HP Universal CMDB

Windows および Linux オペレーティング・システム用

ソフトウェア・バージョン : 9.0

開発者向け参考情報ガイド

ドキュメント・リリース日 : 2010 年 7 月（英語版）
ソフトウェア・リリース日 : 2010 年 7 月（英語版）

利用条件

保証

HP の製品およびサービスの保証は，かかる製品およびサービスに付属する明示的な保証の声明において定められている保証に限ります。本ドキュメントの内容は，追加の保証を構成するものではありません。HP は，本ドキュメントに技術的な間違いまたは編集上の間違い，あるいは欠落があった場合でも責任を負わないものとします。

本ドキュメントに含まれる情報は，事前の予告なく変更されることがあります。

制限事項

本コンピュータ・ソフトウェアは，機密性があります。これらを所有，使用，または複製するには，HP からの有効なライセンスが必要です。FAR 12.211 および 12.212 に従って，商用コンピュータ・ソフトウェア，コンピュータ・ソフトウェアのドキュメント，および商用アイテムの技術データは，HP の標準商用ライセンス条件に基づいて米国政府にライセンスされています。

著作権

商標

Adobe® および Acrobat® は，Adobe Systems Incorporated の商標です。

AMD および AMD の矢印記号は，Advanced Micro Devices, Inc. の商標です。

Google™ および Google マップ ™ は，Google Inc. の商標です。

Intel®，Itanium®，Pentium®，および Intel® Xeon® は，米国およびその他の国における Intel Corporation またはその子会社の商標または登録商標です。

Java™ は Sun Microsystems, Inc. の米国商標です。

Microsoft®，Windows®，Windows NT®，Windows® XP，および Windows Vista® は，Microsoft Corporation の米国登録商標です。

Oracle は，Oracle Corporation およびその関連企業の登録商標です。

UNIX® は The Open Group の登録商標です。

確認

- この製品には，Apache Software Foundation（http://www.apache.org/licenses（英語サイト））によって開発されたソフトウェアが含まれています。

- この製品には，OpenLDAP Foundation（http://www.openldap.org/foundation/（英語サイト））の OpenLDAP コードが含まれています。
- この製品には，Free Software Foundation, Inc.（http://www.fsf.org/）の GNU コードが含まれています。
- この製品には，Dennis M. Sosnoski の JiBX コードが含まれています。
- この製品には，ディストリビューションの一部で JiBX 全体で使用される，インディアナ大学 Extreme! Lab が含まれます。
- この製品には，Robert Futrell（http://sourceforge.net/projects/officelnfs（英語サイト））の Office Look and Feels ライセンスが含まれています。
- この製品には，Netaphor Software, Inc.（http://www.netaphor.com/home.asp（英語サイト））の JEP（Java Expression Parser）コードが含まれています。

文書の更新

本書のタイトル・ページには，次の識別情報が含まれています。

- ソフトウェアのバージョンを示すソフトウェア・バージョン番号
- ドキュメントが更新されるたびに更新されるドキュメント発行日
- 本バージョンのソフトウェアをリリースした日付を示す，ソフトウェア・リリース日付

最新のアップデートまたはドキュメントの最新版を使用していることを確認するには，次の URL にアクセスしてください：

http://h20230.www2.hp.com/selfsolve/manuals

このサイトでは，HP Passport に登録してサインインする必要があります。HP Passport ID の登録は，次の場所で行います。

http://h20229.www2.hp.com/passport-registration.html (英語サイト)

または，HP Passport のログイン・ページの［**New users - please register**］リンクをクリックしてください。

適切な製品サポート・サービスに登録すると，更新情報や最新情報も入手できます。詳細については HP の営業担当にお問い合わせください。

サポート

HP ソフトウェアのサポート Web サイトは，次の場所にあります。

http://support.openview.hp.com

この Web サイトでは，連絡先情報と，HP ソフトウェアが提供する製品，サービス，およびサポートについての詳細が掲載されています。

HP ソフトウェア・オンライン・ソフトウェア・サポートでは，お客様にセルフ・ソルブ機能を提供しています。ビジネス管理に必要な，インタラクティブなテクニカル・サポート・ツールに迅速かつ効率的にアクセスできます。有償サポートをご利用のお客様は，サポート・サイトの次の機能をご利用いただけます。

- 関心のある内容の技術情報の検索
- サポート・ケースおよび機能強化要求の提出および追跡
- ソフトウェア・パッチのダウンロード
- サポート契約の管理
- HP サポートの連絡先の表示
- 利用可能なサービスに関する情報の確認
- ほかのソフトウェア顧客との議論に参加
- ソフトウェアのトレーニングに関する調査と登録

ほとんどのサポート・エリアでは，HP Passport ユーザとして登録し，ログインする必要があります。また，多くの場合，サポート契約も必要です。HP Passport ID の登録は，次の場所で行います。

http://h20229.www2.hp.com/passport-registration.html (英語サイト)

アクセス・レベルの詳細に関しては次を参照してください。

http://h20230.www2.hp.com/new_access_levels.jsp

目次

ようこそ

このガイドでは，外部データ・リポジトリやほかの CMDB のデータを送受信できるアダプタを作成および管理する方法について説明します。

本章の内容

本書の構成

本書は，次の各章で構成されています。

第 I 部　ディスカバリ・アダプタおよびインテグレーション・アダプタの作成

アダプタの作成方法について説明します。

第 II 部　API の使用

API を使用して HP Universal CMDB から設定データを抽出する方法について説明します。

対象読者

本書は，次の HP Universal CMDB 利用者を対象としています。

- HP Universal CMDB 管理者
- HP Universal CMDB プラットフォーム管理者
- HP Universal CMDB アプリケーション管理者
- HP Universal CMDB データを管理する管理者

本書の読者は，エンタープライズ・システム管理に精通し，ITIL の概念を理解していること，そして HP Universal CMDB についての知識を備えている必要があります。

HP Universal CMDB オンライン・ドキュメント

HP Universal CMDB には，次のオンライン・ドキュメントが含まれています。

Readme : バージョンの制限事項および最終更新のリストが表示されます。HP Universal CMDB DVD のルート・ディレクトリから，**readme.html** をダブルクリックします。HP ソフトウェア・サポート の Web サイトからも，最新の Readme ファイルにアクセスできます。

新機能 : 新機能およびバージョンの重要項目のリストが表示されます。HP Universal CMDB で，［**ヘルプ**］＞［**新機能**］を選択します。

印刷用ドキュメント :［**ヘルプ**］＞［**UCMDB ヘルプ**］を選択します。次のガイドは，PDF 形式でのみ提供されています。

- HP Universal CMDB デプロイメント・ガイド : HP Universal CMDB の設定に必要なハードウェアおよびソフトウェア要件，HP Universal CMDB のインストールまたはアップグレード方法, システムのセキュリティを強化する方法，およびアプリケーションへのログイン方法について説明します。
- HP Universal CMDB データベース・ガイド : HP Universal CMDB で必要とされるデータベース（MS SQL Server または Oracle）の設定方法について説明します。
- HP Universal CMDB データ・フロー・コンテンツ・ガイド : ディスカバリを実行して，システムで実行されているアプリケーション，オペレーティング・システム，およびネットワーク・コンポーネントを検出する方法について説明します。統合によってほかのデータ・リポジトリにあるデータを検出する方法についても説明します。

HP Universal CMDB オンライン・ヘルプの内容は次のとおりです。

- **［モデリング］**：IT ユニバース・モデルのコンテンツを管理できます。
- **データ・フロー管理**：HP Universal CMDB をほかのデータ・リポジトリと統合する方法，およびネットワーク・コンポーネントを検出するように HP Universal CMDB を設定する方法について説明します。
- **UCMDB 管理**：HP Universal CMDB で作業する方法について説明します。
- **開発者向けリファレンス**：HP Universal CMDB について高度な知識を持つユーザを対象としています。アダプタを定義して使用する方法，および API を使用してデータにアクセスする方法について説明します。

オンライン・ヘルプは，HP Universal CMDB の個別のウィンドウからも利用できます。ウィンドウをクリックして［**ヘルプ**］ボタンをクリックします。

オンライン・ブックは Adobe Reader を使用して表示および印刷できます。Adobe Reader は Adobe Web サイトからダウンロードできます（www.adobe.com/jp/）。

トピックの種類

このガイドでは，各サブジェクト領域はトピックに分類されています。トピックには，サブジェクトの個別の情報モジュールが含まれています。トピックは通常，含まれる情報のタイプに従って分類されます。

ドキュメントは異なる状況で必要となるさまざまな情報タイプに分割されており，特定情報にアクセスしやすいように設計されています。

使用されている主なトピックの種類は，**概念**，**タスク**，および**参照情報**の 3 つです。これらのトピックの種類は，アイコンで視覚的に分かりやすく分類されています。

トピックの種類	説明	使用法
概念	背景，説明，または概念的な情報。	機能に関する一般情報について学習します。
タスク	**手順タスク**：アプリケーションを使用して目標を達成するための手順が，順を追って説明されています。一部のタスクの手順には，サンプル・データを使用した例が含まれます。 タスクの手順は，番号が付いている場合と付いていない場合があります。 ➤ **番号付きの手順**：各手順を連続した順序で行うことで実行するタスクです。 ➤ **番号が付いていない手順**：任意の順序で実行できる，自己充足型の操作のリストです。	➤ タスクの全体的なワークフローについて学習します。 ➤ 番号が付いているタスクのリストにある手順に従って，タスクを実行します。 ➤ 番号が付いていないタスクの手順を完了することで，独立した操作を実行します。
	使用例シナリオ・タスク：特定の状況でタスクを実行する方法の例です。	現実的なシナリオでタスクを実行する方法を学習します。

トピックの種類	説明	使用法
参照先	**一般的な参照情報**：参考資料に関する詳細なリストおよび説明です。	特定のコンテキストに関連する参照情報を検索します。
	ユーザ・インタフェース参照情報：特定のユーザ・インタフェースを詳細に説明した参照情報トピックです。通常，製品の［ヘルプ］メニューから［**このページのヘルプ**］を選択すると，ユーザ・インタフェースのトピックが開きます。	入力内容またはウィンドウ，ダイアログ・ボックス，ウィザードなど特定のユーザ・インタフェース要素の使用方法に関する個別の情報を検索します。
トラブルシューティングおよび制限事項	**トラブルシューティングおよび制限事項**：よく発生する問題および解決策について説明し，機能または製品領域の制限事項のリストを表示する参照情報トピックです。	機能を使用する前に，またはソフトウェアでユーザビリティに関する問題に遭遇した場合に，重要な問題に対する意識を高めます。

その他のオンライン・リソース

［**トラブルシューティングとナレッジ ベース**］を選択すると，HP ソフトウェアのサポート Web サイトのトラブルシューティング・ページが開き，セルフ・ソルブ技術情報を検索できます。［**ヘルプ**］＞［**トラブルシューティングとナレッジ ベース**］を選択します。この Web サイトの URL は http://support.openview.hp.com/troubleshooting.jsp です。

［**HP ソフトウェア サポート**］を選択すると，HP ソフトウェアのサポート Web サイトが開きます。このサイトでは，セルフ・ソルブ技術情報を参照できます。ユーザ・ディスカッション・フォーラムへの参加と検索，サポート要求の送信，パッチやアップデートされたドキュメントのダウンロードなども行うことができます。［**ヘルプ**］＞［**HP ソフトウェア サポート**］を選択します。この Web サイトの URL は http://support.openview.hp.com です。

ほとんどのサポート・エリアでは，HP Passport ユーザとして登録し，ログインする必要があります。また，多くの場合，サポート契約も必要です。

アクセス・レベルの詳細に関しては次を参照してください。

http://h20230.www2.hp.com/new_access_levels.jsp

HP Passport ユーザ ID の登録は，次の URL にアクセスしてください。

http://h20229.www2.hp.com/passport-registration.html (英語サイト)

［**HP ソフトウェアの Web サイト**］を選択すると，HP ソフトウェアの Web サイトが開きます。このサイトには，HP ソフトウェア製品の最新情報が表示されます。新規ソフトウェア・リリース，セミナーおよび製品発表会，カスタマ・サポートなどの情報が含まれます。［**ヘルプ**］＞［**HP ソフトウェアの Web サイト**］を選択します。この Web サイトの URL は http://welcome.hp.com/country/jp/ja/prodserv/software.html です。

ドキュメントの更新

HP ソフトウェアの製品ドキュメントは，新しい情報で絶えず更新されています。

最新のアップデートまたはドキュメントの最新版を使用していることを確認するには，HP ソフトウェア製品マニュアルの Web サイト（http://h20230.www2.hp.com/selfsolve/manuals）にアクセスしてください。

第 I 部

ディスカバリ・アダプタおよびインテグレーション・アダプタの作成

1

アダプタ開発と記述

本章の内容

概念

タスク

概念

アダプタ開発と記述の概要

新しいアダプタの開発を実際に計画し始める前に，この開発に付き物のプロセスおよびインタラクションを理解することが重要です。

次の項は，ディスカバリ開発プロジェクトの管理と実行を成功させるために必要な知識と手順を理解するのに役立ちます。

本章の概要は次のとおりです。

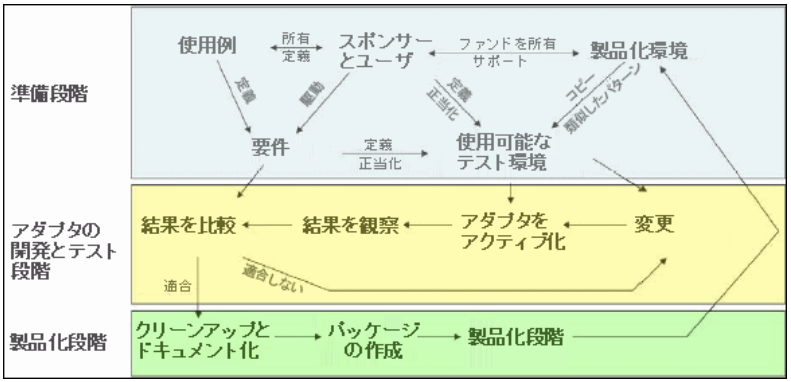
- HP Universal CMDB に関する実用的な知識と，システムの要素に関する基本的な知識があることを前提としています。本章は，ユーザの学習を支援することが目的であり，完全なガイドではありません。
- HP Universal CMDB の新しいディスカバリ・コンテンツの計画，調査，および実装段階を対象としています。また，考慮する必要があるガイドラインと注意事項も示します。
- データ・フロー管理フレームワークの主な API について説明します。使用可能な API の完全なドキュメントについては，『HP Universal CMDB データ・フロー管理 API 参考情報』を参照してください（ほかに非公式の API も存在しますが，用意済みのアダプタに使用されている場合でも，変更されることがあります）。

コンテンツの作成

本項の内容

アダプタの開発サイクル

次の図は，アダプタ記述のフローチャートを示しています。ほとんどの時間は，開発とテストの反復ループである中央の部分に費やされます。



アダプタ開発の各段階は，直前の段階に基づいています。

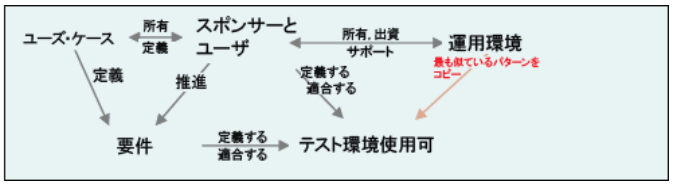
アダプタの記述内容と動作に満足したら，アダプタをパッケージ化できます。UCMDB パッケージ・マネージャを使用するか，または手動でコンポーネントをエクスポートして，パッケージの *.zip ファイルを作成します。ベスト・プラクティスとしては，実運用環境にリリースする前に別の UCMDB システム上でこのパッケージのデプロイとテストを行うことにより，すべてのコンポーネントが組み込まれ，正常にパッケージ化されたことを確認します。パッケージ化の詳細については，『HP UCMDB 管理ガイド』の「パッケージ・マネージャ」を参照してください。

次の各項では，最も重要な手順とベスト・プラクティスを示しながら，各段階についてさらに詳しく説明します。

- 調査と準備の段階
- アダプタの開発とテスト
- アダプタのパッケージ化と製品化

調査と準備の段階

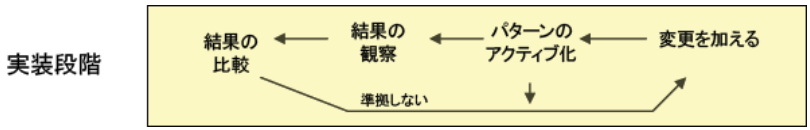
準備の段階



調査と準備の段階には原動力となるビジネス・ニーズとユース・ケースが含まれます。また，アダプタの開発とテストに必要な施設の確保もこの段階で行います。

1. 既存のアダプタを変更する場合は，最初の技術的な措置として，そのアダプタのバックアップを作成し，元の状態に戻れるようにしておきます。新しいアダプタを作成する場合は，最も似ているアダプタをコピーし，それに適切な名前を付けて保存します。詳細については,『HP Universal CMDB データ・フロー管理ガイド』の「[リソース] 表示枠」を参照してください。
2. アダプタによってデータを収集する方法を調べます。
   - 外部のツールやプロトコルを使ってデータを収集する
   - アダプタがデータに基づいて CI を作成する方法を開発する
   - 同じようなアダプタがどのように記述されているかをすでに知っている
3. 次の要因に基づいて，最も似ているアダプタを決定します。
   - 同じ CI が作成されている
   - 同じプロトコル（SNMP）が使用されている
   - ターゲットの種類が同じである（OS のタイプやバージョンなど）
4. パッケージ全体をコピーします。

5 作業領域に展開し，アダプタ（XML）ファイルと Jython（.py）ファイルの名前を変更します。



アダプタの開発とテスト

アダプタの開発とテストの**段階**は，頻繁に反復されるプロセスです。アダプタが具体化し始めたら，最終的なユース・ケースに対するテストを開始し，変更を行い，再度テストします。このプロセスを，アダプタが要件に適合するまで繰り返します。

開始とコピーの準備

- アダプタの XML 部分（1 行目の名前（ID），作成された CI タイプ，および呼び出す Jython スクリプト名）を変更します。
- 実行すると元のアダプタと同じ結果となるコピーを入手します。
- コードの大部分（特に，重要な結果を生成するコード）をコメント・アウトします。

開発とテスト

- ほかのサンプル・コードを使って変更部分を開発します。
- アダプタを実行してテストします。
- 専用のビューを使って複雑な結果を検証し，検索を使って簡単な結果を検証します。

アダプタのパッケージ化と製品化

アダプタのパッケージ化と製品化の段階は，開発の最終段階になります。ベスト・プラクティスとしては，パッケージ化に進む前に，デバッグの残存部分，ドキュメント，およびコメントを削除したり，セキュリティ上の考慮事項を確認したりするための最終パスを実施します。少なくとも Readme ドキュメントを必ず作成して，アダプタの内部構造を説明する必要があります。かなり限定された内容でも，今後誰かが（おそらくユーザ自身も）このアダプタを見る必要があるときに，大いに役立つはずです。

クリーンアップとドキュメント化

- デバッグ処理を削除します。
- すべての関数にコメントを付与し，メイン・セクションに開始コメントを追加します。
- ユーザがテストするためのサンプル TQL とビューを作成します。

パッケージの作成

- パッケージ・マネージャを使って，アダプタや TQL などをエクスポートします。詳細については，『HP UCMDB 管理ガイド』の「パッケージ・マネージャ」を参照してください。
- ほかのパッケージに対する依存関係（ほかのパッケージで作成された CI がアダプタへの入力 CI になっているかどうかなど）を確認します。
- パッケージ・マネージャを使ってパッケージの zip ファイルを作成します。詳細については，『HP UCMDB 管理ガイド』の「パッケージ・マネージャ」を参照してください。
- 新しいコンテンツの一部を削除して再度デプロイするか，またはほかのテスト・システムにデプロイすることによって，デプロイメントをテストします。

データ・フロー管理とインテグレーション

DFM のアダプタは，ほかの製品と統合できます。次の定義を考慮してください。

- DFM では，多くのターゲットから特定のコンテンツが収集されます。
- 統合では，1 つのシステムから複数のタイプのコンテンツが収集されます。

前述の定義では，収集の方法は区別されません。DFM でも同様です。新しいアダプタを開発するプロセスは，新しいインテグレーションを開発するのと同じプロセスです。同じ調査を行い，新しいアダプタと既存のアダプタに関して同じ選択を行い，同じ方法でアダプタを記述します。次のように，変更点はごくわずかです。

- 最終的なアダプタのスケジュール。インテグレーション・アダプタは，ディスカバリによって頻繁に実行される可能性がありますが，その頻度はユース・ケースによって異なります。
- 入力 CI:
  - インテグレーション：CI 以外のトリガによって，入力なしで実行されます。アダプタ・パラメータでファイル名やソースが渡されます。
  - ディスカバリ：通常の CMDB CI が入力に使用されます。

インテグレーション・プロジェクトでは，ほとんどの場合，既存のアダプタを再利用する必要があります。統合の方向（HP Universal CMDB からほかの製品へ，またはほかの製品から HP Universal CMDB へ）が開発の方法に影響することがあります。ユーザが実績のある方法を使って特定の用途向けにコピーできるフィールド・パッケージがあります。

HP Universal CMDB からほかのプロジェクトへの統合は，次のようにして行います。

- エクスポートする CI と関係を生成する TQL を作成します。
- 汎用のラッパー・アダプタを使って，TQL を実行してその結果を XML ファイルに書き込み，それを外部製品で読み取ります。

注：フィールド・パッケージの例については，HP ソフトウェア・サポート にお問い合わせください。

ほかの製品を HP Universal CMDB に統合する場合は，ほかの製品のデータがどのように公開されるかによって，インテグレーション・アダプタの機能が異なります。

統合のタイプ	再利用される参考例
製品のデータベースに直接アクセスする	HP ED
エクスポートによって生成された csv または xml ファイルを読み取る	HP ServiceCenter
製品の API にアクセスする	BMC Atrium/Remedy

ビジネス価値とディスカバリ開発の関連付け

新しいディスカバリ・コンテンツ開発のユース・ケースでは，ビジネス・ケースおよびビジネス計画によって，ビジネス価値を生み出す必要があります。つまり，システム・コンポーネントを CI にマップし，それらを CMDB に追加することの目的は，ビジネス価値を提供することです。

開発されたコンテンツがアプリケーション・マッピングに使用されるとは限りませんが，これは多くのユース・ケースで一般的な中間段階です。コンテンツの最終的な用途に関係なく，計画ではその手法に関する次の疑問に答える必要があります。

- 誰がコンシューマか。コンシューマは CI（および CI 間の関係）によって提供された情報に対してどのように行動するか。CI と関係をどのようなビジネス・コンテキストに表示すべきか。これらの CI のコンシューマは，人間か，製品か，またはその両方か。
- CI と関係の完全な組み合わせが CMDB に存在する場合，それらを使ったビジネス価値の生成をどのように計画するか。
- 完全なマッピングはどのようなものか。
  - 各 CI 間の関係を最もわかりやすく言葉で説明するとどうなるか。
  - 含める必要がある最も重要な CI のタイプは何か。
  - マップの最終的な用途とエンド・ユーザは何か。
- 完全なレポート・レイアウトはどのようなものか。

ビジネス上の判断を確立したら，次の手順として，ビジネス価値をドキュメントで具体化します。つまり，描画ツールを使って完全なマップを作成し，ユース・ケースでの必要性に応じて，CI 間の影響と依存関係，レポート，変化の追跡方法，重要な変化，監視，コンプライアンス，およびその他のビジネス価値を理解します。

この図（またはモデル）を「**青写真**」と呼びます。

たとえば，特定の構成ファイルが変更されたことを検出することがアプリケーションにとって重要な場合は，そのファイルをマップして，作成されたマップ内で適切な（そのファイルに関連する）CI にリンクする必要があります。

開発されたコンテンツのエンド・ユーザである当該分野の SME（各分野のエキスパート）とともに作業します。このエキスパートは，ビジネス価値を提供するために CMDB に存在する必要がある重要なエンティティ（属性と関係を持つ CI）を指摘する必要があります。

1 つの方法として，アプリケーションの所有者（この場合は SME も含む）にアンケートを行うことが考えられます。所有者は，前述の目的や青写真を明確に示すことができるはずです。所有者は，少なくともアプリケーションの現在のアーキテクチャを示す必要があります。

重要なデータのみをマップし，不要なデータをマップしないようにする必要があります。アダプタは，後でいつでも拡張できます。目的は，適切に動作し，価値を提供する限定的なディスカバリを設定することであるべきです。大量のデータをマップすると，印象の強いマップができますが，混乱を招きやすく，開発に多くの時間がかかる可能性があります。

モデルとビジネス価値が明確になったら，次の段階に進みます。この段階は，その後の段階でより具体的な情報が得られるたびに実行し直すことができます。

インテグレーション要件の調査

この段階の前提条件は，DFM で検出する必要がある CI および関係の**青写真**があることです。この青写真には，検出する属性を含める必要があります。詳細については，20 ページの「アダプタ開発と記述の概要」を参照してください。

本項の内容

既存のアダプタの変更

用意済みのアダプタやフィールド・アダプタが存在する場合は，既存のアダプタを変更します。ただし，次の点に注意してください。

- 既存のパターンでは，必要とされる特定の属性は検出されません。
- 特定のタイプのターゲット（OS）が検出されないか，不正に検出されます。
- 特定の関係が検出されないか，作成されません。

既存のアダプタがジョブの（全部でなく）一部を行う場合は，最初の作業として，複数の既存のアダプタを評価し，いずれかのアダプタが必要なほとんどの処理を行うかどうかを確認します。

既存のフィールド・アダプタが利用可能かどうかも評価する必要があります。フィールド・アダプタは，利用可能であるが用意済みではないディスカバリ・アダプタです。フィールド・アダプタの最新リストを受け取るには，HP ソフトウェア・サポートに連絡してください。

新しいアダプタの記述

次のような場合は，新しいアダプタを開発する必要があります。

- CMDB に手動で情報を挿入するよりアダプタを記述した方が速い場合（一般に，CI と関係の数が 50 ～ 100 に及ぶ場合）や，記述したアダプタを再利用する場合。
- 必要性の高さがその労力に見合う場合。
- 用意済みのアダプタやフィールド・アダプタが利用できない場合。
- 結果を再利用できる場合。
- ターゲット環境やそのデータが利用可能な状態である場合（ユーザが確認できないものは検出できません）。

モデルの調査

- UCMDB クラス・モデル（CI タイプ・マネージャ）を参照し，**青写真**から既存の CIT に対してエンティティと関係を照合します。バージョン・アップグレード時の混乱を避けるため，現在のモデルに従うことを強くお勧めします。モデルを拡張する必要がある場合は，用意済みの CIT がアップグレードによって上書きされる可能性があるため，新しい CIT を作成する必要があります。
- 一部のエンティティ，関係，または属性が現在のモデルに見つからない場合は，それらを作成する必要があります。これらの CIT を HP Universal CMDB の各インストールにデプロイできることが必要なため，これらの CIT を含むパッケージを作成することをお勧めします（このパッケージには，後で，すべてのディスカバリ，ビュー，およびこのパッケージに関連するその他の作成物も保持されます）。

テクノロジの調査

CMDB に必要な CI が保持されていることを確認したら，次の段階として，このデータを関連するシステムから取得する方法を決定します。

データを取得するには，通常，何らかのプロトコルを使って，アプリケーションの管理部分，アプリケーションの実際のデータ，またはアプリケーションに関連する構成ファイルやデータベースにアクセスする必要があります。システムに関する情報を提供できるすべてのデータ・ソースでも役に立ちます。テクノロジの調査には，問題のシステムに関する深い知識と，場合によっては創造性も必要です。

自社開発アプリケーションの場合は，アプリケーションの所有者にアンケート用紙を提供すると，参考になることがあります。所有者は，この用紙に，青写真とビジネス価値に関して必要な情報を提供できるアプリケーション内のすべての領域を記入する必要があります。これらの情報には，管理データベース，構成ファイル，ログ・ファイル，管理インタフェース，管理プログラム，Web サービス，および送信されたメッセージやイベントなどが含まれます（ただし，これらに限定されません）。

市販製品の場合は，製品のドキュメント，フォーラム，またはサポートに焦点を合わせる必要があります。管理ガイド，プラグインおよび統合ガイド，運用ガイドなどを調べます。データがまだ管理インタフェースにない場合は，アプリケーションの構成ファイル，レジストリ・エントリ，ログ・ファイル，NT イベント・ログ，およびアプリケーションの正しい運用を制御する作成物について参照します。

データへのアクセス方法の選択に関するガイドライン

関連性 : 最も多くのデータを提供するソースまたはソースの組み合わせを選択します。ほとんどのデータが 1 つのソースから提供され，残りの情報が分散していてアクセスしにくい場合は，残りの情報の価値を取得する労力やリスクとの比較で評価します。価値やコストが投入する労力に見合わない場合は，青写真を縮小することも検討します。

再利用 : 特定の接続プロトコルのサポートが HP Universal CMDB にすでに含まれている場合は，それを使用するのが良い方法です。使用する場合は，DFM フレームワークからその接続用の既製のクライアントと設定が提供されます。使用しない場合は，インフラストラクチャの開発に投資する必要が生じる可能性があります。現在サポートされている HP Universal CMDB の接続プロトコルは，［**ディスカバリ**］＞［**ディスカバリ プローブ設定**］＞［**ドメインとプローブ**］表示枠に表示されます。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［ドメインとプローブ］表示枠」を参照してください。

新しいプロトコルを追加するには，モデルに新しい CI を追加します。詳細については，HP ソフトウェア・サポート までお問い合わせください。

注 : Windows レジストリ・データにアクセスするには，WMI と NTCmd のいずれかを使用します。

セキュリティ：情報へのアクセスには，通常，資格情報（ユーザ名とパスワード）が必要です。資格情報は CMDB に入力され，製品全体でセキュリティ保護されます。可能な場合，およびセキュリティの追加が設定済みのほかの原則と競合しない場合は，アクセス・ニーズに対応する最も機密性の低い資格情報またはプロトコルを選択します。たとえば，JMX（標準の管理インタフェース。限定的）と Telnet のどちらでも情報を取得できる場合は，JMX を使用することをお勧めします。これは，JMX では本質的に限定されたアクセスが提供され，基盤となるプラットフォームへのアクセスが提供されないためです。

使いやすさ：一部の管理インタフェースには，より高度な機能が含まれています。たとえば，解析のために情報ツリーをたどったり，正規表現を作成したりするより，クエリ（SQL や WMI）を発行する方が簡単な場合があります。

使用する開発者：最終的にアダプタを開発するユーザは，特定のテクノロジを好む傾向があります。2 つのテクノロジでほぼ同じ情報が提供され，ほかの要因のコストが同じである場合は，この点を考慮することもできます。

サマリ

この段階の成果物は，アクセス方法と各方法から抽出できる関連情報について記載したドキュメントです。このドキュメントには，各ソースから関連する各青写真データへのマッピングも含める必要があります。

前述の説明に従って，各アクセス方法を採点する必要があります。最終的には，どのソースを検出し，どの情報を各ソースから青写真モデルに抽出するかについての計画ができあがります（青写真モデルは，このときまでに対応する UCMDB モデルにマップされている必要があります）。

インテグレーション・コンテンツの開発

新しいインテグレーションを作成する前に，インテグレーション要件を理解しておく必要があります。

➤ インテグレーションでデータを CMDB にコピーする必要がある。データを履歴で追跡する必要がある。ソースを信頼できない。

ポピュレーションが必要です。

- インテグレーションでビューおよび TQL クエリのデータをオンザフライでフェデレートする必要がある。データ変更の正確さが重要である。CMDB にコピーするにはデータ量が大きすぎるが，要求される通常のデータ量は小さい。

  連携が必要です。

- インテグレーションでデータをリモート・データ・ソースにプッシュする必要がある。

  データ・プッシュが必要です。

注 : 同じインテグレーションに連携とポピュレーションのフローを設定することで，最大限の柔軟性を得ることができます。

各種タイプのインテグレーションの詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「Integration Studio」を参照してください。

インテグレーション・アダプタの作成には，次の異なる 4 つのオプションを使用できます。

- Jython アダプタ
  - 従来のディスカバリ・パターン。
  - Jython で記述。
  - ポピュレーションに使用。

  詳細については，63 ページの「Jython アダプタの開発」を参照してください。

- Java アダプタ
  - Federation SDK Framework のアダプタ・インタフェースのいずれかを実装するアダプタ。
  - 1 つ以上の連携，ポピュレーション，またはデータ・プッシュに使用可能（必要な実装によって異なる）。
  - Java でゼロから記述。任意のソースまたはターゲットに接続するコードを記述できる。
  - それぞれが単一のデータ・ソースまたはターゲットに接続するジョブに最適。

  詳細については，205 ページの「Java アダプタの開発」を参照してください。

- 汎用 DB アダプタ
  - Java アダプタをベースにした抽象アダプタで, Federation SDK Framework を使用。
  - 外部データ・リポジトリに接続するアダプタを作成可能。
  - 連携とポピュレーションの両方をサポート（変更をサポートするために実装される Java プラグインを使用）。
  - 主に XML およびプロパティ構成ファイルに基づいているため比較的定義が容易。
  - 主要な設定は UCMDB クラスとデータベース・カラムをマップする **orm.xml** ファイルに基づいている。
  - それぞれが単一のデータ・ソースに接続するジョブに最適。

  詳細については，119 ページの「汎用データベース・アダプタの開発」を参照してください。

- 汎用プッシュ・アダプタ
  - Java アダプタをベースにした抽象アダプタ（Federation SDK Framework）および Jython アダプタ。
  - リモート・ターゲットにデータをプッシュするアダプタを作成可能。
  - UCMDB クラスと XML のマッピング，およびターゲットにデータをプッシュする Jython スクリプトを定義するだけであり，比較的定義が容易。
  - それぞれが単一のデータ・ターゲットに接続するジョブに最適。
  - データ・プッシュに使用。

  詳細については，249 ページの「プッシュ・アダプタの開発」を参照してください。

次の表は，各アダプタの機能を示します。

フロー / アダプタ	Jython アダプタ	Java アダプタ	GDB アダプタ	プッシュ・アダプタ
ポピュレーション	X	X	X	
連携		X	X	
データ・プッシュ		X		X

ディスカバリ・コンテンツの開発

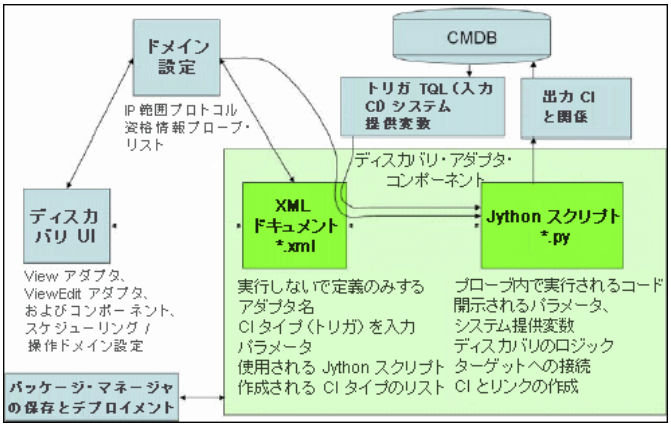
本項の内容

➤ 34 ページの「ディスカバリ・アダプタと関連コンポーネント」

➤ 35 ページの「アダプタの分割」

ディスカバリ・アダプタと関連コンポーネント

次の図は，アダプタのコンポーネントと，各コンポーネントがディスカバリの実行時にやり取りするコンポーネントを示します。緑色のコンポーネントは実際のアダプタを示し，青色のコンポーネントはアダプタとやり取りするコンポーネントを示します。



アダプタの最小概念は，XML ドキュメントと Jython スクリプトという 2 つのファイルです。実行時には，ディスカバリ・フレームワーク（入力 CI，資格情報，およびユーザ定義ライブラリを含む）がアダプタに公開されます。これら 2 つのディスカバリ・アダプタ・コンポーネントは，データ・フロー管理によって管理されます。また，これらは操作によって CMDB 自体に格納されます。外部パッケージは残りますが，操作では参照されません。新しいディスカバリおよびインテグレーション・コンテンツの機能は，パッケージ・マネージャを使って保存できます。

アダプタへの入力 CI は TQL によって提供され，システム定義変数でアダプタ・スクリプトに公開されます。アダプタ・パラメータも宛先データとして指定されるため，アダプタの特定の関数に従ってアダプタの操作を設定できます。

新しいアダプタは，DFM アプリケーションを使って作成およびテストします。アダプタの記述中は，［ディスカバリ コントロール パネル］，［アダプタ管理］，および［Data Flow Probe 設定］ページを使用します。

アダプタは，パッケージとして格納および転送されます。パッケージ・マネージャ・アプリケーションと JMX コンソールを使って，新規作成されたアダプタからパッケージを作成し，新しいシステムにアダプタをデプロイします。

アダプタの分割

技術的には, ディスカバリ全体を 1 つのアダプタで定義することもできます。ただし，適切な設計では，複雑なシステムをより簡単な，管理しやすいコンポーネントに分割することが求められます。

アダプタ・プロセスを分割するためのガイドラインとベスト・プラクティスを次に示します。

- ディスカバリは，複数の段階に分けて行う必要があります。各段階は，システムのある領域や階層をマップするアダプタによって表されます。アダプタは，検出された直前の段階または階層を利用して，システムの検出を続行します。たとえば，アダプタ A は，アプリケーション・サーバ TQL の結果によって起動され，アプリケーション・サーバ層をマップします。このマッピングの中で, JDBC 接続コンポーネントがマップされます。アダプタ B は, JDBC 接続コンポーネントをトリガ TQL として登録し，アダプタ A の結果を使って（たとえば，JDBC URL 属性を介して）データベース層にアクセスし，データベース層をマップします。
- **2 段階接続の枠組み** : ほとんどのシステムでは, システムのデータにアクセスするために資格情報が必要です。つまり，これらのシステムに対してユーザ / パスワードの組み合わせを試行する必要があります。DFM 管理者は，安全な方法でシステムに資格情報を提供します。また，優先順位が設定された複数のログイン資格情報を提供できます。これは，**プロトコル辞書**と呼ばれます。システムに（何らかの理由で）アクセスできない場合は，それ以上ディスカバリを実行しても無意味です。接続に成功した場合は，その後のディスカバリ・アクセスのために，どの資格情報セットを使って成功したかを示す何らかの方法が必要です。

前述の 2 段階に合わせて，アダプタも次の 2 つに分割されます。

- **接続アダプタ**：これは，最初のトリガを受け入れ，そのトリガにリモート・エージェントが存在するかどうかを調べるアダプタです。そのために，このエージェントのタイプと一致するプロトコル辞書内のすべてのエントリを試行します。成功すると，このアダプタはその結果としてリモート・エージェント CI（SNMP や WMI など）を提供し，その後の接続のためにプロトコル辞書内の正しいエントリも示します。このエージェント CI は，コンテンツ・アダプタのトリガの一部になります。
- **コンテンツ・アダプタ**：このアダプタの前提条件は，直前のアダプタで接続に成功することです（TQL によって指定される前提条件）。このタイプのアダプタでは，リモート・エージェント CI から正確な資格情報を取得する方法があり，取得した資格情報を使って検出されたシステムにログインできるため，プロトコル辞書のすべてのエントリを調べる必要はありません。

- スケジュール設定に関するさまざまな考慮事項が，ディスカバリの意思決定に影響を与えることもあります。たとえば，システムのクエリが営業時間外にしか行われない場合は，アダプタを別のシステムを検出する同じアダプタと結合することが妥当であっても，スケジュールが異なるために 2 つのアダプタを作成する必要があります。
- 異なる管理インタフェースやテクノロジを使って同じシステムを検出するディスカバリは，個別のアダプタに分ける必要があります。これによって，各システムまたは組織に適したアクセス方法をアクティブにすることができます。たとえば，一部の組織では，WMI でマシンにアクセスできますが，SNMP エージェントがマシンにインストールされていません。

タスク

ディスカバリ・アダプタの実装

DFM タスクの目的は，リモート（またはローカル）システムにアクセスし，抽出されたデータを CI としてモデル化し，それらの CI を CMDB に保存することです。このタスクは次の手順で構成されます。

1 DFM アダプタ：

アダプタに含めるスクリプトを選択することにより，コンテキスト，パラメータ，および結果タイプを保持するアダプタ・ファイルを設定します。詳細については，次の項を参照してください。

2 ディスカバリ・ジョブ：

スケジュール情報とトリガ TQL を含むジョブを設定します。詳細については，50 ページの「手順 2：アダプタへのジョブの割り当て」を参照してください。

3 ディスカバリ・コード：

アダプタ・ファイルに含まれ，DFM フレームワークを参照する Jython コードまたは Java コードを編集できます。詳細については，52 ページの「手順 3：Jython コードの作成」を参照してください。

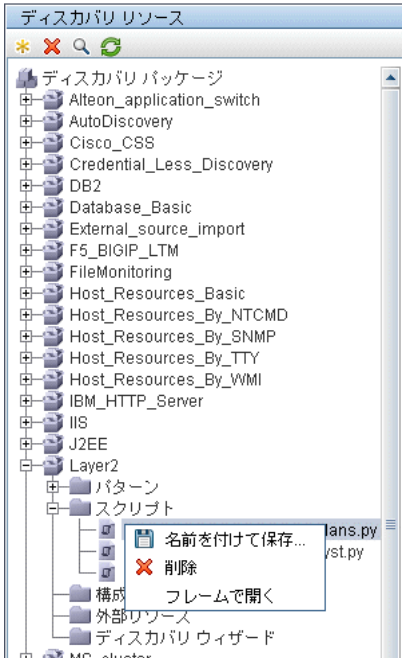
新しいアダプタを記述するには，前述の各コンポーネントを作成します。作成した各コンポーネントは，前の手順のコンポーネントに自動的にバインドされます。たとえば，ジョブを作成して関連するアダプタを選択すると，そのアダプタはジョブにバインドされます。

アダプタ・コード

リモート・システムに接続し，そのデータを照会し，それを CMDB データとしてマップする処理の実際の実装は，Jython コードによって実行されます。コードには，たとえば，データベースに接続してそこからデータを抽出するためのロジックが含まれています。この場合，コードは JDBC URL，ユーザ名，パスワード，ポートなどを受け取ることを期待しています。これらのパラメータは，TQL クエリに応答するデータベースの各インスタンスに固有のものです。これらの変数はアダプタ（トリガ CI データ）で定義され，ジョブを実行すると，これらの詳細がコードに渡されて実行に使用されます。

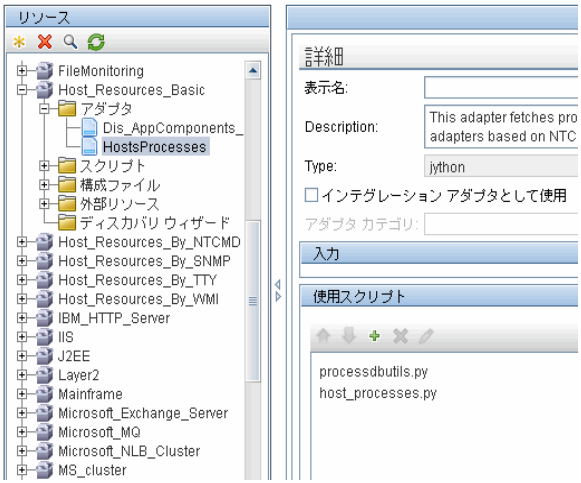
アダプタは，Java クラス名または Jython スクリプト名によってこのコードを参照できます。このセクションでは，Jython スクリプトとして DFM コードを記述する方法について説明します。

アダプタには，ディスカバリの実行時に使用されるスクリプトのリストが含まれています。新しいアダプタを作成するときは，通常，新しいスクリプトを作成し，それをアダプタに割り当てます。新しいスクリプトには基本的なテンプレートが含まれていますが，ほかのいずれかのスクリプトを右クリックして［**名前を付けて保存**］を選択すれば，それをテンプレートとして使用できます。



新しい Jython スクリプトを記述する方法の詳細については，52 ページの「手順 3 : Jython コードの作成」を参照してください。スクリプトを追加するには，［リソース］表示枠を使用します。

スクリプトのリストは，アダプタに定義された順序で 1 つずつ実行されます。



注 : スクリプトは, 別のスクリプトでライブラリとしてのみ使用される場合でも指定する必要があります。その場合は，ライブラリ・スクリプトを使用するスクリプトより先に，ライブラリ・スクリプトを定義します。この例では，processdbutils.py スクリプトは最後の host_processes.py スクリプトが使用するライブラリです。ライブラリは，DiscoveryMain() 関数がないことによって通常の実行可能なスクリプトと区別されます。

手順 1 : アダプタの作成

アダプタは，関数の定義とみなすことができます。この関数は，入力定義を定義し，入力に対してロジックを実行し，出力を定義して，結果を提供します。

各アダプタには入力と出力が指定されます。入力も出力も，そのアダプタで明示的に定義されたトリガ CI です。アダプタは，入力トリガ CI からデータを抽出し，そのデータをコードにパラメータとして渡します（関連 CI からのデータもコードに渡されることがあります。詳細については,『HP Universal CMDB データ・フロー管理ガイド』の「[関連 CI］ウィンドウ」を参照してください。アダプタ・コードは，そのコードに渡される特定の入力トリガ CI のパラメータを除いて，汎用的なものです。

入力コンポーネントの詳細については,『HP Universal CMDB データ・フロー管理ガイド』の「トリガ CI とトリガ・クエリ」を参照してください。

本項の内容

- 41 ページの「アダプタ入力（トリガ CIT と入力クエリ）の定義」
- 47 ページの「アダプタ出力の定義」
- 49 ページの「アダプタ・パラメータの上書き」

アダプタ入力（トリガ CIT と入力クエリ）の定義

特定の CI をアダプタ入力として定義するには，次のようにトリガ CIT と入力クエリ・コンポーネントを使用します。

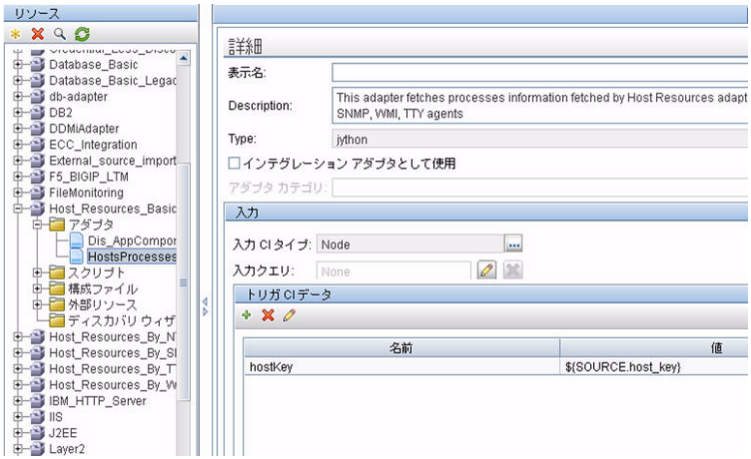
- トリガ CIT は，アダプタの入力としてどの CIT を使用するかを定義します。たとえば，IP を検出するアダプタでは，入力 CIT は Network です。
- 入力クエリは，CMDB に対するクエリを定義する通常の編集可能なクエリです。入力クエリは，CIT に対する追加の制約を定義します（たとえば，hostID 属性または application_ip 属性がタスクに必要な場合など）。また，アダプタに必要な場合は，追加の CI データを定義できます。

  トリガ CI に関連する CI からの追加情報がアダプタに必要な場合は，入力 TQL にノードを追加できます。詳細については，『モデリング・ガイド』の 43 ページの「入力クエリ定義の例」および「TQL クエリへのクエリ・ノードと関係の追加」を参照してください。
- トリガ CI のデータには，トリガ CI に関する必要なすべての情報と，（定義された場合は）入力 TQL 内のほかのノードからの情報が含まれています。DFM では，変数を使用して CI からデータを取得します。Probe にタスクがダウンロードされると，トリガ CI のデータ変数は実際の CI インスタンスの属性に存在する実際の値に置き換えられます。

トリガ CIT 定義の例：

この例では，アダプタ内で IP CI を許可することがトリガ CIT によって定義されています。

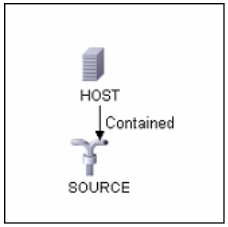
1 ［**データ フロー管理**］＞［**アダプタ管理**］にアクセスします。HostProcesses アダプタを選択します（［**パッケージ**］＞［**Host_Resources_Basic**］＞［**アダプタ**］＞［**HostsProcesses**］）。
2 ［入力 CI タイプ］ボックスを見つけます。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「トリガ CI データ」を参照してください。
3 ボタンをクリックすると，［検出クラスを選択］ダイアログ・ボックスが開きます。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［検出クラスを選択］ダイアログ・ボックス」を参照してください。
4 CIT を選択します。
この例では，アダプタ内で CI（Host）が許可されています。



入力クエリ定義の例

この例では，（前の例でトリガ CIT として設定された）IP CI を Host CI に接続する必要があることが入力 TQL クエリによって定義されています。

1 ［**データ フロー管理**］＞［**アダプタ管理**］にアクセスします。［入力 TQL］ボックスを見つけます。［**編集**］ボタンをクリックして［TQL エディタの入力］を開きます。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［入力クエリ エディタ］ウィンドウ」を参照してください。
2 ［TQL エディタの入力］で，トリガ CI ノードに「**SOURCE**」という名前を付けます。そのためには，ノードを右クリックして，［**ノードのプロパティ**］を選択します。［**要素名**］ボックスで，名前を「**SOURCE**」に変更します。
3 IP CI に Host CI と Contains 関係を追加します。［TQL エディタの入力］を使った作業の詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［入力クエリ エディタ］ウィンドウ」を参照してください。



IP CI が **HOST CI** に接続されます。入力 TQL は，**HOST** と **IP** の 2 つのノードと，その間のリンクで構成されます。**IP** CI には「**SOURCE**」という名前が付けられています。

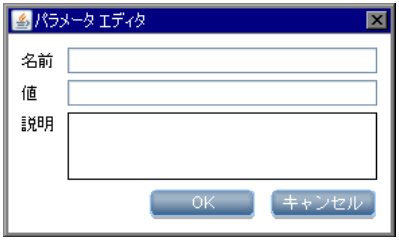
入力 TQL クエリに変数を追加する例：

この例では，前の例で作成した入力 TQL クエリに DIRECTORY 変数と CONFIGURATION_FILE 変数を追加します。これらの変数を使って，検出する必要がある IP にリンクされたホスト上の構成ファイルを見つけるために何を検出する必要があるかを定義します。

1 前の例で作成した入力 TQL を表示します。

［**データ フロー管理**］＞［**アダプタ管理**］にアクセスします。［トリガ CI データ］表示枠を見つけます。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「トリガ CI データ」を参照してください。

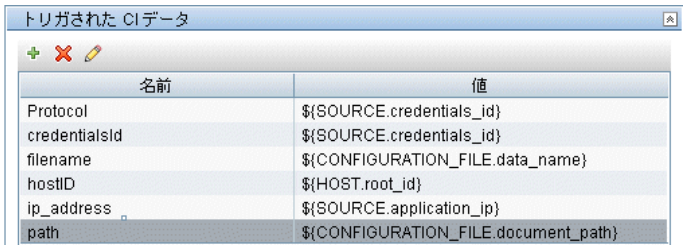
2 入力 TQL に変数を追加します。詳細については，［**データ フロー管理**］＞［**アダプタ管理**］にアクセスします。［トリガ CI データ］表示枠を見つけます。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「トリガ CI データ」の変数フィールドに関する説明を参照してください。



変数を実際のデータに置き換える例：

この例では，IP CI のデータ変数を，システム内の実際の IP CI インスタンスに存在する実際の値に置き換えます。

IP CI のトリガ CI データには，fileName 変数が含まれています。この変数を使って，入力 TQL の CONFIGURATION_FILE ノードを，ホスト上にある構成ファイルの実際の値に置き換えることができます。

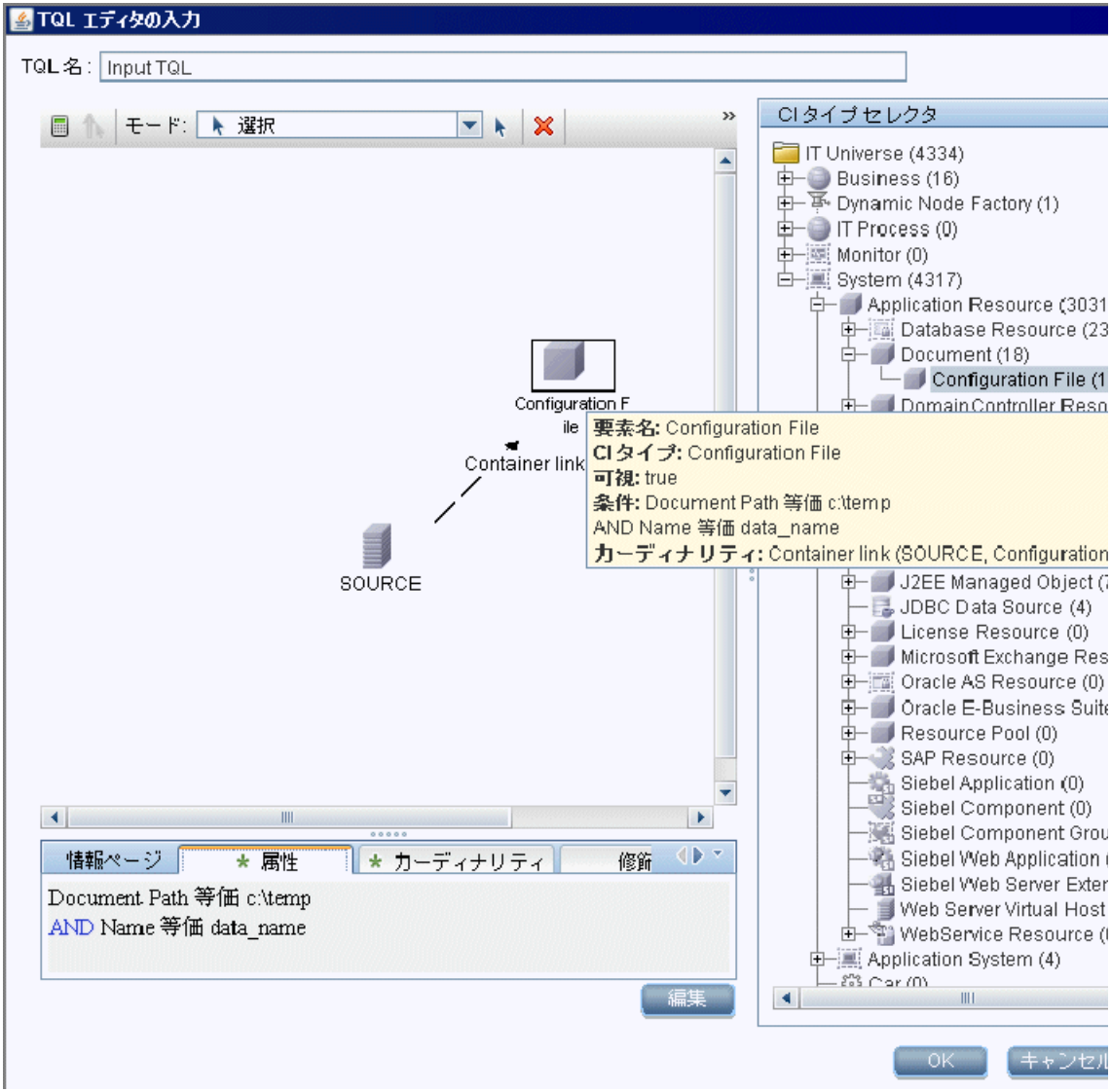


名前	値
Protocol	${SOURCE.credentials_id}
credentialsId	${SOURCE.credentials_id}
filename	${CONFIGURATION_FILE.data_name}
hostID	${HOST.root_id}
ip_address	${SOURCE.application_ip}
path	${CONFIGURATION_FILE.document_path}

トリガ CI データが Probe にアップロードされ，すべての変数が実際の値に置き換えられます。アダプタ・スクリプトには，DFM Framework を使って定義済み変数の実際の値を取得する次のコマンドが含まれています。

```
Framework.getTriggerCIData ('ip_address')
```

fileName および path 変数には，（前の例の入力 TQL で定義された）Configuration File ノードの data_name および document_path 属性が使用されます。

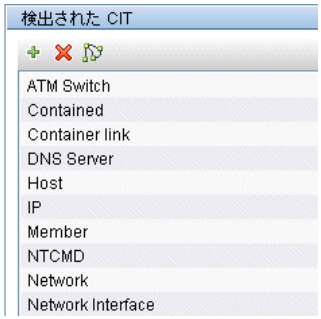


Protocol，credentialsId，および ip_address 変数には，次のように root_class，credentials_id，および application_ip 属性が使用されます。

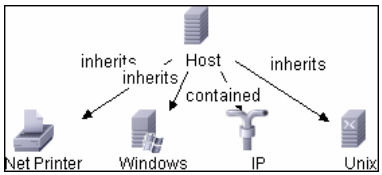
キー	名前	表示名	タイプ	詳細	標準設定値	可視
	ack_cleared_time	ack_cleared_time	long			
	ack_id	ack_id	string			
	BODY_ICON	BODY_ICON	string		ip	
	city	City	string	City locati...		✔
	codepage	CodePage	string	System s...		
	contextmenu	Context Menu	string_list	Context m...	itCIs	
	country	Country	string	Country lo...		✔
	credentials_id	Reference to the c...	string	Referenc...		
	data_adminstate	Admin State	adminstat...	Admin St...	Managed	

アダプタ出力の定義

アダプタの出力は，検出された CI のリスト（[**データ フロー管理**] ＞ [**アダプタ管理**] ＞ [**アダプタ定義**] タブ＞ [**検出 CIT**]）とその間のリンクです。



これらの CIT を，コンポーネントとそれらのリンク方法を表すトポロジ・マップで表示することもできます（[**検出した CIT をマップとして表示**] ボタンをクリックします）。



検出された CI は，DFM コード（Jython スクリプト）によって，UCMDB の ObjectStateHolderVector の形式で返されます。詳細については，72 ページの「Jython スクリプトによる結果の生成」を参照してください。

アダプタ出力の例 :

この例では，IP CI の出力にどの CIT を含めるかを定義します。

1 ［**データ フロー管理**］＞［**アダプタ管理**］にアクセスします。
2 ［リソース］表示枠で，［**Network**］＞［**アダプタ**］＞［**NSLOOKUP_on_Probe**］を選択します。
3 ［アダプタ定義］タブで，［検出 CIT］表示枠を見つけます。
4 アダプタ出力に含める CIT が一覧表示されます。リストに CIT を追加するか，リストから CIT を削除します。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［検出された CIT］表示枠」を参照してください。

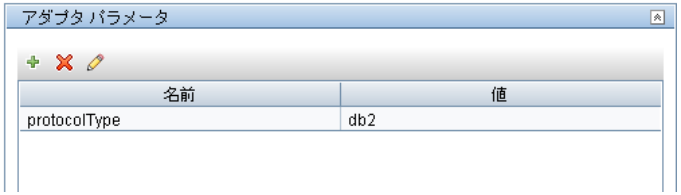
アダプタ・パラメータの上書き

複数のジョブに対してアダプタを設定するために，アダプタ・パラメータを上書きできます。たとえば，アダプタ SQL_NET_Dis_Connection は MSSQL Connection by SQL ジョブと Oracle Connection by SQL ジョブの両方で使用されます。

アダプタ・パラメータを上書きする例：

この例では，1 つのアダプタを使って Microsoft SQL Server と Oracle の両方のデータベースを検出できるようにアダプタ・パラメータを上書きする方法を示します。

1 ［**データ フロー管理**］＞［**アダプタ管理**］にアクセスします。
2 ［リソース］表示枠で，［Database_Basic］＞［**アダプタ**］＞［**SQL_NET_Dis_Connection**］を選択します。
3 ［アダプタ定義］タブで，［**アダプタ パラメータ**］表示枠を見つけます。protocolType パラメータの値は **all** になっています。



4 ［**SQL_NET_Dis_Connection_MsSql**］アダプタを右クリックし，［**ディスカバリ ジョブに移動**］＞［**MSSQL Server Connection by SQL**］を選択します。
5 ［プロパティ］タブを表示します。［パラメータ］表示枠を見つけます。

値 all を値 MicrosoftSQLServer で上書きします。

注：［**Oracle Database Connection by SQL**］ジョブには同じパラメータが含まれていますが，その値は値 Oracle で上書きされます。

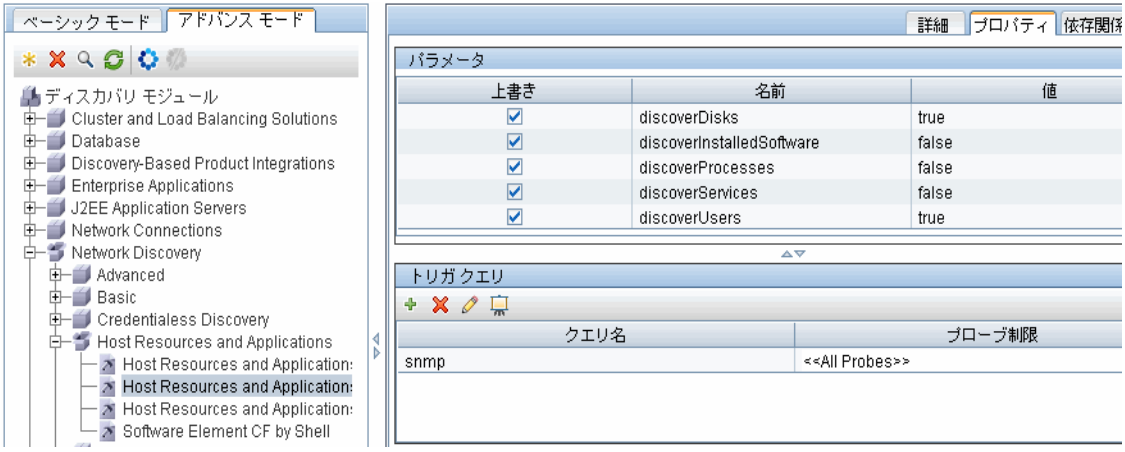
パラメータの追加，削除，編集の詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［アダプタ パラメータ］表示枠」を参照してください。

DFM は，このパラメータに従って Microsoft SQL Server インスタンスの検索を開始します。

手順 2：アダプタへのジョブの割り当て

各アダプタには，実行ポリシーを定義した 1 つ以上のジョブが関連付けられます。ジョブでは，起動された CI の異なるセットに対して異なる方法で同じアダプタのスケジュールを設定できます。また，セットごとに異なるパラメータを設定できます。

ジョブは [ディスカバリ モジュール] ツリーに表示され，ユーザはこのエンティティをアクティブ化します。



トリガ TQL

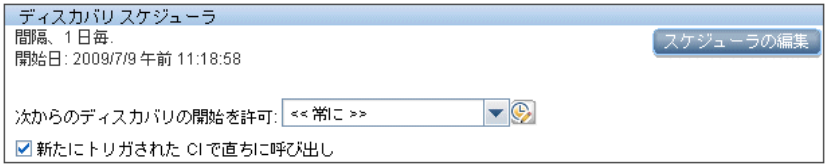
各ジョブは，トリガ TQL に関連付けられます。これらのトリガ TQL は，このジョブのアダプタに対する入力トリガ CI として使用される結果を発行します。

トリガ TQL は，入力 TQL に制約を追加できます。たとえば，入力 TQL の結果が SNMP に接続された IP である場合は，トリガ TQL の結果を SNMP に接続された 195.0.0.0 ～ 195.0.0.10 の範囲内にある IP にすることができます。

注：トリガ TQL は，入力 TQL が参照する同じオブジェクトを参照する必要があります。たとえば，入力 TQL によって SNMP を実行している IP を照会する場合は，入力 TQL で必要とされる SNMP オブジェクトに接続しない IP もあるため，ホストに接続された IP を照会するトリガ TQL を（同じジョブに対して）定義できません。

スケジュール

Probe のスケジュール情報は，トリガ CI に対してコードをいつ実行するかを指定します。［**新しいトリガ CI で直ちに呼び出し**］チェック・ボックスが選択されている場合は，以後のスケジュール設定に関係なく，トリガ CI が Probe に到達したときにも，各トリガ CI に対して 1 回ずつコードが実行されます。



各ジョブにスケジュール設定された時間ごとに，プローブはそのジョブで蓄積されたすべてのトリガ CI に対してコードを実行します。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［ディスカバリ スケジューラ］ダイアログ・ボックス」を参照してください。

パラメータ

ジョブを設定するときは，アダプタ・パラメータを上書きできます。詳細については，49 ページの「アダプタ・パラメータの上書き」を参照してください。

手順 3 : Jython コードの作成

HP Universal CMDB では, アダプタ記述に Jython スクリプトが使用されます。たとえば，SNMP を使用してマシンへの接続を試みる SNMP_NET_Dis_Connection アダプタでは，SNMP_Connection.py スクリプトが使用されます。Jython は，Python に基づき，Java によって強化された言語です。

Jython の使用方法の詳細については，次の Web サイトを参照してください。

- http://www.jython.org (英語サイト)
- http://www.python.org (英語サイト)

詳細については，66 ページの「Jython コードの作成」を参照してください。

2

ディスカバリ・コンテンツ移行ガイドライン

本章の内容

概念

タスク

参照先

概念

ディスカバリ・コンテンツ移行ガイドラインの概要

HP Universal CMDB バージョン 9.00 では，データ・モデルが大幅に進化しているため，以前の Discovery と Dependency Mapping（DDM）コンテンツ・コードに関連する変更が行われました。その結果，DDM コンテンツの一部のコア・メカニズムが変更されました。そのため，バージョン 9.00 より前の UCMDB に対して開発されたコンテンツは，9.00 データ・モデル（BDM : BTO データ・モデル）に合わせてアップグレードする必要があります。このセクションでは, DDM コンテンツを選択し，BDM に合わせて調整するプロセスについて説明します。

HP Universal CMDB のアップグレードの詳細については,『HP Universal CMDB デプロイメント・ガイド』（PDF）の「HP Universal CMDB バージョン 9.0 へのアップグレード」を参照してください。

バージョン 9.00 の新しいインフラストラクチャ機能

注 : BDM のオンライン・ドキュメントにアクセスする方法の詳細については, 60 ページの「BDM オンライン・ドキュメントへのアクセス」を参照してください。

本項の内容

BTO データ・モデル（BDM）

- BTO データ・モデル（BDM）の詳細については，概念データ・モデルに関するドキュメントを参照してください。このドキュメントでは，モデル化される概念とモデルの範囲について説明します。この概念データ・モデルは，モデル化されたドメインの意味論を理解する出発点となります。
- BDM クラスの詳細については，『HP Software BTO Data Model Reference』（英語版）を参照してください。このドキュメントでは，クラスの記述と属性，修飾子，階層情報を含む，すべての BDM クラスについて説明しています。

UCMDB 8.0x クラス・モデルと UCMDB 9.00 データ・モデル間の相違点

UCMDB バージョン 8.0x クラス・モデルと BDM 間の変更は，プローブのディスカバリ構成ファイル **C:¥hp¥UCMDB¥DataFlowProbe¥runtime¥probeManager¥discoveryConfigFiles¥flat-class-model-changes.xml** にダウンロードされます。

bdm_changes.xml: この XML ファイルには，クラス名，属性名，削除されたクラス，属性，修飾子などに関する情報が含まれます。

- UCMDB バージョン 8.0x クラス・モデルと BDM 間のマッピングの詳細については，『Mapping of UCMDB 9.0 (BTO Data Model) to UCMDB 8.0x Class Model』（英語版）を参照してください。
- バージョン 8.0x と 9.00 間でのクラス・モデルに対する変更の詳細については，『UCMDB Class Model Changes Report』（英語版）を参照してください。

新しい CIT 識別メカニズム

バージョン 9.00 より前のバージョンの UCMDB では，CI の識別にキー属性が使用されています。UCMDB バージョン 9.00 では，この概念は一般化され，識別は調整エンジンというサーバ・コンポーネントで行われるようになりました。調整エンジンは，DDA（データ定義アルゴリズム）ルールという論理ルールを使用して CI を識別できます。

この新しいメカニズムは，CIT の識別に関連するトポロジが重要である場合に最も効果的です（たとえば，以前のバージョンでは，Node CITHost は，IP アドレスやインタフェース CIT のような名前と関連するトポロジによって識別されました）。一部の CIT は引き続きキー属性で識別されます。そうした CIT の場合，DDA ルールは定義されません。

調整エンジンの詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「調整の概要」を参照してください。

実行中のソフトウェア・メカニズム

バージョン 8.0x の**ソフトウェア要素** CI は, バージョン 9.00 では**実行中のソフトウェア**と呼ばれます。バージョン 9.00 では，この CIT はキー属性ではなく DDA ルールによって識別されます。

実行中のソフトウェア CIT から派生したカスタム CIT を追加したとします。以前のバージョンでは，このカスタム CIT はそのキー属性で識別されていました。一方，バージョン 9.0 では継承された DDA ルールによって識別されるため，定義されたキー属性は無視されます。

そのため，派生した CIT を追加する場合は，次のことを考慮してください。

- 新しい CIT を実行中のソフトウェア CIT すべてと同じ DDA ルールで識別するには，現在の設定を維持する必要があります。
- 新しい CIT をキー属性で識別するには，新しい DDA ルールを作成してキー属性による識別を定義する必要があります。次に，このような DDA ルールの例を示します。このルールは **object** CIT に対して定義されています。

```
<identification-config type="object">
    <identification-criteria>
        <identification-criterion targetType="root">
            <key-attributes-condition/>
        </identification-criterion>
</identification-criteria>
</identification-config>
```

プローブ側の識別

DDM_ID_ATTRIBUTE: バージョン 9.00 の Data Flow Probe では，CI をキー属性（**ID_ATTRIBUTE**）のみによって識別します。CIT に DDA ルール（調整ルール）が含まれている場合，CIT にはキー属性が含まれていない可能性があります。この場合，CIT のメイン属性は，**DDM_ID_ATTRIBUTE** 修飾子でマークされています。したがって，CI を識別するためには，プローブですべての **DDM_ID_ATTRIBUTE** 修飾子と **ID_ATTRIBUTE** 修飾子を考慮する必要があります。

DDM_REQUIRED_TOPOLOGY: 特定の CIT 用の DDA ルールが，同じバルク内で，検証済み CI とともにレポートされるさまざまな CI に依存する場合があります。たとえば，**J2EE Domain** CIT の識別を行うには，ドメイン名属性だけでなく，メンバ・リンクによって接続されている **J2EE Application Server** CIT も必要です。

必要なすべての CI が検証済み CI を使用してレポートされるようにするには，必要なリンク・タイプを指定するデータ項目が含まれる **DDM_REQUIRED_TOPOLOGY** 修飾子でそれぞれの検証済み CI をマークする必要があります。たとえば，前の例では，**J2EE Domain** CIT は **DDM_REQUIRED_TOPOLOGY** 修飾子と**メンバ**・リンク・データ項目でマークされているため，ディスカバリが J2EE ドメインをレポートすると，サーバもレポートされます。

修飾子の詳細については，『モデリング・ガイド』の「［修飾子］ページ」を参照してください。

変換レイヤ

下位互換性のために，バージョン 9.00 のプローブには新しい変換メカニズムが導入されました。この新しいメカニズムでは，実行時にバージョン 8.0x のトポロジを 9.00 のトポロジに変換できます。これによりプローブは，バージョン 8.0x と互換性のあるトポロジをレポートする Jython スクリプトのようなタスクを引き続き実行できます。

新しい変換メカニズムでは，**bdm_changes.xml** ファイルに保持されているデータを使用し，必要な変更（クラスおよび属性名の変更，属性の削除，階層の変更など）を行って 8.0x のトポロジと BDM の互換性を確保します。同時に（かつプローブが実行するタスクによってレポートされるトポロジとは別に），UCMDB サーバは BDM と互換性のあるトポロジを受け取ります。

パッケージ移行ユーティリティ

UCMDB 9.00 のインストールには，外部のパッケージ移行ユーティリティが含まれています。コンテンツ開発者は，このユーティリティを使用してコンテンツ・パッケージを 8.0x クラス・モデルから 9.0x データ・モデルに変換できます。パッケージ移行ユーティリティは，パッケージ・リソースをサブシステム単位で変換して，新しいクラス・モデルとの互換性を確保します。CIT 定義，クエリ，ジョブ，アダプタ，およびモジュールは，**bdm_changes.xml** ファイルに保持されているデータに従って変換されます。変換後，これらを UCMDB 9.00 サーバにデプロイして使用できます。

詳細については，『HP Universal CMDB デプロイメント・ガイド』（PDF）の「パッケージのバージョン 8.04 から 9.0 へのアップグレード」を参照してください。

パッケージ移行ユーティリティの制限事項

- Jython スクリプトはパッケージ移行ユーティリティではアップグレードされません。UCMDB バージョン 8.0x のクラス・モデルに合わせて設計されたスクリプトをサポートするために，UCMDB 9.00 では新しく**変換レイヤ**・モジュールが導入されました。詳細については，57 ページの「変換レイヤ」を参照してください。
- インテグレーション・タイプのディスカバリ・アダプタは，パッケージ移行ユーティリティではアップグレードされず，手動でアップグレードする必要があります。
- レイヤ 2 トポロジ・ディスカバリ・ジョブ（およびディスカバリ・アダプタ，TQL などの対応するリソース）は大幅に変更されたため，パッケージ移行ユーティリティではアップグレードされずに削除されました。

クロスデータ・モデル・スクリプト開発のガイドライン

次のガイドラインは，バージョン 8.0x と 9.0x の両方に適用されます。

ディスカバリ・スクリプト API ライブラリ

ディスカバリ API ライブラリは完全な下位互換であるため，バージョン 8.0x のすべてのライブラリと API がサポートされます。詳細については，112 ページの「Jython のライブラリとユーティリティ」を参照してください。

9.00 API ではさらに要素とメソッドが追加されています。たとえば，Jython スクリプトでは，文字列のエラー・メッセージではなくエラー・コード（整数）をレポートするようになりました。これにより，エラー・メッセージのローカライズが可能になります。詳細については，115 ページの「エラー記述の表記規則」を参照してください。

実装のヒント

- **実行中のソフトウェア** CIT または関連するメソッドが存在する子孫を作成するには**モデリング**・モジュールを使用します。
- **Node** タイプの CIT を作成するには **HostBuilder** を使用します。
- OSH を ID で復元するには，**modeling.createOshByCmdbIdString** を使用します。
- すべてのシェル・ベースの接続には，**shellutils** モジュールの **ShellUtils** インスタンスを使用します。
- UCMDB バージョンを取得するには，組み込みのメカニズム logger.Version().getVersion(framework) を使用します。たとえば，追加の属性 application_ip は，UCMDB バージョン 9.0 以上にのみ追加されます。

```
versionAsDouble = logger.Version().getVersion(Framework)
if versionAsDouble >= 9:
    appServerOSH.setAttribute('application_ip', ip)
```

- WMI ベースのディスカバリを作成するには，**wmiutils** を使用します。
- SNMP ベースのディスカバリを作成するには，**snmputils** を使用します。

タスク

BDM オンライン・ドキュメントへのアクセス

BDM ドキュメントにアクセスするには，次の手順で行います。

1. HP Universal CMDB にログインします。
2. ［**ヘルプ**］ > ［**UCMDB ヘルプ**］をクリックします。
3. ホーム・ページで，［**Applications**］の下にある［**Modeling**］リンクをクリックして［**Modeling**］ポータルにアクセスします。
4. ［**Data Model**］タブをクリックします。

参照先

トラブルシューティングおよび制限事項

- 標準設定では，**ip_address** 値はパターンには渡されません。トリガ CI データとして明示的にパターンに追加する必要があります。
- 用意済みではない Jython スクリプトのクラスパスで必要な外部 jar またはリソースは，**discoveryResources** というサブフォルダの下の関連パッケージに置く必要があります。
- **StringVector** や **IntegerVector**（**BaseVector** から継承）など **List** タイプの属性を処理する場合，同じリスト・オブジェクトに対して**要素の追加**操作と**要素の削除**操作の両方を実行することはできません。

3

Jython アダプタの開発

本章の内容

概念

タスク

参照先

概念

HP データ・フロー管理 API 参考情報

使用可能な API の完全なドキュメントについては，『HP Universal CMDB データ・フロー管理 API 参考情報』を参照してください。ファイルは次のフォルダにあります。

C:¥hp¥UCMDB¥UCMDBServer¥deploy¥ucmdb-docs¥docs¥eng¥doc_lib¥DevRef_guide¥DDM_JavaDoc¥index.html

エラー・メッセージの概要

ディスカバリの実行中は，接続障害，ハードウェアの問題，例外，タイムアウトなど，多くのエラーが検出される可能性があります。DFM では，ベーシック・モードとアドバンス・モードのどちらのディスカバリ・コントロール・パネルでも，これらのメッセージが表示されます。ユーザは，問題の原因となったトリガ CI からドリルダウンして，エラー・メッセージ自体を表示できます。

DFM は，無視できるエラー（到達不可能なホストなど）と対処の必要なエラー（資格情報の問題，構成ファイルや DDL ファイルの欠落など）を区別します。さらに，その後の実行で同じエラーが発生してもエラーは 1 回しか報告されません。また，1 回しか発生しなかったエラーも報告されます。

エラーはすべて，プローブ・マネージャ・データベース・スキーマの discovery_problems テーブルに保存されています（エラー情報は，サーバへの配信を保証するため，プローブのメモリで処理されるのではなく，データベースに保存されます）。Probe には，各トリガ CI に関する問題の最新のリストが保持されます。各実行の後で，Probe は変化を確認し，それらを［ディスカバリステータス］表示枠に表示します。

パッケージを作成するときに，適切なメッセージをリソースとしてパッケージに追加できます。パッケージのデプロイ時に，メッセージも適切な場所にデプロイされます。メッセージは，115 ページの「エラー記述の表記規則」に記載されている表記規則に従っている必要があります。

DFM は多言語のエラー・メッセージをサポートします。記述したメッセージを，その地域の言語で表示するようにローカライズできます。

エラー・メッセージの検索の詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「[ディスカバリ ステータス]表示枠」を参照してください。

通信ログの設定の詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「[実行オプション] 表示枠」を参照してください。

タスク

Jython コードの作成

HP Universal CMDB では，アダプタ記述に Jython スクリプトが使用されます。たとえば，SNMP を使用してマシンへの接続を試みる SNMP_NET_Dis_Connection アダプタでは，SNMP_Connection.py スクリプトが使用されます。Jython は，Python に基づき，Java によって強化された言語です。

Jython の使用方法の詳細については，次の Web サイトを参照してください。

- http://www.jython.org
- http://www.python.org

次のセクションでは，実際に DFM フレームワーク内で Jython コードを記述する方法について説明します。本項では，特に，Jython スクリプトとそれを呼び出す DDM Framework との接点について取り上げ，できる限り使用する必要がある Jython のライブラリやユーティリティについても説明します。

注 :

- DFM 用に記述されたスクリプトは，Jython バージョン 2.1 と互換性がある必要があります。
- 使用可能な API の完全なドキュメントについては，『HP Universal CMDB データ・フロー管理 API 参考情報』を参照してください

本項の内容

➤ 77 ページの「（接続アダプタ用の）正しい資格情報の検索」

➤ 79 ページの「Java の例外の処理」

Jython 内での外部 Java JAR ファイルの使用

新しい Jython スクリプトを開発するときは，Java ユーティリティ・アーカイブ，接続アーカイブ（JDBC ドライバ JAR ファイルなど），または実行可能ファイル（たとえば，資格情報なしのディスカバリでは **nmap.exe** が使用されます）として，外部 Java ライブラリ（JAR ファイル）またはサードパーティの実行可能ファイルが必要になることがあります。

これらのリソースは，パッケージの **外部リソース・フォルダ**・フォルダ内にバンドルする必要があります。このフォルダに格納されたリソースは，HP Universal CMDB サーバに接続する Probe に自動的に送信されます。

また，ディスカバリが起動されると，JAR ファイル・リソースが Jython のクラスパスに読み込まれ，その中にあるすべてのクラスをインポートして使用できるようになります。

コードの実行

ジョブがアクティブ化されると，必要なすべての情報を含むタスクが Probe にダウンロードされます。

プローブは，タスクに指定された情報を使って DFM コードの実行を開始します。

Jython コードのフローは，スクリプトのメイン・エントリから実行を開始し，CI を検出するコードを実行し，その結果として検出された CI のベクトルを提供します。

用意済みスクリプトの変更

用意済みスクリプトを変更するときは，スクリプトの変更を最小限にとどめて，必要なメソッドを外部スクリプトに配置します。変更をより効率的に追跡できるようになり，新しいバージョンの HP Universal CMDB に移行するときにコードが上書きされません。

たとえば，次に示す用意済みスクリプト内の 1 行のコードは，アプリケーション固有の方法で Web サーバ名を計算するメソッドを呼び出します。

```
        serverName = iplanet_cspecific.PlugInProcessing(serverName, transportHN,
mam_utils)
```

この名前の計算方法を決定するより複雑なロジックは，次の外部スクリプトに含まれています。

```
# implement customer specific processing for 'servername' attribute of httpplugin

#

def PlugInProcessing(servername, transportHN, mam_utils_handle):

        # support application-specific HTTP plug-in naming

        if servername == "appsrv_instance":

                # servername is supposed to match up with the j2ee server name,
however some groups do strange things with their

                # iPlanet plug-in files.this is the best work-around we could find.this join
can't be done with IP address:port

                # because multiple apps on a web server share the same IP:port for
multiple websphere applications

                logger.debug('httpcontext_webapplicationserver attribute has been
changed from [' + servername + '] to [' + transportHN[:5] + '] to facilitate websphere
enrichment')

                servername = transportHN[:5]

        return servername
```

この外部スクリプトを外部リソース・フォルダに保存します。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「[リソース] 表示枠」を参照してください。このスクリプトをパッケージに追加すると，ほかのジョブでもこのスクリプトを使用できるようになります。パッケージ・マネージャを使った作業の詳細については，『HP UCMDB 管理ガイド』の「パッケージ・マネージャ」を参照してください。

アップグレードを行ったときは，この 1 行のコードに対して行った変更が用意済みスクリプトの新しいバージョンによって上書きされるため，行を置き換える必要があります。ただし，外部スクリプトは上書きされません。

Jython ファイルの構造

Jython ファイルは，一定の順序で並んだ次の 3 つの部分で構成されます。

1. インポート
2. メイン関数 - DiscoveryMain
3. 関数定義（任意）

以下に，Jython スクリプトの例を示します。

```
# imports section
from appilog.common.system.types import ObjectStateHolder
from appilog.common.system.types.vectors import ObjectStateHolderVector

# Function definition
def foo:
    # do something

# Main Function
def DiscoveryMain(Framework):
    OSHVResult = ObjectStateHolderVector()

    ## Write implementation to return new result CIs here...

    return OSHVResult
```

インポート

Jython のクラスは，階層構造の名前空間に存在します。バージョン 7.0 以降では，以前のバージョンと異なり，暗黙的なインポートがないため，使用するすべてのクラスを明示的にインポートする必要があります（この変更は，パフォーマンス上の理由と，必要な詳細を隠さないようにすることで Jython スクリプトをわかりやすくする目的で行われました）。

➤ Jython スクリプトをインポートするには，次のようにします。

```
import logger
```

➤ Java クラスをインポートするには，次のようにします。

```
from appilog.collectors.clients import ClientsConsts
```

メイン関数 DiscoveryMain

実行可能な各 Jython スクリプト・ファイルには，メイン関数である DiscoveryMain が含まれています。

DiscoveryMain 関数は，スクリプトのメイン・エントリ（最初に実行される関数）です。関数は，スクリプトのメイン・エントリ（最初に実行される関数）です。

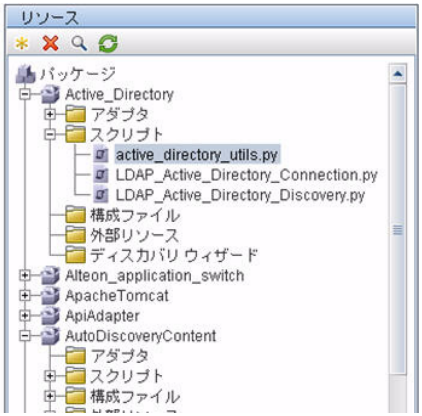
```
def DiscoveryMain(Framework):
```

メイン関数の定義では，Framework 引数を指定する必要があります。この引数は，メイン関数がスクリプトの実行に必要な情報（トリガ CI の情報やパラメータなど）を取得するために使用します。また，スクリプトの実行中に発生したエラーについて報告するためにも使用できます。

メイン・メソッドなしで Jython スクリプトを作成することもできます。このようなスクリプトは，ほかのスクリプトから呼び出されるライブラリ・スクリプトとして使用されます。

関数定義

各スクリプトには，メイン・コードから呼び出される追加の関数を含めることができます。このような関数から，現在のスクリプトまたは（import ステートメントを使って）別のスクリプトに存在する別の関数を呼び出すこともできます。ほかのスクリプトを使用するには，それをパッケージの［スクリプト］セクションに追加する必要があります。



関数から別の関数を呼び出す例 :

次の例では，メイン・コードから doQueryOSUsers(..) メソッドを呼び出し，そこからさらに内部メソッドの doOSUserOSH(..) を呼び出しています。

```
def doOSUserOSH(name):
    sw_obj = ObjectStateHolder('winosuser')

    sw_obj.setAttribute('data_name', name)
    # return the object
    return sw_obj

def doQueryOSUsers(client, OSHVResult):
    _hostObj = modeling.createHostOSH(client.getIpAddress())
    data_name_mib = '1.3.6.1.4.1.77.1.2.25.1.1,1.3.6.1.4.1.77.1.2.25.1.2,string'
    resultSet = client.executeQuery(data_name_mib)
    while resultSet.next():
        UserName = resultSet.getString(2)
        ########## send object ##############
        OSUserOSH = doOSUserOSH(UserName)
        OSUserOSH.setContainer(_hostObj)
        OSHVResult.add(OSUserOSH)

def DiscoveryMain(Framework):
    OSHVResult = ObjectStateHolderVector()
    try:
        client =
Framework.getClientFactory(ClientsConsts.SNMP_PROTOCOL_NAME).createClient()
    except:
        Framework.reportError('Connection failed')
    else:
        doQueryOSUsers(client, OSHVResult)
        client.close()
    return OSHVResult
```

このスクリプトが多くのアダプタに関係するグローバルなライブラリである場合は，個々のアダプタに追加する代わりに，このスクリプトを jythonGlobalLibs.xml 構成ファイル内のスクリプトのリストに追加できます（［**アダプタ管理**］＞［**リソース**］表示枠＞［**AutoDiscoveryContent**］＞［**構成ファイル**］）。

Jython スクリプトによる結果の生成

各 Jython スクリプトは，特定のトリガ CI に対して実行され，DiscoveryMain 関数の戻り値によって返される結果とともに終了します。

スクリプトの結果は，実際には CMDB で挿入または更新される CI とリンクのグループです。スクリプトは，この CI とリンクのグループを ObjectStateHolderVector の形式で返します。

ObjectStateHolder クラスは，CMDB で定義されたオブジェクトまたはリンクを表す手段です。ObjectStateHolder オブジェクトには，CIT の名前と，属性とその値のリストが含まれています。ObjectStateHolderVector は，ObjectStateHolder インスタンスのベクトルです。

ObjectStateHolder の構文

本セクションでは，DFM の結果を UCMDB モデルに組み込む方法について説明します。

CI の属性設定の例 :

ObjectStateHolder クラスは，DFM の結果グラフを記述します。個々の CI とリンク（関係）は，次の Jython コード例のように，ObjectStateHolder クラスのインスタンスの内部に置かれます。

```
# siebel application server
1 appServerOSH = ObjectStateHolder('siebelappserver' )
2 appServerOSH.setStringAttribute('data_name', sblsvrName)
3 appServerOSH.setStringAttribute ('application_ip', ip)
4 appServerOSH.setContainer(appServerHostOSH)
```

- 第 1 行では，**siebelappserver** タイプの CI を作成します。
- 第 2 行では，サーバ名として検索された値が設定された Jython 変数である **sblsvrName** の値を持つ **data_name** という名前の属性を作成します。
- 第 3 行では，CMDB で更新される非キー属性を設定します。
- 第 4 行では，包含関係を構築します（結果はグラフになります）。このアプリケーション・サーバがホスト（範囲内の別の ObjectStateHolder クラス）に含まれることを指定します。

注 : Jython スクリプトによって報告される各 CI には，その CI の CI タイプのすべてのキー属性の値を含める必要があります。

関係（リンク）の例：

次のリンクの例を使って，グラフがどのように表されるかを説明します。

```
1 linkOSH = ObjectStateHolder('route')
2 linkOSH.setAttribute('link_end1', gatewayOSH)
3 linkOSH.setAttribute('link_end2', appServerOSH)
```

- 第 1 行では，リンクを作成します（このリンクは ObjectStateHolder クラスでもあります。唯一の違いは，route がリンク CI タイプであることです）。
- 第 2 行と第 3 行では，各リンクの端にあるノードを指定します。そのためには，リンクの **end1** および **end2** 属性を使用します。これらの属性は，（各リンクの最低限のキー属性であるため）必ず指定します。属性の値は，ObjectStateHolder インスタンスです。エンド 1 とエンド 2 の詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「リンク」を参照してください。

注意：リンクには方向があります。エンド 1 ノードとエンド 2 ノードが各端の有効な CIT に対応していることを確認する必要があります。ノードが有効でない場合は，結果オブジェクトが検証に失敗し，正しく報告されません。詳細については，『モデリング・ガイド』の「CI タイプの関係」を参照してください。

ベクトル（CI 収集）の例：

属性とともにオブジェクトを作成し，端にあるオブジェクトとともにリンクを作成したら，それらをグループにまとめる必要があります。そのためには，次のように ObjectStateHolderVector インスタンスにそれらを追加します。

```
oshvMyResult = ObjectStateHolderVector()
oshvMyResult.add(appServerOSH)
oshvMyResult.add(linkOSH)
```

この複合結果を Framework に報告して CMDB サーバに送信できるようにする方法の詳細については，sendObjects メソッドを参照してください。

結果グラフを ObjectStateHolderVector インスタンスにまとめたら，それを DFM フレームワークに返して CMDB に挿入する必要があります。そのためには，DiscoveryMain() 関数の結果として ObjectStateHolderVector インスタンスを返します。

注：一般的な CIT の **OSH** を作成する方法の詳細については，112 ページの「Jython のライブラリとユーティリティ」の **modeling.py** を参照してください。

Framework インスタンス

Framework インスタンスは, Jython スクリプトのメイン関数に渡される唯一の引数です。これは，スクリプトの実行に必要な情報（トリガ CI の情報やアダプタ・パラメータなど）を取得するために使用できます。また，スクリプトの実行中に発生したエラーについて報告するためにも使用できます。詳細については，64 ページの「HP データ・フロー管理 API 参考情報」を参照してください。

本項では，Framework の最も重要な使用法について説明します。

- 74 ページの「Framework.getTriggerCIData(String attributeName)」
- 75 ページの「Framework.createClient(credentialsId, props)」
- 76 ページの「Framework.getParameter (String parameterName)」
- 77 ページの「Framework.reportError(String message) および Framework.reportWarning(String message)」

Framework.getTriggerCIData(String attributeName)

この API は, アダプタで定義されたトリガ CI データとスクリプトの間の中間段階を提供します。

資格情報を取得する例 :

次のトリガ CI データの情報を要求します。

トリガされた CI データ

名前	値
Protocol	${SOURCE.credentials_id}
credentialsId	${SOURCE.credentials_id}
filename	${CONFIGURATION_FILE.data_name}
hostID	${HOST.root_id}
ip_address	${SOURCE.application_ip}
path	${CONFIGURATION_FILE.document_path}

タスクから資格情報を取得するには，次の API を使用します。

```
credId = Framework.getTriggerCIData('credentialsId')
```

Framework.createClient(credentialsId, props)

リモート・マシンと接続するには，クライアント・オブジェクトを作成し，そのクライアントに対してコマンドを実行します。クライアントを作成するには，ClientFactory クラスを取得します。getClientFactory() メソッドによって，要求されるクライアント・プロトコルのタイプを取得します。プロトコルの定数は，ClientsConsts クラスに定義されています。資格情報とサポートされているプロトコルの詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「ドメイン資格情報リファレンス」を参照してください。

資格情報 ID に対応する Client インスタンスを作成する例 :

資格情報 ID に対応する Client インスタンスを作成するには，次のようにします。

```
properties = Properties()
codePage = Framework.getCodePage()
properties.put( BaseAgent.ENCODING, codePage)
client = Framework.createClient(credentailsID ,properties)
```

これで，Client インスタンスを使って該当するマシンまたはアプリケーションに接続できます。

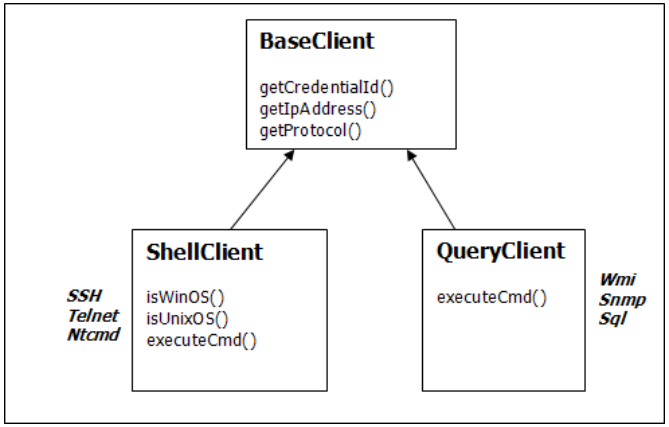
WMI クライアントを作成して WMI クエリを実行する例 :

WMI クライアントを作成し，そのクライアントを使って WMI クエリを実行するには，次のようにします。

```
wmiClient = Framework.createClient(credential)
resultSet = wmiClient.executeQuery("SELECT TotalPhysicalMemory
 FROMWin32_LogicalMemoryConfiguration")
```

注 : createClient() API を動作させるには，［トリガ CI データ］表示枠内で，パラメータ **credentialsId = ${SOURCE.credentials_id}** をトリガ CI データ・パラメータに追加します。または，関数 **wmiClient = clientFactory().createClient(credentials_id)** を呼ぶときに，手動で資格情報 ID を追加できます。

次の図は，クライアントで一般的にサポートされる API とともにクライアントの階層を示します。



クライアントとそれらがサポートする API の詳細については,『HP Universal CMDB データ・フロー管理 API 参考情報』の BaseClient，ShellClient，QueryClient を参照してください。

Framework.getParameter (String parameterName)

トリガ CI に関する情報を取得するのに加えて，多くの場合，アダプタ・パラメータの値を取得する必要があります。たとえば，

パラメータ

上書き	名前	値
☑	protocolType	MicrosoftSQLServer

protocolType パラメータの値を取得する例 :

Jython スクリプトから protocolType パラメータの値を取得するには，次の API を使用します。

```
protocolType = Framework.getParameterValue('protocolType')
```

Framework.reportError(String message) および Framework.reportWarning(String message)

スクリプトの実行中に，何らかのエラー（接続の障害，ハードウェアの問題，タイムアウトなど）が発生することがあります。このようなエラーが検出されたときは，Framework によってその問題を報告できます。報告されたメッセージはサーバに到達し，ユーザに対して表示されます。

エラーとメッセージの報告の例 :

次の例は，reportError(<Error Msg>) API の使用例です。

```
try:
        client =
Framework.getClientFactory(ClientsConsts.SNMP_PROTOCOL_NAME)
        createClient()
except:
        strException = str(sys.exc_info()[1]).strip()
        Framework.reportError ('Connection failed: %s' % strException)
```

問題を報告するには，Framework.reportError(String message) と Framework.reportWarning(String message) のいずれかの API を使用します。2 つの API の違いは，エラーを報告する場合はプローブがセッション全体のパラメータを含む通信ログ・ファイルをファイル・システムに保存することです。これによって，セッションを追跡し，エラーをより深く理解することができます。

（接続アダプタ用の）正しい資格情報の検索

リモート・システムに接続するアダプタでは，可能なすべての資格情報を試行する必要があります。（ClientFactory を使って）クライアントを作成するときに必要なパラメータの 1 つは，資格情報 ID です。接続スクリプトは，可能な資格情報セットにアクセスし，clientFactory.getAvailableProtocols() メソッドを使って資格情報を 1 つずつ試行します。ある資格情報セットが成功すると，アダプタはそのトリガ CI のホスト上にある CI 接続オブジェクトを（その IP と一致する資格情報 ID とともに）CMDB に報告します。その後のアダプタでは，この接続オブジェクト CI を使って資格情報セットに直接接続できます（つまり，各アダプタで再び可能なすべての資格情報を試行する必要はありません）。

次の例は，SNMP プロトコルのすべてのエントリを取得する方法を示します。この例では，IP をトリガ CI データから取得しています（# Get the Trigger CI data values）。

接続スクリプトは，可能なすべてのプロトコル資格情報を要求し（# Go over all the protocol credentials），いずれかが成功するまでループでそれらを試行します（resultVector）。詳細については，35 ページの「アダプタの分割」の「**2 段階接続の枠組み**」を参照してください。

```
import logger
from appilog.collectors.clients import ClientsConsts
from appilog.common.system.types.vectors import ObjectStateHolderVector

        def mainFunction(Framework):
resultVector = ObjectStateHolderVector()

        # Get the Trigger CI data values
        ip_address = Framework.getDestinationAttribute('ip_address')
        ip_domain = Framework.getDestinationAttribute('ip_domain')

        # Create the client factory for SNMP
        clientFactory = framework.getClientFactory(ClientsConsts.SNMP_PROTOCOL_NAME)
        protocols = clientFactory.getAvailableProtocols(ip_address, ip_domain)
```

```
connected = 0
# Go over all the protocol credentials
for credentials_id in protocols:
    client = None
    try:
        # try to connect to the snmp agent
        client = clientFactory.createClient(credentials_id)

        // Query the agent
        .

        # connection succeed
        connected = 1
    except:
        if client != None:
            client.close()
if (not connected):
    logger.debug('Failed to connect using all credentials')
else:
    // return the results as OSHV
    return resultVector
```

Java の例外の処理

一部の Java クラスは障害発生時に例外をスローします。この例外をキャッチして処理することをお勧めします。そうしないと，アダプタが予期せずに終了する原因になります。

既知の例外をキャッチするときは，ほとんどの場合，例外のスタック・トレースをログに出力し，適切なメッセージを UI に発行する必要があります。次に例を示します。

```
try:
        client = Framework.getClientFactory().createClient()
except Exception, msg:
        Framework.reportError('Connection failed')
        logger.debugException('Exception while connecting: %s' % (msg))
        return
```

例外が致命的なものでなく，スクリプトを続行できる場合は，reportError() メソッドの呼び出しを省略して，スクリプトを続行できるようにする必要があります。

Jython アダプタでのローカライズのサポート

多言語ロケール機能によって，DFM をさまざまなオペレーティング・システム（OS）言語で使用し，実行時に適切なカスタマイズを有効にすることができます。

Content Pack 3.00 より前では，DFM では静的に指定したエンコーディングですべてのネットワーク・ターゲットの出力を処理していました。ただし，この方法は多言語 IT ネットワークには適しておらず，さまざまな OS 言語のホストを検出するには，プローブ管理者は，それぞれ別のジョブ・パラメータで DFM ジョブを数回，手動で再実行する必要がありました。この手順では，ネットワークに多大な負荷がかかるうえに，トリガ CI での即時のジョブ呼び出しや，スケジュール・マネージャによる UCMDB のデータの自動更新など，DFM の重要な機能を使用できませんでした。

現在，標準設定でサポートされているロケール言語は，日本語，ロシア語，ドイツ語です。標準設定のロケールは英語です。

本項の内容

新しい言語サポートの追加

このタスクでは，新しい言語のサポートを追加する方法を説明します。

このタスクには次の手順が含まれます。

1 リソース・バンドル（*.properties ファイル）の追加

実行するジョブに合わせて，リソース・バンドルを追加します。次の表に，DFM のジョブと各ジョブで使用するリソース・バンドルを示します。

ジョブ	リソース・バンドルのベース名
File Monitor by Shell	langFileMonitoring
Host Resources and Applications by Shell	langHost_Resources_By_TTY, langTCP
Hosts by Shell using NSLOOKUP in DNS Server	langNetwork
Host Connection by Shell	langNetwork
Collect Network Data by Shell or SNMP	langTCP
Host Resources and Applications by SNMP	langTCP
Microsoft Exchange Connection by NTCMD, Microsoft Exchange Topology by NTCMD	msExchange
MS Cluster by NTCMD	langMsCluster

バンドルの詳細については，86 ページの「リソース・バンドルを使用する作業」を参照してください。

2 Language オブジェクトの宣言と登録

新しい言語を定義するには，次の 2 行のコードを **shellutils.py** スクリプトに追加します。これには，現在，サポートされているすべての言語のリストが含まれています。このスクリプトは AutoDiscoveryContent パッケージに格納されています。スクリプトを表示するには，［アダプタ管理］ウィンドウにアクセスします。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［アダプタ管理 ］ウィンドウ」を参照してください。

a 次のように，言語を宣言します。

```
LANG_RUSSIAN = Language(LOCALE_RUSSIAN, 'rus', ('Cp866', 'Cp1251'),
(1049,), 866)
```

クラス言語の詳細については，87 ページの「API リファレンス」を参照してください。Class Locale オブジェクトの詳細については，http://java.sun.com/j2se/1.5.0/docs/api/java/util/Locale.html を参照してください。既存のロケールを使用するか，新しいロケールを定義できます。

b 言語を次のコレクションに追加して登録します。

```
LANGUAGES = (LANG_ENGLISH, LANG_GERMAN, LANG_SPANISH,
LANG_RUSSIAN, LANG_JAPANESE)
```

標準設定の言語の変更

OS の言語を判別できない場合，標準設定の言語が使用されます。標準設定の言語は，**shellutils.py** ファイルで指定されています。

```
#default language for fallback
DEFAULT_LANGUAGE = LANG_ENGLISH
```

標準設定の言語を変更するには, DEFAULT_LANGUAGE 変数を別の言語で初期化します。詳細については，80 ページの「新しい言語サポートの追加」を参照してください。

エンコーディング文字セットの決定

コマンド出力のデコーディングに適切な文字セットは，実行時に判別されます。多言語の解決は，次の事実と前提条件に基づいています。

1. OS 言語は，ロケールに依存しない方法で判別できます。たとえば，Windows の場合は **chcp** コマンド，Linux の場合は **locale** コマンドを実行して判別できます。
2. 言語とエンコーディングの関係はよく知られており, 静的に定義できます。たとえば，ロシア語には，最もよく使用されるエンコーディングとして Cp866 と Windows-1251 の 2 つがあります。
3. 各言語に 1 つの文字セットを使用することをお勧めします。たとえば，ロシア語の推奨される文字セットは Cp866 です。これによって，ほとんどのコマンドの出力がこのエンコーディングで行われます。

4 次のコマンド出力のエンコーディングは予測できませんが，所定の言語で使用可能なエンコーディングのいずれかです。たとえば，Windows マシンでロシア語ロケールを使用している場合，**ver** コマンドの出力は CP866 で行われますが，**ipconfig** コマンドの出力は Windows-1251 で行われます。

5 既知のコマンドの出力には，既知のキーワードが含まれています。たとえば，**ipconfig** コマンドには，**IP-Address** 文字列が変換されて含まれています。したがって，**ipconfig** コマンドの出力では，英語の OS の場合は **IP-Address** が，ロシア語の OS の場合はIP-Адрес が，ドイツ語の OS の場合は **IP-Adresse** が含まれます。

コマンド出力の言語が検出されると（# 1），使用可能な文字セットは 1 つか 2 つに限定されます（# 2）。さらに，この出力には既知のキーワードが含まれています（# 5）。

したがって，結果でキーワードを検索し，使用可能なエンコーディングのいずれかを使用して，コマンド出力をデコードすることで解決できます。キーワードが見つかった場合，現在の文字セットが適切な文字セットとみなされます。

ローカライズしたデータを使用する新しいジョブの定義

このタスクでは，ローカライズしたデータを使用する新しいジョブを作成する方法について説明します。

Jython スクリプトでは，通常，コマンドを実行して出力を解析します。このコマンド出力を適切にデコーディングして受け取るには，**ShellUtils** クラス用の API を使用します。詳細については，278 ページの「HP Universal CMDB Web サービス API の概要」を参照してください。

このコードは，通常，次の形式になっています。

```
client = Framework.createClient(protocol, properties)
shellUtils = shellutils.ShellUtils(client)
languageBundle = shellutils.getLanguageBundle (langNetwork, shellUtils.osLanguage,
Framework)
strWindowsIPAddress = languageBundle.getString(windows_ipconfig_str_ip_address)
ipconfigOutput = shellUtils.executeCommandAndDecode(ipconfig /all,
strWindowsIPAddress)
#Do work with output here
```

1 クライアントを作成します。

```
client = Framework.createClient(protocol, properties)
```

2 **ShellUtils** クラスのインスタンスを作成し，オペレーティング・システムの言語を追加します。この言語を追加しないと，標準設定の言語（通常は英語）が使用されます。

```
shellUtils = shellutils.ShellUtils(client)
```

オブジェクトの初期化中，DFM によってマシンの言語が自動的に検出され，定義済みの Language オブジェクトから推奨されるエンコーディングが設定されます。推奨されるエンコーディングは，エンコーディング・リストで最初に表示されているインスタンスです。

3 **getLanguageBundle** メソッドを使用して，**shellclient** から適切なリソース・バンドルを取得します。

```
languageBundle = shellutils.getLanguageBundle (langNetwork, shellUtils.osLanguage,
Framework)
```

4 リソース・バンドルから，特定のコマンドに対して適切なキーワードを取得します。

```
strWindowsIPAddress = languageBundle.getString(windows_ipconfig_str_ip_address)
```

5 **executeCommandAndDecode** メソッドを呼び出し，**ShellUtils** オブジェクトでキーワードを渡します。

```
ipconfigOutput = shellUtils.executeCommandAndDecode(ipconfig /all,
strWindowsIPAddress)
```

ShellUtils object（このメソッドの詳細が説明されている）API 参考情報をユーザに示すためにも必要です。

6 出力を通常どおり解析します。

キーワードを使用しないコマンドのデコード

現在のローカライズ方法では，すべてのコマンド出力のデコードにキーワードを使用しています。詳細については，83 ページの「ローカライズしたデータを使用する新しいジョブの定義」の 84 ページ手順「4」を参照してください。

ただし，最初のコマンド出力にのみキーワードを使用し，それ以降のコマンドには，最初のコマンドのデコードに使用した文字セットを使用する方法もあります。ただし，最初のコマンド出力にのみキーワードを使用し，それ以降のコマンドには，最初のコマンドのデコードに使用した文字セットを使用する方法もあります。この方法を使うには，**ShellUtils** オブジェクトの **getCharsetName** メソッドと **useCharset** メソッドを使用します。

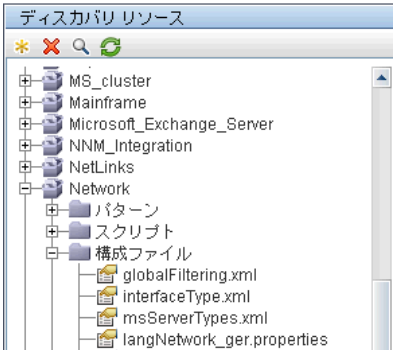
次に，一般的な使用例を示します。

1. **executeCommandAndDecode** メソッドを 1 回呼び出します。
2. **getCharsetName** メソッドにより，直近に使用した文字セットの名前を取得します。
3. **shellUtils** の標準設定でこの文字セットを使用するため，**ShellUtils** オブジェクトで **useCharset** メソッドを呼び出します。
4. **ShellUtils** の **execCmd** メソッドを 1 回以上呼び出します。85 ページ手順「3」で指定した文字セットで出力が返されます。これ以外のデコーディング操作は発生しません。

リソース・バンドルを使用する作業

リソース・バンドルは，プロパティの拡張子（***.properties**）を持つファイルです。プロパティ・ファイルは，データが「key = value」という形式で保存されている辞書とみなすことができます。プロパティ・ファイルの各行には，1 組の「key = value」の関連付けが収められています。リソース・バンドルの主な機能は，キーによって値を返すことです。

リソース・バンドルはプローブ・マシンの **C:¥hp¥UCMDB¥DataFlowProbe¥runtime¥probeManager¥discoveryConfigFiles** に置かれています。これらはほかの構成ファイルと同様に UCMDB サーバからダウンロードします。［リソース］ウィンドウで，編集，追加，削除できます。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［構成ファイル］表示枠」を参照してください。



DFM で宛先を検出するときは，通常，コマンド出力またはファイル・コンテンツからテキストを解析する必要があります。多くの場合，この解析は正規表現に基づいています。解析に使用する正規表現は，言語によって異なります。すべての言語に対応するコードを記述するには，言語固有のすべてのデータをリソース・バンドルに抽出する必要があります。言語ごとにリソース・バンドルがあります（リソース・バンドルに複数の異なる言語のデータが含まれていることもありますが，DFM では，1 つのリソース・バンドルに常に 1 つの言語のデータが含まれています）。

Jython スクリプト自体には，ハード・コーディングされた言語固有のデータ（言語固有の正規表現など）はありません。このスクリプトによって，リモート・システムの言語の判別，適切なリソース・バンドルの読み込み，特定のキーによる言語固有のすべてのデータの取得が行われます。

DFM では，リソース・バンドルは <base_name>_<language_identifier>.properties という固有の名前形式を取り，たとえば langNetwork_spa.properties のようになります（標準のリソース・バンドルの形式は <base_name>.properties で，たとえば langNetwork.properties のようになります）。

形式の base_name は，このバンドルの目的を表します。たとえば，**langMsCluster** は，そのリソース・バンドルに MS Cluster ジョブで使用する言語固有リソースが含まれていることを示しています。

形式の language_identifier は，言語を識別する 3 文字の略語です。たとえば，rus はロシア語を，ger はドイツ語を意味します。言語識別子は，Language オブジェクトの宣言に含まれます。

API リファレンス

本項の内容

Language クラス

このクラスは，リソース・バンドルのポストフィックスや使用可能なエンコーディングなど，言語に関する情報をカプセル化します。

フィールド

名前	説明
locale	ロケールを表す Java オブジェクト。
bundlePostfix	リソース・バンドルのポストフィックス。このポストフィックスは，リソース・バンドル名で言語を示すために使用されます。たとえば，**langNetwork_ger.properties** というバンドルには，**ger** がバンドルのポストフィックスとして含まれています。
charsets	この言語のエンコードに使用する文字セット。1 つの言語に対し，複数の文字セットが存在できます。たとえば，ロシア語には，一般的なエンコーディングとして Cp866 と Windows-1251 があります。
wmiCodes	Microsoft Windows OS が言語を識別するために使用する WMI コードのリスト。使用可能なコードのリストは http://msdn.microsoft.com/en-us/library/aa394239(VS.85).aspx（OSLanguage の項）にあります。OS の言語を識別する方法には，WMI クラス OS に対して OSLanguage プロパティを問い合わせる方法があります。
codepage	特定の言語で使用するコード・ページ。たとえば，ロシア語のマシンでは 866 を，英語のマシンでは 437 を使用します。OS の言語を識別するには，標準設定のコード・ページを取得する方法があります（例 : chcp コマンドを使用）。

executeCommandAndDecode メソッド

このメソッドは，Jython スクリプトのビジネス・ロジックで使用されます。デコーディング操作をカプセル化し，デコーディングしたコマンド出力を返します。

引数

名前	説明
cmd	実行する実際のコマンド。
keyword	デコーディング操作で使用するキーワード。
framework	DFM で実行可能なすべての Jython スクリプトに渡す Framework オブジェクト。
timeout	コマンドのタイムアウト。
waitForTimeout	タイムアウトを超えた場合にクライアントが待機するかどうかを指定します。
useSudo	sudo を使用するかどうかを指定します（UNIX マシン・クライアントのみ）。
language	言語を自動検出せず，直接指定できるようにします。

getCharsetName メソッド

このメソッドは，直近に使用した文字セットの名前を返します。

useCharset メソッド

このメソッドは，ShellUtils インスタンスに文字セットを設定します。ShellUtils インスタンスはこの文字セットを最初のデータ・デコーディングに使用します。

引数

名前	説明
charsetName	文字セットの名前（windows-1251，UTF-8 など）。

89 ページの「getCharsetName メソッド」も参照してください。

getLanguageBundle メソッド

適切なリソース・バンドルを取得するために使用するメソッドです。次の API の代わりになるものです。

```
Framework.getEnvironmentInformation().getBundle()
```

引数

名前	説明
baseName	言語のサフィックスのないバンドルの名前（langNetwork など）。
language	Language オブジェクト。ここで, ShellUtils.osLanguage を渡します。
framework	DFM で実行可能なすべての Jython スクリプトに渡す，一般的な Framework オブジェクト。

osLanguage フィールド

このフィールドには，言語を表すオブジェクトが格納されます。

ディスカバリ・アナライザを使った作業

ディスカバリ・アナライザ・ツールは，パッケージ，スクリプト，またはその他のコンテンツの開発時にデバッグ目的で利用するツールです。このツールは，リモート宛先に対してジョブを実行し，情報，警告，エラーの詳細や，CI の検出結果を含むログを返します。

結果は常に UI に報告されるわけではないことに注意してください。これは，結果が 2 つの方法で報告され，そのうちの一方のみがサポートされるためです。また，通信ログは Eclipse からはサポートされません。

Eclipse からツールを実行する場合は，**DiscoveryProbe.properties** ファイル（**C:¥hp¥UCMDB¥DataFlowProbe¥conf¥DiscoveryProbe.properties**）で次のパラメータを **true** に設定する必要があります。

```
appilog.agent.local.discoveryAnalyzerFromEclipse = true
```

詳細については，99 ページの「Eclipse からのディスカバリ・アナライザの実行」を参照してください。

その他の場合（ツールが **cmd** ファイルから実行されるか，プローブの実行中に実行される場合）ではすべて，このフラグは **false** に設定する必要があります。

```
appilog.agent.local.discoveryAnalyzerFromEclipse = false
```

タスクとレコード

タスク・ファイルには，実行対象のタスクに関するデータが含まれます。タスクは，ジョブの名前や，トリガ CI を定義する必須パラメータ（リモート宛先アドレスなど）などの情報で構成されています。

レコード・ファイルには，タスク情報と個々の実行の結果，つまりプローブまたはディスカバリ・アナライザ（モジュールがタスクを実行した方）とリモート宛先の間の詳細な通信（応答を含む）が含まれます。

タスク・ファイルで定義したタスクはリモート宛先に対して実行できるのに対して，レコード・ファイル（特定の実行に関する特別なデータが含まれるファイル）で定義したタスクは実行と再生（レコード・ファイルに記述されたものと同じ実行の再現）が可能です。

ログ

ログは，最新の実行に関する次のような情報を提供します。

- **一般ログ** : このログには，実行中に発生したすべての情報データ，エラー，および警告が含まれます。
- **通信ログ** : このログには，ディスカバリ・アナライザとリモート宛先間の詳細な通信（応答を含む）が含まれます。実行後，ログはレコード・ファイルとして保存できます。
- **結果ログ** : 検出された CI のリストが表示されます。各 CI が表示される時間は，アダプタとスクリプトの設計に応じて異なります。

すべてのログをまとめて保存することも，各ログを別々に保存することもできます。すべてのログを保存すると，ログはまとめられて 1 つの名前で保存されます。

レコード・ファイルを再生すると，実行時間のみが異なる，同じデータが通信ログに表示されます。

制限事項 : 通信ログおよび結果ログは，ディスカバリ・アナライザが Eclipse を通じて実行されている場合，使用できません。

本項には，次の手順が含まれています。

➤ 97 ページの「タスク結果のサーバへの送信」

➤ 98 ページの「設定のインポート」

➤ 98 ページの「ブレークポイント」

1 前提条件

➤ プローブがインストールされている必要があります（ディスカバリ・アナライザは，プローブのインストール・プロセスの一部としてインストールされ，リソースをプローブと共有します）。

➤ ディスカバリ・アナライザでの作業中にプローブを実行する必要はありません。

ただし，プローブがすでに UCMDB サーバに対して実行されている場合，必要なリソースはすべて，すでにファイル・システムにダウンロードされています。プローブがまだ実行されていない場合は，ディスカバリ・アナライザで必要なリソースを［Settings］メニューからアップロードできます。詳細については，98 ページの「設定のインポート」を参照してください。

➤ CMDB サーバがインストールされている必要はありません。

2 ディスカバリ・アナライザへのアクセス

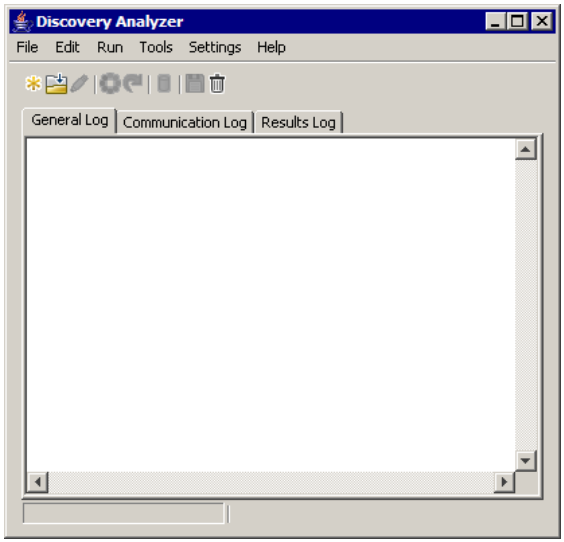
ディスカバリ・アナライザには次のいずれかの方法でアクセスできます。

➤ Eclipse で作業している場合：

プローブのインストールには **C:¥hp¥UCMDB¥DataFlowProbe¥tools¥discoveryAnalyzerWorkspace** にある標準設定の Eclipse ワークスペースが含まれます。このワークスペースには，ディスカバリ・アナライザを起動するための Jython スクリプト（**startDiscoveryAnalyzerScript.py**），およびすべての DFM スクリプトへのリンクが含まれます。この方法でツールを起動すると，デバッグ目的で Jython スクリプト内にブレークポイントを配置できます。

➤ **C:¥hp¥UCMDB¥DataFlowProbe¥tools¥discoveryAnalyzer.cmd** フォルダにあるファイルを直接，ダブルクリックします。詳細については，次の項を参照してください。

［Discovery Analyzer］ウィンドウが開きます。



3 タスクの定義

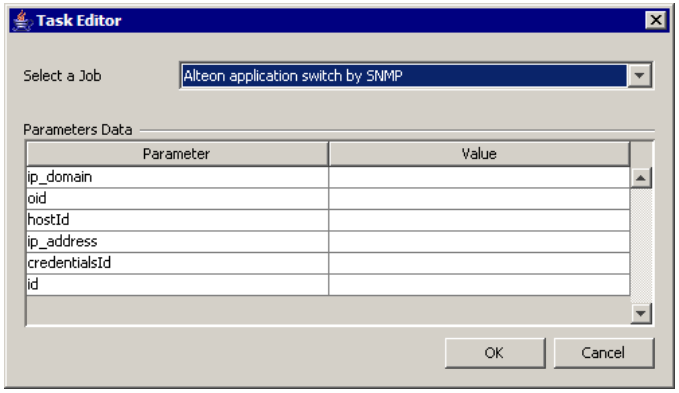
タスクは，次のいずれかの方法で定義します。

- 新しいタスクを定義する。詳細については，95 ページの「新しいタスクの定義」を参照してください。
- レコード・ファイルからタスクをインポートする。詳細については，96 ページの「レコードの取得」を参照してください。
- 保存したタスクをタスク・ファイルからインポートする。詳細については，96 ページの「タスク・ファイルを開く」を参照してください。
- プローブの内部データベースからジョブを取得する。詳細については，96 ページの「データベースからのタスクのインポート」を参照してください。

4 新しいタスクの定義

a ［**New Task**］ボタン をクリックして，タスク・エディタを表示します。

タスク・エディタが開き，ファイル・システムに現在存在するジョブのリストが表示されます。このリストは，プローブがサーバからタスクを受信するたびに，または［Settings］メニューからパッケージを手動で展開するたびに更新されます。



b ジョブを選択します。

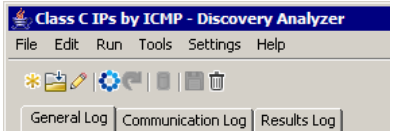
c すべてのパラメータの値を入力します。

ここに表示されるパラメータは，DFM アダプタ・パラメータです。このパラメータは，［パターン シグネチャ］タブの［ディスカバリ パターン パラメータ］表示枠で表示できます。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［アダプタ パラメータ］表示枠」を参照してください。

フィールドは，すべて必須フィールドです（ジョブのスクリプトでフィールドを空白にする必要がある場合を除きます）。

ID または資格情報 ID の入力値を必要とするパラメータの場合，ランダムに作成した ID を使用できます。Value ボックスを右クリックし，［**Generate random CMDB ID**］または［**Credential Chooser**］を選択します。

タスクがアクティブになり，開かれたタスクの名前がタイトル・バーに表示されます。



d タスクを定義する手順を続行します。詳細については，97 ページの「タスクとログの保存」を参照してください。

5 レコードの取得

特定の実行に関するデータが含まれるレコード・ファイルを開いて，タスクを定義できます。タスクをこの方法で定義すると，再生オプションを選択して特定の実行を再現できます（タスクを再生すると，応答は，リモート宛先ではなく，レコード・ファイルに保存されたデータから受信されます）。

［**File**］＞［**Open Record**］を選択します。レコードを保存したフォルダを参照します。レコードがアクティブになり，タスクの名前がタイトル・バーに表示されます。

レコード・ファイルの取得の詳細については，109 ページの「DFM コードの記録」を参照してください。

6 タスク・ファイルを開く

タスク・ファイルからタスクを定義できます。［**File**］＞［**Open Task**］を選択します。

7 データベースからのタスクのインポート

プローブがすでに実行されており，プローブの内部データベースにアクティブなタスクがある場合，プローブ・データベースからタスクを取得できます。パラメータ値を使用して，タスクを定義できます。

a ［**File**］＞［**Import Task from Probe DB**］を選択します。

b 開いたダイアログ・ボックスで，実行するタスクを選択し，［**OK**］をクリックします。

c タスクを定義する手順を続行します。詳細については，97 ページの「タスクとログの保存」を参照してください。

8 タスクの編集

タスクを定義すると，タスク（またはファイル）の名前がタイトル・バーに表示されます。これで，ファイルを編集できます。

a ［**Edit**］＞［**Edit Task**］を選択します。

b タスクに変更を加え，［**OK**］をクリックします。

9 タスクとログの保存

タスク・パラメータを保存できます。［**File**］ ＞ ［**Save Task**］を選択します。

タスクの実行後に限り，次のオプションを選択できます。

- ➤ タスクのレコードを保存します。タスク・パラメータとタスク実行の結果を保存できます。［**File**］ ＞ ［**Save Record**］を選択します。
- ➤ タスクのログを保存できます。［**File**］＞［**Save General Log**］を選択します。
- ➤ 結果を保存します。［**File**］ ＞ ［**Save Result**］を選択します。

10 タスクの実行

手順の次の手順では，作成したタスクを実行します。

a リモート宛先に対してのみタスクを実行するには，［**Run Task** ］ボタンをクリックします。

ディスカバリ・アナライザによってジョブが実行され，3 つのログ・ファイル（**一般**，**通信**，**結果**）に情報が表示されます。

b ログ・ファイルはまとめて，または別々に保存できます。［**File**］の後に，［ **Save General Log**］，［**Save Record**］，［**Save Results**］，［**Save All Logs**］のいずれかを選択します。ログ・ファイルの詳細については，92 ページの「ログ」を参照してください。

c タスクをレコード・ファイルから取得する場合，［**Playback**］ボタンをクリックすると，レコード・ファイルに記述された実行を再現できます。表示される通信ログは同じですが，実行時間は更新されます。

11 タスク結果のサーバへの送信

タスクの実行が終了し，結果が生成された場合（つまり，［結果ログ］タブに検出された CI のリストが表示された場合），結果を UCMDB サーバに送信できます。これは，サーバの停止中にスクリプトのテストを行い，後から結果を送信する場合などに便利です。

注 : 結果の送信先にできるのは，ディスカバリ・アナライザと同じマシンにインストールされているプローブからタスクを受信する UCMDB サーバのみです。

---

12 設定のインポート

プローブがまだ実行されていない場合，ディスカバリ・アナライザで必要なファイルをインポートできます。［Settings］メニューにアクセスし，**domainScopeDocument.xml** または **domainScopeDocument.bin** をインポートします。**.bin** ファイルをインポートする場合，**key.bin** もインポートする必要があります。［**Settings**］＞［**Import packages**］を選択することで，必要なパッケージ（スクリプト，アダプタなど）をデプロイできます。

13 ブレークポイント

Python スクリプトからディスカバリ・アナライザを実行する場合，スクリプトにブレークポイントを追加できます。

14 Eclipse の設定

Jython スクリプトのデバッグ・モードでの実行の詳細については，99 ページの「Eclipse からのディスカバリ・アナライザの実行」を参照してください。

Eclipse からのディスカバリ・アナライザの実行

このタスクでは，Jython スクリプトをデバッグ・モードで実行してジョブ・スレッド，トリガ CI，および結果をわかりやすく表示できるように，Eclipse を設定する方法について説明します。

本項には，次の手順が含まれています。

- 99 ページの「前提条件」
- 100 ページの「Eclipse の展開と開始」
- 100 ページの「標準設定のワークスペースの設定」
- 103 ページの「ディスカバリ・アナライザ・ワークスペースの設定」
- 106 ページの「クラスパスとインタープリタの設定」
- 109 ページの「ディスカバリ・アナライザの実行」

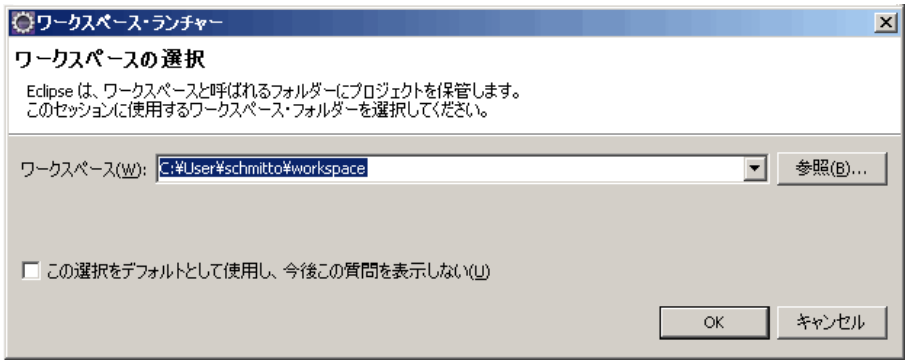
1 前提条件

- コンピュータに Eclipse の最新バージョンをインストールします。このアプリケーションは，www.eclipse.org （英語サイト）で入手できます。
- Data Flow Probe が同じコンピュータにインストールされていることを確認します。
- **DiscoveryProbe.properties** ファイルの **appilog.agent.local.discoveryAnalyzerFromEclipse** パラメータが **true** に設定されていることを確認します。

2 Eclipse の展開と開始

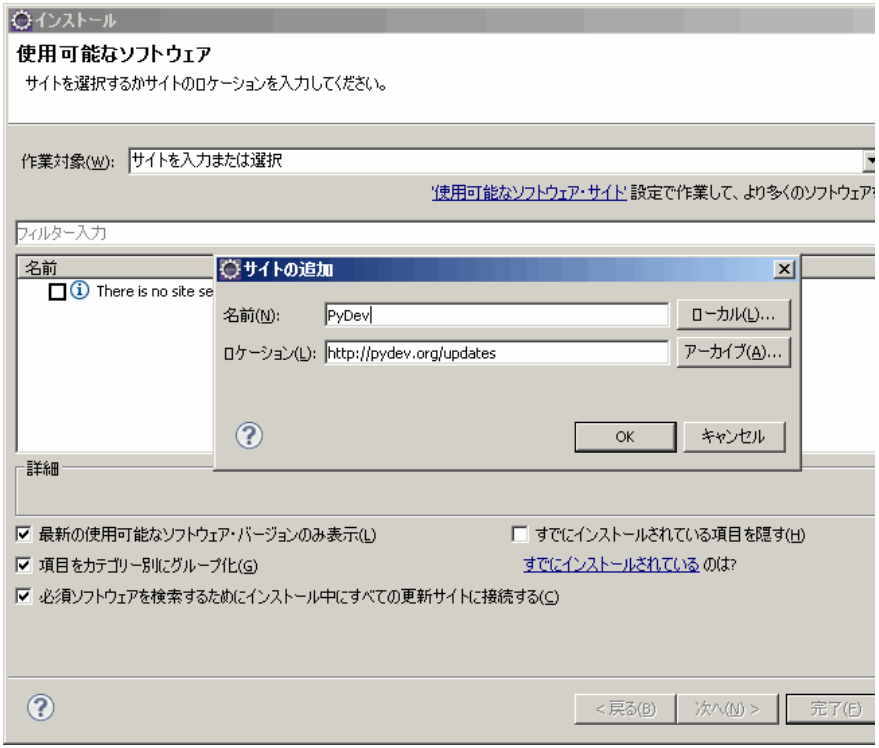
3 標準設定のワークスペースの設定

Eclipse がすべてのプロジェクトと関連データを保存し，格納する標準設定のワークスペースを設定します。



4 PyDev Extensions の設定

a ［**ヘルプ**］＞［**新規ソフトウェアのインストール**］にアクセスし，［**追加**］をクリックして PyDev プラグインの名前を入力し，［ロケーション］フィールドに pydev をダウンロードできるサイトの URL（**http://pydev.org/updates**）を追加します。［**OK**］をクリックします。



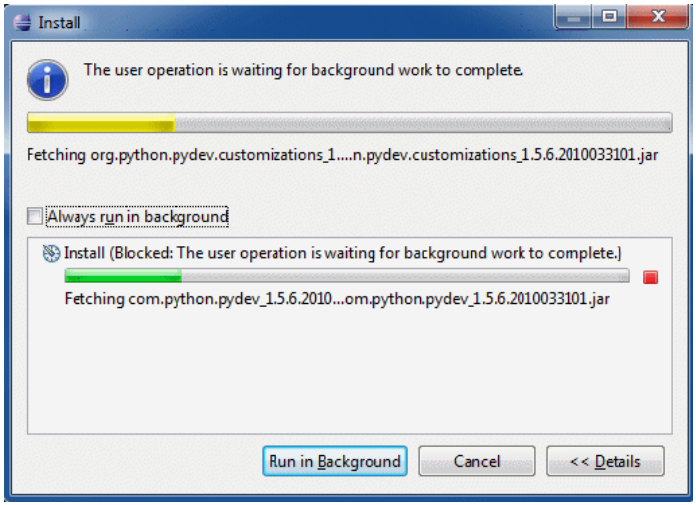
注 : 現在では PyDev Extensions はオープン・ソースであるため，PyDev および PyDev Extensions は 1 つのプラグインに結合されています。追加情報は http://pydev.org (英語サイト) をご覧ください。

---

b 開いたウィンドウで，**Pydev** を選択します。2 つ目のプラグインは，タスク関連の UI のためのプラグインです。［**次へ**］をクリックして，インストールの詳細を確認し，再度［**次へ**］をクリックします。

c ライセンス契約を承認して，［**次へ**］をクリックします。

d Pydev がインストールされています。無署名のコンテンツをインストールするかどうか尋ねられた場合は，［**OK**］をクリックして承諾します。

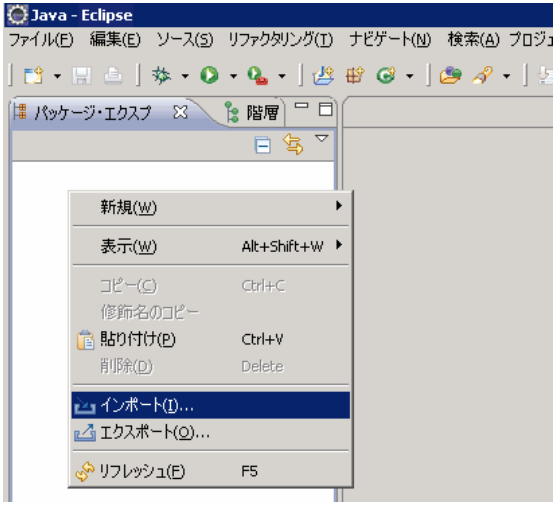


e Eclipse を再起動します。

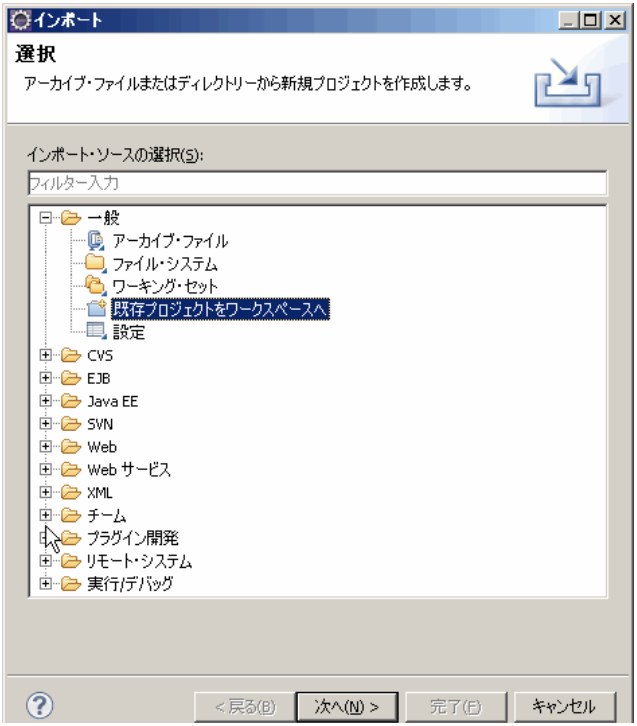
これで PyDev が Eclipse IDE にインストールされました。Eclipse の表示が新しくなり，IDE が Python スクリプト（テキスト強調表示，追加の設定オプションなど）を解釈できるようになります。

5 ディスカバリ・アナライザ・ワークスペースの設定

a 必要なファイルをインポートします。Package Explorer の白色の領域を右クリックして［**インポート**］をクリックし，プローブのインストールに含まれている事前定義済みの **discoveryAnalyzerWorkspace** をインポートします。



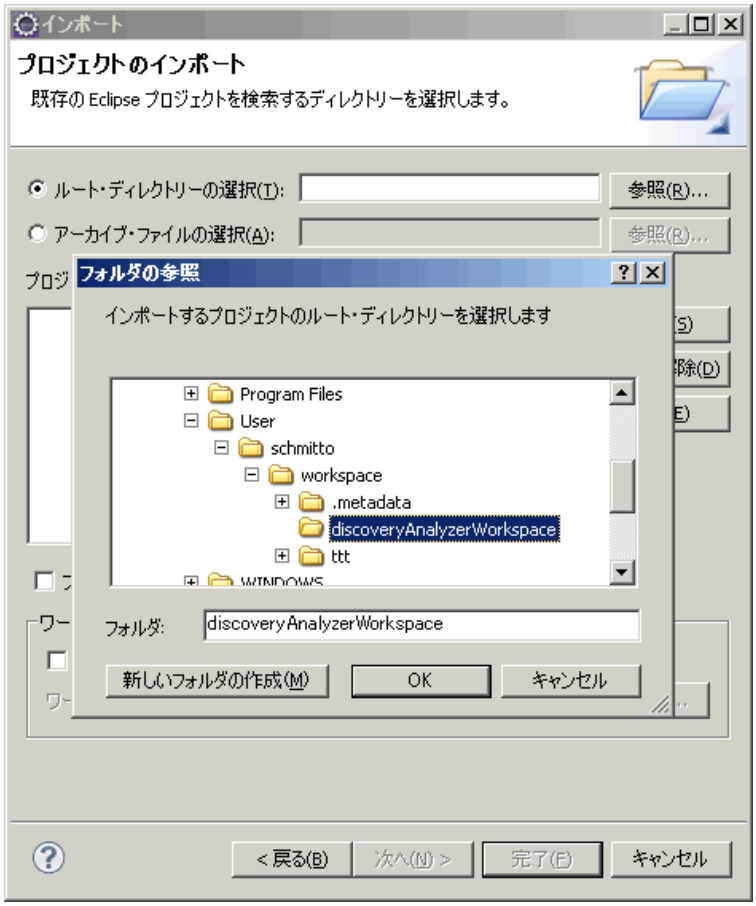
b ［**一般**］の下の［**既存プロジェクトをワークスペースへ**］を選択して，プロジェクトを Eclipse ワークスペースにインポートします。

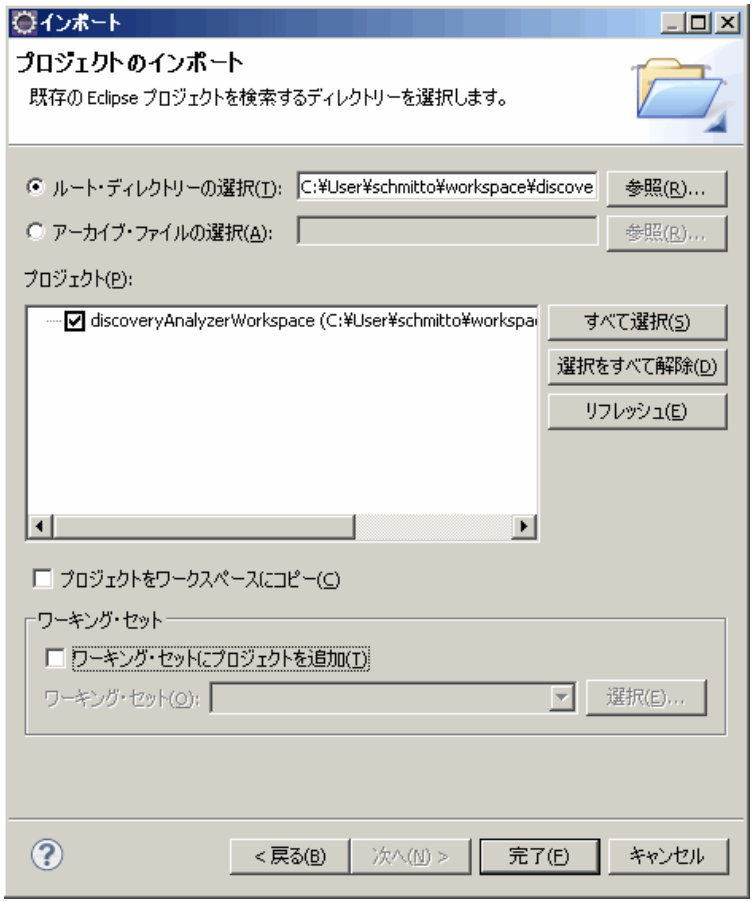


c ［**ルート・ディレクトリーの選択 :**］の下でアナライザ・ワークスペースを選択します。通常は **C:¥hp¥UCMDB¥DataFlowProbe¥tools¥discoveryAnalyzerWorkspace** の下に置かれています。

d ［**プロジェクトをワークスペースにコピー**］を選択して，既存のワークスペースの実際のコピーを作成します。これは重要な手順です。こうすることで，失敗した場合，元の **discoveryAnalyserWorkspace** を再インポートできるようにします。

e ［**完了**］をクリックして，インポートを開始します。





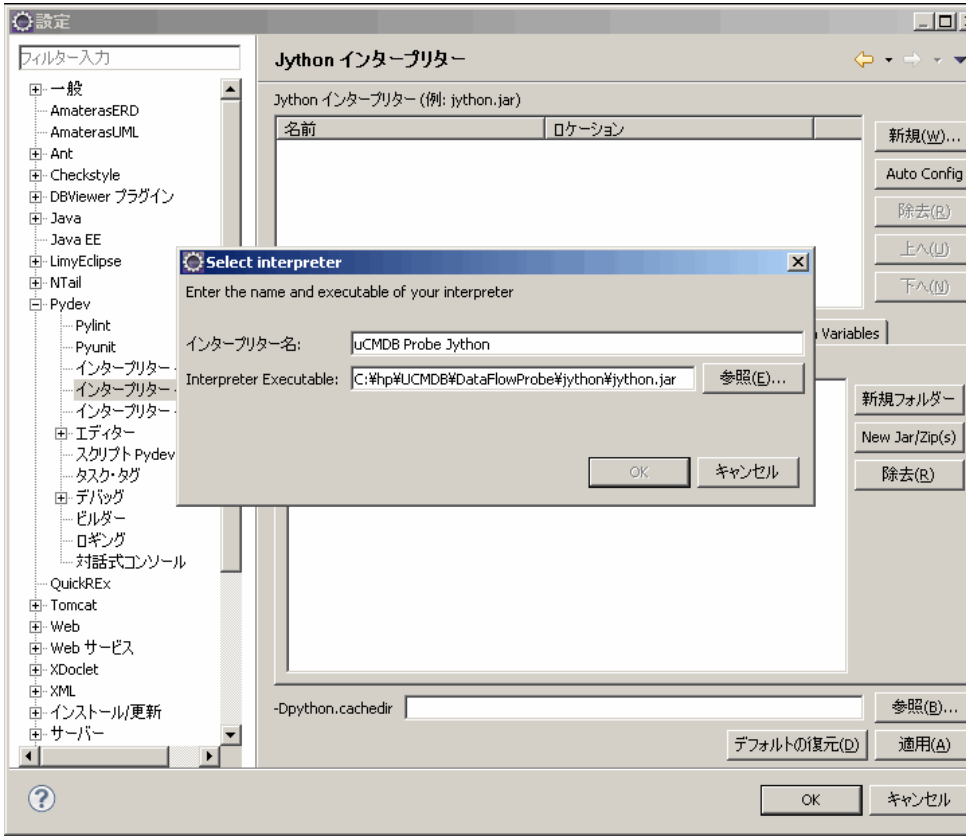
6 クラスパスとインタープリタの設定

a **discoveryAnalyzerWorkspace** を右クリックして［**プロパティ**］を選択し，プロジェクト固有の設定を表示します。

b ［**Pydev**］＞［**Interpreter/Grammar**］に移動し，［**Please configure an interpreter in the related preferences before proceeding**］をクリックします。

この手順では，スクリプトが別の Jython バージョンで解釈されないように，プローブが使用しているものと同じ Jython インタープリタを設定します。

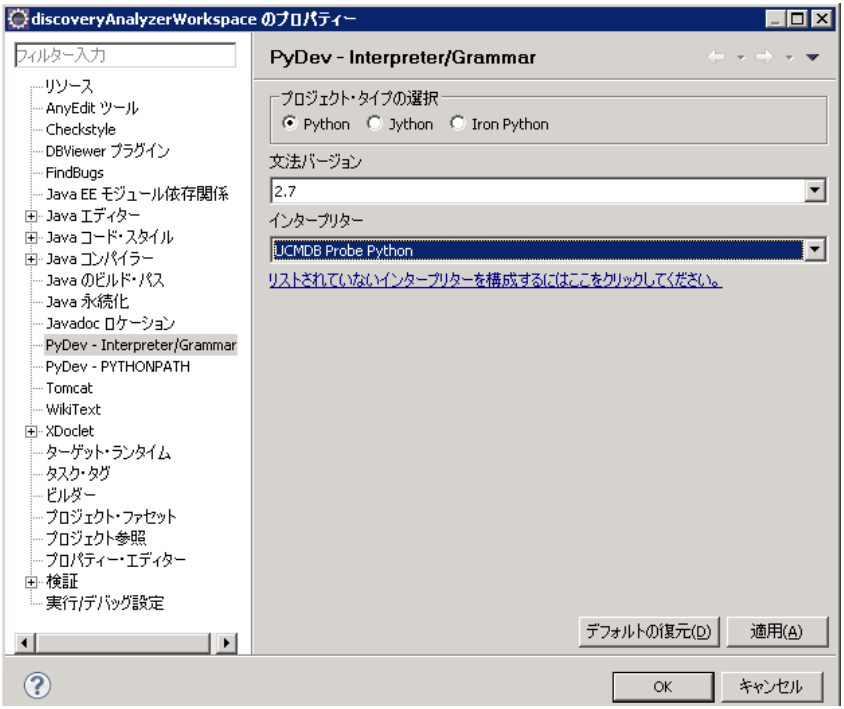
c ［新規］をクリックし，インタープリタの名前を入力して，**C:¥hp¥UCMDB¥DataFlowProbe¥jython¥jython.jar** フォルダからファイルを選択します。



d ［**OK**］をクリックします。Python システム・パスにインポートするフォルダを選択するように促すウィンドウが表示された場合は，何も変更せずに（**C:¥hp¥UCMDB¥DataFlowProbe¥jython** および **C:¥hp¥UCMDB¥DataFlowProbe¥jython¥lib** のまま）［**OK**］をクリックします。

e ［**適用**］をクリックし，［**OK**］をクリックします。

f ［**インタープリター**］をクリックして，作成したばかりのインタープリタを選択します。



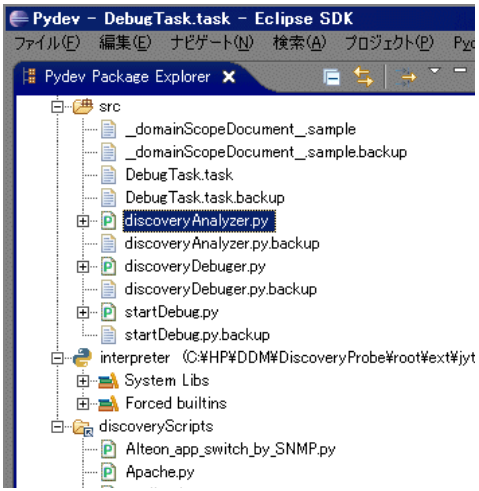
g ［**適用**］をクリックし，［**OK**］をクリックします。

これで Jython インタープリタはプローブが使用しているものと同じインタープリタになりました。

7 ディスカバリ・アナライザの実行

a デバッグ対象の Jython スクリプトにブレークポイントを追加します。

b Discovery Analyzer を使用するには，**discoveryAnalyzerWorkspace** プロジェクトの **startDiscoveryAnalyzerScript.py** ファイルを選択します。このファイルを右クリックして，［**デバッグ**］＞［**Jython 実行**］を選択します。



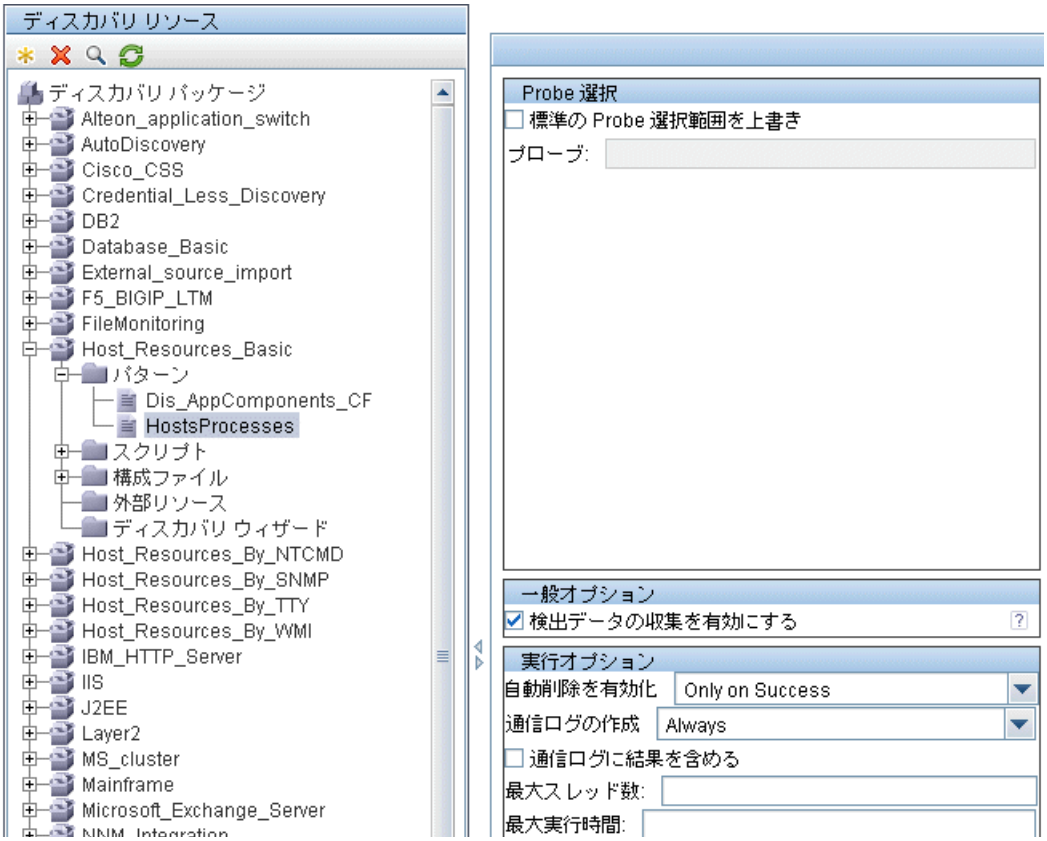
DFM コードの記録

デバッグやコードのテストなどを行うときは，すべてのパラメータを含む実行全体を記録すると非常に便利です。このタスクでは，関連するすべての変数を使って実行全体を記録する方法について説明します。また，通常はデバッグ・レベルでもログ・ファイルに出力されない追加のデバッグ情報を参照することもできます。

DFM コードを記録するには，次の手順で行います。

1 ［**データ フロー管理**］＞［**ディスカバリ コントロール パネル**］に移動します。実行をログに記録する必要があるジョブを右クリックし，［**アダプタの編集**］を選択してアダプタ管理アプリケーションを開きます。

2 ［**パターン管理**］タブの［実行オプション］表示枠を見つけます。



3 ［**通信ログの作成**］ボックスを［**常時**］に変更します。ログ・オプションの設定の詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［実行オプション］表示枠」を参照してください。

次の例は，Host Connection by Shell ジョブを実行し，［**通信ログの作成**］ボックスを［**常時**］または［**失敗時**］に設定したときの XML ログ・ファイルです。

ジョブ名　　トリガ CI データ

```
- <execution jobId="Host Connection by Shell" destinationid="0e9787433d65e4a68839bfa8b224c92d">
  - <destination>
      <destinationData name="ip_domain">DefaultDomain</destinationData>
      <destinationData name="hostId" />
      <destinationData name="ip_address">16.59.63.34</destinationData>
      <destinationData name="id">0e9787433d65e4a68839bfa8b224c92d</destinationData>
    </destination>
```

次の例は，メッセージとスタックトレース・パラメータを示します。

スタックトレース

```
- <exec start="18:41:55" duration="2062" type="ssh" credentialsId="f464999bdfe5a1e1407b479b6f730d5b">
    <cmd>[CDATA: client_connect]</cmd>
    <result IS_NULL="Y" />
  - <error class="com.hp.ucmdb.discovery.probe.services.dynamic.agents.SSHAgentException">
      <message>[CDATA: Failed to connect: Error connecting: Connection refused: connect]</message>
    - <stacktrace>
        <frame class="com.hp.ucmdb.discovery.probe.services.dynamic.agents.SSHAgent" method="connect" file
        <frame class="com.hp.ucmdb.discovery.probe.clients.shell.SSHClient" method="createWrapper" file="SSH
        <frame class="com.hp.ucmdb.discovery.probe.clients.BaseClient" method="initPrivate" file="BaseClient.ja
```

参照先

Jython のライブラリとユーティリティ

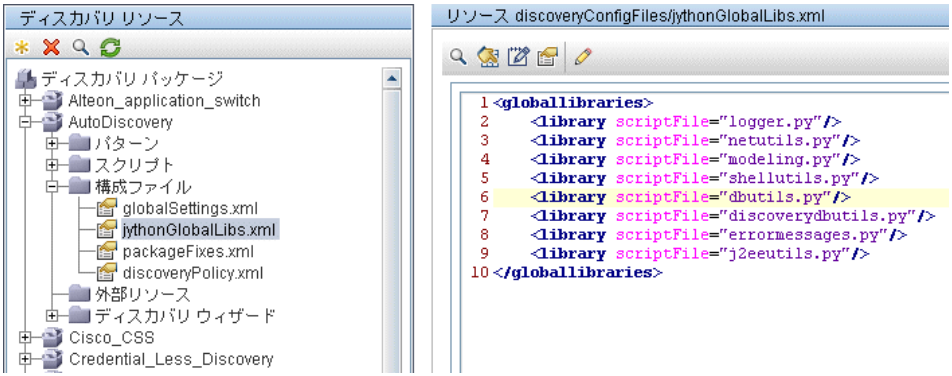
アダプタでは，いくつかのユーティリティ・スクリプトが広く使用されます。これらのスクリプトは AutoDiscovery パッケージの一部で，プローブにダウンロードされたほかのスクリプトとともに **C:¥hp¥UCMDB¥DataFlowProbe¥runtime¥probeManager¥discoveryScripts** の下に置かれています。

注 : discoveryScript フォルダは, Probe が動作を開始したときに動的に作成されます。

ユーティリティ・スクリプトを使用するには，スクリプトのインポート・セクションに以下の import 行を追加します。

```
import <script name>
```

AutoDiscovery Python ライブラリには，Jython ユーティリティ・スクリプトが含まれています。これらのライブラリ・スクリプトは，DFM の外部ライブラリとみなされ，（**構成ファイル・**フォルダにある）jythonGlobalLibs.xml ファイルで定義されています。



jythonGlobalLibs.xml ファイルに表示される各スクリプトは，標準設定ではプローブの起動時に読み込まれるため，それらをアダプタ定義で明示的に使用する必要はありません。

本項の内容

logger.py

logger.py スクリプトには，エラー報告用のログ・ユーティリティとヘルパー関数が含まれています。そのデバッグ API，情報 API，およびエラー API を呼び出して，ログ・ファイルに書き込むことができます。ログ・メッセージは **C:¥hp¥UCMDB¥DataFlowProbe¥runtime¥log** に記録されます。

メッセージは，**C:¥hp¥UCMDB¥DataFlowProbe¥conf¥log¥probeMgrLog4j.properties** ファイルの PATTERNS_DEBUG アペンダに定義されているデバッグ・レベルに応じて，ログ・ファイルに入力されます（標準設定のレベルは DEBUG です）。詳細については，118 ページの「エラーの重大度レベル」を参照してください。

```
##########################################################################
################ PATTERNS_DEBUG log ####################
##########################################################################
log4j.category.PATTERNS_DEBUG=DEBUG, PATTERNS_DEBUG
log4j.appender.PATTERNS_DEBUG=org.apache.log4j.RollingFileAppender
log4j.appender.PATTERNS_DEBUG.File=C:\hp\UCMDB\DataFlowProbe\runtime\log/pr
obeMgr-patternsDebug.log
log4j.appender.PATTERNS_DEBUG.Append=true
log4j.appender.PATTERNS_DEBUG.MaxFileSize=15MB
log4j.appender.PATTERNS_DEBUG.Threshold=DEBUG
log4j.appender.PATTERNS_DEBUG.MaxBackupIndex=10
log4j.appender.PATTERNS_DEBUG.layout=org.apache.log4j.PatternLayout
log4j.appender.PATTERNS_DEBUG.layout.ConversionPattern=<%d> [%-5p] [%t] -
%m%n
log4j.appender.PATTERNS_DEBUG.encoding=UTF-8
```

情報メッセージとエラー・メッセージは，コマンド・プロンプト・コンソールに表示されます。

次の 2 つの API セットがあります。

- logger.<debug/info/warn/error>
- logger.<debugException/infoException/warnException/errorException>

1 つ目のセットは，該当するログ・レベルでそのすべての文字列引数を連結したものを発行します。2 つ目のセットは，連結した文字列とともに，最後にスローされた例外のスタック・トレースを発行して，詳細な情報を提供します。次に例を示します。

```
logger.debug('found the result')
logger.errorException('Error in discovery')
```

modeling.py

modeling.py スクリプトには，ホスト，IP，プロセス CI などを作成するための API が含まれています。これらの API を使って，共通のオブジェクトを作成し，コードを読みやすくできます。たとえば，

```
ipOSH= modeling.createIpOSH(ip)
host = modeling.createHostOSH(ip_address)
member1 = modeling.createLinkOSH('member', ipOSH, networkOSH)
```

netutils.py

netutils.py ライブラリは，オペレーティング・システム名の取得，MAC アドレスの有効性の確認，IP アドレスの有効性の確認など，ネットワークや TCP の情報を取得するために使用されます。たとえば，

```
dnsName = netutils.getHostName(ip, ip)
isValidIp = netutils.isValidIp(ip_address)
address = netutils.getHostAddress(hostName)
```

shellutils.py

shellutils.py ライブラリは，シェル・コマンドを実行して，実行されたコマンドの終了ステータスを取得するための API を提供します。また，その終了ステータスに基づいて複数のコマンドを実行できます。このライブラリは，シェル・クライアントによって初期化され，そのクライアントを使ってコマンドを実行し，結果を取得します。たとえば，

```
ttyClient = clientFactory.createClient(Props)
clientShUtils = shellutils.ShellUtils(ttyClient)
if (clientShUtils.isWinOs()):
        logger.debug ('discovering Windows..')
```

エラー記述の表記規則

- 各エラーは，エラー・メッセージ・コードと引数の配列（**int, String[]**）で識別されます。個々のエラーは，メッセージ・コードと引数の配列の組み合わせにより定義されます。パラメータの配列は null である場合もあります。
- 各エラー・コードは，固定された文字列である**簡略メッセージ**とゼロ個以上の引数を含むテンプレート文字列である**詳細メッセージ**にマップされています。テンプレート内の引数の数と実際のパラメータの数は一致するものと想定されています。

エラー・メッセージ・コードの例 :

10234 を，次のような簡略メッセージのエラーであるとします。

```
Connection Error
```

詳細メッセージは次のとおりです。

```
Could not connect via {0} protocol due to timeout of {1} msec
```

ここで

{0} = 最初の引数 : プロトコル名

{1} = 2 番目の引数 : タイムアウトの長さ（ミリ秒）

本項の内容：

プロパティ・ファイルのコンテンツ

各エラー・メッセージ・コードについて，プロパティ・ファイルには 2 つのキーが含まれています。たとえば，エラー 45 の場合，次のようになります。

- **DDM_ERROR_MESSAGE_SHORT_45**: エラーの短い説明。
- **DDM_ERROR_MESSAGE_LONG_45**: エラーの長い説明（**{0}**,**{1}** などのパラメータを含む可能性があります）。

エラー・メッセージのプロパティ・ファイル

プロパティ・ファイルには，エラー・メッセージ・コードと 2 つのメッセージ（簡略および詳細）のマップがあります。

プロパティ・ファイルがデプロイされると，データは既存のデータと結合されます（つまり，新しいメッセージ・コードは，古いメッセージ・コードを上書きするようにして追加されます）。

インフラストラクチャのプロパティ・ファイルは，**DiscoveryInfra** パッケージの一部です。

命名規則

- 標準設定のロケールの場合 : **<file name>.properties.errors**
- 特定のロケールの場合 : **<file name>_xx.properties.errors**

  ここで，**xx** はロケールです（たとえば **infraerr_fr.properties.errors** または **infraerr_en_us.properties.errors**）。

予約済みのエラー・メッセージ・コード

0-99: インフラ数

100: 未分類コンテンツ

101 - ???: コンテンツ数

未分類コンテンツ・エラー

後退をまねくことなく古いコンテンツをサポートするために，アプリケーションおよび SDK 関連メソッドはメッセージ・コード 100（つまり，未分類スクリプト・エラー）のエラーをそれぞれ別の方法で処理します。

これらの未分類のエラーはメッセージ・コードによりグループ化されていません（つまり，これらは同じタイプのエラーであるとはみなされません）が，メッセージのコンテンツによりグループ化されています。そのため，スクリプトが（メッセージ文字列を含むが，エラー・コードを含まない）古い廃止済みのメソッドによりエラーを報告した場合，すべてのメッセージが同じエラー・コードを受け取りますが，アプリケーションまたは SDK 関連メソッド内で異なるエラーとして別のメッセージが表示されます。

Framework での変更

（com.hp.ucmdb.discovery.library.execution.BaseFramework）

次のメソッドがインタフェースに追加されています。

- void reportError(int msgCode, String[] params);
- void reportWarning(int msgCode, String[] params);
- void reportFatal(int msgCode, String[] params);

次の古いメソッドは，下位互換性のために現在もサポートされていますが，廃止としてマークされています。

- void reportError(String message);
- void reportWarning (String message);
- void reportFatal (String message);

エラーの重大度レベル

アダプタがトリガ CI に対する実行を終了すると，ステータスが返されます。エラーや警告がまったく報告されない場合，ステータスは **Success（成功）**と表示されます。

次に，重大度レベルを範囲が最も狭いものから最も広いものの順に示します。

- **Fatal（致命的）：**このレベルでは，インフラストラクチャに関する問題，DLL ファイルの欠落，例外など，深刻なエラーが報告されます。
- **Error（エラー）：**このレベルでは，DFM のデータ取得を妨げる問題が報告されます。このエラーは通常，何らかの対応（タイムアウトの延長，範囲の変更，パラメータの変更，ほかのユーザの資格情報の追加など）を必要とするため，よく調べるようにしてください。
- **Warning（警告）：**実行は成功で，深刻ではないが認識しておくべき問題がある場合，重大度は **Warning（警告）**となります。より詳細なデバッグ・セッションを開始する前に，CI を調べてデータが欠落していないか確認する必要があります。**警告**には，インストールされているエージェントやリモート・ホストの欠落に関するメッセージや，無効なデータが原因で属性が正しく計算されなかったというメッセージが含まれている可能性があります。
- **Success（成功）：**トリガ CI は正常に実行されました。

4

汎用データベース・アダプタの開発

本章の内容

概念

タスク

参照先

➤ アダプタ構成ファイル（167 ページ）
➤ 用意済みのコンバータ（184 ページ）
➤ プラグイン（188 ページ）
➤ 設定例（188 ページ）
➤ アダプタ・ログ・ファイル（200 ページ）
➤ 外部参照先（202 ページ）

トラブルシューティングと制限事項（202 ページ）

概念

汎用データベース・アダプタの概要

汎用データベース・アダプタ・プラットフォームの目的は，リレーショナル・データベース管理システム（RDBMS）との統合が可能なアダプタを作成し，データベースに対して TQL クエリとポピュレーション・ジョブを実行することです。汎用データベース・アダプタでサポートしている RDBMS は，Oracle，Microsoft SQL Server，および MySQL です。

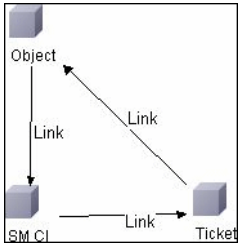
このバージョンのデータベース・アダプタ実装は，JPA（Java Persistence API）と，パーシステンス・プロバイダとしての Hibernate ORM ライブラリに基づいています。

サポートされていない TQL クエリ

汎用データベース・アダプタのみで計算された TQL クエリには，次の制限があります。

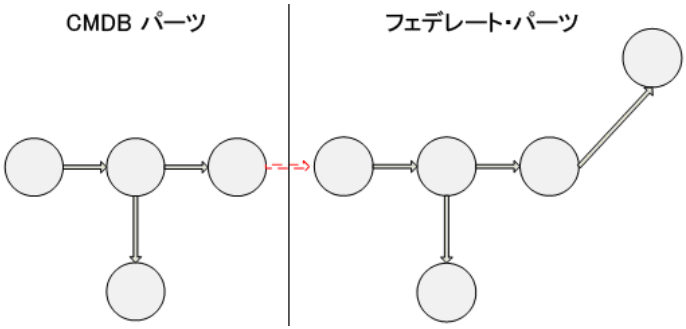
- サブグラフはサポートされていません。
- 複合関係はサポートされていません。
- サイクルやサイクル・パーツはサポートされていません。

  次の TQL クエリはサイクルの一例です。



- 関数レイアウトがサポートされていない。
- 0..0 カーディナリティがサポートされていない。

- Join 関係がサポートされていない。
- 修飾子条件がサポートされていない。
- 2 つの CI を接続するには，テーブルまたは外部キー形式の関係が外部データベース・ソースに存在する必要がある。



調整

調整はアダプタ側で TQL 計算の一部として実行されます。調整が行われるようにするには，CMDB 側を調整 CIT というフェデレート・エンティティにマップします。

マッピング : CMDB の各属性がデータ・ソースのカラムにマップされます。

マッピングは直接実行されますが，マッピング・データの変換関数もサポートされています。新しい関数は Java コードで追加できます (lowercase, uppercase など)。これらの関数の目的は，値を変換できるようにすることです（ある形式で CMDB に保存された値と，別の形式でフェデレート・データベースに保存された値)。

注 :

- CMDB と外部データベース・ソースを接続するには，適切な関連付けがそのデータベースに必要です。詳細については，126 ページの「前提条件」を参照してください。
- CMDB id による調整もサポートされています。

JPA プロバイダとしての Hibernate

Hibernate はオブジェクト関連（OR）マッピング・ツールで，数種類のリレーショナル・データベース（Oracle や Microsoft SQL Server など）上のテーブルに Java クラスをマッピングできます。詳細については，203 ページの「機能上の制限事項」を参照してください。

基本マッピングでは，各 Java クラスが単一テーブルにマップされます。詳細なマッピングでは，継承マッピングができます（CMDB データベースで行うことができます)。

サポートされているほかの機能としては，複数テーブルへのクラスのマッピング，コレクションのサポート，1 対 1，1 対多，および多対 1 タイプの関連付けなどがあります。詳細については，125 ページの「関連付け」を参照してください。

そのために，Java クラスを作成する必要はありません。CMDB クラス・モデル CIT からデータベース・テーブルへのマッピングが定義されます。

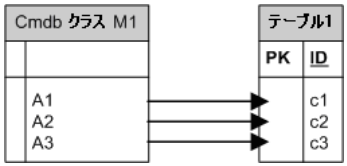
本項の内容 :

オブジェクト関連マッピングの例

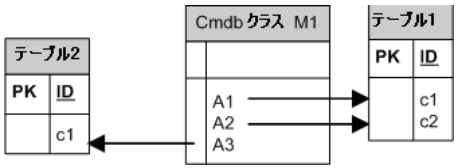
次の例で，オブジェクト関連マッピングについて説明します。

1 つのデータベース・テーブルにマップされた 1 つの CMDB クラスの例 :

クラス M1 と属性 A1，A2，および A3 がテーブル 1 のカラム c1，c2，および c3 にマップされています。つまり，どの M1 インスタンスも，テーブル 1 に一致する行があります。

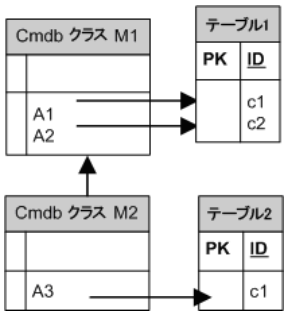


2 つのデータベース・テーブルにマップされた 1 つの CMDB クラスの例 :



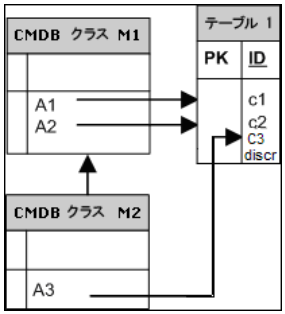
継承の例 :

このケースは CMDB で使用され，各クラスにそれぞれのデータベース・テーブルがあります。



識別子による単一テーブル継承の例 :

クラスの階層全体が単一データベース・テーブルにマップされ，そのカラムはマップされたクラスの全属性のスーパーセットで構成されます。このテーブルには追加カラム（Discriminator）も含まれて，その値は該当エントリにマップされる特定のクラスを示します。



関連付け

関連付けには，1 対多，多対 1，および多対多の 3 つのタイプがあります。異なるデータベース・オブジェクトを接続するには，外部キー・カラム（1 対多の場合）またはマッピング・テーブル（多対多の場合）を使って，これらの関連付けのいずれかを定義する必要があります。

ユーザビリティ

JPA スキーマは非常に広範囲に及ぶので, 定義を簡単にするために簡素化された XML ファイルが用意されています。

この XML ファイルを使用する事例は次のとおりです。フェデレート・データは 1 つのフェデレート・クラスにモデル化されます。このクラスは非フェデレート CMDB クラスと多対 1 の関係になります。また，フェデレート・クラスと非フェデレート・クラス間には，考えられる関係タイプが 1 つしかありません。

タスク

アダプタ作成の準備

このタスクでは，アダプタを作成するために必要な準備について説明します。

このタスクには次の手順が含まれます。

- 126 ページの「前提条件」
- 128 ページの「CI タイプの作成」
- 129 ページの「関係の作成」

1 前提条件

データベースでデータベース・アダプタを使用できるか検証するには，次の点をチェックします。

- 調整クラスとその属性（multinode（マルチノード）ともいう）がデータベースにあるかどうか。たとえば，調整をノード名で実行する場合は，ノード名の記入されたカラムが含まれているテーブルがあるか確認します。調整をノードの cmdb_id に従って実行する場合は，CMDB 内のノードの CMDB ID に一致する CMDB ID を持つカラムがあるか確認します。調整の詳細については，122 ページの「調整」を参照してください。

ID	NAME	IP_ADDRESS
31	BABA	16.59.33.60
33	ext3.devlab.ad	16.59.59.116
46	LABM1MAM15	16.59.58.188
72	cert-3-j2ee	16.59.57.100
102	labm1sun03.devlab.ad	16.59.58.45
114	LABM2PCOE73	16.59.66.79

ID	NAME	IP_ADDRESS
116	CUT	16.59.41.214
117	labm1hp4.devlab.ad	16.59.60.182

➤ 2 つの CIT を関係と関連付けるには，CIT テーブル間に相関データがある必要があります。相関関係は外部キー・カラムまたはマッピング・テーブルによるものになります。たとえば，ノードとチケットを関連付けるには，ノード ID の含まれたチケット・テーブルのカラム，接続するチケット ID の含まれたノード・テーブルのカラム，または end1 がノード ID で，end2 がチケット ID のマッピング・テーブルがある必要があります。相関データの詳細については，123 ページの「JPA プロバイダとしての Hibernate」を参照してください。

次の表に，外部キーの NODE_ID カラムを示します。

NODE_ID	CARD_ID	CARD_TYPE	CARD_NAME
2015	1	シリアル・バス・コントローラ	Intel® 82801EB USB ユニバーサル・ホスト・コントローラ
3581	2	システム	Intel® 631xESB/6321ESB/3100 チップセット LPC
3581	3	ディスプレイ	ATI ES1000
3581	4	基本システム・ペリフェラル	HP ProLiant iLO 2 レガシー・サポート機能

➤ それぞれの CIT は 1 つ以上のテーブルにマップできます。1 つの CIT を複数のテーブルにマップするには，プライマリ・キーがほかのテーブルに存在するプライマリ・テーブルがあり，一意の値カラムがあるかチェックします。

たとえば，チケットは 2 つのテーブル，ticket1 と ticket2 にマップされます。最初のテーブルにはカラム c1 と c2 があり，もう 1 つのテーブルにはカラム c3 と c4 があります。これらを 1 つのテーブルと見なせるようにするには，両方に同じプライマリ・キーを設定する必要があります。あるいは，最初のテーブルのプライマリ・キーをもう 1 つのテーブルのカラムにします。

次の例では，テーブルが CARD_ID という同じプライマリ・キーを共有しています。

CARD_ID	CARD_TYPE	CARD_NAME
1	シリアル・バス・コントローラ	Intel® 82801EB USB ユニバーサル・ホスト・コントローラ
2	システム	Intel® 631xESB/6321ESB/3100 チップセット LPC
3	ディスプレイ	ATI ES1000
4	基本システム・ペリフェラル	HP ProLiant iLO 2 レガシー・サポート機能

CARD_ID	CARD_VENDOR
1	Hewlett-Packard Company
2	（標準 USB ホスト・コントローラ）
3	Hewlett-Packard Company
4	（標準システム・デバイス）
5	Hewlett-Packard Company

2 CI タイプの作成

このステップでは，RDBMS（外部データ・ソース）のデータにマップされるフェデレート CIT を作成します。

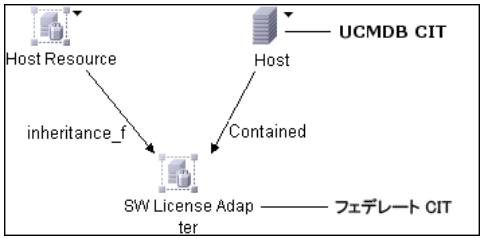
a UCMDB で，CI タイプ・マネージャにアクセスして，新規の CI タイプを作成します。詳細については，『モデリング・ガイド』の「CI タイプの作成」を参照してください。

b CIT に必要な属性（最終アクセス日時，ベンダなど）を追加します。これらは，アダプタによって外部データ・ソースから取得され，CMDB ビューに取り込まれる属性です。

3 関係の作成

この手順では, UCMDB CIT と外部データ・ソースからフェデレートするデータを示す新しい CIT との関係を追加します。

新しい CIT に適切で有効な関係を追加します。詳細については,『モデリング・ガイド』の「関係の追加 / 削除 ダイアログ・ボックス」を参照してください。

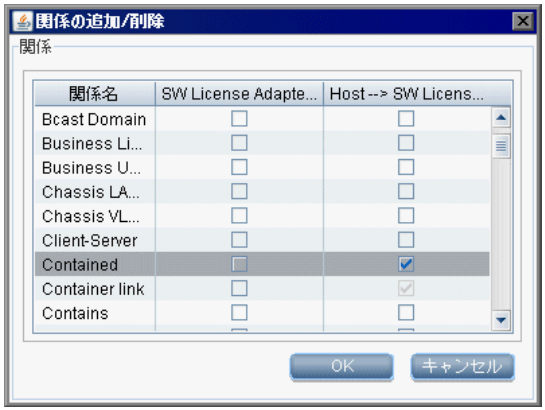


注 : この段階では，データを取り込むメソッドを定義していないので，フェデレート・データをまだ表示できません。

Containment 関係の作成例：

1 CIT マネージャで，2 つの CIT を選択します。

2 2 つの CIT 間で **Containment** 関係を作成します。



アダプタ・パッケージの準備

この手順では，汎用 DB アダプタ・パッケージを見つけて設定します。

1 **C:¥hp¥UCMDB¥UCMDBServer¥content¥adapters** フォルダにある **db-adapter.zip** パッケージを見つけます。

2 このパッケージをローカルの一時ディレクトリに抽出します。

3 アダプタ XML ファイルを編集します。

- **discoveryPatterns¥db_adapter.xml** ファイルをテキスト・エディタで開きます。
- **adapter id** 属性を見つけて，名前を置き換えます。

```
<pattern id="MyAdapter" displayLabel="My Adapter"
xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="Discovery
Pattern Description"
 schemaVersion="9.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
displayName="UCMDB API Population">
```

アダプタでレプリケーション・データをサポートする場合，次の機能を **<adapter-capabilities>** 要素に追加する必要があります。

```
<support-replicatioin-data>
    <source>
        <changes-source/>
    </source>
</support-replicatioin-data>
```

表示ラベルや ID は，HP Universal CMDB の［インテグレーション ポイント］表示枠のアダプタ・リストに表示されます。

CMDB にデータをポピュレートする方法の詳細については,『HP Universal CMDB データ・フロー管理ガイド』の「［Integration Studio］ページ」を参照してください。

➤ アダプタでバージョン 8.x のマッピング・エンジンを使用する場合（新しい調整マッピング・エンジンを使用しない場合），次の要素を

```
<default-mapping-engine/>
```

次の要素で置き換えます。

```
<default-mapping-engine>com.hp.ucmdb.federation.mappingEngine.AdapterMappingEngine</default-mapping-engine>
```

新しいマッピング・エンジンに戻すには，要素を次の値に戻します。

```
<default-mapping-engine/>
```

4 一時ディレクトリで, **adapterCode** フォルダを開いて **GenericDBAdapter** の名前を，手順 3 で使用した **adapter id** の値に変更します。

このフォルダには，アダプタ名，CMDB のクエリとクラス，およびアダプタがサポートする RDBMS のフィールドなど, 連携ロジックを実行する jar ファイルが含まれています。

5 必要に応じてアダプタを設定します。詳細については，133 ページの「アダプタの設定」を参照してください。

6 131 ページ手順「3」で述べているように，**adapter id** 属性に付けたのと同じ名前で *.zip ファイルを作成します。

注 : descriptor.xml ファイルは, すべてのパッケージに存在する標準設定ファイルです。

---

7 前の手順で作成した新しいパッケージを保存します。アダプタの標準設定のディレクトリは，**C:¥hp¥UCMDB¥UCMDBServer¥content¥adapters** です。

アダプタの設定

最小メソッドか高度なメソッドのいずれかを使用してアダプタを設定できます。

これらの構成ファイルは，131 ページの「アダプタ・パッケージの準備」の手順 2 で抽出した**C:¥hp¥UCMDB¥UCMDBServer¥content¥adapters** フォルダの **db-adapter.zip** パッケージにあります。

アダプタ設定 – 最小メソッド

注 : このメソッドを実行した結果として自動的に生成される **orm.xml** ファイルは，高度なメソッドを操作するときに利用できる好例です。

次の手順では，CMDB のクラス・モデルを RDBMS にマップするメソッドについて説明します。この最小メソッドを使用するのは，次の必要がある場合です。

- ノード属性などの単一ノードをフェデレートする。
- 汎用データベース・アダプタの機能を示す。

このメソッドでは，次のものをサポートしています。

- 1 ノード・フェデレーションのみをサポートする
- 多対 1 の仮想関係のみをサポートする

adapter.conf ファイルの設定

このステップでは，データを自動的にフェデレートできるように，adapter.conf ファイルの設定を変更します。

1. **adapter.conf** ファイルをテキスト・エディタで開きます。
2. **use.simplified.xml.config=<true/false>** の行を見つけます。
3. **use.simplified.xml.config=true** に変更します。

simplifiedConfiguration.xml ファイルの設定

この手順では，CMDB の CIT を RDBMS テーブルのフィールドにマップして，**simplifiedConfiguration.xml** ファイルを設定します。

1 **simplifiedConfiguration.xml** ファイルをテキスト・エディタで開きます。

このファイルには，マップする各エンティティに使用するテンプレートが含まれています。

注 : バージョンに関係なく Microsoft Corporation のメモ帳で **simplifiedConfiguration.xml** ファイルを編集しないでください。Notepad++，UltraEdit，またはほかのサードパーティ製テキスト・エディタを使用してください。

---

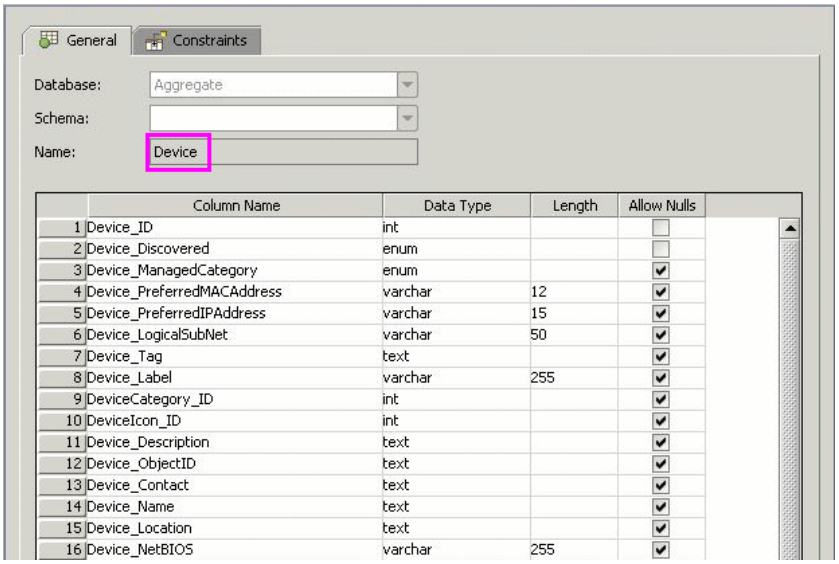
2 属性を次のように変更します。

➤ UCMDB の CIT 名（cmdb-class-name）と RDBMS の対応するテーブル名（default-table-name）：

```
<cmdb-class cmdb-class-name="node" default-table-name="Device">
```

cmdb-class-name 属性はノード CIT から取り出されます。

default-table-name 属性はデバイス・テーブルから取り出されます。



➤ RDBMS の一意の識別子：

```
<primary-key column-name="Device_ID"/>
```

➤ 調整ルール（reconciliation-by-two-nodes）：

```
<reconciliation-by-two-nodes connected-node-cmdb-class-name="ip_address"
cmdb-link-type="containment">
```

➤ UCMDB（cmdb-attribute-name）および RDBMS（column-name）の調整属性：

```
<connected-node-attribute cmdb-attribute-name="name"
column-name="[column_name]"/>
```

➤ CIT の名前（cmdb-class-name）と RDBMS で対応するテーブルの名前（default-table-name）。また，CMDB 関係（connected-cmdb-class-name）と CIT 関係（link-class-name）：

```
<class cmdb-class-name="sw_sub_component"
default-table-name="SWSubComponent" connected-cmdb-class-name="node"
link-class-name="composition">
```

➤ プライマリ・キーと外部キー：

```
<foreign-primary-key column-name="Device_ID"
cmdb-class-primary-key-column="Device_ID"/>
```

➤ RDBMS の一意の識別子：

```
 <primary-key column-name="Device_ID"/>
```

➤ CMDB 属性（cmdb-attribute-name）と RDBMS のカラム名（column-name）間のマッピング：

```
 <attribute cmdb-attribute-name="last_access_time"
column-name="SWSubComponent_LastAccess TimeStamp"/>
```

3 ファイルを保存します。

アダプタ設定 – 高度なメソッド

orm.xml ファイルの設定

この手順では, CMDB の CIT および関係を RDBMS のテーブルにマップします。

1 **orm.xml** ファイルをテキスト・エディタで開きます。

標準設定では，このファイルには，連携に必要な数の CIT と関係をマップするのに使用するテンプレートが含まれています。

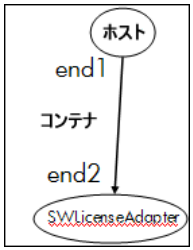
注 : バージョンに関係なく Microsoft Corporation のメモ帳で **orm.xml** ファイルを編集しないでください。Notepad++，UltraEdit，またはほかのサードパーティ製テキスト・エディタを使用してください。

---

2 マップされるデータ・エンティティに従って，ファイルに変更を加えます。詳細については，次の例を参照してください。

次のタイプの関係を **orm.xml** ファイルにマップできます。

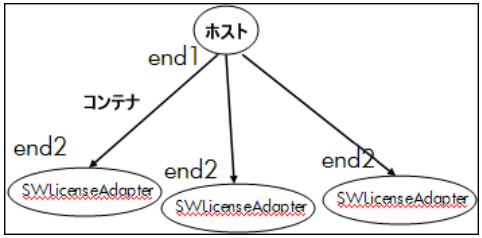
➤ 1 対 1:



このタイプの関係を記述するコードは次のようになります。

```
<one-to-one name="end1" target-entity="node">
        <join-column name=Device_ID" />
</one-to-one>
<one-to-one name="end2" target-entity=sw_sub_component">
        <join-column name=Device_ID" />
        <join-column name=Version_ID" />
</one-to-one>
```

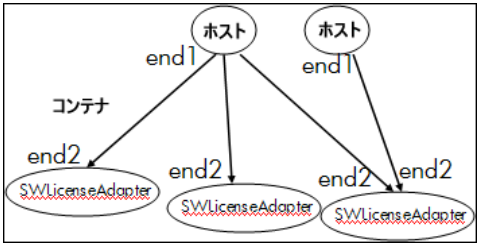
➤ 多対 1:



このタイプの関係を記述するコードは次のようになります。

```
<many-to-one name="end1" target-entity="node">
        <join-column name=Device_ID" />
</many-to-one>
<one-to-one name="end2" target-entity=sw_sub_component">
        <join-column name=Device_ID" />
        <join-column name=Version_ID" />
</one-to-one>
```

➤ 多対多：



このタイプの関係を記述するコードは次のようになります。

```
<many-to-one name="end1" target-entity="node">
        <join-column name=Device_ID" />
</many-to-one>
<many-to-one name="end2" target-entity=sw_sub_component">
        <join-column name=Device_ID" />
        <join-column name=Version_ID" />
</many-to-one>
```

命名規則の詳細については，176 ページの「命名規則」を参照してください。

データ・モデルと RDBMS 間のエンティティ・マッピングの例 :

注 : 設定する必要がない属性は，次の例から除外しています。

➤ CMDB CIT のクラス :

```
<entity class="generic_db_adapter.node">
```

➤ RDBMS にあるテーブルの名前 :

```
<table name="Device"/>
```

➤ RDBMS テーブルにある一意の識別子のカラム名 :

```
<column name="Device ID"/>
```

➤ CMDB CIT にある属性の名前 :

```
<basic name="name">
```

➤ 外部データ・ソースにあるテーブル・フィールドの名前 :

```
<column name="Device_Name"/>
```

➤ 128 ページの「CI タイプの作成」で作成した新規 CIT の名前 :

```
<entity class="generic_db_adapter.MyAdapter">
```

➤ RDBMS で対応するテーブルの名前 :

```
<table name="SW_License"/>
```

➤ RDBMS の一意の識別子 :

```
<id name="id1">
        <column updatable="false" insertable="false" name="Device_ID"/>
        <generated-value strategy="TABLE"/>
</id>
<id name="id2">
        <column updatable="false" insertable="false" name="Version_ID"/>
        <generated-value strategy="TABLE"/>
</id>
```

➤ CMDB CIT にある属性名と，RDBMS で対応する属性の名前 :

```
<basic name="license_required">
        <column updatable="false" insertable="false"
name="MyAdapter_LicenseRequired"/>
```

データ・モデルと RDBMS 間の関係マッピングの例 :

➤ CMDB 関係のクラス :

```
<entity class="generic_db_adapter.node_containment_MyAdapter">
```

➤ 関係が実行される RDBMS テーブルの名前 :

```
<table name="MyAdapter"/>
```

➤ RDBMS にある一意の ID:

```
<id name="id1">
        <column updatable="false" insertable="false" name="Device_ID"/>
        <generated-value strategy="TABLE"/>
</id>
<id name="id2">
        <column updatable="false" insertable="false" name="Version_ID"/>
        <generated-value strategy="TABLE"/>
</id>
```

➤ 関係タイプと CMDB CIT:

```
<many-to-one target-entity="node" name="end1">
```

➤ RDBMS にあるプライマリ・キーおよび外部キー・フィールド :

```
<join-column updatable="false" insertable="false"
referenced-column-name="[column_name]" name="Device_ID"/>
```

reconciliation_types.txt ファイルの設定

reconciliation_types.txt ファイルをテキスト・エディタで開きます。

詳細については，177 ページの「reconciliation_types.txt ファイル」を参照してください。

reconciliation_rules.txt ファイルの設定

この手順では，アダプタで CMDB と RDBMS を調整するルールを定義します（バージョン 8.x との下位互換性のためにマッピング・エンジンが使用されている場合のみ）。

1. テキスト・エディタで，**META-INF¥reconciliation_rules.txt** を開きます。
2. マッピングする CIT に従って，ファイルに変更を加えます。たとえば，ノード CIT をマップするには，次の式を使います。

```
multinode[node] ordered expression[^name]
```

注：

- データベースのデータで大文字 / 小文字を区別する場合は，制御文字（^）を削除しないでください。
- それぞれの左角括弧に対応する右角括弧があるかチェックしてください。

詳細については，177 ページの「reconciliation_rules.txt ファイル（下位互換性用）」を参照してください。

プラグインの実装

このタスクでは，プラグインとともに汎用 DB アダプタを実装，デプロイする方法について説明します。

注 : アダプタのプラグインを記述する前に，131 ページの「アダプタ・パッケージの準備」の必要な手順をすべて完了していることを確認してください。

---

1 プラグインの Java インタフェースを実装した Java クラスを書きます。下の表に，各プラグインで実装する必要があるインタフェースを示します（インタフェースはすべて com.mercury.topaz.fcmdb.adapters.dbAdapter.plugin パッケージ内にあります）。

プラグインのタイプ	インタフェース名	メソッド
全トポロジの同期化	FcmdbPluginForSyncGetFullTopology	getFullTopology
変更の同期化	FcmdbPluginForSyncGetChangesTopology	getChangesTopology
レイアウトの同期化	FcmdbPluginForSyncGetレイアウト	getLayout
サポートされるクエリの取得	FcmdbPluginForSyncGetSupportedQueries	getSupportedQueries
TQL クエリ定義と結果の変更	FcmdbPluginForGetTopologyCmdbFormat	getTopologyCmdbFormat
CI に対するレイアウト要求の変更	FcmdbPluginForGetCIsレイアウト	getCisLayout
リンクに対するレイアウト要求の変更	FcmdbPluginForGetRelationsLayout	getRelationsLayout

2 作成した Java コードをコンパイルする前に，連携 SDK JAR と汎用 DB アダプタ JAR が自分のクラス・パス内にあることを確認してください。連携 SDK は，**C:¥hp¥UCMDB¥UCMDBServer¥lib** ディレクトリにある **federation_api.jar** ファイルです。

3 作成したクラスを jar ファイルにパックし，デプロイする前に，アダプタ・パッケージ内の adapterCode¥< 自分のアダプタ名 > フォルダに入れます。

プラグインは，アダプタの **¥META-INF** フォルダにある **plugins.txt** ファイルを使って設定します。

DDMi アダプタのファイルの例を次に示します。

```
# 完全なトポロジを同期させるための必須プラグイン
[getFullTopology]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin

# トポロジ内の変更を同期させるための必須プラグイン
[getChangesTopology]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin

# レイアウトを同期させるための必須プラグイン
[getLayout]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin

# サポートされているクエリを同期させるためのプラグイン。定義されていない場合は
すべての tqls 名を返す
[getSupportedQueries]

# tql 定義および tql 結果を変更する内部の任意プラグイン
[getTopologyCmdbFormat]

# レイアウト要求と CIs 結果を変更する内部の任意プラグイン
[getCisLayout]

# レイアウト要求と関係結果を変更する内部の任意プラグイン
[getRelationsLayout]
```

凡例：

- コメント行。

［< アダプタのタイプ >］- あるアダプタ・タイプに対する定義部分の開始。

それぞれの［< アダプタのタイプ >］の下には，関連付けられたプラグイン・クラスがない場合は空白行を，またはプラグイン・クラスの完全修飾名を入れることができます。

4 新しい jar ファイルと更新した **plugins.xml** ファイルで, 作成したアダプタをパックします。パッケージ内の残りのファイルは，汎用 DB アダプタをベースにしたほかのアダプタと同じにする必要があります。

アダプタのデプロイ

1 UCMDB で，パッケージ・マネージャにアクセスします。詳細については，『HP UCMDB 管理ガイド』の「[パッケージ マネージャ] ページ」を参照してください。

2 ローカル・ディスクから [**サーバにパッケージをデプロイ**] アイコン をクリックして，作成したアダプタ・パッケージを参照します。パッケージを選択して [**開く**] をクリックし，次に [**デプロイ**] をクリックしてパッケージ・マネージャ内にパッケージを表示します。

3 リストからパッケージを選択して [**パッケージ リソースの表示**] アイコンをクリックし，パッケージ・マネージャによってパッケージの内容が認識されていることを確認します。

アダプタの編集

作成後デプロイしたアダプタは，UCMDB で編集できます。詳細については，99 ページの「アダプタ管理」を参照してください。

インテグレーション・ポイントの作成

このステップでは，フェデレーションが機能しているか，すなわち，接続が有効で，XML ファイルが有効であるかチェックします。ただし，このチェックでは，XML が RDBMS の正しいフィールドにマッピングされるかは確認されません。

1 UCMDB で，Integration Studio にアクセスします（[**データ フロー管理**] > [**Integration Studio**]）。

2 インテグレーション・ポイントを作成します。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「[新規インテグレーション ポイントの作成 / インテグレーションのプロパティの編集] ダイアログ・ボックス」を参照してください。

   [連携] タブには，このインテグレーション・ポイントを使って連携できる CIT がすべて表示されます。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「[連携] タブ」を参照してください。

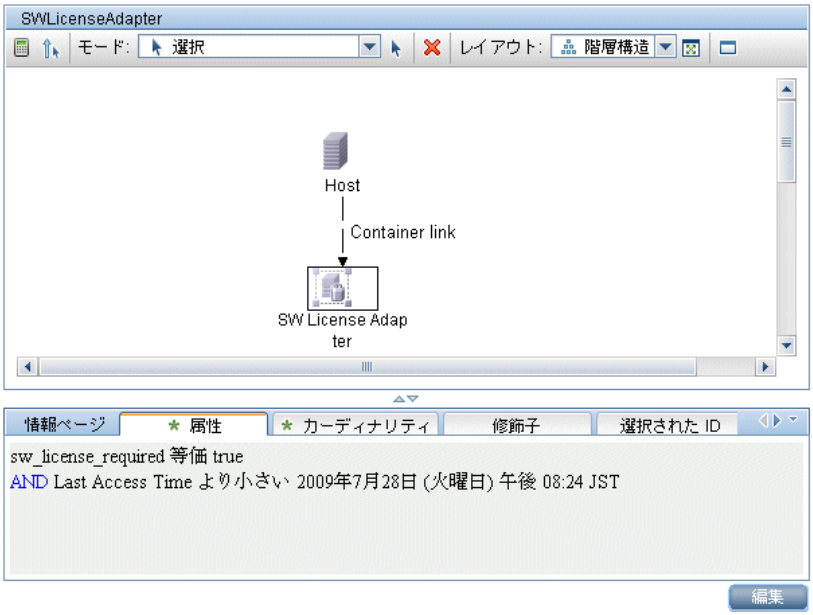
ビューの作成

このステップでは，CIT のインスタンスを表示できるビューを作成します。

1 UCMDB で，モデリング・スタジオにアクセスします（[**モデリング**] > [**モデリング スタジオ**]）。

2 ビューを作成します。詳細については，『モデリング・ガイド』の「テンプレート ベース ビューの作成」を参照してください。

3 TQL に条件（最終アクセス日時が 6 か月を超えるなど）を追加できます。



結果の計算

このステップでは，結果をチェックします。

1 UCMDB で，モデリング・スタジオにアクセスします（[**モデリング**] > [**モデリング スタジオ**]）。

2 ビューを開きます。

3 [クエリ結果数を計算する] ボタンをクリックして結果を計算します。

4 [**プレビュー**] ボタンをクリックしてビューに CI を表示します。

結果の表示

このステップでは，結果を表示し，手順の問題をデバッグします。たとえば，ビューに何も表示されない場合は，**orm.xml** ファイルで定義をチェックし，関係属性を削除して，アダプタを再度読み込みます。

1. UCMDB で，［IT ユニバース マネージャ］にアクセスします（［**モデリング**］＞［**IT ユニバース マネージャ**］）。
2. CI を選択します。

   ［プロパティ］タブに，連携の結果が表示されます。

レポートの表示

このステップでは，トポロジ・レポートを表示します。詳細については，『モデリング・ガイド』の「トポロジ・レポートの概要」を参照してください。

ログ・ファイルの有効化

計算フロー，アダプタ・ライフサイクルを理解したり，デバッグ情報を表示するには，ログ・ファイルを参照します。詳細については，200 ページの「アダプタ・ログ・ファイル」を参照してください。

Eclipse を使用した CIT 属性とデータベース・テーブル間のマッピング

注意：この手順は，コンテンツ開発について高度な知識を持つユーザを対象としています。ご質問やご不明な点は，HP ソフトウェア・サポートまでお問い合わせください。

このタスクでは，Eclipse の J2EE エディションで提供されている JPA プラグインをインストールおよび使用し，次の作業を実行する方法について説明します。

- CMDB クラス属性とデータベース・テーブル・カラム間のグラフィカルなマッピングを可能にする。
- マッピング・ファイル（orm.xml）を手動で編集できるようにし，正確性をチェックする。正確性チェックでは，構文のチェックとともに，クラス属性とマップされたデータベース・テーブル・カラムが正しく記述されているかどうかの検証も行われます。
- マッピング・ファイルを CMDB サーバにデプロイできるようにし，エラーを表示してさらに正確性をチェックする。
- CMDB サーバにサンプル・クエリを定義し，Eclipse から直接実行してマッピング・ファイルをテストする。

このタスクには次の手順が含まれます。

- 148 ページの「前提条件」
- 148 ページの「インストール」
- 149 ページの「作業環境を準備する」
- 153 ページの「アダプタを作成する」
- 153 ページの「CMDB プラグインを設定する」
- 155 ページの「UCMDB クラス・モデルをインポートする」
- 156 ページの「ORM ファイルの作成 – UCMDB クラスをデータベース・テンプレートにマップする」
- 158 ページの「ID をマップする」
- 159 ページの「属性をマップする」

1 前提条件

Eclipse を実行するマシンに **http://java.sun.com/javase/downloads/index.jsp** から **Java ランタイム環境（JRE）6 Update 7** をインストールします。

この手順は，Java 5（以上）のランタイム環境で動作します。

2 インストール

a **Eclipse IDE for Java EE Developers** を **<http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/ganymede/SR1/eclipse-jee-ganymede-SR1-win32.zip>** からローカル・フォルダ（**C:¥Program Files¥eclipse** など）にダウンロードして抽出します。

b **com.hp.plugin.import_cmdb_model_1.0.jar** を **C:¥Program Files¥Eclipse¥plugins** にコピーします。

このファイルには，次の SharePoint サイトからアクセスします。

http://teams1.sharepoint.hp.com/teams/uCMDBF-adapters/default.aspx?RootFolder=%2fteams%2fuCMDBF%2dadapters%2fShared%20Documents%2fGDBA%20UI%20%28eclipse%20based%29%2e%20Compatible%20starting%20uCMDB%207%2e5%2e1&FolderCTID=&View=%7b8BE99EE7%2d790B%2d47BB%2d881A%2dFF51F942BAA9%7d

c **C:¥Program Files¥Eclipse¥eclipse.exe** を起動します（Java 5 以上のランタイム環境が必要です）。Java 仮想マシンが見つからないというメッセージが表示された場合，次のコマンド・ラインで **eclipse.exe** を起動します。

```
"C:¥Program Files¥eclipse¥eclipse.exe" -vm "<JRE installation folder>¥bin"
```

3 作業環境を準備する

この手順では，ワークスペース，データベース，接続，およびドライバのプロパティを設定します。

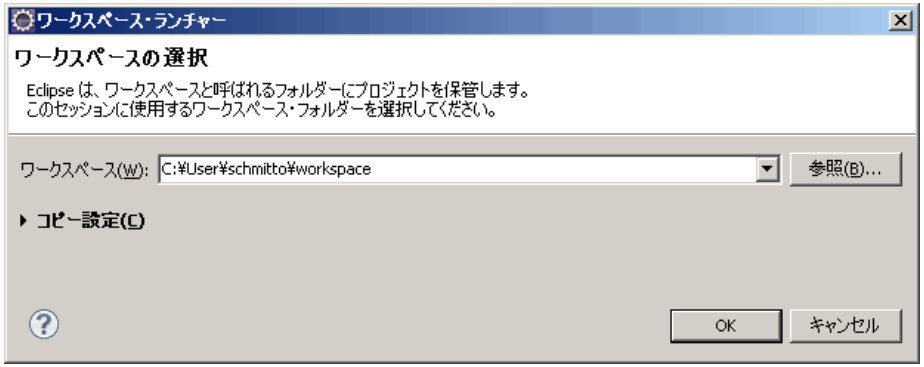
a ファイル **workspaces_gdb.rar** を **C:¥Documents and Settings¥AllUsers¥workspaces** に抽出します。

注 : 正確なフォルダ・パスを使用する必要があります。ファイルを誤ったパスに展開した場合やファイルを展開しなかった場合，この手順は機能しません。

このファイルには，次の SharePoint サイトからアクセスします。

http://teams1.sharepoint.hp.com/teams/uCMDBF-adapters/default.aspx?RootFolder=%2fteams%2fuCMDBF%2dadapters%2fShared%20Documents%2fGDBA%20UI%20%28eclipse%20based%29%2e%20Compatible%20starting%20uCMDB%207%2e5%2e1&FolderCTID=&View=%7b8BE99EE7%2d790B%2d47BB%2d881A%2dFF51F942BAA9%7d

b Eclipse で，［ファイル］＞［ワークスペースの切り替え］＞［その他］を選択します。

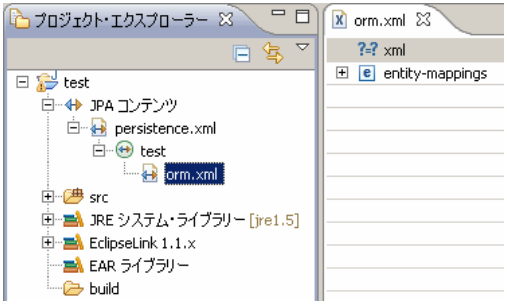


作業環境に応じて次のように選択します。

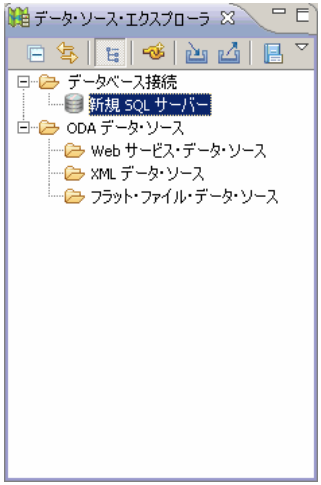
- SQL Server の場合，フォルダ **C:¥Documents and Settings¥AllUsers¥workspace_gdb_sqlserver** を選択します。
- MySQL の場合，フォルダ **C:¥Documents and Settings¥AllUsers¥workspace_gdb_mysql** を選択します。
- Oracle の場合，フォルダ **C:¥Documents and Settings¥AllUsers¥workspace_gdb_oracle** を選択します。

c ［**OK**］をクリックします。

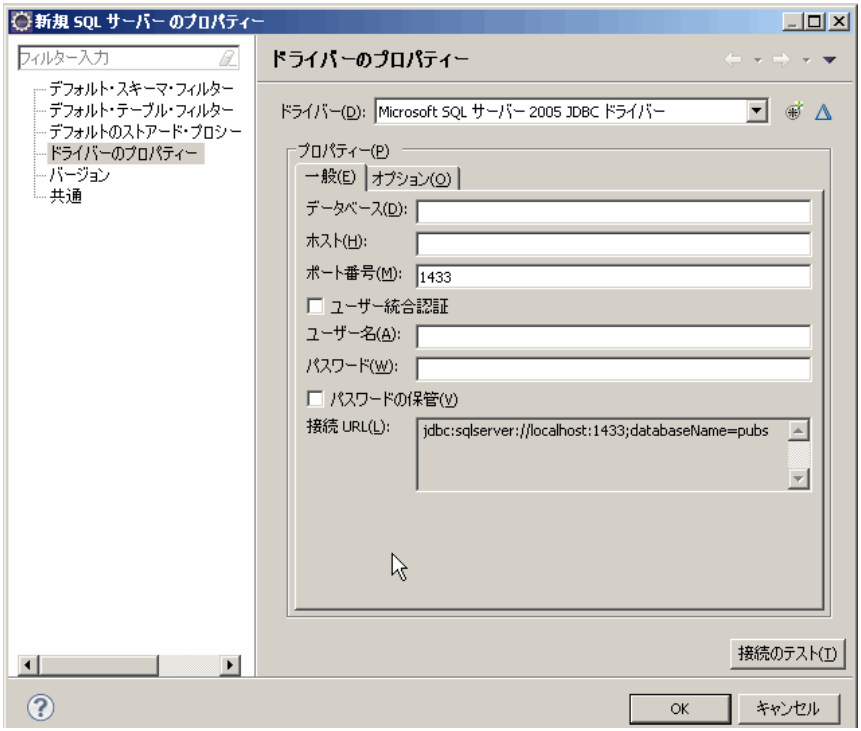
d Eclipse で，［プロジェクト・エクスプローラー］ビューを表示し，［**アクティブなプロジェクト**］＞［**JPA コンテンツ**］＞［**persistence.xml**］＞［**アクティブなプロジェクト**］＞［**orm.xml**］を選択します。



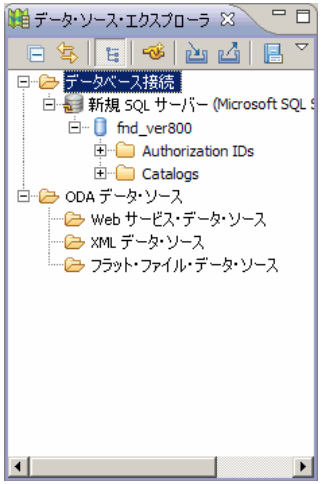
e ［データ・ソース・エクスプローラ］ビュー（左下の表示枠）で，データベース接続を右クリックし，［**プロパティー**］メニューを選択します。



f ［**新規 SQL サーバーのプロパティー**］ダイアログ・ボックスで，［**共通**］を選択し，［**ワークベンチを開始するたびに接続**］チェック・ボックスを選択します。［**ドライバーのプロパティー**］を選択し，接続プロパティを入力します。［**接続のテスト**］をクリックし，接続が機能することを確認します。［**OK**］をクリックします。



g ［データ・ソース・エクスプローラ］ビューで，データベース接続を右クリックし，［**接続**］をクリックします。データベース・スキーマおよびテーブルを含むツリーが，データベース接続アイコンの下に表示されます。

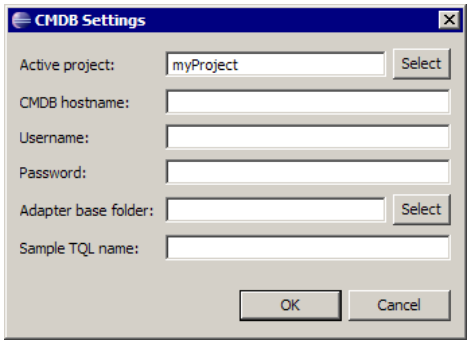


4 アダプタを作成する

40 ページの「手順 1：アダプタの作成」のガイドラインに従ってアダプタを作成します。

5 CMDB プラグインを設定する

a Eclipse で，［**UCMDB**］＞［**Settings**］をクリックして［**CMDB Settings**］ダイアログ・ボックスを開きます。



b 新しく作成した JPA プロジェクトをまだアクティブなプロジェクトとして選択していない場合は選択します。

c CMDB ホスト名を入力します（**localhost** や **labm1.itdep1** など）。アドレスにポート番号や http:// プレフィックスを含める必要はありません。

d CMDB API にアクセスするためのユーザ名とパスワードを入力します。通常は **admin/admin** です。

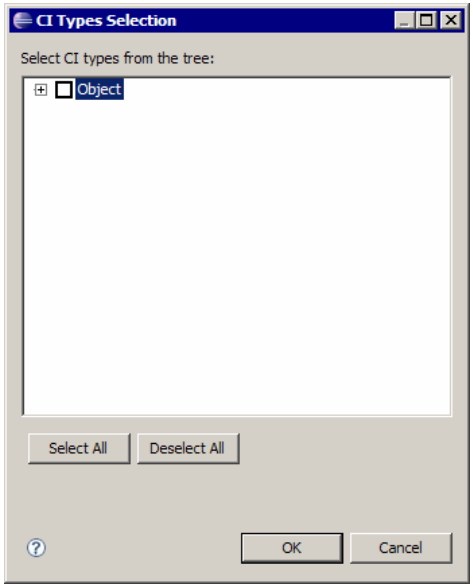
e CMDB サーバ上の **C:¥hp** フォルダがネットワーク・ドライブとしてマップされていることを確認します。

f **C:¥hp** の下にある関連アダプタの基本フォルダを選択します。基本フォルダとは，**dbAdapter.jar** ファイルと **META-INF** サブフォルダが含まれるフォルダです。基本フォルダのパスは，**C:¥hp¥UCMDB¥UCMDBServer¥runtime¥fcmdb¥CodeBase¥<** ***アダプタ名*** **>** になります。末尾に円記号（¥）がないことを確認します。

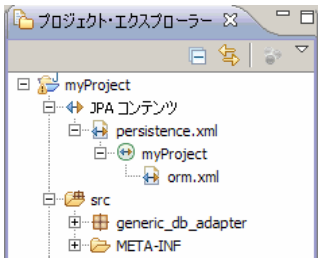
6 UCMDB クラス・モデルをインポートする

この手順では，JPA エンティティとしてマップする CIT を選択します。

a ［**UCMDB**］＞［**Import CMDB Class Model**］をクリックして［**CI Type Selection**］ダイアログ・ボックスを開きます。



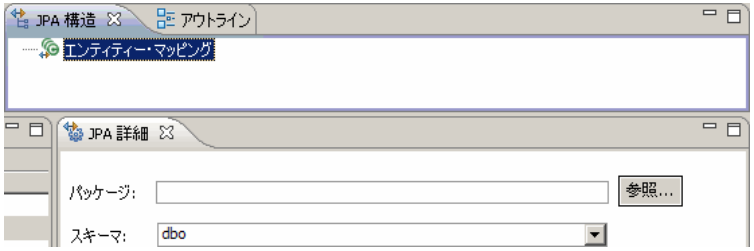
b JPA エンティティとしてマップする CI タイプを選択します。［**OK**］をクリックします。CI タイプは Java クラスとしてインポートされます。インポートした CI タイプがアクティブなプロジェクトの **src** フォルダの下に表示されることを確認します。



7 ORM ファイルの作成 – UCMDB クラスをデータベース・テンプレートにマップする

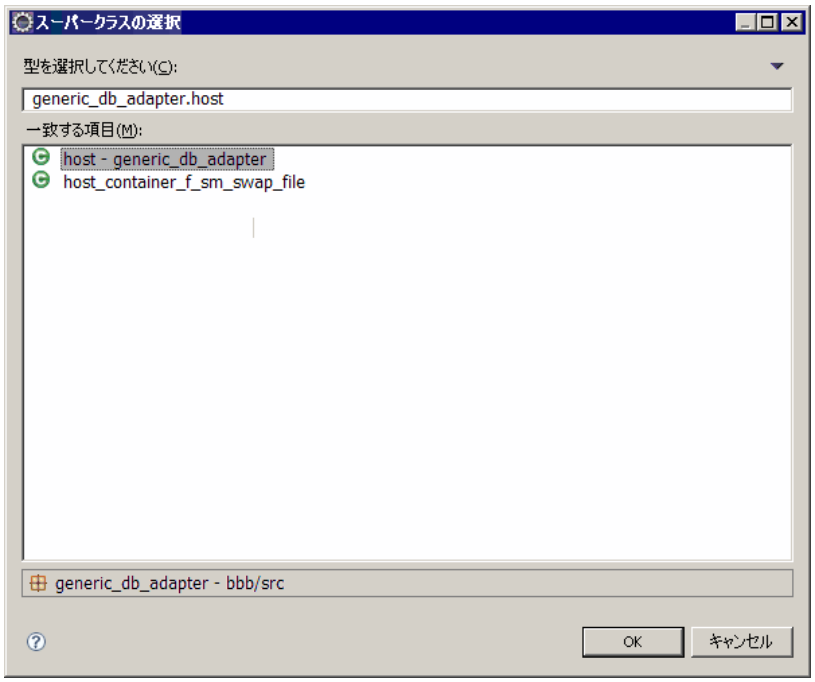
この手順では，Java クラス（前の手順でインポートしたもの）をデータベース・テーブルにマップします。

a データベース接続が接続中であることを確認します。[プロジェクト・エクスプローラー] でアクティブなプロジェクト（標準設定では myProject）を右クリックします。[JPA] ビューを選択し，[**Override default schema from connection**] チェック・ボックスを選択して，関連するデータベース・スキーマを選択します。[**OK**] をクリックします。



b CIT を次のようにマップします。[JPA 構造] ビューで，[**エンティティー・マッピング**] 分岐を右クリックし，[**クラスの追加 ...**] を選択します。[**永続化クラスの選択**] ダイアログ・ボックスが開きます。[**次でマップ :**] フィールド（**Entity**）は変更しないでください。

c ［**参照 ...**］をクリックし，マップする UCMDB クラスを選択します（**generic_db_adapter** パッケージに属するすべての UCMDB クラス）。

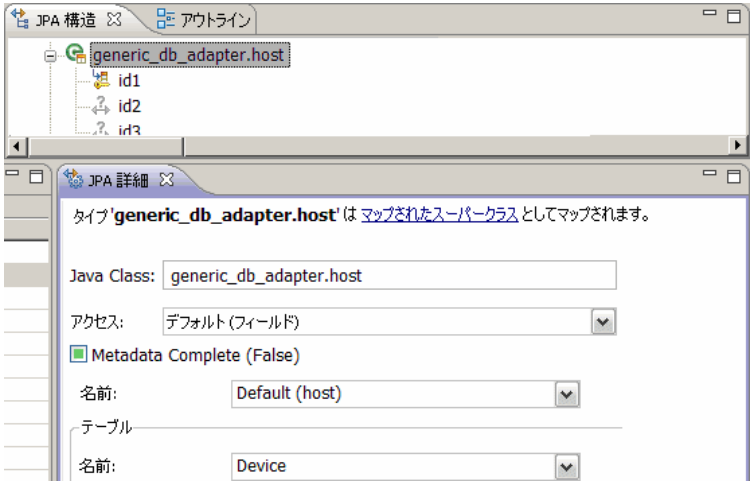


d 両方のダイアログ・ボックスで［**OK**］をクリックします。選択したクラスが，［JPA 構造］ビューの［**エンティティー・マッピング**］分岐の下に表示されます。

注 : エンティティが属性ツリーなしで表示された場合は，［プロジェクト・エクスプローラー］ビューでアクティブなプロジェクトを右クリックします。［**Close**］を選択してから，［**Open**］を選択します。

---

e ［JPA 詳細］ビューで，UCMDB クラスのマップ先となるプライマリ・データベース・テーブルを選択します。ほかのすべてのフィールドは変更しないでください。



8 ID をマップする

JPA 標準では，永続クラスにはそれぞれ少なくとも 1 つの ID 属性が必要です。UCMDB クラスの場合，最大 3 個の属性を ID としてマップできます。ID 属性としては，**id1**，**id2**，**id3** などがあります。ID 属性をマップするには，次の手順で行います。

a ［JPA 構造］ビューの［**エンティティー・マッピング**］分岐の下にある対応するクラスを展開し，関連する属性（**id1** など）を右クリックし，［**XML およびマップに属性を追加 ...**］を選択します。

b ［**属性の追加**］ダイアログ・ボックスが開きます。［**次でマップ :**］フィールドで［**Id**］を選択し，［**OK**］をクリックします。

c ［JPA 詳細］ビューで，ID フィールドのマップ先となるデータベース・テーブル・カラムを選択します。

9 属性をマップする

この手順では，属性をデータベース・カラムにマップします。

a ［JPA 構造］ビューの［**エンティティー・マッピング**］分岐の下にある対応するクラスを展開し，関連する属性（**host_hostname** など）を右クリックし，［**XML およびマップに属性を追加 ...**］を選択します。

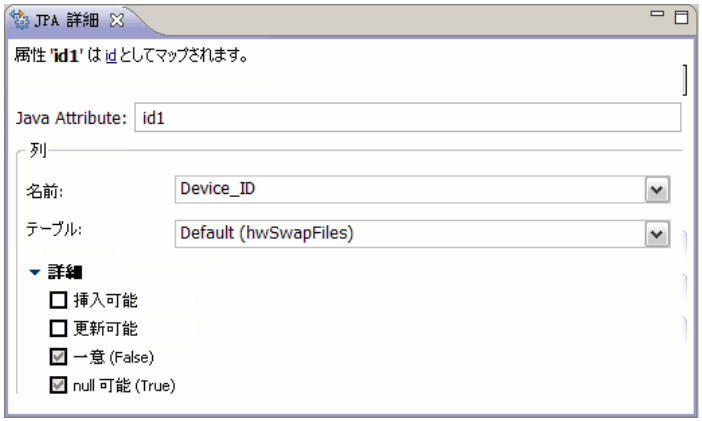
b ［**属性の追加**］ダイアログ・ボックスが開きます。［**次でマップ :**］フィールドで［**基本**］を選択し，［**OK**］をクリックします。

c ［JPA 詳細］ビューで，属性フィールドのマップ先となるデータベース・テーブル・カラムを選択します。

10 有効なリンクをマップする

156 ページ手順「b」で説明されている方法で，有効なリンクを示す UCMDB クラスをマップする手順を実行します。このようなクラスの名前はそれぞれ，**<end1 エンティティ名 >_< リンク名 >_<end 2 エンティティ名 >** という構造になっています。たとえば，ホストと場所の間の **Contains** リンクは，**generic_db_adapter.host_contains_location** という名前の Java クラスで示されます。詳細については，177 ページの「reconciliation_rules.txt ファイル（下位互換性用）」を参照してください。

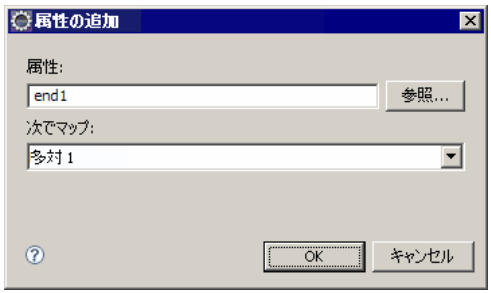
a 158 ページの「ID をマップする」で説明されているように，リンク・クラスの ID 属性をマップします。ID 属性ごとに，[JPA 詳細] ビューで [**詳細**] チェック・ボックス・グループを展開し，[**挿入可能**] および [**更新可能**] チェック・ボックスをクリアします。



b リンク・クラスの **end1** および **end2** 属性をマップします。リンク・クラスの **end1** および **end2** 属性ごとに，次の手順で行います。

➤ [JPA 構造] ビューの [**エンティティー・マッピング**] 分岐の下にある対応するクラスを展開し，関連する属性（**end1** など）を右クリックし，[**XML およびマップに属性を追加 ...**] を選択します。

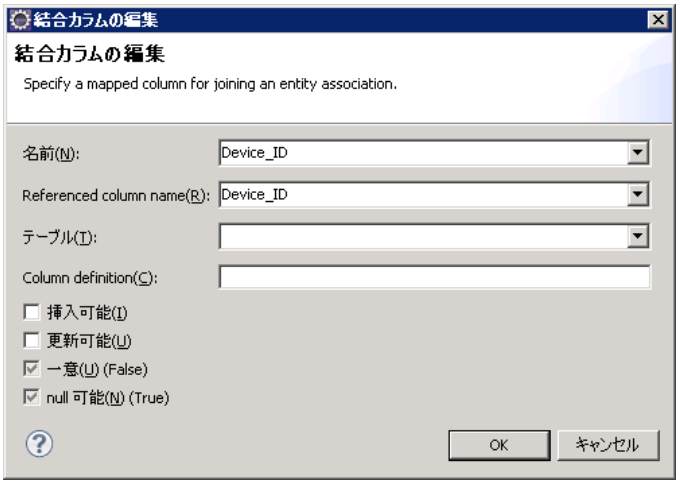
➤ ［**属性の追加**］ダイアログ・ボックスの［**次でマップ :**］フィールドで，［**多対 1**］または［**1 対 1**］を選択します。



➤ 指定した **end1** または **end2** CI にこのタイプのリンクを複数設定できる場合は，［**多対 1**］を選択します。それ以外の場合は，［**1 対 1**］を選択します。たとえば，**host_contains_ip** リンクの場合，1 つのホストに複数の IP を設定できるため，**host** エンドを［**多対 1**］としてマップし，1 つの IP に設定できるホストは 1 つのみであるため，**ip** エンドは［**1 対 1**］としてマップする必要があります。

➤ ［[JPA 詳細］ビューで，［**Target entity**］を選択します（**generic_db_adapter.host** など）。

➤ ［JPA 詳細］ビューの［**Join Columns**］セクションで，［**Override Default**］をチェックします。［**Edit**］をクリックします。［**結合カラムの編集**］ダイアログ・ボックスで，**end1/end2** ターゲット・エンティティ・テーブル内のエントリを指し示す，リンク・データベース・テーブルの外部キー・カラムを選択します。**end1/end2** ターゲット・エンティティ・テーブル内の参照されるカラム名が ID 属性にマップされている場合は，［**Referenced Column Name**］を変更せずに残します。それ以外の場合は，外部キー・カラムが指定するカラムの名前を選択します。［**挿入可能**］および［**更新可能**］チェック・ボックスをクリアし，［**OK**］をクリックします。



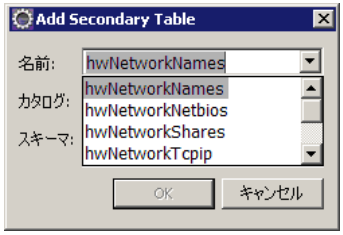
➤ **end1/end2** ターゲット・エンティティに複数の ID がある場合，前の手順で説明されているように，［**追加 ...**］ボタンをクリックしてさらに join カラムを追加し，マップします。

11 ORM ファイルの作成 – セカンダリ・テーブルを使用する

JPA では，Java クラスを複数のデータベース・テーブルにマップできます。たとえば，**Host** を **Device** テーブルにマップして，そのほとんどの属性の永続性を有効にし，さらに **NetworkNames** テーブルにマップして **host_hostName** の永続性を有効にすることができます。この場合，**Device** がプライマリ・テーブルで，**NetworkNames** がセカンダリ・テーブルになります。定義できるセカンダリ・テーブルの数に制限はありません。唯一の条件として，プライマリ・テーブルとセカンダリ・テーブルのエントリ間の関係が 1 対 1 である必要があります。

12 セカンダリ・テーブルを定義する

［JPA 構造］ビューで適切なクラスを選択します。［JPA 詳細］ビューで，［**Secondary Tables**］セクションにアクセスし，［**追加 ...**］をクリックします。［**Add Secondary Table**］ダイアログ・ボックスで，適切なセカンダリ・テーブルを選択します。ほかのフィールドは変更しないでください。



プライマリ・テーブルとセカンダリ・テーブルのプライマリ・キーが異なる場合，［JPA 詳細］ビューの［**Primary Key Join Columns**］セクションで join カラムを設定します。

13 属性をセカンダリ・テーブルにマップする

次のようにクラス属性をセカンダリ・テーブルのフィールドにマップします。

a 159 ページの「属性をマップする」で説明されているように，属性をマップします。

b ［JPA 詳細］ビューの［**Column**］セクションにある［**Table**］フィールドでセカンダリ・テーブル名を選択し，標準設定値を置き換えます。

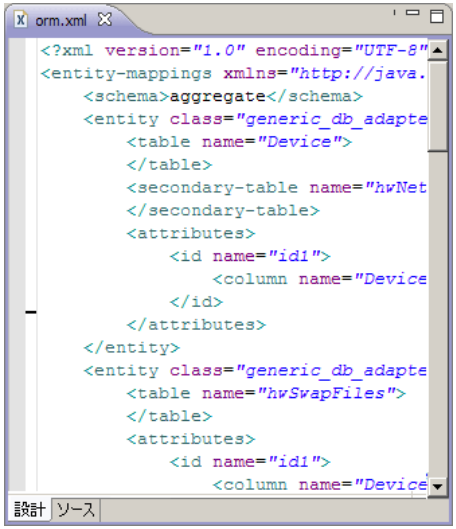
14 既存の ORM ファイルを基礎として使用する

既存の orm.xml ファイルを開発する ORM ファイルの基礎として使用するには，次の手順で行います。

a 既存の **orm.xml** ファイル内でマップされている CIT すべてがアクティブな Eclipse プロジェクトにインポートされていることを確認します。

b 既存のファイルからエンティティ・マッピングの全部または一部を選択してコピーします。

c Eclipse JPA パースペクティブで，**orm.xml** ファイルの［**Source**］タブを選択します。



d コピーしたすべてのエンティティ・マッピングを，編集した **orm.xml** ファイルの **<entity-mappings>** タグの下の，**<schema>** タグの直下に貼り付けます。スキーマ・タグが 156 ページ手順「b」の説明のとおりに設定されていることを確認します。貼り付けたすべてのエンティティが［JPA 構造］ビューに表示されます。これ以降，マッピングは，**orm.xml** ファイルの xml コードを使用してグラフィカルまたは手動の両方で編集できます。

e ［**Save**］をクリックします。

15 アダプタから既存の ORM ファイルをインポートする

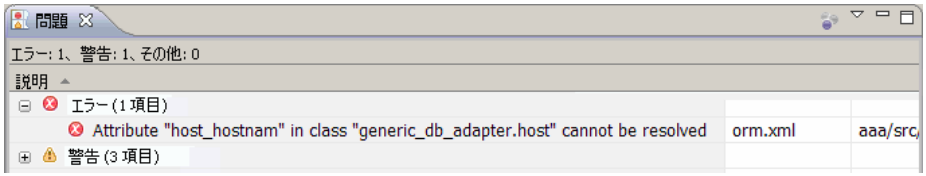
アダプタがすでに存在する場合，Eclipse プラグインを使用してその ORM ファイルをグラフィカルに編集できます。ORM ファイルを Eclipse にインポートしたら，プラグインを使用して編集し，UCMDB マシンに再度デプロイします。ORM ファイルをインポートするには，Eclipse ツールバーのボタンを押します。確認ダイアログが表示されます。［**OK**］をクリックします。ORM ファイルが UCMDB マシンからアクティブな Eclipse プロジェクトにコピーされ，関連するすべてのクラスが UCMDB クラス・モデルからインポートされます。

関連するクラスが［JPA 構造］ビューに表示されない場合は，［プロジェクト・エクスプローラー］ビューでアクティブなプロジェクトを右クリックし，［**Close**］を選択してから，［**Open**］を選択します。

これ以降，ORM ファイルは Eclipse を使用してグラフィカルに編集できるようになり，編集後，UCMDB マシンに再度デプロイできます。詳細については，165 ページの「ORM ファイルを CMDB にデプロイする」を参照してください。

16 ORM ファイルの正確性をチェックする – 組み込みの正確性チェック機能

Eclipse JPA プラグインは，**orm.xml** ファイル内にエラーがあるかどうかをチェックし，あった場合はファイル内のエラーをマークします。構文（タグ名の誤り，終了タグの欠落，ID の不在など）とマッピング・エラー（属性名やデータベース・テーブル・フィールド名の誤りなど）の両方がチェックされます。エラーがある場合，［**問題**］ビューにその説明が表示されます。



17 新規インテグレーション・ポイントを作成する

該当するアダプタについて CMDB 内にインテグレーション・ポイントがない場合，Integration Studio で作成できます。詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「Integration Studio」を参照してください。

開いたダイアログ・ボックスにインテグレーション・ポイント名を入力します。**orm.xml** ファイルがアダプタ・フォルダにコピーされます。インテグレーション・ポイントが作成されます。この際，インポートされたすべての CI タイプがサポート対象クラスとして使用されますが，reconciliation_rules.txt に設定されている multinode（マルチノード）CIT は除外されます。詳細については，177 ページの「reconciliation_rules.txt ファイル（下位互換性用）」を参照してください。

18 ORM ファイルを CMDB にデプロイする

orm.xml ファイルを保存し，UCMDB サーバにデプロイします（［**UCMDB**］>［**Deploy ORM**］をクリックします）。**orm.xml** がアダプタ・フォルダにコピーされ，アダプタが再度読み込まれます。操作結果は，［**Operation Result**］ダイアログ・ボックスに表示されます。再読み込み・プロセス中にエラーが発生した場合，Java 例外スタック・トレースがダイアログ・ボックスに表示されます。インテグレーション・ポイントがアダプタを使用して定義されていない場合，デプロイ時にマッピング・エラーは検出されません。

19 サンプル TQL クエリを実行する

a クエリ・マネージャ（ビュー・マネージャではなく）を使用してクエリを定義します。

b **GenericDBAdapter** アダプタを使用してインテグレーション・ポイントを作成します。詳細については,『HP Universal CMDB データ・フロー管理ガイド』の「[新規インテグレーション ポイントの作成 / インテグレーションのプロパティの編集] ダイアログ・ボックス」を参照してください。

c アダプタの作成時, クエリに含まれる CI タイプがこのインテグレーション・ポイントでサポートされていることを確認します。

d CMDB プラグインを設定するときに, [Settings] ダイアログ・ボックスにこのサンプル・クエリ名を使用します。詳細については, 153 ページの「CMDB プラグインを設定する」を参照してください。

e [**Run TWL**] ボタンをクリックしてサンプル TQL を実行し, サンプル TQL が新しく作成した **orm.xml** ファイルを使用して必要な結果を返すかどうかを確認します。

参照先

アダプタ構成ファイル

このセクションで説明するファイルは，**C:¥hp¥UCMDB¥UCMDBServer¥content¥adapters** フォルダの **db-adapter.zip** パッケージにあります。

本項の内容

一般的な設定

- **adapter.conf:** アダプタ設定ファイルです。詳細については，168 ページの「adapter.conf ファイル」を参照してください。

詳細な設定

- **orm.xml**:CMDB CIT とデータベース・テーブル間のマップを指定するオブジェクト関連マッピング・ファイルです。詳細については，172 ページの「orm.xml ファイル」を参照してください。
- **reconciliation_types.txt**: 調整タイプの設定に使用するルールが含まれます。詳細については，177 ページの「reconciliation_types.txt ファイル」を参照してください。
- **reconciliation_rules.txt**: 調整ルールが含まれています。詳細については，177 ページの「reconciliation_rules.txt ファイル（下位互換性用）」を参照してください。
- **transformations.txt:** CMDB 値をデータベース値に，またその逆に変換するために適用するコンバータを指定する変換ファイルです。詳細については，180 ページの「transformations.txt ファイル」を参照してください。

- **Discriminator.properties:** このファイルでは, サポートされる各 CI タイプを, 使用可能な対応するカンマ区切りリストにマップします。詳細については, 183 ページの「discriminator.properties ファイル」を参照してください。
- **Replication_config.txt:** このファイルには，CI および関係タイプのカンマ区切りリストが含まれていて，そのプロパティ条件はレプリケーション・プラグインでサポートされています。詳細については，184 ページの「replication_config.txt ファイル」を参照してください。
- **fixed_values.txt:** このファイルでは，特定の CIT に関する個別の属性に固定値を設定できます。詳細については，184 ページの「fixed_values.txt ファイル」を参照してください。

Hibernate 設定

- **persistence.xml:** 用意済みの Hibernate 設定を上書きするのに使います。詳細については, 181 ページの「persistence.xml ファイル」を参照してください。

単純構成

- **simplifiedConfiguration.xml:orm.xml**，**transformations.txt**，および **reconciliation_rules.txt** を機能の少ないものに置き換える構成ファイルです。詳細については，169 ページの「simplifiedConfiguration.xml ファイル」を参照してください。

adapter.conf ファイル

このファイルには次の設定が含まれています。

- **use.simplified.xml.config=false:true** : simplifiedConfiguration.xml を使用します。

注 : このファイルを使用することは，orm.xml，transformations.txt，および reconciliation_rules.txt を機能の少ないものに置き換えるということです。

- **dal.ids.chunk.size=300:** この値は変更しないでください。

➤ **dal.use.persistence.xml=false:true** : アダプタが persistence.xml から Hibernate 設定を読み込みます。

注 : Hibernate 設定を上書きするのはお勧めしません。

---

simplifiedConfiguration.xml ファイル

このファイルは，CMDB クラスをデータベース・テーブルに単純にマップする場合に使用されます。ファイルを編集するためのテンプレートにアクセスするには，［**アダプタ管理**］＞［**db-adapter**］＞［**構成ファイル**］に移動します。

本項の内容

➤ 169 ページの「simplifiedConfiguration.xml ファイル・テンプレート」

➤ 171 ページの「制限事項」

simplifiedConfiguration.xml ファイル・テンプレート

CMDB-class-name プロパティは multinode（マルチノード）タイプ（TQL でフェデレート CIT が接続するノード）です。

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
    <CMDB-class CMDB-class-name="node" default-table-name="[table_name]">
        <primary-key column-name="[column_name]"/>
```

reconciliation-by-two-nodes: 調整を行うには，1 つまたは 2 つのノードを使用します。この例では，調整で 2 つのノードを使用しています。

connected-node-CMDB-class-name: 調整 TQL で必要とされる第 2 クラス・タイプ。

CMDB-link-type: 調整 TQL で必要とされる関係タイプ。

link-direction: 調整 TQL における関係の方向（node から ip_address または ip_address から node）

```
<reconciliation-by-two-nodes  connected-node-CMDB-class-name="ip_address"
CMDB-link-type="containment" link-direction="main-to-connected">
```

調整式は OR の形式で，それぞれの OR には AND が含まれます。

is-ordered: 調整をオーダー形式で行うか, 通常の OR 比較で行うか決定します。

```
<or is-ordered="true">
```

調整プロパティをメイン・クラスから取得する場合は（マルチノード）, **attribute** を使用するか，**connected-node-attribute** タグを使用します。

ignore-case:true : UCMDB クラス・モデルのデータを RDBMS のデータと比較する場合，大文字 / 小文字は区別されません。

```
<attribute CMDB-attribute-name="name"
column-name="[column_name]" ignore-case="true"/>
```

カラム名は外部キー・カラム（マルチノード・プライマリ・キー・カラムを指示する値の含まれたカラム）の名前です。

マルチノード・プライマリ・キー・カラムが複数のカラムで構成されている場合は，各プライマリ・キー・カラムごとに 1 つ，複数の外部キー・カラムがある必要があります。

```
<foreign-primary-key column-name="[column_name]"
CMDB-class-primary-key-column="[column_name]"/>
```

プライマリ・キー・カラムが少ない場合は，このカラムを複製します。

```
<primary-key column-name="[column_name]"/>
```

from-CMDB-converter および **to-CMDB-converter** プロパティは，次のインタフェースを実行する Java クラスです。

- com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerFromExternalDB
- com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerToExternalDB

CMDB とデータベースの値が同じでない場合は，これらのコンバータを使います。たとえば，CMDB のノード名には，mer.com というサフィックスが付いています。

この例では，丸括弧内に記入された XML ファイル（**generic-enum-transformer-example.xml**）に従って列挙子を変換するのに，GenericEnumTransformer を使用しています。

```
        <attribute CMDB-attribute-name="[CMDB_attribute_name]"
column-name="[column_name]"
from-CMDB-converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.i
mpl.GenericEnumTransformer(generic-enum-transformer-example.xml)"
to-CMDB-converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(generic-enum-transformer-example.xml)"/>
        <attribute CMDB-attribute-name="[CMDB_attribute_name]"
column-name="[column_name]"/>
        <attribute CMDB-attribute-name="[CMDB_attribute_name]"
column-name="[column_name]"/>
    </class>
</generic-DB-adapter-config>
```

制限事項

- （データベース・ソースで）1 ノードの TQL クエリをマップする場合にのみ使用できます。たとえば，node > ticket と ticket TQL を実行できます。データベースからノードの階層を取り出すには，詳細な **orm.xml** ファイルを使う必要があります。
- 1 対多の関係だけがサポートされています。たとえば，各ノードに 1 つ以上のチケットを持ち込むことができます。複数のノードに属するチケットは持ち込めません。
- 同じクラスを異なるタイプの CMDB CIT に接続することはできません。たとえば，ticket を node に接続すると定義すると，application に接続することはできません。

orm.xml ファイル

このファイルは，CMDB CIT をデータベース・テーブルにマップするのに使用されます。

新規ファイルの作成に使用するテンプレートは，**C:¥hp¥UCMDB¥UCMDBServer¥runtime¥fcmdb¥CodeBase¥GenericDBAdapter¥META-INFMETA-INF** ディレクトリにあります。

デプロイしたアダプタ用の XML ファイルを編集するには，［**アダプタ管理**］＞［**db-adapter**］＞［**構成ファイル**］に移動します。

本項の内容

- 172 ページの「orm.xml ファイル・テンプレート」
- 176 ページの「複数の ORM ファイル」
- 176 ページの「命名規則」
- 176 ページの「テーブル名の代替としてのインライン SQL ステートメントの使用」

orm.xml ファイル・テンプレート

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" version="1.0" xsi:schemaLocation="http://
java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/persistence/
orm_1_0.xsd">
    <description>Generic DB adapter orm</description>
```

パッケージ名は変更しないでください。

```
    <package>generic_db_adapter</package>
```

entity:CMDB CIT 名。これは multinode（マルチノード）エンティティです。

クラスに **generic_db_adapter** というプレフィックスが含まれていることを確認します。

```
    <entity class="generic_db_adapter.node">
        <table name="[table_name]"/>
```

エンティティを複数のテーブルにマップする場合は，セカンダリ・テーブルを使用します。

```
<secondary-table name=""/>
<attributes>
```

識別子による単一テーブル継承の場合は，次のコードを使用します。

```
<inheritance strategy="SINGLE_TABLE"/>
<discriminator-value>node</discriminator-value>
<discriminator-column name="[column_name]"/>
```

id タグのある属性がプライマリ・キー・カラムです。これらのプライマリ・キー・カラムの命名規則は **idX**（id1，id2 など）であり，**X** はプライマリ・キーのカラム・インデックスです。

```
    <id name="id1">
```

プライマリ・キーのカラム名のみを変更します。

```
        <column updatable="false" insertable="false" name="[column_name]"/>
        <generated-value strategy="TABLE"/>
    </id>
```

basic:CMDB 属性を宣言するのに使用します。**name** および **column_name** プロパティだけを編集します。

```
    <basic name="name">
        <column updatable="false" insertable="false" name="[column_name]"/>
    </basic>
```

識別子による単一テーブル継承の場合は，拡張クラスを次のようにマップします。

```
    <entity name="[cmdb_class_name]" class="generic_db_adapter.nt" name="nt">

        <discriminator-value>nt</discriminator-value>
        <attributes/>
    </entity>
    <entity class="generic_db_adapter.unix" name="unix">
        <discriminator-value>unix</discriminator-value>
        <attributes/>
    </entity>
    <entity name="[CMDB_class_name]"
class="generic_db_adapter.[CMDB[cmdb_class_name]">
        <table name="[default_table_name]"/>
        <secondary-table name=""/>
        <attributes>
            <id name="id1">
                <column updatable="false" insertable="false" name="[column_name]"/>
                <generated-value strategy="TABLE"/>
            </id>
            <id name="id2">
                <column updatable="false" insertable="false" name="[column_name]"/>
                <generated-value strategy="TABLE"/>
            </id>
            <id name="id3">
                <column updatable="false" insertable="false" name="[column_name]"/>
                <generated-value strategy="TABLE"/>
            </id>
```

次の例に，プレフィックスのない CMDB 属性名を示します。

```
            <basic name="[CMDB_attribute_name]">
                <column updatable="false" insertable="false" name="[column_name]"/>
            </basic>
            <basic name="[CMDB_attribute_name]">
                <column updatable="false" insertable="false" name="[column_name]"/>
            </basic>
            <basic name="[CMDB_attribute_name]">
                <column updatable="false" insertable="false" name="[column_name]"/>
            </basic>
        </attributes>
    </entity>
```

これは関係エンティティです。命名規則は **end1Type_linkType_end2Type** です。この例では，**end1Type** は **node** で，**linkType** は **composition** です。

```
    <entity name="node_composition_[CMDB_class_name]"
class="generic_db_adapter.node_composition_[CMDB_class_name]">
        <table name="[default_table_name]"/>
        <attributes>
            <id name="id1">
                <column updatable="false" insertable="false" name="[column_name]"/>
                <generated-value strategy="TABLE"/>
            </id>
```

ターゲット・エンティティは，このプロパティが指示するエンティティです。この例では，**end1** が **node** エンティティにマップされます。

many-to-one: 1 つのノードに複数の関係を接続できます。

join-column: end1 ID（ターゲット・エンティティ ID）が含まれているカラム。

referenced-column-name: join カラムに使用する ID が含まれているターゲット・エンティティ（**node**）のカラム名。

```
            <many-to-one target-entity="node" name="end1">
                <join-column updatable="false" insertable="false"
referenced-column-name="[column_name]" name="[column_name]"/>
            </many-to-one>
```

one-to-one: 1 つの **[CMDB_class_name]** に 1 つの関係を接続できます。

```
            <one-to-one target-entity="[CMDB_class_name]" name="end2">
                <join-column updatable="false" insertable="false"
referenced-column-name="" name="[column_name]"/>
            </one-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

複数の ORM ファイル

複数のマッピング・ファイルがサポートされています。各マッピング・ファイル名は，最後に **orm.xml** を付けてください。マッピング・ファイルはすべて，アダプタの META-INF フォルダに置いてください。

命名規則

- 各エンティティでは，クラス・プロパティが generic_db_adapter というプレフィックスの名前プロパティに一致する必要があります。
- プライマリ・キー・カラムは，テーブルにあるプライマリ・キーの番号に従って，**idX** 形式（**X = 1**，**2**, ...,）の名前を取る必要があります。
- 属性名は大文字 / 小文字に関しても，クラス属性名と一致する必要があります。
- この関係名は end1Type_linkType_end2Type という形式を取ります。
- Java の予約語でもある CMDB CIT には，**gdba_** というプレフィックスを付けてください。たとえば，CMDB CIT **goto** の場合，ORM エンティティは **gdba_goto** という名前にします。

テーブル名の代替としてのインライン SQL ステートメントの使用

エンティティをデータベース・テーブルではなく，インライン select 句にマップすることができます。これは，データベースでビューを定義し，エンティティをこのビューにマップするのと同じです。たとえば，

```
<entity class="generic_db_adapter.node">
    <table name="(select d.id as id1, d.name as name , d.os as host_os from Device d)"/>
```

この例では，ノードの属性を，id，name，および os ではなく，id1，name，および host_os というカラムにマップする必要があります。

次の制限が適用されます。

- インライン SQL ステートメントは，JPA プロバイダとしての Hibernate を使用する場合にのみ使用できます。
- インライン SQL select 句を囲む丸括弧は必須です。
- **<schema>** 要素が **orm.xml** ファイルに含まれないようにします。Microsoft SQL Server 2005 の場合は, テーブル名を <schema>dbo</schema> でグローバルに定義するのではなく，すべてのテーブル名に dbo. というプレフィックスを付ける必要があることを意味します。

reconciliation_types.txt ファイル

このファイルは調整タイプを設定するのに使います。

ファイルの各行は，TQL クエリのフェデレート・データベース CIT に接続する CMDB CIT を表します。

reconciliation_rules.txt ファイル（下位互換性用）

このファイルは，アダプタに DBMappingEngine が設定されている場合，調整を実行するための調整ルールの設定に使用します。DBMappingEngine を使用しない場合，汎用的な UCMDB 調整メカニズムが使用されるため，このファイルを設定する必要はありません。

このファイルの各行がルールを示します。たとえば,

```
multinode[node] expression[^node.name OR ip_address.name] end1_type[node]
end2_type[ip_address] link_type[containment]
```

multinode（マルチノード）にはマルチノード名（TQL のフェデレート・データベース CIT に接続する CMDB CIT）を入力します。

この式には, 2 つのマルチノードが同じかどうかを判断するロジックが含まれています (一方のマルチノードは UCMDB にあり, もう一方はデータベース・ソースにあります)。

この式は ORs または ANDs で構成されています。

式の中で属性名に関する規則の部分は [className].[attributeName] です。たとえば, ip_address クラスの attributeName は ip_address.name と記述されます。

順序指定一致の場合は (最初の OR 副次式によってマルチノードが同じでないという応答が返されると，2 番目の OR 副次式は比較されません)，expression ではなく，ordered expression を使用します。

比較で大文字 / 小文字を無視するには，コントロール記号（^）を使います。

end1_type，end2_type，および link_type パラメータを使用するのは，調整 TQL が単なるマルチモードではなく, 2 つのノードが含まれている場合のみとなります。この場合，調整 TQL は end1_type > (link_type) > end2_type です。

関連レイアウトは式から取り出されるので，追加する必要はありません。

調整ルールのタイプ

調整ルールは OR および AND 条件の形を取ります。これらのルールはさまざまなノードに対して定義できます（たとえば，ノードは name from node AND/OR name from ip_address で識別されます）。

次のオプションで一致するものが見つかります。

- **順序一致 :** 調整式は左から右へ読みます。2 つの OR 副次式が値を持ち，それらが等しい場合は，同じであると見なされます。2 つの OR 副次式が値を持ち，それらが等しくない場合は，同じではないと見なされます。その他の場合は決まりがなく，次の OR 副次式で相等性がテストされます。

  node または ip_address の name: UCMDB とデータ・ソースの両方に name が含まれていて，それらが同じであれば，ノードは同じであるとみなされます。両方に name があっても同じでなければ，ノードは同じでないとみなされ，ip_address の name はテストされません。CMDB またはデータ・ソースに name of node がなければ，name of ip_address がチェックされます。

- **正規表現一致 :** OR 副次式のいずれかに同じものがあれば, CMDB とデータ・ソースは同じであるとみなされます。

  node または ip_address の name: name of node が一致しない場合は, name of ip_address で相等性がチェックされます。

複雑な調整の場合は，調整エンティティがクラス・モデルで関係のある複数の CIT（node など）としてモデル化され，スーパーセット・ノードのマッピングに，モデル化されたすべての CIT の関連属性がすべて含まれます。

注 : 結果として，データ・ソースの調整属性はすべて，同じプライマリ・キーを共有するテーブルにある必要があるという制限があります。

別の制限によって，調整 TQL には 2 つのノードしか持てません。たとえば，node > ticket TQL は，CMDB にノードがあり，データ・ソースにチケットがあります。

結果を調整するには，ノードおよび ip_address から name を取得する必要があります。

CMDB の name が *.m.com の形式である場合は，コンバータを使用して CMDB からフェデレート・データベースに，また逆の場合も同様にこれらの値を変換できます。

データベース・チケット・テーブルの node_id カラムを使用して，エンティティ間を接続します（関連付けの定義もノード・テーブルに作成できます）。

DB Node	
PK	node_id
	name

DB IP_Address	
PK	ip_id
	name

DB Ticket	
PK	ticket_id
	node_id

注 : 3 つのテーブルは CMDB データベースではなく，フェデレート RDBMS ソースの一部である必要があります。

---

transformations.txt ファイル

このファイルには，コンバータ定義がすべて含まれています。

この形式では，各行に新しい定義が含まれます。

transformations.txt ファイル・テンプレート

```
entity[[CMDB_class_name]] attribute[[CMDB_attribute_name]]
to_DB_class[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.
transform.impl.GenericEnumTransformer(generic-enum-transformer-example.xml)]
from_DB_class[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(generic-enum-transformer-example.xml)]
```

entity: orm.xml ファイルに表示されるエンティティ名。

attribute: orm.xml ファイルに表示される属性名。

to_DB_class: **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerToExternalDB** インタフェースを実装するクラスの完全な修飾名。丸括弧内の要素が，このクラス・コンストラクタに設定されます。このコンバータは，CMDB 値をデータベース値に変換する（**.com** というサフィックスを各ノード名に付加するなど）のに使用します。

from_DB_class:**com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerFromExternalDB** インタフェースを実装するクラスの完全な修飾名。丸括弧内の要素が，このクラス・コンストラクタに設定されます。このコンバータは，データベース値を CMDB 値に変換する（**.com** というサフィックスを各ノード名に付加するなど）のに使用します。

詳細については，184 ページの「用意済みのコンバータ」を参照してください。

persistence.xml ファイル

このファイルは標準設定の Hibernate 設定をオーバーライドしたり，用意されていないデータベース・タイプのサポートを追加するのに使用します（OOB データベース・タイプは Oracle Server，Microsoft MSSQL Server，および MySQL です）。

新しいデータベース・タイプをサポートする必要がある場合は，接続プール・プロバイダ（標準設定は c3p0）とデータベース用の JDBC ドライバを用意します（*.jar ファイルをアダプタ・フォルダに入れます）。

変更できる Hibernate 値をすべて確認するには，**org.hibernate.cfg.Environment** クラスをチェックします。

persistence.xml ファイルの例 :

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/
xml/ns/persistence
 http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
    <!-- Don't change this value -->
    <persistence-unit name="GenericDBAdapter">
        <properties>
            <!-- Don't change this value -->
            <property name="hibernate.archive.autodetection" value="class, hbm"/>
            <!--The driver class name"/-->
            <property name="hibernate.connection.driver_class"
value="com.mercury.jdbc.MercOracleDriver"/>
            <!--The connection url"/-->
            <property name="hibernate.connection.url" value="jdbc:mercury:oracle://
artist:1521;sid=cmdb2"/>
            <!--DB login credentials"/-->
            <property name="hibernate.connection.username" value="CMDB"/>
            <property name="hibernate.connection.password" value="CMDB"/>
            <!--connection pool properties"/-->
            <property name="hibernate.c3p0.min_size" value="5"/>
            <property name="hibernate.c3p0.max_size" value="20"/>
            <property name="hibernate.c3p0.timeout" value="300"/>
            <property name="hibernate.c3p0.max_statements" value="50"/>
            <property name="hibernate.c3p0.idle_test_period" value="3000"/>
            <!--The dialect to use-->
            <property name="hibernate.dialect"
value="org.hibernate.dialect.OracleDialect"/>
        </properties>
    </persistence-unit>
</persistence>
```

discriminator.properties ファイル

このファイルによって，サポートされている各 CI タイプ（orm.xml で識別子値としても使用されます）が，識別子カラムの対応する値のカンマ区切りリストにマップされます。

識別子マッピングの例：

discriminator.properties ファイルには次のコードが記述されています。

```
node=10001, 10005,10010,10011,10012
nt=10002,10003
unix=10004,10006,10008
```

orm.xml ファイルには次のコードが記述されています。

```
<entity class="generic_db_adapter.node">
    <table name="[table_name]"/>

    <inheritance strategy="SINGLE_TABLE"/>
    <discriminator-value>node</discriminator-value>
    <discriminator-column name="[discriminator_column]"/>

</entity>
<entity class="generic_db_adapter.nt" name="nt">
    <discriminator-value>nt</discriminator-value>
    <attributes/>
</entity>
<entity class="generic_db_adapter.unix" name="unix">
    <discriminator-value>unix</discriminator-value>
    <attributes/>
</entity>
```

［discriminator_column］属性は次のように計算されます。

- 対応するテーブルの識別子カラムに，特定のエントリを示す 10002 が含まれます。このエントリは **nt** CIT にマップされます。
- 対応するテーブルの識別子カラムに，特定のエントリを示す 10006 が含まれます。このエントリは **unix** CIT にマップされます。

- 対応するテーブルの識別子カラムに，特定のエントリを示す 10010 が含まれます。このエントリは **node** CIT にマップされます。

node CIT は **nt** と **unix** の親でもあることに注意してください。

replication_config.txt ファイル

このファイルには, CI および関係タイプのカンマ区切りリストが含まれていて，そのプロパティ条件はレプリケーション・プラグインでサポートされています。詳細については，188 ページの「プラグイン」を参照してください。

fixed_values.txt ファイル

このファイルでは，特定の CIT に関する個別の属性に固定値を設定できます。このような方法で，これらの各属性には，データベースに保管されていない固定値を割り当てることができます。

このファイルには，0 個以上のエントリが次の形式で含まれます。

```
entity[<entityName>] attribute[<attributeName>] value[<value>]
```

たとえば,

```
entity[ip_address] attribute[ip_domain] value[DefaultDomain]
```

用意済みのコンバータ

次のコンバータ（変換子）を使って，フェデレート・クエリおよびレプリケーション・ジョブをデータベースのデータに，またはその逆に変換できます。

本項の内容

enum-transformer コンバータ

このコンバータでは, 入力パラメータとして与えられる XML ファイルを使用します。

XML ファイルはハードコードの CMDB 値とデータベース値 (enum) 間をマップします。いずれかの値が存在しない場合は，同じ値を返すか，NULL を返すか，例外処理を実行するか選択できます。

各エンティティ属性ごとに 1 つの XML マッピング・ファイルを使用します。

注 : このコンバータは，**transformations.txt** ファイルの to_DB_class および from_DB_class フィールドに使用できます。

入力ファイル XSD の例 :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:element name="enum-transformer">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="value" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="DB-type" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="integer"/>
                        <xs:enumeration value="long"/>
                        <xs:enumeration value="float"/>
                        <xs:enumeration value="double"/>
                        <xs:enumeration value="boolean"/>
                        <xs:enumeration value="string"/>
                        <xs:enumeration value="date"/>
                        <xs:enumeration value="xml"/>
                        <xs:enumeration value="bytes"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="CMDB-type" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="integer"/>
                        <xs:enumeration value="long"/>
```

```
                <xs:enumeration value="float"/>
                <xs:enumeration value="double"/>
                <xs:enumeration value="boolean"/>
                <xs:enumeration value="string"/>
                <xs:enumeration value="date"/>
                <xs:enumeration value="xml"/>
                <xs:enumeration value="bytes"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="non-existing-value-action" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="return-null"/>
                <xs:enumeration value="return-original"/>
                <xs:enumeration value="throw-exception"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="value">
  <xs:complexType>
    <xs:attribute name="CMDB-value" type="xs:string" use="required"/>
    <xs:attribute name="external-DB-value" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

sys 値を System 値に変換する例：

この例では，CMDB の sys 値がフェデレート・データベースの System 値に変換され，フェデレート・データベースの System 値が CMDB の sys 値に変換されます。

XML ファイルに値がない場合は（文字列 demo など），コンバータが受信する同じ入力値を返します。

```
<enum-transformer CMDB-type="string" DB-type="string" non-existing-value-action="return-original"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../
META-CONF/generic-enum-transformer.xsd">
    <value CMDB-value="sys" external-DB-value="System"/>
</enum-transformer>
```

SuffixTransformer コンバータ

このコンバータは，CMDB またはフェデレート・データベース・ソース値にサフィックスを追加またはこれらから削除するのに使います。

2 つの実装があります。

- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdbAddSuffixTransformer**: フェデレート・データベース値から CMDB 値に変換するときにサフィックス（入力として指定）を追加し，CMDB 値からフェデレート・データベース値に変換するときにサフィックスを削除します。
- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdbRemoveSuffixTransformer**: フェデレート・データベース値から CMDB 値に変換するときにサフィックス（入力として指定）を削除し，CMDB 値からフェデレート・データベース値に変換するときにサフィックスを追加します。

PrefixTransformer コンバータ

このコンバータは，CMDB またはフェデレート・データベース値にプレフィックスを追加または削除するのに使います。

2 つの実装があります。

- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdbAddPrefixTransformer**: フェデレート・データベース値から CMDB 値に変換するときにプレフィックス（入力として指定）を追加し，CMDB 値からフェデレート・データベース値に変換するときにプレフィックスを削除します。
- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdbRemovePrefixTransformer**: フェデレート・データベース値から CMDB 値に変換するときにプレフィックス（入力として指定）を削除し，CMDB 値からフェデレート・データベース値に変換するときにプレフィックスを追加します。

BytesToStringTransformer コンバータ

このコンバータは，CMDB のバイト配列をフェデレート・データベース・ソースの文字列表現に変換するのに使います。

このコンバータは **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.CmdbToAdapterBytesToStringTransformer** です。

プラグイン

汎用データベース・アダプタでは，次のプラグインをサポートしています。

- トポロジを完全に同期化する任意プラグイン。
- トポロジの変更を同期化する必須プラグイン。
- レイアウトを同期化する任意プラグイン。
- サポートされている同期化のクエリを取得する任意プラグイン。このプラグインが定義されていないと，すべての TQL 名が返されます。
- TQL 定義および TQL 結果を変更する内部の任意プラグイン。
- レイアウト要求および CI 結果を変更する内部の任意プラグイン。
- レイアウト要求および関係結果を変更する内部の任意プラグイン。

プラグインの実装とデプロイの詳細については，142 ページの「プラグインの実装」を参照してください。

設定例

本項では設定例を示します。

本項の内容

➤ 196 ページの「複数のカラムが含まれているプライマリ・キーの使い方」

➤ 198 ページの「変換の仕方」

使用例

使用例 :TQL は次のとおりです。

node >（composition）> card

説明 :

node は，CMDB エンティティです

card はフェデレート・データベース・ソース・エンティティです

composition はそれらの関係です

この例は ED データベースに対して実行されます。ED nodes は Device テーブルに保管され，card は hwCards テーブルに保管されます。次の例では，card がいつも同じ方法でマップされます。

単一ノード調整

この例では，name プロパティに対して調整が実行されます。

簡単な定義

調整は node 単位で行われ，**CMDB-class** という特別なタグで強調されます。

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
    <CMDB-class CMDB-class-name="node" default-table-name="Device">
        <primary-key column-name="Device_ID"/>
        <reconciliation-by-single-node>
            <or>
                <attribute CMDB-attribute-name="name" column-name="Device_Name"/>
            </or>
        </reconciliation-by-single-node>
    </CMDB-class>
    <class CMDB-class-name="card" default-table-name="hwCards"
connected-CMDB-class-name="node" link-class-name="composition">
        <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-column="Device_ID"/>
        <primary-key column-name="hwCards_Seq"/>
        <attribute CMDB-attribute-name="card_class" column-name="hwCardClass"/>
        <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor"/>
        <attribute CMDB-attribute-name="card_name" column-name="hwCardName"/>
    </class>
</generic-DB-adapter-config>
```

詳細な定義

orm.xml ファイル

関係マッピングの追加に注意してください。詳細については，172 ページの「orm.xml ファイル」の定義セクションを参照してください。

orm.xml ファイルの例 :

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/
persistence/orm http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
    <description>Generic DB adapter orm</description>
    <package>generic_db_adapter</package>
```

```
<entity class="generic_db_adapter.node">
    <table name="Device"/>
    <attributes>
        <id name="id1">
            <column name="Device_ID" insertable="false" updatable="false"/>
            <generated-value strategy="TABLE"/>
        </id>
        <basic name="name">
            <column name="Device_Name"/>
        </basic>
    </attributes>
</entity>
<entity class="generic_db_adapter.card">
    <table name="hwCards"/>
    <attributes>
        <id name="id1">
            <column name="hwCards_Seq" insertable="false" updatable="false"/>
            <generated-value strategy="TABLE"/>
        </id>
        <basic name="card_class">
            <column name="hwCardClass" insertable="false" updatable="false"/>
        </basic>
        <basic name="card_vendor">
            <column name="hwCardVendor" insertable="false" updatable="false"/>
        </basic>
        <basic name="card_name">
            <column name="hwCardName" insertable="false" updatable="false"/>
        </basic>
    </attributes>
</entity>
<entity class="generic_db_adapter.node_composition_card">
    <table name="hwCards"/>
    <attributes>
        <id name="id1">
            <column name="hwCards_Seq" insertable="false" updatable="false"/>
            <generated-value strategy="TABLE"/>
        </id>
        <many-to-one name="end1" target-entity="node">
            <join-column name="Device_ID" insertable="false" updatable="false"/>
        </many-to-one>
```

```
            <one-to-one name="end2" target-entity="card">
                <join-column name="hwCards_Seq" referenced-column-name="hwCards_Seq"
insertable="false" updatable="false"/>
            </one-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

reconciliation_types.txt ファイル

詳細については，177 ページの「reconciliation_types.txt ファイル」を参照してください。

```
node
```

reconciliation_rules.txt ファイル

詳細については，177 ページの「reconciliation_rules.txt ファイル（下位互換性用）」を参照してください。

```
multinode[node] expression[node.name]
```

transformation.txt ファイル

この例では値を変換する必要がないので，このファイルは空のままです。

2 ノード調整

この例では，node および ip_address の name プロパティに従ってさまざまなバリエーションで調整が計算されます。

調整 TQL は，**node** >（**containment**）> **ip_address** です。

簡単な定義

この調整は node または ip_address の name 単位で行われます。

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
    <CMDB-class CMDB-class-name="node" default-table-name="Device">
        <primary-key column-name="Device_ID"/>
        <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address"
CMDB-link-type="containment">
            <or>
                <attribute CMDB-attribute-name="name" column-name="Device_Name"/>
                <connected-node-attribute CMDB-attribute-name="name"
column-name="Device_PreferredIPAddress"/>
            </or>
        </reconciliation-by-two-nodes>
    </CMDB-class>
    <class CMDB-class-name="card" default-table-name="hwCards"
connected-CMDB-class-name="node" link-class-name="containment">
        <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-column="Device_ID"/>
        <primary-key column-name="hwCards_Seq"/>
        <attribute CMDB-attribute-name="card_class" column-name="hwCardClass"/>
        <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor"/>
        <attribute CMDB-attribute-name="card_name" column-name="hwCardName"/>
    </class>
</generic-DB-adapter-config>
```

この調整は node および ip_address の name 単位で行われます。

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
    <CMDB-class CMDB-class-name="node" default-table-name="Device">
        <primary-key column-name="Device_ID"/>
        <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address"
CMDB-link-type="containment">
            <and>
                <attribute CMDB-attribute-name="name" column-name="Device_Name"/>
                <connected-node-attribute CMDB-attribute-name="name"
column-name="Device_PreferredIPAddress"/>
            </and>
        </reconciliation-by-two-nodes>
    </CMDB-class>
    <class CMDB-class-name="card" default-table-name="hwCards"
connected-CMDB-class-name="node" link-class-name="containment">
        <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-column="Device_ID"/>
        <primary-key column-name="hwCards_Seq"/>
        <attribute CMDB-attribute-name="card_class" column-name="hwCardClass"/>
        <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor"/>
        <attribute CMDB-attribute-name="card_name" column-name="hwCardName"/>
    </class>
</generic-DB-adapter-config>
```

この調整は ip_address の name 単位で行われます。

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
    <CMDB-class CMDB-class-name="node" default-table-name="Device">
        <primary-key column-name="Device_ID"/>
        <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address"
CMDB-link-type="containment">
            <or>
                <connected-node-attribute CMDB-attribute-name="name"
column-name="Device_PreferredIPAddress"/>
            </or>
        </reconciliation-by-two-nodes>
    </CMDB-class>
    <class CMDB-class-name="card" default-table-name="hwCards"
connected-CMDB-class-name="node" link-class-name="containment">
        <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-column="Device_ID"/>
        <primary-key column-name="hwCards_Seq"/>
        <attribute CMDB-attribute-name="card_class" column-name="hwCardClass"/>
        <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor"/>
        <attribute CMDB-attribute-name="card_name" column-name="hwCardName"/>
    </class>
</generic-DB-adapter-config>
```

詳細な定義

orm.xml ファイル

このファイルには調整式が定義されていないので，どの調整式にも同じバージョンを使用する必要があります。

reconciliation_types.txt ファイル

詳細については，177 ページの「reconciliation_types.txt ファイル」を参照してください。

```
node
```

reconciliation_rules.txt ファイル

詳細については，177 ページの「reconciliation_rules.txt ファイル（下位互換性用）」を参照してください。

```
multinode[node] expression[ip_address.name OR node.name] end1_type[node]
end2_type[ip_address] link_type[containment]
```

```
multinode[node] expression[ip_address.name AND node.name] end1_type[node]
end2_type[ip_address] link_type[containment]
```

```
multinode[node] expression[ip_address.name] end1_type[node] end2_type[ip_address]
link_type[containment]
```

transformation.txt ファイル

この例では値を変換する必要がないので，このファイルは空のままです。

複数のカラムが含まれているプライマリ・キーの使い方

プライマリ・キーが複数のカラムで構成されている場合は，次のコードを XML 定義に追加します。

簡単な定義

複数のプライマリ・キー・タグがあり，各カラムごとにタグがあります。

```
    <class CMDB-class-name="card" default-table-name="hwCards"
connected-CMDB-class-name="node" link-class-name="containment">
        <foreign-primary-key column-name="Device_ID"
CMDB-class-primary-key-column="Device_ID"/>
        <primary-key column-name="Device_ID"/>
        <primary-key column-name="hwBusesSupported_Seq"/>
        <primary-key column-name="hwCards_Seq"/>
        <attribute CMDB-attribute-name="card_class" column-name="hwCardClass"/>
        <attribute CMDB-attribute-name="card_vendor"
column-name="hwCardVendor"/>
        <attribute CMDB-attribute-name="card_name" column-name="hwCardName"/>
    </class>
```

詳細な定義

orm.xml ファイル

プライマリ・キー・カラムにマップされる新しい id エンティティを追加します。この id エンティティを使用するエンティティで，特別なタグを追加する必要があります。

このようなプライマリ・キーに外部キー（join-column タグ）を使用する場合は，外部キーの各カラムをプライマリ・キーのカラムにマップする必要があります。

詳細については，172 ページの「orm.xml ファイル」を参照してください。

orm.xml ファイルの例 :

```
<   <entity class="generic_db_adapter.card">
    <table name="hwCards"/>
    <attributes>
        <id name="id1">
            <column name="Device_ID" insertable="false" updatable="false"/>
            <generated-value strategy="TABLE"/>
        </id>
        <id name="id2">
            <column name="hwBusesSupported_Seq" insertable="false" updatable="false"/>
            <generated-value strategy="TABLE"/>
        </id>
        <id name="id3">
            <column name="hwCards_Seq" insertable="false" updatable="false"/>
            <generated-value strategy="TABLE"/>
        </id>
.
.
.
    <entity class="generic_db_adapter.node_containment_card">
    <table name="hwCards"/>
    <attributes>
        <id name="id1">
            <column name="Device_ID" insertable="false" updatable="false"/>
            <generated-value strategy="TABLE"/>
        </id>
        <id name="id2">
            <column name="hwBusesSupported_Seq" insertable="false" updatable="false"/>
            <generated-value strategy="TABLE"/>
        </id>
        <id name="id3">
            <column name="hwCards_Seq" insertable="false" updatable="false"/>
```

```
            <generated-value strategy="TABLE"/>
        </id>
        <many-to-one name="end1" target-entity="node">
            <join-column name="Device_ID" insertable="false" updatable="false"/>
        </many-to-one>
        <one-to-one name="end2" target-entity="card">
            <join-column name="Device_ID" referenced-column-name="Device_ID" insertable="false"
updatable="false"/>
            <join-column name="hwBusesSupported_Seq"
referenced-column-name="hwBusesSupported_Seq" insertable="false" updatable="false"/>
            <join-column name="hwCards_Seq" referenced-column-name="hwCards_Seq"
insertable="false" updatable="false"/>
        </one-to-one>
    </attributes>
  </entity>
</entity-mappings>
```

変換の仕方

次の例では，一般的な **enum** 変換子によって，name カラムで値 1，2，3 から値 a，b，c にそれぞれ変換されます。

マッピング・ファイルは generic-enum-transformer-example.xml です。

```
<enum-transformer CMDB-type="string" DB-type="string"
non-existing-value-action="return-original" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:noNamespaceSchemaLocation="../META-CONF/
generic-enum-transformer.xsd">
    <value CMDB-value="1" external-DB-value="a"/>
    <value CMDB-value="2" external-DB-value="b"/>
    <value CMDB-value="3" external-DB-value="c"/>
</enum-transformer>
```

簡単な定義

```
    <CMDB-class CMDB-class-name="node" default-table-name="Device">
        <primary-key column-name="Device_ID"/>
        <reconciliation-by-two-nodes  connected-node-CMDB-class-name="ip_address"
CMDB-link-type="containment">
            <or>
                <attribute  CMDB-attribute-name="name"  column-name="Device_Name"
from-CMDB-converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.i
mpl.GenericEnumTransformer(generic-enum-transformer-example.xml)"
to-CMDB-converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(generic-enum-transformer-example.xml)"/>
                <connected-node-attribute CMDB-attribute-name="name"
column-name="Device_PreferredIPAddress"/>
            </or>
        </reconciliation-by-two-nodes>
    </CMDB-class>
.
.
.
```

詳細な定義

transformation.txt ファイルのみに変更があります。

transformation.txt ファイル

属性名とエンティティ名を orm.xml ファイルと同じにします。

```
entity[node] attribute[name]
to_DB_class[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.Generic
EnumTransformer(generic-enum-transformer-example.xml)]
from_DB_class[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.Gene
ricEnumTransformer(generic-enum-transformer-example.xml)]
```

アダプタ・ログ・ファイル

計算フロー，アダプタ・ライフサイクルを理解したり，デバッグ情報を表示するには，次のログ・ファイルを参照します。

本項の内容

➤ 200 ページの「ログ・レベル」

➤ 201 ページの「ログの保管場所」

ログ・レベル

各ログのログ・レベルを設定できます。

テキスト・エディタで，**C:¥hp¥UCMDB¥UCMDBServer¥conf¥log¥fcmdb.gdba.properties** ファイルを開きます。

標準設定のログ・レベルは **ERROR** です。

```
#loglevel can be any of DEBUG INFO WARN ERROR FATAL
loglevel=ERROR
```

➤ すべてのログ・ファイルのログ・レベルを上げるには，**loglevel=ERROR** を **loglevel=DEBUG** または **loglevel=INFO** に変更します。

➤ 特定ファイルのログ・レベルを変更するには，特定の **log4j** カテゴリ行を適宜変更します。たとえば，fcmdb.gdba.dal.sql.log のログ・レベルを **INFO** に変更するには，次の行を変更します。

```
log4j.category.fcmdb.gdba.dal.SQL=${loglevel},fcmdb.gdba.dal.SQL.appender
```

次のように変更します。

```
log4j.category.fcmdb.gdba.dal.SQL=INFO,fcmdb.gdba.dal.SQL.appender
```

ログの保管場所

ログ・ファイルは，**C:¥hp¥UCMDB¥UCMDBServer¥runtime¥log** ディレクトリにあります。

➤ **Fcmdb.gdba.log**

アダプタ・ライフサイクル・ログ。アダプタの開始または停止，当該アダプタでサポートしている CIT に関する詳細を提供します。

開始エラー（アダプタの読み込み / 読み込み解除）を参照してください。

➤ **fcmdb.log**

例外を参照してください。

➤ **cmdb.log**

例外を参照してください。

➤ **Fcmdb.gdba.mapping.engine.log**

マッピング・エンジン・ログ : マッピング・エンジンが使用している調整 TQL，および接続フェーズで比較される調整トポロジに関する詳細を提供します。

データベースに関連 CI があることが分かっていても，TQL クエリが結果を提供しないか，結果が予期しないものである場合は，このログを参照します（調整をチェックします）。

➤ **Fcmdb.gdba.TQL.log**

TQL ログ : TQL クエリとその結果に関する詳細を提供します。

TQL クエリが結果を返さず，マッピング・エンジン・ログがフェデレート・データ・ソースに結果がないことを示す場合は，このログを参照します。

➤ **Fcmdb.gdba.dal.log**

DAL ライフサイクル・ログ : CIT の生成に関する詳細とデータベース接続の詳細を提供します。

データベースに接続できない場合，またはクエリでサポートされていない CIT または属性がある場合は，このログを参照します。

- **Fcmdb.gdba.dal.command.log**

  DAL 動作ログ：呼び出された DAL 内部動作に関する詳細を提供します（このログは cmdb.dal.command.log と似ています）。

- **Fcmdb.gdba.dal.SQL.log**

  DAL SQL クエリ・ログ：呼び出された JPAQL（オブジェクト指向 SQL クエリ）とその結果に関する詳細を提供します。

  データベースに接続できない場合，またはクエリでサポートされていない CIT または属性がある場合は，このログを参照します。

- **Fcmdb.gdba.hibrnate.log**

  Hibernate ログ：実行された SQL クエリ，各 JPAQL から SQL の解析，クエリの結果，Hibernate キャッシュに関するデータなどの詳細を提供します。Hibernate の詳細については，123 ページの「JPA プロバイダとしての Hibernate」を参照してください。

外部参照先

JavaBeans 3.0 の仕様の詳細については，http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html を参照してください。

トラブルシューティングと制限事項

このセクションでは，汎用データベース・アダプタのトラブルシューティングと制限事項について説明します。

一般的な制限事項

- SQL Server NTLM 認証はサポートされていません。
- アダプタ・パッケージを更新する場合，テンプレート・ファイルの編集には Microsoft Corporation が提供するメモ帳（すべてのバージョン）ではなく，Notepad++ や UltraEdit などのサードパーティ製テキスト・エディタを使用します。こうすることで，準備したパッケージのデプロイメントが失敗する原因となる特殊な記号が使用されなくなります。

JPA 制限事項

- すべてのテーブルには，プライマリ・キー・カラムがある必要があります。
- CMDB クラスの属性名は，JavaBeans の命名規則に従う必要があります（たとえば，名前の最初は小文字である必要があります）。
- クラス・モデルの 1 つの関係と接続する 2 つの CI は，データベースに直接関連する必要があります（たとえば，node を ticket に接続する場合は，それらを接続する外部キーまたはリンクがある必要があります）。
- 同じ CIT にマップされる複数のテーブルでは，同じプライマリ・キー・テーブルを共有する必要があります。

機能上の制限事項

- CMDB とフェデレート CIT との間には，手動で関係を作成できません。仮想の関係を定義するには，特別な関係ロジックを定義する必要があります（この関係はフェデレート・クラスのプロパティをベースにできます）。
- フェデレート CIT は，影響ルール内でトリガ CIT にすることはできませんが，影響分析 TQL クエリに含めることはできます。
- フェデレート CIT はエンリッチメント TQL の一部になりますが，エンリッチメントを実行するノードとして使用することはできません（フェデレート CIT を追加，更新，または削除できません）。
- 条件でクラス修飾子を使用することはサポートされていません。
- サブグラフはサポートされていません。
- 複合関係はサポートされていません。
- 外部の CI CMDB id を構成するのは，そのキー属性ではなく，プライマリ・キーです。
- bytes タイプのカラムは，Microsoft SQL Server でプライマリ・キー・カラムとして使用できません。

- **orm.xml** ファイルでは，最初のエンドがフェデレート CIT でもう一方の端が CMDB である場合は，連携の仮想関係を逆順でマップする必要があります。この制限事項は，ポピュレーション・ジョブには該当しません。
- TQL クエリ計算は，フェデレート・ノードに定義されている属性条件の名前が **orm.xml** ファイル内でマップされていない場合，失敗します。

5

Java アダプタの開発

本章の内容

概念

タスク

参照先

概念

Federation Framework の概要

注 :

- 「**関係**」は，「**リンク**」と同じ意味です。
- 「**CI**」は，「**オブジェクト**」と同じ意味です。
- 「グラフ」は，ノードとリンクの集合です。
- 定義と用語の一覧については，『HP UCMDB 管理ガイド』の「用語集」を参照してください。

Federation Framework 機能は，API を使用してフェデレート・ソースから情報を取得します。Federation Framework は主に次の 3 つの機能を提供します。

- オンザフライ**連携** : すべてのクエリが元のデータ・リポジトリに対して実行され，結果がすぐに CMDB に作成されます。
- **ポピュレーション** : データ（トポロジ・データおよび CI プロパティ）を外部データ・ソースから CMDB にポピュレートします。
- **データ・プッシュ** : データ（トポロジ・データおよび CI プロパティ）をリモートのデータ・ソースからローカルの CMDB にプッシュします。

すべてのアクション・タイプについて，データ・リポジトリごとにアダプタが必要です。アダプタは，データ・リポジトリの特定の機能を提供し，必要なデータの取得，更新，またはその両方を実行できます。データ・リポジトリに対する要求はすべて，アダプタを介して行われます。

本項の内容 :

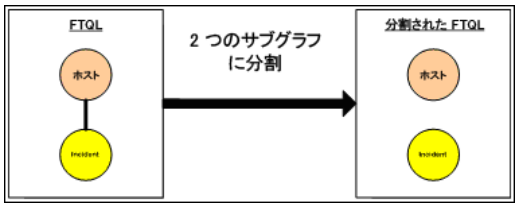
オンザフライ・フェデレーション

フェデレート TQL クエリを使って，任意の外部データ・リポジトリからデータを複製せずに取得できます。

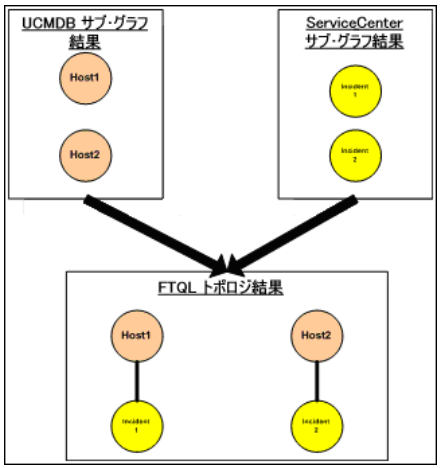
フェデレート TQL クエリは, 外部データ・リポジトリを表すアダプタを使って, さまざまな外部データ・リポジトリからの CI と UCMDB CI の間に適切な外部関係を作成します。

オンザフライ連携フローの例：

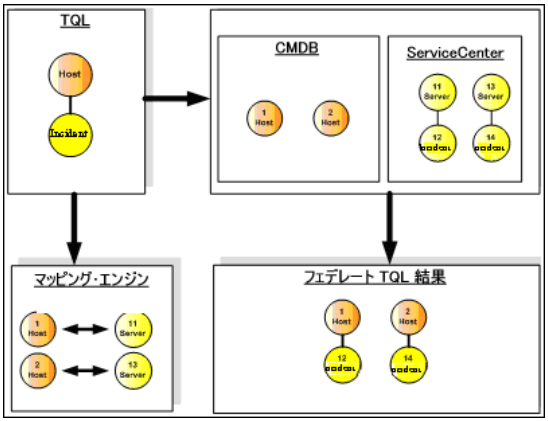
1. Federation Framework は，1 つのフェデレート TQL クエリを複数のサブグラフに分割します。サブグラフ内のすべてのノードは同じデータ・リポジトリを参照します。各サブグラフは，virtual relationship（仮想関係）によってほかのサブグラフに接続されます（ただし，サブグラフ自体に仮想関係は含まれていません）。



2. フェデレート TQL クエリをサブグラフに分割した後で, Federation Framework は各サブグラフのトポロジを計算し，該当するノード間の仮想関係を作成することにより，該当する 2 つのサブグラフを接続します。



3 フェデレート TQL のトポロジを計算した後で, Federation Framework はトポロジ結果のレイアウトを取得します。



データ・プッシュ

データ・プッシュ・フローを使用して，現在のローカル CMDB のデータをリモート・サービスまたはターゲット・データ・リポジトリに同期します。

データ・プッシュでは，データ・リポジトリはソース（ローカル CMDB）およびターゲットの 2 つのカテゴリに分けられます。データは，ソース・データ・リポジトリから取得され，ターゲット・データ・リポジトリに更新されます。データ・プッシュ・プロセスはクエリ名に基づいて行われます。つまり，ソース・データ・リポジトリ（ローカル CMDB）とターゲット・データ・リポジトリ間でデータが同期され，ローカル CMDB の TQL クエリ名によってデータが取得されます。

データ・プッシュ・プロセスのフローには，次の手順が含まれています。

1 ソース・データ・リポジトリから署名を含むトポロジ結果を取得します。

2 新しい結果を前の結果と比較します。

3 変更された結果のみに関して，CI と関係の完全なレイアウト（つまり，すべての CI プロパティ）を取得します。

4 受け取った CI と関係の完全なレイアウトを使用してターゲット・データ・リポジトリを更新します。ソース・データ・リポジトリで CI または関係が削除され，クエリが排他的である場合は，レプリケーション・プロセスによって，ターゲット・データ・リポジトリでもその CI または関係が削除されます。

CMDB には 2 つの非表示データ・ソース（**hiddenRMIDataSource** および **hiddenChangesDataSource**）があり，これらは常にデータ・プッシュ・フロー内のソース・データ・ソースです。データ・プッシュ・フローに新しいアダプタを実装する場合，実装する必要があるのはターゲット・アダプタのみです。

ポピュレーション

ポピュレーション・フローを使用して，CMDB に外部ソースからのデータをポピュレートします。

フローでは常に 1 つの「ソース」データ・ソースを使用してデータを取得し，取得したデータをディスカバリ・ジョブのフローへの同様のプロセス内のプローブにプッシュします。

ポピュレーション・フローに新しいアダプタを実行する場合，Data Flow Probe がターゲットとして機能するため，実装する必要があるのはソース・アダプタのみです。

ポピュレーション・フロー内のアダプタはプローブで実行されます。デバッグ処理およびログ処理は，CMDB でなくプローブ上で実行する必要があります。

ポピュレーション・フローはクエリ名に基づいて行われます。つまり，ソース・データ・リポジトリと Data Flow Probe の間でデータの同期が行われ，ソース・データ・リポジトリ内のクエリ名によってデータが取得されます。たとえば，UCMDB では TQL クエリの名前がクエリ名です。ただし，別のデータ・リポジトリでは，データを返すコード名がクエリ名である可能性があります。アダプタは，クエリ名を正しく処理できるように設計されています。

各ジョブは，排他的ジョブとして定義できます。これは，ジョブ結果の CI と関係がローカル CMDB で一意であり，ほかのクエリではそれらをターゲットに提供できないことを意味します。ソース・データ・リポジトリのアダプタは，特定のクエリをサポートし，そのデータ・リポジトリからデータを取得できます。ターゲット・データ・リポジトリのアダプタは，取得したデータをそのデータ・リポジトリ上で更新できます。

SourceDataAdapter フロー

➤ ソース・データ・リポジトリから署名を含むトポロジ結果を取得します。

➤ 新しい結果を前の結果と比較します。

- 変更された結果のみに関して，CI と関係の完全なレイアウト（つまり，すべての CI プロパティ）を取得します。
- 受け取った CI と関係の完全なレイアウトを使用してターゲット・データ・リポジトリを更新します。ソース・データ・リポジトリで CI または関係が削除され，クエリが排他的である場合は，レプリケーション・プロセスによって，ターゲット・データ・リポジトリでもその CI または関係が削除されます。

SourceChangesDataAdapter フロー

- 特定の最終日以降に発生したトポロジ結果を取得します。
- 変更された結果のみに関して，CI と関係の完全なレイアウト（つまり，すべての CI プロパティ）を取得します。
- 受け取った CI と関係の完全なレイアウトを使用してターゲット・データ・リポジトリを更新します。ソース・データ・リポジトリで CI または関係が削除され，クエリが排他的である場合は，レプリケーション・プロセスによって，ターゲット・データ・リポジトリでもその CI または関係が削除されます。

PopulateDataAdapter フロー

- 要求されたレイアウト結果を含むトポロジを完全に取得します。
- トポロジ・チャンク・メカニズムを使用して，チャンク内のデータを取得します。
- プローブにより，以前の実行ですでに取得されたデータをフィルタします。
- 受け取った CI と関係のレイアウトを使用してターゲット・データ・リポジトリを更新します。ソース・データ・リポジトリで CI または関係が削除され，クエリが排他的である場合は，レプリケーション・プロセスによって，ターゲット・データ・リポジトリでもその CI または関係が削除されます。

PopulateChangesDataAdapter フロー

- 要求されたレイアウト結果で前回の実行以降に変更されたものを含むトポロジを取得します。
- トポロジ・チャンク・メカニズムを使用して，チャンク内のデータを取得します。
- プローブにより，以前の実行（このフローを含む）ですでに取得されたデータをフィルタします。

➤ 受け取った CI と関係のレイアウトを使用してターゲット・データ・リポジトリを更新します。ソース・データ・リポジトリで CI または関係が削除され，クエリが排他的である場合は，レプリケーション・プロセスによって，ターゲット・データ・リポジトリでもその CI または関係が削除されます。

アダプタおよびマッピングの Federation Framework とのやり取り

アダプタは，外部データ（UCMDB に保存されないデータ）を表す UCMDB のエンティティです。フェデレート・フローでは，外部データ・ソースとのすべてのやり取りがアダプタを介して行われます。Federation Framework のやり取りのフローとアダプタ・インタフェースは，レプリケーションとフェデレート TQL クエリで異なります。

本項の内容：

➤ 213 ページの「アダプタのライフサイクル」

➤ 213 ページの「アダプタの assist メソッド」

アダプタのライフサイクル

アダプタ・インスタンスは，外部データ・リポジトリごとに作成されます。アダプタは，そのアダプタに最初に適用されたアクション（「calculate TQL」や「retrieve/update data」など）によってそのライフサイクルを開始します。**start** メソッドが呼び出されると，アダプタはデータ・リポジトリの設定やログ機能などの環境情報を受け取ります。アダプタのライフサイクルは，設定からデータ・リポジトリが削除され，**shutdown** メソッドが呼び出されたときに終了します。つまり，アダプタはステートフルであり，必要な場合は外部データ・リポジトリへの接続をアダプタに含めることができます。

アダプタの assist メソッド

アダプタには，外部データ・リポジトリ設定を追加できる複数の assist メソッドがあります。これらのメソッドは，アダプタのライフサイクルには含まれず，呼び出すたびに新しいアダプタを作成します。

- 最初のメソッドは，特定の設定のために外部データ・リポジトリへの接続をテストします。testConnection は，アダプタのタイプに応じて，UCMDB サーバまたは Data Flow Probe のいずれかで実行できます。
- 2 番目のメソッドは，ソース・アダプタにのみ関係し，レプリケーション用のサポートされているクエリを返します（このメソッドはプローブでのみ実行されます）。
- 3 番目のメソッドは連携フローおよびポピュレーション・フローにのみ関係しており，外部データ・リポジトリによりサポートされている external class（外部クラス）を返します（このメソッドは UCMDB サーバで実行されます）。

これらのメソッドはすべて，インテグレーション設定を作成または表示するときに使用されます。

フェデレート TQL クエリ用の Federation Framework フロー

本項の内容

定義と用語

調整データ：CMDB と外部データ・リポジトリから取得された特定タイプの CI を照合するためのルール。調整ルールには次の 3 種類があります。

- **ID 調整：**これは，外部データ・リポジトリに調整オブジェクトの CMDB ID が含まれている場合にのみ使用されます。
- **プロパティ調整：**これは，調整 CI タイプのプロパティによって照合が行われる場合にのみ使用されます。
- **トポロジ調整：**これは，調整 CI の照合を行うために調整 CIT のプロパティだけでなく，追加の CIT のプロパティが必要な場合に使用されます。たとえば，ip_address CIT に属する name プロパティによって node タイプの調整を実行できます。

調整オブジェクト：このオブジェクトは，アダプタが受け取った調整データに従って作成します。このオブジェクトは，外部 CI を参照する必要があります。マッピング・エンジンは，このオブジェクトを使って外部 CI と CMDB CI を接続します。

調整 CI タイプ：調整オブジェクトを表す CI のタイプ。これらの CI は，CMDB と外部データ・リポジトリの両方に格納されている必要があります。

マッピング・エンジン：相互に仮想関係を持つ，異なるデータ・リポジトリの CI 間の関係を識別するコンポーネント。この識別は，CMDB の調整オブジェクトと外部 CI の調整オブジェクトを調整することによって行われます。

マッピング・エンジン

Federation Framework は，マッピング・エンジンを使ってフェデレート TQL クエリを計算します。マッピング・エンジンは，異なるデータ・リポジトリから取得され，仮想関係によって接続された CI 同士を接続します。マッピング・エンジンは，仮想関係の調整データも提供します。仮想関係の一方のエンドは，CMDB を参照する必要があります。このエンドは，reconciliation タイプになります。2 つのサブグラフの計算では，いずれのエンド・ノードからでも仮想関係を開始できます。

フェデレート・アダプタ

フェデレート・アダプタは，外部 CI データと外部 CI に属する調整オブジェクトという 2 種類のデータを外部データ・リポジトリから取得します。

- **外部 CI データ**：CMDB に存在しない外部データ。外部データ・リポジトリのターゲット・データです。
- **調整オブジェクト・データ**：Federation Framework が CMDB CI と外部データを接続するために使用する補助データ。各調整オブジェクトは，外部 CI を参照する必要があります。調整オブジェクトのタイプは，データが取得される仮想関係のいずれかのエンドのタイプ（サブタイプ）です。調整オブジェクトは，アダプタが受け取る調整データに適合する必要があります。調整オブジェクトには，IdReconciliationObject，PropertyReconciliationObject，または TopologyReconciliationObject の 3 つのタイプがあります。

DataAdapter をベースとするインタフェース（DataAdapter，PopulateDataAdapter，PopulateChangesDataAdapter）では，調整はクエリ定義の一部として要求されます。

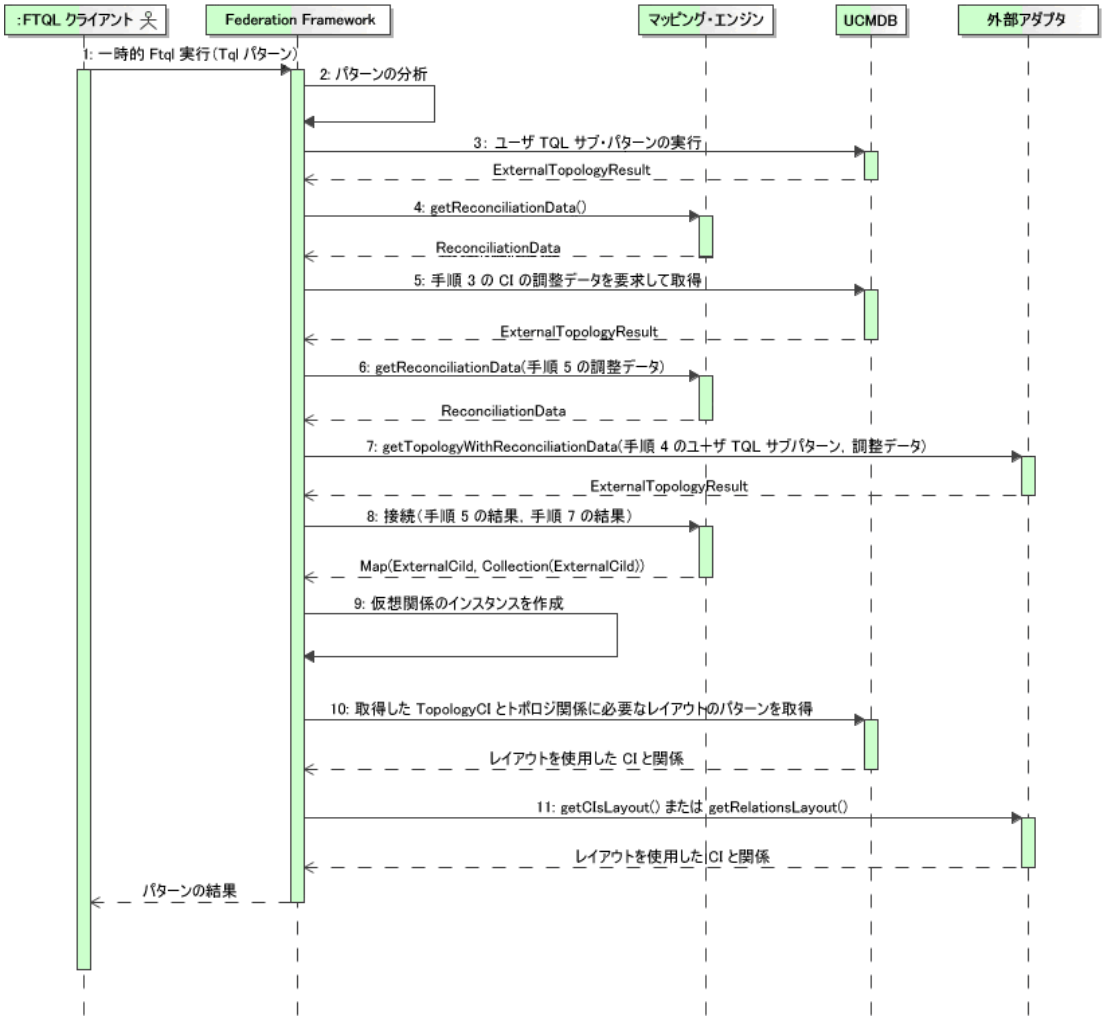
フロー図

次の図は，Federation Framework，UCMDB，アダプタ，およびマッピング・エンジンの間のやり取りを示したものです。図例のフェデレート TQL クエリに含まれる仮想関係は 1 つのみのため，このフェデレート TQL クエリには UCMDB と 1 つの外部データ・リポジトリのみが関与します。

最初の図では UCMDB で計算が開始され，2 番目の図では外部アダプタで計算が開始されます。図内の各手順には，アダプタまたはマッピング・エンジンのインタフェースの適切なメソッド呼び出しへの参照が含まれています。

HP Universal CMDB エンドで開始される計算

次のシーケンス図は，Federation Framework，UCMDB，アダプタ，およびマッピング・エンジンの間のやり取りを示したものです。図例のフェデレート TQL クエリに含まれる仮想関係は 1 つのみのため，このフェデレート TQL クエリには UCMDB と 1 つの外部データ・リポジトリのみが関与します。

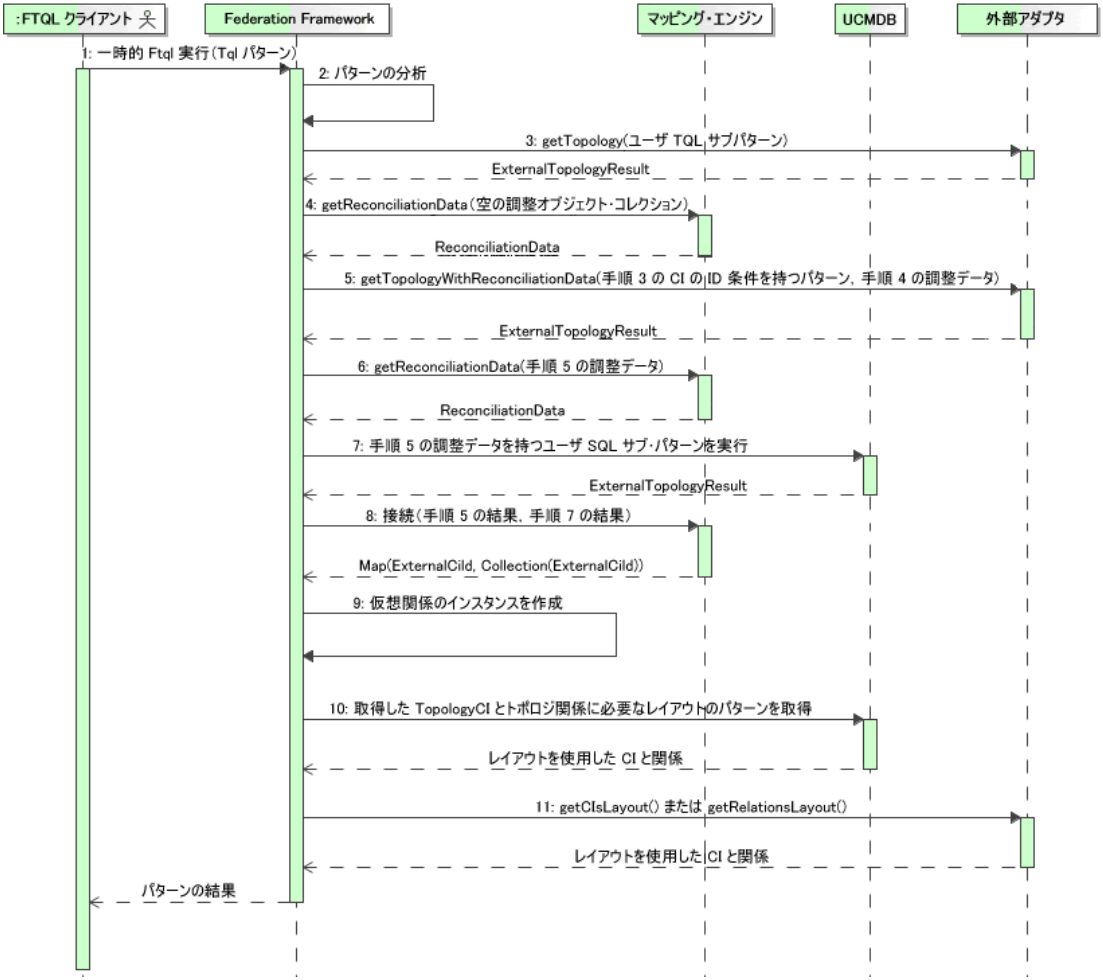


以下では，この図の各番号について説明します。

番号	説明
1	Federation Framework は，フェデレート TQL 計算の呼び出しを受け取ります。
2	Federation Framework はアダプタを分析し，仮想関係を検出し，元の TQL を UCMDB 用と外部データ・リポジトリ用の 2 つのサブアダプタに分割します。
3	Federation Framework は, UCMDB にサブ TQL のトポロジを要求します。
4	Federation Framework は，トポロジ結果を受け取った後，現在の仮想関係に対する適切なマッピング・エンジンを呼び出し，調整データを要求します。この段階では reconciliationObject パラメータは空です。つまり，この呼び出しでは調整データに条件を追加しません。返された調整データには，UCMDB と外部データ・リポジトリの調整 CI を照合するのに必要なデータが定義されています。調整データには，次のタイプがあります。 ➤ **IdReconciliationData:** CI は ID に従って調整されます。 ➤ **PropertyReconciliationData:** CI はいずれかの CI のプロパティに従って調整されます。 ➤ **TopologyReconciliationData:** CI はトポロジに従って調整されます（たとえば，node CI を調整するには，**IP** の IP アドレスも必要です）。
5	Federation Framework は, 手順 3 で受け取った仮想関係エンドの CI の調整データを UCMDB に要求します。
6	Federation Framework は, マッピング・エンジンを呼び出して調整データを取得します。この状況では（手順 3 と比較して），マッピング・エンジンは手順 5 の調整オブジェクトをパラメータとして受け取ります。マッピング・エンジンは，受け取った調整オブジェクトを調整データに対する条件に変換します。
7	Federation Framework は，外部データ・リポジトリにサブ TQL のトポロジを要求します。外部アダプタは，手順 6 の調整データをパラメータとして受け取ります。

番号	説明
8	Federation Framework は，マッピング・エンジンを呼び出して，受け取った結果同士を接続します。firstResult パラメータは，手順 5 で UCMDB から受け取った外部トポロジ結果です。secondResult パラメータは，手順 7 で外部アダプタから受け取った外部トポロジ結果です。マッピング・エンジンは，1 つ目のデータ・リポジトリ（ここでは UCMDB）の外部 CI ID が 2 つ目の（外部）データ・リポジトリの外部 CI ID にマップされたマップを返します。
9	各マッピングに対して，Federation Framework は仮想関係を作成します。
10	（トポロジ段階でのみ）フェデレート TQL クエリの結果を計算した後で，Federation Framework は結果として得られた CI と関係の元の TQL レイアウトを該当するデータ・リポジトリから取得します。

外部アダプタ・エンドで開始される計算



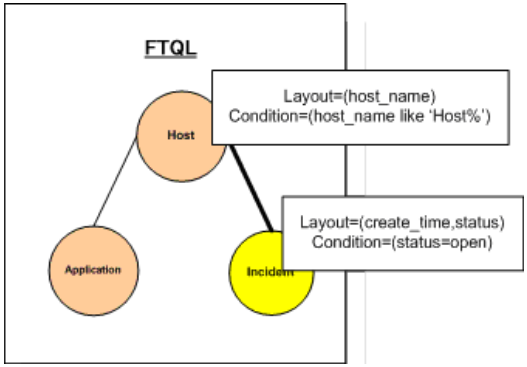
以下では，この図の各番号について説明します。

番号	説明
1	Federation Framework は，フェデレート TQL 計算の呼び出しを受け取ります。
2	Federation Framework はアダプタを分析し，仮想関係を検出し，元の TQL を UCMDB 用と外部データ・リポジトリ用の 2 つのサブアダプタに分割します。
3	Federation Framework は, 外部アダプタにサブ TQL のトポロジを要求します。要求には調整データが含まれないため, 返された ExternalTopologyResult に調整オブジェクトが含まれることはありません。
4	Federation Framework は，トポロジ結果を受け取った後，現在の仮想関係を使って適切なマッピング・エンジンを呼び出し，調整データを要求します。この状況では reconciliationObjects パラメータは空です。つまり，この呼び出しでは調整データに条件を追加しません。返された調整データには，UCMDB と外部データ・リポジトリの調整 CI を照合するのに必要なデータが定義されています。調整データには，次の 3 種類があります。 ➤ **IdReconciliationData:** CI は ID に従って調整されます。 ➤ **PropertyReconciliationData:** CI はいずれかの CI のプロパティに従って調整されます。 ➤ **TopologyReconciliationData:** CI はトポロジに従って調整されます（たとえば，node CI を調整するには，**IP** の IP アドレスも必要です）。
5	Federation Framework は，手順 3 で受け取った CI の調整オブジェクトを外部データ・リポジトリに要求します。Federation Framework は，外部アダプタの **getTopologyWithReconciliationData()** メソッドを呼び出します。要求されるトポロジは，手順 3 で受け取った CI を ID 条件として持ち，手順 4 の調整データを含む 1 ノード・トポロジです。

番号	説明
6	Federation Framework は，マッピング・エンジンを呼び出して調整データを取得します。この状況では（手順 3 と比較して），マッピング・エンジンは手順 5 の調整オブジェクトをパラメータとして受け取ります。マッピング・エンジンは，受け取った調整オブジェクトを調整データに対する条件に変換します。
7	Federation Framework は，UCMDB に手順 6 の調整データを含むサブ TQL のトポロジを要求します。
8	Federation Framework は，マッピング・エンジンを呼び出して，受け取った結果同士を接続します。firstResult パラメータは，手順 5 で外部アダプタから受け取った外部トポロジ結果です。secondResult パラメータは，手順 7 で UCMDB から受け取った外部トポロジ結果です。マッピング・エンジンは，1 つ目のデータ・リポジトリ（ここでは外部データ・リポジトリ）の外部 CI ID が 2 つ目のデータ・リポジトリ（UCMDB）の外部 CI ID にマップされたマップを返します。
9	各マッピングに対して，Federation Framework は仮想関係を作成します。
10	（トポロジ段階でのみ）フェデレート TQL クエリの結果を計算した後で，Federation Framework は結果として得られた CI と関係の元の TQL レイアウトを該当するデータ・リポジトリから取得します。

フェデレート TQL クエリ用の Federation Framework フローの例

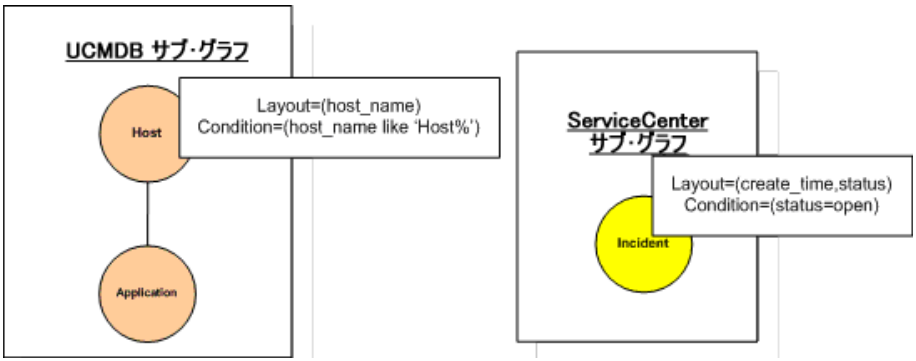
この例では，特定のノード上で開いているすべてのインシデントを表示する方法について説明します。ServiceCenter データ・リポジトリは外部データ・リポジトリです。ノード・インスタンスは UCMDB に格納され，インシデント・インスタンスは ServiceCenter に格納されています。インシデント・インスタンスを適切なノードに接続するには，ホストおよび IP の node および ip_address プロパティが必要であるとします。これらは，UCMDB の ServiceCenter からのノードを識別する調整プロパティです。



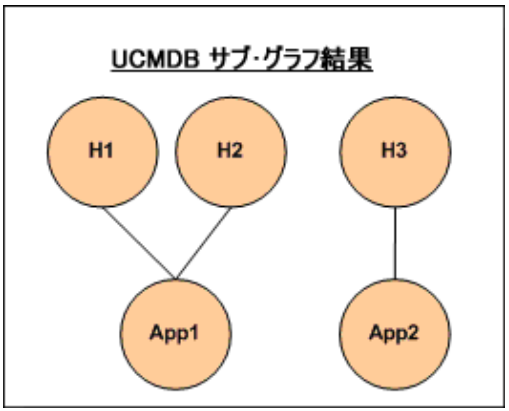
注 : 属性の連携のために, アダプタの **getTopology** メソッドが呼び出されます。調整データは，ユーザ TQL（この場合は CI 要素）で調整されます。

---

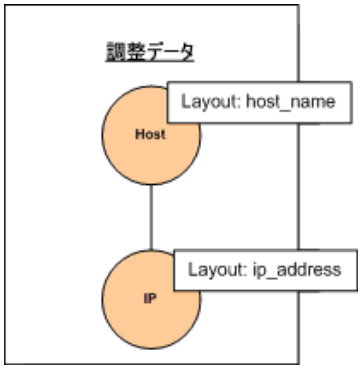
1 Federation Framework は，アダプタを分析した後で，Node と Incident の仮想関係を認識し, フェデレート TQL クエリを次の 2 つのサブグラフに分割します。



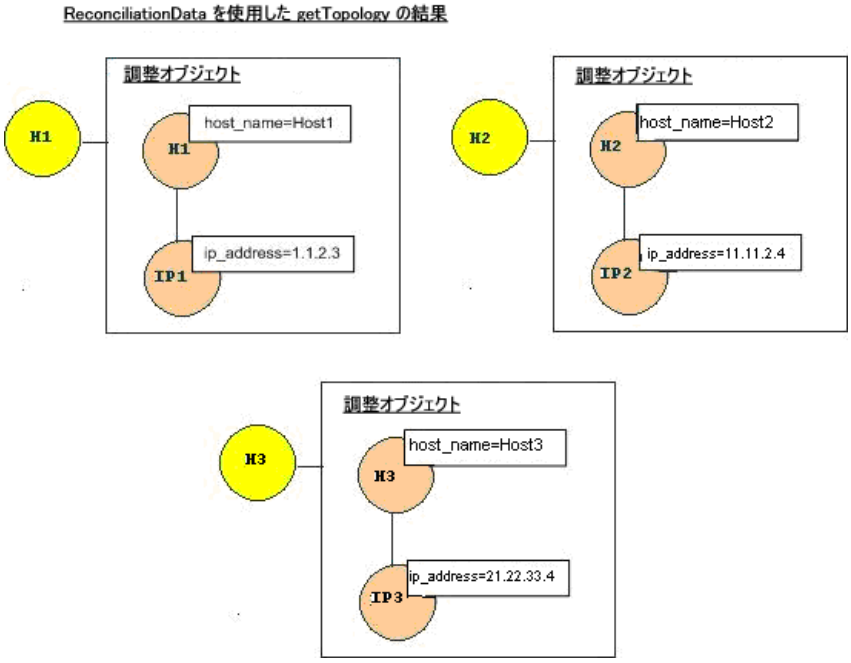
2 Federation Framework は，UCMDB サブグラフを実行してトポロジを要求し，次の結果を受け取ります。



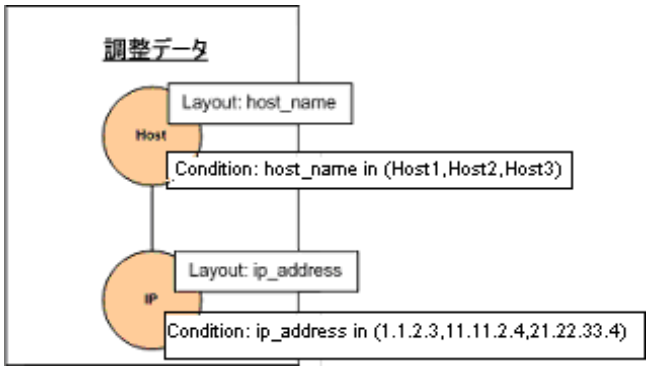
3 Federation Framework は，2 つのデータ・リポジトリから受け取ったデータ同士を接続するための情報を含む 1 つ目のデータ・リポジトリ（UCMDB）の調整データを，適切なマッピング・エンジンに要求します。この場合の調整データは次のとおりです。



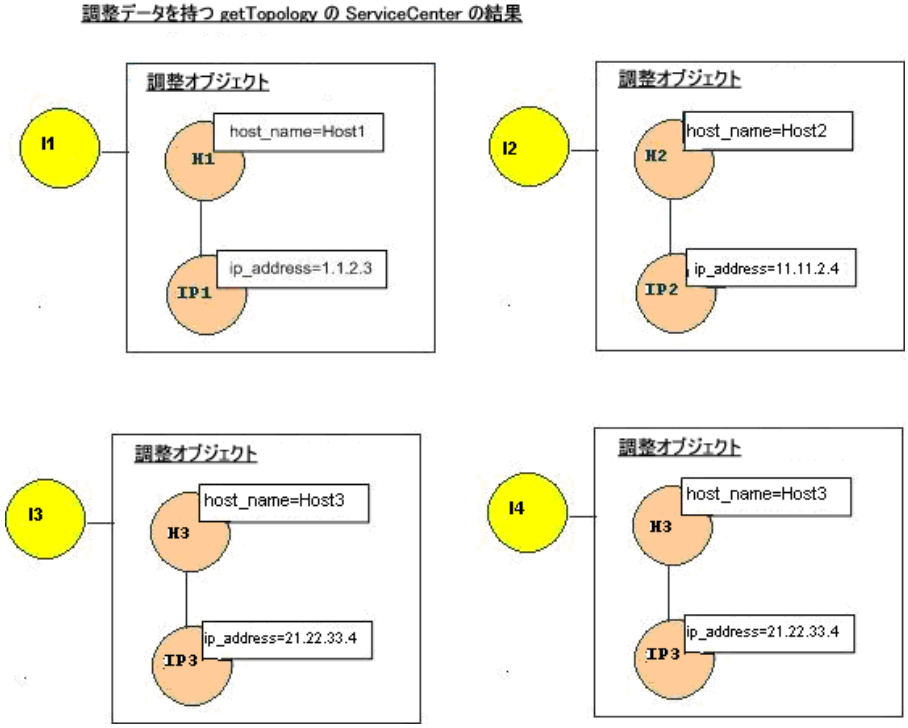
4 Federation Framework は，前の結果（H1，H2，H3 の node）から，ノードとそれに対する ID 条件を含む 1 ノード・トポロジを作成し，そのクエリを UCMDB 上の必要な調整データとともに実行します。この結果には，ID 条件に関連する Node CI と，各 CI の適切な調整オブジェクトが含まれています。



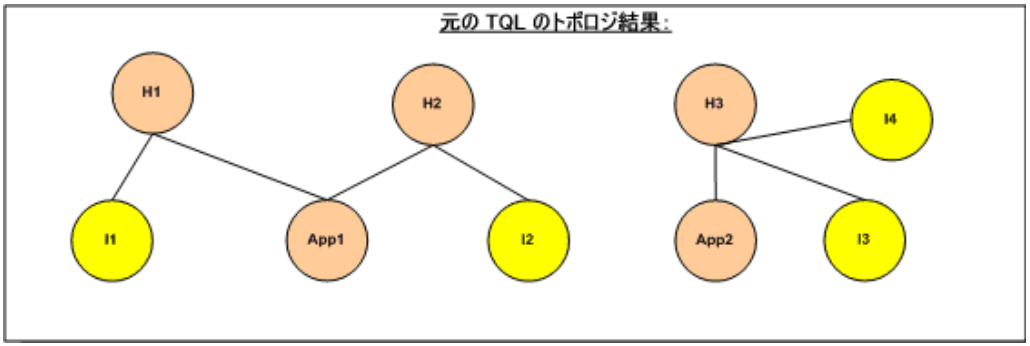
5 ServiceCenter の調整データには，UCMDB から受け取った調整オブジェクトから派生された node と ip の条件が含まれています。



6 Federation Framework は，調整データとともに ServiceCenter サブグラフを実行してトポロジと適切な調整オブジェクトを要求し，次の結果を受け取ります。



7 マッピング・エンジンでの接続と仮想関係の作成後の結果は，次のようになります。



8 Federation Framework は，受け取ったインスタンスの元の TQL レイアウトを UCMDB と ServiceCenter に要求します。

ポピュレーション用の Federation Framework フロー

本項の内容

➤ 229 ページの「定義と用語」

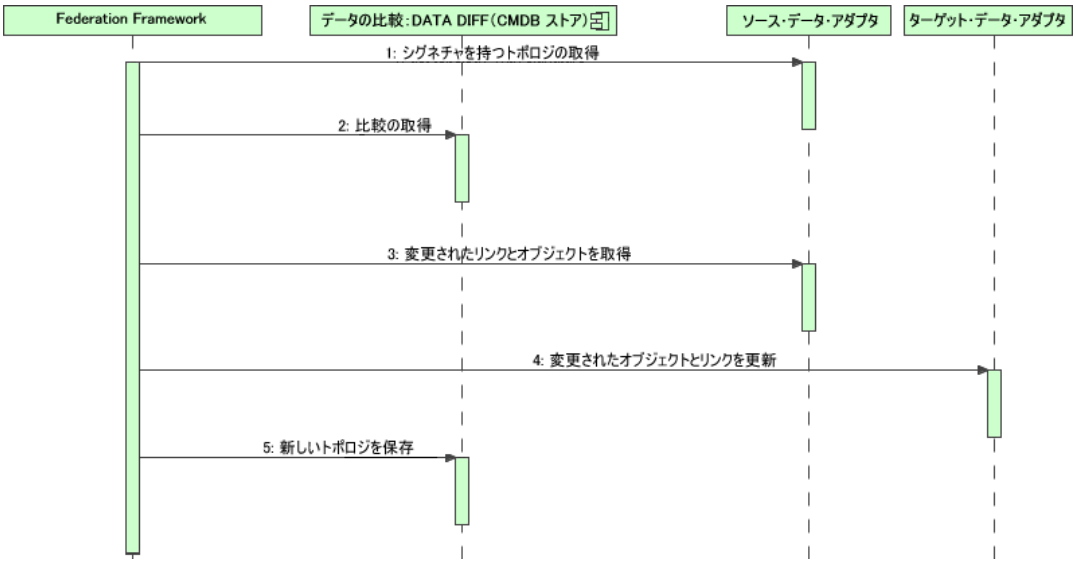
➤ 230 ページの「フロー図」

定義と用語

シグネチャ : CI のプロパティの状態を示します。CI のプロパティ値が変更された場合は，CI シグネチャも変更される必要があります。CI シグネチャによって，すべての CI プロパティを取得して比較しなくても，CI が変更されたかどうかを検出できます。CI と CI シグネチャは，適切なアダプタによって提供されます。アダプタは，CI プロパティが変更されると CI 署名を変更する役割を担います。

フロー図

次のシーケンス図は，ポピュレーション・フローにおける Federation Framework とソースおよびターゲット・アダプタ間のやり取りを示したものです。



1. Federation Framework は，ソース・アダプタからクエリ結果のトポロジを受け取ります。アダプタは，クエリをその名前で認識し，そのクエリを外部データ・リポジトリに対して実行します。トポロジ結果には，結果内の各 CI および関係の ID とシグネチャが含まれています。この ID は，外部データ・リポジトリ内で CI を一意に定義する論理的 ID です。CI または関係が変更された場合は，シグネチャを変更する必要があります。
2. Federation Framework は，シグネチャを使って，新しく受け取ったトポロジ・クエリの結果を保存されている結果と比較し，どの CI が変更されたかを特定します。
3. Federation Framework は，変更された CI と関係を見つけると，変更された CI と関係の ID をパラメータにしてソース・アダプタを呼び出し，それらの完全なレイアウトを取得します。
4. Federation Framework は，ターゲット・アダプタに更新を送信します。ターゲット・アダプタは，受信したデータを使って外部データ・ソースを更新します。
5. 更新後，Federation Framework は最新のクエリ結果を保存します。

アダプタ・インタフェース

本項の内容

定義と用語

外部関係 : 同じアダプタによってサポートされる 2 つの外部 CI タイプ間の関係。

フェデレート TQL クエリ用のアダプタ・インタフェース

次のように，各アダプタの適切なアダプタ・インタフェースを使用します。

アダプタが外部のどの関係もサポートしないときは，**1 ノード・トポロジ・インタフェース**が使用されます。つまり，アダプタが複数の外部 CI についての要求を受け取ることは決してありません。すべての 1 ノード・インタフェースは，ワークフローを簡略化するために作成されます。より詳細なクエリを使用する必要がある場合は，DataAdapter インタフェースを使用します。

UCMDB 9.0 で廃止 : パターン・トポロジ・インタフェース

複合フェデレート・クエリをサポートするアダプタの定義には，**DataAdapter インタフェース**を使用します。これらのアダプタ内の調整要求は，1 つの QueryDefinition パラメータの一部です。これらのアダプタはポピュレーションにも使用できます。

1 ノード・インタフェース

次のインタフェースは，それぞれ異なるタイプの調整データを持っています。

➤ **OneNodeTopologyIdReconciliationDataAdapter:** アダプタが**単一ノード TQL** をサポートし，データ・リポジトリ間の調整が ID によって計算される場合に使用します。

➤ **OneNodeTopologyPropertyReconciliationDataAdapter:** アダプタが**単一ノード TQL** をサポートし，データ・リポジトリ間の調整が 1 つの CI のプロパティによって行われる場合に使用します。

- **OneNodeTopologyDataAdapter:** アダプタが**単一ノード TQL** をサポートし，データ・リポジトリ間の調整がトポロジによって行われる場合に使用します。

データ・アダプタ・インタフェース

- **DataAdapter:** このアダプタは，複合フェデレート TQL クエリをサポートする場合に使用します。このアダプタは，最も幅広い多様性を許容します。
- **PopulateDataAdapter:** このアダプタは，複合フェデレート TQL クエリ・フローおよびポピュレーション・フローをサポートする場合に使用します。ポピュレーション・フローでは，このアダプタはデータ・セット全体を取得し，プローブによって前回のジョブ実行時以降の差分をフィルタします。
- **PopulateChangesDataAdapter:** このアダプタは，複合フェデレート TQL クエリ・フローおよびポピュレーション・フローをサポートする場合に使用します。ポピュレーション・フローでは，このアダプタは，前回のジョブ実行時以降に発生した変更の取得のみをサポートします。

パターン・トポロジ・インタフェース（UCMDB 9.0 で廃止）

次のインタフェースは，それぞれ異なるタイプの調整データを持っています。

- **PatternTopologyIdReconciliationDataAdapter** ：アダプタが**複合 TQL** をサポートし，データ・リポジトリ間の調整が ID によって行われる場合に使用します。
- **PatternTopologyPropertyReconciliationDataAdapter** ：アダプタが**複合 TQL** をサポートし，データ・リポジトリ間の調整が単一ノードのプロパティによって行われる場合に使用します。
- **PatternTopologyDataAdapter** : アダプタが**複合 TQL** をサポートし，データ・リポジトリ間の調整がトポロジによって行われる場合に使用します。

その他のインタフェース

- **SortResultDataAdapter:** 外部データ・リポジトリで結果の CI を並べ替えることができる場合に使用します。
- **FunctionalLayoutDataAdapter:** 外部データ・リポジトリで機能のレイアウトを計算できる場合に使用します。

同期用のアダプタ・インタフェース

- **SourceDataAdapter:** ポピュレーション・フローのソース・アダプタに使用します。
- **TargetDataAdapter:** データ・プッシュ・フローのターゲ ット・アダプタに使用します。

タスク

新規の外部データ・ソースのためのアダプタの追加

このタスクでは，新しい外部データ・ソースをサポートするアダプタを定義する方法について説明します。

このタスクには次の手順が含まれます。

1 前提条件

UCMDB データ・モデルの CI および関係のための，モデルによりサポートされるアダプタ・クラス : アダプタの開発者は，次のことを行う必要があります。

- UCMDB の CI タイプの階層に関する知識を習得し, 外部 CIT が UCMDB の CIT とどのように関連付けられるかを理解する
- 外部 CIT を UCMDB クラス・モデルでモデル化する
- 新しい CI タイプとそれらの関係の定義を追加します。
- アダプタの内部クラス間の有効な関係に対応する，UCMDB クラス・モデル内の有効な関係を定義する（これらの CIT は，UCMDB クラス・モデル・ツリーの任意のレベルに配置できます)。

モデル化は，フェデレーションのタイプ（オンザフライかレプリケーションか）に関係なく同じである必要があります。UCMDB クラス・モデルに新しい CIT 定義を追加する方法の詳細については，『モデリング・ガイド』の「CI の選択を使った作業」を参照してください。

アダプタが CIT のフェデレート属性をサポートできるようにするには，この CIT をサポートされる属性およびこの CIT の調整ルールとともにサポートされるクラスに追加します。

2 仮想関係に対応する有効な関係の定義

注 : このセクションは，連携にのみ関係しています。

ローカル CMDB CIT に接続されているフェデレート CIT を取得するには，CMDB の 2 つの CIT 間に有効なリンク定義が存在する必要があります。

a これらのリンク（まだ存在しない場合）を含む有効なリンク XML ファイルを作成します。

b リンク XML ファイルを ¥**validlinks** フォルダ内のアダプタ・パッケージに追加します。詳細については，『HP UCMDB 管理ガイド』の「パッケージ・マネージャ」を参照してください。

有効な関係定義の例 :

次の例では，node タイプのインスタンスと myclass1 タイプのインスタンス間の containment タイプの関係が有効な関係定義です。

```
<Valid-Links>
    <Valid-Link>
        <Class-Ref class-name="containment"/>
        <End1 class-name="node"/>
        <End2 class-name="myclass1"/>
        <Valid-Link-Qualifiers/>
    </Valid-Link>
</Valid-Links>
```

3 アダプタ設定の定義

a **アダプタ管理**に移動します。

b ［**リソースの新規作成**］ボタンをクリックします。

c ［新規アダプタ］ダイアログ・ボックスで，［**Integration**］および［**Java アダプタ**］を選択します。

d 作成したアダプタ上で右クリックし，ショートカット・メニューから［**アダプタ ソースを編集**］を選択します。

e 次の XML タグを編集します。

```
<pattern xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="newAdapterIdName"
xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="Adapter Description"
schemaVersion="9.0" displayName="New Adapter Display Name">
    <deletable>true</deletable>
    <discoveredClasses>
        <discoveredClass>link</discoveredClass>
        <discoveredClass>object</discoveredClass>
    </discoveredClasses>
    <taskInfo className="com.hp.ucmdb.discovery.probe.services.dynamic.core.AdapterService">
        <params
className="com.hp.ucmdb.discovery.probe.services.dynamic.core.AdapterServiceParams"
enableAging="true" enableDebugg ing="false" enableRecording="false" autoDeleteOnErrors="success"
recordResult="false" maxThreads="1" patternType="java_adapter" maxThreadRuntime="25200000">
            <className >com.yourCompany.adapter.MyAdapter.MyAdapterClass</className>
        </params>
```

```
        <destinationInfo className="com.hp.ucmdb.discovery.probe.tasks.BaseDestinationData">
            <!-- 確認 -->
            <destinationData name="adapterId" description="">${ADAPTER.adapter_id}</
destinationData>
            <destinationData name="attributeValues" description="">${SOURCE.attribute_values}</
destinationData>
            <destinationData name="credentialsId" description="">${SOURCE.credentials_id}</
destinationData>
            <destinationData name="destinationId" description="">${SOURCE.destination_id}</
destinationData>
        </destinationInfo>
        <resultMechanism isEnabled="true">
            <autoDeleteCITs isEnabled="true">
                <CIT>link</CIT>
                <CIT>object</CIT>
            </autoDeleteCITs>
        </resultMechanism>
    </taskInfo>
    <adapterInfo>
        <adapter-capabilities>
            <support-federated-query>
                <!--<supported-classes/> <! サポートされているクラスに関するセクションを参照 -->
            <topology>
                <pattern-topology /> <!or <one-node-topology> -->
            </topology>
            </support-federated-query>
            <!--<support-replicatioin-data>
            <source>
                <changes-source/>
            </source>
    <target/>
        </adapter-capabilities>
        <default-mapping-engine />
            <queries />
    <removedAttributes />
        <full-population-days-interval>-1</full-population-days-interval>
    </adapterInfo>
    <inputClass>destination_config</inputClass>
    <protocols />
```

```
<parameters>
    <!-- 説明属性は平文テキストか HTML で記述できます。-->
    <!-- ホスト属性は UCMDB によって特殊なケースとして扱われ，-->
    <!-- この属性の値に応じて自動的にプローブ名を -->
    <!-- 選択します ( 可能な場合 )。-->
    <parameter name="credentialsId" description="Special type of property, handled by UCMDB for
credentials menu" type="integer" display-name="Credentials ID" mandatory="true" order-index="12" />
    <parameter name="host" description="The host name or IP address of the remote machine"
type="string" display-name="Hostname/IP" mandatory="false" order-index="10" />
    <parameter name="port" description="The remote machine's connection port" type="integer"
display-name="Port" mandatory="false" order-index="11" />
    </parameters>
<parameter name="myatt" description="is my att true?"type="string" display-name="My Att"
mandatory="false" order-index="15" valid-values=True;False/>True</parameters>
    <collectDiscoveredByInfo>true</collectDiscoveredByInfo>
    <integration isEnabled="true">
        <category >My Category</category>
    </integration>
    <overrideDomain>${SOURCE.probe_name}</overrideDomain>
    <inputTQL>
        <resource:XmlResourceWrapper xmlns:resource="http://www.hp.com/ucmdb/1-0-0/
ResourceDefinition" xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:tql="http://
www.hp.com/ucmdb/1-0-0/TopologyQueryLanguage">
            <resource xsi:type="tql:Query" group-id="2" priority="low" is-live="true" owner="Input TQL"
name="Input TQL">
                <tql:node class="adapter_config" id="-11" name="ADAPTER" />
                <tql:node class="destination_config" id="-10" name="SOURCE" />
                <tql:link to="ADAPTER" from="SOURCE" class="fcmdb_conf_aggregation" id="-12"
name="fcmdb_conf_aggregation" />
            </resource>
        </resource:XmlResourceWrapper>
    </inputTQL>
    <permissions />
</pattern>
```

XML タグの詳細については，246 ページの「XML 設定タグとプロパティ」を参照してください。

4 サポートされるクラスの定義

アダプタ・コードに *getSupportedClasses()* メソッド を実装するか，パターン XML ファイルを使用してサポートされるクラスを定義します。

```
<supported-classes>
    <supported-class name="HistoryChange" is-derived="false"
is-reconciliation-supported=false federation-not-supported=false
is-id-reconciliation-supported=false>
    <supported-conditions>
        <attribute-operators attribute-name="change_create_time">
            <operator>GREATER</operator>
            <operator>LESS</operator>
            <operator>GREATER_OR_EQUAL</operator>
            <operator>LESS_OR_EQUAL</operator>
            <operator>CHANGED_DURING</operator>
        </attribute-operators>
    </supported-conditions>
 </supported-class>
```

名前	CI タイプの名前
is-derived	この定義に継承するすべての子を含めるかどうかを指定します
is-reconciliation-supported	このクラスを調整に使用するかどうかを指定します
is-id-reconciliation-supported	このクラスを ID 調整に使用するかどうかを指定します
federation-not-supported	この CIT を連携用に許可しないかどうかを指定します（連携用のみに定義された CIT など，特定の CIT のブロック）
<supported-conditions>	各属性でサポートされる条件を指定します

5 アダプタの実装

アダプタに定義された機能に従って，正しいアダプタ実装クラスを選択します。アダプタ実装クラスは，定義された機能に従って適切なインタフェースを実装します。

6 調整ルールの定義またはマッピング・エンジンの実装

アダプタがフェデレート TQL クエリをサポートする場合，マッピング・エンジンの定義には次の 3 つのオプションがあります。

- 標準指定の CMDB 9.0 マッピング・エンジンを使用します。このエンジンでは，マッピングのために CMDB 内部調整ルールが使用されます。これを使用するには，**<default-mapping-engine/>** XML タグを空のままにします。

  詳細については，177 ページの「reconciliation_types.txt ファイル」を参照してください。

- CMDB 8.0 マッピング・エンジンを使用します。その場合，XML タグ **<default-mapping-engine>com.hp.ucmdb.federation.mappingEngine.AdapterMappingEngine</default-mapping-engine>** を使用します。

  詳細については，177 ページの「reconciliation_rules.txt ファイル（下位互換性用）」を参照してください。

- マッピング・エンジン・インタフェースを実装して，JAR に残りのアダプタ・コードを配置することで，独自のマッピング・エンジンを記述します。これを行うには，XML タグ **<default-mapping-engine>com.yourcompany.map.MyMappingEngine</default-mapping-engine>** を使用します。

7 実装に必要な JAR のクラス・パスへの追加

クラスを実装するには，コード・エディタ・クラス・パスに **federation_api.jar** ファイルを追加します。

8 アダプタのデプロイ

a アダプタ・パッケージをデプロイします。パッケージのデプロイの詳細については，『HP UCMDB 管理ガイド』の「パッケージ・マネージャ」を参照してください。

パッケージには，次のエンティティが含まれている必要があります。

- 新しい CIT の定義（任意選択）：

  アダプタが UCMDB にまだ存在しない新しい CI タイプをサポートする場合にのみ使用されます。

新しい CIT の定義は，パッケージ内の class フォルダにあります。

- 新しいデータ型の定義（任意選択）：

  新しい CIT に新しいデータ型が必要な場合にのみ使用されます。

  新しいデータ型の定義は，パッケージ内の typedef フォルダにあります。

- 新しい有効な関係の定義（任意選択）：

  アダプタがフェデレート TQL をサポートする場合にのみ使用されます。

  新しい有効な関係の定義は，パッケージ内の validlinks フォルダにあります。

- パターン設定の XML ファイルは，パッケージ内の discoveryPatterns フォルダに置く必要があります。
- **記述子 :** パッケージ定義を定義します。
- パッケージの **adapterCode¥<adapter id>** フォルダの下にコンパイルしたクラス（通常 jar ファイル）を置きます。

注 : < adapter id >フォルダの名前は，アダプタ設定の値と同じです。

- 独自の構成ファイルを作成する場合は，ファイルをパッケージ内の **adapterCode¥<adapter id>** フォルダの下に置く必要があります。

9 アダプタの更新

アダプタの非バイナリ・ファイルへの変更は，アダプタ管理モジュールで行うこともできます。アダプタ管理モジュールで構成ファイルに変更を加えると，アダプタは新しい設定を再度読み込みます。

パッケージ内のファイル（バイナリ・ファイル，非バイナリ・ファイルの両方）を編集し，パッケージ・マネージャを使用してパ ッケージを再デプロイすることで，アダプタを更新することもできます。詳細については，『HP UCMDB 管理ガイド』の「パッケージのデプロイ」を参照してください。

マッピング・エンジンの実装

マッピング・エンジンの設定は，どのマッピング・エンジンを使用するかにより異なります。

このタスクには次の手順が含まれます。

➤ 242 ページの「reconciliation_types.txt ファイルを設定します（標準設定の UCMDB 9.0 マッピング・エンジンの場合）。」

➤ 242 ページの「reconciliation_rules.txt ファイルを設定します（UCMDB 8.0 マッピング・エンジンの場合）。」

1 reconciliation_types.txt ファイルを設定します（標準設定の UCMDB 9.0 マッピング・エンジンの場合）。

このファイルは, アダプタの調整に使用する CI タイプの定義に使用されます。

次のように，調整で使用する各 CI タイプを 1 行で記述します。

```
node
business_application
```

ファイルをアダプタ・パッケージの **adapterCode¥<AdapterID>¥META-INF¥** フォルダに置きます。

2 reconciliation_rules.txt ファイルを設定します（UCMDB 8.0 マッピング・エンジンの場合）。

このファイルは調整ルールを設定するのに使います。このファイルの各行がルールを示します。たとえば,

```
reconcilition_type[node] expression[^node.name OR ip_address.name]
end1_type[node] end2_type[ip_address] link_type[containment]
```

reconcilition_type パラメータには，調整が実行される CI のタイプ（TQL のフェデレート・クラスに接続される UCMDB クラス名）を記入します。

expression パラメータは，2 つの調整オブジェクト（UCMDB 側の調整オブジェクトとフェデレート・アダプタ側の調整オブジェクト）が等しいかどうかを判断するロジックです。

この式は OR と AND で構成されています。

式の中で属性名に関する規則の部分は **[className].[attributeName]** です。たとえば，**ip** クラスの属性 **ip_address** は **ip.ip_address** と記述されます。

順序一致を定義できます。順序一致によって，最初の OR 副次式がチェックされます。2 つの調整オブジェクトに副次式の属性に対する値があり，式が false（調整オブジェクトが等しくない）を返した場合，2 つ目の OR 副次式は比較されません。

順序一致の場合，**expression** ではなく，**ordered expression** を使用します。

曲折アクセント記号（^）は，比較時に大文字と小文字を無視するために使用します。

その他のパラメータ（**end1_type**，**end2_type**，**link_type**）は，調整データに 2 つのノードが含まれ，単なる調整タイプのノードではない場合（トポロジ調整データ）にのみ使用されます。この場合，調整データは **end1_type -（link_type）> end2_type** です。

関連レイアウトは式から取得されるので，追加する必要はありません。

UCMDB ID による調整を実行するには，式で属性名として **cmdb_id** を使用します。

ファイルをアダプタ・パッケージの **adapterCode¥<AdapterID>¥META-INF¥** フォルダに置きます。

例：

➤ 調整ルールは，node CIT に対してのみ追加できます。これは，外部 CIT と有効な関係を持つのが node CIT のみであるためです。たとえば，CMDB の node CI は，node.name 属性または ip_address.name 属性に基づいて ServiceCenter 内の node CI と照合されます。

- この場合の調整ルールはトポロジ・ルールであり，式は順序付けられています。このルールによって次のチェックが比較対象の CI に対して行われます。
  - node.name 属性が等しい場合，ルールはノード同士を照合します。
  - node.name 属性が等しくない場合，ルールはノード同士を照合しません。
  - いずれかの比較対象 CI の node.name 属性が null の場合，ルールは ip_address.name 属性をチェックします。ip_address.name 属性が等しい場合，ルールはノード同士を照合します。

サンプル・アダプタの作成

この例では，サンプル・アダプタの作成方法を示しています。

このタスクには次の手順が含まれます。

- 244 ページの「アダプタ・ロジックの選択」
- 245 ページの「プロジェクトの読み込み」

1 アダプタ・ロジックの選択

アダプタを実装する場合，実装内で条件ロジック（プロパティ条件，ID 条件，調整条件，およびリンク条件）を処理する方法を選択する必要があります。

a データ全体をアダプタ・メモリに取得し，それにより必要な CI インスタンスを選択またはフィルタさせます。

b すべての条件をソース言語に変換し，それによりデータをフィルタおよび選択させます。たとえば，

- 条件を SQL クエリに変換します。
- 条件を Java API フィルタ・オブジェクトに変換します。

c 中間的な方法として，データの一部をリモート・サービスでフィルタし，残りをアダプタに選択およびフィルタさせるという方法があります。

MyAdapter の例では，手順 a のロジックが使用されています。

2 プロジェクトの読み込み

a UCMDB バージョン 9.0 以降の SKD が含まれている **DevSampleOneDataAdapter.zip** ファイルの中身を，一時ディレクトリに抽出します。

b 抽出されたファイル内の ¥**DevSampleOneDataAdapter** フォルダに移動します。

c C:¥hp¥UCMDB¥UCMDBServer¥lib の **federation-api.jar** ファイルを，**< 一時フォルダ >¥DevSampleOneDataAdapter** フォルダ内の **production-lib** フォルダにコピーします。

d **test-lib** フォルダにあるすべての JAR ファイルについて，C:¥hp¥UCMDB¥UCMDBServer¥lib フォルダで更新済みのファイルを探し，**< 一時フォルダ >¥DevSampleOneDataAdapter** フォルダ内の **test-lib** フォルダにコピーします。

e アダプタを読み込むには，Eclipse または Intellij（フリー・コミュニティ・バージョンが入手可能）をインストールして開きます。

f サンプル・コードを確認し，テストします。

g Ant スクリプトを使用してパッケージを構築します（myadapter script, goal: all）。Ant スクリプトの詳細については，IDE（Eclipse または Intellij）ドキュメントを参照してください。

注 : このアダプタを例として使用します。大規模なデータ・セットを含むアダプタを使用する場合は，連携のパフォーマンスを改善するために，キャッシングとインデックスを使用する必要がある場合があります。

詳しい情報は，**<temporary folder>¥templates** フォルダで見ることができます。

参照先

XML 設定タグとプロパティ

id="newAdapterIdName"		アダプタの実際の名前を定義します。ログとフォルダ検索に使用されます。
displayName="New Adapter Display Name"		アダプタの表示名を定義します。この名前が UI に表示されます。
<className></className>		Java クラスを実装するアダプタ・インタフェースを定義します。
<category >My Category</category>		アダプタ・カテゴリを定義します。
<parameters>		新しいインテグレーション・ポイントを設定するときに UI で使用可能な設定のプロパティを定義します。
	名前	プロパティの名前（主にコードで使用されます）。
	description	表示されるプロパティの短い説明。
	type	文字列または整数（ブールには文字列を持つ有効値を使用します）。
	display-name	UI に表示されるプロパティの名前。
	mandatory	この設定プロパティを必須とするかどうかを指定します。
	order-index	プロパティの配置順序（small = up）。
	valid-values	文字 ; で区切られた使用可能な有効値のリスト（たとえば, valid-values="Oracle;SQLServer;MySQL または valid-values=True;False）。
<adapterInfo>		アダプタの静的な設定および機能の定義を含みます。
	<support-federated-query>	このアダプタを連携可能として定義します。
	<one-node-topology>	1 つのフェデレート・クエリ・ノードのクエリをフェデレートする機能。
	<pattern-topology>	複合クエリをフェデレートする機能。
	<support-replicatioin-data>	データ・プッシュおよびポピュレーション・フローを実行する機能を定義します。

	<source>	このアダプタはポピュレーション・フローに使用できます。
	<changes-source/>	このアダプタはポピュレーション変更フローに使用できます。
	<target>	このアダプタはデータ・プッシュ・フローに使用できます。
	<default-mapping-engine>	アダプタのマッピング・エンジンの定義を許可します（標準設定では，アダプタは標準設定のマッピング・エンジンを使用します）。ほかのマッピング・エンジンを使用する場合，実装するマッピング・エンジンのクラス名を入力します（UCMDB 8.0 マッピング・エンジンの場合は com.hp.ucmdb.federation.mappingEngine.AdapterMappingEngine を使用します）。
	<removedAttributes>	結果から特定の属性を強制的に削除します。
	<full-population-days-interval>	差分ジョブではなく完全なポピュレーション・ジョブを（x 日ごとに）実行するときに指定します。変更フローとともにエイジング・メカニズムを使用します。

6

プッシュ・アダプタの開発

本章の内容

概念

タスク

参照先

概念

プッシュ・アダプタの開発の概要

汎用プッシュ・アダプタは, UCMDB 9.x データを外部データ・リポジトリ（データベースおよびサードパーティ・アプリケーション）にプッシュするインテグレーションをすばやく開発できるプラットフォームを提供します。汎用プッシュ・アダプタに基づいてカスタム・インテグレーションを開発する場合，次のファイルやスクリプトが必要になります。

- UCMDB CI リンク・タイプと外部データ項目間の XML マッピング・ファイル。
- データ項目を外部データ・リポジトリにプッシュする Jython スクリプト。

タスク

マッピング・ファイルの準備

マッピング・ファイルを準備するには，次の 2 つの異なる方法があります。

➤ 1 つのグローバル・マッピング・ファイルを準備する。

すべてのマッピングは **mappings.xml** という名前の 1 つのファイルに配置されます。

➤ プッシュ・クエリごとに個別のファイルを準備する。

各マッピング・ファイルは **<query name>.xml** という名前になります。

詳細については，257 ページの「ファイルのマッピングのスキーマ」を参照してください。

このタスクには次の手順が含まれます。

➤ 251 ページの「マッピング・ファイルの作成」
➤ 252 ページの「CI のマップ」
➤ 252 ページの「リンクのマップ」

1 マッピング・ファイルの作成

マッピング・ファイルの構造は次のとおりです。

```
<?xml version="1.0" encoding="UTF-8"?>
<integration>
    <info>
        <source name="UCMDB" versions="9.x" vendor="HP" />
        <!-- 例 : -->
        <target name="Oracle" versions="11g" vendor="Oracle" />
    </info>
    <targetcis>
        <!--- CI マッピング --->
    </targetcis>
    <targetrelations>
        <!--- リンク・マッピング --->
    </ targetrelations>
</integration>
```

2 CI のマップ

各 CMDB CIT は，次のサンプルのようにマップされます。

```
<source_ci_type name="node" mode="update_else_insert">
    <apioutputseq>1</apioutputseq>
    <target_ci_type name="host">
        <targetprimarykey><pkey>host_key</pkey></targetprimarykey>
        <target_attribute name="host_os" datatype="STRING">
            <map type="direct" source_attribute="discovered_os_name" />
        </target_attribute>
        <!-- その他のターゲット属性 --->
    </target_ci_type>
</source_ci_type>
```

注 : mode の使用可能な値は，スクリプトの実装によって異なります。

---

3 リンクのマップ

有効な各リンクは，次のサンプルのようにマップされます。

```
<link source_link_type="dependency" target_link_type="dependency"
mode="update_else_insert" source_ci_type_end1="webservice"
source_ci_type_end2="sap_gateway">
    <target_ci_type_end1 name="webservice" />
    <target_ci_type_end2 name="sap_gateway" />
</link>
```

Jython スクリプトの記述

マッピング・スクリプトは通常の Jython スクリプトで，Jython スクリプトのルールに従います。詳細については，63 ページの「Jython アダプタの開発」を参照してください。

スクリプトには，成功した場合に空の **OSHVResult** を返す **DiscoveryMain** 関数を含める必要があります。

失敗をレポートするには，次のようにスクリプトで例外を発生させる必要があります。

```
raise Exception('TopologyUpdateService を使用してリモートの UCMDB に挿入できませんでした。リモート UCMDB のログを参照してください。')
```

`DiscoveryMain` 関数では，外部アプリケーションにプッシュするまたは外部アプリケーションから削除するデータ項目を次のように取得できます。

```
# Framework から add/update/delete の結果オブジェクトを (XML 形式で ) 取得
addResult = Framework.getTriggerCIData('addResult')
updateResult = Framework.getTriggerCIData('updateResult')
deleteResult = Framework.getTriggerCIData('deleteResult')
```

外部アプリケーションのクライアントは次のように取得できます。

```
oracleClient = Framework.createClient()
```

アダプタからフレームワークを介して渡された資格情報 ID，ホスト名およびポート番号が自動的に使用されます。

アダプタに定義した接続パラメータを使用する必要がある場合（詳細については，255 ページの「アダプタ・パッケージの作成」の手順 2 を参照），次のコードを使用します。

```
propValue = str(Framework.getDestinationAttribute('<Connection Property Name'))
```

たとえば，

```
serverName = Framework.getDestinationAttribute('ip_address')
```

このセクションには，次の内容も含まれています。

➤ 254 ページの「マッピングの結果を使った作業」

➤ 254 ページの「スクリプトでのテスト接続の処理」

マッピングの結果を使った作業

汎用プッシュ・アダプタでは，ターゲット・システムで追加，更新，または削除するデータを表す XML 文字列が作成されます。Jython スクリプトでは，この XML が解析され，ターゲットに対する追加，更新，または削除の操作が実行されます。

XML 結果の例

```
<root>
    <data>
        <objects>
            <Object mode="update_else_insert" name="ip" operation="add"
mamId="2ebdc7a93dc7f5bcb33a444763c2a16c">
                <field name="root_lastaccesstime" key="false" datatype="DATE"
length="">1275469266</field>
                <field name="display_label" key="false" datatype="STRING"
length="">16.59.61.67</field>
                <field name="ip_probename" key="false" datatype="STRING"
length="">VMUCMDB05</field>
            </Object>
        </objects>
        <links>
            <link targetRelationshipClass="contained" targetParent="nt" targetChild="ip"
operation="add" mode="update_else_insert"
mamId="8c0a38d53c74c3cc972d6254fb50adba">
                <field name="DiscoveryID1">d5aac653aff428b4a3780111f6389d53</
field>
                <field name="DiscoveryID2">2ebdc7a93dc7f5bcb33a444763c2a16c</
field>
            </link>
        </links>
    </data>
</root>
```

スクリプトでのテスト接続の処理

Jython スクリプトを呼び出して，外部アプリケーションとの接続をテストできます。この場合，testConnection 宛先属性が true になります。この属性は，次のようにフレームワークから取得できます。

```
testConnection = Framework.getTriggerCIData('testConnection')
```

テスト接続モードで実行する場合，外部アプリケーションとの接続を確立できないときにスクリプトで例外を発生させる必要があります。接続に成功した場合は，**DiscoveryMain** 関数から空の **OSHVResult** が返されます。

アダプタ・パッケージの作成

1. **C:¥hp¥UCMDB¥UCMDBServer¥content¥adapters¥push-adapter.zip** のコンテンツを一時フォルダに抽出します。
2. **discoveryPatterns¥push_adapter.xml** ファイルを編集します。
   - **a** 新しい ID と表示ラベルで <pattern> タグを変更します。次のタグを

```
<pattern id="PushAdapter" xsi:noNamespaceSchemaLocation="../../Patterns.xsd"
description="Discovery Pattern Description" schemaVersion="9.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

次の要素で置き換えます。

```
<pattern id="MyPushAdapter" displayLabel="My Push Adapter"
xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="Discovery Pattern
Description" schemaVersion="9.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance">
```

   - **b** 必要な接続属性がパラメータのリストに反映されるようにパラメータ・リストを更新します。**probeName** 属性は削除しないでください。
3. **adapterCode¥PushAdapter** フォルダの名前を，手順 2 で使用したアダプタ ID に変更します（**adapterCode¥MyPushAdapter** など）。
4. **discoveryScripts¥pushScript.py** を，記述したスクリプトで置き換えます（詳細については，252 ページの「Jython スクリプトの記述」を参照）。スクリプトの名前を変更する場合，**adapterCode¥<adapter ID>¥push.properties** の jythonScript.name プロパティを適宜更新する必要があります。
5. **adapterCode¥<adapter ID>¥mappings¥mappings.xml** ファイルを，準備したファイルで置き換えます（詳細については，251 ページの「マッピング・ファイルの準備」を参照）。

各 TQL メソッドのマッピング・ファイルを使用する場合，対応する TQL の名前を各 XML ファイルに割り当てます。ファイル名の後には **.xml** が続きます。この場合，現在の TQL 名に特定のマッピング・ファイルが見つからなければ **mappings.xml** ファイルが標準設定として使用されます。標準設定のマッピング・ファイルの名前を変更するには，**adapterCode¥<adapter ID>¥push.properties** の mappingFile.default プロパティを変更します。

参照先

ファイルのマッピングのスキーマ

要素		属性
名前およびパス	**説明**	
統合	ファイルのマッピング・コンテンツを定義する。最初の行とコメントを除き，ファイルの最も外側のブロックである必要があります。	
info （integration）	統合するデータ・リポジトリに関する情報を定義する	
source （integration > info）	ソース・データ・リポジトリに関する情報を定義する	**名前** : type **説明 :** ソース・データ・リポジトリの名前。 **必須 :** 必須 **タイプ :** 文字列
		[名前] : versions **説明 :** ソース・データ・リポジトリのバージョン。 **必須 :** 必須 **タイプ :** 文字列
		[名前] : vendor **説明 :** ソース・データ・リポジトリのベンダ。 **必須 :** 必須 **タイプ :** 文字列

要素		属性
名前およびパス	**説明**	
target （integration > info）	ターゲット・データ・リポジトリに関する情報を定義する	**[名前]** : type **説明** : ソース・データ・リポジトリの名前。 **必須** : 必須 **タイプ** : 文字列
		[名前] : versions **説明** : ソース・データ・リポジトリのバージョン。 **必須** : 必須 **タイプ** : 文字列
		[名前] : vendor **説明** : ソース・データ・リポジトリのベンダ。 **必須** : 必須 **タイプ** : 文字列
targetcis （integration）	すべての CIT マッピングのコンテナ要素	

要素		属性
名前およびパス	**説明**	
source_ci_type （integration > targetcis）	ソース CIT を定義する	**[名前]** : 名前 **説明** : ソース CIT の名前。 **必須** : 必須 **タイプ** : 文字列
		[名前] : mode **説明** : 現在の CI タイプに必要な更新のタイプ。 **必須** : 必須 **タイプ** : 次のいずれかの文字列になります。 ➤ **insert** – CI がまだ存在していない場合にのみ使用します。 ➤ **update** – CI が存在していることがわかっている場合のみ使用します。 ➤ **update_else_insert** – CI が存在している場合は更新し，存在していない場合は新しい CI を作成します。 ➤ **ignore** – 該当の CI タイプの場合は何もしません。

要素		属性
名前およびパス	**説明**	
target_ci_type （integration > targetcis> source_ci_type)	ターゲット CIT を定義する	**[名前]** : 名前 **説明** : ターゲット CI タイプの名前。 **必須** : 必須 **タイプ** : 文字列
		[名前] : スキーマ **説明** : ターゲットで該当の CI タイプを保存するために使用されるスキーマの名前。 **必須** : 任意 **タイプ** : 文字列
		[名前] : namespace **説明** : ターゲットの該当の CI タイプの名前空間を示す **必須** : 任意 **タイプ** : 文字列
targetprimarykey （integration > targetcis > source_ci_type - または - integration > targetrelations > link）	ターゲット CIT のプライマリ・キー属性を識別する	
pkey （integration > targetcis > source_ci_type > targetprimarykey - または - integration > targetrelations > link > targetprimarykey）	1 つのプライマリ・キー属性を識別する モードが **update** または **insert_else_update** の場合にのみ必要になる	

要素		属性
名前およびパス	**説明**	
target_attribute （integration > targetcis > source_ci_type - または - integration > targetrelations > link）	ターゲット CIT 属性を定義する	**[名前]** : 名前 **説明** : ターゲット CIT 属性の名前。 **必須** : 必須 **タイプ** : 文字列
		[名前] : datatype **説明** : ターゲット CIT 属性のデータ型。 **必須** : 必須 **タイプ** : 文字列
		[名前] : length **説明** : ターゲット属性の整数サイズ（文字列 / 文字のデータ型の場合）。 **必須** : 任意 **タイプ** : Integer
		[名前] : option **説明** : 値に適用する変換関数。 **必須** : False **タイプ** : 次のいずれかの文字列になります。 ➤ **uppercase** – 大文字に変換します ➤ **lowercase** – 小文字に変換します ➤ 該当の属性が空の場合，変換関数は適用されません。

要素		属性
名前およびパス	**説明**	
map （integration > targetcis > source_ci_type > target_attribute - または - integration > targetrelations > link > target_attribute）	ソース CIT 属性値を取得する方法を指定する	**[名前]** : type **説明** : ソース値とターゲット値間のマッピングのタイプ。 **必須** : 必須 **タイプ** : 次のいずれかの文字列になります。 ➤ **direct** – ソース属性値からターゲット属性値への 1 対 1 のマッピングを指定します ➤ **compoundstring** – サブ要素が 1 つの文字列に結合され，ターゲット属性値が設定されます ➤ **childattr** – サブ要素は 1 つ以上の子 CIT 属性になります。子 CIT は **container_f** 関係または **contained** 関係のある CIT として定義されます ➤ **constant** – 静的な文字列
		[名前] : value **説明** : type=**constant** の定数文字列 **必須** : type=**constant** の場合にのみ必要 **タイプ** : 文字列
		[名前] : attr **説明** : type=**direct** のソース属性名 **必須** : type=**direct** の場合にのみ必要 **タイプ** : 文字列

要素		属性
名前およびパス	**説明**	
aggregation （integration > targetcis > source_ci_type > target_attribute > map - または - integration > targetrelations > link > target_attribute > map マップ・タイプが **childattr** の場合にのみ有効）	ソース CI の子 CI 属性値を，ターゲット CI 属性をマップする 1 つの値に結合する方法を指定する。任意指定。	**［名前］:** type **説明 :** aggregation 関数のタイプ **必須 :** 必須 **タイプ :** 次のいずれかの文字列になります。 ➤ **csv** – 含まれるすべての値をカンマ区切りリストに連結します（数値または文字列 / 文字）。 ➤ **count** – 含まれるすべての値の個数を返します。 ➤ **sum** – 含まれるすべての値の総数を返します。 ➤ **average** – 含まれるすべての値の平均数を返します。 ➤ **min** – 含まれる値の最小数 / 文字を返します。 ➤ **max** – 含まれる値の最大数 / 文字を返します。

要素		属性
名前およびパス	**説明**	
validation （integration > targetcis > source_ci_type > target_attribute > map - または - integration > targetrelations > link > target_attribute > map マップ・タイプが **childatt** の場合にのみ¥有効）	属性値に基づいてソース CI の子 CI をフィルタリングして除外できる。aggregation サブ要素とともに使用して，子属性がターゲット CIT の属性値に正確にマップされるようにします。任意指定。	**[名前]** : minlength **説明 :** 指定した値よりも短い文字列を除外します。 **必須 :** 任意 **タイプ :** Integer
		[名前] : maxlength **説明 :** 指定した値よりも長い文字列を除外します。 **必須 :** 任意 **タイプ :** Integer
		[名前] : minvalue **説明 :** 指定した値よりも小さい数値を除外します。 **必須 :** 任意 **タイプ :** 数値
		[名前] : minvalue **説明 :** 指定した値よりも大きい数値を除外します。 **必須 :** 任意 **タイプ :** 数値
targetrelations （integration）	すべての関係マッピングのコンテナ要素。任意指定。	

要素		属性
名前およびパス	**説明**	
link （integration > targetrelations）	ソース関係をターゲット関係にマップする。**targetrelation** が存在している場合のみ必須です。	**[名前]** : source_link_type **説明** : ソース関係の名前。 **必須** : 必須 **タイプ** : 文字列
		[名前] : target_link_type **説明** : ターゲット関係の名前。 **必須** : 必須 **タイプ** : 文字列
		[名前] : nameSpace **説明** : ターゲットで作成されるリンクの名前空間。 **必須** : 任意 **タイプ** : 文字列
		[名前] : mode **説明** : 現在のリンクに必要な更新のタイプ。 **必須** : 必須 **タイプ** : 次のいずれかの文字列になります。 ➤ **insert** – CI がまだ存在していない場合にのみ使用します。 ➤ **update** – CI が存在していることがわかっている場合のみ使用します。 ➤ **update_else_insert** - CI が存在している場合は更新し，存在していない場合は新しい CI を作成します。 ➤ **ignore** – 該当の CI タイプの場合は何もしません。

要素		属性
名前およびパス	**説明**	
link （つづき）		**[名前]** : source_ci_type_end1 **説明** : ソース関係の End1 CI タイプ **必須** : 必須 **タイプ** : 文字列
		[名前] : source_ci_type_end2 **説明** : ソース関係の End2 CI タイプ **必須** : 必須 **タイプ** : 文字列
target_ci_type_end1 （integration > targetrelations > link）	ターゲット関係の End1 CI タイプ	**[名前]** : 名前 **説明** : ターゲット関係の End1 CI タイプの名前。 **必須** : 必須 **タイプ** : 文字列
		[名前] : superclass **説明** : End1 CI タイプのスーパークラスの名前。 **必須** : 任意 **タイプ** : 文字列

要素		属性
名前およびパス	**説明**	
target_ci_type_end2 （integration > targetrelations > link）	ターゲット関係の End2 CI タイプ	**[名前]** : 名前 **説明** : ターゲット関係の End2 CI タイプの名前。 **必須** : 必須 **タイプ** : 文字列
		[名前] : superclass **説明** : End2 CI タイプのスーパークラスの名前。 **必須** : 任意 **タイプ** : 文字列

結果のマッピングのスキーマ

要素		属性
名前およびパス	**説明**	
root	結果ドキュメントのルート	
data （root）	データ自体のルート	
objects （root > data）	更新するオブジェクトのルート要素	

要素		属性
名前およびパス	**説明**	
オブジェクト （root > data > objects）	1 つのオブジェクトとそのすべての属性の更新操作を示す	**[名前]** : 名前 **説明** : CI タイプの名前 **必須** : 必須 **タイプ** : 文字列
		[名前] : mode **説明** : 現在の CI タイプに必要な更新のタイプ。 **必須** : 必須 **タイプ** : 次のいずれかの文字列になります。 ➤ **insert** – CI がまだ存在していない場合にのみ使用します。 ➤ **update** – CI が存在していることがわかっている場合のみ使用します。 ➤ **update_else_insert** – CI が存在している場合は更新し，存在していない場合は新しい CI を作成します。 ➤ **ignore** – 該当の CI タイプの場合は何もしません。

要素		属性
名前およびパス	**説明**	
オブジェクト （つづき）		**[名前]** : operation **説明** : 該当の CI で実行する操作。 **必須** : 必須 **タイプ** : 次のいずれかの文字列になります。 ➤ **add** – CI が追加されます ➤ **update** – CI が更新されます ➤ **delete** – CI が削除されます 値が設定されていない場合，標準設定値である **add** が使用されます。
		[名前] : mamId **説明** : ソース CMDB のオブジェクトの ID。 **必須** : 必須 **タイプ** : 文字列

要素		属性
名前およびパス	**説明**	
field （root > data > objects > Object - または - root > data > links > link）	オブジェクトの 1 つのフィールドの値を示す。フィールドのテキストはフィールドの新しい値です。フィールドにリンクがある場合，値はいずれかの終了 ID になります。各終了 ID は，オブジェクトとして（<objects> の下に）表示されます。	**[名前]** : 名前 **説明** : フィールドの名前。 **必須** : 必須 **タイプ** : 文字列
		[名前] : key **説明** : 該当のフィールドがオブジェクトのキーかどうかを指定します。 **必須** : 必須 **タイプ** : ブール
		[名前] : datatype **説明** : フィールドのタイプ。 **必須** : 必須 **タイプ** : 文字列
		[名前] : length **説明** : これはターゲット属性の整数サイズです（文字列 / 文字のデータ型の場合）。 **必須** : 任意 **タイプ** : Integer

要素		属性
名前およびパス	**説明**	
links （root > data）	更新するリンクのルート要素	**[名前]** : targetRelationshipClass **説明** : ターゲット・システムの関係（リンク）の名前。 **必須** : 必須 **タイプ** : 文字列
		[名前] : targetParent **説明** : リンクの 1 つ目のタイプ（親）。 **必須** : 必須 **タイプ** : 文字列
		[名前] : targetChild **説明** : リンクの 2 つ目のタイプ（子）。 **必須** : 必須 **タイプ** : 文字列

要素		属性
名前およびパス	説明	
links （つづき）		**[名前]** : mode **説明 :** 現在の CI タイプに必要な更新のタイプ。 **必須 :** 必須 **タイプ :** 次のいずれかの文字列になります。 ➤ **insert** – CI がまだ存在していない場合にのみ使用します。 ➤ **update** – CI が存在していることがわかっている場合のみ使用します。 ➤ **update_else_insert** – CI が存在している場合は更新し，存在していない場合は新しい CI を作成します。 ➤ **ignore** – 該当の CI タイプの場合は何もしません。
		[名前] : operation **説明 :** 該当の CI で実行する操作。 **必須 :** 必須 **タイプ :** 次のいずれかの文字列になります。 ➤ **add** – CI が追加されます ➤ **update** – CI が更新されます ➤ **delete** – CI が削除されます 値が設定されていない場合，標準設定値である **add** が使用されます。
		[名前] : mamId **説明 :** ソース CMDB のオブジェクトの ID。 **必須 :** 必須 **タイプ :** 文字列

第 II 部

API の使用

7

API の概要

本章の内容

概念

概念

API の概要

HP Universal CMDB には次の API が含まれています。

- **UCMDB Web サービス API:** UCMDB（Universal Configuration Management database）に構成アイテム定義やトポロジ関係を記述できます。また，TQL やアドホック・クエリを使用して情報を照会することもできます。詳細については，277 ページの「HP Universal CMDB Web サービス API」を参照してください。
- **UCMDB Java API:** サードパーティ製ツールまたはカスタム・ツールで Java API を使用して，データや計算を抽出したり UCMDB（Universal Configuration Management database）にデータを書き込む方法について説明します。詳細については，363 ページの「HP Universal CMDB API」を参照してください。

8

HP Universal CMDB Web サービス API

本章の内容

概念

規則

本章では，次の表記規則を使用します。

- **UCMDB** は，Universal Configuration Management database 自体を指します。**HP Universal CMDB** は，アプリケーションを意味します。
- UCMDB 要素とメソッド引数は，スキーマで指定したのと同じように大文字と小文字を区別して入力します。メソッドの要素または引数は大文字にしません。たとえば，relation は，メソッドに渡される Relation タイプの要素です。

HP Universal CMDB Web サービス API の概要

本章は，オンラインのドキュメント・ライブラリで入手できる UCMDB スキーマに関するドキュメントと併せてご利用ください。

HP Universal CMDB Web サービス API は，アプリケーションを HP Universal CMDB（UCMDB）に統合するために使用します。この API により，次を実施するメソッドが提供されます。

- CMDB での CI と関係の追加，削除，および更新
- クラス・モデルに関する情報の取得
- 影響分析の取得
- 構成アイテムおよび関係に関する情報の取得
- 資格情報の管理 : 表示，追加，更新，削除
- ジョブの管理 : ステータスの表示，アクティブ化，非アクティブ化
- プローブ範囲の管理 : 表示，追加，更新
- トリガの管理 : トリガ CI の追加または削除，およびトリガ TQL の追加，削除，または無効化
- ドメインおよびプローブに関する一般データの表示

構成アイテムと関係に関する情報を取得するメソッドでは，一般的にトポロジ・クエリ言語（TQL）を使用します。詳細については，『モデリング・ガイド』の「トポロジ クエリ言語」を参照してください。

HP Universal CMDB Web サービス API のユーザは，次に関する知識が必要です。

- SOAP の仕様
- オブジェクト指向プログラミング言語（C++ や C#，Java など）
- HP Universal CMDB
- Data Flow Management

本項の内容

- 279 ページの「API の使用」
- 280 ページの「権限」

API の使用

API を使用すると，多くのビジネス要件を満たすことができます。たとえば，

- サードパーティ製のシステムは，利用できる構成アイテム（CI）に関する情報をクラス・モデルに問い合わせることができます。
- サードパーティ製のアセット管理ツールは，そのツールのみで利用できる情報を使って CMDB を更新できるため，アセット管理ツールのデータを HP アプリケーションで収集したデータと統一できます。
- 多くのサードパーティ製のシステムは，CMDB にデータをポピュレートして，変更内容を追跡し影響分析を実行できる中心的な CMDB を作成できます。
- サードパーティ製のシステムは，ビジネス・ロジックに従ってエンティティと関係を作成し，データを CMDB に書き込んで CMDB のクエリ機能を活用できます。
- Change Control Management（CCM）システムなど，ほかのシステムは影響分析手法を使用して変更の分析を行えます。

権限

管理者は，Web サービスに接続するためのログイン証明書を提供します。必要な資格情報は，ユーザが HP Universal CMDB をスタンドアロンのアプリケーションとして使用するか，Business Service Management 内から使用するかによって異なります。

- **HP Universal CMDB をスタンドアロンで使用する場合：**ディスカバリ・リソースおよびインテグレーション・リソースに対する権限を付与されている UCMDB ユーザの資格情報を使用してログインします。

  詳細については，『HP UCMDB 管理ガイド』の「[セキュリティ マネージャ] ページ」を参照してください。

- **Business Service Management に組み込こまれた HP Universal CMDB を使用する場合：**Business Service Management ユーザの資格情報を使用してログインします。ユーザは，Business Service Management の HP Universal CMDB リソースに対して関連する権限を付与されている必要があります。

HP Universal CMDB を通して権限が割り当てられる場合の権限レベルは，表示，更新，および実行です。Business Service Management を通して権限が割り当てられる場合の権限レベルは，表示と更新ですが，この更新には実行も含まれます。各操作に必要な権限を表示するには，各操作の要求についてのドキュメントおよび『Data Flow Management Schema Reference』（英語版）を参照してください。

HP Universal CMDB Web サービス API の参考情報

要求と応答の構造に関する完全なドキュメントについては，HP UCMDB Web Service API の参考情報を参照してください。ファイルは次のフォルダにあります。

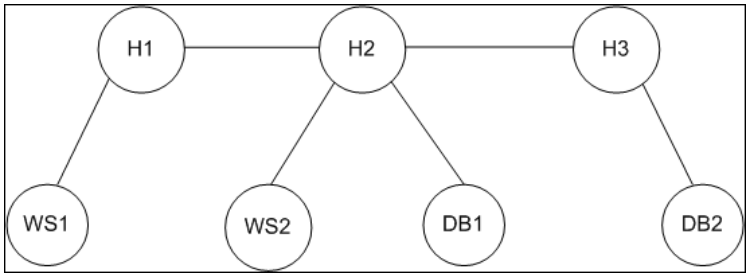
C:¥hp¥UCMDB¥UCMDBServer¥deploy¥ucmdb-docs¥docs¥eng¥doc_lib¥DevRef_guide¥CMDB_Schema¥webframe.html

明確なトポロジ・マップ要素を返す

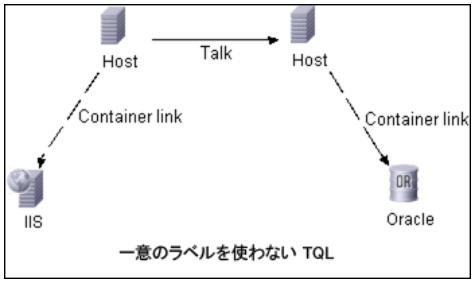
topology または topologyMap 要素のデータを返すクエリ・メソッドは，システムを検索して TQL クエリとの一致を探します。次の図は，結果として得られる topology と topologyMap の構造が，クエリで一意のラベルを使用すると，どのような影響を受けるかを示しています。

ラベルは，特定の設定における関係および構成アイテムに対して，クエリでユーザが指定した名前です。クエリで指定したラベルは，返されるマップでノード・ラベルとして使用されます。ラベルが指定されていない場合は，CI または Relation タイプ名が結果として得られるマップでラベルとして使用されます。次の例では，IISHost ラベルと DBHost ラベルを標準設定の Host ラベルの代わりに指定し，ContainerIIS ラベルと ContainsDB ラベルを標準設定の Container Link ラベルの代わりに指定しています。

次の例は，小規模な IT ユニバース・モデルを示します。ここでは，H1，H2，H3 という 3 つのホストがあり，Web サーバ（WS）とデータベース・マネージャ（DB）をホストしています。WS1 は H1 上に存在します。WS1 は H1 上に存在します。DB2 は H3 上に存在します。



このクエリは，標準設定のラベルを使用して定義します。



この TQL クエリを IT ユニバースで実行した結果，Topology または TopologyMap 要素が得られます。

トポロジの応答

```
CIs: H1, H2, H3, WS1, WS2, DB1, DB2
Relations: H1-WS1, H1-H2, H2-H3, WS2-H2, DB1-H2, DB2-H3
```

TopologyMap の応答

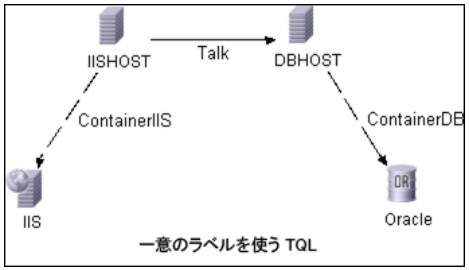
```
CINode:
    label: Host
    CIs: H1, H2
CINode:
    label: Host
    CIs: H2, H3
CINode:
    label: DB
    CIs: DB1, DB2
CINode:
    label: Webserver
CIs: IIS
relationNode:
    label: talk
    relations: H1-H2, H2-H3

relationNode:
    label: Container Link
    relations: WS1-H1, WS2-H2
relationNode:
    label: Container Link
    relations: DB2-H3, DB1-H2
```

前述の TopologyMap の応答では，最初の 2 つの CINode に同一の Host ラベルが含まれ，クエリ内にある 2 つの HostCI に対応しています。これらの CINodes の両方に host H2 が含まれ，H2 が重複する理由は示されていません。

最後の 2 つの relationNode には，同一の Contained ラベルが含まれ，クエリ内の 2 つの Container link 関係に対応しています。

重複が発生するのは，一意のラベルがクエリで指定されなかったためで，その結果，マップ内で標準設定のラベル（タイプ名は，Host と Container）を使用したことが原因です。もっと使いやすいマップを抽出するには，次のクエリに示すように，各設定が一致するように，一意のラベルを使ってクエリを定義します。



topology の結果は，一意のラベルを使わない TQL のものと同一です。ただし，topologyMap の結果は異なります。各ラベルが一意になります。

```
CINode:
    label: IISHOST
    CIs: H1, H2
CINode:
    label: DBHOST
    CIs: H2, H3

...

relationNode:
    label: ContainerIIS
    relations: WS1-H1, WS2-H2
relationNode:
    label: ContainerDB
    relations: DB2-H3, DB1-H2
```

このマップでは，H2 が 2 度返された理由は明らかです。一意のラベルは，H2 が Web サーバ・ホストとして 1 度，データベース・ホストとして 1 度返されたことを示しています。

ヒント：CMDB で可能な場合には，一意のユーザ定義ラベルを特定の設定に適用します。

タスク

Web サービスの呼び出し

HP Universal CMDB Web サービスで標準の SOAP プログラミング技術を使用すると，サーバ側のメソッドを呼び出すことができます。ステートメントを解析できない場合，またはメソッドの呼び出しに問題がある場合は，API メソッドにより，SoapFault 例外がスローされます。SoapFault 例外がスローされると，UCMDB によってエラー・メッセージ，エラー・コード，および例外メッセージ・フィールドの 1 つ以上にデータがポピュレートされます。エラーがなければ，呼び出しの結果が返されます。

SOAP プログラマは，以下のアドレスで WSDL にアクセスできます。

http://<server>[:port]/axis2/services/UcmdbService?wsdl

ポートの指定は，標準とは異なる設定でインストールされている場合のみ必要です。正しいポート番号についてはシステム管理者に問い合わせてください。

サービスを呼び出すための URL は，次のとおりです。

http://<server>[:port]/axis2/services/UcmdbService

たとえば，CMDB への接続例については，317 ページの「使用例」を参照してください。

UCMDB への問い合わせ

CMDB に問い合わせるには，294 ページの「UCMDB クエリ・メソッド」で説明した API を使用します。

クエリと返される CMDB 要素には，常に実際の UMDB ID が含まれています。

クエリ・メソッドの使用例については，323 ページの「クエリの例」を参照してください。

本項の内容

実行時の応答計算

すべてのクエリ・メソッドについて，要求を受信したときに，クエリ・メソッドによって要求された値が UMDB サーバによって計算され，最新のデータに基づいて結果が返されます。結果は，TQL クエリがアクティブで，以前計算した結果が存在する場合でも，要求を受信したときに必ず計算されます。このため，クライアント・アプリケーションに返されるクエリの実行結果は，ユーザ・インタフェースに表示される同じクエリの結果とは異なる場合があります。

ヒント : アプリケーションで特定のクエリの結果を複数回使用し，結果データの使用のたびにデータが大きく違わないことが期待される場合は，同じクエリを繰り返し実行するのではなく，クライアント・アプリケーションにデータを保存することによってパフォーマンスを向上できます。

サイズの大きい応答の処理

クエリに対する応答には，実際のデータは転送されない場合でも，クエリ・メソッドによって要求されたデータの構造が常に含まれます。データがコレクションまたはマップである多くのメソッドについて，応答には ChunkInfo 構造も含まれています。これは，chunksKey と numberOfChunks から構成されています。numberOfChunks フィールドは，取得する必要のあるデータを含むチャンクの数を示します。

データの最大転送サイズは，システム管理者が設定します。クエリから返されるデータが最大サイズより大きい場合は，最初の応答のデータ構造には意味のある情報は含まれず，numberOfChunks フィールドは 2 以上になります。データが最大サイズより大きくない場合，numberOfChunks フィールドは 0（ゼロ）になり，データは最初の応答時に転送されます。このため，応答を処理するときには，numberOfChunks 値を最初にチェックしてください。この値が 1 より大きい場合は，転送されたデータを破棄し，データのチャンクを要求します。この値が 1 を超えていない場合は，応答に含まれているデータを使用します。

チャンクに分割したデータの処理の詳細については，306 ページの「pullTopologyMapChunks」および 307 ページの「releaseChunks」を参照してください。

返されるプロパティの指定

CI と関係には，一般的に多くのプロパティがあります。これらのアイテムのコレクションまたはグラフを返す一部のメソッドは，クエリに一致する各アイテムについて，返すプロパティ値を指定する入力パラメータを受け付けます。CMDB は，空のプロパティを返しません。このため，クエリに対する応答では，クエリで指定したよりもプロパティが少ないことがあります。

本項では，返されるプロパティを指定するために使用するセットの種類について説明します。

プロパティを参照するには，次の 2 つの方法があります。

- 名前を使う。
- 事前に定義したプロパティ・ルールの名前を使う。事前に定義したプロパティ・ルールは，実際のプロパティ名のリストを作成するために CMDB によって使用されます。

アプリケーションが名前を使ってプロパティを参照する場合，アプリケーションによって PropertiesList 要素が渡されます。

ヒント：可能な場合には，ルール・ベースのセットではなく PropertiesList を使用して，必要なプロパティの名前を指定してください。事前に定義したプロパティ・ルールを使用すると，ほぼ必ず必要以上のプロパティが返されるため，パフォーマンスが低下します。

事前定義したプロパティには，修飾子（qualifier）プロパティとシンプル（simple）プロパティの 2 種類があります。

- **修飾子（Qualifier）プロパティ：**クライアント・アプリケーションが QualifierProperties 要素（プロパティに適用できる修飾子のリスト）を渡す必要がある場合に使用します。クライアント・アプリケーションによって渡された修飾子のリストは，CMDB によって，修飾子の少なくとも 1 つを適用するプロパティのリストに変換されます。これらのプロパティの値は，CI または Relation 要素とともに返されます。
- **シンプル（Simple）プロパティ：**シンプルなルール・ベースのプロパティを使用するには，クライアント・アプリケーションは，SimplePredefinedProperty または SimpleTypedPredefinedProperty 要素を渡します。これらの要素には，返すプロパティのリストを CMDB が生成するのに使うルールの名前が含まれています。SimplePredefinedProperty 要素または SimpleTypedPredefinedProperty 要素で指定できるルールは，CONCRETE，DERIVED，および NAMING です。

コンクリート（Concrete）プロパティ

コンクリート（Concrete）プロパティは，指定した CIT に対して定義されたプロパティのセットです。派生クラスによって追加されたプロパティは，これらの派生クラスのインスタンスに対しては返されません。

メソッドによって返されるインスタンスのコレクションは，メソッド呼び出しで指定した CIT のインスタンス，およびその CIT から継承した CIT のインスタンスから構成されます。派生した CIT は，指定した CIT のプロパティを継承します。また，派生した CIT は，プロパティを追加することによって親 CIT を拡張します。

コンクリート（Concrete）プロパティの例：

CIT T1 には，プロパティ P1 と P2 があります。CIT T11 は，T1 から継承し，T1 を，プロパティ P21 と P22 を使って拡張します。

T1 タイプの CI のコレクションには，T1 と T11 のインスタンスが含まれます。このコレクション内のすべてのインスタンスのコンクリート（concrete）プロパティは，P1 と P2 です。

派生（Derived）プロパティ

派生（Derived）プロパティは，指定した CIT に対して定義されたプロパティ，および派生 CIT ごとに，派生 CIT によって追加されたプロパティのセットです。

派生（Derived）プロパティの例 :

コンクリート（concrete）プロパティの例から続けると，T1 のインスタンスの派生（derived）プロパティは，P1 と P2 です。T11 のインスタンスの派生（derived）プロパティは，P1，P2，P21，および P22 です。

命名（Naming）プロパティ

命名（naming）プロパティには，display_label と data_name があります。

その他のプロパティ指定要素

- **PredefinedProperties**: PredefinedProperties は，利用可能なほかのルールごとに，QualifierProperties 要素と SimplePredefinedProperty 要素を含むことができます。PredefinedProperties のセットには，すべての種類のリストが含まれている必要はありません。
- **PredefinedTypedProperties**: PredefinedTypedProperties は，異なるセットのプロパティを各 CIT に適用するために使用します。PredefinedTypedProperties は，利用可能なほかのルールごとに，QualifierProperties 要素と SimpleTypedPredefinedProperty 要素を含むことができます。PredefinedTypedProperties は，各 CIT に個々に適用されるため，派生（derived）プロパティは関係ありません。PredefinedProperties のセットには，すべての適用可能な種類のリストが含まれている必要はありません。
- **CustomProperties**: CustomProperties は，基本的な PropertiesList とルール・ベースのプロパティ・リストの組み合わせを含むことができます。プロパティ・フィルタは，すべてのリストによって返されるすべてのプロパティを結合したものです。
- **CustomTypedProperties**: CustomTypedProperties は，基本的な PropertiesList と適用可能なルール・ベースのプロパティ・リストの組み合わせを含みます。プロパティ・フィルタは，すべてのリストによって返されるすべてのプロパティを結合したものです。
- **TypedProperties**: TypedProperties は, CIT ごとに異なるセットのプロパティを渡すために使用します。TypedProperties は，タイプ名と，すべてのタイプのプロパティ・セットから構成されるペアのコレクションです。各プロパティ・セットは，対応するタイプのみに適用されます。

UCMDB の更新

CMDB の更新は，更新 API を使って実施します。API メソッドの詳細については，308 ページの「UCMDB 更新メソッド」を参照してください。

更新メソッドの使用例については, 340 ページの「更新の例」を参照してください。

このタスクには次の手順が含まれます。

- 289 ページの「UCMDB の更新パラメータ」
- 290 ページの「更新メソッドを使った ID タイプの使用」
- 308 ページの「UCMDB 更新メソッド」

UCMDB の更新パラメータ

このトピックでは，サービスの更新メソッドによってのみ使用されるパラメータについて説明します。詳細については，スキーマのドキュメントを参照してください。

CIsAndRelationsUpdates

CIsAndRelationsUpdates タイプは，CIsForUpdate，relationsForUpdate，referencedRelations，および referencedCIs から構成されます。CIsAndRelationsUpdates インスタンスには，3 つの要素がすべて含まれている必要はありません。

CIsForUpdate は，CI コレクションです。relationsForUpdate は，Relations コレクションです。コレクション内の CI と relation 要素には，props 要素があります。CI または関係を作成するときは，required 属性または key 属性を CI タイプの定義に持っているプロパティに値をポピュレートする必要があります。これらのコレクション内のアイテムは，メソッドによって更新または作成されます。

referencedCIs および referencedRelations は，すでに CMDB に定義されている CI のコレクションです。コレクション内の要素は，すべてのキー・プロパティとともに一時 ID を使って識別されます。これらのアイテムは，更新するために CI と関係の識別に使用します。これらは，メソッドによって作成，更新されることはありません。

これらのコレクション内の各 CI 要素と relation 要素は，プロパティのコレクションを持っています。新しいアイテムは，これらのコレクション内のプロパティ値を使って作成されます。

更新メソッドを使った ID タイプの使用

次に，ID CIT，および CI と関係について説明します。ID が実際の CMDB ID でない場合，タイプ属性とキー属性が必要になります。

構成アイテムの削除と更新

アイテムを削除または更新するメソッドの呼び出し時に，一時 ID または空の ID が，クライアントによって使用されることがあります。この場合，CI を識別する CI タイプとキー属性を設定する必要があります。

関係の削除と更新

関係を削除または更新する場合，関係 ID は空，一時，または本物のいずれでもかまいません。

CI の ID が一時の場合，CI を referencedCIs コレクションで渡して，そのキー属性を指定する必要があります。詳細については，289 ページの「CIsAndRelationsUpdates」の「referencedCIs」を参照してください。

CMDB への新しい構成アイテムの挿入

空の ID または一時 ID を使用して新しい CI を挿入することができます。ただし，ID が空の場合，clientID がないため，サーバは createIDsMap 構造内の実際の CMDB ID を返すことができません。詳細については，308 ページの「addCIsAndRelations」と 294 ページの「UCMDB クエリ・メソッド」を参照してください。

CMDB への新しい関係の挿入

関係 ID は，一時的または空です。ただし，関係が新しく，関係のいずれか一方のエンドの構成アイテムが CMDB ですでに定義されている場合，すでに存在するこれらの CI は，実際の CMDB ID によって識別するか，または referencedCIs コレクションで指定する必要があります。

UCMDB クラス・モデルへの問い合わせ

クラス・モデル・メソッドは，CIT と関係に関する情報を返します。クラス・モデルは，CI タイプ・マネージャを使用して設定します。詳細については，『モデリング・ガイド』の「CI タイプ・マネージャ」を参照してください。

クラス・モデル・メソッドの使用例については，344 ページの「クラス・モデルの例」を参照してください。

本項では，CIT と関係に関する情報を返す，次のメソッドに関する情報を提供します。

- 291 ページの「getClassAncestors」
- 292 ページの「getAllClassesHierarchy」
- 292 ページの「getCmdbClassDefinition」

getClassAncestors

getClassAncestors メソッドは, 特定の CIT とそのルート間のパスを取得します（ルートを含む）。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
className	タイプ名。詳細については，358 ページの「タイプ名」を参照してください。

出力

パラメータ	コメント
classHierarchy	クラス名と親クラス名のペアのコレクション。
コメント	内部使用専用。

getAllClassesHierarchy

getAllClassesHierarchy メソッドは，クラス・モデル・ツリー全体を取得します。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。

出力

パラメータ	コメント
classesHierarchy	クラス名と親クラス名のペアのコレクション。
コメント	内部使用専用。

getCmdbClassDefinition

getCmdbClassDefinition メソッドは，指定したクラスに関する情報を取得します。

getCmdbClassDefinition を使用してキー属性を取得する場合，基本クラスとともに親クラスも問い合わせる必要があります。getCmdbClassDefinition は，className によって指定したクラス定義で設定した ID_ATTRIBUTE を持つ属性のみをキー属性として識別します。継承したキー属性は，指定したクラスのキー属性として認識されません。このため，指定したクラスのキー属性の完全なリストは，クラスとそのすべての親のキーをすべて結合したものです（ルートを含む）。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
className	タイプ名。詳細については，358 ページの「タイプ名」を参照してください。

出力

パラメータ	コメント
cmdbClass	name，classType，displayLabel，description，parentName，修飾子，および属性から構成されるクラス定義。
コメント	内部使用専用。

影響分析のための問い合わせ

影響分析メソッドの Identifier は，サービスの応答データをポイントします。識別子は現在の応答に固有のもので，10 分間使用されなければ，サーバのメモリ・キャッシュから破棄されます。

影響分析メソッドの使用例については，346 ページの「影響分析の例」を参照してください。

参照先

UCMDB クエリ・メソッド

本項では，次のメソッドに関する情報を提供します。

executeTopologyQueryByName

executeTopologyQueryByName メソッドは，指定したクエリに一致するトポロジ・マップを取得します。

ヒント：マップには多くの情報が含まれています。TQL 内の各 CINode と各 relationNode のラベルが一意であれば，理解しやすくなります。詳細については，281 ページの「明確なトポロジ・マップ要素を返す」を参照してください。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
queryName	マップの取得に使う CMDB 内の TQL の名前。
queryTypedProperties	特定の構成アイテム・タイプのアイテムを取得するための，プロパティのセットのコレクション。

出力

パラメータ	コメント
topologyMap	詳細については，360 ページの「TopologyMap」を参照してください。

executeTopologyQueryByNameWithParameters

executeTopologyQueryByNameWithParameters メソッドは，指定したパラメータ化されたクエリに一致する topologyMap 要素を取得します。

クエリ・パラメータの値は，parameterizedNodes 引数で渡されます。指定した TQL には，各 CINode および各 relationNode に対して一意のラベルが定義されている必要があります。定義されていなければ，メソッドの呼び出しは失敗します。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
queryName	マップを取得する CMDB 内のパラメータ化された TQL の名前。
parameterizedNodes	クエリ結果の対象となるために各ノードが満たす必要がある条件。
queryTypedProperties	特定の構成アイテム・タイプのアイテムを取得するための，プロパティのセットのコレクション。

出力

パラメータ	コメント
topologyMap	詳細については，360 ページの「TopologyMap」を参照してください。
chunkInfo	詳細については，361 ページの「ChunkInfo」および 285 ページの「サイズの大きい応答の処理」を参照してください。

executeTopologyQueryWithParameters

executeTopologyQueryWithParameters メソッドは，指定したパラメータ化されたクエリに一致する topologyMap 要素を取得します。

クエリは，queryXML 引数で渡されます。クエリ・パラメータの値は，parameterizedNodes 引数で渡されます。TQL には，各 CINode および各 relationNode に対して一意のラベルが定義されている必要があります。

executeTopologyQueryWithParameters メソッドは，CMDB で定義されているクエリにアクセスするためではなく，アドホック・クエリを渡すために使用します。このメソッドは，UCMDB ユーザ・インタフェースにアクセスしてクエリを定義する権限がないときや，クエリをデータベースに保存しないときに使用できます。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
queryXML	TQL を XML 形式で表現したもの。
parameterizedNodes	クエリ結果の対象となるために各ノードが満たす必要がある条件。

出力

パラメータ	コメント
topologyMap	詳細については，360 ページの「TopologyMap」を参照してください。
chunkInfo	詳細については，361 ページの「ChunkInfo」と 285 ページの「サイズの大きい応答の処理」を参照してください。

getChangedCIs

getChangedCIs メソッドは，指定した CI に関連するすべての CI の変更データを返します。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
ids	関連 CI の変更の有無がチェックされるルート CI の ID のリスト。 このコレクションでは，実際の CMDB ID だけが有効です。
fromDate	CI が変更されたかどうかをチェックする期間の開始点。
toDate	CI が変更されたかどうかをチェックする期間の終了点。

出力

パラメータ	コメント
changeDataInfo	ChangedDataInfo 要素のゼロ個以上のコレクション。

getCINeighbours

getCINeighbours メソッドは，指定した CI の隣接項目を返します。

たとえば，クエリが CI A の隣接項目を対象としており，CI A に，CI C を使う CI B が含まれている場合，CI B は返されますが，CI C は返されません。つまり，指定したタイプの隣接項目のみが返されます。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
ID	隣接項目の取得に使う CI の ID。この値は，実際の CMDB ID である必要があります。
neighbourType	取得する隣接項目の CIT 名。指定したタイプの隣接項目，およびそのタイプから派生したタイプの隣接項目が返されます。 詳細については，358 ページの「タイプ名」を参照してください。
CIProperties	各構成アイテム上の返されるデータ。ユーザ・インタフェースではクエリ・レイアウトと呼びます。 詳細については，288 ページの「TypedProperties: TypedProperties は，CIT ごとに異なるセットのプロパティを渡すために使用します。TypedProperties は，タイプ名と，すべてのタイプのプロパティ・セットから構成されるペアのコレクションです。各プロパティ・セットは，対応するタイプのみに適用されます。」を参照してください。
relationProperties	各関係上の返されるデータ。ユーザ・インタフェースではクエリ・レイアウトと呼びます。詳細については，288 ページの「TypedProperties: TypedProperties は，CIT ごとに異なるセットのプロパティを渡すために使用します。TypedProperties は，タイプ名と，すべてのタイプのプロパティ・セットから構成されるペアのコレクションです。各プロパティ・セットは，対応するタイプのみに適用されます。」を参照してください。

出力

パラメータ	コメント
topology	詳細については，360 ページの「Topology」を参照してください。
コメント	内部使用専用。

getCIsByID

getCIsByID メソッドは，CMDB ID を使って構成アイテムを取得します。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
CIsTypedProperties	タイプ別プロパティのコレクション。詳細については，288 ページの「その他のプロパティ指定要素」を参照してください。
IDs	このコレクションでは，実際の CMDB ID だけが有効です。

出力

パラメータ	コメント
CI	CI 要素のコレクション。
chunkInfo	詳細については，361 ページの「ChunkInfo」および 285 ページの「サイズの大きい応答の処理」を参照してください。

getCIsByType

getCIsByType メソッドは，指定したタイプ，および指定したタイプから継承するすべてのタイプの構成アイテムのコレクションを返します。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
type	クラス名。詳細については，358 ページの「タイプ名」を参照してください。
プロパティ	各構成アイテム上の返されるデータ。 詳細については，288 ページの「CustomProperties」を参照してください。

出力

パラメータ	コメント
CI	CI 要素のコレクション。
chunkInfo	詳細については，361 ページの「ChunkInfo」および 285 ページの「サイズの大きい応答の処理」を参照してください。

getFilteredCIsByType

getFilteredCIsByType メソッドは，メソッドで使用する条件を満たす指定したタイプの CI を取得します。条件には，次の要素が含まれます。

- プロパティの名前を含む名前フィールド。
- 比較演算子を含む演算子フィールド。
- 値または値のリストを含む任意指定の値フィールド。

これらによって，論理式を作成します。

```
<item>.property.value [operator] <condition>.value
```

たとえば，条件名が root_actualdeletionperiod で，条件値が 40 で，演算子が Equal の場合，論理式は次のようになります。

```
<item>.root_actualdeletionperiod.value = = 40
```

ほかに条件を指定しなければ，クエリによって，root_actualdeletionperiod が 40 のアイテムがすべて返されます。

conditionsLogicalOperator 引数が AND の場合，クエリによって，conditions コレクションで指定した条件をすべて満たすアイテムが返されます。conditionsLogicalOperator が OR の場合は，クエリによって，conditions コレクションで指定した条件の少なくとも 1 つを満たすアイテムが返されます。

次の表は，比較演算子を示します。

演算子	条件の種類とコメント
ChangedDuring	Date これにより範囲をチェックします。条件値は時間単位で指定します。date プロパティの値が，メソッドが呼び出された時点に条件値を加えた，または引いた範囲内にある場合，条件は true となります。 たとえば，条件値が 24 の場合，date プロパティの値が昨日のこの時刻と明日のこの時刻の間であれば，条件は true となります。 **注 :** ChangedDuring という名前は，下位互換性を維持するために予約されています。以前のバージョンでは，この演算子は Create Time，Modify Time にのみ使用されていました。
等価	文字列および数値
EqualIgnoreCase	文字列
より大きい	数値
GreaterEqual	数値
含む	文字列，数値，およびリスト 条件の値はリストです。プロパティの値がリスト内の値の 1 つであれば，条件は true になります。
InList	リスト 条件の値とプロパティの値はリストです。 条件のリスト内のすべての値がアイテムのプロパティ・リストにもあれば，条件は true になります。条件の真偽に影響を与えることなく，条件で指定したより多くのプロパティ値を利用できます。

演算子	条件の種類とコメント
IsNull	文字列，数値，およびリスト アイテムのプロパティに値がありません。演算子 IsNull を使用すると，条件の値は無視され，場合によっては nil 扱いとなります。
より小さい	数値
LessEqual	数値
類似	文字列 条件の値はプロパティ値の部分文字列です。条件の値は，パーセント記号（%）で囲む必要があります。たとえば，%Bi% は Bismark と Bay of Biscay に一致しますが，biscuit には一致しません。
LikeIgnoreCase	文字列 Like 演算子を使用するのと同様に，LikeIgnoreCase 演算子を使用します。ただし，大文字と小文字の区別は一致条件に入りません。このため，%Bi% は biscuit に一致します。
NotEqual	文字列および数値
UnchangedDuring	Date これにより範囲をチェックします。条件値は時間単位で指定します。date プロパティの値が，メソッドが呼び出された時点に条件値を加えた，または引いた範囲内にある場合，条件は false となります。この値が範囲外にある場合，条件は true となります。 たとえば，条件値が 24 の場合，date プロパティの値が昨日のこの時刻の前，または明日のこの時刻の後であれば，条件は true となります。 **注 :** UnchangedDuring という名前は，下位互換性を維持するために予約されています。以前のバージョンでは，この演算子は Create Time，Modify Time にのみ使用されていました。

条件設定の例：

```
FloatCondition fc = new FloatCondition();
FloatProp fp = new FloatProp();
fp.setName("attr_name");
fp.setValue(11);
fc.setCondition(fp);
fc.setFloatOperator(FloatCondition.floatOperatorEnum.Equal);
```

継承したプロパティへの問い合わせの例：

ターゲット CI は, name と size という 2 つの属性を持つ sample です。sampleII では，level と grade という 2 つの属性によって CI は拡張されます。この例では，名前を使って指定することによって，sample から継承された sampleII のプロパティに対するクエリをセットアップします。

```
GetFilteredCIsByType request = new GetFilteredCIsByType()
request.setCmdbContext(cmdbContext)
request.setType("sampleII")
CustomProperties customProperties = new CustomProperties();
PropertiesList propertiesList = new PropertiesList();
propertiesList.addPropertyName("name");
propertiesList.addPropertyName("size");
customProperties.setPropertiesList(propertiesList);
request.setProperties(customProperties)
```

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
type	クラス名。詳細については，358 ページの「タイプ名」を参照してください。タイプは，CI タイプ・マネージャを使用して定義したタイプのいずれでもかまいません。詳細については,『モデリング・ガイド』の「CI タイプ・マネージャ」を参照してください。

パラメータ	コメント
プロパティ	各 CI 上の返されるデータ（ユーザ・インタフェースではクエリ・レイアウトと呼びます）。 詳細については，288 ページの「CustomProperties: CustomProperties は，基本的な PropertiesList とルール・ベースのプロパティ・リストの組み合わせを含むことができます。プロパティ・フィルタは，すべてのリストによって返されるすべてのプロパティを結合したものです。」を参照してください。
conditions	名前と値のペアのコレクションと，一方を他方に関連付ける演算子。たとえば，host_hostname like QA などです。
conditionsLogicalOperator	➤ **AND** : すべての条件を満たす必要があります。 ➤ **OR** : 少なくとも 1 つの条件を満たす必要があります。

出力

パラメータ	コメント
CI	CI 要素のコレクション。
chunkInfo	詳細については，361 ページの「ChunkInfo」と 285 ページの「サイズの大きい応答の処理」を参照してください。

getQueryNameOfView

getQueryNameOfView メソッドは，指定したビューの基となる TQL の名前を取得します。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
viewName	ビューの名前。つまり CMDB 内のクラス・モデルのサブセット。

出力

パラメータ	コメント
queryName	ビューの基となる CMDB 内の TQL の名前。

getTopologyQueryExistingResultByName

getTopologyQueryExistingResultByName メソッドは，指定した TQL の最新の実行結果を取得します。呼び出しを実行しても TQL は実行されません。前回の実行結果が存在しない場合は，何も返しません。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
queryName	TQL の名前。
queryTypedProperties	特定の構成アイテム・タイプのアイテムを取得するための，プロパティのセットのコレクション。

出力

パラメータ	コメント
queryName	ビューの基となる CMDB 内の TQL の名前。

getTopologyQueryResultCountByName

getTopologyQueryResultCountByName メソッドは，指定したクエリに一致する各ノードのインスタンスの数を取得します。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
queryName	TQL の名前。
countInvisible	true の場合，クエリで非表示として定義された CI が出力に含まれます。

出力

パラメータ	コメント
queryName	ビューの基となる CMDB 内の TQL の名前。

pullTopologyMapChunks

pullTopologyMapChunks メソッドは，メソッドへの応答を含むチャンクの 1 つを取得します。

各チャンクは，応答の一部である topologyMap 要素を含みます。1 つ目のチャンクは 1 という番号が付けられているため，取得ループ・カウンタは，1 から < 応答オブジェクト >.getChunkInfo().getNumberOfChunks() を反復します。

詳細については，361 ページの「ChunkInfo」と 284 ページの「UCMDB への問い合わせ」を参照してください。

クライアント・アプリケーションは，部分的なマップを処理できる必要があります。CI コレクションの処理に関する次の例，およびチャンクをマップに結合する例 323 ページの「クエリの例」を参照してください。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
ChunkRequest	取得するチャンク，およびクエリ・メソッドによって返される ChunkInfo の数。

出力

パラメータ	コメント
topologyMap	詳細については，360 ページの「TopologyMap」を参照してください。
コメント	内部使用専用。

チャンクの処理例：

```
GetCIsByType request =
    new GetCIsByType(cmdbContext, typeName, customProperties);
GetCIsByTypeResponse response =
    ucmdbService.getCIsByType(request);
ChunkRequest chunkRequest = new ChunkRequest();
chunkRequest.setChunkInfo(response.getChunkInfo());
for(int j=1 ; j < response.getChunkInfo().getNumberOfChunks() ; j++) {
     chunkRequest.setChunkNumber(j);
     PullTopologyMapChunks req = new PullTopologyMapChunks(cmdbContext,
chunkRequest);
     PullTopologyMapChunksResponse res =
          ucmdbService.pullTopologyMapChunks(req);
          for(int m=0 ;
               m < res.getTopologyMap().getCINodes().sizeCINodeList() ;
               m++) {
               CIs cis =
                    res.getTopologyMap().getCINodes().getCINode(m).getCIs();
               for(int i=0 ; i < cis.sizeCIList() ; i++) {
                    // CI を処理するコード
               }
          }
}
```

releaseChunks

releaseChunks メソッドは，クエリからのデータを含むチャンクのメモリを解放します。

ヒント：10 分後にサーバによってデータは破棄されます。読み取りが終了したらすぐにデータを破棄するためにこのメソッドを呼び出すと，サーバのリソースを節約できます。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
chunksKey	チャンクに分けられたサーバ上のデータの識別子。キーは，ChunkInfo の要素です。

UCMDB 更新メソッド

本項では，次のメソッドに関する情報を提供します。

- 308 ページの「addCIsAndRelations」
- 309 ページの「addCustomer」
- 310 ページの「deleteCIsAndRelations」
- 310 ページの「removeCustomer」
- 310 ページの「updateCIsAndRelations」

addCIsAndRelations

addCIsAndRelations メソッドは，CI および関係を追加または更新します。

CI または関係が CMDB に存在しない場合は，これらは追加され，それぞれのプロパティが CIsAndRelationsUpdates 引数の内容に従って設定されます。

CI または関係が CMDB に存在する場合は，updateExisting が **true** であれば，これらは新しいデータを使って更新されます。

updateExisting が **false** の場合は，CIsAndRelationsUpdates は，既存の構成アイテムまたは関係を参照できません。updateExisting が false の場合に既存のアイテムを参照しようとすると，例外が発生します。

updateExisting が **true** であれば，ignoreValidation の値に関係なく，CI を検証することなく追加操作または更新操作が実行されます。

updateExisiting が **false** で，ignoreValidation が **true** の場合，CI を検証することなく追加操作が実行されます。

updateExisiting が **false** で ignoreValidation が **false** の場合，追加操作の前に CI が検証されます。

関係は検証されません。

CreatedIDsMap は，クライアントの一時 ID を，対応する実際の CMDB ID に結び付ける ClientIDToCmdbID タイプのマップまたは辞書です。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
updateExisting	true に設定すると，CMDB にすでに存在するアイテムが更新されます。false に設定すると，アイテムが存在する場合に例外がスローされます。
CIsAndRelationsUpdates	更新または作成するアイテム。詳細については，289 ページの「CIsAndRelationsUpdates」を参照してください。
ignoreValidation	true の場合，CMDB を更新する前にチェックは行われません。

出力

パラメータ	コメント
CreatedIDsMap	クライアント ID と CMDB ID のマップ。詳細については，308 ページの「addCIsAndRelations」を参照してください。
コメント	内部使用専用。

addCustomer

addCustomer メソッドはカスタマを追加します。

入力

パラメータ	コメント
CustomerID	カスタマの数値 ID。

deleteCIsAndRelations

deleteCIsAndRelations メソッドは, 指定した構成アイテムと関係を CMDB から削除します。

CI を削除して，CI が 1 つ以上の Relation アイテムの一方のエンドにしかない場合，これらの Relation アイテムも削除されます。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
CIsAndRelationsUpdates	削除するアイテム。詳細については，289 ページの「CIsAndRelationsUpdates」を参照してください。

removeCustomer

removeCustomer メソッドはカスタマ・レコードを削除します。

入力

パラメータ	コメント
CustomerID	カスタマの数値 ID。

updateCIsAndRelations

updateCIsAndRelations メソッドは，指定した CI と関係を更新します。

更新には，CIsAndRelationsUpdates 引数のプロパティ値が使用されます。CI または関係のいずれかが CMDB に存在しない場合，例外がスローされます。

CreatedIDsMap は，クライアントの一時 ID を，対応する実際の CMDB ID に結び付ける ClientIDToCmdbID タイプのマップまたは辞書です。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
CIsAndRelationsUpdates	更新するアイテム。詳細については，289 ページの「CIsAndRelationsUpdates」を参照してください。
ignoreValidation	true の場合，CMDB を更新する前にチェックは行われません。

出力

パラメータ	コメント
CreatedIDsMap	クライアント ID と CMDB ID のマップ。詳細については，308 ページの「addCIsAndRelations」を参照してください。

UCMDB の影響分析メソッド

本項では，次のメソッドに関する情報を提供します。

➤ 311 ページの「calculateImpact」

➤ 312 ページの「getImpactPath」

➤ 313 ページの「getImpactRulesByNamePrefix」

calculateImpact

calculateImpact メソッドは，CMDB に定義したルールに従って，どの CI が特定の CI の影響を受けるかを計算します。

これにより，ルールのイベント・トリガの効果がわかります。calculateImpact の identifier 出力は，getImpactPath の入力として使用します。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
impactCategory	シミュレートするルールを起動するイベントのタイプ。
IDs	ID 要素のコレクション。
impactRulesNames	ImpactRuleName 要素のコレクション。
severity	トリガ・イベントの重要度。

出力

パラメータ	コメント
impactTopology	詳細については，360 ページの「Topology」を参照してください。
identifier	サーバ応答に対するキー。

getImpactPath

getImpactPath メソッドは，影響を受ける CI と影響を与える CI の間のパスのトポロジ・グラフを取得します。

calculateImpact の identifier 出力は，getImpactPath の identifier 入力引数として使用します。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
identifier	calculateImpact によって返されたサーバ応答に対するキー。
relation	impactTopology 要素の calculateImpact によって返された ShallowRelation の 1 つに基づく関係。

出力

パラメータ	コメント
impactPathTopology	CIs コレクションと ImpactRelations コレクション。
コメント	内部使用専用。

ImpactRelations 要素は，ID，type，end1ID，end2ID，rule および action から構成されます。

getImpactRulesByNamePrefix

getImpactRulesByNamePrefix メソッドは，プレフィックス・フィルタを使用してルールを取得します。

このメソッドは，適用先の内容を示すプレフィックスを名前に含む影響ルールに適用されます。たとえば，SAP_myrule，ORA_myrule などです。このメソッドは，すべての影響ルール名をフィルタして，ruleNamePrefixFilter 引数で指定したプレフィックスで始まるものを探します。

入力

パラメータ	コメント
cmdbContext	詳細については，356 ページの「CmdbContext」を参照してください。
ruleNamePrefixFilter	一致するルール名の最初の文字を含む文字列。

出力

パラメータ	コメント
impactRules	impactRules は，ゼロ個以上の impactRule から構成されます。変更の効果を指定する impactRule は，ruleName，description，queryName，および isActive から構成されます。

Data Flow Management のメソッド

本項では，Web サービス操作とその使用方法の要約の一覧を示します。各操作の要求と応答の完全な説明については，『Data Flow Management Schema Reference』（英語版）を参照してください。

本項の内容

DFM ジョブ・メソッドの管理

- **activateJob**

  指定されたジョブをアクティブにします。

- **deactivateJob**

  指定されたジョブを非アクティブにします。

- **dispatchAdHocJob**

  プローブに対してジョブを一時的にディスパッチします。プローブに対してジョブを一時的にディスパッチします。ジョブはアクティブである必要があり，指定されたトリガ CI を含んでいる必要があります。

- **getDiscoveryJobsNames**

  ジョブ名のリストを返します。

- **isJobActive**

  ジョブがアクティブかどうかをチェックします。

トリガ・メソッドの管理

- **addTriggerCI**

  指定されたジョブに新しいトリガ CI を追加します。

➤ **addTriggerTQL**

指定されたジョブに新しいトリガ TQL を追加します。

➤ **disableTriggerTQL**

TQL がジョブを起動しないようにしますが，ジョブを起動するクエリのリストからその TQL を永久的に削除することはありません。

➤ **removeTriggerCI**

ジョブを起動する CI のリストから，指定された CI を削除します。

➤ **removeTriggerTQL**

ジョブを起動するクエリのリストから，指定された TQL を削除します。

➤ **setTriggerTQLProbesLimit**

指定されたリストに対して，ジョブ内で TQL がアクティブになるプローブを制限します。

ドメインおよび Probe データ・メソッド

➤ **getDomainType**

ドメイン・タイプを返します。

➤ **getDomainsNames**

現在のドメインの名前を返します。

➤ **getProbeIPs**

指定されたプローブの IP アドレスを返します。

➤ **getProbesNames**

指定されたドメイン内のプローブの名前を返します。

➤ **getProbeScope**

指定されたプローブの対象範囲の定義を返します。

➤ **isProbeConnected**

指定されたプローブが接続されているかどうかをチェックします。

➤ **updateProbeScope**

指定されたプローブの対象範囲を設定し，既存の範囲を上書きします。

資格情報データ・メソッド

➤ **addCredentialsEntry**

指定されたドメインについて，指定されたプロトコルに資格情報エントリを追加します。

➤ **getCredentialsEntriesIDs**

指定されたプロトコルについて定義された資格情報の ID を返します。

➤ **getCredentialsEntry**

指定されたプロトコルについて定義された資格情報を返します。暗号化された属性は空のデータとして返されます。

➤ **removeCredentialsEntry**

指定された資格情報をプロトコルから削除します。

➤ **updateCredentialsEntry**

指定された資格情報エントリのプロパティに新しい値を設定します。

データ更新メソッド

➤ **rediscoverCIs**

指定された CI オブジェクトを検出したトリガを見つけて，それらのトリガを返します（再実行コマンドは，ほかのスケジュールされたアイテムよりも優先度が高くなります）。

rediscoverCIs は非同期的に実行されます。再検出がいつ完了するかを調べるには，**checkDiscoveryProgress** を呼び出します。

➤ **checkDiscoveryProgress**

指定された ID について，最新の **rediscoverCIs** 呼び出しの進行状況を返します。応答は，0 から 1 までの値です。応答が 1 の場合，**rediscoverCIs** 呼び出しは完了しています。

➤ **rediscoverViewCIs**

指定されたビューに表示されるデータを作成したトリガを見つけて，それらのトリガを返します（再実行コマンドは，ほかのスケジュールされたアイテムよりも優先度が高くなります）。

rediscoverViewCIs は非同期的に実行されます。再検出がいつ完了するかを調べるには，**checkViewDiscoveryProgress** を呼び出します。

➤ **checkViewDiscoveryProgress**

指定されたビューについて，最新の **rediscoverViewCIs** 呼び出しの進行状況を返します。応答は，0 から 1 までの値です。応答が 1 の場合，**rediscoverCIs** 呼び出しは完了しています。

使用例

次の使用例は 2 つのシステムを想定しています。

➤ HP Universal CMDB サーバ

➤ 構成アイテムのリポジトリを含むサードパーティ製のシステム

本項の内容

CMDB のポピュレート

使用例：

➤ サードパーティ製のアセット管理は，アセット管理でのみ使用できる情報で CMDB を更新します。

➤ 多くのサードパーティ製のシステムは，CMDB にデータをポピュレートして，変更内容を追跡し影響分析を実行できる中心的な CMDB を作成します。

➤ サードパーティ製のシステムは，サードパーティのビジネス・ロジックに従って構成アイテムと関係を作成し，CMDB クエリ機能を活用します。

CMDB への問い合わせ

使用例：

- サードパーティ製のシステムは，SAP TQL の結果を取得することによって，SAP システムを表す構成アイテムと関係を取得します。
- サードパーティ製のシステムは，過去 5 時間以内に追加または変更された Oracle サーバのリストを取得します。
- サードパーティ製のシステムは，ホスト名に部分文字列 lab が含まれるサーバのリストを取得します。
- サードパーティ製のシステムは，隣接項目を取得することによって，特定の CI に関係する要素を検出します。

クラス・モデルへの問い合わせ

使用例：

- サードパーティ製のシステムでは，ユーザは CMDB から取得するデータのセットを指定できます。ユーザ・インタフェースはクラス・モデル上に構築し，ユーザに利用可能なプロパティを表示して，必要なデータを求めることができます。ユーザは，取得する情報を選択できます。
- サードパーティ製のシステムは，ユーザが UCMDB ユーザ・インタフェースにアクセスできないときに，クラス・モデルを探索します。

変更の影響の分析

使用例：

サードパーティ製のシステムは，指定したホストに対する変更の影響を受けるビジネス・サービスのリストを出力します。

例

本項の内容

➤ 340 ページの「更新の例」
➤ 344 ページの「クラス・モデルの例」
➤ 346 ページの「影響分析の例」
➤ 351 ページの「資格情報の追加の例」

基本クラスの例

```
package com.hp.ucmdb.demo;

import com.hp.ucmdb.generated.services.UcmdbService;
import com.hp.ucmdb.generated.services.UcmdbServiceStub;
import com.hp.ucmdb.generated.types.CmdbContext;
import org.apache.axis2.AxisFault;
import org.apache.axis2.transport.http.HTTPConstants;
```

```
import org.apache.axis2.transport.http.HttpTransportProperties;

import java.net.MalformedURLException;
import java.net.URL;
```

```
/**
* User: hbarkai
* Date: Jul 12, 2007
*/
abstract class Demo {
```

```
UcmdbService stub;
CmdbContext context;
```

```
public void initDemo() {
    try {
        setStub(createUcmdbService("admin", "admin"));
        setContext();
    } catch (Exception e) {
        // 例外の処理
    }
}
```

```
public UcmdbService getStub() {
    return stub;
}
```

```
public void setStub(UcmdbService stub) {
    this.stub = stub;
}
```

```
public CmdbContext getContext() {
    return context;
}
```

```
public void setContext() {
    CmdbContext context = new CmdbContext();
    context.setCallerApplication("demo");
    this.context = context;
}
```

```
// サービスへの接続 - axis2/jibx クライアント用

private static final String PROTOCOL = "http";
private static final String HOST_NAME = "host_name";
private static final int PORT = 8080;
private static final String FILE = "/axis2/services/UcmdbService";

protected UcmdbService createUcmdbService
    (String username, String password) throws Exception{
        URL url;
        UcmdbServiceStub serviceStub;
```

```
        try {
            url = new URL
                (Demo.PROTOCOL, Demo.HOST_NAME,
                Demo.PORT, Demo.FILE);
            serviceStub = new UcmdbServiceStub(url.toString());
            HttpTransportProperties.Authenticator auth =
                new HttpTransportProperties.Authenticator();
            auth.setUsername(username);
            auth.setPassword(password);
serviceStub._getServiceClient().getOptions().setProperty
    (HTTPConstants.AUTHENTICATE,auth);
```

```
    } catch (AxisFault axisFault) {
            throw new Exception
            ("Failed to create SOAP adapter for "
                + Demo.HOST_NAME , axisFault);
```

```
        } catch (MalformedURLException e) {

            throw new Exception
                ("Failed to create SOAP adapter for "
                    + Demo.HOST_NAME, e);
        }
        return serviceStub;
 }
}
```

クエリの例

```
package com.hp.ucmdb.demo;

import com.hp.ucmdb.generated.params.query.*;
import com.hp.ucmdb.generated.services.UcmdbFaultException;
import com.hp.ucmdb.generated.services.UcmdbService;
import com.hp.ucmdb.generated.types.*;
import com.hp.ucmdb.generated.types.props.*;

import java.rmi.RemoteException;
```

```
public class QueryDemo extends Demo{

    UcmdbService stub;
    CmdbContext context;
```

```
    public void getCIsByTypeDemo() {
        GetCIsByType request = new GetCIsByType();
        // cmdbcontext の設定
        CmdbContext cmdbContext = getContext();
        request.setCmdbContext(cmdbContext);
        // CI タイプの設定
        request.setType("anyType");
        // 取得する CI プロパティの設定
        CustomProperties customProperties = new CustomProperties();
        PredefinedProperties predefinedProperties =
            new PredefinedProperties();
        SimplePredefinedProperty simplePredefinedProperty =
            new SimplePredefinedProperty();
        simplePredefinedProperty.setName
            (SimplePredefinedProperty.nameEnum.DERIVED);
        SimplePredefinedPropertyCollection
            simplePredefinedPropertyCollection =
                new SimplePredefinedPropertyCollection();
```

```
        simplePredefinedPropertyCollection.addSimplePredefinedProperty
            (simplePredefinedProperty);
        predefinedProperties.setSimplePredefinedProperties
            (simplePredefinedPropertyCollection);
        customProperties.setPredefinedProperties(predefinedProperties);
        request.setProperties(customProperties);
        try {
            GetCIsByTypeResponse response =
                getStub().getCIsByType(request);
            TopologyMap map =
                getTopologyMapResultFromCIs
                    (response.getCIs(), response.getChunkInfo());
        } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }
    }
```

```
    public void getCIsByIdDemo() {
        GetCIsById request = new GetCIsById();
        CmdbContext cmdbContext = getContext();
        // cmdbcontext の設定
        request.setCmdbContext(cmdbContext);
        // ID の設定
        ID id1 = new ID();
        id1.setBase("cmdbobjectidCIT1");
        ID id2 = new ID();
        id2.setBase("cmdbobjectidCIT2");
        IDs ids = new IDs();
        ids.addID(id1);
        ids.addID(id2);
        request.setIDs(ids);
        // 取得する CI プロパティの設定
        TypedPropertiesCollection properties =
            new TypedPropertiesCollection();
```

```
    TypedProperties typedProperties1 =
            new TypedProperties();
        typedProperties1.setType("CIT1");
```

```
        CustomTypedProperties customProperties1 =
            new CustomTypedProperties();
        PredefinedTypedProperties predefinedProperties1 =
            new PredefinedTypedProperties();
        SimpleTypedPredefinedProperty simplePredefinedProperty1 =
            new SimpleTypedPredefinedProperty();
        simplePredefinedProperty1.setName
            (SimpleTypedPredefinedProperty.nameEnum.CONCRETE);
        SimpleTypedPredefinedPropertyCollection
            simplePredefinedPropertyCollection1 =
                new SimpleTypedPredefinedPropertyCollection();
        simplePredefinedPropertyCollection1
            .addSimpleTypedPredefinedProperty
                (simplePredefinedProperty1);
```

```
predefinedProperties1.
        setSimpleTypedPredefinedProperties
            (simplePredefinedPropertyCollection1);
        customProperties1.
            setPredefinedTypedProperties
                (predefinedProperties1);
        typedProperties1.setProperties(customProperties1);
        properties.addTypedProperties(typedProperties1);
```

```
        TypedProperties typedProperties2 =
            new TypedProperties();
        typedProperties2.setType("CIT2");
        CustomTypedProperties customProperties2 =
            new CustomTypedProperties();
        PredefinedTypedProperties predefinedProperties2 =
            new PredefinedTypedProperties();
        SimpleTypedPredefinedProperty simplePredefinedProperty2 =
            new SimpleTypedPredefinedProperty();
        simplePredefinedProperty2.setName
            (SimpleTypedPredefinedProperty.nameEnum.NAMING);
        SimpleTypedPredefinedPropertyCollection
            simplePredefinedPropertyCollection2 =
                new SimpleTypedPredefinedPropertyCollection();
```

```
        simplePredefinedPropertyCollection2.
            addSimpleTypedPredefinedProperty
                (simplePredefinedProperty2);
```

```
        predefinedProperties2.setSimpleTypedPredefinedProperties
            (simplePredefinedPropertyCollection2);
        customProperties2.setPredefinedTypedProperties
            (predefinedProperties2);
        typedProperties2.setProperties(customProperties2);
        properties.addTypedProperties(typedProperties2);
```

```
    request.setCIsTypedProperties(properties);
        try {
            GetCIsByIdResponse response =
                getStub().getCIsById(request);
            CIs cis = response.getCIs();
        } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }

    }
```

```
    public void getFilteredCIsByTypeDemo() {
        GetFilteredCIsByType request = new GetFilteredCIsByType();
        CmdbContext cmdbContext = getContext();
        // cmdbcontext の設定
        request.setCmdbContext(cmdbContext);
        // CI タイプの設定
        request.setType("anyType");
        // フィルタ条件の設定
        Conditions conditions = new Conditions();
        IntConditions intConditions = new IntConditions();
        IntCondition intCondition = new IntCondition();
        IntProp intProp = new IntProp();
        intProp.setName("int_attr1");
```

```
intProp.setValue(100);
intCondition.setCondition(intProp);
intCondition.setIntOperator
    (IntCondition.intOperatorEnum.Greater);
intConditions.addIntCondition(intCondition);
```

```
conditions.setIntConditions(intConditions);
request.setConditions(conditions);
// 条件の論理演算子の設定
request.setConditionsLogicalOperator
    (GetFilteredCIsByType.conditionsLogicalOperatorEnum.AND);
// 取得する CI プロパティの設定
CustomProperties customProperties =
    new CustomProperties();
PredefinedProperties predefinedProperties =
    new PredefinedProperties();
SimplePredefinedProperty simplePredefinedProperty =
    new SimplePredefinedProperty();
simplePredefinedProperty.setName
    (SimplePredefinedProperty.nameEnum.NAMING);
```

```
SimplePredefinedPropertyCollection
    simplePredefinedPropertyCollection =
        new SimplePredefinedPropertyCollection();
simplePredefinedPropertyCollection.
    addSimplePredefinedProperty
        (simplePredefinedProperty);
predefinedProperties.setSimplePredefinedProperties
    (simplePredefinedPropertyCollection);
customProperties.setPredefinedProperties
    (predefinedProperties);
```

```
request.setProperties(customProperties);
try {
    GetFilteredCIsByTypeResponse response =
        getStub().getFilteredCIsByType(request);
    TopologyMap map =
        getTopologyMapResultFromCIs
            (response.getCIs(), response.getChunkInfo());
```

```
        } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }

    }
```

```
    public void executeTopologyQueryByNameDemo() {
        ExecuteTopologyQueryByName request = new
ExecuteTopologyQueryByName();
        CmdbContext cmdbContext = getContext();
        // cmdbcontext の設定
        request.setCmdbContext(cmdbContext);
        // クエリ名の設定
        request.setQueryName("queryName");
```

```
        try {
            ExecuteTopologyQueryByNameResponse response =
                getStub().executeTopologyQueryByName(request);
            TopologyMap map =
                getTopologyMapResult
                    (response.getTopologyMap(), response.getChunkInfo());
        } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }

    }
```

```
/

// クエリ名 : exampleQuery
// クエリの概略図 :
//                              ホスト
//                              / \
//                              ip  ディスク
// クエリのパラメータ :
//    ホスト -
//         host_os (like)
//    ディスク -
//         disk_failures (equal)
```

```
public void executeTopologyQueryByNameWithParametersDemo() {
    ExecuteTopologyQueryByNameWithParameters request =
        new ExecuteTopologyQueryByNameWithParameters();
    CmdbContext cmdbContext = getContext();
    // cmdbcontext の設定
    request.setCmdbContext(cmdbContext);
    // クエリ名の設定
    request.setQueryName("queryName");
    // パラメータの設定
    ParameterizedNode hostParametrizedNode =
        new ParameterizedNode();
    hostParametrizedNode.setNodeLabel("Host");
    CIProperties parameters = new CIProperties();
    StrProps strProps = new StrProps();
    StrProp strProp = new StrProp();
    strProp.setName("host_os");
    strProp.setValue("%2000%");
    strProps.addStrProp(strProp);
    parameters.setStrProps(strProps);
    hostParametrizedNode.setParameters(parameters);
    request.addParameterizedNodes(hostParametrizedNode);
    ParameterizedNode diskParametrizedNode =
        new ParameterizedNode();
```

```
    diskParametrizedNode.setNodeLabel("Disk");
    CIProperties parameters1 = new CIProperties();
    IntProps intProps = new IntProps();
```

```
IntProp intProp = new IntProp();
    intProp.setName("disk_failures");
    intProp.setValue(30);
    intProps.addIntProp(intProp);
    parameters1.setIntProps(intProps);
    diskParametrizedNode.setParameters(parameters1);
```

```
    request.addParameterizedNodes(diskParametrizedNode);
    try {
        ExecuteTopologyQueryByNameWithParametersResponse
            response =
            getStub().executeTopologyQueryByNameWithParameters
                (request);
        TopologyMap map =
            getTopologyMapResult
                (response.getTopologyMap(), response.getChunkInfo());
    } catch (RemoteException e) {
        // 例外の処理
    } catch (UcmdbFaultException e) {
        // 例外の処理
    }

}
```

```
/   // 次のクエリが UCMDB で定義されていると想定
    // クエリ名 : exampleQuery
    // クエリの概略図 :
    //                          ホスト
    //                          / \
    //                          ip  ディスク
    // クエリのパラメータ :
    //    ホスト -
    //        host_os (like)
    //    ディスク -
    //        disk_failures (equal)
```

```
public void executeTopologyQueryWithParametersDemo() {
    ExecuteTopologyQueryWithParameters request =
        new ExecuteTopologyQueryWithParameters();
    CmdbContext cmdbContext = getContext();
    // cmdbcontext の設定
    request.setCmdbContext(cmdbContext);
    // クエリ定義の設定
    String queryXml = "<xml that represents the query above>";
    request.setQueryXml(queryXml);
    // パラメータの設定
    ParameterizedNode hostParametrizedNode =
        new ParameterizedNode();
```

```
hostParametrizedNode.setNodeLabel("Host");
    CIProperties parameters = new CIProperties();
    StrProps strProps = new StrProps();
    StrProp strProp = new StrProp();
    strProp.setName("host_os");
    strProp.setValue("%2000%");
    strProps.addStrProp(strProp);
    parameters.setStrProps(strProps);
    hostParametrizedNode.setParameters(parameters);
    request.addParameterizedNodes(hostParametrizedNode);
    ParameterizedNode diskParametrizedNode =
        new ParameterizedNode();
    diskParametrizedNode.setNodeLabel("Disk");
    CIProperties parameters1 = new CIProperties();
    IntProps intProps = new IntProps();
    IntProp intProp = new IntProp();
    intProp.setName("disk_failures");
    intProp.setValue(30);
    intProps.addIntProp(intProp);
    parameters1.setIntProps(intProps);
    diskParametrizedNode.setParameters(parameters1);
    request.addParameterizedNodes(diskParametrizedNode);
```

```
        try {
            ExecuteTopologyQueryWithParametersResponse
            response = getStub().executeTopologyQueryWithParameters
                (request);
            TopologyMap map =
                getTopologyMapResult
                    (response.getTopologyMap(), response.getChunkInfo());
```

```
         } catch (RemoteException e) {
             // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }

    }
```

```
    public void getCINeighboursDemo() {
        GetCINeighbours request = new GetCINeighbours();
        // cmdbcontext の設定
        CmdbContext cmdbContext = getContext();
        request.setCmdbContext(cmdbContext);
        // CI の ID 設定
        ID id = new ID();
        id.setBase("cmdbobjectidCIT1");
        request.setID(id);
        // 隣接項目のタイプの設定
        request.setNeighbourType("neighbourType");
        // 取得する隣接 CI のプロパティの設定
        TypedPropertiesCollection properties =
            new TypedPropertiesCollection();
        TypedProperties typedProperties1 = new TypedProperties();
        typedProperties1.setType("neighbourType");
        CustomTypedProperties customProperties1 =
            new CustomTypedProperties();
        PredefinedTypedProperties predefinedProperties1 =
            new PredefinedTypedProperties();
```

```
QualifierProperties qualifierProperties =
    new QualifierProperties();
qualifierProperties.addQualifierName("ID_ATTRIBUTE");
predefinedProperties1.setQualifierProperties(qualifierProperties);
customProperties1.setPredefinedTypedProperties
    (predefinedProperties1);
typedProperties1.setProperties(customProperties1);
properties.addTypedProperties(typedProperties1);
request.setCIProperties(properties);
```

```
TypedPropertiesCollection relationsProperties =
    new TypedPropertiesCollection();
TypedProperties typedProperties2 = new TypedProperties();
typedProperties2.setType("relationType");
CustomTypedProperties customProperties2 =
    new CustomTypedProperties();
```

```
PredefinedTypedProperties predefinedProperties2 =
    new PredefinedTypedProperties();
SimpleTypedPredefinedProperty simplePredefinedProperty2 =
    new SimpleTypedPredefinedProperty();
simplePredefinedProperty2.setName
```

```
    (SimpleTypedPredefinedProperty.nameEnum.CONCRETE);
SimpleTypedPredefinedPropertyCollection
    simplePredefinedPropertyCollection2 =
        new SimpleTypedPredefinedPropertyCollection();
simplePredefinedPropertyCollection2.
    addSimpleTypedPredefinedProperty
        (simplePredefinedProperty2);
predefinedProperties2.
    setSimpleTypedPredefinedProperties
        (simplePredefinedPropertyCollection2);
customProperties2.setPredefinedTypedProperties
    (predefinedProperties2);
typedProperties2.setProperties(customProperties2);
relationsProperties.addTypedProperties(typedProperties2);
request.setRelationProperties(relationsProperties);
```

```
        try {
            GetCINeighboursResponse response =
                getStub().getCINeighbours(request);
            Topology topology = response.getTopology();
    } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
             // 例外の処理
        }

    }
```

```
// チャンク化された結果 / チャンク化されていない結果のトポロジ・マップの取得

    private TopologyMap getTopologyMapResult(TopologyMap topologyMap, ChunkInfo
chunkInfo) {
        if(chunkInfo.getNumberOfChunks() == 0) {
            return topologyMap;
        } else {
```

```
            topologyMap = new TopologyMap();
            for(int i=1 ; i <= chunkInfo.getNumberOfChunks() ; i++) {
                ChunkRequest chunkRequest = new ChunkRequest();
                chunkRequest.setChunkInfo(chunkInfo);
                chunkRequest.setChunkNumber(i);
                PullTopologyMapChunks req =
                    new PullTopologyMapChunks();
                req.setChunkRequest(chunkRequest);
                req.setCmdbContext(getContext());
                PullTopologyMapChunksResponse res = null;
```

```
                try {
                    res = getStub().pullTopologyMapChunks(req);
                    TopologyMap map = res.getTopologyMap();
                    topologyMap = mergeMaps(topologyMap, map);
                } catch (RemoteException e) {
                    // 例外の処理
                } catch (UcmdbFaultException e) {
                    // 例外の処理
                }
            }
        }
        return topologyMap;

    }
```

```
    private TopologyMap getTopologyMapResultFromCIs(CIs cis, ChunkInfo chunkInfo)
{
        TopologyMap topologyMap = new TopologyMap();
        if(chunkInfo.getNumberOfChunks() == 0) {
            CINode ciNode = new CINode();
            ciNode.setLabel("");
            ciNode.setCIs(cis);
            CINodes ciNodes = new CINodes();
            ciNodes.addCINode(ciNode);
            topologyMap.setCINodes(ciNodes);
        } else {
```

```
            for(int i=1 ; i <= chunkInfo.getNumberOfChunks() ; i++) {
                ChunkRequest chunkRequest =
                    new ChunkRequest();
                chunkRequest.setChunkInfo(chunkInfo);
                chunkRequest.setChunkNumber(i);
                PullTopologyMapChunks req =
                    new PullTopologyMapChunks();
                req.setChunkRequest(chunkRequest);
                req.setCmdbContext(getContext());
                PullTopologyMapChunksResponse res = null;
```

```
                try {
                    res = getStub().pullTopologyMapChunks(req);
                } catch (RemoteException e) {
                    // 例外の処理
                } catch (UcmdbFaultException e) {
                    // 例外の処理
                }
                TopologyMap map = res.getTopologyMap();
                topologyMap = mergeMaps(topologyMap, map);
            }
```

```
            // チャンクの解放
            ReleaseChunks req = new ReleaseChunks();
            req.setChunksKey(chunkInfo.getChunksKey());
            req.setCmdbContext(getContext());
```

```
            try {
                getStub().releaseChunks(req);
            } catch (RemoteException e) {
                // 例外の処理
            } catch (UcmdbFaultException e) {
                // 例外の処理
            }
        }
        return topologyMap;

    }
```

```
//=====================================================
/* 警告 : 結合は，各ノードに
    一意の名前が付けられている場合のみ 正しく行われます。This applies to both CI
and Relation nodes .*/
//=====================================================
    private TopologyMap mergeMaps(TopologyMap topologyMap, TopologyMap
newMap) {
        for(int i=0 ; i < newMap.getCINodes().sizeCINodeList() ; i++ ) {
            CINode ciNode = newMap.getCINodes().getCINode(i);
            boolean alreadyExist = false;
            if(topologyMap.getCINodes() == null) {
                topologyMap.setCINodes(new CINodes());
            }
```

```
        for(int j=0 ; j < topologyMap.getCINodes().sizeCINodeList() ; j++) {
            CINode ciNode2 = topologyMap.getCINodes().getCINode(j);
            if(ciNode2.getLabel().equals(ciNode.getLabel())){
```

```
                CIs cisTOAdd = ciNode.getCIs();
                CIs cis =
                    mergeCIsGroups
                    (topologyMap.getCINodes().getCINode(j).getCIs(),
                        cisTOAdd);
                topologyMap.getCINodes().getCINode(j).setCIs(cis);
                alreadyExist = true;
            }
        }
        if(!alreadyExist) {
            topologyMap.getCINodes().addCINode(ciNode);
        }

    }
```

```
    for(int i=0 ; i < newMap.getRelationNodes().sizeRelationNodeList() ; i++ ) {
        RelationNode relationNode =
            newMap.getRelationNodes().getRelationNode(i);
        boolean alreadyExist = false;
        if(topologyMap.getRelationNodes() == null) {
            topologyMap.setRelationNodes(new RelationNodes());
        }
```

```
            for(int j=0 ;
                    j < topologyMap.getRelationNodes().sizeRelationNodeList() ;
                    j++) {
                RelationNode relationNode2 =
                    topologyMap.getRelationNodes().getRelationNode(j);
                if(relationNode2.getLabel().equals(relationNode.getLabel())){
                    Relations relationsTOAdd = relationNode.getRelations();
                    Relations relations =
                        mergeRelationsGroups
                        (topologyMap.getRelationNodes().
                            getRelationNode(j).getRelations(),
                            relationsTOAdd);
                    topologyMap.getRelationNodes().
                        getRelationNode(j).setRelations(relations);
                    alreadyExist = true;
                }
            }
```

```
            if(!alreadyExist) {
                topologyMap.getRelationNodes().addRelationNode(relationNode);
            }
        }

        return topologyMap;

    }
```

```
    private Relations mergeRelationsGroups(Relations relations1, Relations relations2)
{
        for(int i=0 ; i < relations2.sizeRelationList() ; i++) {
            relations1.addRelation(relations2.getRelation(i));
        }
        return relations2;
    }
```

```
    private CIs mergeCIsGroups(CIs cis1, CIs cis2) {
        for(int i=0 ; i < cis2.sizeCIList() ; i++) {
            cis1.addCI(cis2.getCI(i));
        }
        return cis1;
    }


}
```

更新の例

```
package com.hp.ucmdb.demo;

import com.hp.ucmdb.generated.params.update.AddCIsAndRelations;
import com.hp.ucmdb.generated.params.update.AddCIsAndRelationsResponse;
import com.hp.ucmdb.generated.params.update.UpdateCIsAndRelations;
import com.hp.ucmdb.generated.params.update.DeleteCIsAndRelations;
import com.hp.ucmdb.generated.services.UcmdbFaultException;
import com.hp.ucmdb.generated.types.*;
import com.hp.ucmdb.generated.types.update.CIsAndRelationsUpdates;
import com.hp.ucmdb.generated.types.update.ClientIDToCmdbID;

import java.rmi.RemoteException;

public class UpdateDemo extends Demo{
```

```
    public void getAddCIsAndRelationsDemo() {
        AddCIsAndRelations request = new AddCIsAndRelations();
        request.setCmdbContext(getContext());
        request.setUpdateExisting(true);
        CIsAndRelationsUpdates updates = new CIsAndRelationsUpdates();
        CIs cis = new CIs();
        CI ci = new CI();
        ID id = new ID();
        id.setBase("temp1");
        id.setTemp(true);
```

```
        ci.setID(id);
        ci.setType("host");
```

```
        CIProperties props = new CIProperties();
        StrProps strProps = new StrProps();
        StrProp strProp = new StrProp();
        strProp.setName("host_key");
        String value = "blabla";
        strProp.setValue(value);
```

```
        strProps.addStrProp(strProp);
        props.setStrProps(strProps);
        ci.setProps(props);
        cis.addCI(ci);
        updates.setCIsForUpdate(cis);
        request.setCIsAndRelationsUpdates(updates);
```

```
        try {
            AddCIsAndRelationsResponse response =
                getStub().addCIsAndRelations(request);
            for(int i = 0 ; i < response.sizeCreatedIDsMapList() ; i++) {
                ClientIDToCmdbID idsMap = response.getCreatedIDsMap(i);
                // 何らかの処理を実行
            }
        } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }

    }
```

```
    public void getUpdateCIsAndRelationsDemo() {
        UpdateCIsAndRelations request = new UpdateCIsAndRelations();
        request.setCmdbContext(getContext());
```

```
        CIsAndRelationsUpdates updates =
            new CIsAndRelationsUpdates();
        CIs cis = new CIs();
        CI ci = new CI();
        ID id = new ID();
```

```
        id.setBase("temp1");
        id.setTemp(true);
        ci.setID(id);
        ci.setType("host");
        CIProperties props = new CIProperties();
        StrProps strProps = new StrProps();
```

```
        StrProp hostKeyProp = new StrProp();
        hostKeyProp.setName("host_key");
        String hostKeyValue = "blabla";
        hostKeyProp.setValue(hostKeyValue);
        strProps.addStrProp(hostKeyProp);

        StrProp hostOSProp = new StrProp();
        hostOSProp.setName("host_os");
        String hostOSValue = "winXP";
        hostOSProp.setValue(hostOSValue);
        strProps.addStrProp(hostOSProp);

        StrProp hostDNSProp = new StrProp();
        hostDNSProp.setName("host_dnsname");
        String hostDNSValue = "dnsname";
        hostDNSProp.setValue(hostDNSValue);
        strProps.addStrProp(hostDNSProp);

        props.setStrProps(strProps);
        ci.setProps(props);
        cis.addCI(ci);
        updates.setCIsForUpdate(cis);
        request.setCIsAndRelationsUpdates(updates);

        try {
            getStub().updateCIsAndRelations(request);
        } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }

    }
```

```
    public void getDeleteCIsAndRelationsDemo() {
        DeleteCIsAndRelations request =
            new DeleteCIsAndRelations();
        request.setCmdbContext(getContext());
        CIsAndRelationsUpdates updates =
            new CIsAndRelationsUpdates();
        CIs cis = new CIs();
        CI ci = new CI();
        ID id = new ID();
        id.setBase("stam");
        id.setTemp(true);
        ci.setID(id);
        ci.setType("host");
```

```
        CIProperties props = new CIProperties();
        StrProps strProps = new StrProps();
        StrProp strProp1 = new StrProp();
        strProp1.setName("host_key");
        String value1 = "for_delete";
        strProp1.setValue(value1);
        strProps.addStrProp(strProp1);
        props.setStrProps(strProps);
        ci.setProps(props);
        cis.addCI(ci);
        updates.setCIsForUpdate(cis);
        request.setCIsAndRelationsUpdates(updates);
```

```
        try {
            getStub().deleteCIsAndRelations(request);
        } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }

    }


}
```

クラス・モデルの例

```
package com.hp.ucmdb.demo;

import com.hp.ucmdb.generated.params.classmodel.*;
import com.hp.ucmdb.generated.services.UcmdbFaultException;
import com.hp.ucmdb.generated.types.classmodel.UcmdbClassModelHierarchy;
import com.hp.ucmdb.generated.types.classmodel.UcmdbClass;

import java.rmi.RemoteException;

public class ClassmodelDemo extends Demo{
```

```
    public void getClassAncestorsDemo() {
        GetClassAncestors request =
            new GetClassAncestors();
        request.setCmdbContext(getContext());
        request.setClassName("className");
```

```
        try {
            GetClassAncestorsResponse response =
                getStub().getClassAncestors(request);
            UcmdbClassModelHierarchy hierarchy =
                response.getClassHierarchy();
        } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }

    }
```

```
    public void getAllClassesHierarchyDemo() {
        GetAllClassesHierarchy request =
            new GetAllClassesHierarchy();
        request.setCmdbContext(getContext());
        try {
            GetAllClassesHierarchyResponse response =
                getStub().getAllClassesHierarchy(request);
            UcmdbClassModelHierarchy hierarchy =
                response.getClassesHierarchy();
        } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }

    }
```

```
    public void getCmdbClassDefinitionDemo() {
        GetCmdbClassDefinition request =
            new GetCmdbClassDefinition();
        request.setCmdbContext(getContext());
        request.setClassName("className");
```

```
        try {
            GetCmdbClassDefinitionResponse response =
            getStub().getCmdbClassDefinition(request);
            UcmdbClass ucmdbClass = response.getUcmdbClass();
        } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }

    }



}
```

影響分析の例

```
package com.hp.ucmdb.demo;

import com.hp.ucmdb.generated.params.impact.*;
import com.hp.ucmdb.generated.services.UcmdbFaultException;
import com.hp.ucmdb.generated.types.*;
import com.hp.ucmdb.generated.types.impact.*;

import java.rmi.RemoteException;
```

```
/**
 * Date: Jul 17, 2007
 */
public class ImpactDemo extends Demo{

// 影響ルール名 : impactExample
// 影響のクエリ :
//              ネットワーク
//                |
//              ホスト
//                |
//                IP
// 影響の作用 : IP へのネットワークの影響 : 重大度 100%，カテゴリ : 変更
//
public void calculateImpactAndGetImpactPathDemo() {
    CalculateImpact request = new CalculateImpact();
    request.setCmdbContext(getContext());
    // ルート・コーズ ID の設定
    IDs ids = new IDs();
    ID id = new ID();
    id.setBase("rootCauseCmdbID");
    ids.addID(id);
```

```
request.setIDs(ids);
// 影響のカテゴリ設定
request.setImpactCategory("change");
// ルール名の設定
ImpactRuleNames impactRuleNames = new ImpactRuleNames();
ImpactRuleName impactRuleName = new ImpactRuleName();
impactRuleName.setBase("impactExample");
impactRuleNames.addImpactRuleName(impactRuleName);
request.setImpactRuleNames(impactRuleNames);
// 重大度の設定
request.setSeverity(100);
CalculateImpactResponse response =
    new CalculateImpactResponse();
```


```
request.setIDs(ids);
// 影響のカテゴリ設定
request.setImpactCategory("change");
// ルール名の設定
ImpactRuleNames impactRuleNames = new ImpactRuleNames();
ImpactRuleName impactRuleName = new ImpactRuleName();
impactRuleName.setBase("impactExample");
impactRuleNames.addImpactRuleName(impactRuleName);
request.setImpactRuleNames(impactRuleNames);
// 重大度の設定
request.setSeverity(100);
CalculateImpactResponse response =
    new CalculateImpactResponse();
```


```
try {
    response = getStub().calculateImpact(request);
} catch (RemoteException e) {
    // 例外の処理
```

```
    } catch (UcmdbFaultException e) {
        // 例外の処理
    }
    Identifier identifier= response.getIdentifier();
    Topology topology = response.getImpactTopology();
    Relation relation = topology.getRelations().getRelation(0);
    GetImpactPath request2 = new GetImpactPath();
    // cmdb コンテキストの設定
    request2.setCmdbContext(getContext());
    // 影響識別子の設定
    request2.setIdentifier(identifier);
    //set shallowRelation
    ShallowRelation shallowRelation = new ShallowRelation();
    shallowRelation.setID(relation.getID());
    shallowRelation.setEnd1ID(relation.getEnd1ID());
    shallowRelation.setEnd2ID(relation.getEnd2ID());
    shallowRelation.setType(relation.getType());
    request2.setRelation(shallowRelation);
```

```
    try {
        GetImpactPathResponse response2 =
            getStub().getImpactPath(request2);
        ImpactTopology impactTopology =
            response2.getImpactPathTopology();
    } catch (RemoteException e) {
        // catch ステートメントの内容を変更するには
        // ファイル | 設定 | File Templates を使用
        e.printStackTrace();
    } catch (UcmdbFaultException e) {
        // catch ステートメントの内容を変更するには
        // ファイル | 設定 | File Templates を使用
        e.printStackTrace();
    }
}
```

```
public void getImpactRulesByGroupName() {
    GetImpactRulesByGroupName request =
        new GetImpactRulesByGroupName();
    // cmdb コンテキストの設定
    request.setCmdbContext(getContext());
    // グループ名リストの設定
    request.addRuleGroupNameFilter("groupName1");
    request.addRuleGroupNameFilter("groupName2");
```

```
    try {
        GetImpactRulesByGroupNameResponse response =
            getStub().getImpactRulesByGroupName(request);
        ImpactRules impactRules = response.getImpactRules();
    } catch (RemoteException e) {
        // 例外の処理
    } catch (UcmdbFaultException e) {
        // 例外の処理
    }

}
```

```
public void getImpactRulesByNamePrefix() {
        GetImpactRulesByNamePrefix request =
            new GetImpactRulesByNamePrefix();
        // cmdb コンテキストの設定
        request.setCmdbContext(getContext());
        // プレフィックス・リストの設定
        request.addRuleNamePrefixFilter("prefix1");
```

```
        try {
            GetImpactRulesByNamePrefixResponse response =
                getStub().getImpactRulesByNamePrefix(request);
            ImpactRules impactRules = response.getImpactRules();
        } catch (RemoteException e) {
            // 例外の処理
        } catch (UcmdbFaultException e) {
            // 例外の処理
        }
}

}
```

資格情報の追加の例

```
import java.net.URL;

import org.apache.axis2.transport.http.HTTPConstants;
import org.apache.axis2.transport.http.HttpTransportProperties;

import com.hp.ucmdb.generated.params.discovery.*;
import com.hp.ucmdb.generated.services.DiscoveryService;
import com.hp.ucmdb.generated.services.DiscoveryServiceStub;
import com.hp.ucmdb.generated.types.BytesProp;
import com.hp.ucmdb.generated.types.BytesProps;
import com.hp.ucmdb.generated.types.CIProperties;
import com.hp.ucmdb.generated.types.CmdbContext;
import com.hp.ucmdb.generated.types.StrList;
import com.hp.ucmdb.generated.types.StrProp;
import com.hp.ucmdb.generated.types.StrProps;
```

```
public class test {
    static final String HOST_NAME = "hostname";
 static final int PORT = 8080;

    private static final String PROTOCOL = "http";
    private static final String FILE = "/axis2/services/DiscoveryService";

    private static final String PASSWORD = "admin";
    private static final String USERNAME = "admin";

    private static CmdbContext cmdbContext = new CmdbContext("ws tests");
```

```
    public static void main(String[] args) throws Exception {
        // スタブ・オブジェクトを取得
        DiscoveryService discoveryService = getDiscoveryService();

        // ジョブをアクティブ化
        discoveryService.activateJob(new ActivateJobRequest("Range IPs by ICMP",
cmdbContext));

        // ドメインとプローブ情報を取得
        getProbesInfo(discoveryService);

        // ntcmd プロトコルに対する資格情報エントリを追加
        addNTCMDCredentialsEntry();
    }
```

```
    public static void addNTCMDCredentialsEntry() throws Exception {
        DiscoveryService discoveryService = getDiscoveryService();

        // ドメイン名を取得
        StrList domains =
            discoveryService.getDomainsNames(new
GetDomainsNamesRequest(cmdbContext)).getDomainNames();
        if (domains.sizeStrValueList() == 0) {
            System.out.println("No domains were found, can't create credentials");
            return;
        }
        String domainName = domains.getStrValue(0);

        // 1 バイトのパラメータでプロパティを作成
        CIProperties newCredsProperties = new CIProperties();

        // パスワード・プロパティを追加 (bytes 型 )
        newCredsProperties.setBytesProps(new BytesProps());
        setPasswordProperty(newCredsProperties);

        // ユーザ・プロパティとドメイン・プロパティを追加 (string 型 )
        newCredsProperties.setStrProps(new StrProps());
        setStringProperties("protocol_username", "test user", newCredsProperties);
        setStringProperties("ntadminprotocol_ntdomain", "test doamin",
newCredsProperties);

        // 新しい資格情報エントリを追加
        discoveryService.addCredentialsEntry(new
AddCredentialsEntryRequest(domainName, "ntadminprotocol", newCredsProperties,
cmdbContext));

        System.out.println("new credentials craeted for domain: " + domainName + " in
ntcmd protocol");
    }
```

```
    private static void setPasswordProperty(CIProperties newCredsProperties) {
        BytesProp bProp = new BytesProp();
        bProp.setName("protocol_password");
        bProp.setValue(new byte[] {101,103,102,104});
        newCredsProperties.getBytesProps().addBytesProp(bProp);
    }
```

```
    private static void setStringProperties(String propertyName, String value,
CIProperties newCredsProperties) {
        StrProp strProp = new StrProp();
        strProp.setName(propertyName);
        strProp.setValue(value);
        newCredsProperties.getStrProps().addStrProp(strProp);
    }
```

```
    private static void getProbesInfo(DiscoveryService discoveryService) throws
Exception {
        GetDomainsNamesResponse result =
discoveryService.getDomainsNames(new GetDomainsNamesRequest(cmdbContext
));

        // すべてのドメインに対して実行
        if (result.getDomainNames().sizeStrValueList() > 0) {
            String domainName = result.getDomainNames().getStrValue(0);
            GetProbesNamesResponse probesResult =
                discoveryService.getProbesNames(new
GetProbesNamesRequest(domainName, cmdbContext));

            // すべてのプローブに対して実行
            for (int i=0; i<probesResult.getProbesNames().sizeStrValueList(); i++) {
                String probeName = probesResult.getProbesNames().getStrValue(i);

                // 接続したかどうか確認
                IsProbeConnectedResponce connectedRequest =
                    discoveryService.isProbeConnected(new
IsProbeConnectedRequest(domainName, probeName, cmdbContext));
                Boolean isConnected = connectedRequest.getIsConnected();

                // 何らかの処理を実行
                System.out.println("probe " + probeName + " isconnect=" +
isConnected);
            }
        }
    }
```

```
    private static DiscoveryService getDiscoveryService() throws Exception {
        DiscoveryService discoveryService = null;
        try {

            // サービスを作成
            URL url = new URL(PROTOCOL,HOST_NAME,PORT, FILE);
            DiscoveryServiceStub serviceStub = new
DiscoveryServiceStub(url.toString());

            // 認証情報
            HttpTransportProperties.Authenticator auth = new
HttpTransportProperties.Authenticator();
            auth.setUsername(USERNAME);
            auth.setPassword(PASSWORD);

serviceStub._getServiceClient().getOptions().setProperty(HTTPConstants.AUTHENTIC
ATE,auth);

            discoveryService = serviceStub;
        } catch (Exception e) {
            throw new Exception("cannot create a connection to service ", e);
        }

        return discoveryService;

    }
} // クラスの終わり
```

UCMDB の一般的なパラメータ

このセクションでは，サービスのメソッドの最も一般的なパラメータについて説明します。詳細については，スキーマに関するドキュメントを参照してください。

本項の内容

➤ 358 ページの「タイプ名」

➤ 358 ページの「構成アイテム（CI）」

➤ 358 ページの「関係」

CmdbContext

すべての UCMDB Web Service API サービスの呼び出しには，CmdbContext 引数が必要です。CmdbContext は，サービスを呼び出すアプリケーションを識別する callerApplication 文字列です。CmdbContext は，ログの記録とトラブルシューティングに使用します。

ID

すべての CI と関係には ID フィールドがあります。このフィールドは，大文字と小文字が区別される ID 文字列と，ID が一時かどうかを示す任意指定の temp フラグから構成されています。

キー属性

状況によっては，CI または Relation を識別する際に，キー属性を CMDB ID の代わりに使用できます。キー属性は，クラス定義で設定した ID_ATTRIBUTE を持つ属性です。

ユーザ・インタフェースの構成アイテム・タイプ属性のリストにおいて，キー属性の横にはキー・アイコンが表示されます。詳細については，『モデリング・ガイド』の「属性の追加 / 属性の編集ダイアログ・ボックス」を参照してください。API クライアント・アプリケーション内からのキー属性の識別の詳細については，292 ページの「getCmdbClassDefinition」を参照してください。

ID のタイプ

ID 要素には，実際の ID と一時 ID があり，空の場合もあります。

実際の ID は，CMDB によって割り当てられた文字列で，データベース内のエンティティを識別します。一時 ID は，現在の要求において一意である任意の文字列です。空の ID は，値が割り当てられていないことを意味します。

一時 ID はクライアントによって割り当てられ，多くの場合，クライアントによって保存されている CI の ID を表します。この ID は，必ずしも CMDB にすでに作成されているエンティティを表す必要はありません。一時 ID がクライアントによって渡されると，CI のキー・プロパティを使用して CMDB によって既存のデータ構成アイテムが識別できる場合には，実際の ID を使って識別されたのと同じように，その CI は状況に適したものとして使用されます。

CI の実際の ID は，CI のタイプとキー・プロパティの組み合わせに基づいて CMDB によって計算されます。Relation の実際の ID は，関係のタイプ，関係に属する 2 つの CI の ID，関係のキー・プロパティに基づいています。このため，キー属性値は CI または **関係**の作成時に設定する必要があります。CI の作成時にキー・プロパティ値が指定されていない場合には，次に示す 2 つの可能性が考えられます。

- CIT に RANDOM_GENERATED_ID 修飾子が含まれている場合，サーバによって一意の ID が生成されます。
- CIT に RANDOM_GENERATED_ID 修飾子が含まれていない場合，例外がスローされます。

詳細については，『モデリング・ガイド』の「CI タイプ・マネージャ」を参照してください。

CIProperties

CIProperties 要素はコレクションから構成され，それぞれにコレクション名によって示されるタイプのプロパティを指定する，一連の名前と値の要素が含まれます。これらは必須のコレクションではないため，CIProperties 要素は任意の組み合わせのコレクションを含むことができます。

CIProperties は CI 要素および Relation 要素によって使用されます。詳細については，358 ページの「構成アイテム（CI）」と 358 ページの「関係」を参照してください。

プロパティのコレクションを次に示します。

- dateProps : DateProp 要素のコレクション
- doubleProps : DoubleProp 要素のコレクション
- floatProps : FloatProp 要素のコレクション
- intListProps : intListProp 要素のコレクション
- intProps : IntProp 要素のコレクション
- strProps : StrProp 要素のコレクション

- strListProps : StrListProp 要素のコレクション
- longProps : LongProp 要素のコレクション
- bytesProps : BytesProp 要素のコレクション
- xmlProps : XmlProp 要素のコレクション

タイプ名

タイプ名は，構成アイテム・タイプまたは関係タイプのクラス名です。タイプ名は，クラスを参照するためにコード内で使用します。表示名と間違えないように注意してください。表示名はクラスが示されるユーザ・インタフェースに表示されますが，コード内では意味を持ちません。

構成アイテム（CI）

CI 要素は ID，type，および props コレクションから構成されます。

UCMDB 更新メソッドを使用して CI を更新する場合，ID 要素には，実際の CMDB ID またはクライアントによって割り当てられた一時 ID を含めることができます。一時 ID を使用する場合は，temp フラグを true に設定します。アイテムを削除する場合，ID は空でもかまいません。UCMDB クエリ・メソッドは，実際の ID を入力パラメータとして取り，実際の ID をクエリ結果に返します。

type は，CI タイプ・マネージャで定義した任意のタイプ名を指定できます。詳細については，『モデリング・ガイド』の「CI タイプ・マネージャ」を参照してください。

props 要素は，CIProperties コレクションです。詳細については，357 ページの「CIProperties」を参照してください。

関係

Relation は，2 つの構成アイテムをリンクするエンティティです。Relation 要素は，ID，type，リンク対象の 2 つのアイテムの識別子（end1ID と end2ID）および props コレクションから構成されます。

UCMDB 更新メソッドを使用して Relation を更新する場合，Relation の ID の値には，実際の CMDB ID または一時 ID を使用できます。アイテムを削除する場合，ID は空でもかまいません。UCMDB クエリ・メソッドは，実際の ID を入力パラメータとして取り，実際の ID をクエリ結果に返します。

関係タイプは, 関係のインスタンスが作成される UCMDB クラスの Type Name です。タイプは，CMDB に定義した関係タイプのいずれでもかまいません。クラスまたはタイプの詳細については，291 ページの「UCMDB クラス・モデルへの問い合わせ」を参照してください。

詳細については,『モデリング・ガイド』の「CI タイプ・マネージャ」を参照してください。

関係の 2 つの終了 ID は，現在の関係の ID を作成するのに使用されるため空の ID を指定することはできません。ただし，これらの終了 ID には，クライアントによって割り当てられた一時 ID を使用することができます。

props 要素は，CIProperties コレクションです。詳細については，357 ページの「CIProperties」を参照してください。

UCMDB 出力パラメータ

本項では，サービス・メソッドの最も一般的な出力パラメータについて説明します。詳細については，スキーマに関するドキュメントを参照してください。

本項の内容

CI

CIs は，CI 要素のコレクションです。

ShallowRelation

ShallowRelation は, 2 つの構成アイテムをリンクするエンティティで, ID, type, およびリンク対象の 2 つのアイテムの識別子（end1ID と end2ID）から構成されます。関係タイプは，関係のインスタンスが作成される CMDB クラスの Type Name です。タイプは, CMDB に定義した関係タイプのいずれでもかまいません。

Topology

Topology は，CI 要素と関係のグラフです。Topology は，CIs コレクション，および 1 つ以上の Relations 要素を含む Relation コレクションから構成されています。

CINode

CINode は，label を持つ CIs コレクションから構成されています。CINode の label は，クエリで使用する TQL のノードで定義したラベルです。

RelationNode

RelationNode は，label を持つ Relation コレクションのセットです。RelationNode の label は，クエリで使用する TQL のノードで定義したラベルです。

TopologyMap

TopologyMap は，TQL クエリに一致するクエリ計算を出力したものです。TopologyMap の labels は，クエリで使用する TQL で定義したノード・ラベルです。

TopologyMap のデータは，次の形式で返されます。

- CINodes :1 つ以上の CINode です（360 ページの「CINode」を参照してください）。
- relationNodes :1 つ以上の RelationNode です（360 ページの「RelationNode」を参照してください）。

これら 2 つの構造内の label によって, 構成アイテムと関係のリストが配列されます。

ChunkInfo

クエリによって大量のデータが返されると，サーバはデータをチャンクというセグメントに分割して保存します。チャンクに分割したデータを取得するためにクライアントが使用する情報は，クエリによって返される ChunkInfo 構造に配置されます。ChunkInfo は，取得する必要がある numberOfChunks と chunksKey から構成されます。chunksKey は，この特定のクエリ呼び出しに対するサーバ上のデータの一意の識別子です。

詳細については，285 ページの「サイズの大きい応答の処理」を参照してください。

9

HP Universal CMDB API

本章の内容

概念

タスク

参照先

概念

規則

本章では，次の表記規則を使用します。

- **UCMDB** は，Universal Configuration Management database 自体を指します。**HP Universal CMDB** は，アプリケーションを意味します。
- UCMDB の要素およびメソッド引数は，インタフェース内で指定される場合に記述します。

HP Universal CMDB API の使用

本章は，オンラインのドキュメント・ライブラリで使用できる API Javadoc と併せて使用してください。

HP Universal CMDB API は，アプリケーションと Universal CMDB（CMDB）を統合するために使用します。この API により，次を実施するメソッドが提供されます。

- CMDB での CI と関係の追加，削除，および更新
- クラス・モデルに関する情報の取得
- what-if シナリオの実行
- 構成アイテムおよび関係に関する情報の取得

構成アイテムと関係に関する情報を取得するメソッドでは，一般的にトポロジ・クエリ言語（TQL）を使用します。詳細については，『モデリング・ガイド』の「トポロジ クエリ言語」を参照してください。

HP Universal CMDB API のユーザは，次のことを十分理解している必要があります。

- Java プログラミング言語
- HP Universal CMDB

本項の内容

API の使用

API を使用すると，多くのビジネス要件を満たすことができます。たとえば，サードパーティ製のシステムは，利用できる構成アイテム（CI）に関する情報をクラス・モデルに問い合わせることができます。詳しい使用例については，371 ページの「使用例」を参照してください。

権限

管理者により，API に接続するためのログイン資格情報が提供されます。API クライアントには，CMDB で定義されているインテグレーション・ユーザのユーザ名とパスワードが必要です。これらのユーザは CMDB の実際のユーザを表すものではなく，CMDB に接続するアプリケーションを表します。

詳細については，368 ページの「インテグレーション・ユーザの作成」を参照してください。

アプリケーションの一般的な構造

静的ファクトリ「UcmdbServiceFactory」のみが存在します。このファクトリは，アプリケーションのエントリ・ポイントです。UcmdbServiceFactory は getServiceProvider メソッドを公開します。これらのメソッドは **UcmdbServiceProvider** インタフェースのインスタンスを返します。

クライアントは，インタフェース・メソッドを使用してほかのオブジェクトを作成します。たとえば，新しいクエリ定義を作成するには，クライアントは次の手順で行います。

1 メイン CMDB サービス・オブジェクトからクエリ・サービスを取得します。

2 サービス・オブジェクトからクエリ・ファクトリ・オブジェクトを取得します。

3 ファクトリから新しいクエリ定義を取得します。

```
UcmdbServiceProvider provider =
  UcmdbServiceFactory.getServiceProvider(HOST_NAME, PORT);
UcmdbService = provider.connect(provider.createCredentials(USERNAME,
  PASSWORD), provider.createClientContext("Test"));
TopologyQueryService queryService = ucmdbService.getTopologyQueryService();
TopologyQueryFactory factory = queryService.getFactory();
QueryDefinition queryDefinition = factory.createQueryDefinition("Test Query");
queryDefinition.addNode("Node").ofType("host");
Topology topology = queryService.executeQuery(queryDefinition);
System.out.println("There are " + topology.getAllCIs().size() + " hosts in uCMDB");
```

UcmdbService から使用できるサービスは次のとおりです。

サービス・メソッド	用途
getClassModelService	CI と関係のタイプに関する情報
getDDMConfigurationService	ディスカバリおよび依存関係管理システムの設定
getDDMManagementService	ディスカバリおよび依存関係管理システムの進行状況，結果，エラーを分析して表示
getImpactAnalysisService	影響分析シナリオ（**相関**とも呼ばれます）を実行
getQueryManagementService	クエリへのアクセスの管理（既存アイテムの保存，削除，リスト表示）。クエリの検証やクエリ依存関係のディスカバリも行います。
getResourceBundleManagement Service	リソースのタグ付け（バンドル化）サービス。新しいタグの明示的な生成や，すべてのタグ付きリソースからのタグ削除を行います。
getSoftwareSignatureService	ディスカバリおよび依存関係管理システムで検出するソフトウェア・アイテムの定義
getTopologyQueryService	IT ユニバースに関する情報を取得
getTopologyUpdateService	IT ユニバース内の情報を変更

サービス・メソッド	用途
getViewService	実行サービス（定義や保存済みアイテムの実行）と管理サービス（既存アイテムの保存，削除，リスト表示）の表示。ビューの検証や依存関係のディスカバリも行います。
getViewArchiveService	結果アーカイブ・サービスの表示。現在のビュー結果の保存や以前保存した結果の取得を行います。

クライアントは HTTP を通じてサーバと通信します。

タスク

クラスパスへの API Jar ファイルの配置

この API セットを使用するには，**ucmdb-api.jar** ファイルが必要です。このファイルにアクセスするには，UCMDB バージョン 9.0 ダウンロード・パッケージを展開します。jar ファイルは，バージョン 9.0 の **setup.exe** ファイルとともに，UCMDB_Java_API サブフォルダ内にあります。jar ファイルは，バージョン 9.0 のインストール・プロセスではインストールされません。

アプリケーションをコンパイルまたは実行する前に，jar ファイルをクラスパスに置いてください。

インテグレーション・ユーザの作成

ほかの製品と UCMDB のインテグレーションでは，専用のユーザを作成できます。このユーザは，UCMDB クライアント SDK を使用する製品を有効にして，サーバ SDK での認証と API の実行を行います。この API セットを使って書かれたアプリケーションは，インテグレーション・ユーザの資格情報を使ってログオンする必要があります。

注意：この API セットから CMDB に接続できるのはインテグレーション・ユーザのみです。ほかのタイプのユーザが接続しようとすると，LDAP 検証を使っていても，エラーが発生することがあります。

インテグレーション・ユーザを作成するには，次の手順で行います。

1. Web ブラウザを起動して，次のサーバ・アドレスを入力します。

   http://localhost:8080/jmx-console :

   ユーザ名およびパスワードを使ってログインする必要がある場合があります（標準設定では sysadmin/sysadmin です）。

2. UCMDB の下の **service=UCMDB Security Services** をクリックして，JMX MBEAN ページを開きます。

3 **CreateIntegrationUser** 操作を見つけます。このメソッドには，次のパラメータがあります。

- **customerId**: カスタマ ID です。
- **ユーザ名** : インテグレーション・ユーザの名前です。
- **password**: インテグレーション・ユーザのパスワードです。
- **dataStoreOrigin**: このインテグレーション・ユーザを使用する製品の名前です。

  インテグレーション・ユーザの管理には，次のような便利な操作が用意されています。

- **DeleteIntegrationUser**: 任意のインテグレーション・ユーザを削除します。
- **ExportIntegrationUser**: インテグレーション・ユーザを，サーバ・マシン上の任意のパスにある XML ファイルにエクスポートします。
- **getIntegrationUser**: インテグレーション・ユーザの情報を表示します。
- **changeIntegrationUserPassword**: インテグレーション・ユーザのパスワードを変更します。
- **canUserAuthenticate:isIntegrationUser** が **true** のとき : 任意の資格情報でインテグレーション・ユーザを認証できるようにします。

4 ［**Invoke**］をクリックします。

［**Back to MBean View**］をクリックしてさらにユーザを作成するか，JMX コンソールを閉じます。

5 管理者として UCMDB にログオンします。

6 ［**管理**］タブで，［**パッケージ マネージャ**］を実行します。

7 ［**新規作成**］アイコンをクリックします。

8 新しいパッケージ名を入力して，［**次へ**］をクリックします。

9 ［**管理**］の下の［リソースの選択］タブで，［**インテグレーション ユーザ**］をクリックします。

10 JMX コンソールを使用して作成したユーザ（複数可）を選択します。

11 ［**次へ**］をクリックし，次に［**終了**］をクリックします。新しいパッケージが，パッケージ・マネージャの［パッケージ名］の一覧に表示されます。

12 API アプリケーションを実行するユーザにパッケージをデプロイします。

詳細については，『HP UCMDB 管理ガイド』の「パッケージのデプロイ」を参照してください。

注 :

インテグレーション・ユーザはカスタマごとに作成します。複数のカスタマで使用できる，より強い権限のインテグレーション・ユーザを作成するには，**isSuperIntegrationUser** フラグを **true** にセットして，**systemUser** を使用してください。**systemUser** のメソッド（**createSystemUser**，**removeSystemUser**，**showAllSystemUsers**，**changeSystemUserPassword**，**canSuperIntegrationUserAuthenticate** など）を使用します。

次の 2 つのシステム・ユーザが用意されています。インストール後，**changeSystemUserPassword** メソッドを使って両方のパスワードを変更することをお勧めします。

- **sysadmin/sysadmin**
- **UISysadmin/UISysadmin**（このユーザはスーパー・インテグレーション・ユーザ **SuperIntegrationUser** でもあります）

changeSystemUserPassword を使って UISysadmin パスワードを変更した場合，次のメソッドを実行する必要があります。まず，JMX コンソールで **UCMDB-UI:name=UCMDB Integration** サービスを選択します。その後，インテグレーション・ユーザのユーザ名と新しいパスワードを使って，**setCMDBSuperIntegrationUser** を実行します。

参照先

HP Universal CMDB API 参考情報

用意されている API の全ドキュメントについては，第 7 章：「API の概要」を参照してください。

使用例

次の使用例は 2 つのシステムを想定しています。

- HP Universal CMDB サーバ
- 構成アイテムのリポジトリを含むサードパーティ製のシステム

本項の内容

- 371 ページの「UCMDB のポピュレート」
- 372 ページの「UCMDB への問い合わせ」
- 372 ページの「クラス・モデルへの問い合わせ」
- 372 ページの「変更の影響の分析」

UCMDB のポピュレート

使用例：

- サードパーティ製のアセット管理は，アセット管理でのみ使用できる情報を使って UCMDB を更新します。
- 多くのサードパーティ製システムは，UCMDB にデータをポピュレートして，変更内容を追跡し影響分析を実行できる中心的な CMDB を作成します。
- サードパーティ製のシステムは，サードパーティのビジネス・ロジックに従って構成アイテムと関係を作成し，UCMDB クエリ機能を活用します。

UCMDB への問い合わせ

使用例：

- サードパーティ製のシステムは，SAP TQL の結果を取得することによって，SAP システムを表す構成アイテムと関係を取得します。
- サードパーティ製のシステムは，過去 5 時間以内に追加または変更された Oracle サーバのリストを取得します。
- サードパーティ製のシステムは，ホスト名に部分文字列 lab が含まれるサーバのリストを取得します。
- サードパーティ製のシステムは，隣接項目を取得することによって，特定の CI に関係する要素を検出します。

クラス・モデルへの問い合わせ

使用例：

- サードパーティ製のシステムでは，ユーザは CMDB から取得するデータのセットを指定できます。ユーザ・インタフェースはクラス・モデル上に構築し，ユーザに利用可能なプロパティを表示して，必要なデータを求めることができます。ユーザは，取得する情報を選択できます。
- サードパーティ製のシステムは，ユーザが UCMDB ユーザ・インタフェースにアクセスできないときに，クラス・モデルを探索します。

変更の影響の分析

使用例：

サードパーティ製のシステムは，指定したホストに対する変更の影響を受けるビジネス・サービスのリストを出力します。

例

本項の内容

- 373 ページの「エントリ・ポイントの例」
- 373 ページの「クエリの例」

➤ 375 ページの「トポロジ・クエリの例」
➤ 376 ページの「トポロジ更新の例」
➤ 376 ページの「影響分析の例」

エントリ・ポイントの例

```
final String HOST_NAME = "localhost";
final int PORT = 8080;
UcmdbServiceProvider provider =
    UcmdbServiceFactory.getServiceProvider(HOST_NAME, PORT);
final String USERNAME = "integration_user";
final String PASSWORD = "integration_password";
Credentials credentials =
    provider.createCredentials(USERNAME, PASSWORD),
ClientContext clientContext = provider.createClientContext("Example");
UcmdbService ucmdbService = provider.connect(credentials, clientContext);
```

クエリの例

次の例では，単一クラス定義の取得や，すべての CIT 定義とその属性のリストの取得を行います。

クラス定義の取得

```
ClassModelService classModelService
    = ucmdbService.getClassModelService();
String typeName = "disk";
ClassDefinition def =
    classModelService.getClassDefinition(typeName);
System.out.println("Type " + typeName + " is derived from type "
    + def.getParentClassName());
System.out.println("Has " + def.getChildClasses().size() +
    " derived types");
System.out.println("Defined and inherited attributes:");
for (Attribute attr : def.getAllAttributes().values()) {
    System.out.println("Attribute " + attr.getName() +
    " of type " + attr.getType());
}
```

CIT 定義と属性のリストを取得

この例では，1 つの CIT の属性のクエリを実行し，その名前とタイプを印刷します。

```
ClassModelService classModelService =
      ucmdbService.getClassModelService();
for (ClassDefinition def : classModelService.getAllClasses()) {
     System.out.println("Type " + def.getName() +
          " (" + def.getDisplayName() + ") is derived from type "
          + def.getParentClassName());
     System.out.println
          ("Has " + def.getChildClasses().size() + " derived types");
     System.out.println
          ("Defined and inherited attributes:");
     for (Attribute attr : def.getAllAttributes().values()) {
          System.out.println
           ("Attribute " + attr.getName() +
           " of type " + attr.getType());
     }
}
```

トポロジ・クエリの例

```
TopologyQueryService queryService =
    ucmdbService.getTopologyQueryService();
TopologyQueryFactory queryFactory =
    queryService.getFactory();
QueryDefinition queryDefinition =
    queryFactory.createQueryDefinition
        ("Get hosts with more than one network interface");
String hostNodeName = "Host";
QueryNode hostNode =

queryDefinition.addNode(hostNodeName).ofType("host").queryProperty("display_label"
);
QueryNode ipNode =
    queryDefinition.addNode("IP").ofType("ip").queryProperty("ip_address");
hostNode.linkedTo(ipNode).withLinkOfType("contained").atLeast(2);
Topology topology = queryService.executeQuery(queryDefinition);
Collection<TopologyCI> hosts = topology.getCIsByName(hostNodeName);
for (TopologyCI host : hosts) {
    System.out.println("Host " + host.getPropertyValue("display_label"));
    for (TopologyRelation relation : host.getOutgoingRelations()) {
         System.out.println
        (" has IP " + relation.getEnd2CI().getPropertyValue("ip_address"));
    }
}
```

トポロジ更新の例

```
TopologyUpdateService topologyUpdateService =
    ucmdbService.getTopologyUpdateService();
TopologyUpdateFactory topologyUpdateFactory =
    topologyUpdateService.getFactory();
TopologyModificationData topologyModificationData =
    topologyUpdateFactory.createTopologyModificationData();
CI host = topologyModificationData.addCI("host");
host.setPropertyValue("host_key", "test1");
CI ip = topologyModificationData.addCI("ip");
ip.setPropertyValue("ip_address", "127.0.0.10");
ip.setPropertyValue("ip_domain", "DefaultDomain");
topologyModificationData.addRelation("contained", host, ip);
topologyUpdateService.create
    (topologyModificationData, CreateMode.IGNORE_EXISTING);
```

影響分析の例

```
ImpactAnalysisService impactAnalysisService =
    ucmdbService.getImpactAnalysisService();
ImpactAnalysisFactory impactFactory =
    impactAnalysisService.getFactory();
ImpactAnalysisDefinition definition =
    impactFactory.createImpactAnalysisDefinition();
definition.addTriggerCI(disk).withSeverity
    (impactFactory.getSeverityByName("Warning(2)"));
definition.useAllRules();
ImpactAnalysisResult impactResult =
    impactAnalysisService.analyze(definition);
AffectedTopology affectedCIs =
    impactResult.getAffectedCIs();
for (AffectedCI affectedCI : affectedCIs.getAllCIs()) {
     System.out.println("Affected " +
     affectedCI.getType() + " " + affectedCI.getId() +
     " - severity " + affectedCI.getSeverity());
}
```

索引

L

M

N

O

P

R

S

T

U

W

X

あ

い

え

お

く

こ

し

す

た

て

と

な

は

ひ

ふ

ま

め

も

り

れ

ろ