーとグループを表すためのプライマリ・インタフェースとして機能します。この 2 つのクラスは、リポジトリでの操作に関する各ユーザーの設定項目と、ユーザーをグループに関連付ける方法を表すための、少数の SystemObject クラスでサポートされます。図 15-1 に、Oracle 9*i*FS 内のディレクトリ情報の構造を示します。

図 15-1 ディレクトリ・オブジェクト・モデル



DirectoryUser

各 Oracle 9*i*FS ユーザーは、DirectoryUser クラスのインスタンスにより表されます。DirectoryUser は、ユーザーの管理権限の割当て、ユーザーへの一意名の割当て、ユーザー認証に使用する資格証明マネージャの指定など、ユーザーを管理するための属性とメソッドを持っています。

表 15-1 に、DirectoryUser で最も使用頻度の高い属性とメソッドを示します。

表 15-1 DirectoryUser の属性およびメソッド

AdminEnabled isAdminEnabled() setAdminEnabled()	ユーザーが Oracle 9*i*FS システムでの管理権限を持っているかどうかを判断します。
CredentialManager getCredentialManager() setCredentialManager()	ユーザーが Oracle 9*i*FS システムにログインするときに、ユーザーの資格証明の認証に使用される資格証明マネージャを判断します。
DistinguishedName UniqueName getDistinguishedName() setDistinguishedName()	システム内でユーザーを一意に識別します。

ユーザー・プロファイル（プライマリおよび拡張）

各 DirectoryUser は、ユーザーのホーム・フォルダやコンテンツ・クオータなど、ユーザーによるリポジトリの操作方法に関する設定項目のプロファイルを持っています。Oracle 9*i*FS では、次の 3 つのクラスを使用してユーザー設定項目が管理されます。

- **PrimaryUserProfile** は、ユーザーのホーム・フォルダやコンテンツ・クオータなど、Oracle 9*i*FS でデフォルトで定義されている設定項目を管理するための属性とメソッドを持ちます。
- **ExtendedUserProfile** には、カスタム設定項目を管理するための基本コンテンツ・タイプが用意されています。
- **ContentQuota** は、ユーザーが Oracle 9*i*FS リポジトリに格納できるコンテンツの量を管理するために使用されます。

表 15-2 に、PrimaryUserProfile で最も使用頻度の高い属性とメソッドを示します。

表 15-2 PrimaryUserProfile の属性およびメソッド

ContentQuota getContentQuota() setContentQuota()	リポジトリへのコンテンツ格納に関するユーザーのクオータを判断します。
DefaultACLs getDefaultACLs() setDefaultACLs()	ユーザーがリポジトリ内で作成する情報にデフォルトで適用される AccessControlList を判断します。 DefaultACLs 属性は、デフォルトの AccessControlList を含む PropertyBundle を参照します。
HomeFolder getHomeFolder() setHomeFolder()	ユーザーの個人用記憶領域として機能するフォルダを判断します。ユーザーが Oracle 9*i*FS クライアントにログインすると、デフォルトでホーム・フォルダに置かれます。 HomeFolder 属性は、ユーザーのホーム・フォルダとして機能する Folder のインスタンスを参照します。

表 15-3 に、ExtendedUserProfile で最も使用頻度の高い属性とメソッドを示します。

表 15-3 ExtendedUserProfile の属性およびメソッド

Application getApplication() setApplication()	ExtendedUserProfile が適用されるカスタム・アプリケーションを示します。同じ Oracle 9*i*FS システム用に複数のカスタム・アプリケーションを構築できます。各ユーザーは、各アプリケーションでの操作用に異なるプロファイルを持つことができます。

表 15-4 に、ContentQuota で最も使用頻度の高い属性とメソッドを示します。

表 15-4 ContentQuota の属性およびメソッド

AllocatedStorage getAllocatedStorage() setAllocatedStorage()	ユーザー用の記憶領域を割り当てます。
ConsumedStorage calculateConsumedStorage() getConsumedStorage()	ユーザーがクオータのうち消費した領域の量を判断します。
AssociatedPublicObject getAssociatedPublicObject()	ContentQuota が適用される PublicObject を指定します。
Enabled isEnabled() setEnabled()	ユーザーのコンテンツ・クオータが有効かどうかを判断します。管理者は、ユーザーのコンテンツ・クオータを無効化して、リポジトリ内で無制限の記憶領域を与えることができます。

DirectoryGroup

Oracle 9*i*FS では、グループは DirectoryGroup および GroupMemberRelationship という 2 つのコンテンツ・タイプで表されます。DirectoryGroup は PublicObject の派生クラスのため、グループのメンバーシップを管理するためのプライマリ・インタフェースとして機能します。GroupMemberRelationship は SystemObject の派生クラスであり、ユーザーとグループ間の関係を表します。GroupMemberRelationship のインスタンスは、各ユーザーまたはグループのメンバーシップを表すように作成されます。

DirectoryGroup を拡張してグループを分類し、そのタイプのグループ専用の特別な属性と動作を追加できます。たとえば、DirectoryGroup を拡張する Organization グループを作成し、属性 VicePresident および MissionStatement を含めることができます。また、GroupMemberRelationship を拡張し、TeamRole など、Organization グループの各ユーザーのメンバーシップに関する特別なメタデータを管理できます。

表 15-5 に、DirectoryGroup で最も使用頻度の高い属性とメソッドを示します。

表 15-5 DirectoryGroup の属性およびメソッド

addMember() addMembers()	グループのメンバーとしてユーザーまたはグループ、あるいはその両方を追加します。

表 15-5 DirectoryGroup の属性およびメソッド（続く）

getAllMembers() getAllUserMembers() getDirectMembers() isMember()	グループのメンバーであるユーザーまたはグループ、あるいはその両方を判断します。直接のメンバーは、このグループに含まれる他のグループに属しているユーザーやグループではなく、このグループに直接関連しているユーザーまたはグループです。
removeMember() removeMembers()	グループからユーザーまたはグループ、あるいはその両方を削除します。

表 15-6 に、GroupMemberRelationship で最も使用頻度の高い属性とメソッドを示します。

表 15-6 GroupMemberRelationship の属性およびメソッド

LeftObject getLeftObject()	LeftObject 属性は DirectoryGroup を参照します。
RightObject getRightObject()	RightObject 属性は、DirectoryGroup に属しているユーザーまたはグループを参照します。

DirectoryUser および PrimaryUserProfile の作成、変更および削除

Oracle 9*i*FS には、DirectoryUser を PrimaryUserProfile とともに作成、変更および削除できるように、次の 3 つの方法が用意されています。

- **XML:** Oracle 9*i*FS へのインポート時に DirectoryObject を自動的に定義する XML ファイルを作成します。
- **Java:** Oracle 9*i*FS Java API 経由で DirectoryObject をプログラムで定義します。
- **Oracle 9*i*FS Manager:** Graphical User Interface（GUI）を使用して DirectoryObject を定義します。

XML: XML ファイルを使用して、DirectoryUser および PrimaryUserProfile を作成したり変更できます。XML ファイルが Oracle 9*i*FS にインポートされると、IfsSimpleXmlParser は自動的にそれを使用して DirectoryUser インスタンスを作成または変更します。IfsSimpleXmlParser は、DirectoryUser を簡単に作成できるようにする特別な XML スキーマ SimpleUser をサポートします。SimpleUser スキーマを使用すると、DirectoryUser および PrimaryUserProfile クラスに対応するスキーマの 2 つの要素を含む XML ファイルを作成するのではなく、DirectoryUser 属性と PrimaryUserProfile 属性を同じ要素内で指定できます。ただし、DirectoryUser および PrimaryUserProfile を XML で更新するには、XML ファイルのスキーマが、DirectoryUser クラスまたは PrimaryUserProfile クラスに対応している必要があります。構文規則は、第 10 章「XML および Oracle Internet File System」を参照してください。

注意： Oracle 9*i*FS では、XML を使用した DirectoryUser の削除はサポートされていません。

XML を使用して DirectoryUser および PrimaryUserProfile を作成する方法と変更する方法は、例 15-1 および例 15-2 を参照してください。

Java: Oracle 9*i*FS Java API を使用すると、DirectoryUser および PrimaryUserProfile をプログラムで作成、変更および削除できます。Oracle 9*i*FS Java API のパッケージ `oracle.ifs.adk.user` には特殊なヘルパー・クラス UserManager が含まれており、DirectoryUser を PrimaryUserProfile とともに簡単に作成したり削除できます。作成後に DirectoryUser および PrimaryUserProfile を変更するには、それぞれのクラスで定義されているメソッドを直接操作します。

注意： UserManager クラスの詳細は、Oracle 9*i*FS の Javadoc を参照してください。

Java API を使用して DirectoryUser および PrimaryUserProfile を作成、変更および削除する方法は、例 15-4、例 15-5 および例 15-6 を参照してください。

Oracle 9*i*FS クライアント： Oracle 9*i*FS クライアントには、DirectoryUser および PrimaryUserProfile の作成用の GUI が用意されています。

注意： Oracle 9*i*FS クライアントで DirectoryUser を定義する手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

例 15-1 XML を使用した DirectoryUser の作成

```
<?xml version = '1.0' standalone = 'yes'?>
<SIMPLEUSER>
  <USERNAME>gking</USERNAME>
  <PASSWORD>ifs</PASSWORD>
  <DISTINGUISHEDNAMESUFFIX>.ambiguity.com</DISTINGUISHEDNAMESUFFIX>
  <ADMINENABLED>true</ADMINENABLED>
  <HOMEFOLDERROOT>/home</HOMEFOLDERROOT>
  <EMAILADDRESSSUFFIX>@ambiguity.com</EMAILADDRESSSUFFIX>
</SIMPLEUSER>
```

例 15-2 XML を使用した DirectoryUser の変更

```
<?xml version = '1.0' standalone = 'yes'?>
 <DIRECTORYUSER>
  <UPDATE RefType = "Name">gking</UPDATE>
  <DISTINGUISHEDNAME>gking.clarity.com</DISTINGUISHEDNAME>
 </DIRECTORYUSER>
```

例 15-3 XML を使用した PrimaryUserProfile の変更

```
<?xml version = '1.0' standalone = 'yes'?>
 <PRIMARYUSERPROFILE>
  <UPDATE RefType = "Name">gking Primary Profile</UPDATE>
  <HOMEFOLDER RefType = "Path">/home/system</HOMEFOLDER>
 </PRIMARYUSERPROFILE>
```

例 15-4 Java を使用したユーザーの作成

1. ユーザーを作成するには、管理モードを有効化する必要があります。

   ```
   session.setAdministrationMode(true);
   ```

2. UserManager の新規インスタンスを構成します。

   ```
   // Creating an user using all the default options;
   UserManager usermanager = new UserManager(session);
   ```

3. ユーザー・オプションを保持する HashTable を構成します。

   ```
   Hashtable userOptions = new Hashtable();

   userOptions.put(UserManager.ADMIN_ENABLED, new Boolean(true));
   userOptions.put(UserManager.EMAIL_ADDRESS, "gking@oracle.com");
   ```

4. ユーザーを作成します。

   ```
   DirectoryUser dUser = usermanager.createUser("gkings",
                                                "ifs",
                                                userOptions);
   ```

例 15-5 Java を使用した DirectoryUser の変更

1. 変更する DirectoryUser をフェッチします。

   ```
   Collection duColl = session.getDirectoryUserCollection();
   DirectoryUser dUser = (DirectoryUser) duColl.getItems("gking");
   ```

2. ユーザーを変更するには、管理モードを有効化する必要があります。

   ```
   session.setAdministrationMode(true);
   ```

3. ユーザーの PrimaryUserProfile を更新し、このユーザーが作成するすべてのドキュメントとフォルダに関するデフォルト ACL を設定します。

```
PrimaryUserProfile pup = dUser.getPrimaryUserProfile();

Collection aclColl = session.getSystemAccessControlListCollection();
AccessControlList defaultAcl = (AccessControlList) aclColl.getItems("PUBLIC");

pup.putDefaultAcl("DOCUMENT", defaultAcl);
pup.putDefaultAcl("FOLDER", defaultAcl);
```

例 15-6 Java を使用した DirectoryUser の削除

1. ユーザーを削除するには、管理モードを有効化する必要があります。

```
session.setAdministrationMode(true);
```

2. 削除する DirectoryUser インスタンスをフェッチします。

```
Collection duColl = session.getDirectoryUserCollection();
DirectoryUser dUser = (DirectoryUser) duColl.getItems("gking");
```

3. ユーザーの DistinguishedName を取得します。

```
String distinguishedName = dUser.getDistinguishedName();
```

4. UserManager オブジェクトをインスタンス化します。

```
UserManager manager = new UserManager(session);
```

5. 削除オプション・セットを設定します。

```
Hashtable options = new Hashtable();
```

6. ユーザーの DistinguishedName を指定します。

```
options.put(UserManager.DISTINGUISHED_NAME,
        AttributeValue.newAttributeValue(distinguishedName));
```

7. ユーザーの HomeFolder を削除するように指定します。

```
options.put(UserManager.FREE_HOME_FOLDER,
        AttributeValue.newAttributeValue(true));
```

8. 資格証明マネージャ・ユーザーを削除するように指定します。

```
options.put(UserManager.FREE_CREDENTIAL_MANAGER_USER,
        AttributeValue.newAttributeValue(true));
```

9. このユーザーが所有する PublicObject について、所有者をシステムに変更するように指定します。

```
options.put(UserManager.CHANGE_OWNER,
        AttributeValue.newAttributeValue(true));
options.put(UserManager.NEW_OWNER_USER_NAME,
        AttributeValue.newAttributeValue("system"));
```

10. ユーザーを削除します。

```
manager.deleteUser(distinguishedName, options);
```

ExtendedUserProfile クラスの定義

ExtendedUserProfile クラスを定義する手順は、カスタム・クラスを定義する場合と同じです。ExtendedUserProfile クラスを表す SchemaObject インスタンスのセットを作成します。その SchemaObject を編集または削除して、ExtendedUserProfile を変更したり削除できます。このような開発タスクは、Java API、XML ファイルまたは Oracle 9*i*FS Manager を使用して実行できます。

XML を使用して ExtendedUserProfile クラスを作成および変更する方法は、例 15-7 を参照してください。クラスの削除には、XML は使用できません。

Java を使用して ExtendedUserProfile クラスを定義、変更および削除する方法は、例 15-8 および例 15-12 を参照してください。

ExtendedUserProfile クラスの作成

新規 ExtendedUserProfile クラスを作成するには、Oracle 9*i*FS にデフォルトで用意されている ExtendedUserProfile クラスをサブクラス化します。拡張クラスは、カスタム・ユーザー設定項目を保持するための属性を持ちます。たとえば、ExtendedUserProfile を拡張して拡張属性 DefaultLayout および DefaultRole を含めるように、クラス CatalogUserProfile を作成できます。

ExtendedUserProfile クラスを定義する手順は、次のとおりです。

1. 管理権限を持っているかどうかを確認します。XML 構成ファイルを使用する場合は、そのファイルを管理者としてインポートします。Java API を使用する場合は、Oracle 9*i*FS でのセッション用に管理モードを有効化します。
2. ExtendedUserProfile クラスを表す ClassObject のインスタンスを作成します。
3. 各ユーザー設定項目を表す Attribute のインスタンスを作成します。
4. オプションで、ExtendedUserProfile クラス用のカスタム Java クラスを作成し、カスタム・メソッドを定義します。

注意： カスタム Java クラスの作成方法は、第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照してください。

例 15-7 XML を使用した ExtendedUserProfile クラスの作成

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <NAME>CatalogUserProfile</NAME>
    <SUPERCLASS RefType = "Name">ExtendedUserProfile</SUPERCLASS>
    <DESCRIPTION>User preferences for the Product Catalog.</DESCRIPTION>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>DefaultLayout</NAME>
            <DATATYPE>String</DATATYPE>
        </ATTRIBUTE>
        <ATTRIBUTE>
            <NAME>DefaultRole</NAME>
            <DATATYPE>String</DATATYPE>
        </ATTRIBUTE>
    </ATTRIBUTES>
</CLASSOBJECT>
```

注意： ExtendedUserProfile クラスの作成に使用する XML ファイルの各要素の詳細は、第 5 章「コンテンツ・タイプと属性の拡張」を参照してください。

例 15-8 Java を使用した ExtendedUserProfile クラスの作成

1. 管理モードを有効化します。

   ```
   session.setAdministrationMode(true);
   ```

2. ExtendedUserProfile を表す ClassObject の値を保持する ClassObjectDefinition を作成します。

   ```
   ClassObjectDefinition codef = new ClassObjectDefinition(session);
   codef.setAttributeByUpperCaseName(ClassObject.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("CatalogUserProfile"));
   ```

3. 新規クラスで ExtendedUserProfile を拡張するように指定します。

   ```
   ClassObject superClass = session.getClassObjectByName("EXTENDEDUSERPROFILE");
   codef.setSuperclass(superClass);
   ```

4. ExtendedUserProfile タイプに関連する属性ごとに AttributeDefinition を作成します。AttributeDefinition を ClassObjectDefinition に追加します。

```
AttributeDefinition attdef1 = new AttributeDefinition(session);
attdef1.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("DefaultLayout"));
attdef1.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_STRING));

codef.addAttributeDefinition(attdef1);

AttributeDefinition attdef2 = new AttributeDefinition(session);
attdef2.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("DefaultRole"));
attdef2.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_STRING));

codef.addAttributeDefinition(attdef2);
```

5. 新規 ExtendedUserProfile クラスの ClassObject および Attribute を作成します。

```
ClassObject eupCo = (ClassObject) session.createSchemaObject(codef);
```

ExtendedUserProfile クラスの変更

作成した ExtendedUserProfile クラスの属性を追加、削除および更新できます。

注意： ExtendedUserProfile クラスの属性の追加や削除には、XML を使用できません。この操作には、Oracle 9*i*FS Manager または Java API を使用する必要があります。

例 15-9 Java を使用した ExtendedUserProfile クラスの属性の追加、削除および更新

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. ExtendedUserProfile クラスを表す ClassObject を取得します。

```
ClassObject co = session.getClassObjectByName("CATALOGUSERPROFILE");
```

3. Attribute を ClassObject に追加します。

```
AttributeDefinition attdef1 = new AttributeDefinition(session);
attdef1.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
```

```
        AttributeValue.newAttributeValue("DefaultWorkArea"));
attdef1.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(
          Attribute.ATTRIBUTEDATATYPE_PUBLICOBJECT));
co.addAttribute(attdef1);
```

4. Attribute を ClassObject から削除します。

```
Attribute att2 = catco.getAttributeFromLabel("DEFAULTROLE");
co.removeAttribute(att2);
```

5. ClassObject の Attribute を更新します。

```
Attribute att3 = co.getAttributeFromLabel("DEFAULTWORKAREA");
Collection cdColl = session.getClassDomainCollection();
ClassDomain cd = (ClassDomain) cdColl.getItems("Folder...");
att3.setClassDomain(cd);
```

例 15-10 Java を使用した ExtendedUserProfile クラスの削除

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. ExtendedUserProfile クラスを表す ClassObject を取得します。

```
ClassObject co = session.getClassObjectByName("CATALOGUSERPROFILE");
```

3. `free()` メソッドをコールして ClassObject を削除します。

```
co.free();
```

DirectoryUser への ExtendedUserProfile の適用

ExtendedUserProfile クラスを定義すると、ExtendedUserProfile を DirectoryUser に適用できます。ExtendedUserProfile を適用するには、DirectoryUser を参照する ExtendedUserProfile クラスのインスタンスを作成し、ユーザーの設定項目を拡張属性に保持します。ExtendedUserProfile インスタンスでは、ExtendedUserProfile クラスから継承される Application 属性で、プロファイルを使用するアプリケーションを指定できます。その後は、Oracle 9*i*FS の他のオブジェクトの場合と同じ方法で、ExtendedUserProfile インスタンスを変更したり削除できます。このような開発タスクは、Java API、XML ファイルまたは Oracle 9*i*FS Manager を使用して実行できます。

XML を使用して ExtendedUserProfile インスタンスを作成する方法は、例 15-11 を参照してください。インスタンスの削除には、XML は使用できません。

Java を使用して ExtendedUserProfile インスタンスを作成および削除する方法は、それぞれ例 15-12 および例 15-13 を参照してください。

例 15-11 XML を使用した ExtendedUserProfile インスタンスの作成

```
<?xml version="1.0" standalone="yes"?>
<CATALOGUSERPROFILE>
  <DIRECTORYUSER RefType = "Name">gking</DIRECTORYUSER>
  <APPLICATION>ProductCatalog</APPLICATION>
  <DEFAULTLAYOUT>TreeView</DEFAULTLAYOUT>
  <DEFAULTROLE>Consumer</DEFAULTROLE>
</CATALOGUSERPROFILE>
```

例 15-12 Java を使用した ExtendedUserProfile インスタンスの作成

1. 拡張ユーザー・プロファイルを持つ DirectoryUser をフェッチします。

```
Collection duColl = session.getDirectoryUserCollection();
DirectoryUser dUser = (DirectoryUser) duColl.getItems("gking");
```

2. 新規 UserProfileDefinition を作成します。

```
UserProfileDefinition eupDef = new UserProfileDefinition(session);
```

3. 新規 ExtendedUserProfile インスタンスのタイプを指定します。

```
ClassObject co = session.getClassObjectByName("CATALOGUSERPROFILE");
eupDef.setClassObject(co);
```

4. ExtendedUserProfile の属性値を設定します。

```
eupDef.setAttributeByUpperCaseName(ExtendedUserProfile.DIRECTORYUSER_ATTRIBUTE,
     AttributeValue.newAttributeValue(dUser));
eupDef.setAttributeByUpperCaseName(ExtendedUserProfile.APPLICATION_ATTRIBUTE,
         AttributeValue.newAttributeValue(
           "ProductCatalog"));
eupDef.setAttributeByUpperCaseName("DEFAULTLAYOUT",
     AttributeValue.newAttributeValue("TreeView"));
eupDef.setAttributeByUpperCaseName("DEFAULTROLE",
     AttributeValue.newAttributeValue("Consumer"));
```

5. UserProfileDefinition を LibrarySession に渡して、新規 ExtendedUserProfile インスタンスを作成します。

```
ExtendedUserProfile eup = (ExtendedUserProfile) session.createPublicObject(eupDef);
```

例 15-13 Java を使用した ExtendedUserProfile インスタンスの削除

1. Selector を使用して ExtendedUserProfile インスタンスを取得します。

```
Selector s = new Selector(session);
s.setSearchClassname("CATALOGUSERPROFILE");
```

```
LibraryObject[] eups = s.getItems();
ExtendedUserProfile eup;
```

2. ExtendedUserProfile インスタンスを削除します。

```
int i;
int count = (eups == null) ? 0 : eups.length;

for (i = 0; i < count; i++)
{
   eup = (ExtendedUserProfile) eups[i];
   eup.free();
}
```

DirectoryGroup の作成、変更および削除

Oracle 9*i*FS には、DirectoryGroup を作成、変更および削除できるように、次の 3 つの方法が用意されています。

- **XML:** XML ファイルを使用して、DirectoryGroup を作成したり変更できます。XML ファイルが Oracle 9*i*FS にインポートされると、IfsSimpleXmlParser は自動的にそれを使用して DirectoryGroup インスタンスを作成または変更します。IfsSimpleXmlParser は、特別な XML タグ <MEMBERS> および <REF> をサポートしており、DirectoryUser を DirectoryGroup に簡単に追加できます。XML を使用して DirectoryGroup を作成および変更する方法は、例 15-14 および例 15-15 を参照してください。

  注意： Oracle 9*i*FS では、XML を使用した DirectoryGroup の削除はサポートされていません。

- **Java:** Oracle 9*i*FS Java API を使用すると、DirectoryGroup をプログラムで作成、変更および削除できます。Java API を使用して DirectoryGroup を作成、変更および削除する方法は、例 15-16、例 15-17 および例 15-18 を参照してください。
- **Oracle 9*i*FS クライアント：** Oracle 9*i*FS クライアントには、DirectoryGroup 作成用の GUI が用意されています。

  注意： Oracle 9*i*FS クライアントで DirectoryGroup を定義する手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

例 15-14 XML を使用した DirectoryGroup インスタンスの作成

```
<?xml version = '1.0' standalone = 'yes'?>
 <DIRECTORYGROUP>
  <NAME>Managers</NAME>
  <MEMBERS>
    <REF reftype='name'>msmith</REF>
    <REF reftype='name'>rjones</REF>
    <REF reftype='name'>tturner</REF>
  </MEMBERS>
 </DIRECTORYGROUP>
```

例 15-15 XML を使用した DirectoryGroup インスタンスの変更

```
<?xml version = '1.0' standalone = 'yes'?>
 <DIRECTORYGROUP>
  <UPDATE RefType = "Name">Managers</UPDATE>
  <MEMBERS>
    <REF RefType = 'Name'>csapo</REF>
  </MEMBERS>
 </DIRECTORYGROUP>
```

注意： DirectoryGroup から DirectoryUser を削除する場合は、XML を使用できません。この操作には、Oracle 9*i*FS クライアントまたは Java API を使用する必要があります。

例 15-16 Java を使用した DirectoryGroup インスタンスの作成

1. 新規の DirectoryGroupDefinition を作成します。

   ```
   DirectoryGroupDefinition dgDef = new DirectoryGroupDefinition(session);
   ```

2. DirectoryGroup の属性値を設定します。

   ```
   dgDef.setAttributeByUpperCaseName("NAME",
            AttributeValue.newAttributeValue(
              "Managers"));
   ```

3. DirectoryGroupDefinition を LibrarySession に渡して、新規 DirectoryGroup インスタンスを作成します。

   ```
   DirectoryGroup dg = (DirectoryGroup) session.createPublicObject(dgDef);
   ```

4. DirectoryGroup に属する DirectoryUser を取得します。

   ```
   DirectoryUser du;
   LibraryObject[] dus;
   ```

```
Selector s = new Selector(session);
s.setSearchClassname("DIRECTORYUSER");
s.setSearchSelection("NAME IN ('msmith', 'tturner', 'rjones')");
dus = s.getItems();

int icount = (dus == null) ? 0 : dus.length;
for (i = 0; i < icount; i++)
{
    du = (DirectoryUser) du[i];
```

5. DirectoryUser を DirectoryGroup に追加します。

```
    dg.addMember(du);
}
```

例 15-17 Java を使用した DirectoryGroup インスタンスの変更

```
/*
 * The following example moves DirectoryUsers
 * from one group to another.
 */
```

1. アプリケーションが DirectoryGroup を取得しているとします。

```
DirectoryGroup dg1 = ...
DirectoryGroup dg2 = ...
```

2. 最初のグループから DirectoryUser をフェッチします。

```
DirectoryObject[] dus = dg1.getDirectMembers();
```

3. DirectoryUser を 2 番目のグループに追加します。addMembers() メソッドは、追加するメンバーが 1 人でもそのグループに存在すると失敗します。

```
dg2.addMembers(dus);
```

4. 最初のグループから DirectoryUser を削除します。

```
dg1.removeMembers(dus);
```

例 15-18 Java を使用した DirectoryGroup インスタンスの削除

1. アプリケーションが DirectoryGroup を取得しているとします。

```
DirectoryGroup dg = ...
```

2. DirectoryGroup を削除します。

```
dg.free();
```

注意： DirectoryGroup を削除しても、そのグループから DirectoryUser が自動的に削除されることはありません。DirectoryUser を削除する方法は、「DirectoryUser および PrimaryUserProfile の作成、変更および削除」を参照してください。

カスタム・グループ・クラスの定義

カスタム・グループ・クラスを定義する手順は、他のカスタム・クラスを定義する場合と同じです。カスタム・グループ・クラスの作成は、Java API、XML ファイルまたは Oracle 9*i*FS Manager を使用して実行できます。

XML を使用して ExtendedUserProfile クラスを作成および変更する方法は、例 15-19 を参照してください。クラスの削除には、XML は使用できません。

Java を使用して ExtendedUserProfile クラスを定義、変更および削除する方法は、例 15-20 および例 15-21 を参照してください。

注意： Oracle 9*i*FS Manager でカスタム・グループ・クラスを定義する手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

グループ・クラスの作成

カスタム・グループ・クラスを作成するには、DirectoryGroup クラスをサブクラス化し、そのタイプのグループのカスタム属性および動作を追加します。たとえば、DirectoryGroup を拡張して拡張属性 VicePresident および MissionStatement を含めるクラス Organization を作成できます。

カスタム・グループ・クラスを定義する手順は、次のとおりです。

1. 管理権限を持っているかどうかを確認します。XML 構成ファイルを使用する場合は、そのファイルを管理者としてインポートします。Java API を使用する場合は、Oracle 9*i*FS でのセッション用に管理モードを有効化します。
2. グループ・クラスを表す ClassObject のインスタンスを作成します。
3. 各拡張属性を表す Attribute のインスタンスを作成します。
4. オプションで、グループ・クラス用のカスタム Java クラスを作成し、カスタム・メソッドを定義します。

注意： カスタム Java クラスの作成方法は、第 17 章「コンテンツ・タイプの動作のカスタマイズ」を参照してください。

例 15-19 XML を使用したグループ・クラスの作成

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <NAME>Organization</NAME>
    <SUPERCLASS RefType = "Name">DirectoryGroup</SUPERCLASS>
    <DESCRIPTION>Custom group class to manage organizations.</DESCRIPTION>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>VicePresident</NAME>
            <DATATYPE>DirectoryObject</DATATYPE>
        </ATTRIBUTE>
        <ATTRIBUTE>
            <NAME>MissionStatement</NAME>
            <DATATYPE>String</DATATYPE>
        </ATTRIBUTE>
    </ATTRIBUTES>
</CLASSOBJECT>
```

注意： グループ・クラスの作成に使用する XML ファイルの各要素の詳細は、第 5 章「コンテンツ・タイプと属性の拡張」を参照してください。

例 15-20 Java を使用したグループ・クラスの作成

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. グループ・クラスを表す ClassObject の値を保持する ClassObjectDefinition を作成します。

```
ClassObjectDefinition codef = new ClassObjectDefinition(session);
codef.setAttributeByUpperCaseName(ClassObject.NAME_ATTRIBUTE,
     AttributeValue.newAttributeValue("Organization"));
```

3. 新規カテゴリ・タイプで Category コンテンツ・タイプを拡張するように指定します。

```
ClassObject catco = session.getClassObjectByName("DIRECTORYGROUP");
codef.setSuperclass(catco);
```

4. カテゴリ・タイプに関連する属性ごとに AttributeDefinition を作成します。AttributeDefinition を ClassObjectDefinition に追加します。

```
AttributeDefinition attdef1 = new AttributeDefinition(session);
attdef1.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("VicePresident"));
attdef1.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_
DIRECTORYOBJECT));

codef.addAttributeDefinition(attdef1);

AttributeDefinition attdef2 = new AttributeDefinition(session);
attdef2.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("MissionStatement"));
attdef2.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_STRING));

codef.addAttributeDefinition(attdef2);
```

5. 新規 Group クラスの ClassObject および Attribute を作成します。

```
ClassObject groupCo = (ClassObject) session.createSchemaObject(codef);
```

グループ・クラスの変更

作成したグループ・クラスの属性を追加、削除および更新できます。

注意： グループ・クラスの属性の追加や削除には、XML を使用できません。この操作には、Oracle 9*i*FS Manager または Java API を使用する必要があります。

例 15-21 Java を使用したグループ・クラスの属性の追加、削除および更新

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. グループ・クラスを表す ClassObject を取得します。

```
ClassObject co = session.getClassObjectByName("ORGANIZATION");
```

3. Attribute を ClassObject に追加します。

```
AttributeDefinition attdef1 = new AttributeDefinition(session);
attdef1.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
```

```
        AttributeValue.newAttributeValue("OrganizationWorkArea"));
attdef1.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
        AttributeValue.newAttributeValue(
          Attribute.ATTRIBUTEDATATYPE_PUBLICOBJECT));
co.addAttribute(attdef1);
```

4. Attribute を ClassObject から削除します。

```
Attribute att2 = co.getAttributeFromLabel("MISSIONSTATEMENT");
co.removeAttribute(att2);
```

5. ClassObject の Attribute を更新します。

```
Attribute att3 = co.getAttributeFromLabel("ORGANIZATIONWORKAREA");
Collection cdColl = session.getClassDomainCollection();
ClassDomain cd = (ClassDomain) cdColl.getItems("Folder...");
att3.setClassDomain(cd);
```

例 15-22 Java を使用したグループ・クラスの削除

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. Group クラスを表す ClassObject を取得します。

```
ClassObject groupCo = session.getClassObjectByName("ORGANIZATION");
```

3. `free()` メソッドをコールして ClassObject を削除します。

```
groupCo.free();
```

カスタム・グループの作成、変更および削除

カスタム・グループ・クラスを定義した後に、そのクラスのグループを作成してメンバーシップを管理できます。カスタム・グループの作成と管理には、DirectoryGroup の作成時と同じ多数の XML タグと Java メソッドを使用します。ただし、カスタム・グループ・インスタンスを操作する場合は、グループのクラスを識別する必要があります。

例 15-23 および例 15-24 に、XML を使用してカスタム・グループ・クラスのインスタンスを作成および変更する方法を示します。インスタンスの削除には、XML は使用できません。

Java を使用してカスタム・グループ・インスタンスを作成、変更および削除する方法は、それぞれ例 15-25、例 15-26 および例 15-27 を参照してください。

例 15-23 XML を使用したカスタム・グループ・インスタンスの作成

```
<?xml version = '1.0' standalone = 'yes'?>
 <ORGANIZATION>
```

```
  <NAME>Development</NAME>
  <VICEPRESIDENT RefType = "Name">ztolls</VICEPRESIDENT>
  <MISSIONSTATEMENT>To develop the company products.</MISSIONSTATEMENT>
  <MEMBERS>
    <REF RefType ="Name">czacks</REF>
    <REF RefType = "Name">mwallen</REF>
    <REF RefType = "Name">ztolls</REF>
  </MEMBERS>
 </ORGANIZATION>
```

例 15-24 XML を使用したカスタム・グループ・インスタンスの変更

```
<?xml version = '1.0' standalone = 'yes'?>
 <ORGANIZATION>
  <UPDATE RefType = "Name">Development</UPDATE>
  <VICEPRESIDENT RefType = "Name">lrichards</VICEPRESIDENT>
  <MEMBERS>
    <REF RefType = "Name">lrichards</REF>
  </MEMBERS>
 </ORGANIZATION>
```

注意： カスタム・グループから DirectoryUser を削除する場合は、XML を使用できません。この操作には、Oracle 9*i*FS Manager または Java API を使用する必要があります。

例 15-25 Java を使用したカスタム・グループ・インスタンスの作成

1. 新規の DirectoryGroupDefinition を作成します。

```
DirectoryGroupDefinition dgDef = new DirectoryGroupDefinition(session);
```

2. グループのクラスを指定します。

```
ClassObject co = session.getClassObjectByName("ORGANIZATION");
dgDef.setClassObject(co);
```

3. DirectoryGroup の属性値を設定します。

```
dgDef.setAttributeByUpperCaseName(DirectoryGroup.NAME_ATTRIBUTE,
         AttributeValue.newAttributeValue(
           "Managers"));

Collection dus = session.getDirectoryUserCollection();
DirectoryUser du = (DirectoryUser) dus.getItems("tturner")
dgDef.setAttributeByUpperCaseName("VICEPRESIDENT",
         AttributeValue.newAttributeValue(du));
```

```
dgDef.setAttributeByUpperCaseName("MISSIONSTATEMENT",
         AttributeValue.newAttributeValue(
           "To develop the company products."));
```

4. DirectoryGroupDefinition を LibrarySession に渡して、新規 DirectoryGroup インスタンスを作成します。

```
DirectoryGroup dg = (DirectoryGroup) session.createPublicObject(dgDef);
```

5. DirectoryGroup に属する DirectoryUser を取得します。

```
Collection groupDus = session.getDirectoryUserCollection();

int i;
int icount = groupDus.getItemCount();
for (i = 0; i < icount; i++)
{
  du = (DirectoryUser) groupDus.getItems(i);
```

6. DirectoryUser を DirectoryGroup に追加します。

```
    dg.addMember(du);
}
```

例 15-26 Java を使用したカスタム・グループ・インスタンスの変更

```
/*
 * The following example adds a DirectoryUser to a custom group.
 */
```

1. カスタム・グループ・インスタンスを検索します。グループのクラスを指定します。

```
Selector s = new Selector(session);
s.setSearchClassname("ORGANIZATION");
s.setSearchSelection("NAME = 'Development'");
```

2. グループ・インスタンスをフェッチします。カスタム・グループは DirectoryGroup を拡張するため、このインスタンスは DirectoryGroup として宣言できます。

```
DirectoryGroup dg = (DirectoryGroup) s.getItems(0);
```

3. グループに DirectoryUser を追加します。

```
DirectoryUser du = session.getDirectoryUser();
dg.addMembers(du);
```

例 15-27 Java を使用したカスタム・グループ・インスタンスの削除

```
/*
 * The following example deletes all instances of a custom group.
 */
```

1. すべてのカスタム・グループ・インスタンスを検索します。グループのクラスを指定します。

```
Selector s = new Selector(session);
s.setSearchClassname("ORGANIZATION");
LibraryObject[] dgs = s.getItems();
```

2. グループ・インスタンスをフェッチします。カスタム・グループは DirectoryGroup を拡張するため、このインスタンスは DirectoryGroup として宣言できます。

```
DirectoryGroup dg;

int i;
int icount = (dgs == null) ? 0 : dgs.length;
for (i = 0; i < icount; i++)
{
  dg = (DirectoryGroup) dgs[i];
```

3. DirectoryGroup を削除します。

```
  dg.free();
}
```

注意： カスタム・グループ・インスタンスを削除しても、そのグループから DirectoryUser が自動的に削除されることはありません。DirectoryUser を削除する方法は、「DirectoryUser および PrimaryUserProfile の作成、変更および削除」を参照してください。

DirectoryUser の認証

ユーザーがリポジトリ内の情報へのアクセスを試みると、Oracle 9*i*FS はまずユーザーの資格証明を認証します。ユーザーは次の手順で認証を受けます。

1. ユーザーが、リポジトリへのログインに使用しているアプリケーション経由で資格証明を入力します。
2. アプリケーションが、資格証明を保持するように `oracle.ifs.common` パッケージ内のクラスをインスタンス化します。アプリケーションは、ユーザー認証に使用する資格証明のタイプに応じて様々なクラスから選択できます。資格証明の管理に最も一般的に使用されるクラスは、CleartextCredential です。

3. アプリケーションが資格証明を LibraryService に渡して、リポジトリとのセッションの確立を試みます。
4. LibraryService は、指定されたユーザーの DistinguishedName に対応する DirectoryUser インスタンスを取得します。DirectoryUser インスタンスが存在しない場合、LibraryService は接続を拒否します。
5. DirectoryUser インスタンスが存在する場合、LibraryService はユーザーの資格証明を DirectoryUser CredentialManager 属性で指定されている Credential Manager に渡します。

注意： このリリースの Oracle 9*i*FS には、ユーザー全員の認証に使用されるビルトインの資格証明マネージャが用意されています。Oracle 9*i*FS の今後のリリースでは、外部資格証明マネージャのサポートが予定されています。

---

6. Credential Manager が資格証明を検証します。資格証明が有効な場合、LibraryService はアプリケーションを Oracle 9*i*FS に接続する LibrarySession を作成します。LibrarySession は、リポジトリ内でユーザーが実行するすべての操作（リポジトリ・オブジェクトの作成、更新および削除など）のファクトリとして機能します。

注意： 例 15-28 に、アプリケーションが CleartextCredential を使用してユーザーを認証する方法を示します。この項で説明した以外の LibraryService および LibrarySession の操作の詳細は、第 4 章「Oracle Internet File System ドキュメントの作成」を参照してください。

例 15-28 Java を使用したユーザーの認証

```
//Connect to Oracle 9iFS.
LibraryService service =
       LibraryService.startService("IfsDefault", "ifssys");
ConnectOptions cop = new ConnectOptions();
CleartextCredential cred =
       new CleartextCredential("system", "manager9ifs");
LibrarySession session = service.connect(cred, cop);
```

PublicObject へのアクセスの管理

この項では、Oracle 9*i*FS で AccessControlList を使用して PublicObject へのアクセスを管理する方法について説明します。この項の内容は、次のとおりです。

- AccessControlList の機能
- AccessControlList の構造
- AccessControlList の作成、変更および削除
- PublicObject への ACL の適用
- AccessControlList へのアクセスの管理

AccessControlList の機能

各 PublicObject へのアクセスは、AccessControlList により管理されます。AccessControlList は、PublicObject へのアクセス権が付与または取り消されているユーザーおよびグループのリストです。ユーザーまたはグループのアクセス権が付与または取り消されるたびに、AccessControlList 内で新規エントリが作成されます。エントリには、Grantee（ユーザーまたはグループ）、AccessLevel（権限のセット）および権限が付与されるか取り消されるかが記録されます。

アクセス権の解決

Grantee がグループの場合、エントリはそのグループまたはサブグループに属するユーザー全員に適用されます。1 人のユーザーに複数の AccessControlEntry が適用される場合、Oracle 9*i*FS はエントリを AccessControlList 内で作成された順に集計し、ユーザーの有効アクセス権を解決します。たとえば、ユーザーに最初のエントリで Discover および GetContent が付与され、2 番目のエントリで SetAttribute および SetContent が付与され、3 番目のエントリで SetContent が取り消されている場合、Oracle 9*i*FS は有効アクセス権を Discover、GetContent および SetAttribute に解決します。

AccessControlList の共有

AccessControlList はリポジトリに独立して存在し、1 つ以上の PublicObject に割り当てることができます。AccessControlList を PublicObject に割り当てるには、PublicObject の ACL 属性で AccessControlList を参照します。

各 AccessControlList は属性 Shared を持ちます。この属性は、AccessControlList が単一の PublicObject に対してプライベートであるか、複数の PublicObject 間で共有されるかを示します。AccessControlList を共有にすると、同様のセキュリティ制約を持つ多数の PublicObject へのアクセスを簡単に制御できます。たとえば、AccessControlList を作成して、特定のプロジェクトに関連するすべての PublicObject へのアクセスを制御するとします。この場合、プロジェクト・ドキュメントを読み込めるユーザー、プロジェクトのコンテンツ作成者として指定されているユーザー、プロジェクト作業領域内のフォルダを管理するユーザーを定義できます。その後、すべてのプロジェクト・ドキュメントとフォルダ間で

AccessControlList を共有にします。これにより、プロジェクトのセキュリティをドキュメントおよびフォルダごとに別個にではなく、単一の AccessControlList で一度にメンテナンスできます。

システム単位のセキュリティ・レベルの定義

管理者は、特殊なタイプの AccessControlList である SystemAccessControlList を作成できます。SystemAccessControlList では、いずれの Oracle 9*i*FS ユーザーでもリポジトリ内の情報に適用できるアクセス権を定義します。Oracle 9*i*FS は、Private、Protected、Published および Public という 4 つの SystemAccessControlList とともにインストールされます。

注意： SystemAccessControlList を作成、変更および削除する手順の詳細は、この章では説明しません。

カスタム権限の定義

Oracle 9*i*FS は、Discover、GetContent、SetAttribute、SetContent、Delete など、AccessControlList で付与または取り消すことができる標準権限セットで事前に構成済みの状態でインストールされます。また、カスタム・アクセス権 ExtendedPermission を定義して、ユーザーがカスタム・アプリケーション・コードにより実装されるタスクを実行できるかどうかの制御に使用することもできます。たとえば、法的契約書の管理アプリケーションをサポートするために、ドキュメントにデジタル署名を添付できるかどうかについて、カスタム権限を定義できます。

注意： ExtendedPermission を作成、変更および削除する手順の詳細は、この章では説明しません。

より広範囲なアクセス・レベルの定義

Oracle 9*i*FS でデフォルトで定義される権限を使用して、情報についてファイングレイン・アクセス・コントロールを行うことができます。また、Oracle 9*i*FS では、アクセス権を PermissionBundle にグループ化することで、より広範囲の権限レベルを定義することもできます。たとえば、ユーザーにフォルダの変更権限を簡単に付与できるように、Discover、Add Item および Remove Item の権限を ModifyFolderPermissionBundle にグループ化できます。

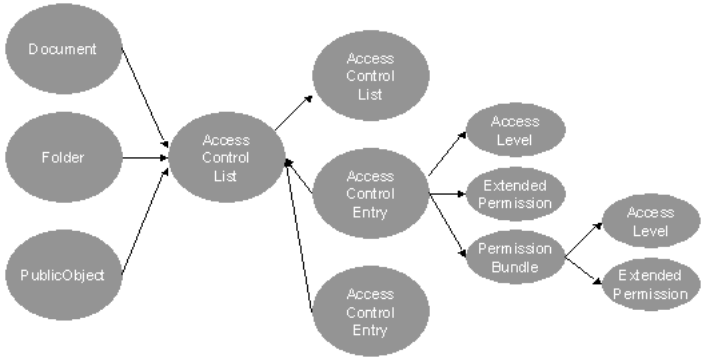
注意： PermissionBundle を作成、変更および削除する手順の詳細は、この章では説明しません。

AccessControlList の構造

各 AccessControlList は、AccessControlList クラスのインスタンスにより表されます。ユーザーやグループのアクセス権を付与または取り消す各エントリは、AccessControlEntry クラスのインスタンスで表されます。各 AccessControlEntry は属性 ACL を持ち、属している AccessControlList を参照します。また、AccessControlEntry は、どのユーザーまたはグループにどの権限が付与されたか、取り消されたかを記録する属性も持っています。AccessControlEntry は、1 つの AccessLevel と 1 つ以上の ExtendedPermission および PermissionBundle を参照できます。AccessLevel は、Oracle 9*i*FS で定義されている標準権限セットを表します。ExtendedPermission は、ユーザーが定義したカスタム権限を表します。PermissionBundle は、権限グループで構成される高度なアクセス・レベルを表します。

図 15-2 に AccessControlList の構造を示します。

図 15-2 AccessControlList の構造



AccessControlList

AccessControlList クラスは、リスト内のユーザーとグループについて権限を付与および取り消すための、プライマリ・インタフェースとして機能します。表 15-7 に、AccessControlList で最も使用頻度の高い属性とメソッドを示します。

表 15-7 AccessControlList の属性およびメソッド

Shared isShared()	AccessControlList を複数の PublicObject 間で共有できるかどうかを決定します。

表 15-7 AccessControlList の属性およびメソッド（続く）

checkEffectiveAccess() checkGrantedAccess() getEffectiveAccessLevel() getGrantedAccessLevel()	AccessControlList でユーザーまたはグループに付与されているアクセス権を判断します。
grantAccess() removeAccessControlEntry() revokeAccess() revokeAllAccess() updateAccessControlEntry()	ユーザーが AccessControlList でアクセス権を付与または取り消すことができるように、便利なインタフェースを提供します。
ACL	この AccessControlList での権限の付与と取消し、またはこの AccessControlList の削除ができるユーザーとグループを制御する、別の AccessControlList を参照します。

AccessControlEntry

AccessControlEntry クラスは、AccessControlList 内の各エントリ、つまり、権限受領者、権限およびその権限が付与されるのか取り消されるのかを定義するための属性とメソッドを持ちます。AccessControlList と AccessControlEntry の関係は、AccessControlEntry の 2 つの属性 ACL および SortSequence によりメンテナンスされます。ACL 属性は、AccessControlEntry が属する AccessControlList を参照します。SortSequence 属性は、AccessControlList 内の各 AccessControlEntry の相対位置を決定します。AccessControlEntry の SortSequence は、付与または取り消す権限の優先順位の決定に使用されます。

表 15-8 に、AccessControlEntry で最も使用頻度の高い属性とメソッドを示します。

表 15-8 AccessControlEntry の属性およびメソッド

ACL getACL()	この AccessControlEntry が属している AccessControlList を判断します。
Grantee getGrantee()	権限が付与または取り消されるユーザーまたはグループを判断します。
AccessLevel getDistinctAccessLevel() getMergedAccessLevel()	付与または取り消される権限を判断します。
Granted	権限が付与されるか取り消されるかを示します。

表 15-8 AccessControlEntry の属性およびメソッド（続く）

SortSequence getSortSequence()	この AccessControlEntry の、AccessControlList でのソート順序を決定します。
PermissionBundles getPermissionBundles()	AccessControlEntry に設定されている PermissionBundle を判断します。
ExtendedPermissions getExtendedPermissions()	この AccessControlEntry で付与または取り消された拡張権限を判断します。

AccessLevel

Oracle 9*i*FS は AccessLevel クラスを使用して、AccessControlEntry 内で付与または取り消されている標準権限セットを表します。AccessLevel クラスは、Oracle 9*i*FS により定義された各標準権限を表す定数を持ちます。表 15-9 に、PublicObject に関連する標準権限を示します。

表 15-9 PublicObject の権限

定数	使用方法
AccessLevel_Discover	オブジェクトを検出し、その属性を読み込む権限（ただし、Document の場合、その内容とはかぎりません）。
AccessLevel_GetContent	オブジェクトの内容を読み込む権限（Document にのみ適用されます）。
AccessLevel_SetAttribute	すべてのオブジェクトの属性を直接更新する権限。
AccessLevel_SetContent	オブジェクトの内容を更新する権限（Document にのみ適用されます）。
AccessLevel_Delete	オブジェクトを削除する権限。PublicObject の ExpirationDate 属性を設定または消去する場合にも必要です。
AccessLevel_Lock	PublicObject をロックまたはロック解除する権限。
AccessLevel_Grant	PublicObject の ACL、Owner または AdministrationGroup を他のユーザーに変更する権限。
AccessLevel_AddMember	DirectoryGroup に DirectoryObject メンバーを追加する権限。権限はターゲット DirectoryGroup に適用されます。
AccessLevel_ RemoveMember	DirectoryGroup から DirectoryObject メンバーを削除する権限。権限はターゲット DirectoryGroup に適用されます。
AccessLevel_AddItem	Folder に PublicObject アイテムを追加する権限。権限はターゲット Folder に適用されます。

表 15-9 PublicObject の権限（続く）

定数	使用方法
AccessLevel_RemoveItem	Folder から PublicObject アイテムを削除する権限。権限はターゲット Folder に適用されます。
AccessLevel_AddRelationship	ある PublicObject（RightObject）を別の PublicObject（LeftObject）に関連付ける権限。権限は LeftObject に適用されます。
AccessLevel_RemoveRelationship	ある PublicObject（RightObject）と別の PublicObject（LeftObject）との関連付けを解除する権限。権限は LeftObject に適用されます。
AccessLevel_AddVersonSeries	Family に VersionSeries を追加する権限。権限は Family に適用されます。
AccessLevel_RemoveVersionSeries	Family から VersionSeries を削除する権限。権限は Family に適用されます。
AccessLevel_AddVerson	VersionSeries に新規 Version を追加する権限。権限は VersionSeries に適用されます。
AccessLevel_RemoveVersion	VersionSeries から Version を削除する権限。権限は VersionSeries に適用されます。
AccessLevel_SetDefaultVersion	Family または VersionSeries のデフォルト・バージョンまたは VersionSeries を変更する権限。
AccessLevel_SetPolicy	PublicObject に対応付けられている PolicyPropertyBundle を変更する権限。権限は PolicyPropertyBundle に適用されます。

- Relationship の詳細は、第 6 章「任意のメタデータと動作の適用」を参照してください。
- Versioning の詳細は、第 14 章「バージョニングの実装」を参照してください。

表 15-10 に、AccessLevel で最も使用頻度の高いメソッドを示します。

表 15-10 AccessLevel の属性およびメソッド

add() equal() subtract()	この AccessLevel について、別の AccessLevel に存在する権限を追加または削除します。

表 15-10 AccessLevel の属性およびメソッド（続く）

clearAllPermissions() clearAllStandardPermissions() disableAllPermissions() disableAllStandardPermissions() enableAllPermissions() enableAllStandardPermissions()	AccessLevel での権限を消去、有効または無効にします。
getAllDefinedStandardPermissions() getAllDefinedStandardPermissionNames() enableEnabledStandardPermissions() getEnabledStandardPermissionNames() isStandardPermissionEnabled() isSufficientEnabled()	AccessLevel で有効化され、定義されている権限を判断します。

ExtendedPermissions

Oracle 9*i*FS では、ExtendedPermissions クラスのインスタンスとしてカスタム権限を定義できます。たとえば、法的契約書の管理アプリケーションをサポートするために、ドキュメントにデジタル署名を添付できるかどうかについて、カスタム権限を定義できます。このカスタム権限を定義するには、一意の名前（Sign など）でカスタム権限を識別する、ExtendedPermission クラスのインスタンスを作成します。その後は、ExtendedPermission を保持するように AccessLevel を構成し、AccessControlEntry 内で ExtendedPermission を付与および取り消すことができます。

Java API の AccessLevel クラスには、ExtendedPermission を操作するためのメソッド・セットが用意されています。表 15-11 に、ExtendedPermission の操作に使用する AccessLevel クラスのメソッドを示します。

表 15-11 ExtendedPermission に関連する AccessLevel のメソッド

enableExtendedPermission()	AccessLevel 内で ExtendedPermission を設定するために使用されます。
disableExtendedPermission()	AccessLevel から ExtendedPermission を削除するために使用されます。
getExtendedPermissions()	AccessLevel に保持されているすべての ExtendedPermission を取得するために使用されます。
isExtendedPermissionEnabled()	ExtendedPermission が AccessLevel に保持されているかどうかをチェックするために使用されます。

ExtendedPermission が AccessControlList に適用された後は、ExtendedPermission を使用してユーザーによる PublicObject の操作方法を制御するように、カスタム・アプリケーションを構築できます。アプリケーションでは、ユーザーが PublicObject に対する ExtendedPermission を付与されているかどうかをチェックし、検出結果に基づいて操作の実行を許可または禁止するオーバーライドを実装できます。

表 15-12 に、ExtendedPermission の操作に使用される S_PublicObject のメソッドを示します。

表 15-12 ExtendedPermission に関連する S_PublicObject のメソッド

checkAccess()	S_PublicObject のユーザーに ExtendedPermission が付与されているかどうかを判断します。
extendedPostInsert() extendedPreInsert() extendedPostUpdate() extendedPreUpdate() extendedPostDelete() extendedPreDelete()	ユーザーが実行できる操作を ExtendedPermission に基づいて制御する、カスタム・ビジネス・ルールを実装します。

PermissionBundle

AccessLevel には広範囲な権限リストが用意されており、情報へのアクセス方法を厳密に制御できます。ただし、ユーザーがより広範囲のアクセス権を PublicObject に簡単に適用できるように、セキュリティ・レベルを定義できます。

Oracle 9*i*FS では、PermissionBundle クラスのインスタンスとして、より広範囲のセキュリティ・レベルを定義できます。これは、Oracle 9*i*FS に永続的に格納されます。たとえば、ユーザーにフォルダの変更権限を簡単に付与できるように、Discover、Add Item および Remove Item の権限を ModifyFolderPermissionBundle にグループ化できます。

表 15-13 に、PermissionBundle で最も使用頻度の高い属性とメソッドを示します。

表 15-13 PermissionBundle の属性およびメソッド

AccessLevel getAccessLevel() setAccessLevel()	PermissionBundle での AccessLevel を判断します。

AccessControlList の作成、変更および削除

Oracle 9*i*FS には、AccessControlList を作成、変更および削除できるように、次の 3 つの方法が用意されています。

- **XML:** Oracle 9*i*FS へのインポート時に AccessControlList を自動的に定義する XML ファイルを作成します。
- **Java:** Oracle 9*i*FS Java API 経由で AccessControlList をプログラムで定義します。
- **Oracle 9*i*FS クライアント:** **Oracle 9*i*FS** のウィンドウおよび Web クライアントで AccessControlList を定義します。

この項では、XML および Java を使用して AccessControlList を定義する方法について説明します。Oracle 9*i*FS クライアントで AccessControlList を定義する手順は、各クライアントの Oracle 9*i*FS オンライン・ヘルプを参照してください。

XML を使用した AccessControlList の作成および変更

Oracle 9*i*FS では、XML ファイルを使用して AccessControlList を定義できます。XML ファイルが Oracle 9*i*FS にインポートされると、IfsSimpleXmlParser は自動的にそれを使用して AccessControlList と AccessControlEntry インスタンスを作成または変更します。Oracle 9*i*FS では、XML を使用した AccessControlList の削除はサポートされていません。

例 15-29 および例 15-30 に、XML を使用して AccessControlList を作成する方法と変更する方法を示します。

例 15-29 XML を使用した AccessControlList の作成

```
<?xml version = '1.0' standalone = 'yes'?>
<ACCESSCONTROLLIST>
  <NAME>ContentManagementProjectACL</NAME>
  <DESCRIPTION>
     ACL used to manage access to all information
     pertaining to the Content Management project.
  </DESCRIPTION>
  <OWNER RefType = "Name">guest</OWNER>
  <ACEs>
    <ACCESSCONTROLENTRY>
      <GRANTEE ClassName="DirectoryUser" RefType="Name">
         guest
      </GRANTEE>
      <ACCESSLEVEL>
          <DISCOVER> true </DISCOVER>
          <GETCONTENT> true </GETCONTENT>
          <SETATTRIBUTE> true </SETATTRIBUTE>
          <SETCONTENT> true </SETCONTENT>
          <DELETE> true </DELETE>
      </ACCESSLEVEL>
```

```
      <GRANTED>true</GRANTED>
    </ACCESSCONTROLENTRY>
  </ACEs>
</ACCESSCONTROLLIST>
```

例 15-30 XML を使用した AccessControlList の変更

```
<?xml version = '1.0' standalone = 'yes'?>
<ACCESSCONTROLLIST>
  <UPDATE RefType = "Name">ContentManagementProjectACL</UPDATE>
    <OWNER RefType = 'Name'>system</OWNER>
  <ACEs>
    <ACCESSCONTROLENTRY>
      <GRANTEE ClassName='DirectoryUser' RefType="Name"> guest </GRANTEE>
      <ACCESSLEVEL>
          <SETATTRIBUTE> true </SETATTRIBUTE>
          <SETCONTENT> true </SETCONTENT>
          <DELETE> true </DELETE>
      </ACCESSLEVEL>
      <GRANTED>false</GRANTED>
    </ACCESSCONTROLENTRY>
  </ACEs>
</ACCESSCONTROLLIST>
```

XML 構成ファイルの各要素を使用して、AccessControlList および AccessControlEntry インスタンスの属性の値を指定します。

注意： XML 構成ファイルの構文規則の詳細は、第 10 章「XML および Oracle Internet File System」を参照してください。

表 15-14 に、AccessControlList 構成ファイルの各 XML 要素を示します。この表は、Java クラスとその要素に対応する Oracle 9*i*FS Java API の属性、各要素の使用方法および要素の有効な XML 属性を示しています。

表 15-14 AccessControlList の XML 構成ファイルの要素

要素	クラス情報	属性	使用方法
<ACCESSCONTROLLIST>	AccessControlList PublicObject のサブクラス		AccessControlList 構成ファイルのルート要素。AccessControlList クラスの名前に対応する必要があります。
<NAME>	AccessControlList クラスの Name 属性		<NAME> 要素は、AccessControlList の名前を指定するために使用されます。
<ACEs>	AccessControlList クラスの属性との直接の対応関係はなし		AccessControlEntry インスタンスの宣言は、<ACEs> 要素で囲む必要があります。
<ACCESSCONTROLENTRY>	AccessControlEntry SystemObject のサブクラス		各 AccessControlEntry は <ACCESSCONTROLENTRY> 要素で宣言されます。
<GRANTEE>	AccessControlEntry クラスの Grantee 属性	RefType ClassName Existing	<GRANTEE> 要素は、AccessControlEntry 内で権限が付与または取り消されるユーザーまたはグループを指定するために使用されます。
<ACCESSLEVEL>	AccessControlEntry クラスの AccessLevel 属性		<ACCESSLEVEL> 要素は、付与または取消し対象となる権限を保持するために使用されます。

表 15-14 AccessControlList の XML 構成ファイルの要素（続く）

要素	クラス情報	属性	使用方法
<DISCOVER> <GETCONTENT> <SETATTRIBUTE> <SETCONTENT> <DELETE> <LOCK> <GRANT> <ADDMEMBER> <REMOVEMEMBER> <ADDITEM> <REMOVEITEM> <ADDRELATIONSHIP> <REMOVERELATIONSHIP> <ADDVERSIONSERIES> <REMOVEVERSIONSERIES> <ADDVERSION> <REMOVEVERSION> <SETDEFAULTVERSION> <SETPOLICY> <CREATE> <SELECTORACCESS>	対応する権限を表す AccessLevel クラスの定数		各要素は、AccessLevel に特定の権限を含めるために使用されます。各要素の値は、AccessLevel に権限を含める場合は TRUE に設定されます。
<EXTENDEDPERMISSIONS>	AccessControlEntry クラスの ExtendedPermissions 属性		<EXTENDEDPERMISSIONS> 要素は、AccessControlEntry で付与または取消し対象となる ExtendedPermission を保持するために使用されます。
<EXTENDEDPERMISSION>	SystemObject を拡張する ExtendedPermission クラス	RefType ClassName	各 ExtendedPermission は、<EXTENDEDPERMISSION> 要素で表されます。この要素の値は、ExtendedPermission の Name を参照します。

表 15-14 AccessControlList の XML 構成ファイルの要素（続く）

要素	クラス情報	属性	使用方法
<GRANTED>	AccessControlEntry クラスの Granted 属性		<GRANTED> 要素は、権限を付与するか取り消すかを指定するために使用されます。この要素の値は、権限を付与する場合は TRUE、取り消す場合は FALSE に設定されます。
<PERMISSIONBUNDLES>	AccessControlEntry クラスの PermissionBundles 属性		<PERMISSIONBUNDLES> 要素は、AccessControlEntry で付与または取消し対象となる PermissionBundle を保持するために使用されます。
<PERMISSIONBUNDLE>	SystemObject クラスを拡張する PermissionBundle クラス	RefType ClassName	各 PermissionBundle は、要素の RefType 属性を使用して PermissionBundle オブジェクトを参照する <PERMISSIONBUNDLE> 要素で表されます。

Java を使用した AccessControlList の作成、変更および削除

Oracle 9*i*FS Java API には、AccessControlList の各コンポーネントを操作するための、より確実なインタフェースが用意されています。例 15-31、例 15-32 および例 15-33 に、Java API を使用して AccessControlList を作成、変更および削除する方法を示します。

例 15-31 Java を使用した AccessControlList の作成

1. アプリケーションは、AccessControlList で権限が付与される DirectoryUser をすでに取得しているとします。

```
DirectoryGroup du = ...
```

2. 新規 AccessControlListDefinition を作成します。

```
AccessControlListDefinition aclDef = new AccessControlListDefinition(session);
```

3. AccessControlList の属性値を設定します。

```
aclDef.setAttributeByUpperCaseName("NAME",
         AttributeValue.newAttributeValue(
```

```
            "ContentManagementProjectACL"));

aclDef.setShared(TRUE);
```

4. AccessControlListDefinition を LibrarySession に渡して、新規 AccessControlList を作成します。

```
AccessControlList acl = (AccessControlList) session.createPublicObject(aclDef);
```

5. DirectoryUser に権限を付与する AccessControlEntryDefinition を作成します。AccessControlList で AccessControlEntryDefinition を設定します。

```
AccessControlEntryDefinition aceDef1 =
             new AccessControlEntryDefinition(session);
aceDef1.setGrantee(du);

String[] levels = {"Discover",
                   "GetContent",
                   "SetAttribute",
                   "SetContent",
                   "Delete"};
AccessLevel al1 = new AccessLevel(levels);
aceDef1.setDistinctAccessLevel(al1);

acl.grantAccess(aceDef1);
```

6. DirectoryUser から Delete 権限を取り消す AccessControlEntryDefinition を作成します。AccessControlList で AccessControlEntryDefinition を設定します。

```
AccessControlEntryDefinition aceDef2 =
             new AccessControlEntryDefinition(session);
aceDef2.setGrantee(du);

AccessLevel al2 = new AccessLevel(AccessLevel.ACCESSLEVEL_DELETE);
aceDef2.setDistinctAccessLevel(al2);

acl.revokeAccess(aceDef2);
```

例 15-32 Java を使用した AccessControlList の変更

1. アプリケーションは、AccessControlList と、そこで新規の権限が付与される DirectoryUser をすでに取得しているとします。

```
AccessControlList acl = ...
DirectoryUser du = ...
```

2. ユーザーにすべての権限を付与する新規 AccessControlEntryDefinition を作成します。

```
AccessControlEntryDefinition aceDef =
```

```
    new AccessControlEntryDefinition(session);
aceDef.setGrantee(du);
aceDef.setGranted(true);

AccessLevel al = new AccessLevel();
al.enableAllStandardPermissions();
aceDef.setDistinctAccessLevel(al);
```

3. 新規 AccessControlEntry を AccessControlList の先頭に挿入する必要があるかどうかを指定します。先頭に挿入すると、Oracle 9*i*FS で解決されるときには、そのグループのメンバーから権限を取り消す AccessControlEntry が常に優先されます。

```
AccessControlEntry firstAce = acl.getAccessControlEntrys(0);
aceDef.setInsertBeforeAccessControlEntry(firstAce);
```

4. AccessControlList に AccessControlEntry を追加します。

```
acl.addAccessControlEntry(aceDef);
```

例 15-33 Java を使用した AccessControlList の削除

1. アプリケーションがすでに AccessControlList を取得しているとします。

```
AccessControlList acl = ...
```

2. AccessControlList を削除します。Oracle 9*i*FS により、その AccessControlList 内のすべての AccessControlEntry が自動的に削除されます。

```
acl.free();
```

PublicObject への ACL の適用

Oracle 9*i*FS で作成される各 PublicObject に、AccessControlList を適用する必要があります。AccessControlList を適用するには、次の 3 つの方法があります。

- **ACL 属性の明示的な設定：** 通常、AccessControlList を PublicObject に適用するには、PublicObject の ACL 属性を設定します。ACL 属性は、その PublicObject へのアクセス管理に使用する必要がある AccessControlList を参照します。
- **PublicObject の ACL のデフォルト設定：** 新規 PublicObject の作成時に ACL 属性を明示的に設定していない場合、ACL 属性は作成者のそのクラスのオブジェクトのデフォルト AccessControlList に自動的に設定されます。PublicObject の所有者やシステム管理者など、PublicObject に対する Grant アクセス権を持っているユーザーは、作成後の ACL 属性を更新できます。
- **プロキシによる ACL の適用：** ACL 属性を使用して AccessControlList を適用するかわりに、別の PublicObject の AccessControlList をプロキシで適用できます。ある PublicObject を別の PublicObject の SecuringPublicObject として指定すると、保護して

いる PublicObject の AccessControlList が他方の PublicObject へのアクセス制御に使用されます。

この項では、XML および Java を使用して PublicObject に AccessControlList を適用する方法について説明します。

注意： Oracle 9*i*FS クライアントで AccessControlList を適用する手順は、各クライアントの Oracle 9*i*FS オンライン・ヘルプを参照してください。

ACL 属性の明示的な設定

この項では、ACL 属性を明示的に設定して PublicObject に AccessControlList を適用する方法について説明します。例 15-34、例 15-35 および例 15-36 に、XML を使用して ACL 属性を設定する方法を示します。例 15-37 および例 15-38 には、この操作に Java を使用する方法を示します。

例 15-34 XML を使用した新規 PublicObject への ACL の適用

```
<?xml version="1.0" standalone="yes"?>
<FOLDER>
  <NAME>My Work-in-Progress</NAME>
  <DESCRIPTION>
     ACL used to manage access to all information pertaining to
     the Content Management project.
  </DESCRIPTION>
  <ACL>
    <NAME>MyCMWorkInProgressACL</NAME>
    <ACL RefType = "Name">Private</ACL>
    <ACEs>
      <ACCESSCONTROLENTRY>
        <GRANTEE ClassName='DirectoryGroup' RefType="Name">
           CMProjectGroup
        </GRANTEE>
        <ACCESSLEVEL>
          <DISCOVER> true </DISCOVER>
          <GETCONTENT> true </GETCONTENT>
        </ACCESSLEVEL>
        <GRANTED>true</GRANTED>
      </ACCESSCONTROLENTRY>
    </ACEs>
  </ACL>
</FOLDER>
```

注意： 例 15-34 では、他のユーザーが新規 AccessControlList を使用できないように、AccessControlList の ACL 属性を Private SystemAccessControlList に設定しています。

例 15-35 XML を使用した新規 PublicObject への既存 ACL の適用

```
<?xml version="1.0" standalone="yes"?>
<FOLDER>
    <NAME>Content Management Research</NAME>
    <ACL RefType = "Name">
      ContentManagementProjectACL
    </ACL>
    <FOLDERPATH>.</FOLDERPATH>
</FOLDER>
```

例 15-36 XML を使用した既存 PublicObject の ACL の変更

```
<?xml version="1.0" standalone="yes"?>
<FOLDER>
    <UPDATE RefType = "Name">Content Management Research</UPDATE>
    <ACL RefType = "Name">
      Public
    </ACL>
    <FOLDERPATH>.</FOLDERPATH>
</FOLDER>
```

例 15-37 Java を使用した新規 PublicObject への既存 ACL の適用

1. AccessControlList をフェッチします。

```
Collection aclColl = session.getSystemAccessControlListCollection();
AccessControlList acl =
    (AccessControlList) aclColl.getItems("Protected");
```

2. FolderDefinition を作成して ACL 属性を設定します。

```
FolderDefinition fd = new FolderDefinition(session);
fd.setAttributeByUpperCaseName(Folder.NAME_ATTRIBUTE,
     AttributeValue.newAttributeValue("Collateral"));
fd.setAttributeByUpperCaseName(Folder.ACL_ATTRIBUTE,
     AttributeValue.newAttributeValue(acl));
```

3. FolderDefinition を LibrarySession に渡して、新規 Folder を作成します。

```
Folder f = (Folder) session.createPublicObject(fd);
```

例 15-38 Java を使用したサブフォルダに対するフォルダの ACL の再帰的な適用

```
/** This example illustrates how to recursively
 * apply an ACL to multiple PublicObjects.
 */
```

1. 検出されたすべてのサブフォルダの ACL を設定する権限がユーザーに対して暗黙的に付与されるように、管理モードを有効化します。有効化しないと、サブフォルダの所有者でないユーザーや、サブフォルダの ACL で Grant 権限が付与されていないユーザーは、ACL を適用できません。

   ```
   session.setAdministrationMode(true);
   ```

2. ACL が再帰的に適用されるサブフォルダの親フォルダを取得します。

   ```
   PublicObject parentFolder, item;

   FolderPathResolver fpr = new FolderPathResolver(session);
   fpr.setRootFolder();
   parentFolder = fpr.findPublicObjectByPath("/public");
   ```

3. 親フォルダの ACL を取得します。

   ```
   AccessControlList acl = parentFolder.getAcl();
   ```

4. すべてのサブフォルダの配列を取得します。

   ```
   PublicObject[] folderItems = parentFolder.getItems();

   for (int i = 0 ; i < parentFolder.getItemCount(); i++ )
   {
   ```

5. アイテムがサブフォルダの場合は、親フォルダの ACL を適用します。

   ```
      item = ( PublicObject ) folderItems[i];
      if( item instanceof Folder )
      {
         item.setAcl( acl );
      }
   }
   ```

PublicObject の ACL のデフォルト設定

PublicObject の作成時に ACL 属性を明示的に設定しなかった場合、ACL はユーザーのそのクラスのオブジェクトのデフォルト AccessControlList を参照するように自動的に設定されます。デフォルト AccessControlList は、ユーザーの PrimaryUserProfile 内で識別されます。PrimaryUserProfile は、ユーザーのデフォルト AccessControlList を保持する次の 2 つの属性を持っています。

- **DefaultACLs:** バージョニング対象外の PublicObject のデフォルト AccessControlList の指定に使用されます。
- **DefaultVersionedACLs:** バージョニング対象の PublicObject のデフォルト AccessControlList の指定に使用されます。

どちらの属性も、名前 / 値のペアである Property のリストで構成される PropertyBundle を参照します。各 Property の Name は、AccessControlList が適用されるオブジェクトのクラス（DOCUMENT、FOLDER など）を指定します。Property の値は、そのクラスのインスタンスについて、デフォルトで適用する必要のある AccessControlList を参照します。

ユーザーのデフォルト AccessControlList を指定する方法は、「ユーザー・プロファイルの作成」を参照してください。Java を使用してユーザーの PrimaryUserProfile 内で DefaultACLs PropertyBundle に AccessControlList を追加する方法は、例 15-5 を参照してください。

注意： PropertyBundle の操作方法の詳細は、第 6 章「任意のメタデータと動作の適用」を参照してください。

プロキシによる ACL の適用

Oracle 9*i*FS には、PublicObject へのアクセスを制御するための代替手段が用意されています。ACL 属性を設定して PublicObject に AccessControlList を直接適用するかわりに、SecuringPublicObject 属性を設定して別の PublicObject の AccessControlList を適用できます。

各 PublicObject には属性 SecuringPublicObject が含まれており、この属性をオプションで使用して、この PublicObject の保護に使用する別の PublicObject を指定できます。SecuringPublicObject 属性が別の PublicObject を参照している場合は、その PublicObject の AccessControlList がかわりに使用されます。

プロキシ・セキュリティは、バージョニング対象の PublicObject を操作する場合に特に役立ちます。たとえば、ドキュメントのすべてのバージョンについて、そのドキュメントの Family を参照するように SecuringPublicObject を設定できます。これにより、ドキュメントの全バージョンのセキュリティを、バージョンごとに個別にではなく一元的にメンテナンスできます。

例 15-39 Java を使用した PublicObject の SecuringPublicObject 属性の設定

1. DocumentDefinition を作成します。

```
DocumentDefinition dd = new DocumentDefinition(session);
dd.setAttribute(Document.NAME_ATTRIBUTE,
   AttributeValue.newAttributeValue(docname));

Collection formatColl = session.getFormatCollection();
Format fmt = (Format) formatColl.getItems("Text");
```

```
dd.setFormat(fmt);

dd.setContent("A new versioned document.");

Folder hf = session.getUser().getPrimaryUserProfile().getHomeFolder();
dd.setAddToFolderOption(hf);
```

2. VersionDescriptionDefinition を作成します。

```
VersionDescriptionDefinition vdd = new VersionDescriptionDefinition(session);
vdd.setAttribute(VersionDescription.NAME_ATTRIBUTE,
    AttributeValue.newAttributeValue("Versioning1 VersionDescription"));
vdd.setAttribute("VERSIONLABEL",
    AttributeValue.newAttributeValue("Version 1.0"));
vdd.setPublicObjectDefinition(dd);
```

3. VersionDescriptionDefinition を LibrarySession に渡して、Document、VersionDescription、VersionSeries および Family を暗黙的に作成し、VersionDescription を戻します。

```
VersionDescription vd = (VersionDescription)session.createPublicObject(vdd);
```

4. VersionDescription を使用して他のコンポーネントをフェッチします。

```
Family f = vd.getFamily();
VersionSeries vs = vd.getVersionSeries();
Document d = (Document) vd.getResolvedPublicObject();
```

5. VersionDescription および VersionSeries の SecuringPublicObject を、ドキュメントの Family になるように設定します。

```
d.setSecuringPublicObject(f);
vd.setSecuringPublicObject(f);
vs.setSecuringPublicObject(f);
```

AccessControlList へのアクセスの管理

いずれのエンド・ユーザーも AccessControlList を作成できます。ユーザーが AccessControlList を作成すると、Oracle 9*i*FS では自動的にそのユーザーが AccessControlList の所有者として指定されます。

所有者は、他のユーザーが AccessControlList にアクセスする方法を制御できます。つまり、AccessControlList を検出して PublicObject に適用できるユーザーや、AccessControlList を変更して権限を付与または取り消すことができるユーザーを指定できます。AccessControlList の所有者は、それを検出して変更できるユーザーを決定するために、他の PublicObject の場合と同様に AccessControlList の ACL 属性を設定します。たとえば、他のユーザーに対して AccessControlList の検出や PublicObject へのアクセス管理への使用を禁止する場合、所有者は AccessControlList の ACL 属性を、Private SystemAccessControlList

を参照するように設定できます。また、他のユーザーに Grant 権限を付与する AccessControlList を適用し、AccessControlList をメンテナンスする職責を共有にすることもできます。Grant 権限は、AccessControlList に新規 AccessControlEntry を追加できるユーザーを決定するために使用されます。

例 15-40 XML を使用した AccessControlList へのアクセス権の付与

```
<?xml version = '1.0' standalone = 'yes'?>
<OBJECTLIST>
```

1. 他の AccessControlList へのアクセスを制御する AccessControlList を作成します。

```
 <ACCESSCONTROLLIST>
  <NAME>CMProjectACLAdministration</NAME>
  <DESCRIPTION>ACL used to allow the Content Managmeent project members to use the
AccessControlList, and project leads to manage AccessControlLists.</DESCRIPTION>
  <OWNER RefType = "Name">tturner</OWNER>
  <ACEs>
    <ACCESSCONTROLENTRY>
      <GRANTEE ClassName="DirectoryGroup" RefType="Name">
         CMProjectGroup
      </GRANTEE>
      <ACCESSLEVEL>
          <DISCOVER> true </DISCOVER>
      </ACCESSLEVEL>
      <GRANTED>true</GRANTED>
    </ACCESSCONTROLENTRY>
    <ACCESSCONTROLENTRY>
      <GRANTEE ClassName="DirectoryGroup" RefType="Name">
         CMProjectLeads
      </GRANTEE>
      <ACCESSLEVEL>
          <GRANT> true </GRANT>
      </ACCESSLEVEL>
      <GRANTED>true</GRANTED>
    </ACCESSCONTROLENTRY>
  </ACEs>
 </ACCESSCONTROLLIST>
```

2. AccessControlList の ACL 属性を設定します。

```
 <ACCESSCONTROLLIST>
  <NAME>ContentManagementProjectACL</NAME>
  <DESCRIPTION>
     ACL used to manage access to all information
     pertaining to the Content Management project.
  </DESCRIPTION>
  <OWNER RefType = "Name">tturner</OWNER>
```

```
  <ACL RefType = "Name">CMProjectACLAdministration</ACL>
  <ACEs>
    <ACCESSCONTROLENTRY>
      <GRANTEE ClassName="DirectoryGroup" RefType="Name">
         CMProjectGroup
      </GRANTEE>
      <ACCESSLEVEL>
          <DISCOVER> true </DISCOVER>
          <GETCONTENT> true </GETCONTENT>
          <SETCONTENT> true </SETCONTENT>
          <DELETE> true </DELETE>
      </ACCESSLEVEL>
      <GRANTED>true</GRANTED>
    </ACCESSCONTROLENTRY>
  </ACEs>
 </ACCESSCONTROLLIST>

</OBJECTLIST>
```

ClassObject へのアクセスの管理

ClassAccessControlList を使用すると、特定クラスの情報を作成して検索できるユーザーおよびグループを制御できます。

この項の内容は、次のとおりです。

- ClassAccessControlList の機能
- ClassAccessControlList の作成
- ClassObject への ClassAccessControlList の適用

ClassAccessControlList の機能

ClassAccessControlList は AccessControlList のサブクラスであり、ClassObject にのみ適用されます。ClassAccessControlList は AccessControlList を拡張するため、「AccessControlList の機能」および「AccessControlList の構造」で説明したすべての属性と動作を持っています。さらに、ClassAccessControlList は、ClassAccessControlList を文字列で一意に識別するための属性 UniqueName も持っています。Oracle 9*i*FS では、リポジトリを検索せずに LibrarySession から簡単に取得できるように、ClassAccessControlList のコレクションもメンテナンスされます。

ClassAccessControlList を ClassObject に適用するには、ClassObject の ClassACL 属性で ClassAccessControlList を参照します。ClassAccessControlList は、ClassObject に 2 つの特殊な権限 Create および SelectorAccess を適用します。Oracle 9*i*FS Java API では、この 2 つの権限は AccessLevel クラスの定数で表されます。

表 15-15 に、AccessLevel の定数とその使用方法を示します。

表 15-15 ClassObject の権限

定数	使用方法
AccessLevel_Create	Oracle 9*i*FS は、ユーザーに対してコンテンツ・タイプのインスタンスの作成を許可する前に、そのユーザーに ClassObject の ClassAccessControlList 内で Create 権限が付与されているかどうかを検証します。
AccessLevel_SelectorAccess	ユーザーがリポジトリ内の情報を検索すると、Oracle 9*i*FS はそのタイプの情報を戻す前に、そのユーザーに ClassObject に対する SelectorAccess が付与されているかどうかをチェックします。

ClassAccessControlList の作成

ClassAccessControlList の作成方法は、AccessControlList の場合と同じです。主な違いは、次のとおりです。

- ClassAccessControlList クラスはインスタンス化されます。
- 含まれるのは、AccessLevel_Create および AccessLevel_SelectorAccess 権限のみです。
- UniqueName が自動的に生成されます。

例 15-41 および例 15-42 に、XML および Java を使用して ClassAccessControlList を作成する方法を示します。

注意： Oracle 9*i*FS Manager で ClassAccessControlList を作成する手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

例 15-41 XML を使用した ClassAccessControlist の作成

```
<?xml version = '1.0' standalone = 'yes'?>
<CLASSACCESSCONTROLLIST>
  <NAME>ProductCatalogClassACL</NAME>
  <DESCRIPTION>
    ClassACL used to manage access any custom classes
    created for the ProductCatalog application.
  </DESCRIPTION>
  <ACEs>
    <ACCESSCONTROLENTRY>
      <GRANTEE ClassName="DirectoryGroup" RefType="Name">
         CMProjectGroup
      </GRANTEE>
```

```
      <ACCESSLEVEL>
          <SELECTORACCESS> true </SELECTORACCESS>
      </ACCESSLEVEL>
      <GRANTED>true</GRANTED>
    </ACCESSCONTROLENTRY>
    <ACCESSCONTROLENTRY>
      <GRANTEE ClassName="DirectoryGroup" RefType="Name">
         CMProjectLeads
      </GRANTEE>
      <ACCESSLEVEL>
          <CREATE> true </CREATE>
      </ACCESSLEVEL>
      <GRANTED>true</GRANTED>
    </ACCESSCONTROLENTRY>
  </ACEs>
</CLASSACCESSCONTROLLIST>
```

例 15-42 Java を使用した ClassAccessControList の作成

1. アプリケーションは、AccessControlList で権限が付与される DirectoryUser および DirectoryGroup をすでに取得しているとします。

```
DirectoryGroup CMGroup = ...
DirectoryGroup CMLeads = ...
```

2. ClassAccessControlList を作成するには、管理モードを有効化する必要があります。

```
session.setAdministrationMode(true);
```

3. 新規の ClassAccessControlListDefinition を作成します。

```
ClassAccessControlListDefinition aclDef =
      new ClassAccessControlListDefinition(session);
```

4. ClassAccessControlList の属性値を設定します。

```
aclDef.setAttributeByUpperCaseName("NAME",
         AttributeValue.newAttributeValue(
           "ProductCatalogClassACL"));
aclDef.setShared(TRUE);
```

5. ClassAccessControlListDefinition を LibrarySession に渡して、新規の ClassAccessControlList および AccessControlEntry インスタンスを作成します。

```
ClassAccessControlList classAcl =
     (ClassAccessControlList) session.createPublicObject(aclDef);
```

6. Content Management グループのメンバーにクラスのインスタンスの検索権限を付与するように、AccessControlEntryDefinition を作成します。

```
AccessControlEntryDefinition aceDef1 =
             new AccessControlEntryDefinition(session);
aceDef1.setGrantee(dg1);

AccessLevel al1 = new AccessLevel(AccessLevel.ACCESSLEVEL_SELECTORACCESS);
aceDef1.setDistinctAccessLevel(al1);

acl.grantAccess(aceDef1);
```

7. Content Management リーダー・グループのメンバーにクラスのインスタンスの作成権限を付与するように、AccessControlEntryDefinition を作成します。

```
AccessControlEntryDefinition aceDef2 =
             new AccessControlEntryDefinition(session);
aceDef2.setGrantee(dg2);

AccessLevel al2 = new AccessLevel(AccessLevel.ACCESSLEVEL_CREATE);
aceDef2.setDistinctAccessLevel(al2);

acl.grantAccess(aceDef2);
```

ClassObject への ClassAccessControlList の適用

ClassObject の ClassACL 属性で ClassAccessControlList を参照すると、ClassObject に ClassAccessControlList を適用できます。

例 15-43 に、新規 ClassObject に既存の ClassAccessControlList を適用する方法を示します。

注意： ClassObject の作成方法の詳細は、第 5 章「コンテンツ・タイプと属性の拡張」を参照してください。

例 15-43 XML を使用した ClassObject への ClassAccessControlList の適用

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <NAME>Image</NAME>
    <SUPERCLASS RefType = "Name">Document</SUPERCLASS>
    <CLASSACL RefType = "Name">ProductCatalogClassACL</CLASSACL>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>Width</NAME>
            <DATATYPE>Integer</DATATYPE>
        </ATTRIBUTE>
        <ATTRIBUTE>
            <NAME>Height</NAME>
```

```
            <DATATYPE>Integer</DATATYPE>
        </ATTRIBUTE>
    </ATTRIBUTES>
</CLASSOBJECT>
```

例 15-44 Java を使用した ClassObject への ClassAccessControlList の適用

1. 管理モードを有効化します。

```
session.setAdministrationMode(true);
```

2. 新規 ClassObject の値を保持する ClassObjectDefinition を作成します。

```
ClassObjectDefinition codef = new ClassObjectDefinition(session);
codef.setAttributeByUpperCaseName(ClassObject.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("IMAGE"));
codef.setAttributeByUpperCaseName(ClassObject.DESCRIPTION_ATTRIBUTE,
       AttributeValue.newAttributeValue("Image files"));

ClassObject superClass = session.getClassObjectByName("DOCUMENT");
codef.setSuperclass(superClass);

AttributeDefinition adef1 = new AttributeDefinition(session);
adef1.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("Height"));
adef1.setAttributeByUpperCaseName(Attribute.DESCRIPTION_ATTRIBUTE,
     AttributeValue.newAttributeValue("Image Height"));
adef1.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
     AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_INTEGER));
adef1.setAttributeByUpperCaseName(Attribute.REQUIRED_ATTRIBUTE,
              AttributeValue.newAttributeValue(false));
codef.addAttributeDefinition(adef1);

AttributeDefinition adef2 = new AttributeDefinition(session);
adef2.setAttributeByUpperCaseName(Attribute.NAME_ATTRIBUTE,
        AttributeValue.newAttributeValue("Width"));
adef2.setAttributeByUpperCaseName(Attribute.DESCRIPTION_ATTRIBUTE,
     AttributeValue.newAttributeValue("Image Width"));
adef2.setAttributeByUpperCaseName(Attribute.DATATYPE_ATTRIBUTE,
     AttributeValue.newAttributeValue(Attribute.ATTRIBUTEDATATYPE_INTEGER));
adef2.setAttributeByUpperCaseName(Attribute.REQUIRED_ATTRIBUTE,
              AttributeValue.newAttributeValue(false));
codef.addAttributeDefinition(adef2);
```

3. ClassAccessControlList コレクションから ClassAccessControlList をフェッチし、ClassObjectDefinition に適用します。

```
Collection classAclColl = session.getClassAccessControlListCollection();
```

```
ClassAccessControlList classACL =
     (ClassAccessControlList) classAclColl.getItems("PRODUCTCATALOGCLASSACL");
codef.setAttributeByUpperCaseName(ClassObject.CLASSACL_ATTRIBUTE,
    AttributeValue.newAttributeValue(classACL));
```

4. ClassObjectDefinition を LibrarySession に渡して、コンテンツ・タイプの ClassObject および Attribute インスタンスを作成します。

```
ClassObject classObj = (ClassObject) session.createSchemaObject(codef);
```

オブジェクトへの暗黙的なアクセス権の付与

Oracle 9*i*FS では、次の 2 つの場合に、情報へのすべてのアクセス権が暗黙的に付与されます。

- 管理： ユーザーがシステム管理者の場合です。
- 所有権： ユーザーがオブジェクト所有者の場合です。

管理

システム管理者には、リポジトリ内の全情報へのすべてのアクセス権が暗黙的に付与されます。いずれのユーザーも、自分を表す DirectoryUser オブジェクトの AdminEnabled 属性を設定して、管理権限を受け取ることができます。その後、そのユーザーは Oracle 9*i*FS とのセッション中に管理モードを有効化し、リポジトリ内の全情報へのすべてのアクセスを取得できます。

XML および Java を使用してユーザーに管理権限を付与する方法は、例 15-45 および例 15-46 を参照してください。

注意： Oracle 9*i*FS クライアントを使用してユーザーに管理権限を付与する手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

例 15-45 に、クライアント・アプリケーションで Oracle 9*i*FS とのユーザー・セッションの管理モードを有効化する方法を示します。

例 15-45 XML を使用したユーザーへの管理権限の付与

```
<?xml version = '1.0' standalone = 'yes'?>
<SIMPLEUSER>
  <USERNAME>gking</USERNAME>
  <PASSWORD>ifs</PASSWORD>
  <DISTINGUISHEDNAMESUFFIX>.ambiguity.com</DISTINGUISHEDNAMESUFFIX>
```

```
  <ADMINENABLED>true</ADMINENABLED>
  <HOMEFOLDERROOT>/home</HOMEFOLDERROOT>
  <EMAILADDRESSSUFFIX>@ambiguity.com</EMAILADDRESSSUFFIX>
</SIMPLEUSER>
```

例 15-46 Java を使用したユーザーへの管理権限の付与

1. ユーザーを作成するには、管理モードを有効化する必要があります。

```
session.setAdministrationMode(true);
```

2. UserManager の新規インスタンスを構成します。

```
// Creating an user using all the default options;
UserManager usermanager = new UserManager(session);
```

3. Admin_Enabled 属性を 'true' に設定するオプションの HashTable を構成します。

```
Hashtable userOptions = new Hashtable();
userOptions.put(UserManager.ADMIN_ENABLED, new Boolean(true));
```

4. ユーザーを作成します。

```
DirectoryUser dUser = usermanager.createUser("Gary.King",
                                             "ifs",
                                             userOptions);
```

例 15-47 Java を使用した管理モードの有効化

1. アプリケーションが LibrarySession を取得しているとします。

```
LibrarySession session = ....
```

2. ユーザーを作成するには、管理モードを有効化する必要があります。

```
session.setAdministrationMode(true);
```

所有権

Oracle 9*i*FS の各オブジェクトは、そのオブジェクトへのすべてのアクセス権を持つユーザーにより所有されます。ユーザーは、オブジェクトの Owner 属性で参照されることでオブジェクトの所有権が付与されます。リポジトリ内でオブジェクトが作成されると、Oracle 9*i*FS は自動的に Owner 属性を作成者に設定します。Owner 属性は、いつでも他のユーザーを参照するように変更できます。

例 15-48 XML を使用した新規 PublicObject の所有者の設定

```
<?xml verssion="1.0" standalone="yes"?>
<FOLDER>
```

```
  <NAME>My Work-in-Progress</NAME>
  <OWNER RefType = "Name">tturner</OWNER>
</FOLDER>
```

例 15-49 XML を使用した既存 PublicObject の所有者の変更

```
<?xml verssion="1.0" standalone="yes"?>
<FOLDER>
  <UPDATE RefType = "Name">My Work-in-Progress</UPDATE>
  <OWNER RefType = "Name">gking</OWNER>
</FOLDER>
```

例 15-50 Java を使用した新規 PublicObject の所有者の設定

1. FolderDefinition を作成します。

```
FolderDefinition fd = new FolderDefinition(session);
fd.setAttributeByUpperCaseName(Folder.NAME_ATTRIBUTE,
     AttributeValue.newAttributeValue("Collateral"));
```

2. FolderDefinition の Owner 属性を設定します。

```
Collection duColl = session.getDirectoryUserCollection();
DirectoryUser du = (DirectoryUser) duColl.getItems("tturner");
fd.setAttributeByUpperCaseName(Folder.OWNER_ATTRIBUTE,
    AttributeValue.newAttributeValue(du));
```

3. FolderDefinition を LibrarySession に渡して、新規 Folder を作成します。

```
Folder f = (Folder) session.createPublicObject(fd);
```

例 15-51 Java を使用した既存 PublicObject の所有者の変更

1. アプリケーションがすでにフォルダを取得しているとします。

```
Folder f = ....
```

2. Folder の Owner 属性を設定します。

```
Collection duColl = session.getDirectoryUserCollection();
DirectoryUser du = (DirectoryUser) duColl.getItems("gking");
f.setOwner(du);
```

サンプル・コード

Oracle 9*i*FS には、Java API の操作の出発点として使用できるように、2 組のサンプル・コード・ファイルが用意されています。

- ドキュメント・サンプル・ファイル
- API サンプル・コード

ドキュメント・サンプル・ファイル

Oracle 9*i*FS は、この章の例について実行可能なサンプル・コード・ファイルとともにインストールされます。各サンプル・コード・ファイルは、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/devdoc/security` ディレクトリにあります。これらのサンプル・コード・ファイルは、この章の例をテストする場合に役立ちます。ただし、適切な順序で実行しないと、2 回以上実行した場合にネーミングの競合が生じることがあります。

表 15-16 に、サンプル・コード・ファイルとそれに対応する例を示します。

表 15-16 例とサンプル・コード・ファイル

例	サンプル・コード・ファイル
例 15-1「XML を使用した DirectoryUser の作成」	`CreateSimpleUser.xml`
例 15-2「XML を使用した DirectoryUser の変更」	`UpdateDirectoryUser.xml`
例 15-3「XML を使用した PrimaryUserProfile の変更」	`UpdatePrimaryUserProfile.xml`
例 15-4「Java を使用したユーザーの作成」	`CreateUser.java`
例 15-5「Java を使用した DirectoryUser の変更」	`CreateUser.java`
例 15-6「Java を使用した DirectoryUser の削除」	`DeleteUser.java`
例 15-7「XML を使用した ExtendedUserProfile クラスの作成」	`CreateExtendedUserProfileType.xml`
例 15-8「Java を使用した ExtendedUserProfile クラスの作成」	`CreateExtendedUserProfileType.java`
例 15-9「Java を使用した ExtendedUserProfile クラスの属性の追加、削除および更新」	`UpdateExtendedUserProfileType.java`

表 15-16 例とサンプル・コード・ファイル（続く）

例	サンプル・コード・ファイル
例 15-10「Java を使用した ExtendedUserProfile クラスの削除」	DeleteExtendedUserProfileType.java
例 15-11「XML を使用した ExtendedUserProfile インスタンスの作成」	CreateExtendedUserProfile.xml
例 15-12「Java を使用した ExtendedUserProfile インスタンスの作成」	CreateExtendedUserProfile.java
例 15-13「Java を使用した ExtendedUserProfile インスタンスの削除」	DeleteExtendedUserProfiles.java
例 15-14「XML を使用した DirectoryGroup インスタンスの作成」	CreateDirectoryGroup.xml
例 15-15「XML を使用した DirectoryGroup インスタンスの変更」	UpdateDirectoryGroup.xml
例 15-16「Java を使用した DirectoryGroup インスタンスの作成」	CreateDirectoryGroup.java
例 15-17「Java を使用した DirectoryGroup インスタンスの変更」	UpdateDirectoryGroup.java
例 15-18「Java を使用した DirectoryGroup インスタンスの削除」	DeleteDirectoryGroup.java
例 15-19「XML を使用したグループ・クラスの作成」	CreateDirectoryGroupType.xml
例 15-20「Java を使用したグループ・クラスの作成」	CreateDirectoryGroupType.java
例 15-21「Java を使用したグループ・クラスの属性の追加、削除および更新」	UpdateDirectoryGroupType.java
例 15-22「Java を使用したグループ・クラスの削除」	DeleteDirectoryGroupType.java
例 15-23「XML を使用したカスタム・グループ・インスタンスの作成」	CreateCustomDirectoryGroup.xml
例 15-24「XML を使用したカスタム・グループ・インスタンスの変更」	UpdateCustomDirectoryGroup.xml
例 15-25「Java を使用したカスタム・グループ・インスタンスの作成」	CreateCustomDirectoryGroup.java
例 15-26「Java を使用したカスタム・グループ・インスタンスの変更」	UpdateCustomDirectoryGroup.java

表 15-16 例とサンプル・コード・ファイル（続く）

例	サンプル・コード・ファイル
例 15-27「Java を使用したカスタム・グループ・インスタンスの削除」	DeleteCustomDirectoryGroups.java
例 15-28「Java を使用したユーザーの認証」	どのサンプル・コード・ファイルでも、ユーザーは Oracle 9*i*FS で認証されます。
例 15-29「XML を使用した AccessControlList の作成」	CreateACL.xml
例 15-30「XML を使用した AccessControlList の変更」	UdpateACL.xml
例 15-31「Java を使用した AccessControlList の作成」	CreateACL.java
例 15-32「Java を使用した AccessControlList の変更」	UpdateACL.java
例 15-33「Java を使用した AccessControlList の削除」	DeleteACLs.java
例 15-34「XML を使用した新規 PublicObject への ACL の適用」	ApplyNewACLtoNewPO.xml
例 15-35「XML を使用した新規 PublicObject への既存 ACL の適用」	ApplyExistingACLtoNewPO.xml
例 15-36「XML を使用した既存 PublicObject の ACL の変更」	UpdateACLonExistingPO.xml
例 15-37「Java を使用した新規 PublicObject への既存 ACL の適用」	ApplyExistingACLtoNewPO.java
例 15-38「Java を使用したサブフォルダに対するフォルダの ACL の再帰的な適用」	ApplyACLRecursively.java
例 15-39「Java を使用した PublicObject の SecuringPublicObject 属性の設定」	ApplyACLbyProxy.java
例 15-40「XML を使用した AccessControlList へのアクセス権の付与」	GrantAccesstoACL.xml
例 15-41「XML を使用した ClassAccessControList の作成」	CreateCACL.xml
例 15-42「Java を使用した ClassAccessControList の作成」	CreateCACL.java

表 15-16 例とサンプル・コード・ファイル（続く）

例	サンプル・コード・ファイル
例 15-43「XML を使用した ClassObject への ClassAccessControlList の適用」	ApplyCACL.xml
例 15-44「Java を使用した ClassObject への ClassAccessControlList の適用」	ApplyCACL.java
例 15-45「XML を使用したユーザーへの管理権限の付与」	CreateAdminEnabledUser.xml
例 15-46「Java を使用したユーザーへの管理権限の付与」	CreateUser.java
例 15-47「Java を使用した管理モードの有効化」	CreateUser.java など、多くのサンプル・コード・ファイルでは、タスクを実行できるように管理モードが有効化されます。
例 15-48「XML を使用した新規 PublicObject の所有者の設定」	SetOwneronPO.xml
例 15-49「XML を使用した既存 PublicObject の所有者の変更」	UpdateOwneronPO.xml
例 15-50「Java を使用した新規 PublicObject の所有者の設定」	SetOwneronNewPO.java
例 15-51「Java を使用した既存 PublicObject の所有者の変更」	SetOwneronExistingPO.java

Java サンプル・コード・ファイルの実行

Java サンプル・コード・ファイルを実行する手順は、次のとおりです。

1. Oracle 9*i*FS ホスト・マシン上で、CLASSPATH に Java API の JAR ファイルが含まれているかどうかを確認します。
2. JDK 1.2.2 で、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/devdoc/security` ディレクトリにあるサンプル・コード・ファイルをコンパイルします。

例 15-52 Solaris でのサンプル・コード・ファイルのコンパイル

```
> cd $ORACLE_HOME/9ifs/samplecode/oracle/ifs/examples/devdoc/security
> javac CreateUser.java
```

3. ファイルのコメントをチェックし、このファイルの前に他のファイルの実行が予期されているかどうかを調べます。

4. クラスを実行して引数を指定します。引数の入力値が表示されます。すべての引数に値が必要です。値を指定しない場合は、一部またはすべての引数にデフォルトが設定されます。クラスにより結果が画面に出力されます。

例 15-53 Solaris でのサンプル・コード・ファイルの実行

```
> java oracle.ifs.examples.devdoc.security.CreateUser.java
newuser=lrichards admin=false emailAddress=lrichards@ambiguity.com acl=Public

  Running with arguments :
    user = system
    password = manager9ifs
    service = IfsDefault
    schemapassword = ifssys
    newuser = lrichards
  -----------Results----------
New User Created : lrichards
  DistinguishedName = lrichards
  AdminEnabled = false
```

XML サンプル・コード・ファイルの実行

XML 構成ファイルを実行する手順は、次のとおりです。

1. XML 構成ファイルの解析をサポートする Oracle 9*i*FS クライアントを開きます。
2. XML 構成ファイルにより実行されるタスクに該当する管理権限を持つユーザーでログインします。
3. XML 構成ファイルをインポートします。
4. 結果を Oracle 9*i*FS Manager で表示します。

API サンプル・コード

Oracle 9*i*FS は、例のサンプル・コード・ファイルのみでなく、Java API の操作開始の出発点として役立つ高度なサンプル・コードとともにインストールされます。この API サンプル・コードは、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/api` ディレクトリにあります。この API サンプル・コードは何度実行しても名前の競合が生じることはないため、ドキュメント・サンプル・コード・ファイルより操作が簡単です。

表 15-17 に、この章に関連する API サンプル・コードを示します。

表 15-17 API サンプル・コード

クラス	使用方法
`GroupSample.java`	グループを作成します。ユーザーとグループを追加してグループに移入します。
`CreateLargeGroups.java`	グループを作成します。トランザクションの使用方法を示します。
`AclSample.java`	AccessControlList を作成して変更します。
`ApplyAclToFolder.java`	フォルダ・ツリーの下位へと ACL を再帰的に適用します。プライベート・クラスを使用してメソッドをオーバーライドする方法を示します。

16

セッションおよびトランザクションの管理

第 4 章「Oracle Internet File System ドキュメントの作成」では、開発者がカスタム・アプリケーションの構築を開始できるように、Oracle 9*i*FS とのセッションを確立し、そのセッションを使用してオブジェクトを作成する方法を説明しました。この章では、セッション管理の詳細について説明します。この章の内容は、次のとおりです。

- セッションの管理
- トランザクションの管理
- サンプル・コード

セッションの管理

エージェント、プロトコル・サーバーまたは Web クライアントなど、カスタム・アプリケーションを構築する場合は、まず Oracle 9*i*FS とのセッションを確立する必要があります。このセッションで、アプリケーションはユーザーを認証し、ユーザーが操作中の情報をキャッシュし、リポジトリ内で情報の検索、フォルダのブラウズ、オブジェクトの作成、更新および削除などの操作を実行できます。

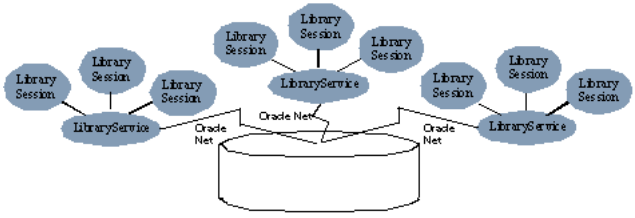
Oracle 9*i*FS のセッション管理の機能

Oracle 9*i*FS には、セッション管理用に高度な拡張性を持った高パフォーマンスのフレームワークが用意されています。このフレームワークにより、アプリケーションで処理の負荷を複数の中間層マシンに分散し、水平方向に拡張できます。また、アプリケーションは情報をキャッシュし、データベース・リソースをプールして、メモリー使用量とパフォーマンスを最適化できます。

このフレームワークでは、セッション管理に次の 2 つの Java オブジェクトが使用されます。

- LibraryService
- LibrarySession

図 16-1 セッション管理オブジェクト・モデル



LibraryService

LibraryService（サービス）は、次の操作を受け持ちます。

- データベースへの接続
- ユーザーの認証
- セッションの管理
- 情報のキャッシュ

データベースへの接続 ： 各 LibraryService は、Oracle9*i* データベースへの接続とデータベース・リソースのプーリングを受け持ちます。サービスは Oracle Net 経由でデータベースに接続し、データベースが稼働中のマシン上でのローカル接続、または中間層マシンからの

リモート接続を可能にします。その結果、サービスにより、処理負荷とメモリー需要を複数のホスト・マシン間で分散することで、アプリケーションを拡張できます。この方法でアプリケーションを分散すると、アプリケーションを各種ユーザーがアクセスできるホスト・マシンに配置することで、そのアクセス可能性も管理できます。たとえば、インターネット・ユーザーがアクセスできるように、Web アプリケーションを会社のファイアウォールの外側にあるホスト・マシンに配置できます。SMB ベース・アプリケーションは、社内のユーザーが社内ネットワーク経由でアクセスできるホスト・マシンに配置できます。会社が世界各地に分散している場合は、イントラネット・アプリケーションを各サイトのホスト・マシン間で分散させることができます。

ユーザーの認証 ： アプリケーションは、サービスを使用して Oracle 9*i*FS とのセッションを取得します。アプリケーション用のセッションを確立するために、サーバーはまずユーザーを認証します。ユーザー資格証明が有効である場合、サービスはセッションを提供し、アプリケーションはそのセッションで Oracle 9*i*FS と通信できます。

セッションの管理 ： 各サービスはセッション・セットを管理します。つまり、アクティブでないセッションのタイムアウト、セッション・アクティビティのトレース、セッションでアクセスされる情報のキャッシュのメンテナンス、セッションで実行される操作によるイベントの公開を受け持ちます。

情報のキャッシュ ： Oracle 9*i*FS サービスにより、アプリケーションのパフォーマンス、ネットワーク需要およびメモリー使用量も最適化できます。各サービスは、セッションでアクセスされる情報のキャッシュをメンテナンスします。サービスは情報をキャッシュすることで、セッションにより構成される Java オブジェクトの数と、データベースからデータを取り出すコールの数を減らします。

イベントの処理 ： Oracle 9*i*FS サービスは、イベント処理にも使用されます。サービスは、セッションで処理できるように、Oracle 9*i*FS で発生したイベントを公開します。

注意： イベント処理の手順は、第 13 章「カスタム・サーバーの作成」を参照してください。サービスの構成手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

LibrarySession

LibrarySession（セッション）は、次の操作を受け持ちます。

- セッションの状態の管理
- 操作の実行
- イベントの処理
- トランザクションの管理

- コミットされていない変更情報のキャッシュ

セッションの状態の管理 ： Oracle 9*i*FS セッションは、それ自体の状態をメンテナンスします。たとえば、セッションでは、現行のユーザーと、管理モードが有効化されているかどうかが追跡されます。

操作の実行 ： セッションは、アプリケーションで実行されるすべての操作のファクトリとして機能します。アプリケーションは、セッションを使用してオブジェクトを作成、変更および削除します。また、セッションを使用して Oracle 9*i*FS 内の情報の問合せを実行し、DirectoryUser コレクション、ClassObject コレクションおよび SystemAccessControlList コレクションなど、頻繁に使用するオブジェクトのコレクションを取得します。

トランザクションの管理 ： Oracle 9*i*FS セッションは、トランザクション内で操作を同期で実行するためにも使用されます。アプリケーションはセッションを使用してトランザクションを開始し、トランザクション中の操作はすべてセッションのキャッシュ内でのみ反映されます。その後、アプリケーションはセッションを使用し、トランザクションを強制終了してすべての変更を効率的にロールバックするか、トランザクションをコミットしてリポジトリ内で効率的に変更できます。

コミットされていない変更情報のキャッシュ ： セッションでは、そのセッションで処理中の情報のキャッシュがメンテナンスされます。セッションのキャッシュには、情報のセッション固有の状態が反映されます。たとえば、トランザクション中に実行されたすべての操作は、セッションのキャッシュ内で参照可能ですが、トランザクションがコミットされるまで他のセッションでは参照できません。セッションが変更を Oracle 9*i*FS にコミットすると、サービスは他のすべてのセッションに変更内容を通知するために、各セッションがそれぞれのキャッシュを適切に更新できます。

セッション管理クラス

Oracle 9*i*FS Java API には、アプリケーションでセッションを管理できるように、`LibraryService` および `LibrarySession` という 2 つの Java クラスが用意されています。この 2 つのクラスは、`oracle.ifs.common` パッケージにあってセッション・ユーザーの認証に使用される他の少数のクラスでサポートされます。

LibraryService

サービスは、`LibraryService` クラスのインスタンスとして表されます。`LibraryService` クラスに用意されているメソッドを使用すると、アプリケーションでは Oracle データベースに接続して Oracle 9*i*FS とのセッションを確立できます。

表 16-1 に、`LibraryService` で最も使用頻度の高いメソッドを示します。

表 16-1 LibraryService のメソッド

connect()	Oracle 9*i*FS への接続を確立します。ユーザーの資格証明と他の接続情報を保持する ConnectionOption のインスタンスを受け取ります。

LibrarySession

セッションは、`LibrarySession` クラスのインスタンスとして表されます。`LibrarySession` クラスに用意されているメソッドを使用すると、アプリケーションは Oracle 9*i*FS で Oracle 9*i*FS オブジェクトの作成、変更および削除や、リポジトリの検索などの操作を実行できます。

表 16-2 に、`LibrarySession` で最も使用頻度の高いメソッドを示します。

表 16-2 LibrarySession のメソッド

abortTransaction() beginTransaction() completeTransaction() disconnect()	トランザクションの状態を管理します。
createPublicObject() createSchemaObject() createSystemObject()	新規の永続オブジェクトを作成します。
createView()	Oracle 9*i*FS のクラスに対してカスタム SQL ビューを作成します。

表 16-2 LibrarySession のメソッド（続く）

getClassAccessControlListCollection() getClassDomainCollection() getClassObjectCollection() getDirectoryUserCollection() getExtendedPermissionCollection() getExtendedUserProfileCollection() getFormatCollection() getFormatExtensionCollection() getPermissionBundleCollection() getPolicyCollection() getSharedAccessControlListCollection() getSystemAccessControlListCollection() getValueDefaultCollection() getValueDomainCollection()	オブジェクトの特定クラスのコレクションを取得します。
getDefaultFolderRelationshipClassname() getDefaultFolderSortSpecification() getFolderPathDelimiter() getRootFolder() getWorldDirectoryGroup() setDefaultFolderRelationshipClassname() setDefaultFolderSortSpecification() setFolderPathDelimiter()	システムのデフォルトを決定し、操作します。
getDirectoryObject() getPublicObject() getSchemaObject() getSystemObject() lookupInverseInstanceLabel()	オブジェクトを識別子別に検索します。

表 16-2 LibrarySession のメソッド（続く）

getId() getSchemaVersionNumber() getSchemaVersionString() getServerName() getServerProperty() getServerVersionNumber() getServerVersionString() getServiceId() getVersionNumber() getVersionString()	サーバー構成およびバージョン情報を決定します。
grantAdministration() isAdministrationMode() setAdministrationMode()	セッションの管理モードを取得および設定します。
getObjectsLockedForSession() unlockForSession() getUser() impersonateUser() isConnected()	接続ユーザーに関する情報を取得します。
deregisterClassEventHandler() deregisterEventHandler() invokeServerMethod() processEvents() registerClassEventHandler() registerEventHandler()	イベントを処理します。

共通クラス

LibraryService による接続の確立時には、接続先となる Oracle 9*i*FS サーバーなどの接続オプション・セットとユーザーの資格証明を渡す必要があります。`oracle.ifs.common` パッケージの `ConnectOptions` クラスには、Locale、Service name および Service password など、LibraryService に渡される接続情報が保持されます。また、CleartextCredential クラスはアプリケーションが LibraryService にユーザー資格証明を渡すために使用され、LibraryService は資格証明を使用してユーザーを認証します。

表 16-3 に、ConnectOptions で最も使用頻度の高いメソッドを示します。

表 16-3 ConnectOptions のメソッド

getLocale() setLocale()	作成されるセッションのロケールを決定します。
getServiceName() setServiceName()	Oracle 9*i*FS への接続の確立に使用する LibraryService を決定します。
getServicePassword() setServicePassword()	LibraryService で Oracle 9*i*FS データベース・スキーマに接続するために必要なパスワードを決定します（ifssys データベース・ユーザー・パスワード）。

表 16-4 に、ClearTextCredential で最も使用頻度の高いメソッドを示します。

表 16-4 ClearTextCredential のメソッド

getName() setName()	ユーザー名を決定します。
getPassword() setPassword()	ユーザーのパスワードを決定します。パスワードは暗号化されずに渡されます。

セッション管理フレームワークの使用

アプリケーションは、セッション管理クラスを使用して Oracle 9*i*FS とのセッションを確立すると、Oracle 9*i*FS サービスのキャッシュ、データベース・リソース・プーリングおよびセッション管理の各機能を本質的に活用できます。また、セッション管理クラスを明示的にコールして、Oracle 9*i*FS で基本的なタスクを実行することもできます。

表 16-5 に、アプリケーションがセッション管理クラスで実行できる各種タスクと、タスクの実行方法の詳細が記載されている項または章も示します。

表 16-5 セッション管理タスク

タスク	章 / 項
データベースへの接続とユーザーの認証	第 4 章「Oracle Internet File System ドキュメントの作成」。セッション管理クラスを使用してデータベースに接続し、ユーザーの資格証明を認証し、Oracle 9*i*FS とのセッションを確立する方法を説明しています。

表 16-5 セッション管理タスク（続く）

タスク	章 / 項
オブジェクトの作成	第 4 章「Oracle Internet File System ドキュメントの作成」。LibrarySession クラスと Definition クラスを使用してドキュメントを作成する方法を説明しています。
リポジトリの問合せ	第 8 章「検索アプリケーションの構築」。LibrarySession クラスを使用して、リポジトリの問合せに使用する Selector と Search を構成する方法を説明しています。
トランザクションの管理	第 16 章の「トランザクションの管理」の項。アプリケーションでセッション中にメソッドをコールして、トランザクション中に操作を同期で実行する方法を説明しています。
イベントの処理	第 13 章「カスタム・サーバーの作成」。アプリケーションでセッション管理クラスを使用してイベントを転送し、処理する方法を説明しています。
トレース	このマニュアルでは、このトピックは取り扱いません。

トランザクションの管理

アプリケーションが Oracle 9*i*FS 内でオブジェクトの作成、変更または削除などの操作を実行すると、変更内容はデータベースに即時にコミットされます。ただし、アプリケーションが一連の依存操作をまとめてコミットする必要が生じることがあります。つまり、アプリケーションは、すべての操作を正常に実行できない場合に、操作がまったく実行されないことを保証する必要があります。したがって、操作はアトミックにコミットまたはロールバックされることが必要です。Oracle 9*i*FS のセッション管理フレームワークでは、アプリケーションは操作をトランザクション・ブロックにラップして、変更がデータベースにコミットされる時期を制御できます。

トランザクションの開始、強制終了およびコミット

Oracle 9*i*FS Java API の `LibrarySession` クラスには、トランザクション管理に使用するメソッド `beginTransaction()`、`abortTransaction()` および `completeTransaction()` が用意されています。この 3 つのメソッドにより、アプリケーションはトランザクション・ブロックの境界を定義し、トランザクションでの操作の実現可能性をテストし、操作による変更を成否に基づいてコミットまたはロールバックできます。

トランザクションは次の手順で管理されます。

1. **トランザクションの開始：** アプリケーションが LibrarySession の `beginTransaction()` をコールして、トランザクションを開始します。
2. **操作の実行：** トランザクション中に、Oracle 9*i*FS オブジェクトを操作する操作はセッションのキャッシュ内で実行され、実際にリポジトリ内で実行されるわけではありません。これにより、セッションは操作をすべて正常に実行できることを保証できます。

3. **トランザクションの強制終了：** 操作が失敗した場合、アプリケーションは `abortTransaction()` メソッドをコールして、セッションのキャッシュに対して行われた変更をロールバックできます。
4. **トランザクションの完了：** すべての操作が成功すると、アプリケーションは `completeTransaction()` メソッドをコールして、リポジトリ内で変更をコミットできます。変更がリポジトリにコミットされると、サービスは変更内容を他のセッションでアクセスできるようにサービス・キャッシュに伝播させます。

例 16-1 に、アプリケーションで LibrarySession を使用してトランザクションを管理する方法を示します。

例 16-1 LibrarySession を使用したトランザクションの管理

```
/*
 * The following example illustrates how to delete a group
 * and all of its members in a single transaction.
 * If a member cannot be removed, the group will not be deleted.
 */
```

1. アプリケーションがすでにグループを取得しているとします。

```
DirectoryGroup dg = ....
DirectoryObject[] members = dg.getAllMembers();
DirectoryObject m;
```

2. トランザクションを開始します。

```
Transaction transaction = session.beginTransaction();
```

3. try ブロックを開始して操作をテストし、例外をキャッチします。

```
try
{
```

4. グループをループして各ユーザーを削除します。

```
  int count = (members == null) ? 0 : members.length;
  for (int i = 0; i < size; i++)
  {
    m = members[i];
    dg.removeMember(m);
  }
```

5. 例外が発生せずにトランザクションが for ループの終わりに達した場合は、トランザクションを完了し、変更をデータベースにコミットします。

```
  session.completeTransaction(transaction);
}
catch (IfsException e)
```

```
{
```

6. メンバーの削除中に例外が発生した場合は、トランザクションを強制終了し、セッション・キャッシュ内の変更をロールバックします。

```
  session.abortTransaction(transaction);
  throw e;
}
```

トランザクションでの DDL 操作

データベース内でデータ定義言語（DDL）操作（表、ビューおよび列の作成など）を実行するトランザクション中の Oracle 9*i*FS の操作も、データベースに即時にコミットされます。`abortTransaction()` メソッドをコールしても、データベースの変更はロールバックされません。次の Java API メソッドは、データベース内で DDL 操作を実行します。

- `LibrarySession.createSchemaObject(ClassObjectDefinition def)` は、新規 ClassObject の表およびビューを作成します。
- `ClassObject.free()` は、ClassObject の表およびビューを削除します。
- `ClassObject.addAttribute(AttributeDefinition def)` は、ClassObject の表およびビューに Attribute の列を作成します。
- `ClassObject.removeAttribute(Attribute attr)` は、ClassObject の表およびビューから Attribute の列を削除します。

変更された LibraryObject の可用性

変更された LibraryObject は、次の規則に従ってセッションで使用可能になります。

- LibraryObject に対するコミットされていない変更を参照できるのは、その変更を行ったセッション（起点セッション）のみです。セッションは、トランザクションが完了するまでは他のセッションで変更されないように、変更対象の LibraryObject のロックを保持します。
- コミット済みの変更は起点セッションに同期で伝播し、そのセッションで即時に参照できるようになります。
- コミット済みの変更は、起点セッションを管理するサービス内の他のすべてのセッションに非同期で伝播します。この種の変更は、そのサービス内の他のすべてのセッションで `EventPollerPeriod` 秒以内に参照できるようになります。`EventPollerPeriod` は、サービス・プロパティ・ファイルに指定された値です。
- コミット済みの変更は、サービスにより他の全サービスの他の全セッションに非同期で伝播されます。変更内容は、各サービス内でそのサービス用に構成可能な期間中に参照できます。この期間は、サービスのプロパティ・ファイル内のプロパティから次のように計算できます。

```
1000 × (PollForEventsFromOtherServicesPeriod  +
```

```
TransportEventsToOtherServicesPeriod) +
EventPollerPeriod milliseconds
```

トランザクションへの Java データベース・コール（JDBC）の埋込み

一部のカスタム・アプリケーションでは、Oracle 9*i*FS のトランザクション管理機能を、Oracle 9*i*FS で実行される操作の管理のみでなく、外部データベース・スキーマで実行される操作の管理にも使用する必要があります。たとえば、Oracle 9*i*FS を、製品データが格納される別のデータベース・スキーマと統合するアプリケーションを構築するとします。カスタム・アプリケーションは、Oracle 9*i*FS 内で設計ドキュメントが更新されるたびに、他方のデータベース・スキーマ内の対応する製品のステータスも更新する必要があります。製品を正常に更新できない場合は、設計ドキュメントの変更内容をロールバックする必要があります。

Oracle 9*i*FS Java API の `oracle.ifs.server.sql` パッケージには、サーバー側 Java クラスのセットが用意されており、カスタム・アプリケーションは、Oracle 9*i*FS オブジェクトと外部データベースに対する操作を単一トランザクションで実行するオーバーライドを実装できます。各クラスで、アプリケーションは外部データベース・スキーマに接続し、SQL 文を作成し、その文を実行できます。各クラスでは、外部データベースに対してこの種のタスクを実行する JDBC コールがカプセル化されるため、Oracle 9*i*FS では操作がコミットおよびロールバックされる方法を絶えず制御できます。

サーバー側で JDBC 操作の実行に使用される Java クラスのパッケージは、次のとおりです。

- **IfsConnection:** `IfsConnection class` は、外部データベースへの JDBC 接続の確立に使用されます。このクラスにより java.sql.Connection がカプセル化されます。
- **IfsStatement:** `IfsStatement` クラスは、静的 SQL 文の作成と実行に使用されます。このクラスにより java.sql.Statement がカプセル化されます。
- **IfsPreparedStatement:** `IfsPreparedStatement` は、パラメータ変数を含むプリコンパイル済みの SQL 文の作成と実行に使用されます。このクラスにより java.sql.PreparedStatement がカプセル化されます。
- **IfsCallableStatement:** `IfsCallableStatement` クラスは、SQL ストアド・プロシージャの作成と実行に使用されます。このクラスにより java.sql.CallableStatement がカプセル化されます。

IfsConnection

カスタム・アプリケーションは、データベース・スキーマとのセッションを表すために IfsConnection クラスをインスタンス化します。そのために、アプリケーションのセッションのサーバー側表現（S_LibrarySession など）の `getIfsConnection()` メソッドがコールされます。

例 16-2 IfsConnection を使用したデータベース・スキーマへの接続

S_LibrarySession を使用して IfsConnection オブジェクトを構成します。

```
IfsConnection conn = m_Session.getIfsConnection();
```

その後、IfsConnection インスタンスを使用して、SQL 文を表すために使用される Java オブジェクト IfsStatement、IfsPreparedStatement および IfsCallableStatement が構成されます。IfsConnection を使用してこれらのオブジェクトを構成する方法は、例 16-3、例 16-4 および例 16-5 を参照してください。

IfsStatement

IfsConnection オブジェクトの構成後は、それを使用して IfsStatement を取得できます。IfsStatement は、静的 SQL 文、つまり、バインド変数を使用しない文の構成に使用されます。IfsStatement を使用する手順は、次のとおりです。

1. IfsConnection を、外部データベース・スキーマに接続するように構成します。
2. IfsConnection の `createStatement()` メソッドをコールして IfsStatement オブジェクトを構成します。
3. SQL 文を保持する文字列を構成します。
4. SQL 文を IfsStatement オブジェクトの `execute()` または `executeUpdate()` メソッドに渡して SQL を実行します。
5. 発生した SQLException をキャッチします。SQLException をキャッチすると、サーバー側オーバーライドは、JDBC 操作を含むトランザクションに対する影響を処理できます。
6. 最後に、IfsStatement の `dispose()` メソッドをコールします。

例 16-3 IfsStatement を使用した静的 SQL 文の構成および実行

```
protected void extendedPreInsert(OperationState opState,
  S_LibraryObjectDefinition def) throws IfsException
{
  super.extendedPreInsert(opState, def);
```

1. S_LibrarySession を使用して IfsConnection を構成します。

   ```
   IfsConnection conn = m_Session.getIfsConnection();

   try
   {
   ```

2. IfsConnection を使用して IfsStatement を構成します。

   ```
   IfsStatement stmt = conn.createStatement();
   ```

3. SQL 文を保持する文字列を構成します。

```
AttributeValue av = def.getAttribute("NAME");
String recipe = av.getString(m_Session);

S_DirectoryUser user = m_Session.getUser();
String name = user.getName();

String sqlstmt = "insert into your_custom_table (chef, recipe) "
                 + "values ('wtalman','Curry Lamb')";
```

4. SQL 文を IfsStatement オブジェクトに渡して SQL を実行します。

```
  stmt.execute(sqlstmt);
}
```

5. 発生した SQLException をキャッチします。

```
catch (SQLException e)
{
  throw new IfsException(98765, e);
}
```

6. 最後に、IfsStatement をクローズします。

```
  finally
  {
    if (stmt != null)
    {
      stmt.dispose();
    }
  }
}
```

IfsPreparedStatement

IfsStatement のかわりに、IfsPreparedStatement を使用してプリコンパイル済み SQL 文をコールできます。IfsPreparedStatement を使用する手順は、次のとおりです。

1. IfsConnection を、外部データベース・スキーマに接続するように構成します。
2. バインド変数のプレースホルダを持つ SQL 文を保持するように、文字列を構成します。
3. SQL 文を IfsConnection オブジェクトの `prepareStatement()` メソッドに渡して、IfsPreparedStatement オブジェクトを構成します。
4. バインド変数を IfsPreparedStatement オブジェクトの `setString()` メソッドに渡します。

5. `IfsPreparedStatement` メソッドの `execute()` メソッドをコールして SQL 文を実行します。
6. 発生した SQLException をキャッチします。SQLException をキャッチすると、サーバー側オーバーライドは、JDBC 操作を含むトランザクションに対する影響を処理できます。
7. 最後に、IfsPreparedStatement の `dispose()` メソッドをコールします。

例 16-4 IfsPreparedStatement を使用したプリコンパイル済み SQL 文のコール

```
protected void extendedPreUpdate(OperationState opState,
  S_LibraryObjectDefinition def) throws IfsException
{
  super.extendedPreUpdate(opState, def);
```

1. S_LibrarySession を使用して IfsConnection を構成します。

```
IfsConnection conn = m_Session.getIfsConnection();

try
{
```

2. SQL 文を保持する文字列を構成します。

```
AttributeValue av = def.getAttribute("NAME");
String recipe = av.getString(m_Session);
S_DirectoryUser user = m_Session.getUser();
String name = user.getName();
String sqlstmt = "insert into your_custom_table (chef, recipe) " +
                 "values (?, ?)";
```

3. SQL 文を IfsConnection オブジェクトに渡して IfsPreparedStatement オブジェクトを構成します。

```
IfsPreparedStatement pstmt = conn.prepareStatement(sqlstmt);
```

4. バインド変数を渡します。

```
pstmt.setString(1, name);
pstmt.setString(2, recipe);
```

5. SQL 文を実行します。

```
  pstmt.execute(sqlstmt);
}
```

6. SQLException をキャッチします。

```
catch (SQLException e)
{
```

```
    throw new IfsException(98765, e);
  }
```

7. 最後に、IfsPreparedStatement をクローズします。

```
    finally
    {
      if (pstmt != null)
      {
         pstmt.dispose();
      }
    }
  }
```

IfsCallableStatement

IfsConnection を使用して、SQL ストアド・プロシージャを実行する IfsCallableStatement を構成することもできます。IfsCallableStatement を使用する手順は、次のとおりです。

1. IfsConnection を、外部データベース・スキーマに接続するように構成します。
2. 引数のプレースホルダを持つ SQL プロシージャ・コールを保持するように、文字列を構成します。
3. SQL 文を IfsConnection オブジェクトの `prepareCall()` メソッドに渡して、IfsCallableStatement オブジェクトを構成します。
4. 引数を IfsCallableStatement オブジェクトの `setString()` メソッドに渡します。
5. IfsCallableStatement メソッドの `execute()` メソッドをコールして SQL 文を実行します。
6. 発生した SQLException をキャッチします。SQLException をキャッチすると、サーバー側オーバーライドは、JDBC 操作を含むトランザクションに対する影響を処理できます。
7. 最後に、IfsCallableStatement の `dispose()` メソッドをコールします。

例 16-5 IfsCallableStatement を使用した SQL ストアド・プロシージャの実行

```
protected void extendedPreFree(OperationState opState,
  S_LibraryObjectDefinition def) throws IfsException
{
  super.extendedPreFree(opState, def);
```

1. S_LibrarySession を使用して IfsConnection を構成します。

```
    IfsConnection conn = m_Session.getIfsConnection();

    try
```

```
{
```

2. ストアド・プロシージャの SQL コールを保持するように文字列を構成します。

```
AttributeValue av = def.getAttribute("NAME");
String recipe = av.getString(m_Session);
String sqlstmt = "{call yourcustompackage.test(?)}";
```

3. SQL 文を IfsConnection オブジェクトに渡して IfsCallableStatement オブジェクトを構成します。

```
IfsCallableStatement cstmt = conn.prepareCall(sqlstmt);
```

4. バインド変数を渡します。

```
cstmt.setString(1, recipe);
```

5. SQL 文を実行します。

```
  cstmt.execute();
}
```

6. SQLException をキャッチします。

```
 catch (SQLException e)
  {
    // should write your own exception message
    throw new IfsException(98765, e);
  }
```

7. 最後に、IfsCallableStatement をクローズします。

```
  finally
  {
    if (cstmt != null)
    {
       cstmt.dispose();
    }
  }
}
```

制限事項

Oracle 9*i*FS では、Oracle 9*i*FS トランザクション内での JDBC 操作の実行に次の制限が適用されます。

Oracle 9*i*FS 表でのデータ操作

Oracle 9*i*FS の表で許可される有効な操作はすべて、Oracle 9*i*FS Java API を通じて公開されます。Oracle 9*i*FS では、Oracle 9*i*FS スキーマ内の表に対する JDBC 操作の実行はサポートされません。スキーマ・オブジェクト（表やビューなど）やスキーマに格納されているデータ（ドキュメント・データなど）を直接操作すると、Oracle 9*i*FS サービスによりメンテナンスされているキャッシュが失効し、Oracle 9*i*FS オブジェクトが破損することがあります。

DDL 文

CREATE TABLE や DROP TABLE などの DDL 文をコールする JDBC 接続間のトランザクションの管理には、Oracle 9*i*FS は使用できません。DDL 文は Oracle データベースにより暗黙的にコミットされ、Oracle 9*i*FS のトランザクション管理に干渉する可能性があります。

IfsConnection のメソッド

IfsConnection の次のメソッドは JDBC トランザクションに対応付けられており、Oracle 9*i*FS 接続中のこの種の操作の使用を制限するために、即時に SQLException を発生させます。

- `setAutoCommit(boolean autoCommit)`
- `public boolean isReadOnly()`
- `public String getCatalog()`
- `public void setTransactionIsolation(int level)`

明示的なコミットおよびロールバック

読取り / 書込み接続プールのキャッシュに使用される接続は、明示的にコミットまたはロールバックしないでください。明示的に実行すると、Oracle 9*i*FS のトランザクション管理機能は IfsException に応答して効率的にロールバックできません。

サンプル・コード

Oracle 9*i*FS は、Java API の操作開始の出発点として役立つサンプル・コードとともにインストールされます。この API サンプル・コードは、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/api` ディレクトリにあります。

表 16-6 に、この章に関連する API サンプル・コードを示します。

表 16-6 API サンプル・コード

クラス	使用方法
`BaseSample.java`	サービスを開始し、セッションを確立し、切断します。
`CreateLargeGroups.java`	トランザクションの使用方法を示します。

17

コンテンツ・タイプの動作のカスタマイズ

この章では、コンテンツ・タイプの動作をカスタマイズする方法について説明します。この章の内容は、次のとおりです。

- **コンテンツ・タイプの動作のカスタマイズ： 概要：** Oracle 9*i*FS での動作をカスタマイズする方法を説明します。以降の項では、動作をカスタマイズするための 3 つのメソッドの使用手順を説明します。
- **カスタムの Bean 側 Java クラスの作成：** 各クラスを使用して、カスタム・アプリケーション・インタフェースを実装します。
- **サーバー・オーバーライドの作成：** Oracle 9*i*FS サーバーの動作を変更するための、サーバー側 Java クラスのオーバーライドです。
- **Tie クラスの置換：** Oracle 9*i*FS にデフォルトで用意されているコンテンツ・タイプの動作をカスタマイズするために使用します。
- サンプル・コード

コンテンツ・タイプの動作のカスタマイズ： 概要

第 5 章「コンテンツ・タイプと属性の拡張」で説明したように、カスタム・コンテンツ・タイプを作成すると、拡張するコンテンツ・タイプの動作が自動的に継承されます。Java でプログラミングしなくても、Oracle 9*i*FS に組み込まれているすべてのコンテンツ管理機能をカスタムの情報タイプに使用できます。アプリケーション・ユーザーは、Oracle 9*i*FS にデフォルトで用意されているクライアントとプロトコル・サーバーを使用して、カスタム・コンテンツ・タイプのインスタンスのライフサイクルを解析、レンダリングおよび管理できます。

コンテンツ・タイプの動作の実装方法

第 3 章「Java API の概要」で説明したように、デフォルトの各コンテンツ・タイプは、その動作を実装する Java クラスのセットを持っています。HTML へのドキュメント変換やドキュメントのサマリー生成などの動作は、これらの Java クラスのメソッドを通じて実装されます。

- **Bean 側 Java クラス：** Bean 側 Java クラスは、コンテンツ・タイプを操作するためのプライマリ・インタフェースを提供します。各コンテンツ・タイプの Bean 側 Java クラスは、コンテンツ・タイプを表す ClassObject の BeanClassPath 属性で参照されます。この種のクラスは、`oracle.ifs.beans` パッケージで公開されています。
- **サーバー側 Java クラス：** サーバー側 Java クラスには、コンテンツ・タイプの動作を実装するコードが含まれています。サーバー側 Java クラスのネーミング規則 S_<classname> は、Bean 側クラスのネーミングにパラレルになっています。各コンテンツ・タイプのサーバー側 Java クラスは、コンテンツ・タイプを表す ClassObject の ServerClassPath 属性で参照されます。この種のクラスは、`oracle.ifs.server` パッケージで公開されています。

コンテンツ・タイプの動作をカスタマイズするために、Bean 側とサーバー側の Java クラスを拡張できます。拡張 Bean 側 Java クラスを作成すると、アプリケーションはコンテンツ・タイプ用に実装されたカスタム属性および動作にアクセスできます。コンテンツ・タイプのサーバー側 Java クラスを拡張すると、Oracle 9*i*FS での挿入、更新および削除のような特定の操作の実行方法を変更できます。

表 17-1 のように、Bean 側とサーバー側の Java クラスにより、コンテンツ・タイプ階層がパラレル化されます。たとえば、S_PublicObject.class は PublicObject コンテンツ・タイプの動作を実装します。S_Document.class は Document コンテンツ・タイプの動作を実装します。Document コンテンツ・タイプが PublicObject 型を拡張して PublicObject 型の属性を継承するのと同様に、S_Document.class は S_PublicObject.class を拡張してそのメソッドを継承し、Document コンテンツ・タイプに関連する新規メソッドを追加します。

各コンテンツ・タイプの Bean 側 Java クラスとサーバー側 Java クラスの間には、Tie クラスがあります。Tie クラスは、デフォルトの Java クラスを変更せずに、クラス階層内の必要な位置にカスタム・コードを挿入するためのプレースホルダです。

表 17-1 コンテンツ・タイプの Java クラス階層

コンテンツ・タイプ	Bean 側 Java クラス	サーバー側 Java クラス
LibraryObject	java.lang.Object oracle.ifs.beans.LibraryObject oracle.ifs.beans.TieLibraryObject	java.lang.Object oracle.ifs.server.S_LibraryObject oracle.ifs.server.S_TieLibraryObject
PublicObject	oracle.ifs.beans.PublicObject oracle.ifs.beans.TiePublicObject	oracle.ifs.server.S_PublicObject oracle.ifs.server.S_TiePublicObject
Document	oracle.ifs.beans.Document oracle.ifs.beans.TieDocument	oracle.ifs.server.S_Document oracle.ifs.server.S_TieDocument
Folder	oracle.ifs.beans.Folder oracle.ifs.beans.TieFolder	oracle.ifs.server.S_Folder oracle.ifs.server.S_TieFolder

カスタム・コンテンツ・タイプを作成する場合、カスタム Java クラスを実装する必要はありません。カスタム・コンテンツ・タイプは、拡張対象となるコンテンツ・タイプの Java クラスに自動的に対応付けられます。たとえば、Document コンテンツ・タイプを拡張するコンテンツ・タイプ Image を作成すると、Oracle 9*i*FS では自動的に Document.class と S_Document.class を使用して Image コンテンツ・タイプの動作が実装されます。

Oracle 9*i*FS Java API を使用した動作のカスタマイズ

Oracle 9*i*FS の Java クラスを拡張して、コンテンツ・タイプのカスタム動作を実装できます。Java API を拡張するには、次の 3 つの方法があります。

- **カスタムの Bean 側 Java クラスの作成：** カスタムの Bean 側 Java クラスを実装して、カスタム・コンテンツ・タイプ用のカスタム・アプリケーション・インタフェースを提供します。たとえば、メソッド `generateThumbnail()` を含むカスタムの Image Java クラスを実装し、イメージのサムネールにアクセスする特殊メソッドを提供できます。
- **サーバー・オーバーライドの作成：** オーバーライドを実装し、Oracle 9*i*FS サーバーにより操作が実行された時点で実行されるカスタム・タスクをトリガーします。たとえば、ドキュメントが更新されるたびにドキュメント所有者に通知する必要があるという、カスタム・ビジネス・ルールを実装するとします。このルールを実装するには、通知を生成する `extendedPreUpdate()` オーバーライドを作成します。このオーバーライドは、ドキュメントの更新操作が実行される前に、Oracle 9*i*FS により自動的にコールされます。
- **Tie クラスの置換：** Tie クラスを置き換えて、デフォルトのコンテンツ・タイプの動作を拡張します。たとえば、TiePublicObject クラスを、メソッド `archive()` を含むカスタム TiePublicObject クラスに置換できます。`archive()` メソッドは、Document や Folder など、TiePublicObject のすべての派生クラスに継承されます。

カスタム Java クラスを実装する場合の推奨事項は、次のとおりです。

- **Bean 側クラスとサーバー側クラスの分離：** Bean 側クラスとサーバー側クラスの両方を作成し、カスタム・コンテンツ・タイプの動作を実装することをお薦めします。コンテンツ・タイプの動作を実装するコードをサーバー側に置いてから、そのコンテンツ・タイプの属性と機能性にアプリケーションからアクセス可能にするコンビニエンス・メソッドを提供します。Bean 側コードとサーバー側コードを分離すると、アプリケーションは将来のリリースの Oracle 9*i*FS Java API で軽量クライアント用のリモート可能 API を提供できます。
- **Tie クラスの拡張：** カスタム・コンテンツ・タイプの動作を拡張する場合、カスタム Java クラスは、コンテンツ・タイプのスーパークラスの Java クラス（Document など）ではなく Tie クラス（TieDocument など）を拡張する必要があります。このプログラミングにより、クラスは Tie クラスに実装されているカスタム・メソッドも確実に継承します。

動作の拡張を開始する前に

動作の拡張は複雑なタスクです。カスタム JavaBeans、サーバー・オーバーライドおよび Tie クラスに取り組むのは、Oracle 9*i*FS およびその Java API に十分な経験を積むまで延期することをお薦めします。特に、次の分野の知識と実務経験が必要です。

- Oracle 9*i*FS オブジェクトの動作を知るための Bean 側 API に関する十分な知識
  - オブジェクトの作成方法
  - オブジェクト属性の設定方法
  - Document や Folder などの関連オブジェクトの連動
- 属性値と定義クラスの動作を知るための十分な知識
  - 定義オブジェクト内の属性値を保持する Oracle 9*i*FS のデータ構造体である AttributeValue クラスの使用方法
  - Bean 側の属性値を設定および取得する方法
  - サーバー側の属性値を設定および取得する方法
  - 属性値セットの操作方法
  - 基本的なデータ型を使用して属性値を定義する方法
  - 値のデフォルトの使用方法
  - 値のドメインの使用方法
- Oracle 9*i*FS のトランザクション・モデルに関する十分な知識

カスタムの Bean 側 Java クラスの作成

Oracle 9*i*FS Java API の Bean 側 Java クラスには、アプリケーションで情報の作成と取出し、属性と内容の取得と設定および関係の横断を行うためのプライマリ・インタフェースが用意されています。カスタム・コンテンツ・タイプを作成すると、アプリケーションは Oracle 9*i*FS の Bean 側 Java クラスを使用して、そのコンテンツ・タイプのインスタンスを操作できます。たとえば、カスタム属性を取得および設定するには、アプリケーションは LibraryObject クラスの `getAttribute()` および `setAttribute()` メソッドをコールします。カスタム・コンテンツ・タイプの新規インスタンスを作成するには、アプリケーションは LibraryObjectDefinition の `getAttributeByUpperCaseName()` および `setAttributeByUpperCaseName()` をコールします。

Bean 側 Java クラスを拡張すると、アプリケーションで情報を簡単に操作できるようにするメソッドを提供できます。たとえば、`getImageWidth()` や `setImageWidth()` など、カスタム属性を取得および設定するためのコンビニエンス・メソッドを実装できます。また、`getBookComponents()` など、ユーザーが複合コンテンツ・タイプを構成できるように、カスタムの関係を作成および取得するメソッドを実装できます。

Bean 側 Java クラスの実装例

カスタムの Bean 側 Java クラスを実装する手順は、次のとおりです。

1. Bean 側クラスの宣言
2. コンストラクタの作成
3. カスタム・メソッドの作成

Bean 側クラスの宣言

最初に、クラス宣言を含める必要があります。Bean 側クラスは、カスタム・コンテンツ・タイプが拡張するコンテンツ・タイプ（Document など）の Bean 側 Tie クラスを拡張する必要があります。クラスを `MyCompany.MyApp.beans` などの新規カスタム・パッケージに入れます（このクラスを `oracle.ifs.beans` パッケージに追加しないでください）。

例 17-1 クラスの宣言

```
public class Image extends TieDocument
```

コンストラクタの作成

Bean 側 Java クラスでコンストラクタを実装し、そのコンストラクタでスーパークラスのコンストラクタをコールする必要があります。表 17-2 に、コンストラクタのパラメータを示します。

表 17-2 コンストラクタのパラメータ

パラメータ	データ型	説明
session	S_LibrarySession	現行の LibrarySession。
ID	java.lang.Long	コンテンツ・タイプの既存インスタンスの ID。
classID	LONG	作成中のプロセスに含まれるコンテンツ・タイプのインスタンスのクラス ID。
data	S_LibraryObjectData	コンテンツ・タイプの既存インスタンスのデータ・コンポーネント。

例 17-2 コンストラクタの作成

```
public Image(LibrarySession session,
             Long id,
             Long classID,
             S_LibraryObjectData data)
throws IfsException
{
  super(session, id, classID, data);
}
```

カスタム・メソッドの作成

Bean 側 Java クラスは、アプリケーションでコンテンツ・タイプを操作するためのプライマリ・インタフェースを提供します。コンテンツ・タイプのカスタム属性および動作にクライアント・アプリケーションからアクセスできるように、必要なメソッドを実装する必要があります。たとえば、コンテンツ・タイプのカスタム属性を取得および設定するコンビニエンス・メソッドを実装できます。

例 17-3 コンビニエンス・メソッドの作成

```
  public void setImageWidth(int newValue)
  throws IfsException
  {
    AttributeValue av = AttributeValue.newAttributeValue(newValue);
    setAttribute("WIDTH", av);
  }

  public void setImageHeight(int newValue)
  throws IfsException
  {
```

```
    AttributeValue av = AttributeValue.newAttributeValue(newValue);
    setAttribute("HEIGHT", av);
  }

  public int getImageWidth(LibrarySession session)
  throws IfsException
  {
    AttributeValue av = getAttribute("WIDTH");
    int width = av.getInteger(session);
    return width;
  }

  public int getImageHeight(LibrarySession session)
  throws IfsException
  {
    AttributeValue av = getAttribute("HEIGHT");
    int height = av.getInteger(session);
    return height;
  }
```

また、処理タスクを実行するメソッドを組み込むこともできます。たとえば、インスタンスの属性値に対して算術関数を実行してデータを導出するメソッドを提供したり、あるドキュメントに関連する全ドキュメントを取り出すメソッドを提供できます。

例 17-4 処理メソッドの作成

```
public PublicObject[] getBooks(LibrarySession session)
throws IfsException
{
  PublicObject[] relObjs =
     sel.getLeftwardRelationshipObjects("BOOK_RELATIONSHIP");

  return relObjs;
 }
```

注意： 例 17-4 では、Selector を使用して Image を含む Book を取得しています。Selector の使用方法は、第 8 章「検索アプリケーションの構築」を参照してください。

詳細例

次の例に、カスタム・コンテンツ・タイプ Image 用の Bean 側 Java クラスに関するサンプル・コード全体を示します。

例 17-5 Bean 側 Java クラス Image の作成

```
package MyCompany.MyApp.beans;

import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.TieDocument;
import oracle.ifs.beans.PublicObject;
import oracle.ifs.common.AttributeValue;
import oracle.ifs.common.IfsException;
import oracle.ifs.server.S_LibraryObjectData;

public class Image extends TieDocument
{
  public Image(LibrarySession session,
               Long id,
               Long classID,
               S_LibraryObjectData data)
  throws IfsException
  {
    super(session, id, classID, data);
  }

    public void setImageWidth(int newValue)
  throws IfsException
  {
    AttributeValue av = AttributeValue.newAttributeValue(newValue);
    setAttribute("WIDTH", av);
  }

  public void setImageHeight(int newValue)
  throws IfsException
  {
    AttributeValue av = AttributeValue.newAttributeValue(newValue);
    setAttribute("HEIGHT", av);
  }

  public int getImageWidth(LibrarySession session)
  throws IfsException
  {
    AttributeValue av = getAttribute("WIDTH");
    int width = av.getInteger(session);
    return width;
```

```
  }

  public int getImageHeight(LibrarySession session)
  throws IfsException
  {
    AttributeValue av = getAttribute("HEIGHT");
    int height = av.getInteger(session);
    return height;
  }

public PublicObject[] getBooks(LibrarySession session)
throws IfsException
{
  PublicObject[] relObjs = getLeftwardRelationshipObjects("BOOK_RELATIONSHIP");

  return relObjs;
 }
}
```

カスタムの Bean 側 Java クラスの配置

コンテンツ・タイプ用のカスタムの Bean 側 Java クラスを配置する手順は、次のとおりです。

1. Oracle 9*i*FS サーバーのホスト・マシン上で Java クラスをコンパイルし、そのクラスのパッケージに対応するディレクトリに置きます（例 17-6）。パッケージのディレクトリ構造を、Oracle 9*i*FS ソフトウェアのホーム・ディレクトリ内で `custom_classes` ディレクトリの真下に作成することをお薦めします（例 17-7）。`custom_classes` ディレクトリは、Oracle 9*i*FS により設定される環境変数 CLASSPATH にすでに含まれています。
2. 環境変数 CLASSPATH が、Oracle 9*i*FS の Java Runtime Engine からクラスにアクセスできるように構成されているかどうかを確認します。パッケージのディレクトリを `custom_classes` ディレクトリの真下に作成した場合、環境変数 CLASSPATH を変更する必要はありません（例 17-8）。
3. コンテンツ・タイプの ClassObject の BeanClassPath 属性を、カスタム・クラスの完全修飾クラス名を参照するように設定します（例 17-9）。

例 17-6 Java クラスのコンパイル

```
javac MyCompany.MyApp.beans.Image.java
```

例 17-7 Bean 側 Java クラスのディレクトリ

```
$ORACLE_HOME/9ifs/custom_classes/MyCompany/MyApp/beans/Image.class
```

例 17-8 XML を使用した新規 ClassObject の BeanClassPath の設定

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <NAME>Image</NAME>
    <SUPERCLASS RefType = "Name">Document</SUPERCLASS>
    <BEANCLASSPATH>MyCompany.MyApp.beans.Image</BEANCLASSPATH>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>Width</NAME>
            <DATATYPE>Integer</DATATYPE>
        </ATTRIBUTE>
        <ATTRIBUTE>
            <NAME>Height</NAME>
            <DATATYPE>Integer</DATATYPE>
        </ATTRIBUTE>
        <ATTRIBUTE>
            <NAME>Artists</NAME>
            <DATATYPE>DirectoryObject</DATATYPE>
            <CLASSDOMAIN RefType = "Name">DirectoryUser...</CLASSDOMAIN>
       </ATTRIBUTE>
    </ATTRIBUTES>
</CLASSOBJECT>
```

例 17-9 XML を使用した ClassObject の BeanClassPath の更新

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <UPDATE RefType = "Name">Image</UPDATE>
    <BEANCLASSPATH>MyCompany.MyApp.beans.Image</BEANCLASSPATH>
</CLASSOBJECT>
```

注意： ClassObject の作成手順と更新手順は、第 5 章「コンテンツ・タイプと属性の拡張」を参照してください。

テスト

カスタム Java クラスをテストする手順は、次のとおりです。

1. クラスのインスタンスを構成して拡張メソッドをコールする、単純な Java アプリケーションを記述します（例 17-10）。
2. Java クラスをコンパイルし、そのパッケージに該当するディレクトリに置きます（例 17-11 および例 17-12）。
3. Java クラスを実行します（例 17-13）。

例 17-10 Bean 側 Java クラスのテストの記述

```
// Copyright (c) 2001 Oracle Corporation
package MyCompany.MyApp.tests;

import oracle.ifs.beans.LibraryObject;
import oracle.ifs.beans.LibraryService;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.Relationship;
import oracle.ifs.beans.Selector;
import oracle.ifs.beans.PublicObject;

import oracle.ifs.common.AttributeValue;
import oracle.ifs.common.CleartextCredential;
import oracle.ifs.common.IfsException;

import MyCompany.MyApp.beans.Image;

public class TestBeanSideClass extends Object {

  public static void main(String[] args)
  {
     TestBeanSideClass TestBeanSideClass = new TestBeanSideClass(args);
  }

  public TestBeanSideClass(String[] args)
  {
    try
    {
      String userName = args[0];
      String userPassword = args[1];
      String serviceName = args[2];
      String schemaPassword = args[3];

      //Connect to Oracle 9iFS.
      LibraryService service =
             LibraryService.startService(serviceName, schemaPassword);
      CleartextCredential cred =
             new CleartextCredential(userName, userPassword);
      LibrarySession session = service.connect(cred, null);

      try
      {
        int i, icount, j, jcount;
        Image image;
        PublicObject[] images, books;
        PublicObject book;
        Selector s;
```

```
    //Obtain the Image instance
    s = new Selector(session);
    s.setSearchClassname("IMAGE");

    //Execute the query and retrieve each Image.
    images = (PublicObject[])s.getItems();

    jcount = (images == null) ? 0 : images.length;
    System.out.println("Images : " + jcount);

    for (j = 0; j < jcount; j++)
    {

      image = (Image) images[j];
      System.out.println("  Image : " + image.getName());
      System.out.println("    Width : " + image.getImageWidth(session));
      System.out.println("    Height : " + image.getImageHeight(session));

      System.out.println("   Setting Image Height and Width");

      image.setImageWidth(600);
      image.setImageHeight(800);

      System.out.println("    Width : " + image.getImageWidth(session));
      System.out.println("    Height : " + image.getImageHeight(session));

      books = image.getBooks(session);

      icount = (books == null) ? 0 : books.length;
      System.out.println("  Books : " + icount);

      for (i = 0; i < icount; i++)
      {
        book = books[i];
        System.out.println("  " + book.getName());
      }
    }
  }
  catch (IfsException e)
  {
    System.out.println("An error occured.");
    System.out.println("=====================");
    System.out.println(e.toString());
    e.printStackTrace();
  }
```

```
      //Disconnect.
      session.disconnect();
    }
    catch (IfsException e)
    {
      System.out.println("Unable to connect to 9iFS.");
      System.out.println("======================");
      System.out.println(e.toString());
      e.printStackTrace();
    }
    catch (Exception e)
    {
      System.out.println("Unable to connect to 9iFS.");
      System.out.println("Ensure that you have supplied the correct");
      System.out.println("connection information in the following arguments: ");
      System.out.println("  args[0] = User Name");
      System.out.println("  args[1] = User Password");
      System.out.println("  args[2] = Service Name");
      System.out.println("  args[3] = Schema Password");
      System.out.println("Example:  ");
      System.out.println("  java " );
      System.out.println("    MyCompany.MyApp.tests.TestBeanSideClass");
      System.out.println("    system manager9ifs IfsDefault ifssys");
      System.out.println("======================");
     System.out.println(e.toString());
     e.printStackTrace();
    }
  }
}
```

例 17-11 テスト用 Java クラスのコンパイル

```
javac TestBeanSideClass.java
```

例 17-12 テスト用 Java クラスのディレクトリ

```
$ORACLE_HOME/9ifs/custom_classes/MyCompany/MyApp/tests/TestBeanSideClass.class
```

例 17-13 テスト用 Java クラスの実行

```
java MyCompany.MyApp.tests.TestBeanSideClass system manager IfsDefault ifssys
```

サーバー・オーバーライドの作成

コンテンツ・タイプの動作は、サーバー側 Java クラスで実装されます。サーバー側クラスは、挿入、更新および削除などの操作について、Oracle 9*i*FS による実行方法を決定するコードを持ちます。

サーバー側オーバーライドを実装すると、これらの操作について Oracle 9*i*FS による実行方法をカスタマイズできます。オーバーライドにより、Oracle 9*i*FS サーバーでの標準的な処理フローに事前定義済みの特定箇所で割り込むことができます。この箇所で、操作が発生する前または後に実行する必要のあるカスタム処理タスクを実装できます。

たとえば、特定のプロジェクトに関連する新規ドキュメントに、そのプロジェクトのメンバーが確実にアクセスできるようにするビジネス・ルールを実装するとします。ドキュメントの Project 属性の値をチェックして、そのプロジェクトに対応付けられた DirectoryGroup にドキュメントの AccessControlList にある Discover アクセス権を自動的に付与するように、カスタム処理タスクを実装できます。このタスクを事前挿入のオーバーライドとして実装すると、ドキュメントが Oracle 9*i*FS に挿入される前に、必ず処理タスクを発生させることができます。

エージェントではタスクが非同期で実行されるのに対して、オーバーライドは操作と同期で実行されます。つまり、カスタム処理タスクは操作と同じトランザクション内で実行されます。どちらかが失敗すると、カスタム・タスクと操作によって行われた変更はロールバックされます。

オーバーライドの実装方法

Oracle 9*i*FS Java API には、カスタム・オーバーライドの実装に使用できるように、サーバー側 Bean にフックが用意されています。このフックは、S_LibraryObject および S_Relationship クラスに置かれた操作前メソッドと操作後メソッドで構成されています。各メソッドは、対応する操作の実行前および実行後に Oracle 9*i*FS によりコールされます。インストール時には、この操作前メソッドと操作後メソッドは空です。

Oracle 9*i*FS によるこれらの操作の実行方法をカスタマイズするには、操作前メソッドと操作後メソッドをオーバーライドするメソッドを含む、カスタムのサーバー側 Java クラスを作成します。

注意： Oracle 9*i*FS の機能性を実装するコードはサーバー側 Bean に実装されるため、Oracle 9*i*FS の操作と同期で実行する必要のあるメソッドも、サーバー側 Bean に置く必要があります。

オーバーライド・メソッドでは、サーバー側 Java クラスの他のカスタム・メソッドをコールできます。たとえば、サーバー側 Java クラスを拡張し、カスタム属性を取得および設定するためのコンビニエンス・メソッドや、リポジトリ内の情報を操作する処理メソッドを含めることができます。

注意： 拡張メソッドをコールできるのは、操作前または操作後のオーバーライド・メソッドのみです。クライアント・アプリケーションからは、サーバー側 Java クラスの拡張メソッドをコールできません。

表 17-3 に、オーバーライド可能な操作前メソッドと操作後メソッドを持つすべての操作を示します。

表 17-3 操作のオーバーライド・メソッド

操作	オーバーライド・メソッド	使用方法
Insert	S_LibraryObject の extendedPreInsert()	Oracle 9*i*FS にオブジェクトが挿入される前にタスクを実行するために使用します。 たとえば、Oracle 9*i*FS に挿入される前に、ある属性の値の妥当性を別の属性の値に基づいてチェックします。
Insert	S_LibraryObject の extendedPostInsert()	Oracle 9*i*FS にオブジェクトが挿入された後にタスクを実行するために使用します。 たとえば、ドキュメントを、その Project 属性の値に対応するプロジェクト・フォルダに自動的に挿入します。
Update	S_LibraryObject の extendedPreUpdate()	Oracle 9*i*FS 内でオブジェクトが更新される前にタスクを実行するために使用します。 たとえば、ドキュメントの PublicationStatus 属性を Published に設定する前に、ReviewStatus 属性が Approved に設定されているかどうかを確認します。
Update	S_LibraryObject の extendedPostUpdate()	Oracle 9*i*FS 内でオブジェクトが更新された後にタスクを実行するために使用します。 たとえば、ドキュメントの PublicationStatus 属性が Published に設定されると、ACL 属性を Published の SystemAccessControlList で自動的に設定します。

表 17-3 操作のオーバーライド・メソッド（続く）

操作	オーバーライド・メソッド	使用方法
Free	S_LibraryObject の extendedPreFree()	Oracle 9*i*FS からオブジェクトが削除される前にタスクを実行するために使用します。 たとえば、Oracle 9*i*FS からドキュメントが削除される前に、そのコピーをアーカイブします。
Free	S_LibraryObject の extendedPostFree()	Oracle 9*i*FS からオブジェクトが削除された後にタスクを実行するために使用します。 たとえば、複合ドキュメントが Oracle 9*i*FS から削除された後に、それに関連するすべてのドキュメントを連鎖的に削除します。
AddRelationship	S_Relationship の extendedPreAddRelationship()	オブジェクトがフォルダに追加される前にタスクを実行するために使用します。 たとえば、属性の比較に基づいて、オブジェクトが適切なフォルダに追加されることを確認します。
AddRelationship	S_Relationship の extendedPostAddRelationship()	オブジェクトがフォルダに追加された後にタスクを実行するために使用します。 たとえば、オブジェクトにフォルダの ACL を適用します。
RemoveRelationship	S_Relationship の extendedPreRemoveRelationship()	オブジェクトがフォルダから削除される前にタスクを実行するために使用します。 たとえば、ドキュメントが別のフォルダにあり、パス経由で引き続きアクセスできることを確認します。
RemoveRelationship	S_Relationship の extendedPostRemoveRelationship()	オブジェクトがフォルダから削除された後にタスクを実行するために使用します。 たとえば、削除する関係がカスタム Reference_Relationships の場合に、参照として関連付けられているオブジェクトの数を示すオブジェクトの属性を更新します。

操作前メソッドと操作後メソッド（つまり extendedPreFree()）をオーバーライドする方が、Oracle 9*i*FS の操作を実装するメソッド（つまり Free()）をオーバーライドするよりも簡単かつ安全です。Oracle 9*i*FS の操作を実装するメソッドをオーバーライドするには、Oracle 9*i*FS のソース・コードを再生成する必要があります。Insert、Update および Delete など、Oracle 9*i*FS での操作のコードは、いずれも複雑であり、この方法でオーバーライドするのは危険です。1 つでも誤りがあるとデータベースが破損する恐れがあります。操作前メソッドと操作後メソッドを使用するとカスタムの機能性を実装でき、操作の実行とデータベース内のデータ整合性のメンテナンスに伴う複雑さは Oracle 9*i*FS で処理されます。このような理由から、Oracle 9*i*FS では操作前メソッドと操作後メソッドのオーバーライドのみがサポートされます。

オーバーライドの使用例

オーバーライドを記述する手順は、次のとおりです。

1. サーバー側クラスの宣言
2. コンストラクタの作成
3. 操作前および操作後のオーバーライド・メソッドの実装

サーバー側クラスの宣言

最初に、サーバー内でカスタム・クラスを表すサーバー側 Java クラス S_Image を作成します。S_Image は S_TieDocument を拡張します。このクラスを MyCompany.MyApp.server などの新規カスタム・パッケージに入れます（このクラスを oracle.ifs.server パッケージに追加しないでください）。

例 17-14 クラスの宣言

```
public class S_Image extends S_TieDocument
```

コンストラクタの作成

各サーバー側 Java クラスでは、次の 2 つのコンストラクタを実装する必要があります。

- データベースに現在存在するオブジェクトに使用するコンストラクタ。
- データベース内でまだ作成していないオブジェクトに使用するコンストラクタ。

表 17-4 に、コンストラクタのパラメータを示します。

表 17-4 コンストラクタのパラメータ

パラメータ	データ型	説明
session	S_LibrarySession	現行の LibrarySession。
data	S_LibraryObjectData	データ・コンポーネント。
classID	LONG	作成中のプロセスに含まれるオブジェクトのクラス ID。

例 17-15 コンストラクタの作成

```
public S_Image(S_LibrarySession session, S_LibraryObjectData data)
throws IfsException
{
  super(session, data);
}

public S_Image(S_LibrarySession session, java.lang.Long classID)
throws IfsException
{
  super(session, classID);
}
```

操作前および操作後のオーバーライド・メソッドの実装

操作前メソッドと操作後メソッドをオーバーライドするメソッドを作成します。

たとえば、システム設定属性を設定する extendedPreInsert() メソッドをオーバーライドするとします。システム設定属性は、クライアント・アプリケーションでは設定できません。そのかわり、値はサーバー側に実装されているビジネス・ルールにより決定されます。

また、ある属性を別の属性の値に基づいて自動的に設定するように、extendedPreInsert() メソッドをオーバーライドするとします。属性の値は別の属性値に基づいているため、ValueDefault を使用して属性を自動的に設定するのみでは不十分です。サーバー側で属性の値を決定するビジネス・ルールを、オーバーライドとして実装できます。

表 17-5 に、このメソッドのパラメータを示します。

表 17-5 メソッドのパラメータ

パラメータ	データ型	説明
opState	OperationState	システムで操作の現在の状態を追跡するために使用されます。

表 17-5 メソッドのパラメータ（続く）

パラメータ	データ型	説明
def	S_LibraryObjectDefinition	システム属性で更新される現行のオブジェクト定義です。

例 17-16 オーバーライド・メソッドの実装

システム設定属性のカスタム妥当性チェックを実行する extendedPreInsert() オーバーライドの例を示します。このコード例には太字で表記されているプレースホルダがあり、アプリケーション用の適切なコードに置換する必要があるため注意してください。

```
// Override Pre Insert Operation
  public void extendedPreInsert(OperationState opState,
                                S_LibraryObjectDefinition def)
  throws IfsException
  {
    // ALWAYS call super
    super.extendedPreInsert(opState, def);

    // get our session
    S_LibrarySession session = getSession();

    // get the approver attribute so we can check it
    AttributeValue av1 = def.getAttribute("APPROVER");
    S_DirectoryUser approver = (av1 == null) ? null :
       (S_DirectoryUser) av1.getDirectoryObject(session);

    //validate the approver
    boolean verification = verifyApprover(approver);

    if (! verification)
    {
      // this will rollback the operation
      throw new IfsException( <your custom error code>, this );
    }

    // otherwise, set the APPROVED attribute.
    // the APPROVED bit is not settable or updateable
    // by users, but the server can set it thusly
    AttributeValue av2 = AttributeValue.newAttributeValue(true);
    def.setSystemSetAttribute("APPROVED", av2);
  }

  public boolean verifyApprover(S_DirectoryUser approver)
  {
```

```
    boolean verification = false;
    <your validation code>
    return verification;
  }
}
```

このメソッド開始後の最初のコールは、super にする必要があります。このコールは、スーパークラス（S_Document など）について同じメソッドで実装された処理を実装します。その後、必要なカスタム処理ロジックを実装します。

オーバーライド・メソッドでは、このサーバー側 Java クラスまたは他のサーバー側 Java クラスの拡張メソッドをコールできます。たとえば、コンテンツ・タイプのカスタム属性を取得および設定するコンビニエンス・メソッドを実装したり、リポジトリ内の情報を操作したりデータを導出する処理メソッドを実装できます。

詳細例

次のサンプル・コードに、サーバー側オーバーライドの詳細な操作例を示します。この例は、.gif ファイルをカスタム・コンテンツ・タイプ Image のインスタンスとして自動的に格納するように、Oracle 9*i*FS を拡張しています。このコンテンツ・タイプは、カスタム属性 Artists を別の属性の値に基づいて自動的に設定するサーバー側オーバーライドを持ちます。

1. 属性 Width、Height および Artists を指定してコンテンツ・タイプ Image を作成します（例 17-17）。
2. .gif ファイルがインポート時に自動的に Image として格納されるように、.gif ファイル拡張子を Image コンテンツ・タイプに登録します（例 17-18）。
3. Image コンテンツ・タイプ用に、Artists 属性を自動的に設定する `extendedPreInsert()` オーバーライドを含むサーバー側 Java クラスを作成します（例 17-19）。

例 17-17 XML を使用した Image コンテンツ・タイプの作成

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <NAME>Image</NAME>
    <SUPERCLASS RefType = "Name">Document</SUPERCLASS>
    <ATTRIBUTES>
        <ATTRIBUTE>
            <NAME>Width</NAME>
            <DATATYPE>Integer</DATATYPE>
        </ATTRIBUTE>
        <ATTRIBUTE>
            <NAME>Height</NAME>
            <DATATYPE>Integer</DATATYPE>
        </ATTRIBUTE>
        <ATTRIBUTE>
```

```
            <NAME>Artists</NAME>
            <DATATYPE>DirectoryObject</DATATYPE>
            <CLASSDOMAIN RefType = "Name">DirectoryUser...</CLASSDOMAIN>
        </ATTRIBUTE>
    </ATTRIBUTES>
</CLASSOBJECT>
```

例 17-18 XML を使用した Image コンテンツ・タイプ用のファイル拡張子の登録

```
<?xml version='1.0' standalone='yes'?>
<OBJECTLIST>
 <PROPERTYBUNDLE>
  <UPDATE RefType='ValueDefault'>ParserLookupByFileExtension</UPDATE>
   <PROPERTIES>
    <PROPERTY Action = 'add'>
     <NAME> gif </NAME>
      <VALUE DataType = 'String'>
         oracle.ifs.beans.parsers.ClassSelectionParser
      </VALUE>
   </PROPERTY>
  </PROPERTIES>
 </PROPERTYBUNDLE>

 <PROPERTYBUNDLE>
  <UPDATE RefType='ValueDefault'>
     IFS.PARSER.ObjectTypeLookupByFileExtension
  </UPDATE>
   <PROPERTIES>
    <PROPERTY Action = 'add'>
     <NAME>gif</NAME>
     <VALUE DataType='String'>Image</VALUE>
    </PROPERTY>
   </PROPERTIES>
  </PROPERTYBUNDLE>
</OBJECTLIST>
```

例 17-19 Image コンテンツ・タイプ用のサーバー側 Java クラスの作成

```
package MyCompany.MyApp.server;

import oracle.ifs.common.AttributeValue;
import oracle.ifs.common.IfsException;
import oracle.ifs.server.S_DirectoryObject;
import oracle.ifs.server.S_LibraryObjectData;
import oracle.ifs.server.S_LibraryObjectDefinition;
import oracle.ifs.server.S_LibrarySession;
import oracle.ifs.server.OperationState;
```

```
import oracle.ifs.server.S_TieDocument;

public class S_Image extends S_TieDocument
{
  // constructors
  public S_Image(S_LibrarySession session, S_LibraryObjectData data)
  throws IfsException
  {
    super(session, data);
  }

  public S_Image(S_LibrarySession session, Long classId)
  throws IfsException
  {
    super(session, classId);
  }

  // Overrides
  public void extendedPreInsert(OperationState opState,
                                S_LibraryObjectDefinition def)
  throws IfsException
  {
    // Always call super
    super.extendedPreInsert(opState, def);

    // Get the LibrarySession
    S_LibrarySession session = getSession();

    // Get the related objects
    S_DirectoryObject owner;
    AttributeValue av1, av2;

    av1 = def.getAttribute("OWNER");
    owner = (S_DirectoryObject) av1.getDirectoryObject(session);

    av2 = def.getAttribute("ARTISTS");

    if (av2 == null)
    {
      S_DirectoryObject[] artists = {owner};
      def.setUserSetAttribute("ARTISTS",
                      AttributeValue.newAttributeValue(artists));
    }
  }
}
```

オーバーライド・クラスの配置

コンテンツ・タイプのオーバーライドを配置する手順は、次のとおりです。

1. Oracle 9*i*FS サーバーのホスト・マシン上でサーバー側 Java クラスをコンパイルし、そのクラスのパッケージに対応するディレクトリに置きます（例 17-20）。パッケージのディレクトリ構造を、Oracle 9*i*FS ソフトウェアのホーム・ディレクトリ内で `custom_classes` ディレクトリの真下に作成してください（例 17-21）。`custom_classes` ディレクトリは、Oracle 9*i*FS により設定される環境変数 CLASSPATH にすでに含まれています。
2. 環境変数 CLASSPATH が、Oracle 9*i*FS の Java Runtime Engine からクラスにアクセスできるように構成されているかどうかを確認します。パッケージのディレクトリを `custom_classes` ディレクトリの真下に作成した場合、環境変数 CLASSPATH を変更する必要はありません（例 17-22）。
3. コンテンツ・タイプの ClassObject の ServerClassPath 属性を、カスタム・クラスの完全修飾クラス名を参照するように設定します。

注意： ClassObject の作成手順と更新手順は、第 5 章「コンテンツ・タイプと属性の拡張」を参照してください。

例 17-20 Java クラスのコンパイル

```
javac MyCompany.MyApp.server.S_Image.java
```

例 17-21 サーバー側 Java クラスのディレクトリ

```
$ORACLE_HOME/9ifs/custom_classes/MyCompany/MyApp/server/S_Image.class
```

例 17-22 XML を使用した ClassObject の ServerClassPath の更新

```
<?xml version="1.0" standalone="yes"?>
<CLASSOBJECT>
    <UPDATE RefType = "Name">Image</UPDATE>
    <SERVERCLASSPATH>MyCompany.MyApp.server.S_Image</SERVERCLASSPATH>
</CLASSOBJECT>
```

テスト

サーバー側オーバーライドをテストするには、コンテンツ・タイプのインスタンスを構成してオーバーライドされたメソッドをコールするように、サンプル Java アプリケーションを記述します。操作結果をチェックして、操作前と操作後のタスクが実行されたかどうかを確認します。

例 17-23 Image コンテンツ・タイプのサーバー・オーバーライドのテスト

Image コンテンツ・タイプのサーバー側オーバーライドを実装する「詳細例」をテストするには、新規 .gif ファイルを Oracle 9*i*FS にインポートするのみで済みます。.gif ファイル拡張子は Image コンテンツ・タイプ用に登録されているため、Oracle 9*i*FS ではイメージが Image のインスタンスとして自動的に格納されます。挿入時に、Oracle 9*i*FS は自動的に `extendedPreInsert()` オーバーライドをコールし、Artists 属性の設定に関するビジネス・ルールを実装します。

Tie クラスの置換

Oracle 9*i*FS にデフォルトで用意されているコンテンツ・タイプのカスタム動作を実装し、これらの動作をすべての派生コンテンツ・タイプに継承させる場合は、対応する Tie クラスを置換します。Tie クラスは、デフォルトのコンテンツ・タイプの Bean 側およびサーバー側 Java クラスを直接変更せずに、クラス階層内の必要な箇所にカスタム・コードを挿入するためのプレースホルダです。

Oracle 9*i*FS Java API では、クラスごとに対応する Tie クラスがあります。各派生クラスは、クラス自体ではなく Tie クラスを拡張します。たとえば、カスタム・クラス Image は、Document ではなく TieDocument を拡張します。

```
java.lang.Object
  +--oracle.ifs.beans.LibraryObject
    +--oracle.ifs.beans.TieLibraryObject
      +--oracle.ifs.beans.PublicObject
        +--oracle.ifs.beans.TiePublicObject
          +--oracle.ifs.beans.Document
            +--oracle.ifs.beans.TieDocument
          +--oracle.ifs.beans.Folder
            +--oracle.ifs.beans.TieFolder
```

Tie クラスを使用すると、階層内の必要なレベルに連結することで、Oracle 9*i*FS のクラスのデフォルト動作を変更できます。Tie クラスは Oracle 9*i*FS 階層内の位置を保持するため、Oracle 9*i*FS の既存クラスの動作をカスタマイズし、継承構造の新規動作部分にすることができます。Tie クラスは、拡張メソッドを含むカスタム Tie クラスで置換できます。Tie クラスは実質的には空のため、拡張対象となる Oracle 9*i*FS クラスのデフォルト動作を実装するコード全体を再生成せずに置換できます。派生コンテンツ・タイプはすべて Oracle 9*i*FS クラス自体ではなく Tie クラスを拡張するため、カスタム・メソッドを継承します。

たとえば、ドキュメント、フォルダ、ユーザーおよびグループなど、Oracle 9*i*FS に格納されるすべてのパブリック情報について、カスタム・アーカイブ機能を実装できます。これはすべてのパブリック情報に適用されるため、Document、Folder、User および Group コンテンツ・タイプを拡張して機能を複数回実装するのではなく、動作を PublicObject コンテンツ・タイプに実装するのが最も適切な方法です。TiePublicObject クラスは、PublicObject を拡張してメソッド `archive()` および `restore()` を含めるカスタム TiePublicObject クラスで置換できます。Document、Folder、User および Group クラスはいずれも TiePublicObject を

拡張するため、PublicObject のメソッドと TiePublicObject で実装したカスタム・メソッドを自動的に継承します。

Bean 側 Tie クラスの置換

Bean 側 Tie クラスを置換する手順は、次のとおりです。

1. Bean 側 Tie クラスの宣言
2. コンストラクタの作成
3. カスタム・メソッドの作成

Bean 側 Tie クラスの宣言

最初に、クラス制限を含めます。Bean 側 Tie クラスは、カスタム動作（PublicObject など）を持つ必要のある Oracle 9*i*FS のコンテンツ・タイプの Bean 側 Tie クラスを拡張する必要があります。このクラスを `oracle.ifs.beans` パッケージに含めます。

例 17-24 クラスの宣言

```
public class TiePublicObject extends PublicObject
```

コンストラクタの作成

Bean 側 Tie クラスでコンストラクタを実装し、そのコンストラクタでスーパークラスのコンストラクタをコールする必要があります。表 17-6「コンストラクタのパラメータ」に、コンストラクタのパラメータを示します。

表 17-6 コンストラクタのパラメータ

パラメータ	データ型	説明
`session`	S_LibrarySession	現行の LibrarySession。
ID	java.lang.Long	コンテンツ・タイプの既存インスタンスの ID。
`classID`	LONG	作成中のプロセスに含まれるコンテンツ・タイプのインスタンスのクラス ID。
`data`	S_LibraryObjectData	コンテンツ・タイプの既存インスタンスのデータ・コンポーネント。

例 17-25 コンストラクタ

```
public TiePublicObject(LibrarySession session,
                       Java.lang.Long id,
                       Java.lang.Long classID,
                       S_LibraryObjectData data, )
throws IfsException
{
  super(session, id, classID, data);
}
```

カスタム・メソッドの作成

クラスを宣言してコンストラクタを組み込んだ後に、メソッドを追加し、Oracle 9*i*FS のコンテンツ・タイプとその派生コンテンツ・タイプのすべてのインスタンスのカスタム動作を実装できます。

例 17-26 カスタム・メソッドの作成

```
public LibraryObject[] getAllRelationships(LibrarySession session,
                                           String relationshipClass)
 throws IfsException
 {
    Long id = getId();
    LibraryObject[] relObjects, rightObjects, leftObjects;
    rightObjects = getRightwardRelationships(relationshipClass);
    leftwardObjects = getLeftwardRelationships(relationshipClass);

    int i;
    int rcount = (rightwardObjects == null) ? 0 : rightwardObjects.length;
    for (i = 0; i < rcount; i++)
    {
        relObjects[i] = rightwardObjects[i];
    }

    int lcount = (leftwardObjects == null) ? 0 : leftwardObjects.length;
    for (i = 0; i < lcount; i++)
    {
        relObjects[rcount + i] = rightwardObjects[i];
    }

    return relObjects;
 }
```

サーバー側 Tie クラスの置換

サーバー側 Tie クラスを置換する手順は、次のとおりです。

1. サーバー側 Tie クラスの宣言
2. コンストラクタの作成
3. オーバーライド・メソッドの作成

サーバー側 Tie クラスの宣言

最初にサーバー側 Tie クラスを宣言します。サーバー側 Tie クラスは、カスタム・メソッド（S_Relationship など）を持つ必要のあるコンテンツ・タイプのサーバー側クラスを拡張する必要があります。そのクラスを `oracle.ifs.server` パッケージに入れます。

例 17-27 クラスの宣言

```
public class S_TieRelationship extends S_Relationship
```

コンストラクタの作成

各サーバー側 Tie クラスでは、次の 2 つのコンストラクタを実装する必要があります。

- データベースに現在存在するオブジェクトに使用するコンストラクタ： このコンストラクタには、アプリケーションが Oracle 9*i*FS に対して操作を実行するために保持する S_LibrarySession のパラメータと、既存のコンテンツ・タイプのインスタンスに関するデータを保持する S_LibraryObjectData オブジェクトのパラメータが必要です。
- データベース内でまだ作成していないオブジェクトに使用するコンストラクタ： このコンストラクタには、S_LibrarySession とインスタンス化するコンテンツ・タイプの識別子のパラメータが必要です。コンテンツ・タイプを表す ClassObject のインスタンスは、そのコンテンツ・タイプを一意に識別するための属性 ID を持ちます。

表 17-7「コンストラクタのパラメータ」に、コンストラクタのパラメータを示します。

表 17-7 コンストラクタのパラメータ

パラメータ	データ型	説明
`session`	S_LibrarySession	現行の LibrarySession。
`data`	S_LibraryObjectData	コンテンツ・タイプの既存インスタンスのデータ・コンポーネント。
`classID`	LONG	作成中のプロセスに含まれるコンテンツ・タイプのインスタンスのクラス ID。

例 17-28 コンストラクタの作成

```
public S_TieFolderRelationship(S_LibrarySession session, S_LibraryObjectData data)
throws IfsException
{
  super(session, data);
}

public S_TieFolderRelationship(S_LibrarySession session, java.lang.Long classID)
throws IfsException
{
  super(session, classID);
}
```

オーバーライド・メソッドの作成

Oracle 9*i*FS サーバー上で実行するカスタム動作を実装するオーバーライドを含めます。たとえば、`extendedPreAddRelationship()` メソッドをオーバーライドして、オブジェクトの ACL 属性を親フォルダの AccessControlList に自動的に設定するように、オーバーライドを実装できます。サーバー側 Tie クラスのサブクラスは、すべてこれらのメソッドを継承します。

注意： オーバーライドの実装手順は、「サーバー・オーバーライドの作成」を参照してください。

例 17-29 オーバーライド・メソッドの実装

```
public void extendedPostAddRelationship(S_LibraryObjectDefinition def,
                                        OperationState opState)
throws IfsException
{
    // Always call super
    super.extendedPostAddRelationship(opState, def);

    // Get the LibrarySession
    S_LibrarySession session = getSession();

    // Get the related objects
    AttributeValue av1, av2;
    S_PublicObject f, po;
    av1 = def.getAttribute("LEFTOBJECT");
    f = (S_PublicObject) av1.getPublicObject(session);
    av2 = def.getAttribute("RIGHTOBJECT");
    po = (S_PublicObject) av2.getPublicObject(session);
```

```
    // Apply the Folder's ACL to the item being added to the folder
    S_AccessControlList acl = f.getAcl();
    po.setAttribute("ACL",
                    AttributeValue.newAttributeValue(acl));
}
```

詳細例

次の例に、PublicObject コンテンツ・タイプの Bean 側およびサーバー側 Tie クラスの詳細表現を示します。

例 17-30 Bean 側 Tie Java クラスの置換

```
package oracle.ifs.beans;

import oracle.ifs.beans.LibraryObject;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.PublicObject;
import oracle.ifs.beans.Selector;

import oracle.ifs.common.IfsException;

import oracle.ifs.server.S_LibraryObjectData;

public class TiePublicObject
 extends PublicObject
{
 /**
  * Constructs a TiePublicObject.
  *
  * @param session  the session
  * @param id       the id
  * @param classId  the class id
  * @param data     the data
  *
  * @exception IfsException if the operation fails
  * @pub
  */
 protected TiePublicObject
 (
  LibrarySession session,
  Long id,
  Long classId,
  S_LibraryObjectData data
 ) throws IfsException
 {
  super(session, id, classId, data);
```

```
 }

 public LibraryObject[] getAllRelationships(LibrarySession session,
                                            String relationshipClass)
 throws IfsException
 {
    Long id = getId();
    LibraryObject[] relObjects = null, rightwardObjects, leftwardObjects;
    rightwardObjects = getRightwardRelationships(relationshipClass);
    leftwardObjects = getLeftwardRelationships(relationshipClass);

    int i;
    int rcount = (rightwardObjects == null) ? 0 : rightwardObjects.length;
    for (i = 0; i < rcount; i++)
    {
        relObjects[i] = rightwardObjects[i];
    }

    int lcount = (leftwardObjects == null) ? 0 : leftwardObjects.length;
    for (i = 0; i < lcount; i++)
    {
        relObjects[rcount + i] = rightwardObjects[i];
    }

    return relObjects;
  }
}
```

例 17-31 サーバー側 Tie Java クラスの置換

```
package oracle.ifs.server;

import oracle.ifs.common.IfsException;
import oracle.ifs.common.AttributeValue;

import oracle.ifs.server.S_AccessControlList;
import oracle.ifs.server.S_PublicObject;
import oracle.ifs.server.OperationState;
import oracle.ifs.server.S_FolderRelationship;
import oracle.ifs.server.S_LibraryObjectDefinition;

public class S_TieFolderRelationship
extends S_FolderRelationship
{
  /**
  * Constructs a TiePublicObject.
  *
```

```
* @param session  the session
* @param id       the id
* @param classId  the class id
* @param data     the data
*
* @exception IfsException if the operation fails
* @pub
*/

// Constructors

public S_TieFolderRelationship(S_LibrarySession session,
                               S_LibraryObjectData data)
throws IfsException
{
  super(session, data);
}

public S_TieFolderRelationship(S_LibrarySession session,
                               java.lang.Long classID)
throws IfsException
{
  super(session, classID);
}

// Overrides
public void extendedPostAddRelationship(OperationState opState,
                              S_LibraryObjectDefinition def)
throws IfsException
{
  // Always call super
  super.extendedPostInsert(opState, def);

  // Get the LibrarySession
  S_LibrarySession session = getSession();

  // Get the related objects
  AttributeValue av1, av2;
  S_PublicObject f, po;
  av1 = def.getAttribute("LEFTOBJECT");
  f = (S_PublicObject) av1.getPublicObject(session);
  av2 = def.getAttribute("RIGHTOBJECT");
  po = (S_PublicObject) av2.getPublicObject(session);

  // Apply the Folder's ACL to the item being added to the folder
  S_AccessControlList acl = f.getAcl();
  po.setAttribute("ACL",
```

```
                    AttributeValue.newAttributeValue(acl));
  }
}
```

カスタム Tie クラスの配置

カスタム Tie クラスを配置する手順は、次のとおりです。

1. カスタム Tie クラスをコンパイルします（例 17-32）。
2. カスタム Tie クラスを保持するディレクトリを作成します（`tie_classes` など）。このディレクトリは、Oracle 9*i*FS ソフトウェアのホーム・ディレクトリにある `custom_classes` ディレクトリに置くことをお薦めします（例 17-33）。カスタム Tie クラスのディレクトリ内で、Tie クラスのパッケージに対応するサブディレクトリ（`oracle/ifs/beans` および `oracle/ifs/server` など）を作成します。カスタム・クラスを該当するディレクトリに移動します。
3. Oracle 9*i*FS ホスト・マシン上で環境変数 CLASSPATH を変更し、カスタム Tie クラスのディレクトリを含めます。これらのディレクトリへのパスが、CLASSPATH 内で `repos.jar` ファイルへのパスより前にあることを確認してください。これにより、Java Virtual Machine（JavaVM）は `repos.jar` の Oracle 9*i*FS Tie クラスをロードするかわりに、カスタム Tie クラスをロードできます。
4. Oracle 9*i*FS を停止してから起動し、JRE の CLASSPATH をリフレッシュします（例 17-34）。

例 17-32 Java クラスのコンパイル

```
javac TiePublicObject.java
javac S_TieFolderRelationship.java
```

例 17-33 Solaris での Tie クラス用ディレクトリの作成

```
$ORACLE_HOME/9ifs/custom_classes/tie_classes/oracle/ifs/beans
$ORACLE_HOME/9ifs/custom_classes/tie_classes/oracle/ifs/server
```

例 17-34 Solaris での Oracle 9*i*FS の停止と起動

```
$ORACLE_HOME/9ifs/bin/ifsstopdomain
$ORACLE_HOME/9ifs/bin/ifsstartdomain
```

注意： Oracle 9*i*FS の停止および起動手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。

テスト

カスタム Tie クラスをテストする手順は、次のとおりです。

1. コンテンツ・タイプのインスタンスを構成し、そのインスタンスの拡張メソッドをコールする、単純な Java アプリケーションを記述します（例 17-35）。
2. Java クラスをコンパイルし（例 17-36）、そのパッケージに該当するディレクトリに置きます（例 17-37）。
3. Java クラスを実行します（例 17-38）。

例 17-35 Bean 側 Tie クラスのテストの記述

```
package MyCompany.MyApp.tests;

import oracle.ifs.beans.Folder;
import oracle.ifs.beans.LibraryObject;
import oracle.ifs.beans.LibraryService;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.Relationship;
import oracle.ifs.beans.TiePublicObject;

import oracle.ifs.common.CleartextCredential;
import oracle.ifs.common.IfsException;

public class TestTiePublicObject extends Object {


  public static void main(String[] args)
  {
     TestTiePublicObject TestTiePublicObject = new TestTiePublicObject(args);
  }

  public TestTiePublicObject(String[] args)
  {
    try
    {
      String userName = args[0];
      String userPassword = args[1];
      String serviceName = args[2];
      String schemaPassword = args[3];

      //Connect to Oracle 9iFS.
      LibraryService service =
             LibraryService.startService(serviceName, schemaPassword);
      CleartextCredential cred =
             new CleartextCredential(userName, userPassword);
```

```
  LibrarySession session = service.connect(cred, null);

  try
  {
      session.setAdministrationMode(true);

      Folder f = session.getPrimaryUserProfile().getHomeFolder();
      System.out.println("Mome Folder : " + f.getName());

      LibraryObject[] rels = f.getAllRelationships(session, "RELATIONSHIP");

      int i, icount;
      Relationship rel;

      icount = (rels == null) ? 0 : rels.length;
      System.out.println(" Relationships : " + icount);

      for (i = 0; i < icount; i++)
      {
         rel = (Relationship) rels[i];
         System.out.println("   " + rel.getLeftObject().getName() +
                            " : " + rel.getRightObject().getName());
      }
  }
  catch (IfsException e)
  {
    System.out.println("An error occured.");
    System.out.println("=====================");
    System.out.println(e.toString());
    e.printStackTrace();
  }

  //Disconnect.
  session.disconnect();
}
catch (IfsException e)
{
  System.out.println("Unable to connect to 9iFS.");
  System.out.println("=====================");
  System.out.println(e.toString());
  e.printStackTrace();
}
catch (Exception e)
{
  System.out.println("Unable to connect to 9iFS.");
  System.out.println("Ensure that you have supplied the correct");
  System.out.println("connection information in the following arguments: ");
```

```
      System.out.println("  args[0] = User Name");
      System.out.println("  args[1] = User Password");
      System.out.println("  args[2] = Service Name");
      System.out.println("  args[3] = Schema Password");
      System.out.println("Example:  ");
      System.out.println("  java " );
      System.out.println("    MyCompany.MyApp.tests.TestTiePublicObject");
      System.out.println("    system manager9ifs IfsDefault ifssys");
      System.out.println("======================");
     System.out.println(e.toString());
     e.printStackTrace();
    }
  }
}
```

例 17-36 サーバー側 Tie クラスのテストの記述

```
package MyCompany.MyApp.tests;

import oracle.ifs.beans.AccessControlList;
import oracle.ifs.beans.Folder;
import oracle.ifs.beans.FolderDefinition;
import oracle.ifs.beans.LibraryService;
import oracle.ifs.beans.LibrarySession;

import oracle.ifs.common.AttributeValue;
import oracle.ifs.common.CleartextCredential;
import oracle.ifs.common.Collection;
import oracle.ifs.common.IfsException;

public class TestServerTieClass extends Object {

  public static void main(String[] args)
  {
      TestServerTieClass TestServerTieClass = new TestServerTieClass(args);
  }

  public TestServerTieClass(String[] args)
  {
    try
    {
      String userName = args[0];
      String userPassword = args[1];
      String serviceName = args[2];
      String schemaPassword = args[3];

      //Connect to Oracle 9iFS.
```

```
LibraryService service =
       LibraryService.startService(serviceName, schemaPassword);
CleartextCredential cred =
       new CleartextCredential(userName, userPassword);
LibrarySession session = service.connect(cred, null);

try
{
   AccessControlList defACL, propagatedACL, privateACL, publicACL;
   Folder propagatedF, privateF, publicF;
   String propagatedFName, privateFName, publicFName;

   // Get the session's ID to generate unique folder names.
   Long sessionId = session.getId();

   // Create a FolderDefinition which will be used
   // to create three folders to test the Tie class.
   FolderDefinition fd = new FolderDefinition();

   // Create a new folder which will be orphaned and
   // therefore acquire the user's default ACL.
   fd.setAttributeByUpperCaseName("NAME",
      AttributeValue.newAttributeValue("PropACLFolder" + sessionId));
   propagatedF = (Folder) session.createPublicObject(fd);
   propagatedFName = propagatedF.getName();

   System.out.println(propagatedFName + " created.");

   defACL = propagatedF.getAcl();
   System.out.println("  The folder is currently orphaned");
   System.out.println("  so it has the default ACL : "
       + defACL.getName());

   // Get the Private and Public ACLs to be set explicitly
   // on two new folders.
   Collection systemACLs = session.getSystemAccessControlListCollection();
   privateACL = (AccessControlList) systemACLs.getItems("Private");
   publicACL = (AccessControlList) systemACLs.getItems("Public");

   // Create another folder which has the Private ACL.
   fd.setAttributeByUpperCaseName("NAME",
      AttributeValue.newAttributeValue("PrivFolder" + sessionId));
   fd.setAttributeByUpperCaseName("ACL",
      AttributeValue.newAttributeValue(privateACL));
   privateF = (Folder) session.createPublicObject(fd);
   privateFName = privateF.getName();
```

```
      System.out.println(privateFName + " created.");
      System.out.println(privateFName + "'s ACL is : "
             + privateF.getAcl().getName());

      // Add the original folder to the Private folder to automatically
      // apply the Private ACL with the server override on the Tie class.
      privateF.addItem(propagatedF);
      System.out.println("  Added " + propagatedFName + " to "
         + privateFName);

      propagatedACL = propagatedF.getAcl();
      System.out.println("  " + propagatedFName + "'s ACL is now : "
        + propagatedACL.getName());

      // Create another folder which has the Public ACL.
      fd.setAttributeByUpperCaseName("NAME",
         AttributeValue.newAttributeValue("PubFolder" + sessionId));
      fd.setAttributeByUpperCaseName("ACL",
         AttributeValue.newAttributeValue(publicACL));
      publicF = (Folder) session.createPublicObject(fd);
      publicFName = publicF.getName();

      System.out.println(publicFName + " created");
      System.out.println(publicFName + "'s ACL is : "
           + publicF.getAcl().getName());

      // Add the original folder to the Public folder to automatically
      // apply the Public ACL with the server override on the Tie class.
      publicF.addItem(propagatedF);
      System.out.println("  Added " + propagatedFName + " to "
         + publicFName);

      propagatedACL = propagatedF.getAcl();
      System.out.println("  " + propagatedFName + "'s ACL is now : "
        + propagatedACL.getName());
    }
    catch (IfsException e)
    {
      System.out.println("An error occured.");
      System.out.println("======================");
      System.out.println(e.toString());
      e.printStackTrace();
    }

    //Disconnect.
    session.disconnect();
  }
```

```
    catch (IfsException e)
    {
      System.out.println("Unable to connect to 9iFS.");
      System.out.println("=====================");
      System.out.println(e.toString());
      e.printStackTrace();
    }
    catch (Exception e)
    {
      System.out.println("Unable to connect to 9iFS.");
      System.out.println("Ensure that you have supplied the correct");
      System.out.println("connection information in the following arguments: ");
      System.out.println("  args[0] = User Name");
      System.out.println("  args[1] = User Password");
      System.out.println("  args[2] = Service Name");
      System.out.println("  args[3] = Schema Password");
      System.out.println("Example:  ");
      System.out.println("  java " );
      System.out.println("    MyCompany.MyApp.tests.TestServerTieClass");
      System.out.println("    system manager9ifs IfsDefault ifssys");
      System.out.println("=====================");
      System.out.println(e.toString());
      e.printStackTrace();
    }
  }
}
```

例 17-37 テスト用 Java クラスのコンパイル

```
javac TestTiePublicObject.java
javac TestServerTieClass.java
```

例 17-38 テスト用 Java クラスのディレクトリ

```
$ORACLE_HOME/9ifs/custom_classes/MyCompany/MyApp/tests/
```

例 17-39 テスト用 Java クラスの実行

```
java MyCompany.MyApp.tests.TestTiePublicObject system manager9ifs IfsDefault ifs
java MyCompany.MyApp.tests.TestServerTieClass system manager9ifs IfsDefault ifs
```

サンプル・コード

Oracle 9*i*FS は、この章の例について実行可能なサンプル・コード・ファイルとともにインストールされます。サンプル・コード・ファイルは、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/devdoc/customizingbehavior` ディレクトリにあります。表 17-8 に、サンプル・コード・ファイルとそれに対応する例を示します。

表 17-8 例とサンプル・コード・ファイル

例	サンプル・コード・ファイル
例 17-5「Bean 側 Java クラス Image の作成」	`Image.java`
例 17-8「XML を使用した新規 ClassObject の BeanClassPath の設定」	`DeployBeanSideClassforNewCT.xml`
例 17-9「XML を使用した ClassObject の BeanClassPath の更新」	`DeployBeanSideClassforExistingCT.xml`
例 17-10「Bean 側 Java クラスのテストの記述」	`TestBeanSideClass.java`
例 17-17「XML を使用した Image コンテンツ・タイプの作成」	`CreateContentType.xml`
例 17-18「XML を使用した Image コンテンツ・タイプ用のファイル拡張子の登録」	`RegisterContentTypeExtensions.xml`
例 17-19「Image コンテンツ・タイプ用のサーバー側 Java クラスの作成」	`S_Image.java`
例 17-22「XML を使用した ClassObject の ServerClassPath の更新」	`DeployServerSideClass.xml`
例 17-30「Bean 側 Tie Java クラスの置換」	`TiePublicObject.java`
例 17-31「サーバー側 Tie Java クラスの置換」	`S_TieFolderRelationship.java`
例 17-35「Bean 側 Tie クラスのテストの記述」	`TestTiePublicObject.java`
例 17-36「サーバー側 Tie クラスのテストの記述」	`TestServerTieClass.java`

Oracle 9*i*FS は、例のサンプル・コード・ファイルのみでなく、Java API の操作開始の出発点として役立つ高度なサンプル・コードとともにインストールされます。この API サンプル・コードは、`<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/api` ディレクトリにあります。

表 17-9 に、この章に関連する API サンプル・コードを示します。

表 17-9 API サンプル・コード

クラス	使用方法
`OverrideSample.java`	サーバー側オーバーライド extendedPreAddItem(ReportFolder) および extendedPreFree (Report) のデモを行います。JDBC 接続を使用して、異なるスキーマにある表を更新します。

18

プログラムによる電子メールの送受信

この章では、カスタム・アプリケーションで JavaMail API と Oracle 9*i*FS Java API を使用して、Oracle 9*i*FS で電子メールをプログラムにより送受信する方法について説明します。この章の内容は、次のとおりです。

- Oracle 9iFS での電子メールの機能
- メッセージ機能の実装
- サンプル・コード

Oracle 9*i*FS での電子メールの機能

Oracle 9*i*FS には、ユーザーが電子メールを送受信し、アプリケーション・プログラマが電子メール通知を生成してビジネス・プロセスを簡便化できるように、ビルトイン・メッセージ機能が用意されています。Oracle 9*i*FS のメッセージ機能のフレームワークは、次のコンポーネントで構成されています。

- Oracle 9*i*FS SMTP Server
- Oracle 9*i*FS IMAP Server
- Sendmail 8.9.3
- Java クラスのセット

メッセージのルーティング

Oracle 9*i*FS には、ユーザーがメッセージを送受信できるように、SMTP プロトコル・サーバーおよび IMAP プロトコル・サーバーという 2 つのプロトコル・サーバーが組み込まれています。

Oracle 9*i*FS SMTP Server は、サーバー側でメッセージの送受信に使用されます。SMTP サーバーは Sendmail をメッセージ転送エージェント（MTA）として使用します。これは、Oracle 9*i*FS で送受信される各電子メールが Sendmail を通過する必要があることを意味します。メッセージが Oracle 9*i*FS に送信されると、Sendmail 経由で Oracle 9*i*FS SMTP Server にルーティングされ、Oracle 9*i*FS に格納されます。Oracle 9*i*FS から送信されたメッセージは、Oracle 9*i*FS の送信ボックスに置かれてから、Oracle 9*i*FS Outbox Agent により Sendmail にルーティングされます。メッセージが他の Oracle 9*i*FS ユーザー宛の場合も、Outbox Agent は Sendmail を使用してメッセージをルーティングします。

Oracle 9*i*FS IMAP Server は、Oracle 9*i*FS 内のメッセージをメッセージ・クライアントからアクセスできるようにします。たとえば、ユーザーは IMAP 準拠の Netscape Communicator を使用して受信ボックスにあるメッセージを表示し、それを（Oracle 9*i*FS ドメイン内外にある）どのようなメール・アドレスにも送信できます。Oracle 9*i*FS IMAP Server は、IMAP サーバーと電子メール・クライアント間にセキュリティを提供するために SSL をサポートしています。

注意： Oracle 9*i*FS を Sendmail とともに動作するように構成する方法の詳細は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。Sendmail の構成は、http://www.sendmail.org/ のサイトを参照してください。

メッセージの格納

Oracle 9*i*FS ユーザー宛に送信されたメッセージは、リポジトリに格納されます。デフォルトでは、メッセージは Oracle 9*i*FS に Rfc822Message クラスのインスタンスとして格納されます。Oracle 9*i*FS は、メッセージの内容のうちメッセージ受信者間で共有される部分の単一コピーを格納することで、メッセージによる記憶領域の使用量を最小限に抑えます。

Oracle 9*i*FS には、電子メール・クライアントにより署名または暗号化されたメッセージを格納する機能があります。メッセージを暗号化したり署名する手順は、メッセージ・クライアントのユーザーズ・ガイドを参照してください。

Oracle 9*i*FS では、ユーザーはメッセージ・テキストに Oracle Text で索引を付けてメッセージを検索できます。メッセージ・テキストは索引付けされますが、このリリースの Oracle Text で索引付けされるのは、コンテンツ・タイプが TEXT のメッセージ添付ファイルのみです。

メッセージは、各受信者のメール・フォルダ内で参照されます。各ユーザーのメール・フォルダは、そのユーザーの PrimaryUserProfile で指定されます。

注意： ユーザーのメール・フォルダの構成手順は、『Oracle Internet File System セットアップおよび管理ガイド』を参照してください。ユーザーのメール・フォルダをプログラムで設定する方法は、第 15 章「セキュリティ」を参照してください。

メッセージ機能の実装

Oracle 9*i*FS では、標準的な電子メール・クライアントを使用してメッセージを送受信するのみでなく、メッセージ機能 API を使用してカスタム・アプリケーションにメッセージ機能を実装できます。

たとえば、Oracle 9*i*FS で特定のタスクが実行された時点で電子メール通知を自動的に生成するように、カスタム・アプリケーションを構築できます。このアプリケーションでは、パスワードに変更があったことを Oracle 9*i*FS ユーザーにいつでも通知できます。このアプリケーションは、サーバー側オーバーライドを使用して、新規パスワードを含むメッセージを生成し、ユーザーに送信します。Oracle 9*i*FS のトランザクション管理機能を使用することで、メッセージが正常に送信されない場合に、アプリケーションはアカウントの変更をロールバックできます。

この項の内容は、次のとおりです。

- メッセージ機能 API
- メッセージ操作へのトランザクション管理の適用
- メッセージの作成と送信
- メッセージ・フォルダの操作

メッセージ機能 API

メッセージ機能を実装するには、カスタム・アプリケーションで次の 2 つの API を使用します。

- JavaMail API
- Oracle 9*i*FS Java API

JavaMail API

Oracle 9*i*FS では、Oracle 9*i*FS リポジトリ内のメッセージとの相互作用に介入できるように、標準的な API である JavaMail API がサポートされます。JavaMail API の詳細は、このマニュアルでは説明しません。ただし、この項では、次のような一部の API クラスの使用例を示します。

- javax.mail.Folder
- javax.mail.Message
- javax.mail.session
- javax.mail.internet.MimeMessage
- javax.mail.internet.InternetAddress

注意： Sun 社の JavaMail API については、http://java.sun.com/products/javamail/index.html を参照してください。

Oracle 9*i*FS Java API

Oracle 9*i*FS Java API には、Oracle 9*i*FS に格納されているメッセージをアプリケーションで管理できるように、クラスのパッケージ `oracle.ifs.adk.mail` が含まれています。このパッケージの次の 2 つのクラスを使用して、JavaMail アプリケーションで Oracle 9*i*FS のメッセージ機能を活用できます。

- **IfsStore:** IfsStore クラスを使用すると、JavaMail アプリケーションから Oracle 9*i*FS に格納されているメッセージにアクセスできます。
- **IfsTransport:** IfsTransport クラスを使用すると、JavaMail アプリケーションから **Oracle 9*i*FS** 経由でメッセージを送信できます。

メッセージ操作へのトランザクション管理の適用

JavaMail API では、トランザクション管理はサポートされません。カスタム・アプリケーションでメッセージ操作をコミットおよびロールバックできるように、`oracle.ifs.adk.mail` パッケージの各クラスには、メッセージ操作にトランザクション管理を適用するためのメソッドが含まれています。

注意： Oracle 9*i*FS でトランザクション管理を使用する手順は、第 16 章「セッションおよびトランザクションの管理」を参照してください。

第 16 章「セッションおよびトランザクションの管理」で説明したように、LibrarySession クラスには、トランザクションの開始、強制終了および完了に使用するためのメソッドが用意されています。LibrarySession のトランザクション管理機能を Oracle 9*i*FS でのメッセージ操作に適用するには、IfsStore および IfsTransport オブジェクトの LibrarySession を設定します。LibrarySession を IfsStore または IfsTransport オブジェクトで設定するには、そのオブジェクトの `setLibrarySession()` メソッドをコールします。この 2 つのオブジェクトの LibrarySession を設定すると、LibrarySession にアクセスして、メッセージ・アプリケーション内のトランザクション・ブロックを定義し、管理できます。

注意： Oracle 9*i*FS のトランザクション管理の使用は、JavaMail API に準拠していません。したがって、アプリケーションでは、Oracle 9*i*FS 以外の JavaMail Store を処理できなくなります。Oracle 9*i*FS のトランザクション機能にアクセスする必要がない場合は、他のシステムとの互換性を保つために、JavaMail API を直接操作するように選択できます。

単一トランザクションを使用して、メッセージ・オブジェクトの操作とともに、他の Oracle 9*i*FS オブジェクトの操作を管理できます。そのためには、IfsStore および IfsTransport オブジェクトで設定する LibrarySession を、他の Oracle 9*i*FS オブジェクトの操作に使用する LibrarySession と一致させる必要があります。その後は、共有の LibrarySession を使用して、LibrarySession で実行される操作のトランザクションを開始、強制終了およびコミットできます。

Oracle 9*i*FS のトランザクション管理機能をメッセージ操作に適用するには、アプリケーションで次の手順を実行します。

1. トランザクション中の例外をキャッチする try 句で、トランザクションをラップします。
2. JavaMail とのセッションを確立します。
3. メッセージおよびその他の Oracle 9*i*FS オブジェクトの操作に使用する Oracle 9*i*FS との LibrarySession を確立します。

4. トランザクション管理に使用できるように、IfsStore または IfsTransport の LibrarySession を設定します。
5. LibrarySession を使用してトランザクションを開始します。メッセージや他のオブジェクトに対する操作をトランザクションに含めます。
6. トランザクションが正常に完了しない場合は、そのエラーを finally 句でキャッチしてトランザクションを強制終了します。トランザクションを catch 句で強制終了すると、エラーの原因がキャッチされない Throwable だった場合に問題が生じる可能性があります。finally 句で強制終了すると、トランザクションが完了するかロールバックされるかのどちらかになることが保証されます。

例 18-1 に、メッセージ操作のトランザクション管理に使用されるサンプル・コードから抜粋したコードを示します。

例 18-1 メッセージ操作へのトランザクション管理の使用

1. トランザクション中の例外をキャッチする try 句で、トランザクションをラップします。

```
try
{
```

2. JavaMail とのセッションを確立します。

```
Session javaMailSession = Session.getDefaultInstance(
  new java.util.Properties(), null);
```

3. メッセージおよびその他の Oracle 9*i*FS オブジェクトの操作に使用する Oracle 9*i*FS との LibrarySession を確立します。

```
LibraryService ifsService =
     LibraryService.startService(serviceName, schemaPassword);
CleartextCredential cred =
     new CleartextCredential(userName, userPassword);
LibrarySession ifsSession = service.connect(cred, null);
```

4. トランザクション管理に使用できるように、IfsStore または IfsTransaction の LibrarySession を設定します。

```
IfsStore store = (IfsStore) javaMailSession.getStore("ifs");
store.setLibrarySession(ifsSession);

IfsTransport transport = (IfsTransport) javaMailSession.getTransport("ifs1");
transport.setLibrarySession(ifsSession);
```

5. LibrarySession を使用してトランザクションを開始します。メッセージや他の Oracle 9*i*FS オブジェクトに対する操作をトランザクションに含めます。

```
Transaction transaction = ifsSession.beginTransaction();
```

```
// < operations against messages and other objects >
```

6. LibrarySession を使用して、すべての操作をアトミックにコミットし、トランザクションを完了します。

```
  ifsSession.completeTransaction(transaction);
  transaction = null;
}
catch (Throwable e)
{
  e.printStackTrace();
}
```

7. トランザクションが正常に完了しない場合は、そのエラーを finally 句でキャッチしてトランザクションを強制終了します。

```
finally
{
  try
  {
    if (transaction != null)
    {
      ifsSession.abortTransaction(transaction);
        transaction = null;
    }
    if (ifsSession != null)
    {
      ifsSession.disconnect();
    }
  }
  catch (IfsException e)
  {
     e.printStackTrace();
  }
}
```

メッセージの作成と送信

Oracle 9*i*FS でサポートされているメッセージ機能 API を使用し、メッセージをプログラムで作成して移送できます。そのためには、アプリケーションで次の手順を実行します。

1. javax.mail.Session を取得します。
2. Oracle 9*i*FS との LibrarySession を取得します。
3. javax.mail.Session から IfsTransport を作成します。IfsTransport で Oracle 9*i*FS の LibrarySession を設定します。
4. Oracle 9*i*FS の LibrarySession とのトランザクションを開始します。

5. javax.mail.internet.MimeMessage を構成します。
6. javax.mail.internet.InternetAddress を構成します。
7. MimeMessage のテキストを作成します。
8. MimeMessage の Subject を設定します。
9. MimeMessage の受信者を識別します。
10. IfsTransport 経由で MimeMessage を送信します。
11. トランザクションを完了します。
12. Oracle 9*i*FS の LibrarySession を切断します。トランザクションが正常に完了しない場合は、強制終了します。

例 18-2 に、メッセージ機能 API でメッセージの作成と送信に使用するサンプル・コードから抜粋したコードを示します。

例 18-2 メッセージの作成と送信

1. JavaMail とのセッションを確立します。

```
try
{

  Session javaMailSession = Session.getDefaultInstance(
    new java.util.Properties(), null);
```

2. メッセージおよびその他の Oracle 9*i*FS オブジェクトの操作に使用する Oracle 9*i*FS との LibrarySession を確立します。

```
LibraryService ifsService =
     LibraryService.startService(serviceName, schemaPassword);
CleartextCredential cred =
     new CleartextCredential(userName, userPassword);
LibrarySession ifsSession = service.connect(cred, null);
```

3. メッセージの送信に使用する IfsTransport オブジェクトを作成します。IfsTransaction の LibrarySession を設定します。

```
IfsTransport transport =
    (IfsTransport) javaMailSession.getTransport("ifs1");
transport.setLibrarySession(ifsSession);
```

4. LibrarySession を使用してトランザクションを開始します。

```
Transaction transaction = ifsSession.beginTransaction();
```

5. MimeMessage を構成します。

```
MimeMessage curMsg = new MimeMessage(session);
```

6. MimeMessage の受信者の InternetAddress を構成します。MimeMessage のアドレスを設定します。

```
String addressStr = parameterTable.getString("address");
InternetAddress address = new InternetAddress[1];
address[0] = new InternetAddress(addressStr);
```

7. MimeMessage のテキストを作成します。

```
String subjectStr = "Mailbox Scan Results";
String msgTxt = "";
msgTxt = "To: " + addressStr + "¥n";
msgTxt += "Subject: " + subjectStr + "¥n";
msgTxt += "¥n";
msgTxt += "There were " + outFolder1Count;
msgTxt +=" messsages with subjects containing " + keyword + ".¥n";
msgTxt += "There were " + outFolder2Count;
msgTxt += " messsages with subjects not containing " + keyword + ".¥n";

curMsg.setText(msgTxt);
```

8. MimeMessage の件名を設定します。

```
curMsg.setSubject(subjectStr);
```

9. MimeMessage の受信者を識別します。

```
curMsg.addRecipients(Message.RecipientType.TO, address);
```

10. IfsTransport を使用して MimeMessage を InternetAddress に送信します。

```
transport.sendMessage(curMsg, address);
```

11. LibrarySession を使用して、すべての操作をアトミックにコミットし、トランザクションを完了します。

```
  ifsSession.completeTransaction(transaction);
  transaction = null;
}
catch (Throwable e)
{
  e.printStackTrace();
}
```

12. トランザクションが正常に完了しない場合は、そのエラーを finally 句でキャッチしてトランザクションを強制終了します。トランザクションを catch 句で強制終了すると、エラーの原因がキャッチされない Throwable だった場合に問題が生じる可能性があります。finally 句で強制終了すると、トランザクションが完了するかロールバックされるかのどちらかになることが保証されます。

```
finally
{
  try
  {
    if (transaction != null)
    {
      ifsSession.abortTransaction(transaction);
      transaction = null;
    }
    if (ifsSession != null)
    {
       ifsSession.disconnect();
    }
  }
  catch (IfsException e)
  {
     e.printStackTrace();
  }
}
```

メッセージ・フォルダの操作

javax.mail の API および Oracle 9*i*FS Java API を使用すると、メッセージ・フォルダをプログラムで操作できます。たとえば、アプリケーションで次の手順を実行して、Oracle 9*i*FS 内のメール・フォルダ間でメッセージを移動できます。

1. javax.mail.Session を取得します。
2. Oracle 9*i*FS との LibrarySession を取得します。
3. javax.mail.Session から IfsStore を取得します。
4. IfsStore で Oracle 9*i*FS の LibrarySession を設定します。
5. Oracle 9*i*FS の LibrarySession とのトランザクションを開始します。
6. メッセージ・フォルダを IfsStore から javax.mail.Folder クラスのインスタンスとしてフェッチします。
7. open() メソッドを使用して javax.mail.Folders をオープンします。
8. getMessages() メソッドを使用して、javax.mail.Folders 内のメッセージをフェッチします。

9. appendMessages() メソッドを使用して、javax.mail.Folders にメッセージを追加します。
10. setFlags() メソッドを使用して、javax.mail.Folder 内のメッセージに削除済みフラグを付けます。
11. close() メソッドを使用して、javax.mail.Folder をクローズし、すべての削除済みメッセージを消去します。
12. トランザクションを完了します。
13. Oracle 9*i*FS の LibrarySession を切断します。
14. トランザクションが正常に完了しない場合は、強制終了します。

例 18-3 に、メッセージ操作のトランザクション管理に使用されるサンプル・コードから抜粋したコードを示します。

例 18-3 メッセージ・フォルダの操作

```
try
{
```

1. JavaMail とのセッションを確立します。

```
Session javaMailSession = Session.getDefaultInstance(
  new java.util.Properties(), null);
```

2. メッセージおよびその他の Oracle 9*i*FS オブジェクトの操作に使用する Oracle 9*i*FS との LibrarySession を確立します。

```
LibraryService ifsService =
     LibraryService.startService(serviceName, schemaPassword);
CleartextCredential cred =
     new CleartextCredential(userName, userPassword);
LibrarySession ifsSession = service.connect(cred, null);
```

3. JavaMail Session から IfsStore を取得します。トランザクション管理に使用できるように、IfsStore の LibrarySession を設定します。

```
IfsStore store = (IfsStore) javaMailSession.getStore("ifs");
store.setLibrarySession(ifsSession);
```

4. LibrarySession を使用してトランザクションを開始します。メッセージや他の Oracle 9*i*FS オブジェクトに対する操作をトランザクションに含めます。

```
Transaction transaction = ifsSession.beginTransaction();
```

5. メッセージ・フォルダを IfsStore から javax.mail.Folder クラスのインスタンスとしてフェッチします。

```
Folder inFolder = store.getFolder(parameterTable.getString("inFolder"));
Folder outFolder1 = store.getFolder(parameterTable.getString("outFolder1"));
Folder outFolder2 = store.getFolder(parameterTable.getString("outFolder2"));
```

6. javax.mail.Folder クラスの open() メソッドを使用して、メッセージ・フォルダを開きます。

```
inFolder.open(Folder.READ_WRITE);
```

7. javax.mail.Folder の getMessages() メソッドを使用して、フォルダ内のメッセージをフェッチします。

```
Message messages = inFolder.getMessages();

String keyword = parameterTable.getString("keyword");
String subject;
MimeMessage curMsg;
MimeMessage msgList[] = new MimeMessage[1];

for (int i = 0; i < messages.length; i++)
{
  curMsg = (MimeMessage) messages[i];
  msgList[0] = curMsg;
  subject = curMsg.getSubject();

  if(subject.indexOf(keyword) != -1)
  {
```

8. appendMessages() メソッドを使用して、javax.mail.Folder にメッセージを追加します。

```
   outFolder1.appendMessages(msgList);
   outFolder1Count++;
 }
 else
{
   outFolder2.appendMessages(msgList);
   outFolder2Count++;
 }
```

9. setFlags() メソッドを使用して、javax.mail.Folder 内のメッセージに削除済みフラグを付けます。

```
  inFolder.setFlags(msgList, deletedFlag, true);
}
```

10. close() メソッドを使用して、すべての削除済みメッセージを消去し、javax.mail.Folder をクローズします。

```
inFolder.close(true);
```

11. LibrarySession を使用して、すべての操作をアトミックにコミットし、トランザクションを完了します。

```
  ifsSession.completeTransaction(transaction);
  transaction = null;
}
catch (Throwable e)
{
  e.printStackTrace();
}
```

12. トランザクションが正常に完了しない場合は、そのエラーを finally 句でキャッチしてトランザクションを強制終了します。

```
finally
{
  try
  {
    if (transaction != null)
    {
        ifsSession.abortTransaction(transaction);
       transaction = null;
    }
    if (ifsSession != null)
    {
       ifsSession.disconnect();
    }
  }
  catch (IfsException e)
  {
     e.printStackTrace();
  }
}
```

サンプル・コード

javax.mail API と Oracle 9*i*FS のメッセージ機能クラスの具体例を示すために、例 18-4「IfsJavamailExample1.java」のサンプル・プログラムは、リポジトリに格納されている電子メール・メッセージに対して基本的な操作を実行しています。

注意： このサンプル・プログラムは、Oracle 9*i*FS のインストール時に `<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/devdoc/email` ディレクトリに組み込まれます。

サンプル・プログラムの要件

例 18-4 のコードを実行するには、次の依存性が必要です。

- Sendmail をインストールし、Oracle 9*i*FS で構成する必要があります。
- `BaseExample.java` をコンパイルし、生成された Java クラスを `<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/devdoc/email` ディレクトリに置く必要があります。
- `IfsJavamailExample1.java` をコンパイルし、生成された Java クラスを `<ORACLE_HOME>/9ifs/samplecode/oracle/ifs/examples/devdoc/email` ディレクトリに置く必要があります。

プログラムの実行内容

`IfsJavamailExample1.java` は `BaseExample.java` を拡張し、Oracle 9*i*FS に格納されている電子メール・メッセージに対してプログラムによる単純な操作のデモを行います。このプログラムは、コマンドラインからの実行時に次の引数を取ります。

- 基本的なシステム設定 － サーバー名、サーバー・パスワード
- セッション資格証明 － Oracle 9*i*FS ユーザーのユーザー名、パスワード
- 受信ボックス・フォルダ名 － 検索対象のフォルダ
- 送信ボックス・フォルダ名 1 － 選択した電子メール・メッセージがコピーされるフォルダ
- 送信ボックス・フォルダ名 2 － 選択した電子メール・メッセージがコピーされるフォルダ
- キーワード － 検索ワード（またはトークン）
- 電子メール・アドレス － プログラムによるアクティビティのログが送信される電子メール・アドレス

このプログラムは、実行時に inFolder に格納されているメッセージ内のキーワードを検索し、検索ワードを含むメッセージを outFolder1 に移動し、検索ワードを含んでいないメッセージを outFolder2 に移動します。最後に、結果を示す電子メール・レポートを生成し、引数で指定された電子メール・アドレスに送信します。コマンドライン入力の例を次に示します。

```
java IfsJavamailExample1 server=Local serverpassword=ifsuser username=tuser60
password=tuser60 name-IfsMailExample inFolder=inbox outFolder1=out1 outFolder2=out2
keyword=test address=tuser51@us.oracle.com
```

例 18-4 IfsJavamailExample1.java

```
package oracle.ifs.examples.devdoc.email;

/* Copyright (c) 2001 by Oracle Corporation. All Rights Reserved. */

import java.util.Enumeration;
import javax.mail.Flags;
import javax.mail.Flags.Flag;
import javax.mail.Folder;
import javax.mail.Header;
import javax.mail.Message;
import javax.mail.Provider;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import oracle.ifs.adk.mail.IfsStore;
import oracle.ifs.adk.mail.IfsTransport;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.common.IfsException;
import oracle.ifs.common.ParameterTable;
import oracle.ifs.common.Transaction;

public class IfsJavamailExample1 extends BaseExample
{
  public IfsJavamailExample1(String args[])
  {
    super(args);
  }

  public static void main(String args[])
  {
    new Thread (new IfsJavamailExample1(args)).start();
  }

  public void run()
```

```
{
  Flags           deletedFlag = new Flags(Flag.DELETED);
  Folder          inFolder;
  Folder          outFolder1;
  Folder          outFolder2;
  IfsStore        store;
  IfsTransport    transport;
  InternetAddress address[];
  LibrarySession  ifsSession = null;
  Message         messages[];
  MimeMessage     curMsg;
  MimeMessage     msgList[] = new MimeMessage[1];
  ParameterTable  parameterTable = getParameterTable();
  Session         session;
  String          keyword;
  Transaction     transaction = null;
  int             outFolder1Count = 0;
  int             outFolder2Count = 0;

  try
  {
    try
    {
       // Create Service
       session = Session.getDefaultInstance(new java.util.Properties(), null);

       // Create Store
       store = (IfsStore) session.getStore("ifs");

       // Authenticate to user
       ifsSession = connect();
       store.setLibrarySession(ifsSession);

       // Start the transaction
       transaction = ifsSession.beginTransaction();

       // Get inFolder Folder
       inFolder = store.getFolder(parameterTable.getString("inFolder"));

       inFolder.open(Folder.READ_WRITE);

       // Get outFolder1 Folder
       outFolder1 = store.getFolder(parameterTable.getString("outFolder1"));

       // Get outFolder2 Folder
       outFolder2 = store.getFolder(parameterTable.getString("outFolder2"));
```

```
// Get all messages from inFolder folder
messages = inFolder.getMessages();

// get the keyword
keyword = parameterTable.getString("keyword");

// list all messages from inFolder folder by subject
for (int i = 0; i < messages.length; i++)
{
  String subject;

  curMsg = (MimeMessage)messages[i];
  msgList[0] = curMsg;

  subject = curMsg.getSubject();
  if(subject.indexOf(keyword) != -1)
  {
     System.out.println("Placing msg " + subject + " in first folder");
    outFolder1.appendMessages(msgList);
    outFolder1Count++;
  }
  else
  {
     System.out.println("Placing msg " +
        subject + " in second folder");
    outFolder2.appendMessages(msgList);
    outFolder2Count++;
  }

  inFolder.setFlags(msgList, deletedFlag, true);
}

// Close the folder , expunging any deleted messages.
inFolder.close(true);

String msgTxt = "";
String addressStr = parameterTable.getString("address");
String subjectStr = "Mailbox Scan Results";

msgTxt = "To: " + addressStr + "¥n";
msgTxt += "Subject: " + subjectStr + "¥n";
msgTxt += "¥n";
msgTxt += "There were " + outFolder1Count;
msgTxt +=" messsages with subjects containing " + keyword + ".¥n";
msgTxt += "There were " + outFolder2Count;
msgTxt += " messsages with subjects not containing " +
          keyword + ".¥n";
```

```
      System.out.println("Creating outgoing message");
      curMsg = new MimeMessage(session);

      transport = (IfsTransport) session.getTransport("ifs1");
      transport.setLibrarySession(ifsSession);
      address = new InternetAddress[1];
      address[0] = new InternetAddress(addressStr);
      curMsg.addRecipients(Message.RecipientType.TO, address);
      curMsg.setSubject(subjectStr);
      curMsg.setText(msgTxt);
      transport.sendMessage(curMsg, address);

      // Commit the transaction
      ifsSession.completeTransaction(transaction);
      transaction = null;

      // This is very important as it is what dispatches events to
      //other processes.
      ifsSession.disconnect();
      ifsSession = null;
    }
    catch (Throwable e)
    {
      e.printStackTrace();
    }
    finally
    {
      if(transaction != null)
      {
        ifsSession.abortTransaction(transaction);
        transaction = null;
       }

       if(ifsSession != null)
       {
         // If this throws an exception we give up.  The events
         //caused by this program will be lost.
         ifsSession.disconnect();
        }
    }
  }
  catch (IfsException e)
  {
    // This catches an exception that may be thrown by the above
    // ifsSession.abortTransaction().
    e.printStackTrace();
```

```
        }
    }
}
```

A

エラー・メッセージ

この付録では、アプリケーションの開発中に発生する代表的なエラー・メッセージについて説明します。メッセージごとに、エラーの原因と、エラー条件を訂正するために実行できる措置が記載されています。

表 A-1 一般的なエラー・メッセージ

IFS-10170	**無効な名前 / 資格証明**
原因 :	入力された ProductShortName のログインおよびパスワードが不正です。
可能な措置 :	1. 適切なログインとパスワードを再入力します。 2. 指定した名前を持つユーザーが ProductShortName に存在するかどうかを確認します。
IFS-10200	**オブジェクトにアクセスできません（権限が不十分です）。**
原因 :	ユーザーがアクセス権を持っていない PublicObject（Document、Folder など）へのアクセスを試みました。
可能な措置 :	1. ユーザーがオブジェクトにアクセスできるように、所有者に権限（ACL）の変更を依頼します。 2. 権限が適切に設定されており、ユーザーはオブジェクトへのアクセスを禁止されている場合、他に該当する措置はありません。

表 A-1 一般的なエラー・メッセージ（続く）

IFS-10406	**無効な属性値の変換（{0} を Java{1} へ変換）。**
原因 :	通常、このエラーが発生するのは、Java API に直接書き込んでいる場合です。このエラーは、属性値を不正なデータ型に強制変換しています。無効な変換の例は、DATE から BOOLEAN への強制変換などです。有効な変換の例は、INTEGER から STRING への強制変換などです。 特に注意する必要があるのは、PUBLICOBJECT から PUBLICOBJECT への変換です。この変換は、ユーザーが AttributeValue 内で参照される PublicObject へのアクセス権を持っていない場合は失敗します。 パラメータ {0} および {1} は、エラーの原因を示す実際の値になります。
可能な措置 :	1. データ型をチェックし、互換性のあるデータ型に変更します。 2. PublicObject に強制変換している場合は、ユーザーが AttributeValue で参照される PublicObject にアクセスできるかどうかをチェックします。
IFS-10600	**ライブラリ接続を構築できません。**
原因 :	このエラーが発生するのは、ProductShortName リポジトリからデータベースに接続できない場合です。通常、このエラーの原因は、サービス・プロパティ・ファイルに指定されている無効なデータベース・ユーザー名または無効なデータベース・パスワードです。また、サービス・プロパティ・ファイル内の DatabaseUrl 設定が無効な場合にも、このエラーが発生することがあります。
可能な措置 :	1. SQL*Plus または同様のツールを使用してデータベースに接続し、データベース・ユーザー名、パスワードおよび TNS 名が適切に設定されているかどうかを確認します。 2. この例外により、より詳細に原因を記述する別の例外がカプセル化されているかどうかをチェックします。
IFS-10620	**接続プールを構築できません。**
原因 :	通常、このエラーが発生するのは、データベース接続を確立できない場合です。一般に、エラー 10633 が発生すると、このエラーが発生します。
可能な措置 :	このエラーの原因がエラー 10633 の場合は、SQL*Plus または同様のツールを使用してデータベースに接続し、データベース・ユーザー名、パスワードおよび TNS 名が適切に設定されているかどうかを確認します。

表 A-1 一般的なエラー・メッセージ（続く）

IFS-10633	**ライブラリ接続を作成できません。**
原因 :	通常、このエラーが発生するのは、データベース接続を確立できない場合です。一般に、エラー 10600 が発生すると、このエラーが発生します。
可能な措置 :	このエラーの原因がエラー 10600 の場合は、SQL*Plus または同様のツールを使用してデータベースに接続し、データベース・ユーザー名、パスワードおよび TNS 名が適切に設定されているかどうかを確認します。
IFS-12200	**無効な項目名が指定されました（{0}）。**
原因 :	ProductShortName Collection の 1 つで名前によるオブジェクトの参照を試みましたが、その名前を持つオブジェクトは存在しません。このエラーが発生するのは、getItems() メソッドを Collection クラスで直接コールするか、Collection の 1 つにアクセスする操作を実行して間接的にコールする場合です。後者の例として、新規 Document の作成などの操作で ClassObject 名を指定したが、その名前を持つ ClassObject が存在しない場合などがあります。
可能な措置 :	指定した名前をチェックして、有効な名前を再入力します。
IFS-12620	**パーサー : 構文エラー（{0}）**
原因 :	ProductShortName リポジトリに取り込むドキュメント・ストリームの解析中に、パーサーにより構文エラーが検出されました。パラメータでは、構文エラーを受け持つトークンが識別されます。たとえば、SimpleXmlParser は、XML ファイルの解析中に不明なタグが見つかると、この例外を発生させます。パラメータは不明なタグの値です。
可能な措置 :	ドキュメント本体を修正し、構文エラーを解決して、ProductShortName に再発行します。
IFS-20000	**リポジトリ・パラメータ（{0}）を取得できません。**
原因 :	通常、このエラーが発生するのは、部分的にインストール済みの ProductShortName インスタンスまたは極端に古い ProductShortName インスタンス（1.0.8.0.0 より前など）に対して、ProductShortName の実行を試みた場合です。これは特に、表示されるパラメータが文字列 SCHEMAVERSION の場合に該当します。
可能な措置 :	このエラーが発生した ProductShortName インスタンスが正常にインストールされており、バージョン 1.0.8.0.0 以上であることを確認します。

表 A-1 一般的なエラー・メッセージ（続く）

IFS-20001	**スキーマ・バージョンを取得できません。**
原因 :	通常、このエラーの原因はエラー 20000 で、つまり、リポジトリ・パラメータ SCHEMAVERSION を取得できないことです。一般に、このエラーが発生するのは、部分的にインストール済みの ProductShortName インスタンスまたは極端に古い ProductShortName インスタンス（1.0.8.0.0 より前など）に対して、ProductShortName の実行を試みた場合です。
可能な措置 :	このエラーが発生した ProductShortName インスタンスが正常にインストールされており、バージョン 1.0.8.0.0 以上であることを確認します。
IFS-20010	**サービス構成プロパティ（{0}）を取得できません。**
原因 :	このエラーが発生するのは、ProductShortName プロセスの起動を試みるときに、ProductShortName リポジトリで検索できないサービス・プロパティ・ファイル名を指定した場合です。エラーで示されるパラメータは、指定したサービス・プロパティ・ファイルの名前です。指定したサービス・プロパティ・ファイルは、プロセスの CLASSPATH 設定に含まれるディレクトリの 1 つから派生する、oracle.ifs.server.properties パッケージに存在する必要があります。
可能な措置 :	指定した名前を持つサービス・プロパティ・ファイルが存在し、現行の CLASSPATH からアクセス可能であるかどうかをチェックします。
IFS-21008	***i*FS サービスに接続できません。**
原因 :	ProductShortName セッションの確立に失敗しました。通常、これは指定された資格証明（名前 / パスワードの組合せ）が不正であることが原因です。この場合、このエラーにはエラー 10170 がカプセル化されています。他のすべての認証エラーの場合は（発生はまれですが）、エラー 10150 がカプセル化されています。
可能な措置 :	ログイン・エラーの原因が無効な資格証明の場合は、有効な資格証明を再入力して ProductShortName セッションを確立します。

表 A-1 一般的なエラー・メッセージ（続く）

IFS-30002	**新規ライブラリ・オブジェクトを作成できません。**
原因 :	新規 ProductShortName オブジェクトの作成に失敗しました。このエラーの実際の原因は、この例外にカプセル化された例外に記述されます。たとえば、新規オブジェクトの作成時に一意制約に違反した場合、最上位の例外は 30002 で、そこに例外 30010: 「属性は一意（{0}）になりません。」がカプセル化されています。
可能な措置 :	オブジェクト作成エラーの原因を調べ、対処措置を実行してから再試行します。
IFS-34611	**バージョン・シリーズの予約中にエラーが発生しました。**
原因 :	バージョニング対象の PublicObject に対応付けられている VersionSeries オブジェクトの予約など、バージョニング対象の PublicObject のチェックアウト中にエラーが発生しました。このエラーの原因には、次のように様々な条件が考えられ、ほとんどの場合はカプセル化されている例外を検査することで記述されます。たとえば、他のユーザーがすでにを予約している場合、カプセル化される例外は 34602: 「許可されない操作です。バージョン・シリーズは予約されています。」です。
可能な措置 :	1. VersionSeries が他のユーザーにより予約またはロックされていないかどうかを確認します。 2. Family が他のユーザーによりロックされていないかどうかを確認します。 3. VersionSeries の最終バージョンが他のユーザーによりロックされていないかどうかを確認します。 4. 現行のユーザーがターゲット VersionSeries に対して権限 AddVersion を持っているかどうかを確認します。
IFS-46113	**そのようなサーバー（{0}）はありません。**
原因 :	ServerManager インタフェースの 1 つを使用して、名前による ProductShortName Server の参照を試みましたが、指定されたサーバーは存在しません。このエラーが発生するのは、名前を正しく指定していない場合、またはこの名前と一致するサーバーが実行中でない場合です。
可能な措置 :	1. ServerManager のコマンドライン・インタフェースで `list servers` コマンドを使用するなど、ServerManager インタフェースを使用して、アクティブ・サーバー・リストを再チェックします。 2. ServerManager またはプロトコル・サーバー、あるいはその両方のログをチェックして、指定した名前を持つサーバーが予期せずに停止していないかどうかを調べます。

表 A-1 一般的なエラー・メッセージ（続く）

IFS-46114	**サーバー名 {0} があいまいです。サーバーの識別子を指定します：（{1}）**
原因：	ServerManager のコマンドライン・インタフェースで名前による ProductShortName Server の参照を試みましたが、この名前を持つサーバーが複数存在します。例外テキストにリスト表示される識別子は、指定された名前を持つサーバーの一意サーバー識別子です。これらの識別子をサーバー名のかわりに使用して、サーバー上で操作を実行できます。
可能な措置：	ServerManager のコマンドライン・インタフェースで、-i オプションを使用してサーバーをリスト表示します。次に、サーバー名のかわりに識別子を使用して、オリジナルのサーバー要求を再発行します。

索引

A

B

C

D

E

F

G

H

I

J

L

M

O

P

R

S

T

U

V

W

X

あ

い

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

つ

て

と

な

に

の

は

ひ

ふ

へ

ほ

ま

め

も

ゆ

よ

ら

り

る

れ

ろ

わ