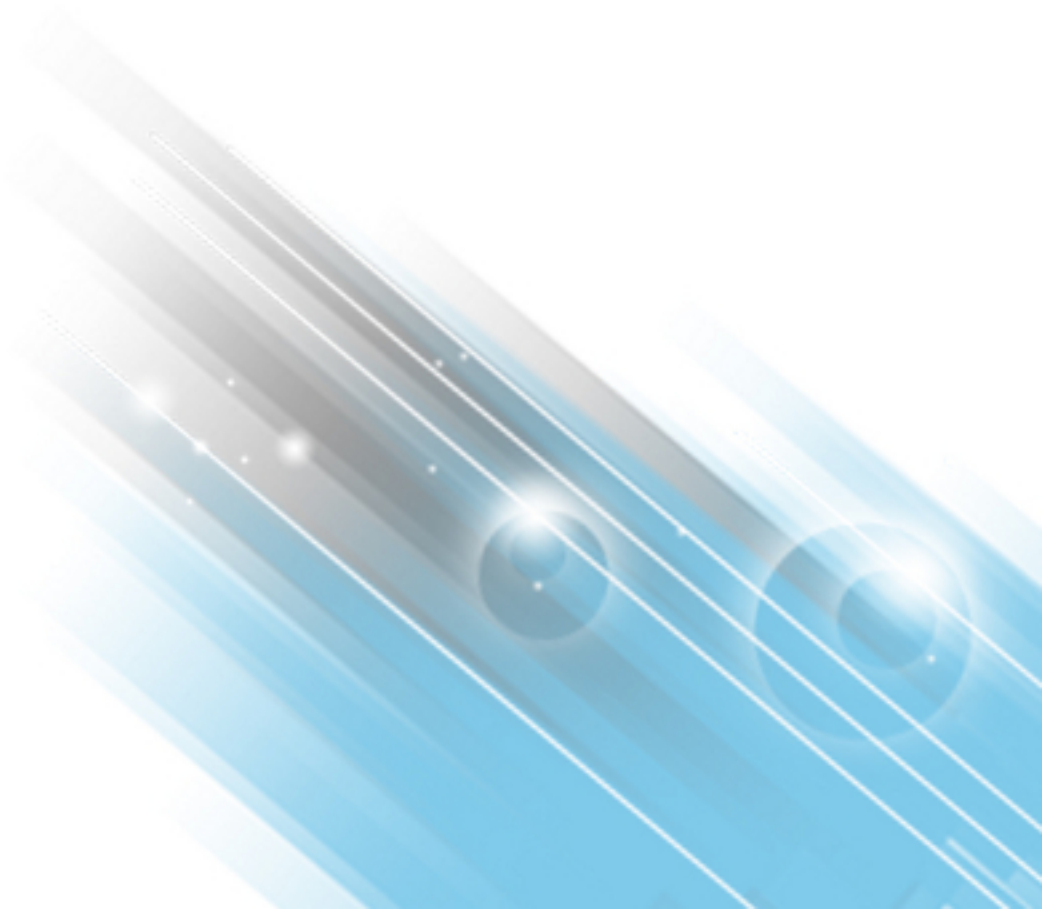
HP Universal CMDB

软件版本： 10.20

开发人员参考指南



文档发布日期： 2015 年 1 月
软件发布日期： 2015 年 1 月

法律声明

担保

HP 产品和服务的唯一担保已在此类产品和服务随附的明示担保声明中提出。此处的任何内容均不构成额外担保。HP 不会为此处出现的技术或编辑错误或遗漏承担任何责任。

此处所含信息如有更改，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

© Copyright 2002 - 2015 Hewlett-Packard Development Company, L.P.

商标声明

Adobe™ 是 Adobe Systems Incorporated 的商标。

Microsoft® 和 Windows® 是 Microsoft Corporation 在美国注册的商标。

UNIX® 是 The Open Group 的注册商标。

产品包括 'zlib' 通用压缩库的接口，版权所有 © 1995-2002 Jean-loup Gailly and Mark Adler。

文档更新

此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。
- 文档发布日期，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：
https://softwaresupport.hp.com

需要注册 HP passport 才能登录此网站。要注册 HP passport ID，请访问：
https://hpp12.passport.hp.com/hppcf/createuser.do

或单击“HP Software Support”页面顶部的“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

请访问 HP 软件联机支持网站：**https://softwaresupport.hp.com**

此网站提供了联系信息，以及有关 HP 软件提供的产品、服务和支持的详细信息。

HP 软件联机支持提供客户自助解决功能。通过该联机支持，可快速高效地访问用于管理业务的各种交互式技术支持工具。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：

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

要查找有关访问级别的详细信息，请访问：

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

HP Software Solutions Now 可访问 HPSW 解决方案和集成门户网站。此网站将帮助您寻找可满足您业务需求的 HP 产品解决方案，包括 HP 产品之间的集成的完整列表以及 ITIL 流程的列表。此网站的 URL 为 **http://h20230.www2.hp.com/sc/solutions/index.jsp**

关于此 PDF 版本联机帮助

本文档是联机帮助的 PDF 版本。提供此 PDF 文件是为了便于您打印帮助信息的多个主题，或者阅读 PDF 格式的联机帮助。由于此内容最初创建时是作为联机帮助在 Web 浏览器中查看的，因此某些主题可能无法正常显示。某些交互主题可能无法在该 PDF 版本中显示。这些主题可以通过联机帮助成功打印出来。

目录

第 I 部分: 创建搜寻和集成适配器 11
第 1 章: 适配器开发和编写 12
适配器开发和编写概述 12
内容创建 12
适配器开发周期 13
数据流管理和集成 15
关联业务价值与搜寻开发 16
研究集成要求 16
开发集成内容 18
开发搜寻内容 20
搜寻适配器和相关组件 20
分离适配器 21
实施搜寻适配器 22
步骤 1: 创建适配器 25
步骤 2: 将作业分配到适配器 31
步骤 3: 创建 Jython 代码 32
配置远程进程执行 32
第 2 章: 开发 Jython 适配器 34
HP 数据流管理 API 参考 34
创建 Jython 代码 34
在 Jython 中使用外部 Java JAR 文件 35
执行代码 35
修改现成脚本 35
Jython 文件结构 36
导入 36
主函数 – DiscoveryMain 36
函数定义 37
Jython 脚本生成的结果 38
ObjectStateHolder 语法 38
发送大量数据 39
框架实例 40
查找正确的凭据（针对连接适配器） 43
处理 Java 抛出的异常 45
Jython 版本 2.1 至 2.5.3 的迁移疑难解答 45
支持 Jython 适配器中的本地化 47
添加新的语言支持 47
更改默认语言 48
确定用于编码的字符集 48
定义使用本地化数据运行的新作业 49
解码命令，但不使用关键字 50
使用资源捆绑包 50

API 参考 51
记录数据流管理代码 53
Jython 库和实用程序 54
第 3 章: 错误消息 57
错误消息概述 57
错误编写约定 57
错误严重度级别 59
第 4 章: 映射使用者-提供程序依赖关系 61
依赖关系搜寻概述 61
提供程序和使用者 61
依赖关系签名 62
依赖关系映射流 62
依赖关系签名文件 62
依赖关系签名文件的结构 63
变量和概念 63
默认值 66
IP 地址变量类型 66
定义使用者的描述符 66
定义依赖关系 68
编写搜索表达式 68
使用变量的默认值 70
指定配置文档的路径 71
配置文档替代 72
在多个文档间定义的依赖关系 73
属性配置文档 76
XML 配置文档 79
文本配置文档 82
指定搜索范围 84
定义 TQL 查询 86
打包和部署多个依赖关系签名文件 87
编译错误 87
依赖关系搜索适配器 88
创建依赖关系搜索适配器 89
定义使用者-提供程序适配器 90
定义输入 TQL 查询和目标数据 91
指定变量值 91
指定概念变量值 92
编写 Jython 脚本 94
适配器限制 96
完整示例 96
开发工作流 96
开发依赖关系签名 97
开发适配器 99
第 5 章: 开发常规数据库适配器 103
常规数据库适配器概述 103
常规数据库适配器的 TQL 查询 104

调节 105
Hibernate 作为 JPA 提供程序 105
准备创建适配器 107
准备适配器包 112
配置适配器 – 最小方法 114
配置 adapter.conf 文件 115
示例：使用简便方法填充节点和 IP 地址 115
配置适配器 – 高级方法 118
实施插件 121
部署适配器 124
编辑适配器 124
创建集成点 124
创建视图 124
计算结果 124
查看结果 125
查看报告 125
启用日志文件 125
使用 Eclipse 在 CIT 属性和数据库表之间进行映射 125
适配器配置文件 130
adapter.conf 文件 132
simplifiedConfiguration.xml 文件 132
orm.xml 文件 134
reconciliation_types.txt 文件 145
reconciliation_rules.txt 文件（用于向后兼容） 145
transformations.txt 文件 146
discriminator.properties 文件 147
replication_config.txt 文件 148
fixed_values.txt 文件 148
Persistence.xml 文件 149
使用 NT 身份验证连接到数据库 149
配置 Persistence.xml 文件以便 SCCM 集成使用 NTLM 身份验证 150
现成的转换器 151
插件 155
配置示例 156
适配器日志文件 163
外部参考 165
疑难解答和局限性 – 开发常规数据库适配器 165
第 6 章: 开发 Java 适配器 166
联合框架概述 166
适配器与联合框架的映射交互 170
联合 TQL 查询的联合框架 170
联合框架、服务器、适配器和映射引擎之间的交互 171
用于填入的联合框架流 180
适配器接口 181
调试适配器资源 182
为新外部数据源添加适配器 182

创建示例适配器 188
XML 配置标记和属性 189
DataAdapterEnvironment 界面 191
OutputStream openResourceForWriting(String resourceName) throws FileNotFoundException; 191
InputStream openResourceForReading(String resourceName) throws FileNotFoundException; 191
Properties openResourceAsProperties(String propertiesFile) throws IOException; 192
String openResourceAsString(String resourceName) throws IOException; 192
public void saveResourceFromString(String relativeFileName, String value) throws IOException; 193
boolean resourceExists(String resourceName); 193
boolean deleteResource(String resourceName); 193
Collection<String> listResourcesInPath(String path); 193
DataAdapterLogger getLogger(); 194
DestinationConfig getDestinationConfig(); 194
int getChunkSize(); 194
int getPushChunkSize(); 194
ClassModel getLocalClassModel(); 194
CustomerInformation getLocalCustomerInformation(); 194
Object getSettingValue(String name); 195
Map<String, Object> getAllSettings(); 195
boolean isMTEnabled(); 195
String getUcmdbServerHostName(); 195
第 7 章: 开发推送适配器 196
开发和部署推送适配器 196
生成适配器包 196
疑难解答 198
推送适配器的 TQL 最佳实践 199
创建映射 199
构建映射文件 199
准备映射文件 200
编写 Jython 脚本 202
支持差异同步 205
常规 XML 推送适配器 SQL 查询 206
常规 Web 服务推送适配器 207
映射文件引用 224
映射文件架构 226
映射结果架构 234
自定义 236
第 8 章: 开发常规适配器 238
实例同步 238
使用常规适配器实现数据推送 238
推送概述 239
映射文件 239
Groovy Traveler 241

编写 Groovy 脚本 244
实施 PushAdapterConnector 接口 245
使用常规适配器实现数据填入 246
填入框架体系结构 246
填入中涉及的主要项目 247
填入 TQL 查询 248
填入映射文件 248
自动链接填入 251
手动链接填入 251
填入连接器 252
填入请求输入 254
填入请求输出 256
填入适配器模式 257
显式外部 ID 映射 258
全局 ID 推回 258
使用常规适配器实现数据联合 259
联合映射方法 259
常规适配器联合 API 260
用于联合的常规适配器连接器接口 261
支持的联合查询 262
如何设置联合 262
配置适配器设置 263
从日志文件设置联合查询 263
联合设置示例 267
调节 272
常规适配器 API 272
资源定位器 API 273
创建常规适配器包 273
生成适配器包 274
填入 TQL 查询 275
示例包 276
推送和填入映射之间的差异 278
常规适配器日志文件 278
使用常规适配器框架的适配器 279
常规适配器 XML 架构参考 279

第 II 部分: 使用 API 280
第 9 章: API 简介 281
API 概述 281
第 10 章: HP Universal CMDB API 282
约定 282
使用 HP Universal CMDB API 282
应用程序的常规结构 283
将 API Jar 文件放入类路径中 285
创建集成用户 285
UCMDB API 用例 287

示例 ... 288
第 11 章: HP Universal CMDB Web 服务 API ... 289
约定 ... 289
HP Universal CMDB Web 服务 API 概述 ... 289
调用 HP Universal CMDB Web 服务 ... 292
查询 CMDB ... 292
更新 CMDB ... 295
查询 UCMDB 类模型 ... 296
getClassAncestors ... 296
getAllClassesHierarchy ... 296
getCmdbClassDefinition ... 297
影响分析查询 ... 297
UCMDB 常规参数 ... 297
UCMDB 输出参数 ... 299
UCMDB 查询方法 ... 300
executeTopologyQueryByNameWithParameters ... 301
executeTopologyQueryWithParameters ... 301
getChangedCIs ... 302
getCINeighbours ... 303
getCIsByID ... 304
getCIsByType ... 304
getFilteredCIsByType ... 304
getQueryNameOfView ... 307
getTopologyQueryExistingResultByName ... 308
getTopologyQueryResultCountByName ... 308
pullTopologyMapChunks ... 309
releaseChunks ... 310
UCMDB 更新方法 ... 311
addCIsAndRelations ... 311
addCustomer ... 312
deleteCIsAndRelations ... 312
removeCustomer ... 312
updateCIsAndRelations ... 313
UCMDB 影响分析方法 ... 313
calculateImpact ... 313
getImpactPath ... 314
getImpactRulesByNamePrefix ... 315
实际状态 Web 服务 API ... 315
UCMDB Web 服务 API 用例 ... 317
示例 ... 317
第 12 章: 数据流管理 Java API ... 319
使用数据流管理 Java API ... 319
第 13 章: 数据流管理 Web 服务 API ... 320
数据流管理 Web 服务 API 概述 ... 320
约定 ... 320
调用 HP 数据流管理 Web 服务 ... 321

数据流管理方法和数据结构 321
数据结构 321
管理搜寻作业方法 322
管理触发器方法 324
域和探测器数据方法 325
凭据数据方法 328
数据刷新方法 330
代码示例 332
添加凭据示例 334

发送文档反馈 338

第 I 部分: 创建搜寻和集成适配器

第 1 章: 适配器开发和编写

本章包括:

- 适配器开发和编写概述 12
- 内容创建 12
- 开发集成内容 18
- 开发搜寻内容 20
- 实施搜寻适配器 22
- 步骤 1: 创建适配器 25
- 步骤 2: 将作业分配到适配器 31
- 步骤 3: 创建 Jython 代码 32
- 配置远程进程执行 32

适配器开发和编写概述

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。

以下各节介绍了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。）

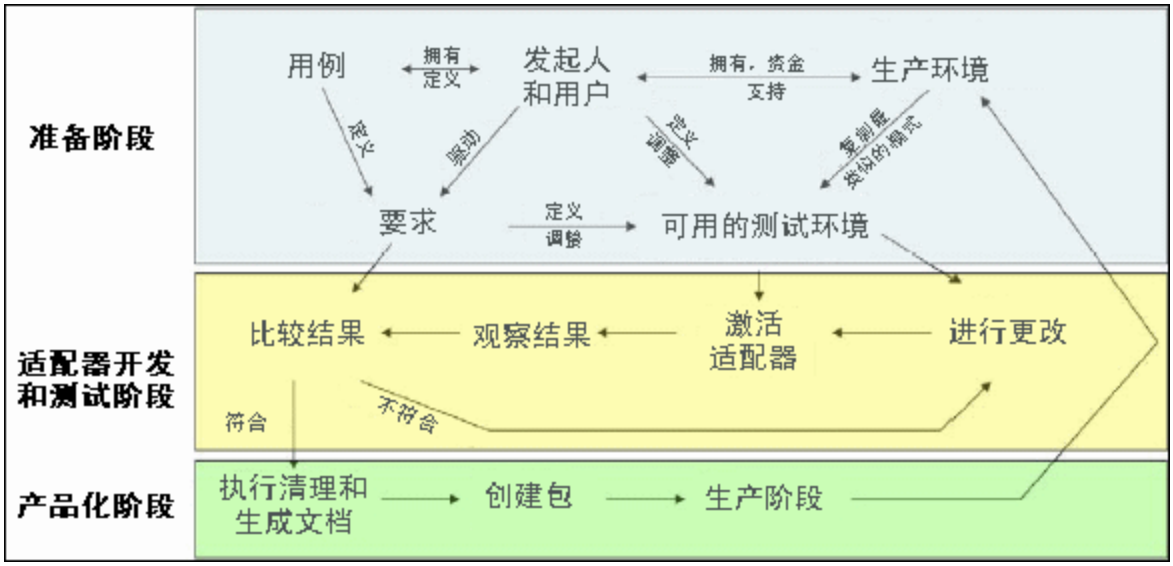
内容创建

本节包括:

- 适配器开发周期 (第 13 页)
- 数据流管理和集成 (第 15 页)
- 关联业务价值与搜寻开发 (第 16 页)
- 研究集成要求 (第 16 页)

适配器开发周期

下图为适配器编写流程图。其中的大部分时间都用于中间环节，该环节涉及开发和测试过程的迭代循环。



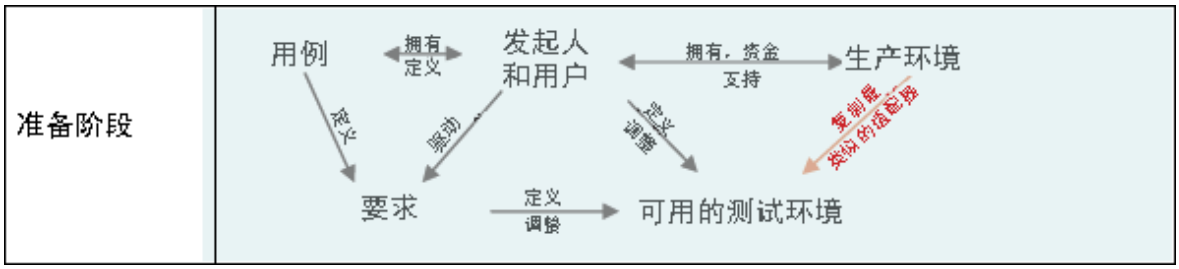
适配器开发过程中的每个阶段都基于上一阶段。

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，来创建包 *.zip 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 (第 13 页)
- 适配器开发和测试 (第 14 页)
- 适配器的包装和产品化 (第 14 页)

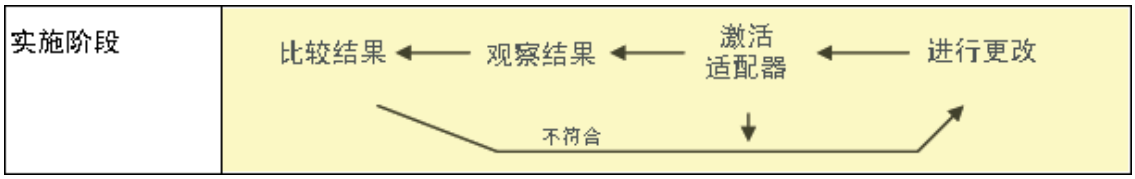
研究和准备阶段



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. 了解适配器用来收集数据的方法:
   - 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. 将程序包内容解压缩到工作区，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

适配器的包装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

数据流管理适配器可以与其他产品集成。请考虑以下定义：

- 数据流管理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，数据流管理也一样。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。

> **备注:** 有关字段包的示例，请联系 HP 软件支持。

要将其他产品与 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，您的计划都应该回答以下问题:

- 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，变更的跟踪方式、重要变更，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数据流管理搜寻到的 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，请参阅适配器开发和编写概述 (第 12 页)。

本节包括以下主题:

- 修改现有适配器 (第 17 页)
- 编写新适配器 (第 17 页)
- 模型研究 (第 17 页)

- 技术研究 (第 17 页)
- 有关选择数据访问方式的准则 (第 18 页)
- 摘要 (第 18 页)

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
- 不会搜索或创建特定的关系

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 软件支持，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 到 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。因为这意味着数据流管理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。您可以在“数据流管理”>“Data Flow Probe 设置”>“域和探测器”窗格中查看当前支持的 HP Universal CMDB 连接协议。有关各协议的详细信息，请参阅《HP UCMDB Discovery and Integrations Content Guide》中描述受支持协议的章节。

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 软件支持。

备注: 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。

摘要

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？

  如果这些问题的答案是肯定的，则需要“填入”。
- 集成联合数据是否为视图和 TQL 查询的实时数据？数据变更的准确性是否重要？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？

  如果这些问题的答案是肯定的，则需要“联合”。
- 集成是否需要将数据推送到远程数据源？

如果这些问题的答案是肯定的，则需要“数据推送”。

- 是否有 CI 的 ID 长度大于 60 个字符？

  如果此问题的答案是肯定的，则**减小**所有相关 CI 的 ID 长度，以便其 ID 长度不会超过最大 60 个字符。

> **备注:** “联合”与“填入”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”。

可使用五个不同的选项创建集成适配器：

1. Jython 适配器：
   - 经典搜寻模式
   - 在 Jython 中写入
   - 用于填入

   有关详细信息，请参阅开发 Jython 适配器 (第 34 页)。
2. 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”、“填入”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 166 页)。
3. 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“填入”（使用为支持变更而实施的 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 103 页)。
4. 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 196 页)。

5. 增强的常规推送适配器：
   - 常规推送适配器的所有上述功能
   - 基于根元素的适配器
   - 将 UCMDB 树数据结构映射到目标树数据结构

   有关详细信息，请参阅使用常规适配器实现数据推送 (第 238 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	常规 DB 适配器	常规推送适配器	增强的常规推送适配器
填入	✓	✓	✓	×	×
联合	×	✓	✓	×	×
数据推送	×	✓	×	✓	✓

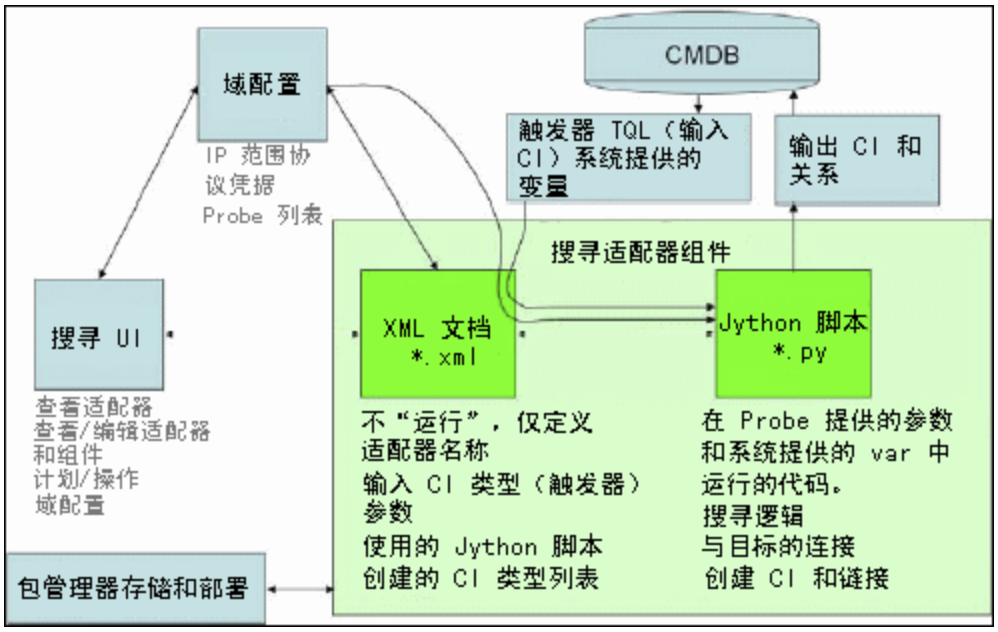
开发搜寻内容

本节包括：

- 搜寻适配器和相关组件 (第 20 页)
- 分离适配器 (第 21 页)

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”进行管理，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。

数据流管理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“Universal Discovery”、“适配器管理”和“Data Flow 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

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。数据流管理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。

实施搜寻适配器

数据流管理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. **创建适配器。**

   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，请参阅步骤 1：创建适配器 (第 25 页)。

2. **创建搜寻作业。**

   使用进度信息和触发查询配置作业。有关详细信息，请参阅步骤 2：将作业分配到适配器 (第 31 页)。

3. **编辑搜寻代码。**

   您可以编辑适配器文件中的 Jython 或 Java 代码，以及涉及数据流管理框架的代码。有关详细信息，请参阅步骤 3：创建 Jython 代码 (第 32 页)。

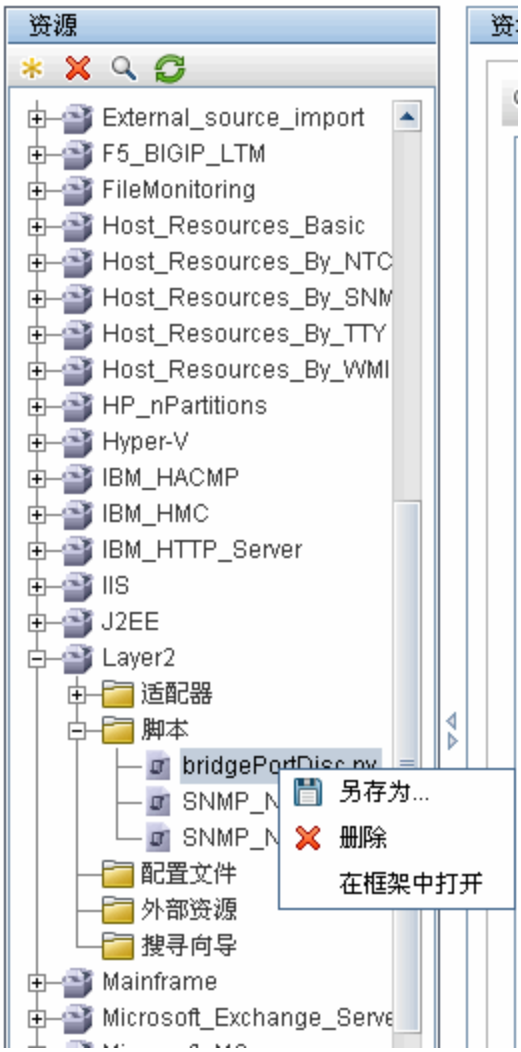
要编写新适配器，需要创建所有上述组件，其中每一个组件都与上一步中的组件自动捆绑。例如，创建作业并选择相关适配器后，适配器文件将与作业绑定。

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 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。

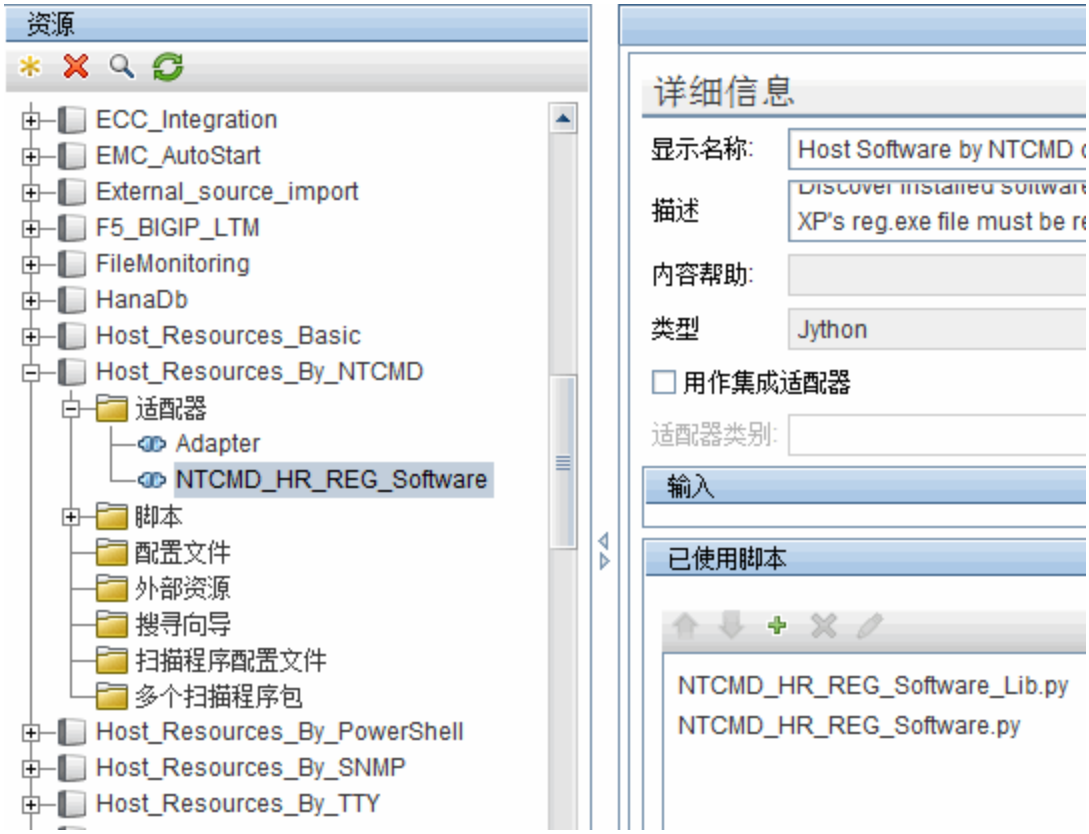
适配器可以通过 Java 类名称或 Jython 脚本名称引用此代码。本节中，我们将讨论以 Jython 脚本的形式编写数据流管理代码。

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 32 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:



备注: 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 CI。适配器从输入触发 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 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“数据流管理概念”。

本节包括以下主题:

- 定义适配器输入（触发 CIT 和输入查询）(第 25 页)
- 定义适配器输出 (第 28 页)
- 替代适配器参数 (第 29 页)
- 替代探测器选择 - 可选 (第 30 页)
- 配置远程进程的类路径 - 可选 (第 31 页)

1. 定义适配器输入（触发 CIT 和输入查询）

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

   - 触发 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。

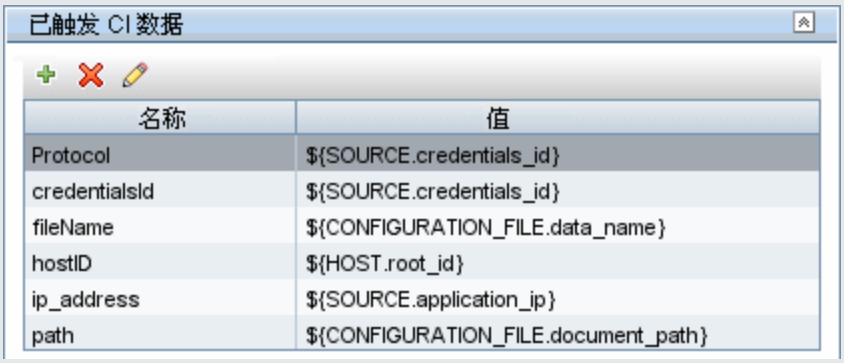
     如果适配器需要与触发 CI 相关的其他 CI 信息，则可以向输入 TQL 中添加其他节点。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的“如何将查询节点和关系添加到 TQL 查询”。
   - 触发 CI 数据中包含有关触发 CI 的全部所需信息，以及输入 TQL 中定义的其他节点的信息。数据流管理可以使用变量从 CI 检索数据。当任务下载到探测器后，触发 CI 数据变量将替换为真实 CI 实例的属性中所存在的实际值。
   - 如果目标数据的值是列表，则可以定义该列表中要发送到探测器的项数。要定义项数，请在后跟项数的默认值后添加一个冒号。如果目标数据没有默认值，则输入两个冒号。

     例如，如果输入以下目标数据：name=portId, value= ${PHYSICALPORT.root_id:NA:1} 或 name=portId, value= ${PHYSICALPORT.root_id::1}，则仅将此端口列表中的第一个端口发送到探测器。

   将变量替换为实际数据的示例:

在本示例中，变量可以将 **IpAddress** CI 数据替换为系统的真实 **IpAddress** CI 实例中存在的实际值。

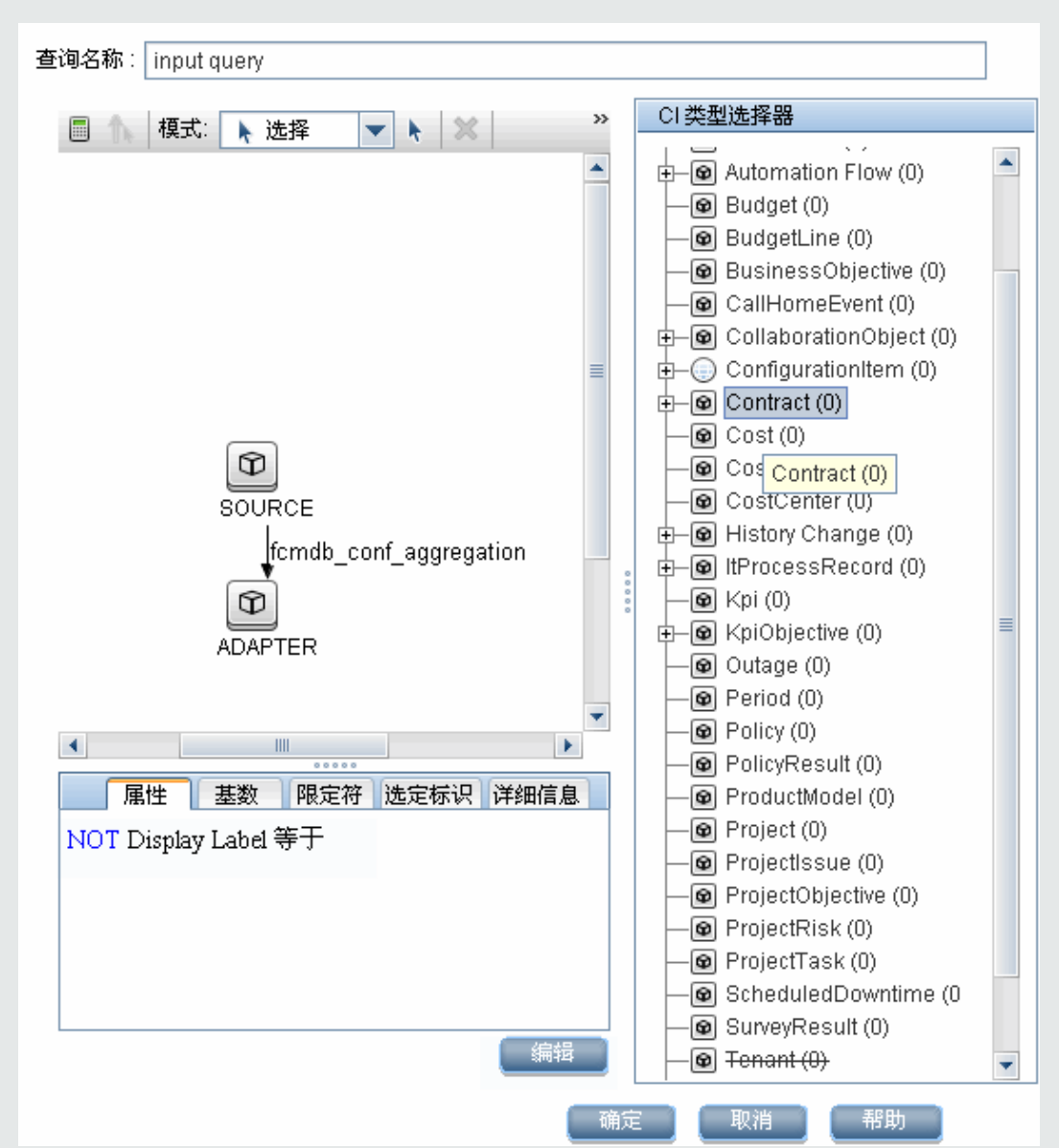
IpAddress CI 的触发的 CI 数据包括 fileName 变量。使用此变量可以将输入 TQL 中的 **CONFIGURATION_DOCUMENT** 节点替换为主机上配置文件的实际值：



触发 CI 数据将上载到探测器，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 Framework检索已定义变量的实际值：

Framework.getTriggerCIData ('ip_address')

fileName 和 path 变量将使用 **CONFIGURATION_DOCUMENT** 节点（在“输入查询编辑器”中定义，详见上一示例）的 data_name 和 document_path 属性。



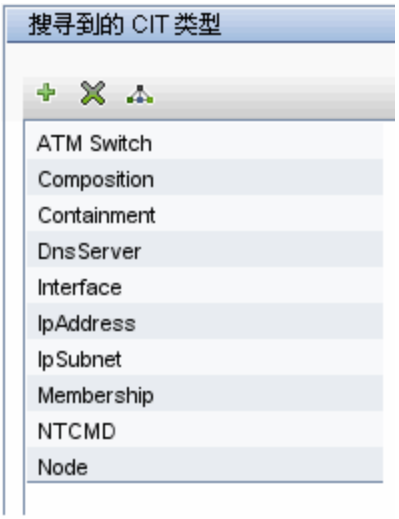
单击缩略图以查看完整大小的图像。

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

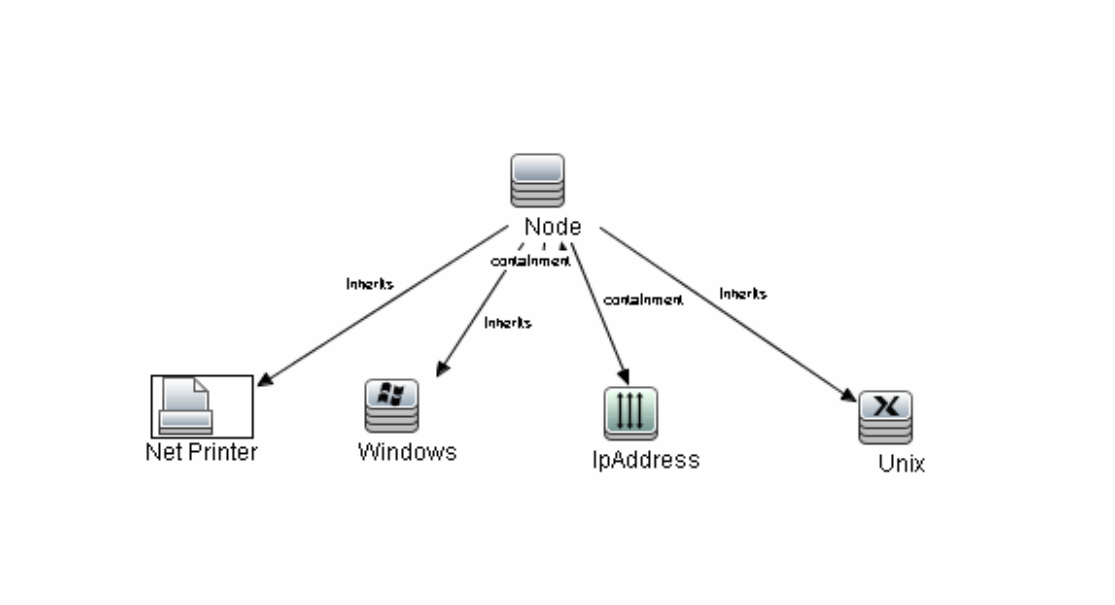
密钥	显示名称	名称	类型	描述	默认值	可见
	ack_cleared_time	ack_cleared_time	ong			
	ack_id	ack_id	string			
	BODY_ICON	BODY_ICON	string		host	
	city	City	string	City location		✔
	codepage	CodePage	string	System su...		
	contextmenu	Context Menu	string_list	Context me...	itCIs	
	country	Country	string	Country loc...		✔
	credentials_id	Reference to the cre	string	Reference ...		
	data_adminstate	Admin State	adminstate	Admin State	Managed	

2. 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 CIT” 按钮）:



搜寻到的 CI 将由数据流管理代码（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 38 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。

a. 访问“数据流管理”>“适配器管理”。
b. 在“资源”窗格中，选择“Network”>“适配器”>“NSLOOKUP_on_Probe”。
c. 在“适配器定义”选项卡中，查找“搜寻到的 CIT 类型”窗格。
d. 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“适配器定义选项卡”。

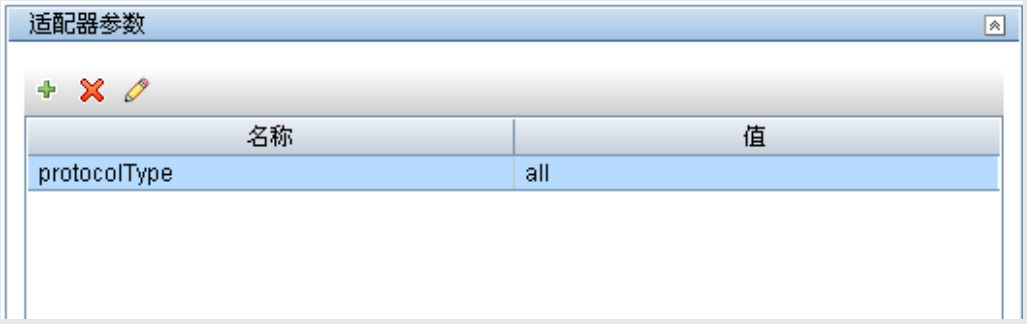
3. 替代适配器参数

如果要为多个作业配置适配器，则可以替代适配器参数。例如，适配器 SQL_NET_Dis_Connection 可由 MSSQL Connection by SQL 和 Oracle Connection by SQL 作业同时使用。

替代适配器参数的示例：

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

a. 访问“数据流管理”>“适配器管理”。
b. 在“资源”窗格中，选择“Database_Basic”>“适配器” >“SQL_NET_Dis_Connection”。
c. 在“适配器定义”选项卡中，查找“适配器参数”窗格。protocolType 参数值为 **All**:



d. 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”。

e. 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。

有关在“适配器参数”窗格中添加、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“适配器定义选项卡”。

数据流管理将根据此参数开始查找 Microsoft SQL Server 实例。

4. 替代探测器选择 - 可选

UCMDB 服务器中的分派机制可以获取 UCMDB 收到的触发 CI，并自动根据以下选项之一选择应为每个触发器 CI 运行作业的探测器：

- **对于 IP 地址 CI 类型：**选择为此 IP 定义的探测器。
- **对于运行软件 CI 类型：**请使用属性 **application_ip** 和 **application_ip_domain**，并选择在相关域中为 IP 定义的探测器。
- **对于其他 CI 类型：**根据 CI 的相关节点（如果存在）获取此节点的 IP。

将根据 CI 的相关节点自动完成探测器选择。在获取该 CI 的相关节点之后，分派机制将选择此节点的一个 IP，并根据探测器的网络范围定义选择探测器。

在以下情况中，需要手动指定探测器，而不使用自动分派机制：

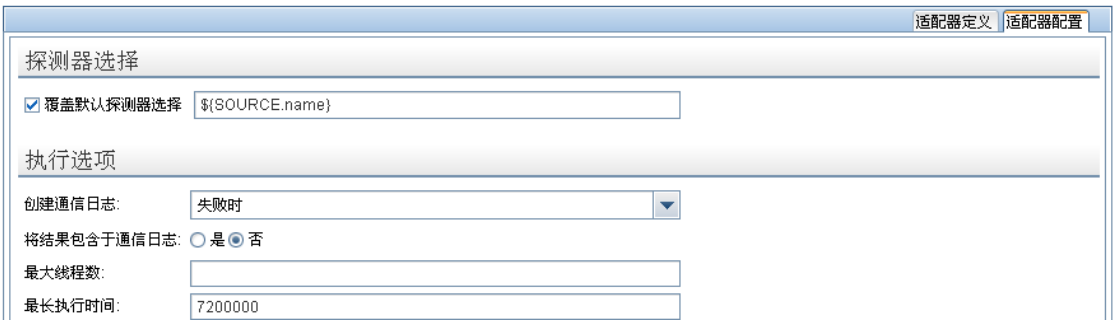
- 已经知道应为适配器运行哪个探测器，并且不需要使用自动分派机制选择探测器（例如，触发 CI 是 Probe Gateway 时）。
- 自动探测器选择可能失败。在以下情况下可能会发生上述失败：
  - 触发 CI 没有相关节点（例如 network CIT）
  - 触发 CI 的节点具有多个 IP，并且每个 IP 属于不同的探测器。

要手动指定与适配器一起使用的探测器，请执行以下操作：

- 选择适配器并单击“适配器配置”选项卡。
- 在“触发器分派选项”下，选择“替代默认探测器选择”。
- 在框中，使用以下任一格式输入探测器：

探测器名称	探测器的名称
IP 地址	探测器的 IP 地址，可采用 IPv4 或 IPv6 格式定义
IP,域	IPv4 格式：**16.59.63.86,默认域** IPv6 格式：**2001:0:9d46:953c:34a9:1e6b:f2ff:fffe,自定义域**
域名	应从中选择探测器的域。

例如：



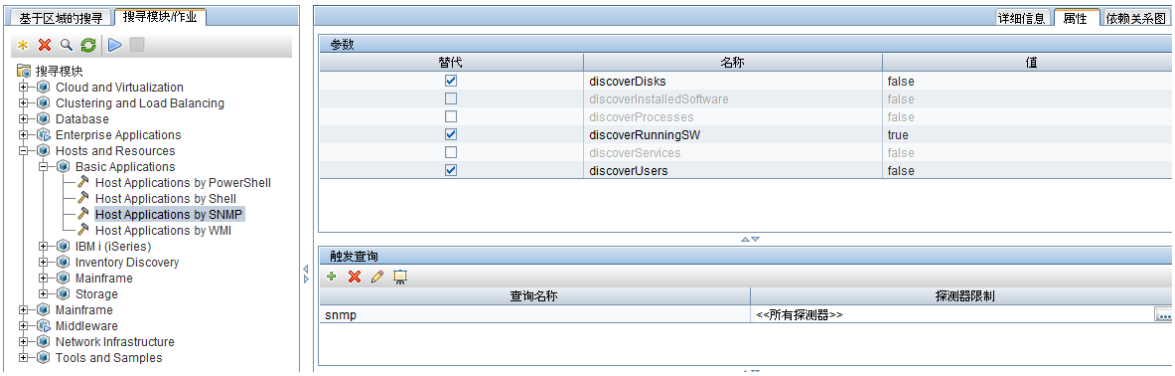
5. 配置远程进程的类路径 - 可选

   有关详细信息，请参阅配置远程进程执行 (第 32 页)。

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触发的 CI 集采用不同的方式计划同一个适配器，还可以为各个触发的 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，如下图所示。



选择触发器 TQL

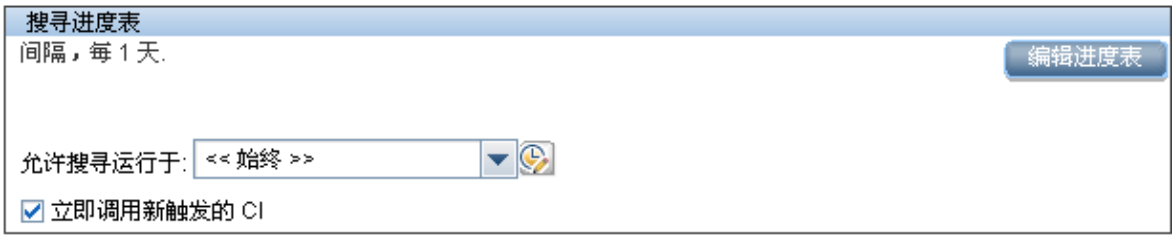
每个作业均与多个触发器 TQL 关联。这些触发器 TQL 发布的结果将用作此作业的适配器的输入触发 CI。

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。

备注: 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。

设置进度信息

探测器的进度信息可以指定在触发 CI 上运行代码的时间。选中“立即调用新触发 CI”复选框后，代码也会在传递到探测器时在各触发 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，探测器将在该作业累积的所有触发 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”。

替代适配器参数

配置作业时可以替代适配器参数。有关详细信息，请参阅替代适配器参数 (第 29 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

有关详细信息，请参阅创建 Jython 代码 (第 34 页)。

配置远程进程执行

可以在一个独立于“Data Flow Probe”的进程中对搜寻作业运行搜寻。

例如，如果作业使用的 **.jar** 库与探测器的库版本不同或者与探测器的库不兼容，则可以在单独的远程进程中运行作业。

另外，如果作业可能消耗大量内存（引入大量数据），您希望使探测器免受潜在 **OutOMemory** 问题的影响，也可以在独立的远程进程中运行作业。

要将作业配置为作为远程进程运行，请在作业的适配器配置文件中定义以下参数：

参数	描述
remoteJVMArgs	远程 Java 进程的 JVM 参数。
runInSeparateProcess	设置为“True”时，搜寻作业在单独进程中运行。
remoteJVMClasspath	（可选）启用远程进程的类路径自定义，并替代默认探测器类路径。如果探测器的 jar 与客户定义搜寻所需的自定义 jar 之间存在版本不兼容问题，这十分有用。 如果未定义 **remoteJVMClasspath** 参数或者将此参数保留为空，则使用默认的探测器类路径。 如果开发新搜寻任务，并且希望确保探测器 jar 库版本与作业的 jar 库不冲突，则必须至少使用执行基本搜寻所需的最小类路径。最小类路径在 **DataFlowProbe.properties** 文件的 **basic_discovery_minimal_classpath** 参数中定义。 **remoteJVMClasspath** 自定义示例： • 要将自定义 jar 填加到默认探测器类路径之前或之后，请按如下所示自定义 **remoteJVMClasspath** 参数： custom1.jar;%classpath%;custom2.jar - 在本例中，**custom1.jar** 放在默认探测器类路径之前，**custom2.jar** 附加到探测器类路径之后。 • 要使用最小类路径，请按如下所示自定义 **remoteJVMClasspath** 参数： custom1.jar;%minimal_classpath%;custom2.jar

第 2 章: 开发 Jython 适配器

本章包括:

- HP 数据流管理 API 参考 34
- 创建 Jython 代码 34
- 支持 Jython 适配器中的本地化 47
- 记录数据流管理代码 53
- Jython 库和实用程序 54

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:

<UCMDB 安装目录>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_JavaDoc\index.html

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数据流管理框架内实际编写 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。

备注:

- 为 Universal Discovery 编写的脚本应当与 Jython 2.5.3 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:

- 在 Jython 中使用外部 Java JAR 文件 (第 35 页)
- 执行代码 (第 35 页)
- 修改现成脚本 (第 35 页)
- Jython 文件结构 (第 36 页)
- Jython 脚本生成的结果 (第 38 页)

- 框架实例 (第 40 页)
- 查找正确的凭据（针对连接适配器）(第 43 页)
- 处理 Java 抛出的异常 (第 45 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 服务器的探测器。

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探测器。

探测器使用在该任务中指定的信息开始运行数据流管理代码。

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变更，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变更，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做的变更，因此您需要替换该行。但是，外部脚本不会被覆盖。

Jython 文件结构

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：

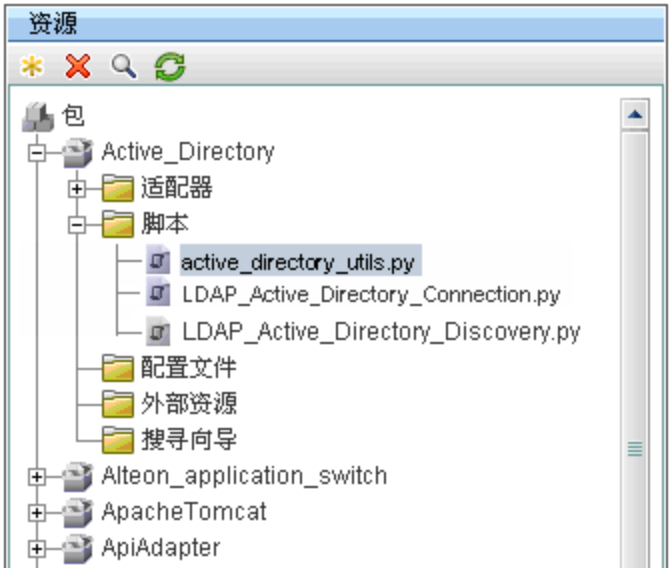
```
def DiscoveryMain(Framework):
```

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 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.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID))
except:
   Framework.reportError('Connection failed')
else:
   doQueryOSUsers(client, OSHVResult)
   client.close()
return OSHVResult
```

如果此脚本是相关到很多适配器的全局库，则可以在 jythonGlobalLibs.xml 配置文件中将此脚本添加到脚本列表，而不是添加到每个适配器（“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 模型中包含数据流管理结果。

关于在 CI 上设置属性的示例：

ObjectStateHolder 类描述了数据流管理结果图。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 的数据流管理框架中。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modeling.py (第 56 页)。

发送大量数据

在 UCMDB 中难以处理发送大量数据（通常大于 20 KB）这一操作。在发送到 UCMDB 之前，此大小的数据应拆分为较小的块。要使所有块能正确插入到 UCMDB，每块均需包含 CI 在块中所需的标识信息。这是部署 Jython 集成时的常见场景。sendObjects 方法可用于发送块中的结果。如果 Jython 脚本要发送大量结果（默认值为 20,000，但此值可以在“DataFlowProbe.properties 文件”中使用 **appilog.agent.local.maxTaskResultSize** 键进行配置），则它应该根据其拓扑对结果进行分块。执行这种分块时，应考虑标识规则，这样结果才能正确输入到 UCMDB 中。如果 Jython 脚本没有对结果进行分块，则探测器将尝试对其分块；但是，这可能会降低大型结果集的性能。

备注: 分块操作应用于 Jython 集成适配器，而不应用于常规的搜寻作业。这是因为，搜寻作业通常搜寻与特定触发器相关的信息，但是不发送大量信息。使用 Jython 集成可搜寻到单个集成触发器中的大量数据。

此外，还可以对少量结果使用分块操作。在这种情况下，不同块的 CI 之间存在一种关系，且 Jython 脚本的开发人员将拥有两个选项：

- 再次发送每个包含 CI 链接的块中的整个 CI 及其所有 CI 标识信息。
- 使用 CI 的 UCMDB ID。要执行此操作，Jython 脚本必须等到在 UCMDB 服务器中处理完每个块后才能获取 UCMDB ID。要启用此模式（称为“同步结果发送”），请将 SendJythonResultsSynchronously 标记添加到适配器。此标记可确保完成块的发送之后，探测器早已收到块中 CI 的 UCMDB ID。适配器开发人员可以使用 UCMDB ID 生成下一个块。要使用 UCMDB ID，请使用框架 API getIdMapping。

使用 getIdMapping 的示例

在第一个块中发送节点。在第二个块中发送进程。进程的根容器是一个节点。不要在进程 root_container 属性中发送节点的整个 objectStateHolder，而要使用 getIdMapping API 获取节点的 UCMDB ID，并仅使用进程 root_container 属性中的节点 ID 来减小块的大小。

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 34 页)。

框架实例的正确使用是将实例作为参数传递到使用该实例的每个方法。

示例：

```
def DiscoveryMain(Framework):
      OSHVResult = helperMethod (Framework)
        return OSHVResult
def helperMethod (Framework):
  ....
   probe_name    = Framework.getDestinationAttribute('probe_name')
   ...
   return result
```

本节描述了一些最重要的框架用法：

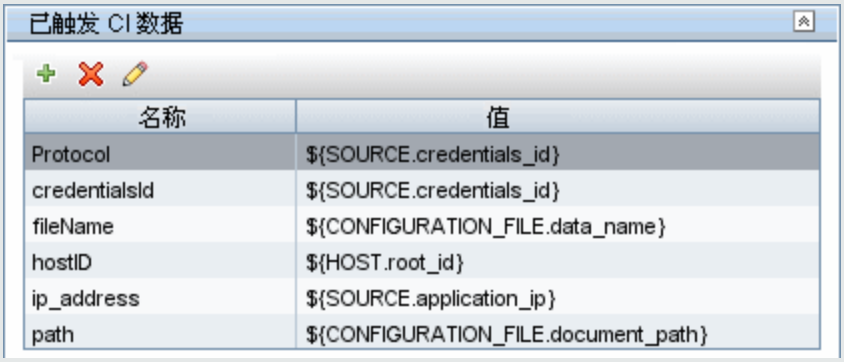
- Framework.getTriggerCIData(String attributeName) (第 40 页)
- Framework.createClient(credentialsId, props) (第 41 页)
- Framework.getParameter (String parameterName) (第 42 页)
- Framework.reportError(String message) 和 Framework.reportWarning(String message) (第 42 页)

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 UCMDB Discovery and Integrations Content Guide》。

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。

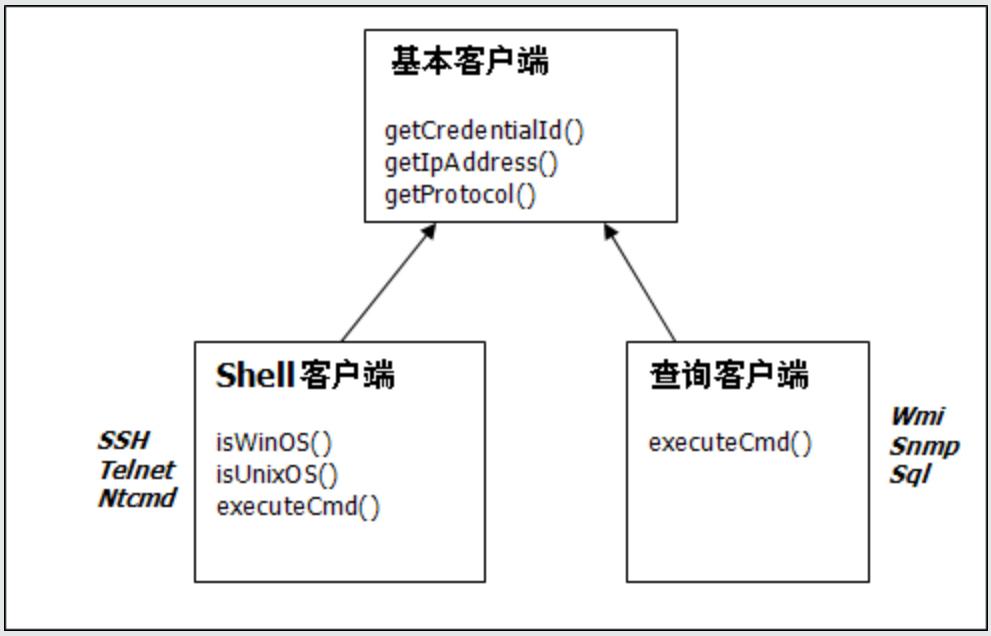
关于创建 WMI 客户端和运行 WMI 查询的示例:

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
                                    FROM   Win32_LogicalMemoryConfiguration")
```

注意: 要使用 createClient() API，请将以下参数添加到“触发的 CI 数据”窗格的触发 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分层以及它们通常支持的 API:



有关客户端及其受支持的 API 的详细信息，请参阅 DFM Framework中的 BaseClient、ShellClient 和 QueryClient。这些文件位于以下文件夹中:

<UCMDB 根目录>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_Schema\webframe.html

Framework.getParameter (String parameterName)

除了检索有关触发 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.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID))

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，探测器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。如果探测器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 57 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 Framework.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 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。

备注: 使用以下协议类型时将阻止访问敏感数据（密码、私钥等）：

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

使用上述协议类型应通过使用专用客户端来完成。

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 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 21 页)中的“两个阶段的连接范例”条目。

示例

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

# Java imports
from java.util import Properties
from com.hp.ucmdb.discovery.common import CollectorsConstants
from appilog.common.system.types.vectors import ObjectStateHolderVector
```

```
from com.hp.ucmdb.discovery.library.clients import ClientsConsts
from com.hp.ucmdb.discovery.library.scope import DomainScopeManager

TRUE = 1
FALSE = 0

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

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

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

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

```
                    isMultiOid = client.supportMultiOid()
                    logger.debug('snmp server isMultiOid state=%s' %isMultiOid)

            client.close()
            client = None

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

    return (_vector, error)
```

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。

备注: 由于基异常类在 Python 中显示时将使用相同的名称，因此按下例所示导入 Java 基异常类十分重要。

```
from java.lang import Exception as JException
try:
    client = Framework.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID))
except JException, ex:
    # process java exceptions only
    Framework.reportError('Connection failed')
    logger.debugException(str(ex))
     return
```

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。

Jython 版本 2.1 至 2.5.3 的迁移疑难解答

Universal Discovery 目前使用 Jython 版本 2.5.3。所有现成脚本均已正确迁移。如果您在 Discovery 使用此升级之前已开发了自身的 Jython 脚本，则可能会遇到以下问题，并且必须按照指示进行修复。

备注: 您必须是经验丰富的 Jython 开发人员才能进行这些变更。

字符串格式

- **错误消息：** TypeError:int argument required
- **可能原因：在包含整型数据的字符串变量中，将字符串格式用于十进制整数。**
- **存在问题的 Jython 2.1 代码：**

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

- **正确的 Jython 2.5.3 代码：**

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

  或

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

检查字符串类型

如果输入中包含 Unicode 字符串，则以下代码可能不会正常运行：

- **存在问题的 Jython 2.1 代码：** isinstance(unicodeStringVariable,'')
- **正确的 Jython 2.5.3 代码：** isinstance(unicodeStringVariable,basestring)
  应使用 basestring **进行对比，以便测试对象是** str **的实例，还是** unicode **的实例。**

文件中的非 ASCII 字符

- **错误消息：**
  SyntaxError:Non-ASCII character in file 'x', , but no encoding declared; see
  http://www.python.org/peps/pep-0263.html for details
- **正确的 Jython 2.5.3 代码：（将此代码添加到文件中的第一行）**

  ```
  # coding:utf-8
  ```

导入子包

- **错误消息：**
  AttributeError:'module' object has no attribute 'sub_package_name'
- **可能原因：在** import **语句中没有明确指定子包的名称也可导入子包。**
- **存在问题的 Jython 2.1 代码：**

  ```
  import a
  print dir(a.b)
  ```

  子包未显式导入。
- **正确的 Jython 2.5.3 代码：**

  ```
  import a.b
  ```

  或

  ```
  from a import b
  ```

迭代器变更

从 Jython 2.2 开始，__iter__ 方法可用于在 for-in 块范围的集合内进行循环。迭代器应实现 next 方法，并返回相应的元素或在到达集合结尾处时引发 StopIteration 错误。如果未实现 __iter__ 方法，则会改用 getitem 方法。

引发异常

- **引发异常的 Jython 2.1 方法已过时：**
  raise Exception, 'Failed getting contents of file'
- **引发异常后建议使用的 Jython 2.5.3 方法：**
  raise Exception('Failed getting contents of file')

支持 Jython 适配器中的本地化

多语言环境功能使得数据流管理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。

本节包括：

- 添加新的语言支持 (第 47 页)
- 更改默认语言 (第 48 页)
- 确定用于编码的字符集 (第 48 页)
- 定义使用本地化数据运行的新作业 (第 49 页)
- 解码命令，但不使用关键字 (第 50 页)
- 使用资源捆绑包 (第 50 页)
- API 参考 (第 51 页)

添加新的语言支持

此任务描述如何添加新的语言支持。

此任务包括以下步骤：

- 添加资源捆绑包（*.properties 文件）(第 47 页)
- 声明并注册语言对象 (第 48 页)

1. 添加资源捆绑包（*.properties 文件）

   根据要运行的作业添加资源包。下表列出了数据流管理作业和每个作业使用的资源捆绑包：

作业	资源捆绑包的基本名称
File Monitor by Shell	langFileMonitoring
Host Resources and Applications by Shell	langHost_Resources_By_TTY, langTCP

作业	资源捆绑包的基本名称
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，请参阅使用资源捆绑包 (第 50 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 51 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 OS 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 47 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 OS 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 OS，**ipconfig** 命令输出包含 **IP-Address**；对于俄语 OS，包含 IP-Адрес；对于德语 OS，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

此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 289 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，数据流管理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** 中检索相应的资源捆绑包：

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

4. 从资源捆绑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 object，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 49 页)中关于从资源捆绑包检索关键字的步骤。

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。

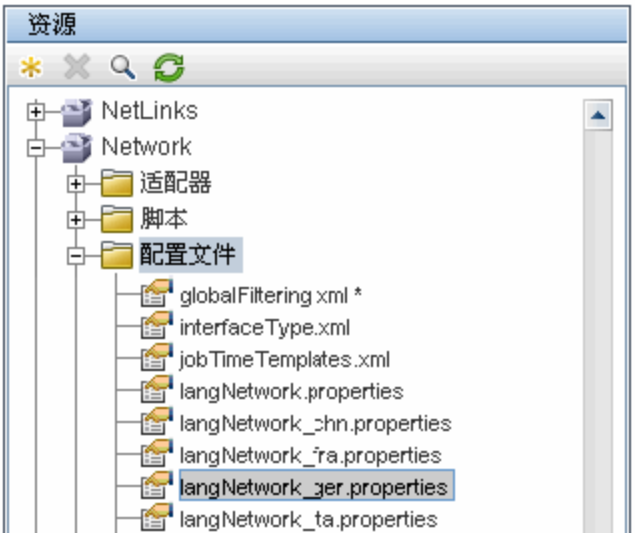
常规用例如下：

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。此时将使用在上一步中指定的字符集返回输出。无需执行其他解码操作。

使用资源捆绑包

资源捆绑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。资源捆绑包的主要功能是按照相应的键返回值。

资源捆绑包位于探测器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 服务器下载资源捆绑包。您还可以“资源”窗口中编辑、添加或删除资源捆绑包。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，数据流管理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捆绑包中。每种语言都有一个资源捆绑包。（尽管一个资源捆

绑包可以包含针对多种不同语言的数据，但在数据流管理中，一个资源捆绑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捆绑包，并通过特定键获取特定于语言的所有数据。

在数据流管理中，资源捆绑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。（默认资源捆绑包采用以下格式：<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** 表示此资源捆绑包包含由 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 51 页)
- executeCommandAndDecode 方法 (第 52 页)
- getCharsetName 方法 (第 52 页)
- useCharset 方法 (第 52 页)
- getLanguageBundle 方法 (第 52 页)
- osLanguage 字段 (第 53 页)

语言类

此类封装了有关语言的信息，例如资源捆绑包后缀、可能的编码等等。

字段

名称	描述
locale	表示语言环境的 Java 对象。
bundlePostfix	资源捆绑包后缀。此后缀在资源捆绑包文件名中使用，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 OS 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 OS 语言的方法之一是查询 OS 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 OS 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	传递给数据流管理中每个可执行的 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 52 页)。

getLanguageBundle 方法

可以使用此方法获取正确的资源捆绑包。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。应在此处传递 ShellUtils.osLanguage。

名称	描述
framework	传递给数据流管理中每个可执行的 Jython 脚本的常用框架对象。

osLanguage 字段

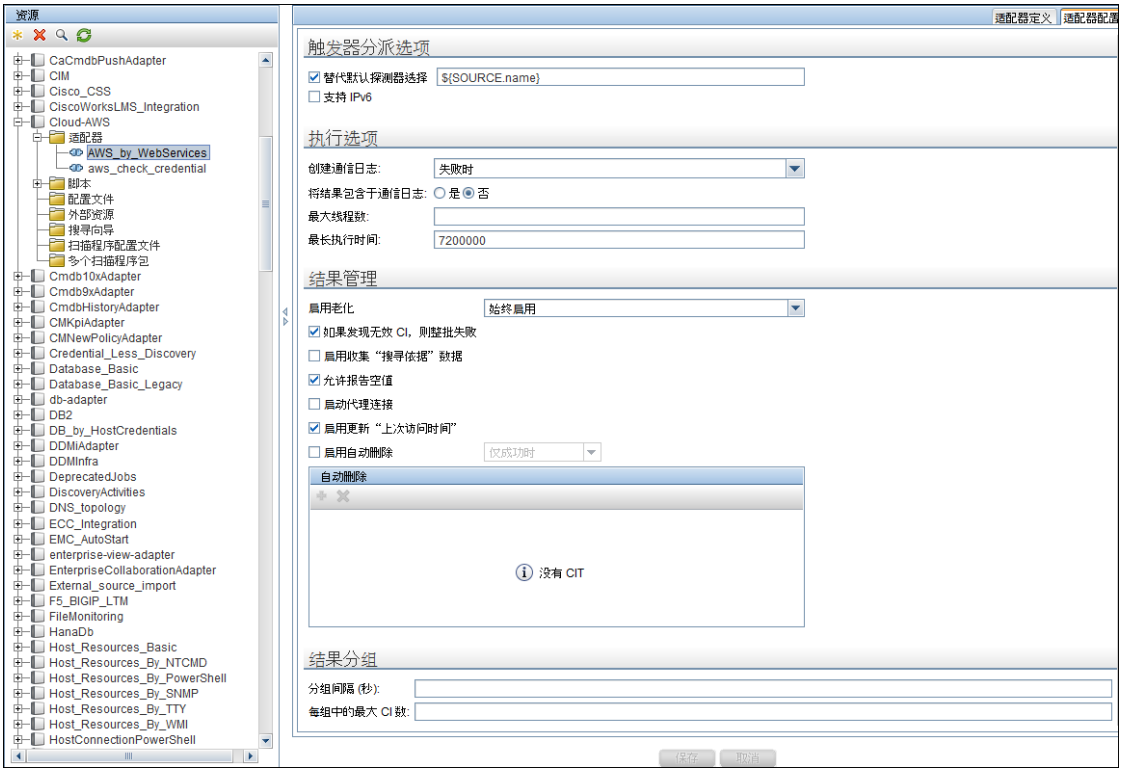
此字段包含一个表示语言的对象。

记录数据流管理代码

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）。此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。

要记录数据流管理代码，请执行以下操作：

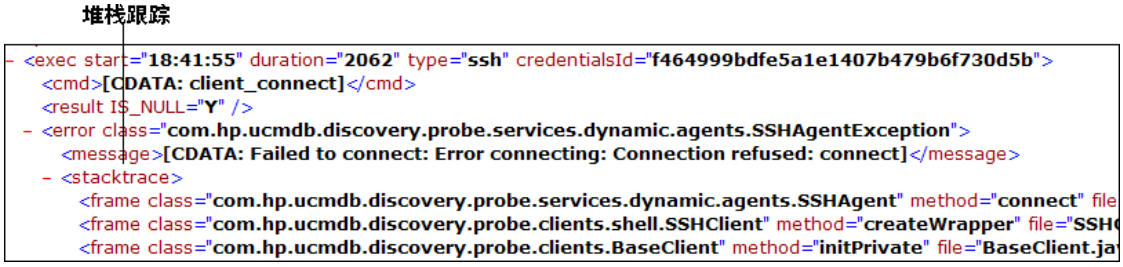
1. **访问“数据流管理”>“Universal Discovery”。右键单击必须记录其运行过程的作业，然后选择“转到适配器”，以打开适配器管理应用程序。**
2. **在“适配器配置”选项卡中找到“执行选项”窗格，如下所示。**



3. **将“创建通信日志”框更改为“始终”。有关设置日志记录选项的详细信息，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:



Jython 库和实用程序

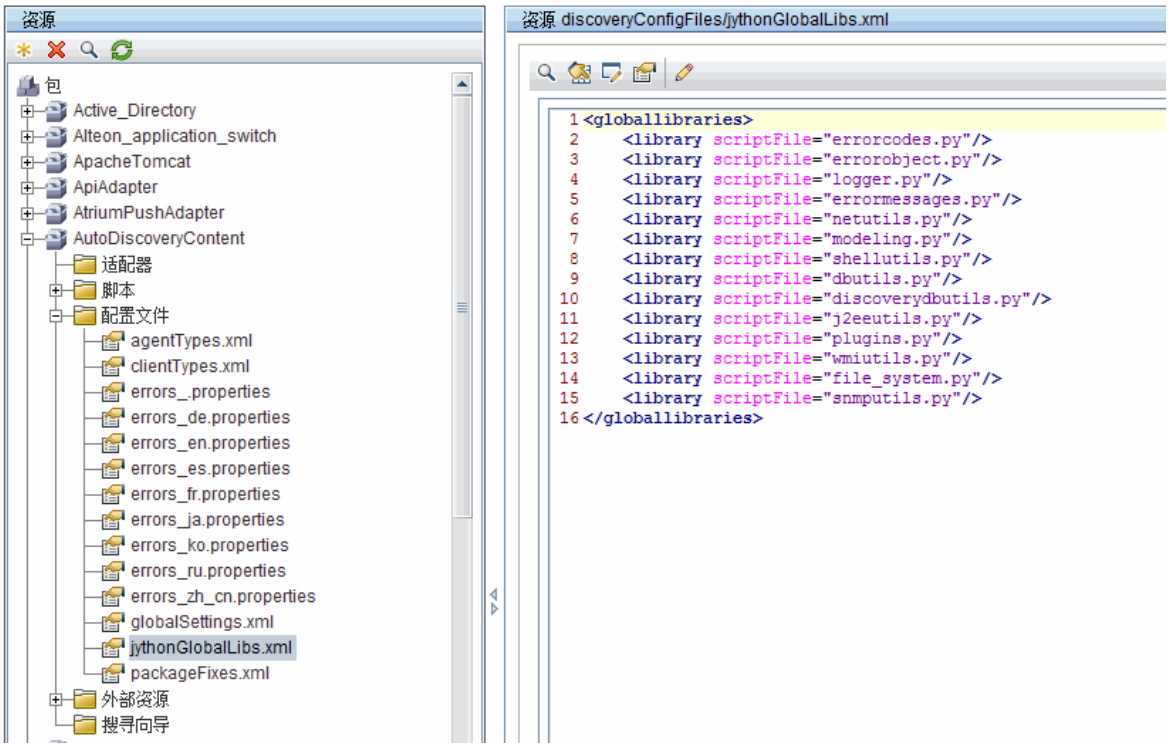
某些实用程序脚本在适配器中使用得很广泛。这些脚本包含在 AutoDiscovery 包中，它们与下载到探测器中的其他脚本一起位于以下位置：**C:\hp\UCMDB\DataFlowProbe\runtime\probeManager\discoveryScripts**。

备注: discoveryScript 文件夹是在探测器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。这些库脚本被视为数据流管理的外部库。可以在 jythonGlobalLibs.xml 文件（位于“配置文件”文件夹下）中定义这些脚本。



在探测器启动时，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 (第 55 页)
- modeling.py (第 56 页)
- netutils.py (第 56 页)
- shellutils.py (第 56 页)

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 59 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/probeMgr-
patternsDebug.log
```

```
log4j.appender.PATTERNS_DEBUG.Append=true
log4j.appender.PATTERNS_DEBUG.MaxFileSize=15MB
log4j.appender.PATTERNS_DEBUG.Threshold=DEBUG
log4j.appender.PATTERNS_DEBUG.MaxBackupIndex=10
log4j.appender.PATTERNS_DEBUG.layout=org.apache.log4j.PatternLayout
log4j.appender.PATTERNS_DEBUG.layout.ConversionPattern=<%d> [%-5p] [%t] - %m%n
log4j.appender.PATTERNS_DEBUG.encoding=UTF-8
```

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 = Framework.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_ID), Props)
clientShUtils = shellutils.ShellUtils(ttyClient)
if (clientShUtils.isWinOs()):
    logger.debug ('discovering Windows..')
```

第 3 章: 错误消息

本章包括:

- 错误消息概述 57
- 错误编写约定 57
- 错误严重度级别 59

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。每当常规搜寻流失败时，这些错误就会显示在“Universal Discovery”窗口中。您可以从引发问题的触发 CI 向下搜索，以查看错误消息本身。

数据流管理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，数据流管理也只会报告一次，但是数据流管理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 57 页)中所述。

数据流管理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“搜寻进度和结果”。

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”。

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 (第 58 页)
- 错误消息属性文件 (第 58 页)
- 语言环境命名约定 (第 58 页)
- 错误消息代码 (第 58 页)
- 未分类的内容错误 (第 59 页)
- 框架中的变更 (第 59 页)

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替代旧消息代码。

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
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

错误名称	错误代码	描述
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。

框架中的变更

(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 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（未发现探测器、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

此级别报告导致数据流管理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，数据流管理会将严重度标记为“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

第 4 章: 映射使用者-提供程序依赖关系

本章包括:

- 依赖关系搜寻概述 61
- 依赖关系签名文件 62
- 依赖关系搜索适配器 88
- 完整示例 96

依赖关系搜寻概述

依赖关系映射提供了一种灵活的方法来搜寻可部署组件或运行软件之间的关系。此方法允许使用用户定义的依赖关系映射规则（使用简单的编程语法），而 Universal Discovery 进程将使用这些规则自动搜寻依赖关系。

服务可以是业务服务，也可以是 IT 服务。业务服务是企业为其他企业提供的服务 (B2B)，或一个组织为企业中的另一个组织（如付款处理）提供的服务。IT 服务是 IT 组织为支持业务服务或 IT 自身的运营而提供的业务服务。

可部署组件是在运行软件中部署的软件组件，如应用程序服务器或 Web 服务器。可部署组件的示例为 JEE EAR 组件或 Oracle 数据库中的架构。为搜寻依赖关系，运行软件将被视为可部署组件。

“提供程序”可部署组件可提供服务并声明其他可部署组件如何使用该服务。“使用者”可部署组件“使用”提供程序可部署组件所提供的服务。这些可部署组件之间的依赖关系是“使用者-提供程序”依赖关系。

备注:

- 为了能够创建使用者-提供程序依赖关系适配器和使用依赖关系映射框架，必须在安装期间设置 UCMDB 架构时选择“启用搜索”选项。
- 只能以联合模式在 Data Flow Probe 上执行使用者-提供程序依赖关系适配器。

有关详细信息，请参阅:

- 提供程序和使用者 (第 61 页)
- 依赖关系签名 (第 62 页)
- 依赖关系映射流 (第 62 页)

提供程序和使用者

使用连接字符串连接到提供程序。例如，如果 Oracle 数据库是提供程序，则要连接到其服务，您可能需要:

- 计算机的 IP 地址
- SID

- TCP 端口

以上三种信息构成了使用者所需的连接字符串，而连接到该提供程序提供的服务需要这些连接字符串。例如，Oracle 连接字符串可能包含以下信息：

- IP 地址：1.1.1.1、2.2.2.2
- 端口： 1521
- SID：abcd

使用者知道提供程序的至少一个连接字符串，且此连接字符串在已知位置找到，例如配置文档、数据库表、Windows 注册表等。通过在这些位置搜索，可以搜寻使用者和提供程序之间的依赖关系。

如果在某个配置文档中找到提供程序的连接字符串，则配置文档的提供程序和容器将通过使用者-提供程序关系连接。

这样搜寻使用者-提供程序依赖关系的过程将变得非常简单：在使用者的配置文档中搜索提供程序的连接字符串，搜索结果包含指定提供程序的使用者所拥有的所有配置文档。

有关详细信息，请参阅定义依赖关系 (第 68 页)。

依赖关系签名

每个配置文档和提供程序类型可以使用不同的搜索术语。这些搜索术语在依赖关系签名文件中定义。

依赖关系签名是使用提供程序的连接字符串和使用者的配置文档，定义给定提供程序和给定使用者之间是否存在使用者-提供程序依赖关系的规则。

依赖关系签名由搜索表达式组成。这些搜索表达式依赖于连接字符串的定义，而非特定提供程序的实际值。此外，搜索表达式还依赖于使用者的配置文档名称、位置和格式，而不是文件的实际内容。如果提供程序的连接字符串已知，则这些字符串会注入搜索表达式，生成具体的搜索表达式。然后使用使用者的配置文档对具体的搜索表达式进行计算。只有当提供程序的连接字符串在使用者的配置文档中以某种特定形式存在时，搜索表达式才返回“True”。

有关详细信息，请参阅依赖关系签名文件 (第 62 页)。

依赖关系映射流

本节简要概述了依赖关系映射期间出现的基本流：

1. 将搜寻可部署组件及其连接字符串。
2. 每种类型的提供程序可部署组件将触发特定的依赖关系映射作业。每个作业的适配器了解如何为特定的可部署组件类型提取相关连接字符串。适配器将搜索使用该服务的其他可部署组件。
3. 将在找到的每个可部署组件及其触发器（提供程序）之间创建使用者-提供程序关系。

依赖关系签名文件

本节包括：

- 依赖关系签名文件的结构 63
- 打包和部署多个依赖关系签名文件 87
- 编译错误 87

依赖关系签名文件的结构

依赖关系签名文件定义可部署组件之间的一个或多个依赖关系。

依赖关系的使用者部分定义为 <Deployable> 元素。每个 <Deployable> 元素可能包含一个或多个 <Dependency> 元素。每个 <Dependency> 元素包含使用者（ <Deployable> 元素 ）和提供程序之间存在的依赖关系的条件。

可将 <Dependency> 元素视为布尔函数，作为提供程序 CI 和使用者配置文档的输入，并且仅在这两个 CI 之间存在依赖关系（从使用者到提供程序）时返回“True”。

在 <Dependency> 元素中，仅使用其 CI 类型在文件中标识提供程序。每个依赖关系可以用于提供程序的一个 CI 类型。以下示例定义任何“运行软件”使用者与“运行软件”提供程序之间的依赖关系函数。

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

变量和概念

在执行程序期间，变量可能包含不同的值。对于依赖关系映射，变量可能包含不同的值，以执行不同的依赖关系搜索。定义搜索表达式时，将使用这些变量确定配置文档中是否存在连接字符串。由于连接字符串随提供程序的不同而异，因此变量允许您定义通用搜索表达式，而不管连接字符串的特定值如何。

变量

备注: 术语“变量”表示变量和概念变量。

使用变量的语法为 ${VARIABLE_NAME}。每个变量值必须是字符串值或字符串列表。例如，部分连接字符串可能为提供程序的 IP 地址（或地址）。要在配置文档中搜索 IP 地址，可以定义名为 IP_ADDRESS 的变量，并按以下方式在表达式中使用它：${IP_ADDRESS}。

必须先在整个依赖关系签名文件范围（称为全局范围）或依赖关系范围（称为局部范围）内定义变量。变量定义允许您使用此变量。

使用未定义的变量将导致在部署签名文件时生成错误。

全局范围

全局范围变量可以由定义它的文件中的任何依赖关系使用。定义全局范围变量:

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

以上语法定义了两个全局范围变量：IPADDRESS 和 PROTOCOL。与所有变量一样，它们可以保留任何字符串值或字符串列表。

全局变量的值只能由触发的目标数据设置。

局部范围

局部范围变量只能由定义该变量所针对的依赖关系使用。它对任何其他依赖关系均不可见，不管是否针对相同的可部署组件。

可以在不同的依赖关系中定义具有相同名称的多个局部范围变量。每个变量是完全独立的变量，其值只能在定义它的范围中使用。

备注: 不能定义与全局范围变量同名的局部范围变量，也不能在同一依赖关系上下文中定义具有相同名称的两个局部范围变量。

要定义局部范围变量，请使用以下语法：

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

只能通过使用注入语句来为局部范围变量分配值。根据从其提取值的文件类型，存在不同的特定注入语句。注入语句可以出现在配置文档的两个位置。

- 在专用 <Variables> 部分中。如果整个文件的搜索表达式计算为 True，则将仅计算 <Variables> 部分中的注入语句。
- 在作为搜索条件一部分的 <Variables> 部分中。使用此选项，将仅在搜索条件计算为 True 时才注入变量。使用备用表达式时不支持此选项。有关详细信息，请参阅使用变量的默认值 (第 70 页)。

有关分配变量的详细信息，请参阅：

- 属性配置文档 (第 76 页)
- XML 配置文档 (第 79 页)
- 文本配置文档 (第 82 页)

概念

针对每个相关的配置文档应用搜索表达式。连接字符串中的某些值彼此紧密耦合，且应仅在具体的搜索表达式中彼此结合使用。每组这样的耦合连接字符串必须具有唯一名称，这称为概念。

例如，可以从 1.1.1.1:8080 或 2.2.2.2:85 访问提供程序可部署组件。由此可见，提供程序的任何使用者必须在其某个配置文件中保留 1.1.1.1:8080 或 2.2.2.2:85。但是，在此提供程序的任何使用者的

配置文档中不可能找到 1.1.1.1:85，因为此提供程序不侦听 1.1.1.1:85。即使在某个使用者的其中一个配置文档中找到了 1.1.1.1:85，也不表示与此提供程序存在依赖关系，而是与某些其他可部署组件（与提供程序在相同的节点上运行且侦听 1.1.1.1:85）存在依赖关系。

搜索必须查找 IP 地址和端口的正确匹配项，否则搜索将返回误报的依赖关系。

此外，搜索还必须查找每个 IP 地址名称和端口的正确匹配项，因为每个 IP 地址可能具有多个名称（例如，一个权威 DNS 名称和多个别名）。

因此，很可能在该提供程序的使用者配置文档中找到以下匹配项或匹配对：XYZ:8080、FOO:8080 或 ABC:85（其中 XYZ、FOO 和 ABC 是提供程序地址的其他名称），这些匹配项表示与该提供程序存在依赖关系。但是，匹配项 ABC:8080 并不表示与该提供程序存在依赖关系，因此不应该搜索到该匹配项。

为此，将使用概念。应在同一概念中定义与其他连接字符串属性结合使用的每个连接字符串属性。每个连接字符串属性在该概念中应该为变量。

只能针对全局范围定义概念。与变量一样，必须由适配器设置每个概念实例（表示一组紧密耦合的连接字符串）的值。

每个概念由正好一个关键变量和附加变量组成。每个这种变量（关键变量和附加变量）都用于存储紧密耦合的连接字符串。关键变量用于区分同一概念的不同实例。这意味着同一概念的两个概念实例的关键变量的值不可能相同。有关详细信息，请参阅指定概念变量值 (第 92 页)。

备注: 关键变量可以像任何其他概念变量一样使用。其重要性在于触发器执行期间实例化概念实例的方式。

要定义概念及其变量，请使用以下语法：

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

要在搜索表达式中使用概念的变量，请使用以下语法：

```
${ConceptName.VariableName}。
```

对于搜索表达式中使用的每个概念，都必须存在一个包含该概念的所有相关变量且不包含其他概念变量的逻辑运算符。

以下是包含概念的有效搜索表达式示例：

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

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

下面是无效搜索表达式的示例：

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

默认值

全局变量和概念变量可以具有默认值（可选）。如果变量从值与默认值相同的适配器注入，则可以定义备用搜索表达式。有关备用搜索表达式的详细信息，请参阅编写搜索表达式 (第 68 页)**。**

使用以下语法定义默认值:

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

IP 地址变量类型

IP 地址是一种重要的连接字符串；但是，配置文件中并非始终指明提供程序地址。

当使用者和提供程序位于同一台主机上时，这种情况很常见。在这种情况下，可能会显示“localhost”或“127.0.0.1”等字符串，而非提供程序地址，或者完全不显示任何地址。

通过使用类型 IP 地址标记变量，框架将忽略 IP 地址变量并查找其余连接字符串位于同一主机的提供程序，自动尝试查找局部依赖关系。

将变量标记为 IP 地址

- **对于全局变量**

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

- **对于概念变量**

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

备注: 局部变量不能具有 IP 地址类型。

定义使用者的描述符

要定义使用者-提供程序依赖关系，搜索适配器将查找具有使用提供程序连接字符串的配置文档的使用者。但是，有时限制可能是特定服务使用者的可部署组件非常有意义，即使该使用者的配置文档包含连

接字符串或具有不同的输出变量（取决于可部署组件的属性）时也是如此。

为此，可以使用 <Descriptor> 元素更详细地描述可部署组件，而不仅仅是其 CI 类型。可使用以下对象描述可部署组件：

- CI 类型，即依赖关系仅与“J2EE Application”类型的可部署组件相关。（必需）
- 其字符串属性的条件；例如，应用程序名称为“MyShop”的“J2EE Application”类型的可部署组件。条件只能测试相等性。（可选）
- 指向其他 CI 的必需组合链接。如果在描述符中指定了组合链接，则可部署组件必须具有指向给定类型 CI 的组合链接。（可选）
- 已连接 CI（如果在上一个选项中指定）的字符串属性的条件。（可选）
- 包含可部署组件的节点的路径（TQL 查询）。使用属性 **nodeToDeployableQuery** 指定查询名称。如果使用组合链接将节点直接连接到可部署组件，则无需提及 TQL 查询作为路径。但是，如果节点仍然用于描述可部署组件（带或不带条件），则必须仍然使用 <ConnectedCICondition> 标记提及这一点。如果可部署组件在其父分层中的任何位置都不存在包含节点，请使用属性 hasContainingNode = “false” 提及这一点。提及路径时，务必还要添加 hasContainingNode = “true”。有关详细信息，请参阅定义 TQL 查询 (第 86 页)。（可选）

备注: 仅可部署描述符或已连接 CI 支持 STRING 属性类型。属性不能为静态，也不能包含计算的属性。

仅提及 CI 类型的描述符示例：

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

具有字符串属性的描述符示例：

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

具有已连接 CI（测试不区分大小写的相等性）的描述符示例：

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

备注: 对于 ConnectedCiCondition，您只能使用 linkType=” composition” 和 isDirectionForward=” true”。

定义依赖关系

对于每个使用者，可以定义多个依赖关系。每个依赖关系都与特定的提供程序 CI 类型关联。在为提供程序 CI 计算依赖关系签名期间，将计算与该提供程序 CI 类型关联的所有依赖关系。每个依赖关系都具有来自一个或多个配置文档的搜索表达式。如果搜索表达式计算为 True，则使用者和提供程序之间存在依赖关系（使用者-提供程序关系）。即使同一使用者和提供程序 CI 类型之间存在多个依赖关系且都计算为 True，也只会创建一个关系。

依赖关系语法示例如下:

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

每个依赖关系都还有一个范围。范围是所有使用者中遵循描述符并与此特定依赖关系相关的使用者。有关详细信息，请参阅定义使用者的描述符 (第 66 页)。

编写搜索表达式

备注: 仅可部署描述符或已连接 CI 支持 STRING 属性类型。属性不能为静态，也不能包含计算的属性。

搜索表达式由逻辑运算符和条件组成。支持的逻辑运算符为“And”和“Or”。

条件是使用提供程序的连接字符串和使用者的配置文档计算的表达式。条件根据文件类型的不同而异，即属性配置文档的条件不同于 XML 文档。

下面是属性配置文档的搜索表达式示例：只有当提供程序的 IP 地址存在于使用者的 **MyConfig.properties** 配置文档的键 IPADDRESS 下时，才创建提供程序和使用者之间的依赖关系。将在依赖关系配置文档中编写这种条件，如下所示:

```
<PropertiesConfigurationDocument name="MyConfig.properties"> (Type and name of the
configuration document)
  <Condition> (Beginning of the search expression)
    <Operator type="and"> (Operator)
      <KeyCondition key="IPADDRESS"> (Start a condition and state its type)
        <Values>
          <Value>${IP_ADDRESS}</Value> (Use a variable stating the provider’ s IP address)
        </Values>
      </KeyCondition>
    </Operator>
  </Condition>
</PropertiesConfigurationDocument>
```

备注: <Condition> 元素中必须至少有一个 <Operator> 元素，即使像以上示例中只有一个搜索条件一样，在逻辑上并不必要。

下面是更复杂的搜索表达式：提供程序的 IP 地址存在于使用者的 **MyConfig.properties** 配置文档的键 IPADDRESS 下，或提供程序的主机名存在于同一文件的键 HOST 下：

```
<PropertiesConfigurationDocument name="MyConfig.properties"> (Type and Name of the
configuration document)
  <Condition> (Beginning of the search expression)
    <Operator type="or"> (Operator)
      <KeyCondition key="IPADDRESS"> (Start a condition and state its type)
        <Values>
          <Value>${IP_ADDRESS}</Value> (Use a variable stating the provider’ s IP address)
        </Values>
      </KeyCondition>
      <KeyCondition key="HOST"> (Start another condition, under the same operator)
        <Values>
          <Value>${HOSTNAME}</Value> (Use a variable stating the provider’ s host name)
        </Values>
      </KeyCondition>
    </Operator>
  </Condition>
</PropertiesConfigurationDocument>
```

可以嵌入逻辑运算符以创建类似 (C1 AND (C2 OR C3)) 的条件，XML 将如下所示：

```
<PropertiesConfigurationDocument name="MyConfig.properties"> (Type and Name of the
configuration document)
  <Condition>
    <Operator type="and">
      C1
      <Operator type=” or” >
        C2
        C3
      </Operator>
    </Operator>
  </Condition>
</PropertiesConfigurationDocument>
```

其中 C1、C2、C3 是所需搜索条件的 XML 片段。

支持以下三种文件类型，每种都具有自己的搜索表达式类型：

- PropertiesConfigurationDocument – 每行的格式为 key=value 的文件。有关详细信息，请参阅属性配置文档 (第 76 页)。

  KeyCondition – 给定键值的条件。
- XmlConfigurationDocument – XML 文件。有关详细信息，请参阅XML 配置文档 (第 79 页)。

  XPathCondition – 计算 XPath 查询
- TextConfigurationDocument – 任何文本配置文档。有关详细信息，请参阅文本配置文档 (第 82 页)。

  RegExp – 计算正则表达式

有关默认值的详细信息，请参阅使用变量的默认值 (第 70 页)。

疑难解答

- **条件树的同一节点中不允许存在不同的概念。同样，同一运算符下也不允许存在两个具有不同概念的叶。**

  例如，以下条件定义无效:

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

使用变量的默认值

在某些情况下，如果特定连接字符串的值等于某个默认值，则配置文件将不包含该值。例如，定义 HTTP URL 时，URL 中可能不会明确提及端口值 80。这意味着配置文件更可能包含 http://www.hp.com/，而非 http://www.hp.com:80/，即使这两个 URL 都正确。这可能会导致在编写搜索条件时出现问题，因为如果 PORT 变量包含值 80 且条件要求存在此值，则在未指定此值时，测试 URL 将失败。

要解决此问题，每个变量可以包含默认值（可选）。如果变量的值与其默认值相同，则可以修改搜索表达式。

修改搜索表达式的方式有两种:

- **如果变量的值与其默认值相同，则忽略条件**

  如果搜索条件中的一个或多个变量的值与其默认值相同，则使用此选项可完全忽略条件。如果忽略条件，则不会返回 True 或 False，忽略后逻辑运算符的结果取决于运算符和其他操作数。例如:

  - **如果条件的形式为 A AND B，其中 A 和 B 是其他表达式，如果忽略 A，则结果将与 B 的结果相同。**
  - **对于表达式A AND B AND C，如果忽略 A，则结果将与 B AND C 相同。**
  - **对于表达式A OR B OR C，如果忽略 A，则结果将与 B OR C 相同。**

  在少数情况下，会忽略所有子条件，这意味着忽略整个条件，配置文档条件的结果将为 False。

  要忽略条件，请将 ignoreIfDefaultValue 属性添加到条件，后跟将触发忽略此条件的变量列表。例如:

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

- **如果变量的值与其默认值相同，则使用其他条件**

  下面我们继续以 PORT 80 和 URL 为例。使用以下正则表达式测试 URL:

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

很显然，此正则表达式绝不会将字符串 http://www.hp.com/ 计算为 True，因为此字符串中不存在冒号 (:)。因此，我们还要使用以下正则表达式测试 PORT 变量是否具有值 **80**：

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

在这些类型的场景中，可以使用备用表达式。要定义备用表达式，请在同一条件下定义多个搜索表达式，并声明当变量为默认值时应使用哪个表达式。例如：

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

使用 **alternativeFor** 属性说明如果变量为默认值，则将对备用表达式而非原始表达式进行计算。原始表达式是没有 **alternativeFor** 属性或具有空 **alternativeFor** 属性的表达式。

对于出现在表达式中且具有默认值的所有变量，必须定义备用表达式以包含这些变量的任意组合。否则，您将收到编译错误。如果只有一个变量，则只需要一个备用表达式，如以上示例所示。如果有多个变量具有默认值，则需要多个备用表达式。

例如，如果 DOMAIN 变量也为默认值，则使用 **alternativeFor** 语句即表示具有以下所有备用条件：

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

运行时将仅计算其中一个组合。

有关指定变量的默认值的详细信息，请参阅默认值 (第 66 页)。

指定配置文档的路径

配置文档路径不是指主机文件系统上文件的路径，而是指使用者可部署文档与配置文件之间的拓扑路径。

默认情况下，如果未指定路径，则假设配置文档由组合链接连接到可部署文档。换句话说，假设可部署组件 CI 拥有配置文档 CI。

要指定其他路径，请执行以下操作：

1. 按定义 TQL 查询 (第 86 页)中的规定定义 TQL 查询。
2. 使用 <DocumentCILocation> 元素从配置文档引用 TQL 查询，如下所示：

```
<TextConfigurationDocument name="cmdb.properties">
```

```
  <DocumentCILocation>
    <ReferenceLocation>YourQueryName</ReferenceLocation>
  <DocumentCILocation>
  <Condition>
    ...
  </Condition>
</TextConfigurationDocument>
```

在此示例中，YourQueryName 是对 <Queries> 部分中查询名称的引用。

生成 TQL 查询以指定路径时：

- TQL 查询必须定义可部署组件与配置文档之间的路径。此路径必须简单（无循环）。
- TQL 查询必须包括以下两个端节点，具有特定名称且区分大小写：
  - Deployable - 路径中的可部署组件 CI
  - Configuration_document - 路径中指定配置文档的 CI
- Deployable 和 Configuration_document 节点不允许使用条件。
- 所有节点之间的基数必须为 1..1。
- 仅支持常规链接类型。不允许使用复合链接、联接或子图。

配置文档替代

在某些情况下，配置文档可以替代其他配置文档或被其他配置文档替代。例如，主机上的配置文档可以替代该主机所属的群集中的配置文档。在这种情况下，键值必须浏览所有可能替代这些值的文件，并选择具有最高优先级的文件。在此示例中，主机的本地配置文档的优先级高于群集文档。

要在依赖关系签名中定义文件替代，请使用以下语法将具有优先级的多个路径添加到配置文档：

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

所有引用位置的文件名（以上示例中为 **resources.xml**）必须相同。

请记住，引用位置是对指定从使用者可部署组件到配置文档的拓扑路径的 TQL 查询的引用。这也意味着该路径必须存在于可部署组件和所有其他位置之间。有关 <DocumentCILocation> 的详细信息，请参阅指定配置文档的路径 (第 71 页)。

添加具有优先级的多个路径时，条件自身不会发生更改。运行时（计算搜索表达式时），将使用取决于优先级的正确值。例如，在属性文件中，如果为键 K 赋予优先级 1 和优先级 2（其值具有条件），则仅对具有优先级 1 的文档计算该条件。如果键 K 仅存在于优先级 2 中，则仅对具有优先级 2 的文档计算该条件。

有关属性条件的详细信息，请参阅属性配置文档 (第 76 页)。

在 XML 文档中，每个 XPath 都被视为一个键。例如，\Root\Element\@Attribute 表示具有该路径的属性“Attribute”是一个键，可在不同的文件中替代它的值。

如果 XPath 还具有某些常量条件（不涉及变量的条件），则该条件将是键的一部分。例如：\Root\Element[@name = ‘name’]\@Attribute。

但是，如果 XPath 具有非常量条件，则这些条件将从键中剥离，并被视为值的一部分。所以对于路径 \Root\Element[@name = ${NAME}]\@Attribute，键将为路径 \Root\Element\@Attribute，并仅对存在该键的最高优先级文档计算条件 Element[@name = ‘name’]。

如果 XML 文档使用优先级，则务必确保按以上所述生成 XPath 条件，以免出现错误的意外行为。

有关 XPath 条件的详细信息，请参阅XML 配置文档 (第 79 页)。

备注: 只有属性和 XML 配置文档允许使用具有优先级的多个引用位置。文本文档不允许使用它们。

具有相同优先级的多个配置文档

可以为不同的 <ReferenceLocation> 元素赋予相同的优先级。在这种情况下，计算条件时，框架将查看具有相同优先级的所有文件并尝试匹配所有文件。

如果至少有规定数量的文件计算为 True，则条件将计算为 True。文件数由 **samePriorityMatchAtLeast** 属性定义。

例如:

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

这意味着计算条件时，至少有一个引用的文件位置是指向条件计算为 True 的文件。

备注: 当前，**samePriorityMatchAtLeast** 属性仅支持值 **1**。

在多个文档间定义的依赖关系

在很多情况下，确定使用者是否提供服务需要搜索多个配置文档，以查找提供程序的所有连接字符串。

搜索多个不相关的配置文档

配置文档可能彼此不相关，某些连接字符串显示在一个文档中，某些则显示在其他文档中。例如，使用者可能有两个配置文档：**A.conf** 和 **B.conf**。提供程序的连接字符串是 C1（在 **A.conf** 中定义）和 C2（在 **B.conf** 中定义）。确定使用者和提供程序之间是否确实存在依赖关系需要 C1 和 C2。在这种情况下，存在两个必需的搜索表达式，每个文档中一个。这样 <Dependency> 标记将具有以下结构:

```
<Dependency name="dependency_name" providerCiType="webmodule" scope="my_scope">
```

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

备注:

- <Dependency> 元素中可以显示的文件数没有限制。
- 每个文件的类型可能不同（属性、XML 或文本）。
- 将忽略 <Dependency> 元素中的文件顺序。将按任意顺序测试文件。
- 所有文件中的搜索表达式都必须返回 True，依赖关系才存在。
- 由于连接字符串分布在多个文件中，因此范围的搜索表达式必须够宽，以便至少可从该范围返回一个所需文件。有关详细信息，请参阅指定搜索范围 (第 84 页)。

要在搜索多个文件时对流求和，请执行以下操作：

1. 执行筛选范围，如指定搜索范围 (第 84 页)中所述。如果未返回任何所需文件，则表示使用者和提供程序之间不存在依赖关系。
2. 如果筛选器返回至少一个文件，则连接到该文件的使用者以及其余所需配置将被下载到 Data Flow Probe。
3. 以某种任意顺序计算每个文件中的条件。
4. 如果所有条件均返回 True，则表示依赖关系存在；否则表示不存在。

搜索多个存在依赖关系的配置文档

在其他情况下，使用者的配置文档彼此相关。例如，**DBConnections.conf** 文件定义多个数据库和架构的多个连接字符串，每个连接字符串都有一个名称。在 **MyApp.conf** 文件中，将使用其中一个连接名称，这将定义 MyApp 使用该特定数据库和构架。

具有依赖关系的配置文档使用局部范围变量。在其中一个配置文档中注入局部范围变量的值（使用注入语句），以便变量值（如连接字符串的名称）存在于该文档中。之后，该变量可在不同配置文档的条件中使用（例如，在 **MyApp.conf** 文件使用该连接名称以确保 MyApp 使用提供程序 (DB)）。有关详细信息，请参阅局部范围 (第 64 页)。

使用变量注入语句将值注入变量。每个文档类型使用不同的语句:

- 属性文件 - KeyVariable 和 KeyRegExpVariable。有关详细信息，请参阅属性配置文档 (第 76 页)。
- XML 文件 – XPathVariable。有关详细信息，请参阅XML 配置文档 (第 79 页)。
- 文本文件 – RegExpVariable。有关详细信息，请参阅文本配置文档 (第 82 页)。

在这种情况下，存在两个必需的搜索表达式，每个文件中一个。因此 <Dependency> 标记将具有以下结构:

```
<Dependency name="dependency_name" providerCiType="webmodule" scope="my_scope">
   <VariableDeclarations>
     <Variable name=" REFERENCE_NAME" />
   </VariableDeclarations>
  <PropertiesConfigurationDocument name="DBConnections.conf">
   <Condition>
     <Operator type="and">
       <KeyCondition key="C1">
         <Values>
           <Value>${VAR_1}</Value>
         </Values>
       </KeyCondition>
     </Operator>
   </Condition>
   <Variables>
     <KeyVariable variable=" REFERENCE_NAME" key=" reference_name" />
   </Variables>
  </PropertiesConfigurationDocument>
  <TextConfigurationDocument name="MyApp.conf">
   <Condition>
     <Operator type="and">
       <RegExpCondition>
         <RegExp>C2?${REFERENCE_NAME}</RegExp>
       </RegExpCondition>
     </Operator>
   </Condition>
  </TextConfigurationDocument>
</Dependency>
```

备注:

- 使用 <VariableDeclarations> 元素定义一个或多个要在依赖关系中使用的局部变量。
- <Variables> 元素用于将值注入局部变量。仅在 <Condition> 元素返回 True 时执行此部分。
- 在此示例中，<KeyVariable> 用于将“reference_name”键的值注入 REFERENCE_NAME 变量。

- ${REFERENCE_NAME} 变量用于 **MyApp.conf** 条件。此变量依赖于 **DBConnections.conf** 文件，且仅在 **DBConnections.conf** 后并且在搜索表达式计算为 True 时对其进行计算。
- 文件之间不允许存在循环依赖关系。
- 计算 **MyApp.conf** 文档时，${REFERENCE_NAME} 变量已包含 **DBConnections.conf** 中的值。
- 由于连接字符串分布在多个文件中，因此范围的搜索表达式必须够宽，以便可从该范围返回没有依赖关系的文件。有关详细信息，请参阅指定搜索范围 (第 84 页)。

要在搜索多个存在依赖关系的文件时对流求和，请执行以下操作：

1. 执行筛选范围，如指定搜索范围 (第 84 页)中所述。如果未返回任何所需文件或文件彼此不相关，则表示使用者和提供程序之间不存在依赖关系。
2. 筛选器返回至少一个不存在依赖关系的文件。
3. 连接到该文件的使用者以及其余所需配置将被下载到 Data Flow Probe。

例如，如果引用的键不存在，则注入语句可能不会返回值。在这种情况下，变量将获得空值。但是，这可能不是期望的行为。如果所需值不存在，则可能意味着依赖关系不存在，因此也就无需继续计算后续文件。对于这种情况，请在注入语句中使用 allowNull=”false”。例如：

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

属性配置文档

属性配置文档以 Key=value 格式存储配置。例如，属性配置文档可能为：

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

其中 Hostname 是键，MyNode 是其关联值。

“#”标记注释行。测试搜索表达式时将忽略注释行。

要声明配置文档是属性配置文档，请使用 <PropertiesConfigurationDocument> 元素。例如：

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

定义条件

可采用多种方式测试属性文件中是否存在连接字符串变量。对于所有条件类型，对照某些值测试在 **key** 属性中指定的键值。键始终是常量值（不允许使用变量）：

- 测试键值是否为常量值。

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

```
    <Value>ConstantValue2</Value>
  </Values>
</KeyCondition>
```

只有键的值等于（不区分大小写）其中一个常量值时，条件才返回 True。

- 测试键值是否为变量值。

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

只有键的值等于（不区分大小写）其中一个变量值时，条件才返回 True。如果变量值是值列表，则将测试所有值，且只要键与列表中的其中一个值匹配就会返回 True。

备注: 可以在相同条件下指定变量和常量值。

- 测试键值是否符合某些正则表达式。

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

正则表达式还可以包含一个或多个变量作为条件的一部分。运行期间，将在生成具体的搜索表达式时使用实际值替换变量。

如果变量包含值列表，则将生成类似于下面的表达式：

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

在这种情况下，任何一个值都可能为条件返回 True。

属性配置文档中的条件示例

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

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

这意味着名为 MyConfig.properties 的属性配置文档必须具有以下所有项:

- **名为 TYPE 的键包含值 HTTP 或 HTTPS 其中之一。**
- **名为 IPADDRESS 的键包含提供程序的 IP 地址（或其 IP 地址之一）或其主机名。**
- **名为 SITE 的键以“/”开头，后跟提供程序的 SITE_NAME、另一个“/”，然后是任意字符串。**

注入变量值

可采用两种方式将键的值或部分值提取到变量中:

- **通过注入键的值。**
  - **在专用 <Variables> 部分中，请使用以下语法:**

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

  - **作为搜索条件的一部分，请使用以下语法:**

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

    VARIABLE_NAME 变量的键在 <KeyCondition> 元素中定义。

    如果使用 <RegExp> 而非 <Values>，则此语法将相同。不论在哪种情况下，都将为变量注入键的完整值。

- **通过注入键的部分值。**

- 在专用 <Variables> 部分中，请使用以下语法：

```
<KeyRegExpVariable variable=" VARIABLE_NAME" key=" KEY_NAME"
expression=" REGULAR_EXPRESSION" group=" GROUP_NUMBER" />
```

在这种情况下，GROUP_NUMBER 是 REGULAR_EXPRESSION 所定义的正则表达式中的组的索引（从 0 开始）。例如，在包含以下行

```
myKey = 123Value123
```

和以下语句

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

的属性文档中，将值“Value”注入变量“VAR”。

可以使用在同一文件或其他文件中定义的变量。例如：

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

假设 PREV_VAR 包含值“123”，则 VAR 将获得值“Value”，与注入键的整个值的方式相同。不管使用哪个选项，变量都始终包含一个值。

- 作为搜索条件的一部分，请使用以下语法：

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

VARIABLE_NAME 变量的键在 <KeyCondition> 元素中定义。

仅在使用 <RegExp> 时才支持此操作，变量的“group”属性所引用的正则表达式与 <RegExp> 中的正则表达式相同。

XML 配置文档

在 XML 配置文档中，可以使用 XPath 查询基于 XML 标准为配置文档轻松编写搜索表达式。

要声明配置文档是 XML 配置文档，请使用 <XmlConfigurationDocument> 元素。例如：

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

定义条件

XML 配置文档中允许使用的唯一搜索条件是有效的 XPath 2.0 查询。如果从 XPath 选择了节点，则条件返回 True；否则返回 False。查询可在以下位置包含变量:

- 在元素名称中

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

如果变量包含多个值，框架将执行多个 XPath 语句。例如，如果 SITE_NAME 具有值 SITE1 和 SITE2，则此示例中的语句将生成以下两个具体的搜索:

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

- 在相等性测试中

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

如以上示例所示，相等性仅适用于包含单个值的变量。如果变量可能包含多个值，请改用以下语法:

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

备注:

- 使用 ${equals()} 测试区分大小写的相等性，使用 ${equals-ignore-case()} 测试不区分大小写的相等性。
- 第一个参数应该为 XPath 函数或属性名称。
- 第二个参数应该始终为变量名称。显示的变量名称应该没有 ${}。
- 此语法适用于具有一个值或多个值的变量，建议在所有情况下都使用此语法。
- 如果存在多个值，且第一个参数的值正好等于变量的某个值，则函数返回 True。

- 在正则表达式中

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

如果变量包含值列表，则将生成类似于下面的表达式:

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

任何一个值都将为条件返回 True。

XML 配置文档中的条件示例

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

此示例显示 **MyConfig.xml** XML 配置文档必须具有以下所有项:

- \URL\Protocol\@name 必须具有值 HTTP 或 HTTPS。
- \URL\Host\@name 必须包含提供程序的 IP 地址（或其 IP 地址之一）或其主机名。
- \URL\Site\text() 必须以 / 开头，接着是提供程序的 SITE_NAME，包含另一个 /，然后包含任意字符串。

注入变量值

可以使用 XPath 查询文本值并将其注入变量。不能将整个元素注入变量。要在专用 <Variables> 部分中达到此目的，请使用以下语法:

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

下面是几个示例:

- 选择属性值

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

  从 XML 文档的所有 \Root\Element 元素的 **Att** 属性中选择值。
- 选择元素文本

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

选择与 \Root\Element 匹配的所有元素的文本。

- 选择具有条件的属性值（使用相同或不同配置文档中的变量）

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

从 @Att=${VAR} 的所有 \Root\Element 中选择 @AnotherAtt 的值。

可以将 XPath 变量用作 XPath 条件的一部分。在此模式中，假设存在结果（即条件计算为 True），XPath 还可以包含条件的结果节点的相对路径。使用以下语法：

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

例如：

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

假设 XPathCondition 计算为 True，且某个 XML 节点（具体来说是指此示例中的元素）被 <XPath> 选中，则变量将包含该节点的“type”属性值。

通过从文档的根目录（以“/”开头）指定 XPath，可以为与该变量无关的变量定义 XPath。

进行 XPath 注入后，变量可能包含值列表。

文本配置文档

文本配置文档是格式对内容开发人员已知的文本文档，但不是属性配置文档或 XML 配置文档。可以使用正则表达式在文本配置文档中编写搜索表达式。

要声明配置文档是文本文件类型，请使用 <TextConfigurationDocument> 元素。例如：

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

定义条件

文本配置文档中允许使用的唯一搜索条件是正则表达式。如果模式匹配，条件将返回 True。

正则表达式可能包含一个或多个变量作为条件的一部分。运行期间，将在生成具体的搜索表达式时使用实际值替换变量。

如果变量包含值列表，则将生成类似于下面的表达式：

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

任何一个值都将为条件返回 True。

文本配置文档中的条件示例

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

此示例显示 **MyConfig.txt** 文本配置文档必须具有以下所有项：

- \URL\Protocol\@name 必须具有值 HTTP 或 HTTPS。
- \URL\Host\@name 必须包含提供程序的 IP 地址（或其 IP 地址之一）或其主机名。
- \URL\Site\text() 必须以 / 开头，接着是提供程序的 SITE_NAME，包含另一个 /，然后包含任意字符串。

注入变量值

可以使用包含组的正则表达式将值注入文本文档的变量中。这些组将标记应注入的文本。在专用 <Variables> 部分中，请使用以下语法：

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

在这种情况下，GROUP_NUMBER 是 REGULAR_EXPRESSION 所定义的正则表达式中的组的索引（从 0 开始）。

例如，对于包含以下行

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

和以下语句

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

的文件，将值“document”放入变量 VAR 中。可以在同一文档中同时使用在不同文档中定义的变量。例如：

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

如果 PREV_VAR 的值为“configuration”，则将使用字符串“document”注入 VAR。

通过使用以下语法，可以将 <RegExpVariable> 用作 <RegExpCondition> 标记的一部分：

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

变量的“group”属性所引用的正则表达式与 <RegExp> 中的正则表达式相同。

注入的变量始终使用一个值注入。

指定搜索范围

搜索范围的目的是找到可能是具有特定提供程序类型的依赖关系一部分的所有使用者。范围具有两种用途：

- 从 UCMDB 中筛选出不相关的可部署组件，以便搜索的结果更少。（必需）

  要达到此目的，请尝试在所有配置文档中查找提供程序的连接字符串的子集，并将搜索范围限制为仅搜索连接到这些配置文档的可部署组件。

  此搜索表达式与特定于提供程序的依赖关系搜索不同，因为它必须尽快返回所有可能的使用者，而不是返回表示使用者-提供程序依赖关系是否存在的确切结果。

  搜索表达式语法的组成方式与配置文档搜索表达式的编写方式类似；但不是每个文件类型都有任何特殊条件，也不需要指定各个文件的名称。有关详细信息，请参阅编写搜索表达式 (第 68 页)。

  下面的示例表示以下表达式：(x | ((y & z & w) & (h | g)))。

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

```
        <Operand value="g"/>
      </Operator>
    </Operator>
    <Operand value="x"/>
  </Operator>
</ConfigurationDocumentSearchCondition>
```

x、y、z、w、h 和 g 可能是常量值、变量或概念变量。

将使用此表达式在 UCMDB 的所有相关使用者配置文档中搜索这些值（或变量值）。

只有拥有这些文件的使用者才会继续执行特定于提供程序的搜素，并使用精确的搜索表达式进行计算。

- 将可部署组件限制为提供程序的已连接组件的子集；例如，仅搜索是提供程序的 J2EE 域的一部分的依赖关系。（可选）

  要达到此目的，可使用 TQL 查询查找以某些方式连接到提供程序类型的所有可部署组件。有关详细信息，请参阅定义 TQL 查询 (第 86 页)。

  TQL 查询的结果（给定的特定提供程序 CI）定义可能与该提供程序存在依赖关系的所有可能的可部署组件。TQL 查询必须遵循以下规则：

  - 可部署组件的查询节点必须称为“Deployable”（区分大小写）。此查询节点的结果是范围的可能的可部署组件。
  - “Deployable”组件表示可能处于同一范围内的使用者和提供程序组件。因此，查询节点 CI 类型必须是使用者和提供程序的可部署 CI 类型的原始节点。
  - TQL 查询中必须只有一个其他查询节点（除“Deployable”以外）。其他查询节点的名称不受限制。此节点称为 Scope 查询节点。
  - Deployable 和 Scope 查询节点通过包含这两个节点之间的所有可用路径的复合链接连接。

- 所有范围会隐式筛选出与提供程序不在同一路由域中的使用者可部署组件（假设提供程序由特定的路由域搜寻）。使用路由域进行筛选需要可部署描述符，以便具有节点路径。有关详细信息，请参阅定义使用者的描述符 (第 66 页)。

  例如，如果提供程序可部署组件是某个类型的运行软件，则该运行软件的路由域将与其包含的节点相同。该节点的路由域由其 IP 地址的路由域确定。如果其 IP 地址来自不同的路由域，则该节点可能与多个路由域相关。另一方面，如果提供程序可部署组件是与任何节点或 IP 地址不相关的 CI（如业务服务），则可以将其连接到任何使用者可部署组件，而不管其路由域如何。

例如，要定义 J2EE 域范围，请执行以下操作：

1. 创建类型为“J2EE Deployed Object”的 Deployable 查询节点。这是表示 J2EE 域中可部署组件的所有 CI 类型中的最小公分母类。例如，此范围内的可部署组件可能为 Web Module 或 J2EE Application。
2. 创建类型为“J2EE Domain”的 Scope 查询节点。
3. 创建 Dependency 和 Scope 查询节点之间的复合链接，该链接具有 J2EE Deployed Object 和 J2EE Domain 之间的所有路径。例如，复合链接可能包含以下内容：
   - J2EE Application – Composition – J2EE Domain
   - Web Module – Composition – J2EE Application

TQL 查询定义必须嵌入到 Scope XML 中。

范围搜索中变量的默认值

让我们使用以下示例:

- 您将使用默认值为 80 的 PORT 变量。在某些配置文件中可能并未显示此默认值。
- 您将使用 IP_ADDRESS AND PORT 作为范围搜索表达式，对于某些提供程序，PORT 变量的值为 80。此范围的搜索不会返回连接到不包含默认端口值 80 的配置文档的可部署组件（即使应该返回），因为我们在搜索表达式中规定端口必须存在于配置文件中。

因此，当 PORT 变量为默认值时，将从范围搜索表达式中删除它，其方式与在依赖关系搜索表达式中忽略它的方式相同。有关详细信息，请参阅使用变量的默认值 (第 70 页)。

当变量包含的值不是默认值时，将按通常的方式将此值注入搜索表达式。

疑难解答

要测试范围是否返回预期的使用者，请在 UCMDB 服务器上的 Discovery Manager 服务中使用 **searchConfigurationDocuments** JMX 方法。“searchString”参数应为搜索表达式 XML，如指定搜索范围 (第 84 页)中所述。XML 应仅包含常量值，而非变量。

还可以从某个搜索适配器中的提供程序通信日志中检索此 XML。有关详细信息，请参阅依赖关系搜索适配器 (第 88 页)。

定义 TQL 查询

1. 将 TQL 查询定义添加到依赖关系签名的 <Queries> 部分中，如下所示:

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

2. a. 使用文本编辑器创建空文档。
   b. 添加以下内容:

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

</tql:query>
```

   c. 在建模工作室中生成 TQL 查询。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的“建模工作室”。
   d. 将 TQL 查询导出到 XML。
   e. 打开您导出的 XML 文档，并将所有文本复制到 <resource> 元素中。

f. 将此文本粘贴到您在步骤 1 中创建的文档的 <tql:query> 元素中。

g. 复制文本文档的完整内容并将其放到依赖关系签名文件的 <Query> 元素中。

备注: TQL 查询嵌入在依赖关系签名 XML 文件中，且不会对其进行验证。但是，如果查询 XML 文件本身无效，部署期间将发生异常。有关详细信息，请参阅编译错误 (第 87 页)。

打包和部署多个依赖关系签名文件

可在 UCMDB 中部署多个依赖关系签名文件。搜索使用者-提供程序链接时，将使用所有已部署的文件。

依赖关系签名文件是前缀为 **dependencies/**、后缀为 **.xml** 的搜寻配置文件，例如 **dependencies/JEE.xml**。

每个依赖关系签名文件都是完全独立的。注意以下内容:

- 只能在定义全局变量定义的文件中使用全局变量定义。强烈建议尽可能对类似变量使用相同名称。例如，如果变量包含 IP 地址且存在两个依赖关系文件，则应在这两个文件中均定义名为 IP_ADDRESS 的变量。这样，不同的搜索适配器可以使用名称为 IP_ADDRESS 的单个目标数据。有关详细信息，请参阅指定变量值 (第 91 页)。
- 只能在定义概念定义的文件中使用概念定义。强烈建议尽可能对类似用途的概念使用相同的概念名称和概念变量名称。有关详细信息，请参阅指定概念变量值 (第 92 页)。
- 如果在不同的文件中定义可部署组件，则这些组件可以具有相同的名称。但它们将被视为不同的可部署组件。
- 如果在不同的文件中定义 TQL 查询，则这些查询可以具有相同的名称，但将被视为不同的 TQL 查询。在依赖关系签名文件中按名称引用 TQL 查询时，具有该名称的 TQL 查询必须存在于同一文件中。
- 如果在不同的文件中定义范围，则这些范围可以具有相同的名称。但它们将被视为不同的范围。在依赖关系签名文件中按名称引用范围时，具有该名称的范围必须存在于同一文件中。
- 如果在不同的文件中定义条件函数，则这些函数可以具有相同的名称。在依赖关系签名文件中按名称引用函数时，具有该名称的函数必须存在于同一文件中。
- 变量和概念变量的默认值特定于定义它们的文件。具有相同名称的两个变量或概念变量在不同的文件中可以具有不同的默认值。
- 不建议在不同的依赖关系签名文件中为同一概念设置不同的键属性。否则会导致维护困难以及难以从目标数据变量中的不同适配器正确分配值。有关详细信息，请参阅指定概念变量值 (第 92 页)。

编译错误

编译验证

- 同一文件中有两个或更多可部署组件具有相同名称。
- 同一文件中有两个或更多范围具有相同名称。
- 同一文件中有两个或更多 TQL 查询具有相同名称。
- 同一文件中有两个或更多条件函数具有相同名称。
- 同一可部署组件下有两个依赖关系具有相同名称。

- 在配置文件条件中使用未声明为同一文件中的全局变量或配置文件所属的依赖关系中的局部变量的变量名称。
- 在条件函数中使用未声明为全局变量或函数参数的变量名称。
- 变量之间存在循环依赖关系（例如，变量 VAR_A 的注入语句使用 VAR_B 的值，VAR_B 的注入值则使用变量 VAR_A 的值）。
- 引用不存在于同一文件中的 TQL 查询。
- 引用不存在于同一文件中的范围。
- 引用不存在于同一文件中的条件函数。
- 在 ignore 语句或备用表达式中使用没有默认值的变量会导致编译错误。有关详细信息，请参阅使用变量的默认值 (第 70 页)。
- 如果搜索表达式包含具有默认值的变量，则必须处理这些变量的任意组合将得到其默认值的所有情况。通过完全忽略某些变量的表达式或使用备用表达式，可以实现这一点。必须对未显示在 ignore 语句中的所有变量组合使用备用表达式。否则将导致编译错误。有关详细信息，请参阅使用变量的默认值 (第 70 页)。
- 在 <DocumentCILocation> 标记中使用 TQL 查询，但未遵循以下规则：
  - TQL 查询必须定义可部署组件与配置文档之间的路径。此路径必须简单（无循环）。
  - TQL 查询必须包括以下两个端节点，具有特定名称且区分大小写：
    - Deployable - 路径中的可部署组件 CI
    - Configuration_document - 路径中指定配置文档的 CI
  - Deployable 和 Configuration_document 节点不允许使用条件。
  - 所有节点之间的基数必须为 1..1。
  - 仅支持常规链接类型。不允许使用复合链接、联接或子图。
- 在范围中使用 TQL 查询，但未遵循以下规则：
  - 可部署组件的查询节点必须称为“Deployable”（区分大小写）。此查询节点的结果是范围的可能的可部署组件。
  - TQL 查询中必须只有一个其他查询节点（除“Deployable”以外）。其他查询节点的名称不受限制。此节点称为 Scope 查询节点。
  - Deployable 和 Scope 查询节点通过包含这两个节点之间的所有可用路径的复合链接连接。
  - “Deployable”组件表示可能处于同一范围内的使用者和提供程序组件。因此，查询节点 CI 类型必须是使用者和提供程序的可部署 CI 类型的原始节点。
- 文本配置文档包含多个具有不同优先级的 <ReferenceLocation> 标记。有关详细信息，请参阅配置文档替代 (第 72 页)。

依赖关系搜索适配器

依赖关系搜索框架必须由专用适配器执行。适配器负责执行以下操作：

- 收集提供程序的所有相关连接字符串。
- 执行依赖关系搜索框架。
- 将搜索结果（依赖关系）报告给 UCMDB。

每个适配器处理一种类型的提供程序。例如，一个适配器可以处理 JAR 提供程序。

还可以让多个适配器处理相同的提供程序类型。但是在这种情况下，请确保对每个提供程序仅执行每个适配器一次，以获得一致的结果。

对于其他适配器，一旦依赖关系搜索适配器就绪，就必须创建搜寻作业以执行适配器逻辑。

本节包含:

- 创建依赖关系搜索适配器 89
- 适配器限制 96

创建依赖关系搜索适配器

1. 选择“数据流管理”>“适配器管理”。
2. 单击“新建”  ，然后选择“新建适配器”。
3. 输入适配器的详细信息，然后单击“确定”。
4. 在“资源”窗格中，右键单击您刚创建的适配器，然后选择“编辑适配器源”。
5. 将以下行:

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

替换为

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

6. 将以下行:

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

替换为

```
<params
className="com.hp.ucmdb.discovery.probe.services.dynamic.core.WorkflowServiceParams"
patternType="workflow_adapter" enableAging="true"
ignoreMissingReconciliationRules="false" enableRecording="false"
autoDeleteOnErrors="success" recordResult="false" maxThreadRuntime="86400000"
useDefaultValueForAging="false">
  <workflow>
```

```
    <steps>
      <step name="Dependencies Discvoery" failure-policy="mandatory">
        <module
type="java">com.hp.ucmdb.discovery.probe.agents.probemgr.accuratedependencies.processi
ng.DependenciesDiscoveryWorkflowStep</module>
        <timeoutParking>
          <initialTimeout>180000</initialTimeout>
          <retriesThreshold>12</retriesThreshold>
          <multipleBy>2</multipleBy>
          <maxRetry>10</maxRetry>
          <timeoutThreshold>10800000</timeoutThreshold>
        </timeoutParking>
      </step>
    </steps>
    <finalStep />
    <libraryScripts />
  </workflow>
</params>
```

7. 准备将处理依赖关系签名搜索结果的脚本。有关详细信息，请参阅编写 Jython 脚本 (第 94 页)。
8. 将以下步骤添加到工作流适配器：

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

9. 在“输入”窗格中，单击“选择 CI 类型”  ，然后选择此适配器的提供程序 CI 类型。
10. 单击“编辑输入查询”  ，然后准备输入 TQL 查询。有关详细信息，请参阅定义输入 TQL 查询和目标数据 (第 91 页)。
11. 在“搜寻到的 CIT”窗格中，单击  并添加提供程序类型、所有可能的使用者类型以及提供程序-使用者链接类型。
12. 完成后，单击“保存”。

定义使用者-提供程序适配器

搜索功能受搜寻适配器支持。每个适配器搜索一种提供程序类型的一组特定连接字符串。适配器的输入 TQL 查询负责获取可找到这些连接字符串的所有 CI，并使用触发器的目标数据将值注入连接字符串变量和概念。有关详细信息，请参阅开发 Jython 适配器 (第 34 页)。

这些适配器的类型是“工作流适配器”，并执行多个搜索步骤搜寻使用者-提供程序关系：

1. 通过将连接字符串的值注入依赖关系签名全局变量和实例化概念，适配器可以为范围的配置文档筛选器运行编写具体搜索表达式的逻辑。
2. 适配器将范围的具体搜索表达式提交到 UCMDB 服务器中运行的 UCMDB Solr 引擎。

3. 适配器向 UCMDB 执行 TQL 查询以获取所有必需信息（例如，配置文档、可部署描述符等），以便可以执行依赖关系签名搜索。
4. 适配器将所需配置文档的内容下载并存储到 Data Flow Probe。
5. 然后适配器执行在步骤 2 和 3 中由范围找到的与提供程序和使用者相关的依赖关系搜索。
6. 最后，Jython 脚本报告依赖关系搜索的结果。

定义输入 TQL 查询和目标数据

要使用依赖关系签名范围和条件组成搜索表达式，则该搜索表达式中提及的每个变量和概念必须替换为具体的连接字符串（由 TQL 查询返回）。此类替换根据您定义的特定于适配器的映射完成。

依赖关系映射适配器的输入 TQL 查询必须定义：

- 用于查找提供程序的使用者的触发 CI（SOURCE 查询节点）
- 特定提供程序所需的所有连接字符串

这些 TQL 查询应返回分散在 CI 拓扑间的所有连接字符串并发现提供服务的提供程序。例如，如果提供程序是 Oracle RAC，则 TQL 查询应返回包含使用者连接到 RAC 并使用其服务所需的任何连接字符串的所有 CI。因此，TQL 查询的布局定义应提及用于存储此类连接字符串的任何属性。例如，RAC 的 TQL 查询应返回可用于访问 RAC 的每个实例的 IP 地址和端口以及 RAC 服务名称。

必须使用适配器的目标数据在输入 TQL 查询中定义将变量和概念映射到查询节点的属性。每个模式元素应具有唯一名称，以便正确映射。

对于任何适配器，将为与此适配器的关联作业的触发 TQL 查询匹配的任何触发执行搜索适配器的输入 TQL 查询。

使用 Service Discovery 适配器时，可能不仅仅要将触发限制为服务器搜寻到的触发（自动发生），还要限制为连接到业务服务 CI 的触发（通过某个路径）。要完成此操作，请定义输入 TQL 查询，以便包含触发和业务服务 CI 之间的路径（通常为某些复合链接），并将业务服务 CI 的查询节点命名为“SERVICE”（区分大小写）。然后，当框架看到“SERVICE”查询节点时，它会自动将结果仅限制为触发 CI 所属的一个或多个服务。

指定变量值

每个搜索适配器需要为显示在适配器应该搜寻的依赖关系的依赖关系签名的搜索条件中的所有全局变量和概念设置值。这些值表示提供程序的连接字符串（有关详细信息，请参阅变量和概念 (第 63 页)），并将用于查找提供程序的使用者（触发器）。

> **备注:** 如果其中一个映射属性为空，请将其保留为如以下示例中所示：
>
> CONTEXT_ROOT = ${WEB_MODULE.j2eemanagedobject_contextroot:}
>
> 在此示例中，CONTEXT_ROOT 是将接收依赖关系签名中的空值的属性，将被忽略。

要设置变量的值，请执行以下操作：

1. 添加名称正好与变量相同（区分大小写）的目标数据值。
2. 将目标数据值设置为硬编码值或变量。

有关详细信息，请参阅开发 Jython 适配器 (第 34 页)。

指定概念变量值

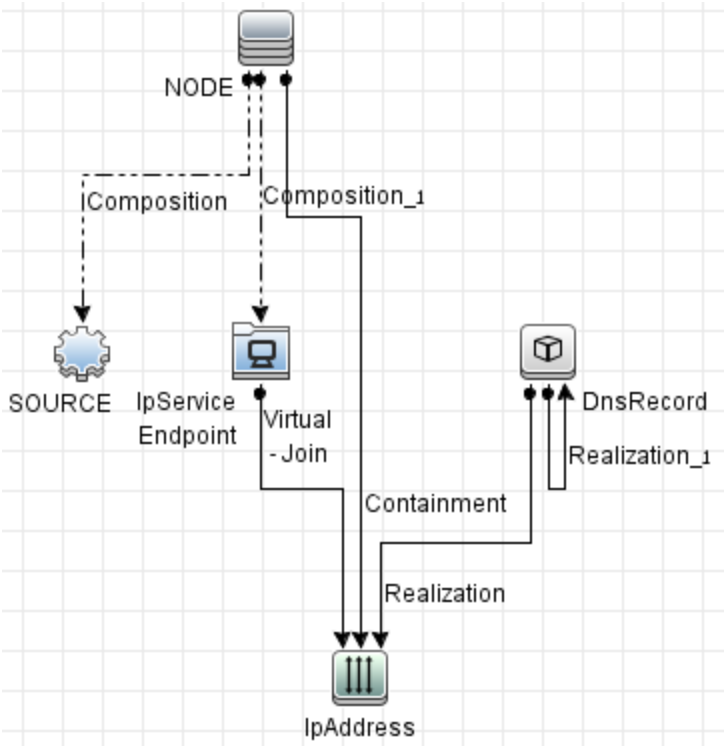
应实例化概念以表示一组不可分离、紧密耦合的连接字符串。输入 TQL 查询可能会返回多组不可分离的连接字符串。例如，运行软件实例可能会侦听多个 IP:端口对。对于每个这样的组，应实例化单独的概念实例。概念的键属性用于区分概念实例。键引用 TQL 查询的模式元素。对于为键模式元素返回的每个 CI 实例，将创建新的概念实例。每个此类 CI 实例称为“键 CI 实例”。

备注: TQL 查询不能包含同一概念的多个映射定义。

每个概念实例的连接字符串应取自匹配的键 CI 实例以及连接到该键 CI 的 CI。下面是包含此概念定义的依赖关系签名文件示例:

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

适配器的输入 TQL 查询如下所示:



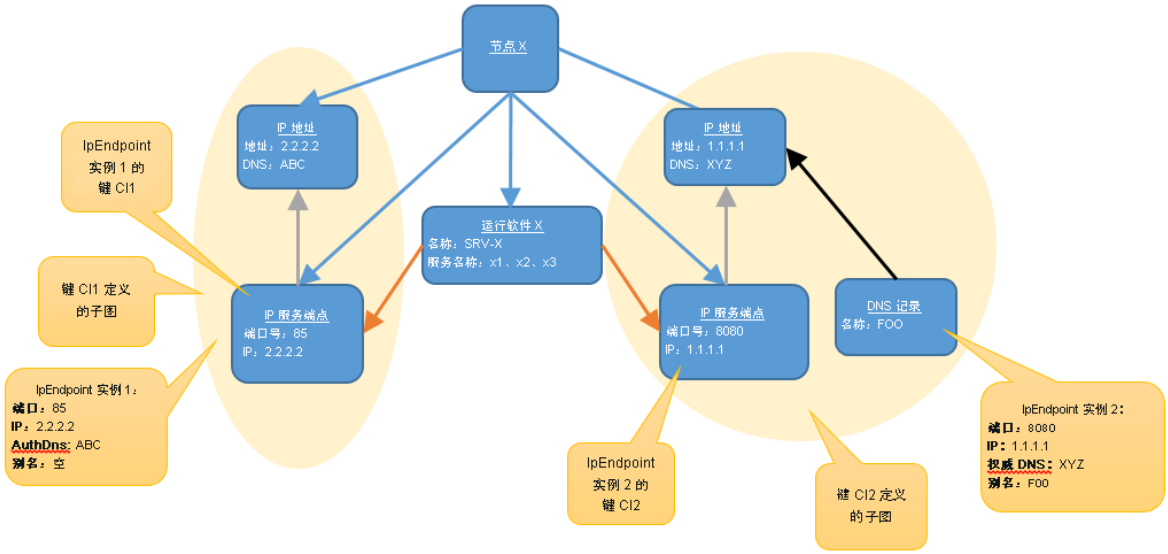
与变量一样，要实例化适配器中的概念，每个概念变量需要映射到目标数据变量。有关详细信息，请参阅指定变量值 (第 91 页)。

适配器的目标数据如下所示:

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

因此，**IpServiceEndpoint** 查询元素是此适配器中 **IpEndpoint** 概念的键 CI。

在 TQL 查询图示例（表示运行软件 X 的 TQL 查询结果）中，可以识别两个键 CI。用于填入每个 **IpEndpoint** 概念实例的属性的连接字符串取自每个键 CI 的子图。这样每个概念实例表示一组不同的不可分离的连接字符串。



与变量不同，映射概念变量时，必须在输入 TQL 查询中提及从触发器的查询节点（始终命名为 SOURCE）到变量将绑定到查询节点的路径。此外，设置关键变量后，必须将关键变量的路径用作所有其他概念变量的前缀。

在此示例中，可以看到:

- 目标数据变量的名称是概念名称后跟概念变量名称。
- 通过指定目标数据变量中每个概念的名称，可以在同一适配器中映射多个概念。
- 路径始终以 SOURCE 开头。
- 不是关键变量的概念变量的路径将使用关键变量的路径作为前缀。
- 路径仅由查询节点名称组成，不包含链接。但是，路径中指定的两个查询节点名称之间必须存在至少一个链接。

如果路径包含以下一项或多项，则触发器将在分派阶段失败:

- 输入 TQL 查询中不存在的查询节点名称。
- 输入 TQL 查询中未链接的两个相邻查询节点名称。
- 不是结果 CI 类型一部分的属性名称。

如果不是关键查询节点的查询节点有多个结果 CI，则目标数据变量将为这些 CI 中所有属性值的列表。只能对包含同一概念的任何其他目标数据变量所不包含的路径的目标数据变量创建列表。否则，触发器将在分派阶段失败。在上述示例中，IpEndpoint.IPADDRESS 目标数据变量不能包含列表，因为其路径包

含在 IpEndpoint.DNS 目标数据变量中。IpEndpoint.DNS 可以包含列表，因为其路径不包含在任何其他变量中。这意味着连接到 **IpServiceEndpoint** 查询节点中的结果的 **IpAddress** 查询节点只能有一个结果 CI。计划 TQL 查询和路径时，应考虑这一点。

如果 TQL 查询中存在自助链接，则创建列表的方式将与不是关键查询节点的查询节点中有多个结果 CI 时的创建方式类似。因此，具有自助链接的查询节点存在相同的限制。

有关详细信息，请参阅变量和概念 (第 63 页)。

编写 Jython 脚本

创建依赖关系映射适配器的最后一步是编写 Jython 脚本以报告结果。

要访问依赖关系搜索的结果，必须使用以下代码行从工作流状态检索它们:

```
workflowState = Framework.getWorkflowState()
searchResult = workflowState.getProperty(DependenciesDiscoveryConsts.DEPENDENCIES_
DISCOVERY_RESULT)
```

如果搜索步骤成功，可确保 searchResult 变量为非空。建议在工作流中强制定义搜索步骤，以便在搜索步骤失败时不会执行 Jython 脚本步骤。变量将包含类型为 **com.hp.ucmdb.discovery.probe.agents.probemgr.accuratedependencies.search.ConsumerDeployableSearchResult** 的对象。

使用此对象执行以下操作:

1. 检查依赖关系搜索找到的所有使用者可部署组件。请记住：搜索将获取提供程序的连接字符串作为输入，并返回依赖于提供程序的所有使用者可部署组件。
2. 对于每个使用者，可能存在多个依赖关系签名。您还应该对它们进行迭代操作。每个依赖关系签名可能具有不同的输出变量值。事实上，依赖关系签名中使用的所有变量（全局、局部或概念变量）的值对搜索结果中的 Jython 脚本可用。
3. 创建包含使用者与提供程序之间的链接的对象状态持有者 (OSH)。还可以从搜索结果对象中获取提供程序的详细信息。
4. 将 OSH 添加到结果矢量 (OSHV) 并将结果发送到 UCMDB。

下面是执行上述流程的 Jython 脚本片段:

```
# Get the search results object from the Workflow State
workflowState = Framework.getWorkflowState()
searchResult = workflowState.getProperty(DependenciesDiscoveryConsts.DEPENDENCIES_
DISCOVERY_RESULT)

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

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

# Loop through all the consumer deployable components
```

```
for index in range(0, searchResult.size()):
  deployable = searchResult.get(index)

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

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

  references = []

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

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

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

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

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

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

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

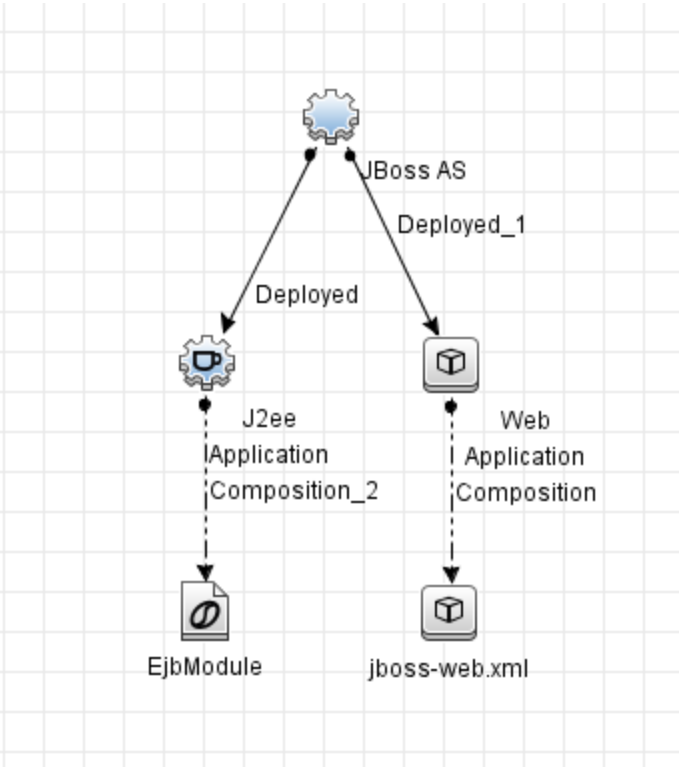
备注: 将基于脚本发送的信息执行服务边框规则（适用于搜寻规则引擎的一组规则）。因此，尽量多发送有关关系、使用者和提供程序的信息非常重要。建议创建从搜索结果对象中检索到的使用者与提供程序 OSH 之间的关系，如上所示。这些 OSH 将包含服务边框规则的所有必需信息（及其他规则引擎规则），以便正常运行。

适配器限制

- 以单独模式安装的 Data Flow Probe 不支持依赖关系映射适配器。
- UCMDB 服务器重新启动时，依赖关系映射适配器作业的触发器可能会由于超时而失败。触发器应该会在下次计划运行期间成功。

完整示例

本节将提供有关如何开发依赖关系签名和适配器的完整示例，用于查找在同一 JBoss AS 中部署的 J2ee 应用程序和 Web 应用程序之间的依赖关系。



本节包含以下主题:

开发工作流

开发依赖关系签名和适配器的一般流程如下:

1. 了解您要开发的签名的提供程序 CI 类型。在此示例中，提供程序类型为 J2ee Application。
2. 确定与要涵盖的提供程序相关的使用者可部署组件。由于签名特定于使用者的配置文档，因此每个使用者都拥有自己的依赖关系签名。

   在此示例中，我们将选择一个使用者可部署组件 - JBoss 中的 Web 应用程序。
3. 确定是将这些签名添加到现有依赖关系签名文件还是创建新的签名文件。

这两种方法在功能方面没有区别。选择方便维护依赖关系签名的选项。例如，如果使用者可部署组件已存在于某个依赖关系签名文件中，则将此新依赖关系添加到现有可部署组件可能更易于维护。在此示例中，我们将创建新的依赖关系签名文件来包含新的依赖关系签名。

4. 检查是否需要新的适配器，或者提供程序是否已被现有适配器涵盖。请记住，搜索适配器的触发 CI 是提供程序 CI 类型。您需要检查：
   a. 是否存在已将此 CI 类型作为触发 CI 的现有搜索适配器？

      如果没有，则必须创建此类适配器。在此示例中，我们将执行此操作。
   b. 如果搜索适配器存在，所需的所有连接字符串变量和概念是否已映射到目标数据？

      如果没有，则必须更新现有适配器并将这些变量添加到目标数据。请记住，目标数据提取自输入 TQL 查询。这意味着，输入 TQL 查询可能需要更改，以便提供新信息。
5. 如果适配器作业尚不存在，请为适配器准备作业。
6. 确保触发 TQL 查询也将提供程序作为触发器。

开发依赖关系签名

开发 JBoss J2EE Application 和 Web Application 之间的依赖关系签名。

1. 了解创建使用者和提供程序之间的依赖关系所需的连接字符串。在此示例中，所需连接字符串为提供程序 J2EE Application 中包含的 EJB 模块的 EJB 的 JNDI 名称。
2. 所有连接字符串应该定义为全局变量或概念。这些变量的值将由适配器注入。但是，在依赖关系签名文件（新的或现有的）中定义这些变量之前，请检查所有文件，查看是否已存在类似的变量和概念。如果存在，请为您的变量、概念和概念变量指定与现有项相同的名称。这样适配器的开发将变得更加轻松，因为需要注入的全局变量数不会增加。在此示例中，我们会将这些定义添加到新文件，以便文件如下所示：

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

3. 分析配置文件并标识：
   - 每个配置文件中各个连接字符串的位置。
   - 每个文件的类型：属性、XML 或其他文本文件。
   - 每个文件之间的关系（如果存在多个文件）。

   在此示例中，JBoss Web 应用程序将在 **jboss-web.xml** 配置文件中定义 EJB 引用，如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
  <!-- A reference to an EJB in the same server with a custom JNDI binding -->
  <ejb-ref>
    <ejb-ref-name>ejb/BHome</ejb-ref-name>
    <jndi-name>someapp/ejbs/beanB</jndi-name>
  </ejb-ref>
```

```
    <!-- A reference to an EJB in an external server -->
    <ejb-ref>
      <ejb-ref-name>ejb/RemoteBHome</ejb-ref-name>
      <jndi-name>jnp://otherserver/application/beanB</jndi-name>
    </ejb-ref>
  </jboss-web>
```

<ejb-ref-name> 部分中的值是连接字符串的引用。由于这是 XML 文件，因此我们可以使用以下 XPath 表达式匹配 EJB 的 JNDI 名称：

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

4. 您必须定义包含默认搜索表达式的范围，这样有助于查找依赖关系映射所需的配置文件。在此示例中，关键字是 JNDI 名称本身。以下是范围定义：

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

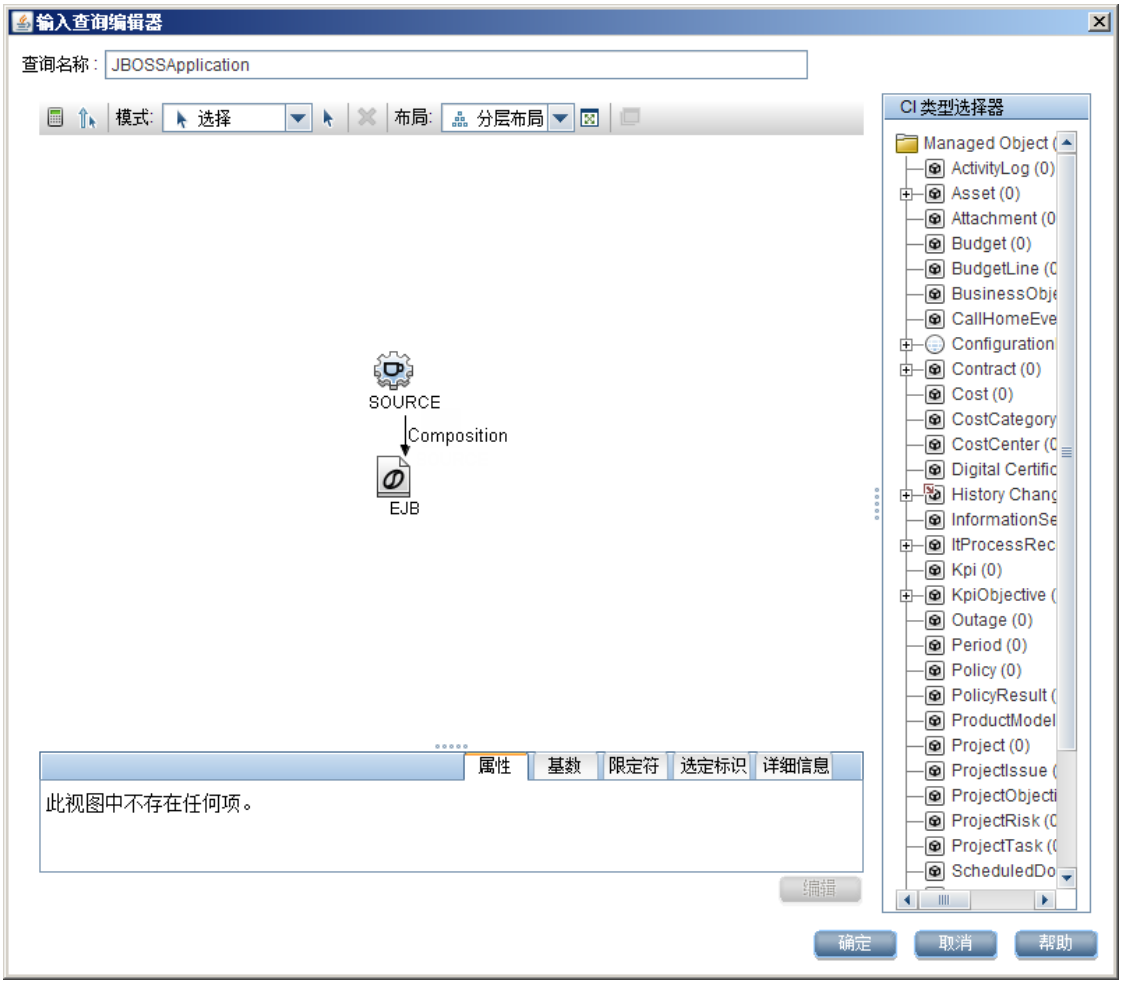
完整的依赖关系签名示例如下所示：

```
<?xml version="1.0"?>
<DependencySignatures xmlns="http://www.hp.com/ucmdb/1-0-0/Dependencies">
  <VariableDeclarations>
    <Variable name="EJB_JNDI_NAME"/>
  </VariableDeclarations>
  <Deployable name="JBoss J2EE Application to Web Application by JNDI" >
    <Descriptor cit="webapplication"/>
    <Dependency name="J2EE Application with Internal EJB" providerCiType="j2eeapplication"
scope="JBoss_EJB_Same">
      <XmlConfigurationDocument name="jboss-web.xml">
        <Condition>
          <Operator type="or">
            <XPathCondition>
              <XPath>//ejb-ref/jndi-name[matches(., '^${EJB_JNDI_NAME}$', 'i')]</XPath>
            </XPathCondition>
          </Operator>
        </Condition>
      </XmlConfigurationDocument>
    </Dependency>
  </Deployable>
  <ScopeDefinitions>
```

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

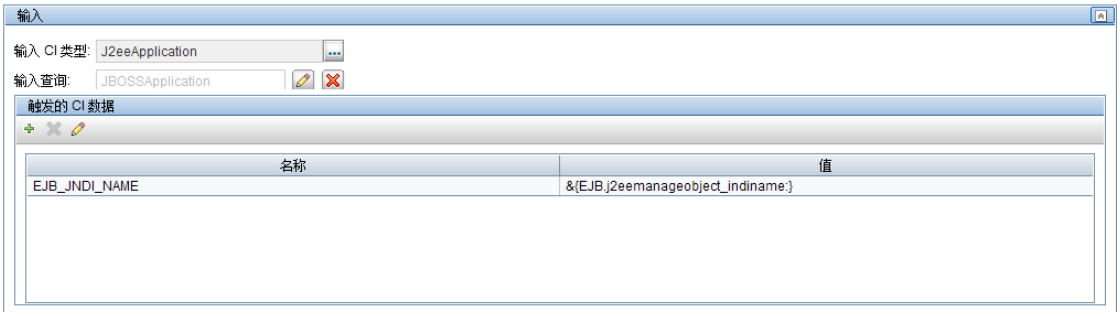
开发适配器

1. 创建新的工作流适配器，如创建依赖关系搜索适配器 (第 89 页)中所述。
2. 将触发 CI 类型设置为提供程序类型。在此示例中，J2EE Application 是提供程序。
3. 为适配器定义输入 TQL 查询，获取所需 CI 及其属性。在此示例中，我们需要触发 J2EEApplication CI 和 EJBModule CI（属于提供程序 J2EE Application）。



4. 使用触发 CI 数据为所需连接字符串变量和概念映射签名中定义的全局变量。在此示例中，我们将

EJBModule CI 的 j2eemanagedobject_jndiname 属性映射到 EJB_JNDI_NAME 目标数据。



5. 在“工作流步骤”部分中，粘贴以下工作流定义：

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

可通过调整 **timeoutParking** 参数更改超时持续时间。

6. 为新适配器创建触发 TQL 查询。
7. 将作业定义添加到“Service Discovery 活动类型”中，如下所示：

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

```
    ...
  </JobsDefinitions>
</ServiceDiscoveryActivityType>
```

第 5 章: 开发常规数据库适配器

本章包括:

- 常规数据库适配器概述 103
- 常规数据库适配器的 TQL 查询 104
- 调节 105
- Hibernate 作为 JPA 提供程序 105
- 准备创建适配器 107
- 准备适配器包 112
- 配置适配器 – 最小方法 114
- 配置适配器 – 高级方法 118
- 实施插件 121
- 部署适配器 124
- 编辑适配器 124
- 创建集成点 124
- 创建视图 124
- 计算结果 124
- 查看结果 125
- 查看报告 125
- 启用日志文件 125
- 使用 Eclipse 在 CIT 属性和数据库表之间进行映射 125
- 适配器配置文件 130
- 现成的转换器 151
- 插件 155
- 配置示例 156
- 适配器日志文件 163
- 外部参考 165
- 疑难解答和局限性 – 开发常规数据库适配器 165

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。

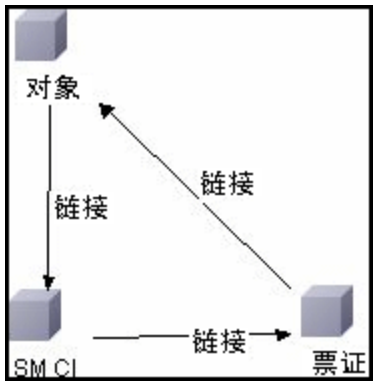
常规数据库适配器的 TQL 查询

对于填入作业，必须在建模工作室的“布局设置”对话框中选中 CI 的每个必需布局。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的“查询节点/关系属性对话框”。请务必注意，CI 可能需要属性才能被识别，如果没有这些属性，CI 将无法添加到 UCMDB。

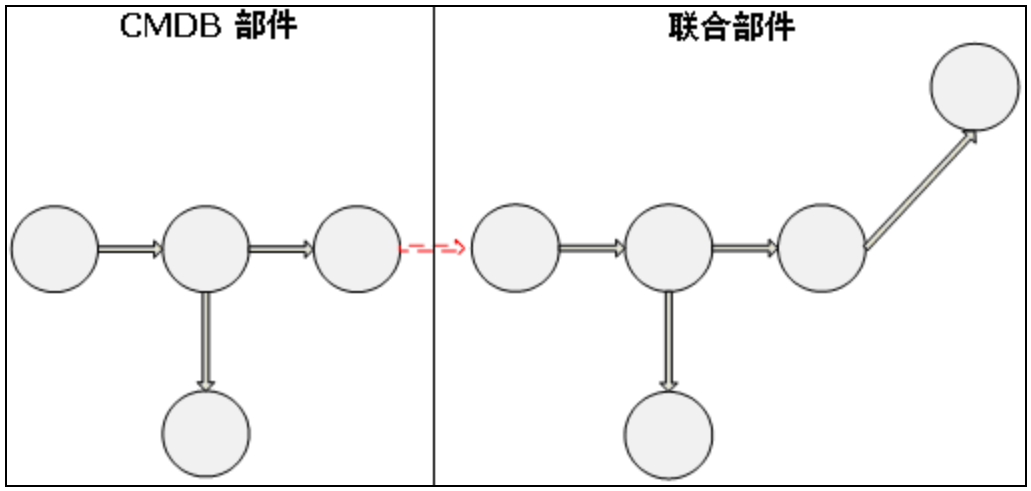
对于仅通过常规数据库适配器计算的 TQL 查询，存在以下限制：

- 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。



调节

调节过程将作为 TQL 计算的一部分在适配器端上执行。要执行调节，请将 CMDB 端映射到称为调节 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。

备注:

- 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 107 页)。
- 具有 CMDB ID 的调节也受支持
- 全局 ID 的调节也受支持。

Hibernate 作为 JPA 提供程序

Hibernate 是一种“对象-关系”(OR) 映射工具，此工具支持将 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 165 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，请参阅下文所述的关联 (第 107 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:

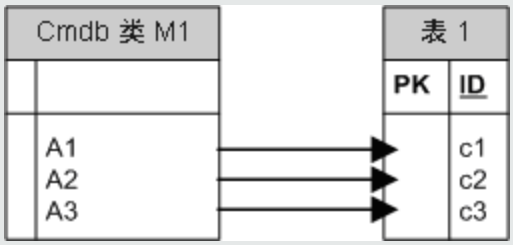
- “对象-关系”映射示例 (第 105 页)
- 关联 (第 107 页)
- 可用性 (第 107 页)

“对象-关系”映射示例

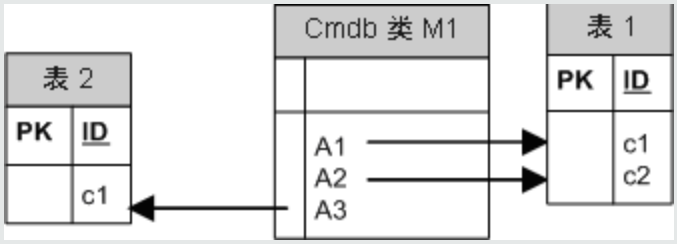
以下示例描述了“对象-关系”映射:

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。

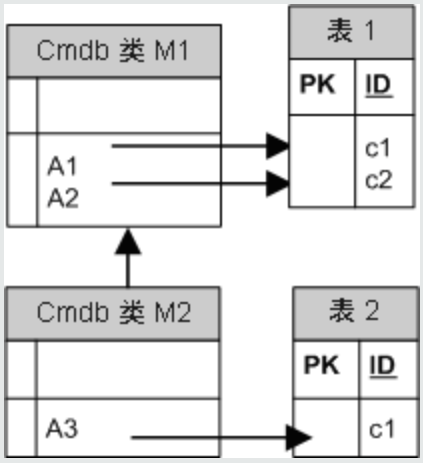


一个 CMDB 类映射到两个数据库表的示例：



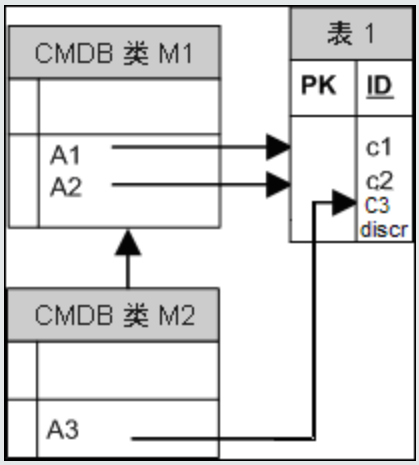
继承示例：

此情况用于 CMDB，其中每个类都具有自己的数据库表。



具有鉴别器的单表继承示例：

类的整个分层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（鉴别器），该列的值表示要映射到此条目的特定类。



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 文件以方便定义关联。

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。

准备创建适配器

此任务描述了在创建适配器前必须执行的准备工作。

备注: **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，以及某些插件界面的实施。**

此任务包括以下步骤:

- 先决条件 (第 107 页)
- 创建 CI 类型 (第 109 页)
- 创建关系 (第 109 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节类及其属性（也称为多节点）。例如，如果按节点名称运行调节，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 运行调节，请验证是否存在一个与 CMDB 中节点的 CMDB ID 匹配的 CMDB ID 列。有关调节的详细信息，请参阅调节 (第 105 页)。**

ID	名称	IP 地址

ID	名称	IP 地址
31	BABA	16.59.33.60
33	ext3.devlab.ad	16.59.59.116
46	LABM1MAM15	16.59.58.188
72	cert-3-j2ee	16.59.57.100
102	labm1sun03.devlab.ad	16.59.58.45
114	LABM2PCOE73	16.59.66.79
116	CUT	16.59.41.214
117	labm1hp4.devlab.ad	16.59.60.182

- 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 105 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 (R) 82801EB USB 通用主控制器
3581	2	系统	Intel (R) 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

CARD_ID	CARD_TYPE	CARD_NAME
1	串行总线控制器	Intel (R) 82801EB USB 通用主控制器
2	系统	Intel (R) 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，将创建一个表示 RDBMS（外部数据源）中数据的 CIT。

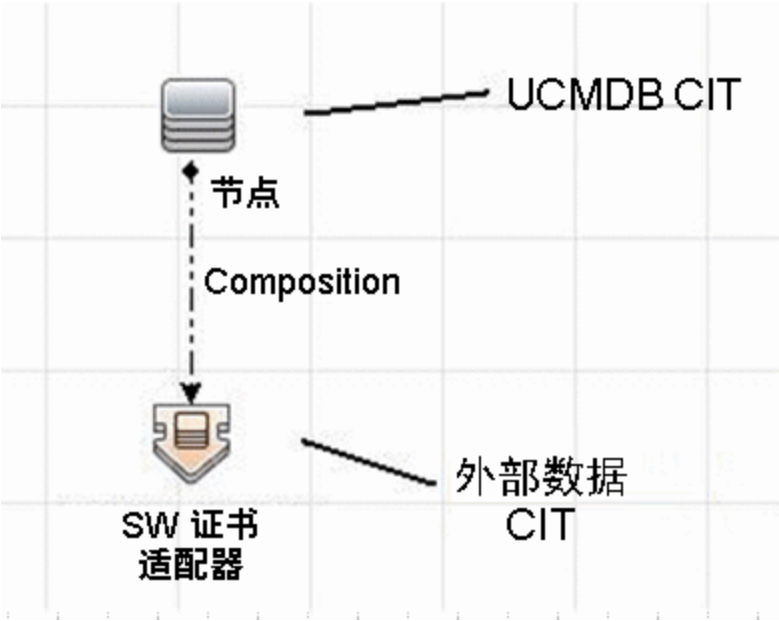
   a. 在 UCMDB 中，访问“CI 类型管理器”，然后创建新 CI 类型。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的“如何创建 CI 类型”。

   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，将在 UCMDB CIT 和新 CIT 之间添加关系，表示来自外部数据源的数据。

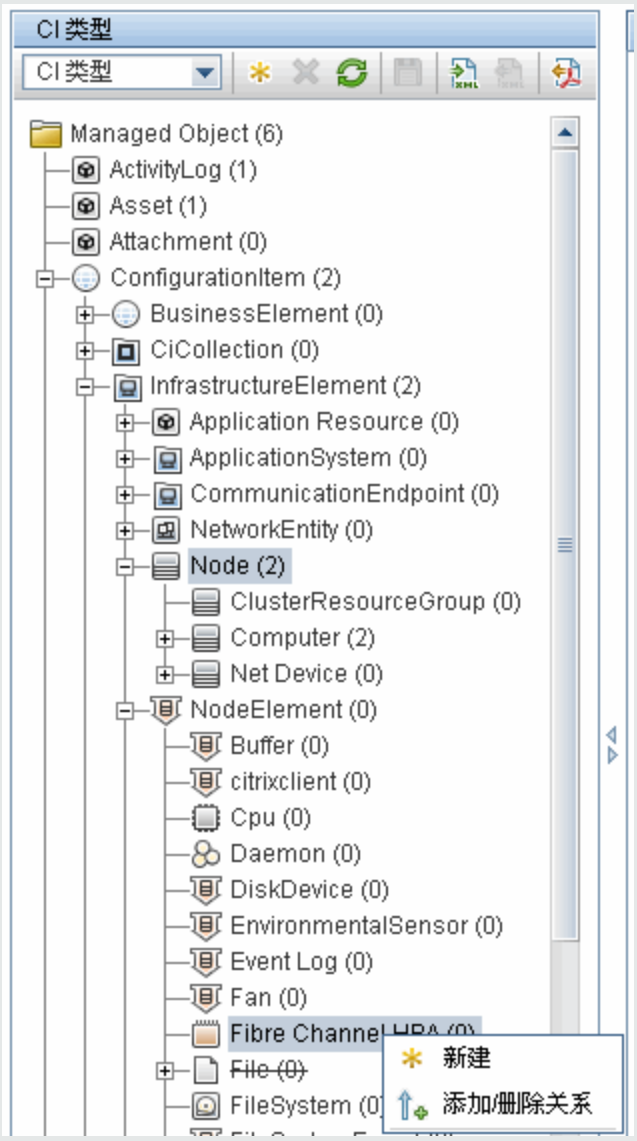
   向新 CIT 添加合适的有效关系。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的“添加/删除关系对话框”。



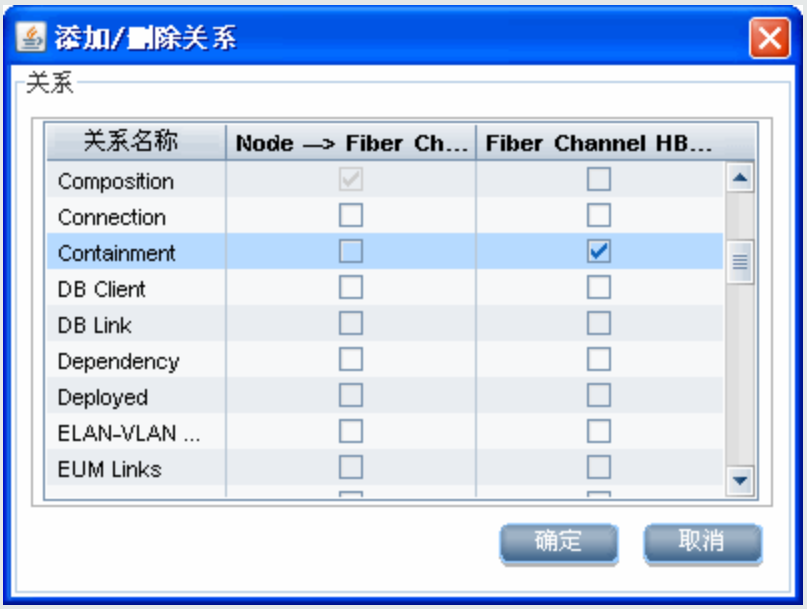
备注: 在此阶段，您还无法查看联合数据，也无法填充外部数据，因为尚未定义引入数据的方法。

创建 Containment 关系的示例：

a. 在 CIT 管理器中，选择两个 CIT：



b. 在这两个 CIT 之间创建 **Containment** 关系:



准备适配器包

您将在此步骤中查找并配置常规 DB 适配器包。

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

如果适配器支持填入，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 将出现在“集成点”窗格的适配器列表中。

创建常规 DB 适配器时，无需在 **support-replicatioin-data** 标记中编辑 **changes-source** 标记。如果已实施 **FcmdbPluginForSyncGetChangesTopology** 插件，则将返回上次运行时已更改的拓扑。如果未实施插件，则将返回完整拓扑，并将根据返回的 CI 执行自动删除。

有关在 CMDB 中填充数据的详细信息，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，不是新的调节映射引擎），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。

  备注: 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。

- 可以使用用户名（架构）、密码、数据库类型、数据库主机名称和数据库名称或 SID 来描述与数据库的连接。

  对于此类型的连接，适配器 XML 文件的 **parameter** 部分中具有以下参数元素：

```
<parameters>
  <!--The description attribute may be written in simple text or HTML.-->
  <!--The host attribute is treated as a special case by UCMDB-->
  <!--and will automatically select the probe name (if possible)-->
  <!--according to this attribute's value.-->
  <!--Display name and description may be overwritten by I18N values-->
    <parameter name="host" display-name="Hostname/IP" type="string" description="The host name or IP address
of the remote machine" mandatory="false" order-index="10" />
    <parameter name="port" display-name="Port" type="integer" description="The remote machine's connection
port" mandatory="false" order-index="11" />
    <parameter name="dbtype" display-name="DB Type" type="string" description="The type of database" valid-
values="Oracle;SQLServer;MySQL;BO" mandatory="false" order-index="13">Oracle</parameter>
    <parameter name="dbname" display-name="DB Name/SID" type="string" description="The name of the database
or its SID (in case of Oracle)"  mandatory="false" order-index="13" />
    <parameter name="credentialsId" display-name="Credentials ID" type="integer" description="The credentials to
be used" mandatory="true" order-index="12" />
</parameters>
```

备注: 这是默认配置。因此，**db_adapter.xml** 文件已包含此定义。

在某些情况下，无法以这种方式配置与数据库的连接。例如，与 Oracle RAC 的连接，或使用非 CMDB 附带的数据库驱动程序的连接。

对于这些情况，可以使用用户名（架构）、密码和连接 URL 字符串描述连接。

要定义此连接，请按如下方式编辑适配器的 XML 参数部分：

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

使用现成 Data Direct 驱动程序连接到 Oracle RAC 的 URL 示例如下：**jdbc:mercury:oracle://labm3amdb17:1521;ServiceName=RACQA;AlternateServers=(labm3amdb18:1521);LoadBalancing=true**.

4. 在临时目录中，打开 **adapterCode** 文件夹，并将 **GenericDBAdapter** 重命名为在上一步中使用的 **adapter id** 值。

   此文件夹包含适配器配置，例如，适配器名称、CMDB 中的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 – 最小方法 (第 114 页)。
6. 创建一个 *.zip 文件，其名称与 **adapter id** 属性的名称相同，如步骤编辑适配器 XML 文件： (第 112 页)所述。

   备注: 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**。

配置适配器 – 最小方法

简化（最小）方法是一种用于创建由适配器使用的 **simplifiedConfiguration.xml** 映射文件的方法。此方法支持单个 CIT 的基本填入或联合。

本节提供的说明介绍了如何将 CMDB 中特定 CI 类型的类模型映射到 RDBMS。

本节中提到的所有配置文件均位于 **C:\hp\UCMDB\UCMDBServer\content\adapters** 文件夹的 **db-adapter.zip** 包中，这些文件是在准备适配器包 (第 112 页)中提取的。

备注: 因运行此方法而自动生成的 **orm.xml** 文件是一个在使用高级方法时可以使用的有效示例。

您可以在需要执行以下操作时使用此最小方法：

- 联合/填充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/填入
- 仅支持多对一虚拟关系

配置 adapter.conf 文件

要更改 adapter.conf 文件中的设置以便适配器使用简化的配置方法，请执行以下操作:

1. 在文本编辑器中打开 **adapter.conf** 文件。
2. 找到以下行：**use.simplified.xml.config=<true/false>**。
3. 将此行更改为 **use.simplified.xml.config=true**。

示例：使用简便方法填充节点和 IP 地址

此示例演示如何将通过包含链接与“IP 地址”相关的“节点”填充至 UCMDB。RDBMS 具有一个名为 **simpleNode** 的表，其中包含有关计算机名称、计算机节点和计算机 IP 地址的数据。

simpleNode 表的内容如下所示:

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

整个填入过程分为以下三个阶段:

1. 创建 simplifiedConfiguration.xml (第 115 页)
2. 创建 TQL (第 116 页)
3. 创建集成点 (第 117 页)

创建 simplifiedConfiguration.xml

将 **simplifiedConfiguration.xml** 创建如下:

1. 将 **cmdb-class** 实体创建如下:

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

   “CI 类型”为 **node**，且 RDBMS 表名称为 **simpleNode**。
2. 将表的主键设置如下:

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

   此主键等同于 **orm.xml** 文件中的实体 ID。
3. 将 **reconciliation-by-two-nodes** 规则设置如下:

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

此标记定义“Node”和“IpAddress” CI 类型之间的关系。该关系类型为包含链接。调节由两个连接的 CI 类型执行。连接节点（在此示例中为“IpAddress”）的属性映射是在 **connected-node** 属性中定义的。

4. 在调节属性之间添加 **or** 条件，如下所示：

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

   此标记将在调节属性之间定义 OR 关系，这意味着第一个为 **true** 的调节属性会将整个调节规则设置为 **true**。

5. 添加以下属性：

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

   此标记将在 UCMDB 的 **node.name** 和 **simpleNode** 表的 **host_name** 列之间设置映射。

   对 **data_note** 属性执行相同的操作：

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

   添加连接节点的属性：

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

   此标记将在 **ip_address.name** 和 **simpleNode** 表中的 **ip_address** 之间设置映射。

6. 按顺序关闭打开的标记：

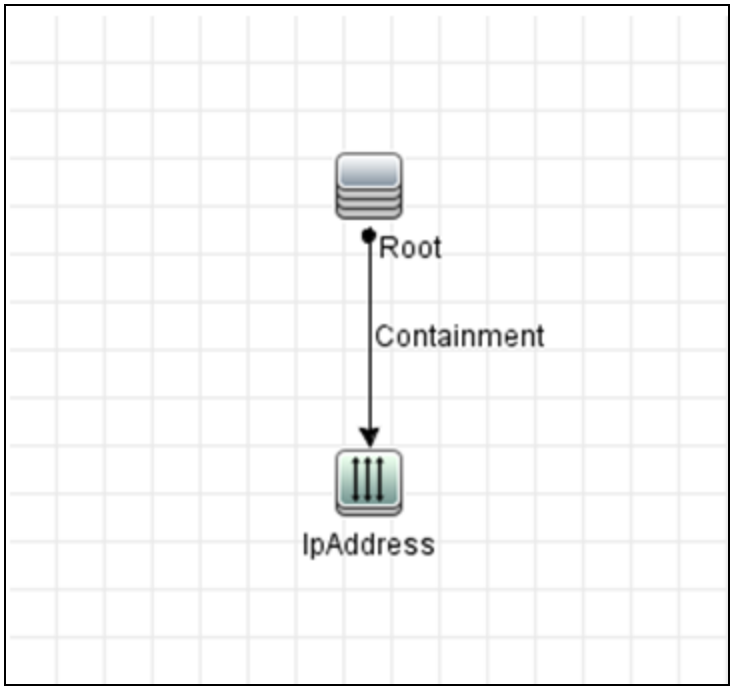
   ```
   </or>

   </reconciliation-by-two-nodes>

   </cmdb-class>
   ```

现在，simplifiedConfiguration.xml 文件的内容将显示如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-db-adapter-config xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
  <cmdb-class cmdb-class-name="node" default-table-name="simpleNode">
    <primary-key column-name="host_id"/>
  <reconciliation-by-two-nodes connected-node-cmdb-class-name="ip_address" cmdb-link-
type="containment">
    <or is-ordered="true">
    <attribute cmdb-attribute-name="name" column-name="host_name" ignore-case="true"/>
    <attribute cmdb-attribute-name="data_note" column-name="note" ignore-case="true"/>
      <connected-node-attribute cmdb-attribute-name="name" column-name="ip_address"/>
    </or>
    </reconciliation-by-two-nodes>
  </cmdb-class>
</generic-db-adapter-config>
```

创建 TQL

TQL 是 **node** 通过包含链接连接到 **node**。此节点应标记为“Root”，如下所示。



要创建 TQL，请执行以下操作:

1. 转到“建模”>“建模工作室”。
2. 单击“新建”按钮，创建新查询。
3. 转到“CI 类型”选项卡，并将“Node”和“IpAddress”CI 类型拖放到 TQL 屏幕。
4. 将“Node”与“IpAddress”用 Containment 关系连接。
5. 右键单击“节点”元素，并选择“查询节点属性”。
6. 将“元素名称”更改为“Root”。
7. 转到“元素布局”选项卡。选择“特定属性”作为“属性”条件。从“可用属性”窗口中选择“名称”和“备注”，并将它们移到“特定属性”窗口。
8. 右键单击“IpAddress”元素，并选择“查询节点属性”。
9. 转到“元素布局”选项卡。选择“特定属性”作为“属性”条件。从“可用属性”窗口中选择“名称”，并将它移到“特定属性”窗口。
10. 保存 TQL。

创建集成点

按照以下方式创建集成点:

1. 转到“数据流管理”>“集成工作室”，然后单击“新建集成点”按钮。
2. 插入集成点的详细信息，然后单击“确定”。
3. 在“填入”选项卡中，选择“新建集成作业”按钮，然户添加之前创建的 TQL。
4. 保存集成点，然后单击“运行完全同步”按钮。

配置适配器 – 高级方法

这些配置文件位于 **C:\hp\UCMDB\UCMDBServer\content\adapters** 文件夹的 **db-adapter.zip** 包中，这个文件夹是在准备适配器包时提取的。有关详细信息，请参阅准备适配器包 (第 112 页)。

此任务包括以下步骤：

- 配置 orm.xml 文件 (第 118 页)
- 配置 reconciliation_rules.txt 文件 (第 121 页)

配置 orm.xml 文件

在此步骤中，将 CMDB 中的 CIT 和关系映射到 RDBMS 的表中。

1. 在文本编辑器中打开 **orm.xml** 文件。

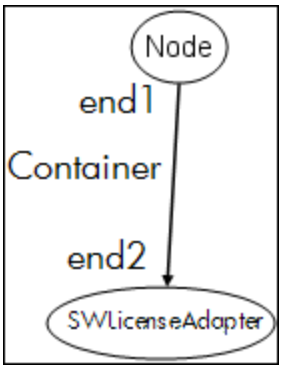
   默认情况下，此文件包含一个模板，可用于根据需要映射多个 CIT 和关系。

   > **备注:** 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：

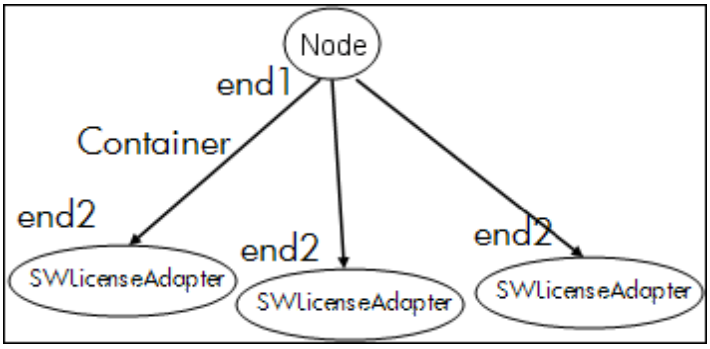
   - 一对一：

   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>
   ```

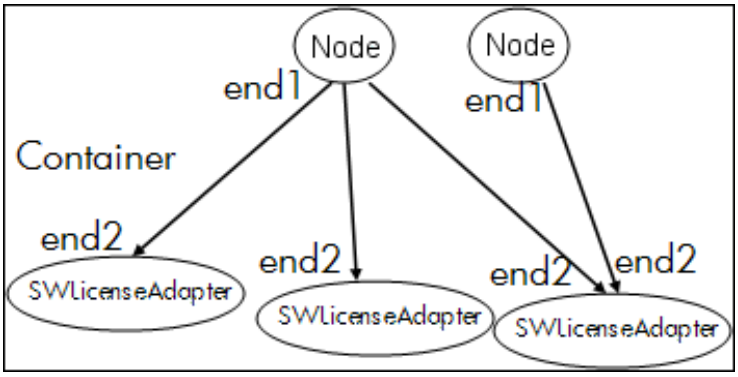
   - 多对一：



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 138 页)**。**

数据模型和 RDBMS 之间的实体映射示例：

备注: 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 109 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:

- CMDB CIT 中的属性名称和 RDBMS 中相应属性的名称:

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_rules.txt 文件

在此步骤中，将定义适配器调节 CMDB 和 RDBMS 时所用的规则（仅适用于使用了映射引擎的情况，以实现与版本 8.x 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]
   ```

   备注:
   - 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 145 页)。

实施插件

本任务描述了如何使用插件实施和部署常规 DB 适配器。

备注: 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 112 页)中所述的所有必需步骤。

1. 选项 1 – 编写基于 Java 的插件
   a. 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\db-adapter-framework** 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。

      备注: 有关开发插件的更多信息，可查看以下位置中的 **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\APIs\DBAdapterFramework_JavaAPI\index.html**
      - **C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\Federation_JavaAPI\index.html**

   b. 编写用于实施插件的 Java 接口的 Java 类。这些接口在 **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	FcmdbPluginForSyncGetFullTopology	getFullTopology
同步变更	FcmdbPluginForSyncGetChangesTopology	getChangesTopology
同步布局	FcmdbPluginForSyncGetLayout	getLayout
检索受支持的查询	FcmdbPluginForSyncGetSupportedQueries	getSupportedQueries
更改 TQL 查询的定义和结果	FcmdbPluginGetTopologyCmdbFormat	getTopologyCmdbFormat
更改 CI 的布局请求	FcmdbPluginGetCIsLayout	getCisLayout
更改链接的布局请求	FcmdbPluginGetRelationsLayout	getRelationsLayout
后推 ID	FcmdbPluginPushBackIds	getPushBackIdsSQL

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。

如果实施 **FcmdbPluginForSyncGetChangesTopology**，可使用两种不同方式报告变更:

- **始终报告整个根拓扑。**自动删除功能会按照此拓扑查找应当删除的 CI。在这种情况下，应使用以下语句启用自动删除功能:

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

```
    <CIT>object</CIT>
  </autoDeleteCITs>
```

- **报告每个已删除/更新的 CI 实例。**在这种情况下，应使用以下语句禁用自动删除机制：

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

c. 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 为 **federation_api.jar** 文件，此文件位于 **C:\hp\UCMDB\UCMDBServer\lib** 目录中。

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

2. 选项 2 – 编写基于 Groovy 的插件

a. 在适配器包配置文件下的“适配器管理”菜单中创建 Groovy 代码文件 (MyPlugin.groovy)。

b. 在 Groovy 类中，实现相应的接口。这些接口已在 db-interfaces.jar 文件中定义，请参阅上表。

3. 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]
# internal not mandatory plugin to change action on pushBackIds
[pushBackIds]
```

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。

4. 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“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”，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，验证“包管理器”是否可以识别包内容。

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《HP Universal CMDB 数据流管理指南》中的“适配器管理”。

创建集成点

将在此步骤中检查联合是否在正常运行。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“新建集成点/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支持查看 CIT 实例的视图。

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“如何创建模式视图”。

计算结果

将在此步骤中检查结果。

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”按钮计算结果。
4. 单击“预览”按钮，查看视图中的 CI。

查看结果

在此步骤中，将查看结果并调试在过程中遇到的问题。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，将查看拓扑报告。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 163 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 软件支持。

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。

插件版本 1.1 与 UCMDB 版本 9.01（或更高版本）以及 Eclipse IDE for Java EE Developers 版本 1.2.2.20100217-2310（或更高版本）相兼容。

此任务包括以下步骤:

- 先决条件 (第 126 页)
- 安装 (第 126 页)
- 准备工作环境 (第 126 页)
- 创建适配器 (第 127 页)
- 配置 CMDB 插件 (第 127 页)
- 导入 UCMDB 类模型 (第 127 页)
- 生成 ORM 文件 – 将 UCMDB 类映射到数据库表 (第 127 页)
- 映射 ID (第 128 页)

- 映射属性 (第 128 页)
- 映射有效链接 (第 128 页)
- 生成 ORM 文件 – 使用次级表 (第 129 页)
- 定义次级表 (第 129 页)
- 将属性映射到次级表 (第 129 页)
- 将现有 ORM 文件作为基础 (第 129 页)
- 从适配器导入现有 ORM 文件 (第 129 页)
- 检查 orm.xml 文件的正确性 - 内置的正确性检查 (第 130 页)
- 创建新集成点 (第 130 页)
- 将 ORM 文件部署到 CMDB (第 130 页)
- 运行示例 TQL 查询 (第 130 页)

1. 先决条件

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

2. 安装

   a. 从 **http://www.eclipse.org/downloads** 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**。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.zip** 从 **C:\hp\UCMDB\UCMDBServer\tools\db-adapter-eclipse-plugin\workspace** 提取到 **C:\Documents and Settings\All Users**。

   > **备注:** 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，则选择以下文件夹：**C:\Documents and Settings\All Users\workspace_gdb_sqlserver**。
   - MySQL，则选择以下文件夹：**C:\Documents and Settings\All Users\workspace_gdb_mysql**。
   - Oracle，则选择以下文件夹：**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包含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 25 页)中的说明创建适配器。

5. 配置 CMDB 插件

a. 在 Eclipse 中，单击“UCMDB”>“Settings”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，其路径应为 **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”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。

> **备注:** 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

执行步骤生成 ORM 文件 – 将 UCMDB 类映射到数据库表 (第 127 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：**<端 1 实体名>_<链接名>_<端 2 实体名>**。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 145 页)。

a. 映射链接类的 ID 属性，如映射 ID (第 128 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“Updateable”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，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“One to One”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 128 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）。确保按步骤生成 ORM 文件 – 将 UCMDB 类映射到数据库表 (第 127 页)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.xml 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 以图形方式对 ORM 文件进行编辑，然后将其重新部署到 UCMDB 计算机，如将 ORM 文件部署到 CMDB (第 130 页)所述。

16. 检查 orm.xml 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 145 页)。

18. 将 ORM 文件部署到 CMDB

    保存 **orm.xml** 文件，并通过单击“UCMDB”>“Deploy ORM”，将其部署到 UCMDB 服务器。此时，**orm.xml** 文件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“Operation Result”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. 在“建模工作室”中定义一个查询（而非视图）。有关详细信息，请参阅《HP Universal CMDB 建模指南》中的“建模工作室”。
    b. 使用在步骤创建新集成点 (第 130 页)中创建的适配器创建集成点。有关详细信息，请参阅《HP Universal CMDB 数据流管理指南》中的“新建集成点/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，在“设置”对话框中使用该实例查询名称。有关详细信息，请参阅配置 CMDB 插件 (第 127 页)中的步骤。
    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。

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。

本节描述了以下配置文件：

- adapter.conf 文件 (第 132 页)
- simplifiedConfiguration.xml 文件 (第 132 页)
- orm.xml 文件 (第 134 页)
- reconciliation_types.txt 文件 (第 145 页)

- reconciliation_rules.txt 文件（用于向后兼容）(第 145 页)
- transformations.txt 文件 (第 146 页)
- discriminator.properties 文件 (第 147 页)
- replication_config.txt 文件 (第 148 页)
- fixed_values.txt 文件 (第 148 页)
- Persistence.xml 文件 (第 149 页)

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 132 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 132 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 134 页)。
- **reconciliation_rules.txt**。包含调节规则。有关详细信息，请参阅reconciliation_rules.txt 文件（用于向后兼容）(第 145 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，或者进行相反转换的转换器。有关详细信息，请参阅transformations.txt 文件 (第 146 页)。
- **Discriminator.properties**。此文件将每个支持的 CI 类型映射到一个由可能相应值组成的逗号分隔列表。有关详细信息，请参阅discriminator.properties 文件 (第 147 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 148 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 148 页)。

Hibernate 配置

- **persistence.xml**。用于替代现成的 Hibernate 配置。有关详细信息，请参阅Persistence.xml 文件 (第 149 页)。

启用适配器的临时表支持

启用临时表可支持适配器以更高的效率使用远程数据库，从而减轻数据库和网络的压力，同时还能增强性能。

要启用常规数据库适配器中的临时表支持，则必须满足以下条件：

- **已提供连接数据库的凭据，包括创建、修改和删除临时表的权限。**
- **配置 adapter.conf 配置文件中的以下设置：**

  temp.tables.enabled=true

  performance.enable.single.sql=true

备注: 仅 Microsoft SQL 和 Oracle 支持临时表。

adapter.conf 文件

此文件包含以下设置:

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

  备注: 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。

  备注: 建议不要替代 Hibernate 配置。

- **performance.memory.id.filtering=true**。在某些情况下，GDBA 执行 TQLS 时，可能会检索到大量 ID 并将通过 SQL 其发回到数据库中。为避免这种过度工作并提高性能，GDBA 会尝试读取整个视图/表格，并筛选内存中的结果。
- **id.reconciliation.cmdb.id.type=string/bytes**。使用“ID 调节”映射常规数据库适配器时，可以通过更改 **META-INF/ adapter.conf** 属性将 **cmdb_id** 映射到 **string** 或 **bytes/raw** 列类型。
- **performance.enable.single.sql=true**。此参数为可选参数。如果文件中未出现此参数，则其默认值为 **true**。为 **true** 时，常规数据库适配器会尝试为所执行的每个查询（填入查询或联合查询）生成一个 SQL 语句。使用单一 SQL 语句可以改善常规数据库适配器的性能和内存消耗情况。为 **false** 时，常规数据库适配器会生成多个 SQL 语句，与使用单一语句相比，这将需要更多的执行时间和更多的内存。在以下情况下，即使此属性设置为 **true**，适配器也不会生成单一 SQL 语句:
  - 适配器所连接的数据库不在 Oracle 或 SQL Server 上。
  - 所执行的 TQL 包含并非 0..* 和 1..* 的基数条件（例如，存在 2..* 或 0..2 等基数条件）。
- **in.expression.size.limit=950**（默认值）。当达到参数列表的大小限制时，此参数将拆分已执行 SQL 语句的“IN”表达式。
- **stringlist.delimiter.of.<CIT 名称>.<属性名称>=<分隔符>**。为了将字符串列表属性映射到常规数据库适配器中的数据库列，需要将属性映射到包含已连接值列表的字符串列。例如，如果要映射 CI 类型为 **policy** 的属性 **policy_category**，且字符串列包含以下值的列表：value1##value2##value3（定义值 1、值 2、值 3 这三个值的列表），请使用以下设置：**stringlist.delimiter.of.policy.policy_category=##**。
- **temp.tables.enabled=true**。支持您使用临时表提高性能。只有在启用 **performance.enable.single.sql** 时才可用（仅在 Microsoft SQL 和 Oracle 中受支持）。可能需要数据库服务器中的特定权限。
- **temp.tables.min.value=50**。定义使用临时表所需的条件值（或 ID）的数目。

simplifiedConfiguration.xml 文件

本文件用于将 UCMDB 类简单地映射到数据库表。要访问用于编辑文件的模板，请转到“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 133 页)
- 局限性 (第 134 页)

simplifiedConfiguration.xml 文件模板

- **CMDB-class-name** 属性是多节点类型（TQL 中联合 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节。在此示例中，调节过程使用了两个节点。
- **connected-node-CMDB-class-name**。调节 TQL 中需要的次要类类型。
- **CMDB-link-type**。调节 TQL 中需要的关系类型。
- **link-direction**。调节 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">
  ```

  调节表达式采用 OR 形式，并且每个 OR 都包含 AND。
- **is-ordered**。确定调节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）中检索到调节属性，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。

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.impl. GenericEnumTransformer
(generic-enum-transformer-example.xml)" to-CMDB-onverter="com.
mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl. GenericEnumTransformer(generic-enum-transformer-example.xml)" />

<attribute CMDB-attribute-name="[CMDB_attribute_name]" column-name="[column_name]" />

<attribute CMDB-attribute-name="[CMDB_attribute_name]" column-name="[column_name]" />

</class>

</generic-DB-adapter-config>

局限性

- **可用于仅映射包含一个节点的** 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分层，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 目录中。

要编辑已部署适配器的 XML **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:

- orm.xml 文件模板 (第 134 页)
- 多个 ORM 文件 (第 138 页)
- 命名约定 (第 138 页)
- 使用内嵌式 SQL 语句而非表名称 (第 138 页)
- orm.xml 架构 (第 138 页)
- 创建 orm.xml 文件的示例 (第 141 页)
- 为每个远程产品版本创建一个特定的 orm.xml (第 145 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。

```
  <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>
```

对于具有鉴别器的单个表格继承，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>
```

对于具有鉴别器的单表格继承，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 **end1 Type_linkType_end2 Type**。在以下示例中，**end1 Type** 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。包含 end1 ID（目标实体 ID）的列。

referenced-column-name。目标实体 (node) 中的列名称，包含在连接列中使用的 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>
```

node attribute。以下是如何添加节点属性的示例。

```
<entity class="generic_db_adapter.host_node">
 <discriminator-value>host_node</discriminator-value>
  <attributes/>
</entity>
<entity class="generic_db_adapter.nt">
 <discriminator-value>nt</discriminator-value>
  <attributes>
   <basic name="nt_servicepack">
    <column updatable="false" insertable="false" name="specific_type_value"/>
   </basic>
  </attributes>
</entity>
```

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 实体的名称应为 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 中找到完整的架构。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。

元素名称和路径	描述	属性
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）	
package (entity-mappings)	要包含映射类的 Java 包的名称。应始终包含文本 `generic_db_`	1. **名称：** name

元素名称和路径	描述	属性
	adapter。	**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 中的链接，则此实体的名称应采用以下格式 <end_1>_<link_name>_<end_2>。例如，node_composition_cpu 定义将映射到节点和 CPU 之间的组合链接的实体。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 2. **名称：**class **描述：**要为此 DB 实体创建的 Java 类的完全限定名称。Java 类的包的名称应与 package 元素中指定的名称相同。您不能将 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，这样接口将变为 generic_db_adapter.gdba_interface。 **是否必需？：**必需 **类型：**字符串
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其所属的架构，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其所属的架构，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

元素名称和路径	描述	属性
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
inheritance (entity-mappings > entity)	如果当前实体是某组 DB 实体的父实体，则使用此元素对其进行标记。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：** 鉴别器列的名称。 **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。	
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 为 1 到 9 之间的数字。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。

元素名称和路径	描述	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。	1. **名称：** name **描述：** 字段的名称。 **是否必需？：** 必需 **类型：** 字符串 2. **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
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。	1. **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` 2. **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。	1. **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 2. **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

创建 orm.xml 文件的示例

此处提供的示例演示如何创建 orm.xml 文件。在此示例中，远程数据库中的 SQL 表映射到 UCMDB 中的 CI 类型。

假设远程数据库中的表使用以下格式，使用节点填充“Hosts”表，使用 IP 地址填充 **IP_Addresses** 表，并在节点和 IP 地址之间创建如下所示的链接：

Hosts 表

host_name	host_id
Test1	1
Test2	2
Test3	3

IP_Addresses 表

ip_address	ip_id
10.1.1.1	1
10.2.2.2	2
10.3.3.2	3
10.4.4.4	4

Host_IP_Link 表（ “节点”和“IP 地址”之间的链接 ）

host_id	ip_id
1	1
2	2
2	3
3	4

Hosts 表的主键为 **host_id** 字段，**IP_Addresses** 表的主键为 **ip_id** 字段。在 **Host_IP_Link** 表中，**host_id** 和 **ip_id** 是来自 **Hosts** 表和 **IP_Addresses** 表的外键。

按照上面的表，创建 **orm.xml** 文件，如下所示：此示例中使用的实体为 **node**、**ip_address** 和 **node_containment_ip_address**。

1. 通过映射来自 **Hosts** 表中的 **host_id** 创建 **node** 实体，如下所示：

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

```
 <package>generic_db_adapter</package>
 <entity class="generic_db_adapter.node">
     <table name="Hosts"/>
     <attributes>
<id name="id1">
     <column updatable="false" insertable="false" name=
         "host_id"/>
     <generated-value strategy="TABLE"/>
</id>
<basic name="name">
     <column updatable="false" insertable="false" name=
         "host_name"/>
</basic>
     </attributes>
 </entity>
```

实体 **class** 必须是 UCMDB 中已有的 CI 类型。表 **name** 是指包含 ID 和主机信息的数据库内的表。ID 属性是标识特定主机所必需的属性，将用于稍后的映射中。在此示例中，此实体的 **name** 属性使用 Hosts 表中的 **host_name** 列填充。

2. 对于下一个实体，请从 Interfaces 表中映射 IP 地址：

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

3. 接下来，“节点”和“IP 地址”之间的链接必须通过映射表的方式创建，并引用 **ip_id** 字段（即使

该链接可以根据需要同时引用 **host_id** 和 **ip_id** 字段）。

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

容器的实体名称采用以下格式：[端 1 CIT]_[链接 CIT]_[端 2 CIT]。对于此示例，由于链接 CI 类型为“Containment”，因此容器的实体名称为 **node_containment_ip_address**，实体类为 **generic_db_adapter.node_containment_ip_address**。ID 在此代码块中是必需的，虽然此示例只使用了一个接口 ID，但两个列都可以引用 id1 和 id2。相关代码为：

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

此链接的两端为“many-to-one”和“one-to-one”，这表示每个 IP 地址都链接到一个节点，但一个节点可以链接到多个 IP 地址。包含的列来自“Links”表，并引用“Hosts”表和“Interfaces”表。

为每个远程产品版本创建一个特定的 orm.xml

可以配置特定的 **orm.xml** 文件，使适配器对指定的远程产品版本使用特定的 **orm.xml**。例如，如果远程数据存储有两个产品版本 x 和 y，则每个版本可以具有不同的实体映射。

要为每个远程产品版本配置特定的 orm.xml，请执行以下操作：

1. 将一个参数添加到名为 **version** 的 **adapter.xml** 文件，并将可能的版本值指定为 **valid-values**。
2. 在适配器包的 META-INF 文件夹下，创建名为 **VersionOrm** 的文件夹。
3. 在 **VersionOrm** 文件夹中，为每个特定版本创建 **orm.xml** 文件。文件名应包含版本前缀。例如，如果版本名为 **x**，则文件名应为 **x_orm.xml**。

备注: 可为任何远程产品版本加载 META-INF 文件夹中的 **orm.xml** 文件，无论您是否已为远程产品版本创建特定的 **orm.xml** 文件。它还可以拥有以相同方式为所有版本映射的实体。

reconciliation_types.txt 文件

从 UCMDB 10.00 开始，**reconciliation_types.txt** 文件不再相关。任何 CIT 都可用于调节。联合引擎自动执行映射。

reconciliation_rules.txt 文件（用于向后兼容）

如果要在适配器中配置 DBMappingEngine 后执行调节，则可以使用本文件配置调节规则。如果不使用 DBMappingEngine，则使用常规 UCMDB 调节机制，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]

多节点以多节点名称（与 TQL 查询中的联合数据库 CIT 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 ordered expression 而不是 expression。

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 仅适用于调节 TQL 查询包含两个节点而非一个多节点的情形。在这种情况下，调节 TQL 查询为 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。

调节规则的类型

调节规则的形式为 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 nodeAND/ORname from ip_address 标识节点）。

可以通过以下选项查找匹配项:

- **Ordered match。**从左向右读取调节表达式。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。
- **Regular match。**如果 OR 子表达式中的一个是相等的，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。

对于复杂调节，如果在类模型中按照多个含有关系的 CIT（如 node）对调节实体进行建模，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。

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

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

要调节结果，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，或相反。

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）:

DB IP_Address	
PK	node_id
	名称

DB Node	
PK	ip_id
	名称

DB Ticket	
PK	ticket_id
	node_id

备注: 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.FcmdbDalTransformerTo ExternalDB** 的类的完全限定名称。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 151 页)。

discriminator.properties 文件

本文件可将每个受支持的 CI 类型（也用作 orm.xml 中的鉴别器值）映射到鉴别器列的相应可能值列表（以逗号分隔），或映射到用于匹配鉴别器列可能值的条件。

如果使用条件，请使用以下语法：like(condition)，其中 condition 是字符串，可以包含以下通配符：

- **%**（百分比符号）- 可用于匹配任意长度的任意字符串，包括零长度的字符串
- **_**（下划线）- 可用于匹配单个字符

例如，like(%unix%) 可匹配 unix、Linux、unix-aix 等等。like 条件只适用于字符串列。

通过声明 'all-other'，还可以将单个鉴别器值映射到不属于其他区分器的任意值。

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

鉴别器映射示例：

例如，适配器支持 CI 类型 node、nt 和 unix，并且数据库包含一个名为 t_nodes 的表，表中有一个名为 **type** 的列。如果 type 为 10001，则行表示节点；如果 type 为 10004，则行表示 unix 计算机，等等。**discriminator.properties** 文件可能类似如下所示：

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

orm.xml 文件包括以下代码：

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

```
    <table name="t_nodes" />
    ...
    <inheritance strategy="SINGLE_TABLE" />
    <discriminator-value>node</discriminator-value>
    <discriminator-column name="type" />
    ...
</entity>
<entity class="generic_db_adapter.nt" name="nt">
    <discriminator-value>nt</discriminator-value>
    <attributes>
</entity>
<entity class="generic_db_adapter.unix" name="unix">
    <discriminator-value>unix</discriminator-value>
    <attributes>
</entity>
```

然后，discriminator_column 属性按如下方式进行计算：

- 如果对于某个特定条目，**type** 包含 10002 或 10003，则该条目将映射到 **nt** CIT。
- 如果对于某个特定条目，**type** 包含 10001 或 10005，则该条目将映射到 **node** CIT。
- 如果对于某个特定条目，**type** 以 2 开头，则该条目将映射到 **unix** CIT。
- 如果 **type** 列中包含任何其他值，则映射到 **mainframe** CIT。

备注: 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 155 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]

文件也支持常量列表。要定义常量列表，请使用以下语法：

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

Persistence.xml 文件

本文件用于替代默认 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 SQL 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（有关详细信息，请参阅 http://www.hibernate.org）。

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>
```

使用 NT 身份验证连接到数据库

可以连接到需要 NT 身份验证的 MS SQL Server。要执行此操作，则需要可解析域的驱动程序（即 jTDS JDBC 驱动程序）。

身份验证将根据指定的参数（域、用户名、密码）执行，而不使用当前运行的进程 NT 凭据。

1. **在 persistence.xml 中，按如下所示编辑以下属性：**

```
<!--The driver class name"/-->
<property name="hibernate.connection.driver_class"
value="net.sourceforge.jtds.jdbc.Driver"/>
<property name="hibernate.connection.url" value="jdbc:jtds:sqlserver://[host name]:
[port];DatabaseName=[database name];domain=[the domain]"/>
<!--DB login credentials"/-->
<property name="hibernate.connection.username" value="[username]"/>
<property name="hibernate.connection.password" value="[password]"/>
```

2. **将 JDBC 驱动程序文件置于以下位置：<探测器安装文件夹>\lib\。**
3. **重新启动探测器。**

配置 Persistence.xml 文件以便 SCCM 集成使用 NTLM 身份验证

注意：此部分仅适用于 SCCM 集成。

要使 SCCM 集成使用 NTLM 身份验证，请按以下操作配置 persistence.xml 文件：

1. **将 JDBC 驱动程序文件置于以下位置：<探测器安装文件夹>\lib\。**

   例如，您可以将来自 http://sourceforge.net/projects/jtds/files/ 的 **jtds-1.3.1.jar** 文件放到 **DataFlowProbe\lib** 文件夹中。
2. **启动服务器和探测器。**
3. **在 UCMDB 中，转到“数据流管理”>“适配器管理”>“SCCMAdapter”。**
4. **在“资源”窗格中，在“包”>“SCCMAdapter”>“配置文件”文件夹中选择 SCCM 适配器配置文件。**
5. **在 adapter.conf 文件中，设置 dal.use.persistence.xml=true。**
6. **在 persistence.xml 文件中，添加以下内容：**

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://java.sun.com/xml/ns/persistence
 http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
  <persistence-unit name="GenericDBAdapter">
    <properties>
      <!-- added to fix:org.hibernate.HibernateException:'hibernate.dialect' must be set when
no Connection avalable -->
      <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>

      <!--The driver class name"/-->
      <property name="hibernate.connection.driver_class"
value="net.sourceforge.jtds.jdbc.Driver"/>
      <property name="hibernate.connection.url" value="jdbc:jtds:sqlserver://<DB_
```

```
host>:<port>;DatabaseName=<DB_name>;domain=<domain_name> "/>
    </properties>
  </persistence-unit>
</persistence>
```

注意： 使用连接 URL 替换突出显示的部分。

7. **persistence.xml** 文件中不需要用户名或密码。
8. 转到“数据流管理”>“集成工作室”，然后单击“新建集成点”  按钮。
9. 为必需字段提供值。

   当要求输入凭据 ID 时，执行以下操作：
   a. 在“选择凭据”对话框中，从左侧“协议”窗格中选择“通用 DB 协议 (SQL)”。
   b. 在右侧“凭据”窗格中，单击“为选定协议类型创建新连接详细信息”  按钮。
   c. 在新对话框中选择 **MicrosoftSQLServerNTLM** 作为数据库类型。
   d. 输入端口号。
   e. 按以下格式提供用户名：**域\用户名**。
   f. 提供密码。

现成的转换器

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：

- 现成的转换器 (第 151 页)
- SuffixTransformer 转换器 (第 154 页)
- PrefixTransformer 转换器 (第 154 页)
- BytesToStringTransformer 转换器 (第 154 页)
- StringDelimitedListTransformer 转换器 (第 155 页)
- 自定义转换器 (第 155 页)

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、返回空值或抛出异常。

转换器可使用区分大小写方法或不区分大小写方法对两个字符串执行比较。默认行为为区分大小写。要将其定义为不区分大小写，请在 enum-transformer 元素中使用：case-sensitive="false"。

每个实体属性使用一个 XML 映射文件。

备注: 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:

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

```
            <xs:enumeration value="xml"/>
            <xs:enumeration value="bytes"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="non-existing-value-action" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="return-null"/>
            <xs:enumeration value="return-original"/>
            <xs:enumeration value="throw-exception"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="case-sensitive" use="optional">
        <xs:simpleType>
          <xs:restriction base="xs:boolean">
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="value">
    <xs:complexType>
      <xs:attribute name="cmdb-value" type="xs:string" use="required"/>
      <xs:attribute name="external-db-value" type="xs:string" use="required"/>
      <xs:attribute name="is-cmdb-value-null" type="xs:boolean" use="optional"/>
      <xs:attribute name="is-db-value-null" type="xs:boolean" use="optional"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

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>
```

将外部值或 CMDB 值转换为空值的示例：

在本示例中，远程数据库中的值 **NNN** 转换为 CMDB 数据库中的一个空值。

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

在本示例中，CMDB 中的 **OOO** 值转换为远程数据库中的空值。

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

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**。

StringDelimitedListTransformer 转换器

此转换器用于将单个字符串列表转换为 CMDB 中的整数/字符串列表。

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

自定义转换器

您可以从零开始编写自己的自定义转换器。这样就可以根据自己的需求创建转换器。

编写自动义转换器的方式有两种：

1. **编写已编译的 Java 转换器**
   a. **采用 Java IDE（如 Eclipse、Intellij 或 Netbeans）创建 Java 项目。**
   b. **将 federation-api.jar 和 db-interfaces.jar 添加到类路径。**
   c. **创建用于实现以下接口的 Java 类（从 db-interfaces.jar）：**
      - FcmdbDalTransformerFromExternalDB
      - FcmdbDalTransformerValuesToExternalDB
      - FcmdbDalTransformerInit
   d. **编译项目并创建 jar 文件。**
   e. **将 jar 文件置于适配器包中（位于 adapterCode\<适配器 ID> 下面）**
   f. **部署包。**
   g. **将新的转换器类名称添加到 transformations.txt 文件。**
2. **编写 Groovy（基于脚本）转换器**
   在原始 GDBA 包中找到示例 GroovyExampleTransformer.groovy。
   a. **在适配器包中创建 Groovy 文件（位于 adapterCode\<适配器 ID> 下面）。您可以使用“适配器管理”菜单直接执行此操作。**
   b. **创建用于实现以下接口的 Groovy 类（从 db-interfaces.jar）：**
      - FcmdbDalTransformerFromExternalDB
      - FcmdbDalTransformerValuesToExternalDB
      - FcmdbDalTransformerInit
   c. **将新的转换器 Groovy 类名称添加到对应的 transformations.txt 文件。**

备注: Groovy 是一种可扩展 Java 的脚本语言。常规的 Java 代码也是有效的 Groovy 代码。

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变更的插件，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。
- 用于更改“后推 ID”操作的可选内部插件。

有关实施和部署插件的详细信息，请参阅实施插件 (第 121 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 (第 156 页)
- 单个节点调节 (第 156 页)
- 两个节点调节 (第 159 页)
- 使用包含多个列的主键 (第 161 页)
- 使用转换功能 (第 162 页)

用例

TQL 查询为:

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。

单个节点调节

在本示例中，调节按照 name 属性运行。

简单定义

调节通过 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 134 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_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 145 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。

两个节点调节

在本示例中，将根据包含不同变量的 node **和** ip_address **的** name **属性计算调节。**

调节 TQL 查询为 node > (containment) > ip_address。

简单定义

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

```
<?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 运算来计算调节:**

```
<?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 **计算调节**:

```
<?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节表达式，所以所有调节表达式将使用同一版本。

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 145 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。

使用包含多个列的主键

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 134 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.impl.GenericEnumTransformer(generic-enum-transformer-example.
       xml)" to-CMDB-converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.
       transform.impl.GenericEnumTransformer(generic-enum-transformer-example.
       xml)" />
      <connected-node-attribute CMDB-attribute-name="name"
       column-name="Device_PreferredIPAddress" />
    </or>
  </reconciliation-by-two-nodes>
</CMDB-class>
```

高级定义

仅 transformation.txt 文件有变更。

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.
GenericEnumTransformer(generic-enum-transformer-example.xml)] from_DB_class
[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(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 (第 163 页)
- 日志位置 (第 164 页)

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节 TQL 查询，以及在连接阶段进行比较的调节拓扑的详细信息。

  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（检查调调节），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 105 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 – 开发常规数据库适配器

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

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 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 CMDBid 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。

第 6 章: 开发 Java 适配器

本章包括:

- 联合框架概述 166
- 适配器与联合框架的映射交互 170
- 联合 TQL 查询的联合框架 170
- 联合框架、服务器、适配器和映射引擎之间的交互 171
- 用于填入的联合框架流 180
- 适配器接口 181
- 调试适配器资源 182
- 为新外部数据源添加适配器 182
- 创建示例适配器 188
- XML 配置标记和属性 189
- DataAdapterEnvironment 界面 191

联合框架概述

备注:

- 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。

“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。
- **填入**。将数据（拓扑数据和 CI 属性）从外部数据源填充到 CMDB。
- **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 (第 166 页)
- 数据推送 (第 168 页)
- 填入 (第 168 页)

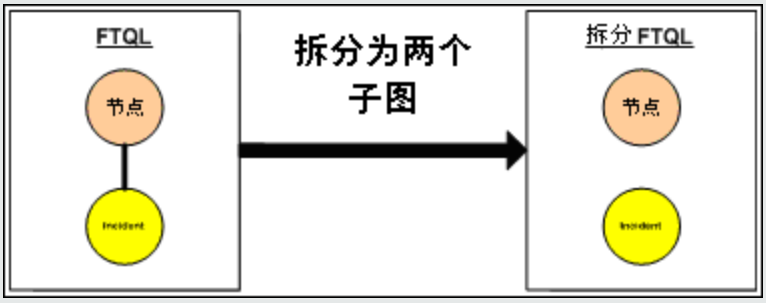
动态联合

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。

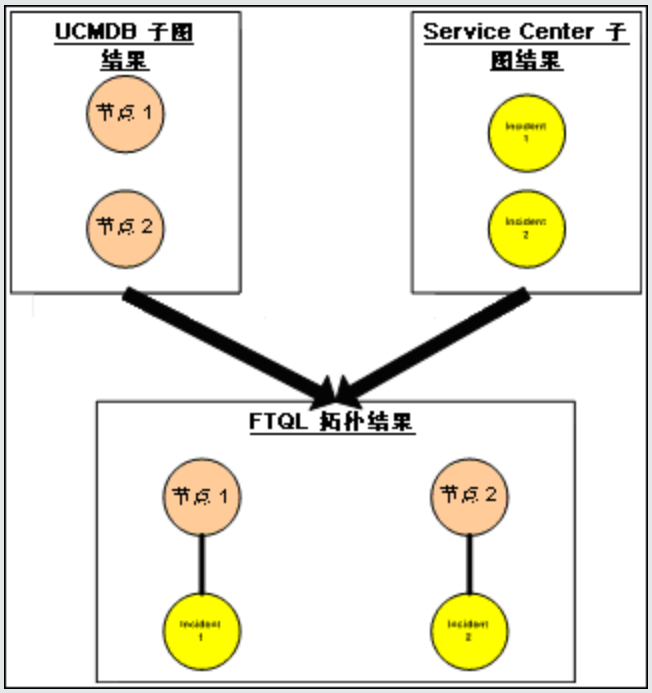
联合的 TQL 查询使用表示外部数据库的适配器，在来自不同外部数据库的 CI 和 UCMDB CI 之间创建相应的外部关系。

动态联合流示例：

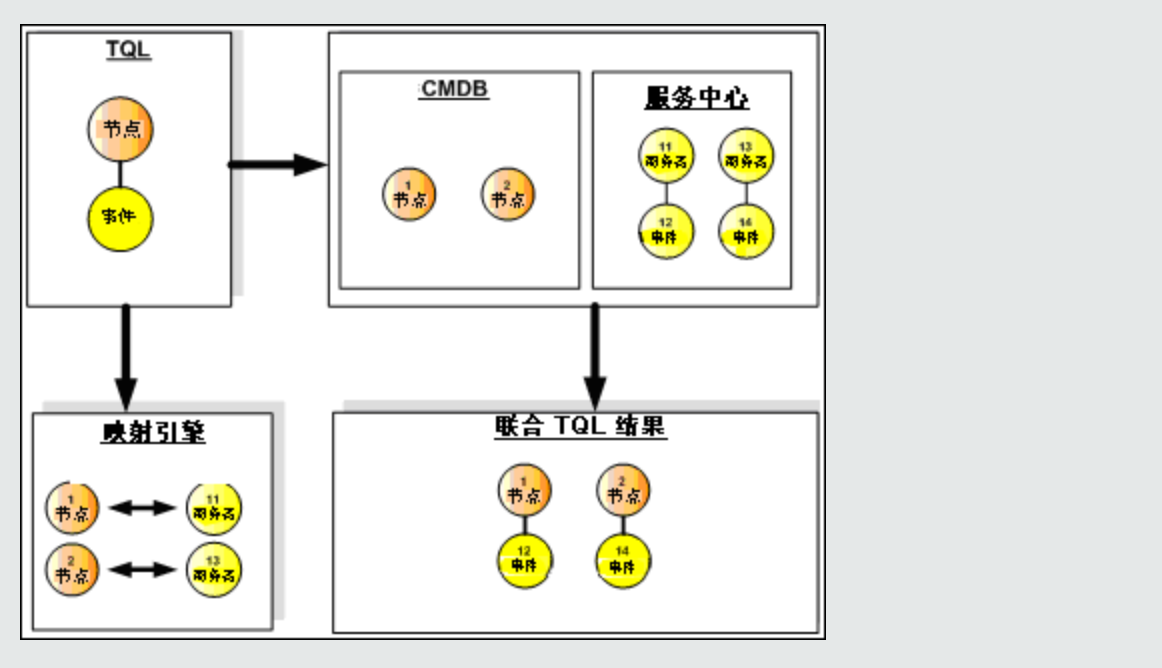
1. 联合框架会将联合的 TQL 查询分割成多个子图，子图中的所有节点都将引用相同的数据库。每个子图通过虚拟关系连接到其他子图（但自身并不包含虚拟关系）。



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将当前本地 CMDB 中的数据同步到远程服务或目标数据库。

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来执行，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变更的结果。
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中的“源”数据源。要对数据推送流实施新适配器，只需实施“目标”适配器即可。

填入

您可以使用填入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探测器中，其推送过程与搜寻作业流的过程类似。

要为填入流程实施新适配器，只需实施源适配器即可，因为 Data Flow Probe 将充当目标。

填入流程中的适配器将在探测器上执行。所以，应在探测器而不是 CMDB 上完成调试和登录。

填入流程基于查询名称，这意味着将在源数据库和 Data Flow 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，查询名称为 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变更的结果。
- 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变更的结果。
- 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。
- 探测器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变更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。
- 探测器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。

基于实例的填入流程

如果将适配器定义为支持基于实例的流（通过 **<instance-based-data>** 标记，如 XML 配置标记和属性 (第 189 页)中所述），则填入引擎将自动查找实例中已删除的 CI，并从 UCMDB 中将其删除（假设特定填入作业允许删除）。每个实例都必须具有根 CI，并使用名称“根”在 TQL 定义中标记出来。每次传递根 CI 时，将其整个实例（连接到实例的所有 CI）与上次发送到 UCMDB 的实例进行比较，并从 UCMDB 删除曾连接到根但现在不再连接的任何 CI。为了使适配器能正确支持基于实例的流，在整个实例中对任何 CI 或属性做出的变更都必须触发将整个实例重新发送到 UCMDB。

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 170 页)
- 适配器 assist 方法 (第 170 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 Data Flow 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。（只能在探测器上执行此方法。）
- 第三种方法仅适用于联合和填入流程，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 TQL 查询的联合框架

本节包括以下主题:

- 定义和术语 (第 170 页)
- 映射引擎 (第 171 页)
- 联合适配器 (第 171 页)

请参阅联合框架、服务器、适配器和映射引擎之间的交互 (第 171 页)，查看演示联合框架、UCMDB、适配器和映射引擎之间交互的图表。

定义和术语

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

- **ID 调节**。只有当外部数据库包含调节对象的 CMDB ID 时，才可使用此规则。
- **属性调节**。只有能够按照调节 CI 类型的属性完成匹配时，才可使用此规则。

- **拓扑调节。**在需要其他 CIT（不仅仅是调节 CIT）属性才能对调节 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节。

调节对象。对象由适配器根据所接收的调节数据创建。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。

调节 CI 类型。表示调节对象的 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 调节对象和外部 CI 调节对象，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节数据。虚拟关系的一端必须是 CMDB，此端为调节类型。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 的调节对象。

- **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。
- **调节对象数据。**一种辅助数据，由联合框架用于连接 CMDB CI 和外部数据。每个调节对象引用一个外部 CI。调节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）。调节对象应与调节数据接收的适配器相匹配。调节对象可以是以下三种类型之一：IdReconciliationObject、PropertyReconciliationObject 或 TopologyReconciliationObject。

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

联合框架、服务器、适配器和映射引擎之间的交互

下图演示了联合框架、UCMDB 服务器、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。

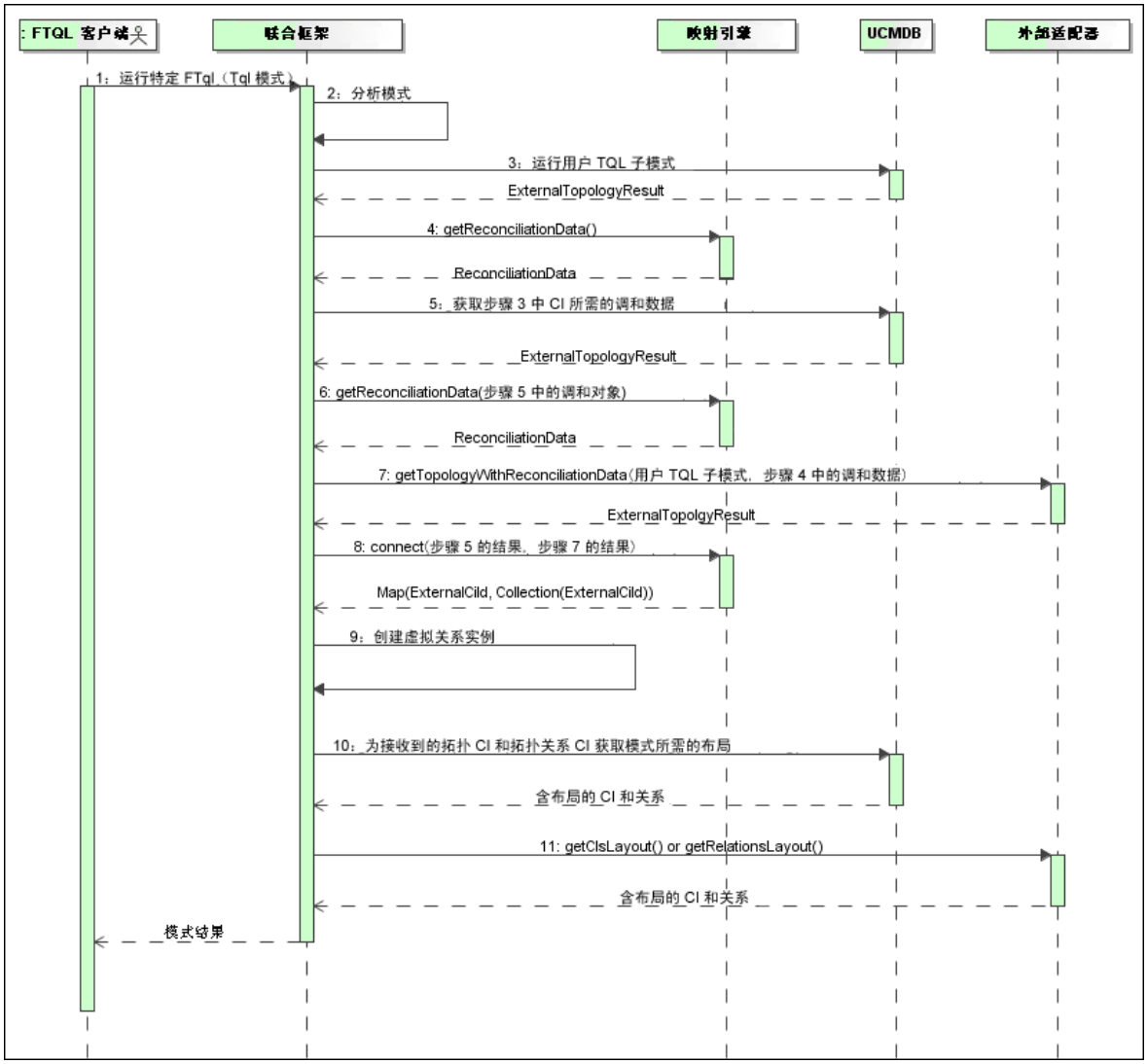
本节包括以下主题：

- 从服务器端开始计算 (第 171 页)
- 从外部适配器端开始计算 (第 174 页)
- 联合 TQL 查询的联合框架流示例 (第 175 页)

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。

从服务器端开始计算

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。

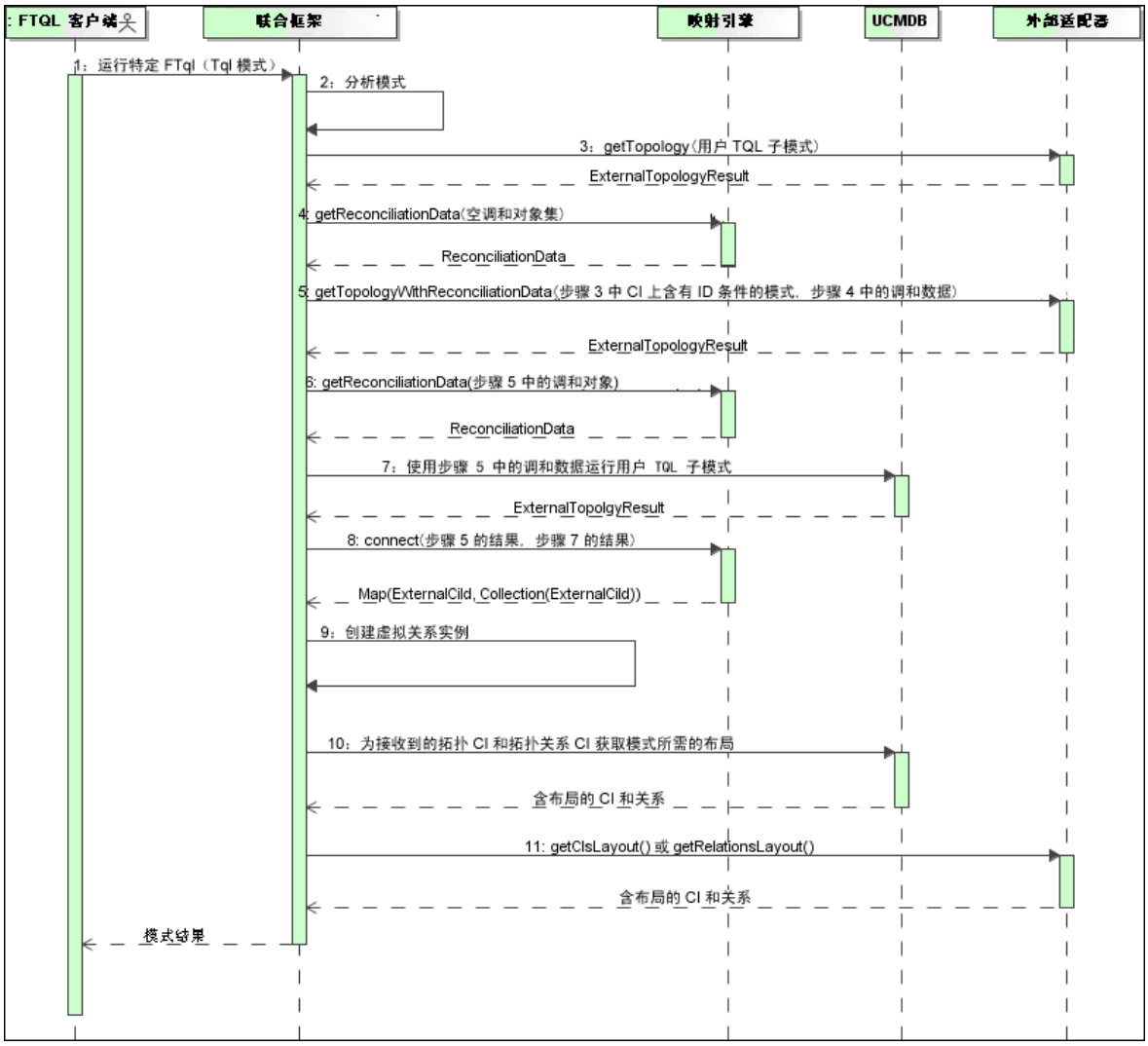


对图中编号的说明如下:

编号	说明
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节数据。在此阶段，reconciliationObject 参数为空，也就是说，在此调用中未向调节数据添加任何条件。返回的调节数据定义了将 UCMDB 与外部数据库中的调节 CI 进行匹配所需的数据。调节数据可以是以下类型之一:

编号	说明
	• **IdReconciliationData**。按照 ID 调节的 CI。 • **PropertyReconciliationData**。按照某个 CI 的属性调节的 CI。 • **TopologyReconciliationData**。按照拓扑（例如，如果要调节节点 CI，则还需要 **IP** 的 IP 地址）调节的 CI。
5	联合框架从 UCMDB 请求在步骤 3 (第 172 页)中接收的虚拟关系末端 CI 的调节数据。
6	联合框架调用映射引擎检索调节数据。在此阶段（与步骤 3 (第 172 页)对照），映射引擎接收步骤 5 (第 173 页)中的调节对象作为参数。映射引擎将接收的调节对象转换成调节数据条件。
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 (第 173 页)中的调节数据作为参数。
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 (第 173 页)中从 UCMDB 接收的外部拓扑结果；secondResult 参数是在步骤 7 (第 173 页)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



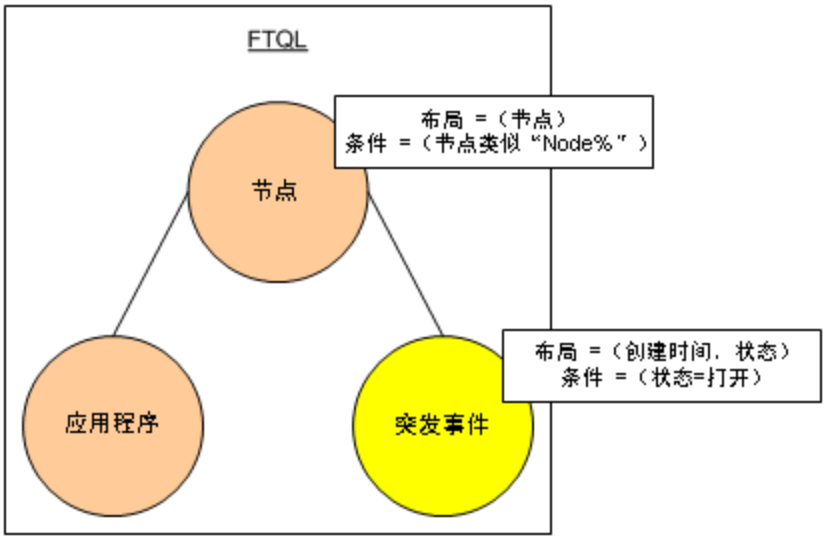
对图中编号的说明如下:

编号	说明
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节数据，预计返回的 ExternalTopologyResult 不包含任何调节对象。
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节数据。在此阶段，reconciliationObjects 参数为空，也就是说，在此调用中未向调节

编号	说明
	数据添加任何条件。返回的调节数据定义了将 UCMDB 与外部数据库中的调节 CI 进行匹配所需的数据。调节数据可以是以下类型之一: • **IdReconciliationData**。按照 ID 调节的 CI。 • **PropertyReconciliationData**。按照某个 CI 的属性调节的 CI。 • **TopologyReconciliationData**。按照拓扑（例如，如果要调节节点 CI，则还需要 **IP** 的 IP 地址）调节的 CI。
5	联合框架从外部数据库请求步骤 3 中接收的 CI 的调节对象。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 条件和调节数据。
6	联合框架调用映射引擎检索调节数据。在此阶段（与步骤 3 对照），映射引擎接收步骤 5 中的调节对象作为参数。映射引擎将接收的调节对象转换成调节数据条件。
7	联合框架使用步骤 6 中的调节数据从 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。

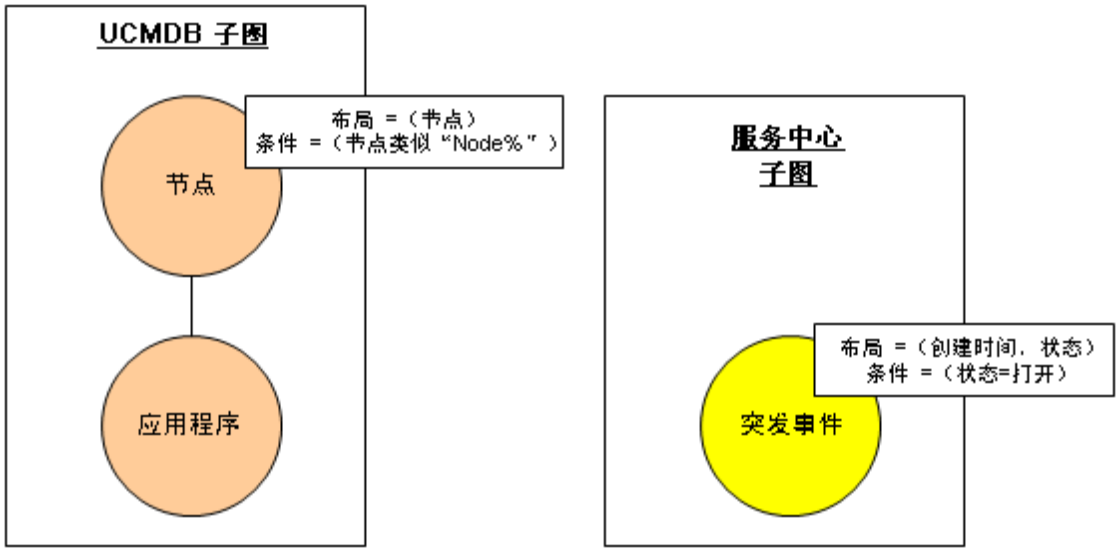
联合 TQL 查询的联合框架流示例

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。这些属性是在 UCMDB 中用于标识 ServiceCenter 的节点的调节属性。

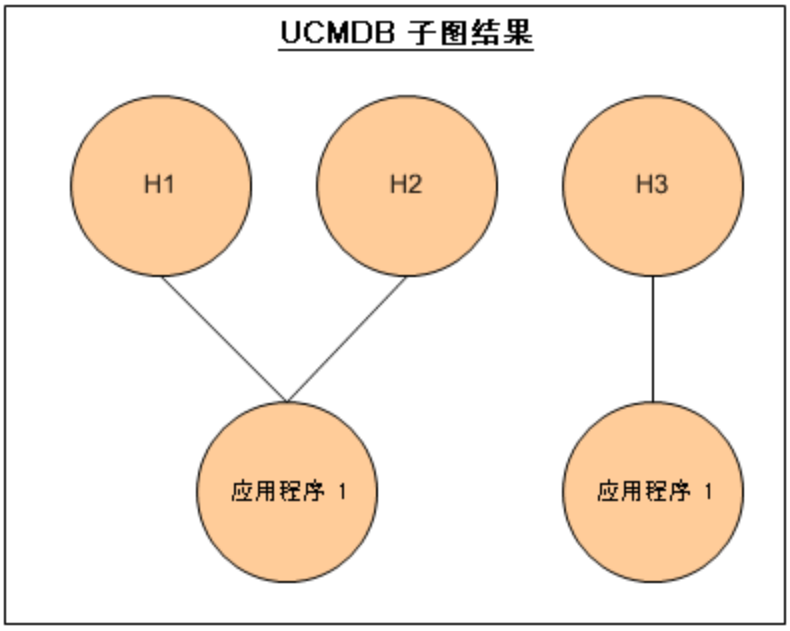


备注: 对于属性联合，将调用适配器的 **getTopology** 方法。在用户 TQL 中调整调节数据（在此例中是 CI 元素）。

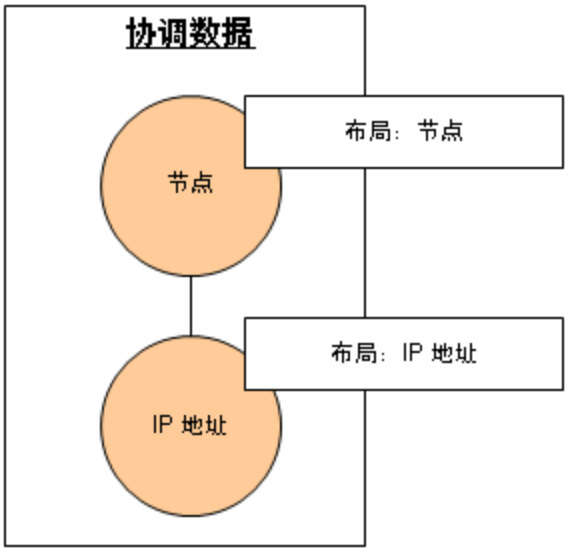
1. 在分析适配器之后，联合框架将识别出节点和事件之间的虚拟关系，并将联合 TQL 查询拆分成两个子图:



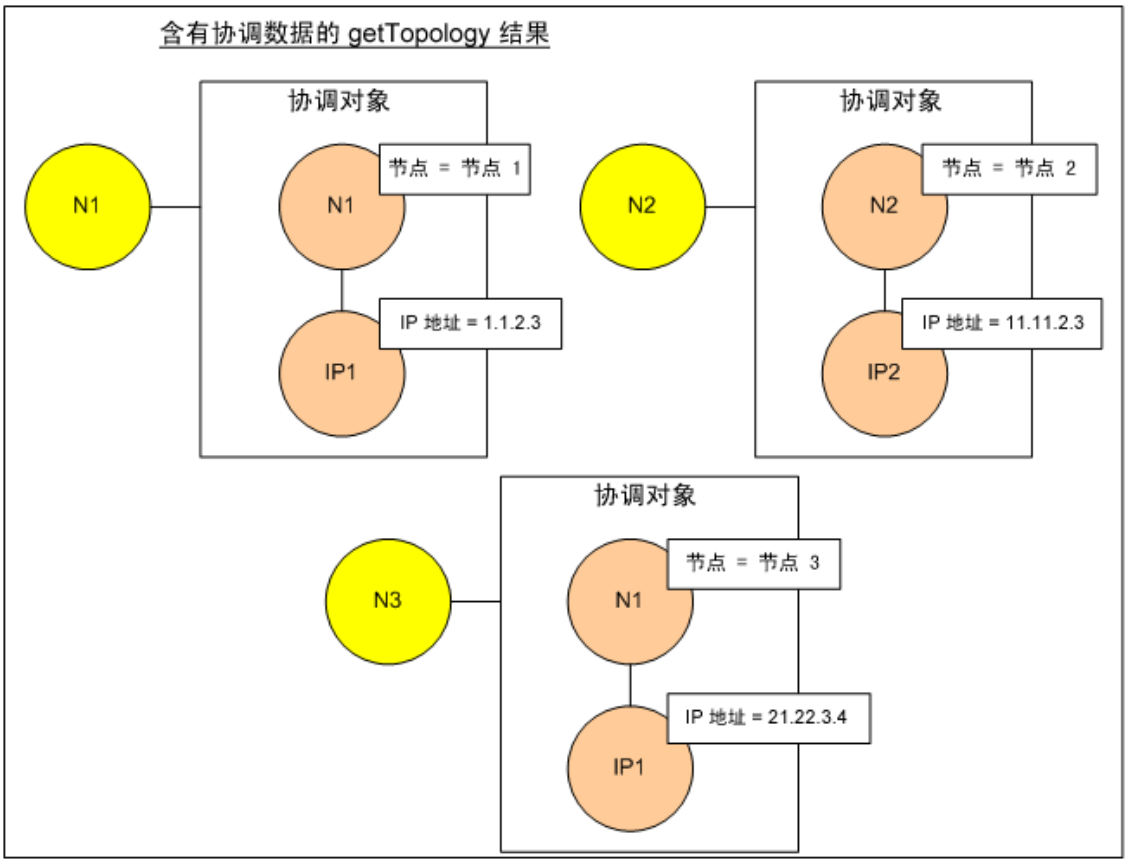
2. 联合框架将运行 UCMDB 子图请求拓扑，并且收到以下结果:



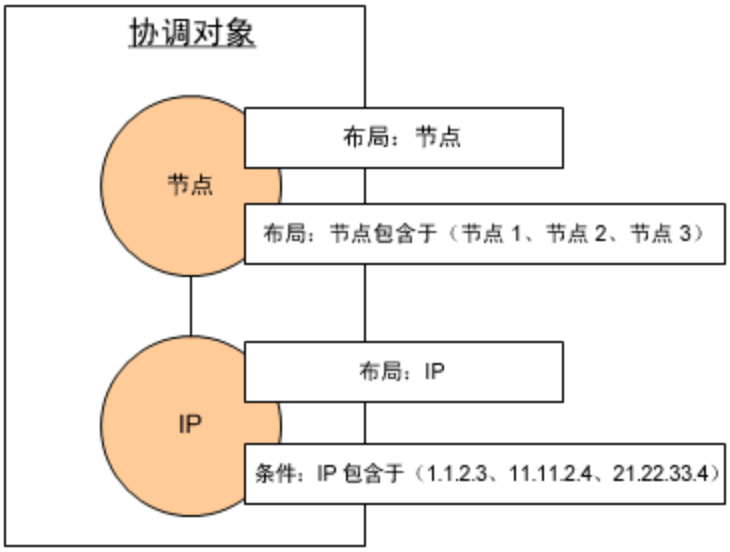
3. 联合框架从相应的映射引擎请求第一个数据库 (UCMDB) 的调节数据，其中包含用于连接从两个数据库收到的数据的信息。在此情况下，调节数据为：



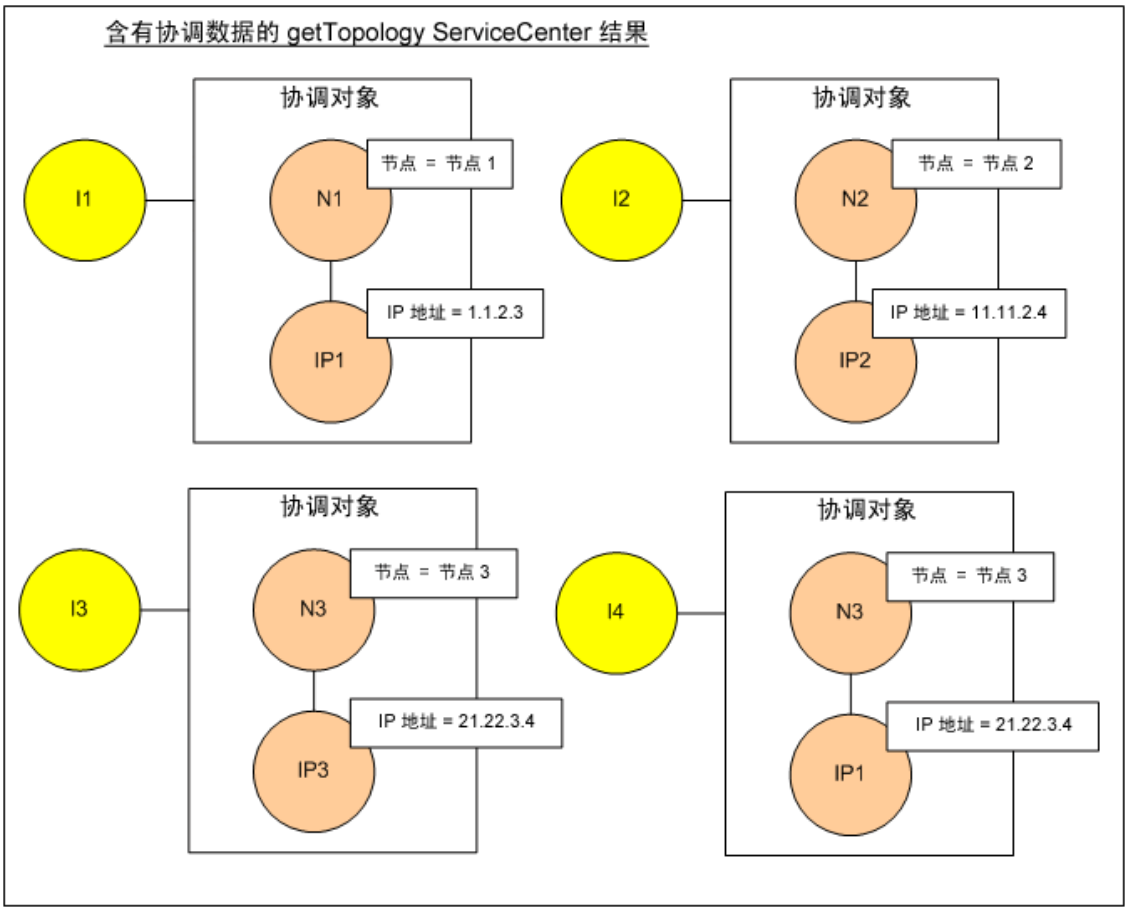
4. 联合框架使用先前的结果（H1、H2 和 H3 中的 node ）中的节点和 ID 条件创建单节点拓扑查询，并用所需的调节数据在 UCMDB 上运行此查询。结果包括与 ID 条件相关的节点 CI，以及每个 CI 相应的调节对象：



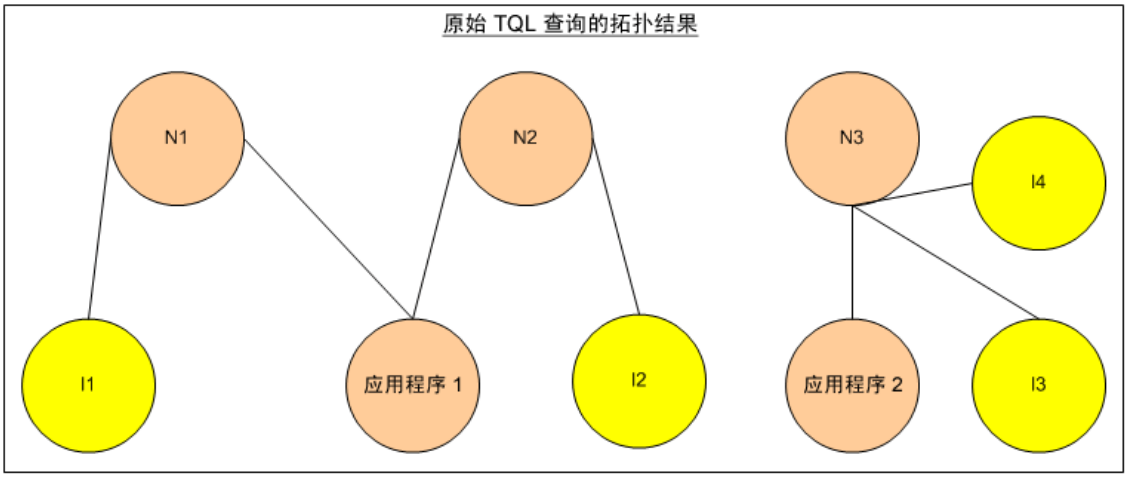
5. ServiceCenter 调节数据应包含从 UCMDB 接收的调节对象派生出的 node 和 IP 条件：



6. 联合框架使用调节数据运行 ServiceCenter 子图，请求拓扑和相应的调节对象，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。

用于填入的联合框架流

本节包括以下主题:

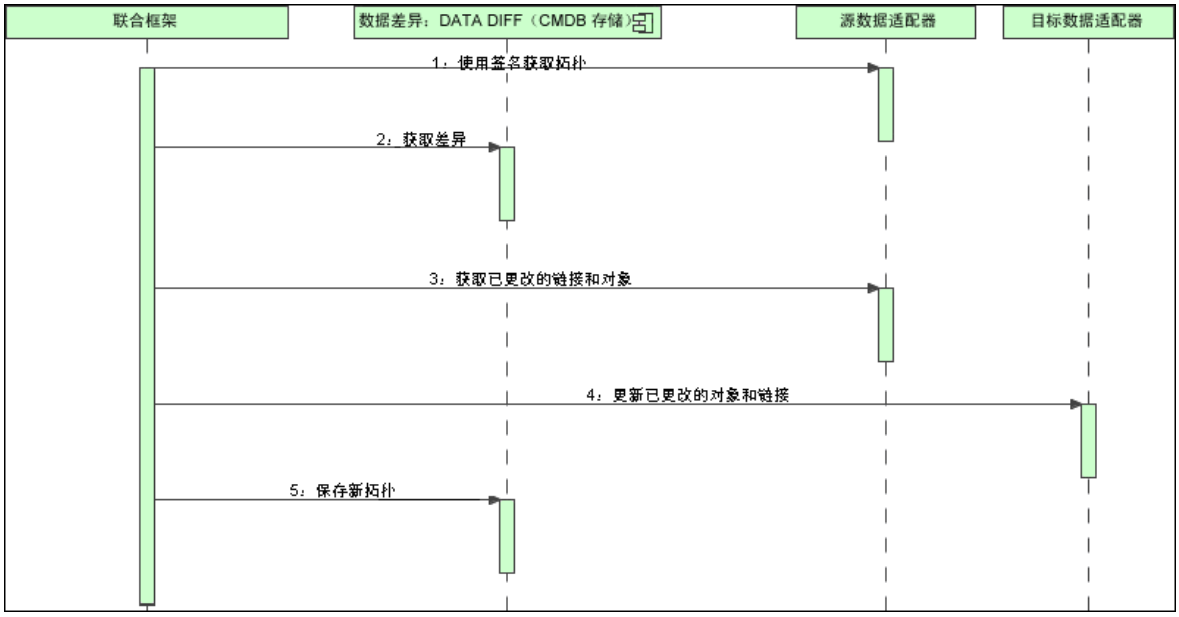
- 定义和术语 (第 180 页)
- 流程图 (第 180 页)

定义和术语

签名。表示 CI 中各个属性的状态。如果 CI 中的属性值发生变更，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 变更。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入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 发生了变更。**
3. **当联合框架找到发生变更的 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 (第 181 页)
- 联合 TQL 查询的适配器接口 (第 181 页)

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具有多个外部 CI 的请求。完成操作所需的调节数据可描述为复杂查询（请参阅下文所述的 SingleNodeFederationTopologyReconciliationAdapter）。

  创建 SingleNode 接口的目的是简化工作流；对于需要使用更广泛查询的情况，可使用 **FederationTopologyAdapter** 接口。
- **FederationTopologyAdapter 接口**用于定义支持复杂联合查询的适配器。这些适配器中的调节请求是 **QueryDefinition** 参数的组成部分。

  联合引擎使用调节数据将联合数据连接到正确的本地 CI。可能会在多个请求中获取调节数据（根据结果以递归方式计算）。在这种情况下，适配器可接收仅有调节数据的请求。

SingleNode 接口

以下接口包含不同类型的调节数据:

- **SingleNodeFederationIdReconciliationAdapter。**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节，则使用此接口。
- **SingleNodeFederationPropertyReconciliationAdapter。**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节，则使用此接口。
- **SingleNodeFederationTopologyReconciliationAdapter。**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节，则使用此接口。在查询元素为空且仅请求调节拓扑的情况下，适配器应提供支持。

数据适配器接口

- **FederationTopologyAdapter。**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。在查询定义仅描述调节数据的情况下，适配器应提供支持。
- **PopulateDataAdapter。**使用此适配器可支持复杂的联合 TQL 查询和填入流程。在填入流程中，此适配器将检索整个数据集，并允许探测器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 查询和填入流程。在填入流程中，此适配器仅支持检索自上次执行作业以来发生的变更。

备注: 当开发可能返回大型数据集的适配器时，务必实施 ChunkGetter 接口，以便对数据进行分块。有关详细信息，请参阅特定适配器的 Java 文档。

资源报告接口

以下接口支持适配器报告可配置的资源，以便自定义适配器的行为。这将支持您从集成工作室直接编辑这些资源。除了上述常规适配器接口之外，还应使用以下接口。

- **PopulationQueriesResourcesLocator**。定义可能要为每个特定填入查询编辑的资源。
- **PushQueriesResourceLocator**。定义可能要为每个数据推送查询编辑的资源。
- **GeneralResourcesLocator**。定义可以在此适配器中编辑的常规资源。

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**。用于填入流程中的源适配器。
- **TargetDataAdapter**。用于数据推送流中的目标适配器。

调试适配器资源

本任务描述如何使用 JMX 控制台创建、查看和删除适配器状态资源（即在 DataAdapterEnvironment 界面中使用资源操作方法创建的任何资源，这些资源保存在 UCMDB 数据库或探测器数据库中）进行调试和开发。

1. 启动 Web 浏览器，并输入以下服务器地址：
   - 对于 UCMDB 服务器：http://localhost:8080/jmx-console
   - 对于探测器：http://localhost:1977

   您可能需要使用用户名和密码登录。
2. 要打开 JMX MBEAN 视图页面，请执行以下操作之一：
   - 在 UCMDB 服务器上：单击 **UCMDB:service=FCMDB Adapter State Resource Services**
   - 在探测器上：单击 **type=AdapterStateResources**
3. 在要使用的操作中输入值，然后单击“Invoke”。

为新外部数据源添加适配器

此任务说明如何定义适配器以便支持新的外部数据源。

此任务包括以下步骤：

- 先决条件 (第 183 页)
- 为虚拟关系定义有效关系 (第 183 页)
- 定义适配器配置 (第 183 页)
- 定义支持的类 (第 186 页)
- 实施适配器 (第 187 页)

- 定义调节规则或实施映射引擎 (第 187 页)
- 添加实施类路径所需的 JAR (第 188 页)
- 部署适配器 (第 188 页)
- 更新适配器 (第 188 页)

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 类型的分层，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 的支持属性和调节规则的支持类中。

2. 为虚拟关系定义有效关系

   备注: 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 文件添加到适配器包的 **\validlinks** 文件夹中。有关详细信息，请参阅《HP Universal CMDB 管理指南》中的。

   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”  按钮并选择“新建适配器”。

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.BaseDestinationDa
ta">
<!-- check -->
<destinationData name="adapterId"
description="">${ADAPTER.adapter_id}</destinationData>
<destinationData name="attributeValues"
description="">${SOURCE.attribute_values}</destinationData>
<destinationData name="credentialsId"
description="">${SOURCE.credentials_id}</destinationData>
<destinationData name="destinationId"
description="">${SOURCE.destination_id}</destinationData>
</destinationInfo>
<resultMechanism isEnabled="true">
<autoDeleteCITs isEnabled="true">
```

```
<CIT>link</CIT>
<CIT>object</CIT>
</autoDeleteCITs>
</resultMechanism>
</taskInfo>
<adapterInfo>
<adapter-capabilities>
<support-federated-query>
<!--<supported-classes/> <!—see the section about supported
classes-->
<topology>
<pattern-topology /> <!—or <one-node-topology> -->
</topology>
</support-federated-query>
<!--<support-replicatioin-data>
<source>
<changes-source/>
</source>
<target/>
</adapter-capabilities>
<default-mapping-engine />
<queries />
<removedAttributes />
<full-population-days-interval>-1</full-population-days-
interval>
</adapterInfo>
<inputClass>destination_config</inputClass>
<protocols />
<parameters>
<!--The description attribute may be written in simple text or
HTML.-->
<!--The host attribute is treated as a special case by UCMDB-->
<!--and will automatically select the probe name (if possible)-
->
<!--according to this attribute’s value.-->
<parameter name="credentialsId" description="Special type of
property, handled by UCMDB for credentials menu" type="integer"
```

```
display-name="Credentials ID" mandatory="true" order-index="12"
/>
<parameter name="host" description="The host name or IP address
of the remote machine" type="string" display-name="Hostname/IP"
mandatory="false" order-index="10" />
<parameter name="port" description="The remote machine's
connection port" type="integer" display-name="Port"
mandatory="false" order-index="11" />
</parameters>
<parameter name="myatt" description="is my att true?"
type="string" display-name="My Att" mandatory="false" order-
index="15" valid-values="True;False"/>True</parameters>
<collectDiscoveredByInfo>true</collectDiscoveredByInfo>
<integration isEnabled="true">
<category >My Category</category>
</integration>
<overrideDomain>${SOURCE.probe_name}</overrideDomain>
<inputTQL>
<resource:XmlResourceWrapper
xmlns:resource="http://www.hp.com/ucmdb/1-0-
0/ResourceDefinition" xmlns:ns4="http://www.hp.com/ucmdb/1-0-
0/ViewDefinition" xmlns:tql="http://www.hp.com/ucmdb/1-0-
0/TopologyQueryLanguage">
<resource xsi:type="tql:Query" group-id="2" priority="low" is-
live="true" owner="Input TQL" name="Input TQL">
<tql:node class="adapter_config" id="-11" name="ADAPTER" />
<tql:node class="destination_config" id="-10" name="SOURCE" />
<tql:link to="ADAPTER" from="SOURCE" class="fcmdb_conf_
aggregation" id="-12" name="fcmdb_conf_aggregation" />
</resource>
</resource:XmlResourceWrapper>
</inputTQL>
<permissions />
</pattern>
```

有关 XML 标记的详细信息，请参阅 XML 配置标记和属性 (第 189 页)**。**

4. 定义支持的类

通过实现 *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节
is-id-reconciliation-supported	指定此类是否用于 ID 调节
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。

   如果适配器实现 **getTopologyWithReconciliationData**，且适配器功能包括要用作开始点的功能，则适配器也应支持请求带有调节数据的拓扑，且不含任何条件（仅类型）。在这种情况下，适配器应返回所找到结果的全部调节数据。

   您可以根据 **global_id** 定义适配器的调节。在这种情况下，**global_id** 必须定义为适配器支持类中的调节属性的一部分。如果根据 **global_id** 定义适配器的调节，则 **getTopologyWithReconciliationData()** 应将 **global_id** 作为协调对象属性的一部分返回。UCMDB 使用 **global_id** 调节 CIT、而非标识规则的联合结果。

   联合 API 的其中一部分是 DataAdapterEnvironment 界面。此界面表示数据适配器的环境。它还包含运行适配器所需的环境 API。有关 DataAdapterEnvironment 界面的详细信息，请参阅 DataAdapterEnvironment 界面 (第 191 页)。

6. 定义调节规则或实施映射引擎

   如果适配器支持联合 TQL 查询，则可通过下列两种方式来定义映射引擎：

   - 使用默认映射引擎，该引擎使用 CMDB 的内部调节规则进行映射。要使用该引擎，请将 **<default-mapping-engine/>** XML 标记保留为空。

- **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

   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\<适配器 ID> 文件夹下。**

     备注: 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\<适配器 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“如何部署包”。

创建示例适配器

本示例演示如何创建示例适配器。此任务包括以下步骤：

- 选择适配器逻辑 (第 189 页)
- 加载项目 (第 189 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、调节条件和链接条件）。

   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. 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 示例中使用的是选项 *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。

   备注: 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\APIs\DBAdapterFramework_JavaAPI\index.html

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。
	<start-point-adapter>	指定此适配器是 TQL 查询计算的开始点。
	<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和填入流程的功能。
	<source>	此适配器可用于填入流程。
	<push-back-ids>	将 CI 的全局 ID 推回到表的 global_id 列（必须在 orm.xml 中定义）。此行为可通过实施 **FcmdbPluginPushBackIds** 插件加以替代。
	<changes-source>	此适配器可用于填入变更流程。
	<instance-based-data>	此标记定义适配器支持基于实例的填入流程。
	<target>	此适配器可用于数据推送流。
	<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。
	<removedAttributes>	强制从结果中删除特定属性。
	<full-population-days-interval>	指定执行完整填入作业而不是差异作业的时间（每隔“x”天）。将老化机制用于变更流程。
	<adapter-settings>	适配器的设置列表。
	<list.attributes.for.set>	确定哪些属性将替代之前的值（如有）。

DataAdapterEnvironment 界面

OutputStream openResourceForWriting(String resourceName) throws FileNotFoundException;

此方法可打开带有指定名称的资源，以供写入。它可用来保存持久的集成数据。应使用此方法，而不是使用 java 方法来加载文件。用户应确保在完成流写入之后，关闭此流。close()/flush() 将保存资源。此方法可创建运行时资源（它可能不会覆盖进入适配器包的文件）。

参数

- **resourceName：** 要检索的资源名称。此名称在相同适配器的所有集成中应该是唯一的。

返回值

返回要写入到的流。

异常

- 此方法将在以下情况下引发 *FileNotFoundException*：资源类型是文件且此文件不存在、资源是目录而不是常规文件，或者由于某种其他原因无法打开资源进行读取。
- 此方法将在以下情况下引发 *SecurityException*：安全管理器存在且其 *checkRead* 方法拒绝访问文件。

InputStream openResourceForReading(String resourceName) throws FileNotFoundException;

此方法可打开带有指定名称的资源，以供读取。它可用来读取持久的集成数据。应使用此方法，而不是使用 java 方法来加载文件。用户应确保完成读取后，关闭此流。它将首先尝试加载进入适配器包的文件。如果未找到，它将尝试从 *DataAdapterEnvironment.openResourceForWriting(String)* 加载运行时创建的资源。可使用 JMX（属于对应的探测器和服务器）查看运行时资源。

参数

- **resourceName：** 要检索的资源名称。此名称在相同适配器的所有集成中应该是唯一的。

返回值

返回要读取的流。

异常

- 此方法将在以下情况下引发 *FileNotFoundException*：资源类型是“文件”且此文件不存在、资源是目录而不是常规文件，或者由于某种其他原因无法打开资源进行读取。
- 此方法将在以下情况下引发 *SecurityException*：安全管理器存在且其 *checkRead* 方法拒绝对文件的读取访问。

Properties openResourceAsProperties(String propertiesFile) throws IOException;

此方法可打开带有指定名称的资源，并将其加载为*属性* 结构。它可用来读取持久的集成数据。应使用此方法，而不是使用 java 方法来加载 **.properties** 文件。它将首先尝试加载进入适配器包的文件。如果未找到，它将尝试从 *DataAdapterEnvironment.openResourceForWriting(String)* 加载运行时创建的资源。可使用 JMX（属于对应的探测器和服务器）查看运行时资源。

参数

- **propertiesFile：** 要检索的资源名称。此名称在相同适配器的所有集成中应该是唯一的。

返回值

返回在属性中表示的文件内容。

异常

- 此方法将在以下情况下引发 *FileNotFoundException*：资源类型是“文件”且此文件不存在、资源是目录而不是常规文件，或者由于某种其他原因无法打开资源进行读取。
- 此方法将在以下情况下引发 *SecurityException*：安全管理器存在且其 *checkRead* 方法拒绝对文件的读取访问。
- 此方法将在以下情况下引发 *IOException*：属性文件无法转换为*属性* 对象。

String openResourceAsString(String resourceName) throws IOException;

此方法可打开带有指定名称的资源，并将其加载为字符串。它可用来读取持久的集成数据。应使用此方法，而不是使用 java 方法来加载文件。

它将首先尝试加载进入适配器包的文件。如果未找到，它将尝试从 *DataAdapterEnvironment.openResourceForWriting(String)* 加载运行时创建的资源。可使用 JMX（属于对应的探测器和服务器）查看运行时资源。

参数

- **resourceName：** 要检索的资源名称。此名称在相同适配器的所有集成中应该是唯一的。

返回值

返回以字符串格式表示的文件内容。

异常

- 此方法将在以下情况下引发 *FileNotFoundException*：资源类型是“文件”且此文件不存在、资源是目录而不是常规文件，或者由于某种其他原因无法打开资源进行读取。
- 此方法将在以下情况下引发 *SecurityException*：安全管理器存在且其 *checkRead* 方法拒绝对文件的读取访问。
- 此方法将在以下情况下引发 *IOException*：出现 I/O 错误。

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

此方法可接收字符串，并将其另存为资源。它可用来保存持久的集成数据。应使用此方法，而不是使用 java 方法来保存文件。此方法可将字符串转换为流，并将其保存到资源中。它可创建运行时资源，但是无法覆盖进入适配器包的文件。可使用 JMX（属于对应的探测器和服务器）查看运行时资源。

参数

- **relativeFileName：** 要检索的资源名称。此名称在相同适配器的所有集成中应该是唯一的。
- **value：** 要另存为资源的字符串。

异常

如果出现 I/O 错误，则此方法将引发 IOException。

boolean resourceExists(String resourceName);

此方法可检查指定的资源名称是否存在。它不仅查找进入适配器包的文件，还将从 *DataAdapterEnvironment.openResourceForWriting(String)* 查找运行时创建的资源。

参数

- **resourceName：** 要检索的资源名称。此名称在相同适配器的所有集成中应该是唯一的。

返回值

如果 *resourceName* 存在，则返回“True”。

boolean deleteResource(String resourceName);

此方法可从持久数据中删除指定的资源。它可删除运行时资源，但是可能不会删除进入适配器包的文件。可使用 JMX（用于对应的探测器和服务器）查看运行时资源。

参数

- **resourceName：** 要删除的资源名称。此名称在相同适配器的所有集成中应该是唯一的。

返回值

如果成功删除资源，则返回“True”。

Collection<String> listResourcesInPath(String path);

此方法可检索指定资源路径中的资源列表。它不仅查找进入适配器包的文件，还将从 *DataAdapterEnvironment.openResourceForWriting(String)* 查找运行时创建的资源。可使用 JMX（用于对应的探测器和服务器）查看运行时资源。

参数

- **path：** 资源路径。例如“META-INF/myfiles/”

返回值

返回路径中的资源列表。

DataAdapterLogger getLogger();

检索适配器要使用的记录器。此记录器用于在适配器中记录事件。

返回值

返回 DataAdapter 使用的记录器。

DestinationConfig getDestinationConfig();

此方法可检索集成的目标配置。此配置可为集成保留所有连接和运行设置。

返回值

返回适配器的 DestinationConfig。

int getChunkSize();

此方法可检索为此集成请求的填入块大小。

返回值

返回填入块大小。

int getPushChunkSize();

此方法可检索为此集成请求的推送块大小。

返回值

返回推送块大小。

ClassModel getLocalClassModel();

此方法可检索类模型，以便查询有关本地 UCMDB 类模型的信息。此模型可引入已更新的 ClassModel。ClassModel 对象返回后，不会针对任何类模型变更而更新。要检索已更新的类模型，请再次使用此方法检索该类模型。

返回值

返回 UCMDB 的类模型。

CustomerInformation getLocalCustomerInformation();

此方法可为正在执行适配器的客户检索客户信息。

返回值

为正在执行适配器的客户返回客户信息。

Object getSettingValue(String name);

此方法可检索特定的适配器设置。

参数

name：设置的名称。

返回值

返回对象设置的值。

Map<String, Object> getAllSettings();

此方法可检索所有适配器设置。

返回值

返回适配器设置。

boolean isMTEnabled();

此方法可检查服务器环境是否支持多租赁 (MT)。

返回值

如果服务器环境支持 MT，则将返回“true”，否则将返回“false”。

String getUcmdbServerHostName();

此方法可返回本地 UCMDB 服务器的主机名。

返回值

返回本地 UCMDB 服务器的主机名。

第 7 章: 开发推送适配器

本章包括:

- 开发和部署推送适配器 196
- 生成适配器包 196
- 创建映射 199
- 编写 Jython 脚本 202
- 支持差异同步 205
- 常规 XML 推送适配器 SQL 查询 206
- 常规 Web 服务推送适配器 207
- 映射文件引用 224
- 映射文件架构 226
- 映射结果架构 234
- 自定义 236

开发和部署推送适配器

常规推送适配器提供了一个常见平台，通过该平台可以快速开发用于将 UCMDB 数据推送至外部数据库（数据库和第三方应用程序）的集成。常规推送适配器将根据用来推送数据的协议进行分类。有关通过 XML（即：使用常规 XML 推送适配器）进行推送的详细信息，请参阅常规 XML 推送适配器 SQL 查询 (第 206 页)。有关通过 Web 服务（即：使用常规 Web 服务推送适配器）进行推送的详细信息，请参阅常规 Web 服务推送适配器 (第 207 页)。

基于常规推送适配器开发自定义集成时，需要:

- 从相应的常规推送适配器模板文件中构建新的适配器包。有关详细信息，请参阅生成适配器包 (第 196 页)。
- UCMDB CI 链接类型和外部数据项之间的映射。将映射存储为 XML，并且将其自定义到每个外部数据库。有关详细信息，请参阅创建映射 (第 199 页)。
- 将数据项推送到外部数据库的 Jython 脚本。有关详细信息，请参阅编写 Jython 脚本 (第 202 页)。
- 其他特定于适配器的步骤。例如，为 XML 推送适配器选择要写入的文件路径，或为 Web 服务推送适配器创建数据接收器。

生成适配器包

要创建新的 MDR 特定推送适配器，应创建一个常规适配器副本，然后编辑并自定义该副本，使其变为特定推送目标的适配器。

常规适配器包位于以下两个位置之一:

- 常规 XML 推送适配器：**hp\UCMDB\UCMDBServer\content\adapters\push-adapter.zip**
- 常规 Web 服务适配器：**hp\UCMDB\UCMDBServer\content\adapters\web-service-push-adapter.zip**

要从常规推送适配器中创建新的推送适配器，请执行以下操作：

1. 将选定包 zip 文件的内容提取到工作文件夹。
2. 审查以下目录，为重命名和替换阶段做好准备：
   - **adapterCode：**包含部署到 **C:\hp\UCMDB\UCMDBServer\runtime\fcmdb\CodeBase directory** 的目录。此处部署的 Jar 不会自动重新启动探测器，且不会自动显示在探测器的 CLASSPATH 中。
   - **discoveryConfigFiles：**包含适配器的映射定义，并指向正确的 Jython 脚本 (**push.properties**)
   - **discoveryPatterns：**包含部署在 UCMDB 服务器上的适配器 XML 定义
   - **discoveryScripts：**包含适配器的 Jython 脚本，使用此脚本连接到第三方数据存储并推送数据
   - **discoveryResources：**包含具有 Web 服务 Java 集成类的 **UCMDBDataReceiver.jar**。

   备注：部署此包时，系统将重新启动探测器，以使此 **.jar** 包含在探测器的 CLASSPATH 中。除了部署此包之外，无需采取任何操作。
3. 在未解压缩的适配器目录结构内进行以下变更：
   a. **discoveryConfigFiles\<您的推送适配器名称>：**将目录“PushAdapter”或“XMLtoWebService”重命名为新推送适配器的名称（例如，“myPushAdapter”）。
   b. **discoveryConfigFiles\<您的推送适配器名称>\push.properties：**在 **push.properties** 文件中，执行以下操作：
      - 将 **jythonScript.name** 的名称更新为新推送适配器将使用的 Jython 脚本的名称（例如，**pushToMyService.py**）。
      - 更新新推送适配器将使用的映射文件的名称（例如，myPushAdapter_mappings）。不要添加 **.xml** 扩展名，系统将自动填写该扩展名。
   c. **discoveryPatterns\<推送适配器名称>.xml：**将此文件重命名为新适配器的定义 XML 文件的名称（例如，**my_push_adapter.xml**）。
   d. **discoveryPatterns\<您的推送适配器名称>.xml：**按照如下方式更新此文件：
      - 对于 XML 元素 **<pattern>：**设置相应的 ID 和描述属性。例如：

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

        更改为：

        <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">
      - 对于 XML 元素 **<parameters>：**根据适配器的需求，更新子元素。默认情况下，以下子元素用于定义推送适配器。当完成适配器配置后，在“集成工作室”中定义集成点时，系统将分配这些值。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。

- **host：**托管 Web 服务的服务器名称
- **port：**侦听 UCMDB 数据接收器服务的端口
- **Web Service Push Adapter：uri** - 构成数据接收器的服务端点地址的其余 URL。
- **probeName：**定义运行推送作业的 Data Flow Probe

◦ 对于 XML 元素 **<integration>：**将子元素 **<category>** 的值更新为“常规”以外的其他值。默认情况下，属于“常规”类别的集成适配器不会显示在“集成工作室”中。如果将与第三方数据存储集成，则将此值设置为“第三方”。如果将与 HP BTO 产品集成，则将此值设置为“HP BTO 产品”。

e. **adapterCode\PushAdapter：**使用在上一步中使用的适配器 ID 重命名此文件夹（例如，**adapterCode\MyPushAdapter**）。

f. **discoveryScripts\<您的 Jython 推送脚本>.py：**创建一个名称与在 **push.properties jythonScript.name** 属性中定义的名称相同的文件。在 **discoveryScript** 文件中，有一个脚本可将 CI 和链接插入外部 Oracle 数据库。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 202 页)）。如果重命名该脚本，则应相应地更新 **adapterCode\<适配器 ID>\push.properties** 中的 **jythonScript.name** 属性。

◦ XML 推送适配器：**pushScript.py**

◦ Web 服务推送适配器：**XMLtoWebService.py**

g. **tql\<您的集成 TQL>：**与常规包类似，请将集成 TQL 的 TQL XML 定义放在此目录中。在部署适配器包后，将部署此文件夹中的所有 TQL。

h. **discoveryConfigFiles\<您的推送适配器名称>\mappings：**为要在集成中使用的每个 TQL 创建 XML 映射文件。请注意，推送适配器将映射文件中的转换应用于集成 TQL 的结果，然后将特别任务的三个参数（**addResult**、**updateResult** 和 **deleteResult**）中的数据发送到 Data Flow Probe。

i. **adapterCode\<适配器 ID>\mappings：**使用准备的映射文件替换 **mappings.xml** 文件（有关详细信息，请参阅创建映射 (第 199 页)）。

XML 推送适配器：此映射示例对应于在 ORACLE 的 sql_queries 文件中创建的表示例。

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。

4. 进行上述所有变更后，通过选择在上述步骤 3 中指定的文件夹和文件，创建一个 **.zip** 文件（例如，**my_Push_Adapter.zip**）。
5. 通过包管理器在 UCMDB 服务器上部署新创建的 **.zip** 文件（转到“管理”>“包管理器”）。
6. 在“数据流管理”>“集成工作室”中创建集成点，定义集成点使用的集成 TQL。为自动数据推送设置计划。

疑难解答

构建新推送适配器的过程需要进行完整且正确的重命名以及替换操作。任何错误都有可能影响适配器。必须对包进行正确的解压缩和重新压缩，使其充当 UCMDB 包。请参考现成包为例。常见错误包括：

- ZIP 文件中包含包目录顶部的另一个目录。
  解决方案：对与包目录 **discoveryResources**、**adapterCode** 等相同目录中的包进行 ZIP 压缩。请勿将此目录顶部的另一个目录级别包含在此 ZIP 文件中。

- 省略了目录、文件或文件中字符串的关键重命名。
  解决方案：请认真遵循本节的说明。
- 目录、文件或文件中字符串的关键重命名拼写错误。
  解决方案：开始重命名过程后，请勿中途更改命名约定。如果您发现需要更改名称，请从头重新开始，而不是试图以回溯方式更正名称，因为这样操作出现错误的风险很高。此外，也可以使用搜索和替换功能，而不是手动替换字符串，以便降低错误风险。
- 部署文件名与其他适配器文件名相同的适配器（尤其是在 **discoveryResources** 和 **adapterCode** 目录中）。
  解决方案：可以使用具有已知问题的 UCMDB 版本，该版本可防止映射文件具有与同一 UCMDB 环境中的任何其他适配器相同的文件名。如果尝试部署名称重复的包，则包部署将失败。即使这些文件位于不同的目录中，也可能出现这一问题。此外，无论重复发生在包内还是发生在先前已部署的其他包中，均会出现这一问题。

此时，可以使用刚部署的新适配器，在“集成工作室”中创建新的推送适配器作业。

推送适配器的 TQL 最佳实践

1. 在“TQL”和“视图”树中创建文件夹结构，并在该处保留所有新的 TQL 和视图。使用命名约定。
2. 除非 TQL 较小，否则应首先复制最相似的 TQL。
3. 一次进行一项更改。在每次更改之后进行保存、测试并预览。重复此过程，直到结果符合您的要求。

创建映射

原始 TQL 结果数据以 UCMDB 类模型架构的形式显示。使用者可能会使用不同的数据模型。推送适配器可提供一种映射机制，将数据转换为更适合使用的格式。映射可执行直接转换和复杂转换，范围从直接、命名类型约定到父/子聚合以及引用函数。

映射规范位于映射文件引用 (第 224 页)章节中。使用引用来创建映射文件。

备注: 适配器属性文件引用映射文件的名称。在适配器配置文件中，适配器使用适配器的名称实现文件夹结构。实现适配器以保持包管理器所需的唯一性时，重命名此文件夹。

构建映射文件

1. 从默认映射文件开始。
2. 部署适配器并运行一次。
3. 观察结果。
4. 标识并记录应更改哪些内容。
5. 对上一步中标识的内容进行更改。以下列表可帮助对变更顺序进行指导。
   a. 从顶部不可转换的部分开始。请确保在每次更改后运行适配器。
   b. 将来源 CI 部分更改为 TQL 结果中的 UCMDB 名称。
   c. 首先映射密钥。
   d. 然后添加所有直接映射。

e. 添加复杂映射。

f. 添加链接映射。

重复步骤 2-5，直到映射的数据适合使用。选择创建新推送适配器所使用的常规适配器包。

映射文件的工作方式与所有类型的推送适配器工作方式相同。常规 XML 推送适配器将映射的结果写入文件。常规 Web 服务推送适配器将 XML 结果发送到数据接收器。有关更多详细信息，请参阅常规 Web 服务推送适配器 (第 207 页)。

准备映射文件

备注: 不创建 **mappings.xml** 文件，您可以在不映射的情况下，像在 CMDB 中一样检索所有 CI 和关系。检索时将返回所有 CI 和关系及其所有属性。

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 226 页)。

此任务包括以下步骤:

- 创建 mappings.xml 文件 (第 200 页)
- 映射 CI (第 200 页)
- 映射链接 (第 201 页)

1. 创建 mappings.xml 文件

   如下所示创建映射文件结构（使用现有文件作为模板）:

```
<?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

可通过两种方式映射 CMDB CI 类型:

- **映射一个 CI 类型，便该类型和所有继承类型的 CI 都以相同的方式映射:**

```
<source_ci_type_tree name="node" mode="update_else_insert">
  <apioutputseq>1</apioutputseq>
  <target_ci_type  name="host">
    <targetprimarykey>
      <pkey>name</pkey>
    </targetprimarykey
    <target_attribute name=" name" datatype="STRING">
      <map type="direct" source_attribute="name" >
    </target_attribute>
    <!-- more target attributes --->
  </target_ci_type>
</source_ci_type_tree>
```

- **映射一个 CI 类型，以便仅处理该类型的 CI。继承类型的 CI 不会被处理，除非其类型也以两种方式之一进行了映射:**

```
<source_ci_type name="node" mode="update_else_insert">
  <apioutputseq>1</apioutputseq>
  <target_ci_type  name="host">
    <targetprimarykey>
      <pkey>name</pkey>
    </targetprimarykey
    <target_attribute name=" name" datatype="STRING">
      <map type="direct" source_attribute="name" >
    </target_attribute>
    <!-- more target attributes --->
  </target_ci_type>
</source_ci_type>
```

间接映射的 CI 类型（其原始类型之一使用 source_ci_type_tree 映射）通过使其出现在自己的 source_ci_type_tree 或 source_ci_type 中还会替代其父类型的映射。

建议尽可能使用 source_ci_type_tree。否则，所生成 CI 类型未出现在映射文件中的 CI 不会传输到 Jython 脚本。

3. 映射链接

可通过两种方式映射链接:

- **映射一个链接，以便该类型的链接和所有继承链接都以相同的方式映射:**

```
<source_link_type_tree name="dependency" target_link_type="dependency"
mode="update_else_insert" source_ci_type_end1="webservice" source_ci_type_
end2="sap_gateway">
  <target_ci_type_end1 name="webservice" >
  <target_ci_type_end2 name="sap_gateway" >
```

```
    <target_attribute name="name" datatype="STRING">
      <map type="direct" source_attribute="name" >
    </target_attribute>
</source_link_type_tree>
```

- **映射一个链接，以便仅处理该类型的链接。继承类型的链接不会被处理，除非其类型也以两种方式之一进行了映射：**

```
<link source_link_type="dependency" target_link_type="dependency" mode="update_
else_insert" source_ci_type_end1="webservice" source_ci_type_end2="sap_gateway">
    <target_ci_type_end1 name="webservice" >
    <target_ci_type_end2 name="sap_gateway" >
    <target_attribute name="name" datatype="STRING">
      <map type="direct" source_attribute="name" >
  </target_attribute>
</link>
```

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 34 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 196 页)**中有关编辑 discoveryPatterns\push_adapter.xml 文件的步骤），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 203 页)
- 在脚本中处理测试连接 (第 205 页)

使用映射结果

常规推送适配器将创建 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。

在 XML 中，CI 的 id 属性包含 cmdbId 作为外部 ID，或者该 CI 的 ExternalId（如果 CI 在被发送到脚本时获得了一个 ExternalId）。链接的 end1Id 和 end2Id 字段针对每个链接端包含 cmdbId 作为外部 ID 或该链接端的 ExternalId（如果该链接端在被发送到脚本时获得了一个 ExternalId）。

处理 Jython 脚本中的 CI 时，脚本的返回值为 CI 的 CMDB ID 与给定 ID（在脚本中为每个 CI 指定的 ID）之间的映射。如果是第一次推送 CI，该 CI 的 XML 中的 ID 为 CMDB ID。如果不是第一次推送 CI，该 CI 的 ID 为在第一次推送该 CI 时在脚本中为其指定的 ID。

通过如下方式从 CI XML 脚本中检索 ID:

1. 在 XML 的 CI 元素中，从 ID 属性检索 ID。例如： id = objectElement.getAttributeValue('id')。
2. 从 XML 检索 ID 后，从属性 (string) 恢复 ID。例如： objectId = CmdbObjectID.Factory.restoreObjectID(id)。
3. 检查上一步中接收的 objectId 是否为 CMDB ID。您可以通过检查 objectId 的新 ID 是否为脚本为其指定的 ID 来执行此操作。如果是，则返回的 ID 不是 CMDB ID。例如:
   newId = objectId.getPropertyValue(<脚本指定的 ID 属性的名称>)。

   如果 newId 为空，则在 XML 中返回的 ID 为 CMDB ID。
4. 如果 ID 为 CMDB ID（即 newId 为空），则执行以下操作（如果 ID 不是 CMDB ID，则转到步骤 5）:
   a. 为该 CI 创建包含新 ID 的属性。例如：propArray =[TypesFactory.createProperty('<脚本指定的 ID 属性的名称>', '<新 ID>')]。
   b. 为该 CI 创建 externalId。例如:
   cmdbId = extI.getPropertyValue('internal_id')
   className = extI.getType()
   externalId = ExternalIdFactory.createExternalCiId(className, propArray)
   c. 将 CMDB ID 映射到新创建的 externalId （在下一个步骤中，将该映射返回到适配器）。例如:
   objectMappings.put(cmdbId, externalId)
   d. 当映射所有 CI 和链接时:
   updateResult = DataPushResultsFactory.createDataPushResults(objectMappings, linkMappings);
   return updateResult
5. 如果 ID 是新 ID（即 newId 不为空），则 externalId 为 newId。

还可以按照如下方式报告每个 CI 和链接的推送状态:

1. updateStatus = ReplicationActionDataFactory.createUpdateStatus();

   其中 updateStatus 是包含 CI 和链接状态的 UpdateStatus 类的实例。
2. 通过调用 reportCIStatus 或 reportRelationStatus 方法，将状态添加到 updateStatus。

   例如：

   status = ReplicationActionDataFactory.createStatus(Severity.FAILURE, 'Failed', ERROR_CODE_CI, errorParams,Action.ADD);

   updateStatus.reportCIStatus(externalId, status);

   其中 ERROR_CODE_CI 是适配器 **properties.errors** 文件中显示的错误消息数（有关 **properties.errors** 文件的详细信息，请参阅错误编写约定 (第 57 页)），errorParams 包含要传递给消息的参数。有关更多详细信息，请参阅 **ReplicationActionDataFactory** javadoc。
3. 创建具有如下状态的推送结果：

   updateResult = DataPushResultsFactory.createDataPushResults(objectMappings, linkMappings, updateStatus);

   return updateResult

XML 结果示例

```
<root>
 <data>
  <objects>
   <Object mode="update_else_insert" name="UCMDB_UNIX" operation="add"
mamId="0c82f591bc3a584121b0b85efd90b174"
id="HiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A">
    <field name="NAME" key="false" datatype="char" length="255">UNIX5</field>
    <field name="DATA_NOTE" key="false" datatype="char" length="255"></field>
   </Object>
  </objects>
  <links>
    <link targetRelationshipClass="TALK" targetParent="unix" targetChild="unix" operation="add"
mode="update_else_insert"
mamId="265e985c6ec51a8543f461b30fa58f81"
id="end1id%5BHiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D41372a1cbcaba27b214b84a2ec9eb535%0A%5D%0Aend2id%
5BHiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A%5D%0AHiddenRmi
DataSource%0Atalk%0A1%0Ainternal_
id%3DSTRING%3D265e985c6ec51a8543f461b30fa58f81%0A">
    <field name="DiscoveryID1">41372a1cbcaba27b214b84a2ec9eb535</field>
```

```
    <field name="DiscoveryID2">0c82f591bc3a584121b0b85efd90b174</field>
    <field name="end1Id">HiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D41372a1cbcaba27b214b84a2ec9eb535%0A</field>
    <field name="end2Id">HiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A</field>
    <field name="NAME" key="false" datatype="char" length="255">TALK4</field>
    <field name="DATA_NOTE" key="false" datatype="char" length="255"></field>
    </link>
  </links>
 </data>
</root>
```

备注: 如果 datatype="BYTE"，则返回的结果值为生成的 **String**，如下所示：new String([the byte array attribute])。可以通过以下方式重新构建 byte[] object：<the received String>.getBytes()。如果服务器和探测器之间的默认语言环境存在差异，则应按照服务器的默认语言环境重新构建。

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

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。

ExternalIdUtil.restoreExternal 命令接收 CMDB 中的 CI ID 作为参数，可从 CMDB 中的 CI ID 恢复外部 ID。例如，执行差异同步时，如果接收到的链接的一端不在批中（链接已经同步），则可以使用此命令。

如果 Jython 脚本中推送适配器所基于的 **DiscoveryMain** 方法返回空的 **ObjectStateHolderVector** 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。

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 196 页)。

此任务支持推送适配器执行差异同步。

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）

    ```
    externaCIId = 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)
    ```

    备注: 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。

备注: 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。

常规 XML 推送适配器 SQL 查询

在适配器包中，位于 **adapterCode > PushAdapter > sqlTablesCreation** 中的 **sql_queries** 文件包含在 Oracle 中用于测试适配器的新架构中创建表所需的查询。表对应于 adapterCode\<适配器 ID>\mappings\mappings.xml 文件。

备注: 适配器并不需要 **sql_queries** 文件。它只是一个示例而已。

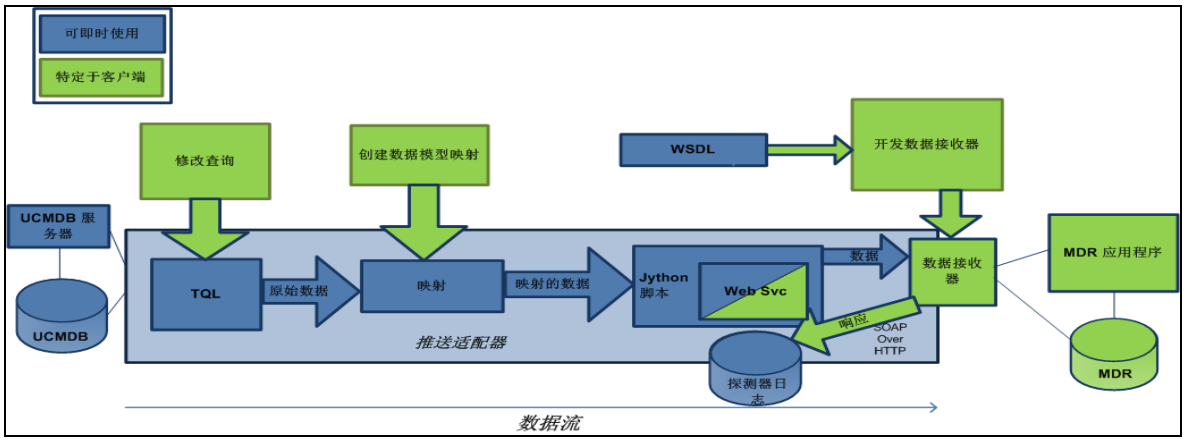
常规 Web 服务推送适配器

常规 Web 服务推送适配器可通过 UCMDB 将包含查询数据的 SOAP 消息推送到 Web 服务数据接收器。映射结果将通过 HTTP POST 协议以标准 SOAP 消息的形式发送到数据接收器。数据接收器必须了解推送适配器生成的 SOAP 消息。为了帮助开发正确的数据接收器，此推送适配器提供了 WSDL。

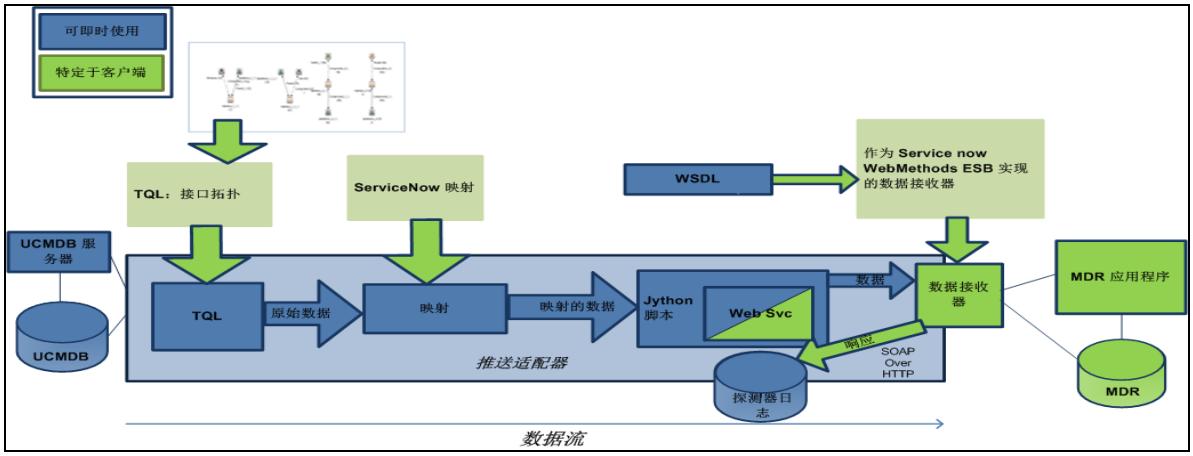
可以采用 Jython 脚本自定义处理 SOAP 消息响应 XML。

要了解传入映射数据的格式，数据接收器的开发人员应与映射文件的开发人员进行沟通。此版本的 Web 服务推送适配器目前不提供 **.xsd**，因此系统必须以反映传入数据的方式处理数据，该方式结合了原始 TQL 和应用的映射。

用于将数据推送到客户端的 Web 服务推送适配器的功能如下所示。绿色项由客户端自定义或提供，用于为特定推送目标实现适配器。蓝色项为现成的组件。



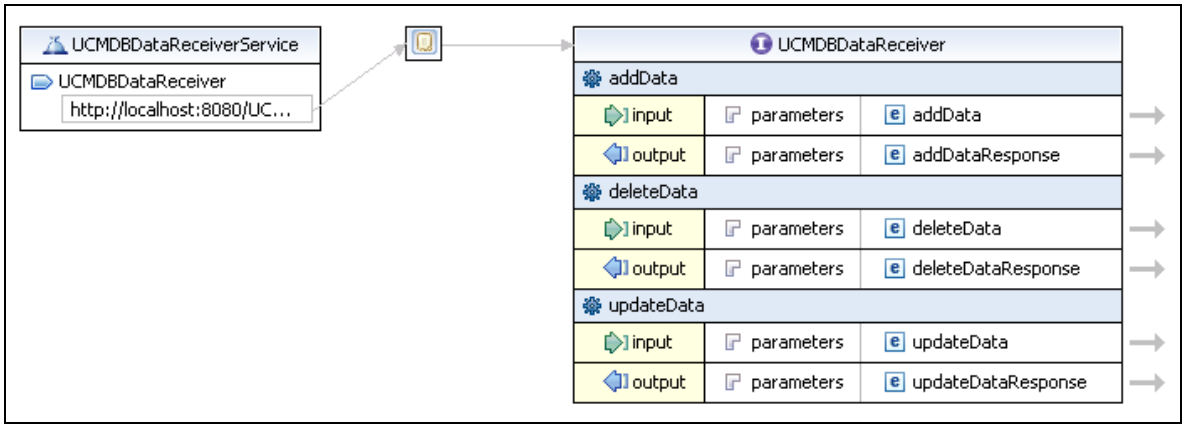
使用企业服务总线 (ESB) 将常规 Web 服务推送适配器实现为 MDR 特定推送适配器的示例如下所示：



WSDL

WSDL 提供给客户端开发人员，用于创建可通过 Web 服务与 UCMDB 推送适配器进行通信的数据接收

器。**UCMDBDataReceiver.wsdl** 描述用于在 UCMDB 和数据接收器之间进行数据通信的 SOAP 消息。WSDL 中的设计图如下所示：



数据接收器（实际上称为服务器，或在 SOAP 术语中称为“服务端点”）应实现三个方法：**addData**、**deleteData** 以及 **updateData**，对应于 UCMDB 推送的数据集。HTTP 标头包含正确的 **SoapAction** 关键字，表示正在发送的数据类型。数据接收器负责实现业务逻辑和处理数据。

默认 WSDL URL 为：

- http://localhost:8080/UCMDBDataReceiver/services/UCMDBDataReceiver?wsdl

由于 URL 由数据接收器实现，因此可能如下所示：

- http://testWSPAserver:4444/MyCo.IT.SvcMgt.ws.us:provider/UCMDBDataReceiver?wsdl

Web 服务的 URL 与结尾处不带“?wsdl”的 WSDL URL 相同。

WSDL 的源包括以下内容：

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="http://ucmdb.hp.com"
xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://ucmdb.hp.com"
xmlns:intf="http://ucmdb.hp.com" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!--WSDL created by Apache Axis version:1.4 Built on Apr 22, 2006 (06:55:48 PDT)-->

    <wsdl:types>

        <schema elementFormDefault="qualified" targetNamespace="http://ucmdb.hp.com"
        xmlns="http://www.w3.org/2001/XMLSchema">

            <element name="addData">

                <complexType>

                    <sequence>

                    <element name="xmlAdded" type="xsd:string"/>

                    </sequence>
```

```
                </complexType>
            </element>
            <element name="addDataResponse">
                <complexType/>
            </element>
            <element name="deleteData">
                <complexType>
                    <sequence>
                    <element name="xmlDeleted" type="xsd:string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="deleteDataResponse">
                <complexType/>
            </element>
            <element name="updateData">
                <complexType>
                    <sequence>
                    <element name="xmlUpdate" type="xsd:string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="updateDataResponse">
                <complexType/>
            </element>
        </schema>
    </wsdl:types>

    <wsdl:message name="addDataRequest">
        <wsdl:part element="impl:addData" name="parameters">
        </wsdl:part>
```

```
</wsdl:message>
<wsdl:message name="deleteDataResponse">
    <wsdl:part element="impl:deleteDataResponse" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="updateDataResponse">
    <wsdl:part element="impl:updateDataResponse" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="deleteDataRequest">
    <wsdl:part element="impl:deleteData" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="addDataResponse">
    <wsdl:part element="impl:addDataResponse" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:message name="updateDataRequest">
    <wsdl:part element="impl:updateData" name="parameters">
    </wsdl:part>
</wsdl:message>
<wsdl:portType name="UCMDBDataReceiver">
    <wsdl:operation name="addData">
        <wsdlsoap:operation soapAction="addDataRequest"/>
        <wsdl:input message="impl:addDataRequest" name="addDataRequest">
        </wsdl:input>
        <wsdl:output message="impl:addDataResponse" name="addDataResponse">
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="deleteData">
        <wsdlsoap:operation soapAction="deleteDataRequest"/>
```

```
            <wsdl:input message="impl:deleteDataRequest" name="deleteDataRequest">
            </wsdl:input>
            <wsdl:output message="impl:deleteDataResponse" name="deleteDataResponse">
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="updateData">
            <wsdlsoap:operation soapAction="updateDataRequest"/>
            <wsdl:input message="impl:updateDataRequest" name="updateDataRequest">
            </wsdl:input>
            <wsdl:output message="impl:updateDataResponse"
            name="updateDataResponse">
            </wsdl:output>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="UCMDBDataReceiverSoapBinding" type="impl:UCMDBDataReceiver">
        <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"
        />
        <wsdl:operation name="addData">
            <wsdl:input name="addDataRequest">
                <wsdlsoap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="addDataResponse">
                <wsdlsoap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="deleteData">
            <wsdl:input name="deleteDataRequest">
                <wsdlsoap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="deleteDataResponse">
                <wsdlsoap:body use="literal" />
            </wsdl:output>
```

```
            </wsdl:operation>
            <wsdl:operation name="updateData">
                <wsdl:input name="updateDataRequest">
                    <wsdlsoap:body use="literal" />
                </wsdl:input>
                <wsdl:output name="updateDataResponse">
                    <wsdlsoap:body use="literal" />
                </wsdl:output>
            </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="UCMDBDataReceiverService">
            <wsdl:port binding="impl:UCMDBDataReceiverSoapBinding"
            name="UCMDBDataReceiver">
                <wsdlsoap:address location="http://localhost:8080/UCMDBDataReceiver/services/
                UCMDBDataReceiver"/>
            </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

响应处理

数据接收器应返回结构为 addDataResponse、deleteDataResponse 或 updateDataResponse 的字符串。适配器将未处理的响应数据传递到探测器的 probeMgr-adaptersDebug.log。接收器可以返回任何字符串数据，响应将打包到 SOAP 兼容的 XML 中。在 Jython 脚本中，可以使用 SOAPMessage 和相关的 Java 类来解析响应消息。以下是数据接收器中响应消息的一个示例：

```
<2012-03-16 15:47:38,080> [INFO ] [Thread-110] - XMLtoWebService.py:addData received response:
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<intf:addDataResponse xmlns:intf="http://ucmdb.hp.com">
    <xml>&lt;result&gt;&lt;status&gt;error&lt;/status&gt;
    &lt;message&gt;Error publishing config item changes&lt;/message&gt;
    &lt;/result&gt;</xml>
</intf:addDataResponse>
</soapenv:Body>
```

所显示的消息是一条错误消息 <Error publishing config item changes>，但内容可以为任何内容，因为数据接收器旨在进行响应。响应为错误消息，因为这就是目的所在，由于设计人员说它是错误消息，因

此推送适配器希望响应指出成功还是失败。内容可以是所有成功添加的 CI 的调节 ID，或特定 CI 的错误消息。自定义 GWSPA 可以包括解析响应消息，执行重新发送特定 CI 或执行其他日志记录等操作。

测试 WSDL

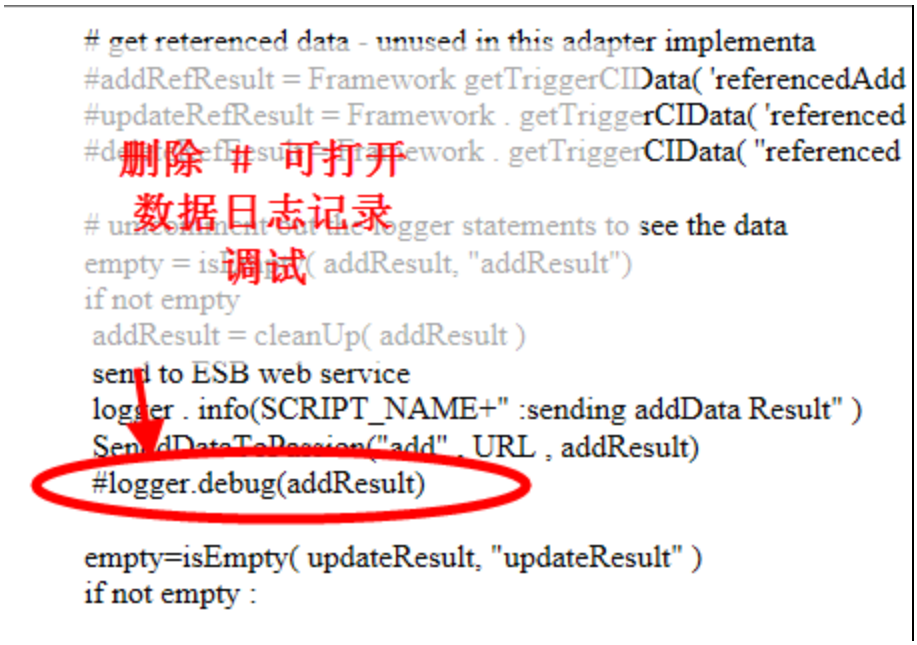
SOAPUI Eclipse 插件用于在开发期间测试 Web 服务层。可以使用 SOAPUI 帮助您自定义 Web 服务。SOAPUI 提供集成的开发环境 (IDE) 来测试 SOAP 消息的构建、发送和接收。在 SOAPUI 透视中，第 208-212 页上的 WSDL 生成了以下示例消息：

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ucm="http://ucmdb.hp.com">
    <soapenv:Header/>
    <soapenv:Body>
        <ucm:addData>
            <ucm:xmlAdded>?</ucm:xmlAdded>
        </ucm:addData>
    </soapenv:Body>
</soapenv:Envelope>
```

上述 **xmlAdded** 元素中的“?”是指数据的位置，该属性由 Web 服务推送适配器集成提供。

观察结果

推送适配器在非调试模式下正常运行时，直到写入最终结果，系统才将数据写入到文件（在所有日志文件中，通常不显示中间 TQL 结果和映射数据结果）。但是，通过取消注释 DiscoveryMain 部分中的 **logger.debug** 语句（删除“#”字符），可以将结果写入到探测器的调试文件，如下所示：



请确保记录器语句从与其他前面行和后面行相同的列开始。Jython 区分缩进，如果所有行的缩进不正确，则脚本将失败。

此处探测器上的调试日志文件 probeMgr-adaptersDebug.log 显示了输出的内容:

```
<2011-12-07 14:02:23,019> [INFO ] [Thread-273] - XMLtoWebService.py started

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - ESB Push parameters:

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - Wshost=harpy.trtc.com

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - WShostport=5555

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] -
WSuri=ws/DtITServiceManagement.esla.v1.ws.provider:UMDBDataReceiver

<2011-12-07 14:02:23,019> [INFO ] [Thread-273] - URL is
http://harpy.trtc.com:5555/ws/DtITServiceManagement.esla.v1.ws.
provider:UMDBDataReceiver

<2011-12-07 14:02:23,035> [DEBUG] [Thread-273] - Connected to
http://harpy.trtc.com:5555/ws/DtITServiceManagement.esla.v1.ws.
provider:UMDBDataReceiver

<2011-12-07 14:02:23,035> [ERROR] [Thread-273] - sending results

<2011-12-07 14:02:23,035> [DEBUG] [Thread-273] - <?xml version="1.0" encoding="UTF-8"?>

<root>

    <data>

        <objects>

            <Object mode="" name="u_imp_ip_switch" operation="add"
            mamId="9e8c2f6bdfe4b7d0864c79e70833902c">

                <field name="Correlation ID" key="true" datatype="char"
                length="">9e8c2f6bdfe4b7d0864c79e70833902c</field>

                <field name="name" key="false" datatype="char" length="">nma_
                09sw</field>

                <field name="location" key="false" datatype="char" length="" />

                <field name="u_chassis_vendor_type" key="false" datatype="char"
                length="">ciscoCat2960-24TT</field>

                <field name="serial_number" key="false" datatype="char" length="" />

                <field name="ram" key="false" datatype="char" length="" />

                <field name="os_version" key="false" datatype="char" length="" />

            </Object>
```

修改 Jython 脚本

XMLtoWebService.py

Web 服务推送适配器使用的 Jython 脚本与 XML 推送适配器使用的脚本极为相似。该脚本使用 **UCMDBDataReceiver.jar**（随附在适配器中）。该脚本实现 **SendDataToReceiver()** 方法。**SendDataToReceiver()** 使用三个参数:

1. 操作（add、update 或 delete）
2. 数据接收器的 URL
3. 数据

例如，添加块如下所示：**SendDataToReceiver(“add”, URL, addResult)**

所有 Web 服务和 SOAP 层均已打包。URL 是 UCMDB 数据接收器的服务端点地址。此 URL 与用于通过“?wsdl”后缀获得 wsdl 的 URL 相同。

Jython 脚本的源代码如下所示。Web 服务集成打包行以绿色突出显示。

```
#####################################

# script:XMLtoWebService.py

#####################################

# This jython script accepts TQL data results (adds, updates, and deletes) from the Integration adapter.

# and sends it to a web service.The web service is called UCMDBDataReceiver.

# A web service client of this name must be addressable at the URL provided by the parameters.

# The SendDataToReceiver.jar exposes the SendDataToReceiver function, as well as the service locator.

# examples of the service locator are in the testconnection section.

# regular expressions

import re

# logging

import logger

# web service interface

from com.hp.ucmdb import SendDataToReceiver

from com.hp.ucmdb.SendDataToReceiver import locateService

from com.hp.ucmdb.SendDataToReceiver import SendData

################################################

########    VARIABLES        ##########

################################################
```

```
SCRIPT_NAME = "XMLtoWebService.py"
logger.info(SCRIPT_NAME+" started")
def cleanUp(str):

        # replace mode=""
        str = re.sub("mode=\"\w+\"\s+", "", str)

        # replace mamId with id
        str = re.sub("\smamId=\"", " id=\"", str)

        # replace empty attributes
        str = re.sub("[\n|\s|\r]*<field name=\"\w+\" datatype=\"\w+\" />", "", str)

        # replace targetRelationshipClass with name
        str = re.sub("\stargetRelationshipClass=\"", " name=\"", str)

        # replace Object with object with name
        str = re.sub("<Object mode=\"", "<object mode=\"", str)
        str = re.sub("<Object operation=\"", "<object operation=\"", str)
        str = re.sub("<Object name=\"", "<object name=\"", str)
        str = re.sub("</Object>", "</object>", str)

        # replace field to attribute
        str = re.sub("<field name=\"", "<attribute name=\"", str)
        str = re.sub("</field>", "</attribute>", str)

        #logger.debug("String = %s" % str)
        #logger.debug("cleaned up")

        return str
def isEmpty(xml, type = ""):
```

```
        objectsEmpty = 0
        linksEmpty = 0

        m = re.findall("<objects />", xml)
        if m:
                #logger.warn("\t[%s] No objects found" % type)
                objectsEmpty = 1

        m = re.findall("<links />", xml)
        if m:
                #logger.warn("\t[%s] No links found" % type)
                linksEmpty = 1

        if objectsEmpty and linksEmpty:
                return 1
        return 0
################################################
########     MAIN            ###########
################################################
def DiscoveryMain(Framework):
        #fix this for web service export
        errMsg = "UCMDBDataReceiver Service not found."
        testConnection = Framework.getTriggerCIData("testConnection")
        # Get Web Service Push variables
        WShostName = Framework.getTriggerCIData("Host Name")
        WShostport = Framework.getTriggerCIData("Protocol Port")
        WSuri = Framework.getTriggerCIData("URI")

        logger.info(SCRIPT_NAME+":ESB Push parameters:")
        logger.info("Host Name="+WShostName)
        logger.info("Protocol Port="+WShostport)
```

```
logger.info("URI="+WSuri)
URL = "http://"+WShostName+":"+WShostport+"/"+WSuri
logger.info("URL="+URL)
if testConnection == 'true':
    # locate the service
    test_receiver = SendDataToReceiver()
    locator = test_receiver.locateService(URL)
    #locator = locateService(URL)
    if(locator):
        logger.info(SCRIPT_NAME+":Test connection was successful")
        return
    else:
        raise Exception, errMsg
        return
# do same thing here if not just a test connection -
receiver = SendDataToReceiver()
locator = receiver.locateService(URL)
if(locator):
    logger.info(SCRIPT_NAME+":Connected to "+URL)
else:
    logger.error(SCRIPT_NAME+":no locator")
    raise Exception, errMsg
    return

# get add/update/delete result objects from the Framework
addResult = Framework.getTriggerCIData('addResult')
updateResult = Framework.getTriggerCIData('updateResult')
deleteResult = Framework.getTriggerCIData('deleteResult')
logger.debug(deleteResult)

# get referenced data - unused in this adapter implementation
```

```
#addRefResult = Framework.getTriggerCIData('referencedAddResult')
#updateRefResult = Framework.getTriggerCIData('referencedUpdateResult')
#deleteRefResult = Framework.getTriggerCIData('referencedDeleteResult')
# uncomment out the logger statements to see the data
empty = isEmpty(addResult, "addResult")
if not empty:
        addResult = cleanUp(addResult)
        # send to ESB web service
        logger.info(SCRIPT_NAME+":sending addData Result")
        rcvr = SendDataToReceiver()
        resp = rcvr.SendData("add", URL, addResult)
        logger.info(SCRIPT_NAME+":addData received response:"+resp)
        #logger.debug(addResult)
empty = isEmpty(updateResult, "updateResult")
if not empty:
        updateResult = cleanUp(updateResult)
        # send to ESB web service
        #logger.debug(updateResult)
        logger.info(SCRIPT_NAME+":sending updateData Result")
        rcvr = SendDataToReceiver()
        resp = rcvr.SendData("update", URL, updateResult)
        logger.info(SCRIPT_NAME+":received response:"+resp)


empty = isEmpty(deleteResult, "deleteResult")
if not empty:
        deleteResult = cleanUp(deleteResult)
        # send to ESB web service
        #logger.debug(deleteResult)
        logger.info(SCRIPT_NAME+":sending deleteData Result")
        rcvr = SendDataToReceiver()
        resp = rcvr.SendData("delete", URL, deleteResult)
```

```
            logger.info(SCRIPT_NAME+":received response:"+resp)
        logger.info(SCRIPT_NAME+" ended")
```

自定义响应消息处理

数据接收器应返回包含任何所需响应或状态的字符串。默认情况下，Web 服务推送适配器将响应传递到探测器的信息级别日志。响应消息为内部包含所返回响应字符串的 SOAP 格式的 XML。接收器可以返回分组错误、单个错误或成功消息等任何数据。如果需要对响应进行其他处理，则响应可由适配器的 Jython 脚本处理。无需进行任何 Java 编程。

返回响应消息的示例，使用以下程序发送:

```
// stub example for building your own UCMDBDataData Receiver
public class UCMDBDataReceiver {


public String addData (String xmlAdd){
System.out.println(xmlAdd); // do something with the data
// send back a response message based on what you did
String tr = new String("a test response from addData!");
return tr;
}
```

如下所示:

```
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
      <addDataResponse xmlns="http://ucmdb.hp.com">
            <addDataReturn>a test response from addData!</addDataReturn>
      </addDataResponse>
</soapenv:Body>
```

修改数据接收器

Java 客户端可以实现包含在 **UCMDBDataReceiver.jar** 中的类，并采用与 Jython 相同的方式调用 Web 服务。此外，也可以调用解包的方法。对于 **UCMDBDataReceiver.jar** 类而言，存在 Javadoc。以下源代码显示了如何使用这些必要的方法打包 SOAP 消息中的数据，并通过 HTTP 将其发送到接收器。

该过程可创建 **UCMDBDataReceiverServiceLocator** 对象，然后将 **UCMDBDataReceiverEndPointAddress** 分配给数据接收器的 URL。

要发送数据，请调用定位程序的 **getUCMDBDataReceiver** 方法创建 **UCMDBDataReceiver** 对象。**UCMDBDataReceiver** 对象实现用于实际发送 add/change/delete 数据的方法。系统有三个相同的代码块处理每种类型的请求。

下面列出了 SendDataToReceiver 类的源代码。突出显示的对象和方法是要使用的必要元素。

```
/**
* Test SendData for the UCMDB Data Receiver for the UCMDB Web Service Push Adapter
*/
package com.hp.ucmdb;
import com.hp.ucmdb.SendDataToReceiver;
/**
* TestSendData can be used to verify the SOAP classes are working.
* TestSendData creates a SendDataToReceiver class and invokes its SendData method.
* a response String is returned.
* The test URL is typically appended with "?wsdl" to get the WSDL of the service.
*/
public class TestSendData {
	/**
	* @param args - test SOAP message.
	*  optional arguments [0] a test string [1] a service endpoint URL of a Data Receiver.
	* the default URL is sent the incoming argument as a test message.
	* the default URL is "http://localhost:8080/UCMDBDataReceiver/services/UCMDBDataReceiver".
	* If any errors are encountered, TestClient will attempt to throw exceptions.
	*/
	public static void main(String[] args) {
		// use test message if supplied, otherwise supply a default test string
		String teststring = new String("Test SOAP message from UCMDBDataReceiver
		TestSendData.");
		if(args.length > 0) {
			teststring = args[0];
		}
		// use test URL if supplied, otherwise supply the default URL
		String URL = new String("");
		if(args.length > 1) {
			URL = args[1];
		}
```

```
            // return response
            String response = new String("");
            // perform the tests
            try {
                  if(URL.equals("")) {
                        UCMDBDataReceiverServiceLocator locator = new
                        UCMDBDataReceiverServiceLocator();
                        UCMDBDataReceiver receiver = locator.getUCMDBDataReceiver();
                        URL = locator.getUCMDBDataReceiverAddress();
                        System.out.println("TestClient:tested
                        URL="+locator.getUCMDBDataReceiverAddress());
                        System.out.println("TestClient:receiver="+receiver.toString());
                  }
                  SendDataToReceiver sdtr = new SendDataToReceiver();
                  // this sends a test push and gets a response message
                  response = sdtr.SendData("add", URL, args[0]);
                  System.out.println("Response received was:"+response);
            } catch(Exception e){
                  System.out.println("TestClient:Remote Error:");
                  e.printStackTrace();
            }
      }
}
```

源代码也包括在其他类的 **UCMDBDataReceiver.jar** 文件中:

- TestClient.java
- UCMDBDataReceiver.java
- UCMDBDataReceiverProxy.java
- UCMDBDataReceiverService.java
- UCMDBDataReceiverServiceLocator.java
- UCMDBDataReceiverSoapBindingStub.java

源已在 Eclipse IDE 中生成和修改。修改 UCMDB 代码时请务必小心谨慎，因为这些代码是自动生成的，要与 SOAP 规范和 UCMDB 数据接收器相匹配。

Javadoc

完全注释的 **javadoc** 由常规 Web 服务推送适配器提供。**javadoc** 包含在文档文件夹 **javadoc** 中。以 **index.html** 开头。概述页面可提供 SDK 中所有类和方法的文档访问权限。

所有类

- **SendDataToReceiver：** Web 服务打包的 API
- **TestClient：** 测试客户端，用于验证其与服务端点的连接性
- **UCMDBDataReceiver：** Web 服务打包

以下其余类由 Web 服务生成器自动生成：

- UCMDBDataReceiverProxy
- UCMDBDataReceiverService
- UCMDBDataReceiverServiceLocator
- UCMDBDataReceiverSoapBindingStub

概述

包中的文档将说明 SDK 的基本用法（包括源代码示例）。此 **javadoc** 适用于 UCMDB Web 服务推送适配器。API 可从 Jython 或 Java 调用。

SDK 提供两个源示例：**TestClient** 和 **SendDataToReceiver**。**TestClient** 提供极为有限的响应本地客户端测试。**SendDataToReceiver** 是用于将数据发送到 Web 服务的主类。

首先，使用此 SDK（主要是包含的 WSDL）实现 UCMDB 数据接收器与此 Web 服务进行通信。随后使用此 SDK 在 UCMDB 中创建推送适配器，将 UCMDB TQL 结果数据推送到数据接收器。下面说明了此 API 的基本用法，包括 Jython 和 Java 实现。

实现 SendDataToReceiver()

SendDataToReceiver() 使用一种方法打包所有函数：

- Jython：SendDataToReceiver("add",yourURL,"Hello!")
- Java：SendDataToReceiver("add",yourURL,"Hello!");

或创建 **SendDataToReceiver** 对象（例如，操作其他设置），然后分别调用 **SendData** 方法，如下所示：

- Jython：

  ```
  rcvr = SendDataToReceiver()
  responseMsg = rcvr.SendData(“add”, yourURL, “Hello!”)
  ```

- Java：

  ```
  SendDataToReceiver rcvr = new SendDataToReceiver();
  String responseMsg = rcvr.SendData(“add”, yourURL, “Hello!”);
  ```

如果您需要一次性完成该操作，则可以执行以下操作：

1. 创建新的 **UCMDBDataReceiverServiceLocator()** 对象 x，然后设置该对象的端点地址，如下所示：
   - **Jython：**

     ```
     x = UCMDBDataReceiverServiceLocator()
     ```

x.setUCMDBDataReceiverEndPointAddress(URL)

- **Java:**

  UCMDBDataReceiverServiceLocator x = new UCMDBDataReceiverServiceLocator();

  x. setUCMDBDataReceiverEndPointAddress(URL);

2. 接着采用以下方式创建 UCMDBDataReceiver
   - **Jython:** y = x.getUCMDBDataReceiver()
   - **Java:** UCMDBDataReceiver y = x.getUCMDBDataReceiver();
3. 然后通过如下所示的 SOAP Web 服务发送数据:
   - **Jython:**
     - y.addData(yourData)
     - 或 y.updateData(yourData)
     - 或 y.deleteData(yourData)
   - **Java:**
     - y.addData(yourData);
     - 或 y.updateData(yourData);
     - 或 y.deleteData(yourData);
4. 您可能需要测试连接性，如果测试成功，则重用同一定位程序对象返回 **UCMDBDataReceiver** 用于数据传输。

该类不包含任何析构函数，且不进行内存管理。

映射文件引用

使用映射

必须在已转换的 XML 输出中为每个目标属性创建映射。映射将指定获取数据的位置和方式。如果数据位于 UCMDB 中的其他对应属性中，则将使用直接映射。

要从多个属性中拉出数据，或从 UCMDB 的子 CI 或父 CI 属性中拉出属性，则可能需要使用其他复杂映射。以下映射架构显示了所有可能的映射。

映射文件是一种 XML 文件，可定义将 UCMDB 中的哪些 CI/关系类型映射到目标数据存储中的哪些 CI/关系类型。以下是格式的详细说明。映射文件可控制要推送的 CI 和关系类型，以及准确控制要推送的属性。

映射条目针对要推送到目标 MDR 的每个属性而存在。每个映射条目可能包括原始 UCMDB 推送数据中的一个或多个属性。对于要推送到目标 MDR 的数据的最终结构和命名，映射条目支持对其进行完全的细化控制。

直接映射

映射可将一种数据模型转换为另一种数据模型（在这种情况下，将 UCMDB 转换为推送目标 MDR）。转

换操作可能十分简单，如果 UCMDB 属性和目标之间是一对一的关系，则仅名称（可能还有类型）不同。

大多数属性映射都是直接映射。例如，可以将服务器名称“ServerX”在 UCMDB 中表示为 **unix** 类型的 CI，其属性名称为 **primary_server_name**，类型为 **string** 且长度为 50。目标 MDR 的数据模型可以指定 CI 类型为 **linux** 的相同逻辑实体，其属性名称为 **hostname**，类型为 **char[]** 且最大长度为 250。直接映射可以完成所有上述类型的转换任务。

以下是直接映射的示例：

```
<target_attribute name="dns_domain" datatype="char">
<map type="direct" source_attribute="domain_name" />
</target_attribute>
```

此直接映射将 UCMDB 属性 **dns_domain** 映射到目标数据模型中的 **domain_name** 属性。

无论实际数据类型是什么，均使用 **char** 数据类型，除非需要使用实际数据类型。

复杂映射

大多数复杂映射均支持其他转换：

- 将属性值从多个 CI 映射到一个目标 CI。
- 将子 CI（具有 **container_f** 或包含关系的 CI）的属性映射到目标数据存储中的父 CI。例如，在目标主机 CI 中设置名为“CPU 数量”的值。另一个示例是在目标主机 CI 中设置值“总内存”（通过合计 UCMDB 中主机 CI 的所有内存 CI 的内存大小值）。
- 将父 CI（具有 **container_f** 或包含关系的 CI）的属性映射到目标数据存储中的 CI。例如，通过从 UCMDB 中软件 CI 的包含主机中获取值，在名为“安装的软件”CI 的目标属性中，设置名为“容器服务器”的值。

以下是复杂映射的示例，采用由逗号字符分隔的两个源属性，创建目标属性 **os**：

```
<target_attribute name="os" datatype="char">
    <map type="compoundstring">
        <source_attribute name="discovered_os_name" />
        <constant value="," />
        <source_attribute name="host_osinstalltype" />
    </map>
</target_attribute>
```

反转链接方向

对于不同的源来说，UCMDB 包含的数据可能在结构上各有不同。例如，IpAddress CI 和接口 CI 之间的关系可能是“父级”，这种情况也可能出现在 HP 网络节点管理器集成中。或者，它可能是通常由 Universal Discovery 创建的“包含”链接。此外，这些链接的方向可能是相反的。

当前无法反转映射文件中的链接方向。反转 **_end1** 和 **_end2** 变量可切换已转换 XML 中的数据顺序，也可导致源数据中缺少链接。

此问题的一个可能解决方案是定义扩展规则，具体操作如下：

1. 扩展的 TQL 部分是推送适配器使用的 TQL 子集。此 TQL 专门选择已转换 XML 中所需的按相反方向排列的所有链接。
2. 扩展部分将定义具有正确方向和所需类型的新链接。
3. 扩展此时已激活，并且稍后将创建正确的链接。
4. 集成作业 TQL 现在将引用已扩展的链接，而不是原始链接。
5. 推送适配器中的 <链接> 映射稍后也将引用已扩展的链接，并生成类型和方向一致的链接集。

映射文件架构

元素名称和路径	描述	属性
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。	1. **名称：** type **描述：** 源数据库的名称。 **是否必需？：** 必需 **类型：** 字符串 2. **名称：** versions **描述：** 源数据库的版本。 **是否必需？：** 必需 **类型：** 字符串 3. **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
target (integration > info)	定义有关目标数据库的信息。	1. **名称：** type **描述：** 源数据库的名称。 **是否必需？：** 必需 **类型：** 字符串 2. **名称：** versions **描述：** 源数据库的版本。 **是否必需？：** 必需 **类型：** 字符串 3. **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

元素名称和路径	描述	属性
targetcis (integration)	所有 CIT 映射的容器元素。	
source_ci_type_tree (integration > targetcis)	定义源 CIT 以及从其继承的所有 CI 类型。	1. **名称：** name **描述：** 源 CIT 的名称。 **是否必需？：** 必需 **类型：** 字符串 2. **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： a. **insert**：只有当 CI 不存在时才使用此字符串。 b. **update**：只有当 CI 存在时才使用此字符串。 c. **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。 d. **ignore**：不对此 CI 类型执行任何操作。
source_ci_type (integration > targetcis)	定义源 CIT，但不定义从其继承的 CI 类型。	1. **名称：** name **描述：** 源 CIT 的名称。 **是否必需？：** 必需 **类型：** 字符串 2. **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： a. **insert**：只有当 CI 不存在时才使用此字符串。 b. **update**：只有当 CI 存在时才使用此字符串。 c. **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。 d. **ignore**：不对此 CI 类型执行任何操作。
target_ci_type (integration > targetcis > source_ci_type 或 integration > targetcis > source_ci_type_tree)	定义目标 CIT。	1. **名称：** name **描述：** 目标 CI 类型名称。 **是否必需？：** 必需 **类型：** 字符串 2. **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 3. **名称：** namespace **描述：** 表示此 CI 类型在目标上的命名空间。

元素名称和路径	描述	属性
		是否必需？： 非必需 **类型：** 字符串
targetprimarykey (integration > targetcis > source_ci_type) **或** (integration > targetcis > source_ci_type_tree **或** (integration > targetrelations > link) **或** (integration > targetrelations > source_link_type_tree)	表示目标 CIT 主键属性。	
pkey (integration > targetcis > source_ci_type > targetprimarykey **或** integration > targetcis > source_ci_type_tree > targetprimarykey **或** (integration > targetrelations > link > targetprimarykey) **或** integration > targetrelations > source_link_type_tree >	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。	

元素名称和路径	描述	属性
targetprimarykey)		
target_attribute (integration > targetcis > source_ci_type 或 integration > targetcis > source_ci_type_tree 或 integration > targetrelations > link 或 integration > targetrelations > source_link_type_tree)	定义目标 CIT 的属性。	1. **名称：** name **描述：** 目标 CIT 属性的名称。 **是否必需？：** 必需 **类型：** 字符串 2. **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 3. **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 4. **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： a. **uppercase** – 转换成大写 b. **lowercase** – 转换成小写 如果此属性为空，则不应用转换函数。
map (integration > targetcis > source_ci_type > target_attribute 或 integration > targetcis > source_ci_type_tree > target_attribute) 或 (integration > targetrelations > link > target_attribute 或 integration > targetrelations >	指定如何获取源 CIT 的属性值。	1. **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： a. **direct** – 指定从源属性值到目标属性值的一对一映射。 b. **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。 c. **childattr** – 子元素是一个或多个子 CIT 的属性。子 CIT 被定义为具有 **composition**或 **containment** 关系的 CIT。 d. **constant** – 静态字符串 2. **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 3. **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

元素名称和路径	描述	属性
source_link_type_tree > target_attribute)		
aggregation (integration > targetcis > source_ci_type > target_attribute > map 或 integration > targetcis > source_ci_type_tree > target_attribute > map 或 (integration > targetrelations > link > target_attribute > map 或 integration > targetrelations > source_link_type_tree > target_attribute > map) 只有当映射类型为 **childattr** 时才有效	指定如何将来源 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总和。 • **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。
source_child_ci_type (integration > targetcis > source_ci_type > target_attribute > map 或 integration > targetcis > source_ci_type_tree > target_attribute > map	指定从中获取子属性的已连接 CI。	1. **名称。**name **描述。**子 CI 的类型 **是否必需。**必需 **类型。**字符串 2. **名称。**source_attribute **描述。**映射的子 CI 的属性。 **是否必需。**只有 **childAttr** 聚合类型（在同一路径上）不是 = count 时才是必需的。 **类型。**字符串

元素名称和路径	描述	属性
或 (integration > targetrelations > link > target_ attribute > map 或 integration > targetrelations > source_link_type_ tree > target_ attribute > map) 只有当映射类型为 **childattr** 时才有效。		
validation (integration > targetcis > source_ci_type > target_attribute > map 或 integration > targetcis > source_ci_type_ tree > target_attribute > map 或 (integration > targetrelations > link > target_attribute > map 或 integration > targetrelations > source_link_type_ tree > target_attribute > map) 只有当映射类型为	允许根据属性值对来源 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。	1. **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 2. **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 3. **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 4. **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

元素名称和路径	描述	属性
childatt 时才有效		
targetrelations (integration)	所有关系映射的容器元素。可选。	
source_link_type_tree (integration > targetrelations)	将源关系类型映射到目标关系，但不映射从源关系类型继承的类型。只有当 **targetrelation** 存在时才会强制执行。	1. **名称：** name **描述。**源关系名称。 **是否必需？：** 必需 **类型。**字符串 2. **名称：** target_link_type **描述。**目标关系名称 **是否必需？：** 必需 **类型。**字符串 3. **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 4. **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。 5. **名称：** source_ci_type_end1 **描述：**源关系的端 1 CI 类型。 **是否必需？：** 必需 **类型：**字符串 6. **名称：** source_ci_type_end2 **描述：**源关系的端 2 CI 类型。 **是否必需？：** 必需 **类型：**字符串
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。	1. **名称：** source_link_type **描述：**源关系名称。 **是否必需？：** 必需 **类型：**字符串

元素名称和路径	描述	属性
		2. **名称：** target_link_type **描述：** 目标关系名称。 **是否必需？：** 必需 **类型：** 字符串 3. **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 4. **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。 5. **名称：** source_ci_type_end1 **描述**：源关系的端 1 CI 类型 **是否必需？：** 必需 **类型：** 字符串 6. **名称：** source_ci_type_end2 **描述：** 源关系的端 2 CI 类型 **是否必需？：** 必需 **类型：** 字符串
target_ci_type_end1 (integration > targetrelations > link 或 integration > targetrelations > source_link_type_tree)	目标关系的端 1 CI 类型。	1. **名称：** name **描述：** 目标关系的端 1 CI 类型的名称。 **是否必需？：** 必需 **类型：** 字符串 2. **名称：** superclass **描述：** 端 1 CI 类型的超类的名称。 **是否必需？：** 非必需 **类型：** 字符串
target_ci_type_end2	目标关系的端 2 CI 类型。	1. **名称：** name **描述：** 目标关系的端 2 CI 类型的名称。 **是否必需？：** 必需

元素名称和路径	描述	属性
(integration > targetrelations > link 或 integration > targetrelations > source_link_type_tree)		**类型：** 字符串 2. **名称：** superclass **描述：** 端 2 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

元素名称和路径	描述	属性
root	结果文档的根。	
data (root)	数据自身的根。	
objects (root > data)	要更新对象的根元素。	
Object (root > data > objects)	描述单个对象及其所有属性的更新操作。	1. **名称：** name **描述：** CI 类型的名称。 **是否必需？：** 必需 **类型：** 字符串 2. **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: a. **insert** – 只有当 CI 不存在时才使用此字符串。 b. **update** – 只有当 CI 存在时才使用此字符串。 c. **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。 d. **ignore** – 不对此 CI 类型执行任何操作。 3. **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: a. **add** – 将添加此 CI b. **update** – 将更新此 CI

元素名称和路径	描述	属性
		c. **delete** – 将删除此 CI 如果不设置值，则会使用默认值 **add**。 4. **名称：** mamId **描述：** 源 CMDB 上对象的 ID。 **是否必需？：** 必需 **类型：** 字符串
field (root > data > objects > Object 或 root > data > links > link)	描述对象的一个字段的值。字段文本是字段中的新值，如果字段包含链接，则值为某链接端的 ID。各端 ID 作为对象出现（在 <objects> 下）。	1. **名称：** name **描述：** 字段的名称。 **是否必需？：** 必需 **类型：** 字符串 2. **名称：** key **描述：** 指定此字段是否为对象的键。 **是否必需？：** 必需 **类型：** 布尔型 3. **名称：** datatype **描述：** 字段的类型。 **是否必需？：** 必需 **类型：** 字符串 4. **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
links (root > data)	要更新链接的根元素。	1. **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 2. **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 3. **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 4. **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: a. **insert** – 只有当 CI 不存在时才使用此字符串。 b. **update** – 只有当 CI 存在时才使用此字

元素名称和路径	描述	属性
		符串。 c. **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。 d. **ignore** – 不对此 CI 类型执行任何操作。 5. **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： a. add – 将添加此 CI b. update – 将更新此 CI c. delete – 将删除此 CI 如果不设置值，则会使用默认值 **add**。 6. **名称：** mamId **描述：** 源 CMDB 上对象的 ID。 **是否必需？：** 必需 **类型：** 字符串

自定义

本节介绍推送适配器的常见自定义类型的一些基本过程。

添加属性

1. 请确保属性已包含在 TQL 结果中。
2. 在正确的 CI 映射部分，将属性映射添加到映射文件中。
3. 请确保数据接收器已准备好接收数据中的其他属性。

删除属性

要删除属性，请从映射文件中删除此属性。如果此属性不再用于结果或不再用作条件节点，您还应从 TQL 中删除此属性。

添加 CI 类型

1. 将 CI 类型添加到 TQL。
2. 请确保 CI 类型及其属性数据会显示在 TQL 结果中（使用计算和预览）。
3. 将 CI 类型的映射添加到映射文件中。复制其他 CI 类型的映射以快速创建新的 CI 类型。
4. 修改复制的 XML 的名称和属性映射，与新的 CI 类型及其属性相对应。有关可用的映射类型，请参阅映射文件引用 (第 224 页)。

删除 CI 类型

1. 从 TQL 删除 CI 类型
2. 删除位于映射文件中的 CI 类型的映射部分。

添加链接

1. 请确保两端的 CI 均出现在数据中。
2. 请确保需要添加的链接实际上是有效链接（在 CI 类型管理器中检查）。
3. 在映射 XML 的关系部分添加链接元素。

删除链接

1. 删除要在映射文件中删除的链接的链接部分。
2. 请尽量从 TQL 删除链接（除非会影响 TQL 的效率或功能）。

第 8 章: 开发常规适配器

本章包括:

- 实例同步 238
- 使用常规适配器实现数据推送 238
- 使用常规适配器实现数据填入 246
- 使用常规适配器实现数据联合 259
- 调节 272
- 常规适配器 API 272
- 资源定位器 API 273
- 创建常规适配器包 273
- 推送和填入映射之间的差异 278
- 常规适配器日志文件 278
- 使用常规适配器框架的适配器 279
- 常规适配器 XML 架构参考 279

实例同步

推送和填入常规适配器操作使用实例数据。有关*实例*和*根*概念的详细信息，请参阅基于实例的填入流程 (第 169 页)和使用常规适配器实现数据推送 (第 238 页)。

使用常规适配器实现数据推送

数据推送将使用稍微更改了 XML 架构的现有增强的常规推送适配器框架。

备注: 常规适配器在实例模式下工作（即不使用单个 CI 类型，而使用由主根 CI 分组到一起的 CI 集合）。有关详细信息，请参阅基于实例的填入流程 (第 169 页)。

为适应双向映射语义所需进行的 XML 架构更改如下:

- **<targetcis>** 标记已重命名为 **<target_entities>**。
- **<source_instance_type>** 标记已重命名为 **<source_instance>**。
- **<target_ci_type>** 标记已重命名为 **<target_entity>**。
- **<for-each-source-ci>** 标记已重命名为 **<for-each-source-entity>**。
- 标头 **versions** 属性已重命名为 **version**，且不再需要是十进制。

本节提供有关使用常规适配器框架推送数据的信息:

- 推送概述 239

- 映射文件 239
- Groovy Traveler 241
- 编写 Groovy 脚本 244
- 实施 PushAdapterConnector 接口 245

推送概述

常规适配器适用于表示 TQL 查询结果的数据结构。在常规适配器框架上生成的每个适配器均会处理此数据结构，并将其推送至所需目标。

该数据结构名为“ResultTreeNode (RTN)”。RTN 将根据适配器的映射文件和 TQL 查询结果创建。常规适配器框架使用的查询必须以根为基础，即：查询必须包含一个元素名称为 **root** 的查询节点，或者包含一个或多个以前缀 **root** 开头的关系元素。这一 CI 或关系相当于查询的根元素。有关详细信息，请参阅《HP Universal CMDB 数据流管理指南》中的“数据推送”。

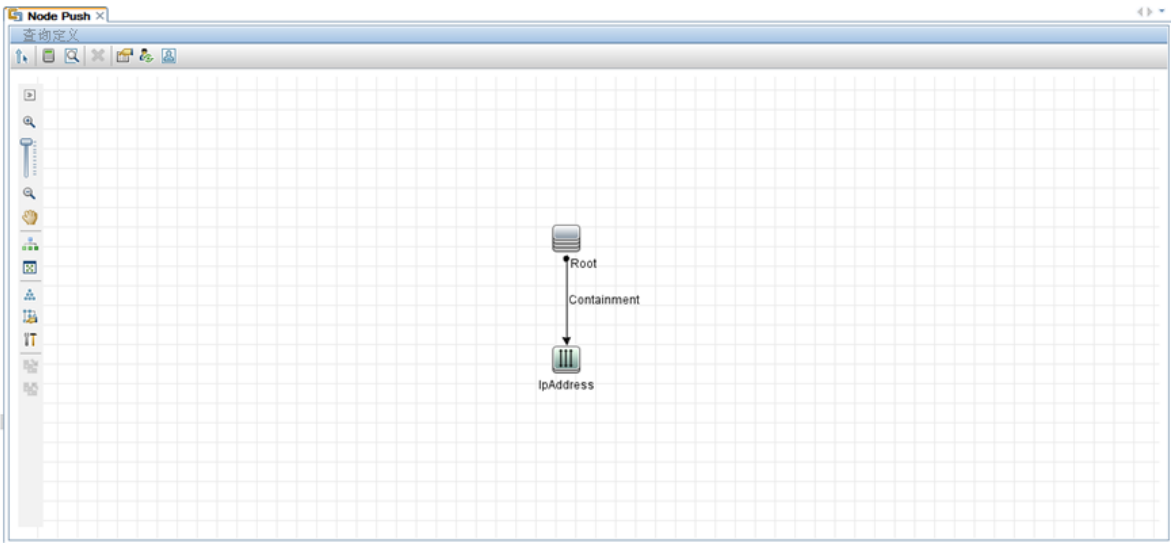
开发增强的推送适配器包括以下两个基本步骤：

1. 实施 PushAdapterConnector 接口 - 此接口将接收作为 RTN 列表而添加、更新和删除的数据，并针对目标执行推送操作。
2. 创建映射文件 – 通过映射 CI 和 TQL 结果属性，映射文件将确定 RTN 结构的创建方法。

映射文件

以下示例演示如何创建映射文件。

在此示例中，我们将模拟节点和 IP 地址的推送。我们将创建一个名为 **Node Push** 的 TQL 查询，如下所示：

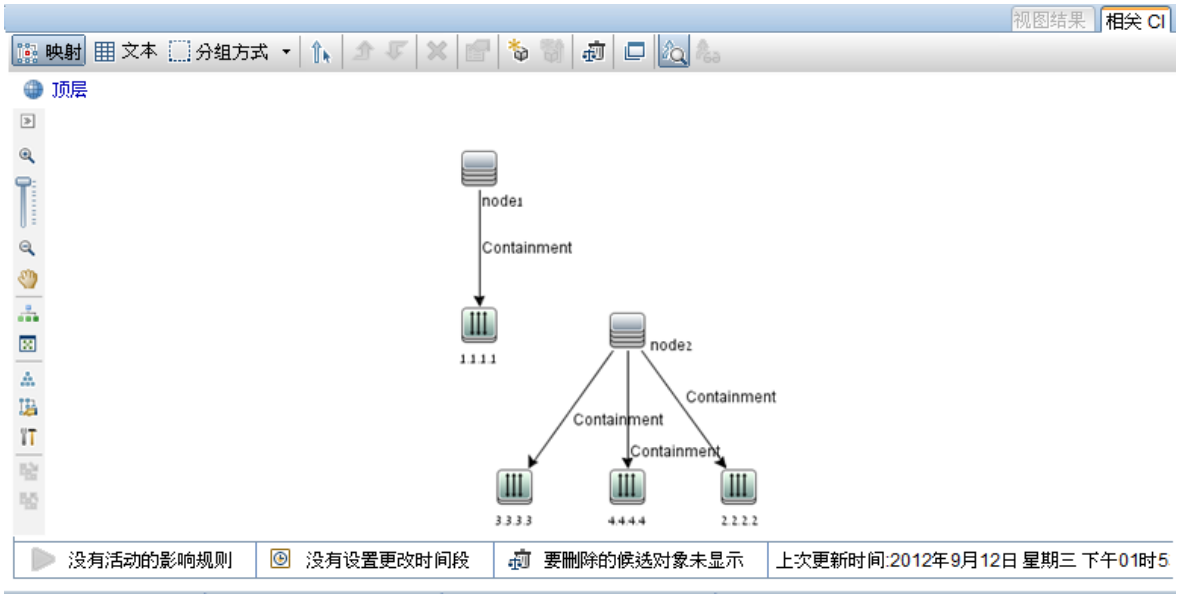


在映射文件中，我们创建两个目标 CI 类型：“Computer”和“IP”。“Computer”有一个变量和两个属性。“IP”有一个属性。

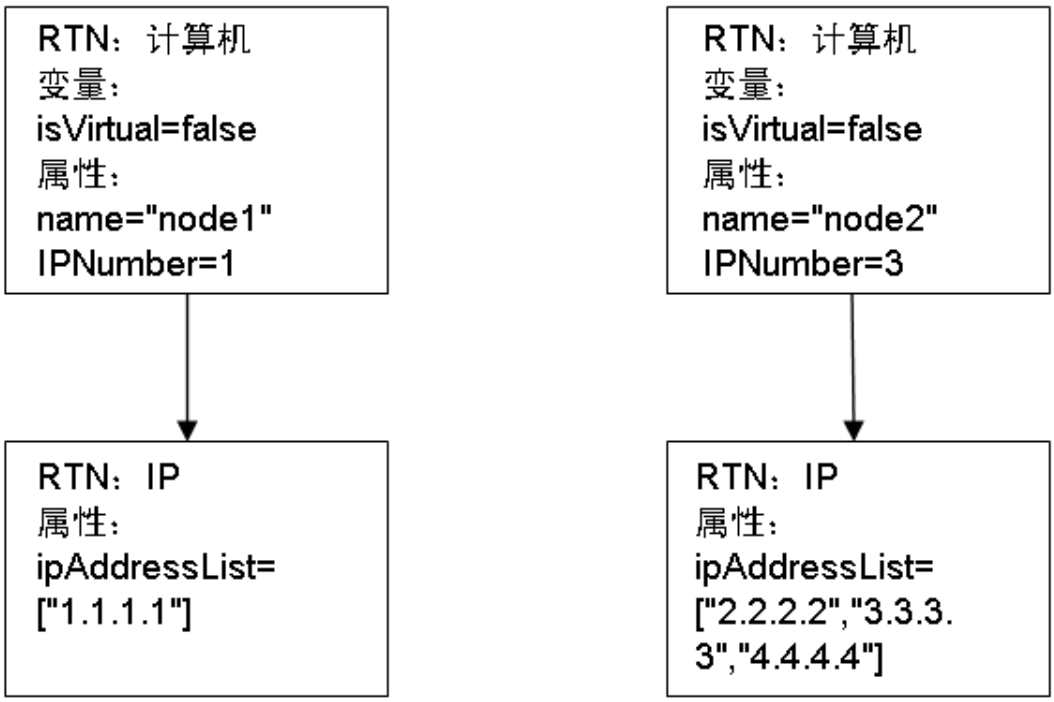
以下是映射的 XML 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <info>
        <source name="UCMDB" version="10.20" vendor="HP"/>
        <target name="PushProduct" version="9.3" vendor="HP"/>
    </info>

    <import>
        <scriptFile path="mappings.scripts.PushFunctions"/>
    </import>

    <target_entities>
        <source_instance query-name="Node Pusy" root-element-name="Root">
            <target_entity name="Computer" is-valid="(Root['root_iscandidatefordeletion'] == null) ? true : !Root['root_iscandidatefordeletion']">
                <variable name="isVirtual" datatype="BOOLEAN" value="PushFunctions.isVirtual(Root['root_class'])"/>
                <target_mapping name="name" datatype="STRING" value="Root['name']"/>
                <target_mapping name="ipNumber" datatype="INTEGER" value="Root.IpAddress.size()"/>
                <target_mapping name="description" datatype="STRING" value="PushFunctions.getDescription(isVirtual)"/>
                <target_entity name="IP">
                    <target_mapping name="IpAddressList" datatype="STRING_LIST" value="Root.IpAddress*.getAt('name')"/>
                </target_entity>
            </target_entity>
        </source_instance>
    </target_entities>

</integration>
```

查询结果如下所示:



以下是根据此映射文件生成的 RTN 列表:



每个根实例都是使用映射文件单独映射的。因此，在此示例中，PushAdapterConnector 将收到两个 RTN 根的列表。

备注: 以前的推送适配器能够为 CI 类型创建常规映射。新的推送适配器将按 TQL 查询进行映射。当运行使用名为 **x** 查询的推送作业时，适配器将查找相关的映射文件（即：具有属性 query-name=x 的文件）。

您可以使用 Groovy 脚本语言计算映射文件中的值。有关详细信息，请参阅Groovy Traveler (第 241 页)。

Groovy Traveler

通过以下方式访问 TQL 查询结果：

- **Root[attr]** 将返回 Root 元素的属性 **attr**。
- **Root.Query_Element_Name** 将返回 TQL 中已命名为 Query_Element_Name、且链接至当前根 CI 的 CI 实例列表。
- **Root.Query_Element_Name[2][attr]** 将返回链接至当前根 CI 的第三个 Query_Element_Name 的属性 **attr**。
- **Root.Query_Element_Name*.getAt(attr)** 将返回 TQL 中已命名为 Query_Element_Name、且链接至当前根 CI 的 CI 实例属性 **attr** 的列表。

Groovy Traveler 还可以访问其他属性：

- **cmdb_id** – 将 CI 或关系的 UCMDB ID 作为字符串返回。
- **external_cmdb_id** – 将 CI 或关系的外部 ID 作为字符串返回。

- **Element_type** – 将 CI 或关系的元素类型作为字符串返回。

“import”标记：

```
<import>
<scriptFile path="mappings.scripts.PushFunctions"/>
</import>
```

这表示我们正在声明导入映射文件中的所有 groovy 脚本。在此示例中，**PushFunctions** 是一个包含部分静态函数的 groovy 脚本文件，我们可以在映射过程中加以访问（即 value=” PushFunctions.foo ()” ）。

source_instance_type

每个 TQL 均执行一次映射，查询名称值为当前映射的相关 TQL。“*”表示此映射文件与以 **Node Push** 前缀开头的所有 TQL 查询关联。

```
<source_instance_type query-name="Node Push*" root-element-name="Root">
```

source_instance_type 标记将指定映射的根元素。

root-element-name 应与根在 TQL 中的名称完全相同。

target_entity

此标记用于创建 RTN。

名称属性表示 target_entity 名称：name=Computer

is-valid 属性是映射过程中计算得出的布尔值，它可确定当前 target_ci 是否有效。无效的 target_ entities 不会添加到 RTN 中。在此示例中，我们无需创建其 **root_iscandidatefordeletion** 属性在 UCMDB 中为“True”的 target_entity 实例。

target_entity 可能会有一些在映射过程中计算得出的变量：

```
<variable name="vSerialNo" datatype="STRING" value="Root['serial_number']"/>
```

变量 **vSerialNo** 可求出当前根的 **serial_number** 值。

RTN 属性由 **target_mapping** 标记创建。Groovy 脚本在“value”字段中的执行结果将分配给 RTN 属性。

```
<target_mapping name="SerialNo" datatype="STRING" value="vSerialNo"/>
```

SerialNo 将分配变量 **vSerialNo** 的值。

可以将 target_entity 定义为另一个 target_entity 的子项，如下所示：

```
<target_entity name="Portfolio">
<variable name="vSerialNo" datatype="STRING" value="Root['global_id']"/>
<target_mapping name="CMDBId" datatype="STRING" value="globalId"/>
```

```
<target_entity name=Asset">
<target_mapping name="SerialNo" datatype="STRING" value="vSerialNo"/>
</target_entity>
</target_entity>
```

RTN **产品组合**将有名为“Asset”的子 RTN。

for-each-source-entity

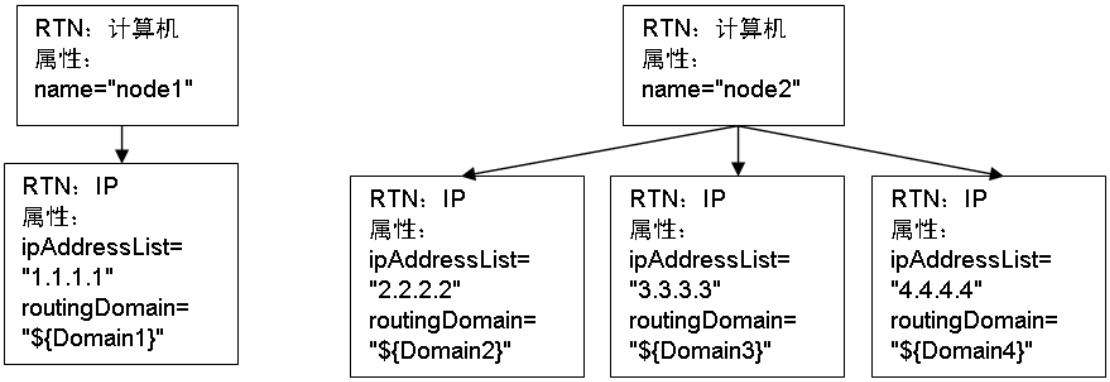
此标记将列出根实例的特定 CI。它具有以下字段：

- **source-entities=""** – 已创建目标 CI 的 CI 列表。此列表由 Groovy Traveler 在 **Root.IpAddress** 字段中定义。
- **count-index=""** – 用于保存当前循环迭代中 CI 索引的变量。
- **var-name=""** – CI 在当前循环迭代中的名称。

下面，我们一起来修改映射示例文件：

```
<target_entity name="Computer">
    <target_mapping name="name" datatype="STRING" value="Root['name']"/>
    <for-each-source-entity count-index="i" var-name="currIP" source-entities="Root.IpAddress">
        <target_entity name="IP">
            <target_mapping name="ipAddress" datatype="STRING" value="Root.IpAddress[i]['name']"/>
            <target_mapping name="routingDomain" datatype="STRING" value="currIP['routing_domain']"/>
        </target_entity>
</target_entity>
```

根据上述映射文件构建的 RTN 列表如下所示：



dynamic_mapping

此标记能够在 RTN 结构的创建期间从目标数据库中创建数据映射。

示例：假设目标数据库有一个表，名为 **Computer**。此表的“id”列和“name”列与 UCMDB 中的 **Node.name** 关联。这两列均具有唯一性。此数据库还有一个表，名为 **IP**，其中包含对 Computer 表中 **parentID** 的引用键。“dynamic_mapping”可创建一个将 name 和 ID 存储为 <name,id> 的映射。根据此映射，适配器可将 ID 与计算机匹配，并将正确的值推送到 IP 表中的 **parentID** 属性。您可以在创建 RTN 的同时，使用此映射将值分配给 parentID 属性。

此映射由 **map_property** 确定。每个块执行一次 dynamic_mapping。

```
<dynamic_mapping name="IdByName " keys-unique="true">
```

name 属性表示映射的名称。**keys-unique** 属性表明键是否唯一（每个键映射到一个值，或者一组值）。

在此示例中，映射的名称为 **IdByName**，它具有唯一键。要使用脚本访问映射，请执行以下命令：

```
DynamicMapHolder.getMap（‘IdByName’）
```

它将返回对该映射的引用。

map_property 标记将创建映射的基础属性。

示例：

```
<map_property property-name="SQLQuery" datatype="STRING"
property-value="SELECT name, id FROM Computer"/>
```

在此示例中，属性的名称为 **SQLQuery**，其值为创建映射的 SQL 语句。PushConnector 接口的 **retrieveUniqueMapping** 和 **retrieveNonUniqueMapping** 方法的实施将确定返回映射的实际内容。

全局变量

映射文件中的 Groovy 脚本可访问以下全局变量：

- **Topology** – 类型：Topology。当前块的拓扑实例。
- **QueryDefinition** - 类型：QueryDefinition。当前 TQL 的查询定义实例。
- **OutputCI** – 类型：ResultTreeNode。根元素在当前树映射中的 RTN。
- **ClassModel** – 类型：ClassModel。类模型的实例。
- **CustomerInformation** – 类型：CustomerInformation。指运行该作业的客户的相关信息。
- **Logger** – 类型：DataAdapterLogger。此记录器可在适配器中用于将日志写入 UCMDB 日志记录框架。

编写 Groovy 脚本

在本节中，我们将创建 **PushFunctions.groovy** 文件。此文件将包括在映射根实例过程中使用的静态函数。

```
package mappings.scripts

public class PushFunctions {

    public static boolean isVirtual(def nodeRole){
        return isListContainsOne(def list, "MY_VM", "MY_SIMULATOR");
    }
```

```
public static String getDescription(boolean isVirtual){
        if(isVirtual){
            return "This is a VM";
        }
        else{
            return "This is physical machine";
        }
    }



private static boolean isListContainsOne(def list, ...stringList){
        //returns true if the list contains one of the values.
    }
}
```

实施 PushAdapterConnector 接口

实施过程应支持以下基本步骤:

```
public class PushExampleAdapter implements PushAdapterConnector
{

public UpdateResult pushTreeNodes(PushConnectorInput input) throws DataAccessException{

// 1. build an UpdateResult instance - the UpdateResult is used to return mappings between the
sent ids to the actual ids that entered the data store.
// Also has an update status which allows to pass the status of data that was actually pushed,
detailed status reports on failed IDs, and actions actually performed on successful ids.
// 2. handle the data:
// a. handle data to add. Can be retrieved by:input.getResultTreeNodes.getDataToAdd();
//  b. handle data to update.
//  c. handle data to delete.
// 3. Return the Update result.
    }
public void start(PushDataAdapterEnvironment env) throws DataAccessException{
    // this method is called when the integration point created,
or when the adapter is reloaded
    //(i.e after changing one of the mapping files
    // and pressing ‘save’).

}
```

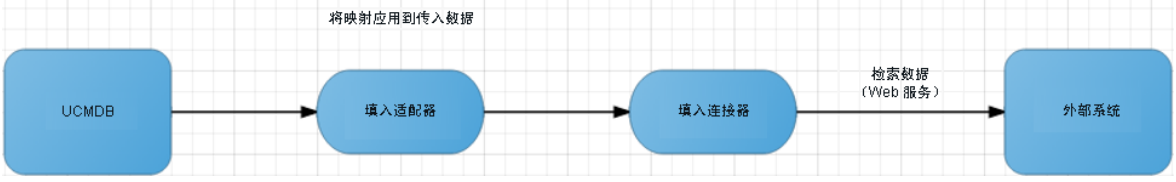
```
public void testConnection(PushDataAdapterEnvironment env) throws DataAccessException {
        // this method is called when pressing the 'test
connection' button in the
        //creation of the integration point.
        // For example if we push data to RDBMS this method
can create a connection
        //to the database and will run a dummy SQL statement.
        // If it fails it writes an error message to the log
and throws an exception.
    }



Map<Object, Object> retrieveUniqueMapping(MappingQuery mappingQuery){
//This method will create the map according to the given mappingQuery.It will be called in the
// mapping stage of the adapter execution, before the 'UpdateResult pushTreeNodes' method.
// This method is called when the 'keys-unique' attribute of the 'dynamic_mapping' tag is
true.
}

Map<Object, Set<Object>> retrieveNonUniqueMapping(MappingQuery mappingQuery){
 // This method is called when the 'keys-unique' attribute of the 'dynamic_mapping' tag is
false.
//  In this case a key can be mapped to several values.
}
}
```

使用常规适配器实现数据填入

本节包含以下主题:

- 填入框架体系结构 246
- 填入中涉及的主要项目 247
- 填入适配器模式 257
- 显式外部 ID 映射 258
- 全局 ID 推回 258

填入框架体系结构



该机制类似于推送适配器框架的机制，即此填入适配器框架的用户必须提供映射文件和连接器实施，并将其一起捆绑到 UCMDB 适配器包中。

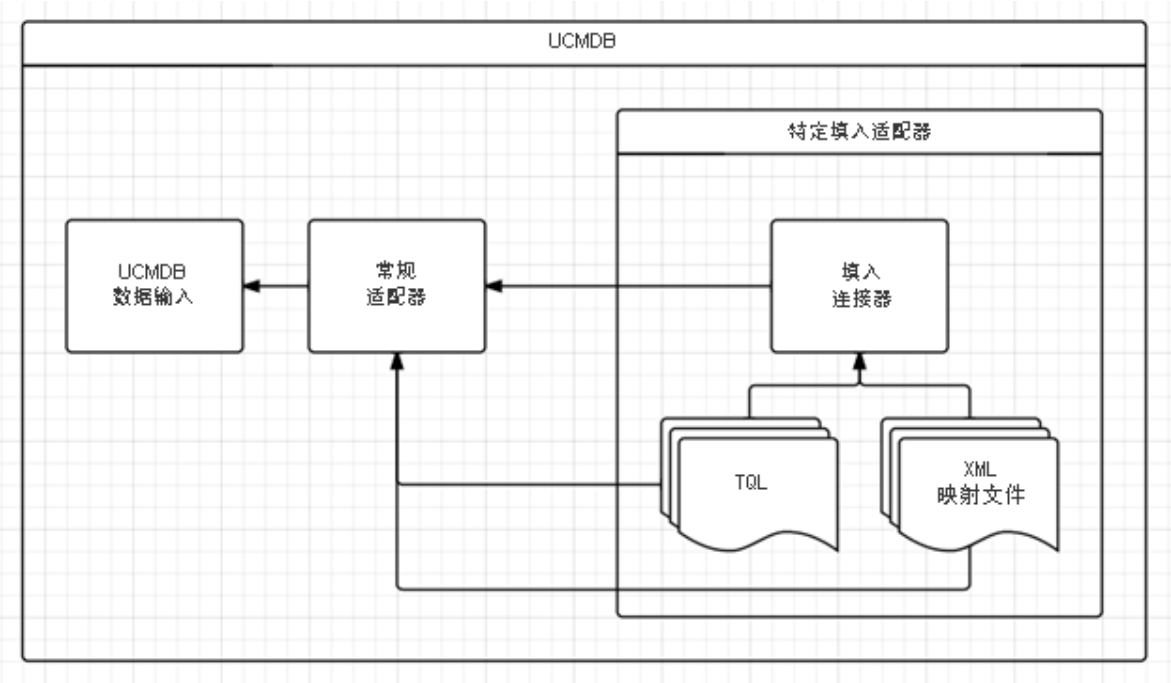
操作流程包括以下步骤

1. UCMDB 用户从 UI 触发填入操作。
2. 命令发送到填入适配器。
3. 填入适配器调用填入连接器并检索成块数据。
4. 填入适配器将定义的映射应用于每个块中的数据，然后将其发送到 UCMDB 服务器。

填入中涉及的主要项目

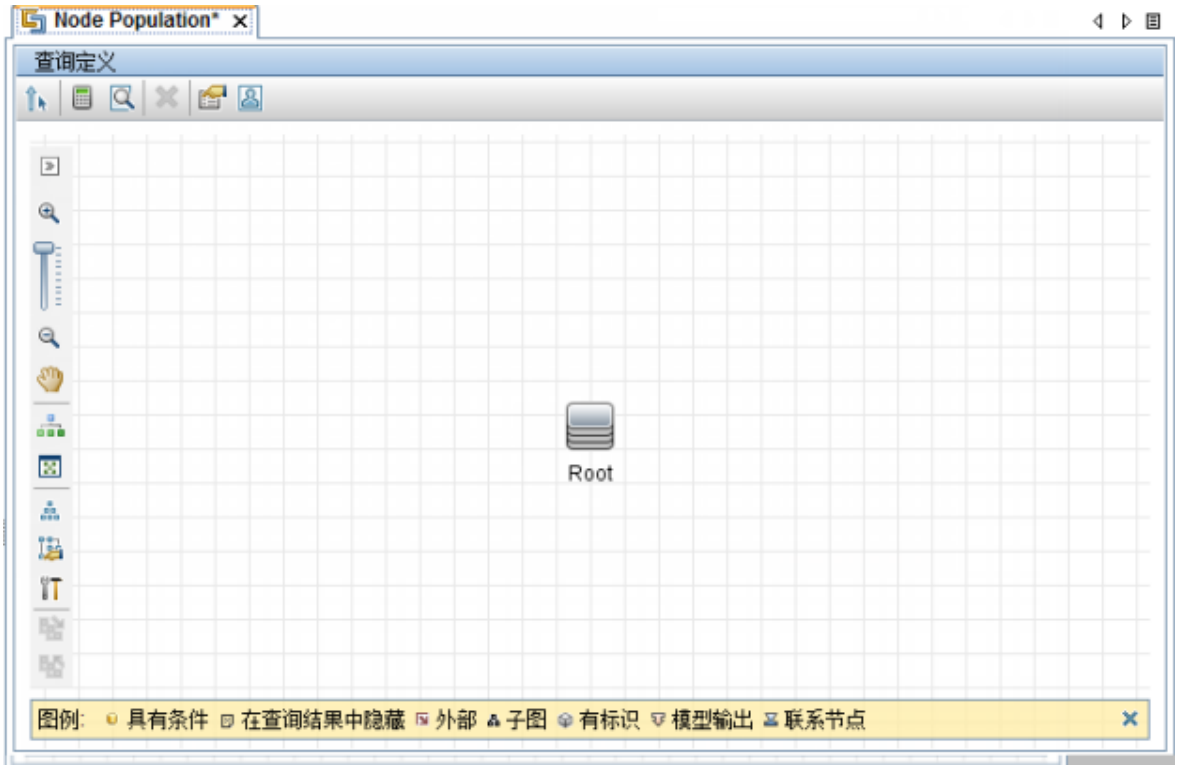
填入中涉及的主要项目有:

- 用于指定将填充到 UCMDB 中的数据的 TQL 查询
- 用于指定如何将连接器返回的数据映射到 UCMDB 的 XML 映射文件
- 必需数据
- 填入连接器负责检索外部系统数据并将其返回到 UCMDB 常规适配器。



填入 TQL 查询

填入 TQL 查询的角色用于指示将填充到 UCMDB 中的数据。例如，下图中的 TQL 用于将 Node 实例包含在 UCMDB 中。



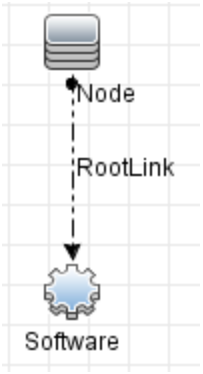
填入连接器负责了解填入 TQL 查询并提供外部系统中的所需数据。

填入映射文件

XML 映射文件与推送操作的用途相同，不同之处在于其方向相反。这些映射文件描述如何将连接器返回的数据映射到 UCMDB 数据。

此处提供的信息与填入映射相关，是增强的常规推送适配器尚未提及的。

以下是 UCMDB 节点和运行软件的映射示例。第一个图像显示节点和软件填入 TQL 查询。第二个图像显示节点和软件填入映射。




```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Nodes And Software Population" root-element-name="PC">
        <!-- need to match case in UCMDB TQL -->
        <target_entity name="RootLink">
            <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
        </target_entity>
        <target_entity name="Node" type="Util.getNodeType(PC)">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC['description']"/>
        </target_entity>
        <target_entity name="Software">
            <target_mapping name="name" datatype="STRING" value="PC.Programs[0]['name']"/>
            <target_mapping name="root_container_name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="product_name" datatype="STRING" value="'vmware_hypervisor'"/>
        </target_entity>
    </source_instance>
</target_entities>
```

此填入作业将以 ResultTreeNode (RTN) PC 的形式获取外部系统中的数据。ResultTreeNode API 由增强的常规推送适配器引入，可在 UCMDB 服务器 **lib** 文件夹的 **push-interfaces.jar** 文件中找到。有关详细信息，请参阅使用常规适配器实现数据推送 (第 238 页)。

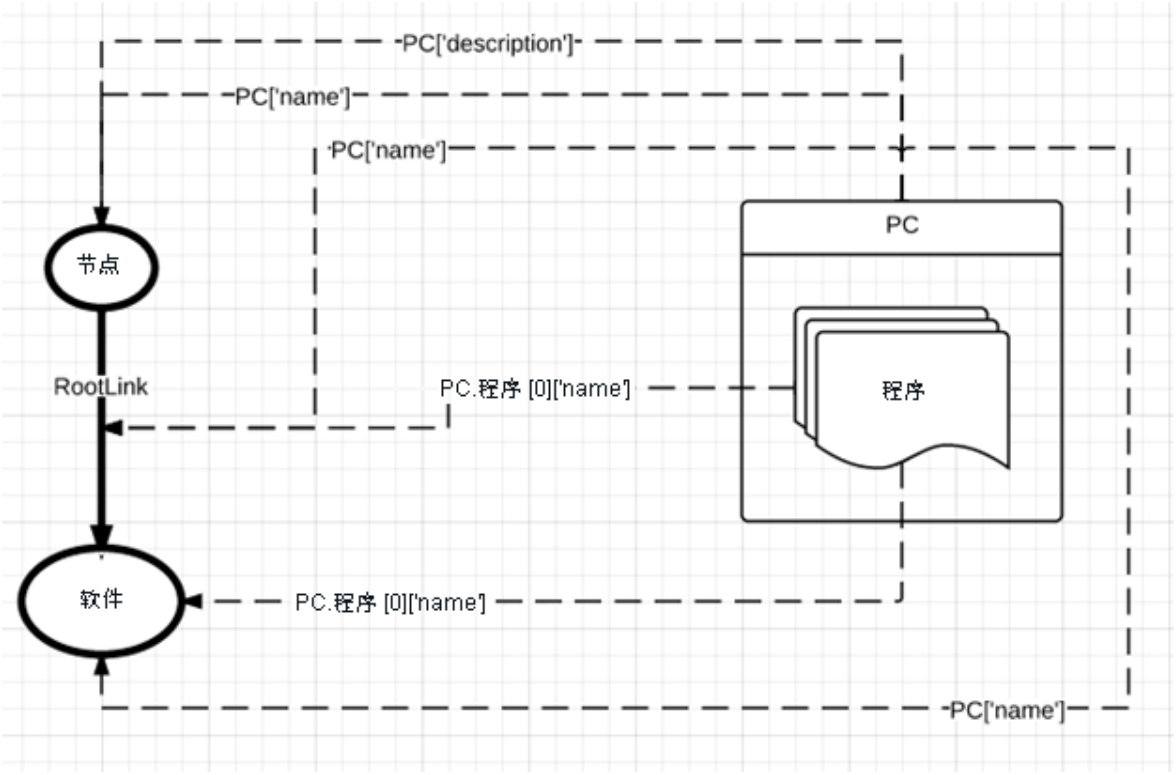
PC RTN 包含属性形式的常规节点信息，以及包含具有相关属性的软件类型实体的嵌入式 Programs 实体。

一个 PC 实例将映射到 UCMDB 中的 3 个实体:

- 节点 CI
- 运行软件 CI
- 组合链接 CI

有关 PC 实例格式的详细信息，请参阅填入请求输出 (第 256 页)。

连接器数据映射到 UCMDB 数据的方式如下图所示:



让我们分析关键行:

```
<!--The query name must match the one selected in the UI-->
<source_instance query-name="Nodes And Software Population" root-element-name="PC">
```

source_instance 定义声明我们会将实体包含到 UCMDB 中，而对这些实体进行分组的 UCMDB 拓扑由节点和软件填入 TQL 查询来定义。此外，连接器返回的将用于创建 UCMDB 数据的数据结构为 **ResultTreeNode**（名称为 **PC**）。

```
<target_entity name="RootLink">
    <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
</target_entity>
```

target_entity 标记声明新的 UCMDB 实体从此处开始，且此实体对应于节点和软件填入 TQL 查询中的 **RootLink** 元素。此指示还包括新实体的 UCMDB CI 类型。

将创建的 **RootLink** 实体具有一个属性 **name**，其值将类似于 **Computer_22 has MySQL Server**。

此示例映射使用手动链接填入。我们建议使用自动方法，如自动链接填入 (第 251 页)中所述。

填入 type 属性

```
<target_entity name="Node" type="Util.getNodeType(PC)">
```

请注意，Node 实体具有 **type** 属性。此 **type** 表示此实体在 UCMDB 中的确切 CI 类型。**type** 属性不是必需的，因为实体的默认创建类型是从其引用的 TQL 元素获取的（在这种情况下为 Node）。但是，如果要返回 UCMDB Node CI 类型的多个实例，且部分实例是 Windows，而其他实例是 Unix，则可以使用 **type** 属性指定确切的 UCMDB 创建类型。因此，在这种情况下，我们在 **Util** 函数脚本文件中创建

getNodeType 函数，该函数接收 PC 树作为输入并返回有效的 UCMDB CI 类型标识符（Unix 则为“unix”，Windows 则为“nt”）。

备注: target_entity type 属性仅在填入流上下文中可用，其值必须是有效的 Groovy 表达式。

我们可以相同的方式描述 Software 实体的创建。

自动链接填入

在填入映射文件的映射示例中，我们可以看到显式映射已填入的链接所需的对象。每个 TQL 链接元素必须存在映射的目标实体，如下所示：

```
<target_entity name="RootLink">
    <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
</target_entity>
```

使用常规适配器自动链接填入机制，不再需要使用映射部分映射 TQL 链接元素（如上所示）。框架将生成具有空属性以及 TQL 中所指定类型的链接 CI 实例。将为填入 TQL 查询中的所有链接执行此操作。

示例映射导致创建组合类型的链接 CI，该 CI 具有 Node 和 Running Software CI 实例这两个链接端（end1 和 end2）。

如果要填入的链接需要以下内容，则应使用手动链接填入：

- 动态链接类型（使用 **type** 属性）
- 链接属性

手动链接填入

通过定义（映射）链接所需的以下三个实体，常规适配器可实现链接填入：

- 链接实体
- 链接的端 1 实体
- 链接的端 2 实体

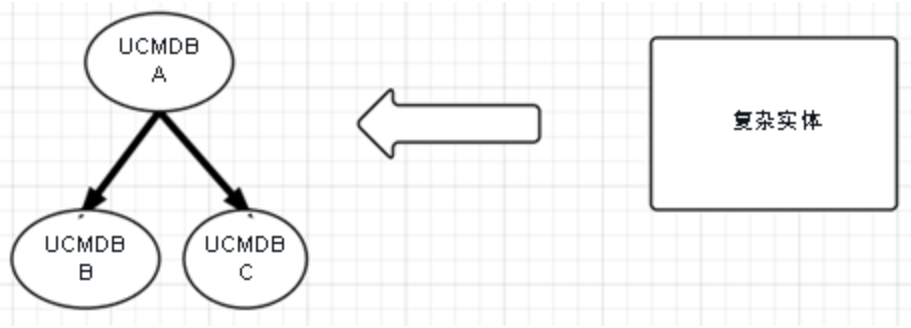
让我们来分析填入映射文件 (第 248 页)中所示的映射示例。在这种情况下，我们将在 UCMDB 中填入三种实体类型：节点、运行软件以及它们之间的组合链接。因为我们想填充链接（类型为“组合”、名称为 **RootLink** 的链接），所以还需要映射两个链接端。因此，从 TQL 查询来看，我们发现需要映射的实体为 Node（端 1）和 Software（端 2）。常规适配器框架了解链接结构的方式是通过查看 TQL 查询中已创建实体的元素名称和定义。因为填入作业还必须提供节点和运行软件的实例，所以所需端的映射已就绪。

链接填入的类型

有两种类型的链接填入情况：

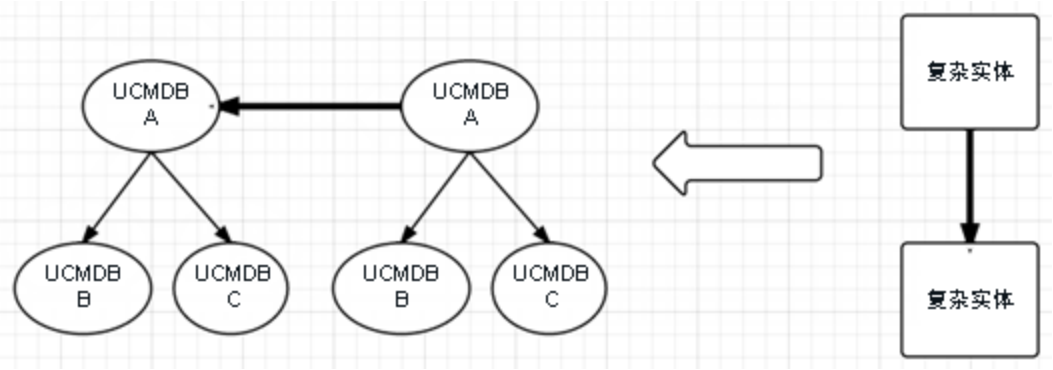
- 将复杂外部实体分解为多个相关的 UCMDB 实体

  在这种情况下，复杂的外部实体（如 PC）将转换为 UCMDB **Node** 和 **Running Software** 类型，而这些类型需要由“组合”链接进行连接。此类型的链接仅存在于 UCMDB 上下文中。



- **复杂外部实体之间的链接**

  在这种情况下，我们需要对两个复杂外部实体（如 PC）之间的链接进行建模。



填入连接器

填入连接器负责检索外部系统数据。这些数据将以既定的 API 格式 (ResultTreeNode) 传递到常规适配器，然后映射到 UCMDB 数据结构并通过数据输入过程插入到 UCMDB 中。

与推送连接器类似，可以 Java 和 Groovy 实施填入连接器，且必须实施下图中所示的填入连接器 Java 接口。

要配置填入连接器，请在适配器配置 XML 文件中添加以下行：

```
<adapter-settings>
    <adapter-setting name="PopulationConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPopulationConnector</adapter-setting>
```

```
public interface PopulationAdapterConnector extends GenericConnector, DynamicMappingConnector {

    /**
     * Executes a population request and provides the entities that will be populated in the format of
     * {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode}
     * Used for both population and federation with the flow type flag which is present in the
     * {@link com.hp.ucmdb.adapters.population.connector.PopulationConnectorInput}
     *
     * @param input population request
     * @return a population response
     * @throws DataAccessException
     */
    PopulationConnectorOutput populate(PopulationConnectorInput input) throws DataAccessException;

    /**
     * Returns the collection of queries the current connector supports for population
     * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
     * <p/>
     * The method also supports return of queries with their folders path: E.g.
     * "Folder/Secondary Folder/Query Name 1"
     * "Folder/Secondary Folder/Query Name 2"
     *
     * @param env the adapter environment
     * @return a collection of the supported queries
     */
    Collection<String> getPopulationQueries(DataAdapterEnvironment env);

    /**
     * Returns the collection of queries the current connector supports for federation
     * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
     * <p/>
     * The method also supports return of queries with their folders path: E.g.
     * "Folder/Secondary Folder/Query Name 1"
     * "Folder/Secondary Folder/Query Name 2"
     *
     * @param env the adapter environment
     * @return a collection of the supported queries
     */
    Collection<String> getFederationQueries(DataAdapterEnvironment env);

    /**
     * Update target id (global id) for each source object.
     *
     * @param idMapping mapping between source id (external id) and target id (string)
     */
    void updateGlobalIDsFromTarget(FCmdbExtenalToTargetIdMappingSet idMapping);

    /**
     * This methods reports population queries resources used by the adapter.<br>
     * This allows editing these resources directly from the Integration Studio.
     *
     * @param input  the requested information
     * @param output the returned resources
     * @see com.hp.ucmdb.federationspi.adapter.resource.PushQueriesResourceLocator
     */
    void locatePopulationQueriesResources(DataAdapterEnvironment env, LocatePopulationQueriesResourcesInput input,
                                          LocatePopulationQueriesResourcesOutput output);

    /**
     * Returns the collection of classes the current adapter supports for query.<br>
     * Notes:<br>
     * 1. This method is independent of the adapter life cycle (i.e. start/shutdown methods).<br>
     * 2. If adapter configuration (xml) defines supported classes, this method doesn't need to be implemented (return null).<br>
     *
     * @param env the Adapter env
     * @return collection of the supported classes
     * @throws DataAccessException in case of an error
     */
    Collection<SupportedClassConfig> getSupportedClasses(DataAdapterEnvironment env);
}
```

第一种方法 populate 是负责检索外部系统中的数据的主要连接器方法。此方法接收填入 TQL 查询作为输入，并以常规 *ResultTreeNode* 格式返回结果。有关详细信息，请参阅使用常规适配器实现数据推送 (第 238 页)。除主要业务数据之外，连接器还返回状态和分块信息。

第二种方法 getSupportedQueries 指示填入连接器支持的 TQL。

第三种和第四种方法引用更高级的用例，推回已填充数据的 ID 并为特定查询查找适配器中的相关填入资源。有关这些 API 的详细信息，请参阅 **push-interfaces.jar** 文件。

填入请求输入

填入请求由描述 UCMDB 填入查询的 QueryDefinition 对象定义。填入连接器负责读取此查询对象并将其转换为外部系统的查询语言。

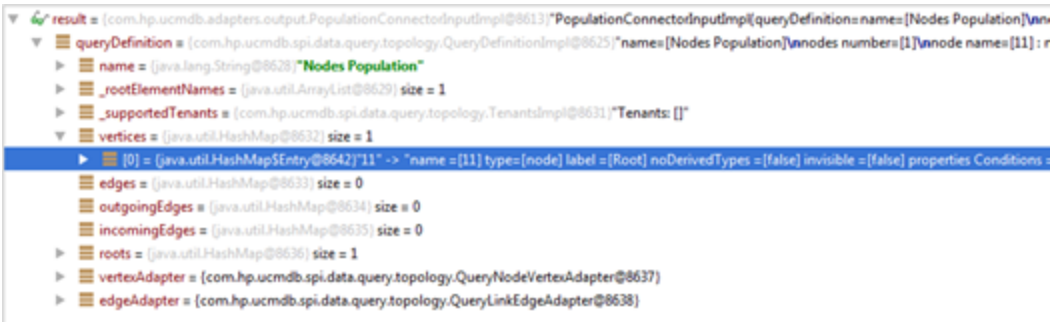
```
 * @since 10.20
 */
public interface PopulationConnectorInput {

    QueryDefinition getQueryDefinition();

    /**
     * Indicates the required {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode} structure
     * that the population result must return.
     *
     * For the target mapping <target_mapping name="lMaxMemory" datatype="LONG" value="Root.VMWare_Host_Resource['vm_memory_limit']"/>
     * the resulting tree node should have the following structure: VMWare_Host_Resource will be a child of Root and vm_memory_limit will
     * be an attribute of VMWare_Host_Resource
     *
     * @return a map containing simplified result trees
     * @see PopulationConnectorOutput#getResultTreeNodes()
     */
    Map<String, ResultTreeNodeStructure> getResultTreeNodeStructure();

    /**
     * Returns the flow type of the operation.
     * Can be POPULATION or FEDERATION
     *
     * @return the flow type
     */
    FederationTopologyAdapterInput.FlowType getFlowType();

    /**
     * Returns the date from the last sync
     *
     * @return the date
     */
    Date getFromDate();
```

除了 QueryDefinition 对象之外，还存在以下对象：

- **getResultTreeNodeStructure** – 表示填入结果必须返回的所需结构。
- **getFlowType** – 用于确定发送到连接器的请求类型是 POPULATION 还是 FEDERATION。

getFromDate – 表示上次同步的日期。如果日期为空，将运行 FULL POPULATION，否则将运行 Diff POPULATION（如果流类型为 FEDERATION，则 getFromDate 方法将始终返回空值）。

下图显示了示例填入请求：



在此示例中，请求包含 Nodes Population 查询。我们可以看到查询仅包含一个 Node 类型的 TQL 元素。

ResultTreeNodeStructure

要实施 **PopulationAdapterConnector**，必须读取 UCMDB 填入 TQL，了解 UCMDB 的需求并使用外部系统实体提供结果。例如，UCMDB 可能需要外部系统中与业务服务实例相关的所有节点，可能等同于计算机的外部系统为 **PC**，它与 **Service** 实体相关。因此，填入连接器必须返回连接到 **Service** 实例的 **PC** 的实例。在这种情况下，映射将如下所示：

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="PC">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService">
            <target_mapping name="name" datatype="STRING" value="PC.Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC.Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

在这种情况下，为了返回与 **Service** 实例相关的 **PC** 实例，我们将返回包含 **Service** 作为子节点的 **PC** RTN。但是，我们本该选择以 **Service** RTN（具有 **PC** 子节点）的格式创建映射（映射无效）：

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="Service">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="Service.PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService">
            <target_mapping name="name" datatype="STRING" value="Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

因此，为帮助开发填入连接器，常规适配器发送的填入请求还将包括映射文件中使用的数据的 RTN 结构。这向实施连接器指示了返回的 RTN 的所需格式。

在第一种情况下，**ResultTreeNodeStructure** 为：

PC
- name

  Service
  - name
  - description

在第二种情况下，**ResultTreeNodeStructure** 为：

Service
- name
- description

  PC
  - name

填入请求输出

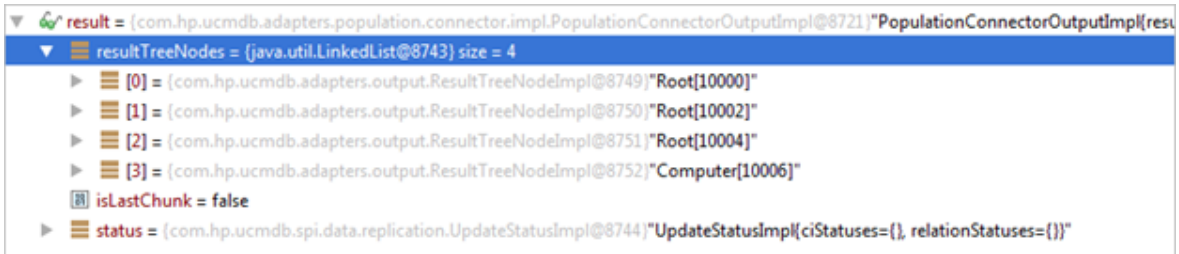
处理填入请求时，填入连接器必须返回 PopulationConnectorOutput。

```
public interface PopulationConnectorOutput {

    /**
     * The result trees representing the external entities in (@link com.hp.ucmdb.adapters.push.output.ResultTreeNode) format.
     *
     * @return a list of result trees or an empty list
     */
    List<ResultTreeNode> getResultTreeNodes();

    /**
     * Adds a result tree to the population output object.
     *
     * @param resultTreeNode the result tree to add
     */
    void addResultTreeNode(ResultTreeNode resultTreeNode);

    /**
     * This method indicates if the result received is the last chunk of data.
     *
     * @return true if this is the last chunk, false otherwise.
     */
    boolean isLastChunk();

    /**
     * Set whether this result object is the last chunk or not.
     */
    void setLastChunk(boolean isLastChunk);

    /**
     * Returns the status information about the population result.
     */
    UpdateStatus getStatus();

    /**
     * Set the population result status.
     */
    void setStatus(UpdateStatus status);

}
```

此输出对象包含:

- **查询的数据（** ResultTreeNode **格式）**
- **状态信息（失败时需要）**
- **块信息**

下图显示了示例填入响应:



在上述响应中，连接器返回了四个数据实例（对应于 UCMDB Node **）、一个空状态（表示成功）以及一个表示这不是最后一块的标记。**

填入适配器模式

UCMDB 适配器框架允许使用两种类型的填入适配器:

- 标准填入适配器
  - 其特征为适配器 XML 文件中没有 **<changes-source/>** 标记
  - 将始终从外部系统获取完整查询数据。在这种情况下，UCMDB 探测器框架负责确定两次连续运行之间的差值。探测器框架通过将给定查询的先前结果与当前结果进行比较并计算差值来实现这一点。通过不比较当前查询结果并将其视为最终结果，实现完整填入。此流程意味着不会按“起始日期”筛选已填充的数据，因为按日期筛选会使数据比较变得毫无意义。
- changes-source 填入适配器
  - 通过在适配器 XML 文件中使用 **<changes-source/>** 标记来配置:

```
<adapterInfo>
    <adapter-capabilities>
        <support-federated-query/>
        <support-replicatioin-data>
            <source>
                <changes-source/>
                <push-back-ids/>
                <re-populate/>
            </source>
            <target/>
        </support-replicatioin-data>
```

  - changes-source 适配器负责计算两次连续运行之间的差值。

使用 changes-source 适配器时删除 CI

如果使用 changes-source 填入适配器，则适配器负责显式删除 CI。通过使用 **is-deleted** 映射文件 XML 属性（接收有效的 Groovy 表达式），可以完成此操作。

例如，在下面显示的映射文件中，填入连接器返回 **Service** 实例。虽然实例仍然有效，但将删除属于这些 **Service** 实例的部分 CI。为表示删除了这些 CI，需要对 **BusinessService** 映射使用 **is-deleted** 属性。

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="Service">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="Service.PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService" is-deleted="Functions.isOlderThanThreeMonths()">
            <target_mapping name="name" datatype="STRING" value="Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

显式外部 ID 映射

可能存在已填充数据 (CI) 需要具有连接器/适配器控制的 ExternalId 的情况。使用以下映射构造执行此操作:

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Node with ID" root-element-name="Computer">
        <!-- need to match case in UCMDB TQL -->
        <target_entity name="Root">
            <!--This is how the RTN External ID is set-->
            <variable name="external_id_obj" datatype="STRING" value="Computer['external_id_obj']"/>
            <!--RTN Attributes-->
            <target_mapping name="name" datatype="STRING" value="Computer.Asset[0]['name']"/>
            <target_mapping name="description" datatype="STRING" value="Computer['name']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

在这种情况下，将为 Root CI 填充在连接器级别创建并放置到 Computer[‘external_id_obj’] 的 ExternalId。还可以使用 Groovy 脚本在映射级别创建 ExternalId。

备注: 显式创建外部 ID 的机制会覆盖 target_entity **type** 属性。因此，使用映射脚本文件或在连接器内创建外部 ID 时，将忽略 **type** 属性，已填充 CI 的最终 UCMDB 类型为 **ExternalId** 对象中设置的 UCMDB 类型。

全局 ID 推回

在某些情况下，UCMDB 中的已填充 CI 还需要与外部系统保持同步。对于这种情况，常规适配器框架允许启用推回 ID。要使用此功能，请对 UCMDB 中填充的所有 CI 执行回调，从而通知填入适配器有关分配给每个 CI 的全局 ID。

为启用此功能，请将以下示例中标记的行添加到适配器配置 XML 文件中:

```
<adapterInfo>
    <adapter-capabilities>

        <support-federated-query>
        <supported-classes>
            <supported-class is-derived="true" all-attributes-supported="true" name="node" is-reconciliation-supported="true"/>
            <supported-class is-derived="true" all-attributes-supported="true" name="business_service" is-reconciliation-supported="true"/>
            <supported-class is-derived="true" all-attributes-supported="true" name="incident" is-reconciliation-supported="true"/>
        </supported-classes>
        </support-federated-query>
        <support-replicatioin-data>
            <source>
                <push-back-ids/>
                <instance-based-data/>
                <population-queries-resources-locator/>
            </source>
            <target>
                <instance-based-data>true</instance-based-data>
                <push-queries-resources-locator/>
            </target>
        </support-replicatioin-data>
        <general-resources-locator/>
    </adapter-capabilities>
```

还必须实施 PopulationAdapterConnector 方法，如下所示：

```
/**
* <b>Description:</b><br>
 * This Interface is used for implementing
 * an adapter to allow the definition of Integration Points.<br>
 * An adapter that implements this Interface will expose
 * ths ability to run Population flows.<br>
 * The only Population flow exposed by this adapter is the Simple Flow using {@link #getDataResult(com.hp.ucmdb.federationspi.adapter.federation.
 * which will return the entire data set of data each time, and compare it to the last data set (handled by framework).<p>
 * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}<br>
 * If possible, it's recommended to implement the {@link PopulationChangesAdapter} instead,
 * allowing the adapter to have better performance and control.<p>
 *
 * @see com.hp.ucmdb.federationspi.data.query.topology.TopologyFactory
 * @see com.hp.ucmdb.federationspi.adapter.ChunkTopologyResultGetter
 * @see PopulationAdapter
 * @since 10.10
 *
*/
public interface PopulationAdapter extends BasicSourceDataAdapter{

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the population framework.<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by the framework
     * @return {@link com.hp.ucmdb.federationspi.adapter.federation.FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

使用常规适配器实现数据联合

通过使用以下方式实现数据联合：

- 联合映射方法 259
- 常规适配器联合 API 260
- 如何设置联合 262

联合映射方法

通过映射 UCMDB 联合框架使用的子 TQL 来处理联合请求，可实现联合映射。一般而言，当常规适配器接收到联合请求时，会发生以下情况：

1. 分析动态联合 TQL 查询并将其与联合连接器所提供的静态联合 TQL 查询列表进行比较。
2. 进行静态 TQL 查询匹配。此 TQL 查询用于标识给定联合请求的所需映射，以及用于创建将提供给联合连接器的 RTN 结构（描述连接器所需的树节点结构的 Java 对象）输入参数。（有关详细信息，请参阅 **push-interfaces.jar** 文件）
3. 将具有 TQL 参数的联合调用发送到连接器。
4. 按照填入映射方式映射连接器发送的传入 RTN 树。请参阅使用常规适配器实现数据填入 (第 246 页)。

联合链接映射

与填入链接映射一样，将自动执行联合链接映射。请参阅自动链接填入 (第 251 页)。

常规适配器联合 API

常规适配器联合 API 与常规适配器填入 API 极其相似。这是因为常规联合适配器 Java 接口与常规填入适配器接口相同。

```
/**
* <b>Description:</b><br>
 * This Interface is used for implementing
 * an adapter to allow the definition of Integration Points.<br>
 * An adapter that implements this Interface will expose
 * ths ability to run Population flows.<br>
 * The only Population flow exposed by this adapter is the Simple Flow using {@link #getDataResult(com.hp.ucmdb.federationspi.adapter.federation.F
 * which will return the entire data set of data each time, and compare it to the last data set (handled by framework).<p>
 * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}<br>
 * If possible, it's recommended to implement the {@link PopulationChangesAdapter} instead,
 * allowing the adapter to have better performance and control.<p>
 *
 * @see com.hp.ucmdb.federationspi.data.query.topology.TopologyFactory
 * @see com.hp.ucmdb.federationspi.adapter.ChunkTopologyResultGetter
 * @see PopulationAdapter
 * @since 10.10
 *
*/
public interface PopulationAdapter extends BasicSourceDataAdapter{

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the population framework.<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by the framework
     * @return {@link com.hp.ucmdb.federationspi.adapter.federation.FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

```
import ...

/**
 * <b>Description:</b><br>
 * This Interface is used for implementing
 * an adapter to allow the definition of Integration Points.<br>
 * An adapter that implements this Interface will expose to run Federation flows.<br>
 * The adapter can report status per CI during the flow with {@link com.hp.ucmdb.federationspi.data.replication.UpdateStatus}.<br>
 * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}
 * </p>
 *
 */
public interface FederationTopologyDataAdapter extends FTqlDataAdapter {

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the federation framework when the user query includes any
     * federated elements(node or link).<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by th
     * @return {@link FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

用于联合的常规适配器连接器接口

联合请求使用的方法与填入请求相同，因此可以使用相同的填入连接器实施。
PopulationConnectorInput Java 类中已添加名为 **FlowType** 的新属性。**FlowType** 属性可以具有两个值：FEDERATION 或 POPULATION。常规适配器通过此属性了解请求类型。

```
/**
 * Holds data needed to process a population request.
 *
 * @author Sergiu Indrie
 * @since 10.20
 */
public interface PopulationConnectorInput {

    QueryDefinition getQueryDefinition();

    /**
     * Indicates the required {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode} structure
     * that the population result must return.
     *
     * For the target mapping <target_mapping name="lMaxMemory" datatype="LONG" value="Root.VMWare_Host_Resource['vm_memory_limit']"/>
     * the resulting tree node should have the following structure: VMWare_Host_Resource will be a child of Root and vm_memory_limit will
     * be an attribute of VMWare_Host_Resource
     *
     * @return a map containing simplified result trees
     * @see PopulationConnectorOutput#getResultTreeNodes()
     */
    Map<String, ResultTreeNodeStructure> getResultTreeNodeStructure();

    /**
     * Returns the flow type of the operation.
     * Can be POPULATION or FEDERATION
     *
     * @return the flow type
     */
    FederationTopologyAdapterInput.FlowType getFlowType();
```

```
import ...

/**
 * Population Connector that will be used by the UCMDB's Generic Population Adapter to provide the external data. The
 * data provided by the connector will be mapped to the UCMDB entities by the adapter configured mapping files.
 * <p/>
 * The Population Connector must be able to process population requests by analyzing the input query and the providing
 * corresponding data from the external system.
 *
 * @author Sergiu Indrie
 * @since 10.20
 */
public interface PopulationAdapterConnector extends GenericConnector, DynamicMappingConnector {

    /**
     * Executes a population request and provides the entities that will be populated in the format of
     * {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode}
     * Used for both population and federation with the flow type flag which is present in the
     * {@link com.hp.ucmdb.adapters.population.connector.PopulationConnectorInput}
     *
     * @param input population request
     * @return a population response
     * @throws DataAccessException
     */
    PopulationConnectorOutput populate(PopulationConnectorInput input) throws DataAccessException;
```

支持的联合查询

联合查询和填入查询位于不同的文件夹中。PopulationAdapterConnector Java 接口提供以下两种方法来指示支持的填入查询和联合查询:

- **getPopulationQueries** – 返回当前连接器支持填入的查询集合。
- **getFederationQueries** – 返回当前连接器支持联合的查询集合。

```
/**
 * Returns the collection of queries the current connector supports for population
 * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
 * <p/>
 * The method also supports return of queries with their folders path: E.g.
 * "Folder/Secondary Folder/Query Name 1"
 * "Folder/Secondary Folder/Query Name 2"
 *
 * @param env the adapter environment
 * @return a collection of the supported queries
 */
Collection<String> getPopulationQueries(DataAdapterEnvironment env);

/**
 * Returns the collection of queries the current connector supports for federation
 * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
 * <p/>
 * The method also supports return of queries with their folders path: E.g.
 * "Folder/Secondary Folder/Query Name 1"
 * "Folder/Secondary Folder/Query Name 2"
 *
 * @param env the adapter environment
 * @return a collection of the supported queries
 */
Collection<String> getFederationQueries(DataAdapterEnvironment env);
```

如何设置联合

本节包含:

- 配置适配器设置 263
- 从日志文件设置联合查询 263
- 联合设置示例 267

配置适配器设置

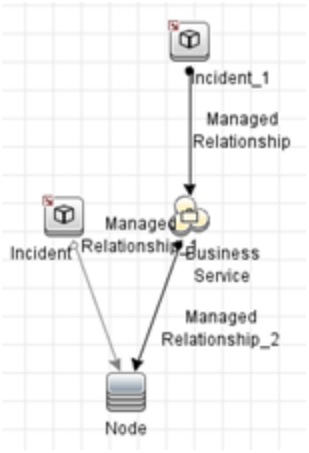
对于给定 TQL 查询，常规适配器需要在 **<supported-classes>** 标记中声明该 TQL 查询的所有节点。例如，如果 TQL 查询的形式为“事件链接到节点”，则必须在适配器包 ZIP 文件的 **discoveryPatterns** 文件夹中的适配器设置 XML 文件中将节点和事件声明为受支持的类。


```
<supported-classes>
    <supported-class is-derived="true" all-attributes-supported="true" name="node" is-reconciliation-supported="true"/>
    <supported-class is-derived="true" all-attributes-supported="true" name="incident" is-reconciliation-supported="true"/>
</supported-classes>
```

从日志文件设置联合查询

使用联合框架之前，必须满足某些先决条件。联合框架通过不同的 TQL 查询向适配器发出若干请求，以便检索所需数据。对于联合框架发送的每个 TQL 查询，必须以与填入流类似的方式在适配器中创建动态 TQL 查询。

联合与填入的区别在于联合 TQL 查询由框架动态发送，所以不能提前识别它们。以下面的 TQL 查询为例：



框架会将以下查询发送到适配器：

- 仅包含事件的查询
- 通过连接类型关系将事件链接到节点的查询

- 通过成员资格类型关系将事件链接到节点的查询
- 通过连接类型关系将事件链接到业务服务的查询
- 通过成员资格类型关系将事件链接到业务服务的查询

备注: 联合引擎发送到适配器以及需要保存的所有查询具有名称 **User mapping union FTQL**。

处理 **User mapping union FTQL** 查询的结果后，将进行其他调用以检索对象的属性。这些调用包含名为 **objects layout** 的查询。联合引擎将尝试获取 CI 的所有属性，但连接器不需要全部提供；仅返回映射文件所需的属性就已足够。

发送具有不同关系的同一查询的原因是：在 TQL 查询中，节点与事件/业务服务与事件之间存在 **managed_relationship** 类型链接，但尝试一起链接这些 CI 类型时唯一的有效链接为连接和成员资格。

```
<tql:link from="incident_12" to="node_10050" class="connection" name="connection_1" id="1"/>
<tql:link from="incident_12" to="node_1000050" class="connection" name="connection_2" id="2"/>
<tql:link from="datacenter_20050" to="node_10050" class="composition" name="composition_30050" id="30050"/>
```

```
<tql:link from="incident_12" to="node_10050" class="membership" name="membership_1" id="1"/>
<tql:link from="incident_12" to="node_1000050" class="membership" name="membership_2" id="2"/>
<tql:link from="datacenter_20050" to="node_10050" class="composition" name="composition_30050" id="30050"/>
```

通过使用此方法，我们只需要定义一个具有常规 **managed_relationship** 类型链接的静态 TQL，而非定义两个除链接类型不同外几乎相同的 TQL。

```
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >> Received federation call with the following query:
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >>
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tql:query name="User mapping union FTQL" is-live="true" priority="low" xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyRuleD
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="membership_1"/>
                    <tql:link-ref name="membership_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>MyBusServ</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201334714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:link from="incident_16" to="business_service_10050" class="membership" name="membership_1" id="1"/>
    <tql:link from="incident_16" to="business_service_1000050" class="membership" name="membership_2" id="2"/>
</tql:query>
```

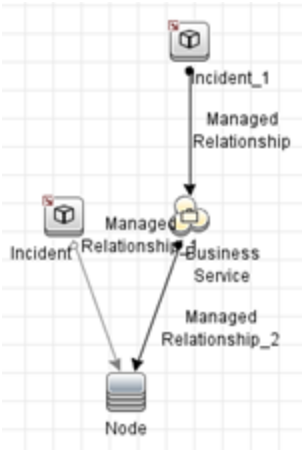
这些静态 TQL 查询必须由适配器提供。要帮助开发适配器，请在 **fcmdb.push.mapping.log** 文件（启用了 TRACE 日志级别）中编写由联合框架发送的 TQL 查询。要简化开发工作，请使用设置 <adapter-setting name="dev.mode">true</adapter-setting>。如果在运行联合查询后将此设置设置为 True，则框架将为当前不匹配的 TQL 查询自动创建空的联合结果。

fcmdb.push.mapping.log 中的联合查询示例：

```
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >> Received federation call with the following query:
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >>
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tql:query name="User mapping union FTQL" is-live="true" priority="low" xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyRuleD
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="membership_1"/>
                    <tql:link-ref name="membership_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>MyBusServ</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201394714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:link from="incident_16" to="business_service_10050" class="membership" name="membership_1" id="1"/>
    <tql:link from="incident_16" to="business_service_1000050" class="membership" name="membership_2" id="2"/>
</tql:query>
```

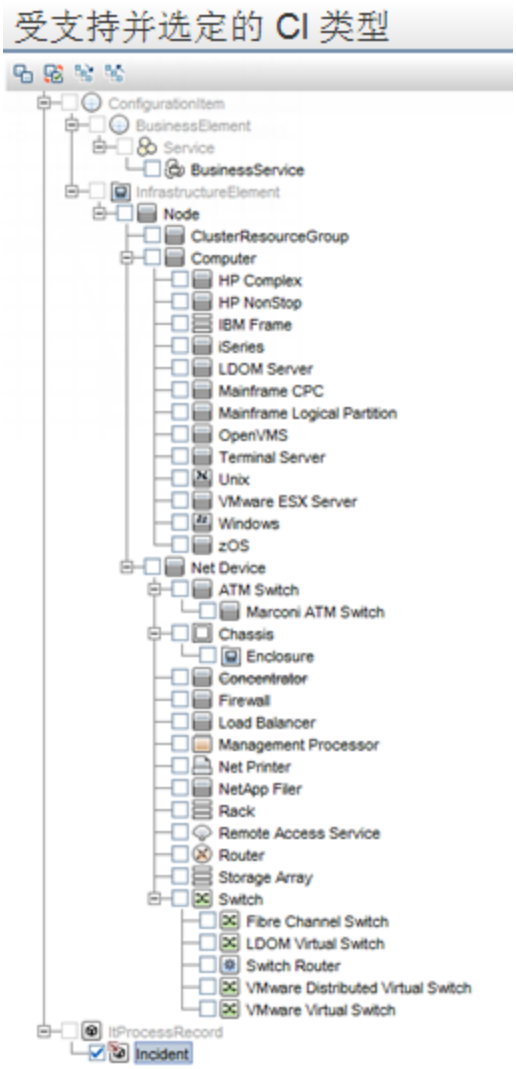
联合设置示例

示例将使用以下联合 TQL：



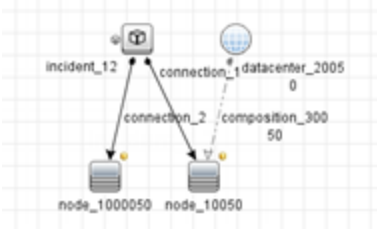
对于此 TQL 查询，适配器必须在适配器包 ZIP 文件的 discoveryPatterns 文件夹中的适配器设置 XML 文件中声明受支持的类。受支持的类为 node、incident 和 business_service。

在“集成工作室”中，必须在“联合”选项卡上选择事件，如下所示：

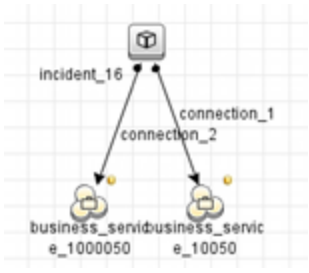


对于此 TQL 查询，必须在适配器中包括以下三个 TQL 查询：

事件



事件链接到节点



事件链接到 business_ service

有关如何获取静态 TQL 的信息，请参阅从日志文件设置联合查询 (第 263 页)。虽然这些 TQL 查询具有依赖于 UCMDB 中的数据的条件，但这不会影响 TQL 查询的结果或执行映射的方式。

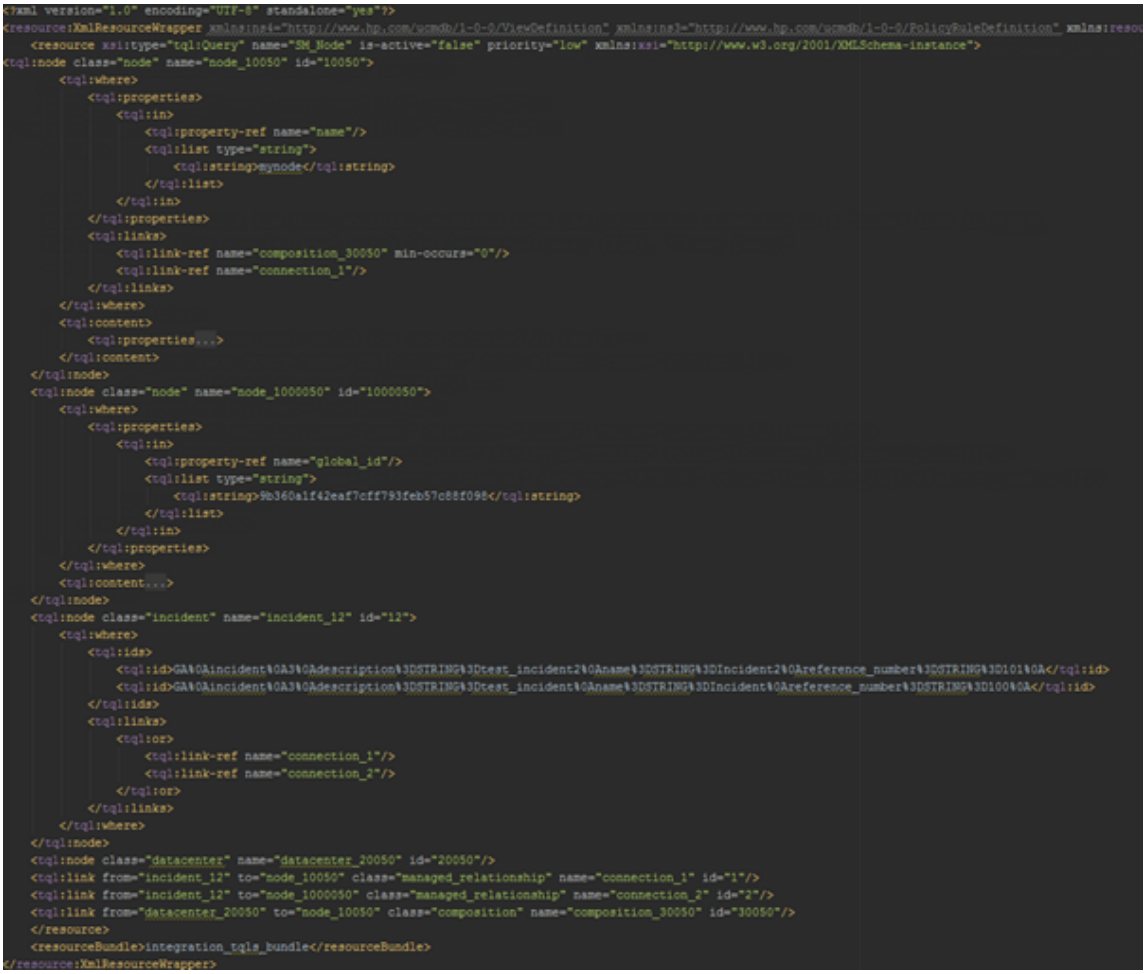
对于每个这样的 TQL 查询，适配器中都需要映射文件：

- **事件**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resource:XmlResourceWrapper xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyRuleDefiniti
    <resource xsi:type="tql:Query" name="SM_Incident" is-active="false" priority="low" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<tql:node class="incident" name="incident_12" id="12"/>
    </resource>
    <resourceBundle>integration_tqls_bundle</resourceBundle>
</resource:XmlResourceWrapper>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>

    <!--
    This mapping converts the Root entities received from the population connector to a "node" item in UCMDB,
    The output of this mapping must match the indicated query (TQL), "Dummy Query"
    -->

    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_Incident" root-element-name="incident_12">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="incident_12">
                <target_mapping name="name" datatype="STRING" value="incident_12['name']"/>
                <target_mapping name="description" datatype="STRING" value="incident_12['description']"/>
                <target_mapping name="reference_number" datatype="STRING" value="incident_12['reference_number']"/>
            </target_entity>
        </source_instance>
    </target_entities>

</integration>
```

- **事件链接到节点**



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resource:XmlResourceWrapper xmlns:ns1="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns2="http://www.hp.com/ucmdb/1-0-0/PolicyRuleDefinition" xmlns:resou
    <resource xsi:type="tql:Query" name="SM_Node" is-active="false" priority="low" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<tql:node class="node" name="node_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>mynode</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
            <tql:links>
                <tql:link-ref name="composition_30050" min-occurs="0"/>
                <tql:link-ref name="connection_1"/>
            </tql:links>
        </tql:where>
        <tql:content>
            <tql:properties...>
        </tql:content>
    </tql:node>
    <tql:node class="node" name="node_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>9b360a1f42eaf7cff793feb57c88f098</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content...>
    </tql:node>
    <tql:node class="incident" name="incident_12" id="12">
        <tql:where>
            <tql:ids>
                <tql:id>GA%0Aincident%0A3%0Adescription%3DSTRING%3Dtest_incident2%0Aname%3DSTRING%3DIncident2%0Areference_number%3DSTRING%3D101%0A</tql:id>
                <tql:id>GA%0Aincident%0A3%0Adescription%3DSTRING%3Dtest_incident%0Aname%3DSTRING%3DIncident%0Areference_number%3DSTRING%3D100%0A</tql:id>
            </tql:ids>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="connection_1"/>
                    <tql:link-ref name="connection_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="datacenter" name="datacenter_20050" id="20050"/>
    <tql:link from="incident_12" to="node_10050" class="managed_relationship" name="connection_1" id="1"/>
    <tql:link from="incident_12" to="node_1000050" class="managed_relationship" name="connection_2" id="2"/>
    <tql:link from="datacenter_20050" to="node_10050" class="composition" name="composition_30050" id="30050"/>
    </resource>
    <resourceBundle>integration_tqls_bundle</resourceBundle>
</resource:XmlResourceWrapper>
```

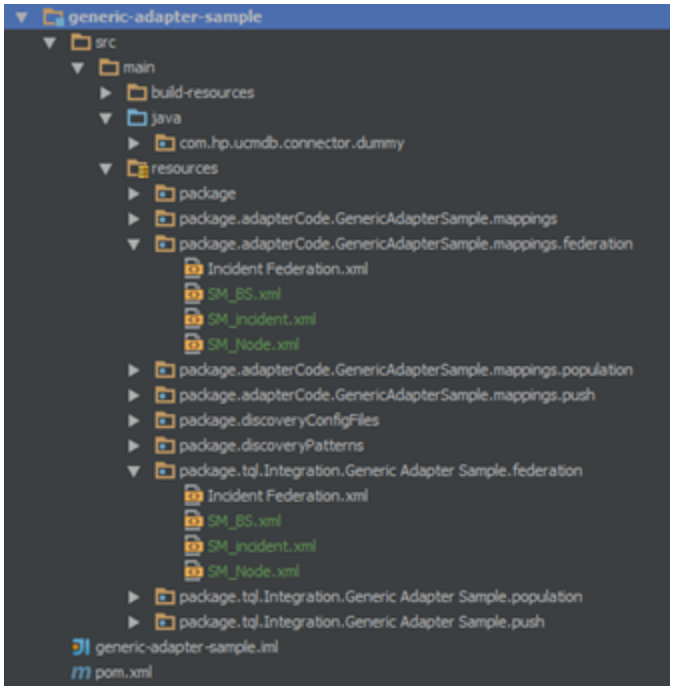
```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>
    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_Node" root-element-name="Computer">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="node_1000050">
                <target_mapping name="name" datatype="STRING" value="Computer['name']"/>
            </target_entity>


            <for-each-source-entity count-index="i" source-entities="Computer.incident_12" var-name="currIP">
                <target_entity name="incident_12">
                    <target_mapping name="name" datatype="STRING" value="currIP['name']"/>
                    <target_mapping name="description" datatype="STRING" value="currIP['description']"/>
                    <target_mapping name="reference_number" datatype="STRING" value="currIP['reference_number']"/>
                </target_entity>
            </for-each-source-entity>


        </source_instance>
    </target_entities>
</integration>
```

- **事件链接到 business_service**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resource:XmlResourceWrapper xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyRu
    <resource xsi:type="tql:Query" name="SM_BS" is-active="false" priority="low" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="connection_1"/>
                    <tql:link-ref name="connection_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>MyBusServ</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201334714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
        <tql:link from="incident_16" to="business_service_10050" class="managed_relationship" name="connection_1" id="1"/>
        <tql:link from="incident_16" to="business_service_1000050" class="managed_relationship" name="connection_2" id="2"/>
    </resource>
    <resourceBundle>integration_tqls_bundle</resourceBundle>
</resource:XmlResourceWrapper>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>
    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_BS" root-element-name="BS">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="business_service_10050">
                <target_mapping name="name" datatype="STRING" value="BS['name']"/>
            </target_entity>
            <for-each-source-entity count-index="i" source-entities="BS.incident_16" var-name="currIP">
                <target_entity name="incident_16">
                    <target_mapping name="name" datatype="STRING" value="currIP['name']"/>
                    <target_mapping name="description" datatype="STRING" value="currIP['description']"/>
                    <target_mapping name="reference_number" datatype="STRING" value="currIP['reference_number']"/>
                </target_entity>
            </for-each-source-entity>
        </source_instance>
    </target_entities>
</integration>
```

这些 TQL 查询和映射文件必须位于适配器中，如下所示：



调节

使用常规适配器框架填充或联合数据时，CI 必须始终具有所需调节数据，以便 UCMDB 接受这些数据。填充需要容器 CI 类型的 CI 类型（如运行软件）时，始终确保填充所需的容器字段（例如，**root_container_name** 和 **product_name**）以及容器 CI（例如 **Node**）。要填充依赖于根容器的 CI，则必须在同一步骤中创建该 CI、其根容器以及它们之间的链接（使用显式链接填入或自动完成这两个 CI 之间的链接填入）。

此外，映射已填充/联合的 CI 时，请考虑映射 **global_id** 属性，因为这样将极大地帮助 UCMDB 调节引擎保证精确的 CI 调节。

常规适配器 API

常规适配器框架提供的 API 为：

<UCMDB 服务器>\lib\push-interfaces.jar
<UCMDB 服务器>\lib\integrationFramework\GenericAdapter\generic-adapter-api-factory.jar

常规适配器实例的开发可能还需要联合 API：

<UCMDB 服务器>\lib\federation-api.jar

资源定位器 API

在编辑常规适配器作业时可使用资源定位器 API。实施常规和填入资源定位器 API 可帮助查找与选定作业的 TQL 查询相关的适配器资源。

以下图像显示了 GenericConnector Java 接口中的常规资源定位器 API:

```
/**
 * This methods reports general resources used by the adapter.<br>
 * This allows editing these resources directly from the Integration Studio.
 *
 *
 * @param env the Adapter's environment
 * @param input  the requested information
 * @param output the returned resources
 * @see com.hp.ucmdb.federationspi.adapter.resource.GeneralResourcesLocator
 */
void locateGeneralResources(DataAdapterEnvironment env, LocateGeneralResourcesInput input, LocateGeneralResourcesOutput output);
```

以下图像显示了 PopulationAdapterConnector Java 接口中的填入查询资源定位器 API:

```
/**
 * This methods reports population queries resources used by the adapter.<br>
 * This allows editing these resources directly from the Integration Studio.
 *
 * @param input  the requested information
 * @param output the returned resources
 * @see com.hp.ucmdb.federationspi.adapter.resource.PushQueriesResourceLocator
 */
void locatePopulationQueriesResources(DataAdapterEnvironment env, LocatePopulationQueriesResourcesInput input,
                                      LocatePopulationQueriesResourcesOutput output);
```

要查看作业的 TQL 查询的相关资源，请执行以下操作：

1. 在“集成工作室”中，选择集成点。
2. 在“集成作业”窗格中，选择作业并单击“编辑查询资源”  。

创建常规适配器包

常规适配器包类似于增强的常规推送适配器包。要创建初始 skeleton ZIP 存档，建议复制现有常规适配器包并根据需要对其进行自定义。有关适配器包的详细信息，请参阅使用常规适配器实现数据推送 (第 238 页)。

现有增强的常规推送适配器包和常规适配器包之间的区别如下：

- 适配器 XML 不同
  - 适配器类从 **PushAdapter** 更改为 **GenericAdapter**:

    ```
    <className>com.hp.ucmdb.adapters.push.PushAdapter</className>
    ```

    ```
    <className>com.hp.ucmdb.adapters.GenericAdapter</className>
    ```

  - 适配器功能包括填入

```
<support-replicatioin-data>
    <source>
        <push-back-ids/>
        <instance-based-data/>
        <population-queries-resources-locator/>
    </source>
```

- 以及由适配器设置执行的填入连接器的定义:

```
<adapter-setting name="PopulationConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPopulationConnector</adapter-setting>
```

常规适配器（使用填入功能）还需要推送连接器类的定义:

```
<adapter-setting name="PushConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPushConnector</adapter-setting>
```

- 映射文件文件夹

  与增强的常规推送适配器（要求映射文件位于 **<适配器包 zip>/adapterCode/<*适配器名称*>/mappings** 文件夹中）相反，常规适配器要求其映射放在三个不同的文件夹（推送、填入和联合各一个）中。所需文件夹为:

  <适配器包 zip>/adapterCode/<*适配器名称*>/mappings/push
  <适配器包 zip>/adapterCode/<*适配器名称*>/mappings/population
  <适配器包 zip>/adapterCode/<*适配器名称*>/mappings/federation

  其中 **<适配器包 zip>** 是指将为常规适配器包创建的 zip 存档。

备注: 虽然常规适配器支持所有三种类型的数据同步（推送、填入和联合），但特定的常规适配器还是可以选择仅提供部分类型。

从现有适配器创建新适配器时需记住的要点

- **TestAdapter\discoveryPatterns\TestAdapter.xml**
- 修改 **TestAdapter.xml** 文件:
  - <pattern xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="TestAdapter" xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="…" schemaVersion="9.0" displayName="…">
  - <adapter-id>TestAdapter</adapter-id>
- 包含新适配器的 ZIP 文件应与适配器本身具有相同的名称 TestAdapter。

生成适配器包

确保适配器包包含以下文件夹:

- **adapterCode**。在此文件夹下，创建名为 **PushExampleAdapter** 的文件夹，它将包含我们从 PushExampleAdapter.java 中创建的 .jar 文件。此外，它还包含名为 **mappings** 的文件夹，您可以将先前创建的映射文件 **computerIPMapping.xml** 放在此文件夹中。它还应该包含另一个具有 **PushFunctions.groovy** 文件、且名为 **scripts** 的文件夹。
- **discoveryConfigFiles**。包含配置文件，例如，在使用 UpdateResult 的情况下报告错误时使用的错误代码。在此示例中，该文件夹为空。

- **discoveryPatterns**。包含 **push_example_adapter.xml**。
- **tql**。包含为此示例创建的 TQL 查询。此文件夹是可选的，但部署包时会自动创建该 TQL 查询。

启用/禁用适配器级别的属性和链接验证

通过添加以下设置，可以为常规适配器启用或禁用适配器级别的属性和链接验证：

```
<adapter-settings>
    <adapter-setting name="enable.attributes.links.validation">true</adapter-setting>
</adapter-settings>
```

要启用属性和链接的适配器级别验证，请将适配器设置 **enable.attributes.links.validation** 设置为 **true**。

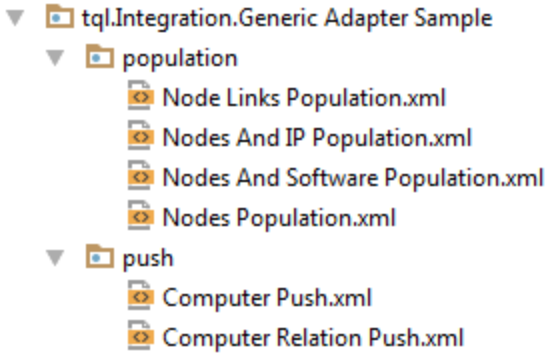
要禁用属性和链接的适配器级别验证，请将适配器设置 **enable.attributes.links.validation** 设置为 **false**。

注意：如果设置未显示，则为默认值 **true**，这意味着将默认启用属性和链接验证。

填入 TQL 查询

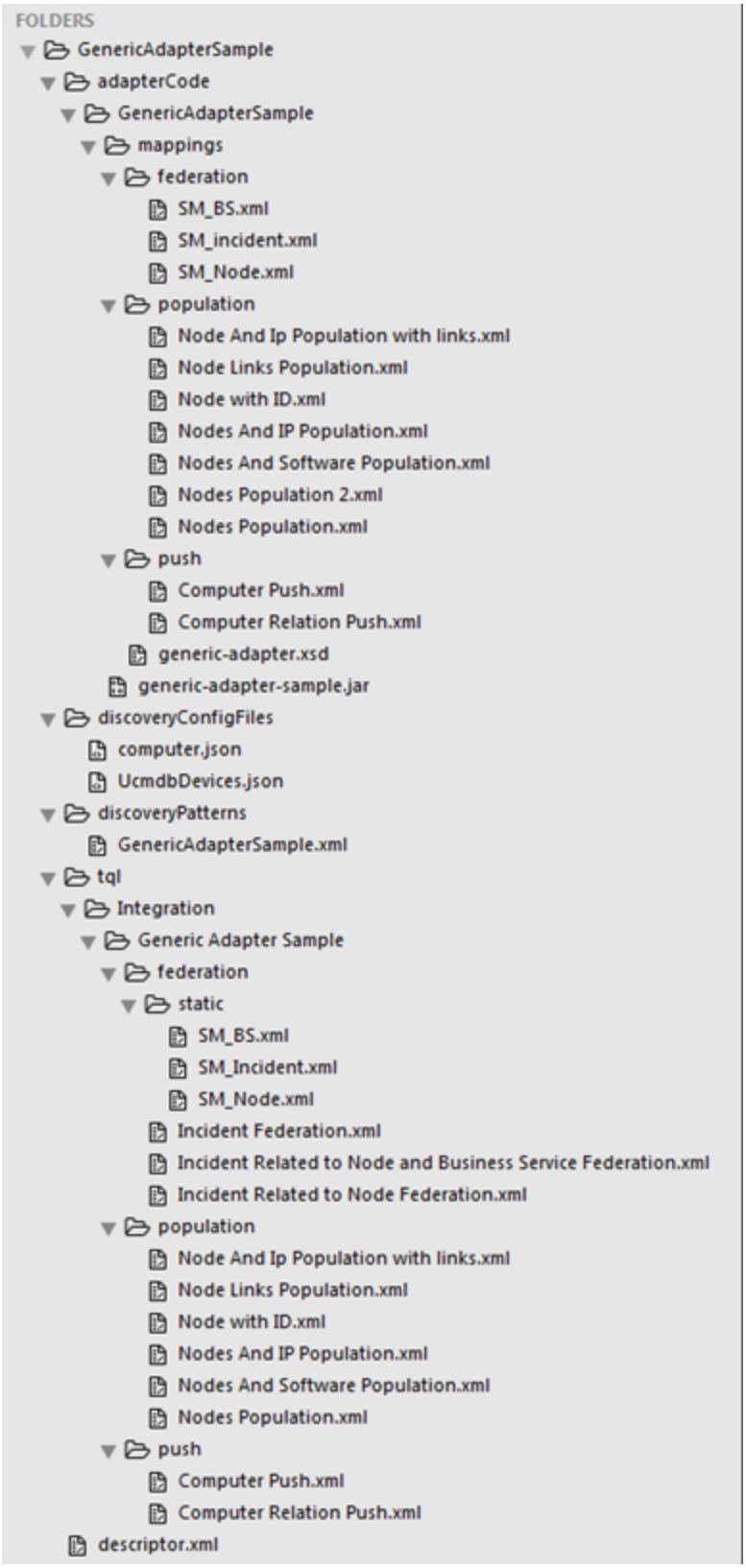
用于填入作业的 TQL 查询必须包含在常规适配器的 ZIP 存档中并使用 UCMDB 中的适配器进行部署。在填入流期间发出填入请求时，指示的 TQL 查询必须存在于 UCMDB 中。

这些 TQL 查询必须包含在 **<zip>/tql/<文件夹 1>/../<文件夹 n>** 中。下面是文件夹结构的示例：



虽然填入 TQL 查询位于上述文件夹中，但填入连接器还必须使用 Java 接口中的相应方法确认支持的填入 TQL 查询。有关详细信息，请参阅填入连接器 (第 252 页)。

示例包


FOLDERS
GenericAdapterSample
adapterCode
GenericAdapterSample
mappings
federation
SM_BS.xml
SM_incident.xml
SM_Node.xml
population
Node And Ip Population with links.xml
Node Links Population.xml
Node with ID.xml
Nodes And IP Population.xml
Nodes And Software Population.xml
Nodes Population 2.xml
Nodes Population.xml
push
Computer Push.xml
Computer Relation Push.xml
generic-adapter.xsd
generic-adapter-sample.jar
discoveryConfigFiles
computer.json
UcmdbDevices.json
discoveryPatterns
GenericAdapterSample.xml
tql
Integration
Generic Adapter Sample
federation
static
SM_BS.xml
SM_Incident.xml
SM_Node.xml
Incident Federation.xml
Incident Related to Node and Business Service Federation.xml
Incident Related to Node Federation.xml
population
Node And Ip Population with links.xml
Node Links Population.xml
Node with ID.xml
Nodes And IP Population.xml
Nodes And Software Population.xml
Nodes Population.xml
push
Computer Push.xml
Computer Relation Push.xml
descriptor.xml

推送和填入映射之间的差异

虽然推送和填入映射文件具有相同的基础 XML 架构，但是文件的解释稍有不同。有关详细信息，请参阅常规适配器 XML 架构参考 (第 279 页)。

在以下推送映射示例中，解释为：获取“Computer Push”TQL 查询（在 UCMDB 中运行）的结果并呈现在 Root 树结构中，创建稍后将发送到 AM 的 amComputer 实体。

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Computer Push" root-element-name="Root">
        <target_entity name="amComputer">
            <target_mapping name="TcpIpHostName" datatype="STRING" value="Root['name']"/>
            <target_mapping name="ComputerDesc" datatype="STRING" value="Root['os_description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

在以下填入映射示例中，解释为：获取“Nodes Population”TQL 查询（在外部系统中运行）的结果并呈现在 PC 树结构中，然后创建稍后将添加到 UCMDB 中的 UCMDB Root 实体（类型为 Node；使用 TQL 查询表示）。

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Nodes Population" root-element-name="PC">
        <!-- need to match case in UCMDB class model-->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

常规适配器日志文件

如需疑难解答和调试，请使用以下文件：

- 调整以下文件中的日志记录级别（将 *loglevel* 变量设置为 TRACE 可获取最详细的结果）：
  - **<UCMDB_DataFlowProbe>\conf\log\fcmdb.push.properties**

    <UCMDB_DataFlowProbe> 是 UCMDB Data Flow Probe 安装目录。
  - **<UCMDB 服务器>\conf\log\reconciliation.properties**

    <UCMDB 服务器> 是 UCMDB 服务器安装目录。
- 分析以下常规适配器日志文件：
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.all.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.configuration.log**

- **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.connector.all.log**
- **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.connector.configuration.log**
- **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.mapping.log**
- **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.all.log**

- 分析以下常规日志文件:
  - **<UCMDB_DataFlowProbe>\runtime\log\probe-error.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\WrapperProbeGw.log**
  - **<UCMDB 服务器>\runtime\log\error.log**
  - **<UCMDB 服务器>\runtime\log\cmdb.reconciliation.log**

使用常规适配器框架的适配器

作为开发自定义常规适配器的借鉴，请参考下面 UCMDB 附带的适配器作为实施准则，该准则应该可以加速适配器开发:

- Asset Manager 适配器
- Service Manager 适配器

常规适配器 XML 架构参考

可在 **schema** 目录下的 **cmdb.jar** 文件中找到常规适配器 XML 架构。在外部编辑器中编写常规适配器映射文件时，应引用架构文件。XSD 文件的完整路径为:

<UCMDB 服务器安装目录>/lib/cmdb.jar/schema/generic-adapter.xsd

第 II 部分: 使用 API

第 9 章: API 简介

本章包括:

- API 概述 281

API 概述

HP Universal CMDB 中附带有如下 API:

- **UCMDB Java API。**描述第三方或自定义工具如何使用 Java API 来提取数据和执行计算，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 282 页)。
- **UCMDB Web 服务 API。**支持将配置项定义和拓扑关系写入 UCMDB，并支持使用 TQL 和特别查询来查询信息。有关详细信息，请参阅HP Universal CMDB Web 服务 API (第 289 页)。
- **数据流管理 Java API。**支持对数据流管理的探测器、作业、触发器和凭据进行管理。有关详细信息，请参阅数据流管理 Java API (第 319 页)。
- **数据流管理 Web 服务 API。**支持对数据流管理的探测器、作业、触发器和凭据进行管理。有关详细信息，请参阅数据流管理 Web 服务 API (第 320 页)。

备注: 要充分利用 API 文档，建议访问联机文档。在 PDF 版本中，没有指向以 html 格式生成的 API 文档的链接。

第 10 章: HP Universal CMDB API

本章包括:

- 约定 282
- 使用 HP Universal CMDB API 282
- 应用程序的常规结构 283
- 将 API Jar 文件放入类路径中 285
- 创建集成用户 285
- UCMDB API 用例 287
- 示例 288

约定

本章使用以下约定:

- UCMDB 是指通用配置管理数据库本身。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。

有关可用 API 的完整文档，请参考HP UCMDB API Reference。

这些文件位于以下文件夹中:

\\<UCMDB 根目录>\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\UCMDB_JavaAPI\index.html

使用 HP Universal CMDB API

备注: 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
- 从 UCMDB 历史记录检索信息
- 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“拓扑查询语言”。

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 283 页)
- 权限 (第 283 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UCMDB API 用例 (第 287 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。

此外，用户必须具有“访问 SDK”常规操作的权限才能登录。

警告: API 客户端还适用于常规用户，只要这些用户具有 API 身份验证权限即可。但是，不建议使用此选项。

有关详细信息，请参阅创建集成用户 (第 285 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 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
getAuthorizationModelService	执行授权操作（创建用户和用户组、将角色分配给用户和组，等等）。

服务方法	用途
getClassModelService	获取有关 CI 和关系类型的信息
getConfigurationService	服务器配置的基础结构设置管理
getDataStoreMgmtService	查询数据存储信息，包括将要联合的 CI 和属性。
getDDMConfigurationService	配置数据流管理系统
getDDMManagementService	分析和查看数据流管理系统的进度、结果和错误
getDDMZoneService	导入和导出管理区域（及其活动）。
getHistoryService	受监控 CI 历史记录的相关信息（变更、删除等等）
getImpactAnalysisService	运行影响分析场景（也称为**关联**）。
getLicensingService	查询有关系统中已安装许可证的信息。
getMultipleCMDBService	在全局 ID 和 UCMDB ID 之间转换。
getMultiTenancyService	创建、读取、更新和删除租户。
getPersistencyService	将二进制数据持续保存到键值对中。
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。
getReconciliationService	提供标识和合并功能。
getResourceBundleManagementService	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。
getResourceManagementService	将（TQL 查询、视图、用户等）资源包部署到系统。
getSecurityService	验证凭据是否有效。
getServerService	查询有关系统的一般信息。
getSnapshotService	提供用于管理快照的服务（获取、保存、比较等等）
getSoftwareSignatureService	定义将通过数据流管理系统进行搜寻的软件项
getStateService	提供用于管理状态的服务（列出、添加、删除等等）
getSystemHealthService	提供系统运行状况服务（基本系统性能指示器、容量和可用性指标）

服务方法	用途
getTopologyQueryService	获取有关 IT 世界的信息
getTopologyUpdateService	更改 IT 世界中的信息
getUcmdbVersion	查询 UCMDB 和内容包版本以及内部版本信息。
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。
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。

客户端通过 HTTP(S) 与服务器通信。

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（其中 localhost 是安装 UCMDB 的计算机），并单击“API Client Download”链接可下载该文件。

在编译或运行应用程序之前，将 **.jar** 文件放入类路径中。

备注: 使用 UCMDB Java API Jar 时，需要安装 JRE 6 或更高版本。

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: 也可与常规 UCMDB 用户（例如，管理员）连接。但是，不建议使用此选项。要与 UCMDB 用户连接，必须为此用户授予 API 身份验证权限。

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
   您可能需要使用用户名和密码登录。
2. 在 UCMDB 下，单击 **service=UCMDB Authorization Services**。
3. 找到 **createUser** 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。

- **userDisplayName**。集成用户的显示名称。
- **userLoginName**。集成用户的登录名。
- **password**。集成用户的密码。
  默认密码策略要求 UCMDB 密码必须包括以下四种类型字符且每种类型至少有一个:
  - 大写字母字符
  - 小写字母字符
  - 数字字符
  - 符号字符 ,\:/. _?&%=+-[]()|

  还要求密码符合“密码最小长度”设置中所设置的最小长度。

4. 单击“Invoke”。
5. 在单租户环境下，找到 **setRolesForUser** 方法并输入以下参数:
   - **userName**。集成用户的用户名。
   - **roles**。超级管理员。

   单击“Invoke”。
6. 在多租户环境中，找到 **grantRolesToUserForAllTenants** 方法，并输入以下参数，分配与所有租户相关的角色:
   - **userName**。集成用户的用户名。
   - **roles**。超级管理员。

   单击“Invoke”。

   此外，为了分配与特定租户相关的角色，还可以使用相同的“userName”和“roles”参数值调用 **grantRolesToUserForTenants** 方法。对于 **tenantNames** 参数，请输入所需租户。
7. 创建更多用户，或关闭 JMX 控制台。
8. 以管理员身份登录到 UCMDB。
9. 在“管理”选项卡中，运行“包管理器”。
10. 单击“创建自定义包”图标。
11. 输入新程序包的名称，并单击“下一步”。
12. 在“资源选择”选项卡中，单击“设置”下的“用户”。
13. 选择一个或多个使用 JMX 控制台创建的用户。
14. 单击“下一步”，然后单击“完成”。此时，新程序包会出现在包管理器的“包名”列表中。
15. 将该包部署到要运行 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“如何部署包”章节。

备注: 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（**removeUser**、**resetPassword**、**UserAuthenticate** 等）。

存在两个现成系统用户。建议在使用 **resetPassword** 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 **SuperIntegrationUser**）。
  如果使用 **resetPassword** 更改 UISysadmin 密码，必须执行以下操作：
  i. 在 JMX 控制台中，找到 **UCMDB-UI:name=UCMDB Integration** 服务。
  ii. 以集成用户的用户名和新密码运行 **setCMDBSuperIntegrationUser**。

UCMDB API 用例

本节列出的用例假定使用两个系统：

- 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 287 页)
- 查询 CMDB (第 287 页)
- 查询类模型 (第 287 页)
- 分析变更影响 (第 288 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 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。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。

分析变更影响

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

请参阅以下代码示例:

- Create a Connection
- Create and Execute an Ad Hoc Query
- Create and Execute a View
- Add and Delete Data
- Execute an Impact Analysis
- Query the Class Model
- Query a History Sample

这些文件位于以下目录中:

\\<UCMDB 根目录>\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\JavaSDK_Samples

第 11 章: HP Universal CMDB Web 服务 API

本章包括:

- 约定 289
- HP Universal CMDB Web 服务 API 概述 289
- 调用 HP Universal CMDB Web 服务 292
- 查询 CMDB 292
- 更新 CMDB 295
- 查询 UCMDB 类模型 296
- 影响分析查询 297
- UCMDB 常规参数 297
- UCMDB 输出参数 299
- UCMDB 查询方法 300
- UCMDB 更新方法 311
- UCMDB 影响分析方法 313
- 实际状态 Web 服务 API 315
- UCMDB Web 服务 API 用例 317
- 示例 317

约定

本章使用以下约定:

- UCMDB 是指通用配置管理数据库本身。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。

有关请求和响应结构的完整文档，请参考HP UCMDB Web Service API Reference。这些文件位于以下文件夹中:

<UCMDB 根目录>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\CMDB_Schema\webframe.html

HP Universal CMDB Web 服务 API 概述

备注: 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
- 管理凭据：查看、添加、更新和删除
- 管理作业：查看状态、激活和停用
- 管理探测器范围：查看、添加和更新
- 管理触发器：添加或删除触发 CI，以及添加、删除或禁用触发器 TQL
- 查看域和探测器的常规数据

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“拓扑查询语言”。

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 290 页)
- 权限 (第 290 页)

API 的用途

UCMDB Web 服务 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 来创建一个可跟踪变更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变更分析。

权限

要访问 Web 服务的 WSDL 文件，请转到：**http://localhost:8080/axis2/services/UcmdbService?wsdl**。您需提供服务器管理员的用户凭据才能查看 WSDL 文件。

> **备注:** 不能访问 Axis2 管理控制台。

用户必须具有“运行旧 API”的常规操作权限，才能登录。

下表显示了每个 Web 服务 API 命令所需的其他权限：

Web 服务 API 命令	所需权限
addCIsAndRelations deleteCIsAndRelations updateCIsAndRelations	常规操作：**数据更新**
executeTopologyQueryByName(AdHoc) executeTopologyQueryByNameWithParameters(AdHoc) executeTopologyQueryWithParameters(AdHoc)	常规操作：**按定义运行查询** 对于每个查询：**查看**权限
getTopologyQueryExistingResultByName getTopologyQueryResultCountByName releaseChunks pullTopologyMapChunks getCINeighbours getFilteredCIsByType getCIsById getCIsByType getRelationsById	常规操作：**查看 CI** 对于每个查询：**查看**权限
getQueryNameOfView	常规操作：**查看 CI** 对于每个视图：**查看**权限
getChangedCIs	常规操作：**查看历史记录**、**查看 CI**
calculateImpact getImpactPath getImpactRulesByGroupName getImpactRulesByNamePrefix	常规操作：**运行影响分析**
getAllClassesHierarchy getClassAncestors getCmdbClassDefinition	无

备注: UCMDB 中的根上下文发生更改后，请执行以下步骤访问 Web 服务 API:

1. 打开 **\UCMDB\UCMDBServer\deploy\axis2\WEB-INF\web.xml** 配置文件并找到以下部分:

   ```
   <servlet-class>
   org.apache.axis2.transport.http.AxisServlet
   </servlet-class>
   ```

   在该部分后添加以下行:

   ```
   <init-param>
   <param-name>axis2.find.context</param-name>
   <param-value>false</param-value>
   </init-param>
   ```

2. 打开 **\UCMDB\UCMDBServer\deploy\axis2\WEB-INF\conf\axis2.xml** 配置文件并找到以下行:

   <parameter name="enableSwA" locked="false">false</parameter>

   在该行后添加以下行:

   <parameter name="contextRoot" locked="false">test1/setup1/axis2
   </parameter>

   其中 **test1/setup1** 是根上下文。
   （要删除根上下文，请删除添加到此路径的文本。）
3. 重新启动 UCMDB 服务器。

调用 HP Universal CMDB Web 服务

您可以使用 HP Universal CMDB Web 服务 API 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UCMDB Web 服务 API 用例 (第 317 页)。

查询 CMDB

您可以使用 UCMDB 查询方法 (第 300 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 UCMDB ID。有关使用查询方法的示例，请参阅Query Example。

本节包括以下主题:

- 及时响应计算 (第 293 页)
- 处理大量响应 (第 293 页)
- 指定要返回的属性 (第 293 页)
- 具体属性 (第 294 页)
- 派生属性 (第 294 页)
- 命名属性 (第 294 页)
- 其他属性规范元素 (第 294 页)

及时响应计算

对于所有查询方法，UCMDB 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。

有关处理成块数据的信息，请参阅pullTopologyMapChunks (第 309 页)和releaseChunks (第 310 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数量，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）时可使用此属性。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包括 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 CMDB

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 311 页)。

此任务包括以下步骤:

- UCMDB 更新参数 (第 295 页)
- 使用 ID 类型和更新方法 (第 295 页)

UCMDB 更新参数

本主题介绍了仅由服务的更新方法所使用的参数。

- **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。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键属性一起，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，便需要类型和键属性。

- **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 键属性。

- **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，而且必须指定其键属性。有关详细信息，请参阅 CIsAndRelationsUpdates (第 295 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 311 页)和 UCMDB 查询方法 (第 300 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 必须由真实 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”。

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 296 页)
- getAllClassesHierarchy (第 296 页)
- getCmdbClassDefinition (第 297 页)

getClassAncestors

getClassAncestors 方法检索指定 CIT 与其根之间（包括根）的路径。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
className	类型名称。有关详细信息，请参阅类型名称 (第 299 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 298 页)。

输出

参数	注释

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 检索键属性，还必须从父类向上查询到基类。
getCmdbClassDefinition 作为键属性只对包含在按 className 指定的类定义中设置的 ID_ATTRIBUTE 的属性进行标识。继承的键属性不会被识别为指定类的键属性。因此，指定类的键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 298 页)。
className	类型名称。有关详细信息，请参阅UCMDB 常规参数 (第 297 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。

影响分析查询

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Impact Analysis Example。

UCMDB 常规参数

本节介绍了服务方法最常用的参数。

本节包括以下主题：

- CmdbContext (第 298 页)
- ID (第 298 页)

- 键属性 (第 298 页)
- ID 类型 (第 298 页)
- CIProperties (第 298 页)
- 类型名称 (第 299 页)
- 配置项 (CI) (第 299 页)
- Relation (第 299 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。

键属性

在某些上下文中，键属性可用于代替 CMDB ID 来标识 CI 或 Relation。键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”属性列表中，键属性旁边会显示一个钥匙图标。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键属性的信息，请参阅 getCmdbClassDefinition (第 297 页)。

ID 类型

ID 元素可以包含一个真实 ID 或一个临时 ID。

真实 ID 是由 CMDB 所分配的用来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；

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 可以使用 CI 键属性标识现有的数据配置项，则在客户端传递某个临时 ID 时，即使某个 CI 使用真实 ID 进行了标识，该 CI 仍适用于上下文。

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 299 页)和Relation (第 299 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 UCMDB 常规参数 (第 297 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，也可以是临时 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 Type Name。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 296 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 298 页)。

UCMDB 输出参数

本节介绍了服务方法最常用的输出参数。有关更多详细信息，请参考online schema documentation.

本节包括以下主题:

- CIs (第 300 页)
- ShallowRelation (第 300 页)

- Topology (第 300 页)
- CINode (第 300 页)
- RelationNode (第 300 页)
- TopologyMap (第 300 页)
- ChunkInfo (第 300 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 Type Name。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 Relation 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 300 页)）。
- relationNodes。指一个或多个 RelationNode（请参阅RelationNode (第 300 页)）。

这两种结构中的 labels 用来将配置项和关系列表排序。

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 293 页)。

UCMDB 查询方法

本节提供以下方法的信息：

- executeTopologyQueryByNameWithParameters (第 301 页)
- executeTopologyQueryWithParameters (第 301 页)
- getChangedCIs (第 302 页)
- getCINeighbours (第 303 页)
- getCIsByID (第 304 页)
- getCIsByType (第 304 页)
- getFilteredCIsByType (第 304 页)
- getQueryNameOfView (第 307 页)
- getTopologyQueryExistingResultByName (第 308 页)
- getTopologyQueryResultCountByName (第 308 页)
- pullTopologyMapChunks (第 309 页)
- releaseChunks (第 310 页)

executeTopologyQueryByNameWithParameters

executeTopologyQueryByNameWithParameters 方法可检索与指定的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 298 页)。
queryName	CMDB 中要为其获取图的参数化 TQL 的名称。
parameterizedNodeList	每个节点要包含在查询结果中时所必须符合的条件。
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 300 页)。
chunkInfo	有关详细信息，请参阅ChunkInfo (第 300 页)和处理大量响应 (第 293 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 parameterizedNodeList 参数中传递。TQL 必须包含为每个 CINode 和 relationNode 定义的唯一标签。

executeTopologyQueryWithParameters 方法用于传递特别查询，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。

要将导出的 TQL 用作此方法的输入，请执行以下操作:

1. 启动 Web 浏览器并输入以下地址:
   http://localhost:8080/jmx-console。
   您可能需要使用用户名和密码登录。
2. 单击 **UCMDB:service=TQL Services**。
3. 找到 **exportTql** 操作。
   - 在 **customerId** 参数框中，输入 **1**（默认值）。
   - 在 **patternName** 参数框中，输入有效的 TQL 名称。
4. 单击“Invoke”。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
queryXML	XML 字符串，表示一个不包含资源标记的 TQL。
parameterizedNodeList	每个节点要包含在查询结果中时所必须符合的条件。

输出

参数	注释
topologyMap	有关详细信息，请参阅TopologyMap (第 300 页)。
chunkInfo	有关详细信息，请参阅ChunkInfo (第 300 页)和处理大量响应 (第 293 页)。

getChangedCIs

getChangedCIs 方法会返回所有与指定 CI 相关的 CI 的变更数据。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。

参数	注释
ids	系统将检查其相关 CI 的变更情况的根 CI 的 ID 列表。此集合中只有真实 CMDB ID 才是有效的。
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
getChangedCIsResponseList	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 298 页)。
ID	用于检索相邻项的 CI 的 ID。此 ID 必须是一个真实的 CMDB 或全局 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 299 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 294 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 294 页)。

输出

参数	注释
topology	有关详细信息，请参阅Topology (第 300 页)。
comments	仅供内部使用。

getClsByID

getClsByID 方法按配置项的 CMDB 或全局 ID 检索配置项。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
ClsTypedProperties	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 294 页)。
IDs	此集合中只有真实 CMDB 或全局 ID 才是有效的。

输出

参数	注释
Cls	CI 元素的集合。
chunkInfo	有关详细信息，请参阅ChunkInfo (第 300 页)和处理大量响应 (第 293 页)。

getClsByType

getClsByType 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 298 页)。
type	类名称。有关详细信息，请参阅类型名称 (第 299 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 294 页)。

输出

参数	注释
Cls	CI 元素的集合。
chunkInfo	有关详细信息，请参阅：ChunkInfo (第 300 页)和处理大量响应 (第 293 页)。

getFilteredClsByType

getFilteredClsByType 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。 **备注:** 保留名称 ChangedDuring 是为了保持向后兼容性。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 条件值和属性值均为列表。 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。
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。

运算符	条件/注释类型
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。 **备注:** 保留名称 UnchangedDuring 是为了保持向后兼容性。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(11f);
fc.setCondition(fp);
fc.setFloatOperator(FloatCondition.FloatOperator.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.setPropertyNames(Arrays.asList("name","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 298 页)。
type	类名称。有关详细信息，请参阅类型名称 (第 299 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 294 页)。
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 300 页)和处理大量响应 (第 293 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 298 页)。
viewName	视图的名称，即 CMDB 中类模型的子集。

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 298 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
topologyMap	有关详细信息，请参阅TopologyMap (第 300 页)。
chunkInfo	有关详细信息，请参阅ChunkInfo (第 300 页)和处理大量响应 (第 293 页)。

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 298 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
getTopologyQueryResultCountByNameResponse	与查询匹配的实例数。

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 300 页)和查询 CMDB (第 292 页)。

客户端应用程序必须能够处理部分图。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
ChunkRequest	要检索的块和由查询方法返回的 ChunkInfo 的数量。
queryTypedProperties	特定 CI 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 300 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
         }
      }
}

GetCIsByType request =
    new GetCIsByType(cmdbContext, typeName, customProperties);
GetCIsByTypeResponse response =
    ucmdbService.getCIsByType(request);
ChunkRequest chunkRequest = new ChunkRequest();
chunkRequest.setChunkInfo(response.getChunkInfo());
for(int j=1 ; j <= response.getChunkInfo().getNumberOfChunks() ; j++) {
   chunkRequest.setChunkNumber(j);
   PullTopologyMapChunks req = new PullTopologyMapChunks(cmdbContext, chunkRequest);
   PullTopologyMapChunksResponse res =
      ucmdbService.pullTopologyMapChunks(req);
      for(int m=0 ;
         m < res.getTopologyMap().getCINodes().getCINodes().size();
         m++) {
         CIs cis =
         res.getTopologyMap().getCINodes().getCINodes().get(m).getCIs();
         for(int i=0 ; i < cis.getCIs().size(); i++) {
            // your code to process the CIs
         }
      }
}
```

releaseChunks

releaseChunks 方法会释放包含查询数据的块的内存。

提示: 服务器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 298 页)。

参数	注释
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 311 页)
- addCustomer (第 312 页)
- deleteCIsAndRelations (第 312 页)
- removeCustomer (第 312 页)
- updateCIsAndRelations (第 313 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 298 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 295 页)。

参数	注释
ignoreValidation	如果为 True，则在更新 CMDB 之前不会执行检查。
dataStore	更改者信息。

输出

参数	注释
createdIDsMapList	客户端 ID 到 CMDB ID 的映射列表。有关详细信息，请参阅上面的描述。
comments	仅供内部使用。

addCustomer

addCustomer 方法可以添加客户。

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 298 页)。
CIsAndRelationsUpdates	要删除的项。有关详细信息，请参阅 CIsAndRelationsUpdates (第 295 页)。
dataStore	更改者信息。

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 298 页)。
CIsAndRelationsUpdates	要更新的项。有关详细信息，请参阅CIsAndRelationsUpdates (第 295 页)。
ignoreValidation	如果为 True，则在更新 CMDB 之前不会执行检查。
dataStore	更改者信息。

输出

参数	注释
createdIDsMapList	客户端 ID 到 CMDB ID 的映射列表。有关详细信息，请参阅addCIsAndRelations (第 311 页)。

UCMDB 影响分析方法

本节提供以下方法的信息：

- calculateImpact (第 313 页)
- getImpactPath (第 314 页)
- getImpactRulesByNamePrefix (第 315 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 (第 314 页) 的输入。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
impactCategory	会触发要模拟的规则的事件类型。
IDs	CMDB 或全局 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 300 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 (第 313 页) 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 298 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。

参数	注释
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 298 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，可指定变更效果。

实际状态 Web 服务 API

“实际状态 Web 服务 API”主要由 Service Manager 用来检索特定 CMDB ID 或全局 ID 和特定客户 ID 的实际状态信息。API 会在 **Integration/SM Query** 文件夹下查找匹配的查询，然后以 CMDB ID 或全局 ID 为条件执行 TQL，并返回查询输出。

Web 服务 URL： http://[计算机名]:8080/axis2/services/ucmdbSMService

Web 服务架构： http://[计算机名]:8080/axis2/services/ucmdbSMService?xsd=xsd0

流

调用 API 方法时，该方法会尝试在 **Integration/SM Query** 文件夹中查找相应查询。首先尝试将后一文件夹中的一个查询与请求的 CMDBID/GlobalID 类型相匹配，方法是查找名称为 **Root** 的 **QueryElement**，如果未找到，则尝试使用与请求的 CMDBID/GlobalID 同类型的任何 **QueryNode**。找到相应和查询和 QueryNode 后，便会将 CMDBID/GlobalID 作为对 QueryNode 的条件，并执行该查询。然后，将结果返回到 API 的调用方。

使用转换功能操纵结果

在某些情况下，您可能要对生成的 XML 应用其他转换（例如，汇总所有磁盘的大小并将总和作为附加属性添加到 CI）。要对 TQL 结果添加其他转换，请按如下所示在适配器配置中放置一个名为 [tql_name].xslt 的资源：“适配器管理”>“ServiceDeskAdapter7-1”>“配置文件”>“[tql_name].xslt”。

实际状态 Web 服务 API 的日志

UCMDB 的日志配置位于以下位置：UCMDBServer/Conf/log 的各种 *.properties 文件中。

要查看 SM 实际状态流的日志：

1. **打开 cmdb_soaapi.properties 文件，并将日志级别更改为 DEBUG，如下所示：loglevel=DEBUG。**
2. **打开 fcmdb.properties 文件，并将日志级别更改为 DEBUG，如下所示：loglevel=DEBUG。**
3. **等待 1 分钟让服务器检索变更。**
4. **从 SM 运行实际状态。**
5. **查看 UCMDBServer/Runtime/log 中的以下日志文件：**
   - cmdb.soaapi.log
   - fcmdb.log

更改根上下文后支持已复制 CI 的实际状态

如果已更改用于访问 UCMDB 的根上下文，则必须更改以下配置，启用“已复制 CI 的实际状态”：

1. **在 UCMDBServer\deploy\axis2\WEB-INF 下，打开文件 web.xml。**
2. **将以下 servlet init 参数添加到 AxisServlet（将以下四行粘贴到第 28 行之后）：**

   ```
   <init-param>
   <param-name>axis2.find.context</param-name>
   <param-value>false</param-value>
   </init-param>
   ```

   此设置可防止 Axis2 尝试计算上下文根，并指示 Axis2 在 axis2.xml 中明确查找该根。
3. **在 UCMDBServer\deploy\axis2\WEB-INF\conf 下，打开文件 axis2.xml。**
4. **在第 58 行，从参数 contextRoot 中删除注释，然后编辑如下：**

   ```
   <parameter name="contextRoot" locked="false">test/axis2</parameter>
   ```

   （其中，test 是指 cmdb.xml 中的新根上下文）。

   备注: test/axis2 的开头没有正斜杠。

UCMDB Web 服务 API 用例

以下用例假定使用两个系统:

- **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 317 页)**
- **查询 CMDB (第 317 页)**
- **查询类模型 (第 317 页)**
- **分析变更影响 (第 317 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**
- **大量第三方系统可以通过填充 CMDB 来创建一个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。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。**

分析变更影响

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

请参阅以下代码示例:

- The Example Base Class
- Query Example
- Update Example
- Class Model Example
- Impact Analysis Example

这些文件位于以下目录中:

\\<UCMDB 根目录>\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\WebServiceAPI_Samples

第 12 章: 数据流管理 Java API

本章包括:

- 使用数据流管理 Java API 319

使用数据流管理 Java API

备注: 请将本章与联机文档库中的 DFM API Javadoc 配合使用。

本章描述第三方工具或自定义工具如何使用 HP 数据流管理 Java API 来管理数据流。通过该 API 可以:

- **管理凭据**。查看、添加、更新和删除。
- **管理作业**。查看状态、激活和停用。
- **管理探测器范围**。查看、添加和更新。
- **管理触发器**。添加或删除触发 CI，以及添加、删除或禁用触发器 TQL。
- **查看常规数据**。域和探测器上的数据。

搜寻服务包中提供以下服务:

- **DDMConfigurationService**。用于配置 Data Flow Probe、群集、IP 范围和凭据的服务。可使用 XML 文件或通过 Data Flow Probe 配置 Universal Discovery 服务器。
- **DDMManagementService**。用于分析和查看 Universal Discovery 运行的进度、结果和错误的服务。
- **DDMSoftwareSignatureService**。用于定义 Data Flow Probe 组件将要搜寻到的软件项的服务。这些定义适用于整个系统。如果定义多个 Data Flow Probe 组件，则这些定义将应用于所有这些组件。
- **DDMZoneService**。用于管理基于区域的搜寻的服务。

除这些服务以外，还提供数据流管理客户端 API，用于创建 Jython 适配器。有关详细信息，请参阅开发 Jython 适配器 (第 34 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。

此外，用户必须具有“访问 SDK”常规操作的权限才能登录。

警告: API 客户端还适用于常规用户，只要这些用户具有 API 身份验证权限即可。但是，不建议使用此选项。

有关详细信息，请参阅创建集成用户 (第 285 页)。

第 13 章: 数据流管理 Web 服务 API

本章包括:

- 数据流管理 Web 服务 API 概述 320
- 约定 320
- 调用 HP 数据流管理 Web 服务 321
- 数据流管理方法和数据结构 321
- 代码示例 332
- 添加凭据示例 334

数据流管理 Web 服务 API 概述

本章描述第三方工具或自定义工具如何使用 HP 数据流管理 Web 服务 API 来管理数据流。

HP 数据流管理 Web 服务 API 用于将应用程序与 HP Universal CMDB 集成。通过该 API 可以:

- **管理凭据**。查看、添加、更新和删除。
- **管理作业**。查看状态、激活和停用。
- **管理探测器范围**。查看、添加和更新。
- **管理触发器**。添加或删除触发 CI，以及添加、删除或禁用触发器 TQL。
- **查看常规数据**。域和探测器上的数据。

HP 数据流管理 Web 服务的用户应该熟悉:

- 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

备注:

- 用户必须具有“运行旧 API”常规操作的权限才能登录。
- 已登录用户必须具有“运行搜寻和集成”常规操作的权限才能访问任何方法。

有关可用操作的完整文档，请参阅《HP Universal Discovery Schema Reference》。这些文件位于以下文件夹中:

<UCMDB 根目录>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_Schema\webframe.html

约定

本章使用以下约定:

- 此样式 Element 表示该项目是数据库中的实体或架构中定义的元素，包括传递给方法的结构或由方法返回的结构。纯文本表示该项目在一般语境的叙述中出现。
- 数据流管理元素和方法参数的拼写使用架构中指定的大小写。这通常意味着，类名称或对类实例的一般引用使用大写形式。元素或方法参数不必大写。例如，credential 是传递到某方法的类型 Credential 的元素。

调用 HP 数据流管理 Web 服务

HP 数据流管理 Web 服务 API 支持使用标准 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，该服务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。

要调用此项服务，请使用以下内容：

- Protocol：http 或 https（取决于服务器配置）
- URL：<UCMDB 服务器>:8080/axis2/services/DiscoveryService
- Default password："admin"
- Default username："admin"

SOAP 程序员可通过以下地址访问 WSDL：

- axis2/services/DiscoveryService?wsdl

数据流管理方法和数据结构

本节列出了数据流管理 Web 服务 API 方法和数据结构，以及这些方法的用途简介。有关针对每个操作的请求和响应的完整文档，请参阅《HP Universal Discovery Schema Reference》。

本节包括以下主题：

- 数据结构 (第 321 页)
- 管理搜寻作业方法 (第 322 页)
- 管理触发器方法 (第 324 页)
- 域和探测器数据方法 (第 325 页)
- 凭据数据方法 (第 328 页)
- 数据刷新方法 (第 330 页)

数据结构

以下是在数据流管理 Web 服务 API 中使用的部分数据结构。

CIProperties

CIProperties 是一组集合。每个集合都包含不同数据类型的属性。例如，可能有 dateProps 集合、strListProps 集合、xmlProps 集合等等。

每个类型的集合都包含给定类型的各个属性。这些属性元素的名称与容器相同，但为单数形式。例如，dateProps 包含 dateProp 元素。每个属性均以“名称-值”的方式成对出现。

请参阅《HP Universal Discovery Schema Reference》中的 CIProperties。

IPList

IP 元素的列表，每个元素都包含一个 IPv4 或 IPv6 地址。

请参阅《HP Universal Discovery Schema Reference》中的 IPList。

IPRange

IPRange 有两个元素，即 Start 和 End。每个元素都包含一个 Address 元素，即一个 IPv4 或 IPv6 地址。

请参阅《HP Universal Discovery Schema Reference》中的 IPRange。

Scope

两个 IPRanges。Exclude 是要从作业中排除的 IPRanges 的集合。Include 是要包括在作业中的 IPRanges 的集合。

请参阅《HP Universal Discovery Schema Reference》中的 Scope。

管理搜寻作业方法

activateJob

激活指定作业。

请参阅代码示例 (第 332 页)。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
JobName	作业的名称。

deactivateJob

停用指定作业。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
JobName	作业的名称。

dispatchAdHocJob

在探测器上分派特别作业。该作业必须处于活动状态，并且包含指定的触发 CI。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
JobName	作业的名称。
CIID	触发 CI 的 ID。
ProbeName	探测器的名称。
Timeout	单位为毫秒

getDiscoveryJobsNames

返回作业名称的列表。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。

输出

参数	注释
strList	作业名称的列表。

isJobActive

检查作业是否处于活动状态。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
JobName	要检查的作业名称。

输出

参数	注释
JobState	当作业处于活动状态时为 True。

管理触发器方法

addTriggerCI

将新触发 CI 添加到指定作业。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
JobName	作业的名称。
CIID	触发 CI 的 ID。

addTriggerTQL

将新触发器 TQL 添加到指定作业。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
JobName	作业的名称。
TqlName	要添加的 TQL 的名称。

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。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
JobName	作业的名称。

removeTriggerCI

从触发作业的 CI 列表中删除指定 CI。

输入

参数	注释

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
JobName	作业名称。
CIID	触发 CI 的 ID。

removeTriggerTQL

从触发作业的查询列表中删除指定 TQL。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
JobName	要检查的作业名称集合。
CIID	要删除的 TQL 的 ID。

setTriggerTQLProbesLimit

将作业中 TQL 处于活动状态的探测器限制到指定列表。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
JobName	作业的名称。
tqlName	TQL 名称。
probesLimit	TQL 处于活动状态的探测器的列表。

域和探测器数据方法

getDomainType

返回域类型。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。

参数	注释
domainName	域的名称。

输出

参数	注释
domainType	域类型。

getDomainsNames

返回当前域的名称。

请参阅代码示例 (第 332 页)。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。

输出

参数	注释
domainNames	域名列表。

getProbeIPs

返回指定探测器的 IP 地址。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
domainName	要检查的域。
probeName	在该域上使用的探测器的名称。

输出

参数	注释
probeIPs	探测器中地址的IPList。

getProbesNames

返回指定域中探测器的名称。

请参阅代码示例 (第 332 页)。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
domainName	要检查的域。

输出

参数	注释
probesName	域上探测器的列表。

getProbeScope

返回指定探测器的范围定义。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
domainName	要检查的域。
probeName	探测器的名称。

输出

参数	注释
probeScope	探测器的 Scope。

isProbeConnected

检查指定探测器是否已连接。

请参阅代码示例 (第 332 页)。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
domainName	要检查的域。
probeName	要检查的探测器。

输出

参数	注释
isConnected	当探测器已连接时为 True。

updateProbeScope

设置指定探测器的范围，以替代现有范围。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
domainName	域。
probeName	要更新的探测器。
newScope	要为探测器设置的Scope。

凭据数据方法

addCredentialsEntry

将凭据条目添加到指定域的指定协议。

请参阅代码示例 (第 332 页)。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
domainName	要更新的域。
protocolName	协议的名称。
credentialsEntryParameters	新凭据的 CIProperties集合。

输出

参数	注释
credentialsEntryID	新凭据条目的 CI ID。

getCredentialsEntriesIDs

返回为指定协议定义的凭据的 ID。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
domainName	要获取凭据的域。
protocolName	在该域上所使用协议的名称。

输出

参数	注释
credentialsEntryIDs	域上协议的凭据 ID 列表。

getCredentialsEntry

返回为指定协议定义的凭据。将返回空加密属性。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
domainName	要获取凭据的域。
protocolName	在该域上所使用协议的名称。
credentialsEntryID	要获取的凭据 ID。

输出

参数	注释
credentialsEntryParameters	凭据的 CIProperties集合。

removeCredentialsEntry

从协议删除指定证书。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。

参数	注释
domainName	域。
protocolName	在该域上所用协议的名称。
credentialsEntryID	要删除的凭据的 ID。

updateCredentialsEntry

设置指定凭据条目的属性的新值。

将删除现有属性，并设置这些属性。未在该调用中设置值的任何属性都会保持未定义状态。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
domainName	要更新凭据的域。
protocolName	在该域上所用协议的名称。
credentialsEntryID	要更新的凭据的 ID。
credentialsEntryParameters	要设置为凭据属性的 CIProperties集合。

数据刷新方法

rediscoverCIs

找到搜寻指定 CI 对象的触发器并重新运行这些触发器。rediscoverCIs 以异步方式运行。调用 checkDiscoveryProgress 可以确定重新搜寻完成的时间。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
CmdbIDs	要重新搜寻的对象的 ID 集合。

输出

参数	注释
isSucceed	CI 重新搜寻成功时为 True。

checkDiscoveryProgress

返回对指定 ID 最近一次调用 rediscoverCIs 的进度。响应是一个 0 至 1 之间的值。当响应为 1 时，说明 rediscoverCIs 调用已经完成。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
CmdbIDs	要跟踪的重新搜寻调用中对象的 ID 集合。

输出

参数	注释
progress	已完成作业的进度为 1。尚未完成作业的进度为小于 1 的小数。

rediscoverViewCIs

找到已创建数据用于填充指定视图的触发器，并重新运行这些触发器。rediscoverViewCIs 以异步方式运行。调用 checkViewDiscoveryProgress 可以确定重新搜寻完成的时间。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
viewName	要检查的视图。

输出

参数	注释
isSucceed	CI 重新搜寻成功时为 True。

checkViewDiscoveryProgress

返回对指定视图最近一次调用 rediscoverViewCIs 的进度。响应为一个 0 至 1 之间的值。当响应为 1 时，说明 rediscoverCIs 调用已经完成。

输入

参数	注释
cmdbContext	有关详细信息，请参阅CmdbContext (第 298 页)。
viewName	要检查的视图集合。

输出

参数	注释
progress	已完成作业的进度为 1。尚未完成作业的进度为小于 1 的小数。

代码示例

```
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.*;
import com.hp.ucmdb.generated.types.*;
public class test {
   static final String HOST_NAME = "<my_hostname>";
   static final int PORT = 8080;
   private static final String PROTOCOL = "http";
   private static final String FILE = "/axis2/services/DiscoveryService";

   private static final String PASSWORD = "<my_password>";
   private static final String USERNAME = "<my_username>";

   private static CmdbContext cmdbContext = new CmdbContext("ws tests");

   public static void main(String[] args) throws Exception {
      // Get the stub object
      DiscoveryService discoveryService = getDiscoveryService();

      // Activate Job
      discoveryService.activateJob(new ActivateJobRequest(
         "Range IPs by ICMP", cmdbContext));

      // Get domain & probes info
      getProbesInfo(discoveryService);
      // Add credentilas entry for ntcmd protcol
      addNTCMDCredentialsEntry();
   }

   public static void addNTCMDCredentialsEntry() throws Exception {
      DiscoveryService discoveryService = getDiscoveryService();

      // Get domain name
      StrList domains =
         discoveryService.getDomainsNames(
            new GetDomainsNamesRequest(cmdbContext)).
            getDomainNames();
      if (domains.sizeStrValueList() == 0) {
         System.out.println("No domains were found, can't create credentials");
         return;
```

```
        }
        String domainName = domains.getStrValue(0);
        // Create propeties with one byte param
        CIProperties newCredsProperties = new CIProperties();

        // Add password property - this is of type bytes
        newCredsProperties.setBytesProps(new BytesProps());
        setPasswordProperty(newCredsProperties);

        // Add user & domain properties - these are of type string
        newCredsProperties.setStrProps(new StrProps());
        setStringProperties("protocol_username", "test user", newCredsProperties);
        setStringProperties("ntadminprotocol_ntdomain",
          "test doamin", newCredsProperties);

        // Add new credentials entry
        discoveryService.addCredentialsEntry(
          new AddCredentialsEntryRequest(domainName,
          "ntadminprotocol",  newCredsProperties, cmdbContext));
        System.out.println("new credentials craeted for domain:" + domainName + " in ntcmd protocol");
    }

    private static void setPasswordProperty(CIProperties newCredsProperties) {
        BytesProp bProp = new BytesProp();
        bProp.setName("protocol_password");
        bProp.setValue(new byte[] {101,103,102,104});
        newCredsProperties.getBytesProps().addBytesProp(bProp);
    }

    private static void setStringProperties(String propertyName, String value, CIProperties
newCredsProperties) {
        StrProp strProp = new StrProp();
        strProp.setName(propertyName);
        strProp.setValue(value);
        newCredsProperties.getStrProps().addStrProp(strProp);
    }


    private static void getProbesInfo(DiscoveryService discoveryService) throws Exception {
        GetDomainsNamesResponse result = discoveryService.getDomainsNames(new
GetDomainsNamesRequest(cmdbContext ));
        // Go over all the domains
        if (result.getDomainNames().sizeStrValueList() > 0) {
          String domainName =
          result.getDomainNames().getStrValue(0);
          GetProbesNamesResponse probesResult =
            discoveryService.getProbesNames(
              new GetProbesNamesRequest(domainName, cmdbContext));
          // Go over all the probes
          for (int i=0; i<probesResult.getProbesNames().sizeStrValueList(); i++) {
```

```
            String probeName = probesResult.getProbesNames().getStrValue(i);
            // Check if connected
            IsProbeConnectedResponce connectedRequest =
               discoveryService.isProbeConnected(
                  new IsProbeConnectedRequest(
                  domainName, probeName, cmdbContext));
            Boolean isConnected = connectedRequest.getIsConnected();
            // Do something ...
            System.out.println("probe " + probeName + " isconnect=" + isConnected);
         }
      }
   }

   private static DiscoveryService getDiscoveryService() throws Exception {
      DiscoveryService discoveryService = null;
      try {
         // Create service
         URL url = new URL(PROTOCOL,HOST_NAME,PORT, FILE);
         DiscoveryServiceStub serviceStub =
            new DiscoveryServiceStub(url.toString());

         // Authenticate info
         HttpTransportProperties.Authenticator auth =
            new HttpTransportProperties.Authenticator();
         auth.setUsername(USERNAME);
         auth.setPassword(PASSWORD);
         serviceStub._getServiceClient().getOptions().setProperty(
            HTTPConstants.AUTHENTICATE,auth);

         discoveryService = serviceStub;
      } catch (Exception e) {
         throw new Exception("cannot create a connection to service ", e);
      }
      return discoveryService;
   }
}
```

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.AUTHENTICATE,auth);

            discoveryService = serviceStub;
        } catch (Exception e) {
            throw new Exception("cannot create a connection to service ", e);
        }
        return discoveryService;
    }
} // End class
```

发送文档反馈

如果您对本文档有任何意见或建议，可以通过电子邮件联系文档团队。如果在此系统上配置了电子邮件客户端，请单击以上链接，此时将打开一个电子邮件窗口，主题行中为以下信息：

开发人员参考指南 (Universal CMDB 10.20) 反馈

只需在电子邮件中添加反馈并单击发送即可。

如果没有可用的电子邮件客户端，请将以上信息复制到 Web 邮件客户端的新邮件中，然后将您的反馈发送至 cms-doc@hp.com 。

非常感谢您的反馈！