Sun WBEM SDK 開発ガイド

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part Number 816-0093-10
2001 年 5 月

原典: *Sun WBEM SDK Developer's Guide*

Part No: 806-6831-10

Revision A

目次

表

図

例

はじめに

本書『*Sun WBEM SDK* 開発ガイド』では、Sun WBEM ソフトウェア開発者ツールキットについて説明します。このツールキットを使うことによってソフトウェア開発者は、WBEM 対応のオブジェクトを管理する標準に準拠したアプリケーションを開発することができます。このソフトウェアは、プロバイダ (データにアクセスするために管理対象オブジェクトと通信するプログラム) の作成にも使用できます。

対象読者

このマニュアルは、次のような開発者を対象としています。

- システムとネットワークアプリケーションの開発者

  このマニュアルは、CIM のクラスとインスタンスに格納される情報を管理するアプリケーションを開発するプログラマに役立ちます。アプリケーションプログラマは、通常、Sun WBEM API を使用して定義済み CIM クラスおよび CIM インスタンスのプロパティの取得や設定を行います。

- システム設計技術者

  リソース (プロセッサ、メモリー、ルーターなどの管理可能デバイス) を提供する設計技術者は、標準の CIM 形式のデバイス情報を CIM Object Manager に伝える必要があります。この伝達は、一般にプロバイダと呼ばれるソフトウェアによって行われます。システム設計技術者は、WBEM API を使用してクラス、インスタンス、およびプロパティを作成します。

システム設計技術者は、管理対象リソースの新しいグループ (CIM クラス) を記述するクラス設計者、および CIM クラスの集まり (スキーマ) を記述するスキーマ設計者と共同で作業を行う場合もあります。スキーマは、Microsoft Windows 32 ビットオペレーティング環境や Solaris オペレーティング環境のような特定のシステム環境における管理対象オブジェクトを記述します。

お読みになる前に

このマニュアルは、管理アプリケーションを作成する場合に Sun WBEM SDK コンポーネントとツールをどのように使用するかについて説明しています。

このマニュアルは、読者に次の知識があることを前提としています。

- オブジェクト指向プログラミングの概念
- Java プログラミング
- CIM (Common Information Model) の概念についての十分な知識

知識が不十分な場合には、次の書籍を参考にすることをお勧めします。

- 『*Java™ How to Program*』

  H. M. Deitel、P. J. Deitel 著、Prentice Hall 発行、ISBN 0–13–263401–5
- 『*The Java Class Libraries*』第 2 版、第 1 巻、Patrick Chan、Rosanna Lee、Douglas Kramer、Addison-Wesley 著、ISBN 0–201–31002–3
- 『*CIM Tutorial*』、Distributed Management Task Force 提供

次に、WBEM 技術に携わる場合に有用な Web サイトを示します。

- DMTF (Distributed Management Task Force)

  このサイト (`http://www.dmtf.org`) には、CIM の最新の開発情報、各種の作業グループについての情報、CIM スキーマの拡張についての問い合わせ方法などが掲載されています。
- Rational Software

このサイト (`http://www.rational.com/uml`) では、Unified Modeling Language (UML) と Rose CASE ツールの関連文書を入手できます。

内容の紹介

第 1 章では、WBEM (Web-Based Enterprise Management)、CIM (Common Information Model)、Sun WBEM SDK、Solaris WBEM Services について紹介しています。

第 2 章では、CIM WorkShop を使用して CIM のクラス、インスタンス、メソッド、およびプロパティを処理する方法について説明しています。

第 3 章では、クライアント API の概要を説明し、クライアント API を使用して CIM オブジェクトの作成と処理を行う例を示しています。

第 4 章では、クライアント API を使用してクライアントアプリケーションを作成する方法について説明しています。

第 5 章では、プロバイダ API の概要を説明し、プロバイダ (管理対象オブジェクトと CIM Object Manager 間の仲介となるクラス) の作成方法について説明しています。

第 6 章では、最初に CIM イベントモデルとは何かを説明し、プロバイダがどのようにして CIM イベントを生成し、アプリケーションがどのようにして CIM イベント発生の通知を予約するかを説明しています。

第 7 章では、Sun WBEM SDK に付属しているコード例を実行する方法について説明しています。

第 8 章では、Sun WBEM SDK API のエラーメッセージについて説明しています。

付録 A では、CIM の一般的な用語と概念について説明しています。

用語集では、このマニュアルで使用されている用語の意味について説明しています。

関連情報

CIM (Common Information Model) の概念や、WBEM (Web-based Enterprise Management) サービスを Solaris™ 環境で管理する方法については、『*Solaris WBEM Services* の管理』を参照してください。

さらに、WBEM のアプリケーションプログラミングインタフェースについては、`/usr/sadm/lib/wbem/doc/index.html` の Javadoc リファレンスページを、CIM クラスや Solaris スキーマクラスについては、`/usr/sadm/lib/wbem/doc/mofhtml` の Javadoc リファレンスページをそれぞれ参照してください。

マニュアルの注文方法

Sun™ Software Shop プログラムを利用して、米国 Sun Microsystems™, Inc. (以降、Sun とします) のマニュアルまたは AnswerBook2 CD をご注文いただけます。

マニュアルのリストと注文方法については、`http://www.sun.com/software/shop/` を参照してください。

Sun のオンライン文書へのアクセス

`http://docs.sun.com` では、Sun が提供しているオンラインマニュアルを参照することができます。`docs.sun.com` にあるマニュアルタイトルや、特定の主題などをキーワードとして検索することもできます。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、またはコード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力とは区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、または強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を越える場合、バックスラッシュは継続を示します。	sun% **grep '^#define ** **XV_VERSION_STRING'**

ただし AnswerBook2™ では、ユーザーが入力する文字と画面上のコンピュータ出力は区別して表示されません。

コード例は次のように表示されます。

- C シェルプロンプト

```
system% command y|n [filename]
```

- Bourne シェルおよび Korn シェルのプロンプト

```
system$ command y|n [filename]
```

- スーパーユーザーのプロンプト

```
system# command y|n [filename]
```

[ ] は省略可能な項目を示します。上記の場合、*filename* は省略してもよいことを示します。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

一般規則

- このマニュアルでは、英語環境での画面イメージを使っています。このため、実際に日本語環境で表示される画面イメージとこのマニュアルで使っている画面イメージが異なる場合があります。本文中で画面イメージを説明する場合には、日本語のメニュー、ボタン名などの項目名と英語の項目名が適宜、併記されています。

WBEM の概要

この章では、WBEM (Web-Based Enterprise Management) について詳しく説明します。内容は次のとおりです。

- WBEM について
- CIM (Common Information Model)
- MOF (Managed Object Format)
- CIM と Solaris

WBEM について

WBEM (Web-Based Enterprise Management) は、先駆的な総合技術です。WBEM には、インターネット技術を使用してシステム、ネットワーク、ユーザー、およびアプリケーションを管理するための標準規格が含まれています。技術面では、管理アプリケーションに対し、ベンダー、プロトコル、オペレーティングシステム、管理基準などに依存することなく管理データを共有する方法を提供しています。WBEM 方式に基づいて管理アプリケーションを開発すると、連携して動作する製品を低コストで簡単に提供できます。

コンピュータ業界とテレコミュニケーション業界の企業を代表するグループの 1 つである DMTF (Distributed Management Task Force) は、デスクトップ環境、企業規模のシステム、およびインターネットを管理するための標準規格の開発と普及では主導的な立場にあります。DMTF の目的は、さまざまなプラットフォームおよびプロトコルに渡ってネットワークを管理する統合的な手法を開発し、費用効率の高い相

互運用性に優れた製品を提供することにあります。DMTF の提唱とその現況については、このグループの Web サイト `http://www.dmtf.org` を参照してください。

CIM (Common Information Model)

CIM (Common Information Model) は、システムとネットワークを管理する手法です。CIM は、ネットワーク環境の各部の分類と定義を行い、それらの統合方法を表現するための概念的な共通の枠組みを提供します。この概念は、技術の実装には依存せず、あらゆる管理領域に適用できます。

CIM の用語

CIM (Common Information Model) は、このモデル固有の用語と、オブジェクト指向プログラミングの用語を使用しています。CIM の用語と意味については、付録 A を参照してください。CIM 独自の意味を持つ用語については、用語集を参照してください。

CIM の構造

CIM (Common Information Model) は、情報を一般的なものから特定のものへと分類します。Solaris の環境などの特定の情報は、このモデルを拡張して記述されています。CIM は、次に示す 3 つの情報層から構成されます。

- コアモデル – プラットフォームに依存しない、CIM のサブセット
- 共通モデル – ネットワーク管理の特定の領域に関連するエンティティ (システム、デバイス、アプリケーションなど) の概念、機能性、および表示方法を視覚的に表す情報モデル
- エクステンション (拡張)– CIM スキーマをサポートし、限定されたプラットフォーム、プロトコル、または企業独自のものを表す情報モデル

コアモデルと 共通モデルを、総称して CIM スキーマと呼びます。

コアモデル

コアモデルは、管理環境の基本となる一般的な前提事項を提供します (たとえば、要求された特定のデータはある場所に格納され、要求元のアプリケーションまたはユーザーに配付されなければならないなど)。これらの前提事項は、管理環境の基盤を概念的に形成する、クラスと関連のセットとして示されます。コアモデルは、管理環境の特定の側面を表現するスキーマに一貫性を持たせます。

コアモデルは、管理対象システムを表現し、共通モデルを拡張する方法を決定するための手掛かりとして使用できる、クラス、関連、およびプロパティのセットをアプリケーション開発者に提供します。コアモデルは、その他の管理環境をモデル化する概念的な枠組みを確立します。

コアモデルは、 共通モデルとエクステンション (拡張) により、システム、アプリケーション、ネットワーク、デバイスなどのネットワーク機能に関する特定の情報を拡張するためのクラスと関連を提供します。コアモデルのシステム的な側面、および関連するクラスと関連については、209ページの「コアモデルの概念」を参照してください。

共通モデル

共通モデルで示されるネットワーク管理の領域は、特定の技術や実装には依存しない管理アプリケーションの開発基盤を提供します。このモデルは、指定された 5 つの技術別スキーマ、Systems、Devices、Applications、Networks、および Physical に、拡張用の基底クラスセットを提供します。

CIM エクステンション (拡張)

拡張スキーマは、このモデルに特定の技術を関連づけるために CIM に組み込まれます。CIM を拡張すると、多数のユーザーと管理者が Solaris などの特定のオペレーティング環境を使用できるようになります。拡張スキーマのクラスを使用して、ソフトウェア開発者は拡張された技術を管理するアプリケーションを開発することができます。

MOF (Managed Object Format)

MOF は、CIM (Common Information Model) の要素の定義に使用される標準言語です。MOF 言語は、CIM のクラスとインスタンスを定義する構文を指定します。MOF を使用すると、開発者や管理者は CIM Repository を簡単にかつ短時間で変更できます。MOF についての詳細は、DMTF Web ページ `http://www.dmtf.org` を参照してください。

MOF は Java に変換できるので、MOF で開発されたアプリケーションは Java がサポートしていれば、どのようなシステムあるいは環境でも動作します。

MOF の構文

プログラマは、CIM API を使用し、MOF で開発された CIM オブジェクトを Java クラスとして表現できます。CIM Object Manager は、これらの CIM オブジェクトを調べて、CIM 2.1 仕様に準拠するようにします。場合によっては、 MOF ファイル内で、CIM 仕様を厳守していなくても構文上は正しいものは表示されることがあります。そのような MOF ファイルがコンパイルされると、CIM Object Manager はエラーメッセージを返します。

たとえば、MOF ファイルの修飾子定義でスコープを指定すると、CIM Object Manager はコンパイルエラーを返します。これは、CIM 修飾子型の定義内でしかスコープは指定できないためです。CIM Qualifier は、CIM Qualifier Type で指定されたスコープを変更することはできません。

スキーマ MOF ファイル

Solaris WBEM Services をインストールすると、CIM スキーマと Solaris スキーマを形成する MOF ファイルがディレクトリ `/usr/sadm/mof` に置かれます。これらのファイルは、CIM Object Manager の起動時に自動的にコンパイルされ実行されます。

ファイル名の中に CIM を含む CIM スキーマファイルが、標準の CIM オブジェクトになります。CIM スキーマの各構成については、CIM Specification のバージョン 2.1 (`http://dmtf.org/spec/cims.html` で入手可) を参照してください。

Solaris スキーマは、標準の CIM スキーマを拡張することによって Solaris オブジェクトを記述したものです。Solaris スキーマを構成する MOF ファイルは、ファイル

名にある Solaris 接頭辞を使用しますが、使用しない場合は CIM スキーマ MOF ファイルと同じファイルの命名規則に従います。Solaris スキーマを構成する MOF ファイルは、任意のテキストエディタで表示できます。

CIM と Solaris

Sun Microsystems, Inc. は、CIM の原理とクラスを Solaris オペレーティング環境用に拡張しています。Sun の製品は、あらゆる Java 対応プラットフォームで動作するように Java で開発されており、2 つの製品、Sun WBEM SDK と Solaris WBEM Services から構成されています。

Sun WBEM SDK

Sun WBEM Software Development Kit (SDK) には、WBEM 対応のあらゆる管理デバイスと通信することができる管理アプリケーションを作成するためのコンポーネントが含まれています。このツールキットを使用すると、プロバイダ (データにアクセスするために管理対象オブジェクトと通信を行うプログラム) も作成できます。Sun WBEM SDK を使用して開発された管理アプリケーションはすべて、Java プラットフォームで動作します。

Sun WBEM SDK は、任意の Java 環境にインストールし、実行することができます。さらに、Sun WBEM SDK は、スタンドアロンアプリケーションとして使用することも、Solaris WBEM Services と共に使用することもできます。

Solaris WBEM Services

Solaris WBEM Services は、ルーティングサービスとセキュリティサービスを提供します。CIM Object Manager は、コンポーネント間のオブジェクトとイベントに関するデータのルーティングを行います。Sun WBEM User Manager は、GUI で特定の作業領域にユーザーのアクセス権を設定できるアプリケーションです。Solaris WBEM Services や Solaris WBEM のコンポーネントについては、『*Solaris WBEM Services* の管理』を参照してください。

第 2 章

CIM WorkShop

この章では、CIM WorkShop を使用して、作成したクラスとインスタンスに新しいプロパティ、メソッド、修飾子を追加する方法、およびそれらの修飾子のスコープとフレーバを設定する方法について説明します。この章の内容は、次のとおりです。

- CIM WorkShop について
- CIM WorkShop の起動
- CIM WorkShop 内での移動
- クラス特性の表示
- ネームスペース内での作業
- クラスの処理
- クラスの追加
- クラスとクラス属性の削除
- インスタンスの処理
- メソッドの呼び出し
- リファレンス:「CIM WorkShop」ウィンドウとダイアログボックス

CIM WorkShop について

CIM WorkShop は、クラスとインスタンスの表示と作成を行うことができる GUI を提供します。CIM WorkShop では、次の作業を実行できます。

- ネームスペースの表示と選択
- ネームスペースの追加
- クラスの表示と作成
- 新しいクラスへのプロパティ、修飾子、およびメソッドの追加
- インスタンスの表示と作成
- インスタンス値の表示と変更

注 - CIM スキーマクラスまたは Solaris スキーマクラスの既存のプロパティ、メソッド、または修飾子の変更 (編集) は、CIM ガイドラインによって禁止されています。しかし、クラスとクラスのインスタンスを新たに作成することはできます。新しいクラスまたはインスタンスを作成すると、プロパティ、メソッド、または修飾子を、追加または削除できます。また、新しいクラス、インスタンス、プロパティ、またはメソッドに対して作成した新しい修飾子の値 (スコープやフレーバ) も変更できます。継承されたプロパティ、メソッド、および修飾子の値は変更できません。

CIM WorkShop の起動

CIM WorkShop は Sun WBEM SDK に含まれています。

CIM WorkShop を実行するには、CIM Object Manager がサーバー側にインストールされている必要があります。Solaris オペレーティング環境に Solaris WBEM Services をインストールしているときは、CIM Object Manager はローカルホストで動作します。Sun WBEM SDK だけをインストールする場合は、CIM Object Manager がすでに起動しているホストを指定する必要があります。この情報は、CIM WorkShop の起動時に表示される「ログイン (Login)」ダイアログボックスの「ホスト (Host)」フィールドに入力できます。CIM WorkShop の各ダイアログボックスとフィールドについての詳細は、52ページの「リファレンス:「CIM WorkShop」ウィンドウとダイアログ」を参照してください。

▼ CIM WorkShop を起動する方法

1. **CIM WorkShop** を起動します。
   - システムプロンプトで次のコマンドを入力します。

```
% /usr/sadm/bin/cimworkshop
```

「CIM WorkShop」ウィンドウが表示され、続いて「ログイン (Login)」ダイアログボックスが表示されます。「ログイン (Login)」ダイアログボックスには、CIM WorkShop がインストールされているホストコンピュータの名前、デフォルトネームスペースのパス `root\cimv2` が表示されます。「ログイン (Login)」ダイアログボックスの右側には、コンテキストヘルプとしてこのダイアログボックスの指定方法が表示されます。フィールドをクリックすると、コンテキストヘルプの内容が変わり、そのフィールドへの情報の入力方法とフィールドの意味が表示されます。

2. **CIM WorkShop** の「ログイン **(Login)**」ダイアログボックスで、次の操作を行います。
   - 「ホスト名 (Host Name)」フィールドで、CIM Object Manager を実行しているホスト名を入力します。

注 - デフォルトでは、CIM WorkShop はローカルホストのデフォルトネームスペース `root\cimv2` で動作している CIM Object Manager に接続します。Solaris オペレーティング環境または Microsoft Windows 環境で WBEM SDK の一部として CIM WorkShop を起動する場合には、すでに CIM Object Manager を実行しているホスト名を指定する必要があります。

   - 「ネームスペース (Namespace)」フィールド内をクリックして使用するネームスペース名を入力するか、あるいはデフォルトのネームスペース名のままにしておきます。
   - 「ユーザー名 (User Name)」フィールドで、システム権限とネットワーキング権限を実行する場合に通常使用しているユーザー名を入力します。
   - 「パスワード (Password)」フィールドで、システム権限とネットワーキング権限を実行する場合に通常使用しているパスワードを入力します。

注 - ユーザー名とパスワードを指定しない場合は、デフォルトのユーザーアカウント *guest* を使用してログインできます。この場合許可されるのは読み取り権だけです。CIM Object Manager の管理者は、ユーザー名とパスワードに対応した書き込み権を設定できます。

- デフォルトでは、CIM WorkShop は RMI プロトコルを使って、ローカルホスト (デフォルトのネームスペースは `root\cimv2`) 上の CIM Object Manager に接続します。Desktop Management Task Force の標準 XML/HTTP プロトコルを使って CIM Object Manager と通信したい場合は、HTTP を選択できます。

3. 「了解 **(OK)**」をクリックします。

   クラス継承ツリー内のクラスが列挙されていることを示すメッセージが表示されます。「CIM WorkShop」ウィンドウの左側には、CIM クラスが表示されます。

CIM WorkShop 内での移動

CIM WorkShop を起動すると、「CIM WorkShop」ウィンドウの左側に CIM スキーマのクラスが階層的に表示されます。このクラス配置を、クラス継承ツリーと呼びます。クラスを選択すると、そのクラスに対応するプロパティが、ウィンドウの右側に表示されます。 次の図では、クラス `solaris_computersystem` のプロパティが 「CIM WorkShop」ウィンドウの右側に表示されています。

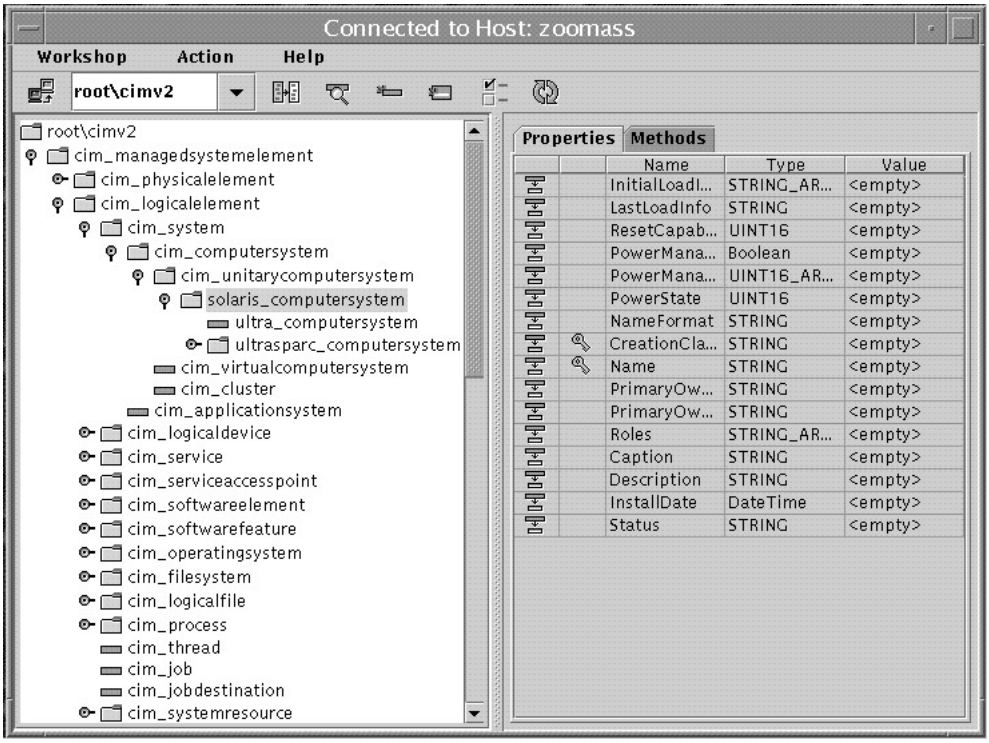

図 2–1　「CIM WorkShop」ウィンドウ内のクラス継承ツリー

「CIM WorkShop」ウィンドウのツールバー、メニュー、レイアウトについての詳細は、52ページの「リファレンス:「CIM WorkShop」ウィンドウとダイアログ」を参照してください。

クラス継承ツリーの表示

サブクラスを持つ各クラスは、2 つのアイコン (フォルダアイコンとイネーブラアイコン) によって示されます。イネーブラアイコンは、フォルダアイコンの左にある鍵の形をしたアイコンです。

フォルダアイコンは、そのクラスがそのサブクラスのコンテナになっていることを示します。イネーブラアイコンを使用すると、操作が簡単になります。

イネーブラアイコンが水平に表示されている場合、そのクラスフォルダは閉じており、サブクラスが入っています。水平のイネーブラアイコンをクリックすると、クラスフォルダが開かれ、サブクラスが表示されます。そして、イネーブラアイコンは垂直に表示され、クラスフォルダが開かれていることを示します。

▼ クラスの内容を表示する方法

- クラスに含まれているものを表示するには、そのクラスのイネーブラアイコンをクリックします。

▼ クラスのプロパティとメソッドを表示する方法

- クラスのクラスフォルダアイコンをクリックします。

  「CIM WorkShop」ウィンドウの右側のフレームに、クラスのプロパティとメソッドが表示されます。

クラスの検索

CIM WorkShop では、特定のクラスをすばやく検索できます。

▼ クラスを検索する方法

1. ツールバーで、「クラスを検索 **(Find Class)**」アイコンをクリックします。
2. 「クラスを検索 **(Find Class)**」ダイアログボックスで、検索したいクラス名を入力して「了解 **(OK)**」をクリックします。

   指定したクラスが見つかると、「CIM WorkShop」ウィンドウの右側のフレームにその詳細が表示されます。

クラス特性の表示

クラス継承ツリーでフォルダアイコンをクリックしてクラスを選択すると、「CIM WorkShop」ウィンドウの右側にクラスのプロパティとメソッドを示す 2 つのタブが表示されます。

クラスの選択

クラス継承ツリー内では、サブクラスが入ったクラスはフォルダアイコンで示されます。サブクラスを含まないクラスは、紫色の四角形で示されます。クラスを選択するには、クラス継承ツリー内でクラスのフォルダまたはページアイコンをクリックします。

クラスプロパティの表示

デフォルトでは、「CIM WorkShop」ウィンドウが表示される時に、「CIM WorkShop」ウィンドウの右側に「プロパティ (Properties)」タブが表示されます。「CIM WorkShop」ウィンドウの左側でクラス継承ツリーからクラスを選択すると、そのクラスのすべてのプロパティが「プロパティ (Properties)」タブに表示されます。継承されたプロパティは、紫色の四角形、白い四角形、および白い四角形を指す黒い矢印から構成されるアイコンによって示されます。キー修飾子が割り当てられたプロパティは、金色の鍵の形をしたアイコンによって示されます。「プロパティ (Properties)」タブでプロパティがどのように表示されるかについては、54ページの「「プロパティ (Properties)」タブ」を参照してください。

クラスメソッドの表示

クラス継承ツリーでクラスを選択した後、「メソッド (Methods)」タブをクリックしてそのクラスに関連付けられたメソッドを表示できます。「メソッド (Methods)」タブでメソッドがどのように表示されるかについては、55ページの「「メソッド (Methods)」タブ」を参照してください。

修飾子の表示

CIM では、修飾子は、クラス、インスタンス、プロパティ、およびメソッドの属性を意味します。CIM WorkShop では、クラス、プロパティ、またはメソッドにおいてマウスの右ボタンをクリックしてポップアップメニューを表示し、「修飾子 (Qualifiers)」を選ぶことにより修飾子を表示できます。「修飾子 (Qualifiers)」をクリックすると、「修飾子 (Qualifiers)」ダイアログボックスが表示されます。「修飾子 (Qualifiers)」ダイアログボックスにおける修飾子情報についての詳細は、59ページの「「修飾子 (Qualifiers)」ダイアログボックス」を参照してください。

修飾子のスコープの表示

「修飾子 (Qualifiers)」ダイアログボックスの「スコープ (Scope)」ボタンをクリックすると、「スコープ (Scope)」ダイアログボックスが表示されます。スコープダイアログボックスでは、修飾子のスコープを見ることができます。「スコープ (Scope)」ダイアログボックスについての詳細は、52ページの「リファレンス:「CIM WorkShop」ウィンドウとダイアログ」を参照してください。

修飾子フレーバの表示

「修飾子 (Qualifiers)」ダイアログボックスで「フレーバ (Flavor)」ボタンをクリックすると、「フレーバ (Flavors)」ダイアログボックスが表示されます。「フレーバ (Flavors)」ダイアログボックスでは、修飾子のフレーバを表示できます。「フレーバ (Flavors)」ダイアログボックスについての詳細は、60ページの「「フレーバ (Flavors)」ダイアログボックス」を参照してください。

ネームスペース内での作業

ネームスペースは、管理対象オブジェクトを抽象化した論理的なエンティティです。ここには、クラスとインスタンスが格納されます。ネームスペースは、ディレクトリ構造、データベース、フォルダなどのさまざまな形式で実装できます。デフォルトでは、CIM WorkShop は、ローカルホストのデフォルトネームスペース `root\cimv2` で動作している CIM Object Manager に接続します。デフォルトネームスペースに含まれるクラスはすべて、「CIM WorkShop」ウィンドウの左側に表示されます。現在のネームスペース名は、「CIM WorkShop」ウィンドウのツールバーに表示されます。CIM WorkShop では、個々のホストに存在するネームスペースのクラスを表示し、別のネームスペースに位置を変更できます。

特定のネームスペースにユーザー権限を設定する場合は、Sun WBEM User Manager を使用します。Sun WBEM User Manager ツールについては、『*Solaris WBEM Services* の管理』の「セキュリティの管理」を参照してください。

この節では、次の点について説明します。

- ネームスペースを作成する
- ネームスペースを変更する
- ホストを変更する

- ネームスペースのクラス継承ツリーを再表示する

ネームスペースの作成

すでにあるネームスペースの中に 1 つまたは複数のネームスペースを作成できます。

▼ ネームスペースを作成する方法

1. 「**CIM WorkShop**」ウィンドウで、「ワークショップ **(WorkShop)**」メニューの「ネームスペースを変更 **(Change Namespace)**」を選択します。
2. 新しいネームスペースを作成する先のネームスペースを右ボタンでクリックし、「ネームスペースを追加 **(Add Namespace)**」を選択します。
3. 入力ダイアログボックスに新しいネームスペースの名前を入力し、「了解 **(OK)**」をクリックします。

ネームスペースの変更

Sun WBEM SDK ではデフォルトのネームスペースとして `root\cimv2` が設定されていますが、ほかのネームスペースをデフォルトとして使用するように変更できます。

▼ ネームスペースを変更する方法

1. 「**CIM WorkShop**」ウィンドウで、「ワークショップ **(Workshop)**」、「ネームスペースを変更 **(Change Namespace)**」の順にクリックします。
2. 「ネームスペースを変更 **(Change Namespace)**」ダイアログボックスで、使用したいネームスペースのアイコンをクリックします。「了解 **(OK)**」をクリックします。

   選択したネームスペースが、現在のネームスペースになります。

ホストの変更

ネームスペースまたはプロセスを表示するホストを変更できます。

▼ ホストを変更する方法

1. 「ワークショップ **(Workshop)**」、「ホストを変更 **(Change Host)**」の順にクリックするか、「**CIM WorkShop**」ツールバーの「ホストを変更 **(Change Hosts)**」アイコンをクリックします。
2. 「ホスト名 **(Host Name)**」フィールドで、表示したいネームスペースが入っているホスト名を入力します。
3. 「ユーザ名 **(User Name)**」フィールドに自分のユーザー名を入力し、「パスワード **(Password)**」フィールドにパスワードを入力します。
4. 「了解 **(OK)**」をクリックします。

クラスとネームスペースの再表示

ネームスペース内のクラス継承ツリーを再表示して、そのネームスペースで作業を行なっているほかのユーザーが加えた変更を反映させることができます。

▼ クラス継承ツリーを再表示する方法

1. クラス継承ツリー内で、再表示したいクラスのフォルダをクリックします。
2. 「アクション **(Action)**」、「再表示 **(Refresh)**」の順にクリックするか、「**CIM WorkShop**」ツールバーの「選択したクラスを再表示 **(Refresh Selected Class)**」アイコンをクリックします。

クラスの処理

クラスは、アプリケーションの基礎となるものです。CIM WorkShop は、起動時に CIM スキーマと Solaris スキーマを構成するクラスを読み込みます。これらのクラスは、DMTF (Distributed Management Task Force) に準拠しています。これらのクラスの固有のプロパティ、メソッド、および修飾子の値は変更できません。

既存のクラスに新しい値を設定するには、そのクラスに新しいインスタンスまたはクラスを作成します。CIM スキーマと Solaris スキーマのクラスは、テンプレートとして機能します。新しいインスタンスまたはクラスを作成するには、それらのプロパティ、メソッド、および修飾子の値を追加できるように、選択したクラスのコピーを作成します。このような方法で、CIM スキーマまたは Solaris スキーマを独自に拡張することができます。

注 - 継承されたプロパティ、メソッド、修飾子の値は変更できません。

クラスのインスタンスを作成する方法については、47ページの「インスタンスの処理」を参照してください。新しいクラスの作成方法については、以下の節を参照してください。

クラスの追加

既存のクラスにクラスを追加するには、次の作業を行います。

- 既存のクラスを選択する
- 新しいクラスを作成する
- クラスに新しい修飾子を追加する
- クラスに新しいプロパティを追加する
- プロパティに新しい修飾子を追加する
- 修飾子の値 (スコープとフレーバ) を設定する

新しいクラスの作成

あるクラスに新しいサブクラスを作成するには、まずその既存のクラス名を指定します。CIM WorkShop では、クラス名は標準の CIM 構文 *SchemaIndicator_ClassName* で表示されます。CIM スキーマクラスに新しくクラスを作成する場合、クラス名の前に頭字語 *CIM* を使用します。Solaris スキーマクラスにクラスを作成する場合、クラス名の前に *Solaris* を使用します。キー修飾子を継承するクラスの名前には、下線 (_) を指定する必要があります。

▼ クラスを追加する方法

1. 「**CIM WorkShop**」ウィンドウのクラス継承ツリーで、新しくクラスを作成する既存のクラスを選択します。

2. クラスを作成するには、次のいずれかの操作を行います。
   - 「アクション (Action)」、「クラスを追加 (Add Class)」の順にクリックする
   - 「CIM WorkShop」ウィンドウでツールバーの「新規クラス (New Class)」アイコンをクリックする
   - 選択したクラス上で右ボタンをクリックし、「クラスを追加 (Add Class)」を選択する

     「新規クラス (New Class)」ダイアログボックスが表示されます。

3. 「クラス名 **(Class Name)**」フィールドで、新しいクラス名を入力します。

   たとえば、クラス `Solaris_ComputerSystem` に `Ultra1_ComputerSystem` という名前のクラスを作成できます。

4. クラスから継承されたプロパティとメソッドをそのまま使用するには、「了解 **(OK)**」をクリックします。新しいプロパティを追加するには、「プロパティを追加 **(Add Property)**」をクリックします。

   「了解 (OK)」をクリックすると、継承されたプロパティ、メソッド、修飾子、およびそれらの値を使用するクラスが作成されます。「プロパティを追加 (Add Property)」をクリックすると、「プロパティを追加 (Add Properties)」ダイアログボックスが表示されます。このダイアログボックスで新しいクラスに追加するプロパティを指定できます。プロパティの追加方法については、44ページの「クラスへの新しいプロパティの追加」を参照してください。

修飾子の追加

新しく作成したクラスには修飾子を追加できます。クラスを変更する継承されたクラスの修飾子の値は、変更も取り消しもできません。また、継承された修飾子を削除することもできません。

▼ 修飾子を追加する方法

1. 「新規クラス **(New Class)**」ダイアログボックスで、新しいクラス名を指定し、「クラス修飾子 **(Class Qualifiers)**」をクリックします。

2. 「修飾子 **(Qualifiers)**」ダイアログボックスで、新しい値を設定したい修飾子を右ボタンでクリックし、「修飾子の追加 **(Add Qualifier)**」を選択します。

3. 「修飾子の追加 **(Add Qualifier)**」ダイアログボックスで、リストから修飾子の名前を選択して「了解 **(OK)**」をクリックします。

4. 修飾子のスコープを設定するには、次の操作を行います。
   a. 「スコープ **(Scope)**」をクリックします。
   b. 「スコープ **(Scope)**」ダイアログボックスで、修飾子のスコープを選択して「了解 **(OK)**」をクリックします。

5. 修飾子のフレーバを設定するには、次の操作を行います。
   a. 「フレーバ **(Flavors)**」をクリックします。
   b. 「フレーバ **(Flavors)**」ダイアログボックスで、修飾子のフレーバを選択して「了解 **(OK)**」をクリックします。

6. 「修飾子 **(Qualifiers)**」ダイアログボックスで「了解 **(OK)**」をクリックし、このダイアログボックスを閉じます。

クラスへの新しいプロパティの追加

クラスに新しいプロパティを追加し、それらの値を変更できます。継承されたプロパティの値は変更できません。また、継承されたプロパティは削除できません。

▼ クラスに新しいプロパティを追加する方法

1. 新しいクラス名を指定した後、「新規クラス **(New Class)**」ダイアログボックスで「プロパティを追加 **(Add Property)**」をクリックします。

   「プロパティを追加 (Add Properties)」ダイアログボックスが表示されます。

2. 「名前 **(Name)**」フィールドで、新しいプロパティ名を入力します。

3. 「タイプ **(Type)**」フィールドでプロパティのデータ型を選択し、「了解 **(OK)**」をクリックします。

   「新規クラス (New Class)」ダイアログボックスの「プロパティ (Properties)」タブに新しいプロパティが表示されます。プロパティリストが長い場合は、スクロールバーをクリックして新しく追加したプロパティを表示できます。

4. 「新規クラス **(New Class)**」ダイアログボックスで「了解 **(OK)**」をクリックします。

   新しいプロパティまたはクラスに新しい修飾子を追加する方法、および修飾子の値を設定する方法については、以下の節を参照してください。

新しいプロパティへの修飾子の追加

クラスに新たに作成したプロパティには、修飾子の値も設定できます。継承されたプロパティまたはメソッドの修飾子の値は、変更も取り消しも行うことはできません。また、継承された修飾子を削除することはできません。

▼ 新しいプロパティに修飾子を追加する方法

1. 「新規クラス **(New Class)**」ダイアログボックスで、作成した新しいプロパティをクリックし、「プロパティ修飾子 **(Property Qualifiers)**」をクリックします。

作成したプロパティの「修飾子 (Qualifiers)」ダイアログボックスが表示されます。

2. 「修飾子の追加 **(Add Qualifier)**」をクリックします。

3. 「修飾子の追加 **(Add Qualifier)**」ダイアログボックスの「名前 **(Name)**」フィールドで、修飾子を選択して「了解 **(OK)**」をクリックします。

4. 「修飾子 **(Qualifiers)**」ダイアログボックスと「新規クラス **(New Class)**」ダイアログボックスで、「了解 **(OK)**」をクリックします。

   選択したプロパティに修飾子と修飾子のデータ型が設定されます。

クラスとクラス属性の削除

CIM WorkShop では、不要になったクラス、プロパティ、メソッド、および修飾子は削除できます。

注 - クラスを削除すると、そのクラスに含まれるサブクラスがすべて削除されます。また、クラスとそのサブクラスに対応するプロパティ、メソッド、修飾子もすべて削除されます。

クラスの削除

クラス継承ツリーからクラスを削除するには、次のようにします。

▼ クラスを削除する方法

1. 削除したいクラスを選択します。

2. 「アクション **(Actions)**」、「クラスを削除 **(Delete Class)**」の順にクリックし、クラスを削除してもいいかどうかの確認を求めるダイアログボックスで「了解 **(OK)**」をクリックします。

選択したクラスが削除されます。

クラスプロパティの削除

新しいクラスに作成したプロパティだけが、削除することができます。既存のクラスのプロパティは、表示はできますが、変更と削除はできません。サブクラスに継承されたプロパティを削除することもできません。ただし、新しいクラスを作成する場合には、そのクラスに新たに追加したプロパティはどれでも削除できます。クラスの作成方法については、41ページの「クラスの追加」を参照してください。

▼ クラスプロパティを削除する方法

1. 「新規クラス **(New Class)**」ダイアログボックスの「プロパティ **(Properties)**」タブで、プロパティ名を選択するか入力し、「プロパティを削除 **(Delete Property)**」をクリックします。

修飾子の削除

新しいクラスを作成した場合は、親クラスから継承されたプロパティ修飾子またはメソッド修飾子を削除できます。クラスの作成方法については、41ページの「クラスの追加」を参照してください。

▼ プロパティ修飾子を削除する方法

1. 「新規クラス **(New Class)**」ダイアログボックスの「プロパティ **(Properties)**」タブで、削除したいプロパティを選択します。

2. 「プロパティ修飾子 **(Property Qualifiers)**」をクリックします。

3. 「修飾子 **(Qualifiers)**」ダイアログボックスで、削除する修飾子を選択し、「修飾子を削除 **(Delete Qualifier)**」をクリックしてから、「了解 **(OK)**」をクリックします。

   選択した修飾子が削除され、「新規クラス (New Class)」ダイアログボックスが表示されます。

▼ メソッド修飾子を削除する方法

1. 「新規クラス **(New Class)**」 ダイアログボックスの「メソッド **(Methods)**」タブで、削除したいメソッドを右ボタンでクリックします。
2. ポップアップメニューで「修飾子 **(Qualifiers)**」をクリックします。
3. 削除する修飾子を決めます。
4. 削除する修飾子を右ボタンでクリックし、「修飾子を削除 **(Delete Qualifier)**」を選択します。または、削除する修飾子を選択し、「修飾子を削除 **(Delete Qualifier)**」ボタンをクリックします。
5. このメソッドの修飾子をリスト表示するには、「修飾子 **(Qualifier)**」を選択します。

▼ メソッドに修飾子を追加する方法

1. 「新規クラス **(New Class)**」ダイアログボックスの「メソッド **(Methods)**」タブで、修飾子を追加する先のメソッドを右ボタンでクリックし、「修飾子 **(Qualifiers)**」を選択します。
2. 「修飾子 **(Qualifiers)**」ダイアログボックスの「修飾子を追加 **(Add Qualifier)**」ボタンをクリックします。
3. 追加する修飾子の型を選択し、「了解 **(OK)**」をクリックします。次に「了解 **(OK)**」をクリックして「修飾子 **(Qualifiers)**」ダイアログボックスを閉じます。

インスタンスの処理

CIM WorkShop ではクラスのインスタンスを作成できます。 インスタンスは、クラスの特性を継承します。続いて、新しいインスタンスの属性を変更して、固有のクラスインスタンスを作成できます。

インスタンスの表示

新しいクラスインスタンスを作成する前に、既存のクラスインスタンスを表示してそれらに含まれるプロパティとメソッドを確認しておくことをお勧めします。

▼ 既存のクラスのインスタンスを表示する方法

1. 「**CIM WorkShop**」ウィンドウのクラス継承ツリーで、インスタンスを表示したいクラスを選択します。

2. 次のどちらかの操作を行い、「インスタンス **(Instances)**」ウィンドウを表示します。
   - 「アクション (Action)」、「インスタンス (Instances)」、「詳細列挙 (Deep Enumeration)」または「簡易列挙 (Shallow Enumeration)」の順にクリックする。

     または
   - 「CIM WorkShop」ツールバーの「インスタンスを表示 (Show Instances)」アイコンをクリックする

     または
   - クラスを右ボタンでクリックし、ポップアップダイアログボックスから「詳細列挙 (Deep Enumeration)」または「簡易列挙 (Shallow Enumeration)」を選択する。

「インスタンス (Instances)」ウィンドウが表示されます。選択したクラスにインスタンスが含まれている場合、「インスタンス (Instances)」ウィンドウの左側のフレームにそれらのインスタンスが表示されます。インスタンスが含まれていない場合、このフレームには何も表示されません。

インスタンスの追加

継承されたオブジェクト修飾子を変更したい場合は、クラスにインスタンスを追加します。

▼ クラスにインスタンスを追加する方法

1. 「**CIM WorkShop**」ウィンドウで、次のどちらかの操作を行います。

   - 「アクション (Action)」、「インスタンス (Instances)」、「詳細列挙 (Deep Enumeration)」の順にクリックして、現在のクラスとそのすべての下位クラスのインスタンスをリスト表示する。

     または

   - 「アクション (Action)」、「インスタンス (Instances)」、「簡易列挙 (Shallow Enumeration)」の順にクリックして、現在のクラスのインスタンスをリスト表示する。

     「インスタンス (Instances)」ウィンドウが表示され、クラスのすべてのインスタンスが左側のフレームに表示されます。

2. 「インスタンス **(Instances)**」ウィンドウに表示されたインスタンスの **1** つで右ボタンをクリックします。

   「インスタンスを追加 (Add Instances)」ダイアログボックスが表示されます。これには、オプションとして「再表示 (Refresh)」、「インスタンスを追加 (Add Instance)」、「インスタンスを削除 (Delete Instance)」のボタンがあります。

3. 「インスタンスを追加 **(Add Instance)**」をクリックします。

4. 継承されたインスタンスプロパティを変更するには、次のようにします。

   a. 変更する値フィールドをクリックします。

      ダイアログボックスが表示され、プロパティに値を指定できます。表示されるダイアログボックスは、選択されるプロパティのデータ型によって異なります。たとえば、データ型 `string` を持つプロパティを選択すると、「文字列 (String)」ダイアログボックスが表示されます。このダイアログボックスの「値 (Value)」フィールドは、文字列だけを入力できます。

   b. ダイアログボックスの「値 **(Values)**」フィールドで、必要な値を入力します。

5. 「了解 **(OK)**」をクリックして、「インスタンスを追加 **(Add Instances)**」ウィンドウを閉じます。

インスタンスの削除

不要になったインスタンスは削除できます。

▼ インスタンスを削除する方法

1. 「**CIM WorkShop**」ウィンドウの左側のフレームで、削除したいインスタンスが入っているクラスを右ボタンでクリックします。

2. ポップアップメニューで、「インスタンス **(Instances)**」、「詳細列挙 **(Deep Enumeration)**」の順にクリックして、選択したクラスとそのサブクラスのインスタンスをリスト表示します。または、「インスタンス **(Instances)**」、「簡易列挙 **(Shallow Enumeration)**」の順にクリックして、選択したクラスのインスタンスをリスト表示します。

3. 「インスタンス **(Instance)**」ウィンドウで、削除したいインスタンスを右ボタンでクリックし、ポップアップメニューの「インスタンスの削除 **(Delete Instance)**」をクリックします。

   選択したインスタンスが削除されます。

メソッドの呼び出し

CIM WorkShop では、メソッドのパラメータへの入力値を設定してからメソッドを呼び出すことができます。メソッドの関数は、文字列、ブール式、整数といった入力パラメータの値を受け取り、関数を実行します。このメソッドを呼び出すと、出力パラメータの形で追加のデータが返されます。

パラメータの値を設定し、メソッドを呼び出すためのダイアログボックスについては、61ページの「値のデータ型を指定するダイアログボックス」と 67ページの「「メソッドの呼び出し (Invoke Methods)」 ダイアログボックス」を参照してください。

▼ メソッドを呼び出す方法

1. 「**CIM WorkShop**」ウィンドウで次のようにします。

2. 「アクション **(Action)**」、「インスタンス **(Instances)**」、「詳細列挙 **(Deep Enumeration)**」の順にクリックして、選択したクラスとそのサブクラスのインスタンスをリスト表示します。または、「アクション **(Action)**」、「インスタンス **(Instances)**」、「簡易列挙 **(Shallow Enumeration)**」の順にクリックして、選択したクラスのインスタンスをリスト表示します。

3. 「メソッド **(Methods)**」タブをクリックします。

4. 呼び出すメソッドを右ボタンでクリックし、「メソッドの呼び出し **(Invoke Method)**」を選択します。

5. 「メソッドの呼び出し **(Invoke Methods)**」ダイアログボックスの「入力値 **(Input Value)**」列で、追加する値のセルをクリックします。
   「入力値 (Input Value)」ダイアログボックスが開きます。

6. パラメータのセルに値を入力し、「了解 **(OK)**」をクリックします。

7. 「メソッドの呼び出し **(Invoke Method)**」をクリックします
   すべての出力値と戻り値が、自動的に表示されます。

8. 別の入力値を追加する場合は、「入力値 **(Input Value)**」列の対応するセルに値を入力します。

9. 新しい入力値の追加とメソッドの呼び出しが終ったら、「閉じる **(Close)**」をクリックします。

リファレンス:「CIM WorkShop」ウィンドウとダイアログ

次の節では、「CIM WorkShop」ウィンドウを構成するフレーム、ツールバーアイコン、およびフィールドについて説明します。また、CIM WorkShop のダイアログボックスについても説明します。

「CIM WorkShop」ウィンドウ

「CIM WorkShop」ウィンドウは、2 つのフレームに分かれています。左側のフレームには、現在のホストのクラス継承ツリーが表示されます。右側のフレームには、選択されているクラスのプロパティとメソッドが表示されます。

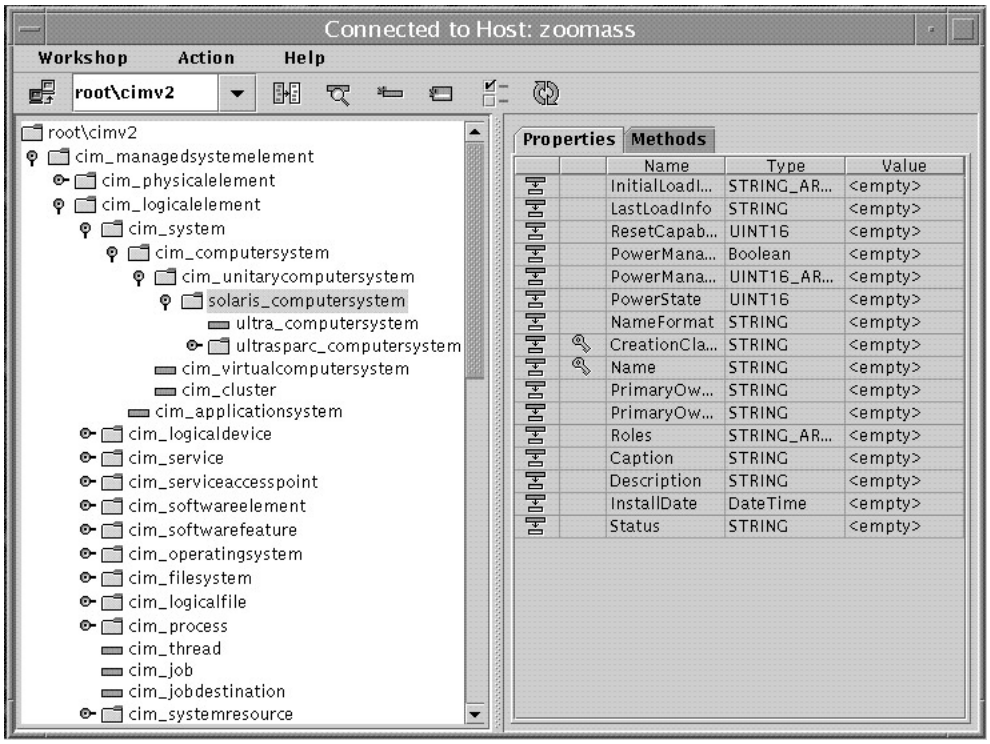

図 2–2　「CIM WorkShop」ウィンドウ

表 2–1 「CIM WorkShop」ウィンドウのフレーム

フレーム	説明
左側のフレーム	現在のホストのネームスペースに含まれるクラスとインスタンスを表示する。CIM WorkShop の左側のフレームには、選択したネームスペースの内容を表示する。ネームスペースに含まれるクラスは、階層表示される。このクラス編成を、クラス継承ツリーと呼ぶ。サブクラスを含むクラスは、キーアイコンとフォルダとして示される。キーをクリックするか、フォルダをダブルクリックすると、サブクラスリストが表示される。サブクラスを含まないクラスは、ページアイコンとして示される。
右側のフレーム	クラスのプロパティの値を表示できる「プロパティ (Properties)」タブと、メソッドの値を表示できる「メソッド (Methods)」タブが入っている。プロパティまたはメソッドを右クリックすると、修飾子とフレーバの属性と値を表示できる。
ツールバー	ホストの変更、デフォルトネームスペース `root\cimv2` 内の別のネームスペースへの位置の変更、クラス継承ツリー内でのクラスの検索、サブクラスの作成、選択したクラスのインスタンスと修飾子の表示、選択したクラスの再表示などを行うアイコンを表示する。
タイトルバー	「CIM WorkShop」ウィンドウのタイトルを表示する。

「CIM WorkShop」ツールバーのアイコン

「CIM WorkShop」ツールバー上のアイコンは、ネームスペースの表示と変更、およびクラスとインスタンスの検索に使用します。

図 2–3 「CIM WorkShop」ツールバー

表 2–2 「CIM WorkShop」ツールバーのアイコン

アイコン	説明
ホストを変更	別のホストまたはネームスペースに接続し、別のユーザー名とパスワードでログインし、転送プロトコルを設定する。
ネームスペースを変更	「ネームスペースを変更 (Change Name Space)」ダイアログボックスを呼び出し、表示するネームスペースを変更する。
クラスを検索	ネームスペース内で特定のクラスを検索する。
新規クラスを追加	「新規クラス (New Class)」ダイアログボックスを表示し、選択されているクラスの新しいサブクラスを作成する。
インスタンスを表示	「インスタンスを表示 (Show Instances) ダイアログボッスを表示し、選択したクラスのインスタンスを表示する。
修飾子を表示	「修飾子 (Qualifiers)」ダイアログボックスを表示し、選択したクラスの修飾子を表示する。
選択したクラスを再表示	クラス階層ツリーの表示をリセットする。開かれているクラスフォルダが閉じられ、ツリーは最初に表示された時の状態に戻る。

「プロパティ (Properties)」タブ

「プロパティ (Properties)」タブは、選択したプロパティの情報を表示します。矢印付きのフォルダの形をしたアイコンは、そのプロパティがスーパークラスから継承されたものであることを示します。金色の鍵のアイコンは、そのプロパティがキーであることを示します。キープロパティは、ドメインクラスのインスタンスに固有の識別子を提供します。固有のインスタンスは、キー修飾子によって示されます。

「プロパティ (Properties)」タブでは、プロパティの名前、データ型および値が表示されます。ドメインクラスに新しいクラスを作成する場合は、プロパティの値を変更できます。

「メソッド (Methods)」タブ

メソッドとは、クラスの動作を記述した関数です。メソッドの例としては、サービスの開始や停止、ディスクのフォーマットなどの動作があります。「メソッド (Methods)」タブを選択すると、クラスのすべてのメソッドを表示できます。メソッドは順次に表示されます。

メソッドは、シグニチャと本体で構成されています。シグニチャは、メソッド名と、パラメータの名前、型、順序、メソッドの戻り型からなります。メソッド本体は一連の命令からなります。

メソッドには、左から右に次の 3 つの部分が含まれています。

- 戻り値データ型 — このメソッドから返される戻り値のデータ型です。
- メソッドの名前
- パラメータ — パラメータリスト全体がかっこで囲まれ、個々のパラメータがコンマで区切られています。個々のパラメータは名前とデータ型からなります。メソッドへの入力となるパラメータの前には [IN] が、メソッドからの出力となるパラメータの前には [OUT] がそれぞれ指定されています。パラメータには、1 つまたは複数の修飾子が指定されていることがあります。

次の例では、メソッド `SetDateTime` は、型が `datetime` の入力パラメータ `Time` を受け取り、ブール値を返します。

```
boolean SetDateTime([IN(true)] datetime Time);
```

CIM WorkShop のメニュー

次の表で、CIM WorkShop のメニューとメニュー項目について説明します。

表 2–3 CIM WorkShop のメニューとメニュー項目

メニュー	メニュー項目	説明
Workshop	ホストを変更 (Change Host)	「ログイン (Login)」ダイアログボックスを表示する。このダイアログボックスでは、ホストとネームスペースを変更できる。
	ネームスペースを変更 (Change Namespace)	「ネームスペースを変更 (Change Namespace)」ダイアログボックスを表示する。このダイアログボックスでは、`root\cimv2` ネームスペース内のデフォルトネームスペース以外の場所に移ることができる。
	終了 (Exit)	CIM WorkShop を終了する。

表 2–3 CIM WorkShop のメニューとメニュー項目 続く

メニュー	メニュー項目	説明
アクション (Action)	クラスを追加 (Add Class)	「新規クラス (New Class)」ダイアログボックスを表示し、選択されているクラスのサブクラスを作成する。
	クラスを削除 (Delete Class)	選択されているクラスを削除する。
	クラスを検索 (Find Class)	クラス継承ツリー内で検索するクラスを指定する。
	インスタンス (Instances)	選択したクラスの「インスタンス (Instances)」ダイアログボックスを表示する。このダイアログボックスでは、クラスに含まれるすべてのインスタンスの表示、新しいインスタンスの追加、インスタンスの削除を行うことができる。
	修飾子 (Qualifiers)	「修飾子 (Qualifiers)」ダイアログボックスを表示する。このダイアログボックスでは、選択したクラスの修飾子の値、スコープ、およびフレーバを表示する。インスタンスの詳細列挙または簡易列挙を選択できる。
	関連の表示 (Association Traversal)	「関連の表示 (Association Traversal)」ダイアログボックスを表示する。このダイアログボックスでは、クラスの関連を表示し、たどることができる
再表示 (Refresh)	CIM WorkShop に加えた最新の変更を元に戻し、選択されたクラスまたはネームスペースを表示する	

「ログイン (Login)」ダイアログボックス

「ログイン (Login)」ダイアログボックスは、CIM WorkShop の起動時に表示されます。「ログイン (Login)」ダイアログボックスでは、次のパラメータを指定します。

- CIM Object Manager が動作しており、使用したいネームスペースが入っているホスト
- 作業を行うネームスペース
- ユーザー名
- パスワード
- 転送プロトコル

ユーザー名とパスワードを指定しない場合は、CIM にゲストとしてログインすることになります。この場合許可されるのは読み取り権だけです。

デフォルトでは、CIM WorkShop は RMI プロトコルを使って、ローカルホスト (デフォルトのネームスペースは `root\cimv2`) 上の CIM Object Manager に接続します。Desktop Management Task Force の標準 XML/HTTP プロトコルを使って CIM Object Manager と通信したい場合は、HTTP を選択できます。接続が確立されると、デフォルトのネームスペースに含まれているすべてのクラスが「CIM WorkShop」ウィンドウ内の左側に表示されます。

「新規クラス (New Class)」ダイアログボックス

「新規クラス (New Class)」ダイアログボックスでは、クラスに新しいクラスを作成できます。

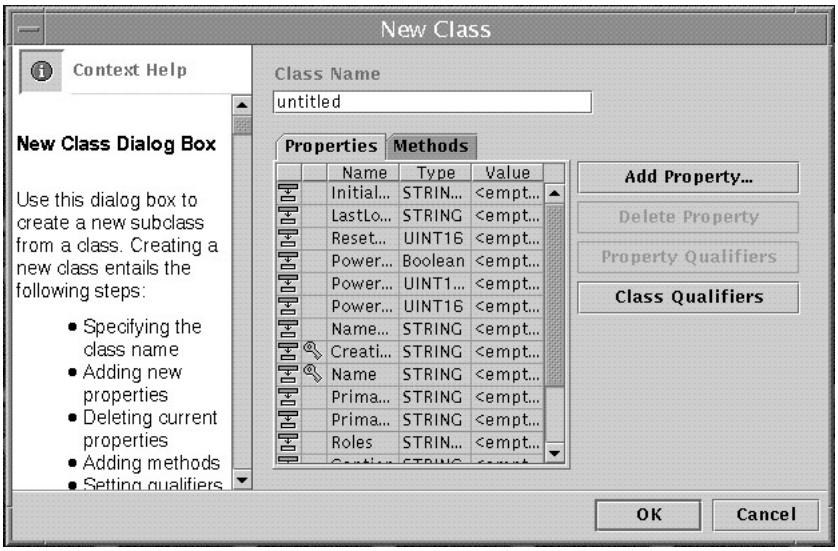

図 2–4　「新規クラス (New Class)」ダイアログボックス

「プロパティを追加 (Add Properties)」ダイアログボックス

「プロパティを追加 (Add Properties)」ダイアログボックスでは、作成したクラスに新しいプロパティを追加できます。「名前 (Name)」フィールドでは、プロパティ名を指定します。「タイプ (Type)」フィールドでは、データ型を選択して「了解 (OK)」をクリックします。

「修飾子 (Qualifiers)」ダイアログボックス

「修飾子 (Qualifiers)」ダイアログボックスでは、選択されているクラス、プロパティ、メソッドの修飾子を表示できます。新しいクラスを作成する場合は、このダイアログボックスでクラスへの修飾子の追加、クラス、そのプロパティ、またはそのメソッドの修飾子の変更を行うことができます。「修飾子 (Qualifiers)」ダイアログボックスのタイトルバーには、修飾子を表示しているクラス名、または修飾子の追加または変更を行うクラス名が表示されます。

次の表で、「修飾子 (Qualifiers)」ダイアログボックスの各フィールドについて説明します。

表 2–4 「修飾子 (Qualifiers)」ダイアログボックスのフィールド

フィールド名	説明	例
名前 (Name)	修飾子の名前を表示する。	プロバイダ
タイプ (Type)	修飾子が提供する値のデータ型を示す。	`string`
値 (Value)	修飾子の値を表示する。	`Solaris`

次の表で、「修飾子 (Qualifiers)」ダイアログボックスのボタンについて説明します。

表 2–5 「修飾子 (Qualifiers)」ダイアログボックスのボタン

ボタン名	説明
スコープ (Scope)	「スコープ (Scope)」ダイアログボックスを表示し、選択されている修飾子のスコープを表示する。
フレーバ (Flavors)	「フレーバ (Flavors)」ダイアログボックスを表示し、選択されている修飾子のフレーバを表示する。
修飾子を追加 (Add Qualifier)	「修飾子を追加 (Add Qualifier)」ダイアログボックスを表示し、新しいサブクラス、プロパティ、またはメソッドに追加できる修飾子を選択する。
修飾子を削除 (Delete Qualifier)	選択されている修飾子を「修飾子 (Qualifiers)」ダイアログボックスから削除する。

「スコープ (Scope)」ダイアログボックス

「スコープ (Scope)」ダイアログボックスでは、既存のクラス、プロパティ、またはメソッドを変更する修飾子のスコープを表示できます。

「フレーバ (Flavors)」ダイアログボックス

「フレーバ (Flavors)」ダイアログボックスでは、修飾子のフレーバを表示できます。

値のデータ型を指定するダイアログボックス

クラスのプロパティを新たに作成したり、メソッドの入力パラメータを設定する場合には、特定のデータ型を持つ値を指定するために CIM WorkShop で提供されている任意のダイアログボックスを使用できます。これらのダイアログボックスは、適切なデータ型の値だけを入力できるように設定されています。使用できるダイアログボックスを次に示します。

- 「実数型整数 (Real Integer)」ダイアログボックス
- 「符号付き整数 (Signed Integer)」ダイアログボックス
- 「符号なし整数 (Unsigned Integer)」ダイアログボックス
- 「文字列 (String)」ダイアログボックス
- 「配列 (Array)」ダイアログボックス
- 「ブール (Boolean)」ダイアログボックス
- 「日付/時刻 (Date/Time)」ダイアログボックス

「実数型整数 (Real Integer)」ダイアログボックス

このダイアログボックスの「値 (Values)」フィールドには、実数型の整数だけを入力できます。実数型の整数は正、負のどちらでもよく、小数点も使用できます。データ型が実数型の整数のプロパティを作成する場合は、このダイアログボックスの「値 (Values)」フィールドに実数型の整数を入力してください。

「符号付き整数 (Signed Integer)」ダイアログボックス

このダイアログボックスの「値 (Values)」フィールドには、指定したサイズの符号付き整数だけを入力できます。符号付き整数は、正または負の整数です。符号付き整数である値を持つ CIM プロパティのサイズは、8 ビット、16 ビット、32 ビット、または 64 ビットです。プロパティの値を構成する符号付き整数のサイズに基づき、このダイアログボックスの「値 (Values)」フィールドに次の値を入力してください。

- 8 ビットの符号付き整数であるプロパティには、8 ビット相当の正または負の数値を入力する。
- 16 ビットの符号付き整数であるプロパティには、16 ビット相当の正または負の数値を入力する。

- 32 ビットの符号付き整数であるプロパティには、32 ビット相当の正または負の数値を入力する。
- 64 ビットの符号付き整数であるプロパティには、64 ビット相当の正または負の数値を入力する。

「符号なし整数 (Unsigned Integer)」ダイアログボックス

このダイアログボックスの「値 (Values)」フィールドには、指定したサイズの符号なし整数だけを入力できます。符号なし整数は正の整数です。符号なし整数である値を持つ CIM プロパティのサイズは、8 ビット、16 ビット、32 ビット、または 64 ビットです。プロパティの値を構成する符号なし整数のサイズに基づき、このダイアログボックスの「値 (Values)」フィールドに次の値を入力してください。

- 8 ビットの符号なし整数であるプロパティには、8 ビット相当の正の数値を入力する。
- 16 ビットの符号なし整数であるプロパティには、16 ビット相当の正の数値を入力する。
- 32 ビットの符号なし整数であるプロパティには、32 ビット相当の正の数値を入力する。
- 64 ビットの符号なし整数であるプロパティには、64 ビット相当の正の数値を入力する。

「文字列 (String)」ダイアログボックス

このダイアログボックスの「値 (Values)」フィールドには、英数字を入力できます。タイプが文字列であるプロパティの値を指定する場合は、このダイアログボックスの「値 (Values)」フィールドに文字列 (例: Processor_Type) を入力します。文字列には、整数は使用できません。

「配列 (Array)」ダイアログボックス

「配列 (Array)」ダイアログボックスには、プロパティの値として配列を指定できます。次に、配列を返すために使用できる「配列 (Array)」ダイアログボックスを示します。

- 「8 ビット符号なし整数配列 (8–Bit Unsigned Integer Array)」ダイアログボックス – サイズが 8 ビット相当の正の整数のコレクションを返す。
- 「16 ビット符号なし整数配列 (16–Bit Unsigned Integer Array)」ダイアログボックス – サイズが 16 ビット相当の正の整数のコレクションを返す。
- 「32 ビット符号なし整数配列 (32–Bit Unsigned Integer Array)」ダイアログボックス – サイズが 32 ビット相当の正の整数のコレクションを返す。
- 「64 ビット符号なし整数配列 (64–Bit Unsigned Integer Array)」ダイアログボックス – サイズが 64 ビット相当の正の整数のコレクションを返す。
- 「8 ビット符号付き整数配列 (8–Bit Signed Integer Array)」ダイアログボックス – サイズが 8 ビット相当の正または負の整数のコレクションを返す。
- 「16 ビット符号付き整数配列 (16–Bit Signed Integer Array)」ダイアログボックス – サイズが 16 ビット相当の正または負の整数のコレクションを返す。
- 「32 ビット符号付き整数配列 (32–Bit Signed Integer Array)」ダイアログボックス – サイズが 32 ビット相当の正または負の整数のコレクションを返す。
- 「64 ビット符号付き整数配列 (64–Bit Signed Integer Array)」ダイアログボックス – サイズが 64 ビット相当の正または負の整数のコレクションを返す。
- 「文字列配列 (String Array)」ダイアログボックス – 英字文字列と数字文字列のコレクションを返す。
- 「ブール式配列 (Boolean Array)」ダイアログボックス – ブール式 (TRUE または FALSE) のコレクションを返す。
- 「32 ビット実数配列 (32–Bit Real Array)」ダイアログボックス – サイズが 32 ビット相当の、小数点付きまたは小数点なしの正または負の実数のコレクションを返す。
- 「64 ビット実数配列 (64–Bit Real Array)」ダイアログボックス – サイズが 64 ビット相当の、小数点付きまたは小数点なしの正または負の実数のコレクションを返す。
- 「16 ビット文字配列 (16–Bit Character Array)」ダイアログボックス – サイズが 16 ビット相当の、英字文字列と数字文字列のコレクションを返す。
- 「日付 / 時間配列(Date/Time Array)」ダイアログボックス – *mm-dd-yy* 形式による日付と、*hh:mm:ss* 形式による時間のコレクションを返す。

「ブール (Boolean)」ダイアログボックス

「ブール (Boolean)」ダイアログボックスでは、選択したプロパティの値として TRUE または FALSE を指定できます。

「日付/時刻 (Date/Time)」ダイアログボックス

「日付/時刻 (Date/Time)」ダイアログボックスでは、CIM 仕様に基づいて、新しいプロパティやメソッドの日付や時刻の値を設定できます。「日付/時刻 (Date/Time)」ダイアログボックスは、次のダイアログボックスから表示します。

- DateTime 値を使用する新しいプロパティの値を選択するときに「新規クラス (New Class)」ダイアログボックスから。
- メソッドのパラメータの「入力値 (Input Value)」を選択するときに「メソッドの呼び出し (Invoke Methods)」ダイアログボックスから。

値を設定するためのダイアログボックスが表示されます。このダイアログボックスの「追加 (Add)」をクリックして、「日付/時刻 (Date/Time)」ダイアログボックスを表示します。

「日付/時刻 (Date/Time)」ダイアログボックスのフィールドに、日付と時刻を次の形式で入力します。

- 4 桁の年を *yyyy* で
- 月を *mm* で
- 日を *dd* で
- 時を 24 時間単位の *hh* で
- 分を *mm* で
- 秒を *ss* で
- マイクロ秒を *mmmmmm* で
- 世界標準時 (UTC) からの時差 (差分) を表す正符号 (+)、負符号 (-)、または (:) で

注 - *(:)* を指定すると値は時間間隔とみなされるため、*yyyymm* は日数として解釈されます。

- UTC からのオフセット (分) を *utc* で

たとえば、1999 年 6 月 7 日 月曜日、午後 1:30:15 (EST) は *19990607133015.000000-300* となります。

「インスタンス (Instance)」ウィンドウ

「インスタンス (Instance)」ウィンドウには、選択したクラスのすべてのインスタンスが表示されます。また、各インスタンスに関連付けられたプロパティ、メソッド、および修飾子も表示できます。

「インスタンス (Instance)」ウィンドウは、次のどちらかの操作で表示します。

- 「CIM WorkShop」ウィンドウでクラスを右ボタンでクリックし、ポップアップメニューの「インスタンス (Instances)」をクリックする
- 「アクション (Action)」、「インスタンス (Instances)」、「詳細列挙 (Deep Enumeration)」または「簡易列挙 (Shallow Enumeration)」を順にクリックします。

「インスタンス (Instances)」ウィンドウのフレーム

選択したクラスにインスタンスが含まれる場合、それらのインスタンスは「インスタンス (Instances)」ウィンドウの左側のフレームに表示されます。各インスタンスは、Name (名前)、CreationClassName (属するクラス名)、および TargetOperatingSystem (対象オペレーティングシステム) と共に表示されます。選択したクラスにインスタンスが含まれない場合、1 つのメッセージが表示されます。

「CIM WorkShop」ウィンドウと同様に、「インスタンス (Instances)」ウィンドウの右側のフレームには 2 つのタブ、「プロパティ (Properties)」タブと「メソッド (Methods)」タブがあります。選択したインスタンスのプロパティはすべて、「プロパティ (Properties)」タブの表に表示されます。この表の左端の列に現れる「継承プロパティ (Inherited Properties)」アイコン (紫色の四角形と、白い四角形を指す矢印から構成される) は、そのインスタンスの作成に使用されたクラスからプロパティが継承されたことを示します。金色の鍵の形をした「キー修飾子 (Key Qualifiers)」アイコンは、プロパティに継承されたキー修飾子が含まれていることを示します。

「インスタンス (Instances)」ウィンドウのツールバーのアイコン

「インスタンス (Instances)」ウィンドウのツールバーには、次の表に示すアイコンが含まれます。

表 2–6 「インスタンス (Instances)」ウィンドウのツールバー上のアイコン

アイコン名	説明
新しいインスタンスを追加	「インスタンスを追加 (Add Instance)」ダイアログボックスを表示し、クラス継承ツリーに追加する新しいインスタンスを作成する。
選択したインスタンスを削除	選択したインスタンスを削除する。
現在のインスタンスプロパティ値を保存	現在のインスタンスのプロパティ値を更新する。
インスタンスリストを更新	新たに作成したインスタンスとインスタンスに加えた最新の変更が反映されるように、「インスタンス (Instances)」ウィンドウの左側に表示されたインスタンスリストを更新する。

「インスタンス (Instances)」ウィンドウのメニュー

「インスタンス (Instances)」ウィンドウには、次のメニューとメニュー項目が含まれます。

表 2–7 「インスタンス (Instances)」ウィンドウのメニュー

メニュー名	メニュー項目	説明
インスタンスエディタ (Instance Editor)	終了 (Exit)	「インスタンス (Instances)」ウィンドウを閉じる。
アクション (Action)	インスタンスを追加 (Add Instance)	「インスタンスを追加 (Add Instances)」ダイアログボックスを表示し、クラス継承ツリーに追加する新しいインスタンスを作成する。

表 2–7 「インスタンス (Instances)」ウィンドウのメニュー 続く

メニュー名	メニュー項目	説明
	インスタンスを削除 (Delete Instance)	選択したインスタンスを削除する。
	関連の表示 (Association Traversal)	「関連の表示 (Association Traversal)」ダイアログボックスを表示する。このダイアログボックスでは、クラスのすべての関連付けを表示、トラバースする (たどる) ことができる。
	再表示 (Refresh)	選択したクラスまたはネームスペースに対する最新の変更を CIM Object Manager から検索し、CIM WorkShop に表示する。
ヘルプ (Help)		CIM WorkShop 著作権情報を表示する。

「インスタンスを追加 (Add Instance)」ダイアログボックス

プロパティが変更可能な場合は、「インスタンスを追加 (Add Instance)」ダイアログボックスの値フィールドをクリックして、「値の編集」ダイアログボックスを開くことができます。そして、このダイアログボックスで値フィールドをクリックし、値ダイアログボックスを開きます。継承されたプロパティの値を変更することはできません。

「メソッドの呼び出し (Invoke Methods)」 ダイアログボックス

「メソッドの呼び出し (Invoke Methods)」ダイアログボックスは、メソッドが含まれているクラスのインスタンスの「メソッド (Methods)」タブから表示します。それには、メソッドを右ボタンでクリックし、メニューから「メソッドの呼び出し (Invoke Methods)」を選択します。「メソッドの呼び出し (Invoke Methods)」ダイアログボックスでは、メソッドの変数 (またはパラメータ) への入力値を設定

し、メソッドを呼び出すことができます。パラメータの例としては、符号付きの整数や、日付/時刻の値などがあります。

「メソッドの呼び出し (Invoke Methods)」ダイアログボックスでは、3 つのパラメータの型がサポートされます。

- 入力パラメータ: 関数実行時に渡すデータを指定します。
- 出力パラメータ: 関数からアプリケーションまたはプリンタの画面に返される戻り値を指定します。
- 入出力パラメータ: 関数に渡すデータと戻り値を指定します。

「メソッドの呼び出し (Invoke Methods)」ダイアログボックスの「パラメータの型 (Parameter Type)」列は、メソッドのパラメータが、入力、出力、入出力のいずれであるかを表します。入力パラメータの値は「入力値 (Input Value)」列に表示されます。出力パラメータの値は「出力値 (Output Value)」列に表示されます。

メソッドを呼び出す方法については、50ページの「メソッドの呼び出し」を参照してください。

第 3 章

アプリケーションプログラミングインタフェース

Sun WBEM SDK アプリケーションは、アプリケーションプログラミングインタフェース (API) を介して CIM (Common Information Model) Object Manager に情報またはサービスを要求します。この章で説明する内容は、次のとおりです。

- API について
- API パッケージ

CIM API、クライアント API、プロバイダ API についての詳細は、Javadoc リファレンスページを参照してください。

API について

API によって、CIM オブジェクトが記述され、処理されます。これらの API により、CIM オブジェクトが Java クラスとして記述されます。オブジェクトとは、プリンタ、ディスクドライブ、CPU などの管理対象リソースをコンピュータ用に記述したモデルです。CIM Object Manager は、CIM (Common Information Model) 2.1 の仕様に準拠しています。そのため、API によってモデル化されるオブジェクトは、標準の CIM オブジェクトに準拠しています。

プログラマは、これらのインタフェースを使用して管理対象オブジェクトを記述したり、特定のシステム環境内の管理対象オブジェクト情報を取り出すことができます。CIM を使用して管理対象オブジェクトをモデル化する場合の利点は、CIM に準拠するシステム間でそれらのオブジェクトを共有できることです。

API パッケージ

API は、次の 3 つのカテゴリに分類できます。

- CIM API – アプリケーションがすべての基本的な CIM 要素を表現するために使用する、共通のクラスとメソッド。CIM API は、オブジェクトをローカルシステムに作成します。
- クライアント API – アプリケーションが CIM Object Manager との間でデータ転送を行うために使用するメソッド。クライアント API は、ローカルシステムに作成されたオブジェクトを CIM Object Manager に転送します。
- プロバイダ API – CIM Object Manager とオブジェクトプロバイダが相互通信のために使用するインタフェース。

CIM API パッケージ (com.sun.wbem.cim)

次の表で、CIM API パッケージ内のインタフェースについて説明します。

表 3–1 CIM クラス

クラス	説明
CIMClass	CIM クラス。共通の型をサポートする CIM インスタンスの集まり (プロパティとメソッドのセットなど) を記述するオブジェクト。このインタフェースは、開発者が作成するオブジェクトグループに必要な CIM 値を入力するテンプレートを作成する。
CIMDataType	CIM のデータ型 (CIM 仕様の定義に基づく)。
CIMDateTime	CIM の日時表記。
CIMElement	CIM 要素。管理されるシステム要素の基底クラス。
CIMFlavor	CIM 修飾子フレーバ。修飾子の特性の 1 つであり、修飾子を派生クラスまたは派生インスタンスに伝達できるかどうか、および派生クラスまたは派生インスタンスが修飾子の本来の値をオーバーライドできるかどうかを指定する規則を記述する。

表 3–1　CIM クラス　続く

クラス	説明
CIMInstance	CIM データの単位。このインタフェースは、特定のクラスに属する管理対象オブジェクトの記述に使用される。インスタンスには実際のデータが含まれる。
CIMMethod	メソッド名、戻り値のデータ型、およびパラメータを含む宣言。
CIMNameSpace	CIM ネームスペース。ほかのネームスペース、クラス、インスタンス、修飾子のデータ型、および修飾子を含むことができるディレクトリに似た構造を持つ。
CIMObjectPath	CIM オブジェクトのパス名。オブジェクト名は、2 つの部分、ネームスペースとモデルパスから構成される。モデルパスは、ネームスペース内のオブジェクトを個々に識別する。
CIMParameter	CIM パラメータ。呼び出し側メソッドから CIM メソッドに渡される値。
CIMProperty	CIM クラスのインスタンスの特性を示す値。プロパティは、プロパティ値を設定する機能と、そのプロパティ値を返す機能の組み合わせと考えることができる。プロパティは、名前と 1 つのドメイン (そのプロパティを所有するクラス) を持つ。
CIMQualifier	クラス、インスタンス、メソッド、またはプロパティを記述する修飾子。このクラスは、管理対象オブジェクトの属性の変更に使用される (たとえば、ディスクに読み取り権だけを追加するなど)。修飾子は、2 つのカテゴリ、CIM (Common Information Model) によって定義されるものと、開発者によって定義されるものに分類される。
CIMQualifierType	CIM 修飾子の型。CIM 修飾子を作成するテンプレート。

表 3–1　CIM クラス　続く

クラス	説明
CIMScope	CIM スコープ。これは、修飾子を使用できる CIM オブジェクトを示す修飾子属性である。たとえば、修飾子 ABSTRACT は Scope (Class Association Indication) を持つが、これは ABSTRACT が、クラス、関連、インジケーションにだけ使用できることを意味する。
CIMValue	CIM 値。これはプロパティ、参照、および修飾子に割り当てることができる値である。CIM 値は、データ型 (CIMDataType) と実際の値を持つ。
UnsignedInt8	符号なし 8 ビット整数。
UnsignedInt16	符号なし 16 ビット整数。
UnsignedInt32	符号なし 32 ビット整数。
UnsignedInt64	符号なし 64 ビット整数。

例外クラス

例外クラスは、Sun WBEM SDK クラスで起こり得るエラー状態を記述します。CIMException クラスは、CIM 例外の基底クラスです。CIM のほかの例外クラスはすべて、CIMException クラスのサブクラスです。

次の表で、CIM の例外クラスについて説明します。

表 3–2　例外クラス

クラス	説明
CIMClassException	CIM クラスで発生する意味上の例外。MOF コンパイラ (mofcomp) は、このクラスを使用してコンパイル時に見つかる意味上のエラーを処理する。
CIMException	例外的な CIM の状態。これは、CIM 例外の基底クラスである。

表 3–2　例外クラス　続く

クラス	説明
CIMInstanceException	CIM インスタンスで発生する意味上の例外。
CIMMethodException	CIM メソッドで発生する意味上の例外。
CIMNameSpaceException	CIM ネームスペースで発生する意味上の例外。
CIMPropertyException	CIM プロパティで発生する意味上の例外。
CIMProviderException	CIM Object Manager のプロバイダで発生し得る例外状態。
CIMQualifierTypeException	CIM 修飾子のデータ型で発生し得る例外状態。
CIMRepositoryException	CIM リポジトリで発生し得る例外状態。
CIMSemanticException	CIM 要素で発生し得る意味上の例外。この例外は、通常、CIM Object Manager が CIM 要素の追加、変更、または削除を試みる場合と、CIM の仕様に準拠しない不正な状況が発生する場合にスローされる。
CIMTransportException	CIM トランスポートインタフェース (RMI と XML) で発生する例外状態。

クライアント API パッケージ (com.sun.wbem.client)

クライアント API パッケージには、クライアントアプリケーションと CIM Object Manager 間でデータを転送するクラスとメソッドが含まれています。アプリケーションは、CIMClient クラスを使用して CIM Object Manager に接続し、次の表に示す CIMClient クラスのメソッドを使用して CIM Object Manager との間でデータ転送を行います。

表 3–3　クライアントメソッド

メソッド	説明
associators	指定された CIM クラスまたはインスタンスに関連付けられている CIM クラスまたはインスタンスを取得する。
associatorNames	指定された CIM クラスまたはインスタンスに関連付けられている CIM クラスまたはインスタンスの名前を取得する。
close	CIM Object Manager へのクライアント接続を閉じる。このインタフェースは、クライアントセッションに使用されているリソースを解放する。
createClass	CIM クラスを、指定されたネームスペースに追加する。
createInstance	指定されたインスタンスが存在しない場合は、それを作成する。その CIM インスタンスがすでにある場合は、ID が CIM_ERR_ALREADY_EXISTS の CIMInstanceException をスローする。
createNameSpace	CIM ネームスペース (クラスとインスタンスが入ったディレクトリ) を作成する。クライアントアプリケーションは、CIM Object Manager に接続する場合、ネームスペースを指定する。その後の処理はすべて、CIM Object Manager ホスト上のそのネームスペース内で発生する。
createQualifierType	指定された CIM 修飾子型を、指定されたネームスペースに追加する。
deleteNameSpace	指定されたホスト上の指定されたネームスペースを削除する。
deleteClass	指定されたクラスを削除する。
deleteInstance	指定されたインスタンスを削除する。

表 3–3 クライアントメソッド 続く

`enumClass(CIMObjectPath path, boolean deep)`	*Path* で指定されたクラスを列挙する。このメソッドは、`CIMObjectPath` オブジェクトの列挙として 、各クラスの (内容ではなく) 名前を返す。*deep* が指定されていると、メソッドは、列挙されるクラスから派生したすべてのクラスの名前を列挙する。*shallow* が指定されていると、列挙するクラスの第一レベルの子の名前だけを列挙する。
`enumClass(CIMObjectPath path, boolean deep, boolean localonly)`	*Path* で指定されたクラスを列挙する。このメソッドは、CIMClass オブジェクトの列挙として、クラスの名前だけでなく、クラスの内容全体を返す。*deep* が指定されていると、メソッドは、列挙されるクラスから派生したすべてのクラスを列挙する。*shallow* が指定されていると、列挙するクラスの第一レベルの子だけを列挙する。*localOnly* が真の場合は、継承されたプロパティやメソッド以外のプロパティやメソッドだけを返し、真でない場合は、すべてのプロパティとメソッドを返す。
`enumInstances(CIMObjectPath path, boolean deep)`	*Path* で指定されたクラスのインスタンスを列挙する。このメソッドは、`CIMObjectPath` オブジェクトの列挙として、*path* で指定されたクラスのインスタンスの名前を返す。*deep* が真の場合は、指定されたクラスとこのクラスのすべての派生クラスのすべてのインスタンスの名前を返す。真でない場合は、指定されたクラスに属するインスタンスの名前だけを返す。
`enumInstances(CIMObjectPath path, boolean deep, boolean localOnly)`	*Path* で指定されたクラスのインスタンスを列挙する。このメソッドは、`CIMInstance` オブジェクトの列挙として、指定されたクラスのインスタンス (インスタンスの名前だけでなく、インスタンス全体) を返す。
`enumNameSpace`	ネームスペースのリストを取得する。
`enumQualifierTypes`	指定されたクラス (1 つまたは複数) の、修飾子のデータ型を取得する。

表 3–3 クライアントメソッド 続く

`execQuery(CIMObjectPath relNS, java.lang.String query, int ql)`	照会に指定されたプロパティ値と同じプロパティを持つインスタンスの列挙を返す。照会言語には ql を使用する。現在は、WQL (WBEM Query Language) だけがサポートされる。この照会言語は、CIM オブジェクトモデルを SQL テーブルにマップする。
`getClass`	指定された CIM オブジェクトパスの CIM クラスを取得する。
`getInstance`	指定された CIM オブジェクトパスの CIM インスタンスを取得する。
`getProperty`	指定されたプロパティの値を返す。
`getQualifierType`	指定された CIM オブジェクトパスの修飾子型を取得する。
`invokeMethod`	指定されたオブジェクトに対して指定されたメソッドを実行する。メソッドは、メソッド名、戻り値のデータ型、およびメソッド内のパラメータを含む宣言である。
`references`	指定された CIM クラスまたはインスタンスを参照する関連を取得する。
`referenceNames`	指定された CIM クラスまたはインスタンスを参照する関連の名前を取得する。
`setClass`	指定された CIM クラスが存在する場合は、それを更新する。その CIM クラスが存在しない場合は、エラーを返す。
`setInstance`	指定された CIM インスタンスが存在する場合は、それを更新する。その CIM インスタンスが存在しない場合は、エラーを返す。
`setProperty`	指定されたプロパティに、指定された値を設定する。

次の表で、`com.sun.wbem.client` パッケージに含まれているインスタンスについて説明します。

表 3–4　com.sun.wbem.client パッケージに含まれているインスタンス

インタフェース	説明
CIMOMHandle	CIM Object Manager への参照をクライアントに提供する。このインタフェースには、クライアントが CIM Object Manager との間でデータを転送するときに使用するメソッドが含まれている。
ProviderCIMOMHandle	CIM Object Manager への参照をプロバイダに提供する。このインタフェースには、プロバイダが CIM Object Manager との間でデータを転送するときに使用するメソッドが含まれている。

次の表で、ProviderCIMOMHandle インタフェースに含まれているメソッドについて説明します。

表 3–5　ProviderCIMOMHandle インタフェースのメソッド

メソッド	説明
decryptData	指定された文字列の値を認証セッションキーを使って復号化する (暗号化されている場合)。
getCurrentAuditId	リモートクライアント接続を識別するためのレコードを監査するときに使用されるセッション識別子 (一般には固有のもの) を返す。
getCurrentRole	現在の認証済みユーザーが担う現在の役割を返す。
getCurrentUser	プロバイダが呼び出される原因になった現在のユーザーを返す。
getInternalProvider	内部インスタンスプロバイダへの参照を返す。この参照を使えば、プロバイダの静的インスタンスの情報を格納できる。

プロバイダ API パッケージ

プロバイダ API パッケージ (com.sun.wbem.provider と com.sun.wbem.provider20) には、CIM Object Manager とオブジェクトプロバイダが相互に通信するときに使用するインタフェースが含まれています。プロバイダは、これらのインタフェースを使って CIM Object Manager の動的データを提供することができます。

クライアントアプリケーションが CIM Object Manager に動的なデータを要求する場合、CIM Object Manager はこれらのインタフェースを使用して要求をプロバイダに渡します。プロバイダは、CIM Object Manager の要求に応答して次に示す機能を実行するクラスです。

- 管理対象デバイスからの情報を CIM Java クラスに対応付ける
  - デバイスから情報を取得する
  - 取得した情報を、CIM Java クラスの形式で CIM Object Manager に渡す
- CIM Java クラスからの情報を、管理対象デバイスの形式に対応付ける
  - CIM Java クラスから必要な情報を取得する
  - 取得した情報を、デバイスに固有のデバイス形式で渡す

次の表で、Provider パッケージ内のインタフェースについて説明します。

表 3–6 com.sun.wbem.provider インタフェース

インタフェース	説明
CIMProvider	すべてのプロバイダによって実装される基底インタフェース。
InstanceProvider	インスタンスプロバイダによって実装されるインタフェース。インスタンスプロバイダは、クラスの動的インスタンスを提供する。このインタフェースは非推奨。代わりに com.sun.wbem.provider20 の InstanceProvider インタフェースを使用してください。

表 3–6　com.sun.wbem.provider インタフェース　続く

インタフェース	説明
MethodProvider	メソッドプロバイダによって実装されるインタフェース。メソッドプロバイダは、CIM クラスの全メソッドの実装を提供する。
PropertyProvider	プロパティプロバイダによって実装されるインタフェース。プロパティプロバイダは、動的プロパティの検出と更新に使用される。動的データは、CIM Object Manager Repository には格納されない。

次の表で、com.sun.wbem.provider20 パッケージに含まれているインタフェースについて説明します。

表 3–7　com.sun.wbem.provider20 のインタフェース

インタフェース	説明
AssociatorProvider	動的関連のプロバイダによって実装されるインタフェース。
Authorizable	CIM Object Manager にユーザー認証を依頼するのではなく、認証検査を独自に行うプロバイダによって実装されるインタフェース。
InstanceProvider	インスタンスプロバイダによって実装されるインタフェース。インスタンスプロバイダはクラスの動的インスタンスを提供する。

第 4 章

クライアントアプリケーションの作成

この章では、クライアントアプリケーションプログラミングインタフェース (クライアント API) を使用してクライアントアプリケーションを作成する方法について説明します。

- 概要
- クライアントの接続の開始と終了
- インスタンスの処理
- オブジェクトの列挙
- 照会
- 関連
- メソッドの呼び出し
- クラス定義の検出
- 例外の処理
- 高度なプログラミング
- プログラム例

CIM API およびクライアント API についての詳細は、Javadoc リファレンスページを参照してください。

概要

WBEM (Web-Based Enterprise Management) アプリケーションは、Sun WBEM SDK API を使用して CIM オブジェクトを操作する標準的な Java プログラムです。クライアントアプリケーションは、一般に CIM API を使用してオブジェクト (ネームスペース、クラス、インスタンスなど) を構築し、続いてそのオブジェクトを初期化します。その後、クライアント API を使用してオブジェクトを CIM Object Manager に渡し、WBEM の処理 (CIM ネームスペース、クラス、またはインスタンスの作成など) を要求します。

クライアントアプリケーションの処理手順

Sun WBEM SDK アプリケーションは通常、次の手順で処理を行います。

1. CIM Object Manager に接続します (`CIMClient`)。

   クライアントアプリケーションは、WBEM のオペレーション (CIM クラスの作成や CIM インスタンスの更新など) を実行する必要があるたびに、CIM Object Manager に接続します。

2. 1 つ以上の API を使用してプログラミング作業を行います。

   CIM Object Manager への接続が完了すると、プログラムは API を使用して処理を要求します。

3. CIM Object Manager へのクライアント接続を閉じます (`close`)。

   アプリケーションは、終了時に現在のセッションを閉じる必要があります。`CIMClient` インタフェースを使用して、現在のクライアントセッションを閉じてクライアントセッションが使用しているリソースをすべて解放します。

例 — 一般的な Sun WBEM SDK アプリケーション

例 4–1 に、すべてのデフォルト値を使用して CIM Object Manager に接続する簡単なアプリケーションを示します。このプログラムは、クラスを取得し、そのクラス内のインスタンスを列挙して出力します。

例 4–1　一般的な Sun WBEM SDK アプリケーション

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;

/**
 * 指定されたクラスのすべてのインスタンスを返す。
 * このメソッドは、引数としてホスト名 (args[0]) と
 * リストするクラスの名前 (args[1]) を受け取る。
 */
public class WBEMsample {
    public static void main(String args[]) throws CIMException {
        CIMClient cc = null;
            try {
                /* args[0] にはネームスペースが入っている。
                ここでは、指定されたホスト上のデフォルトの
               root\cimv2 ネームスペースを指す
               CIM ネームスペース (cns) を作成する。 */
         CIMNameSpace cns = new CIMNameSpace(args[0]);
         /* CIM Object manager に接続し、ネームスペースを含む
        ネームスペースオブジェクトを渡す。 */
         cc = new CIMClient(cns, "root", "root_password");
         /* クラス名から CIMObjectPath を作成する。 */
         CIMObjectPath cop = new CIMObjectPath(args[1]);
         /* クラスを取得する (修飾子、クラス起点、
               プロパティなど)。*/
               cc.getClass(cop, true, true, true, null);
                // このクラスに属するすべてのインスタンス名を返す。
                Enumeration e = cc.enumerateInstanceNames(cop);
                while(e.hasMoreElements()) {
                    CIMObjectPath op = (CIMObjectPath)e.nextElement();
                    System.out.println(op);
                  } // while の終り
            } catch (Exception e) {
                System.out.println("Exception: "+e);
            }
            if(cc != null) {
               cc.close();
            }
     } // main の終り
 } // WBEMsample の終り
```

一般的なプログラミング作業

CIM Object Manager への接続が完了すると、クライアントアプリケーションは API を使用して処理を要求します。プログラムの機能セットは、どの処理を要求すべきかを決定します。次に、ほとんどのプログラムが実行する一般的な処理を示します。

- インスタンスの処理 (作成、削除、および更新)
- オブジェクトの列挙
- メソッドの呼び出し
- クラス定義の検出
- エラー処理

アプリケーションは、次の処理を実行する場合もあります。

- ネームスペースの作成
- ネームスペースの削除
- クラスの作成
- クラスの削除
- 修飾子の処理

クライアント接続の開始と終了

アプリケーションは、最初に CIM Object Manager に対しクライアントセッションを開きます。WBEM クライアントアプリケーションは、CIM Object Manager にオブジェクト管理サービスを要求します。クライアントと CIM Object Manager は、同じホスト上でも異なるホスト上でも動作します。同じ CIM Object Manager に対して、複数のクライアントから接続できます。

この節では、ネームスペースの基本概念と、次に示すクラスとメソッドの使用方法を説明します。

- CIM Object Manager に接続するための `CIMClient` クラス
- クライアント接続を閉じるための `close` メソッド

ネームスペースの使用

アプリケーションを作成する前に、ネームスペースの CIM 概念を理解する必要があります。ネームスペースは、ディレクトリに似た構造を持ち、ほかのネームスペース、クラス、インスタンス、および修飾子のデータ型を含むことができます。ネームスペース内のオブジェクトの名前は固有にする必要があります。オペレーションはすべて、ネームスペース内で行われます。Solaris WBEM Services をインストールすると、次に示す 2 つのネームスペースが作成されます。

- `root\cimv2` – Solaris WBEM Services がインストールされているシステム上のオブジェクトを表すデフォルトの CIM クラスが含まれます。これは、デフォルトネームスペースです。
- `root\security` – セキュリティ関連のクラスが含まれます。

CIM Object Manager に接続する場合、アプリケーションはデフォルトのネームスペース (`root\cimv2`) に接続するか、別のネームスペース (`root\security` や独自に作成したネームスペースなど) を指定する必要があります。

特定のネームスペースで CIM Object Manager への接続が完了すると、その後の処理はすべてそのネームスペース内で発生します。ネームスペースに接続すると、そのネームスペース内のすべてのクラスやインスタンス (存在する場合) にアクセスできるだけでなく、そのネームスペース内のすべてのネームスペースにもアクセスできます。たとえば、`root\cimv2` ネームスペースに `child` というネームスペースが作成されている場合、`root\cimv2` に接続することによって、`root\cimv2` ネームスペースと `root\cimv2\child` ネームスペース内のすべてのクラスやインスタンスにアクセスできます。

アプリケーションは、ネームスペース内で別のネームスペースに接続できます。これは、ディレクトリ内でサブディレクトリに位置を変更するのに似ています。アプリケーションが別のネームスペースに接続すると、その後の処理はすべてその新しいネームスペース内で発生します。たとえば、`root\cimv2\child` への接続を新たに開くと、このネームスペース内のクラスやインスタンスにはアクセスできますが、親ネームスペース `root\cimv2` 内のクラスやインスタンスにはアクセスできません。

CIM Object Manager への接続

クライアントアプリケーションは、WBEM のオペレーション (CIM クラスの作成や CIM インスタンスの更新など) を実行する必要があるたびに、CIM Object Manager に接続します。アプリケーションは、`CIMClient` クラスを使用して CIM Object

Manager 上のクライアントのインスタンスを作成します。CIMClient クラスには、次の 3 つの引数を指定できます。

- ネームスペース

  このクライアント接続のために使用するホスト名とネームスペース名が入っている CIMNameSpace オブジェクトです。デフォルトはローカルホスト上の root\cimv2 です。

- ユーザー名

  有効な Solaris のユーザーアカウント名。CIM Object Manager は、CIM オブジェクトに対してどのタイプのアクセスを許可するかを決定するために、ユーザーのアクセス権を確認します。デフォルトのユーザーアカウントは guest です。guest アカウントでは、デフォルトで、すべてのネームスペースのすべての CIM オブジェクトへの読み取りアクセス権がユーザーに与えられます。

- パスワード

  ユーザーアカウントのパスワード。パスワードは、ユーザーの Solaris アカウントとして有効なパスワードにする必要があります。デフォルトのパスワードは guest です。

CIM Object Manager への接続が完了すると、その後の CIMClient オペレーションはすべて指定されたネームスペース内で発生します。

例 — CIM Object Manager への接続

次の例は、CIMClient インタフェースを使用して CIM Object Manager に接続する 2 つの方法を示しています。

例 4–2 では、アプリケーションはすべてデフォルト値を使用しています。つまり、デフォルトのユーザーアカウントとパスワード guest を使用し、ローカルホスト (クライアントアプリケーションが動作しているホスト) のデフォルトのネームスペース (root\cimv2) で動作している CIM Object Manager に接続します。

例 4–2 デフォルトのネームスペースへの接続

```
/* パスワード guest を持つユーザー guest として
ローカルホスト上の root\cimv2 ネームスペースに接続 */

cc = new CIMClient();
```

例 4–3 のアプリケーションは、ローカルホストのデフォルトネームスペース (root\cimv2) で動作している CIM Object Manager に接続し、root アカウントの UserPrincipal オブジェクトを作成します。このオブジェクトは、デフォルトネームスペース内のすべての CIM オブジェクトに対する読み取り/書き込みアクセス権を備えています。

例 4–3 root アカウントへの接続

```
{
    ...

    root としてのホスト。2 つの null 文字列によって初期化される
    ネームスペースオブジェクトを作成する。2 つの null 文字列は
    デフォルトのホスト (ローカルホスト) とデフォルトの
    ネームスペース (root\cimiv2) を表す */

    CIMNameSpace cns = new CIMNameSpace("", "");

    UserPrincipal up = new UserPrincipal("root");
    PasswordCredential pc = new PasswordCredential("root_password");
    /* root パスワードを使い、root として
    ネームスペースに接続する */

    CIMClient cc = new CIMClient(cns, up, pc");
    ...
}
```

例 4–4 では、アプリケーションはホスト happy 上のネームスペース A に接続します。アプリケーションは、初めにこのネームスペースの文字列名 (A) を含むためにネームスペースのインスタンスを作成します。続いて、CIMClient クラスを使用して CIM Object Manager に接続し、ネームスペースオブジェクト、ユーザー名、およびホスト名を渡します。

例 4–4 デフォルト以外のネームスペースへの接続

```
{
    ...
    /* ホスト happy 上のA (ネームスペース名) に
    よって初期化されるネームスペースオブジェクトを作成する */
    CIMNameSpace cns = new CIMNameSpace("happy", "A");
    UserPrincipal up = new UserPrincipal("Mary");
    PasswordCredential pc = new PasswordCredential("marys_password");

    // このネームスペースにユーザー Mary として接続
    cc = new CIMClient(cns, "Mary", "marys_password");
    ...

}
```

例 4–5　RBAC の役割としての認証

ユーザーの役割を認証するには、SolarisUserPrincipal と SolarisPasswordCredential クラスを使用する必要があります。次のコード例では、Mary の役割を Admin として認証します。

```
{
...
CIMNameSpace cns = new CIMNameSpace("happy", "A");
SolarisUserPrincipal sup = new SolarisUserPrincipal("Mary", "Admin");
SolarisPasswordCredential spc = new
        SolarisPasswordCredential("marys_password", "admins_password");
CIMClient cc = new CIMClient(cns, sup, spc);
```

クライアント接続の終了

アプリケーションは、現在のクライアントセッションの終了時にセッションを閉じる必要があります。現在のクライアントセッションを閉じてこのセッションによって使用されているリソースをすべて解放するには、close メソッドを使用します。次に、クライアント接続を閉じるコード例を示します。インスタンス変数 *cc* は、このクライアント接続をしたインスタンスを表します。

```
cc.close();
```

インスタンスの処理

この節では、CIM インスタンスの作成、削除、および更新 (1 つ以上のインスタンスのプロパティ値の取得と設定) の各方法について説明します。

インスタンスの作成

既存のクラスのインスタンスを作成するには、newInstance メソッドを使用します。既存のクラスがキープロパティを持つ場合、アプリケーションはそのプロパティを固有の値に設定する必要があります。インスタンスは、必要に応じてそのクラスに定義されていない別の修飾子を定義することもできます。それらの修飾子をインスタンスまたは特定のインスタンスプロパティ用に定義できますが、クラス宣言内で定義する必要はありません。

アプリケーションは、クラスに定義されている一連の修飾子を getQualifiers メソッドを使用して取得できます。

例 — インスタンスの作成

例 4–6 のコードセグメントは、newInstance メソッドを使用して、Solaris_Package クラスの CIM インスタンス (Solaris パッケージなど) を表す Java クラスを作成します。

例 4–6 インスタンスの作成 (newInstance())

```
...
{
/*ローカルホストの root\cimv2 ネームスペースの
CIM Object Manager に接続。root\cimv2namespace 内のオブジェクトに
対する書き込みアクセス権を持つアカウントのユーザー名とパスワードを
指定する */

CIMClient cc = new CIMClient(cns, "root", "root_password");

// Solaris_Package クラスを取得
cimclass = cc.getClass(new CIMObjectPath("Solaris_Package"), true, true, true, null);

/* プロパティのデフォルト値を使用して生成された Solaris_Package
 クラスの新しいインスタンスを作成。このクラスのプロバイダが
 デフォルト値を指定しない場合、プロパティの値は NULL であり、
 明示的に設定する必要がある。*/
 ci = cimclass.newInstance();
}
...
```

インスタンスの削除

インスタンスの削除には、deleteInstance メソッドを使用します。

例 — インスタンスの削除

例 4–7 は、クライアントアプリケーションを CIM Object Manager に接続し、次に示すインタフェースを使用してクラスのインスタンスをすべて削除します。

- CIMObjectPath – 削除されるオブジェクトの CIM オブジェクトパスを含むオブジェクトを構築する
- enumInstance – インスタンスおよびそのサブクラスのすべてのインスタンスを取得する
- deleteInstance – 各インスタンスを削除する

例 4–7　インスタンスの削除 (deleteInstance)

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;

/**
 * このプログラム例では、必須の 4 つのコマンド行引数を受け取り、
 * 指定されたクラスとそのサブクラスのすべてのインスタンスを削除する。
 * このプログラムでは、指定されたネームスペースへの書き込みアクセス権
 * を持つアカウントのユーザー名とパスワードを指定する必要がある
 * /
public class DeleteInstances {
    public static void main(String args[]) throws CIMException {

  // CIMClient クラスのインスタンスを初期化する
  CIMClient cc = null;

  // 4 つのコマンド引数が必要。すべてが入力されない場合は、コマンド文字列を出力する

  if(args.length != 4) {
       System.out.println("Usage: DeleteClass host className username password");
       System.exit(1);
   }
   try {

     /**
      * ネームスペースオブジェクト (cns) を作成する。これには、コマンド行から
      * 入力されたホスト名 (args[0]) が格納される。
      */
     CIMNameSpace cns = new CIMNameSpace(args[0]);

     /**
      * CIM Object Manager に接続し、ネームスペース
      * オブジェクト (cns) と、コマンド行から入力されたユーザー名
      * (args[2]) とパスワード (args[3]) を渡す
      */

     cc = new CIMClient(cns, args[2], args[3]);

     /**
      * 削除するクラスの CIM オブジェクトパス (コマンドから入力された
      * args[1] ) を持つオブジェクトを構築する。
      */

     CIMObjectPath cop = new CIMObjectPath(args[1]);

     /**
      * クラスとそのクラスのすべてのサブクラスのインスタンスオブジェクトパス
      * を列挙する。インスタンスオブジェクトパスは、CIM Object Manager が
      * このインスタンスを見つけるときに参照として使用される。
      */
```

```
        Enumeration e = cc.enumerateInstanceNames(cop);

        /**
         * 列挙内のすべてのインスタンスオブジェクトパスを 1 つずつ処理する。
         * その中で、列挙された各インスタンスオブジェクトパスを格納する
         * オブジェクトを構築し、そのインスタンスを出力し、削除する。
         */

        while(e.hasMoreElements()) {
            CIMObjectPath op = (CIMObjectPath)e.nextElement();
           System.out.println(op);
           cc.deleteInstance(op);
           }
      } catch (Exception e) {
            System.out.println("Exception: "+e);
      }
      if(cc != null) {
            cc.close();
      }
      }
}
```

インスタンスの取得と設定

アプリケーションが、 CIM Object Manager から CIM インスタンスを取得する場合、`getInstance` メソッドがよく使用されます。クラスのインスタンスが作成されるときに、インスタンスはその派生元クラスとそのクラス階層にあるすべての親クラスのプロパティを継承します。`getInstance` メソッドはブール値引数 *localOnly* を受け取り、localOnly が真である場合、指定されたインスタンスによって継承されたプロパティ以外のプロパティだけを返します。これらのプロパティは、そのインスタンス自体によって定義されたものです。*localOnly* が偽の場合は、そのクラスのすべてのプロパティが返されます (つまり、インスタンス自体によって定義されたプロパティと、クラス階層にあるすべての親クラスから継承されたすべてのプロパティ)。

新しいインスタンスを作成する場合は、`CIMClass` クラスの `CIMInstance` メソッドを使ってインスタンスをローカルシステム上に作成します。そして、`CIMClient.setInstance` メソッドを使ってネームスペース内の既存のインスタンスを更新するか、`CIMClient.createInstance` メソッドを使って新しいインスタンスをネームスペースに追加します。

例 — インスタンスの取得

例 4–8 のコードセグメントは、特定のシステム上のすべてのプロセスを表示します。この例では、`enumerateInstanceNames` メソッドを使用して `CIM_Process`

クラスのインスタンス名を取得します。このコードを Microsoft Windows 32 システムで実行すると、Windows 32 のプロセスが返されます。このコードを Solaris システムで実行すると、Solaris プロセスが返されます。

例 **4–8**　クラスインスタンスの取得 (getInstance)

```
...
{
//ネームスペース cns を作成
CIMnameSpace cns = new CIMNameSpace();

//CIM Object Manager 上で cns ネームスペースに接続
cc = new CIMClient(cns, "root", "root_password");

/* CIM_Process クラスの CIM オブジェクトパスを
CIM Object Manager に渡す (このクラスのインスタンスを取得する)。 */

CIMObjectPath cop = new CIMObjectPath("CIM_Process");

/* CIM Object Manager が、オブジェクトパスの列挙
(CIM_Process クラスのインスタンス名) を返す。 */
Enumeration e = cc.enumerateInstanceNames(cop);

/* インスタンスオブジェクトパスを含む列挙のサイズ分だけ繰り返す。
各オブジェクト名によって参照されるインスタンスを取得するには、
CIM Client の getInstance クラスを使用する。 */

while(e.hasMoreElements()) {
    CIMObjectPath op = (CIMObjectPath)e.nextElement();
    // インスタンスを取得する。インスタンスに対してローカルである
  // プロパティだけを返す (localOnly が真)
  CIMInstance ci = cc.getInstance(op, true);
  }
...
```

例 — プロパティの取得

例 4–9 は、すべての Solaris プロセスの lockspeed プロパティの値を出力します。このコードセグメントは、次のメソッドを使用します。

- `enumInstances` – Solaris プロセッサのすべてのインスタンス名を取得する
- `getProperty` – 各インスタンスの lockspeed の値を取得する
- `println` – lockspeed の値を出力する

例 **4–9**　プロセッサ情報の出力 (getProperty)

```
...
{
/* オブジェクト (CIMObjectPath) を作成し、Solaris_Processor
```

```
クラスの名前を格納する */

CIMObjectPath cop = new CIMObjectPath("Solaris_Processor");

/* Solaris_Processor クラスとそのすべてのサブクラス (cc.DEEP) の
インスタンスの名前が含まれている列挙を CIM Object Manager が返す */

Enumeration e = cc.enumInstances(cop, cc.DEEP);

/* インスタンスオブジェクトパスの列挙のサイズ分だけ繰り返す。
getProperty メソッドを使って、各 Solaris プロセッサの
lockspeed 値を取得する */

while(e.hasMoreElements()) {
   CIMValue cv = cc.getProperty(e.nextElement(), "lockspeed");
     System.out.println(cv);
}
...
}
```

例 — プロパティの設定

例 4–10のコードセグメントでは、すべての Solaris プロセッサの仮定の lockspeed 値を設定します。この例では、次のメソッドを使用します。

- `enumInstances` – Solaris プロセッサのすべてのインスタンスの名前を取得する。
- `setProperty` – 各インスタンスの lockspeed 値を設定する。

例 4–10 プロセッサ情報の設定 (`setProperty`)

```
...
{
    /* オブジェクト (CIMObjectPath) を作成し、Solaris_Processor
    クラスの名前を格納する*/

    CIMObjectPath cop = new CIMObjectPath("Solaris_Processor");

    /* Solaris_Processor クラスとそのすべてのサブクラスの
    インスタンスの名前が含まれている列挙を CIM Object Manager が返す */

    Enumeration e = cc.enumerateInstanceNames(cop);

    /* インスタンスオブジェクトパスの列挙のサイズ分だけ繰り返す。
    その中で setProperty メソッドを使って、各 Solaris プロセッサの
    lockspeed 値に 500 を設定する。 */

    for (; e.hasMoreElements(); cc.setProperty(e.nextElement(), "lockspeed",
        new CIMValue(new Integer(500))));
```

```
...
}
```

例 — インスタンスの設定

例 4–11 のコードセグメントは、CIM インスタンスを取得してそのプロパティ値の 1 つを変更し、変更後のインスタンスを CIM Object Manager に渡します。

CIM プロパティは、CIM クラスの特性を記述するために使用される値です。プロパティは、プロパティ値を設定する機能と、プロパティ値を取得する機能の組み合わせと考えることができます。

例 4–11 インスタンスの設定 (setInstance)

```
...
{
    // オブジェクトパスを作成する (CIM 名「myclass」を持つ
    // オブジェクト)
    CIMObjectPath cop = new CIMObjectPath("myclass");
    /* 列挙内の各インスタンスオブジェクトパスのインスタンスを取得し、
    各インスタンスのプロパティ値 b に 10 を設定し、
    更新済みインスタンスを CIM Object Manager に渡す。 */

    while(e.hasMoreElements()) {
      CIMInstance ci = cc.getInstance(CIMObjectPath)(e.nextElement(),
                true, true, true, null);
        ci.setProperty("b", new CIMValue(new Integer(10)));
    cc.setInstance(new CIMObjectPath(),ci);
  }
}
...
```

ネームスペース、クラス、インスタンスの列挙

列挙とはオブジェクトの集合です。列挙は一度に 1 つずつ取り出すことができます。Sun WBEM SDK には、ネームスペース、クラス、インスタンスを列挙する API があります。

次の各例は、列挙メソッドを使ってネームスペース、クラス、インスタンスを列挙する方法を示したものです。

詳細 (deep) 列挙と簡易 (shallow) 列挙

列挙メソッドは、値として *deep* または *shallow* を持つブール値引数を受け取ります。*deep* および *shallow* の動作は、表 4–1 に示すように、使用するメソッドによって異なります。

表 4–1 詳細列挙と簡易列挙

メソッド	***deep***	***shallow***
`enumNameSpace`	列挙するネームスペース内のネームスペース階層全体を返す。	列挙するネームスペース内の第一レベルの子を返す。
`enumClass`	列挙するクラスのすべてのサブクラスを返すが、クラス自体は返さない。	このクラスの直接のサブクラスを返す。
`enumInstances`	クラスのインスタンスとそのクラスのサブクラスのすべてのインスタンスを返す。	そのクラスのインスタンスを返す。

クラスやインスタンスのデータを取得

次の列挙メソッドは、クラスやインスタンスのデータを返します。

- `enumInstances`(*CIMObjectPath path, boolean deep, boolean localOnly*) – *Path* で指定されたクラスのインスタンスを返します。このメソッドは、deep が真である場合は、指定されたクラスとこのクラスから派生したすべてのクラスの各インスタンスを返し、*shallow* が真である場合は、指定されたクラスの各インスタンスを返します。

  クラスのインスタンスが作成されると、インスタンスは、その派生元クラスとそのクラス階層にあるすべての親クラスのプロパティを継承します。`enumInstances` は、*localOnly* が真である場合は、継承されたプロパティ以外のプロパティだけを返し、*localOnly* が偽である場合は、クラスのすべてのプロパティを返します。

- `enumClass`(*CIMObjectPath path, boolean deep, boolean localOnly*) – *Path* で指定されたクラスの派生クラス (クラスの名前だけでなく、クラス全体) を返します。この

メソッドは、*deep* が真である場合は、列挙するクラスから派生したすべてのクラスを返し、*shallow* が真である場合は、列挙するクラスの第一レベルの子だけを返します。

作成されたクラスは、その派生元クラスとそのクラス階層のすべての親クラスのメソッドとプロパティを継承します。このメソッドは、*localOnly* が真である場合は、継承されたプロパティとメソッド以外のプロパティとメソッドを返し、*localOnly* が偽である場合は、そのクラス内のすべてのプロパティを返します。

クラス名やインスタンス名の取得

CIM WorkShop は、列挙メソッドを使ってクラスやインスタンスの名前を返すアプリケーションの一例です。オブジェクト名のリストを取得すれば、オブジェクトのインスタンスや、プロパティ、その他の情報を取得できます。

次の各列挙メソッドは、列挙するクラスやインスタンスの名前を返します。

- `enumerateInstanceNames`(*CIMObjectPath path*) — 指定されたクラスの各インスタンスの名前を返します。
- `enumerateClassNames`(*CIMObjectPath path, boolean deep*) — *Path* で指定されたクラスの派生クラスの名前を返します。このメソッドは、*deep* が真である場合は、列挙するクラスから派生したすべてのクラスの名前を返し、*shallow* が真である場合は、列挙するクラスの第一レベルの子の名前だけを返します。

例 — ネームスペースの列挙

例 4–12 のプログラムは、CIM クライアントクラスの `enumNameSpace` メソッドを使用して、ネームスペースおよびその中に含まれるすべてのネームスペースの名前を出力します。

例 4–12 ネームスペースの列挙 (`enumNameSpace`)

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
```

```
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import java.util.Enumeration;

/ **
  * このプログラムは、CIMObjectPath(cop) で指定されるネームスペース、
  * およびそのネームスペース内のすべてのネームスペースの
  * リストを取得するために CIMClient.DEEP を条件として、ネームスペース引数を取り、
  * CIMClient の enumNameSpace インタフェースを呼び出す。
  * 続いて、指定されたネームスペース名を
  * 出力する (CIMClient.SHALLOW)。
 /**

public class EnumNameSpace {

  // EnumNameSpace が、引数の文字列を取る。
  public static void main (String args[ ]) {
   CIMClient cc = null;

   try {
   // 引数として渡されるネームスペースのネームスペースオブジェクトを作成
   CIMNameSpace cns = new CIMNameSpace(args[0], "");

   // 引数として渡されるネームスペース内の CIM Object Manager に接続
   CIMClient cc = new CIMClient(cns);

   // 現在のホスト上のネームスペース名を保存するためにオブジェクトパスを作成
   CIMObjectPath cop = new CIMObjectPath("",args[1]);

   // ネームスペースおよびその中に含まれるすべてのネームスペースを列挙
      // (CIMClient.DEEP に deep が 設定される)
   Enumeration e = cc.enumNameSpace(cop, CIMClient.DEEP);

   // ネームスペースの表示を繰り返し、それぞれの名前を出力
   for (; e.hasMoreElements();
      System.out.println(e.nextElement()));
      System.out.println("++++++");
   // CIMClient.SHALLOW でネームスペースの表示を繰り返し、
      // それぞれの名前を出力
   e = cc.enumNamesSpace(cop, CIMClient.SHALLOW);
   for (; e.hasMoreElements();
     System.out.println(e.nextElement()));
 } catch (Exception e) {
    System.out.println("Exception: "+e);
   }

// クライアント接続が開かれている場合、接続を閉じる。
  if(cc != null) {
            cc.close();
     }

  }
}
```

コード例 — クラス名の列挙

Java GUI アプリケーションでは、例 4–13 のコードセグメントを使ってクラスやサブクラスのリストをユーザーに表示することがあります。

例 4–13 クラス名の列挙 (enumClass)

```
...
{
    /* CIMObjectPath オブジェクトを作成し、列挙する CIM クラスの
    名前 (myclass) で初期化する */
    CIMObjectPath cop = new CIMObjectPath(myclass);

    /* この列挙には、列挙するクラス内のすべてのクラスとサブクラスの
    名前が含まれる */
    Enumeration e = cc.enumClass(cop, cc.DEEP);
}
...
```

アプリケーションでは、例 4–14 のコードセグメントを使ってクラスやそのサブクラスの内容を表示することがあります。

例 4–14 クラスデータの列挙 (enumClass)

```
...
{
    /* CIMObjectPath オブジェクトを作成し、列挙する CIM クラスの
    名前 (myclass) で初期化する  */

    CIMObjectPath cop = new CIMObjectPath(myclass);

    /* この列挙には、列挙するクラス内のすべてのクラスとサブクラスが
    含まれる (cc.DEEP)。この列挙では、各クラスやサブクラスから
    継承されたメソッドやプロパティ以外のものだけが返される
    (localOnly が真) */

    Enumeration e = cc.enumClass(cop, cc.DEEP, true);
}
...
```

例 4–15 のサンプルプログラムでは、クラスおよびインスタンスを詳細列挙および簡易列挙します。さらにこの例では、*localOnly* フラグを使って、(クラスやインスタンスの名前ではなく) クラスやインスタンスのデータを返します 。

例 4–15 クラスおよびインスタンスの列挙

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
```

```
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;

/**
 * この例では、クラスやインスタンスを列挙する。
 * この例では、コマンド行から入力されたクラスを詳細列挙
 * および簡易列挙する。さらに、localOnly フラグを使って
 * クラスやインスタンスの詳細を返す。
 */
 public class ClientEnum {
     public static void main(String args[]) throws CIMException {
       CIMClient cc = null;
      CIMObjectPath cop = null;
      if(args.length != 2) {
         System.out.println("Usage: ClientEnum host className");
         System.exit(1);
        }
      try {
        // ホスト名 (コマンド行から入力された args[0]) を含む
        // CIMNameSpace オブジェクトを作成する。
           CIMNameSpace cns = new CIMNameSpace(args[0]);

        // 指定されたホスト (args[0]) 上の
        // CIM Object Manager へのクライアント接続を作成する
           cc = new CIMClient(cns);

           // コマンド行から入力されたクラス名を取得する
           cop = new CIMObjectPath(args[1]);

           // クラスを詳細列挙し、クラス名を返す
        // returns the class names.
           Enumeration e = cc.enumClass(cop, cc.DEEP);

           // 列挙するクラスのすべてのサブクラスの名前を出力する
           for (; e.hasMoreElements(); System.out.println(e.nextElement()));
               System.out.println("+++++");

           // クラスを簡易列挙し、クラス名を返す
        // returns the class names.
           e = cc.enumClass(cop, cc.SHALLOW);

           // 第一レベルのサブクラスの名前を出力する
           for (; e.hasMoreElements(); System.out.println(e.nextElement()));
               System.out.println("+++++");

           // クラスを簡易列挙し、
        // クラス名に加えクラスデータを
        // 返す (localOnly が真)
           e = cc.enumClass(cop, cc.SHALLOW, true);

           // 第一レベルのサブクラスの詳細を出力する
           for (; e.hasMoreElements(); System.out.println(e.nextElement()));
               System.out.println("+++++");

        // クラスのインスタンスを詳細列挙し、
        // インスタンスの名前を返す
           e = cc.enumInstances(cop, cc.DEEP);
```

```
            // クラスとそのサブクラスのすべてのインスタンスの名前を出力する
            for (; e.hasMoreElements(); System.out.println(e.nextElement()));
                System.out.println("+++++");

        // クラスのインスタンスを詳細列挙し、
        // インスタンス名に加え、実際のインスタンスデータを
        // 返す (localOnly が真)
            e = cc.enumInstances(cop, cc.DEEP);

            // クラスとそのサブクラスのすべてのインスタンスの詳細を出力する
            for (; e.hasMoreElements(); System.out.println(e.nextElement()));
                System.out.println("+++++");

        // クラスのインスタンスを簡易列挙し、
        // インスタンスの名前を返す
            e = cc.enumInstances(cop, cc.SHALLOW);

        // クラスのインスタンスの名前を出力する
            for (; e.hasMoreElements(); System.out.println(e.nextElement()));
               System.out.println("+++++");
          } catch (Exception e) {
                System.out.println("Exception: "+e);
        }
       // セッションを閉じる
       if(cc != null) {
           cc.close();
       }
    }
}
```

照会

列挙 API は、クラスまたはクラス階層のすべてのインスタンスを返します。この場合、ユーザーは、インスタンス名またはインスタンスの詳細を返すように指定することができます。照会によって、照会文字列を指定することによって、検索範囲を狭めることができます。ユーザーは、特定のクラスまたは特定のネームスペース内のすべてのクラスにおいて、指定の照会と一致するインスタンスを検索できます。たとえば、`Storage_Capacity` プロパティに特定の値を持つ `Solaris_DiskDrive` クラスを検索できます。

execQuery メソッド

execQuery メソッドは、照会文字列と一致する CIM インスタンスの列挙を検索します。照会文字列の形式は、WQL (WBEM Query Language) に準拠していなければなりません。

構文

execQuery メソッドの構文は、次のとおりです。

```
Enumeration execQuery(CIMObjectPath relNS, java.lang.String query, int ql)
```

execQuery メソッドは、次のパラメータを受け取り、CIM インスタンスの列挙を返します。

パラメータ	データ型	説明
relNS	CIMObjectPath	接続されているネームスペースとの相対的なネームスペース。たとえば、root ネームスペースに接続されている場合に root\cimv2 ネームスペース内のクラスを照会するには、新たに new CIMObjectPath("", "cimv2"); を渡す必要がある。
query	String	WQL (WBEM Query Language) に準拠した照会テキスト
ql	Integer constant	照会言語を指定する。現在サポートされている照会言語は WQL level 1 のみ

例

次の execQuery 呼び出しは、現在のネームスペースにある CIM_device クラスのすべてのインスタンスの列挙を返します。

```
cc.execQuery(new CIMObjectPath(), SELECT * FROM CIM_device, cc.WQL)
```

WQL の使用

WQL (WBEM Query Language) は、ANSI SQL (ANSI 構造化照会言語、ANSI Structured Query Language) のサブセットです。WQL には、Solaris で WBEM をサポートするためにセマンティクスの変更が加えられています。SQL とは異なり、このリリースの WQL は検索だけが可能な言語です。つまり、WQL では、情報の変更、挿入、削除を行うことはできません。

SQL はデータベースの照会を行うために作成された言語です。SQL では、データは、行列構造からなるテーブル (表) に格納されています。WQL は、CIM データモデルを使って格納されたデータを照会できるようになっています。CIM モデルでは、オブジェクトの情報は CIM クラスや CIM インスタンスに格納されています。CIM インスタンスには、名前、データ型、値からなるプロパティを持つことができます。WQL は CIM オブジェクトモデルを SQL テーブルにマップします。

表 4–2 SQL データと WQL データの対応

SQL	**WQL** での表現
テーブル	CIM クラス
行	CIM インスタンス
列	CIM プロパティ

サポートされる WQL キーワード

Sun WBEM SDK では、Level 1 WBEM SQL がサポートされます。Level 1 WBEM SQL では、join のない簡単な select 操作を行うことができます。次の表に、Sun WBEM SDK でサポートされる WQL キーワードを示します。

表 4–3　サポートされる WQL キーワード

キーワード	説明
AND	2 つのブール式を結合し、両方の式が TRUE なら TRUE を返す。
FROM	SELECT 文にリストされているプロパティを持つクラスを指定する。
NOT	NULL と共に使用される比較演算子。
OR	2 つの条件を結合する。1 つの文に複数の論理演算子が使用されていると、OR 演算子 (論理和演算子) は AND 演算子 (論理積演算子) より後に評価される。
SELECT	照会で使用されるプロパティを指定する。
WHERE	照会のスコープを狭める。

WQL の演算子

次の表に、SELECT 文の WHERE 節で使用できる標準の WQL 演算子を示します。

表 4–4　WQL 演算子

演算子	説明
=	等しい
<	より小さい
>	より大きい
<=	以下
>=	以上
<>	等しくない

データ照会の実行

データ照会とは、クラスのインスタンスを要求する文です。データ照会を発行するには、`execQuery` メソッドを使って WQL 文字列を CIM Object Manager に渡します。

SELECT 文

情報を検索する SQL 文は SELECT 文です。これには、WQL 固有の制約と拡張がいくつかあります。SQL SELECT 文は通常データベース環境でテーブルから特定の列を取り出すために使用されますが、WQL SELECT 文は単一クラスのインスタンスを取り出すために使用されます。WQL では、複数のクラスに渡る照会はサポートされません。

SELECT 文では、FROM 節に指定されたオブジェクトで検索するプロパティを指定します。

SELECT 文の基本的な構文は、次のとおりです。

`SELECT` *instance* `FROM` *class*

次の表に、SELECT 節の引数の使用例を示します。

表 4–5 SELECT 文

照会例	説明
SELECT * FROM *class*	指定されたクラスとそのすべてのサブクラスのすべてのインスタンスを選択する。
SELECT PropertyA FROM class	指定されたクラスとそのすべてのサブクラスのうち、`PropertyA` を持つクラスまたはサブクラスのインスタンスだけを選択する。
SELECT PropertyA, PropertyB FROM *class*	指定されたクラスとそのすべてのサブクラスのうち、`PropertyA` か `PropertyB` を持つクラスまたはサブクラスのインスタンスだけを選択する。

WHERE 節

WHERE 節を使うと、照会のスコープを狭めることができます。WHERE 節には、プロパティまたはキーワード、演算子、定数を指定します。WHERE 節には、あらかじめ定義された WQL 演算子の 1 つを必ず指定する必要があります。

WHERE 節を SELECT 文の後に追加する基本的な構文は、次のとおりです。

```
SELECT instance FROM class WHERE expression
```

expression には、プロパティまたはキーワード、演算子、定数を指定します。
WHERE 節を SELECT 文の後に追加するには、次のどちらかの形式を使用します。

```
SELECT instance FROM class [WHERE property operator constant]
```

```
SELECT instance FROM class [WHERE constant operator property]
```

WHERE 節は次の規則に従っている必要があります。

- 定数の値は、プロパティに対して適切なデータ型のものである。
- 演算子は、表 4–4 に示す WQL 演算子のいずれかでなければならない。
- 演算子のどちらかの側は、プロパティ名または定数でなければならない。
- 任意の算術式を指定することはできない。たとえば、次の照会では、ready 状態のプリンタを持つ `Solaris_Printer` クラスのインスタンスだけが返されます。

  ```
  SELECT * FROM Solaris_Printer WHERE Status = ''ready''
  ```

  次の照会は無効です。

  ```
  SELECT * FROM PhysicalDisk WHERE Partitions < (8 + 2 - 2)
  ```

WHERE 節中には、論理演算子やかっこ式を使って、プロパティ、演算子、定数からなる複数のグループを結合することができます。次の表に示すように、個々のグループは、AND、OR、NOT 演算子で結合されている必要があります。

表 4–6 論理演算子を使った照会

照会例	説明
SELECT * FROM Solaris_FileSystem WHERE Name= "home" OR Name= "files"	Name プロパティに `home` か `files` が設定されている `Solaris_FileSystem` クラスのすべてのインスタンスを検索する。
SELECT * FROM Solaris_FileSystem WHERE (Name = “home” OR Name = “files”) AND AvailableSpace > 2000000 AND FileSystem = “Solaris”	名前が `home` か `files` のディスクのうち、一定の使用可能な容量が残っており、かつ Solaris ファイルシステムを持つディスクを検索する。

関連

ここでは、関連という CIM 概念と、関連の情報を取得するときに使用する `CIMClient` メソッドについて説明します。

関連について

関連とは、コンピュータとそのシステムディスクなど、2 つ以上の管理リソース間の関係を表したものです。この関係は、関連修飾子を持つ特殊なクラス型である関連クラスで表されます。

さらに、関連クラスには、その個々の管理リソースを表す CIM インスタンスへの 2 つ以上の参照が含まれています。参照は REF キーワードで宣言される特殊なプロパティ型であり、これが他のインスタンスへのポインタであることを示します。参照は、個々の管理リソースが関連において担う役割を定義します。

次の図には、`Teacher` と `Student` という 2 つのクラスが示されています。これらのクラスは、`TeacherStudent` という関連によってリンクされています。`TeacherStudent` には、`Teaches` と `TaughtBy` という 2 つの参照がありま

す。Teaches は、Teacher クラスのインスタンスを参照するプロパティであり、TaughtBy は Student クラスのインスタンスを参照するプロパティです。

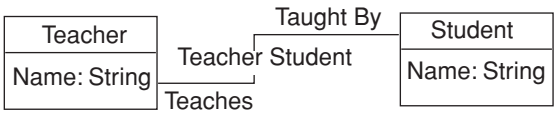

図 4–1　Teacher と Student の関連

参照の 1 つを削除する場合は、その前に関連を削除する必要があります。2 つまたはそれ以上のオブジェクト間の関連を追加または変更しても、オブジェクト自体には影響を与えません。

関連メソッド

CIMClient クラスの次のメソッドは、クライアントとインスタンス間の関連 (関係) に関する情報を返します。

表 4–7　CIMClient の関連メソッド

メソッド	説明
associators	指定された CIM クラスやインスタンスに関連付けられている CIM クラスやインスタンスを取得する。
associatorNames	指定された CIM クラスやインスタンスに関連付けられている CIM クラスやインスタンスの名前を取得する。
references	指定された CIM クラスやインスタンスを参照する関連を取得する
referenceNames	指定された CIM クラスやインスタンスを参照する関連の名前を取得する。

ソースクラスまたはソースインスタンスの指定

関連メソッドは、唯一の必須引数として CIMObjectPath を受け取ります。この引数は、検索したい関連や、関連付けされたクラスまたはインスタンスを持つソースの CIM クラスまたは CIM インスタンスの名前です。CIM Object Manager は、関連、関連付けされたクラス、またはインスタンスを見つけることができないと、何も返しません。

関連メソッドは、CIMObjectpath がクラスである場合は、関連付けされたクラスとそのクラスのサブクラスを返し、CIMObjectpath がインスタンスである場合は、関連付けされたインスタンスと各インスタンスの派生元クラスを返します。

モデルパスによるインスタンスの指定

インスタンスやクラスの名前を指定するには、そのモデルパスを指定する必要があります。クラスのモデルパスには、ネームスペースとクラスの名前が含まれています。インスタンスのモデルパスは、特定の管理リソースを固有に識別します。インスタンスのモデルパスには、ネームスペース名と、クラス名、キーが含まれています。キーは、管理リソースを固有に識別するために使用されるプロパティまたは一連のプロパティです。キープロパティは、キー修飾子で識別します。

モデルパス
\\myserver\\Root\cimv2\Solaris_ComputerSystem.Name=mycomputer:CreationClassName=Solaris_ComputerSystem は、3 つの部分から構成されています。

- \\myserver\Root\cimv2 – ホスト myserver 上のデフォルト CIM ネームスペース
- Solaris_ComputerSystem – このインスタンスの派生元クラスの名前
- Name=mycomputer, CreationClassName=Solaris_ComputerSystem – 「キープロパティ = 値」の形式による 2 つのキープロパティ

API によるインスタンスの指定

実際の使用では通常、あるクラスのすべてのインスタンスを enumInstances メソッドを使って列挙し、ループ構造で各インスタンスを処理します。ループでは、各インスタンスを関連メソッドに渡すことができます。次のコード例では、次のことを行なっています。

1. 現在のクラス (op) とそのクラスのサブクラス内のすべてのインスタンスを列挙する。
2. while ループを使って、個々のインスタンスを CIMObjectPath (op) にキャストする。
3. 各インスタンスを、associators メソッドへの最初の引数として渡す。

このコード例では、他のすべてのパラメータの値としては、 null または false を渡します。

例 **4–16** インスタンスを associators メソッドに渡す

```
{
   ...
   Enumeration e = cc.enumInstances(op, true);
   while (e.hasMoreElements()) {
       op = (CIMObjectPath)e.nextElement();
       Enumeration e1 = cc.associators(op, null, null,
               null, null, false, false, null);
   ...
   }
```

返されるクラスおよびインスタンスのフィルタリング (オプションの引数を使用)

関連メソッドは、返されるクラスやインスタンスをフィルタリングする次のオプション引数も受け取ります。すべての引数が処理されるまで、オプションの各パラメータ値は、その結果をフィルタリング処理のために次に続くパラメータに渡します。

1 つのオプション引数の値、またはオプション引数を組み合わせたものの値を渡すことができます。各パラメータには値を指定する必要があります。返されるクラスやインスタンスをフィルタリングする引数には、assocClass、 resultClass、role、 resultRole があります。これらの引数を使用すると、これらのパラメータに指定された値と一致するクラスやインスタンスだけが返されます。返されるクラスやインスタンスに含まれている情報をフィルタリングする引数には、includeQualifiers、includeClassOrigin、propertyList があります。

次の表に、関連メソッドへのオプション引数を示します。

表 4–8 関連メソッドへのオプション引数

引数	型	説明	値
`assocClass`	String	ソースの CIM クラスまたはインスタンスとこのタイプで関連付けられているターゲットオブジェクトを返す。Null の場合は、返すオブジェクトを関連でフィルタリングしない。	有効な CIM 関連クラス名または Null
`resultClass`	String	`resultClass` またはそのいずれかのサブクラスのインスタンスであるターゲットオブジェクト、または `resultClass` かそのいずれかのサブクラスと一致するオブジェクトを返す。	CIM クラスの有効な名前または Null
`role`	String	この関連でソースの CIM クラスまたはインスタンスが担っている役割を指定する。ソースオブジェクトがこの役割を担っている関連のターゲットオブジェクトを返す。	有効なプロパティ名または Null
`resultRole`	String	指定された役割をこの関連で担っているターゲットオブジェクトを返す。	有効なプロパティ名または Null
`includeQualifiers`	Boolean	true の場合、各ターゲットオブジェクトのすべての修飾子 (このオブジェクトと返されるプロパティのすべての修飾子) を返す。false の場合、修飾子を返さない。	True または False

表 4–8 関連メソッドへのオプション引数 続く

includeClassOrigin	Boolean	true の場合、返される各オブジェクトのすべての適切な要素に CLASSORIGIN 属性を含める。false の場合、CLASSORIGIN 属性を含めない。	True または False
propertyList	String array	このリストに指定されたプロパティの要素だけを含むオブジェクトを返す。空配列の場合は、返される各オブジェクトにプロパティは含まれない。NULL の場合は、返す各オブジェクトにすべてのプロパティが含まれる。無効なプロパティ名は無視する。 プロパティリストを指定する場合は、resultClass に Null 以外の値を指定する必要がある。	有効なプロパティ名の配列、空の配列、または Null

例 — associators メソッドと associatorNames メソッド

このセクションの各例は、associators メソッドや associatorNames メソッドを使って、次の図に示す Teacher クラスや Student クラスに関連付けられているクラスの情報を取得する方法を示したものです。associatorNames メソッドでは、引数 *includeQualifiers*、*includeClassOrigin*、*propertyList* は使用されません。インスタンスまたはクラスの名前だけ (内容全体ではなく) を返すメソッドには、これらの引数は関係ないからです。

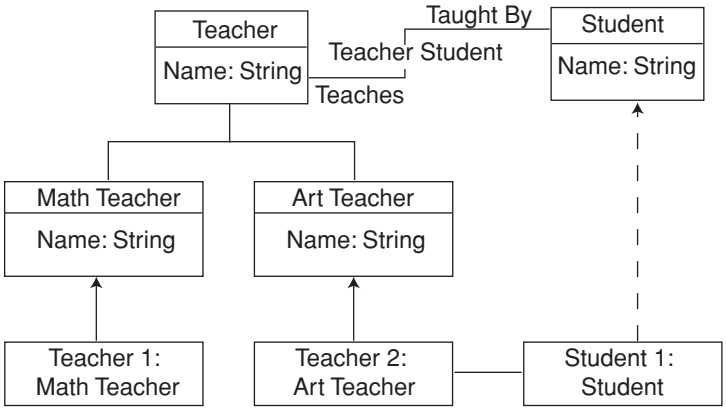

図 4–2　Teacher と Student の関連の例

表 4–9　associators メソッドと associatorNames メソッド

例	出力	説明
associators(Teacher, null, null, null, null, false, false, null)	Student クラス	関連付けられているクラスとそれらのサブクラスを返す。Student は、TeacherStudent 関連によって Teacher とリンクされている。
associators(Student, null, null, null, null, false, false, null)	Teacher クラス、MathTeacher クラス、 ArtTeacher クラス	関連付けられているクラスとそれらのサブクラスを返す。Teacher は、TeacherStudent 関連によって Student とリンクされている。MathTeacher と ArtTeacher は、Teacher から TeacherStudent 関連を継承する。

表 4–9　associators メソッドと associatorNames メソッド　続く

associatorNames (Teacher, null, null, null, null)	Student クラスの名前	関連付けられているクラスとそれらのサブクラスの名前を返す。Student は、TeacherStudent 関連によって Teacher とリンクされている。
associatorNames (Student, null, null, null, null)	Teacher、MatchTeacher、ArtTeacher のクラス名	関連付けられているクラスとそれらのサブクラスの名前を返す。Teacher は、TeacherStudent 関連によって Student とリンクされている。MathTeacher と ArtTeacherr は、Teacher から TeacherStudent 関連を継承する。

例 — references メソッドと referenceNames メソッド

このセクションの各例は、references メソッドや referenceNames メソッドを使って、図 4–2 に示す Teacher クラスと Student クラスの関連の情報を取得する方法を示したものです。 referenceNames メソッドでは、引数 *includeQualifiers*、*includeClassOrigin*、*propertyList* は使用されません。インスタンスまたはクラスの名前だけ (内容全体ではなく) を返すメソッドには、これらの引数は関係ないからです。

表 4–10　references メソッドと referenceNames メソッド

例	出力	説明
references(Student, null, null. false, false, null)	TeacherStudent	Student が関与している関連を返す。
references(Teacher, null, null. false, false, null)	TeacherStudent	Teacher が関与している関連を返す。
referenceNames(Teacher, null, null)	TeacherStudent クラスの名前	Teacher が関与している関連の名前を返す。

メソッドの呼び出し

プロバイダによってサポートされるクラス内のメソッドを呼び出すには、invokeMethod インタフェースを使用します。メソッドのシグニチャを取得するには、まず、そのメソッドが属するクラスの定義を取得する必要があります。invokeMethod インタフェースは、次の表に示す 4 つの引数を受け取ります。

表 4–11 invokeMethod へのパラメータ

パラメータ	データ型	説明
name	CIMObjectPath	インスタンスの名前。このインスタンスでメソッドを呼び出す必要がある。
methodName	String	呼び出すメソッドの名前。
inParams	Vector	メソッドに渡す入力パラメータ。
outParams	Vector	メソッドから受け取る出力パラメータ。

invokeMethod メソッドは CIMValue を返します。呼び出したメソッドが戻り値を定義していない場合には、戻り値は null です。

例 — メソッドの呼び出し

例 4–17 のコードセグメントでは、CIM_Service クラスのインスタンス (デバイスやソフトウェアの機能を管理するサービス) を取得してから、invokeMethod メソッドを使って各サービスを停止します。

例 4–17 メソッドの呼び出し (invokeMethod)

```
{
    ...
    /* CIM_Service クラスの CIMObjectPath を CIM Object Manager に渡す。
    この例では、このクラスに定義されているメソッドを呼び出す */

    CIMObjectPath op = new CIMObjectPath("CIM_Service");
```

```
    /* CIM Object Manager がインスタンスオブジェクトパスの列挙
    (CIM_Service クラスの各インスタンスの名前) を返す */

    Enumeration e = cc.enumInstances(op, cc.DEEP);

    /* インスタンスオブジェクトパスを含む列挙のサイズ分だけ繰り返す。
    各オブジェクトパスによって参照されるインスタンスを取得するには、
    CIMClient の getInstance クラスを使用する。*/

    while(e.hasMoreElements()) {
      // インスタンスを取得する
      CIMInstance ci = cc.getInstance(e.nextElement(), true);
      // StopService メソッドを使って CIM サービスを停止する
      cc.invokeMethod(ci, "StopService", null, null);
  }
}
```

クラス定義の検出

CIM クラスを取得するには `getClass` メソッドを使用します。クラスを作成すると、その派生元クラスとそのクラス階層のすべての親クラスのメソッドとプロパティを継承します。`getClass` メソッドは、ブール値引数 *localOnly* を受け取ります。そして、*localOnly* が真である場合は、継承されたプロパティとメソッド以外のプロパティとメソッドを返し、*localOnly* が偽である場合は、そのクラス内のすべてのプロパティを返します。

例 — クラス定義の検出

例 4–18 のプログラムは、次のメソッドを使用してクラス定義を検出します。

- `CIMNameSpace` – 新しいネームスペースを作成する
- `CIMClient` – CIM Object Manager への新しいクライアント接続を作成する
- `CIMObjectPath` – オブジェクトパス (検出するクラス名を含むオブジェクト) を作成する
- `getClass` – CIM Object Manager からクラスを検出する

例 **4–18**　クラス定義の検出 (getClass)

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;
/**
 * コマンド行に指定されたクラスを取得。デフォルトの
 * ネームスペース root\cimv2 で作業する。
 */
public class GetClass {
    public static void main(String args[]) throws CIMException {
        CIMClient cc = null;
        try {
            CIMNameSpace cns = new CIMNameSpace(args[0]);
            cc = new CIMClient(cns);
            CIMObjectPath cop = new CIMObjectPath(args[1]);
      // 指定されたクラスに対してローカルであるメソッドと
      // プロパティだけを返す (localOnly が true)。
            cc.getClass(cop, cc.DEEP);
        } catch (Exception e) {
            System.out.println("Exception: "+e);
        }
        if(cc != null) {
            cc.close();
        }
    }
}
```

例外の処理

各インタフェースには、CIM の例外を定義する throws 節が含まれます。例外とは、エラー状態を意味します。CIM Object Manager は、Java の例外処理を使用して WBEM 固有の例外の階層を作成します。`CIMException` クラスは、CIM 例外の基底クラスです。`CIMException` 以外の CIM 例外クラスは、`CIMException` クラスのサブクラスです。

CIM 例外の各クラスは、API コードが処理する特定のエラー状態を定義します。CIM 例外 の API については、表 3–2 を参照してください。

Try / Catch 節の使用

クライアント API は、標準 Java の try/catch 節を使用して例外を処理します。一般にアプリケーションは、例外をキャッチした後、何らかの修正アクションを実行するか、あるいはエラー情報をユーザーに渡します。

CIM 仕様では、CIM の規則は明白に定義されていません。CIM の規則は、通常、例によって暗黙に示されます。ほとんどの場合、エラーコードによって示されるのは一般的な問題 (データ型の不一致など) です。プログラマは、その問題に対して具体的な処置を行う必要があります (この場合、正しいデータ型を決定)。

構文上のエラーと意味上のエラー

MOF コンパイラ (mofcomp) は、.mof テキストファイルを Java クラス (bytecode) にコンパイルします。MOF コンパイラは、MOF ファイルの構文検査を行います。CIM Object Manager は、各種のアプリケーションからアクセスできるので、構文と意味の両方の検査を行います。

例 4–19 の MOF ファイルは、2 つのクラス、A と B を定義します。このサンプルファイルをコンパイルすると、CIM Object Manager は意味上のエラーを返します。これは、キーをオーバーライドできるのは別のキーだけであるためです

例 4–19 意味上のエラーの検査

```
Class A          \\クラス A を定義

{      [Key]
int a;
}

Class B:A        \\Class B はクラス A のサブクラス

{   [overrides ("c", key (false) ]
int b;
}
```

高度なプログラミング

この節では、高度なプログラミング作業と、使用頻度の低いプログラミング作業について説明します。

ネームスペースの作成

インストールを行うと、標準の CIM MOF ファイルがデフォルトのネームスペース root\cimv2 と root\security にコンパイルされます。新しいネームスペースを作成する場合は、そのネームスペースにオブジェクトを作成する前に適切な CIM MOF ファイルをネームスペースにコンパイルします。たとえば、標準の CIM 要素を使用するクラスを作成する場合は、ネームスペースに CIM コアスキーマをコンパイルします。CIM アプリケーションスキーマを拡張するクラスを作成する場合は、ネームスペースに CIM アプリケーションをコンパイルします。

例 — ネームスペースの作成

例 4–20 のコードセグメントは、2 段階の処理により既存のネームスペース内にネームスペースを作成します。

- 最初に、CIMNameSpace メソッドを使用してネームスペースオブジェクトを構築します。このネームスペースオブジェクトには、ネームスペースが実際に作成される際に CIM Object Manager に渡されるパラメータが含まれます。
- 次に、CIMClient クラスを使用して CIM Object Manager に接続し、作成したネームスペースオブジェクトを渡します。CIM Object Manager は、このネームスペースオブジェクト内のパラメータを使用してネームスペースを作成します。

例 4–20 ネームスペースの作成 (CIMNameSpace)

```
{
    ...
    /* ネームスペースオブジェクトをクライアント上に作成する。ネームスペースオブジェクトは、
    コマンド行から入力されたパラメータを格納する。args[0] にはホスト名
    (たとえば myhost)、args[1] には親ネームスペース (たとえば
    toplevel ディレクトリ) がそれぞれ含まれている。 */

    CIMNameSpace cns = new CIMNameSpace (args[0], args[1]);

    /* CIM Object Manager に接続し、パラメータとして、ホスト名
```

```
    (args[0]) と親ネームスペース名 (args[1]) を持つネームスペース
    オブジェクト (cns) と、ユーザー名文字列 (args[3])、パスワード文字列
    (args[4]) の 3 つを渡す。 */

    CIMClient cc = new CIMClient (cns, "root", "secret");

    /* CIM Object Manager に別のネームスペースオブジェクトを渡す。
    これには、null 文字列 (ホスト名) と args[2] (
    子ネームスペースの名前 (たとえば、secondlevel) ) が含まれる。 */

    CIMNameSpace cop = new CIMNameSpace("", args[2]);

    /* secondlevel という新しいネームスペースを、myhost 上の
    toplevel ネームスペースの下に作成する。 /*

    cc.createNameSpace(cop);
    ...
}
```

ネームスペースの削除

ネームスペースを削除するには、deleteNameSpace メソッドを使用します。

例 — ネームスペースの削除

例 4–21 のサンプルプログラムは、指定されたホスト上の指定されたネームスペースを削除します。このプログラムは、5 つの必須文字列引数 (ホスト名、親ネームスペース、子ネームスペース、ユーザー名、パスワード) を受け取ります。このプログラムを実行するユーザーは、削除するネームスペースへの書き込みアクセス権を持つアカウントのユーザー名とパスワードを指定する必要があります。

例 4–21 ネームスペースの削除 (deleteNameSpace)

```
{
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;
 /**
 * このサンプルプログラムは、指定されたホスト上の指定されたネームスペースを
 * 削除する。このプログラムを実行するユーザーは、指定するネームスペースへの
 * 書き込みアクセス権を持つアカウントのユーザー名とパスワードを指定する必要
 * がある
 */
 public class DeleteNameSpace {
```

```
    public static void main(String args[]) throws CIMException {
        // CIMClient クラスのインスタンスを初期化する
    CIMClient cc = null;
        // 5 つのコマンド行引数が必要。全部が入力されない場合は、
        // コマンド文字列を出力する
    if(args.length != 5) {
           System.out.println("Usage: DeleteNameSpace host parentNS
               childNS username password");
          System.exit(1);
    }
    try {
           /**
       * ネームスペースオブジェクト (cns) を作成し、
       * ホスト名と親ネームスペースを格納する
       */
           CIMNameSpace cns = new CIMNameSpace(args[0], args[1]);
         /**
       * CIM Object Manager に接続し、ネームスペース
       * オブジェクト (cns) と、コマンド行引数のユーザー名と
          * パスワードを渡す
          */
          cc = new CIMClient(cns, args[3], args[4]);
         /**
       * もう 1 つネームスペースオブジェクト (cop) を作成し、ホスト名
       * として null 文字列を、子ネームスペースとしてコマンド行引数の
       * 文字列をそれぞれ格納する
       */
         CIMNameSpace cop = new CIMNameSpace("",args[2]);
       /**
       * 親ネームスペースの下にある子ネームスペースを削除する
           */
           cc.deleteNameSpace(cop);
    } catch (Exception e) {
           System.out.println("Exception: "+e);
    }
     // セッションを閉じる
    if(cc != null) {
          cc.close();
    }
     }
}
```

基底クラスの作成

アプリケーションは、MOF 言語またはクライアント API のどちらかを使用してクラスを作成できます。MOF の構文に慣れている場合は、テキストエディタを使用して MOF ファイルを作成し、その後 MOF コンパイラを使用してそのファイルを Java クラスにコンパイルすることをお勧めします。この節では、クライアント API を使用して基底クラスを作成する方法を説明します。

CIM クラスを表す Java クラスを作成するには、`CIMClass` クラスを使用します。ほとんどの基底クラスは、クラス名を指定するだけで宣言できます。ほとんどのク

ラスには、クラスのデータを表すプロパティが含まれます。プロパティを宣言するには、プロパティのデータ型、名前、およびオプションのデフォルト値を含めます。プロパティのデータ型は、CIMDataType (事前に定義された CIM のデータ型の 1 つ) のインスタンスにします。

プロパティには、キープロパティであることを識別するキー修飾子を指定できます。キープロパティは、クラスのインスタンスを個別に定義します。インスタンスを持てるのは、キーが指定されたクラスだけです。そのため、キープロパティが定義されないクラスは、abstract クラスとしてしか使用できません。

新しいネームスペース内でクラスにキープロパティを定義する場合は、最初にコア MOF ファイルをそのネームスペースにコンパイルする必要があります。コア MOF ファイルには、標準の CIM 修飾子 (キー修飾子など) の宣言が含まれます。

クラス定義は、別名、修飾子、修飾子フレーバなどの MOF 機能が含まれることにより複雑化します。

例 — CIM クラスの作成

例 4–22 は、ローカルホスト上のデフォルトネームスペース (root\cimv2) に新しい CIM クラスを作成します。このクラスは 2 つのプロパティを持ち、その 1 つはこのクラスのキープロパティです。続いて、newInstance メソッドを使用して、この新しいクラスのインスタンスを作成します。

例 4–22 CIM クラスの作成 (CIMClass)

```
{
...
    /* ローカルホスト上の root\cimv2 ネームスペースに接続し、
    新しいクラス myclass を作成 */

    // ローカルホスト上のデフォルトネームスペースに接続
    CIMClient cc = new CIMClient();

    // 新しい CIMClass オブジェクトを構築
    CIMClass cimclass = new CIMClass();

    // CIM クラス名を myclass に設定
    cimclass.setName("myclass");

    // 新しい CIM プロパティオブジェクトを構築
    CIMProperty cp = new CIMProperty();

    // プロパティ名を設定
    cp.setName("keyprop");

    // プロパティの型として、あらかじめ定義されている CIM データ型のいずれか 1 つを設定
    cp.setType(CIMDatatype.getPredefinedType(CIMDataType.STRING));
```

```
    // 新しい CIM Qualifier オブジェクトを構築
    CIMQualifier cq = new CIMQualifier();

    // 修飾子名を設定
    cq.setName("key");

    // 新しいキー修飾子名をプロパティに追加
    cp.addQualfier(cq);

    /* 整数プロパティを作成し、10 に初期化 */

    // 新しい CIM プロパティオブジェクトを構築
    CIMProperty mp = new CIMProperty();

    // プロパティ名に myprop を設定
    mp.setName("myprop");

    // プロパティの型に、あらかじめ定義されている CIM データ型のいずれかを設定
    mp.setType(CIMDatatype.getPredefinedType(CIMDataType.SINT16));

    // mp を CIMValue に初期化する。CIM Object Manager は、
    // 値 10 を持つ新しい整数オブジェクトであるこの CIMValue を、
    // 上の mp.setType 文でプロパティに対して指定されている
    // CIM データ型 (SINT16) に変換する。CIMValue (Integer 10)
    // がプロパティ (SINT16) の CIM データ型で定義される値の
    // 範囲内にない場合、CIM Object Manager は例外をスローする。
    mp.setValue(new CIMValue(new Integer(10)));

    /* myclass にこの新しいプロパティを追加した後、クラスを
    作成するために CIM Object Manager を呼び出す。 */

    // クラスオブジェクトにキープロパティを追加
    cimclass.addProperty(cp);

    // クラスオブジェクトに整数プロパティを追加
    cimclass.addProperty(mp);

    /* CIM Object Manager に接続し、新しいクラスを渡す。 */
    cc.createClass(new CIMObjectPath(),cimclass);

    // myclass に新しい CIM インスタンスを作成
    ci = cc.newInstance();

    // クライアント接続が開かれている場合、接続を閉じる。
  if(cc != null) {
        cc.close();
    }
}
```

クラスの削除

クラスを削除するには、CIMClient の deleteClass メソッドを使用します。クラスを削除すると、クラス、そのサブクラス、およびそのすべてのインスタンスが削除されます。削除されるクラスを参照する関連は削除されません。

例 — クラスの削除

例 4–23 のサンプルプログラムは、deleteClass メソッドを使って、デフォルトネームスペース root\cimv2 にあるクラスを削除します。このプログラムは、4 つの必須文字列引数 (ホスト名、クラス名、ユーザー名、パスワード) を受け取ります。このプログラムを実行するユーザーは、root\cimv2 ネームスペースへの書き込みアクセス権を持つアカウントのユーザー名とパスワードを指定する必要があります。

例 4–23 クラスの削除 (deleteClass)

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;
/**
* コマンド行に指定されるクラスを削除。
* デフォルトネームスペース root\cimv2 で作業を行う。
*/
public class DeleteClass {
     public static void main(String args[]) throws CIMException {
       CIMClient cc = null;
       if(args.length != 4) {
            System.out.println("Usage:
                      DeleteClass host className username password");
            System.exit(1);
      }
      try {
           /**
             * ネームスペースオブジェクト (cns) を作成し、ホスト名を渡す
            */
        CIMNameSpace cns = new CIMNameSpace(args[0]);

          /**
        * CIM Object Manager に接続し、ネームスペース
        * オブジェクト (cns) と、コマンド行引数から入力された
            * ユーザー名とパスワードを渡す
            */
        cc = new CIMClient(cns, args[2], args[3]);

         /**
         * クラスの名前 (args[1]) を持つオブジェクト (CIMObjectPath)
         * を作成する
         */
        CIMObjectPath cop = new CIMObjectPath(args[1]);

        /**
         * CIM オブジェクトパスで参照されるクラスを削除する
         */
```

```
            cc.deleteClass(cop);
        } catch (Exception e) {
            System.out.println("Exception: "+e);
        }
        if(cc != null) {
            cc.close();
         }
       }
}
```

修飾子のデータ型と修飾子の処理

CIM 修飾子は、CIM クラス、インスタンス、プロパティ、メソッド、またはパラメータの特性を示す要素です。修飾子の属性は、次のとおりです。

- データ型
- 値
- 名前

MOF (Managed Object Format) 構文では、各 CIM 修飾子は同じ MOF ファイルでそのデータ型を宣言する必要があります。修飾子には、スコープ属性はありません。スコープは、どの CIM 要素がその修飾子を使用できるかを示します。スコープを定義できるのは、修飾子のデータ型宣言内だけです。スコープは、修飾子では変更できません。

次に、CIM 修飾子のデータ型を宣言する MOF 構文を示します。この文は、ブール型 (デフォルト値は false) を使用して修飾子のデータ型 `key` を定義します。このデータ型が記述できるのは、プロパティと、オブジェクトに対する参照だけです。`DisableOverride` フレーバは、このキー修飾子がそれらの値を変更できないことを意味します。

```
Qualifier Key : boolean = false, Scope(property, reference),
        Flavor(DisableOverride);
```

次のコード例は、CIM 修飾子の MOF 構文を示します。このサンプル MOF ファイルでは、`key` と `Description` はプロパティ `test` の修飾子です。プロパティのデータ型は、値 `a` を持つ整数です。

```
{
[key, Description("test")]
int a
}
```

例 — CIM 修飾子の取得

例 4–24 のコードセグメントは、CIMQualifier クラスを使用して CIM 要素のベクトル内の CIM 修飾子を識別します。この例は、各 CIM 修飾子ごとにプロパティ名、値、およびデータ型を返します。

修飾子フレーバは、修飾子の使用を制御するフラグです。フレーバは、派生クラスとインスタンスに修飾子を継承できるかどうか、および派生クラスまたはインスタンスが修飾子の本来の値をオーバーライドできるかどうかを指定する規則を記述します。

例 4–24 CIM 修飾子の取得 (CIMQualifier)

```
...
    } else if (tableType == QUALIFIER_TABLE) {
  CIMQualifier prop = (CIMQualifier)cimElements.elementAt(row);
  if (prop != null) {
      if (col == nameColumn) {
   return prop.getName();
      } else if (col == typeColumn) {
   CIMValue cv = prop.getValue();
   if (cv != null) {
       return cv.getType().toString();
   } else {
       return "NULL";          }
    }

...
```

例 — CIM 修飾子の設定

例 4–25 のコードセグメントは、新しいクラスの CIM 修飾子のリストをそのスーパークラス内の修飾子に設定します。

例 4–25 修飾子の設定 (setQualifiers)

```
   ...
 try  {
      cimSuperClass = cimClient.getClass(new CIMObjectPath(scName));
      Vector v = new Vector();
      for (Enumeration e = cimSuperClass.getQualifiers().elements();
     e.hasMoreElements();) {
     CIMQualifier qual = (CIMQualifier)((CIMQualifier)e.nextElement()).clone();
     v.addElement(qual);
      }
      cimClass.setQualifiers(v);
  } catch (CIMException exc) {
```

```
        return;
    }
}
```

プログラム例

例が置かれているディレクトリには、クライアント API を使用して機能を実行するプログラム例が入っています。これらの例を使用すると、独自のアプリケーション開発を簡単に開始できます。これらのプログラム例については、第 7 章で説明しています。

プログラム例を実行するには、次のコマンドを入力します。

`java` *program_name*

たとえば、`java createNameSpace` を入力します。

第 5 章

プロバイダプログラムの作成

この章では、プロバイダの作成方法について説明します。内容は次のとおりです。

プロバイダ API についての詳細は、Javadoc リファレンスページを参照してください。

プロバイダについて

プロバイダは、データにアクセスするために管理対象オブジェクトと通信するクラスです。プロバイダは、統合と解釈を行うために情報を CIM Object Manager に送ります。CIM Object Manager Repository に存在しないデータの要求を管理アプリケーションから受け取る場合、CIM Object Manager はその要求をプロバイダに送ります。

オブジェクトプロバイダは、CIM Object Manager と同じマシンにインストールされている必要があります。CIM Object Manager は、オブジェクトプロバイダ API を使用して、ローカルマシンにインストールされているプロバイダと通信します。

アプリケーションが CIM Object Manager に動的データを要求した場合、CIM Object Manager はプロバイダインタフェースを使用してその要求をプロバイダに渡します。

プロバイダは、CIM Object Manager からの要求に応答して次の機能を実行します。

- デバイスに固有な情報形式を CIM Java クラスに対応付ける
  - デバイスから情報を取得する
  - 取得した情報を、CIM Java クラスの形式で CIM Object Manager に渡す
- CIM Java クラスからの情報を、デバイスに固有なデバイス形式に対応付ける
  - CIM Java クラスから必要な情報を取得する
  - 取得した情報を、デバイスに固有なデバイス形式で渡す

プロバイダの種類

プロバイダは扱うことができるサービスの要求タイプによって分類されます。Sun WBEM SDK では、次の 3 種類のプロバイダをサポートしています。

- インスタンス – 指定されたクラス (Solaris パッケージなど) の動的インスタンスを提供します。インスタンスプロバイダは、次に示す処理の 1 つ以上をサポートします。
  - インスタンス検出
  - 列挙
  - 変更
  - 削除
- プロパティ – 動的なプロパティ値 (ディスク容量など) を提供します。
- メソッド – 1 つ以上のクラスのメソッドを提供します。メソッドは、クラスの動作を表現する関数です。メソッドは、プロバイダによって実装される必要があります。
- 関連付け – 動的関連クラスのインスタンスを提供します。

- イベント — CIM イベントの発生を処理します。イベントプロバイダについては、第 6 章を参照してください。

プロバイダ 1 つでインスタンス、プロパティ、メソッド、関連を提供できるため便利です。

ほとんどのプロバイダは、プルプロバイダです。つまり、必要に応じてデータを動的に生成することによりそれ自体のデータを管理します。プルプロバイダは、CIM Object Manager および CIM Repository と最小限の対話しか行いません。プルプロバイダによって管理されるデータは、通常、頻繁に変化します。そのため、プロバイダはアプリケーションから要求が出るたびにデータを動的に生成するか、あるいはローカルキャッシュからデータを取り出す必要があります。

プロバイダは、単体でインスタンスプロバイダ、プロパティプロバイダ、およびメソッドプロバイダの役割を同時に果たすことができます。このためには、関連するすべてのメソッドを適切に登録、実装する必要があります。

プロバイダインタフェースの実装

プロバイダは、それらの役割に固有のサービスをサポートするプロバイダインタフェースを実装します。インタフェースを実装するには、プロバイダクラスは初めに `implements` 節でそのインタフェースを宣言し、続いてそのインタフェースのすべての抽象メソッドに対して実装 (本体) を提供する必要があります。

CIM Object Manager は、`initialize` メソッドを使用してプロバイダと通信できます。`initialize` メソッドには、CIM Object Manager への参照である、`CIMOMhandle` 型の引数を指定できます。`CIMOMhandle` クラスには、プロバイダが CIM Object Manager との間でデータ転送に使用できるメソッドが含まれます。

次の表で、`com.sun.wbem.provider` パッケージに含まれているプロバイダインタフェースについて説明します。`com.sun.wbem.provider20` パッケージの `InstanceProvider` インタフェースを使用してください。 メソッドプロバイダ、インスタンスプロバイダ、およびプロパティプロバイダは、単一の Java クラスファイルに含めることも、あるいは個々のファイルに格納することもできます。

表 5–1　プロバイダインタフェース

インタフェース	説明
CIMProvider	すべてのプロバイダが実装する基本インタフェース
InstanceProvider	インスタンスプロバイダが実装する基本インタフェース。インスタンスプロバイダは、クラスの動的インスタンスを提供する
MethodProvider	メソッドプロバイダが実装する基本インタフェース。メソッドプロバイダは、CIM クラスのすべてのメソッドを実装する
PropertyProvider	プロパティプロバイダが実装する基本インタフェース。プロパティプロバイダは、動的プロパティの検出と更新のために使用される。動的データは、CIM Object Manager Repository には格納されない
AssociatorProvider	動的関連インスタンスのプロバイダによって実装される基本インタフェース
Authorizable	マーカーインタフェースは、プロバイダが独自に承認検査を行うことを CIM Object Manger に知らせる

インスタンスプロバイダインタフェース (InstanceProvider)

次の表で、Provider パッケージ (com.sun.wbem.provider20) に含まれるインスタンスプロバイダインタフェースのメソッドについて説明します。

これらの各メソッドは引数 *op* を取ります。*op* は、指定された CIM クラスまたは CIM インスタンスの CIMObjectPath です。オブジェクトパスには、ネームスペース、クラス名、およびキー (オブジェクトがインスタンスの場合) が含まれます。ネームスペースは、ほかのネームスペース、クラス、インスタンス、および修飾子のデータ型を格納できるディレクトリです。キーは、クラスインスタンスを個別に識別するプロパティです。キープロパティには、キー修飾子が含まれます

たとえば、次のオブジェクトパスは 2 つの部分から構成されます。

```
\\myserver\root\cimv2\Solaris_ComputerSystem:Name=mycomputer:
   CreationClassName=Solaris_ComputerSystem
```

- `\\myserver\root\cimv2`

  これは、ホスト `myserver` 上のデフォルトの CIM ネームスペースです。

- `Solaris_ComputerSystem:Name=mycomputer:`
  `CreationClassName=Solaris_ComputerSystem`

  これは、ホスト `myserver` 上のデフォルトネームスペースに存在する特定の Solaris Computer System オブジェクトです。この Solaris コンピュータシステムは、(プロパティ=値) という書式で示される 2 つのキー修飾子値によって個別に識別されます。

  - `Name=mycomputer`
  - `CreationClassName=Solaris_ComputerSystem`

表 5–2 `InstanceProvider` インタフェースメソッド

メソッド	説明
CIMObjectPath `createInstance(CIMObjectPath op, CIMInstance ci)`	*op* によって指定されたインスタンス *ci* が存在しない場合は、それを作成する。この CIM インスタンスがすでにある場合は、ID が `CIM_ERR_ALREADY_EXISTS` の `CIMInstanceException` をプロバイダがスローする必要がある。作成されたインスタンスの `CIMObjectPath` を返す。
void `deleteInstance(CIMObjectPath op)`	オブジェクトパス (*op*) に指定されたインスタンスを削除する。

表 5–2 InstanceProvider インタフェースメソッド 続く

メソッド	説明
Vector enumInstances(CIMObjectPath path, boolean deep, CIMClass cc)	*path* に指定されたクラスの各インスタンスの「名前」を返す。*deep* が *true* の場合は、指定されたクラスとそのクラスから派生したすべてのクラスのすべてのインスタンスの名前を返す。*deep* が *true* でない場合は、指定されたクラスに属するインスタンスの名前だけを返す。 プロバイダがインスタンスをゼロから作成したくない場合は、新しいインスタンスのテンプレートを作成できる。それには、そのインスタンスが属するクラス (*cc*) の newInstance() メソッドを呼び出す。
Vector enumInstances(CIMObjectPath path, boolean deep, CIMClass cc, boolean localOnly)	*path* に指定されたクラスのインスタンス (インスタンスの名前だけでなく、インスタンス全体) を返す。 プロバイダがインスタンスをゼロから作成したくない場合は、新しいインスタンスのテンプレートを作成できる。それには、そのインスタンスが属するクラス (*cc*) の newInstance() メソッドを呼び出す。 *localOnly* が true の場合は、列挙されたインスタンスのローカルプロパティ (継承されたプロパティではない) だけを返す。*localOnly* が true でない場合は、すべての継承されたプロパティとローカルプロパティを返す。
Vector execQuery (CIMObjectPath op, String query, int ql, CIMClass cc	照会を実行して CIM オブジェクトを検索する。このメソッドは、指定された照会文字列に一致する、指定された CIM クラス (*cc*) の CIM インスタンスのベクトルを返す。
CIMInstance getInstance(CIMObjectPath op, CIMClass cc, boolean localOnly)	オブジェクトパス (*op*) に指定されたインスタンスを返す。 プロバイダがインスタンスをゼロから作成したくない場合は、新しいインスタンスのテンプレートを作成できる。それには、そのインスタンスが属するクラス (*cc*) の newInstance() メソッドを呼び出す。 *localOnly* が true の場合は、列挙されたインスタンスのローカルプロパティ (継承されたプロパティではない) だけを返す。*localOnly* が true でない場合は、すべての継承されたプロパティとローカルプロパティを返す。
void setInstance(CIMInstance ci)	指定された CIM インスタンスが存在する場合は、それを更新する。このインスタンスが存在しない場合は、ID が CIM_ERR_NOT_FOUND の CIMInstanceException をスローする。

例 — インスタンスプロバイダの実装

次のコード例は、インスタンスプロバイダ SimpleInstanceProvider の Java ソースコードです。このインスタンスプロバイダは、Ex_SimpleInstanceProvider クラスの enumInstances インタフェースと getInstance インタフェースを実装しています。わかりやすくするために、この例では、CIMException をスローすることによって deleteInstance、createInstance、setInstance、execQuery の各インタフェースを実装します。実際には、インスタンスプロバイダがすべての InstanceProvider インスタンスを実装する必要があります。

例 5–1 SimpleInstanceProvider インスタンスプロバイダ

```
/*
 * "@(#)SimpleInstanceProvider.java"
 */
import com.sun.wbem.cim.*;
import com.sun.wbem.client.*;
import com.sun.wbem.provider.CIMProvider;
import com.sun.wbem.provider20.InstanceProvider;
import com.sun.wbem.provider.MethodProvider;
import java.util.*;
import java.io.*;

public class SimpleInstanceProvider implements InstanceProvider{
     static int loop = 0;
     public void initialize(CIMOMHandle cimom) throws CIMException {
     }
     public void cleanup() throws CIMException {
     }
     public Vector enumInstances(CIMObjectPath op, boolean deep, CIMClass cc,
            boolean localOnly) throws CIMException {
                return null;
     }
     /*
      * enumInstances:
      * 名前だけでなく、インスタンス全体が返される。
      * 詳細列挙または簡易列挙が可能だが、現在のところ
      * CIMOM は簡易列挙だけを要求する。
      */
     public Vector enumInstances(CIMObjectPath op, boolean deep, CIMClass cc)
             throws CIMException {
         if (op.getObjectName().equalsIgnoreCase("Ex_SimpleInstanceProvider"))
             {
             Vector instances = new Vector();
             CIMObjectPath cop = new CIMObjectPath(op.getObjectName(),
                     op.getNameSpace());
                 if (loop == 0){
                     cop.addKey("First", new CIMValue("red"));
                     cop.addKey("Last", new CIMValue("apple"));
                     // このクラスを削除する場合は、この次の行を
                     // コメントにしてコンパイルする。
                     instances.addElement(cop);
                     loop += 1;
                 } else {
                     cop.addKey("First", new CIMValue("red"));
```

```
                    cop.addKey("Last", new CIMValue("apple"));
                    // このクラスを削除する場合は、この次の行を
                    // コメントにしてコンパイルする。
                    instances.addElement(cop);
                    cop = new CIMObjectPath(op.getObjectName(),
                            op.getNameSpace());
                    cop.addKey("First", new CIMValue("green"));
                    cop.addKey("Last", new CIMValue("apple"));
                    // このクラスを削除する場合は、この次の行を
                    // コメントにしてコンパイルする。
                    instances.addElement(cop);
                }
            return instances;
            }
        return new Vector();
    }

    public CIMInstance getInstance(CIMObjectPath op,
            CIMClass cc, boolean localOnly) throws CIMException {
            if (op.getObjectName().equalsIgnoreCase("Ex_SimpleInstanceProvider"))
            {
                CIMInstance ci = cc.newInstance();
                ci.setProperty("First", new CIMValue("yellow"));
                ci.setProperty("Last", new CIMValue("apple"));
                return ci;
            }
           return new CIMInstance();
     }

    public Vector execQuery(CIMObjectPath op, String query, int ql, CIMClass cc)
            throws CIMException {
                throw(new CIMException(CIMException.CIM_ERR_NOT_SUPPORTED));
   }

    public void setInstance(CIMObjectPath op, CIMInstance ci)
            throws CIMException {
                throw(new CIMException(CIMException.CIM_ERR_NOT_SUPPORTED));
   }

    public CIMObjectPath createInstance(CIMObjectPath op, CIMInstance ci)
             throws CIMException {
                throw(new CIMException(CIMException.CIM_ERR_NOT_SUPPORTED));
   }

    public void deleteInstance(CIMObjectPath cp) throws CIMException {
        throw(new CIMException(CIMException.CIM_ERR_NOT_SUPPORTED));
    }
}
```

プロパティプロバイダインタフェース (PropertyProvider)

次の表で、プロパティプロバイダインタフェースのメソッドについて説明します。

表 **5–3** PropertyProvider インタフェースメソッド

メソッド	説明
CIMValue `getPropertyValue(CIMObjectPath op, String originClass, String propertyName)`	*op* に指定されたインスタンスの *propertyName* によって指定されたプロパティの値を含む `CIMValue` を返す。*originClass* には、このプロパティを最初に定義した、このクラス階層のクラスの名前を指定する。
void `setPropertyValue(CIMObjectPath op, String originClass, String propertyName, CIMValue cv)`	*op* に指定されたインスタンスの *propertyName* によって指定されたプロパティの値に `CIMValue` *cv* を設定する。*originClass* には、このプロパティを最初に定義した、このクラス階層のクラスの名前を指定する。

例 — プロパティプロバイダの実装

例 5–2 のコードセグメントは、例 5–2 で登録されるプロパティプロバイダ (`fruit_prop_provider`) クラスを作成します。`fruit_prop_provider` は、`PropertyProvider` インタフェースを実装します。

このサンプルプロパティプロバイダは、指定されたクラスのプロパティ値、親クラス、およびプロパティ名を返す `getPropertyValue` メソッドを示しています。CIM プロパティは、その名前と親クラスによって定義されます。複数のプロパティで同じ名前を使用できますが、親クラスはプロパティを個別に識別します。

例 **5–2** プロパティプロバイダの実装

```
...

public class SimplePropertyProvider implements PropertyProvider{
    public void initialize(CIMOMHandle cimom)
    throws CIMException {
    }

    public void cleanup()
    throws CIMException {
    }

    public CIMValue getPropertyValue(CIMObjectpath op, string originclass,
            string PropertyName){
      if (PropertyName.equals("A")
          return new CIMValue("ValueA")
      else
         return new CIMValue("ValueB");
    }
    ...
```

}

メソッドプロバイダインタフェース (MethodProvider)

次の表で、メソッドプロバイダインタフェースのメソッドについて説明します。

表 5–4 MethodProvider インタフェースメソッド

メソッド	説明
CIMValue invokeMethod(CIMObjectPath op, String methodName, Vector inParams, Vector outParams)	CIM Object Manager は、op によって参照されるインスタンス内の methodName が呼び出されると、このメソッドを呼び出す。 *inParams* は、呼び出されるメソッドへの入力パラメータとなる CIMValues のベクトル、*outParams* は、呼び出されるメソッドからの出力パラメータとなる CIMValues のベクトル。

例 — メソッドプロバイダの実装

例 5–3 のコードセグメントは、CIM Object Manager からのメソッド実行要求を 1 つ以上の特化されたプロバイダに送る Solaris プロバイダクラスを作成します。これらの特化されたプロバイダは、特定の Solaris オブジェクトの動的データの要求に対するサービスを行います。たとえば、Solaris_Package プロバイダは、Solaris_Package クラスのメソッドを実行する要求に対応します。

この例のメソッドプロバイダは、適切なプロバイダを呼び出して次に示す処理の 1 つを実行する単一のメソッド invokeMethod を実装します。

- Solaris システムをリブートする
- Solaris システムをリブートまたは停止する
- Solaris シリアルポートを削除する

例 5–3 メソッドプロバイダの実装

```
...
public class Solaris implements MethodProvider {
```

```
    public void initialize(CIMONHandle, ch) throws CIMException {
    }
    public void cleanup() throws CIMException {
    }
    public CIMValue invokeMethod(CIMObjectPath op, String methodName,
             Vector inParams, Vector outParams) throws CIMException {
        if (op.getObjectName().equalsIgnoreCase("solaris_computersystem")) {
            Solaris_ComputerSystem sp = new Solaris_ComputerSystem();
            if (methodName.equalsIgnoreCase("reboot")) {
                return new CIMValue (sp.Reboot());
            }
        }
        if (op.getObjectName().equalsIgnoreCase("solaris_operatingsystem")) {
            Solaris_OperatingSystem sos = new Solaris_OperatingSystem();
            if (methodName.equalsIgnoreCase("reboot")) {
                return new CIMValue (sos.Reboot());
            }
            if (methodName.equalsIgnoreCase("shutdown")) {
                return new CIMValue (sos.Shutdown());
            }
        }
        if (op.getObjectName().equalsIgnoreCase("solaris_serialport")) {
            Solaris_SerialPort ser = new Solaris_SerialPort();
            if (methodName.equalsIgnoreCase("disableportservice")) {
                return new CIMValue (ser.DeletePort(op));
            }
        }
        return null;
    }
}
...
```

アソシエータプロバイダインタフェース (AssociatorProvider)

次の表で、AssociatorProvider インタフェースのメソッドについて説明します。これらのメソッドに渡す引数についての詳細は、107ページの「関連メソッド」を参照してください。

表 5–5 AssociatorProvider インタフェースのメソッド

メソッド	説明
Vector associators(CIMObjectPath assocName, CIMObjectPath objectName, String role, String resultRole, boolean includeQualifiers, boolean includeClassOrigin, String[] propertyList)	objectName によって指定されたインスタンスに関連付けられている CIM インスタンスのベクトルを返す。
Vector associatorNames(CIMObjectPath assocName, CIMObjectPath objectName, String role, String resultRole)	objectName によって指定された CIM インスタンスに関連付けられている CIM インスタンスの名前のベクトルを返す。
Vector references(CIMObjectPath assocName, CIMObjectPath objectName, String role, boolean includeQualifiers, boolean includeClassOrigin, String[] propertyList)	objectName によって指定された CIM インスタンスが関与する関連ベクトルを返す。
Vector referenceNames(CIMObjectPath assocName, CIMObjectPath objectName, String role)	objectName によって指定された CIM インスタンスが関与する関連の名前のベクトルを返す。

例 — アソシエータプロバイダの実装

完全な関連付けプロバイダには、すべての AssociatorProvider メソッドを実装する必要があります。わかりやすくするために、次のコード例には、associators メソッドだけを実装しています。CIM Object Manager は、assocName、objectName、 role、 resultRole、 includeQualifiers、includeClassOrigin、 propertyList のそれぞれの値を関連付けプロバイダに渡します。

例 5–4 では、CIM 関連付けクラスの名前と、返される関連付けされたオブジェクトが属する CIM クラスまたはインスタンスを出力します。このプロバイダは、example_teacher クラスと example_student クラスのインスタンスを扱います。

例 5–4 アソシエータプロバイダの実装

```
public Vector associators(CIMObjectPath assocName,
        CIMObjectPath objectName, String role,
     String resultRole, boolean includeQualifiers,
     boolean includeClassOrigin, String propertyList[]) throws CIMException {
    System.out.println("Associators "+assocName+" "+objectName);
    if (objectName.getObjectName()equalsIgnoreCase("example_teacher")) {
```

```
        Vector v = new Vector();
        if ((role != null)  &&
            (!role.equalsIgnoreCase("teaches"))) {
             // teacher は teaches という役割だけを担う
             return v;
        }
    // teacher のアソシエータを取得する
    CIMProperty nameProp = (CIMProperty)objectName.getKeys().elementAt(0);
    String name = (String)nameProp.getValue().getValue();
    // student クラスを取得する
    CIMObjectPath tempOp = new CIMObjectPath("example_student");
    tempOp.setNameSpace(assocName.getNameSpace());
    CIMClass cc = cimom.getClass(tempOp, false);
     // objectName によって渡されたインスタンス名をテストし、
     // student クラスの関連付けされたインスタンスを返す
     if(name.equals("teacher1")) {
        // teacher1 の student (複数) を取得する
        CIMInstance ci = cc.newInstance();
        ci.setProperty("name", new CIMValue("student1"));
        v.addElement(ci.filterProperties(propertyList,
               includeQualifiers, includeClassOrigin));
        ci = cc.newInstance();
        ci.setProperty("name", new CIMValue("student2"));
        v.addElement(ci.filterProperties(propertyList,
               includeQualifiers, includeClassOrigin));
        return v;
     }
}
```

ネイティブプロバイダの作成

プロバイダは、管理対象デバイスに関する情報の取得と設定を行います。ネイティブプロバイダは、管理対象デバイスで動作するように作成されるマシン固有のプログラムです。たとえば、Solaris システム上のデータにアクセスするプロバイダは、通常、C 関数を組み込んで Solaris システムを照会します。ネイティブプロバイダは、一般に次のような理由で作成されます。

- 効率 – 速度が重視されるコードの一部を低レベルのプログラミング言語 (アセンブラなど) で実装し、その後 Java アプリケーションでそれらの機能を呼び出すと便利な場合がある
- プラットフォーム固有の機能にアクセスする必要がある – 標準の Java クラスライブラリが、アプリケーションに必要なプラットフォームに固有の機能をサポートしていない場合がある
- 従来のコード - Java 以外のプログラミング言語で作成された従来のコードを Java プロバイダと共に継続して使用したい場合がある

JDK の一部である JNI (Java Native Interface) は、Java のネイティブプログラミングインタフェースです。JNI を使用してプログラムを作成すると、ほとんどのプ

ラットフォームで完全に移植可能です。Java Virtual Machine (VM) で動作する Java コードで JNI を使用すると、そのコードは C、C++、アセンブラのようなほかの言語で作成されたアプリケーションおよびライブラリで実行できます。

Java プログラムの作成、および Java プログラムとネイティブメソッドの統合についての詳細は、 Java Web サイト `http://www.javasoft.com/docs/books/tutorial/native1.1/index.html` を参照してください。

プロバイダのインストール

プロバイダを作成したら、プロバイダクラスファイルと共有ライブラリファイルの場所を指定し、その後 CIM Object Manager の停止と再起動を行う必要があります。

▼ プロバイダをインストールする方法

1. **次に示す方法の 1 つを使用して、共有ライブラリファイルの場所を指定します。**
   - LD_LIBRARY_PATH 環境変数を、共有ライブラリファイルのある場所に設定します。C シェルを使用している場合の入力例を次に示します。

     注 - ある 1 つのシェルで LD_LIBRARY_PATH 環境変数を設定する場合は、設定した新しい値を認識させるために、そのシェルで CIM Object Manager の停止と再起動を行なってください。

     ```
     % setenv LD_LIBRARY_PATH /wbem/provider/
     ```

     Borne シェルを使用している場合の入力例を次に示します。

     ```
     % LD_LIBRARY_PATH = /wbem/provider/
     ```

- 共有ライブラリファイルを、LD_LIBRARY_PATH 環境変数によって指定されているディレクトリのうちの 1 つにコピーします。WBEM のインストールを行うと、この環境変数は /usr/sadm/lib/wbem と /usr/snadm/lib に設定されます。次に入力例を示します。

```
% cp libnative.so /usr/sadm/lib/wbem
% cp native.c /usr/sadm/lib/wbem
```

2. プロバイダクラスファイルを、/usr/sadm/lib/wbem に移動します。

   プロバイダクラスファイルをそれらが定義されているパッケージと同じパスに移動します。たとえば、プロバイダが com.sun.providers.myprovider.* としてパッケージされている場合は、プロバイダクラスファイルを /usr/sadm/lib/wbem/com/sun/providers/myprovider/ に移動します。

3. **Solaris** プロバイダの **CLASSPATH** 変数に、プロバイダクラスファイルがあるディレクトリを設定します。これについては、142ページの「プロバイダの CLASSPATH を設定する方法」を参照してください。

4. 次のコマンドを入力して **CIM Object Manager** を停止します。

注 - ある 1 つのシェルで LD_LIBRARY_PATH 環境変数を設定する場合は、設定した新しい値を認識させるために、そのシェルで CIM Object Manager の停止と再起動を行なってください。

```
# /etc/init.d/init.wbem -stop
```

5. 次のコマンドを入力して **CIM Object Manager** を再起動します。

```
# /etc/init.d/init.wbem -start
```

Solaris プロバイダの CLASSPATH の設定

Solaris プロバイダの CLASSPATH を設定するには、クライアント API を使って Solaris_ProviderPath クラスのインスタンスを作成し、その pathurl プロパティにプロバイダクラスファイルの場所を設定します。Solaris_ProviderPath クラスは \root\system ネームスペースに格納されています。

プロバイダの CLASSPATH には、プロバイダクラスファイルの場所を設定することもできます。さらに、クラスパスには、jar ファイルや、クラスを含む任意のディレクトリを設定できます。CLASSPATH の設定には、Java が使用する標準の URL を使用します。

プロバイダの **CLASSPATH**	構文
ディレクトリへの絶対パス	`file:///a/b/c/`
CIM Object Manager が起動された、ディレクトリへの相対パス (/)..	`file://a/b/c`

▼ プロバイダの CLASSPATH を設定する方法

1. Solaris_ProviderPath クラスのインスタンスを作成します。以下に例を示します。

```
/* root\system (ネームスペースの名前) で初期化されたネームスペース
オブジェクトをローカルホスト上に作成する  */
CIMNameSpace cns = new CIMNameSpace("", "root\system");

// root\system ネームスペースに root として接続する
cc = new CIMClient(cns, "root", "root_password");

// Solaris_ProviderPath クラスを取得する
cimclass = cc.getClass(new CIMObjectPath("Solaris_ProviderPath");

// Solaris_ProviderPath の新しいインスタンスを作成する
class ci = cimclass.newInstance();
```

2. pathurl プロパティにプロバイダクラスファイルの場所を設定します。以下に例を示します。

```
...
/*  プロバイダの CLASSPATH に //com/mycomp/myproviders/ を設定する */
ci.setProperty("pathurl", new CIMValue(new String("//com/mycomp/myproviders/"));
...
```

3. インスタンスを更新します。以下に例を示します。

```
// 更新されたインスタンスを CIM Object Manager に渡す
cc.setInstance(new CIMObjectPath(), ci);
```

プロバイダの登録

サポートするデータと処理、およびそれらの物理的な実装についての情報を公開するため、CIM Object Manager にプロバイダを登録します。CIM Object Manager は、この情報を使用してプロバイダのロードと初期化、および特定のクライアント要求に適切なプロバイダを決定します。すべてのタイプのプロバイダについて、同じ方法で登録します。

▼ プロバイダを登録する方法

1. **CIM** クラスを定義する **MOF** ファイルを作成します。

2. そのクラスに、***provider*** 修飾子を割り当てます。***provider*** 修飾子には、プロバイダ名を割り当てます。

   プロバイダ名により、そのクラスのプロバイダとしての役目をする Java クラスが識別されます。クラス名は、完全に指定する必要があります。次に例を示します。

```
[Provider("com.kailee.wbem.providers.provider_name")]
Class_name {
…
};
```

注 - プロバイダ名が固有になるように、Java クラスおよび Java パッケージの命名規則に従うことを推奨します。固有のパッケージ名の接頭辞は小文字の ASCII 文字で、トップレベルのドメイン名にする必要があります。現在は、`com`、`edu`、`gov`、`mil`、`net`、`org`、または、ISO 標準 3166、1981 で指定されている国名を識別する 2 文字コードなどです。

パッケージ名の後に続く名前は、組織内部の命名規則によって異なります。たとえば、あるディレクトリ名のコンポーネントは、部名、課名、プロジェクト名、マシン名、あるいはログイン名などになります (例 : `com.mycompany.wbem.myprovider`)。

---

3. **MOF** ファイルをコンパイルします。次に例を示します。

```
% mofcomp class_name
```

MOF コンパイラを使って MOF ファイルをコンパイルする方法については、『*Solaris WBEM Services* の管理』を参照してください。

MOF ファイルの変更

コンパイル済みの MOF ファイルに入っているクラス定義を変更する場合は、MOF ファイルをコンパイルし直す前に、そのクラスを CIM Object Manager Repository から削除する必要があります。削除しないと、クラスがすでに存在するためエラーになり、新しい情報が CIM Object Manager に伝達されません。45ページの「クラスとクラス属性の削除」で説明しているように、CIM WorkShop を使用してクラスを削除することができます。

例 — プロバイダの登録

次の例は、`SimpleInstanceProvider` によって提供される `Ex_SimpleInstanceProvider` クラスを CIM Object Manager に宣言する MOF

ファイルを示したものです (例 5–1) 。MOF ファイルのプロバイダ名とクラス名は、次の規則に従っていなければなりません。

- クラス名は有効な CIM スキーマ名でなければなりません。つまり、クラス名には、接頭辞とその後に続く下線が必要です。たとえば、green_apples や red_apples は有効な CIM スキーマ名ですが、apples は有効な CIM スキーマ名ではありません。
- クラス名は、MOF ファイルのプロバイダに指定されているクラス名と同じでなければなりません。例 5–5 の MOF ファイルには、以下のように Ex_SimpleInstanceProvider クラスが定義されています。

  ```
  class Ex_SimpleInstanceProvider
  ```

  例 5–1 のプロバイダ中の enumInstances メソッドには、以下のようにこれと同じクラス名が指定されています。

  ```
  public Vector enumInstances(CIMObjectPath op, boolean deep, CIMClass cc)
      throws CIMException {
          if (op.getObjectName().equalsIgnoreCase("Ex_SimpleInstanceProvider"))
  ```

- MOF ファイルに指定するプロバイダ名は、プロバイダクラスファイルの名前と同じでなければなりません。例 5–5 の MOF ファイルには、以下のように Ex_SimpleInstanceProvider クラスのプロバイダとして SimpleInstanceProvider が指定されています。

  ```
  [Provider("SimpleInstanceProvider")]
  class Ex_SimpleInstanceProvider
  ```

例 5–5 SimpleInstanceProvider MOF ファイル

```
// ==================================================================
// Title:       SimpleInstanceProvider
// Filename:    SimpleInstanceProvider.mof
// Description:
// ==================================================================

// ==================================================================
// Pragmas
// ==================================================================
#pragma Locale ("en-US")

// ==================================================================
//   SimpleInstanceProvider
// ==================================================================
[Provider("SimpleInstanceProvider")]
```

```
class Ex_SimpleInstanceProvider
{
   // プロパティ
      [Key, Description("First Name of the User")]
   string First;
      [Description("Last Name of the User")]
   string Last;
};
```

プロバイダの変更

プロバイダクラスは、CIM Object Manager とプロバイダの動作中に変更できます。しかし、加えた変更を有効にするためには、CIM Object Manager を停止して再起動する必要があります。

▼ プロバイダを変更する方法

1. プロバイダソースファイルを編集します。
2. プロバイダソースファイルをコンパイルします。次に例を示します。

```
% javac MyProvider.java
```

3. システムプロンプトで次のコマンドを入力し、システム上でスーパーユーザーになります。

```
% su
```

4. プロンプトに対し、スーパーユーザーのパスワードを入力します。

5. 次のコマンドを入力して、init.wbem コマンドがある位置にディレクトリを変更します。

```
# cd /etc/init.d/
```

6. 次のコマンドを入力して、**CIM Object Manager** を停止します。

```
# init.wbem -stop
```

7. 次のコマンドを入力して、**CIM Object Manager** を再起動します。

```
# init.wbem -start
```

WQL 照会の処理

WBEM クライアントは、CIMClient クラスの execQuery メソッドを使って、一連の検索条件に一致するインスタンスを検索します。CIM Object Manager は、CIM Object Manager Repository に格納されている CIM データのクライアント照会を処理すると共に、特定のプロバイダから提供される CIM データの照会をプロバイダに渡します。

すべてのインスタンスプロバイダは、com.sun.wbem.provider20 パッケージにある execQuery インタフェースを実装して、インスタンスプロバイダ自身が提供する動的データのクライアント照会を処理する必要があります。プロバイダは、com.sun.wbem.query パッケージにあるクラスやメソッドを使って、WQL (WBEM Query Language) の照会文字列をフィルタリングすることができます。イ

ンデックスを処理するエンティティへのアクセス権を持つプロバイダは、照会文字列をこのエンティティに渡して解析してもらうことができます。

照会 API による照会文字列の解析

com.sun.wbem.query パッケージ中のクラスやメソッドは、WQL パーサーと、解析する WQL 文字列を表しています。このパッケージには、照会文字列内の節を表すクラスと、これらの節の中の文字列を処理するメソッドが含まれています。

現在のところ解析可能な WQL 式のタイプは SELECT 式だけです。SELECT 式には次の部分が含まれています。

- SELECT 文
- FROM 節
- WHERE 節

WQL の式

次の図は、WQL 式中の節と WBEM クラスの対応を表したものです。

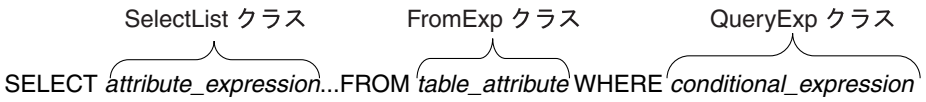

図 5–1　WQL 式を表す WBEM クラス

WQL	**WBEM** 照会クラス
SELECT *attribute_expression*	SelectList
FROM *table_attribute*	FromExp
WHERE *conditional_expression*	QueryExp

WQL は、CIM データモデルに基づいて格納されるデータを照会できるようになっています。CIM モデルでは、オブジェクトに関する情報は CIM クラスや CIM インスタンスに格納されます。CIM インスタンスには、名前、データ型、値からなるプロパティを持含めることができます。WQL は、次の表に示すように CIM オブジェクトモデルを SQL テーブルにマップします。

SQL	WQL
テーブル	CIM クラス
行	CIM インスタンス
列	CIM プロパティ

CIM の WQL 式は、次の形式で表すことができます。

SELECT *CIM property* ...**FROM** *CIM class* **WHERE** *propertyA = 40*

次はより実際的な WQL 式の例です。

```
SELECT * FROM Solaris_FileSystem WHERE
(Name="home" OR Name="files") AND AvailableSpace > 2000000
```

SELECT 文

`SelectExp` クラスは、SELECT 分を表します。

SELECT 文は、情報検索に使用される SQL 文ですが、WQL SELECT 文には、WQL 特有の制約と拡張がいくつかあります。SQL SELECT 文は、通常データベース環境でテーブルから特定の列を検索するのに使用されます。WQL SELECT 文は、単一クラスのインスタンスを検索するのに使用されます。WQL では、複数のクラスに渡る照会はサポートされません。

検索リストは SELECT 式で表されます。SELECT 文には次の形式があります。

SELECT 文	選択されるもの
`SELECT *`	指定されたクラスとそのすべてのサブクラスのすべてのインスタンス。
`SELECT attr_exp, attr_exp...attr_exp`	指定されたクラスとそのすべてのサブクラスのすべてのインスタンスのうち、指定された識別子をもっているものだけ。

FROM 節

FROM 節は、abstract クラス `fromExp` によって表されます。現在のところ、`fromExp` の直接のサブクラスは `NonJoinExp` だけです。`NonJoinExp` は、1 つのテーブル (CIM クラス) だけを指定した FROM 節を表しています。select 操作はこのテーブルに対して行われます。

FROM 節では、照会文字列に一致するインスタンスが含まれているクラスを指定します。SQL では、FROM 節に修飾子付き属性式を指定します。これが検索するクラスの名前になります。修飾子付き属性式には、そのテーブルとクラスを指定します。現在のところ WQL FROM 節では、非 join 式だけがサポートされます。したがって、WQL FROM 節には 1 つのクラスしか指定できません。

WHERE 節

`QueryExp` クラスは、その各サブクラスが条件式を表している abstract クラスです。特定の `CIMInstance` がこれらの条件式に適用されると、ブール値を返します。

WHERE 節は、照会のスコープを狭めるためのものです。WHERE 節には条件式が含まれています。これらの条件式には、プロパティまたはキーワード、演算子、定数が含まれています。WHERE 節には、あらかじめ定義された WQL 演算子のいずれかを指定する必要があります。

SELECT 文の後に追加する WHERE 節の基本的な構文は、次のとおりです。

SELECT *CIM instance* **FROM** *CIM class***WHERE** *conditional_expression*

WHERE 節の条件式は、次の形式です。

property operator constant

`QueryExp` クラスの次の各サブクラスは、WHERE 節における条件式の特定のタイプに対応しています。

- `AndQueryExp`
- `BinaryRelQueryExp`
- `NotQueryExp`
- `OrQueryExp`

WHERE 節の条件式は `QueryExp` クラスで表されます。条件式はツリー構造になっています。たとえば、条件式 (a=2 and b=3 or c=4) は、次のようなツリー構造で表されます。

`QueryExp` クラスは照会式ツリーのトップレベルだけを返します (上の例では ORQueryExp)。これによって、プロバイダは、そのクラス内のメソッドを使って照会式ツリーの分岐を下方にたどることができます。

canonize メソッドの使用

プロバイダが WQL 照会文字列を別のエンティティに渡し、この文字列を解析させるには、次のメソッドが便利です。

- `canonizeDOC` - 式を「論理積の論理和」形式にします (比較式と比較式の AND を OR にする)。これによって、式をツリー形式ではなく、複数のリストが含まれる 1 つのリストとして処理できるため、評価が容易になります。例: (x > 5 and y > 6) or (y > 6 and z=7)
- `canonizeCOD` - 式を「論理和の論理積」形式にします (比較式と比較式の OR を AND にする)。これによって、式をツリー形式ではなく複数のリストが含まれる 1 つのリストとして処理できるため、評価が容易になります。例: (x > 5 or y > 6) and (y > 6 or z=7)

WQL 照会文字列を解析するプロバイダの作成

照会 API を使って WQL 照会文字列を解析するプロバイダを作成する一般的な手順は、次のとおりです。

▼ WQL 照会文字列を解析するプロバイダを作成する方法

1. **WQL** パーサーを初期化します。以下に例を示します。

```
/* CIM Object Manager から execQuery に渡された
   照会文字列を入力データストリームに読み込む */
ByteArrayInputStream in = new ByteArrayInputStream(query.getBytes());

/* 入力データストリームを使ってパーサーを初期化する */
WQLParser parser = new WQLParser(in);
```

2. 照会結果を格納するベクトルを作成します。以下に例を示します。

```
Vector result = new Vector();
```

3. 照会から **select** 式を取得します。以下に例を示します。

```
/* querySpecification がパーサーから WQL 式を返す
  (SelectExp)parser が WQL 式を select 式にキャストする */
SelectExp q = (SelectExp)parser.querySpecification();
```

4. **select** 式から **select** リストを取得します。以下に例を示します。

```
/* SelectExp クラスの SelectList メソッドを使って
   select リストを返す。select リストとは、
   属性または CIM プロパティのリスト */
SelectList attrs = q.getSelectList();
```

5. **From** 節を取得します。以下に例を示します。

```
/* SelectExp クラスの getFromClause メソッドを使って
   From 節を返す。From 節を非 join 式
   (単一の CIM クラスを表すテーブル) にキャストする */
NonJoinExp from = (NonJoinExp)q.getFromClause();
```

6. enumInstances メソッドを使ってクラスの詳細列挙を返します。以下に例を示します。

```
/* 指定されたクラス (cc) に属するすべてのインスタンス
   (継承されたプロパティとローカルプロパティを含む) を返す */
 Vector v = new Vector();
 v = enumInstances(op, true, cc, true);
 ...
```

7. 列挙のインスタンスを **1** つずつ処理し、その中で各インスタンスを照会式および **select** リストと比較します。以下に例を示します。

```
/* WHERE 節の照会式が CIM インスタンスと
  一致するかを比較する。select リストを CIM インスタンスに
   適用し、select リストと一致するインスタンス
   (CIM プロパティのリスト) を結果に追加する。 */
for (int i = 0; i < v.size(); i++) {
 if ((where == null) || // WHERE 節があるか
     (where.apply((CIMInstance)v.elementAt(i)) == true)) {
        result.addElement(attrs.apply((CIMInstance)v.elementAt(i)));
 ...
```

8. 照会結果を返します。以下に例を示します。

```
return result;
```

例 — execQuery メソッドの実装

例 5–6 のサンプルプログラムでは、execQuery メソッドによって渡された WQL 文字列を、照会 API を使って解析します。このプログラムは、照会文字列中の Select 式を解析し、クラスを詳細列挙します。そして、列挙の各インスタンスを 1 つずつ処理して、各インスタンスを照会式および select リストと比較します。最後にプログラムは、照会文字列と一致するインスタンスの列挙が含まれているベクトルを返します。

例 5–6 execQuery メソッドを実装するプロバイダ

```
/*
     * execQuery メソッドは、部分的なキーマッチングに基づく限られた照会
     * しかサポートしない。照会によって選択されたエントリがないと、
     * 空の Vector を返す
     *
     * @param op    返される CIM インスタンスの CIM オブジェクトパス
     * @param query CIM 照会式
     * @param ql    CIM 照会言語のインジケータ
     * @param cc    CIM クラス参照
     *
     * @return CIM オブジェクトインスタンスのリスト
```

```
     *
     * @version    1.19 01/26/00
     * @author    Sun Microsystems, Inc.
     */
    public Vector execQuery(CIMObjectPath op,
       String query,
       int ql,
       CIMClass cc)
     throws CIMException {

 ByteArrayInputStream in = new ByteArrayInputStream(query.getBytes());
 WQLParser parser = new WQLParser(in);
 Vector result = new Vector();
 try {
     SelectExp q = (SelectExp)parser.querySpecification();
     SelectList attrs = q.getSelectList();
     NonJoinExp from = (NonJoinExp)q.getFromClause();
     QueryExp where = q.getWhereClause();

     Vector v = new Vector();
     v = enumInstances(op, false, cc, true);

     // インスタンスをフィルタリングする
     for (int i = 0; i < v.size(); i++) {
      if ((where == null) || (where.apply((CIMInstance)v.elementAt(i)) == true)) {
          result.addElement(attrs.apply((CIMInstance)v.elementAt(i)));
       }
     }
 } catch (Exception e) {
     throw new CIMException(CIMException.CIM_ERR_FAILED, e.toString());
 }
 return result;
    } // execQuery
}
```

第 6 章

CIM イベントの処理

この章では、最初に CIM イベントモデルについて説明し、次にプロバイダで CIM イベントを生成する方法と、CIM イベントの発生をアプリケーションに通知するように予約する方法について説明します。

- CIM イベントモデル
- 予約 (subscription) の作成
- イベントインジケーションの生成

CIM イベントモデル

イベントとは、1 つの事象 (発生したもの) を表す言葉です。プログラミング用語では、イベントはコンピュータシステム内での 1 つの事象を意味します。そして、通常アプリケーションは、これに応答する必要があります。たとえば、GUI 上のマウスクリックに応じてダイアログボックスを開く場合です。CIM イベントは、管理環境において目的の事象に変化があったことを意味します。

CIM イベントモデル (CIM イベントを処理するフレームワーク) は、DMTF (Desktop Management Task Force) によって公表された CIM (Common Information Model) Indications Specification に準拠しています。CIM イベントモデルでは、イベントとそのイベントの通知 (インジケーション) は区別されます。CIM では、発行されるのはインジケーションであり、イベントではありません。

CIM イベントは、内部イベントと外部イベントに分類されます。内部イベントとは、ネームスペース、クラス、クラスインスタンスの、作成、変更、削除といったデータの変更に対応して発生する組み込み型の CIM イベントです。外部イベントとは、内部イベントによって記述されていないユーザー定義型のイベントです。

現在は、CIM インスタンスの作成、変更、削除に対応する内部イベントだけが処理されます。内部イベントを報告するクラスには、次のものがあります。

- `CIM_InstCreation` — 新しいインスタンスが作成されたことを表します。
- `CIM_InstDeletion` — 既存のインスタンスが削除されたことを表します。
- `CIM_InstModification` — インスタンスが変更されたことを表します。

イベントインジケーションの生成方法

デフォルトでは、CIM Object Manager は、内部イベントのインジケーションを一定間隔でポーリングします。イベントポーリングの間隔や CIM Object Manager のデフォルトのポーリング動作を変更するには、`cimom.properties` ファイルのプロパティを編集します。`cimom.properties` ファイルの編集方法については、『*Solaris WBEM Services* の管理』を参照してください。

CIM Object Manager は、指定されたポーリング間隔でインスタンスを列挙し、列挙された 2 つのインスタンスセットの間にインスタンスの追加や変更、削除があったかどうかを判定します。プロバイダがイベントのインジケーションを生成する場合には、CIM Object Manager はこのプロバイダをポーリングしません。プロバイダは、可能な場合は、サポートするクラスの動的インスタンスで発生する内部イベントのインジケーションを生成する必要があります。これによって、CIM Object Manager がイベントをポーリングすることによるパフォーマンスの低下を避けることができます。

CIM Object Manager Repository は、内部インスタンスが作成、変更、または削除されると、インジケーションを生成します。

予約の作成方法

クライアントアプリケーションでは、CIM イベント発生が通知されるように予約することができます。予約は、1 つまたは複数の目的のインジケーションを宣言する

ことによって行います。現在は、プロバイダがイベントインジケーションを予約することはできません。

CIM イベントのインジケーションの通知予約するアプリケーションには、次の指定が必要です。

- 目的のイベント
- イベントが発生した時に、CIM Object Manager が行うアクション

イベントの発生は、CIM_Indication クラスのいずれかのサブクラスのインスタンスとして表されます。インジケーションは、そのイベントがクライアントによって予約されているときだけ生成されます。クライアントが予約しようとするイベントをどのプロバイダも生成しない場合には、その予約は失敗します。

予約の作成

予約を作成するには、CIMListener インタフェースを追加し、次のクラスのインスタンスを作成します。

- CIM_IndicationFilter — インジケーションを生成する基準と、インジケーションとしてどのようなデータを返すかを定義します。
- CIM_IndicationHandler — インジケーションをどのように処理し、送信する (扱う) かを定義します。ここでは、インジケーションの送信に使用する宛先とプロトコルを定義することがあります。
- CIM_IndicationSubscription — 特定のイベントフィルタと特定のイベントハンドラをバインドする関連付け。

アプリケーションは、1 つまたは複数のイベントフィルタと 1 つまたは複数のイベントハンドラを作成できます。イベントインジケーションは、アプリケーションがイベントの予約を作成するまで送信されません。

CIM リスナーの追加

クライアントアプリケーションは、CIMListener インタフェースを追加することによって、CIM イベントインジケーションの予約を登録する必要があります。CIM Object Manager は、クライアント予約が作成されたときにイベントフィルタに指定された CIM イベントのインジケーションを生成します。

CIMListener インタフェースには、indicationOccured メソッドを実装する必要があります。このメソッドは、引数として CIMEvent (CIMListener から返される CIM イベント) を受け取ります。

例 6–1 CIM リスナーの追加

```
// CIM Object Manager に接続する
cc = new CIMClient();

// CIM リスナーを登録する
cc.addCIMListener(
new CIMListener() {
    public void indicationOccured(CIMEvent e) {
    }
});
```

イベントフィルタの作成

イベントフィルタでは、送信するイベントのタイプと、どのような条件の下で送信するかを指定しますす。アプリケーションでは、CIM_IndicationFilter クラスのインスタンスを作成し、そのプロパティの値を定義することによって、イベントフィルタを作成します。イベントフィルタは特定のネームスペースに属します。個々のイベントフィルタは、そのフィルタが属するネームスペースに属するイベントに対してのみ有効です。

CIM_IndicationFilter クラスには、フィルタを特定して照会文字列を指定するためにアプリケーションが設定できる文字列プロパティ、およびこの文字列を解析するための照会言語があります。現在は、WQL (WBEM Query Language) だけがサポートされます。

表 6–1 CIM_IndicationFilter クラスのプロパティ

プロパティ	説明	必須/任意
SystemCreationClassName	このフィルタを作成するクラスがあるシステムの名前、またはこのクラスが適用されるシステムの名前。	任意。このキープロパティのデフォルトは CIM_System.CreationClassName。
SystemName	このフィルタがあるシステムの名前、またはこのフィルタが適用されるシステムの名前。	任意。このキープロパティのデフォルトは、CIM Object Manager が動作しているシステムの名前。

表 6–1　CIM_IndicationFilter クラスのプロパティ　続く

CreationClassName	このフィルタの作成に使用するクラスまたはサブクラスの名前。	任意。CIM Object Manager は、このキープロパティのデフォルトとして CIM_IndicationFilter を割り当てる。
Name	このフィルタの固有名。	必須。クライアントアプリケーションは固有名を指定する必要がある。
SourceNamespace	CIM インジケーション生成元であるローカルネームスペースへのパス。	任意。デフォルトは \root\cimv2。
Query	インジケーションをどのような条件のときに生成するかを定義する照会式。現在は、Level 1 の WQL 式だけがサポートされる。WQL 照会式の作成方法については、100ページの「照会」を参照。	必須。
QueryLanguage	照会を表現する言語。	必須。デフォルト WQL (WBEM Query Language)。

▼ イベントフィルタを作成する方法

1. CIM_IndicationFilter クラスのインスタンスを作成します。以下に例を示します。

```
CIMClass cimfilter = cc.getClass
        (new CIMObjectPath(''CIM_IndicationFilter''),
         true, true, true, null);CIMInstance ci = cimfilter.newInstance();
```

2. イベントフィルタの名前を指定します。以下に例を示します。

```
Name = ''filter_all_new_solarisdiskdrives''
```

3. **WQL** 文字列を作成し、返されるイベントインジケーションを指定します。以下に例を示します。

```
String filterString = ''SELECT *
        FROM CIM_InstCreation WHERE sourceInstance
        is ISA Solaris_DiskDrive'';
```

4. cimfilter インスタンスの各プロパティ値に、フィルタの名前、**CIM** イベントを選択するフィルタ文字列、照会文字列を解析する照会言語を設定します。

   現在、照会文字列の解析に使用できるのは WQL だけです。以下に例を示します。

```
ci.setProperty(''Name'', new
        CIMValue("filter_all_new_solarisdiskdrives''));
ci.setProperty("Query", new CIMValue(filterString));
ci.setProperty("QueryLanguage", new CIMValue("WQL");)
```

5. cimfilter インスタンスから filter というインスタンスを作成し、それを **CIM Object Manager Repository** に格納します。以下に例を示します。

```
CIMObjectPath filter = cc.createInstance(new CIMObjectPath(), ci);
```

イベントハンドラの作成

イベントハンドラは、CIM_IndicationHandler クラスのインスタンスです。CIM イベント MOF には、HTTP プロトコルを使ってクライアントアプリケーションに送信されるインジケーションの宛先を指定する CIM_IndicationHandlerXMLHTTP クラスが定義されています。イベントの HTTP 送信はまだ定義されていないため、HTTP クライアントへのイベントの送信はサポートされていません。

Solaris イベント MOF は、Solaris_RMIDelivery クラスを作成して CIM_IndicationHandler クラスを拡張し、RMI プロトコルを使用して CIM イベントインジケーションのクライアントアプリケーションへの送信を処理します。RMI クライアントは、Solaris_RMIDelivery クラスのインスタンスを作成して、RMI 送信の場所を設定する必要があります。

アプリケーションでは、CIM_IndicationHandler クラスのプロパティにハンドラの固有名とその所有者の UID を指定します。

表 6–2 CIM_IndicationHandler クラスのプロパティ

プロパティ	説明	必須/任意
SystemCreationClassName	このハンドラを作成するクラスがあるシステムの名前、またはこのクラスが適用されるシステムの名前。	任意。このキープロパティのデフォルトは CIM_System クラスを作成するクラス
SystemName	このハンドラがあるシステムの名前、またはこのハンドラが適用されるシステムの名前。	任意。このキープロパティのデフォルト値は、CIM Object Manager が動作しているシステムの名前。
CreationClassName	このハンドラの作成に使用するクラスまたはサブクラス。	任意。CIM Object Manager は、このキープロパティのデフォルトとして CIM_IndicationFilter を割り当てる。
Name	このハンドラの固有名。	必須。クライアントアプリケーションは固有名を指定する必要がある。
Owner	このハンドラを作成した、または保持するエンティティの名前。プロバイダは、この値を検査して、インジケーションの受信をハンドラに承認するかどうかを確認できる。	任意。デフォルトは、このインスタンスを作成するユーザーの Solaris ユーザー名。

次のコード例は、CIM イベントハンドラの作成方法を示したものです。

例 6–2 CIM イベントハンドラの作成

```
// Solaris_RMIDelivery クラスのインスタンスを作成する
CIMClass rmidelivery = cc.getClass(new CIMObjectPath
        (''Solaris_RMIDelivery''), false, true, true, null);

CIMInstance ci = rmidelivery.newInstance();
```

```
// rmidelivery インスタンスから新しい
// インスタンス (delivery) を作成する
CIMObjectPath delivery = cc.createInstance(new CIMObjectPath(), ci);
```

イベントフィルタとイベントハンドラのバインド

アプリケーションは、CIM_IndicationSubscription クラスのインスタンスを作成して、イベントフィルタとイベントハンドラをバインドします。CIM_IndicationSubscription を作成すると、イベントフィルタによって指定されたイベントのインジケーションが送信されます。

次の例では、予約 (filterdelivery) を作成し、次に 159ページの「イベントフィルタを作成する方法」で作成した filter オブジェクトに filter プロパティを定義し、例 6–2 で作成した delivery オブジェクトに handler プロパティを設定します。

例 6–3 イベントフィルタとイベントハンドラのバインド

```
CIMClass filterdelivery = cc.getClass(new
        CIMObjectPath(''CIM_IndicationSubscription''),
        true, true, true, null);
ci = filterdelivery.newInstance():

// filter インスタンスを参照する filter というプロパティを作成する
ci.setProperty("filter", new CIMValue(filter));

// delivery インスタンスを参照する handler というプロパティを作成する
ci.setProperty("handler", new CIMValue(delivery));
```

イベントインジケーションの生成

CIM イベントのインジケーションを生成するには、次の処理が行われます。

- EventProvider インタフェースのメソッドを使って、CIM イベントインジケーションの送信をいつ開始および停止するかを検出する。
- CIM_Indication クラスの 1 つまたは複数のサブクラスのインスタンスを作成して、発生した CIM イベントの情報を格納する。

- ProviderCIMOMHandle インタフェースの deliverEvent メソッドを使って、インジケーションを CIM Object Manager に送信する。

EventProvider インタフェースのメソッド

イベントプロバイダは、EventProvider インタフェースを実装する必要があります。CIM Object Manager は、このインタフェースのメソッドを使って、クライアントが CIM イベントのインジケーションを予約したり、CIM イベントの予約を取り消したことをプロバイダに知らせます。さらにこれらのメソッドによって、プロバイダは、CIM Object Manager が特定のイベントインジケーションのポーリングを行うべきかどうか、インジケーションをハンドラに返すことを承認するかどうかを指定します。

次の表に、イベントプロバイダが実行しなければならない EventProvider インタフェースのメソッドを示します。

表 6–3 EventProvider インタフェースのメソッド

メソッド	説明
activateFilter	クライアントが予約を登録すると、CIM Object Manager は、このメソッドを呼び出して CIM イベントの検査をプロバイダに依頼する。
authorizeFilter	クライアントが予約を登録すると、CIM Object Manager は、このメソッドを呼び出して指定されたフィルタ式が許可されているかを検査する。
deActivateFilter	クライアントが予約を削除すると、CIM Object Manager は、このメソッドを呼び出して指定されたイベントフィルタの停止をプロバイダに依頼する。
mustPoll	クライアントが予約を削除すると、CIM Object Manager は、このメソッドを呼び出して、指定されたフィルタ式をプロバイダが許可するかどうか、そのフィルタ式のポーリングが必要かどうかを検査する。

CIM Object Manager は、すべてのメソッドに次の引数の値を渡します。

- filter — インジケーションを生成する必要がある CIM イベントを指定する `SelectExp` 型。
- eventType — CIM イベントの型を指定する `String` 型。これは、select 式の FROM 節から抽出することもできます。
- classPath — このイベントを必要とするクラスの名前を指定する `CIMObjectPath` 型。

さらに、`activateFilter` メソッドは、これがこのイベント型の最初のフィルタであることを示すブール値 `firstActivation` を受け取ります。`deActivateFilter` メソッドは、これが最後のフィルタであることを示すブール値 `lastActivation` を受け取ります。

インジケーションの作成と送信

クライアントアプリケーションが `CIM_IndicationSubscription` クラスのインスタンスを作成して CIM イベントのインジケーションを予約すると、CIM Object Manager はこの要求を適切なプロバイダに転送します。プロバイダが `EventProvider` インタフェースを実装していれば、CIM Object Manager は、プロバイダの `activateFilter` メソッド/`deActivateFilter` メソッドを呼び出して、指定するイベントのインジケーションの送信をいつ開始/停止するかをプロバイダに通知します。

プロバイダは、インスタンスを作成、変更、削除するたびに、インジケーションを作成、送信して、CIM Object Manager の要求に応答します。通常プロバイダは、CIM Object Manager が `activateFilter` メソッド/`deActivateFilter` メソッドを呼び出した時に設定/クリアされるフラグ変数を定義します。そして、インスタンスを作成、変更、または削除するメソッドの中で、アクティブなフラグのステータスを検査します。フラグが設定されている場合、プロバイダは、作成した CIM インスタンスオブジェクトを含むインジケーションを作成し、`deliverEvent` メソッドを使ってこのインジケーションを CIM Object Manager に返します。フラグが設定されていない場合、プロバイダは、イベントインジケーションの作成や送信を行いません。

承認

機密データを扱うプロバイダは、インジケーションの要求に対する承認を検査することができます。その場合、プロバイダは `Authorizable` インタフェースを実装して、そのプロバイダが承認を検査することを示す必要があります。さらに、プロバイダは `authorizeFilter` メソッドを実装する必要があります。CIM Object Manager は、このメソッドを呼び出して、イベントハンドラの所有者 (UID) がフィルタ式の評価に基づいて送信されるインジケーションを受信することを承認されているかどうかを検査します。イベントの宛先の所有者 (イベントハンドラ) の UID は、フィルタをアクティブにすることを要求するクライアントアプリケーションの所有者と同じである必要はありません。

CIM インジケーションクラス

プロバイダは、`CIM_Indication` クラスのサブクラスのインスタンスを作成することによって、CIM イベントのインジケーションを生成します。

次の表に、プロバイダが生成すべき内部 CIM イベントを示します。

表 6–4 CIM イベントインジケーションクラス

イベントクラス	説明
`CIM_InstCreation`	新しいインスタンスが作成されたことを知らせる。
`CIM_InstDeletion`	既存のインスタンスが削除されたことを知らせる。
`CIM_InstModification`	インスタンスが変更されたことを知らせる。インジケーションには、変更される前のインスタンスのコピーが含まれていなければならない。

▼ イベントインジケーションを生成する方法

1. `EventProvider` インタフェースを実装します。以下に例を示します。

```
public class sampleEventProvider implements
        InstanceProvider EventProvider{

    // プロバイダが CIM Object Manager に接続するための参照
    private ProviderCIMOMHandle cimom;
   }
```

2. プロバイダが処理するインスタンスインジケーションに対して、表 6–3 に示すそれぞれのメソッドを実行します。

3. 作成、変更、削除のインスタンスのイベント型ごとに、表 6–4 に示すインジケーションを作成します。以下に createInstance メソッドの例を示します。

```
public CIMObjectPath createInstance(CIMObjectPath op,
        CIMInstance ci)
    throws CIMException {
        CIMObjectpath newop = ip.createInstance(op, ci);
        CIMInstance indication = new CIMInstance();
        indication.setClassName("CIM_InstCreation");
        CIMProperty cp = new CIMProperty();
        cp.setName("SourceInstance");
        cp.setValue(new CIMValue(ci));
        Vector v = new Vector();
        v.addElement(cp);
        indication.setProperties(v);
        ...
    }
```

4. イベントインジケーションを **CIM Object Manager** に送信します。以下に例を示します。

```
cimom.deliverEvent(op.getNameSpace(), indication);
        return newop;
```

第 7 章

Sun WBEM SDK サンプルの使用

この章では、Sun WBEM SDK に付属のプログラム例について説明します。内容は次のとおりです。

- プログラム例について
- アプレットの使用
- クライアント例の使用
- プロバイダ例の使用

プログラム例について

Sun WBEM SDK には、サンプル Java プログラムが付属しています。このプログラムは、`/usr/demo/wbem` にインストールされます。このソースコードは、独自のプログラムを開発するためのベースとして使用できます。提供されているプログラム例は次の 2 種類です。

- アプレット – Java 対応の Web ブラウザや Java Development Kit (JDK) のアプレットビューアで動作するプログラム
- クライアントプログラム – クライアント API と CIM API を使用して CIM Object Manager からの WBEM オペレーションを要求するプログラム
- プロバイダプログラム – データにアクセスするために管理対象オブジェクトと通信を行うプログラム

アプレットの使用

Solaris WBEM Services が動作するシステムにインストールされている Solaris ソフトウェアパッケージをリストするには、`GetPackageInfoAp` という Java アプレットを使用します。このアプレットを使用して、パッケージを選択し、そのパッケージの詳しい情報を表示できます。さらに、このアプレットでは、ローカルシステムまたはリモートシステムで動作する CIM Object Manager に接続できます。

このアプレットを実行するには、次のいずれかが必要です。

- Java Development Kit (JDK) 1.2 アップレットビューア
- Java 対応の Web ブラウザ。ただし、JRE 1.2.2 が使用されているか、Java Plug-in 1.2.2 ソフトウェアが有効になっている必要があります。

アプレットの実行に関する詳細は、`/usr/demo/wbem/README` を参照してください。

クライアント例の使用

クライアント例は、クライアント API を使用して、クラス、インスタンス、ネームスペースの、作成、削除、表示を行います。次の 4 種類のクライアントプログラムが提供されています。

- 列挙 – クラスとインスタンスを列挙します。このプログラムは、コマンド行から渡されるクラスの詳細 (deep) 列挙と簡易 (shallow) 列挙を行います。
- ロギング – ログ記録の書き込みと読み取りを行います。
- その他 – クラスとインスタンスを削除します。
- ネームスペース – ネームスペースの作成と削除を行います。
- システム情報 – 選択されるシステムおよびネットワークの Solaris プロセス情報を表示します。

クライアントサンプルファイル

次の表において、クライアントプログラム例のファイルについて説明し、各例を実行するコマンドと引数を示します。

表 7–1 クライアントファイル例

サンプルファイル名	説明	実行するコマンド
`CreateNameSpace`	指定されたユーザーとして CIM Object Manager に接続し、指定されたホスト上にネームスペースを作成する。root ユーザー名とパスワードを指定する必要がある。	`java CreateNameSpace` *host parentNS childNS username password*
`DeleteNameSpace`	指定されたホスト上の指定されたネームスペースを削除する。root ユーザー名とパスワードを指定する必要がある。	`java DeleteNameSpace` *host parentNS childNS username password*
`ClientEnum`	指定されたホスト上のデフォルトネームスペース `root\cimv2` に存在する指定されたクラスのサブクラスとインスタンスを列挙する。	`java ClientEnum` *host className*
`CreateLog`	指定されたホスト上にログレコードを作成する。root ユーザー名とパスワードを指定する必要がある。	`java CreateLog` *host username password*
`ReadLog`	指定されたホスト上のログレコードを読み取る。root ユーザー名とパスワードを指定する必要がある。	`java ReadLog` *host username password*

表 7–1 クライアントファイル例 続く

サンプルファイル名	説明	実行するコマンド
`DeleteClass`	指定されたホスト上のデフォルトネームスペース `root\cimv2` に存在する指定されたクラスを削除する。root ユーザー名とパスワードを指定する必要がある。	`java DeleteClass` *host className username password*
`DeleteInstances`	指定されたホスト上のデフォルトネームスペース `root\cimv2` に存在する指定されたクラスのインスタンスを削除する。root ユーザー名とパスワードを指定する必要がある。	`java DeleteInstances` *host className username password*
`CreateQualifierType`	指定されたホスト上の指定されたネームスペースに、指定された修飾子型を作成する。root のユーザー名とパスワードを指定する必要がある。	`java CreateQualifierType` *host parentNS username password qualifierTypeName*
`SystemInfo`	指定されたホストの Solaris プロセッサとシステムの情報をウィンドウに表示する。	`java SystemInfo` *host*

クライアント例の実行

クライアントプログラム例を実行するには、次のコマンドを入力します。

```
% java program_name
```

プログラム例のほとんどは、デフォルト値を持つ必須引数を取ります。たとえば、CreateNameSpace プログラム例には、次の 5 つの引数を指定します。

- ホスト名
- 親ネームスペース
- 子ネームスペース
- ユーザー名
- パスワード

コマンド行引数にデフォルト値を指定するには、次の構文を使用します。

引数	デフォルト値	構文
Host name	local host	.
Parent namespace	root\cimv2	" "
Child namespace	Null	" "
User name	GUEST	" "
Password	GUEST	" "

次に、ローカルホスト上のデフォルトネームスペース root\cimv2 に、パスワード secret を持つユーザー admin として接続する CreateNameSpace を実行する例を示します。

```
% java CreateNameSpace . "" root secret
```

プロバイダ例の使用

プロバイダ例は、システムプロパティを返し、文字列「Hello World」を出力する Java プログラムです。プロバイダは、ネイティブ C メソッドを呼び出してコードを実行し、値をプロバイダに返します。

プロバイダファイルの例

次の表で、プロバイダプログラムの各ファイルについて説明します。

表 7–2 プロバイダファイルの例

ファイル	目的
NativeProvider	CIM Object Manager からの要求に応答し、それらを Native_Example プロバイダに送るトップレベルのプロバイダプログラム。NativeProvider プログラムは、instanceProvider API と method Provider API を実装し、Native_Example クラスのインスタンスを列挙するメソッドと、取得するメソッドを宣言する。このプログラムは、文字列「Hello World」を出力するメソッドを呼び出すメソッドも宣言する。
Native_Example.mof	NativeProvider プロバイダを CIM Object Manager に登録するクラスを作成する。Native_Example.mof ファイルは、NativeProvider を、Native_Example クラスの動的データ要求に対してサービスを行うプロバイダとして識別する。この MOF ファイルは、NativeProvider によって実装されるプロパティとメソッドの宣言も行う。
Native_Example.java	NativeProvider プログラムは、このプロバイダを呼び出して、Native_Example クラスのインスタンスを列挙するメソッドと取得するメソッドを実装する。Native_Example プロバイダは、API を使用してオブジェクトの列挙とオブジェクトインスタンスの作成を行う。Native_Example クラスは、native.c ファイル内の C 関数を呼び出してシステム固有の値 (ホスト名、シリアル番号、リリース、マシン、アーキテクチャ、メーカーなど) を取得するネイティブメソッドの宣言も行う。
native.c	Native_Example Java プロバイダからの呼び出しをネイティブ C コードで実装する C プログラム。

表 7–2　プロバイダファイルの例　続く

ファイル	目的
Native_Example.h	Native_Example クラスに対して自動的に生成されるヘッダーファイル。Java ネイティブメソッド名とそれらのメソッドを実行するネイティブ C 関数間の対話を定義する。
libnative.so	native.c ファイルからコンパイルされるバイナリネイティブ C コード。

ネイティブプロバイダの作成

Java プログラムの作成、および Java プログラムとネイティブメソッドの統合についての詳細は、Java の Web サイト http://www.javasoft.com/docs/books/tutorial/native1.1/TOC.html を参照してください。

プロバイダ例の設定

プロバイダプログラム例 NativeProvider は、Native_Example クラスのインスタンスの列挙と、インスタンスプロパティの取得を行います。Native_Example クラスとそのインスタンスは、CIM WorkShop を使用して表示できます。

▼ プロバイダ例を設定する方法

1. 次に示す方法の 1 つを使用して、共有ライブラリファイルの場所を指定します。
   - LD_LIBRARY_PATH 環境変数を、共有ライブラリファイルの場所に設定します。次に C シェルを使用する入力例を示します。

     注 - ある 1 つのシェルで LD_LIBRARY_PATH 環境変数を設定する場合は、この環境変数の新しい値が認識されるように、そのシェルで CIM Object Manager を停止および再起動してください。

```
% setenv LD_LIBRARY_PATH /wbem/provider/
```

たとえば、Bourne シェルを使用する場合には、次のように指定します。

```
% LD_LIBRARY_PATH = /wbem/provider/
```

- 共有ライブラリファイルを、LD_LIBRARY_PATH 環境変数によって指定されているディレクトリにコピーします。WBEM のインストールを行うと、この環境変数は /usr/sadm/lib/wbem に設定されます。次に入力例を示します。

```
% cp libnative.so /usr/sadm/lib/wbem
% cp native.c /usr/sadm/lib/wbem
% cp Native_Example.h /usr/sadm/lib/wbem
```

2. プロバイダクラスファイルを /usr/sadm/lib/wbem に移動します。

   プロバイダクラスファイルは、それらのファイルが定義されているパッケージと同じパスに移動する必要があります。たとえば、プロバイダが com.sun.providers.myprovider.* のようにパッケージされている場合には、プロバイダクラスファイルを /usr/sadm/lib/wbem/com/sun/wbem/myprovider/.class に移動します。

3. **Solaris** プロバイダの **CLASSPATH** 変数に、プロバイダクラスファイルが含まれているディレクトリを設定します。これについては、142ページの「プロバイダの CLASSPATH を設定する方法」を参照してください。

4. 次のコマンドを入力して **CIM Object Manager** を停止します。

```
# /etc/init.d/init.wbem -stop
```

5. 次のコマンドを入力して **CIM Object Manager** を再起動します。

```
# /etc/init.d/init.wbem -start
```

6. Native_Example.mof ファイルをコンパイルします。次に入力例を示します。

```
% mofcomp Native_Example.mof
```

この MOF ファイルをコンパイルすると、CIM Object Manager に Native_Example クラスが読み込まれ、NativeProvider がそのプロバイダとして識別されます。

7. **CIM WorkShop** を実行し、Native_Example クラスを表示します。次に入力例を示します。

```
% /usr/sadm/bin/cimworkshop &
```

8. ツールバーで「クラスを検索 **(Find Class)**」アイコンをクリックします。

9. 「入力 **(Input)**」ダイアログボックスで、Native_Example と入力して「了解 **(OK)**」をクリックします。

第 8 章

エラーメッセージ

この章では、Solaris WBEM Services と Sun WBEM SDK のコンポーネントが生成するエラーメッセージについて説明します。取り上げる内容は次のとおりです。

- エラーメッセージの生成
- エラーメッセージの構成
- エラーメッセージ情報の検索
- 生成されるエラーメッセージ

エラーメッセージの生成

CIM Object Manager が、MOF コンパイラと CIM WorkShop で使用されるエラーメッセージを最初に生成します。MOF コンパイラにより、`.mof` ファイルのどこでそのエラーが発生したかを示す行がその後に追加されます。

エラーメッセージの構成

エラーメッセージは、次の要素から構成されます。

- 固有の識別子 – そのエラーメッセージをほかのエラーメッセージと区別するための文字列
- 例外メッセージ – エラーメッセージの説明

- パラメータ – 例外メッセージに示される特定のクラス、メソッド、および修飾子の可変部分

エラーメッセージの例

MOF コンパイラは、たとえば次のようなエラーメッセージを返します。

```
REF_REQUIRED = Association class CIM_Docked needs
 at least two refs. Error in line 12.
```

- REF_REQUIRED は固有の識別子
- Association class CIM_Docked needs at least two refs は、例外メッセージ
- CIM_Docked はパラメータです。パラメータは、該当するクラス、プロパティ、メソッド、または修飾子の名前に置換される

開発者向け: エラーメッセージテンプレート

WBEM は発生し得るすべてのエラーメッセージを例外テンプレート (API の ErrorMessages_en.properties ファイル) として提供しています。パラメータが必要な例外テンプレートでは、最初のパラメータは {0}、2 つめのパラメータは {1} として示されています。

前述の例では、次の例外テンプレートが使用されています。

```
REF_REQUIRED = Association class {0} needs at least two refs.
```

エラーメッセージ情報の検索

Javadoc リファレンスページでエラーメッセージの固有の識別子を検索し、エラーメッセージの説明を表示できます。

次の節では、各エラーメッセージについて詳しく説明しています。これらのエラーメッセージは、固有の識別子で編成されています。各メッセージごとに、次の中から該当する情報を示します。

- 固有の識別子: ヘッダーとして表示される
- 説明: エラーメッセージ内に使用されているパラメータの説明
- 例: ユーザーに表示されるエラーメッセージ例。この例は、エラーメッセージがパラメータを使用する場合に、それらのパラメータが要素 (クラス名など) に置き換わったときにエラーメッセージがどのように表示されるかを示している
- 原因: そのエラーメッセージが生成された理由と、メッセージの理解に役立つ背景 (参照) 情報を示す
- 解決方法: そのエラーを解決する方法がある場合、その手順などが示される

生成されるエラーメッセージ

この節では、MOF コンパイラ、CIM Object Manager、および CIM WorkShop が生成するエラーメッセージについて説明します。

ABSTRACT_INSTANCE

説明

ABSTRACT_INSTANCE エラーメッセージは パラメータ {0} を使用しますが、このパラメータは abstract クラスの名前に置換されています。

例

ABSTRACT_INSTANCE = Abstract class ExampleClass cannot have instances.

原因

指定されたクラスにインスタンスが設定されましたが、このクラスは abstract クラスです。abstract クラスは、インスタンスを持てません。

解決方法

設定されたインスタンスを削除します。

CHECKSUM_ERROR

説明

CHECKSUM_ERROR エラーメッセージは、パラメータを使用しません。

例

CHECKSUM_ERROR = Checksum not valid.

原因

メッセージは、壊れているため送信できませんでした。この損傷は、送信中に偶然に生じたか、あるいは第三者によって故意に壊された可能性があります。

注 - このエラーメッセージは、CIM Object Manager が無効なチェックサムを受ける場合に表示されます。チェックサムは、ネットワーク上で転送されるデータパケットのビット数です。この数は、伝送が安全であり、かつ送信中にデータの破損や意図的な変更がなかったことを情報の送信側と受信側が確認するために使用されます。

送信前に、データに対してアルゴリズムが実行されます。この実行により生成されたチェックサムがデータに含まれ、データパケットのサイズを示します。メッセージを受信すると、受信者はチェックサムを再計算し、送信者のチェックサムと比較できます。チェックサムが一致すれば、送信は安全に行われ、データの破損や変更が起きなかったと言えます。

解決方法

Solaris WBEM Services のセキュリティ機能を使用してメッセージを再送信します。Solaris WBEM Services のセキュリティについての詳細は、『*Solaris WBEM Services* の管理』の「セキュリティの管理」を参照してください。

CIM_ERR_ACCESS_DENIED

説明

CIM_ERR_ACCESS_DENIED エラーメッセージは、パラメータを使用しません。

例

CIM_ERR_ACCESS_DENIED = Insufficient privileges.

原因

このエラーメッセージは、アクションを実行するための適切な特権および権限がユーザーにない場合に表示されます。

解決方法

CIM Object Manager の管理者に、処理を行うための特権を要求します。

CIM_ERR_ALREADY_EXISTS

例 *1: CIM_ERR_ALREADY_EXISTS*

説明

この場合の CIM_ERR_ALREADY_EXISTS エラーメッセージはパラメータ {0} を使用しますが、このパラメータは重複したクラスの名前に置換されています。

例

CIM_ERR_ALREADY_EXISTS = Duplicate class CIMRack

原因

作成しようとしたクラスに、既存のクラスと同じ名前が使用されています。

解決方法

CIM WorkShop で既存のクラスを検索して使用されている名前を確認し、固有の名前を使用してクラスを作成します

例 *2: CIM_ERR_ALREADY_EXISTS*

説明

この場合の CIM_ERR_ALREADY_EXISTS エラーメッセージはパラメータ {0} を使用しますが、このパラメータは重複したインスタンスの名前に置換されています。

例

CIM_ERR_ALREADY_EXISTS = Duplicate instance SolarisRack

原因

作成しようとしたクラスのインスタンスに、既存のインスタンスと同じ名前が使用されています。

解決方法

CIM WorkShop で既存のインスタンスを検索して使用されている名前を確認し、固有の名前を使用してインスタンスを作成します。

例 *3: CIM_ERR_ALREADY_EXISTS*

説明

この場合の CIM_ERR_ALREADY_EXISTS エラーメッセージはパラメータ {0} を使用しますが、このパラメータは重複したネームスペースの名前に置換されています。

例

CIM_ERR_ALREADY_EXISTS = Duplicate namespace root\cimv2

原因

作成が試みられたネームスペースに、既存のネームスペースと同じ名前が使用されています。

解決方法

CIM WorkShop で既存のネームスペースを検索して使用されている名前を確認し、固有の名前を使用してネームスペースを作成します。

例 *4: CIM_ERR_ALREADY_EXISTS*

説明

この場合の CIM_ERR_ALREADY_EXISTS エラーメッセージはパラメータ {0} を使用しますが、このパラメータは重複した修飾子型の名前に置換されています。

例

CIM_ERR_ALREADY_EXISTS = Duplicate qualifier type Key

原因

作成しようとした修飾子型に、変更されるプロパティの既存の修飾子型と同じ名前が使用されています。

解決方法

CIM WorkShop でプロパティの既存の修飾子型を検索して使用されている名前を確認し、固有の名前を使用して修飾子型を作成します。

CIM_ERR_FAILED

説明

CIM_ERR_FAILED エラーメッセージはパラメータ {0} を使用しますが、このパラメータは文字列 (エラー状態と推測される原因を説明したメッセージ) に置換されています。

例

CIM_ERR_FAILED=Invalid entry

原因

CIM_ERR_FAILED エラーメッセージは、さまざまなエラー状況に対して表示される一般的なメッセージです。

解決方法

CIM_ERR_FAILED は一般的なエラーメッセージであるため、このメッセージの原因となり得る状況は多数考えられます。解決方法は、エラー状況によって異なります。

CIM_ERR_INVALID_PARAMETER

説明

CIM_ERR_INVALID_PARAMETER エラーメッセージはパラメータ {0} を使用しますが、このパラメータはスキーマ接頭辞が指定されていないクラスの名前に置換されています。

例

CIM_ERR_INVALID_PARAMETER = Class System has no schema prefix.

原因

クラス名の前にスキーマ接頭辞のないクラスが作成されました。CIM (Common Information Model) では、すべてのクラスにスキーマ接頭辞を付ける必要があります。たとえば、CIM スキーマの一部として開発されるクラスには、CIM 接頭辞 CIM_Container を付けます。Solaris スキーマの一部として開発されたクラスには、Solaris 接頭辞 Solaris_System を付けます。

解決方法

クラス定義に該当するスキーマ接頭辞を指定します。接頭辞のないクラスのインスタンスをすべて見つけ、クラス名と接頭辞に置き換えます。

CIM_ERR_INVALID_SUPERCLASS

説明

メッセージ CIM_ERR_INVALID_SUPERCLASS は、次の 2 つのパラメータを使用します。

- {0} は、指定されたスーパークラスの名前に置換されます。
- {1} は、指定されたスーパークラスが存在しないクラスの名前に置換されます。

例

CIM_ERR_INVALID_SUPERCLASS = Superclass CIM_Chassis for class CIM_Container does not exist.

原因

特定のスーパークラスに属するクラスが指定されましたが、そのスーパークラスは存在しません。指定されたスーパークラスにスペルミスがあるか、あるいは意図したスーパークラス名の代わりに誤って存在しないスーパークラス名が指定されたことが考えられます。また、そのスーパークラスとサブクラスに変更があった可能性もあります。たとえば、指定されたスーパークラスは、実際は指定されたクラスのサブクラスであるかもしれません。この例では、CIM_Container のスーパークラスとして CIM_Chassis が指定されていますが、CIM_Chassis は CIM_Container のサブクラスです。

解決方法

スーパークラスのスペルと名前が正しいか確認し、ネームスペース内にそのスーパークラスが存在することを確認します。

```
CIM_ERR_NOT_FOUND
```

例 *1: CIM_ERR_NOT_FOUND*

説明

この場合の CIM_ERR_NOT_FOUND エラーメッセージはパラメータ {0} を使用しますが、このパラメータは存在しないクラスの名前に置換されています。

例

CIM_ERR_NOT_FOUND = Class Solaris_Device does not exist.

原因

指定されたクラスが存在しません。指定されたクラスにスペルミスがあるか、あるいは意図したクラス名の代わりに誤って存在しないクラス名が指定されたことが考えられます。

解決方法

クラスのスペルと名前が正しいか確認し、ネームスペース内にそのクラスが存在することを確認します。

例 *2: CIM_ERR_NOT_FOUND*

説明

この場合の CIM_ERR_NOT_FOUND エラーメッセージ は、次の 2 つのパラメータを使用します。

- {0} は、指定されたインスタンスの名前に置換されます。

- {1} は、指定されたクラスの名前に置換されます。

例

CIM_ERR_NOT_FOUND = Instance Solaris_EnterpriseData does not exist for class Solaris_ComputerSystem.

原因

インスタンスが存在しません。

解決方法

インスタンスを作成します。

例 *3: CIM_ERR_NOT_FOUND*

説明

この場合の CIM_ERR_NOT_FOUND エラーメッセージはパラメータ {0} を使用しますが、このパラメータは指定されたネームスペースの名前に置換されています。

例

CIM_ERR_NOT_FOUND = Namespace verdant does not exist.

原因

指定されたネームスペースが見つかりません。このエラーは、入力ミスまたはスペルミスのために入力されたネームスペースの名前が正しくない場合に発生します。

解決方法

ネームスペースの名前を入力し直し、入力とスペルが正しいことを確認します。

CLASS_REFERENCE

説明

CLASS_REFERENCE エラーメッセージは、次の 2 つのパラメータを使用します。

- {0} パラメータは、参照関係を定義されたクラスの名前に置換されます。
- {1} パラメータは、参照の名前に置換されます。

例

CLASS_REFERENCE = Class SolarisExample1 must be declared as an association to have reference SolarisExample2

原因

あるクラスに、そのクラスが参照を持つことを示すプロパティが定義されました。しかし、そのクラスは関連の一部ではありません。クラスが参照をプロパティとして持つことを定義できるのは、別のクラスとの関連がある場合だけです。

解決方法

関連を作成し、その関連に対する参照をこのクラスのプロパティとして設定します。

INVALID_CREDENTIAL

説明

INVALID_CREDENTIAL エラーメッセージは、パラメータを使用しません。

例

INVALID_CREDENTIAL = Invalid credentials.

原因

このエラーメッセージは、無効なパスワードが入力された場合に表示されます。

解決方法

このメッセージを CIM WorkShop から受けた場合は、「CIM WorkShop 認証 (CIM WorkShop authentication)」ダイアログボックスの「パスワード (Password)」フィールドから無効なパスワードを削除し、パスワードを入力し直します。このエラーメッセージを MOF コンパイラから受け取った場合は、システムプロンプトでログインし直し、正しいスペルでパスワードを入力します。

INVALID_QUALIFIER_NAME

説明

INVALID_QUALIFIER_NAME エラーメッセージはパラメータ {0} を使用しますが、このパラメータは空の修飾子名を表す MOF (Managed Object Format) 表記に置換されています。

例

INVALID_QUALIFIER_NAME = Invalid qualifier name `` ``

原因

プロパティの修飾子が作成されましたが、修飾子の名前が指定されませんでした。

解決方法

修飾子の定義文に修飾子の名前を含めます。

KEY_OVERRIDE

説明

KEY_OVERRIDE エラーメッセージは、次の 2 つのパラメータを使用します。

- {0} パラメータは、1 つ以上のキー修飾子を持つクラスとのオーバーライド関係に置かれる非 abstract クラスの名前に置換されます。
- {1} パラメータは、キー修飾子を持つ非 abstract クラスの名前に置換されます。

例

KEY_OVERRIDE = Non-key Qualifier SolarisCard cannot override key Qualifier SolarisLock.

原因

非 abstract クラスが、1 つ以上のキー修飾子を持つ非 abstract クラスをオーバーライドするようになっています。CIM では、すべての非 abstract クラスは 1 つ以上のキー修飾子を必要とし、キークラス以外のクラスはキーを持つクラスをオーバーライドできません。

解決方法

非キークラスにキー修飾子を作成します。

KEY_REQUIRED

説明

KEY_REQUIRED エラーメッセージはパラメータ {0} を使用しますが、このパラメータはキーを必要とするクラスの名前に置換されています。

例

KEY_REQUIRED = Concrete (non-abstract) class ClassName needs at least one key.

原因

非 abstract にキー修飾子が指定されませんでした。CIM では、非 abstract クラスはすべて、1 つ以上の修飾子を必要とします。

解決方法

クラスにキー修飾子を作成します。

METHOD_OVERRIDDEN

説明

METHOD_OVERRIDDEN コマンドは、次の 3 つのパラメータを使用します。

- {0} は、パラメータ {1} で示されるメソッドのオーバーライドを試みているメソッドの名前に置換されます。
- {1} は、パラメータ {2} で示されるメソッドによってすでにオーバーライドされているメソッドの名前に置換されます。
- {2} は、パラメータ {1} をオーバーライドしたメソッドの名前に置換されます。

例

METHOD_OVERRIDDEN = Method `Resume ()` cannot override `Stop()` which is already overridden by `Start()`

原因

別のメソッドによってすでにオーバーライドされているメソッドのオーバーライドを試みるメソッドが指定されました。オーバーライド済みのメソッドを再度オーバーライドすることはできません。

解決方法

オーバーライドする別のメソッドを指定します。

NEW_KEY

説明

NEW KEY エラーメッセージは、次の 2 つのパラメータを使用します。

- {0} は、キーの名前に置換されます。
- {1} は、新しいキーの定義を試みているクラスの名前に置換されます。

例

NEW_KEY = Class `CIM_PhysicalPackage` cannot define new key `[Key]`

原因

あるクラスが新しいキーの定義を試みていますが、スーパークラス内にキーがすでに定義されています。スーパークラスにいったんキーが定義されると、サブクラスに新しいキーを設定することはできません。

解決方法

解決方法はありません。

NO_CIMOM

説明

NO_CIMOM エラーメッセージはパラメータ {0} を使用しますが、このパラメータは CIM Object Manager の実行ホストに指定されたホストの名前に置換されています。

例

NO_CIMOM = CIMOM molly not detected.

原因

CIM Object Manager が指定されたホストで動作していません。

解決方法

接続を試みているホストで CIM Object Manager が動作していることを確認します。そのホストで CIM Object Manager が動作していない場合は、このソフトウェアが動作しているホストに接続します。

NO_INSTANCE_PROVIDER

説明

NO_INSTANCE_PROVIDER エラーメッセージは、次の 2 つのパラメータを使用します。

- {0} は、インスタンスプロバイダが見つからないクラスの名前に置換されます。
- {1} は、指定されたインスタンスプロバイダの名前に置換されます。

例

NO_INSTANCE_PROVIDER = Instance provider RPC_prop for class RPC_Agent not found.

原因

指定されたインスタンスプロバイダの Java クラスが見つかりません。このエラーメッセージは、CIM Object Manager のクラスパスに以下のすべてが適切であるクラスが含まれていないことを示します。

- プロバイダクラスの名前
- プロバイダクラスのパラメータ
- プロバイダが定義される CIM クラス

解決方法

CIM Object Manager のクラスパスを設定します。

`NO_METHOD_PROVIDER`

説明

`NO_METHOD_PROVIDER` エラーメッセージは、次の 2 つのパラメータを使用します。

- `{0}` は、メソッドプロバイダが見つからないクラスの名前に置換されます。
- `{1}` は、指定されたメソッドプロバイダの名前に置換されます。

例

`NO_METHOD_PROVIDER` = Method provider `Start_prop` for class `RPC_Agent` not found.

原因

指定されたメソッドプロバイダの Java クラスが見つかりません。このエラーメッセージは、CIM Object Manager のクラスパスに以下のすべてが適切であるクラスが含まれていないことを示します。

- プロバイダクラスの名前
- プロバイダクラスのパラメータ
- プロバイダが定義される CIM クラス

解決方法

CIM Object Manager のクラスパスを設定します。

NO_OVERRIDDEN_METHOD

説明

エラーメッセージ NO_OVERRIDDEN_METHODは、次の 2 つのパラメータを使用します。

- {0} は、{1} で示されるメソッドをオーバーライドしたメソッドの名前に置換されます。
- {1} は、オーバーライドされたメソッドの名前に置換されます。

例

NO_OVERRIDDEN_METHOD = Method `Write` overridden by `Read` does not exist in class hierarchy.

原因

サブクラスのメソッドがスーパークラスのメソッドのオーバーライドを試みていますが、スーパークラスのメソッドはすでに別のサブクラスに属しているメソッドによってオーバーライドされています。

メソッドをオーバーライドすると、その実装と署名もオーバーライドされます。

解決方法

スーパークラス内にそのメソッドが存在することを確認します。

NO_OVERRIDDEN_PROPERTY

説明

NO_OVERRIDDEN_PROPERTY エラーメッセージは、次の 2 つのパラメータを使用します。

- {0} は、{1} をオーバーライドしたプロパティの名前に置換されます。
- {1} は、プロパティをオーバーライドする名前に置換されます。

例

NO_OVERRIDDEN_PROPERTY = Property `A` overridden by `B` does not exist in class hierarchy.

原因

サブクラスのプロパティがスーパークラスのプロパティのオーバーライドを試みていますが、スーパークラスのプロパティはすでにオーバーライドされているため、このオーバーライドを行うことはできません。

解決方法

スーパークラスにそのプロパティが存在することを確認します。

`NO_PROPERTY_PROVIDER`

説明

`NO_PROPERTY_PROVIDER` エラーメッセージは、次の 2 つのパラメータを使用します。

- `{0}` は、プロパティプロバイダが見つからないクラスの名前に置換されます。
- `{1}` は、指定されたプロパティプロバイダの名前に置換されます。

例

`NO_PROPERTY_PROVIDER` = Property provider `Write_prop` for class `RPC_Agent` not found.

原因

指定されたプロパティプロバイダの Java クラスが見つかりません。このエラーメッセージは、CIM Object Manager のクラスパスに以下のすべてが適切であるクラスが含まれていないことを示します。

- プロバイダクラスの名前
- プロバイダクラスのパラメータ
- プロバイダが定義される CIM クラス

解決方法

CIM Object Manager のクラスパスを設定します。

`NO_QUALIFIER_VALUE`

説明

`NO_QUALIFIER_VALUE` エラーメッセージは、次の 2 つのパラメータを使用します。

- `{0}` は、要素 `{1}` を変更する修飾子の名前に置換されます。

- {1} は、修飾子の参照先である要素です。{1} は、修飾子に応じてクラス、プロパティ、メソッド、または参照のいずれかです。

例

NO_QUALIFIER_VALUE = Qualifier [SOURCE] for Solaris_ComputerSystem has no value.

原因

プロパティまたはメソッドに修飾子が指定されましたが、修飾子に値が含まれていません。たとえば、修飾子 VALUES には文字列配列を指定する必要があります。必要な文字列配列なしで VALUES 修飾子が指定されると、NO_QUALIFIER_VALUE エラーメッセージが表示されます。

解決方法

修飾子に必要なパラメータを指定します。各修飾子に必要な属性については、Distributed Management Task Force による CIM Specification (URL: http://dmtf.org/spec/cims.html) を参照してください。

NO_SUCH_METHOD

説明

NO_SUCH_METHOD エラーメッセージは、次の 2 つのパラメータを使用します。

- {0} は、指定されたメソッドの名前に置換されます。
- {1} は、指定されたクラスの名前に置換されます。

例

NO_SUCH_METHOD = Method Configure() does not exist in class Solaris_ComputerSystem

原因

指定されたクラスにメソッドが定義されなかったことが考えられます。指定されたクラスにメソッドが定義されている場合には、定義内でスペルミスにより別のメソッドが指定されたか、入力ミスの可能性があります。

解決方法

指定されたクラスのオペレーションとしてメソッドを定義するか、あるいはメソッド名とクラス名が正しく入力されているか確認します。

NO_SUCH_PRINCIPAL

説明

NO_SUCH_PRINCIPAL エラーメッセージはパラメータ {0} を使用しますが、このパラメータはプリンシパル (ユーザーアカウント) の名前に置換されています。

例

NO_SUCH_PRINCIPAL = Principal molly not found.

原因

指定されたユーザーアカウントが見つかりません。ログイン時にユーザー名の入力が正しく行われなかったか、あるいはそのユーザーにユーザーアカウントが設定されていません。

解決方法

ログイン時にユーザー名を正しく入力します。そのユーザーにユーザーアカウントが設定されていることを確認します

NO_SUCH_QUALIFIER1

説明

NO_SUCH_QUALIFIER1 エラーメッセージはパラメータ {0} を使用しますが、このパラメータは未定義の修飾子の名前に置換されています。

例

NO_SUCH_QUALIFIER1 = Qualifier [LOCAL] not found.

原因

新しい修飾子が指定されましたが、この修飾子は拡張スキーマの一部として定義されていません。特定のクラスのプロパティまたはメソッドに有効な修飾子として認識されるように、この修飾子を CIM スキーマまたは拡張スキーマの一部として定義する必要があります。

解決方法

この修飾子を拡張スキーマの一部として定義するか、あるいは標準の CIM 修飾子を使用します。標準の CIM 修飾子と CIM スキーマの修飾子の使用法については、Distributed Management Task Force による CIM Specification (URL: http://www.dmtf.org/spec/cims.html) を参照してください。

NO_SUCH_QUALIFIER2

説明

NO_SUCH_QUALIFIER2 エラーメッセージは、次の 2 つのパラメータを使用します。

- {0} は、修飾子が変更するクラス、プロパティ、またはメソッドの名前に置換されます。
- {1} は、定義されていない修飾子の名前に置換されます。

例

NO_SUCH_QUALIFIER2 = Qualifier [LOCAL] not found for CIM_LogicalElement

原因

特定のクラスのプロパティまたはメソッドを変更するために新しい修飾子が指定されましたが、その修飾子は拡張スキーマの一部として定義されていません。特定のクラスのプロパティまたはメソッドに有効な修飾子として認識されるように、この修飾子を CIM スキーマまたは拡張スキーマの一部として定義する必要があります。

解決方法

この修飾子を拡張スキーマの一部として定義するか、あるいは標準の CIM 修飾子を使用します。標準の CIM 修飾子と CIM スキーマの修飾子の使用法については、Distributed Management Task Force による CIM Specification (URL: http://www.dmtf.org/spec/cims.html) を参照してください。

NO_SUCH_SESSION

説明

エラーメッセージ NO_SUCH_SESSION はパラメータ {0} を使用しますが、このパラメータはセッション識別子に置換されています。

例

NO_SUCH_SESSION = No such session 4002.

原因

このメッセージは、セッションが不正侵入者によって侵害された場合に表示されます。CIM Object Manager は、第三者が故意にデータ変更を試みていることを検出すると、セッションを削除します。Solaris WBEM Services のセキュリティ機能についての詳細は、『*Solaris WBEM Services* の管理』の「セキュリティの管理」を参照してください。

解決方法

CIM 環境のセキュリティが保護されていることを確認します。

NOT_HELLO

説明

NOT_HELLO エラーメッセージは、パラメータを使用しません。

例

NOT_HELLO = Not a Hello message.

原因

このエラーメッセージは、hello メッセージ (CIM Object Manager に送信される最初のメッセージ) 内のデータが破損している場合にセキュリティが侵害されたことを示すために表示されます。

解決方法

このエラーメッセージに対しての解決方法はありません。Solaris WBEM Services のセキュリティ機能についての詳細は、『*Solaris WBEM Services* の管理』の「セキュリティの管理」を参照してください。

NOT_INSTANCE_PROVIDER

説明

NOT_INSTANCE_PROVIDER エラーメッセージは、次の 2 つのパラメータを使用します。

- {0} は、InstanceProvider インタフェースの定義が試みられているインスタンスの名前に置換されます。
- {1} は、InstanceProvider インタフェースを実装していない Java プロバイダクラスの名前に置換されます。指定されたクラスのインスタンスをすべて列挙するには、InstanceProvider インタフェースを指定する必要があります。

例

NOT_INSTANCE_PROVIDER = device_prop_provider for class Solaris_Provider does not implement InstanceProvider.

原因

CLASSPATH 環境変数で指定されている Java プロバイダクラスが、InstanceProvider インタフェースを実装していません。

解決方法

クラスパスに存在する Java プロバイダクラスが InstanceProvider インタフェースを実装していることを確認します。プロバイダを宣言するには、クラス定義に public class <Solaris> implements InstanceProvider を使用します。Solaris WBEM Services プロバイダの実装方法については、第 5 章を参照してください。

NOT_METHOD_PROVIDER

説明

NOT_METHOD_PROVIDER エラーメッセージは、次の 2 つのパラメータを使用します。

- {0} は、MethodProvider インタフェースの定義が試みられているメソッドの名前に置換されます。MethodProvider が定義されると、指定されたメソッドがプログラム内で実装され、実行されます。
- {1} は、MethodProvider インタフェースを実装していない Java プロバイダクラスの名前に置換されます。

例

NOT_METHOD_PROVIDER = Provider device_method_provider for class Solaris_Provider does not implement MethodProvider.

原因

クラスパスに存在する Java プロバイダクラスが MethodProvider インタフェースを実装していません。

解決方法

クラスパスに存在する Java プロバイダクラスが MethodProvider インタフェースを実装していることを確認します。プロバイダを宣言するには、クラス定義に public class <Solaris> implements MethodProvider を使用します。Solaris WBEM Services プロバイダの実装方法については、第 5 章を参照してください。

NOT_PROPERTY_PROVIDER

説明

NOT_PROPERTY_PROVIDER エラーメッセージは、次の 2 つのパラメータを使用します。

- {0} は、PropertyProvider インタフェースの定義が試みられているメソッドの名前に置換されます。PropertyProvider インタフェースは、指定されたプロパティの値の検出に使用されます。
- {1}は、PropertyProvider インタフェースを実装していない Java プロバイダクラスの名前に置換されます。

例

NOT_PROPERTY_PROVIDER = Provider device_property_provider for class Solaris_Provider does not implement PropertyProvider.

原因

クラスパスに存在する Java プロバイダクラスが PropertyProvider インタフェースを実装していません。

解決方法

クラスパスに存在する Java プロバイダクラスが PropertyProvider インタフェースを実装していることを確認します。プロバイダを宣言するには、コマンド public class <Solaris> implements PropertyProvider を使用します。Solaris WBEM Services プロバイダの実装方法については、第 5 章を参照してください。

NOT_RESPONSE

説明

NOT_RESPONSE エラーメッセージは、パラメータを使用しません。

例

NOT_RESPONSE = Not a response message.

原因

このエラーメッセージは、CIM Object Manager からの最初の応答メッセージが破損している場合にセキュリティが侵害されていることを示すために表示されます。

解決方法

このエラーメッセージに対しての解決方法はありません。Solaris WBEM Services のセキュリティ機能についての詳細は、『*Solaris WBEM Services* の管理』の「セキュリティの管理」を参照してください。

PROPERTY_OVERRIDDEN

説明

PROPERTY_OVERRIDDEN エラーメッセージは、次の 3 つのパラメータを使用します。

- {0} は、パラメータ {1} によって示されるプロパティのオーバーライドを試みているプロパティの名前に置換されます。
- {1} は、すでにオーバーライドされているプロパティの名前に置換されます。
- {2} は、パラメータ {1} で示されるプロパティをオーバーライドしたプロパティの名前に置換されます。

例

PROPERTY_OVERRIDDEN = Property Volume cannot override MaxCapacity which is already overridden by RawCapacity

原因

別のプロパティによってすでにオーバーライドされているプロパティのオーバーライドを試みるプロパティが指定されました。オーバーライド済みのプロパティを再度オーバーライドすることはできません。

解決方法

オーバーライドする別のプロパティを指定します。

PS_CONFIG

説明

PS_CONFIG エラーメッセージは、エラー原因となる状況の説明に置換されるパラメータ {0} を使用します。この説明は、リポジトリに使用されているデータベースの種類と、エラーメッセージを生成する状況の種類によって異なります。

例

PS_CONFIG = The persistent store configuration is incorrect or has not been completed. You may need to run the wbemconfig script.

原因

Solaris WBEM Services では、インストール後 wbemconfig スクリプトを実行する必要があります。wbemconfig スクリプトは、固定記憶域の設定と、CIM スキーマクラスと Solaris スキーマクラスを提供する MOF ファイルのコンパイルを

行います。このエラーメッセージは、Solaris WBEM Services がインストールされた後 wbemconfig スクリプトが実行されなかった場合に表示されます。インストールの後にリポジトリが設定されてからこのエラーメッセージが表示される場合は、データベースの設定が壊れている可能性があります。

解決方法

wbemconfig スクリプトを実行します。wbemconfig スクリプトについての詳細は、『*Solaris WBEM Services* の管理』を参照してください。

PS_UNAVAILABLE

説明

PS_UNAVAILABLE エラーメッセージはパラメータ {0} を使用しますが、このパラメータは固定記憶域が使用できなくなった理由を説明するメッセージに置換されています。

例

PS_UNAVAILABLE = The persistent store is unavailable. The exception thrown by the repository is 'segmentation fault.'

原因

このエラーメッセージは、CIM Repository が使用できない場合に表示されます。このような状況は、CIM Repository が常駐しているホストが保守のために一時的に停止される場合や、このホストが破損したためにリポジトリが停止されて別のホストに復元される場合などに発生します。

解決方法

CIM WorkShop での作業中にこのメッセージが表示される場合は、「CIM WorkShop オーセンティケーション (CIM WorkShop authentication)」ダイアログボックスを表示するアイコンをクリックします。続いて、「ホスト (Host)」フィールドで、CIM Repository と CIM Object Manager を実行している別のホストの名前を入力します。「ネームスペース (Namespace)」フィールドにネームスペース名を入力し、続いてユーザー名とパスワードを入力してログインします。MOF コンパイラの実行時にこのメッセージを受け取る場合は、CIM Repository と CIM Object Manager を実行している別のホストを指定するコマンド mofcomp -c *hostname* を入力します。mofcomp は MOF コンパイラを起動するコマンド、-c は CIM Object Manager を実行しているホストコンピュータを指定するためのパラメータ、*hostname* は指定されるコンピュータの名前です。

QUALIFIER_UNOVERRIDABLE

説明

QUALIFIER_UNOVERRIDABLE エラーメッセージは、次の 2 つのパラメータを使用します。

- {0} は、DisableOverride フレーバが設定されている修飾子の名前に置換されます。
- {1} は、{0} によって無効になるように設定されている修飾子の名前に置換されます。

例

QUALIFIER_UNOVERRIDABLE = Test cannot override qualifier Standard because it has DisableOverride flavor.

原因

指定された修飾子のフレーバが DisableOverride または Override=False に設定されているため、この修飾子は別の修飾子をオーバーライドできません。

解決方法

この修飾子の特性を、EnableOverride または Override=True に設定し直します。

REF_REQUIRED

説明

REF_REQUIRED エラーメッセージはパラメータ {0} を使用しますが、このパラメータは関連を持つように指定されたクラスの名前に置換されています。

例

REF_REQUIRED = Association class CIM_Chassis needs at least two refs.

原因

関連を持つようにあるクラスが設定されましたが、参照が指定されていません。CIM (Common Information Model) では、関連は 2 つ以上の参照を含む必要があります。

解決方法

クラスに参照を設定し、その後関連を設定します。

SCOPE_ERROR

説明

SCOPE_ERROR コマンドは、次の 3 つのパラメータを使用します。

- {0} は、指定された修飾子が変更するクラスの名前に置換されます。
- {1} は、指定された修飾子の名前に置換されます。
- {2} は、修飾子が変更する属性のタイプに置換されます。

例

SCOPE_ERROR = Qualifier [UNITS] for CIM_Container does not have a Property scope.

原因

CIM Specification に準拠しない方法で修飾子が指定されました。たとえば、[READ] 修飾子は、CIM Specification ではプロパティを変更するように定義されます。[READ] 修飾子のスコープは、プロパティを変更するように [READ] 修飾子に指示する定義です。[READ] 修飾子がそのスコープの指示以外の方法で使用されると (たとえばメソッドを変更するように指定されるなど)、SCOPE_ERROR メッセージが返されます。

注 - CIM Specification は、CIM 修飾子が変更できる CIM 要素の種類を定義しています。修飾子の使用方法についてのこの定義は、修飾子のスコープと呼ばれます。ほとんどの修飾子は、プロパティまたはメソッド、あるいはこの両方の変更を指示するスコープを持ちます。また、ほとんどの修飾子は、パラメータ、クラス、関連、インジケーション、またはスキーマの変更を指示するスコープを持ちます。

解決方法

指定された修飾子のスコープを確認します。CIM 修飾子の標準の定義については、Distributed Management Task Force により提供されている CIM Specification の「1. Qualifiers」(URL: http://www.dmtf.org/spec/cim_spec_v20) を参照してください。別の修飾子を使用するか、あるいは CIM 定義に従って修飾子を使用するようにプログラムを変更します。

SIGNATURE_ERROR

説明

SIGNATURE_ERROR エラーメッセージは、パラメータを使用しません。

例

SIGNATURE_ERROR = Signature not verified

原因

このメッセージは、メッセージが偶然にまたは故意に破損された場合に表示されます。このメッセージは、メッセージが有効なチェックサムを持つチェックサムエラーとは異なりますが、署名はクライアントの公開鍵では検証できません。この保護により、セッションキーが侵害されることがあっても、セッションを作成した最初のクライアントだけが認証されます。

解決方法

セッションが不正侵入者によって侵害される場合に表示されるこのメッセージに対しての解決方法はありません。Solaris WBEM Services のセキュリティ機能についての詳細は、『*Solaris WBEM Services* の管理』の「セキュリティの管理」を参照してください。

TYPE_ERROR

説明

TYPE_ERROR エラーメッセージは、次の 5 つのパラメータを使用します。

- {0} は、指定された要素 (プロパティ、メソッド、修飾子など) の名前に置換されます。
- {1} は、指定された要素が属するクラスの名前に置換されます。
- {2} は、要素に定義されたデータ型に置換されます。
- {3} は、割り当てられた値のデータ型に置換されます。
- {4} は、割り当てられた実際の値に置換されます。

例

TYPE_ERROR = Cannot convert sint16 4 to a string for VolumeLabel in class Solaris_DiskPartition

原因

プロパティまたはメソッドのパラメータ値と、定義されたそのデータ型が一致しません。

解決方法

プロパティまたはメソッドの値を、定義されたそのデータ型に一致させます。

UNKNOWNHOST

説明

UNKNOWNHOST エラーメッセージはパラメータ {0} を使用しますが、このパラメータはホストの名前に置換されています。

例

UNKNOWNHOST = Unknown host molly

原因

指定されたホストが使用できないか、あるいはこのホストが見つかりません。ホスト名のスペルが間違っている可能性もあります。このホストコンピュータが別のドメインに移されたか、あるいはドメインに属するホストのリストにそのホスト名が登録されていない可能性もあります。また、システム状況が原因でそのホストが一時的に使用できないことも考えられます。

解決方法

ホスト名のスペルを調べ、入力ミスがないか確認します。ping コマンドを使用して、そのホストコンピュータが応答していること、そのホストのシステム状況を確認します。また、そのホストが指定されたドメインに属していることを確認します。

VER_ERROR

説明

VER_ERROR エラーメッセージはパラメータ {0} を使用しますが、このパラメータは動作中の CIM Object Manager のバージョン番号に置換されています。

例

VER_ERROR = Unsupported version 0.

原因

Solaris WBEM Services のこのアップグレードバージョンは、現在の CIM Object Manager をサポートしていません。

解決方法

サポートされているバージョンをインストールします。

付録 **A**

CIM の用語と概念

CIM の概念

ネットワークエンティティと管理機能が CIM (Common Information Model) のコンテキスト内でどのように表現され関連しているかを理解する上で重要な、CIM の基本的な用語と概念について説明します。CIM とオブジェクト指向モデルの慣例 (独自のスキーマのモデル化など) についての詳細は、Distributed Management Task Force が提供している `http://dmtf.org/spec/cim_tutorial` の CIM Tutorial を参照してください。

オブジェクト指向モデル

CIM では、物理的または論理的に存在するオブジェクト、エンティティ、概念、または機能の表現手段として、オブジェクト指向モデルの原理を使用しています。オブジェクト指向モデルの目的は、物理的なエンティティをフレームワーク (モデル) で設定し、エンティティの特性と機能、およびエンティティとほかのエンティティとの関係を表現することです。CIM では、オブジェクト指向モデルを使用して、ハードウェア要素とソフトウェア要素をモデル化します。

UML (Uniform Modeling Language)

UML (Uniform Modeling Language) モデルは、図と言語で表されます。モデルを表現するための CIM 規則は、UML の図示概念に基づいています。UML は、図を使

用して物理的なエンティティを表現し、線を使用して関係を表現します。たとえば、UML ではクラスは矩形として表されます。各矩形には、表現対象であるクラスの名前が入ります。2 つの矩形間の線は、それらのクラスの関係を示します。2 つのクラスを上位クラスに結合するために分岐する線は、関連を示します。

CIM ダイアグラムは、色を使用して関係を詳しく説明します。

- 赤色の線 → 関連
- 青色の線 → 継承関係
- 緑色の線 → 集合

CIM の用語

次の用語は、CIM スキーマ固有の意味を持ちます。

スキーマ

モデル、スキーマ、およびフレームワークは同義語です。これらはそれぞれ、物理的または論理的に存在するエンティティの抽象表現です。CIM では、スキーマはクラスの名前付けと管理に使用される、名前が付けられたクラスの集まりを意味します。スキーマ内では、クラスとそのサブクラスは構文 `Schemaname_classname` によって階層的に表現されます。スキーマ内の各クラス名は、一意である必要があります。Solaris WBEM には、Solaris スキーマが付属しています。Solaris スキーマには、CIM 機能を Solaris 用に独自に拡張したすべてのクラスが含まれます。

クラスとインスタンス

WBEM では、クラスは最も基本的な管理ユニットを表現するオブジェクトの集まりを意味します。たとえば、Solaris WBEM の主要な機能クラスとして、`CIMClass`、`CIMProperty`、`CIMInstance` が挙げられます。

クラスは、管理対象オブジェクトを作成するために使用される抽象的な概念です。クラスの特性は、そのクラスから作成される子オブジェクト (インスタンス) によって継承されます。たとえば、`CIMClass` を使用してインスタンス `CIMClass (Solaris_Computer_System)` を作成できます。

この CIMClass インスタンスは、「そのコンピュータシステムは何か」という質問に答えます。インスタンスの値は、Solaris_Computer_System です。同じクラスタイプのインスタンスはすべて、同じクラステンプレートから作成されます。この例では、CIMClass が、タイプ Computer_System の管理対象オブジェクトを作成するテンプレートを提供します。

クラスには、静的クラスと動的クラスがあります。静的クラスのインスタンスは、CIM Object Manager によって格納され、要求がある場合に CIM Repository から取り出すことができます。動的クラス (システム使用量のような常に変化するデータを含むクラス) のインスタンスは、データの変化に伴いプロバイダアプリケーションによって作成されます。

カスタムクラス: CIM の拡張機能

管理環境に固有の管理対象オブジェクトをサポートするために、CIM の拡張機能としてカスタムクラスを開発できます。CIM Object Manager API は、Solaris オペレーティング環境向けに CIM を拡張する新しいクラスを提供します。

プロパティ

プロパティは、クラスの特性を定義します。たとえば、CIMProperty クラスを使用して、キーを特定の CIM クラスのプロパティとして定義できます。プロパティの値は、文字列、またはプロパティ範囲のベクトルとして CIM Object Manager から渡すことができます。各プロパティは、固有の名前と 1 つのドメイン (そのプロパティを所有するクラス) を持ちます。一定のクラスのプロパティは、そのサブクラスのプロパティによってオーバーライドすることができます。

Sun WBEM のプロパティとしては、CIMClass のプロパティである CIMProperty などがあります。

メソッド

プロパティと同様に、メソッドはそれらを所有するクラスに属します。メソッドは、一定のクラスのオブジェクトが実行するアクションです。たとえば、メソッド public String getName() は、インスタンスの名前を、そのキーとキーの値を連続した値として返します。これらのアクションは、集合的にクラスの動作を表現します。メソッドは、そのメソッドを所有するクラスにしか属すことができません。1 つのクラスのコンテキスト内では、各メソッドは固有の名前を持つ必要があ

ります。一定のクラスのメソッドは、そのサブクラスのメソッドによってオーバーライドすることができます。

新しいクラスはスーパークラスからメソッドの定義を継承しますが、スーパークラスの実装は継承しません。修飾子によって示されるメソッドの定義は、実装される新しいメソッドを提供できるプレースホルダとしての役割を果たします。CIM Object Manager は、ツリー内で下位レベルクラスからルートクラスの方向にメソッドを順に検査し、メソッドを示す修飾子型を検索します。

ドメイン

プロパティおよびメソッドは、クラス内で宣言されます。プロパティまたはメソッドを所有するクラスは、プロパティまたはメソッドのドメインと呼ばれます。

修飾子とフレーバ

CIM 修飾子は、CIM のクラス、プロパティ、メソッド、およびパラメータの特性を示すために使用されます。修飾子は、新しいクラスによって継承される固有の属性 (名前、型、値など) を持ちます。

インジケーション

インジケーション (オブジェクトとクラスのタイプ) は、イベント発生の結果として作成され、タイプ階層で示されます。インジケーションは、プロパティ、メソッド、およびトリガーを持つことができます。トリガーは、既存のクラスに対する変更や、新しいインジケーションインスタンスの作成を引き起こすイベントなどのシステムオペレーションです。

関連

関連は、2 つ以上のクラス間の関係を表現するクラスです。関連を使用すると、一定のクラスに複数の関連インスタンスを作成し、システムコンポーネントをさまざまな方法で関連付けることができます。関連は、システムコンポーネントの関係を表現する手段を提供します。

関連の定義方法によって、関係するどのクラスにも影響を与えずにクラス間の関係を構築できます。関連を追加しても、関係するクラスのインタフェースには影響しません。関連だけが、参照を含むことができます。

参照と範囲

参照はプロパティの一種で、関連に関係するオブジェクトの役割を定義します。参照は、関連におけるクラスの役割名を指定します。参照のドメインは、関連です。参照の範囲は、参照の種類を示す文字列で表されます。

オーバーライド

オーバーライド関係は、スーパークラスから継承されたプロパティまたはメソッドを、サブクラスから継承されたプロパティまたはメソッドで置換することを示すために使用されます。CIM では、プロパティとメソッドのどの修飾子がオーバーライドできるかをガイドラインで定めています。たとえば、CIM ガイドラインはキープロパティはオーバーライドできないと定めているため、クラスの修飾子型がキーと設定される場合、キーをオーバーライドできません。

コアモデルの概念

次の節では、CIM のコアモデルについて説明しています。

システムとしてのコアモデル

コアモデルでは、システムとそれらの機能が管理対象オブジェクトとして記述されるアプリケーションを開発するために使用できるクラスと関連を提供します。これらのクラスと関連は、システムを構成するすべての要素 (物理的な要素および論理的な要素) に固有の特性を与えています。物理的な特性とは、空間を占有し、物理的な基本原則に従う性質を意味します。論理的な特性とは、物理的な環境面 (システム状態やシステムの能力など) を管理、調整するために使用される抽象概念を意味します。

コアモデルには次の論理要素が存在します。

表 A–1　コアモデルの要素

要素名	説明
システム	ほかの論理要素をグループ化したもの。システムはそれ自体が論理要素であるため、ほかのシステムのコンポーネントとなり得る
ネットワークコンポーネント	ネットワークの接続形態を提供するクラス
サービスとアクセスポイント	システム機能にアクセスできる構造を構成するための機構を提供する
デバイス	ハードウェアエンティティの抽象概念またはエミュレーション。デバイスは、実際のハードウェアとしての形をとらないこともある

次の節では、システムの特性をエミュレートするためにコアモデルに提供されているクラスと関連について説明します。

コアモデルが提供するシステムクラス

次の表に、システムとしてのコアスキーマを表すクラスを示します。これらのクラスのインスタンスは、通常、そのクラスに含まれるオブジェクトの下位に属します。

表 A–2　コアモデルのシステムクラス

クラス名	説明	例
`Managed System Element`	システム要素階層の基底クラス。識別可能なシステムコンポーネントはすべて、このクラスに含めることができる	ファイルやデバイス (ディスクドライブやコントローラなど) のようなソフトウェアコンポーネント、およびチップやカードのような物理的なコンポーネント
`Logical Element`	抽象的なシステムコンポーネントを表すすべてのシステムコンポーネントの基底クラス	論理デバイスの形をしたプロファイル、プロセス、またはシステム機能

表 A–2 コアモデルのシステムクラス 続く

System	一連の管理対象システム要素の列挙可能な集合である論理要素。この集合は、機能的に一体として動作する。System のすべてのサブクラスには、インスタンスの集合が必要な Managed System Element クラスの明確なリストが存在する	LAN、WAN、サブネット、イントラネット
Service	デバイスまたはソフトウェア機能 (あるいはこの両方) によって提供される機能の記述と管理に必要な情報を含む論理要素。Service は、機能の実装を設定、管理する多目的オブジェクトである。Service 自体は機能ではない	プリンタ、モデム、ファックスマシン

コアモデルが提供するシステム関連

関連は、ほかのクラスによって共有される関係を定義するクラスです。関連クラスには、そのクラスの目的を示す ASSOCIATION 修飾子が付けられます。関連クラスは、2 つ以上の参照 (特定の関係を共有するクラスの名前) を持つ必要があります。関連のインスタンスは、常に関連クラスに属します。

関連には、次のような形式があります。

- 1 対 1
- 1 対多
- 1 対ゼロ
- 集約 (システムとそのコンポーネント間の包含関係など)

関連は、システムとそのコンポーネントである管理対象要素との間の関係を表現します。クラス間の関係の定義には、一般的な 2 種類の関連が使用されます。

CIM スキーマは、次に示す基本的な 2 種類の関連を定義します。

- コンポーネント関連 — あるクラスが別のクラスの一部であることを示す

- 依存関連 — あるクラスが別のクラスなしには機能することも、存在することも不可能であることを示す

これらの関連タイプは抽象型です。つまり、関連クラスは、インスタンスを単独では所有しません。インスタンスはその下位クラスのうちの 1 つに属す必要があります。

コンポーネント関連

コンポーネント関連は、システムのコンポーネントとシステム自体の関係を表します。コンポーネント関連は、どの要素がシステムを構成するかを示します。コンポーネント関連を表す abstract クラスは、下位クラスにこのタイプの具体的な関連を作成するために使用されます。下位の具体的な関連は、「コンポーネント (クラス) が、ほかのコンポーネントとどのような構成関係にあるか」を表します。

コンポーネント関連は、その特有の役割として、システムと、システムの論理的なコンポーネントおよび物理的なコンポーネント間の関係を表します。

依存関連

依存関連は、互いに依存し合うオブジェクト間の関係を確立します。コアモデルは、次の依存関係を提供します。

- 機能的—依存オブジェクトは、依存対象であるオブジェクトなしでは機能できない
- 存在 — 依存オブジェクトは、依存対象であるオブジェクトなしでは存在できない

コアモデルには、次の依存関係があります。

表 A–3 コアモデルの依存関係

依存関連	説明
HostedService	サービスとその機能が存在するシステム間の関連。この関連の多重度は、1 対多である。1 つのシステムは、多くのサービスを管理できるため、サービスは、それらを管理しているシステムに強く依存している。 一般に、サービスは、そのサービスを実装する論理デバイスまたはソフトウェア機能が入ったシステム上で管理されるので、このモデルでは、複数のシステムに渡って管理されるサービスは表されない。この場合システムは、単一のホストにそれぞれ置かれているサービスの集合ポイントの役割を果たすアプリケーションシステムとしてモデル化される。
HostedAccessPoint	サービスアクセスポイント (SAP) と SAP 上で提供されているシステム間の関連。この関連の多重度は 1 対多であり、システムに強く依存している。各システムは多数の SAP を管理できる。 このモデルの特徴は、サービスアクセスポイントを、サービスがアクセスできるシステムと同じホストに置くことも別のホストに置くこともできることである。このため、このモデルは分散システム (コンポーネントサービスが複数のホストに置かれるアプリケーションシステム) と分散アクセス (アクセスポイントがほかのシステム上で管理されるサービス) の両方を表すことができる。
ServiceSAPDependency	サービスとサービスアクセスポイント間の関連。この関連は、サービスがその機能を提供するためには参照されている SAP が必要であることを示す。
SAPSAPDependency	2 つの SAP 間の関連のある SAP がそのサービスを利用またはそのサービスに接続するためには別の SAP が必要であることを示す。
ServiceAccessBySAP	サービスのアクセスポイントを識別する関連。たとえば、プリンタは、複数のシステム上で管理される Netware、Apple Macintosh、または Windows のサービスアクセスポイントによってアクセスされる可能性がある。

コアモデルの拡張例

コアモデルから、多数の拡張機能を開発できます。たとえば、Managed System Element クラスの抽象化した Managed Element クラスを追加できます。コアモデルには、この Managed Element クラスの下位クラス (ユーザーや管理者など、管理対象のシステムドメインに含まれないオブジェクトを表現するクラス) を追加できます。

共通モデルスキーマ

共通モデルスキーマは、以下のようなモデルを提供します。

システムモデル

システムモデルは、管理対象の環境を構成する最上位レベルのシステムオブジェクトであるコンピュータ、アプリケーション、およびネットワークシステムを表しています。

デバイスモデル

デバイスモデルは、システムの基本機能 (ストレージ、処理、通信、入出力など) を提供しているシステム上の個々の論理ユニットを表します。システムデバイスはシステムの物理的なコンポーネントそのものと考えがちですが、これは誤りです。これは、管理されるのは実際のコンポーネント自体ではなく、オペレーティングシステム上でのデバイスの記述であるためです。

オペレーティングシステムの記述は、システムの実際のコンポーネントと 1 対 1 では対応していません。たとえば、モデムは個々の実際のコンポーネントに対応できます。モデムは、LAN アダプタとモデムをサポートする多機能カードによって提供することも、システム上で動作する通常のプロセスによって提供することもできます。このモデルを使用または拡張するためには、論理的なデバイスと物理的なコンポーネント間の違いを混同することなく理解することが必要です。

アプリケーション管理モデル

CIM アプリケーション管理モデルは、ソフトウェア製品とアプリケーションの管理に通常必要な情報を詳細に記述した情報モデルです。このモデルは、スタンドアロンのデスクトップアプリケーションから、高性能、マルチプラットフォーム、分散型、インターネットベースのアプリケーションなど、多様なアプリケーション構造に使用できます。このモデルは、単一のソフトウェア製品を記述するのにも、ビジネスシステムを形成している相互に依存した複数のアプリケーションを記述するのにも使用できます。

このアプリケーションモデルの主要な特性の 1 つに、アプリケーションのライフサイクルという考えがあります。アプリケーションは、配付可能、インストール可能、実行可能、実行中のいずれかの状態にあります。各種のオブジェクトの解釈と特性は、アプリケーションをある状態から別の状態に変換するために使用される機構と多くの場合関連づけられて、アプリケーションを記述しています。

ネットワークモデル

ネットワークモデルは、ネットワークトポロジ、ネットワークの接続性、ネットワークアクセスの制御と提供に必要な各種のプロトコルとサービスなど、ネットワーク環境のさまざまな側面を表現します。

物理モデル

物理モデルは、実際の物理的な環境を表します。管理される環境のほとんどは、論理オブジェクト (実際のオブジェクトではなく環境の情報面を記述するオブジェクト) で表されます。システム管理のほとんどは、システム状態を表現、制御する情報の操作に関係しています。実際の物理的な環境に対する操作 (物理ドライブの読み取りヘッドの移動やファンの起動など) はすべて、通常、論理的な環境の操作による間接的な結果として発生するにすぎません。そのため、物理的な環境は一般に直接的な重要性はありません。

一般にシステムの物理的な各部には、計測機構は付けられていません。それらの現在の状態 (およびそれらの存在そのもの) は、システムについてのほかの情報から間接的に推測できるだけです。CIM では、物理モデルは環境のこの側面を表します。このモデルは、システムごとに大きく異なり、また技術の進展に伴い劇的に変化すると考えられます。また、管理環境の物理的な側面についての情報提供を具体的な目的として、アプリケーション、ツール、および環境を個別に使用するために、物理的な環境を追跡、計測することは、常に困難を極めると予測されます。

用語集

この用語集では、Solaris WBEM のマニュアルで使用されている用語について説明します。これらの用語は開発者の間ではよく使用されていますが、WBEM 環境独自のものや、異なる意味で使用されているものもあります。

Backus-Naur Form (BNF)
プログラミング言語の構文を指定するメタ言語。

CIM Object Manager Repository
CIM Object Manager (Common Information Model Object Manager) によって管理される主要な格納領域。この Repository には、管理対象オブジェクトを表現するクラスとインスタンスの定義、およびそれらの間の関係が格納される。

CIM スキーマ
各管理環境で発生する管理対象オブジェクトの表現に使用されるクラス定義の集まり。

「コアモデル」、「共通モデル」、「拡張スキーマ」も参照。

CIM は、メタモデルと標準スキーマに分類される。メタモデルは、スキーマを構成するエンティティの種類を示すと共に、管理対象オブジェクトを表すオブジェクトにそれらのエンティティをどのように結合できるかを定義する。

DMTF (Distributed Management Task Force)
パーソナルコンピュータ (PC) の使用、理解、構成、および管理の簡易化を行なっている業界のコンソーシアム。

IDL (Interface Definition Language)
ある言語で記述されたプログラムまたはオブジェクトが、認識できない言語で記述された別のプログラムと通信できるようにする言語の総称。

MOF (Managed Object Format)
クラスとインスタンスを定義するコンパイラ型言語。MOF コンパイラ (`mofcomp`) は、`.mof` テキストファイルを Java クラスにコンパイルし、そのデータを CIM Object Manager Repository に追加する。MOF を使用するとコードを記述する必要がないため、CIM Object Manager Repository を簡単にかつ速く変更できる。

MOF ファイル
MOF (Managed Object Format) 言語を使用するクラスとインスタンスの定義が入ったテキストファイル。

Solaris スキーマ
CIM スキーマを Sun の Solaris 用に固有に拡張したスキーマ。Solaris スキーマには、一般的な Solaris オペレーティング環境に存在する管理対象オブジェクトを表現するクラスとインスタンスの定義が含まれる。

Simple Network Management Protocol (SNMP)
インターネット参照モデルのプロトコルの 1 つで、ネットワーク管理に使用される。

Unified Modeling Language (UML)
ソフトウェアシステムを説明するために使用される表記言語。ボックスと線を使用してオブジェクトと関係を表す。

Unicode
多くの既知の文字をコード化できる 16 ビットの文字セット。Unicode は、文字のコード化における標準として全世界的に使用されている。

UTF-8
Unicode 文字データの変換形式としての機能を果たす 8 ビットの変換形式。

Win32 スキーマ
CIM スキーマの Microsoft 固有の拡張機能。Win32 スキーマには、一般的な Win32 環境に存在する管理対象オブジェクトを表現するクラスとインスタンスの定義が含まれる。

インジケーション
インスタンスの作成、変更、または削除、インスタンスへのアクセス、プロパティの変更またはプロパティへのアクセスなどのアクションの結果として実行されるオペレーション。インジケーションは、指定された時間の経過からも起きる。インジケーションの結果、通常はイベントが発生する。

インスタンス
特定のクラス、または特定のイベントから作成された管理対象オブジェクトを表す。インスタンスには、実際のデータが含まれる。

インスタンスプロバイダ	システム固有のクラスおよびプロパティ固有のクラスのインスタンスをサポートするプロバイダ。インスタンスプロバイダは、データの検索、変更、削除、列挙をサポートできる。インスタンスプロバイダは、メソッドの呼び出しも行うことができる。 「プロパティプロバイダ」も参照。
インタフェースクラス	オブジェクトセットへのアクセスに使用されるクラス。インタフェースクラスには、列挙範囲を表す abstract クラスを使用できる。 「列挙」と「スコープ」も参照。
オーバーライド	派生クラス内のプロパティ、メソッド、または参照が、継承ツリー内の親クラス内または指定された親クラス内の類似した構成体を上書きすること。
オブジェクトパス	ネームスペース、クラス、およびインスタンスへのアクセスに使用される定形の文字列。システム上の各オブジェクトは、ローカルに、またはネットワーク上でそれ自体を識別する一意のパスを持つ。オブジェクトパスは、概念的には URL (Universal Resource Locator) に似ている。
拡張スキーマ	CIM スキーマの 3 つめの層であり、Solaris や UNIX などのプラットフォーム固有の CIM スキーマの拡張機能を含む。 「共通モデル」と「コアモデル」も参照。
仮想関数テーブル **(Virtual Function Table: VTBL)**	関数ポインタ (クラスの実装など) のテーブル。VTBL 内のポインタは、オブジェクトがサポートするインタフェースのメンバーを指す。
管理アプリケーション	管理環境内の 1 つ以上の管理対象オブジェクトから発生する情報を使用するアプリケーションまたはサービス。管理アプリケーションは、CIM Object Manager およびプロバイダから CIM Object Manager API を呼び出すことによってこの情報を検索する。
管理情報ベース	管理対象オブジェクトのデータベース。
管理対象オブジェクト	CIM クラスのインスタンスとして表現される、ハードウェアコンポーネントまたはソフトウェアコンポーネント。管理対象オブジェ

クトについての情報は、プロバイダおよび CIM Object Manager から提供される。

管理リソース　管理アプリケーションを使用して管理されるハードウェアコンポーネントまたはソフトウェアコンポーネント。例として、ハードディスク、CPU、オペレーティングシステムなどがあります。管理リソースは WBEM クラスの中に記述されます。

「管理対象オブジェクト」も参照。

関連クラス　2 つのクラス間、または 2 つのクラスのインスタンス間の関係を表現するクラス。関連クラスのプロパティには、2 つのクラスまたはインスタンスを指すポインタ (参照) が含まれる。WBEM クラスはすべて、1 つ以上の関連に含めることができる。

キー　クラスのインスタンスに一意の識別子を提供するために使用されるプロパティ。キープロパティは、キー修飾子によって示される。

キー修飾子　クラス内のプロパティ (そのクラスのキーの一部として機能する) に付加される必要がある修飾子。

共通モデル　CIM スキーマの 2 つめの層であり、ドメイン固有であるがプラットフォームには依存しない一連のクラスを含む。ここでのドメインは、管理関連のデータ (システム、ネットワーク、アプリケーションなど) を意味する。共通モデルは、コアモデルを元に作成されたもの。

「拡張スキーマ」も参照。

クラス　類似したプロパティを持ち、類似した目的を果たすオブジェクトの集まり。

継承　クラスとインスタンスが親クラス (スーパークラス) からどのように派生するかを表す関係。クラスは、新しいサブクラス (子クラスとも言う) を生成できる。サブクラスは、その親クラスのメソッドとプロパティをすべて含む。継承は、WBEM 環境内で WBEM のクラスが実際の管理対象オブジェクトのテンプレートとしての役割を果たすために必要な機能の 1 つである。

コアモデル	CIM スキーマの最初の層であり、最上位レベルのクラス、およびそれらのプロパティと関連を含む。コアモデルは、ドメインにもプラットフォームにも依存しない。 「共通モデル」と「拡張スキーマ」も参照。
サブクラス	スーパークラスから派生するクラス。サブクラスはそのスーパークラスのすべての機能を継承するが、新しい機能を追加することも、あるいは既存の機能を定義し直すこともできる。
サブスキーマ	スキーマの一部であり、特定の編成によって所有される。サブスキーマには、Win32 スキーマ、Solaris スキーマなどがある。
参照	参照修飾子によって示される特殊な文字列のプロパティであり、ほかのインスタンスを指すポインタであることを示す。
修飾子	クラス、インスタンス、プロパティ、メソッド、またはパラメータを表現する情報を含む。修飾子には、3 つのカテゴリがあり、CIM (Common Information Model) によって定義されるもの、WBEM (標準の修飾子) によって定義されるもの、および開発者によって定義されるものに分類される。標準の修飾子は、CIM Object Manager によって自動的に付加される。
修飾子フレーバ	CIM 修飾子の属性の 1 つで、修飾子の使用を制御する。修飾子フレーバは、修飾子が派生クラスと派生インスタンスに影響するかどうか、および派生クラスまたは派生インスタンスが修飾子の本来の値を上書きできるかどうかを指定する規則について記述する。
集約関係	1 つのエンティティがいくつかのほかのエンティティの集合から構成されている関係。
推移的な依存性	少なくても 3 つの属性を持つ関係、R (A, B, C)。この場合、A は B を決定し、B は C を決定するが、B は A を決定しない。
スーパークラス	サブクラスの継承元であるクラス。
スキーマ	特定の環境における管理対象オブジェクトについて説明するクラス定義の集まり。

スコープ　CIM 修飾子の属性の 1 つであり、どの CIM 要素がその修飾子を使用できるかを示す。スコープを定義できるのは Qualifier Type の宣言内だけであり、修飾子内では変更できない。

静的クラス　定義が永続的な WBEM クラス。定義は、明示的に削除されるまで CIM Object Manager Repository に格納される。CIM Object Manager は、プロバイダの支援を受けずに静的クラスの定義を提供できる。静的クラスは、静的インスタンス、動的インスタンスのどちらでもサポートできる。

静的インスタンス　CIM Object Manager Repository に永続的に格納されるインスタンス。

選択的な継承　下位クラスが上位クラスのプロパティを削除または上書きできること。

多重度　特定のエンティティの属性に適用できる値の数。

多相性　名前またはインタフェースを変更することなく、派生クラス内のメソッドとプロパティを変更できること。たとえば、サブクラスは、そのスーパークラスから継承されたメソッドまたはプロパティの実装を再定義できる。そのため、プロパティまたはメソッドは、スーパークラスがインタフェースクラスとして使用される場合でも再定義される。

したがって、LogicalDevice クラスは変数の状態を文字列として定義し、値として「オン」または「オフ」を返すことができる。LogicalDevice の Modem サブクラスは、「オン」、「オフ」、および「接続中」を返すことによって、状態を再定義 (オーバーライド) できる。LogicalDevice がすべて列挙される場合、その時点でモデムの役割を務める LogicalDevice はすべて、状態プロパティに対して「接続中」の値を返すことができる。

単一クラス　単一インスタンスだけをサポートする WBEM クラス。

動的インスタンス　必要時にプロバイダから提供されるインスタンスであり、CIM Object Manager Repositoty には格納されない。動的インスタンスは、静的クラス、動的クラスのどちらからも提供される。クラスのインスタンスを動的にサポートすることにより、プロバイダは最新のプロパティ値を提供できる。

動的クラス　必要に応じて実行時にプロバイダから定義が提供されるクラス。動的クラスはプロバイダ固有の管理対象オブジェクトを表現するのに使用され、CIM Object Manager Repository に永続的に格納されることはない。代わりに、動的クラスを提供するプロバイダがその場所についての情報を格納する。アプリケーションが動的クラスを要求する場合、CIM Object Manager はそのプロバイダを見つけて要求を送る。動的クラスがサポートするのは、動的インスタンスだけである。

ドメイン　プロパティまたはメソッドが属するクラス。たとえば、状態が Logical Device のプロパティの場合、Logical Device ドメインに属すると考えられる。

トリガー　クラスインスタンスの状態変化 (作成、削除、更新、アクセスなど)、およびプロパティの更新またはアクセスが発生すること。WBEM の実装には、トリガーを表現する明示的なオブジェクトは存在しない。トリガーは、システムの基本オブジェクトに対するオペレーション (クラス、インスタンス、およびネームスペースの作成、削除、変更) が起るか、または管理環境内のイベントによっては特定の指定をしなくても動作する。

名前付き要素　メタスキーマ内でオブジェクトとして表現できる実体。

ネームスペース　クラス、インスタンス、およびほかのネームスペースを含むことができる、ディレクトリに似た構造。

範囲　参照プロパティによって参照されるクラス。

必須プロパティ　値が必要なプロパティ。

標準スキーマ　システム、ネットワーク、またはアプリケーションの現在のオペレーション状態を表現する各種のクラスの編成と関連付けを行う、概念的な共通の枠組み。標準スキーマは、CIM (Common Information Model) では DMTF (Distributed Management Task Force) によって定義される。

フレーバ　「修飾子フレーバ」を参照。

プロパティ　クラスインスタンスの特性を示すために使用される値。プロパティ名は数字で開始することはできず、空白を含めることもできない。

	プロパティ値は、有効な MOF (Managed Object Format) のデータ型でなければならない。
プロパティプロバイダ	さまざまなソース (Solaris オペレーティング環境、Simple Network Management Protocol (SNMP) デバイスなど) のデータとイベント通知にアクセスするために管理対象オブジェクトと通信を行うプログラム。プロバイダは、統合と解釈を行うためにこの情報を CIM Object Manager に送る。
別名	MOF (Managed Object Format) ファイルの別の場所にあるオブジェクトに対するクラス宣言またはインスタンス宣言内のシンボリック参照。別名は、インスタンス名およびクラス名と同じ規則に従う。別名は、一般に比較的長いパスのショートカットとして使用される。
メソッド	あるクラスの動作について述べた関数。クラスにメソッドを組み込んでも、メソッドの実装を保証することにはならない。
メタスキーマ	CIM (Common Information Model) の公式の定義であり、このモデルの内容、使用法、および意味の説明に使用される用語を定義する。
メタモデル	管理対象オブジェクトを表現するエンティティと関係について説明する CIM コンポーネント。たとえば、クラス、インスタンス、関連など。
列挙	オブジェクトリストの取得を意味する Java 用語。Java は、オブジェクトリストを列挙するメソッドが含まれる `Enumeration` インタフェースを提供する。このリスト上の列挙される個々のオブジェクトは、要素と呼ばれる。

索引