HP Universal CMDB

适用于 Windows 和 Linux 操作系统

软件版本：9.02

开发人员参考指南

文档发行日期：2010 年 10 月
软件发布日期：2010 年 10 月

法律声明

担保

随 HP 产品及服务提供的明示性担保声明中阐明了适用于 HP 产品及服务的专用担保条款。本文档所含信息均不构成额外的担保。HP 对本文档中的技术或编辑错误以及缺漏不负任何责任。

本文档中所包含的信息如有更改，恕不另行通知。

受限权利声明

机密计算机软件。必须拥有 HP 授予的有效许可证，方可拥有、使用或复制本软件。按照 FAR 12.211 和 12.212，并根据供应商的标准商业许可的规定，“商业计算机软件”、“计算机软件文档”与 “商品技术数据”授权给美国政府使用。

版权声明

商标声明

Adobe® 和 Acrobat® 是 Adobe Systems Incorporated 的商标。

AMD 和 AMD 箭头符号是 Advanced Micro Devices, Inc. 的商标。

Google™ 和 Google Maps™ 是 Google Inc 的商标。

Intel®、 Itanium®、 Pentium® 和 Intel® Xeon® 是 Intel Corporation 在美国和其他国家 / 地区的商标。

Java™ 是 Sun Microsystems, Inc 在美国的商标。

Microsoft®、 Windows®、 Windows NT®、 Windows® XP 和 Windows Vista® 是 Microsoft Corporation 在美国的注册商标。

Oracle 是 Oracle Corporation 和 / 或其附属机构的注册商标。

UNIX® 是 The Open Group 的注册商标。

致谢

- 本产品包括由 Apache 软件基金会 (http://www.apache.org/licenses) 开发的软件。

- 本产品包括 OpenLDAP Foundation (http://www.openldap.org/foundation/) 提供的 OpenLDAP 代码。
- 本产品包括 Free Software Foundation, Inc (http://www.fsf.org/) 提供的 GNU 代码。
- 本产品包括 Dennis M. Sosnoski 提供的 JiBX 代码。
- 本产品包括印第安纳大学 Extreme! 实验室提供的 XPP3 XMLPull 解析器，该解析器在发行时随附，并用于整个 JiBX 中。
- 本产品包括 Robert Futrell 提供的 Office 外观和体验许可证 (http://sourceforge.net/projects/officelnfs)。
- 本产品包括 Netaphor Software, Inc (http://www.netaphor.com/home.asp) 提供的 JEP - Java 表达式解析器代码。

文档更新

本文档的标题页包含以下标识信息：

- 软件版本号，用于指示软件版本。
- 文档发行日期，该日期将在每次更新文档时更改。
- 软件发布日期，用于指示该版本软件的发布日期。

要检查是否有最新的更新，或者验证您是否正在使用最新版本的文档，请访问：

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

需要注册 HP passport 才能登录此网站。要注册 HP passport ID，请访问：

http://h20229.www2.hp.com/passport-registration.html

或单击“HP Passport”登录页面上的“New users - please register”链接。

此外，如果订阅了相应的产品支持服务，则还会收到更新的版本或新版本。有关详细信息，请与您的 HP 销售代表联系。

支持

可以访问 HP Software 支持网站：

http://www.hp.com/go/hpsoftwaresupport

本网站介绍了有关 HP Software 提供的产品、服务和支持的联系人信息和其他详细信息。

HP Software 联机支持为客户提供了独立解决问题的功能。该联机支持提供了一种快速访问交互式技术支持工具的有效方法，这些技术支持工具对于业务管理是必不可少的。作为我们的尊贵客户，您可以通过该支持网站获得下列支持：

- 搜索感兴趣的知识文档
- 提交并跟踪支持案例和改进请求
- 下载软件修补程序
- 管理支持合同
- 查找 HP 支持联系人
- 查看有关可用服务的信息
- 参与其他软件客户的讨论
- 研究和注册软件培训

大多数提供支持的区域都要求您注册为 HP Passport 用户再登录，很多区域还会要求您提供支持合同。 要注册 HP Passport ID，请访问：

http://h20229.www2.hp.com/passport-registration.html

要查找有关访问级别的详细信息，请访问：

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

目录

欢迎使用本指南

本指南介绍如何创建和管理适配器，这些适配器用于与外部数据库和其他 CMDB 之间传输数据。

本章包括：

本指南的结构

本指南包括以下章节：

第 I 部分　创建搜寻和集成适配器

说明如何创建适配器。

第 II 部分 使用 API

说明如何使用 API 从 HP Universal CMDB 中提取配置数据。

本指南的目标读者

本指南的目标读者包括以下 HP Universal CMDB 用户：

- HP Universal CMDB 管理员
- HP Universal CMDB 平台管理员
- HP Universal CMDB 应用程序管理员
- HP Universal CMDB 数据管理员

本指南的读者应了解企业系统管理方面的知识并且熟悉 ITIL 概念，同时具备 HP Universal CMDB 的相关知识。

HP Universal CMDB 联机文档

HP Universal CMDB 包括以下联机文档：

自述文件。提供了有关版本局限性和最近更新的信息。在 HP Universal CMDB DVD 根目录下双击 **readme.html**，即可访问该文件。另外，您还可以从 HP Software 支持网站访问最近更新的自述文件。

新增功能。提供了有关新增功能和版本要点的信息。在 HP Universal CMDB 中，选择 “帮助” > “新增功能”。

适合打印的文档。选择 “帮助” > “UCMDB 帮助”。以下指南仅以 PDF 格式提供：

- 《HP Universal CMDB 部署指南》 PDF 文档。说明在安装 HP Universal CMDB 时所需的硬件和软件要求、如何安装或升级 HP Universal CMDB、如何强化系统以及如何登录应用程序。
- 《HP Universal CMDB 数据库指南》 PDF 文档。说明如何设置 HP Universal CMDB 所需的数据库 （MS SQL Server 或 Oracle）。

- 《HP Universal CMDB Discovery and Integration Content Guide》PDF 文档。说明如何运行搜寻功能，以查找在系统上运行的应用程序、操作系统和网络组件；还说明了如何通过集成来搜寻其他数据库中的数据。

HP Universal CMDB 联机帮助包括：

- **建模**。支持您管理 IT 世界模型的内容。
- **数据流管理**。说明如何将 HP Universal CMDB 与其他数据库集成，以及如何通过设置 HP Universal CMDB 来搜寻网络组件。
- **UCMDB 管理**。说明如何使用 HP Universal CMDB。
- **开发人员参考**。适用于对 HP Universal CMDB 十分了解的用户。说明了如何定义和使用适配器，以及如何使用 API 访问数据。

您还可以通过 HP Universal CMDB 的特定窗口来访问联机帮助，方法是单击特定窗口，然后单击 “帮助”按钮。

可以使用 Adobe Reader 软件查看和打印联机丛书。可以从 Adobe 网站 (www.adobe.com) 下载此软件。

主题类型

在本指南中，每个主题区域均由一些主题组成，每个主题包含不同的主题信息模块。这些主题通常按照它们所包含的信息类型进行分类。

这种结构将文档分为适用于不同情况的信息类型，便于您轻松访问特定信息。

本指南中的主题类型主要有三种：**概念**、**任务**和**参考**。不同的主题类型使用不同的图标进行区分，简洁直观。

主题类型	描述	用途
概念 	背景性、描述性或概念性信息。	了解有关功能用途的一般信息。
任务 	**指导性任务。**分步指导，用于帮助您使用应用程序并完成任务。某些任务步骤包含一些示例，这些示例会使用样本数据来进行更好的说明。 任务步骤有的带编号，有的不带编号： ➤ **带编号的步骤。**按照顺序依次执行任务中的各步骤。 ➤ **不带编号的步骤。**一系列独立的操作，可以按照任意顺序执行。	➤ 了解任务的整体工作流程。 ➤ 执行带编号的任务中列出的步骤，以完成任务。 ➤ 完成不带编号的任务中列出的步骤，以执行独立的操作。
	用例场景任务。演示如何在特定情况下执行任务的示例。	了解如何在真实场景中执行任务。

主题类型	描述	用途
参考 	**一般参考**。参考材料的详细列表和解释。	查找与特定上下文相关的特定参考信息。
	用户界面参考。具有针对性的参考主题，对特定的用户界面进行详细描述。通常，在产品的“帮助”菜单中选择“本页帮助”会打开用户界面主题。	查找有关输入内容的特定信息，或如何使用一个或多个特定用户界面元素（例如窗口、对话框或向导）的特定信息。
疑难解答和局限性 	**疑难解答和局限性**。具有针对性的参考主题，描述常见问题及其解决办法，并列出功能或产品区域的局限性。	用于在使用功能或遇到软件使用问题之前，提高您对重要问题的认知程度。

其他联机资源

疑难解答和知识库。可访问 HP Software 支持网站上的“疑难解答”页面，可在该页面搜索“自助解决”知识库。要访问此网站，请选择“帮助”>“疑难解答和知识库”。此网站的 URL 是 http://h20230.www2.hp.com/troubleshooting.jsp。

HP Software 支持。可访问 HP Software 支持网站。通过此网站，您不但可以浏览“Self-solve knowledge base”，还可以搜索用户论坛并将信息发布到论坛、提交支持请求、下载修补程序和最新文档等。要访问此网站，请选择“帮助”>“HP Software 支持”。此网站的 URL 是 www.hp.com/go/hpsoftwaresupport。

大多数提供支持的区域都要求您注册为 HP Passport 用户再登录，很多区域还会要求您提供支持合同。

要查找有关访问级别的详细信息，请访问：

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

要注册 HP Passport 用户 ID，请访问：

http://h20229.www2.hp.com/passport-registration.html

HP Software 网站。可访问 HP Software 网站。此站点提供了 HP Software 产品的最新信息，具体包括新软件版本、研讨会和展销会、客户支持等。要访问此网站，请选择 “帮助” > “HP Software 网站”。此网站的 URL 是 www.hp.com.cn/software。

文档更新

HP Software 将不断更新其产品文档。

要检查是否有最新更新，或验证所使用的文档是否为最新版本，请访问 HP Software 产品手册网站 (http://h20230.www2.hp.com/selfsolve/manuals)。

第 I 部分

创建搜寻和集成适配器

1

适配器开发与写入

本章包括：

概念

任务

概念

适配器开发与编写概述

在开始实际规划新适配器的开发之前，首先了解开发相关的一般流程和交互至关重要。

以下各节介绍要成功管理和执行搜寻开发项目，您必须掌握并执行的操作。

本章：

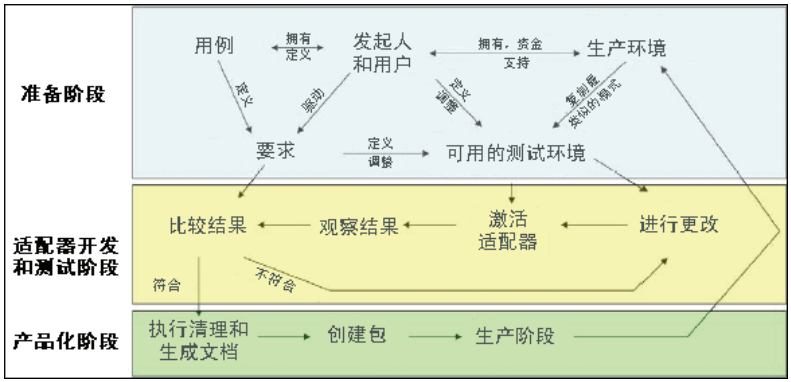
- 假定您已掌握了 HP Universal CMDB 的应用知识，并对系统元素有一些基本了解。旨在帮助您完成学习过程，并没有包含完整的说明。
- 包含为 HP Universal CMDB 规划、研究和执行新搜寻内容的各个阶段，并包含相关的准则和注意事项。
- 提供有关数据流管理框架关键 API 的信息。有关可用 API 的完整文档，请参阅《HP Universal CMDB Data Flow Management API Reference》。（虽然也存在其他非正式 API，但是这些 API 只适用于现成的适配器，而且可能还需要更改。）

内容创建

本节包括：

适配器开发周期

下图为适配器开发流程图。其中的大部分时间耗费在中间环节，该环节涉及开发和测试过程的迭代循环。



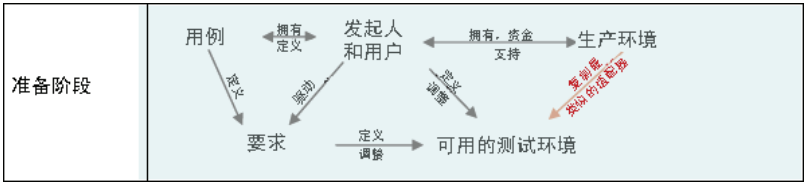
适配器开发过程中的每个阶段都基于上一阶段。

在适配器的外观和工作方式满足要求之后，便可以对其进行打包。您既可使用 UCMDB 包管理器，也可使用组件的手动导出功能，来创建 *.zip 打包文件。最佳做法是，在将打包文件发布到生产环境之前，在其他 UCMDB 系统上对其进行部署和测试，以确保所有组件都经过验证并成功打包。有关如何打包的详细信息，请参阅 《HP Universal CMDB 管理指南》中的 “包管理器”。

以下各节详细介绍了各个阶段中最关键的步骤和最佳做法：

- 研究和准备阶段
- 适配器开发和测试
- 适配器的包装和产品化

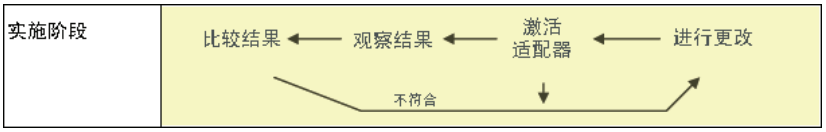
研究和准备阶段



研究和准备阶段包括驱动性的业务需求和用例，还需要确保开发和测试适配器时所需的安全设施。

1. 计划修改现有适配器时，第一个技术步骤是对适配器进行备份，并确保可以将其恢复到原始状态。如果计划创建新适配器，请复制最相似的适配器，然后使用合适的名称保存。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “资源窗格”。
2. 研究适配器的数据收集方式：
   - 使用外部工具 / 协议获取数据
   - 研究适配器如何基于这些数据创建 CI
   - 了解相似的适配器
3. 根据以下条件确定最相似的适配器：
   - 创建的 CI 相同
   - 使用的协议相同 (SNMP)
   - 目标类型相同 （按 OS 类型、版本等判断）
4. 复制整个程序包。

5 将程序包解压缩到工作区，然后重命名适配器 (XML) 和 Jython (.py) 文件。



适配器开发和测试

适配器开发和测试阶段是一个高度迭代的过程。在适配器开始成型时，即可基于最终用例开始测试，执行更改，然后重新测试。重复此过程，直到适配器符合要求为止。

复制的启动和准备

- 修改适配器的 XML 部分：第 1 行中的名称 (id)、创建的 CI 类型和调用的 Jython 脚本名称。
- 运行副本，并生成与原始适配器相同的结果。
- 注释掉大部分代码，尤其是产生结果的关键代码。

开发和测试

- 使用其他示例代码开发变更
- 运行适配器以对其进行测试
- 使用专用视图来验证复杂的结果，或通过搜索来验证简单的结果

适配器的包装和产品化

适配器打包和产品化阶段是开发过程的最后一个阶段。作为最佳做法，在打包之前，需要先清理调试的残余部分、文档和注释，以及查看安全注意事项等。您至少需要阅读自述文件，才能了解适配器的内部工作状况。将来，如果有人（也许就是您自己）需要查看此适配器，则可以从最小范围的文档中获得诸多帮助。

清理工作和文档编写

- 删除调试代码
- 对所有函数进行注释，并在主要部分中添加开放的注释
- 创建示例 TQL 和视图，供用户进行测试

创建包

- 使用包管理器导出适配器和 TQL 等。有关详细信息，请参阅 《HP Universal CMDB 管理指南》中的 “包管理器”。
- 检查您的包与其他包的关系，例如，某些包所创建的 CI 是您的适配器的输入 CI。
- 使用包管理器创建 zip 包。有关详细信息，请参阅 《HP Universal CMDB 管理指南》中的 “包管理器”。
- 通过删除部分新内容和重新部署，或通过在其他测试系统上进行部署来实施测试部署。

数据流管理和集成

DFM 适配器可以与其他产品集成。请考虑以下定义：

- DFM 从多个目标收集特定内容。
- 集成项目从一个系统收集多种类型的内容。

请注意，这些定义在收集方式上并无不同， DFM 也一样。开发新适配器的过程与开发新集成的过程相同。与现有适配器比较，对新适配器进行的研究、选择以及编写操作都相同。在这其中仅会发生少数更改：

- 最终适配器的计划。集成适配器可能比搜寻运行得更加频繁，但需要视用例而定。
- 输入 CI：
  - 集成：要运行的非 CI 触发器无输入：文件名或源通过适配器参数传递。
  - 搜寻：使用常规 CMDB CI 输入。

对于集成项目，始终需要重用现有适配器。集成方向 （从 HP Universal CMDB 到其他产品，或从其他产品到 HP Universal CMDB）可能会影响开发方式。提供了一些形成包以供您复制。您可使用成熟的技术自行使用这些包。

从 HP Universal CMDB 到其他项目：

- 创建 TQL，生成 CI 及其关系供导出。
- 使用常规打包适配器执行 TQL，并将结果写入到 XML 文件，供外部产品读取。

注意：有关字段包的示例，请联系 HP Software 支持。

将其他产品与 HP Universal CMDB 集成：根据其他产品的数据提供方式的不同，集成适配器的操作也有所差异：

集成类型	要重用的参考示例
直接访问产品的数据库	HP ED
在导出操作生成的 CSV 或 XML 文件中读取	HP ServiceCenter
访问产品的 API	BMC Atrium/Remedy

关联业务价值与搜寻开发

开发新搜寻内容的用例应以业务情况为基础，并计划生产业务价值。也就是说，将系统组件映射到 CI 并将 CI 添加到 CMDB 的目的是为了提供业务价值。

虽然这些内容是许多用例的常用中间步骤，但是不可能始终适用于应用程序映射。无论内容的最终用途如何，您的计划需要解答下列问题：

- 谁是消费者？消费者要如何处理 CI 提供的信息以及 CI 之间的关系？您要查看的 CI 及其关系所在的业务环境如何？这些 CI 的消费者是个人、产品还是两者兼有？
- 当 CMDB 中存在最佳的 CI 和关系组合时，如何使用这些 CI 和关系进行计划以生成业务价值？
- 最佳映射应该是什么样的？
  - 什么术语最适合于描述各 CI 之间的关系？
  - 要包括的最重要的 CI 类型是什么？
  - 映射的最终用途是什么？最终用户是谁？
- 最佳报告布局应该是什么？

建立业务调整之后，下一步就需要将业务价值包括在文档中。这意味着需要使用绘图工具绘制最佳映射，需要了解 CI、报告之间的影响和依赖关系，更改的跟踪方式、重要更改，以及用例要求的监控、合规性和其他业务价值。

此绘图 （或模型）称为**蓝图**。

例如，如果应用程序必须获知某个配置文件发生更改的时间，则需要将该文件映射并链接到所绘图中的相应 CI （与该文件相关）。

与区域的 SME（主题问题专家），即所开发内容的最终用户合作。此专家将会指出 CMDB 中必须存在的关键实体 （具有属性和关系的 CI），以提供业务价值。

方法之一是向应用程序所有者 （在这种情况下也包括 SME）提供问卷表，该所有者不但能够指定上述目标和蓝图，而且至少要提供该应用程序当前使用的一个体系结构。

只需映射关键数据，而不映射无关数据：您可在以后随时增强适配器的功能。这样做的目的是设置一个有限的搜寻，用于处理和提供值。映射大量数据虽然可以提供更加清晰直观的图，但是在开发时容易引起混淆，而且耗时过长。

了解模型和业务价值之后，便可以继续到下一个阶段。鉴于以下各阶段提供了更多具体信息，所以您可以随时重新查看本阶段的内容。

研究集成要求

本阶段的先决条件是要由 DFM 搜寻到的 CI 及关系的**蓝图**，其中还包括要搜寻到的属性数据。有关详细信息，请参阅 “适配器开发与编写概述”（第 20 页）。

本节包括以下主题：

修改现有适配器

当存在现成适配器或现场适配器时，可以修改现有适配器，但是：

- 它不会搜寻所需的特定属性
- 不会搜寻特定类型的目标 (OS)，或搜寻到错误的特定类型目标。
- 不会搜寻或创建特定的关系

如果现有适配器可以执行上述部分作业，但并非全部作业，则首先需要评估现有适配器，并验证这些适配器中是否有任何一个适配器可以完成所需的工作；如果有，则可以修改现有适配器。

此外，需要评估现有现场适配器是否可用。现场适配器是可用的搜寻适配器，但并非现成适配器。您可联系 HP Software 支持，获取现场适配器的当前列表。

编写新适配器

出现下列情况时，需要开发新适配器：

- 如果写入适配器比手动将信息插入到 CMDB（通常为大约 50-100 个 CI 及其关系）的速度要快，或者手动操作无法一次完成该任务。
- 根据需求调整工作量时。
- 没有可用的现成适配器或现场适配器时。
- 结果可以重用时。
- 目标环境或其数据可用时 （无法搜寻未显示的内容）。

模型研究

- 浏览 UCMDB 类模型 （CI 类型管理器），并将**蓝图**中的实体和关系与现有 CIT 匹配。强烈建议您遵循当前模型，以避免版本升级过程复杂化。如果需要扩展该模型，则需要创建新的 CIT，因为升级可能会覆盖现成 CIT。
- 如果当前模型中缺少某些实体、关系或属性，则需要进行创建。因为在每次安装 HP Universal CMDB 时都需要部署这些 CIT，所以最好创建一个包含这些 CIT 的包 （以后还将涵盖与此包相关的所有搜寻、视图和其他项目）。

技术研究

验证 CMDB 确实包含相关 CI 后，下一阶段便是确定如何从相关系统中检索这些数据。

检索数据通常会涉及到使用协议访问应用程序的管理部分、应用程序的实际数据，或与应用程序相关的配置文件或数据库。凡是能够提供系统相关信息的所有数据源都非常有价值。技术研究不但需要全面了解所涉及的系统，有时也需要创造力。

对于自行开发的应用程序，向应用程序所有者进行问卷调查可能会有所帮助。在此问卷表中，所有者需要列出应用程序中可以提供蓝图和业务价值所需信息的所有方面。这些信息包括 （但不限于）管理数据库、配置文件、日志文件、管理界面、管理程序、 Web 服务、消息或事件发送等。

对于现成的产品，需要关注文档、论坛或产品支持，并查找管理指南、插件和集成指南、管理指南等。如果管理界面中仍然缺失数据，请阅读应用程序的配置文件、数据表项、日志文件、NT 事件日期，以及用于控制正确操作的其他应用程序指导信息。

有关选择数据访问方式的准则

相关性：选择提供大部分数据的源或源组合。如果单个源提供大部分信息，而剩余的信息较分散或难于访问，则需比较剩余信息的工作量和风险，评估这些信息的价值。如果其价值或成本与投入的工作量不匹配，则有时需要决定减少蓝图的数据量。如果值或成本不担保投入的工作量，则有时需要决定减少蓝图。

重用：如果 HP Universal CMDB 已经包括特定连接协议支持，则使用该协议合情合理。因为这意味着 DFM 框架能够提供就绪的客户端和配置进行连接。否则，您可能需要投入到基础结构开发中。您可以查看当前受 HP Universal CMDB 支持的连接协议：“搜寻”>“设置搜寻 Probe”>“域和 Probe”窗格。有关详细信息，请参阅《HP Universal CMDB 数据流管理指南》中的“域和 Probe 窗格”。

您可以通过向模型中添加新 CI 来添加新协议。有关详细信息，请联系 HP Software 支持。

注意：要访问 Windows 注册表数据，可以使用 WMI 或 NTCmd。

安全：访问信息通常需要凭据（用户名、密码），在 CMDB 中输入凭据后，这些凭据在整个产品中都会受到安全保护。如果有可能，并且增加安全性不会与已设置的其他原则相冲突时，则请选择敏感性最低的凭据或协议（但仍然可以满足访问需求）。例如，如果既可通过 JMX（有限的标准管理界面），又可以通过 Telnet 访问信息，则建议使用 JMX，因为 JMX 只提供有限的访问权限，通常不提供对基础平台的访问权限。

支持：某些管理界面可能包括更高级的功能。例如，问题查询（SQL、WMI）可能比导航信息树或构建正则表达式进行解析更容易。

开发人员受众：最终负责开发适配器的人员可能倾向于掌握某种技术。如果两种技术以相同的其他方面成本提供几乎相同的信息，也需要考虑这一问题。

总结

本阶段将生成一个文档，描述访问方式以及可从各种方式提取的相关信息。本文档还将包含从各个源到各个相关蓝图数据的映射。

每种访问方式都将按照上述说明进行标记。最后，对于要搜寻的源以及需要从各个源提取到蓝图模型中的信息（此时可能已经映射到相应的 UCMDB 模型），您需要制定相应的计划。

开发集成内容

在创建新集成之前，必须了解集成的要求：

- 集成是否需要将数据复制到 CMDB 中？数据是否按历史记录跟踪？源是否可靠？

  需要**填充**。

➤ 集成联合数据是否为视图和 TQL 查询的实时数据？数据更改的准确性是否重要？数据量是否太大而无法复制到 CMDB 中，而所请求的数据量通常又太小？

需要**联合**。

➤ 集成是否需要将数据推送到远程数据源？

需要**数据推送**。

注意：“联合”与 “填充”流可能需要配置相同的集成，从而达到最大灵活性。

有关不同类型的集成的详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “集成工作室”。

可使用四个不同的选项创建集成适配器：

➤ Jython 适配器

  ➤ 经典搜寻模式

  ➤ 在 Jython 中写入

  ➤ 用于填充

  有关详细信息，请参阅 “开发 Jython 适配器”（第 63 页）。

➤ Java 适配器

  ➤ 在 Federation SDK Framework 中实现某个适配器接口的适配器。

  ➤ 可用于一个或多个 “联合”、“填充”或 “数据推送”（具体取决于所需的实施措施）。

  ➤ 采用 Java 从零编写，允许编写可连接到任何源或目标的代码。

  ➤ 适用于连接到单个数据源或目标的作业。

  有关详细信息，请参阅 “开发 Java 适配器”（第 217 页）。

- 常规 DB 适配器
  - 基于 Java 适配器并使用 Federation SDK Framework 的抽象适配器。
  - 允许创建连接到外部数据库的适配器。
  - 支持 “联合”和 “填充”（使用为支持更改而实施的 Java 插件）。
  - 较易于定义，因为它主要基于 XML 文件和属性配置文件。
  - 主配置基于 UCMDB 类和数据库列之间映射的 **orm.xml** 文件。
  - 适用于连接到单个数据源的作业。

  有关详细信息，请参阅 “开发常规数据库适配器”（第 125 页）。
- 常规推送适配器
  - 基于 Java 适配器 （即 Federation SDK Framework）和 Jython 适配器的抽象适配器。
  - 允许创建将数据推送到远程目标的适配器。
  - 较易于定义，因为只需要定义 UCMDB 类和 XML 以及将数据推送到目标的 Jython 脚本之间的映射。
  - 适用于连接到单个数据目标的作业。
  - 用于数据推送。

  有关详细信息，请参阅 “开发推送适配器”（第 261 页）。

下表显示了各个适配器的功能：

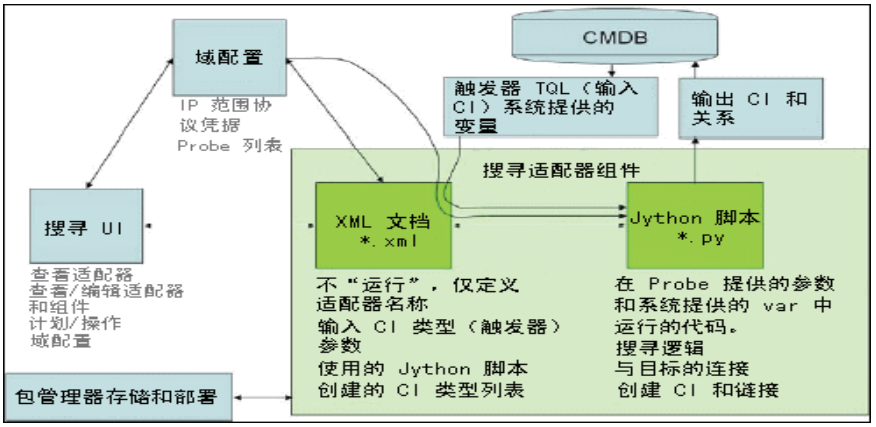
流 / 适配器	Jython 适配器	Java 适配器	GDB 适配器	推送适配器
填充	X	X	X	
联合		X	X	
数据推送		X		X

开发搜寻内容

本节包括：

搜寻适配器和相关组件

下图显示了适配器的组件，以及与适配器交互以执行搜寻的组件。绿色的组件是实际适配器，蓝色的组件是与适配器交互的组件。



请注意，适配器至少要有两个文件：XML 文档和 Jython 脚本。搜寻框架包括输入 CI、凭据和用户提供的库，均在运行时呈现给适配器。上述两种搜寻适配器组件均由数据流管理管理，它们通过操作过程存储在 CMDB 中；尽管它们的外部包仍存在，但在操作时不会涉及。包管理器会保留新搜寻和集成内容功能。

适配器的输入 CI 由 TQL 提供，并以系统提供的变量形式显示在适配器脚本中。适配器参数也可以作为目标数据提供，因此，您可以根据适配器的特定函数来配置适配器的操作。

DFM 应用程序可用于创建和测试新适配器。在适配器编写期间，您可以使用“搜寻控制面板”、“适配器管理”和 “数据流 Probe 设置”页面。

适配器以包的形式进行存储和传输。包管理器应用程序和 JMX 控制台可用于从新创建的适配器创建包，并在新系统上部署适配器。

分离适配器

从技术上而言，可以在单个适配器中定义整个搜寻过程，但是优秀的设计方案要求将复杂的系统分为更简单、更易于管理的组件。

下面列出了用于划分适配器过程的准则和最佳实践：

- 搜寻过程应当分阶段进行。每个阶段都通过映射系统某个区域或某层的适配器表示。适配器会依赖要搜寻过程的上一阶段或上一层来继续对系统进行搜寻。例如，适配器 A 由应用程序服务器 TQL 结果触发，并映射应用程序服务器层。作为此映射的一部分，JDBC 连接组件将会得到映射。适配器 B 将 JDBC 连接组件注册为触发器 TQL，然后使用适配器 A 的结果访问数据库层（例如，通过 JDBC URL 属性），并映射数据库层。
- **两个阶段的连接范例**：大部分系统需要凭据才能访问系统数据。这意味着，需要用户 / 密码组合才能尝试登录这些系统。DFM 管理员以安全的方式向系统提供凭据信息，并能提供多个具有优先顺序的登录凭据，这称为**协议字典**。如果系统无法访问 （无论原因如何），则无需继续执行搜寻。如果连接成功，则需要采用某种方式指出成功使用的凭据集，以继续进行搜寻访问。

在上述两个阶段中，如果出现以下情况，两个适配器将会分离：

- **连接适配器**：该适配器将接受初始触发器并查询该触发器上是否存在远程代理。通过使用“协议字典”中与此代理类型匹配的所有条目可以实现此目的。如果操作成功，则此适配器将提供一个远程代理 CI（SNMP、WMI 等），并指出“协议字典”中将来可用于连接的相应条目。此代理 CI 即成为内容适配器的触发器的一部分。
- **内容适配器**：此适配器的前提条件是成功连接前一个适配器（前提条件由 TQL 指定）。这些类型的适配器不再需要检查全部“协议字典”，因为它们可以从远程代理 CI 获取正确的凭据并使用这些凭据登录搜寻到的系统。

- 不同的计划考虑事项也可以影响搜寻的划分。例如，系统可能只在关闭期间进行查询，所以即使能够将适配器加入到搜寻其他系统的同一适配器，不同的计划仍然表明您需要创建两个适配器。
- 通过搜寻不同的管理接口或技术来搜寻同一系统时，也将使用不同的适配器。这样可以为各个系统或组织激活合适的访问方式。例如，某些组织可通过 WMI 访问计算机，但是计算机上并未安装 SNMP 代理。

任务

实施搜寻适配器

DFM 任务的目标包括访问远程 （或本地）系统、将提取的数据模拟为 CI，以及将 CI 保存到 CMDB。该任务包括以下步骤：

1 DFM 适配器。

通过选择属于适配器的脚本，可以配置包含上下文、参数和结果类型的适配器文件。有关详细信息，请参阅下一节。

2 搜寻作业。

使用计划信息和触发器 TQL 配置作业。有关详细信息，请参阅“步骤 2：将作业分配到适配器”（第 50 页）。

3 搜寻代码。

您可以编辑适配器文件中的 Jython 或 Java 代码，以及涉及 DFM 框架的代码。有关详细信息，请参阅“步骤 3：创建 Jython 代码”（第 52 页）。

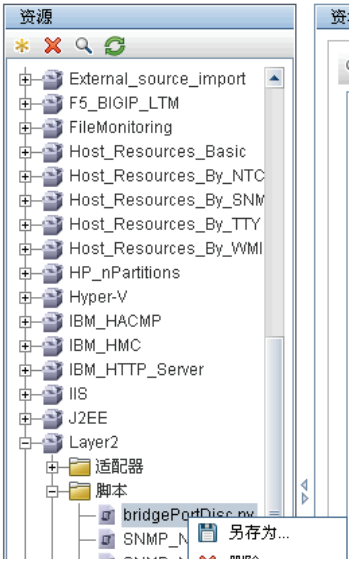
要编写新适配器，需要创建所有上述组件，其中每一个组件都与上一步中的组件自动捆绑。例如，创建作业并选择相关适配器后，适配器文件将与作业绑定。

适配器代码

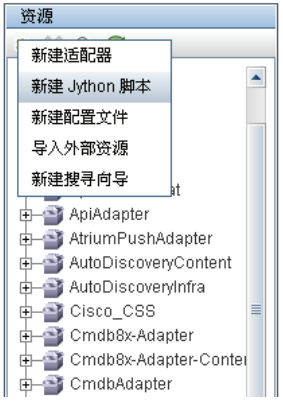
连接到远程系统、查询数据以及作为 CMDB 数据进行映射的实际实施过程由 Jython 代码执行。例如，代码中包含用于连接到数据库以及从数据库提取数据的逻辑。在这种情况下，代码将接收 JDBC URL、用户名、密码、端口等。这些参数特定于应答 TQL 查询的各数据库实例。您可以在适配器（在触发器 CI 数据中）中定义这些变量，而且变量的详细信息将在作业运行时传递到代码，以便执行。

适配器可以通过 Java 类名称或 Jython 脚本名称引用此代码。本节中，我们将讨论以 Jython 脚本的形式编写 DFM 代码。

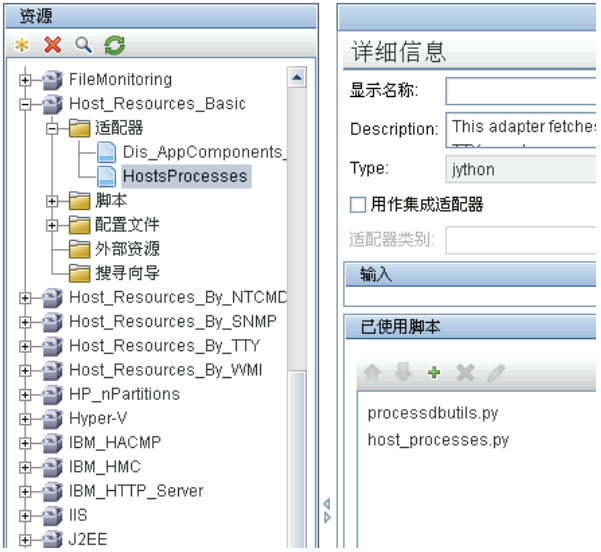
一个适配器可包含一系列用于运行搜寻过程的脚本。创建新适配器时，通常会创建新脚本并将其分配到适配器。新脚本包括基本模板，但是您可以通过右键单击其他脚本并选择 “另存为”将其另存为特定模板：



有关编写新 Jython 脚本的详细信息，请参阅“步骤 3：创建 Jython 代码”（第 52 页）。可通过 “资源”窗格添加脚本：



将按照脚本在适配器中的定义顺序逐个运行脚本：



注意：必须指定一个脚本，即使该脚本仅由其他脚本用作库。因此，库脚本必须在被脚本使用之前进行定义。在此示例中， processdbutils.py 脚本是由上一个 host_processes.py 脚本使用的库。由于缺少 DiscoveryMain() 函数，库将与常规可运行的脚本不同。

步骤 1：创建适配器

可以将适配器视为一个函数的定义。此函数可以指定输入定义、对输入内容运行逻辑、定义输出，以及提供结果。

各适配器将指定输入和输出：输入和输出是在适配器中专门定义的触发器 CI。适配器从输入触发器 CI 提取数据，并将这些数据以参数的形式传递到代码。（有时，相关 CI 中的数据也会传递到代码。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的“相关 CI 窗口”。）适配器的代码是常规代码，与传递到该代码的特定输入触发器 CI 参数无关。

有关输入组件的详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “触发器 CI 和触发器查询”。

本节包括以下主题：

定义适配器输入（触发器 CIT 和输入查询）

您可以使用触发器 CIT 和输入查询组件，来定义作为适配器输入的特定 CI：

- 触发器 CIT 可以定义用作适配器输入的 CIT。例如，对于将要搜寻 IP 的适配器，输入 CIT 为网络。
- 输入查询是一个可编辑的常规查询，可用于定义 CMDB 查询。输入查询可定义对 CIT 的其他约束（例如，当任务需要 hostID 或 application_ip 属性时），并能根据适配器的需要定义更多的 CI 数据。

  如果适配器需要与触发器 CI 相关的其他 CI 信息，则可以向输入 TQL 中添加其他节点。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的“**输入查询定义示例**”（第 43 页）和“向 TQL 查询添加查询节点和关系”。
- 触发器 CI 数据中包含有关触发器 CI 的全部所需信息，以及输入 TQL 中定义的其他节点的信息。DFM 可以使用变量从 CI 检索数据。当任务下载到 Probe 后，触发器 CI 数据变量将替换为真实 CI 实例的属性中所存在的实际值。

触发器 CIT 定义的示例：

在此示例中，触发器 CIT 将定义适配器中允许的 IP CI。

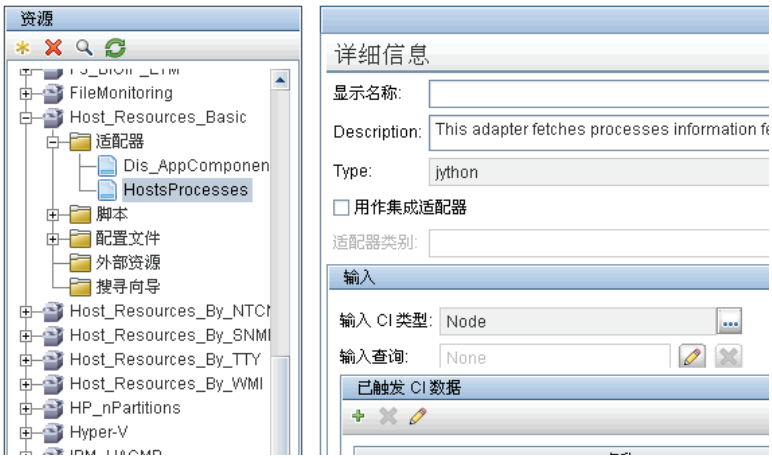
1 访问 “数据流管理” > “适配器管理”。选择 HostProcesses 适配器 （“包” > “Host_Resources_Basic” > “适配器” > “HostProcesses”）。

2 查找 “输入 CI 类型” 框。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “触发的 CI 数据”。

3 单击此按钮可打开 “选择搜寻到的类” 对话框。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “选择搜寻的类对话框”。

4 选择 CIT。

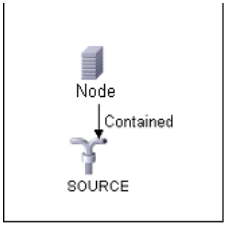
在本示例中，适配器中允许 IP CI （主机）：



输入查询定义示例

在此示例中，输入 TQL 查询指定 IP CI（在之前的示例中配置为触发器 CIT）必须连接到 Host CI。

1. 访问 “数据流管理” > “适配器管理”。查找 “输入 TQL” 框。单击 “编辑” 按钮可打开 “输入 TQL 编辑器”。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “输入查询编辑器”。
2. 在 “输入 TQL 编辑器” 中，将触发器 CI 节点命名为 **SOURCE**：右键单击该节点，并选择 “节点属性”。在 “元素名称” 框中，将名称更改为 **SOURCE**。
3. 向 IP CI 添加 Host CI 和 Contains 关系。有关使用输入 TQL 编辑器的详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “输入查询编辑器”。



IP CI 将连接到 **HOST** CI。输入 TQL 包含两个节点（**HOST** 和 **IP**），这两个节点之间由一个链接连接。 **IP** CI 命名为 **SOURCE**。

关于向输入 TQL 查询中添加变量的示例：

在本示例中，您可以向输入 TQL 查询中添加在上一示例中创建的 DIRECTORY 和 CONFIGURATION_FILE 变量。这些变量可用于定义必须搜寻的内容，即可用于查找主机上链接到需要搜寻的 IP 的配置文件。

1 显示在上一示例中创建的输入 TQL。

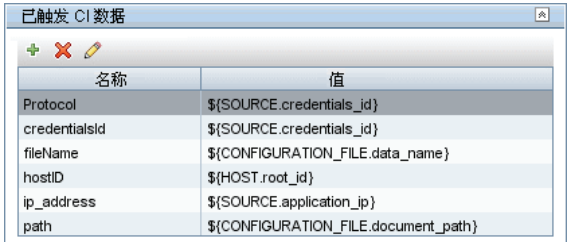
访问 “数据流管理” > “适配器管理”。查找 “已触发 CI 数据” 窗格。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “触发的 CI 数据”。

2 向输入 TQL 中添加变量。有关详细信息，请访问 “数据流管理” > “适配器管理”。查找 “已触发 CI 数据” 窗格。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中 “触发的 CI 数据” 的 “变量” 字段。

将变量替换为实际数据的示例：

在本示例中，变量可以将 IP CI 数据替换为系统中真实 IP CI 实例中存在的实际值。

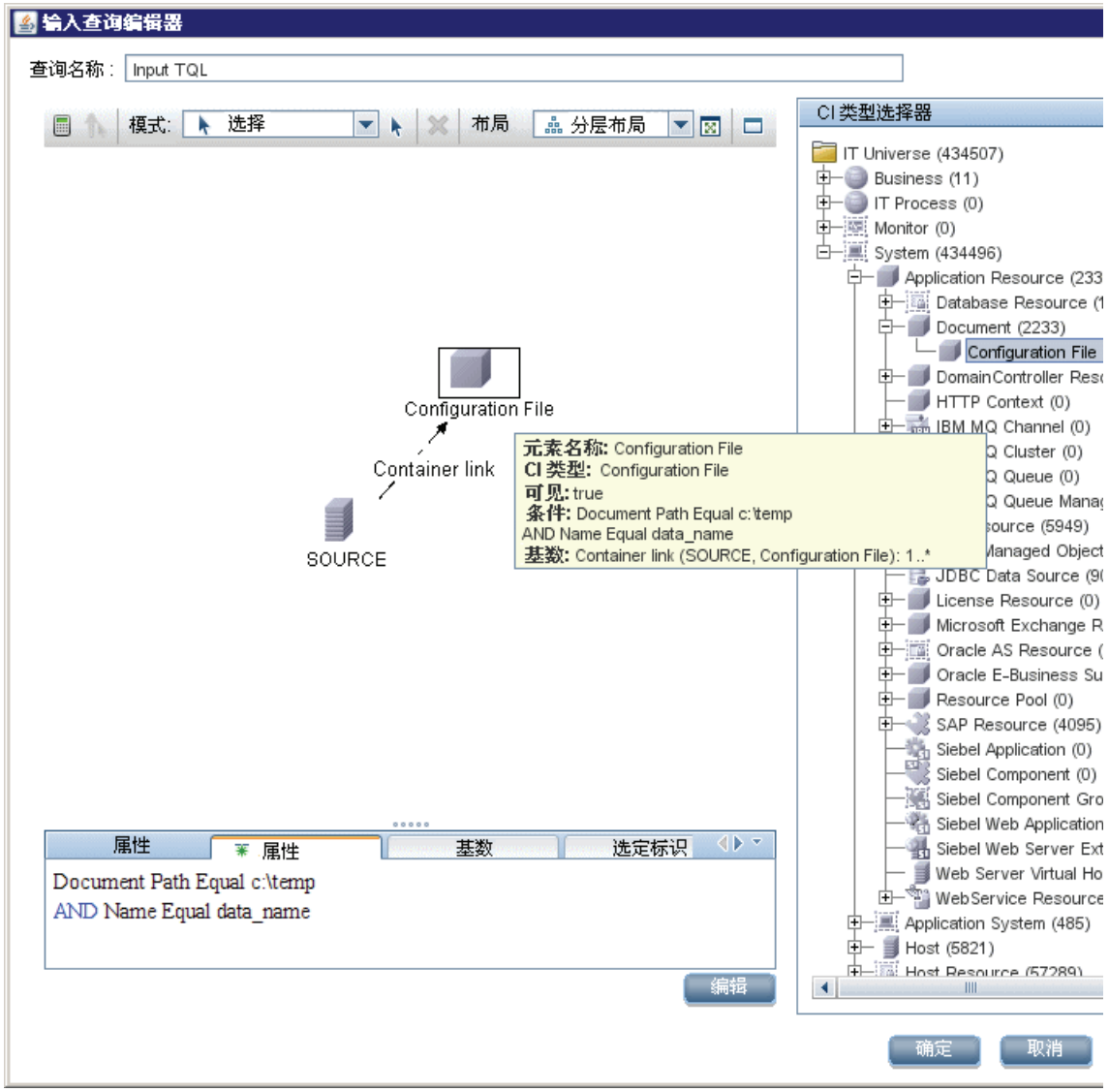
IP CI 的已触发 CI 数据包括 fileName 变量。使用此变量可以将输入 TQL 中的 CONFIGURATION_FILE 节点替换为主机上配置文件的实际值：



触发器 CI 数据将上载到 Probe，同时所有的变量将替换为实际值。适配器脚本中包含一个命令，该命令将使用 DFM 框架检索已定义变量的实际值：

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

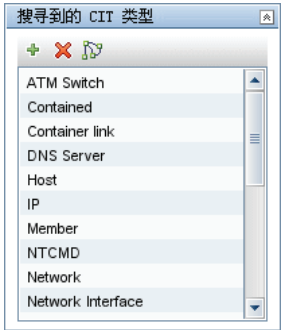
fileName 和 path 变量将使用配置文件节点的 data_name 和 document_path 属性 （在输入 TQL 中定义的。详见上一示例）。



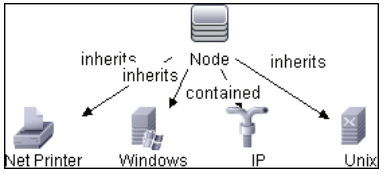
Protocol、credentialsId 和 ip_address 变量使用 root_class、credentials_id 和 application_ip 属性：

定义适配器输出

适配器的输出指的是搜寻到的一组 CI（“数据流管理”>“适配器管理”>“适配器定义”选项卡 >**“搜寻到的 CIT 类型”**）以及这些 CI 之间的链接：



您还可以以拓扑图形式（即各组件以及组件间的链接方式）查看 CIT（单击“以图方式查看搜寻到的 CI 类型”按钮）：



搜寻到的 CI 将由 DFM 代码（即 Jython 脚本）以 UCMDB 的 ObjectStateHolderVector 格式返回。有关详细信息，请参阅 “Jython 脚本生成的结果”（第 71 页）。

适配器输出的示例：

在本示例中，您可以定义要包含在 IP CI 输出中的 CIT。

1. 访问 “数据流管理” > “适配器管理”。
2. 在 “资源” 窗格中，选择 “Network” > “适配器” > “NSLOOKUP_on_Probe”。
3. 在 “适配器定义” 选项卡中，查找 “搜寻到的 CIT 类型” 窗格。
4. 此时，将会列出要包含在适配器输出中的 CIT。在列表中添加或删除 CIT。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 ““搜寻到的 CIT” 窗格”。

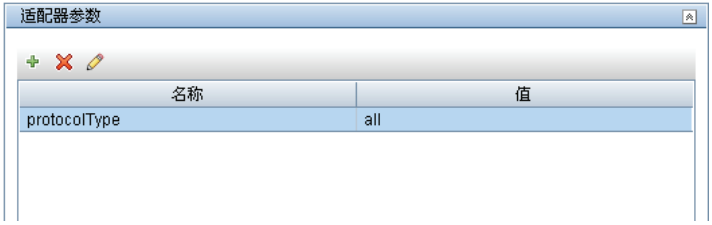
覆盖适配器参数

如果要为多个作业配置适配器，则可以覆盖适配器参数。例如，适配器 SQL_NET_Dis_Connection 可由“通过 SQL 进行 MSSQL 连接”和“通过 SQL 进行 Oracle 连接”作业同时使用。

覆盖适配器参数的示例：

本示例说明了如何覆盖适配器参数，以便使用适配器搜寻 Microsoft SQL Server 和 Oracle 数据库。

1. 访问 “数据流管理” > “适配器管理”。
2. 在 “资源”窗格中，选择 “Database_Basic” > “适配器” > “SQL_NET_Dis_Connection”。
3. 在 “适配器定义”选项卡中，查找 “搜寻模式参数”窗格。 protocolType 参数值为 **All**：



4. 右键单击 **SQL_NET_Dis_Connection_MsSql** 适配器，并选择 “转到搜寻作业” > “MSSQL Connection by SQL”。
5. 显示 “属性”选项卡。查找 “参数”窗格：

All 值将被 MicrosoftSQLServer 值覆盖。

注意：“Oracle Connection by SQL”作业包括相同的参数，但是值将被 Oracle 值覆盖。

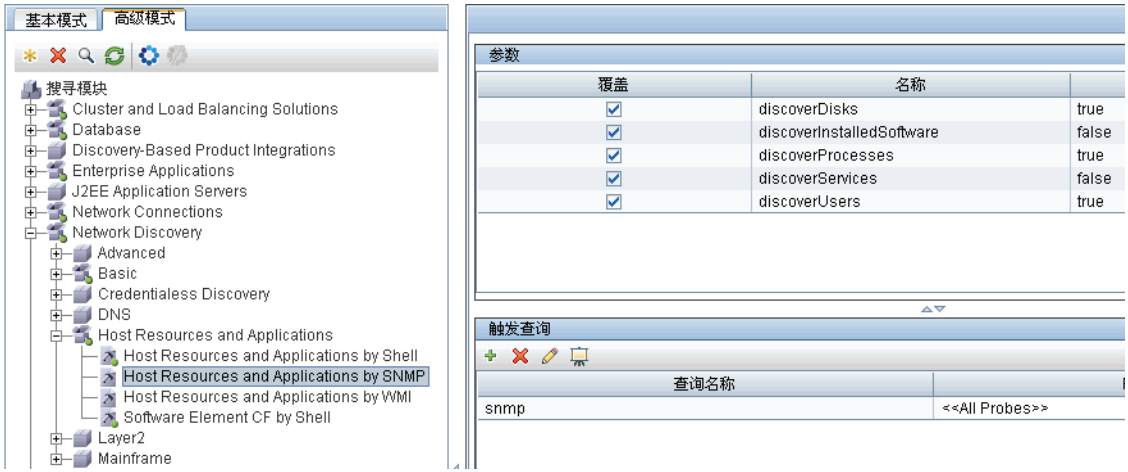
有关添加、删除或编辑参数的详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “ “适配器参数”窗格”。

DFM 将根据此参数开始查找 Microsoft SQL Server 实例。

步骤 2：将作业分配到适配器

每个适配器均具有一个或多个可定义执行策略的关联作业。通过这些作业，不但可以根据不同的已触发 CI 集采用不同的方式计划同一个适配器，还可以为各个已触发 CI 集提供不同的参数。

这些作业将显示在 “搜寻模块”树中，这是用户激活的实体。



触发器 TQL

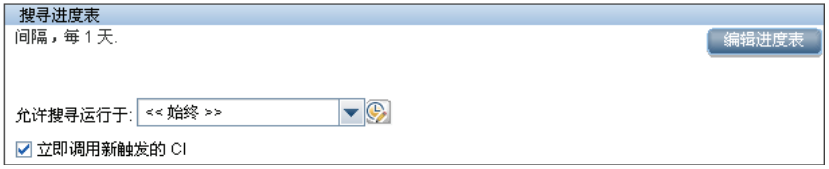
每个作业均与多个触发器 TQL 关联。这些触发器 TQL 发布的结果将用作此作业的适配器的输入触发器 CI。

触发器 TQL 可以向输入 TQL 添加约束。例如，如果输入 TQL 的结果是连接到 SNMP 的 IP，则触发器 TQL 的结果可以是连接到 SNMP 的 IP，范围为 195.0.0.0-195.0.0.10。

注意：触发器 TQL 必须与输入 TQL 引用相同的对象。例如，如果输入 TQL 查询运行 SNMP 的 IP，则不得将触发器 TQL（针对同一个作业）定义为查询连接到主机的 IP，因为根据输入 TQL 的要求，有些 IP 可能未连接到 SNMP 对象。

计划

Probe 的计划信息可以指定在触发器 CI 上运行代码的时间。选中“立即调用新触发的 CI”复选框后，代码也会在传递到 Probe 时在各触发器 CI 上运行一次，而与以后的计划设置无关。



对于各作业的各个作业事件， Probe 将在该作业累积的所有触发器 CI 上运行代码。有关详细信息，请参阅《HP Universal CMDB 数据流管理指南》中的“‘搜寻计划程序”对话框”。

参数

配置作业时可以覆盖适配器参数。有关详细信息，请参阅 “覆盖适配器参数”（第 49 页）。

步骤 3：创建 Jython 代码

HP Universal CMDB 使用 Jython 脚本进行适配器编写。例如，SNMP_NET_Dis_Connection 适配器使用 SNMP_Connection.py 脚本，尝试通过 SNMP 连接到计算机。 Jython 是一种基于 Python 并支持 Java 的语言。

有关如何使用 Jython 的详细信息，可以参考以下网站：

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

有关详细信息，请参阅 “创建 Jython 代码”（第 65 页）。

2

搜寻内容迁移规则

本章包括：

概念

任务

参考

概念

搜寻内容迁移规则概述

在 HP Universal CMDB 9.0x 版中，已显著改进了数据模型，从而需要对之前的搜寻和依赖关系映射 (DDM) 内容代码执行关联更改。因此， DDM 内容的某些核心机制已发生更改。所以，必须升级为 UCMDB 9.0x 之前的版本开发的内容，以便与 9.0x 数据模型 （BDM：BTO 数据模型）兼容。本节将说明有关修改 DDM 内容，并使其与 BDM 兼容的过程。

有关升级 HP Universal CMDB 的详细信息，请参阅《HP Universal CMDB 部署指南》 PDF 文档中的 “将 HP Universal CMDB 从版本 8.0x 升级到版本 9.0x”。

9.0x 版本的新增基础结构功能

注意：有关访问 BDM 联机文档的详细信息，请参阅 “联机访问 BTO 数据模型文档”（第 60 页）。

本节包括：

BTO 数据模型 (BDM)

- 有关 BTO 数据模型 (BDM) 的详细信息，请参阅 《Conceptual Data Model》文档。此文档概述了要建模的概念以及模型范围。此概念性数据模型是了解建模域语义的起点。
- 有关 BDM 类的详细信息，请参阅 《HP Software BTO Data Model Reference》文档。本文档涵盖所有 BDM 类，包括类描述信息和参数、限定符和层次结构信息。

UCMDB 8.0x 类模型与 UCMDB 9.0x 数据模型之间的差异。

UCMDB 8.0x 版类模型与 BDM 之间的更改已下载到以下搜寻配置文件中的 Probe 中：**C:\hp\UCMDB\DataFlowProbe\runtime\probeManager\discoveryConfigFiles\flat-class-model-changes.xml**。

bdm_changes.xml。此 XML 文件包含对类名称、属性名称、已删除类、属性和限定符等内容进行的更改的相关信息。

- 有关 UCMDB 8.0x 版的类模型和 BDM 之间映射的详细信息，请参阅 《Mapping of UCMDB 9.0x (BTO Data Model) to UCMDB 8.0x ClassModel》文档。
- 有关 8.0x 版和 9.0x 版之间类模型的不同之处的详细信息，请参阅 《 UCMDB Class Model Changes Report》文档。

新 CIT 标识机制

在 UCMDB 9.0x 之前的版本中，使用键属性来标识 CI。在 UCMDB 9.0x 版中，此概念已推广，而且现在将在名为 “调和引擎” 的服务器组件中完成标识工作。调和引擎能够根据 DDA （数据定义算法）逻辑规则来标识 CI。

此新机制主要用于相关拓扑对其标识而言十分重要的 CIT，例如，先前版本中的“Node CIT - Host”将使用其名称和相关拓扑（例如 IP Address 和 Interface CIT）来标识。而另一些 CIT 仍然使用键属性进行标识。对于这些 CIT，未定义 DDA 规则。

有关调和引擎的详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “调和概述”。

运行软件机制

8.0x 版的 **Software Element** CI 在 9.0x 版 BDM 中称为 **Running Software**。在 9.0x 中，将使用 DDA 规则而不是键属性来标识此 CIT。

假设您添加了一个派生自 **Running Software** CIT 的自定义 CIT。在先前的版本中，将使用其键属性来标识此自定义 CIT。但是，在 9.0x 版中，该 CIT 将由继承的 DDA 规则来标识，因此将忽略已定义的键属性。

因此，如果要添加一个派生的 CIT，请考虑以下事项：

- 要使用与所有 Running Software CIT 相同的 DDA 规则来标识新 CIT，请保留当前配置。
- 要使用键属性来标识新 CIT，请创建一个新的 DDA 规则，用于按照键属性定义标识。下面显示了一个为 **object** CIT 定义的此类 DDA 规则示例：

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

Probe 端标识

DDM_ID_ATTRIBUTE。数据流 Probe 9.0x 版仅会按照 CI 的键属性 （即 **ID_ATTRIBUTE**）来标识 CI。如果 CIT 包含 DDA 规则（即调和规则），则此 CIT 可能不包含键属性。在这种情况下，将使用 **DDM_ID_ATTRIBUTE** 限定符来标记 CIT 主要属性。因此，为了标识 CI， Probe 会将 **DDM_ID_ATTRIBUTE** 视为 **ID_ATTRIBUTE** 限定符。

DDM_REQUIRED_TOPOLOGY。特定 CIT 的 DDA 规则取决于同一批中报告的不同 CI 以及已检查的 CI。例如， **J2EE Domain** CIT 标识将由域名属性以及通过成员关系连接的 **J2EE Application Server** CIT 来实现。

要确保报告所有需要的 CI 时带有已检查的 CI，则应使用 **DDM_REQUIRED_TOPOLOGY** 限定符标记每个已检查的 CI，该限定符中包含一个用于指定所需链接类型的数据项。例如，在上述示例中， **J2EE Domain** CIT 标有 **DDM_REQUIRED_TOPOLOGY** 限定符和 **member** 链接数据项，因此，当搜寻程序报告 J2EE Domain 时，也会报告服务器。

有关限定符的详细信息，请参阅 《HP Universal CMDB 建模指南》中的 ““限定符”页面”。

转换层

为了确保向下兼容，Probe 上的 9.0x 版中引入了一种新的转换机制。这种机制能够在运行时将 8.0x 版的拓扑转换为 9.0x 版，同时还支持 Probe 继续运行任务，如 Jython 脚本，以报告与 8.0x 版兼容的拓扑。

新转换机制使用 **bdm_changes.xml** 文件中保存的数据，并且执行所需的更改（更改类和属性名称、删除属性、更改层次结构等），使 8.0x 版拓扑与 BMD 兼容。另外，在 Probe 执行的任务报告拓扑的同时， UCMDB 服务器将独立地接收到与 BDM 兼容的拓扑。

包迁移实用程序

UCMDB 9.0x 包含一个外部包迁移实用程序，它支持内容开发人员将内容包从 8.0x 类模型转换为 9.0x 数据模型。该包迁移实用程序能够逐个子系统地转换包资源，使这些资源与新类模型相兼容。将根据 **bdm_changes.xml** 文件中存储的数据转换 CIT 定义、查询、作业、适配器和模块。因此，它们可以由 UCMDB 9.0x 服务器部署和使用。

有关详细信息，请参阅《HP Universal CMDB 部署指南》PDF 文档中的“将包从版本 8.04 升级到版本 9.02”。

包迁移实用程序限制

- 包迁移实用程序不会升级 Jython 脚本。为了支持专用于 UCMDB 8.0x 版类模型的脚本，UCMDB 9.0x 中引入了一个新的**转换层**模块。有关详细信息，请参阅 “转换层”（第 57 页）。
- 包迁移实用程序不能升级集成类型的搜寻适配器，因此需手动升级这类适配器。
- “层 2 拓扑”搜寻作业（及其对应的资源，如搜寻适配器、TQL 等）已发生重大更改，并且已被包迁移实用程序删除，将不会进行升级。

交叉数据模型开发规则

以下规则适用于 8.0x 版和 9.0x 版。

搜寻脚本 API 库

搜寻脚本 API 库可完全向下兼容，因此支持所有 8.0x 版的库和 API。有关详细信息，请参阅 “Jython 库和实用程序”（第 112 页）。

9.0x API 包括更多的元素和方法。例如， Jython 脚本现在将报告错误代码 （整数）而不是字符串错误消息，从而支持本地化的搜寻错误消息。有关详细信息，请参阅 “错误编写约定”（第 119 页）。

实施提示

- 使用**建模**模块创建 **Running Software** CIT，或提供了相关方法的任何子级。
- 使用 **HostBuilder** 创建 **Node** 类型的 CIT。
- 使用 **modeling.createOshByCmdbIdString** 按 ID 恢复 OSH。
- 将 **shellutils** 模块的 **ShellUtils** 实例用于所有基于 shell 的连接。
- 使用内置机制检索 UCMDB 版本：logger.Version().getVersion(framework)。例如，如果仅为 UCMDB 9.0x 或更高版本添加了一个附加属性 application_ip：

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

- 使用 **wmiutils** 创建基于 WMI 的搜寻。
- 使用 **snmputils** 创建基于 SNMP 的搜寻。

任务

联机访问 BTO 数据模型文档

要访问 BDM 文档，请执行以下操作：

1. 登录 HP Universal CMDB。
2. 单击 “帮助” > “UCMDB 帮助”。
3. 在主页上单击 “应用程序” 下方的 “建模” 链接，访问 “建模” 门户。
4. 单击 “数据模型” 选项卡。

参考

疑难解答和限制

- 默认情况下，不会将 **ip_address** 值传递给模式，必须将其作为触发器 CI 数据明确添加到模式中。
- 如果非现成的 Jython 脚本需要类路径中的外部 Jar 或资源，应将该脚本放在名为 **discoveryResources** 的子文件夹下的相关包中。
- 在处理**列表**类型的属性时（如 **StringVector** 和 **IntegerVector**（从 **BaseVector** 继承）），不能同时对同一个列表对象执行**添加元素**和**删除元素**操作。

3

开发 Jython 适配器

本章包括：

概念

任务

参考

概念

HP 数据流管理 API 参考

有关可用 API 的完整文档，请参阅 《HP Universal CMDB Data Flow Management API Reference》。这些文件位于以下文件夹中：

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

任务

创建 Jython 代码

HP Universal CMDB 使用 Jython 脚本进行适配器编写。例如，SNMP_NET_Dis_Connection 适配器使用 SNMP_Connection.py 脚本尝试连接使用 SNMP 的计算机。 Jython 是一种基于 Python 并支持 Java 的语言。

有关如何使用 Jython 的详细信息，可以参考以下网站：

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

下一节描述了如何在 DFM 框架内实际编写 Jython 代码。本节专门介绍了 Jython 脚本与其调用的框架之间的联系点，还介绍了可能会用到的任何 Jython 库和实用工具。

注意：

- 为 DFM 编写的脚本应当与 Jython 2.1 版兼容。
- 有关可用 API 的完整文档，请参阅 《HP Universal CMDB Data Flow Management API Reference》。

本节包括以下主题：

➤ “查找正确的凭据 （针对连接适配器）”（第 77 页）

➤ “处理 Java 抛出的异常”（第 78 页）

在 Jython 中使用外部 Java JAR 文件

开发新 Jython 脚本时，有时需要使用外部 Java 库 （JAR 文件）或第三方可执行文件作为 Java 实用程序存档文件、连接存档文件 （例如 JDBC 驱动程序 JAR 文件）或可执行文件，例如，**nmap.exe** 用于无凭据搜寻）。

这些资源应当捆绑在**外部资源**文件夹下的数据包中。放在此文件夹中的所有资源均会自动发送到任何连接到 HP Universal CMDB 服务器的 Probe。

此外，启动搜寻时，会将任何 JAR 文件资源加载到 Jython 的类路径中，以便导入和使用该类路径中的所有类。

执行代码

激活某个作业之后，会将具有所需全部信息的任务下载到 Probe。

Probe 使用在该任务中指定的信息开始运行 DFM 代码。

Jython 代码流从脚本中的主索引项开始运行，执行代码以搜寻 CI，并提供已搜寻到的 CI 的矢量结果。

修改现成脚本

修改现成脚本时，请尽量减少对脚本的更改，并将所有必需的方法放在外部脚本中。这样便于更有效地跟踪更改，并且在迁移到更新的 HP Universal CMDB 版本时，不会覆盖您的代码。

例如，某个现成脚本中的以下代码行可调用一个计算 Web 服务器名称（以特定于应用程序的方式）的方法：

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

用于确定应如何计算此名称的更复杂的逻辑包含在外部脚本中：

```
# 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
```

将此外部脚本保存在 “外部资源”文件夹中。有关详细信息，请参阅《HP Universal CMDB 数据流管理指南》中的 ““资源”窗格”。如果将此脚本添加到包中，则可以将此脚本用于其他作业。有关使用包管理器的详细信息，请参阅 《HP Universal CMDB 管理指南》中的 “包管理器”。

在升级过程中，新版本的现成脚本将覆盖您对单行代码所作的更改，因此您需要替换该行。但是，外部脚本不会被覆盖。

Jython 文件的结构

Jython 文件包含以特定顺序排列的三个部分：

1 导入

2 主函数 - DiscoveryMain

3 函数定义 （可选）

以下是一个 Jython 脚本示例：

```
#   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
```

导入

Jython 类分布在分层命名空间中。与先前的版本不同，在 7.0 版或更高版本中没有隐式导入，因此必须显式导入要使用的每个类。（进行此更改是为了提高性能，并且通过不再隐藏必需的详细信息使 Jython 脚本更易于理解。）

➤ 要导入 Jython 脚本，请使用：

```
import logger
```

➤ 要导入 Java 类，请使用：

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

主函数 - DiscoveryMain

每个 Jython 可运行脚本文件均包含一个主函数：DiscoveryMain。

DiscoveryMain 函数是脚本中的主索引项，它是第一个运行的函数。主函数可以调用在脚本中定义的其他函数：

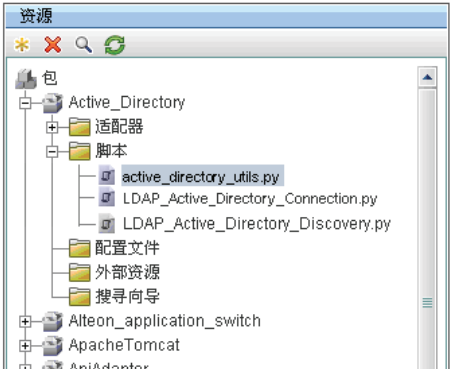
```
def DiscoveryMain(Framework):
```

必须在主函数定义中指定框架参数。此参数由主函数用于检索运行脚本所需的信息（例如有关触发器 CI 和参数的信息），还可用于报告在脚本运行期间发生的错误。

您可以创建不包含任何主方法的 Jython 脚本。可以将此类脚本用作从其他脚本调用的库脚本。

函数定义

每个脚本可以包含多个从主代码调用的附加函数。这种类型的每个函数可以调用当前脚本或其他脚本中的其他函数（使用 import 语句）。请注意，如果要使用另一个脚本，必须将其添加到包的脚本部分中：



关于调用其他函数的函数示例：

在以下示例中，主代码调用 doOSUserOSH(..) 方法，后者调用内部方法 doQueryOSUsers(..)：

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

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

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

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

如果此脚本是与多个适配器相关的全局库，则可以将其添加到 jythonGlobalLibs.xml 配置文件中的脚本列表，而无须添加到每个适配器（**“适配器管理”** > **“资源”窗格** > **“AutoDiscoveryContent”** > **“配置文件”**）。

Jython 脚本生成的结果

每个 Jython 脚本在特定触发器 CI 上运行，以 DiscoveryMain 函数的返回值所返回的结果结束。

脚本结果实际上是一组要在 CMDB 中插入或更新的 CI 和链接。脚本以 ObjectStateHolderVector 的形式返回 CI 与链接。

ObjectStateHolder 类是一种表示在 CMDB 中定义的对象或链接的方式。ObjectStateHolder 对象包含 CIT 名称、属性列表和属性值。ObjectStateHolderVector 是 ObjectStateHolder 实例的矢量。

ObjectStateHolder 语法

本节介绍如何在 UCMDB 模型中包含 DFM 结果。

关于在 CI 上设置属性的示例：

ObjectStateHolder 类描述了 DFM 结果图。每个 CI 和链接 （关系）放置在一个 ObjectStateHolder 类实例中，如以下 Jython 代码示例所示：

```
# 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 行创建了一个 **siebelappserver** 类型的 CI。
- 第 2 行创建了一个名为 **data_name** 的属性，其值为 **sblsvrName**。它是以搜寻到的服务器名称值进行设置的 Jython 变量。
- 第 3 行设置了一个在 CMDB 中更新的非键属性。
- 第 4 行生成了包含关系（结果是图形）。该关系指定此应用程序服务器包含于在一个主机 （范围中的另一个 ObjectStateHolder 类）中。

注意：Jython 脚本报告的每个 CI 必须包含该 CI 所属类型的所有键属性的值。

关系 （链接）示例：

以下链接示例介绍了图形的表示方法：

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

- 第 1 行创建了链接 （也是 ObjectStateHolder 类的一部分。唯一的不同在于 route 是链接 CI 类型）。
- 第 2 行和第 3 行指定了每个链接端的节点。这是通过使用 **end1** 和 **end2** 属性来完成的。必须指定这两个属性，因为它们是每个链接的最小键属性。属性的值是 ObjectStateHolder 实例。有关 End 1 和 End 2 的详细信息，请参阅《HP Universal CMDB 数据流管理指南》中的 “链接”。

警告：链接具有方向。应当验证 End 1 和 End 2 节点在每个端是否对应于有效的 CIT。如果节点无效，则结果对象将无法通过验证，也无法正确地得到报告。有关详细信息，请参阅 《HP Universal CMDB 建模指南》中的 “CI 类型关系”。

矢量示例 （收集 CI）：

在创建含有属性的对象，并将这些对象的端相链接之后，就可以将这些对象分为一组。可以通过将这些对象添加到 ObjectStateHolderVector 实例中来完成此操作，如下所示：

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

有关如何向框架报告此组合结果以便将其发送到 CMDB 服务器的详细信息，请参阅 sendObjects 方法。

一旦在 ObjectStateHolderVector 实例中组合得到了结果图，就必须将其返回到要插入 CMDB 的 DFM 框架中。通过将 ObjectStateHolderVector 实例作为 DiscoveryMain() 函数的结果返回，可以完成此操作。

注意：有关创建常见 CIT 的 **OSH** 的详细信息，请参阅 “Jython 库和实用程序”（第 112 页）中的 **modeling.py**。

框架实例

框架实例是在 Jython 脚本的主函数中提供的唯一参数。这是用于检索在运行脚本时所需的信息（例如，有关触发器 CI 和适配器参数的信息）的接口，还可用于报告在脚本运行期间发生的错误。有关详细信息，请参阅 “HP 数据流管理 API 参考”（第 64 页）。

本节描述了一些最重要的框架用法：

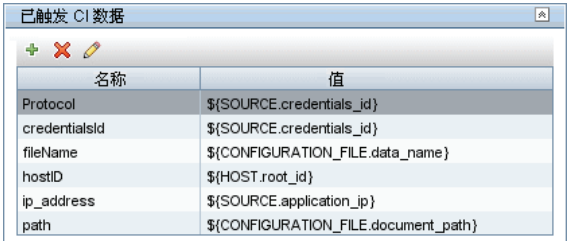
- “Framework.getTriggerCIData(String attributeName)”（第 73 页）
- “Framework.createClient(credentialsId, props)”（第 74 页）
- “Framework.getParameter (String parameterName)”（第 75 页）
- “Framework.reportError(String message) 和 Framework.reportWarning(String message)”（第 76 页）

Framework.getTriggerCIData(String attributeName)

此 API 提供了在适配器和脚本中定义的触发器 CI 数据之间的中间步骤。

凭据信息检索示例：

您请求了以下触发器 CI 数据信息：



要从任务中检索凭据信息，请使用此 API：

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

Framework.createClient(credentialsId, props)

可以通过创建客户端对象并在该客户端上执行命令，来连接到远程计算机。要创建客户端，请检索 ClientFactory 类。 getClientFactory() 方法接收所请求的客户端协议类型。协议常量是在 ClientsConsts 类中定义的。有关凭据和受支持协议的详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “域凭据参考”。

关于为凭据 ID 创建客户端实例的示例：

要为凭据 ID 创建 Client 实例，请使用：

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

现在，可以使用 Client 实例连接到相关的计算机或应用程序。

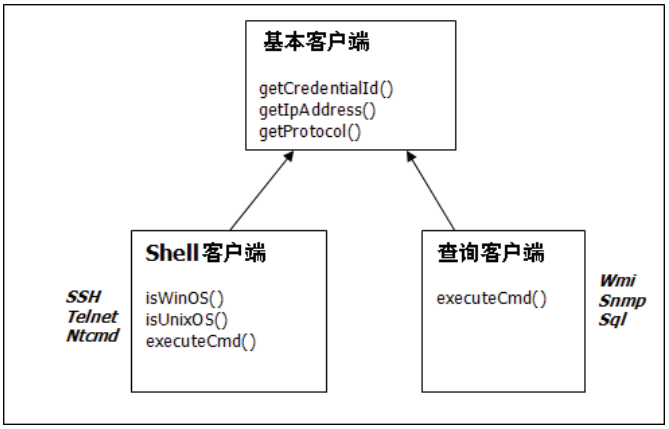
关于创建 WMI 客户端和运行 WMI 查询的示例：

要创建 WMI 客户端并使用此客户端运行 WMI 查询，请使用：

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

注意：要使用 createClient() API，请将以下参数添加到 “已触发 CI 数据”窗格的触发器 CI 数据参数中：**credentialsId = ${SOURCE.credentials_id}**。或者，您也可以在调用以下函数时手动添加凭据 ID：
wmiClient = clientFactory().createClient(credentials_id)。

下图演示了客户端的层次结构以及它们通常支持的 API：



有关客户端及其支持的 API 的详细信息，请参阅《HP Universal CMDB Data Flow Management API Reference》中的 BaseClient、 ShellClient 和 QueryClient。

Framework.getParameter (String parameterName)

除了检索有关触发器 CI 的信息之外，通常还需要检索适配器参数值。例如：

参数

覆盖	名称	值
☑	protocolType	MicrosoftSQLServer

关于检索 protocolType 参数值的示例：

要从 Jython 脚本中检索 protocolType 参数值，请使用以下 API：

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

Framework.reportError(String message) 和 Framework.reportWarning(String message)

在脚本运行期间，可能会发生一些错误 （例如连接失败、硬件问题、超时）。在检测到此类错误时，框架可报告问题。所报告的消息会提交到服务器，并显示给用户。

报告错误和消息示例：

以下示例演示了 reportError （< 错误消息 >） API 的用法：

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

您可以使用 Framework.reportError(String message) API 或 Framework.reportWarning(String message) API 报告问题。这两个 API 之间的区别在于：报告错误时， Probe 是否将含有完整会话参数的通信日志文件保存到文件系统中。如果 Probe 将通信日志文件保存到文件系统，您就可以跟踪会话，并更好地了解错误原因。

有关错误消息的详细信息，请参阅 “错误消息”（第 117 页）。

查找正确的凭据 （针对连接适配器）

适配器在尝试连接到远程系统时，需要尝试所有可能的凭据。创建客户端 （通过 ClientFactory）时需要的其中一个参数是凭据 ID。连接脚本将访问所有可能的凭据集，然后使用 clientFactory.getAvailableProtocols() 方法逐一进行尝试。当一个凭据集成功时，适配器会将此触发器 CI （具有与 IP 匹配的凭据 ID）的主机上的 CI 连接对象报告给 CMDB。后续适配器可以使用此连接对象 CI 直接连接到凭据集，而不需要重新尝试所有可能的凭据。

以下示例显示了如何获取 SNMP 协议的所有条目。请注意，此处的 IP 是从触发器 CI 数据中获取的 (# Get the Trigger CI data values)。

连接脚本将请求所有可能的协议凭据 (# Go over all the protocol credentials)，并且循环尝试它们，直至其中一个成功为止 (resultVector)。有关详细信息，请参阅 “分离适配器”（第 35 页）中的 **“两个阶段的连接范例”** 条目。

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

        def mainFunction(Framework):
resultVector = ObjectStateHolderVector()

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

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

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

        // Query the agent
        ….

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

处理 Java 抛出的异常

某些 Java 类会在失败时抛出异常。建议您捕获并处理异常，否则，异常将导致适配器意外终止运行。

捕获到已知异常时，大多数情况下应将其堆栈跟踪输出到日志，并向 UI 发出相应的消息，例如：

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

如果异常不是致命的异常，而且脚本可以继续执行，则应忽略 reportError() 方法调用，使脚本继续运行。

支持 Jython 适配器中的本地化

多语言环境功能使得 DFM 能够在不同的操作系统 (OS) 语言下工作，并且支持您在运行时进行适当的自定义。

在 Content Pack 3.00 之前，DFM 使用静态指定的编码来处理所有网络目标的输出。然而，此方法不适用于多语言的 IT 网络，要搜寻不同操作系统语言的主机，Probe 管理员必须多次手动运行 DFM，并在每次运行时使用不同的作业参数。这个过程将产生很高的网络负载，更重要的是，它没有使用 DFM 的几个关键功能，例如，计划管理器对触发器 CI 的即时作业调用或者在 UCMDB 中进行的自动数据刷新。

默认情况下，支持以下语言环境：日语、俄语和德语。默认语言环境为英语。

本节包括：

添加新的语言支持

本任务描述如何添加新的语言支持。

本任务包括以下步骤：

1 添加资源数据包 （*.properties 文件）

根据要运行的作业添加资源包。下表列出了 DFM 作业和每个作业使用的资源数据包：

作业	资源数据包的基本名称
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

有关数据包的详细信息，请参阅 “使用资源数据包”（第 85 页）。

2 声明并注册语言对象

要定义新语言，请将以下两行代码添加到 **shellutils.py** 脚本中。该脚本当前包含所有受支持语言的列表。该脚本包含在 AutoDiscoveryContent 包中。要查看该脚本，请访问 “适配器管理”窗口。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 ““适配器管理”窗口”。

a 声明语言，如下所示：

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

有关类语言的详细信息，请参阅“API 参考”（第 86 页）。有关类语言环境对象的详细信息，请参阅 http://java.sun.com/j2se/1.5.0/docs/api/java/util/Locale.html。您既可以使用现有语言环境，也可以定义新语言环境。

b 可通过将该语言添加到以下集合来对其进行注册：

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

更改默认语言

如果无法确定操作系统语言，则会使用默认语言。**shellutils.py** 文件中指定了默认语言。

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

要更改默认语言，请使用其他语言初始化 DEFAULT_LANGUAGE 变量。有关详细信息，请参阅 “添加新的语言支持”（第 79 页）。

确定用于编码的字符集

用于对命令输出进行解码的相应字符集将在运行时确定。多语言解决方案基于以下事实和假设：

1. 可以使用与语言环境设置无关的方式确定操作系统语言，例如，在 Windows 上运行 **chcp** 命令，或者在 Linux 上运行 **locale** 命令。
2. 关系语言编码是众所周知的，因此可以进行静态定义。例如，对于俄语有两个最常用的编码：Cp866 和 Windows-1251。
3. 每种语言都有一个首选字符集，例如，俄语的首选字符集是 Cp866。这意味着大多数命令均会使用此编码进行输出。

4 无法预测将以何种编码提供下一个命令，但可以确定是，该编码必定是特定语言的可能编码之一。例如，在使用俄语语言环境的 Windows 计算机时，系统将以 Cp866 编码提供 **ver** 命令输出，但以 Windows-1251 编码提供 **ipconfig** 命令输出。

5 已知命令可以在其输出中生成已知的关键字。例如，**ipconfig** 命令中将包含经转换的 **IP-Address** 字符串形式。因此，对于英语操作系统，**ipconfig** 命令输出包含 **IP-Address**；对于俄语操作系统，包含 IP-Адрес；对于德语操作系统，则包含 **IP-Adresse**，等等。

一旦搜寻到用于生成命令输出的语言 (# 1)，就会将可能的字符集范围限定为一个或两个 (# 2)。此外，此输出中包含的关键字也是已知的 (# 5)。

因此，解决方法是通过在结果中搜索关键字，以便使用一种可能的编码对命令输出进行解码。如果发现了关键字，则会将当前字符集视为正确的字符集。

定义使用本地化数据运行的新作业

本任务描述如何编写可以使用本地化数据的新作业。

Jython 脚本通常用于执行命令，并解析其输出。要以适当的解码方式接收此命令输出，请使用 **ShellUtils** 类的 API。有关详细信息，请参阅“HP Universal CMDB Web 服务 API 概述”（第 292 页）。

此代码通常采用以下形式：

```
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 创建客户端：

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

2 创建一个 **ShellUtils** 类实例，并向其添加操作系统语言。如果未添加语言，则使用默认语言 （通常为英语）：

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

在初始化对象的期间， DFM 将自动检测计算机的语言，并且在预定义的 Language 对象中设置首选编码。首选编码是编码列表中的第一个实例。

3 可使用 **getLanguageBundle** 方法从 **shellclient** 中检索相应的资源数据包：

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

4 从资源数据包中检索适合于特定命令的关键字：

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

5 对 **ShellUtils** 对象调用 **executeCommandAndDecode** 方法并，将关键字传递给该对象：

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

还需要 ShellUtils 对象，以将用户链接到 API 参考（其中详细描述了该方法）。

6 照常解析输出。

解码命令，但不使用关键字

当前的本地化方法是使用关键字来解码所有命令输出。有关详细信息，请参阅“定义使用本地化数据运行的新作业”（第 82 页）中的步骤 “4”（第 83 页）。

不过，另一种方法仅使用关键字对第一个命令输出进行解码，然后通过用于解码第一个命令的字符集对后续命令进行解码。要执行此操作，请使用 **ShellUtils** 对象的 **getCharsetName** 方法和 **useCharset** 方法。

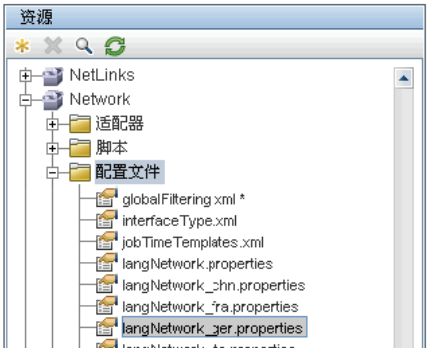
常规用例如下：

1. 调用 **executeCommandAndDecode** 方法一次。
2. 通过 **getCharsetName** 方法获取最近使用的字符集名称。
3. 通过在 **ShellUtils** 对象上调用 **useCharset** 方法，使 **shellUtils** 在默认情况下使用该字符集。
4. 调用 **ShellUtils** 的 **execCmd** 方法一次或多次。此时将使用在步骤 3 中指定的字符集返回输出。无需执行其他解码操作。

使用资源数据包

资源数据包是一种具有属性扩展名 (***.properties**) 的文件。属性文件被视为以**键 = 值**格式存储数据的字典。属性文件中的每一行包含一个**键 = 值**关联。资源数据包的主要功能是按照相应的键返回值。

资源数据包位于 Probe 计算机的以下位置中：**C:\hp\UCMDB\DataFlowProbe\runtime\probeManager\discoveryConfigFiles**。您可以像下载任何其他配置文件一样，从 UCMDB 服务器下载资源数据包。您还可以 “资源”窗口中编辑、添加或删除资源数据包。有关详细信息，请参阅《HP Universal CMDB 数据流管理指南》中的 ““配置文件”窗格”。



搜寻目标时，DFM 通常需要解析命令输出或文件内容中的文本。此解析通常基于正则表达式进行。不同的语言需要使用不同的正则表达式来进行解析。为了能够一次性编写适用于所有语言的代码，必须将所有特定于语言的数据提取到资源数据包中。每种语言都有一个资源数据包。（尽管一个资源数据包可以包含针对多种不同语言的数据，但在 DFM 中，一个资源数据包只包含一种语言的数据。）

Jython 脚本本身并不包含特定于语言的硬编码数据（例如，特定于语言的正则表达式）。但是，该脚本可确定远程系统的语言，然后加载正确的资源数据包，并通过特定键获取特定于语言的所有数据。

在 DFM 中，资源数据包采用以下特定名称格式：**< 基本名称 >_< 语言标识符 >**.properties，例如 langNetwork_spa.properties。（默认资源数据包采用以下格式：**< 基本名称 >**.properties，例如 langNetwork.properties。）

基本名称格式反映了此数据包的用途。例如， **langMsCluster** 表示此资源数据包包含由 MS 群集作业使用的特定于语言的资源。

语言标识符是一个包含 3 个字符的首字母缩略词，用于标识语言。例如， rus 代表俄语， ger 代表德语。此语言标识符包含在 Language 对象的声明中。

API 参考

本节包括：

- “语言类”（第 86 页）
- “executeCommandAndDecode 方法”（第 88 页）
- “getCharsetName 方法”（第 88 页）
- “useCharset 方法”（第 89 页）
- “getLanguageBundle 方法”（第 89 页）
- “osLanguage 字段”（第 89 页）

语言类

此类封装了有关语言的信息，例如资源数据包后缀、可能的编码等等。

字段

名称	描述
语言环境	表示语言环境的 Java 对象。
bundlePostfix	资源数据包后缀。此后缀在资源数据包文件名中使用，用于标识语言。例如，数据包 **langNetwork_ger.properties** 包含 **ger** 数据包后缀。
charsets	用于编码此语言的字符集。每种语言可以有多个字符集。例如，俄语通常采用 Cp866 和 Windows-1251 编码。
wmiCodes	Microsoft Windows 操作系统用于确定语言的 WMI 代码的列表。 http://msdn.microsoft.com/en-us/library/aa394239 (VS.85).aspx （OSLanguage 部分）中列出了所有可能的代码。用于确定操作系统语言的方法之一是查询操作系统的 OSLanguage 属性的 WMI 类。
codepage	用于特定语言的代码页。例如， 866 用于俄语语言的计算机， 437 用于英语语言的计算机。用于确定操作系统语言的方法之一是检索其默认代码页 （例如，通过使用 chcp 命令）。

executeCommandAndDecode 方法

此方法由业务逻辑 Jython 脚本使用。它可以封装解码操作，并返回已解码的命令输出。

参数

名称	描述
cmd	实际要执行的命令。
keyword	用于解码操作的关键字。
framework	传递给 DFM 中每个可执行的 Jython 脚本的框架对象。
timeout	命令超时。
waitForTimeout	指定在超时发生时客户端是否要等待。
useSudo	指定是否要使用 sudo （仅适用于 UNIX 计算机客户端）。
language	可用于直接指定语言，而不是自动检测语言。

getCharsetName 方法

此方法可返回最近使用的字符集名称。

useCharset 方法

此方法在 ShellUtils 实例上设置字符集，该实例使用此字符集进行初始数据解码。

参数

名称	描述
charsetName	字符集的名称，例如 windows-1251 或 UTF-8。

另请参阅 “getCharsetName 方法”（第 88 页）。

getLanguageBundle 方法

可以使用此方法获取正确的资源数据包。它将替换以下 API：

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

参数

名称	描述
baseName	不带语言后缀的数据包名称，例如 langNetwork。
language	语言对象。将在此处传递 ShellUtils.osLanguage。
framework	传递给 DFM 中每个可执行的 Jython 脚本的常用框架对象。

osLanguage 字段

此字段包含一个表示语言的对象。

使用搜寻分析器

搜寻分析器工具用于在开发包、脚本或任何其他内容时执行调试操作。该工具对远程目标运行作业，并返回含有信息、警告和错误详细信息的日志，以及已搜寻到的 CI 的结果。

请注意，并不会始终将这些结果报告给 UI。这是因为用于报告结果的方式有两种，但仅有一种受支持。而且， Eclipse 不支持通信日志。

通过 Eclipse 执行此工具时， **DiscoveryProbe.properties** 文件 (**C:\hp\UCMDB\DataFlowProbe\conf\DiscoveryProbe.properties**) 必须包含设置为 **true** 的以下参数：

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

有关详细信息，请参阅 “通过 Eclipse 运行搜寻分析器”（第 99 页）。

在所有其他情况下（通过 **cmd** 文件执行工具时，或者当 Probe 运行时），必须将此标记设置为 **false**：

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

任务和记录

任务文件中包含有关要执行的任务的数据。任务由作业名称和用于定义触发器 CI 的参数 （例如远程目标地址）等信息组成。

记录文件中包含任务信息以及特定执行操作的结果，即 Probe 或搜寻分析器 （执行任务的模块）与远程目标之间的详细通信信息 （包括响应）。

可以对远程目标执行由任务文件定义的任务，然而由记录文件 （包含有关特定执行过程的额外数据）定义的任务不但可以执行，而且还可以回放 （也就是说，可以重新生成在记录文件中记录的执行过程）。

日志

日志可提供有关最新运行的信息，如下所示：

- **常规日志**。此日志包含在运行期间发生的所有信息数据、错误和警告。
- **通信日志**。此日志包含有关搜寻分析器和远程目标之间的通信 （包括其响应）的详细信息。在执行之后，可以将日志另存为记录文件。
- **结果日志**。显示已搜寻到的 CI 的列表。每个 CI 的出现时间取决于适配器和脚本的设计方式。

您既可以将所有日志保存在一起，也可以单独保存每个日志。保存所有日志时，应将它们一同保存在同一个名称下。

如果重播记录文件，通信日志中会显示相同的数据，唯一的区别在于执行时间不同。

限制：通过 Eclipse 运行搜寻分析器时，通信日志和结果日志不可用。

本节包括以下步骤：

- “先决条件”（第 92 页）
- “访问搜寻分析器”（第 92 页）
- “定义任务”（第 93 页）
- “定义新任务”（第 94 页）
- “检索记录”（第 95 页）
- “打开任务文件”（第 95 页）
- “从数据库导入任务”（第 95 页）
- “编辑任务”（第 95 页）
- “保存任务和日志”（第 96 页）
- “运行任务”（第 96 页）

➤ “将任务结果发送到服务器”（第 97 页）

➤ “导入设置”（第 97 页）

➤ “断点”（第 98 页）

1 先决条件

➤ 必须安装 Probe。（搜寻分析器是在安装 Probe 时安装的，并且可以与 Probe 共享资源。）

➤ 在使用搜寻分析器时，不需要运行 Probe。

但是，如果已经对 UCMDB 服务器运行 Probe，则需要的所有资源已下载到文件系统中。如果 Probe 尚未运行，可以通过 “设置”菜单上载搜寻分析器所需的资源。有关详细信息，请参阅 “导入设置”（第 97 页）。

➤ 不需要安装 CMDB 服务器。

2 访问搜寻分析器

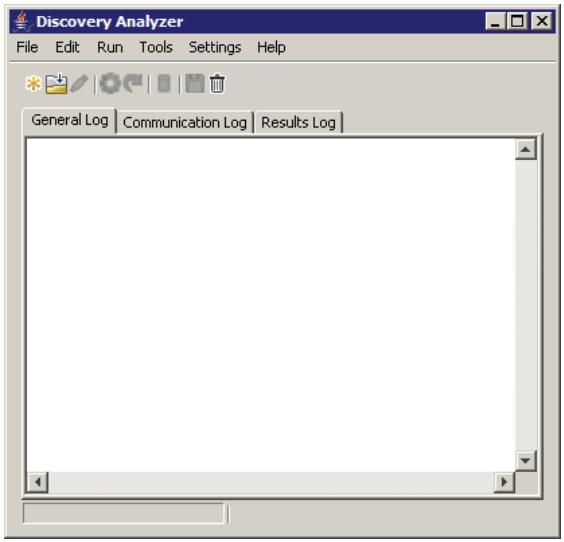
您可以通过以下方式访问搜寻分析器：

➤ 使用 Eclipse 时。

Probe 安装附带默认的 Eclipse 工作区，位于 **C:\hp\UCMDB\DataFlowProbe\tools\discoveryAnalyzerWorkspace** 中。此工作区包括一个用于启动搜寻分析器的 Jython 脚本 (**startDiscoveryAnalyzerScript.py**) ，以及一个指向所有 DFM 脚本的链接。如果通过这种方式启动搜寻分析器，则可以在 Jython 脚本中查找断点，以便进行调试。

➤ 通过双击以下文件夹中的文件，直接访问搜寻分析器：**C:\hp\UCMDB\DataFlowProbe\tools\discoveryAnalyzer.cmd**。有关详细信息，请参阅下一节。

将打开 “Discovery Analyzer”窗口：



3 定义任务

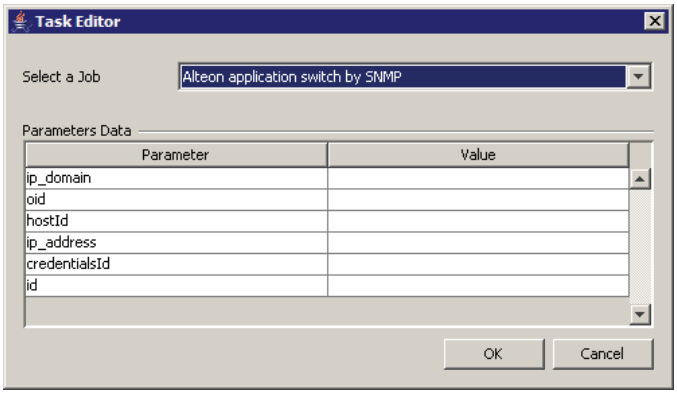
可以使用以下方法之一定义任务：

- 定义新任务。有关详细信息，请参阅 “定义新任务”（第 94 页）。
- 从记录文件导入任务。有关详细信息，请参阅 “检索记录”（第 95 页）。
- 从任务文件导入已保存的任务。有关详细信息，请参阅 “打开任务文件”（第 95 页）。
- 从 Probe 的内部数据库检索作业。有关详细信息，请参阅 “从数据库导入任务”（第 95 页）。

4 定义新任务

a 显示 “任务编辑器”：单击 “新建任务” 按钮

“任务编辑器” 将显示文件系统中当前存在的作业的列表。每次 Probe 从服务器接收任务时，或者在 “设置” 菜单中手动部署数据包时，此列表均会更新。



b 选择作业。

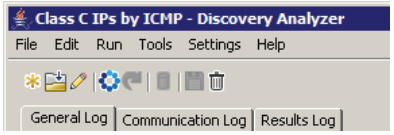
c 输入所有参数的值。

此处显示的参数是 DFM 适配器参数。可以在 “模式签名” 选项卡的 “搜寻模式参数” 窗格中查看这些参数。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 ““适配器参数” 窗格”。

所有字段均为必填字段 （除非作业脚本要求该字段为空）。

对于需要 ID 或凭据 ID 输入值的参数，可以使用随机创建的 ID：右键单击值框，然后选择 “Generate random CMDB ID” 或 “Credential Chooser”。

此时任务将处于活动状态，已打开的任务的名称将显示在标题栏中：



d 继续完成用于定义任务的步骤。有关详细信息，请参阅 “保存任务和日志”（第 96 页）。

5 检索记录

通过打开含有特定执行数据的记录文件，可以定义任务。如果通过这种此方式定义任务，可以通过选择回放选项来重新生成特定执行过程。（如果重播任务，则接收的响应将来自记录文件中存储的数据，而不是远程目标。）

选择 “File” > “Open Record”。浏览到用于保存记录的文件夹。此时任务将处于活动状态，任务名称将显示在标题栏中。

有关如何获取记录文件的详细信息，请参阅 “记录 DFM 代码”（第 109 页）。

6 打开任务文件

可以从任务文件定义任务：选择 “File” > “Open Task”。

7 从数据库导入任务

如果 Probe 已经运行，并且其内部数据库中有活动任务，则可以从 Probe 数据库中检索任务。可以使用参数值定义任务。

a 选择 “File” > “Import Task from Probe Database”。

b 在打开的对话框中，选择要运行的任务，然后单击 “OK”。

c 继续完成用于定义任务的步骤。有关详细信息，请参阅 “保存任务和日志”（第 96 页）。

8 编辑任务

定义任务之后，此任务 （或文件）的名称将显示在标题栏中。此时，即可编辑文件。

a 选择 “Edit” > “Edit Task”。

b 对任务进行更改，然后单击 “OK”。

9 保存任务和日志

可以保存任务参数：选择 “File” > “Save Task”。

只有在执行任务之后，才可使用以下选项。

- 保存任务记录。您可以保存任务参数和任务运行的结果：选择 “File” > “Save Record”。
- 保存任务日志：选择 “File” > “Save General Log”。
- 保存结果：选择 “File” > “Save Results”。

10 运行任务

此过程的下一个步骤是运行所创建的任务。

a 导入凭据 / 范围配置文件。有关详细信息，请参阅“导入设置”（第 97 页）。

b 要仅对远程目标执行任务，请单击 “Run Task”按钮。

搜寻分析器将执行作业，并在三个日志文件中显示以下信息：“常规”、“通信”和 “结果”。

c 您可以将所有日志文件保存在一起，也可以单独保存各个日志：选择 “File”>“Save General Log”、“Save Record”、“Save Results”或“Save All Logs”。有关日志文件的详细信息，请参阅 “日志”（第 91 页）。

d 如果在记录文件中检索到任务，则可以通过单击 “Playback”按钮，重新生成此文件中记录的执行过程。此时将显示相同的通信日志，但执行时间会更新。

11 将任务结果发送到服务器

如果任务执行结束时生成了结果（即“结果日志”选项卡显示已搜寻到的 CI 的列表），则可以将结果发送到 UCMDB 服务器。例如，如果您在服务器关闭时测试脚本，则这十分有用。

注意：只能将结果发送到从特定 Probe（该 Probe 与搜寻分析器安装在同一计算机上）接收任务的 UCMDB 服务器。

---

12 导入设置

要运行任务或回放记录文件，必须导入 **domainScopeDocument.bin** 文件。导入时，请输入密码。

a 启动 Web 浏览器，并输入以下 URL：**http://localhost:8080/jmx-console**。您可能需要使用用户名和密码登录。

b 单击“UCMDB:service=DiscoveryManager”，打开“JMX MBEAN 视图”页面。

c 查找“exportCredentialsAndRangesInformation”操作。执行以下操作：

- 输入客户 ID（默认值是 **1**）。
- 为导出的文件输入名称。
- 输入密码。
- 将 **isEncrypted** 设置为 **False**。

d 单击 “调用”，导出 **domainScopeDocument.bin** 文件。

导出过程成功完成后，文件将保存到此位置：
C:\hp\UCMDB\UCMDBServer\conf\discovery\< 客户目录 >。

e 将 **domainScopeDocument.bin** 文件复制到数据流 Probe 文件系统，并且通过选择以下选项来导入该文件：“设置” > “导入 domainScopeDocument”。

注意：在 **domainScopeDocument** 文件导入过程中，会要求您提供密码。此外，当搜寻服务器每次重新启动时，以及在执行第一个任务和记录之前，均会要求您提供密码。

---

13 断点

如果通过 Python 脚本运行搜寻分析器，可以将断点添加到脚本中。

14 配置 Eclipse

有关在调试模式下运行 Jython 脚本的详细信息，请参阅 “通过 Eclipse 运行搜寻分析器”（第 99 页）。

通过 Eclipse 运行搜寻分析器

本任务说明如何配置 Eclipse，以便在调试模式下运行 Jython 脚本，从而更好地显示作业线程、触发器 CI 和结果。

本节包括以下步骤：

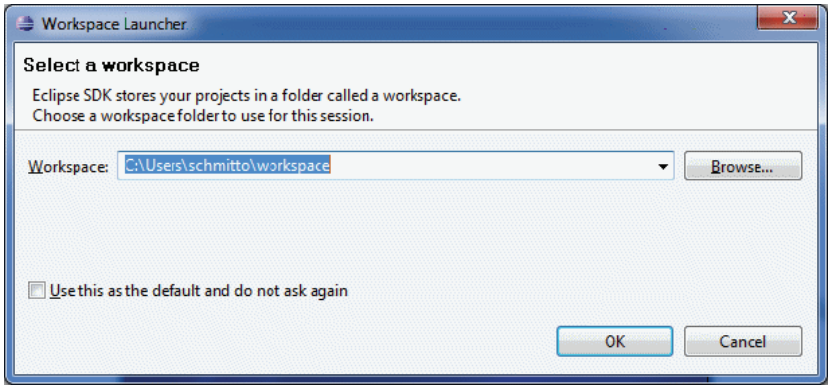
1 先决条件

- 在计算机上安装最新的 Eclipse 版本。可以在 www.eclipse.org 上获取此应用程序。
- 验证同一计算机上是否已安装数据流 Probe。
- 验证 **DiscoveryProbe.properties** 文件中的 **appilog.agent.local.discoveryAnalyzerFromEclipse** 参数是否已设置为 **true**。

2 解压缩并启动 Eclipse

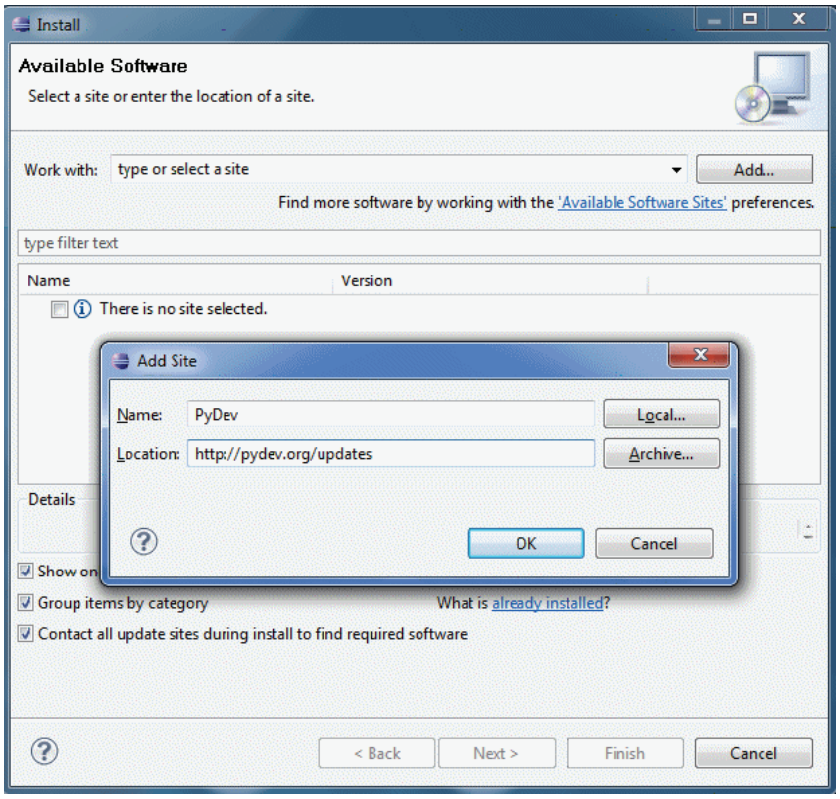
3 配置默认工作区

配置 Eclipse 用于保存和存储所有项目和相关数据的默认工作区。



4 配置 PyDev 扩展

a 访问 “帮助” > “安装新软件”，单击 “添加”，键入 PyDev 插件的名称，并在 “Location” 字段中添加可以下载 pydev 的站点 URL：**http://pydev.org/updates**。单击 “OK”。

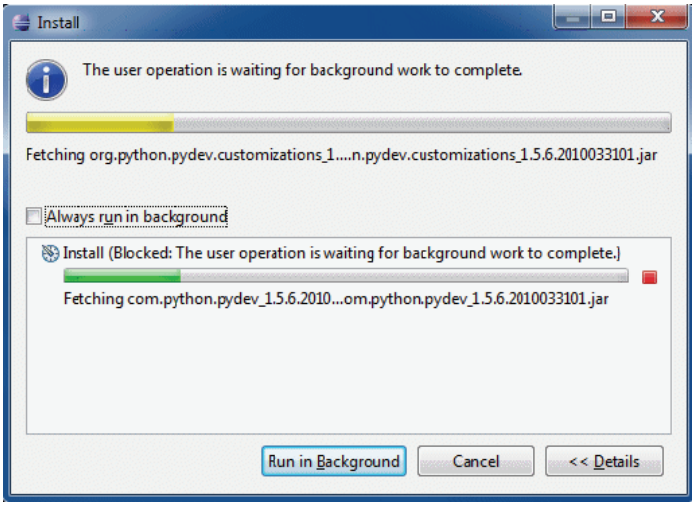


注意：由于 PyDev 扩展已是开放资源，所以现在已将 PyDev 和 PyDev 扩展合并到一个插件中。有关其他信息，请访问 http://pydev.org。

b 在打开的窗口中，选择 “Pydev”。第二个插件是以任务为中心的 UI 的插件。单击 “Next”，检查安装详细信息，并再次单击 “Next”。

c 接受许可证协议，然后单击 “Next”。

d 此时将安装 Pydev。如果要求您安装未签名的内容，单请击 “OK” 进行确认。

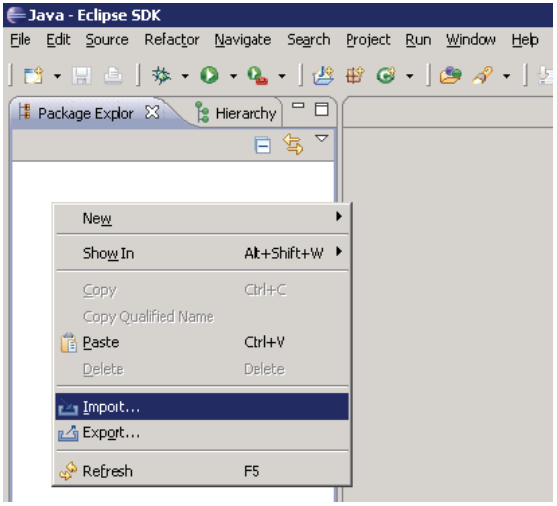


e 重新启动 Eclipse。

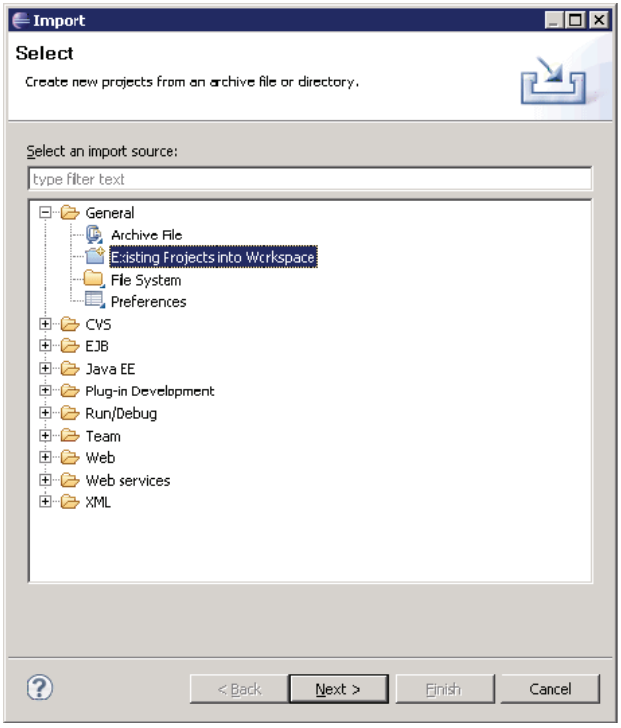
此时已将 PyDev 安装在 Eclipse IDE 中。Eclipse 中已有新透视，而且 IDE 能够解释 Python 脚本 （文本突出显示、其他配置选项等等）。

5 配置搜寻分析器工作区

a 导入所需文件：在包浏览器的白色区域中右键单击，然后单击“Import”以导入 Probe 安装程序附带的预配置 **discoveryAnalyzerWorkspace**。



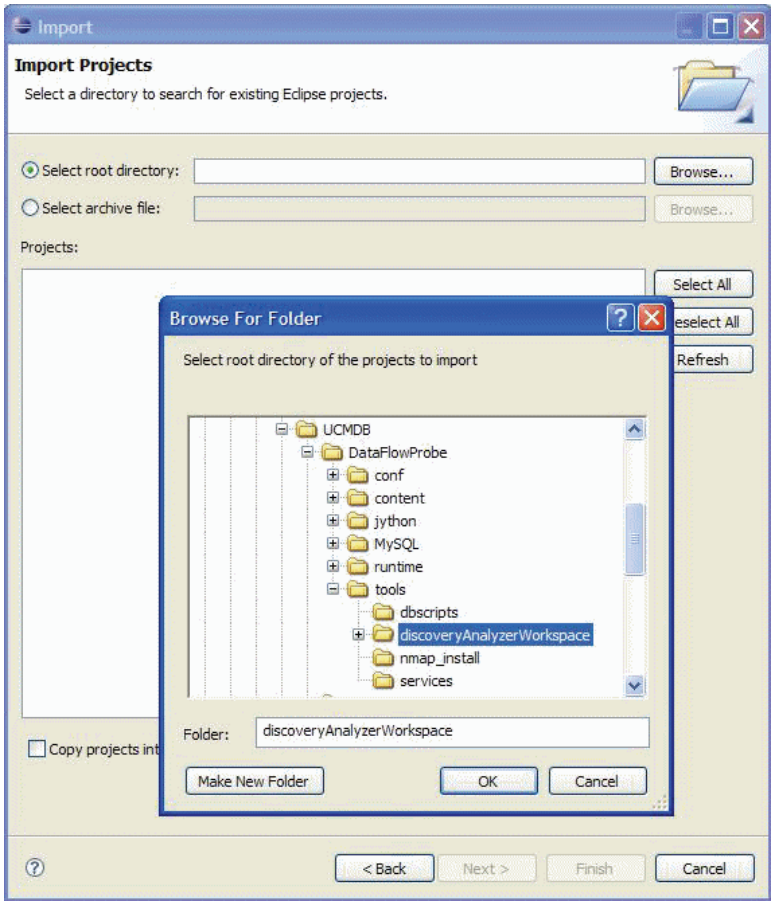
b 在 “General” 下，选择 “Existing Projects into Workspace”，将项目导入到 Eclipse 工作区中。

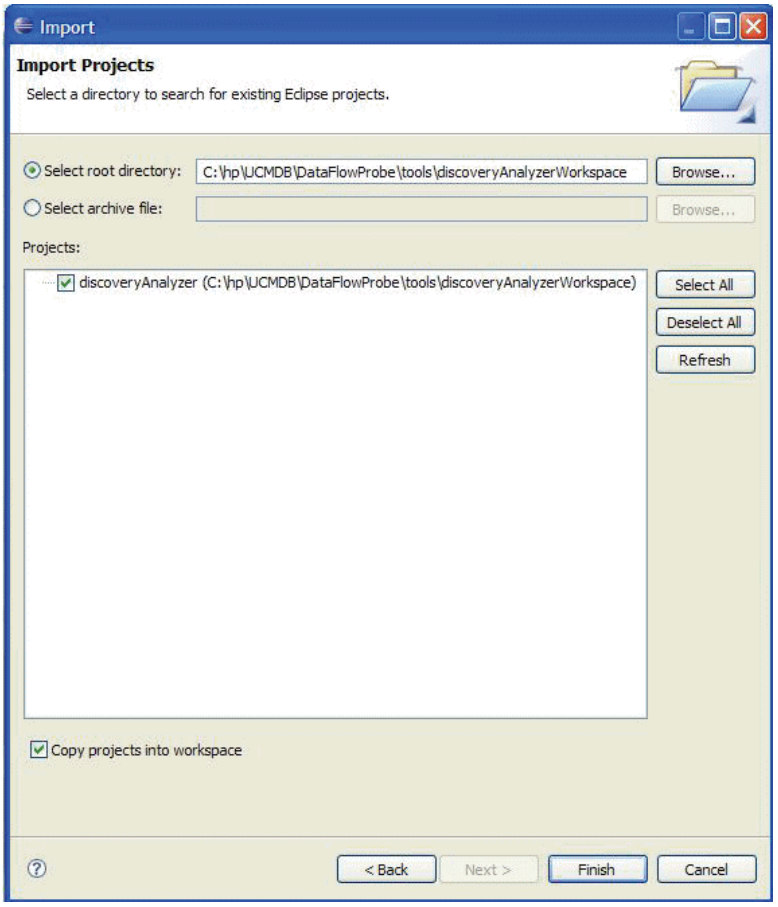


c 在 “Select root directory” 下，选择分析器工作区 （该空间通常位于 **C:\hp\UCMDB\DataFlowProbe\tools\discoveryAnalyzerWorkspace**）。

d 选择 “Copy projects into workspace”，创建现有工作区的真实副本。此步骤很重要：如果失败，您可以重新导入原始的 **discoveryAnalyserWorkspace**。

e 单击 “Finish”，开始导入。





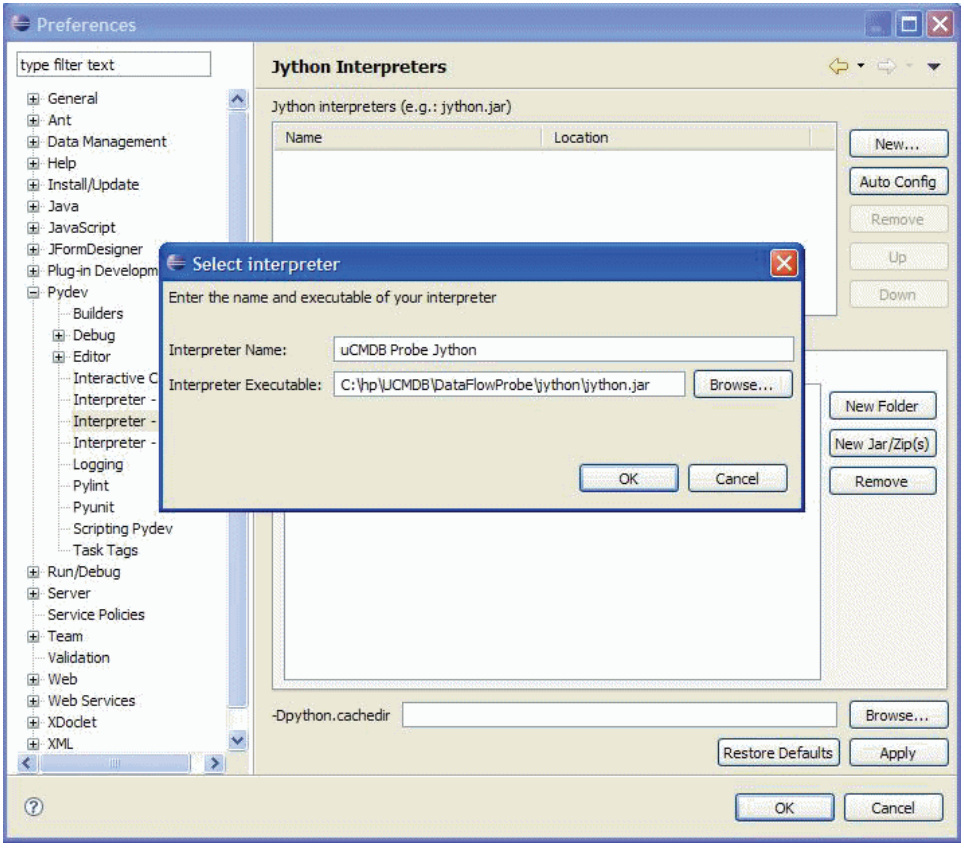
6 配置类路径和解释程序

a 右键单击 **discoveryAnalyzerWorkspace**，然后选择“Properties”，显示项目的特定设置。

b 转到**“Pydev”** > **“Interpreter/Grammar”**，然后单击“Please configure an interpreter in the related preferences before proceeding”。

此步骤配置的 Jython 解释程序与 Probe 使用的程序相同，以确保使用相同的 Jython 版本解释脚本。

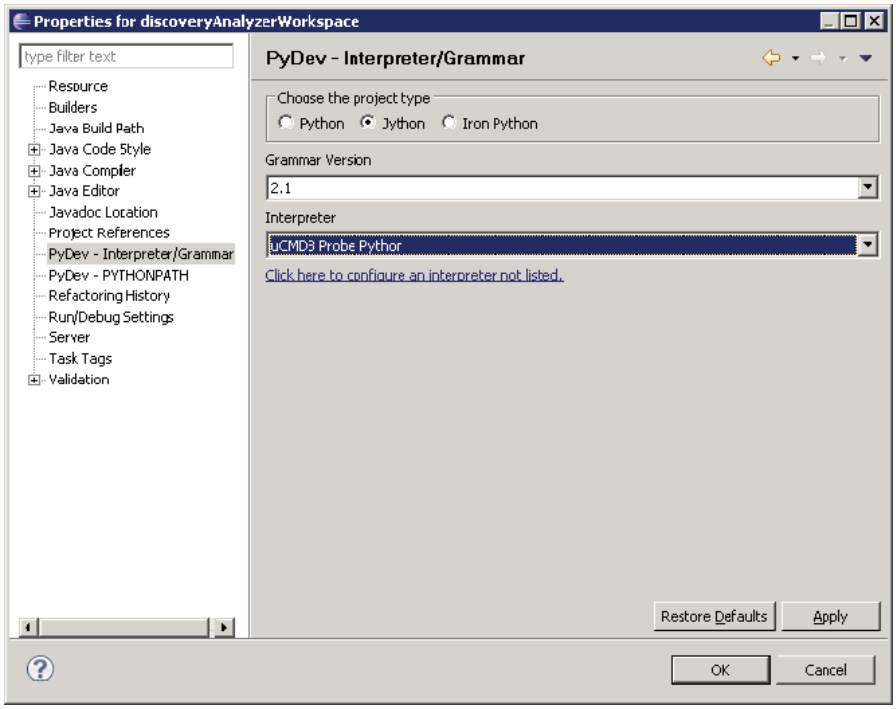
c 单击“New”，键入解释程序的名称，然后从以下文件夹中选择文件：**C:\hp\UCMDB\DataFlowProbe\jython\jython.jar**。



d 单击“OK”。如果此时显示一个窗口，要求您选择要导入 Python 系统路径的文件夹，请不要进行任何更改（应当是“C:\hp\UCMDB\DataFlowProbe\jython”和“C:\hp\UCMDB\DataFlowProbe\jython\lib”），然后单击“OK”。

e 单击“Apply”，然后单击“OK”。

f 单击 “Interpreter”，然后选择刚才创建的解释程序。



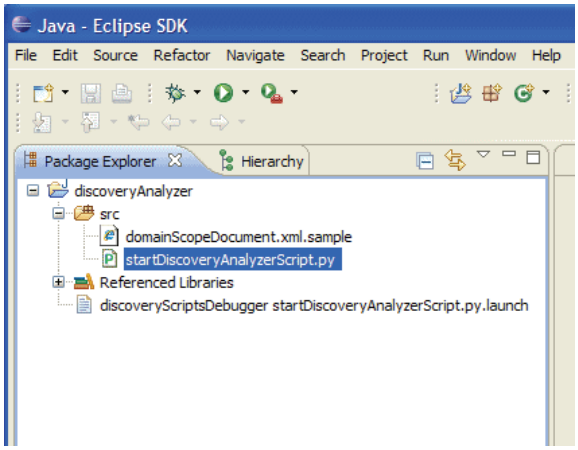
g 单击 “Apply”，然后单击 “OK”。

此时的 Jython 解释程序与 Probe 所使用的解释程序相同。

7 运行搜寻分析器

a 在要调试的 Jython 脚本中添加断点。

b 要启动搜寻分析器，请在 “discoveryAnalyzerWorkspace\src” 项目中选择 “startDiscoveryAnalyzerScript.py”。右键单击此文件，然后选择 “Debug as” > “Jython run”。



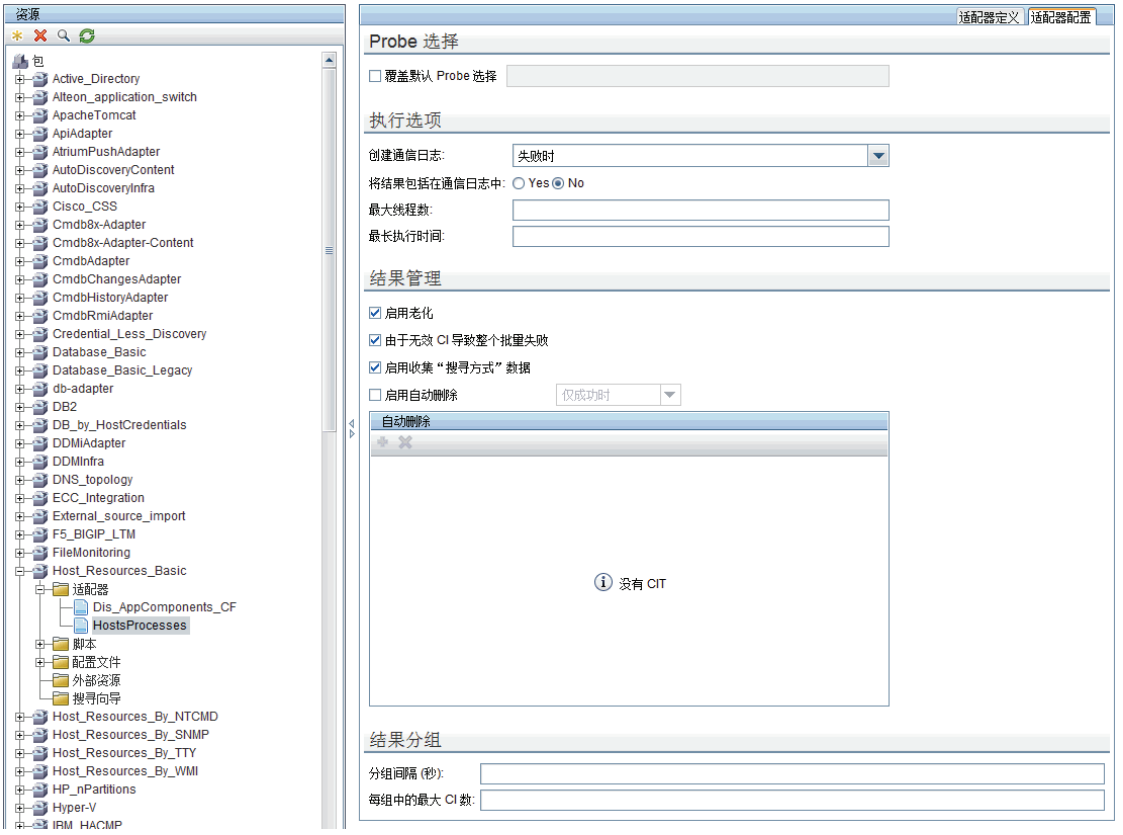
记录 DFM 代码

记录完整的执行过程（包括所有参数）十分有用（例如，在调试和测试代码时）。本任务描述如何记录包含所有相关参数的完整执行过程。此外，还可以查看即使在调试级别也不会输出到日志文件中的额外调试信息。

要记录 DFM 代码，请执行以下操作：

1 访问 “数据流管理” > “搜寻控制面板”。右键单击必须记录其运行过程的作业，然后选择 “编辑适配器”，以打开适配器管理应用程序。

2 在“模式管理”选项卡中查找“执行选项”窗格：

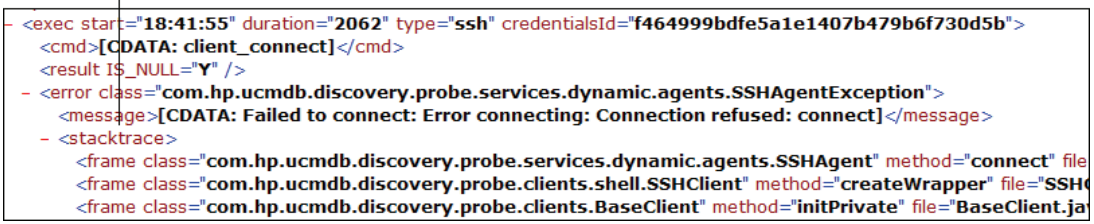


3 将“创建通信日志”框更改为“始终”。有关设置日志选项的详细信息，请参阅《HP Universal CMDB 数据流管理指南》中的“执行选项”窗格”。

以下示例是在运行 Host Connection by Shell 作业且“创建通信日志”框设置为“始终”或“失败时”时，所创建的 XML 日志文件：

以下示例显示了消息和堆栈跟踪参数：

堆栈跟踪



参考

Jython 库和实用程序

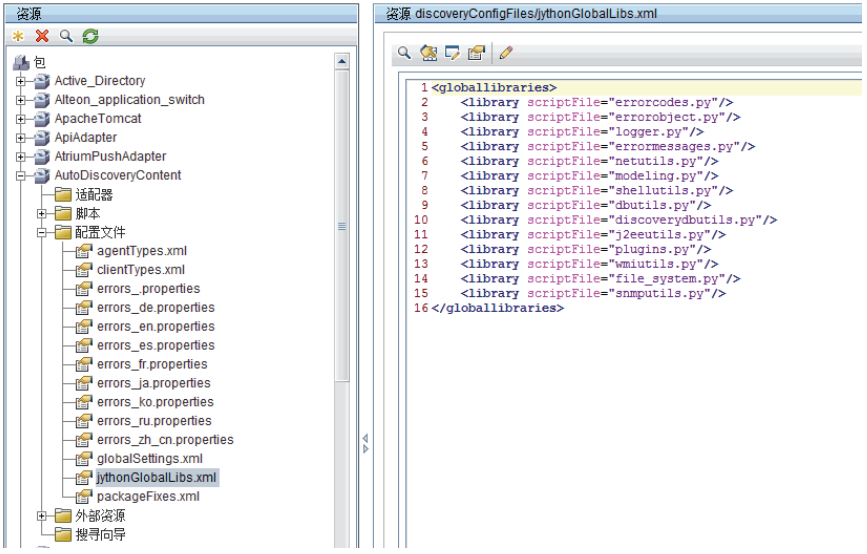
某些实用程序脚本在适配器中使用得很广泛。这些脚本包含在 AutoDiscovery 数据包中，它们与下载到 Probe 中的其他脚本一起位于以下位置：**C:\hp\UCMDB\DataFlowProbe\runtime\probeManager\discoveryScripts**。

注意：discoveryScript 文件夹是在 Probe 开始工作时动态创建的。

要使用某个实用程序脚本，请将以下导入行添加到脚本的导入部分：

```
import <script name>
```

AutoDiscovery Python 库包含 Jython 实用程序脚本。这些库脚本被视为 DFM 的外部库。可以在 jythonGlobalLibs.xml 文件 （位于 “配置文件” 文件夹下）中定义这些脚本。



在 Probe 启动时，默认情况下会加载 jythonGlobalLibs.xml 文件中出现的每个脚本，因此无需在适配器定义中明确的使用这些脚本。

本节包括以下主题：

logger.py

logger.py 脚本包含用于报告错误的日志实用程序和帮助程序函数。您可以调用其调试、信息和错误 API，以写入日志文件。日志消息记录在 **C:\hp\UCMDB\DataFlowProbe\runtime\log** 中。

将根据为 **C:\hp\UCMDB\DataFlowProbe\conf\log\probeMgrLog4j.properties** 文件中的 PATTERNS_DEBUG 输出目标定义的调试级别，在日志文件中输入消息。（默认情况下，该级别为 “DEBUG”。）有关详细信息，请参阅 “错误严重度级别”（第 122 页）。

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

信息和错误消息还会显示在 “命令提示符”控制台中。

有两组 API：

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

第一组 API 在相应的日志级别发出所有字符串参数的串联；第二组 API 既发出串联，也发出最近抛出的异常的堆栈跟踪，以提供更多信息：

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

modeling.py

modeling.py 脚本包含用于创建主机、 IP 和进程 CI 等对象的 API。这些 API 支持创建常见对象，并使代码更具可读性。例如：

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

netutils.py

netutils.py 库用于检索网络和 TCP 信息，例如检索操作系统名称、检查 MAC 地址是否有效、检查 IP 地址是否有效，等等。例如：

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

shellutils.py

shellutils.py 库提供了用于执行 shell 命令和检索已执行命令的结束状态的 API，并且支持基于该结束状态运行多个命令。此库随 Shell 客户端初始化，使用该客户端运行命令和检索结果。例如：

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

4

错误消息

本章包括：

概念

参考

概念

错误消息概述

搜寻期间可能会发生多种错误，例如连接失败、硬件问题、异常、超时等。每当常规搜寻流程失败时， DFM 就会采用 “基本”和 “高级”两种模式，将这些错误显示在“搜寻控制面板”中。您可以从引发问题的触发器 CI 向下搜索，以查看错误消息本身。

DFM 能够将有时可忽略的错误 （例如主机无法访问）和必须处理的错误 （例如凭据问题，配置或 DLL 文件丢失）区分开来。另外，即使在连续运行时发生同样的错误， DFM 也只会报告一次，但是 DFM 也不会漏掉只出现一次的错误。

创建程序包时，可以将合适的消息作为资源添加到该程序包内。在部署程序包期间，这些消息也会部署到正确的位置。消息必须遵从某些约定，如 “错误编写约定”（第 119 页）中所述。

DFM 支持多语言错误消息。您可以将写入的消息本地化，以便它们显示为本地语言。

有关搜索错误的详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 ““搜寻状态”窗格”。

有关设置通信日志的详细信息，请参阅《HP Universal CMDB 数据流管理指南》中的 ““执行选项”窗格”。

参考

错误编写约定

- 每个错误均通过一个错误消息代码和一个参数数组（**int, String[]**）进行标识。消息代码和参数数组的组合可以定义一个特定错误。参数数组可以为空。
- 每个错误代码都会映射到一个**短消息**（固定字符串）和一个**详细消息**（包含零个或多个参数的模板字符串）。假定模板参数的数量等于实际参数的数量。

  错误消息代码的示例：

  10234 可表示简短消息错误：

  ```
  Connection Error
  ```

  和详细消息错误：

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

  其中

  {0} = 第一个参数：协议名称

  {1} = 第二个参数：超时长度 （以毫秒为单位）

本节还包括以下主题：

属性文件内容

对于每个错误消息代码，属性文件必须包含它的两个密钥。例如，对于错误 45：

- **DDM_ERROR_MESSAGE_SHORT_45**。短错误描述。
- **DDM_ERROR_MESSAGE_LONG_45**。长错误描述 （可以包含参数，如 {**0**}、{**1**}）。

错误消息属性文件

在属性文件中，错误消息代码和两条消息 （简短消息和详细消息）之间存在一个映射。

部署某个属性文件后，该文件中的数据将与现有数据合并，即添加新消息代码时会覆盖旧消息代码。

基础结构属性文件是 **AutoDiscoveryInfra** 包的一部分。

语言环境命名约定

- 对于默认语言环境：< **文件名** >.properties.errors
- 对于特定语言环境：< **文件名** >_xx.properties.errors

  其中 **xx** 为语言环境 （例如 **infraerr_fr.properties.errors** 或 **infraerr_en_us.properties.errors**）。

错误消息代码

默认情况下，HP Universal CMDB 包括以下错误代码。您可以将自己的错误消息添加到此列表中。

错误名称	错误代码	描述
内部	100-199	主要通过在 Jython 脚本运行期间所引发的异常解析
连接	200-299	连接失败、目标计算机没有代理、目标无法访问等

错误名称	错误代码	描述
凭据相关	300-399	由于缺少凭据，权限遭到拒绝，连接尝试被阻止
超时	400-499	连接 / 命令期间超时
非预期或无效行为	500-599	配置文件丢失、非预期中断等
信息检索	600-699	目标计算机丢失信息，代理信息查询失败等
资源相关	700-799	内存不足错误或客户端发布错误
解析	800-899	解析文本时出错
编码	900	输入错误，编码不受支持
SQL 相关	901-903, 924	收到 SQL 操作错误
HTTP 相关	904-909	HTTP 连接期间发生的错误，通过 HTTP 错误代码进行解析。
特定应用程序	910-923	特定应用程序问题导致的错误，例如 LSOF 版本错误、未发现队列管理器，等等

未分类的内容错误

要支持旧内容但不引起回归，应用程序和 SDK 相关方法需要采用不同的方式处理消息代码为 100 的错误 （即未分类脚本错误）。

这些错误的分组 （也就是说，它们并不被视为同一类型的错误）并非基于消息代码，而是消息内容。也就是说，如果某个脚本通过已弃用的旧方法 （包含消息字符串，但不包含错误代码）报告错误，则所有消息都会接收到相同的错误代码，但在应用程序中或在 SDK 相关方法中，不同的消息会显示为不同的错误。

框架中的更改

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

将以下方法添加到接口：

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

系统仍然支持以下旧方法以实现向后兼容，但会将它们标记为 “已弃用”：

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

错误严重度级别

适配器在触发器 CI 上结束运行后，会返回一个状态。如果未报告错误或警告，则该状态将为 “成功”。

以下列出了从最窄到最宽范围的严重度级别：

致命错误

此级别报告严重错误，如基础结构问题、 DLL 文件丢失或异常：

- 生成任务失败 （未发现 Probe、未发现变量等）
- 无法运行脚本
- 无法在服务器上处理结果，导致数据无法写入 CMDB

错误

此级别报告导致 DFM 无法检索数据的问题。通常需要执行某些操作才能检查这些错误，如增加超时、更改范围、更改参数、添加其他用户凭据等。

- 在用户干预可帮助解决问题的情况下，系统会报告一个错误。可能是需要进一步调查的凭据问题或网络问题。（这些问题并不是搜寻错误，而是配置错误。）
- 内部失败，通常由搜寻到的计算机或应用程序的非预期行为引发，如配置文件丢失等

警告

当运行成功，但是可能存在需注意的非严重问题时，DFM 会将严重度标记为“警告”。在开始进行更详细的调试会话之前，您需要查看这些 CI 才能获知数据是否丢失。“警告”可以包含此类消息：远程主机上缺少已安装的代理、无效数据导致某个属性无法正确计算。

- 连接代理丢失 （SNMP、 WMI）
- 搜寻成功，但并未搜寻到所有的可用信息

5

开发常规数据库适配器

本章包括：

概念

任务

➤ 使用 Eclipse 在 CIT 属性和数据库表之间进行映射 （第 155 页）

参考

➤ 适配器配置文件 （第 174 页）

➤ 现成的转换器 （第 196 页）

➤ 插件 （第 200 页）

➤ 配置示例 （第 201 页）

➤ 适配器日志文件 （第 212 页）

➤ 外部参考 （第 214 页）

“疑难解答和局限性”（第 214 页）

概念

常规数据库适配器概述

常规数据库适配器平台用于创建可与关系数据库管理系统 (RDBMS) 集成，并可对数据库运行 TQL 查询和填入作业的适配器。常规数据库适配器支持的 RDBM 包括 Oracle、 Microsoft SQL Server 和 MySQL。

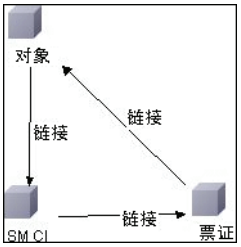
此版本的数据库适配器基于 JPA (Java Persistence API) 标准，并将 Hibernate ORM 库作为持久性提供程序。

不受支持的 TQL 查询

对于仅通过常规数据库适配器计算的 TQL 查询，存在以下限制：

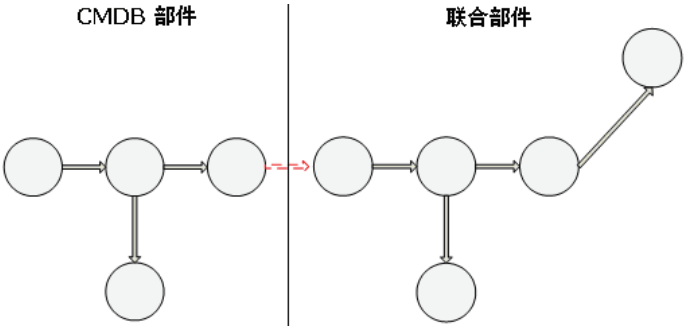
- 不支持子图
- 不支持复合关系
- 不支持周期或周期部分

  以下 TQL 查询是一个周期示例：



- 不支持功能布局。
- 不支持 0..0 基数。

- 不支持**连接**关系。
- 不支持限定符条件。
- 要连接两个 CI，外部数据库源中必须存在表格或外键形式的关系。



调和

调和过程将作为 TQL 计算的一部分在适配器端上执行。要执行调和，请将 CMDB 端映射到称为调和 CIT 的联合实体。

映射。将 CMDB 中的每个属性映射到数据源中的一列。

尽管映射可以直接完成，但是仍然支持对映射数据执行转换。您可以通过 Java 代码添加新功能（例如小写、大写）。这些功能可用于启用值（即在 CMDB 和联合数据库中以不同格式存储的值）转换。

注意：

- 要连接 CMDB 与外部数据库源，数据库中必须存在合适的关联。有关详细信息，请参阅 “先决条件”（第 132 页）。
- 具有 CMDB id 的调和也受支持。

Hibernate 作为 JPA 提供程序

Hibernate 是一种“对象 - 关系”(OR) 映射工具，通过此工具，可以将 Java 类映射到多种类型的关系数据库（例如 Oracle 和 Microsoft SQL Server）中的表。有关详细信息，请参阅 “功能限制”（第 215 页）。

在初级映射中，每个 Java 类将映射到单个表中。较高级的映射则支持继承映射（该映射可在 CMDB 数据库中发生）。

其他受支持的功能包括将一个类映射到多个表中、支持集合，以及一对一、一对多和多对一的关联。有关详细信息，请参阅 “关联”（第 131 页）。

对我们而言，无需创建 Java 类。映射被定义为从 CMDB 类模型 CIT 映射到数据库表。

本节还包括以下主题：

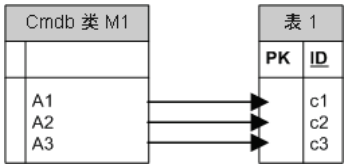
- ““对象 - 关系”映射示例”（第 130 页）
- “关联”（第 131 页）
- “可用性”（第 131 页）

“对象 - 关系”映射示例

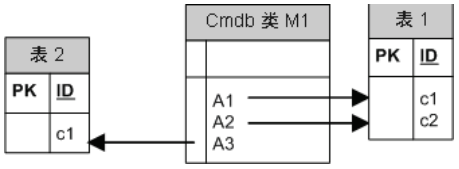
以下示例描述了 “对象 - 关系”映射：

一个 CMDB 类映射到一个数据库表的示例：

包含属性 A1、 A2 和 A3 的类 M1 映射到表 1 的列 c1、 c2 和 c3。这意味着任何 M1 实例在表 1 中都有匹配行。

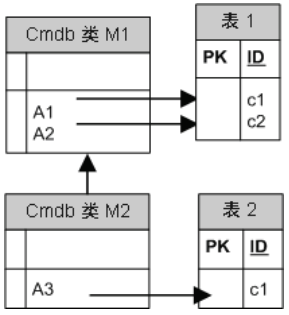


一个 CMDB 类映射到两个数据库表的示例：



继承示例：

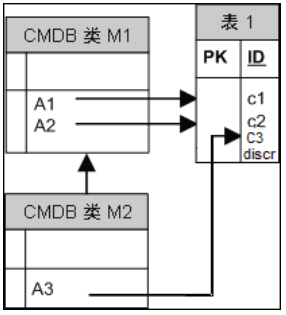
此情况用于 CMDB，其中每个类都具有自己的数据库表。



具有区分器的单表继承示例：

类的整个层次结构将映射到单个数据库表，此表中的各列包含所映射类的所有属性的超集，还包含一个附加列 （**区分器**），该列的值表示要映射到此条目的特定类。

使用区分器功能时，不能跳过层次结构中的类，也就是说，因为 C3 继承自 C2， C2 继承自 C1，所以不能仅定义 C1 和 C3，而必须对这三个类都进行定义。



关联

存在三种类型的关联：一对多、多对一和多对多。要连接不同的数据库对象，必须使用外键列 （适用于一对多的情况）或映射表 （适用于多对多的情况）来定义这些关联类型之一。

可用性

鉴于 JPA 架构的用途非常广泛，因此提供了一个精简的 XML 文件以方便进行定义。

此 XML 文件的用例如下: 联合数据被建模到一个联合类中。此类与非联合 CMDB 类之间具有多对一关系。此外，此联合类和非联合类之间只能存在一种关系类型。

任务

准备创建适配器

本任务描述了在创建适配器前必须执行的准备工作。

注意：您可以在 UCMDB API 中查看常规 DB 适配器的示例。特别地，DDMi 适配器示例包含一个复杂的 **orm.xml** 文件，以及某些插件界面。

本任务包括以下步骤：

- “先决条件”（第 132 页）
- “创建 CI 类型”（第 135 页）
- “创建关系”（第 135 页）

1 先决条件

要验证您是否可以在数据库中使用数据库适配器，请检查以下各项：

- 数据库中是否存在调和类及其属性（也称为多节点）。例如，如果按节点名称运行调和，则验证是否存在一个包含节点名称列的表。如果按节点 cmdb_id 运行调和，则验证是否存在一个与 CMDB 中节点的 CMDB ID 匹配的 CMDB ID 列。有关调和的详细信息，请参阅 “调和”（第 128 页）。

ID	名称	IP 地址
31	BABA	16.59.33.60
33	ext3.devlab.ad	16.59.59.116

ID	名称	IP 地址
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

- 要通过某种关系将两个 CIT 关联，CIT 表之间必须存在关联数据。此关联既可以通过外键列，也可以通过映射表实现。例如，要将节点与票证关联，则票证表中必须存在一个包含节点 ID 的列，节点表中必须存在一个包含与此表相连接的票证 ID 的列，或者映射表的 end1 是节点 ID，而 end2 是票证 ID。有关关联数据的详细信息，请参阅 “Hibernate 作为 JPA 提供程序”（第 129 页）。

  下表显示的是外键 NODE_ID 列：

NODE_ID	CARD_ID	CARD_TYPE	CARD_NAME
2015	1	串行总线控制器	Intel 82801EB USB 通用主控制器
3581	2	系统	Intel 631xESB/6321ESB/3100 芯片集 LPC
3581	3	显示	ATI ES1000
3581	4	基础系统外围	HP ProLiant iLO 2 旧版支持功能

- 每个 CIT 可以映射到一个或多个表。要将一个 CIT 映射到多个表，请检查是否存在一个主表，其主键存在于其他表中并且是唯一的值列。

例如，票证将映射到两个表中：ticket1 和 ticket2。第一个表包含列 c1 和 c2，第二个表包含列 c3 和 c4。要将这两个表视为一个表，则这两个表的主键必须相同。或者，第一个表的主键是第二个表中的某列。

以下示例中，各表共享称为 CARD_ID 的同一主键。

CARD_ID	CARD_TYPE	CARD_NAME
1	串行总线控制器	Intel 82801EB USB 通用主控制器
2	系统	Intel 631xESB/6321ESB/3100 芯片集 LPC
3	显示	ATI ES1000
4	基础系统外围	HP ProLiant iLO 2 旧版支持功能

CARD_ID	CARD_VENDOR
1	Hewlett-Packard Company
2	（标准 USB 主控制器）
3	Hewlett-Packard Company
4	（标准系统设备）
5	Hewlett-Packard Company

2 创建 CI 类型

在此步骤中，可以创建一个要映射到 RDBMS （外部数据源）中数据的联合 CIT。

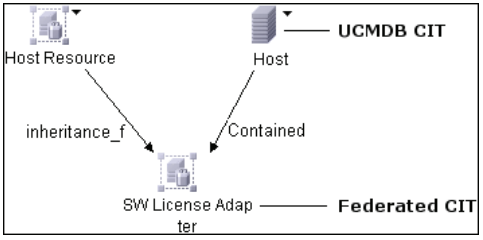
a 在 UCMDB 中，访问 “CI 类型管理器”，然后创建新 CI 类型。有关详细信息，请参阅 《HP Universal CMDB 建模指南》中的 “创建 CI 类型”。

b 向 CIT 添加必需属性，如上次访问时间、供应商等。适配器将从外部数据源检索这些属性，并将其包含到 CMDB 视图中。

3 创建关系

在此步骤中，可以在 UCMDB CIT 和新 CIT 之间添加关系，表示要从外部数据源联合的数据。

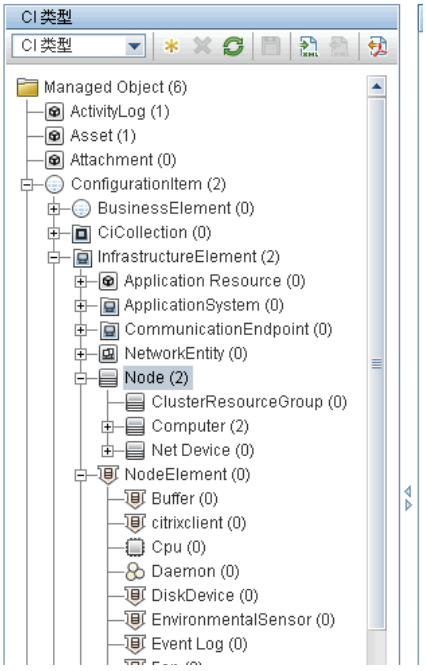
向新 CIT 添加合适的有效关系。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的 ““添加 / 删除关系”对话框”。



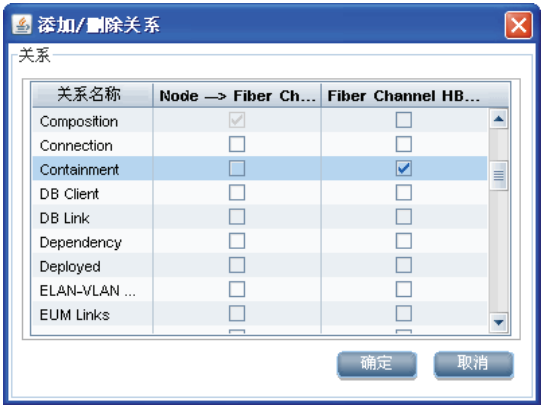
注意：在此阶段，您还无法查看联合数据，因为尚未对用于带入数据的方法进行定义。

创建包含关系的示例：

1 在 CIT 管理器中，选择两个 CIT：



2 在这两个 CIT 之间创建 **Containment** 关系：



准备适配器包

在此步骤中，可以查找并配置常规 DB 适配器包。

1 在 **C:\hp\UCMDB\UCMDBServer\content\adapters** 文件夹中找到 **db-adapter.zip**。

2 将此包提取到本地临时目录中。

3 编辑适配器 XML 文件：

- 在文本编辑器中打开 **discoveryPatterns\db_adapter.xml** 文件。
- 找到 **adapter id** 属性，并替换名称：

```
<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">
```

如果适配器支持数据复制，则将以下功能添加到 **<adapter-capabilities>** 元素：

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

在 HP Universal CMDB 中，显示标签或 ID 将出现 “集成点” 窗格的适配器列表中。

有关使用数据填充 CMDB 的详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 ““集成工作室” 页面”。

➤ 如果适配器使用的是 8.x 版中的映射引擎 （即，不是新的调和映射引擎），则将以下元素：

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

替换为

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

要还原到新映射引擎，请将元素返回到以下值：

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

➤ 查找 **category** 定义：

```
<category>Generic</category>
```

将 **Generic** 类别名称更改为所选的类别。

注意：在创建新集成点时，类别被指定为 **Generic** 的适配器不会显示在集成工作室中。

---

4 在临时目录中，打开 **adapterCode** 文件夹，并将 **GenericDBAdapter** 重命名为在步骤 3 中使用的 **adapter id** 值。

此文件夹包含用于执行联合逻辑的 jar 文件，例如， CMDB 中的适配器名称、查询和类以及适配器支持的 RDBMS 中的字段。

5 根据需要配置适配器。有关详细信息，请参阅 “配置适配器”（第 140 页）。

6 创建一个 *.zip 文件，其名称与 **adapter id** 属性的名称相同，如步骤 “3”（第 137 页）所述。

注意：descriptor.xml 文件是一个存在于每个包中的默认文件。

---

7 保存在上一步中创建的新包。适配器的默认目录是：
C:\hp\UCMDB\UCMDBServer\content\adapters。

将常规 DB 适配器从 9.00 或 9.01 更新为 9.02 及更高的版本

1 将适配器包复制到本地临时目录。

2 提取文件。

3 从 **adapterCode\< 您的适配器名称 >** 文件夹中删除以下文件：

- **asm.jar**
- **asm-attrs.jar**
- **cglib.jar**
- **db-adapter.jar**
- **jboss-archive-browsing.jar**
- **saxon-b.jar**

4 重新创建适配器包。

注意：对于您可能拥有的所有已部署的常规 DB 适配器， UCMDB 安装程序将移除 UCMDB 和 Probe 文件系统中的必需文件。但是，您仍需自行修复适配器包，才能在需要时重新部署包。

配置适配器

可以使用以下方法之一配置适配器：

- “适配器配置 – 最小方法”（第 140 页）
- “适配器配置 – 高级方法”（第 143 页）

这些配置文件位于 **C:\hp\UCMDB\UCMDBServer\content\adapters** 文件夹的 **db-adapter.zip** 包中，是在 “准备适配器包”（第 137 页）的步骤 2 中提取的。

适配器配置 – 最小方法

注意：由于运行此方法而自动生成的 **orm.xml** 文件可以当作使用高级方法时的一个好示例。

以下过程描述了如何将 CMDB 中的类模型映射到 RDBMS。您可以在需要执行以下操作时使用此最小方法：

- 联合诸如节点属性之类的单个节点。
- 演示常规数据库适配器的功能。

此方法：

- 仅支持单节点联合
- 仅支持多对一虚拟关系

本任务包括以下步骤：

- “配置 adapter.conf 文件”（第 141 页）
- “配置 simplifiedConfiguration.xml 文件”（第 141 页）

配置 adapter.conf 文件

在此步骤中，可以更改 adapter.conf 文件中的设置，从而使数据自动联合。

1 在文本编辑器中打开 **adapter.conf** 文件。

2 查找行：**use.simplified.xml.config=<true/false>**。

3 将此行更改为 **use.simplified.xml.config=true**。

配置 simplifiedConfiguration.xml 文件

在此步骤中，可以通过将 CMDB 中的 CIT 映射到 RDBMS 表中的字段，来配置 **simplifiedConfiguration.xml** 文件。

1 在文本编辑器中打开 **simplifiedConfiguration.xml** 文件。

此文件包括一个模板，适用于每个要映射的实体。

注意：请不要在任何版本的 Microsoft Corporation Notepad 中编辑 **simplifiedConfiguration.xml** 文件。请使用 Notepad++、UltraEdit 或其他第三方文本编辑器。

---

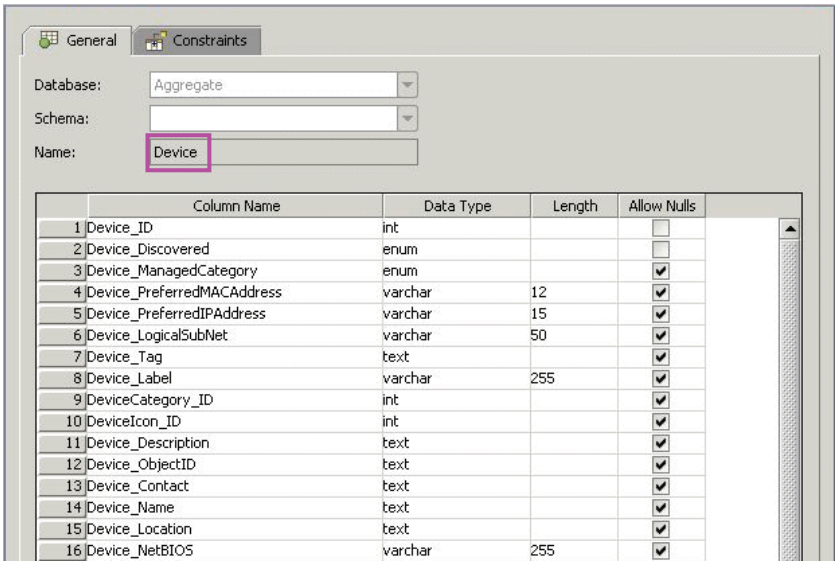
2 对以下属性进行更改：

➤ UCMDB 中的 CIT 名称 (cmdb-class-name) 和 RDBMS 中相应的表名称 (default-table-name)：

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

cmdb-class-name 属性来自节点 CIT：

default-table-name 属性来自设备表：



	Column Name	Data Type	Length	Allow Nulls
1	Device_ID	int		☐
2	Device_Discovered	enum		☐
3	Device_ManagedCategory	enum		☑
4	Device_PreferredMACAddress	varchar	12	☑
5	Device_PreferredIPAddress	varchar	15	☑
6	Device_LogicalSubNet	varchar	50	☑
7	Device_Tag	text		☑
8	Device_Label	varchar	255	☑
9	DeviceCategory_ID	int		☑
10	DeviceIcon_ID	int		☑
11	Device_Description	text		☑
12	Device_ObjectID	text		☑
13	Device_Contact	text		☑
14	Device_Name	text		☑
15	Device_Location	text		☑
16	Device_NetBIOS	varchar	255	☑

- RDBMS 中的唯一标识符：

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

- 调和规则 (reconciliation-by-two-nodes)：

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

- UCMDB 中的调和属性 (cmdb-attribute-name) 和 RDBMS 中的调和属性 (column-name)：

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

- CIT 名称 (cmdb-class-name) 和 RDBMS 中相应的表名称 (default-table-name)。以及 CMDB 关系 (connected-cmdb-class-name) 和 CIT 关系 (link-class-name)：

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

- 主键和外键：

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

- RDBMS 中的唯一标识符：

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

- CMDB 属性 (cmdb-attribute-name) 和 RDBMS 中的列名称 (column-name) 之间的映射：

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

3 保存该文件。

适配器配置 – 高级方法

本任务包括以下步骤：

- “配置 orm.xml 文件”（第 144 页）
- “配置 reconciliation_types.txt 文件”（第 148 页）
- “配置 reconciliation_rules.txt 文件”（第 148 页）

配置 orm.xml 文件

在此步骤中，可以将 CMDB 中的 CIT 和关系映射到 RDBMS 中的表中。

1 在文本编辑器中打开 **orm.xml** 文件。

默认情况下，此文件包含一个模板，可用于根据联合的需要映射尽可能多的 CIT 和关系。

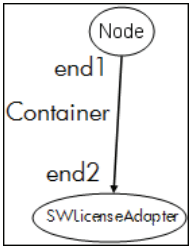
注意：请不要在任何版本的 Microsoft Corporation Notepad 中编辑 **orm.xml** 文件。请使用 Notepad++、 UltraEdit 或其他第三方文本编辑器。

---

2 根据要映射的数据实体对文件进行更改。有关详细信息，请参阅以下示例。

可以在 **orm.xml** 文件中映射以下类型的关系：

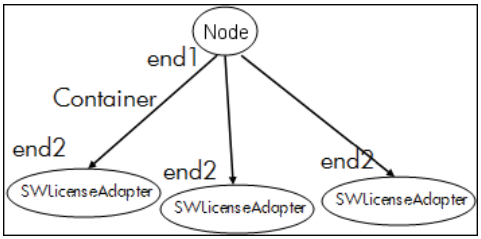
➤ 一对一：



此类关系的代码为：

```
<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>
```

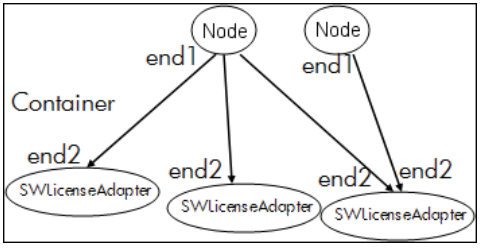
➤ 多对一：



此类关系的代码为：

```
<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>
```

➤ 多对多：



此类关系的代码为：

```
<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>
```

有关命名约定的详细信息，请参阅 “命名约定”（第 183 页）。

数据模型和 RDBMS 之间的实体映射示例：

注意：以下示例中省略了不必配置的属性。

- CMDB CIT 的类：
  <entity class="generic_db_adapter.node">
- RDBMS 中表的名称：
  <table name="Device"/>
- RDBMS 表中唯一标识符的列名称：
  <column name="Device ID"/>
- CMDB CIT 中的属性名称：
  <basic name="name">
- 外部数据源中的表字段的名称：
  <column name="Device_Name"/>
- 在“创建 CI 类型”（第 135 页）中创建的新 CIT 的名称：
  <entity class="generic_db_adapter.MyAdapter">
- RDBMS 中的相应表的名称：
  <table name="SW_License"/>
- RDBMS 中的唯一标识：

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

- CMDB CIT 中的属性名称和 RDBMS 中相应属性的名称：

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

数据模型和 RDBMS 之间的关系映射示例：

- CMDB 关系的类：

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

- 作为关系执行位置的 RDBMS 表的名称：

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

- RDBMS 中的唯一 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>
```

- 关系类型和 CMDB CIT：

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

- RDBMS 中的主键字段和外键字段：

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

配置 reconciliation_types.txt 文件

在文本编辑器中打开 **reconciliation_types.txt** 文件。

有关详细信息，请参阅 “reconciliation_types.txt 文件”（第 190 页）。

配置 reconciliation_rules.txt 文件

在此步骤中，将定义适配器调和 CMDB 和 RDBMS 时所用的规则 （仅在使用映射引擎的情况下执行，以实现与 8.x 版的向后兼容）：

1 在文本编辑器中打开 **META-INF\reconciliation_rules.txt**。

2 根据要映射的 CIT 对文件进行更改。例如，要映射节点 CIT，请使用以下表达式：

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

注意：

- 如果数据库中的数据区分大小写，请不要删除控制字符 (^)。
- **检查每个左方括号是否具有相匹配的右方括号。**

有关详细信息，请参阅 “reconciliation_rules.txt 文件 （用于向后兼容）”（第 190 页）。

实现插件

本任务描述了如何使用插件实施和部署常规 DB 适配器。

注意：在编写适配器的某个插件之前，请确保已完成 “准备适配器包”（第 137 页）中所述的所有必需步骤。

---

1 将 UCMDB 服务器安装目录中的以下 jar 文件复制到开发类路径：

- 从 **tools\adapter-dev-kit** 文件夹复制 **db-interfaces.jar** 文件和 **db-interfaces-javadoc.jar** 文件。
- 从 **\tools\adapter-dev-kit\SampleAdapters\production-lib** 文件夹复制 **federation-api.jar** 文件和 **federation-api-javadoc.jar** 文件。

注意：有关开发插件的更多信息，可查看 **db-interfaces-javadoc.jar** 和 **federation-api-javadoc.jar** 文件以及联机文档：

- **C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\doc_lib\DevRef_guide\DBAdapterFramework_JavaAPI\index.html**
- **C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\doc_lib\DevRef_guide\Federation_JavaAPI\index.html**

---

2 编写用于实施插件的 Java 接口的 Java 类。这些 Java 接口在 **db-interfaces.jar** 文件中进行定义。下表指定了必须为每个插件实施的接口：

插件类型	接口名称	方法
同步整个拓扑	FcmdbPluginForSyncGetFull Topology	getFullTopology
同步更改	FcmdbPluginForSyncGet ChangesTopology	getChangesTopology
同步布局	FcmdbPluginForSyncGet Layout	getLayout
检索受支持的查询	FcmdbPluginForSyncGet SupportedQueries	getSupportedQueries
更改 TQL 查询的定义和结果	FcmdbPluginGetTopology CmdbFormat	getTopologyCmdbFormat
更改 CI 的布局请求	FcmdbPluginGetCIsLayout	getCisLayout
更改链接的布局请求	FcmdbPluginGetRelations Layout	getRelationsLayout

插件的类必须具有公共的默认构造函数。此外，所有接口都提供一种称为 initPlugin 的方法。在调用其他任何方法之前，必定会先调用此方法，用于初始化适配器以及包含适配器的环境对象。

3 在编译 Java 代码之前，请确保类路径中包含联合 SDK JAR 和常规 DB 适配器 JAR。联合 SDK 是 **federation_api.jar** 文件，此文件位于 **C:\hp\UCMDB\UCMDBServer\lib** 目录中。

4 将类打包为一个 jar 文件并将其放置到 adapterCode\< 适配器名称 > 文件夹下，然后对其进行部署。

使用位于适配器的 **\META-INF** 文件夹中的 **plugins.txt** 文件配置插件。

以下是一个 DDMi 适配器文件示例：

```
# 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]
```

图例：

- 注释行。

[< 适配器类型 >] - 特定适配器类型的定义部分的开头。

每个 [< 适配器类型 >] 下可有一个空行，表示不存在关联的插件类，或者不存在可以列出的插件类的完全限定名称。

5 使用新 jar 文件和已更新的 **plugins.xml** 文件将适配器打包。包中的文件余数必须与基于常规 DB 适配器的所有适配器中的文件余数相同。

部署适配器

1 在 UCMDB 中访问“程序包管理器”。有关详细信息，请参阅《HP Universal CMDB 管理指南》中的 ““包管理器”页面”。

2 单击“将包部署到服务器 （从本地磁盘）”图标，然后浏览到适配器包。选择包，单击“打开”，然后单击“部署”，以便在“包管理器”中显示此包。

3 在列表中选择所需的包，然后单击“查看包资源”图标以验证程序包管理器是否可以识别此包。

编辑适配器

创建并部署适配器之后，可以在 UCMDB 中对适配器进行编辑。有关详细信息，请参阅 “适配器管理” （第 101 页）。

创建集成点

在此步骤中，将检查联合是否可正常工作，即连接是否有效以及 XML 文件是否有效。但是，此项检查不会验证 XML 是否将映射到 RDBMS 中的正确字段。

1 在 UCMDB 中，访问 “集成工作室”（“数据流管理” > “集成工作室”）。

2 创建集成点。有关详细信息，请参阅《HP Universal CMDB 数据流管理指南》中的 ““创建新的集成点 / 编辑集成点”对话框”。

“联合”选项卡将显示可以使用此集成点联合的所有 CIT。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 ““联合”选项卡”。

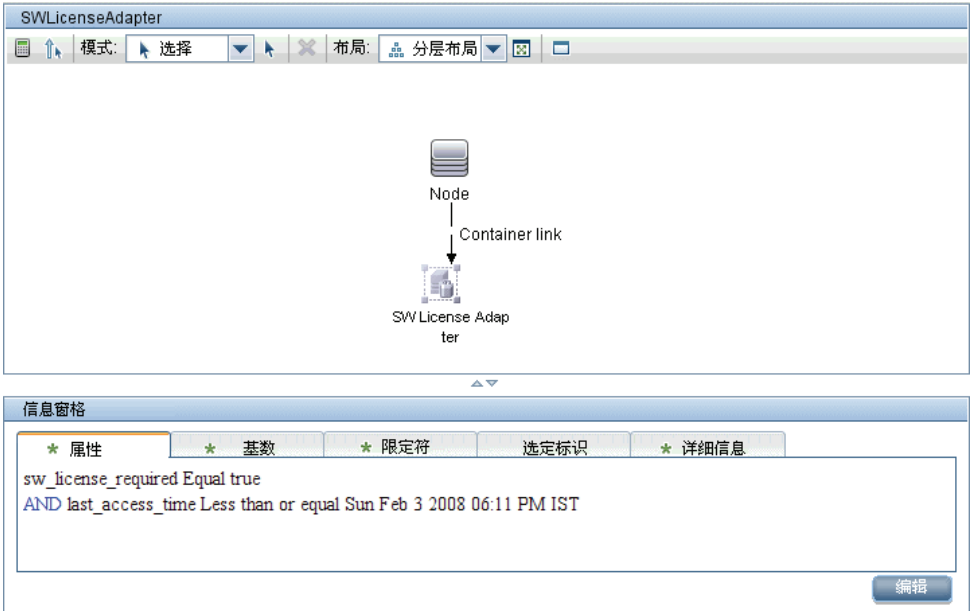
创建视图

在此步骤中将创建一个视图，以便查看 CIT 实例。

1 在 UCMDB 中，访问 “建模工作室”（“建模” > “建模工作室”）。

2 创建视图。有关详细信息，请参阅 《HP Universal CMDB 建模指南》中的 “创建基于模板的视图”。

3 可以向 TQL 添加条件 （例如，上次访问时间超过六个月）：



计算结果

在此步骤中，可以检查结果。

1 在 UCMDB 中，访问 “建模工作室”（“建模” > “建模工作室”）。

2 打开视图。

3 单击 “计算查询结果计数”按钮，以计算结果。

4 单击 “预览”按钮，以查看视图中的 CI。

查看结果

在此步骤中，可以查看结果并调试过程中遇到的问题。例如，如果视图中未显示任何内容，则请检查 **orm.xml** 文件中的定义、删除关系属性并重新加载适配器。

1 在 UCMDB 中，访问 “IT 世界管理器”（“建模” > “IT 世界管理器”）。

2 选择 CI。

“属性”选项卡将显示联合的结果。

查看报告

在此步骤中，可以查看拓扑报告。有关详细信息，请参阅 《HP Universal CMDB 建模指南》中的 “拓扑报告概述”。

启用日志文件

要了解计算流程和适配器生命周期以及查看调试信息，可以查阅日志文件。有关详细信息，请参阅 “适配器日志文件”（第 212 页）。

使用 Eclipse 在 CIT 属性和数据库表之间进行映射

警告：本过程适用于精通内容开发的用户。如有任何疑问，请联系 HP Software 支持。

本任务描述了如何安装和使用 J2EE 版 Eclipse 随附的 JPA 插件来执行以下操作：

- 在 CMDB 类属性和数据库表列之间启用图形映射。
- 启用对映射文件 (orm.xml) 的手动编辑功能，同时确保其正确性。正确性检查包括语法检查，以及验证类属性和已映射的数据库表列的描述是否正确。
- 将映射文件部署到 CMDB 服务器并查看错误，以进一步检查正确性。
- 在 CMDB 服务器上定义示例查询，并直接从 Eclipse 运行此查询，以测试映射文件。

本任务包括以下步骤：

- “先决条件”（第 156 页）
- “安装”（第 156 页）
- “准备工作环境”（第 157 页）
- “创建适配器”（第 160 页）
- “配置 CMDB 插件”（第 160 页）
- “导入 UCMDB 类模型”（第 162 页）
- “生成 ORM 文件 - 将 UCMDB 类映射到数据库表”（第 163 页）
- “映射 ID”（第 165 页）
- “映射属性”（第 166 页）

1 先决条件

在要运行 Eclipse 的计算机上安装 **Java Runtime Environment (JRE) 6 Update 7**。可从以下网址获取此程序：**http://java.sun.com/javase/downloads/index.jsp**。

本过程适用于 Java 5 （或更高版本） Runtime Environment。

2 安装

a 从 **<http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/ganymede/SR1/eclipse-jee-ganymede-SR1-win32.zip>** 下载 **Eclipse IDE for Java EE Developers**，并将其提取到本地文件夹 （例如 **C:\Program Files\eclipse**）。

b 将 **C:\hp\UCMDB\UCMDBServer\tools\db-adapter-eclipse-plugin\bin** 中的 **com.hp.plugin.import_cmdb_model_1.0.jar** 复制到 **C:\Program Files\Eclipse\plugins**。

c 启动 **C:\Program Files\Eclipse\eclipse.exe** （要求至少具有 Java 5 Runtime Environment）。如果显示一条消息指出找不到 Java 虚拟机，则可通过以下命令行启动 **eclipse.exe**：

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

3 准备工作环境

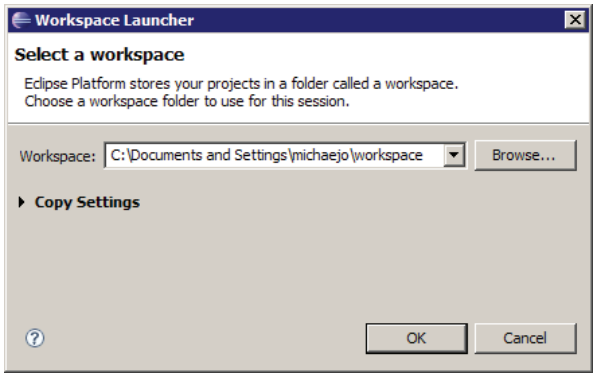
在此步骤中，可以设置工作区、数据库、连接和驱动程序属性。

a 将文件 **workspaces_gdb.rar** 从 **C:\hp\UCMDB\UCMDBServer\tools\db-adapter-eclipse-plugin\workspace** 提取到 **C:\Documents and Settings\All Users\workspaces**。

注意：必须使用正确的文件夹路径。如果将该文件解压缩到错误的路径或者不解压缩该文件，则无法完成此过程。

---

b 在 Eclipse 中，选择 “File” > “Switch Workspace” > “Other”：

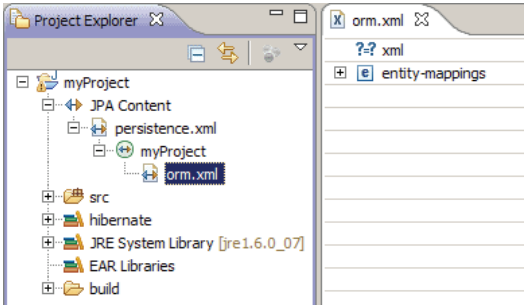


如果使用的是：

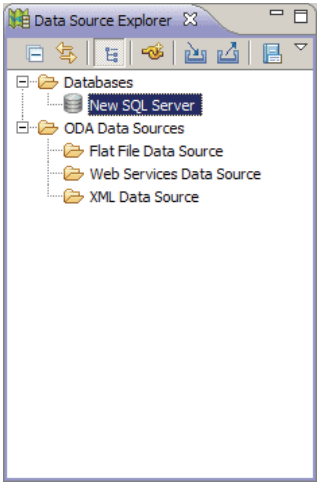
- SQL Server，则选择文件夹 **C:\Documents and Settings\All Users\workspace_gdb_sqlserver**。
- MySQL，则选择文件夹 **C:\Documents and Settings\All Users\workspace_gdb_mysql**。
- Oracle，则选择文件夹 **C:\Documents and Settings\All Users\workspace_gdb_oracle**.

c 单击 “OK”。

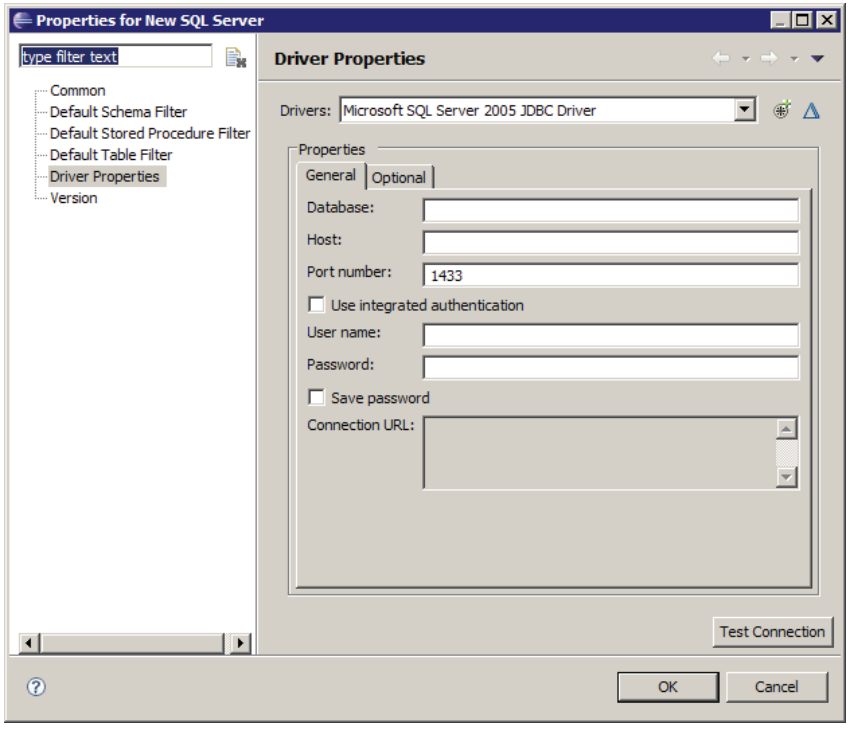
d 在 Eclipse 中，显示“Project Explorer”视图，并选择“<活动项目>”>“JPA Content”>“persistence.xml”>“<活动项目名称>”>“orm.xml”。



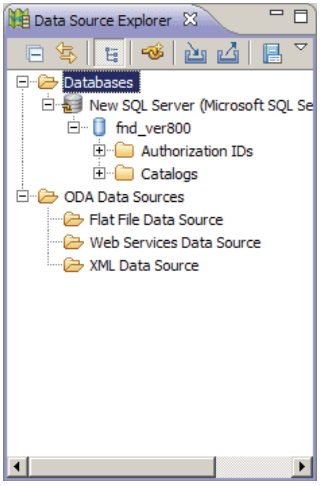
e 在“Data Source Explorer”视图（左下角窗格）中，右键单击数据库连接并选择“Properties”菜单。



f 在 “Properties for < 连接名称 >” 对话框中，选择 “Common”，然后选中 “Connect every time the workbench is started” 复选框。选择 “Driver Properties” 并填入连接属性。单击 “Test Connection” 并验证连接是否有效。单击 “OK”。



g 在 “Data Source Explorer” 视图中，右键单击数据库连接，然后单击 “Connect”。将在数据库连接图标下方显示一个含有数据库架构和数据库表的树。

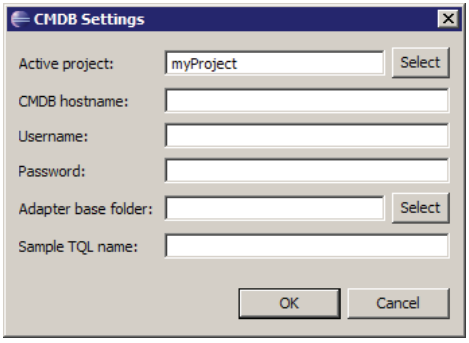


4 创建适配器

根据 “步骤 1：创建适配器”（第 40 页）中的说明创建适配器。

5 配置 CMDB 插件

a 在 Eclipse 中，单击 “UCMDB” > “Settings” 以打开 “CMDB Settings” 对话框：



b 如果尚未选择任何活动项目，则选择新建的 JPA 项目作为活动项目。

c 输入 CMDB 主机名，例如 **localhost** 或 **labm1.itdep1**。无需在地址中包含端口号或 “http://” 前缀。

d 填入用于访问 CMDB API 的用户名和密码，通常为 **admin**/**admin**。

e 确保将 CMDB 服务器上的 **C:\hp** 文件夹映射为网络驱动器。

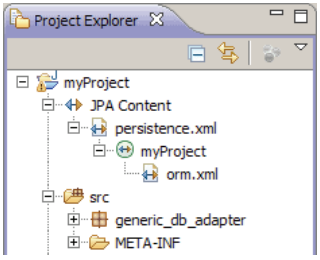
f 在 **C:\hp** 下选择相关适配器的基础文件夹。基础文件夹包含 **dbAdapter.jar** 文件和 **META-INF** 子文件夹，其路径为 **C:\hp\UCMDB\UCMDBServer\runtime\fcmdb\CodeBase\< 适配器名称 >**。确保结尾处没有反斜杠 (\)。

6 导入 UCMDB 类模型

在此步骤中，可以选择要映射为 JPA 实体的 CIT。

a 单击 “UCMDB” > “Import CMDB Class Model” 以打开 “CI Types Selection” 对话框：

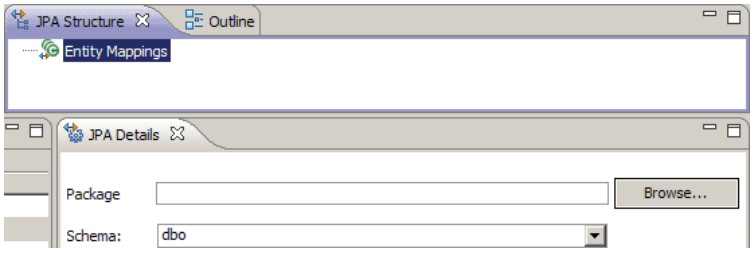
b 选择要映射为 JPA 实体的 CI 类型。单击 “OK”。此时， CI 类型将导入为 Java 类。验证这些类是否出现在活动项目的 **src** 文件夹下：



7 生成 ORM 文件 - 将 UCMDB 类映射到数据库表

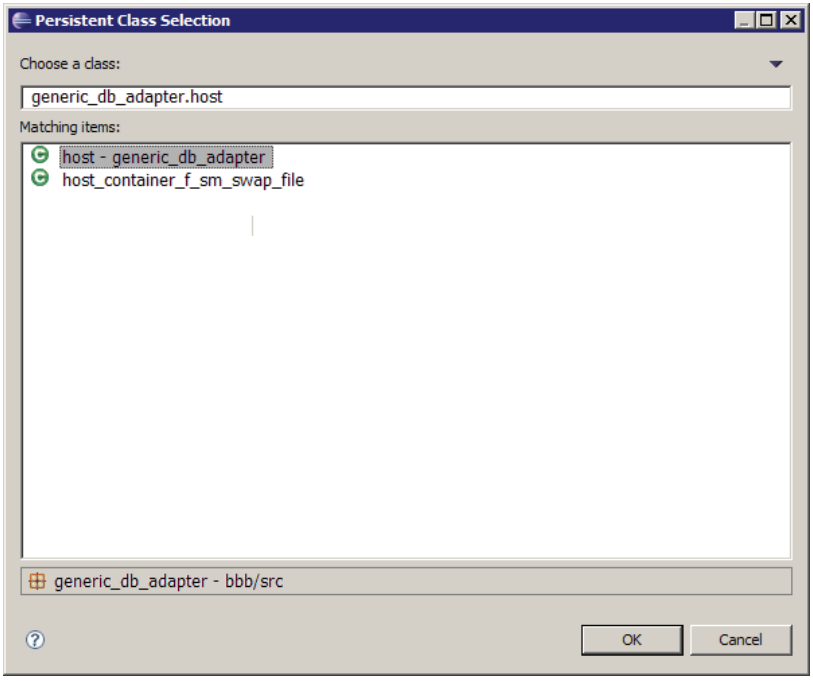
在此步骤中，可以将上一步中导入的 Java 类映射到数据库表。

a 确保 DB 连接已连接。在 “Project Explorer” 中右键单击活动项目 （默认情况下称为 myProject）。选择 JPA 视图，选中 “Override default schema from connection” 复选框，并选择相关数据库架构。单击 “OK”。



b 映射 CIT：在 “JPA Structure” 视图中，右键单击 “Entity Mappings” 分支并选择 “Add Class”。此时，将打开 “Add Persistent Class” 对话框。请不要更改 “Map as” 字段 （**实体**）。

c 单击 “Browse”，并选择要映射的 UCMDB 类 （所有 UCMDB 类都属于 **generic_db_adapter** 包）。

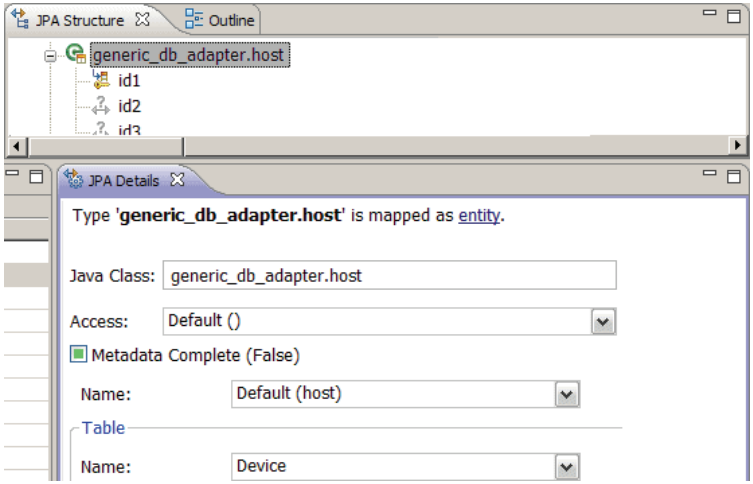


d 在这两个对话框中单击 “OK”。此时，所选类将显示在 “JPA Structure” 视图的 “Entity Mappings” 分支下。

注意：如果显示的实体没有属性树，则在 “Project Explorer” 视图中右键单击活动项目。选择 “Close”，然后选择 “Open”。

---

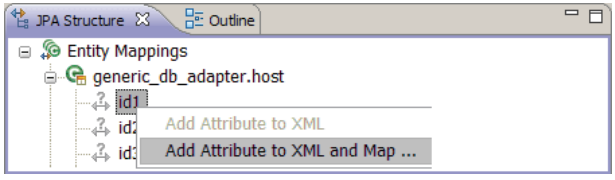
e 在“JPA Details”视图中，选择要将 UCMDB 类映射到的主数据库表。将所有其他字段保留不变。



8 映射 ID

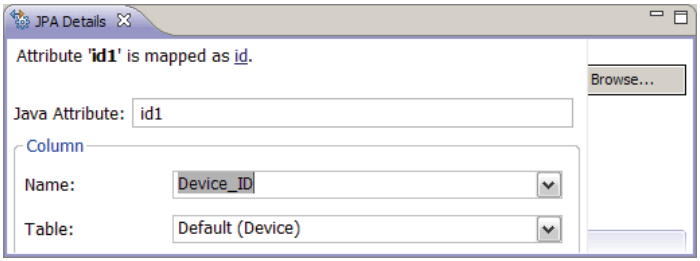
按照 JPA 标准，每个持久类都必须至少包含一个 ID 属性。对于 UCMDB 类，最多可以映射三个属性作为 ID。可能的 ID 属性为 **id1**、**id2** 和 **id3**。要映射 ID 属性，请执行以下操作：

a 在“JPA Structure”视图的“Entity Mappings”分支下展开相应的类，右键单击相关属性 （例如 **id1**），然后选择 “Add Attribute to XML and Map...”：



b 此时，将打开 “Add Persistent Attribute”对话框。在 “Map as”字段中选择 “Id”，然后单击 “OK”。

c 在 “JPA Details” 视图中，选择要将 ID 字段映射到的数据库表列。



9 映射属性

在此步骤中，可以将属性映射到数据库列。

a 在 “JPA Structure” 视图的 “Entity Mappings” 分支下展开相应的类，右键单击相关属性 （例如 **host_hostname**），然后选择 “Add Attribute to XML and Map...”：

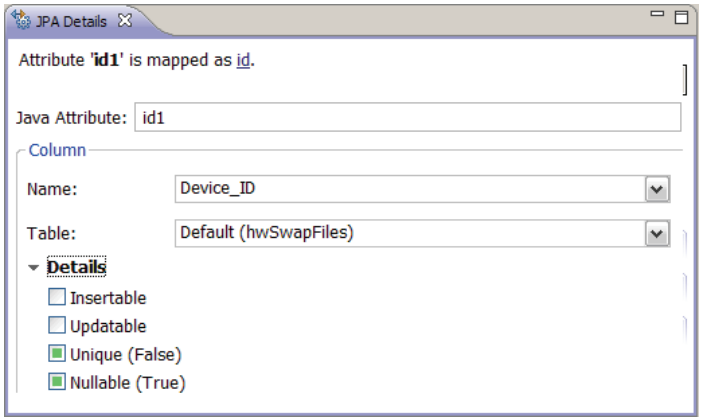
b 此时，将打开 “Add Persistent Attribute” 对话框。在 “Map as” 字段中选择 “Basic”，然后单击 “OK”。

c 在 “JPA Details” 视图中，选择要将属性字段映射到的数据库表列。

10 映射有效链接

执行步骤 “b”（第 163 页）中描述的步骤，以映射一个表示有效链接的 UCMDB 类。每个类名称的结构如下：**<end1 实体名称 >_< 链接名称 >_<end2 实体名称 >**。例如，可以用一个名为 **generic_db_adapter.host_contains_location** 的 Java 类表示主机和位置之间的 **Contains** 链接。有关详细信息，请参阅 “reconciliation_rules.txt 文件（用于向后兼容）”（第 190 页）。

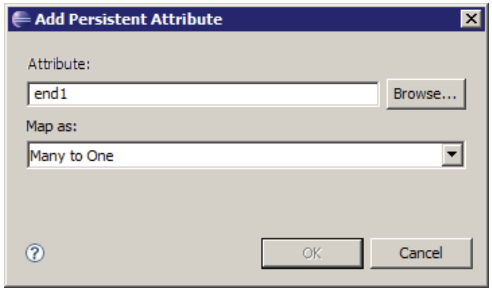
a 映射链接类的 ID 属性，如 “映射 ID”（第 165 页）所述。对于每个 ID 属性，在 “JPA Details” 视图中展开 “Details” 复选框组，然后清除 “Insertable” 和 “Updatable” 复选框。



b 要映射链接类的 **end1** 和 **end2** 属性，请执行以下操作 （适用于链接类的所有 **end1** 和 **end2** 属性）：

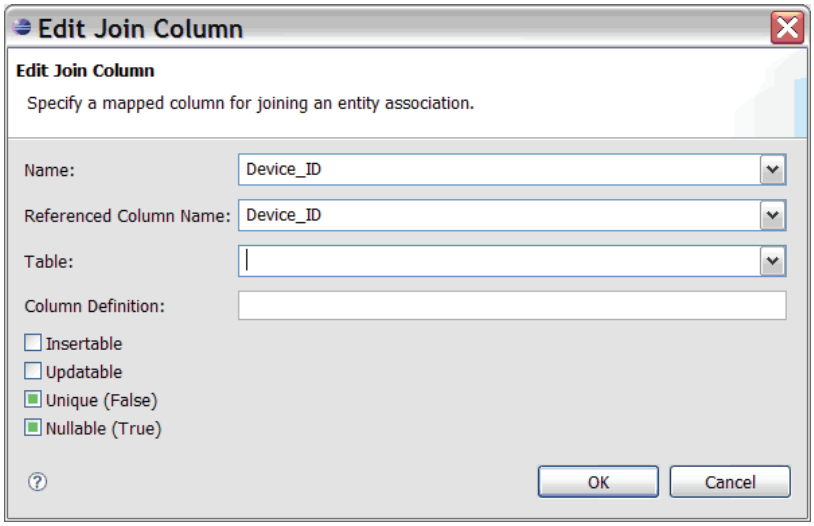
➤ 在 “JPA Structure” 视图的 “Entity Mappings” 分支下展开相应的类，并右键单击相关属性（例如 **end1**），然后选择 “Add Attribute to XML and Map...”：

- 在“Add Persistent Attribute”对话框的“Map as”字段中，选择“Many to One”或“One to One”。



- 如果指定的 **end1** 或 **end2** CI 可以包含多个此类链接，则选择“Many to One”。否则，选择“One to One”。例如，对于 **host_contains_ip** 链接，**host** 端将按“Many to One”进行映射（因为一个主机可以有多个 IP），而 **ip** 端将按“一对一”进行映射（因为一个 IP 只能有一个主机）。
- 在“JPA Details”视图中，选择“Target entity”，例如 **generic_db_adapter.host**。

- 在“JPA Details”视图的“Join Columns”部分中，检查“Override Default”。单击“Edit”。在“Edit Join Column”对话框中，选择链接数据库表的外键列（此列指向 **end1/end2** 目标实体表中的条目）。如果 **end1/end2** 目标实体表中引用的列名称已映射到其 ID 属性，则保留“Referenced Column Name”不变。否则，选择外键列所指向的列的名称。清除“Insertable”和“Updatable”复选框，然后单击“OK”。



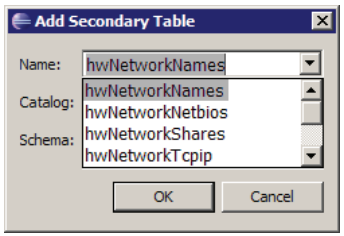
- 如果 **end1/end2** 目标实体具有多个 ID，则单击“Add”按钮以添加其他连接列，并按照上一步中描述的方式对其进行映射。

11 生成 ORM 文件 – 使用次级表

通过 JPA 可将 Java 类映射到多个数据库表。例如，可将 **Host** 映射到 **Device** 表以启用大多数属性的持久性，还可以将其映射到 **NetworkNames** 表以启用 **host_hostName** 的持久性。在这种情况下， **Device** 是主表，而 **NetworkNames** 是次级表。可以定义任意数量的次级表。唯一条件是主表和次级表的条目之间必须存在一对一关系。

12 定义次级表

在“JPA Structure”视图中选择相应的类。在“JPA Details”视图中，访问“Secondary Tables”部分，然后单击“Add”。在“Add Secondary Table”对话框中，选择相应的次级表。将其他字段保留不变。



如果主表和次级表的主键不同，则在“JPA Details”视图的“Primary Key Join Columns”部分中配置连接列。

13 将属性映射到次级表

要将类属性映射到次级表的字段，请执行以下操作：

a 按照“映射属性”（第 166 页）所述映射属性。

b 在“JPA Details”视图“Column”部分的“Table”字段中，选择次级表名称，以替换默认值。

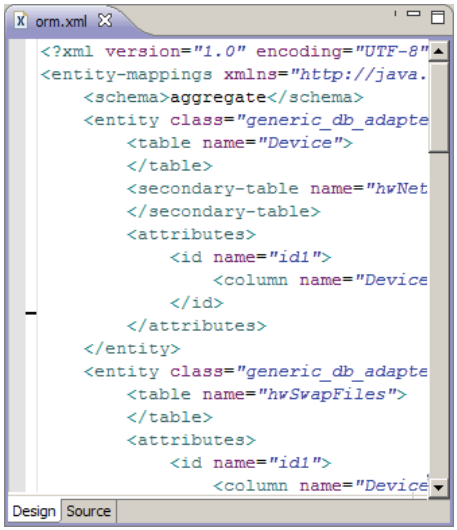
14 将现有 ORM 文件作为基础

要将现有 orm.xml 文件作为要开发的文件的基础，请执行以下步骤：

a 验证是否已将现有 **orm.xml** 文件中映射的所有 CIT 导入 Eclipse 活动项目中。

b 在现有文件中选择和复制所有实体映射或部分实体映射。

c 在 Eclipse JPA 透视中选择 **orm.xml** 文件的 “Source”选项卡。



d 将复制的所有实体映射粘贴到经过编辑的 **orm.xml** 文件的 < **实体映射** > 标记下（位于 < **架构** > 标记下）。确保按 “b”（第 163 页）中所述配置架构标记。此时，已粘贴的所有实体都将显示在 “JPA Structure”视图中。从此时起，既可以通过图形对映射进行编辑，又可以通过 **orm.xml** 文件的 xml 代码对映射进行手动编辑。

e 单击 “Save”。

15 从适配器导入现有 ORM 文件

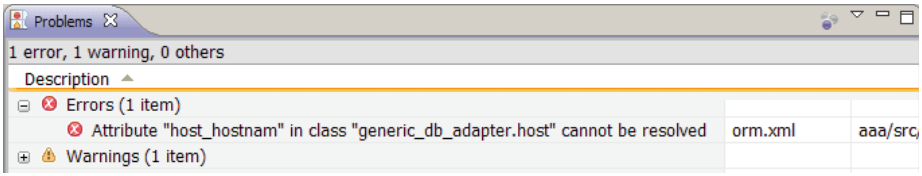
如果适配器已存在，则可以使用 Eclipse 插件对其 ORM 文件进行图形方式的编辑。将 ORM 文件导入 Eclipse 中，使用插件对其进行编辑，然后将其重新部署到 UCMDB 计算机中。要导入 ORM 文件，请按 Eclipse 工具栏上的按钮。此时，将显示一个确认对话框。单击 “确定” 。此时 UCMDB 计算机中的 ORM 文件将被复制到 Eclipse 活动项目，并且将从 UCMDB 类模型中导入所有相关类。

如果相关类没有出现在 “JPA Structure”视图中，则右键单击 “Project Explorer”视图中的活动项目，选择 “Close”，然后选择 “Open”。

从此时起，可以使用 Eclipse 对 ORM 文件进行图形方式的编辑，然后将其重新部署到 UCMDB 计算机，如 “将 ORM 文件部署到 CMDB”（第 172 页）所述。

16 检查 ORM 文件的正确性 - 内置的正确性检查

Eclipse JPA 插件将检查是否存在任何错误，并在 **orm.xml** 文件中标记这些错误。将检查语法错误（例如标记名称错误、标记未闭合、ID 丢失），以及映射错误 （例如属性名称或数据库表字段名称错误）。如果存在错误，将在“Problems”视图中显示有关这些错误的描述：



17 创建新集成点

如果此适配器的 CMDB 中不存在任何集成点，则可以在集成工作室中创建一个集成点。有关详细信息，请参阅 《HP Universal CMDB 数据流管理指南》中的 “集成工作室”。

在打开的对话框中填入集成点名称。此时， **orm.xml** 文件将复制到适配器文件夹。将创建一个集成点，并将所有导入的 CI 类型作为此集成点支持的类，但多节点 CIT 除外 （如果在 reconciliation_rules.txt 文件中配置了这些 CI 类型）。有关详细信息，请参阅“reconciliation_rules.txt 文件（用于向后兼容）”（第 190 页）。

18 将 ORM 文件部署到 CMDB

保存 **orm.xml** 文件并将其部署到 UCMDB 服务器，方法是：单击“UCMDB” > “部署 ORM”。此时，**orm.xml** 将复制到适配器文件夹，并将重新加载适配器。操作结果将显示在“操作结果”对话框中。如果重新加载过程中出现任何错误，对话框中将显示 Java 异常堆栈跟踪。如果尚未使用适配器对任何集成点进行定义，则部署时不会检测到任何映射错误。

19 运行示例 TQL 查询

a 使用查询管理器而不是视图管理器对查询进行定义。

b 使用 **GenericDBAdapter** 适配器创建集成点。有关详细信息，请参阅《HP Universal CMDB 数据流管理指南》中的 “‘创建新的集成点 / 编辑集成点”对话框”。

c 在创建适配器期间，验证此集成点是否支持要参与查询的 CI 类型。

d 配置 CMDB 插件时，使用 “设置”对话框中的该实例查询名称。有关详细信息，请参阅 “配置 CMDB 插件”（第 160 页）。

e 单击 “运行 TWL”按钮以运行示例 TQL，并使用新建的 **orm.xml** 文件验证该示例 TQL 是否返回所需结果。

参考

适配器配置文件

本节讨论的文件位于 **C:\hp\UCMDB\UCMDBServer\content\adapters** 文件夹的 **db-adapter.zip** 包中。

本节包括以下主题：

常规配置

- **adapter.conf.** 适配器配置文件。有关详细信息，请参阅“adapter.conf 文件”（第 175 页）。

高级配置

- **orm.xml**。在其中映射 CMDB CIT 和数据库表的“对象 - 关系”映射文件。有关详细信息，请参阅 “orm.xml 文件”（第 179 页）。
- **reconciliation_types.txt**。包含用于配置调和类型的规则。有关详细信息，请参阅 “reconciliation_types.txt 文件”（第 190 页）。
- **reconciliation_rules.txt**。包含调和规则。有关详细信息，请参阅“reconciliation_rules.txt 文件 （用于向后兼容）”（第 190 页）。
- **transformations.txt**。转换文件，在此文件中指定要用于将 CMDB 值转换为数据库值，或者将数据库值转换为 CMDB 值的转换器。有关详细信息，请参阅“transformations.txt 文件”（第 192 页）。

- **Discriminator.properties**。此文件将每个支持的 CI 类型映射到一个相应值列表（以逗号分隔）。有关详细信息，请参阅“discriminator.properties 文件”（第 194 页）。
- **Replication_config.txt**。此文件包含以逗号分隔的 CI 和关系类型列表，这些 CI 和关系类型的属性条件受复制插件支持。有关详细信息，请参阅“replication_config.txt 文件”（第 196 页）。
- **Fixed_values.txt**。通过此文件，可以为某些 CIT 的特定属性配置固定值。有关详细信息，请参阅“fixed_values.txt 文件”（第 196 页）。

Hibernate 配置

- **persistence.xml**。用于覆盖现成的 Hibernate 配置。有关详细信息，请参阅“persistence.xml 文件”（第 193 页）。

简单配置

- **simplifiedConfiguration.xml**。用于替换 **orm.xml**、**transformations.txt** 和 **reconciliation_rules.txt** 的配置文件，以提供较少的功能。有关详细信息，请参阅“simplifiedConfiguration.xml 文件”（第 176 页）。

adapter.conf 文件

本文件包含以下设置：

- **use.simplified.xml.config=false. true**：使用 simplifiedConfiguration.xml。

注意：如果使用此文件，则表示 orm.xml、transformations.txt 和 reconciliation_rules.txt 将替换为具有较少功能的文件。

- **dal.ids.chunk.size=300**。请不要更改此值。

➤ **dal.use.persistence.xml=false.true**：适配器从 persistence.xml 读取 Hibernate 配置。

注意：建议不要覆盖 Hibernate 配置。

---

simplifiedConfiguration.xml 文件

本文件用于将 UCMDB 类简单地映射到数据库表。要访问模板以便编辑文件，请导航到 “适配器管理” > “db-adapter” > “配置文件”。

本节包括以下主题：

➤ “simplifiedConfiguration.xml 文件模板”（第 176 页）

➤ “限制”（第 178 页）

simplifiedConfiguration.xml 文件模板

CMDB-class-name 属性是多节点类型 （TQL 中与联合 CIT 连接的节点）：

```
<?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。可以使用一个或两个节点完成调和。在此示例中，调和过程使用了两个节点。

connected-node-CMDB-class-name。调和 TQL 中需要的次要类类型。

CMDB-link-type。调和 TQL 中需要的关系类型。

link-direction。调和 TQL 中关系的方向 （从 node 到 ip_address 或从 ip_address 到 node）：

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

调和表达式的形式是 OR，并且每个 OR 都包含 AND。

is-ordered。确定调和是以指令形式还是以常规 OR 比较来完成。

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

如果从主类 （多节点）中检索到调和属性，则使用 **attribute** 标记，否则使用 **connected-node-attribute** 标记。

ignore-case.true. 将 UCMDB 类模型中的数据与 RDBMS 中的数据进行比较时，不区分大小写：

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

列名称是外键列 （此列包含指向多节点主键列的值）的名称。

如果多节点主键列由若干个列组成，则需要若干个外键列，一个外键列对应一个主键列。

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

如果主键列很少，则复制此列。

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

from-CMDB-converter 和 **to-CMDB-converter** 属性是用于实现以下接口的 Java 类：

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

如果 CMDB 中的值与数据库中的值不同，则使用这些转换器。例如， CMDB 中的节点名称具有后缀 mer.com。

在此示例中，GenericEnumTransformer 用于根据括号内的 XML 文件转换枚举器 (**generic-enum-transformer-example.xml**)：

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

限制

- 只能用于映射单节点 TQL 查询（在数据库源中）。例如，您既可以运行**节点 > 票证** TQL 查询，又可以运行**票证** TQL 查询。要从数据库获取节点的层次结构，必须使用高级 **orm.xml** 文件。
- 仅支持一对多关系。例如，您可以在每个节点上获取一个或多个票证，但无法获取属于多个节点的票证。
- 无法将同一个类与不同类型的 CMDB CIT 相连接。例如，如果定义为将**票证**连接到**节点**，则票证将无法连接到**应用程序**。

orm.xml 文件

本文件用于将 CMDB CIT 映射到数据库表。

用于创建新文件的模板位于 **C:\hp\UCMDB\UCMDBServer\runtime\fcmdb\CodeBase\GenericDBAdapter\META-INF** **META-INF** 目录中。

要编辑已部署适配器的 XML 文件，请导航到 “Adapter Management” > “db-adapter” > “配置文件”。

本节包括以下主题：

- “orm.xml 文件模板”（第 179 页）
- “多个 ORM 文件”（第 183 页）
- “命名约定”（第 183 页）
- “使用内嵌式 SQL 语句而非表名称”（第 183 页）
- “orm.xml 架构”（第 184 页）

orm.xml 文件模板

```
<?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>
```

不要更改包名称。

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

entity。 CMDB CIT 名称。它是多节点实体。

确保 **class** 包括 **generic_db_adapter.** 前缀。

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

如果将实体映射到多个表中，则使用次级表。

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

对于具有区分器的单个表格继承，请使用以下代码：

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

具有标记 **id** 的属性是主键列。确保这些主键列的命名约定是 **idX**（id1、id2 等），其中 **X** 是主键中的列索引。

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

仅更改主键的列名称。

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

basic。用于声明 CMDB 属性。确保仅编辑 **name** 和 **column_name** 属性。

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

对于具有区分器的单表格继承，请按照以下指令对扩展类进行映射：

```
    <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>
```

以下示例显示了一个没有前缀的 CMDB 属性名称：

```
            <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>
```

这是一个关系实体。命名约定为 **end1Type_linkType_end2Type**。在以下示例中， **end1Type** 是 **node**， **linkType** 是 **composition**。

```
    <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>
```

目标实体是此属性当前指向的实体。在以下示例中， **end1** 将映射到 **node** 实体。

many-to-one。可将多种关系连接到一个节点。

join-column。含有 **end1** ID （目标实体 ID）的列。

referenced-column-name。目标实体 (**node**) 中的列名称，含有在连接列中使用的 ID。

```
            <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。一种关系只能连接到一个 **[CMDB_class_name]**。

```
            <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>
```

多个 ORM 文件

支持多个映射文件。每个映射文件的名称应以 **orm.xml** 结尾。应将所有映射文件放置在 META-INF 文件夹下。

命名约定

- 在每个实体中，类属性必须与具有 generic_db_adapter 前缀的名称属性相匹配。
- 主键列名称的形式必须为 **idX**，其中 **X = 1**、 **2...**，具体取决于主键在表中的编号。
- 属性名称必须与类属性名称匹配，包括大小写。
- 关系名称的形式为 end1Type_linkType_end2Type。
- CMDB CIT（也是 Java 中的保留字）必须具有 **gdba_** 前缀。例如，对于 CMDB CIT **goto**， ORM 实体的名称将为 **gdba_goto**。

使用内嵌式 SQL 语句而非表名称

您可以将实体映射到内嵌式 select 子句而非数据库表。这相当于在数据库中定义一个视图并将实体映射到此视图。例如：

```
    <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)"/>
```

在上述示例中，节点属性将映射到列 id1、 name 和 host_os，而不是 id、 name 和 os。

存在以下限制：

- 只有在将 Hibernate 用作 JPA 提供程序时，才能使用内嵌式 SQL 语句。
- 必须在内嵌式 SQL select 子句两边加上圆括号。
- **<schema>** 元素不得出现在 **orm.xml** 文件中。如果使用的是 Microsoft SQL Server 2005，则意味着所有表名称必须具有前缀 dbo.，而不能通过 <schema>dbo</schema> 对这些表名称进行全局定义。

orm.xml 架构

下表说明了 **orm.xml** 文件的常见元素。可以在 http://java.sun.com/xml/ns/persistence/orm_1_0.xsd 中找到完整构架。此列表并不完整，主要说明了适用于常规数据库适配器的标准 Java Persistence API 的特定行为。

元素		属性
名称和路径	**描述**	
entity-mappings	实体映射文档的根元素。此元素必须与 GDBA 示例文件中指定的元素完全相同。	
description (entity-mappings)	对实体映射文档的自由文本描述。可选。	

元素		属性
名称和路径	**描述**	
package (entity-mappings)	要包含映射类的 Java 包的名称。应始终包含文本 generic_db_adapter。	**名称**。 name **描述。**此实体将映射到的 UCMDB CI 类型的名称。如果此实体映射到 CMDB 中链接，则此实体的名称格式为 <end_1>_<link_name>_<end_2>。例如，node_composition_cpu 定义将映射到节点和 CPU 之间的复合链接的实体。如果 CI 类型的名称与无包前缀的 Java 类的名称相同，则可以忽略此字段。 **是否必需。**可选 **类型。**字符串
		名称。 class **描述。**要为此 DB 实体创建的 Java 类的完全限定名。Java 类的包的名称应与 package 元素中指定的名称相同。您可以不将 Java 保留字（如接口或开关）用作类名称。但是，可以将前缀 gdba_ 添加到名称，这样接口将变成 generic_db_adapter.gdba_interface。 **是否必需。**必需 **类型。**字符串

元素		属性
名称和路径	**描述**	
table (entity-mappings > entity)	此元素用于定义 DB 实体的主表。只能出现一次。必需。	**名称**。 name **描述。**主表的名称。如果表的名称不包含它所属的架构，则只能在用于创建集成点的用户架构中搜索此表。此属性还可以是任何有效 SELECT 语句。如果是 SELECT 语句，则必须在其两侧加上圆括号。 **是否必需。**必需 **类型。**字符串
secondary-table (entity-mappings > entity)	此元素可用于定义 DB 实体的次级表。必须通过一对一关系将此表与主表连接。可以定义多个次级表。可选。	**名称**。 name **描述。**次级表的名称。如果表的名称不包含它所属的架构，则只能在用于创建集成点的用户架构中搜索此表。此属性还可以是任何有效 SELECT 语句。如果是 SELECT 语句，则必须在其两侧加上圆括号。 **是否必需。**必需 **类型。**字符串
primary-key-join-column (entity-mappings > entity > secondary-table)	如果不使用具有相同名称的字段连接次级表和主表，则此元素将定义次级表中需要与主表的主键字段相连接的主键字段的名称。	**名称**。 name **描述。**次级表中的主键字段的名称。如果此元素不存在，则假定主键字段的名称与主表的主键字段名称相同。 **是否必需。**可选 **类型。**字符串

元素		属性
名称和路径	**描述**	
inheritance (entity-mappings > entity)	如果当前实体是某组 DB 实体的父实体，则使用此元素可以对其进行标记。此元素可选。	**名称**。 strategy **描述。**定义在 DB 中实现继承的方式。 **是否必需。**必需 **类型。**以下值之一： ➤ SINGLE_TABLE – 此实体和所有子实体位于同一个表中。 ➤ JOINED – 子实体位于联接的表中。 ➤ TABLE_PER_CLASS – 每个实体完全通过一个单独的表进行定义。
discriminator-column (entity-mappings > entity)	如果继承类型为 SINGLE_TABLE，则此元素将定义用于确定每行的实体类型的字段名称。	**名称**。 name **描述。**区分器列的名称。 **是否必需。**必需 **类型。**字符串
discriminator-value (entity-mappings > entity)	此元素可以定义继承树中的特定实体类型。此名称必须与在 **discriminator.properties** 文件中为此特定实体类型的值组定义的名称相同。	
attributes (entity-mappings > entity)	某实体的所有属性映射的根元素。	

元素		属性
名称和路径	**描述**	
id (entity-mappings > entity attributes)	此元素可以定义实体的键字段。必须至少定义一个 id 字段。如果存在多个 id 元素，则这些元素的字段将为实体创建一个复合键。必须尽力避免为 CI 实体（而不是链接）创建复合键。	**名称**。 name **描述。** idX 类型的字符串，其中， X 是 1 到 9 之间的一个数字。第一个 id 将标记为 id1，第二个为 id2，依次类推。此名称不是 UCMDB 中键属性的名称。 **是否必需。**必需 **类型。**字符串
basic (entity-mappings > entity attributes)	此元素可以定义表中某非主键字段与某 UCMDB 属性之间的映射。	**名称**。 name **描述。**此字段将映射到的 UCMDB 属性的名称。当前实体要映射到的 UCMDB CI 类型中必须存在此属性。 **是否必需。**必需 **类型。**字符串
column (entity-mappings > entity > attributes > id 或 (entity-mappings > entity > attributes > basic)	定义基本映射表或 id 字段表中列的名称。	**名称**。 name **描述。**字段的名称。 **是否必需。**必需 **类型。**字符串
		名称。 table **描述。**字段所属的表的名称。此表必须是主表或为实体定义的次级表之一。如果忽略该属性，则会假定字段属于主表。 **是否必需。**可选 **类型。**字符串

元素		属性
名称和路径	**描述**	
one-to-one (entity-mappings > entity > attributes)	定义其值位于其他表（两个表通过一对一关系连接）中的列。仅链接实体映射支持该元素，其他 CI 类型不支持该元素。这是用于定义表与 UCMDB 链接之间的映射的唯一方式。	**名称**。 name **描述。**此字段代表的两端中的其中一端。 **是否必需。**必需 **类型。** end1 或 end2
		名称。 target-entity **描述。**末端所引用的实体的名称。 **是否必需。**必需 **类型。**在实体映射文档中定义的实体名称之一
join-column (entity-mappings > entity attributes > one-to-one)	定义用于连接在一对一父元素中定义的目标实体和当前实体的方法。	**名称**。 name **描述。**当前表中将用于执行一对一联接的字段的名称。 **是否必需。**必需 **类型。**字符串
		名称。 name **描述。**联接实体中用于执行联接的字段的名称。如果忽略该属性，则会假定联接表包含一个与在名称属性中定义的字段名称相同的列。 **是否必需。**可选 **类型。**字符串

reconciliation_types.txt 文件

本文件用于配置调和类型。

文件中的每一行代表一个与 TQL 查询中的联合数据库 CIT 相连接的 CMDB CIT。

reconciliation_rules.txt 文件 （用于向后兼容）

如果要在适配器中配置 DBMapppingEngine 后执行调和，则可以使用本文件配置调和规则。如果不使用 DBMappingEngine，则使用常规 UCMDB 调和机制，而不需要配置该文件。

文件中的每一行代表一个规则。例如：

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

多节点使用多节点名称（与 TQL 中的联合数据库连接的 CMDB CIT）进行填充。

该表达式中包含用于确定两个多节点是否相等的逻辑 （一个多节点位于 CMDB 中，另一个位于数据库源中）。

该表达式由 OR 或 AND 组成。

关于表达式部分中属性名称的约定是 [className].[attributeName]。例如，ip_address 类中的 attributeName 以 ip_address.name 的形式写入。

对于有序的匹配 （如果第一个 OR 子表达式返回与多节点不相等的答案，则不比较第二个 OR 子表达式），然后将使用**有序表达式**而不是**表达式**。

要在比较时忽略大小写，请使用控制符号 (^)。

参数 end1_type、 end2_type 和 link_type 仅适用于调和 TQL 包含两个节点而非一个多节点的情形。在这种情况下，调和 TQL 为 end1_type > (link_type) > end2_type。

由于将通过表达式获取相关布局，所以无需添加相关布局。

调和规则的类型

调和规则的形式为 OR 和 AND 条件 . 可以在若干个不同节点上定义这些规则（例如，通过 name from node AND/OR name from ip_address 标识节点）。

可以通过以下选项查找匹配项：

- **Ordered match.** 从左向右读取调和表达式。如果两个 OR 子表达式具有相等的值，则认为这两个子表达式相等。如果两个 OR 子表达式具有不相等的值，则认为这两个子表达式不相等。在任何其他情况下，将无法确定子表达式是否相等，并且需要测试下一个 OR 子表达式以确定是否相等。

  name from node OR from ip_address. 如果 CMDB 和数据源都包含 name 并且相等，则认为这些节点相等。如果它们都包含 name 但不相等，则认为这些节点不相等，而且无需测试 ip_address 的 name。如果 CMDB 或数据源缺失 name of node，则会检查 name of ip_address。

- **常规匹配 .** 如果 OR 子表达式中的其中一个是相等的，则认为 CMDB 和数据源相等。

  name from node OR from ip_address. 如果 name of node 没有匹配项，则将检查 name of ip_address 是否相等。

对于复杂调和，如果在类模型中按照多个含有关系的 CIT（如 node）对调和实体进行建模，则超集节点的映射将包含所有已建模的 CIT 中的所有相关属性。

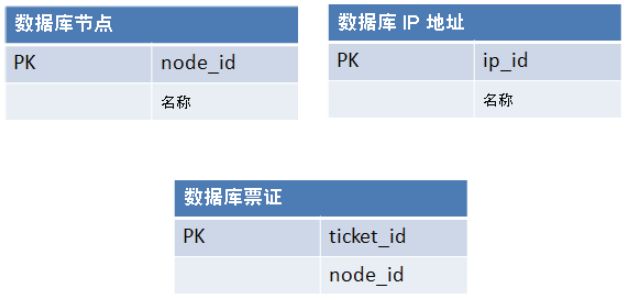
注意：因此，存在一种限制，即数据源中的所有调和属性必须位于共享同一主键的表中。

其他限制条件要求调和 TQL 所包含的节点不得超过两个。例如， node > ticket TQL 具有一个位于 CMDB 中的节点和一个位于数据源中的票证。

要调和结果，则必须从节点和 / 或 ip_address 中检索 name。

如果 name 在 CMDB 中的格式为 *.m.com，则可以使用转换器将这些值从 CMDB 转换到联合数据库，或从联合数据库转换到 CMDB。

数据库票证表中的 node_id 列用于连接实体 （也可以在节点表中实现所定义的关联）：



注意：以上三个表必须属于联合 RDBMS 源而不属于 CMDB 数据库。

transformations.txt 文件

本文件包含所有转换器定义。

格式是每行包含一个新定义。

transformations.txt 文件模板

```
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。在 orm.xml 文件中显示的实体名称。

attribute。在 orm.xml 文件中显示的属性名称。

to_DB_class。实施接口 **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerToExternalDB** 的类的完全限定名。括号中的元素将指定给该类构造函数。使用该转换器可将 CMDB 值转换成数据库值，例如，给每个节点名称附加后缀 **.com**。

from_DB_class。实施 **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerFromExternalDB** 接口的类的完全限定名。括号中的元素将指定给该类构造函数。使用该转换器可将数据库值转换成 CMDB 值，例如，给每个节点名称附加后缀 **.com**。

有关详细信息，请参阅 “现成的转换器”（第 196 页）。

persistence.xml 文件

本文件用于覆盖默认 Hibernate 设置，以及添加对非现成数据库类型的支持（OOB 数据库类型为 Oracle Server、 Microsoft MSSQL Server 和 MySQL）。

如果需要支持新的数据库类型，则必须提供连接池提供程序（默认值是 c3p0）和适用于您的数据库的 JDBC 驱动程序 （将 *.jar 文件放置在适配器文件夹中）。

要查看可更改的所有可用 Hibernate 值，请检查 **org.hibernate.cfg.Environment** 类。

persistence.xml 文件示例：

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/
xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
    <!-- Don't change this value -->
    <persistence-unit name="GenericDBAdapter">
        <properties>
            <!--  Don't change this value  -->
            <property name="hibernate.archive.autodetection" value="class, hbm"/>
            <!--The driver class name"/-->
            <property name="hibernate.connection.driver_class"
value="com.mercury.jdbc.MercOracleDriver"/>
            <!--The connection url"/-->
            <property name="hibernate.connection.url" value="jdbc:mercury:oracle://
artist:1521;sid=cmdb2"/>
            <!--DB login credentials"/-->
            <property name="hibernate.connection.username" value="CMDB"/>
            <property name="hibernate.connection.password" value="CMDB"/>
            <!--connection pool properties"/-->
            <property name="hibernate.c3p0.min_size" value="5"/>
            <property name="hibernate.c3p0.max_size" value="20"/>
            <property name="hibernate.c3p0.timeout" value="300"/>
            <property name="hibernate.c3p0.max_statements" value="50"/>
            <property name="hibernate.c3p0.idle_test_period" value="3000"/>
            <!--The dialect to use-->
            <property name="hibernate.dialect"
value="org.hibernate.dialect.OracleDialect"/>
        </properties>
    </persistence-unit>
</persistence>
```

discriminator.properties 文件

本文件可将每个受支持的 CI 类型（也可以用作 orm.xml 中的区分器值）映射到区分器列的相应可能值列表（以逗号分隔）。

如果要创建的适配器使用的是区分器功能，则必须对 **discriminator.properties** 文件中的所有区分器值进行定义。

区分器映射示例：

discriminator.properties 文件包含以下代码：

```
node=10001, 10005,10010,10011,10012
nt=10002,10003
unix=10004,10006,10008
```

orm.xml 文件包含以下代码：

```
    <entity class="generic_db_adapter.node" >
        <table name="[table_name]"/>
        …
        <inheritance strategy="SINGLE_TABLE"/>
        <discriminator-value>node</discriminator-value>
        <discriminator-column name="[discriminator_column]"/>
        …
    </entity>
    <entity class="generic_db_adapter.nt" name="nt">
        <discriminator-value>nt</discriminator-value>
        <attributes/>
    </entity>
    <entity class="generic_db_adapter.unix" name="unix">
        <discriminator-value>unix</discriminator-value>
        <attributes/>
    </entity>
```

[discriminator_column] 属性的计算方式如下：

- 对应表的区分器列对于特定条目包含 10002。该条目将映射到 **nt** CIT。
- 对应表的区分器列对于特定条目包含 10006。该条目将映射到 **unix** CIT。
- 对应表的区分器列对于特定条目包含 10010。该条目将映射到 **node** CIT。

请注意， **node** CIT 也是 **nt** 和 **unix** 的父项。

replication_config.txt 文件

此文件包含以逗号分隔的 CI 和关系类型列表，这些 CI 和关系类型的属性条件受复制插件支持。有关详细信息，请参阅 “插件”（第 200 页）。

fixed_values.txt 文件

通过此文件，可以为某些 CIT 的特定属性配置固定值。因此，可以向这些属性中的每个属性分配一个不存储在数据库中的固定值。

此文件必须包含零个或多个格式如下的条目：

```
entity[<entityName>] attribute[<attributeName>] value[<value>]
```

例如：

```
entity[ip_address] attribute[ip_domain] value[DefaultDomain]
```

现成的转换器

您可以使用以下转换器将联合查询和复制作业转入和转出数据库数据。

本节包括以下主题：

enum-transformer 转换器

该转换器将使用作为输入参数给定的 XML 文件。

XML 文件可在硬编码 CMDB 值和数据库值 (enums) 之间进行映射。如果这些值中的其中一个不存在，则可以选择返回相同值、返回 NULL 或抛出异常。

每个实体属性使用一个 XML 映射文件。

注意：该转换器可用于 **transformations.txt** 文件中的 to_DB_class 和 from_DB_class 字段。

输入文件 XSD 示例：

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:element name="enum-transformer">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="value" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="DB-type" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="integer"/>
                        <xs:enumeration value="long"/>
                        <xs:enumeration value="float"/>
                        <xs:enumeration value="double"/>
                        <xs:enumeration value="boolean"/>
                        <xs:enumeration value="string"/>
                        <xs:enumeration value="date"/>
                        <xs:enumeration value="xml"/>
                        <xs:enumeration value="bytes"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="CMDB-type" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="integer"/>
                        <xs:enumeration value="long"/>
```

```
                        <xs:enumeration value="float"/>
                        <xs:enumeration value="double"/>
                        <xs:enumeration value="boolean"/>
                        <xs:enumeration value="string"/>
                        <xs:enumeration value="date"/>
                        <xs:enumeration value="xml"/>
                        <xs:enumeration value="bytes"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="non-existing-value-action" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="return-null"/>
                        <xs:enumeration value="return-original"/>
                        <xs:enumeration value="throw-exception"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
        </xs:complexType>
    </xs:element>
    <xs:element name="value">
        <xs:complexType>
            <xs:attribute name="CMDB-value" type="xs:string" use="required"/>
            <xs:attribute name="external-DB-value" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

用于将 “sys”值转换为 “System”值的示例：

在本示例中，CMDB 中的 sys 值将转换为联合数据库中的 System 值，而联合数据库中的 System 值将转换为 CMDB 中的 sys 值。

如果 XML 文件中不存在值（例如字符串 demo），转换器将返回接收的相同输入值。

```
<enum-transformer CMDB-type="string" DB-type="string" non-existing-value-action="return-original"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../
META-CONF/generic-enum-transformer.xsd">
    <value CMDB-value="sys" external-DB-value="System"/>
</enum-transformer>
```

SuffixTransformer 转换器

该转换器用于在 CMDB 或联合数据库中添加或删除源值后缀。

有两种实现方式：

- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdbAddSuffixTransformer**。将联合数据库值转换为 CMDB 值时，添加后缀 （指定为输入）；将 CMDB 值转换为联合数据库值时，删除后缀。
- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdbRemoveSuffixTransformer**。将联合数据库值转换为 CMDB 值时，删除后缀 （指定为输入）；将 CMDB 值转换为联合数据库值时，添加后缀。

PrefixTransformer 转换器

该转换器用于在 CMDB 或联合数据库中添加或删除源值前缀。

有两种实现方式：

- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdbAddPrefixTransformer**。将联合数据库值转换为 CMDB 值时，添加前缀 （指定为输入）；将 CMDB 值转换为联合数据库值时，删除前缀。
- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdbRemovePrefixTransformer**。将联合数据库值转换为 CMDB 值时，删除前缀 （指定为输入）；将 CMDB 值转换为联合数据库值时，添加前缀。

BytesToStringTransformer 转换器

该转换器用于将 CMDB 中的字节数组转换为这些数组在联合数据库源中的字符串表示形式。

转换器为：
com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.CmdbToAdapterBytesToStringTransformer。

插件

常规数据库适配器支持以下插件：

- 用于执行完整拓扑同步的可选插件。
- 用于同步拓扑变更的可选插件。如果未实施任何用于同步更改的插件，则可以执行差异同步，但是此差异同步实际上是完全同步。
- 用于同步布局的可选插件。
- 用于检索受支持的查询以执行同步的可选插件。如果没有定义该插件，则将返回所有 TQL 名称。
- 用于更改 TQL 定义和 TQL 结果的可选内部插件。
- 用于更改布局请求和 CI 结果的可选内部插件。
- 用于更改布局请求和关系结果的可选内部插件。

有关实施和部署插件的详细信息，请参阅 “实现插件”（第 149 页）。

配置示例

本节介绍了一些配置示例。

本节包括以下主题：

用例

用例。 TQL 是：

node > (composition) > card

其中：

node 是 CMDB 实体

card 是联合数据库源实体

composition 是这两个实体之间的关系

本示例对 ED 数据库运行。ED nodes 存储在 Device 表中，card 存储在 hwCards 表中。在以下示例中，始终以相同的方式对 card 进行映射。

单个节点调和

在本示例中，调和按照 name 属性运行。

简单定义

调和通过 node 完成，并使用特殊标记 **CMDB-class** 进行强调。

```
<?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>
```

高级定义

orm.xml 文件

请注意添加关系映射。有关详细信息，请参阅“orm.xml 文件”（第 179 页）中的定义部分。

orm.xml 文件示例：

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

```
<entity class="generic_db_adapter.node" >
    <table name="Device"/>
    <attributes>
        <id name="id1">
            <column name="Device_ID" insertable="false" updatable="false"/>
            <generated-value strategy="TABLE"/>
        </id>
        <basic name="name">
            <column name="Device_Name"/>
        </basic>
    </attributes>
</entity>
<entity class="generic_db_adapter.card" >
    <table name="hwCards"/>
    <attributes>
        <id name="id1">
            <column name="hwCards_Seq" insertable="false" updatable="false"/>
            <generated-value strategy="TABLE"/>
        </id>
        <basic name="card_class">
            <column name="hwCardClass" insertable="false" updatable="false"/>
        </basic>
        <basic name="card_vendor">
            <column name="hwCardVendor" insertable="false" updatable="false"/>
        </basic>
        <basic name="card_name">
            <column name="hwCardName" insertable="false" updatable="false"/>
        </basic>
    </attributes>
</entity>
<entity class="generic_db_adapter.node_composition_card" >
    <table name="hwCards"/>
    <attributes>
        <id name="id1">
            <column name="hwCards_Seq" insertable="false" updatable="false"/>
            <generated-value strategy="TABLE"/>
        </id>
        <many-to-one name="end1" target-entity="node">
            <join-column name="Device_ID" insertable="false" updatable="false"/>
        </many-to-one>
```

```
            <one-to-one name="end2" target-entity="card">
                <join-column name="hwCards_Seq" referenced-column-name="hwCards_Seq"
insertable="false" updatable="false"/>
            </one-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

reconciliation_types.txt 文件

有关详细信息，请参阅 “reconciliation_types.txt 文件”（第 190 页）。

```
node
```

reconciliation_rules.txt 文件

有关详细信息，请参阅 “reconciliation_rules.txt 文件 （用于向后兼容）”（第 190 页）。

```
multinode[node] expression[node.name]
```

transformation.txt 文件

此文件保留为空，因为本示例中没有需要转换的值。

两个节点调和

在本示例中，将根据含有不同变量的 node 和 ip_address 的 name 属性计算调和。

调和 TQL 为 **node > (containment) > ip_address**。

简单定义

通过对 node 和 ip_address 的 name 执行 OR 运算来计算调和：

```
<?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>
```

通过对 node 和 ip_address 的 name 执行 AND 运算来计算调和：

```
<?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>
```

通过 ip_address 的 name 计算调和：

```
<?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>
```

高级定义

orm.xml 文件

因为该文件中未定义调和表达式，所以所有调和表达式将使用同一版本。

reconciliation_types.txt 文件

有关详细信息，请参阅 “reconciliation_types.txt 文件”（第 190 页）。

```
node
```

reconciliation_rules.txt 文件

有关详细信息，请参阅 “reconciliation_rules.txt 文件 （用于向后兼容）”（第 190 页）。

```
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 文件

此文件保留为空，因为本示例中没有需要转换的值。

使用含有多个列的主键

如果主键由多个列组成，则会向 XML 定义中添加以下代码：

简单定义

存在多个主键标记，并且每一列有一个标记。

```
    <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>
```

高级定义

orm.xml 文件

将添加一个映射到主键列的新 id 实体。必须给使用此 id 实体的实体添加特殊标记。

如果将外键 （join-column 标记）用于此类主键，则必须在每个外键列和主键列之间进行映射。

有关详细信息，请参阅 “orm.xml 文件”（第 179 页）。

orm.xml 文件示例：

```
<   <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>
```

使用转换功能

在以下示例中，将常规 **enum** 转换器从值 1、2、3 分别转换为 name 列中的 a、b、c。

映射文件是 generic-enum-transformer-example.xml。

```
<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>
```

简单定义

```
    <CMDB-class CMDB-class-name="node" default-table-name="Device">
        <primary-key column-name="Device_ID"/>
        <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address"
CMDB-link-type="containment">
            <or>
                <attribute CMDB-attribute-name="name" column-name="Device_Name"
from-CMDB-converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.i
mpl.GenericEnumTransformer(generic-enum-transformer-example.xml)"
to-CMDB-converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(generic-enum-transformer-example.xml)"/>
                <connected-node-attribute CMDB-attribute-name="name"
column-name="Device_PreferredIPAddress"/>
            </or>
        </reconciliation-by-two-nodes>
    </CMDB-class>
.
.
.
```

高级定义

仅对 **transformation.txt** 文件进行更改。

transformation.txt 文件

确保 orm.xml 文件中的属性名称和实体名称相同。

```
entity[node] attribute[name]
to_DB_class[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.Generic
EnumTransformer(generic-enum-transformer-example.xml)]
from_DB_class[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.Gene
ricEnumTransformer(generic-enum-transformer-example.xml)]
```

适配器日志文件

要了解计算流程和适配器生命周期以及查看调试信息，可以查阅以下日志文件。

本节包括以下主题：

日志级别

可以为每个日志配置日志级别。

在文本编辑器中，打开 **C:\hp\UCMDB\UCMDBServer\conf\log\fcmdb.gdba.properties** 文件。

默认的日志级别是 **ERROR**：

```
#loglevel can be any of DEBUG INFO WARN ERROR FATAL
loglevel=ERROR
```

➤ 要提高所有日志文件的日志级别，请将 **loglevel=ERROR** 更改为 **loglevel=DEBUG** 或 **loglevel=INFO**。

➤ 要更改特定文件的日志级别，请对特定 **log4j** 类别行进行相应的更改。例如，要将 fcmdb.gdba.dal.sql.log 的日志级别更改为 **INFO**，请将

```
log4j.category.fcmdb.gdba.dal.SQL=${loglevel},fcmdb.gdba.dal.SQL.appender
```

更改为：

```
log4j.category.fcmdb.gdba.dal.SQL=INFO,fcmdb.gdba.dal.SQL.appender
```

日志位置

日志文件位于 **C:\hp\UCMDB\UCMDBServer\runtime\log** 目录中。

- **Fcmdb.gdba.log**

  适配器生命周期日志。提供有关适配器的开始或停止时间，以及受该适配器支持的 CIT 的详细信息。

  查看初始化错误 （适配器加载 / 卸载）。

- **fcmdb.log**

  查看异常。

- **cmdb.log**

  查看异常。

- **Fcmdb.gdba.mapping.engine.log**

  映射引擎日志。提供有关映射引擎所使用的调和 TQL，以及在连接阶段进行比较的调和拓扑的详细信息。

  如果 TQL 查询没有提供任何结果 （即使您知道数据库中存在相关 CI）或者产生非预期结果 （检查调和），请查看该日志。

- **Fcmdb.gdba.TQL.log**

  TQL 日志。提供关于 TQL 查询及其结果的详细信息。

  如果 TQL 查询不返回结果并且映射引擎日志显示联合数据源中不存在任何结果，请查看该日志。

- **Fcmdb.gdba.dal.log**

  DAL 生命周期日志。提供关于 CIT 生成和数据库连接的详细信息。

  如果您无法连接到数据库，或者存在不受查询支持的 CIT 或属性，请查看该日志。

- **Fcmdb.gdba.dal.command.log**

  DAL 操作记录。提供有关调用的内部 DAL 操作的详细信息。（该日志与 cmdb.dal.command.log 相似）。

- **Fcmdb.gdba.dal.SQL.log**

  DAL SQL 查询日志。提供有关调用的 JPAQL（面向对象的 SQL 查询）及其结果的详细信息。

  如果您无法连接到数据库，或者存在不受查询支持的 CIT 或属性，请查看该日志。

- **Fcmdb.gdba.hibrnate.log**

  Hibernate 日志。提供有关运行的 SQL 查询、每个 JPAQL 到 SQL 的解析、查询结果、Hibernate 缓存数据等的详细信息。有关 Hibernate 的详细信息，请参阅 “Hibernate 作为 JPA 提供程序”（第 129 页）。

外部参考

有关 JavaBeans 3.0 规范的详细信息，请参阅 http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html。

疑难解答和局限性

本节描述了有关常规数据库适配器的疑难解答和局限性。

常规限制

- 不支持 SQL Server NTLM 身份验证。
- 更新适配器包时，请使用 Notepad++、UltraEdit 或某些其他第三方文本编辑器代替 Microsoft Corporation 的 Notepad（任何版本）对模板文件进行编辑。这样可以避免使用特殊符号，这些特殊符号会导致无法部署已准备就绪的包。

JPA 限制

- 所有表都必须具有主键列。
- CMDB 类属性名称必须遵守 JavaBeans 命名约定（例如，名称必须以小写字母开头）。
- 类模型中通过某种关系连接的两个 CI 必须在数据库中直接关联，例如，如果将 node 连接到 ticket，则必须存在用于连接这两个 CI 的外键或连接表。
- 映射到同一 CIT 的多个表必须共享同一个主键表。

功能限制

- 不能在 CMDB 和联合 CIT 之间创建手动关系。要定义虚拟关系，必须定义一种特殊的关系逻辑（该逻辑可基于联合类的属性）。
- 在影响规则中，联合 CIT 不能是触发器 CIT，但是这两种 CIT 可以包含在同一个影响分析 TQL 查询中。
- 联合 CIT 可以是扩展 TQL 的一部分，但不能用作执行扩展的节点（不能添加、更新或删除联合 CIT）。
- 不支持在条件中使用类限定符。
- 不支持子图。
- 不支持复合关系。
- 外部 CI CMDB id 由其主键而非其键属性组成。
- 类型为**字节**的列不能用作 Microsoft SQL Server 中的主键列。
- 如果没有在 **orm.xml** 文件中映射为联合节点定义的属性条件的名称，则 TQL 查询计算将失败。
- 常规 DB 适配器不支持对 SQL Server 进行 Windows 身份验证。

6

开发 Java 适配器

本章包括：

概念

任务

参考

概念

联合框架概述

注意：

- 术语**关系**等价于术语**链接**。
- 术语 **CI** 等价于术语**对象**。
- 图形是节点和链接的集合。
- 有关定义和术语的词汇表，请参阅 《HP Universal CMDB 管理指南》中的“词汇表”。

“联合框架”功能使用 API 从联合源中检索信息。“联合框架”提供三种主要功能

- 动态**联合**。所有查询在原始数据库中运行，并在 CMDB 中动态生成结果。
- **填充。**将数据 （拓扑数据和 CI 属性）从外部数据源填充到 CMDB 中。
- **数据推送。**将数据 （拓扑数据和 CI 属性）从本地 CMDB 推送到远程数据源。

所有操作类型都要求对每个数据库使用适配器，该适配器可以提供数据库的特定功能，并且能够检索和 / 或更新所需的数据。对数据库发出的每个请求将通过其适配器完成。

本节还包括以下主题：

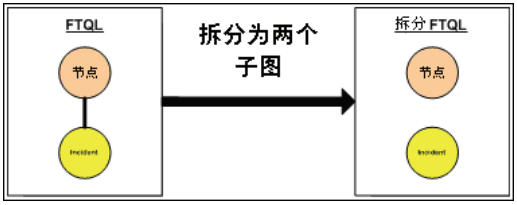
动态联合

通过联合的 TQL 查询，可对任何外部数据库进行数据检索，而不需要复制其数据。

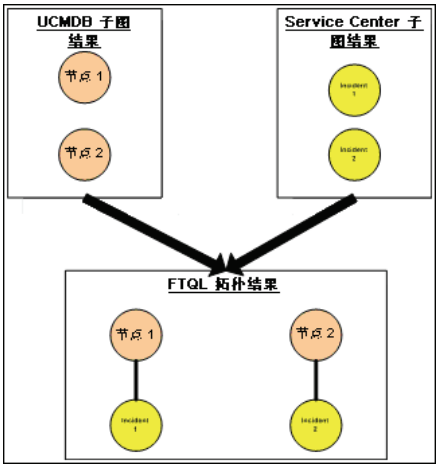
联合的 TQL 查询使用表示外部数据库的适配器，在来自不同外部数据库的 CI 和 UCMDB CI 之间创建相应的外部关系。

动态联合流示例：

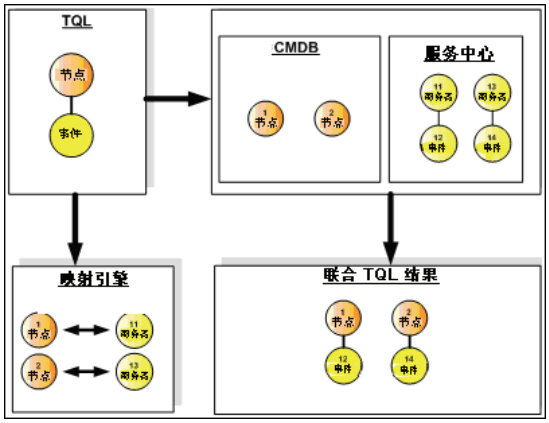
1 联合框架会将联合的 TQL 查询分割成多个子图，子图中的所有节点都将引用相同的数据库。每个子图通过虚拟关系连接到其他子图（但自身并不包含虚拟关系）。



2 当联合的 TQL 查询分割成多个子图后，联合框架将计算每个子图的拓扑，并通过在相应节点之间创建虚拟关系来连接两个相应的子图。



3 在计算联合的 TQL 拓扑之后，联合框架即可检索拓扑结果的布局。



数据推送

您可以使用数据推送流程将当前本地 CMDB 中的数据与远程服务或目标数据库同步。

在数据推送中，数据库可分为两个类别：源 （本地 CMDB）和目标。将在源数据库中检索数据，然后将数据更新到目标数据库。数据推送过程基于查询名称执行，这意味着将在源 （本地 CMDB）和目标数据库之间进行数据同步，并且按 TQL 查询名称在本地 CMDB 中检索数据。

数据推送过程包括以下步骤：

1 使用签名在源数据库中检索拓扑结果。

2 将新结果与之前的结果进行比较。

3 检索 CI 和关系的完整布局 （即所有 CI 属性）；此操作仅针对已发生更改的结果。

4 使用收到的 CI 和关系的完整布局更新目标数据库。如果在源数据库中删除了任何 CI 或关系，并且查询是独占式查询，则复制过程也会删除目标数据库中的对应 CI 或关系。

CMDB 有 2 个隐藏的数据源（**hiddenRMIDataSource** 和 **hiddenChangesDataSource**），它们始终是数据推送流程中的“源”数据源。要对数据推送流程实施新适配器，只需实施“目标”适配器即可。

填充

您可以使用填充流程将外部源的数据填充到 CMDB 中。

该流程会始终使用一个“源”数据源来检索数据，并将检索到的数据推送到 Probe 中，其推送过程与搜寻作业流的过程类似。

要为填充流程实施新适配器，只需实施源适配器即可，因为数据流 Probe 将充当目标。

填充流程中的适配器将在 Probe 上执行。所以，应在 Probe 而不是 CMDB 上完成调试和登录。

填充流程基于查询名称，这意味着将在源数据库和数据流 Probe 之间同步数据，并按查询名称在源数据库中检索数据。例如，在 UCMDB 中，查询名称是 TQL 查询的名称。但是，在另一个数据库中，查询名称则可能是返回数据的代码名称。适配器用于正确地处理查询名称。

可将每个作业定义为独占作业。这意味着作业结果中的 CI 和关系在本地 CMDB 中是唯一的，任何其他查询均不能将它们带入到目标中。源数据库的适配器支持特定查询，并且可以在源数据库中检索数据。目标数据库的适配器支持在源数据库上更新检索到的数据。

SourceDataAdapter 流

- 使用签名在源数据库中检索拓扑结果。
- 将新结果与之前的结果进行比较。

- 检索 CI 和关系的完整布局 （即所有 CI 属性）；此操作仅针对已发生更改的结果。
- 使用所收到的 CI 及关系的完整布局更新目标数据库。如果在源数据库中删除了任何 CI 或关系，并且查询是独占式查询，则复制过程也会删除目标数据库中的对应 CI 或关系。

SourceChangesDataAdapter 流

- 检索自上个指定日期以来发生的拓扑结果。
- 检索 CI 和关系的完整布局 （即所有 CI 属性）；此操作仅针对已发生更改的结果。
- 使用所收到的 CI 及关系的完整布局更新目标数据库。如果在源数据库中删除了任何 CI 或关系，并且查询是独占式查询，则复制过程也会删除目标数据库中的对应 CI 或关系。

PopulateDataAdapter 流

- 使用请求的布局结果检索完整拓扑。
- 使用拓扑块机制检索块中的数据。
- Probe 筛选出之前的运行中引入的所有数据。
- 使用收到的 CI 和关系的布局更新目标数据库。如果在源数据库中删除了任何 CI 或关系，并且查询是独占式查询，则复制过程也会删除目标数据库中的对应 CI 或关系。

PopulateChangesDataAdapter 流

- 使用自上次运行以来已发生更改的请求布局结果检索拓扑。
- 使用拓扑块机制检索块中的数据。
- Probe 筛选出之前的运行中 （包括此流）引入的所有数据。

- 使用收到的 CI 和关系的布局更新目标数据库。如果在源数据库中删除了任何 CI 或关系，并且查询是独占式查询，则复制过程也会删除目标数据库中的对应 CI 或关系。

适配器与联合框架的映射交互

适配器是 UCMDB 中代表外部数据 （未保存在 UCMDB 中的数据）的实体。在联合流中，与外部数据源的所有交互将通过适配器执行。用于复制和用于联合 TQL 查询的联合框架交互流和适配器接口各不相同。

本节还包括以下主题：

- “适配器生命周期”（第 224 页）
- “适配器 assist 方法”（第 225 页）

适配器生命周期

将为每个外部数据库创建一个适配器实例。适配器的生命周期从应用于它的第一个操作 （例如，**计算 TQL** 或**检索 / 更新数据**）开始计算。调用 **start** 方法后，适配器即可接收环境信息，例如数据库配置、记录器等。从配置中删除数据库，并且调用 **shutdown** 方法后，适配器生命周期将结束。这表示适配器具有状态，如果需要，可以包含外部数据库的连接。

适配器 assist 方法

适配器有多个 assist 方法，可用于添加外部数据库配置。这些方法不包含在适配器生命周期内，并会在每次被调用时创建一个新适配器。

- 第一个方法测试指定配置的外部数据库连接。根据适配器的类型，可以在 UCMDB 服务器或数据流 Probe 上执行 testConnection。
- 第二个方法仅适用于源适配器，将返回可用于复制的支持查询。（只能在 Probe 上执行此方法。）
- 第三种方法仅适用于联合和填充流程，将返回外部数据库支持的外部类。（只能在 UCMDB 服务器上执行此方法。）

在创建或查看集成配置时将使用上述所有方法。

联合的 TQL 查询的联合框架流

本节包括以下主题：

定义和术语

调和数据。一种规则，用于匹配从 CMDB 和外部数据库接收的指定类型 CI。有三种类型的调和规则：

- **ID 调和。**只有当外部数据库包含调和对象的 CMDB ID 时，才可使用此规则。
- **属性调和。**只有能够按照调和 CI 类型的属性完成匹配时，才可使用此规则。
- **拓扑调和。**在需要其他 CIT（不仅仅是调和 CIT）属性才能对调和 CI 执行匹配时，可使用此规则。例如，您可以通过属于 ip_address CIT 的**名称**属性执行节点类型的调和。

调和对象。对象由适配器根据所接收的调和数据创建。此对象将引用外部 CI，并由映射引擎用于连接外部 CI 和 CMDB CI。

调和 CI 类型。表示调和对象的 CI 类型。这些 CI 必须存储在 CMDB 和外部数据库中。

映射引擎。一个组件，用于标识不同数据库（之间存在虚拟关系）中的 CI 之间的关系。通过协调 CMDB 调和对象和外部 CI 调和对象，可完成标识操作。

映射引擎

联合框架使用映射引擎计算联合 TQL 查询。映射引擎连接从不同数据库（这些数据库通过虚拟关系连接）接收的 CI。映射引擎还将提供虚拟关系的调和数据。虚拟关系的一端必须是 CMDB，此端为**调和**类型。要计算两个子图，虚拟关系可以从任何一个端节点开始。

联合适配器

联合适配器从外部数据库引入两种数据：外部 CI 数据和属于外部 CI 的调和对象。

- **外部 CI 数据。** CMDB 中不存在外部数据。它是外部数据库的目标数据。
- **调和对象数据。**一种辅助数据，由联合框架用于连接 CMDB CI 和外部数据。每个调和对象引用一个外部 CI。调和对象的类型是要从其检索数据的某个虚拟关系端的类型 （或子类型）。调和对象应与调和数据接收的适配器相匹配。调和对象可以是以下三种类型之一：IdReconciliationObject、PropertyReconciliationObject 或 TopologyReconciliationObject。

在基于 DataAdapterd 的接口 （DataAdapter、 PopulateDataAdapter 和 PopulateChangesDataAdapter）中，请求调和时，它包含在查询定义中。

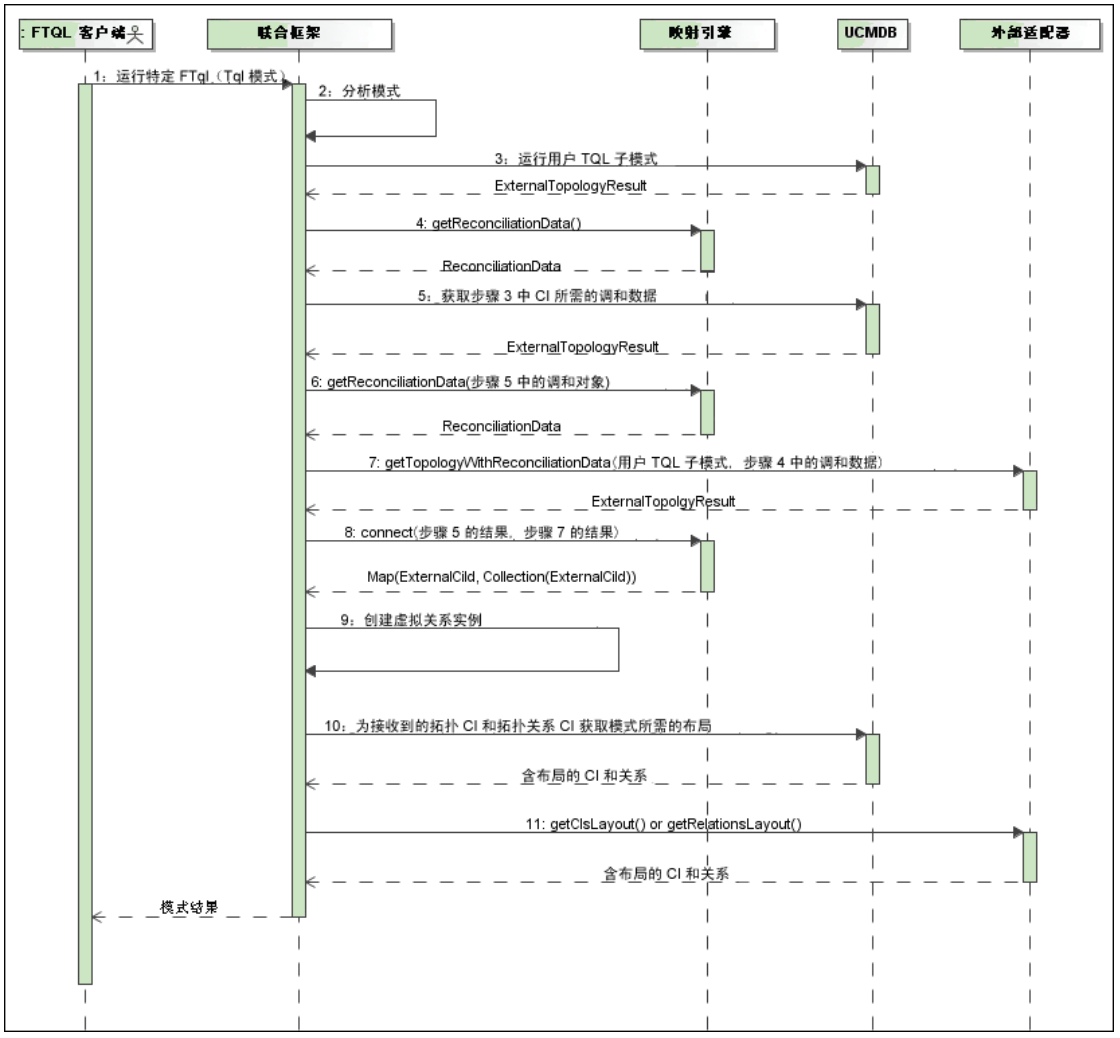
流图

下图演示了联合框架、 UCMDB、适配器和映射引擎之间的交互。这些示例图中的联合 TQL 查询只有一种虚拟关系，因此联合 TQL 查询中仅涉及 UCMDB 和一个外部数据库。

在第一个图中，从 UCMDB 中开始计算；在第二个图表中，从外部适配器中开始计算。图中的每个步骤均包含了对适配器或映射引擎接口的相应方法调用的引用。

从 HP Universal CMDB 端开始计算

以下序列图演示了联合框架、 UCMDB、适配器和映射引擎之间的交互。示例图中的联合 TQL 查询只有一种虚拟关系，因此联合 TQL 查询中仅涉及 UCMDB 和一个外部数据库。

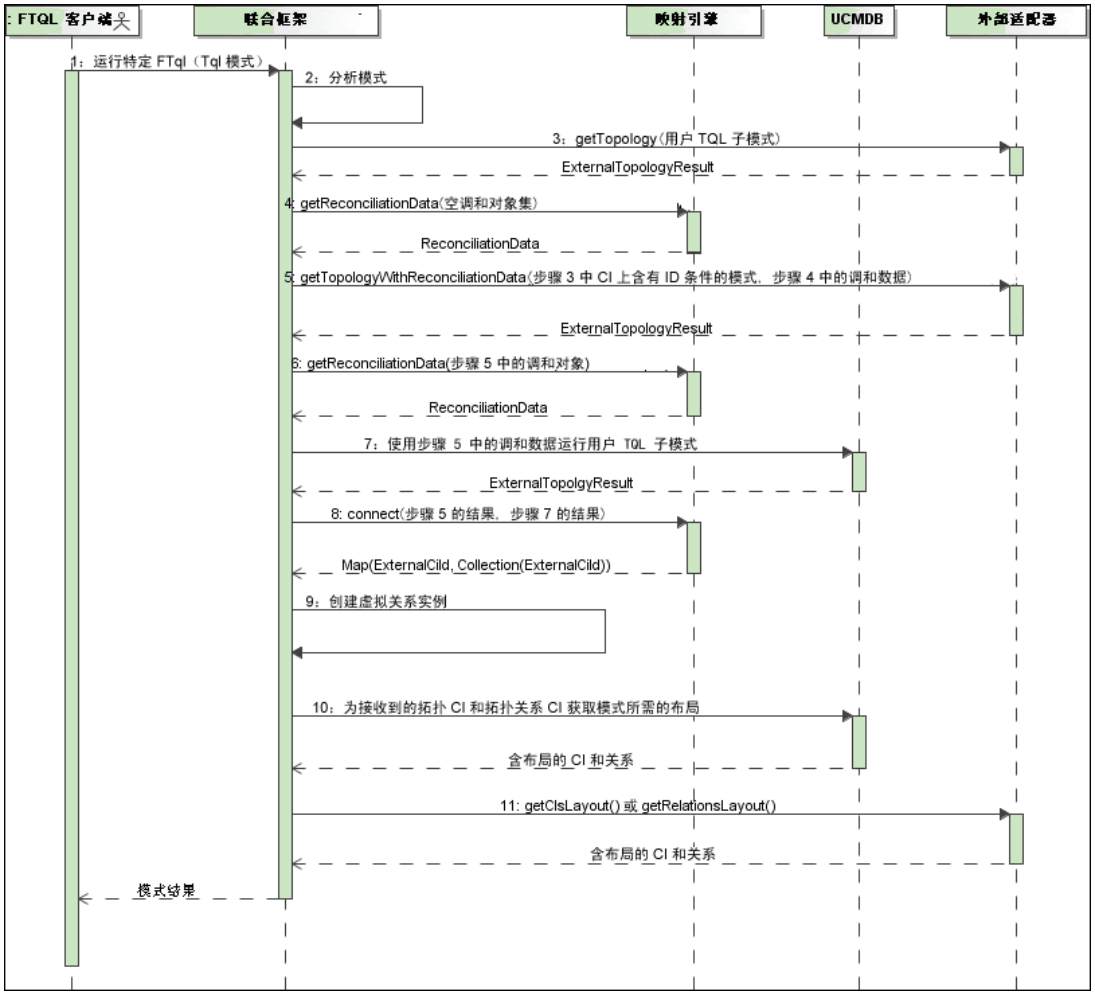


对图中编号的解释如下：

编号	解释
1	联合框架接收联合 TQL 计算的调用。
2	联合框架分析适配器，查找虚拟关系，并将原始 TQL 划分成两个子适配器：一个用于 UCMDB，另一个用于外部数据库。
3	联合框架从 UCMDB 请求子 TQL 的拓扑。
4	在接收拓扑结果之后，联合框架将为当前虚拟关系调用相应的映射引擎并请求调和数据。在此阶段， reconciliationObject 参数为空，也就是说，在此调用中未向调和数据添加任何条件。返回的调和数据定义了将 UCMDB 与外部数据库中的调和 CI 进行匹配所需的数据。调和数据可以是以下类型之一： ➤ **IdReconciliationData**. 按照 ID 调和的 CI。 ➤ **PropertyReconciliationData**. 按照某个 CI 的属性调和的 CI。 ➤ **TopologyReconciliationData**. 按照拓扑 （例如，如果要调和节点 CI，则还需要 **IP** 的 IP 地址）调和的 CI。
5	联合框架从 UCMDB 请求在步骤 3 中接收的虚拟关系末端 CI 的调和数据。
6	联合框架调用映射引擎以检索调和数据。在此阶段 （与步骤 3 对照），映射引擎接收步骤 5 中的调和对象作为参数。映射引擎将接收的调和对象转换成调和数据条件。
7	联合框架从外部数据库请求子 TQL 的拓扑。外部适配器接收步骤 6 中的调和数据作为参数。

编号	解释
8	联合框架调用映射引擎，以连接所收到的结果。 firstResult 参数是在步骤 5 中从 UCMDB 接收的外部拓扑结果； secondResult 参数是在步骤 7 中从外部适配器接收的外部拓扑结果。映射引擎将返回一个图，该图中第一个数据库（在此例中为 UCMDB）的外部 CI ID 将映射到第二个（外部）数据库的外部 CI ID。
9	联合框架为每个映射创建一个虚拟关系。
10	计算联合 TQL 查询结果后 （仅在拓扑阶段），联合框架将从相应的数据库中检索结果 CI 和关系的原始 TQL 布局。

从外部适配器端开始计算



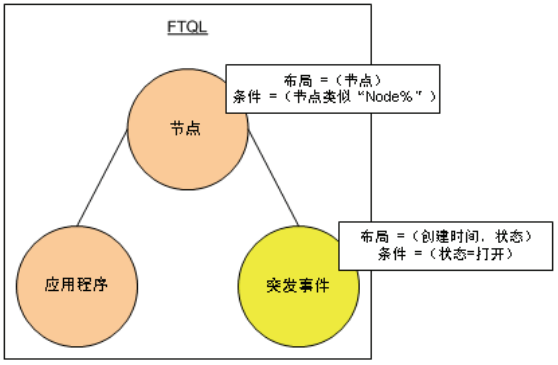
对图中编号的解释如下：

编号	解释
1	联合框架接收联合 TQL 计算的调用。
2	联合框架分析适配器，查找虚拟关系，并将原始 TQL 划分成两个子适配器：一个用于 UCMDB，另一个用于外部数据库。
3	联合框架从外部适配器请求子 TQL 的拓扑。由于请求中不包含调和数据，预计返回的 ExternalTopologyResult 不包含任何调和对象。
4	在接收拓扑结果之后，联合框架将为当前虚拟关系调用相应的映射引擎并请求调和数据。在此阶段，reconciliationObjects 参数为空，也就是说，在此调用中未向调和数据添加任何条件。返回的调和数据定义了将 UCMDB 与外部数据库中的调和 CI 进行匹配所需的数据。调和数据可以是以下类型之一： ➤ **IdReconciliationData**. 按照 ID 调和的 CI。 ➤ **PropertyReconciliationData**. 按照某个 CI 的属性调和的 CI。 ➤ **TopologyReconciliationData**. 按照拓扑（例如，如果要调和节点 CI，则还需要 **IP** 的 IP 地址）调和的 CI。
5	联合框架从外部数据库请求步骤 3 中接收的 CI 的调和对象。联合框架在外部适配器中调用 **getTopologyWithReconciliationData()** 方法，其中，请求的拓扑是一个单节点拓扑，该拓扑将步骤 3 中接收的 CI 作为步骤 4 中的 ID 条件和调和数据。

编号	解释
6	联合框架调用映射引擎检索调和数据。在此阶段 （与步骤 3 对照），映射引擎接收步骤 5 中的调和对象作为参数。映射引擎将接收的调和对象转换成调和数据条件。
7	联合框架使用 步骤 6 中的调和数据从 UCMDB 中请求子 TQL 的拓扑。
8	联合框架调用映射引擎，以连接所收到的结果。 firstResult 参数是在步骤 5 中从外部适配器接收的外部拓扑结果； secondResult 参数是在步骤 7 中从 UCMDB 接收的外部拓扑结果。映射引擎将返回一个图，在该图中第一个数据库（在此情况下为外部数据库）的外部 CI ID 将映射到第二个数据库 (UCMDB) 的外部 CI ID。
9	联合框架为每个映射创建一个虚拟关系。
10	计算联合 TQL 查询结果后 （仅在拓扑阶段），联合框架将从相应的数据库中检索结果 CI 和关系的原始 TQL 布局。

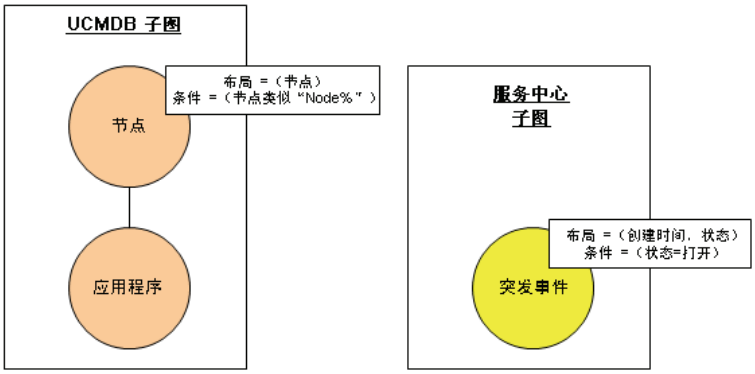
联合 TQL 查询的联合框架流示例

此示例解释了如何在特定节点上查看所有打开的事件。ServiceCenter 数据库是外部数据库。节点实例存储在 UCMDB 中，事件实例存储在 ServiceCenter 中。假定需要将事件实例连接到相应的节点，则需要主机和 IP 的 node 和 ip_address 属性。它们是在 UCMDB 中用于标识 ServiceCenter 的节点的调和属性。

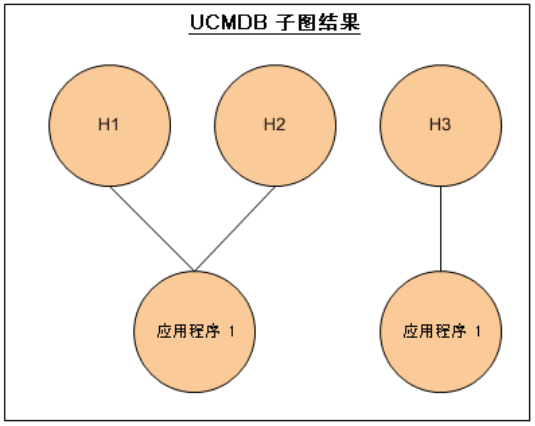


注意：对于属性联合，将调用适配器的 **getTopology** 方法。在用户 TQL 中调整调和数据 （在此例中是 CI 元素）。

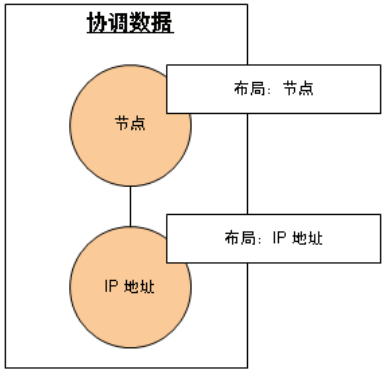
1 在分析适配器之后，联合框架将识别出**节点**和**事件**之间的虚拟关系，并将联合 TQL 查询拆分成两个子图：



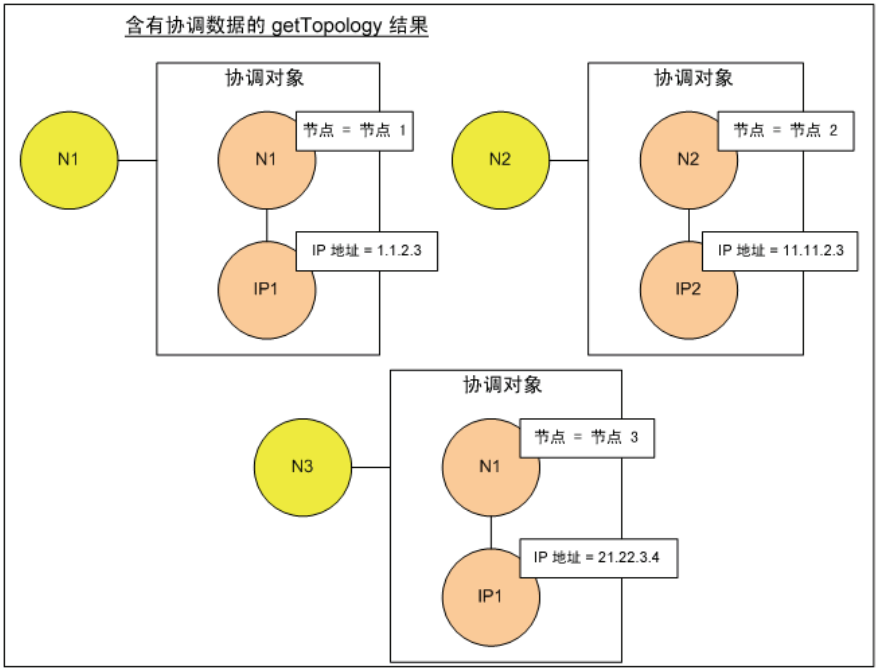
2 联合框架将运行 UCMDB 子图请求拓扑，并且收到以下结果：



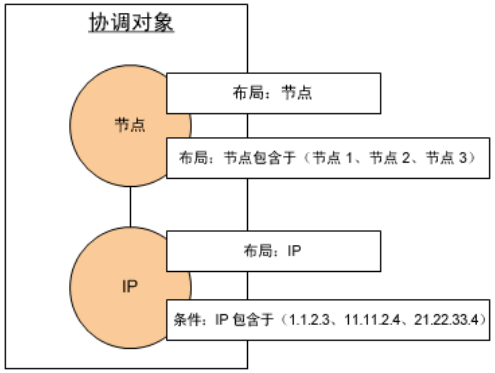
3 联合框架从相应的映射引擎请求第一个数据库 (UCMDB) 的调和数据，其中含有用于连接从两个数据库收到的数据的信息。在此情况下，调和数据为：



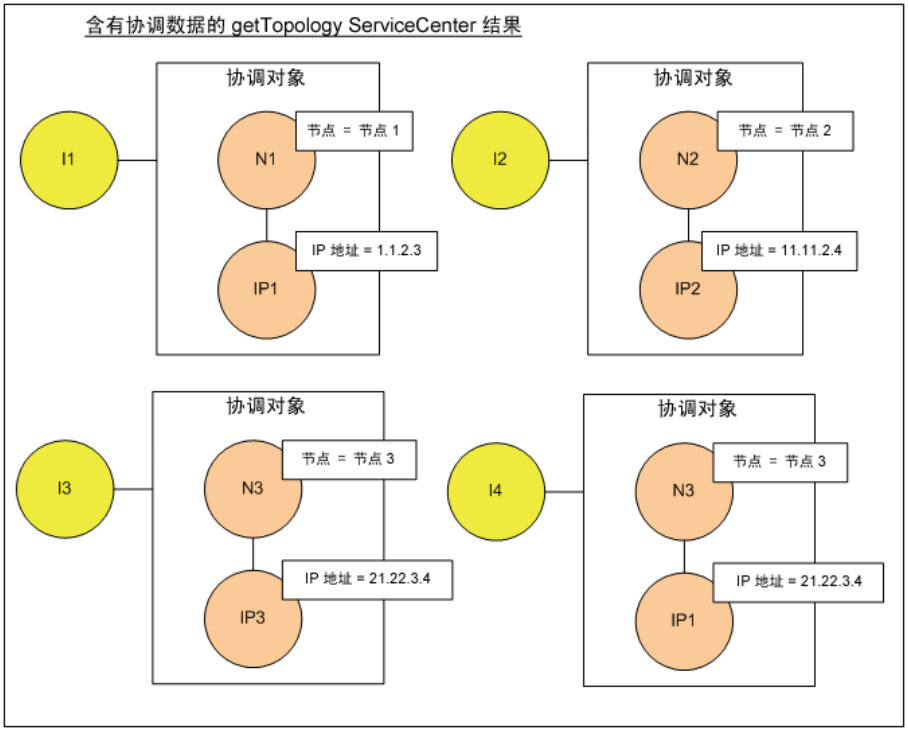
4 联合框架使用先前的结果（H1、H2 和 H3 中的**节点**）中的节点和 ID 条件创建单节点拓扑查询，并用所需的调和数据在 UCMDB 上运行此查询。结果包括与 ID 条件相关的节点 CI，以及每个 CI 相应的调和对象：



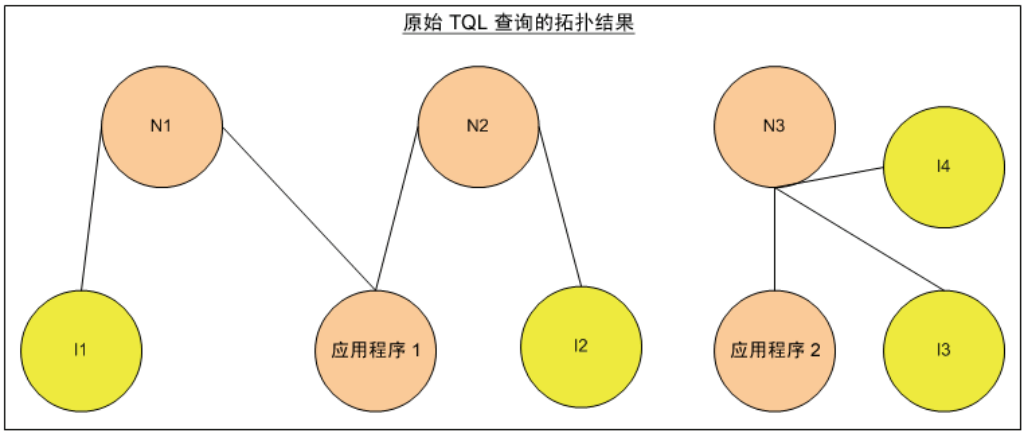
5 ServiceCenter 的调和数据应包含从 UCMDB 接收的调和对象派生出的**节点**和 IP 的条件：



6 联合框架使用调和数据运行 ServiceCenter 子图，请求拓扑和相应的调和对象，并收到以下结果：



7 在映射引擎中进行连接并且创建虚拟关系之后，结果如下：



8 联合框架从 UCMDB 和 ServiceCenter 请求已接收实例的原始 TQL 布局。

用于填充的的联合框架流

本节包括以下主题：

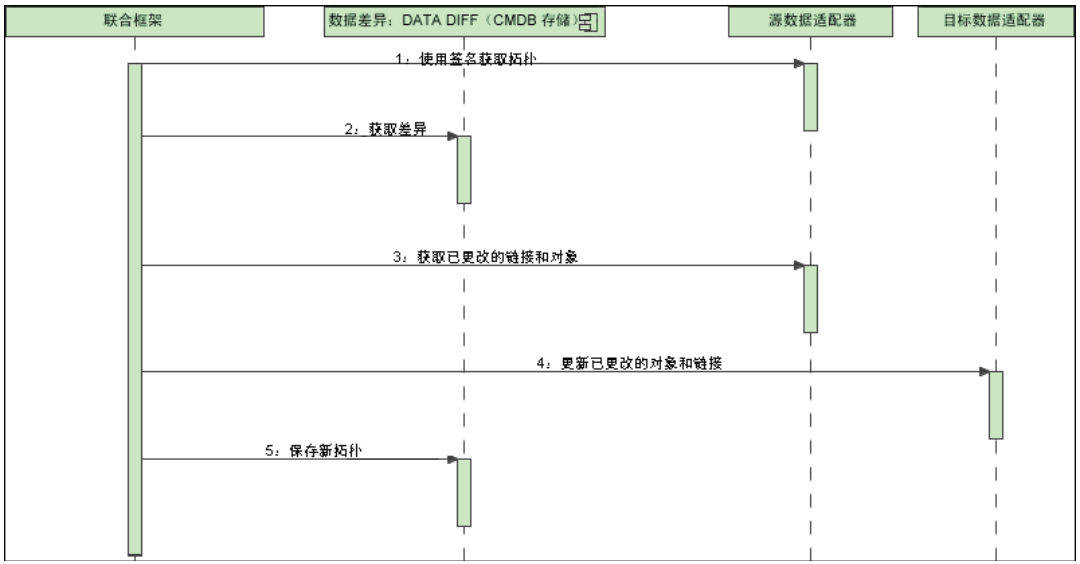
- “定义和术语”（第 241 页）
- “流程图”（第 242 页）

定义和术语

签名。表示 CI 中各个属性的状态。如果 CI 中的属性值发生更改，则必须更改 CI 签名。通过 CI 签名，可以在不检索和比较所有 CI 属性的情况下检测是否发生了 CI 更改。 CI 和 CI 签名均由相应的适配器提供。适配器负责在 CI 属性更改时更改 CI 签名。

流程图

以下序列图演示了联合框架与填充流程中源适配器和目标适配器之间的交互。



1. 联合框架从源适配器接收查询结果的拓扑。适配器通过查询的名称识别查询，并且在外部数据存储库中运行查询。拓扑结果在结果信息中包含每个 CI 和关系的 ID 和签名。ID 是用于在外部数据存储库中唯一地定义 CI 的逻辑 ID。如果修改了 CI 或关系，则应修改签名。
2. 联合框架使用签名来比较新接收的拓扑查询结果与已保存的查询结果，以确定哪些 CI 发生了更改。
3. 当联合框架找到发生更改的 CI 和关系之后，将使用已更改的 CI 或关系的 ID 作为参数来调用源适配器，以检索这些 CI 和关系的完整布局。
4. 联合框架将更新发送到目标适配器。目标适配器使用接收到的数据更新外部数据源。
5. 更新之后，联合框架保存最后一个查询结果。

适配器接口

本节包括以下主题：

定义和术语

外部关系。受同一适配器支持的两个外部 CI 类型之间的关系。

联合 TQL 查询的适配器接口

为每个适配器使用合适的适配器接口，如下所示：

当适配器不支持任何外部关系时，可使用**单节点拓扑接口**，这意味着适配器将不接受多个外部 CI 的请求。创建 OneNode 接口的目的是简化工作流程；对于需要使用更广泛查询的情况，可使用 DataAdapter 接口。

对于 UCMDB 9.00，建议不要使用：模式拓扑接口

DataAdapter 接口用于定义可支持复杂查询的适配器。这些适配器中的调和请求是单个 QueryDefinition 参数的组成部分。同时，这些适配器还可用于填充。

OneNode 接口

以下接口含有不同类型的调和数据：

➤ **OneNodeTopologyIdReconciliationDataAdapter。**如果适配器支持**单节点 TQL**，并且根据 ID 计算数据库之间的调和，则使用此接口。

➤ **OneNodeTopologyPropertyReconciliationDataAdapter。**如果适配器支持**单节点 TQL**，并且根据一个 CI 的属性计算数据库之间的调和，则使用此接口。

➤ **OneNodeTopologyDataAdapter**。如果适配器支持**单节点 TQL**，并且根据拓扑计算数据库之间的调和，则使用此接口。

数据适配器接口

➤ **DataAdapter**. 使用此适配器可支持复杂的联合 TQL 查询。可实现很高的多样性。

➤ **PopulateDataAdapter**. 使用此适配器可支持复杂的联合 TQL 查询和填充流程。在填充流程中，此适配器将检索整个数据集，并允许 Probe 筛选自上次执行作业以来的差异。

➤ **PopulateChangesDataAdapter**. 使用此适配器可支持复杂的联合 TQL 查询和填充流程。在填充流程中，此适配器仅支持检索自上次执行作业以来发生的更改。

模式拓扑接口 （建议不在 UCMDB 9.00 之后的版本中使用）

以下接口含有不同类型的调和数据：

➤ **PatternTopologyIdReconciliationDataAdapter**. 如果适配器支持**复杂 TQL**，并且根据 ID 计算数据库之间的调和，则使用此接口。

➤ **PatternTopologyPropertyReconciliationDataAdapter**. 如果适配器支持**复杂 TQL**，并且根据单节点属性计算数据库之间的调和，则使用此接口。

➤ **PatternTopologyDataAdapter**. 如果适配器支持**复杂 TQL**，并且根据拓扑计算数据库之间的调和，则使用此接口。

其他接口

➤ **SortResultDataAdapter**. 如果能够在外部数据库中对生成的 CI 进行排序，则使用此接口。

➤ **FunctionalLayoutDataAdapter**. 如果能够在外部数据库中计算功能布局，则使用此接口。

用于同步的适配器接口

- **SourceDataAdapter**. 用于填充流程中的源适配器。
- **TargetDataAdapter**. 用于数据推送流程中的目标适配器。

任务

为新外部数据源添加适配器

本任务说明如何定义适配器以便支持新的外部数据源。

本任务包括以下步骤：

1 先决条件

UCMDB 数据模型中模型支持的 CI 和关系的适配器类：作为适配器开发人员，应当：

- 熟悉 UCMDB CI 类型的层次结构，了解外部 CIT 与 UCMDB CIT 的关联方式
- 可以在 UCMDB 类模型中为外部 CIT 建模
- 为新的 CI 类型及其关系添加定义
- 在 UCMDB 类模型中为适配器内部类之间的有效关系定义有效关系。（可以将 CIT 放在 UCMDB 类模型树的任意级别中）。

无论联合类型如何 （动态或复制），建模方式都应相同。有关将新 CIT 定义添加到 UCMDB 类模型中的详细信息，请参阅《HP Universal CMDB 建模指南》中的 “使用 CI 选择器”。

要使适配器支持 CIT 上的联合属性，请将此 CIT 添加到具有该 CIT 的支持属性和调和规则的支持类中。

2 为虚拟关系定义有效关系

注意：本节仅与联合相关。

要检索已连接到本地 CMDB CIT 的联合 CIT，必须在 CMDB 中的两个 CIT 之间建立有效的链接定义。

a 创建一个包含这些链接 （如果链接不存在）的有效链接 XML 文件。

b 将链接 XML 文件添加到 **\validlinks** 文件夹的适配器包中。有关详细信息，请参阅 《HP Universal CMDB 管理指南》中的 “包管理器”。

有效关系定义示例：

在以下示例中，类型 node 类型实例与 myclass1 类型实例之间的 containment 类型关系是有效关系定义。

```
<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 定义适配器配置

a 导航到 “适配器管理”。

b 单击 “创建新资源”按钮。

c 在 “新建适配器”对话框中，选择 “集成”和 “Java 适配器”。

d 右键单击已创建的适配器，从快捷菜单中选择 “编辑适配器源”。

e 编辑以下 XML 标记：

```
<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.BaseDestinationData">
            <!-- 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/> <! 案 ee the section about supported classes-->
            <topology>
                <pattern-topology /> <! 桕 r <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>
```

有关 XML 标记的详细信息，请参阅 “XML 配置标记和属性”（第 258 页）。

4 定义支持的类

通过执行 “getSupportedClasses()” 方法或通过使用模式 XML 文件，定义受支持的类或适配器代码。

```
<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 类型的名称
is-derived	指定此定义是否包括所有继承子级
is-reconciliation-supported	指定此类是否用于调和
is-id-reconciliation-supported	指定此类是否用于 ID 调和
federation-not-supported	指定是否不允许在联合中使用此 CIT（阻止某些 CIT，例如专为联合而定义的 CIT）
<supported-conditions>	指定每个属性的支持条件

5 实施适配器

根据定义的功能选择正确的适配器实施类。适配器实施类根据所定义的功能实现相应的接口。

6 定义调和规则或实施映射引擎

如果适配器支持联合 TQL 查询，则可通过下列三种方式来定义映射引擎：

- 使用默认 CMDB 9.0x 默认映射引擎，该引擎使用 CMDB 的内部调和规则进行映射。要使用该引擎，请将 **<default-mapping-engine/>** XML 标记保留为空。

  有关详细信息，请参阅 “reconciliation_types.txt 文件”（第 190 页）。

- 使用 CMDB 8.0x 映射引擎。要使用该引擎，请使用以下 XML 标记：
  <default-mapping-engine>com.hp.ucmdb.federation.mappingEngine.AdapterMappingEngine</default-mapping-engine>

  有关详细信息，请参阅 “reconciliation_rules.txt 文件 （用于向后兼容）”（第 190 页）。

- 通过实施映射引擎接口和并将其余的适配器代码放在 JAR 中，编写您自己的映射引擎。要使用该引擎，请使用以下 XML 标记：
  <default-mapping-engine>com.yourcompany.map.MyMappingEngine</default-mapping-engine>

7 添加实施类路径所需的 JAR

要实施类，请将 **federation_api.jar** 文件添加到代码编辑器类路径中。

8 部署适配器

a 部署适配器包。有关部署包的一般详细信息，请参阅《HP Universal CMDB 管理指南》中的 “包管理器”。

部署包应当包含以下实体：

- 新 CIT 定义 （可选）：

  只有当适配器支持 UCMDB 中不存在的新 CI 类型时，才使用此实体。

  新 CIT 定义位于包的 class 文件夹中。

- 新数据类型定义 （可选）：

  只有当新 CIT 需要新数据类型时，才使用此实体。

  新数据类型定义位于包的 typedef 文件夹中。

- 新有效关系定义 （可选）：

  只有当适配器支持联合 TQL 时，才使用此实体。

  新的有效关系定义位于包的 validlinks 文件夹中。

- 模式配置 XML 文件应当位于包的 discoveryPatterns 文件夹中。

- **描述符。**定义包的定义。

- 将编译类 （通常是 jar 文件）放在包的 **adapterCode\<adapter id>** 文件夹下。

  注意：adapter id 文件夹名称与适配器配置中的值相同。

- 如果您要创建自己的配置文件，则应将文件放在包中的 **adapterCode\<adapter id>** 文件夹下。

9 更新适配器

可以在适配器管理模块中更改适配器的任何非二进制文件。在适配器管理模块中更改配置文件会导致适配器重新加载新配置。

此外，通过编辑包中的文件 （二进制和非二进制文件），然后使用包管理器重新部署包，也可以进行更新。有关详细信息，请参阅《HP Universal CMDB 管理指南》中的 “部署包”。

实施映射引擎

映射引擎的配置取决于所使用的映射引擎。

本任务包括以下步骤：

- “配置 reconciliation_types.txt 文件 （针对 UCMDB 9.0x 默认映射引擎）”（第 254 页）
- “配置 reconciliation_rules.txt 文件 （针对 UCMDB 8.0x 映射引擎）”（第 255 页）

1 配置 reconciliation_types.txt 文件 （针对 UCMDB 9.0x 默认映射引擎）

此文件可定义要用于适配器中的调和的 CI 类型。

在单行中写入用于调和的每个 CI 类型，如下所示：

```
node
business_application
```

将文件放在适配器包中的 **adapterCode\<AdapterID>\META-INF** 文件夹下。

2 配置 reconciliation_rules.txt 文件（针对 UCMDB 8.0x 映射引擎）

此文件用于配置调和规则。文件中的每一行代表一个规则。例如：

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

使用作为调和执行目标的 CI 类型填充 **reconcilition_type** 参数（连接到 TQL 中联合类的 UCMDB 类名称）。

expression 参数是用于确定两个调和对象是否相等的逻辑参数。两个调和对象中一个来自 UCMDB 端，另一个来自联合适配器端。

表达式包含 OR 和 AND。

有关表达式中属性名称的约定是 **[className].[attributeName]**。例如，**ip** 类中的属性 **ip_address** 写为 **ip.ip_address**。

可以定义有序的匹配。有序的匹配首先会检查第一个 OR 子表达式。如果两个调和对象都具有子表达式属性上的值，并且该值返回 false（调和对象不相等），则不比较第二个 OR 子表达式。

对于有序匹配，请使用 “有序表达式”，而不要使用 “表达式”。

抑扬符号 (^) 用于在比较时忽略大小写。

只有当调和数据包含两个节点，且这两个节点并非调和类型的节点（拓扑调和数据）时，才使用其他参数（**end1_type**、**end2_type** 和 **link_type**）。在这种情况下，调和数据是 **end1_type -(link_type)> end2_type**。

由于将通过表达式检索相关布局，所以无需添加相关布局。

要按 UCMDB ID 执行调和，请在表达式中使用 **cmdb_id** 作为属性名称。

将文件放在适配器包中的 **adapterCode\<AdapterID>\META-INF** 文件夹下。

示例：

- 只能为节点 CIT 添加调和规则。这是因为只在节点 CIT 与外部 CIT 之间才存在有效关系。例如， CMDB 中的节点 CI 通过 node.name 属性或通过 ip_address.name 属性与 ServiceCenter 中的节点 CI 相匹配。
- 在这种情况下，调和规则为拓扑规则，且表达式为有序表达式。该规则在进行比较后对 CI 执行以下检查：
  - 如果 node.name 属性相等，则该规则将对节点进行匹配。
  - 如果 node.name 属性不相等，则该规则将不会对节点进行匹配。
  - 如果某个比较的 CI 中的 node.name 属性为 null，则该规则将检查 ip_address.name 属性。如果 ip_address.name 属性相等，则该规则将对节点进行匹配。

创建示例适配器

本示例演示如何创建示例适配器。

本任务包括以下步骤：

- “选择适配器逻辑”（第 257 页）
- “加载项目”（第 257 页）

1 选择适配器逻辑

在实施适配器时，必须选择如何在实施过程中处理条件逻辑（属性条件、ID 条件、调和条件和链接条件）。

a 将完整的数据检索到适配器内存中，以便适配器选择或筛选所需的 CI 实例。

b 将所有条件转换成数据源语言，以便适配器筛选和选择数据。例如：

- 将条件转换为 SQL 查询。
- 将条件转换为 Java API 筛选对象。

c 另外一种方法是：筛选远程服务上的某些数据，然后让适配器选择和筛选其余数据。

在 MyAdapter 示例中使用的是步骤 a 中的逻辑。

2 加载项目

复制 **C:\hp\UCMDB\UCMDBServer\tools\adapter-dev-kit\SampleAdapters** 文件夹中的文件，并按照自述文件中的说明进行操作。

注意：如果将适配器用于大型数据集，则可能需要使用缓存和索引来提高联合的性能。

联机 javadocs 文档位于：

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

参考

XML 配置标记和属性

id="newAdapterIdName"		定义适配器的真实名称。用于查找日志和文件夹
displayName="New Adapter Display Name"		定义适配器在 UI 中的显示名称。
<className>…</className>		定义用于实施 Java 类的适配器接口。
<category >My Category</ category>		定义适配器的类别。
<parameters>		定义在设置新集成点时在 UI 中提供的配置属性。
	name	属性的名称 （主要由代码使用）
	description	属性的提示显示内容
	type	字符串或整数 （使用有效值和布尔字符串）。
	display-name	UI 中的属性名称。
	mandatory	指定此配置属性是否是用户必须设置的属性。
	order-index	属性的位置顺序 （值越小，位置越靠前）
	valid-values	可能的有效值的列表，以 “;” 字符分隔；例如 valid-values= “Oracle;SQLServer;MySQL” 或 valid-values= “True;False”。
<adapterInfo>		包含适配器静态设置和功能的定义。
	<support-federated-query>	将此适配器定义为可支持联合。
	<one-node-topology>	可将多个查询联合到单个联合查询节点。
	<pattern-topology>	可联合多个复杂查询。
	<support-replicatioin-data>	定义用于运行数据推送流程和填充流程的功能。

	<source>	此适配器可用于填充流程。
	<changes-source/>	此适配器可用于填充更改流程。
	<target>	此适配器可用于数据推送流程。
	<default-mapping-engine>	允许定义适配器的映射引擎；默认情况下，适配器将使用默认映射引擎。对于任何其他映射引擎，请输入映射引擎的实施类名称。对于 UCMDB 8.0x 映射引擎，则使用：com.hp.ucmdb.federation.mappingEngine.AdapterMappingEngine
	<removedAttributes>	强制从结果中删除特定属性。
	<full-population-days-interval>	指定执行完整填充作业而不是差异作业的时间 （每隔 “x” 天）。将老化机制用于更改流程。

7

开发推送适配器

本章包括：

概念

任务

参考

概念

推送适配器开发概述

常规推送适配器提供了一个平台，通过该平台可以快速开发用于将 UCMDB 9.0x 数据推送至外部数据库 （数据库和第三方应用程序）的集成。基于常规推送适配器开发自定义集成时，需要：

- UCMDB CI 链接类型和外部数据项之间的 XML 映射文件。
- 将数据项推送到外部数据库的 Jython 脚本。

差异同步

如果 Jython 脚本中推送适配器所基于的 **DiscoveryMain** 方法返回空的 **OSHVResult** 实例，则此适配器不支持差异同步。这意味着即使运行了差异同步作业，实际上也执行的是完全同步。因此，由于会在每次同步时将所有数据添加到 CMDB 中，所以不能在远程系统上更新或删除任何数据。

要使推送适配器支持差异同步， **DiscoveryMain** 函数必须返回用于实施 **DataPushResults** 接口的对象，其中包含 Jython 脚本从 XML 接收的 ID 与 Jython 脚本在远程计算机上创建的 ID 之间的映射。Jython 脚本在远程计算机上创建的 ID 属于 ExternalId 类型。

任务

准备映射文件

可使用两种不同的方法准备映射文件：

- 准备一个全局映射文件。

  将所有映射放到名为 **mappings.xml** 的单个文件中。

- 为每个推送查询准备一个独立的文件。

  每个映射文件的名称为 **< 查询名称 >.xml**。

有关详细信息，请参阅 “映射文件架构”（第 271 页）。

本任务包括以下步骤：

- “创建映射文件”（第 263 页）
- “映射 CI”（第 264 页）
- “映射链接”（第 264 页）

1 创建映射文件

映射文件结构如下

```
<?xml version="1.0" encoding="UTF-8"?>
<integration>
    <info>
        <source name="UCMDB"  versions="9.x" vendor="HP" />
        <!-- for example: -->
        <target name="Oracle" versions="11g" vendor="Oracle" />
    </info>
    <targetcis>
        <!--- CI Mappings --->
    </targetcis>
    <targetrelations>
        <!--- Link Mappings --->
    </ targetrelations>
</integration>
```

2 映射 CI

如以下示例所示映射每个 CMDB CIT：

```
<source_ci_type name="node" mode="update_else_insert">
    <apioutputseq>1</apioutputseq>
    <target_ci_type  name="host">
        <targetprimarykey><pkey>host_key</pkey></targetprimarykey>
        <target_attribute name="host_os" datatype="STRING">
            <map type="direct" source_attribute="discovered_os_name" />
        </target_attribute>
        <!-- more target attributes --->
    </target_ci_type>
</source_ci_type>
```

注意：mode 的可能值取决于脚本的实施。

---

3 映射链接

如以下示例所示映射每个有效链接：

```
<link source_link_type="dependency" target_link_type="dependency"
mode="update_else_insert" source_ci_type_end1="webservice"
source_ci_type_end2="sap_gateway">
    <target_ci_type_end1 name="webservice" />
    <target_ci_type_end2 name="sap_gateway" />
</link>
```

编写 Jython 脚本

映射脚本是一种常规 Jython 脚本，并应遵循 Jython 脚本的规则。有关详细信息，请参阅 “开发 Jython 适配器”（第 63 页）。

该脚本应包含 **DiscoveryMain** 函数，该函数可在成功时返回空的 **OSHVResult** 或 **DataPushResults** 实例。

要报告失败，则该脚本应引发异常，例如：

```
raise Exception('Failed to insert to remote UCMDB using TopologyUpdateService. See log of the remote UCMDB')
```

在 `DiscoveryMain` 函数中，可以通过以下方式获取要为外部应用程序推送或删除的数据项：

```
# get add/update/delete result objects (in XML format) from the Framework
addResult = Framework.getTriggerCIData('addResult')
updateResult = Framework.getTriggerCIData('updateResult')
deleteResult = Framework.getTriggerCIData('deleteResult')
```

可以通过以下方式获取外部应用程序的客户端对象：

```
oracleClient = Framework.createClient()
```

此客户端对象将自动使用由适配器通过框架传递的凭据 ID、主机名和端口号。

如果需要使用您为适配器定义的连接参数 （有关详细信息，请参阅 “生成适配器包”（第 269 页）中的步骤 2），请使用以下代码：

```
propValue = str(Framework.getDestinationAttribute('<Connection Property Name'))
```

例如：

```
serverName = Framework.getDestinationAttribute('ip_address')
```

本节还包括：

- “使用映射结果”（第 266 页）
- “在脚本中处理测试连接”（第 267 页）

使用映射结果

常规推送适配器将创建一些 XML 字符串，用于描述要在目标系统中添加、更新或删除的数据。 Jython 脚本需要分析此 XML，然后对目标执行添加、更新和删除操作。

在 Jython 脚本收到的添加操作 XML 中，在对象和链接的类型、属性或其他信息更改为远程系统的架构之前，这些对象和链接的 mamId 属性始终是原始对象或链接的 UCMDB 标识符。

在针对更新或删除操作的 XML 中，每个对象或链接的 mamId 属性均包含一个字符串，该字符串表示 Jython 脚本从上次同步中返回的 ExternalId。

XML 结果示例

```
<root>
    <data>
        <objects>
            <Object mode="update_else_insert" name="ip" operation="add"
mamId="2ebdc7a93dc7f5bcb33a444763c2a16c">
                <field name="root_lastaccesstime" key="false" datatype="DATE"
length="">1275469266</field>
                <field name="display_label" key="false" datatype="STRING"
length="">16.59.61.67</field>
                <field name="ip_probename" key="false" datatype="STRING"
length="">VMUCMDB05</field>
            </Object>
        </objects>
        <links>
            <link targetRelationshipClass="contained" targetParent="nt"
targetChild="ip" operation="add" mode="update_else_insert"
mamId="8c0a38d53c74c3cc972d6254fb50adba">
                <field name="DiscoveryID1">d5aac653aff428b4a3780111f6389d53</
field>
                <field
name="DiscoveryID2">2ebdc7a93dc7f5bcb33a444763c2a16c</field>
            </link>
        </links>
    </data>
</root>
```

在脚本中处理测试连接

可以调用 Jython 脚本来测试外部应用程序连接。在这种情况下，testConnection 目标属性为 true。可以通过以下方式从框架中获取该属性：

```
testConnection = Framework.getTriggerCIData('testConnection')
```

在测试连接模式下运行脚本时，如果无法与外部应用程序建立连接，脚本将引发异常。如果连接成功，则 **DiscoveryMain** 函数将返回空的 **OSHVResult**。

支持差异同步

重要信息：如果要在使用 9.00 或 9.01 版创建的现有适配器上实施差异同步，则必须使用 9.02 版或更高版本中的 push-adapter.zip 文件来重新创建适配器包。有关详细信息，请参阅 “生成适配器包”（第 269 页）。

此任务支持推送适配器执行差异同步。有关详细信息，请参阅 “差异同步”（第 262 页）。

Jython 脚本返回 **DataPushResults** 对象，该对象包含两个 Java 映射，一个用于对象 ID 映射（键和值是 ExternalCiId 类型对象），另一个用于链接 ID（键和值是 ExternalRelationId 类型对象）。

➤ 将以下 **from** 语句添加到 Jython 脚本中：

```
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
```

➤ 使用 **DataPushResultsFactory** 工厂类从 **DiscoveryMain** 函数获取 **DataPushResults** 对象。

```
# Create the UpdateResult object
updateResult = DataPushResultsFactory.createDataPushResults(objectMappings,
linkMappings);
```

➤ 使用以下命令为 **DataPushResults** 对象创建 Java 映射：

```
# Prepare the maps to store the mappings if IDs
objectMappings = HashMap()
linkMappings = HashMap()
```

➤ 使用 **ExternalIdFactory** 类创建以下 ExternalId ID：

  ➤ 在 CMDB 中生成的对象或链接的 ExternalId（例如，添加操作中的所有 CI 均来自 CMDB）

```
externaCIIId = ExternalIdFactory.createExternalCmdbCiId(ciType, ciIDAsString)
externalRelationId = ExternalIdFactory.createExternalCmdbRelationId(linkType,
end1ExternalCIId, end2ExternalCIId, linkIDAsString)
```

  ➤ 不是在 CMDB 中生成的对象或链接的 ExternalId （通常，每个更新和删除操作均包含此类对象）：

```
myIDField = TypesFactory.createProperty("systemID", "1")
myExternalId = ExternalIdFactory.createExternalCiId(type, myIDField)
```

注意：如果 Jython 脚本已更新现有信息，并且对象 （或链接）的 ID 已更改，则必须返回先前的外部 ID 与新外部 ID 之间的映射。

➤ 可使用 **ExternalIdFactory** 类中的 **restoreCmdbCiIDString** 或 **restoreCmdbRelationIDString** 方法，从在 UCMDB 中生成的对象或链接的外部 ID 中检索 UCMDB ID 字符串。

➤ 使用 **ExternalIdUtil** 类中的 **restoreExternalCiId** 和 **restoreExternalRelationId** 方法，从更新或删除操作 XML 的 mamId 属性值还原 **ExternalId** 对象。

注意：ExternalId 对象实际上是一系列属性。这意味着您可以使用 ExternalId 对象来存储需要用于在远程系统上标识数据的任何信息。

生成适配器包

1 将 **C:\hp\UCMDB\UCMDBServer\content\adapters\push-adapter.zip** 中的内容提取至临时文件夹。

2 编辑 **discoveryPatterns\push_adapter.xml** 文件。

a 使用新 ID 修改 <pattern> 标记并显示标签。将下列代码：

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

替换为

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

b 更新参数列表，使得该参数列表能够反映所需的连接属性。请不要删除 **probeName** 属性。

3 使用步骤 2 中的适配器 ID 重命名 **adapterCode\PushAdapter** 文件夹（例如， **adapterCode\MyPushAdapter**）。

4 使用您编写的脚本替换 **discoveryScripts\pushScript.py**（有关详细信息，请参阅 “编写 Jython 脚本”（第 264 页））。如果重命名该脚本，则应对 **adapterCode\< 适配器 ID>\push.properties** 中的 jythonScript.name 属性将进行相应更新。

5 使用您准备的映射文件替换 **adapterCode\< 适配器 ID>\mappings\ mappings.xml** 文件（有关详细信息，请参阅 “准备映射文件”（第 263 页）。）

如果要为每个 TQL 方法使用一个映射文件，请将相应 TQL 的名称分配给每个 XML 文件，并为文件添加后缀名 **.xml**。在这种情况下，如果没有找到当前 TQL 名称的特定映射文件，则会使用 **mappings.xml** 文件作为默认文件。通过更改 **adapterCode\< 适配器 ID>\push.properties** 中的 mappingFile.default 属性，可以修改默认映射文件的名称。

参考

映射文件架构

元素		属性
名称和路径	**描述**	
integration	定义文件的映射内容。必须位于文件最外层位置，但不能是起始行和任何注释。	
info (integration)	定义有关要集成的数据库的信息	
source (integration > info)	定义有关源数据库的信息	**名称。** type **描述。**源数据库的名称。 **是否必需。**必需 **类型。**字符串
		名称。 versions **描述。**源数据库的版本。 **是否必需。**必需 **类型。**字符串
		名称。 vendor **描述。**源数据库的提供程序。 **是否必需。**必需 **类型。**字符串

元素		属性
名称和路径	**描述**	
target (integration > info)	定义有关目标数据库的信息	**名称**。 type **描述。**源数据库的名称。 **是否必需。**必需 **类型。**字符串
		名称。 versions **描述。**源数据库的版本。 **是否必需。**必需 **类型。**字符串
		名称。 vendor **描述。**源数据库的提供程序。 **是否必需。**必需 **类型。**字符串
targetcis (integration)	所有 CIT 映射的容器元素	

元素		属性
名称和路径	**描述**	
source_ci_type (integration > targetcis)	定义源 CIT	**名称**。 name **描述。**源 CIT 的名称。 **是否必需。**必需 **类型。**字符串
		名称。 mode **描述。**当前 CI 类型所需的更新类型。 **是否必需。**必需 **类型。**以下字符串之一： ➤ **insert** – 只有当 CI 不存在时才使用此字符串。 ➤ **update** – 只有当 CI 存在时才使用此字符串。 ➤ **update_else_insert** – 如果存在 CI，则更新 CI；如果不存在 CI，则创建一个新 CI。 ➤ **ignore** – 不对此 CI 类型执行任何操作。

元素		属性
名称和路径	**描述**	
target_ci_type (integration > targetcis > source_ci_type)	定义目标 CIT	**名称**。 name **描述。**目标 CI 类型名称。 **是否必需。**必需 **类型。**字符串
		名称。 schema **描述。**用于在目标上存储此 CI 类型的架构的名称。 **是否必需。**非必需 **类型。**字符串
		名称。 namespace **描述。**表示此 CI 类型在目标上的命名空间 **是否必需。**非必需 **类型。**字符串
targetprimarykey (integration > targetcis > source_ci_type 或 integration > targetrelations > link)	表示目标 CIT 主键属性	
pkey (integration > targetcis > source_ci_type > targetprimarykey 或 integration > targetrelations > link > targetprimarykey)	标识一个主键属性 只有在模式是 **update** 或 **insert_else_update** 时才是必需的。	

元素		属性
名称和路径	**描述**	
target_attribute (integration > targetcis > source_ci_type 或 integration > targetrelations > link)	定义目标 CIT 的属性	**名称**。 name **描述。**目标 CIT 属性的名称。 **是否必需。**必需 **类型。**字符串
		名称。 datatype **描述。**目标 CIT 属性的数据类型。 **是否必需。**必需 **类型。**字符串
		名称。 length **描述。**适用于字符串 / 字符数据类型，表示目标属性的整数大小。 **是否必需。**非必需 **类型。**整型
		名称。 option **描述。**要应用于值的转换函数。 **是否必需。** False **类型。**以下字符串之一： ➤ **uppercase** – 转换为大写 ➤ **lowercase** – 转换为小写 ➤ 如果此属性为空，则不应用转换函数。

元素		属性
名称和路径	**描述**	
map (integration > targetcis > source_ci_type > target_attribute 或 integration > targetrelations > link > target_attribute)	指定如何获取源 CIT 的属性值	**名称**。 type **描述。**源值和目标值之间的映射类型。 **是否必需。**必需 **类型。**以下字符串之一： ➤ **direct** – 指定从源属性值到目标属性值的一对一映射。 ➤ **compoundstring** – 将多个子元素连接为一个单个字符串，并设置目标属性值。 ➤ **childattr** – 子元素是一个或多个子 CIT 的属性。子 CIT 被定义为具有 **container_f** 或 **contained** 关系的 CIT。 ➤ **constant** – 静态字符串
		名称。 value **描述。**当类型为 **constant** 时，为常量字符串 **是否必需。**只有当类型为 **constant** 时才是必需的 **类型。**字符串
		名称。 attr **描述。**当类型为 **direct** 时，为源属性名称 **是否必需。**只有当类型为 **direct** 时才是必需的 **类型。**字符串

元素		属性
名称和路径	**描述**	
aggregation （integration > targetcis > source_ci_type > target_attribute > map 或 integration > targetrelations > link > target_attribute > map 只有当映射类型为 **childattr** 时才有效）	指定如何将源 CI 的子 CI 属性值合并为一个值，以映射到目标 CI 属性。可选。	**名称**。 type **描述。**聚合函数的类型 **是否必需。**必需 **类型。**以下字符串之一： ➤ **csv** – 将包含的所有值连接为一个以逗号分隔的列表（数字、字符串或字符）。 ➤ **count** – 返回所有包含的值的数字计数。 ➤ **sum** – 返回所有包含的值的数字计数。 ➤ **average** – 返回所有包含的值的数字平均值。 ➤ **min** – 返回包含的数值 / 字符的最小值。 ➤ **max** – 返回包含的数值 / 字符的最大值。

元素		属性
名称和路径	**描述**	
validation （integration > targetcis > source_ci_type > target_attribute > map 或 integration > targetrelations > link > target_attribute > map 只有当映射类型为 **childatt** 时才有效）	允许根据属性值对源 CI 的子 CI 进行排除筛选。与聚合子元素一起使用，以准确了解哪些子属性会映射到目标 CIT 的属性值。可选。	**名称**。 minlength **描述。**排除长度小于指定值的字符串。 **是否必需。**非必需 **类型。**整型
		名称。 maxlength **描述。**排除长度大于指定值的字符串。 **是否必需。**非必需 **类型。**整型
		名称。 minvalue **描述。**排除小于指定值的数字。 **是否必需。**非必需 **类型。**数字
		名称。 maxvalue **描述。**排除大于指定值的数字。 **是否必需。**非必需 **类型。**数字
targetrelations (integration)	所有关系映射的容器元素。可选。	

元素		属性
名称和路径	**描述**	
link (integration > targetrelations)	将源关系映射到目标关系。只有当 **targetrelation** 存在时才会强制执行。	**名称**。 source_link_type **描述。**源关系名称。 **是否必需。**必需 **类型。**字符串
		名称。 target_link_type **描述。**目标关系名称。 **是否必需。**必需 **类型。**字符串
		名称。 nameSpace **描述。**要在目标上创建的链接的命名空间。 **是否必需。**非必需 **类型。**字符串
		名称。 mode **描述。**当前链接所需的更新类型。 **是否必需。**必需 **类型。**以下字符串之一： ➤ **insert** – 只有当 CI 不存在时才使用此字符串。 ➤ **update** – 只有当 CI 存在时才使用此字符串。 ➤ **update_else_insert** – 如果存在 CI，则更新 CI；如果不存在 CI，则创建一个新 CI。 ➤ **ignore** – 不对此 CI 类型执行任何操作。

元素		属性
名称和路径	**描述**	
link （续）		**名称**。 source_ci_type_end1 **描述。**源关系的 End1 CI 类型 **是否必需。**必需 **类型。**字符串
		名称。 source_ci_type_end2 **描述。**源关系的 End2 CI 类型 **是否必需。**必需 **类型。**字符串
target_ci_type_end1 (integration > targetrelations > link)	目标关系的 End1 CI 类型	**名称**。 name **描述。**目标关系的 End1 CI 类型的名称。 **是否必需。**必需 **类型。**字符串
		名称。 superclass **描述。** End1 CI 类型的超类的名称。 **是否必需。**非必需 **类型。**字符串

元素		属性
名称和路径	**描述**	
target_ci_type_end2 (integration > targetrelations > link)	目标关系的 End2 CI 类型	**名称**。 name **描述。**目标关系的 End2 CI 类型的名称。 **是否必需。**必需 **类型。**字符串
		名称。 superclass **描述。** End2 CI 类型的超类的名称。 **是否必需。**非必需 **类型。**字符串

映射结果架构

元素		属性
名称和路径	**描述**	
root	结果文档的根	
data (root)	数据自身的根	
objects (root > data)	要更新的对象的根元素	

元素		属性
名称和路径	**描述**	
Object (root > data > objects)	描述单个对象及其所有属性的更新操作	**名称**。 name **描述。** CI 类型的名称。 **是否必需。**必需 **类型。**字符串
		名称。 mode **描述。**当前 CI 类型所需的更新类型。 **是否必需。**必需 **类型。**以下字符串之一： ➤ **insert** – 只有当 CI 不存在时才使用此字符串。 ➤ **update** – 只有当 CI 存在时才使用此字符串。 ➤ **update_else_insert** – 如果存在 CI，则更新 CI；如果不存在 CI，则创建一个新 CI。 ➤ **ignore** – 不对此 CI 类型执行任何操作。

元素		属性
名称和路径	**描述**	
Object （续）		**名称**。 operation **描述**。要对此 CI 执行的操作。 **是否必需**。必需 **类型**。以下字符串之一： ➤ **add** – 将添加此 CI ➤ **update** – 将更新此 CI ➤ **delete** – 将删除此 CI 如果不设置值，则会使用默认值 **add**。
		名称。 mamId **描述**。源 CMDB 上的对象的 ID。 **是否必需**。必需 **类型**。字符串

元素		属性
名称和路径	**描述**	
field (root > data > objects > Object 或 root > data > links > link)	描述对象的单个字段的值。字段的文本是字段中的新值，如果字段中包含链接，则值为某个端的 ID。每一端的 ID 均显示为对象 （在 < 对象 > 下）。	**名称**。 name **描述**。字段的名称。 **是否必需**。必需 **类型**。字符串
		名称。 key **描述**。确定该字段对于对象是否关键。 **是否必需**。必需 **类型**。布尔型
		名称。 datatype **描述**。字段的类型。 **是否必需**。必需 **类型**。字符串
		名称。 length **描述**。适用于字符串 / 字符数据类型，是目标属性的整数大小。 **是否必需**。非必需 **类型**。整型

元素		属性
名称和路径	**描述**	
links (root > data)	要更新的链接的根元素	**名称**。 targetRelationshipClass **描述** . 目标系统中关系 （链接）的名称。 **是否必需** . 必需 **类型** . 字符串
		名称。 targetParent **描述** . 链接的第一个端 （父项）的类型。 **是否必需** . 必需 **类型** . 字符串
		名称。 targetChild **描述** . 链接的第二个端 （子项）的类型。 **是否必需** . 必需 **类型** . 字符串

元素		属性
名称和路径	**描述**	
links （续）		**名称**。 mode **描述**。当前 CI 类型所需的更新类型。 **是否必需**。必需 **类型**。以下字符串之一： ➤ **insert** – 只有当 CI 不存在时才使用此字符串。 ➤ **update** – 只有当 CI 存在时才使用此字符串。 ➤ **update_else_insert** – 如果存在 CI，则更新 CI；如果不存在 CI，则创建一个新 CI。 ➤ **ignore** – 不对此 CI 类型执行任何操作。
		名称。 operation **描述**。要对此 CI 执行的操作。 **是否必需**。必需 **类型**。以下字符串之一： ➤ **add** – 将添加此 CI ➤ **update** – 将更新此 CI ➤ **delete** – 将删除此 CI 如果不设置值，则会使用默认值 **add**。
		名称。 mamId **描述**。源 CMDB 上的对象的 ID。 **是否必需**。必需 **类型**。字符串

第 II 部分

使用 API

8

API 简介

本章包括：

概念

概念

API 概述

HP Universal CMDB 中附带的 API 如下：

- **UCMDB Web Service API**. 支持将配置项定义和拓扑关系写入 UCMDB（通用配置管理数据库），并查询含有 TQL 和特殊查询的信息。有关详细信息，请参阅 “HP Universal CMDB Web 服务 API”（第 291 页）。
- **UCMDB Java API**. 描述第三方或自定义工具如何使用 Java API 来提取数据和计算，以及如何将数据写入 UCMDB（通用配置管理数据库）。有关详细信息，请参阅 “HP Universal CMDB API”（第 377 页）。

9

HP Universal CMDB Web 服务 API

本章包括：

概念

任务

参考

概念

约定

本章使用以下约定：

- **UCMDB** 是指通用配置管理数据库本身，而 **HP Universal CMDB** 是指应用程序。
- UCMDB 元素和方法参数以在架构中指定的方式进行拼写。元素或方法参数不必大写。例如， relation 是传递到某方法的类型 Relation 的元素。

HP Universal CMDB Web 服务 API 概述

请将本章与联机文档库中的 UCMDB 架构文档配合使用。

HP Universal CMDB Web 服务 API 用于将应用程序与 HP Universal CMDB (UCMDB) 集成。通过该 API 可以：

- 在 CMDB 中添加、删除以及更新 CI 和关系
- 检索有关类模型的信息
- 检索影响分析
- 检索有关配置项和关系的信息
- 管理认证：查看、添加、更新和删除
- 管理作业：查看状态、激活和停用
- 管理 Probe 范围：查看、添加和更新
- 管理触发器：添加或删除触发器 CI，以及添加、删除或禁用触发器 TQL
- 查看域和 Probe 的常规数据

通常情况下，用于检索有关配置项和关系信息的方法会使用拓扑查询语言 (TQL)。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的“拓扑查询语句”。

HP Universal CMDB Web 服务 API 的用户必须熟悉：

- SOAP 规范
- 面向对象的编程语言，如 C++、 C# 或 Java
- HP Universal CMDB
- 数据流管理

本节包括以下主题：

- “API 的用途”（第 293 页）
- “权限”（第 294 页）

API 的用途

API 用于满足多种业务需求。例如：

- 第三方系统可以查询类模型，了解有关可用配置项 (CI) 的信息。
- 第三方资产管理工具可以使用仅适用于自己的信息更新 CMDB，从而将自己的数据与 HP 应用程序收集的数据统一。
- 大量第三方系统可以通过填充 CMDB 来创建一个可跟踪更改并执行影响分析的中心 CMDB。
- 第三方系统可以按照自己的业务逻辑创建实体和关系，然后将数据写入 CMDB 以利用 CMDB 查询功能。
- 其他系统 （如版本控制 (CCM) 系统）可以使用影响分析方法进行更改分析。

权限

管理员负责提供用于连接 Web 服务的登录凭据。所需的凭据取决于是将 HP Universal CMDB 作为独立应用程序使用，还是在 Business Service Management 内部使用：

- **独立 HP Universal CMDB。**使用已获得搜寻资源和集成资源权限的 UCMDB 用户的凭据进行登录。

  有关详细信息，请参阅 《HP Universal CMDB 管理指南》中的 “安全管理器”页面”。

- **Business Service Management 中嵌入的 HP Universal CMDB**。使用 Business Service Management 用户的凭据登录。该用户必须已获得访问 Business Service Management 中 HP Universal CMDB 资源的相关权限。

通过 HP Universal CMDB 分配权限后，权限级别为 “查看”、“更新”和 “执行”。使用 Business Service Management 分配权限后，权限级别将会是“查看”和 “更新”，其中 “更新”包括 “执行”。要查看每个操作所需的权限，请参考每个操作的请求文档，参阅 《数据流管理 Schema Reference》。

HP Universal CMDB Web 服务 API 参考

有关请求和响应结构的完整文档，请参阅 HP UCMDB Web 服务 API 参考。这些文件位于以下文件夹中：

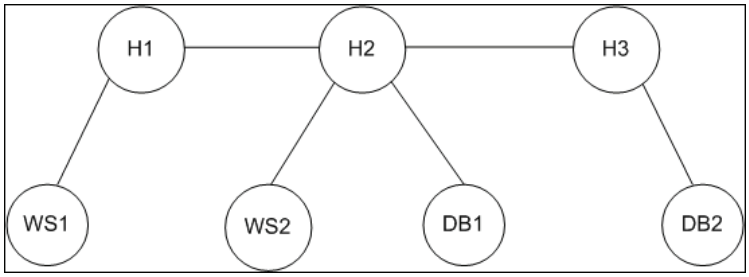
C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\doc_lib\DevRef_guide\CMDB_Schema\webframe.html

返回清晰拓扑图元素

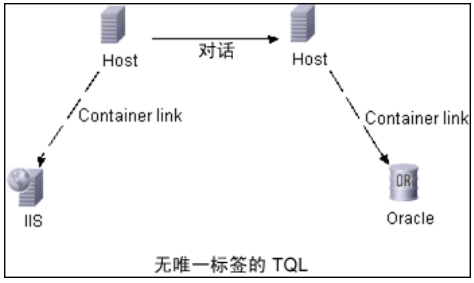
返回 topology 或 topologyMap 元素中数据的查询方法会在系统中搜索 TQL 查询的匹配项。下图所示为在查询中使用唯一标签时，如何对生成的 topology 和 topologyMap 结构造成影响。

标签是在特定配置下查询关系和配置项时用户指定的名称。查询中指定的标签将在返回图中用作节点标签。如果未指定标签，则系统将使用 CI 或 Relation 类型名称作为结果图中的标签。以下示例指定标签 IISHost 和 DBHost 来替换默认的 Host 标签， ContainerIIS 和 ContainsDB 来替换默认的 Container Link 标签。

以下示例为一个小型 IT 世界模型。模型中包含三个主机：H1、H2、H3，它们承载了 Web 服务器 (WS) 和数据库管理器 (DB)。 WS1 位于 H1 上，DB1 和 WS2 位于 H2 上，DB2 位于 H3 上。



此查询使用默认标签进行定义：



在 IT 世界上运行此 TQL 查询既可以生成 Topology 元素，又可以生成 TopologyMap 元素。

拓扑响应

```
CIs: H1, H2, H3, WS1, WS2, DB1, DB2
Relations: H1-WS1, H1-H2, H2-H3, WS2-H2, DB1-H2, DB2-H3
```

TopologyMap 响应

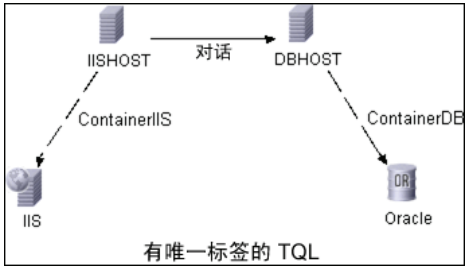
```
CINode:
    label: Host
    CIs: H1, H2
CINode:
    label: Host
    CIs: H2, H3
CINode:
    label: DB
    CIs: DB1, DB2
CINode:
    label: Webserver
CIs: IIS
relationNode:
    label: talk
    relations: H1-H2, H2-H3

relationNode:
    label: Container Link
    relations: WS1-H1, WS2-H2
relationNode:
    label: Container Link
    relations: DB2-H3, DB1-H2
```

在上述 TopologyMap 响应中，前两个 CINode 包含一个相同的 Host 标签，对应于查询中的两个 Host CI。这两个 CINode 均包含主机 H2，但并没有给出 H2 重复的原因。

最后两个 relationNode 包含相同的 Contained 标签，对应于查询中的两个 Container link 关系。

出现重复的原因是查询中没有指定唯一标签，导致系统在图中使用默认标签（类型名称为 Host 和 Container）。要提取一个更易用的拓扑图，请使用唯一标签为每个要匹配的配置定义查询，如以下查询中所示：



此 topology 结果与不包含唯一标签的 TQL 的结果相同。但是，它们的 topologyMap 结果却不同：此时每个标签都是唯一的。

```
CINode:
    label: IISHOST
    CIs: H1, H2
CINode:
    label: DBHOST
    CIs: H2, H3

...

relationNode:
    label: ContainerIIS
    relations: WS1-H1, WS2-H2
relationNode:
    label: ContainerDB
    relations: DB2-H3, DB1-H2
```

在此图中，很清楚地说明了两次返回 H2 的原因。唯一标签表示 H2 一次是作为 Web 服务器主机返回，另一次是作为数据库主机返回。

提示：因此，请尽可能在 CMDB 中对特定配置使用唯一的用户定义标签。

任务

调用 Web 服务

您可以使用 HP Universal CMDB Web 服务中的标准 SOAP 编程技术来调用服务器端的方法。如果语句无法解析或者调用方法时出现问题，则 API 方法将会引发 SoapFault 异常。引发 SoapFault 异常后，UCMDB 将填充一条或多条错误消息，以及一个或多个错误代码和异常消息字段。如果没有错误，则会返回调用的结果。

SOAP 程序员可通过以下地址访问 WSDL：

http://< 服务器 >[:port]/axis2/services/UcmdbService?wsdl

仅需为非标准安装指定端口。请咨询系统管理员获取正确的端口号。

用于调用服务的 URL 为：

http://< 服务器 >[:port]/axis2/services/UcmdbService

有关连接到 CMDB 的示例，请参阅 “用例”（第 331 页）。

查询 CMDB

您可以使用 “UCMDB 查询方法”（第 308 页）中所述的 API 查询 CMDB。

查询和已返回的 CMDB 元素始终包含真实的 UMDB ID。

有关使用查询方法的示例，请参阅 “查询示例”（第 337 页）。

本节包括以下主题：

及时响应计算

对于所有查询方法，UMDB 服务器会在收到查询方法的请求后计算该请求的值，并基于最新数据返回结果。即使 TQL 查询处于活动状态，而且之前的计算结果已存在，服务器也将始终在收到请求的同时计算结果。因此，运行某个返回到客户端应用程序的查询后，结果可能与用户界面上显示的同一个查询的结果不同。

提示：如果您的应用程序多次使用指定查询的结果，而且数据在两次使用结果数据期间未出现大幅更改，则您可以使用客户端应用程序存储数据，而无须重复运行查询，从而提高应用程序的性能。

处理大量响应

即使当前并未传输实际数据，对查询的响应也始终包括查询方法所请求的数据的结构。对于数据为集合或图的各种方法，响应还包括 ChunkInfo 结构，由 chunksKey 和 numberOfChunks 组成。numberOfChunks 字段表示包含必须检索的数据的块数。

数据的最大传输大小由系统管理员设置。如果从查询返回的数据大于最大大小，则第一个响应中的数据结构将不包含任何有意义的信息，而且 numberOfChunks 字段的值将会大于等于 2。如果数据未超过最大大小，则 numberOfChunks 字段的值将为 0（零），并且数据将在第一个响应中传输。因此，处理响应时应首先检查 numberOfChunks 值。如果值大于 1，则应放弃传输中的数据并请求数据块。否则，请使用响应中的数据。

有关处理块数据的信息，请参阅 “pullTopologyMapChunks”（第 320 页）和“releaseChunks”（第 321 页）。

指定要返回的属性

CI 和关系通常包含多种属性。某些返回这些项的集合或图形的方法可以接受输入参数，这些参数用于指定要与匹配查询的各项一同返回的属性值。CMDB 不会返回空属性。因此，对某查询的响应所包含的属性可能比查询中请求的属性要少。

本节描述了用于指定要返回的属性的集合类型。

引用属性的方式有两种：

- 按名称
- 按预定义属性规则的名称。预定义属性规则由 CMDB 使用，用于创建真实属性名称的列表。

当某个应用程序按名称引用属性时，该应用程序会传递一个 PropertiesList 元素。

提示：请尽可能使用 PropertiesList 而非基于规则的集合来指定所需属性的名称。使用预定义属性规则通常会导致返回的属性数量超过所需量，并损失一部分性能。

预定义属性包含两种类型：限定符属性和简单属性。

- **限定符属性**. 当客户端应用程序应传递 QualifierProperties 元素（可应用于属性的限定符列表）时使用。CMDB 会将由客户端应用程序传递的限定符列表转换为至少有一个限定符适用的属性列表。这些属性的值将与 CI 或 Relation 元素一同返回。
- **简单属性**. 要使用基于规则的简单属性，客户端应用程序会传递一个 SimplePredefinedProperty 或 SimpleTypedPredefinedProperty 元素。这些元素包含 CMDB 生成要返回的属性列表所依据规则的名称。可以在 SimplePredefinedProperty 或 SimpleTypedPredefinedProperty 元素中指定的规则包括 CONCRETE、 DERIVED 和 NAMING。

具体属性

具体属性是为指定 CIT 定义的属性集合。派生类的实例将不返回由这些派生类所添加的属性。

某个方法返回的实例集合可能包含在该方法调用中指定的某个 CIT 的实例，以及从此 CIT 继承的 CIT 的实例。派生 CIT 将会继承指定 CIT 的属性。此外，派生 CIT 可通过添加属性扩展父 CIT。

具体属性示例：

CIT T1 包含属性 P1 和 P2。 CIT T11 从 T1 继承，并使用属性 P21 和 P22 扩展 T1。

类型为 T1 的 CI 的集合包括 T1 和 T11 的实例。该集合中所有实例的具体属性都包含 P1 和 P2。

派生属性

派生属性是为指定 CIT 和每个派生 CIT 所定义的属性以及由每个派生 CIT 所添加属性的集合。

派生属性示例：

继续具体属性的示例，T1 实例的派生属性为 P1 和 P2。T11 实例的派生属性为 P1、 P2、 P21 和 P22。

命名属性

命名属性包含 display_label 和 data_name。

其他属性规范元素

- **PredefinedProperties**

  对于其他每项可能的规则，PredefinedProperties 可以包含 QualifierProperties 元素和 SimplePredefinedProperty 元素。 PredefinedProperties 集合不一定包含列表中的所有类型。

- **PredefinedTypedProperties**

  PredefinedTypedProperties 用于将不同的属性集合应用到每个 CIT。对于其他每项适用的规则，PredefinedTypedProperties 可以包含 QualifierProperties 元素和 SimpleTypedPredefinedProperty 元素。因为 PredefinedTypedProperties 已分别应用到每个 CIT，所以派生属性并不适用。PredefinedProperties 集合不一定包含列表中的所有适用类型。

- **CustomProperties**

  CustomProperties 可以包含基本 PropertiesList 和基于规则的属性列表的任意组合。属性筛选器是所有列表返回的所有属性的联合。

- **CustomTypedProperties**

  CustomTypedProperties 可以包含基本 PropertiesList 和基于规则的适用属性列表的任意组合。属性筛选器是所有列表返回的所有属性的联合。

➤ **TypedProperties**

TypedProperties 用于传递每个 CIT 的不同属性集合。TypedProperties 是所有类型的类型名称和属性集合组成的对集合。每个属性集合仅适用于相应的类型。

更新 UCMDB

您可以使用更新 API 对 CMDB 进行更新。有关 API 方法的详细信息，请参阅“UCMDB 更新方法”（第 322 页）。

有关使用更新方法的示例，请参阅 “更新示例”（第 354 页）。

本任务包括以下步骤：

➤ “UCMDB 更新参数”（第 303 页）

➤ “使用 ID 类型和更新方法”（第 304 页）

➤ “UCMDB 更新方法”（第 322 页）

UCMDB 更新参数

本主题讲述了仅由服务的更新方法使用的参数。有关详细信息，请参阅架构文档。

CIsAndRelationsUpdates

CIsAndRelationsUpdates 类型由 CIsForUpdate、 relationsForUpdate、referencedRelations 和 referencedCIs 组成。CIsAndRelationsUpdates 实例不一定包括所有三个元素。

CIsForUpdate 是一个 CI 集合。relationsForUpdate 是一个 Relations 集合。这些集合中的 CI 和 relation 元素均包含一个 props 元素。创建 CI 或关系时，必须使用值填充包含 CI 类型定义中的 required 或 key 属性的各种属性。这些集合中的项由方法更新或创建。

referencedCIs 和 referencedRelations 是 CMDB 中已定义 CI 的集合。该集合中的元素与所有密钥属性一起，通过临时 ID 标识。这些项用于解决要更新的 CI 和关系的标识。它们并不由方法创建或更新。

这些集合中的每个 CI 和 relation 元素均包含一个属性集合。在这些集合中，新项将随属性值一起创建。

使用 ID 类型和更新方法

下文描述了 ID CIT 以及 CI 和关系。当 ID 不是真实的 CMDB ID 时，便需要类型和密钥属性。

删除或更新配置项

当调用某个方法删除或更新某一项时，客户端可能会使用临时 ID 或空 ID。在这种情况下，必须设置标识 CI 的 CI 类型和密钥属性。

删除或更新关系

删除或更新关系时，关系 ID 可以为空、临时或真实 ID。

如果某个 CI 的 ID 为临时 ID，则该 CI 必须在 referencedCIs 集合中传递，而且必须指定其密钥属性。有关详细信息，请参阅 “CIsAndRelationsUpdates”（第 303 页）中的 referencedCIs。

将新配置项插入 CMDB

可以使用空 ID 或临时 ID 插入新 CI。但是，如果 ID 为空，则服务器无法返回结构 createIDsMap 中的真实 CMDB ID，因为 clientID 不存在。有关详细信息，请参阅 “addCIsAndRelations”（第 322 页）和 “UCMDB 查询方法”（第 308 页）。

将新关系插入 CMDB

关系 ID 可以是临时 ID，也可以是空 ID。但是，如果关系为新关系，但关系任一端的配置项已在 CMDB 中定义，则已退出的 CI 必须按真实 CMDB ID 标识，或在某个 referencedCIs 集合中指定。

查询 UCMDB 类模型

类模型方法会返回有关 CIT 和关系的信息。类模型可以使用 CI 类型管理器进行配置。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的“CI 类型管理器”。

有关使用类模型方法的示例，请参阅 “类模型示例”（第 358 页）。

本节提供以下方法的信息，用于返回有关 CIT 和关系的信息：

- “getClassAncestors”（第 305 页）
- “getAllClassesHierarchy”（第 306 页）
- “getCmdbClassDefinition”（第 306 页）

getClassAncestors

getClassAncestors 方法会检索给定 CIT 及其根之间的路径，包括根。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
className	类型名称。有关详细信息，请参阅 “类型名称”（第 372 页）。

输出

参数	注释
classHierarchy	成对的类名称和父类名称的集合。
comments	仅供内部使用。

getAllClassesHierarchy

getAllClassesHierarchy 方法会检索整个类模型树。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。

输出

参数	注释
classesHierarchy	成对的类名称和父类名称的集合。
comments	仅供内部使用。

getCmdbClassDefinition

getCmdbClassDefinition 方法会检索有关指定类的信息。

如果使用 getCmdbClassDefinition 检索密钥属性，还必须从父类向上查询到基类。getCmdbClassDefinition 作为密钥属性只对包含在按 className 指定的类定义中设置的 ID_ATTRIBUTE 的属性进行标识。继承的密钥属性不会被识别为指定类的密钥属性。因此，指定类的密钥属性的完整列表是指该类及其父类 （向上到根）的所有密钥联合。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
className	类型名称。有关详细信息，请参阅 “类型名称”（第 372 页）。

输出

参数	注释
cmdbClass	类定义由 name、 classType、 displayLabel、 description、 parentName、限定符和属性组成。
comments	仅供内部使用。

查询影响分析

影响分析方法中的 Identifier 指向服务的响应数据。对于当前响应而言，它是唯一的，如果 10 分钟内未使用，将会从服务器的内存缓存中丢弃。

有关使用影响分析方法的示例，请参阅 “影响分析示例”（第 360 页）。

参考

UCMDB 查询方法

本节提供以下方法的信息：

executeTopologyQueryByName

executeTopologyQueryByName 方法可检索与指定查询相匹配的拓扑图。

提示：拓扑图还包含其他信息，如果 TQL 中每个 CINode 和 relationNode 的标签都唯一，则了解这些信息将会更容易。有关详细信息，请参阅 “返回清晰拓扑图元素”（第 295 页）。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
queryName	CMDB 中用于检索图的 TQL 名称。
queryTypedProperties	特定配置项类型的项要检索的属性集的集合。

输出

参数	注释
topologyMap	有关详细信息，请参阅 “TopologyMap”（第 374 页）。

executeTopologyQueryByNameWithParameters

executeTopologyQueryByNameWithParameters 方法可检索与指定参数化查询相匹配的 topologyMap 元素。

查询参数的值将在 parameterizedNodes 参数中传递。指定的 TQL 必须包含为每个 CINode 和 relationNode 定义的唯一标签，否则，方法调用将会失败。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
queryName	CMDB 中用于获取图的参数化 TQL 的名称。
parameterizedNodes	要包含在查询结果中，且每个节点都必须符合的条件。
queryTypedProperties	特定配置项类型的项要检索的属性集的集合。

输出

参数	注释
topologyMap	有关详细信息，请参阅 “TopologyMap”（第 374 页）。
chunkInfo	有关详细信息，请参阅：“ChunkInfo”（第 375 页） “处理大量响应”（第 299 页）

executeTopologyQueryWithParameters

executeTopologyQueryWithParameters 方法可检索与参数化查询相匹配的 topologyMap 元素。

该参数化查询将在 queryXML 参数中传递。查询参数的值将在 parameterizedNodes 参数中传递。TQL 必须包含为每个 CINode 和 relationNode 定义的唯一标签。

executeTopologyQueryWithParameters 方法用于传递特殊查询，而不是访问 CMDB 中定义的查询。当您无法通过访问 UCMDB 用户界面对某个查询进行定义，或不希望将该查询保存到数据库时，可以使用此方法。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
queryXML	XML 字符串，表示一个不包含资源标记的 TQL。
parameterizedNodes	要包含在查询结果中，且每个节点都必须符合的条件。

输出

参数	注释
topologyMap	有关详细信息，请参阅 “TopologyMap”（第 374 页）。
chunkInfo	有关详细信息，请参阅 “ChunkInfo”（第 375 页）和 “处理大量响应”（第 299 页）。

getChangedCIs

getChangedCIs 方法会返回所有与指定 CI 相关的 CI 的更改数据。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
ids	系统将检查其相关 CI 的更改情况的根 CI 的 ID 列表。在此集合中，只有真实 CMDB ID 才有效。
fromDate	期间的开始日期，用于检查 CI 在这期间是否更改。
toDate	期间的结束日期，用于检查 CI 在这期间是否更改。

输出

参数	注释
changeDataInfo	ChangedDataInfo 元素的零个或更多个集合。

getCINeighbours

getCINeighbours 方法会返回指定 CI 的直接相邻项。

例如，如果查询位于 CI A 的相邻项上，并且 CI A 包含使用 CI C 的 CI B，则会返回 CI B，但不会返回 CI C。也就是说，只有指定类型的相邻项才会返回。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
ID	用于检索相邻项的 CI 的 ID。该 ID 必须是一个真实的 CMDB ID。
neighbourType	要检索的相邻项的 CIT 名称。所指定类型的相邻项以及由该类型派生类型的相邻项都将返回。有关详细信息，请参阅 “类型名称”（第 372 页）。
CIProperties	每个配置项上要返回的数据，已调用用户界面中的查询布局。有关详细信息，请参阅 “TypedProperties”（第 303 页）。
relationProperties	针对每种关系要返回的数据 （已调用用户界面中的查询布局）。有关详细信息，请参阅 “TypedProperties”（第 303 页）

输出

参数	注释
topology	有关详细信息，请参阅 “Topology”（第 374 页）。
comments	仅供内部使用。

getCIsByID

getCIsByID 方法按配置项的 CMDB ID 对它们进行检索。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
CIsTypedProperties	已键入的属性集合。有关详细信息，请参阅 “其他属性规范元素”（第 302 页）。
IDs	此集合中只有真实 CMDB ID 才是有效的。

输出

参数	注释
CIs	CI 元素的集合。
chunkInfo	有关详细信息，请参阅：“ChunkInfo”（第 375 页）“处理大量响应”（第 299 页）

getCIsByType

getCIsByType 方法会返回指定类型以及从该指定类型继承的所有类型的配置项集合。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
type	类名称。有关详细信息，请参阅 “类型名称”（第 372 页）。
properties	每个配置项上要返回的数据。有关详细信息，请参阅 “CustomProperties”（第 302 页）。

输出

参数	注释
CIs	CI 元素的集合。
chunkInfo	有关详细信息，请参阅：“ChunkInfo”（第 375 页） “处理大量响应”（第 299 页）

getFilteredCIsByType

getFilteredCIsByType 方法会检索符合该方法使用条件的指定类型的 CI。条件的组成部分如下：

- 一个包含属性名称的名称字段
- 一个包含比较运算符的运算符字段
- 一个包含值或值列表的可选值字段

它们组合在一起便是布尔表达式：

```
<item>.property.value [operator] <condition>.value
```

例如，如果条件名称为 root_actualdeletionperiod，条件值为 40，运算符为 Equal，则布尔语句为：

```
<item>.root_actualdeletionperiod.value = = 40
```

假定不存在其他条件，查询将返回 root_actualdeletionperiod 为 40 的所有项。

如果 conditionsLogicalOperator 参数为 AND，则查询将返回符合 conditions 集合中所有条件的项。如果 conditionsLogicalOperator 为 OR，则查询将返回至少符合 conditions 集合中一个条件的项。

下表列出了各种比较运算符：

运算符	条件 / 注释类型
ChangedDuring	日期 用于范围检查。条件值以小时为单位指定。如果日期属性的值处于调用方法的时间加上或减去条件值的时间范围内，则条件为 True。 例如，如果条件值为 24，而且日期属性的值介于昨天此时和明天此时之间，则条件为 True。 **注意：**名称 ChangedDuring 保留向后兼容的特性。在之前的版本中，运算符只能在创建和修改时间属性时使用。
Equal	字符串和数字
EqualIgnoreCase	字符串
Greater	数字
GreaterEqual	数字
In	字符串、数字和列表 该条件的值是一个列表。如果属性值为列表中的某一个值，则条件为 True。
InList	列表 条件和属性的值均为列表。 如果条件列表中的所有值同样也显示在项的属性列表中，则条件为 True。条件中可包含比指定数量更多的属性值，而会不影响条件的真假。

运算符	条件 / 注释类型
IsNull	字符串、数字和列表 该项的属性不包含任何值。使用运算符 IsNull 时，条件值会被忽略，而且某些情况下条件值可以是 nil。
Less	数字
LessEqual	数字
Like	字符串 该条件的值是属性值的一个子字符串。条件值必须用百分比符号 (%) 括起来。例如，%Bi% 与 Bismark 和 Bay of Biscay 相匹配，但不与 biscuit 匹配。
LikeIgnoreCase	字符串 Like 运算符的使用方法与 LikeIgnoreCase 运算符相同。但是，匹配不区分大小写。因此， %Bi% 与 biscuit 匹配。
NotEqual	字符串和数字
UnchangedDuring	日期 用于范围检查。条件值以小时为单位指定。如果日期属性的值处于调用方法的时间加上或减去条件值的时间范围内，则条件为 False。如果在该范围之外，则条件为 True。 例如，如果条件值为 24，而且日期属性的值在昨天此时之前或明天此时之后，则条件为 True。 **注意：**名称 UnchangedDuring 保留向后兼容的特性。在之前的版本中，运算符只能在创建和修改时间属性时使用。

关于设置条件的示例：

```
FloatCondition fc = new FloatCondition();
FloatProp fp = new FloatProp();
fp.setName("attr_name");
fp.setValue(11);
fc.setCondition(fp);
fc.setFloatOperator(FloatCondition.floatOperatorEnum.Equal);
```

关于查询继承属性的示例：

目标 CI 为 sample，包含 name 和 size 两个属性。 sampleII 使用 level 和 grade 这两个属性扩展 CI。该示例通过按名称指定从 sample 继承的 sampleII 属性的方式，为这些属性设置了一个查询。

```
GetFilteredCIsByType request = new GetFilteredCIsByType()
request.setCmdbContext(cmdbContext)
request.setType("sampleII")
CustomProperties customProperties = new CustomProperties();
PropertiesList propertiesList = new PropertiesList();
propertiesList.addPropertyName("name");
propertiesList.addPropertyName("size");
customProperties.setPropertiesList(propertiesList);
request.setProperties(customProperties)
```

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
type	类名称。有关详细信息，请参阅 “类型名称”（第 372 页）。该类型可以是使用 CI 类型管理器定义的任意类型。有关详细信息，请参阅 《HP Universal CMDB 建模指南》中的 “CI 类型管理器”。
properties	针对每个 CI 要返回的数据 （已调用用户界面中的查询布局）。有关详细信息，请参阅 “CustomProperties”（第 302 页）。

参数	注释
conditions	“名称 - 值”对和互相关联运算符的集合。例如，host_hostname like QA。
conditionsLogicalOperator	➤ **AND**。必须符合所有条件。 ➤ **OR**。必须至少符合一个条件。

输出

参数	注释
CIs	CI 元素的集合。
chunkInfo	有关详细信息，请参阅 “ChunkInfo”（第 375 页）和 “处理大量响应”（第 299 页）。

getQueryNameOfView

getQueryNameOfView 方法会检索作为指定视图基础的 TQL 的名称。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
viewName	视图的名称，即 CMDB 中类模型的子集合。

输出

参数	注释
queryName	CMDB 中作为视图基础的 TQL 的名称。

getTopologyQueryExistingResultByName

getTopologyQueryExistingResultByName 方法用于检索指定 TQL 的最新运行结果。该调用不会运行 TQL。如果上一次运行没有结果，则不会返回任何内容。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
queryName	TQL 的名称。
queryTypedProperties	特定配置项类型的项要检索的属性集的集合。

输出

参数	注释
queryName	CMDB 中作为视图基础的 TQL 的名称。

getTopologyQueryResultCountByName

getTopologyQueryResultCountByName 方法会检索与指定查询相匹配的每个节点的实例数。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
queryName	TQL 的名称。
countInvisible	如果为 True，则输出将包含查询中定义为不显示的 CI。

输出

参数	注释
queryName	CMDB 中作为视图基础的 TQL 的名称。

pullTopologyMapChunks

pullTopologyMapChunks 方法会检索一个包含方法响应的块。

每个块均包含一个 topologyMap 元素，该元素是此方法响应的一部分。第一个块的编号为 1，因此检索循环计数器将 1 到 < 响应对象 >.getChunkInfo(). getNumberOfChunks() 进行迭代。

有关详细信息，请参阅 “ChunkInfo”（第 375 页）和 “查询 CMDB”（第 298 页）。

客户端应用程序必须能够处理部分图。请查看以下处理 CI 集合的示例，以及在 “查询示例”（第 337 页）中将块合并到某个图的示例。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
ChunkRequest	要检索的块和由查询方法返回的 ChunkInfo 的数量。

输出

参数	注释
topologyMap	有关详细信息，请参阅 “TopologyMap”（第 374 页）。
comments	仅供内部使用。

关于处理块的示例：

```
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
               }
          }
}
```

releaseChunks

releaseChunks 方法会释放包含查询数据的块的内存。

提示：服务器会在十分钟后丢弃这些数据。因此，在读取数据之后，立即调用此方法丢弃这些数据可节省服务器资源。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
chunksKey	服务器上已分块数据的标识符。密钥是 ChunkInfo 的一个元素。

UCMDB 更新方法

本节提供以下方法的信息：

- “addCIsAndRelations”（第 322 页）
- “addCustomer”（第 323 页）
- “deleteCIsAndRelations”（第 324 页）
- “removeCustomer”（第 324 页）
- “updateCIsAndRelations”（第 324 页）

addCIsAndRelations

addCIsAndRelations 方法可添加或更新 CI 和关系。

如果 CI 或关系在 CMDB 中不存在，请进行添加，并根据 CIsAndRelationsUpdates 参数的内容设置它们的属性。

如果 CI 或关系存在于 CMDB 中，则当 updateExisting 为 **True** 时，系统将使用新数据对它们进行更新。

如果 updateExisting 为 **False**，则 CIsAndRelationsUpdates 无法引用现有的配置项或关系。如尝试在 updateExisting 为 False 时引用现有项，则会导致异常。

如果 updateExisting 为 **True**，则无论 ignoreValidation 的值如何，系统都会在不验证 CI 的情况下执行添加或更新操作。

如果 updateExisiting 为 **False**，但 ignoreValidation 为 **True**，则系统会执行添加操作，但不会验证 CI。

如果 updateExisiting 为 **False**，而且 ignoreValidation 也为 **False**，则系统会在执行添加操作前验证 CI。

关系始终不会被验证。

CreatedIDsMap 是连接客户端临时 ID 与相应的真实 CMDB ID 的 ClientIDToCmdbID 类型的映射或词典。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
updateExisting	如果设置为 True，则可更新 CMDB 中已存在的项。如果已有项存在，则设置为 False 将会引发异常。
CIsAndRelationsUpdates	要更新或创建的项。有关详细信息，请参阅 “CIsAndRelationsUpdates”（第 303 页）。
ignoreValidation	如果为 True，则在更新 CMDB 之前不会执行检查。

输出

参数	注释
CreatedIDsMap	客户端 ID 到 CMDB ID 的映射。有关详细信息，请参阅 “addCIsAndRelations”（第 322 页）。
comments	仅供内部使用。

addCustomer

addCustomer 方法可添加一个客户。

输入

参数	注释
CustomerID	客户的数字 ID。

deleteCIsAndRelations

deleteCIsAndRelations 方法会从 CMDB 中删除指定配置项和关系。

对于一个或多个 Relation 项某一端的 CI，如果删除该 CI，则这些 Relation 项也将删除。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
CIsAndRelationsUpdates	要删除的项。有关详细信息，请参阅 “CIsAndRelationsUpdates”（第 303 页）

removeCustomer

removeCustomer 方法可删除客户记录。

输入

参数	注释
CustomerID	客户的数字 ID。

updateCIsAndRelations

updateCIsAndRelations 方法可更新指定 CI 和关系。

更新时会使用 CIsAndRelationsUpdates 参数的属性值。如果 CMDB 中不存在任何 CI 或关系，则会引发异常。

CreatedIDsMap 是连接客户端临时 ID 与相应的真实 CMDB ID 的 ClientIDToCmdbID 类型的映射或词典。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
CIsAndRelationsUpdates	要更新的项。有关详细信息，请参阅 “CIsAndRelationsUpdates”（第 303 页）。
ignoreValidation	如果为 True，则在更新 CMDB 之前不会执行检查。

输出

参数	注释
CreatedIDsMap	客户端 ID 到 CMDB ID 的映射。有关详细信息，请参阅 “addCIsAndRelations”（第 322 页）。

UCMDB 影响分析方法

本节提供以下方法的信息：

- “calculateImpact”（第 325 页）
- “getImpactPath”（第 326 页）
- “getImpactRulesByNamePrefix”（第 327 页）

calculateImpact

calculateImpact 方法会根据 CMDB 中定义的规则计算受指定 CI 影响的 CI。

结果会显示规则触发事件的影响。 calculateImpact 的 identifier 输出可用作 getImpactPath 的输入。

输入

参数	注释
cmdbContext	有关详细信息，请参阅“CmdbContext”（第 370 页）。
impactCategory	会触发要模拟的规则的事件类型。
IDs	ID 元素的集合。
impactRulesNames	ImpactRuleName 元素的集合。
severity	触发事件的严重度。

输出

参数	注释
impactTopology	有关详细信息，请参阅“Topology”（第 374 页）。
identifier	服务器响应的密钥。

 getImpactPath

getImpactPath 方法可检索受影响 CI 与影响该 CI 的 CI 之间路径的拓扑图形。

calculateImpact 的 identifier 输出可作为 getImpactPath 的 identifier 输入参数使用。

输入

参数	注释
cmdbContext	有关详细信息，请参阅“CmdbContext”（第 370 页）。
identifier	由 calculateImpact 返回的服务器响应的密钥。
relation	以 impactTopology 元素中由 calculateImpact 返回的 ShallowRelation 之一为基础的 Relation。

输出

参数	注释
impactPathTopology	CIs 集合和 ImpactRelations 集合。
comments	仅供内部使用。

ImpactRelations 元素由 ID、 type、 end1ID、 end2ID、 rule 和 action 组成。

 getImpactRulesByNamePrefix

getImpactRulesByNamePrefix 方法会使用前缀筛选器检索规则。

此方法适用于这样的影响规则：在命名时使用一个前缀来表示这些影响规则所应用的环境，如 SAP_myrule、 ORA_myrule 等。此方法会筛选所有采用 ruleNamePrefixFilter 参数指定的前缀开头的影响规则名称。

输入

参数	注释
cmdbContext	有关详细信息，请参阅 “CmdbContext”（第 370 页）。
ruleNamePrefixFilter	一个字符串，包含要匹配的规则名称的前几个字母。

输出

参数	注释
impactRules	impactRules 包含零个或多个 impactRule。impactRule 由 ruleName、 description、 queryName 和 isActive 组成，可指定更改效果。

数据流管理方法

本节包含一系列 Web 服务操作，以及这些操作的用途简介。有关针对每个操作的请求和响应的完整文档，请参阅数据流管理 Schema Reference。

本节包括以下主题：

管理 DFM 作业方法

- **activateJob**

  激活指定作业。

- **deactivateJob**

  停用指定作业。

- **dispatchAdHocJob**

  在 Probe 上分配作业。该作业必须处于活动状态，并且包含指定的触发器 CI。

- **getDiscoveryJobsNames**

  返回作业名称的列表。

- **isJobActive**

  检查作业是否处于活动状态。

管理触发器方法

- **addTriggerCI**

  将新触发器 CI 添加到指定作业。

➤ **addTriggerTQL**

将新触发器 TQL 添加到指定作业。

➤ **disableTriggerTQL**

防止 TQL 触发作业，但不会将其从触发作业的查询列表中永久删除。

➤ **removeTriggerCI**

从触发作业的 CI 列表中删除指定 CI。

➤ **removeTriggerTQL**

从触发作业的查询列表中删除指定 TQL。

➤ **setTriggerTQLProbesLimit**

将作业中 TQL 处于活动状态的 Probe 限制到指定列表。

域和 Probe 数据方法

➤ **getDomainType**

返回域类型。

➤ **getDomainsNames**

返回当前域的名称。

➤ **getProbeIPs**

返回指定 Probe 的 IP 地址。

➤ **getProbesNames**

返回指定域中 Probe 的名称。

➤ **getProbeScope**

返回指定 Probe 的范围定义。

➤ **isProbeConnected**

检查指定 Probe 是否已连接。

➤ **updateProbeScope**

设置指定 Probe 的范围，以覆盖现有范围。

凭据数据方法

➤ **addCredentialsEntry**

将凭据条目添加到指定域的指定协议。

➤ **getCredentialsEntriesIDs**

返回为指定协议定义的凭据的 ID。

➤ **getCredentialsEntry**

返回为指定协议定义的凭据。将返回空加密属性。

➤ **removeCredentialsEntry**

从协议删除指定证书。

➤ **updateCredentialsEntry**

设置指定凭据条目的属性的新值。

数据刷新方法

➤ **rediscoverCIs**

查找搜寻到指定 CI 对象的触发器并重新运行这些触发器。（请注意，重新运行命令的优先级比其他计划项的优先级更高。）

rediscoverCIs 以异步方式运行。调用 **checkDiscoveryProgress** 可以确定重新搜寻完成的时间。

➤ **checkDiscoveryProgress**

返回对指定 ID 最近一次调用 **rediscoverCIs** 的进度。响应为一个 0 到 1 之间的值。当响应为 1 时，说明 **rediscoverCIs** 调用已完成。

➤ **rediscoverViewCIs**

查找已创建用于填充指定视图的数据的触发器，并重新运行这些触发器。（请注意，重新运行命令的优先级比其他计划项的优先级更高。）

rediscoverViewCIs 以异步方式运行。调用 **checkViewDiscoveryProgress** 可以确定重新搜寻完成的时间。

- **checkViewDiscoveryProgress**

  返回对指定视图最近一次调用 **rediscoverViewCIs** 的进度。响应为一个 0-1 之间的值。当响应为 1 时，说明 **rediscoverCIs** 调用已经完成。

用例

以下用例假定两个系统为：

- HP Universal CMDB 服务器
- 包含配置项库的第三方系统

本节包括以下主题：

- “填充 CMDB”（第 331 页）
- “查询 CMDB”（第 332 页）
- “查询类模型”（第 332 页）
- “分析更改影响”（第 332 页）

填充 CMDB

用例：

- 第三方资产管理仅使用可在资产管理中使用的信息更新 CMDB
- 很多第三方系统通过填充 CMDB 来创建可跟踪变更并执行影响分析的中心 CMDB
- 第三方系统按照第三方业务逻辑创建配置项和关系，以利用 CMDB 查询功能

查询 CMDB

用例：

- 第三方系统通过了解 SAP TQL 的结果，来获取表示 SAP 系统的配置项和关系
- 第三方系统获取过去五个小时内添加或更改的 Oracle 服务器的列表
- 第三方系统获取主机名包含子字符串 lab 的服务器的列表
- 第三方系统通过获取指定 CI 的相邻项，来查找与给定 CI 相关的元素

查询类模型

用例：

- 用户通过第三方系统指定要从 CMDB 中检索的数据集合。通过类模型可以生成用户界面，从而向用户显示可能的属性，并提示用户输入所需的数据。然后，用户可以选择要检索的信息。
- 当用户无法访问 UCMDB 用户界面时，第三方系统将搜索类模型。

分析更改影响

用例：

第三方系统输出一个受指定主机的变更影响的业务服务列表。

示例

本节包括以下主题：

- “基类示例”（第 334 页）
- “查询示例”（第 337 页）

基类示例

```
package com.hp.ucmdb.demo;

import com.hp.ucmdb.generated.services.UcmdbService;
import com.hp.ucmdb.generated.services.UcmdbServiceStub;
import com.hp.ucmdb.generated.types.CmdbContext;
import org.apache.axis2.AxisFault;
import org.apache.axis2.transport.http.HTTPConstants;
```

```
import org.apache.axis2.transport.http.HttpTransportProperties;

import java.net.MalformedURLException;
import java.net.URL;
```

```
/**
* User: hbarkai
* Date: Jul 12, 2007
*/
abstract class Demo {
```

```
UcmdbService stub;
CmdbContext context;
```

```
public void initDemo() {
    try {
        setStub(createUcmdbService("admin", "admin"));
        setContext();
    } catch (Exception e) {
        //handle exception
    }
}
```

```
public UcmdbService getStub() {
    return stub;
}
```

```
public void setStub(UcmdbService stub) {
    this.stub = stub;
}
```

```
public CmdbContext getContext() {
    return context;
}
```

```
public void setContext() {
    CmdbContext context = new CmdbContext();
    context.setCallerApplication("demo");
    this.context = context;
}
```

```
//connection to service  - for axis2/jibx client

private static final String PROTOCOL = "http";
private static final String HOST_NAME = "host_name";
private static final int PORT = 8080;
private static final String FILE = "/axis2/services/UcmdbService";

protected UcmdbService createUcmdbService
    (String username, String password) throws Exception{
        URL url;
        UcmdbServiceStub serviceStub;
```

```
        try {
            url = new URL
                (Demo.PROTOCOL, Demo.HOST_NAME,
                Demo.PORT, Demo.FILE);
            serviceStub = new UcmdbServiceStub(url.toString());
            HttpTransportProperties.Authenticator auth =
                new HttpTransportProperties.Authenticator();
            auth.setUsername(username);
            auth.setPassword(password);
serviceStub._getServiceClient().getOptions().setProperty
    (HTTPConstants.AUTHENTICATE,auth);
```

```
    } catch (AxisFault axisFault) {
            throw new Exception
            ("Failed to create SOAP adapter for "
                + Demo.HOST_NAME , axisFault);
```

```
        } catch (MalformedURLException e) {

            throw new Exception
                ("Failed to create SOAP adapter for "
                    + Demo.HOST_NAME, e);
        }
        return serviceStub;
 }
}
```

查询示例

```
package com.hp.ucmdb.demo;

import com.hp.ucmdb.generated.params.query.*;
import com.hp.ucmdb.generated.services.UcmdbFaultException;
import com.hp.ucmdb.generated.services.UcmdbService;
import com.hp.ucmdb.generated.types.*;
import com.hp.ucmdb.generated.types.props.*;

import java.rmi.RemoteException;
```

```
public class QueryDemo extends Demo{

    UcmdbService stub;
    CmdbContext context;
```

```
    public void getCIsByTypeDemo() {
        GetCIsByType request = new GetCIsByType();
        //set cmdbcontext
        CmdbContext cmdbContext = getContext();
        request.setCmdbContext(cmdbContext);
        //set CIs type
        request.setType("anyType");
        //set CIs propeties to be retrieved
        CustomProperties customProperties = new CustomProperties();
        PredefinedProperties predefinedProperties =
            new PredefinedProperties();
        SimplePredefinedProperty simplePredefinedProperty =
            new SimplePredefinedProperty();
        simplePredefinedProperty.setName
            (SimplePredefinedProperty.nameEnum.DERIVED);
        SimplePredefinedPropertyCollection
            simplePredefinedPropertyCollection =
                new SimplePredefinedPropertyCollection();
```

```
        simplePredefinedPropertyCollection.addSimplePredefinedProperty
            (simplePredefinedProperty);
        predefinedProperties.setSimplePredefinedProperties
            (simplePredefinedPropertyCollection);
        customProperties.setPredefinedProperties(predefinedProperties);
        request.setProperties(customProperties);
        try {
            GetCIsByTypeResponse response =
                getStub().getCIsByType(request);
            TopologyMap map =
                getTopologyMapResultFromCIs
                    (response.getCIs(), response.getChunkInfo());
        } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }
    }
```

```
    public void getCIsByIdDemo() {
        GetCIsById request = new GetCIsById();
        CmdbContext cmdbContext = getContext();
        //set cmdbcontext
        request.setCmdbContext(cmdbContext);
        //set ids
        ID id1 = new ID();
        id1.setBase("cmdbobjectidCIT1");
        ID id2 = new ID();
        id2.setBase("cmdbobjectidCIT2");
        IDs ids = new IDs();
        ids.addID(id1);
        ids.addID(id2);
        request.setIDs(ids);
        //set CIs properties to be retrieved
        TypedPropertiesCollection properties =
            new TypedPropertiesCollection();
```

```
        TypedProperties typedProperties1 =
            new TypedProperties();
        typedProperties1.setType("CIT1");
```

```
        CustomTypedProperties customProperties1 =
            new CustomTypedProperties();
        PredefinedTypedProperties predefinedProperties1 =
            new PredefinedTypedProperties();
        SimpleTypedPredefinedProperty simplePredefinedProperty1 =
            new SimpleTypedPredefinedProperty();
        simplePredefinedProperty1.setName
            (SimpleTypedPredefinedProperty.nameEnum.CONCRETE);
        SimpleTypedPredefinedPropertyCollection
            simplePredefinedPropertyCollection1 =
                new SimpleTypedPredefinedPropertyCollection();
        simplePredefinedPropertyCollection1
            .addSimpleTypedPredefinedProperty
                (simplePredefinedProperty1);
```

```
predefinedProperties1.
        setSimpleTypedPredefinedProperties
            (simplePredefinedPropertyCollection1);
        customProperties1.
            setPredefinedTypedProperties
                (predefinedProperties1);
        typedProperties1.setProperties(customProperties1);
        properties.addTypedProperties(typedProperties1);
```

```
        TypedProperties typedProperties2 =
            new TypedProperties();
        typedProperties2.setType("CIT2");
        CustomTypedProperties customProperties2 =
            new CustomTypedProperties();
        PredefinedTypedProperties predefinedProperties2 =
            new PredefinedTypedProperties();
        SimpleTypedPredefinedProperty simplePredefinedProperty2 =
            new SimpleTypedPredefinedProperty();
        simplePredefinedProperty2.setName
            (SimpleTypedPredefinedProperty.nameEnum.NAMING);
        SimpleTypedPredefinedPropertyCollection
            simplePredefinedPropertyCollection2 =
                new SimpleTypedPredefinedPropertyCollection();
```

```
        simplePredefinedPropertyCollection2.
            addSimpleTypedPredefinedProperty
                (simplePredefinedProperty2);
```

```
        predefinedProperties2.setSimpleTypedPredefinedProperties
            (simplePredefinedPropertyCollection2);
        customProperties2.setPredefinedTypedProperties
            (predefinedProperties2);
        typedProperties2.setProperties(customProperties2);
        properties.addTypedProperties(typedProperties2);
```

```
    request.setCIsTypedProperties(properties);
        try {
            GetCIsByIdResponse response =
                getStub().getCIsById(request);
            CIs cis = response.getCIs();
        } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }

    }
```

```
    public void getFilteredCIsByTypeDemo() {
        GetFilteredCIsByType request = new GetFilteredCIsByType();
        CmdbContext cmdbContext = getContext();
        //set cmdbcontext
        request.setCmdbContext(cmdbContext);
        //set CIs type
        request.setType("anyType");
        //sets Filter conditions
        Conditions conditions = new Conditions();
        IntConditions intConditions = new IntConditions();
        IntCondition intCondition = new IntCondition();
        IntProp intProp = new IntProp();
        intProp.setName("int_attr1");
```

```
intProp.setValue(100);
intCondition.setCondition(intProp);
intCondition.setIntOperator
    (IntCondition.intOperatorEnum.Greater);
intConditions.addIntCondition(intCondition);
```

```
conditions.setIntConditions(intConditions);
request.setConditions(conditions);
//set logical operator for conditions
request.setConditionsLogicalOperator
    (GetFilteredCIsByType.conditionsLogicalOperatorEnum.AND);
//set CIs properties to be retrieved
CustomProperties customProperties =
    new CustomProperties();
PredefinedProperties predefinedProperties =
    new PredefinedProperties();
SimplePredefinedProperty simplePredefinedProperty =
    new SimplePredefinedProperty();
simplePredefinedProperty.setName
    (SimplePredefinedProperty.nameEnum.NAMING);
```

```
SimplePredefinedPropertyCollection
    simplePredefinedPropertyCollection =
        new SimplePredefinedPropertyCollection();
simplePredefinedPropertyCollection.
    addSimplePredefinedProperty
        (simplePredefinedProperty);
predefinedProperties.setSimplePredefinedProperties
    (simplePredefinedPropertyCollection);
customProperties.setPredefinedProperties
    (predefinedProperties);
```

```
request.setProperties(customProperties);
try {
    GetFilteredCIsByTypeResponse response =
        getStub().getFilteredCIsByType(request);
    TopologyMap map =
        getTopologyMapResultFromCIs
            (response.getCIs(), response.getChunkInfo());
```

```
        } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }

    }
```

```
    public void executeTopologyQueryByNameDemo() {
        ExecuteTopologyQueryByName request = new
ExecuteTopologyQueryByName();
        CmdbContext cmdbContext = getContext();
        //set cmdbcontext
        request.setCmdbContext(cmdbContext);
        //set query name
        request.setQueryName("queryName");
```

```
        try {
            ExecuteTopologyQueryByNameResponse response =
                getStub().executeTopologyQueryByName(request);
            TopologyMap map =
                getTopologyMapResult
                    (response.getTopologyMap(), response.getChunkInfo());
        } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }

    }
```

```
// assume the follow query was defined at UCMDB
// Query Name: exampleQuery
// Query sketch:
//                                  Host
//                                  / \
//                                 ip  Disk
// Query Parameters:
//     Host-
//          host_os (like)
//     Disk-
//          disk_failures (equal)
```

```
public void executeTopologyQueryByNameWithParametersDemo() {
    ExecuteTopologyQueryByNameWithParameters request =
        new ExecuteTopologyQueryByNameWithParameters();
    CmdbContext cmdbContext = getContext();
    //set cmdbcontext
    request.setCmdbContext(cmdbContext);
    //set query name
    request.setQueryName("queryName");
    //set parameters
    ParameterizedNode hostParametrizedNode =
        new ParameterizedNode();
    hostParametrizedNode.setNodeLabel("Host");
    CIProperties parameters = new CIProperties();
    StrProps strProps = new StrProps();
    StrProp strProp = new StrProp();
    strProp.setName("host_os");
    strProp.setValue("%2000%");
    strProps.addStrProp(strProp);
    parameters.setStrProps(strProps);
    hostParametrizedNode.setParameters(parameters);
    request.addParameterizedNodes(hostParametrizedNode);
    ParameterizedNode diskParametrizedNode =
        new ParameterizedNode();
```

```
    diskParametrizedNode.setNodeLabel("Disk");
    CIProperties parameters1 = new CIProperties();
    IntProps intProps = new IntProps();
```

```
        IntProp intProp = new IntProp();
            intProp.setName("disk_failures");
            intProp.setValue(30);
            intProps.addIntProp(intProp);
            parameters1.setIntProps(intProps);
            diskParametrizedNode.setParameters(parameters1);
```

```
            request.addParameterizedNodes(diskParametrizedNode);
            try {
                ExecuteTopologyQueryByNameWithParametersResponse
                    response =
                    getStub().executeTopologyQueryByNameWithParameters
                        (request);
                TopologyMap map =
                    getTopologyMapResult
                        (response.getTopologyMap(), response.getChunkInfo());
            } catch (RemoteException e) {
                //handle exception
            } catch (UcmdbFaultException e) {
                //handle exception
            }

        }
```

```
/       // assume the follow query was defined at UCMDB
        // Query Name: exampleQuery
        // Query sketch:
        //                              Host
        //                              /  \
        //                            ip  Disk
        // Query Parameters:
        //     Host-
        //         host_os (like)
        //     Disk-
        //         disk_failures (equal)
```

```
public void executeTopologyQueryWithParametersDemo() {
    ExecuteTopologyQueryWithParameters request =
        new ExecuteTopologyQueryWithParameters();
    CmdbContext cmdbContext = getContext();
    //set cmdbcontext
    request.setCmdbContext(cmdbContext);
    //set query definition
    String queryXml = "<xml that represents the query above>";
    request.setQueryXml(queryXml);
    //set parameters
    ParameterizedNode hostParametrizedNode =
        new ParameterizedNode();
```

```
hostParametrizedNode.setNodeLabel("Host");
    CIProperties parameters = new CIProperties();
    StrProps strProps = new StrProps();
    StrProp strProp = new StrProp();
    strProp.setName("host_os");
    strProp.setValue("%2000%");
    strProps.addStrProp(strProp);
    parameters.setStrProps(strProps);
    hostParametrizedNode.setParameters(parameters);
    request.addParameterizedNodes(hostParametrizedNode);
    ParameterizedNode diskParametrizedNode =
        new ParameterizedNode();
    diskParametrizedNode.setNodeLabel("Disk");
    CIProperties parameters1 = new CIProperties();
    IntProps intProps = new IntProps();
    IntProp intProp = new IntProp();
    intProp.setName("disk_failures");
    intProp.setValue(30);
    intProps.addIntProp(intProp);
    parameters1.setIntProps(intProps);
    diskParametrizedNode.setParameters(parameters1);
    request.addParameterizedNodes(diskParametrizedNode);
```

```
        try {
            ExecuteTopologyQueryWithParametersResponse
            response = getStub().executeTopologyQueryWithParameters
                (request);
            TopologyMap map =
                getTopologyMapResult
                    (response.getTopologyMap(), response.getChunkInfo());
```

```
         } catch (RemoteException e) {
              //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }

    }
```

```
    public void getCINeighboursDemo() {
        GetCINeighbours request = new GetCINeighbours();
        //set cmdbcontext
        CmdbContext cmdbContext = getContext();
        request.setCmdbContext(cmdbContext);
        // set CI id
        ID id = new ID();
        id.setBase("cmdbobjectidCIT1");
        request.setID(id);
        //set neighbour type
        request.setNeighbourType("neighbourType");
        //set Neighbours CIs propeties to be retrieved
        TypedPropertiesCollection properties =
            new TypedPropertiesCollection();
        TypedProperties typedProperties1 = new TypedProperties();
        typedProperties1.setType("neighbourType");
        CustomTypedProperties customProperties1 =
            new CustomTypedProperties();
        PredefinedTypedProperties predefinedProperties1 =
            new PredefinedTypedProperties();
```

```
QualifierProperties qualifierProperties =
    new QualifierProperties();
qualifierProperties.addQualifierName("ID_ATTRIBUTE");
predefinedProperties1.setQualifierProperties(qualifierProperties);
customProperties1.setPredefinedTypedProperties
    (predefinedProperties1);
typedProperties1.setProperties(customProperties1);
properties.addTypedProperties(typedProperties1);
request.setCIProperties(properties);
```

```
TypedPropertiesCollection relationsProperties =
    new TypedPropertiesCollection();
TypedProperties typedProperties2 = new TypedProperties();
typedProperties2.setType("relationType");
CustomTypedProperties customProperties2 =
    new CustomTypedProperties();
```

```
PredefinedTypedProperties predefinedProperties2 =
    new PredefinedTypedProperties();
SimpleTypedPredefinedProperty simplePredefinedProperty2 =
    new SimpleTypedPredefinedProperty();
simplePredefinedProperty2.setName
```

```
    (SimpleTypedPredefinedProperty.nameEnum.CONCRETE);
SimpleTypedPredefinedPropertyCollection
    simplePredefinedPropertyCollection2 =
        new SimpleTypedPredefinedPropertyCollection();
simplePredefinedPropertyCollection2.
    addSimpleTypedPredefinedProperty
        (simplePredefinedProperty2);
predefinedProperties2.
    setSimpleTypedPredefinedProperties
        (simplePredefinedPropertyCollection2);
customProperties2.setPredefinedTypedProperties
    (predefinedProperties2);
typedProperties2.setProperties(customProperties2);
relationsProperties.addTypedProperties(typedProperties2);
request.setRelationProperties(relationsProperties);
```

```
        try {
            GetCINeighboursResponse response =
                getStub().getCINeighbours(request);
            Topology topology = response.getTopology();
      } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
             //handle exception
        }

    }
```

```
//get Topology Map for chunked/non-chunked result

    private TopologyMap getTopologyMapResult(TopologyMap topologyMap, ChunkInfo
chunkInfo) {
        if(chunkInfo.getNumberOfChunks() == 0) {
            return topologyMap;
        } else {
```

```
            topologyMap = new TopologyMap();
            for(int i=1 ; i <= chunkInfo.getNumberOfChunks() ; i++) {
                ChunkRequest chunkRequest = new ChunkRequest();
                chunkRequest.setChunkInfo(chunkInfo);
                chunkRequest.setChunkNumber(i);
                PullTopologyMapChunks req =
                    new PullTopologyMapChunks();
                req.setChunkRequest(chunkRequest);
                req.setCmdbContext(getContext());
                PullTopologyMapChunksResponse res = null;
```

```
                try  {
                    res = getStub().pullTopologyMapChunks(req);
                    TopologyMap map = res.getTopologyMap();
                    topologyMap = mergeMaps(topologyMap, map);
                } catch (RemoteException e) {
                    //handle exception
                } catch (UcmdbFaultException e) {
                    //handle exception
                }
            }
        }
        return topologyMap;

    }
```

```
    private TopologyMap getTopologyMapResultFromCIs(CIs cis, ChunkInfo chunkInfo)
{
        TopologyMap topologyMap = new TopologyMap();
        if(chunkInfo.getNumberOfChunks() == 0) {
            CINode ciNode = new CINode();
            ciNode.setLabel("");
            ciNode.setCIs(cis);
            CINodes ciNodes = new CINodes();
            ciNodes.addCINode(ciNode);
            topologyMap.setCINodes(ciNodes);
        } else {
```

```
            for(int i=1 ; i <= chunkInfo.getNumberOfChunks() ; i++) {
                ChunkRequest chunkRequest =
                    new ChunkRequest();
                chunkRequest.setChunkInfo(chunkInfo);
                chunkRequest.setChunkNumber(i);
                PullTopologyMapChunks req =
                    new PullTopologyMapChunks();
                req.setChunkRequest(chunkRequest);
                req.setCmdbContext(getContext());
                PullTopologyMapChunksResponse res = null;
```

```
                try  {
                    res = getStub().pullTopologyMapChunks(req);
                } catch (RemoteException e) {
                    //handle exception
                } catch (UcmdbFaultException e) {
                    //handle exception
                }
                TopologyMap map = res.getTopologyMap();
                topologyMap = mergeMaps(topologyMap, map);
            }
```

```
            //release chunks
            ReleaseChunks req = new ReleaseChunks();
            req.setChunksKey(chunkInfo.getChunksKey());
            req.setCmdbContext(getContext());
```

```
            try {
                getStub().releaseChunks(req);
            } catch (RemoteException e) {
                //handle exception
            } catch (UcmdbFaultException e) {
                //handle exception
            }
        }
        return topologyMap;

    }
```

```
//=================================================
/*  WARNING merge will be correct only if a each node is given
    a unique name. This applies to both CI and Relation nodes .*/
//=================================================
    private TopologyMap mergeMaps(TopologyMap topologyMap, TopologyMap
newMap) {
        for(int i=0 ; i < newMap.getCINodes().sizeCINodeList() ; i++ ) {
            CINode ciNode = newMap.getCINodes().getCINode(i);
            boolean alreadyExist = false;
            if(topologyMap.getCINodes() == null) {
                topologyMap.setCINodes(new CINodes());
            }
```

```
        for(int j=0 ; j < topologyMap.getCINodes().sizeCINodeList() ; j++) {
            CINode ciNode2 = topologyMap.getCINodes().getCINode(j);
            if(ciNode2.getLabel().equals(ciNode.getLabel())){
```

```
                CIs cisTOAdd = ciNode.getCIs();
                CIs cis =
                    mergeCIsGroups
                    (topologyMap.getCINodes().getCINode(j).getCIs(),
                        cisTOAdd);
                topologyMap.getCINodes().getCINode(j).setCIs(cis);
                alreadyExist = true;
            }
        }
        if(!alreadyExist) {
            topologyMap.getCINodes().addCINode(ciNode);
        }

    }
```

```
    for(int i=0 ; i < newMap.getRelationNodes().sizeRelationNodeList() ; i++ ) {
        RelationNode relationNode =
            newMap.getRelationNodes().getRelationNode(i);
        boolean alreadyExist = false;
        if(topologyMap.getRelationNodes() == null) {
            topologyMap.setRelationNodes(new RelationNodes());
        }
```

```
            for(int j=0 ;
                    j < topologyMap.getRelationNodes().sizeRelationNodeList() ;
                    j++) {
                RelationNode relationNode2 =
                    topologyMap.getRelationNodes().getRelationNode(j);
                if(relationNode2.getLabel().equals(relationNode.getLabel())){
                    Relations relationsTOAdd = relationNode.getRelations();
                    Relations relations =
                        mergeRelationsGroups
                        (topologyMap.getRelationNodes().
                            getRelationNode(j).getRelations(),
                            relationsTOAdd);
                    topologyMap.getRelationNodes().
                        getRelationNode(j).setRelations(relations);
                    alreadyExist = true;
                }
            }
```

```
            if(!alreadyExist) {
                topologyMap.getRelationNodes().addRelationNode(relationNode);
            }
        }

        return topologyMap;

    }
```

```
    private Relations mergeRelationsGroups(Relations relations1, Relations relations2)
{
        for(int i=0 ; i < relations2.sizeRelationList() ; i++) {
            relations1.addRelation(relations2.getRelation(i));
        }
        return relations2;
    }
```

```
    private CIs mergeCIsGroups(CIs cis1, CIs cis2) {
        for(int i=0 ; i < cis2.sizeCIList() ; i++) {
            cis1.addCI(cis2.getCI(i));
        }
        return cis1;
    }


}
```

更新示例

```
package com.hp.ucmdb.demo;

import com.hp.ucmdb.generated.params.update.AddCIsAndRelations;
import com.hp.ucmdb.generated.params.update.AddCIsAndRelationsResponse;
import com.hp.ucmdb.generated.params.update.UpdateCIsAndRelations;
import com.hp.ucmdb.generated.params.update.DeleteCIsAndRelations;
import com.hp.ucmdb.generated.services.UcmdbFaultException;
import com.hp.ucmdb.generated.types.*;
import com.hp.ucmdb.generated.types.update.CIsAndRelationsUpdates;
import com.hp.ucmdb.generated.types.update.ClientIDToCmdbID;

import java.rmi.RemoteException;

public class UpdateDemo extends Demo{
```

```
    public void getAddCIsAndRelationsDemo() {
        AddCIsAndRelations request = new AddCIsAndRelations();
        request.setCmdbContext(getContext());
        request.setUpdateExisting(true);
        CIsAndRelationsUpdates updates = new CIsAndRelationsUpdates();
        CIs cis = new CIs();
        CI ci = new CI();
        ID id = new ID();
        id.setBase("temp1");
        id.setTemp(true);
```

```
        ci.setID(id);
        ci.setType("host");
```

```
        CIProperties props = new CIProperties();
        StrProps strProps = new StrProps();
        StrProp strProp = new StrProp();
        strProp.setName("host_key");
        String value = "blabla";
        strProp.setValue(value);
```

```
        strProps.addStrProp(strProp);
        props.setStrProps(strProps);
        ci.setProps(props);
        cis.addCI(ci);
        updates.setCIsForUpdate(cis);
        request.setCIsAndRelationsUpdates(updates);
```

```
        try {
            AddCIsAndRelationsResponse response  =
                getStub().addCIsAndRelations(request);
            for(int i = 0  ;  i < response.sizeCreatedIDsMapList() ; i++) {
                ClientIDToCmdbID idsMap = response.getCreatedIDsMap(i);
                //do something
            }
        } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }

    }
```

```
    public void getUpdateCIsAndRelationsDemo() {
        UpdateCIsAndRelations request = new UpdateCIsAndRelations();
        request.setCmdbContext(getContext());
```

```
        CIsAndRelationsUpdates updates =
            new CIsAndRelationsUpdates();
        CIs cis = new CIs();
        CI ci = new CI();
        ID id = new ID();
```

```
        id.setBase("temp1");
        id.setTemp(true);
        ci.setID(id);
        ci.setType("host");
        CIProperties props = new CIProperties();
        StrProps strProps = new StrProps();
```

```
        StrProp hostKeyProp = new StrProp();
        hostKeyProp.setName("host_key");
        String hostKeyValue = "blabla";
        hostKeyProp.setValue(hostKeyValue);
        strProps.addStrProp(hostKeyProp);
```

```
        StrProp hostOSProp = new StrProp();
        hostOSProp.setName("host_os");
        String hostOSValue = "winXP";
        hostOSProp.setValue(hostOSValue);
        strProps.addStrProp(hostOSProp);
```

```
        StrProp hostDNSProp = new StrProp();
        hostDNSProp.setName("host_dnsname");
        String hostDNSValue = "dnsname";
        hostDNSProp.setValue(hostDNSValue);
        strProps.addStrProp(hostDNSProp);
```

```
        props.setStrProps(strProps);
        ci.setProps(props);
        cis.addCI(ci);
        updates.setCIsForUpdate(cis);
        request.setCIsAndRelationsUpdates(updates);
```

```
        try {
            getStub().updateCIsAndRelations(request);
        } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }

    }
```

```
    public void getDeleteCIsAndRelationsDemo() {
        DeleteCIsAndRelations request =
            new DeleteCIsAndRelations();
        request.setCmdbContext(getContext());
        CIsAndRelationsUpdates updates =
            new CIsAndRelationsUpdates();
        CIs cis = new CIs();
        CI ci = new CI();
        ID id = new ID();
        id.setBase("stam");
        id.setTemp(true);
        ci.setID(id);
        ci.setType("host");
```

```
        CIProperties props = new CIProperties();
        StrProps strProps = new StrProps();
        StrProp strProp1 = new StrProp();
        strProp1.setName("host_key");
        String value1 = "for_delete";
        strProp1.setValue(value1);
        strProps.addStrProp(strProp1);
        props.setStrProps(strProps);
        ci.setProps(props);
        cis.addCI(ci);
        updates.setCIsForUpdate(cis);
        request.setCIsAndRelationsUpdates(updates);
```

```
        try {
            getStub().deleteCIsAndRelations(request);
        } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }

    }


}
```

类模型示例

```
package com.hp.ucmdb.demo;

import com.hp.ucmdb.generated.params.classmodel.*;
import com.hp.ucmdb.generated.services.UcmdbFaultException;
import com.hp.ucmdb.generated.types.classmodel.UcmdbClassModelHierarchy;
import com.hp.ucmdb.generated.types.classmodel.UcmdbClass;

import java.rmi.RemoteException;

public class ClassmodelDemo extends Demo{
```

```
    public void getClassAncestorsDemo() {
        GetClassAncestors request =
            new GetClassAncestors();
        request.setCmdbContext(getContext());
        request.setClassName("className");
```

```
        try {
            GetClassAncestorsResponse response  =
                getStub().getClassAncestors(request);
            UcmdbClassModelHierarchy hierarchy =
                response.getClassHierarchy();
        } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }

    }
```

```
    public void getAllClassesHierarchyDemo() {
        GetAllClassesHierarchy request =
            new GetAllClassesHierarchy();
        request.setCmdbContext(getContext());
        try {
            GetAllClassesHierarchyResponse response  =
                getStub().getAllClassesHierarchy(request);
            UcmdbClassModelHierarchy hierarchy =
                response.getClassesHierarchy();
        } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }

    }
```

```
    public void getCmdbClassDefinitionDemo() {
        GetCmdbClassDefinition request =
            new GetCmdbClassDefinition();
        request.setCmdbContext(getContext());
        request.setClassName("className");
```

```
        try {
            GetCmdbClassDefinitionResponse response  =
            getStub().getCmdbClassDefinition(request);
            UcmdbClass ucmdbClass = response.getUcmdbClass();
        } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }

    }


}
```

影响分析示例

```
package com.hp.ucmdb.demo;

import com.hp.ucmdb.generated.params.impact.*;
import com.hp.ucmdb.generated.services.UcmdbFaultException;
import com.hp.ucmdb.generated.types.*;
import com.hp.ucmdb.generated.types.impact.*;

import java.rmi.RemoteException;
```

```
/**
 * Date: Jul 17, 2007
 */
public class ImpactDemo extends Demo{

//Impact Rule Name : impactExample
//Impact Query:
//              Network
//                 |
//               Host
//                 |
//                IP
//Impact Action: network affect on ip ;severity 100% ; category: change
//
public void calculateImpactAndGetImpactPathDemo() {
    CalculateImpact request = new CalculateImpact();
    request.setCmdbContext(getContext());
    //set root cause ids
    IDs ids = new IDs();
    ID id = new ID();
    id.setBase("rootCauseCmdbID");
    ids.addID(id);
```

```
request.setIDs(ids);
//set impact category
request.setImpactCategory("change");
//set rule Names
ImpactRuleNames impactRuleNames = new ImpactRuleNames();
ImpactRuleName impactRuleName = new ImpactRuleName();
impactRuleName.setBase("impactExample");
impactRuleNames.addImpactRuleName(impactRuleName);
request.setImpactRuleNames(impactRuleNames);
//set severity
request.setSeverity(100);
CalculateImpactResponse response =
    new CalculateImpactResponse();
```

```
request.setIDs(ids);
//set impact category
request.setImpactCategory("change");
//set rule Names
ImpactRuleNames impactRuleNames = new ImpactRuleNames();
ImpactRuleName impactRuleName = new ImpactRuleName();
impactRuleName.setBase("impactExample");
impactRuleNames.addImpactRuleName(impactRuleName);
request.setImpactRuleNames(impactRuleNames);
//set severity
request.setSeverity(100);
CalculateImpactResponse response =
    new CalculateImpactResponse();
```

```
try {
    response  = getStub().calculateImpact(request);
} catch (RemoteException e) {
    //handle exception
```

```
    } catch (UcmdbFaultException e) {
        //handle exception
    }
    Identifier identifier= response.getIdentifier();
    Topology topology = response.getImpactTopology();
    Relation relation = topology.getRelations().getRelation(0);
    GetImpactPath request2 = new GetImpactPath();
    //set cmdb context
    request2.setCmdbContext(getContext());
    //set impact identifier
    request2.setIdentifier(identifier);
    //set shallowRelation
    ShallowRelation shallowRelation = new ShallowRelation();
    shallowRelation.setID(relation.getID());
    shallowRelation.setEnd1ID(relation.getEnd1ID());
    shallowRelation.setEnd2ID(relation.getEnd2ID());
    shallowRelation.setType(relation.getType());
    request2.setRelation(shallowRelation);
```

```
    try {
        GetImpactPathResponse response2 =
            getStub().getImpactPath(request2);
        ImpactTopology impactTopology =
            response2.getImpactPathTopology();
    } catch (RemoteException e) {
        //To change body of catch statement
        // use File | Settings | File Templates.
        e.printStackTrace();
    } catch (UcmdbFaultException e) {
        //To change body of catch statement
        // use File | Settings | File Templates.
        e.printStackTrace();
    }
}
```

```
public void getImpactRulesByGroupName() {
    GetImpactRulesByGroupName request =
        new GetImpactRulesByGroupName();
    //set cmdb context
    request.setCmdbContext(getContext());
    //set group names list
    request.addRuleGroupNameFilter("groupName1");
    request.addRuleGroupNameFilter("groupName2");
```

```
    try {
        GetImpactRulesByGroupNameResponse response  =
            getStub().getImpactRulesByGroupName(request);
        ImpactRules impactRules = response.getImpactRules();
    } catch (RemoteException e) {
        //handle exception
    } catch (UcmdbFaultException e) {
        //handle exception
    }

}
```

```
public void getImpactRulesByNamePrefix() {
        GetImpactRulesByNamePrefix request =
            new GetImpactRulesByNamePrefix();
        //set cmdb context
        request.setCmdbContext(getContext());
        //set prefixes list
        request.addRuleNamePrefixFilter("prefix1");
```

```
        try {
            GetImpactRulesByNamePrefixResponse response  =
                getStub().getImpactRulesByNamePrefix(request);
            ImpactRules impactRules = response.getImpactRules();
        } catch (RemoteException e) {
            //handle exception
        } catch (UcmdbFaultException e) {
            //handle exception
        }
}

}
```

添加凭据示例

```
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.AUTHENTIC
ATE,auth);

            discoveryService = serviceStub;
        } catch (Exception e) {
            throw new Exception("cannot create a connection to service ", e);
        }

        return discoveryService;

    }
} // End class
```

UCMDB 常规参数

本节介绍了服务方法最常用的参数。有关详细信息，请参阅架构文档。

本节包括以下主题：

- “CmdbContext”（第 370 页）
- “ID”（第 370 页）
- “密钥属性”（第 370 页）
- “ID 类型”（第 370 页）
- “CIProperties”（第 371 页）

➤ “类型名称”（第 372 页）
➤ “配置项 (CI)”（第 372 页）
➤ “Relation”（第 372 页）

CmdbContext

所有的 UCMDB Web 服务 API 服务调用都需要一个 CmdbContext 参数。CmdbContext 是一个 callerApplication 字符串，可识别调用该服务的应用程序。CmdbContext 用于进行日志记录和疑难解答。

ID

每个 CI 和 Relation 都包含一个 ID 字段。该字段由一个区分大小写的 ID 字符串和一个可选 temp 标记组成，后者用来表示此 ID 是否为临时 ID。

密钥属性

在某些上下文中，密钥属性可用于代替 CMDB ID 标识 CI 或 Relation。密钥属性是指在类定义中设置了 ID_ATTRIBUTE 的属性。

在用户界面的 “配置项类型”列表中，密钥属性旁边会显示一个密钥图标。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的 “‘添加 / 编辑属性’对话框”。有关从 API 客户端应用程序中识别密钥属性的信息，请参阅 “getCmdbClassDefinition”（第 306 页）。

ID 类型

一个 ID 元素可以包括一个真实 ID、一个临时 ID，或者为空。

真实 ID 是 CMDB 所分配的用于标识数据库中实体的字符串；临时 ID 可以是当前请求中的任意唯一字符串；空 ID 表示没有分配任何值。

临时 ID 可按客户端指定，通常表示该客户端存储的 CI 的 ID。但是，它不一定表示 CMDB 中已经创建的实体。如果 CMDB 能够使用 CI 密钥属性标识现有的数据配置项，则在客户端传递某个临时 ID 后，即使某 CI 使用真实 ID 进行了标识，该 CI 仍适用于上下文。

CI 的真实 ID 由 CMDB 根据该 CI 的类型和密钥属性计算得到； Relation 的 ID 则基于该关系的类型、该关系所包含的两个 CI 的 ID，以及该关系的密钥属性。因此，必须在创建 CI 或 Relation 期间设置密钥属性值。如果创建 CI 时未指定密钥属性值，则会出现以下两种可能情况：

- 如果 CIT 包含一个 RANDOM_GENERATED_ID 限定符，则服务器将生成一个唯一的 ID。
- 如果 CIT 不包含 RANDOM_GENERATED_ID 限定符，则会引发异常。

有关详细信息，请参阅 《HP Universal CMDB 建模指南》中的 “CI 类型管理器”。

CIProperties

CIProperties 元素由多个集合组成，每个集合均包含一系列的名称值元素，而这些元素可以指定该集合名称所表示类型的属性。因为这些集合并不是必需的，所以 CIProperties 元素可以包含集合的任意组合。

CIProperties 由 CI 和 Relation 元素使用。有关详细信息，请参阅 “配置项 (CI)”（第 372 页）和 “Relation”（第 372 页）。

属性集合包括：

- dateProps - DateProp 元素的集合
- doubleProps - DoubleProp 元素的集合
- floatProps - FloatProp 元素的集合
- intListProps - intListProp 元素的集合
- intProps - IntProp 元素的集合
- strProps - StrProp 元素的集合

- strListProps - StrListProp 元素的集合
- longProps - LongProp 元素的集合
- bytesProps - BytesProp 元素的集合
- xmlProps - XmlProp 元素的集合

类型名称

类型名称是某个配置项类型或关系类型的类名称，用于在代码中引用类。请不要将类型名称与显示名称相混淆，后者可在涉及类的用户接口上看到，但在代码中却毫无意义。

配置项 (CI)

CI 元素由一个 ID、一个 type 和一个 props 集合组成。

使用 UCMDB 更新方法更新 CI 时，ID 元素可以包含一个真实的 CMDB ID 或客户端指定的临时 ID。如果使用临时 ID，请将 temp 标记设置为 True。删除某一项时， ID 可以为空。 UCMDB 查询方法会将真实 ID 作为输入参数，并在查询结果中返回真实 ID。

type 可以是 CI 类型管理器中定义的任意类型名称。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的 “CI 类型管理器”。

props 元素是一个 CIProperties 集合。有关详细信息，请参阅 “CIProperties”（第 371 页）。

Relation

Relation 是链接两个配置项的实体。 Relation 元素由一个 ID、一个 type、两个链接项 （end1ID 和 end2ID）的标识符和一个 props 集合组成。

使用 UCMDB 更新方法更新 Relation 时，该 Relation 的 ID 的值既可以是真实 CMDB ID，又可以是临时 ID。删除某一项后， ID 可以为空。 UCMDB 查询方法会将真实 ID 作为输入参数，并在查询结果中返回真实 ID。

关系类型是指作为关系实例化基础的 UCMDB 类的**类型名称**。此类型可以是 CMDB 中定义的任意关系类型。有关类或类型的详细信息，请参阅 “查询 UCMDB 类模型”（第 305 页）。

有关详细信息，请参阅 《HP Universal CMDB 建模指南》中的 “CI 类型管理器”。

两个关系端的 ID 不得为空，因为创建当前关系的 ID 时需要使用这些 ID。但是，它们都可以通过客户端分配一个临时 ID。

props 元素是一个 CIProperties 集合。有关详细信息，请参阅 “CIProperties”（第 371 页）。

UCMDB 输出参数

本节介绍了服务方法最常用的输出参数。有关详细信息，请参阅架构文档。

本节包括以下主题：

- “CIs”（第 373 页）
- “ShallowRelation”（第 374 页）
- “Topology”（第 374 页）
- “CINode”（第 374 页）
- “RelationNode”（第 374 页）
- “TopologyMap”（第 374 页）
- “ChunkInfo”（第 375 页）

CIs

CIs 是 CI 元素的集合。

ShallowRelation

ShallowRelation 是链接两个配置项的实体，由一个 ID、一个 type、两个链接项（end1ID 和 end2ID）的标识符组成。关系类型是指作为关系实例化基础的 CMDB 类的**类型名称**。此类型可以是 CMDB 中定义的任意关系类型。

Topology

Topology 是 CI 元素和关系的图形。 Topology 由一个 CIs 集合，以及一个包含一个或多个 Relation 元素的 Relations 集合所组成。

CINode

CINode 由一个 CIs 集合和一个 label 组成。CINode 中的 label 是指在查询所使用的 TQL 节点中定义的标签。

RelationNode

RelationNode 是一组带有 label 的 Relations 集合。RelationNode 中的 label 是指在查询所使用的 TQL 节点中定义的标签。

TopologyMap

TopologyMap 是查询计算的输出，与 TQL 查询相匹配。TopologyMap 中的 labels 是指在查询所使用的 TQL 中定义的节点标签。

TopologyMap 的数据会以下列形式返回：

- CINodes。指一个或多个 CINode （请参阅 “CINode”（第 374 页））。
- relationNodes。指一个或多个 RelationNode （请参阅 “RelationNode”（第 374 页））。

这两种结构中的 label 构成了配置项和关系的列表。

ChunkInfo

当查询返回大量数据时，服务器会将这些数据划分为段（称为块）进行存储。您可以在查询所返回的 ChunkInfo 结构中找到客户端用于检索成块数据的信息。ChunkInfo 由必须检索的 numberOfChunks 和 chunksKey 组成。 chunksKey 是此特定查询所调用服务器上的数据的唯一标识符。

有关详细信息，请参阅 “处理大量响应”（第 299 页）。

10

HP Universal CMDB API

本章包括：

概念

任务

参考

概念

约定

本章使用以下约定：

- **UCMDB** 是指通用配置管理数据库本身，而 **HP Universal CMDB** 是指应用程序。
- UCMDB 元素和方法参数以在接口中指定的方式进行拼写。

使用 HP Universal CMDB API

请将本章与联机文档库中的 API Javadoc 配合使用。

HP Universal CMDB API 可用于将应用程序与 Universal CMDB (CMDB) 集成。通过该 API 可以：

- 在 CMDB 中添加、删除以及更新 CI 和关系
- 检索有关类模型的信息
- 运行假设分析场景
- 检索有关配置项和关系的信息

通常情况下，用于检索有关配置项和关系信息的方法会使用拓扑查询语言 (TQL)。有关详细信息，请参阅 《HP Universal CMDB 建模指南》中的 “拓扑查询语句”。

HP Universal CMDB API 的用户必须熟悉：

- Java 编程语言
- HP Universal CMDB

本节包括以下主题：

➤ “API 的用途”（第 379 页）

➤ “权限”（第 379 页）

API 的用途

API 用于满足多种业务需求。例如，第三方系统可以查询类模型，了解有关可用配置项 (CI) 的信息。有关更多用例，请参阅 “用案”（第 385 页）。

权限

管理员可以提供用于连接 API 的登录凭据。 API 客户端需要 CMDB 中定义的集成用户的用户名和密码。这些用户并不表示 CMDB 的人类用户，而是连接到 CMDB 的应用程序。

有关详细信息，请参阅 “创建集成用户”（第 382 页）。

应用程序的常规结构

静态工厂只存在一个，即 UcmdbServiceFactory。此工厂是应用程序的入口点。UcmdbServiceFactory 会采用 getServiceProvider 方法。这些方法将返回 **UcmdbServiceProvider** 接口的实例。

客户端会使用接口方法创建其他对象。例如，如果要创建新的查询定义，客户端将会：

1 从主 CMDB 服务对象获取查询服务

2 从服务对象获取查询工厂对象

3 从工厂获取新查询定义

```
UcmdbServiceProvider provider =
  UcmdbServiceFactory.getServiceProvider(HOST_NAME, PORT);
UcmdbService = provider.connect(provider.createCredentials(USERNAME,
  PASSWORD), provider.createClientContext("Test"));
TopologyQueryService queryService = ucmdbService.getTopologyQueryService();
TopologyQueryFactory factory = queryService.getFactory();
QueryDefinition queryDefinition = factory.createQueryDefinition("Test Query");
queryDefinition.addNode("Node").ofType("host");
Topology topology = queryService.executeQuery(queryDefinition);
System.out.println("There are " + topology.getAllCIs().size() + " hosts in uCMDB");
```

UcmdbService 中可用的服务如下：

服务方法	用途
getClassModelService	获取有关 CI 和关系类型的信息
getDDMConfigurationService	配置搜寻和依赖关系管理系统
getDDMManagementService	分析和查看搜寻和依赖关系管理系统的进度、结果和错误
getImpactAnalysisService	运行影响分析场景 （也称为**关联**）。
getQueryManagementService	管理对查询的访问操作 - 保存、删除和列出现有项。同时，还提供查询验证和查询依赖关系搜寻。
getResourceBundleManagement Service	资源标记 （捆绑服务）。允许显式创建新标记，以及从所有已标记资源中删除标记。
getSoftwareSignatureService	定义将通过搜寻和依赖关系管理系统进行搜寻的软件项
getTopologyQueryService	获取有关 IT 世界的信息
getTopologyUpdateService	更改 IT 世界中的信息

服务方法	用途
getViewService	查看执行服务 （执行定义、已保存的执行）和管理服务 （保存、删除、列出现有项）。同时，还提供视图验证和依赖关系搜寻。
getViewArchiveService	视图结果存档服务。允许保存当前视图结果并检索之前保存的结果。

客户端通过 HTTP 与服务器通信。

任务

将 API Jar 文件放入类路径中

使用此 API 集合时，需要文件 **ucmdb-api.jar**。通过在 Web 浏览器中输入 http://localhost:8080，并单击 “API 客户端下载” 链接可下载该文件。

在编译或运行应用程序之前，将 jar 文件放入类途径中。

创建集成用户

您可以为其他产品和 UCMDB 间的集成创建一个专用用户。此用户可以在服务器 SDK 中对使用 UCMDB 客户端 SDK 的产品进行身份验证，以及执行 API。以该 API 集合编写的应用程序必须具有集成用户凭据才能登录。

警告：只有集成用户才能够通过此 API 集合连接到 CMDB。如果其他类型的用户尝试连接，则即使使用 LDAP 验证，也可能会出现错误。

要创建集成用户，请执行以下操作：

1 启动 Web 浏览器并输入以下服务器地址。

http://localhost:8080/jmx-console。

可能需要用户名和密码 （默认为 sysadmin/sysadmin）才能登录。

2 在 UCMDB 下，单击 “service=UCMDB Security Services”，打开 “JMX MBEAN 视图” 页面。

3 查找 “CreateIntegrationUser” 操作。此方法可以使用以下参数：

- **customerId**。客户 ID。
- **username**。集成用户的用户名。
- **password**。集成用户的密码。
- **dataStoreOrigin**。将使用此集成用户的产品名称。

以下操作适用于集成用户管理：

- **DeleteIntegrationUser**。删除指定的集成用户。
- **ExportIntegrationUser**。将集成用户导出到指定路径中的 XML 文件 （位于服务器计算机上）。
- **getIntegrationUser**。显示集成用户信息。
- **changeIntegrationUserPassword**。更改集成用户的密码。
- **canUserAuthenticate**。 **isIntegrationUser** 为 **True**：集成用户能够使用指定的凭据进行身份验证吗？

4 单击 “调用”。

单击 “返回到 MBean 视图”，创建更多用户，或关闭 JMX 控制台。

5 以管理员身份登录到 UCMDB。

6 在 “管理” 选项卡中，运行 “程序包管理器”。

7 单击 “新建” 图标。

8 输入新程序包的名称，并单击 “下一步”。

9 在 “资源选择” 选项卡中，单击 “管理” 下的 “集成用户”。

10 选择一个或多个使用 JMX 控制台创建的用户。

11 单击 “下一步”，然后单击 “完成”。此时，新程序包会出现在程序包管理器的 “程序包名称” 列表中。

12 将该程序包部署到要运行 API 应用程序的用户。

有关详细信息，请参阅 《HP Universal CMDB 管理指南》中的 “部署包”。

注意：

集成用户基于客户。要创建功能更强的集成用户以供客户使用，请使用 **systemUser**，并将其中的 **isSuperIntegrationUser** 标记设置为 **True**。使用 **systemUser** 方法 （**createSystemUser**、 **removeSystemUser**、 **showAllSystemUsers**、 **changeSystemUserPassword**、 **canSuperIntegrationUserAuthenticate** 等）。

系统中存在两个现成用户。建议在使用 **changeSystemUserPassword** 方法进行安装后，更改这两个用户的密码。

- **sysadmin/sysadmin**
- **UISysadmin/UISysadmin** （此用户同时也是超级集成用户 **SuperIntegrationUser**）。

如果使用 **changeSystemUserPassword** 更改 UISysadmin 密码，则必须执行以下方法：在 JMX 控制台中，查找 “UCMDB-UI:name=UCMDB Integration” 服务。以集成用户的用户名和新密码运行 “setCMDBSuperIntegrationUser”。

参考

HP Universal CMDB API 参考

有关可用 API 的完整文档，请参阅 “API 简介”（第 289 页）。

用案

以下用例假定两个系统为：

- HP Universal CMDB 服务器
- 包含配置项库的第三方系统

本节包括以下主题：

填充 CMDB

用例：

- 第三方资产管理仅使用可在资产管理中使用的信息更新 CMDB
- 很多第三方系统通过填充 CMDB 来创建可跟踪更改并执行影响分析的中心 CMDB
- 第三方系统按照第三方业务逻辑创建配置项和关系，以利用 UCMDB 查询功能

查询 CMDB

用例：

- 第三方系统通过检索 SAP TQL 的结果，来获取表示 SAP 系统的配置项和关系
- 第三方系统获取过去五个小时内添加或更改的 Oracle 服务器的列表
- 第三方系统获取主机名包含 lab 子字符串的服务器的列表
- 第三方系统通过获取指定 CI 的相邻项，来查找与给定 CI 相关的元素

查询类模型

用例：

- 用户通过第三方系统指定要从 CMDB 中检索的数据集合。通过类模型可以生成用户界面，从而向用户显示可能的属性，并提示用户输入所需的数据。然后，用户可以选择要检索的信息。
- 当用户无法访问 UCMDB 用户界面时，第三方系统将搜索类模型。

分析更改影响

用例：

第三方系统输出一个受指定主机的变更影响的业务服务列表。

示例

本节包括以下主题：

- “入口点示例”（第 387 页）
- “查询示例”（第 387 页）

➤ “拓扑查询示例”（第 389 页）
➤ “拓扑更新示例”（第 390 页）
➤ “影响分析示例”（第 390 页）

入口点示例

```
final String HOST_NAME = "localhost";
final int PORT = 8080;
UcmdbServiceProvider provider =
    UcmdbServiceFactory.getServiceProvider(HOST_NAME, PORT);
final String USERNAME = "integration_user";
final String PASSWORD = "integration_password";
Credentials credentials =
    provider.createCredentials(USERNAME, PASSWORD),
ClientContext clientContext = provider.createClientContext("Example");
UcmdbService ucmdbService = provider.connect(credentials, clientContext);
```

查询示例

以下示例为如何获取单个类定义，以及如何获取所有 CIT 定义及其属性的列表。

检索类定义

```
ClassModelService classModelService
    = ucmdbService.getClassModelService();
String typeName = "disk";
ClassDefinition def =
    classModelService.getClassDefinition(typeName);
System.out.println("Type " + typeName + " is derived from type "
    + def.getParentClassName());
System.out.println("Has " + def.getChildClasses().size() +
    " derived types");
System.out.println("Defined and inherited attributes:");
for (Attribute attr : def.getAllAttributes().values()) {
    System.out.println("Attribute " + attr.getName() +
    " of type " + attr.getType());
}
```

检索 CIT 定义和属性的列表

此示例将查询某个 CIT 的属性，并打印这些属性的名称和类型。

```
ClassModelService classModelService =
      ucmdbService.getClassModelService();
for (ClassDefinition def : classModelService.getAllClasses()) {
     System.out.println("Type " + def.getName() +
          " (" + def.getDisplayName() + ") is derived from type "
          + def.getParentClassName());
     System.out.println
          ("Has " + def.getChildClasses().size() + " derived types");
     System.out.println
          ("Defined and inherited attributes:");
     for (Attribute attr : def.getAllAttributes().values()) {
          System.out.println
            ("Attribute " + attr.getName() +
             " of type " + attr.getType());
     }
}
```

拓扑查询示例

```
TopologyQueryService queryService =
    ucmdbService.getTopologyQueryService();
TopologyQueryFactory queryFactory =
    queryService.getFactory();
QueryDefinition queryDefinition =
    queryFactory.createQueryDefinition
        ("Get hosts with more than one network interface");
String hostNodeName = "Host";
QueryNode hostNode =

queryDefinition.addNode(hostNodeName).ofType("host").queryProperty("display_label"
);
QueryNode ipNode =
    queryDefinition.addNode("IP").ofType("ip").queryProperty("ip_address");
hostNode.linkedTo(ipNode).withLinkOfType("contained").atLeast(2);
Topology topology = queryService.executeQuery(queryDefinition);
Collection<TopologyCI> hosts = topology.getCIsByName(hostNodeName);
for (TopologyCI host : hosts) {
    System.out.println("Host " + host.getPropertyValue("display_label"));
    for (TopologyRelation relation : host.getOutgoingRelations()) {
         System.out.println
        ("  has IP " + relation.getEnd2CI().getPropertyValue("ip_address"));
    }
}
```

拓扑更新示例

```
TopologyUpdateService topologyUpdateService =
    ucmdbService.getTopologyUpdateService();
TopologyUpdateFactory topologyUpdateFactory =
    topologyUpdateService.getFactory();
TopologyModificationData topologyModificationData =
    topologyUpdateFactory.createTopologyModificationData();
CI host = topologyModificationData.addCI("host");
host.setPropertyValue("host_key", "test1");
CI ip = topologyModificationData.addCI("ip");
ip.setPropertyValue("ip_address", "127.0.0.10");
ip.setPropertyValue("ip_domain", "DefaultDomain");
topologyModificationData.addRelation("contained", host, ip);
topologyUpdateService.create
    (topologyModificationData, CreateMode.IGNORE_EXISTING);
```

影响分析示例

```
ImpactAnalysisService impactAnalysisService =
    ucmdbService.getImpactAnalysisService();
ImpactAnalysisFactory impactFactory =
    impactAnalysisService.getFactory();
ImpactAnalysisDefinition definition =
    impactFactory.createImpactAnalysisDefinition();
definition.addTriggerCI(disk).withSeverity
    (impactFactory.getSeverityByName("Warning(2)"));
definition.useAllRules();
ImpactAnalysisResult impactResult =
    impactAnalysisService.analyze(definition);
AffectedTopology affectedCIs =
    impactResult.getAffectedCIs();
for (AffectedCI affectedCI : affectedCIs.getAllCIs()) {
     System.out.println("Affected " +
     affectedCI.getType() + " " + affectedCI.getId() +
     " - severity " + affectedCI.getSeverity());
}
```

索引

W

X

Y

Z