

北京邮电大学

硕士学位论文

企业服务总线（Mule ESB）的研究与实现

姓名：韩纬禧

申请学位级别：硕士

专业：软件工程

指导教师：朱其亮

20070601

企业服务总线（Mule ESB）的研究与实现

摘 要

随着 SOA 技术的不断发展以及应用的不断扩大, 作为实现 SOA 的主要技术之一的中间件技术也越来越受到人们的关注。通过中间件技术, 人们可以更好地把各个服务连接到一起, 也可以把已存在的系统无缝隙地结合在一起, 避免了资源的浪费以及重新开发的难度。

本文是根据作者本人在实习期间所参与的 Mule ESB 的研究与应用项目而写成。该项目的主要目的是通过研究国外开源项目 Mule, 了解它的原理以及应用, 并将第三方产品实时数据库 Agilor 和 Mule 相结合, 为今后利用 Mule 进行开发打下基础。

本文第三、四章是本人在这次项目中对 Mule 的剖析, 主要介绍了 Mule 的技术架构以及它的工作原理, 对读者了解 Mule 有着重要的作用, 同时介绍了 Mule 目前已支持的技术, 开发人员可以有目的的利用它进行相应的开发。第五章则是本人在此次项目中的开发记录, 相信对开发人员利用 Mule 进行开发一定会有所帮助。

此次项目所涉及的工作比较多也比较杂, 因此本文重点描述的是 Mule 相关的理论和一些技术的应用, 而至于 Mule IDE 汉化、Jar 包重组和 Mule API 汉化则由于只是和具体编程相关, 因而不在于描述之列。最后则对此次项目进行总结。

关键词: 面向服务架构 企业服务总线 Java 消息服务 Web 服务

RESEARCH AND IMPLEMENT OF MULE ENTERPRISE SERVICE BUS

ABSTRACT

With the development and using of SOA, the enterprise service bus, as one of the skills which implements SOA, is becoming more and more important and catches much more public attention. People can connect different services so fluently and handle different systems together so tightly that avoid the waste of resources and the difficulties of redevelopment.

This paper was written with the writer's experience in researching and implementing of Mule ESB when taking practice. The primary purpose of this project is to understand the principle and the uses of Mule ESB by researching the foreign open project named Mule, and then combine the real time database named Agilor to Mule. It is good for us to develop systems with mule in the future.

The third and forth chapters of this paper are analyses of Mule, including the technical overview construction and the principle of work of Mule. It is important for readers to understand Mule. At the same time I will introduce the skills which Mule has supported, so the developers

can carry on the corresponding development by using Mule. The chapter five is my records in this project development, I believe it will be good for developers who using Mule.

During the period of this project, there are too many jobs I have done. But I can't write it down completely. so the technical overhead construction and the principle of work and Some important technology application such as JMS、 Web Service of Mule and the technique of how to combine the real time database named Agilor to Mule are the key descriptions. But the jobs I have done like Chinese charactering the Mule IDE and Mule API and the reorganization of Jar files are not mentioned in this paper, because they are just with programming correlation. And the last is the conclusion of the project.

KEY WORDS: SOA ESB JMS Web Service

独创性（或创新性）声明

本人声明所呈交的论文是本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京邮电大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名： 韩伟强 日期： 2007.6.29

关于论文使用授权的说明

本人完全了解北京邮电大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属北京邮电大学。学校有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许学位论文被查阅和借阅；学校可以公布学位论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存、汇编学位论文。

本学位论文不属于保密范围，适用本授权书。

本人签名： 韩伟强 日期： 2007.6.29
导师签名： 牛其立 日期： 2007.6.29.

第一章 引言

1.1 课题背景

1.1.1 ESB 产生的背景

大约在 2003 年中的时候，SOA^[1]的概念逐渐进入人们的视野，一时间众人乐此不疲地发表各自对 SOA 的见解。SOA 已经成为 IT 业，尤其是软件开发及系统集成领域从业者的热门话题。

关于 SOA 的概念，可以找到很多的文章从不同的角度来描述它，不同的软件提供商也有不同的定义方式。BEA 有 流体计算，微软有 Indigo 和 SOA-building，SAP 有 ESA。每个人都可以从不同的视角来理解 SOA，从程序员的角度，SOA 是一种全新的开发技术，新的组件模型，比如说 Web Service^[2]；从架构设计师的角度，SOA 就是一种新的设计模式，方法学；从业务分析人员的角度，SOA 就是基于标准的业务应用服务。从概念的角度，IBM 对 SOA 的定义是最为全面的，既 SOA 是一种构造分布式系统的方法，它将业务应用功能以服务的形式提供给最终用户应用或其他服务。

SOA 的要素有：一个体系架构，用开放的标准将软件资产(Asset)化为服务；提供标准的方法来表示软件资产及其交互；单独的软件资产作为构造单元，被重复使用来开发其他应用；将关注点从细节实现转移到应用(application)组装；整合企业外部的应用（B2B）的方式；开发（现在）和整合（未来）的统一。

SOA 出现的必然性：

- 面向机器语言(Monolithic)的开发模式：需要根据不同平台的机器语言来开发代码。
- 面向过程(Procedure)的开发模式：独立于机器的程序语言(C, Pascal 等)使开发过程变得简单了，用过程来代表一个抽象的代码集合，包装重用

现成的代码。

- 面向对象(Object)的开发模式：用更接近现实的对象来表述一个相对完整的事物。面向对象的语言(Smalltalk, Java 等), 提供了更抽象的封装和重用模式。面向对象的开发强调从现实世界问题域到软件程序的直接映射, 更接近人类的自然思维方式。
- 面向组件(Component)的模式：随着软件开发规模的扩大, 在涉及分布式、异构等复杂特征的环境中, 代码级别的重用性差, 可维护性差, 效率低的弱点是不可逾越的, 因此人们以架构运行环境(如.Net, J2ee 等)来提供完善的支撑平台, 从而把开发者解放出来, 更专注于业务核心的开发。而这些业务功能(Business Function) 以组件的形式(DCOM^[3], EJB 等)发布运行在架构运行环境中。软件开发的重用模式也上升到业务组件的级别。
- 面向服务(SOA)的模式：当软件的使用范围扩展到更广阔的范围, 往往会面对更加复杂的 IT 环境和更加灵活多变的需求。服务(Service)的概念出现了, 人们将应用(Application)以业务服务(Business Service)的形式公布出来供别人使用, 而完全不需要去考虑这些业务服务运行在哪一个架构体系上, 因为所有的服务都讲着同样的语言。SOA 考虑了业务发展的长期性, 体现了“变化就是永恒”的思想。SOA 的核心体现在企业应用或者业务功能上的“重用”和“互操作”, 而不再把 IT 与业务对立起来, 这可以被视为在 IT 驱动业务的方向上迈出的重要一步。

典型的 SOA 架构的基本要求：SOA 在相对较粗的粒度上对应用服务或业务模块进行封装与重用；服务间保持松散耦合，基于开放的标准，服务的接口描述与具体实现无关；灵活的架构-服务的实现细节，服务的位置乃至服务请求的底层协议都应该透明；

假如我们已经按照 SOA 的思想提炼出了各种业务服务，公布出来，需要相互调用各自的服务，那怎么调用呢，如果如图 1-1 所示，只有服务，而服务的请求者和服务的提供者之间仍然需要这种显式的点到点的调用，那么这就不是一个典型的 SOA 架构。因此，在 SOA 中，我们还需要这样一个中间层，能够帮助实现在 SOA 架构中不同服务之间的智能化管理。最容易想到的是这样一个

HUB-Spoke 结构，在 SOA 架构中的各服务之间设置一个类似于 Hub 的中间件，由它充当整个 SOA 架构的中央管理器的作用。如果如图 1-2 所示，现在服务的请求者和提供者之间有了一个智能的中转站，服务的请求者不再需要了解服务提供者的细节。这看上去是一个好的 SOA 结构。事实上，传统的 EAI 就是通过这样一种方式来试图解决企业内部的应用整合问题。

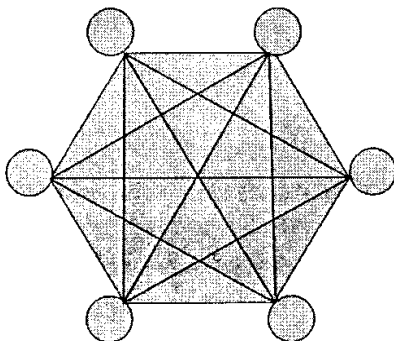


图 1-1 节点间互联

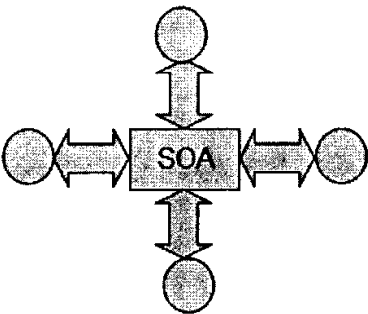


图 1-2 采用中间层

EAI 的目标是支持对现有 IT 系统的重新利用，通过 EAI 技术能够将不同的软件和系统串联起来，延长这些应用系统的生命周期。传统的 EAI，往往使用如 CORBA 和 COM 等的消息中间件进行分布式，跨平台的程序交互，修改企业资源规划以达到新的目标，使用中间件、XML 等方法来进行数据分配。因此，实际上传统的 EAI 是部件级的重用。很不幸的是，基于部件的架构没有统一的标准，因此，各个厂商都有各自不同的 EAI^[4]解决方案，你会看到各种各样的中间件平台。如果 EAI 碰到了异构的 IT 环境，就必须分别考虑怎样在各个不同的中间件之间周旋，来实现合理的互联方式，你不得不考虑各种复杂的可能性。因此，你所见过的大多数传统 EAI 解决方案都比较笨重。

如果选择 Hub 的模式来构建 SOA 基础架构，从纯粹逻辑的角度，可能会出现哪些问题呢？首先，整个 SOA 架构的性能，如果每个服务的请求都经过中央 Hub 的中转，那么 Hub 的负担会很重，速度会随着参与者的增多而愈来愈慢；其次，这样的系统会很脆弱，一旦 Hub 出错，整个 SOA 架构都会瘫痪；最后，这样的架构会破坏 SOA 的开放性原则，参与者运行在一个相对封闭的环境中，扩展起来十分麻烦。因此，这也不是理想的 SOA 架构。这时我们就引进了 ESB

的概念。如图 1-3 所示：

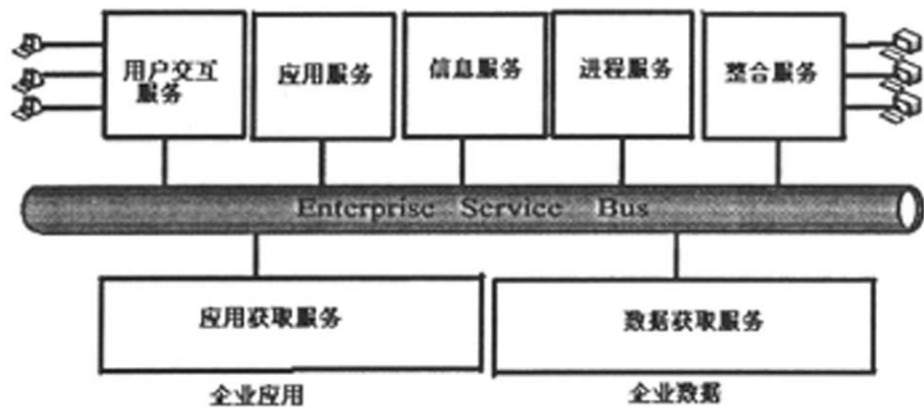


图 1-3 ESB 应用

它与前面的 Hub 结构有什么不同呢？首先，它比单一 Hub 的形式更开放，总线结构有无限扩展的可能；其次，真正体现了 SOA 的理念，一切皆为服务，服务在总线(BUS)中处于平等的地位。即使我们需要一些 Hub，那么它们也是以某种服务的形式部署在总线上，相比上面的结构要灵活的多。这就是 ESB，我们需要给它一个明确的定义：ESB 是一种在松散耦合的服务和应用之间标准的集成方式。它可以作用于：面向服务的架构：分布式的应用由可重用的服务组成；面向消息的架构：应用之间通过 ESB 发送和接受消息；事件驱动的架构：应用之间异步产生和接收消息。用一句较通俗的话来描述它：ESB 就是在 SOA 架构中实现服务间智能化集成与管理的中介^[3]。

1.1.2 ESB 当前主要的开源产品

当前主要有三种开源的 ESB 产品：

1. Object Web 的 Celtix：它原来是一个商业软件，因为卖的不好，所以转到 OW 下面，想增加一点知名度。
2. 第二个是 codehaus 的 ServiceMix：它是真正意义上的兼容 JSR208 JBI^[6]标准的 ESB 容器。它的主要特性包含：JBI 容器，脚本支持，规则引擎，工作流引擎，BPEL 引擎，Web Services Notification 实现。
3. 第三个是 codehaus 的 Mule：Mule 不算真正意义上的 JBI 实现。它更像是一个对象代理。但是因为它先走一步，所以已经非常成熟。它对 EIP 企

业应用集成提供了一套完整的支持。另外他也提供了 JBI 的接口。

1.2 论文工作

通过课题调研,了解 ESB 需要的相关知识,了解 Mule ESB 的技术架构以及工作原理,部署开发环境,了解并应用 Mule ESB 的某些技术如 JMS、JDBC、Web Service 等的具体使用,实现 Agilor 实时数据库和 Mule 的结合。此外还完成了项目文档的撰写以及对 Mule IDE 的汉化等工作。

1.3 论文结构

本文共分为五章,内容安排如下:

第一章 引言,介绍本课题的意义、任务、预期目标等。

第二章 介绍了 Mule ESB,如 ESB 的概念和功能等。

第三章 阐述了 Mule ESB 的架构,并简要介绍目前 Mule 已经支持的第三方产品以及一些技术。

第四章 Mule ESB 工作原理,具体介绍 Mule 的工作原理。

第五章 Mule ESB 开发详述,具体介绍本次课题所使用的 Mule 技术。

最后是结语,对系统和论文期间本人的工作做了总结,并对系统提出了改进建议。

第二章 Mule ESB 简介

2.1 ESB 的概念

ESB (Enterprise Service Bus, 即企业服务总线) 是传统中间件技术与 XML、Web 服务等技术结合的产物。ESB 提供了网络中最基本的连接中枢, 是构筑企业神经系统的必要元素。它支持 SOA 中的服务交互并对其进行管理。使服务交互在服务提供者和服务请求者之间进行, 并可以使用各种中间件技术和编程模型加以实现。使服务交互参与方不直接交互而是通过总线进行交互。它为 SOA 提供与企业需要保持一致的基础架构, 提供虚拟化和管理功能来实现和扩展 SOA 的核心定义。ESB 作为 SOA 的切入点, 使企业以最小投入将已有系统纳入到 SOA 架构中来^[7]。

2.2 ESB 的功能

2.3.1 ESB 的功能简单概括

ESB 的功能可以简单概括为这几个方面: 在服务与服务之间路由消息; 在请求者与服务者之间转换传输协议; 在请求者与服务者之间转换消息格式; 处理来自于各种异构源的业务事件。

2.3.2 ESB 的功能详细描述

ESB 在通信、服务交换、集成等方面都有应用, 它的主要功能^[8]如表 2-1 所示。

具体介绍如下:

- 通信: 路由、寻址、通信技术、协议和标准, 发布/订阅, 请求/响应, 事

件，同步和异步消息传递

- 服务交互：服务接口定义，支持替代服务实现，通信和集成所需的服务消息传递模型，服务目录和发现
- 集成：数据库，服务聚合，遗留系统和应用程序适配器，EAI 中间件的连接性，服务映射，协议转换，应用程序服务器环境，服务调用的语言接口
- 服务质量：事务(原子事务、补偿、WS-Transaction)，各种确定的传递范例(如 WS-ReliableMessaging 或对 EAI 中间件的支持)
- 安全性：身份验证，授权，不可抵赖性，机密性，安全标准(如 Kerberos、WS-Security)
- 服务级别：性能，吞吐量，可用性，其他可以构成契约或协定的持久评估方法
- 消息处理：编码的逻辑，基于内容的逻辑，消息和数据转换，有效性，中介，对象标识映射
- 管理和自治：服务预置和注册，记录、测量和监控，发现，系统管理和工具集成，自监控和自管理
- 建模：对象建模，通用业务对象建模，数据格式库，B2B 集成的公共与私有模型，开发和部署工具
- 基础架构智能：业务规则，策略驱动的行为，特别是对于服务级别、服务功能的安全和质量(WS-Policy)，模式识别

表 2-1 ESB 的主要应用领域

通信	服务交互
集成	服务质量
安全性	服务级别
消息处理	管理和自治
建模	基础架构智能

2.3.3 ESB 具体领域应用

大规模分布式的企业应用需要相对简单而实用的中间件技术来简化和统一

越来越复杂、繁琐的企业级信息系统平台。面向服务体系架构（SOA）是能够将应用程序的不同功能单元通过服务之间定义良好的接口和契约联系起来。SOA 使用户可以不受限制地重复使用软件、把各种资源互连起来，只要 IT 人员选用标准接口包装旧的应用程序、把新的应用程序构建成服务，那么其他应用系统就可以很方便的使用这些功能服务。

支撑 SOA 的关键是其消息传递架构-企业服务总线（ESB）。ESB 是传统中间件技术与 XML、Web 服务等技术相互结合的产物，用于实现企业应用不同消息和信息的准确、高效和安全传递。ESB 的出现改变了传统的软件架构，可以提供比传统中间件产品更为廉价的解决方案，同时它还可以消除不同应用之间的技术差异，让不同的应用服务协调运作，实现不同服务之间的通信与整合。ESB 在不同领域具有非常广泛的用途^[9]：

- 电信领域：ESB 能够在全方位支持电信行业 OSS 的应用整合概念。是理想的电信级应用软件承载平台。
- 电力领域：ESB 能够在全方位支持电力行业 EMS 的数据整合概念，是理想的 SCADA 系统数据交换平台。
- 金融领域：ESB 能够在全方位支持银企间业务处理平台的流程整合概念，是理想的 B2B 交易支撑平台。
- 电子政务：ESB 能够在全方位支持电子政务应用软件业务基础平台、信息共享交换平台、决策分析支撑平台和政务门户的平台化实现。

2.3 Mule ESB 关键特性

Mule ESB 主要有以下几个方面的特性：

- 基于 J2EE 1.4 的企业消息总线（Enterprise Service Bus (ESB)）和消息代理（broker）
- 可插入性连接，比如 JMS (1.0.2b 和 1.1) ^[10]，VM (嵌入)，JDBC，Tcp，Udp，Multicast，Http，Servlet，SMTP，Pop3，File，Xmpp 等
- 支持任何传输之上的异步，同步和请求响应事件处理机制
- 支持 Axis 或者 Glue 的 Web Service.

- 灵活的部署结构[Topologies]包括 Client/Server, P2P, ESB 和 Enterprise Service Network.
- 支持声明性和编程性事务, 包括 XA^[11] 支持
- 对事件的路由、传输和转换的端到端支持
- Spring 框架集成。可用作 ESB 容器, 而 Mule 也可以很容易的嵌入到 Spring 应用中。
- 使用基于 SEDA^[12] 处理模型的高度可伸缩的企业服务器
- 支持 REST API 来提供技术独立和语言中立的基于 web 的对 Mule 事件的访问
- 强大的基于 EIP 模式的事件路由机制
- 动态、声明性的, 基于内容和基于规则的路由选项
- 非入侵式的方式。任何对象都可以通过 ESB 容器管理
- 强大的应用集成框架
- 完整的可扩展的开发模式

2.4 何时使用 Mule ESB

一般在这些情形下使用 Mule ESB: 集成两个或者多个需要互相通信的或者多个现有的系统;需要完全和周围环境去耦合的应用, 或者需要在系统中伸缩不止一个组件的系统;开发人员不知道未来是否会将其应用分发或者伸缩需求的单 VM 应用。

2.5 ESB 技术与革新

由于更大任务所带来的要求, 消息传递技术现在正处于发展之中。为了给当今的实时企业提供其所需的灵活性, 就需要一种混合的消息传递模型将 Web 服务的优点与传统的异步消息传送结合在一起。

传统消息排队中间件将很快被企业服务总线(ESB)技术所取代, 从而将消息传递带到新的高度。新的 ESB 骨干(催生了下一代集成和应用平台产品)将显著改善多数企业的软件基础架构。行业正转向消息传递和 ESB, 并以此作为核心应用

平台基础架构模型，这将标志着一个转折点：围绕企业对其信息资源的使用而触发了新一轮巨大的革新浪潮；企业都正在利用事件架构。这都将消除最近人们对 IT 在战略性业务区分中可扮演关键角色的所有疑虑。

2.5.1 简介

在过去的 10 年中，竞争压力和日新月异的技术根本地改变了企业的运行节奏。在过去，企业可以根据月底的成批报告来进行决策。现在，实时流程意味着如果原材料在早上出现问题，或者有停电事故发生，那么就会造成下午无法交付和托运成品。于是，企业不得不以越来越快的速度应对突发事件——否则，它就要靠边站了。“零时延企业(zero latency enterprise)”的时代已经来临。

当今的企业环境正在一点一点地发展以应对这个挑战。异构存储、网络和硬件支持着“孤岛计算”(应用程序与数据相互孤立或者条块分割)，这导致环境的利用和管理都过度复杂，并使之变为资源密集型。对于企业所必须面对的大多数关键挑战而言，这种复杂性无疑是一种障碍，这些挑战包括：满足对利用多渠道传递大量信息服务的不断增长的需求；实时管理基础架构以满足不断变化的业务需求；使业务多样化以促进业务灵活地增长，并降低与固定产品线相关的经济风险；确保对客户、合作伙伴和雇员的信息服务请求做出快速且高质量的响应。

在过去几年中，EAI、B2B 和应用开发等方面的迅速发展推动了几种关键技术和标准的发展，这些技术和标准又推动了基础架构领域的显著进步：

XML 作为通用的、自解释的数据交换格式，已经为大多数应用程序所采用。面向 Web 的信息交换以及其后的基础架构，与 XML 一起使 Web 服务的使用成为不可避免的事情。Java 已经作为用于服务器端的一个主要技术而被接受，并且 J2EE 已经作为应用服务器的标准而被接受。企业服务总线在事务性消息交换和实时事件通知领域的使用已经围绕 Java 消息服务(JMS)而被标准化了。通过 Java 管理扩展(JMX)标准已经实现了服务器端组件的公共管理框架。基础架构必须像业务一样运转。

瞬息万变的市场需要通过多渠道传递大量的信息服务。下一代的企业要求松散耦合的资源能够共享跨越多领域的公共通信和管理基础架构。企业基础架构不得不像有形的业务那样运转，允许对资源进行动态管理以应对客户和合作伙伴的

需求波动，同时处理系统资源的供应和可用性变化。企业应用程序也需要一个基于标准的协作模型以最大程度地利用该基础架构。为此，实时企业使用了来自实时基础架构的最好做法和服务端端的网格技术(gridtEChnology)。

2.5.2 实时企业的组件

形成实时企业的一些概念与用于定义服务器端网格环境的概念相同，用来描述其核心组件的结构类似于 Gartner 的 5 层网络技术模型，如表 2-2 所示：

表 2-2 实时企业模型

第四层	管理支付	手动和自动化管理
第三层	应用程序	分布式处理和服务
第二层	分布式编程模式	服务和分布式处理模型
第一层	虚拟操作系统	分布式操作系统
第零层	基础架构资源	网络、操作系统和存储

一个建立在现有的而且是被广泛采用的技术和开放标准之上的 ESB 可为服务协作、管理和控制提供一个可适应的分布式架构。ESB 支持在企业内部的任何地方进行业务服务的运行时部署，并提供协作和通知服务作为其核心基础架构的一部分。让我们看一下 ESB 技术是如何映射到 Gartner 的 5 层模型的。

2.5.3 基础架构资源和虚拟操作系统

第 0 层由基础架构资源组成，包括网络、服务器、存储和每台服务器的操作系统环境。第 1 层位于基础设施层之上，并建立了一个多资源的分布式操作系统，它支持的功能如进行工作计划、将资源名集成到总体结构中以及确保不同系统间的一致认证。尽管 Gartner 将 J2EE 作为一个第 2 层的技术，我们相信分布式 JMX 和一台基于 J2EE 的应用服务器的结合会具有虚拟操作系统的特点。使用对所有组件和服务提供部署和完全 JMX 管理的容器或者微内核，从而允许对服务进行远程激活和管理。JMX 作为一种技术最初设计用于管理单个代理，如一台应用服务器。JMX 通过与 JMS 的结合，其范围就可以扩展到管理单个代理、群集或松耦合的联合体(如果您喜欢，亦可称之为超级群集)，允许对联合的 ESB 基础架构进行全生命周期和部署管理。由于 JMX 同时也集成了许多传统

的管理协议，如 SNMP，因此 ESB 基础架构可以为 Java、Web 服务和传统平台提供按需应变的(on-DEMAND)的热部署和自我修复(self-annealing)式的基础架构。

2.5.4 分布式编程模型

分布式编程模型构成了实时企业的第 1 层：可在应用程序和服务(无论是内部还是外部)之间进行协作和通知的核心基础架构。ESB 提供事件通知、动态路由选择和事务性确保传递；并且使用一个定义明确的过程语言以使应用程序通过一个公共 API 进行活动协调。

实时企业要求在恰当的时间将正确的数据传递到正确的位置；JMS(Java 消息服务)提供事件分布和事务性确保传递的方法。同时也需要智能数据结构(datafabric)，它可以在需要的时候在网络范围内进行信息分发，目的是提高吞吐量和降低宝贵的后台系统的负载。该结构的骨干是通过 JCache (Java 通用缓冲框架)所形成的。

一个类似于 Linda 的元组空间(tuplespace)将消息队列的“一个且只有一个”传递语义与发布/订阅的广播功能和对等系统的松耦合结合到一起。元组空间就如同由无限数目的进程所共享的相连内存。进程可以向该空间中添加元组(本质上就是数据对象)，或者从中取出元组来以独占的方式工作——如果需要的话，可以一直处于等待状态，直到匹配对象的出现。进程也可以读取元组而不需要将其从空间中删除。该范例(将消息队列的“一个且只有一个”传递语义与发布/订阅的广播功能和对等系统的松耦合结合到一起)被映射到 Jcache 的顶部，它提供一个该概念的高性能分布式实现。

这也可以和一个业务流程模型引擎(例如，jBPM: www.jbpm.org)结合起来以提供一套丰富的分布式编程域。进程之间独立工作——从元组空间那里获得适当的输入，并将输出放回元组空间以便进行后续任务。进程在元组上的执行顺序比在传统工作流系统上的执行顺序所受到的约束要少。该模型提供分布式共享内存、通用的群集、并行计算以及分布式工作流和 BPM 的基础。

2.5.5 应用程序

构成实时企业第 3 层的应用程序依赖于企业基础架构的资源, 以及使用协作编程模型进行相互通信。架构师们已经意识到更松散耦合的和多层组件模型的优越性, 而不是开发独立的或简单的两层客户/服务器(C/S)应用程序。为定义、发现和实际执行该模型(例如 WSDL、UDDI 以及用于 Web 服务的 SOAP)而采用的标准有助于面向服务架构的实现。

作为虚拟操作系统基础的 J2EE 应用服务器为基础架构提供了一个基于事务性的安全服务的集成点。由于分布式 ESB 是一个像网格一样的使能技术, 所以基于 OGSI 源代码组织定义的 Web 服务接口是一个很自然的选择。OGSI 当前是外化网格技术的事实标准, 它允许在一个环境下所书写的网格服务可以很容易地部署在其他环境中。

此外, ESB 可以提供一个基于优化 Tete 算法的可扩展规则引擎。外化业务规则使得在较低层对迅速变化的业务流程、决策机制进行管理以及使消息过滤和路由选择变得可能, 而不需要对基础应用程序进行代码级的改变。它将业务从对缓慢的代码开发周期的依赖中解放出来, 允许精通业务的分析师进行必要的变化以支持新产品或法规需求的引进, 而无需中断系统的运行。

2.5.6 管理与支持

实时企业要求服务在宏观和微观两个层面上管理和协调应用程序及其服务。第 4 层提供了实现安全策略、定义资源使用指南和集成操作流程所需的管理支持。基本功能包括:

- 监视: 整理事件和统计数据以了解应用程序的性能、资源使用情况和操作行为。它允许对整个基础架构进行模拟、错误判定以及对资源利用进行手动和自动平衡。
- 反应协调: 需要通过启发分析、动态规则和灵活的工作流对应用程序进行智能管理、控制、自我修复以及微调。通过使用有效的动态拓扑布局(在正确的位置运行正确数量的应用程序)、实时企业管理利用负载, 并选择正确的硬件和位置来运行应用程序。

ESB 管理结构将分布式 JMX 与统计事件收集和对比与用在应用级的基于相

同标准的 Java 规范框架进行了结合。这为资源使用、性能监视和警告通知提供了位置透明性、发现、远程控制和统计数据的整理。这些技术允许对在整个实时企业中的智能性资源可视化、协同合作和供应环境进行预言性的决策,从而为 IT 经理和业务经理赋予了洞察力。

2.5.7 总结

使用一个分布式企业服务总线,企业可以通过利用标准以提供灵活实时的“按需服务”基础架构来最大化利用其在硬件和软件的现有投资。该灵活的基础架构包括:提供可主动调整 IT 资源的技术,使业务领导可以转变核心信息服务以满足不断变化的市场;创建一个基于开放标准的统一 IT 基础,它可灵活变化以满足未来的需求;降低 IT 基础架构的成本,同时保持高水平的性能。

这些目标是通过企业消息传递、实时缓冲以及分布式主动管理技术的大量结合而实现的。结果就是一个具有更低总体成本和具有更高应对业务变化能力的 IT 基础。通过依赖于标准,实时基础架构将不同的技术结合到一个连续的结构中,这个结构提供了快速调整软件和硬件基础架构以满足企业实时业务需求的方法。

2.6 ESB 应用前景

企业级应用系统一直是中国软件产业发展的主要方向之一,占有至关重要的地位。同时,它也受到整个世界 IT 发展潮流的影响,当前 IT 软件领域的主要技术趋势是 SOA 和 ESB,原因是信息技术的不断发展和成熟使各个企业有机会在更大的范围内整合自己的资源,提高经营运行效率。

二十一世纪信息共享与整合对企业的变革发展日趋重要,而企业对网络环境的依赖及应用创新的追求,将是我们面临的主要挑战。

第三章 Mule ESB 的架构

3.1 ESB 架构

ESB 提供了一种开放的、基于标准的消息机制，通过简单的标准适配器和接口，来完成粗粒度应用（服务）和其他组件之间的互操作，能够满足大型异构企业环境的集成需求。它可以在不改变现有基础结构的情况下让几代技术实现互操作。

通过使用 ESB，可以在几乎不更改代码的情况下，以一种无缝的非侵入方式使企业已有的系统具有全新的服务接口，并能够在部署环境中支持任何标准。更重要的是，充当“缓冲器”的 ESB（负责在诸多服务之间转换业务逻辑和数据格式）与服务逻辑相分离，从而使得不同的应用程序可以同时使用同一服务，用不着在应用程序或者数据发生变化时，改动服务代码。它的架构模型如图 3-1 所示：

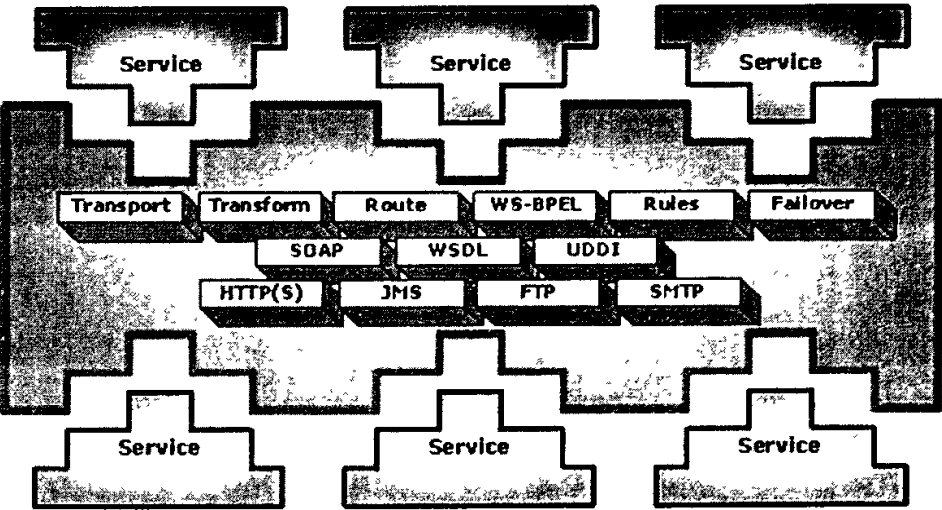


图 3-1 ESB 的架构模型

3.2 Mule ESB 架构

Mule 是一个轻量级的分布式消息框架，通过不同的协议与各种服务和应用

程序通信。其关键特性如下所示：插件式的连接组件支持 JMS、JDBC、TCP、UDP、Multicast、Http、SMTP、POP3、File 等传输协议；支持异步通信、同步通信和请求—应答式的事件处理方式；对标准 WebService 规范的支持；支持强大的应用集成的能力。

其技术架构如图 3-2 所示：

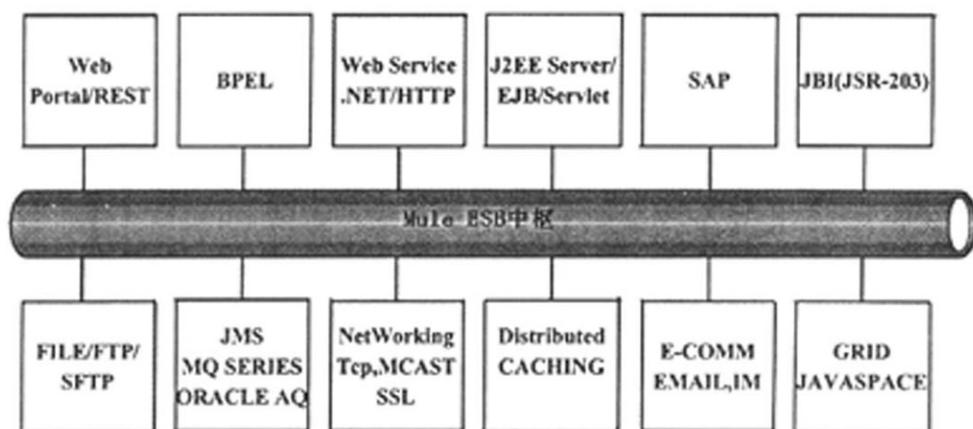


图 3-2 Mule ESB 架构

Mule ESB 是一个消息 ESB 框架，一个消息代理，一个分级事件驱动的框架（SEDA）。SEDA 定义了一个依照分级队列、高度并行的企业级平台。Mule ESB 使用 SED 的概念增加事件处理的性能。

事实上，Mule ESB 超出了传统意义上的 ESB，我们更愿意把它定义为一个轻量级的消息框架，它包含一个在不同系统之间分配信息的对象经纪人。它的目的是管理消息组件，这些组件通常称为 UMO(Universal Message Objects: 通用消息对象)或 UMOs，它们可以共存在同一个 VM 中或者分散在你的网络中。UMOs 和其它应用软件系统之间的通信都是通过 endpoints 来进行的。这些终端为许多不同的技术，例如 Jms, Smtpp, Jdbc, Http, Xmpp, File 等等提供了简单而稳定的接口。

Mule ESB 支持同步、异步的请求响应事件，事件处理和传输实用不同的技术例如 JMS, HTTP, 电子邮件和基于 XML 的 RPC。Mule ESB 能很容易地嵌入到任何应用框架中，明确支持 Spring 框架。Mule ESB 也支持动态的，预定义的，基于内容的和基于规则的消息路由。Mule ESB 使得预定义的和计划性的事务更容易，包括 XA 事务支持。Mule ESB 提供一个有代表性的状态调用（REST）

API 提供给与 Web 的事件访问。

Mule ESB 模式驱动系统中所有服务，这个系统有着一个分离的消息通讯中枢。服务注册在总线上，但不知道其他任何被注册的消息；因此，每个服务只关心处理它收到的事件。Mule ESB 也把容器，传输，转换细节从服务中分离出来，允许任何对象作为服务注册到总线的。

Mule ESB 主要包含下列内容：

- 通用消息对象(UMO)API：定义了所有被 Mule ESB 管理的服务和对象交互。
- 通用消息对象(UMO)组件：在 Mule 系统中，UMO 组件可以是任何在系统中接收、处理和发送事件消息的组件。
- Mule 服务器：一个在 Mule 应用环境中自动加载的服务器应用程序。
- 描述器：描述器组件描述一个 Mule ESB UMO 属性。新的 Mule ESB MUO 对象能被它们所关联的描述器初始化。一个描述器包含： UMO 组件名，UMO 组件版本，UMO 组件实现类，异常策略，入站和出站提供者，入站和出站路由器，拦截器，接收和发送切入点，入站和出站转换器，各种各样的特性
- 连接器：连接器是一些组件，它们可以连接到外部系统或其他协议、管理那些系统或协议的状态。一个连接器负责发送消息到外部消息接收器、管理消息接收器的注册和注销。
- 提供者：提供者是一些组件，管理把事件数据发送到外部系统、从外部系统接受事件数据和转换事件数据等事项。在 Mule ESB 框架里，他们能连接到外部系统或其他组件。一个提供者就像一个从外部系统进入 Mule ESB 或从 Mule ESB 内部访问外部系统的桥接器。实际上，提供者有一组对象组成，可以与下层系统连接并与之通信。提供者的组成部件有：连接器：负责连接到下层系统；消息接收器：从系统接收事件；连接调度者：传送系统到系统；转换器：转换从系统接收到的或要发送到系统的数据；终端：所建立连接的通道地址；事务配制：定义连接的事务属性
- 终端调解者：当 UMO 组件接收到一个事件时，终端调解者决定去调用

它的什么方法。

- **转换器：**转换器组件负责双向转换消息或事件的有效载荷。当一个事件到达接收的对象之前，转换器可以链接到一起去执行一系列的装换操作。
- **消息适配器：**消息适配器提供一种公共的方式去读外部系统的异构数据。
- **消息接收器：**消息接收器是一系列终端监听线程，负责从外部系统接收数据。
- **消息调度者：**消息调度者发送（同步）或派遣（异步）事件到下层系统。
- **消息路由器：**消息路由器是一系列组件，可以使被配制的 UMO 组件依据消息或其他配制路由一个消息到不同的提供者。
- **代理：**代理是一些绑定到外部服务的组件，例如 JME 服务器。
- **Mule 模型：**一个 Mule ESB 模型封装和管理一个 Mule ESB 服务器实例运行时的行为。一个模型包含：描述器，UMO 组件，一个终端调解者，一个生命周期适配器工厂，一个组件调解者，一个池化工厂，一个异常策略
- **Mule 管理器：**Mule ESB 管理器，如图 3-3 所示，它维护和提供以下服务：代理提供者，连接器，终端，转换器，拦截器堆栈，一个 Mule ESB 模型，一个 Mule ESB 服务器，事务管理器，应用程序属性，Mule ESB 配制

3.3 消息交换模型

ESB 是一种体系结构模式，支持在面向服务的体系结构中虚拟化服务交互并对其进行管理。

ESB 使交互可以在服务提供者和服务请求者之间进行，并且可以使用各种中间件技术和编程模型加以实现。

在 ESB 模式中，服务交互的参与方并不直接交互，而是通过一个总线交互，该总线提供虚拟化和管理功能来实现和扩展 SOA 的核心定义。

ESB 的消息交换模型有：单向(one-way messages)，请求/响应(Request and

response), 中转/传送(Brokered and delivery), 群发(hub and spoke) ,发布/订阅(publish and subscribe), 同步(synchronous), 异步(asynchronous), 复杂事件(orchestrated workflow), 其他。

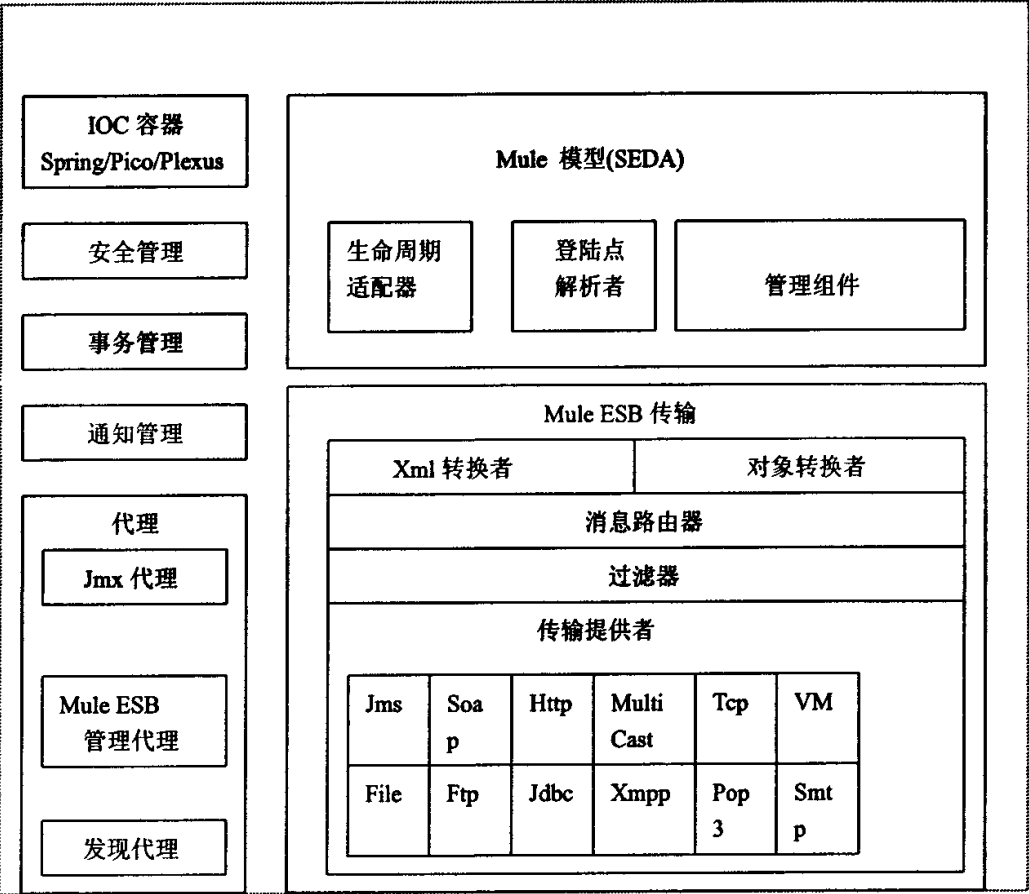


图 3-3 Mule ESB 管理器

ESB 交换模式的虚拟化有以下几个特性：

- 位置和标识：参与方不需要知道其他参与方的位置或标识。
- 交互协议：参与方不需要采用相同的通信协议或交互方式。
- 接口：请求者和提供者不需要就公共接口达成协议。
- 服务质量：参与方声明其 QoS 要求，包括性能和可靠性、请求的授权、消息内容的加密/解密、服务交互的自动审核以及如何对请求进行路由。

ESB 的基本交换模式如图 3-4 所示：

它主要特性有：

- 参与方调用其他参与方提供的服务

- 其他参与方则会向感兴趣的使用者发布信息
- 端点与 ESB 交互的位置称为服务交互点 SIP
- 服务注册将捕获描述以下内容的元数据: SIP 的要求和功能(如提供或需要的接口), SIP 希望与其他 SIP 的交互方式(如同步或异步), SIP 的 QoS 要求(如安全、可靠交互), SIP 支持与其他 SIP 交互的其他信息(如语义注释)

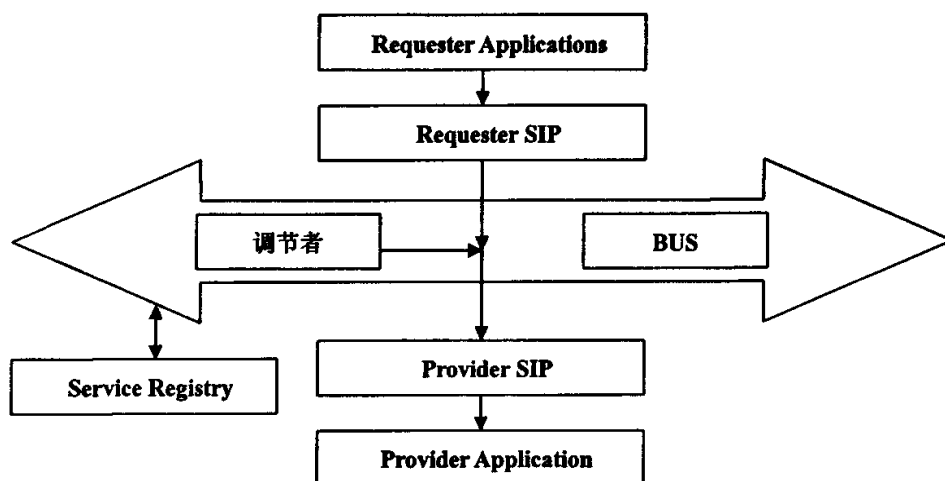


图 3-4 ESB 基本交换模式图

ESB 交换模式有下面几类:

1. 交互模式

如图 3-5 所示, 服务交互点将消息发送到总线或从总线接收消息, 服务交互点与总线进行交互。方式有三种: 请求/响应、请求/多响应和事件传播

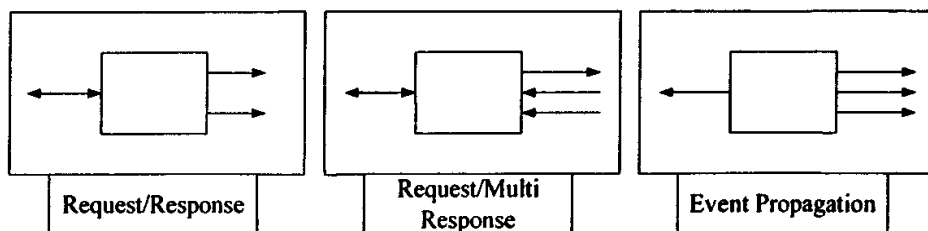


图 3-5 交互模式

2. 中介模式

如图 3-6 所示，处理总线上的消息。由请求者发出的消息转换为提供者能够理解的消息。方式有：协议变换，转换，充实，路由，分发，监视，关联

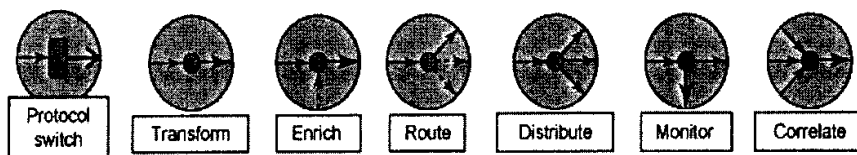


图 3-6 中介模式

3. 复杂模式

如图 3-7 所示，中介模式和交互模式可以进行组合，以实现更为复杂的模式。方式有：规范化适配器(协议变换后转换格式,将消息标准化为规范的格式)，转换、记录和路由，网关

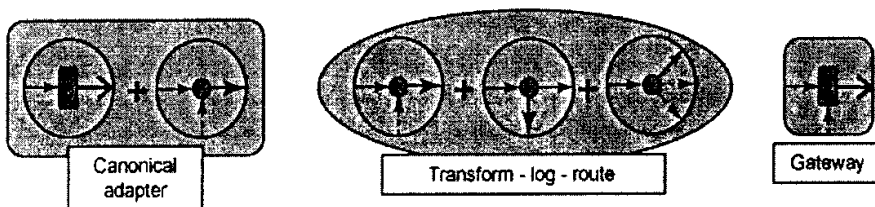


图 3-7 复杂模式

4. 部署模式

如图 3-8 所示，提供多种 ESB 拓扑：全局 ESB, 直接连接 ESB, 代理 ESB, 联合 ESB

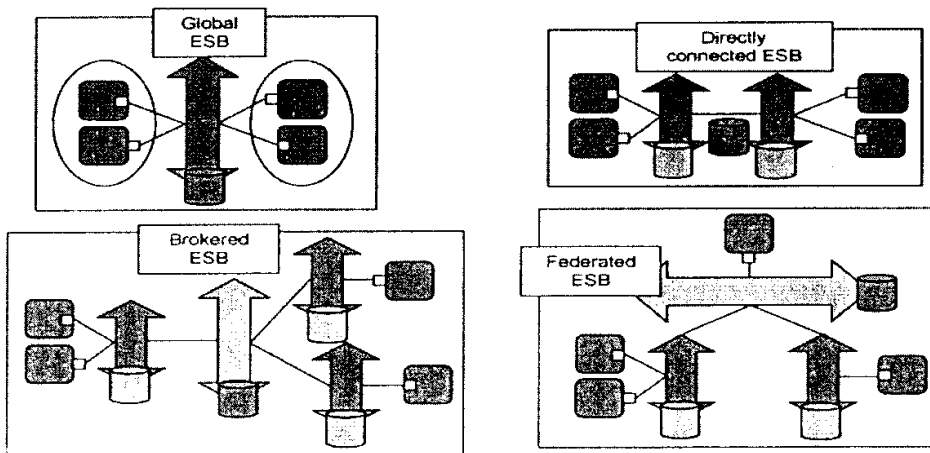


图 3-8 部署模式

下面我们对 ESB 交换模式进行总结：

- ESB 模式扩展了 SOA 的虚拟化功能。可以由标准功能单元组成中介，然后进行部署，以帮助不匹配的请求者和提供者进行交互。
- ESB 提供了用于部署和管理服务的通用模型。
- ESB 综合编程模型、组件化工具以及基础设施极大地支持了 SOA 原则的实现

3.4 Mule ESB 对第三方的支持

为了更加便于开发，Mule 已经将许多技术融入其中。目前已经融入的产品或技术主要有以下这些：

➤ Email 提供者(Email Provider)

用来和 pop3 和 Imap(消息访问协议)邮箱进行连接并利用 javax.mail API 通过 SMTP 协议发送事件。对于每个协议都提供了 SSL 机制。Email 连接者可以通过 Mail 的所有协议以及其它相关的协议发送和接收事件的有效载荷。

➤ Servlet 提供者(Servlet Provider)

Servlet 连接者只面对一个 Servlet 执行，该执行主要是接收请求，然后连接者将请求传送给任何合法的接收者。这个连接者有分发器的概念，它可以被一个请求触发，但有可能不返回响应。

➤ Xmpp 提供者(Xmpp Provider)

利用 Smack API 通过 Xmpp(jabber)实例消息协议来激活它。

➤ WSDL 提供者(WSDL Provider)

通过获取服务的 wsdl 来调用远程的 web 服务。Mule ESB 会为服务创建一个动态的代理并调用它。

➤ Udp 提供者(Udp Provider)

Udp 连接者可以以自带寻址信息的、独立地从数据源行走到终点的数据包的形式发送和接收事件。

➤ Multicast 提供者(Multicast Provider)

Multicast(多点传送)连接者通过多点传送组件发送和接收事件。

➤ SSL 提供者(SSL Provider)

SSL 连接者利用 SSL 或者 TLS 进行可靠安全的 socket 通信。

➤ **Stream 提供者(Stream Provider)**

允许读写 streaming 数据。这个类似于 java 的 System.out 和 System.in。

➤ **Soap 提供者(Soap Provider)**

通过 Apache 的 Axis 或者 WebMethods 的 Glue 来发布和调用服务。

➤ **Tcp 提供者(Tcp Provider)**

通过 Tcp 协议发送和接收事件。

➤ **EJB 提供者(EJB Provider)**

EJB 连接者允许 EJB 状态 beans 作为事件流的一部分被调用。通过给组件一个 EJB 输出的端点，它就可以调用远程对象和返回一个结果。

➤ **JMS 提供者(JMS Provider)**

这是标准的 Jms1.0.2b 和 1.1 连接者。这个连接者展示了 1.0.2b/1.1 规范的所有特征。

➤ **IMAP 提供者(IMAP Provider)**

IMAP(消息访问协议)传输提供者接收来自 IMAP 收件箱的消息并可以利用 SSL(IMAPs)和 IMAP 邮箱进行连接。

➤ **Pop3 提供者(Pop3 Provider)**

Pop3(邮局协议 3)传输提供者接收来自 Pop3 收件箱的消息并可以利用 SSL(IMAPs)和 Pop3 邮箱进行连接。

➤ **File 提供者(File Provider)**

通过文件连接器可以读写本地文件系统。可以通过配置文件连接器对读写的文件数据进行过滤。

➤ **FTP 提供者(FTP Provider)**

通过 Ftp 连接器可以读写远程 Ftp 服务器上的文件。

➤ **HTTP 提供者(HTTP Provider)**

Http 连接者器通过 http 协议发送和接收事件。它还含有一个 https 连接器用来进行安全可靠的 http 通信。

➤ **Quartz 提供者(Quartz Provider)**

提供时序安排特征给 Mule ESB。它基于 OpenSymphony 的 Quartz 调度程序。

➤ **VM 提供者(VM Provider)**

通过 VM 连接器，组建之间可以进行内部的 VM 通信。它还提供了 VM 短暂的或是长久的队列可选配置。

➤ **SMTP 提供者(SMTP Provider)**

通过 SMTP 和 SMTPs(安全的)协议发送消息。

➤ **JDBC 提供者(JDBC Provider)**

通过 JDBC 连接关系数据库。支持简单表的读写操作。

➤ **AS400DQ 提供者(AS400DQ Provider)**

用来和 IBM AS400 DQ 消息服务器进行连接。

➤ **RMI 提供者(RMI Provider)**

通过 `jmp` 发送和接收事件(目前只能进行分发操作)。

➤ **VFS 提供者(VFS Provider)**

使有权使用多种协议例如 WebDav, Samba, Jar/Zip 等等。目前只在 Sandbox 里面可用得到。

第四章 Mule ESB 工作原理

Mule 应用通常是由网络中的许多 Mule 实例组成。每一个实例都是一个驻留一个或者多个 UMO 组件的轻量级容器。每一个 UMO 组件都有一个或者多个通过它（们）发送和接收事件的端点。如图 4-1 所示：

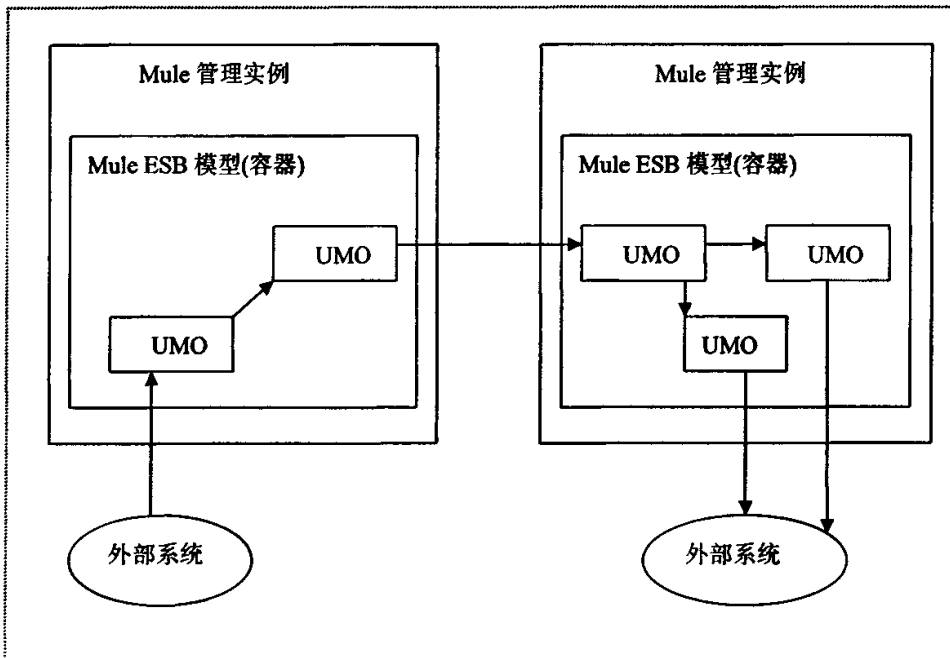


图 4-1 Mule ESB 应用实例

容器通过 UMO 组件提供各种各样的服务，比如事务管理、事件转换，路由，事件关联、日志、审计和管理等等。Mule 将对象构造从管理手段中分离出来，通常流行框架和 IoC/DI 容器，如 Spring, PicoContainer 或者 Plexus 可用这种管理手段来构建你的 UMO 组件。

下面我们通过消息的传递过程，如图 4-2 所示来进一步了解 Mule ESB 的工作过程：

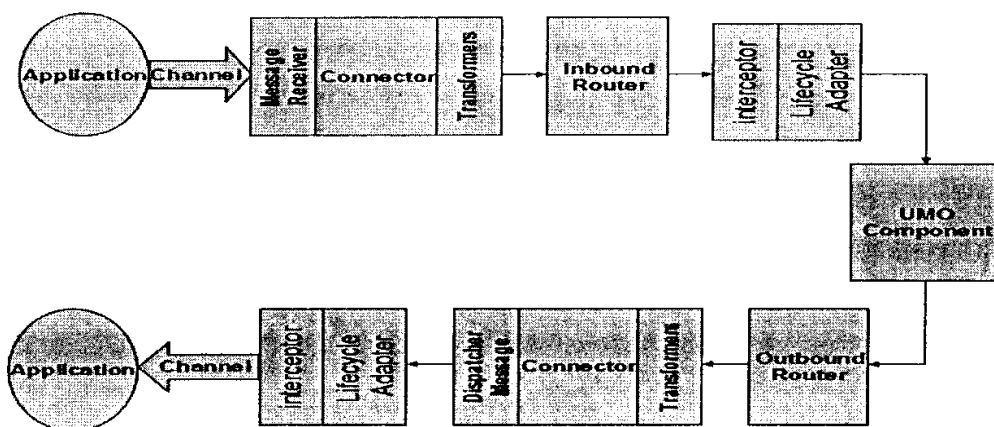


图 4-2 Mule ESB 消息流模型

外部应用(Appilcation)可以是任何应用，从应用服务器到遗留的传统应用，主机程序，或者 C/S 系统。基本上是可以产生和操纵数据的应用。因为 Mule 通过 endpoints 执行所有通信，UMO 组件并不打算在其中包含应用产生数据，以及应用主流，以及使用传输协议的部分。

消息通道(Channel) 是消息系统中的逻辑地址。Mule ESB 支持可供选择的通道，那就是传输提供者，如 jdbc, jms 等等。

4.1 传输提供者(Provider)

它主要包含以下几个要素：Connector：用来连接底层的资源；Message Receiver：从系统接收事件；Connector Dispatchers：给系统传递数据；Transformers：用来转换输入和输出的数据

➤ 消息连接器(Connector)

它的主要功能是：通过通道接收和传送数据；一个消息接收器和一个连接器以及一个由传输提供者组成的转换器紧密结合；连接器主要作用是发送数据给一个资源和管理连接器的监听器以便从资源处接收数据。

➤ 消息接收器(MessageReceiver)

它的主要功能是：从应用系统处接收数据并支持多种传输提供者。

➤ 消息调度者(Dispatcher)

它是一个接口，主要用来分发事件给底层支持的协议。该接口定义了三个重要的方法：Dispatch()：异步发送数据给外部系统；Send()：同步发送数据给外部

系统并返回响应信息；Receive()：发送请求给底层所支持的技术并返回得到的结果。这个方法有可配置的响应时间。

➤ 消息转换器(Transformers)

它的主要功能是：用来将接收到的原数据转换成 UMO 组件需要的数据对象；在端点处进行配置以保证 UMO 组件能得到它所需要的数据对象；在输出端点处进行配置可以确保在分发事件前端点所接收到的数据对象是正确的；可以在同一个端点配置多个转换者，这需要在两个转换者之间加入空格就可以了。

4.2 路由(Router)

路由的类型有三种：输入：控制如何接收哪个事件，过滤，再序列化事件；输出：控制哪个组件如何接收分发的事件；响应。

它主要有下面几个作用：控制系统中的组件怎么发送和接收事件；当事件被接收时调用输入路由(Inbound Router)；当事件被分发时调用输出路由(OutBound Router)；当注册事件被接收时响应路由被调用。

4.3 Mule 服务器端

Mule 服务器端主要有：

➤ 端点(Endpoints)

一个逻辑的，可配置的实体，它绑定在组件或外部网络资源上。用来控制事件的发送者和接收者。它有下面几个属性：Endpoint URI：表示一个资源提供者或者本地或远程接收者的地址。它必须是合法的 URI；Connector：用来连接底层所支持的传输；Filter：对端点接收到的数据进行过滤；Transaction：当一个事件发送或被接收时可以一个事务就可以开始执行；Properties：根据不同端点实例的连接者的需求不同可以对需要的特性进行重新设置。例如，当使用 SMTP 端点的时候你可能会需要重新设定来源地址。

下面是一些端点的设置举例：

- POP3/SMTP: Pop3://user:password@mail.mycompany.com
Smtplib:// user:password@mail.mycompany.com
- JMS/Queue: Jms://topic:mytopic

- Http: <http://mycompany.com/esb>
- VM: Vm://myUMO
- SOAP: Axis:http://mycompany.com/esb

➤ 通用消息对象组件(UMO Component)

UMO 代表 Universal Message Object; 它是一个可以接收来自于任何地方的消息的对象。UMO 组件就是你的业务对象。它们是执行引入的事件之上的具体业务逻辑的组件。这些组件是标准的 JavaBean, 组件中并没有任何 Mule ESB 特定的代码。Mule ESB 基于你的组件的配置处理所有进出组件的事件的路由和转换。

➤ Mule 管理器(Mule ESB Manager)

Mule ESB Manager 是 Mule ESB server 实例的中心(也称为一个节点户或者 Mule ESB Node)。其主要的角色是管理各种对象, 比如 Mule ESB 实例的连接器、端点和转换器。这些对象然后被用来控制进出你的服务组件的消息流, 并且为 Model 和它所管理的组件提供服务。

➤ Mule 模型(Mule ESB Model)

Model 是管理和执行组件的容器。它控制进出组件的消息流, 管理线程、生命周期和缓冲池。默认的 Mule ESB Model 是基于 SEDA 的, 它使用一个有效的基于事件的队列模型来获取的最大的性能和直通性。

➤ Mule ESB 服务器通知(Mule Notification Manager)

Mule ESB 有一个内置的消息机制, 通过它可以接进服务器通知信息, 例如添加组件, 初始化模型或者启动管理者的通知消息。

目前有 10 种服务器通知信息: 管理者通知: Mule ESB 管理器状态发生变化时产生, 如启动, 停止等; 模型通知: Mule ESB 模型状态发生变化时产生, 如初始化, 启动等; 组件通知: 特殊组件启动或停止等状态变化时产生; 管理通知 (Management): 监控的资源还为实现时产生; 定制通知: 在对象本身定制消息监听器时产生或用来在代理、组件连接器等等上面定制消息; 管理通知(Admin): 当请求被 Mule ESB 接收到时产生; 连接通知: 连接着尝试连接它的资源时产生; 消息通知: 系统发送或接收到事件时产生; 安全通知: 请求没有通过安全认证时产生; 空间监控通知: 由空间执行如 JavaSpaces 等产生。

➤ **Mule ESB 安全管理(Mule Security Manager)**

Mule 安全管理有三种方式：端点利用传输的特定细节或普通的认证方法对请求信息进行认证、在 Mule ESB 管理者处定义安全管理者和配置安全过滤器。

➤ **Mule ESB 事务(Mule Transaction Manager)**

Mule 事务有两个特性：对底层事务管理者来说是不可知的；在输入端点处进行配置。

➤ **Mule ESB 事件(Mule Event)**

Mule ESB 对事件提供三种响应方式：异步：许多事件可以在同一时间被同一组件以不同的线程进行处理；同步：所有的事件在同一个执行线程中执行；请求-响应：发送的请求必须在事先定好的时间内响应。

4.4 Mule ESB 客户端(Mule Client)

为 java 客户端提供的简单接口，通过它可以从 Mule ESB 服务器端或其它应用系统处接收和发送事件。它提供以下几种功能：发送或接收事件给本地的服务器；发送或接收事件给远程的服务器；利用 Mule ESB 提供的传输和其它应用系统进行通信；在 Mule ESB 服务器端注册或注销组件

第五章 Mule ESB 开发详述

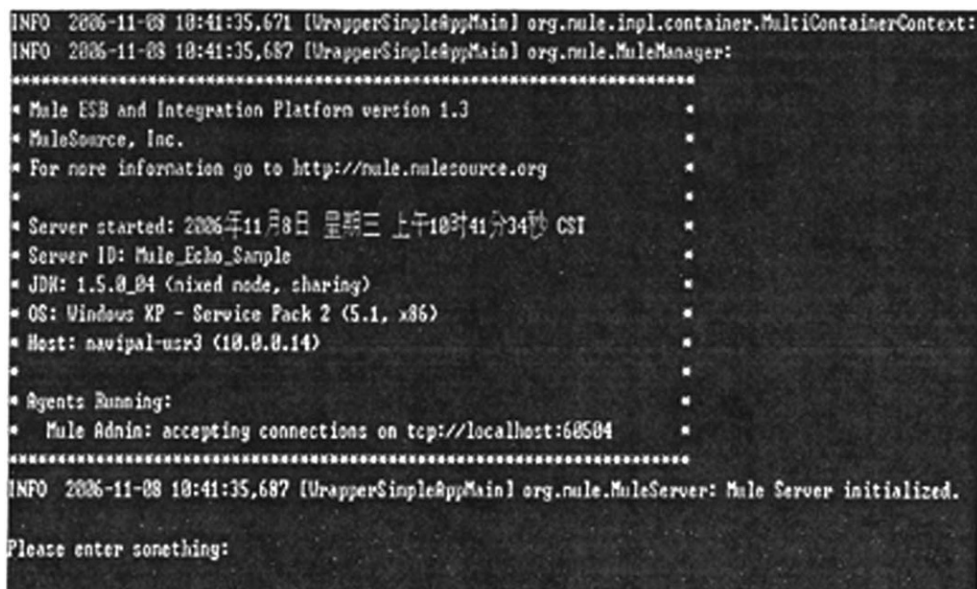
这一章主要讲述了本次项目中学习 Mule ESB 的过程以及如何使用 Mule ESB 所提供的一些技术，如 JDBC Provider、JMS Provider、Soap Provider 等等，以及如何将第三方产品融入到 Mule ESB 中。

5.1 Mule ESB 初步了解

目前 Mule 的版本为 1.3，下载后，解压缩，并设置 MULE_HOME 为解压缩目录。由于有些例子需要用到 ant 或者 maven，所以也先下载这两软件并设置相应环境变量。Mule 总共提供了 9 个例子，下面首先将对其中的几个例子进行分析，其中将重点讲 loanbroker-esb 例子，以便加深对 Mule 的了解。

5.1.1 Echo 例子

Mule 提供的 echo 例子程序是个比较好的测试工具，放在 %MULE_HOME%\examples\ant\echo 下，双击 echo.bat，即可看到如图 5-1 所示：



```
INFO 2006-11-08 10:41:35.671 [WrapperSimpleAppMain] org.mule.inpl.container.MultiContainerContext:
INFO 2006-11-08 10:41:35.687 [WrapperSimpleAppMain] org.mule.MuleManager:
*****
* Mule ESB and Integration Platform version 1.3 *
* MuleSource, Inc. *
* For more information go to http://mule.mulesource.org *
* *
* Server started: 2006年11月8日 星期三 上午10时41分34秒 CST *
* Server ID: Mule_Echo_Sample *
* JDN: 1.5.0_04 (mixed mode, charing) *
* OS: Windows XP - Service Pack 2 (5.1, x86) *
* Host: navipal-usr3 (10.0.0.14) *
* *
* Agents Running: *
* Mule Admin: accepting connections on tcp://localhost:60504 *
*****
INFO 2006-11-08 10:41:35.687 [WrapperSimpleAppMain] org.mule.MuleServer: Mule Server initialized.
Please enter something:
```

图 5-1 echo 例子运行图

接下来按照提示一步一步往下走即可。该例子的流程如图 5-2 所示，

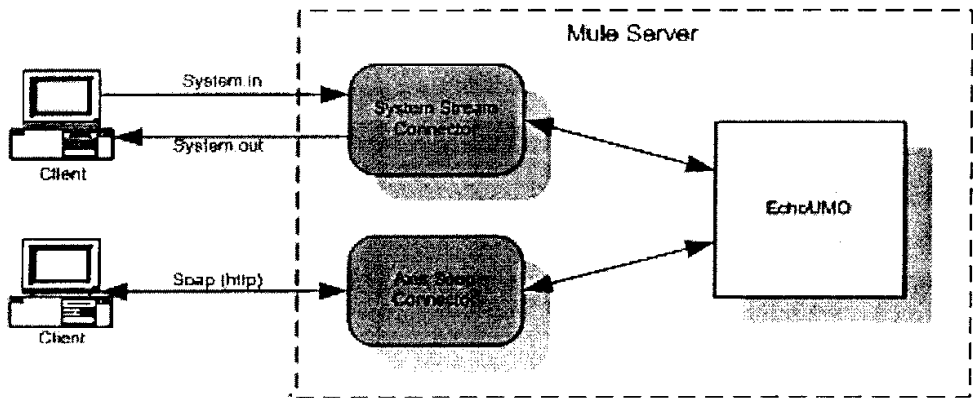


图 5-2 echo 例子流程图

描述为：

1. 通过两种方式接入（红色标识）：一为 System.in，另一为 Soap(http)方式
 2. 每种接入都可以通过接入器(Connector)经过 NMR 路由转发后，进行输出，目前通过两种方式输出：一为 System.out，另一为 Soap(http)方式
- Echo 例子的 Mule 配置文件请参看附录 1：echo 配置文件。

5.1.2 Loanbroker-esb 例子

这是一个现实生活中的例子，首先我们先了解现实生活中贷款的流程，如图 5-3 所示：

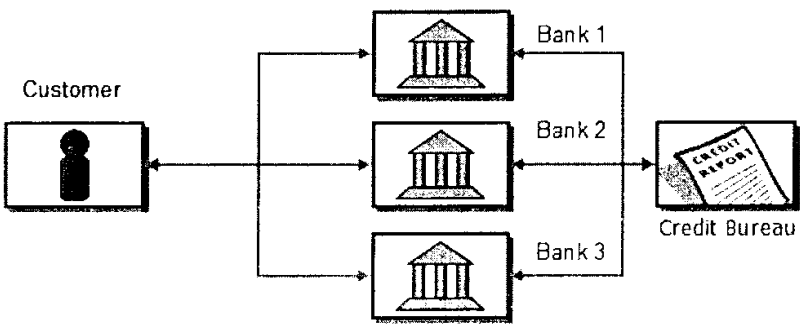


图 5-3 贷款过程图示

详细过程如下：

1. 客户根据不同银行的贷款利率选择贷款利率最低的银行
2. 每个银行会索取客户的姓名，社会安全号码，贷款的金额以及还贷的期限

3. 根据客户提供的信息每个银行会调查客户的信用背景，这通常通过和征信所联系获得
4. 银行根据得到的信息返回客户相应的贷款报价
5. 客户根据银行的报价选择最低报价者

该例子的流程如图 5-4 所示：

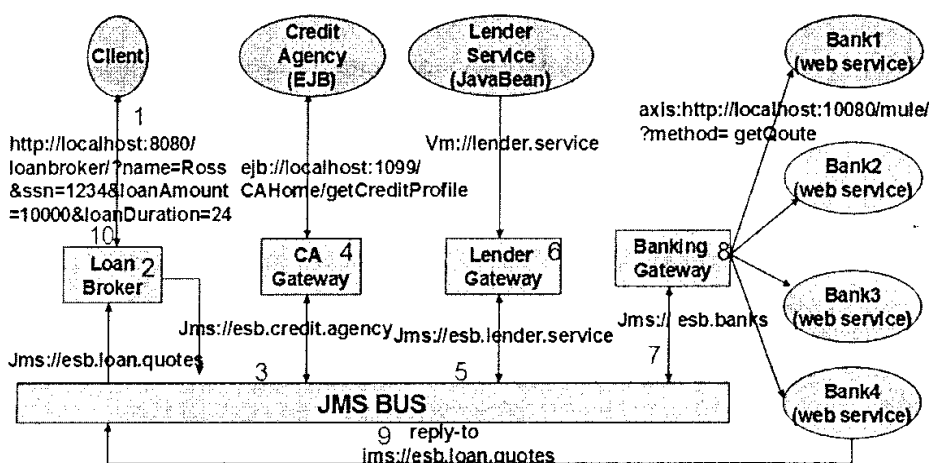


图 5-4 Loanbroker-esb 流程图

之所以这样设计，主要从以下两个方面来考虑：

1. 限制：应用程序需要支持请求/响应的处理模式；可能会面临大量的请求；同步处理模式效果不会太好。
2. 传输使用 JMS 消息总线；需要通过 JMS、Http/Rest、VM 和 Soap 来调用服务；需要调用在内部应用系统容器（EJB）里面的服务；将组建作为 Web 服务来展示。

在这过程中，贷款经纪人(loan broker)的作用如图 5-5 所示，一旦受到客户的贷款请求，贷款经纪人会做一下几个步骤：

1. 从征信所(Credit Bureau)获得客户的信用信息。
2. 将得到的信息发送给银行。
3. 将银行返回的信息整合后发给客户以供选择。

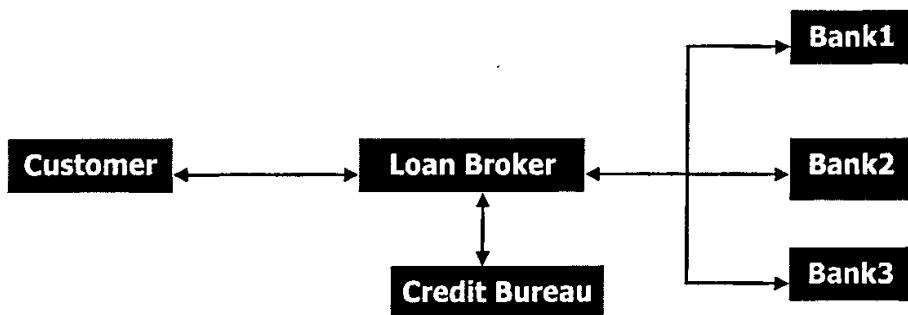


图 5-5 贷款经纪人示意图

在这个例子中，Mule 总共有六个组件：

1. **Loan Broker Service:** 接收客户请求（用户名，社会安全号，贷款金额，还贷期限），并整合银行返回的信息后发送给客户。
2. **Credit Agency Service:** 一个外部服务提供者。提供对客户信用度查询的服务，以保证客户的贷款请求是可行的。
3. **Credit Agency Gateway:** 对来自消息总线的请求进行排列并相继发送到征信所应用系统。
4. **Lender Service:** 根据客户的信用度、贷款金额以及还贷期限选择不同的银行。
5. **Lender Gateway:** 对来自消息总线的请求进行排列并相继发送到 Lender Service 应用程序。
6. **Banking Gateway:** 将贷款请求发送给一个或多个银行。

在这个例子中，如图 5-6 所示，Mule 总共使用了 5 种通信方式：

1. **LoanBroker (Http/Rest):** 通过 Http 协议接收来自客户段的请求。
2. **Credit Agency (EJB):** 由贷款经纪人管理的一个 EJB 应用系统。它提供一个可调用的含有一个 `getCreditProfile` 方法的 EJB `creditAgency`。
3. **Lender Application (VM):** 一个本地组件，一个决定使用哪个贷方的贷款请求。
4. **Banks (SOAP):** 提供 web 服务的银行。
5. **Gateways (JMS):** 从消息总线接收请求经过排列后相继发送到外部应用系统或服务。

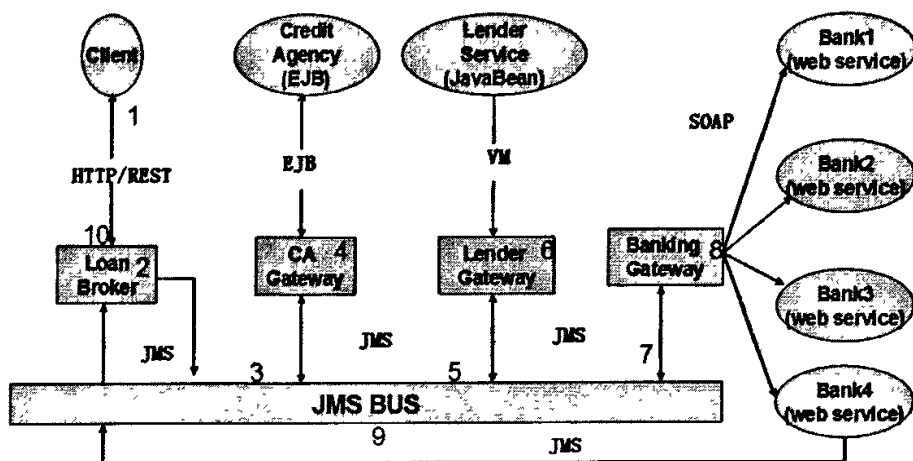


图 5-6 Loanbroker-esb 所使用的通信方式

下面我们相信分析该例子的事件流：

1. 客户发送一个 CustomerQuoteRequest 请求消息给 LoanBroker。
2. LoanBroker 创建一个 LoanQuoteRequest 消息。
3. Mule 通过 JMS 将这个消息发送给 Credit Agency Gateway。
4. Gateway 对请求消息进行排列并调用 CreditAgency EJB,将客户的信用度添加到 LoanQuoteRequest 消息中。
5. Mule 通过 JMS 将这个消息发送给 Lender Gateway。
6. Gateway 通过 VM 传输调用 Lender 应用程序。
7. Mule 通过 JMS 将消息发送给 Banking Gateway。
8. Banking Gateway 通过 Soap 和银行进行通信。
9. 每个银行将自身的贷款利率附加到请求中并根据 Banking Gateway 提供的响应地址通过 JMS 将它返回给 LoanBroker。
10. Loan Broker 服务上的 ResponseRouter 对接收到响应消息进行处理，选择最低贷款利率的银行和利率返回给客户。

具体的配置文件请参考附录 2：loanbroker-esb 配置文件

5.2 使用 Mule ESB 进行开发

在这一节中，将讲解如何利用 Mule 已有技术如 JDBC Provider、Http Provider、JMS Provider 和 Soap Provider 等几个主要的技术进行开发。

5.2.1 JDBC Provider

通过 JDBC Provider 可以对关系数据库进行查询、插入、删除、更新等操作。主要有以下两种方式：通过配置文件方式、通过 MuleClient 方式。

- 通过配置文件方式对关系数据库进行操作：

由于 Mule 提供了对关系数据库的支持，因此只要在配置文件中进行配置即可对关系数据库进行操作。

首先是数据源的配置，在 Mule 配置文件中配置 JDBC 连接。这有两种方法：

1. 在 spring 配置文件中对 jdbc 连接进行配置，然后在 mule 配置文件中对 spring 的配置文件进行引用。
2. 通过 jndi repository 来获取数据源

我们通过例子(以 MySQL 数据库为例)来进行说明：

1. Spring 方式

首先，在 spring 配置文件中的配置如下：

```
<bean id="myDataSource"
class="org.enhydra.jdbc.standard.StandardDataSource"
destroy-method="shutdown">
  <property name="driverName"><value>com.mysql.jdbc.Driver</value>
</property><property name="url">
<value>jdbc:mysql://localhost/test</value>
</property><property name="user" value="root"/>
<property name="password" value="root"/></bean>
```

接下来在 mule 配置文件中引用 spring 的配置文件：

```
<container-context
className="org.mule.extras.spring.SpringContainerContext"><properties>
<property value="conf/applicationContext.xml" name="configFile"/>
</properties></container-context>
```

其中 configFile 的 value 值为 spring 配置文件所在的目录

2. jndi repository 方式

Mule 配置文件如下：


```

<connector name="jdbcConnector"
className="org.mule.providers.jdbc.JdbcConnector">
<properties><property name="jndiInitialFactory" value="..."/>
<property name="jndiProviderUrl" value="..."/>
<property name="dataSourceJndiName" value="..."/>
</properties></connector>

```

jndiInitialFactory: 当获取对象时，定义初始的上下文工厂 (Context Factory)

jndiProviderUrl: JNDI 提供者的 url

dataSourceJndiName: Jndi 设置的数据源名称

数据源配置完后接下来就可以对数据库进行操作了。Mule 对数据库操作提供了三种操作模式：

1. Read 查询:即 select 查询语句
2. Ask 查询:即 update 或 delete 查询语句
3. Write 查询:即 insert 查询语句

具体配置如下：

```

<map name="queries">
<property name="getTest"
value="SELECT ID, TYPE, DATA, ACK, RESULT FROM TEST WHERE
TYPE = ${type} AND ACK IS NULL"/>
<property name="getTest.ack"
value="UPDATE TEST SET ACK = ${NOW} WHERE ID = ${id} AND
TYPE =${type} AND DATA = ${data}"/>
<property name="writeTest"
value="INSERT INTO TEST(ID, TYPE, DATA, ACK, RESULT) VALUES
(NULL, ${type}, ${payload}, NULL, NULL)"/>
</map>

```

然后在端点地址对它进行引用：

对于接收端点(receiver)：

```

<mule-descriptor name="..." implementation="..."

```

```
inboundEndpoint="jdbc://getTest?type=1" outboundEndpoint="..."/>
```

对于分发端点(dispatcher):

```
<mule-descriptor name="..." implementation="..." inboundEndpoint="..."  
    outboundEndpoint="jdbc://writeTest?type=1"/>
```

例如, 在本例中用到了 Read 查询:

```
<map name="queries">  
    <property name="getPeople"  
        value="SELECT firstname, lastname from person"/>  
</map>
```

然后在端点地址对它进行引用:

```
<inbound-router>  
    <endpoint address="jdbc://getPeople?type=1"/>  
</inbound-router>
```

- 通过 MuleClient 对关系数据库进行操作

Mule Client 是一个简单的接口。通过它, java 客户端可以向 Mule 服务器和其它应用程序发送和接收事件。

Mule Client 提供以下几种功能:

1. 从/向本地 Mule 服务器接收/发送事件
2. 从/向远端 Mule 服务器接收/发送事件
3. 和 Mule 提供传输支持的第三方应用程序进行通信
4. 在 Mule 服务器端注册或注销组件

我们通过 MuleClient 对关系数据库正是利用了它提供的第三种功能。

通过 MuleClient 对关系数据库进行操作有两个步骤:

1. 配置好关系数据库的配置文件
 - A. 配置好数据源, 具体参见上面的通过 spring 对数据源的设置
 - B. 配置 JDBC 连接器 (具体可以参考如下配置):

```
<connector name="jdbcConnector"  
    className="org.mule.providers.jdbc.JdbcConnector">  
    <properties>  
        <container-property name="dataSource" reference="myDataSource"/>
```

```

        <property value="30000" name="pollingFrequency"/>
    </properties>
</connector>

```

2. 在程序中对数据库进行操作

代码如下：

```

Try {
    MuleXmlConfigurationBuilder builder = new MuleXmlConfigurationBuilder();
    builder.configure(config, null);
    MuleClient client = new MuleClient();
    UMOMessage message = client.receive("SELECT * FROM taginfo", 10000);
} catch (Exception e) {
    e.printStackTrace();
}

```

其中 config 表示配置文件的路径，10000 表示等待的响应事件，message 表示查询结果，可以通过 `Object obj = message.getPayload()` 来对查询结果进行处理。若使对数据库进行 insert 或 update 操作，则可以通过 MuleClient 的 send 方法

5.2.2 Http Provider

在 Mule 中，Http 连接器(Connector)配置属性如表 5-1 所示：

表 5-1 Http 连接器属性表

属性	描述	默认值	是否必须
proxyHostname	使用代理时的代理主机名		No
proxyPort	代理端口		No
proxyUsername	代理的用户名		No
proxyPassword	代理的密码		No
timeout	等等时间	5000	Yes
bufferSize	读写数据的缓存大小	64*1024	Yes
secure	决定是否使用 https 连接.	false	No

而端点(Endpoints)的配置属性如表 5-2 所示:

表 5-2 Http 端点属性表

属性	描述	默认值
http.status	指定响应的状态	200
Content-Type	http 响应的有效载荷的类型	text/plain
Location	重定向的 URL	

下面我们通过例子来进行说明:

配置文件如下:

```
<endpoint name="httpEndpoint" address="http://localhost:8888"
    responseTransformers="HttpRequestToQueryString">
    <filter className="org.mule.routing.filters.logic.NotFilter">
        <filter pattern="/favicon.ico"
className="org.mule.providers.http.filters.HttpRequestWildcardFilter"/>
    </filter><properties>
        <property name="Content-Type" value="text/plain"/>
    </properties></endpoint>
```

代码如下:

```
protected Object doTransform(Object src, String encoding)
    throws TransformerException {
    String param = null;
    if (src instanceof byte[]) {
        if (encoding != null) {try {param = new String((byte[]) src, encoding);
        } catch (UnsupportedEncodingException ex) {
            param = new String((byte[]) src);}
        } else {param = new String ((byte []) src) ;}
    } else {param = src.toString () ;}
```

```

int equals = param.indexOf("=");
if (equals > -1) {String tagNames = param.substring(equals + 1);
    String sql = "select * from taginfo where tagName = '" + tagNames + "'";
    try {List list = this.query(sql);
        for (Iterator it = list.iterator(); it.hasNext();) {
            Object obj = it.next();
            HashMap m = (HashMap) obj;
            for (Iterator iterator = m.keySet().iterator(); iterator
                .hasNext();) {Object tagName = iterator.next();
                Object value = m.get(tagName);
                System.out.println(tagName + " = " + value); }}
    String s = "<html><head><meta http-equiv='Content-Type' content='text/html; charset=utf-8'><body></body>" + list.toString() + "</head></html>";
    return s;
    } catch (Exception e) {
        e.printStackTrace();
        return null; }
    } else {throw new TransformerException(Message
        .createStaticMessage("Failed to parse param string:"
            + param), this); }}

```

5.2.3 Web Service

Mule 对 Web Service 提供了三种产品支持形式

- Apache Axis
- Webmethods Glue
- Codehaus XFire

下面主要针对 Apache Axis 进行讲解

当 Axis 和 Mule 结合在一起时就意味着不再需要一个单独的 servlet 容器。

当一个 axis 端点例如 `axis:http://localhost:81/services` 被定义时, 如图 5-7 所示, mule 做了两件事:

- 在 Mule 中创建一个 Axis 组件, 这和 Axis Servlet 类似
- 为 `localhost:81` 创建一个 http 连接者, 并将它作为组件的接收者在服务器中注册

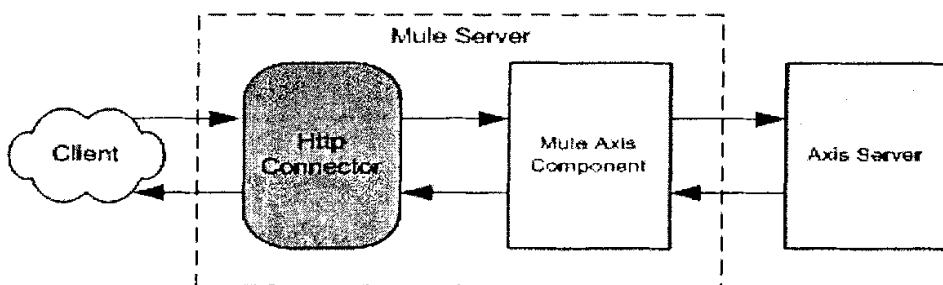


图 5-7 Mule、Axis 结合图

由于 Axis 和 mule 整合在一起, 因此你可以将 mule 组件作为一个 Axis 服务, 并从 mule 调用 Axis 的服务。要把 mule 组件作为 Axis 服务展露出来 mule 组件的接收端点必须是一个 Axis URL。例如:

```
<mule-descriptor name="queryService"
    inboundEndpoint="axis:http://localhost:81/services"
    implementation="org.mule.samples.soap.SoapJdbcComponent">
</mule-descriptor>
```

当 mule 服务器启动之后, 这个 Web Services 在 `http://localhost:81/services/queryService` 就可以使用了。SoapJdbcComponent 实现 queryService 接口, 该接口有一个对关系数据库进行查询的方法 query, 该方法的传入参数是 String 型。

那么如何调用这个 Web Service 呢, 对此, Mule 提供了两种调用方法:

a) 通过 MuleClient

我们通过下面这个例子来说明:

```
public static void main(String[] args) {
    MuleClient client; try {client = new MuleClient();
    UMOMessage result = client.send (
    "axis:http://localhost:81/services/queryService?method=query",
    "Simul_12",    null);
    System.out.println("Message Echoed is: "+result.getPayload());
```

```

client.dispose();
} catch (UMOException e) {
    e.printStackTrace();}

```

b) 普通的 Web Services 调用

```

public static void main(String[] args) {
    Service service = new Service();

    try {
        Call call = (Call) service.createCall();
        call.setTargetEndpointAddress
            ("http://localhost:81/services/queryService");
        call.setOperationName("query");
        String str = "Simul_12";
        try {System.out.println(call.invoke(new Object[] {str}));}
        } catch (RemoteException e) {
            e.printStackTrace();}
    } catch (ServiceException e) {
        e.printStackTrace();}
}

```

5.2.4 JMS Provider

由于 JMS 在 Mule ESB 中起着非常重要的作用，因此本小节重点介绍 Mule 和 JMS 的融合。

1) 背景

当前，CORBA、DCOM、RMI 等 RPC 中间件技术已广泛应用于各个领域。但是面对规模和复杂度都越来越高的分布式系统，这些技术也显示出其局限性：

- 同步通信：客户发出调用后，必须等待服务对象完成处理并返回结果后才能继续执行；
- 客户和服务对象的生命周期紧密耦合：客户进程和服务对象进程都必须正常运行；如果由于服务对象崩溃或者网络故障导致客户的请求不可达，客户会接收到异常；

- 点对点通信：客户的一次调用只发送给某个单独的目标对象。

面向消息的中间件（Message Oriented Middleware, MOM^[13]）较好的解决了以上问题。发送者将消息发送给消息服务器，消息服务器将消息存放在若干队列中，在合适的时候再将消息转发给接收者。这种模式下，发送和接收是异步的，发送者无需等待；二者的生命周期未必相同：发送消息的时候接收者不一定运行，接收消息的时候发送者也不一定运行；一对多通信：对于一个消息可以有多个接收者。

2) JMS 概念

已有的 MOM 系统包括 IBM 的 MQSeries、Microsoft 的 MSMQ 和 BEA 的 MessageQ 等。由于没有一个通用的标准，这些系统很难实现互操作和无缝连接。Java Message Service (JMS) 是 SUN 提出的旨在统一各种 MOM 系统接口的规范，它包含点对点 (Point to Point, PTP) 和发布/订阅 (Publish/Subscribe, pub/sub) 两种消息模型，提供可靠消息传输、事务和消息过滤等机制。

JAVA 消息服务(JMS)定义了 Java 中访问消息中间件的接口。JMS 只是接口，并没有给予实现，实现 JMS 接口的消息中间件称为 JMS Provider。

JMS 一般都包含至少三个组成部分：

两个 JMS 客户和一个 JMS 服务器。两个客户通过 JMS 服务器相互通信。JMS 客户是使用 JMS API 发送和接收消息的常规应用程序。

JMS 服务器可以是任何实现 JMS 规范的应用程序。一些 JMS 服务器是更大的应用程序的一部分；还有一些是专门负责 JMS 任务的应用程序。有很多第三方商业资源和一些开放源代码资源的 JMS 服务器可供选择使用。

应用程序使用 JMS 相互通信有两个方法可以选用：JMS 主题和 JMS 队列。主题和队列只在很少一些地方有区别，其中最明显的区别是它们发送消息的方式不同。

JMS 主题从一个 JMS 客户接收消息然后将这些消息分发给所有注册为主题监听者的 JMS 客户。相反，JMS 队列只将消息分发给一个客户，不管有多少客户注册为队列监听者。如果两个或者多个客户注册到一个队列，同时一个消息存储在队列中，那么只有一个客户能接收到这个消息。JMS 规范没有规定哪个客户将接收这个消息。

3) JMS 接口

JMS 支持两种消息类型 PTP 和 Pub/Sub, 分别称作: PTP Domain 和 Pub/Sub Domain, 这两种接口都继承统一的 JMS 父接口, JMS 主要接口如表 6-3 所示:

表 5-3 JMS 接口属性表

JMS 父接口	PTP	Pub/Sub
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

详细解释如下:

- ConnectionFactory : 连接工厂, JMS 用它创建连接
- Connection : JMS 客户端到 JMS Provider 的连接
- Destination : 消息的目的地
- Session: 一个发送或接收消息的线程
- MessageProducer: 由 Session 对象创建的用来发送消息的对象
- MessageConsumer: 由 Session 对象创建的用来接收消息的对象

4) JMS 消息模型

JMS 消息由以下几部分组成: 消息头, 属性, 消息体。

● 消息头

消息头包含消息的识别信息和路由信息, 消息头包含一些标准的属性 JMSDestination, JMSMessageID 等, 如表 6-4 所示:

● 属性

除了消息头中定义好的标准属性外, JMS 提供一种机制增加新属性到消息头 中, 这种新属性包含以下几种:

- 1) 应用需要用到的属性
- 2) 消息头中原有的一些可选属性
- 3) JMS Provider 需要用到的属性

标准的 JMS 消息头包含以下属性，如表：5—5 所示。

表 5-4 消息头属性表

消息头	由谁设置
JMSDestination	send 或 publish 方法
JMSDeliveryMode	send 或 publish 方法
JMSExpiration	send 或 publish 方法
JMSPriority	send 或 publish 方法
JMSMessageID	send 或 publish 方法
JMSTimestamp	send 或 publish 方法
JMSCorrelationID	客户
JMSReplyTo	客户
JMSType	客户
JMSRedelivered	JMS Provider

● 消息体

JMS API 定义了 5 种消息体格式，也叫消息类型，你可以使用不同形式发送/接收数据并可以兼容现有的消息格式。

这 5 种类型如表 5—6 所示。

例如，创建并发送一个 `TextMessage` 到一个队列：

```
TextMessage message = queueSession.createTextMessage();  
message.setText(msg_text); // msg_text is a String ; queueSender.send(message);
```

接收消息并转换为合适的消息类型：

```
Message m = queueReceiver.receive();  
if (m instanceof TextMessage) {  
    TextMessage message = (TextMessage) m;  
    System.out.println("Reading message: " + message.getText());  
    else { // Handle error }
```

5) JMS 消息的同步/异步接收

消息的同步接收是指客户端主动去接收消息，JMS 客户端可以采用 `MessageConsumer` 的 `receive` 方法去接收下一个消息。

消息的异步接收是指当消息到达时，主动通知客户端。JMS 客户端可以通过注册一个实现 MessageListener 接口的对象到 MessageConsumer，这样，每当消息到达时，JMS Provider 会调用 MessageListener 中的 onMessage 方法。

表 5-5 标准 JMS 消息头属性

JMSDestination	消息发送的目的地
JMSDeliveryMode	传递模式，有两种模式：PERSISTENT 和 NON_PERSISTENT，PERSISTENT 表示该消息一定要被送到目的地，否则会导致应用错误。NON_PERSISTENT 表示偶然丢失该消息是被允许的，这两种模式使开发者可以在消息传递的可靠性和吞吐量之间找到平衡点。
JMSMessageID	唯一识别每个消息的标识，由 JMS Provider 产生。
JMSTimestamp	一个消息被提交给 JMS Provider 到消息被发出的时间。
JMSCorrelationID	用来连接到另外一个消息，典型的应用是在回复消息中连接到原消息。
JMSReplyTo	提供本消息回复消息的目的地址
JMSRedelivered	如果一个客户端收到一个设置了 JMSRedelivered 属性的消息，则表示可能该客户端曾经在早些时候收到过该消息，但并没有签收(acknowledged)。
JMSType	消息类型的识别符。
JMSExpiration	消息过期时间，等于 QueueSender 的 send 方法中的 timeToLive 值或 TopicPublisher 的 publish 方法中的 timeToLive 值加上发送时刻的 GMT 时间值。如果 timeToLive 值等于零，则 JMSExpiration 被设为零，表示该消息永不过期。如果发送后，在消息过期时间之后消息还没有被发送到目的地，则该消息被清除。
JMSPriority	消息优先级，从 0-9 十个级别，0-4 是普通消息，5-9 是加急消息。JMS 不要求 JMS Provider 严格按照这十个优先级发送消息，但必须保证加急消息要先于普通消息到达。

6) PTP 消息类型

PTP(Point-to-Point)模型是基于队列的，发送方发消息到队列，接收方从队列接收消息，队列的存在使得消息的异步传输成为可能。和邮件系统中的邮箱一样，队列可以包含各种消息，JMS Provider 提供工具管理队列的创建、删除。JMS PTP 模型定义了客户端如何向队列发送消息，从队列接收消息，浏览队列中的消息。JMS PTP 模型中的主要概念和对象如表 5-7 所示：

7) Pub/Sub 消息模型

JMS Pub/Sub 模型定义了如何向一个内容节点发布和订阅消息，这些节点被称作主题(topic)。

主题可以被认为是消息的传输中介，发布者(publisher)发布消息到主题，订阅者(subscribe)从主题订阅消息。主题使得消息订阅者和消息发布者保持互相独立，不需要接触即可保证消息的传送。

表 5-6 JMS 消息类型

消息类型	消息体
TextMessage	java.lang.String 对象, 如 xml 文件内容
MapMessage	名/值对的集合, 名是 String 对象, 值类型可以是 Java 任何基本类型
BytesMessage	字节流
StreamMessage	Java 中的输入输出流
ObjectMessage	Java 中的可序列化对象
Message	没有消息体, 只有消息头和属性

表 5-7 PTP 消息模型

名称	描述
Queue	由 JMS Provider 管理, 队列由队列名识别, 客户端可以通过 JNDI 接口用队列名得到一个队列对象。
TemporaryQueue	由 QueueConnection 创建, 而且只能由创建它的 QueueConnection 使用。
QueueConnectionFactory	客户端用 QueueConnectionFactory 创建 QueueConnection 对象。
QueueConnection	一个到 JMS PTP provider 的连接, 客户端可以用 QueueConnection 创建 QueueSession 来发送和接收消息。
QueueSession	提供一些方法创建 QueueReceiver、QueueSender、QueueBrowser 和 TemporaryQueue。如果在 QueueSession 关闭时, 有一些消息已经被收到, 但还没有被签收(acknowledged), 那么, 当接收者下次连接到相同的队列时, 这些消息还会被再次接收。
QueueReceiver	客户端用 QueueReceiver 接收队列中的消息, 如果用户在 QueueReceiver 中设定了消息选择条件, 那么不符合条件的消息会留在队列中, 不会被接收到。
QueueSender	客户端用 QueueSender 发送消息到队列。
QueueBrowser	客户端可以 QueueBrowser 浏览队列中的消息, 但不会收走消息。
QueueRequestor	JMS 提供 QueueRequestor 类简化消息的收发过程。QueueRequestor 的构造函数有两个参数: QueueSession 和 queue, QueueRequestor 通过创建一个临时队列来完成最终的收发消息请求。
可靠性(Reliability)	队列可以长久地保存消息直到接收者收到消息。接收者不需要因为担心消息会丢失而时刻和队列保持激活的连接状态, 充分体现了异步传输模式的优势。

8) JMS Pub/Sub 模型中的主要概念和对象如表 5-8 所示。JMS 开发步骤
广义上说, 一个 JMS 应用是几个 JMS 客户端交换消息, 开发 JMS 客户端应用由以下几步构成:

1. 用 JNDI 得到 ConnectionFactory 对象;
2. 用 JNDI 得到目标队列或主题对象, 即 Destination 对象;
3. 用 ConnectionFactory 创建 Connection 对象;
4. 用 Connection 对象创建一个或多个 JMS Session;

5. 用 Session 和 Destination 创建 MessageProducer 和 Consumer;
6. 通知 Connection 开始传递消息。

表 5-8 Pub/Sub 消息模型

名称	描述
订阅(subscription)	消息订阅分为非持久订阅(non-durable subscription)和持久订阅(durable subscription), 非持久订阅只有当客户端处于激活状态, 也就是和 JMS Provider 保持连接状态才能收到发送到某个主题的消息, 而当客户端处于离线状态, 这个时间段发到主题的消息将会丢失, 永远不会收到。持久订阅时, 客户端向 JMS 注册一个识别自己身份的 ID, 当这个客户端处于离线时, JMS Provider 会为此 ID 保存所有发送到主题的消息, 当客户再次连接到 JMS Provider 时, 会根据自己的 ID 得到所有当自己处于离线时发送到主题的消息。
Topic	主题由 JMS Provider 管理, 主题由主题名识别, 客户端可以通过 JNDI 接口用主题名得到一个主题对象。JMS 没有给出主题的组织和层次结构的定义, 由 JMS Provider 自己定义。
TemporaryTopic	临时主题由 TopicConnection 创建, 而且只能由创建它的 TopicConnection 使用。临时主题不能提供持久订阅功能。
TopicConnectionFactory	客户端用 TopicConnectionFactory 创建 TopicConnection 对象。
TopicConnection	TopicConnection 是一个到 JMS Pub/Sub provider 的连接, 客户端可以用 TopicConnection 创建 TopicSession 来发布和订阅消息。
TopicSession	TopicSession 提供一些方法创建 TopicPublisher、TopicSubscriber、TemporaryTopic。它还提供 unsubscribe 方法取消消息的持久订阅。
TopicPublisher	客户端用 TopicPublisher 发布消息到主题。
TopicSubscriber	客户端用 TopicSubscriber 接收发布到主题上的消息。可以在 TopicSubscriber 中设置消息过滤功能, 这样, 不符合要求的消息不会被接收。
Durable TopicSubscriber	如果一个客户端需要持久订阅消息, 可以使用 Durable TopicSubscriber, TopicSession 提供一个方法 createDurableSubscriber 创建 Durable TopicSubscriber 对象。
恢复和重新派送 (Recovery and Redelivery)	非持久订阅状态下, 不能恢复或重新派送一个未签收的消息。只有持久订阅才能恢复或重新派送一个未签收的消息。
TopicRequestor	JMS 提供 TopicRequestor 类简化消息的收发过程。TopicRequestor 的构造函数有两个参数: TopicSession 和 topic。TopicRequestor 通过创建一个临时主题来完成最终的发布和接收消息请求。
可靠性(Reliability)	当所有的消息必须被接收, 则用持久订阅模式。当丢失消息能够被容忍, 则用非持久订阅模式。

9) Mule 提供的 JMS 属性

JMS 属性如表 5-9 所示:

表 5-9 JMS 属性表

属性	描述	默认值	是否
----	----	-----	----

			必须
acknowledgementMode	接收到消息时产生的签收类型	1	是
persistentDelivery	JMS Provider 是否保存消息	False	是
specification	JMS Server 支持的 JMS 规范。 1.0.2b 或者是 1.1	1.0.2b	是
durable	如果客户端需要持久订阅消息， 它使用了一个 Durable TopicSubscriber	False	是
clientId	用来唯一确定连接的客户端，如 果使用 durable subscribers，则必 须设置该项		是 (如 果使 用 durabl e)
noLocal	发布的消息只能通过 subscriber 自己的连接来获取	False	否
jndiInitialFactory	定义使用的初始化上下文工程		是
jndiProviderUrl	JNDI 提供者的 URL		是 (如 果使 用 JNDI)
jndiProviderProperties	要传给初始化上下文的其它属性		否
connectionFactoryProperties	connectionFactory 的属性		否
connectionFactoryJndiName	已配置的 JNDI 的连接工厂名		是 (如 果使 用 JNDI)
connectionFactory	使用的 JMS 连接工厂		否 (如 果使 用 JNDI)
connectionFactoryClasses	本地连接工厂类名		否
jndiDestinations	判断是否存在 jndi destinations， 如果没有则创建一个默认的 non-jndi destinations	False	否
forceJndiDestinations	和 jndiDestinations 属性一起使 用，如果 jndiDestinations 属性设 置为 true 但却没有找到时抛出异 常	False	否
username	创建连接时使用的用户名		否
password	创建连接时使用的密码		否
maxRedelivery	重新派送最大次数	0	否

redeliveryHandler	开发者可以通过实现 org.mule.providers.jms.RedeliveryHandler 接口来重写默认再投递行为，并将该属性设置为该实现的完整类名	org.mule.providers.jms.DefaultRedeliveryHandler	是
recoverJmsConnections	是否重新恢复未签收的消息	True	否

10) 在 Mule 中配置 JMS

首先是 Topic、Queue 配置如果没有配置 destinations 和 JndiDestinations 时就创建默认的 Queue 或 Topic。例如：一个 JMS 端点为：jms://my.destination，则创建一个默认的名称为：my.destination 的 JMS Queue。若端点为：jms://topic:my.destination，则创建一个默认的名称为 my.destination 的 JMS Topic。如果需要认证，则配置如下：

jms://ross:secret@my.destination?createConnector=ALWAYS

或 jms://ross:secret@topic:my.destination?createConnector=ALWAYS，加上 ALWAYS 是为了确保用户认证时创建一个新的连接

其次是 JMS 配置，Mule 提供了 12 种 JMS 服务器的配置方法：ActiveMQ、JBoss MQ、Joram、OpenJms、Oracle AQ、SeeBeyond、SonicMQ、Spirit Wave、Tibco EMS、UberMQ、Weblogic Jms 和 IBM WebSphere MQ。下面主要讲 ActiveMQ 的配置：

➤ 内置的 broker 配置文件如下：

```
<connector
name="jmsConnector" className="org.mule.providers.jms.JmsConnector">
  <properties><property name="specification" value="1.1"/>
    <property name="connectionFactoryJndiName" value="ConnectionFactory"/>
    <property name="jndiInitialFactory"
      value="org.activemq.jndi.ActiveMQInitialContextFactory"/>
    <map name="connectionFactoryProperties">
      <property name="brokerURL" value="vm://localhost"/>
      <property name="brokerXmlConfig"
        value="classpath:/org/mule/test/activemq-config.xml"/>
    </map></properties></connector>
```

➤ mule-config.xml 配置文件如下:

```
<container-context className="org.mule.extras.spring.SpringContainerContext"
name="spring">
  <properties><propertyname="configFile"
    value="classpath:/org/mule/test/activemq-spring.xml"/>
  </properties></container-context>
<connector name="jmsConnector"
className="org.mule.providers.jms.JmsConnector">
  <properties><property name="specification" value="1.1"/>
    <container-property name="connectionFactory"
      reference="activeMqConnectionFactory"
      container="spring"
    </properties></connector>
```

➤ activemq-spring.xml 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans><bean id="activeMqConnectionFactory"
  <property name="brokerURL" value="vm://localhost"/>
  <property name="brokerXmlConfig"
value="classpath:/org/mule/test/activemq-config.xml"/> </bean></beans>
```

➤ activemq-config.xml 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//ACTIVEMQ//DTD//EN"
"http://activemq.org/dtd/activemq.dtd">
<beans><broker><connector>
  <serverTransport uri="vm://localhost"/></connector>
<persistence><vmPersistence/></persistence></broker></beans>
```

11) 通过独立的客户端获取 JMS 消息

通过下面这个例子可以很清楚地明白如何编写客户端程序来获取 JMS 消息。

配置文件如下：

```
<connector name="jmsConnector"
  className="org.mule.providers.jms.JmsConnector">
  <properties><property name="specification" value="1.1"/>
  <property name="connectionFactoryJndiName"
    value="ConnectionFactory"/>
  <property name="durable" value="true"></property>
  <property name="clientId" value="14"></property>
  <property name="jndiInitialFactory"
    value="org.activemq.jndi.ActiveMQInitialContextFactory"/>
  <map name="connectionFactoryProperties">
    <property name="brokerURL" value="tcp://localhost:61616"/>
    <property name="useEmbeddedBroker" value="true"/>
    <property value="true" name="forceJndiDestinations"></property>
  </map> </properties></connector>
```

该连接配置文件使用内置 Broker，使用 Tcp 协议对消息进行传输。

```
<mule-descriptor name="DBSend"
  implementation="org.mule.components.simple.BridgeComponent">
  <inbound-router><endpoint address="jdbc://getTagInfo">
    <properties><map name="queries">
      <property name="getTagInfo"
        value="SELECT tagName,carNo FROM taginfo where id = '1'"/>
    </map></properties></endpoint></inbound-router>
  <outbound-router><router
    className="org.mule.routing.outbound.OutboundPassThroughRouter">
    <endpoint address="jms://topic:topic.MyTopic"
      connector="jmsConnector"></endpoint></router>
  </outbound-router></mule-descriptor>
```

配置说明：对数据库进行查询，将查询结果发送到 JMS 服务器上，消息的类型是 Topic，名称为 topic.MyTopic。

代码说明：首先定义了 url，即 tcp://localhost:61616，消息类型是否为 Topic，这里是 true，subject 设置为 Topic 名，这里为 topic.Topic，maxiumMessages 表示要接收的消息的条数，当接收到指定的条数时自动和服务器断开连接，如果设置为 0 则表示监听服务器端 名为 topic.Topic 的 Topic，clientID 为客户端的 ID，transacted 表示是否进行事务处理，sleepTime 表示接收到消息后客户端隔多长时间再和服务器连接。

由于通过 Mule 传递到 JMS 服务器上的消息的类型为 ActiveMQMapMessage，它属于 MapMessage 的一种，因此接收到消息后还应再作进一步的处理：

```
if (message instanceof MapMessage) {  
    ActiveMQMapMessage mqMsg = (ActiveMQMapMessage) message;  
    if (verbose) {Map map = mqMsg.getTable();  
        Set set = map.keySet();String name;  
        for (Iterator it = set.iterator(); it.hasNext();) {  
            name = (String) it.next();  
            System.out.println(map.get(name)); }  
        System.out.println("Received: " + mqMsg); }  
}
```

5.2.5 实时数据库 Agilor 和 Mule 的结合

由于 Mule 没有提供对实时数据库的支持，不过却提供了如何与第三方产品进行结合的方法，即 Transport Provider。

1. Transport Provider

传输供应者主要用于提供 mule 和底层的数据源或消息源进行紧密的连接。由于 Mule 没有提供对实时数据库的操作，所以必须自己开发 Mule 传输供应者，以便使实时数据库和 Mule 紧密结合起来。

● 模式

Mule 传输供应者有三种模式：

第一: **inbound-only**——组件只能订阅事件, 不能分发事件

第二: **outbound-only**——组件只能分发事件, 不能订阅事件

第三: **inbound-outbound**——组件能分发和订阅事件

由于之需要从实时数据库 Agilor 订阅消息事件, 所以相应的模式为 **inbound-only**。

● 接口

Mule 提供了一组接口来定义 Mule 和其它技术之间的契约, 开发一个 Mule 传输供应者必须要实现这些接口, 这些接口有:

UMOConnector: 通过连接器, Mule 可以在传输时注册监听者和创建消息分发者。

UMOMessageReceiver: 用以接收输入数据并将它打包为事件(events)。它本质上是服务器端的传送实现(消息分发器 Message Dispatcher 是客户端的实现)。

UMOMessageDispatcher: 用来发送事件, 创建一个对底层技术的调用。

UMOMessageDispatcherFactory: 用来创建 UMOMessageDispatcher 实例。

UMOMessageAdapter: 为 Mule 读取数据提供一条稳定的道路。它提供读取消息有效载荷(payload)和读取消息特征的方法。

因为需要开发的模式为 **inbound-only**, 不支持 **outbound** 通讯, 所以只需要实现 **UMOConnector**、**UMOMessageReceiver** 和 **UMOMessageAdapter** 接口即可。

● 实现

Mule 为上面提到的接口提供了一个抽象实现, 它实现了大部分的 Mule 细节只留下一部分方法以供客户自己实现。

Connector: Mule 提供了一个 **AbstractConnector** 抽象类, 它实现了 Mule 连接所必须的默认功能, 例如线程配置、接收/分发管理等。剩下需要实现的方法有:

getProtocol(): 返回供应者的协议, 例如“SMTP”或者“JMS”, 必须要实现。

createReceiver(): 创建 UMOMessageReceiver 实例, 必须要实现

doInitialise(): 当所有的 bean 特征都已在连接上设置好时被调用, 用以验证和初始化连接状态。不是必须要实现。

doStart(): 执行必须的动作来使接收者开始接收事件。非必须要实现的。

doStop(): 执行必须的动作来使接收者停止接收事件。这不是必须要实现的。

doDispose(): 当连接被消除时调用, 清除任何资源。当它被调用时, **doDisconnect()**和 **doStop()**方法也被暗中调用。它不是必须要实现的。

Message Receivers: Mule 提供了一些标准的实现用以资源的检测和传输管理, 它有两种类型:

a) **polling:** 检测类似于文件系统、数据库的资源。

b) **listener-based:** 注册它本身作为传输的监听者。例如 JMS 或 Pop3。

- **Mule 提供的抽象方法**

- a) **AbstractMessageReceiver**

提供事件路由的方法。开发者继承这个类必须编写代码, 注册一个对象作为传输的监听者。需要实现的方法有:

doConnect(): 和底层传输进行连接。这是必须要实现的。

doDisconnect(): 断开连接, 清除 **doConnect()**方法时所使用的资源。这是必须要实现的。

doStart(): 执行必须的动作来使接收者开始接收事件。这不是必须要实现的。

doStop(): 执行必须的动作来使接收者停止接收事件。这不是必须要实现的。

doDispose(): 当连接被消除时调用, 清除任何资源。它不是必须要实现的。

- b) **PollingMessageReceiver**

实现建立和消除一个监听线程的功能, 提供一个在特定的周期内科重复调用的 **poll()**方法。需要实现的方法有:

poll(): 执行读取数据和返回数据。这是必须要实现的。

- c) **TransactedMessageReceiver**

根据请求进行传输管理。接收者利用事务模版处理请求和根据端点的配置创建事务本身。需要实现的方法有：

getMessages(): 返回一个代表单独事件有效载荷的 list 对象。这个有效载荷可以是任何类型的对象，并封装在一个 MuleEvent 对象中发送到 Mule 服务。这是必须要实现的。

processMessage(Object): 对 getMessages()得到的 list 对象进行处理。这是必须要实现的。

d) Connection Strategy

当连接失败或者 Mule 连接器启动时，通过它来订制传输的连接。它主要在控制什么时候在 UMOMessageReceiver 端如何调用 doConnect()方法。

e) Thread Management

线程管理

f) Message Dispatchers

Message Dispatchers 是传输的客户端实现，它们主要产生传输时的各种请求，例如向一个 socket 写数据或者调用一个 web 服务。

Mule 提供的 AbstractMessageDispater 抽象方法实现了基本的实现，只剩下三个方法需要用户根据实际情况完成：

doSend(UMOEEvent): 在传输器上传送事件的有效载荷。它是必须要实现的。

doDispatch(UMOEEvent): 当终点是异步时或调用传输器但不返回任何结果时被调用。它是必须的。

doReceive(UMOImmutableEndpoint, long): 用来产生一个请求到传送资源。它是必须要实现的。

doDispose(): 当 Dispatchers 被消除时调用，清除任何公开的资源。它是必须要实现的。

doConnect(UMOImmutableEndpoint): 和底层传输建立连接。它是必须要实现的。

doDisconnect(): 断开连接，清除资源。它是必须要实现的。

g) Message Adapters

Message Adapters 通常是简单的对象用来提供一个统一的方式来获取事件的有效载荷和底层传输过来的原数据进行关联。需要实现的方法有：

getPayload(): 返回消息的有效载荷。它是必须要实现的。

getPayloadAsString(): 以 **String** 的形式返回消息的有效载荷。它是必须要实现的。

getUniqueId(): 为消息返回一个独一无二的 **id**。它是必须要实现的。

h) Service Descriptors

provider 的内在配置文件

在 **META-INF/services/org/mule/providers/<protocol>** 添加 **provider** 使用的协议，如实时数据库 **provider** 自定义的协议 **rtdb**，它是一个文件类型，在这个文件中包含有以下信息：

Connector: 用以连接的类名。它是必须要有的

connector.factory: 不是必须的。

message.receiver: 不是必须的。如果是 **outbound** 类型则是必须的。

transacted.message.receiver: 不是必须的。

dispatcher.factory: 不是必须的。如果是 **inbound** 类型则是必须的。

message.adapter: 不是必须的。如果是 **outbound** 类型则是必须的。

inbound.transformer: 不是必须的。

outbound.transformer: 不是必须的。

endpoint.builder: 不是必须的。

service.finder: 不是必须的。

2. 实时数据库 Provider

由于实时数据库 **Agilor** 模式为 **inbound-only**，所以需要实现的接口有 **UMOConnector**、**UMOMessageReceiver** 和 **UMOMessageAdapter**。

● 服务描述文件

在 **META-INF/services/org/mule/providers** 目录下新建一个 **rtdb** 文件(自定义了一个 **rtdb** 协议)

然后在 **rtdb** 文件中定义如下配置：

connector=org.mule.providers.rtdb.RtdbConnector

message.receiver=org.mule.providers.rtdb.RtdbMessageReceiver

`message.adapter=org.mule.providers.rtdb.RtdbMessageAdapter`

- **RtdbConnector**

它继承 Mule 提供的 `AbstractServiceEnabledConnector` 抽象类，是 `UMOConnector` 的实现，并重写 `getProtocol()`，返回连接协议“rtdb”。

- **RtdbMessageReceiver**

是 `UMOMessageReceiver` 接口的实现，由于实时数据库采用的是订阅机制，因此继承了 Mule 提供的 `AbstractMessageReceiver` 方法，同时由于 Agilor 自身提供了当订阅到数据时的处理方法：`onSubDataArrival()`，因此 `RtdbMessageReceiver` 还应实现这个接口，以便接收到数据时对它进行转发，由 `routeMessage()` 方法实现。同时在该方法中还实现了读取配置文件元素属性信息的方法。

- **RtdbMessageAdapter**

继承了 Mule 提供的 `AbstractMessageAdapter` 方法。定义订阅到的消息的有效载荷为 `Object`。是 `UMOMessageAdapter` 接口的实现。

3. 实时数据库的使用配置

实时数据库的 Provider 写好之后，我们接下来举例讲解如何在 Mule 中使用它。首先看它的连接配置：

```
<connector name="rtdbConnector"
  className="org.mule.providers.rtdb.RtdbConnector">
</connector>
```

接下来看它的操作配置：

```
<inbound-router><endpoint address="rtdb://tagNames"><properties>
  <property value="127.0.0.1" name="host"/>
  <property value="userName" name="user"/>
  <property value="passWord" name="pass"/>
  <property value="Simul_11" name="tagName1"/>
  <property value="Simul_12" name="tagName2"/>
</properties></endpoint></inbound-router>
```

这样，Mule 就可以从 Agilor 中接收传来的实时数据了。

结 语

1. 系统

本次课题主要是针对煤矿监控这个实际环境而产生的,面对越来越多的煤矿事故,为了确保工作人员的生命安全,数字煤矿的概念应运而生。此次项目主要针对现实环境中采煤运煤进行模拟监控。本人所在的公司主要通过研究 Mule 对第三方公司给以技术指导,目前该项目已经通过验收,第三方公司对此比较满意,主要的要求都已经实现,如实时数据库 Agilor 与 Mule 的结合、和实际应用相关的技术如 JMS、WebService 等都已有了较为详细的使用文档,目前第三方公司正在利用 Mule 进行开发,并未出现问题。

2. 本人的论文工作

在本次项目中,我主要参与了系统模型的设计,负责对后台数据的传输以及前台数据的接收提供技术支持。在此次项目中,完成了:实时数据库 Agilor 和 Mule 的结合,实时数据库 Agilor 采集来的数据和关系数据库中的数据都通过 Mule 传输支持的方式来进行;利用 Mule 中的 Web Services 技术、HTTP 技术、JDBC 技术、MuleClient 技术实现对关系数据库的查询、更新、删除等工作;利用 Mule 中的 JMS 技术实现消息的发布以及消费;此外,在此次项目中我还负责完成 Mule IDE 的汉化工作、Mule API 帮助文档的汉化工作、Mule JAR 包的重组工作、撰写 Mule ESB 的使用文档、开发文档等。

3. 改进建议

通过此次对 Mule 的研究,发现有以下两点需要进一步进行修改:

- a) Agilor 实时数据库提供了两种服务,一是抓取到数据之后立即发送出去,另外一种是将查询结果立即发送出去。目前已经实现第一种服务和 Mule 相结合,下一步将根据实际需要实现第二种服务和 Mule 结合。
- b) Mule 提供了 JDBC 的支持,不过 JDBC 传输提供者有一个缺陷,即查询时操作只能是 Inbound 形式,而不能将接收到的消息作为查询参数对关系数据库进行查询!即查询时它不能是 OutBound 形式。

这些问题将会在接下来的项目中得到解决。

参考文献

- [1] 许琼, 鲁鹏 用于实现 Web 服务的 SOA 编程模型 软件导刊 03 期 2007 年 2 月 62
- [2] 飞思科技产品研究中心 Java Web 服务应用开发详解[M] 电子工业出版社 2002 年 190-191
- [3] DCOM <http://www.itisedu.com/phrase/200604152041495.html>
- [4] 石玉晶, 刘辉, 李建华 基于 web service 的 EAI 技术研究 计算机与信息技术 Z01 期 2007 年 48
- [5] 企业服务总线解决方案剖析 <http://caijinrong.bokee.com/3876954.html>
- [6] James Strachan What is Java Business Integration (JBI) or JSR 208
<http://radio.weblogs.com/0112098/2005/07/07.html#530>
- [7] 相鹏, 段友翔 企业服务总线(ESB)-企业集成关键技术初探 信息技术与信息化 06 期 2005 年 88
- [8] 邵欢庆, 康建初 企业服务总线的应用 计算机工程 02 期 2007 年 220-222
- [9] 陈廷彬, 夏勤, 刘业 基于 Web 服务的 ESB 在电信网管中的应用研究 计算机工程与设计 10 期 2006 年 1800-1804
- [10] 夏炎, 宋喜莲 Java 的消息服务技术应用研究 沈阳工程学院学报(自然科学版) Z1 期 2005 年 110-112
- [11] 了解 XA 事务 <http://msdn2.microsoft.com/zh-cn/library/aa342335.aspx>
- [12] Matt Welsh David Culler Eric Brewer
SEDA: An Architecture for Well-Conditioned, Scalable Internet Services
www.eecs.harvard.edu/~mdw/papers/seda-sosp01.pdf
- [13] 冯战申, 李雪臣 基于 MOM 的消息代理研究与实现 微机计算机信息 03 期 2006 年 252-254

致 谢

特别感谢我的导师朱其亮老师。朱老师一直给予我无私的帮助，在朱老师的悉心指导下，我才得以完成论文。他那严肃认真的态度和一丝不苟的敬业精神，给了我极大的启发和帮助。在此对他表示最诚挚的谢意。

此外，还要感谢北京协客网信息技术有限公司的领导陶鹏总经理和同事殷华、王彪、张磊和连宏武，在实习期间给了我很多指导和帮助。在此也向他们表示最衷心的感谢。

最后，感谢在软件学院一起学习、生活的所有同学。

附录 1

```
<?xml version="1.0" encoding="UTF-8"?>
<! DOCTYPE mule-configuration PUBLIC "-//SymphonySoft //DTD mule-configuration XML
V1.0//EN"

"http://www.symphonysoft.com/dtds/mule/mule-configuration.dtd">
<mule-configuration id="Mule_Echo_Sample" version="1.0">
  <description>
    To invoke the EchoUMO component as a webservice hit the following URL -
    http://localhost:8081/services/EchoUMO?method=echo&amp;param=Is there an echo?
    (remember to replace the '&amp;' with an ampersand)
    To view the WSDL for the EchoUMO service go to -
    http://localhost:8081/services/EchoUMO?wsdl
  </description><connector name="SystemStreamConnector"
className="org.mule.providers.stream.SystemStreamConnector">
  <properties>
    <property name="promptMessage" value="Please enter something: "/>
    <property name="messageDelayTime" value="1000"/>
  </properties>
</connector><transformers><transformer name="HttpRequestToSoapRequest"
className="org.mule.providers.soap.transformers.HttpRequestToSoapRequest"/>
</transformers>
  <mule-descriptor
    name="EchoUMO"
    implementation="org.mule.components.simple.EchoComponent">
    <inbound-router>
      <endpoint address="stream://System.in"/>
      <endpoint
        address="xfire:http://localhost:8081/services"
        transformers="HttpRequestToSoapRequest" />
    </inbound-router>
    <outbound-router>
      <router
className="org.mule.routing.outbound.OutboundPassThroughRouter">
        <endpoint address="stream://System.out"/>
      </router>
    </outbound-router>
  </mule-descriptor>
</model>
</mule-configuration>
```

附录 2

```
<? xml version="1.0" encoding="UTF-8"?>
<! DOCTYPE mule-configuration PUBLIC "-//SymphonySoft //DTD mule-configuration XML
V1.0//EN"
```

```
"http://www.symphonysoft.com/dtds/mule/mule-configuration.dtd">
```

```
<mule-configuration id="Mule_Loan_broker_ESB_with_Ejb_Container" version="1.0">
```

```
  <description>
```

The Loan Broker ESB example implements the Loan Broker Example using an ESB topology.

See <http://mule.codehaus.org/LoanBroker+ESB> for details.

Instead of using an ejb Connector as in the other configuration (see mule-config.xml in this directory)

We configure an EJB container context and reference the EJB session bean directly as a Mule Managed component

The advantage of using an EjbContainerContext over an Ejb Connector is that the container is early binding where as

connector is only invoked once an event is received for the referenced Ejb.

```
  </description>
```

```
  <container-context className="org.mule.impl.container.EjbContainerContext">
```

```
    <properties>
```

```
      <property name="securityPolicy" value="security.policy"/>
```

```
      <map name="environment">
```

```
        <property
```

```
          name="java.naming.factory.initial"
```

```
          value="org.openejb.client.LocalInitialContextFactory"/>
```

```
        <property
```

```
          name="java.naming.provider.url" value="rmi://localhost:1099"/>
```

```
      <property name="openejb.base" value="."/>
```

```
      <property
```

```
        name="openejb.configuration"
```

```
        value="../conf/openejb.conf"/>
```

```
      <property name="openejb.nobanner" value="true"/>
```

```
    </map>
```

```
  </properties>
```

```
</container-context>
```

```
  <connector
```

```
    name="jmsConnector" className="org.mule.providers.jms.JmsConnector">
```

```
    <properties>
```

```
      <property
```

```
        name="connectionFactoryIndiName" value="ConnectionFactory"/>
```

```
    </property>
```

```

        name="jndiInitialFactory"
        value="org.activemq.jndi.ActiveMQInitialContextFactory"/>
    <property name="specification" value="1.1"/>
    <map name="jndiProviderProperties">
        <property name="brokerURL" value="tcp://localhost:61616"/>
    </map>
</properties>
</connector>
<endpoint-identifiers>
    <endpoint-identifier
        name="LoanBrokerRequestsREST"
        value="jetty:rest://localhost:8080/loanbroker"/>
    <endpoint-identifier
        name="LoanBrokerRequests" value="vm://loan.broker.requests"/>
    <endpoint-identifier name="LoanQuotes" value="jms://esb.loan.quotes"/>
    <endpoint-identifier
        name="CreditAgencyGateway" value="jms://esb.credit.agency"/>
    <endpoint-identifier
        name="CreditAgency"
        value="vm://credit.agency.service?method=getCreditProfile"/>
    <endpoint-identifier
        name="LenderGateway" value="jms://esb.lender.service"/>
    <endpoint-identifier name="LenderService" value="vm://lender.service"/>
    <endpoint-identifier name="BankingGateway" value="jms://esb.banks"/>
    <endpoint-identifier
        name="Bank1" value="axis:http://localhost:10080/mule"/>
    <endpoint-identifier
        name="Bank2" value="axis:http://localhost:20080/mule"/>
    <endpoint-identifier
        name="Bank3" value="axis:http://localhost:30080/mule"/>
    <endpoint-identifier
        name="Bank4" value="axis:http://localhost:40080/mule"/>
    <endpoint-identifier
        name="Bank5" value="axis:http://localhost:50080/mule"/>
</endpoint-identifiers>
<transformers>
    <transformer
        name="RestRequestToCustomerRequest"
        className="org.mule.samples.loanbroker.esb.transformers.RestRequestToCustomerRequest"/>
    <transformer
        name="LoanQuoteRequestToCustomer"
        className="org.mule.samples.loanbroker.esb.transformers.LoanQuoteRequestToCustomer"/>

```

```

    <transformer
      name="LoanQuoteRequestToCreditProfileArgs"
      className="org.mule.samples.loanbroker.esb.transformers.LoanQuoteRequestToC
reditProfileArgs"/>
    <transformer
      name="CreditProfileXmlToCreditProfile"
      className="org.mule.samples.loanbroker.esb.transformers.CreditProfileXmlToCre
ditProfile"/>
  </transformers>
  <model name="loan-broker">
    <mule-descriptor name="LoanBroker"
      implementation="org.mule.samples.loanbroker.esb.LoanBroker">
      <inbound-router>
        <endpoint
          address="LoanBrokerRequestsREST"
          transformers="RestRequestToCustomerRequest"/>
        <endpoint address="LoanBrokerRequests"/>
      </inbound-router>
      <outbound-router>
        <router
          className="org.mule.routing.outbound.OutboundPassThroughRouter">
            <endpoint address="CreditAgencyGateway"/>
          </router>
        </outbound-router>
        <response-router timeout="1000000">
          <endpoint address="LoanQuotes"/>
        </router>
      </response-router>
    </mule-descriptor>
    <mule-descriptor
      name="CreditAgencyGateway"
      implementation="org.mule.components.builder.ReflectionMessageBuilder">
      <inbound-router>
        <endpoint address="CreditAgencyGateway"/>
      </inbound-router>
      <outbound-router>
        <router className="org.mule.routing.outbound.FilteringOutboundRouter">
          <endpoint
            remoteSync="true"
            transformers="LoanQuoteRequestToCreditProfileArgs"
            responseTransformers="CreditProfileXmlToCreditProfile"
            address="CreditAgency"/>
          <endpoint address="LenderGateway"/>
        </router>
      </outbound-router>
    </mule-descriptor>
  </model>

```

```

        </router>
    </outbound-router>
</mule-descriptor>
<mule-descriptor
    name="CreditAgency" implementation="local/CreditAgency">
    <inbound-router>
        <endpoint address="CreditAgency"/>
    </inbound-router>
</mule-descriptor>
<mule-descriptor
    name="LenderGateway"
    implementation="org.mule.components.simple.BridgeComponent">
    <inbound-router>
        <endpoint address="LenderGateway"/>
    </inbound-router>
    <outbound-router>
        <router className="org.mule.routing.outbound.ChainingRouter">
            <endpoint remoteSync="true" address="LenderService"/>
            <endpoint address="BankingGateway"/>
        </router>
    </outbound-router>
</mule-descriptor>
<mule-descriptor
    name="LenderService"
    implementation="org.mule.samples.loanbroker.esb.lender.LenderService">
    <inbound-router>
        <endpoint address="LenderService"/>
    </inbound-router>
</mule-descriptor>
<mule-descriptor name="BankingGateway"
    implementation="org.mule.components.simple.BridgeComponent">
    <inbound-router>
        <endpoint address="BankingGateway"/>
    </inbound-router>
    <outbound-router>
        <router className="org.mule.routing.outbound.StaticRecipientList">
            <reply-to address="LoanQuotes"/>
        </router>
    </outbound-router>
</mule-descriptor>
<mule-descriptor name="Bank1"
    inboundEndpoint="Bank1"
    implementation="org.mule.samples.loanbroker.esb.bank.Bank">
</mule-descriptor>

```

```
<mule-descriptor name="Bank2"
  inboundEndpoint="Bank2"
  implementation="org.mule.samples.loanbroker.esb.bank.Bank">
</mule-descriptor>
<mule-descriptor name="Bank3"
  inboundEndpoint="Bank3"
  implementation="org.mule.samples.loanbroker.esb.bank.Bank">
</mule-descriptor>
<mule-descriptor name="Bank4"
  inboundEndpoint="Bank4"
  implementation="org.mule.samples.loanbroker.esb.bank.Bank">
</mule-descriptor>
<mule-descriptor name="Bank5"
  inboundEndpoint="Bank5"
  implementation="org.mule.samples.loanbroker.esb.bank.Bank">
</mule-descriptor>
</model>
</mule-configuration>
```