BEA WebLogic Server™

WebLogic Security サービスの開発

BEA WebLogic Server バージョン 7.0
マニュアルの改訂 : 2003 年 3 月 28 日

著作権

限定的権利条項

商標または登録商標

WebLogic Security サービスの開発

パート番号	マニュアルの改訂	ソフトウェアのバージョン
なし	2003 年 3 月 28 日	BEA WebLogic Server バージョン 7.0

目次

4. ID アサーション プロバイダ

5. プリンシパル検証プロバイダ

6. 認可プロバイダ

7. 裁決プロバイダ

8. ロール マッピング プロバイダ

9. 監査プロバイダ

このマニュアルの内容

このマニュアルでは、セキュリティ ベンダとアプリケーション開発者向けに、BEA WebLogic Server™ 用の新しいセキュリティ プロバイダを開発するのに必要な情報を提供します。

マニュアルの内容は以下のとおりです。

- 第 1 章「WebLogic Server で使用するセキュリティ プロバイダの開発について」では、WebLogic Server 用のセキュリティ プロバイダの開発に関する基本事項について説明します。このガイドの対象読者と前提条件を示し、開発プロセスの概要について説明します。
- 第 2 章「設計上の考慮事項」では、セキュリティ プロバイダの一般的なアーキテクチャ、および SSPI の実装と MBean タイプの生成について理解しておく必要のある背景的な情報について説明します。また、管理ユーティリティの使い方と、セキュリティ プロバイダが WebLogic リソースと対話する方法についても説明します。さらに、この章では、カスタム セキュリティ プロバイダをセキュリティ プロバイダが必要とする情報を格納したデータベースで動作させる方法を提案します。
- 第 3 章「認証プロバイダ」では、認証プロセス (単純認証) について説明し、カスタム認証プロバイダに関連する各タイプのセキュリティ サービス プロバイダ インタフェース (SSPI) の実装方法を解説します。またこの章では、JAAS LoginModule についても説明します。
- 第 4 章「ID アサーション プロバイダ」では、認証プロセス (トークンを使用した境界認証) について説明し、カスタム ID アサーション プロバイダに関連する各タイプのセキュリティ サービス プロバイダ インタフェース (SSPI) の実装方法を解説します。
- 第 5 章「プリンシパル検証プロバイダ」では、プリンシパル検証プロバイダが、サブジェクトに格納されたプリンシパルに対する署名と信頼性の確認によって、認証プロバイダを補佐するしくみについて説明します。また、カスタム プリンシパル検証プロバイダの開発方法についても説明します。

- 第 6 章「認可プロバイダ」では、認可プロセスについて説明し、カスタム認可プロバイダに関連する各タイプのセキュリティ サービス プロバイダ インタフェース (SSPI) の実装方法を解説します。
- 第 7 章「裁決プロバイダ」では、裁決プロセスについて説明し、カスタム裁決プロバイダに関連する各タイプのセキュリティ サービス プロバイダ インタフェース (SSPI) の実装方法を解説します。
- 第 8 章「ロール マッピング プロバイダ」では、ロール マッピング プロセスについて説明し、カスタム ロール マッピング プロバイダに関連する各タイプのセキュリティ サービス プロバイダ インタフェース (SSPI) の実装方法を解説します。
- 第 9 章「監査プロバイダ」では、監査プロセスについて説明し、カスタム監査プロバイダに関連する各タイプのセキュリティ サービス プロバイダ インタフェース (SSPI) の実装方法を解説します。またこの章では、他のタイプのセキュリティ プロバイダから監査を行う方法についても説明します。
- 第 11 章「資格マッピング プロバイダ」では、資格マッピング プロセスについて説明し、カスタム資格マッピング プロバイダに関連する各タイプのセキュリティ サービス プロバイダ インタフェース (SSPI) の実装方法を解説します。
- 第 10 章「カスタム セキュリティ プロバイダからのイベントの監査」では、開発の対象となるカスタム セキュリティ プロバイダに監査機能を追加する方法について説明します。
- 第 12 章「カスタム セキュリティ プロバイダ用のコンソール拡張の記述」では、特にカスタム セキュリティ プロバイダと共に使用するコンソール拡張の記述について説明します。
- 付録 A「MBean 定義ファイル (MDF) 要素の構文」では、有効な MDF で使用可能なすべての要素と属性について説明します。MDF は MBean タイプを生成するのに用いられる XML ファイルで、それらの MBean タイプによってカスタム セキュリティ プロバイダの管理が可能になります。

このガイドの対象読者

『WebLogic Security サービスの開発』は、WebLogic Server 用の独自のセキュリティ プロバイダを作成しようとお考えの独立系ソフトウェア ベンダ (ISV) を対象としています。ここでは、このマニュアルをお読みになる ISV の大半はセキュリティの概念をしっかりと理解している高機能アプリケーションの開発者であり、セキュリティの基礎概念については説明を要しないものと想定しています。また、このマニュアルでは、セキュリティ ベンダおよびアプリケーション開発者は BEA WebLogic Server と Java (JMX (Java Management eXtensions) を含む) に精通しているものと想定しています。

このガイドの前提条件

このガイドを読む前に、『WebLogic Security の紹介』の以下の節に目を通しておいてください。

- 「セキュリティ プロバイダ」
- 「WebLogic Security フレームワーク」

WebLogic Server のセキュリティには、この他に理解しておく必要のある固有の用語や概念が多数あります。これらの用語と概念は WebLogic Server のセキュリティに関するマニュアルに登場しますが、用語については『WebLogic Security の紹介』の「用語」の節で、概念については「セキュリティの基礎概念」の節で定義されています。

e-docs Web サイト

BEA 製品のドキュメントは、BEA の Web サイトで入手できます。BEA のホーム ページで [製品のドキュメント] をクリックします。

このマニュアルの印刷方法

Web ブラウザの [ファイル | 印刷] オプションを使用すると、Web ブラウザからこのマニュアルを一度に 1 章ずつ印刷できます。

このマニュアルの PDF 版は、Web サイトで入手できます。PDF を Adobe Acrobat Reader で開くと、マニュアルの全体 (または一部分) を書籍の形式で印刷できます。PDF を表示するには、WebLogic Server ドキュメントのホーム ページを開き、[ドキュメントのダウンロード] をクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader は Adobe の Web サイト (http://www.adobe.co.jp) で無料で入手できます。

関連情報

BEA の Web サイトでは、WebLogic Server の全マニュアルを提供しています。セキュリティ ベンダや、セキュリティ プロバイダを扱うアプリケーション開発者が関心を持ちそうな WebLogic Server マニュアルには、このマニュアルの他に以下のものがあります。

- 『WebLogic Security の紹介』
- 『WebLogic Security の管理』
- 『WebLogic Security プログラマーズ ガイド』
- 『WebLogic リソースのセキュリティ』
- 『プロダクション環境のロックダウン』
- 『BEA WebLogic Server 7.0 へのアップグレード』の「WebLogic Server 6.x からバージョン 7.0 へのアップグレード」の「セキュリティのアップグレード」

さらに、上記以外のリソースとして以下のものがあります。

- 『BEA WebLogic Server FAQ 集』の「FAQ: セキュリティ」
- 「WebLogic クラスの Javadoc」

サポート情報

BEA のドキュメントに関するユーザからのフィードバックは弊社にとって非常に重要です。質問や意見などがあれば、電子メールで docsupport-jp@beasys.com までお送りください。寄せられた意見については、ドキュメントを作成および改訂する BEA の専門の担当者が直に目を通します。

電子メールのメッセージには、ご使用のソフトウェアの名前とバージョン、およびドキュメントのタイトルと日付をお書き添えください。本バージョンの BEA WebLogic Server について不明な点がある場合、または BEA WebLogic Server のインストールおよび動作に問題がある場合は、BEA WebSupport (www.bea.com) を通じて BEA カスタマ サポートまでお問い合わせください。カスタマ サポートへの連絡方法については、製品パッケージに同梱されているカスタマ サポート カードにも記載されています。

カスタマ サポートでは以下の情報をお尋ねしますので、お問い合わせの際はあらかじめご用意ください。

- お名前、電子メール アドレス、電話番号、ファクス番号
- 会社の名前と住所
- お使いの機種とコード番号
- 製品の名前とバージョン
- 問題の状況と表示されるエラー メッセージの内容

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

表記法	適用
〔Ctrl〕+〔Tab〕	複数のキーを同時に押すことを示す。
斜体	強調または書籍のタイトルを示す。

等幅テキスト	コード サンプル、コマンドとそのオプション、データ構造体とそのメンバー、データ型、ディレクトリ、およびファイル名とその拡張子を示す。等幅テキストはキーボードから入力するテキストも示す。
	例:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

斜体の等幅テキスト	プレースホルダを示す。 例: `String *CustomerName*;`

大文字の等幅テキスト	デバイス名、環境変数、および論理演算子を示す。
	例:
	`LPT1`
	`BEA_HOME`
	`OR`

{ }	構文の中で複数の選択肢を示す。

[ ]	構文の中で任意指定の項目を示す。
	例:
	`java utils.MulticastTest -n *name* -a *address*`
	`[-p *portnumber*] [-t *timeout*] [-s *send*]`

表記法	適用

|	構文の中で相互に排他的な選択肢を区切る。
	例：
	`java weblogic.deploy [list|deploy|undeploy|update]`
	`password {application} {source}`

...	コマンドラインで以下のいずれかを示す。 ■ 引数を複数回繰り返すことができる。 ■ 任意指定の引数が省略されている。 ■ パラメータや値などの情報を追加入力できる。
. . .	コード サンプルまたは構文で項目が省略されていることを示す。

1 WebLogic Server で使用するセキュリティ プロバイダの開発について

以下の節では、セキュリティ プロバイダの開発についての概要を示します。

このガイドの対象読者

この『WebLogic Security サービスの開発』マニュアルは、WebLogic Server 用のセキュリティ プロバイダを独自に作成しようとお考えの独立系ソフトウェア ベンダ (ISV) を対象としています。ここでは、このマニュアルをお読みになる ISV の大半はセキュリティの概念をしっかりと理解している高機能アプリケーションの開発者であり、セキュリティの基礎概念については説明を要しないものと想定しています。また、このマニュアルでは、セキュリティ ベンダおよびアプリケーション開発者は BEA WebLogic Server と Java (JMX (Java Management eXtensions) を含む) に精通しているものと想定しています。

このガイドの前提条件

このガイドを読む前に、『WebLogic Security の紹介』の以下の節に目を通しておいてください。

- 「セキュリティ プロバイダ」
- 「WebLogic Security フレームワーク」

WebLogic Server のセキュリティには、この他に理解しておく必要のある固有の用語や概念が多数あります。これらの用語と概念は WebLogic Server のセキュリティに関するマニュアルに登場しますが、用語については『WebLogic Security の紹介』の「用語」の節で、概念については「セキュリティの基礎概念」の節で定義されています。

開発プロセスの概要

この節では、新しいセキュリティ プロバイダを開発するプロセスの概要を説明します。各手順の詳細については、このマニュアルの中で後述します。

カスタム セキュリティ プロバイダの主な開発手順は以下のとおりです。

- 1-2 ページの「カスタム セキュリティ プロバイダの設計」
- 1-4 ページの「SSPI の実装によるカスタム セキュリティ プロバイダ用の実行時クラスの作成」
- 1-4 ページの「カスタム セキュリティ プロバイダをコンフィグレーションおよび管理する MBean タイプの生成」
- 1-6 ページの「コンソール拡張の記述」
- 1-7 ページの「カスタム セキュリティ プロバイダのコンフィグレーション」
- 1-8 ページの「セキュリティ ポリシー、セキュリティ ロール、および資格マップを管理するためのメカニズムを提供する」

カスタム セキュリティ プロバイダの設計

設計プロセスには、次の手順があります。

1. WebLogic セキュリティ プロバイダの説明を検討して、カスタム セキュリティ プロバイダを作成する必要があるかどうか判断します。

WebLogic セキュリティ プロバイダの説明は、『WebLogic Security の紹介』の「WebLogic セキュリティ プロバイダ」とこのマニュアルの「カスタム <*Provider_Type*> プロバイダを開発する必要があるか」という節にあります。<*Provider_Type*> は、認証、ID アサーション、プリンシパル検証、認可、裁決、ロール マッピング、監査、または資格マッピングのいずれかです。

2. 作成するカスタム セキュリティ プロバイダのタイプを決めます。

   『WebLogic Security の紹介』の「セキュリティ プロバイダのタイプ」で説明されているように、タイプは認証、ID アサーション、プリンシパル検証、認可、裁決、ロール マッピング、監査、または資格マッピングのいずれかです。カスタム セキュリティ プロバイダは、WebLogic Server にあらかじめ用意されている WebLogic セキュリティ プロバイダの補強または代わりとして使用できます。

3. カスタム セキュリティ プロバイダ用の実行時クラスを作成するために実装するセキュリティ サービス プロバイダ インタフェース (SSPI) を、作成するセキュリティ プロバイダのタイプに基づいて特定します。

   さまざまなセキュリティ プロバイダ タイプの SSPI の説明については、2-3 ページの「セキュリティ サービス プロバイダ インタフェース (SSPI)」を参照してください。要約は、2-10 ページの「SSPI クイック リファレンス」で参照できます。

4. SSPI を 1 つまたは 2 つのどちらの実行時クラスで実装するかを決めます。

   それらのオプションの詳細については、2-7 ページの「SSPI 階層を理解し、実行時クラスを 1 つ作成するのか 2 つ作成するのかを決定する」を参照してください。

5. カスタム セキュリティ プロバイダを管理する MBean タイプを生成するために拡張する必要のある必須 SSPI MBean を特定します。ユーザ、グループ、セキュリティ ロール、およびセキュリティ ポリシーの処理など、カスタム セキュリティ プロバイダを管理する追加機能を提供するには、実装する任意 SSPI MBean を特定することも必要です。

   SSPI MBean の詳細については、2-11 ページの「セキュリティ サービス プロバイダ インタフェース (SSPI) MBean」を参照してください。要約は、2-20 ページの「SSPI MBean クイック リファレンス」で参照できます。

6. カスタム セキュリティ プロバイダで必要なデータベースを初期化する方法を決めます。カスタム セキュリティ プロバイダに単純なデータベースを作成させるか、情報の格納された既存のデータベースを使用するようにカスタム セキュリティ プロバイダをコンフィグレーションすることができます。

   それら 2 つのデータベース初期化オプションの詳細については、2-37 ページの「セキュリティ プロバイダ データベースの初期化」を参照してください。

7. カスタム セキュリティ プロバイダが WebLogic リソースのセキュリティ ポリシーとの対話の一部として実行するのに必要なデータベースの「準備」を特定します。この準備には、デフォルト グループ、セキュリティ ロール、またはセキュリティ ポリシーの作成などがあります。

   詳細については、2-24 ページの「セキュリティ プロバイダと WebLogic リソース」を参照してください。

SSPI の実装によるカスタム セキュリティ プロバイダ用の実行時クラスの作成

1 つまたは 2 つの実行時クラスにおいて、各メソッドの実装を提供することによって特定された SSPI を実装します。それらのメソッドには、カスタム セキュリティ プロバイダから提供されるセキュリティ サービスの具体的なアルゴリズムが組み込まれていなければなりません。それらのメソッドには、サービスがどのように動作すべきであるかが記述されています。

このタスクの手順は、作成するセキュリティ プロバイダのタイプによって異なります。この手順については、各セキュリティ プロバイダの詳細に関する節の「適切な SSPI を使用して実行時クラスを作成する」を参照してください。

カスタム セキュリティ プロバイダをコンフィグレーションおよび管理する MBean タイプの生成

MBean タイプを生成するには、次の手順に従います。

1. 必須 SSPI MBean を拡張し、任意 SSPI MBean を実装し、プロバイダのコンフィグレーションと管理に必要なカスタム属性および操作を追加するカスタム セキュリティ プロバイダの MBean 定義ファイル (MDF) を作成します。

   MDF の詳細については、2-13 ページの「MBean 定義ファイル (MDF) の基本的な要素を理解する」を参照してください。このタスクの手順については、各セキュリティ プロバイダの詳細に関する節の「MBean 定義ファイル (MDF) を作成する」を参照してください。

2. WebLogic MBeanMaker を通じて MDF を実行し、カスタム セキュリティ プロバイダの MBean タイプの中間ファイル (MBean インタフェース、MBean 実装、および MBean 情報ファイルを含む) を生成します。

   WebLogic MBeanMaker について、および WebLogic MBeanMaker がどのように MDF を使用して Java ファイルを生成するかについては、2-17 ページの「WebLogic MBeanMaker によって提供されるものを理解する」を参照してください。このタスクの手順については、各セキュリティ プロバイダの詳細に関する節の「WebLogic MBeanMaker を使用して MBean タイプを生成する」を参照してください。

3. MBean の実装ファイルを編集して、任意 SSPI MBean の実装から継承されるすべてのメソッドの内容のほか、MDF に追加されたカスタム属性および操作の結果として生成されるメソッド スタブの内容も入力します。

4. 修正された中間ファイル (MBean タイプ用) と実行時クラスを WebLogic MBeanMaker を通じて実行し、MBean JAR ファイル (MJF) という JAR ファイルを生成します。

   このタスクの手順については、各セキュリティ プロバイダの詳細に関する節の「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」を参照してください。

5. MBean JAR ファイル (MJF) を WebLogic Server 環境にインストールします。

   このタスクの手順については、各セキュリティ プロバイダの詳細に関する節の「WebLogic Server 環境に MBean タイプをインストールする」を参照してください。

コンソール拡張の記述

コンソール拡張を使用すると、JavaServer Pages (JSP) を WebLogic Server Administration Console に追加して、カスタム セキュリティ プロバイダの管理とコンフィグレーションをサポートできます。コンソール拡張を使用すると、存在しない Administration Console サポートを追加したり、管理目的の対話をカスタマイズしたりできます。

WebLogic Server Administration Console でカスタム セキュリティ プロバイダ用の完全なコンフィグレーションおよび管理サポートを利用するには、以下の場合にコンソール拡張を記述する必要があります。

- カスタム セキュリティ プロバイダ用の MBean タイプを生成するときに任意 SSPI MBean を実装しないことを決定したが、Administration Console でカスタム セキュリティ プロバイダをコンフィグレーションおよび管理する場合 (つまり、WebLogic Server コマンドライン インタフェースを代わりに使用しない場合)

  BEA では、MBean タイプの生成 (1-4 ページの「カスタム セキュリティ プロバイダをコンフィグレーションおよび管理する MBean タイプの生成」を参照) によってカスタム セキュリティ プロバイダをコンフィグレーションおよび管理することをお勧めします。しかし、ユーザが記述したコンソール拡張を介してカスタム セキュリティ プロバイダを完全にコンフィグレーションおよび管理することもできます。

- カスタム認証プロバイダ以外の任意 SSPI MBean を実装する場合

  任意 SSPI MBean を実装してカスタム認証プロバイダを開発する場合は、Administration Console で MBean タイプの属性が自動的にサポートされることになります (任意 SSPI MBean から継承される)。カスタム認可プロバイダなどの他のタイプのカスタム セキュリティ プロバイダでは、このサポートは受けられません。

- シンプルなデータ型として表すことができないカスタム属性を、MBean 定義ファイル (MDF) (カスタム セキュリティ プロバイダの MBean タイプを作成するために使用される) に追加する場合

  カスタム セキュリティ プロバイダの [詳細] タブにカスタム属性が自動的に表示されるのは、それらがシンプルなデータ型 (文字列、MBean、ブール、整数値) として表される場合のみです。非定型のデータ型 (フィンガープリントのイメージなど) として表されるカスタム属性がある場合、

Administration Console ではカスタマイズを行わない限りその属性を表示できません。

- カスタム操作を MBean 定義ファイル (MDF) (カスタム セキュリティ プロバイダの MBean タイプを作成するために使用される) に追加する場合

  カスタム操作はさまざまなので、Administration Console ではそれらを自動的に表示または処理する方法が分かりません。カスタム操作の例としては、声紋用のマイクや、インポート / エクスポート ボタンなどがあります。Administration Console では、カスタマイズしない限りこれらのプロセスを表示できません。

上記のような状況に遭遇した場合でも、WebLogic Server Administration Console の使用を可能にするコンソール拡張機能を記述したくない場合は、代わりに WebLogic Server コマンドライン インタフェースを使用してカスタム セキュリティ プロバイダを管理およびコンフィグレーションできます。WebLogic Server コマンドライン インタフェースの詳細については、『管理者ガイド』の「WebLogic Server コマンドライン インタフェース リファレンス」を参照してください。

Administration Console は、以下の場合にも拡張します。

- 企業ブランディング ― たとえば、カスタム セキュリティ プロバイダのコンフィグレーションおよび管理に使用するページにロゴを表示する場合など
- 連結 ― たとえば、カスタム セキュリティ プロバイダのコンフィグレーションおよび管理に使用するすべてのフィールドを、個別のタブや場所ではなく 1 ページに表示する場合など

コンソール拡張の詳細については、『Administration Console の拡張』および第 12 章「カスタム セキュリティ プロバイダ用のコンソール拡張の記述」を参照してください。

カスタム セキュリティ プロバイダのコンフィグレーション

注意： コンフィグレーション プロセスは、カスタム セキュリティ プロバイダを開発した本人、または指定された管理者が遂行できます。

コンフィグレーション プロセスでは、WebLogic Server Administration Console (または WebLogic Server コマンドライン インタフェース) を使用してカスタム セキュリティ プロバイダにコンフィグレーション情報を提供します。カスタム セキュリティ プロバイダを管理するための MBean タイプを生成している場合、Administration Console でカスタム セキュリティ プロバイダを「コンフィグレーション」することは、MBean タイプの特定のインスタンスを作成することも意味します。

Administration Console を使用したセキュリティ プロバイダのコンフィグレーションの詳細については、『WebLogic Security の管理』の「デフォルト セキュリティ コンフィグレーションのカスタマイズ」を参照してください。WebLogic Server コマンドライン インタフェースの使い方については、『管理者ガイド』の「WebLogic Server コマンドライン インタフェース リファレンス」を参照してください。

セキュリティ ポリシー、セキュリティ ロール、および資格マップを管理するためのメカニズムを提供する

特定のタイプのセキュリティ プロバイダでは、それらに関連付けられているセキュリティ データを管理する方法を管理者に提供する必要があります。たとえば認可プロバイダでは、セキュリティ ポリシーを管理する方法を管理者に提供しなければなりません。同じように、ロール マッピング プロバイダではセキュリティ ロールを、資格マッピング プロバイダでは資格マップを管理する方法が管理者に必要になります。

WebLogic 認可、ロール マッピング、および資格マッピングの各プロバイダの場合、管理者用の管理メカニズムが WebLogic Server Administration Console に既に用意されています。ただし、これらのセキュリティ プロバイダのカスタム版を開発する場合はこれらのメカニズムを継承しないで、セキュリティ ポリシー、セキュリティ ロール、および資格マップを管理するための独自の方法を提供する必要があります。これらのメカニズムにはカスタム セキュリティ プロバイダのデータベースから適切なセキュリティ データを読み書きすることが必須となりますが、Administration Console とは統合しても統合しなくてもかまいません。

詳細については、以下のいずれかの節を参照してください。

- 6-26 ページの「セキュリティ ポリシーを管理するためのメカニズムを提供する」(カスタム認可プロバイダの場合)
- 8-29 ページの「セキュリティ ロールを管理するためのメカニズムを提供する」(カスタム ロール マッピング プロバイダの場合)
- 11-20 ページの「資格マップを管理するためのメカニズムを提供する」(カスタム資格マッピング プロバイダの場合)

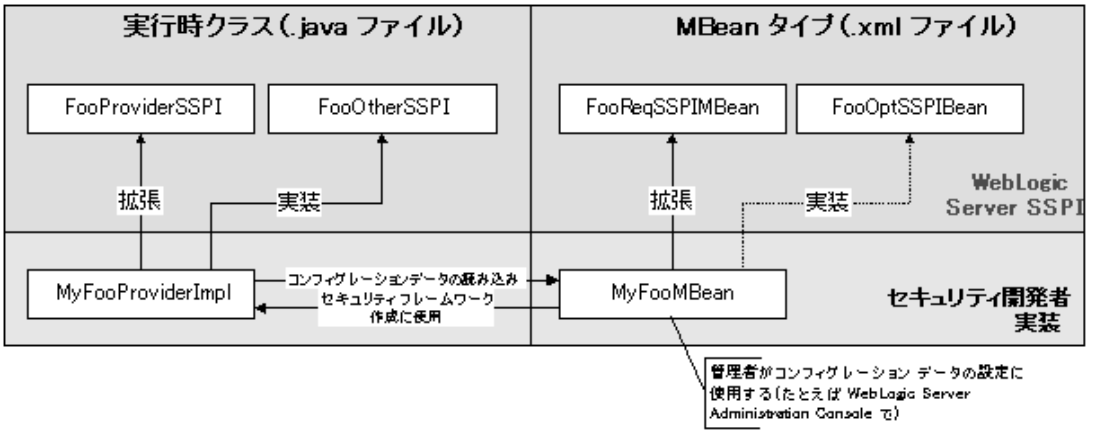
2　設計上の考慮事項

開発作業を綿密に計画すると、カスタム セキュリティ プロバイダの開発に要する時間と労力を大幅に削減できます。以下の節では、最初に知っておくと役立つ、セキュリティ プロバイダの概念と機能について詳しく説明します。

セキュリティ プロバイダの一般的なアーキテクチャ

作成できるセキュリティ プロバイダにはさまざまなタイプがありますが (『WebLogic Security の紹介』の「セキュリティ プロバイダのタイプ」を参照)、それらのプロバイダはすべて同じアーキテクチャに基づきます。図 2-1 は、セキュリティ プロバイダの一般的なアーキテクチャを示しています。その説明は、図の下を参照してください。

図 2-1　セキュリティ プロバイダのアーキテクチャ



注意： SSPI および SSPI を使用して作成する実行時クラス (実装) は両方とも `.java` ファイルです。図 2-1 の左側を参照してください。

図 2-1 の右側のファイルと同様、`MyFooMBean` は `.xml` ファイルとして開始します。このファイルで、SSPI MBean を拡張 (および任意で実装) します。この MBean 定義ファイル (MDF) が WebLogic MBeanMaker ユーティリティで実行される場合、このユーティリティでは MBean タイプ用の `.java` ファイルが生成されます。1-4 ページの「カスタム セキュリティ プロバイダをコンフィグレーションおよび管理する MBean タイプの生成」を参照してください。

図 2-1 は、カスタム セキュリティ プロバイダの開発時に作成された 1 つの実行時クラス (`MyFooProviderImpl`) と MBean タイプ (`MyFooMBean`) の関係を示しています。プロセスは WebLogic Server インスタンスの起動時に始まり、WebLogic Security フレームワークは以下のことを行います。

1. セキュリティ レルムでセキュリティ プロバイダと関連付けられた MBean タイプを見つけます。
2. MBean タイプからセキュリティ プロバイダの実行時クラス (2 つある場合は「Provider」SSPI を実装する方の実行時クラス) の名前を取得します。
3. セキュリティ プロバイダが初期化 (コンフィグレーション データの読み込み) に使用する適切な MBean インスタンスを渡します。

このため、実行時クラスと MBean タイプの両方で「セキュリティ プロバイダ」と呼ばれるものが形成されます。

セキュリティ サービス プロバイダ インタフェース (SSPI)

1-2 ページの「開発プロセスの概要」で説明されているように、カスタム セキュリティ プロバイダの開発ではまず最初に多くのセキュリティ サービス プロバイダ インタフェース (SSPI) を実装して実行時クラスを作成します。この節では、以下の助けになる情報を提供します。

- 2-3 ページの「重要な制限を理解する」
- 2-4 ページの「「Provider」SSPI の目的を理解する」
- 2-5 ページの「実装する「Provider」インタフェースを決定する」
- 2-7 ページの「SSPI 階層を理解し、実行時クラスを 1 つ作成するのか 2 つ作成するのかを決定する」

また、この節では各タイプのセキュリティ プロバイダでどの SSPI を実装できるのかを示す SSPI クイック リファレンスも提供します。

重要な制限を理解する

カスタム セキュリティ プロバイダの実行時クラスの実装には、WebLogic Security フレームワークによって実行されるセキュリティ チェックを必要とするコードを含めないでください。セキュリティ プロバイダは、実際にセキュリティ チェックを行い WebLogic リソースへのアクセスを許可する WebLogic Security フレームワークのコンポーネントなので、そうしたコードを含めると無限反復が発生します。

「Provider」SSPI の目的を理解する

名前が「Provider」という接尾辞で終わる各 SSPI (たとえば `CredentialProvider`) は、セキュリティ プロバイダのサービスを WebLogic Security フレームワークに公開します。このことで、セキュリティ プロバイダの操作 (初期化、開始、停止など) が可能となります。

図 2-2　「Provider」SSPI

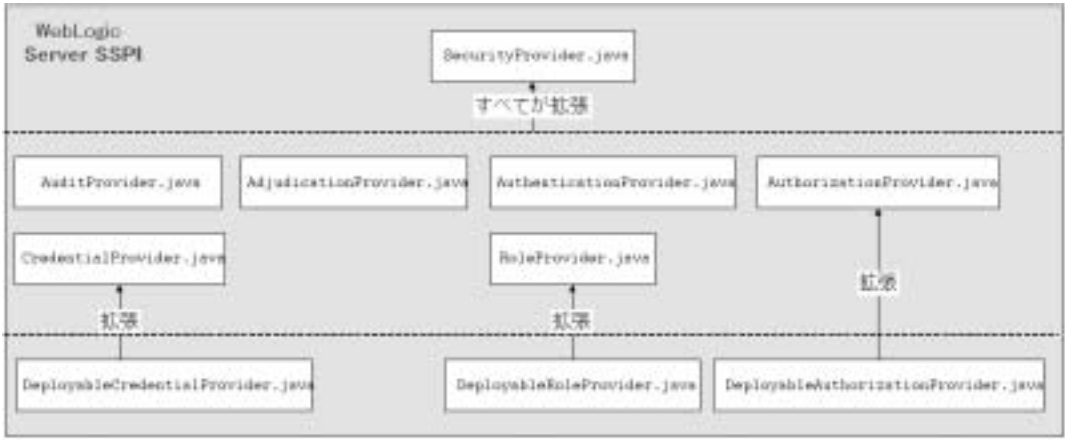


図 2-2 で示されているように、セキュリティ サービスを WebLogic Security フレームワークに公開する SSPI は WebLogic Server によって提供され、そのすべてが以下のメソッドを持つ `SecurityProvider` インタフェースを拡張します。

initialize

```
public void initialize(ProviderMBean providerMBean,
SecurityServices securityServices)
```

`initialize` メソッドは、セキュリティ プロバイダの関連付けられた MBean インスタンスに限定できる `ProviderMBean` を引数として取ります。その MBean インスタンスは生成する MBean タイプから作成され、カスタム セキュリティ プロバイダを WebLogic Server 環境で管理できるようにするコンフィグレーション データを含みます。このコンフィグレーション データがある場合は、`initialize` メソッドを使用してそれを抽出する必要があります。

`securityServices` 引数は、カスタム セキュリティ プロバイダで監査サービスを取得して使用する際の取得先となるオブジェクトです。監査サービスと監査の詳細については、第 9 章「監査プロバイダ」および第 10 章「カスタム セキュリティ プロバイダからのイベントの監査」を参照してください。

getDescription

```
public String getDescription()
```

このメソッドは、カスタム セキュリティ プロバイダについて簡単に説明したテキストを返します。

shutdown

```
public void shutdown()
```

このメソッドは、カスタム セキュリティ プロバイダを停止させます。

`SecurityProvider` を拡張するものなので、名前が「Provider」で終わる SSPI を実装する実行時クラスは継承したメソッドの実装を提供する必要があります。

実装する「Provider」インタフェースを決定する

名前が「Deployable」で始まり、「Provider」で終わる SSPI の実装 (`DeployableCredentialProvider` など) は、カスタム セキュリティ プロバイダのサービスを WebLogic Security フレームワークに公開します (2-4 ページの「「Provider」SSPI の目的を理解する」を参照)。ただし、それらの SSPI の実装では追加のタスクも実行されます。これらの SSPI は、サーブレット デプロイメント記述子 (`web.xml`、`weblogic.xml`)、EJB デプロイメント記述子 (`ejb-jar.xml`、`weblogic-ejb.jar.xml`)、EAR デプロイメント記述子 (`application.xml`、`weblogic-application.xml`) などのデプロイメント記述子でのセキュリティに対するサポートも提供します。

認可プロバイダ、資格マッピング プロバイダ、およびロール マッピング プロバイダには、デプロイ可能バージョンの「Provider」SSPI があります。

注意 : セキュリティ ポリシー、セキュリティ ロール、および資格のデータベースが読み込み専用の場合、認可セキュリティ プロバイダ、資格マッピング セキュリティ プロバイダ、およびロール マッピング セキュリティ プ

ロバイダ用のデプロイ可能ではない SSPI を実装します。ただし、その場合でも、デプロイメントを処理するセキュリティ プロバイダのデプロイ可能バージョンをコンフィグレーションする必要があります。

DeployableAuthorizationProvider SSPI

Web アプリケーションまたはエンタープライズ JavaBean (EJB) のデプロイメントに代わってセキュリティ ポリシーをデプロイできる認可プロバイダでは、`AuthorizationProvider` SSPI ではなく `DeployableAuthorizationProvider` SSPI を実装する必要があります。しかし、`DeployableAuthorizationProvider` SSPI は `AuthorizationProvider` SSPI の拡張なので、実際には両方の SSPI のメソッドを実装する必要があります。というのも、Web アプリケーションや EJB のデプロイメント作業では、認可プロバイダがセキュリティ ポリシーの作成や削除といった付加的な作業を実行する必要があるからです。セキュリティ レルムでは、少なくとも 1 つの認可プロバイダが `DeployableAuthorizationProvider` SSPI をサポートする必要があり、そうでなければ、Web アプリケーションや EJB をデプロイすることが不可能になります。

注意： セキュリティ ポリシーの詳細については、『WebLogic リソースのセキュリティ』の「セキュリティ ポリシー」を参照してください。

DeployableRoleProvider SSPI

Web アプリケーションまたはエンタープライズ JavaBean (EJB) のデプロイメントに代わってセキュリティ ロールをデプロイできるロール マッピング プロバイダでは、`RoleProvider` SSPI ではなく `DeployableRoleProvider` SSPI を実装する必要があります。しかし、`DeployableRoleProvider` SSPI は `RoleProvider` SSPI の拡張なので、実際には両方の SSPI のメソッドを実装する必要があります。というのも、Web アプリケーションや EJB のデプロイメント作業では、ロール マッピング プロバイダがセキュリティ ロールの作成や削除といった付加的な作業を実行する必要があるからです。セキュリティ レルムでは、少なくとも 1 つのロール マッピング プロバイダが この SSPI をサポートする必要があり、そうでなければ、Web アプリケーションや EJB をデプロイすることが不可能になります。

注意： セキュリティ ロールの詳細については、『WebLogic リソースのセキュリティ』の「セキュリティ ロール」を参照してください。

DeployableCredentialProvider SSPI

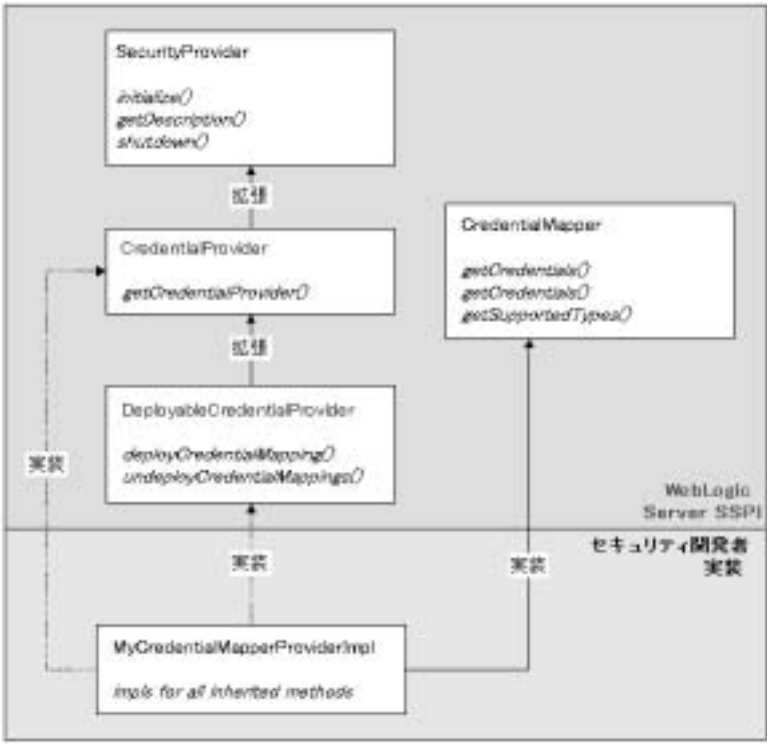
リソース アダプタ (RA) のデプロイメントに代わってセキュリティ ポリシーをデプロイできる資格マッピング プロバイダでは、`CredentialProvider` SSPI ではなく `DeployableCredentialProvider` SSPI を実装する必要があります。しかし、`DeployableCredentialProvider` SSPI は `CredentialProvider` SSPI の拡張なので、実際には両方の SSPI のメソッドを実装する必要があります。というのも、リソース アダプタのデプロイメント作業では、資格マッピング プロバイダが資格およびマッピングの作成や削除といった付加的な作業を実行する必要があるからです。セキュリティ レルムでは、少なくとも 1 つの資格マッピング プロバイダがこの SSPI をサポートする必要があり、そうでなければ、リソース アダプタをデプロイすることが不可能になります。

注意： 資格の詳細については、11-1 ページの「資格マッピングの概念」を参照してください。セキュリティ ポリシーの詳細については、『WebLogic リソースのセキュリティ』の「セキュリティ ポリシー」を参照してください。

SSPI 階層を理解し、実行時クラスを 1 つ作成するのか 2 つ作成するのかを決定する

図 2-3 は、資格マッピング プロバイダを使用してすべての SSPI に共通の継承階層を示すとともに、実行時クラスでどのようにそれらのインタフェースを実装できるのかを示します。この例で、BEA は `SecurityProvider` インタフェースと、`CredentialProvider`、`DeployableCredentialProvider`、および `CredentialMapper` SSPI を提供します。図 2-3 は、`DeployableCredentialProvider` SSPI および `CredentialMapper` SSPI を実装する `MyCredentialMapperProviderImpl` という 1 つの実行時クラスを示しています。

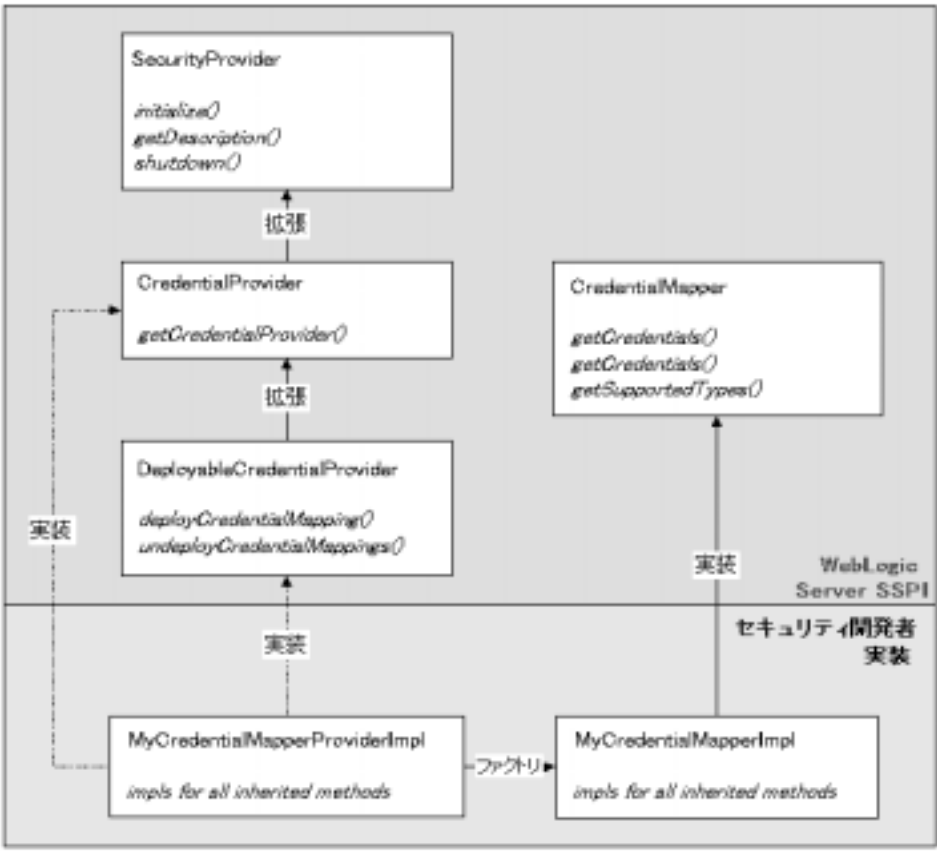
図 2-3　資格マッピング SSPI と 1 つの実行時クラス



ただし、図 2-3 は SSPI を実装できる 1 つの方法のみを示します (1 つの実行時クラスを作成する方法)。必要に応じて、図 2-4 で示されているように 2 つの実行時クラスを使用することもできます。2 つとは、名前が「Provider」で終わる SSPI (`CredentialProvider` や `DeployableCredentialProvider` など) の実装のための実行時クラスと、他の SSPI (`CredentialMapper` SSPI など) の実装のための実行時クラスです。

2 つの実行時クラスがある場合、名前が「Provider」で終わる SSPI を実装するクラスは他の SSPI を実装する実行時クラスを生成するためのファクトリとして機能します。たとえば図 2-4 では、`MyCredentialMapperProviderImpl` が `MyCredentialMapperImpl` を生成するためのファクトリとして機能します。

図 2-4　資格マッピング SSPI と 2 つの実行時クラス



注意： 実行時実装クラスを 2 つにする場合は、セキュリティ プロバイダの MBean タイプを生成するときに必ず両方の実行時実装クラスを MBean JAR ファイル (MJF) に格納する必要があります。詳細については、1-4 ページの「カスタム セキュリティ プロバイダをコンフィグレーションおよび管理する MBean タイプの生成」を参照してください。

SSPI クイック リファレンス

表 2-1 は、セキュリティ プロバイダのタイプ (およびそれらのコンポーネント) をそれらを開発するための SSPI およびその他のインタフェースと対応させて示しています。

表 2-1 セキュリティ プロバイダ、それらのコンポーネント、および対応する SSPI

タイプ / コンポーネント	SSPI/ インタフェース
認証プロバイダ	AuthenticationProvider
LoginModule (JAAS)	LoginModule
ID アサーション プロバイダ	AuthenticationProvider
ID アサータ	IdentityAsserter
プリンシパル検証プロバイダ	PrincipalValidator
認可	AuthorizationProvider DeployableAuthorizationProvider
アクセス決定	AccessDecision
裁決プロバイダ	AdjudicationProvider
裁決	Adjudicator
ロール マッピング プロバイダ	RoleProvider DeployableRoleProvider
ロール マッパー	RoleMapper
監査プロバイダ	AuditProvider
監査チャネル	AuditChannel
資格マッピング プロバイダ	CredentialProvider DeployableCredentialProvider
資格マッパー	CredentialMapper

注意： カスタム セキュリティ プロバイダの実行時クラスを作成するために使用する SSPI は、`weblogic.security.spi` パッケージに配置されます。このパッケージの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

セキュリティ サービス プロバイダ インタフェース (SSPI) MBean

1-2 ページの「開発プロセスの概要」で説明されているように、カスタム セキュリティ プロバイダ開発の 2 番目の手順ではカスタム セキュリティ プロバイダの MBean タイプを生成します。この節では、以下の助けになる情報を提供します。

- MBean タイプが必要な理由を理解する
- 拡張および実装する SSPI MBean を決定する
- MBean 定義ファイル (MDF) の基本的な要素を理解する
- SSPI MBean の階層と Administration Console に対する影響を理解する
- WebLogic MBeanMaker によって提供されるものを理解する

また、この節ではどの必須 SSPI MBean を拡張する必要があるのか、そしてどの任意 SSPI MBean を各タイプのセキュリティ プロバイダで実装できるのかを示す SSPI MBean クイック リファレンスも提供します。

MBean タイプが必要な理由を理解する

カスタム セキュリティ プロバイダの実行時クラスを作成することに加えて、MBean タイプを生成することも必要です。**MBean** という用語は管理対象 Bean (managed bean) の略で、JMX (Java Management eXtensions) で管理可能なリソースを表す Java オブジェクトのことです。

注意： JMX は Sun Microsystems によって策定された仕様で、標準的な管理アーキテクチャ、API、および管理サービスを定義しています。詳細については、「Java Management Extensions White Paper」を参照してください。

MBean タイプは、MBean のインスタンスのファクトリです。MBean のインスタンスは、WebLogic Server Administration Console を使用して作成できます。MBean のインスタンスが作成されたら、Administration Console でそのインスタンスを使用してカスタム セキュリティ プロバイダをコンフィグレーションおよび管理できます。

注意： すべての MBean インスタンスは自分の親タイプを知っているので、MBean タイプのコンフィグレーションを変更すると、Administration Console を使用して作成したすべてのインスタンスもコンフィグレーションが更新されます。詳細については、2-15 ページの「SSPI MBean の階層と Administration Console に対する影響を理解する」を参照してください。

拡張および実装する SSPI MBean を決定する

MBean タイプを作成するには **SSPI MBean** という MBean インタフェースを使用します。カスタム セキュリティ プロバイダ用の MBean タイプを作成するのに利用できる SSPI MBean には、以下の 2 つのタイプがあります。

- **必須 SSPI MBean**。セキュリティ プロバイダを WebLogic Server 環境内でコンフィグレーションおよび管理できるようにするための基本的なメソッドが定義されるので拡張する必要があります。
- **任意 SSPI MBean**。セキュリティ プロバイダを管理するための追加メソッドが定義されるので実装することができます。セキュリティ プロバイダのタイプが異なれば、利用できる任意 SSPI MBean も異なります。

詳細については、2-20 ページの「SSPI MBean クイック リファレンス」を参照してください。

MBean 定義ファイル (MDF) の基本的な要素を理解する

MBean 定義ファイル (MDF) は、WebLogic MBeanMaker ユーティリティで、MBean タイプを構成する Java ファイルを生成するために使用される XML ファイルです。すべての MDF では、作成済みのセキュリティ プロバイダのタイプに固有の必須 SSPI MBean を拡張する必要があり、さらに任意 SSPI MBean を実装することができます。

コード リスト 2-1 は、サンプルの MBean 定義ファイル (MDF) を示しています。その内容の説明については、リストの下を参照してください。具体的には、WebLogic 資格マッピング プロバイダの MBean タイプを生成するために使用する MDF です。

注意： MDF 要素の構文についての完全なリファレンスは、付録 A「MBean 定義ファイル (MDF) 要素の構文」に収められています。

コード リスト 2-1　DefaultCredentialMapper.xml

```
<?xml version="1.0"?>
<!DOCTYPE MBeanType SYSTEM "commo.dtd">

<MBeanType
 Name = "DefaultCredentialMapper"
 DisplayName = "DefaultCredentialMapper"
 Package = "weblogic.security.providers.credentials"
 Extends = "weblogic.management.security.credentials.
DeployableCredentialMapper"
 Implements = "weblogic.management.security.credentials.
UserPasswordCredentialMapEditor"
 PersistPolicy = "OnUpdate"
 Description = "This MBean represents configuration attributes for
the WebLogic Credential Mapping provider.&lt;p&gt;"
>

<MBeanAttribute
 Name = "ProviderClassName"
 Type = "java.lang.String"
 Writeable = "false"
 Default = "&quot;weblogic.security.providers.credentials.
DefaultCredentialMapperProviderImpl&quot;"
 Description = "The name of the Java class that loads the WebLogic
Credential Mapping provider."
/>
```

```
<MBeanAttribute
 Name = "Description"
 Type = "java.lang.String"
 Writeable = "false"
 Default = "&quot;Provider that performs Default Credential
Mapping&quot;"
 Description = "A short description of the WebLogic Credential
Mapping provider."
/>

<MBeanAttribute
 Name = "Version"
 Type = "java.lang.String"
 Writeable = "false"
 Default = "&quot;1.0&quot;"
 Description = "The version of the WebLogic Credential Mapping
provider."
/>

</MBeanType>
```

<MBeanType> タグ内の太字の属性は、この MDF が DefaultCredentialMapper という名前であり、DeployableCredentialMapper という必須 SSPI MBean を拡張するものであることを示しています。また、この MDF では、UserPasswordCredentialMapEditor 任意 SSPI MBean を実装することで付加的な管理機能も組み込んでいます。

<MBeanAttribute> タグで定義される ProviderClassName、Description、および Version の各属性は、セキュリティ プロバイダの基本的なコンフィグレーション方法を定義するものであり、Provider という基本必須 SSPI MBean から継承されるものなので、セキュリティ プロバイダの MBean タイプを生成するために使用されるどのような MDF でも必須です 図 2-6 を参照)。ProviderClassName 属性は特に重要です。ProviderClassName 属性の値は、セキュリティ プロバイダの実行時クラスの Java ファイル名 (すなわち、「Provider」という文字列で終わる適切な SSPI の実装) です。コード リスト 2-1 で示されているサンプルの実行時クラスは、DefaultCredentialMapperProviderImpl.java です。

コード リスト 2-1 では示されていませんが、<MBeanAttribute> タグおよび <MBeanOperation> タグを使用して MDF に属性と操作を追加することができます。ほとんどのカスタム属性は、WebLogic Administration Console のカスタム セキュリティ プロバイダの [詳細] タブに自動的に表示されます。例については

図 2-5 を参照してください。ただし、カスタム操作を表示するには、コンソール拡張機能を記述する必要があります。1-6 ページの「コンソール拡張の記述」を参照してください。

図 2-5 [詳細] タブの例

注意： サンプル監査プロバイダ (dev2dev Web サイトの「Code Samples: WebLogic Server」で入手可能) では、カスタム属性を追加する例が提供されています。

SSPI MBean の階層と Administration Console に対する影響を理解する

MBean 定義ファイル (MDF) で拡張する必須 SSPI MBean (基本 SSPI MBean である `Provider` に至るまで) に指定されている属性と操作はすべて、関連するセキュリティ プロバイダの WebLogic Server Administration Console ページに自動的に表示されます。これらの属性と操作は、カスタム セキュリティ プロバイダをコンフィグレーションおよび管理するために使用します。

注意： 認証セキュリティ プロバイダに限っては、MDF で実装される任意 SSPI MBean で指定された属性と操作も Administration Console で自動的にサポートされます。他のタイプのセキュリティ プロバイダの場合、任意 SSPI MBean から継承された属性 / 操作を Administration Console で使用するには、コンソール拡張機能を記述する必要があります。詳細については、1-6 ページの「コンソール拡張の記述」を参照してください。

図 2-6 は、WebLogic 資格マッピング MDF を例に使用してセキュリティ プロバイダの SSPI MBean 階層を示すとともに、どの属性 / 操作が WebLogic 資格マッピング プロバイダの Administration Console で表示されるのかを示します。

図 2-6　資格マッピング プロバイダの SSPI MBean 階層

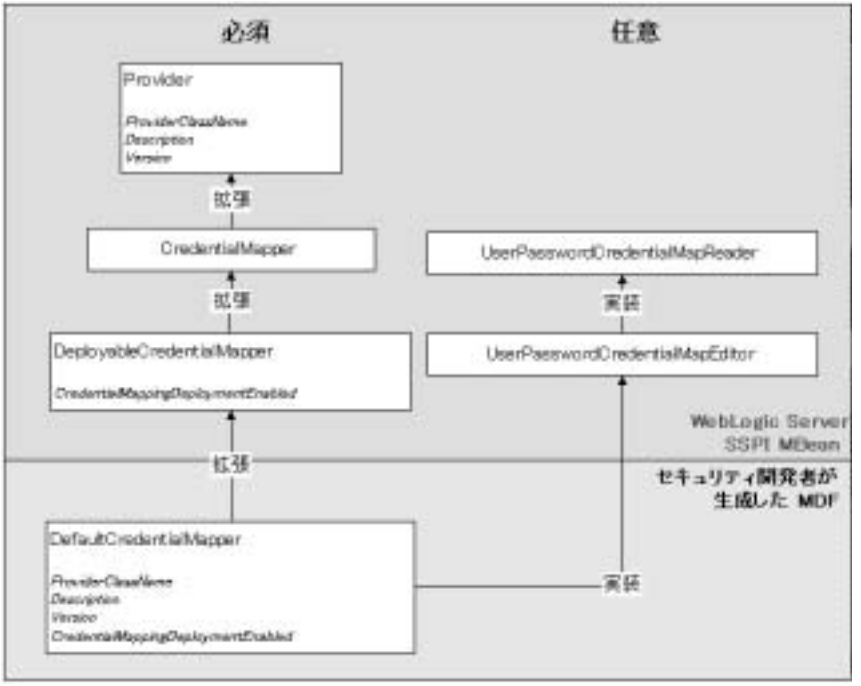


図 2-6 で示されているように、`DefaultCredentialMapper` MDF の SSPI MBean の階層を実装することにより、図 2-7 のページが Administration Console で表示されます。`DefaultCredentialMapper` MDF のリストはコード リスト 2-1 に示してあります。

図 2-7 DefaultCredentialMapper の場合の Administration Console ページ

[名前]、[記述]、および [バージョン] の各フィールドは、`Provider` という基本必須 SSPI MBean から継承され `DefaultCredentialMapper` MDF に指定されている同名の属性から得られたものです。`DefaultCredentialMapper` MDF 内の `DisplayName` 属性から [名前] フィールドの値が生成されるほか、`Description` と `Version` の各属性からも、それぞれ対応するフィールドの値が生成されることに注意してください。なお、[資格マッピング デプロイメントを有効化] フィールドが表示されているのは、`DefaultCredentialMapper` MDF の拡張元である `DeployableCredentialMapper` 必須 SSPI MBean に `CredentialMappingDeploymentEnabled` 属性が定義されているからです。この Administration Console ページには、`UserPasswordCredentialMapEditor` 任意 SSPI MBean の `DefaultCredentialMapper` MDF の実装に対応するフィールドは表示されません。

WebLogic MBeanMaker によって提供されるものを理解する

WebLogic MBeanMaker は、MBean 定義ファイル (MDF) を入力とし、MBean タイプのファイルを出力するコマンドライン ユーティリティです。WebLogic MBeanMaker を使用して作成した MDF を実行すると、以下のことが行われます。

- 必須 SSPI MBean から継承された属性 (MDF に追加されたカスタム属性も同様) により、WebLogic MBeanMaker によって MBean タイプの情報ファイルに完全なゲッター / セッター メソッドが生成される (MBean 情報ファイルは

図 2-8 には示されていない)。MBean 情報ファイルの詳細については、2-19 ページの「MBean 情報ファイルについて」を参照してください。

開発者の必要なアクション : なし。これらのメソッドでさらなる作業は必要ありません。

- 任意 SSPI MBean から継承された操作により、MBean の実装ファイルでメソッドが継承される (メソッドの実装は一から記述しなければならない)

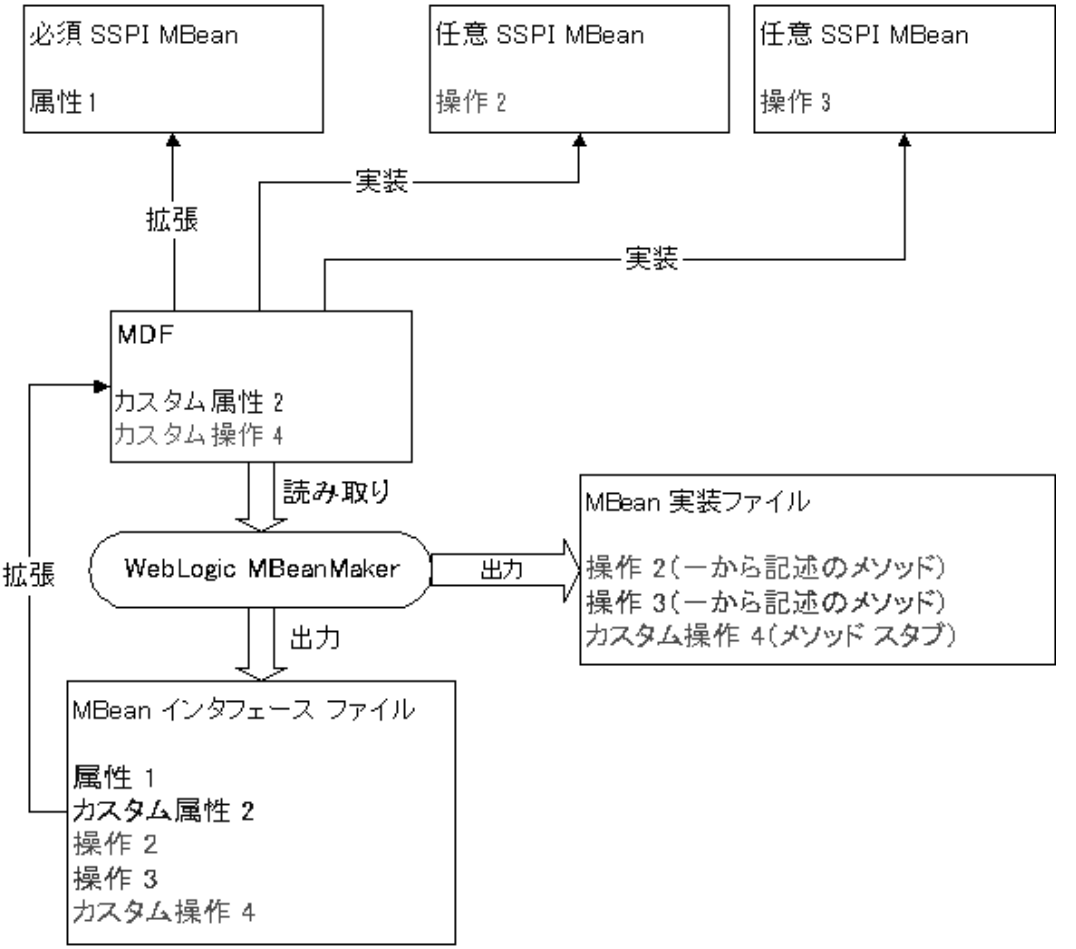
  開発者の必要なアクション : 現時点では、WebLogic MBeanMaker では継承されたメソッドのメソッド スタブが生成されません。したがって、「Mapping MDF Operation Declarations to Java Method Signatures Document」(dev2dev Web サイトの「Code Samples: WebLogic Server」で入手可能) を利用して適切な実装を提供する必要があります。

- MDF に追加されたカスタム操作により、WebLogic MBeanMaker によってメソッド スタブが生成される

  開発者の必要なアクション : メソッドの実装を提供する必要があります。ただし、WebLogic MBeanMaker でスタブが生成されるので、Java メソッドのシグネチャを参照する必要はありません。

このことを 図 2-8 に示します。

図 2-8　WebLogic MBeanMaker によって提供されるもの



MBean 情報ファイルについて

MBean 情報ファイルには、MBean 定義ファイル内のコンパイルされたデータ定義が JMX Model MBean で要求される形式で含まれます。このファイルのフォーマットは属性、操作、および通知のリストであり、エンティティごとにそのエンティティを記述する記述子タグのセットがあります。さらに MBean 自体にも記述子タグのセットが付きます。このフォーマットの例は次のとおりです。

```
MBean + タグ

attribute1 + タグ , attribute2 + タグ ...

operation1 + タグ , operation2 + タグ ...

notification1 + タグ , notification2 + タグ ...
```

必要に応じて、標準の JMX サーバの `getMBeanInfo` メソッドを呼び出して `ModelMBeanInfo` を取得することで、実行時にこの情報にアクセスできます。

注意： JMX 仕様を参照して、返される構造の解釈方法を確認してください。

SSPI MBean クイック リファレンス

カスタム セキュリティ プロバイダの開発の一部として実装する必要のある SSPI のリストに基づいて、拡張する必要のある必須 SSPI MBean を 表 2-2 で特定してください。また、表 2-3 から 表 2-5 を使用して、セキュリティ プロバイダを管理するために実装する必要のある任意 SSPI MBean を特定してください。

表 2-2 必須 SSPI MBean

タイプ	パッケージ名	必須 SSPI MBean
認証プロバイダ	`authentication`	`Authenticator`
ID アサーション プロバイダ	`authentication`	`IdentityAsserter`
認可プロバイダ	`authorization`	`Authorizer` または `DeployableAuthorizer`
裁決プロバイダ	`authorization`	`Adjudicator`
ロール マッピング プロバイダ	`authorization`	`RoleMapper` または `DeployableRoleMapper`
監査プロバイダ	`audit`	`Auditor`
資格マッピング プロバイダ	`credentials`	`CredentialMapper` または `DeployableCredentialMapper`

注意： 表 2-2 に示した必須 SSPI MBean は、`weblogic.management.security.<` パッケージ名 `>` というパッケージに定義されています。

表 2-3 認証用任意 SSPI MBean

任意 SSPI MBean	用途
`GroupEditor`	グループを作成する。当該グループが既に存在する場合には、例外が送出される
`GroupMemberLister`	グループのメンバーを列挙する
`GroupReader`	グループに関するデータを読み込む
`GroupRemover`	グループを削除する
`MemberGroupLister`	あるユーザまたはグループが所属するグループを列挙する
`UserEditor`	ユーザを作成、編集、および削除する
`UserPasswordEditor`	ユーザのパスワードを変更する
`UserReader`	ユーザに関するデータを読み込む
`UserRemover`	ユーザを削除する

注意： 表 2-3 に示す認証用任意 SSPI MBean は、`weblogic.management.security.authentication` パッケージに定義されています。また、これらは WebLogic Server Administration Console でもサポートされます。

表 2-3 で示されている認証用任意 SSPI MBean の実装方法を示す例については、Manageable Sample Authentication Provider (dev2dev Web サイトの「Code Samples: WebLogic Server」で入手可能) のコードを参照してください。

表 2-4 認可用任意 SSPI MBean

任意 SSPI MBean	用途
PolicyEditor	セキュリティ ポリシーを作成、編集、および削除する
PolicyReader	セキュリティ ポリシーに関するデータを読み込む
RoleEditor	セキュリティ ロールを作成、編集、および削除する
RoleReader	セキュリティ ロールに関するデータを読み込む

注意： 表 2-4 に示す認可用任意 SSPI MBean は、weblogic.management.security.authorization パッケージに定義されています。

表 2-5 資格マッピング用任意 SSPI MBean

任意 SSPI MBean	用途
UserPasswordCredentialMapEditor	WebLogic ユーザをリモートのユーザ名およびパスワードにマップする資格マップを編集する
UserPasswordCredentialMapReader	WebLogic ユーザをリモートのユーザ名およびパスワードにマップする資格マップを読み込む

注意： 表 2-5 に示す資格マッピング用任意 SSPI MBean は、weblogic.management.security.credentials パッケージに定義されています。

セキュリティ プロバイダの開発者向け管理ユーティリティ

weblogic.management.utils パッケージには、開発者にとって特にカスタム セキュリティ プロバイダの MBean タイプを生成する際に役立つ管理インタフェースと例外が含まれています。これらのインタフェースと例外の実装には、カスタム セキュリティ プロバイダの MDF で任意 SSPI MBean を実装してこれらを継承する場合を除いて、カスタム セキュリティ プロバイダを開発する必要がありません。

注意： このインタフェースとクラスは、一般的な用途のユーティリティ、つまり非セキュリティ MBeean にも使用できるものなので、weblogic.management.security ではなくこのパッケージに入っています。MBean のさまざまなタイプについては、『WebLogic JMX Service プログラマーズ ガイド』の「WebLogic Server 管理対象リソースと MBean」を参照してください。

weblogic.management.utils パッケージには、以下のユーティリティが含まれています。

- 一般例外
- 大きなデータ リストを扱うためのメソッドを提供するインタフェース
- 外部 LDAP サーバと通信するために必要なコンフィグレーション属性を含むインタフェース

注意： Manageable Sample Authentication Provider (dev2dev Web サイトの「Code Samples: WebLogic Server」から入手可能なサンプル セキュリティ プロバイダ) は、データ リストを扱うためだけでなく、例外用にも weblogic.management.utils パッケージを使用します。

詳細については、WebLogic Server 7.0 API リファレンス Javadoc の weblogic.management.utils パッケージを参照してください。

セキュリティ プロバイダと WebLogic リソース

WebLogic リソースは、権限のないアクセスから保護することができる WebLogic Server エンティティを表す構造化オブジェクトです。カスタムの認可プロバイダ、ロールマッピング プロバイダ、および資格マッピング プロバイダの開発者は、これらのセキュリティ プロバイダが、WebLogic リソースおよびそれらのリソースのセキュリティに使用されるセキュリティ ポリシーとどのように対話するかを理解しておく必要があります。

注意： セキュリティ ポリシーは、以前のリリースの WebLogic Server で WebLogic リソースを保護するために使用していたアクセス制御リスト (ACL) とパーミッションに代わるものです。

以下の節では、セキュリティ プロバイダと WebLogic リソースについて説明します。

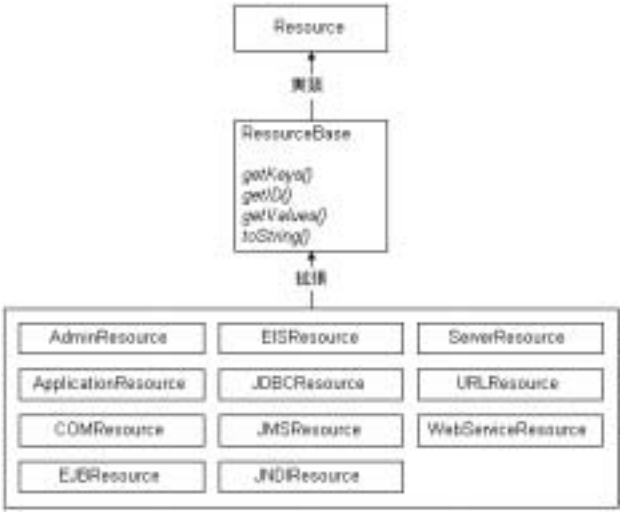
- 2-25 ページの「WebLogic リソースのアーキテクチャ」
- 2-26 ページの「WebLogic リソースのタイプ」
- 2-26 ページの「WebLogic リソース識別子」
- 2-28 ページの「WebLogic リソースのデフォルト グループの作成」
- 2-29 ページの「WebLogic リソースのデフォルト セキュリティ ロールの作成」
- 2-30 ページの「WebLogic リソースのデフォルト セキュリティ ポリシーの作成」
- 2-32 ページの「単一親リソース階層」
- 2-35 ページの「ContextHandler と WebLogic リソース」

注意： 詳細については、『WebLogic リソースのセキュリティ』を参照してください。

WebLogic リソースのアーキテクチャ

weblogic.security.spi パッケージに定義されている Resource インタフェースは、権限のないアクセスから保護できる WebLogic リソースを表すオブジェクトを定義します。weblogic.security.service パッケージに定義されている ResourceBase クラスは、より詳細な WebLogic リソース タイプの抽象基底クラスで、リソースを拡張するためのモデルです。詳細については、図 2-9 と 2-26 ページの「WebLogic リソースのタイプ」を参照してください。

図 2-9 WebLogic リソースのアーキテクチャ



ResourceBase クラスには、BEA が提供する getID、getKeys、getValues、および toString メソッドの実装が含まれています。詳細については、WebLogic Server 7.0 API リファレンス Javadoc の ResourceBase クラスを参照してください。

このアーキテクチャにより、特定の WebLogic リソースを意識することなくセキュリティ プロバイダを開発できます。このため、新しいリソース タイプが追加されても、セキュリティ プロバイダを修正する必要がありません。

WebLogic リソースのタイプ

図 2-9 に示したように、`weblogic.security.service` パッケージの特定のクラスは `ResourceBase` クラスの拡張であるため、WebLogic リソースの特定のタイプの実装を提供します。以下の WebLogic リソースの実装を使用できます。

- 管理リソース
- アプリケーション リソース
- COM リソース
- EIS リソース
- EJB リソース
- JDBC リソース
- JMS リソース
- JNDI リソース
- サーバ リソース
- URL リソース
- Web サービス リソース

注意： これらの各 WebLogic リソースの詳細については、『WebLogic リソースのセキュリティ』および WebLogic Server 7.0 API リファレンス Javadoc の weblogic.security.service パッケージを参照してください。

WebLogic リソース識別子

各 WebLogic リソース (2-26 ページの「WebLogic リソースのタイプ」を参照) は、その `toString()` 表現か、または `getID` メソッドを使用して取得される ID によって識別できます。

toString() メソッド

これらの WebLogic リソースの実装の toString() メソッドを使用する場合、その WebLogic リソースの記述が String 形式で返されます。最初に、WebLogic リソースのタイプが不等号括弧内に出力されます。次に、各キーとその値が順番に出力されます。これらのキーはカンマで区切られます。示される値はカンマで区切られ、中括弧で囲まれます。各値はそのまま出力されますが、カンマ (,)、開き中括弧 ({)、閉じ中括弧 (})、およびバック スラッシュ (\) はそれぞれバック スラッシュでエスケープされます。たとえば、次のような EJB リソースがあるとします。

```
EJBResource ("myApp",
             "MyJarFile",
             "myEJB",
             "myMethod",
             "Home",
             new String[ ] {"argumentType1", "argumentType2"}
            );
```

このリソースは、以下の toString 出力を生成します。

```
type=<ejb>, app=myApp, module=”MyJarFile”, ejb=myEJB,
method="myMethod", methodInterface="Home",
methodParams={argumentType1, argumentType2}
```

toString() メソッドによって提供される WebLogic リソース記述の形式は公開されており (つまり Resource オブジェクトを使用せずに記述を作成できる)、逆変換できます (つまりこの String 形式を元の WebLogic リソースに戻すことができます)。

注意： コード リスト 2-2 に、toString() メソッドを使用して WebLogic リソースを識別する方法を示します。

リソース ID と getID() メソッド

定義済みの各 WebLogic リソース タイプに定義されている getID() メソッドは、セキュリティ プロバイダで WebLogic リソースをユニークに識別するために使用できる 64 ビット ハッシュコードを返します。リソース ID は、以下のアルゴリズムを使用して、高速の実行時キャッシングに効果的に使用できます。

1. WebLogic リソースを取得します。
2. `getID()` メソッドを使用して WebLogic リソースのリソース ID を取得します。
3. キャッシュ内でそのリソース ID をルックアップします。
4. リソース ID が見つかれば、それに対応するセキュリティ ポリシーを返します。
5. リソース ID が見つからなければ、以下を実行します。
   a. `toString()` メソッドを使用して、セキュリティ プロバイダ データベース内で WebLogic リソースをルックアップします。
   b. リソース ID とセキュリティ ポリシーをキャッシュに格納します。
   c. セキュリティ ポリシーを返します。

注意： コード リスト 2-3 に、認可プロバイダで `getID()` メソッドを使用して WebLogic リソースを識別する方法と、そのアルゴリズムの実装例を示します。

このメソッドは何度実行しても同じ値を返すという保証はないので、このリソース ID を使用してセキュリティ プロバイダ データベースに WebLogic リソースに関する情報を格納してはいけません。代わりに、WebLogic リソースの `toString()` メソッドを使用して、リソース対セキュリティ ポリシーおよびリソース対セキュリティ ロール マッピングを、対応するセキュリティ プロバイダ データベースに格納するようにしてください。

注意： セキュリティ プロバイダ データベースの詳細については、2-37 ページの「セキュリティ プロバイダ データベースの初期化」を参照してください。`toString()` メソッドの詳細については 2-27 ページの「toString() メソッド」を参照してください。

WebLogic リソースのデフォルト グループの作成

カスタム認証プロバイダの実行時クラスを記述する場合は、デフォルト グループをいくつか作成しておく必要があります。表 2-6 は、このタスクを行う場合に役立つ情報を示しています。

表 2-6 デフォルト グループとグループ メンバーシップ

グループ名	グループ メンバーシップ
Administrators	空、または管理ユーザ
Deployers	空
Monitors	空
Operators	空

WebLogic リソースのデフォルト セキュリティ ロールの作成

カスタム ロール マッピング プロバイダの実行時クラスを記述する場合は、デフォルト グローバル ロールをいくつか作成しておく必要があります。表 2-7 は、このタスクを行う場合に役立つ情報を示しています。

表 2-7 デフォルト グローバル ロールとグループの関連付け

グローバル ロール名	グループの関連付け
Admin	Administrators グループ
Anonymous	weblogic.security.WLSPrincipals.getEveryoneGroupname() グループ
Deployer	Deployers グループ
Monitor	Monitors グループ
Operator	Operators グループ

注意： グローバル ロールとスコープ ロールの詳細については、『WebLogic リソースのセキュリティ』の「セキュリティ ロール」を参照してください。

WebLogic リソースのデフォルト セキュリティ ポリシーの作成

カスタム認可プロバイダの実行時クラスを記述する場合は、デフォルト セキュリティ ポリシーをいくつか作成しておく必要があります。これらのデフォルト セキュリティ ポリシーは、初期状態でさまざまなタイプの WebLogic リソースを保護します。表 2-8 は、このタスクを行う場合に役立つ情報を示しています。

表 2-8 WebLogic リソースのデフォルト セキュリティ ポリシー

WebLogic リソース コンストラクタ	セキュリティ ポリシー
`new AdminResource(null, null, null)`	`Admin` グローバル ロール
`new AdminResource("Configuration", null, null)`	`Admin`、`Deployer`、`Monitor`、または `Operator` グローバル ロール
`new AdminResource("FileUpload", null, null)`	`Admin` または `Deployer` グローバル ロール
`new EISResource(null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` グループ
`new EJBResource(null, null, null, null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` グループ
`new JDBCResource(null, null, null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` グループ
`new JNDIResource(null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` グループ
`new JMSResource(null, null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` グループ
`new ServerResource(null, null, null)`	`Admin` または `Operator` グローバル ロール

表 2-8 WebLogic リソースのデフォルト セキュリティ ポリシー (続き)

WebLogic リソース コンストラクタ	セキュリティ ポリシー
`new URLResource(null, null, null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` グループ
`new WebServiceResource(null, null, null, null)`	`weblogic.security.WLSPrincipals.getEveryoneGroupname()` グループ

注意 : アプリケーションおよび COM リソースにはデフォルト セキュリティ ポリシーを設定しないでください (つまり、デフォルトですべてのユーザにパーミッションを付与しないでください)。

セキュリティ プロバイダの実行時クラスでの WebLogic リソースのルックアップ

コード リスト 2-2 に、認可プロバイダの実行時クラスで WebLogic リソースをルックアップする方法を示します。このアルゴリズムは、認可プロバイダのセキュリティ プロバイダ データベースに WebLogic リソースとセキュリティ ポリシーのマッピングが格納されていることを前提にしています。コード リスト 2-2 に示すアルゴリズムと `getParentResource` メソッドの使用は必須ではありません。`getParentResource` メソッドの詳細については 2-32 ページの「単一親リソース階層」を参照してください。

コード リスト 2-2 認可プロバイダで WebLogic リソースをルックアップする方法 : toString メソッドの使用

```
Policy findPolicy(Resource resource) {
   Resource myResource = resource;
   while (myResource != null) {
      String resourceText = myResource.toString();
      Policy policy = lookupInDB(resourceText);
      if (policy != null) return policy;
      myResource = myResource.getParentResource();
   }
   return null;
}
```

getID メソッドを使用することで、WebLogic リソースのルックアップ アルゴリズムを最適化できます。コード リスト 2-2 に示すように toString メソッドを使用すると文字列の連結のためにパフォーマンスが低下する場合があります。getID メソッドは WebLogic リソース自身の内部で計算およびキャッシュされるハッシュ処理なのでより高速で効率的です。このため、getID メソッドを使用する場合、toString 値の計算はリソースにつき一度しか必要ありません。コード リスト 2-3 を参照してください。

コード リスト 2-3　認可プロバイダで WebLogic リソースをルックアップする方法 : getID メソッドの使用

```
Policy findPolicy(Resource resource) {
   Resource myResource = resource;
   while (myResource != null) {
      long id = myResource.getID();
      Policy policy = lookupInCache(id);
      if (policy != null) return policy;
      String resourceText = myResource.toString();
      Policy policy = lookupInDB(resourceText);
      if (policy != null) {
         addToCache(id, policy);
         return policy;
      }
      myResource = myResource.getParentResource();
   }
   return null;
}
```

注意 : getID メソッドは、サービス パック間または将来の WebLogic Server リリース間で保証されません。このため、getID 値はセキュリティ プロバイダ データベースに格納しないでください。

単一親リソース階層

WebLogic リソースの粒度は自由に設定できます。たとえば、Web アプリケーション全体、その Web アプリケーション内の特定のエンタープライズ JavaBean (EJB)、あるいはその EJB 内の単一のメソッドを WebLogic リソースと見なすことができます。

WebLogic リソースは、特殊性が最も高いものから最も低いものまでの階層構造に整理されます。各 WebLogic リソース タイプでは、getParentResource メソッドを使用できますが、必須ではありません。

WebLogic セキュリティ プロバイダは、次のように単一親リソース階層を使用します。WebLogic セキュリティ プロバイダが特定の WebLogic リソースにアクセスしようとして、そのリソースが見つからない場合、WebLogic セキュリティ プロバイダはそのリソースの getParentResource メソッドを呼び出します。このメソッドによって現在の WebLogic リソースの親リソースが返され、WebLogic セキュリティ プロバイダはリソース階層を上って次の (特殊性のより低い) リソースを保護できるようになります。たとえば、呼び出し側が以下の URL リソースにアクセスしようとしたとします。

```
type=<url>, application=myApp, contextPath=”/mywebapp”,
uri=foo/bar/my.jsp
```

この URL リソースが見つからない場合、WebLogic セキュリティ プロバイダは以下のリソースを検索して保護しようとします。順序は以下のとおりです。

```
type=<url>, application=myApp, contextPath="/mywebapp", uri=/foo/bar/*
type=<url>, application=myApp, contextPath="/mywebapp", uri=/foo/*
type=<url>, application=myApp, contextPath="/mywebapp", uri=*.jsp
type=<url>, application=myApp, contextPath="/mywebapp", uri=/*
type=<url>, application=myApp, contextPath="/mywebapp"
type=<url>, application=myApp
type=<app>, application=myApp
type=<url>
```

注意： getParentResource メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc で任意の定義済み WebLogic リソース タイプまたは Resource インタフェースを参照してください。

URL リソースのパターン マッチング

Java サーブレット仕様 2.3 のセクション SRV.11.1 および SRV.11.2 では、サーブレット コンテナのパターン マッチング ルールが説明されています。そのルールは、URL リソースにも使用します。以下の例は、URL リソースのパターン マッチングに関する重要な概念を示しています。

例 1

URL リソース `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/my.jsp, httpMethod=GET` では、次のようなリソース階層が使用されます。行 3 と行 4 に注目してください。URL パターンが、予想とは異なるかもしれません。

1. `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/my.jsp, httpMethod=GET`
2. `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/my.jsp`
3. `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/my.jsp/*, httpMethod=GET`
4. `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/my.jsp/*`
5. `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/*, httpMethod=GET`
6. `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/*`
7. `type=<url>, application=myApp, contextPath=/mywebapp, uri=*.jsp, httpMethod=GET`
8. `type=<url>, application=myApp, contextPath=/mywebapp, uri=*.jsp`
9. `type=<url>, application=myApp, contextPath=/mywebapp, uri=/*, httpMethod=GET`
10. `type=<url>, application=myApp, contextPath=/mywebapp, uri=/*`
11. `type=<url>, application=myApp, contextPath=/mywebapptype=<url>, application=myApp`
12. `type=<app>, application=myApp`
13. `type=<url>`

例 2

URL リソース `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo` では、次のようなリソース階層が使用されます。行 2 に注目してください。URL パターンが、予想とは異なるかもしれません。

1. `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo`
2. `type=<url>, application=myApp, contextPath=/mywebapp, uri=/foo/*`
3. `type=<url>, application=myApp, contextPath=/mywebapp, uri=/*`
4. `type=<url>, application=myApp, contextPath=/mywebapp`
5. `type=<url>, application=myApp`
6. `type=<app>, application=myApp`
7. `type=<url>`

ContextHandler と WebLogic リソース

ContextHandler は、リソース コンテナから追加のコンテキスト情報およびコンテナ固有の情報を取得し、その情報をアクセス決定やロール マッピング決定を行うセキュリティ プロバイダに渡す高性能な WebLogic クラスです。`ContextHandler` インタフェースを使用すると、内部 WebLogic リソース コンテナが WebLogic Security フレームワーク呼び出しに追加情報を渡すことができるようになります。その結果、セキュリティ プロバイダは、特定のメソッドの引数で提供される以上のコンテキスト情報を取得できるようになります。ContextHandler は本質的には名前と値のリストなので、セキュリティ プロバイダは検索する名前を認識しておく必要があります。つまり、ContextHandler の使用には、WebLogic リソース コンテナとセキュリティ プロバイダの間の緊密な連携が不可欠です。ContextHandler の名前と値の各組み合わせは、**コンテキスト要素**と呼ばれ、`ContextElement` オブジェクトによって表されます。

注意： `ContextHandler` インタフェースおよび `ContextElement` クラスの詳細については、WebLogic Server 7.0 API リファレンス Javadoc の weblogic.security.service パッケージを参照してください。

現在、2 種類の WebLogic リソース コンテナ (サーブレット コンテナと EJB コンテナ) が ContextHandler を WebLogic Security フレームワークに渡します。そのため、URL (Web) および EJB のリソース タイプには、異なるコンテキスト要素が含まれます。これらのコンテキスト要素の値は、カスタム認可プロバイダまたはカスタム ロール マッピング プロバイダの開発の一部として調べることができます。表 2-9 および 表 2-10 は、URL リソースおよび EJB リソースの ContextHandler の各コンテキスト要素を示しています。

表 2-9 URL (Web) リソースの ContextHandler

コンテキスト要素名	コンテキスト要素値
`HttpServletRequest`	`javax.servlet.http.HttpServletRequest`
`HttpServletResponse`	`javax.servlet.http.HttpServletResponse`

表 2-10 エンタープライズ JavaBean (EJB) リソースの ContextHandler

コンテキスト要素名	コンテキスト要素値
Parameter1	EJB の `ejb-jar.xml` デプロイメント記述子の `<method-param>` 要素を介して、各パラメータのオブジェクト タイプとセマンティクスを判別する。
Parameter2 ...	
ParameterN	

コード リスト 2-4 では、URL (Web) リソースの ContextHandler を介して、`HttpServletRequest` および `HttpServletResponse` コンテキスト要素オブジェクトにアクセスする方法を示します。たとえば、このコードを `AccessDecision` SSPI 実装の `isAccessAllowed()` メソッドで使用できます。詳細については、6-8 ページの「AccessDecision SSPI を実装する」を参照してください。

コード リスト 2-4 例 : URL リソースの ContextHandler でのコンテキスト要素へのアクセス

```
static final String SERVLETREQUESTNAME = "HttpServletRequest";

if (resource instanceof URLResource) {
   HttpServletRequest req =
```

```
        (HttpServletRequest)handler.getValue(SERVLETREQUESTNAME);
}
```

注意： RoleMapper SSPI 実装の getRoles() メソッドや、AuditContext インタフェース実装の getContext() メソッドでこれらのコンテキスト要素にアクセスすることもできます。詳細については、それぞれ 8-10 ページの「RoleMapper SSPI を実装する」と 10-9 ページの「監査コンテキスト」を参照してください。

セキュリティ プロバイダ データベースの初期化

注意： この節を読む前に、『WebLogic Security の紹介』の「セキュリティ プロバイダ データベース」に目を通しておいてください。

セキュリティ プロバイダのデータベースは、少なくとも、認証、認可、ロール マッピング、および資格マッピングの各プロバイダで必要なデフォルトのユーザ、グループ、セキュリティ ポリシー、セキュリティ ロール、または資格で初期化する必要があります。セキュリティ プロバイダのデータベースを初期化してから、セキュリティ プロバイダを使用してください。また、カスタム セキュリティ プロバイダ用の実行時クラスを記述する場合にこのデータベースがどのように機能するかについても考えておく必要があります。セキュリティ プロバイダのデータベースを初期化する方法は、数多くの要因によって決まります。要因としては、外部で管理されるデータベースがユーザ、グループ、セキュリティ ポリシー、セキュリティ ロール、資格の情報の格納に使用されるかどうか、データベースが既に存在しているかどうか (作成する必要があるかどうか)、などがあります。

以下の節では、セキュリティ プロバイダ データベースを初期化するためのベスト プラクティスについて説明します。

- ベスト プラクティス : データベースがない場合のシンプルなデータベースの作成
- ベスト プラクティス : 既存のデータベースのコンフィグレーション

■ ベスト プラクティス : データベース初期化の委託

ベスト プラクティス : データベースがない場合のシンプルなデータベースの作成

認証プロバイダ、認可プロバイダ、ロール マッピング プロバイダ、または資格マッピング プロバイダは、それが初めて使用されるときには、セキュリティ サービスを提供するために必要な情報のあるデータベースを見つけようとします。セキュリティ プロバイダがデータベースを見つけられない場合は、セキュリティ プロバイダにデータベースを作成させて、デフォルトのユーザ、グループ、セキュリティ ポリシー、セキュリティ ロール、および資格を自動的に格納することができます。このオプションは、開発およびテストに便利です。

WebLogic セキュリティ プロバイダとサンプル セキュリティ プロバイダは両方ともこのプラクティスに従います。WebLogic の認証プロバイダ、認可プロバイダ、ロール マッピング プロバイダ、および資格マッピング プロバイダは、ユーザ、グループ、セキュリティ ポリシー、セキュリティ ロール、および資格の情報を組み込み LDAP サーバに格納します。これらの WebLogic セキュリティ プロバイダのいずれかを使用する場合は、『WebLogic Security の管理』の「組み込み LDAP サーバのコンフィグレーション」で説明されている指示に従う必要があります。

注意 : dev2dev Web サイトの「Code Samples: WebLogic Server」で入手可能なサンプル セキュリティ プロバイダは、単純に properties ファイルを作成してそれをデータベースとして使用します。たとえば、サンプル認証プロバイダでは、ユーザおよびグループについての必要な情報が格納される `SampleAuthenticatorDatabase.java` というファイルが作成されます。

ベスト プラクティス：既存のデータベースのコンフィグレーション

外部 LDAP サーバなどのデータベースが既にある場合は、認証プロバイダ、認可プロバイダ、ロール マッピング プロバイダ、および資格マッピング プロバイダが必要とするユーザ、グループ、セキュリティ ポリシー、セキュリティ ロール、および資格の情報をそのデータベースに格納できます。既存のデータベースへの情報の格納は、それを目的として既に用意されているどのようなツールを使用しても行うことができます。

データベースに必要な情報が格納されたら、そのデータベースを参照するようにセキュリティ プロバイダをコンフィグレーションする必要があります。そのためには、セキュリティ プロバイダの MBean 定義ファイル (MDF) でカスタム属性を追加します。カスタム属性の例には、データベースのホスト、ポート、パスワードなどがあります。WebLogic MBeanMaker を通じて MDF を実行し、さらにいくつかのステップを行ってカスタム セキュリティ プロバイダの MBean タイプを生成したら、WebLogic Server Administration Console を使用してそれらの属性がデータベースを指すように設定します。

注意： MDF、MBean タイプ、および WebLogic MBeanMaker の詳細については、1-4 ページの「カスタム セキュリティ プロバイダをコンフィグレーションおよび管理する MBean タイプの生成」を参照してください。

コード リスト 2-5 は、WebLogic LDAP 認証プロバイダの MDF に属する一部のカスタム属性を例として示しています。これらの属性を使用することにより、WebLogic LDAP 認証プロバイダのデータベース (外部 LDAP サーバ) についての情報を指定することができるので、プロバイダでユーザおよびグループについての情報を見つけることができます。

コード リスト 2-5 LDAPAuthenticator.xml

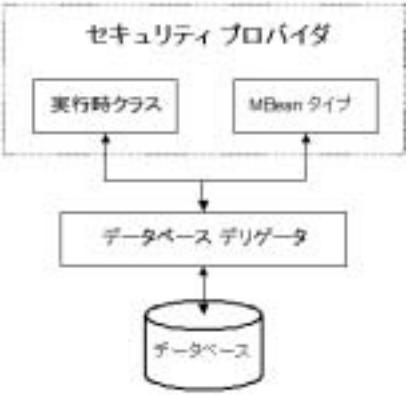
```
...

<MBeanAttribute
 Name = "UserObjectClass"
 Type = "java.lang.String"
 Default = "&quot;person&quot;"
 Description = "The LDAP object class that stores users."
/>
```

```
<MBeanAttribute
 Name = "UserNameAttribute"
 Type = "java.lang.String"
 Default = "&quot;uid&quot;"
 Description = "The attribute of an LDAP user object that specifies the name of
   the user."
/>

<MBeanAttribute
 Name = "UserDynamicGroupDNAttribute"
 Type = "java.lang.String"
 Description = "The attribute of an LDAP user object that specifies the
   distinguished names (DNs) of dynamic groups to which this user belongs.
   If such an attribute does not exist, WebLogic Server determines if a
   user is a member of a group by evaluating the URLs on the dynamic group.
   If a group contains other groups, WebLogic Server evaluates the URLs on
   any of the descendents of the group."
/>

<MBeanAttribute
 Name = "UserBaseDN"
 Type = "java.lang.String"
 Default = "&quot;ou=people, o=example.com&quot;"
 Description = "The base distinguished name (DN) of the tree in the LDAP directory
   that contains users."
/>

<MBeanAttribute
 Name = "UserSearchScope"
 Type = "java.lang.String"
 Default = "&quot;subtree&quot;"
 LegalValues = "subtree,onelevel"
 Description = "Specifies how deep in the LDAP directory tree to search for Users.
   Valid values are &lt;code&gt;subtree&lt;/code&gt;
   and &lt;code&gt;onelevel&lt;/code&gt;."
/>

...
```

ベスト プラクティス：データベース初期化の委託

可能な限り、セキュリティ プロバイダとセキュリティ プロバイダ データベースの間の初期化呼び出しは、**データベース デリゲータ**という中間クラスを介して行う必要があります。データベース デリゲータは、図 2-10 で示されているようにセキュリティ プロバイダの実行時クラスおよび MBean タイプと対話する必要があります。

図 2-10　データベース デリゲータ クラスの配置



データベース デリゲータは、WebLogic の認証プロバイダと資格マッピング プロバイダで使用されます。たとえば、WebLogic 認証プロバイダはデータベース デリゲータに働きかけてデフォルトのユーザおよびグループで組み込み LDAP サーバを初期化します。デフォルトのユーザおよびグループは、デフォルト セキュリティ レルムの認証サービスを提供するために必要な情報です。

カスタム セキュリティ プロバイダを開発するアプリケーション開発者およびセキュリティ ベンダは、データベース デリゲータを使用することをお勧めします。データベース デリゲータを使用すると、セキュリティ プロバイダのデータベースが隠蔽され、データベースへの呼び出しが一元化されるメリットがあります。

3　認証プロバイダ

認証とは、呼び出し側が、特定のユーザまたはシステムの代わりに動作していることを証明する際に使用するメカニズムのことです。認証は、ユーザ名とパスワードの組み合わせなどの資格を使用して「あなたは誰」という問いに答えます。

WebLogic Server では、認証プロバイダをユーザまたはシステム プロセスの ID を証明するために使用します。認証プロバイダでは、ID 情報を記憶したり、転送したり、その情報が必要な場合にサブジェクトを通じてシステムのさまざまなコンポーネントで利用できるようにしたりします。認証プロセスでは、プリンシパル検証プロバイダが、サブジェクト内に格納されるプリンシパル (ユーザおよびグループ) のセキュリティを強化するため、それらのプリンシパルに署名して信頼性を検証します。詳細については、第 5 章「プリンシパル検証プロバイダ」を参照してください。

以下の節では、認証プロバイダの概念と機能、およびカスタム認証プロバイダの開発手順について説明します。

注意： ID アサーション プロバイダは、ユーザまたはシステム プロセスがトークンを使用してそれぞれの ID を証明する特殊な形態の認証プロバイダです。詳細については、第 4 章「ID アサーション プロバイダ」を参照してください。

認証の概念

カスタム認証プロバイダの開発の詳細に立ち入る前に、以下の概念を理解しておくことが大切です。

ユーザ / グループ、プリンシパル、サブジェクト

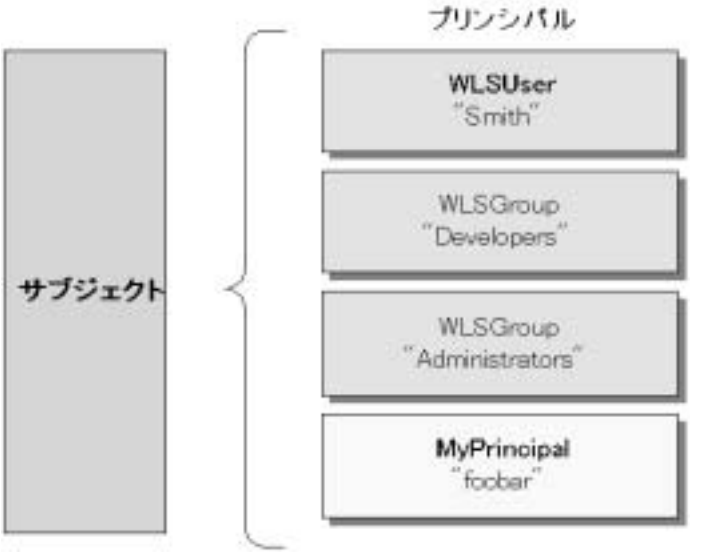
ユーザは、人を表すという点でオペレーティング システムのユーザに似ています。一方、**グループ**はユーザのカテゴリで、肩書きなどの共通の特徴で分類されたものです。ユーザをグループに分類すると、多数のユーザのアクセス パーミッションを管理しやすくなります。ユーザとグループの詳細については、『WebLogic リソースのセキュリティ』の「ユーザとグループ」を参照してください。

WebLogic Server のようなアプリケーション サーバでは、ユーザもグループもプリンシパルとして使用することができます。**プリンシパル**は、認証の結果としてユーザまたはグループに割り当てられる ID です。JAAS (Java Authentication and Authorization Service) では、プリンシパルなどの認証情報のコンテナとして**サブジェクト**を使用することになっています。同じサブジェクト内に格納されている各プリンシパルは、ある人の財布に入っている複数のカードのように、同じユーザの ID を異なる観点で表しています。たとえば、ATM カードは持ち主の身元を銀行に示し、メンバーシップ カードは持ち主の所属する組織に示します。JAAS の詳細については、3-6 ページの「JAAS (Java Authentication and Authorization Service)」を参照してください。

注意： サブジェクトは、WebLogic Server 6.x のユーザに取って代わるものです。

図 3-1 に、ユーザ、グループ、プリンシパル、およびサブジェクト間の関係を示します。

図 3-1 ユーザ、グループ、プリンシパル、およびサブジェクトの間の関係



認証に成功すると、その一環として、プリンシパルは署名され、その後の使用に備えてサブジェクトに格納されます。プリンシパルに署名するのはプリンシパル検証プロバイダであり、実際にプリンシパルをサブジェクトに格納するのは LoginModule です。後で、サブジェクト内に格納されたプリンシパルに呼び出し側がアクセスしようとすると、そのプリンシパルが署名されてから変更されていないことをプリンシパル検証プロバイダが確認したうえで、そのプリンシパルが呼び出し側に返されます (それ以外のセキュリティ条件がすべて満たされている場合)。

注意： プリンシパル検証プロバイダと LoginModule の詳細については、それぞれ第 5 章「プリンシパル検証プロバイダ」と 3-4 ページの「LoginModule」を参照してください。

さらに、WebLogic Server ユーザまたはグループを表すプリンシパルは、`weblogic.security.spi` パッケージの `WLSUser` インタフェースと `WLSGroup` インタフェースを実装する必要があります。

LoginModule

LoginModule は、認証プロバイダの必須コンポーネントであり、境界認証用に別個の LoginModule を開発する必要がある場合は ID アサーション プロバイダのコンポーネントにもなります。

LoginModule は、認証処理における「馬車馬」のような存在です。すべての LoginModule は、セキュリティ レルム内のユーザの認証と、サブジェクト内への必要なプリンシパル (ユーザ / グループ) の格納を担当します。境界認証に使用されない LoginModule も、提示された証明情報 (たとえば、ユーザのパスワード) が正しいかどうかを確認します。

注意： ID アサーション プロバイダと境界認証の詳細については、第 4 章「ID アサーション プロバイダ」を参照してください。

セキュリティ レルムに複数の認証プロバイダがコンフィグレーションされている場合、各認証プロバイダの LoginModule は同じサブジェクトにプリンシパルを格納します。したがって、ある認証プロバイダの LoginModule によって、WebLogic Server ユーザ (`WLSUser` インタフェースの実装) を表すプリンシパル「Joe」が追加された場合、セキュリティ レルム内の他のすべての認証プロバイダは、「Joe」に出会ったときに同じ人物を参照する必要があります。つまり、他の認証プロバイダの LoginModule は、同じ人物を参照するために、WebLogic Server ユーザを表す別のプリンシパル (「Joseph」など) をサブジェクトに追加しようとしてはなりません。ただし、別の認証プロバイダの LoginModule は、`WLSUser` 以外のタイプのプリンシパルを「Joseph」という名前で追加することができます。

LoginModule インタフェース

LoginModule は、ユーザ名とパスワードの組み合わせ、スマート カード、バイオメトリック装置などのさまざまな認証メカニズムを扱うように作成することができます。LoginModule を開発するには、JAAS (Java Authentication and Authorization Service) に基づき、認証情報のコンテナとしてサブジェクトを使用する `javax.security.auth.spi.LoginModule` インタフェースを実装します。`LoginModule` インタフェースを使用すると、単一アプリケーション用にさまざまな種類の認証技術をプラグインすることができ、WebLogic Security フレームワークはさまざまな要素から成る認証用に複数の `LoginModule` 実装をサポート

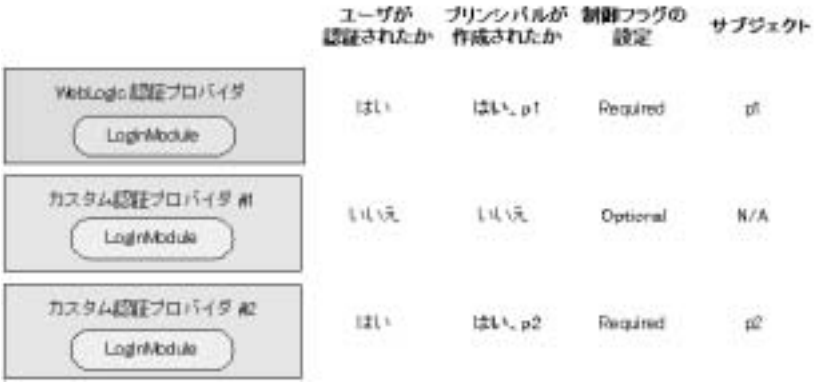
するように設計されています。さらに、LoginModule インスタンス間に依存関係を設けたり、それらのインスタンス間で資格を共有することもできます。ただし、LoginModule と認証プロバイダの関係は 1 対 1 です。つまり、網膜スキャン認証を扱う LoginModule と、スマート カードのようなハードウェア デバイスとのインタフェースを取る LoginModule を用意するには、それぞれに `LoginModule` インタフェースの実装が含まれる 2 つの認証プロバイダを開発およびコンフィグレーションする必要があります。詳細については、3-16 ページの「JAAS LoginModule インタフェースを実装する」を参照してください。

注意： LoginModule は、独自に開発するのではなくサードパーティのセキュリティ ベンダから入手することもできます。

LoginModule とさまざまな要素から成る認証

複数の認証プロバイダ (その結果として複数の LoginModule) をコンフィグレーションする方法は、認証プロセスの全体的な結果に影響します。これは、さまざまな要素から成る認証で特に重要です。まず、LoginModule は認証プロバイダのコンポーネントであるため、認証プロバイダがコンフィグレーションされた順序で呼び出されます。通常、認証プロバイダのコンフィグレーションには、WebLogic Server Administration Console を使用します。詳細については、3-35 ページの「Administration Console を使用してカスタム認証プロバイダをコンフィグレーションする」を参照してください。次に、各 LoginModule の制御フラグがどのように設定されるかによって、認証プロセスでのエラーの処理方法が決まります。図 3-2 は、それぞれ異なる認証プロバイダに属する 3 つの LoginModule () が関わるサンプル フローを示すとともに、異なる認証結果でサブジェクトに何が起こるのかを示しています。

図 3-2　LoginModule のサンプル フロー



カスタム認証プロバイダ #1 の制御フラグが REQUIRED に設定されていた場合、ユーザ認証ステップで認証が失敗すると、認証プロセス全体が失敗します。また、ユーザが WebLogic 認証プロバイダ (またはカスタム認証プロバイダ #2) によって認証されなかった場合、認証プロセス全体が失敗します。認証プロセスがこのように失敗した場合、3 つの LoginModule すべてがロールバックされ、サブジェクトにプリンシパルが格納されません。

注意： LoginModule 制御フラグの設定と LoginModule インタフェースの詳細については、それぞれ『Java Authentication and Authorization Service (JAAS) 1.0 LoginModule Developer's Guide』および Java 2 Enterprise Edition, v1.3.1 API 仕様 Javadoc の LoginModule インタフェースを参照してください。

JAAS (Java Authentication and Authorization Service)

クライアントが認証の必要なアプリケーション、アプレット、エンタープライズ JavaBean (EJB)、あるいはサーブレットのいずれであろうと、WebLogic Server では、JAAS (Java Authentication and Authorization Service) クラスを用いて、信頼性とセキュリティを確保しつつクライアントに対する認証を行います。JAAS では PAM (プラグイン可能な認証モジュール) フレームワークの Java 版を実装しており、それを使用することでアプリケーションは基礎となる認証技術からの独

立性を保つことができるようになります。このため、PAM フレームワークを利用することで、アプリケーションに修正を加えることなく新しいまたは最新版の認証技術を使用することができます。

WebLogic Server は、リモートのファット クライアントの認証および内部の認証で JAAS を使用します。したがって、JAAS に直に関与する必要があるのは、カスタム認証プロバイダの開発者とファット クライアント アプリケーションの開発者だけです。シン クライアントのユーザまたはコンテナ内のファット クライアント アプリケーション (サーブレットからエンタープライズ JavaBeans を呼び出すものなど) の開発者は、JAAS を直接使用したり、その知識を身につけたりする必要はありません。

JAAS が WebLogic Security フレームワークとどう連携するか

通常、JAAS クラスと WebLogic Server フレームワークを用いた認証は以下のように実行されます。

1. クライアントサイド アプリケーションがユーザまたはシステム プロセスから認証情報を取得します。このときのメカニズムは、クライアントのタイプごとに異なります。
2. クライアントサイド アプリケーションは、認証情報が格納された `CallbackHandler` を任意に作成できます。
   a. クライアントサイド アプリケーションは、`LoginContext` クラスを使用して `CallbackHandler` をローカル (クライアントサイド) LoginModule に渡します。ローカル LoginModule は WebLogic Server の一部として提供されている `UsernamePasswordLoginModule` の場合があります。
   b. ローカル LoginModule は、認証情報が格納された `CallbackHandler` を適切な WebLogic Server コンテナ (RMI、EJB、サーブレット、IIOP など) に渡します。

   注意： `CallbackHandler` は、可変個の引数を複合オブジェクトとしてメソッドに渡すことができるようにする高度に柔軟な JAAS 規格です。`CallbackHandler` のタイプは、`NameCallback`、`PasswordCallback`、および `TextInputCallback` の 3 つで、いずれも `javax.security.auth.callback` パッケージに収められています。`NameCallback` と `PasswordCallback` は、それぞれユーザ名とパス

ワードを返します。TextInputCallback は、ユーザがログイン フォームの追加フィールド (ユーザ名とパスワードを取得するフィールド以外のフィールド) に入力したデータにアクセスするために使用します。TextInputCallback はフォームの追加フィールドにつき 1 つ必要で、各 TextInputCallback のプロンプト文字列はフォームのフィールド名と一致する必要があります。WebLogic Server は、フォームベースの Web アプリケーション ログインに対してだけ TextInputCallback を使用します。CallbackHandler の詳細については、Java 2 Enterprise Edition, v1.4.0 API 仕様 Javadoc を参照して CallbackHandler インタフェースを調べてください。

LoginContext クラスの詳細については、Java 2 Enterprise Edition, v1.3.1 仕様 Javadoc を参照して LoginContext クラスを調べてください。

UsernamePasswordLoginModule の詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照して UsernamePasswordLoginModule クラスを調べてください。

クライアントサイド LoginModule を使用しない場合、ユーザ名とパスワードを他の方法で (たとえば初期 JNDI ルックアップの一部として) 指定できます。

3. WebLogic Server コンテナは、WebLogic Security フレームワークに働きかけを行います。認証情報が格納されたクライアントサイド CallbackHandler が存在する場合、それが WebLogic Security フレームワークに渡されます。

4. コンフィグレーションされた認証プロバイダごとに、WebLogic Security フレームワークは、渡された認証情報が格納された CallbackHandler を作成します。これらは WebLogic Security フレームワークによってサーバ上に作成された内部的な CallbackHandler であり、クライアントの CallbackHandler とは関係ありません。

5. WebLogic Security フレームワークは、認証プロバイダに関連付けられた LoginModule (認証情報を処理するために指定された特定の LoginModule) を呼び出します

   注意： LoginModule の詳細については、3-4 ページの「LoginModule」を参照してください。

   LoginModule が認証情報を利用してクライアントを認証しようとします。

6. 認証に成功すれば、以下の処理が行われます。

a. プリンシパル検証プロバイダによってプリンシパル (ユーザおよびグループ) が署名され、プログラムによるサーバ呼び出し間での信頼性が確保されます。プリンシパル検証プロバイダの詳細については、第 5 章「プリンシパル検証プロバイダ」を参照してください。

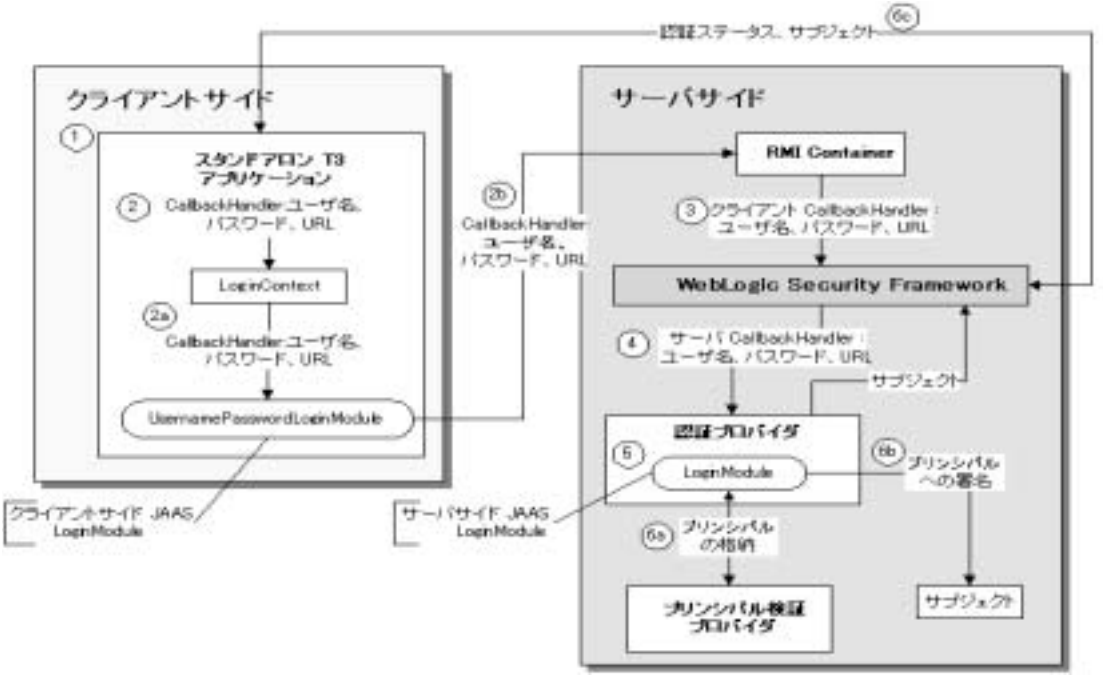
b. LoginModule は署名済みのプリンシパルをサブジェクトに関連付けます。そのオブジェクトが認証対象のユーザまたはシステム プロセスを表します。サブジェクトとプリンシパルの詳細については、3-2 ページの「ユーザ / グループ、プリンシパル、サブジェクト」を参照してください。

注意： サーバサイドですべてが行われる認証の場合、プロセスは手順 3 で始まり、WebLogic Server コンテナは手順 4 の前に `weblogic.security.services.authentication.login()` メソッドを呼び出します。`weblogic.security.services.authentication.login` メソッドは、WebLogic Server 7.0 SP01 のみで使用できます。

例：スタンドアロンの T3 アプリケーション

スタンドアロンの T3 アプリケーションの場合に JAAS クラスが WebLogic Security フレームワークとどう連携するかを図 3-3 に示し、その後それについて説明します。

図 3-3　JAAS クラスと WebLogic Server を用いた認証



この例で、JAAS クラスと WebLogic Server フレームワークを用いた認証は以下のように実行されます。

1. T3 アプリケーションがユーザまたはシステム プロセスから認証情報 (ユーザ名、パスワード、および URL) を取得します。
2. T3 アプリケーションは、認証情報が格納された `CallbackHandler` を作成します。
   a. T3 アプリケーションは、`LoginContext` クラスを使用して `CallbackHandler` を `UsernamePasswordLoginModule` に渡します。

      注意： `weblogic.security.auth.login.UsernamePasswordLoginModule` は標準の JAAS `javax.security.auth.spi.LoginModule` インタフェースを実装し、クライアントサイドの API を使用して WebLogic Server インスタンスに対する WebLogic クライアントの認証を行います。LoginModule は、T3 クライアントと IIOP クラ

イアントの両方で使用できます。この LoginModule の呼び出し側では、`CallbackHandler` を実装してユーザ名 (`NameCallback`)、パスワード (`PasswordCallback`)、および URL (`URLCallback`) を渡す必要があります。

b. `UsernamePasswordLoginModule` は、認証情報 (ユーザ名、パスワード、および URL) が格納された `CallbackHandler` を WebLogic Server RMI コンテナに渡します。

3. WebLogic Server RMI コンテナは、WebLogic Security フレームワークに働きかけを行います。認証情報が格納されたクライアントサイド `CallbackHandler` が WebLogic Security フレームワークに渡されます。

4. コンフィグレーションされた認証プロバイダごとに、WebLogic Security フレームワークが渡されたユーザ名、パスワード、および URL の格納された `CallbackHandler` を作成します。これらは WebLogic Security フレームワークによってサーバ上に作成された内部的な `CallbackHandler` であり、クライアントの `CallbackHandler` とは関係ありません。

5. WebLogic Security フレームワークは、認証プロバイダに関連付けられた LoginModule (認証情報を処理するために指定された特定の LoginModule) を呼び出します。

   LoginModule が認証情報を利用してクライアントを認証しようとします。

6. 認証に成功すれば、以下の処理が行われます。

   a. プリンシパル検証プロバイダによってプリンシパル (ユーザおよびグループ) が署名され、プログラムによるサーバ呼び出し間での信頼性が確保されます。

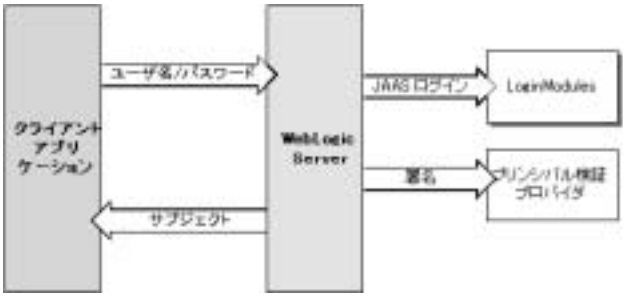
   b. LoginModule は署名済みのプリンシパルをサブジェクトに関連付けます。そのオブジェクトが認証対象のユーザまたはシステムを表します。

   c. WebLogic Security フレームワークは認証ステータスを T3 クライアント アプリケーションに返し、T3 クライアント アプリケーションは認証済みのサブジェクトを WebLogic Security フレームワークから受け取ります。

認証プロセス

図 3-4 に、ファットクライアント ログインの認証プロセスの仕組みを示します。JAAS はサーバ上で動作してログインを実行します。シンクライアント ログイン (ブラウザ クライアント) の場合でも、JAAS はサーバ上で動作します。

図 3-4　認証プロセス



注意： JASS プロセスを直接扱うのは、カスタム認証プロバイダの開発者だけです。クライアント アプリケーションは、JNDI 初期コンテキスト作成または JAAS のいずれかを使用してユーザ名とパスワードを受け渡します。

ユーザがユーザ名とパスワードの組み合わせを使用してシステムにログインしようとすると、WebLogic Server はそのユーザのユーザ名とパスワードを検証することによって信頼を確立し、JAAS の要件に従って、プリンシパルが格納されたサブジェクトを返します。図 3-4 に示すように、このプロセスでは LoginModule とプリンシパル検証プロバイダを使用する必要があります。これらについては、それぞれ 3-4 ページの「LoginModule」と第 5 章「プリンシパル検証プロバイダ」で詳しく解説します。

呼び出し側の ID の確認に成功すると、認証コンテキストが確立され、ID が確認されたユーザまたはシステムに関しては、そのコンテキストを通じて他のエンティティに対する認証を行うことができます。認証コンテキストはまた、アプリケーション コンポーネントに委託することもでき、それによって、そのコンポーネントは別のアプリケーション コンポーネントを呼び出しつつ、元の呼び出し側として動作できるようになります。

カスタム認証プロバイダを開発する必要があるか

WebLogic Server のデフォルト (アクティブ) セキュリティ レルムには、WebLogic 認証プロバイダが含まれます。

注意： WebLogic 認証プロバイダを WebLogic 認可プロバイダと組み合わせれば、WebLogic Server リリース 6.x で利用できたファイル レルムの機能の代わりになります。

WebLogic 認証プロバイダは、ユーザ名とパスワードの委託認証をサポートし、組み込み LDAP サーバを利用してユーザとグループの情報を格納します。WebLogic 認証プロバイダを使用すると、ユーザとグループ メンバーシップを編集、表示、および管理できます。追加の認証タスクを実行する場合は、カスタム認証プロバイダを開発する必要があります。

注意： X509 証明書または CSIv2 (CORBA Common Secure Interoperability version 2) を使用して境界認証を行うには、カスタム ID アサーション プロバイダを開発する必要があります。詳細については、第 4 章「ID アサーション プロバイダ」を参照してください。

カスタム認証プロバイダの開発方法

WebLogic 認証プロバイダが開発者の要求を満たさない場合、次の手順でカスタム認証プロバイダを開発することができます。

1. 3-14 ページの「適切な SSPI を使用して実行時クラスを作成する」
2. 3-27 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 3-35 ページの「Administration Console を使用してカスタム認証プロバイダをコンフィグレーションする」

適切な SSPI を使用して実行時クラスを作成する

実行時クラスを作成する前に、以下の作業が必要です。

- 2-4 ページの「「Provider」SSPI の目的を理解する」
- 2-7 ページの「SSPI 階層を理解し、実行時クラスを 1 つ作成するのか 2 つ作成するのかを決定する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム認証プロバイダの実行時クラスを作成します。

- 3-14 ページの「AuthenticationProvider SSPI を実装する」
- 3-16 ページの「JAAS LoginModule インタフェースを実装する」

カスタム認証プロバイダの実行時クラスの作成例については、3-19 ページの「例 : サンプル認証プロバイダの実行時クラスの作成」を参照してください。

AuthenticationProvider SSPI を実装する

`AuthenticationProvider` SSPI を実装するには、2-4 ページの「「Provider」SSPI の目的を理解する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

getLoginModuleConfiguration

```
public AppConfigurationEntry getLoginModuleConfiguration()
```

`getLoginModuleConfiguration` メソッドは、認証プロバイダの関連付けられた LoginModule に関する情報を取得します。その情報は、`AppConfigurationEntry` として返されます。`AppConfigurationEntry` は、LoginModule のクラス名、LoginModule の制御フラグ (認証プロバイダの関連する MBean を通じて渡されていた)、および LoginModule のコンフィグレーション オプション マップ (他のコンフィグレーション情報を LoginModule に渡すことを可能にする) の格納された JAAS (Java Authentication and Authorization Service) クラスです。

`AppConfigurationEntry` クラス (`javax.security.auth.login` パッケージ内に定義されている) と LoginModule 用の制御フラグ オプションの詳細については、Java 2 Enterprise Edition, v1.3.1 API 仕様 Javadoc を

参照して AppConfigurationEntry クラスと Configuration クラスを調べてください。LoginModule の詳細については、3-4 ページの「LoginModule」を参照してください。セキュリティ プロバイダと MBean の詳細については、2-11 ページの「MBean タイプが必要な理由を理解する」を参照してください。

getAssertionModuleConfiguration

```
public AppConfigurationEntry
getAssertionModuleConfiguration()
```

`getAssertionModuleConfiguration` メソッドは、ID アサーション プロバイダの関連付けられた LoginModule に関する情報を取得します。その情報は、`AppConfigurationEntry` として返されます。`AppConfigurationEntry` は、LoginModule のクラス名、LoginModule の制御フラグ (ID アサーション プロバイダの関連する MBean を通じて渡されていた)、および LoginModule のコンフィグレーション オプション マップ (他のコンフィグレーション情報を LoginModule に渡すことを可能にする) の格納された JAAS クラスです。

注意： `getAssertionModuleConfiguration` メソッドの実装は `null` を返す場合があります (ID アサーション プロバイダが認証プロバイダと同じ LoginModule を使用する場合)。

getPrincipalValidator

```
public PrincipalValidator getPrincipalValidator()
```

`getPrincipalValidator` メソッドは、プリンシパル検証プロバイダの実行時クラス (`PrincipalValidator` SSPI 実装) の参照を取得します。WebLogic プリンシパル検証プロバイダは、ほとんどの場合で使用できます。WebLogic プリンシパル検証プロバイダの返し方の例については、コード リスト 3-1 を参照してください。プリンシパル検証プロバイダの詳細については、第 5 章「プリンシパル検証プロバイダ」を参照してください。

getIdentityAsserter

```
public IdentityAsserter getIdentityAsserter()
```

`getIdentityAsserter` メソッドは、ID アサーション プロバイダの実行時クラス (`IdentityAsserter` SSPI 実装) の参照を取得します。ほとんどの場合、このメソッドの戻り値は `null` になります。例については、コード リスト 3-1 を参照してください。ID アサーション プロバイダの

詳細については、第 4 章「ID アサーション プロバイダ」を参照してください。

AuthenticationProvider SSPI と上記のメソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

JAAS LoginModule インタフェースを実装する

JAAS javax.security.auth.spi.LoginModule インタフェースを実装するには、以下のメソッドを実装する必要があります。

initialize

```
public void initialize (Subject subject, CallbackHandler
callbackHandler, Map sharedState, Map options)
```

initialize メソッドは LoginModule を初期化します。引数として取るのは、結果として得られるプリンシパルを格納するサブジェクト、認証プロバイダが認証情報のコンテナにコールバックするのに使用する CallbackHandler、任意の共有状態情報のマップ、およびコンフィグレーション オプション (すなわち、LoginModule に渡したい任意の付加的情報) のマップです。

CallbackHandler は、可変個の引数を複合オブジェクトとしてメソッドに渡すことができるようにする高度に柔軟な JAAS 規格です。CallbackHandler の詳細については、Java 2 Enterprise Edition, v1.3.1 API 仕様 Javadoc を参照して CallbackHandler インタフェースを調べてください。

login

```
public boolean login() throws LoginException
```

login メソッドは、ユーザを認証し、認証情報のコンテナにコールバックすることでそのユーザのプリンシパルを作成しようと試みます。複数の LoginModule が複数の認証プロバイダの一部としてコンフィグレーションされている場合、このメソッドは LoginModule ごとにそのコンフィグレーションの順序で呼び出されます。ログインが成功したかどうか (すなわち、プリンシパルが作成されたかどうか) についての情報は、LoginModule ごとに格納されます。

commit

```
public boolean commit() throws LoginException
```

commit メソッドは、login() メソッドで作成されたプリンシパルをサブジェクトに追加しようと試みます。このメソッドも、複数の認証プロバイダの一部としてコンフィグレーションされている LoginModule ごとに呼び出され、順番に実行されます。コミットが成功したかどうかの情報は、LoginModule ごとに格納されます。

abort

```
public boolean abort() throws LoginException
```

abort メソッドは、コンフィグレーションされている認証プロバイダの一部としてコンフィグレーションされている LoginModule のコミットが失敗した (つまり、関連する REQUIRED、REQUISITE、SUFFICIENT、および OPTIONAL LoginModule が成功しなかった) 場合に LoginModule ごとに呼び出されます。abort メソッドは、LoginModule のプリンシパルをサブジェクトから削除し、実行されたアクションを効果的にロールバックします。利用可能な制御フラグの設定の詳細については、Java 2 Enterprise Edition, v1.3.1 API 仕様 Javadoc を参照して LoginModule インタフェースを調べてください。

logout

```
public boolean logout() throws LoginException
```

logout メソッドは、ユーザをシステムからログアウトさせようとします。また、サブジェクトのリセットも行うので、関連付けられているプリンシパルは格納されなくなります。

注意： LoginModule.logout メソッドは、WebLogic 認証プロバイダに対しても、カスタム認証プロバイダに対しても呼び出されません。これは単純に、いったんプリンシパルが作成されサブジェクトに配置されると、WebLogic Security フレームワークではサブジェクトのライフサイクルを制御できなくなるからです。そのため、開発者によって記述された、ログインおよびサブジェクトの取得を行う JAAS LoginContext を作成するユーザ コードでは、LoginContext.logout メソッドも呼び出す必要があります。そうしたユーザ コードが JAAS を直接使用する Java クライアントで実行される場合は、必要に応じてコードで LoginContext.logout メソッドを呼び出すことができます。これによってサブジェクトがクリアされます。ユーザ コードがサーブレットで実行される場合、サーブレットではサーブレット セッションからユーザをログアウトできます。これによってサブジェクトがクリアされます。

JAAS LoginModule インタフェースと上記のメソッドの詳細については、『Java Authentication and Authorization Service (JAAS) 1.0 Developer's Guide』および Java 2 Enterprise Edition, v1.3.1 API 仕様 Javadoc の LoginModule インタフェースを参照してください。

LoginModule からのカスタム例外の送出

作成した LoginModule からカスタム例外を送出することが必要な場合もあります。送出されたカスタム例外はアプリケーションで捕捉でき、適切なアクションが実行されます。たとえば、PasswordChangeRequiredException が LoginModule から送出された場合は、アプリケーション内でその例外を捕捉し、それを使用して、パスワードを変更できるページにユーザを誘導することができます。

LoginModule からカスタム例外を送出してそれをアプリケーションで捕捉するには、以下の条件を満たす必要があります。

1. 例外を捕捉するアプリケーションがサーバ上で動作している (ファット クライアントはカスタム例外を捕捉できない)。
2. コンパイル時とデプロイ時の両方で、サーブレットがカスタム例外クラスにアクセスできる。この状態にするためには、嗜好に応じて以下のいずれかの手段を使用します。
   - 3-18 ページの「手段 1 : カスタム例外をシステムおよびコンパイラのクラスパス経由でアクセス可能にする」
   - 3-19 ページの「手段 2 : カスタム例外をアプリケーションのクラスパス経由でアクセス可能にする」

手段 1 : カスタム例外をシステムおよびコンパイラのクラスパス経由でアクセス可能にする

1. LoginException を拡張する例外クラスを記述する。
2. LoginModule および AuthenticationProvider インタフェースを実装するクラスでカスタム例外クラスを使用する。
3. セキュリティ プロバイダの実行時クラスをコンパイルするときに、システムとコンパイラの両方のクラスパスでカスタム例外クラスを設定する。

4. 「WebLogic MBeanMaker を使用して MBean タイプを生成する」

手段 2 : カスタム例外をアプリケーションのクラスパス経由でアクセス可能にする

1. `LoginException` を拡張する例外クラスを記述する。
2. `LoginModule` および `AuthenticationProvider` インタフェースを実装するクラスでカスタム例外クラスを使用する。
3. アプリケーションのビルドのクラスパスでカスタム例外のソースを設定し、アプリケーションの JAR/WAR ファイルのクラスパスでそれをインクルードする。
4. 「WebLogic MBeanMaker を使用して MBean タイプを生成する」
5. WebLogic MBeanMaker で生成された MJF (MBean JAR ファイル) にカスタム例外クラスを追加する。
6. アプリケーションのコンパイル時に MJF をインクルードする。

例 : サンプル認証プロバイダの実行時クラスの作成

コード リスト 3-1 は、サンプル認証プロバイダの 2 つの実行時クラスの 1 つである `SampleAuthenticationProviderImpl.java` クラスを示しています。実行時クラスには以下の実装が含まれています。

- `initialize`、`getDescription`、および `shutdown` という `SecurityProvider` インタフェースから継承した 3 つのメソッド (2-4 ページの「「Provider」 SSPI の目的を理解する」 を参照)
- `getLoginModuleConfiguration`、`getAssertionModuleConfiguration`、`getPrincipalValidator`、および `getIdentityAsserter` という `AuthenticationProvider` SSPI の 4 つのメソッド (3-14 ページの「AuthenticationProvider SSPI を実装する」 を参照)

注意 : コード リスト 3-1 の太字のコードは、クラス宣言とメソッド シグネチャを示しています。

コード リスト 3-1　SampleAuthenticationProviderImpl.java

```
package examples.security.providers.authentication;

import java.util.HashMap;
import javax.security.auth.login.AppConfigurationEntry;
import javax.security.auth.login.AppConfigurationEntry.LoginModuleControlFlag;
import weblogic.management.security.ProviderMBean;
import weblogic.security.provider.PrincipalValidatorImpl;
import weblogic.security.spi.AuthenticationProvider;
import weblogic.security.spi.IdentityAsserter;
import weblogic.security.spi.PrincipalValidator;
import weblogic.security.spi.SecurityServices;

public final class SampleAuthenticationProviderImpl implements
AuthenticationProvider
{
   private String description;
   private SampleAuthenticatorDatabase database;
   private LoginModuleControlFlag controlFlag;

   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SampleAuthenticationProviderImpl.initialize");
      SampleAuthenticatorMBean myMBean = (SampleAuthenticatorMBean)mbean;
      description = myMBean.getDescription() + "\n" + myMBean.getVersion();
      database = new SampleAuthenticatorDatabase(myMBean);

      String flag = myMBean.getControlFlag();
      if (flag.equalsIgnoreCase("REQUIRED")) {
        controlFlag = LoginModuleControlFlag.REQUIRED;
      } else if (flag.equalsIgnoreCase("OPTIONAL")) {
        controlFlag = LoginModuleControlFlag.OPTIONAL;
      } else if (flag.equalsIgnoreCase("REQUISITE")) {
        controlFlag = LoginModuleControlFlag.REQUISITE;
      } else if (flag.equalsIgnoreCase("SUFFICIENT")) {
        controlFlag = LoginModuleControlFlag.SUFFICIENT;
      } else {
        throw new IllegalArgumentException("invalid flag value" + flag);
      }
   }

   public String getDescription()
   {
      return description;
   }

   public void shutdown()
   {
```

```
      System.out.println("SampleAuthenticationProviderImpl.shutdown");
   }

   private AppConfigurationEntry getConfiguration(HashMap options)
   {
      options.put("database", database);
      return new
        AppConfigurationEntry(
          "examples.security.providers.authentication.SampleLoginModuleImpl",
          controlFlag,
          options
        );
   }

   public AppConfigurationEntry getLoginModuleConfiguration()
   {
      HashMap options = new HashMap();
      return getConfiguration(options);
   }

   public AppConfigurationEntry getAssertionModuleConfiguration()
   {
      HashMap options = new HashMap();
      options.put("IdentityAssertion","true");
      return getConfiguration(options);
   }

   public PrincipalValidator getPrincipalValidator()
   {
      return new PrincipalValidatorImpl();
   }

   public IdentityAsserter getIdentityAsserter()
   {
      return null;
   }

}
```

コード リスト 3-2 は、サンプル認証プロバイダの 2 つの実行時クラスの 1 つである `SampleLoginModuleImpl.java` クラスを示しています。この実行時クラスは、JAAS LoginModule インタフェースを実装します (3-16 ページの「JAAS LoginModule インタフェースを実装する」を参照)。したがってその `initialize`、`login`、`commit`、`abort`、および `logout` メソッドの実装が含まれています。

注意： コード リスト 3-2 の太字のコードは、クラス宣言とメソッド シグネチャを示しています。

コード リスト 3-2　SampleLoginModuleImpl.java

```
package examples.security.providers.authentication;

import java.io.IOException;
import java.util.Enumeration;
import java.util.Map;
import java.util.Vector;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.spi.LoginModule;
import weblogic.management.utils.NotFoundException;
import weblogic.security.spi.WLSGroup;
import weblogic.security.spi.WLSUser;
import weblogic.security.principal.WLSGroupImpl;
import weblogic.security.principal.WLSUserImpl;

final public class SampleLoginModuleImpl implements LoginModule
{
   private Subject subject;
   private CallbackHandler callbackHandler;
   private SampleAuthenticatorDatabase database;

   // ログインかどうかを判別するか、ID を断定する
   private boolean isIdentityAssertion;

   // 認証ステータス
   private boolean loginSucceeded;
   private boolean principalsInSubject;
   private Vector principalsForSubject = new Vector();

  public void initialize(Subject subject, CallbackHandler callbackHandler, Map
   sharedState, Map options)
   {
      // コンストラクタの後でログインの前にのみ、一度だけ呼び出される
```

```
    System.out.println("SampleLoginModuleImpl.initialize");
    this.subject = subject;
    this.callbackHandler = callbackHandler;

    // ID アサーション オプションを確認する
    isIdentityAssertion =
       "true".equalsIgnoreCase((String)options.get("IdentityAssertion"));

    database = (SampleAuthenticatorDatabase)options.get("database");
 }

 public boolean login() throws LoginException
 {
    // initialize の後にのみ、一度だけ呼び出される

    System.out.println("SampleLoginModuleImpl.login");

    // loginSucceeded       は false
    // principalsInSubject  は false
    // user                 は null
    // group                は null

    Callback[] callbacks = getCallbacks();

    String userName = getUserName(callbacks);

    if (userName.length() > 0) {
       if (!database.userExists(userName)) {
          throwFailedLoginException("Authentication Failed: User " + userName
          + " doesn't exist.");
       }
      if (!isIdentityAssertion) {
       String passwordWant = null;
       try {
          passwordWant = database.getUserPassword(userName);
       } catch (NotFoundException shouldNotHappen) {}
          String passwordHave = getPasswordHave(userName, callbacks);
          if (passwordWant == null || !passwordWant.equals(passwordHave)) {
             throwFailedLoginException(
               "Authentication Failed: User " + userName + " bad password. " +
               "Have " + passwordHave + ".  Want " + passwordWant + "."
             );
          }
       }
       } else {
        // 匿名ログイン - 許可する ?
       System.out.println("\tempty userName");
       }
```

```
            loginSucceeded = true;
            principalsForSubject.add(new WLSUserImpl(userName));
            addGroupsForSubject(userName);

            return loginSucceeded;
      }

      public boolean commit() throws LoginException
      {
         // ログインの後にのみ、一度だけ呼び出される

         // loginSucceeded      は true または false
         // principalsInSubject は false
         // user      は !loginSucceeded の場合は null、それ以外の場合は null または非
null
         // group     は user == null の場合は null、それ以外の場合は null または非 null

         System.out.println("SampleLoginModule.commit");
         if (loginSucceeded) {
            subject.getPrincipals().addAll(principalsForSubject);
            principalsInSubject = true;
            return true;
         } else {
            return false;
         }
      }

      public boolean abort() throws LoginException
      {
         // 認証プロセスを中止するために abort メソッドが呼び出される
         // これは、フェーズ 1 が失敗した場合の、認証のフェーズ 2 であり、
         // LoginContext の全体的な認証が失敗した場合に呼び出される

         // loginSucceeded      は true または false
         // user      は !loginSucceeded の場合は null、それ以外の場合は null または非
null
         // group     は user == null の場合は null、それ以外の場合は null または非 null
         // principalsInSubject      は user が null の場合は false、それ以外の場合は
true
         //                          または false

         System.out.println("SampleLoginModule.abort");
         if (principalsInSubject) {
            subject.getPrincipals().removeAll(principalsForSubject);
            principalsInSubject = false;
         }
```

```
    return true;
  }

  public boolean logout() throws LoginException
  {
    // 呼び出し禁止
    System.out.println("SampleLoginModule.logout");
    return true;
  }

  private void throwLoginException(String msg) throws LoginException
  {
    System.out.println("Throwing LoginException(" + msg + ")");
    throw new LoginException(msg);
  }

  private void throwFailedLoginException(String msg) throws FailedLoginException
  {
    System.out.println("Throwing FailedLoginException(" + msg + ")");
    throw new FailedLoginException(msg);
  }

  private Callback[] getCallbacks() throws LoginException
  {
    if (callbackHandler == null) {
      throwLoginException("No CallbackHandler Specified");
    }

    if (database == null) {
      throwLoginException("database not specified");
    }

    Callback[] callbacks;
    if (isIdentityAssertion) {
      callbacks = new Callback[1];
    } else {
      callbacks = new Callback[2];
      callbacks[1] = new PasswordCallback("password:", false);
    }
    callbacks[0] = new NameCallback("username: ");

    try {
       callbackHandler.handle(callbacks);
    } catch (IOException e) {
      throw new LoginException(e.toString());
    } catch (UnsupportedCallbackException e) {
      throwLoginException(e.toString() + " " + e.getCallback().toString());
    }
```

```
      return callbacks;
   }

   private String getUserName(Callback[] callbacks) throws LoginException
   {
      String userName = ((NameCallback)callbacks[0]).getName();
      if (userName == null) {
         throwLoginException("Username not supplied.");
      }
      System.out.println("\tuserName\t= " + userName);
      return userName;
   }

   private void addGroupsForSubject(String userName)
   {
      for (Enumeration e = database.getUserGroups(userName);
         e.hasMoreElements();) {
            String groupName = (String)e.nextElement();
            System.out.println("\tgroupName\t= " + groupName);
            principalsForSubject.add(new WLSGroupImpl(groupName));
      }
   }

   private String getPasswordHave(String userName, Callback[] callbacks) throws
   LoginException
   {
      PasswordCallback passwordCallback = (PasswordCallback)callbacks[1];
      char[] password = passwordCallback.getPassword();
      passwordCallback.clearPassword();
      if (password == null || password.length < 1) {
         throwLoginException("Authentication Failed: User " + userName + ".
            Password not supplied");
      }
      String passwd = new String(password);
      System.out.println("\tpasswordHave\t= " + passwd);
      return passwd;
   }

}
```

WebLogic MBeanMaker を使用して MBean タイプを生成する

カスタム セキュリティ プロバイダの MBean タイプを生成する前に、以下の作業が必要です。

- 2-11 ページの「MBean タイプが必要な理由を理解する」
- 2-12 ページの「拡張および実装する SSPI MBean を決定する」
- 2-13 ページの「MBean 定義ファイル (MDF) の基本的な要素を理解する」
- 2-15 ページの「SSPI MBean の階層と Administration Console に対する影響を理解する」
- 2-17 ページの「WebLogic MBeanMaker によって提供されるものを理解する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム認証プロバイダの MBean タイプを作成します。

1. 3-27 ページの「MBean 定義ファイル (MDF) を作成する」
2. 3-28 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 3-32 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」
4. 3-33 ページの「WebLogic Server 環境に MBean タイプをインストールする」

注意： これらの手順の実行方法は、いくつかのサンプル セキュリティ プロバイダ (dev2dev Web サイトの「Code Samples: WebLogic Server」で入手可能) に示されています。

この節で説明する手順はすべて、Windows 環境での作業を想定しています。

MBean 定義ファイル (MDF) を作成する

MBean 定義ファイル (MDF) を作成するには、次の手順に従います。

1. サンプル認証プロバイダの MDF をテキスト ファイルにコピーします。

   注意： サンプル認証プロバイダの MDF は、`SampleAuthenticator.xml` です。

2. MDF で `<MBeanType>` 要素と `<MBeanAttribute>` 要素の内容をカスタム認証プロバイダに合わせて修正します。

3. カスタム属性および操作 (つまり、`<MBeanAttribute>` および `<MBeanOperation>` 要素) を MDF に追加します。

4. ファイルを保存します。

注意： MDF 要素の構文についての完全なリファレンスは、付録 A「MBean 定義ファイル (MDF) 要素の構文」に収められています。

WebLogic MBeanMaker を使用して MBean タイプを生成する

MDF を作成したら、WebLogic MBeanMaker を使用してそれを実行できます。WebLogic MBeanMaker は現在のところコマンドライン ユーティリティで、入力として MDF を受け取り、MBean インタフェース、MBean 実装、関連する MBean 情報ファイルなどの中間 Java ファイルをいくつか出力します。これらの中間ファイルが合わさって、カスタム セキュリティ プロバイダの **MBean タイプ**になります。

MBean タイプの作成手順は、カスタム認証プロバイダの設計に応じて異なります。必要な設計に合わせて適切な手順を実行してください。

- 3-28 ページの「任意 SSPI MBean とカスタム操作を追加しない場合」
- 3-29 ページの「任意 SSPI MBean またはカスタム操作を追加する場合」

任意 SSPI MBean とカスタム操作を追加しない場合

カスタム認証プロバイダの MDF に任意 SSPI MBean もカスタム操作も実装しない場合、次の手順に従います。

1. 新しい DOS シェルを作成します。

2. 次のコマンドを入力します。

```
java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
weblogic.management.commo.WebLogicMBeanMaker
```

ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

`xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

`-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。したがって、MDF (つまり認証プロバイダ) が複数ある場合には、このプロセスを繰り返す必要があります。

3. 3-32 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

任意 SSPI MBean またはカスタム操作を追加する場合

カスタム認証プロバイダの MDF に任意 SSPI MBean またはカスタム操作を実装する場合、以下の質問に答えながら手順を進めてください。

- MBean タイプを作成するのは初めてですか。初めての場合は、次の手順に従ってください。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

```
java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
weblogic.management.commo.WebLogicMBeanMaker
```

ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

`xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

`-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。したがって、MDF (つまり認証プロバイダ) が複数ある場合には、このプロセスを繰り返す必要があります。

3. 任意 SSPI MBean を MDF に実装した場合は、次の手順に従います。
   a. MBean 実装ファイルを見つけます。

      WebLogic MBeanMaker によって生成される MBean 実装ファイルには、*`MBeanName`*`Impl.java` という名前が付けられます。たとえば、`SampleAuthenticator` という名前の MDF の場合、編集される MBean 実装ファイルは `SampleAuthenticatorImpl.java` という名前になります。

   b. MDF で実装した任意 SSPI MBean ごとに、メソッド スタブを「Mapping MDF Operation Declarations to Java Method Signatures Document」(dev2dev Web サイトで入手可能) から MBean 実装ファイルにコピーし、各メソッドを実装します。任意 SSPI MBean が継承するメソッドもすべて実装してください。
4. MDF にカスタム属性 / 操作を含めた場合は、メソッド スタブを使用してメソッドを実装します。
5. ファイルを保存します。
6. 3-32 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

■ 既存の MBean タイプの更新ですか。更新の場合は、次の手順に従ってください。

1. WebLogic MBeanMaker によって現在のメソッドの実装が上書きされないように、既存の MBean 実装ファイルを一時ディレクトリにコピーします。
2. 新しい DOS シェルを作成します。
3. 次のコマンドを入力します。

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、

filesdir は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

xmlfile が入力されるたびに、新しい出力ファイル群が生成されます。*filesdir* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

`-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。したがって、MDF (つまり認証プロバイダ) が複数ある場合には、このプロセスを繰り返す必要があります。

4. 任意 SSPI MBean を MDF に実装した場合は、次の手順に従います。

   a. MBean 実装ファイルを見つけて開きます。

      WebLogic MBeanMaker によって生成される MBean 実装ファイルには、`<MBeanName>Impl.java` という名前が付けられます。たとえば、`SampleAuthenticator` という名前の MDF の場合、編集される MBean 実装ファイルは `SampleAuthenticatorImpl.java` という名前になります。

   b. 手順 1 で一時ディレクトリに保存した既存の MBean 実装ファイルを開きます。

   c. 既存の MBean 実装ファイルを、WebLogic MBeanMaker によって生成された MBean 実装ファイルと同期させます。

      これには、メソッドの実装を既存の MBean 実装ファイルから新しく生成された MBean 実装ファイルにコピー (または、新しく生成された MBean 実装ファイルから既存の MBean 実装ファイルに新しいメソッドを追加) し、いずれの MBean 実装ファイルにも入っているメソッドのメソッド シグネチャへの変更が使用する MBean 実装ファイルに反映されていることを確認するといった作業が必要です。

   d. MDF を修正して元の MDF にはない任意 SSPI MBean を実装した場合は、メソッド スタブを「Mapping MDF Operation Declarations to Java Method Signatures Document」(dev2dev Web サイトで入手可能) から MBean 実装ファイルにコピーし、各メソッドを実装します。任意 SSPI MBean が継承するメソッドもすべて実装してください。

5. MDF を変更して元の MDF にはないカスタム操作を含めた場合、メソッド スタブを使用してメソッドを実装します。
6. 完成した、つまりすべてのメソッドを実装した MBean 実装ファイルを保存します。
7. この MBean 実装ファイルを、WebLogic MBeanMaker が MBean タイプの実装ファイルを配置したディレクトリにコピーします。このディレクトリは、手順 3 で *`filesdir`* として指定しました (手順 3 の結果として WebLogic MBeanMaker で生成された MBean 実装ファイルがオーバーライドされる)。
8. 3-32 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

生成される MBean インタフェース ファイルについて

MBean インタフェース ファイルは、実行時クラスまたは MBean 実装ファイルがコンフィグレーション データを取得するために使用する MBean とのクライアント サイド API です。2-4 ページの「「Provider」SSPI の目的を理解する」で説明されているように、これは initialize メソッドで使用するのが一般的です。

WebLogic MBeanMaker では、作成済みの MDF から MBean タイプを生成するので、生成される MBean インタフェース ファイルの名前は、その MDF 名の後に「MBean」というテキストが付いたものになります。たとえば、WebLogic MBeanMaker を使用して `SampleAuthenticator` MDF を実行すると、MBean インタフェース ファイルの名前が `SampleAuthenticatorMBean.java` になります。

WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する

WebLogic MBeanMaker を使用して MDF を実行して中間ファイルを生成し、MBean 実装ファイルを編集して適切なメソッドの実装を提供したら、カスタム認証プロバイダの MBean ファイルと実行時クラスを MBean JAR ファイル (MJF) にパッケージ化する必要があります。このプロセスも、WebLogic MBeanMaker によって自動化されます。

カスタム認証プロバイダの MJF を作成するには、次の手順を行います。

1. 新しい DOS シェルを作成します。

2. 次のコマンドを入力します。

   ```
   java -DMJF=jarfile -Dfiles=filesdir
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、-DMJF フラグは WebLogic MBeanMaker が新しい MBean タイプを格納する JAR ファイルをビルドする必要があることを示し、*jarfile* は MJF の名前、*filesdir* は WebLogic MBeanMaker で MJF に JAR 化する対象ファイルが存在する場所です。

   この時点でコンパイルが行われるので、エラーが発生するおそれがあります。*jarfile* が指定されていて、エラーが発生しなかった場合には、指定された名前の MJF が作成されます。

注意： 既存の MJF を更新する場合は、MJF をいったん削除してから再生成します。WebLogic MBeanMaker にも -DIncludeSource オプションがあり、それを指定すると、生成される MJF にソース ファイルを含めるかどうかを制御することができます。ソース ファイルには、生成されたソースと MDF そのものがあります。デフォルトは false です。このオプションは、-DMJF を使用しない場合には無視されます。

生成された MJF は、自らの WebLogic Server 環境にインストールすることも、顧客に配布してそれぞれの WebLogic Server 環境にインストールしてもらうこともできます。

WebLogic Server 環境に MBean タイプをインストールする

MBean タイプを WebLogic Server 環境にインストールするには、MJF を *WL_HOME*\server\lib\mbeantypes ディレクトリにコピーします。ここで、*WL_HOME* は WebLogic Server の最上位のインストール ディレクトリです。これで、カスタム認証プロバイダが「デプロイ」されます。つまり、カスタム認証プロバイダが WebLogic Server Administration Console から管理できるようになります。

注意： *WL_HOME*\server\lib\mbeantypes は、MBean タイプをインストールするデフォルトのディレクトリです。ただし、WebLogic Server が追加ディレクトリで MBean タイプを検索するようにするには、サーバを起動するときに -Dweblogic.alternateTypesDirectory=<*dir*> コマンドライン フラグを使用します (<*dir*> はディレクトリ名のカンマ区切りのリスト)。このフラグを使用すると、WebLogic Server は常にまず *WL_HOME*\server\lib\mbeantypes から MBean タイプをロードし、その

後に追加ディレクトリを検索して、(拡張子に関係なく) そのディレクトリに存在するすべての有効なアーカイブをロードします。たとえば `-Dweblogic.alternateTypesDirectory = dirX,dirY` の場合、WebLogic Server はまず *`WL_HOME`*`\server\lib\mbeantypes` から MBean タイプをロードし、その後に、`dirX` および `dirY` に存在する有効なアーカイブをロードします。

追加ディレクトリで MBean タイプを検索するように WebLogic Server に指示し、かつ Java セキュリティ マネージャを使用する場合は、`weblogic.policy` ファイルを更新して、MBean タイプ (したがってカスタム セキュリティ プロバイダも) に対する適切なパーミッションを与える必要があります。詳細については、『WebLogic Security プログラマーズ ガイド』の「Java セキュリティ マネージャを使用しての WebLogic リソースの保護」を参照してください。

非セキュリティ プロバイダの JAR (バックアップ ファイルを含む) は、*`WL_HOME`*`\server\lib\mbeantypes` ディレクトリ以外の場所に置くことをお勧めします。

カスタム認証プロバイダをコンフィグレーションすることによって (3-35 ページの「Administration Console を使用してカスタム認証プロバイダをコンフィグレーションする」を参照)、MBean タイプのインスタンスを作成して、GUI、他の Java コード、または API からそれらの MBean インスタンスを使用することができます。たとえば、WebLogic Server Administration Console を使用して、属性を取得 / 設定したり操作を呼び出したりすることもできますし、他の Java オブジェクトを開発して、そのオブジェクトで MBean をインスタンス化し、それらの MBean から提供される情報に自動的に応答させることもできます。なお、これらの MBean インスタンスをバックアップしておくことをお勧めします。詳細については、『WebLogic Server ドメイン管理』の「障害が発生したサーバの回復」の「セキュリティ コンフィグレーション データのバックアップ」を参照してください。

Administration Console を使用してカスタム認証プロバイダをコンフィグレーションする

カスタム認証プロバイダをコンフィグレーションするということは、そのカスタム認証プロバイダをセキュリティ レルムに追加するということです。追加されたカスタム認証プロバイダには、認証サービスを必要とするアプリケーションからアクセスできます。

カスタム セキュリティ プロバイダのコンフィグレーションは管理タスクですが、カスタム セキュリティ プロバイダの開発者が行うこともできます。この節では、カスタム認証プロバイダのコンフィグレーション担当者にとって重要な情報を提供します。

■ 3-35 ページの「ユーザ ロックアウトの管理」

注意： WebLogic Server Administration Console を使用してカスタム認証プロバイダをコンフィグレーションする手順は、『WebLogic Security の管理』の「カスタム セキュリティ プロバイダのコンフィグレーション」で説明されています。

ユーザ ロックアウトの管理

カスタム認証プロバイダを使用する一環として、ユーザ ロックアウトをコンフィグレーションおよび管理する方法を検討する必要があります。以下の 2 つの選択肢があります。

■ 3-35 ページの「レルムワイドのユーザ ロックアウト マネージャの使用」

■ 3-36 ページの「独自のユーザ ロックアウト マネージャの実装」

レルムワイドのユーザ ロックアウト マネージャの使用

WebLogic Security フレームワークは、WebLogic Security フレームワークと直接連携してユーザ ロックアウトを管理するレルムワイドのユーザ ロックアウト マネージャを提供します。

注意： レルムワイドのユーザ ロックアウト マネージャと WebLogic Server 6.1 `PasswordPolicyMBean` (レルム アダプタ レベル) の両方をアクティブにすることができます。詳細については、WebLogic Server 6.1 API リファレンス Javadoc の PasswordPolicyMBean インタフェースを参照してください。

レルムワイドのユーザ ロックアウト マネージャを使用する場合、それをカスタム認証プロバイダと連携させるためには、WebLogic Server Administration Console を使用して以下のことを行います。

1. ユーザ ロックアウトを有効にします (デフォルトで有効)。
2. 必要に応じてユーザ ロックアウトのパラメータを修正します。

注意： ユーザ ロックアウト マネージャへの変更は、サーバを再起動するまで有効になりません。Administration Console を使用して上記のタスクを実行する手順は、『WebLogic Security の管理』の「ユーザ アカウントの保護」で説明されています。

独自のユーザ ロックアウト マネージャの実装

カスタム認証プロバイダの一部として独自のユーザ ロックアウト マネージャを実装する場合は、次の手順を行う必要があります。

1. レルムワイドのユーザ ロックアウト マネージャを無効にして二重ロックアウトを防止します。WebLogic Server Administration Console を使用して新しいセキュリティ レルムを作成するとユーザ ロックアウト マネージャが必ず作成されます。このタスクの手順については、『WebLogic Security の管理』の「ユーザ アカウントの保護」を参照してください。
2. WebLogic Security フレームワークのレルムワイドの実装からは何も借用できないので、以下のタスクを行うことも必要です。
   a. ユーザ ロックアウト マネージャの実装を提供します。ユーザ ロックアウト マネージャに関して、セキュリティ サービス プロバイダ インタフェース (SSPI) は提供されません。
   b. ユーザ ロックアウト マネージャの管理に使用できる MBean を作成します。

c. ユーザ ロックアウト マネージャをコンフィグレーションするための新しい JavaServer Pages (JSP) を作成し、コンソール拡張機能を使用してそれを Administration Console に組み込みます。詳細については、『Administration Console の拡張』および第 12 章「カスタム セキュリティ プロバイダ用のコンソール拡張の記述」を参照してください。

4　ID アサーション プロバイダ

ID アサーション プロバイダは、ユーザまたはシステム プロセスがトークンを使用してそれぞれの ID を証明する特殊な形態の認証プロバイダです (つまり境界認証)。ID アサーション プロバイダ用の LoginModule を作成すれば、認証プロバイダの代わりに ID アサーション プロバイダを使用できます。また、認証プロバイダの LoginModule を使用する場合は、認証プロバイダに加えて ID アサーション プロバイダも使用できます。ID アサーション プロバイダは境界認証を有効にし、シングル サインオンをサポートします。

以下の節では、ID アサーション プロバイダの概念と機能、およびカスタム ID アサーション プロバイダの開発手順について説明します。

ID アサーションの概念

ID アサーション プロバイダを開発する前に、以下の概念を理解しておく必要があります。

ID アサーション プロバイダと LoginModule

LoginModule と一緒に用いると、ID アサーション プロバイダはシングル サインオンをサポートします。たとえば、ID アサーション プロバイダはデジタル証明書からトークンを生成することができ、そのトークンをシステム内で回すことができるため、ユーザは何度もサインオンを求められることはありません。

ID アサーション プロバイダが使用する LoginModule は、以下のことが可能です。

- 開発したカスタム認証プロバイダの一部となる。詳細については、第 3 章「認証プロバイダ」を参照してください。
- BEA が開発し WebLogic Server と一緒にパッケージ化した WebLogic 認証プロバイダの一部になる。詳細については、3-13 ページの「カスタム認証プロバイダを開発する必要があるか」を参照してください。
- サードパーティのセキュリティ ベンダの認証プロバイダの一部となる。

単純認証の場合と違って (3-12 ページの「認証プロセス」を参照)、ID アサーション プロバイダが使用する LoginModule は証明情報 (ユーザ名とパスワードなど) を検証しません (単にユーザが存在することを検証するだけです)。

注意 : LoginModule の詳細については、3-4 ページの「LoginModule」を参照してください。

ID アサーションとトークン

ID アサーション プロバイダは、ユーザまたはシステム プロセスの ID を断定するために使用する特定のトークン タイプをサポートするために開発します。ID アサーション プロバイダは複数のトークン タイプをサポートするように開発できますが、通常はただ 1 つの「アクティブ」なトークン タイプを検証するようにコンフィグレーションされます。同じトークン タイプを検証する複数の ID アサーション プロバイダを 1 つのセキュリティ レルムに組み込むことも可能ですが、実際に検証を行うのは 1 つの ID アサーション プロバイダだけです。

注意： トークン タイプを「サポート」するということは、ID アサーション プロバイダの実行時クラス (`IdentityAsserter` SSPI 実装) がその `assertIdentity` メソッドでトークン タイプを検証できるということです。詳細については、4-13 ページの「IdentityAsserter SSPI を実装する」を参照してください。

以下の節では、新しいトークン タイプを使用するための手順について説明します。

- 4-3 ページの「新しいトークン タイプの作成方法」
- 4-4 ページの「新しいトークン タイプを ID アサーション プロバイダのコンフィグレーションで利用可能にする方法」

新しいトークン タイプの作成方法

カスタム ID アサーション プロバイダを開発する場合には、新しいトークン タイプも作成できます。**トークン タイプ**とは、文字列で表される 1 つのデータにすぎません。どのようなトークン タイプを作成して使用するかは、完全にユーザに任されています。たとえば、現在、WebLogic ID アサーション プロバイダ用に定義されているトークン タイプは、`X.509`、`CSI.PrincipalName`、`CSI.ITTAnonymous`、`CSI.X509CertChain`、および `CSI.DistinguishedName` です。

新しいトークン タイプを作成するには、コード リスト 4-1 に示すとおり、新しい Java ファイルを作成し、文字列型の変数として新しいトークン タイプを宣言します。`PerimeterIdentityAsserterTokenTypes.java` ファイルでは、`Test 1`、`Test 2`、および `Test 3` というトークン タイプの名前が文字列として定義されています。

コード リスト 4-1 PerimeterIdentityAsserterTokenTypes.java

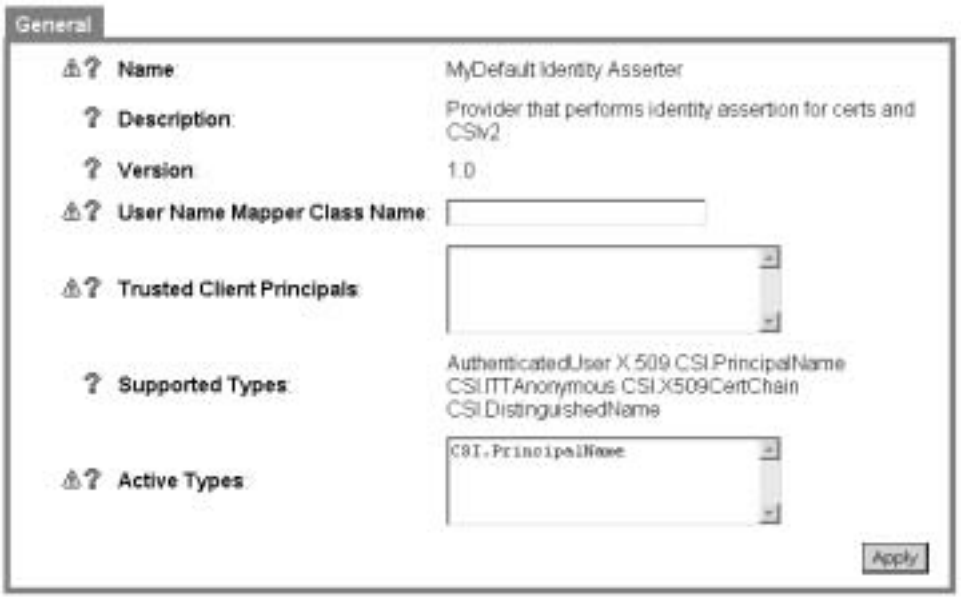
```
package sample.security.providers.authentication.perimeterATN;

public class PerimeterIdentityAsserterTokenTypes
{

   public final static String TEST1_TYPE = "Test 1";
   public final static String TEST2_TYPE = "Test 2";
   public final static String TEST3_TYPE = "Test 3";
}
```

注意： 1 つの新しいトークン タイプを定義するだけの場合は、4-14 ページのコード リスト 4-4「SampleIdentityAsserterProviderImpl.java」で示されているように ID アサーション プロバイダの実行時クラスで行うこともできます。

新しいトークン タイプを ID アサーション プロバイダのコンフィグレーションで利用可能にする方法

カスタム ID アサーション プロバイダをコンフィグレーションする場合には (4-26 ページの「Administration Console を使用してカスタム ID アサーション プロバイダをコンフィグレーションする」を参照)、ID アサーション プロバイダのサポートするトークン タイプのリストが [サポート タイプ] フィールドに表示されます。図 4-1 に示すとおり、サポート タイプの 1 つを、[アクティブなタイプ] フィールドに入力できます。

図 4-1　サンプル ID アサーション プロバイダのコンフィグレーション



[サポート タイプ] フィールドの内容は、カスタム ID アサーション プロバイダの MBean タイプを生成するために使用する MBean 定義ファイル (MDF) の SupportedTypes 属性から取得します。サンプル ID アサーション プロバイダからの例が、コード リスト 4-2 で示されています。MDF および MBean タイプの詳細については、4-18 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」を参照してください。

コード リスト 4-2 SampleIdentityAsserter MDF: SupportedTypes 属性

```
<MBeanType>

...

   <MBeanAttribute
    Name = "SupportedTypes"
    Type = "java.lang.String[]"
    Writeable = "false"
    Default = "new String[] {&quot;SamplePerimeterAtnToken&quot;}"
   />

...

</MBeanType>
```

同様に、[アクティブなタイプ] フィールドの内容も MBean 定義ファイル (MDF) の ActiveTypes 属性から取得します。MDF ファイルの ActiveTypes 属性をデフォルトに設定すると、WebLogic Server Administration Console で手動設定する必要がなくなります。サンプル ID アサーション プロバイダからの例が、コード リスト 4-3 で示されています。

コード リスト 4-3 SampleIdentityAsserter MDF: Default を使用した ActiveTypes 属性

```
<MBeanAttribute
 Name= "ActiveTypes"
 Type= "java.lang.String[]"
 Default = "new String[] { &quot;SamplePerimeterAtnToken&quot; }"
/>
```

ActiveTypes 属性をデフォルト化することは便利ですが、それは他の ID アサーション プロバイダがそのトークン タイプを検証することがない場合のみ行います。その場合以外は、無効なセキュリティ レルムがコンフィグレーションされることになるおそれがあります (複数の ID アサーション プロバイダが同じトークン タイプを検証しようとする)。一番良いのは、ID アサーション プロバイダのすべての MDF がデフォルトでそのトークン タイプをオフにすることです。その場合、そのトークン タイプを検証する ID アサーション プロバイダをコンフィグレーションして手作業でアクティブにすることができます。

注意 : ID アサーション プロバイダが目的のトークン タイプを検証して受け入れるように開発およびコンフィグレーションされていない場合、認証プロセスは失敗します。ID アサーション プロバイダのコンフィグレーションの詳細については、4-26 ページの「Administration Console を使用してカスタム ID アサーション プロバイダをコンフィグレーションする」を参照してください。

境界認証用のトークンを渡す

ID アサーション プロバイダは、境界認証のために Java クライアントからサーブレットにトークンを渡すことができます。トークンは、HTTP ヘッダ、クッキー、SSL 証明書などのメカニズムを用いて渡すことができます。たとえば、HTTP ヘッダを使用すれば、Base64 でコード化された文字列 (バイナリ データの送信が可能) をサーブレットに送信することができます。この文字列の値は、ユーザ名などのユーザの ID を表す文字列です。境界認証に使用される ID アサーション プロバイダは、その文字列を受け取り、そこからユーザ名を抽出できます。

トークンが HTTP ヘッダまたはクッキーを通じて渡される場合には、そのトークンはヘッダまたはクッキーの名前に等しく、リソース コンテナは WebLogic Security フレームワークの認証を処理するパートにトークンを渡します。WebLogic Security フレームワークは、そのトークンをそのまま ID アサーション プロバイダに渡します。

CSIv2 (Common Secure Interoperability Version 2)

WebLogic Server では、IIOP (Internet Inter-ORB) (GIOP バージョン 1.2) および CORBA CSIv2 (Common Secure Interoperability version 2) 仕様に基づいた EJB (Enterprise JavaBean) 相互運用性プロトコルをサポートしています。WebLogic Server で CSIv2 をサポートすることにより、以下のことが可能になります。

- J2EE (Java 2 Enterprise Edition) バージョン 1.3 の参照実装と相互運用できる。
- WebLogic Server IIOP クライアントで T3 クライアントの場合と同じようにユーザ名とパスワードを指定できるようになる。
- GSSAPI (Generic Security Services Application Programming Interface) 初期コンテキスト トークンをサポートできるようになる。今回のリリースでは、ユーザ名 / パスワードと GSSUP (Generic Security Services Username Password) トークンのみがサポートされています。

注意： WebLogic Server における CSIv2 実装は、J2EE CTS (Compatibility Test Suite) 適合性テストに合格しています。

CSIv2 実装への外部インタフェースは、CORBA オブジェクトのユーザ名とパスワードを取得する JAAS LoginModule です。JAAS LoginModule は、WebLogic Java クライアント、あるいは別の J2EE アプリケーション サーバに対するクライアントの役目をする WebLogic Server インスタンスで使用することができます。CSIv2 サポート用の JAAS LoginModule は `UsernamePasswordLoginModule` という名前で、`weblogic.security.auth.login` パッケージに格納されています。

CSIv2 は以下のように動作します。

1. IOR (Interoperable Object Reference) へのセキュリティ拡張機能が作成されると、その中に、CORBA オブジェクトでサポートされるセキュリティ メカニズムを識別するタグ付きコンポーネントが追加されます。このタグ付きコンポーネントには、転送情報、クライアント認証情報、および ID 証明トークン / 認可トークン情報が入っています。
2. クライアントは、IOR 内のセキュリティ メカニズムを評価し、サーバが必要とするオプションをサポートするメカニズムを選択します。

3. クライアントは SAS プロトコルを用いて、WebLogic Server とのセキュリティ コンテキストを確立します。SAS プロトコルでは、リクエストと応答のサービス コンテキスト内に格納されるメッセージが定義されます。コンテキストはステートフルでもステートレスでもかまいません。

CSIv2 の利用については、『WebLogic Security の紹介』の「Common Secure Interoperability Version 2」を参照してください。JAAS LoginModule の詳細については、3-4 ページの「LoginModule」を参照してください。

ID アサーション プロセス

境界認証では、WebLogic Server の外部にあるシステムがトークンを通じて信頼を確立します (これは 3-12 ページの「認証プロセス」で説明されている認証のタイプとは対照的です。その認証では、WebLogic Server がユーザ名とパスワードを通じて信頼を確立します)。ID アサーション プロバイダは、次に示す境界認証プロセスの一部として使用されます (図 4-2 を参照)。

1. WebLogic Server の外部のトークンが、そのタイプのトークンの検証を担当し、「アクティブ」としてコンフィグレーションされている ID アサーション プロバイダに渡されます。
2. そのトークンの有効性が問題なく検証されれば、ID アサーション プロバイダはそのトークンをユーザ名にマップし、そのユーザ名を WebLogic Server に送り返します。その後 WebLogic Server では、3-12 ページの「認証プロセス」で説明した方法で認証プロセスを続行します。具体的には、ユーザ名が JAAS (Java Authentication and Authorization Service) `CallbackHandler` を通じて送信され、コンフィグレーションされている各認証プロバイダの LoginModule に渡されて、サブジェクト内に適切なプリンシパルを格納できるようになります。

図 4-2 境界認証

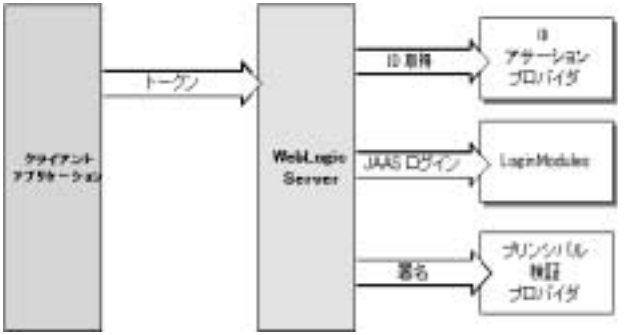


図 4-2 でも示されているように、境界認証では 3-12 ページの「認証プロセス」で説明されている認証プロセスと同じ構成要素が必要ですが、ID アサーション プロバイダも追加されます。

カスタム ID アサーション プロバイダを開発する必要があるか

WebLogic ID アサーション プロバイダでは、X509 証明書を使用した証明書認証および CSIv2 (CORBA Common Secure Interoperability version 2) ID アサーションがサポートされています。

WebLogic ID アサーション プロバイダはトークン タイプを検証し、X509 デジタル証明書と X501 識別名を WebLogic ユーザ名にマップします。また、CSIv2 ID アサーションに使用する信頼性のあるクライアント プリンシパルのリストも指定します。ワイルドカード文字 (*) を使用すると、すべてのプリンシパルに信頼性があると指定できます。クライアントが信頼性のあるクライアント プリンシパルのリストにない場合は、CSIv2 ID アサーションが失敗し、呼び出しが拒否されます。

注意： X.501 および X.509 証明書に対して WebLogic ID アサーション プロバイダを使用するには、`weblogic.security.providers.authentication.UserNameMapper` インタフェースの実装を提供する必要があります。このインタフェースは、

必要に応じた方法に基づいて X.509 証明書を WebLogic Server ユーザ名にマップします。また、このインタフェースを使用して X.501 識別名をユーザ名にマップすることもできます。Administration Console を使用して ID アサーション プロバイダをコンフィグレーションするときに、このインタフェースの実装を指定します。詳細については、『WebLogic Security の管理』の「WebLogic ID アサーション プロバイダでのユーザ名マッパーの使用」を参照してください。

WebLogic ID アサーション プロバイダでは、以下のトークン タイプがサポートされています。

- `AU_TYPE` — WebLogic `AuthenticatedUser` がトークンとして使用される場合に使用します。
- `X509_TYPE` — X509 クライアント証明書がトークンとして使用される場合に使用します。
- `CSI_PRINCIPAL_TYPE` — CSIv2 プリンシパル名 ID がトークンとして使用される場合に使用します。
- `CSI_ANONYMOUS_TYPE` — CSIv2 匿名 ID がトークンとして使用される場合に使用します。
- `CSI_X509_CERTCHAIN_TYPE` — CSIv2 X509 証明書チェーン ID がトークンとして使用される場合に使用します。
- `CSI_DISTINGUISHED_NAME_TYPE` — CSIv2 識別名 ID がトークンとして使用される場合に使用します。

追加的な ID アサーション タスクを実行したい場合、または新しいトークン タイプを作成したい場合は、カスタム ID アサーション プロバイダを開発する必要があります。

カスタム ID アサーション プロバイダの開発方法

WebLogic ID アサーション プロバイダが開発者の要求を満たさない場合、次の手順でカスタム ID アサーション プロバイダを開発することができます。

1. 4-11 ページの「適切な SSPI を使用して実行時クラスを作成する」
2. 4-18 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 4-26 ページの「Administration Console を使用してカスタム ID アサーション プロバイダをコンフィグレーションする」

適切な SSPI を使用して実行時クラスを作成する

実行時クラスを作成する前に、以下の作業が必要です。

- 2-4 ページの「「Provider」SSPI の目的を理解する」
- 2-7 ページの「SSPI 階層を理解し、実行時クラスを 1 つ作成するのか 2 つ作成するのかを決定する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム ID アサーション プロバイダの実行時クラスを作成します。

- 4-11 ページの「AuthenticationProvider SSPI を実装する」
- 4-13 ページの「IdentityAsserter SSPI を実装する」

注意： カスタム ID アサーション プロバイダ用に別の LoginModule を作成する (認証プロバイダの LoginModule を使用しない) 場合は、JAAS `LoginModule` インタフェースも実装する必要があります (3-16 ページの「JAAS LoginModule インタフェースを実装する」を参照)。

カスタム ID アサーション プロバイダの実行時クラスの作成例については、4-14 ページの「例 : サンプル ID アサーション プロバイダの実行時クラスの作成」を参照してください。

AuthenticationProvider SSPI を実装する

`AuthenticationProvider` SSPI を実装するには、2-4 ページの「「Provider」SSPI の目的を理解する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

getLoginModuleConfiguration

```
public AppConfigurationEntry getLoginModuleConfiguration()
```

getLoginModuleConfiguration メソッドは、認証プロバイダの関連付けられた LoginModule に関する情報を取得します。その情報は、AppConfigurationEntry として返されます。AppConfigurationEntry は、LoginModule のクラス名、LoginModule の制御フラグ (認証プロバイダの関連する MBean を通じて渡されていた)、および LoginModule のコンフィグレーション オプション マップ (他のコンフィグレーション情報を LoginModule に渡すことを可能にする) の格納された JAAS (Java Authentication and Authorization Service) クラスです。

AppConfigurationEntry クラス (javax.security.auth.login パッケージ内に定義されている) と LoginModule 用の制御フラグ オプションの詳細については、Java 2 Enterprise Edition, v1.3.1 API 仕様 Javadoc を参照して AppConfigurationEntry クラスと Configuration クラスを調べてください。LoginModule の詳細については、3-4 ページの「LoginModule」を参照してください。セキュリティ プロバイダと MBean の詳細については、2-11 ページの「MBean タイプが必要な理由を理解する」を参照してください。

getAssertionModuleConfiguration

```
public AppConfigurationEntry
getAssertionModuleConfiguration()
```

getAssertionModuleConfiguration メソッドは、ID アサーション プロバイダの関連付けられた LoginModule に関する情報を取得します。その情報は、AppConfigurationEntry として返されます。AppConfigurationEntry は、LoginModule のクラス名、LoginModule の制御フラグ (ID アサーション プロバイダの関連する MBean を通じて渡されていた)、および LoginModule のコンフィグレーション オプション マップ (他のコンフィグレーション情報を LoginModule に渡すことを可能にする) の格納された JAAS クラスです。

getPrincipalValidator

```
public PrincipalValidator getPrincipalValidator()
```

getPrincipalValidator メソッドは、プリンシパル検証プロバイダの実行時クラス (PrincipalValidator SSPI 実装) の参照を取得します。詳細については、第 5 章「プリンシパル検証プロバイダ」を参照してください。

getIdentityAsserter

```
public IdentityAsserter getIdentityAsserter()
```

`getIdentityAsserter` メソッドは、ID アサーション プロバイダの実行時クラス (`IdentityAsserter` SSPI 実装) の参照を取得します。詳細については、4-13 ページの「IdentityAsserter SSPI を実装する」を参照してください。

注意： ID アサーション プロバイダ用の LoginModule が既存の認証プロバイダ用の LoginModule と同じである場合、ID アサーション プロバイダ用の `AuthenticationProvider` SSPI 内のメソッドの実装 (`getIdentityAsserter` メソッドを除く) は `null` を返す場合があります。その例が 4-14 ページのコード リスト 4-4「SampleIdentityAsserterProviderImpl.java」で示されています。

`AuthenticationProvider` SSPI と上記のメソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

IdentityAsserter SSPI を実装する

`IdentityAsserter` SSPI を実装する際には、以下のメソッドの実装を提供する必要があります。

assertIdentity

```
public CallbackHandler assertIdentity(String type, Object
token) throws IdentityAssertionException;
```

`assertIdentity` メソッドは、提供されたトークンの ID 情報に基づいて ID を断定します。言い換えれば、このメソッドの目的は、現時点で「信頼性のない」トークンがあれば、それを信頼性のあるクライアント プリンシパルに照らし合わせて検証することです。`type` パラメータは、ID アサーション (ID の断定) に使用するトークン タイプを表します。ID アサーション タイプには大文字 / 小文字の区別がないことに注意してください。また、`token` パラメータには実際の ID 情報が格納されています。`assertIdentity` から返される `CallbackHandler` は、断定されたユーザ名を格納する必要があり、プリンシパル マッピングを行うためにコンフィグレーションされているすべての認証プロバイダの LoginModule に渡されます。`CallbackHandler` が `null` の場合には、匿名ユーザを使用することを意味します。

CallbackHandler は、可変個の引数を複合オブジェクトとしてメソッドに渡すことができるようにする高度に柔軟な JAAS 規格です。CallbackHandler の詳細については、Java 2 Enterprise Edition, v1.3.1 API 仕様 Javadoc を参照して CallbackHandler インタフェースを調べてください。

IdentityAsserter SSPI と上記のメソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

例：サンプル ID アサーション プロバイダの実行時クラスの作成

コード リスト 注意 : は、サンプル ID アサーション プロバイダの実行時クラスである SampleIdentityAsserterProviderImpl.java クラスを示しています。実行時クラスには以下の実装が含まれています。

- initialize、getDescription、および shutdown という SecurityProvider インタフェースから継承した 3 つのメソッド (2-4 ページの「「Provider」SSPI の目的を理解する」を参照)
- getLoginModuleConfiguration、getAssertionModuleConfiguration、getPrincipalValidator、および getIdentityAsserter という AuthenticationProvider SSPI の 4 つのメソッド (4-11 ページの「AuthenticationProvider SSPI を実装する」を参照)
- assertIdentity という IdentityAsserter SSPI のメソッド (4-13 ページの「IdentityAsserter SSPI を実装する」を参照)

注意 : コード リスト 4-4 の太字のコードは、クラス宣言とメソッド シグネチャを示しています。

コード リスト 4-4 SampleIdentityAsserterProviderImpl.java

```
package examples.security.providers.identityassertion;

import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.AppConfigurationEntry;
import weblogic.management.security.ProviderMBean;
import weblogic.security.spi.AuthenticationProvider;
import weblogic.security.spi.IdentityAsserter;
```

```
import weblogic.security.spi.IdentityAssertionException;
import weblogic.security.spi.PrincipalValidator;
import weblogic.security.spi.SecurityServices;

public final class SampleIdentityAsserterProviderImpl implements
AuthenticationProvider, IdentityAsserter
{
   final static private String TOKEN_TYPE   = "SamplePerimeterAtnToken";
   final static private String TOKEN_PREFIX = "username=";

   private String description;

   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SampleIdentityAsserterProviderImpl.initialize");
      SampleIdentityAsserterMBean myMBean = (SampleIdentityAsserterMBean)mbean;
      description = myMBean.getDescription() + "\n" + myMBean.getVersion();
   }

   public String getDescription()
   {
      return description;
   }

   public void shutdown()
   {
      System.out.println("SampleIdentityAsserterProviderImpl.shutdown");
   }

   public AppConfigurationEntry getLoginModuleConfiguration()
   {
      return null;
   }

   public AppConfigurationEntry getAssertionModuleConfiguration()
   {
      return null;
   }

   public PrincipalValidator getPrincipalValidator()
   {
      return null;
   }

   public IdentityAsserter getIdentityAsserter()
   {
      return this;
   }
```

```
   public CallbackHandler assertIdentity(String type, Object token) throws
    IdentityAssertionException
    {
       System.out.println("SampleIdentityAsserterProviderImpl.assertIdentity");
       System.out.println("\tType\t\t= "  + type);
       System.out.println("\tToken\t\t= " + token);

       if (!(TOKEN_TYPE.equals(type))) {
          String error = "SampleIdentityAsserter received unknown token type \""
             + type + "\"." + " Expected " + TOKEN_TYPE;
          System.out.println("\tError: " + error);
          throw new IdentityAssertionException(error);
       }

       if (!(token instanceof byte[])) {
          String error = "SampleIdentityAsserter received unknown token class \""
             + token.getClass() + "\"." + " Expected a byte[].";
          System.out.println("\tError: " + error);
          throw new IdentityAssertionException(error);
       }

       byte[] tokenBytes = (byte[])token;
       if (tokenBytes == null || tokenBytes.length < 1) {
          String error = "SampleIdentityAsserter received empty token byte array";
          System.out.println("\tError: " + error);
          throw new IdentityAssertionException(error);
       }

       String tokenStr = new String(tokenBytes);

       if (!(tokenStr.startsWith(TOKEN_PREFIX))) {
          String error = "SampleIdentityAsserter received unknown token string \""
             + type + "\"." + " Expected " + TOKEN_PREFIX + "username";
          System.out.println("\tError: " + error);
          throw new IdentityAssertionException(error);
       }

       String userName = tokenStr.substring(TOKEN_PREFIX.length());
       System.out.println("\tuserName\t= " + userName);
       return new SampleCallbackHandlerImpl(userName);
    }

}
```

コード リスト 4-5 は、SampleIdentityAsserterProviderImpl.java 実行時クラスとともに使用する CallbackHandler のサンプル実装を示しています。この CallbackHandler の実装は、認証プロバイダの LoginModule にユーザ名を送り返すために使用します。

コード リスト 4-5 SampleCallbackHandlerImpl.java

```
package examples.security.providers.identityassertion;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;

/*package*/ class SampleCallbackHandler implements CallbackHandler
{
   private String userName;

   /*package*/ SampleCallbackHandlerImpl(String user)
   {
      userName = user;
   }

   public void handle(Callback[] callbacks) throws UnsupportedCallbackException
   {
      for (int i = 0; i < callbacks.length; i++) {

            Callback callback = callbacks[i];

            if (!(callback instanceof NameCallback)) {
               throw new UnsupportedCallbackException(callback, "Unrecognized
                  Callback");
            }

            NameCallback nameCallback = (NameCallback)callback;
            nameCallback.setName(userName);
      }
   }
}
```

WebLogic MBeanMaker を使用して MBean タイプを生成する

カスタム セキュリティ プロバイダの MBean タイプを生成する前に、以下の作業が必要です。

- 2-11 ページの「MBean タイプが必要な理由を理解する」
- 2-12 ページの「拡張および実装する SSPI MBean を決定する」
- 2-13 ページの「MBean 定義ファイル (MDF) の基本的な要素を理解する」
- 2-15 ページの「SSPI MBean の階層と Administration Console に対する影響を理解する」
- 2-17 ページの「WebLogic MBeanMaker によって提供されるものを理解する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム ID アサーション プロバイダの MBean タイプを作成します。

1. 4-18 ページの「MBean 定義ファイル (MDF) を作成する」
2. 4-19 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 4-24 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」
4. 4-25 ページの「WebLogic Server 環境に MBean タイプをインストールする」

注意： これらの手順の実行方法は、いくつかのサンプル セキュリティ プロバイダ (dev2dev Web サイトの「Code Samples: WebLogic Server」で入手可能) に示されています。

この節で説明する手順はすべて、Windows 環境での作業を想定しています。

MBean 定義ファイル (MDF) を作成する

MBean 定義ファイル (MDF) を作成するには、次の手順に従います。

1. サンプル ID アサーション プロバイダの MDF をテキスト ファイルにコピーします。

   注意： サンプル ID アサーション プロバイダの MDF は、`SampleIdentityAsserter.xml` です。

2. MDF で `<MBeanType>` 要素と `<MBeanAttribute>` 要素の内容をカスタム ID アサーション プロバイダに合わせて修正します。

3. カスタム属性および操作 (つまり、`<MBeanAttribute>` および `<MBeanOperation>` 要素) を MDF に追加します。

4. ファイルを保存します。

注意： MDF 要素の構文についての完全なリファレンスは、付録 A「MBean 定義ファイル (MDF) 要素の構文」に収められています。

WebLogic MBeanMaker を使用して MBean タイプを生成する

MDF を作成したら、WebLogic MBeanMaker を使用してそれを実行できます。WebLogic MBeanMaker は現在のところコマンドライン ユーティリティで、入力として MDF を受け取り、MBean インタフェース、MBean 実装、関連する MBean 情報ファイルなどの中間 Java ファイルをいくつか出力します。これらの中間ファイルが合わさって、カスタム セキュリティ プロバイダの **MBean タイプ**になります。

MBean タイプの生成手順は、カスタム ID アサーション プロバイダの設計に応じて異なります。必要な設計に合わせて適切な手順を実行してください。

- 4-19 ページの「任意 SSPI MBean とカスタム操作を追加しない場合」
- 4-20 ページの「任意 SSPI MBean またはカスタム操作を追加する場合」

任意 SSPI MBean とカスタム操作を追加しない場合

カスタム ID アサーション プロバイダの MDF に任意 SSPI MBean もカスタム操作も実装しない場合、次の手順に従います。

1. 新しい DOS シェルを作成します。

2. 次のコマンドを入力します。

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

   `xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

   `-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

   注意 : WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり ID アサーション プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

3. 4-24 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

任意 SSPI MBean またはカスタム操作を追加する場合

カスタム ID アサーション プロバイダの MDF に任意 SSPI MBean またはカスタム操作を実装する場合、以下の質問に答えながら手順を進めてください。

- MBean タイプを作成するのは初めてですか。初めての場合は、次の手順に従ってください。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

   ```
   java -DMDF=xmlfile -DFiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

xmlfile が入力されるたびに、新しい出力ファイル群が生成されます。*filesdir* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

`-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり ID アサーション プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

3. 任意 SSPI MBean を MDF に実装した場合は、次の手順に従います。

   a. MBean 実装ファイルを見つけます。

      WebLogic MBeanMaker によって生成される MBean 実装ファイルには、*MBeanName*`Impl.java` という名前が付けられます。たとえば、`SampleIdentityAsserter` という名前の MDF の場合、編集される MBean 実装ファイルは `SampleIdentityAsserterImpl.java` という名前になります。

   b. MDF で実装した任意 SSPI MBean ごとに、メソッド スタブを「Mapping MDF Operation Declarations to Java Method Signatures Document」(dev2dev Web サイトで入手可能) から MBean 実装ファイルにコピーし、各メソッドを実装します。任意 SSPI MBean が継承するメソッドもすべて実装してください。

4. MDF にカスタム操作を含めた場合、メソッド スタブを使用してメソッドを実装します。
5. ファイルを保存します。
6. 4-24 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

■ 既存の MBean タイプの更新ですか。更新の場合は、次の手順に従ってください。

1. WebLogic MBeanMaker によって現在のメソッドの実装が上書きされないように、既存の MBean 実装ファイルを一時ディレクトリにコピーします。
2. 新しい DOS シェルを作成します。
3. 次のコマンドを入力します。

```
java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
weblogic.management.commo.WebLogicMBeanMaker
```

ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

`xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

`-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意 : WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり ID アサーション プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

4. 任意 SSPI MBean を MDF に実装した場合は、次の手順に従います。

   a. MBean 実装ファイルを見つけて開きます。

      WebLogic MBeanMaker によって生成される MBean 実装ファイルには、*`MBeanName`*`Impl.java` という名前が付けられます。たとえば、`SampleIdentityAsserter` という名前の MDF の場合、編集される MBean 実装ファイルは `SampleIdentityAsserterImpl.java` という名前になります。

   b. 手順 1 で一時ディレクトリに保存した既存の MBean 実装ファイルを開きます。

   c. 既存の MBean 実装ファイルを、WebLogic MBeanMaker によって生成された MBean 実装ファイルと同期させます。

      これには、メソッドの実装を既存の MBean 実装ファイルから新しく生成された MBean 実装ファイルにコピー (または、新しく生成された MBean 実装ファイルから既存の MBean 実装ファイルに新しいメソッドを追加) し、いずれの MBean 実装ファイルにも入っているメソッドのメソッド シグネチャへの変更が使用する MBean 実装ファイルに反映されていることを確認するといった作業が必要です。

d. MDF を修正して元の MDF にはない任意 SSPI MBean を実装した場合は、メソッド スタブを「Mapping MDF Operation Declarations to Java Method Signatures Document」(dev2dev Web サイトで入手可能) から MBean 実装ファイルにコピーし、各メソッドを実装します。任意 SSPI MBean が継承するメソッドもすべて実装してください。

5. MDF を変更して元の MDF にはないカスタム操作を含めた場合、メソッド スタブを使用してメソッドを実装します。
6. 完成した、つまりすべてのメソッドを実装した MBean 実装ファイルを保存します。
7. この MBean 実装ファイルを、WebLogic MBeanMaker が MBean タイプの実装ファイルを配置したディレクトリにコピーします。このディレクトリは、手順 3 で *`filesdir`* として指定しました (手順 3 の結果として WebLogic MBeanMaker で生成された MBean 実装ファイルがオーバーライドされる)。
8. 4-24 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

生成される MBean インタフェース ファイルについて

MBean インタフェース ファイルとは、実行時クラスまたは MBean 実装がコンフィグレーション データを取得するために使用する MBean のクライアントサイド API です。2-4 ページの「「Provider」SSPI の目的を理解する」で説明されているように、これは initialize メソッドで使用するのが一般的です。

WebLogic MBeanMaker では、作成済みの MDF から MBean タイプを生成するので、生成される MBean インタフェース ファイルの名前は、その MDF 名の後に「MBean」というテキストが付いたものになります。たとえば、WebLogic MBeanMaker を使用して `SampleIdentityAsserter` MDF を実行すると、MBean インタフェース ファイルの名前が `SampleIdentityAsserterMBean.java` になります。

WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する

WebLogic MBeanMaker を使用して MDF を実行して中間ファイルを生成し、MBean 実装ファイルを編集して適切なメソッドの実装を提供したら、カスタム ID アサーション プロバイダの MBean ファイルと実行時クラスを MBean JAR ファイル (MJF) にパッケージ化する必要があります。このプロセスも、WebLogic MBeanMaker によって自動化されます。

カスタム ID アサーション プロバイダの MJF を作成するには、次の手順を行います。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

   ```
   java -DMJF=jarfile -Dfiles=filesdir
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMJF` フラグは WebLogic MBeanMaker が新しい MBean タイプを格納する JAR ファイルをビルドする必要があることを示し、*`jarfile`* は MJF の名前、*`filesdir`* は WebLogic MBeanMaker で MJF に JAR 化する対象ファイルが存在する場所です。

   この時点でコンパイルが行われるので、エラーが発生するおそれがあります。*`jarfile`* が指定されていて、エラーが発生しなかった場合には、指定された名前の MJF が作成されます。

注意： 既存の MJF を更新する場合は、MJF をいったん削除してから再生成します。WebLogic MBeanMaker にも `-DIncludeSource` オプションがあり、それを指定すると、生成される MJF にソース ファイルを含めるかどうかを制御することができます。ソース ファイルには、生成されたソースと MDF そのものがあります。デフォルトは `false` です。このオプションは、`-DMJF` を使用しない場合には無視されます。

生成された MJF は、自らの WebLogic Server 環境にインストールすることも、顧客に配布してそれぞれの WebLogic Server 環境にインストールしてもらうこともできます。

WebLogic Server 環境に MBean タイプをインストールする

MBean タイプを WebLogic Server 環境にインストールするには、MJF を *WL_HOME*\server\lib\mbeantypes ディレクトリにコピーします。ここで、*WL_HOME* は WebLogic Server の最上位のインストール ディレクトリです。これで、カスタム ID アサーション プロバイダが「デプロイ」されます。つまり、カスタム ID アサーション プロバイダが WebLogic Server Administration Console から管理できるようになります。

注意： *WL_HOME*\server\lib\mbeantypes は、MBean タイプをインストールするデフォルトのディレクトリです。ただし、WebLogic Server が追加ディレクトリで MBean タイプを検索するようにするには、サーバを起動するときに -Dweblogic.alternateTypesDirectory=<*dir*> コマンドライン フラグを使用します (<*dir*> はディレクトリ名のカンマ区切りのリスト)。このフラグを使用すると、WebLogic Server は常にまず *WL_HOME*\server\lib\mbeantypes から MBean タイプをロードし、その後に追加ディレクトリを検索して、(拡張子に関係なく) そのディレクトリに存在するすべての有効なアーカイブをロードします。たとえば -Dweblogic.alternateTypesDirectory = dirX,dirY の場合、WebLogic Server はまず *WL_HOME*\server\lib\mbeantypes から MBean タイプをロードし、その後に、dirX および dirY に存在する有効なアーカイブをロードします。

追加ディレクトリで MBean タイプを検索するように WebLogic Server に指示し、かつ Java セキュリティ マネージャを使用する場合は、weblogic.policy ファイルを更新して、MBean タイプ (したがってカスタム セキュリティ プロバイダも) に対する適切なパーミッションを与える必要があります。詳細については、『WebLogic Security プログラマーズ ガイド』の「Java セキュリティ マネージャを使用しての WebLogic リソースの保護」を参照してください。

非セキュリティ プロバイダの JAR (バックアップ ファイルを含む) は、*WL_HOME*\server\lib\mbeantypes ディレクトリ以外の場所に置くことをお勧めします。

カスタム ID アサーション プロバイダをコンフィグレーションすることによって (4-26 ページの「Administration Console を使用してカスタム ID アサーション プロバイダをコンフィグレーションする」を参照) MBean タイプのインスタンスを作成して、GUI、他の Java コード、または API からそれらの MBean インスタ

ンスを使用することができます。たとえば、WebLogic Server Administration Console を使用して、属性を取得 / 設定したり操作を呼び出したりすることもできますし、他の Java オブジェクトを開発して、そのオブジェクトで MBean をインスタンス化し、それらの MBean から提供される情報に自動的に応答させることもできます。なお、これらの MBean インスタンスをバックアップしておくことをお勧めします。詳細については、『WebLogic Server ドメイン管理』の「障害が発生したサーバの回復」の「セキュリティ コンフィグレーション データのバックアップ」を参照してください。

Administration Console を使用してカスタム ID アサーション プロバイダをコンフィグレーションする

カスタム ID アサーション プロバイダをコンフィグレーションするということは、そのカスタム ID アサーション プロバイダをセキュリティ レルムに追加するということです。追加されたカスタム ID アサーション プロバイダには、ID 断定サービスを必要とするアプリケーションからアクセスできます。

カスタム セキュリティ プロバイダのコンフィグレーションは管理タスクですが、カスタム セキュリティ プロバイダの開発者が行うこともできます。

注意： WebLogic Server Administration Console を使用してカスタム ID アサーション プロバイダをコンフィグレーションする手順は、『WebLogic Security の管理』の「カスタム セキュリティ プロバイダのコンフィグレーション」で説明されています。

5　プリンシパル検証プロバイダ

認証プロバイダは、プリンシパル検証プロバイダを利用して、サブジェクトに格納されるプリンシパル (ユーザとグループ) に署名し、信頼性を検証します。こうした検証によって、信頼度は向上し、悪意のあるプリンシパルによる改ざんの可能性を低くすることができます。サブジェクトのプリンシパルの検証は、WebLogic Server で各呼び出しの RMI クライアント リクエストがデマーシャリングされる際に行われます。サブジェクトのプリンシパルの信頼性も、認可判定の際に検証されます。

以下の節では、プリンシパル検証プロバイダの概念と機能、およびカスタム プリンシパル検証プロバイダの開発手順について説明します。

プリンシパル検証の概念

プリンシパル検証プロバイダを開発する前に、以下の概念を理解しておく必要があります。

プリンシパル検証とプリンシパル タイプ

ID アサーション プロバイダが特定のタイプのトークンをサポートするように、プリンシパル検証プロバイダも特定のタイプのプリンシパルをサポートします。たとえば、WebLogic プリンシパル検証プロバイダ (5-4 ページの「カスタム プリンシパル検証プロバイダを開発する必要があるか」を参照) は WebLogic Server プリンシパルの署名と信頼性の検証を行います。

コンフィグレーション済みの認証プロバイダに関連付けられているプリンシパル検証プロバイダ (5-2 ページの「プリンシパル検証プロバイダと他のタイプのセキュリティ プロバイダの違い」を参照) は、サブジェクトに格納され、そのプリンシパル検証プロバイダがサポートするタイプのすべてのプリンシパルに対して署名と検証を行います。

プリンシパル検証プロバイダと他のタイプのセキュリティ プロバイダの違い

プリンシパル検証プロバイダは、主に認証プロバイダの「ヘルパー」として機能する特殊なセキュリティ プロバイダです。プリンシパル検証プロバイダの主要な機能は、サブジェクトに格納されたプリンシパルが悪意のある個人によって改ざんされるのを防止することです。

`AuthenticationProvider` SSPI (3-14 ページの「AuthenticationProvider SSPI を実装する」を参照) には、`getPrincipalValidator` というメソッドが含まれています。このメソッドでは、認証プロバイダと共に使用するプリンシパル検証プロバイダの実行時クラスを指定します。プリンシパル検証プロバイダの実行時クラスとしては、BEA 提供のもの (WebLogic プリンシパル検証プロバイダ) か、独自に開発したもの (カスタム プリンシパル検証プロバイダ) を使用できます。認証プロバイダの `getPrincipalValidator` メソッドで WebLogic プリンシパル検証プロバイダを使用する例については、3-20 ページのコード リスト 3-1「SampleAuthenticationProviderImpl.java」を参照してください。

WebLogic Server Administration Console を使用して認証プロバイダの MBean タイプを生成し、認証プロバイダをコンフィグレーションするので、プリンシパル検証プロバイダでそれらの手順を行う必要はありません。

無効なプリンシパルが原因のセキュリティ例外

認証 (または認可) 処理を行うときに、WebLogic Security フレームワークはサブジェクトのプリンシパルを調べてそれが有効かどうかを確認します。プリンシパルが無効の場合、WebLogic Security フレームワークはサブジェクトが無効であることを示すテキストを付けてセキュリティ例外を送出します。サブジェクトが無効になる原因には以下のものがあります。

- サブジェクトのプリンシパルに対応するプリンシパル検証プロバイダがコンフィグレーションされていない (つまり WebLogic Security フレームワークでそのサブジェクトを検証する手段がない)

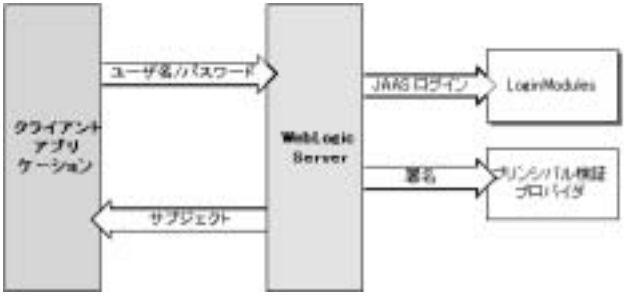
  注意： サブジェクトには複数のプリンシパルを格納できるため (それぞれ異なる認証プロバイダの LoginModule によって格納される)、プリンシパルは複数の異なるプリンシパル検証プロバイダを持つことができます。

- このセキュリティドメインとは異なる資格を使用する別の WebLogic Server セキュリティドメインで署名されたプリンシパルがあり、呼び出し側がそれを現在のドメインで使用しようとしている
- セキュリティを低下させる目的で作成された無効な署名を持つプリンシパルがある
- プリンシパルが署名されていないサブジェクトがある

プリンシパル検証プロセス

図 5-1 に示すとおり、ユーザはユーザ名とパスワードの組み合わせを使ってシステムにログインしようとします。WebLogic Server は、コンフィグレーション済みの認証プロバイダの LoginModule を呼び出すことによって信頼を確立します。LoginModule は、ユーザのユーザ名とパスワードを検証し、JAAS (Java Authentication and Authorization Service) の要件に従って、プリンシパルが格納されたサブジェクトを返します。

図 5-1　プリンシパル検証プロセス



WebLogic Server は、指定されたプリンシパル検証プロバイダにサブジェクトを渡します。プリンシパル検証プロバイダは、そのプリンシパルに署名して、それらを WebLogic Server を通じてクライアント アプリケーションに返します。他のセキュリティ操作のためにサブジェクト内のプリンシパルが必要になった場合、同じプリンシパル検証プロバイダが、そのプリンシパルが署名時から変更されていないかどうかを検証します。

カスタム プリンシパル検証プロバイダを開発する必要があるか

WebLogic Server のデフォルト (アクティブ) セキュリティ レルムには、WebLogic プリンシパル検証プロバイダが含まれます。ID アサーション プロバイダが特定のタイプのトークンをサポートするのと同じように、プリンシパル検証プロバイダは特定のタイプのプリンシパルに対する署名と信頼性の確認を行います。WebLogic プリンシパル検証プロバイダは、WebLogic Server プリンシパルの署名と検証を行います。言い換えると、WebLogic Server ユーザまたは WebLogic Server グループを表すプリンシパルに対して署名と検証を行います。

注意： `weblogic.security` パッケージの `WLSPrincipals` クラスを使用すると、プリンシパル (ユーザまたはグループ) が WebLogic Server に対して特別な意味を持っているかどうか (それが定義済みの WebLogic Server ユーザまたは WebLogic Server グループであるかどうか) が分かります。さら

に、WebLogic Server ユーザまたはグループを表すプリンシパルは、`weblogic.security.spi` パッケージの `WLSUser` インタフェースと `WLSGroup` インタフェースを実装する必要があります。

WebLogic プリンシパル検証プロバイダには、`WLSUserImpl` および `WLSGroupImpl` という名前の `WLSUser` インタフェースおよび `WLSGroup` インタフェースの実装が含まれています。それらは、`weblogic.security.principal` パッケージに定義されています。また、`PrincipalValidatorImpl` という名前の `PrincipalValidator` SSPI の実装も含まれます。これは、`weblogic.security.provider` パッケージに定義されています。`PrincipalValidatorImpl` クラスの `sign()` メソッドは、ランダム シードを生成し、そのランダム シードに基づいてダイジェストを計算します。`PrincipalValidator` SSPI の詳細については、5-6 ページの「PrincipalValidator SSPI を実装する」を参照してください。

WebLogic プリンシパル検証プロバイダの使い方

名前が付いているだけの単純なユーザおよびグループ プリンシパルがある状態で、WebLogic プリンシパル検証プロバイダを使用する場合には、以下の方法があります。

- `weblogic.security.principal.WLSUserImpl` クラスと `weblogic.security.principal.WLSGroupImpl` クラスを使用する
- `weblogic.security.provider.PrincipalValidatorImpl` クラスを使用する

名前の他に追加のデータ メンバーがあるユーザおよびグループ プリンシパルがある状態で、WebLogic プリンシパル検証プロバイダを使用する場合には、以下の方法があります。

- 独自の `UserImpl` クラスと `GroupImpl` クラスを記述する
- `weblogic.security.principal.WLSAbstractPrincipal` クラスを拡張する
- `weblogic.security.spi.WLSUser` インタフェースと `weblogic.security.spi.WLSGroup` インタフェースを実装する

- `equals()` メソッドを実装して、追加のデータ メンバーを含める。実装では、`WLSAbstractPrincipal` が残りのデータを検証できるように、完了時に `super.equals()` メソッドを呼び出す必要があります。

  注意： デフォルトでは、ユーザ名とグループ名だけが検証されます。追加のデータ メンバーを検証する場合は、続いて `getSignedData()` メソッドを実装します。

- `weblogic.security.provider.PrincipalValidatorImpl` クラスを使用する

独自の検証スキームがあり、WebLogic プリンシパル検証プロバイダを使用しない場合や、WebLogic Server プリンシパル以外のプリンシパルを検証する場合は、カスタム プリンシパル検証プロバイダを開発する必要があります。

カスタム プリンシパル検証プロバイダの開発方法

カスタム プリンシパル検証プロバイダを開発するには、以下の方法があります。

- 以下の実装を行うことによって、独自の `UserImpl` クラスと `GroupImpl` クラスを記述する
  - `weblogic.security.spi.WLSUser` インタフェースと `weblogic.security.spi.WLSGroup` インタフェースの実装
  - `java.io.Serializable` インタフェースの実装
- `weblogic.security.spi.PrincipalValidator` SSPI を実装することによって、独自の `PrincipalValidationImpl` クラスを記述する (5-6 ページの「PrincipalValidator SSPI を実装する」を参照)

PrincipalValidator SSPI を実装する

`PrincipalValidator` SSPI を実装するには、以下のメソッドの実装を提供する必要があります。

validate

```
public boolean validate(Principal principal) throws
SecurityException;
```

validate メソッドはプリンシパルを 1 つ引数に取り、その有効性を検証しようとします。すなわち、このメソッドは、そのプリンシパルが署名されてから変更されていないかどうかを確認します。

sign

```
public boolean sign(Principal principal);
```

sign メソッドはプリンシパルを 1 つ引数に取り、それに署名して信頼を保証します。これにより、validate メソッドを使って、後でそのプリンシパルを検証できるようになります。

sign メソッドの実装は、悪意のある個人が容易に再現することのできない機密アルゴリズムでなければなりません。そのアルゴリズムを sign メソッド自体に組み込むか、sign メソッドがプリンシパルの署名に使用するトークンをサーバに要求するようにするか、またはプリンシパルを署名する他の手段を実装できます。

getPrincipalBaseClass

```
public Class getPrincipalBaseClass();
```

getPrincipalBaseClass メソッドは、このプリンシパル検証プロバイダが検証と署名の方法を理解しているプリンシパルの基本クラスを返します。

PrincipalValidator SSPI と上記のメソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

6　認可プロバイダ

認可とは、ユーザの ID などの情報に基づいて、ユーザと WebLogic リソースとのやり取りを管理するプロセスのことです。言い換えれば、認可とは「自分は何にアクセスできますか」という質問に答えるものです。WebLogic Server では、認可プロバイダはユーザと WebLogic リソースとの対話を制限して、整合性、機密性、および可用性を確保します。

以下の節では、認可プロバイダの概念と機能、およびカスタム認可プロバイダの開発手順について説明します。

認可の概念

認可プロバイダを開発する前に、以下の概念を理解しておく必要があります。

アクセス決定

認証プロバイダの LoginModule と同じく、**アクセス決定**は、「アクセスは許可されるのか」という質問に答える認可プロバイダのコンポーネントです。具体的には、指定された操作を WebLogic リソースに対して実行するパーミッションをサ

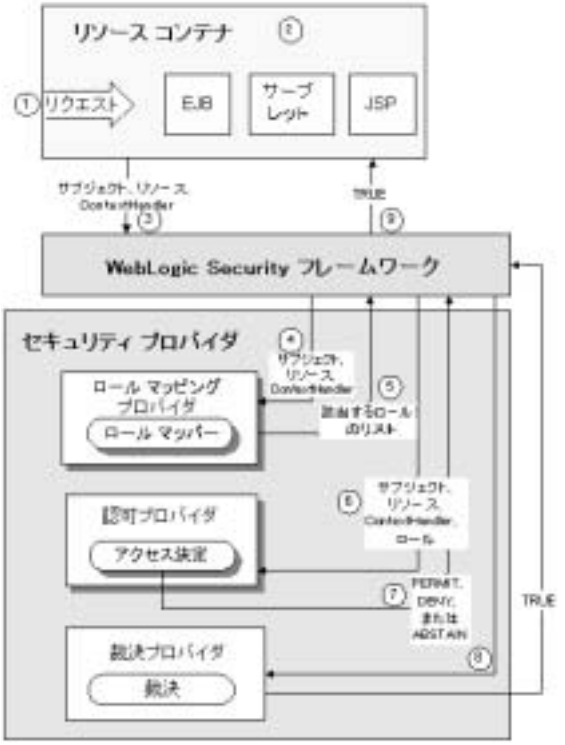
ブジェクトが持っているかどうかが、アプリケーション内の特定のパラメータを用いてアクセス決定に質問されます。この情報が与えられると、アクセス決定はそれに対する回答を PERMIT、DENY、または ABSTAIN のいずれかで返します。

注意： アクセス決定の詳細については、6-8 ページの「AccessDecision SSPI を実装する」を参照してください。

認可プロセス

図 6-1 に、認可プロセスにおける認可プロバイダ (および関連する裁決プロバイダとロール マッピング プロバイダ) と WebLogic Security フレームワークとの対話を示します。その後、この図について説明します。

図 6-1　認可プロバイダと認可プロセス



一般に、認可は以下のように実行されます。

1. ユーザまたはシステム プロセスが WebLogic リソースを要求し、特定の操作を実行しようとします。
2. 要求された WebLogic リソースのタイプを処理するリソース コンテナがリクエストを受け取ります。たとえば EJB コンテナは EJB リソースに対するリクエストを受け取ります。

   注意： リソース コンテナは、2-24 ページの「セキュリティ プロバイダと WebLogic リソース」で説明されている WebLogic リソースのいずれかを処理するコンテナです。

3. リソース コンテナは、ContextHandler オブジェクトを作成します。このオブジェクトは、コンフィグレーション済みのロール マッピング プロバイダと認可プロバイダのアクセス決定がそのリクエストのコンテキストに関連付けられている情報を取得するために使用できます。

   注意： ContextHandler の詳細については、2-35 ページの「ContextHandler と WebLogic リソース」を参照してください。アクセス決定の詳細については、6-1 ページの「アクセス決定」を参照してください。ロール マッピング プロバイダの詳細については、第 8 章「ロール マッピング プロバイダ」を参照してください。

   リソース コンテナは、サブジェクト、リソース、そして場合によっては決定用の追加入力を提供する ContextHandler オブジェクトを渡して WebLogic Security フレームワークを呼び出します。

4. WebLogic Security フレームワークは、コンフィグレーション済みのロール マッピング プロバイダを呼び出します。

5. ロール マッピング プロバイダは ContextHandler を用いて、リクエストに関するさまざまな情報を要求します。また、ロール マッピング プロバイダは、要求する情報のタイプを表す一連の Callback オブジェクトを作成します。その Callback オブジェクト群は、handle メソッドを通じて、配列として ContextHandler に渡されます。

   ロール マッピング プロバイダは、Callback オブジェクトに格納されている値、サブジェクト、およびリソースを用いて、リクエスト元のサブジェクトに付与されるロールを計算し、該当するロールを WebLogic Security フレームワークに渡します。

6. WebLogic Security は、要求されたアクションをリソースに対して実行する資格がサブジェクトにあるかどうかに関する実際の判定を認可プロバイダに委託します。

   認可プロバイダのアクセス決定ではまた、ContextHandler を用いてリクエストに関するさまざまな情報を要求します。アクセス決定は、要求する情報のタイプを表す一連の Callback オブジェクトを作成します。その Callback オブジェクト群は、handle メソッドを通じて、配列として ContextHandler に渡されます。このプロセスは、手順 5 のロール マッピング プロバイダの場合と同じです。

7. コンフィグレーション済みの各認可プロバイダのアクセス決定の `isAccessAllowed()` メソッドが呼び出され、要求されたアクセスを実行する権限がサブジェクトにあるかどうかが、`ContextHandler`、サブジェクト、リソース、およびロールに基づいて判定されます。各 `isAccessAllowed()` メソッドは、以下の 3 つの値のいずれかを返します。
   - `PERMIT` — 要求されたアクセスが許可されることを示す
   - `DENY` — 要求されたアクセスが明示的に拒否されることを示す
   - `ABSTAIN` — 明示的な判定をアクセス決定が下せなかったことを示す

   このプロセスは、すべてのアクセス決定が使用されるまで続きます。
8. WebLogic Security フレームワークは、コンフィグレーション済みの認可プロバイダのアクセス決定から返された判定結果に食い違いがあった場合、その調停を裁決プロバイダに委託します。裁決プロバイダでは、認可判定の最終結果を決定します。

   注意： 裁決プロバイダの詳細については、第 7 章「裁決プロバイダ」を参照してください。
9. 裁決プロバイダは `TRUE` か `FALSE` の判定を返し、その判定は WebLogic Security フレームワークを通してリソース コンテナに転送されます。
   - 判定が `TRUE` の場合、リソース コンテナはリクエストを保護対象リソースにディスパッチします。
   - 判定が `FALSE` の場合、リソース コンテナはセキュリティ例外を送出します。この例外は、リクエスト元には要求したアクセスを保護対象リソースに対して行う権限がなかったことを示します。

カスタム認可プロバイダを開発する必要があるか

WebLogic Server のデフォルト (つまりアクティブな) セキュリティ レルムには WebLogic 認可プロバイダが含まれています。WebLogic 認可プロバイダは、このバージョンの WebLogic Server の認可機能をデフォルトで適用するものです。WebLogic 認可プロバイダは、特定のユーザが保護対象の WebLogic リソースへのアクセスを許可されているかどうかを判定するためのポリシーベースの認可エ

ンジンを使用して、アクセス決定を返します。また、WebLogic 認可プロバイダは、システム内のセキュリティ ポリシーのデプロイメントとアンデプロイメントもサポートしています。自社の既存の認可メカニズムを使用する場合は、カスタム認可プロバイダを作成してそれを既存のメカニズムに結合できます。

カスタム認可プロバイダの開発方法

WebLogic 認可プロバイダが開発者の要求を満たさない場合、次の手順でカスタム認可プロバイダを開発することができます。

1. 6-6 ページの「適切な SSPI を使用して実行時クラスを作成する」
2. 6-14 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 6-22 ページの「Administration Console を使用してカスタム認可プロバイダをコンフィグレーションする」
4. 6-26 ページの「セキュリティ ポリシーを管理するためのメカニズムを提供する」

適切な SSPI を使用して実行時クラスを作成する

実行時クラスを作成する前に、以下の作業が必要です。

- 2-4 ページの「「Provider」SSPI の目的を理解する」
- 2-5 ページの「実装する「Provider」インタフェースを決定する」
- 2-7 ページの「SSPI 階層を理解し、実行時クラスを 1 つ作成するのか 2 つ作成するのかを決定する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム認可プロバイダの実行時クラスを作成します。

- 6-7 ページの「AuthorizationProvider SSPI を実装する」または 6-8 ページの「DeployableAuthorizationProvider SSPI を実装する」

- 6-8 ページの「AccessDecision SSPI を実装する」

注意： セキュリティ レルムでは、少なくとも 1 つの認可プロバイダが DeployableAuthorizationProvider SSPI を実装する必要があり、そうでなければ、Web アプリケーションや EJB をデプロイすることが不可能になります。

カスタム認可プロバイダの実行時クラスの作成例については、6-10 ページの「例 : サンプル認可プロバイダの実行時クラスの作成」を参照してください。

AuthorizationProvider SSPI を実装する

AuthorizationProvider SSPI を実装するには、2-4 ページの「「Provider」SSPI の目的を理解する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

getAccessDecision

```
public AccessDecision getAccessDecision();
```

getAccessDecision メソッドは、AccessDecision SSPI の実装を取得します。MyAuthorizationProviderImpl.java という 1 つの実行時クラスの場合、getAccessDecision メソッドの実装は次のようになります。

```
return this;
```

実行時クラスが 2 つの場合、getAccessDecision メソッドの実装は次のようになります。

```
return new MyAccessDecisionImpl;
```

この理由は、AuthorizationProvider SSPI を実装する実行時クラスが、AccessDecision SSPI を実装するクラスを取得するためのファクトリとして使用されるからです。

AuthorizationProvider SSPI および getAccessDecision メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

DeployableAuthorizationProvider SSPI を実装する

DeployableAuthorizationProvider SSPI を実装するには、2-4 ページの「「Provider」SSPI の目的を理解する」および 6-7 ページの「AuthorizationProvider SSPI を実装する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

deployPolicy

```
public void deployPolicy(Resource resource,
java.lang.String[] roleNames) throws
ResourceCreationException
```

deployPolicy メソッドは、ポリシーの適用対象となるリソースとポリシー内に記載されるロール名を基に、デプロイ済みの Web アプリケーション や EJB に代わってポリシーを作成します。

undeployPolicy

```
public void undeployPolicy(Resource resource) throws
ResourceRemovalException
```

undeployPolicy メソッドは、ポリシーの適用対象であったリソースを基に、アンデプロイされた Web アプリケーション や EJB に代わってポリシーを削除します。

DeployableAuthorizationProvider SSPI および deployPolicy メソッドと undeployPolicy メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

AccessDecision SSPI を実装する

AccessDecision SSPI を実装する際には、以下のメソッドの実装を提供する必要があります。

isAccessAllowed

```
public Result isAccessAllowed(Subject subject, Map roles,
Resource resource, ContextHandler handler, Direction
direction) throws InvalidPrincipalException
```

isAccessAllowed メソッドは、サブジェクトに格納されている情報を利用して、リクエスト元に保護対象リソースへのアクセスを許可すべきかどうかを決定します。isAccessAllowed メソッドはリクエストの前または後に呼び出すことができ、PERMIT、DENY、または ABSTAIN のい

ずれかの値を返します。複数のアクセス決定がコンフィグレーションされていて、それらが相反する値を返す場合、最終結果を決定するには裁決プロバイダが必要になります。詳細については、第 7 章「裁決プロバイダ」を参照してください。

isProtectedResource

```
public boolean isProtectedResource(Subject subject, Resource
resource) throws InvalidPrincipalException
```

`isProtectedResource` メソッドは、指定された WebLogic リソースが保護対象かどうかを実際にアクセス チェックを行わずに決定するために使用します。これは呼び出し側のサブジェクトに付与できる一連のロールを計算するわけではないので、軽量のメカニズムにすぎません。

`AccessDecision` SSPI および `isAccessAllowed` メソッドと `isProtectedResource` メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

レルム アダプタ認証プロバイダと互換性のあるカスタム認可プロバイダの開発

認証プロバイダは、サブジェクト内へのユーザおよびグループの格納を担当するセキュリティ プロバイダです。ユーザおよびグループは、認可プロバイダなどの他のタイプのセキュリティ プロバイダによってサブジェクトから抽出されます。セキュリティ レルムでコンフィグレーションされている認証プロバイダがレルム アダプタ認証プロバイダである場合、ユーザおよびグループの情報は、他の認証プロバイダとは若干異なる方法でサブジェクトに格納されます。そのため、こうしたユーザおよびグループの情報の抽出も、若干異なる方法で行う必要があります。

サブジェクトへの格納にレルム アダプタ認証プロバイダが使用された場合に、サブジェクトがユーザ名またはグループ名に一致するかどうかを調べるためにカスタム認可プロバイダで使用できるコードをコード リスト 6-1 に示します。このコードは、`isAccessAllowed` メソッドおよび `isProtectedResource` メソッドに記述できます。

コード リスト 6-1　サブジェクトがユーザ名またはグループ名に一致するかどうかを調べるためのサンプル コード

```
/**
 * サブジェクトがユーザ名またはグループ名に一致するかどうかを調べる
 *
 * @param principalWant このロールのプリンシパル名を含む文字列
 * ( つまり、ロール定義 )
 *
 * @param subject リソースにアクセスしようとしているユーザおよびそのユーザのグループを
 * 識別するプリンシパルを含むサブジェクト
 *
 * @return ブール値。現在のサブジェクトがロールのプリンシパルに
 * 一致する場合は true、それ以外の場合は false
 */
private boolean subjectMatches(String principalWant, Subject subject)
{
   // まず、グループ名が一致しているかどうかを調べる
   if (SubjectUtils.isUserInGroup(subject, principalWant)) {
      return true;
   }
   // 次に、ユーザ名が一致しているかどうかを調べる
   if (principalWant.equals(SubjectUtils.getUsername(subject))) {
      return true;
   }
   // 一致しなかった場合
   return false;
}
```

例：サンプル認可プロバイダの実行時クラスの作成

コード リスト 6-2 は、サンプル認可プロバイダの実行時クラスである SampleAuthorizationProviderImpl.java クラスを示しています。実行時クラスには以下の実装が含まれています。

- initialize、getDescription、および shutdown という SecurityProvider インタフェースから継承した 3 つのメソッド (2-4 ページの「「Provider」SSPI の目的を理解する」を参照)
- getAccessDecision という AuthorizationProvider SSPI から継承したメソッド (6-7 ページの「AuthorizationProvider SSPI を実装する」を参照)

- `deployPolicy` および `undeployPolicy` という `DeployableAuthorizationProvider` SSPI の 2 つのメソッド (6-8 ページの「DeployableAuthorizationProvider SSPI を実装する」を参照)
- `isAccessAllowed` および `isProtectedResource` という `AccessDecision` SSPI の 2 つのメソッド (6-8 ページの「AccessDecision SSPI を実装する」を参照)

注意 : コード リスト 6-2 の太字のコードは、クラス宣言とメソッド シグネチャを示しています。

コード リスト 6-2 SampleAuthorizationProviderImpl.java

```
package examples.security.providers.authorization;

import java.security.Principal;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import javax.security.auth.Subject;
import weblogic.management.security.ProviderMBean;
import weblogic.security.WLSPrincipals;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.AccessDecision;
import weblogic.security.spi.DeployableAuthorizationProvider;
import weblogic.security.spi.Direction;
import weblogic.security.spi.InvalidPrincipalException;
import weblogic.security.spi.Resource;
import weblogic.security.spi.ResourceCreationException;
import weblogic.security.spi.ResourceRemovalException;
import weblogic.security.spi.Result;
import weblogic.security.spi.SecurityServices;

public final class SampleAuthorizationProviderImpl implements
DeployableAuthorizationProvider, AccessDecision
{
   private String description;
   private SampleAuthorizerDatabase database;

   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SampleAuthorizationProviderImpl.initialize");
      SampleAuthorizerMBean myMBean = (SampleAuthorizerMBean)mbean;
      description = myMBean.getDescription() + "\n" + myMBean.getVersion();
```

```
      database = new SampleAuthorizerDatabase(myMBean);
   }

   public String getDescription()
   {
      return description;
   }

   public void shutdown()
   {
      System.out.println("SampleAuthorizationProviderImpl.shutdown");
   }

   public AccessDecision getAccessDecision()
   {
      return this;
   }

   public Result isAccessAllowed(Subject subject, Map roles, Resource resource,
   ContextHandler handler, Direction direction) throws InvalidPrincipalException
   {
      System.out.println("SampleAuthorizationProviderImpl.isAccessAllowed");
      System.out.println("\tsubject\t= " + subject);
      System.out.println("\troles\t= " + roles);
      System.out.println("\tresource\t= " + resource);
      System.out.println("\tdirection\t= " + direction);

      Set principals = subject.getPrincipals();

      for (Resource res = resource; res != null; res = res.getParentResource()) {
         if (database.policyExists(res)) {
            return isAccessAllowed(res, principals, roles);
         }
      }
      return Result.ABSTAIN;
   }

   public boolean isProtectedResource(Subject subject, Resource resource) throws
   InvalidPrincipalException
   {
      System.out.println("SampleAuthorizationProviderImpl.
        isProtectedResource");
      System.out.println("\tsubject\t= " + subject);
      System.out.println("\tresource\t= " + resource);

      for (Resource res = resource; res != null; res = res.getParentResource()) {
         if (database.policyExists(res)) {
            return true;
         }
      }
```

```
   return false;
}

public void deployPolicy(Resource resource, String[] roleNamesAllowed)
throws ResourceCreationException
{
   System.out.println("SampleAuthorizationProviderImpl.deployPolicy");
   System.out.println("\tresource\t= " + resource);

   for (int i = 0; roleNamesAllowed != null && i < roleNamesAllowed.length;
    i++) {
      System.out.println("\troleNamesAllowed[" + i + "]\t= " +
      roleNamesAllowed[i]);
   }
   database.setPolicy(resource, roleNamesAllowed);
}

public void undeployPolicy(Resource resource) throws ResourceRemovalException
{
   System.out.println("SampleAuthorizationProviderImpl.undeployPolicy");
   System.out.println("\tresource\t= " + resource);

   database.removePolicy(resource);
}

private boolean principalsOrRolesContain(Set principals, Map roles, String
principalOrRoleNameWant)
{
   if (roles.containsKey(principalOrRoleNameWant)) {
      return true;
   }
   {
      for (Iterator i = principals.iterator(); i.hasNext();) {
         Principal principal = (Principal)i.next();
         String principalNameHave = principal.getName();
         if (principalOrRoleNameWant.equals(principalNameHave)) {
            return true;
         }
      }
   }
   return false;
}

private Result isAccessAllowed(Resource resource, Set principals, Map roles)
{
   for (Enumeration e = database.getPolicy(resource); e.hasMoreElements();)
   {
    String principalOrRoleNameAllowed = (String)e.nextElement();
    if (WLSPrincipals.getEveryoneGroupname().
      equals(principalOrRoleNameAllowed) ||
```

```
            (WLSPrincipals.getUsersGroupname().equals(principalOrRoleNameAllowed)
            && !principals.isEmpty()) || principalsOrRolesContain(principals,
            roles, principalOrRoleNameAllowed))
            {
               return Result.PERMIT;
            }
         }
         return Result.DENY;
      }
   }
```

WebLogic MBeanMaker を使用して MBean タイプを生成する

カスタム セキュリティ プロバイダの MBean タイプを生成する前に、以下の作業が必要です。

- 2-11 ページの「MBean タイプが必要な理由を理解する」
- 2-12 ページの「拡張および実装する SSPI MBean を決定する」
- 2-13 ページの「MBean 定義ファイル (MDF) の基本的な要素を理解する」
- 2-15 ページの「SSPI MBean の階層と Administration Console に対する影響を理解する」
- 2-17 ページの「WebLogic MBeanMaker によって提供されるものを理解する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム認可プロバイダの MBean タイプを作成します。

1. 6-15 ページの「MBean 定義ファイル (MDF) を作成する」
2. 6-15 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 6-20 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」
4. 6-21 ページの「WebLogic Server 環境に MBean タイプをインストールする」

注意： これらの手順の実行方法は、いくつかのサンプル セキュリティ プロバイダ (dev2dev Web サイトの「Code Samples: WebLogic Server」で入手可能) に示されています。

この節で説明する手順はすべて、Windows 環境での作業を想定しています。

MBean 定義ファイル (MDF) を作成する

MBean 定義ファイル (MDF) を作成するには、次の手順に従います。

1. サンプル認可プロバイダの MDF をテキスト ファイルにコピーします。

   注意： サンプル認可プロバイダの MDF は、`SampleAuthorizer.xml` という名前です。

2. カスタム認可プロバイダに合わせて、作成する MDF の `<MBeanType>` および `<MBeanAttribute>` 要素の内容を変更します。

3. カスタム属性および操作 (つまり、`<MBeanAttribute>` および `<MBeanOperation>` 要素) を MDF に追加します。

4. ファイルを保存します。

注意： MDF 要素の構文についての完全なリファレンスは、付録 A「MBean 定義ファイル (MDF) 要素の構文」に収められています。

WebLogic MBeanMaker を使用して MBean タイプを生成する

MDF を作成したら、WebLogic MBeanMaker を使用してそれを実行できます。WebLogic MBeanMaker は現在のところコマンドライン ユーティリティで、入力として MDF を受け取り、MBean インタフェース、MBean 実装、関連する MBean 情報ファイルなどの中間 Java ファイルをいくつか出力します。これらの中間ファイルが合わさって、カスタム セキュリティ プロバイダの **MBean タイプ**になります。

MBean タイプの作成手順は、カスタム認可プロバイダの設計に応じて異なります。必要な設計に合わせて適切な手順を実行してください。

- 6-16 ページの「任意 SSPI MBean とカスタム操作を追加しない場合」
- 6-16 ページの「任意 SSPI MBean またはカスタム操作を追加する場合」

任意 SSPI MBean とカスタム操作を追加しない場合

カスタム認可プロバイダの MDF に任意 SSPI MBean もカスタム操作も実装しない場合、次の手順に従います。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

   `xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

   `-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

   注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり認可プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。
3. 6-20 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

任意 SSPI MBean またはカスタム操作を追加する場合

カスタム認可プロバイダの MDF に任意 SSPI MBean またはカスタム操作を実装する場合、以下の質問に答えながら手順を進めてください。

- MBean タイプを作成するのは初めてですか。初めての場合は、次の手順に従ってください。

1. 新しい DOS シェルを作成します。

2. 次のコマンドを入力します。

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

   `xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

   `-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

   注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり認可プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

3. 任意 SSPI MBean を MDF に実装した場合は、次の手順に従います。

   a. MBean 実装ファイルを見つけます。

      WebLogic MBeanMaker によって生成される MBean 実装ファイルには、*`MBeanName`*`Impl.java` という名前が付けられます。たとえば、`SampleAuthorizer` という MDF の場合には、編集すべき MBean 実装ファイルの名前は `SampleAuthorizerImpl.java` です。

   b. MDF に実装した任意 SSPI MBean ごとに、メソッド スタブを「Mapping MDF Operation Declarations to Java Method Signatures Document」(dev2dev Web サイトで入手可能) から MBean 実装ファイルにコピーし、各メソッドを実装します。任意 SSPI MBean が継承するメソッドもすべて実装してください。

4. MDF にカスタム操作を含めた場合、メソッド スタブを使用してメソッドを実装します。

5. ファイルを保存します。

6. 6-20 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

■ 既存の MBean タイプの更新ですか。更新の場合は、次の手順に従ってください。

1. WebLogic MBeanMaker によって現在のメソッドの実装が上書きされないように、既存の MBean 実装ファイルを一時ディレクトリにコピーします。
2. 新しい DOS シェルを作成します。
3. 次のコマンドを入力します。

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

   `xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

   `-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

   注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり認可プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

4. 任意 SSPI MBean を MDF に実装した場合は、次の手順に従います。

   a. MBean 実装ファイルを見つけます。

      WebLogic MBeanMaker によって生成される MBean 実装ファイルには、*`MBeanName`*`Impl.java` という名前が付けられます。たとえば、`SampleAuthorizer` という MDF の場合には、編集すべき MBean 実装ファイルの名前は `SampleAuthorizerImpl.java` です。

   b. 手順 1 で一時ディレクトリに保存した既存の MBean 実装ファイルを開きます。

   c. 既存の MBean 実装ファイルを、WebLogic MBeanMaker によって生成された MBean 実装ファイルと同期させます。

      これには、メソッドの実装を既存の MBean 実装ファイルから新しく生成された MBean 実装ファイルにコピー (または、新しく生成された MBean 実装ファイルから既存の MBean 実装ファイルに新しいメソッドを追加) し、いずれの MBean 実装ファイルにも入っているメソッドのメソッド シ

グネチャへの変更が使用する MBean 実装ファイルに反映されていることを確認するといった作業が必要です。

d. MDF を修正して元の MDF にはない任意 SSPI MBean を実装した場合は、メソッド スタブを「Mapping MDF Operation Declarations to Java Method Signatures Document」(dev2dev Web サイトで入手可能) から MBean 実装ファイルにコピーし、各メソッドを実装します。任意 SSPI MBean が継承するメソッドもすべて実装してください。

5. MDF を変更して元の MDF にはないカスタム操作を含めた場合、メソッド スタブを使用してメソッドを実装します。
6. 完成した、つまりすべてのメソッドを実装した MBean 実装ファイルを保存します。
7. この MBean 実装ファイルを、WebLogic MBeanMaker が MBean タイプの実装ファイルを配置したディレクトリにコピーします。このディレクトリは、手順 3 で *`filesdir`* として指定しました (手順 3 の結果として WebLogic MBeanMaker で生成された MBean 実装ファイルがオーバーライドされる)。
8. 6-20 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

生成される MBean インタフェース ファイルについて

MBean インタフェース ファイルとは、実行時クラスまたは MBean 実装がコンフィグレーション データを取得するために使用する MBean のクライアントサイド API です。2-4 ページの「「Provider」SSPI の目的を理解する」で説明されているように、これは initialize メソッドで使用するのが一般的です。

WebLogic MBeanMaker では、作成済みの MDF から MBean タイプを生成するので、生成される MBean インタフェース ファイルの名前は、その MDF 名の後に「MBean」というテキストが付いたものになります。たとえば、WebLogic MBeanMaker で `SampleAuthorizer` MDF を実行すると、その結果、`SampleAuthorizerMBean.java` という MBean インタフェース ファイルが生成されます。

WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する

WebLogic MBeanMaker を使用して MDF を実行して中間ファイルを生成し、MBean 実装ファイルを手作業で編集してその中にメソッドの内容を記述したら、カスタム認可プロバイダの MBean ファイルと実行時クラスを MBean JAR ファイル (MJF) にパッケージ化する必要があります。このプロセスも、WebLogic MBeanMaker によって自動化されます。

カスタム認可プロバイダの MJF を作成するには、次の手順に従います。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

   ```
   java -DMJF=jarfile -Dfiles=filesdir
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMJF` フラグは WebLogic MBeanMaker が新しい MBean タイプを格納する JAR ファイルをビルドする必要があることを示し、*`jarfile`* は MJF の名前、*`filesdir`* は WebLogic MBeanMaker で MJF に JAR 化する対象ファイルが存在する場所です。

   この時点でコンパイルが行われるので、エラーが発生するおそれがあります。*`jarfile`* が指定されていて、エラーが発生しなかった場合には、指定された名前の MJF が作成されます。

注意： 既存の MJF を更新する場合は、MJF をいったん削除してから再生成します。WebLogic MBeanMaker にも `-DIncludeSource` オプションがあり、それを指定すると、生成される MJF にソース ファイルを含めるかどうかを制御することができます。ソース ファイルには、生成されたソースと MDF そのものがあります。デフォルトは `false` です。このオプションは、`-DMJF` を使用しない場合には無視されます。

生成された MJF は、自らの WebLogic Server 環境にインストールすることも、顧客に配布してそれぞれの WebLogic Server 環境にインストールしてもらうこともできます。

WebLogic Server 環境に MBean タイプをインストールする

MBean タイプを WebLogic Server 環境にインストールするには、MJF を *WL_HOME*\server\lib\mbeantypes ディレクトリにコピーします。ここで、*WL_HOME* は WebLogic Server の最上位のインストール ディレクトリです。このインストール コマンドによって、カスタム認可プロバイダが「デプロイ」されます。つまり、カスタム認可プロバイダを WebLogic Server Administration Console から管理できるようになります。

注意： *WL_HOME*\server\lib\mbeantypes は、MBean タイプをインストールするデフォルトのディレクトリです。ただし、WebLogic Server が追加ディレクトリで MBean タイプを検索するようにするには、サーバを起動するときに -Dweblogic.alternateTypesDirectory=<*dir*> コマンドライン フラグを使用します (<*dir*> はディレクトリ名のカンマ区切りのリスト)。このフラグを使用すると、WebLogic Server は常にまず *WL_HOME*\server\lib\mbeantypes から MBean タイプをロードし、その後に追加ディレクトリを検索して、(拡張子に関係なく) そのディレクトリに存在するすべての有効なアーカイブをロードします。たとえば -Dweblogic.alternateTypesDirectory = dirX,dirY の場合、WebLogic Server はまず *WL_HOME*\server\lib\mbeantypes から MBean タイプをロードし、その後に、dirX および dirY に存在する有効なアーカイブをロードします。

追加ディレクトリで MBean タイプを検索するように WebLogic Server に指示し、かつ Java セキュリティ マネージャを使用する場合は、weblogic.policy ファイルを更新して、MBean タイプ (したがってカスタム セキュリティ プロバイダも) に対する適切なパーミッションを与える必要があります。詳細については、『WebLogic Security プログラマーズ ガイド』の「Java セキュリティ マネージャを使用しての WebLogic リソースの保護」を参照してください。

非セキュリティ プロバイダの JAR (バックアップ ファイルを含む) は、*WL_HOME*\server\lib\mbeantypes ディレクトリ以外の場所に置くことをお勧めします。

カスタム認可プロバイダをコンフィグレーションすることによって (6-22 ページの「Administration Console を使用してカスタム認可プロバイダをコンフィグレーションする」を参照)、MBean タイプのインスタンスを作成して、GUI、他の Java コード、または API からそれらの MBean インスタンスを使用することが

できます。たとえば、WebLogic Server Administration Console を使用して、属性を取得 / 設定したり操作を呼び出したりすることもできますし、他の Java オブジェクトを開発して、そのオブジェクトで MBean をインスタンス化し、それらの MBean から提供される情報に自動的に応答させることもできます。なお、これらの MBean インスタンスをバックアップしておくことをお勧めします。詳細については、『WebLogic Server ドメイン管理』の「障害が発生したサーバの回復」の「セキュリティ コンフィグレーション データのバックアップ」を参照してください。

Administration Console を使用してカスタム認可プロバイダをコンフィグレーションする

カスタム認可プロバイダをコンフィグレーションするということは、認可サービスを必要とするアプリケーションがアクセス可能なセキュリティ レルムにカスタム認可プロバイダを追加するということです。

カスタム セキュリティ プロバイダのコンフィグレーションは管理タスクですが、カスタム セキュリティ プロバイダの開発者が行うこともできます。この節では、カスタム認可プロバイダのコンフィグレーション担当者向けの重要な情報を取り上げます。

- 6-22 ページの「認可プロバイダとデプロイメント記述子の管理」
- 6-25 ページの「ポリシー デプロイメントの有効化」

注意： WebLogic Server Administration Console を使用したカスタム認可プロバイダのコンフィグレーション手順については、『WebLogic Security の管理』の「カスタム セキュリティ プロバイダのコンフィグレーション」を参照してください。

認可プロバイダとデプロイメント記述子の管理

エンタープライズ JavaBean (EJB) や Web アプリケーションなどのアプリケーション コンポーネントの中には、関連デプロイメント情報を Java 2 Enterprise Edition (J2EE) デプロイメント記述子および WebLogic Server デプロイメント記述子に格納するものがあります。Web アプリケーションの場合、デプロイメン

ト記述子ファイル (web.xml と weblogic.xml) には、セキュリティ ポリシーの宣言を含む J2EE セキュリティ モデルの実装情報が格納されます。この情報は、WebLogic Server Administration Console で認可プロバイダを初めてコンフィグレーションするときに格納するのが一般的です。

Administration Console には、この目的のために [デプロイメント記述子内のセキュリティ データを無視] チェックボックスが用意されています。開発者または管理者がカスタム認可プロバイダを初めてコンフィグレーションするときには、このチェックボックスのチェックがはずれていることを確認する必要があります。

注意： [デプロイメント記述子内のセキュリティ データを無視] チェックボックスは、デフォルトではチェックがはずれています。このチェックボックスを設定するには、WebLogic Server Administration Console の左ペインで [セキュリティ | レルム | *realm*] をクリックします。*realm* はセキュリティ レルムの名前です。次に [一般] タブを選択します。

このチェックボックスのチェックをはずして Web アプリケーションをデプロイすると、WebLogic Server は、web.xml および weblogic.xml デプロイメント記述子ファイル (コード リスト 6-3 およびコード リスト 6-4 の web.xml および weblogic.xml の例を参照) からセキュリティ ポリシー情報を読み込みます。この情報は、認可プロバイダのセキュリティ プロバイダ データベースにコピーされます。

コード リスト 6-3 web.xml ファイルのサンプル

```
<web-app>

   <welcome-file-list>
      <welcome-file>welcome.jsp</welcome-file>
   </welcome-file-list>

   <security-constraint>
      <web-resource-collection>
         <web-resource-name>Success</web-resource-name>
         <url-pattern>/welcome.jsp</url-pattern>
         <http-method>GET</http-method>
         <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
         <role-name>developers</role-name>
      </auth-constraint>
   </security-constraint>
```

```
   <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>default</realm-name>
   </login-config>

   <security-role>
      <role-name>developers</role-name>
   </security-role>

</web-app>
```

コード リスト 6-4　Sample weblogic.xml File

```
<weblogic-web-app>
   <security-role-assignment>
      <role-name>developers</role-name>
      <principal-name>myGroup</principal-name>
   </security-role-assignment>
</weblogic-web-app>
```

注意： web.xml/weblogic.xml デプロイメント記述子ファイルでも WebLogic Server Administration Console でも追加のセキュリティ ポリシーを設定することができますが、Web アプリケーションのデプロイメント記述子に定義されているセキュリティ ポリシーをいったんコピーしてから Administration Console を使用して追加のセキュリティ ポリシーを定義することをお勧めします。この理由は、認可プロバイダのコンフィグレーション中に Administration Console を使用してセキュリティ ポリシーを変更すると、その内容が web.xml および weblogic.xml ファイルに保持されないからです。Administration Console を使用して再デプロイする、ディスク上で Web アプリケーションを変更した、または WebLogic Server を再起動したといった場合に、Web アプリケーションを再デプロイするときには、[デプロイメント記述子内のセキュリティ データを無視] チェックボックスをチェックする必要があります。チェックボックスをチェックしないと、Administration Console で定義したセキュリティ ポリシーがデプロイメント記述子に定義されているセキュリティ ポリシーによって上書きされます。詳細については、『WebLogic リソースのセキュリティ』の「組み合わせた方法による URL (Web) リソースとエンタープライズ JavaBean (EJB) リソースの保護」を参照してください。

EJB についてもプロセスは同じですが、`ejb-jar.xml/weblogic-ejb-jar.xml` デプロイメント記述子を使用します。

[デプロイメント記述子内のセキュリティ データを無視] チェックボックスは、ロール マッピング プロバイダおよび資格マッピング プロバイダにも影響します。詳細については、それぞれ 8-26 ページの「ロール マッピング プロバイダとデプロイメント記述子の管理」と 11-17 ページの「資格マッピング プロバイダ、リソース アダプタ、およびデプロイメント記述子の管理」を参照してください。

ポリシー デプロイメントの有効化

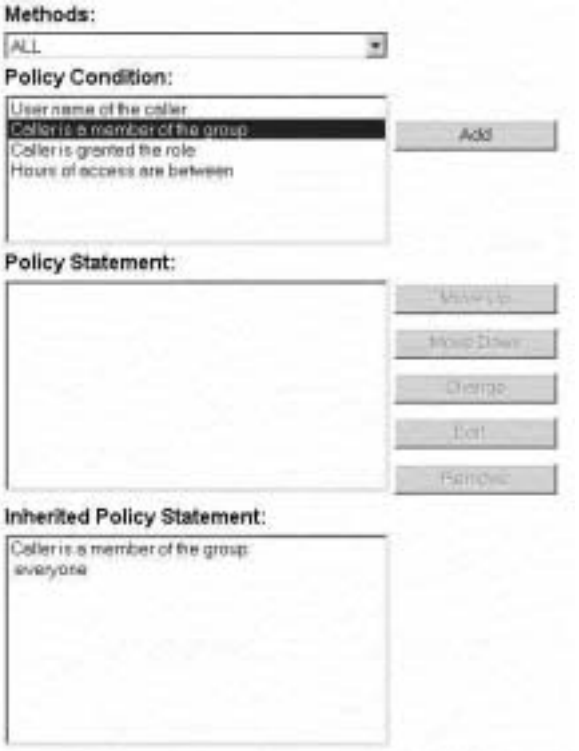
`DeployableAuthorizationProvider` SSPI を実装し、カスタム認可プロバイダでデプロイ可能なセキュリティ ポリシーをサポートしたい場合、カスタム認可プロバイダのコンフィグレーション担当者 (つまり、開発者または管理者) は、WebLogic Server Administration Console で [ポリシー デプロイメントを有効化] チェックボックスがチェックされていることを確認する必要があります。チェックがはずれていると、認可プロバイダは「オフ」と見なされます。このため、複数の認可プロバイダがコンフィグレーションされている場合、[ポリシー デプロイメントを有効化] チェックボックスを使用すると、セキュリティ ポリシーのデプロイメントに使用する認可プロバイダを指定できます。

注意： [デプロイメント記述子内のセキュリティ データを無視] チェックボックス (6-22 ページの「認可プロバイダとデプロイメント記述子の管理」で説明したようにセキュリティ レルム レベルで指定) では、コンフィグレーション済みの認可プロバイダのセキュリティ データベースにセキュリティ ポリシーをコピーするかどうかを指定します。[ポリシー デプロイメントを有効化] チェックボックス (コンフィグレーション済みの認可プロバイダごとに指定) では、認可プロバイダがデプロイ済みのセキュリティ ポリシーを格納するかどうかを指定します。

セキュリティ ポリシーを管理するためのメカニズムを提供する

WebLogic Server Administration Console を使用してカスタム認可プロバイダをコンフィグレーションすると、必要な認可サービスにアプリケーションからアクセスできるようにすることはできますが、このセキュリティ プロバイダに関連付けられたセキュリティ ポリシーを管理する方法を管理者にも提供する必要があります。たとえば WebLogic 認可プロバイダには、ポリシー エディタ ページ (図 6-2 を参照) が管理者向けに用意されており、リソースを右クリックして [ポリシーを定義 ...] オプションを選択すると、さまざまな WebLogic リソースのセキュリティ ポリシーを追加、変更、または削除することができます。

図 6-2　WebLogic 認可プロバイダのポリシー エディタ ページ



管理者は、カスタム認可プロバイダを開発するときに、ポリシー エディタ ページも右クリック メニューも利用できません。したがって、セキュリティ ポリシーを管理するための独自のメカニズムを提供する必要があります。このメカニズムでは、カスタム認可プロバイダのデータベースのセキュリティ ポリシー データ (つまり式) を読み書きできなければなりません。

それには、以下の 3 通りの方法があります。

- 6-27 ページの「オプション 1 : コンソール拡張を使用して独自の「ポリシー エディタ」ページを作成する」
- 6-28 ページの「オプション 2 : セキュリティ ポリシー管理用のスタンドアロン ツールを開発する」
- 6-29 ページの「オプション 3 : 既存のセキュリティ ポリシー管理ツールを Administration Console に統合する」

オプション 1 : コンソール拡張を使用して独自の「ポリシー エディタ」ページを作成する

カスタム認可プロバイダ用にコンソール拡張を作成する方法の主な利点は、コンソール拡張によって WebLogic リソースの ID がわかるので、その WebLogic リソースがリソース階層のどこにあるかもわかるということです。この情報は、認可プロバイダのデータベースから式を読み書きするために必要です。また、WebLogic 認可プロバイダのポリシー エディタ ページのように、作成したページを既存の WebLogic Server Administration Console GUI に統合できるという利点もあります。

この方法を選択した場合、次の作業が必要になります。

1. `weblogic.management.console.extensibility.SecurityExtension` インタフェースの `getExtensionForPolicy()` メソッドを実装し、作成したポリシー エディタ ページをこのメソッドで返します。

   注意 : 詳細については、第 12 章「カスタム セキュリティ プロバイダ用のコンソール拡張の記述」を参照してください。

2. また、以下のいずれかを実行する必要があります。

a. `PolicyEditor` および `PolicyReader` 認可用任意 SSPI MBean を実装し、作成したポリシー エディタ ページと認可プロバイダのデータベースとの間の仲介役となる管理 MBean を開発します。詳細については、2-12 ページの「拡張および実装する SSPI MBean を決定する」と 2-22 ページの表 2-4「認可用任意 SSPI MBean」を参照してください。

   この場合、文字列として表現可能なセキュリティ ポリシーを構成する式 (`Role=Admin` や `Group=Administrators` など) の構文も開発する必要があります。

   注意 : この構文は、認可プロバイダごとに異なっていてもかまいません。式の詳細については、『WebLogic リソースのセキュリティ』の「セキュリティ ポリシーの構成要素 : ポリシー条件、式、およびポリシー文」を参照してください。

b. セキュリティ ポリシーを管理するための MBean API を開発し、それらのインタフェースを実装し、作成したポリシー エディタ ページと認可プロバイダのデータベースとの間の仲介役となる管理 MBean を開発します。

c. MBean に委託しないで、作成したページから直接、カスタム認可プロバイダのデータベースの式を読み書きします。

オプション 2 : セキュリティ ポリシー管理用のスタンドアロン ツールを開発する

WebLogic Server Administration Console とまったく別のツールを開発する場合には、この方法を選択します。

この方法の場合、6-27 ページの「オプション 1 : コンソール拡張を使用して独自の「ポリシー エディタ」ページを作成する」で説明したカスタム認可プロバイダ用のコンソール拡張を記述する必要もなく、管理 MBean を開発する必要もありません。ただし、ツールでは以下のことを行う必要があります。

1. WebLogic リソースの ID を特定します。ID はコンソール拡張によって自動的に提供されません。詳細については、2-26 ページの「WebLogic リソース識別子」を参照してください。

2. セキュリティ ポリシーを構成する式を表す方法を決定します。この表現は完全に任意で、6-27 ページの「オプション 1 : コンソール拡張を使用して独自の「ポリシー エディタ」ページを作成する」と違って文字列である必要はありません。
3. カスタム認可プロバイダのデータベースの式を読み書きします。

オプション 3 : 既存のセキュリティ ポリシー管理ツールを Administration Console に統合する

WebLogic Server Administration Console とは別のツールを持っており、それを Administration Console から起動する場合には、この方法を選択します。

この方法の場合、ツールでは以下のことを行う必要があります。

1. WebLogic リソースの ID を特定します。ID はコンソール拡張によって自動的に提供されません。詳細については、2-26 ページの「WebLogic リソース識別子」を参照してください。
2. セキュリティ ポリシーを構成する式を表す方法を決定します。この表現は完全に任意で、6-27 ページの「オプション 1 : コンソール拡張を使用して独自の「ポリシー エディタ」ページを作成する」と違って文字列である必要はありません。
3. カスタム認可プロバイダのデータベースの式を読み書きします。
4. 『Administration Console の拡張』に説明されているように、基本コンソール拡張を使用して Administration Console にリンクします。

7　裁決プロバイダ

裁決では、複数の認可プロバイダがコンフィグレーションされている場合に発生するおそれのある認可上の衝突を、それぞれの認可プロバイダのアクセス決定の結果を比較検討することによって解消します。WebLogic Server では、裁決プロバイダを使用して、複数のアクセス決定から返される結果を調停し、`PERMIT` か `DENY` の最終判定を下します。また、裁決プロバイダでは、単一の認可プロバイダのアクセス決定から `ABSTAIN` の回答が返されたときにどうすべきかを指定することもできます。

以下の節では、裁決プロバイダの概念と機能、およびカスタム裁決プロバイダの開発手順について説明します。

裁決プロセス

裁決プロバイダは認可プロセスの一部として使用されるので、使用方法は 6-2 ページの「認可プロセス」で説明されています。

カスタム裁決プロバイダを開発する必要があるか

WebLogic Server のデフォルト (つまりアクティブな) セキュリティ レルムには WebLogic 裁決プロバイダが含まれています。WebLogic 裁決プロバイダは、複数の認可プロバイダのアクセス決定から異なる結果が返された場合に裁決を行い、WebLogic リソースへのアクセスを許可するかどうかを最終的に判定します。

WebLogic 裁決プロバイダには、動作を制御する [完全一致の許可が必要] という属性があります。[完全一致の許可が必要] 属性は、デフォルトでは TRUE に設定されており、WebLogic 裁決プロバイダはその設定に従って次のように動作します。

- すべての認可プロバイダのアクセス決定が PERMIT を返した場合、最終的な判定として TRUE を返します (つまり WebLogic リソースへのアクセスは許可されます)。
- 一部の認可プロバイダのアクセス決定が PERMIT を返し、その他が ABSTAIN を返した場合、最終的な判定として FALSE を返します (つまり WebLogic リソースへのアクセスは拒否されます)。
- いずれかの認可プロバイダのアクセス決定が ABSTAIN または DENY を返した場合、最終的な判定として FALSE を返します (つまり WebLogic リソースへのアクセスは拒否されます)。

[完全一致の許可が必要] 属性を FALSE に変更した場合、WebLogic 裁決プロバイダは次のように動作します。

- すべての認可プロバイダのアクセス決定が PERMIT を返した場合、最終的な判定として TRUE を返します (つまり WebLogic リソースへのアクセスは許可されます)。
- 一部の認可プロバイダのアクセス決定が PERMIT を返し、その他が ABSTAIN を返した場合、最終的な判定として TRUE を返します (つまり WebLogic リソースへのアクセスは許可されます)。

- いずれかの認可プロバイダのアクセス決定が DENY を返した場合、最終的な判定として FALSE を返します (つまり WebLogic リソースへのアクセスは拒否されます)。

注意： [完全一致の許可が必要] 属性は、WebLogic 裁決プロバイダをコンフィグレーションするときに設定します。WebLogic 裁決プロバイダのコンフィグレーションの詳細については、『WebLogic Security の管理』の「WebLogic 裁決プロバイダのコンフィグレーション」を参照してください。

上記の説明と異なる動作の裁決プロバイダが必要な場合、カスタム裁決プロバイダを開発する必要があります (裁決プロバイダでは、1 つの認可プロバイダのアクセス決定が ABSTAIN を返した場合に、指定したセキュリティ要件に基づいてどうすべきかも指定できることに注意してください)。

カスタム裁決プロバイダの開発方法

WebLogic 裁決プロバイダが開発者の要求を満たさない場合、次の手順でカスタム裁決プロバイダを開発することができます。

1. 7-3 ページの「適切な SSPI を使用して実行時クラスを作成する」
2. 7-5 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 7-12 ページの「Administration Console を使用してカスタム裁決プロバイダをコンフィグレーションする」

適切な SSPI を使用して実行時クラスを作成する

実行時クラスを作成する前に、以下の作業が必要です。

- 2-4 ページの「「Provider」SSPI の目的を理解する」
- 2-7 ページの「SSPI 階層を理解し、実行時クラスを 1 つ作成するのか 2 つ作成するのかを決定する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム裁決プロバイダの実行時クラスを作成します。

- 7-4 ページの「AdjudicationProvider SSPI を実装する」
- 7-4 ページの「Adjudicator SSPI を実装する」

AdjudicationProvider SSPI を実装する

AdjudicationProvider SSPI を実装するには、2-4 ページの「「Provider」SSPI の目的を理解する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

getAdjudicator

```
public Adjudicator getAdjudicator()
```

getAdjudicator メソッドは、Adjudicator SSPI の実装を取得します。MyAdjudicationProviderImpl.java という 1 つの実行時クラスの場合、getAdjudicator メソッドの実装は次のようになります。

```
return this;
```

実行時クラスが 2 つの場合、getAdjudicator メソッドの実装は次のようになります。

```
return new MyAdjudicatorImpl;
```

この理由は、AdjudicationProvider SSPI を実装する実行時クラスが、Adjudicator SSPI を実装するクラスを取得するためのファクトリとして使用されるからです。

AdjudicationProvider SSPI および getAdjudicator メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

Adjudicator SSPI を実装する

Adjudicator SSPI を実装するには、以下のメソッドの実装を提供する必要があります。

initialize

```
public void initialize(String[] accessDecisionClassNames)
```

`initialize` メソッドは、「アクセスは許されるか」という質問への回答を得るために呼び出されるすべてのコンフィグレーション済み認可プロバイダのアクセス決定の名前を初期化します。`accessDecisionClassNames` パラメータは、裁決プロバイダの `adjudicate` メソッドで特定のアクセス決定による判定結果を支持するために使用することもできます。認可プロバイダとアクセス決定の詳細については、第 6 章「認可プロバイダ」を参照してください。

adjudicate

```
public boolean adjudicate( Result[] results )
```

`adjudicate` メソッドは、コンフィグレーション済みのアクセス決定から返された判定結果をすべて受け取り、「アクセスは許されるか」という質問への回答を決定します。

`Adjudicator` SSPI と `initialize` および `adjudicate` メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

WebLogic MBeanMaker を使用して MBean タイプを生成する

カスタム セキュリティ プロバイダの MBean タイプを生成する前に、以下の作業が必要です。

- 2-11 ページの「MBean タイプが必要な理由を理解する」
- 2-12 ページの「拡張および実装する SSPI MBean を決定する」
- 2-13 ページの「MBean 定義ファイル (MDF) の基本的な要素を理解する」
- 2-15 ページの「SSPI MBean の階層と Administration Console に対する影響を理解する」
- 2-17 ページの「WebLogic MBeanMaker によって提供されるものを理解する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム裁決プロバイダの MBean タイプを作成します。

1. 7-6 ページの「MBean 定義ファイル (MDF) を作成する」

2. 7-7 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 7-10 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」
4. WebLogic Server 環境に MBean タイプをインストールする

注意： これらの手順の実行方法は、いくつかのサンプル セキュリティ プロバイダ (dev2dev Web サイトの「Code Samples: WebLogic Server」で入手可能) に示されています。

この節で説明する手順はすべて、Windows 環境での作業を想定しています。

MBean 定義ファイル (MDF) を作成する

MBean 定義ファイル (MDF) を作成するには、次の手順に従います。

1. サンプル認証プロバイダの MDF をテキスト ファイルにコピーします。

   注意： サンプル認証プロバイダの MDF は、`SampleAuthenticator.xml` という名前です (現在、サンプル裁決プロバイダはありません)。

2. カスタム裁決プロバイダに合わせて、作成する MDF の `<MBeanType>` および `<MBeanAttribute>` 要素の内容を変更します。
3. カスタム属性および操作 (つまり、`<MBeanAttribute>` および `<MBeanOperation>` 要素) を MDF に追加します。
4. ファイルを保存します。

注意： MDF 要素の構文についての完全なリファレンスは、付録 A「MBean 定義ファイル (MDF) 要素の構文」に収められています。

WebLogic MBeanMaker を使用して MBean タイプを生成する

MDF を作成したら、WebLogic MBeanMaker を使用してそれを実行できます。WebLogic MBeanMaker は現在のところコマンドライン ユーティリティで、入力として MDF を受け取り、MBean インタフェース、MBean 実装、関連する MBean 情報ファイルなどの中間 Java ファイルをいくつか出力します。これらの中間ファイルが合わさって、カスタム セキュリティ プロバイダの **MBean タイプ**になります。

MBean タイプの生成手順は、カスタム裁決プロバイダの設計に応じて異なります。必要な設計に合わせて適切な手順を実行してください。

- 7-7 ページの「カスタム操作を追加しない場合」
- 7-8 ページの「カスタム操作を追加する場合」

カスタム操作を追加しない場合

カスタム裁決プロバイダの MDF にカスタム操作を含めない場合、次の手順に従います。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

   `xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

   `-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり裁決プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

3. 7-10 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

カスタム操作を追加する場合

カスタム裁決プロバイダの MDF にカスタム操作を含める場合、質問に答えながら手順を進めてください。

- MBean タイプを作成するのは初めてですか。初めての場合は、次の手順に従ってください。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

```
java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
weblogic.management.commo.WebLogicMBeanMaker
```

ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

`xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

`-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり裁決プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

3. MDF のすべてのカスタム操作に対して、メソッド スタブを使用してメソッドを実装します。
4. ファイルを保存します。
5. 7-10 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

- 既存の MBean タイプの更新ですか。更新の場合は、次の手順に従ってください。

1. WebLogic MBeanMaker によって現在のメソッドの実装が上書きされないように、既存の MBean 実装ファイルを一時ディレクトリにコピーします。
2. 新しい DOS シェルを作成します。
3. 次のコマンドを入力します。

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

   `xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

   `-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

   注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり裁決プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

4. MDF を変更して元の MDF にはないカスタム操作を含めた場合、メソッドスタブを使用してメソッドを実装します。
5. 完成した、つまりすべてのメソッドを実装した MBean 実装ファイルを保存します。
6. この MBean 実装ファイルを、WebLogic MBeanMaker が MBean タイプの実装ファイルを配置したディレクトリにコピーします。このディレクトリは、手順 3 で *`filesdir`* として指定しました (手順 3 の結果として WebLogic MBeanMaker で生成された MBean 実装ファイルがオーバーライドされる)。
7. 7-10 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

生成される MBean インタフェース ファイルについて

MBean インタフェース ファイルとは、実行時クラスまたは MBean 実装がコンフィグレーション データを取得するために使用する MBean のクライアントサイド API です。2-4 ページの「「Provider」 SSPI の目的を理解する」 で説明されているように、これは initialize メソッドで使用するのが一般的です。

WebLogic MBeanMaker では、作成済みの MDF から MBean タイプを生成するので、生成される MBean インタフェース ファイルの名前は、その MDF 名の後に「MBean」というテキストが付いたものになります。たとえば、WebLogic MBeanMaker で `MyAdjudicator` MDF を実行すると、`MyAdjudicatorMBean.java` という MBean インタフェース ファイルが生成されます。

WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する

WebLogic MBeanMaker で MDF を実行して中間ファイルを作成し、MBean 実装ファイルを手作業で編集してその中にメソッドの内容を記述したら、カスタム裁決プロバイダの MBean ファイルと実行時クラスを MBean JAR ファイル (MJF) にパッケージ化する必要があります。このプロセスも、WebLogic MBeanMaker によって自動化されます。

カスタム裁決プロバイダの MJF を作成するには、次の手順に従います。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

   ```
   java -DMJF=jarfile -Dfiles=filesdir
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMJF` フラグは WebLogic MBeanMaker が新しい MBean タイプを格納する JAR ファイルをビルドする必要があることを示し、*`jarfile`* は MJF の名前、*`filesdir`* は WebLogic MBeanMaker で MJF に JAR 化する対象ファイルが存在する場所です。

   この時点でコンパイルが行われるので、エラーが発生するおそれがあります。*`jarfile`* が指定されていて、エラーが発生しなかった場合には、指定された名前の MJF が作成されます。

注意： 既存の MJF を更新する場合は、MJF をいったん削除してから再生成します。WebLogic MBeanMaker にも `-DIncludeSource` オプションがあり、それを指定すると、生成される MJF にソース ファイルを含めるかどうかを制御することができます。ソース ファイルには、生成されたソースと MDF そのものがあります。デフォルトは `false` です。このオプションは、`-DMJF` を使用しない場合には無視されます。

生成された MJF は、自らの WebLogic Server 環境にインストールすることも、顧客に配布してそれぞれの WebLogic Server 環境にインストールしてもらうこともできます。

WebLogic Server 環境に MBean タイプをインストールする

MBean タイプを WebLogic Server 環境にインストールするには、MJF を *`WL_HOME`*`\server\lib\mbeantypes` ディレクトリにコピーします。ここで、*`WL_HOME`* は WebLogic Server の最上位のインストール ディレクトリです。このインストール コマンドによって、カスタム裁決プロバイダが「デプロイ」されます。つまり、カスタム裁決プロバイダを WebLogic Server Administration Console から管理できるようになります。

注意： *`WL_HOME`*`\server\lib\mbeantypes` は、MBean タイプをインストールするデフォルトのディレクトリです。ただし、WebLogic Server が追加ディレクトリで MBean タイプを検索するようにするには、サーバを起動するときに `-Dweblogic.alternateTypesDirectory=<`*`dir`*`>` コマンドライン フラグを使用します (*`<dir>`* はディレクトリ名のカンマ区切りのリスト)。このフラグを使用すると、WebLogic Server は常にまず *`WL_HOME`*`\server\lib\mbeantypes` から MBean タイプをロードし、その後に追加ディレクトリを検索して、(拡張子に関係なく) そのディレクトリに存在するすべての有効なアーカイブをロードします。たとえば `-Dweblogic.alternateTypesDirectory = dirX,dirY` の場合、WebLogic Server はまず *`WL_HOME`*`\server\lib\mbeantypes` から MBean タイプをロードし、その後に、`dirX` および `dirY` に存在する有効なアーカイブをロードします。

追加ディレクトリで MBean タイプを検索するように WebLogic Server に指示し、かつ Java セキュリティ マネージャを使用する場合は、`weblogic.policy` ファイルを更新して、MBean タイプ (したがってカスタム セキュリティ プロバイダも) に対する適切なパーミッションを与え

る必要があります。詳細については、『WebLogic Security プログラマーズ ガイド』の「Java セキュリティ マネージャを使用しての WebLogic リソースの保護」を参照してください。

非セキュリティ プロバイダの JAR (バックアップ ファイルを含む) は、*WL_HOME*\server\lib\mbeantypes ディレクトリ以外の場所に置くことをお勧めします。

カスタム裁決プロバイダをコンフィグレーションすることによって (7-12 ページの「Administration Console を使用してカスタム裁決プロバイダをコンフィグレーションする」を参照) MBean タイプのインスタンスを作成して、GUI、他の Java コード、または API からそれらの MBean インスタンスを使用することができます。たとえば、WebLogic Server Administration Console を使用して、属性を取得 / 設定したり操作を呼び出したりすることもできますし、他の Java オブジェクトを開発して、そのオブジェクトで MBean をインスタンス化し、それらの MBean から提供される情報に自動的に応答させることもできます。なお、これらの MBean インスタンスをバックアップしておくことをお勧めします。詳細については、『WebLogic Server ドメイン管理』の「障害が発生したサーバの回復」の「セキュリティ コンフィグレーション データのバックアップ」を参照してください。

Administration Console を使用してカスタム裁決プロバイダをコンフィグレーションする

カスタム裁決プロバイダをコンフィグレーションするということは、裁決サービスを必要とするアプリケーションがアクセス可能なセキュリティ レルムにカスタム裁決プロバイダを追加するということです。

カスタム セキュリティ プロバイダのコンフィグレーションは管理タスクですが、カスタム セキュリティ プロバイダの開発者が行うこともできます。WebLogic Server Administration Console を使用したカスタム裁決プロバイダのコンフィグレーション手順については、『WebLogic Security の管理』の「カスタム セキュリティプロバイダのコンフィグレーション」を参照してください。

8　ロール マッピング プロバイダ

ロール マッピングとは、実行時にプリンシパル (ユーザまたはグループ) をセキュリティ ロールに動的に割り当てるプロセスのことです。WebLogic Server では、ロール マッピング プロバイダは、WebLogic リソースを操作しようとしているサブジェクト内のプリンシパルに適用されるセキュリティ ロールを調べます。通常、この操作では WebLogic リソースへのアクセスを取得する必要があるので、ロール マッピング プロバイダは認可プロバイダと共に使用するのが一般的です。

以下の節では、ロール マッピング プロバイダの概念と機能、およびカスタム ロール マッピング プロバイダの開発手順について説明します。

- 8-1 ページの「ロール マッピングの概念」
- 8-4 ページの「ロール マッピング プロセス」
- 8-7 ページの「カスタム ロール マッピング プロバイダを開発する必要があるか」
- 8-7 ページの「カスタム ロール マッピング プロバイダの開発方法」

ロール マッピングの概念

ロール マッピング プロバイダを開発する前に、以下の概念を理解しておく必要があります。

- 8-2 ページの「セキュリティ ロール」
- 8-3 ページの「動的セキュリティ ロール計算」
- 2-24 ページの「セキュリティ プロバイダと WebLogic リソース」

セキュリティ ロール

セキュリティ ロールは、WebLogic リソースへの同じアクセス パーミッションを持つユーザまたはグループの集合です。グループと同様、セキュリティ ロールを使用すると、複数のユーザによる WebLogic リソースへのアクセスを一度に制御できます。しかし、セキュリティ ロールは WebLogic Server ドメインの特定のリソースを対象としており (WebLogic Server ドメイン全体を対象とするグループとは異なる)、動的に定義することが可能です。8-3 ページの「動的セキュリティ ロール計算」を参照してください。

注意： セキュリティ ロールの詳細については、『WebLogic リソースのセキュリティ』の「セキュリティ ロール」を参照してください。WebLogic リソースの詳細については、2-24 ページの「セキュリティ プロバイダと WebLogic リソース」および『WebLogic リソースのセキュリティ』の「WebLogic リソース」を参照してください。

`weblogic.security.service` パッケージに定義されている `SecurityRole` インタフェースは、セキュリティ ロールの抽象的な記法を表すのに用いられます。詳細については、WebLogic Server 7.0 API リファレンス Javadoc の SecurityRole インタフェースを参照してください。

プリンシパルをセキュリティ ロールにマップすると、そのプリンシパルがそのセキュリティ ロールに「割り当てられている」かぎり、そのプリンシパルには定義されたアクセス パーミッションが付与されます。たとえば、アプリケーションで「`AppAdmin`」というセキュリティ ロールを定義し、これによってそのアプリケーションのリソースのごく一部に対する書き込みアクセスが提供されるものとします。この場合、`AppAdmin` セキュリティ ロールに割り当てられたプリンシパルはすべて、それらのリソースに対して書き込みアクセスを持つことになります。詳細については、8-3 ページの「動的セキュリティ ロール計算」と『WebLogic リソースのセキュリティ』の「セキュリティ ロール」を参照してください。

なお、多数のプリンシパルを単一のセキュリティ ロールにマップすることができます。プリンシパルの詳細については、3-2 ページの「ユーザ / グループ、プリンシパル、サブジェクト」を参照してください。

セキュリティ ロールの指定は、J2EE (Java 2 Enterprise Edition) デプロイメント記述子ファイルか WebLogic Server Administration Console、あるいはその両方で行われます。詳細については、8-26 ページの「ロール マッピング プロバイダとデプロイメント記述子の管理」を参照してください。

動的セキュリティ ロール計算

セキュリティ ロールは、宣言的なもの (すなわち、Java 2 Enterprise Edition におけるロール) とすることも、リクエストのコンテキストに基づいて動的に計算することもできます。

動的ロール計算とは、このようにプリンシパル (ユーザまたはグループ) を実行時にセキュリティ ロールにレイト バインドすることを意味する用語です。こうしたレイト バインディングは、プリンシパル対セキュリティ ロールの関連付けが静的に定義されるか動的に計算されるかによらず、保護対象 WebLogic リソースについての認可判定の直前に行われます。バインディングが呼び出しシーケンス内で行われるため、あらゆるプリンシパル対セキュリティ ロール計算の結果は、リクエストに対して下される認可判定の一環として、認証用 ID 情報として解釈することができます。

セキュリティ ロールの動的計算には、ビジネス ルールに基づいてユーザまたはグループにセキュリティ ロールを付与できるという、非常に重要な利点があります。たとえば、本来の管理者が長期の出張で不在の間だけユーザに `Manager` セキュリティ ロールを割り当てるといったことができます。このセキュリティ ロールを動的に計算することで、そうした一時的な措置のためにアプリケーションを変更したり再デプロイしたりする必要はなくなります。さらに、本当の管理者が戻ってきたときに、特別に付与した一時的な特権を忘れずに取り消す必要もありません。なお、ユーザを一時的に `Managers` グループに追加した場合には、その必要があるでしょう。

注意： 通常は、WebLogic Server Administration Console で使用可能なロール条件を使用して、ユーザまたはグループにセキュリティ ロールを付与します (このリリースの WebLogic Server では、カスタム ロール条件を書き込むことはできません)。詳細については、『WebLogic リソースのセキュリティ』の「セキュリティ ロール」を参照してください。

計算で得られたセキュリティ ロールは、対象 (利用可能であれば) の ID 情報やリクエストのパラメータ値など、リクエストのコンテキストを構成するさまざまな情報にアクセスすることができます。こうしたコンテキスト情報は通常、WebLogic Security フレームワークで評価される式に含まれるパラメータの値として使用されます。この機能は、デプロイメント記述子または WebLogic Server Administration Console を通じて静的に定義されたセキュリティ ロールの計算も担当します。

注意 : 認証済みユーザに対するセキュリティ ロールの計算は、J2EE (Java 2 Enterprise Edition) 仕様で定義されているロールベース アクセス制御 (RBAC) によるセキュリティ機能を拡張したものです。

動的なセキュリティ ロール計算を作成するには、WebLogic Server Administration Console でロール文を定義します。詳細については、『WebLogic リソースのセキュリティ』の「セキュリティ ロール」を参照してください。

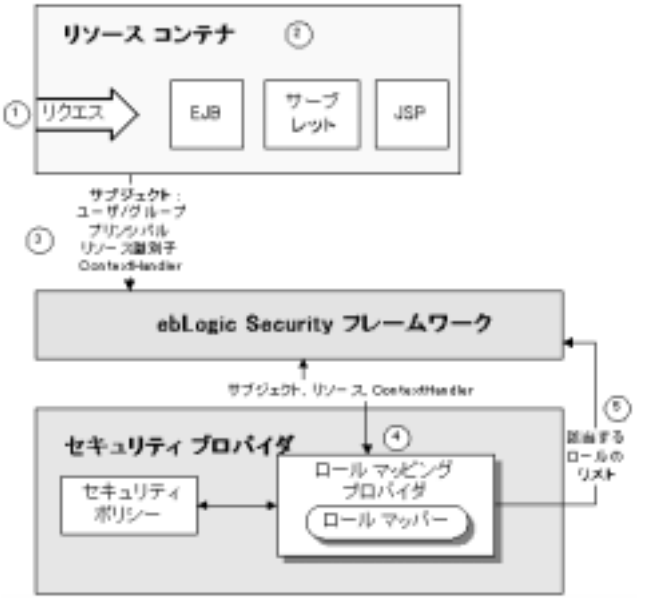
ロール マッピング プロセス

WebLogic Security フレームワークは、認可判定の一環として、セキュリティ レルム用にコンフィグレーションされている各ロール マッピング プロバイダを呼び出します。関連情報については、6-2 ページの「認可プロセス」を参照してください。

ロール マッピング プロバイダによる動的ロール関連計算の結果、一連のセキュリティ ロールが一度にサブジェクト内のプリンシパルに割り当てられます。その後、これらのセキュリティ ロールを用いて、保護対象 WebLogic リソース、リソース コンテナ、およびアプリケーション コードについての認可判定が行われます。たとえば、エンタープライズ JavaBean (EJB) であれば、アクセスを許可するかどうかを決めるビジネス ポリシーを知らなくても、J2EE (Java 2 Enterprise Edition) の `isCallerInRole()` メソッドを用いてデータベース内のレコードからフィールドを取得することができます。

動的ロール計算を作成する場合のロール マッピング プロバイダと WebLogic Security フレームワークとの対話を図 8-1 に示し、その後それについて説明します。

図 8-1 ロール マッピング プロバイダとロール マッピング プロセス



一般に、ロール マッピングは以下のように実行されます。

1. ユーザまたはシステム プロセスが WebLogic リソースを要求し、特定の操作を実行しようとします。
2. 要求された WebLogic リソースのタイプを処理するリソース コンテナがリクエストを受け取ります。たとえば EJB コンテナは EJB リソースに対するリクエストを受け取ります。

   注意： リソース コンテナは、2-24 ページの「セキュリティ プロバイダと WebLogic リソース」で説明されている WebLogic リソースのいずれかを処理するコンテナです。
3. リソース コンテナは、リクエストのコンテキストに関連付けられている情報を取得するのにロール マッピング プロバイダで使用される可能性のある `ContextHandler` オブジェクトを作成します。

   注意： ContextHandler の詳細については、2-35 ページの「ContextHandler と WebLogic リソース」を参照してください。

リソース コンテナは、JAAS サブジェクト (ユーザおよびグループ プリンシパルを含む)、WebLogic リソースの識別子、そして場合によっては追加入力を提供する `ContextHandler` オブジェクトを渡して WebLogic Security フレームワークを呼び出します。

注意 : サブジェクトの詳細については、3-2 ページの「ユーザ / グループ、プリンシパル、サブジェクト」を参照してください。また、リソース識別子の詳細については、2-26 ページの「WebLogic リソース識別子」を参照してください。

4. WebLogic Security フレームワークは、適用するセキュリティ ロールのリストを取得するために、コンフィグレーション済みの各ロール マッピング プロバイダを呼び出します。この仕組みは次のとおりです。

   a. ロール マッピング プロバイダは `ContextHandler` を用いて、リクエストに関するさまざまな情報を要求します。また、ロール マッピング プロバイダは、要求する情報のタイプを表す一連の `Callback` オブジェクトを作成します。その `Callback` オブジェクト群は、`handle` メソッドを通じて、配列として `ContextHandler` に渡されます。

      ロール マッピング プロバイダは、必要なコンテキスト情報を取得するのに、`ContextHandler` を複数回呼び出す場合があります。ロール マッピング プロバイダで `ContextHandler` が呼び出される回数はその実装によって異なります。

   b. コンテキスト情報と、セキュリティ ポリシー、サブジェクト、および WebLogic リソースを格納する関連セキュリティ プロバイダ データベースを使用することで、ロール マッピング プロバイダは、サブジェクト内のユーザ プリンシパルとグループ プリンシパルで表されるリクエスト元に特定のセキュリティ ロールの資格があるかどうかを決定します。

      セキュリティ ポリシーは、指定されたセキュリティ ロールを付与すべきかどうかを判定する際に評価される一連の式すなわちルールとして表されます。これらのルールを使用する際に、ロール マッピング プロバイダは、取得したコンテキスト情報の値を式のパラメータに代入しなければならない場合があります。さらに、ユーザ プリンシパルまたはグループ プリンシパルの ID 情報が、式のパラメータ値として必要になることもあります。

      注意 : セキュリティ ポリシーのルールは、WebLogic Server Administration Console と J2EE (Java 2 Enterprise Edition) デプロイメント記述子で設定します。詳細については、『WebLogic リソー

スのセキュリティ』の「セキュリティ ポリシー」を参照してください。

c. リクエスト元に特定のセキュリティ ロールの資格があるとセキュリティ ポリシーに指定されている場合には、そのセキュリティ ロールがサブジェクトに適用されるセキュリティ ロールのリストに追加されます。

d. このプロセスは、WebLogic リソースまたはリソース コンテナに適用されるセキュリティ ポリシーがすべて評価されるまで続行されます。

5. セキュリティ ロールのリストは WebLogic Security フレームワークに返され、アクセス決定などの操作の一環として使用できるようになります。

カスタム ロール マッピング プロバイダを開発する必要があるか

WebLogic Server のデフォルト (つまりアクティブな) セキュリティ レルムには WebLogic ロール マッピング プロバイダが含まれています。WebLogic ロール マッピング プロバイダは、デフォルト ユーザと WebLogic リソースのそれぞれについて、保護されている特定のリソースに関する特定のユーザ (サブジェクト) の動的セキュリティ ロールを計算します。また、WebLogic ロール マッピング プロバイダは、システム内のセキュリティ ロールのデプロイメントとアンデプロイメントをサポートしています。WebLogic ロール マッピング プロバイダは、WebLogic 認可プロバイダと同じポリシー エンジンを使用します。自社の既存のロール マッピング メカニズムを使用する場合は、カスタム ロール マッピング プロバイダを作成してそれを既存のメカニズムに結合できます。

カスタム ロール マッピング プロバイダの開発方法

WebLogic ロール マッピング プロバイダが開発者の要求を満たさない場合、次の手順でカスタム ロール マッピング プロバイダを開発することができます。

1. 8-8 ページの「適切な SSPI を使用して実行時クラスを作成する」
2. 8-18 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 「Administration Console を使用してカスタム ロール マッピング プロバイダをコンフィグレーションする」
4. 8-29 ページの「セキュリティ ロールを管理するためのメカニズムを提供する」

適切な SSPI を使用して実行時クラスを作成する

実行時クラスを作成する前に、以下の作業が必要です。

- 2-4 ページの「「Provider」 SSPI の目的を理解する」
- 2-5 ページの「実装する「Provider」インタフェースを決定する」
- 「SSPI 階層を理解し、実行時クラスを 1 つ作成するのか 2 つ作成するのかを決定する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム ロール マッピング プロバイダの実行時クラスを作成します。

- 8-9 ページの「RoleProvider SSPI を実装する」または 8-9 ページの「DeployableRoleProvider SSPI を実装する」
- 8-10 ページの「RoleMapper SSPI を実装する」
- 8-12 ページの「SecurityRole インタフェースの実装」

注意 : セキュリティ レルムでは、少なくとも 1 つのロール マッピング プロバイダが `DeployableRoleProvider` SSPI を実装する必要があり、そうでなければ、Web アプリケーションや EJB をデプロイすることが不可能になります。

カスタム ロール マッピング プロバイダの実行時クラスの作成例については、8-13 ページの「例 : サンプル ロール マッピング プロバイダの実行時クラスの作成」を参照してください。

RoleProvider SSPI を実装する

RoleProvider SSPI を実装するには、2-4 ページの「「Provider」 SSPI の目的を理解する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

getRoleMapper

```
public RoleMapper getRoleMapper()
```

getRoleMapper メソッドは、RoleMapper SSPI の実装を取得します。MyRoleProviderImpl.java という 1 つの実行時クラスの場合、getRoleMapper メソッドの実装は次のようになります。

```
return this;
```

実行時クラスが 2 つの場合、getRoleMapper メソッドの実装は次のようになります。

```
return new MyRoleMapperImpl;
```

この理由は、RoleProvider SSPI を実装する実行時クラスが、RoleMapper SSPI を実装するクラスを取得するためのファクトリとして使用されるからです。

RoleProvider SSPI および getRoleMapper メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

DeployableRoleProvider SSPI を実装する

DeployableRoleProvider SSPI を実装するには、2-4 ページの「「Provider」 SSPI の目的を理解する」および 8-9 ページの「RoleProvider SSPI を実装する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

deployRole

```
public void deployRole(Resource resource, java.lang.String
roleName, java.lang.String[] userAndGroupNames) throws
RoleCreationException
```

deployRole メソッドは、セキュリティ ロールの適用対象となる WebLogic リソース、アプリケーション内でのセキュリティ ロールの名前、およびセキュリティ ロールに割り当てられるユーザおよびグループ

名を基に、デプロイ済みの Web アプリケーション や EJB に代わってセキュリティ ロールを作成します。

undeployRole

```
public void undeployRole(Resource resource, java.lang.String
roleName) throws RoleRemovalException
```

`undeployRole` メソッドは、セキュリティ ロールの適用対象であった WebLogic リソースとアプリケーション内でのセキュリティ ロールの名前を基に、アンデプロイされた Web アプリケーションや EJB に代わってセキュリティ ロールを削除します。

`DeployableRoleProvider` SSPI と `deployRole` メソッドおよび `undeployRole` メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

RoleMapper SSPI を実装する

`RoleMapper` SSPI を実装するには、以下のメソッドの実装を提供する必要があります。

getRoles

```
public Map getRoles(Subject subject, Resource  resource,
ContextHandler handler)
```

`getRoles` メソッドは、場合によっては `ContextHandler` を使用して、特定の WebLogic リソースに関して特定のサブジェクトに関連付けられているセキュリティ ロールを返します。ContextHandler の詳細については、2-35 ページの「ContextHandler と WebLogic リソース」を参照してください。

`RoleMapper` SSPI および `getRoles` メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

レルム アダプタ認証プロバイダと互換性のあるカスタム ロール マッピング プロバイダの開発

認証プロバイダは、サブジェクト内へのユーザおよびグループの格納を担当するセキュリティ プロバイダです。ユーザおよびグループは、ロール マッピング プロバイダなどの他のタイプのセキュリティ プロバイダによってサブジェクトから抽出されます。セキュリティ レルムでコンフィグレーションされている認証プ

ロバイダがレルム アダプタ認証プロバイダである場合、ユーザおよびグループの情報は、他の認証プロバイダとは若干異なる方法でサブジェクトに格納されます。そのため、こうしたユーザおよびグループの情報の抽出も、若干異なる方法で行う必要があります。

サブジェクトへの格納にレルム アダプタ認証プロバイダが使用された場合に、サブジェクトがユーザ名またはグループ名に一致するかどうかを調べるためにカスタム ロール マッピング プロバイダで使用できるコードをコード リスト 8-1 に示します。このコードは、getRoles メソッドに記述できます。

コード リスト 8-1 サブジェクトがユーザ名またはグループ名に一致するかどうかを調べるためのサンプル コード

```
/**
 * サブジェクトがユーザ名またはグループ名に一致するかどうかを調べる
 *
 * @param principalWant このロールのプリンシパル名を含む文字列
 * ( つまり、ロール定義 )
 *
 * @param subject リソースにアクセスしようとしているユーザおよびそのユーザのグループを
 * 識別するプリンシパルを含むサブジェクト
 *
 * @return ブール値。現在のサブジェクトがロールのプリンシパルに
 * 一致する場合は true、それ以外の場合は false
 */
private boolean subjectMatches(String principalWant, Subject subject)
{
   // まず、グループ名が一致しているかどうかを調べる
   if (SubjectUtils.isUserInGroup(subject, principalWant)) {
      return true;
   }
   // 次に、ユーザ名が一致しているかどうかを調べる
   if (principalWant.equals(SubjectUtils.getUsername(subject))) {
      return true;
   }
   // 一致しなかった場合
   return false;
}
```

SecurityRole インタフェースの実装

SecurityRole インタフェースのメソッドを使用して、セキュリティ ロールに関する基本的な情報を取得したり、取得した情報を別のセキュリティ ロールと比較したりできます。これらのメソッドは、セキュリティ プロバイダの利便性のために設計されています。

注意： SecurityRole 実装は、getRoles() メソッドによって Map として返されます。8-10 ページの「RoleMapper SSPI を実装する」を参照してください。

SecurityRole インタフェースを実装するには、以下のメソッドの実装を提供する必要があります。

equals

```
public boolean equals( java.lang.Object another )
```

equals メソッドは、渡されたセキュリティ ロールが、このインタフェースの実装によって表されるセキュリティロールに一致する場合に TRUE を返し、それ以外の場合は FALSE を返します。

toString

```
public String toString()
```

toString メソッドは、このセキュリティ ロールを文字列で返します。

hashCode

```
public int hashCode()
```

hashCode メソッドは、このセキュリティ ロールのハッシュコードを整数で返します。

getName

```
public String getName()
```

getName メソッドは、このセキュリティ ロールの名前を文字列で返します。

getDescription

```
public String getDescription()
```

getDescription メソッドは、このセキュリティ ロールの説明を文字列で返します。説明は、このセキュリティ ロールの目的を記述したものです。

例：サンプル ロール マッピング プロバイダの実行時クラスの作成

コード リスト 8-2 は、サンプル ロール マッピング プロバイダの実行時クラスである SampleRoleMapperProviderImpl.java クラスを示しています。実行時クラスには以下の実装が含まれています。

- initialize、getDescription、および shutdown という SecurityProvider インタフェースから継承した 3 つのメソッド (2-4 ページの「「Provider」 SSPI の目的を理解する」 を参照)
- getRoleMapper という RoleProvider SSPI から継承したメソッド (8-9 ページの「RoleProvider SSPI を実装する」 を参照)
- deployRole および undeployRole という DeployableRoleProvider SSPI の 2 つのメソッド (8-9 ページの「DeployableRoleProvider SSPI を実装する」 を参照)
- getRoles という RoleMapper SSPI のメソッド (8-10 ページの「RoleMapper SSPI を実装する」 を参照)

注意： コード リスト 8-2 の太字のコードは、クラス宣言とメソッド シグネチャを示しています。

コード リスト 8-2 SampleRoleMapperProviderImpl.java

```
package examples.security.providers.roles;

import java.security.Principal;
import java.util.Collections;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.Properties;
import java.util.Set;
import javax.security.auth.Subject;
import weblogic.management.security.ProviderMBean;
import weblogic.security.WLSPrincipals;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.DeployableRoleProvider;
import weblogic.security.spi.Resource;
```

```
import weblogic.security.spi.RoleCreationException;
import weblogic.security.spi.RoleMapper;
import weblogic.security.spi.RoleRemovalException;
import weblogic.security.spi.SecurityServices;

public final class SampleRoleMapperProviderImpl implements
DeployableRoleProvider, RoleMapper
{
   private String description;
   private SampleRoleMapperDatabase database;
   private static final Map NO_ROLES = Collections.unmodifiableMap(new
     HashMap(1));

   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SampleRoleMapperProviderImpl.initialize");
      SampleRoleMapperMBean myMBean = (SampleRoleMapperMBean)mbean;
      description = myMBean.getDescription() + "\n" + myMBean.getVersion();
      database = new SampleRoleMapperDatabase(myMBean);
   }

  public String getDescription()
   {
      return description;
   }

   public void shutdown()
   {
      System.out.println("SampleRoleMapperProviderImpl.shutdown");
   }

  public RoleMapper getRoleMapper()
   {
      return this;
   }

   public Map getRoles(Subject subject, Resource resource, ContextHandler
   handler)
   {
      System.out.println("SampleRoleMapperProviderImpl.getRoles");
      System.out.println("\tsubject\t= " + subject);
      System.out.println("\tresource\t= " + resource);

      Map roles = new HashMap();
      Set principals = subject.getPrincipals();

      for (Resource res = resource; res != null; res = res.getParentResource())
      {
         getRoles(res, principals, roles);
      }
```

```
   getRoles(null, principals, roles);

   if (roles.isEmpty()) {
      return NO_ROLES;
   }

   return roles;
}

public void deployRole(Resource resource, String roleName, String[]
principalNames) throws RoleCreationException
{
   System.out.println("SampleRoleMapperProviderImpl.deployRole");
   System.out.println("\tresource\t\t= " + resource);
   System.out.println("\troleName\t\t= " + roleName);

   for (int i = 0; principalNames != null && i < principalNames.length; i++)
   {
      System.out.println("\tprincipalNames[" + i + "]\t= " +
      principalNames[i]);
   }

   database.setRole(resource, roleName, principalNames);
}

public void undeployRole(Resource resource, String roleName) throws
RoleRemovalException
{
   System.out.println("SampleRoleMapperProviderImpl.undeployRole");
   System.out.println("\tresource\t= " + resource);
   System.out.println("\troleName\t= " + roleName);

   database.removeRole(resource, roleName);
}

private void getRoles(Resource resource, Set principals, Map roles)
{
   for (Enumeration e = database.getRoles(resource); e.hasMoreElements();)
   {
      String role = (String)e.nextElement();
      if (roleMatches(resource, role, principals))
      {
         roles.put(role, new SampleSecurityRoleImpl(role, "no description"));
      }
   }
}

private boolean roleMatches(Resource resource, String role, Set
principalsHave)
{
   for (Enumeration e = database.getPrincipalsForRole(resource, role);
```

```
        e.hasMoreElements();)
      {
         String principalWant = (String)e.nextElement();
         if (principalMatches(principalWant, principalsHave))
         {
            return true;
         }
      }
      return false;
   }

   private boolean principalMatches(String principalWant, Set principalsHave)
   {
      if (WLSPrincipals.getEveryoneGroupname().equals(principalWant) ||
        (WLSPrincipals.getUsersGroupname().equals(principalWant) &&
        !principalsHave.isEmpty()) || (WLSPrincipals.getAnonymousUsername().
        equals(principalWant) && principalsHave.isEmpty()) ||
        principalsContain(principalsHave, principalWant))
        {
           return true;
        }
      return false;
   }

   private boolean principalsContain(Set principalsHave, String
   principalNameWant)
   {
      for (Iterator i = principalsHave.iterator(); i.hasNext();)
      {
         Principal principal = (Principal)i.next();
         String principalNameHave = principal.getName();
         if (principalNameWant.equals(principalNameHave))
         {
            return true;
         }
      }
      return false;
   }
}
```

コード リスト 8-3 は、SampleRoleMapperProviderImpl.java 実行時クラスと共に使用される SecurityRole 実装を示しています。

コード リスト 8-3 SampleSecurityRoleImpl.java

```
package examples.security.providers.roles;

import weblogic.security.service.SecurityRole;

public class SampleSecurityRoleImpl implements SecurityRole
{
   private String _roleName;
   private String _description;
   private int _hashCode;

   public SampleSecurityRoleImpl(String roleName, String description)
   {
      _roleName = roleName;
      _description = description;
      _hashCode = roleName.hashCode() + 17;
   }

   public boolean equals(Object secRole)
   {
      if (secRole == null)
      {
         return false;
      }

      if (this == secRole)
      {
         return true;
      }

      if (!(secRole instanceof SampleSecurityRoleImpl))
      {
         return false;
      }

      SampleSecurityRoleImpl anotherSecRole = (SampleSecurityRoleImpl)secRole;

      if (!_roleName.equals(anotherSecRole.getName()))
      {
         return false;
      }

      return true;
   }

   public String toString () { return _roleName; }
   public int hashCode () { return _hashCode; }
   public String getName () { return _roleName; }
```

```
   public String getDescription () { return _description; }
}
```

WebLogic MBeanMaker を使用して MBean タイプを生成する

カスタム セキュリティ プロバイダの MBean タイプを生成する前に、以下の作業が必要です。

- 2-11 ページの「MBean タイプが必要な理由を理解する」
- 2-12 ページの「拡張および実装する SSPI MBean を決定する」
- 2-13 ページの「MBean 定義ファイル (MDF) の基本的な要素を理解する」
- 2-15 ページの「SSPI MBean の階層と Administration Console に対する影響を理解する」
- 2-17 ページの「WebLogic MBeanMaker によって提供されるものを理解する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム ロール マッピング プロバイダの MBean タイプを作成します。

1. 8-19 ページの「MBean 定義ファイル (MDF) を作成する」
2. 8-19 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 8-23 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」
4. 8-24 ページの「WebLogic Server 環境に MBean タイプをインストールする」

注意： これらの手順の実行方法は、いくつかのサンプル セキュリティ プロバイダ (dev2dev Web サイトの「Code Samples: WebLogic Server」で入手可能) に示されています。

この節で説明する手順はすべて、Windows 環境での作業を想定しています。

MBean 定義ファイル (MDF) を作成する

MBean 定義ファイル (MDF) を作成するには、次の手順に従います。

1. サンプル ロール マッピング プロバイダの MDF をテキスト ファイルにコピーします。

   注意： サンプル ロール マッピング プロバイダの MDF は、`SampleRoleMapper.xml` という名前です。

2. カスタム ロール マッピング プロバイダに合わせて、作成する MDF の `<MBeanType>` および `<MBeanAttribute>` 要素の内容を変更します。

3. カスタム属性および操作 (つまり、`<MBeanAttribute>` および `<MBeanOperation>` 要素) を MDF に追加します。

4. ファイルを保存します。

注意： MDF 要素の構文についての完全なリファレンスは、付録 A「MBean 定義ファイル (MDF) 要素の構文」に収められています。

WebLogic MBeanMaker を使用して MBean タイプを生成する

MDF を作成したら、WebLogic MBeanMaker を使用してそれを実行できます。WebLogic MBeanMaker は現在のところコマンドライン ユーティリティで、入力として MDF を受け取り、MBean インタフェース、MBean 実装、関連する MBean 情報ファイルなどの中間 Java ファイルをいくつか出力します。これらの中間ファイルが合わさって、カスタム セキュリティ プロバイダの **MBean タイプ**になります。

MBean タイプの生成手順は、カスタム ロール マッピング プロバイダの設計に応じて異なります。必要な設計に合わせて適切な手順を実行してください。

- 8-20 ページの「カスタム操作を追加しない場合」
- 8-20 ページの「カスタム操作を追加する場合」

カスタム操作を追加しない場合

カスタム ロール マッピング プロバイダの MDF にカスタム操作を含めない場合、次の手順に従います。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

   `xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

   `-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

   注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまりロール マッピング プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

3. 8-23 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

カスタム操作を追加する場合

カスタム ロール マッピング プロバイダの MDF にカスタム操作を含める場合、質問に答えながら手順を進めてください。

- MBean タイプを作成するのは初めてですか。初めての場合は、次の手順に従ってください。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

ここで、-DMDF フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*xmlFile* は MDF (XML MBean 定義ファイル)、*filesdir* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

xmlfile が入力されるたびに、新しい出力ファイル群が生成されます。*filesdir* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

-DcreateStubs=true フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまりロール マッピング プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

3. MDF のすべてのカスタム操作に対して、メソッド スタブを使用してメソッドを実装します。
4. ファイルを保存します。
5. 8-23 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

■ 既存の MBean タイプの更新ですか。更新の場合は、次の手順に従ってください。

1. WebLogic MBeanMaker によって現在のメソッドの実装が上書きされないように、既存の MBean 実装ファイルを一時ディレクトリにコピーします。
2. 新しい DOS シェルを作成します。
3. 次のコマンドを入力します。

```
java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
weblogic.management.commo.WebLogicMBeanMaker
```

ここで、-DMDF フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*xmlFile* は MDF (XML MBean 定義ファイル)、*filesdir* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

xmlfile が入力されるたびに、新しい出力ファイル群が生成されます。*filesdir* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

`-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまりロール マッピング プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

4. MDF を変更して元の MDF にはないカスタム操作を含めた場合、メソッド スタブを使用してメソッドを実装します。
5. 完成した、つまりすべてのメソッドを実装した MBean 実装ファイルを保存します。
6. この MBean 実装ファイルを、WebLogic MBeanMaker が MBean タイプの実装ファイルを配置したディレクトリにコピーします。このディレクトリは、手順 3 で *`filesdir`* として指定しました (手順 3 の結果として WebLogic MBeanMaker で生成された MBean 実装ファイルがオーバーライド される)。
7. 8-23 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

生成される MBean インタフェース ファイルについて

MBean インタフェース ファイルとは、実行時クラスまたは MBean 実装がコンフィグレーション データを取得するために使用する MBean のクライアントサイド API です。2-4 ページの「「Provider」 SSPI の目的を理解する」で説明されているように、これは initialize メソッドで使用するのが一般的です。

WebLogic MBeanMaker では、作成済みの MDF から MBean タイプを生成するので、生成される MBean インタフェース ファイルの名前は、その MDF 名の後に「MBean」というテキストが付いたものになります。たとえば、WebLogic MBeanMaker で `SampleRoleMapper` MDF を実行すると、`SampleRoleMapperMBean.java` という MBean インタフェース ファイルが生成されます。

WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する

WebLogic MBeanMaker で MDF を実行して中間ファイルを作成し、MBean 実装ファイルを手作業で編集してその中にメソッドの内容を記述したら、カスタム ロール マッピング プロバイダの MBean ファイルと実行時クラスを MBean JAR ファイル (MJF) にパッケージ化する必要があります。このプロセスも、WebLogic MBeanMaker によって自動化されます。

カスタム ロール マッピング プロバイダの MJF を作成するには、次の手順に従います。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

   ```
   java -DMJF=jarfile -Dfiles=filesdir
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMJF` フラグは WebLogic MBeanMaker が新しい MBean タイプを格納する JAR ファイルをビルドする必要があることを示し、*`jarfile`* は MJF の名前、*`filesdir`* は WebLogic MBeanMaker で MJF に JAR 化する対象ファイルが存在する場所です。

   この時点でコンパイルが行われるので、エラーが発生するおそれがあります。*`jarfile`* が指定されていて、エラーが発生しなかった場合には、指定された名前の MJF が作成されます。

注意： 既存の MJF を更新する場合は、MJF をいったん削除してから再生成します。WebLogic MBeanMaker にも `-DIncludeSource` オプションがあり、それを指定すると、生成される MJF にソース ファイルを含めるかどうかを制御することができます。ソース ファイルには、生成されたソースと MDF そのものがあります。デフォルトは `false` です。このオプションは、`-DMJF` を使用しない場合には無視されます。

生成された MJF は、自らの WebLogic Server 環境にインストールすることも、顧客に配布してそれぞれの WebLogic Server 環境にインストールしてもらうこともできます。

WebLogic Server 環境に MBean タイプをインストールする

MBean タイプを WebLogic Server 環境にインストールするには、MJF を *WL_HOME*\server\lib\mbeantypes ディレクトリにコピーします。ここで、*WL_HOME* は WebLogic Server の最上位のインストール ディレクトリです。このインストール コマンドによって、カスタム ロール マッピング プロバイダが「デプロイ」されます。つまり、カスタム ロール マッピング プロバイダを WebLogic Server Administration Console から管理できるようになります。

注意： *WL_HOME*\server\lib\mbeantypes は、MBean タイプをインストールするデフォルトのディレクトリです。ただし、WebLogic Server が追加ディレクトリで MBean タイプを検索するようにするには、サーバを起動するときに -Dweblogic.alternateTypesDirectory=<*dir*> コマンドライン フラグを使用します (<*dir*> はディレクトリ名のカンマ区切りのリスト)。このフラグを使用すると、WebLogic Server は常にまず *WL_HOME*\server\lib\mbeantypes から MBean タイプをロードし、その後に追加ディレクトリを検索して、(拡張子に関係なく) そのディレクトリに存在するすべての有効なアーカイブをロードします。たとえば -Dweblogic.alternateTypesDirectory = dirX,dirY の場合、WebLogic Server はまず *WL_HOME*\server\lib\mbeantypes から MBean タイプをロードし、その後に、dirX および dirY に存在する有効なアーカイブをロードします。

追加ディレクトリで MBean タイプを検索するように WebLogic Server に指示し、かつ Java セキュリティ マネージャを使用する場合は、weblogic.policy ファイルを更新して、MBean タイプ (したがってカスタム セキュリティ プロバイダも) に対する適切なパーミッションを与える必要があります。詳細については、『WebLogic Security プログラマーズ ガイド』の「Java セキュリティ マネージャを使用しての WebLogic リソースの保護」を参照してください。

非セキュリティ プロバイダの JAR (バックアップ ファイルを含む) は、*WL_HOME*\server\lib\mbeantypes ディレクトリ以外の場所に置くことをお勧めします。

カスタム ロール マッピング プロバイダをコンフィグレーションすることによって (8-25 ページの「Administration Console を使用してカスタム ロール マッピング プロバイダをコンフィグレーションする」を参照)、MBean タイプのインスタンスを作成して、GUI、他の Java コード、または API からそれらの MBean イ

ンスタンスを使用することができます。たとえば、WebLogic Server Administration Console を使用して、属性を取得 / 設定したり操作を呼び出したりすることもできますし、他の Java オブジェクトを開発して、そのオブジェクトで MBean をインスタンス化し、それらの MBean から提供される情報に自動的に応答させることもできます。なお、これらの MBean インスタンスをバックアップしておくことをお勧めします。詳細については、『WebLogic Server ドメイン管理』の「障害が発生したサーバの回復」の「セキュリティ コンフィグレーション データのバックアップ」を参照してください。

Administration Console を使用してカスタム ロール マッピング プロバイダをコンフィグレーションする

カスタム ロール マッピング プロバイダをコンフィグレーションするということは、ロール マッピング サービスを必要とするアプリケーションがアクセス可能なセキュリティ レルムにカスタム ロール マッピング プロバイダを追加するということです。

カスタム セキュリティ プロバイダのコンフィグレーションは管理タスクですが、カスタム セキュリティ プロバイダの開発者が行うこともできます。この節では、カスタム ロール マッピング プロバイダのコンフィグレーション担当者向けの重要な情報を取り上げます。

- 8-26 ページの「ロール マッピング プロバイダとデプロイメント記述子の管理」
- 8-28 ページの「セキュリティ ロール デプロイメントの有効化」

注意： WebLogic Server Administration Console を使用したカスタム ロール マッピング プロバイダのコンフィグレーション手順については、『WebLogic Security の管理』の「カスタム セキュリティプロバイダのコンフィグレーション」を参照してください。

ロール マッピング プロバイダとデプロイメント記述子の管理

エンタープライズ JavaBean (EJB) や Web アプリケーションなどのアプリケーション コンポーネントの中には、関連デプロイメント情報を Java 2 Enterprise Edition (J2EE) デプロイメント記述子および WebLogic Server デプロイメント記述子に格納するものがあります。Web アプリケーションの場合、デプロイメント記述子ファイル (`web.xml` と `weblogic.xml`) には、セキュリティ ロールを含む J2EE セキュリティ モデルの実装情報が格納されます。この情報は、WebLogic Server Administration Console でロール マッピング プロバイダを初めてコンフィグレーションするときに格納するのが一般的です。

Administration Console には、この目的のために [デプロイメント記述子内のセキュリティ データを無視] チェックボックスが用意されています。開発者または管理者がカスタム ロール マッピング プロバイダを初めてコンフィグレーションするときには、このチェックボックスのチェックがはずれていることを確認する必要があります。

注意： [デプロイメント記述子内のセキュリティ データを無視] チェックボックスは、デフォルトではチェックがはずれています。このチェックボックスを設定するには、WebLogic Server Administration Console の左ペインで [セキュリティ | レルム | *`realm`*] をクリックします。*`realm`* はセキュリティ レルムの名前です。次に [一般] タブを選択します。

このチェックボックスのチェックをはずして Web アプリケーションをデプロイすると、WebLogic Server は、`web.xml` および `weblogic.xml` デプロイメント記述子ファイル (コード リスト 8-4 およびコード リスト 8-5 の `web.xml` および `weblogic.xml` の例を参照) から情報を読み込みます。この情報は、ロール マッピング プロバイダのセキュリティ プロバイダ データベースにコピーされます。

コード リスト 8-4　web.xml ファイルのサンプル

```
<web-app>

   <welcome-file-list>
      <welcome-file>welcome.jsp</welcome-file>
   </welcome-file-list>

   <security-constraint>
      <web-resource-collection>
         <web-resource-name>Success</web-resource-name>
         <url-pattern>/welcome.jsp</url-pattern>
```

```
            <http-method>GET</http-method>
            <http-method>POST</http-method>
         </web-resource-collection>
         <auth-constraint>
            <role-name>developers</role-name>
         </auth-constraint>
      </security-constraint>

      <login-config>
         <auth-method>BASIC</auth-method>
         <realm-name>default</realm-name>
      </login-config>

      <security-role>
         <role-name>developers</role-name>
      </security-role>

</web-app>
```

コード リスト 8-5　weblogic.xml ファイルのサンプル

```
<weblogic-web-app>
   <security-role-assignment>
      <role-name>developers</role-name>
      <principal-name>myGroup</principal-name>
   </security-role-assignment>
</weblogic-web-app>
```

web.xml/weblogic.xml デプロイメント記述子ファイルでも WebLogic Server Administration Console でも追加のセキュリティ ロールを設定することができますが、Web アプリケーションのデプロイメント記述子に定義されているセキュリティ ロールをいったんコピーしてから Administration Console を使用して追加のセキュリティ ロールを定義することをお勧めします。この理由は、ロール マッピング プロバイダのコンフィグレーション中に Administration Console を使用してセキュリティ ロールを変更すると、その内容が web.xml および weblogic.xml ファイルに保持されないからです。Administration Console を使用して再デプロイする、ディスク上で Web アプリケーションを変更した、または WebLogic Server を再起動したといった場合に、Web アプリケーションを再デプロイするときには、[デプロイメント記述子内のセキュリティ データを無視] チェックボックスをチェックする必要があります。チェックボックスをチェックしないと、Administration Console を使用して定義したセキュリティ ロールがデプロイメント記述子で定義されているセキュリティ ロールによって上書きされ

ます。詳細については、『WebLogic リソースのセキュリティ』の「組み合わせた方法による URL (Web) リソースとエンタープライズ JavaBean (EJB) リソースの保護」を参照してください。

注意： EJB についてもプロセスは同じですが、`ejb-jar.xml/weblogic-ejb-jar.xml` デプロイメント記述子を使用します。

[デプロイメント記述子内のセキュリティ データを無視] チェックボックスは、認可プロバイダおよび資格マッピング プロバイダにも影響します。詳細については、それぞれ 6-22 ページの「認可プロバイダとデプロイメント記述子の管理」と 11-17 ページの「資格マッピング プロバイダ、リソース アダプタ、およびデプロイメント記述子の管理」を参照してください。

セキュリティ ロール デプロイメントの有効化

`DeployableRoleProvider` SSPI を実装し、カスタム ロール マッピング プロバイダでデプロイ可能なセキュリティ ロールをサポートしたい場合、カスタム ロール マッピング プロバイダのコンフィグレーション担当者 (つまり、開発者または管理者) は、WebLogic Server Administration Console で [ロール デプロイメントを有効化] チェックボックスがチェックされていることを確認する必要があります。チェックがはずれていると、ロール マッピング プロバイダに対するデプロイメントは「オフ」と見なされます。このため、複数のロール マッピング プロバイダがコンフィグレーションされている場合、[ロール デプロイメントを有効化] チェックボックスを使用すると、セキュリティ ロールのデプロイメントに使用するロール マッピング プロバイダを指定できます。

注意： [デプロイメント記述子内のセキュリティ データを無視] チェックボックス (8-26 ページの「ロール マッピング プロバイダとデプロイメント記述子の管理」で説明したようにセキュリティ レルム レベルで指定) では、コンフィグレーション済みのロール マッピング プロバイダのセキュリティ データベースにセキュリティ ロールをコピーするかどうかを指定します。[ロール デプロイメントを有効化] チェックボックス (コンフィグレーション済みのロール マッピング プロバイダごとに指定) では、ロール マッピング プロバイダがデプロイ済みのセキュリティ ロールを格納するかどうかを指定します。

セキュリティ ロールを管理するためのメカニズムを提供する

WebLogic Server Administration Console を使用してカスタム ロール マッピング プロバイダをコンフィグレーションすると、必要なロール マッピング サービスにアプリケーションからアクセスできるようにすることはできますが、このセキュリティ プロバイダに関連付けられたセキュリティ ロールを管理する方法を管理者にも提供する必要があります。たとえば WebLogic ロール マッピング プロバイダには、さまざまな WebLogic リソースのセキュリティ ロールを追加、変更、または削除するためのロール エディタ ページ (図 8-2 を参照) が管理者向けに用意されています。このページは、特定のグローバル ロールまたはスコープ ロールの [条件] タブに表示されます。

図 8-2 WebLogic ロール マッピング プロバイダのロール エディタ ページ

管理者は、カスタム ロール マッピング プロバイダを開発するときに、ロール エディタ ページも右クリック メニューも利用できません。したがって、セキュリティ ロールを管理するための独自のメカニズムを提供する必要があります。このメカニズムでは、カスタム ロール マッピング プロバイダのデータベースのセキュリティ ロール データ (つまり式) を読み書きできなければなりません。

それには、以下の 3 通りの方法があります。

オプション 1 : コンソール拡張を使用して独自の「ロール エディタ」ページを作成する

カスタム ロール マッピング プロバイダ用にコンソール拡張を作成する方法の主な利点は、コンソール拡張によって WebLogic リソースの ID がわかるので、その WebLogic リソースがリソース階層のどこにあるかもわかるということです。この情報は、ロール マッピング プロバイダのデータベースから式を読み書きするために必要です。また、WebLogic ロール マッピング プロバイダのロール エディタ ページのように、作成したページを既存の WebLogic Server Administration Console GUI に統合できるという利点もあります。

この方法を選択した場合、次の作業が必要になります。

1. `weblogic.management.console.extensibility.SecurityExtension` インタフェースの `getExtensionForRole()` メソッドを実装し、作成したロール エディタ ページをこのメソッドで返します。

   注意 : 詳細については、第 12 章「カスタム セキュリティ プロバイダ用のコンソール拡張の記述」を参照してください。

2. また、以下のいずれかを実行する必要があります。

   a. `RoleEditor` および `RoleReader` 認可用任意 SSPI MBean を実装し、作成したロール エディタ ページとロール マッピング プロバイダのデータベースとの間の仲介役となる管理 MBean を開発します。詳細については、2-12 ページの「拡張および実装する SSPI MBean を決定する」と 2-22 ページの表 2-4「認可用任意 SSPI MBean」を参照してください。

      この場合、文字列として表現可能なセキュリティ ロールを構成する式 (`Role=Admin` や `Group=Administrators` など) の構文も開発する必要があります。

注意： この構文は、ロール マッピング プロバイダごとに異なっていてもかまいません。式の詳細については、『WebLogic リソースのセキュリティ』の「セキュリティ ロールの構成要素：ロール条件、式、およびロール文」を参照してください。

b. セキュリティ ロールを管理するための MBean API を開発し、それらのインタフェースを実装し、作成したロール エディタ ページとロール マッピング プロバイダのデータベースとの間の仲介役となる管理 MBean を開発します。

c. MBean に委託しないで、作成したページから直接、カスタム ロール マッピング プロバイダのデータベースの式を読み書きします。

オプション 2：セキュリティ ロール管理用のスタンドアロン ツールを開発する

WebLogic Server Administration Console とまったく別のツールを開発する場合には、この方法を選択します。

この方法の場合、8-30 ページの「オプション 1：コンソール拡張を使用して独自の「ロール エディタ」ページを作成する」で説明したカスタム ロール マッピング プロバイダ用のコンソール拡張を記述する必要もなく、管理 MBean を開発する必要もありません。ただし、ツールでは以下のことを行う必要があります。

1. WebLogic リソースの ID を特定します。ID はコンソール拡張によって自動的に提供されません。詳細については、2-26 ページの「WebLogic リソース識別子」を参照してください。
2. セキュリティ ロールを構成する式を表す方法を決定します。この表現は完全に任意で、8-30 ページの「オプション 1：コンソール拡張を使用して独自の「ロール エディタ」ページを作成する」と違って文字列である必要はありません。
3. カスタム ロール マッピング プロバイダのデータベースの式を読み書きします。

オプション 3 : 既存のセキュリティ ロール管理ツールを Administration Console に統合する

WebLogic Server Administration Console とは別のツールを持っており、それを Administration Console から起動する場合には、この方法を選択します。

この方法の場合、ツールでは以下のことを行う必要があります。

1. WebLogic リソースの ID を特定します。ID はコンソール拡張によって自動的に提供されません。詳細については、2-26 ページの「WebLogic リソース識別子」を参照してください。
2. セキュリティ ロールを構成する式を表す方法を決定します。この表現は完全に任意で、8-30 ページの「オプション 1 : コンソール拡張を使用して独自の「ロール エディタ」ページを作成する」と違って文字列である必要はありません。
3. カスタム ロール マッピング プロバイダのデータベースの式を読み書きします。
4. 『Administration Console の拡張』に説明されているように、基本コンソール拡張を使用して Administration Console にリンクします。

9　監査プロバイダ

監査とは、リクエストの操作とそれらのリクエストの結果に関する情報を、否認防止を目的として収集、格納、および配布するプロセスのことです。WebLogic Server では、監査はコンピュータのアクティビティの電子的な記録を提供するものです。

以下の節では、監査プロバイダの概念と機能、およびカスタム監査プロバイダの開発手順について説明します。

監査の概念

監査プロバイダを開発する前に、以下の概念を理解しておく必要があります。

監査チャネル

監査チャネルは監査プロバイダのコンポーネントで、セキュリティ イベントを監査すべきかどうかを決定し、サービス品質 (QoS) ポリシーに基づいて監査情報を実際に記録します。

注意： 監査チャネルの詳細については、9-8 ページの「AuditChannel SSPI を実装する」を参照してください。

カスタム セキュリティ プロバイダからのイベントの監査

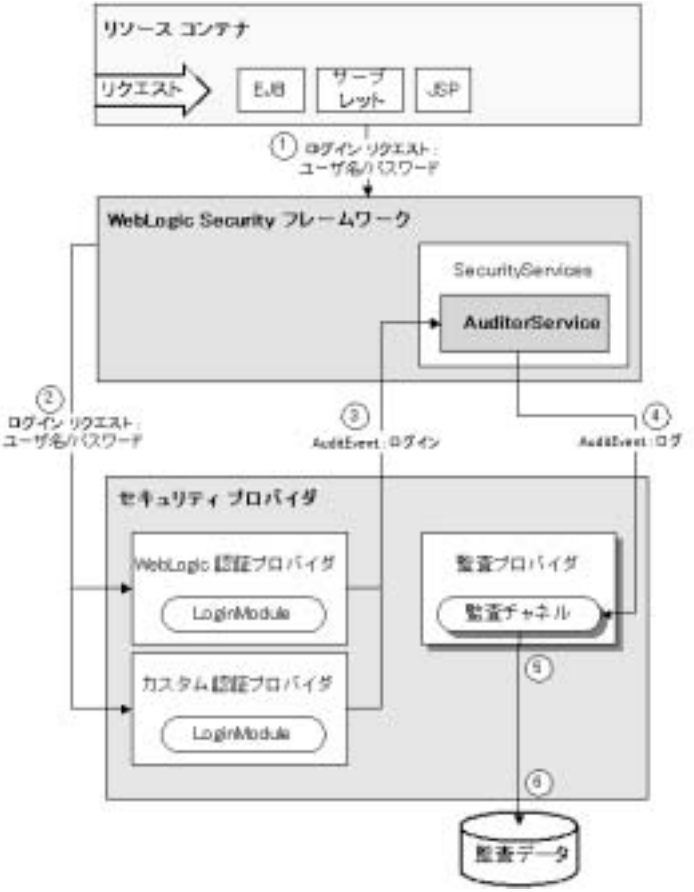
各タイプのセキュリティ プロバイダは、セキュリティ関連イベントの実行前または実行後に、それらのイベントに関する情報を記録するようコンフィグレーション済みの監査プロバイダに要求できます。たとえば、あるユーザが預金口座アプリケーションの (アクセス権を持たない) `withdraw` メソッドにアクセスしようとした場合、認可プロバイダはこの操作を記録するよう要求できます。セキュリティ関連イベントは、監査プロバイダのコンフィグレーションで指定されている重大度レベルと一致するか、それを超えた場合にのみ記録されます。

カスタム セキュリティ プロバイダから監査イベントをポストする方法については、第 10 章「カスタム セキュリティ プロバイダからのイベントの監査」を参照してください。

監査プロセス

図 9-1 に、監査プロバイダが WebLogic Security フレームワークと他のタイプのセキュリティ プロバイダ (ここでは認証プロバイダ) と対話して、選択されたイベントを監査する仕組みを示します。図の後には、詳しい説明が続きます。

図 9-1　セキュリティ プロバイダ、WebLogic Security フレームワーク、および他のセキュリティ プロバイダ



監査プロバイダは、次のように WebLogic Security フレームワークと他のタイプのセキュリティ プロバイダと対話します。

注意： 図 9-1 と以下の説明文で、「他のタイプのセキュリティ プロバイダ」とは WebLogic 認証プロバイダとカスタム認証プロバイダです。ただし、第 10 章「カスタム セキュリティ プロバイダからのイベントの監査」で説明するように、これらはどのようなタイプのセキュリティ プロバイダでも構いません。

1. リソース コンテナは、ログイン リクエストの一部としてユーザの認証情報 (ユーザ名とパスワードの組み合わせなど) を WebLogic Security フレームワークに渡します。
2. WebLogic Security フレームワークは、ログイン リクエストに関連付けられている情報をコンフィグレーション済みの認証プロバイダに渡します。
3. 認証サーバが認証サービスを提供するほかに監査イベントをポストする場合、各認証プロバイダは以下のことを行います。
   a. `AuditEvent` オブジェクトをインスタンス化します。`AuditEvent` オブジェクトには、少なくとも監査対象のイベント タイプに関する情報と監査の重大度レベルが含まれます。

      注意 : `AuditEvent` クラスを作成するには、認証プロバイダの実行時クラスに `AuditEvent` SSPI または `AuditEvent` コンビニエンス インタフェースを実装し、さらにカスタム認証プロバイダが既に実装している必要がある他のセキュリティ プロバイダ インタフェース (SSPI) を実装します。監査イベントと `AuditEvent` SSPI/ コンビニエンス インタフェースについては、10-4 ページの「監査イベントを作成する」を参照してください。
   b. 監査サービスに対して信頼性のある呼び出しを行って、`AuditEvent` オブジェクトを渡します。

      注意 : これが信頼性のある呼び出しである理由は、監査サービスが既にセキュリティ プロバイダの `initialize` メソッドに「Provider」SSPI 実装の一部として追加されているからです。詳細については、2-4 ページの「「Provider」SSPI の目的を理解する」を参照してください。
4. 監査サービスは、`AuditEvent` オブジェクトをコンフィグレーション済みの監査プロバイダの実行時クラス (`AuditChannel` SSPI 実装) に渡します。

   注意 : 認証プロバイダの `AuditEvent` コンビニエンス インタフェースの実装によっては、監査リクエストがイベントに対して 1 回だけでなく、イベントの前後にも発生する場合があります。
5. 監査プロバイダの実行時クラスは、`AuditEvent` オブジェクトから取得したイベント タイプ、監査重大度、およびその他の情報 (監査コンテキストなど) を利用して監査レコードの内容を管理します。一般に、コンフィグレーション済みの監査プロバイダの 1 つだけがすべての監査条件を満たします。

注意： 監査重大度レベルと監査コンテキストの詳細については、それぞれ 10-9 ページの「監査重大度」と 10-9 ページの「監査コンテキスト」を参照してください。

6. `AuditEvent` オブジェクト内に認証プロバイダによって指定された監査条件が満たされると、該当する監査プロバイダの実行時クラス (`AuditChannel` SSPI 実装) はそれらの実装によって指定された方法で監査レコードを書き出します。

   注意： `AuditChannel` SSPI 実装によっては、監査条件が満たされたときに監査記録がファイルやデータベースなどの永続ストレージ メディアに書き込まれます。

カスタム監査プロバイダを開発する必要があるか

WebLogic Server のデフォルト (つまりアクティブな) セキュリティ レルムには WebLogic 監査プロバイダが含まれています。WebLogic 監査プロバイダは、WebLogic Server ドメイン コンフィグレーションに対するすべての変更の情報を記録します。たとえば、変更、編集、または削除された属性値や、実行された操作などです。

WebLogic 監査プロバイダは、その `writeEvent` メソッドで、コンフィグレーション済みの監査重大度レベルとそのメソッドに渡された `AuditEvent` オブジェクトに格納されている監査重大度に基づいて監査するかどうかの決定を行います (`AuditEvent` オブジェクトの詳細については、10-4 ページの「監査イベントを作成する」を参照)。

注意： WebLogic 監査プロバイダのコンフィグレーション済み監査重大度レベルは、WebLogic Server Administration Console で変更できます。詳細については、『WebLogic Security の管理』の「WebLogic 監査プロバイダのコンフィグレーション」を参照してください。

一致が存在する場合、WebLogic 監査プロバイダは監査情報を `DefaultAuditRecorder.log` ファイルに書き込みます。このファイルは、*`bea_home`*`\user_projects\`*`domain`* ディレクトリにあります (*`bea_home`* は 1 つの

マシンにインストールされているすべての BEA 製品の中央サポート ディレクトリ、*domain* は作成したドメインの名前)。コード リスト 9-1 は、DefaultAuditRecorder.log ファイルの一部を例として示しています。

コード リスト 9-1　DefaultAuditRecorder.log ファイル : 出力例

When an attribute value is changed.[SUCCESS]

```
####<Sep 23, 2003 4:29:55 PM EDT> <Info> <Configuration Audit> <PORTNAV>
<myserver> <main> <kernel identity> <> <159904> <USER kernel identity MODIFIED
mydomain:Name=uddiexplorer,Type=Application ATTRIBUTE Deployed FROM false TO
true>####<Sep 23, 2003 4:31:23 PM EDT> <Info> <Configuration Audit> <PORTNAV>
<myserver> <ExecuteThread: '12' for queue: 'default'> <kernel identity> <>
<159904> <USER system MODIFIED mydomain:Name=myserver,Type=Server ATTRIBUTE
StdoutSeverityLevel FROM 64 TO 32>
```

When Operations are invoked.[SUCCESS]

```
####<Sep 23, 2003 4:30:30 PM EDT> <Info> <Configuration Audit> <PORTNAV>
<myserver> <ExecuteThread: '12' for queue: 'default'> <kernel identity> <>
<159907> <USER system INVOKED ON Security:Name=myrealmDefaultAuthenticator METHOD
listUsersPARAMS *; 48>
```

[FAILURE] Any failures(CREATE,DELETE,INVOKE) are followed by the exception that occured during the operation.

```
####<Sep 23, 2003 4:30:25 PM EDT> <Info> <Configuration Audit> <PORTNAV>
<myserver> <ExecuteThread: '12' for queue: 'default'> <kernel identity> <>
<159908> <USER <anonymous> INVOKED ON
Security:Name=myrealmDefaultIdentityAsserter METHOD userExists PARAMS weblogic
FAILED javax.management.MBeanException: An exception occured in
RequiredModelMBean while trying to invoke an operation>
javax.management.ServiceNotFoundException: operation userExists execution not
supported from descriptor data
```

When a new MBean is created.[SUCCESS]

```
<Sep 23, 2003 5:18:45 PM EDT> <Info> <Configuration Audit> <159900> <USER system
CREATED mydomain:Name=managed-1,Type=Server>
```

When an MBean is deleted.[SUCCESS]

```
<Sep 12, 2003 4:39:39 PM EDT> <Info> <Configuration Audit> <159902> <USER kernel
identity REMOVED mydomain:Name=MyServer-1,Type=Server>
```

WebLogic Server インスタンスが起動するたびに、新しい `DefaultAuditRecorder.log` ファイルが作成されます (古い `DefaultAuditRecorder.log` ファイルの名前は `DefaultAuditRecorder.log.old` に変更されます)。

監査情報を WebLogic 監査プロバイダで指定された以外のファイル、または `DefaultAuditRecorder.log` 以外の出力リポジトリ (異なる名前 / 場所のシンプル ファイルまたは既存のデータベース) に書き込む場合、カスタム監査プロバイダを開発する必要があります。

カスタム監査プロバイダの開発方法

WebLogic 監査プロバイダが開発者の要求を満たさない場合、次の手順でカスタム監査プロバイダを開発することができます。

1. 9-7 ページの「適切な SSPI を使用して実行時クラスを作成する」
2. 9-11 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 9-17 ページの「Administration Console を使用してカスタム監査プロバイダをコンフィグレーションする」

適切な SSPI を使用して実行時クラスを作成する

実行時クラスを作成する前に、以下の作業が必要です。

- 2-4 ページの「「Provider」SSPI の目的を理解する」
- 2-7 ページの「SSPI 階層を理解し、実行時クラスを 1 つ作成するのか 2 つ作成するのかを決定する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム監査プロバイダの実行時クラスを作成します。

- 9-8 ページの「AuditProvider SSPI を実装する」

- 9-8 ページの「AuditChannel SSPI を実装する」

カスタム監査プロバイダの実行時クラスの作成例については、9-9 ページの「例 : サンプル監査プロバイダの実行時クラスの作成」を参照してください。

AuditProvider SSPI を実装する

AuditProvider SSPI を実装するには、2-4 ページの「「Provider」SSPI の目的を理解する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

getAuditChannel

```
public AuditChannel getAuditChannel();
```

getAuditChannel メソッドは、AuditChannel SSPI の実装を取得します。MyAuditProviderImpl.java という 1 つの実行時クラスの場合、getAuditChannel メソッドの実装は次のようになります。

```
return this;
```

実行時クラスが 2 つの場合、getAuditChannel メソッドの実装は次のようになります。

```
return new MyAuditChannelImpl;
```

この理由は、AuditProvider SSPI を実装する実行時クラスが、AuditChannel SSPI を実装するクラスを取得するためのファクトリとして使用されるからです。

AuditProvider SSPI および getAuditChannel メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

AuditChannel SSPI を実装する

AuditChannel SSPI を実装する際には、以下のメソッドの実装を提供する必要があります。

writeEvent

```
public void writeEvent(AuditEvent event)
```

writeEvent メソッドは、渡された AuditEvent オブジェクト内に指定されている情報に基づいて監査記録を書き込みます。AuditEvent オブ

ジェクトの詳細については、10-4 ページの「監査イベントを作成する」を参照してください。

AuditChannel SSPI および writeEvent メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

例：サンプル監査プロバイダの実行時クラスの作成

コード リスト 9-2 は、サンプル監査プロバイダの実行時クラスである SampleAuditProviderImpl.java クラスを示しています。実行時クラスには以下の実装が含まれています。

- initialize、getDescription、および shutdown という SecurityProvider インタフェースから継承した 3 つのメソッド (2-4 ページの「「Provider」 SSPI の目的を理解する」 を参照)
- getAuditChannel という AuditProvider SSPI から継承したメソッド (9-8 ページの「AuditProvider SSPI を実装する」を参照)
- writeEvent という AuditChannel SSPI のメソッド (9-8 ページの「AuditChannel SSPI を実装する」を参照)

注意： コード リスト 9-2 の太字のコードは、クラス宣言とメソッド シグネチャを示しています。

コード リスト 9-2 SampleAuditProviderImpl.java

```
package examples.security.providers.audit;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import weblogic.management.security.ProviderMBean;
import weblogic.security.spi.AuditChannel;
import weblogic.security.spi.AuditEvent;
import weblogic.security.spi.AuditProvider;
import weblogic.security.spi.SecurityServices;

public final class SampleAuditProviderImpl implements AuditChannel, AuditProvider
{
```

```
   private String description;
   private PrintStream log;

   public void initialize(ProviderMBean mbean, SecurityServices services)
   {
      System.out.println("SampleAuditProviderImpl.initialize");

      description = mbean.getDescription() + "\n" + mbean.getVersion();

      SampleAuditorMBean myMBean = (SampleAuditorMBean)mbean;
      File file = new File(myMBean.getLogFileName());
      System.out.println("\tlogging to " + file.getAbsolutePath());

      try {
         log = new PrintStream(new FileOutputStream(file), true);
      } catch (IOException e) {
         throw new RuntimeException(e.toString());
      }
   }

   public String getDescription()
   {
      return description;
   }

   public void shutdown()
   {
      System.out.println("SampleAuditProviderImpl.shutdown");
      log.close();
   }

   public AuditChannel getAuditChannel()
   {
      return this;
   }

   public void writeEvent(AuditEvent event)
   {
      // イベントの toString メソッドを使用して
      // サンプル監査プロバイダのログ ファイルにイベントを書き込む
      log.println(event);
   }
}
```

WebLogic MBeanMaker を使用して MBean タイプを生成する

カスタム セキュリティ プロバイダの MBean タイプを生成する前に、以下の作業が必要です。

- 2-11 ページの「MBean タイプが必要な理由を理解する」
- 2-12 ページの「拡張および実装する SSPI MBean を決定する」
- 2-13 ページの「MBean 定義ファイル (MDF) の基本的な要素を理解する」
- 2-15 ページの「SSPI MBean の階層と Administration Console に対する影響を理解する」
- 2-17 ページの「WebLogic MBeanMaker によって提供されるものを理解する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム監査プロバイダの MBean タイプを作成します。

1. 9-11 ページの「MBean 定義ファイル (MDF) を作成する」
2. 9-12 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 9-15 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」
4. 9-16 ページの「WebLogic Server 環境に MBean タイプをインストールする」

注意： これらの手順の実行方法は、いくつかのサンプル セキュリティ プロバイダ (dev2dev Web サイトの「Code Samples: WebLogic Server」で入手可能) に示されています。

この節で説明する手順はすべて、Windows 環境での作業を想定しています。

MBean 定義ファイル (MDF) を作成する

MBean 定義ファイル (MDF) を作成するには、次の手順に従います。

1. サンプル監査プロバイダの MDF をテキスト ファイルにコピーします。

   注意： サンプル監査プロバイダの MDF は、`SampleAuditor.xml` という名前です。

2. カスタム監査プロバイダに合わせて、作成する MDF の `<MBeanType>` および `<MBeanAttribute>` 要素の内容を変更します。

3. カスタム属性および操作 (つまり、`<MBeanAttribute>` および `<MBeanOperation>` 要素) を MDF に追加します。

4. ファイルを保存します。

注意： MDF 要素の構文についての完全なリファレンスは、付録 A「MBean 定義ファイル (MDF) 要素の構文」に収められています。

WebLogic MBeanMaker を使用して MBean タイプを生成する

MDF を作成したら、WebLogic MBeanMaker を使用してそれを実行できます。WebLogic MBeanMaker は現在のところコマンドライン ユーティリティで、入力として MDF を受け取り、MBean インタフェース、MBean 実装、関連する MBean 情報ファイルなどの中間 Java ファイルをいくつか出力します。これらの中間ファイルが合わさって、カスタム セキュリティ プロバイダの **MBean タイプ**になります。

MBean タイプの生成手順は、カスタム監査プロバイダの設計に応じて異なります。必要な設計に合わせて適切な手順を実行してください。

- 9-12 ページの「カスタム操作を追加しない場合」
- 9-13 ページの「カスタム操作を追加する場合」

カスタム操作を追加しない場合

カスタム監査プロバイダの MDF にカスタム操作を含めない場合、次の手順に従います。

1. 新しい DOS シェルを作成します。

2. 次のコマンドを入力します。

```
java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
weblogic.management.commo.WebLogicMBeanMaker
```

ここで、-DMDF フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*xmlFile* は MDF (XML MBean 定義ファイル)、*filesdir* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

xmlfile が入力されるたびに、新しい出力ファイル群が生成されます。*filesdir* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

-DcreateStubs=true フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。したがって、MDF (つまり監査プロバイダ) が複数ある場合には、このプロセスを繰り返す必要があります。

3. 9-15 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

カスタム操作を追加する場合

カスタム監査プロバイダの MDF にカスタム操作を含める場合、質問に答えながら手順を進めてください。

- MBean タイプを作成するのは初めてですか。初めての場合は、次の手順に従ってください。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

```
java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
weblogic.management.commo.WebLogicMBeanMaker
```

ここで、-DMDF フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*xmlFile* は MDF (XML MBean 定義ファイル)、*filesdir* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

xmlfile が入力されるたびに、新しい出力ファイル群が生成されます。<filesdir> で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

`-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。したがって、MDF (つまり監査プロバイダ) が複数ある場合には、このプロセスを繰り返す必要があります。

3. MDF のすべてのカスタム操作に対して、メソッド スタブを使用してメソッドを実装します。
4. ファイルを保存します。
5. 9-15 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

■ 既存の MBean タイプの更新ですか。更新の場合は、次の手順に従ってください。

1. WebLogic MBeanMaker によって現在のメソッドの実装が上書きされないように、既存の MBean 実装ファイルを一時ディレクトリにコピーします。
2. 新しい DOS シェルを作成します。
3. 次のコマンドを入力します。

```
java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
weblogic.management.commo.WebLogicMBeanMaker
```

ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

`xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

`-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。したがって、MDF (つまり監査プロバイダ) が複数ある場合には、このプロセスを繰り返す必要があります。

4. MDF を変更して元の MDF にはないカスタム操作を含めた場合、メソッド スタブを使用してメソッドを実装します。

5. 完成した、つまりすべてのメソッドを実装した MBean 実装ファイルを保存します。
6. この MBean 実装ファイルを、WebLogic MBeanMaker が MBean タイプの実装ファイルを配置したディレクトリにコピーします。このディレクトリは、手順 3 で *filesdir* として指定しました (手順 3 の結果として WebLogic MBeanMaker で生成された MBean 実装ファイルがオーバーライドされる)。
7. 9-15 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

生成される MBean インタフェース ファイルについて

MBean インタフェース ファイルとは、実行時クラスまたは MBean 実装がコンフィグレーション データを取得するために使用する MBean のクライアントサイド API です。2-4 ページの「「Provider」SSPI の目的を理解する」で説明されているように、これは initialize メソッドで使用するのが一般的です。

WebLogic MBeanMaker では、作成済みの MDF から MBean タイプを生成するので、生成される MBean インタフェース ファイルの名前は、その MDF 名の後に「MBean」というテキストが付いたものになります。たとえば、WebLogic MBeanMaker で `SampleAuditor` MDF を実行すると、`SampleAuditorMBean.java` という MBean インタフェース ファイルが生成されます。

WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する

WebLogic MBeanMaker で MDF を実行して中間ファイルを作成し、MBean 実装ファイルを手作業で編集してその中にメソッドの内容を記述したら、カスタム監査プロバイダの MBean ファイルと実行時クラスを MBean JAR ファイル (MJF) にパッケージ化する必要があります。このプロセスも、WebLogic MBeanMaker によって自動化されます。

カスタム監査プロバイダの MJF を作成するには、次の手順に従います。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

```
java -DMJF=jarfile -Dfiles=filesdir
weblogic.management.commo.WebLogicMBeanMaker
```

ここで、-DMJF フラグは WebLogic MBeanMaker が新しい MBean タイプを格納する JAR ファイルをビルドする必要があることを示し、*jarfile* は MJF の名前、<filesdir> は WebLogic MBeanMaker で MJF に JAR 化する対象ファイルが存在する場所です。

この時点でコンパイルが行われるので、エラーが発生するおそれがあります。*jarfile* が指定されていて、エラーが発生しなかった場合には、指定された名前の MJF が作成されます。

注意： 既存の MJF を更新する場合は、MJF をいったん削除してから再生成します。WebLogic MBeanMaker にも -DIncludeSource オプションがあり、それを指定すると、生成される MJF にソース ファイルを含めるかどうかを制御することができます。ソース ファイルには、生成されたソースと MDF そのものがあります。デフォルトは false です。このオプションは、-DMJF を使用しない場合には無視されます。

生成された MJF は、自らの WebLogic Server 環境にインストールすることも、顧客に配布してそれぞれの WebLogic Server 環境にインストールしてもらうこともできます。

WebLogic Server 環境に MBean タイプをインストールする

MBean タイプを WebLogic Server 環境にインストールするには、MJF を *WL_HOME*\server\lib\mbeantypes ディレクトリにコピーします。ここで、*WL_HOME* は WebLogic Server の最上位のインストール ディレクトリです。このインストール コマンドによって、カスタム監査プロバイダが「デプロイ」されます。つまり、カスタム監査プロバイダを WebLogic Server Administration Console から管理できるようになります。

注意： *WL_HOME*\server\lib\mbeantypes は、MBean タイプをインストールするデフォルトのディレクトリです。ただし、WebLogic Server が追加ディレクトリで MBean タイプを検索するようにするには、サーバを起動するときに -Dweblogic.alternateTypesDirectory=<*dir*> コマンドライン フラグを使用します(<*dir*> はディレクトリ名のカンマ区切りのリスト)。このフラグを使用すると、WebLogic Server は常にまず *WL_HOME*\server\lib\mbeantypes から MBean タイプをロードし、その後に追加ディレクトリを検索して、(拡張子に関係なく) そのディレクト

リに存在するすべての有効なアーカイブをロードします。たとえば `-Dweblogic.alternateTypesDirectory = dirX,dirY` の場合、WebLogic Server はまず *`WL_HOME`*`\server\lib\mbeantypes` から MBean タイプをロードし、その後に、`dirX` および `dirY` に存在する有効なアーカイブをロードします。

追加ディレクトリで MBean タイプを検索するように WebLogic Server に指示し、かつ Java セキュリティ マネージャを使用する場合は、`weblogic.policy` ファイルを更新して、MBean タイプ (したがってカスタム セキュリティ プロバイダも) に対する適切なパーミッションを与える必要があります。詳細については、『WebLogic Security プログラマーズ ガイド』の「Java セキュリティ マネージャを使用しての WebLogic リソースの保護」を参照してください。

非セキュリティ プロバイダの JAR (バックアップ ファイルを含む) は、*`WL_HOME`*`\server\lib\mbeantypes` ディレクトリ以外の場所に置くことをお勧めします。

カスタム監査プロバイダをコンフィグレーションすることによって (9-17 ページの「Administration Console を使用してカスタム監査プロバイダをコンフィグレーションする」を参照) MBean タイプのインスタンスを作成して、GUI、他の Java コード、または API からそれらの MBean インスタンスを使用することができます。たとえば、WebLogic Server Administration Console を使用して、属性を取得 / 設定したり操作を呼び出したりすることもできますし、他の Java オブジェクトを開発して、そのオブジェクトで MBean をインスタンス化し、それらの MBean から提供される情報に自動的に応答させることもできます。なお、これらの MBean インスタンスをバックアップしておくことをお勧めします。詳細については、『WebLogic Server ドメイン管理』の「障害が発生したサーバの回復」の「セキュリティ コンフィグレーション データのバックアップ」を参照してください。

Administration Console を使用してカスタム監査プロバイダをコンフィグレーションする

カスタム監査プロバイダをコンフィグレーションするということは、監査サービスを必要とするセキュリティ プロバイダがアクセス可能なセキュリティ レルムにカスタム監査プロバイダを追加するということです。

カスタム セキュリティ プロバイダのコンフィグレーションは管理タスクですが、カスタム セキュリティ プロバイダの開発者が行うこともできます。この節では、カスタム監査プロバイダのコンフィグレーション担当者向けの重要な情報を取り上げます。

- 監査重大度のコンフィグレーション

注意： WebLogic Server Administration Console を使用したカスタム監査プロバイダのコンフィグレーション手順については、『WebLogic Security の管理』の「カスタム セキュリティ プロバイダのコンフィグレーション」を参照してください。

監査重大度のコンフィグレーション

コンフィグレーション手順では、監査プロバイダの監査重大度を下記の重大度レベルのいずれかに設定する必要があります。

- `INFORMATION`
- `WARNING`
- `ERROR`
- `SUCCESS`
- `FAILURE`

この重大度は、監査プロバイダが監査を開始するレベルを表します。

10 カスタム セキュリティ プロバイダからのイベントの監査

第 9 章「監査プロバイダ」で説明したように、**監査**とは、リクエストの操作とそれらのリクエストの結果に関する情報を、否認防止を目的として収集、格納、および配布するプロセスのことです。監査プロバイダは、コンピュータのアクティビティの電子的な記録を提供します。

各タイプのセキュリティ プロバイダは、セキュリティ関連イベントの実行前または実行後に、それらのイベントに関する情報を記録するようコンフィグレーション済みの監査プロバイダに要求できます。たとえば、あるユーザが預金口座アプリケーションの (アクセス権を持たない) `withdraw` メソッドにアクセスしようとした場合、認可プロバイダはこの操作を記録するよう要求できます。セキュリティ関連イベントは、監査プロバイダのコンフィグレーションで指定されている重大度レベルと一致するか、それを超えた場合にのみ記録されます。

以下の節では、カスタム セキュリティ プロバイダに監査機能を追加する前に理解しておく必要がある基本情報と、カスタム セキュリティ プロバイダに監査機能を追加する手順について説明します。

セキュリティ サービスと監査サービス

`weblogic.security.spi` パッケージの `SecurityServices` インタフェースは、セキュリティ サービスのリポジトリです (現在は単なる監査サービス)。`SecurityServices` インタフェースは、以下のメソッドを通じて呼び出し側に AuditorService の参照を提供します。

getAuditorService

```
public AuditorService getAuditorService
```

`getAuditorService` メソッドは、監査プロバイダがコンフィグレーションされている場合に `AuditService` を返します。

同じく `weblogic.security.spi` パッケージにある `AuditorService` インタフェースは、制限された (書き込み専用) 監査機能を持つ他のタイプのセキュリティ プロバイダ (認証プロバイダなど) を提供します。つまり、監査サービスはコンフィグレーション済みの各監査プロバイダの `writeEvent` メソッドの呼び出しを展開します。writeEvent メソッドは、渡された `AuditEvent` オブジェクト内に指定されている情報に基づいて監査記録を書き込みます。`writeEvent` メソッドの詳細については、9-8 ページの「AuditChannel SSPI を実装する」を参照してください。`AuditEvent` オブジェクトの詳細については、10-4 ページの「監査イベントを作成する」を参照してください。`AuditorService` インタフェースには以下のメソッドが定義されています。

providerAuditWriteEvent

```
public void providerAuditWriteEvent (AuditEvent event)
```

`providerAuditWriteEvent` メソッドは、セキュリティ プロバイダに対して、コンフィグレーション済みの監査プロバイダを呼び出す WebLogic Security フレームワーク内のオブジェクトへの書き込みアクセスを提供します。`event` パラメータは、監査するイベントのタイプや監査重大度レベルなどの監査条件を含む `AuditEvent` オブジェクトです。監査イベントと監査重大度レベルの詳細については、それぞれ 10-4 ページの「監査イベントを作成する」と 10-9 ページの「監査重大度」を参照してください。

監査サービスを呼び出すと、監査イベントの実行前または実行後にそれらのイベントを記録できますが、操作の前と後の間のコンテキストは維持されません。監査機能を持つセキュリティ プロバイダは、10-11 ページの「監査サービスを取得および使用して監査イベントを書き込む」の説明に従って監査サービスを取得する必要があります。

注意 : 監査プロバイダがコンフィグレーションされている場合、`SecurityServices` インタフェースと `AuditorService` インタフェースの実装は起動時に WebLogic Security フレームワークによって作成されます。監査プロバイダのコンフィグレーションについては、9-17 ページの

「Administration Console を使用してカスタム監査プロバイダをコンフィグレーションする」を参照してください。このため、開発者がこれらのインタフェースの実装を提供する必要はありません。

また、`SecurityServices` オブジェクトは、セキュリティ プロバイダがコンフィグレーションされているセキュリティ レルムに固有のものです。カスタム セキュリティ プロバイダの実行時クラスは、その `initialize` メソッドの一部としてレルム固有の `SecurityServices` オブジェクトの参照を自動的に取得します。詳細については、2-4 ページの「「Provider」 SSPI の目的を理解する」を参照してください。

これらのインタフェースとそのメソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc の SecurityServices インタフェースと AuditorService インタフェースを参照してください。

カスタム セキュリティ プロバイダからの監査方法

カスタム セキュリティ プロバイダに監査機能を追加するには、次の手順に従います。

- 10-4 ページの「監査イベントを作成する」
- 10-11 ページの「監査サービスを取得および使用して監査イベントを書き込む」

各手順の例については、10-10 ページの「例 : AuditRoleEvent インタフェースの実装」と 10-12 ページの「例 : 監査サービスを取得および使用してロール監査イベントを書き込む」を参照してください。

注意 : カスタム セキュリティ プロバイダが監査イベントを記録する場合、必須のファイルに加えて、以下の手順で作成したすべてのクラスを、そのプロバイダ用の MBean JAR ファイル (MJF) に追加する必要があります。

監査イベントを作成する

セキュリティ プロバイダは、イベントのタイプ (認証イベントなど) や監査重大度 (「エラー」など) といった、監査するイベントに関する情報を提供する必要があります。**監査イベント**には、この情報が格納されます。また、コンフィグレーション済みの監査プロバイダが理解できるコンテキスト データも格納できます。監査イベントを作成するには、以下のいずれかを行います。

- 10-4 ページの「AuditEvent SSPI を実装する」または
- 10-5 ページの「監査イベント コンビニエンス インタフェースを実装する」

AuditEvent SSPI を実装する

`AuditEvent` SSPI を実装するには、以下のメソッドの実装を提供する必要があります。

getEventType

```
public java.lang.String getEventType()
```

`getEventType` メソッドは、監査するイベント タイプの文字列表現を返します。この文字列表現は、監査チャネル (`AuditChannel` SSPI を実装する実行時クラス) によって使用されます。たとえば、BEA 提供の実装のイベント タイプは `"Authentication Audit Event"` です。詳細については、9-1 ページの「監査チャネル」と 9-8 ページの「AuditChannel SSPI を実装する」を参照してください。

getFailureException

```
public java.lang.Exception getFailureException()
```

`getFailureException` メソッドは、`Exception` オブジェクトを返します。監査チャネルは、toString メソッドから提供される情報に加えて、このオブジェクトから監査情報を取得します。

getSeverity

```
public AuditSeverity getSeverity()
```

`getSeverity` メソッドは、監査するイベント タイプに関連付けられている重大度レベル値を返します。この値は、監査チャネルによって使用されます。監査チャネルはこの値を使用して、監査するかどうかを判定

します。詳細については、10-9 ページの「監査重大度」を参照してください。

toString

```
public java.lang.String toString()
```

`toString` メソッドは、プリフォーマットされた監査情報を監査チャネルに返します。

`AuditEvent` SSPI および上記のメソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

監査イベント コンビニエンス インタフェースを実装する

`AuditEvent` SSPI には、ユーザの利便を図り、監査イベントの作成に役立つサブインタフェースがいくつか用意されています。

監査チャネル (`AuditChannel` SSPI を実装する実行時クラス) では、それらの各監査イベント コンビニエンス インタフェースを使用して、拡張イベント タイプ オブジェクトのインスタンス タイプをより効果的に決定し、特定のタイプのセキュリティ プロバイダを識別することができます。たとえば、`AuditAtnEvent` コンビニエンス インタフェースは、拡張認証イベント タイプ オブジェクトのインスタンス タイプを決定する監査チャネルで使用できます。詳細については 9-1 ページの「監査チャネル」と 9-8 ページの「AuditChannel SSPI を実装する」を参照してください。

監査イベント コンビニエンス インタフェースは以下のとおりです。

- 10-6 ページの「AuditAtnEvent インタフェース」
- 10-7 ページの「AuditAtzEvent および AuditPolicyEvent インタフェース」
- 10-8 ページの「AuditMgmtEvent インタフェース」
- 10-8 ページの「AuditRoleEvent および AuditRoleDeploymentEvent インタフェース」

注意 : 上記の監査イベント コンビニエンス インタフェースの 1 つを実装することをお勧めしますが、これは必須ではありません。

AuditAtnEvent インタフェース

AuditAtnEvent は、監査チャネルが拡張認証イベント タイプ オブジェクトのインスタンス タイプを決定するためのコンビニエンス インタフェースです。

AuditAtnEvent インタフェースを実装するには、10-4 ページの「AuditEvent SSPI を実装する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

getUsername

```
public String getUsername()
```

getUsername メソッドは、認証イベントに関連付けられているユーザ名を返します。

AtnEventType

```
public AtnEventType getAtnEventType()
```

AtnEventType メソッドは、認証イベントをより具体的に表すイベント タイプを返します。具体的な認証イベント タイプは以下のとおりです。

AUTHENTICATE — ユーザ名とパスワードを使用する単純な認証が発生した

ASSERTIDENTITY — トークンに基づく境界認証が発生した

IMPERSONATEIDENTITY — 指定されたクライアント ユーザ名を使用してクライアント ID が確立された (kernal ID が必要)

VALIDATEIDENTITY — 指定されたサブジェクト内のプリンシパルの信頼性が検証された

USERLOCKED — 無効なログインの試行によってユーザ アカウントがロックされた

USERUNLOCKED — ユーザ アカウントのロックが解除された

USERLOCKOUTEXPIRED — ユーザ アカウントのロックが期限切れになった

toString

```
public String toString()
```

toString メソッドは、文字列で表された特定の認証情報を監査に返します。

注意： AuditAtnEvent コンビニエンス インタフェースは、AuditEvent と AuditContext の両方のインタフェースを拡張したものです。AuditContext インタフェースの詳細については、10-9 ページの「監査コンテキスト」を参照してください。

AuditAtnEvent コンビニエンス インタフェースおよび上記のメソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

AuditAtzEvent および AuditPolicyEvent インタフェース

AuditAtzEvent および AuditPolicyEvent コンビニエンス インタフェースは、監査チャネルが拡張認可イベント タイプ オブジェクトのインスタンス タイプを決定するのに役立ちます。

注意： AuditPolicyEvent コンビニエンス インタフェースは、AuditAtzEvent コンビニエンス インタフェースと同じですが、AuditEvent インタフェースのみを拡張する点が異なります。AuditContext インタフェースは拡張しません。AuditContext インタフェースの詳細については、10-9 ページの「監査コンテキスト」を参照してください。

AuditAtzEvent または AuditPolicyEvent インタフェースを実装するには、10-4 ページの「AuditEvent SSPI を実装する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

getSubject

```
public Subject getSubject()
```

getSubject メソッドは、認可イベントに関連付けられているサブジェクト (WebLogic リソースにアクセスしようとしているサブジェクト) を返します。

getResource

```
public Resource getResource()
```

getResource メソッドは、認可イベントに関連付けられ、サブジェクトがアクセスしようとしている WebLogic リソースを返します。

これらのコンビニエンス インタフェースとそのメソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc で AuditAtzEvent インタフェースまたは AuditPolicyEvent インタフェースを参照してください。

AuditMgmtEvent インタフェース

`AuditMgmtEvent` は、監査チャネルが拡張セキュリティ管理イベント タイプ オブジェクトのインスタンス タイプ (セキュリティ プロバイダの MBean など) を決定するためのコンビニエンス インタフェースです。このインタフェースには実装する必要があるメソッドは含まれていませんが、監査イベント実装のベストプラクティス構造が保持されています。

注意： MBean の詳細については、2-11 ページの「セキュリティ サービス プロバイダ インタフェース (SSPI) MBean」を参照してください。

`AuditMgmtEvent` コンビニエンス インタフェースの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

AuditRoleEvent および AuditRoleDeploymentEvent インタフェース

`AuditRoleDeploymentEvent` および `AuditRoleEvent` コンビニエンス インタフェースは、監査チャネルが拡張ロール マッピング イベント タイプ オブジェクトのインスタンス タイプを決定するのに役立ちます。このインタフェースには実装する必要があるメソッドは含まれていませんが、監査イベント実装のベストプラクティス構造が保持されています。

注意： `AuditRoleDeploymentEvent` コンビニエンス インタフェースは `AuditRoleEvent` コンビニエンス インタフェースと同じですが、`AuditEvent` インタフェースのみを拡張する点が異なります。`AuditContext` インタフェースは拡張しません。`AuditContext` インタフェースの詳細については、10-9 ページの「監査コンテキスト」を参照してください。

これらのコンビニエンス インタフェースの詳細については、WebLogic Server 7.0 API リファレンス Javadoc の AuditRoleEvent インタフェースまたは AuditRoleDeploymentEvent インタフェースを参照してください。

監査重大度

監査重大度とは、セキュリティ プロバイダが監査イベントを記録するレベルのことです。コンフィグレーション済みの監査プロバイダが監査リクエストを受け取ると、各監査プロバイダは実行されるイベントの重大度を調べます。イベントの重大度が監査プロバイダのコンフィグレーションに使用されたレベル以上であれば、監査プロバイダはその監査データを記録します。

注意： 監査プロバイダのコンフィグレーションは、WebLogic Server Administration Console で行います。詳細については、9-17 ページの「Administration Console を使用してカスタム監査プロバイダをコンフィグレーションする」を参照してください。

`weblogic.security.spi` パッケージに含まれる `AuditSeverity` クラスは、監査重大度レベルを数値およびテキスト値として `AuditEvent` オブジェクトを介して監査チャネル (`AuditChannel` SSPI の実装) に提供します。重大度の数値はロジックで用いるためのもので、テキスト値は出力用監査記録の作成で用いるためのものです。`AuditChannel` SSPI と `AuditEvent` オブジェクトの詳細については、9-8 ページの「AuditChannel SSPI を実装する」と 10-4 ページの「監査イベントを作成する」をそれぞれ参照してください。

監査コンテキスト

一部の監査イベント コンビニエンス インタフェースは、実装にコンテキスト情報も含まれていることを示すために、`AuditContext` インタフェースを拡張します。コンテキスト情報は、監査チャネルによって使用されます。詳細については、9-1 ページの「監査チャネル」と 9-8 ページの「AuditChannel SSPI を実装する」を参照してください。

`AuditContext` インタフェースには以下のメソッドが定義されています。

getContext

```
public ContextHandler getContext()
```

`getContext` メソッドは、`ContextHandler` オブジェクトを返します。実行時クラス (`AuditChannel` SSPI の実装) は、このオブジェクトから追加の監査情報を取得します。ContextHandler の詳細については、2-35 ページの「ContextHandler と WebLogic リソース」を参照してください。

例 : AuditRoleEvent インタフェースの実装

コード リスト 10-1 に、MyAuditRoleEventImpl.java クラスを示します。これは、監査イベント コンビニエンス インタフェース (ここでは AuditRoleEvent コンビニエンス インタフェース) の実装例です。このクラスには以下の実装が含まれています。

- getEventType、getFailureException、getSeverity、および toString という AuditEvent SSPI から継承された 4 つのメソッド (10-4 ページの「AuditEvent SSPI を実装する」を参照)
- ContextHandler を介して追加のコンテキスト情報を返す getContext という 1 つの追加メソッド (ContextHandler の詳細については、2-35 ページの「ContextHandler と WebLogic リソース」を参照)

注意 : コード リスト 10-1 の太字のコードは、クラス宣言とメソッド シグネチャを示しています。

コード リスト 10-1　MyAuditRoleEventImpl.java

```
package mypackage;

import javax.security.auth.Subject;
import weblogic.security.SubjectUtils;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.AuditRoleEvent;
import weblogic.security.spi.AuditSeverity;
import weblogic.security.spi.Resource;

/*package*/ class MyAuditRoleEventImpl implements AuditRoleEvent
{
   private Subject subject;
   private Resource resource;
   private ContextHandler context;
   private String details;
   private Exception failureException;

   /*package*/ MyAuditRoleEventImpl(Subject subject, Resource
resource,
      ContextHandler context, String details, Exception
      failureException) {

         this.subject = subject;
         this.resource = resource;
         this.context = context;
         this.details = details;
```

```
            this.failureException = failureException;
        }

        public Exception getFailureException()
        {
            return failureException;
        }

        public AuditSeverity getSeverity()
        {
            return (failureException == null) ? AuditSeverity.SUCCESS :
               AuditSeverity.FAILURE;
        }

        public String getEventType()
        {
            return "MyAuditRoleEventType";
        }

        public ContextHandler getContext()
        {
            return context;
        }

        public String toString()
        {
            StringBuffer buf = new StringBuffer();
            buf.append("EventType:" + getEventType() + "\n");
            buf.append("\tSeverity: " +
               getSeverity().getSeverityString());
            buf.append("\tSubject: " +
               SubjectUtils.displaySubject(getSubject());
            buf.append("\tResource: " + resource.toString());
            buf.append("\tDetails: " + details);

            if (getFailureException() != null) {
               buf.append("\n\tFailureException:" +
                  getFailureException());
            }

            return buf.toString();
        }
    }
```

監査サービスを取得および使用して監査イベントを書き込む

カスタム セキュリティ プロバイダから監査サービスを取得および使用して監査イベントを書き込むには、次の手順に従います。

1. getAuditorService メソッドを使用して監査サービスを返します。

   注意： SecurityServices オブジェクトは initialize メソッドの一部としてセキュリティ プロバイダの「Provider」SSPI の実装に渡されることに注意してください。詳細については、2-4 ページの「「Provider」SSPI の目的を理解する」を参照してください。AuditorService オブジェクトは、監査プロバイダがコンフィグレーションされている場合にのみ返されます。

2. 10-4 ページの「AuditEvent SSPI を実装する」で作成した監査イベントをインスタンス化し、AuditService.providerAuditWriteEvent メソッドを介して監査サービスに送ります。

例：監査サービスを取得および使用してロール監査イベントを書き込む

コード リスト 10-2 に、カスタム ロール マッピング プロバイダの実行時クラス (MyRoleMapperProviderImpl.java) が監査サービスを取得し、そのサービスを使用して監査イベントを書き込む例を示します。

注意： MyRoleMapperProviderImpl.java クラスは、コード リスト 10-1 の MyAuditRoleEventImpl.java クラスに依存します。

コード リスト 10-2　MyRoleMapperProviderImpl.java

```
package mypackage;

import javax.security.auth.Subject;
import weblogic.management.security.ProviderMBean;
import weblogic.security.SubjectUtils;
import weblogic.security.service.ContextHandler;
import weblogic.security.spi.AuditorService;
import weblogic.security.spi.RoleMapper;
import weblogic.security.spi.RoleProvider;
import weblogic.security.spi.Resource;
import weblogic.security.spi.SecurityServices;

public final class MyRoleMapperProviderImpl implements
RoleProvider, RoleMapper
{
   private AuditorService auditor;
```

```
   public void initialize(ProviderMBean mbean, SecurityServices
      services)
   {
     auditor = services.getAuditorService();
      ...
   }

   public Map getRoles(Subject subject, Resource resource,
      ContextHandler handler)
   {
      ...
      if (auditor != null)
      {
        auditor.providerAuditWriteEvent(
             new MyRoleEventImpl(subject, resource, context,
             "why logging this event",
             null);                 // 例外が発生しなかった
      }
      ...
   }

}
```

注意： コード リスト 10-2 のコードは、セキュリティ プロバイダの実行時クラスから監査イベントをポストする方法の例です。管理メソッドから監査イベントをポストすることもできます。管理メソッドから監査イベントをポストする例については、dev2dev Web サイトの「Code Samples: WebLogic Server」から入手可能なサンプル セキュリティ プロバイダの 1 つである Manageable Sample Authentication Provider を参照してください。

11　資格マッピング プロバイダ

資格マッピングとは、対象リソースにアクセスするユーザを認証するための適切な資格群を、レガシー システムの認証メカニズムを用いて取得するプロセスのことです。WebLogic Server では、資格マッピング プロバイダを使用して資格マッピング サービスを提供し、新しいタイプの資格を WebLogic Server 環境に導入します。

以下の節では、資格マッピング プロバイダの概念と機能、およびカスタム資格マッピング プロバイダの開発手順について説明します。

資格マッピングの概念

サブジェクト、つまり WebLogic リソース リクエストの発信元は、**資格**というセキュリティ関連の属性を持っています。資格には、新しいサービスにアクセスするサブジェクトを認証するための情報が含まれています。こうした資格には、ユーザ名 / パスワードの組み合わせ、Kerberos チケット、および公開鍵証明書などがあります。また、サブジェクトが特定のアクティビティを実行することを許可するデータも資格に含まれています。たとえば、暗号鍵が表す資格を持っていれば、サブジェクトは署名したり、データを暗号化したりすることができます。

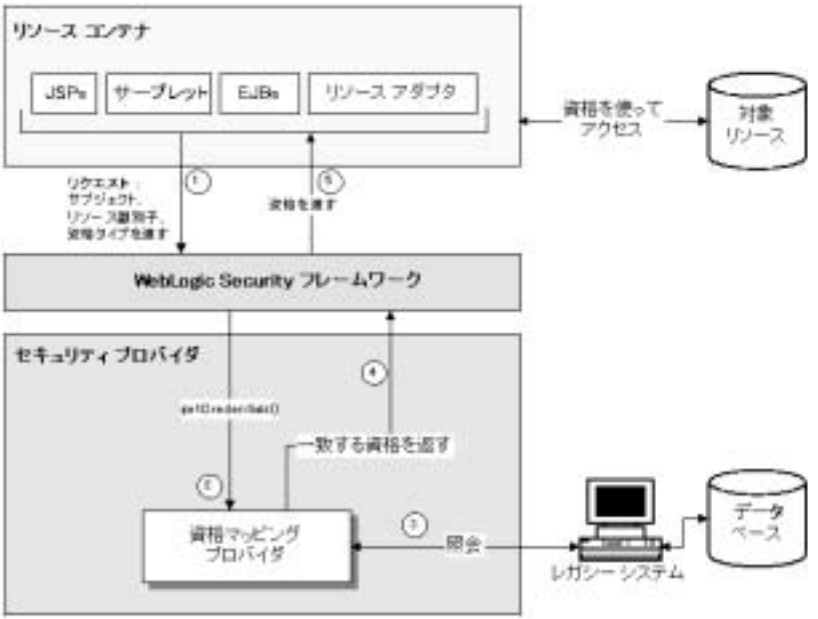
資格マップは、WebLogic Server で使用される資格と、レガシー システム (またはリモート システム) で使用される資格とのマッピングであり、そのシステムの特定のリソースへの接続方法を WebLogic Server に指示します。つまり、資格

マップを使用することで、WebLogic Server は認証済みサブジェクトに代わってリモート システムにログインできます。資格マッピング プロバイダを開発すると、資格をこのようにマップできます。

資格マッピング プロセス

資格マッピング プロバイダと WebLogic Security フレームワークとの対話を図 11-1 に示し、その後それについて説明します。

図 11-1　資格マッピング プロバイダと資格マッピング プロセス



一般に、資格マッピングは以下のように実行されます。

1. JavaServer Page (JSP)、サーブレット、エンタープライズ JavaBean (EJB)、リソース アダプタなどのアプリケーション コンポーネントは、適切なリソース コンテナを通じて WebLogic Security フレームワークを呼び出します。呼び

出しの中で、アプリケーション コンポーネントは、サブジェクト (「誰が」要求しているのか)、WebLogic リソース (「何を」要求しているのか)、および WebLogic リソースにアクセスするために必要な資格のタイプについての情報を渡します。

2. WebLogic Security フレームワークは、資格に対するアプリケーション コンポーネントのリクエストを、アプリケーション コンポーネントが必要としている資格のタイプを処理するコンフィグレーション済み資格マッピング プロバイダに送信します。
3. 資格マッピング プロバイダは、レガシー システムのデータベースを照会して、アプリケーション コンポーネントが要求しているものに一致する資格群を取得します。
4. 資格マッピング プロバイダは、資格を WebLogic Security フレームワークに返します。
5. WebLogic Security フレームワークは、リソース コンテナを通じて要求側のアプリケーション コンポーネントに資格を返します。

   アプリケーション コンポーネントは、資格を使用して外部システムにアクセスします。外部システムには、Oracle や SQL Server などのデータベース リソースがあります。

カスタム資格マッピング プロバイダを開発する必要があるか

WebLogic Server のデフォルト (つまりアクティブな) セキュリティ レルムには WebLogic 資格マッピング プロバイダが含まれています。WebLogic 資格マッピング プロバイダは、WebLogic Server のユーザおよびグループを、他の外部システムが必要とする適切なユーザ名 / パスワード資格にマップします。必要な資格マッピングが WebLogic Server のユーザおよびグループと他のシステムのユーザ名 / パスワードとの間のマッピングであれば、WebLogic 資格マッピング プロバイダで十分です。しかし、WebLogic Server のユーザおよびグループを他のタイプの資格 (Kerberos チケットなど) にマップしたい場合は、カスタム資格マッピング プロバイダを開発する必要があります。

カスタム資格マッピング プロバイダの開発方法

WebLogic 資格マッピング プロバイダが開発者の要求を満たさない場合、次の手順でカスタム資格マッピング プロバイダを開発することができます。

1. 11-4 ページの「適切な SSPI を使用して実行時クラスを作成する」
2. 11-8 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 11-16 ページの「Administration Console を使用してカスタム資格マッピング プロバイダをコンフィグレーションする」
4. 11-20 ページの「資格マップを管理するためのメカニズムを提供する」

適切な SSPI を使用して実行時クラスを作成する

実行時クラスを作成する前に、以下の作業が必要です。

- 2-4 ページの「「Provider」 SSPI の目的を理解する」
- 2-5 ページの「実装する「Provider」 インタフェースを決定する」
- 2-7 ページの「SSPI 階層を理解し、実行時クラスを 1 つ作成するのか 2 つ作成するのかを決定する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム資格マッピング プロバイダの実行時クラスを作成します。

- 11-5 ページの「CredentialProvider SSPI を実装する」または 11-5 ページの「DeployableCredentialProvider SSPI を実装する」
- 11-6 ページの「CredentialMapper SSPI を実装する」

注意： セキュリティ レルムでは、少なくとも 1 つの資格マッピング プロバイダが `DeployableCredentialProvider` SSPI を実装する必要があり、そうでなければリソース アダプタをデプロイすることが不可能になります。

CredentialProvider SSPI を実装する

CredentialProvider SSPI を実装するには、2-4 ページの「「Provider」 SSPI の目的を理解する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

getCredentialProvider

```
public CredentialMapper getCredentialProvider();
```

getCredentialProvider メソッドは、CredentialMapper SSPI の実装を取得します。MyCredentialMapperProviderImpl.java という 1 つの実行時クラスの場合 (図 2-3 を参照)、getCredentialProvider メソッドの実装は次のようになります。

```
return this;
```

実行時クラスが 2 つの場合、getCredentialProvider メソッドの実装は次のようになります。

```
return new MyCredentialMapperImpl;
```

この理由は、CredentialProvider SSPI を実装する実行時クラスが、CredentialMapper SSPI を実装するクラスを取得するためのファクトリとして使用されるからです。

CredentialProvider SSPI および getCredentialProvider メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

DeployableCredentialProvider SSPI を実装する

DeployableCredentialProvider SSPI を実装するには、2-4 ページの「「Provider」 SSPI の目的を理解する」および 11-5 ページの「CredentialProvider SSPI を実装する」で説明されているメソッドと以下のメソッドの実装を提供する必要があります。

deployCredentialMapping

```
public void deployCredentialMapping(Resource resource,
String initiatingPrincipal, String eisUsername, String
eisPassword)throws ResourceCreationException;
```

deployCredentialMapping メソッドは、資格マップをデプロイします (つまり、デプロイ済みのリソース アダプタに代わって資格マッピング

をデータベースに作成します)。マッピングが既に存在する場合、マッピングは削除され、このマッピングによって置き換えられます。`resource` パラメータは、`String` で表される開始元プリンシパルがアクセスを要求している WebLogic リソースを表します。エンタープライズ情報システム (EIS) のユーザ名およびパスワードは、資格のマップ先のレガシー システム (リモート システム) における資格です。

undeployCredentialMappings

```
public void undeployCredentialMappings(Resource resource)
throws ResourceRemovalException;
```

`undeployCredentialMappings` メソッドは、資格マップをアンデプロイします (つまり、アンデプロイ済みのリソース アダプタに代わって資格マッピングをデータベースから削除します)。`resource` パラメータは、マッピングを削除する WebLogic リソースを表します。

注意： `deployCredentialMapping/undeployCredentialMappings` メソッドは、ユーザ名 / パスワード資格のみを操作します。

`DeployableCredentialProvider` SSPI および `deployCredentialMapping/undeployCredentialMappings` メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

CredentialMapper SSPI を実装する

`CredentialMapper` SSPI を実装するには、以下のメソッドの実装を提供する必要があります。

getCredentials

```
public java.util.Vector getCredentials(Subject requestor,
Subject initiator, Resource resource, String[]
credentialTypes);
```

`getCredentials` メソッドは、サブジェクトの ID に基づいて、対象リソースの適切な資格群を取得します。このバージョンのメソッドは、リモート システムのデータベースに照会して、サブジェクト内のすべてのプリンシパルに関して一致する資格のリストを (ベクトルとして) 返します。

getCredentials

```
public java.lang.Object getCredentials(Subject requestor,
String initiator, Resource resource, String[]
credentialTypes);
```

`getCredentials` メソッドは、サブジェクトの ID に基づいて、対象リソースの適切な資格群を取得します。このバージョンのメソッドは、リモート システムのデータベースに照会して、指定されたサブジェクトの資格を 1 つ (オブジェクトとして) 返します。

`CredentialMapper` SSPI および `getCredentials` メソッドの詳細については、WebLogic Server 7.0 API リファレンス Javadoc を参照してください。

レルム アダプタ認証プロバイダと互換性のあるカスタム資格マッピング プロバイダの開発

認証プロバイダは、サブジェクト内へのユーザおよびグループの格納を担当するセキュリティ プロバイダです。ユーザおよびグループは、資格マッピング プロバイダなどの他のタイプのセキュリティ プロバイダによってサブジェクトから抽出されます。セキュリティ レルムでコンフィグレーションされている認証プロバイダがレルム アダプタ認証プロバイダである場合、ユーザおよびグループの情報は、他の認証プロバイダとは若干異なる方法でサブジェクトに格納されます。そのため、こうしたユーザおよびグループの情報の抽出も、若干異なる方法で行う必要があります。

サブジェクトへの格納にレルム アダプタ認証プロバイダが使用された場合に、サブジェクトがユーザ名またはグループ名に一致するかどうかを調べるためにカスタム資格マッピング プロバイダで使用できるコードをコード リスト 11-1 に示します。このコードは、実装するどの形式の `getCredentials` メソッドにも記述できます。

コード リスト 11-1 サブジェクトがユーザ名またはグループ名に一致するかどうかを調べるためのサンプル コード

```
/**
 * サブジェクトがユーザ名またはグループ名に一致するかどうかを調べる
 *
 * @param principalWant このロールのプリンシパル名を含む文字列
 * ( つまり、ロール定義 )
 *
```

```
 * @param subject リソースにアクセスしようとしているユーザおよびそのユーザのグループを
 * 識別するプリンシパルを含むサブジェクト
 *
 * @return ブール値。現在のサブジェクトがロールのプリンシパルに
 * 一致する場合は true、それ以外の場合は false
 */
private boolean subjectMatches(String principalWant, Subject subject)
{
   // まず、グループ名が一致しているかどうかを調べる
   if (SubjectUtils.isUserInGroup(subject, principalWant)) {
      return true;
   }
   // 次に、ユーザ名が一致しているかどうかを調べる
   if (principalWant.equals(SubjectUtils.getUsername(subject))) {
      return true;
   }
   // 一致しなかった場合
   return false;
}
```

WebLogic MBeanMaker を使用して MBean タイプを生成する

カスタム セキュリティ プロバイダの MBean タイプを生成する前に、以下の作業が必要です。

- 2-11 ページの「MBean タイプが必要な理由を理解する」
- 2-12 ページの「拡張および実装する SSPI MBean を決定する」
- 2-13 ページの「MBean 定義ファイル (MDF) の基本的な要素を理解する」
- 2-15 ページの「SSPI MBean の階層と Administration Console に対する影響を理解する」
- 2-17 ページの「WebLogic MBeanMaker によって提供されるものを理解する」

この情報を理解し、設計に関する判断を下したら、次の手順でカスタム資格マッピング プロバイダの MBean タイプを作成します。

1. 11-9 ページの「MBean 定義ファイル (MDF) を作成する」
2. 11-10 ページの「WebLogic MBeanMaker を使用して MBean タイプを生成する」
3. 11-14 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」
4. 11-15 ページの「WebLogic Server 環境に MBean タイプをインストールする」

注意： これらの手順の実行方法は、いくつかのサンプル セキュリティ プロバイダ (dev2dev Web サイトの「Code Samples: WebLogic Server」で入手可能) に示されています。

この節で説明する手順はすべて、Windows 環境での作業を想定しています。

MBean 定義ファイル (MDF) を作成する

MBean 定義ファイル (MDF) を作成するには、次の手順に従います。

1. サンプル認証プロバイダの MDF をテキスト ファイルにコピーします。

   注意： サンプル認証プロバイダの MDF は、`SampleAuthenticator.xml` という名前です (現在、サンプル資格マッピング プロバイダはありません)。

2. カスタム資格マッピング プロバイダに合わせて、作成する MDF の `<MBeanType>` および `<MBeanAttribute>` 要素の内容を変更します。
3. カスタム属性および操作 (つまり、`<MBeanAttribute>` および `<MBeanOperation>` 要素) を MDF に追加します。
4. ファイルを保存します。

注意： MDF 要素の構文についての完全なリファレンスは、付録 A「MBean 定義ファイル (MDF) 要素の構文」に収められています。

WebLogic MBeanMaker を使用して MBean タイプを生成する

MDF を作成したら、WebLogic MBeanMaker を使用してそれを実行できます。WebLogic MBeanMaker は現在のところコマンドライン ユーティリティで、入力として MDF を受け取り、MBean インタフェース、MBean 実装、関連する MBean 情報ファイルなどの中間 Java ファイルをいくつか出力します。これらの中間ファイルが合わさって、カスタム セキュリティ プロバイダの **MBean タイプ**になります。

MBean タイプの生成手順は、カスタム資格マッピング プロバイダの設計に応じて異なります。必要な設計に合わせて適切な手順を実行してください。

- 11-10 ページの「任意 SSPI MBean とカスタム操作を追加しない場合」
- 11-11 ページの「任意 SSPI MBean またはカスタム操作を追加する場合」

任意 SSPI MBean とカスタム操作を追加しない場合

カスタム資格マッピング プロバイダの MDF に任意 SSPI MBean もカスタム操作も実装しない場合、次の手順に従います。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

   ```
   java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

   `xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

   `-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり資格マッピング プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

3. 11-14 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

任意 SSPI MBean またはカスタム操作を追加する場合

カスタム資格マッピング プロバイダの MDF に任意 SSPI MBean またはカスタム操作を実装する場合、以下の質問に答えながら手順を進めてください。

- MBean タイプを作成するのは初めてですか。初めての場合は、次の手順に従ってください。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

```
java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
weblogic.management.commo.WebLogicMBeanMaker
```

ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

`xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

`-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意： WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり資格マッピング プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

3. 任意 SSPI MBean を MDF に実装した場合は、次の手順に従います。

a. MBean 実装ファイルを見つけます。

WebLogic MBeanMaker によって生成される MBean 実装ファイルには、*`MBeanName`*`Impl.java` という名前が付けられます。たとえば、

`MyCredentialMapper` という MDF の場合には、編集すべき MBean 実装ファイルの名前は `SampleCredentialMapperImpl.java` です。

b. MDF に実装した任意 SSPI MBean ごとに、メソッドのスタブを「Mapping MDF Operation Declarations to Java Method Signatures Document」(dev2dev Web サイトで入手可能) から MBean 実装ファイルにコピーし、各メソッドを実装します。任意 SSPI MBean が継承するメソッドもすべて実装してください。

4. MDF にカスタム操作を含めた場合、メソッド スタブを使用してメソッドを実装します。
5. ファイルを保存します。
6. 11-14 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

■ 既存の MBean タイプの更新ですか。更新の場合は、次の手順に従ってください。

1. WebLogic MBeanMaker によって現在のメソッドの実装が上書きされないように、既存の MBean 実装ファイルを一時ディレクトリにコピーします。
2. 新しい DOS シェルを作成します。
3. 次のコマンドを入力します。

```
java -DMDF=xmlfile -Dfiles=filesdir -DcreateStubs=true
weblogic.management.commo.WebLogicMBeanMaker
```

ここで、`-DMDF` フラグは WebLogic MBeanMaker が MDF をコードに翻訳する必要があることを示し、*`xmlFile`* は MDF (XML MBean 定義ファイル)、*`filesdir`* は WebLogic MBeanMaker で生成された MBean タイプの中間ファイルが格納される場所です。

`xmlfile` が入力されるたびに、新しい出力ファイル群が生成されます。*`filesdir`* で指定された場所にファイルが既に存在する場合には、既存のファイルが上書きされることが通知され確認を求められます。

`-DcreateStubs=true` フラグを使用するたびに、既存の MBean 実装ファイルがすべて上書きされます。

注意 : WebLogic MBeanMaker では MDF を一度に 1 つ処理します。そのため、MDF が複数ある (つまり資格マッピング プロバイダが複数ある) 場合には、このプロセスを繰り返す必要があります。

4. 任意 SSPI MBean を MDF に実装した場合は、次の手順に従います。

   a. MBean 実装ファイルを見つけます。

      WebLogic MBeanMaker によって生成される MBean 実装ファイルには、*MBeanName*Impl.java という名前が付けられます。たとえば、SampleCredentialMapper という MDF の場合には、編集すべき MBean 実装ファイルの名前は SampleCredentialMapperImpl.java です。

   b. 手順 1 で一時ディレクトリに保存した既存の MBean 実装ファイルを開きます。

   c. 既存の MBean 実装ファイルを、WebLogic MBeanMaker によって生成された MBean 実装ファイルと同期させます。

      これには、メソッドの実装を既存の MBean 実装ファイルから新しく生成された MBean 実装ファイルにコピー (または、新しく生成された MBean 実装ファイルから既存の MBean 実装ファイルに新しいメソッドを追加) し、いずれの MBean 実装ファイルにも入っているメソッドのメソッド シグネチャへの変更が使用する MBean 実装ファイルに反映されていることを確認するといった作業が必要です。

   d. MDF を修正して元の MDF にはない任意 SSPI MBean を実装した場合は、メソッド スタブを「Mapping MDF Operation Declarations to Java Method Signatures Document」(dev2dev Web サイトで入手可能) から MBean 実装ファイルにコピーし、各メソッドを実装します。任意 SSPI MBean が継承するメソッドもすべて実装してください。

5. MDF を変更して元の MDF にはないカスタム操作を含めた場合、メソッド スタブを使用してメソッドを実装します。

6. 完成した、つまりすべてのメソッドを実装した MBean 実装ファイルを保存します。

7. この MBean 実装ファイルを、WebLogic MBeanMaker が MBean タイプの実装ファイルを配置したディレクトリにコピーします。このディレクトリは、手順 3 で *filesdir* として指定しました (手順 3 の結果として WebLogic MBeanMaker で生成された MBean 実装ファイルがオーバーライドされる)。

8. 11-14 ページの「WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する」に進みます。

生成される MBean インタフェース ファイルについて

MBean インタフェース ファイルとは、実行時クラスまたは MBean 実装がコンフィグレーション データを取得するために使用する MBean のクライアントサイド API です。2-4 ページの「「Provider」 SSPI の目的を理解する」で説明されているように、これは initialize メソッドで使用するのが一般的です。

WebLogic MBeanMaker では、作成済みの MDF から MBean タイプを生成するので、生成される MBean インタフェース ファイルの名前は、その MDF 名の後に「MBean」というテキストが付いたものになります。たとえば、WebLogic MBeanMaker で `MyCredentialMapper` MDF を実行すると、`MyCredentialMapperMBean.java` という MBean インタフェース ファイルが生成されます。

WebLogic MBeanMaker を使用して MBean JAR ファイル (MJF) を作成する

WebLogic MBeanMaker で MDF を実行して中間ファイルを作成し、MBean 実装ファイルを手作業で編集してその中にメソッドの内容を記述したら、カスタム資格マッピング プロバイダの MBean ファイルと実行時クラスを MBean JAR ファイル (MJF) にパッケージ化する必要があります。このプロセスも、WebLogic MBeanMaker によって自動化されます。

カスタム資格マッピング プロバイダの MJF を作成するには、次の手順に従います。

1. 新しい DOS シェルを作成します。
2. 次のコマンドを入力します。

   ```
   java -DMJF=jarfile -Dfiles=filesdir
   weblogic.management.commo.WebLogicMBeanMaker
   ```

   ここで、`-DMJF` フラグは WebLogic MBeanMaker が新しい MBean タイプを格納する JAR ファイルをビルドする必要があることを示し、*`jarfile`* は MJF の名前、*`filesdir`* は WebLogic MBeanMaker で MJF に JAR 化する対象ファイルが存在する場所です。

この時点でコンパイルが行われるので、エラーが発生するおそれがあります。*jarfile* が指定されていて、エラーが発生しなかった場合には、指定された名前の MJF が作成されます。

注意： 既存の MJF を更新する場合は、MJF をいったん削除してから再生成します。WebLogic MBeanMaker にも `-DIncludeSource` オプションがあり、それを指定すると、生成される MJF にソース ファイルを含めるかどうかを制御することができます。ソース ファイルには、生成されたソースと MDF そのものがあります。デフォルトは `false` です。このオプションは、`-DMJF` を使用しない場合には無視されます。

生成された MJF は、自らの WebLogic Server 環境にインストールすることも、顧客に配布してそれぞれの WebLogic Server 環境にインストールしてもらうこともできます。

WebLogic Server 環境に MBean タイプをインストールする

MBean タイプを WebLogic Server 環境にインストールするには、MJF を *WL_HOME*`\server\lib\mbeantypes` ディレクトリにコピーします。ここで、*WL_HOME* は WebLogic Server の最上位のインストール ディレクトリです。このインストール コマンドによって、カスタム資格マッピング プロバイダが「デプロイ」されます。つまり、カスタム資格マッピング プロバイダを WebLogic Server Administration Console から管理できるようになります。

注意： *WL_HOME*`\server\lib\mbeantypes` は、MBean タイプをインストールするデフォルトのディレクトリです。ただし、WebLogic Server が追加ディレクトリで MBean タイプを検索するようにするには、サーバを起動するときに `-Dweblogic.alternateTypesDirectory=<`*dir*`>` コマンドライン フラグを使用します (`<`*dir*`>` はディレクトリ名のカンマ区切りのリスト)。このフラグを使用すると、WebLogic Server は常にまず *WL_HOME*`\server\lib\mbeantypes` から MBean タイプをロードし、その後に追加ディレクトリを検索して、(拡張子に関係なく) そのディレクトリに存在するすべての有効なアーカイブをロードします。たとえば `-Dweblogic.alternateTypesDirectory = dirX,dirY` の場合、WebLogic Server はまず *WL_HOME*`\server\lib\mbeantypes` から MBean タイプをロードし、その後に、`dirX` および `dirY` に存在する有効なアーカイブをロードします。

追加ディレクトリで MBean タイプを検索するように WebLogic Server に指示し、かつ Java セキュリティ マネージャを使用する場合は、`weblogic.policy` ファイルを更新して、MBean タイプ (したがってカスタム セキュリティ プロバイダも) に対する適切なパーミッションを与える必要があります。詳細については、『WebLogic Security プログラマーズ ガイド』の「Java セキュリティ マネージャを使用しての WebLogic リソースの保護」を参照してください。

非セキュリティ プロバイダの JAR (バックアップ ファイルを含む) は、*`WL_HOME`*`\server\lib\mbeantypes` ディレクトリ以外の場所に置くことをお勧めします。

カスタム資格マッピング プロバイダをコンフィグレーションすることによって (11-16 ページの「Administration Console を使用してカスタム資格マッピング プロバイダをコンフィグレーションする」を参照) MBean タイプのインスタンスを作成して、GUI、他の Java コード、または API からそれらの MBean インスタンスを使用することができます。たとえば、WebLogic Server Administration Console を使用して、属性を取得 / 設定したり操作を呼び出したりすることもできますし、他の Java オブジェクトを開発して、そのオブジェクトで MBean をインスタンス化し、それらの MBean から提供される情報に自動的に応答させることもできます。なお、これらの MBean インスタンスをバックアップしておくことをお勧めします。詳細については、『WebLogic Server ドメイン管理』の「障害が発生したサーバの回復」の「セキュリティ コンフィグレーション データのバックアップ」を参照してください。

Administration Console を使用してカスタム資格マッピング プロバイダをコンフィグレーションする

カスタム資格マッピング プロバイダをコンフィグレーションするということは、資格マッピング サービスを必要とするアプリケーションがアクセス可能なセキュリティ レルムにカスタム資格マッピング プロバイダを追加するということです。

カスタム セキュリティ プロバイダのコンフィグレーションは管理タスクですが、カスタム セキュリティ プロバイダの開発者が行うこともできます。この節では、カスタム資格マッピング プロバイダのコンフィグレーション担当者向けの重要な情報を取り上げます。

- 11-17 ページの「資格マッピング プロバイダ、リソース アダプタ、およびデプロイメント記述子の管理」
- 11-19 ページの「デプロイ可能な資格マッピングの有効化」

注意： WebLogic Server Administration Console を使用したカスタム資格マッピング プロバイダのコンフィグレーション手順については、『WebLogic Security の管理』の「カスタム セキュリティプロバイダのコンフィグレーション」を参照してください。

資格マッピング プロバイダ、リソース アダプタ、およびデプロイメント記述子の管理

リソース アダプタ (コネクタ) などのアプリケーション コンポーネントの中には、関連デプロイメント情報を Java 2 Enterprise Edition (J2EE) デプロイメント記述子および WebLogic Server デプロイメント記述子に格納するものがあります。リソース アダプタの場合、デプロイメント記述子ファイル (`weblogic-ra.xml`) には、ユーザ名 / パスワードの組み合わせなど、資格マップを作成するための情報が含まれます。この資格マップ情報は、WebLogic Server Administration Console で資格マッピング プロバイダを初めてコンフィグレーションするときに含めるのが一般的です。

Administration Console には、この目的のために [デプロイメント記述子内のセキュリティ データを無視] チェックボックスが用意されています。開発者または管理者がカスタム資格マッピング プロバイダを初めてコンフィグレーションするときには、このチェックボックスのチェックがはずれていることを確認する必要があります。

注意： [デプロイメント記述子内のセキュリティ データを無視] チェックボックスは、デフォルトではチェックがはずれています。このチェックボックスを設定するには、WebLogic Server Administration Console の左ペインで [セキュリティ | レルム | *`realm`*] をクリックします。*`realm`* はセキュリティ レルムの名前です。次に [一般] タブを選択します。

このチェックボックスのチェックをはずしてリソース アダプタ (コネクタ) をデプロイすると、WebLogic Server は、weblogic-ra.xml デプロイメント記述子ファイル (コード リスト 11-2 の例を参照) から資格マップを読み込みます。この情報は、資格マッピング プロバイダのセキュリティ プロバイダ データベースにコピーされます。

コード リスト 11-2　weblogic-ra.xml ファイルのサンプル

```
<weblogic-connection-factory-dd>
   <connection-factory-name>LogicalNameOfBlackBoxNoTx</connection-factory-name>
   <jndi-name>eis/BlackBoxNoTxConnectorJNDINAME</jndi-name>

   <map-config-property>
     <map-config-property-name>ConnectionURL</map-config-property-name>
     <map-config-property-value>jdbc:pointbase:server://localhost/demo
     <map-config-property-value>
   </map-config-property>

   <security-principal-map>
     <map-entry>
       <initiating-principal>*</initiating-principal>
       <resource-principal>
          <resource-username>examples</resource-username>
          <resource-password>examples</resource-password>
       </resource-principal>
     </map-entry>
   </security-principal-map>

</weblogic-connection-factory-dd>
```

注意： コード リスト 11-2 に示したサンプル リソース アダプタのデプロイメント記述子は、*WL_HOME*\samples\server\src\examples\jconnector\simple\rars\META-INF にあります。ここで、*WL_HOME* は WebLogic Server の最上位のインストール ディレクトリです。

デプロイメント記述子ファイルでも Administration Console でも追加の資格マップを設定することができますが、リソース アダプタのデプロイメント記述子に定義されている資格マップをいったんコピーしてから、Administration Console を使用して追加の資格マップを定義することをお勧めします。この理由は、資格マッピング プロバイダのコンフィグレーション中に Administration Console を使

用して資格マップを変更すると、その内容が weblogic-ra.xml ファイルに保持されないからです。Administration Console を使用して再デプロイする、ディスク上でリソース アダプタ (コネクタ) を変更した、または WebLogic Server を再起動したといった場合に、リソース アダプタ (コネクタ) を再デプロイするときには、[デプロイメント記述子内のセキュリティ データを無視] チェックボックスをチェックする必要があります。チェックボックスをチェックしないと、Administration Console を使用して定義した資格マップがデプロイメント記述子で定義されている資格マップによって上書きされます。

注意： [デプロイメント記述子内のセキュリティ データを無視] チェックボックスは、ロール マッピング プロバイダおよび認可プロバイダにも影響します。詳細については、それぞれ 6-22 ページの「認可プロバイダとデプロイメント記述子の管理」と 8-26 ページの「ロール マッピング プロバイダとデプロイメント記述子の管理」を参照してください。

デプロイ可能な資格マッピングの有効化

DeployableCredentialProvider SSPI を実装し、カスタム資格マッピング プロバイダでデプロイ可能な資格マップをサポートしたい場合、カスタム資格マッピング プロバイダのコンフィグレーション担当者 (つまり、開発者または管理者) は、Administration Console で [資格マッピング デプロイメントを有効化] チェックボックスがチェックされていることを確認する必要があります。チェックがはずれていると、資格マッピング プロバイダに対するデプロイメントは「オフ」と見なされます。このため、複数の資格マッピング プロバイダがコンフィグレーションされている場合、[資格マッピング デプロイメントを有効化] チェックボックスを使用すると、資格マップのデプロイメントに使用する資格マッピング プロバイダを指定できます。

注意： [デプロイメント記述子内のセキュリティ データを無視] チェックボックス (11-17 ページの「資格マッピング プロバイダ、リソース アダプタ、およびデプロイメント記述子の管理」で説明したようにセキュリティ レルム レベルで指定) では、コンフィグレーション済みの資格マッピング プロバイダのセキュリティ データベースに資格マップをコピーするかどうかを指定します。[資格マッピング デプロイメントを有効化] チェックボックス (コンフィグレーション済みの資格マッピング プロバイダごとに指定) では、資格マッピング プロバイダがデプロイ済みの資格マップを格納するかどうかを指定します。

資格マップを管理するためのメカニズムを提供する

WebLogic Server Administration Console を使用してカスタム資格マッピング プロバイダをコンフィグレーションすると、必要な資格マッピング サービスにアプリケーションからアクセスできるようにすることはできますが、このセキュリティ プロバイダに関連付けられた資格マップを管理する方法を管理者にも提供する必要があります。つまり、資格マップを管理するためのメカニズムを提供する必要があります。このメカニズムでは、カスタム資格マッピング プロバイダのデータベースの資格マップを読み書きできなければなりません。

それには、以下の 2 通りの方法があります。

- 11-20 ページの「オプション 1 : 資格マップ管理用のスタンドアロン ツールを開発する」
- 11-21 ページの「オプション 2 : 既存の資格マップ管理ツールを Administration Console に統合する」

注意 : WebLogic Server のこのバージョンでは、資格マッピング プロバイダ用にコンソール拡張を使用して「エディタ」ページを作成することはできません。カスタムの認可プロバイダおよびロール マッピング プロバイダでは可能です。

オプション 1 : 資格マップ管理用のスタンドアロン ツールを開発する

WebLogic Server Administration Console とまったく別のツールを開発する場合には、この方法を選択します。

この方法の場合、ツールでは以下のことを行う必要があります。

1. WebLogic リソースの ID を特定します。詳細については、2-26 ページの「WebLogic リソース識別子」を参照してください。
2. ローカル ユーザとリモート ユーザとの関係を表す方法を決定します。この表現は完全に任意です。
3. カスタム資格マッピング プロバイダのデータベースの式を読み書きします。

オプション 2：既存の資格マップ管理ツールを Administration Console に統合する

WebLogic Server Administration Console とは別のツールを持っており、それを Administration Console から起動する場合には、この方法を選択します。

この方法の場合、ツールでは以下のことを行う必要があります。

1. WebLogic リソースの ID を特定します。詳細については、2-26 ページの「WebLogic リソース識別子」を参照してください。
2. ローカル ユーザとリモート ユーザとの関係を表す方法を決定します。この表現は完全に任意です。
3. カスタム資格マッピング プロバイダのデータベースの式を読み書きします。
4. 『Administration Console の拡張』に説明されているように、基本コンソール拡張を使用して Administration Console にリンクします。

12 カスタム セキュリティ プロバイダ用のコンソール拡張の記述

コンソール拡張を使用すると、標準 WebLogic Server Administration Console に含まれていない機能を追加したり、既存の機能のインタフェースを提供したりできます。こうした機能を提供するには、ナビゲーション ツリーにノードを追加し、タブ付きダイアログとダイアログ画面を追加または置き換えます。

注意： 以下の節に進む前に、『Administration Console の拡張』でコンソール拡張の記述方法について理解しておいてください。

以下の節では、特にカスタム セキュリティ プロバイダと共に使用するコンソール拡張の記述について説明します。

コンソール拡張の記述が必要な場合

WebLogic Server Administration Console でカスタム セキュリティ プロバイダ用の完全なコンフィグレーションおよび管理サポートを利用するには、以下の場合にコンソール拡張を記述する必要があります。

- カスタム セキュリティ プロバイダ用の MBean タイプを生成するときに任意 SSPI MBean を実装しないことを決定したが、Administration Console でカスタム セキュリティ プロバイダをコンフィグレーションおよび管理する場合 (つまり、WebLogic Server コマンドライン インタフェースを代わりに使用しない場合)

  BEA では、MBean タイプの生成 (1-4 ページの「カスタム セキュリティ プロバイダをコンフィグレーションおよび管理する MBean タイプの生成」を参照) によってカスタム セキュリティ プロバイダをコンフィグレーションおよび管理することをお勧めします。しかし、ユーザが記述したコンソール拡張を介してカスタム セキュリティ プロバイダを完全にコンフィグレーションおよび管理することもできます。

- カスタム認証プロバイダ以外の任意 SSPI MBean を実装する場合

  任意 SSPI MBean を実装してカスタム認証プロバイダを開発する場合は、Administration Console で MBean タイプの属性が自動的にサポートされることになります (任意 SSPI MBean から継承される)。カスタム認可プロバイダなどの他のタイプのカスタム セキュリティ プロバイダでは、このサポートは受けられません。

- シンプルなデータ型として表すことができないカスタム属性を、MBean 定義ファイル (MDF) (カスタム セキュリティ プロバイダの MBean タイプを作成するために使用される) に追加する場合

  カスタム セキュリティ プロバイダの [詳細] タブにカスタム属性が自動的に表示されるのは、それらがシンプルなデータ型 (文字列、MBean、ブール、整数値) として表される場合のみです。非定型のデータ型 (フィンガープリントのイメージなど) として表されるカスタム属性がある場合、Administration Console ではカスタマイズを行わない限りその属性を表示できません。

- カスタム操作を MBean 定義ファイル (MDF) (カスタム セキュリティ プロバイダの MBean タイプを作成するために使用される) に追加する場合。

  カスタム操作はさまざまなので、Administration Console ではそれらを自動的に表示または処理する方法が分かりません。カスタム操作の例としては、声紋用のマイクや、インポート / エクスポート ボタンなどがあります。Administration Console では、カスタマイズしない限りこれらのプロセスを表示できません。

Administration Console は、以下の場合にも拡張します。

- 企業ブランディング ― たとえば、カスタム セキュリティ プロバイダのコンフィグレーションおよび管理に使用するページにロゴを表示する場合など
- 連結 ― たとえば、カスタム セキュリティ プロバイダのコンフィグレーションおよび管理に使用するすべてのフィールドを、個別のタブや場所ではなく 1 ページに表示する場合など

開発プロセスでのコンソール拡張の記述

コンソール拡張を構成するさまざまなプログラマティック要素は、1 つの Web アプリケーションにパッケージ化されて WebLogic Server ドメインにデプロイされます。開発プロセスでいつ Web アプリケーションを開発するかは、完全にユーザの自由です。

ただし、コンソール拡張を使用してカスタム セキュリティ プロバイダをコンフィグレーションおよび管理する前に、そのカスタム セキュリティ プロバイダ用の MBean タイプを生成し (1-4 ページの「カスタム セキュリティ プロバイダをコンフィグレーションおよび管理する MBean タイプの生成」を参照)、コンソール拡張 Web アプリケーションを適切にパッケージ化してデプロイしておく必要があります。

注意： コンソール拡張を Web アプリケーションとして開発、パッケージ化、およびデプロイする手順については、12-4 ページの「Administration Console 拡張を記述する主要な手順」を参照してください。

カスタム セキュリティ プロバイダ用のコンソール拡張と基本コンソール拡張との違い

基本コンソール拡張 (『Administration Console の拡張』を参照) が柔軟性と機能に富んでいるのに対し、セキュリティ プロバイダ固有のコンソール拡張で追加されるメカニズムでは以下のことが可能になります。

- カスタム セキュリティ プロバイダをコンフィグレーションおよび管理するための既存の Administration Console ページに緊密に統合できる
- タブ付きダイアログまたはダイアログ画面を複数の異なる場所で統合できる (基本コンソール拡張ではタブ付きダイアログとダイアログ画面は新しいナビゲーション ツリー ノードの一部としてしか追加できない)
- カスタム セキュリティ プロバイダをコンフィグレーションおよび管理するための既存のタブ付きダイアログまたはダイアログ画面を置き換えることができる

Administration Console 拡張を記述する主要な手順

12-3 ページの「カスタム セキュリティ プロバイダ用のコンソール拡張と基本コンソール拡張との違い」で説明したように、セキュリティ プロバイダ固有のコンソール拡張にはいくつかの追加機能が用意されていますが、コンソール拡張を記述する主要な手順は同じです。

1. Administration Console 拡張を定義する Java クラスを作成します。このクラスでは、コンソール拡張をナビゲーション ツリー内のどこに表示するか、およびコンソール拡張で必要とされる追加機能を定義します。詳細については、『Administration Console の拡張』の「NavTreeExtension インタフェースの実装」を参照してください。
2. ナビゲーション ツリーの動作を定義します。この手順では、手順 1 で定義したノードの下に表示する複数のノードを定義できます。また、右クリック メニューとアクションも定義できます。詳細については、『Administration Console の拡張』の「ナビゲーション ツリーの設定」を参照してください。
3. コンソール拡張画面を表示するための JavaServer Pages (JSP) を記述します。ローカライゼーション カタログ内の文字列を参照することによって、ローカライズされたテキストを使用できます。標準のタグ ライブラリを使用すると、標準の Administration Console と同じようなタブ付きダイアログ画面を

作成し、ローカライゼーション カタログにアクセスできます。詳細については、『Administration Console の拡張』の「Console 画面 JSP の記述」を参照してください。

4. 複数の言語で表示するためにコンソール拡張をローカライズします。詳細については、『Administration Console の拡張』の「Administration Console 拡張のローカライズ」を参照してください。
5. JSP、カタログ、および Java クラスを Web アプリケーションとしてパッケージ化します。詳細については、『Administration Console の拡張』の「Administration Console 拡張のパッケージ化」を参照してください。
6. コンソール拡張が収められた Web アプリケーションを WebLogic Server ドメインの管理サーバにデプロイします。詳細については、『Administration Console の拡張』の「Administration Console 拡張のデプロイ」を参照してください。

SecurityExtension インタフェースの使用によるカスタム セキュリティ プロバイダ関連の Administration Console ダイアログ画面の置き換え

`SecurityExtension` インタフェースのメソッドを使用すると、さまざまなカスタム セキュリティ プロバイダ関連の Administration Console ダイアログ画面を置き換えることができます。コンソール拡張を定義するために作成する Java クラスは、`Extension` クラスを拡張するほかに (またはその代わりに) `SecurityExtension` インタフェースを実装できます。`Extension` クラスは、基本コンソール拡張用に使用します (使い方については『Administration Console の拡張』の「NavTreeExtension インタフェースの実装」を参照)。

注意 : このインタフェースのすべてのメソッドを実装する必要があります。置き換えないページについては `null` を返します。

表 12-1 に、置き換えることの多いセキュリティ プロバイダ関連のダイアログ画面と、それらを置き換えるために実装する必要がある SecurityExtension インタフェースのメソッドを示します。

表 12-1 SecurityExtension インタフェースの使い方

置き換えるダイアログ画面の用途	実装するメソッド
新しいカスタム セキュリティ プロバイダをコンフィグレーションし、既存のカスタム セキュリティ プロバイダのコンフィグレーションを編集する	getExtensionForProvider メソッド
新規ユーザの作成と既存のユーザの編集 (カスタム認証プロバイダと共に使用)	getExtensionForUser メソッド
新規グループの作成と既存のグループの編集 (カスタム認証プロバイダと共に使用)	getExtensionForGroup メソッド
新規ロールの作成と既存のロールの編集 (カスタム ロール マッピング プロバイダと共に使用)	getExtensionForRole メソッド
新規セキュリティ ポリシーの作成と既存のセキュリティ ポリシーの編集 (カスタム認可プロバイダと共に使用)	getExtensionForPolicy メソッド

注意 : 詳細については、WebLogic Server 7.0 API リファレンス Javadoc の SecurityExtension インタフェースと Extension クラスを参照してください。

コンソール拡張が Administration Console に与える影響

コンソール拡張を記述する目的が、カスタム セキュリティ プロバイダのコンフィグレーション用の BEA 標準ダイアログ画面を置き換えることであっても、セキュリティ プロバイダに関連するユーザ、グループ、ロール、セキュリティ ポリシーの作成および編集用のダイアログ画面を置き換えることであっても、WebLogic Server Administration Console への影響は同じです。

たとえば、WebLogic Server Administration Console を使用してカスタム セキュリティ プロバイダをコンフィグレーションしようとすると、以下のプロセスが発生します。

1. Administration Console のダイアログ画面の 1 つで [新しい Security_Provider_Type のコンフィグレーション] リンク (図 12-1 の上部がその例) をクリックすると、Administration Console はカスタム セキュリティ プロバイダ用のコンソール拡張を検索しようとします。

図 12-1　サンプル認証プロバイダのコンフィグレーション

カスタム セキュリティ プロバイダのコンフィグレーションを (手順 1 のように追加するのではなく) 編集する場合、Administration Console はそのカスタム セキュリティ プロバイダのハイパーリンク名 (図 12-1 の下部がその例) がクリックされるとコンソール拡張を検索します。

2. Administration Console は、セキュリティ プロバイダのコンソール拡張が使用可能であることを検出すると、`getExtensionForProvider` メソッド (または他の `getExtensionFor*` メソッド (12-6 ページの表 12-1「SecurityExtension インタフェースの使い方」を参照)) から返された URL によって指定された JavaServer Page (JSP) を表示します。
3. BEA の標準インタフェースの代わりにこの JSP を使用して、カスタム セキュリティ プロバイダをコンフィグレーションおよび管理します。

A MBean 定義ファイル (MDF) 要素の構文

MBean 定義ファイル (MDF) は WebLogic MBeanMaker ユーティリティへの入力ファイルです。WebLogic MBeanMaker ユーティリティは、このファイルを使用してカスタム セキュリティ プロバイダを管理するための MBean タイプを作成します。MDF は、単一の MBean タイプを記述する有効な整形式 XML ファイルとしてフォーマットされる必要があります。以下の節では、有効な MDF で使用可能なすべての要素と属性について説明します。

MBeanType (ルート) 要素

すべての MDF には `MBeanType` というルート要素が 1 つ含まれていなければなりません。この要素の構文は次のとおりです。

```
<MBeanType Name= string   optional_attributes>
    subelements
</MBeanType>
```

MBeanType 要素には Name 属性が含まれている必要があります。これは、その MBean タイプのプログラム上の内部的な名前を指定するものです (ユーザ インタフェースに表示される名前を指定するには、DisplayName 属性と LanguageMap 属性を使用します)。他の属性は省略可能です。

以下の例は、MBeanType (ルート) 要素を簡略化して示したものです。

```
<MBeanType Name="MyMBean" Package="com.mycompany">
   <MBeanAttribute Name="MyAttr" Type="java.lang.String" Default="Hello World"/>
</MBeanType>
```

MBeanType (ルート) 要素内に指定されている属性は、その MBean タイプからインスタンス化されるすべての MBean に適用されます。特定の MBean インスタンスの属性をオーバーライドするには、MBeanAttribute 下位要素で属性を指定する必要があります。詳細については、A-16 ページの「MBeanAttribute 下位要素」を参照してください。

表 A-1 では、MBeanType (ルート) 要素で使用可能な属性について説明します。「JMX 仕様 /BEA 拡張機能」カラムは、当該属性が JMX 仕様に対する BEA 拡張機能なのか、それとも JMX の標準属性なのかを示します。なお、BEA 拡張機能は他の J2EE Web サーバ上では動作しない可能性があることに注意してください。

表 A-1MBeanType (ルート) 要素の属性

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Abstract	BEA 拡張機能	true/false	値が true の場合には、当該 MBean タイプは (Java 抽象クラスと同様に) インスタンス化できないことを意味する。ただし、他の MBean タイプは当該 MBean タイプの属性と操作を継承できる。true を指定した場合には、管理タスクを実行するための非抽象 MBean タイプを別途作成する必要がある。この属性の値を指定しない場合には、値が false であると仮定される。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
CachingDisabled	BEA 拡張機能	true/false	無視される。このフラグは、将来の機能をサポートするために存在している。たとえば、キャッシング スタブでキャッシングを提供する場合、このフラグはキャッシングを無効にする場合がある。 現在、WebLogic MBeanMaker で作成された MBean は、JMX Model MBean の仕様により、サーバレベルのキャッシングを提供する。 **注意 :** JMX Model MBean の詳細については、Java Management eXtensions 1.0 仕様を参照のこと。
Classification	BEA 拡張機能	文字列	この文字列を使用すると、MBean タイプを分類またはグループ化して、セキュリティ認可の実装を提供するすべての MBean タイプを識別するといったことができる。この属性には、デフォルト値または仮定される値はない。
CurrencyTimeLimit	JMX 仕様	整数	キャッシュされた値が有効期限内にあるとみなされる許容時間 (秒数)。この時間が経過した後、値にアクセスしようとすると、再計算がトリガされる。 MBeanType 要素に指定すると、この値は MBean タイプのデフォルトと見なされる。同じ属性を MBean の MBeanAttribute または MBeanOperation 下位要素に設定すると、個々の MBean に関してオーバーライドできる。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
`Deprecated`	BEA 拡張機能	`true`/`false`	当該 MBean タイプが非推奨扱いになっていることを示す。この情報は生成された Java ソースに表示されると共に、管理アプリケーションでの使用を想定して `ModelMBeanInfo` オブジェクトにも格納される。この属性を指定しない場合には、値が `false` であると仮定される。
`Description`	JMX 仕様	文字列	当該 MBean タイプに関連付けられる任意の文字列で、生成されたクラスの Javadoc など、さまざまな場所に現れる。デフォルト値または仮定される値はない。 **注意：** ユーザ インタフェース内に表示される概要記述を指定するには、`DisplayName`、`DisplayMessage`、および `PresentationString` の各属性を使用する。
`DisplayMessage`	JMX 仕様	文字列	ユーザ インタフェースに表示される MBean タイプの説明用メッセージ。デフォルト値または仮定される値はない。 `DisplayMessage` はツールのヒントやヘルプで用いられるパラグラフになる場合がある。MBean タイプに設定された `DisplayMessage` は、インスタンスの作成時に個々の MBean に対して別の値を指定しない限り、MBean インスタンスのデフォルトと見なされる。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
DisplayName	JMX 仕様	文字列	ユーザ インタフェースに表示される MBean タイプのインスタンス識別用の名前。タイプ X のインスタンスの場合、デフォルト DisplayName は「タイプ X のインスタンス」となる。通常、この値はインスタンス作成時にオーバーライドされる。 LanguageMap 属性を使用する場合には、DisplayName 値は LanguageMap のリソース バンドル内で名前を見つけるためのキーとして使われる。LanguageMap 属性を指定しない場合や、そのキーがリソース バンドルに存在しない場合には、DisplayName 値そのものがユーザ インタフェースに表示される。 「MessageID」を参照。
Export	JMX 仕様	文字列	WebLogic MBeanMaker では、この属性は使用されない。ただし、これを使用する可能性のあるアプリケーションをサポートするために、WebLogic MBeanMaker では、この値を MBeanInfo オブジェクトに追加する。デフォルト値または仮定される値はない。 **注意：** Export 属性の詳細については、Java Management eXtensions 1.0 仕様を参照のこと。
Extends	BEA 拡張機能	パス名	この MBean タイプが拡張する完全修飾 MBean タイプ名。 「Implements」を参照。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
GenerateExtended Accessors	BEA 拡張機能	true/false	値が true の場合には、Type が array の MBeanAttribute 下位要素はすべて、索引アクセスをサポートするための付加的な操作とインタフェース メソッドを生成できるようになる。値が false の場合には、すべての MBeanAttribute 下位要素が、付加的な操作とメソッドを生成できなくなる。この属性を指定しない場合には、値が true であると仮定される。
Implements	BEA 拡張機能	カンマで区切ったリスト	この MBean タイプを実装する完全修飾 MBean タイプ名からなるカンマ区切りリスト。 「Extends」を参照。
InstanceExtent	BEA 拡張機能	true/false	値が true の場合には、アクセスしやすくなるように、MBean タイプのインスタンスがすべて 1 つのリストに保持されることになる。ただし、この属性を true に設定すると、メモリ消費量が増える。 **注意：** デフォルトでは、セキュリティ関係のすべての MBean タイプが true に設定されている。それらの MBean のインスタンスでこの属性をオーバーライドすると、セキュリティに悪影響が及ぶおそれがある。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
LanguageMap	BEA 拡張機能	文字列	表示可能な文字列のマップを格納したリソース バンドルの完全修飾パス名を指定する。他の属性 (たとえば DisplayMessage や DisplayName など) は、LanguageMap 内の文字列を用いて、MBean タイプに関する情報を表示する。 この属性を指定しない場合には、他の属性 (たとえば DisplayMessage や DisplayName など) は、自分自身の値を表示する (それらの値をキーに使ってリソース バンドル内の適切な文字列を検索するわけではない)。
Listen	BEA 拡張機能	true/false	通知リスナのスタブを MBean 実装オブジェクト内に生成する。この属性を指定しない場合には、値が false であると仮定される。 **注意 :** リスナ スタブの詳細については、Java Management eXtensions 1.0 仕様を参照のこと。
Log	JMX 仕様	true/false	値が true の場合、MBean タイプの通知をログ ファイルに追加するよう指定する (LogFile 属性は情報の出力先となるファイルを指定する)。この属性を指定しない場合には、値が false であると仮定され、通知はログ ファイルに追加されない。 **注意 :** MBean 通知およびログの詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
LogFile	JMX 仕様	パス名	この MBean タイプの通知が発生したときにメッセージが出力される既存の書き込み可能ファイルの完全修飾パス名。ロギングが行われるためには、Log 属性を true に設定しておく必要がある。この属性には、デフォルト値または仮定される値はない。 **注意：** MBean 通知およびログの詳細については、Java Management eXtensions 1.0 仕様を参照のこと。
MBeanClassName	BEA 拡張機能	パス名	BEA から提供される MBean タイプのサブクラスの完全修飾クラス名。これは、主に将来の開発用および Model MBean の拡張を望む開発者向けに用意されている。 **注意：** JMX Model MBean の詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
MessageID	JMX 仕様	文字列	Java Management eXtensions 1.0 仕様に従ってクライアントサイド メッセージ リポジトリからメッセージを取得するためのキーを提供する。MessageID と DisplayMessage のどちらか一方あるいは両方を用いて、通知 MBean タイプを記述することができる。この属性を指定しない場合には、メッセージ ID は利用できない。 **注意：** MessageID は、LanguageMap 属性で指定されるのと同じリソース バンドルを使用せず、通知 MBean タイプ専用である。
Name	JMX 仕様	文字列	当該 MBean タイプのプログラム上の内部的な名前を指定する必須属性。
Package	BEA 拡張機能	文字列	当該 MBean タイプのパッケージ名を指定し、WebLogic MBeanMaker によって作成されるクラス ファイルの場所を決定する。この属性を指定しない場合には、MBean タイプは Java デフォルト パッケージに格納される。 **注意：** パッケージが異なっている限り、MBean タイプ名は同じ名前でもかまわない。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
`PersistLocation`	JMX 仕様	パス名	WebLogic MBeanMaker では、この属性は使用されない。ただし、`PersistLocation` を使用する Model MBean を MBean タイプが拡張するような場合をサポートするために、WebLogic MBeanMaker では、この属性値を `MBeanInfo` クラスに追加する。デフォルト値または仮定される値はない。 **注意：** `PersistLocation` および Model MBean の詳細については、Java Management eXtensions 1.0 仕様を参照のこと。
`PersistName`	JMX 仕様	文字列	WebLogic MBeanMaker では、この属性は使用されない。ただし、`PersistName` を使用する Model MBean を MBean タイプが拡張するような場合をサポートするために、WebLogic MBeanMaker では、この属性値を `MBeanInfo` クラスに追加する。デフォルト値または仮定される値はない。 **注意：** `PersistName` および Model MBean の詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
PersistPeriod	JMX 仕様	整数	OnTimer または NoMoreOftenThan の永続性ポリシーで使用される秒数を指定する。MBeanType または MBeanAttribute 要素でこの属性を指定しない場合には、値が 0 であると仮定される。 PersistPolicy が OnTimer に設定されている場合には、この秒数が経過したら属性が永続化される。一方、PersistPolicy が NoMoreOftenThan に設定されている場合には、指定された秒数以上の間隔を置いて永続化が行われるように制限される。 **注意：** MBeanType 要素内で指定された場合、この属性値は、個々の MBeanAttribute 下位要素内のあらゆる設定をオーバーライドする。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
PersistPolicy	JMX 仕様	Never /OnTimer /OnUpdate /NoMoreOftenThan	永続化がどのように行われるかを、以下のいずれかで指定する。 ■ Never：属性は決して格納されない。これは、非常に変わりやすいデータやセッションのコンテキスト内や実行期間中にしか意味を持たないデータの場合に役に立つ。 ■ OnTimer：属性は、当該 MBean タイプの永続性タイマーの設定時間 (PersistPeriod 属性で定義される) が経過するたびに格納される。 ■ OnUpdate：属性は更新のたびに格納される。 ■ NoMoreOftenThan：属性は、更新間隔が PersistPeriod より短くない限り、更新のたびに格納される。このメカニズムは、非常に変わりやすいデータの影響でパフォーマンスが下がることのないようにする上で役に立つ。 MBeanType または MBeanAttribute 要素でこの属性を指定しない場合には、値が Never であると仮定される。 **注意：** MBeanType 要素内で指定された場合、この属性値は、個々の MBeanAttribute 下位要素内のあらゆる設定をオーバーライドする。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
PresentationString	JMX 仕様	パス名	ユーザ インタフェースで項目の表示に使用できる情報を提供する単一の XML ドキュメントの完全修飾パス名。この XML ドキュメントは、プレゼンテーション ロジックに関係のある付加的なメタデータも提供する。PresentationString のフォーマットは、XML/JMX に準拠した任意の情報である。 **注意：** 現在、BEA では専用のフォーマットを定義していないが、独自に定義するより待つことをお勧めする。PresentationString 属性は、将来使用するためのものである。
Readable	JMX 仕様	true/false	MBean の属性値を MBean API を通じて読み込めるかどうかを決定する。MBeanType または MBeanAttribute 要素でこの属性を指定しない場合には、値が true であると仮定される。 MBeanType 要素に指定すると、この値は個々の MBeanAttribute 下位要素のデフォルトと見なされる。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Servers	BEA 拡張機能	カンマで区切ったリスト	MBean タイプのインスタンスをインスタンス化でき、その MBean タイプのインスタンスを参照できるサーバのリストを定義する。MBean タイプのインスタンスは、たとえ管理サーバが利用できなくても、これらのサーバから必ずアクセス (読み取りアクセス) できる。 カンマで区切ったリストには、MBean タイプと同じ管理ドメインにあるサーバの名前を指定する必要がある。値が指定されない場合には、グローバル スコープであると仮定される。つまり、インスタンスはドメイン内のすべてのサーバから参照できる。リスト内に管理対象サーバが 1 つしかない場合には、サーバ固有のスコープになる。つまり、インスタンスは管理対象サーバからしか参照できず、他のサーバにはレプリケートできない。管理サーバと 1 つ以上の管理対象サーバがリストに含まれている場合には、共有スコープになる。つまり、インスタンスは管理サーバと指定された管理対象サーバから参照できる。管理サーバなしで複数の管理対象サーバを指定するとエラーが発生する。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
VersionID	BEA 拡張機能	*Long*	Java serialVersionUID に変換する。指定された 値は、生成された実装ファイルに次のフォーマットで直接配置される。 static final long serialVersionUID = <user provided ID>; MBean クラスを互換性を持たない方法で変更した場合、Java シリアライゼーションが正しく機能するように、VersionID を使用して serialVersionUID を変更する必要がある。 serialVersionUID の詳細については、Java 2 Platform Standard Edition v1.3.1 API 仕様を参照のこと。
Visibility	JMX 仕様	整数 : 1-4	MBean タイプの重要度を示す。ユーザ インタフェースではこの数値を用いて、特定のコンテキストで特定のユーザに当該 MBean タイプを表示するかどうかを決定する。この値が小さければ小さいほど、重要度は高くなる。1 から 4 までの数値を指定できる。この属性を指定しない場合には、値が 1 であると仮定される。 ユーザの関心の高いアイテムの場合には、Visibility に低い数値を指定する。たとえば、WebLogic Server Administration Console では、Visiblity 値が 1 の MBean が左ペインのナビゲーション ツリーに表示される。

表 A-1MBeanType (ルート) 要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Writeable	JMX 仕様	true/false	値が true の場合には、MBean API を通じて MBeanAttribute の値を設定できるようになる。 MBeanType または MBeanAttribute でこの属性を指定しない場合には、値が true であると仮定される。 MBeanType 要素に指定すると、この値は個々の MBeanAttribute 下位要素のデフォルトと見なされる。

MBeanAttribute 下位要素

MBean タイプ内の属性ごとに、MBeanAttribute 下位要素のインスタンスを 1 つ記述する必要があります。MBeanAttribute 下位要素の形式は次のとおりです。

```
<MBeanAttribute Name=string   optional_attributes />
```

MBeanAttribute 下位要素には Name 属性が含まれている必要があります。この属性は、その MBean タイプの Java 属性のプログラム上の内部的な名前を指定します (ユーザ インタフェースに表示される名前を指定するには、DisplayName 属性と LanguageMap 属性を使用します)。他の属性は省略可能です。

以下の例は、MBeanType 要素内の MBeanAttribute 下位要素を簡略化して示したものです。

```
<MBeanType Name="MyMBean" Package="com.mycompany">
    <MBeanAttribute Name= "WhenToCache"
     Type="java.lang.String"
     LegalValues="'cache-on-reference','cache-at-initialization','cache-never'"
     Default= "cache-on-reference"
```

```
    />
</MBeanType>
```

MBeanAttribute 下位要素内に指定されている属性は、特定の MBean インスタンスに適用されます。ある MBean タイプからインスタンス化されるすべての MBean の属性を設定するには、MBeanType (ルート) 要素内に属性を指定する必要があります。詳細については、A-1 ページの「MBeanType (ルート) 要素」を参照してください。

表 A-2 では、MBeanAttribute 下位要素で使用可能な属性について説明します。「JMX 仕様 /BEA 拡張機能」カラムは、当該属性が JMX 仕様に対する BEA 拡張機能なのかどうかを示します。なお、BEA 拡張機能は他の J2EE Web サーバ上では動作しない可能性があることに注意してください。

表 A-2MBeanAttribute 下位要素の属性

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
CachingDisabled	BEA 拡張機能	true/false	無視される。このフラグは、将来の機能をサポートするために存在している。たとえば、キャッシング スタブでキャッシングを提供する場合、このフラグはキャッシングを無効にする場合がある。 現在、WebLogic MBeanMaker で作成された MBean は、JMX Model MBean の仕様により、サーバレベルのキャッシングを提供する。 **注意：** JMX Model MBean の詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
CurrencyTimeLimit	JMX 仕様	整数	キャッシュされた値が有効期限内にあるとみなされる許容時間 (秒数)。この時間が経過した後、値にアクセスしようとすると、再計算がトリガされる。 MBeanType 要素に指定すると、この値は MBean タイプのデフォルトと見なされる。同じ属性を MBean の MBeanAttribute または MBeanOperation 下位要素に設定すると、個々の MBean に関してオーバーライドできる。
Default	JMX 仕様	文字列	MBeanAttribute 下位要素からゲッター メソッドまたはキャッシュ済みの値が提供されない場合に返すべき値。文字列は、この属性に指定されたデータ型と互換性を持つ型のオブジェクトに評価しなければならない Java 式を表す。 この属性を指定しない場合には、値が null であると仮定される。この仮定値を使用し、かつ LegalNull 属性を false に設定してある場合には、WebLogic MBeanMaker と WebLogic Server から例外が送出される。
DefaultString	JMX 仕様	文字列	Default と同じだが、属性のタイプが「文字列」の場合に使用できる。Default を文字列属性で使用する場合、値を引用符で囲まなければならない。DefaultString を使用する場合、引用符は付けない。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
`Deprecated`	BEA 拡張機能	`true`/`false`	当該 MBean 属性が非推奨扱いになっていることを示す。この情報は生成された Java ソースに表示されると共に、管理アプリケーションでの使用を想定して `ModelMBeanInfo` オブジェクトにも格納される。この属性を指定しない場合には、値が `false` であると仮定される。
`Description`	JMX 仕様	文字列	当該 MBean 属性に関連付けられる任意の文字列で、生成されたクラスの Javadoc など、さまざまな場所に現れる。デフォルト値または仮定される値はない。 **注意：** ユーザ インタフェース内に表示される概要記述を指定するには、`DisplayName`、`DisplayMessage`、および `PresentationString` の各属性を使用する。
`DisplayMessage`	JMX 仕様	文字列	ユーザ インタフェースに表示される MBean 属性の説明用メッセージ。デフォルト値または仮定される値はない。 `DisplayMessage` はツールのヒントやヘルプで用いられるパラグラフになる場合がある。MBean タイプに設定された `DisplayMessage` は、インスタンスの作成時に個々の MBean に対して別の値を指定しない限り、MBean インスタンスのデフォルトと見なされる。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
DisplayName	JMX 仕様	文字列	ユーザ インタフェースに表示される MBean タイプのインスタンス識別用の名前。 通常、MBean タイプの DisplayName のデフォルト値は、インスタンスの作成時にオーバーライドされる。タイプ X のインスタンスの場合、デフォルト DisplayName は「タイプ X のインスタンス」となる。 LanguageMap 属性を使用する場合には、DisplayName 値は LanguageMap のリソース バンドル内で名前を見つけるためのキーとして使われる。LanguageMap 属性を指定しない場合や、そのキーがリソース バンドルに存在しない場合には、DisplayName 値そのものがユーザ インタフェースに表示される。 「MessageID」を参照。
Encrypted	BEA 拡張機能	true/false	値が true の場合、この MBean 属性は設定されると暗号化されることを示す。この属性を指定しない場合には、値が false であると仮定される。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Export	JMX 仕様	文字列	WebLogic MBeanMaker では、この属性は使用されない。ただし、これを使用する可能性のあるアプリケーションをサポートするために、WebLogic MBeanMaker では、この値を MBeanInfo オブジェクトに追加する。デフォルト値または仮定される値はない。 **注意：** Export 属性の詳細については、Java Management eXtensions 1.0 仕様を参照のこと。
GenerateExtended Accessors	BEA 拡張機能	true/false	値が true の場合には、Type が array の MBean 属性はすべて、インデックス付きアクセスをサポートするための付加的な操作とインタフェース メソッドを生成できるようになる。値が false の場合には、すべての MBeanAttribute 下位要素が、付加的な操作とメソッドを生成できなくなる。この属性を指定しない場合には、値が true であると仮定される。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
GetMethod	JMX 仕様	文字列	MBean タイプのデフォルト属性処理ロジックをオーバーライドし、現在の MBeanAttribute 下位要素に使用すべきゲッター メソッドの名前を指定する。その値は、現在の MDF 内でゲッター操作を定義している MBeanOperation 下位要素の Name に一致する必要がある。 この属性を指定しない場合には、MBean ではデフォルト ロジックを用いて、MBean 属性の値を取得する。 **注意：** この属性は Readable 属性に左右される。つまり、Readable が false であれば、現在の MBeanAttribute 下位要素についてはゲッターは呼び出されない。
InterfaceType	BEA 拡張機能	文字列	WebLogic MBeanMaker で生成される MBean インタフェースの代わりに使用すべきインタフェースのクラス名。これは、現時点では無視されるが、今後の拡張で使用される可能性はある。
IsIs	JMX 仕様	true/false	生成される Java インタフェースでは MBean 属性のブール値にアクセスするのに JMX is<AttributeName> メソッド (get<AttributeName> メソッドではなく) が使用されるかどうかを指定する。この属性を指定しない場合には、値が false であると仮定される。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
`Iterable`	JMX 仕様	`true`/`false`	集約属性タイプの場合、属性が反復をサポートするかどうか (つまり、アクセスされるたびに値をインクリメントできるかどうか) を示す。この属性を指定しない場合には、値が `false` であると仮定される。 WebLogic MBeanMaker では、この属性は使用されない。ただし、これを使用する可能性のあるアプリケーションをサポートするために、WebLogic MBeanMaker では、この値を `MBeanInfo` クラスに追加する。 **注意：** `Iterable` 属性の詳細については、Java Management eXtensions 1.0 仕様を参照のこと。
`LanguageMap`	BEA 拡張機能	文字列	表示可能な文字列のマップを格納したリソース バンドルの完全修飾パス名を指定する。他の属性 (たとえば `DisplayMessage` や `DisplayName` など) は、`LanguageMap` 内の文字列を用いて、MBean 属性に関する情報を表示する。 この属性を指定しない場合には、他の属性 (たとえば `DisplayMessage` や `DisplayName` など) は、自分自身の値を表示する (それらの値をキーに使ってリソース バンドル内の適切な文字列を検索するわけではない)。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
LegalNull	BEA 拡張機能	true/false	現在の MBeanAttribute 下位要素が null を値として取り得るかどうかを指定する。この属性を指定しない場合には、値が true であると仮定される。
LegalValues	BEA 拡張機能	カンマで区切ったリスト	現在の MBeanAttribute 下位要素の取り得る値の固定集合を指定する。この属性を指定しない場合、MBean 属性では、Type 属性で指定されているタイプの任意の値を指定できる。 **注意：** このリスト内の要素は、下位要素の Type 属性で指定されるデータ型に変換可能なものでなければならない。
Listen	BEA 拡張機能	true/false	通知リスナのスタブを MBean 実装オブジェクト内に生成する。この属性を指定しない場合には、値が false であると仮定される。 **注意：** リスナスタブの詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Log	JMX 仕様	true/false	値が true の場合、MBean の通知をログ ファイルに追加するよう指定する (LogFile 属性は情報の出力先となるファイルを指定する)。この属性を指定しない場合には、値が false であると仮定され、通知はログ ファイルに追加されない。 **注意：** MBean 通知およびログの詳細については、Java Management eXtensions 1.0 仕様を参照のこと。
LogFile	JMX 仕様	パス名	この MBean 属性の通知が発生したときにメッセージが出力される既存の書き込み可能ファイルの完全修飾パス名。ロギングが行われるためには、Log 属性を true に設定しておく必要がある。この属性には、デフォルト値または仮定される値はない。 **注意：** MBean 通知およびログの詳細については、Java Management eXtensions 1.0 仕様を参照のこと。
Max	BEA 拡張機能	整数	数値型の MBean 属性タイプに限り、属性の最大値 (その値を含む) を表す数値を指定する。この属性を指定しない場合には、値はデータ型の許容最大値まで可能。 **注意：** 最大値がコンテキストに応じて動的に変化する場合、または値の範囲の指定が複雑な場合 (たとえば、1-10 または >100)、代わりに Validator 属性を使用する。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
`MessageID`	JMX 仕様	文字列	Java Management eXtensions 1.0 仕様に従ってクライアントサイド メッセージ リポジトリからメッセージを取得するためのキーを提供する。 `MessageID` と `DisplayMessage` のどちらか一方あるいは両方を用いて、通知 MBean タイプを記述することができる。この属性を指定しない場合には、メッセージ ID は利用できない。 **注意：** `MessageID` は、`LanguageMap` 属性で指定されるのと同じリソース バンドルを使用せず、通知 MBean タイプ専用である。
`Min`	BEA 拡張機能	整数	数値型の MBean 属性タイプに限り、属性の最小値 (その値を含む) を表す数値を指定する。この属性を指定しない場合には、値はデータ型の許容最小値まで可能。 **注意：** 最小値がコンテキストに応じて動的に変化する場合、または値の範囲の指定が複雑な場合 (たとえば、1-10 または >100)、代わりに `Validator` 属性を使用する。
`Name`	JMX 仕様	文字列	当該 MBean 属性のプログラム上の内部的な名前を指定する必須属性。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
NoDump	BEA 拡張機能	true/false	値が true の場合には、WebLogic MBeanDumper ユーティリティで当該 MBean 属性はダンプされなくなる。
PersistLocation	JMX 仕様	パス名	WebLogic MBeanMaker では、この属性は使用されない。ただし、PersistLocation を使用する Model MBean を MBean が拡張するような場合をサポートするために、WebLogic MBeanMaker では、この属性値を MBeanInfo クラスに追加する。デフォルト値または仮定される値はない。 **注意：** PersistLocation および Model MBean の詳細については、Java Management eXtensions 1.0 仕様を参照のこと。
PersistName	JMX 仕様	文字列	WebLogic MBeanMaker では、この属性は使用されない。ただし、PersistName を使用する Model MBean を MBean が拡張するような場合をサポートするために、WebLogic MBeanMaker では、この属性値を MBeanInfo クラスに追加する。デフォルト値または仮定される値はない。 **注意：** PersistName および Model MBean の詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
PersistPeriod	JMX 仕様	整数	OnTimer または NoMoreOftenThan の永続性ポリシーで使用される秒数を指定する。MBeanType または MBeanAttribute 要素でこの属性を指定しない場合には、値が 0 であると仮定される。 PersistPolicy が OnTimer に設定されている場合には、この秒数が経過したら属性が永続化される。一方、PersistPolicy が NoMoreOftenThan に設定されている場合には、指定された秒数以上の間隔を置いて永続化が行われるように制限される。 **注意：** MBeanType 要素内で指定された場合、この属性値は、個々の MBeanAttribute 下位要素内のあらゆる設定をオーバーライドする。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
PersistPolicy	JMX 仕様	Never /OnTimer /OnUpdate /NoMoreOftenThan	永続化がどのように行われるかを、以下のいずれかで指定する。 ■ Never：属性は決して格納されない。これは、非常に変わりやすいデータやセッションのコンテキスト内や実行期間中にしか意味を持たないデータの場合に役に立つ。 ■ OnTimer：属性は、当該 MBean 属性の永続性タイマーの設定時間 (PersistPeriod 属性で定義される) が経過するたびに格納される。 ■ OnUpdate：属性は更新のたびに格納される。 ■ NoMoreOftenThan：属性は、更新間隔が PersistPeriod より短くない限り、更新のたびに格納される。このメカニズムは、非常に変わりやすいデータの影響でパフォーマンスが下がることのないようにする上で役に立つ。 MBeanType または MBeanAttribute 要素でこの属性を指定しない場合には、値が Never であると仮定される。 **注意：** MBeanType 要素内で指定された場合、この属性値は、個々の MBeanAttribute 下位要素内のあらゆる設定をオーバーライドする。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
PresentationString	JMX 仕様	パス名	ユーザ インタフェースで項目の表示に使用できる情報を提供する単一の XML ドキュメントの完全修飾パス名。この XML ドキュメントは、プレゼンテーション ロジックに関係のある付加的なメタデータも提供する。PresentationString のフォーマットは、XML/JMX に準拠した任意の情報である。 **注意：** 現在、BEA では専用のフォーマットを定義していないが、独自に定義するより待つことをお勧めする。PresentationString 属性は、将来使用するためのものである。
ProtocolMap	JMX 仕様	パス名	Model MBean API を使用すれば、記述子の ProtocolMap フィールドを通じて、アプリケーションの Model MBean 属性を既存の管理データ モデルにマップできるようになる。属性の記述子の ProtocolMap フィールドには、Descriptor インタフェースを実装するクラスのインスタンスへの参照が記載されている必要がある。 **注意：** ProtocolMap および Model MBean の詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Readable	JMX 仕様	true/false	MBean の属性値を MBean API を通じて読み込めるかどうかを決定する。MBeanType または MBeanAttribute 要素でこの属性を指定しない場合には、値が true であると仮定される。 MBeanType 要素に指定すると、この値は個々の MBeanAttribute 下位要素のデフォルトと見なされる。
SetMethod	JMX 仕様	*文字列*	MBean タイプのデフォルト属性処理ロジックをオーバーライドし、現在の MBeanAttribute 下位要素に使用すべきセッター メソッドの名前を指定する。その値は、現在の MDF 内でセッター操作を定義している MBeanOperation 下位要素の Name に一致する必要がある。 この属性を指定しない場合には、MBean ではデフォルト ロジックを用いて、MBeanAttribute の値を設定する。 **注意：** この属性は Writeable 属性に左右される。つまり、Writable が false であれば、現在の MBeanAttribute 下位要素についてはセッターは呼び出されない。
Type	JMX 仕様	Java クラス名	この属性のデータ型の完全修飾クラス名。これに対応するクラスがクラスパスに存在していなければならない。この属性を指定しない場合には、値が java.lang.String と仮定される。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
`Validator`	BEA 拡張機能	Java メソッド名	`Validator` 属性には、バリデータのロジックを実装するメソッドの名前を指定する。このメソッドは、WebLogicMBeanMaker で生成されるクラスで宣言される。特定のバリデータを指定しない場合、WebLogic Server では、型の互換性と数値に関する範囲 (`Min` と `Max` で指定した範囲) を検証する基本的なチェックのみが行われる。 バリデータのインストールおよび使用は自動化されているので、バリデータ名を MDF に、バリデータ クラスをクラスパスにそれぞれ指定する以外のアクションは必要ない。
`Visibility`	JMX 仕様	整数 : `1-4`	MBean 属性の重要度を示す。ユーザ インタフェースではこの数値を用いて、特定のコンテキストで特定のユーザに当該 MBean 属性を表示するかどうかを決定する。この値が小さければ小さいほど、重要度は高くなる。`1` から `4` までの数値を指定できる。この属性を指定しない場合には、値が `1` であると仮定される。 ユーザの関心の高いアイテムの場合には、`Visibility` に低い数値を指定する。たとえば、WebLogic Server Administration Console では、`Visiblity` 値が `1` の MBean が左ペインのナビゲーション ツリーに表示される。

表 A-2MBeanAttribute 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Writeable	JMX 仕様	true/false	値が true の場合には、MBean API を通じて MBeanAttribute の値を設定できるようになる。 MBeanType または MBeanAttribute でこの属性を指定しない場合には、値が true であると仮定される。 MBeanType 要素に指定すると、この値は個々の MBeanAttribute 下位要素のデフォルトと見なされる。

MBeanNotification 下位要素

MBean タイプが発行できる通知タイプごとに、MBeanNotification 下位要素のインスタンス (管理イベントのブロードキャスト) を 1 つ記述する必要があります。MBeanNotification の形式は次のとおりです。

```
<MBeanNotification Name=string   optional_attributes  />
```

MBeanNotification 下位要素には Name 属性 (Java 通知のプログラム上の内部的な名前を指定するもの) および NotificationTypes のカンマ区切りのリストが含まれている必要があります (ユーザ インタフェースに表示される名前を指定するには、DisplayName 属性と LanguageMap 属性を使用します)。他の属性は省略可能です。

以下の例は、MBeanType 要素内の MBeanNotification 下位要素を簡略化して示したものです。

```
<MBeanType Name="MyMBean" Package="com.mycompany">
   <MBeanNotification Name="com.mycompany.myNotification"
```

```
    NotificationTypes="1.1.1.1.2.3.5" />
</MBeanType>
```

表 A-3 では、MBeanNotification 下位要素で使用可能な属性について説明します。「JMX 仕様 /BEA 拡張機能」カラムは、当該属性が JMX 仕様に対する BEA 拡張機能なのかどうかを示します。なお、BEA 拡張機能は他の J2EE Web サーバ上では動作しない可能性があることに注意してください。

表 A-3MBeanNotification 下位要素の属性

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Classname	JMX 仕様	文字列	MBeanNotification のクラス名 (Java Management eXtensions 1.0 仕様で定義されているもの)。たいていはデフォルト クラス名で問題ないが、必要であれば、この属性を用いて変更することができる。
Deprecated	BEA 拡張機能	true/false	当該 MBean 通知が非推奨扱いになっていることを示す。この情報は生成された Java ソースに表示されると共に、管理アプリケーションでの使用を想定して ModelMBeanInfo オブジェクトにも格納される。この属性を指定しない場合には、値が false であると仮定される。
Description	JMX 仕様	文字列	当該 MBean 通知に関連付けられる任意の文字列で、生成されたクラスの Javadoc など、さまざまな場所に現れる。デフォルト値または仮定される値はない。 **注意 :** ユーザ インタフェース内に表示される概要記述を指定するには、DisplayName、DisplayMessage、および PresentationString の各属性を使用する。

表 A-3MBeanNotification 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
DisplayMessage	BEA 拡張機能	文字列	ユーザ インタフェースに表示される MBean 通知の説明用メッセージ。デフォルト値または仮定される値はない。 DisplayMessage はツールのヒントやヘルプで用いられるパラグラフになる場合がある。MBean タイプに設定された DisplayMessage は、インスタンスの作成時に個々の MBean に対して別の値を指定しない限り、MBean インスタンスのデフォルトと見なされる。
DisplayName	JMX 仕様	文字列	ユーザ インタフェースに表示される MBean 通知タイプ識別用の名前。デフォルト値または仮定される値はない。 LanguageMap 属性を使用する場合には、DisplayName 値は LanguageMap のリソース バンドル内で名前を見つけるためのキーとして使われる。LanguageMap 属性を指定しない場合や、そのキーがリソース バンドルに存在しない場合には、DisplayName 値そのものがユーザ インタフェースに表示される。 「MessageID」を参照。

表 A-3MBeanNotification 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
LanguageMap	BEA 拡張機能	文字列	表示可能な文字列のマップを格納したリソース バンドルの完全修飾パス名を指定する。他の属性 (たとえば DisplayMessage や DisplayName など) は、LanguageMap 内の文字列を用いて、MBean 通知に関する情報を表示する。 この属性を指定しない場合には、他の属性 (たとえば DisplayMessage や DisplayName など) は、自分自身の値を表示する (それらの値をキーに使ってリソース バンドル内の適切な文字列を検索するわけではない)。
Listen	BEA 拡張機能	true/false	通知リスナのスタブを MBean 実装オブジェクト内に生成する。この属性を指定しない場合には、値が false であると仮定される。 **注意 :** リスナ スタブの詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-3MBeanNotification 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Log	JMX 仕様	true/false	値が true の場合、MBean の通知をログ ファイルに追加するよう指定する (LogFile 属性は情報の出力先となるファイルを指定する)。この属性を指定しない場合には、値が false であると仮定され、通知はログ ファイルに追加されない。 **注意：** MBean 通知およびログの詳細については、Java Management eXtensions 1.0 仕様を参照のこと。
LogFile	JMX 仕様	パス名	この MBean 通知タイプの通知が発生したときにメッセージが出力される既存の書き込み可能ファイルの完全修飾パス名。ロギングが行われるためには、Log 属性を true に設定しておく必要がある。この属性には、デフォルト値または仮定される値はない。 **注意：** MBean 通知およびログの詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-3MBeanNotification 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
MessageID	JMX 仕様	文字列	Java Management eXtensions 1.0 仕様に従ってクライアントサイド メッセージ リポジトリからメッセージを取得するためのキーを提供する。 MessageID と DisplayMessage のどちらか一方あるいは両方を用いて、通知 MBean タイプを記述することができる。この属性を指定しない場合には、メッセージ ID は利用できない。 **注意：** MessageID は、LanguageMap 属性で指定されるのと同じリソース バンドルを使用せず、通知 MBean 専用である。
Name	JMX 仕様	文字列	当該 Java 通知のプログラム上の内部的な名前を指定する必須属性
NotificationTypes	JMX 仕様	カンマで区切ったリスト	汎用的な通知オブジェクトを特徴付ける通知タイプ。このリストは、ドットで区切られた任意の数のコンポーネントをカンマで区切って列挙したもので、通知タイプの命名において任意のユーザ定義構造を作ることが可能である。 **注意：** 通知タイプの詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-3MBeanNotification 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
PresentationString	JMX 仕様	パス名	ユーザ インタフェースで項目の表示に使用できる情報を提供する単一の XML ドキュメントの完全修飾パス名。この XML ドキュメントは、プレゼンテーション ロジックに関係のある付加的なメタデータも提供する。PresentationString のフォーマットは、XML/JMX に準拠した任意の情報である。 **注意：** 現在、BEA では専用のフォーマットを定義していないが、独自に定義するより待つことをお勧めする。PresentationString 属性は、将来使用するためのものである。

表 A-3MBeanNotification 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Severity	JMX 仕様	整数または文字列	通知の重大度を示す。以下のいずれかの値を用いる必要がある。 ■ 0 または unknown ■ 1 または non-recoverable ■ 2、critical、または failure ■ 3、major、または severe ■ 4、minor、marginal、または error ■ 5 または warning ■ 6、normal、cleared、または info このタグの値は、数値でも、定義済みの文字列 (大文字 / 小文字の区別はない) でもかまわない。文字列を使用する場合、それは常に、MBeanInfo オブジェクトによって等価な数値に変換されて格納される。 この属性を指定しない場合には、unknown の重大度が通知される。

表 A-3MBeanNotification 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Visibility	JMX 仕様	整数 : 1-4	MBean 通知の重要度を示す。ユーザ インタフェースではこの数値を用いて、特定のコンテキストで特定のユーザに当該 MBean 通知を表示するかどうかを決定する。この値が小さければ小さいほど、重要度は高くなる。1 から 4 までの数値を指定できる。この属性を指定しない場合には、値が 1 であると仮定される。 ユーザの関心の高いアイテムの場合には、Visibility に低い数値を指定する。たとえば、WebLogic Server Administration Console では、Visiblity 値が 1 の MBean が左ペインのナビゲーション ツリーに表示される。

MBeanConstructor 下位要素

MBeanConstructor 下位要素は現在のところ、WebLogic MBeanMaker では使用されておらず、Java Management eXtensions 1.0 仕様に準拠し、上位互換性を保つためにサポートされているだけです。したがって、ここでは、MBeanConstructor 下位要素 (およびそれに関連する MBeanConstructorArg 下位要素) の属性の詳細については説明を省略します。

MBeanOperation 下位要素

MBean タイプでサポートされている操作 (メソッド) ごとに、MBeanOperation 下位要素のインスタンスを 1 つ記述する必要があります。MBeanOperation の形式は次のとおりです。

```
<MBeanOperation Name=string   optional_attributes >
     <MBeanOperationArg Name=string   optional_attributes />
</MBeanOperation>
```

MBeanOperation 下位要素には Name 属性が付いている必要があります。これは、その操作のプログラム上の内部的な名前を指定するものです (ユーザ インタフェースに表示される名前を指定するには、DisplayName 属性と LanguageMap 属性を使用します)。他の属性は省略可能です。

MBeanOperation 要素内には、その操作 (メソッド) で使用される引数ごとに MBeanOperationArg 下位要素のインスタンスを 1 つ記述する必要があります。MBeanOperationArg の形式は次のとおりです。

```
<MBeanOperationArg Name=string  optional_attributes  />
```

Name 属性には操作の名前を指定する必要があります。MBeanOperationArg のオプション属性は Type だけで、これは、特定タイプの Java 属性の動作を指定する Java クラス名を与えるものです。この属性を指定しない場合には、値が java.lang.String と仮定されます。

次の例は、MBeanType 要素内の MBeanOperation 下位要素と MBeanOperationArg 下位要素を簡略化して示したものです。

```
<MBeanType Name="MyMBean" Package="com.mycompany">

   <MBeanOperation
    Name= "findParserSelectMBeanByKey"
    ReturnType="XMLParserSelectRegistryEntryMBean"
    Description="Given a public ID, system ID, or root element tag, returns the
object name of the corresponding XMLParserSelectRegistryEntryMBean."
   >
      <MBeanOperationArg Name="publicID" Type="java.lang.String"/>
      <MBeanOperationArg Name="systemID" Type="java.lang.String"/>
```

```
    <MBeanOperationArg Name="rootTag" Type="java.lang.String"/>
  </MBeanOperation>
</MBeanType>
```

表 A-4 では、MBeanOperation 下位要素で使用可能な属性について説明します。「JMX 仕様 /BEA 拡張機能」カラムは、当該属性が JMX 仕様に対する BEA 拡張機能なのかどうかを示します。なお、BEA 拡張機能は他の J2EE Web サーバ上では動作しない可能性があることに注意してください。

表 A-4MBeanOperation 下位要素の属性

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
CurrencyTimeLimit	JMX 仕様	整数	キャッシュされた値が有効期限内にあるとみなされる許容時間 (秒数)。この時間が経過した後、値にアクセスしようとすると、再計算がトリガされる。 MBeanType 要素に指定すると、この値は MBean タイプのデフォルトと見なされる。同じ属性を MBean の MBeanAttribute または MBeanOperation 下位要素に設定すると、個々の MBean に関してオーバーライドできる。
Deprecated	BEA 拡張機能	true/false	当該 MBean 操作が非推奨扱いになっていることを示す。この情報は生成された Java ソースに表示されると共に、管理アプリケーションでの使用を想定して ModelMBeanInfo オブジェクトにも格納される。この属性を指定しない場合には、値が false であると仮定される。

表 A-4MBeanOperation 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
`Description`	JMX 仕様	文字列	当該 MBean 操作に関連付けられる任意の文字列で、生成されたクラスの Javadoc など、さまざまな場所に現れる。デフォルト値または仮定される値はない。 **注意：** ユーザ インタフェース内に表示される概要記述を指定するには、`DisplayName`、`DisplayMessage`、および `PresentationString` の各属性を使用する。
`DisplayMessage`	BEA 拡張機能	文字列	ユーザ インタフェースに表示される MBean 操作の説明用メッセージ。デフォルト値または仮定される値はない。 `DisplayMessage` はツールのヒントやヘルプで用いられるパラグラフになる場合がある。MBean タイプに設定された `DisplayMessage` は、インスタンスの作成時に個々の MBean に対して別の値を指定しない限り、MBean インスタンスのデフォルトと見なされる。

表 A-4MBeanOperation 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
DisplayName	JMX 仕様	文字列	ユーザ インタフェースに表示される MBean 操作識別用の名前。デフォルト値または仮定される値はない。 LanguageMap 属性を使用する場合には、DisplayName 値は LanguageMap のリソース バンドル内で名前を見つけるためのキーとして使われる。LanguageMap 属性を指定しない場合や、そのキーがリソース バンドルに存在しない場合には、DisplayName 値そのものがユーザ インタフェースに表示される。 「MessageID」を参照。
Impact	JMX 仕様		MBean 操作のうち、MBean で表される管理対象エンティティに対して操作が及ぼす影響を知らせる部分。 **注意：** 詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-4MBeanOperation 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
LanguageMap	BEA 拡張機能	文字列	表示可能な文字列のマップを格納したリソース バンドルの完全修飾パス名を指定する。他の属性 (たとえば DisplayMessage や DisplayName など) は、LanguageMap 内の文字列を用いて、MBean 操作に関する情報を表示する。 この属性を指定しない場合には、他の属性 (たとえば DisplayMessage や DisplayName など) は、自分自身の値を表示する (それらの値をキーに使ってリソース バンドル内の適切な文字列を検索するわけではない)。
Listen	BEA 拡張機能	true/false	通知リスナのスタブを MBean 実装オブジェクト内に生成する。この属性を指定しない場合には、値が false であると仮定される。 **注意 :** リスナ スタブの詳細については、Java Management eXtensions 1.0 仕様を参照のこと。

表 A-4MBeanOperation 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
MessageID	JMX 仕様	文字列	Java Management eXtensions 1.0 仕様に従ってクライアントサイド メッセージ リポジトリからメッセージを取得するためのキーを提供する。 MessageID と DisplayMessage のどちらか一方あるいは両方を用いて、通知 MBean タイプを記述することができる。この属性を指定しない場合には、メッセージ ID は利用できない。 **注意：** MessageID は、LanguageMap 属性で指定されるのと同じリソース バンドルを使用せず、通知 MBean 専用である。
Name	JMX 仕様	文字列	当該 MBean 操作のプログラム上の内部的な名前を指定する必須属性。

表 A-4MBeanOperation 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
PresentationString	JMX 仕様	パス名	ユーザ インタフェースで項目の表示に使用できる情報を提供する単一の XML ドキュメントの完全修飾パス名。この XML ドキュメントは、プレゼンテーション ロジックに関係のある付加的なメタデータも提供する。PresentationString のフォーマットは、XML/JMX に準拠した任意の情報である。 **注意：** 現在、BEA では専用のフォーマットを定義していないが、独自に定義するより待つことをお勧めする。PresentationString 属性は、将来使用するためのものである。
ReturnType	JMX 仕様	文字列	当該操作から返される Java オブジェクトの完全修飾クラス名を表す文字列
ReturnTypeDescription	JMX 仕様	文字列	当該操作から返される Java オブジェクトについて説明するテキスト。これは、Javadoc またはグラフィカル ユーザ インタフェースで使用することができる。

表 A-4MBeanOperation 下位要素の属性 （続き）

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
Visibility	JMX 仕様	整数 : 1-4	MBean 操作の重要度を示す。ユーザ インタフェースではこの数値を用いて、特定のコンテキストで特定のユーザに当該 MBean 操作を表示するかどうかを決定する。この値が小さければ小さいほど、重要度は高くなる。1 から 4 までの数値を指定できる。この属性を指定しない場合には、値が 1 であると仮定される。 ユーザの関心の高いアイテムの場合には、Visibility に低い数値を指定する。たとえば、WebLogic Server Administration Console では、Visiblity 値が 1 の MBean が左ペインのナビゲーション ツリーに表示される。

表 A-5 では、MBeanOperationArg 下位要素で使用可能な属性について説明します。「JMX 仕様 /BEA 拡張機能」カラムは、当該属性が JMX 仕様に対する BEA 拡張機能なのかどうかを示します。なお、BEA 拡張機能は他の J2EE Web サーバ上では動作しない可能性があることに注意してください。

表 A-5MBeanOperationArg 下位要素の属性

属性	JMX 仕様 /BEA 拡張機能	指定可能な値	説明
`Description`	JMX 仕様	文字列	当該 MBean 操作の引数に関連付けられる任意の文字列で、生成されたクラスの Javadoc など、さまざまな場所に現れる。デフォルト値または仮定される値はない。
`InterfaceType`	BEA 拡張機能	文字列	WebLogic MBeanMaker で生成される MBean インタフェースの代わりに使用すべきインタフェースのクラス名。これは、現時点では無視されるが、今後の拡張で使用される可能性はある。
`Name`	JMX 仕様	文字列	当該引数の名前を指定する必須属性。
`Type`	JMX 仕様	文字列	当該 MBean 操作の引数のデータ型。この属性を指定しない場合には、値が `java.lang.String` と仮定される。

例：有効な整形式 MBean 定義ファイル (MDF)

コード リスト A-1 とコード リスト A-2 に、この付録で説明した属性の多くを使用した MBean 定義ファイル (MDF) の例を示します。コード リスト A-1 には、述語を管理し、述語とそれらの引数に関するデータを読み取る MBean タイプを生成するための MDF を示します。コード リスト A-2 には、WebLogic (デフォルト) 認証プロバイダの MBean タイプを生成するための MDF を示します。

コード リスト A-1　PredicateEditor.xml

```
<?xml version="1.0" ?>
<!DOCTYPE MBeanType SYSTEM "commo.dtd">

<MBeanType
 Name = "PredicateEditor"
 Package = "weblogic.security.providers.authorization"
 Implements = "weblogic.security.providers.authorization.PredicateReader"
 PersistPolicy = "OnUpdate"
 Abstract = "false"
 Description = "This MBean manages predicates and reads data about predicates and
their arguments.&lt;p&gt;"
>

   <MBeanOperation
    Name = "registerPredicate"
    ReturnType = "void"
    Description = "Registers a new predicate with the specified class name."
   >

     <MBeanOperationArg
      Name = "predicateClassName"
      Type = "java.lang.String"
      Description = "The name of the Java class that implements the predicate."
     />

     <MBeanException>weblogic.management.utils.InvalidPredicateException</MBean
Exception>

     <MBeanException>weblogic.management.utils.AlreadyExistsException</MBeanExc
eption>

   </MBeanOperation>

   <MBeanOperation
    Name = "unregisterPredicate"
    ReturnType = "void"
    Description = "Unregisters the currently registered predicate."
   >

     <MBeanOperationArg
      Name = "predicateClassName"
      Type = "java.lang.String"
      Description = "The name of the Java class that implements predicate to be
unregistered."
     />
```

```
     <MBeanException>weblogic.management.utils.NotFoundException</MBeanExceptio
n>

   </MBeanOperation>

</MBeanType>
```

コード リスト A-2　DefaultAuthorizer.xml

```
<?xml version="1.0" ?>
<!DOCTYPE MBeanType SYSTEM "commo.dtd">

<MBeanType
 Name = "DefaultAuthorizer"
 DisplayName = "DefaultAuthorizer"
 Package = "weblogic.security.providers.authorization"
 Extends = "weblogic.management.security.authorization.DeployableAuthorizer"
 Implements = "weblogic.management.security.authorization.PolicyEditor,
weblogic.security.providers.authorization.PredicateEditor"
 PersistPolicy = "OnUpdate"
 Description = "This MBean represents configuration attributes for the WebLogic
Authorization provider. &lt;p&gt;"
>

   <MBeanAttribute
    Name = "ProviderClassName"
    Type = "java.lang.String"
    Writeable = "false"
    Default =
"&quot;weblogic.security.providers.authorization.DefaultAuthorizationProviderIm
pl&quot;"
    Description = "The name of the Java class used to load the WebLogic
Authorization provider."
   />

   <MBeanAttribute
    Name = "Description"
    Type = "java.lang.String"
    Writeable = "false"
    Default = "&quot;Weblogic Default Authorization Provider&quot;"
    Description = "A short description of the WebLogic Authorization provider."
   />

   <MBeanAttribute
    Name = "Version"
    Type = "java.lang.String"
    Writeable = "false"
```

```
    Default = "&quot;1.0&quot;"
    Description = "The version of the WebLogic Authorization provider."
   />

</MBeanType>
```

索引

A

C

E

G

I

J

L

M

P

R

S

T

U

W

あ

い

え

か

き

さ

し

せ

そ

た

て

と

に

ひ

ふ

へ

ほ

ま

め

ゆ

よ

り

れ

ろ