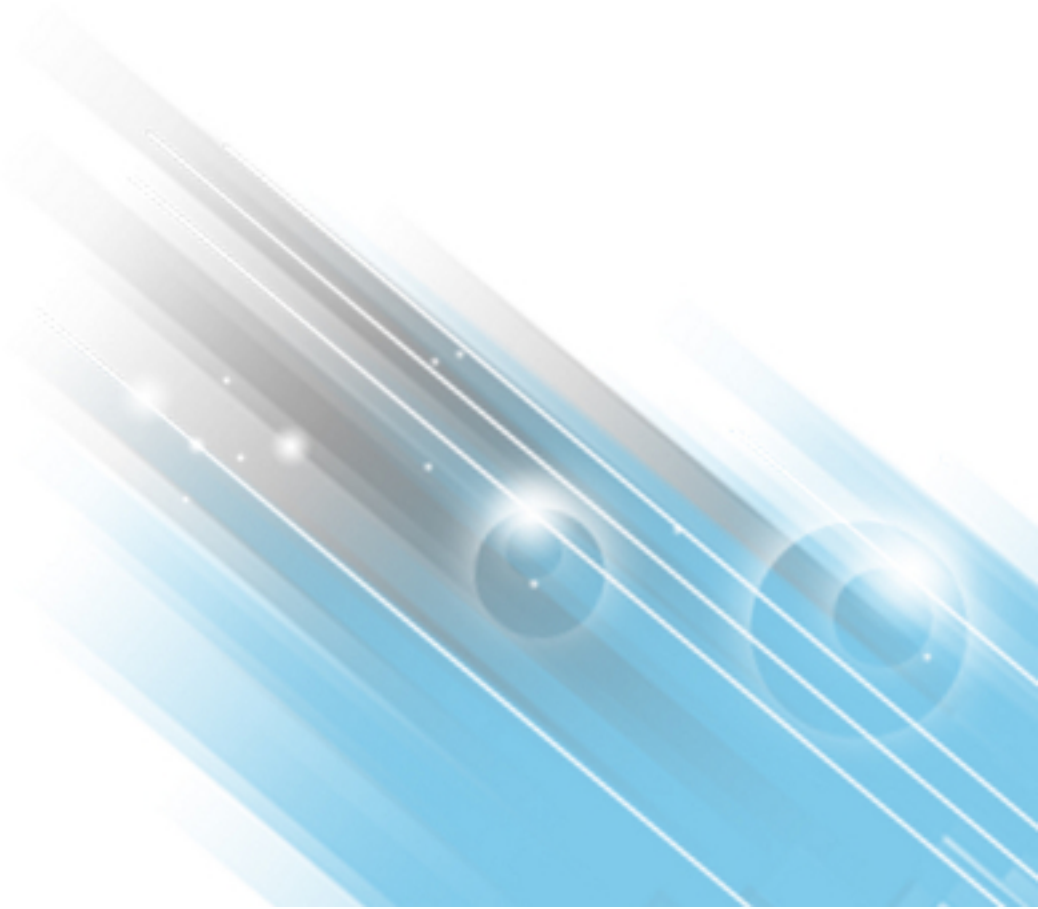
HP Universal CMDB

ソフトウェア・バージョン : 10.20

開発者向け参考情報ガイド



ドキュメント・リリース日 : 2015 年 1 月
ソフトウェア・リリース日 : 2015 年 1 月

ご注意

保証

HP 製品，またはサービスの保証は，当該製品，およびサービスに付随する明示的な保証文によってのみ規定されるものとします。ここでの記載で追加保証を意図するものは一切ありません。ここに含まれる技術的，編集上の誤り，または欠如について，HP はいかなる責任も負いません。

ここに記載する情報は，予告なしに変更されることがあります。

権利の制限

機密性のあるコンピュータ・ソフトウェアです。これらを所有，使用，または複製するには，HP からの有効な使用許諾が必要です。商用コンピュータ・ソフトウェア，コンピュータ・ソフトウェアに関する文書類，および商用アイテムの技術データは，FAR12.211 および 12.212 の規定に従い，ベンダーの標準商用ライセンスに基づいて米国政府に使用許諾が付与されます。

著作権について

商標

Adobe™ は，Adobe Systems Incorporated の商標です。

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

UNIX ® は，The Open Group の登録商標です。

本製品には、Copyright © 1995-2002 Jean-loup Gailly and Mark Adler である「zlib」汎用圧縮ライブラリのインターフェースが含まれています。

文書の更新

このマニュアルの表紙には，以下の識別番号が記載されています。

- ソフトウェアのバージョン番号は，ソフトウェアのバージョンを示します。
- ドキュメント・リリース日は，ドキュメントが更新されるたびに変更されます。
- ソフトウェア・リリース日は，このバージョンのソフトウェアのリリース期日を表します。

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，次の URL にアクセスしてください :**https://softwaresupport.hp.com**

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 URL にアクセスしてください。**https://hpp12.passport.hp.com/hppcf/createuser.do**

または，HP ソフトウェア・サポート・ページの上部にある［**登録**］リンクをクリックします。

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

次の HP ソフトウェアのサポート・オンラインの Web サイトをご覧ください。
https://softwaresupport.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 を登録するには，次の Web サイトにアクセスしてください。

https://hpp12.passport.hp.com/hppcf/createuser.do

アクセスレベルに関する詳細は，以下の Web サイトにアクセスしてください。

https://softwaresupport.hp.com/web/softwaresupport/access-levels

HP Software Solutions Now は，HPSW のソリューションと統合に関するポータル Web サイトです。このサイトでは，お客様のビジネスニーズを満たす HP 製品ソリューションを検索したり，HP 製品間の統合に関する詳細なリストや ITIL プロセスのリストを閲覧することができます。この Web サイトの URL は **http://h20230.www2.hp.com/sc/solutions/index.jsp** です

オンライン・ヘルプのこの PDF バージョンについて

このドキュメントは，オンライン・ヘルプの PDF バージョンです。この PDF ファイルは、ヘルプ情報の複数トピックを簡単に印刷したり，オンライン・ヘルプを PDF 形式で読めるよう提供されるものです。その内容は本来 Web ブラウザでオンライン・ヘルプとして表示するために作成されたものであるため、一部のトピックが適切な形式になっていない可能性があります。インタラクティブ形式のトピックの一部に，この PDF バージョンでは表示されないものがあります。それらのトピックは，オンライン・ヘルプ内から正常に印刷可能です。

目次

第I部: ディスカバリおよび統合アダプタの作成

第1章: アダプタ開発と記述

本章の内容

アダプタの開発と記述の概要

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。

本章の内容

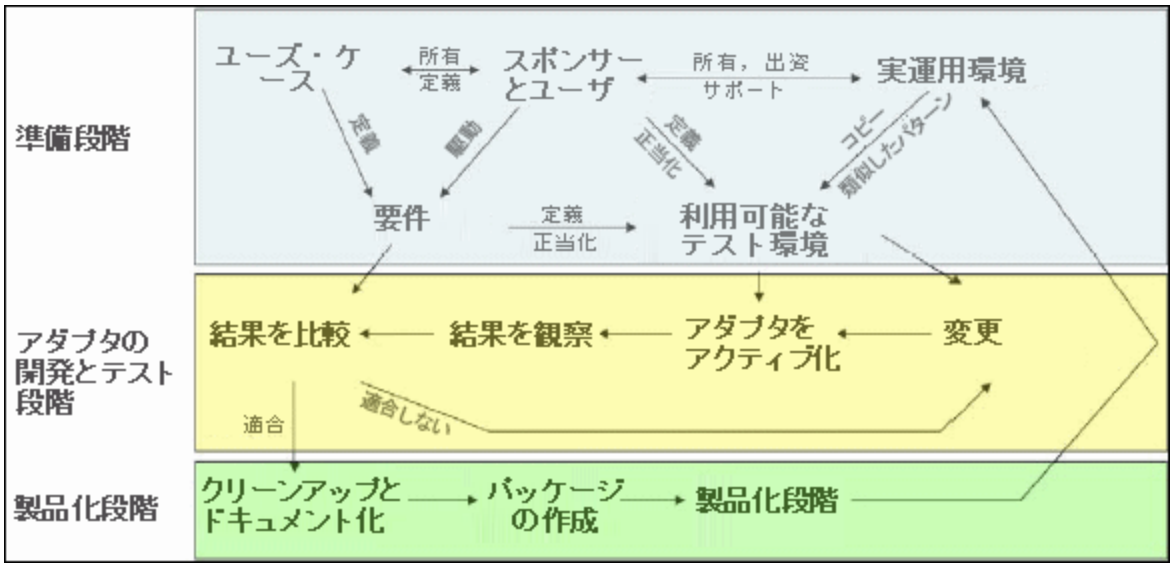
- HP Universal CMDB に関する実用的な知識と，システムの要素に関する基本的な知識があることを前提としています。本章は，ユーザの学習を支援することが目的であり，完全なガイドではありません。
- HP Universal CMDB の新しいディスカバリ・コンテンツの計画，調査，および実装段階を対象としています。また，考慮する必要があるガイドラインと注意事項も示します。
- データ・フロー管理フレームワークの主な API について説明します。利用可能な API の完全なドキュメントについては，*HP Universal CMDB Data Flow Management API Reference*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

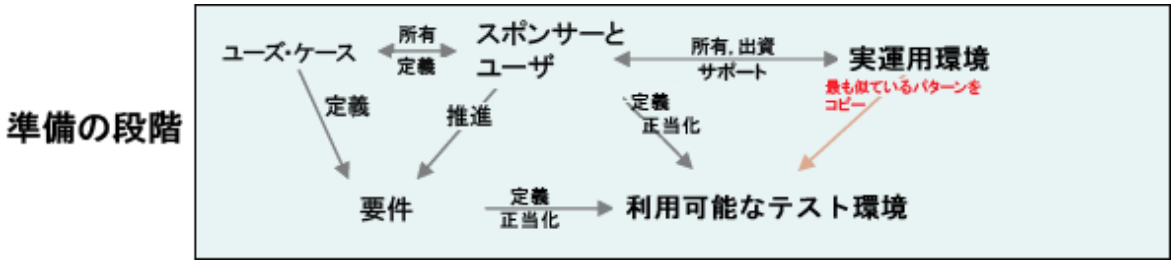


アダプタ開発の各段階は，直前の段階に基づいています。

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 Universal CMDB 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。

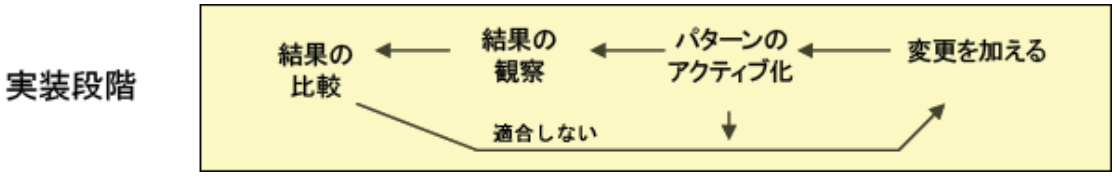
次の各項では，最も重要な手順とベスト・プラクティスを示しながら，各段階についてさらに詳しく説明します。

調査と準備の段階



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. アダプタがデータを収集するために使用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
   - これで似ているアダプタがわかる
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. パッケージの内容を作業領域に展開し，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，大いに役立ちます。

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 Universal CMDB 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 Universal CMDB 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・フローの管理と統合

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。統合アダプタは，ディスカバリより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，統合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。
- レポートの適切な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ます。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，「アダプタの開発と記述の概要」(12ページ)を参照してください。

本項の内容

既存のアダプタの変更

定義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，利用可能であるが定義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。
- 定義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，定義済みの 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，**［データ フロー管理］>［Data Flow Probe 設定］>［ドメインとプローブ］**表示枠で表示できます。各プロトコルの詳細については，『HP UCMDB Discovery and Integrations Content Guide』に記載のサポートされているプロトコルの説明セクション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か。データを履歴で追跡する必要があるか。ソースを信頼できないか。

  これらの質問に「はい」と答える場合は，**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か。データ変更の正確さが重要であるか。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か。

  これらの質問に「はい」と答える場合は，**フェデレーション**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か。

  これらの質問に「はい」と答える場合は，**Dat データ・プッシュ**が必要です。
- いずれかの CI の ID の長さが 60 文字を超えているか

  この質問に「はい」と答える場合は，すべての関連する CI について ID の長さを**縮小**して，それらの ID が 60 文字の最大長を超えないようにします。

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。

統合アダプタの作成には，次の異なる 5 つのオプションを使用できます。

1. 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，「Jython アダプタの開発」(37ページ)を参照してください。
2. 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・プッシュに利用可能（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。
   - 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，「Java アダプタの開発」(192ページ)を参照してください。
3. 汎用 DB アダプタ:
   - Federation SDK Framework を使用する Java アダプタをベースにした抽象アダプタ。
   - 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。
   - 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，「汎用データベース・アダプタの開発」(119ページ)を参照してください。
4. 汎用プッシュ・アダプタ:

- Java アダプタおよび Jython アダプタをベースにした抽象アダプタ（Federation SDK Framework）。
- 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。
- 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，「プッシュ・アダプタの開発」(227ページ)を参照してください。

5. 拡張汎用プッシュ・アダプタ:
   - 汎用プッシュ・アダプタの上記のすべての機能
   - ルート要素ベースのアダプタ
   - UCMDB ツリー・データ構造をターゲット・ツリー・データ構造にマップする

   詳細については，「汎用アダプタを使用したデータ・プッシュのアーカイブ」(275ページ)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	汎用 DB アダプタ	汎用プッシュ・アダプタ	拡張汎用プッシュ・アダプタ
ポピュレーション	✓	✓	✓	×	×
連携	×	✓	✓	×	×
データ・プッシュ	×	✓	×	✓	✓

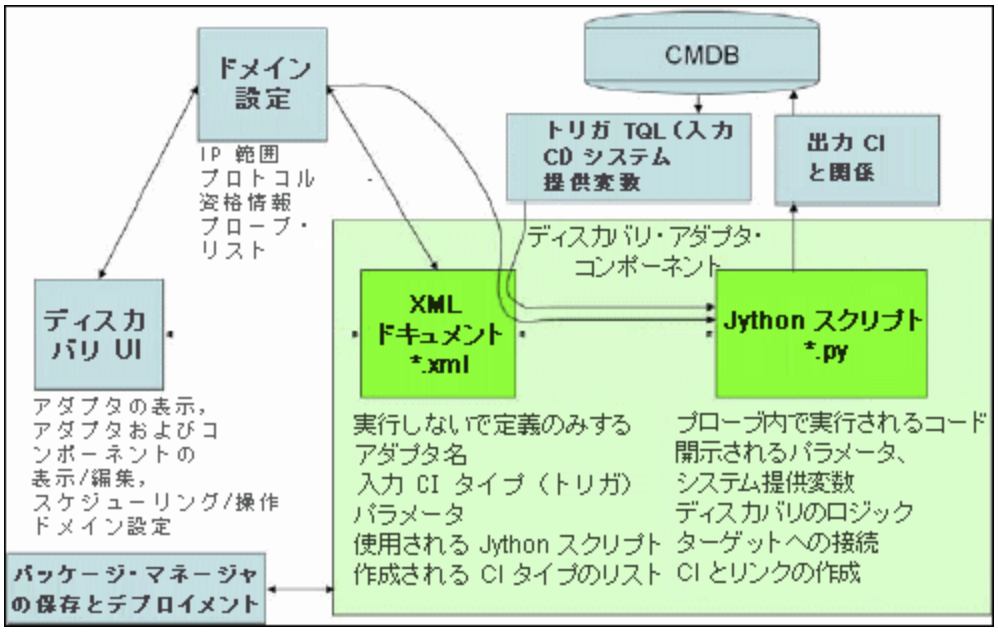
ディスカバリ・コンテンツの開発

本項の内容

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，［Universal Discovery］，［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

ディスカバリ全体を 1 つのアダプタで定義することができます。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。

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. **アダプタを作成します。**

   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，「手順 1 : アダプタの作成」(28ページ)を参照してください。

2. **ディスカバリ・ジョブを作成します。**

   スケジュール情報とトリガ・クエリを含むジョブを設定します。詳細については，「手順 2 : アダプタへのジョブの割り当て」(34ページ)を参照してください。

3. **ディスカバリ・コードを編集します。**

   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，「手順 3 :Jython コードの作成」(35ページ)を参照してください。

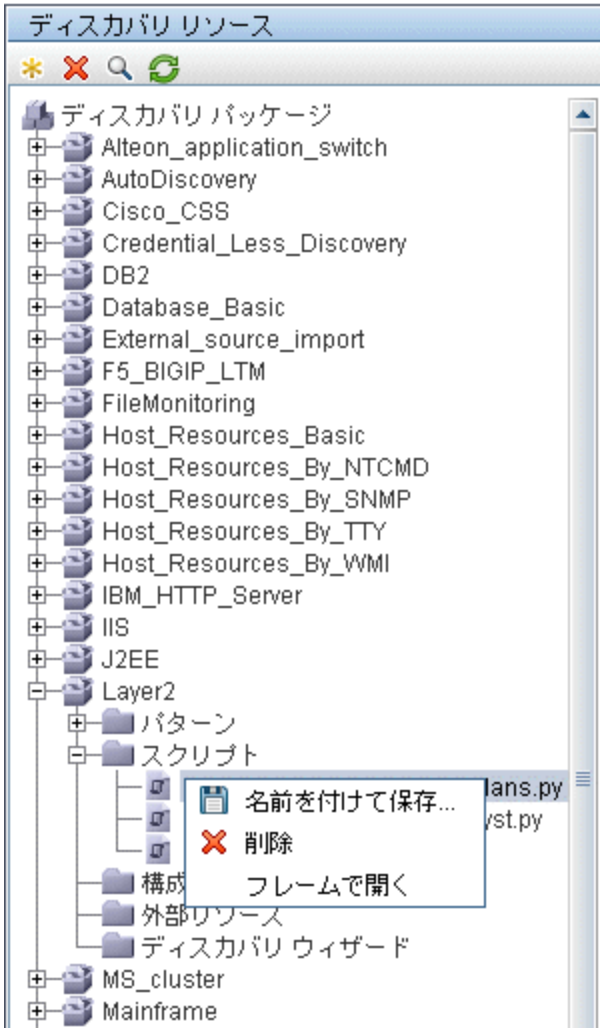
新しいアダプタを記述するには，前述の各コンポーネントを作成します。作成した各コンポーネントは，前の手順のコンポーネントに自動的にバインドされます。たとえば，ジョブを作成して関連するアダプタを選択すると，そのアダプタはジョブにバインドされます。

アダプタ・コード

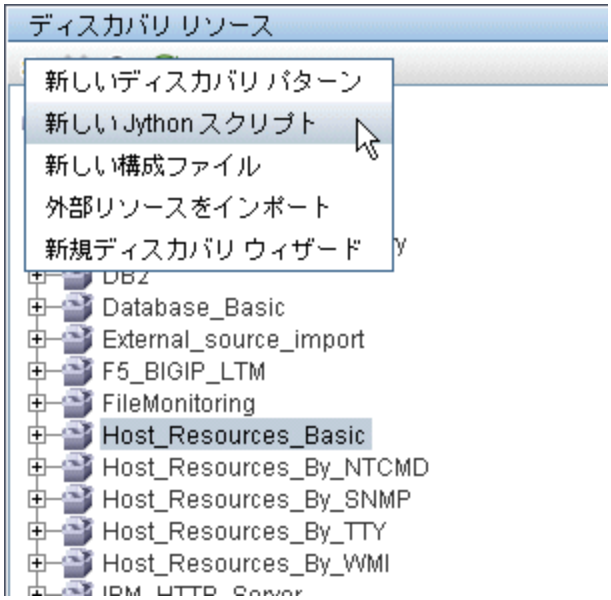
リモート・システムに接続し，そのデータを照会し，それを CMDB データとしてマップする処理の実際の実装は，Jython コードによって実行されます。コードには，たとえば，データベースに接続してそこからデータを抽出するためのロジックが含まれています。この場合，コードは JDBC URL，ユーザ名，パスワード，ポートなどを受け取ることを期待しています。これらのパラメータは，TQL クエリに応答するデータベースの各インスタンスに固有のものです。これらの変数はアダプタ（トリガ CI データ）で定義され，ジョブを実行すると，これらの詳細がコードに渡されて実行に使用されます。

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。本項では，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。

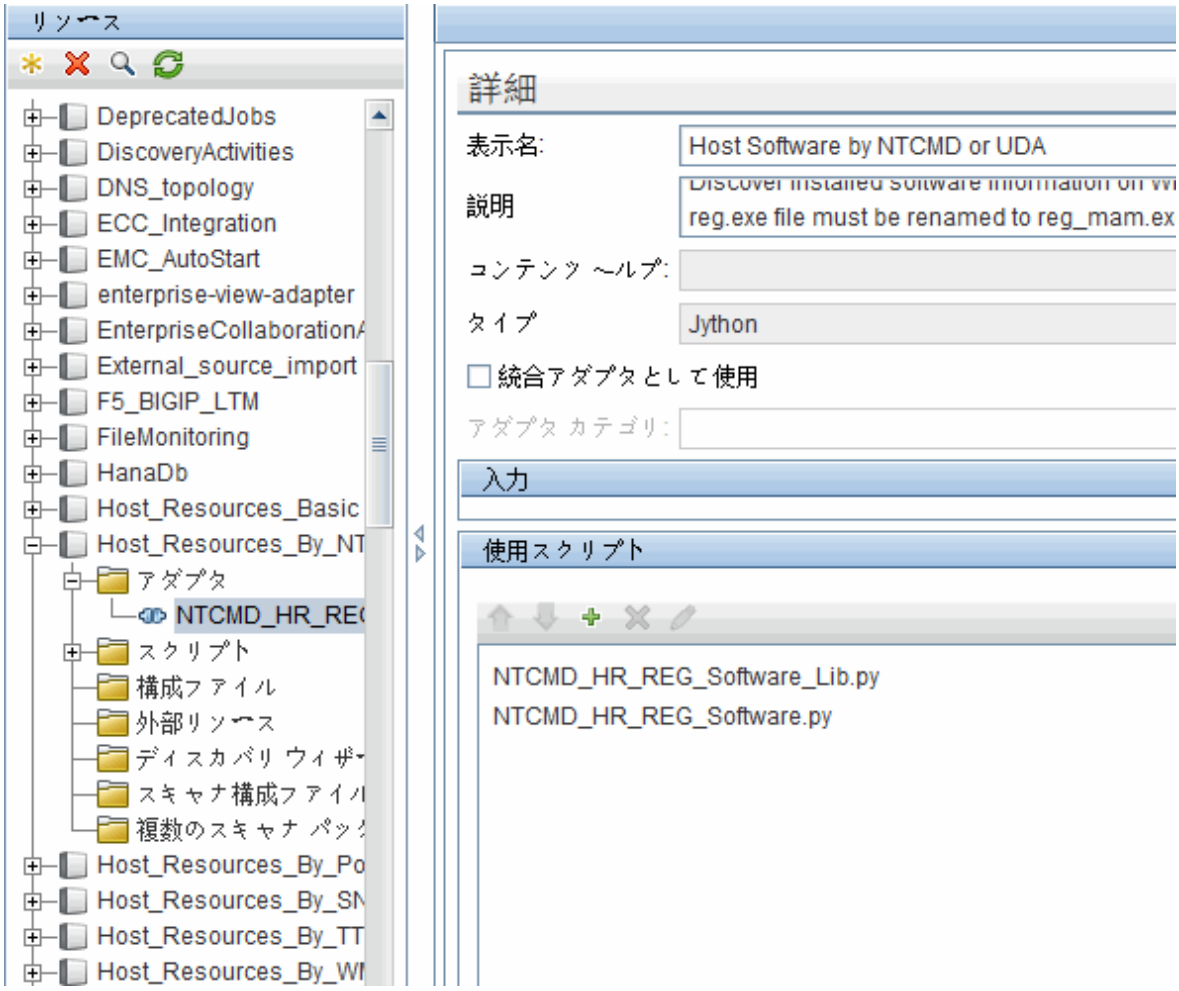
アダプタには，ディスカバリの実行時に使用されるスクリプトのリストが含まれています。新しいアダプタを作成するときは，通常，新しいスクリプトを作成し，それをアダプタに割り当てます。新しいスクリプトには基本的なテンプレートが含まれていますが，ほかのいずれかのスクリプトを右クリックして［**名前を付けて保存**］を選択すれば，それをテンプレートとして使用できます。



新しい Jython スクリプトを記述する方法の詳細については，「手順 3 :Jython コードの作成」(35ページ)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』のデータ・フロー管理の概念を参照してください。

本項の内容

1. 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，『HP Universal CMDB モデリング・ガイド』の「TQL クエリへのクエリ・ノードと関係の追加方法」を参照してください。

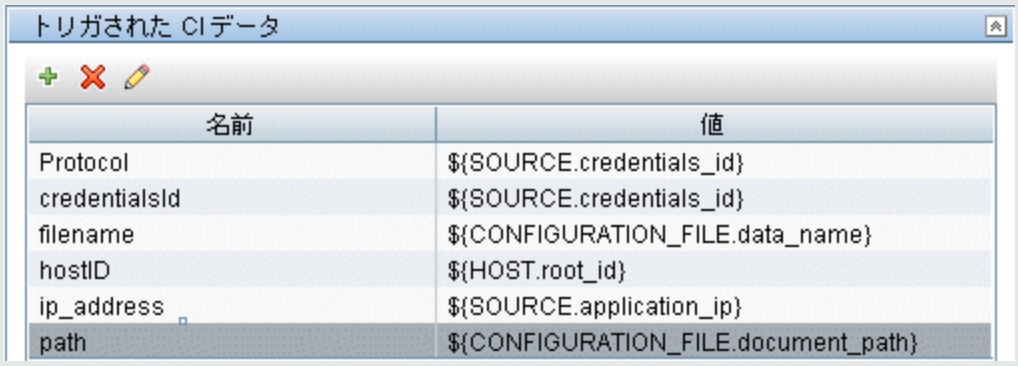
- トリガ CI のデータには，トリガ CI に関する必要なすべての情報と，（定義された場合は）入力 TQL 内のほかのノードからの情報が含まれています。DFM では，変数を使用して CI からデータを取得します。プローブにタスクがダウンロードされると，トリガ CI のデータ変数は実際の CI インスタンスの属性に存在する実際の値に置き換えられます。
- 宛先データの値がリストである場合，プローブに送信する項目数をリストから定義できます。項目数を定義するには，標準設定値の後ろにコロンを追加して，項目数を指定します。宛先データに標準設定値がない場合はコロンを 2 つ入力します。

  たとえば， name=portId, value= ${PHYSICALPORT.root_id:NA:1} または name=portId, value= ${PHYSICALPORT.root_id::1} を入力すると，ポート・リストの最初のポートのみがプローブに送信されます。

変数を実際のデータに置き換える例：

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

IpAddress 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_DOCUMENT** 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。

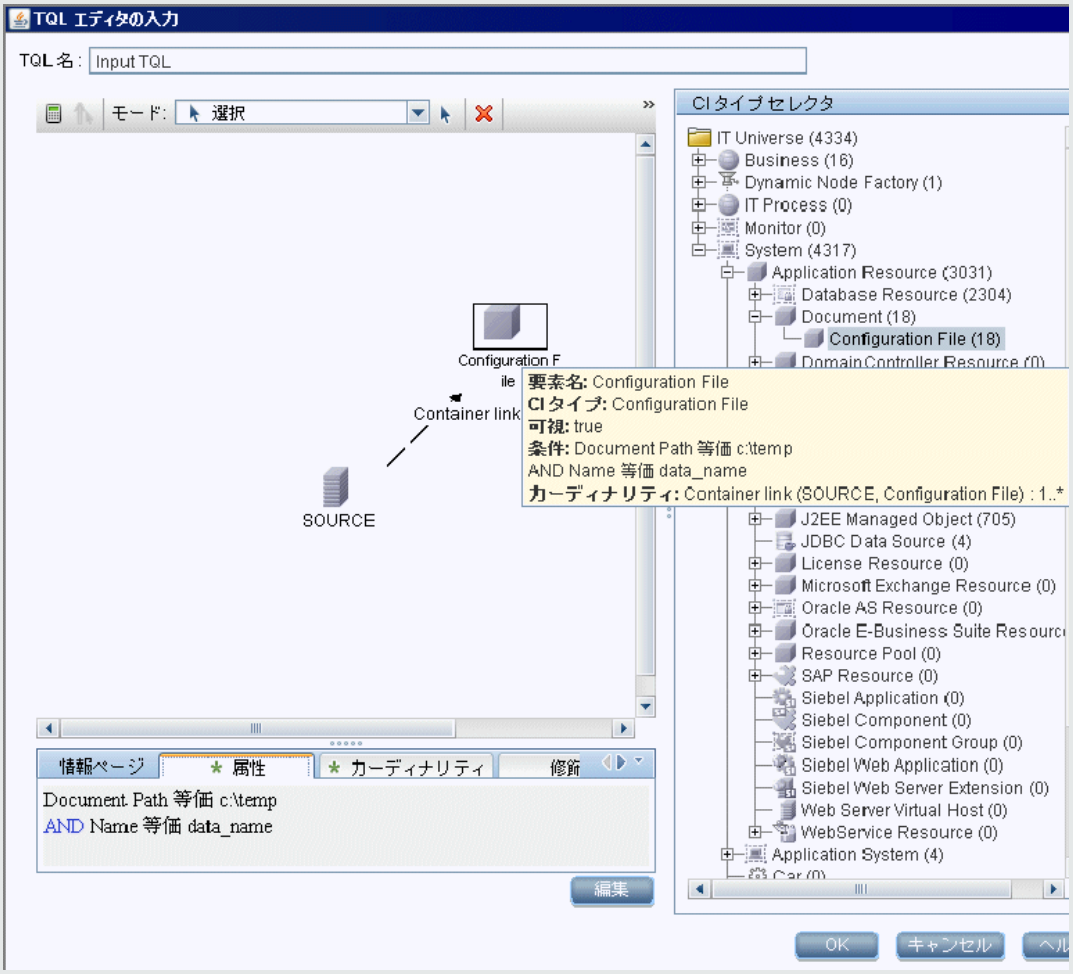


名前	値
Protocol	${SOURCE.credentials_id}
credentialsId	${SOURCE.credentials_id}
filename	${CONFIGURATION_FILE.data_name}
hostID	${HOST.root_id}
ip_address	${SOURCE.application_ip}
path	${CONFIGURATION_FILE.document_path}

トリガ CI データがプローブ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，（前の例の入力クエリ・エディタで定義された）**CONFIGURATION_DOCUMENT** ノードの data_name および document_path 属性が使用されます。



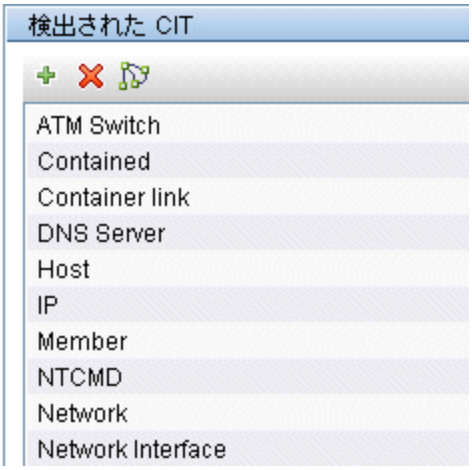
サムネイルをクリックしてフル・サイズ・イメージを表示します。

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。

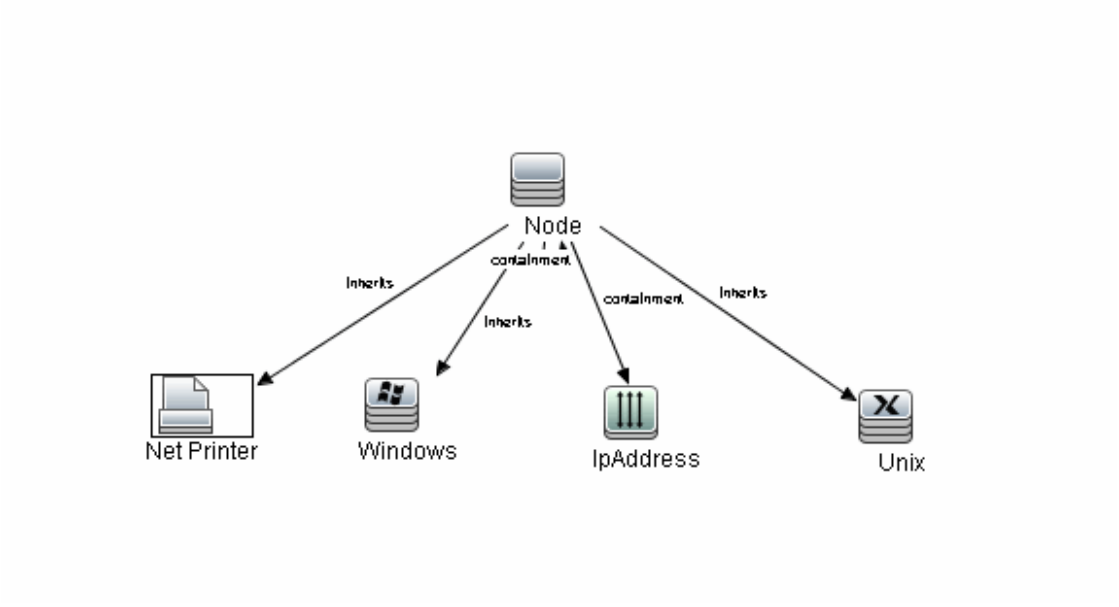
キー	名前	表示名	タイプ	詳細	標準設定値	可視
	ack_cleared_time	ack_cleared_time	long			
	ack_id	ack_id	string			
	BODY_ICON	BODY_ICON	string		ip	
	city	City	string	City locati...		✔
	codepage	CodePage	string	System s...		
	contextmenu	Context Menu	string_list	Context m...	itCIs	
	country	Country	string	Country lo...		✔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

2. 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 >［**検出された 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 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，「Jython スクリプトによる結果生成」(42ページ)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。

a. ［**データ フロー管理**］>［**アダプタ管理**］にアクセスします。

b. ［リソース］表示枠で，［**Network**］>［**アダプタ**］>［**NSLOOKUP_on_Probe**］を選択し

ます。

c. ［アダプタ定義］タブで，［検出された CIT］表示枠を見つけます。

d. 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［アダプタ定義］タブを参照してください。

3. 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。

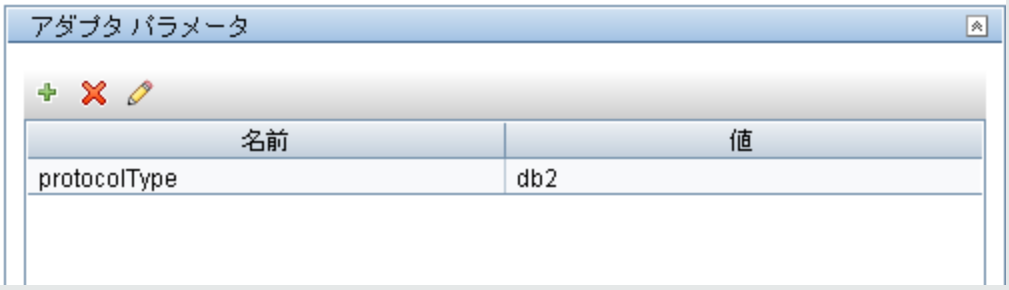
アダプタ・パラメータを上書きする例：

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。

a. **［データ フロー管理］>［アダプタ管理］**にアクセスします。

b. ［リソース］表示枠で，**［Database_Basic］>［アダプタ］>［SQL_NET_Dis_Connection］**を選択します。

c. ［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 **db2** になっています。



d. **［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 Connection by SQL］**を選択します。

e. **［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 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。

［アダプタ パラメータ］表示枠でのパラメータの追加，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［アダプタ定義］タブ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。

4. プローブ選択範囲の上書き（任意指定）

UCMDB サーバには，UCMDB が受信したトリガ CI を取得する配送メカニズムがあり，各トリガ CI に対するジョブをどのプローブが実行するかが，次のオプションのいずれかに従って自動的に選択されます。

- **IP アドレス CI タイプの場合 :** この IP に対して定義されたプローブが取得されます。
- **実行中のソフトウェア CI タイプの場合 : application_ip** 属性と **application_ip_domain** 属性を使用し，関連するドメイン内の IP に対して定義されたプローブが選択されます。
- **その他の CI タイプの場合 :** その CI の関連ノードがある場合，それに従ってノードの IP が取得されます。

自動プローブ選択範囲は，CI の関連ノードに従って実行されます。CI の関連ノードが取得された後，配送メカニズムによってノードの IP の 1 つが選択され，プローブのネットワーク範囲定義に従ってプローブが選択されます。

次の場合，プローブを手動で指定する必要があり，自動配送メカニズムを使用しません。

- アダプタに対して実行するプローブがすでにわかっており，自動配送メカニズムでプローブを選択する必要がない場合（たとえばトリガ CI が Probe Gateway の場合）。
- 自動プローブ選択範囲が失敗する可能性がある場合。これは，次のような状況で発生することがあります。
  - トリガ CI が関連ノードを持っていない場合（network CIT など）。
  - トリガ CI のノードが複数の IP を持ち，それぞれが異なるプローブに属している場合。

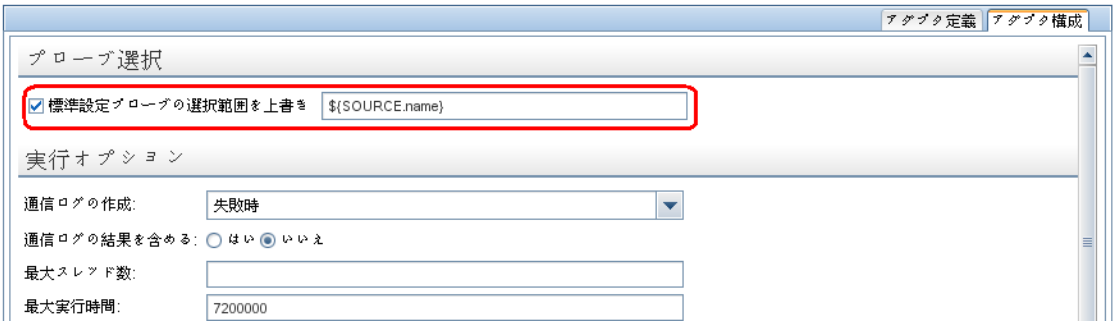
アダプタとともに使用するプローブを手動で指定するには :

- アダプタを選択し，［**アダプタ構成**］タブをクリックします。
- ［**ディスパッチ オプションをトリガする**］で，［**標準設定プローブの選択範囲を上書き**］を選択します。
- 次の形式のいずれかで，ボックスにプローブを入力します。

プローブ名	プローブの名前です
IP アドレス	プローブの IP アドレス ― IPv4 または IPv6 形式のいずれかで定義できます
IP，ドメイン	IPv4 形式 :**16.59.63.86,DefaultDomain**

	IPv6 形式 :**2001:0:9d46:953c:34a9:1e6b:f2ff:fffe,CustomDomain**
ドメイン名	プローブが選択されるドメインです。

たとえば,



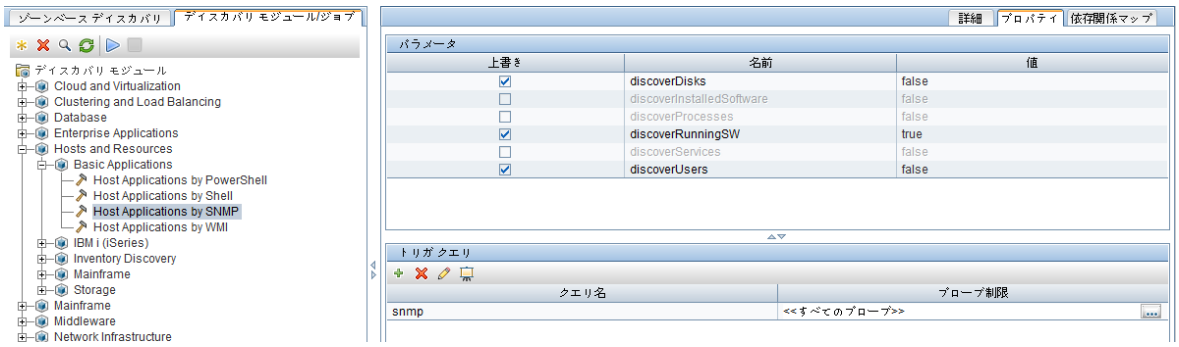
5. リモート・プロセスに対するクラスパスの構成（任意指定）

   詳細については，「リモート・プロセス実行の設定」(36ページ)を参照してください。

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，ユーザは以下の図に示すように，このエンティティをアクティブ化します。



トリガ TQL の選択

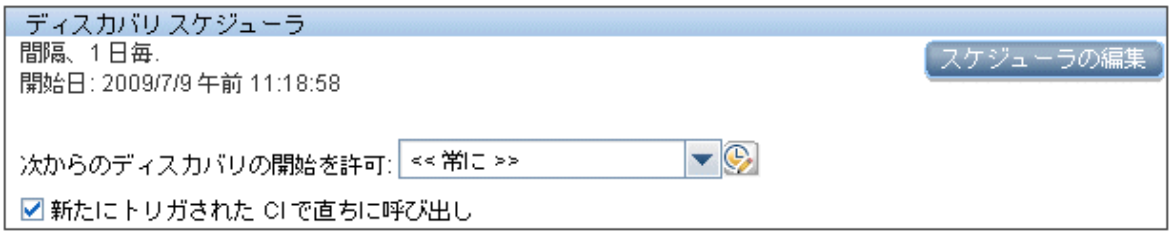
各ジョブは，トリガ TQL に関連付けられます。これらのトリガ TQL は，このジョブのアダプタに対する入力トリガ CI として使用される結果を発行します。

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。

スケジュール情報の設定

プローブ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 がプローブ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「Discovery Scheduler Dialog Box」を参照してください。

アダプタ・パラメータの上書き

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，「アダプタ・パラメータの上書き」(32ページ)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，「Jython コードの作成」(37ページ)を参照してください。

リモート・プロセス実行の設定

プロセス内のディスカバリ・ジョブに対するディスカバリの実行は，Data Flow Probe のプロセスとは別に実行できます。

たとえば，プローブのライブラリとは異なるバージョン，またはプローブのライブラリと互換性がない **.jar** ライブラリがジョブで使用される場合，ジョブを個別のリモート・プロセスで実行できます。

また，（大量のデータを運ぶため）大量のメモリをジョブが消費する可能性があり，プローブを潜在的な**メモリ不足**の問題から切り離す場合，ジョブを個別のリモート・プロセスで実行できます。

ジョブをリモート・プロセスとして実行するように設定するには，アダプタの構成ファイルに次のパラメータを定義します。

パラメータ	詳細
remoteJVMArgs	リモート Java プロセスに対する JVM パラメータ。
runInSeparateProcess	**true** に設定した場合，ディスカバリ・ジョブは個別のプロセスで実行されます。
remoteJVMClasspath	（任意）リモート・プロセスのクラスパスのカスタマイズを有効にし，標準設定のプローブ・クラスパスよりも優先します。これは，プローブの jar と，顧客が定義したディスカバリに必要な jar との間にバージョンの互換性がない可能性がある場合に便利です。 **remoteJVMClasspath** パラメータが定義されていない場合，または空のままの場合，標準設定のプローブ・クラスパスが使用されます。 新しいディスカバリ・ジョブを開発し，プローブ jar ライブラリのバージョンとジョブの jar ライブラリとの不一致が避けられるようにする場合，標準設定のディスカバリの実行に必要な最小限のクラスパスを使用する必要があります。最小限のクラスパスは，**DataFlowProbe.properties** ファイルの **basic_discovery_minimal_classpath** パラメータで定義します。 **remoteJVMClasspath** のカスタマイズの例 : • カスタム jar を標準設定のプローブ・クラスパスの前または後に追加するには，**remoteJVMClasspath** パラメータを次のようにカスタマイズします。 custom1.jar;%classpath%;custom2.jar - この場合，**custom1.jar** は標準設定のプローブ・クラスパスの前に追加され，**custom2.jar** は後に追加されます。 • 最小限のクラスパスを使用するには，**remoteJVMClasspath** パラメータを次のようにカスタマイズします。 custom1.jar;%minimal_classpath%;custom2.jar

第2章: Jython アダプタの開発

本章の内容

HP データ・フロー管理 API 参考情報

利用可能な API の完全なドキュメントについては，『HP Universal CMDB Data Flow Management API Reference』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。

<UCMDB のインストール・ディレクトリ>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_JavaDoc\index.html

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）

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 DFM 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:

- Universal Discovery 用に記述されたスクリプトは，Jython バージョン 2.5.3 と互換性がある必要があります。
- 利用可能な API の完全なドキュメントについては，『HP Universal CMDB Data Flow Management API Reference』を参照してください。

本項の内容

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 **External Resources** 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 サーバに接続するプローブ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プローブ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。

定義済みスクリプトの修正

定義済みの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，次に示す定義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 Universal CMDB 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定義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。しかし，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DiscoveryMain が含まれます。

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。メイン関数は，スクリプトで定義されたほかの関数を呼び出すことがあります。

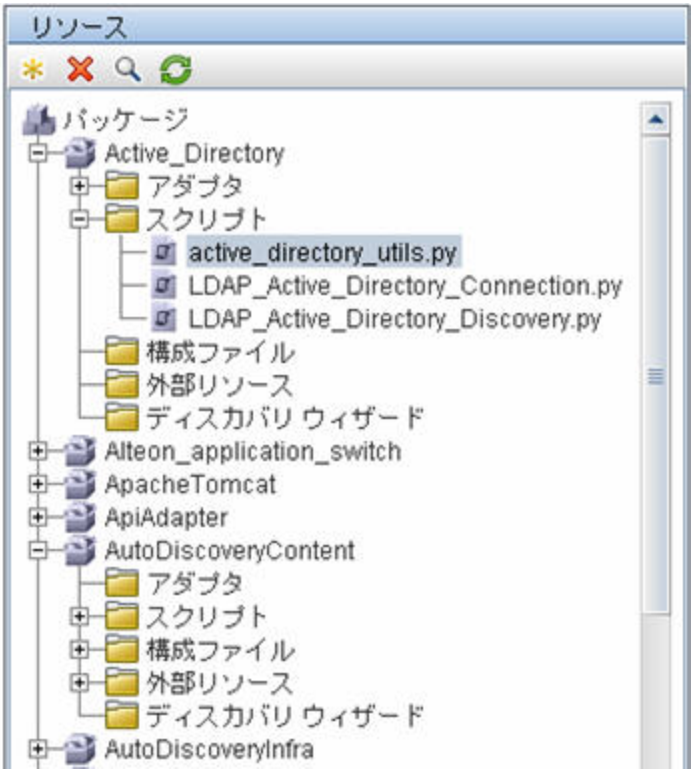
def DiscoveryMain(Framework):

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。

関数の定義

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.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID))
```

```
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

本項では，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，『HP Universal CMDB 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，「modeling.py」(62ページ) を参照してください。

大量データの送信

大量データの送信（通常，20 KB 以上）は UCMDB で処理することが困難です。このサイズのデータは，UCMDB に送信する前に小さなチャンクに分割してください。すべてのチャンクが UCMDB に正しく挿入されるようにするには，チャンクの CI に必要な識別情報を各チャンク含める必要があります。これは Jython 統合を開発する際の共通のシナリオです。sendObjects メソッドを使用して，結果をチャンクに送信します。Jython スクリプトが多数の結果（標準設定値は 20,000 ですが，この値は **appilog.agent.local.maxTaskResultSize** キーを使用してDataFlowProbe.properties ファイルで設定で

きます）を送信する場合は，そのトポロジに応じて結果をチャンキングします。このチャンキングは，結果が正しく UCMDB に入力されるように，識別ルールを考慮して実行する必要があります。Jython スクリプトが結果をチャンキングしない場合は，プローブが結果をチャンキングしようとしますが，これにより大きな結果セットのパフォーマンスが劣化することがあります。

注: チャンキングは Jython 統合アダプタのために使用して，通常のディスカバリ・ジョブには使用しないでください。これは，ディスカバリ・ジョブは通常，特定のトリガに関する情報を検出し，大量の情報を送信しないためです。Jython 統合では，大量のデータは統合の単一トリガで検出されます。

チャンキングは少数の結果にも使用できます。その場合は，異なるチャンクの CI 間に関係が存在するため，スクリプトの開発者には次の 2 つのオプションがあります。

- CI 全体とそのすべての識別情報を，リンクがあるすべてのチャンクに再び送信します。
- CI の UCMDB ID を使用します。これを行うには，UCMDB ID を取得するため，Jython スクリプトは各チャンクが UCMDB サーバで処理されるのを待つ必要があります。 このモード（同期結果送信と呼ばれます）を有効にするには，SendJythonResultsSynchronously タグをアダプタに追加します。このタグは，チャンクの送信を終了したときに，チャンク内の CI の UCMDB ID がプローブによって受信されたことを確認します。アダプタの開発者は UCMDB ID を次のチャンクの生成に使用できます。UCMDB ID を使用するには，framework API の getIdMapping を使用します。

getIdMapping の使用例

最初のチャンクで，ノードを送信します。2 番目のチャンクで，プロセスを送信します。プロセスのルート・コンテナはノードです。プロセス root_container 属性内のノードの objectStateHolder 全体を送信する代わりに，getIdMapping API を使用してノードの UCMDB ID を取得し，プロセス root_container 属性内のノード ID のみを使用してチャンクを小さくすることができます。

フレームワーク・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，「HP データ・フロー管理 API 参考情報」(37ページ)を参照してください。

Framework インスタンスの正しい使い方は，このインスタンスが使用する各メソッドに引数として渡すことです。

例:

```
def DiscoveryMain(Framework):
        OSHVResult = helperMethod (Framework)
          return OSHVResult
def helperMethod (Framework):
    ....
```

```
probe_name    = Framework.getDestinationAttribute('probe_name')
...
return result
```

本項では，Framework の最も重要な使用法について説明します。

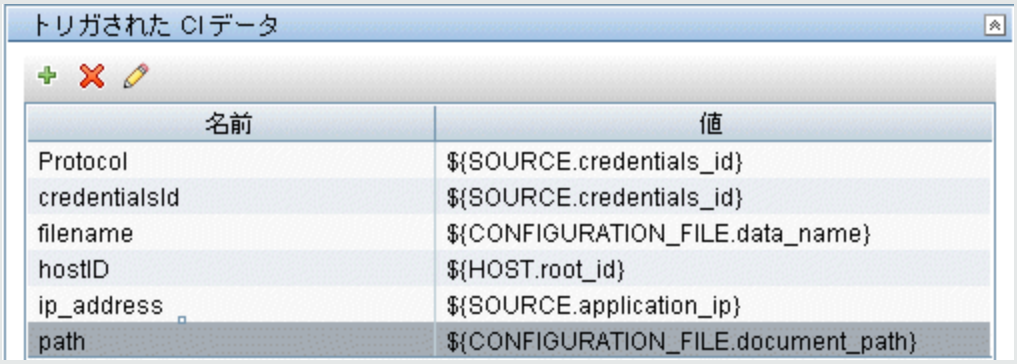
- 「Framework.getTriggerCIData(String attributeName)」(45ページ)
- 「Framework.createClient(credentialsId, props)」(45ページ)
- 「Framework.getParameter (String parameterName)」(47ページ)
- 「Framework.reportError(String message) および Framework.reportWarning(String message)」(47ページ)

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。



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 UCMDB Discovery and Integrations Content Guide』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。

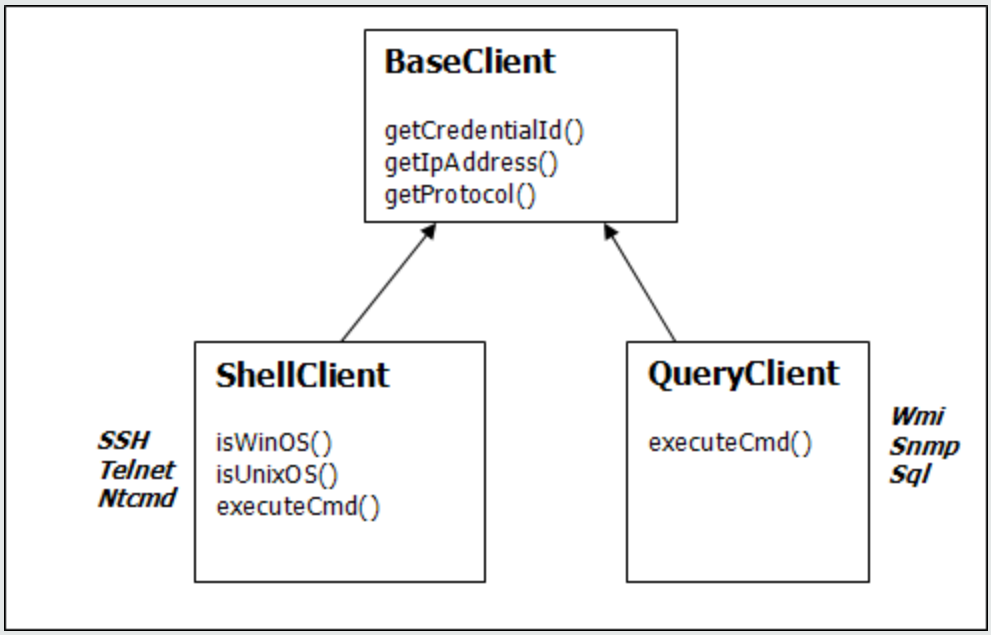
WMI クライアントを作成して WMI クエリを実行する例 :

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
                                    FROM    Win32_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)** を呼び出したときに資格情報 ID を手動で追加できます。

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。



クライアントおよびサポートされる API の詳細については，DFM FrameworkのBaseClient，ShellClient，およびQueryClient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。

<UCMDB のルート・ディレクトリ>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_Schema\webframe.html

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，

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 のユース・ケースです。

```
try:
   client = Framework.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID))

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。

エラー・メッセージの詳細については，「エラー・メッセージ」(64ページ)を参照してください。

（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。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，Framework.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）。

注: 機密データ（パスワード，秘密鍵など）へのアクセスは，次のプロトコル・タイプに対してブロックされます。

sshprotocol, ntadminprotocol, as400protocol, vmwareprotocol, wmiprotocol, vcloudprotocol, sapjmxprotocol, websphereprotocol, siebelgtwyprotocol, sapprotocol, ldapprotocol, udaprotocol, ntcmdprotocol, snmpprotocol, jbossprotocol, telnetprotocol, powershellprotocol, sqlprotocol, weblogicprotocol

これらのプロトコル・タイプの使用は，専用のクライアントを使用して行う必要があります。

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，「アダプタの分割」(23ページ) の「**2 段階接続の枠組み**」を参照してください。

例

```
import logger
import netutils
import sys
import errorcodes
import errorobject

# Java imports
from java.util import Properties
from com.hp.ucmdb.discovery.common import CollectorsConstants
from appilog.common.system.types.vectors import ObjectStateHolderVector
from com.hp.ucmdb.discovery.library.clients import ClientsConsts
from com.hp.ucmdb.discovery.library.scope import DomainScopeManager

TRUE = 1
FALSE = 0

def mainFunction(Framework, isClient, ip_address = None):
    _vector = ObjectStateHolderVector()
    errStr = ''
    ip_domain  = Framework.getDestinationAttribute('ip_domain')
    # Get the Trigger CI data values
    ip_address = Framework.getDestinationAttribute('ip_address')

            if (ip_domain == None):
              ip_domain = DomainScopeManager.getDomainByIp(ip_address, None)
```

```
    protocols = netutils.getAvailableProtocols(Framework, ClientsConsts.SNMP_PROTOCOL_
    NAME, ip_address, ip_domain)
    if len(protocols) == 0:
        errStr = 'No credentials defined for the triggered ip'
        logger.debug(errStr)
        errObj = errorobject.createError(errorcodes.NO_CREDENTIALS_FOR_TRIGGERED_IP,
    [ClientsConsts.SNMP_PROTOCOL_NAME], errStr)
        return (_vector, errObj)

    connected = 0
    # Go over all the protocol credentials
    for protocol in protocols:
        client = None
        try:
            try:
                logger.debug('try to get snmp agent for:%s:%s' % (ip_address, ip_domain))
                if (isClient == TRUE):
                    properties = Properties()
                    properties.setProperty(CollectorsConstants.DESTINATION_DATA_IP_ADDRESS, ip_
    address)
                    properties.setProperty(CollectorsConstants.DESTINATION_DATA_IP_DOMAIN, ip_
    domain)
                    client = Framework.createClient(protocol, properties)
                else:
                    properties = Properties()
                    properties.setProperty(CollectorsConstants.DESTINATION_DATA_IP_ADDRESS, ip_
    address)
                    client = Framework.createClient(protocol, properties)
                logger.debug('Running test connection queries')
                testConnection(client)
                Framework.saveState(protocol)
                logger.debug('got snmp agent for:%s:%s' % (ip_address, ip_domain))
                isMultiOid = client.supportMultiOid()
                logger.debug('snmp server isMultiOid state=%s' %isMultiOid)

    client.close()
    client = None

            except:
                if client != None:
                    client.close()
                    client = None
                logger.debugException('Unexpected SNMP_AGENT Exception:')
                lastExceptionStr = str(sys.exc_info()[1]).strip()
        finally:
            if client != None:
                client.close()
                client = None
```

```
    return (_vector, error)
```

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。

注: Python に同じ名前のベース例外クラスが存在するため，次の例に示すように Java ベースの例外をインポートすることは非常に重要です。

```
from java.lang import Exception as JException
try:
    client = Framework.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID))
except JException, ex:
    # process java exceptions only
    Framework.reportError('Connection failed')
    logger.debugException(str(ex))
     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 バージョン 2.1 から 2.5.3 への移行のトラブルシューティング

Universal Discovery は Jython バージョン 2.5.3 を使用できるようになりました。すべての定義済みスクリプトが適切に移行されています。このアップグレードの前に，ディスカバリが使用する独自の Jython スクリプトを開発した場合は，次の問題に遭遇する場合があり，指定されたフィックスを作成する必要があります。

注: これらの変更は，Jython に精通した開発者が行う必要があります。

文字列フォーマティング

- **エラー・メッセージ :** TypeError:int argument required
- **考えられる原因 :** 整数データを含んだ文字列変数からの 10 進整数に文字列フォーマティングを使用します。
- **Jython 2.1 の問題のあるコード :**

```
variable = "43"
print "%d" % variable
```

- **Jython 2.5.3 の正しいコード :**

  variable = "43"
  print "%s" % variable

  または

  variable = "43"
  print "%d" % int(variable)

文字列型のチェック

下記のコードは，入力に unicode 文字列が含まれている場合，正しく動作しない場合があります。

- **Jython 2.1 の問題のあるコード :** isinstance(unicodeStringVariable,'')
- **Jython 2.5.3 の正しいコード :** isinstance(unicodeStringVariable,basestring)

  オブジェクトのインスタンスが str **か unicode かをテストする比較は** basestring **で行う必要があります。**

ファイルの非 ASCII 文字

- **エラー・メッセージ :**
  SyntaxError:ファイル 'x' に非 ASCII 文字が含まれているが，エンコーディングが宣言されていません。詳細については，http://www.python.org/peps/pep-0263.html を参照してください。
- **Jython 2.5.3 の正しいコード :**（以下をファイルの最初の行に追加します）

  # coding:utf-8

サブパッケージのインポート

- **エラー・メッセージ :**
  AttributeError:'module' object has no attribute 'sub_package_name'
- **考えられる原因 :** インポート・ステートメントでサブパッケージの名前が明示的に指定されずに，サブパッケージがインポートされました。
- **Jython 2.1 の問題のあるコード :**

  import a
  print dir(a.b)

  サブパッケージが明示的にインポートされていません。

- **Jython 2.5.3 の正しいコード :**

  import a.b

  または

  from a import b

Iterator の変更

Jython 2.2 から開始すると，**__iter__** メソッドが **for-in** ブロックの範囲で収集のループに使用されます。iterator は **next** メソッドを実行して，収集の終わりに達したら，適切な要素を返すか，

StopIteration エラーをスローする必要があります。**__iter__** メソッドが実装されていない場合は，**getitem** メソッドが代わりに使用されます。

例外の発生

- **例外を発生させる Jython 2.1 メソッドは廃止されました：**
  raise Exception, 'Failed getting contents of file'
- **例外の発生に推奨される Jython 2.5.3 メソッド：**
  raise Exception('Failed getting contents of file')

Jython アダプタのローカリゼーション・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。

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。

本項の内容

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，「リソース・バンドルの作業」(56ページ)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，「API リファレンス」(57ページ)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（英語サイト）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，「新しい言語サポートの追加」(52ページ)を参照してください。

エンコーディングの文字セットの決定

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，Cp866 と Windows-1251 という 2 つの一般的なエンコーディングがあります。
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，所定の言語で利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），利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，利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・データと動作する新規ジョブの定義

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，「HP Universal CMDB Web Service API の概要」(333ページ)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 は，（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，「「ローカライズ・データと動作する新規ジョブの定義」(54ページ)」のリソース・バンドルからキーワードを取得する手順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。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，一般的なユース・ケースを示します。

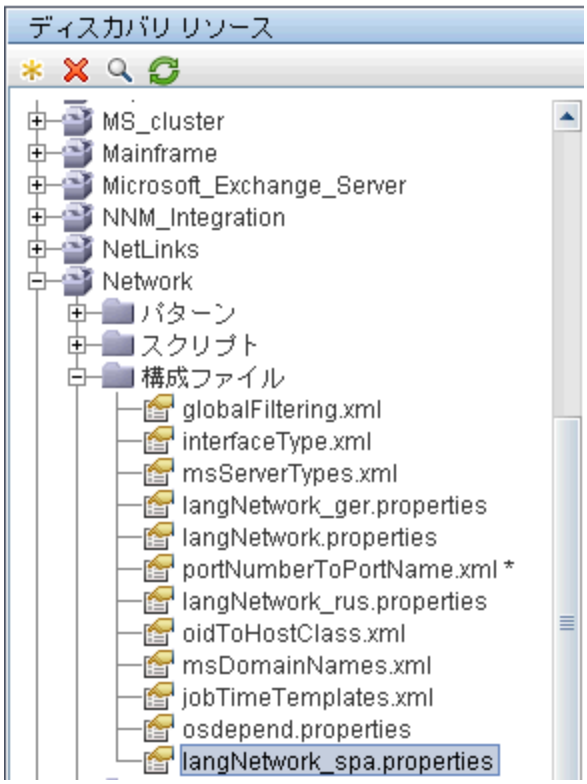
1. **executeCommandAndDecode** メソッドを 1 回呼び出します。
2. **getCharsetName** メソッドにより，直近に使用した文字セットの名前を取得します。
3. **shellUtils** の標準設定でこの文字セットを使用するため，**ShellUtils** オブジェクトで **useCharset** メソッドを呼び出します。

4. **ShellUtils** の **execCmd** メソッドを 1 回以上呼び出します。前の手順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・バンドルの作業

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** にあります。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，各リソース・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，<ベース名>_<言語識別子>.properties という特定の名前形式になります。たとえば，langNetwork_spa.propertiesです（標準設定のリソース・バンドルの形式は<ベース名>.properties です。例:langNetwork.properties)。

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利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	詳細
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。利用可能なコードのリストは http://msdn.microsoft.com/en-us/library/ aa394239

名前	詳細
	(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）。

「getCharsetName メソッド」(58ページ) 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

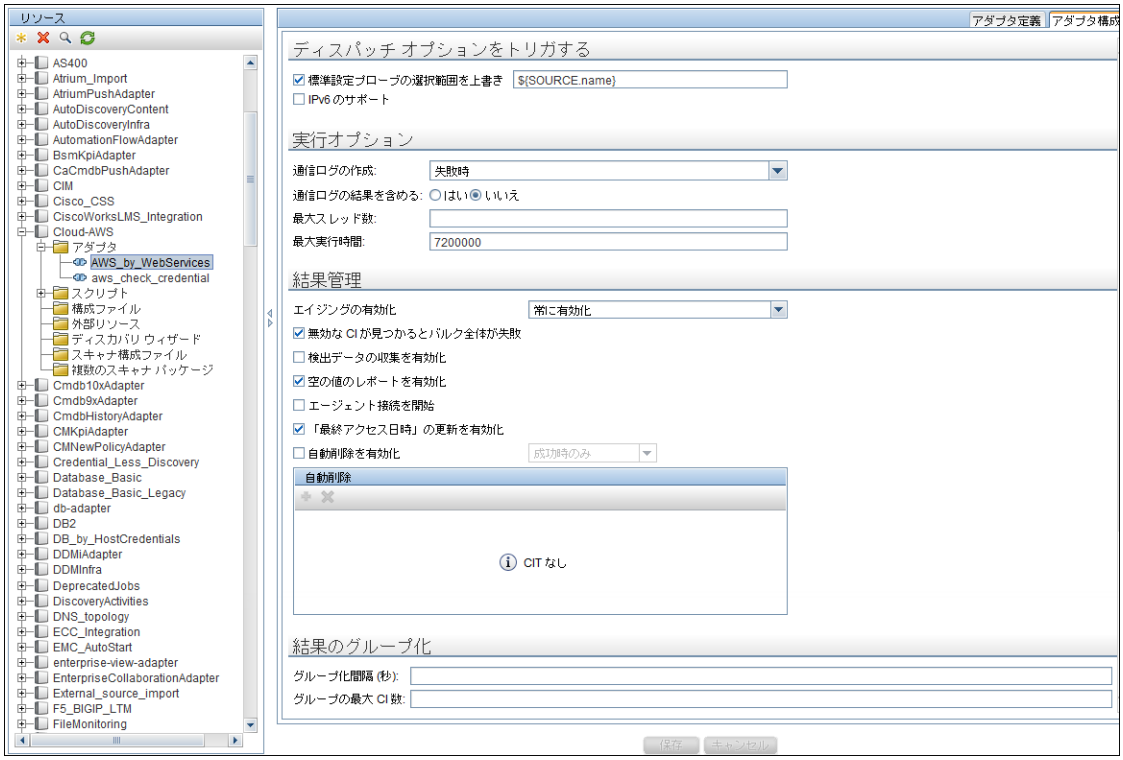
このフィールドには，言語を表すオブジェクトが格納されます。

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。

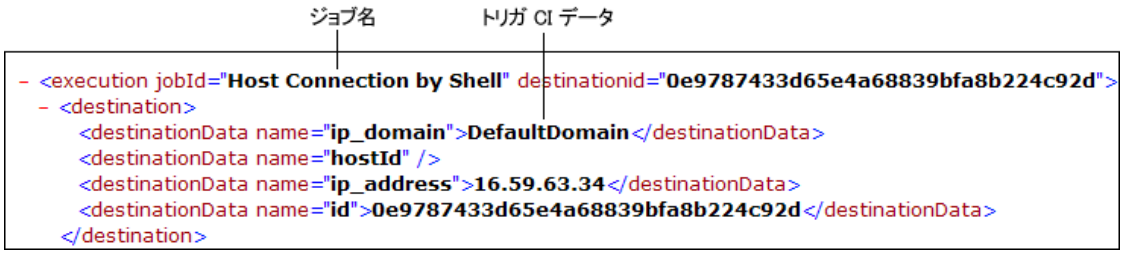
DFM コードを記録するには，次の手順を実行します。

1. ［**データ フロー管理**］>［**Universal Discovery**］にアクセスします。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，［**アダプタへ移動**］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. 下に示すように，［**アダプタ構成**］タブの［**実行オプション**］表示枠を見つけます。

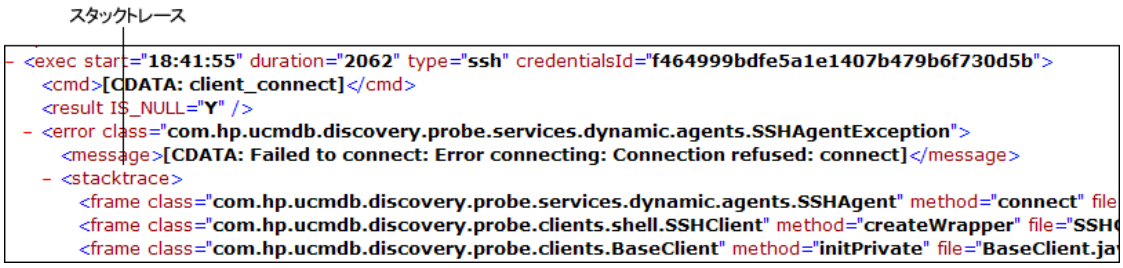


3. ［**通信ログの作成**］ボックスを［**常時**］に変更します。通信ログの設定の詳細については，『HP Universal CMDB データ・フロー管理ガイド』の「［実行オプション］表示枠」を参照してください。

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。



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。



Jython のライブラリとユーティリティ

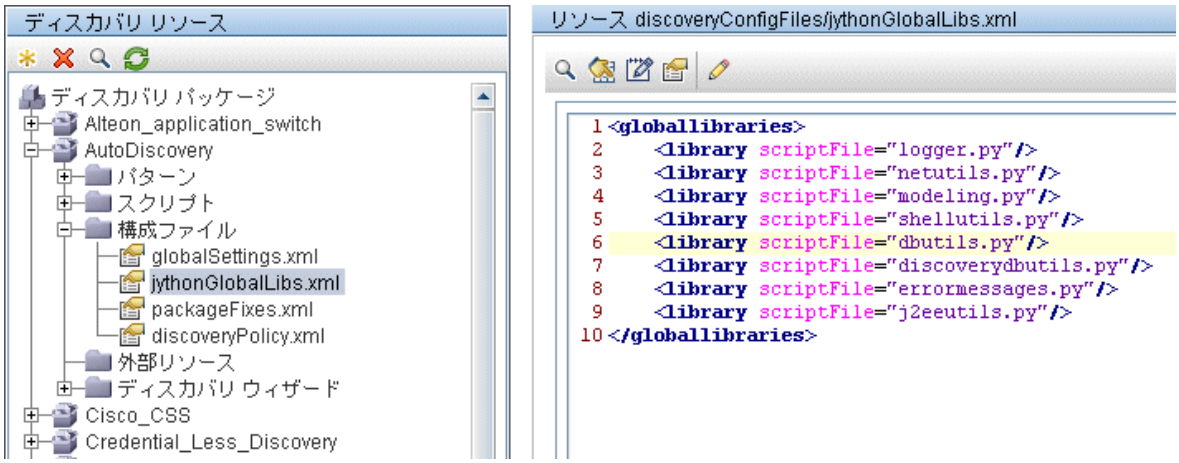
アダプタでは，いくつかのユーティリティ・スクリプトが広く使用されます。これらのスクリプトは，プローブにダウンロードされたほかのスクリプトとともに，AutoDiscovery パッケージに含まれており，**C:\hp\UCMDB\DataFlowProbe\runtime\probeManager\discoveryScripts** にあります。

注: The discoveryScript フォルダは，プローブ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・セクションに次の 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，「エラーの重大度レベル」(67ページ)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/probeMgr-
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 = Framework.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID), Props)
clientShUtils = shellutils.ShellUtils(ttyClient)
if (clientShUtils.isWinOs()):
    logger.debug ('discovering Windows..')
```

第3章: エラー・メッセージ

本章の内容

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。通常のディスカバリ・フローに失敗した場合，これらのエラーは［Universal Discovery］ウィンドウに表示されます。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，時に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 DLL 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。

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，「エラー記述の表記規則」(64ページ)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「ディスカバリの進行状況と結果」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。

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，**AutoDiscoveryInfra** パッケージの一部です。

ロケール命名規則

- 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**）。

エラー・メッセージ・コード

HP Universal CMDB には，標準設定で次のエラー・コードが含まれます。このリストに，独自のエラー・コード・メッセージを追加できます。

エラー名	エラー・コード	詳細
内部	100-199	主に，Jython スクリプトの実行時にスローされた例外から解決される
接続	200-299	接続の失敗，ターゲット・マシンにエージェントがない，宛先に到達できないなど
資格情報関連	300-399	資格情報がないことによる権限拒否，接続試行のブロック
Timeout	400-499	接続 / コマンド中のタイムアウト
予期しないまたは無効な動作	500-599	構成ファイルの欠落，予期しない中断など
情報の取得	600-699	ターゲット・マシンでの情報の欠落，エージェントでの情報のクエリの失敗など
リソース関連	700-799	メモリ不足またはクライアントが適切にリリースされていないことに関するエラー
解析	800-899	テキスト解析エラー
エンコーディング	900	入力時のエラー，サポートされていないエンコーディング
SQL 関連	901-903, 924	SQL 操作から受信したエラー
HTTP 関連	904-909	HTTP 接続中に生成され，HTTP エラー・コードから解析されるエラー
特定アプリケーション	910-923	LSOF バージョンが間違っている，キュー・マネージャがないなど，アプリケーション固有の問題を報告するエラー

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。

致命的なエラー

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。

- タスクの生成に失敗（プローブがない，変数がないなど）
- スクリプトを実行できない
- サーバで結果の処理に失敗し，データがCMDBに書き込まれない

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。

- ユーザの介入が役に立つ場合，詳細な調査を必要とする資格情報またはネットワークに関する問題が報告されます（ディスカバリのエラーではなく構成のエラーです）。
- 検出されたマシンまたはアプリケーションの予期しない動作（構成ファイルの欠落など）が通常原因となる内部エラー。

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，リモート・ホストにインストール済みエージェントが欠落しているというメッセージや，無効なデータが原因で属性が間違って計算されたというメッセージなどが含まれます。

- 接続エージェントの欠落（SNMP，WMI）
- ディスカバリは成功したが，利用可能なすべての情報を検出できるわけではない

第4章: 利用者とプロバイダの依存関係のマッピング

本章の内容

依存関係のディスカバリの概要

依存関係マッピングは，デプロイメント可能コンポーネントまたは実行中のソフトウェアの間の関係を検出できる，柔軟性の高い方法です。この方法では，（簡単なプログラミング構文を使用する）ユーザ定義の依存関係マッピング・ルールを使用できます。Universal Discovery プロセスは，このルールに基づいて依存関係を自動的に検出します。

サービスには，ビジネス・サービスと IT サービスがあります。ビジネス・サービスとは，企業間で提供するサービス（B2B），または企業内の組織間で提供するサービスのことです。IT サービスは，IT 組織がビジネス・サービスまたは IT 組織独自の業務をサポートするために提供するビジネス・サービスです。

デプロイメント可能コンポーネントとは，実行中のソフトウェア（アプリケーション・サーバや Web サーバなど）でデプロイされるソフトウェア・コンポーネントのことです。デプロイメント可能コンポーネントの例として，JEE EAR コンポーネントや，Oracle データベース内のスキーマなどが挙げられます。依存関係のディスカバリの目的上，実行中のソフトウェアはデプロイメント可能コンポーネントとみなされます。

プロバイダのデプロイメント可能コンポーネントは，サービスを提供し，ほかのデプロイメント可能コンポーネントがそのサービスを利用する方法を宣言します。**利用者**のデプロイメント可能コンポーネントは，プロバイダのデプロイメント可能コンポーネントによって提供されるサービスを「利用」します。これらのデプロイメント可能コンポーネント間の依存関係を，**利用者とプロバイダ**の依存関係と言います。

注:

- 利用者とプロバイダの依存関係のアダプタを作成し，依存関係マッピング・フレームワークを使用できるようにするためには，インストール時に UCMDB スキーマを設定するときに［**検索の有効化**］オプションを選択する必要があります。

- 利用者とプロバイダの依存関係のアダプタは，統一モードの Data Flow Probe でのみ実行可能です。

詳細情報

- 「プロバイダと利用者」(70ページ)
- 「依存関係シグネチャ」(70ページ)
- 「依存関係マッピング・フロー」(71ページ)

プロバイダと利用者

プロバイダに接続するには，接続文字列を使用します。たとえば，Oracle データベースがプロバイダである場合，そのサービスに接続するには次の情報が必要です。

- マシンの IP アドレス
- SID
- TCP ポート

利用者が目的のプロバイダによって提供されるサービスに接続するために必要な接続文字列は，この 3 種類の情報で構成されます。たとえば，Oracle の接続文字列には次のような情報が含まれることになります。

- IP アドレス : 1.1.1.1，2.2.2.2
- ポート : 1521
- SID :abcd

利用者は，特定のプロバイダについて少なくとも 1 つの接続文字列がわかります。この接続文字列は，構成ドキュメント，データベース・テーブル，Windows レジストリなどの既知の場所で見つけることができます。これらの場所を探すことにより，利用者とプロバイダの依存関係を検出できます。

プロバイダの接続文字列が特定の構成ドキュメントで見つかった場合，そのプロバイダと構成ドキュメントのコンテナが，利用者とプロバイダの関係で接続されます。

その後は，利用者とプロバイダの依存関係の検出プロセスは簡潔になります。プロバイダの接続文字列が利用者の構成ドキュメントで検索され，指定したプロバイダの利用者が所有するすべての構成ドキュメントが検索結果に出力されます。

詳細については，「依存関係を定義する」(78ページ)を参照してください。

依存関係シグネチャ

構成ドキュメントやプロバイダのタイプごとに，異なる検索用語を使用できます。これらの検索用語は依存関係シグネチャ・ファイルで定義されます。

依存関係シグネチャは，特定のプロバイダと特定の利用者との間に利用者とプロバイダの依存関係が存在するかどうかを，プロバイダの接続文字列と利用者の構成ドキュメントに基づいて定義するルールです。

依存関係シグネチャはいくつかの検索式で構成されます。これらの検索式は，接続文字列の定義によって決まるものであり，プロバイダの実際の値は関係しません。また，検索式は利用者の構成ドキュメントの名前，場所，形式に左右されますが，ファイルの実際のコンテンツの影響は受けません。プロバイダの接続文字列がすでにわかっている場合は，その接続文字列が検索式に挿入されることにより，具体的な検索式ができあがります。この具体的な検索式は，利用者の構成ドキュメントを使用して評価されます。検索式は，プロバイダの接続文字列が一定の形で利用者の構成ドキュメントに存在する場合にのみ "True" を返します。

詳細については，「依存関係シグネチャ・ファイル」(71ページ)を参照してください。

依存関係マッピング・フロー

本項では，依存関係マッピングが行われるときの基本的フローの簡単な概要を説明します。

1. デプロイメント可能コンポーネントとその接続文字列が検出されます。
2. 各タイプのプロバイダのデプロイメント可能コンポーネントによって，特定の依存関係マッピング・ジョブがトリガされます。それぞれのジョブのアダプタごとに，特定のデプロイメント可能コンポーネント・タイプに関連する接続文字列を抽出する方法が決まっています。 アダプタが，サービスを利用するその他のデプロイメント可能コンポーネントを検索します。
3. 検出された各デプロイメント可能コンポーネントとそのトリガ（プロバイダ）の間に利用者とプロバイダの関係が作成されます。

依存関係シグネチャ・ファイル

本項の内容

依存関係シグネチャ・ファイルの構造

依存関係シグネチャ・ファイルでは，デプロイメント可能コンポーネント間の 1 つ以上の依存関係を定義します。

依存関係の利用者側は，<Deployable> 要素として定義されます。各 <Deployable> 要素には，1 つまたは複数の <Dependency> 要素が含まれます。それぞれの <Dependency> 要素には，利用者（<Deployable> 要素）とプロバイダとの間に依存関係が存在するための条件が含まれます。

<Dependency> 要素は，プロバイダ CI および利用者の構成ドキュメントを入力として受け取るブール関数として確認でき，これら 2 つの CI 間に利用者からプロバイダへの依存関係が存在する場合にのみ “True” を返します。

<Dependency> 要素では，プロバイダはその CI タイプのみを使用してファイル内で指定されます。各依存関係は，プロバイダの 1 つの CI タイプに対して使用することが可能です。次の例では，実行中

のソフトウェアである利用者と実行中のソフトウェアであるプロバイダとの間の依存関係の関数を定義しています。

```
<Deployable name="ApolloOnNod">
  <Descriptor cit="running_software">
  </Descriptor>
  <Dependency name="history_db" providerCiType="running_software" scope="default">
    ...
```

変数とコンセプト

変数には，プログラムの実行中に異なる値を割り当てることが可能です。依存関係マッピングの場合は，さまざまな依存関係検索を実行するために異なる値を割り当てることができます。こうした変数は，構成ドキュメントに接続文字列が存在するかどうかを判断するために検索式を定義するときに使用します。接続文字列は各プロバイダによって異なるため，変数を使用することで，接続文字列の固有の値に関係ない汎用の検索式を定義することができます。

変数

注: 「変数」という用語は，変数とコンセプト変数の両方を意味します。

変数を使用するための構文は，${VARIABLE_NAME} です。各変数値は，文字列値または文字列リストのいずれかにする必要があります。たとえば，接続文字列の一部がプロバイダの IP アドレス（1 つまたは複数）だとします。構成ドキュメント内でこの IP アドレスを検索するには，IP_ADDRESS という変数を定義し，${IP_ADDRESS} として式の中で使用します。

変数は，依存関係シグネチャ・ファイル全体の範囲（「グローバル・スコープ」），または依存関係の範囲（「ローカル・スコープ」）に対して，最初に定義する必要があります。変数を定義することで，この変数を使用できるようになります。

定義されていない変数を使用すると，シグネチャ・ファイルをデプロイしたときにエラーが生成されます。

グローバル・スコープ

グローバル・スコープ変数は，定義されていればファイル内のどの依存関係でも使用できます。グローバル・スコープ変数を定義する方法は次のとおりです。

```
<DependencySignatures xmlns="http://www.hp.com/ucmdb/1-0-0/Dependencies">
  <VariableDeclarations>
    <Variable name="IPADDRESS"/>
    <Variable name="PROTOCOL"/>
</DependencySignatures>
```

上記では，IPADDRESS および PROTOCOL という 2 つのグローバル・スコープ変数を定義しています。すべての変数と同様に，これらの変数にも文字列値または文字列リストを割り当てることができます。

グローバル変数の値は，トリガの宛先データでのみ設定可能です。

ローカル・スコープ

ローカル・スコープ変数は，その変数の定義の対象である依存関係でのみ使用できます。デプロイメント可能コンポーネントが同じであるかどうかにかかわらず，その他の依存関係では認識されません。

ローカル・スコープ変数は，同じ名前で複数定義して，別々の依存関係内で使用することができます。この各変数は完全に独立した変数であり，その値はそれぞれが定義されているスコープ内でのみ使用できます。

注: ローカル・スコープ変数をグローバル・スコープ変数と同じ名前を持つ変数として定義する，または同じ依存関係の中で 2 つのローカル・スコープ変数に同じ名前を付けて定義することはできません。

ローカル・スコープ変数を定義するには，次の構文を使用します。

```
<Deployable name="StrongXmlApplication">
  <Descriptor cit="cluster_software"/>
  <Dependency name="app_cluster" providerCiType="running_software" scope="default">
    <VariableDeclarations>
      <Variable name="IPADDRESS"/>
      <Variable name="PROTOCOL"/>
    </VariableDeclarations>
    ...
  </Dependency>
</Deployable>
```

ローカル・スコープ変数に値を割り当てるには，挿入文を使用する必要があります。挿入文は，値の抽出元となるファイルのタイプによってそれぞれ異なります。構成ドキュメント内では次の 2 か所で挿入文を使用できます。

- 専用の <Variables> セクション。この <Variables> セクションの挿入文は，ファイルの検索式全体が True と評価された場合にのみ評価されます。
- 検索条件の一部になっている <Variables> セクション。このオプションを使用すると，該当する検索条件が True と評価された場合にのみ変数に挿入されます。代替式を使用する場合，この方法はサポートされません。詳細については，「変数の標準設定値の使用」(80ページ)を参照してください。

変数への割り当ての詳細については，次のトピックを参照してください。

- 「プロパティ構成ドキュメント」(88ページ)
- 「XML 構成ドキュメント」(92ページ)

- 「テキスト構成ドキュメント」(95ページ)

コンセプト

検索式は，関連する各構成ドキュメントに適用されます。接続文字列内の一部の値は互いに密接に結びついていて，具体的な検索式の中ではその 2 つが一緒でないと使用できません。このような結合した接続文字列のセットにはそれぞれ一意の名前が必要であり，このセットを「コンセプト」と呼びます。

たとえば，プロバイダのデプロイメント可能コンポーネントに 1.1.1.1:8080 または 2.2.2.2:85 のいずれかからアクセスできるとします。プロバイダの利用者にとっては，当然その構成ファイルの 1 つに必ず 1.1.1.1:8080 または 2.2.2.2:85 のいずれかが含まれていなければなりません。一方，このプロバイダの利用者の構成ドキュメント内で 1.1.1.1:85 が見つかる可能性は低いと言えます。このプロバイダは 1.1.1.1:85 をリッスンすることはないためです。利用者の構成ドキュメントの 1 つに 1.1.1.1:85 が含まれていた場合でも，それはこのプロバイダとの依存関係を表すのではなく，そのプロバイダと同じノードで実行中であり 1.1.1.1:85 をリッスンしている別のデプロイメント可能コンポーネントとの依存関係があることを表します。

検索では，正確に一致する IP アドレスとポートを探します。そうしないと，依存関係を誤検出して返すことになるためです。

検索ではさらに，各 IP アドレスとポートの名前についても正確に一致するものを探します。1 つの IP アドレスが複数の名前（たとえば，1 つの正式な DNS 名と複数のエイリアス名）を持つ場合があるためです。

つまり，このプロバイダの利用者の構成ドキュメントでは，XYZ:8080，FOO:8080，または ABC:85 という一致（ペア）が見つかる可能性があります（XYZ，FOO，ABC はこのプロバイダのアドレスの別名）。これらの一致はそのプロバイダとの依存関係を表すものです。しかし，ABC:8080 という一致はそのプロバイダとの依存関係を表すものではないため，このような一致は検索されないようにする必要があります。

上記の目的のために使用されるのが，コンセプトです。ある接続文字列属性と，それと組み合わせて使用されるもう 1 つの接続文字列属性は，同じコンセプト内に定義する必要があります。各接続文字列属性は，そのコンセプトの変数になります。

コンセプトは，グローバル・スコープでのみ定義できます。各コンセプト・インスタンスは密接に結びついた接続文字列のセットを表し，変数と同様に，アダプタで値を設定する必要があります。

各コンセプトは，1 つのキー変数と追加の変数で構成されます。このそれぞれの変数（キー変数と追加の変数）を使用して，密接に結びついた接続文字列を割り当てます。キー変数は，同じコンセプトのインスタンス同士を区別するために使用されます。つまり，同じコンセプトのインスタンスについては，キー変数の値が同一のインスタンスが 2 つ存在することはないということです。詳細については，「概念変数値の指定」(107ページ)を参照してください。

注: キー変数は，その他のコンセプト変数と同じように使用できます。その重要性は，コンセプトのインスタンスがトリガの実行中に作成される点にあります。

コンセプトとその変数を定義するには，次の構文を使用します。

```
<Concept name="ConceptName">
  <Properties>
    <KeyProperty name="KeyVariableName"/>
    <Property name="VariableName1"/>
    <Property name="VariableName2"/>
  </Properties>
</Concept>
```

検索式でコンセプトの変数を使用するには，次の構文を使用します。

```
${ConceptName.VariableName}.
```

検索式でコンセプトを使用する場合，そのコンセプトの関連する変数すべてを含める一方で，ほかのコンセプトの変数を含めないようにするための論理演算子を使用する必要があります。

次に示すのは，コンセプトが含まれている有効な検索式の例です。

```
${C.A} = X AND ${X.B} = Y

(${C1.A} = X AND ${C1.B} = Y) OR ${C2.C} = Z
```

こちらは無効な検索式の例です。

```
${C1.A} = X AND ${C2.B} = Y AND ${C2.X} = Z
```

標準設定値

グローバル変数およびコンセプト変数には，必要に応じて標準設定値を指定できます。アダプタから挿入される変数の値が標準設定値と同じ場合は，代替検索式を定義することが可能です。代替検索式の詳細については，「検索式の構成」(78ページ)を参照してください。

標準設定値を定義するには，次の構文を使用します。

```
<Variable name=” PORT” defaultValue=” 8080” />
```

IP Address 変数タイプ

IP アドレスは重要な接続文字列タイプの 1 つですが，プロバイダのアドレスは必ずしも構成ファイルに記述されているとはかぎりません。

たとえば，利用者とプロバイダが同じホスト・マシンを使用していることがよくあります。この場合，プロバイダのアドレスの代わりに「localhost」または「127.0.0.1」のような文字列が見つかるか，あるいはアドレスがまったく記述されていないこともあります。

変数のタイプを **IP Address** としてマークすることにより，フレームワークでは自動的にローカルの依存関係を見つけようとします。つまり，IP アドレスの変数を無視して，同じホスト上に存在する接続文字列の残りの部分に基づいてプロバイダを検索します。

変数を IP アドレスとしてマークするには

- グローバル変数の場合

```
<DependencySignatures xmlns="http://www.hp.com/ucmdb/1-0-0/Dependencies">
  <VariableDeclarations>
    <Variable name="MY_VARIABLE" type=” IP Address” />
  </VariableDeclarations>
      …
</DependencySignatures>
```

- コンセプト変数の場合

```
<Concept name="IpEndpoint">
  <Properties>
    <KeyProperty name="PORT"/>
    <Property name="MY_VARIABLE" type=” IP Address” />
  </Properties>
</Concept>
```

注: ローカル変数に IP Address タイプを指定することはできません。

利用者の記述子を定義する

利用者とプロバイダの依存関係を定義するために，検索アダプタでは，プロバイダの接続文字列を使用する構成ドキュメントを持つ利用者を検索します。ただし，利用者の構成ドキュメントに接続文字列が含まれている，あるいはデプロイメント可能コンポーネントのプロパティに応じて異なる出力変数が含まれている場合でも，特定のサービスの利用者の可能性があるデプロイメント可能コンポーネントに制限すると効果的な場合もあります。

そこで，<Descriptor> 要素を使用することで，CI タイプよりもさらに詳細にデプロイメント可能コンポーネントを記述することが可能となっています。デプロイメント可能コンポーネントは，次のいずれかを使用して記述できます。

- CI タイプ : 依存関係は，“J2EE Application” タイプのデプロイメント可能コンポーネントのみ該当します。（必須）
- 文字列属性の条件 : たとえば，“J2EE Application” タイプのデプロイメント可能コンポーネントで，アプリケーション名が “MyShop” であるもの。この条件では，相等性のテストのみできます。（任意指定）
- 別の CI への必須の複合リンク : この複合リンクが記述子に指定されている場合は，デプロイメント可能コンポーネントに特定のタイプの CI への複合リンクを含める必要があります。（任意指定）

- 接続先 CI（前のオプションで指定されている場合）の文字列属性の条件。（任意指定）
- デプロイメント可能コンポーネントを含むノードへのパス（TQL クエリ）: クエリ名は，**nodeToDeployableQuery** 属性を使用して指定します。このノードが複合リンクによりデプロイメント可能コンポーネントに直接接続されている場合，パスとして TQL クエリを記述する必要はありません。ただし，ノードが（条件の指定の有無にかかわらず）デプロイメント可能コンポーネントを記述するために使用されている場合は，<ConnectedCICondition> タグを使用してパスを記述する必要があります。デプロイメント可能コンポーネントを含むノードが親階層のどこにもない場合は，属性 hasContainingNode = “false” を使用してそのことを記述します。パスを記述するときは，必ず hasContainingNode = “true” も追加します。詳細については，「TQL クエリを定義する」(100ページ)を参照してください。（任意指定）

注: STRING 属性タイプのみがデプロイ可能な記述子または接続した CI でサポートされます。この属性が静的になることはなく，計算された属性を含むことはできません。

CI タイプのみ指定する記述子の例:

```
<Deployable name="ApolloOnNode_2">
  <Descriptor cit="running_software">
  </Descriptor>
    ...
```

文字列属性を指定する記述子の例:

```
<Deployable name="ApolloOnCluster">
  <Descriptor cit="node">
    <Attribute name="default_gateway_ip_address_type" value="IPv6" />
  </Descriptor>
    ...
```

大文字 / 小文字を区別する相等性テストにより接続先 CI を指定する記述子の例:

```
<Deployable name="MyRunningSoftware">
  <Descriptor cit="running_software">
    <ConnectedCiCondition cit=” node” linkType=” composition” isDirectionForward=” true” >
      <Attribute name=” name” value=” MyNode” operator=” equalIgnoreCase” />
    </ConnectedCiCondition>
  </Descriptor>
    ...
```

注: ConnectedCiCondition では，「linkType=” composition”」および「isDirectionForward=” true”」のみ使用できます。

依存関係を定義する

利用者には複数の依存関係を定義できます。依存関係は，それぞれ特定のプロバイダ CI タイプと関連付けられます。プロバイダ CI の依存関係シグネチャの評価中に，そのプロバイダの CI タイプと関連付けられているすべての依存関係が評価されます。各依存関係には，1 つ以上の構成ドキュメントに基づく検索式があります。その検索式が True として評価された場合は，利用者とプロバイダの間に依存関係（利用者とプロバイダの関係）が存在します。同じ利用者とプロバイダ CI タイプの間に複数の依存関係が存在し，それぞれが True と評価された場合でも，作成される関係は 1 つのみとなります。

次に示すのは依存関係構文の例です。

```
<Deployable name="Websphere J2EE Application">
  <Descriptor cit="j2eeapplication"/>
  <Dependency name="J2EE Application to DB by JNDI" providerCiType="oracle" scope="default">
    ...
```

依存関係には必ずスコープも関係します。スコープとは，記述子に一致するすべての利用者のうち，この特定の依存関係に関連する利用者のことです。詳細については，「利用者の記述子を定義する」(76ページ)を参照してください。

検索式の構成

注: STRING 属性タイプのみがデプロイ可能な記述子または接続した CI でサポートされます。この属性が静的になることはなく，計算された属性を含むことはできません。

検索式は，論理演算子と条件で構成されます。サポートされる論理演算子は「And」と「Or」です。

条件は，プロバイダの接続文字列と利用者の構成ドキュメントを使用して評価される式です。条件はファイルのタイプによって異なります。つまり，プロパティ構成ドキュメントと XML ドキュメントでは異なる条件が使用されます。

次に示すのは，プロパティ構成ドキュメント用の検索式の例です。ここでは，プロバイダと利用者の間の依存関係を作成するためには，キーが IPADDRESS のときにプロバイダの IP アドレスが利用者の **MyConfig.properties** 構成ドキュメントに存在することが唯一の条件です。これは，依存関係の構成ドキュメントでは次のように記述されます。

```
<PropertiesConfigurationDocument name="MyConfig.properties">（構成ドキュメントのタイプおよび名前）
  <Condition>（検索式の開始）
    <Operator type="and">（演算子）
      <KeyCondition key="IPADDRESS">（条件の開始とそのタイプの宣言）
        <Values>
          <Value>${IP_ADDRESS}</Value>（変数を使用してプロバイダの IP アドレスを宣言）
        </Values>
```

```
    </KeyCondition>
   </Operator>
  </Condition>
</PropertiesConfigurationDocument>
```

注: <Condition> 要素内には <Operator> 要素が少なくとも 1 つ必要です。これは，上記の例のように検索条件が 1 つしかないため論理上は不要な場合にもあてはまります。

次に，より複雑な検索式を見てみましょう。ここでの条件は，キーが IPADDRESS の場合にプロバイダの IP アドレスが利用者の **MyConfig.properties** 構成ドキュメントに存在すること，またはキーが HOST の場合にプロバイダのホスト名が同じファイルに存在することです。

```
<PropertiesConfigurationDocument name="MyConfig.properties">（構成ドキュメントのタイプおよび名前）
  <Condition>（検索式の開始）
    <Operator type="or">（演算子）
      <KeyCondition key="IPADDRESS">（条件の開始とそのタイプの宣言）
        <Values>
          <Value>${IP_ADDRESS}</Value>（変数を使用してプロバイダの IP アドレスを宣言）
        </Values>
      </KeyCondition>
      <KeyCondition key="HOST">（同じ演算子の下で別の条件を開始）
        <Values>
          <Value>${HOSTNAME}</Value>（変数を使用してプロバイダのホスト名を宣言）
        </Values>
      </KeyCondition>
    </Operator>
  </Condition>
</PropertiesConfigurationDocument>
```

論理演算子をネストして，(C1 AND (C2 OR C3)) のような条件を作成することが可能です。つまり，XML は次のようになります。

```
<PropertiesConfigurationDocument name="MyConfig.properties">（構成ドキュメントのタイプおよび名前）
  <Condition>
    <Operator type="and">
      C1
      <Operator type=”or”>
        C2
        C3
      </Operator>
    </Operator>
  </Condition>
</PropertiesConfigurationDocument>
```

C1，C2，C3 は，必要な検索条件の XML スニペットです。

サポートされるファイルのタイプは 3 つあり，それぞれ検索式の種類が異なります。

- PropertiesConfigurationDocument – 各行で key=value という形式を使用するファイル。詳細については，「プロパティ構成ドキュメント」(88ページ)を参照してください。

  KeyCondition – 特定のキーの値に対する条件
- XmlConfigurationDocument – XML ファイル。詳細については，「XML 構成ドキュメント」(92ページ)を参照してください。

  XPathCondition – XPath クエリを評価する
- TextConfigurationDocument – テキスト構成ドキュメント。詳細については，「テキスト構成ドキュメント」(95ページ)を参照してください。

  RegExp – 正規表現を評価する

標準設定値の詳細については，「変数の標準設定値の使用」(80ページ)を参照してください。

トラブルシューティング

- 条件ツリーの同じノードに異なるコンセプトを含めることはできません。同様に，同じ演算子の下でコンセプトが異なる 2 つのリーフを含めることもできません。

  たとえば，次のような条件定義は無効となります。

```
<Condition>
  <Operator type="and">
    <XPathCondition>
      <XPath>/Setup/Configuration/Hostname[matches(@Name, '.*?${Concept1.HOSTNAME}.*')]
</XPath>
    </XPathCondition>
    <XPathCondition>
      <XPath>/Setup/Configuration/Port[matches(text(), '${Concept2.PORT}')]</XPath>
    </XPathCondition>
  </Operator>
</Condition>
```

変数の標準設定値の使用

接続文字列の値が標準設定値と等しいと，構成ファイルにその値が含まれていないという場合があります。たとえば，HTTP URL を定義する際，ポート値 80 は URL 内に明示されないことがあります。つまり，構成ファイル内では「http://www.hp.com:80/」ではなく「http://www.hp.com/」として指定されている可能性が高いということです（どちらも正しい指定です）。こうした状況では，検索式を記述するときに問題が発生する可能性があります。PORT 変数に値 80 が割り当てられていて，この値が存在することが条件になっている場合，値が指定されていないと URL のテストが失敗するためです。

この問題を解決するため，必要に応じて変数ごとに標準設定値を指定できるようになっています。変数の値が標準設定値と同じ場合には，検索式を変更できます。

検索式を変更するには，次の 2 つの方法があります。

- 変数の値が標準設定値と同じ場合は条件を無視する

  この方法を使用すると，検索条件内の 1 つまたは複数の変数の値が標準設定値と同じ場合に，条件が完全に無視されます。条件が無視された場合，True も False も返されません。論理演算子とその他のオペランドに応じた結果が出力されます。次に例を示します。

  - 「**A AND B**」という形式の条件の場合（**A** と **B** は別の式），**A** が無視されると，この条件の結果は **B** の結果と同じになります。
  - 式が「**A AND B AND C**」の場合，**A** が無視されると，結果は「**B AND C**」と同じになります。
  - 式が「**A OR B OR C**」の場合，**A** が無視されると，結果は「**B OR C**」と同じになります。

  稀にすべての副条件が無視される，つまり条件全体が無視されることがありますが，その場合，構成ドキュメントの条件の結果は False になります。

  条件を無視するには，目的の条件に **ignoreIfDefaultValue** 属性を追加して，この条件の無視をトリガする変数のリストを指定します。次に例を示します。

```
<KeyCondition key="serverName" ignoreIfDefaultValue="IPADDRESS">
```

- 変数の値が標準設定値と同じ場合は別の条件を使用する

  PORT 80 と URL の例を続けます。次の正規表現を使用して URL のテストを行います。

```
http://${DOMAIN}:${PORT}/
```

  この正規表現が「http://www.hp.com/」のような文字列について True と評価しないことは明らかです。理由は，この文字列にはコロン（:）が含まれていないためです。したがって，PORT 変数の値が **80** の場合には次の正規表現でもテストできるようにする必要があります。

```
http://${DOMAIN}/
```

  このような状況では，代替式を使用できます。代替式を定義するには，同じ条件の下で複数の検索式を定義し，変数に標準設定値が入る場合に使用する方の式を指定します。次に例を示します。

```
<KeyCondition key="url">
  <RegExp>http://${DOMAIN}:${PORT}/</RegExp>
  <RegExp alternativeFor=” PORT” >http://${DOMAIN}/</RegExp>
</KeyCondition>
```

  alternativeFor 属性を使用して，変数に標準設定値が割り当てられている場合は元の式の代わりに代替式が評価されるように宣言します。元の式は，**alternativeFor** 属性の指定がない，または空の **alternativeFor** 属性が指定されている式です。

  式に含まれる，標準設定値を持つすべての変数について，それらの変数のどのような組み合わせにも対応できるように代替式を定義する必要があります。そうしないと，コンパイル・エラーが

発生することになります。変数が 1 つしかない場合は，上記の例のように，必要な代替式は 1 つで済みます。標準設定値を持つ変数が複数ある場合は，複数の代替式が必要です。

たとえば，DOMAIN 変数にも標準設定値が設定されている場合，**alternativeFor** 文を使用して次のすべての代替条件を指定することになります。

```
<KeyCondition key="url">
  <RegExp>http://${DOMAIN}:${PORT}/</RegExp>
  <RegExp alternativeFor=" PORT " >http://${DOMAIN}/</RegExp>
  <RegExp alternativeFor=" DOMAIN " >http://www.hp.com:${PORT}/</RegExp>
  <RegExp alternativeFor=" PORT, DOMAIN " >http://www.hp.com/</RegExp>
</KeyCondition>
```

実行時に評価されるのは，これらの組み合わせの中の 1 つのみです。

変数の標準設定値の指定方法の詳細については，「標準設定値」(75ページ)を参照してください。

構成ドキュメントのパスを指定する

構成ドキュメントのパスとは，ホストのファイル・システム上にあるファイルのパスではなく，利用者のデプロイメント可能コンポーネントと構成ドキュメント間のトポロジ・パスのことを指します。

標準設定では，パスが指定されていない場合，構成ドキュメントはデプロイメント可能コンポーネントへの複合リンクにより接続されている，つまり，デプロイメント可能コンポーネント CI が構成ドキュメント CI を所有すると仮定されます。

異なるパスを指定するには：

1. 「TQL クエリを定義する」(100ページ)の手順に従って，TQL クエリを定義します。
2. <DocumentCILocation> 要素を使用して，構成ドキュメントから TQL クエリを参照します。次に例を示します。

```
<TextConfigurationDocument name="cmdb.properties">
  <DocumentCILocation>
    <ReferenceLocation>YourQueryName</ReferenceLocation>
  <DocumentCILocation>
  <Condition>
    ...
  </Condition>
</TextConfigurationDocument>
```

この例では，YourQueryName が <Queries> セクションのクエリ名への参照です。

パスを指定するために TQL クエリを作成するときは，次のことに注意してください。

- TQL クエリでは，デプロイメント可能コンポーネントと構成ドキュメント間のパスを定義します。このパスは単純な（循環を含まない）パスにする必要があります。
- TQL クエリには，大文字と小文字を区別した固有の名前を持つ，次の 2 種類のエンド・ノードを追

加します。

- Deployable – パス内のデプロイメント可能コンポーネント CI
- Configuration_document – パス内の構成ドキュメントの CI

- Deployable ノードと Configuration_document ノードに対して条件を設定することはできません。
- すべてのノード間のカーディナリティは必ず 1..1 にします。
- 通常のリンク・タイプのみサポートされます。複合リンク，結合リンク，サブグラフは使用できません。

構成ドキュメントの上書き

構成ドキュメントは，ほかの構成ドキュメントで上書きできる場合があります。たとえば，ホストに存在する構成ドキュメントが，そのホストが属するクラスタに存在する構成ドキュメントを上書きすることがあります。このような場合，キーの値はその値を上書きする可能性があるすべてのファイルを調べて，優先度が一番高いものを選びます。この例では，ホストのローカル構成ドキュメントの方が，クラスタのドキュメントよりも高い優先度が設定されていることになります。

依存関係シグネチャでファイルの上書きを定義するには，次の構文を使用して，構成ドキュメントに対する優先度を設定した複数のパスを追加します。

```
<PropertiesConfigurationDocument name="resources.xml">
  <DocumentCILocation>
    <ReferenceLocation priority="2">websphereas_resource_configfiles</ReferenceLocation>
    <ReferenceLocation priority="3">j2ee_cluster_configfiles</ReferenceLocation>
    <ReferenceLocation priority="1">j2eeapplication_configfiles</ReferenceLocation>
  </DocumentCILocation>
  ...
</PropertiesConfigurationDocument
```

ファイル名（上記の例では **resources.xml**）は，すべての参照場所で同じである必要があります。

なお，参照場所は，利用者のデプロイメント可能コンポーネントから構成ドキュメントへのトポロジ・パスを指定する TQL クエリへの参照です。つまり，このパスはデプロイメント可能コンポーネントとこれらすべての場所との間に存在する必要があることも意味します。<DocumentCILocation> の詳細については，「構成ドキュメントのパスを指定する」(82ページ)を参照してください。

優先度を指定した複数のパスを追加した場合でも，条件自体は変わりません。実行時（検索式が評価されるとき）に，優先度に基づいた適切な値が使用されます。たとえば，2 つのプロパティ・ファイルに優先度 1 と優先度 2 が設定されていて，その両方に存在するキー「K」の値に基づく条件が設定されている場合，この条件は優先度 1 のドキュメントに対してのみ評価されます。キー「K」が優先度 2 の方にのみ存在する場合は，条件が評価されるのは優先度 2 のドキュメントのみとなります。

プロパティ条件の詳細については，「プロパティ構成ドキュメント」(88ページ)を参照してください。

XML ドキュメントでは，それぞれの XPath がキーとみなされます。たとえば，\Root\Element\@Attribute は，そのパスの属性「Attribute」がキーであり，その値はファイルごとに上書きされる可能性があります。

XPath に不変の条件（変数を含まない条件）も含まれている場合，その条件はキーの一部となります。次に例を示します。\Root\Element[@name = ‘name’]\@Attribute.

ただし，XPath に可変の条件が含まれている場合は，その条件はキーから除外され，値の一部とみなされます。つまり，\Root\Element[@name = ${NAME}]\@Attribute というパスの場合，キーはパス \Root\Element\@Attribute で，条件 Element[@name = ‘name’] は，そのキーが存在する優先度が一番高いドキュメントに対してのみ評価されます。

XML ドキュメントで優先度を使用する場合は，予期しない不正な動作が起きないようにするために，XPath の条件を上記の説明に従って作成することが重要です。

XPath の条件の詳細については，「XML 構成ドキュメント」(92ページ)を参照してください。

注: 優先度を指定した複数の参照場所は，プロパティ構成ドキュメントおよび XML 構成ドキュメントでのみ使用できます。テキスト・ドキュメントでは使用できません。

同じ優先度が設定された複数の構成ドキュメント

異なる <ReferenceLocation> 要素に対して同じ優先度を指定することも可能です。この場合，フレームワークでは，条件を評価する際に，優先度が同一のすべてのファイルを調べてその全部を照合します。

設定したファイル数以上が True と評価された場合，条件は True と評価されます。このファイル数は，**samePriorityMatchAtLeast** 属性で定義します。

次に例を示します。

```
<PropertiesConfigurationDocument name="resources.xml">
  <DocumentCILocation samePriorityMatchAtLeast=” 1” >
    <ReferenceLocation priority="1">websphereas_resource_configfiles</ReferenceLocation>
    <ReferenceLocation priority="1">j2ee_cluster_configfiles</ReferenceLocation>
    <ReferenceLocation priority="1">j2eeapplication_configfiles</ReferenceLocation>
  </DocumentCILocation>
…
</PropertiesConfigurationDocument>
```

これは，ある条件を評価するときに，参照されるファイルの場所のうち少なくとも 1 つは，その条件が True と評価されるファイルの場所だということです。

注: 現在，**samePriorityMatchAtLeast** 属性でサポートされている値は **1** のみです。

複数ドキュメント間で定義された依存関係

多くの場合，利用者がサービスを提供するかどうかを判別するには，プロバイダのすべての接続文字列を見つけるために複数の構成ドキュメントを検索する必要があります。

関連付けされていない複数の構成ドキュメントの検索

構成ドキュメントは相互に関連付けされていない可能性があり，あるドキュメントと別のドキュメントのそれぞれで接続文字列が定義されていることがあります。たとえば，利用者の構成ドキュメントとして A.conf と B.conf の 2 つがあるとします。プロバイダの接続文字列は，A.conf で定義されている「C1」と，B.conf で定義されている「C2」です。利用者とプロバイダの間に実際に依存関係があるかどうかを判断するには，C1 と C2 の両方が必要です。この場合，必要な検索式は 2 つ（一方のドキュメントに 1 つずつ）あります。つまり，<Dependency> タグの構造は次のようになります。

```
<Dependency name="dependency_name" providerCiType="webmodule" scope="my_scope">
  <PropertiesConfigurationDocument name="A.conf">
    <Condition>
      <Operator type="and">
        <KeyCondition key="C1">
          <Values>
            <Value>${VAR_1}</Value>
          </Values>
        </KeyCondition>
      </Operator>
    </Condition>
  </PropertiesConfigurationDocument>
  <TextConfigurationDocument name="B.conf">
    <Condition>
      <Operator type="and">
        <RegExpCondition>
          <RegExp>C2?${VAR_2}</RegExp>
        </RegExpCondition>
      </Operator>
    </Condition>
  </TextConfigurationDocument>
</Dependency>
```

注:

- **<Dependency> 要素内に記述できるファイルの数に制限はありません。**
- **各ファイルのタイプが異なっていてもかまいません（プロパティ，XML，またはテキスト）。**
- **<Dependency> 要素内のファイルの順序は無視されます。ファイルは任意の順序でテストされます。**
- **すべてのファイルの検索式が True を返せば，依存関係が存在します。**
- **接続文字列が複数のファイルに存在するため，スコープの検索式に十分な余裕を持たせて，必要なファイルのうち少なくとも 1 つがそのスコープから返されるようにする必要があります。詳細については，「検索のスコープを指定する」(97ページ)を参照してください。**

複数のファイルを検索するときの流れをまとめると次のようになります。

1. 「検索のスコープを指定する」(97ページ)の説明に従ってスコープのフィルタリングを実行します。必要なファイルが 1 つも返されない場合，利用者とプロバイダの間に依存関係は存在しません。
2. フィルタの結果少なくとも 1 つのファイルが返された場合，そのファイルに接続されている利用者と必要な構成ドキュメントの残りの部分が Data Flow Probe にダウンロードされます。
3. 各ファイルの条件が任意の順序で評価されます。
4. すべての条件が True として返された場合は依存関係が存在し，それ以外の場合は依存関係は存在しません。

依存関係がある複数の構成ドキュメントの検索

その他の場合では，利用者の構成ドキュメントは相互に関連付けられています。たとえば，**DBConnections.conf** ファイルには複数のデータベースおよびスキーマに対する複数の接続文字列が定義されていて，各接続文字列に名前が付いています。**MyApp.conf** ファイルではこれらの接続名の 1 つが使用され，それが特定のデータベースおよびスキーマを MyApp が使用するという定義になっています。

依存関係がある構成ドキュメントでは，ローカル・スコープ変数が使用されます。ローカル・スコープ変数の値はいずれかの構成ドキュメントに（挿入文を使用して）挿入され，その変数値（たとえば接続文字列の名前）がそのドキュメントに存在するようになります。さらにその変数は，別の構成ドキュメントの条件でも使用できます（たとえば，**MyApp.conf** ファイル内の接続名を使用して，MyApp がプロバイダ（DB）を使用できるようにする）。詳細については，「ローカル・スコープ」(73ページ)を参照してください。

変数に値を挿入するには，変数挿入文を使用します。ドキュメントのタイプごとに異なる文が用意されています。

- プロパティ・ファイル – KeyVariable および KeyRegExpVariable。詳細については，「プロパティ構成ドキュメント」(88ページ)を参照してください。
- XML ファイル – XPathVariable。詳細については，「XML 構成ドキュメント」(92ページ)を参照してください。
- テキスト・ファイル – RegExpVariable。詳細については，「テキスト構成ドキュメント」(95ページ)を参照してください。

この場合，必要な検索式は 2 つ（各ファイルに 1 つずつ）あります。つまり，<Dependency> タグの構造は次のようになります。

```
<Dependency name="dependency_name" providerCiType="webmodule" scope="my_scope">
    <VariableDeclarations>
      <Variable name=” REFERENCE_NAME” />
    </VariableDeclarations>
  <PropertiesConfigurationDocument name="DBConnections.conf">
    <Condition>
      <Operator type="and">
        <KeyCondition key="C1">
```

```
        <Values>
          <Value>${VAR_1}</Value>
        </Values>
      </KeyCondition>
    </Operator>
  </Condition>
  <Variables>
    <KeyVariable variable=" REFERENCE_NAME" key=" reference_name" />
  </Variables>
 </PropertiesConfigurationDocument>
 <TextConfigurationDocument name="MyApp.conf">
  <Condition>
    <Operator type="and">
      <RegExpCondition>
        <RegExp>C2?${REFERENCE_NAME}</RegExp>
      </RegExpCondition>
    </Operator>
  </Condition>
 </TextConfigurationDocument>
</Dependency>
```

注:

- <VariableDeclarations> 要素を使用して，依存関係で使用される 1 つまたは複数のローカル変数を定義します。
- <Variables> 要素は，ローカル変数に値を挿入するために使用します。このセクションは，<Condition> 要素で True が返された場合にのみ実行されます。
- この例では，<KeyVariable> を使用して “reference_name” キーの値を REFERENCE_NAME 変数に挿入します。
- ${REFERENCE_NAME} 変数は，**MyApp.conf** の条件で使用されます。この変数は **DBConnections.conf** ファイルに依存し，この変数が評価されるのは **DBConnections.conf** の後で，かつ検索式が True と評価された場合のみです。
- ファイル間の循環依存関係は許可されません。
- **MyApp.conf** ドキュメントが評価されるとき，${REFERENCE_NAME} 変数にはすでに **DBConnections.conf** からの値が割り当てられています。
- 接続文字列が複数のファイルに存在するため，スコープの検索式に十分な余裕を持たせて，依存関係のないファイルがそのスコープから返されるようにする必要があります。詳細については，「検索のスコープを指定する」(97ページ)を参照してください。

依存関係がある複数のファイルを検索するときの流れをまとめると次のようになります。

1. 「検索のスコープを指定する」(97ページ)の説明に従ってスコープのフィルタリングを実行します。必要なファイルが 1 つも返されない場合，またはほかのファイルに依存するファイルがない場合，利用者とプロバイダの間に依存関係は存在しません。

2. フィルタの結果，依存関係を持たないファイルが少なくとも 1 つ返されます。
3. そのファイルに接続されている利用者と必要な構成ドキュメントの残りの部分が Data Flow Probe にダウンロードされます。

挿入文が参照するキーが存在しない場合など，挿入文の値が返されないことがあります。この場合，変数には空の値が代入されます。ただし，これは予期された動作でない場合があります。必要な値が存在しない場合，依存関係が存在しないことを示唆すると考えられ，後続のファイルの評価を続けても意味がありません。こうした場合に対応するには，挿入文で allowNull=”false” を使用します。次に例を示します。

```
<KeyVariable variable=”REFERENCE_NAME” key=”reference_name” allowNull=”false” />
```

プロパティ構成ドキュメント

プロパティ構成ドキュメントは，Key=value 形式で構成を保存します。次に示すのはプロパティ構成ドキュメントの例です。

```
# My configuration document
Hostname=MyNode
```

ここでは，Hostname がキーで，MyNode がキーに関連付けられた値です。

“#” はコメント行であることを表します。コメント行は，検索式をテストするときは無視されます。

構成ドキュメントがプロパティ構成ドキュメントであることを宣言するには，<PropertiesConfigurationDocument> 要素を使用します。次に例を示します。

```
<Dependency name="history_db" providerCiType="cmdb" scope="default">
  <PropertiesConfigurationDocument name="cmdb.conf">
    ...
```

条件の定義

プロパティ・ファイルに接続文字列変数が存在するかどうかをテストする方法はいくつかあります。すべてのタイプの条件で，**key** 属性で指定されたキーの値が一部の値に対してテストされます。キーは常に定数値です（変数は指定できません）。

- 特定の定数値がキーの値として存在するかどうかテストします。

```
<KeyCondition key="dal.datamodel.name">
  <Values>
    <Value>ConstantValue1</Value>
    <Value>ConstantValue2</Value>
  </Values>
```

```
</KeyCondition>
```

条件が True として返されるのは，キーの値がいずれかの定数値と等しい（大文字と小文字も一致する）場合のみです。

- 特定の変数値がキーの値として存在するかどうかテストします。

```
<KeyCondition key="dal.datamodel.name">
  <Values>
    <Value>${VARIABLE1}</Value>
    <Value>${VARIABLE2}</Value>
  </Values>
</KeyCondition>
```

条件が True として返されるのは，キーの値がいずれかの変数値と等しい（大文字と小文字も一致する）場合のみです。変数値が値のリストの場合はすべての値がテストされ，リストの値のうちいずれか 1 つだけでもキーと一致すれば True が返されます。

注: 同じ条件に変数値と定数値の両方を指定できます。

- キーの値が一定の正規表現に適合するかどうかテストします。

```
<KeyCondition key="dal.datamodel.name">
  <RegExp>
    ^SomeText${VARIABLE}MoreText$
  </RegExp>
</KeyCondition>
```

正規表現には，条件の一部として 1 つ以上の変数を含めることもできます。実行中，具体的な検索式が生成されるときに，変数が実際の値と置き換えられます。

変数に値のリストが入る場合は，次のような式が生成されます。

```
^SomeText(Value1 | Value2  | Value3)MoreText$
```

この場合は，どの値が条件に一致しても True が返されます。

プロパティ構成ドキュメントで定義される条件の例

```
<PropertiesConfigurationDocument name="MyConfig.properties">
  <Condition>
    <Operator type="and">
      <KeyCondition key="TYPE">
        <Values>
          <Value>HTTP</Value>
          <Value>HTTPS</Value>
```

```
      </Values>
    </KeyCondition>
    <KeyCondition key="IPADDRESS">
      <Values>
        <Value>${IP_ADDRESS}</Value>
        <Value>${HOSTNAME}</Value>
      </Values>
    </KeyCondition>
    <KeyCondition key="SITE">
      <RegExp>
        //${SITE_NAME}//.*
      </RegExo>
    </KeyCondition>
  </Operator>
 </Condition>
</PropertiesConfigurationDocument>
```

これは，**MyConfig.properties** というプロパティ構成ドキュメントでは次のすべての条件が満たされる必要があることを意味します。

- TYPE という名前のキーには，値「HTTP」または「HTTPS」のいずれかが入る。
- IPADDRESS という名前のキーには，プロバイダの IP アドレス（IP アドレスが複数ある場合はそのうちの 1 つ）またはそのホスト名のいずれかが入る。
- SITE という名前のキーは，先頭が「/」で，その後にプロバイダのサイト名，もう 1 つの「/」，さらに任意の文字列の順で構成される。

変数値の挿入

キーの値または値の一部を変数に抽出する場合，次の 2 つの方法があります。

- キーの値を挿入する
  - 専用の <Variables> セクションで，次の構文を使用します。

```
<KeyVariable variable=” VARIABLE_NAME” key=” KEY_NAME” />
```

  - 検索条件の一部として，次の構文を使用します。

```
<KeyCondition key=” KEY_NAME” >
  <Values>
    <Value>REQUIRED_VALUE</Value>
  </Values>
  <Variables>
    <KeyVariable variable=” VARIABLE_NAME” />
  </Variables>
</KeyCondition>
```

VARIABLE_NAME 変数のキーは，<KeyCondition> 要素で定義されます。

この構文は，<Values> の代わりに <RegExp> を使用する場合と同じです。どちらの場合でも，変数にはキーの値全体が挿入されます。

- キーの値の一部を挿入する
  - 専用の <Variables> セクションで，次の構文を使用します。

```
<KeyRegExpVariable variable=” VARIABLE_NAME” key=” KEY_NAME” expression=” REGULAR_
EXPRESSION” group=” GROUP_NUMBER” />
```

この場合，GROUP_NUMBER は，REGULAR_EXPRESSION で定義される正規表現に含まれるグループの（0 で始まる）インデックスです。たとえば，プロパティ構成ドキュメントに次の行があるとします。

```
myKey = 123Value123
```

また，次の文も含まれているとします。

```
<KeyRegExpVariable variable=” VAR” key=” myKey” expression=” [0-9]*([a-zA-Z]*)[0-9]*”
group=” 0” />
```

この文は，変数 “VAR” に値 “Value” を挿入します。

変数は，同一のファイルまたはほかのファイルで定義されたものを使用できます。次に例を示します。

```
<KeyRegExpVariable variable=” VAR” key=” myKey” expression=” ${PREV_VAR}([a-zA-Z]*)
${PREV_VAR}” group=” 0” />
```

PREV_VAR に値 “123” が入ると仮定すると，VAR の値は，キーの値全体を挿入する場合と同様に “Value” になります。どちらの方法でも，変数には常に 1 つの値が入ります。

  - 検索条件の一部として，次の構文を使用します。

```
<KeyCondition key=” KEY_NAME” >
  <RegExp>REGULAR_EXPRESSION</RegExp>
  <Variables>
    <KeyRegExpVariable variable=” VARIABLE_NAME” group=” GROUP_NUMBER” />
  </Variables>
</KeyCondition>
```

VARIABLE_NAME 変数のキーは，<KeyCondition> 要素で定義されます。

これは <RegExp> を使用する場合のみサポートされ，変数の “group” 属性が参照する正規表現は <RegExp> の正規表現と同じです。

XML 構成ドキュメント

XML 構成ドキュメントでは，XPath クエリを使用して，XML 標準に準拠した構成ドキュメントの検索式を簡単に記述することができます。

構成ドキュメントが XML 構成ドキュメントであることを宣言するには，<XmlConfigurationDocument> 要素を使用します。次に例を示します。

```
<Dependency name="some_reference" providerCiType="webmodule" scope="default">
  <XmlConfigurationDocument name="web.xml">
    ...
```

条件の定義

XML 構成ドキュメントで使用できる唯一の検索条件は，有効な XPath 2.0 クエリです。XPath からノードが選択された場合，条件は True として返され，それ以外の場合は False が返されます。クエリでは次の場所に変数を含めることができます。

- 要素名の中

  ```
  /Root/${SITE_NAME}
  ```

  変数に複数の値が入る場合，フレームワークでは複数の XPath 文が実行されます。たとえば，SITE_NAME の値が SITE1 と SITE2 の場合，この例の文では次に示す 2 つの具体的な検索が生成されます。

  ```
  \Root\SITE1
  \Root\SITE2
  ```

- 相等性のテストの中

  ```
  /Element[@att = ${VAR}]
  または
  /Element/text() = ${VAR}
  ```

  上記の例で示されているように，等号は 1 つの値が入る変数に対してのみ使用できます。変数に複数の値が入る可能性があるときは，代わりに次の構文を使用します。

  ```
  /Element[${equals(@att, VAR)]
  または
  /Element[${equals-ignore-case(text(), VAR}]
  ```

注:

- 大文字と小文字を区別して相等性をテストする場合は ${equals()} を，大文字と小文字を区別せずに相等性をテストする場合は ${equals-ignore-case()} を使用します。
- 1 つ目のパラメータは，XPath 関数または属性名にします。
- 2 つ目のパラメータは，必ず変数名にします。この変数名は，${} を付けずに記述します。
- この構文は，1 つの値が入る変数にも複数の値が入る変数にも有効であり，すべての場合で使用することをお勧めします。
- 複数の値が存在し，1 つ目のパラメータの値が変数のいずれか 1 つの値と等しい場合，関数は True として返されます。

- **正規表現の中**

```
/datasources/mbean[matches(@name, '.*?,ip=${IPADDRESS}.*')]
```

変数に値のリストが入る場合は，次のような式が生成されます。

```
/datasources/mbean[matches(@name, '.*?,ip=(Value1 | Value2 | Value3).*')]
```

どの値が条件に一致しても True が返されます。

XML 構成ドキュメントで定義される条件の例

```
<XmlConfigurationDocument name="MyConfig.xml">
  <Condition>
    <Operator type="and">
      <XPathCondition>
        <XPath>\URL\Protocol[@name=’ HTTP’ or @name=’ HTTPS’ ]</XPath>
      </XPathCondition>
      <XPathCondition>
        <XPath>\URL\Host[${equals-ignore-case(@name, HOSTNAME)} or ${equals(@name, IP_
ADDRESS)}]</XPath>
      </XPathCondition >
      <XPathCondition>
        <XPath>\URL\Site[matches(text(), ‘//${SITE_NAME}//.*’ )]</XPath>
      </XPathCondition>
    </Operator>
  </Condition>
</XmlConfigurationDocument>
```

この例は，**MyConfig.xml** という XML 構成ドキュメントでは次のすべての条件が満たされる必要があることを示しています。

- \URL\Protocol\@name には，値「HTTP」または「HTTPS」のいずれかが入る。
- \URL\Host\@name には，プロバイダの IP アドレス（IP アドレスが複数ある場合はそのうちの 1 つ）またはそのホスト名のいずれかが入る。
- \URL\Site\text() には，先頭が「/」で，その後にプロバイダのサイト名，もう 1 つの「/」，さらに任意の文字列の順で構成されるテキストが入る。

変数値の挿入

XPath を使用してテキスト値のクエリを実行し，その値を変数に挿入することが可能です。変数に要素全体を挿入することはできません。これを行うには，専用の <Variables> セクションで，次の構文を使用します。

```
<XPathVariable variable=”VARIABLE_NAME” xpath=”XPATH_QUERY” />
```

次にいくつか例を挙げます。

- 属性値を選択する

  ```
  <XPathVariable variable=”VAR” xpath=”\Root\Element\@Att” />
  ```

  この場合，XML ドキュメント内のすべての \Root\Element 要素の **Att** 属性から値が選択されます。
- 要素のテキストを選択する

  ```
  <XPathVariable variable=”VAR” xpath=”\Root\Element\text()” />
  ```

  この場合，\Root\Element と一致するすべての要素のテキストが選択されます。
- 条件を指定して属性値を選択する（同じ構成ドキュメントまたは別の構成ドキュメントの変数を使用）

  ```
  <XPathVariable variable=”VAR” xpath=”\Root\Element[${equals(@Att, VAR)}]\@AnotherAtt” />
  ```

  この場合，@Att=${VAR} という条件に該当する，すべての \Root\Element の @AnotherAtt の値が選択されます。

XPath 変数は，XPath 条件の一部として使用できます。このモードでは，条件に該当するノードがあった（条件が True と評価された）ものと仮定して，そのノードからの相対パスを含めることもできます。次の構文を使います。

```
<XPathCondition>
  <XPath>XPATH_QUERY</XPath>
  <Variables>
```

```
    <XPathVariable variable="VARIABLE_NAME" relativePath="RELATIVE_XPATH_QUERY" />
  </Variables>
</XPathCondition>
```

次に例を示します。

```
<XPathCondition>
  <XPath>/${VAR_1}/chcpCodeToCharsetName[@name='test1' and matches(text(), '.*?${OUTPUT_
VAR2}.*')]</XPath>
  <Variables>
    <XPathVariable variable="OUTPUT_VAR1" relativePath="./@type" />
  </Variables>
</XPathCondition>
```

XPathCondition が True と評価されたものと仮定し，ある XML ノード（特にこの場合はある要素）が <XPath> によって選択されると，変数にはそのノードの “type” 属性値が入ることになります。

“/” で始まるドキュメントのルートから XPath を指定することにより，その変数から独立している特定の変数の XPath を定義できます。

XPath を挿入した後，その変数には値のリストが入る可能性があります。

テキスト構成ドキュメント

テキスト構成ドキュメントは，コンテンツ開発者にとって馴染みのある形式で記述される，プロパティ構成ドキュメントでも XML ドキュメントでもないテキスト・ドキュメントです。テキスト構成ドキュメントでは，正規表現を使用して検索式を記述できます。

構成ドキュメントがテキスト・ファイル・タイプであることを宣言するには，<TextConfigurationDocument> 要素を使用します。次に例を示します。

```
<Dependency name="some_reference" providerCiType="oracle" scope="default">
  <TextConfigurationDocument name="tnsnames.ora">
    ...
```

条件の定義

テキスト構成ドキュメントで使用できる唯一の検索条件は，正規表現です。パターンに一致すれば，条件は True として返されます。

正規表現には，条件の一部として 1 つ以上の変数を含めることができます。実行中，具体的な検索式が生成されるときに，変数が実際の値と置き換えられます。

変数に値のリストが入る場合は，次のような式が生成されます。

```
^SomeText(Value1 | Value2 | Value3)MoreText$
```

どの値が条件に一致しても True が返されます。

テキスト構成ドキュメントで定義される条件の例

```
<TextConfigurationDocument name="MyConfig.txt">
  <Condition>
    <Operator type="or">
      <RegExpCondition>
        <RegExp>^HTTPS?://(${HOSTNAME)/${SITE_NAME}/.*$</RegExp>
      </RegExpCondition>
      <RegExpCondition>
        <RegExp>^HTTPS?://(${IP_ADDRESS)/${SITE_NAME}/.*$</RegExp>
      </RegExpCondition>
    </Operator>
  </Condition>
</TextConfigurationDocument>
```

この例は，**MyConfig.txt** というテキスト構成ドキュメントでは次のすべての条件が満たされる必要があることを示しています。

- \URL\Protocol\@name には，値「HTTP」または「HTTPS」のいずれかが入る。
- \URL\Host\@name には，プロバイダの IP アドレス（IP アドレスが複数ある場合はそのうちの 1 つ）またはそのホスト名のいずれかが入る。
- \URL\Site\text() には，先頭が「/」で，その後にプロバイダのサイト名，もう 1 つの「/」，さらに任意の文字列の順で構成されるテキストが入る。

変数値の挿入

正規表現をグループとして使用し，複数のテキスト・ドキュメントの変数に値を挿入することができます。グループによって，挿入されるテキストがマークされます。専用の <Variables> セクションで，次の構文を使用します。

```
<RegExpVariable variable=” VARIABLE_NAME” expression=” REGULAR_EXPRESSION”
group=” GROUP_NUMBER” />
```

この場合，GROUP_NUMBER は，REGULAR_EXPRESSION で定義される正規表現に含まれるグループの（0 で始まる）インデックスです。

たとえば，次の行を含むファイルがあるとします。

```
This is part of a configuration document
```

また，次の文も含まれているとします。

```
<RegExpVariable variable=” VAR” expression=” (.*)configuration (.*)” group=” 1” />
```

この文は，変数 “VAR” に値 “document” を挿入します。別のドキュメントで定義された変数を同じドキュメント内で使用することもできます。次に例を示します。

```
<RegExpVariable variable=” VAR” expression=” .*${PREV_VAR} (.*)” group=” 0” />
```

PREV_VAR の値が “configuration” の場合，VAR には文字列 “document” が挿入されます。

次の構文を使用すると，<RegExpVariable> を <RegExpCondition> タグの一部として使用できます。

```
<RegExpCondition>
  <RegExp>REGULAR_EXPRESSION</RegExp>
  <Variables>
    <RegExpVariable variable="VARIABLE_NAME" group="GROUP_NUMBER" />
  </Variables>
</RegExpCondition>
```

変数の “group” 属性が参照する正規表現は，<RegExp> の正規表現と同じです。

挿入先の変数には常に 1 つの値が挿入されます。

検索のスコープを指定する

検索のスコープを指定する目的は，特定のプロバイダ・タイプと依存関係を持つ可能性があるすべての利用者を見つけることです。スコープには 2 種類の用途があります。

- 検索結果の数を少なくするために，関係のないデプロイメント可能コンポーネントを UCMDB から除外する。（必須）

  これを行うには，すべての構成ドキュメントでプロバイダの接続文字列のサブセットを見つけ，該当する構成ドキュメントに接続されているデプロイメント可能コンポーネントのみを検索対象として限定するようにします。

  この検索式は，プロバイダ固有の依存関係検索とは異なります。つまり，この検索式では可能性のあるすべての利用者をできる限り速やかに返す必要があり，利用者とプロバイダの依存関係が存在するかどうかについて正確な結果を返すことが目的ではありません。

  この検索式の構文は，構成ドキュメントの検索式を記述する場合と同様の構成ですが，ファイル・タイプごとの特定の条件はなく，ファイルの名前を指定する必要もありません。詳細については，「検索式の構成」(78ページ)を参照してください。

  この例は，次の式を表しています。(x | ((y & z & w) & (h | g))).

  ```
  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <ConfigurationDocumentSearchCondition xmlns="http://www.hp.com/ucmdb/1-0-
  0/DependenciesDefaultSearch">
     <Operator type="Or">
        <Operator type="And">
           <Operand value="y"/>
  ```

```
        <Operand value="z"/>
        <Operand value="w"/>
        <Operator type="Or">
          <Operand value="h"/>
          <Operand value="g"/>
        </Operator>
      </Operator>
      <Operand value="x"/>
    </Operator>
</ConfigurationDocumentSearchCondition>
```

x，y，z，w，h，g は，定数値，変数，またはコンセプト変数です。

これらの値（または変数値）は，UCMDB 内の該当するすべての利用者の構成ドキュメントについてこの式を使用して検索されます。

該当するファイルを持つ利用者のみを対象に，正確な検索式を使用してプロバイダ固有の検索が続けて実行され，評価されます。

- デプロイメント可能コンポーネントをプロバイダの接続コンポーネントのサブセットに制限する（プロバイダの J2EE ドメインの一部である依存関係のみを検索する場合など）。（任意指定）

  これを行うには，TQL クエリを使用して，特定のプロバイダ・タイプと何らかの形で接続されているすべてのデプロイメント可能コンポーネントを見つけます。詳細については，「TQL クエリを定義する」(100ページ)を参照してください。

  （特定のプロバイダ CI を指定した）TQL クエリの結果によって，そのプロバイダとの間に依存関係が存在する可能性があるすべてのデプロイメント可能コンポーネントが特定されます。この TQL クエリは次のルールに従うことが必要です。

  - デプロイメント可能コンポーネントに対するクエリ・ノードの名前は「Deployable」（大文字と小文字を区別）にします。このクエリ・ノードの結果が，スコープに含まれる可能性のあるデプロイメント可能コンポーネントです。
  - “Deployable” コンポーネントは，同じスコープに含まれる可能性がある利用者とプロバイダの両方のコンポーネントを表します。したがって，クエリ・ノードの CI タイプは，利用者とプロバイダのデプロイメント可能 CI タイプの祖先であることが必要です。
  - TQL クエリに含めるクエリ・ノードは，（“Deployable” 以外に）1 つのみにします。このもう 1 つのクエリ・ノードの名前については制限はありません。このノードのことを Scope クエリ・ノードと言います。
  - Deployable クエリ・ノードと Scope クエリ・ノードは，この 2 つのノード間のすべての利用可能なパスを含む複合リンクで接続されます。

- どのスコープも，（プロバイダが特定のルーティング・ドメインで検出されたという仮定の下で）プロバイダと同じルーティング・ドメインに含まれない利用者のデプロイメント可能コンポーネントを黙示的に除外します。ルーティング・ドメインでフィルタリングするには，デプロイメント可能コンポーネントの記述子にノードへのパスを含める必要があります。詳細について

は，「利用者の記述子を定義する」(76ページ)を参照してください。

たとえば，プロバイダのデプロイメント可能コンポーネントが，あるタイプの実行中のソフトウェアの場合，その実行中ソフトウェアのルーティング・ドメインはソフトウェア自体が含まれるノードのルーティング・ドメインと同じになります。ノードのルーティング・ドメインは，そのIPアドレスのルーティング・ドメインによって決まります。異なるルーティング・ドメインで構成されるIPアドレスの場合，ノードには複数のルーティング・ドメインが関連付けられている可能性があります。一方，プロバイダのデプロイメント可能コンポーネントが，どのノードまたはIPアドレスとも関連付けられていないCI（ビジネス・サービスなど）である場合は，そのルーティング・ドメインにかかわらず，任意のデプロイメント可能コンポーネントに接続できます。

たとえば，J2EEドメインのスコープを定めるには，次の手順を実行します。

1. タイプが “J2EE Deployed Object” のDeployableクエリ・ノードを作成します。これは，J2EEドメインのデプロイメント可能コンポーネントに相当するすべてのCIタイプの中で最小限の共通点を持つクラスです。たとえば，このスコープに含まれるデプロイメント可能コンポーネントとしては，Webモジュール・アプリケーションやJ2EEアプリケーションが考えられます。
2. タイプが “J2EE Domain” のScopeクエリ・ノードを作成します。
3. J2EE Deployed ObjectとJ2EE Domain間のすべてのパスを使用して，Dependencyクエリ・ノードとScopeクエリ・ノード間の複合リンクを作成します。たとえば，この複合リンクの場合は次の要素が含まれます。
   - J2EE Application – Composition – J2EE Domain
   - Web Module – Composition – J2EE Application

TQLクエリ定義は，スコープのXMLに埋め込む必要があります。

スコープ検索で使用する変数の標準設定値

次の例を使ってみましょう。

- 標準設定値が80のPORT変数を使用します。この標準設定値は，構成ファイルによっては実際には記述されていない可能性があります。
- スコープ検索式としてIP_ADDRESS AND PORTを使用し，特定のプロバイダのPORT変数には80が値として入ります。ポートの標準設定値「80」が含まれていない構成ドキュメントに接続しているデプロイメント可能コンポーネントは，このスコープの検索では本来は返されるはずですが実際には返されません。その理由は，検索式では構成ファイル内にこのポート番号が存在することを条件にしているためです。

そのため，PORT変数に標準設定値が代入されるときは，標準設定値はスコープの検索式から削除されます（依存関係の検索式で無視されるのと同様です）。詳細については，「変数の標準設定値の使用」(80ページ)を参照してください。

変数に標準設定値以外の値が入る場合は，この値は通常の方法で検索式に挿入されます。

トラブルシューティング

スコープから予想どおりの利用者が返されるかどうかをテストするには，UCMDBサーバのディスカ

バリ・マネージャ・サービスで **searchConfigurationDocuments** JMX メソッドを使用します。“searchString” パラメータは，「検索のスコープを指定する」(97ページ)で説明した検索式の XML であることが必要です。この XML に含めることができるのは定数値のみで，変数は使用できません。

この XML は，いずれかの検索アダプタのプロバイダの通信ログから取得することもできます。詳細については，「依存関係検索アダプタ」(103ページ)を参照してください。

TQL クエリを定義する

1. <Queries> セクションで，次のように依存関係シグネチャに TQL クエリ定義を追加します。

```
...
<Queries>
  <Query name=” YourQueryName” >
    [TQL XML]
  </Query>
</Queries>
```

2. a. テキスト・エディタを使用して空のドキュメントを作成します。
   b. 次のコンテンツを追加します。

```
<tql:query xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3=
"http://www.hp.com/ucmdb/1-0-0/PolicyRuleDefinition" xmlns:tql=
"http://www.hp.com/ucmdb/1-0-0/TopologyQueryLanguage" name="<Query Name>">

</tql:query>
```

   c. モデリング・スタジオで TQL クエリを作成します。詳細については，『HP Universal CMDB モデリング・ガイド』の「モデリング・スタジオ」を参照してください。
   d. TQL クエリを XML にエクスポートします。
   e. エクスポートした XML ドキュメントを開いて，<resource> 要素内のすべてのテキストをコピーします。
   f. このテキストを，手順 1 で作成したドキュメントの <tql:query> 要素に貼り付けます。
   g. このテキスト・ドキュメントのコンテンツ全体をコピーして，依存関係シグネチャ・ファイルの <Query> 要素に貼り付けます。

注: TQL クエリは依存関係シグネチャ・ファイルに埋め込まれ，評価されることはありません。ただし，クエリの XML ファイル自体が無効な場合は，デプロイメント時に例外がスローされます。詳細については，「コンパイル・エラー」(102ページ)を参照してください。

複数の依存関係シグネチャ・ファイルのパッケージングとデプロイ

UCMDB では，複数の依存関係シグネチャ・ファイルをデプロイできます。利用者 - プロバイダのリンクを検索する場合，デプロイされているファイルのすべてが使用されます。

依存関係シグネチャ・ファイルは，**dependencies/** というプレフィックスおよび **.xml** というサフィックスを持つディスカバリ構成ファイルです（例 : **dependencies/JEE.xml**）。

各依存関係シグネチャ・ファイルは完全に独立したファイルです。次の点に注意してください。

- グローバル変数定義は，それ自体が定義されているファイルのみで使用できます。同様の変数に対して同一の名前を可能な限り使用することを強くお勧めします。たとえば，変数に IP アドレスが含まれ，2 つの依存関係ファイルが存在する場合は，両方のファイルで，IP_ADDRESS という名前の変数を定義します。この方法により，異なる検索アダプタが IP_ADDRESS という名前の単一の対象データを使用できるようになります。詳細については，「変数値の指定」(106ページ)を参照してください。
- 概念定義は，それ自体が定義されているファイルのみで使用できます。同様の目的を持つ概念に対して同一の概念名および概念変数名を可能な限り使用することを強くお勧めします。詳細については，「概念変数値の指定」(107ページ)を参照してください。
- デプロイ可能なコンポーネントは，異なるファイルで定義されている場合でも同一の名前を持つことができます。異なるデプロイ可能なコンポーネントとして扱われます。
- TQL クエリは，異なるファイルで定義されている場合でも同一の名前を持つことができますが，異なる TQL クエリとして扱われます。依存関係シグネチャ・ファイルで名前により TQL クエリを参照する場合，その名前を持つ TQL クエリが同一ファイル内に存在する必要があります。
- スコープは，異なるファイルで定義されている場合でも同一の名前を持つことができます。異なるスコープとして扱われます。依存関係シグネチャ・ファイルで名前によりスコープを参照する場合，その名前を持つスコープが同一ファイル内に存在する必要があります。
- 条件関数は，異なるファイルで定義されている場合でも同一の名前を持つことができます。依存関係シグネチャ・ファイルで名前によりスコープを参照する場合，その名前を持つ関数が同一ファイル内に存在する必要があります。
- 変数や概念変数の標準設定値は，それ自体が定義されているファイルに固有の値です。同じ名前を持つ 2 つの変数または概念変数は，異なるファイルで異なる標準設定値を持つことができます。
- 異なる依存関係シグネチャ・ファイルで同一の概念に対して異なるキー・プロパティを設定することはお勧めしません。維持が困難で，異なるアダプタからの値をその対象データ変数に正しく割り当てることが難しくなります。詳細については，「概念変数値の指定」(107ページ)を参照してください。

コンパイル・エラー

コンパイル時の検証

- 同一ファイル内に同じ名前のデプロイメント可能コンポーネントが 2 つ以上存在する。
- 同一ファイル内に同じ名前のスコープが 2 つ以上存在する。
- 同一ファイル内に同じ名前の TQL クエリが 2 つ以上存在する。
- 同一ファイル内に同じ名前の条件関数が 2 つ以上存在する。
- 同一のデプロイメント可能コンポーネントに同じ名前の依存関係が 2 つ存在する。
- 構成ファイルで使用されている変数名が，同一ファイル内でグローバル変数として宣言されていない，またはその構成ファイルが属する依存関係内でローカル変数として宣言されていない。
- 条件関数で使用されている変数名が，その関数のパラメータまたはグローバル変数として宣言されていない。
- 変数間の循環依存関係が存在する（たとえば，変数 VAR_A の挿入文に VAR_B の値が使用され，VAR_B の挿入値に変数 VAR_A の値が使用されている）。
- 同じファイル内に存在しない TQL クエリを参照している。
- 同じファイル内に存在しないスコープを参照している。
- 同じファイル内に存在しない条件関数を参照している。
- ignore 文に標準設定値が指定されていない変数を使用することにより，コンパイル・エラーが発生する。詳細については，「変数の標準設定値の使用」(80ページ)を参照してください。
- 標準設定値が指定された複数の変数が検索式に含まれる場合，それらの変数の任意の組み合わせにその標準設定値が代入されるすべてのケースに対処する必要がある。これは，一部の変数については検索式を完全に無視する，または代替式を使用することで対応できます。ignore 文に記述されない変数のすべての組み合わせに対して代替式を準備する必要があります。そうしないと，コンパイル・エラーが発生することになります。詳細については，「変数の標準設定値の使用」(80ページ)を参照してください。
- TQL クエリが <DocumentCILocation> タグ内で使用されているが，次のルールに従っていない。
  - TQL クエリでは，デプロイメント可能コンポーネントと構成ドキュメント間のパスを定義します。このパスは単純な（循環を含まない）パスにする必要があります。
  - TQL クエリには，大文字と小文字を区別した固有の名前を持つ，次の 2 種類のエンド・ノードを追加します。
    - Deployable – パス内のデプロイメント可能コンポーネント CI
    - Configuration_document – パス内の構成ドキュメントの CI
  - Deployable ノードと Configuration_document ノードに対して条件を設定することはできません。
  - すべてのノード間のカーディナリティは必ず 1..1 にします。

  - 通常のリンク・タイプのみサポートされます。複合リンク，結合リンク，サブグラフは使用できません。

- TQL クエリがスコープ内で使用されているが，次のルールに従っていない。
  - デプロイメント可能コンポーネントに対するクエリ・ノードの名前は「Deployable」（大文字と小文字を区別）にします。このクエリ・ノードの結果が，スコープに含まれる可能性のあるデプロイメント可能コンポーネントです。
  - TQL クエリに含めるクエリ・ノードは，（“Deployable” 以外に）1 つのみにします。このもう 1 つのクエリ・ノードの名前については制限はありません。このノードのことを Scope クエリ・ノードと言います。
  - Deployable クエリ・ノードと Scope クエリ・ノードは，この 2 つのノード間のすべての利用可能なパスを含む複合リンクで接続されます。
  - “Deployable” コンポーネントは，同じスコープに含まれる可能性がある利用者とプロバイダの両方のコンポーネントを表します。したがって，クエリ・ノードの CI タイプは，利用者とプロバイダのデプロイメント可能 CI タイプの祖先であることが必要です。
- テキスト構成ドキュメントに，優先度が異なる複数の <ReferenceLocation> タグが含まれている。詳細については，「構成ドキュメントの上書き」(83ページ)を参照してください。

依存関係検索アダプタ

依存関係検索フレームワークは，専用のアダプタを使用して実行する必要があります。アダプタは次のことを行います。

- 特定のプロバイダに関連するすべての接続文字列を収集する。
- 依存関係検索フレームワークを実行する。
- 検索の結果（依存関係）を UCMDB にレポートする。

各アダプタが処理するのは 1 つのタイプのプロバイダです。たとえば，あるアダプタは JAR プロバイダを処理できます。

複数のアダプタが同じプロバイダ・タイプを処理するようにすることも可能です。ただしその場合は，矛盾のない結果が得られるように，どのプロバイダも必ずアダプタごとに 1 回だけ実行されるようにします。

ほかのアダプタと同様，依存関係検索アダプタの準備ができたら，アダプタ・ロジックを実行するためにディスカバリ・ジョブを作成する必要があります。

本項の内容

依存関係検索アダプタを作成する

1. **［データ フロー管理］ > ［アダプタ管理］**を選択します。
2. ［**新規作成**］  をクリックして，［**新規アダプタ**］を選択します。
3. アダプタの詳細を入力し，［**OK**］をクリックします。
4. ［リソース］表示枠で，作成したアダプタを右クリックして［**アダプタ ソースを編集**］を選択します。
5. 次の行を見つけて，

```
<taskInfo className="com.hp.ucmdb.discovery.probe.services.dynamic.core.DynamicService">
```

次の要素で置き換えます。

```
<taskInfo className="com.hp.ucmdb.discovery.probe.services.dynamic.core.WorkflowService">
```

6. 次の行を見つけて，

```
<params
className="com.hp.ucmdb.discovery.probe.services.dynamic.core.DynamicServiceParams"
ignoreMissingReconciliationRules="false" enableRecording="false" enableAging="true"
useDefaultValueForAging="false" autoDeleteOnErrors="success" recordResult="false" />
```

次の要素で置き換えます。

```
<params className="com.hp.ucmdb.discovery.probe.services.dynamic.core.WorkflowServiceParams"
patternType="workflow_adapter" enableAging="true" ignoreMissingReconciliationRules="false"
enableRecording="false" autoDeleteOnErrors="success" recordResult="false"
maxThreadRuntime="86400000" useDefaultValueForAging="false">
  <workflow>
    <steps>
      <step name="Dependencies Discvoery" failure-policy="mandatory">
        <module
type="java">com.hp.ucmdb.discovery.probe.agents.probemgr.accuratedependencies.processing.Dep
endenciesDiscoveryWorkflowStep</module>
        <timeoutParking>
          <initialTimeout>180000</initialTimeout>
          <retriesThreshold>12</retriesThreshold>
          <multipleBy>2</multipleBy>
          <maxRetry>10</maxRetry>
          <timeoutThreshold>10800000</timeoutThreshold>
        </timeoutParking>
      </step>
```

```
    </steps>
    <finalStep />
    <libraryScripts />
  </workflow>
</params>
```

7. 依存関係シグネチャ検索の結果を処理するスクリプトを作成します。詳細については，「Jython スクリプトの記述」(110ページ)を参照してください。
8. ワークフロー・アダプタに次のステップを追加します。

```
<step name="Default Search Result Awaiting" failure-policy="mandatory">
  <module type="jython">[Your Jython Script Name]</module>
  <noParking />
</step>
```

9. ［入力］表示枠で，［**CI タイプの選択**］  をクリックし，このアダプタのプロバイダ CI タイプを選択します。
10. ［**入力クエリの編集**］  をクリックして入力 TQL クエリを作成します。詳細については，「入力 TQL クエリおよび宛先データを定義する」(106ページ)を参照してください。
11. ［検出された CIT］表示枠で  をクリックし，プロバイダのタイプ，対象となり得るすべての利用者のタイプ，プロバイダと利用者のリンクのタイプを追加します。
12. 完了したら，［**保存**］ をクリックします。

利用者 / プロバイダ・アダプタを定義する

検索機能はディスカバリ・アダプタにより実行されます。各アダプタは，プロバイダ・タイプの特定の接続文字列セットの検索を処理します。アダプタの入力 TQL クエリは，その接続文字列が検出されるすべての CI を取得し，トリガの宛先データを使用して接続文字列の変数とコンセプトに値を挿入します。詳細については，「Jython アダプタの開発」(37ページ)を参照してください。

このアダプタは「ワークフロー・アダプタ」タイプに属し，いくつかの手順を実行して，利用者とプロバイダの関係を検出するための検索を行います。

1. 接続文字列の値を依存関係シグネチャのグローバル変数に挿入し，コンセプトをインスタンス化することによって，スコープの構成ドキュメント・フィルタに対する具体的な検索式となるロジックを実行します。
2. スコープの具体的な検索式を，UCMDB サーバを実行している UCMDB の Solr エンジンに送信します。
3. 依存関係シグネチャ検索を実行できるようにするため，必要なすべての情報（構成ドキュメント，デプロイメント可能コンポーネントの記述子など）を取得する目的で UCMDB に対して TQL クエリを実行します。
4. 必要な構成ドキュメントのコンテンツをダウンロードして Data Flow Probe に保存します。

5. 手順 2 と 3 のスコープで検出されたプロバイダと利用者に関連する依存関係検索を実行します。
6. Jython スクリプトによって，依存関係検索の結果がレポートされます。

入力 TQL クエリおよび宛先データを定義する

依存関係シグネチャのスコープや条件から検索式を作成するには，その検索式で宣言される変数やコンセプトを，それぞれ（TQL クエリによって返される）具体的な接続文字列に置き換える必要があります。こうした置き換えは，開発者が定義するアダプタ固有のマッピングに基づいて実行されます。

依存関係マッピング・アダプタの入力 TQL クエリでは必ず次のことを定義します。

- プロバイダの利用者を見つけるためのトリガ CI（ソース・クエリ・ノード）
- 特定のプロバイダ用に必要なすべての接続文字列

この TQL クエリを実行することで，CI のトポロジに散在するすべての接続文字列が返され，サービスを提供しているプロバイダが特定されます。たとえばプロバイダが Oracle RAC の場合，TQL クエリは，利用者がその RAC に接続してサービスを利用するために必要な接続文字列が含まれるすべての CI を返す必要があります。そのため，TQL クエリのレイアウト定義では，該当する接続文字列を格納する属性をすべて記述することが必要です。たとえば，RAC に対する TQL クエリの場合は，その RAC の各インスタンスがアクセス可能な IP アドレスとポート，および RAC サービス名が返されるはずです。

変数とコンセプトは，アダプタの宛先データ定義を使用して，入力 TQL クエリ内のクエリ・ノードの属性にマップされる必要があります。正しくマッピングできるように，各パターン要素には一意の名前を付けます。

アダプタに関しては，検索アダプタの入力 TQL クエリは，そのアダプタと関連付けられているジョブ用のトリガ TQL クエリと一致するトリガに対して実行されます。

サービス検出用アダプタを使用する際，場合によっては，サーバで（自動的に）検出されたトリガだけでなく，（何らかのパスにより）ビジネス・サービス CI に接続されているトリガについても対象にすることが必要となります。そのためには，トリガとビジネス・サービス CI 間のパス（通常は複合リンク）が含まれるように入力 TQL クエリを定義して，ビジネス・サービス CI のクエリ・ノードに「SERVICE」という名前を付けます（大文字と小文字を区別します）。そうすると，フレームワークで「SERVICE」というクエリ・ノードが検出されたときに，そのトリガ CI が属するサービスにのみ自動で結果が制限されるようになります。

変数値の指定

各検索アダプタは，アダプタが検出すべき依存関係について，依存関係シグネチャの検索条件に表示されるすべてのグローバル変数および概念の値を設定する必要があります。これらの値は，プロバイダの接続文字列（詳細については，「変数とコンセプト」(72ページ)を参照）を表し，そのプロバイダ（トリガ）の利用者を見つけるために使用されます。

注: マッピング属性のいずれか 1 つが空白の場合は，次の例で示すように，そのままにしておき

ます。

CONTEXT_ROOT = ${WEB_MODULE.j2eemanagedobject_contextroot:}

この例では，CONTEXT_ROOT は，依存関係シグネチャに空の値を受け取る属性で，無視されます。

変数の値を設定するには：

1. 変数とまったく同じ名前を持つ（大文字と小文字の区別あり）対象データ値を追加します。
2. 対象データ値をハードコード化された値または変数に設定します。

詳細については，「Jython アダプタの開発」(37ページ)を参照してください。

概念変数値の指定

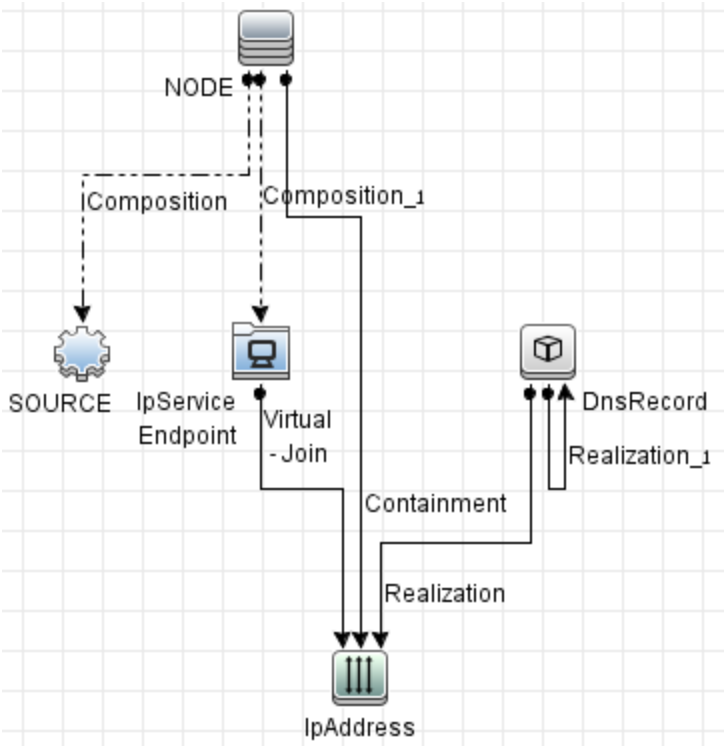
概念は，緊密に連結された分離できない接続文字列セットとして表現するためインスタンス化する必要があります。入力 TQL クエリは分離できない複数の接続文字列セットを返す場合があります。たとえば，実行中のソフトウェアのインスタンスは複数の IP:Port ペアをリッスンする場合があります。そのような各セットでは，別の概念インスタンスをインスタンス化する必要があります。概念のキー属性は，概念インスタンスを分離するために使用されます。キーは TQL クエリのパターン要素を参照します。キー・パターン要素に対して返される各 CI インスタンスには，新規概念インスタンスが作成されます。このような各 CI インスタンスは，**キー CI インスタンス**と呼ばれます。

注: TQL クエリは，同一の概念について複数のマッピング定義を含むことができません。

各概念インスタンスの接続文字列は，一致するキー CI インスタンスおよびそのキー CI インスタンスに接続済みの CI から取得されます。この概念定義を含む依存関係シグネチャ・ファイルの例を次に示します。

```
<Concept name="IpEndpoint">
  <Properties>
    <KeyProperty name="PORT"/>
    <Property name="IPADDRESS"/>
    <Property name="DNS"/>
  </Properties>
</Concept>
```

アダプタの入力 TQL は次のようになります。



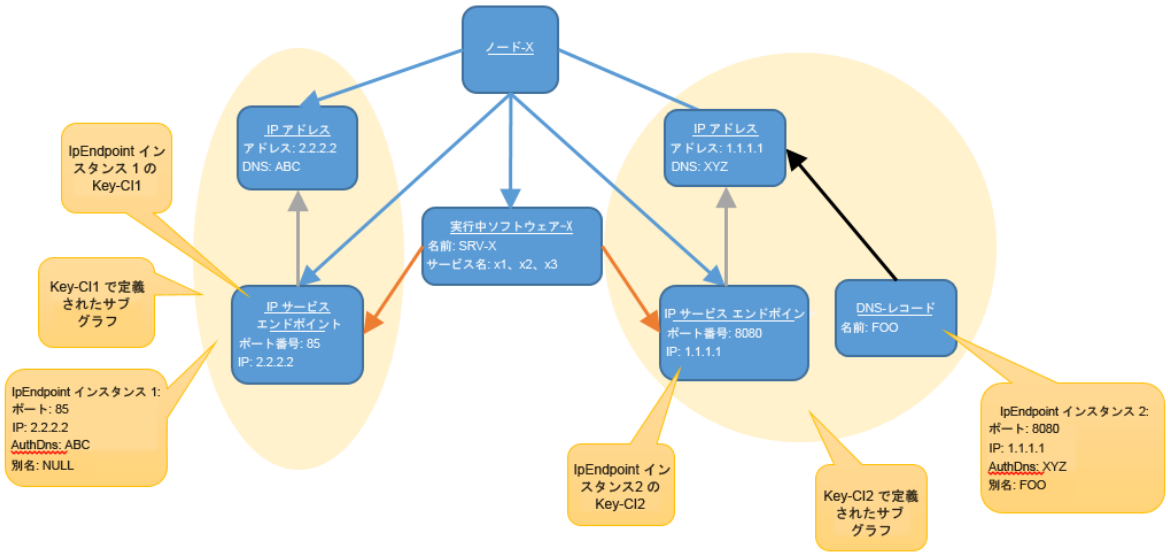
アダプタで概念をインスタンス化するには，変数と同様，各概念変数を対象のデータ変数にマップする必要があります。詳細については，「変数値の指定」(106ページ)を参照してください。

アダプタの対象データを次に示します。

```
IpEndpoint.PORT = SOURCE.NODE.IpServiceEndpoint.name
IpEndpoint.IPADDRESS = SOURCE.NODE.IpServiceEndpoint.IpAddress.name
IpEndpoint.DNS = SOURCE.NODE.IpServiceEndpoint.IpAddress.DnsRecord.name
```

したがって，このアダプタでは，**IpServiceEndpoint** クエリ要素が **IpEndpoint** 概念のキー CI になります。

例の TQL クエリ・グラフィック（実行中ソフトウェア X の TQL クエリの結果）では，2 つのキー CI を認識することが可能です。各 **IpEndpoint** 概念インスタンスの属性を入力するための接続文字列は，各キー CI のサブグラフから取得されます。この方法により，各概念インスタンスは分離できない接続文字列の異なるセットを表すことができます。



変数とは異なり，概念変数をマッピングする場合は，トリガのクエリ・ノード（常に SOURCE という名前）から変数の結合先のクエリ・ノードへのパスを入力 TQL クエリで示す必要があります。さらに，キー変数の設定後に，そのほかのすべての概念変数に対してキー変数のパスをプレフィックスとして付加する必要があります。

この例では，次を確認できます。

- 対象データ変数の名前は，後ろに概念変数の名前が続く概念の名前である。
- 対象データ変数に各概念の名前を指定することで，同一のアダプタに複数の概念をマップできます。
- パスは，常に SOURCE で始まります。
- キー変数でない概念変数のパスは，キー変数のパスがプレフィックスとして付加されます。
- パスは，クエリ・ノード名のみで構成されます。リンクは含まれません。ただし，パスで指定する 2 つのクエリ・ノード名の間に少なくとも 1 つのリンクが存在する必要があります。

パスに次の項目が 1 つ以上含まれる場合，トリガがその配送フェーズ中に失敗します。

- 入力 TQL クエリに存在しないクエリ・ノード名
- 入力 TQL クエリでリンクされていない 2 つの隣接するクエリ・ノード名
- 結果の CI タイプの一部ではない属性名

キー・クエリ・ノードではないクエリ・ノードに対して結果 CI が複数存在する場合，対象データ変数はこれらの CI のすべてのプロパティ値を含むリストになります。リストは，同一の概念のそのほかの対象データ変数に含まれていないパスを持つ対象データ変数についてのみ作成されます。そうでない場合は，トリガがその配送フェーズ中に失敗します。上記の例では，IpEndpoint.IPADDRESS 対象データ変数は，そのパスが IpEndpoint.DNS 対象データ変数に含まれていないため，リストを含むこと

ができません。IpEndpoint.DNS は，そのパスがそのほかの変数に含まれていないため，リストを含むことができます。これは，**IpServiceEndpoint** クエリ・ノードの結果に接続する **IpAddress** クエリ・ノードの結果 CI は 1 つのみしか存在し得ないことを意味します。TQL クエリおよびパスを計画するときは，この点を考慮する必要があります。

TQL クエリに自己リンクが含まれる場合，キー・クエリ・ノードではないクエリ・ノードに複数の結果 CI が含まれる場合と同様に，リストが作成されます。したがって，自己リンクを含むクエリ・ノードについても同様の制限が存在します。

詳細については，「変数とコンセプト」(72ページ)を参照してください。

Jython スクリプトの記述

依存関係マッピング・アダプタの作成における最終ステップは，Jython スクリプトを記述して結果をレポートすることです。

依存関係の検索結果にアクセスするには，次のコードを使用して Workflow State からその結果を取得する必要があります。

```
workflowState = Framework.getWorkflowState()
searchResult = workflowState.getProperty(DependenciesDiscoveryConsts.DEPENDENCIES_DISCOVERY_
RESULT)
```

searchResult 変数は，検索ステップが正常に実行された場合，非 Null であることが保証されます。ワークフローでの検索ステップを必須として定義することをお勧めします。そうすることで，検索ステップが失敗した場合に Jython スクリプト・ステップが実行されないようにすることができます。この変数に，そのタイプのオブジェクト **com.hp.ucmdb.discovery.probe.agents.probemgr.accuratedependencies.search.ConsumerDeployableSearchResult** が含められます。

このオブジェクトを使用して，次の手順を実行します。

1. 依存関係の検索で検出されたすべてのコンポーネント（利用者によるデプロイが可能な）を確認します。検索はプロバイダの接続文字列を入力として取得し，そのプロバイダに依存するすべてのコンポーネント（利用者によるデプロイが可能な）を返すことに注意してください。
2. 各利用者には，複数の依存関係シグネチャが存在する場合があります。それらに対しても反復を実行する必要があります。各依存関係シグネチャには，異なる複数の出力変数値が含まれる場合があります。事実，依存関係シグネチャ（グローバル，ローカル，または概念変数）で使用されたすべての変数の値は，検索結果から Jython スクリプトで利用できます。
3. 利用者とプロバイダの間のリンクを含む Object State Holder（OSH）を作成します。検索結果オブジェクトからプロバイダの詳細を取得することもできます。
4. OSH を結果ベクトル（OSHV）に追加し，結果を UCMDB に送信します。

上記のフローを実行する Jython スクリプト・スニペットを次に示します。

```
# Get the search results object from the Workflow State
```

```
workflowState = Framework.getWorkflowState()
searchResult = workflowState.getProperty(DependenciesDiscoveryConsts.DEPENDENCIES_DISCOVERY_
RESULT)

# Prepare the OSHV that will contain the dependencies
oshv = ObjectStateHolderVector()
dependencyCount = 0

# Retrieve the provider OSH
providerServiceOsh = searchResult.getProviderDeployable().getDeployable()

# Loop through all the consumer deployable components
for index in range(0, searchResult.size()):
  deployable = searchResult.get(index)

  # Get the consumer deployable component’s OSH
  deployableOsh = deployable.getDeployable()

  # Create an OSH for the dependency relationship between the consumer and provider
  consumerProviderLink = modeling.createLinkOSH(‘consumer_provider’, deployableOsh,
providerServiceOsh)

  references = []

  # Iterate through all of the dependencies that were found between the consumer and the provider in
deployable.getDependencies():

    # Extract the name of the dependency as it appears in the dependency signature file
    dependencyName = dependency.getDependencyName()

    # Get the values of all the variables that were used by the dependency
    variables = dependency.getExportVariables()

    # Aggregate the values of the variable named REFERENCE
    # Note:The value of a variable can be a list if the variable contains multiple values
    dependencyNames.append(dependencyName)
    for var in variables:
      varName = var.getName()
      values = var.getValues()
      if varName.lower() == REFERENCES:
        references += list(values)

  reference = references and ','.join(references)
  if reference:
    consumerProviderLink.setAttribute(REFERENCES, reference)

  # Add the link to the results OSHV
  oshv.add(consumerProviderLink)
```

```
# Send the result to the UCMDB
Framework.sendObjects(oshv)
```

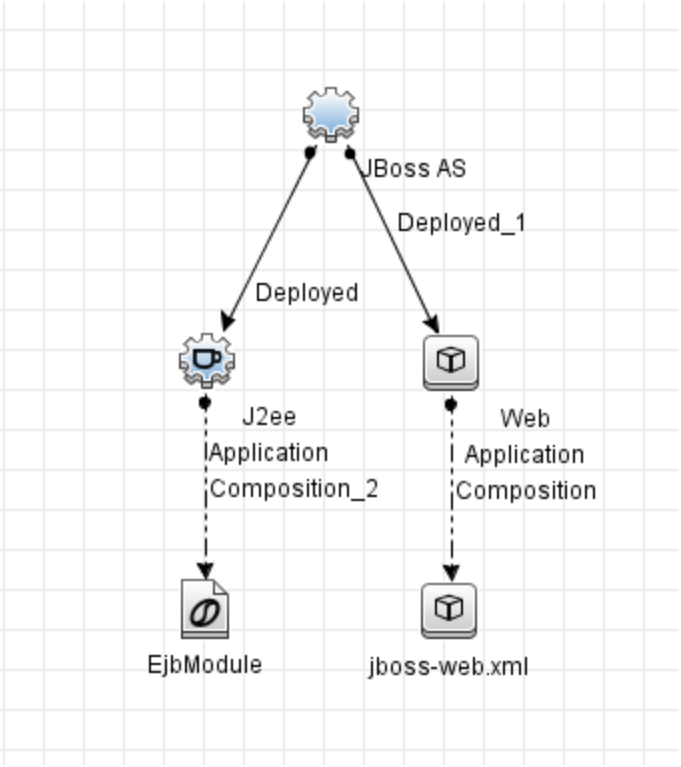
注: サービス境界ルール（ディスカバリ・ルール・エンジンのルールのセット）は，スクリプトによって送信された情報に基づいて実行されます。そのため，関係，利用者，プロバイダについてできる限り多くの情報を送信することが重要になります。上記に示すように，検索結果オブジェクトから取得された利用者とプロバイダの OSH との関係を作成することをお勧めします。これらの OSH には，サービス境界ルール（およびそのほかのルール・エンジンのルール）が正常に機能するために必要なすべての情報が含まれています。

アダプタの制限

- 依存関係マッピング・アダプタは，個別モードでインストールされた Data Flow Probe でサポートされません。
- UCMDB サーバの再起動時に，依存関係マッピング・アダプタのジョブのトリガがタイムアウトにより失敗する場合があります。こうしたトリガは，スケジュール設定された次回の実行で成功します。

完全な例

本項では，依存関係シグネチャとアダプタを開発する方法と，同じ JBoss AS にデプロイされる J2ee アプリケーションと Web アプリケーション間の依存関係を検出する仕組みについて，完全な例を紹介します。



本項の内容

開発ワークフロー

依存関係シグネチャとアダプタを開発する際の一般的な流れは次のとおりです。

1. 開発しようとしているシグネチャのプロバイダ CI タイプは何かを把握します。この例では，プロバイダ・タイプは J2ee アプリケーションになります。
2. 対象のプロバイダに適した利用者のデプロイメント可能コンポーネントを決定します。シグネチャは利用者の構成ドキュメントに固有のものであるため，各利用者がそれぞれ個別の依存関係シグネチャを持つことになります。

   この例では，利用者のデプロイメント可能コンポーネントを 1 つ（JBoss の Web アプリケーション）選んでいます。
3. シグネチャを既存の依存関係シグネチャ・ファイルに追加するか，あるいは新しい依存関係シグネチャ・ファイルを作成するかを決定します。

   この 2 つの方法に機能的な違いはありません。依存関係シグネチャの保守が簡単になる方法を選んでください。たとえば，利用者のデプロイメント可能コンポーネントがいずれかの依存関係シグネチャ・ファイルにすでに存在している場合は，この新しい依存関係を既存のデプロイメント可能コンポーネントに追加した方が保守が簡単になると考えられます。この例では，依存関係シグネチャ・ファイルを新規作成して新しい依存関係シグネチャを追加します。
4. 新しいアダプタが必要か，あるいはプロバイダがすでに既存のアダプタに含まれているかかを

確認します。なお，検索アダプタのトリガ CI はプロバイダ CI タイプです。次の点について確認が必要です。

a. トリガ CI としてこの CI タイプがすでに使用されている既存の検索アダプタがあるか。

ない場合は，そのようなアダプタを作成する必要があります。この例では，こうしたアダプタを作成します。

b. 検索アダプタが存在する場合，必要となる接続文字列のすべての変数やコンセプトは宛先データにマップされているか。

マップされていない場合は，既存のアダプタを更新して，これらの変数を宛先データに追加する必要があります。なお，宛先データは入力 TQL クエリから抽出されます。つまり，新しい情報を取得できるようにするため，TQL クエリは場合によっては変更することが必要になります。

5. アダプタ用のジョブを作成します（まだ作成されていない場合）。
6. トリガ TQL クエリを実行すると，プロバイダがトリガとして返されることを確認します。

依存関係シグネチャを開発する

JBoss J2EE アプリケーションと Web アプリケーション間の依存関係シグネチャを開発します。

1. 利用者とプロバイダ間の依存関係を作成するために必要な接続文字列を把握します。この例の場合，必要な接続文字列は，プロバイダである J2EE アプリケーションに含まれる EJB モジュール内の EJB の JNDI 名です。
2. 接続文字列はすべて，グローバル変数またはコンセプトとして定義される必要があります。これらの変数の値はアダプタによって挿入されます。ただし，依存関係シグネチャ・ファイルでこれらの変数を定義する前に（新規のファイルか既存のファイルかにかかわらず），類似する変数やコンセプトがすでに存在するかどうか確認するためにすべてのファイルをチェックします。存在する場合は，定義する変数，コンセプト，コンセプト変数に，その既存の変数やコンセプトと同じ名前を付けます。こうすると，挿入が必要なグローバル変数の数が増大しないため，アダプタの開発が簡単になります。この例では，これらの定義を新しいファイルに追加します。つまり，ファイルは次のようになります。

```
<VariableDeclarations>
  <Variable name="EJB_JNDI_NAME"/>
</VariableDeclarations>
```

3. 構成ファイルを分析して次のことを特定します。
   - 各構成ファイル内での個々の接続文字列の場所。
   - ファイルのタイプ（プロパティ・ファイル，XML ファイル，その他のテキスト・ファイル）。
   - 各ファイル間の関係（ファイルが複数ある場合）。

この例では，JBoss Web アプリケーションが，**jboss-web.xml** 構成ファイル内で EJB 参照を次のように定義します。

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
  <!-- カスタム JNDI バインディングを持つ同一サーバ内の EJB への参照 -->
  <ejb-ref>
    <ejb-ref-name>ejb/BHome</ejb-ref-name>
    <jndi-name>someapp/ejbs/beanB</jndi-name>
  </ejb-ref>
  <!-- 外部サーバの EJB への参照 -->
  <ejb-ref>
    <ejb-ref-name>ejb/RemoteBHome</ejb-ref-name>
    <jndi-name>jnp://otherserver/application/beanB</jndi-name>
  </ejb-ref>
</jboss-web>
```

<ejb-ref-name> セクションの値は，接続文字列の参照です。これは XML ファイルなので，次の XPath 式を使用して EJB の JNDI 名を照合することができます。

```
//jboss-web/ ejb-ref/ ejb-ref-name[matches(., '^${EJB_JNDI_NAME}$')]
```

4. 標準設定の検索値を含むスコープを定義する必要があります。それにより，依存関係マッピングで必要となる構成ファイルを検出しやすくなります。この例では，キーワードは JNDI 名自体です。次に示すのがスコープの定義です。

```
<ScopeDefinitions>
  <Scope name="JBoss_EJB_Same_Cell">
    <ConfigurationDocumentContentFilter>
      <Operator type="and">
        <Operand value="${EJB_JNDI_NAME}"/>
      </Operator>
    </ConfigurationDocumentContentFilter>
  </Scope>
</ScopeDefinitions>
```

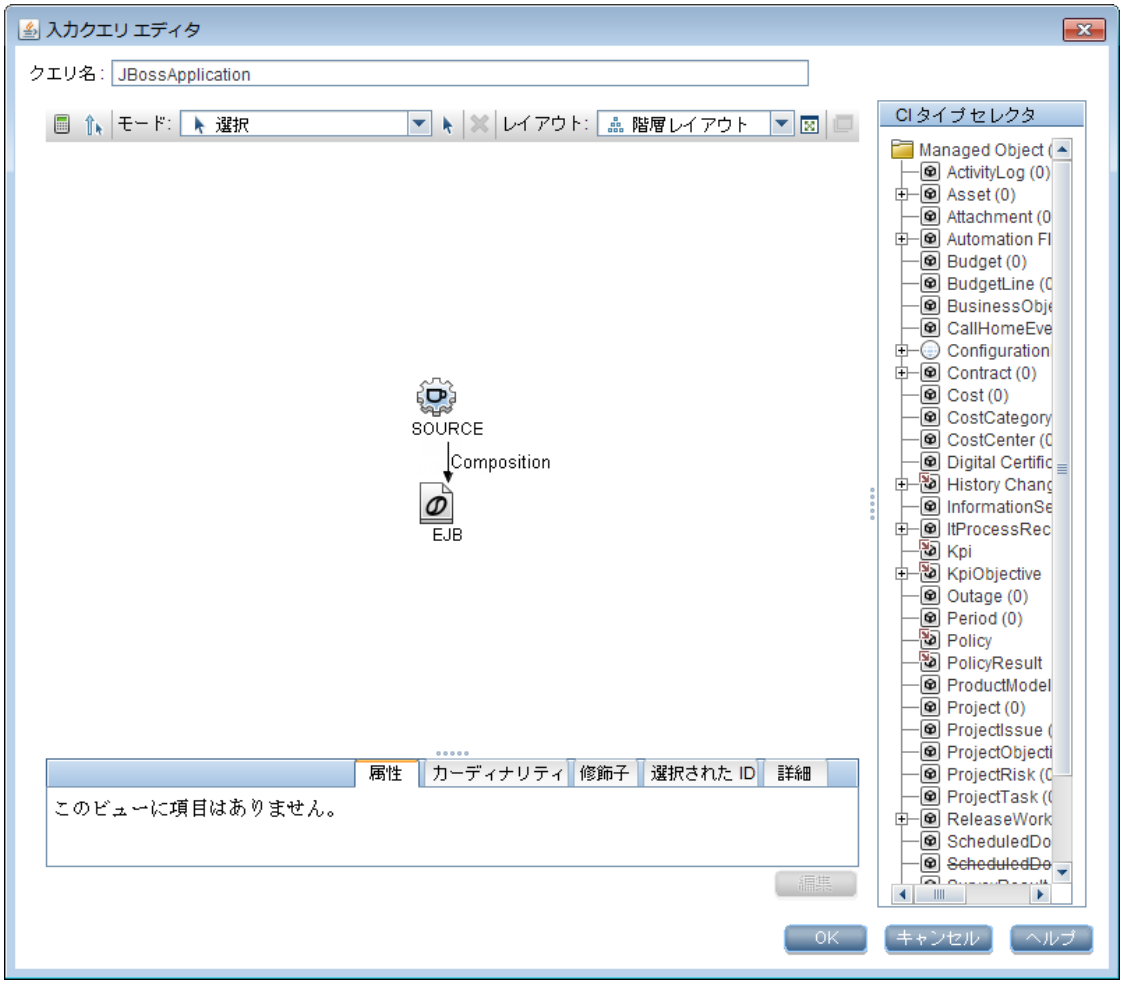
依存関係シグネチャの完全な例は次のようになります。

```
<?xml version="1.0"?>
<DependencySignatures xmlns="http://www.hp.com/ucmdb/1-0-0/Dependencies">
  <VariableDeclarations>
    <Variable name="EJB_JNDI_NAME"/>
  </VariableDeclarations>
  <Deployable name="JBoss J2EE Application to Web Application by JNDI" >
    <Descriptor cit="webapplication"/>
    <Dependency name="J2EE Application with Internal EJB" providerCiType="j2eeapplication"
scope="JBoss_EJB_Same">
```

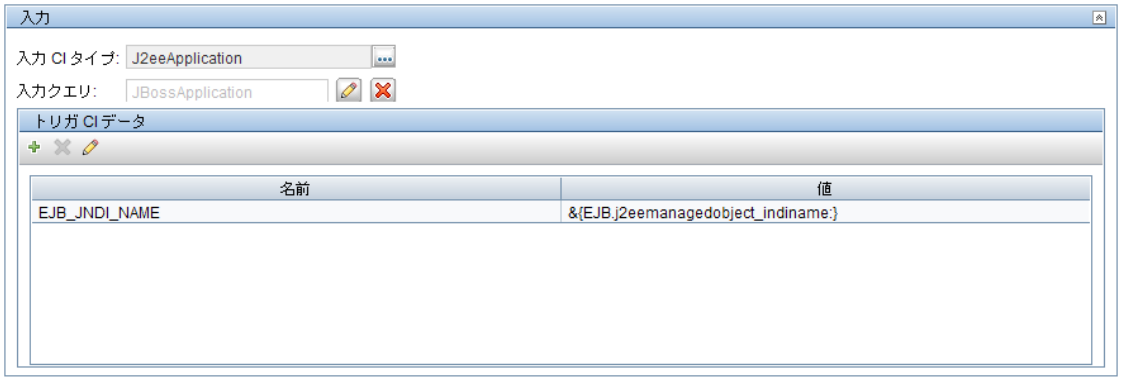
```
        <XmlConfigurationDocument name="jboss-web.xml">
          <Condition>
            <Operator type="or">
              <XPathCondition>
                <XPath>//ejb-ref/jndi-name[matches(., '^${EJB_JNDI_NAME}$', 'i')]</XPath>
              </XPathCondition>
            </Operator>
          </Condition>
        </XmlConfigurationDocument>
      </Dependency>
    </Deployable>
    <ScopeDefinitions>
      <Scope name="JBoss_EJB_Same_Cell">
        <ConfigurationDocumentContentFilter>
          <Operator type="and">
            <Operand value="${EJB_JNDI_NAME}"/>
          </Operator>
        </ConfigurationDocumentContentFilter>
      </Scope>
    </ScopeDefinitions>
</DependencySignatures>
```

アダプタを開発する

1. 「依存関係検索アダプタを作成する」(104ページ)の説明に従って，新しいワークフロー・アダプタを作成します。
2. トリガ CI タイプをプロバイダ・タイプに設定します。この例では，J2EE アプリケーションがプロバイダです。
3. 必要な CI とその属性を取得するために，アダプタの入力 TQL クエリを定義します。この例では，トリガ CI である J2EEApplication CI と，プロバイダである J2EE アプリケーションに属する EJBModule CI が必要です。



4. シグネチャに定義されている必要な接続文字列の変数やコンセプトのグローバル変数を，トリガ CI データとマップします。この例では，EJBModule CI の j2eemanagedobject_jndiname 属性を EJB_JNDI_NAME 宛先データにマップしています。



5. ［ワークフローのステップ］セクションで，次のワークフロー定義を貼り付けます。

```
<workflow>
```

```
  <steps>
    <step name="Accurate Dependency Search" failure-policy="mandatory">
      <module type="jython">DependenciesDiscovery.py</module>
      <timeoutParking>
        <initialTimeout>60000</initialTimeout>
        <retriesThreshold>1</retriesThreshold>
        <multipleBy>1</multipleBy>
        <maxRetry>20</maxRetry>
        <timeoutThreshold>60000</timeoutThreshold>
      </timeoutParking>
    </step>
  </steps>
  <finalStep>
    <module type="jython">AccurateDependencyMapping.py</module>
  </finalStep>
  <libraryScripts />
</workflow>
```

timeoutParking パラメータを調整することでタイムアウト期間を変更できます。

6. 新しいアダプタ用のトリガ TQL クエリを作成します。
7. ［Service Discovery アクティビティ タイプ］で，次のとおりにジョブ定義を追加します。

```
<ServiceDiscoveryActivityType id="top-down" displayName="Top-down">
  <JobsDefinitions>
    ...
    <job id=" JBoss Application to Web Application " displayName=" JBoss Application to Web
Application ">
      <patternId>JBossApplication2WebApplication</patternId>
      <triggers>
        <trigger>jboss_application_trigger</trigger>
      </triggers>
      <parameters/>
    </job>
    ...
  </JobsDefinitions>
</ServiceDiscoveryActivityType>
```

第5章: 汎用データベース・アダプタの開発

本章の内容

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。

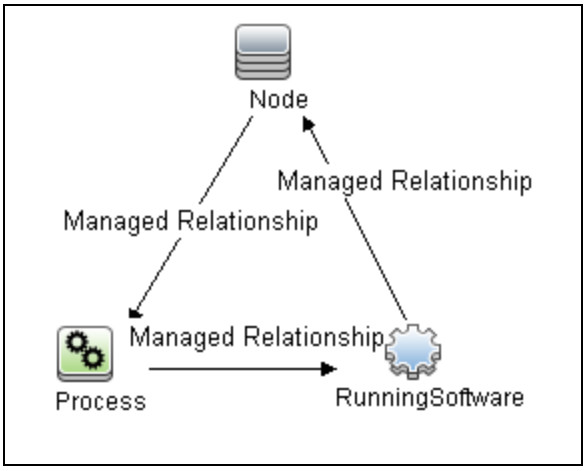
汎用データベース・アダプタ用 TQL クエリ

ポピュレーション・ジョブに対し，要求される CI のすべてのレイアウトは，モデリング・スタジオの［レイアウト設定］ダイアログ・ボックスで確認される必要があります。詳細については，『HP Universal CMDB モデリング・ガイド』の「Query Node/Relationship Properties Dialog Box」を参照してください。CI では，特定される属性が必要な場合があり，それらの属性がないと，CI を UCMDB に追加できないことに注意するのが重要です。

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。

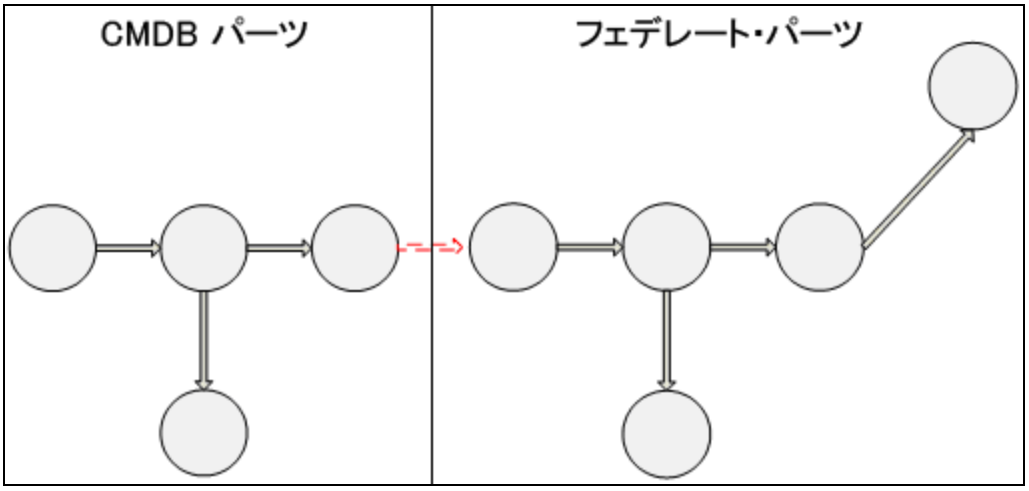
- サブグラフはサポートされていない
- 複合関係がサポートされていない
- サイクルまたはサイクル・パーツがサポートされていない

  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，「前提条件」(124ページ)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。
- グローバル 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，「機能上の制限事項」(191ページ)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，「関連付け」(123ページ)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。

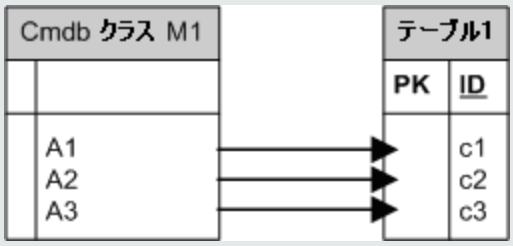
本項の内容

オブジェクト関連マッピングの例

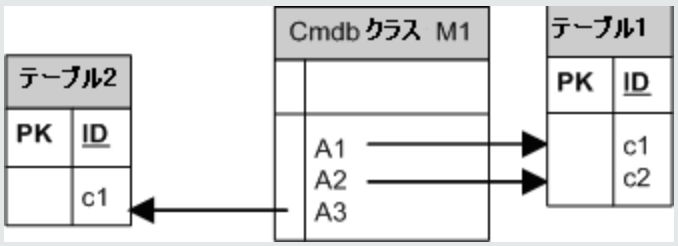
次の例で，オブジェクト関連マッピングについて説明します。

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 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。

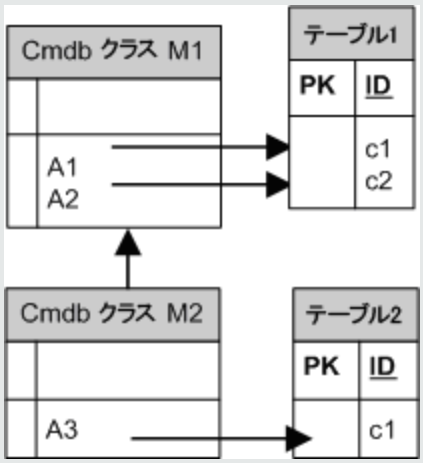


2 つのデータベース・テーブルにマップされた 1 つの CMDB



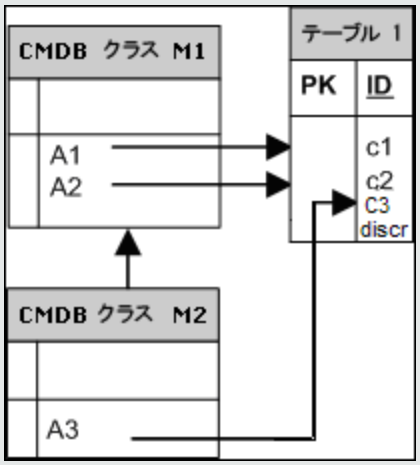
継承の例：

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ため，関連付けの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。

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。

注: UCMDB API では，汎用 DB アダプタのサンプルを表示できます。特に DDMi アダプタのサンプルには複雑な **orm.xml** ファイルが含まれています。また，いくつかのプラグイン・インタフェースの実装も含まれています。

本項の内容

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（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ことを確認します。調整の詳細については，「調整」(121ページ)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
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，「JPA プロバイダとしての Hibernate」(121ページ)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(R) 82801EB USB ユニバーサル・ホスト・コントローラ
3581	2	システム	Intel(R) 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(R) 82801EB USB ユニバーサル・ホスト・コントローラ
2	システム	Intel(R) 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

CARD_ID	CARD_VENDOR
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

   この手順では，RDBMS（外部データ・ソース）のデータを表す CIT を作成します。

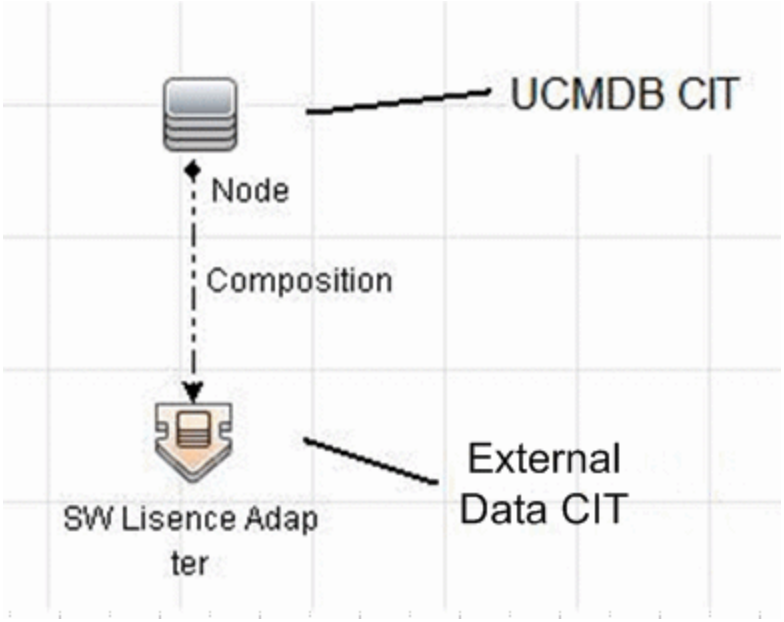
   a. UCMDB で，CI タイプ・マネージャにアクセスして，新しい CI タイプを作成します。詳細については，『HP Universal CMDB モデリング・ガイド』の「CI タイプの作成方法」を参照してください。

   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・ソースからのデータを表す新しい CIT との関係を追加します。

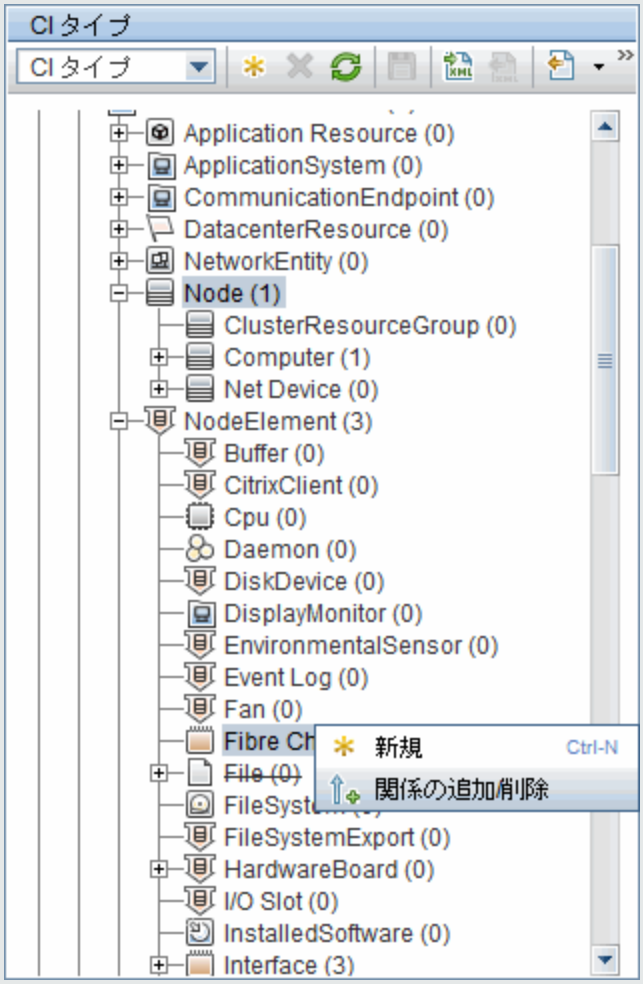
   新しい CIT に適切で有効な関係を追加します。詳細については，『HP Universal CMDB モデリング・ガイド』の「［関係の追加/編集］ダイアログ・ボックス」を参照してください。



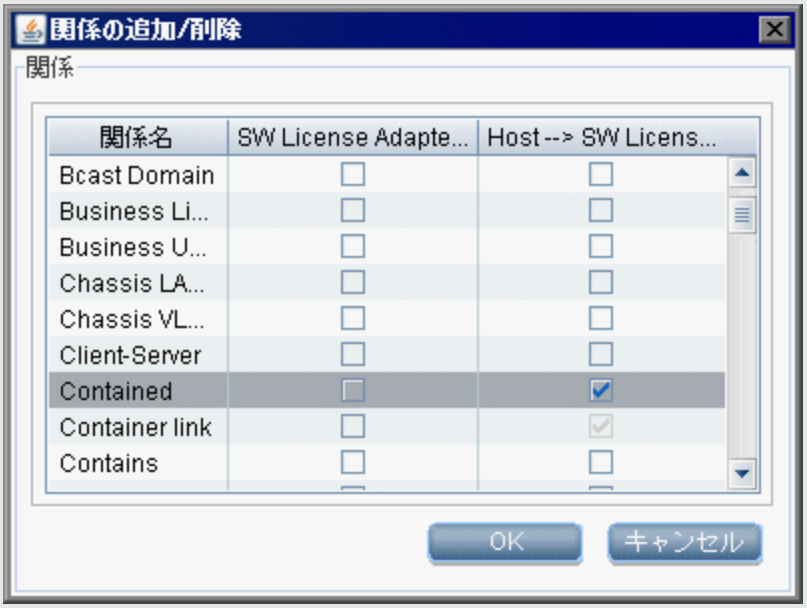
注: この段階では，データを取り込むメソッドを定義していないので，まだフェデレート・データの表示や外部データのポピュレートはできません。

Containment 関係の作成例：

a. CIT マネージャで，2 つの CIT を選択します。



b. 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** フォルダの **db-adapter.zip** パッケージを特定します。
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">
```

アダプタでポピュレーション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［統合ポイント］表示枠のアダプタ・リストに表示されます。

汎用 DB アダプタを作成する場合，**support-replicatioin-data** タグの **changes-source** タグを編集する必要はありません。**FcmdbPluginForSyncGetChangesTopology** プラグインが実装されている場合，最後の実行以降に変更されたトポロジが返されます。プラグインが実装されていない場合，完全トポロジが返され，返された CI に基づき自動削除が実行されます。

データを使用して CMDB 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「［インテグレーション スタジオ］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>

- **カテゴリ**定義を見つけます。

  <category>Generic</category>

  カテゴリ名 **Generic** を任意のカテゴリ名に変更します。

  注: カテゴリが **Generic** と指定されているアダプタは，新規統合ポイントの作成時に Integration Studio に一覧表示されません。

- データベースへの接続は，ユーザ名（スキーマ），パスワード，データベース・タイプ，データベース・ホスト・マシン名，データベース名または SID で記述できます。

  このタイプの接続の場合，アダプタの XML ファイルの **parameter** セクションに，パラメータの次の要素があります。

```
<parameters>
  <!--The description attribute may be written in simple text or HTML.-->
  <!--The host attribute is treated as a special case by UCMDB-->
  <!--and will automatically select the probe name (if possible)-->
```

```
  <!--according to this attribute's value.-->
  <!--Display name and description may be overwritten by I18N values-->
    <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" display-name="Port" type="integer" description="The remote
machine's connection port" mandatory="false" order-index="11" />
    <parameter name="dbtype" display-name="DB Type" type="string" description="The type of
database" valid-values="Oracle;SQLServer;MySQL;BO" mandatory="false" order-
index="13">Oracle</parameter>
    <parameter name="dbname" display-name="DB Name/SID" type="string" description="The
name of the database or its SID (in case of Oracle)"  mandatory="false" order-index="13" />
    <parameter name="credentialsId" display-name="Credentials ID" type="integer"
description="The credentials to be used" mandatory="true" order-index="12" />
</parameters>
```

注: これが標準設定です。そのため，**db_adapter.xml** ファイルにこの定義がすでに含まれています。

場合によっては，データベースへの接続をこの方法で設定できないことがあります。たとえば，Oracle RAC への接続，または CMDB に用意されているドライバ以外のデータベース・ドライバを使用する接続です。

このような場合，ユーザ名（スキーマ），パスワード，接続 URL 文字列を使用した接続を記述します。

これを定義するには，アダプタの XML パラメータのセクションを次のように編集します。

```
<parameters>
  <!--The description attribute may be written in simple text or HTML.-->
  <!--The host attribute is treated as a special case by CMDBRTSM-->
  <!--and will automatically select the probe name (if possible)-->
  <!--according to this attribute's value.-->
  <!--Display name and description may be overwritten by I18N values-->
    <parameter name="url" display-name="Connection String" type="string" description="The
connection string to connect to the database" mandatory="true" order-index="10" />
    <parameter name="credentialsId" display-name="Credentials ID" type="integer" description="The
credentials to be used" mandatory="true" order-index="12" />
</parameters>
```

組み込みのデータ・ダイレクト・ドライバを使用して Oracle RAC に接続する URL の例は次のとおりです。
jdbc:mercury:oracle://labm3amdb17:1521;ServiceName=RACQA;AlternateServers=(labm3amdb18:1521);LoadBalancing=true.

4. 一時ディレクトリで **adapterCode** フォルダを開き，前の手順で使用した **adapter id** の値に **GenericDBAdapter** の名前を変更します。

   このフォルダには，アダプタ名，CMDB のクエリとクラス，アダプタがサポートする RDBMS のフィールドなどのアダプタの設定が含まれています。

5. 必要に応じてアダプタを設定します。詳細については，「アダプタの設定 - 最小メソッド」(131ページ)を参照してください。
6. 手順「アダプタ XML ファイルを編集します。」(128ページ)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ディレクトリは次のとおりです。**C:\hp\UCMDB\UCMDBServer\content\adapters**

アダプタの設定 - 最小メソッド

簡易（最小限の）メソッドは，**simplifiedConfiguration.xml** マッピング・ファイルを作成するためにアダプタが使用するメソッドです。このメソッドは，単一 CIT の基本的ポピュレーションやフェデレーションを可能にします。

本項では，CMDB で特定の CI タイプのクラス・モデルを RDBMS にマッピングするメソッドについて説明します。

本項で言及するすべての構成ファイルは，「アダプタ・パッケージの準備」(128ページ)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。

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。

この最小メソッドを使用するのは，次の操作を実行する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またはポピュレートする。
- 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 つのノードのフェデレーション/ポピュ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

アダプタが簡易設定メソッドを使用できるように，adapter.conf ファイルの設定を変更するには

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

例:簡略化されたメソッドを使用してノードとIPアドレスをポピュレートする

この例では，包含（containment）リンクで関連付けられた **Node** を UCMDB への **IP アドレス** にポピュレートする方法を示します。RDBMS には，コンピュータ名，コンピュータ・ノード，コンピュータの IP アドレスに関するデータを含む **simpleNode** と名付けられたテーブルがあります。

simpleNode テーブルの内容を次に示します。

	host_id	host_name	note	ip_address
	1	Comp1	Test Computer 1	12.33.211.52
	2	Comp2	Test Computer 2	12.33.211.53
	3	Comp3	Test Computer 3	12.33.211.54
	4	Comp4	Test Computer 4	12.33.211.55

ポピュレートは，次に示すように 3 段階で行われます。

1. 「simplifiedConfiguration.xml の作成」(132ページ)
2. 「TQL の作成」(133ページ)
3. 「統合ポイントの作成」(134ページ)

simplifiedConfiguration.xml の作成

次のように **simplifiedConfiguration.xml** を作成します。

1. 次のように **cmdb-class** を作成します。

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

   CI タイプは **node** で，RDBMS テーブル名は **simpleNode** です。
2. 次のようにテーブルのプライマリ・キーを設定します。

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

   プライマリ・キーは **orm.xml** ファイルのエンティティ ID と同じです。
3. 次のように **reconciliation-by-two-nodes** 規則を設定します。

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

   このタグは **Node** と **IpAddress** CI タイプの関係を定義します。関係タイプは包含（Containment）リンクです。調整は 2 つの接続された CI タイプによって行われます。接続ノードの属性マッピング（この場合は IpAddress）は **connected-node** 属性で定義されます。
4. 次のように，調整属性間の **or** 条件を追加します。

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

このタグは，調整属性間の OR 関係を定義します。最初の調整属性が **true** であれば，調整規則全体が **true** に設定されます。

5. 次の属性を追加します。

   <attribute cmdb-attribute-name="name" column-name="host_name" ignore-case="true"/>

   このタグは，UCMDB の **node.name** と，**simpleNode** テーブルの **host_name** カラム間のマッピングを設定します。

   data_note 属性でも同じ操作を実行します。

   <attribute cmdb-attribute-name="data_note" column-name="note" ignore-case="true"/>

   接続ノード属性を追加します。

   <connected-node-attribute cmdb-attribute-name="name" column-name="ip_address"/>

   このタグは，**ip_address.name** と，**simpleNode** テーブルの **ip_address** カラム間のマッピングを設定します。

6. 次の順で開いたタグを閉じます。

   </or>

   </reconciliation-by-two-nodes>

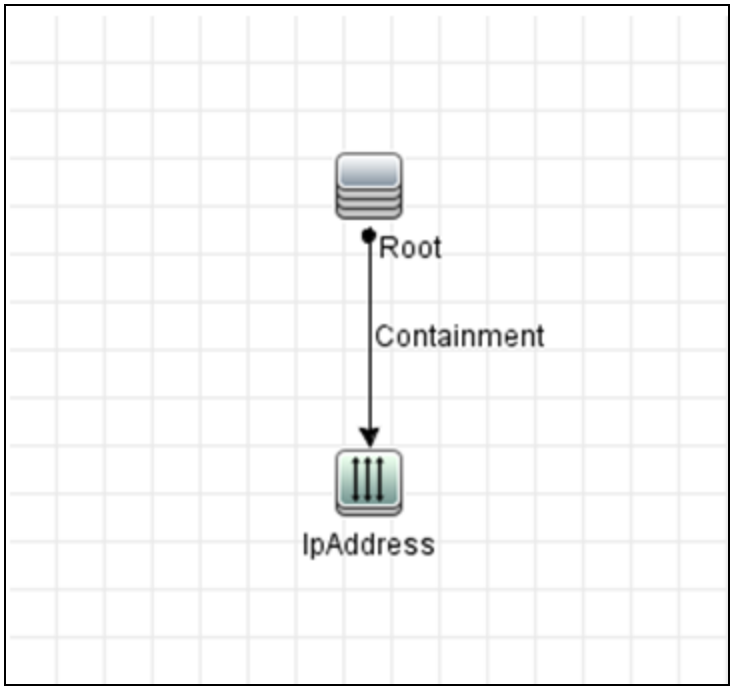
   </cmdb-class>

simplifiedConfiguration.xml ファイルの内容は次のようになります :

```
<?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="simpleNode">
    <primary-key column-name="host_id"/>
  <reconciliation-by-two-nodes connected-node-cmdb-class-name="ip_address" cmdb-link-
type="containment">
    <or is-ordered="true">
    <attribute cmdb-attribute-name="name" column-name="host_name" ignore-case="true"/>
    <attribute cmdb-attribute-name="data_note" column-name="note" ignore-case="true"/>
      <connected-node-attribute cmdb-attribute-name="name" column-name="ip_address"/>
    </or>
    </reconciliation-by-two-nodes>
  </cmdb-class>
</generic-db-adapter-config>
```

TQL の作成

TQL は **ip_address** への Containment（包含）リンクによって接続された **node** です。次に示すように，ノードは **root** としてマークされている必要があります。



TQL を作成するには，次の手順を実行します。

1. **［モデリング］>［モデリング スタジオ］**を選択します。
2. ［**新規作成**］ボタンをクリックして，新しいクエリを作成します。
3. ［CI タイプ］タブで Node CI タイプと IpAddress CI タイプを TQL 画面にドラッグします。
4. **Node** と **IpAddress** を Containment 関係で接続します。
5. **Node** 要素を右クリックし，［クエリ ノードのプロパティ］を選択します。
6. ［**要素名**］を **Root** に変更します。
7. ［**要素レイアウト**］タブに移動します。属性条件として［**特定の属性**］を選択します。［利用可能な属性］ウィンドウから［**Name**］と［**Note**］を選択し，［特定の属性］ウィンドウに移動します。
8. **IpAddress** 要素を右クリックし，［クエリ ノードのプロパティ］を選択します。
9. ［**要素レイアウト**］タブに移動します。属性条件として［**特定の属性**］を選択します。［利用可能な属性］ウィンドウから［**Name**］を選択し，［特定の属性］ウィンドウに移動します。
10. TQL を保存します。

統合ポイントの作成

統合ポイントを作成するには，次の手順に従います。

1. **［データ フロー管理］>［Integration Studio］**に移動し，［**新規統合ポイント**］ボタンをクリックします。
2. 統合ポイントの詳細を挿入し［**OK**］をクリックします。

3. ［ポピュレーション］タブで，［**新規統合ジョブ**］ボタンを選択し，以前作成した TQL を追加します。
4. 統合ポイントを保存し，［**このジョブに関連するすべてのデータを同期するジョブを実行**］ボタンをクリックします。

アダプタの設定 - 高度なメソッド

これらの構成ファイルは，アダプタ・パッケージの準備時に抽出した **C:\hp\UCMDB\UCMDBServer\content\adapters** フォルダの **db-adapter.zip** パッケージにあります。詳細については，「アダプタ・パッケージの準備」(128ページ)を参照してください。

本項の内容

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。

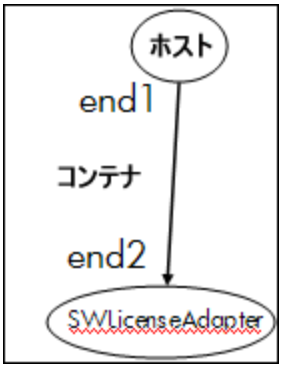
   標準設定では，このファイルには，必要な数の CIT と関係をマップするのに使用するテンプレートが含まれています。

   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。

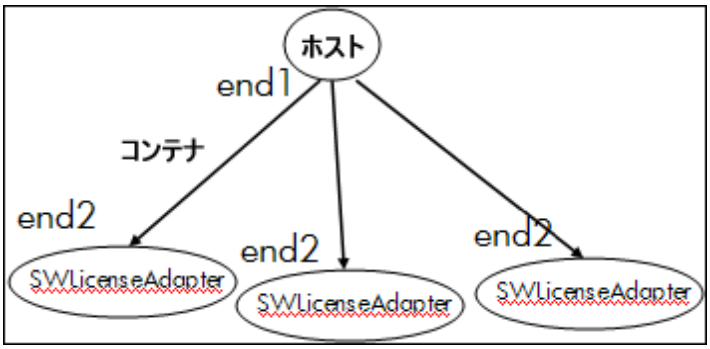
   - 1 対 1:

   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>
```

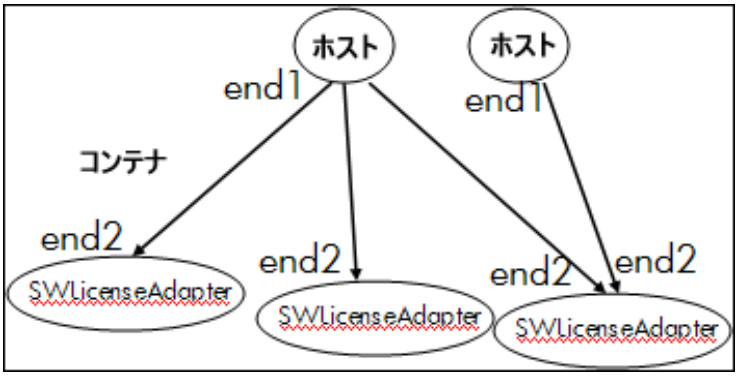
- 多対 1:



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，「命名規則」(159ページ)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"/>

- 「CI タイプの作成」(126ページ)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 :

- CMDB CIT にある属性名と，RDBMS で対応する属性の名前 :

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_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，「reconciliation_rules.txt ファイル（下位互換性用）」(168ページ)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，「アダプタ・パッケージの準備」(128ページ)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. オプション 1 – Java ベースのプラグインを記述する
   a. 次の jar ファイルを UCMDB サーバのインストール・ディレクトリから開発クラスのパスにコピーします。
      - **tools\adapter-dev-kit\db-adapter-framework** フォルダから，**db-interfaces.jar** ファイルおよび **db-interfaces-javadoc.jar** ファイルをコピーします。
      - **\tools\adapter-dev-kit\SampleAdapters\production-lib** フォルダから，**federation-api.jar** ファイルおよび **federation-api-javadoc.jar** ファイルをコピーします。

        注: プラグインの開発の詳細については，**db-interfaces-javadoc.jar** ファイルおよび **federation-api-javadoc.jar** ファイル，および次のオンライン・ドキュメントを参照してください。
        - **C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DBAdapterFramework_JavaAPI\index.html**
        - **C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\Federation_JavaAPI\index.html**

   b. 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。各種インタフェースは，**db-interfaces.jar** ファイル内で定義されています。次の表は，各プラグインに実装する必要があるインタフェースを示しています。

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

プラグインのタイプ	インタフェース名	メソッド
レイアウトの同期化	FcmdbPluginForSyncGetLayout	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	FcmdbPluginGetTopologyCmdbFormat	getTopologyCmdbFormat
CI に対するレイアウト要求の変更	FcmdbPluginGetCIsLayout	getCisLayout
リンクに対するレイアウト要求の変更	FcmdbPluginGetRelationsLayout	getRelationsLayout
ID のプッシュ・バック	FcmdbPluginPushBackIds	getPushBackIdsSQL

プラグインのクラスには，標準設定のパブリック・コンストラクタが必要です。また，すべてのインタフェースは initPlugin メソッドを公開します。このメソッドは，ほかのすべてのメソッドの前に必ず呼び出されます。また，アダプタに付随する環境オブジェクトでアダプタを初期化するために使用されます。

FcmdbPluginForSyncGetChangesTopology が実装されている場合，変更を報告するには，次の 2 種類の異なる方法があります。

- **ルート・トポロジ全体を常に報告する：**このトポロジに従って，自動削除機能でどの CI を削除するかが検出されます。この場合，自動削除機能は，次を使用して有効にする必要があります。

```
<autoDeleteCITs isEnabled="true">
        <CIT>link</CIT>
        <CIT>object</CIT>
      </autoDeleteCITs>
```

- **削除または更新された各 CI インスタンスを報告する：**この場合，自動削除機能は，次を使用して無効にする必要があります。

```
<autoDeleteCITs isEnabled="false">
        <CIT>link</CIT>
        <CIT>object</CIT>
      </autoDeleteCITs>
```

c. 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。

d. 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。

2. オプション 2 – Groovy ベースのプラグインを記述する
   a. アダプタ・パッケージ構成ファイルの［アダプタ管理］メニューで，Groovy コード・ファイル（MyPlugin.groovy）を作成します。
   b. Groovy クラスで，適切なインタフェースを実装します。db-interfaces.jar ファイルにインタフェースが定義されます。上記の表を参照してください。
3. 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。

```
# mandatory plugin to sync full topology
[getFullTopology]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin
# mandatory plugin to sync changes in topology
[getChangesTopology]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin
# mandatory plugin to sync layout
[getLayout]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin
# plugin to get supported queries in sync.If not defined return all tqls names
[getSupportedQueries]
# internal not mandatory plugin to change tql definition and tql result
[getTopologyCmdbFormat]
# internal not mandatory plugin to change layout request and CIs result
[getCisLayout]
# internal not mandatory plugin to change layout request and relations result
[getRelationsLayout]
# internal not mandatory plugin to change action on pushBackIds
[pushBackIds]
```

   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他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 Universal CMDB 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，『HP Universal CMDB データ・フロー管理ガイド』の「アダプタ管理」を参照してください。

統合ポイントの作成

この手順では，フェデレーションが機能していることをチェックします。つまり，接続が有効で，XML ファイルが有効であることをチェックします。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. 統合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「［新規統合ポイント / 統合ポイントの編集］ダイアログ・ボックス」を参照してください。

   ［連携］タブには，この統合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，『HP Universal CMDB モデリング・ガイド』の「パターン・ビューの作成方法」を参照してください。

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（ **[モデリング] > [IT ユニバース マネージャ]** ）にアクセスします。
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，『HP Universal CMDB モデリング・ガイド』の「Topology Reports Overview」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，「アダプタ・ログ・ファイル」(188ページ)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。

プラグインのバージョン 1.1 は，UCMDB バージョン 9.01 以降，Eclipse IDE for Java EE Developers バージョン 1.2.2.20100217-2310 以降と互換性があります。

本項の内容

1. 前提条件

   Eclipse を実行するマシンに，次のサイトから **Java ランタイム環境（JRE）6** の最新アップデートをインストールします。
   http://java.sun.com/javase/downloads/index.jsp（英語サイト）

2. インストール

   a. **Eclipse IDE for Java EE Developers** を **http://www.eclipse.org/downloads（英語サイト）** から **C:\Program Files\eclipse** などのローカル・フォルダ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:\hp\UCMDB\UCMDBServer\tools\db-adapter-eclipse-plugin\bin** から **C:\Program Files\Eclipse\plugins** にコピーします。

   c. **C:\Program Files\Eclipse\eclipse.exe** を起動します。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.zip** を **C:\hp\UCMDB\UCMDBServer\tools\db-adapter-eclipse-plugin\workspace** から **C:\Documents and Settings\All Users** 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デプロイした場合やファイルをデプロイしなかった場合，この手順は機能しません。

   b. Eclipse で，［**File**］>［**Switch Workspace**］>［**Other**］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\All Users\workspace_gdb_sqlserver** を選択します。
      - MySQL の場合，フォルダ **C:\Documents and Settings\All Users\workspace_gdb_mysql** を選択します。
      - Oracle の場合，フォルダ **C:\Documents and Settings\All Users\workspace_gdb_oracle** を選択します。

   c. ［**OK**］をクリックします。

   d. Eclipse で，［Project Explorer］ビューを表示し，［**<アクティブなプロジェクト>**］>［**JPA Content**］>［**persistence.xml**］>［**<アクティブなプロジェクト名>**］>［**orm.xml**］を選択します。

   e. ［Data Source Explorer］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，［**Properties**］メニューを選択します。

   f. ［**Properties for <接続名>**］ダイアログ・ボックスで，［**Common**］を選択し，［**Connect every time the workbench is started**］チェック・ボックスを選択します。［**Driver Properties**］を選択し，接続プロパティを入力します。［**テスト接続**］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. ［Data Source Explorer］ビューで，データベース接続を右クリックし，［**Connect**］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. アダプタの作成

「手順 1 : アダプタの作成」(28ページ)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>** です。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。［Project Explorer］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 Structure］ビューで，［**Entity Mappings**］分岐を右クリックし，［**Add Class**］を選択します。［**Add Persistent Class**］ダイアログ・ボックス

が開きます。［**Map as**］フィールド（**Entity**）は変更しないでください。

c. ［**Browse**］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 Structure］ビューの［**Entity Mappings**］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，［Project Explorer］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 Details］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，次の手順を実行します。

a. ［JPA Structure］ビューの［**Entity Mappings**］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，［**Add Attribute to XML and Map...**］を選択します。

b. ［**Add Persistent Attribute**］ダイアログ・ボックスが開きます。［**Map as**］フィールドで［**Id**］を選択し，［**OK**］をクリックします。

c. ［JPA Details］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 Structure］ビューの［**Entity Mappings**］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，［**Add Attribute to XML and Map...**］を選択します。

b. ［**Add Persistent Attribute**］ダイアログ・ボックスが開きます。［**Map as**］フィールドで［**Basic**］を選択し，［**OK**］をクリックします。

c. ［JPA Details］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

上記の手順「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」(146ページ)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，「reconciliation_rules.txt ファイル（下位互換性用）」(168ページ)を参照してください。

a. 「ID をマップする」(147ページ)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 Details］ビューで［**Details**］チェック・ボックス・グループを展開し，［**Insertable**］および［**Updateable**］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，次の手順を実行します。

- ［JPA Structure］ビューの［**Entity Mappings**］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，［**Add Attribute to XML and Map...**］を選択します。
- ［**Add Persistent Attribute**］ダイアログ・ボックスの［**Map as**］フィールドで，［**Many to One**］または［**One to One**］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，［**Many to One**］を選択します。それ以外の場合は，［**One to One**］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［**Many to One**］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［**One to One**］としてマップする必要があります。
- ［JPA Details］ビューで，［**Target entity**］を選択します（**generic_db_adapter.host** など）。
- ［JPA Details］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。［**Edit Join Column**］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。［**Insertable**］および［**Updatable**］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，［**Add**］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 Structure］ビューで適切なクラスを選択します。［**JPA Details**］ビューで，［**Secondary**

Tables］セクションにアクセスし，［**Add**］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 Details**］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. 上記の「属性をマップする」(147ページ)で説明されているように，属性をマップします。

b. ［JPA Details］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，次の手順を実行します。

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・タグが上記の手順「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」(146ページ)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 Structure］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. ［**保存**］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.xml** 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 Structure］ビューに表示されない場合は，［Project Explorer］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，「ORM ファイルを CMDB にデプロイする」(150ページ)を参照してください。

16. orm.xml ファイルの正確性チェック – 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，エラーが見つかった場合はマークします。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，［**Problems**］ビューにその説明が表示されます。

17. 新規統合ポイントを作成する

    該当するアダプタについて CMDB 内に統合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「モデリング・スタジオ」を参照してください。

    開いたダイアログ・ボックスに統合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。統合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 CIT は除外されます。詳細については，「reconciliation_rules.txt ファイル（下位互換性用）」(168ページ)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** ファイルがアダプタ・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。統合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. モデリング・スタジオでクエリ（ビューではない）を定義します。詳細については，『HP Universal CMDB モデリング・ガイド』の「モデリング・スタジオ」を参照してください。

    b. 「新規統合ポイントを作成する」(150ページ)の手順で作成したアダプタを使用して，統合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「［新規統合ポイント / 統合ポイントの編集］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 タイプがこの統合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，上記の「CMDB プラグインを設定する」(146ページ)の手順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。

アダプタ構成ファイル

この項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。

本項では，次の構成ファイルについて説明します。

一般的な設定

- **adapter.conf:** アダプタ構成ファイルです。詳細については，「adapter.conf ファイル」(152ページ)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，「simplifiedConfiguration.xml ファイル」(153ページ)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，「orm.xml ファイル」(155ページ)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，「reconciliation_rules.txt ファイル（下位互換性用）」(168ページ)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，「transformations.txt ファイル」(170ページ)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，利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，「discriminator.properties ファイル」(171ページ)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，「replication_config.txt ファイル」(172ページ)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，「fixed_values.txt ファイル」(172ページ)を参照してください。

Hibernate 設定

- **persistence.xml:** 定義済みの Hibernate 設定を上書きするのに使います。詳細については，「Persistence.xml ファイル」(173ページ)を参照してください。

アダプタの一時テーブル・サポートの有効化

一時テーブルを有効にすると，アダプタはリモート・データベースとより効率的に連携できるようになるため，データベースやネットワーク上のストレスを減らし，パフォーマンスを向上します。

汎用データベース・アダプタで一時テーブル・サポートを有効にするには，以下の条件を満たす必要があります。

- データベースへの接続に与えられる資格情報に，一時テーブルの作成，変更，削除の権限が含まれていること。
- adapter.conf 構成ファイルで次の設定を行います :

  temp.tables.enabled=true

  performance.enable.single.sql=true

注: 一時テーブルは Microsoft SQL と Oracle でのみサポートされます。

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，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。

- **performance.memory.id.filtering=true**。GDBA が TQLS を実行するときに，大量の ID を取得し，データベースに SQL を使用して戻す場合があります。この過度な負荷を避けてパフォーマンスを改善するため，GDBA はビューとテーブル全体を読み取り，メモリ内の結果をフィルタ処理することを試みます。

- **id.reconciliation.cmdb.id.type=string/bytes**。ID 調整を使用して汎用 DB アダプタをマッピングするときに，**META-INF/ adapter.conf** プロパティを変更することで，**cmdb_id** を **string** または **bytes/raw** カラム・タイプにマッピングできます。
- **performance.enable.single.sql=true**。このパラメータは省略可能です。ファイルにない場合，標準設定の値は **true** になります。**true** の場合，汎用データベース・アダプタでは，実行される各クエリに対し，単一の SQL ステートメントの生成が試行されます（ポピュレーション・クエリまたはフェデレート・クエリのいずれかに対し）。単一の SQL ステートメントを使用することで，汎用データベース・アダプタのパフォーマンスとメモリの消費が改善されます。**false** の場合，汎用データベース・アダプタでは複数の SQL ステートメントが生成され，単一のクエリよりも所要時間が長くなり，メモリの消費量が増える可能性があります。この属性を **true** に設定しても，アダプタは次の場合には単一の SQL ステートメントを生成しません。
  - アダプタの接続先データベースが Oracle または SQL Server ではない。
  - 実行する TQL に，0..* と 1..* 以外のカーディナリティ条件が含まれている（たとえば，2..* または 0..2 などのカーディナリティ条件がある）。
- **in.expression.size.limit=950**（デフォルト）:このパラメータは，引数のリストのサイズ上限に達した場合に，実行された SQL の「IN」表現を分割します。
- **stringlist.delimiter.of.<CIT Name>.<Attribute Name>=<delimiter>** :文字列リスト属性を汎用データベース・アダプタ内のデータベース・カラムにマッピングするには，連結値のリストがある文字列カラムに属性をマッピングする必要があります。たとえば，CI タイプ **policy** で属性 **policy_category** をマッピングするために，文字列カラムに次の値のリストがある場合 :value1##value2##value3（これは 3 つの値 value1, value2, value3 のリストを定義します），次の設定を使用します :**stringlist.delimiter.of.policy.policy_category=##**。
- **temp.tables.enabled=true**:一時テーブルの使用を有効にしてパフォーマンスを向上します。**performance.enable.single.sql** が有効な場合にのみ使用できます（Microsoft SQL と Oracle でのみサポートされます）。データベース・サーバで特定の権限が要求される場合があります。
- **temp.tables.min.value=50**:一時テーブルを使用するために必要な条件値（または ID）の数を定義します。

simplifiedConfiguration.xml ファイル

このファイルは，UCMDB 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

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

- **CMDB-class-name** プロパティは，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。

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.impl. GenericEnumTransformer
(generic-enum-transformer-example.xml)" to-CMDB-onverter="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 クエリのみをマップする場合に使用できます。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-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

- 「orm.xml ファイルの作成例」(164ページ)
- 「各リモート製品バージョンの特定 orm.xml の構成」(168ページ)

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 属性の宣言に使用します。**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>
```

ノード属性: ノード属性の追加方法の例を示します。

```
<entity class="generic_db_adapter.host_node">
 <discriminator-value>host_node</discriminator-value>
  <attributes/>
</entity>
<entity class="generic_db_adapter.nt">
 <discriminator-value>nt</discriminator-value>
```

```
  <attributes>
   <basic name="nt_servicepack">
    <column updatable="false" insertable="false" name="specific_type_value"/>
   </basic>
  </attributes>
</entity>
```

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。**
- **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。

orm.xml スキーマ

次の表は，**orm.xml** の共通要素について説明しています。完全なスキーマは http://java.sun.com/xml/ns/persistence/orm_1_0.xsd にあります。 このリストは完全なものではありません。主に汎用データベース・アダプタの標準 Java Persistence API の特定の動作について概要を示したものです。

要素名およびパス	説明	属性
entity-mappings	エンティティ・マッピング・ドキュメントのルート要素。 この要素は，GDBA サンプル・ファイル内で指定されている要素と完全に一致している必要があります。	
description (entity-mappings)	エンティティ・マッピング・ドキュメントの説明テキスト。（任意指定）。	
package (entity-mappings)	マッピング・クラスを含む Java パッケージの名前。 テキスト `generic_db_adapter` が必ず含まれている必要があります。	1. **名前 :** name **説明 :** このエンティティのマップ先の UCMDB CI タイプの名前。 このエンティティが CMDB 内のリンクにマップされる場合，エンティティの名前は次の形式である必要があります。 `<end_1>_<link_name>_<end_2>` たとえば，`node_composition_cpu` はノードと CPU 間の composition リンクにマップされるエンティティを定義しています。 CI タイプの名前が，パッケージ・プレフィックスがない Java クラスの名前と同じ場合，このフィールドは省略できます。 **必須かどうか :** 任意指定 **タイプ :** 文字列

要素名およびパス	説明	属性
		2. **名前 :** class **説明 :** この DB エンティティ用に作成される Java クラスの完全修飾名。 この Java クラスのパッケージ名は，`package` 要素内で指定されている名前と同じである必要があります。 クラス名には，interface や switch などの Java の予約語は使用できません。 その代わり，名前にプレフィックス `gdba_` を付加することができます。そのため，interface は次のようになります。 `generic_db_adapter.gdba_interface` **必須かどうか :** 必須 **タイプ :** 文字列
table （entity-mappings>entity）	この要素は，DB エンティティのプライマリ・テーブルを定義します。 1 回しか使用できません。 必須 :	**名前 :** name **説明 :**プライマリ・テーブルの名前。このテーブルの名前に所属先のスキーマが含まれていない場合，このテーブルは統合ポイントの作成で使用されたユーザのスキーマ内のみで検索されます。この名前は任意の有効な SELECT ステートメントとすることもできます。 SELECT ステートメントの場合，括弧で囲む必要があります。 **必須かどうか :** 必須 **タイプ :** 文字列
secondary-table （entity-mappings > entity）	この要素は，DB エンティティのセカンダリ・テーブルの定義に使用できます。 このテーブルは，1 対 1 の関係でプライマリ・テーブルと関連付けられている必要があります。 複数のセカンダリ・テーブルを定義できます。 任意指定。	**名前 :** name **説明 :** セカンダリ・テーブルの名前。このテーブルの名前に所属先のスキーマが含まれていない場合，このテーブルは統合ポイントの作成で使用されたユーザのスキーマ内のみで検索されます。この名前は任意の有効な SELECT ステートメントとすることもできます。 SELECT ステートメントの場合，括弧で囲む必要があります。

要素名およびパス	説明	属性
		必須かどうか : 必須 **タイプ :** 文字列
primary-key-join-column （entity-mappings > entity > secondary-table）	セカンダリ・テーブルとプライマリ・テーブルが同じ名前のフィールドを使用して関連付けられていない場合，この要素は，プライマリ・テーブルのプライマリ・キー・フィールドと関連付ける必要があるセカンダリ・テーブル内のプライマリ・キー・フィールドの名前を定義します。	**名前 :** name **説明 :**セカンダリ・テーブル内のプライマリ・キー・フィールドの名前。 この要素が存在しない場合，プライマリ・キー・フィールドの名前が，プライマリ・テーブルのプライマリ・キー・フィールド名と同じとみなされます。 **必須かどうか :** 任意指定 **タイプ :** 文字列
inheritance （entity-mappings > entity）	現在のエンティティが DB エンティティ・ファミリの親エンティティである場合，この要素を使用して親であることを示します。 任意指定。	**名前 :** strategy **説明 :** DB 内の継承の実装方法を定義します。 **必須かどうか :** 必須 **タイプ :**次のいずれかの値を使用します。 • SINGLE_TABLE : このエンティティとすべての子エンティティが同一のテーブル内に存在します。 • JOINED : 子エンティティは結合テーブル内に存在します。 • TABLE_PER_CLASS : 各エンティティはそれぞれ個別のテーブルで定義されます。
discriminator-column （entity-mappings > entity）	継承のタイプが SINGLE_TABLE の場合，この要素は各行のエンティティのタイプを判別するフィールド名の定義に使用されます。	**名前 :** name **説明 :**識別子カラムの名前。 **必須かどうか :** 必須 **タイプ :** 文字列
discriminator-value （entity-mappings > entity）	この要素は，継承ツリー内の特定のエンティティのタイプを定義します。 この名前は，特定のエンティティ・タイプの値グループの **discriminator.properties** ファイルで定義されている名前と同じ	

要素名およびパス	説明	属性
	である必要があります。	
attributes （entity-mappings > entity）	エンティティのすべての属性マッピングのルート要素。	
id （entity-mappings > entity attributes）	この要素は，エンティティのキー・フィールドを定義します。 最低 1 つの id フィールドを定義する必要があります。 複数の id 要素が存在する場合，それらのフィールドによりエンティティの複合キーが作成されます。 CI エンティティに対して複合キーの使用は避けてください（リンクは除く）。	**名前 :** name **説明 :**文字列タイプ idX。ここで X は 1 ~ 9 の数字です。最初の id は id1，2 番目は id2，以降同様に続きます。 これは，UCMDB 内のキー属性の名前ではありません。 **必須かどうか :** 必須 **タイプ :** 文字列
basic （entity-mappings > entity attributes）	この要素は，テーブル内のフィールド間のマッピングを定義します。定義する値はこのテーブルのプライマリ・キーの一部ではなく，UCMDB の属性です。	**名前 :** name **説明 :** フィールドのマッピング先の UCMDB 属性の名前。 この属性は，現在のエンティティのマッピング先の UCMDB の CI タイプ内に存在する必要があります。 **必須かどうか :** 必須 **タイプ :** 文字列
column (entity-mappings > entity > attributes >id -または- (entity-mappings > entity > attributes > basic)	basic マッピングまたは id フィールドのテーブル内のカラム名を定義します。	1. **名前 :** name **説明 :**フィールドの名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** table **説明 :** フィールドが属するテーブルの名前。 このテーブルは，プライマリ・テーブル，またはエンティティに定義されているいずれかのセカンダリ・テーブルである必要があります。 この属性が省略されている場合，フィールドはプライマリ・テーブルに属しているとみなされます。 **必須かどうか :** 任意指定 **タイプ :** 文字列

要素名およびパス	説明	属性
one-to-one (entity-mappings > entity > attributes)	値が別のテーブルにあり，2 つのテーブルが 1 対 1 の関係で関連付けられているカラムを定義します。 この要素は，リンク・エンティティのマッピングのみに対応し，ほかの CI タイプには使用できません。 テーブルと UCMDB リンク間のマッピングはこの方法でのみ定義できます。	1. **名前 :** name **説明 :**このフィールドが 2 つのどちらの端を表しているかを指定します。 **必須かどうか :** 必須 **タイプ :** `end1` または `end2` 2. **名前 :** target-entity **説明 :** 端の参照先のエンティティ名。 **必須かどうか :** 必須 **タイプ :**エンティティ・マッピング・ドキュメントで定義されているいずれかのエンティティ名
join-column (entity-mappings > entity attributes > one-to-one)	1 対 1 の関係の親要素で定義されているターゲット・エンティティと現在のエンティティを結合する方法を定義します。	1. **名前 :** name **説明 :**1 対 1 の結合を行うために使用する，現在のテーブルのフィールド名。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** name **説明 :**結合エンティティ内のフィールド名。このフィールドで結合を実行します。 この属性が省略されている場合，結合テーブルには，name 属性で定義されたフィールド名と同じ名前のカラムがあるとみなされます。 **必須かどうか :** 任意指定 **タイプ :** 文字列

orm.xml ファイルの作成例

ここに示す例は，**orm.xml ファイル**の作成方法を示したものです。 この例で，リモート・データベースの SQL テーブルは，UCMDB の CI タイプにマップされています。

リモート・データベース内のテーブルが次の形式であるとして，**Hosts** テーブルにノードを，**IP_Addresses** テーブルに IP アドレスをポピュレートし，ノードと IP アドレスとの間に次のリンクを作成します。

Hosts テーブル

host_name	host_id
Test1	1
Test2	2
Test3	3

IP_Addresses テーブル

ip_address	ip_id
10.1.1.1	1
10.2.2.2	2
10.3.3.2	3
10.4.4.4	4

Host_IP_Link テーブル（ノードと IP アドレスのリンク）

host_id	ip_id
1	1
2	2
2	3
3	4

Hosts テーブルのプライマリ・キーは **host_id** フィールドであり，**IP_Addresses Table** テーブルのプライマリ・キーは **ip_id** フィールドです。**Host_IP_Link** テーブルで，**host_id** と **ip_id** は **Hosts Table** と **IP_Addresses Table** からもたらされた無関係なキーです。

上記のテーブルに従い，**orm.xml** ファイルを次のように作成します。 この例で使用するエンティティは **node**，**ip_address**，**node_containment_ip_address** です。

1. 次のように **Hosts** テーブルから **host_id** をマッピングして，**node** エンティティを作成します。

```
<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_1_0.xsd">
        <description>test_integration</description>
        <package>generic_db_adapter</package>
        <entity class="generic_db_adapter.node">
```

```
        <table name="hosts"/>
        <attributes>
    <id name="id1">
        <column updatable="false" insertable="false" name=
            "host_id"/>
        <generated-value strategy="TABLE"/>
    </id>
    <basic name="name">
        <column updatable="false" insertable="false" name=
            "host_name"/>
    </basic>
        </attributes>
     </entity>
```

エンティティ・**クラス**は，UCMDB の既存の CI タイプである必要があります。 テーブル**名**は，ID 情報と Host 情報の両方が含まれるデータベース内のテーブルです。 特定のホストを識別するために ID 属性が必要であり，これがあとでマッピングに使用されます。 この例では，このエンティティの **name** 属性は，hosts テーブルの **host_name** 列にポピュレートされます。

2. 次のエンティティについて，Interfaces テーブルから IP アドレスをマップします。

```
<entity name="ip_address" class="generic_db_adapter.ip_address">
       <table name="IP_Addresses"/>
       <attributes>
           <id name="id1">
      <column insertable="false" updatable="false" name="ip_id"/>
      <generated-value strategy="TABLE"/>
           </id>
           <basic name="name">
      <column updatable="false" insertable="false" name="ip_address"/>
           </basic>
       </attributes>
</entity>
```

3. 次に，ノードと IP アドレスのリンクはマッピング・テーブルを使用し，**ip_id** フィールドを参照して作成する必要があります（必要に応じて **host_id** フィールドと **ip_id** フィールドの両方を参

照する場合もあります）。

```
<entity name="node_containment_ip_address"
      class="generic_db_adapter.node_containment_ip_address">
       <table name="Host_IP_Link"/>
       <attributes>
            <id name="id1">
      <column updatable="false" insertable="false" name="ip_id"/>
      <generated-value strategy="TABLE"/>
            </id>
            <many-to-one target-entity="node" name="end1">
      <join-column name="host_id"/>
            </many-to-one>
            <one-to-one target-entity="ip_address" name="end2">
      <join-column name="ip_id"/>
            </one-to-one>
       </attributes>
</entity>
```

コンテナのエンティティ名は[end1 CIT]_[link CIT]_[end2 CIT] の形式となります。この例では，リンク CI タイプが **containment** になるので，コンテナのエンティティ名は**node_containment_ip_address** となり，エンティティ・クラスは **generic_db_adapter.node_containment_ip_address** となります。 このコード・ブロックには ID が必要で，この例はインタフェースの単一の ID で正常に機能しますが，両方のカラムで id1 と id2 を参照できます。 この場合のコードは次のようになります。

```
<id name=” id1” >
      <column updatable=” false” insertable=” false” name=” ip_id” />
      <generated-value strategy=” TABLE” />
</id>
<id name=” id2” >
      <column updatable=” false” insertable=” false” name=” host_id” />
      <generated-value strategy=” TABLE” />
</id>
```

このリンクの 2 つの終端は ‘many-to-one’ と ‘one-to-one’ です。つまり，各 IP アドレスは 1 つのノードに対してリンクされますが，1 つのノードは複数の IP アドレスに対してリンクされる可能性があります。 含まれるこれらのカラムは，Links テーブルからのもので，ここから Hosts テーブルと Interfaces テーブルが参照されます。

各リモート製品バージョンの特定 orm.xml の構成

指定したリモート製品バージョンの特定の **orm.xml** をアダプタが使用するように，特定の **orm.xml** ファイルを構成できます。たとえば，リモート・データストアに 2 つの製品バージョン x と y がある場合，各バージョンにエンティティの異なるマッピングを行うことができます。

リモート製品バージョンごとに特定の orm.xml ファイルを構成するには :

1. **version** とい名前のパラメータを **adapter.xml** ファイルに追加して，可能なバージョン値を **valid-values** として指定します。
2. アダプタ・パッケージで，META-INF フォルダに **VersionOrm** という名前のフォルダを作成します。
3. **VersionOrm** フォルダで，各バージョンに **orm.xml** ファイルを作成します。ファイル名にはバージョンのプレフィックスを含めます。たとえば，バージョンが **x** の場合は，ファイル名は **x_ orm.xml** とします。

注: META-INF フォルダ内の **orm.xml** ファイルは，リモート製品バージョンに特定の **orm.xml** ファイルを作成するかどうかにかかわらず，任意のリモート製品バージョンにロードされます。すべてのバージョンに同じ方法でエンティティをマッピングできます。

reconciliation_types.txt ファイル

UCMDB 10.00 より，**reconciliation_types.txt** ファイルは該当しなくなりました。どの CIT でも調整に使用できます。連携エンジンが自動的にマッピングを実行します。

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 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 CMDB 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 クエリ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 クエリは 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 nodeAND/ORname 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 からの n ame OR ip_address からの name : CMDB 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 からの name OR ip_address からの 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 クエリには 2 つのノードしか持てません。たとえば，node > ticket TQL クエリは，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 : インタフェース **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerToExternalDB** を実装するクラスの完全修飾名。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，「定義済みのコンバータ」(175ページ)を参照してください。

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。または，条件が識別子カラムの対応する値にマップされます。

条件を使用する場合，使用する構文は，like(condition) になります。condition の条件には，次のワイルドカードを含む文字列を使用できます。

- **%**（パーセント記号）：任意の長さ（長さ 0 の文字列を含む）の任意の文字列と一致します。
- **_**（アンダースコア）：単一の文字と一致します。

たとえば，like(%unix%) を指定すると，unix，linux，unix-aix などに一致します。Like 条件は文字列カラムにのみ適用できます。

'all-other' と指定することで，単一の識別子の値を，別の識別子に属さないすべての値とマップすることもできます。

作成しているアダプタで識別子機能を使用している場合，**discriminator.properties** ファイルですべての識別子値を定義する必要があります。

識別子マッピングの例：

たとえば，アダプタがCIタイプ node，nt，unix をサポートし，データベースに t_nodes という名前の単一のテーブルが格納され，このテーブルに **type** という名前にカラムがあるとします。この場合，タイプが 10001 であれば，この行はノードを表し，タイプが 10004 であれば unix マシンを表すなどになります。**discriminator.properties** ファイルの内容は次のようになります。

```
node=10001, 10005
nt=10002,10003
unix=2%
mainframe=all-other
```

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="t_nodes" />
   ...
   <inheritance strategy="SINGLE_TABLE" />
   <discriminator-value>node</discriminator-value>
   <discriminator-column name="type" />
   ...
</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。

- 特定のエントリの **type** に 10002 または 10003 が含まれる場合，このエントリは **nt** CIT にマップされます。
- 特定のエントリの **type** に 10001 または 10005 が含まれる場合，このエントリは **node** CIT にマップされます。
- 特定のエントリの **type** が 2 で始まる場合，このエントリは **unix** CIT にマップされます。
- **type** カラムのその他すべての値は，**mainframe** CIT にマップされます。

> **注: node** CIT は，**nt** と **unix** の親でもあります。

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，「プラグイン」(180 ページ)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]

このファイルでも定数のリストがサポートされます。定数リストを定義するには，次の構文を使用します。

entity[<entityName>] attribute[<attributeName>] value[{<Val1>, <Val2>, <Val3>, ... }]

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，定義済みではないデータベース・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 SQL 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（詳細については，http://www.hibernate.org（英語サイト）を参照）。

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">
   <!-- この値は変更しないでください -->
   <persistence-unit name="GenericDBAdapter">
      <properties>
         <!-- この値は変更しないでください -->
         <property name="hibernate.archive.autodetection" value="class,
           hbm" />
         <!--ドライバ・クラス名"/-->
         <property name="hibernate.connection.driver_class" value="com.
          mercury.jdbc.MercOracleDriver" />
         <!--接続 url"/-->
         <property name="hibernate.connection.url" value="jdbc:mercury:
          oracle://artist:1521;sid=cmdb2" />
         <!--DB ログイン資格情報"/-->
         <property name="hibernate.connection.username" value="CMDB" />
         <property name="hibernate.connection.password" value="CMDB" />
         <!--接続プール・プロパティ"/-->
         <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" />
         <!--使用するダイアレクト-->
         <property name="hibernate.dialect" value="org.hibernate.dialect.
           OracleDialect" />
      </properties>
   </persistence-unit>
</persistence>
```

NT 認証を使用したデータベースへの接続

NT 認証を要求する MS SQL Server に接続できます。これを行うには，ドメインを解析できるドライバが必要です（すなわち，jTDS JDBC ドライバ）。

認証は指定されたパラメータ（ドメイン，ユーザ名，パスワード）に応じて行われ，現在実行しているプロセスの NT 資格情報では行われません。

1. **persistence.xml** で，次のプロパティを以下によって編集します :

```
<!--ドライバ・クラス名"/-->
<property name="hibernate.connection.driver_class" value="net.sourceforge.jtds.jdbc.Driver"/>
<property name="hibernate.connection.url" value="jdbc:jtds:sqlserver://[host name]:
[port];DatabaseName=[database name];domain=[the domain]"/>
<!--DB ログイン資格情報"/-->
<property name="hibernate.connection.username" value="[username]"/>
<property name="hibernate.connection.password" value="[password]"/>
```

2. JDBC ドライバ・ファイルを以下に配置します :**<プローブのインストール・フォルダ>\lib**。
3. Probe を再起動します。

SCCM 統合で NTLM 認証を使用するように，Persistence.xml ファイルを設定します。

注 : 本項は SCCM 統合にのみ適用されます。

SCCM 統合で NTLM 認証を使用するには，**persistence.xml** ファイルを次のように設定します。

1. JDBC ドライバ・ファイルを以下に配置します :**<プローブのインストール・フォルダ>\lib**。

   たとえば，http://sourceforge.net/projects/jtds/files/ から **jtds-1.3.1.jar** ファイルを **DataFlowProbe\lib** フォルダに配置します。
2. サーバおよびプローブを開始します。
3. UCMDB で **［データ フロー管理］** > **［アダプタ管理］** > **［SCCMAdapter］** に移動します。
4. ［リソース］表示枠で，SCCM アダプタ構成ファイル（ **［パッケージ］** > **［SCCMAdapter］** > **［構成ファイル］** フォルダ）を選択します。
5. **adapter.conf** ファイルで，**dal.use.persistence.xml=true** を設定します。
6. **persistence.xml** ファイルに次を追加します。

```
<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">
    <persistence-unit name="GenericDBAdapter">
```

```
    <properties>
      <!-- added to fix:org.hibernate.HibernateException:'hibernate.dialect' must be set when no
Connection avalable -->
      <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>

      <!--ドライバ・クラス名"/-->
      <property name="hibernate.connection.driver_class"
value="net.sourceforge.jtds.jdbc.Driver"/>
      <property name="hibernate.connection.url" value="jdbc:jtds:sqlserver://<DB ホスト>:<ポー
ト>;DatabaseName=<DB 名>;domain=<ドメイン名> "/>
    </properties>
  </persistence-unit>
</persistence>
```

注 : 強調表示された部分を任意の接続 URL に置き換えます。

7. **persistence.xml** ファイルでは，ユーザまたはパスワードは不要です。
8. **［データ フロー管理］** > **［Integration Studio］** に移動し，**［新規統合ポイント］**  ボタンをクリックします。
9. 必須フィールドに値を指定します。

   資格情報 ID を入力する必要がある場合は，次の手順を実行します。

   a. ［資格情報の選択］ダイアログ・ボックスで，左側の［プロトコル］表示枠から［**Generid DB プロトコル（SQL）**］を選択します。

   b. 右側の［資格情報］表示枠で，**［選択したプロトコル タイプの新しい接続詳細を作成する］**  ボタンをクリックします。

   c. ［新規作成］ダイアログ・ボックスで，データベース・タイプとして［**MicrosoftSQLServerNTLM**］を選択します。

   d. ポート番号を入力します。

   e. 次の形式でユーザ名を入力します。**domain\username**

   f. パスワードを入力します。

定義済みのコンバータ

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

- 「PrefixTransformer コンバータ」(179ページ)
- 「BytesToStringTransformer コンバータ」(179ページ)
- 「StringDelimitedListTransformer コンバータ」(179ページ)
- 「カスタム・コンバータ」(179ページ)

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。

この変換では，大文字と小文字の区別，または大文字と小文字の区別なしの方法で 2 つの文字列比較を実行します。標準設定の動作では，大文字と小文字は区別されます。大文字と小文字の区別なしで定義するには，case-sensitive="false" を enum-transformer element に指定します。

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 :

```
<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:attribute name="case-sensitive" use="optional">
    <xs:simpleType>
        <xs:restriction base="xs:boolean">
```

```
          </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:attribute name="is-cmdb-value-null" type="xs:boolean" use="optional"/>
      <xs:attribute name="is-db-value-null" type="xs:boolean" use="optional"/>
    </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>
```

外部または CMDB 値を Null 値に変換する例：

この例では，リモート・データベースの **NNN** の値が CMDB データベースの null 値に変換されます。

```
<value cmdb-value="null" is-cmdb-value-null="true" external-db-value="NNN"/>
```

この例では，CMDBの **OOO** の値がリモート・データベースの null 値に変換されます。

```
<value cmdb-value="OOO" external-db-value="null" is-db-value-null="true"/>
```

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.AdapterToCmdb AddSuffixTransformer**.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.AdapterToCmdb RemoveSuffixTransformer**.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.AdapterToCmdb AddPrefixTransformer**.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.AdapterToCmdb RemovePrefixTransformer**.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。

このコンバータは次のとおりです。
com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.CmdbToAdapterBytes ToStringTransformer.

StringDelimitedListTransformer コンバータ

このコンバータは，単一の文字列リストを CMDB 内の整数 / 文字列リストに変換するのに使います。

このコンバータは次のとおりです。
com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl. StringDelimitedListTransformer

カスタム・コンバータ

It is possible to write 独自のカスタム・コンバータ（トランスフォーマ）をゼロから記述することができます。これにより，ニーズに合ったコンバータを作成できます。

カスタム・コンバータを記述するには，次の 2 つの方法があります。

1. コンパイルする Java コンバータを記述する
   a. Java IDE（Eclipse，Intellij，または Netbeans など）で Java プロジェクトを作成する。
   b. federation-api.jar および db-interfaces.jar をクラス・パスに追加します。
   c. 次のインタフェースを（**db-interfaces.jar** から）実装する Java クラスを作成します :
      - FcmdbDalTransformerFromExternalDB
      - FcmdbDalTransformerValuesToExternalDB
      - FcmdbDalTransformerInit
   d. プロジェクトをコンパイルし，jar ファイルを作成します。
   e. jar ファイルをアダプタのパッケージ（adapterCode\<Adapter ID>）に配置します。
   f. パッケージをデプロイします。
   g. 新しいコンバータ・クラス名を **transformations.txt** ファイルに追加します。
2. Groovy（スクリプト・ベース）コンバータを記述する

   元の GDBA パッケージに例として **GroovyExampleTransformer.groovy** があります。
   a. アダプタのパッケージ（adapterCode\<Adapter ID>）に Groovy ファイルを作成します。［アダプタ管理］メニューを使用すると，これを直接行うことができます。
   b. 次のインタフェースを（**db-interfaces.jar** から）実装する Groovy クラスを作成します。
      - FcmdbDalTransformerFromExternalDB
      - FcmdbDalTransformerValuesToExternalDB
      - FcmdbDalTransformerInit
   c. 新しいコンバータの Groovy クラス名を **transformations.txt** ファイルに追加します。

   注: Groovy は Java を拡張するスクリプト言語です。通常の Java コードは有効な Groovy コードでもあります。

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任意プラグイン。変更を同期するプラグインをまったく実装していない場合，差分同期の実行は可能ですが実際の同期は完全同期となります。
- 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。
- Back ID をプッシュする動作を変更する，内部的なオプションのプラグイン。

プラグインの実装とデプロイの詳細については，「プラグインの実装」(139ページ)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

ユース・ケース

TQL クエリは次のとおりです。

node > (composition) > card

詳細 :

- **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，「orm.xml ファイル」(155ページ)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_rules.txt ファイル

詳細については，「reconciliation_rules.txt ファイル（下位互換性用）」(168ページ)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 クエリは，**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_rules.txt ファイル

詳細については，「reconciliation_rules.txt ファイル（下位互換性用）」(168ページ)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， **「orm.xml ファイル」(155ページ)****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.impl.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.
GenericEnumTransformer(generic-enum-transformer-example.xml)] from_DB_class
[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(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

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 クエリと，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 があることがわかっていても，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，「JPA プロバイダとしての Hibernate」(121ページ)を参照してください。

外部参照

JavaBeans 3.0 仕様の詳細については，http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html（英語サイト）を参照してください。

トラブルシューティングおよび制限事項 - 汎用データベース・アダプタの開発

本項では，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

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ない。
- 複合関係はサポートされていません。
- 外部の CI CMDBid 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。
- 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。

第6章: Java アダプタの開発

本章の内容

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。

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

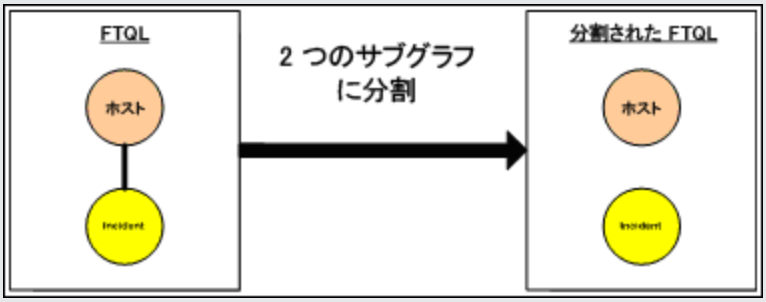
オンザフライ・フェデレーション

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。

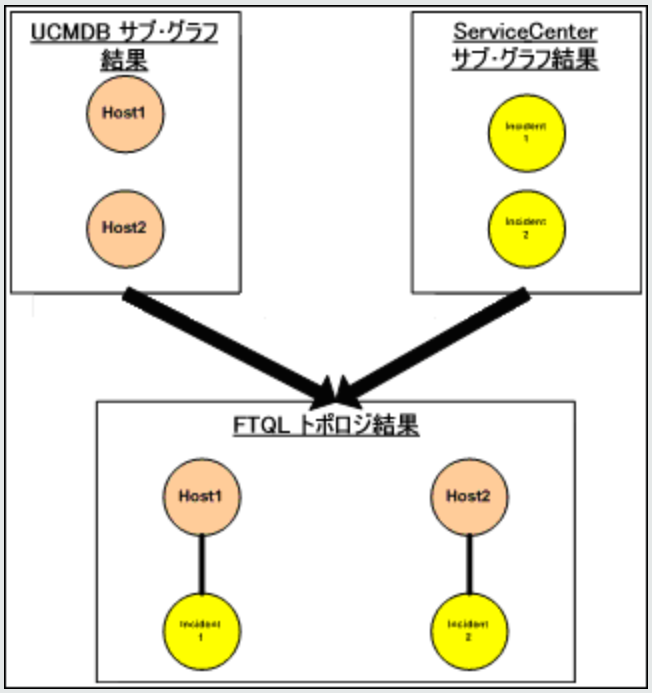
フェデレート TQL クエリは，外部データ・リポジトリを表すアダプタを使って，さまざまな外部データ・リポジトリからの CI と UCMDB CI の間に適切な外部関係を作成します。

オンザフライ連携フローの例：

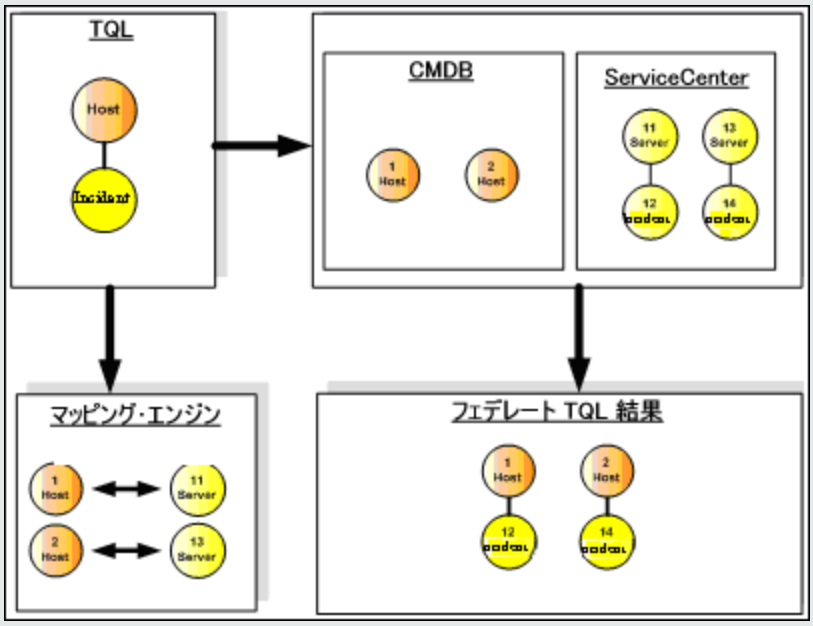
1. Federation Framework は，1 つのフェデレート TQL クエリを複数のサブグラフに分割します。サブグラフ内のすべてのノードは同じデータ・リポジトリを参照します。各サブグラフは，仮想関係によってほかのサブグラフに接続されます（ただし，サブグラフ自体に仮想関係は含まれていません）。



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）とターゲットの 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。

インスタンス・ベースのポピュレーション・フロー

アダプタがインスタンス・ベースのフローをサポートするように定義（「XML 設定タグとプロパティ」(219ページ)で説明されている **<instance-based-data>** タグを使用して）されている場合は，ポピュレーション・エンジンはインスタンス内で削除された CI を自動的に見つけ，それを UCMDB から削除します（削除が特定のポピュレーション・ジョブで許可されている場合）。各インスタンスに

は，TQL 定義に名前 **Root** で印を付けたルート CI が必要です。ルート CI が渡されるごとに，そのインスタンス全体（それに接続されたすべての CI）が最後に UCMDB に送信されたものと比較され，ルートに接続されていた CI で，今は接続されていないものが UCMDB から削除されます。アダプタがインスタンス・ベースのフローを正しくサポートするには，インスタンス全体にある CI や属性の変更により，インスタンス全体の UCMDB への再送信がトリガされなければなりません。

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

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

本項の内容

Federation Framework，UCMDB，アダプタ，マッピング・エンジン間のやり取りを示す図については，「Federation Framework，サーバ，アダプタ，マッピング・エンジン間のやり取り」(199ページ)を参照してください。

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，外部データ・リポジトリから 2 種類のデータを取得します。1 つは外部 CI データであり，もう 1 つは外部 CI に属する調整オブジェクトです。

- **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，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。

Federation Framework，サーバ，アダプタ，マッピング・エンジン間のやり取り

次の図は，Federation Framework，UCMDB，サーバ，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。

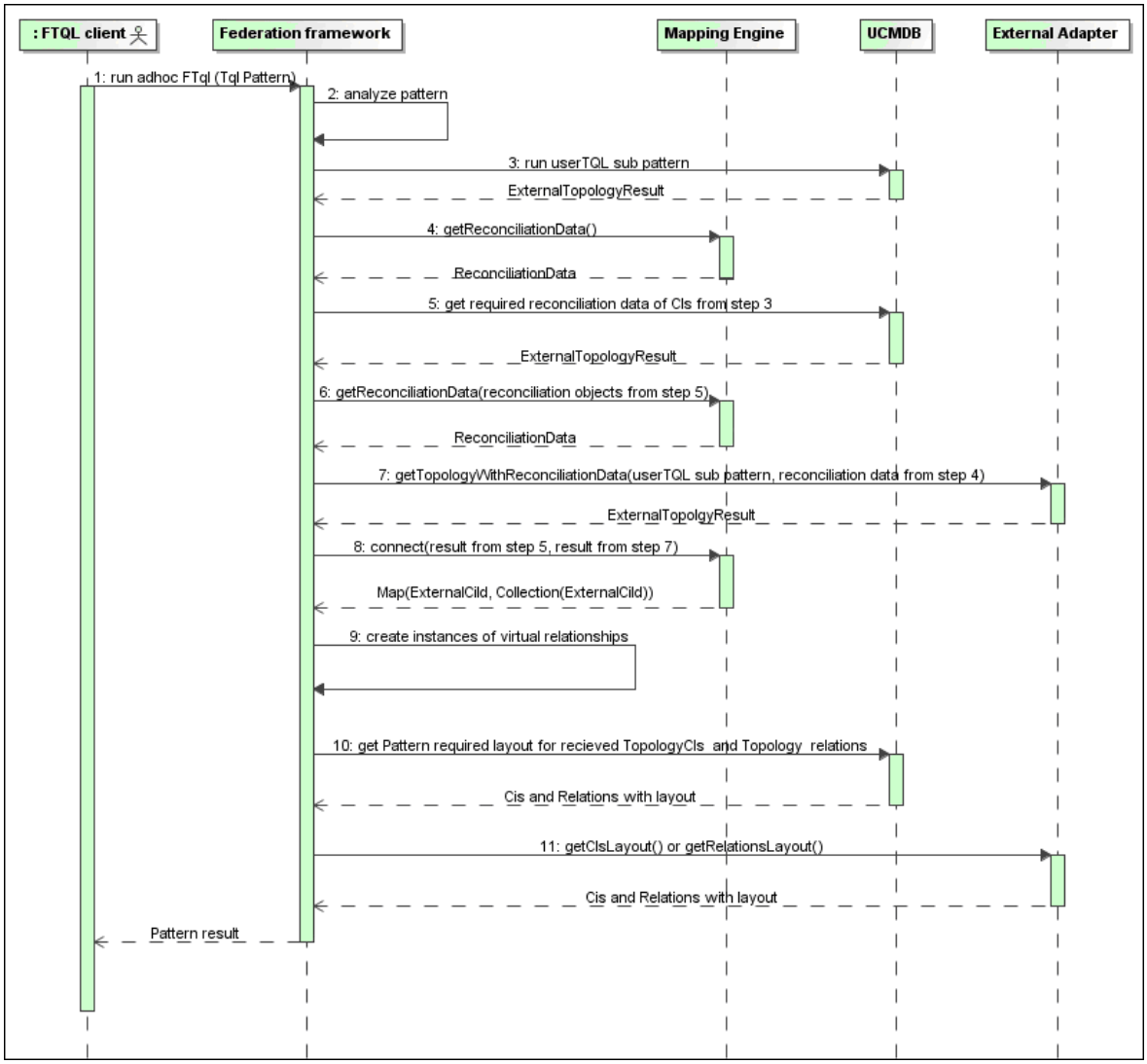
本項の内容

- 「サーバ・エンドで開始される計算」(199ページ)
- 「外部アダプタ・エンドで開始される計算」(202ページ)
- 「フェデレート TQL クエリ用の Federation Framework フローの例」(203ページ)

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。

サーバ・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，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。

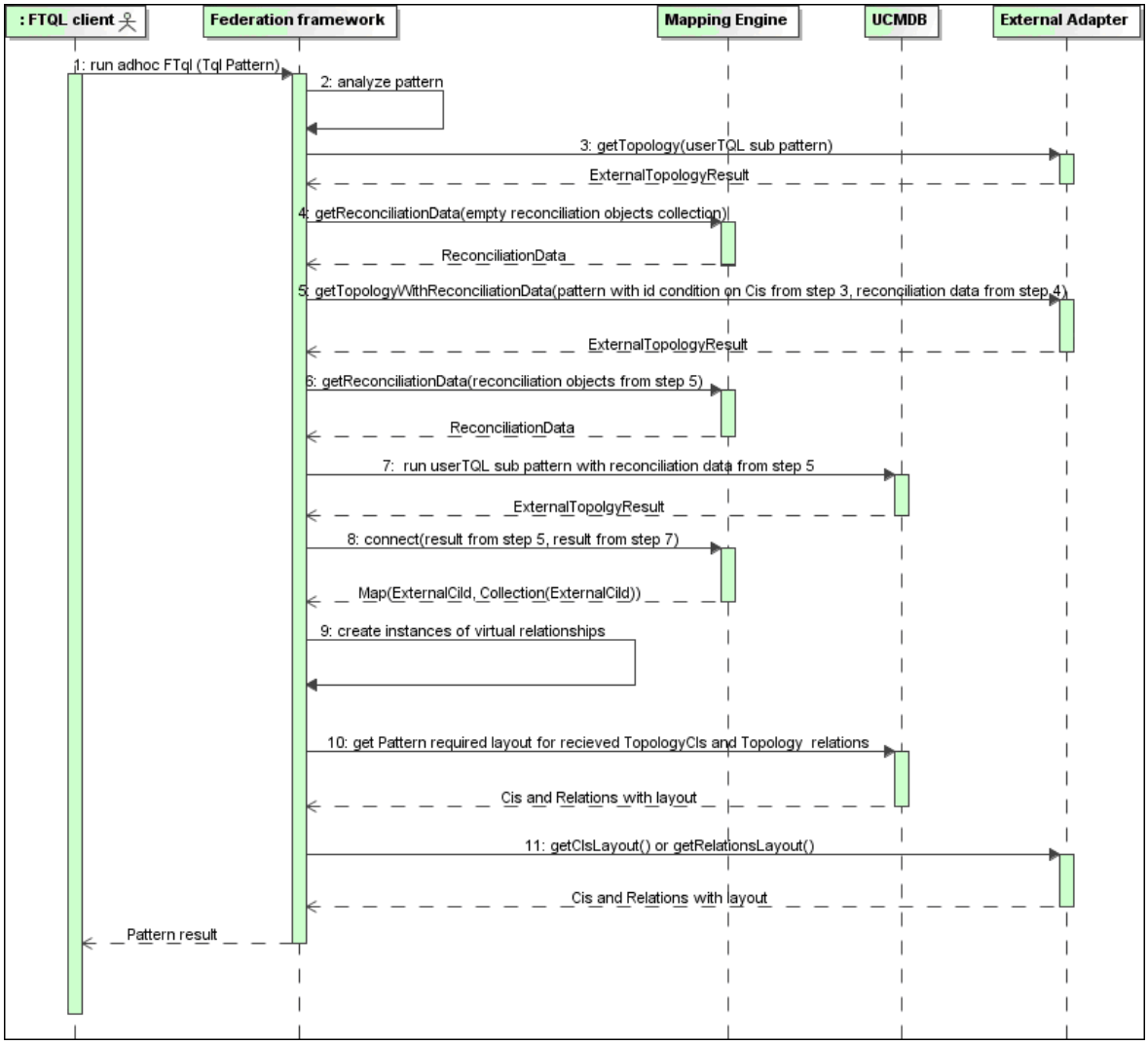


以下では，この図の各番号について説明します。

番号	詳細
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・

番号	詳細
	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 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」(200ページ) 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」(200ページ)3 と比較して），マッピング・エンジンは手順「5」(201ページ) 3 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，手順の「6」(201ページ)調整データをパラメータとして受け取ります。
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」(201ページ) 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」(201ページ) 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。

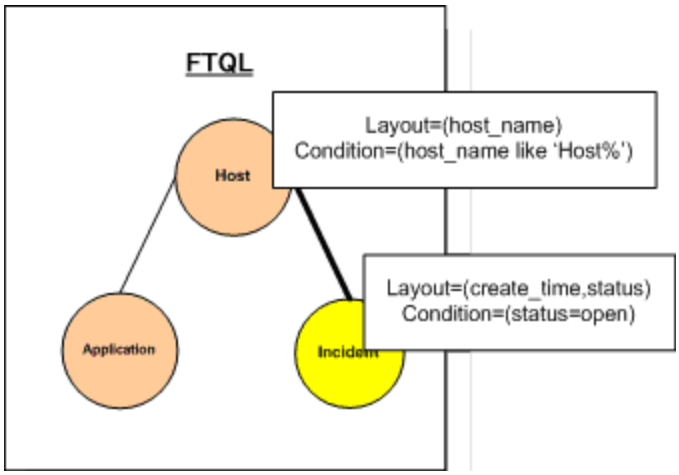
番号	詳細
1	Federation Framework は，フェデレート TQL 計算の呼び出しを受け取ります。
2	Federation Framework はアダプタを分析し，仮想関係を検出し，元の TQL を UCMDB 用と外部データ・リポジトリ用の 2 つのサブアダプタに分割します。
3	Federation Framework は，外部アダプタにサブ TQL のトポロジを要求します。要求には調整データが含まれないため，返された ExternalTopologyResult に調整オブジェクトが含まれることはありません。
4	Federation Framework は，トポロジ結果を受け取った後，現在の仮想関係を使って適

番号	詳細
	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 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。
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

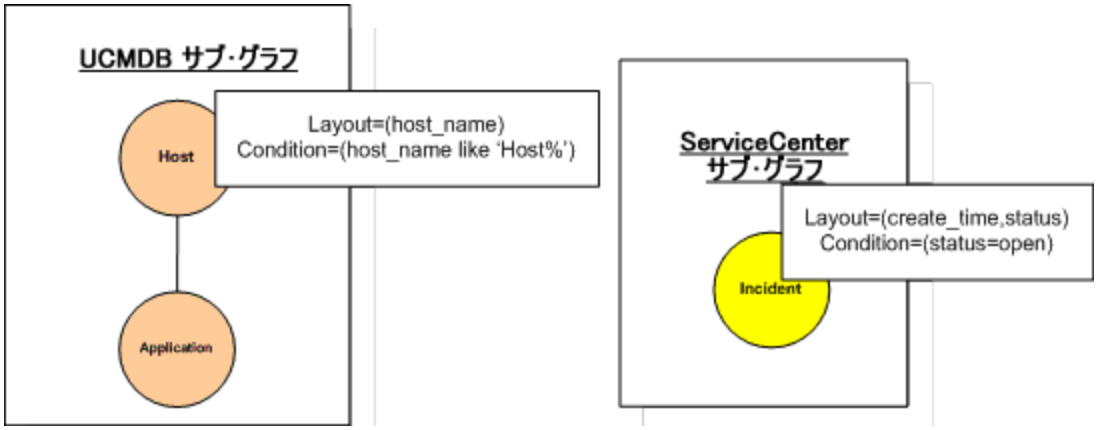
この例では，特定のノード上で開いているすべてのインシデントを表示する方法について説明します。ServiceCenter データ・リポジトリは外部データ・リポジトリです。ノード・インスタンスは UCMDB に格納され，インシデント・インスタンスは ServiceCenter に格納されています。インシデン

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。

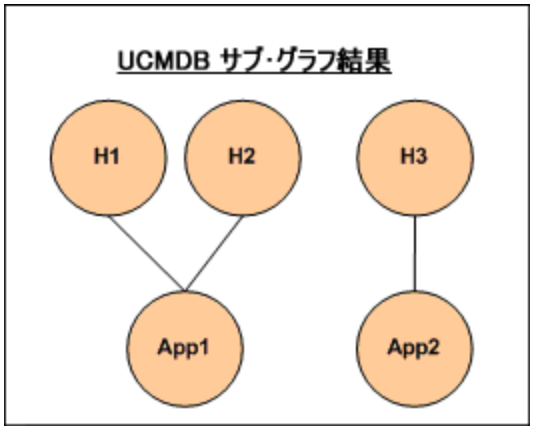


注: 属性のフェデレーションのために，アダプタの**getTopology** メソッドが呼び出されます。調整データは，ユーザ TQL（この場合は CI 要素）で調整されます。

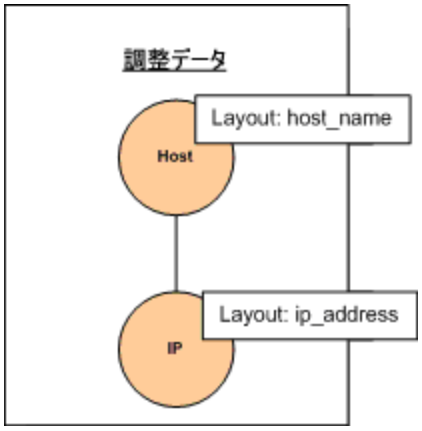
1. Federation Framework は，アダプタを分析した後で，Node と Incident の仮想関係を認識し，フェデレート TQL クエリを次の 2 つのサブグラフに分割します。



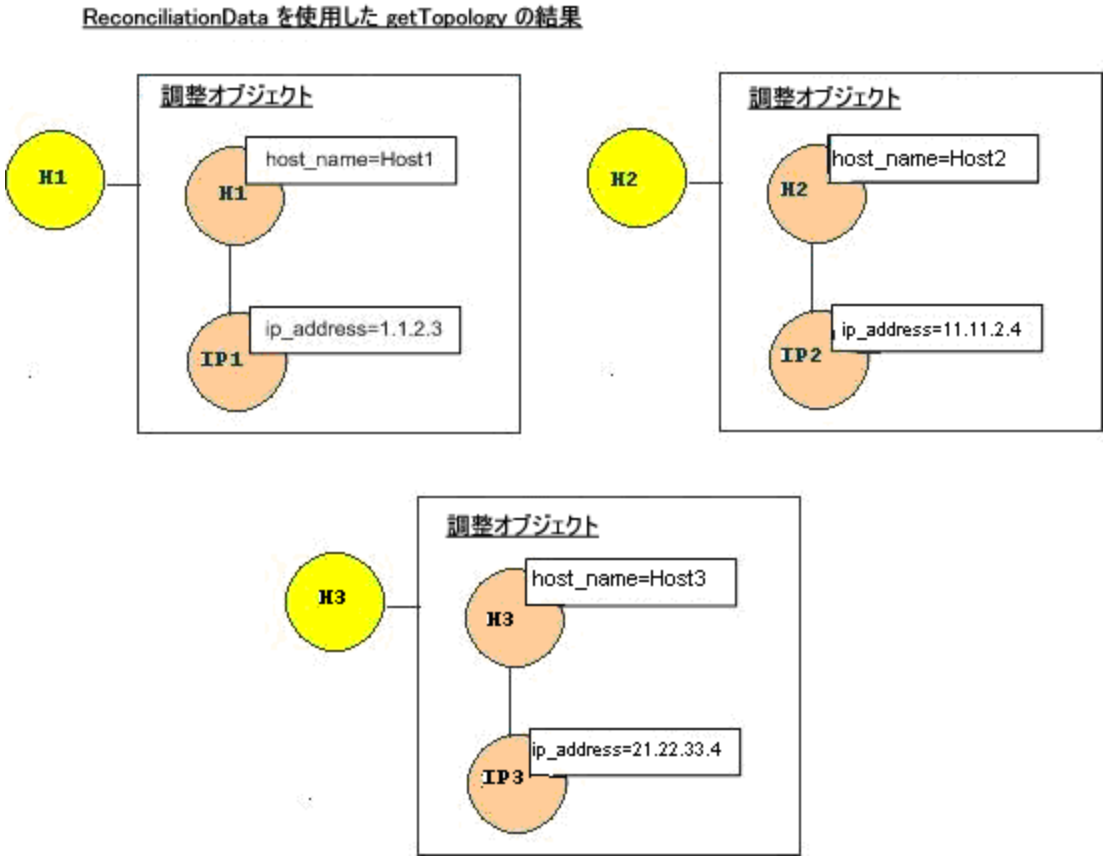
2. Federation Framework は，UCMDB サブグラフを実行してトポロジを要求し，次の結果を受け取ります。



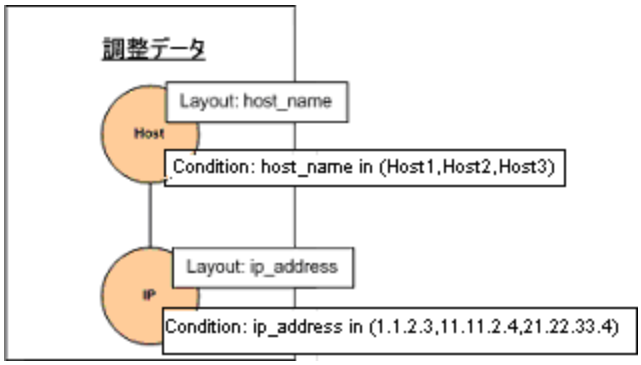
3. Federation Framework は，2 つのデータ・リポジトリから受け取ったデータ同士を接続するための情報を含む 1 つ目のデータ・リポジトリ（UCMDB）の調整データを，適切なマッピング・エンジンに要求します。この場合の調整データは次のとおりです。



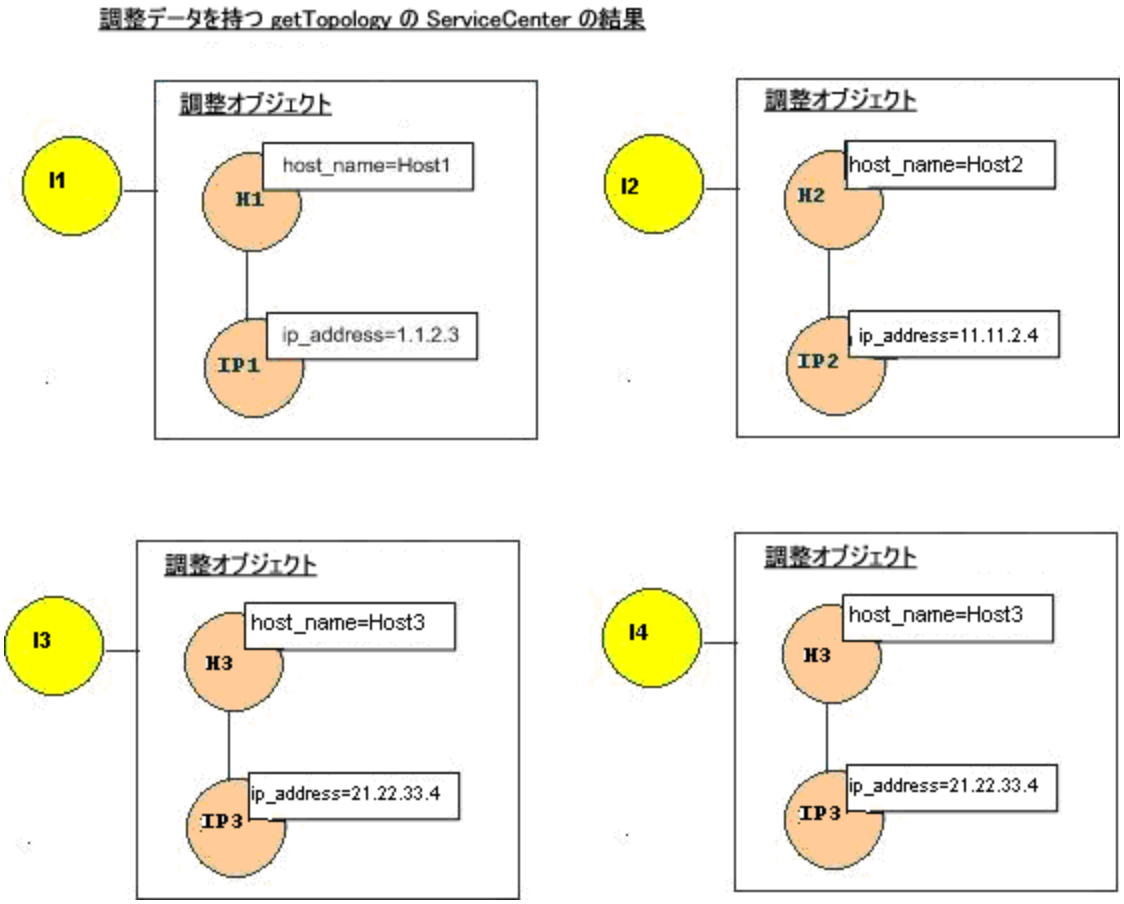
4. Federation Framework は，前の結果（H1，H2，H3 の node）から，ノードとそれに対する ID 条件を含む 1 ノード・トポロジを作成し，そのクエリを UCMDB 上の必要な調整データとともに実行します。この結果には，ID 条件に関連する Node CI と，各 CI の適切な調整オブジェクトが含まれています。



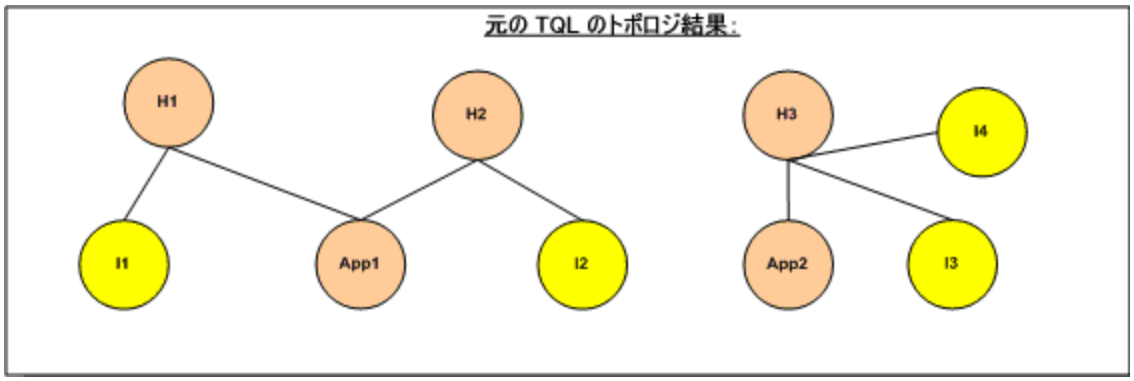
5. ServiceCenter の調整データには，UCMDB から受け取った調整オブジェクトから派生された node と ip の条件が含まれています。



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

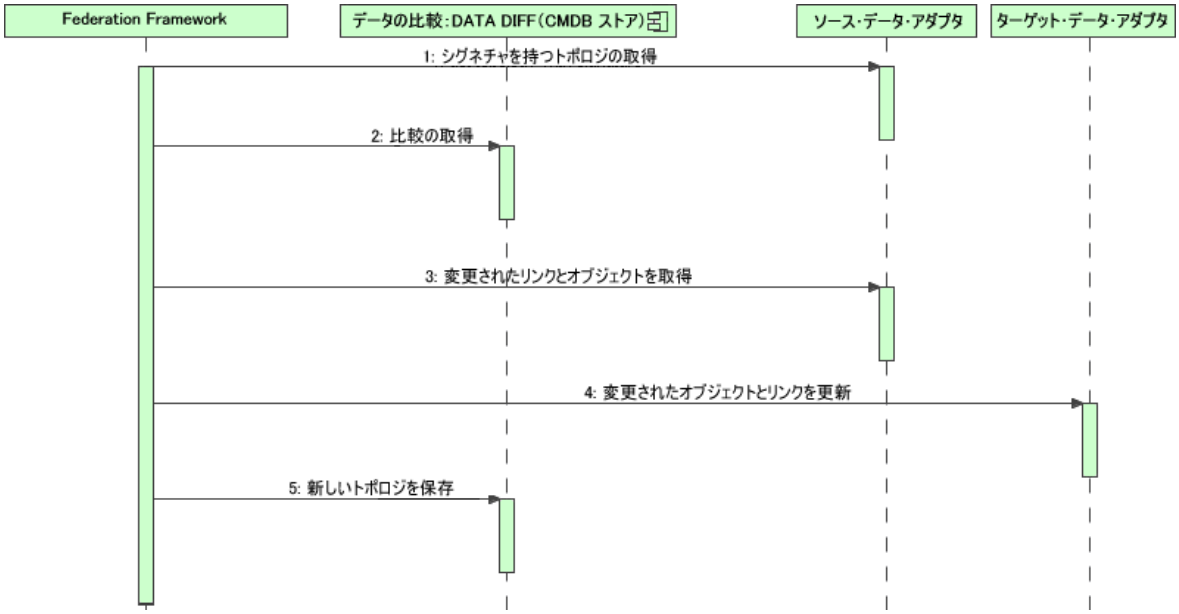
本項の内容

定義と用語

シグネチャ：CI プロパティの状態を示します。CI 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，新しく受けとったトポロジ・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，**単一 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。操作を完了するために必要な調整データは，複合クエリとして記述できます（下の SingleNodeFederationTopologyReconciliationAdapter を参照してください）。

  すべての単一ノード・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，**FederationTopologyAdapter** インタフェースを使用します。
- 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，**FederationTopologyAdapter 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，**QueryDefinition** パラメータの一部です。

  フェデレーション・エンジンは，フェデレート・データを適切なローカル CI に接続するために調整データを使用します。調整データは（結果に応じて繰り返し計算される）複数の要求で取得される場合があります。この場合，アダプタは調整データのみを持った要求を受け取ります。

単一ノード・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。

- **SingleNodeFederationIdReconciliationAdapter:** 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。
- **SingleNodeFederationPropertyReconciliationAdapter:** 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 プロパティによって行われる場合に使用します。
- **SingleNodeFederationTopologyReconciliationAdapter:** 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。アダプタは，クエリ要素が空白で，調整トポロジのみが要求される場合をサポートする必要があります。

データ・アダプタ・インタフェース

- **FederationTopologyAdapter:** 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。アダプタは，クエリ定義が調整データのみを記述している場合をサポートする必要があります。
- **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。

注: 大規模なデータ・セットのデータを返す可能性があるアダプタを開発する場合，ChunkGetter インタフェースを実装することで，データをチャンクに分割できるようにすることが重要です。詳細については，特定のアダプタの Java ドキュメントを参照してください。

リソース・レポーティング・インタフェース

次のインタフェースでは，アダプタはアダプタの動作をカスタマイズする設定が可能なリソースをレポートできます。これにより，Integration Studio から直接これらのリソースを編集できます。これらのインタフェースは，上記の通常のアダプタ・インタフェースに加えて使用する必要があります。

- **PopulationQueriesResourcesLocator:** 特定のポピュレーション・クエリのそれぞれに対して編集されるリソースを定義します。
- **PushQueriesResourceLocator:** データ・プッシュ・クエリのそれぞれに対して編集されるリソースを定義します。
- **GeneralResourcesLocator:** このアダプタで編集される一般的なリソースを定義します。

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。

デバッグ・アダプタのリソース

このタスクでは，デバッグと開発の目的で，JMX コンソールを使用してアダプタ状態のリソース（UCMDB データベースまたはプローブ・データベースに格納されている DataAdapterEnvironment インタフェースのリソース操作方法を使用して作成したリソース）を作成，表示，削除する方法について説明します。

1. Web ブラウザを起動し，サーバ・アドレスとして次を入力します。
   - UCMDB サーバの場合 : http://localhost:8080/jmx-console
   - プローブの場合 :http://localhost:1977

   ユーザ名とパスワードを使用してログインする必要がある場合もあります。
2. ［JMX MBEAN View］ページを開くには，次のいずれかを行います。
   - UCMDB サーバの場合 : 次をクリックします。
     UCMDB:service=FCMDB Adapter State Resource Services
   - プローブ の場合 :次をクリックします : **type=AdapterStateResources**
3. 使用する操作に値を入力し，［**Invoke**］をクリックします。

新しい外部データ・ソース用アダプタの追加

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。

本項の内容

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，次の作業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る）

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，『HP Universal CMDB モデリング・ガイド』の「CI セレクタでの作業」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: 本項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 Universal CMDB 管理ガイド』の「」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. アダプタ構成の定義

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，［**新規アダプタ**］を選択します。
c. ［新規アダプタ］ダイアログ・ボックスで，［**統合**］および［**Java アダプタ**］を選択しま

す。

d. 作成したアダプタを右クリックし，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"
enableDebugging="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.BaseDestinationDa
ta">
<!-- check -->
<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/< >!—see the section about supported
classes-->
<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>
<!--The description attribute may be written in simple text or
HTML.-->
<!--The host attribute is treated as a special case by UCMDB-->
<!--and will automatically select the probe name (if possible)-
->
<!--according to this attribute’s value.-->
<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，「XML 設定タグとプロパティ」(219ページ)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>
```

name	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。

アダプタが **getTopologyWithReconciliationData** を実装し，アダプタの機能に起点として使用される機能が含まれている場合，アダプタは調整データを持ったトポロジの要求を無条件で（タイプのみ）サポートする必要があります。この場合，アダプタは見つかった結果の全調整データを返す必要があります。

アダプタ調整サポートは **global_id** に従って定義できます。この場合 **global_id** はアダプタがサポートしているクラスの調整属性の一部として定義されている必要があります。アダプタ調整サポートが **global_id** に従って定義されている場合，**getTopologyWithReconciliationData()** は調整オブジェクト・プロパティの一部として **global_id** を返します。UCMDB は **global_id** を，識別ルールではなく CIT の連携結果の調整に使用します。

フェデレーション API の一部は DataAdapterEnvironment インタフェースです。このインタフェースはデータ・アダプタの環境を表します。これには，アダプタが機能するために必要な環境 API が含まれています。DataAdapterEnvironment インタフェースの詳細については，「DataAdapterEnvironment インタフェース」(221ページ) を参照してください。

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 2 つのオプションがあります。

- デフォルトの マッピング・エンジンを使用します。このエンジンはマッピングに CMDB の内部調整ルールを使用します。これを使用するには，**<default-mapping-engine>** XML タグを空のままにします。
- 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，次の XML タグを使用します : **<default-mapping-engine>com.yourcompany.map.MyMappingEngine</default-mapping-engine>**

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

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 Universal CMDB 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，アダプタ構成の値と同じです。

- 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 Universal CMDB 管理ガイド』の「パッケージのデプロイ方法」を参照してください。

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。本項の内容

- 「アダプタ・ロジックの選択」(218ページ)
- 「プロジェクトの読み込み」(219ページ)

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. 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ます。

   MyAdapter の例では，オプション *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

ファイルを **C:\hp\UCMDB\UCMDBServer\tools\adapter-dev-kit\SampleAdapters** フォルダからコピーし，readme ファイルの指示に従います。

注: 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。

オンラインの Javadoc ドキュメントは，次で入手できます。

C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DBAdapterFramework_JavaAPI\index.html

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>		アダプタのカテゴリを定義します。
<parameters>		新しい統合ポイントを設定するときに UI で利用可能な設定のプロパティを定義します。
	name	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 ; で区切られた利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。
	<start-point-adapter>	このアダプタが TQL クエリ計算の起点であることを指定します。
	<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。
	<push-back-ids>	CI のグローバル ID をテーブルの global_id カラムにプッシュバックします（orm.xml で定義されていなければなりません）。この動作は **FcmdbPluginPushBackIds** プラグインを実行することでオーバーライドされます。
	<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。
	<instance-based-data>	このタグは，アダプタがインスタンス・ベースのポピュレーション・フローをサポートすることを定義します。
	<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
	<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。

	<adapter-settings>	アダプタの設定リスト
	<list.attributes.for.set>	どの属性が以前の値を上書きするかを決定します（存在する場合）。

DataAdapterEnvironment インタフェース

OutputStream openResourceForWriting(String resourceName) throws FileNotFoundException;

このメソッドは，与えられた名前を持ったリソースを書き込みのために開きます。これは，統合のための継続的なデータを保存するために使用されます。このメソッドは，java メソッドを使用してファイルをロードする代わりに使用します。ユーザは，ストリームへの書き込みが終了しときに，ストリームが閉じられていることを確認する必要があります。close()/flush() はリソースを保存します。このメソッドはランタイム・リソースを作成します（アダプタ・パッケージに入れたファイルを上書きしない場合があります）。

パラメータ

- **resourceName:** 取得するリソースの名前。この名前は同じアダプタのすべての統合で一意でなければばりません。

戻り値

書き込むストリームを返します。

例外

- リソース・タイプがファイルで，ファイルが存在しない場合，リソースが通常のファイルではなくディレクトリの場合，またはその他の理由でリソースが読み取りのために開くことができない場合に，このメソッドは *FileNotFoundException* をスローします。
- セキュリティ・マネージャが存在し，その *checkRead* メソッドがファイルへのアクセスを拒否する場合に，このメソッドは *SecurityException* をスローします。

InputStream openResourceForReading(String resourceName) throws FileNotFoundException;

このメソッドは，与えられた名前を持ったリソースを読み取りのために開きます。これは，統合のための継続的なデータを読取るために使用されます。このメソッドは，java メソッドを使用してファイルをロードする代わりに使用します。ユーザは，ストリームへの読み取りが終了しときに，ストリームが閉じられていることを確認する必要があります。これは初めにアダプタ・パッケージに入ったファイルのロードを試みます。ファイルが見つからない場合は，*DataAdapterEnvironment.openResourceForWriting(String)* により作成されたランタイム・リソースの

ロードを試みます。ランタイム・リソースは，（プローブおよびサーバの）JMX を使用すると表示できます。

パラメータ

- **resourceName:** 取得するリソースの名前。この名前は同じアダプタのすべての統合で一意でなければばりません。

戻り値

読み取るストリームを返します。

例外

- リソース・タイプが**ファイル**で，ファイルが存在しない場合，リソースが通常のファイルではなくディレクトリの場合，またはその他の理由でリソースが読み取りのために開くことができない場合に，このメソッドは *FileNotFoundException* をスローします。
- セキュリティ・マネージャが存在し，その *checkRead* メソッドがファイルへの読み取りアクセスを拒否する場合に，このメソッドは *SecurityException* をスローします。

Properties openResourceAsProperties(String propertiesFile) throws IOException;

このメソッドは与えられた名前を持ったリソースを開き，*Properties* 構造としてロードします。これは，統合のための継続的なデータを読取るために使用されます。このメソッドは，java メソッドを使用して **.properties** ファイルをロードする代わりに使用します。これは初めにアダプタ・パッケージに入ったファイルのロードを試みます。ファイルが見つからない場合は，*DataAdapterEnvironment.openResourceForWriting(String)* により作成されたランタイム・リソースのロードを試みます。ランタイム・リソースは，（プローブおよびサーバの）JMX を使用すると表示できます。

パラメータ

- **propertiesFile:** 取得するリソースの名前。この名前は同じアダプタのすべての統合で一意でなければばりません。

戻り値

Properties で表されたファイルの内容を返します。

例外

- リソース・タイプが**ファイル**で，ファイルが存在しない場合，リソースが通常のファイルではなくディレクトリの場合，またはその他の理由でリソースが読み取りのために開くことができない場合に，このメソッドは *FileNotFoundException* をスローします。
- セキュリティ・マネージャが存在し，その *checkRead* メソッドがファイルへの読み取りアクセスを拒否する場合に，このメソッドは *SecurityException* をスローします。
- properties ファイルが *Properties* オブジェクトへの変換に失敗した場合は，このメソッドは *IOException* をスローします。

String openResourceAsString(String resourceName) throws IOException;

このメソッドは与えられた名前を持ったリソースを開き，それを文字列としてロードします。これは，統合のための継続的なデータを読取るために使用されます。このメソッドは，java メソッドを使用してファイルをロードする代わりに使用します。

これは初めにアダプタ・パッケージに入ったファイルのロードを試みます。ファイルが見つからない場合は，*DataAdapterEnvironment.openResourceForWriting(String)* により作成されたランタイム・リソースのロードを試みます。ランタイム・リソースは，（プローブおよびサーバの）JMX を使用すると表示できます。

パラメータ

- **resourceName:** 取得するリソースの名前。この名前は同じアダプタのすべての統合で一意でなければばりません。

戻り値

文字列形式で表されたファイルの内容を返します。

例外

- リソース・タイプが**ファイル**で，ファイルが存在しない場合，リソースが通常のファイルではなくディレクトリの場合，またはその他の理由でリソースが読み取りのために開くことができない場合に，このメソッドは *FileNotFoundException* をスローします。
- セキュリティ・マネージャが存在し，その *checkRead* メソッドがファイルへの読み取りアクセスを拒否する場合に，このメソッドは *SecurityException* をスローします。
- I/O エラーが発生すると，このメソッドは *IOException* をスローします。

public void saveResourceFromString(String relativeFileName, String value) throws IOException;

このメソッドは文字列を受け取り，それをリソースとして保存します。これは，統合のための継続的なデータを保存するために使用されます。このメソッドは，java メソッドを使用してファイルを保存しようとする代わりに使用します。このメソッドは文字列をストリームに変換して，それをリソースに保存します。これはランタイム・リソースを作成しますが，アダプタ・パッケージに入れたファイルを上書きすることはできません。ランタイム・リソースは，（プローブおよびサーバの）JMX を使用すると表示できます。

パラメータ

- **relativeFileName:** 取得するリソースの名前。この名前は同じアダプタのすべての統合で一意でなければばりません。
- **値 :** リソースとして保存される文字列

例外

I/O エラーが発生すると，このメソッドは IOException をスローします。

boolean resourceExists(String resourceName);

このメソッドは，与えられたリソース名が存在するかどうかをチェックします。これはアダプタ・パッケージに入ったファイルおよび *DataAdapterEnvironment.openResourceForWriting(String)* により作成されたランタイム・リソースを探します。

パラメータ

- **resourceName:** 取得するリソースの名前。この名前は同じアダプタのすべての統合で一意でなければばりません。

戻り値

resourceName が存在する場合は，**True** を返します。

boolean deleteResource(String resourceName);

このメソッドは与えられたリソースを継続的なデータから削除します。これはランタイム・リソースを削除しますが，アダプタ・パッケージに入れたファイルを削除できません。ランタイム・リソースは，（プローブおよびサーバの）JMX を使用すると表示できます。

パラメータ

- **resourceName:** 削除するリソースの名前。この名前は同じアダプタのすべての統合で一意でなければばりません。

戻り値

リソースが正常に削除されると，**True** を返します。

Collection<String> listResourcesInPath(String path);

このメソッドは与えられたリソース・パスにあるリソースのリストを取得します。これはアダプタ・パッケージに入ったファイルおよび *DataAdapterEnvironment.openResourceForWriting(String)* により作成されたランタイム・リソースを探します。ランタイム・リソースは，（プローブおよびサーバの）JMX を使用すると表示できます。

パラメータ

- **パス :** リソース・パス。例，"META-INF/myfiles/"

戻り値

パスにあるリソースのリストを返します。

DataAdapterLogger getLogger();

アダプタが使用するロガーを取得します。このロガーは，アダプタがイベントのログ作成に使用します。

戻り値

DataAdapter が使用するロガーを返します。

DestinationConfig getDestinationConfig();

このメソッドは統合の宛先設定を取得します。この設定には統合のすべての接続と実行設定が含まれています。

戻り値

アダプタの DestinationConfig を取得します。

int getChunkSize();

このメソッドはこの統合のポピュレーション・チャンク・サイズを取得します。

戻り値

ポピュレーション・チャンク・サイズを返します。

int getPushChunkSize();

このメソッドはこの統合のプッシュ・チャンク・サイズを取得します。

戻り値

プッシュ・チャンク・サイズを返します。

ClassModel getLocalClassModel();

このメソッドは，ローカル UCMDB のクラス・モデルについての情報をクエリするクラス・モデルを取得します。このメソッドは更新された ClassModel を取り込みます。ClassModel オブジェクトが返されると，任意のクラス・モデルの変更に対して更新されません。更新されたクラス・モデルを取得するには，このメソッドを再び使用して取得します。

戻り値

UCMDB のクラス・モデルを返します。

CustomerInformation getLocalCustomerInformation();

このメソッドは，アダプタを実行している顧客の顧客情報を取得します。

戻り値

アダプタを実行している顧客の顧客情報を返します。

Object getSettingValue(String name);

このメソッドは特定のアダプタ設定を取得します。

パラメータ

名前 : 設定の名前。

戻り値

オブジェクト設定値を返します。

Map<String, Object> getAllSettings();

このメソッドはすべてのアダプタ設定を取得します。

戻り値

アダプタ設定を返します。

boolean isMTEnabled();

このメソッドは，サーバ環境がマルチ・テナンシー（MT）をサポートしているかどうかをチェックします。

戻り値

サーバ環境が MT をサポートしている場合は，**true** を返し，そうでない場合は **false** を返します。

String getUcmdbServerHostName();

このメソッドはローカル UCMDB サーバ・ホスト名を返します。

戻り値

ローカル UCMDB サーバ・ホスト名を返します。

第7章: プッシュ・アダプタの開発

本章の内容

プッシュ・アダプタの開発とデプロイ

汎用プッシュ・アダプタは，UCMDB データを外部データリポジトリ（データベースやサードパーティ・アプリケーション）にプッシュする統合の迅速な開発を可能にする共通のプラットフォームを提供します。汎用プッシュ・アダプタは，データのプッシュに使用されるプロトコルに応じて分類されます。汎用 XML プッシュ・アダプタを使用した，XML によるプッシュの詳細については，「汎用 XML プッシュ・アダプタ SQL クエリ」(240ページ)を参照してください。汎用 Web Service プッシュ・アダプタを使用した，Web Service によるプッシュの詳細については，「汎用 Web Service プッシュ・アダプタ」(240ページ)を参照してください。

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。

- 適切な汎用プッシュ・アダプタのテンプレート・ファイルから新しいアダプタ・パッケージを作成します。詳細については，「アダプタ・パッケージの作成」(228ページ)を参照してください。
- UCMDB CI リンク・タイプと外部データ項目間をマッピングします。マッピングは XML として保存され，各外部データ・リポジトリに対してカスタマイズされます。詳細については，「マッピングの作成」(231ページ)を参照してください。
- 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。詳細については，「Jython スクリプトの記述」(234ページ)を参照してください。
- その他のアダプタ固有の手順。たとえば，XML プッシュ・アダプタ用に記述されるファイルのパスの選択や，Web Service プッシュ・アダプタ用のデータ・レシーバの作成など。

アダプタ・パッケージの作成

新しい MDR 固有のプッシュ・アダプタを作成するには，汎用アダプタのコピーを作成し，特定のプッシュ・ターゲットのアダプタになるようにカスタマイズします。

汎用アダプタ・パッケージは次の 2 つの場所のいずれかで見つけます。

- 汎用 XML プッシュ・アダプタ :**hp\UCMDB\UCMDBServer\content\adapters\push-adapter.zip**
- 汎用 Web Service アダプタ :**hp\UCMDB\UCMDBServer\content\adapters\web-service-push-adapter.zip**

汎用プッシュ・アダプタから新しいプッシュ・アダプタを作成するには

1. 選択したパッケージの zip ファイルの内容を作業フォルダに展開します。
2. 名前の変更と置き換え段階の準備のため次のディレクトリを次のようにレビューします。
   - **adapterCode: C:\hp\UCMDB\UCMDBServer\runtime\fcmdb\CodeBase directory** にデプロイされたディレクトリが含まれています。ここにデプロイされた Jar はプローブを自動的に再起動せず，プローブの CLASSPATH に自動的に表示されません。
   - **discoveryConfigFiles:** アダプタのマッピング定義が含まれており，正しい Jython スクリプト（**push.properties**）をポイントします
   - **discoveryPatterns:** UCMDB サーバにデプロイされるアダプタの XML 定義が含まれています
   - **discoveryScripts:** サードパーティ・データ・ストアへの接続が行われ，データがプッシュされるアダプタの Jython スクリプトが含まれています
   - **discoveryResources:** Web Service の Java 統合クラスを含んだ **UCMDBDataReceiver.jar** が含まれています。

   注: このパッケージをデプロイすると，プローブが再起動され，この **.jar** をプローブの CLASSPATH に含めます。パッケージのデプロイ以外にアクションは不要です。
3. 解凍されたアダプタ・ディレクトリ構造で次の変更を行います。
   a. **discoveryConfigFiles\<Your_Push_Adapter_Name>:**ディレクトリ "PushAdapter" または "XMLtoWebService" を新しいプッシュ・アダプタの名前に変更します（たとえば，“myPushAdapter” など）。
   b. **discoveryConfigFiles\<Your_Push_Adapter_Name>\push.properties:push.properties** ファイルで以下を行います。
      - **jythonScript.name** の名前を新しいプッシュ・アダプタによって使用される Jython スクリプトの名前に更新します（たとえば，**pushToMyService.py** など）。
      - マッピング・ファイルの名前を更新して，新しいプッシュ・アダプタによって使用されるようにします（たとえば，myPushAdapter_mappings など）。**.xml** 拡張子を加えません。これは自動的に入力されます。

c. **discoveryPatterns\<プッシュ・アダプタ名>.xml:** このファイルを新しいアダプタの定義 XML ファイルの名前に変更します（たとえば，**my_push_adapter.xml** など）。

d. **discoveryPatterns\<your_push_adapter>.xml:** このファイルを次のように更新します。

- XML 要素 **<pattern>:** id と説明の属性をそれぞれ設定します。 たとえば，

  <pattern id="PushAdapter" xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="Discovery Pattern Description" schemaVersion="9.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  を次のように変更します :

  <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">

- XML 要素 **<parameters>:** アダプタの必要に応じて子要素を更新します。標準設定では，次の子要素がプッシュ・アダプタを定義するために使用されます。これらの値は，アダプタが設定された後で，統合ポイントが Integration Studio で定義されるときに割り当てられます。パラメータのリストに要求された接続属性が反映されるように，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。
  - **host:** Web Service をホストするサーバ名
  - **port:** UCMDB Data Receiver サービスをリッスンするポート
  - **Web Service Push Adapter:uri** - データ・レシーバのサービス・エンドポイント・アドレスを形成するための URL の残り。
  - **probeName:** プッシュ・ジョブが実行される Data Flow Probe を定義します
- XML 要素 **<integration>:** 子要素 **<category>** の値を Generic 以外のものに更新します。標準設定では，Generic カテゴリに属する統合アダプタは Integration Studio に表示されません。サードパーティのデータ・ストアで統合を行う場合は，この値を「Third Party」に設定します。HP BTO 製品で統合を行う場合は，この値を「HP BTO Products」に設定します。

e. **adapterCode\PushAdapter:** このフォルダの名前を前の手順で使用したアダプタ ID で変更します（たとえば，**adapterCode\MyPushAdapter** など）。

f. **discoveryScripts\<your_Jython_push_script>.py:push.properties jythonScript.name** プロパティで定義した同じ名前でファイルを作成します。**discoveryScript** ファイルには，CI を挿入し，外部 Oracle データベースにリンクするスクリプトがあります。**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，「Jython スクリプトの記述」(234ページ)を参照）。スクリプトの名前を変更する場合は，**adapterCode\<adapter ID>\push.properties** の **jythonScript.name** プロパティをそれに応じて更新する必要があります。

- XML プッシュ・アダプタ :**pushScript.py**
- Web Service プッシュ・アダプタ :**XMLtoWebService.py**

g. **tql\<your_integration_TQLs>:**通常のパッケージと同様に，統合 TQL の TQL XML 定義をこのディレクトリに配置します。このフォルダ内のすべての TQL は，アダプタ・パッケージがデプロイされるときにデプロイされます。

h. **discoveryConfigFiles\<Your_Push_Adapter_Name>\mappings:**統合で使用する TQL ごとに XML マッピング・ファイルを作成します。プッシュ・アダプタはマッピング・ファイルの変換を統合 TQL の結果に適用し，一時的なタスクの 3 つのパラメータ（**addResult**, **updateResult**, **deleteResult**）にあるそのデータを Data Flow Probe に送信します。

i. **adapterCode\<adapter ID>\mappings:mappings.xml** ファイルを準備したマッピング・ファイルで置き換えます（詳細については，「マッピングの作成」(231ページ)を参照してください）。

XML プッシュ・アダプタ :このマッピング例は ORACLE の sql_queries ファイルで作成されたテーブルの例に対応します。

各 TQL メソッドのマッピング・ファイルを使用するには，対応する TQL の名前を各 XML ファイルに割り当て，ファイル名の後に **.xml** を付けます。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。

4. 上記の変更をすべて行ったら，上記の手順 3 で指定したフォルダとファイルを選択して **.zip** ファイルを作成します（たとえば，**my_Push_Adapter.zip** など）。
5. 新しく作成した **.zip** ファイルをパッケージ・マネージャ（**［管理］> ［パッケージ・マネージャ］**に移動）により UCMDB サーバにデプロイします。
6. **［データ フロー管理］> ［Integration Studio］**で統合ポイントを作成し，統合ポイントが使用する統合 TQL を定義します。自動データ・プッシュのスケジュールを設定します。

トラブルシューティング

新しいプッシュ・アダプタを作成する手順には，完全で正しい名前変更を置き換えが必要です。エラーがあるとアダプタに影響を与える可能性があります。パッケージが UCMDB パッケージとして動作するには，正しく解凍して，再度圧縮する必要があります。例として定義済みのパッケージを参照してください。一般的エラー :

- ZIP ファイルのパッケージ・ディレクトリの上に別のディレクトリが含まれている。
  解決策 : パッケージを **discoveryResources**, **adapterCode** などのパッケージ・ディレクトリと同じディレクトリで ZIP します。ZIP ファイルの上に別のディレクトリ・レベルを含めないようにします。
- ディレクトリ，ファイル，ファイルの文字列の重要な名前変更を忘れている。
  解決策 : 本項の説明に注意深く従います。
- ディレクトリ，ファイル，ファイルの文字列の重要な名前変更でのスペルミス。
  解決策 : 名前の変更手順を開始したら，途中で命名規則を変更しないようにします。名前の変更が必要なことに気付いた場合は，エラーの危険性が高くなるので前に遡って名前を訂正しようとせずに，完全にやり直します。また，エラーの危険性を減らすために，文字列を手動で置き換えようとせずに，検索と置換を使用します。

- アダプタを他のアダプタと同じファイル名でデプロイする。特に **discoveryResources** や **adapterCode** ディレクトリでデプロイする。
  解決策 : マッピング・ファイルが同じ UCMDB 環境で他のアダプタと同じ名前になるのを防ぐため，既知の問題を知った上で UCMDB バージョンを使用します。パッケージを重複した名前でデプロイしようとすると，パッケージのデプロイは失敗します。この問題は，これらのファイルが別のディレクトリにある場合でも発生する可能性があります。さらに，この問題は重複がパッケージ内や前にディプロイされたパッケージにあっても発生することがあります。

この時点では，Integration Studio でデプロイしたばかりの新しいアダプタを使用して，新しいプッシュ・アダプタ・ジョブを作成できます。

プッシュ・アダプタの TQL ベスト・プラクティス

1. TQL とビュー・ツリーでフォルダ構造を作成し，新しい TQL とビューをすべてそこで保持します。命名規則を使用します。
2. TQL が小さい場合を除き，最も類似した TQL を初めにコピーします。
3. 変更は一度に 1 つだけ行います。変更するごとに，保存，テスト，プレビューを行います。結果が要件を満たすまで繰り返します。

マッピングの作成

未処理の TQL 結果データは UCMDB クラス・モデル・スキーマの形式です。コンシューマはおそらく異なるデータ・モデルを使用します。プッシュ・アダプタは，データをよりコンシューマに適した形式に変換するマッピング・メカニズムを提供します。マッピングは，直接的なネーミング・タイプの変換から，親/子の集計や参照関数まで，直接的な変換と複合的な変換の両方を実行します。

マッピング仕様については，「ファイル参照のマッピング」(259ページ)の項を参照してください。マッピング・ファイルを作成するには，リファレンスを使用します。

注: アダプタのプロパティ・ファイルはマッピング・ファイルの名前を参照します。アダプタ構成ファイルに，アダプタはアダプタの名前を使用してフォルダ構造を実装します。パッケージ・マネージャにより要求される一意性を維持するため，アダプタを実装するときにこのフォルダの名前を変更します。

マッピング・ファイルの作成

1. 標準設定のマッピング・ファイルから始めます。
2. アダプタをデプロイして 1 回実行します。
3. 結果を観察します。
4. 何を変更する必要があるか特定し，書き留めます。
5. 前の手順で特定した変更を行います。次のリストは変更の順序のガイドとして使用できます。

a. 最上部の変換が不要なセクションから始めます。各変更の後で必ずアダプタを実行します。
b. TQL 結果のソース CI セクションを UCMDB 名に変更します。
c. 初めにキーをマッピングします。
d. 続いて，すべてのダイレクト・マッピングを追加します。
e. 複合マッピングを追加します。
f. リンク・マッピングを追加します。

マッピングしたデータが使用に適するようになるまで，手順 2 ~ 5 を繰り返します。新しいプッシュ・アダプタを作成するのに適したテンプレートとして汎用アダプタ・パッケージを選択します。

マッピング・ファイルはプッシュ・アダプタのすべてのタイプに対して同じやり方で機能します。汎用 XML プッシュ・アダプタはマッピングされた結果をファイルに書き込みます。汎用 Web Service プッシュ・アダプタは XML 結果をデータ・レシーバに送信します。詳細については，「汎用 Web Service プッシュ・アダプタ」(240ページ)を参照してください。

マッピング・ファイルの準備

注: mappings.xml ファイルを作成せずに，マッピングなしで，すべての CI と関係を CMDB 内での状態のまま取得できます。これは，すべての CI と関係をそのすべての属性とともに返します。

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，「ファイルのマッピングのスキーマ」(261ページ)を参照してください。

本項の内容

1. mappings.xml ファイルを作成する

   マッピング・ファイルの構造は，次のように作成されます（既存のファイルをテンプレートとして使用）。

```
<?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

CMDB CI タイプをマップする方法は 2 つあります。

- CI タイプを，そのタイプの CI および継承されたすべてのタイプが同じ方法でマップされるようにマップします。

```
<source_ci_type_tree name="node" mode="update_else_insert">
  <apioutputseq>1</apioutputseq>
  <target_ci_type  name="host">
    <targetprimarykey>
      <pkey>name</pkey>
    </targetprimarykey
    <target_attribute name=" name" datatype="STRING">
      <map type="direct" source_attribute="name" >
    </target_attribute>
    <!-- 追加のターゲット属性 --->
  </target_ci_type>
</source_ci_type_tree>
```

- CI タイプを，そのタイプの CI だけが処理されるようにマップします。継承された CI タイプは，それらのタイプも（2 つのうちいずれかの方法で）マップされない限り，処理されません。

```
<source_ci_type name="node" mode="update_else_insert">
  <apioutputseq>1</apioutputseq>
  <target_ci_type  name="host">
    <targetprimarykey>
      <pkey>name</pkey>
    </targetprimarykey
    <target_attribute name=" name" datatype="STRING">
      <map type="direct" source_attribute="name" >
    </target_attribute>
    <!-- 追加のターゲット属性 --->
```

```
  </target_ci_type>
</source_ci_type>
```

間接的にマップされた CI タイプ（その祖先の 1 つが **source_ci_type_tree** を使用してマップされている）は，親のマップを，自身の **source_ci_type_tree** または **source_ci_type** に表示することによって上書きできます。

できるだけ **source_ci_type_tree** を使用することが推奨されています。そうしない場合，マッピング・ファイルに表示されない CI タイプの結果 CI は，Jython スクリプトに移行されません。

3. リンクのマップ

リンクをマップする方法は 2 つあります。

- リンクを，そのタイプのリンクおよび継承されたすべてのリンクが同じ方法でマップされるようにマップします。

```
<source_link_type_tree name="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" >
    <target_attribute name="name" datatype="STRING">
      <map type="direct" source_attribute="name" >
    </target_attribute>
</source_link_type_tree>
```

- リンクを，そのタイプのリンクだけが処理されるようにマップします。継承されたタイプのリンクは，それらのタイプも（2 つのうちいずれかの方法で）マップされない限り，処理されません。

```
<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" >
    <target_attribute name="name" datatype="STRING">
      <map type="direct" source_attribute="name" >
  </target_attribute>
</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，「Jython アダプタの開発」(37ページ)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** または **DataPushResults** インスタンスを返す **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('Failed to insert to remote UCMDB using TopologyUpdateService.See log of the remote UCMDB')
```

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。

```
# get add/update/delete result objects (in XML format) from the Framework
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・オブジェクトは次のように取得できます。

```
oracleClient = Framework.createClient()
```

このクライアント・オブジェクトでは，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は（詳細については，「アダプタ・パッケージの作成」(228ページ)の **discoveryPatterns\push_adapter.xml** ファイルの編集方法についての手順を参照してください），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')
```

本項の内容

マッピングの結果を使った作業

汎用プッシュ・アダプタは，ターゲット・システムでのデータの追加，更新，削除を記述した XML 文字列を作成します。Jython スクリプトはこの XML を解析し，ターゲットに対する追加，更新，または削除の操作を実行する必要があります。

Jython スクリプトが受け取る追加操作の XML では，オブジェクトとリンクの mamId 属性は常に，タイプ，属性，またはその他の情報がリモート・システムのスキーマに変更される前の元のオブジェクトまたはリンクの UCMDB 識別子になります。

更新または削除操作の XML では，各オブジェクトまたは各リンクの mamId 属性に，前回の同期で Jython スクリプトから返されたものと同じ ExternalId の文字列表現が含まれます。

XML では，CI の id 属性は，外部 id として cmdbId を持つか，または CI がスクリプトに送信されたときに CI に ExternalId がある場合はその CI の ExternalId を持ちます。リンクの end1Id フィールドと end2Id フィールドは，各リンクのエンドに cmdbId を外部 ID として持つか，または，スクリプトに送信されたときにリンクのエンドの CI に ExternalId がある場合は，そのリンクのエンドの ExternalId を持ちます。

Jython スクリプトで CI を処理する際，スクリプトの戻り値は，CI の CMDB id と，指定された id（スクリプト内の各 CI に指定される id）とのマッピングになります。CI が最初にプッシュされる際，その CI の XML にある id は CMDB id です。CI のプッシュが初回でない場合，CI の id は，最初にプッシュされたときにスクリプトでその CI に指定されたのと同じ id です。

id は，次のように CI XML スクリプトから取得されます。

1. XML の CI 要素から，id 属性の id を取得します。たとえば， id = objectElement.getAttributeValue ('id')。
2. XML から id を取得後，属性（文字列）から id を復元します。たとえば， objectId = CmdbObjectID.Factory.restoreObjectID(id)。
3. 前の手順で受け取った objectId が CMDB id であることを確認します。これは，objectId に，スクリプトによって指定された新しい id があるかどうかを確認することで行えます。ある場合，返される id は CMDB id ではありません。例 :
   newId = objectId.getPropertyValue(<スクリプトによって指定された id 属性の名前>)。

   newId が null の場合，XML に返される id は CMDB id です。
4. id が CMDB id の場合（つまり，newId が null の場合）は，次を実行します（id が CMDB id ではない場合は手順 5 に進んでください）。
   a. 新しい id を持つ CI プロパティを作成します。例 : propArray =[TypesFactory.createProperty ('<スクリプトで指定された id 属性の名前>', '<新しい id>')]。
   b. その CI の externalId を作成します。たとえば，
   cmdbId = extI.getPropertyValue('internal_id')
   className = extI.getType()
   externalId = ExternalIdFactory.createExternalCiId(className, propArray)
   c. CMDB id を，新たに作成された externalId にマップします（そして，次の手順でそのマッピングをアダプタに返します）。たとえば， objectMappings.put(cmdbId, externalId)
   d. すべての CI とリンクがマップされると次のようになります。
   updateResult = DataPushResultsFactory.createDataPushResults(objectMappings, linkMappings);
   return updateResult
5. id が新しい id の場合（つまり newId が null でない場合），externalId は newId です。

また，各 CI とリンクのプッシュ・ステータスについて次のようにレポートできます。

1. updateStatus = ReplicationActionDataFactory.createUpdateStatus();

   ここで updateStatus は，CI とリンクのステータスを持った UpdateStatus クラスのインスタンスです。
2. reportCIStatus または reportRelationStatus メソッドを呼び出して，ステータスを updateStatus に追加します。

   たとえば，

   status = ReplicationActionDataFactory.createStatus(Severity.FAILURE, 'Failed', ERROR_CODE_CI, errorParams,Action.ADD);

updateStatus.reportCIStatus(externalId, status);

ここで ERROR_CODE_CI は，アダプタ **properties.errors** ファイルに表示されたエラー・メッセージ数であり（**properties.errors** ファイルの詳細については，「エラー記述の表記規則」(64ページ)を参照），errorParams にはメッセージに渡されたパラメータが含まれています。詳細については，**ReplicationActionDataFactory** javadoc を参照してください。

3. ステータスを持ったプッシュ結果は次のように作成します。

updateResult = DataPushResultsFactory.createDataPushResults(objectMappings, linkMappings, updateStatus);

return updateResult

XML 結果の例

<root>

<data>

<objects>

<Object mode="update_else_insert" name="UCMDB_UNIX" operation="add" mamId="0c82f591bc3a584121b0b85efd90b174" id="HiddenRmiDataSource%0Aunix%0A1%0Ainternal_id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A">

<field name="NAME" key="false" datatype="char" length="255">UNIX5</field>

<field name="DATA_NOTE" key="false" datatype="char" length="255"></field>

</Object>

</objects>

<links>

<link targetRelationshipClass="TALK" targetParent="unix" targetChild="unix" operation="add" mode="update_else_insert" mamId="265e985c6ec51a8543f461b30fa58f81" id="end1id%5BHiddenRmiDataSource%0Aunix%0A1%0Ainternal_id%3DSTRING%3D41372a1cbcaba27b214b84a2ec9eb535%0A%5D%0Aend2id%5BHiddenRmiDataSource%0Aunix%0A1%0Ainternal_id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A%5D%0AHiddenRmiDataSource%0Atalk%0A1%0Ainternal_id%3DSTRING%3D265e985c6ec51a8543f461b30fa58f81%0A">

<field name="DiscoveryID1">41372a1cbcaba27b214b84a2ec9eb535</field>

<field name="DiscoveryID2">0c82f591bc3a584121b0b85efd90b174</field>

<field name="end1Id">HiddenRmiDataSource%0Aunix%0A1%0Ainternal_id%3DSTRING%3D41372a1cbcaba27b214b84a2ec9eb535%0A</field>

<field name="end2Id">HiddenRmiDataSource%0Aunix%0A1%0Ainternal_id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A</field>

<field name="NAME" key="false" datatype="char" length="255">TALK4</field>

```
    <field name="DATA_NOTE" key="false" datatype="char" length="255"></field>
    </link>
  </links>
 </data>
</root>
```

注: datatype="BYTE" の場合，返される結果の値は**文字列**で，new String([the byte array attribute]) として生成されます。byte[] object は，<受け取った文字列>.getBytes() で再構築できます。サーバとプローブの間では，標準設定のロケールに差異がある場合，再構築は，サーバの標準設定のロケールに応じて行われます。

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。

差分同期のサポート

プッシュ・アダプタで差分同期をサポートするには，**DiscoveryMain** 関数は，Jython スクリプトが XML から受け取る ID と Jython スクリプトがリモート・マシンで作成する ID の間のマッピングを含む，**DataPushResults** インタフェースを実装するオブジェクトを返す必要があります。後者の ID はタイプ **ExternalId** です。

CMDB にある CI の ID をパラメータとして受け取る **ExternalIdUtil.restoreExternal** コマンドは，外部 ID を CMDB にある CI の ID から復元します。このコマンドはたとえば，差分同期の実行中，いずれかのエンドがバルクにない（すでに同期済み）ときにリンクを受信する場合に使用できます。

プッシュ・アダプタがベースとする Jython スクリプトの **DiscoveryMain** メソッドが，空の **ObjectStateHolderVector** インスタンスを返す場合，アダプタでは差分同期はサポートされません。つまり，差分同期ジョブを実行しても，実際には完全同期が実行されます。同期するたびにすべてのデータが CMDB に追加されるため，リモート・システムでデータを更新または削除することはできません。

重要 : バージョン 9.00 または 9.01 で作成された既存のアダプタで差分同期を実装する場合，バージョン 9.02 以降の push-adapter.zip ファイルを使用してアダプタ・パッケージを再作成する

必要があります。詳細については，「アダプタ・パッケージの作成」(228ページ)を参照してください。

このタスクにより，プッシュ・アダプタで差分同期を実行できるようになります。

Jython スクリプトは，オブジェクト ID マッピング用（キーおよび値は ExternalCiId タイプ・オブジェクト）およびリンク ID 用（キーおよび値は ExternalRelationId タイプ・オブジェクト）の 2 つの Java マップを含む **DataPushResults** オブジェクトを返します。

- 次の **from** ステートメントを Jython スクリプトに追加します。

  ```
  from com.hp.ucmdb.federationspi.data.query.types import ExternalIdFactory
  from com.hp.ucmdb.adapters.push import DataPushResults
  from com.hp.ucmdb.adapters.push import DataPushResultsFactory
  from com.mercury.topaz.cmdb.server.fcmdb.spi.data.query.types import ExternalIdUtil
  ```

- **DataPushResultsFactory** ファクトリ・クラスを使用して，**DiscoveryMain** 関数から **DataPushResults** オブジェクトを取得します。

  ```
  # UpdateResult オブジェクトを作成
  updateResult = DataPushResultsFactory.createDataPushResults(objectMappings, linkMappings);
  ```

- 次のコマンドを使用して，**DataPushResults** オブジェクトの Java マップを作成します。

  ```
  # ID のマッピング項目を格納するマップを用意
  objectMappings = HashMap()
  linkMappings = HashMap()
  ```

- **ExternalIdFactory** クラスを使用して，次の ExternalId ID を作成します。
  - CMDB で発生したオブジェクトまたはリンクの ExternalId（たとえば，追加操作のすべての CI は CMDB から発生します）

    ```
    externaCIId = ExternalIdFactory.createExternalCmdbCiId(ciType, ciIDAsString)
    externalRelationId = ExternalIdFactory.createExternalCmdbRelationId(linkType, end1ExternalCIId, end2ExternalCIId, linkIDAsString)
    ```

  - CMDB で発生していないオブジェクトまたはリンクの ExternalId（通常は，すべての更新および削除操作にこのようなオブジェクトが含まれます）

    ```
    myIDField = TypesFactory.createProperty("systemID", "1")
    myExternalId = ExternalIdFactory.createExternalCiId(type, myIDField)
    ```

    注: Jython スクリプトにより既存の情報が更新され，オブジェクト（またはリンク）の ID が変更された場合，前の外部 ID と新しい外部 ID 間のマッピングを返す必要があります。

- **ExternalIdFactory** クラスの **restoreCmdbCiIDString** または **restoreCmdbRelationIDString** メソッドを使用して，UCMDB で発生したオブジェクトまたはリンクの外部 ID から UCMDB ID 文字列を取得します。

- **ExternalIdUtil** クラスの**restoreExternalCiId** および **restoreExternalRelationId** メソッドを使用して，更新操作または削除操作の XML の mamId 属性値から **ExternalId** オブジェクトを復元します。

  注: ExternalId オブジェクトは，実際にはプロパティの配列です。つまり，**ExternalId** オブジェクトを使用して，リモート・システム上のデータの識別に必要な情報を保存できます。

汎用 XML プッシュ・アダプタ SQL クエリ

アダプタ・パッケージの **adapterCode > PushAdapter > sqlTablesCreation** にある **sql_queries** ファイルには，アダプタのテスト用に，Oracle の新しいスキーマでテーブルを作成するために必要なクエリが含まれています。テーブルは adapterCode\<adapter ID>\mappings\mappings.xml ファイルに対応しています。

注: sql_queries ファイルは，アダプタには必要ありません。これは単なるサンプルです。

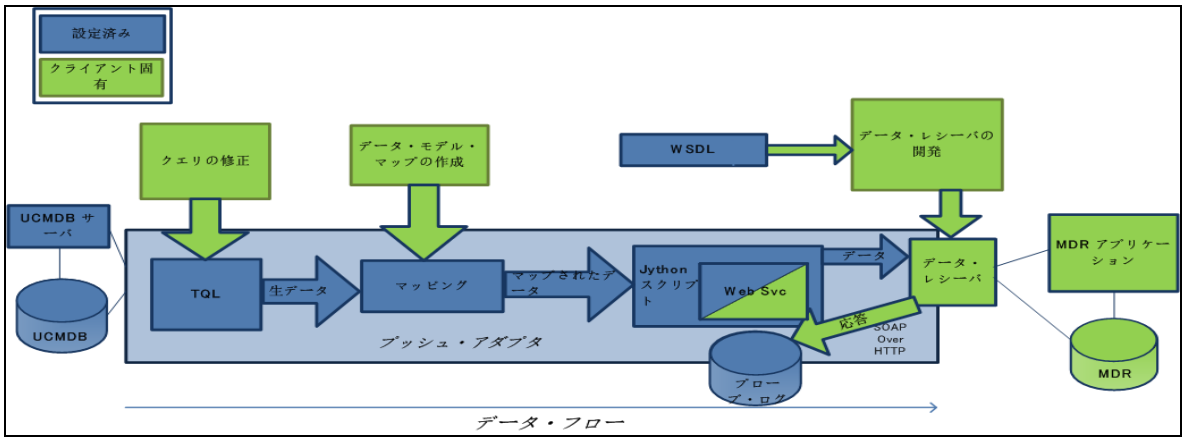
汎用 Web Service プッシュ・アダプタ

汎用 Web Service プッシュ・アダプタは，Web Service データ・レシーバへのクエリ・データを含んだ SOAP メッセージの UCMDB 起動プッシュを提供します。マッピングされた結果は，標準 SOAP メッセージで HTTP POST プロトコルを介してデータ・レシーバに送信されます。データ・レシーバはプッシュ・アダプタによって生成された SOAP メッセージを理解できる必要があります。適切なデータ・レシーバの開発を促進するため，WSDL がこのプッシュ・アダプタに付属しています。

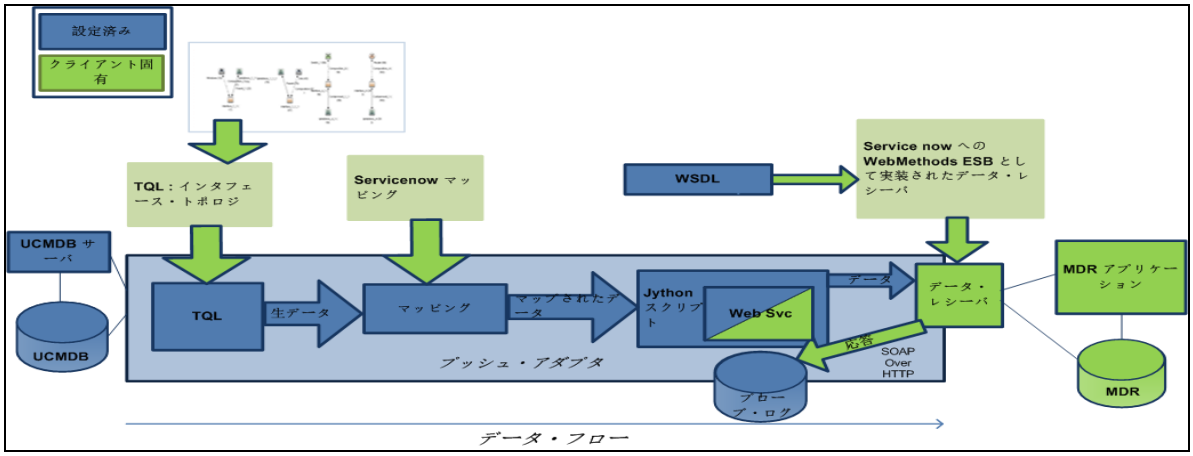
SOAP メッセージ応答 XML のカスタム処理は Jython スクリプトで可能です。

着信マッピング・データの形式を知るため，データ・レシーバの開発者はマッピング・ファイルの開発者と連絡を取る必要があります。**.xsd** は現在このバージョンの Web Service プッシュ・アダプタには付属していないため，データは着信データを反映した方法で処理される必要があります。着信データは元の TQL と適用されたマッピングが組み合わされています。

データをクライアントにプッシュする Web Service プッシュ・アダプタの機能を以下に示します。緑色の項目は，アダプタを特定のプッシュ・ターゲットに実装するためにクライアントによってカスタマイズまたは供給されます。青色の項目は定義済みのコンポーネントです。

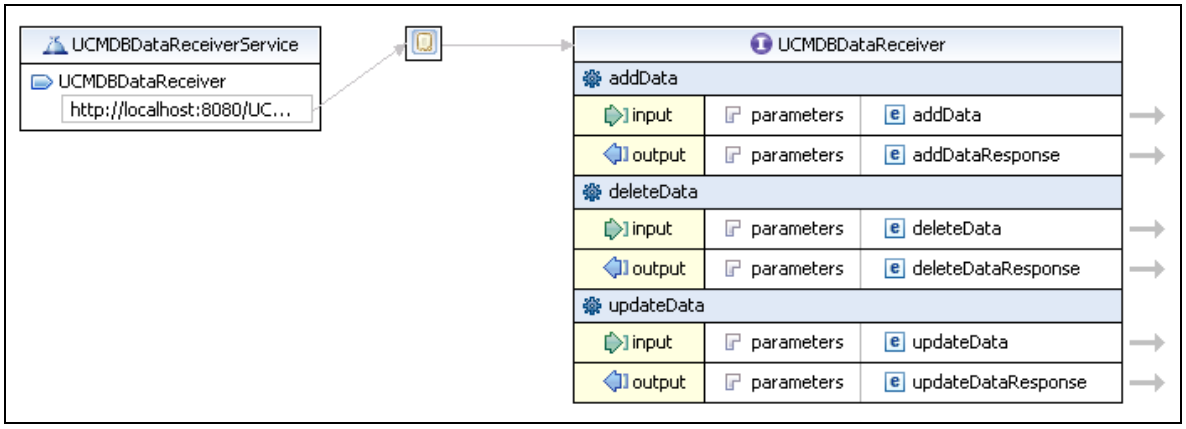


以下に，Enterprise Service Bus (ESB) を使用して汎用 Web Service プッシュ・アダプタを MDR 固有のプッシュ・アダプタに実装する例を示します。



WSDL

Web Service を介して UCMDB プッシュ・アダプタと通信できるデータ・レシーバを作成するため，WSDL がクライアント開発者に供給されます。**UCMDBDataReceiver.wsdl** は，UCMDB からのデータをデータ・レシーバに連絡するために使用される SOAP メッセージを記述します。WSDL の設計図を以下に示します。



データ・レシーバ（実際にはサーバまたは SOAP 用語では「service endpoint」）には**addData**，**deleteData**，**updateData** の 3 つのメソッドを実装する必要があります。これらは UCMDB がプッシュするデータ・セットに対応します。HTTP ヘッダーには送信されるデータのタイプを示す正しい **SoapAction** キーワードが含まれています。データ・レシーバはビジネス・ロジックの実装やデータの処理を担当します。

標準設定の WSDL URL は：

- http://localhost:8080/UCMDBDataReceiver/services/UCMDBDataReceiver?wsdl

データ・レシーバによって実装されると，URL は以下に類似したものになります。

- http://testWSPAserver:4444/MyCo.IT.SvcMgt.ws.us:provider/UCMDBDataReceiver?wsdl

Web Service の URL は末尾から “?wsdl” を削除した WSDL URL と同じです。

WSDL のソースを以下に示します。

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="http://ucmdb.hp.com"
xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://ucmdb.hp.com"
xmlns:intf="http://ucmdb.hp.com" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!--Apache Axis バージョンによって作成された WSDL:1.4 Built on Apr 22, 2006 (06:55:48 PDT)-->

    <wsdl:types>

        <schema elementFormDefault="qualified" targetNamespace="http://ucmdb.hp.com"
        xmlns="http://www.w3.org/2001/XMLSchema">

            <element name="addData">

                <complexType>

                    <sequence>

                    <element name="xmlAdded" type="xsd:string"/>
```

```
                    </sequence>
                </complexType>
            </element>
            <element name="addDataResponse">
                <complexType/>
            </element>
            <element name="deleteData">
                <complexType>
                    <sequence>
                    <element name="xmlDeleted" type="xsd:string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="deleteDataResponse">
                <complexType/>
            </element>
            <element name="updateData">
                <complexType>
                    <sequence>
                    <element name="xmlUpdate" type="xsd:string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="updateDataResponse">
                <complexType/>
            </element>
        </schema>
    </wsdl:types>

    <wsdl:message name="addDataRequest">
        <wsdl:part element="impl:addData" name="parameters">
```

```
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="deleteDataResponse">
        <wsdl:part element="impl:deleteDataResponse" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="updateDataResponse">
        <wsdl:part element="impl:updateDataResponse" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="deleteDataRequest">
        <wsdl:part element="impl:deleteData" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="addDataResponse">
        <wsdl:part element="impl:addDataResponse" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="updateDataRequest">
        <wsdl:part element="impl:updateData" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:portType name="UCMDBDataReceiver">
        <wsdl:operation name="addData">
            <wsdlsoap:operation soapAction="addDataRequest"/>
            <wsdl:input message="impl:addDataRequest" name="addDataRequest">
            </wsdl:input>
            <wsdl:output message="impl:addDataResponse" name="addDataResponse">
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="deleteData">
```

```
            <wsdlsoap:operation soapAction="deleteDataRequest"/>
            <wsdl:input message="impl:deleteDataRequest" name="deleteDataRequest">
            </wsdl:input>
            <wsdl:output message="impl:deleteDataResponse" name="deleteDataResponse">
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="updateData">
            <wsdlsoap:operation soapAction="updateDataRequest"/>
            <wsdl:input message="impl:updateDataRequest" name="updateDataRequest">
            </wsdl:input>
            <wsdl:output message="impl:updateDataResponse" name="updateDataResponse">
            </wsdl:output>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="UCMDBDataReceiverSoapBinding" type="impl:UCMDBDataReceiver">
        <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="addData">
            <wsdl:input name="addDataRequest">
                <wsdlsoap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="addDataResponse">
                <wsdlsoap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="deleteData">
            <wsdl:input name="deleteDataRequest">
                <wsdlsoap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="deleteDataResponse">
                <wsdlsoap:body use="literal" />
            </wsdl:output>
```

```
            </wsdl:operation>
            <wsdl:operation name="updateData">
                <wsdl:input name="updateDataRequest">
                    <wsdlsoap:body use="literal" />
                </wsdl:input>
                <wsdl:output name="updateDataResponse">
                    <wsdlsoap:body use="literal" />
                </wsdl:output>
            </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="UCMDBDataReceiverService">
            <wsdl:port binding="impl:UCMDBDataReceiverSoapBinding" name="UCMDBDataReceiver">
                <wsdlsoap:address location="http://localhost:8080/UCMDBDataReceiver/services/
                UCMDBDataReceiver"/>
            </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

応答処理

データ・レシーバは文字列を addDataResponse，deleteDataResponse，または updateDataResponse 構造で返す必要があります。アダプタは未処理の応答データをプローブの probeMgr-adaptersDebug.log に渡します。レシーバは任意の文字列データを返すことができ，応答は SOAP 対応 XML でラップされます。Jython スクリプトでは，SOAPMessage および関連する Java クラスを使用して応答メッセージを解析できます。以下に，データ・レシーバからの応答メッセージの例を示します。

```
<2012-03-16 15:47:38,080> [INFO ] [Thread-110] - XMLtoWebService.py:addData received response:
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<intf:addDataResponse xmlns:intf="http://ucmdb.hp.com">
    <xml>&lt;result&gt;&lt;status&gt;error&lt;/status&gt;
    &lt;message&gt;Error publishing config item changes&lt;/message&gt;
    &lt;/result&gt;</xml>
</intf:addDataResponse>
</soapenv:Body>
```

表示されたメッセージはエラー・メッセージ **<Error publishing config item changes>** ですが，内容はデータ・レシーバが応答するために設計されたものなら何でも可能です。応答がエラー・メッセージである理由は，それが目的であり，設計者がエラー・メッセージと言っているためであり，プッシュ・アダプタが成功か失敗の何らかの表示に対する応答を予期しているためです。内容は正常に追加されたすべての CI の調整 ID や，特定の CI のエラー・メッセージにすることができます。GWSPA のカスタマイズには，応答メッセージの解析や特定の CI の再送などのアクションまたは他のログ作成の実行などを含めることができます。

WSDL のテスト

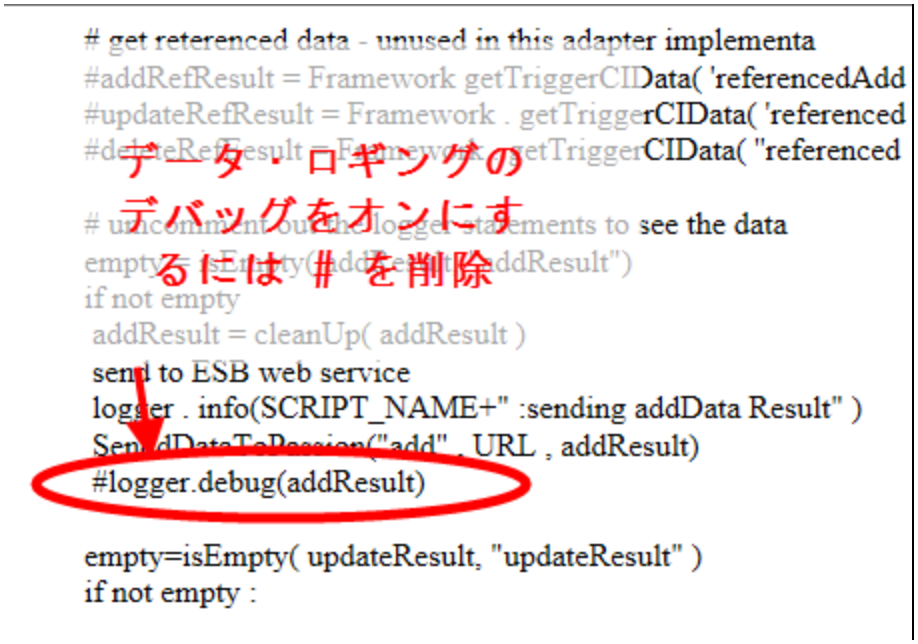
SOAPUI Eclipse プラグインは開発中の Web Service レイヤのテストに使用されます。SOAPUI を使用すると Web Service のカスタマイズを支援できます。SOAPUI は SOAP メッセージの作成，送信，受信をテストする統合開発環境（IDE）を提供します。SOAPUI の観点では，ページ 242-246 の WSDL は次のサンプル・メッセージを生成しました。

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ucm="http://ucmdb.hp.com">
    <soapenv:Header/>
    <soapenv:Body>
        <ucm:addData>
            <ucm:xmlAdded>?</ucm:xmlAdded>
        </ucm:addData>
    </soapenv:Body>
</soapenv:Envelope>
```

上記の **xmlAdded** 要素の「**?**」はデータの場所であり，Web Service プッシュ・アダプタ統合によって供給されます。

結果の監視

プッシュ・アダプタが非デバッグ・モードで正常に稼働していると，データは終結果が記述されるまでファイルに記述されません（TQL 中間結果とマッピングされたデータ結果は通常はログファイルに表示されません）。ただし，ここに示す DiscoveryMain セクションで **logger.debug** ステートメントをコメント解除（“#” 文字を削除）すると，結果をプローブのデバッグ・ファイルに書き込むことができます。



logger ステートメントが他の先立つ行および次の行と同じカラムで開始することを確認します。Jython はインデントを区別し，すべての行のインデントが正しくない場合はスクリプトが失敗します。

以下のプローブのデバッグ・ログ・ファイル **probeMgr-adaptersDebug.log** は出力の内容を示します。

```
<2011-12-07 14:02:23,019> [INFO ] [Thread-273] - XMLtoWebService.py started

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - ESB Push parameters:

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - Wshost=harpy.trtc.com

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - WShostport=5555

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] -
WSuri=ws/DtITServiceManagement.esla.v1.ws.provider:UMDBDataReceiver

<2011-12-07 14:02:23,019> [INFO ] [Thread-273] - URL is
http://harpy.trtc.com:5555/ws/DtITServiceManagement.esla.v1.ws.
provider:UMDBDataReceiver

<2011-12-07 14:02:23,035> [DEBUG] [Thread-273] - Connected to
http://harpy.trtc.com:5555/ws/DtITServiceManagement.esla.v1.ws.
provider:UMDBDataReceiver

<2011-12-07 14:02:23,035> [ERROR] [Thread-273] - sending results

<2011-12-07 14:02:23,035> [DEBUG] [Thread-273] - <?xml version="1.0" encoding="UTF-8"?>

<root>

      <data>
```

```
<objects>
    <Object mode="" name="u_imp_ip_switch" operation="add"
    mamId="9e8c2f6bdfe4b7d0864c79e70833902c">
        <field name="Correlation ID" key="true" datatype="char"
        length="">9e8c2f6bdfe4b7d0864c79e70833902c</field>
        <field name="name" key="false" datatype="char" length="">nma_09sw</field>
        <field name="location" key="false" datatype="char" length="" />
        <field name="u_chassis_vendor_type" key="false" datatype="char"
        length="">ciscoCat2960-24TT</field>
        <field name="serial_number" key="false" datatype="char" length="" />
        <field name="ram" key="false" datatype="char" length="" />
        <field name="os_version" key="false" datatype="char" length="" />
    </Object>
```

Jython スクリプトの変更

XMLtoWebService.py

Web Service プッシュ・アダプタによって使用される Jython スクリプトは XML プッシュ・アダプタに非常によく似ています。このスクリプトは **UCMDBDataReceiver.jar** を使用し，アダプタに含まれます。このスクリプトは **SendDataToReceiver()** メソッドを実装します。**SendDataToReceiver()** は次の 3 つのパラメータを使用します。

1. アクション（追加，更新，または削除）
2. データ・レシーバの URL
3. データ

たとえば，add block は次のようになります :**SendDataToReceiver(“add” , URL, addResult)**

すべての Web Service と SOAP レイヤがラップされます。URL は UCMDB データ・レシーバのサービス・エンドポイント・アドレスです。これは「?wsdl」サフィックスにより wsdl を取得するために使う URL と同じです。

Jython スクリプトのソースを以下に示します。Web Service 統合ラッパー行は 緑色で示されています。

```
#####################################
# script:XMLtoWebService.py
#####################################
```

```
# この jython スクリプトは統合アダプタから TQL データ結果（追加，更新，削除）を受け入れます。
# またそれを Web Service に送信します。Web Service は UCMDBDataReceiver と呼ばれます。
# この名前の Web Service クライアントはパラメータによって与えられた URL でアドレス可能でなければなりません。
# SendDataToReceiver.jar は SendDataToReceiver 関数とともに service locator を公開します。
# service locator の例は testconnection セクションにあります。
# 正規表現
import re
# ロギング
import logger
# Web Service インタフェース
from com.hp.ucmdb import SendDataToReceiver
from com.hp.ucmdb.SendDataToReceiver import locateService
from com.hp.ucmdb.SendDataToReceiver import SendData
##############################################
########    VARIABLES         ###########
##############################################
SCRIPT_NAME = "XMLtoWebService.py"
logger.info(SCRIPT_NAME+" started")
def cleanUp(str):

    # replace mode=""
    str = re.sub("mode=\"\w+\"\s+", "", str)

    # mamId を id で置き換え
    str = re.sub("\smamId=\"", " id=\"", str)

    # 空属性を置換
    str = re.sub("[\n|\s|\r]*<field name=\"\w+\" datatype=\"\w+\" />", "", str)

    # targetRelationshipClass を名前で置き換え
```

```
        str = re.sub("\stargetRelationshipClass=\"", " name=\"", str)

        # Object を名前付きオブジェクトで置き換え
        str = re.sub("<Object mode=\"", "<object mode=\"", str)
        str = re.sub("<Object operation=\"", "<object operation=\"", str)
        str = re.sub("<Object name=\"", "<object name=\"", str)
        str = re.sub("</Object>", "</object>", str)

        # フィールドを属性に置き換え
        str = re.sub("<field name=\"", "<attribute name=\"", str)
        str = re.sub("</field>", "</attribute>", str)

        #logger.debug("String = %s" % str)
        #logger.debug("cleaned up")

        return str
def isEmpty(xml, type = ""):
        objectsEmpty = 0
        linksEmpty = 0

        m = re.findall("<objects />", xml)
        if m:
                #logger.warn("\t[%s] No objects found" % type)
                objectsEmpty = 1

        m = re.findall("<links />", xml)
        if m:
                #logger.warn("\t[%s] No links found" % type)
                linksEmpty = 1

        if objectsEmpty and linksEmpty:
```

```
            return 1
        return 0
##############################################
########     MAIN             ###########
##############################################
def DiscoveryMain(Framework):
        #これを Web Service エクスポートにフィックス
        errMsg = "UCMDBDataReceiver Service not found."
        testConnection = Framework.getTriggerCIData("testConnection")
        # Web Service Push 変数を取得
        WShostName = Framework.getTriggerCIData("Host Name")
        WShostport = Framework.getTriggerCIData("Protocol Port")
        WSuri = Framework.getTriggerCIData("URI")

        logger.info(SCRIPT_NAME+":ESB Push parameters:")
        logger.info("Host Name="+WShostName)
        logger.info("Protocol Port="+WShostport)
        logger.info("URI="+WSuri)
        URL = "http://"+WShostName+":"+WShostport+"/"+WSuri
        logger.info("URL="+URL)
        if testConnection == 'true':
            # Service を見つける
            test_receiver = SendDataToReceiver()
            locator = test_receiver.locateService(URL)
            #locator = locateService(URL)
            if(locator):
                logger.info(SCRIPT_NAME+":Test connection was successful")
                return
            else:
                raise Exception, errMsg
                return
```

```
# 単なるテスト接続でない場合は，ここで同じことをする -
receiver = SendDataToReceiver()
locator = receiver.locateService(URL)
if(locator):
        logger.info(SCRIPT_NAME+":Connected to "+URL)
else:
        logger.error(SCRIPT_NAME+":no locator")
        raise Exception, errMsg
        return

# フレームワークで結果オブジェクトの追加/更新/削除を取得する
addResult = Framework.getTriggerCIData('addResult')
updateResult = Framework.getTriggerCIData('updateResult')
deleteResult = Framework.getTriggerCIData('deleteResult')
logger.debug(deleteResult)

# 参照データを取得 - このアダプタ実装で未使用
#addRefResult = Framework.getTriggerCIData('referencedAddResult')
#updateRefResult = Framework.getTriggerCIData('referencedUpdateResult')
#deleteRefResult = Framework.getTriggerCIData('referencedDeleteResult')
# データを見るには logger ステートメントをコメント・アウト解除
empty = isEmpty(addResult, "addResult")
if not empty:
        addResult = cleanUp(addResult)
        # ESB web service に送信
        logger.info(SCRIPT_NAME+":sending addData Result")
        rcvr = SendDataToReceiver()
        resp = rcvr.SendData("add", URL, addResult)
        logger.info(SCRIPT_NAME+":addData received response:"+resp)
        #logger.debug(addResult)
empty = isEmpty(updateResult, "updateResult")
```

```
if not empty:
        updateResult = cleanUp(updateResult)
        # ESB web service に送信
        #logger.debug(updateResult)
        logger.info(SCRIPT_NAME+":sending updateData Result")
        rcvr = SendDataToReceiver()
        resp = rcvr.SendData("update", URL, updateResult)
        logger.info(SCRIPT_NAME+":received response:"+resp)


empty = isEmpty(deleteResult, "deleteResult")
if not empty:
        deleteResult = cleanUp(deleteResult)
        # ESB web service に送信
        #logger.debug(deleteResult)
        logger.info(SCRIPT_NAME+":sending deleteData Result")
        rcvr = SendDataToReceiver()
        resp = rcvr.SendData("delete", URL, deleteResult)
        logger.info(SCRIPT_NAME+":received response:"+resp)
logger.info(SCRIPT_NAME+" ended")
```

応答メッセージ処理のカスタマイズ

データ・レシーバは応答や希望するステータスを含んだ文字列を返す必要があります。Web Service プッシュ・アダプタは標準設定で応答をプローブの情報レベル・ログに応答を渡します。応答メッセージは返された応答文字列を内部に持った SOAP 形式の XML です。レシーバによってグループ化や個別のエラーまたは成功メッセージなどの任意のデータを返すことができます。その他の処理が必要な場合は，応答はアダプタの Jython スクリプトで処理できます。Java プログラミングは不要です。

以下を使用して送信された返信応答メッセージの例 :

```
// stub example for building your own UCMDBDataData Receiver
public class UCMDBDataReceiver {


public String addData (String xmlAdd){
System.out.println(xmlAdd); // do something with the data
```

```
// send back a response message based on what you did
String tr = new String("a test response from addData!");
return tr;
}
```

を次に示します。

```
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <addDataResponse xmlns="http://ucmdb.hp.com">
        <addDataReturn>a test response from addData!</addDataReturn>
    </addDataResponse>
</soapenv:Body>
```

データ・レシーバの変更

Java クライアントは **UCMDBDataReceiver.jar** に含まれたクラスを実装して，Web Service を Jython 同じやり方で呼び出すことができます。さらに，ラップ解除メソッドも呼び出すことができます。Javadoc は **UCMDBDataReceiver.jar** クラス用に存在します。以下のソース・コードは，データを SOAP メッセージでラップして，それを HTTP を介してレシーバに送信する，これらの重要なメソッドの使い方を示します。

プロセスは **UCMDBDataReceiverServiceLocator** オブジェクトを作成し，**UCMDBDataReceiverEndPointAddress** をデータ・レシーバの URL に割り当てます。

データを送信するには，ロケータの **getUCMDBDataReceiver** メソッドを呼び出して，**UCMDBDataReceiver** オブジェクトを作成します。**UCMDBDataReceiver** オブジェクトはメソッドを実装して，実際にデータの追加/変更/削除を送信します。要求の各タイプを処理する 3 つの同じコード・ブロックがあります。

SendDataToReceiver クラスのソース・コードを以下にリストします。強調表示したオブジェクトとメソッドはその使用が重要な要素です。

```
/**
* UCMDB Web Service プッシュ・アダプタの UCMDB データ・レシーバの SendData をテスト
*/
package com.hp.ucmdb;
import com.hp.ucmdb.SendDataToReceiver;
/**
* TestSendData は SOAP クラスが機能していることを確認するために使用できます。
```

```
* TestSendData は SendDataToReceiver クラスを作成して，その SendData メソッドを呼び出します。
* 応答文字列が返されます。
* テスト URL は一般的に "?wsdl" が末尾に付加され，サービスの WSDL を取得します。
*/
パブリック・クラス TestSendData {
    /**
    * @param args - テスト SOAP メッセージ
    *  データ・レシーバの元の引数 [0] テスト文字列 [1] サービス・エンドポイント URL。
    * 標準設定の URL はテスト・メッセージとして着信引数に送信されます。
    * 標準設定の URL は "http://localhost:8080/UCMDBDataReceiver/services/UCMDBDataReceiver"。
    * 任意のエラーが発生した場合は，TestClient が例外をスローしようとします。
    */
    public static void main(String[] args) {
        // 供給されている場合は，テスト・メッセージを使用します。そうでない場合は，標準設定のテスト文字列を供給します
        String teststring = new String("Test SOAP message from UCMDBDataReceiver TestSendData.");
        if(args.length > 0) {
            teststring = args[0];
        }
        // 供給されている場合は，テスト URL を使用し，そうでない場合は標準設定の URL を供給します
        String URL = new String("");
        if(args.length > 1) {
            URL = args[1];
        }
        // 応答を返す
        String response = new String("");
        // テストを実行する
        try {
            if(URL.equals("")) {
                UCMDBDataReceiverServiceLocator locator = new UCMDBDataReceiverServiceLocator
                ();
```

```
                UCMDBDataReceiver receiver = locator.getUCMDBDataReceiver();
                URL = locator.getUCMDBDataReceiverAddress();
                System.out.println("TestClient:tested URL="+locator.getUCMDBDataReceiverAddress
                ());
                System.out.println("TestClient:receiver="+receiver.toString());
            }
            SendDataToReceiver sdtr = new SendDataToReceiver();
            // これはテスト・プッシュを送信し，応答メッセージを取得する
            response = sdtr.SendData("add", URL, args[0]);
            System.out.println("Response received was:"+response);
        } catch (Exception e) {
            System.out.println("TestClient:Remote Error:");
            e.printStackTrace();
        }
    }
}
```

他のクラスのソース・コードは **UCMDBDataReceiver.jar** ファイルにも含まれています :

- TestClient.java
- UCMDBDataReceiver.java
- UCMDBDataReceiverProxy.java
- UCMDBDataReceiverService.java
- UCMDBDataReceiverServiceLocator.java
- UCMDBDataReceiverSoapBindingStub.java

ソースは Eclipse IDE で生成され，変更されました。UCMDB コードを変更する場合は注意が必要です。その多くが SOAP 仕様と UCMDB データ・レシーバに合うように自動生成されています。

Javadoc

十分にコメントが付けられた **javadoc** が汎用 Web Service プッシュ・アダプタに付属しています。**javadoc** は docs フォルダ **javadoc** に含まれています。**index.html** から開始します。概要ページには SDK のすべてのクラスとメソッド用のドキュメンテーションへのアクセスがあります。

すべてのクラス

- **SendDataToReceiver:** web service ラッパー用 API
- **TestClient:** サービス・エンドポイントへの接続を確認するテスト・クライアント

- **UCMDBDataReceiver:** web service ラッパー

その他は web service builder によって自動的に生成されます :

- UCMDBDataReceiverProxy
- UCMDBDataReceiverService
- UCMDBDataReceiverServiceLocator
- UCMDBDataReceiverSoapBindingStub

概要

SDK の基本的な使用方法は，ソース・コードのサンプルとともに，パッケージのドキュメンテーションで説明されています。この **javadoc** は UCMDB web service プッシュ・アダプタ用です。API は Jython または Java から呼び出すことができます。

SDK には 2 つのソース・サンプル **TestClient** と **SendDataToReceiver** があります。**TestClient** にはローカル・クライアントへの応答の非常に限定されたテストがあります。**SendDataToReceiver** は web service にデータを送信するために使用されるメイン・クラスです。

初めに，この SDK（主に同梱された WSDL）を使用して UCMDB データ・レシーバを実装し，この web service と通信します。さらに，この SDK を使用して UCMDB のプッシュ・アダプタを作成し，UCMDB TQL 結果データをデータ・レシーバにプッシュします。この API の基本的な使用方法は，Jython 実装と Java 実装の両方により以下に説明されています。

Implementing SendDataToReceiver()

SendDataToReceiver() すべての関数を単一のメソッドでラップします :

- Jython:SendDataToReceiver("add",yourURL,"Hello!")
- Java:SendDataToReceiver("add",yourURL,"Hello!");

または，**SendDataToReceiver** オブジェクトを作成し（たとえば，他の設定を操作するため），以下に示すように **SendData** メソッドを別個に呼び出します。

- Jython:

  ```
  rcvr = SendDataToReceiver()
  responseMsg = rcvr.SendData(“add”, yourURL, “Hello!”)
  ```

- Java:

  ```
  SendDataToReceiver rcvr = new SendDataToReceiver();
  String responseMsg = rcvr.SendData(“add”, yourURL, “Hello!”);
  ```

または，一度に 1 ステップで行う必要がある場合は，以下のように行うことができます。

1. 新しい **UCMDBDataReceiverServiceLocator()** オブジェクト x を作成し，以下に示すようにオブジェクトのエンドポイント・アドレスを後で設定します。
   - **Jython:**

     ```
     x = UCMDBDataReceiverServiceLocator()
     x.setUCMDBDataReceiverEndPointAddress(URL)
     ```

- **Java :**

  UCMDBDataReceiverServiceLocator x = new UCMDBDataReceiverServiceLocator();

  x. setUCMDBDataReceiverEndPointAddress(URL);

2. さらに，UCMDBDataReceiver を次のように作成します
   - **Jython:** y = x.getUCMDBDataReceiver()
   - **Java:** UCMDBDataReceiver y = x.getUCMDBDataReceiver();
3. さらに，次のような SOAP web service を介してデータを送信します。
   - **Jython:**
     - y.addData(yourData)
     - または y.updateData(yourData)
     - または y.deleteData(yourData)
   - **Java :**
     - y.addData(yourData);
     - または y.updateData(yourData);
     - または y.deleteData(yourData);
4. 接続性をテストする必要があります。それに成功したら，同じロケータ・オブジェクトを再利用して **UCMDBDataReceiver** を返してデータ転送に使用します。

クラスにはデストラクタが含まれていないので，メモリ管理を実行しないでください。

ファイル参照のマッピング

マッピングの使用

マッピングは，変換された XML 出力の各ターゲット属性に作成する必要があります。このマッピングはデータを取得する場所と方法を指定します。データが UCMDB 内の別の対応する属性にある場合は，ダイレクト・マッピングが使用されます。

データを複数の属性から，または UCMDB CI の子または親 CI の属性からプルするには，他の複合マッピングが必要な場合があります。以下のマッピング・スキーマに，可能なすべてのマッピングを示します。

マッピング・ファイルは，UCMDB のどの CI/関係タイプがターゲット・データ・ストアのどの CI/関係タイプにマッピングされるかを定義する XML ファイルです。以下に，形式について詳しく説明します。マッピング・ファイルは，どの CI タイプと関係タイプがプッシュされるかをコントロールするとともに，その属性がプッシュされるかをコントロールする。

マッピング・エントリが，ターゲット MDR にプッシュされる各属性に存在する。各マッピング・エントリは，未加工の UCMDB プッシュ・データ内に 1 つまたは複数の属性で構成されている場合がある。マッピング・エントリは，最終構成の非常に詳細なコントロールと，ターゲット MDR へプッシュされるデータの命名を可能にする。

ダイレクト・マッピング

マッピングはあるデータ・モデルを別のモデルに変換します（この場合は，UCMDB をプッシュ・ターゲット MDR に）。UCMDB 属性とターゲットが 1:1 の関係の場合は，変換は単純であり，名前とおそらくタイプが異なるだけです。

ほとんどの属性マッピングはダイレクトです。たとえば，サーバ名 “ServerX” は，UCMDB では **unix** タイプの CI，属性名 **primary_server_name**，長さが 50 の **smtring** タイプとして表すことができます。ターゲット MDR のデータ・モデルは **linux** の CI タイプと同じ論理エンティティ，**hostname** の属性名，**char[]** のタイプ，最大長さが 250 に指定できます。ダイレクト・マッピングは前述の変換タスクをすべて実現できます。

以下に，ダイレクト・マッピングの例を示します。

```
 <target_attribute name="dns_domain" datatype="char">
<map type="direct" source_attribute="domain_name" />
</target_attribute>
```

このダイレクト・マッピングは UCMDB 属性 **dns_domain** をターゲット・データ・モデルの **domain_name** 属性にマッピングします。

実際のデータ・タイプを使用する必要がない限り，実際のデータ・タイプに関係なく **char** データ・タイプを使用します。

複合マッピング

次のように複合マッピングを増やすと，追加の変換が可能になります。

- 属性値を複数の CI から 1 つのターゲット CI にマッピングする。
- 複数の子 CI の属性（**container_f** を持つものや関係を持つもの）をターゲット・データ・ストアの親 CI にマッピングします。たとえば，**Number of CPUs** という値をターゲット Host CI に設定します。別の例としては，値 **Total Memory** を（UCMDB のホスト CI のすべてのメモリ CI のメモリ・サイズ値を合計して）ターゲット Host CI に設定できます。
- 複数の親 CI の属性（**container_f** を持つものや関係を持つもの）をターゲット・データ・ストアの CI にマッピングします。たとえば，UCMDB 内のソフトウェア CI の格納ホストから値を取得して，**Container Server** という値を **Installed Software** CI というターゲット属性に設定します。

以下に，カンマで区切られた 2 つのソース属性を使用してターゲット属性 **os** を作成する複合マッピングの例を示します。”

```
<target_attribute name="os" datatype="char">
    <map type="compoundstring">
        <source_attribute name="discovered_os_name" />
```

```
            <constant value="," />
            <source_attribute name="host_osinstalltype" />
        </map>
</target_attribute>
```

リンク方向の逆転

UCMDB にソースからソースへの構造が異なるデータを含めることができます。たとえば，HP Network Node Manager 統合で行うように，IpAddress CI と Interface CI の関係を **parent** にできます。または，Universal Discovery で一般的に作成される **containment** リンクにすることができます。さらに，これらのリンクの方向を反対にできます。

マッピング・ファイルでのリンク方向の逆転は現在はできません。**_end1** 変数と **_end2** 変数の逆転は，変換された XML でのデータの順序が切り替わるか，ソース・データでリンクが失われます。

この問題に対して考えられる解決策の 1 つは，エンリッチメント・ルールを次のように定義することです。

1. エンリッチメントの TQL パートは，プッシュ・アダプタによって使用される TQL のサブセットです。この TQL は特に，変換された xml で望ましい方向とは反対のすべてのリンクを選択します。
2. エンリッチメント・パートは正しい方向と望ましいタイプの新しいリンクを定義します。
3. エンリッチメントはアクティブになり，正しいリンクを作成します。
4. これにより統合ジョブ TQL は，元のリンクではなくエンリッチされたリンクを参照します。
5. さらにプッシュ・アダプタの <link> マッピングもエンリッチされたリンクを参照して，タイプと方向が一致したリンクのセットを生成します。

ファイルのマッピングのスキーマ

要素名およびパス	詳細	属性
integration	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	ソース・データ・リポ	1. **名前 :** type

要素名およびパス	詳細	属性
（integration > info）	ジトリに関する情報を定義する。	**説明 :** ソース・データ・リポジトリの名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** versions **詳細 :** ソース・データ・リポジトリのバージョン。 **必須かどうか :** 必須 **タイプ :** 文字列 3. **名前 :** vendor **説明 :** ソース・データ・リポジトリのベンダ。 **必須かどうか :** 必須 **タイプ :** 文字列
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。	1. **名前 :** type **説明 :** ソース・データ・リポジトリの名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** versions **詳細 :** ソース・データ・リポジトリのバージョン。 **必須かどうか :** 必須 **タイプ :** 文字列 3. **名前 :** vendor **説明 :** ソース・データ・リポジトリのベンダ。 **必須かどうか :** 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。	
source_ci_type_tree （integration > targetcis）	ソース CIT と，その CIT から継承するすべての CI タイプを定義する。	1. **名前 :** name **説明 :** ソース CIT の名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** mode **説明 :** 現在の CI タイプに必要な更新のタイプ。

要素名およびパス	詳細	属性
		必須かどうか : 必須 **タイプ :** 次のいずれかの文字列になります。 a. **insert :** CI がまだ存在していない場合にのみ使用する。 b. **update :** CI が存在していることがわかっている場合のみ使用する。 c. **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 を作成する。 d. **ignore :** この CI タイプの場合は何もしない。
source_ci_type （integration > targetcis）	ソース CIT を定義する（この CIT から継承する CI タイプは含まない）。	1. **名前 :** name **説明 :** ソース CIT の名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** mode **説明 :** 現在の CI タイプに必要な更新のタイプ。 **必須かどうか :** 必須 **タイプ :** 次のいずれかの文字列になります。 a. **insert :** CI がまだ存在していない場合にのみ使用する。 b. **update :** CI が存在していることがわかっている場合のみ使用する。 c. **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 を作成する。 d. **ignore :** この CI タイプの場合は何もしない。
target_ci_type （integration > targetcis > source_ci_type -または- integration > targetcis > source_ci_type_tree）	ターゲット CIT を定義する。	1. **名前 :** name **説明 :** ターゲット CI タイプの名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** schema **詳細 :** 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。 **必須かどうか :** 任意

要素名およびパス	詳細	属性
		タイプ : 文字列 3. **名前 :** namespace **詳細 :** ターゲットの該当の CI タイプの名前空間を示す。 **必須かどうか :** 任意 **タイプ :** 文字列
targetprimarykey （integration > targetcis > source_ci_type） -または- （integration > targetcis > source_ci_type_tree -または- （integration > targetrelations > link） -または- （integration > targetrelations > source_link_type_tree）	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 > targetcis > source_ci_type_tree > 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。	

要素名およびパス	詳細	属性
-または- （integration > targetrelations > source_link_type_tree > targetprimarykey）		
target_attribute （integration > targetcis> source_ci_type -または- integration > targetcis > source_ci_type_tree -または- integration > targetrelations > link -または- integration > targetrelations > source_link_type_tree）	ターゲット CIT の属性を定義する。	1. **名前 :** name **説明 :**ターゲット CIT の属性の名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** datatype **詳細 :**ターゲット CIT の属性のデータ型。 **必須かどうか :** 必須 **タイプ :** 文字列 3. **名前 :** length **詳細 :** 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）。 **必須かどうか :** 任意 **タイプ :** Integer 4. **名前 :** option **詳細 :** 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。 a. **uppercase** - 大文字に変換する b. **lowercase** - 小文字に変換する 該当の属性が空の場合，変換関数は適用されません。
map （integration > targetcis > source_ci_type > target_attribute -または- integration > targetcis > source_ci_type_tree > target_attribute）	ソース CIT の属性値を取得する方法を指定する。	1. **名前 :** type **詳細 :** 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。 a. **direct** - ソース属性の値からターゲット属性の値への 1 対 1 のマッピングを指定する。 b. **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

要素名およびパス	詳細	属性
-または- （integration > targetrelations > link > target_attribute -または- integration > targetrelations > source_link_type_tree > target_attribute）		定される。 c. **childattr** - サブ要素は 1 つ以上の子 CIT の属性になる。子 CIT は **composition** 関係または **containment** 関係のある CIT として定義されます d. **constant** - 静的な文字列。 2. **名前 :** value **詳細 :** 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 3. **名前 :** attr **詳細 :** 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
aggregation （integration > targetcis > source_ci_type > target_attribute > map -または- integration > targetcis > source_ci_type_tree > target_attribute > map -または- （integration > targetrelations > link > target_attribute > map -または- integration > targetrelations > source_link_type_tree > target_attribute > map） マップのタイプが **childattr** の場合にのみ	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 **詳細 :** aggregation 関数のタイプ **必須かどうか :** 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 – 含まれるすべての数値の合計を返します。 • **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。

要素名およびパス	詳細	属性
有効		
source_child_ci_type （integration > targetcis> source_ci_type > target_attribute > map -または- integration > targetcis > source_ci_type_tree > target_attribute > map -または- （integration > targetrelations > link > target_attribute > map -または- integration > targetrelations > source_link_type_tree > target_attribute > map） マップのタイプが **childattr** の場合にのみ有効	接続されたどの CI から子属性を取得するかを定義する。	1. **名前 :** name **詳細 :** 子 CI のタイプ **必須 :** 必須 **タイプ :** 文字列 2. **名前 :** source_attribute **詳細 :** マッピングされる子 CI の属性。 **必須 :**（同じパス上にある）**childAttr** aggregation タイプが count でない場合にのみ必要になる。 **タイプ :** 文字列
validation （integration > targetcis > source_ci_type > target_attribute > map -または- integration > targetcis > source_ci_type_tree > target_attribute > map -または- （integration > targetrelations > link >	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。	1. **名前 :** minlength **詳細 :** 指定した値よりも短い文字列を除外します。 **必須かどうか :** 任意 **タイプ :** Integer 2. **名前 :** maxlength **詳細 :** 指定した値よりも長い文字列を除外します。 **必須かどうか :** 任意 **タイプ :** Integer 3. **名前 :** minvalue **詳細 :** 指定した値よりも小さい数値を除外します。 **必須かどうか :** 任意

要素名およびパス	詳細	属性
target_attribute > map -または- integration > targetrelations > source_link_type_tree > target_attribute > map） マップのタイプが **childatt** の場合にのみ有効		**タイプ :** 数値 4. **名前 :** maxvalue **詳細 :** 指定した値よりも大きい数値を除外します。 **必須かどうか :** 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。	
source_link_type_tree （integration > targetrelations）	ソース関係タイプを（このタイプから継承するものは除外して）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。	1. **名前 :** name **詳細 :** ソース関係の名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** target_link_type **詳細 :** ターゲット関係の名前 **必須かどうか :** 必須 **タイプ :** 文字列 3. **名前 :** nameSpace **説明 :** ターゲットで作成されるリンクの名前空間。 **必須かどうか :** 任意 **タイプ :** 文字列 4. **名前 :** mode **説明 :** 現在のリンクに必要な更新のタイプ。 **必須かどうか :** 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。

要素名およびパス	詳細	属性
		• **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。 5. **名前 :** source_ci_type_end1 **詳細 :** ソース関係の End1 CI タイプ。 **必須かどうか :** 必須 **タイプ :** 文字列 6. **名前 :** source_ci_type_end2 **説明 :** ソース関係の End2 CI タイプ。 **必須かどうか :** 必須 **タイプ :** 文字列
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。	1. **名前 :** source_link_type **詳細 :** ソース関係の名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** target_link_type **詳細 :** ターゲット関係の名前。 **必須かどうか :** 必須 **タイプ :** 文字列 3. **名前 :** nameSpace **説明 :** ターゲットで作成されるリンクの名前空間。 **必須かどうか :** 任意 **タイプ :** 文字列 4. **名前 :** mode **詳細 :** 現在のリンクに必要な更新のタイプ。 **必須かどうか :** 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

要素名およびパス	詳細	属性
		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。 5. **名前 :** source_ci_type_end1 **詳細 :**ソース関係の End1 CI タイプ **必須かどうか :** 必須 **タイプ :** 文字列 6. **名前 :** source_ci_type_end2 **説明 :** ソース関係の End2 CI タイプ **必須かどうか :** 必須 **タイプ :** 文字列
target_ci_type_end1 （integration > targetrelations > link -または- integration > targetrelations > source_link_type_tree）	ターゲット関係の End1 CI タイプ。	1. **名前 :** name **説明 :** ターゲット関係の End1 CI タイプの名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** superclass **説明 :** End1 CI タイプのスーパークラスの名前。 **必須かどうか :** 任意 **タイプ :** 文字列
target_ci_type_end2 （integration > targetrelations > link -または- integration > targetrelations > source_link_type_tree）	ターゲット関係の End2 CI タイプ。	1. **名前 :** name **説明 :** ターゲット関係の End2 CI タイプの名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** superclass **説明 :** End2 CI タイプのスーパークラスの名前。 **必須かどうか :** 任意 **タイプ :** 文字列

結果のマッピングのスキーマ

要素名およびパス	詳細	属性
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。	
Object（root > data > objects）	単一のオブジェクトとその属性すべての更新操作を記述します。	1. **名前 :** name **説明 :** CI タイプの名前 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** mode **説明 :** 現在の CI タイプに必要な更新のタイプ。 **必須かどうか :** 必須 **タイプ :** 次のいずれかの文字列になります。 a. **insert** - CI がまだ存在していない場合にのみ使用する。 b. **update** - CI が存在していることがわかっている場合のみ使用する。 c. **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 を作成する。 d. **ignore** - この CI タイプの場合は何もしない。 3. **名前 :** operation **説明 :** 該当の CI で実行する操作。 **必須かどうか :** 必須 **タイプ :** 次のいずれかの文字列になります。 a. **add** - CI が追加される b. **update** - CI が更新される c. **delete** - CI が削除される 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。 4. **名前 :** mamId

要素名およびパス	詳細	属性
		説明 : ソース CMDB のオブジェクトの ID。 **必須かどうか :** 必須 **タイプ :** 文字列
field （root > data > objects> Object -または- root > data > links > link ）	オブジェクトの単一フィールドの値を記述します。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含まれる場合，値はエンドのいずれかの ID になります。各エンド ID はオブジェクトとして（<objects> の下に）表示されます。	1. **名前 :** name **説明 :** フィールドの名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** key **詳細 :** このフィールドがオブジェクトのキーであるかどうかを指定します。 **必須かどうか :** 必須 **タイプ :** ブール 3. **名前 :** datatype **詳細 :** フィールドのタイプ。 **必須かどうか :** 必須 **タイプ :** 文字列 4. **名前 :** length **詳細 :** 文字列 /文字のデータ型の場合，ターゲット属性の整数サイズです。 **必須かどうか :** 任意 **タイプ :** Integer
links (root > data)	更新するリンクのルート要素。	1. **名前 :** targetRelationshipClass **詳細 :** ターゲット・システム内の関係（リンク）の名前。 **必須かどうか :** 必須 **タイプ :** 文字列 2. **名前 :** targetParent **詳細 :** リンクの最初のエンドのタイプ（親）。 **必須かどうか :** 必須 **タイプ :** 文字列 3. **名前 :** targetChild **詳細 :** リンクの 2 番目のエンドのタイプ（子）。 **必須かどうか :** 必須 **タイプ :** 文字列 4. **名前 :** mode

要素名およびパス	詳細	属性
		説明 : 現在の CI タイプに必要な更新のタイプ。 **必須かどうか :** 必須 **タイプ :** 次のいずれかの文字列になります。 a. **insert** - CI がまだ存在していない場合にのみ使用する。 b. **update** - CI が存在していることがわかっている場合のみ使用する。 c. **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 を作成する。 d. **ignore** - この CI タイプの場合は何もしない。 5. **名前 :** operation **説明 :** 該当の CI で実行する操作。 **必須かどうか :** 必須 **タイプ :** 次のいずれかの文字列になります。 a. add - CI が追加される b. update - CI が更新される c. delete - CI が削除される 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。 6. **名前 :** mamId **説明 :** ソース CMDB のオブジェクトの ID。 **必須かどうか :** 必須 **タイプ :** 文字列

カスタマイズ

本項では，プッシュ・アダプタに対する一般的なカスタマイズの基本的手順のいくつかについて説明します。

属性の追加

1. 属性が TQL 結果に含まれていることを確認します。
2. 属性マッピングを正しい CI マッピング・セクション内のマッピング・ファイルに追加します。
3. データ・レシーバがデータ内の追加属性を受け取る準備ができていることを確認します。

属性の削除

属性を削除するには，属性をマッピング・ファイルから削除します。また結果や条件ノードでもはや使用されていない場合は，属性を TQL から削除する必要があります。

CI タイプの追加

1. CI タイプを TQL に追加します。
2. CI タイプとその属性データが TQL 結果に表示されていることを確認します（use calculate と preview を使用）。
3. CI タイプのマッピングをマッピング・ファイルに追加します。他の CI タイプのマッピングをコピーして，新しい CI タイプを素早く作成します。
4. コピーした XML の名前と属性マッピングを新しい CI タイプとその属性に対応するように変更します。使用できるマッピングのタイプについては，「ファイル参照のマッピング」(259ページ)を参照してください。

CI タイプの削除

1. CI タイプを TQL から削除します。
2. マッピング・ファイルでその CI タイプのマッピング・セクションを削除します。

リンクの追加

1. 2 つのエンド CI がデータに存在することを確認します。
2. 追加する必要があるリンクが実際に有効なリンクであることを確認します（CI タイプ・マネージャでチェック）。
3. mappings xml の関係セクションにリンク要素を追加します。

リンクの削除

1. マッピング・ファイルで削除するリンクのリンク・セクションを削除します。
2. 可能な場合は，TQL からリンクを削除します（TQL の効率や関数に影響を与えない場合）。

第8章: 汎用アダプタの開発

本章の内容

インスタンスの同期

プッシュおよびポピュレーションの汎用アダプタ操作は，インスタンス・データに対して使用します。*インスタンス*および*ルート*の概念の詳細については，「インスタンス・ベースのポピュレーション・フロー」(196ページ)および「汎用アダプタを使用したデータ・プッシュのアーカイブ」(275ページ)を参照してください。

汎用アダプタを使用したデータ・プッシュのアーカイブ

データ・プッシュは，XML スキーマに少量の変更を加えて，既存の拡張汎用プッシュ・アダプタ・フレームワークを使用します。

注: 汎用アダプタは，インスタンス・モードで動作します（単一 CI のタイプでは機能しないが，メイン・ルート CI 別にグループ化された CI の集合で機能することを意味します）。詳細については，「インスタンス・ベースのポピュレーション・フロー」(196ページ)を参照してください。

双方向マッピング動作に対応するために必要とされる XML スキーマの変更を次に示します。

- **<targetcis>** タグの名前が **<target_entities>** に変更されています。
- **<source_instance_type>** タグの名前が **<source_instance>** に変更されています。
- **<target_ci_type>** タグの名前が **<target_entity>** に変更されています。
- **<for-each-source-ci>** タグの名前が **<for-each-source-entity>** に変更されています。
- ヘッダの **versions** 属性の名前が **version** に変更され，小数点が不要になりました。

本項では，汎用アダプタ・フレームワークを使用したデータ・プッシュに関する情報を提供します。

プッシュの概要

汎用アダプタは，TQL クエリの結果を表すデータ構造に対して機能します。汎用アダプタ・フレームワークに基づき構築された各アダプタは，このデータ構造を処理して，必須ターゲットにプッシュします。

データ構造は **ResultTreeNode (RTN)** という名前が付けられています。RTN はアダプタのマッピング・ファイルと TQL クエリの結果にもとづいて作成されます。汎用アダプタ・フレームワークに使用するクエリはルートに基づいたものでなければなりません。つまりクエリには，要素名が **root** となっている 1 つのクエリ・ノードが含まれているか，プレフィックス **root** で始まる 1 つまたは複数の関係要素が含まれている必要があります。この CI または関係は，クエリのルート要素となります。詳細については，『HP Universal CMDB データ・フロー管理ガイド』のData Pushを参照してください。

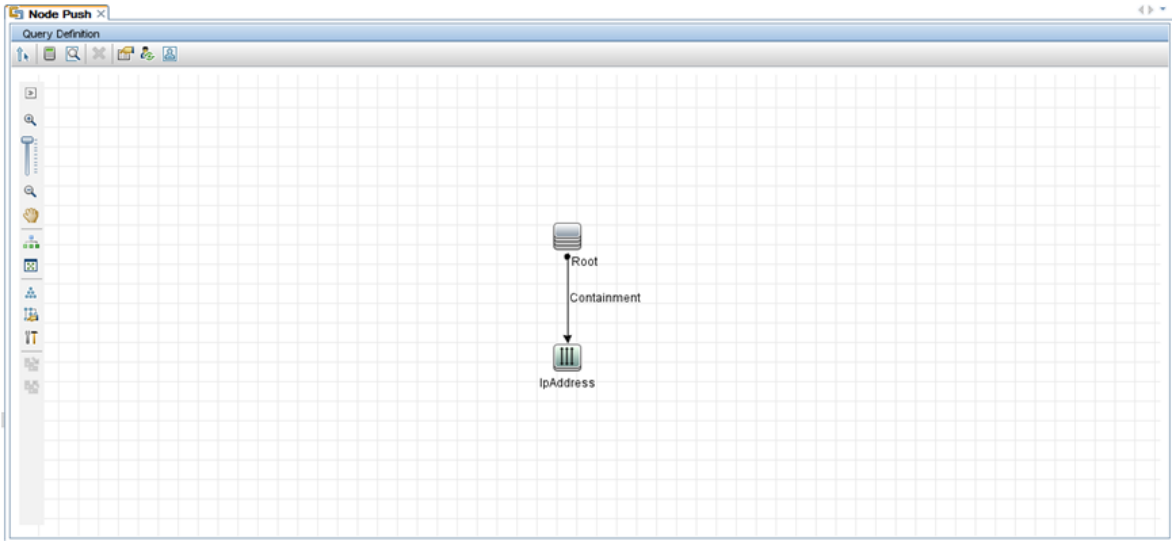
拡張プッシュ・アダプタの作成には，次の 2 つの基本ステップが関係しています。

1. PushAdapterConnector インタフェースの実行 - インタフェースは，追加，更新，削除するデータを，RTN のリストとして受信し，ターゲットへのプッシュを実行します。
2. マッピング・ファイルの作成 - マッピング・ファイルは，CI と TQL 結果の属性をマッピングすることにより，RTN 構造の作成を決定します。

マッピング・ファイル

次の例では，マッピング・ファイルの作成方法について示します。

この例では，ノードと IP アドレスのプッシュをシミュレーションします。次のように**Node Push** と呼ばれる TQL クエリを作成します。

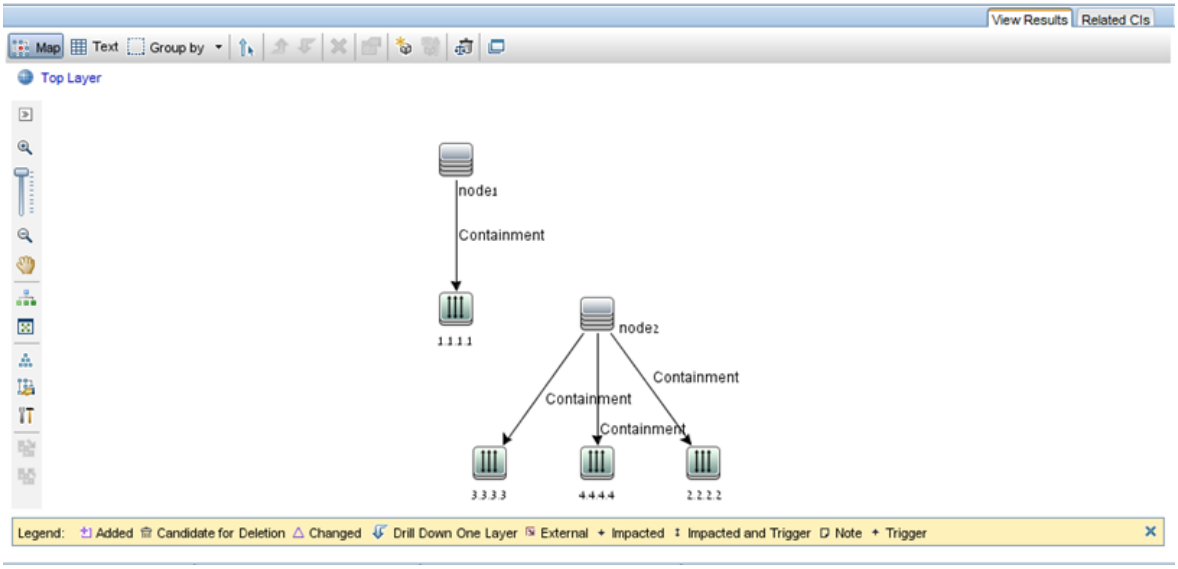


このマッピング・ファイルでは，**Computer** と **IP** という 2 つのターゲット CI タイプを作成します。Computer には 1 つの変数と 2 つの属性があります。IP には 1 つの属性があります。

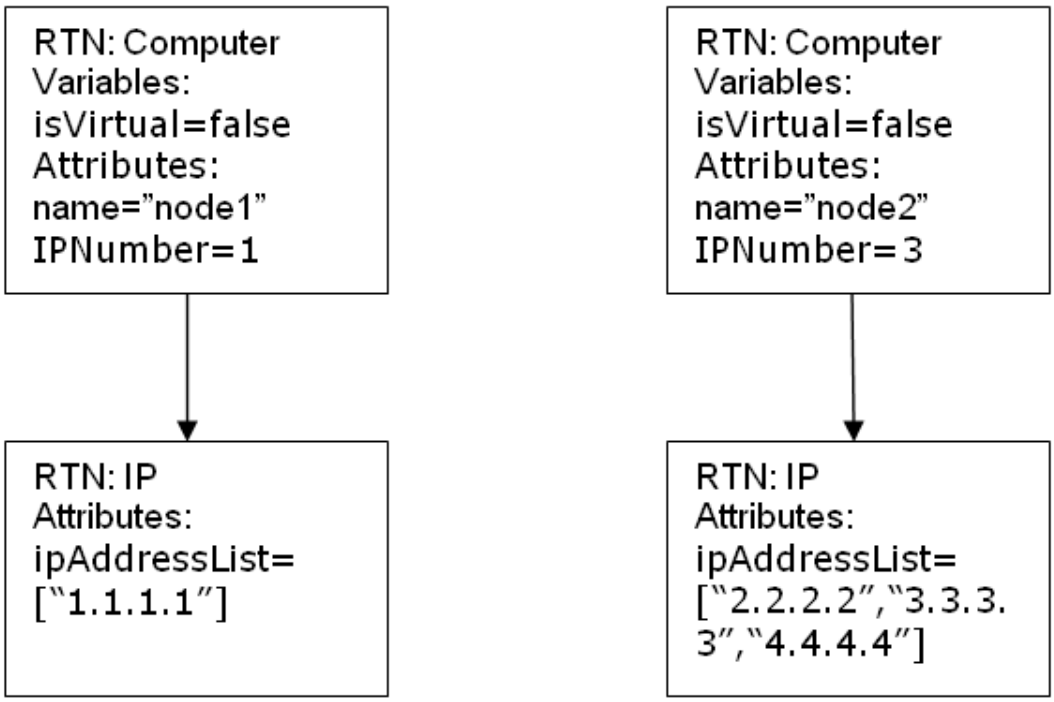
次にマッピング XML ファイルを示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <info>
        <source name="UCMDB" version="10.20" vendor="HP"/>
        <target name="PushProduct" version="9.3" vendor="HP"/>
    </info>

    <import>
        <scriptFile path="mappings.scripts.PushFunctions"/>
    </import>

    <target_entities>
        <source_instance query-name="Node Pusy" root-element-name="Root">
            <target_entity name="Computer" is-valid="(Root['root_iscandidatefordeletion'] == null) ? true : !Root['root_iscandidatefordeletion']">
                <variable name="isVirtual" datatype="BOOLEAN" value="PushFunctions.isVirtual(Root['root_class'])"/>
                <target_mapping name="name" datatype="STRING" value="Root['name']"/>
                <target_mapping name="ipNumber" datatype="INTEGER" value="Root.IpAddress.size()"/>
                <target_mapping name="description" datatype="STRING" value="PushFunctions.getDescription(isVirtual)"/>
                <target_entity name="IP">
                    <target_mapping name="IpAddressList" datatype="STRING_LIST" value="Root.IpAddress*.getAt('name')"/>
                </target_entity>
            </target_entity>
        </source_instance>
    </target_entities>

</integration>
```

クエリ結果は次のように表示されます。



このマッピング・ファイルに従って構成された RTN リストを次に示します。



各ルート・インスタンスは，マッピング・ファイルを使用して別個にマップされます。この例では，PushAdapterConnector は 2 つの RTN ルートのリストを受信します。

注: 以前のプッシュ・アダプタには，CI タイプの全体マッピングを作成する機能がありました。新しいプッシュ・アダプタ・マッピングは TQL クエリごとになっています。**x** と名付けられたクエリを使用するプッシュ・ジョブの実行中に，アダプタは関連するマッピング・ファイル（属性

query-name=x を持つもの）を検索します。

Groovy スクリプト言語を使用してマッピング・ファイルの値を計算できます。詳細については，「Groovy Traveler」(279ページ)を参照してください。

Groovy Traveler

TQL クエリ結果に次の方法でアクセスします。

- **Root[attr]** は，Root 要素の **attr** 属性を返します。
- **Root.Query_Element_Name** は，TQL で Query_Element_Name で名付けられ，現在のルート CI にリンクされた CI インスタンスのリストを返します。
- **Root.Query_Element_Name[2][attr]** は，現在のルート CI にリンクされた 3 番目の Query_Element_Name の属性 **attr** を返します。
- **Root.Query_Element_Name*.getAt(attr)** は，TQL で Query_Element_Name と名付けられ，現在のルート CI にリンクされた CI インスタンスの属性 **attr** のリストを返します。

次の属性にも Groovy Traveler でアクセスできます。

- **cmdb_id** – CI または関係の UCMDB ID を文字列として返します。
- **external_cmdb_id** – CI または関係の外部 ID を文字列として返します。
- **Element_type** – CI または関係の要素タイプを文字列として返します。

インポート・タグ：

```
<import>
<scriptFile path="mappings.scripts.PushFunctions"/>
</import>
```

これは，マッピング・ファイルのすべての Groovy スクリプトに対するインポートの宣言を意味します。この例では，**PushFunctions** は，静的関数を含む Groovy スクリプト・ファイルで，マッピング時にアクセスできます（つまり value=” PushFunctions.foo()” となります）。

source_instance_type

マッピングは TQL ごとに行われ，クエリ名の値は現在のマッピングの関連 TQL となります。 ‘*’ は，マッピング・ファイルがプレフィックス「Node Push」で始まるすべての TQL クエリに関連付けられていることを意味します。

```
<source_instance_type query-name="Node Push*" root-element-name="Root">
```

source_instance_type タグは，マッピングするルート要素を指定します。

root-element-name は，TQL のルート名と完全に同じである必要があります。

target_entity

このタグは RTN の作成に使用されます。

名前属性は target_entity 名 :name=Computer を表します。

is-valid属性は，マッピング時に計算されるブール値で，現在の target_ci が有効であるかどうかを判別します。無効な target_entities は RTN には追加されません。この例では UCMDB の **root_iscandidatefordeletion** 属性が true となる target_entity インスタンスは作らないほうがよいでしょう。

target_entity は，次に示すように，マッピング時に計算される変数を持っています。

```
<variable name="vSerialNo" datatype="STRING" value="Root['serial_number']"/>
```

変数 **vSerialNo** は，現在のルートの **serial_number** の値を取得します。

RTN の属性は **target_mapping** タグによって作成されます。**value** フィールドの Groovy スクリプトの実行結果は RTN 属性に割り当てられます。

```
<target_mapping name="SerialNo" datatype="STRING" value="vSerialNo"/>
```

SerialNo は，変数 **vSerialNo** の値を割り当てます。

次のように，target_entity 別の target_entity 子として定義することが可能です。

```
<target_entity name="Portfolio">
<variable name="vSerialNo" datatype="STRING" value="Root['global_id']"/>
<target_mapping name="CMDBId" datatype="STRING" value="globalId"/>
<target_entity name=Asset">
<target_mapping name="SerialNo" datatype="STRING" value="vSerialNo"/>
</target_entity>
</target_entity>
```

RTN **Portfolio** には **Asset** と名付けられた子 RTN があります。

for-each-source-entity

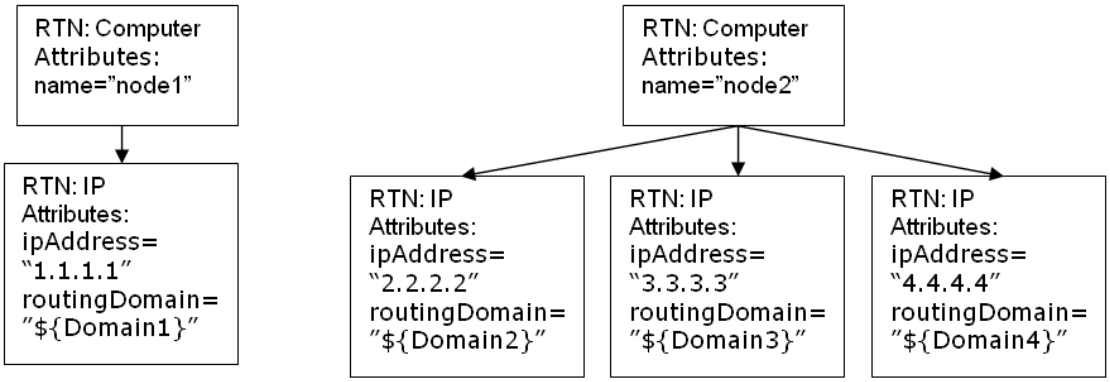
このタグは，ルート・インスタンスの特定 CI のリストを示します。次のフィールドがあります。

- **source-entities=" "** – ターゲット CI が作成される CI のリスト。このリストは，**Root.IpAddress** フィールドで Groovy Traveler によって定義されます。
- **count-index=" "** – 現在のループ反復で CI のインデックスを保持する変数。
- **var-name=" "** – 現在のループ反復の CI の名前。

ここでサンプルのマッピング・ファイルを修正してみましょう。

```
<target_entity name="Computer">
    <target_mapping name="name" datatype="STRING" value="Root['name']"/>
    <for-each-source-entity count-index="i" var-name="currIP" source-entities="Root.IpAddress">
        <target_entity name="IP">
            <target_mapping name="ipAddress" datatype="STRING" value="Root.IpAddress[i]['name']"/>
            <target_mapping name="routingDomain" datatype="STRING" value="currIP['routing_domain']"/>
        </target_entity>
</target_entity>
```

このマッピング・ファイルに従って構成される RTN リストは以下のようになります。



dynamic_mapping

このタグは，RTN 構造の作成時に，ターゲット・データ・ストアからデータのマッピングを作成する機能を追加します。

例:ターゲットが UCMDB で **Node.name** に関連付けられた **id** 列と **name** 列がある **Computer** と名付けられたテーブルを持つデータベースであると仮定しましょう。どちらの列も一意のものです。このデータベースには，コンピュータ・テーブルで **parentID** への参照キーを持つ **IP** と名付けられたテーブルもあります。「dynamic_mapping」は，名前と ID を <name,id> として格納するマップを作成できます。このマップに基づき，アダプタは ID とコンピュータを一致させ，IP テーブルの **parentID** 属性に正しい値をプッシュできるようになります。RTN の作成時には，このマップを使用して，値を parentID 属性にマッピングできます。

マッピングは **map_property** により決定されます。dynamic_mapping は各チャンクで 1 回実行されます。

```
<dynamic_mapping name="IdByName " keys-unique="true">
```

name 属性はマップの名前を表します。**keys-unique** 属性は，キーが一意のものであるかどうかを示します（各キーは 1 つの値または 1 つの値セットにマッピングされます）。

この例のマップの名前は **IdByName** で，一意のキーを持っています。スクリプトでマップにアクセスするには，次のコマンドを実行します。

```
DynamicMapHolder.getMap(‘IdByName’)
```

そのマップへの参照が返されます。

map_property タグは，マッピングの元となるプロパティを作成します。

例 :

```
<map_property property-name="SQLQuery" datatype="STRING"
```

```
property-value="SELECT name, id FROM Computer"/>
```

この例では，プロパティの名前は **SQLQuery** で，その値は，マップを作成する SQL ステートメントです。PushConnector インタフェースのメソッド **retrieveUniqueMapping** と **retrieveNonUniqueMapping** を実行すると，返されたマップの実際の内容を決定できます。

グローバル変数

次のグローバル変数は，マッピング・ファイルの Groovy スクリプトにアクセスできます。

- **Topology** – タイプ :Topology。現在のチャンクのトポロジのインスタンス。
- **QueryDefinition** - タイプ :QueryDefinition。現在の TQL のクエリ定義のインスタンス。
- **OutputCI** – タイプ :ResultTreeNode。現在のツリー・マッピングのルート要素の RTN。
- **ClassModel** – タイプ :ClassModel。クラスモデルのインスタンス。
- **CustomerInformation** – タイプ :CustomerInformation。ジョブを実行している顧客に関する情報。
- **Logger** – タイプ :DataAdapterLogger。このロガーは，UCMDB ログ・フレームワークにログを記録するためのアダプタで使用できます。

Groovy スクリプトの記述

本項では，**PushFunctions.groovy** ファイルを作成します。このファイルには，ルート・インスタンスのマッピング時に使用される静的関数が含まれます。

```
package mappings.scripts

public class PushFunctions {

    public static boolean isVirtual(def nodeRole){
        return isListContainsOne(def list, "MY_VM", "MY_SIMULATOR");
    }



public static String getDescription(boolean isVirtual){
        if(isVirtual){
            return "This is a VM";
        }
        else{
            return "This is physical machine";
        }
    }



private static boolean isListContainsOne(def list, ...stringList){
        //リストが値のいずれかを含んでいる場合に true を返します。
```

```
    }
}
```

PushAdapterConnector インタフェースの実装

実装は，次の基本ステップをサポートしている必要があります。

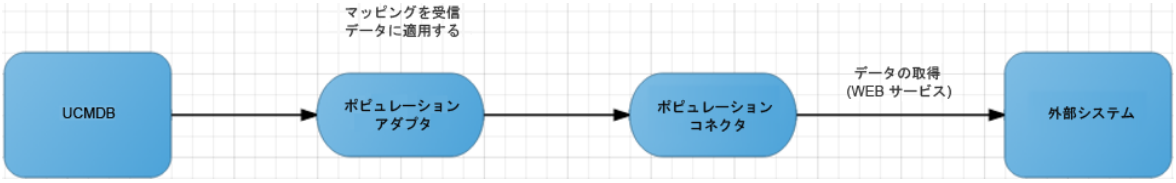
```
public class PushExampleAdapter implements PushAdapterConnector

{

public UpdateResult pushTreeNodes(PushConnectorInput input) throws DataAccessException{

// 1. UpdateResult インスタンスのビルド - UpdateResult は，送信された ID 間のマッピングを，
データ・ストアに入力された実際の ID に返すために使用されます。
// また，実際にプッシュされたデータのステータス，失敗した ID をレポートする詳細ステータ
ス，および成功した ID で実際に実行されたアクションを渡せるようにする更新ステータスもあ
ります。
// 2. データの処理 :
// a. 追加するデータを処理します。input.getResultTreeNodes.getDataToAdd();
//  b. 更新するデータを処理します。
//  c. 削除するデータを処理します。
// 3. 更新結果を返します。
    }

public void start(PushDataAdapterEnvironment env) throws DataAccessException{
    // このメソッドは統合ポイントの作成時または
アダプタが再読み込み時
    //（たとえば，マッピング・ファイルのいずれかを変更し，
    //「保存」を押した後）に呼び出されます。

}



public void testConnection(PushDataAdapterEnvironment env) throws DataAccessException {
        // このメソッドは，統合ポイントの
作成で「テスト接続」ボタンを
        //押したときに呼び出されます。
        // たとえば，データを RDBMS にプッシュすると，
このメソッドはデータベースへの接続を
        //作成し，ダミーの SQL ステートメントを実行します。
        // 失敗した場合，ログにエラー·メッセージを書き込み，
例外をスローします。
    }
```

```
Map<Object, Object> retrieveUniqueMapping(MappingQuery mappingQuery){
//このメソッドは，所定の mappingQuery に従ってマップを作成します。このメソッドは，
「UpdateResult pushTreeNodes」 メソッドの前に，
// アダプタ実行のマッピング段階で呼び出されます。
// このメソッドは，「dynamic_mapping」タグの「keys-unique」属性が true の場合に呼び出されます。
}

Map<Object, Set<Object>> retrieveNonUniqueMapping(MappingQuery mappingQuery){
 // このメソッドは，「dynamic_mapping」タグの「keys-unique」属性が false の場合に呼び出されます。
// この場合，キーはいくつかの値にマッピングできます。
}
}
```

汎用アダプタを使用したデータ・ポピュレーションのアーカイブ

本項の内容

ポピュレーション・フレームワークのアーキテクチャ



ポピュレーション・フレームワークのメカニズムは，プッシュ・アダプタ・フレームワークのメカニズムと似ています。ポピュレーション・アダプタ・フレームワークのユーザは，マッピング・ファイルおよびコネクタ実装を提供し，それらを UCMDB アダプタ・パッケージにバンドル化する必要があります。

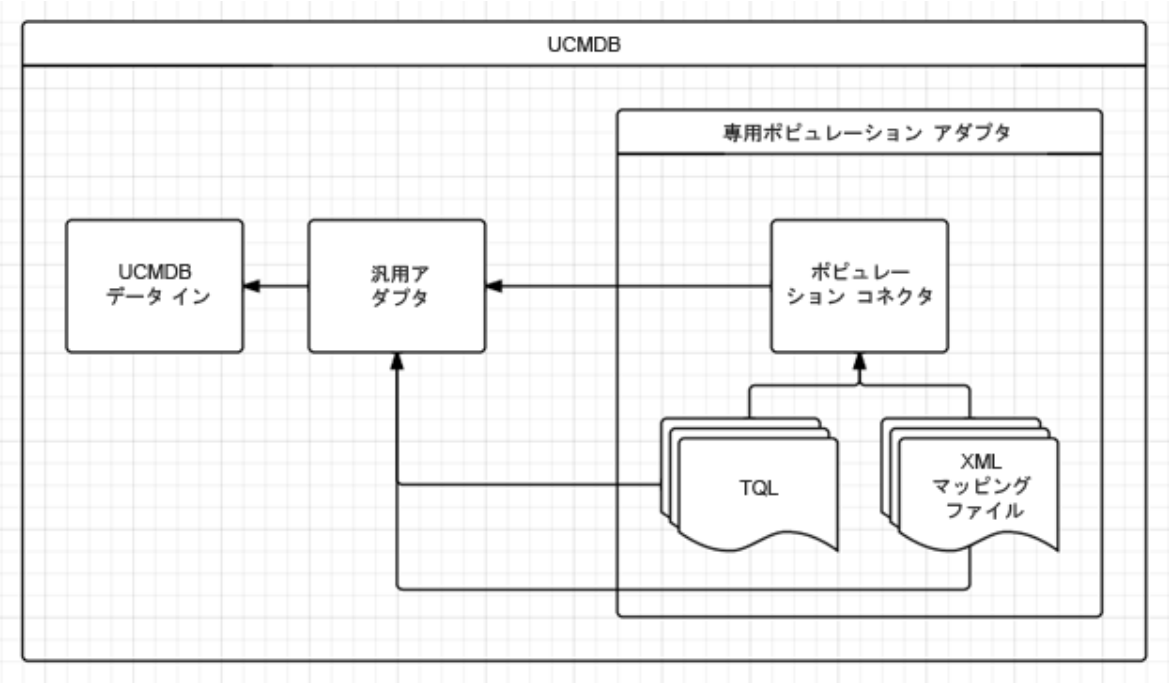
この操作には，次のステップが含まれます。

1. UCMDB ユーザがポピュレーション操作を UI からトリガする。
2. コマンドがポピュレーション・アダプタに送信される。
3. ポピュレーション・アダプタがポピュレーション・コネクタを呼び出し，データをチャンクで受信する。
4. ポピュレーション・アダプタが定義済みのマッピングを各チャンクのデータに適用し，そのデータを UCMDB サーバに転送する。

ポピュレーションに関連する主な作成物

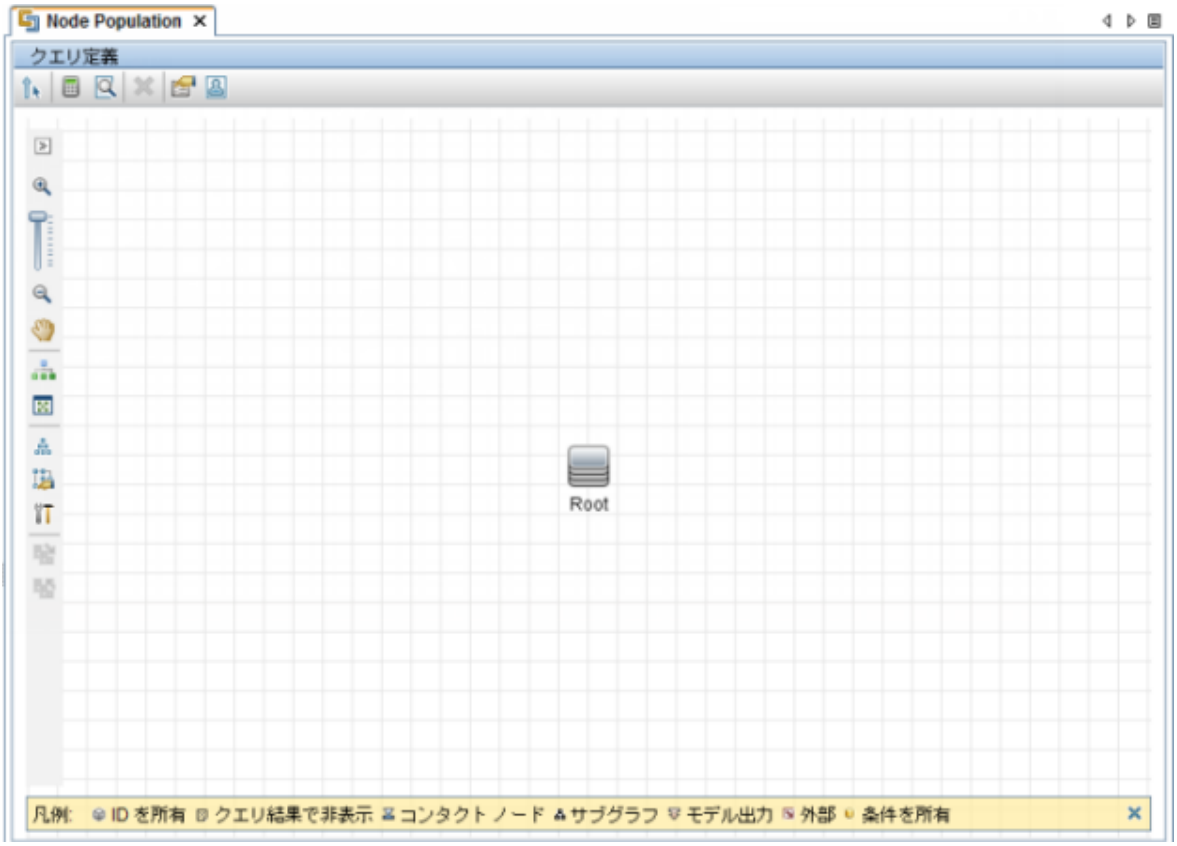
ポピュレーションに関連する主な作成物を次に示します。

- UCMDB にポピュレートされるデータを指定する TQL クエリ
- コネクタによって返されたデータがどのように UCMDB データにマップされるかを指定する XML マッピング
- 必須データ
- 外部システム・データを取得し，それを UCMDB 汎用アダプタに返すポピュレーション・コネクタ



ポピュレーション TQL クエリ

ポピュレーション TQL クエリの役割は，UCMDB にポピュレートされるデータを示すことにあります。たとえば，次の図の TQL は，ノード・インスタンスを UCMDB に取り込むために使用されます。



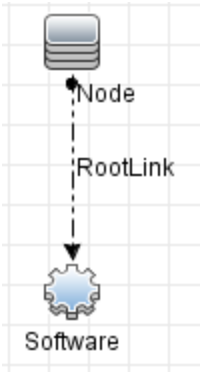
ポピュレーション・コネクタは，ポピュレーション TQL クエリを理解し，外部システムから必須データを提供します。

ポピュレーション・マッピング・ファイル

XML マッピング・ファイルは，その方向が逆であるという点を除いて，プッシュ操作と同じ目的を持ちます。このマッピング・ファイルは，コネクタによって返されたデータがどのように UCMDB データにマップされるかを記述します。

ここで提供する情報は，拡張汎用プッシュ・アダプタですでに取り上げられていない，ポピュレーション・マッピングに関連します。

UCMDB ノードおよび実行中のソフトウェアのマッピングの例を次に示します。最初の画像は，ノードとソフトウェアのポピュレーション TQL クエリを示しています。2 番目の画像は，ノードとソフトウェアのポピュレーション・マッピングを示しています。




```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Nodes And Software Population" root-element-name="PC">
        <!-- need to match case in UCMDB TQL -->
        <target_entity name="RootLink">
            <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
        </target_entity>
        <target_entity name="Node" type="Util.getNodeType(PC)">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC['description']"/>
        </target_entity>
        <target_entity name="Software">
            <target_mapping name="name" datatype="STRING" value="PC.Programs[0]['name']"/>
            <target_mapping name="root_container_name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="product_name" datatype="STRING" value="'vmware_hypervisor'"/>
        </target_entity>
    </source_instance>
</target_entities>
```

このポピュレーション・ジョブは，ResultTreeNode（RTN）PC の形式で外部システムからデータをフェッチします。ResultTreeNode API は，拡張汎用プッシュ・アダプタによって導入され，UCMDB Server **lib** フォルダの **push-interfaces.jar** ファイル内で見つけることができます。詳細については，「汎用アダプタを使用したデータ・プッシュのアーカイブ」(275ページ)を参照してください。

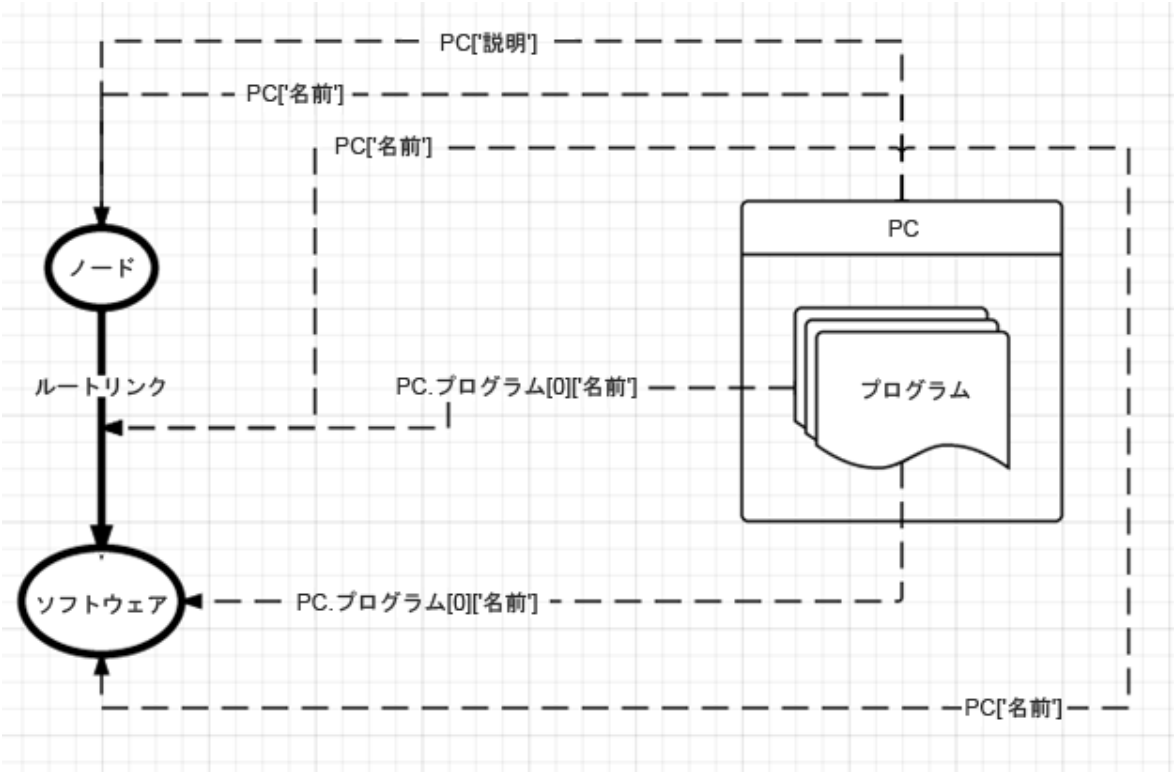
PC RTN には，属性の形式による一般的なノード情報が含まれています。さらに，関連する属性を持つソフトウェア・タイプ・エンティティを含む埋め込み Programs エンティティも含まれています。

1 つの PC インスタンスが UCMDB の 3 つのエンティティにマップされます。

- ノード CI
- 実行中のソフトウェア CI
- 構成リンク CI

PC インスタンスの形式の詳細については，「ポピュレーション要求の出力」(296ページ)を参照してください。

コネクタ・データがUCMDBデータにマップされる方法を次の図に示します。



次に，重要な行の分析を示します。

```
<!--The query name must match the one selected in the UI-->
<source_instance query-name="Nodes And Software Population" root-element-name="PC">
```

source_instance 定義は，エンティティをUCMDBに送ること，およびこれらのエンティティをグループ化するUCMDBトポロジがノードとソフトウェアのポピュレーションTQLクエリによって定義されることを示します。さらに，UCMDBデータの作成に使用される，コネクタによって返されるデータ構造は，**PC** という名前の **ResultTreeNode** です。

```
<target_entity name="RootLink">
    <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
</target_entity>
```

target_entity タグは，新規UCMDBエンティティがここから始まること，およびこのエンティティがノードとソフトウェアのポピュレーションTQLクエリ内部の **RootLink** 要素に対応することを示します。この指示には，新規エンティティのUCMDB CIタイプも含まれます。

作成される **RootLink** エンティティには，**name** という1つの属性が含まれます。この属性の値は，**Computer_22 has MySQL Server** のようになります。

このサンプルのマッピングでは，手動によるリンク・ポピュレーションが使用されています。「自動リンク・ポピュレーション」(289ページ)で説明されている自動の方法を使用することをお勧めします。

ポピュレーション type 属性

```
<target_entity name="Node" type="Util.getNodeType(PC)">
```

ノード・エンティティに **type** 属性があることに注意してください。この **type** は，このエンティティが UCMDB 内で持つことになる CI タイプを正確に示します。エンティティの標準設定の作成タイプはそのエンティティが参照する TQL 要素（この例では，ノード）から取得されるため，**type** 属性は必須ではありません。ただし，UCMDB ノード CI タイプの複数のインスタンスを返したいときに，一部のインスタンスが Windows で，ほかのインスタンスが Unix である場合，**type** 属性を使用して，正確な UCMDB 作成タイプを指定することができます。したがって，このケースでは，PC を入力として受け取り，有効な UCMDB CI タイプ識別子（Unix の場合「unix」，Windows の場合「nt」）を返す **getNodeType** 関数を **Util** 関数スクリプト・ファイル内部に作成します。

> **注: target_entity type** 属性は，ポピュレーション・フローのコンテキストでのみ利用できます。その値は有効な Groovy 表現である必要があります。

同様の方法で，ソフトウェア・エンティティの作成も記述することができます。

自動リンク・ポピュレーション

「ポピュレーション・マッピング・ファイル」のマッピング・サンプルでは，ポピュレートされたリンクを明示的にマップするために必要な内容を確認しました。マップされたターゲット・エンティティは，次に示すような各 TQL リンク要素に存在する必要があります。

```
<target_entity name="RootLink">
    <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
</target_entity>
```

汎用アダプタの自動リンク・ポピュレーション・メカニズムを使用すると，次に示すように，マッピング・セクションを含む TQL リンク要素をマップする必要がなくなります。フレームワークにより，TQL で指定したタイプのリンク CI インスタンスが空のプロパティで生成されます。この操作は，ポピュレーション TQL クエリ内のすべてのリンクに対して実行されます。

サンプル・マッピングでは，ノードおよび実行中のソフトウェア CI インスタンスというリンク・エンド（end1 および end2）を持つ作成中のタイプ構成を使用したリンク CI が結果として生じます。

ポピュレートするリンクで次の項目が必要な場合は，手動リンク・ポピュレーションを使用する必要があります。

- 動的リンク・タイプ（**type** 属性を使用）
- リンク・プロパティ

手動リンク・ポピュレーション

汎用アダプタは，リンクによって必要とされる 3 つのエンティティを定義（マッピング）することで，リンクのポピュレーションをアーカイブします。

- リンク・エンティティ
- リンクのエンド 1 エンティティ
- リンクのエンド 2 エンティティ

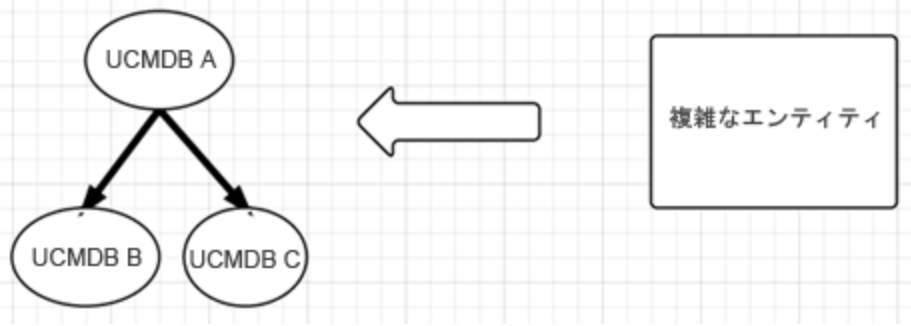
「ポピュレーション・マッピング・ファイル」(286ページ) に示すマッピングの例を分析してみましょう。この場合，ノード，実行中のソフトウェア，それらの間の構成リンクという 3 つのエンティティを UCMDB にポピュレートします。ここでは，リンク（タイプ「構成」の **RootLink** という名前のリンク）をポピュレートするため，2 つのリンク・エンドをマップする必要があります。したがって，TQL クエリを見ると，マッピングが必要なエンティティがノード（エンド 1）およびソフトウェア（エンド 2）であることがわかります。汎用アダプタ・フレームワークは，TQL クエリにおける作成済みのエンティティの要素名と定義に基づき，リンク構造を理解します。ポピュレーション・ジョブはノードおよび実行中のソフトウェアのインスタンスも取得する必要があるため，必要なエンドのマッピングがすでに配置されています。

リンク・ポピュレーションのタイプ

リンク・ポピュレーションを使用する状況は次の 2 種類があります。

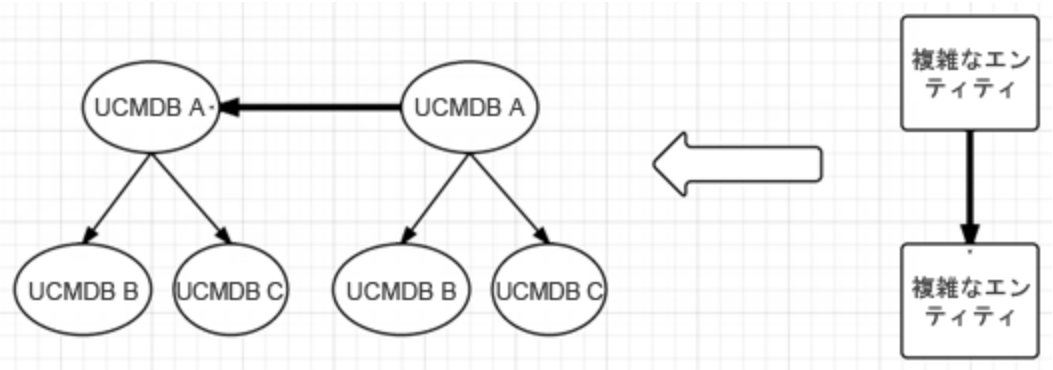
- 複雑な外部エンティティを複数の関連 UCMDB エンティティに分解する

  この場合，PC などの複雑な外部エンティティが UCMDB **ノード**および**実行中のソフトウェア**のタイプに変換されます。これらのタイプは**構成**リンクによりリンクされる必要があります。この種類のリンクは，UCMDB のコンテキストにのみ存在します。



- 複雑な外部エンティティ間のリンク

  この場合，PC などの 2 つの複雑な外部エンティティ間でリンクをモデル化する必要があります。



ポピュレーション・コネクタ

ポピュレーション·コネクタは，外部システム・データの取得を行います。このデータは，確立された API 形式（**ResultTreeNode**）により汎用アダプタに渡されます。次に，UCMDB データ・構造にマップされ，データイン・プロセスを介して UCMDB に挿入されます。

プッシュ・コネクタと同様に，ポピュレーション・コネクタは，Java および Groovy の両方で実装可能で，次の図に示すように，ポピュレーション・コネクタ Java インタフェースを実装する必要があります。

ポピュレーション・コネクタを設定するには，次の行をアダプタ構成 XML ファイルに追加します。

```
<adapter-settings>
    <adapter-setting name="PopulationConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPopulationConnector</adapter-setting>
```

```
public interface PopulationAdapterConnector extends GenericConnector, DynamicMappingConnector {

    /**
     * Executes a population request and provides the entities that will be populated in the format of
     * {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode}
     * Used for both population and federation with the flow type flag which is present in the
     * {@link com.hp.ucmdb.adapters.population.connector.PopulationConnectorInput}
     *
     * @param input population request
     * @return a population response
     * @throws DataAccessException
     */
    PopulationConnectorOutput populate(PopulationConnectorInput input) throws DataAccessException;

    /**
     * Returns the collection of queries the current connector supports for population
     * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
     * <p/>
     * The method also supports return of queries with their folders path: E.g.
     * "Folder/Secondary Folder/Query Name 1"
     * "Folder/Secondary Folder/Query Name 2"
     *
     * @param env the adapter environment
     * @return a collection of the supported queries
     */
    Collection<String> getPopulationQueries(DataAdapterEnvironment env);

    /**
     * Returns the collection of queries the current connector supports for federation
     * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
     * <p/>
     * The method also supports return of queries with their folders path: E.g.
     * "Folder/Secondary Folder/Query Name 1"
     * "Folder/Secondary Folder/Query Name 2"
     *
     * @param env the adapter environment
     * @return a collection of the supported queries
     */
    Collection<String> getFederationQueries(DataAdapterEnvironment env);

    /**
     * Update target id (global id) for each source object.
     *
     * @param idMapping mapping between source id (external id) and target id (string)
     */
    void updateGlobalIDsFromTarget(FCmdbExtenalToTargetIdMappingSet idMapping);

    /**
     * This methods reports population queries resources used by the adapter.<br>
     * This allows editing these resources directly from the Integration Studio.
     *
     * @param input  the requested information
     * @param output the returned resources
     * @see com.hp.ucmdb.federationspi.adapter.resource.PushQueriesResourceLocator
     */
    void locatePopulationQueriesResources(DataAdapterEnvironment env, LocatePopulationQueriesResourcesInput input,
                                          LocatePopulationQueriesResourcesOutput output);

    /**
     * Returns the collection of classes the current adapter supports for query.<br>
     * Notes:<br>
     * 1. This method is independent of the adapter life cycle (i.e. start/shutdown methods).<br>
     * 2. If adapter configuration (xml) defines supported classes, this method doesn't need to be implemented (return null).<br>
     *
     * @param env the Adapter env
     * @return collection of the supported classes
     * @throws DataAccessException in case of an error
     */
    Collection<SupportedClassConfig> getSupportedClasses(DataAdapterEnvironment env);
}
```

最初のメソッド populate は，外部システムからデータを取得するメイン・コネクタ・メソッドです。このメソッドは，ポピュレーション TQL クエリを入力として受け取り，汎用 *ResultTreeNode* 形式でその結果を返します。詳細については，「汎用アダプタを使用したデータ・プッシュのアーカイブ」(275ページ)を参照してください。メイン・ビジネス・データとともに，コネクタはステータスおよびチャンキング情報も返します。

2 番目のメソッド getSupportedQueries は，ポピュレーション・コネクタによってサポートされる TQL を示します。

3 番目と 4 番目のメソッドは，ポピュレートされたデータの ID のプッシュ・バック，特定のクエリのアダプタ内での関連するポピュレーション・リソースの特定といった，より高度な使用例で使用します。これらの API の詳細については，**push-interfaces.jar** ファイルを参照してください。

ポピュレーション要求の入力

ポピュレーション要求は，UCMDB ポピュレーション・クエリを記述する **QueryDefinition** オブジェクトによって定義されます。このクエリ・オブジェクトの読み取りおよび外部システムのクエリ言語への翻訳はポピュレーション・コネクタが行います。

```
 * @since 10.20
 */
public interface PopulationConnectorInput {

    QueryDefinition getQueryDefinition();

    /**
     * Indicates the required {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode} structure
     * that the population result must return.
     *
     * For the target mapping <target_mapping name="lMaxMemory" datatype="LONG" value="Root.VMWare_Host_Resource['vm_memory_limit']"/>
     * the resulting tree node should have the following structure: VMWare_Host_Resource will be a child of Root and vm_memory_limit will
     * be an attribute of VMWare_Host_Resource
     *
     * @return a map containing simplified result trees
     * @see PopulationConnectorOutput#getResultTreeNodes()
     */
    Map<String, ResultTreeNodeStructure> getResultTreeNodeStructure();

    /**
     * Returns the flow type of the operation.
     * Can be POPULATION or FEDERATION
     *
     * @return the flow type
     */
    FederationTopologyAdapterInput.FlowType getFlowType();

    /**
     * Returns the date from the last sync
     *
     * @return the date
     */
    Date getFromDate();
```

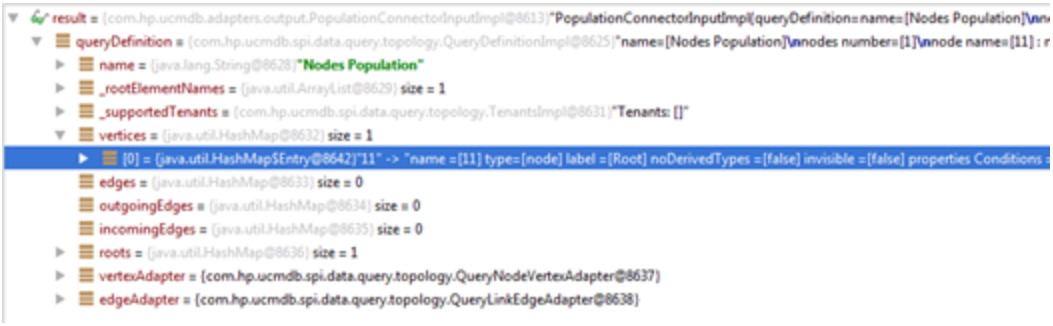
さらに，QueryDefinition オブジェクトに加え，次の項目が存在します。

- **getResultTreeNodeStructure** – ポピュレーション結果が返す必要のある，要求された構造を示します。
- **getFlowType** – コネクタに対する要求が POPULATION または FEDERATION のタイプであるかを判別するために使用されます。

getFromDate – 前回の同期の日付を示します。日付が NULL の場合，FULL POPULATION が実行されま

す。そうでない場合は，Diff POPULATION が実行されます（フロー・タイプが FEDERATION である場合，getFromDate メソッドは常に NULL を返します）。

例のポピュレーション要求を次の図に示します。



この例では，要求に **Nodes Population** クエリが含まれています。このクエリには，タイプが**ノード**の 1 つのみの TQL 要素が含まれています。

ResultTreeNodeStructure

PopulationAdapterConnector を実装するには，UCMDB ポピュレーション TQL を読み取り，UCMDB が要求している内容を理解し，外部システム・エンティティを使用して結果を提供する必要があります。たとえば，UCMDB が外部システムのビジネス・サービス・インスタンスに関連するすべてのノードを要求している場合に，コンピュータの外部システム同等が**サービス**・エンティティに関連する **PC** であると想定します。したがって，ポピュレーション・コネクタは**サービス**のインスタンスに接続されている **PC** のインスタンスを返す必要があります。この場合，マッピングは次のようになります。

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="PC">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService">
            <target_mapping name="name" datatype="STRING" value="PC.Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC.Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

この場合，**サービス**・インスタンスに関連する **PC** インスタンスを返すには，子ノードとして**サービス**を含む **PC** RTN を返すことになります。ただし，次のように，**PC** の子を含む**サービス** RTN の形式でマッピングを作成するように選択することができます（マッピングを無効の状態にする）。

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="Service">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="Service.PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService">
            <target_mapping name="name" datatype="STRING" value="Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

したがって，ポピュレーション・コネクタ開発の支援を目的として，汎用アダプタによって送信されるポピュレーション要求には，マッピング・ファイルで使用されるデータの RTN 構造も含まれます。これにより，返される RTN の必須形式が実装中のコネクタに示されます。

最初のケースでは，**ResultTreeNodeStructure** は次のようになります。

PC
- name

  Service
  - name
  - description

2 番目のケースでは，**ResultTreeNodeStructure** は次のようになります。

Service
- name
- description

  PC
  - name

ポピュレーション要求の出力

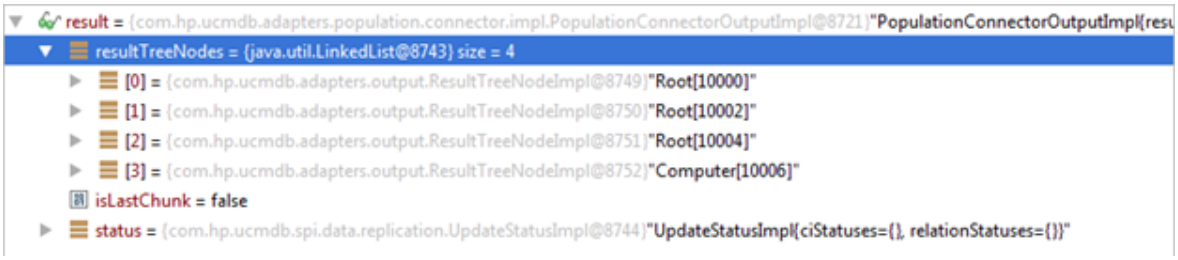
ポピュレーション要求の処理時に，ポピュレーション・コネクタは PopulationConnectorOutput を返す必要があります。

```
public interface PopulationConnectorOutput {

    /**
     * The result trees representing the external entities in (@link com.hp.ucmdb.adapters.push.output.ResultTreeNode) format.
     *
     * @return a list of result trees or an empty list
     */
    List<ResultTreeNode> getResultTreeNodes();

    /**
     * Adds a result tree to the population output object.
     *
     * @param resultTreeNode the result tree to add
     */
    void addResultTreeNode(ResultTreeNode resultTreeNode);

    /**
     * This method indicates if the result received is the last chunk of data.
     *
     * @return true if this is the last chunk, false otherwise.
     */
    boolean isLastChunk();

    /**
     * Set whether this result object is the last chunk or not.
     */
    void setLastChunk(boolean isLastChunk);

    /**
     * Returns the status information about the population result.
     */
    UpdateStatus getStatus();

    /**
     * Set the population result status.
     */
    void setStatus(UpdateStatus status);

}
```

この出力オブジェクトには次が含まれます。

- **クエリされたデータ（ResultTreeNode 形式）**
- **ステータス情報（障害発生の場合に必要）**
- **チャンク情報**

サンプルのポピュレーション応答を次の図に示します。



上記の応答では，コネクタによって，4 つのデータ・インスタンス（UCMDB ノードに対応），空のステータス（成功のシグナリング），最後のチャンクではないことを示すフラグが返されます。

ポピュレーション・アダプタ・モード

UCMDB アダプタ・フレームワークでは，2 つの種類のポピュレーション・アダプタを使用できます。

- 標準ポピュレーション・アダプタ
  - アダプタ XML ファイルでの **<changes-source/>** タグの欠落が特徴
  - 常に外部システムから完全なクエリ・データを取得します。この場合，UCMDB Probe Framework が 2 つの連続した実行間の差異を判別します。Probe Framework は，所定のクエリの前回の結果を現在の結果と比較し，差異を計算することで，この差異をアーカイブします。完全なポピュレーションは，現在のクエリ結果を比較せずに，最終的な結果として扱うことでアーカイブされます。このフローは，ポピュレートされたデータが「開始日」によってフィルタリングされないことを示唆しています。これは，日付によるフィルタリングを行うと，データ比較が無意味になるためです。
- changes-source ポピュレーション・アダプタ
  - アダプタ XML ファイルで **<changes-source/>** タグを使用することで設定

```
<adapterInfo>
    <adapter-capabilities>
        <support-federated-query/>
        <support-replicatioin-data>
            <source>
                <changes-source/>
                <push-back-ids/>
                <re-populate/>
            </source>
            <target/>
        </support-replicatioin-data>
```

  - changes-source アダプタが 2 つの連続した実行間の差異を計算します。

changes-source アダプタを使用するときに CI を削除する

changes-source ポピュレーション・アダプタを使用する場合，アダプタによって CI が明示的に削除される必要があります。これは，有効な Groovy 表現を受け入れる **is-deleted** マッピング・ファイル XML 属性を使用することで実現できます。

たとえば，次に示すマッピング・ファイルでは，ポピュレーション・コネクタによって**サービス**・インスタンスが返されます。インスタンスはまだ有効ですが，これらの**サービス**・インスタンスの一部であるいくつかの CI が削除されています。これらの CI が削除されていることを伝えるには，**BusinessService** マッピング上で **is-deleted** 属性を使用する必要があります。

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="Service">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="Service.PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService" is-deleted="Functions.isOlderThanThreeMonths()">
            <target_mapping name="name" datatype="STRING" value="Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

明示的な外部 ID マッピング

ポピュレートしたデータ（CI）に，コネクタ / アダプタにより制御される **ExternalId** がなければならない場合があります。その場合は，次のマッピング構造を使用します。

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Node with ID" root-element-name="Computer">
        <!-- need to match case in UCMDB TQL -->
        <target_entity name="Root">
            <!--This is how the RTN External ID is set-->
            <variable name="external_id_obj" datatype="STRING" value="Computer['external_id_obj']"/>
            <!--RTN Attributes-->
            <target_mapping name="name" datatype="STRING" value="Computer.Asset[0]['name']"/>
            <target_mapping name="description" datatype="STRING" value="Computer['name']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

この場合，ルート CI は，コネクタ・レベルで作成され，Computer[‘external_id_obj’] に配置された ExternalId でポピュレートされます。ExternalId の作成は，Groovy スクリプトを使用してマッピング・レベルでも可能です。

注: 明示的に作成した外部 ID は，target_entity **type** 属性をオーバーライドします。したがって，マッピング・スクリプト・ファイルを使用して，またはコネクタの内部で外部 ID を作成する場合，**type** 属性が無視され，ポピュレート済みの CI の最終的な UCMDB タイプが **ExternalId** オブジェクトに設定されます。

グローバル ID プッシュバック

UCMDB にポピュレートした CI を外部システムとも同期の状態を保持しなければならない場合があります。このシナリオでは，汎用アダプタ・フレームワークでプッシュバック ID を有効化できます。この機能を使用するには，UCMDB にポピュレート済みのすべての CI に対してコールバックを実行し，ポピュレーション・アダプタに対して各 CI に割り当て済みのグローバル ID を伝えます。

この機能を有効にするには，次の例でマークが付けられている行をアダプタ構成 XML ファイルに追加します。

```
<adapterInfo>
    <adapter-capabilities>

        <support-federated-query>
        <supported-classes>
            <supported-class is-derived="true" all-attributes-supported="true" name="node" is-reconciliation-supported="true"/>
            <supported-class is-derived="true" all-attributes-supported="true" name="business_service" is-reconciliation-supported="true"/>
            <supported-class is-derived="true" all-attributes-supported="true" name="incident" is-reconciliation-supported="true"/>
        </supported-classes>
        </support-federated-query>
        <support-replicatioin-data>
            <source>
                <push-back-ids/>
                <instance-based-data/>
                <population-queries-resources-locator/>
            </source>
            <target>
                <instance-based-data>true</instance-based-data>
                <push-queries-resources-locator/>
            </target>
        </support-replicatioin-data>
        <general-resources-locator/>
    </adapter-capabilities>
```

次のように，PopulationAdapterConnector メソッドも実装する必要があります。

```
/**
* <b>Description:</b><br>
 * This Interface is used for implementing
 * an adapter to allow the definition of Integration Points.<br>
 * An adapter that implements this Interface will expose
 * ths ability to run Population flows.<br>
 * The only Population flow exposed by this adapter is the Simple Flow using {@link #getDataResult(com.hp.ucmdb.federationspi.adapter.federation.
 * which will return the entire data set of data each time, and compare it to the last data set (handled by framework).<p>
 * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}<br>
 * If possible, it's recommended to implement the {@link PopulationChangesAdapter} instead,
 * allowing the adapter to have better performance and control.<p>
 *
 * @see com.hp.ucmdb.federationspi.data.query.topology.TopologyFactory
 * @see com.hp.ucmdb.federationspi.adapter.ChunkTopologyResultGetter
 * @see PopulationAdapter
 * @since 10.10
 *
*/
public interface PopulationAdapter extends BasicSourceDataAdapter{

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the population framework.<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by the framework
     * @return {@link com.hp.ucmdb.federationspi.adapter.federation.FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

汎用アダプタを使用したデータ連携のアーカイブ

データ連携は次の方法でアーカイブされます。

連携マッピング・アプローチ

連携マッピングは，連携要求を処理するために UCMDB 連携フレームワークによって使用されるサブ TQL のマッピングにより，アーカイブされます。一般的に，汎用アダプタが連携要求を受信すると，次の内容が発生します。

1. 動的な連携 TQL クエリの分析，連携コネクタによって提供される静的な連携 TQL クエリのリストとその分析の比較。
2. 静的な TQL クエリの一致の作成。このクエリは，所定の連携要求に必要とされるマッピングを特定し，連携コネクタに供給される RTN 構造（コネクタから必要とされるツリー・ノード構造を示す Java オブジェクト）の入力引数を作成します(詳細については，**push-interfaces.jar** ファイルを参照してください）。
3. TQL 引数を含む連携呼び出しのコネクタへの送信。
4. ポピュレーションの場合と同じ，コネクタによって送信された入力 RTN ツリーのマップ。「汎用アダプタを使用したデータ・ポピュレーションのアーカイブ」(284ページ)を参照してください。

連携リンク・マッピング

連携リンク・マッピングは，ポピュレーション・リンク・マッピングの場合と同様に，自動的に実行されます。「自動リンク・ポピュレーション」(289ページ)を参照してください。

汎用アダプタ連携 API

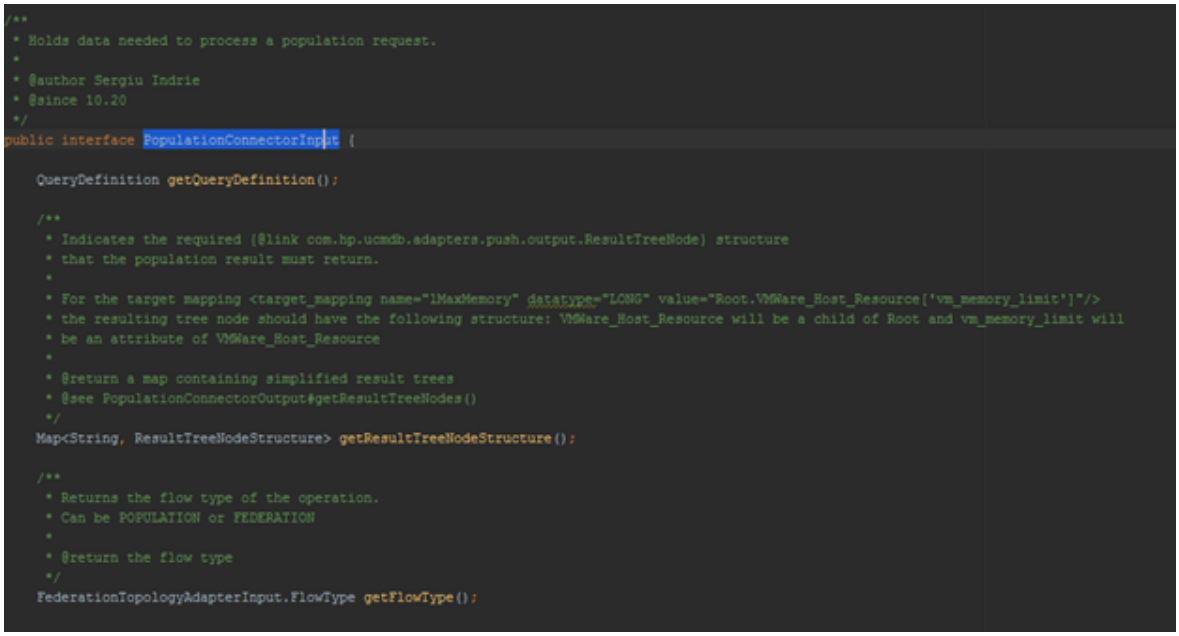
汎用アダプタ連携 API は 汎用アダプタ・ポピュレーション API とほぼ同じです。これは，汎用連携アダプタ Java インタフェースが汎用ポピュレーション・アダプタ・インタフェースと同一であるためです。

```
/**
* <b>Description:</b><br>
 * This Interface is used for implementing
 * an adapter to allow the definition of Integration Points.<br>
 * An adapter that implements this Interface will expose
 * ths ability to run Population flows.<br>
 * The only Population flow exposed by this adapter is the Simple Flow using {@link #getDataResult(com.hp.ucmdb.federationspi.adapter.federation.
 * which will return the entire data set of data each time, and compare it to the last data set (handled by framework).<p>
 * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}<br>
 * If possible, it's recommended to implement the {@link PopulationChangesAdapter} instead,
 * allowing the adapter to have better performance and control.<p>
 *
 * @see com.hp.ucmdb.federationspi.data.query.topology.TopologyFactory
 * @see com.hp.ucmdb.federationspi.adapter.ChunkTopologyResultGetter
 * @see PopulationAdapter
 * @since 10.10
 *
 */
public interface PopulationAdapter extends BasicSourceDataAdapter{

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the population framework.<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by the framework
     * @return {@link com.hp.ucmdb.federationspi.adapter.federation.FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

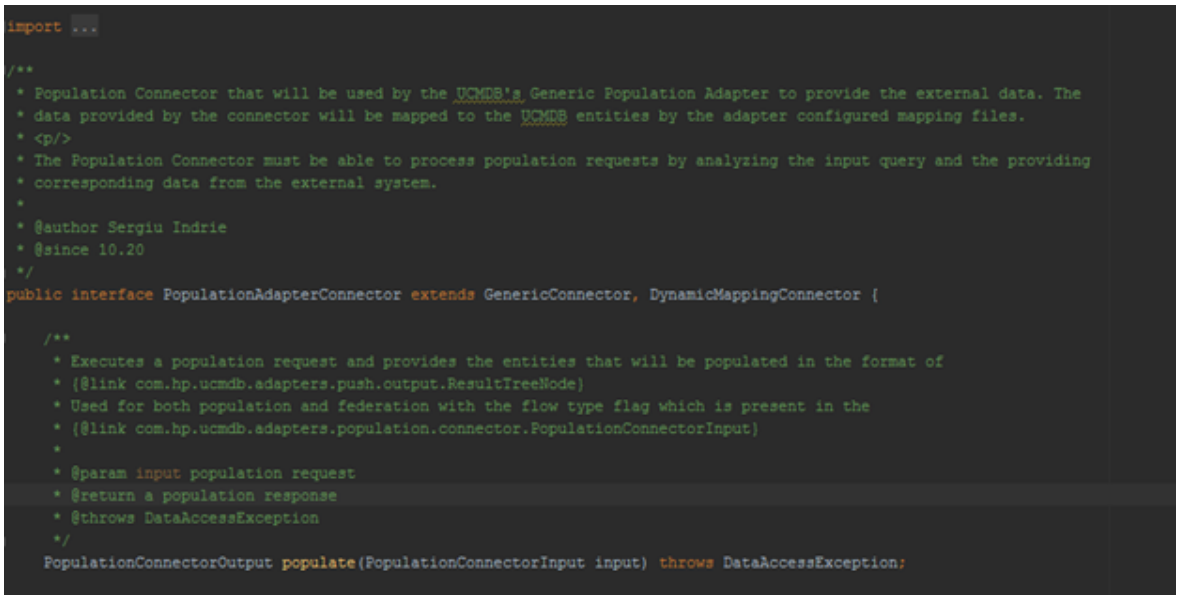
```
import ...

/**
  * <b>Description:</b><br>
  * This Interface is used for implementing
  * an adapter to allow the definition of Integration Points.<br>
  * An adapter that implements this Interface will expose to run Federation flows.<br>
  * The adapter can report status per CI during the flow with {@link com.hp.ucmdb.federationspi.data.replication.UpdateStatus}.<br>
  * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}
  * </p>
  *
 */
public interface FederationTopologyDataAdapter extends FTqlDataAdapter {

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the federation framework when the user query includes any
     * federated elements(node or link).<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by th
     * @return {@link FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

連携の汎用アダプタ・コネクタ・インタフェース

連携要求では，ポピュレーション要求で使用されるものと同じメソッドが使用されるため，同じポピュレーション・コネクタ実装を使用できます。PopulationConnectorInput Java クラス **FlowType** に新しい属性が追加されました。**FlowType** 属性は，FEDERATION または POPULATION の 2 つの値を持つことができます。汎用アダプタは，この属性に基づいて要求タイプを認識します。



```
/**
 * Holds data needed to process a population request.
 *
 * @author Sergiu Indrie
 * @since 10.20
 */
public interface PopulationConnectorInput {

    QueryDefinition getQueryDefinition();

    /**
     * Indicates the required {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode} structure
     * that the population result must return.
     *
     * For the target mapping <target_mapping name="lMaxMemory" datatype="LONG" value="Root.VMWare_Host_Resource['vm_memory_limit']"/>
     * the resulting tree node should have the following structure: VMWare_Host_Resource will be a child of Root and vm_memory_limit will
     * be an attribute of VMWare_Host_Resource
     *
     * @return a map containing simplified result trees
     * @see PopulationConnectorOutput#getResultTreeNodes()
     */
    Map<String, ResultTreeNodeStructure> getResultTreeNodeStructure();

    /**
     * Returns the flow type of the operation.
     * Can be POPULATION or FEDERATION
     *
     * @return the flow type
     */
    FederationTopologyAdapterInput.FlowType getFlowType();
```



```
import ...

/**
 * Population Connector that will be used by the UCMDB's Generic Population Adapter to provide the external data. The
 * data provided by the connector will be mapped to the UCMDB entities by the adapter configured mapping files.
 * <p/>
 * The Population Connector must be able to process population requests by analyzing the input query and the providing
 * corresponding data from the external system.
 *
 * @author Sergiu Indrie
 * @since 10.20
 */
public interface PopulationAdapterConnector extends GenericConnector, DynamicMappingConnector {

    /**
     * Executes a population request and provides the entities that will be populated in the format of
     * {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode}
     * Used for both population and federation with the flow type flag which is present in the
     * {@link com.hp.ucmdb.adapters.population.connector.PopulationConnectorInput}
     *
     * @param input population request
     * @return a population response
     * @throws DataAccessException
     */
    PopulationConnectorOutput populate(PopulationConnectorInput input) throws DataAccessException;
```

サポートされる連携クエリ

連携クエリおよびポピュレーション・クエリは異なるフォルダにあります。
PopulationAdapterConnector Java インタフェースには，サポートされるポピュレーション・クエリおよび連携クエリを示すための次の 2 つのメソッドがあります。

- **getPopulationQueries** – 現在のコネクタがポピュレーションについてサポートするクエリの集合を返します。
- **getFederationQueries** – 現在のコネクタが連携についてサポートするクエリの集合を返します。

```
/**
 * Returns the collection of queries the current connector supports for population
 * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
 * <p/>
 * The method also supports return of queries with their folders path: E.g.
 * "Folder/Secondary Folder/Query Name 1"
 * "Folder/Secondary Folder/Query Name 2"
 *
 * @param env the adapter environment
 * @return a collection of the supported queries
 */
Collection<String> getPopulationQueries(DataAdapterEnvironment env);

/**
 * Returns the collection of queries the current connector supports for federation
 * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
 * <p/>
 * The method also supports return of queries with their folders path: E.g.
 * "Folder/Secondary Folder/Query Name 1"
 * "Folder/Secondary Folder/Query Name 2"
 *
 * @param env the adapter environment
 * @return a collection of the supported queries
 */
Collection<String> getFederationQueries(DataAdapterEnvironment env);
```

連携の設定方法

本項の内容

アダプタ設定

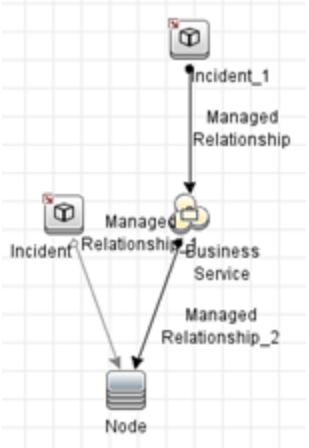
汎用アダプタは，所定の TQL クエリについて，そのすべてのノードを **<supported-classes>** タグに宣言する必要があります。たとえば，TQL クエリがノードにリンクされたインシデントの形式を持つ場合，それらのノードとインシデントの両方をサポートされるクラスとして **discoveryPatterns** フォルダのアダプタ・パッケージ ZIP ファイルにあるアダプタ設定 xml ファイルで宣言する必要があります。


```
<supported-classes>
    <supported-class is-derived="true" all-attributes-supported="true" name="node" is-reconciliation-supported="true"/>
    <supported-class is-derived="true" all-attributes-supported="true" name="incident" is-reconciliation-supported="true"/>
</supported-classes>
```

ログ・ファイルからの連携クエリの設定

連携フレームワークを使用する前に，いくつかの前提条件を満たす必要があります。連携フレームワークは，必要なデータを取得するために，異なる TQL クエリを含むいくつかの要求をアダプタに対して行います。連携フレームワークが送信する各 TQL クエリに対して，ポピュレーション・フローの場合と同様に，アダプタで 1 つの動的な TQL クエリが作成される必要があります。

連携とポピュレーションの違いは，連携 TQL クエリはフレームワークによって動的に送信されるため，事前に認識できない点です。TQL クエリの例を示します。



フレームワークはアダプタに次のクエリを送信します。

- インシデントのみを含むクエリ
- タイプ接続の関係を持つノードにリンクされたインシデントを含むクエリ
- タイプ・メンバーシップの関係を持つノードにリンクされたインシデントを含むクエリ
- タイプ接続の関係を持つビジネス・サービスにリンクされたインシデントを含むクエリ
- タイプ・メンバーシップの関係を持つビジネス・サービスにリンクされたインシデントを含むクエリ

注: 連携エンジンがアダプタに送信し，保存が必要なすべてのクエリには，**User mapping union FTQL** という名前が付きます。

User mapping union FTQL クエリの結果が処理された後に，ほかの呼び出しが実行され，オブジェクトの属性が取得されます。これらの呼び出しには，**objects layout** と呼ばれるクエリが含まれます。連携エンジンは CI のすべての属性を取得しようとしますが，コネクタはそれらのすべてを提供する必要はありません。マッピング・ファイルによって要求される属性のみを返すだけで十分です。

異なる関係を持つ同一のクエリを送信する理由は，TQL クエリに，ノード，インシデント / ビジネス・サービス，インシデント間に **managed_relationship** タイプ・リンクが存在するためですが，これらの CI タイプをリンクしようとするときの有効なリンクは接続とメンバーシップのみです。

```
<tql:link from="incident_12" to="node_10050" class="connection" name="connection_1" id="1"/>
<tql:link from="incident_12" to="node_1000050" class="connection" name="connection_2" id="2"/>
<tql:link from="datacenter_20050" to="node_10050" class="composition" name="composition_30050" id="30050"/>
```

```
<tql:link from="incident_12" to="node_10050" class="membership" name="membership_1" id="1"/>
<tql:link from="incident_12" to="node_1000050" class="membership" name="membership_2" id="2"/>
<tql:link from="datacenter_20050" to="node_10050" class="composition" name="composition_30050" id="30050"/>
```

この方法では，異なるリンク・タイプを持つほぼ同一の 2 つの TQL を定義する代わりに，汎用 **managed_relationship** タイプ・リンクを含む静的 TQL を 1 つだけ定義する必要があります。

```
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >> Received federation call with the following query:
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >>
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tql:query name="User mapping union FTQL" is-live="true" priority="low" xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyRuleD
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="membership_1"/>
                    <tql:link-ref name="membership_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>MyBusServ</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201334714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:link from="incident_16" to="business_service_10050" class="membership" name="membership_1" id="1"/>
    <tql:link from="incident_16" to="business_service_1000050" class="membership" name="membership_2" id="2"/>
</tql:query>
```

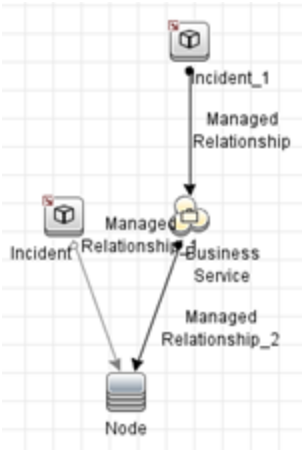
この静的 TQL クエリはアダプタによって提供される必要があります。アダプタ開発を支援するため，連携フレームワークによって送信される TQL クエリは **fcmdb.push.mapping.log** ファイル（TRACE ログ・レベルが有効化された）に書き込まれます。<adapter-setting name="dev.mode">true</adapter-setting> を使用すると，開発にかかる労力を軽減できます。連携クエリを実行した後にこの設定を True に設定すると，フレームワークによって，現在不一致の TQL クエリに対して空の連携結果が自動的に作成されます。

fcmdb.push.mapping.log からの連携クエリの例 :

```
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >> Received federation call with the following query:
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >>
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tql:query name="User mapping union FTQL" is-live="true" priority="low" xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyRuleD
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="membership_1"/>
                    <tql:link-ref name="membership_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>MyBusServ</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201334714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:link from="incident_16" to="business_service_10050" class="membership" name="membership_1" id="1"/>
    <tql:link from="incident_16" to="business_service_1000050" class="membership" name="membership_2" id="2"/>
</tql:query>
```

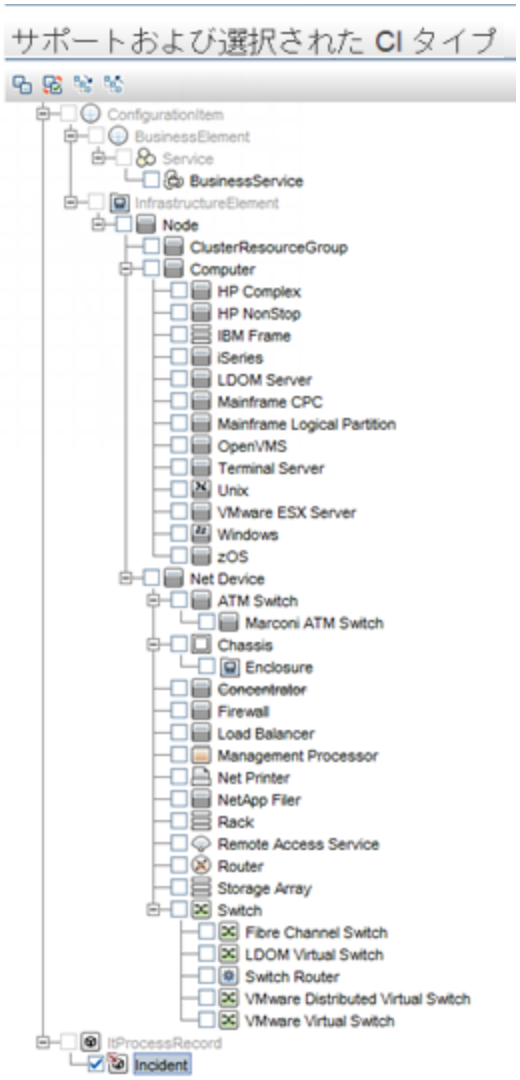
連携設定の例

この例では，次の連携 TQL が使用されています。



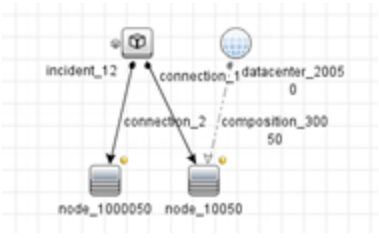
この TQL クエリの場合，アダプタは，**discoveryPatterns** フォルダのアダプタ・パッケージ ZIP ファイル内にあるアダプタ設定 XML ファイルで，サポートされるクラスを宣言する必要があります。サポートされるクラスは，**node**，**incident** および **business_service** です。

Integration Studio では，インシデントは，次に示すように，［**連携**］タブで選択する必要があります。

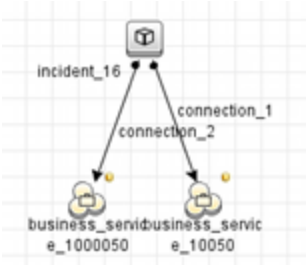


この TQL クエリでは，アダプタに次の 3 つの TQL クエリがあります。

インシデント



ノードにリンクされたインシデント



ビジネス・サービスにリンクされたインシデント

静的 TQL の取得方法の詳細については，「ログ・ファイルからの連携クエリの設定」(304ページ)を参照してください。これらの TQL クエリは UCMDB に存在するデータに依存する条件を持ちますが，TQL クエリの構造またはマッピングの実行方法に対して影響しません。

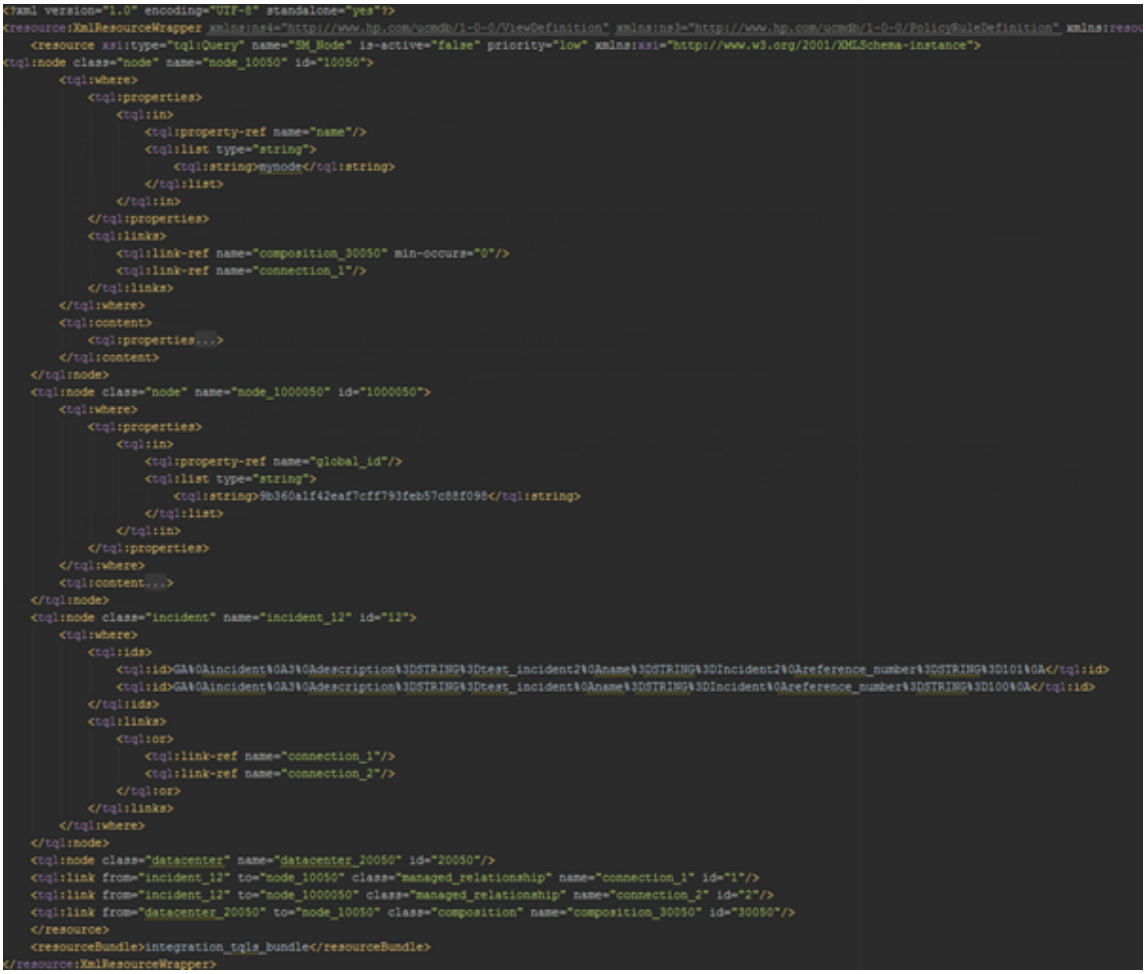
これらの TQL クエリのそれぞれについて，アダプタ内にマッピング・ファイルが存在している必要があります。

- インシデント

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resource:XmlResourceWrapper xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyRuleDefiniti
    <resource xsi:type="tql:Query" name="SM_Incident" is-active="false" priority="low" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<tql:node class="incident" name="incident_12" id="12"/>
    </resource>
    <resourceBundle>integration_tqls_bundle</resourceBundle>
</resource:XmlResourceWrapper>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>

    <!--
    This mapping converts the Root entities received from the population connector to a "node" item in UCMDB.
    The output of this mapping must match the indicated query (TQL), "Dummy Query"
    -->

    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_Incident" root-element-name="incident_12">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="incident_12">
                <target_mapping name="name" datatype="STRING" value="incident_12['name']"/>
                <target_mapping name="description" datatype="STRING" value="incident_12['description']"/>
                <target_mapping name="reference_number" datatype="STRING" value="incident_12['reference_number']"/>
            </target_entity>
        </source_instance>
    </target_entities>

</integration>
```

- ノードにリンクされたインシデント



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resource:XmlResourceWrapper xmlns:ns="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns1="http://www.hp.com/ucmdb/1-0-0/PolicyRuleDefinition" xmlns:resou
    <resource xsi:type="tql:Query" name="SM_Node" is-active="false" priority="low" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<tql:node class="node" name="node_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>mynode</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
            <tql:links>
                <tql:link-ref name="composition_30050" min-occurs="0"/>
                <tql:link-ref name="connection_1"/>
            </tql:links>
        </tql:where>
        <tql:content>
            <tql:properties...>
        </tql:content>
    </tql:node>
    <tql:node class="node" name="node_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>9b360a1f42eaf7cff793feb57c88f098</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content...>
    </tql:node>
    <tql:node class="incident" name="incident_12" id="12">
        <tql:where>
            <tql:ids>
                <tql:id>GA%0Aincident%0A3%0Adescription%3DSTRING%3Dtest_incident2%0Aname%3DSTRING%3DIncident2%0Areference_number%3DSTRING%3D101%0A</tql:id>
                <tql:id>GA%0Aincident%0A3%0Adescription%3DSTRING%3Dtest_incident%0Aname%3DSTRING%3DIncident%0Areference_number%3DSTRING%3D100%0A</tql:id>
            </tql:ids>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="connection_1"/>
                    <tql:link-ref name="connection_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="datacenter" name="datacenter_20050" id="20050"/>
    <tql:link from="incident_12" to="node_10050" class="managed_relationship" name="connection_1" id="1"/>
    <tql:link from="incident_12" to="node_1000050" class="managed_relationship" name="connection_2" id="2"/>
    <tql:link from="datacenter_20050" to="node_10050" class="composition" name="composition_30050" id="30050"/>
    </resource>
    <resourceBundle>integration_tqls_bundle</resourceBundle>
</resource:XmlResourceWrapper>
```

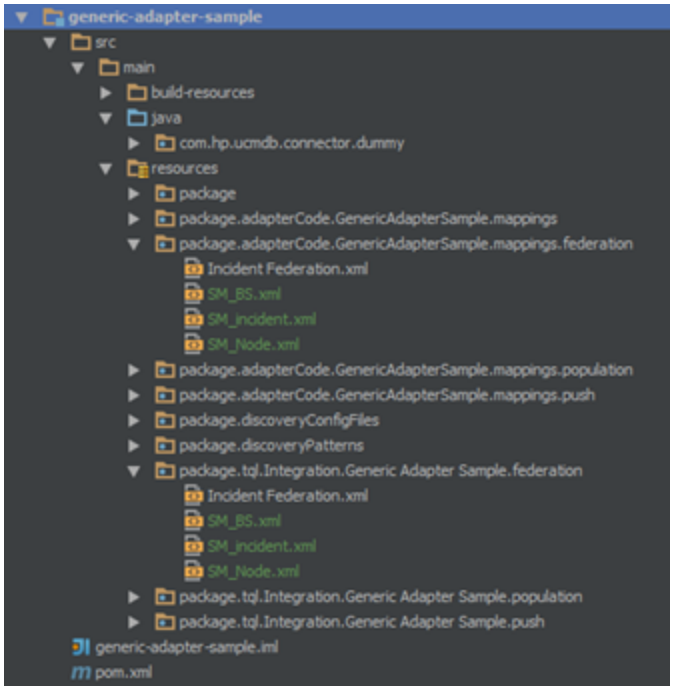
```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>
    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_Node" root-element-name="Computer">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="node_1000050">
                <target_mapping name="name" datatype="STRING" value="Computer['name']"/>
            </target_entity>


            <for-each-source-entity count-index="i" source-entities="Computer.incident_12" var-name="currIP">
                <target_entity name="incident_12">
                    <target_mapping name="name" datatype="STRING" value="currIP['name']"/>
                    <target_mapping name="description" datatype="STRING" value="currIP['description']"/>
                    <target_mapping name="reference_number" datatype="STRING" value="currIP['reference_number']"/>
                </target_entity>
            </for-each-source-entity>


        </source_instance>
    </target_entities>
</integration>
```

- ビジネス・サービスへのインシデント

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resource:XmlResourceWrapper xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyBu
    <resource xsi:type="tql:Query" name="SM_BS" is-active="false" priority="low" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="connection_1"/>
                    <tql:link-ref name="connection_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>MyBusServ</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201334714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
        <tql:link from="incident_16" to="business_service_10050" class="managed_relationship" name="connection_1" id="1"/>
        <tql:link from="incident_16" to="business_service_1000050" class="managed_relationship" name="connection_2" id="2"/>
    </resource>
    <resourceBundle>integration_tqls_bundle</resourceBundle>
</resource:XmlResourceWrapper>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>
    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_BS" root-element-name="BS">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="business_service_10050">
                <target_mapping name="name" datatype="STRING" value="BS['name']"/>
            </target_entity>
            <for-each-source-entity count-index="i" source-entities="BS.incident_16" var-name="currIP">
                <target_entity name="incident_16">
                    <target_mapping name="name" datatype="STRING" value="currIP['name']"/>
                    <target_mapping name="description" datatype="STRING" value="currIP['description']"/>
                    <target_mapping name="reference_number" datatype="STRING" value="currIP['reference_number']"/>
                </target_entity>
            </for-each-source-entity>
        </source_instance>
    </target_entities>
</integration>
```

これらの TQL クエリおよびマッピング・ファイルは，次に示すように，アダプタ内に存在する必要があります。



調整

汎用アダプタ・フレームワークを使用してデータをポピュレートまたは連携する場合，その CI が UCMDB に受け入れられるには，その CI が必要な調整データを常に持つ必要があります。実行中のソフトウェアなど，コンテナ CI タイプを必要とする CI タイプをポピュレートする場合，必要なコンテナ・フィールド（例 : **root_container_name** および **product_name**）およびコンテナ CI（例 : **ノード**）をポピュレートする必要があります。ルート・コンテナに依存する CI をポピュレートするには，CI，そのルート・コンテナ，それらの間のリンクを同一のステップ（2 つの CI 間の明示的なリンク・ポピュレーションまたはオートコンプリート・リンク・ポピュレーションのいずれか）で作成する必要があります。

さらに，ポピュレート済み / 連携済みの CI をマップする場合は，**global_id** 属性をマップすることを考慮してください。そうすることにより，UCMDB 構成エンジンを大きく支援でき，正確な CI 調整を実現することができます。

汎用アダプタ API

汎用アダプタ・フレームワークによって公開される API を次に示します。

<UCMDB_Server>\lib\push-interfaces.jar
<UCMDB_Server>\lib\integrationFramework\GenericAdapter\generic-adapter-api-factory.jar

汎用アダプタ・インスタンスの開発では，連携 API も必要な場合があります。

<UCMDB_Server>\lib\federation-api.jar

リソース・ロケータ API

リソース・ロケータ API は汎用アダプタ・ジョブを編集するときに使用できます。ポピュレーションおよび一般的なリソース・ロケータ API を実装することで，選択したジョブの TQL クエリに関連するアダプタ・リソースの検索に役立てることができます。

次の画像に，GenericConnector Java インタフェースの一般的なリソース・ロケータ API を示します。

```
/**
 * This methods reports general resources used by the adapter.<br>
 * This allows editing these resources directly from the Integration Studio.
 *
 *
 * @param env the Adapter's environment
 * @param input  the requested information
 * @param output the returned resources
 * @see com.hp.ucmdb.federationspi.adapter.resource.GeneralResourcesLocator
 */
void locateGeneralResources(DataAdapterEnvironment env, LocateGeneralResourcesInput input, LocateGeneralResourcesOutput output);
```

次の画像に，PopulationAdapterConnector Java インタフェースのポピュレーション・クエリ・リソース・ロケータ API を示します。

```
/**
 * This methods reports population queries resources used by the adapter.<br>
 * This allows editing these resources directly from the Integration Studio.
 *
 * @param input  the requested information
 * @param output the returned resources
 * @see com.hp.ucmdb.federationspi.adapter.resource.PushQueriesResourceLocator
 */
void locatePopulationQueriesResources(DataAdapterEnvironment env, LocatePopulationQueriesResourcesInput input,
                                      LocatePopulationQueriesResourcesOutput output);
```

ジョブの TQL クエリに関連するリソースを表示するには：

1. Integration Studio で統合ポイントを選択します。
2. ［統合ジョブ］表示枠で，ジョブを選択し，［**統合クエリの編集**］  をクリックします。

汎用アダプタ・パッケージの作成

汎用アダプタ・パッケージは，拡張汎用プッシュ・アダプタ・パッケージに似ています。その初期スケルトン ZIP アーカイブを作成する場合は，既存の汎用アダプタ・パッケージをコピーし，必要に応じてカスタマイズすることをお勧めします。アダプタ・パッケージの詳細については，「汎用アダプタを使用したデータ・プッシュのアーカイブ」(275ページ)を参照してください。

既存の拡張汎用プッシュ・アダプタ・パッケージと汎用アダプタ・パッケージの違いを次に示します。

- アダプタ XML の違い
  - アダプタ・クラスが **PushAdapter** から **GenericAdapter** に変更される

```
<className>com.hp.ucmdb.adapters.push.PushAdapter</className>
```

```
<className>com.hp.ucmdb.adapters.GenericAdapter</className>
```

  - アダプタ機能にポピュレーションが含まれる

```
<support-replicatioin-data>
    <source>
        <push-back-ids/>
        <instance-based-data/>
        <population-queries-resources-locator/>
    </source>
```

  - ポピュレーション・コネクタの定義と同時に，次がアダプタ設定により実行される

```
<adapter-setting name="PopulationConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPopulationConnector</adapter-setting>
```

    汎用アダプタ（ポピュレーション機能を使用）はプッシュ・コネクタ・クラスの定義も必要とする

```
<adapter-setting name="PushConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPushConnector</adapter-setting>
```

- マッピング・ファイル・フォルダ

  拡張汎用プッシュ・アダプタ（そのマッピング・ファイルが **<adapter_package_zip>/adapterCode/<*アダプタ名*>/mappings** フォルダに存在しなければならない）とは異なり，汎用アダプタでは，そのマッピングが 3 つの個別フォルダ（プッシュ，ポピュレーション，連携）に配置される必要があります。これらの必須フォルダは次のとおりです。

  <adapter_package_zip>/adapterCode/<*アダプタ名*>/mappings/push
  <adapter_package_zip>/adapterCode/<*アダプタ名*>/mappings/population
  <adapter_package_zip>/adapterCode/<*アダプタ名*>/mappings/federation

  ここで，**<adapter_package_zip>** は，汎用アダプタ・パッケージに対して作成する zip アーカイブを指します。

注: 汎用アダプタは 3 つのすべてのデータ同期タイプ（プッシュ，ポピュレーション，連携）をサポートしますが，特定の汎用アダプタではそれらのタイプのサブセットのみを提供するように選択できます。

既存のアダプタから新規アダプタを作成する際の注意

- **TestAdapter\discoveryPatterns\TestAdapter.xml**
- **TestAdapter.xml** ファイルを変更します。
  - <pattern xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="TestAdapter" xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="…" schemaVersion="9.0"

displayName="…">

- <adapter-id>TestAdapter</adapter-id>

- 新規アダプタを含む ZIP ファイルは，アダプタ自体の名前（TestAdapter）と同じ名前を持つ必要があります。

アダプタ・パッケージの作成

アダプタ・パッケージに，次のフォルダが含まれていることを確認します。

- **adapterCode:** このフォルダに **PushExampleAdapter** という名前のフォルダを作成します。このフォルダには PushExampleAdapter.java から作成した .jar ファイルが含まれます。また **mappings** という名前のフォルダも含まれます。このフォルダには以前作成したマッピング・ファイル **computerIPMapping.xml** を配置できます。このフォルダには，**PushFunctions.groovy** ファイルが含まれる **scripts** という名前の別のフォルダも含まれます。
- **discoveryConfigFiles :**UpdateResult を使用してエラーを報告する際に使用するエラー・コードなどの構成ファイルが含まれます。この例では，フォルダは空になっています。
- **discoveryPatterns :push_example_adapter.xml** が含まれます。
- **tql :**サンプルとして作成された TQL クエリが含まれます。このフォルダはオプションですが，パッケージを実装する場合，TQL クエリは自動的に作成されます。

アダプタ・レベルでの属性およびリンクの検証の有効化 / 無効化

次の設定を追加することで，汎用アダプタのアダプタ・レベルでの属性およびリンクの検証を有効化または無効化できます。

```
<adapter-settings>
    <adapter-setting name="enable.attributes.links.validation">true</adapter-setting>
</adapter-settings>
```

属性およびリンクのアダプタ・レベルでの検証を有効にするには，アダプタ設定 **enable.attributes.links.validation** を **true** に設定します。

属性およびリンクのアダプタ・レベルでの検証を無効にするには，アダプタ設定 **enable.attributes.links.validation** を **false** に設定します。

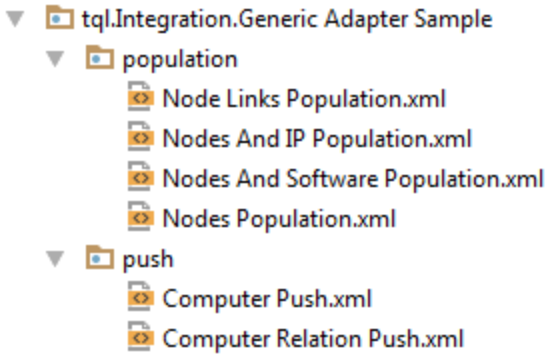
注 : この設定が存在しない場合，その標準設定値が **true** に設定されます。これは，標準設定により，属性とリンクの検証が有効化されることを意味します。

ポピュレーション TQL クエリ

ポピュレーション・ジョブに使用する TQL クエリは，汎用アダプタの ZIP アーカイブに含め，アダプタとともに UCMDB にデプロイする必要があります。ポピュレーション要求の実行時には，記述され

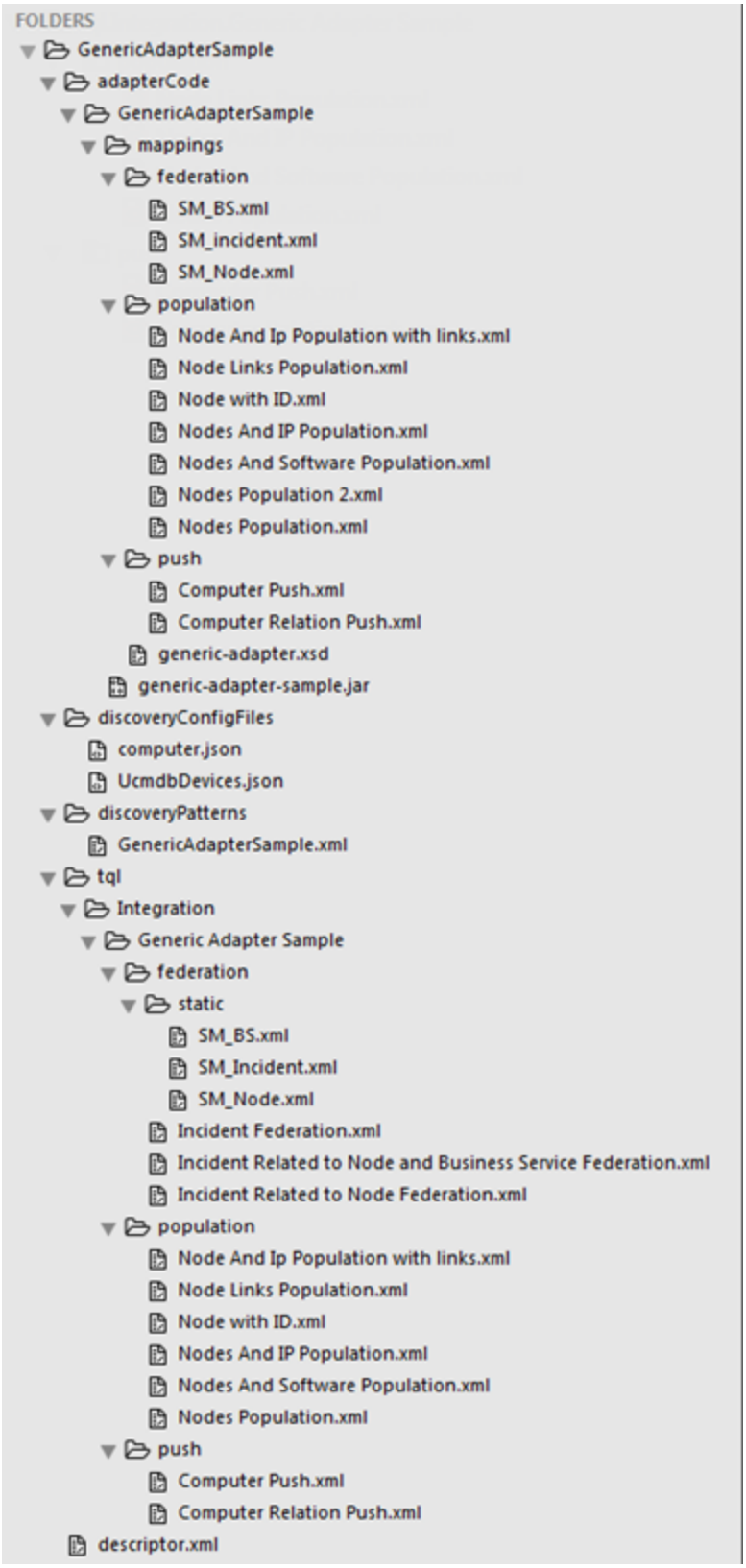
たTQLクエリがポピュレーション・フロー中にUCMDBに存在しなければなりません。

これらのTQLクエリは，**<zip>/tql/<folder_1>/../<folder_n>** に含める必要があります。このフォルダ構造の例を次に示します。



ポピュレーションTQLクエリは上記に示すフォルダに配置されますが，ポピュレーション・コネクタにより，サポートされるポピュレーションTQLクエリがJavaインタフェースの対応するメソッドで確認される必要もあります。詳細については，「ポピュレーション・コネクタ」(291ページ)を参照してください。

サンプル・パッケージ


FOLDERS
GenericAdapterSample
adapterCode
GenericAdapterSample
mappings
federation
SM_BS.xml
SM_incident.xml
SM_Node.xml
population
Node And Ip Population with links.xml
Node Links Population.xml
Node with ID.xml
Nodes And IP Population.xml
Nodes And Software Population.xml
Nodes Population 2.xml
Nodes Population.xml
push
Computer Push.xml
Computer Relation Push.xml
generic-adapter.xsd
generic-adapter-sample.jar
discoveryConfigFiles
computer.json
UcmdbDevices.json
discoveryPatterns
GenericAdapterSample.xml
tql
Integration
Generic Adapter Sample
federation
static
SM_BS.xml
SM_Incident.xml
SM_Node.xml
Incident Federation.xml
Incident Related to Node and Business Service Federation.xml
Incident Related to Node Federation.xml
population
Node And Ip Population with links.xml
Node Links Population.xml
Node with ID.xml
Nodes And IP Population.xml
Nodes And Software Population.xml
Nodes Population.xml
push
Computer Push.xml
Computer Relation Push.xml
descriptor.xml

プッシュ・マッピングとポピュレーション・マッピングの違い

プッシュ・マッピングとポピュレーション・マッピングのファイルは，両方とも同じ基盤 XML スキーマを使用しますが，ファイル自体は多少違った意味を持ちます。詳細については，「汎用アダプタ XML スキーマ・リファレンス」(321ページ)を参照してください。

以下のプッシュ・マッピングの例は次のように解釈できます。「Computer Push」 TQL クエリ（UCMDB で実行）の結果を取得し，ルート・ツリー構造に提示し，後で AM に送信される amComputer を作成する。

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Computer Push" root-element-name="Root">
        <target_entity name="amComputer">
            <target_mapping name="TcpIpHostName" datatype="STRING" value="Root['name']"/>
            <target_mapping name="ComputerDesc" datatype="STRING" value="Root['os_description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

以下のポピュレーション・マッピングの例は次のように解釈できます。「Nodes Population」 TQL クエリ（外部システムで実行）の結果を取得し，PC ツリー構造に提示し，後で UCMDB に追加される UCMDB ルート・エンティティ（TQL クエリによって示されるタイプ「ノード」）を作成する。

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Nodes Population" root-element-name="PC">
        <!-- need to match case in UCMDB class model-->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

汎用アダプタ・ログ・ファイル

トラブルシューティングおよびデバッグを行う場合は，次のいずれかを使用します。

- これらのファイルでログ記録のレベルを調整します（最も詳細な結果を得るには，*loglevel* 変数を TRACE に設定します）。
  - **<UCMDB_DataFlowProbe>\conf\log\fcmdb.push.properties**

    <UCMDB_DataFlowProbe> は，UCMDB Data Flow Probe のインストール・ディレクトリです。

  - **<UCMDB_Server>\conf\log\reconciliation.properties**

    <UCMDB_Server> は，UCMDB サーバのインストール・ディレクトリです。

- 次の汎用アダプタ・ログ・ファイルを分析します。
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.all.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.configuration.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.connector.all.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.connector.configuration.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.mapping.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.all.log**
- 次の汎用ログ・ファイルを分析します。
  - **<UCMDB_DataFlowProbe>\runtime\log\probe-error.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\WrapperProbeGw.log**
  - **<UCMDB_Server>\runtime\log\error.log**
  - **<UCMDB_Server>\runtime\log\cmdb.reconciliation.log**

汎用アダプタ・フレームを使用するアダプタ

独自のカスタム汎用アダプタを開発するときの参考として，実装ガイドラインとして UCMDB に付属する次のアダプタを参照してください。アダプタ開発の高速化を図ることができます。

- アセット・マネージャ・アダプタ
- サービス・マネージャ・アダプタ

汎用アダプタ XML スキーマ・リファレンス

汎用アダプタ XML スキーマは，**schema** ディレクトリの **cmdb.jar** ファイル内にあります。スキーマ・ファイルは，汎用アダプタ・マッピング・ファイルを外部エディタで作成するときに参照する必要があります。XSD ファイルの完全パスを次に示します。

<UCMDB_Server_Install_dir>/lib/cmdb.jar/schema/generic-adapter.xsd

第II部: API の使用

第9章: API について

本章の内容

API の概要

HP Universal CMDB には次の API が含まれています。

- **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，「HP Universal CMDB API」(324ページ)を参照してください。
- **UCMDB Web サービス API :** UCMDB 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，「HP Universal CMDB Web サービス API」(332ページ)を参照してください。
- **データ・フロー管理 Java API :** データ・フロー管理のためのプローブ，ジョブ，トリガ，資格情報を管理できます。詳細については，「データ・フロー管理 Java API」(367ページ)を参照してください。
- **データ・フロー管理 Web サービス API :** データ・フロー管理のためのプローブ，ジョブ，トリガ，資格情報を管理できます。詳細については，「データ・フロー管理 Web サービス API」(369ページ)を参照してください。

注: API ドキュメントのすべての値を参照するには，オンライン・ドキュメントにアクセスすることをお勧めします。PDF バージョンには，html 形式で生成された API ドキュメントへのリンクはありません。

第10章: HP Universal CMDB API

本章の内容

表記規則

本章では，次の表記を使用します。

- 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指します。
- 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。

  利用可能な API の完全なドキュメントについては，HP UCMDB API Reference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。

  \\<UCMDB のルート・ディレクトリ>\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\UCMDB_JavaAPI\index.html

HP Universal CMDB API の使用

注: 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（UCMDB）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，更新
- クラス・モデルに関する情報の取得
- UCMDB 履歴からの情報の取得
- 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，『HP Universal CMDB モデリング・ガイド』の「Topology Query Language」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」(325ページ)
- 「権限」(325ページ)

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。詳しいユース・ケースについては，「UCMDB API のユーズ・ケース」(330ページ)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 で定義されている統合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。

さらに，ユーザがログインするには，［**SDK へアクセス**］一般アクション権限がなければなりません。

> **注意:** API クライアントも，API 認証権限が付与されていれば，標準のユーザで操作できます。ただし，この方法は推奨されません。

詳細については，「統合ユーザの作成」(328ページ)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を実行します。

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 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
getAuthorizationModelService	認証操作を実行します（ユーザやユーザ・グループを作成し，ロールをユーザやグループに割り当てる，など）。
getClassModelService	CI と関係のタイプに関する情報
getConfigurationService	サーバ設定のためのインフラストラクチャ設定管理
getDataStoreMgmtService	どの CI や属性がフェデレートされるかを含むデータ・ストア情報のクエリ。
getDDMConfigurationService	データ・フロー管理システムの設定
getDDMManagementService	データ・フロー管理システムの進行状況，結果，エラーを分析して表示
getDDMZoneService	管理ゾーン（ゾーンのアクティビティを含む）をインポートおよびエクスポート。
getHistoryService	監視対象の CI の履歴に関する情報（変更，削除など）
getImpactAnalysisService	影響分析シナリオ（**相関**とも呼ばれます）を実行
getLicensingService	システムにインストールされているライセンスに関する情報のクエリ。
getMultipleCMDBService	グローバル ID と UCMDB ID の変換。
getMultiTenancyService	テナントの作成，読み取り，更新，削除。
getPersistencyService	バイナリ・データをキーと値のペアで保持。
getQueryManagementService	クエリへのアクセスの管理（既存アイテムの保存，削

サービス・メソッド	用途
	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。
getReconciliationService	識別機能と結合機能を提供します。
getResourceBundleManagementService	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。
getResourceManagementService	システムに（TQL クエリ，ビュー，ユーザなどの）リソース・パッケージをデプロイします。
getSecurityService	資格情報が有効化どうかを確認します。
getServerService	システムに関する汎用情報のクエリ。
getSnapshotService	スナップショットを管理するためのサービスの提供（取得，保存など）
getSoftwareSignatureService	データ・フロー管理システムで検出するソフトウェア・アイテムの定義
getStateService	状態を管理するためのサービスの提供（一覧表示，追加，削除など）
getSystemHealthService	システム状況サービスを提供（基本的なシステム・パフォーマンス・インジケータ，容量と可用性の測定値）
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
getUcmdbVersion	UCMDB，コンテンツ・パック・バージョン，ビルド情報のクエリ。
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。
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。

クライアントは HTTP(S) を介してサーバと通信します。

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。ファイルをダウンロードするには，Web ブラウザに http://<localhost>:8080 （localhost は UCMDB がインストールされているマシン）と入力し，［**API Client Download**］リンクをクリックします。

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。

注: UCMDB Java API Jar を使用するには，JRE バージョン 6 以降がインストールされている必要があります。

統合ユーザの作成

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，統合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: 通常の UCMDB ユーザで接続することもできます（例 : admin）。ただし，この方法は推奨されません。UCMDB ユーザで接続するには，そのユーザに API 認証権限を付与する必要があります。

統合ユーザを作成するには，次の手順を実行します。

1. Web ブラウザを起動し，サーバ・アドレスとして次を入力します。

   http://localhost:8080/jmx-console

   ユーザ名とパスワードを使用してログインする必要がある場合もあります。
2. UCMDB の下の **service=UCMDB Authorization Services** をクリックします。
3. **createUser** 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** :顧客 ID です。
   - **ユーザ名** : 統合ユーザの名前です。
   - **userDisplayName** :統合ユーザの表示名です。
   - **userLoginName** :統合ユーザのログイン名です。
   - **password** :統合ユーザのパスワードです。

     標準設定のパスワード・ポリシーでは，次の 4 種類の文字をそれぞれ 1 文字以上パスワードに含める必要があります。

- 大文字のアルファベット文字
- 小文字のアルファベット文字
- 数字
- 記号文字 ,\:/. _?&%=+-[]()|

また，［**パスワードの最小長**］で設定されたパスワードの最短長に従う必要もあります。

4. ［**Invoke**］をクリックします。
5. シングルテナント環境で，**setRolesForUser** メソッドを見つけて，次のパラメータを入力します。
   - **username**:統合ユーザの名前です。
   - **roles:** SuperAdmin。

   ［**Invoke**］をクリックします。
6. マルチテナント環境では，**grantRolesToUserForAllTenants** メソッドを検索し，次のパラメータを入力して，すべてのテナントに関連するロールを割り当てます。
   - **username**:統合ユーザの名前です。
   - **roles :** SuperAdmin。

   ［**Invoke**］をクリックします。

   特定のテナントに関連するロールを割り当てるには，代わりに同じ userName とロール・パラメータ値を使用して **grantRolesToUserForTenants** メソッドを呼び出します。**tenantNames** パラメータについては，必須テナントを入力します。
7. さらにユーザを作成するか，JMX コンソールを閉じます。
8. 管理者として UCMDB にログオンします。
9. ［**管理**］タブで，［**パッケージ マネージャ**］を実行します。
10. ［**カスタム パッケージを作成しています**］アイコンをクリックします。
11. 新しいパッケージ名を入力して，［**次へ**］をクリックします。
12. ［リソースの選択］タブで，［**管理**］の下の［**ユーザ**］をクリックします。
13. 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。
14. ［**次へ**］をクリックし，次に［**完了**］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。
15. 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 Universal CMDB 管理ガイド』の「パッケージのデプロイ方法」の項を参照してください。

注: 統合ユーザは顧客ごとに作成します。複数の顧客で使用できる，より強い権限の統合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（**removeUser**，**resetPassword**，

UserAuthenticate など）を使用します。

次の 2 つのシステム・ユーザが定義済みです。インストール後，**resetPassword** 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 **SuperIntegrationUser** でもあります）。

  resetPassword を使って UISysadmin パスワードを変更した場合，次の操作を実行する必要があります。

  i. まず，JMX コンソールで **UCMDB-UI:name=UCMDB Integration** サービスを選択します。

  ii. その後，統合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。

UCMDB API のユース・ケース

このセクションに示されたユース・ケースは次の 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

ユース・ケース:

- 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，UCMDB クエリ機能を活用します。

CMDB へのクエリ

ユース・ケース:

- 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・モデルへのクエリ

ユース・ケース:

- 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

ユース・ケース:

- 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

次のコード例を参照してください。

- Create a Connection
- Create and Execute an Ad Hoc Query
- Create and Execute a View
- Add and Delete Data
- Execute an Impact Analysis
- Query the Class Model
- Query a History Sample

ファイルは次のディレクトリにあります。

\\<UCMDB のルート・ディレクトリ>\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\JavaSDK_Samples

第11章: HP Universal CMDB Web サービス API

本章の内容

表記規則

本章では，次の表記を使用します。

- 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指します。
- 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。

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 Reference 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。

<UCMDB のルート・ディレクトリ>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\CMDB_Schema\webframe.html

HP Universal CMDB Web Service API の概要

注: 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 Service 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，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，『HP Universal CMDB 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++，C#，Java など）
- HP Universal CMDB
- データ・フロー管理

本項の内容

- 「API の使用」(333ページ)
- 「権限」(334ページ)

API の使用

UCMDB Web Services 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。
- Release Control（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

web service の WSDL ファイルにアクセスするには，以下に移動します。
http://localhost:8080/axis2/services/UcmdbService?wsdlWSDL ファイルを表示するには，サーバ管理者ユーザ資格情報を提供する必要があります。

注: Axis2 管理コンソールにはアクセスできません。

ユーザがログインするためには **Run Legacy API** 一般アクション権限がなければなりません。

次のテーブルは，各 Web サービス API コマンドに必要な追加の権限を示したものです。

Web サービス API コマンド	**必要な権限**
addCIsAndRelations deleteCIsAndRelations updateCIsAndRelations	一般アクション:**データを更新**
executeTopologyQueryByName(AdHoc) executeTopologyQueryByNameWithParameters(AdHoc) executeTopologyQueryWithParameters(AdHoc)	一般アクション:**定義に従ってクエリを実行** クエリの場合:**ビュー**権限
getTopologyQueryExistingResultByName getTopologyQueryResultCountByName releaseChunks pullTopologyMapChunks getCINeighbours getFilteredCIsByType getCIsById getCIsByType getRelationsById	一般アクション:**CI を表示** クエリの場合:**ビュー**権限
getQueryNameOfView	一般アクション:**CI を表示** 表示ごとに，次の操作を実行します。**ビュー**権限
getChangedCIs	一般アクション:**履歴を表示**，**CI を表示**

Web サービス API コマンド	必要な権限
calculateImpact getImpactPath getImpactRulesByGroupName getImpactRulesByNamePrefix	一般アクション :**影響分析を実行**
getAllClassesHierarchy getClassAncestors getCmdbClassDefinition	なし

注: UCMDB でルート・コンテキストを変更した場合，次の手順を実行して Web Service API を有効にします。

1. **\UCMDB\UCMDBServer\deploy\axis2\WEB-INF\web.xml** 構成ファイルを開き，次のセクションを見つけます。

   ```
   <servlet-class>
   org.apache.axis2.transport.http.AxisServlet
   </servlet-class>
   ```

   上記の後ろに次の行を追加します。

   ```
   <init-param>
   <param-name>axis2.find.context</param-name>
   <param-value>false</param-value>
   </init-param>
   ```

2. **\UCMDB\UCMDBServer\deploy\axis2\WEB-INF\conf\axis2.xml** 構成ファイルを開き，次の行を見つけます。

   ```
   <parameter name="enableSwA" locked="false">false</parameter>
   ```

   上記の行の後ろに次の行を追加します。

   ```
   <parameter name="contextRoot" locked="false">test1/setup1/axis2
   </parameter>
   ```

   test1/setup1 はルート・コンテキストです。

   （ルート・コンテキストを削除するには，パスに追加したテキストを削除します。）

3. UCMDB サーバを再起動します。

HP Universal CMDB Web サービスの呼び出し

HP Universal CMDB Web Service API で 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，「UCMDB Web Service API のユース・ケース」(364ページ)を参照してください。

CMDB へのクエリ

CMDB をクエリするには，「UCMDB クエリ・メソッド」(346ページ)で説明した API を使用します。クエリおよび返される CMDB 要素には，常に実際の UCMDB 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，Query Example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 UCMDB 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ある場合でも，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，「pullTopologyMapChunks」(356ページ)と「releaseChunks」(357ページ)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，通常は必要以上のプロパティが返されるため，コスト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，これを使用します。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具体的なプロパティは，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。

CMDB 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，「UCMDB 更新メソッド」(358ページ)を参照してください。

本項の内容

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。

- **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 タイプ，および CI を識別する「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，「CIsAndRelationsUpdates」(339ページ)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，「addCIsAndRelations」(358ページ)と「UCMDB クエリ・メソッド」(346ページ)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・モデルのクエリ

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，『HP Universal CMDB 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。

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。

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，スキーマに関するドキュメントの「CmdbContext」(343ページ)。
className	タイプ名。詳細については，「タイプ名」(344ページ)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。
comments	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，「CmdbContext」(343ページ)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。
comments	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は，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 によって指定され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，「CmdbContext」(343ページ)を参照してください。
className	タイプ名。詳細については，「UCMDB の一般的なパラメータ 」(342ページ)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。
comments	内部使用専用。

影響分析へのクエリ

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ユース・ケースについては，Impact Analysis Exampleを参照してください。

UCMDB の一般的なパラメータ

本項では，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。

本項の内容

- 「構成アイテム（CI）」(344ページ)
- 「関係（Relation）」(344ページ)

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，『HP Universal CMDB モデリング・ガイド』の「［属性の追加/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，のトピックの「getCmdbClassDefinition」(341ページ)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。

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。

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，「構成アイテム（CI）」(344ページ)と「関係（Relation）」(344ページ)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，『HP Universal CMDB 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，「UCMDB の一般的なパラメータ」(342ページ)を参照してください。

関係（Relation）

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。関係要素は，ID，タイプ，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，「UCMDB クラス・モデルのクエリ」(340ページ)を参照してください。

詳細については，『HP Universal CMDB 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。しかし，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，「CIProperties」(343ページ)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，online schema documentation.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，タイプ，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（「CINode」(345ページ)を参照してください）。
- relationNodes :1 つ以上の RelationNode です（「RelationNode」(346ページ)を参照してください）。

これら 2 つの構造内の labels 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，「サイズの大きい応答の処理」(337ページ)を参照してください。

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。

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，「CmdbContext」(343ページ)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。
parameterizedNodeList	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，「TopologyMap」(346ページ)を参照してください。
chunkInfo	詳細については，「ChunkInfo」(346ページ)と「サイズの大きい応答の処理」(337ページ)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，parameterizedNodeList 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。

エクスポートされた TQL をこのメソッドの入力として使用するには，次の手順を実行します。

1. Web ブラウザを起動して次のアドレスを入力します。
   http://localhost:8080/jmx-console
   ユーザ名とパスワードを使用してログインする必要がある場合もあります。
2. **UCMDB:service=TQL Services** をクリックします。
3. **exportTql** 操作を見つけます。
   - ［**customerID**］パラメータ・ボックスに **1**（標準設定）を入力します。
   - ［**patternName**］パラメータ・ボックスに，有効な TQL 名を入力します。
4. ［**Invoke**］をクリックします。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
queryXML	リソース・タグなしで TQL を XML 文字列で表現したもの。
parameterizedNodeList	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，「TopologyMap」(346ページ)を参照してください。
chunkInfo	詳細については，「ChunkInfo」(346ページ)と 「サイズの大きい応答の処理」(337ページ)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，「CmdbContext」(343ページ)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。

パラメータ	コメント
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
getChangedCIsResponseList	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，「CmdbContext」(343ページ)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 またはグローバル 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，「タイプ名」(344ページ)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，「TypedProperties」(339ページ)のスキーマに関するドキュメント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，「TypedProperties」(339ページ)のスキーマに関するドキュメント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，「Topology」(345ページ)のスキーマに関するドキュメントを参照してください。
comments	内部使用専用。

getClsByID

getClsByID メソッドは，CMDB ID または グローバル 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，「CmdbContext」(343ページ)を参照してください。
ClsTypedProperties	タイプ別プロパティのコレクション。詳細については，「その他のプロパティ指定要素」(338ページ)のスキーマに関するドキュメント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 またはグローバル ID のみが有効です。

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，「ChunkInfo」(346ページ)と「サイズの大きい応答の処理」(337ページ)を参照してください。

getClsByType

getClsByType 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，「CmdbContext」(343ページ)を参照してください。
type	クラス名。詳細については，「タイプ名」(344ページ)を参照してください。
properties	各構成アイテム上の返されるデータ。詳細については，「CustomProperties」(339ページ)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，次を参照してください。「ChunkInfo」(346ページ)と「サイズの大きい応答の処理」(337ページ)。

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	日付 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。
Equal	文字列および数値
EqualIgnoreCase	文字列
Greater	数値
GreaterEqual	数値
In	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。
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。
Less	数値
LessEqual	数値
Like	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

演算子	条件の種類とコメント
	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	日付 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(11f);
fc.setCondition(fp);
fc.setFloatOperator(FloatCondition.FloatOperator.EQUAL);
```

継承したプロパティへのクエリの例：

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.setPropertyNames(Arrays.asList("name","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，「CmdbContext」(343ページ)を参照してください。
type	クラス名。詳細については，「タイプ名」(344ページ)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，『HP Universal CMDB 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。
properties	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，「CustomProperties」(339ページ)のスキーマに関するドキュメント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，「ChunkInfo」(346ページ)と「サイズの大きい応答の処理」(337ページ)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，「CmdbContext」(343ページ)を参照してください。
viewName	ビューの名前。これは CMDB 内のクラス・モデルのサブセットです。

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。

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，「CmdbContext」(343ページ)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
topologyMap	詳細については，「TopologyMap」(346ページ)を参照してください。
chunkInfo	詳細については，「ChunkInfo」(346ページ)と「サイズの大きい応答の処理」(337ページ)を参照してください。

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，「CmdbContext」(343ページ)を参照してください。
queryName	TQL の名前。

パラメータ	コメント
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
getTopologyQueryResultCountByNameResponse	クエリに一致するインスタンスの数。

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，「ChunkInfo」(346ページ)と「CMDB へのクエリ」(336ページ)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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)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。
queryTypedProperties	特定の CI タイプの項目について取得するプロパティのセットの集合。

出力

パラメータ	コメント
topologyMap	詳細については，「TopologyMap」(346ページ)を参照してください。
comments	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++) {
            // your code to process the CIs
         }
      }
}
```

```
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().getCINodes().size();
         m++) {
         CIs cis =
         res.getTopologyMap().getCINodes().getCINodes().get(m).getCIs();
         for(int i=0 ; i < cis.getCIs().size(); i++) {
            // your code to process the CIs
         }
      }
}
```

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，「CmdbContext」(343ページ)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。

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ディクショナリです。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)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， 「CIsAndRelationsUpdates」(339ページ)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。
dataStore	変更者情報。

出力

パラメータ	コメント
createdIDsMapList	CMDB ID にマッピングされたクライアント ID のリスト。詳細については，上の説明を参照してください。
comments	内部使用専用。

addCustomer

addCustomer メソッドは顧客を追加します。

入力

パラメータ	コメント
customerID	顧客の数値 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，「CmdbContext」(343ページ)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，「CIsAndRelationsUpdates」(339ページ) を参照してください。
dataStore	変更者情報。

removeCustomer

removeCustomer メソッドは顧客レコードを削除します。

入力

パラメータ	コメント
customerID	顧客の数値 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ディクショナリです。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)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，「CIsAndRelationsUpdates」(339ページ)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。
dataStore	変更者情報。

出力

パラメータ	コメント
createdIDsMapList	CMDB ID にマッピングされたクライアント ID のリスト。詳細については，「addCIsAndRelations」(358ページ)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。

- 「calculateImpact」(361ページ)
- 「getImpactPath」(362ページ)
- 「getImpactRulesByNamePrefix」(362ページ)

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」(362ページ) 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，「CmdbContext」(343ページ)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	CMDB またはグローバル ID 要素の集合。
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，「Topology」(345ページ)のスキーマに関するドキュメント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」(361ページ) 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，「CmdbContext」(343ページ)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。
comments	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，「CmdbContext」(343ページ)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。

Actual State Web Service API

Actual State Web Service API は主に，Service Manager によって，特定の CMDB ID かグローバル ID，および特定の顧客 ID の Actual State 情報を取得するために使用されます。API は，**Integration/SM Query** フォルダの下で一致するクエリを検索し，CMDB ID またはグローバル ID を条件として TQL を実行し，クエリの出力を返します。

Web Service の URL : http://[machine_name]:8080/axis2/services/ucmdbSMService

Web Service のスキーマ : http://[machine_name]:8080/axis2/services/ucmdbSMService?xsd=xsd0

フロー

API メソッドが呼び出されると，このメソッドは，適切なクエリを **Integration/SM Query** フォルダで探そうとします。このメソッドは，要求された CMDB ID またはグローバル ID のタイプが，そのフォルダ内のいずれかのクエリと一致するかどうかを確認しようとします。まず，**Root** という名前の **QueryElement** がないかどうかを探し，それが見つからない場合は，同じタイプの **QueryNode** を，要求された CMDB ID またはグローバル ID として使用します。適切な Query と QueryNode が見つかると，CMDB ID とグローバル ID を QueryNode に条件として置き，クエリを実行します。続いて，API の呼び出し元に結果が返されます。

変換を使用した結果の操作

XML の結果に対しては，変換を追加することもできます（たとえば，すべてのディスクのサイズの合計を計算して，それを CI の属性として追加するなど）。TQL の結果に変換を追加するには，アダプタ構成に **[tql_name].xslt** という名前のリソースを配置します（［**アダプタ管理**］> ［**ServiceDeskAdapter7-1**］> ［**構成ファイル**］> ［**[tql_name].xslt**］）。

Actual State Web Service API のログ

UCMDB のログ設定は，**UCMDBServer/Conf/log** の下にある各種 ***.properties** ファイルにあります。

SM Actual State フローのログを表示するには，次の手順を実行します。

1. **cmdb_soaapi.properties** ファイルを開き，ログ・レベルを DEBUG に変更します（**loglevel=DEBUG**）。
2. **fcmdb.properties** ファイルを開き，ログ・レベルを DEBUG に変更します（**loglevel=DEBUG**）。
3. サーバが変更を取得するまで 1 分待ちます。

4. Actual State を SM から実行します。
5. **UCMDBServer/Runtime/log** で次のログ・ファイルを表示します。
   - cmdb.soaapi.log
   - fcmdb.log

ルート・コンテキストの変更後に複製された CI の実際の状態の有効化

UCMDB へのアクセスに使用するルート・コンテキストを変更した場合，次の設定変更を行なって，複製された CI の実際の状態を有効化しなければなりません。

1. **UCMDBServer\deploy\axis2\WEB-INF** で，**web.xml** ファイルを開きます。
2. 次の **servlet init** パラメータを AxisServlet に追加します（これらの 4 行を行 28 の後に貼り付けます）。

   ```
   <init-param>
   <param-name>axis2.find.context</param-name>
   <param-value>false</param-value>
   </init-param>
   ```

   この設定により Axis2 がコンテキスト・ルートの計算を試みることを防ぎ，**axis2.xml** を検索することを明示的に指示できます。
3. **UCMDBServer\deploy\axis2\WEB-INF\conf** で，**axis2.xml** ファイルを開きます。
4. 行 58 で，パラメータ **contextRoot** のコメント指定を解除し，次のように編集します。

   ```
   <parameter name="contextRoot" locked="false">test/axis2</parameter>
   ```

   （**test** は **cmdb.xml** の新しいルート・コンテキストです）。

   注: test/axis2 の最初にはスラッシュはありません。

UCMDB Web Service API のユース・ケース

次のユース・ケースは 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

ユース・ケース:

- **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 へのクエリ

ユース・ケース:

- **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・モデルへのクエリ

ユース・ケース:

- **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

ユース・ケース:

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

次のコード例を参照してください。

- The Example Base Class
- Query Example
- Update Example
- Class Model Example
- Impact Analysis Example

ファイルは次のディレクトリにあります。

\\<UCMDB のルート・ディレクトリ>\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\WebServiceAPI_Samples

第12章: データ・フロー管理 Java API

本章の内容

データ・フロー管理 Java API の使用

注: 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 DFM API Javadoc と併せて使用してください。

本章では，サードパーティ・ツールまたはカスタム・ツールを使用して HP データ・フロー管理 Java API でのデータ・フロー管理を管理する方法について説明します。この API により，次を実行するためのメソッドが提供されます。

- **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。
- **一般データの表示**: ドメインおよびプローブに関するデータ。

Discovery Services パッケージでは次のサービスを使用できます。

- **DDMConfigurationService :** Data Flow Probe，クラスタ，IP 範囲，資格情報を構成するサービスです。Universal Discoveryサーバは XML ファイルを使用するか，Data Flow Probe を介して構成できます。
- **DDMManagementService :** Universal Discovery の進行状況，結果，エラーを分析して表示するサービスです。
- **DDMSoftwareSignatureService :** Data Flow Probe コンポーネントで検出するソフトウェア項目を定義するサービスです。この定義はシステム全体に適用されます。複数の Data Flow Probe コンポーネントを定義している場合，これらすべてのコンポーネントにこの定義が適用されます。
- **DDMZoneService :** ゾーンベースのディスカバリを管理するサービスです。

これらのサービスに加え，Jython アダプタの作成で使用するデータ・フロー管理クライアント API も存在します。詳細については，「Jython アダプタの開発」(37ページ)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 で定義されている統合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。

さらに，ユーザがログインするには，［**SDK へアクセス**］一般アクション権限がなければなりません。

注意: API クライアントも，API 認証権限が付与されていれば，標準のユーザで操作できます。ただし，この方法は推奨されません。

詳細については，「統合ユーザの作成」(328ページ)を参照してください。

第13章: データ・フロー管理 Web サービス API

本章の内容

データ・フロー管理 Web サービス API の概要

本章では，サードパーティ・ツールまたはカスタム・ツールを使った HP データ・フロー管理 Web サービス API でのデータ・フローの管理方法について説明します。

HP データ・フロー管理 Web サービス API は，アプリケーションを HP Universal CMDB に統合するために使用されます。この API により，次を実施するメソッドが提供されます。

- **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。
- **一般データの表示**: ドメインおよびプローブに関するデータ。

HP データ・フロー管理 Web サービスのユーザは，次のことを十分理解している必要があります。

- SOAP の仕様
- オブジェクト指向プログラミング言語（C++，C#，Java など）
- HP Universal CMDB
- データ・フロー管理

注:

- ユーザがログインするためには **Run Legacy API** 一般アクション権限がなければなりません。
- ログイン・ユーザがメソッドのいずれかにアクセスするには「**ディスカバリおよび統合を実行**」一般アクション権限がなければなりません。

実行できる操作の完全なドキュメントについては，*HP Universal Discovery Schema Reference*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。

<UCMDB のルート・ディレクトリ>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_Schema\webframe.html

表記規則

本章では，次の表記を使用します。

- Element :この書体は，その項目がデータベース内のエンティティか，スキーマ内で定義されている要素であることを示します。これには，メソッドに渡されたりメソッドから返されたりする構造体も含まれます。通常の書体は，その項目が一般的な説明であることを示します。
- データ・フロー管理要素およびメソッドの引数は，それらがスキーマ内で指定されているとおりに，大文字と小文字を区別して表記されます。これは通常，クラス名や，クラス・インスタンスに対する一般的な参照において，先頭の文字が大文字で表記されることを意味します。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，credential は，メソッドに渡される Credential タイプの要素です。

HP データ・フロー管理 Web サービスの呼び出し

HP データ・フロー管理 Web サービス API では，標準の SOAP プログラミング技術を使ってサーバ側のメソッドを呼び出せます。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 例外が発生すると，サービスは 1 つ以上のエラー・メッセージ，エラー・コード，および例外メッセージ・フィールドに情報を書き込みます。エラーがなければ，呼び出しの結果が返されます。

サービスを呼び出すには，以下の情報を使用します。

- Protocol:http または https（サーバ設定に依存）
- URL : <UCMDB サーバ>:8080/axis2/services/DiscoveryService
- 標準設定のパスワード : "admin"
- 標準設定のユーザ名 :"admin"

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。

- axis2/services/DiscoveryService?wsdl

データ・フロー管理メソッドとデータ構造

本項では，データ・フロー管理 Web サービス API メソッドとデータ構造を示し，その使用方法の概要を示します。各操作の要求と応答の完全な説明については，『HP Universal Discovery Schema Reference』を参照してください。

本項の内容

データ構造

これらはデータ・フロー管理 Web サービス API で使用するデータ構造の一部です。

CIProperties

CIProperties は，コレクションのコレクションです。各コレクションには，異なるデータ・タイプのプロパティが含まれます。たとえば，dateProps コレクション，strListProps コレクション，xmlProps コレクションなどになります。

各タイプ・コレクションには，所定のタイプの個別のプロパティが含まれます。これらのプロパティ要素の名前はコンテナと同じですが，単数形であることが異なります。たとえば，dateProps には dateProp 要素が含まれます。各プロパティは名前と値のペアです。

*HP Universal Discovery Schema Reference*の CIProperties を参照してください。

IPList

IP 要素のリストです。各リストには，IPv4 または IPv6 アドレスが含まれます。

『HP Universal Discovery Schema Reference』の IPList を参照してください。

IPRange

IPRange には，Start と End の 2 つの要素が含まれます。それぞれの要素には Address 要素が含まれます。これは IPv4 または IPv6 アドレスです。

『HP Universal Discovery Schema Reference』の IPRange を参照してください。

Scope

2 つの IPRanges です。Exclude はジョブから除外する IPRanges のコレクションです。Include はジョブに含める IPRanges のコレクションです。

『HP Universal Discovery Schema Reference』の Scope を参照してください。

ディスカバリ・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。

「コード・サンプル」(382ページ)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，「CmdbContext」(343ページ)を参照してください。
JobName	ジョブの名前です。

deactivateJob

指定されたジョブを非アクティブにします。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
JobName	ジョブの名前です。

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。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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
JobName	ジョブの名前です。
CIID	トリガ CI の ID です。
ProbeName	プローブの名前です。
Timeout	ミリ秒単位です。

getDiscoveryJobsNames

ジョブ名のリストを返します。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。

出力

パラメータ	コメント
strList	ジョブ名のリストです。

isJobActive

ジョブがアクティブかどうかをチェックします。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
JobName	チェックするジョブの名前です。

出力

パラメータ	コメント
JobState	ジョブがアクティブな場合は True です。

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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
JobName	ジョブの名前です。
CIID	トリガ CI の ID です。

addTriggerTQL

指定されたジョブに新しいトリガ TQL を追加します。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
JobName	ジョブの名前です。
TqlName	追加する TQL の名前です。

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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
JobName	ジョブの名前です。

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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
JobName	ジョブ名です。
CIID	トリガ CI の ID です。

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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
JobName	チェックするジョブ名のコレクションです。
CIID	削除する TQL の ID です。

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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
JobName	ジョブの名前です。
tqlName	TQL の名前です。
probesLimit	TQL がアクティブなプローブのリストです。

ドメイおよびプローブ・データ・メソッド

getDomainType

ドメイン・タイプを返します。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
domainName	ドメインの名前です。

出力

パラメータ	コメント
domainType	ドメイン・タイプです。

getDomainsNames

現在のドメインの名前を返します。

「コード・サンプル」(382ページ)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，「CmdbContext」(343ページ)を参照してください。

出力

パラメータ	コメント
domainNames	ドメイン名のリストです。

getProbeIPs

指定されたプローブの IP アドレスを返します。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
domainName	チェックするドメインです。
probeName	ドメインで使用するプローブの名前です。

出力

パラメータ	コメント
probeIPs	プローブのアドレスの「IPList」です。

getProbesNames

指定されたドメイン内のプローブの名前を返します。

「コード・サンプル」(382ページ)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，「CmdbContext」(343ページ)を参照してください。
domainName	チェックするドメインです。

出力

パラメータ	コメント
probesName	ドメインのプローブのリストです。

getProbeScope

指定されたプローブの対象範囲の定義を返します。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
domainName	チェックするドメインです。
probeName	プローブの名前です。

出力

パラメータ	コメント
probeScope	プローブの「Scope」です。

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。

「コード・サンプル」(382ページ)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，「CmdbContext」(343ページ)を参照してください。
domainName	チェックするドメインです。
probeName	チェックするプローブです。

出力

パラメータ	コメント
isConnected	プローブが接続されている場合は True です。

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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
domainName	ドメインです。
probeName	更新するプローブです。
newScope	プローブに対して設定する「Scope」です。

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。

「コード・サンプル」(382ページ)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，「CmdbContext」(343ページ)を参照してください。
domainName	更新するドメインです。
protocolName	プロトコルの名前です。
credentialsEntryParameters	新しい資格情報の「CIProperties」コレクションです。

出力

パラメータ	コメント
credentialsEntryID	新しい資格情報エントリの CI ID です。

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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
domainName	資格情報を取得するドメインです。
protocolName	ドメインで使用するプロトコルの名前です。

出力

パラメータ	コメント
credentialsEntryIDs	ドメインのプロトコルに対する資格情報 ID のリストです。

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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
domainName	資格情報を取得するドメインです。
protocolName	ドメインで使用するプロトコルの名前です。
credentialsEntryID	取得する資格情報 ID です。

出力

パラメータ	コメント
credentialsEntryParameters	資格情報の「CIProperties」コレクションです。

removeCredentialsEntry

指定された資格情報をプロトコルから削除します。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
domainName	ドメインです。
protocolName	ドメインで使用するプロトコルの名前です。
credentialsEntryID	削除する資格情報の ID です。

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。

既存のプロパティは削除され，これらのプロパティが設定されます。この呼び出しで値が設定されていないプロパティは，定義されないままになります。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。

パラメータ	コメント
domainName	資格情報を更新するドメインです。
protocolName	ドメインで使用するプロトコルの名前です。
credentialsEntryID	更新する資格情報の ID です。
credentialsEntryParameters	資格情報のプロパティとして設定する「CIProperties」コレクションです。

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。 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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
CmdbIDs	再検出するオブジェクト ID のコレクションです。

出力

パラメータ	コメント
isSucceed	CI の再検出が成功した場合は True です。

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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
CmdbIDs	追跡する再検出呼び出しのオブジェクト ID のコレクションです。

出力

パラメータ	コメント
progress	完了したジョブの進行状況は 1 です。完了していないジョブは 1 未満の小数になります。

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。**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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
viewName	チェックするビューです。

出力

パラメータ	コメント
isSucceed	CI の再検出が成功した場合は True です。

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。

入力

パラメータ	コメント
cmdbContext	詳細については，「CmdbContext」(343ページ)を参照してください。
viewName	チェックするビューのコレクションです。

出力

パラメータ	コメント
progress	完了したジョブの進行状況は 1 です。完了していないジョブは 1 未満の小数になります。

コード・サンプル

```
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.*;
import com.hp.ucmdb.generated.types.*;
public class test {
   static final String HOST_NAME = "<my_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 = "<my_password>";
   private static final String USERNAME = "<my_username>";

   private static CmdbContext cmdbContext = new CmdbContext("ws tests");

   public static void main(String[] args) throws Exception {
      // Get the stub object
      DiscoveryService discoveryService = getDiscoveryService();

      // Activate Job
      discoveryService.activateJob(new ActivateJobRequest(
         "Range IPs by ICMP", cmdbContext));

      // Get domain & probes info
      getProbesInfo(discoveryService);
      // Add credentilas entry for ntcmd protcol
      addNTCMDCredentialsEntry();
   }

   public static void addNTCMDCredentialsEntry() throws Exception {
      DiscoveryService discoveryService = getDiscoveryService();

      // Get domain name
      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);
      // Create propeties with one byte param
```

```
        CIProperties newCredsProperties = new CIProperties();

        // Add password property - this is of type bytes
        newCredsProperties.setBytesProps(new BytesProps());
        setPasswordProperty(newCredsProperties);

        // Add user & domain properties - these are of type string
        newCredsProperties.setStrProps(new StrProps());
        setStringProperties("protocol_username", "test user", newCredsProperties);
        setStringProperties("ntadminprotocol_ntdomain",
           "test doamin", newCredsProperties);

        // Add new credentials entry
        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 ));
        // Go over all the domains
        if (result.getDomainNames().sizeStrValueList() > 0) {
            String domainName =
            result.getDomainNames().getStrValue(0);
            GetProbesNamesResponse probesResult =
               discoveryService.getProbesNames(
                  new GetProbesNamesRequest(domainName, cmdbContext));
            // Go over all the probes
            for (int i=0; i<probesResult.getProbesNames().sizeStrValueList(); i++) {
               String probeName = probesResult.getProbesNames().getStrValue(i);
               // Check if connected
               IsProbeConnectedResponce connectedRequest =
                  discoveryService.isProbeConnected(
                     new IsProbeConnectedRequest(
```

```
                domainName, probeName, cmdbContext));
            Boolean isConnected = connectedRequest.getIsConnected();
            // Do something ...
            System.out.println("probe " + probeName + " isconnect=" + isConnected);
        }
      }
    }

    private static DiscoveryService getDiscoveryService() throws Exception {
      DiscoveryService discoveryService = null;
      try {
        // Create service
        URL url = new URL(PROTOCOL,HOST_NAME,PORT, FILE);
        DiscoveryServiceStub serviceStub =
          new DiscoveryServiceStub(url.toString());

        // Authenticate info
        HttpTransportProperties.Authenticator auth =
          new HttpTransportProperties.Authenticator();
        auth.setUsername(USERNAME);
        auth.setPassword(PASSWORD);
        serviceStub._getServiceClient().getOptions().setProperty(
          HTTPConstants.AUTHENTICATE,auth);

        discoveryService = serviceStub;
      } catch (Exception e) {
        throw new Exception("cannot create a connection to service ", e);
      }
      return discoveryService;
    }
}
```

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 {
      // Get the stub object
      DiscoveryService discoveryService = getDiscoveryService();

      // Activate Job
      discoveryService.activateJob(new ActivateJobRequest("Range IPs by ICMP", cmdbContext));

      // Get domain & probes info
      getProbesInfo(discoveryService);
      // Add credentilas entry for ntcmd protcol
      addNTCMDCredentialsEntry();
   }

   public static void addNTCMDCredentialsEntry() throws Exception {
      DiscoveryService discoveryService = getDiscoveryService();

      // Get domain name
      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);
      // Create propeties with one byte param
      CIProperties newCredsProperties = new CIProperties();

      // Add password property - this is of type bytes
      newCredsProperties.setBytesProps(new BytesProps());
      setPasswordProperty(newCredsProperties);

      // Add user & domain properties - these are of type string
      newCredsProperties.setStrProps(new StrProps());
      setStringProperties("protocol_username", "test user", newCredsProperties);
      setStringProperties("ntadminprotocol_ntdomain", "test doamin", newCredsProperties);

      // Add new credentials entry
      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 ));
        // Go over all the domains
        if (result.getDomainNames().sizeStrValueList() > 0) {
            String domainName = result.getDomainNames().getStrValue(0);
            GetProbesNamesResponse probesResult =
                discoveryService.getProbesNames(new GetProbesNamesRequest(domainName, cmdbContext));
            // Go over all the probes
            for (int i=0; i<probesResult.getProbesNames().sizeStrValueList(); i++) {
                String probeName = probesResult.getProbesNames().getStrValue(i);
                // Check if connected
                IsProbeConnectedResponce connectedRequest =
                    discoveryService.isProbeConnected(new IsProbeConnectedRequest(domainName, probeName,
cmdbContext));
                Boolean isConnected = connectedRequest.getIsConnected();
                // Do something ...
                System.out.println("probe " + probeName + " isconnect=" + isConnected);
            }
        }
    }

    private static DiscoveryService getDiscoveryService() throws Exception {
        DiscoveryService discoveryService = null;
        try {
            // Create service
            URL url = new URL(PROTOCOL,HOST_NAME,PORT, FILE);
            DiscoveryServiceStub serviceStub = new DiscoveryServiceStub(url.toString());

            // Authenticate info
            HttpTransportProperties.Authenticator auth = new HttpTransportProperties.Authenticator();
            auth.setUsername(USERNAME);
            auth.setPassword(PASSWORD);
```

```
            serviceStub._getServiceClient().getOptions().setProperty(HTTPConstants.AUTHENTICATE,auth);

            discoveryService = serviceStub;
        } catch (Exception e) {
            throw new Exception("cannot create a connection to service ", e);
        }
        return discoveryService;
    }
} // End class
```

ドキュメントに関するフィードバックの送信

このドキュメントに関するコメントについては，電子メールでドキュメント・チームまでご連絡ください。ご使用のシステムに電子メール・クライアントが設定されている場合は，上記のリンクをクリックすると電子メールウィンドウが開き，以下の情報が件名の行に表示されます。

開発者向け参考情報ガイド（Universal CMDB 10.20）に関するフィードバック

電子メールにフィードバックを記入して，送信ボタンをクリックしてください。

使用できる電子メール・クライアントがない場合は，上記の情報を Web メール・クライアントの新しいメッセージにコピーして，フィードバックを cms-doc@hp.com に送信してください。

お客様からのご意見をお待ちしております。