EMP 技术手册

Technology Menual
For Version2.2

目录

版权说明

文档信息

文档版本编号:	1.1	文档版本日期:	2007 年 7 月 20 日
起草人:	张佳巍，彭楫洲，李嘉，高琳，刘必强	起草日期:	2007 年 7 月 10 日

版本记录

版本编号	版本日期	创建/修改	说明
1.1	2007/07/10	张佳巍，彭楫洲，李嘉，高琳，刘必强	版本初建
1.2	2007/07/19	张佳巍，高琳	增加报文格式处理章节和上传文件功能
1.3	2007/07/20	彭楫洲	修改访问控制章节
2.0	2008/04/22	彭楫洲	修改相关章节
2.2	2011/07/22	刘景应	调整格式

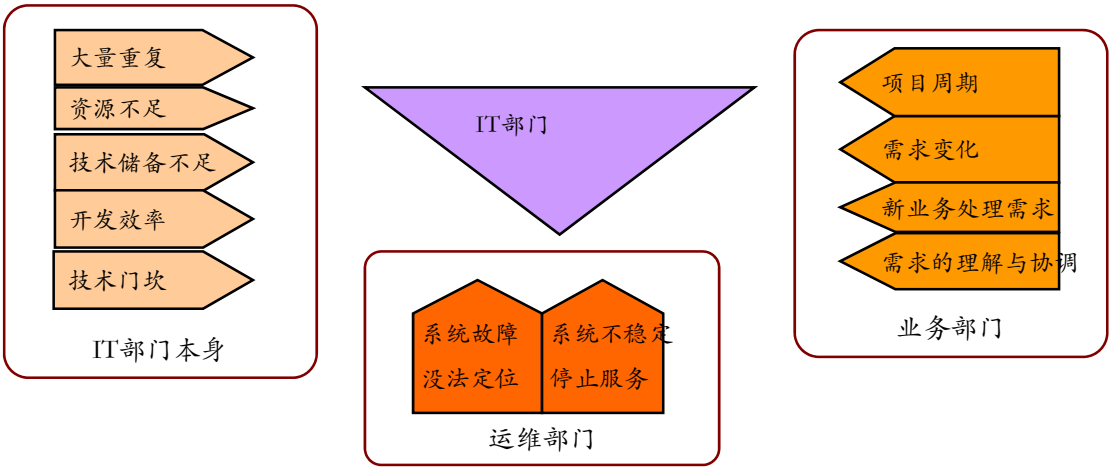
1. EMP平台概念

EMP 平台产生的过程伴随着中国金融 IT 建设的过程。随着金融 IT 系统越来越复杂，响应要求越来越高，系统压力越来越大，技术更新速度越来越快，我们需要从技术实施和管理本身来寻找技术发展与 IT 建设的适应之路。

EMP 平台的产生并非单纯从技术角度来设计一个应用平台所必须的架构、稳定性、灵活性、安全和效率，更多的是从技术发展和管理的角度来看待如何有效地管理客户的技术资源，通过平台来规范应用开发和实施流程，从而更好地迎接业务发展对 IT 技术应用的挑战。

1.1. IT建设需要平台

企业客户的 IT 部门在实际运作中，面临着越来越大的压力，这些压力来自于其他部门，同时也来自于 IT 部门自身。



如上图所示，IT 部门面临是三个方面的压力。

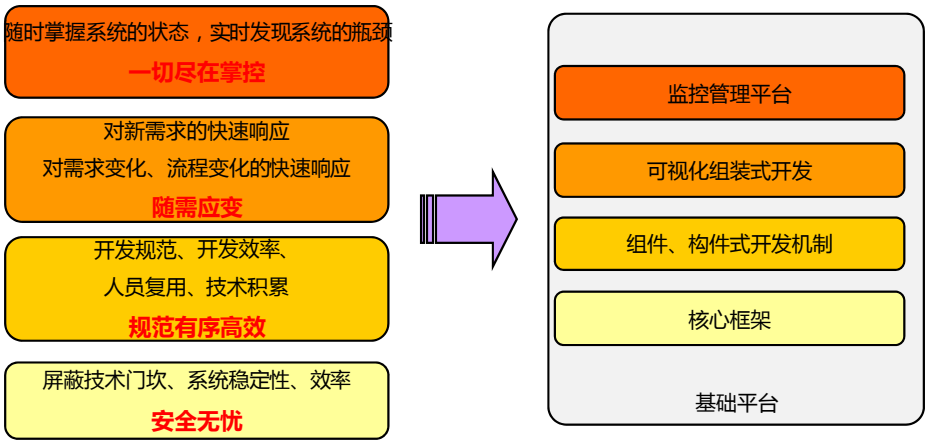
一、业务部门的快速实现的压力。这种快速实现越来越多地体现为对现有系统的随需应变的快速改造能力的需求上，而不是重新建设一套新系统的能力。这就要求 IT 部门在建设新的系统时，必须充分考虑系统设计的灵活性和强大的应变能力，这些要求导致系统更加的复杂，设计难度越来越大，但项目周期在竞争激烈的业务环境下却要求越来越短。IT 部门迫切需要一个高度灵活和高适应性的技术框架来降低系统设计的压力，缩短设计周期，提高项目实施效率。

二、运维部门的压力。运维部门提出的需求总是最迫切的，因为它直接面对客户体验。这些系统故障、性能问题需要 IT 部门快速定位、迅速排除。

三、来自 IT 部门自身的压力。项目越来越多，周期越来越短，同时面临着多种技术体系的折磨，技术储备严重不足，人力资源严重不足。所有这一切都要求 IT 部门提高开发效率，降低重复工作量，更好的积累技术资源，降低重复劳动。这些都要求有一个良好的平台来帮助 IT 部门解决以上的问题。

宇信易诚公司自成立以来，一直致力于为行业用户提供先进可靠的应用系统产品和方案，了解行业用户在 IT 应用系统上的面临的困境。我们知道：

IT 建设需要一个平台，企业用户需要稳定高效的系统架构来帮助简化应用架构的设计工作和提高应用架构的质量，并能快速开发和部署，在随需应变的商业环境下，提供需求变化和流程变化的灵活调整能力。在系统运行期间，能有效地管理和监控系统运行信息，防范于未然保证应用有效运行。这些强烈的技术需求都要求有一个建立在基础应用平台之上的完整解决方案。这正是 EMP 的价值所在。

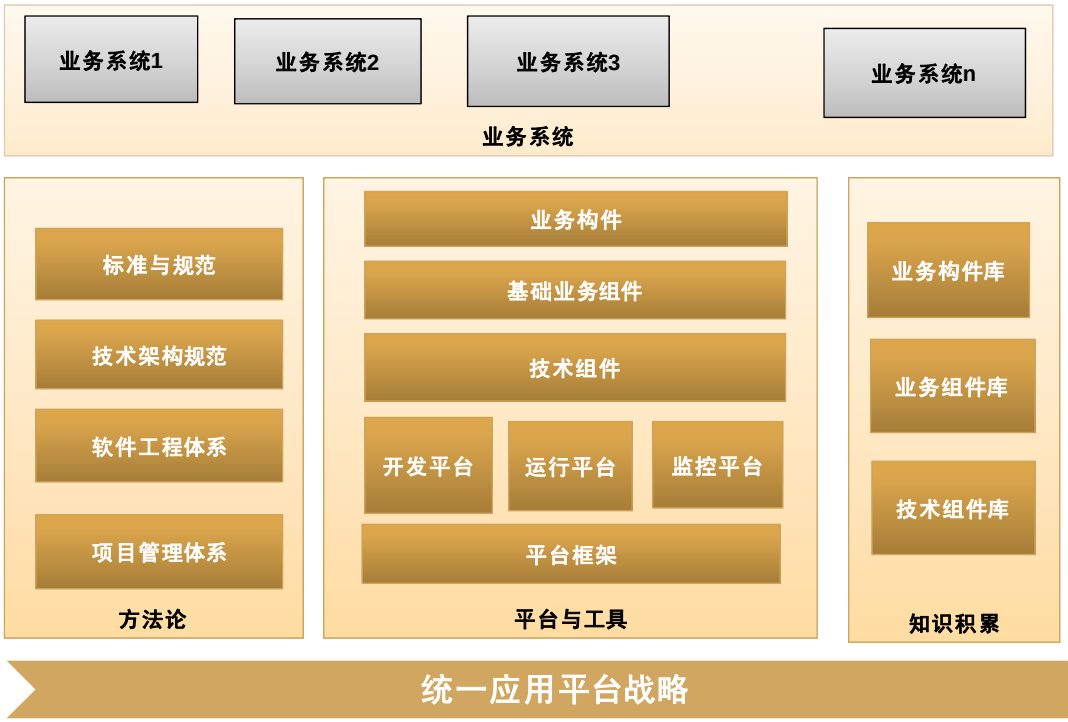


EMP 不但提供稳定、完全、灵活的系统核心框架和运行平台，并基于此提供可积累、可重用、可视化的开发工具来支撑应用系统的快速构建与部署。EMP 平台还集成了可视化、可配置、可预警的监控管理平台保证用户随时把握系统状态，一切尽在掌控。

1.2. EMP平台要解决的问题

EMP 平台力图通过统一的技术平台发展战略从方法论、技术层次和自循环供血体系三个方面来帮助用户面对复杂多变的业务环境和 IT 技术体系时所面临的问题。

如下图所示，是宇信易诚公司建立于统一应用平台战略基础之上的企业 IT 应用系统技术发展道路。



首先从方法论入手，在统一应用平台基础上制定符合企业实际情况的软件开发和管理规范和标准，建立自身的项目管理体系、技术架构规范等应用规则和操作准则，通过标准和规范来保证应用系统在可控范围内得到开发和部署以及后续发展升级。

第二，通过统一应用平台战略为企业提供技术实现保障。建立企业内部统一的平台框架和应用开发及管理相关工具。一个完整的应用平台一定是分层设计的，可以通过组装模式来完成重用和扩展的。一个完整的应用平台一定关注应用系统本身运行所涉及到的三个重要工作：应用开发、应用运行和应用监控。一个完整的平台一定会提供相应的开发工具，运行平台和监控管理工具。在基础平台之上，提供基于平台的大量技术组件，通过这些技术组件的组装和配置，可以产生符合本企业自身特点需要的基础业务组件和业务构件模型。

最后，统一平台战略一定有自身发展的模式。统一应用平台战略通过各种方式和手段来加强知识积累的能力和可行性。由于企业的绝大多数应用均按照统一的应用规范采用统一的基础技术平台进行开发、运行和管理，在这种机制中可以通过项目开发过程中不断地积累平台可重用的技术组件和业务组件，逐步丰富和壮大统一技术平台的丰富内容。由此产生平台发展的自身造血功能，从根本机制上来保证平台发展战略的可行性。

从本质上说，EMP 平台所要解决的问题实际上是企业在面对 IT 技术快速发展而带来的应用系统开发吃力导致系统质量下降，功能缩水，进而影响企业自身业务发展的困境。

面对这样的困境，企业可以选择培养自己的技术队伍，从根本上扭转面对 IT 技术发展的鸿沟。但这种投入是巨大的，而且必须是持续的，这与企业自身主营业务是无关联的，这种投入的风险性极大。企业也可以选择通过与专业公司合作建立技术应对机制，通过专业 IT 公司的帮助来克服新技术带来的负面效应。EMP 平台为用户提供正是这样的解决方案，它能帮助企业从以下几个方面得到益处：

一、 通过建立统一平台，可以有效缩短开发周期并保证应用开发质量，在第一时间完成应用系统的高质量实施，提高业务服务激动能力和响应能力，赢得市场先机。

二、 通过建立统一平台，能够有效地管理基于平台下的所有应用系统，在应用系统越来越多，应用环境越来越复杂的情况下，能够帮助企业获得清晰化和条理化的应用系统的运行维护管理能力，最大化降低系统风险和运维成本。

三、 通过建立统一平台，降低‘单个人’的效应，用机制来管理知识的积累，有效降低企业对技术人员的依赖性和技术成本。

四、 通过建立统一平台，解决企业 IT 发展面临的问题，可以让企业将更多的精力投入到主营业务服务中去，创造更多价值，获得领先优势。

2. EMP平台概览

2.1. EMP平台概览

2.1.1. EMP平台概述

EMP 平台是一个产品丰富的产品家族。它的基本信息描述如下表所示：

- 是一个包含开发、运行、监控管理的满足SOA体系架构的轻量级J2EE应用平台
- 是一个基于J2EE的多渠道整合平台
- 开放式的用户可扩展的应用框架
- 实现构件化的业务处理逻辑组装
  - 基于基础组件的组装，完成业务构件的定义
  - 提供大量的基础组件及构件
- 实现构件化的前端架构
  - AJAX技术，支持未来互联网技术
  - 提供前端框架，实现丰富的用户交互
- 提供基于JMX标准的监控管理能力
- 提供基于模型维护的可视开发工具

EMP 平台是一个包含开发、运行、监控管理的满足 SOA 体系架构的轻量级 J2EE 应用平台。在每一方面，EMP 都提供了子产品进行支持。它是一个纯 J2EE 应用平台，完全符合 J2EE 框架规范。

EMP 平台是一个专业的银行多渠道整合平台，为多渠道处理提供了框架模型，可以很容易地在 EMP 平台上搭建多渠道整合的应用。

EMP 平台采用的是开放式接口设计模式，为用户基于平台的扩展提供了良好的支持。

EMP 平台实现了构件化的业务处理逻辑组装功能。通过 EMP 提供的基础组件采用外部化 XML 文件的配置完成业务构件的定义，EMP 提供了大量的基础组件及构件帮助用户快速进行开发实施。

EMP 平台实现了构件化的前端架构，在前端采用 AJAX 技术进行了表现组件的封装，并在此基础上提供了前端处理框架，包括前端流程处理、菜单、页面布局处理框架等等。

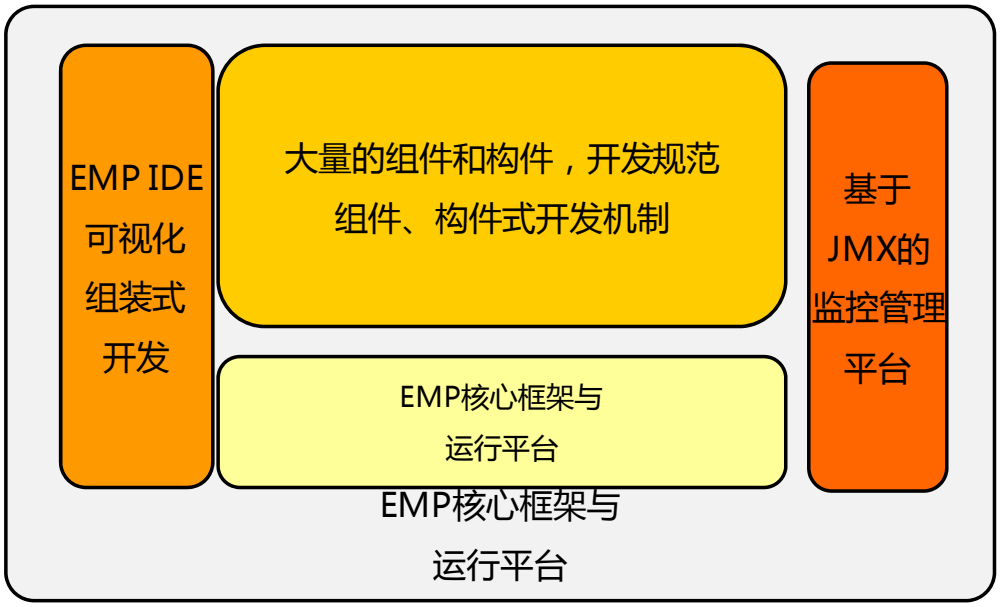
EMP 平台在监控管理上提供了独立的基于标准的 JMX 框架的监控管理平台，可以快速地将 EMP 应用加载到监控平台中，完成相应的监控管理功能。

EMP 平台提供了基于模型维护的可视化开发工具 IDE，IDE 采用嵌入式 Eclipse 技术，构建在 Eclipse 平台之上，与用户的开发环境能够良好的集成。

2.1.2. EMP平台的特点

2.1.2.1. 稳定、高效、快速实施

EMP 平台从开发、运行、监控管理三个方面来帮助用户快速实施、高效稳定运行和轻松管理应用。



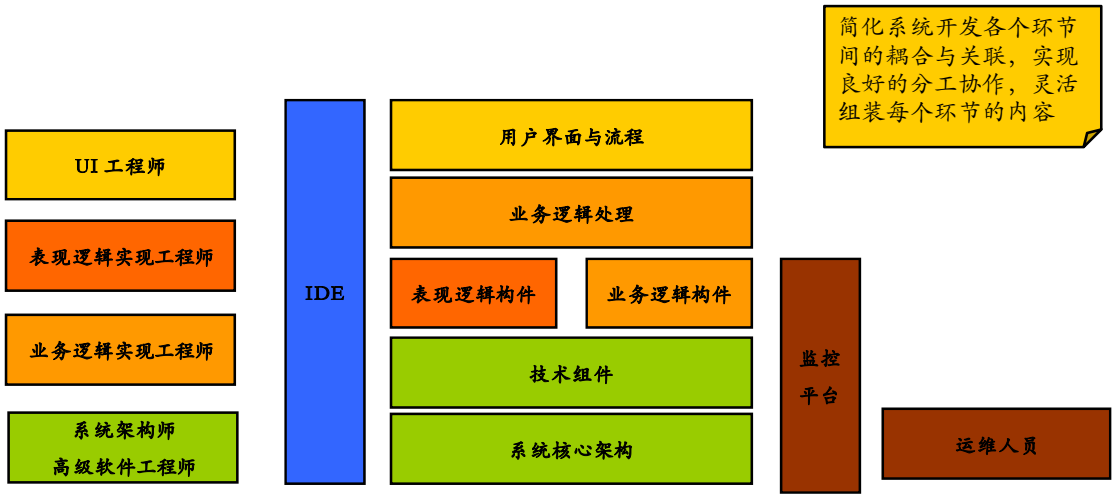
EMP 平台从发展到今天，一直努力跟进软件技术的发展前沿，EMP 核心框架和运行平台经过无数银行项目的检验是稳定和高效的，EMP 平台提供了大量的组件和构件，并通过可视化的开发工具实现组装式开发，可以快速实施和部署 EMP 应用，EMP 平台的所有组件和构件都可以通过外部化配置文件的方式直接开放为 JMX MBean 对象，基于 JMX 的监控管理平台可以与 EMP 应用轻松集成，完成监控管理功能。

2.1.2.2. 充分解藕能力

EMP 平台的设计采用了分层式设计，各层之间充分解藕，接口扩展和 XML 文件配置方式给 EMP 平台上的应用开发和功能扩展提供了良好的技术支持和实现。

EMP 平台作为一个金融行业的专业应用平台，不但从技术角度上考虑到系统设计上的分层解藕，更是从开发流程和人员分工上提供了充分解藕能力。

如下图所示：



从银行人员分工来说，基本可以将应用系统相关人员分为以下几个角色，他们与 EMP 平台之间的分工合作关系描述如下：

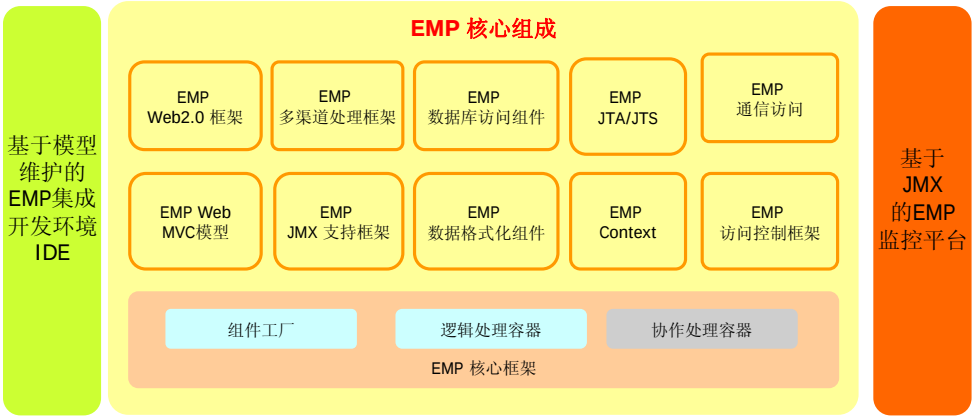
- 系统架构师：主要负责应用系统架构设计并提供系统应用的核心组件的设计。EMP 平台提供了基础的系统应用框架，系统架构师可以采用 EMP 平台框架作为系统应用框架，并在此基础上进行核心组件设计。
- 高级软件工程师：主要负责系统核心组件的设计和实现。EMP 平台提供了基础技术组件和组件扩展接口和注入机制，高级软件工程师可根据应用需要在 EMP 平台所提供的接口规范上进行核心组件的设计和开发实现。
- 业务逻辑实现工程师：负责应用系统的业务逻辑实现，如交易流程，报文组织，通信服务等等。EMP 平台采用基础组件+xml 参数配置的方式来实现业务流程和数据配置。EMP 平台无法提供的组件将通过高级软件工程师对核心组件的扩展来提供。业务逻辑实现工程师原则上是不尽兴编程的，他主要是通过 IDE 工具对 EMP 提供的组件进行参数配置和流程配置，完成业务逻辑处理。
- 表现逻辑实现工程师：负责应用系统表现逻辑的实现。如 Jsp 页面的输入，提交、页面跳转的流程配置等。EMP 平台的 Web MVC 框架为用户提供了 jsp 页面流程的管理和页面提交/响应的处理封装。EMP IDE 工具对快速开发和配置 Web MVC 处理提供了可视化的支持。
- Ui 工程师：负责应用系统界面设计。EMP 平台提供了页面布局管理框架和菜单处理组件，借助于 EMP IDE，可以帮助 UI 工程师快速构建应用系统的界面框架和菜单表现。

- 运维人员：对应用系统上线的系统运行情况进行监控管理，系统参数维护，系统状态管理等等。EMP 平台提供了 EMP Monitor 监控平台帮助用户快速完成对 EMP 应用的监控功能的实现和管理。

人员的分工由平台提供良好的支持，松散藕合的能力使得应用系统的开发清晰有序，特别适合于大型应用系统和具有延续性的项目的开发实施。

2.2. EMP平台架构与模型

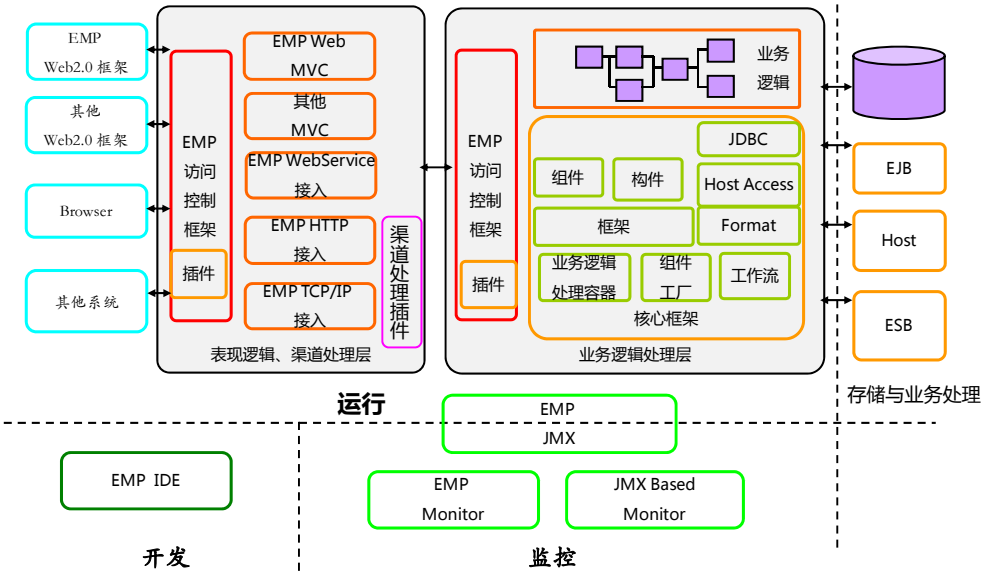
e-Channels EMP 核心平台是以组件及参数化技术驱动的层次化体系架构。



EMP 平台的核心由三大部分构成，分别是运行平台、开发平台和管理平台。这三大部分也构成基于 EMP 平台的应用解决方案的完整产品模型。

EMP 逻辑架构是一个分层设计模型。具体可分为业务逻辑处理层和表现逻辑处理层。两个层次上分别提供了业务逻辑处理框架和表现逻辑处理框架来提供功能实现。

如下图所示：



2.3. EMP开发平台IDE

2.3.1. EMP IDE概要介绍

EMP IDE 为 EMP 产品提供了专业化的开发环境，从项目的构建、建模、设计、测试、运行等多个角度提供支撑功能，通过可视化友好的操作界面来屏蔽系统复杂度，促进应用系统的快速建立。

EMP IDE 采用 Eclipse 插件技术，可集成于 Eclipse 中，与 J2EE 开发和 Web 开发一起形成 All in one 的集成开发环境。

EMP IDE 为 EMP 的应用项目提供了建设方法和开发规范，由此保证应用开发的有序进行和高质量。

2.3.2. IDE开发的典型过程

- **需求分析阶段**

采用用例分析（USECASE）方法

- **系统设计阶段**

通过 EMP IDE 建立数据模型：包括数据字典，公共数据定义，数据库和应用系统的数据映射等。

通过 EMP IDE 设计总体处理流程

组件设计，通过 EMP IDE 进行属性封装

- **详细设计和开发阶段**

通过 ECLIPSE 实现组件开发与调试

通过 EMP IDE 定义业务逻辑处理构件

通过 EMP IDE 生成业务逻辑处理构件的测试用例，并完成单元测试

通过 EMP IDE 定义表现逻辑处理构件

通过 EMP IDE 定义 JSP 页面和主布局页面

通过 EMP IDE 生成构件规格文档

- **测试阶段**

通过 ECLIPSE 的环境，直接完成单元测试与功能测试

测试过程管理工具(EMP Bug Manager)

CVS 的代码版本管理

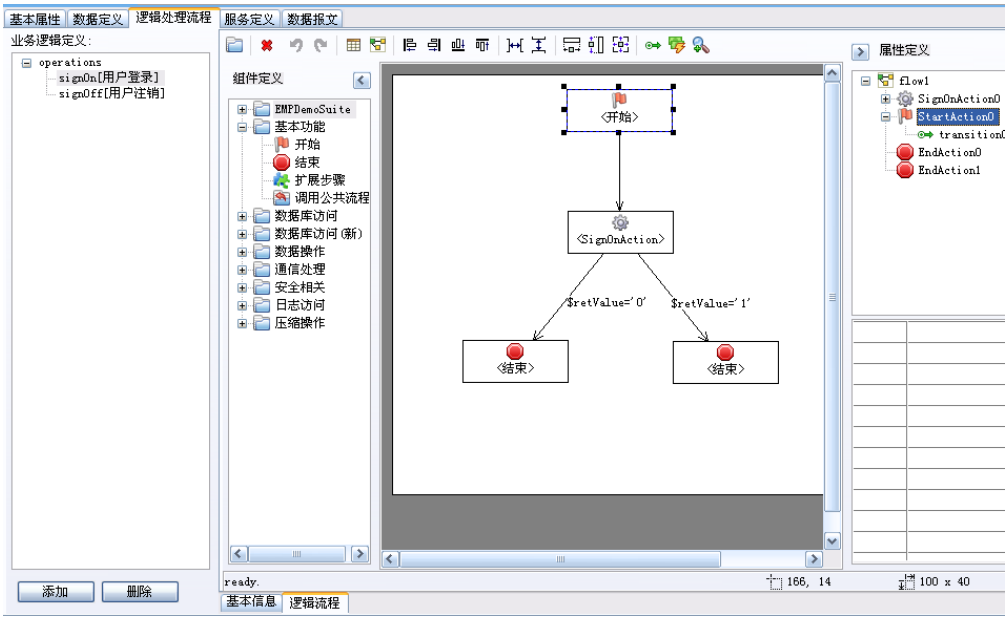
2.3.3. EMP IDE的构成

- **EMP Explorer**

EMP Explorer 提供了强大而便捷的操作视图，根据分类有序的组织 EMP 应用开发的相关部件，用户也可以根据该视图来查看部件间的关联性，并可进行部件的增删操作。

- **模型定义器**

EMP IDE 提供多种模型定义器，定义器通过可视化、拖拽等操作，将各种技术组件和业务构件组装为与具体应用相关的设计模型。



- **模型编译器**

当通过模型定义器完成部件模型的设计后，EMP IDE 自动调用编译器将设计模型转化为 EMP 运行文件。在编译过程中，编译器还将对模型内容进行合法性检查，如果存在关联错误，则显示在 Eclipse 的问题视图中。

- **文档生成器**

EMP IDE 可以根据设计模型生成相关的说明文档（Word 文档格式）。

- **自动测试套件**

EMP 自动测试套件可以针对业务逻辑部件进行单元测试并形成测试报告，该套件结合了主流的单元测试框架 JUnit，根据业务逻辑构件的定义辅助生成测试用例，并自动生成测试报告。

2.4. EMP监控平台

2.4.1. EMP监控平台介绍

JMX 功能是 EMP 平台提供新一代 J2EE 体系下的监控管理框架的实现。

EMP 平台对监控管理支持上提供了监控服务器实现和监控客户端实现。

监控服务器实现的功能主要是将 EMP 应用开放为标准的 JMX 服务器，将 EMP 组件开放为标准的 JMX MBean 对象共外部 JMX 客户端应用访问。

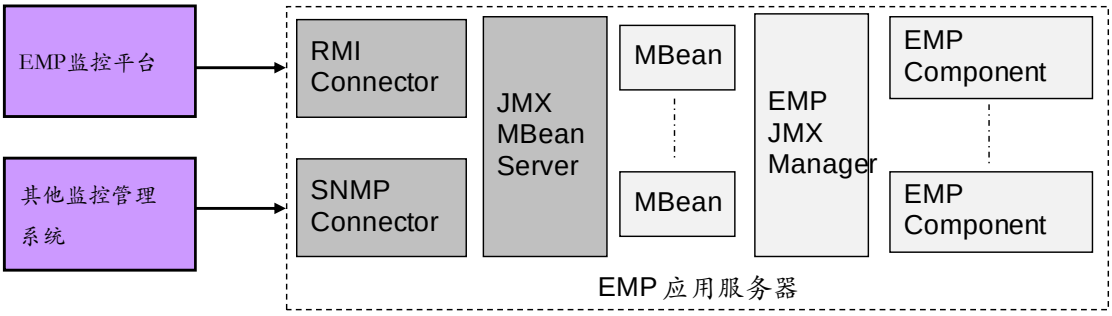
监控客户端是一个基于 EMP 平台上实现的一个 web 应用，可用来提供与任意 JMX 服务器对象进行连接，并按照一定规范展现 JMX 服务器上可监控对象信息。并可根据监控需要在客户端进行数据信息采集和监控报警设置等处理。

JMX 框架的设计初衷是监控应用不应该影响到被监控应用的组件对象。传统的监控设计总是需要从被监控组件中设置监控接口以便获得所需的监控信息。这些需要被监控组件去扩展的监控接口一旦加入到应用系统中，就将破坏被监控系统设计的完整性，提高应用系统与监控系统之间的耦合度，相应地就会为后期升级或系统维护带来问题。

JMX 采用将需要监控的组件对象进行外部封装，通过封装将被监控对象的实际方法或属性开放出来能够让外部系统访问。这种机制保证了被监控对象是无须知道自己被监控，也不必为监控做任何的组件升级。这也是 JMX 框架的价值所在。

EMP 平台在平台内部缺省内置了可配置的 JMX 服务器对象，可以快速将 EMP 平台组件封装为可用的 MBean 对象并放置在内部的 JMX 服务器上共外部监控系统访问。

EMP 为实现 JMX 框架进行了如下设计：



如上图所示，JMX MBean Server 是 JMX 框架所提供的 JMX 服务器，EMP 平台只需要将 EMP Component 通过 EMPJMXManager 对象进行封装生成对应的 MBean 对象，并将这些 MBean 对象在 JMX 服务器上注册即可实现 EMP 平台对 JMX 框架的支持。

RMI Connector 与 JMXMP Connector 组件是 JMX 框架提供的外部应用系统远程访问接口，在 EMP 平台可通过配置来定义平台应用内置的 JMX 服务器对象和外部访问的连接器对象。

2.4.2. EMP监控平台基本功能

EMP 监控管理平台提供如下系统监控和业务监控两大类功能。

模块	功能点	功能描述
系统监控	服务器操作	启动应用服务
		停止应用服务
	服务器集群管理	机群状态查询
	系统状态监控	内存、数据库连接数、通信连接池状态、CPU 消耗值等
	应用参数修改	查询和修改应用系统开放的应用参数
业务监控	可定制的业务信息查看	实时交易信息、交易并发统计信息
	交易状态管理	交易限额、可用性状态、最大并发数管理等
	定时数据采集	可定制的数据采集器
	报警功能	可定制的报警处理器和报警条件
	报警通知功能	可发送 JMS 报警消息，可发送 email、短信等报警信息（需要其他相关消息发送平台支持）

3. EMP核心与框架

3.1. EMP 组件工厂 – IOC容器

3.1.1. IOC容器介绍

组件工厂模式是 EMP 平台的技术核心构建模式。EMP 平台通过外部化 xml 文件形式对 EMP 应用组件进行定义和配置。在运行期间通过组件工厂类进行 EMP 应用组件的实例化处理。不同的应用框架和组件可通过扩展 EMP 组件工厂的方法来获得支持。

EMP 组件工厂模式采用了 java 的 IOC 技术来实现的。EMP 的核心是一个轻量级的容器，实现了 IoC（Inversion of Control）模式的容器。

EMP 组件工厂采用 IOC 容器使用，将对象之间的依赖关系通过外部化配置文件的方式来进行表达。在对象创建和使用的过程中，根据配置文件中的描述关系，由容器主动提供满足关联条件的组件对象。

EMP 平台采用 IOC 容器作为其核心的组件工厂，为整个平台带来了一些重大的改变，这些改变体现在以下几个方面：

控制反转——EMP 通过一种称作控制反转（IoC）的技术促进了松耦合。当应用了 IoC，一个对象依赖的其它对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。你可以认为 IoC 与 JNDI 相反——不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。

容器——EMP 包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，你可以配置你的每个 bean 如何被创建——基于一个可配置原型（prototype），你的 bean 可以创建一个单独的实例或者每次需要时都生成一个新的实例——以及它们是如何相互关联的。EMP 将这种依赖关系的描述放在外部化的 xml 配置文件中，使用组件的维护和修改更加的轻松。

框架——EMP 可以将简单的组件配置、组合成为复杂的应用。在 EMP 中，应用对象被声明式地组合，典型地是在一个 XML 文件里。EMP 也提供了很多基础功能（事务管理、持久化框架集成等等），这些基础的功能最大的作用是帮助用户在开发一个应用时能够忽略掉大量复杂的技术细节，而更加关注于应用逻辑的开发。这些技术细节的高质量和稳定性以

及通过简单方式配置使用的特性也正式 EMP 平台框架的重大价值所在。从根本上能够帮助用户提供应用的稳定性和质量，并为高效率的开发提供保障。

3.1.2. 组件工厂——XML定义与实现类

EMP 平台采用 XML 外部化配置文件来定义 Java 组件。

在 EMP 组件工厂中，每一个 XML 定义元素（XML Tag）都对应一个 Java 实现类。在组件配置文件中可以采用三种方式来定义一个 XML 定义元素所映射的 Java 实现类名称。

一、 在具体使用组件时，进行类名称声明

```
<fString id=“abc” dataName=“aaa” implClass=
“com.ecc.emp.format.string.FStringFormat”/>
```

在上述的例子中，通过 implClass 属性值来显式的声明该 XML 定义元素 fString 所对应的实现类。在 XML 解析时，将采用该类进行对象实例化。

二、 通过 XML 定义文件中<classMap>元素声明

```
<classMap>
    <map id=" fString" class=" com.ecc.emp.format.string.FStringFormat "/>
    …
</classMap>
```

如上示例，在每一份 XML 配置文件中，都可以加入 classMap 的标签项进行统一的类声明处理。

进行了类声明处理的标签，可以在 XML 配置文件中直接使用，而不需要进行‘implClass’属性的设置。

EMP 组件容器实例化处理寻找类映射的顺序是：首先在 implClass 属性中寻找类声明，否则在 classMap 定义中寻找类声明。

三、 **一种特殊声明方式：**业务处理逻辑容器中可在公共文件中声明，以便公用该声明

这种声明处理方式，只在业务处理逻辑容器处理中使用，通过一份公共的配置文件“settings.xml”中，集中进行在业务逻辑处理中可能需要公共使用的 XML 定义元素。产生

这种处理方式的原因是，前两种声明方式都只在被声明的配置文件中才产生作用，如果在其他配置文件中需要使用声明的定义元素，则必须重新进行类声明定义。而在业务逻辑处理中，业务逻辑流程的配置文件大多采用独立的配置文件进行描述，在每一份独立的业务逻辑配置文件中，可能要重复用到大量的业务逻辑处理元素，如 Action，service，format 等等，通过采用一份公共的配置文件来定义公用的 XML 元素的方式来共享 XML 定义元素，避免重复地在每一份配置文件中都需要进行 XML 定义元素声明，提高配置效率。

3.1.3. 组件工厂——XML定义中的属性注入

在 EMP 组件工厂中，每个 XML 定义元素对应于一个 Java 类。XML 文件的属性定义将映射到 Java 类的属性值。

每个 XML 定义的属性定义，组件工厂将通过查找 setter 方法的方式将属性定义注入到实现的类中

```
<fString id=“abc” dataName=“aaa”append=“true” class=
“com.ecc.emp.format.string.FStringFormat”/>
```

如上所示，EMP 组件工厂将通过类 FStringFormat 的方法 setDataName(String value)将属性 dataName 的值注入，对于标准的 Java 类型，容器会通过标准属性转换器转换，然后注入，如 boolean、Int 型属性，上例中将调用 setAppen(boolean value)方法。

另外，如果属性的取值比较长，或者有比较多的 XML 保留字符，则可以用子节点的方式来注入属性：

```
<DynamicSQLDefine id="DynamicSQL_Velocity" iCollName="userIColl" sqlType="select"
        <SQLStr><![CDATA[
SELECT * FROM USERINFO WHERE 1=1
#if ($_DATA.get('id')) AND ID=?id #end
#if ($_DATA.get('name')) AND NAME=?name #end
#if ($_DATA.get('address')) AND ADDR=?address #end
        ]]></SQLStr>
</DynamicSQLDefine>
```

如上，如果 SQLStr 子标签没有用任何一种方法定义它的生成类，则试图查找和调用父标签 DynamicSQLDefine 的属性注入方法 setSQLStr。注意这种方法只能用于参数类型为 String 的 setter 方法。

3.1.4. 组件工厂——依赖注入

EMP 组件工厂提供了基于 XML 节点的父子节点结构形式的组件组装。一个复杂组件通过父子结构的 XML 配置节点来描述其相互关系，并通过 EMP 组件工厂在实例化时自动进行组装。

```
<RecordFormat id=”testFormat”>
        <fString id=“abc” dataName=“aaa”append=“true” implClass=
“com.ecc.emp.format.string.FStringFormat”/>
        <fString id=“bbb” dataName=“bbb”append=“true” implClass =
“com.ecc.emp.format.string.FStringFormat”/>
</RecordFormat>
```

如上表格中示例，RecordFormat 与 fString 元素是父子节点关系。它们实例化后，实际上是构成了一个 RecordFormat 的复合对象，该对象中包含两个 FStringFormat 对象。

对于定义在 XML 节点中的子节点定义，组件工厂将通过如下规则将子节点实例注入到父节点实例中：

首先查找 set+className 的方法，如果成功，则调用，否则查找 add+ClassName 方法调用。

如果都不成功，则查找 set/add + beanTagName 的方法调用。

如果都不成功，则查找 set/add + superClassName 的方法调用。

如果还不成功，则查找 set/add + interface 的方法调用。

如果还不成功，则查找 set + 'name'的方法调用,其中 name 为在 tag 中定义的 name 属性值。

3.1.5. 组件工厂——继承关系

EMP 组件工厂在某些需要多次实例化，且有较多的公共属性的 Bean,在定义时可以通过继承关系，实现统一定义。

```
<pElement id=“pe” attr1=“abc” attr2=“aaa” class=“com.ecc.emp.Testbean
“>
        <a id=“abb” name=“ddd”…/>
</pElement>
<p1 id=“p1” parent=“pe” attr2=“bbb” attr3=“bcd”>
</p1>
<p2 id=“p2” parent=“pe” attr3=“bbb”/>
```

实例中 p1 定义将得到 TestBean 的实例，且属性：id=p1 attr1=abc attr2=bbb attr3=bcd，具有 a 的实例注入。

实例中 p2 定义将得到 TestBean 的实例，且属性：id=p2 attr1=abc attr2=aaa attr3=bbb。

3.1.6. 组件工厂——特殊属性注入

组件工厂为一些特殊属性注入提供了全面支持，这些特殊属性主要有列表属性和散列属性的注入支持。

3.1.6.1. List属性注入

有时候需要为某个组件设定 List 类型的属性，比如提供一系列的 String 可选值，EMP 组件工厂提供了对这类熟悉的支持。

```
<a id=“a” class=“com.ecc.emp.TestBean”>
    <List name=“validValueList” class=“java.util.ArrayList”>    <ListEntry
value=“abc” class=“com.ecc.emp.component.xml.ListEntry”/>
        <ListEntry value=“value2039” class=
“com.ecc.emp.component.xml.ListEntry”/>
    </List>
</a>
```

上述的定义 EMP 组件工厂将通过 TestBean 的 setValidValue(List value)方法设定属性 validValue，值包含:”abc”、“value2039”。

ListEntry 也可以是其他实现类，并不规定必须是 String。

3.1.6.2. Map属性注入

有时候需要为某个组件设定散列属性，也就是一系列 Key 索引的对象，EMP 组件工厂也对此支持。

```
<a id=“a” class=“com.ecc.emp.TestBean”>
    <Map name=“validValueMap” class=“java.util.HashMap”>
        <MapEntry key=“key1” value=“abc” class=
“com.ecc.emp.component.xml.MapEntry”/>
        <MapEntry key=“key2” value=“value2039” class=
“com.ecc.emp.component.xml.MapEntry”/>
    </Map>
</a>
```

上例中，容器将通过 TestBean 的 setValidValueMap(Map value)方法注入 Map 类型的属性。

3.1.6.3. 特殊Map属性注入

特殊 Map 属性是指以 JavaBean 作为 Map 的元素。

EMP 组件工厂将按照：getId getName getKey 的顺序，从定义元素类实例中获得 key 值，然后将此实例 put 到 Map 中。

```
<a id=“a” class=“com.ecc.emp.TestBean”>
    <Map name=“validValueMap” class=“java.util.HashMap”>
        <MapEntry key=“key1” attr1=“abc” class=“com.ecc.emp.TestMapBean”
/>
        <MapEntry key=“key2” ttr1=“value2039” class=
“com.ecc.emp.TestMapBean”/>
    </Map>
</a>
```

上例中，容器将通过 TestBean 的 setValidValueMap(Map value)方法注入 Map 类型的属性。

类 TestMapBean 必须提供 getId 或 getName 或 getKey 方法，提供 Key 值。

3.2. EMP 业务逻辑处理容器

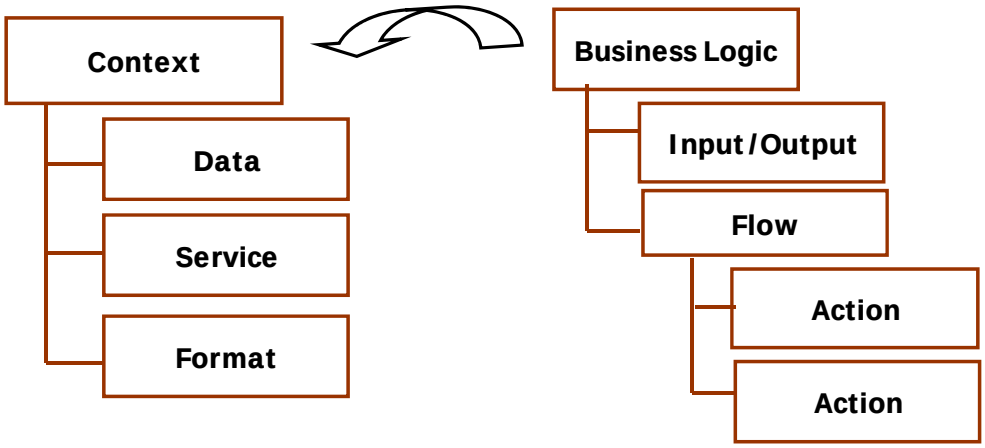
3.2.1. 概要介绍

EMP 业务逻辑处理框架是 EMP 的核心组成，它通过对 J2EE 应用系统中具体应用逻辑处理的抽象，实现具体应用逻辑处理的外部参数化（XML）配置实现机制，成功实现可扩展的基于组件化的应用系统开发模式，实现最大化的组件重用。

通过应用逻辑处理容器的应用逻辑处理方式，可以较好的解决实际应用逻辑的灵活性和维护要求的一致性的问题，同时通过这种逻辑处理方式，能够积累大量的通用的应用逻辑处理流程和组件，新增加的应用逻辑处理的开发将会更加快速，并且开发的质量更加有保证。

3.2.2. EMP业务逻辑处理模型

EMP 业务逻辑处理模型通过将现实中的业务逻辑处理对象进行要素分解，形成由 7 大核心要素构成的松耦合结构的逻辑模型，如下图所示：



Context——EMP 业务逻辑处理器中的资源访问与管理器，用于管理每个逻辑处理器需要用到的资源，以树形结构管理公共资源

BusinessLogic——具体的业务逻辑单元，包括了多个定义了输入/输出访问接口及处理流程的业务处理逻辑

Data——业务逻辑处理器中对数据对象的抽象

Format——数据报文格式处理器，用于将数据对象转换为相关的报文如 ISO8583 报文，或反之

Service——完成特定功能的功能组件，如：数据库表访问，后台通信处理等

Flow——具体的业务逻辑处理流程，它是由一系列基本操作步骤 Action 的逻辑组合而成，完成具体的业务处理流程

Action——具体的业务操作单元

可以从两个方面来理解业务逻辑处理模型。

首先从业务流程描述来理解：一个业务逻辑单元对象 BusinessLogic 是一个具体的业务逻辑处理入口。一个 BusinessLogic 对象中有自己完整的输入输出接口，可包含多个可执行的具体的业务逻辑处理流程 Flow。Flow 通过具体的业务操作单元 Action 进行组合产生具体的业务逻辑流程。

再从业务流程处理所涉及到的资源管理来理解：BusinessLogic 通过 Context 资源结点来组织数据和其它服务资源。Context 结点是业务逻辑所需要的所有数据资源和服务资源的管理对象，它通过在运行时注入到业务逻辑中完成数据处理。

3.2.3. 业务逻辑处理要素

3.2.3.1. Context资源结点

Context 对象是 EMP 业务逻辑处理容器中业务处理逻辑资源的管理者，各个模块通过它来访问和交换资源，如访问数据，获得服务。

Context 对象以树型结构组织，每个节点可以直接获得父节点及以上节点的资源

Context 对象实现类：com.ecc.emp.core.Context。

Context 对象作为 EMP 业务逻辑处理容器的资源管理者，对外提供了丰富的访问接口，在运行时系统可以通过这些访问接口来获得所需要的资源，如数据内容、服务对象、格式化对象等等。

Context 对象所提供的数据访问接口如下表所示：

类型	访问方法	方法说明
数据	getDataElement(String	得到 dataName 名称的数据定义

访问	dataName):DataElement	
	getDataValue(String dataName):Object	取 dataName 名称的数据域的值
	setDataValue(String dataName, Object value)	设定 dataName 名称的数据域的值为 value
	addDataElement(DataElement element)	添加数据域
	addDataField(DataField field)	添加数据域
	addDataField(String name, String value)	添加数据域
服务访问	getService(String serviceId):Service	得到一个 serviceId 为服务 Id 的服务对象
格式化对象访问	getFormat（String formatId）	得到一个以 formatId 为格式化对象 ID 的格式化对象
传递参数	setAttribute(String name, Object value)	设置一个参数的值
	getAttribute(String name)	得到一个参数的值

在应用运行时，系统中一般会存在 4 种 Context 对象，根据不同的情况进行动态挂接，从而使用户请求访问到各种资源。如下：

RootContext：存放全局公用的资源，如数据源服务、公告数据等；系统启动之后便会一直存在；

SessionContext：存放用户会话信息，如用户名、其他登陆信息等；在用户登陆时创建，签退时销毁；

ResultContext：存放表现层所使用的数据（Web 渠道下），一般在服务器接到请求时创建，并从 request 获取请求数据；在请求结束时用于页面的数据展现。在向导或多次请求的模式下，ResultContext 的生命周期也可以跨越多个请求-响应（暂存在 Session 中），从而在相关的功能模块中维持临时的公共数据。

BizContext：存放某个业务逻辑所使用的数据（及静态资源定义），在进入业务逻辑之前创建并挂接到 RootContext 下（为了访问公共资源），业务逻辑结束时销毁。在调用业务逻辑时会从 SessionContext 和该次请求的 ResultContext 中获得 biz 所需要的数据，执行完毕后再将数据写回。

Context 在运行时寻找资源是从本地结点首先查找是否存在资源对象，然后依序查找父结点寻找资源，直到找到资源或查找到顶级结点后结束。

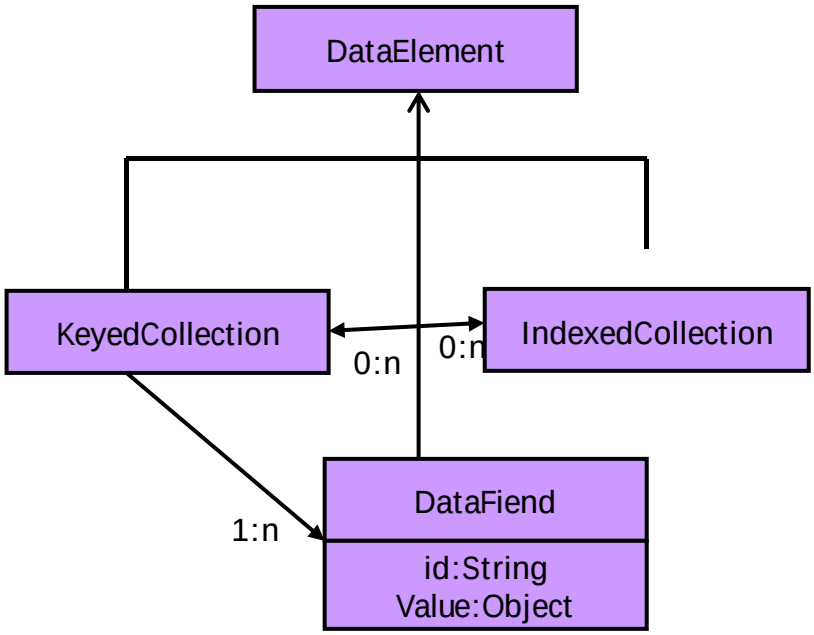
一个 Context 结点的 XML 描述：

```
<Context id=”testCtx”>
    <refKColl id=”testKColl”/>
    <refService refId=”testService”/>
</Context>
```

Context 结点下所包含的 refKColl 和 refService 分别为引用的数据对象和服务对象。这个两个对象可以通过 Context 结点来进行访问。

3.2.3.2. 数据定义

EMP 数据定义有三种类型：DataField、KeyedCollection、IndexedCollection，分别对应着单个数据、数据集合和数据列表集合三种。三种数据类型都继承于 EMP 平台的抽象数据定义类：DataElement。类实现关系如下图所示：



3.2.3.2.1. DataField基本数据定义

DataFiled 是指单个数据的定义，它的 XML 描述如下：

```
<field id=”testField” value=”test” type=”String”/>
```

一个 DataField 对象有自己的 Id，value 值和类型，类型如果不定义，则缺省为 String 类型，定义时，一般将 DataField 定义在一个 KeyedCollection 结构化对象中，以便在业务逻辑处理中统一进行管理。

DataField 在编程接口上主要提供了 getValue（）和 setValue(Object value)两个方法，分别用于获取数据值和设置数据值。

3.2.3.2.2. KeyedCollection结构化数据定义

KeyedCollection 是指集合型的对象，它可以包含多个 DataField、KeyedCollection 或 IndexedCollection 对象。我们用 KeyedCollection 对象来描述一个特定的业务对象，如账户对象，可能包含账号、账户名称、余额、币种等等数据域，这些数据域集中定义在一个 KeyedCollection 对象中，组合代表为一个账户的信息。KeyedCollection 对象的 XML 描述如下，其中标签 kColl 映射为 KeyedCollection 类实现定义名称。

```
<kColl   id=”accountKColl”>
        <field id="accountNo"/>
        <field id="accountName"/>
        <field id="openDate"/>
        <field id="accountRate"/>
        <field id="accountType"/>
        <field id="accountBalance" dataType="Currency"/>
        <field id="CurrencyCode"/>
        <field id="freezeBalance"/>
    </kColl>
```

KeyedCollection 主要提供的编程接口是对数据元素的操作，基本接口方法如下所述：

- getDataValue（String dataName）：得到一个数据值
- setDataValue（StringdataName，Object alue）：设置一个数据值
- getDataElement(String dataName)：得到一个数据对象
- addDataField(String dataName, Object value）：添加一个数据对象（有可能是一个 KeyedCollection 对象或者 IndexedCollection 对象）

- addDataDataField(DataField field)：添加一个数据对象

3.2.3.2.3. IndexedCollection结构化数组定义

IndexedCollection 对象是结构化的数据数组，包含多个重复结构的数据对象，如账户列表的描述，就只能采用 IndexedCollection 对象来进行描述。IndexedCollection 对象的 XML 定义如下：

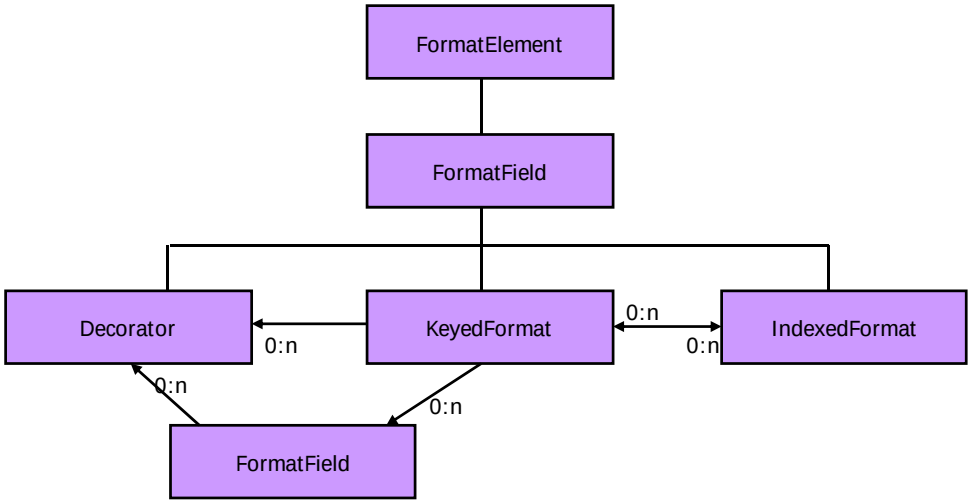
```
<iColl id="accountCollection">
        <kColl>
            <field id="accountNo"/>
            <field id="accountName"/>
            <field id="openDate"/>
            <field id="accountRate"/>
            <field id="accountType"/>
            <field id="accountBalance" dataType="Currency"/>
            <field id="CurrencyCode"/>
            <field id="freezeBalance"/>
        </kColl>
    </iColl>
```

以上描述的是一个账户列表的信息。IndexedCollection 对象提供的编程接口也是基于多数据对象的操作，由于它是基于数据数组模型的操作，所有有一些基于标记位的接口方法，如下：

- getElementAt(int idx) 取第 idx 条数据定义
- addDataElement（DataElement element)添加一条数据

3.2.3.3. 数据格式化

数据格式化对象是对 EMP 业务逻辑处理中的数据进行报文格式处理的对象。EMP 平台的数据格式化处理元素由以下类定义提供基本实现。



3.2.3.3.1. FormatField

FormatField 对象是所有 Format 定义的父类，定义了所需要的接口，主要访问接口包括：

- locate（Object src int offset):int： 在解析报文时用于定位本格式定义报文在 src 的位置
- extract（Object src, int offset):int ：确定在解析报文时本格式定义报文在 src 的长度
- public Object format(DataField dataField)：对一个数据对象进行格式化
- public void unformat(Object src, DataField dataField) ：将被格式化内容进行反格式化到数据对象中
- public Object addDecoration(Object src) ：对一个数据格式化处理增加一个修饰符对象
- public Object removeDecoration(Object src) throws：对一个数据格式化对象删除一个修饰符对象

通常用户需要实现的类需要继承此类，并实现上述接口即可。

以下是一个 FormatField 的 XML 描述：

```
<ISO8583Field dataName="accountNo" fieldType="AN" fieldState="M" fieldIdx="13"
IOType="IO" fieldLength="19"/>
```

上面显示的例子是一个 ISO8583 数据域的格式化定义。ISO8583Field 标签实际对应着一个继承于 FormatField 类的 8583 域的格式化处理类的名称。

3.2.3.3.2. KeyedFormat

KeyedFormat 对象是 EMP 中 Format 的集合，其内可以包含多个 FormatField、keyedFormat 及 IndexedFormat。KeyedFormat 对象用于处理结构化的数据格式化、反格式化，处理中将逐个调用其相应的处理方法实现数据报文的处理。

以下是一个 KeyedFormat 的 XML 描述：

```
<requestDataFormat id="requestDataFormat"
class="com.ecc.emp.format.FormatElement">
        <record>
            <fString dataName="userid">
                <delim delimChar="|"/>
            </fString>
            <fString dataName="password">
                <delim delimChar="|"/>
            </fString>
            <delim delimChar="#"/>
        </record>
    </requestDataFormat>
```

其中的 record 标签对应的就是 KeyedFormat 类的实现。它里面包含两个数据 userid 和 password 域，分别采用字符串+分隔符的方式进行格式化组包。Delim 是修饰符的一种，将在后面部分进行介绍。

3.2.3.3.3. IndexedFormat

IndexedFormat 对象是 EMP 中用于处理重复数据（如帐号列表，明细）类型的数据格式化，其内包含一个 FormatField。

以下是一个 IndexedFormat 的 XML 定义描述：

```
<iCollFmt dataName="accountCollection">
                <record>
                <fString dataName="accountNo">
```



```
                <delim delimChar="|"/>
            </fString>
            <fString dataName="accountType">
                <delim delimChar="|"/>
            </fString>
            <fString dataName="accountName">
                <delim delimChar="|"/>
            </fString>
            <delim delimChar="*"/>
            </record>
        </iCollFmt>
```

iCollFmt 标签对应的映射类即位 IndexedFormat 类。在该定义中包含一个 record 定义，指明该格式化需要对一个 IColl 进行格式化，对每一个列表数据进行 record 格式化处理，并以 delim 标签中定义“*”标记作为分隔符。

3.2.3.3.4. Decorator修饰符

Decorator 修饰符对象是 EMP 中数据格式化处理过程中负责对报文的修饰，比如：压缩，添加修饰符，MAC 等

每个 FormatField 可以包含多个 Decorator,处理时逐个调用，格式化时顺序调用，反格式化时倒序调用

接口包括：

- public Object addDecoration(Object src) ： 添加修饰
- public Object removeDecoration(Object src) ：去掉修饰
- public int extract(Object src, int offset) ： 定位数据

参考一个 fString 字符串格式化的定义：

```
            <fString dataName="userid">
                <delim delimChar="|"/>
            </fString>
```

Delim 标签就是修饰符的一种，该格式化处理得到的格式化后的内容如下：

假定数据域 userid 的值为：Admin，则格式化后的内容为：Admin|， ‘|’为修饰符内容。

3.2.3.4. 服务对象

服务对象是 EMP 平台提供可被调用的技术组件。通过配置处理，这些 Java 类将配置为一个一个的服务组件，在运行时可通过名称进行对象实例化并进行调用。

服务对象可以是 EMP 平台的 Service 接口的实现类，也可以是任意的 POJO 类对象，EMP 平台的组件工厂通过 IOC 技术可以很便捷的将这些类对象注入到组件容器中提供给系统访问。

所有的服务对象都通过 Context 结点进行管理，我们通过 XML 配置文件独立定义一个 Service 对象，但如果想在运行时调用，则必须将该 Service 定义在 Context 结点中进行引用，如下所示：

```
<TCPIPServerService name="aService" keepAlive="false" dual="false"
class="com.ecc.emp.tcpip.TCPIPServerService">
        <ListenPort id="listenPort" keepAlive="false" port="8010"
maxConnection="50" poolSize="1" poolThread="true" waitTime="2000"
class="com.ecc.emp.tcpip.ListenPort"/>
        <CommProcessor name="commProcessor"
class="com.ecc.emp.tcpip.EMPCommProcessor"/>
</TCPIPServerService>
```

以上是一个 TcPIP 访问服务的配置信息，但如果该服务想要在运行时获取，则必须在需要的 Context 对象中进行引用定义，如下：

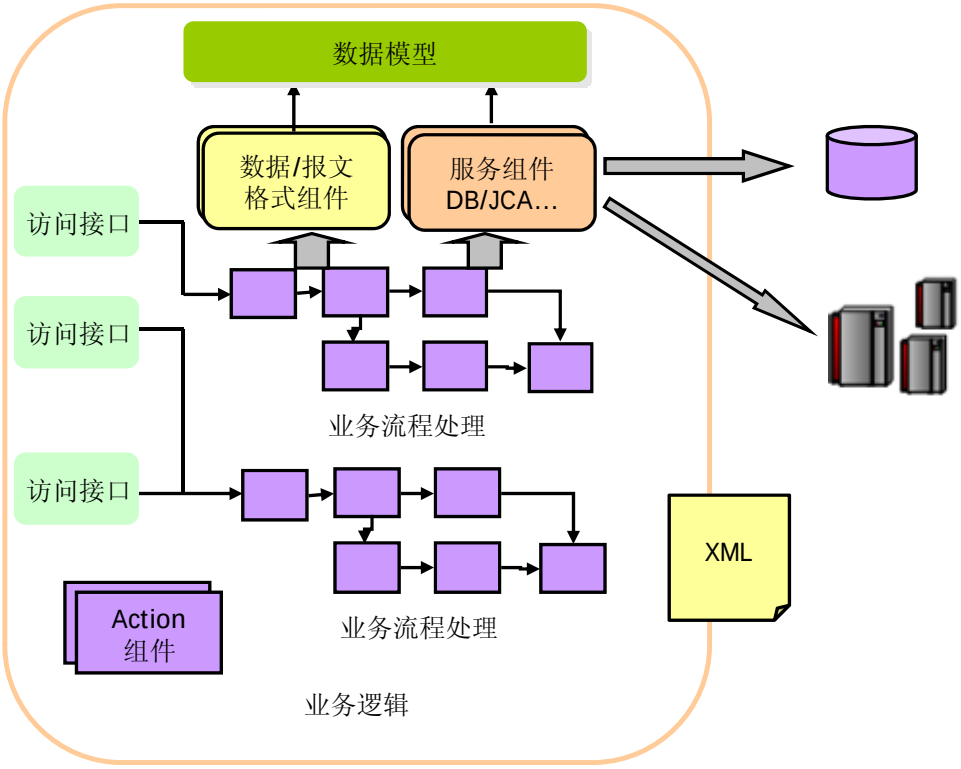
```
<Context id=”testCtx”>
    <refKColl id=”testKColl”/>
    <refService name=”aService” refId=” aService”/>
</Context>
```

Context 在寻找 Service 对象时（通过 Context .getService(name)方法），将通过 name 名称去寻找对应的 Service 配置对象。

3.2.4. 业务逻辑处理实现

3.2.4.1. 组件化参数驱动的业务逻辑处理

EMP 业务逻辑处理容器采用组件化参数驱动的方式完成业务逻辑处理。采用 XML 配置文件来描述业务逻辑处理流程和相关资源内容，运行时装载 XML 配置文件所设定的组件对象按照配置流程进行调用执行。



如上图，业务逻辑处理通过 XML 文件来描述流程的跳转，通过数据模型（即前一章节说明的 Context、数据定义、格式定义和服务组件）的访问来进行资源调用和数据交换。业务逻辑处理通过 Action 原子操作单元的组装来完成业务逻辑流程的定义。

如下所示，是一个标准的业务逻辑处理单元的 XML 配置文件示例：

```
<EMPBusinessLogic id="testEMPLogic" operationContext="testEMPLogicSrvCtx">
<operation id="signOn" name="签到">
<input>
<field id="UserID"/>
<field id="password"/>
</input>
<output>
   <field id="errorCode"/>
   <field id="errorMsg"/>
```

```
    <field id="sessionId"/>
    <iColl id="accountInfos">
        <kColl>
            <field id="AcctNo"/>
            <field id="Balance"/>
            <field id="AcctName"/>
            <field id="currencyCode"/>
            <field id="acctType"/>
        </kColl>
    </iColl>
</output>
<flow>
<action id="SignOnAction0" implClass="com.ecc.emp.test.actions.SignOnAction">
    <transition dest="END0" condition="$retValue='0'"/>
    <transition dest="END1" condition="$retValue='1'"/>
</action>
    <action id="END0" result="0" errorMsgField="errorMsg"
implClass="com.ecc.emp.flow.EMPEndAction" errorCodeField="errorCode"
errorCode="0"/>
    <action id="END1" result="2000" errorMsgField="errorMsg"
implClass="com.ecc.emp.flow.EMPEndAction" errorCodeField="errorCode"
errorMessage="signOnError" errorCode="2000"/>
</flow>
</operation>
</EMPBusinessLogic>
<context id="testEMPLogicSrvCtx" type="op">
        <refKColl refId="testEMPLogicSrvData"/>
</context>
<kColl id="testEMPLogicSrvData">
        <field id="UserID"/>
        <field id="password"/>
</kColl>
```

第一句定义了一个 BuzinessLogic 对象，它的 id 为 testEMPLogic，它可以包含若干个 Operation，在本例中只包含了一个 Operation，名称为 signOn。一个 Operation 代表一个可以独立访问的业务操作，它主要包含了针对该业务操作的输入输出接口（input，output）和业务逻辑流程定义（flow）。

Input 定义的数据代表了该业务操作职能接收 input 中定义的数据，如果送来的数据不在 input 定义范围内，则将拒绝该数据。

Output 定义的数据代表了该业务操作执行成功后可能返回的数据结果。如果外部访问想获得超出 output 定义范围内的数据，应用将拒绝。

Flow 是该业务操作的流程定义，它由一系列 Action 组合而成，每个 action 之间通过 transition 对象定义的条件进行流程的跳转。

每一个 BuzinessLogic 都有一个 context 资源结点（前述 BizContext），用来管理它的专有资源，如数据、服务或格式化对象等等。在该例中，context 名称为 testEMPLogicSrvCtx，它只管理了一个数据集合对象：testEMPLogicSrvData。

以上描述构成了一个完成得 BuzinessLogic 的定义。在描述一个业务逻辑处理时，EMP 平台提供了 flow、action 和 transition 对象来构造业务流程的描述。

BuzinessLogic 是与会话无关的，它只关注自己所需要的数据，以及公共服务。因此具有良好的构件特性，可以将各个渠道发来的请求进行无差别处理，包括可以将自身发布为 WebService。

3.2.4.2. 业务逻辑处理基本要素

3.2.4.2.1. 业务逻辑流程中的Flow和Action

Flow 是业务逻辑流程的描述。在运行时是我们执行的对象。Flow 由一系列业务操作单元 Action 组合而成。

Action 是指具体的业务操作单元，完成特定的处理步骤，如向数据表写入一笔记录，发送一笔交易到特定主机等。Action 是流程逻辑中最小的重用单元，EMP 平台提供了大量的通用的 Action，如数据库操作、文件操作、通信操作、数据转换操作、数据安全操作等等。通过对这些 Action 进行参数配置和状态迁移来组装产生各种业务需要的处理逻辑流程。

每一个 Action 执行结束后都返回一个字符串，EMP 平台可以通过返回的字符串来获取 Action 的执行结果。

EMP 平台提供了抽象类 EMPAction 作为 Action 扩展的接口，EMPAction 提供了抽象方法：String execute(Context context)作为执行时的接口方法。应用需要扩展 Action 时，只需要实现继承该抽象类并实现其接口方法即可。

```
<action id="SignOnAction0" implClass="com.ecc.emp.test.actions.SignOnAction">
    <transition dest="END0" condition="$retValue='0'"/>
    <transition dest="END1" condition="$retValue='1'"/>
</action>
```

上表为一个 Action 的标准配置示例。每一个 Action 有它的 id，同时通过 implClass 属性来指定它的实现类，如果有其他属性可以配置在其后。

Action 的执行结束后，将通过检查 transition 对象的条件来判断应该转到哪一个 Action 继续执行或者终止执行。Transition 对象是 EMP 业务逻辑处理容器提供的流程跳转条件对象，上面的例子中：condition= “$retValue='0'” 就是一个跳转条件。RetValue 是系统得一个保留字段,它表示为一个 Action 执行结束后的返回字符串。途中的公式表示为：当该 action 执行返回值为 0 时，则执行‘END0’操作步骤，如果返回值为‘1’时，则执行‘END1’操作步骤。

3.2.4.2.2. 流程跳转：Transition

状态迁移 Transition:由 Action 的执行结果、或执行后的数据内容决定下一个步骤，完成逻辑判断，判断采用表达式定义的方式完成。

```
<transition dest="callBusinessLogic" condition="$retValue='0'"/>
```

表达式中特定的变量：retValue 代表本步骤执行的返回值，在 EMP 中他是一个字符串值。

EMP 平台的条砖条件语法并不单单只是根据 action 的返回值来判断跳转条件，而是可以采用业务逻辑单元相关的数据（前题是该数据域可以通过该业务逻辑操作得到，也就是说可访问的数据范围为本业务操作单元所定义的数据域、以及用户相关的 Session 数据域和应用共享的根数据域对象）通过基本的表达式语法来组成判断公式进行跳转。

基本表达式语法请参考表达式介绍章节。

3.2.4.3. For循环逻辑

EMP 处理中允许定义 For 循环执行的业务逻辑流程。基本配置方式如下：

```
    <For id="for1" cycleCounter="0" cycleCondition="$counterValue &lt;
@SizeOfICollection($accountInfos)" cycleUpdator="$counterValue + 1" >
        <action id="getCollection" iCollName="accountInfos" collectionName="iAcct"
cycleId="for1" class="com.ecc.emp.processor.GetKCollFromICollByCycleIndex"/>
    <transition dest="END0" condition="$retValue='1'"/>
```

```
<transition dest="END1" condition="$retValue='0'"/>
</For>
```

上例中定义了一个根据 IndexedCollection 的大小循环执行 getCollceion 的例子。

- 循环变量： cycleCounter。初始值是一个表达式定义，通过表达式给他赋初始值。
- 退出条件： cycleCondition。也是一个表达式定义
- 变量循环： cycleUpdator。同样也是表达式，用于更新循环变量。
- 表达式中的特殊变量： counterValue 代表循环变量的值。
- For 循环中定义的 Action 实际上是指 for 中的操作。在 for 循环中可以包含多个可执行的 Action
- Transition 是指 for 操作结束后的跳转条件

3.2.4.4. Switch/Case逻辑

EMP 处理中允许定义 switch/case 条件判断执行的业务逻辑流程。

```
<switch id="sw1>
    <case id="cs1" caseCondition="$iAcct.acctType = '01'">
        <action id="showData1" message="Type 01 DataValue "
dataName="iAcct.AcctNo" />
    </case>
    <case id="cs2" caseCondition="$iAcct.acctType = '02'" >
       <action id="cs2ac" message="Type 02 DataValue " dataName="iAcct.AcctNo" />
    </case>
    <case id="cs3" caseCondition="$iAcct.acctType = '03'" >
        <action id="cs3ac" message="Type 03 DataValue " dataName="iAcct.AcctNo" />
    </case>
</switch>
```

上例中定义了一个根据账户中帐户类型不同执行不同的流程的例子，在每个 Case 内允许执行一系列有逻辑关系定义（等同于流程定义）的逻辑步骤。

在每个 Switch/case 定义中，需要单独定义条件，满足条件则执行相应的流程。Switch 只是执行第一个满足条件的 case 内容。

3.2.4.5. 公共流程调用

应用系统开发中，往往存在一些处理流程是大家可以共用的，将这些公用的流程提取出来组成公共流程对象共系统共享可以有效地提高系统重用能力和降低系统复杂度。

EMP 平台从流程上根据应用需求的类型提供了两类公共流程调用的方式，一类为公共子流程对象，可以在自有流程中调用该公共子流程完成一项公共的功能，如用户密码校验的处理就是一个典型的公共子流程的应用。另一类为公共主流程的应用，系统提供公共主流程的定义，并在定义中根据应用的流程需求在不同的地方提供流程扩展点，所有的业务均采用此主流程作为业务主流程，在实际应用中根据自身业务流程的需要在流程扩展点上进行本业务所需要的特殊业务处理，以这种方式来共享主体的业务流程逻辑，如上送主机的交易流程就是一个主流程几乎相同的逻辑处理流程，只是在一些细节上，如记录日志、判断帐户等环节有所差异，这种情况下采用公共主流程的方式， 能很好地提高系统效率。

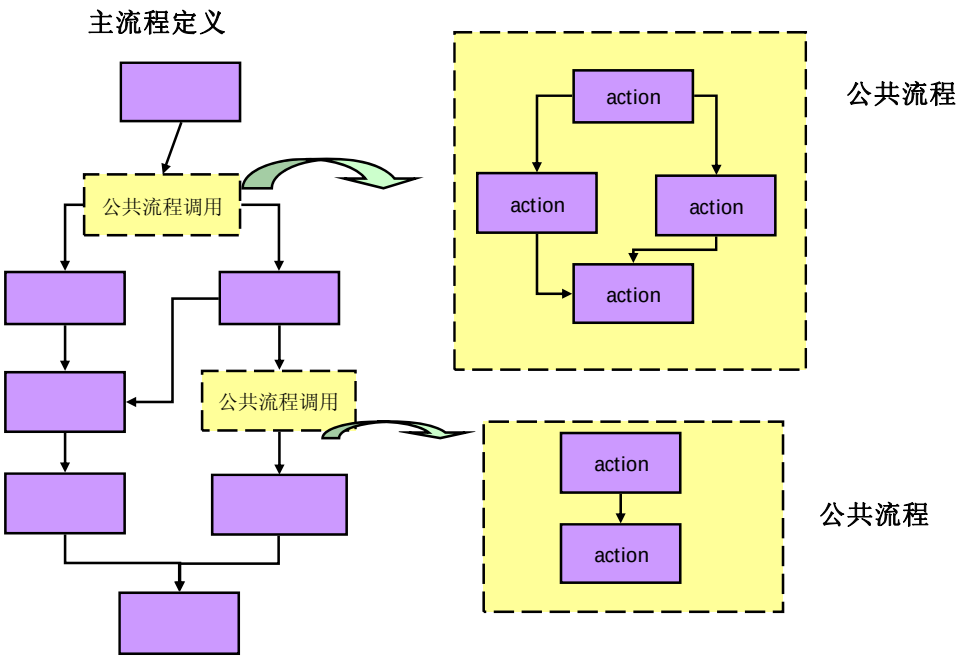
3.2.4.5.1. 调用一个公共流程

EMP 业务逻辑处理中，允许将一些很多地方都用到的一些流程抽取出来，供具体的业务流程调用，这样也有利于当流程变化时集中修改。

```
<action id=“callCommFlow” flowId=“comm1” class=
“com.ecc.emp.flow.CallCommFlow”/>
```

如上所示，公共流程的调用实质上也是一个 Action 的配置，这个 Action 是系统提供的 callCommFlow，通过配置 flowId 名称，即公共流程名称来调用指定的公共流程对象。

具体方式如下图所示：



在业务流程定义中，在需要调用公共流程的地方加 callCommFlow 的操作步骤，并指定所要调用的公共流程 ID 即可。

公共流程的定义在公共的配置文件中，其定义方式与普通流程定义完全一致。

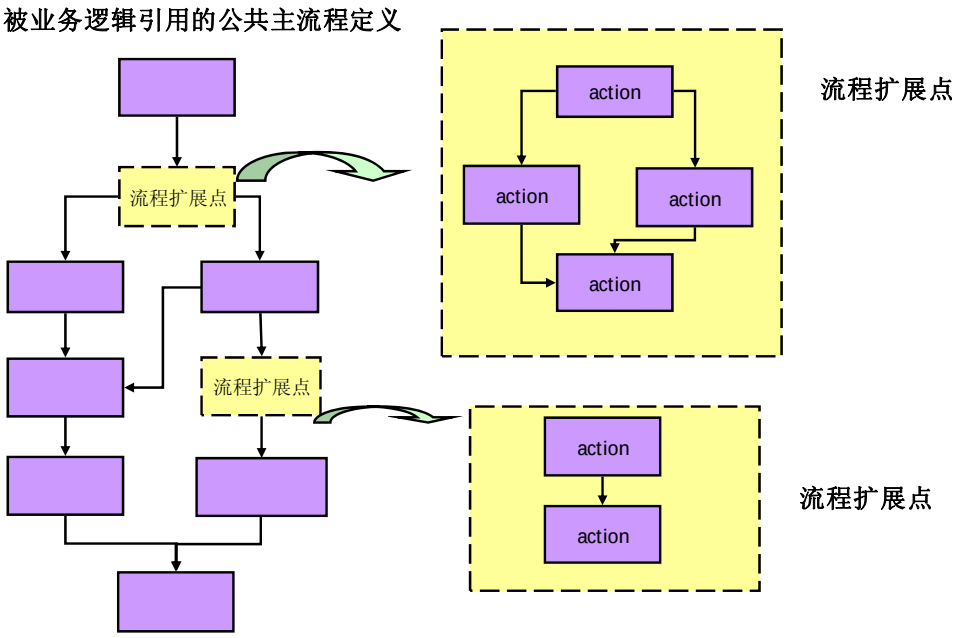
公共流程定义中不涉及到数据的定义，它默认要求公共流程与需要调用的业务的数据字典是一致的，所需要的数据在调用公共流程的业务逻辑对象中可以找到。

3.2.4.5.2. 公共主流程与流程扩展

在某些业务处理系统中，可能所有的流程都大致相当，只是有些局部的变动不同，针对这种情况，EMP 提供了公共流程继承与扩展点扩展的处理方式，具体的业务处理流程可以从公共定义继承，只是对预留的扩展点进行必要的处理即可。

```
<EMPFlow refFlowId=“comm1” >
<extendedAction extendId=“ex1” >
<action …
<action …
</extendedAction>
</EMPFlow>
```

上例中从公共流程 comm1 继承，扩展了其扩展点 ex1 的处理。公共主流程的调用方式如下图所示：



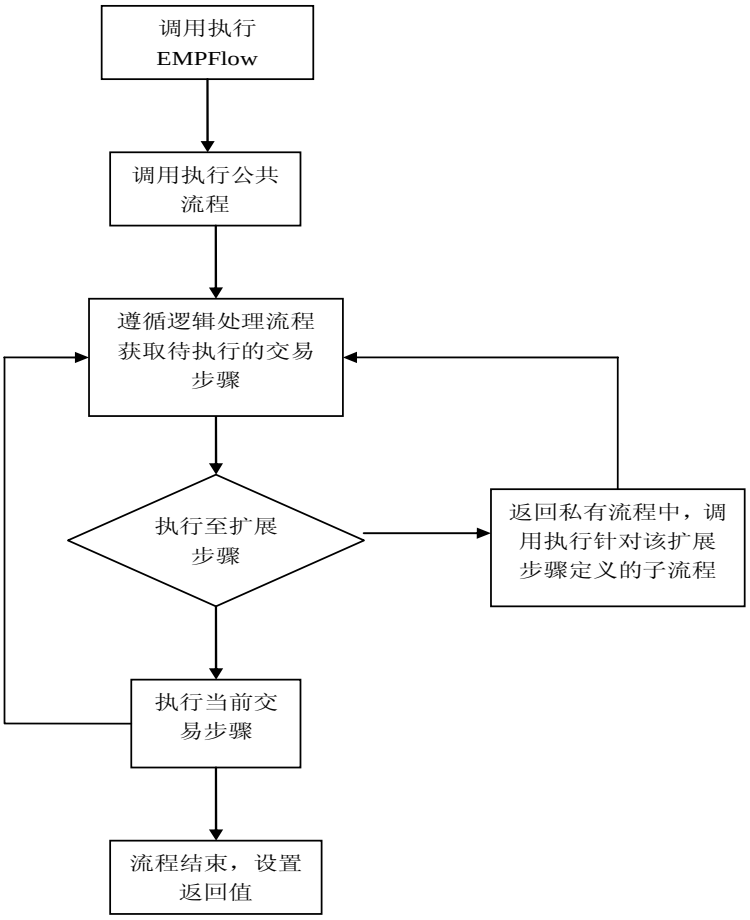
一个业务逻辑可以引用一个已经定义好了的公共主流程，这个公共子流程已经提供了一些流程扩展点，在这些扩展点上，定义具体业务逻辑时可以增加本业务逻辑所需要业务操作步骤，从而满足本业务逻辑处理的需要。

引用公共流程后，业务逻辑的处理流程

在定义普通的业务逻辑流程时，如果选择引用公共流程，EMP 系统中的业务逻辑处理容器会将两者进行组装，该流程的执行过程将遵循下图所示的方式进行。

整个流程在被调用执行时，将直接跳转至所引用的公共流程，并遵循公共流程中定义的状态迁移方向进行跳转执行，该过程与普通业务流程的执行过程基本相同。而两种方式的区别在于，当公共流程执行至扩展步骤定义组件 ExtendedAction 时，业务逻辑处理容器将会根据当前扩展步骤的扩展点标识，在私有业务流程中检索到对应的扩展步骤，并执行其中自

定义的子流程，执行结束后，又将返回值公共流程定义，继续调用下一交易步骤。



公共流程的定义

公共流程需要在业务逻辑分组中进行定义，与普通业务流程类似，同样是以状态迁移标识将当前流程中所需调用的所有交易步骤组件，联接成为完整的处理流程。

在公共流程中，可以定义多个空白的扩展步骤 ExtendedAction，用于标记公共流程中允许插入自定义子流程的位置。而子流程的具体实现，则需要定义在应用该公共流程的普通业务流程中。

扩展步骤的定义

扩展步骤是将公共流程与引用它的普通业务流程进行组装时所需的切入点，它需要在两种业务流程中进行分别定义：

- 在公共流程中，可以定义多个空白的扩展步骤 ExtendedAction，用于标记公共流程中允许插入自定义子流程的位置。

- 在引用某公共流程的普通业务流程中，需要为在该公共流程中已定义的扩展步骤定义其具体的子流程实现。该子流程所涉及到的交易步骤和跳转条件，将被包含在扩展步骤定义中，以供公共流程调用执行。

扩展步骤的实现类为 com.ecc.emp.flow.ExtendedAction，它的唯一属性 extendedId 用于标识公共流程中待实现的各个子流程扩展点。

3.2.4.5.3. 公共流程的配置定义

业务逻辑分组中的公共流程定义（operations.xml）

公共流程的配置信息定义在所属业务逻辑分组目录下的 operations.xml 文件中。该文件中允许定义多个公共流程，配置的定义规则与普通业务流程基本相同。当前业务逻辑分组中的私有业务逻辑流程可以通过公共流程的 ID 标识对其进行引用；同时公共流程中允许定义空白的扩展步骤 ExtendedAction，提供给私有业务流程实现。

下面是一个公共流程在 operations.xml 文件中的配置实例。

```
<operations.xml>
   <flow id="commonFlow">
      <action id="StartAction0">
         <transition dest="eFlowTestAction0"/>
      </action>
      <action id="eFlowTestAction0">
         <transition dest="ExtendedAction0"/>
      </action>
      <!-- 扩展步骤0定义开始 -->
      <action id="ExtendedAction0" extendId="extendedPoint0">
         <transition dest="eFlowTestAction1"/>
      </action>
      <!-- 扩展步骤0定义结束 -->
      <action id="eFlowTestAction1">
         <transition dest="eFlowTestAction2"/>
      </action>
      <action id="eFlowTestAction2">
         <transition dest="ExtendedAction1"/>
      </action>
      <!-- 扩展步骤1定义开始 -->
      <action id="ExtendedAction1" extendId=" extendedPoint1">
         <transition dest="EndAction0"/>
      </action>
      <!-- 扩展步骤1定义结束 -->
```



```
        <!-- 扩展步骤1定义结束 -->
        <action id="EndAction0"/>
    </flow>
</operations.xml>
```

在上面的实例中，为当前业务分组定义了公共流程 commonFlow，并添加了 2 个扩展步骤 **ExtendedAction0** 和 **ExtendedAction1**，扩展步骤中未定义子元素节点，将其提供给引用该流程的私用流程进行自定义。其中扩展步骤的 extendId 属性用于标识提供给私用流程自定义的子流程的插入位置。

buzinessLogic构件中的扩展步骤定义

在创建 buzinessLogic 构件的业务流程时，可以引用当前业务分组中已定义的公共流程，方法是在定义业务流程时设置 refFlowId 属性。已经引用了公共流程的私有流程，只需要对该公共流程中提供的扩展步骤所包含的子流程进行自定义，不同扩展步骤以 extendId 属性进行标识。需要注意的是，对于已经引用了公共流程的私有流程，在私有流程中定义的任何与扩展步骤无关的流程跳转将被视为无效，公共流程将只调用私有流程中实现的扩展步骤实现。

下面是一个引用了公共流程 commonFlow 的私有流程配置实例。

```
<EMPBusinessLogic id="eFlowBiz">
    <!-- 引用公共流程commonFlow -->
    <operation id="privateFlow" refFlowId="commonFlow">
        <flow>
            <!-- 扩展步骤0子流程的实现 -->
            <action id="ExtendedAction0" extendId=" extendedPoint0">
                <action id="eFlowTestAction3">
                    <transition dest="eFlowTestAction4"/>
                </action>
                <action id="eFlowTestAction4"/>
            </action>
            <!-- 扩展步骤1子流程的实现 -->
            <action id="ExtendedAction1" extendId=" extendedPoint1">
                <action id="eFlowTestAction5">
                    <transition dest="eFlowTestAction6"/>
                </action>
                <action id="eFlowTestAction6"/>
            </action>
        </flow>
```

```
    </operation>
</EMPBusinessLogic>
```

在上面的实例中，为 buzinessLogic 构件定义了私有的业务流程 **privateFlow**，同时通过设置 **refFlowId** 属性引用了上一节中定义的公共流程 **commonFlow**。私有流程中对公共流程提供的 2 个扩展步骤 **ExtendedAction0** 和 **ExtendedAction1** 进行了自定义，方法是将需要插入到公共流程的子流程定义，添加为对应扩展步骤的子元素。

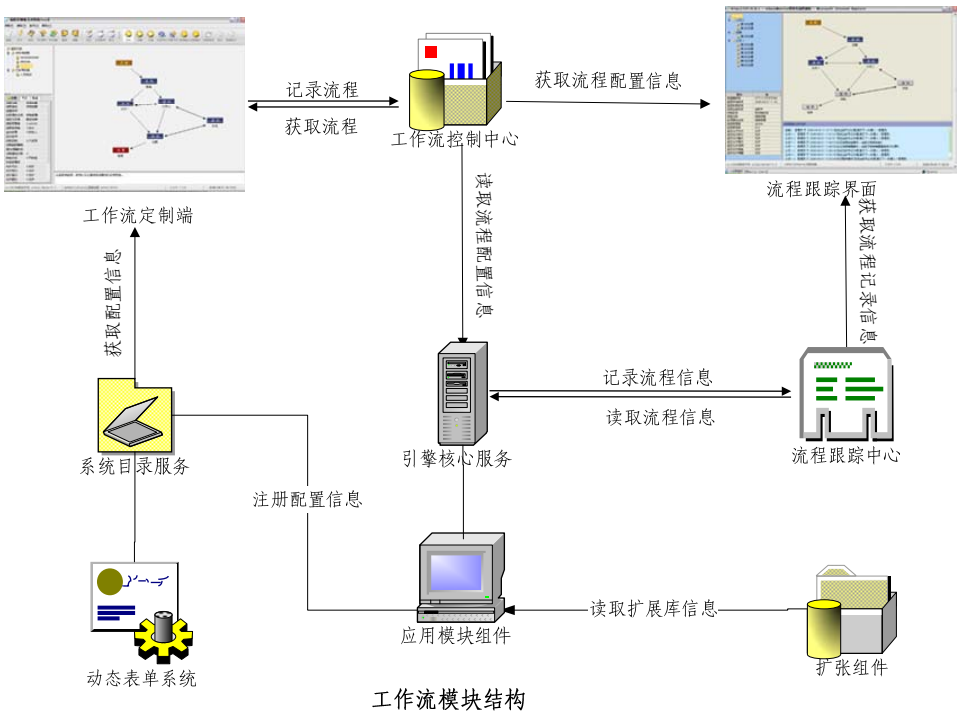
3.3. EMP 工作流引擎

EMP 工作流引擎提供一个完整的工作流应用系统解决方案，既可以单独运行也可以与 EMP 其他应用无缝融合产生更强大的组合应用。系统由若干个独立的功能模块组成，各模块功能如下所示：

序号	产品模块	功能描述
1	eChainStudio流程定制工作室	核心模块，负责以图形化的方式定制流程的业务规则
2	eChainWorkFlow易擎核心服务	核心模块，工作流引擎是驱动流程流动的主要部件，它负责解释工作流流程定义，创建并初始化流程实例，控制流程流动的路径，记录流程运行状态，挂起或唤醒流程，终止正在运行的流程，与其他引擎之间通讯等等工作
3	eChainMonitor图形跟踪监控	可选模块，流程跟踪监控为工作流提供监督和记录机制，可起到防篡改防抵赖的审计功能，能够准确的再现流程的办理情况。系统提供了图形化的流程跟踪模式，树型流程跟踪模式以及简单的流程跟踪模式
4	eChainStatistic统计分析工具	可选模块，利用eChainStatistic提供的统计分析接口，应用程序可以方便的取得各模块的流程办理情况，并生成相应的统计报表或者直接以图表方式只管反应流程任务的办理情况
5	eChainOU用户管理模块	可选模块，也可以读取客户已有的用户系统 完善的组织机构设定功能，能够确切地反映用户实际的组织机构。通过工作流平台的组织机构功能，能够实现按个人、部门、群组或角色多种方式来设定和显示组织成员
6	eChainTransaction个人事务管理	可选模块，也可以集成用户已有的应用系统（如信息门户），做为个人办公的统一入口，提供任务发起、工作

		任务查询、工作任务办理、委托授权等；
7	eChainDyForm动态表单系统	可选模块，支持集成客户已有的表单系统 eChainDyForm动态表单快速创建易用表单，更灵活、准确地创建信息；最主要的是数据与显示分离，数据和显示自由结合，实现灵活的数据输出；每个表单可以对应多个数据实例，能够通过一张表单提交多个数据实例；每个数据实例都独立于表单外观，能够被应用程序灵活的操作；内置强大的数据校验，不需要编程即可满足常规的业务需求；支持b/s结构的技术应用模式，无需安装客户端和任何插件；
8	eChainMessage即时消息模块	可选模块，支持消息、邮件、短信三种方式

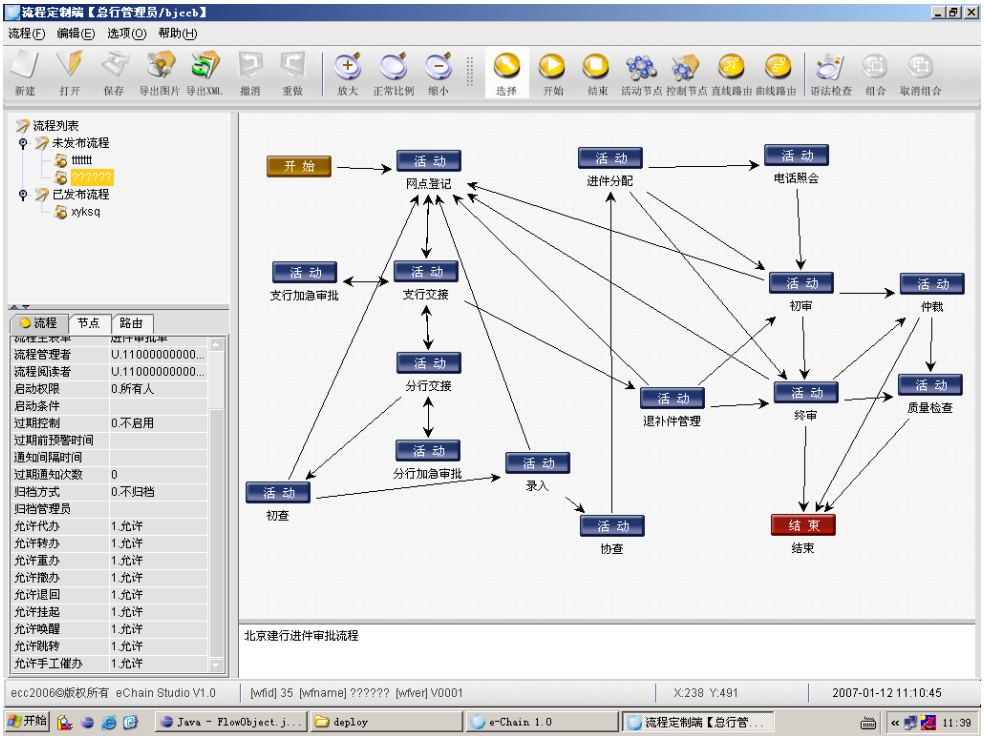
EMP 工作流平台参照 WFMC 提供的工作流模型，按照多层结构技术设计开发的一套基于数据库和 WEB 的工作流系统。其中核心的工作流引擎以组件形式封装，用户可以调用其中的接口,自行开发用户界面或内嵌到其他软件系统。在工作流引擎内部，采用了缓存和多线程等技术来提高系统性能。能够适应企事业单位与政府机关业务现状并满足信息化快速发展的业务流程需要。平台移植性好，扩展能力强，其核心工作流引擎采用状态机消息驱动机制，整个底层引擎由统一的消息服务器进行消息队列调度，保证平台运行的可靠性，有效均衡负载，提高系统的容错能力，健壮性好。



工作流模块结构

工作流控制中心是整个工作流平台的核心与基础，工作流控制中心包括工作流引擎、工作流核心服务、工作流对外 API 接口三个主要模块，工作流引擎是驱动流程流动的主要部件，它负责解释工作流流程定义，创建并初始化流程实例，控制流程流动的路径，记录流程运行状态，挂起或唤醒流程，终止正在运行的流程，与其他引擎之间通讯等等工作。工作流控制中心定义并解释工作流文档的业务流转规则，工作流控制中心记录了一个流程所有业务细节，包括流程名称、各节点名称、各节点办理人、办理属性、权限控制、流转控制等一系列标识信息。

工作流平台最基本也是最主要的功能是必须能够按照预定的业务流程(Business Process)驱动应用实例一步步往下正常流转，以及在此基础上更好的扩充流程的灵活性与提高办理的权限控制。



EMP 工作流引擎详细介绍请参考“eChain 易擎工作流应用平台技术手册”文档。

3.4. EMP 应用管理模型 – JMX MBean

3.4.1. 概要介绍

JMX 功能是 EMP 平台提供新一代 J2EE 体系下的监控管理框架的实现。

JMX：Java Management Extension，Java 管理扩展，是 Java 的一个管理标准。现在的版本为 1.2 版本，基于 jre1.5 以上版本中提供，在 1.2 版本中与以前版本的主要变化集中在对远程监控功能的支持上。在 1.2 版本中提供了基于 RMI 协议的远程访问和控制接口的实现，而在 1.5 以上版本中并不存在。

EMP 平台对监控管理支持上提供了监控服务器实现和监控客户端实现。

监控服务器实现的功能主要是将 EMP 应用开放为标准的 JMX 服务器，将 EMP 组件开放为标准的 JMX MBean 对象共外部 JMX 客户端应用访问。

监控客户端是一个基于 EMP 平台上实现的一个 web 应用，可用来提供与任意 JMX 服务器对象进行连接，并按照一定规范展现 JMX 服务器上可监控对象信息。并可根据监控需要在客户端进行数据信息采集和监控报警设置等处理。

监控客户端是一个独立的应用系统，不在 EMP 平台组件范围之内。如果想了解这部分内容，可参考相关 EMP JMXMonitor 系统的相关文档。

本部分内容主要目的是描述 EMP 平台如何实现 JMX 框架，并提供灵活的配置手段将 EMP 应用组件开放为可监控的 MBean 的原理和配置方法。

3.4.2. 工作原理

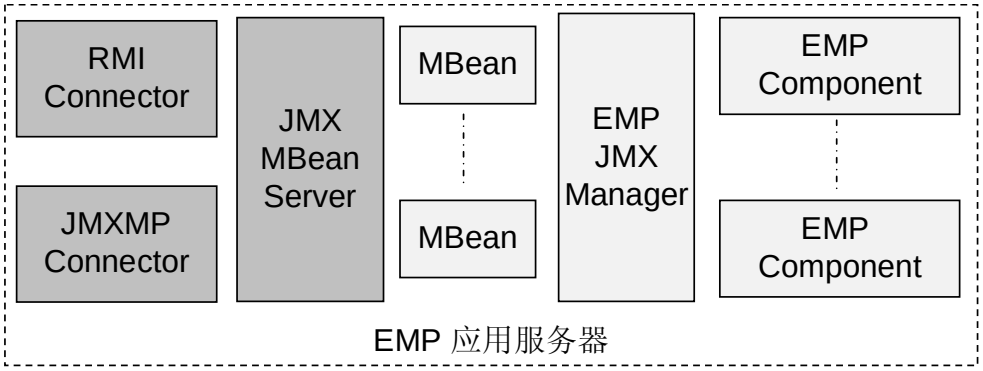
3.4.2.1. 设计框架

JMX 框架的设计初衷是监控应用不应该影响到被监控应用的组件对象。传统的监控设计总是需要从被监控组件中设置监控接口以便获得所需的监控信息。这些需要被监控组件去扩展的监控接口一旦加入到应用系统中，就将破坏被监控系统设计的完整性，提高应用系统与监控系统之间的耦合度，相应地就会为后期升级或系统维护带来问题。

JMX 采用将需要监控的组件对象进行外部封装，通过封装将被监控对象的实际方法或属性开放出来能够让外部系统访问。这种机制保证了被监控对象是无须知道自己被监控，也不必为监控做任何的组件升级。这也是 JMX 框架的价值所在。

EMP 平台在平台内部缺省内置了可配置的 JMX 服务器对象，可以快速将 EMP 平台组件封装为可用的 MBean 对象并放置在内部的 JMX 服务器上共外部监控系统访问。

EMP 为实现 JMX 框架进行了如下设计：



如上图所示，JMX MBean Server 是 JMX 框架所提供的 JMX 服务器，EMP 平台只需要将 EMP Component 通过 EMPJMXManager 对象进行封装生成对应的 MBean 对象，并将这些 MBean 对象在 JMX 服务器上注册即可实现 EMP 平台对 JMX 框架的支持。

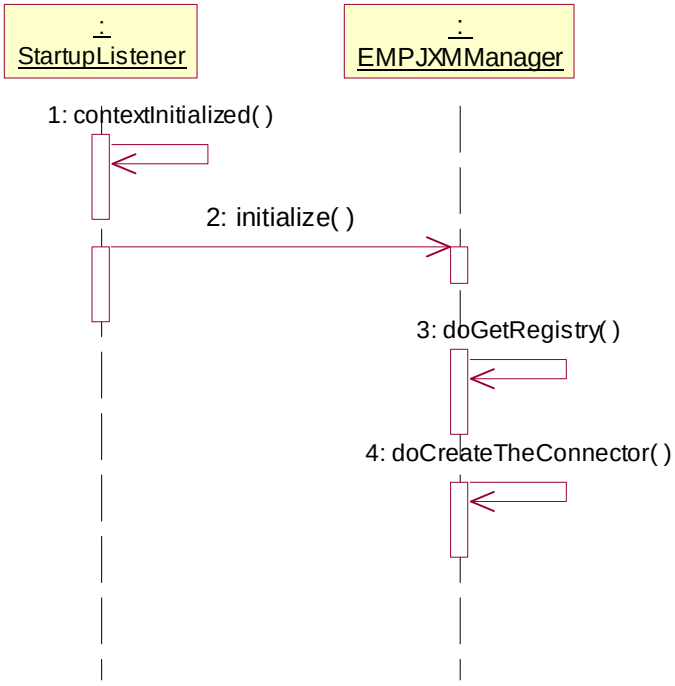
RMI Connector 与 JMXMP Connector 组件是 JMX 框架提供的外部应用系统远程访问接口，在 EMP 平台可通过配置来定义平台应用内置的 JMX 服务器对象和外部访问的连接器对象。

3.4.2.2. EMP平台内建JMX服务器

EMP 平台对 JMX 的支持的实现全部集中的 EMP 代码包：com.ecc.emp.jmx.support 包中。该包大部分类是由一系列描述 EMP 组件如何外部化 MBean 的配置对象类组成。其中与 JMX 框架进行融合的实现关键类只有一个就是 EMP JMX 管理器类：**EMPJMXManager。**

EMPJMXManager 类在一个 EMP 应用中静态存在，只能有一个 JMX 管理器对象。该管理器对象负责在 EMP 应用初始化时，按照外部配置信息创建 JMX 服务器和对外连接适配器对象，并根据配置的对外开放的 EMP 组件信息，将这些组件封装成 MBean 对象注册到 JMX 服务器上。

EMP 应用创建 JMX 服务器的过程如下所示：



1、通过 servlet 中的初始化事件监听对象调用 EMPJMXManager 的初始化方法

2、在 EMPJMXManager 静态初始化方法中，创建 JMXMBeanServer 对象

3、创建 JMXMBeanServer 对象中的注册器对象，用为为每一个创建出来的 MBean 进行注册

4、创建连接器对象，提供给外部系统访问

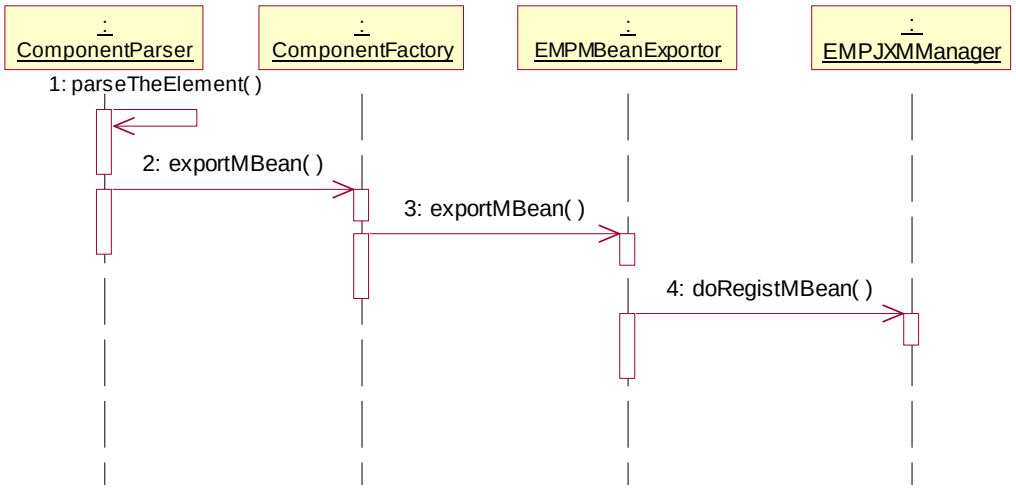
EMP 平台创建 JMX 服务器是可以通过外部化文件来配置所生成的服务器对象的端口，及开放的访问协议及访问端口信息。具体配置信息是在配置文件：JMXContext.xml 文件中。

3.4.2.3. 将EMP组件封装为MBean

将 EMP 组件封装为 MBean 对象，最关键的步骤是需要将 EMP 组件对象的对象指针传递给 MBean 对象。即通过 MBean 对象能够找到 EMP 组件对象实体，才能正常访问 EMP 组件的方法和属性。

在 EMP 平台中，当 EMP 组件工厂在创建每一个 EMP 组件时，都会询问该组件是否需要被开放为一个 MBean 对象。当该组件需要被开放为一个 MBean 对象时，EMP 通过外部化配置信息能够找到该组件将被开放的属性和方法定义，最终封装为满足要求的 MBean 组件并在 JMX 服务器中进行注册。

EMP 平台创建并注册 MBean 组件的过程如下所示：



创建 MBean 的步骤如下描述：

1、ComponentParser 通过外部化配置文件创建对应的 EMP Component 对象

2、EMP Component 对象创建后，将调用 ComponentFactory 组件工厂对象的 exportBean 方法，该方法将判断该 EMP 组件是否需要被开放为 MBean 对象

3、组件工厂调用 EMPMBeanExportor 对象的 exportMBean()方法。EMPMBeanExportor 对象是用来存放所有的 EMP 组件需要被开放为 MBean 时的所需要开放的属性信息和方法信息。它是一个集合对象。exportMBean 方法将判断该创建的 EMP 组件是否在 EMPMBeanExportor 对象中存在，如何存在就意味着该 EMP 组件将被开放为一个 MBean

4、如果一个 EMP 组件需要被开放为一个 MBean，最终将调用 EMPJMXManager 对象的 doRegisterMBean 方法，将 EMP 组件封装为一个 MBean 对象，并加载该 EMP 组件开放的属性和方法，在 JMX 服务器中注册该 EMP 组件所封装的 MBean 对象。

在 EMP 平台上，所有需要被开放为 MBean 对象的 EMP 组件的信息均采用外部化文件的方式进行配置。这些外部化配置文件在系统初始化时将被组装为：EMPMBeanInfo 对象，存放在 EMPMBeanExportor 对象容器中。EMPMBeanInfo 对象就是描述 EMP 组件需要被开放的属性和方法的信息对象。它包含一系列的描述对象类，如下表所示，是这些描述对象类的具体含义，他们所具有的意义就是将外部化文件中的配置信息组装为一个可描述的 EMP 组件被封装为一个 MBean 所必须的信息对象。

类名	类描述
EMPMBeanInfo	EMP 组件被开放为一个 MBean 的描述类，该类下包含该 EMP 组件开放的属性对象和方法对象信息的描述
AttrInfo	EMP 组件被开放的属性对象描述

OperationInfo	EMP 组件被开放的方法对象描述
ParameterInfo	EMP 组件被开放的方法的参数对象描述

3.4.3. 使用说明

3.4.3.1. JMX服务器信息配置说明

JMX 服务器信息主要配置内容为 JMX 服务器开放的访问端口和访问名称字符串和访问连接器的访问字符串定义。

该信息由一份独立的配置文件提供配置。该配置文件位于应用项目下的 WEB-INF 目录下的 EMPJMXContext.xml 文件。文件基本内容如下所示：

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

<EMPJMXManager>
    <classMap>
        <map id="EMPJMXManager"
class="com.ecc.emp.jmx.support.EMPJMXManager"/>
        <map id="registry"
class="com.ecc.emp.jmx.support.RegistryInfo"/>
        <map id="connector"
class="com.ecc.emp.jmx.support.ConnectorInfo"/>
    </classMap>

    <registry host="localhost" port="10330"/>
    <connector id="RMIConnector"
    serviceURL="service:jmx:rmi:///jndi/rmi://localhost:10330/jmxServ
er"/>

</EMPJMXManager>
```

看上表中信息，classMap 标签内定义的是本文件中的 tag 与类之间的映射关系。

Registry 标签中的定义内容为 JMX 服务器的 IP 地址与开放端口，该端口应与下面的访问端口保持一致。

Connector 标签中定义的是一个 RMI 协议支持的访问字符串。

‘**service:jmx:rmi:///jndi/rmi://**’为该协议的访问字符串编写格式规范要求，

‘**localhost:10330**’为开放饿服务器地址与端口，‘**jmxServer**’是为该连接适配器对象定义的访问名称。在客户端访问时必须配置相同的访问字符串名称。

3.4.3.2. MBean信息配置说明

MBean 信息配置是一份独立的配置文件，它是应用项目中的业务逻辑分组中的 mbeans.xml 文件。该文件的关键内容信息如下：

```
<mbeanExportor id="mbeanExportor">
       <!--  直接实例化JavaClass 并导出为MBean -->
       <EMPMBean id="systemInfoBean" name="systemInfo"
objectClass="com.ecc.emp.jmx.support.SystemInfoMBean"
description="System info bean" type="native">
           <attr id="freeMemory" name="freeMemory"
displayName="freeMemory" description="current free memory" access="R"/>
           <attr id="totalMemory" name="totalMemory"
displayName="totalMemory" description="current total memory"
access="R"/>
           <attr id="maxMemory" name="maxMemory"
displayName="maxMemory" description="current max memory" access="R"/>
           <op id="doGC" name="doGC" displayName="doGarbageCollection"
description="doGarbageColl"/>
       </EMPMBean>
      <!--  将EMP组件并导出为MBean -->
       <EMPMBean id="dataSource" name="dataSource"
objectId="dataSource" description="The dataJDBCSource" type="JDBC">
           <attr id="driverName" name="driverName"
displayName="driverName" description="the task tree define xml file"
access="R"/>
           <attr id="dbURL" name="dbURL" displayName="dbURL"
description="reload the task tree define xml file" access="R"/>
           <op id="getConnectionCount" name="getConnectionCount"
displayName="获取当前连接数" description="获取当前连接数"
displayFile="getConnectionCount.vm"/>
       </EMPMBean>
   </mbeanExportor>
```

如上表内容，mbeanExportor 标签下的内容就是需要在 JMX 服务器上进行注册的组件信息。分为两类组件。一类是根据定义信息直接创建对象实例，然后产生 MBean 并进行注册。另一类就是 EMP 组件对象，需要在此进行该组件的 mbean 开放属性和方法的定义，应用将根据该配置信息将指定的 EMP 组件开放为 MBean。

systemInfoBean 是将一个 class 直接生成并开放为一个 MBean 对象。它需要定义对应的生成类属性：objectClass。在该标签下是该类所需要开放的属性定义和方法。

dataSource 对象是对一个 EMP 组件开放为一个 MBean 组件的定义。EMP 组件开放为 MBean 的信息配置中的关键属性为：id 和 objectid 两个属性。这两个属性一定要与 EMP 组件的定义 ID 相同。在组件工厂创建了该 EMP 组件时，将根据组件的 ID 在 mbeanExportor 容器中去找相同 ID 的 mbean 对象定义。只有 EMP 组件 ID 与 EMPMBean 对象 ID 和 objectid 定义相同时，系统才认为该 EMP 组件将开放为一个 MBean 对象。

开放的属性标签定义与开放的方法标签定义通过配置文件内容能够很好地理解，在这里不再具体描述了。

4. EMP核心组件

4.1. 表达式定义

表达式定义是 EMP 提供的基于脚本的条件判断和计算组件。通过表达式完成数值及逻辑运算。表达式支持基本的四则运算、逻辑运算、字符串处理。表达式支持：变量、函数处理，函数包括平台提供的函数和用户自定义的函数。

表达式在 EMP 应用中随处可见，它是 EMP 配置灵活性的重要实现手段，在跳转条件判断、数据有效性校验等应用上，通过使用表达式提供强大的运算能力让条件判断更为灵活和。

4.1.1. 表达式语法

- **常量定义**

字符串：用单引号“”括起来，如：'sdfadfadfasfas'

数值：直接书写，如:100,10.05 等

Bloolean 值：直接书写，且必需是小写，如：true，false

日期常量值： '$2007/10/11'

- **变量定义**

变量通过符号“$”进行标识，如： $retValue

表达式处理中，是通过登记的变量提供者插件(ValuableController)的方式获变量值

在 EMP 中使用的表达式，变量值通常取自 Context 中的数据

- **函数定义**

函数定义通过符号“@” 如：@subString（’abcdefgrioer’,2,4)

函数中的参数也是一个表达式，参数间用‘，’分割

- **转义符**

通过在需要转义的符号前面增加’\‘比如需要在字符串： 'sdfadfadfasf\'as'

- **运算符**

四则运算：加+ 减- 乘* 除/

左括号(

右括号)

逻辑运算： 等于= 非! 小于< 大于> 不等于!= 小于等于<= 大于等于>= 与& (或 and) 或 | (或 or) 。

4.1.2. EMP已提供的表达式处理函数

函数名	说明	参数
@Length	取字符串长度	src
@StringToBigInteger	字符串转大整形	src
@StringToDate	字符串转为日期	src, fmt
@StringToDecimal	字符串转数值型	src
@StringToFloat	字符串转浮点	src
@StringToInt	字符串转整数	src
@subString	取子字符串	Src，offset，len
@toString	将表达式的值转为字符串	src,fmt (fmt 可有可无)
@toChinessCurrencyStr	转换为中文金额	src
@indexOfSubString	子字符串在主串的位置	src, subStr
@currentDate	取当前日期	无

4.1.3. EMP表达式函数扩展

EMP 表达式扩展提供了对表达式函数的扩展。扩展了自己的函数类后，将该类注入到系统提供的 EMP 函数配置文件中就可以在应用中使用该函数了。

EMP 表达式函数扩展分以下两步：

- **编写自己的函数**

从类：com.ecc.util.formula.function.Function 继承，实现方法：public abstract FormulaValue getValue(List argList)throws FormulaException;即可，其中 argList 中包含类型为 FormulaValue 的参数列表

- **将用户自己编写的函数注入到 EMP**

在函数文件中定义自定义函数：一般来说，该文件叫 function.xml 文件，位于 WEB-INF 目录下的 common 目录下。我们在应用的 web.xml 文件中的 context 参数配置中对该文件的位置和名称进行定义，系统初始化时将通过该参数将函数配置文件进行初始化。

```
<context-param>
        <param-name>functionFileName</param-name>
        <param-value>WEB-INF/commons/function.xml</param-value>
</context-param>
```

在 function.xml 文件中，要注入一个扩展的函数类，如下表所示：

```
<Fuction id="SizeOfICollection" name="SizeOfICollection"
class="com.ecc.emp.processor.function.SizeOfICollection"/>
```

其中 name 就是指函数名，使用时 @SizeofICollection($abc.d)。

4.1.4. 表达式使用举例

```
<action id="TCPIPAccessAction0"
implClass="com.ecc.emp.tcpip.TCPIPAccessAction" timeOut="5000"
sendFormatName="transfer" serviceName="tcpipService"
receiveFormatName="transfer">
                    <transition dest="EndAction0"
condition="($retValue='0') and ($hostResult='S')"/>
                    <transition dest="SetFieldErrorMsgAction0"
condition="($retValue='0') and ($hostResult='L')"/>
                    <transition dest="SetFieldErrorMsgAction1"
condition="($retValue='0') and ($hostResult='F')"/>
                    <transition dest="EndAction1"/>
```

```
</action>
```

如上表所示，是一个 Action 执行后的跳转设置，`($retValue='0') and ($hostResult='L')`就是一个表达式的例子，它是两个判断条件取合的条件设置。注意在 EMP 的判断条件中，判断相等只是用一个‘=’，而不是‘==’。

4.2. 文件日志

EMP 平台中的 EMPLog 文件日志组件是采用配置文件的方式，在平台初始化时由配置文件决定平台所使用的具体日志管理系统，目前平台自带的日志管理系统是 Log4j 系统。在项目开发中，只需要引进 log4j 相关的 jar 包（如 log4j-1.2.7.jar），并在配置文件中将文件日志设置为使用 log4j，就可以使用 EMP 平台的 EMPLog 组件来记录文件日志。同时，不需要编程，只需要通过配置少量的配置文件就可以生成特定的文件日志。此外可以通过扩展接口，实现不同类型的文件日志管理（非 log4j），可以在不改变平台其它代码的情况下，透明的转换整个平台的日志管理系统。

4.2.1. 文件日志概述

无论是在项目开发还是在程序的运行过程中，我们往往需要了解程序运行的相关信息，包括所抛出的异常信息，关键步骤的信息等等。例如在网银系统中，客户在进行转帐交易时，无论是成功或者是失败都应该将转帐的相关信息进行保存，如果失败还应该保存失败的原因。这样，今后客户对该笔交易提出质疑或者是需要对该笔交易进行检查时都可以通过相关的保存来获得需要的信息。同时由于在程序运行中会产生大量这样的数据，一般都是通过文件的形式进行保存，并通过一定的时间或文件大小对保存的数据进行清理。这些日志信息的管理，在 EMP 平台中是通过 EMPLog 组件进行管理。另外，为了在系统运行过程中可以随时监控和设置日志管理基本信息，平台还将日志管理纳入了监控平台中。

在 EMP 平台中，平台在初始化的时候通过初始化设置获得日志处理的实现类，并获得它的一个实例。在平台其它地方使用到文件日志时，就会调用该实例中的相关方法进行处理。这样就可以通过扩展实现类的接口实现不同的日志处理。

4.2.2. Log4j的基础工作原理

4.2.2.1. 日志信息的级别

在项目开发中，由于源代码的不可见，对于文件日志的跟踪变成了开发过程中排错的最有效途径，但是，生产阶段的很多日志信息在项目上线之后又不想输出到文件日志中，而日志信息的级别划分就是为了解决这种情况。在 EMP 平台中，日志信息被分为六个级别：

- EMP.DEBUG
- EMP.TRACE
- EMP.INFO
- EMP.WARN
- EMP.ERROR
- EMP.FATAL

它们之间的级别关系是：EMP.DEBUG < EMP.TRACE < EMP.INFO < EMP.WARN < EMP.ERROR < EMP.FATAL，即 DEBUG 时输出的日志信息最多，FATAL 输出的日志信息最少。

同时，日志信息的输出遵循的也是 log4j 的规则：假设配置输出的级别为”a”（六种级别中的一种），如果外部配置的日志信息的级别 b 比 a 低（或相同），则可以输出，否则屏蔽掉。通常调试性信息应以较低日志级别输出，而各种错误或关键信息则会以较高的级别输出。这样，在开发阶段时，使用低级别配置，可以展现更多的输出信息，达到调试的作用。生产运行阶段，设定日志级别较高，则只输出错误或关键信息。

4.2.2.2. 日志信息的类别

日志信息要根据类别保存在不同的文件中，以便进行查看，例如：与数据库相关的日志放在一个文件中，而与通信相关的日志信息放在另一个文件中。此外，再某些时刻，我们只需要输出某些类别的日志，而不是全部类别，这就需要将日志信息进行分类。

目前在 EMP 平台中共有几个基本的类别，例如：EMPConstance.EMP_FLOW，EMPConstance.EMP_JDBC，EMPConstance.EMP_TCPIP 等等（这些类别可以在

EMPConstance 的静态变量中得到）。在定义日志的输出方式时就可以针对专门的类别定义是否输出及输出的文件等等。

4.2.2.3. LOG4J的输出方式

LOG4J 的日志信息可以定义具体的输出方式，例如：输出到控制台、每日产生一个日志文件等。具体的方式有：

- org.apache.log4j.ConsoleAppender（控制台）；
- org.apache.log4j.FileAppender（文件）；
- org.apache.log4j.DailyRollingFileAppender（每天产生一个日志文件）；
- org.apache.log4j.RollingFileAppender（文件大小到达指定尺寸的时候产生一个新的文件）；
- org.apache.log4j.WriterAppender（将日志信息以流格式发送到任意指定的地方）

4.2.2.4. LOG4J的输出样式

另外还可以定义日志信息的输出样式，例如：以表格形式、输出信息包含日期等等，具体的样式有：

- org.apache.log4j.HTMLLayout（以 HTML 表格形式布局）；
- org.apache.log4j.PatternLayout（可以灵活地指定布局模式）；
- org.apache.log4j.SimpleLayout（包含日志信息的级别和信息字符串）；
- org.apache.log4j.TTCCLayout（包含日志产生的时间、线程、类别等等信息）

4.2.3. EMPLog组件的使用

4.2.3.1. 配置文件使用

对于 EMP 平台中已经有的日志信息，如果使用平台自带的 log4j 日志处理，则只需要配置相关的配置文件就可以选择需要输出的类别、方式、样式等。配置文件的相关位置可以在 web.xml 中找到：

```
    <context-param>
```

```
      <param-name>logImplClass</param-name>

   <param-value>com.ecc.emp.log.EMPLog4jLog</param-value>
   </context-param>
   <context-param>
      <param-name>logSettingFile</param-name>

   <param-value>WEB-INF/commons/logging.xml</param-value>
   </context-param>
   <context-param>
      <param-name>logFileName</param-name>
      <param-value>WEB-INF/empLog.log</param-value>
   </context-param>
```

首先，设置 logImplClass 属性为所需要的日志处理的实现类，在这里，我们使用的是平台自带的 log4j 的日志处理，因此该属性的值就为 EMPLog4jLog 类名（包括包名）。

另外，对于日志处理类的相关的配置文件是由 logSettingFile 属性设置，在这里是 WEB-INF/commons/logging.xml。在这里，我们以 log4j 日志处理为例，在 logging.xml 中对日志信息的相关属性进行配置：

```
    <appender name="LOGFILE"
class="org.apache.log4j.FileAppender">
      <param name="File"
value="${ROOTPATH}WEB-INF/logs/emp.log" />
      <param name="Append" value="true" />
      <layout class="org.apache.log4j.PatternLayout">
         <param name="ConversionPattern"
            value="%d{ISO8601} %-5p %10c - %m%n" />
      </layout>
   </appender>
   <category name="FLOW">
      <priority value="debug" />
      <appender-ref ref="LOGFILE" />
   </category>
```

首先定义一个名称为 LOGFILE 的日志输出的相关配置信息，其中 appender 标签的 class 属性定义了日志输出的方式是：org.apache.log4j.FileAppender（文件）。该输出方式的属性 File 设置了日志输出的文件名称：${ROOTPATH}WEB-INF/logs/emp.log，在这里设定的是项目名称下的 WEB-INF/logs 下的 emp.log 文件中，而属性 layout 设定的则是日志信息的样式：org.apache.log4j.PatternLayout（可以灵活地指定布局模式）。该样式可以灵活的布局，其中的属性 ConversionPattern 就定义了输出的样式。

在定义完日志信息输出的相关信息后，接下来就可以定义不同类别的日志信息是否输出以及是以哪种方式输出。对于需要输出的日志信息类别，定义一个 category 标签，该标签的名称就是需要输出的日志信息的类别名称，在这里，我们需要输出类别名称为 FLOW 的日志信息。输出的级别由属性 priority 设定，在这里输出的级别为 debug 级别，这说明类别为 FLOW 的日志信息，只要级别为 debug 及高于 debug 的都可以输出，而另外一个属性 appender-ref 则是将该类别日志信息的输出的各种配置关联到刚才定义的 LOGFILE 上。

这样，对于系统运行过程中产生的 FLOW 类别的日志信息，只要是 debug 及以上级别的信息都可以以 LOGFILE 所配置的相关属性输出。而对于其它的输出情况只需要再定义一个日志输出的配置信息，然后再关联需要以该配置输出的日志信息类别就可以了。

4.2.3.2. 程序代码中设置日志

在用户开发的代码中，如果需要输出日志信息，则只需要在相关的位置写下面一段代码：

```
EMPLog.log("IRP", EMPLog.INFO, 0, "Test！ ");
```

其中，第一个属性是该日志信息的类别（IRP）；第二个属性定义的是该日志信息的级别（info 级别）；第三个属性目前保留，一般情况下都设置为 0；第四个属性设定的就是该日志信息输出的字符串。

在程序里添加了这一段代码之后，只需要在 logging.xml 文件中增加一个名称为 IRP 的日志信息类别并关联到某一个输出的配置上（如之前的 LOGFILE）。

在项目运行之后，就会在 emp.log 文件中打出类似下面的日志信息：

```
2007-04-25 16:06:56,418 INFO        IRP - Test！
2007-04-25 16:07:11,650 INFO        IRP - Test！
2007-04-25 16:07:28,975 INFO        IRP - Test！
```

4.2.4. EMPLog组件的扩展

除了使用目前平台已有的log4j日志处理以外，用户可以通过扩展com.ecc.emp.log.Log接口实现自己的日志处理。同时在 web.xml 文件中设置所使用的日志处理类名为扩展后的处理类名：

```
public class EMPLog4jLog implements Log{
    ......
    public void log(String component, int type, int level, String message, Throwable
```



```
exception )
        {
            ......
        }
    ......
}
```

在 web.xml 文件中：

```
    <context-param>
       <param-name>logImplClass</param-name>
    <param-value>com.ecc.emp.log.EMPLog4jLog</param-value>
    </context-param>
```

这样，就可以在不改变平台其它代码的情况下，转换整个平台的日志管理系统。而在平台需要记录日志的地方，依然使用 EMPLog.log 方法。

4.2.5. EMPLog拦截器

4.2.5.1. 拦截器实现机制

EMPLog 拦截器是针对多线程处理模式下的日志输出记录顺序交叉混合，无法有效提取同一次访问日志内容的情况而设计的。EMPLog 拦截器采用 EMP 平台提供的访问控制插件模式实现。EMP 允许用户注入自己的 Log 拦截器，在调用 Log 输出 log 信息时，首先调用注册的拦截器的相关方法，实现对日志输出的特殊处理。

EMP 提供了日志拦截器，实现同一用户的一个请求中的日志输出给定唯一标识，应用服务器中的 Web 容器是多线程处理的，日志输出也会交错，这样很难甄别同一一个用户的同一请求的日志输出。EMP 提供的 Log 拦截器基于 EMP 的访问控制器的机制，提供用户将 Log 输出的 component 名称映射为：渠道+请求 ID（或映射）的机制，实现 Log 的按照请求合理输出。

EMP 提供了 Log 拦截器接口：LogInterceptor。该接口提供了一个方法：

```
public boolean log(String component, int type, int level, String msg, Throwable te)
```

EMP 中调用 EMPLog 时，如果在应用中配置了 Log 拦截器实例，则首先调用拦截器的 log 方法，如果该方法返回 true，则该输出将被拦截不在标准输出中输出。被拦截的 log 在在拦截器中的 log 方法中进行处理。

4.2.5.2. 拦截器配置说明

在应用的 ServletContext.xml 文件中增加：

```
<accessManager id="accessManager">
  <logInterceptor componentIdPrefix="HTTP" appendUniqId="true" uniqIdLen="6" class="com.ecc.emp.log.EMPLogInterceptor">
    <logInstance class="com.ecc.emp.log.EMPLog4jLog"/>
  </logInterceptor>
 </accessManager>
```

这样的话，如果最终访问的是“testCMCC”，应用这端的日志会是：

```
2007-11-27 11:36:28,405 INFO  HTTP-testCMCC - 000001 Begin to instance EMPFlow [test]...
2007-11-27 11:36:28,405 INFO  HTTP-testCMCC - 000001 Instance EMPFlow [test] from file test.xml
2007-11-27 11:36:28,495 INFO  HTTP-testCMCC - 000001 Instance EMPFlow [test] OK.
2007-11-27 11:36:28,495 INFO  HTTP-testCMCC - 000001 execute the Flow: test.hello...
2007-11-27 11:36:28,556 INFO  HTTP-testCMCC - 000001 execute the Flow: test retValue:succ
```

如果第二次请求交易的话，日志会是

```
2007-11-27 11:40:14,931 INFO  HTTP-testCMCC - 000002 execute the Flow: test.hello...
2007-11-27 11:40:14,931 INFO  HTTP-testCMCC - 000002 execute the Flow: test retValue:succ
```

<logInterceptor componentIdPrefix="HTTP" appendUniqId="true" uniqIdLen="6" class="com.ecc.emp.log.EMPLogInterceptor">

其中，componentIdPrefix，标识的前缀，appendUniqId，是否要打印出唯一标识序号，uniqIdLen，唯一标识序号的长度。在上面的日志中可以看到，可以通过交易的代码以及唯一标识号来区别一个交易的所有日志信息。

还可以对交易码进行映射，在 logInterceptor 中增加映射配置：

```
<accessManager id="accessManager">
  <logInterceptor componentIdPrefix="HTTP" appendUniqId="true" uniqIdLen="6" class="com.ecc.emp.log.EMPLogInterceptor">
    <logInstance class="com.ecc.emp.log.EMPLog4jLog"/>
    <ComponentIdMap>
     <MapEntry key="testCMCC" value="移动缴费"/>
    </ComponentIdMap>
  </logInterceptor>
 </accessManager>
```

可以得到如下的日志：

```
2007-11-27 11:59:23,583 INFO  HTTP-移动缴费 - 000001 Begin to instance EMPFlow [test]...
2007-11-27 11:59:23,583 INFO  HTTP-移动缴费 - 000001 Instance EMPFlow [test] from file test.xml
2007-11-27 11:59:23,673 INFO  HTTP-移动缴费 - 000001 Instance EMPFlow [test] OK.
```

```
2007-11-27 11:59:23,673 INFO     HTTP-移动缴费  - 000001 execute the Flow: test.hello...
2007-11-27 11:59:23,753 INFO     HTTP-移动缴费  - 000001 execute the Flow: test retValue:succ
```

AccessManager 如果配置在渠道的 servletContext.xml 文件中：

如果是 HttpAccess 渠道：上面例子中的 testCMCC 指的是 httpRequestService 的名称。

如果是 TcpipAccess 渠道：上面例子中的 testCMCC 指的是 tcpipRequestService 的名称。

如果是 WebServiceAccess 渠道：上面例子中的 testCMCC 指的是 WebService 的名称。

如果是 html MVC：上面例子中的 testCMCC 指的是对应的 action 的名称。

AccessManager 如果配置在业务逻辑处理容器的 context 中，则上面例子中的 testCMCC 指的是 EMPFlow 的名称。

4.3. 访问控制

访问控制是一个系统应用安全中很关键的一部分，用于控制什么样的访问者在什么规则下能够访问系统某一部分应用功能。由于不同的应用系统所面临的用户和访问控制的设计规则都是千差万别的，客观上造成了在一个应用系统的设计中，访问控制部分往往是系统设计上的一个难点。

随着 J2EE 应用的发展，越来越多的功能被采用组件封装和插件模式的方式进行开发和应用，这种模式带来的好处是最大化的实现了组件之间的解藕，通过外部化配置的方式和标准接口设计来完成组件之间的调用和互动。

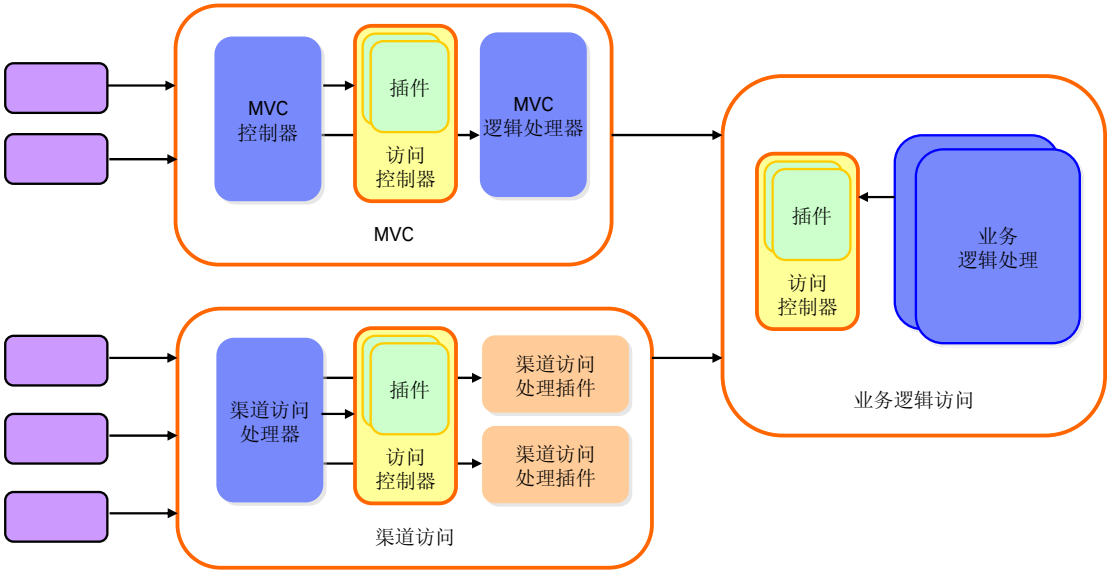
访问控制的设计也逐渐开始从具体业务设计中抽取出来，作为一个独立的访问控制插件来使用。对于一个应用系统的具体业务功能我们可以在完全不考虑访问控制安全的情况下进行开发和业务测试，在具体应用中，通过外部化对象的访问控制规则的配置来限制访问对象对应用对象之间的访问权限关系。

EMP 平台所提供的访问控制机制就是基于标准接口实现的一种可配置的插件方式的实现。

4.3.1. EMP访问控制的基本原理

4.3.1.1. EMP访问控制器的基本原理

EMP 平台试图提供一种基于插件方式的访问控制管理机制。在 EMP 平台中访问控制器插件与应用平台的关系如下图所示：



往往一个应用系统对应用功能的访问控制的灵活度是有很高的期望的，大部分应用服务器软件提供了基于角色的外部可配置化的访问控制器文件，每个用户都会拥有自己的角色，系统通过角色定义所关联的功能应用来确定一个用户是否可以访问该功能。这种方式就是一种基于插件的访问控制解决方案。

EMP 平台采用了插件的访问控制设计方式，但 EMP 在访问控制机制上想为用户提供的控制能力主要分为三种功能。

一、平台内部提供访问控制的行为模式，而各个行为具体实现由插件来完成。EMP 运行期间将根据插件的配置参数，动态调用外部化的访问控制插件，在 EMP 内部提供的访问控制行为逻辑中执行具体的插件实现的访问控制规则。

映射到 Java 设计思想上，这是一个基于 java 接口实现的典型应用。

二、平台提供的访问控制机制可以在平台应用系统的各个接口组件部分进行配置和部署。EMP 平台在 J2EE 分层体系的各个层面上都提供了接入及访问的控制入口，在业务逻辑处理层上、在 web 表现应用层上、在渠道访问应用层上都提供了接口一致的访问控制插件处理机制。

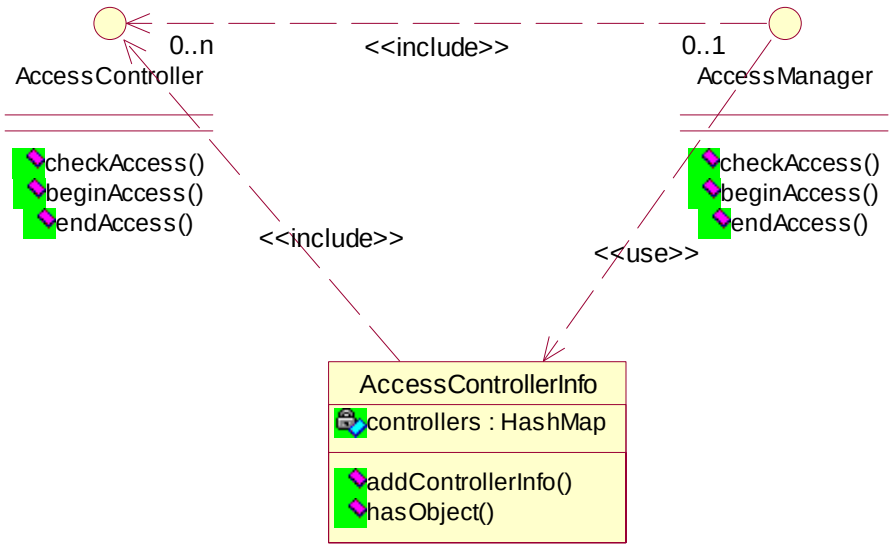
这种设计可以为用户带来应用系统访问控制处理的灵活性，可以在不同层面上进行控制和处理，也可以集中在某一个层面来进行控制和处理。由于采用了接口设计的方式，在各个层面上的访问控制处理器的实现是可以完全不同的，可以让用户根据不同层面的访问控制需求去扩展自己的访问控制器。

三、访问控制器在每个层面上并非只能实现一个。用户可根据自己的需要增加自己的访问控制器，应用系统会将所有的访问控制器实现进行统一配置和管理，并在进行访问控制检查时对所有的控制器进行访问处理。这也意味着访问控制插件处理机制并不意味着只是去做与访问控制相关的工作，也完全可以根据应用流程的需要与访问控制器一起同步插入我们所需要的功能处理。

4.3.1.2. EMP访问控制器的设计实现

EMP 平台访问控制插件通过接口方式进行设计，并内嵌在应用平台中自动产生作用。当权限检查不通过时，应用平台可抛出异常，拒绝访问。抛出的具体异常由具体的访问控制器实现类来提供。

下图为 EMP 平台访问控制插件的接口设计类图：



EMP 平台访问控制器接口设计了两个接口类，一个为 AccessManager，一个为 AccessController。这两个接口类的接口方法都是一样的，这样设计的目的是为了能够满足在一个访问控制插件中管理多个访问控制器对象。

接口方法定义描述如下：

Object checkAccess(Context context，Object rqeustObj,String actionId) throws EMPException

该方法为访问控制器的检查方法，传入的三个参数：context 代表着访问者的数据信息对象，requestObj 代表着 httprequest 对象，控制器可以从该对象中获取访问者的身份信息及其他关键数据信息， actionId 代表着一个被检查的功能点 ID，根据不同层面的功能对象的不同，该 ID 的组合形式也可能发生改变。checkAccess 的实现是不同控制器根据自身访问检查控制的需要实现的逻辑处理。如果访问控制检查成功，则返回一个访问控制对象，可在后续处理中对该访问控制对象进行处理。如果访问控制检查失败，则抛出非法访问异常，该异常的具体实现由访问控制器接口实现时具体进行定义。

beginAccess(Object accessInfo)

该方法是在访问控制器 checkAccess 处理成功后，对返回的控制对象进行处理的方法。在具体实现中也是根据具体需求进行扩展。当我们只需要检查访问权限时，可以设置该方法为空。

endAccess(Object accessInfo,long beginTimeStamp)

该方法是在访问结束时调用的方法，同样也是对访问控制对象进行处理，比如统计或记录痕迹等等操作，均可以在该步骤进行。该方法与 beginAccess 方法相比，多传入了一个时间戳的参数，主要是提供给用户当需要进行访问时间处理的情况下能够获得访问启动时间戳。

以上三个方法是 EMP 平台的访问控制器行为模式的基本描述。我们在扩展我们的访问控制器时一定是需要实现这三个方法的。

可以注意到，两个接口类的接口方法定义是一样的。这种设计的原因是因为要支持在一个访问控制插件中可能会同时多个访问控制处理器对象的情况。在实际应用中，当我们确定我们只有一个访问控制器处理时，我们可以直接采用 AccessManager 接口进行扩展并在相应的配置文件进行配置即可。当我们需要多个访问控制器对象进行访问控制时，我们需要针对每一个访问控制器的实际处理需求通过扩展 AccessController 接口来实现多个访问控制器对象，在这种情况下可采用 EMP 平台提供基于 AccessManager 接口实现的类 EMPAccessManager 对象来管理所有的 AccessController 访问控制器对象。

当我们需要在一个控制插件中管理多个访问控制器对象时，我们可以直接使用 EMPAccessManager 对象进行配置后来管理多个 AccessController 访问控制器对象，这种

情况下我们就需要 AccessControllerInfo 类来帮助 EMPAccessManager 来管理多个访问控制器对象。

AccessControllerInfo 类是一个当前有效控制器对象的容器。它是一个平台内部使用的容器类，主要由 EMPAccessManager 对象来使用它。它用于保存在针对一个功能点的多个有效的访问控制器对象的集合。我们在即使使用中无须关心它的存在。

4.3.2. EMP访问控制器的使用

访问控制器在 EMP 平台只提供了接口类，需要在应用中根据需要进行接口扩展生成应用所需要的访问控制器对象类。

我们建议在使用 EMP 平台的访问控制插件功能时，采用 EMPAccessManager 作为访问控制插件的管理器，对 AccessController 接口类进行扩展生成相关的访问控制器对象。这样即使当前应用只使用一个访问控制器对象的情况也是可以满足，同时在未来应用需要增加访问控制器对象的情况也是可以很方便的支持，对原有组件无须进行代码级的修改。

在基于 EMP 平台的实际使用中，我们根据所建立的应用系统的不同，可以在 EMP 平台应用的三个层面上进行访问控制插件的处理。在 web 表现访问层、渠道访问处理层和业务逻辑处理层上均可以加载不同的访问控制插件对应用的访问进行控制。

任何访问控制器插件均需要实现 AccessController 接口类来实现应用所需要的访问控制器对象，并将该对象配置到需要进行访问控制的应用层中。

4.3.2.1. Web表现层的访问控制插件使用

首先在表现层我们需要实现一个基于 AccessController 接口类的实现。在 EMP 平台中 web 表现层的请求表现为“*.do”的字符串，在扩展 AccessController 接口中的 checkAccess（）方式时，actionId 的值可直接采用具体访问的“*.do”作为传入参数。在 AccessController 中可通过扩展定义 “*.do”的访问控制权限来完成对请求的访问权限检查。

在实现了对 AccessController 的扩展后，需要在 web 表现层中进行插件配置。这些都是基于外部化配置文件的配置，在运行时刻会自动将插件对象装载。

Web 表现层的基础组件配置文件是在每一个 mvc 应用目录下的 empServletContext.xml 文件中。在该文件中的任一位置加入访问控制器插件的配置信息即

可，系统在 web 表现逻辑调用时将自动调用插件中所配置的访问控制器对 web 请求进行访问控制处理。

具体配置示例如下所示，下面的例子是一个通过访问控制进行请求并发控制处理的一个配置实例。

accessController 对象的 class 属性表明了控制器的实现类，这是一个对当前并发数进行限制处理的控制器的实现。maxConcurrent 表示当前最大并发上限数。

accessItemGroup 对象是对当前并发统计的一个分组，意味着在该组内的所有的 accessItem 对象当前所有的并发数汇总后进行并发数量是否超过限制的比较，如果没有超过则允许执行，否则抛出异常。accessItem 中的 id 实际对应着一个 EMP 平台的.do 的 request 请求。

```
<accessManager class="com.ecc.emp.accesscontrol.EMPAccessManager">
    <accessController
    class="com.ecc.emp.accesscontrol.EMPConcurrentAccessController
    " maxConcurrent="10" slop="3">
          <accessItemGroup id="group1"
          class="com.ecc.emp.accesscontrol.AccessItemGroup">
              <accessItem
              class="com.ecc.emp.accesscontrol.AccessItem"
              id="showBranchDetail"/>
              <accessItem
              class="com.ecc.emp.accesscontrol.AccessItem"
              id="showUserDetail"/>
        </accessItemGroup>
    </accessController>
    <accessController>
        …other access controller配置….
    </accessController>
</accessManager>
```

标签 accessManager 是平台要求的配置标签,不能进行修改，必须使用，用于表明这是一个访问控制管理器插件，它对应着一个实现类，在这里采用的是 EMP 平台提供的实现类：EMPAccessManager。

标签 accessController 是平台要求的一个表示具体访问控制器实现对象的配置标签，不能进行修改，必须使用。它对应着一个实现类，就是我们在具体应用中所需要实现的访问控

制接口类。看上面的例子在 accessManager 标签下定义了很多别的对象，这些对象的定义都是根据具体应用访问控制器所需要的访问控制设计而定义的。

EMP 平台采用的是 IOC 的注入机制的设计，在 xml 配置文件上有足够的灵活性，当我们需要在 accessManager 对象下配置我们的属性或下属对象时，我们按照标准的 javabean 的 xml 配置方式进行配置即可。EMP 在运行期会自动进行对象创建和加载。

4.3.2.2. 渠道访问层的访问控制插件使用

渠道访问层是指非 web 应用方式的对应用系统的访问，在 EMP 平台中提供了三种基础的外部渠道接入访问方式，分别是：TCP/IP 接入访问处理、HTTP 接入访问处理和 WebService 接入访问处理。

三种渠道访问处理采用的统一的渠道访问实现框架，具体的实现设计可参考“渠道访问框架设计专题”内容，在这里不进行描述。

统一的渠道访问实现框架使得三种渠道访问方式，在调用一个应用系统的功能时，均采用 serviceId 的方式进行定义。三种访问方式的基础配置文件为：TCPIPServletContext.xml、HTTPServletContext.xml、WebServiceServletContext.xml 文件。我们将在这三个配置文件中配置我们的访问控制器插件即可。其配置方式与 web 表现层的插件配置方式完全相同。

渠道访问层的访问控制器的接口实现上与 web 应用层有所不同。主要区别在于 web 应用层的功能请求 ID 为“*.do”的方式，而在渠道访问层的功能请求 ID 为渠道访问层所定义的 serviceId 字符串。也就是说，我们在实现 checkAccess（）方法时，参数 actionId 的值可直接采用具体访问的渠道访问层所定义的 serviceId 作为传入参数。

4.3.2.3. 业务逻辑层的访问控制插件使用

业务逻辑层的访问控制插件在功能请求调用上需要两个参数才能确定唯一的业务功能请求。（具体请参考业务逻辑框架专题内容）这两个参数分别是请求功能所在的商业逻辑封装组件的 ID，也就是 BizLogic 配置文件的 ID，另一个参数是具体操作流程的 ID，也就是 BizLogic 配置文件中具体的 operationID。在业务逻辑层实现的 AccessController 访问控制接口的方法 checkAccess()中的参数“actionId”的传入值将采用“BizLogicId.OperationId”的方式进行参数传入，以保证所处理的功能点是唯一的。

在业务逻辑层的访问控制插件的外部化文件配置上，与其他两种模式的配置方式也有所不同。在业务逻辑层中，一个访问控制插件以 service 的方式存在于业务逻辑层的全局服务配置中，我们在 services.xml 文件中进行“accessManager”对象的配置，配置方式与其他两种模式下的插件配置方式相同。Services.xml 文件配置完成后，将该 service 挂结到当前业务结点上 context 上，以保证访问控制插件是可以在整个业务逻辑处理中发生作用的。

关于 context 和 service 文件的含义和配置方式请参考专题“业务逻辑框架内容”。

4.4. SESSION管理

4.4.1. EMP的SESSION管理机制

4.4.1.1. EMP的SESSION管理机制概述

对于大多数的系统而言，都需要对当前登陆的用户的各种会话数据进行管理，包括用户名、密码等，这些数据在用户登陆的状态下都应该是有效的并且在用户任何交易中都可得到，同时不同的用户（或同一用户，不同时间登陆），都应该拥有不同的会话数据，任何用户都无法看到其他人的会话数据。平台是将这些会话数据放在 session 中，另外平台还要对在一定的时间内未操作的会话进行超时管理，这些就需要用到 session 的管理了。

在 EMP 平台中，session 管理是以一种插件的方式嵌入到平台中的。用户可以通过实现平台的 session 管理接口并对配置文件进行修改就可以实现自定义的 session 管理机制。目前，EMP 平台提供了两种 session 管理机制：基于应用服务器的 session 管理机制和 EMP 平台自建的 session 管理机制。用户可以通过修改相关的配置文件使平台在两种 session 管理机制之间进行互换。

4.4.1.2. 基于应用服务器的session管理机制

基于应用服务器的 session 管理机制是 EMP 平台在应用服务器的 session 管理基础上进行了进一步的封装，增加了对会话超时的管理以及增加了监控的操作，使得可以在监控平台中可以对会话进行实时的监控和管理。其中，session 产生是由应用服务器来管理的，即使用 HttpSession。

4.4.1.3. EMP平台自建的session管理机制

基于应用服务器的 session 管理机制只适用于 HTTP 连接的情况，但在项目中，有很多时候并不是一定都通过 HTTP 连接的，例如：直接的 TCPIP 连接，在这种情况下，我们就不能够使用基于应用服务器的 session 管理机制。为了解决这种情况，EMP 平台还有平台自建的 session 管理机制。其中，对于 session 的产生是由该 session 管理机制生成的，并不通过应用服务器。另外，除了会话超时管理和监控的操作以外，EMP 平台自建的 session 管理机制还拥有两种会话跟踪机制：cookie 和 URL_Rewrite。如果是 cookie 机制，则客户的 sessionId 是保存在 cookie 中，如果是 URL_Rewrite 机制，则客户的 sessionId 就会在生成的 URL 中出现（必须使用平台的 tagLib 标签、或者是自己扩展的标签但要支持 URL_Rewrite 机制），通过每次提交的 URL 来重写用户会话的 sessionId。通过配置文件的修改，可以自由的切换这两种机制，当需要跟踪当前会话的 sessionId 时（例如在生产模式下，需要对会话进行跟踪），可以将会话跟踪机制设置为 URL_Rewrite，这样就可以通过产生的 URL 来获知当前的 sessionId。

4.4.1.4. 负载均衡

对于 session 的管理，负载均衡是其中非常重要的一部分。对于基于应用服务器的 session 管理机制而言，可以通过应用服务器的配置实现负载均衡。另外由于是对应用服务器的 session 进行了一定的封装，所以与 EMP 平台自建的 session 管理机制一样，可以通过数据库的操作实现负载均衡，即将 session 对象保存在数据库，利用数据库来实现负载均衡。

4.4.2. SESSION管理的配置

Session 管理是内嵌于 servlet 中的，它的配置是在 empServletContext.xml 文件中，通常的配置如下：

```
<servletContext>
    <sessionManager                                    name="BrowserReqSessionMgr"
        sessionCheckInterval="60000"                                        class="
        com.ecc.emp.session.EMPSessionManager " sessionTimeOut="600000"/>
```

```
    ......
</servletContext>
```

如果要更换成其它的 session 管理机制，则只需要在接口名称不变的情况下，更改实现的类名及相关的属性。

4.4.2.1. 基于应用服务器的session管理机制

com.ecc.emp.session.HTTPSessionManager	
属性名	属性描述
sessionCheckInterval	session 检查的周期。缺省为：60000ms
sessionTimeOut	session 的超时时间。 平台会以 sessionCheckInterval 时间为周期对所有的会话进行检查，如果有哪个会话的时间超过了 sessionTimeOut 规定的时间，就会将该会话删除。该属性需要与应用服务器的 session 超时时间配合使用，即取两者最小值。 缺省为：15*60000ms
sessionTimeoutListener	SessionTimeoutListener 接口，如果定义了该接口，那么在会话检查时，如果发现哪个会话超时了，则会调用该接口进行一定的处理

4.4.2.2. EMP平台自建的session管理机制

com.ecc.emp.session.EMPSessionManager	
属性名	属性描述
sessionCheckInterval	session 检查的周期。缺省为：5*60000ms
sessionTimeOut	session 的超时时间。 平台会以 sessionCheckInterval 时间为周期对所有的会话进行检查，如果有哪个会话的时间超过了 sessionTimeOut 规定的时间，就会将该会话删除 缺省为：15*60000ms
sessionTimeoutListener	SessionTimeoutListener 接口，如果定义了该接口，那么在会话检查时，如果发现哪个会话超时了，则会调用该接口进行一定的处理
sessionTraceType	Session 跟踪机制：COOKIE 和 URLREWRITE。缺省为 COOKIE 机制
sessionIdLabel	若选择 URLREWRITE 的 session 跟踪机制，则该属性用于设置传递 sessionId 的参数名。 缺省为：EMP_SID

4.4.3. SESSION管理的扩展

除了平台提供的两种 session 管理机制，用户可以通过扩展实现客户化的 session 管理。对于扩展的 session 管理，只需要实现 com.ecc.emp.session.SessionManager 接口就可以了。

```
public class HTTPSessionManager implements SessionManager
{
    ......
}
```

定义完实现类后，只需修改 empServletContext.xml 文件中的配置就可以实现 session 管理机制的转换了。

4.5. 事务管理组件

4.5.1. EMP的事务管理机制

4.5.1.1. 事务管理的概念

- 事务管理的概念：

这个概念出自于数据库管理系统中。事务是一个单元的工作，要么全做，要么全不做。事务管理对于维持数据库系统内部保存的数据逻辑上的一致性、完整性，起着至关重要的作用。

- J2EE 的事务管理概念：

传统上，J2EE 开发者有两个事务管理的选择：全局事务或局部事务。全局事务由应用服务器管理，使用 JTA。局部事务是和资源相关的：例如，一个和 JDBC 连接关联的事务。全局事务可以用于多个事务性的资源（需要指出的是多数应用使用单一事务性 的资源）。使用局部事务，应用服务器不需要参与事务管理，并且不能帮助确保 跨越多个资源的事务的正确性。

在本文中，我们要讨论的主要内容是这种局部事务。

- 声明式事务管理和编程式事务管理：

所谓声明式事务管理就是通过外部配置来定义事务管理，通过平台或框架，或者 AOP 机制来执行事务管理。而编程式事务管理，则在应用逻辑中编写特定代码来达到事务管理的目的。

编码式事务管理适用于小规模的、较为简单的应用环境，具有编写简单灵活的优点，而对于应用中有大量事务操作的情况，使用声明式事务管理则更为适合，使得事务处理逻辑与业务逻辑分离，降低了开发难度和维护成本。

4.5.1.2. EMP事务管理机制

4.5.1.2.1. EMP线程安全的事务管理

在面对大量并发访问的情况下，EMP 平台必须保证线程安全。而事务是与当前处理线程密切相关的，所以事务管理要考虑线程安全的问题。

虽然，EMP 的数据库连接是通过应用服务器的资源池来获取的，但是资源池本身解决的是数据连接的缓存问题，并非数据连接的线程安全问题。

按照传统 J2EE 模式，如果某个对象是非线程安全的，在多线程环境下，对对象的访问必须采用 synchronized 进行线程同步。但 EMP 的事务管理并未采用线程同步机制，因为线程同步限制了并发访问，会带来很大的性能损失。

EMP 通过 ThreadLocal 机制来保证线程安全，ThreadLocal 是当前线程的一个局部变量，只有本线程才能访问该对象，（其他线程是无法访问的）。即，当使用 ThreadLocal 维护变量时，ThreadLocal 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

EMP 通过 ThreadLocal 机制来保存当前线程所对应的 Transaction 和数据库连接等资源，并保证线程安全。

4.5.1.2.2. EMP的声明式事务管理

- **交易步骤内置声明事务管理功能**

在 EMPAction 中，就已经实现了 getTransactionDef 方法，而所有的交易步骤代码均继承自 EMPAction 类，所以交易步骤组件内置了声明自己事务类型的功能。

在默认情况下（不去重载该方法），则不需要进行事务管理。

- **TRX_REQUIRED 和 TRX_REQUIRE_NEW**

一个交易步骤可以声明开启事务（沿用当前事务）TRX_REQUIRED，或声明创建新事务 TRX_REQUIRE_NEW。EMP 的 JDBC 交易步骤组件（action）通过重载了 getTransactionDef 方法，将本交易步骤声明为需要事务管理。此外允许在交易步骤的 XML 配置中，定义不同的两种事务模式，一种为“需要统一的事务管理 TRX_REQUIRED”（整个交易的全过程范围）（这种方式为默认方式），另一种为“需要独立的事务管理 TRX_REQUIRE_NEW”（仅在本交易步骤内生效）。

4.5.1.2.3. 事务管理的应用范围

EMP 是以交易步骤（Action）为单位进行事务管理的，每个 Action 可以声明其事务管理要求。

而事务管理的范围则体现为：

交易步骤	事务申明	事务处理
Action a	REQUIRED	T1
Action b	REQUIRE_NEW	T2
Action c	REQUIRE_NEW	T3
Action d	REQUIRED	T1

- 交易步骤 a，申请开启事务（T1）；
- 执行到交易步骤 b，该步骤申请启动一个新事务（T2），T2 事务仅在交易步骤 b 范围内生效，交易步骤 b 成功后，该事务将被 commit，失败时被 rollback，执行完成

T2 事务被销毁；

- 执行到交易步骤 c，该步骤也申请一个新事务（T3），T3 事务仅在交易步骤 c 范围内生效，交易步骤 c 成功后，该事务将被 commit，失败时被 rollback，执行完毕 T3 事务被销毁；
- 执行到交易步骤 d，该步骤申请沿用统一的事务，这里，要注意 T3 事务只在交易步骤 c 范围中有意义，到 d 步骤时，申请沿用的将是上一个 required 声明所开启的事务。
- 当整个处理流程完成后，如果返回成功，则事务 T1 被 commit，否则 T1 被 rollback。

4.5.1.2.4. 用户可干预的事务回滚操作

在交易步骤（Action）中，用户可以根据特定业务逻辑手工干预事务管理，通过编码，获得当前事务，并设置其回滚状态（setRollbackOnly）。

类似下面代码：

在 Action 代码中，

```
EMPTransaction transaction = this.getTransaction();
transaction.setRollbackOnly(true);//or false
```

4.5.2. EMP事务管理的配置和使用

4.5.2.1. 通过XML文件应用事务管理

- 配置事务管理器

事务管理器配置在 services.xml 文件中，通常配置如下：

```
<DataSourceTransactionManager id="transactionManager"
implClass="com.ecc.emp.transaction.DataSourceTransactionManager" />
```

这里是 EMP 平台实现的事务管理器。如果对事务管理器进行重新实现，则在此进行类定义配置即可应用于系统中。

- 配置交易步骤的事务声明

如前文所示，在设计数据库的交易步骤中，只需要声明事务类型即可：

```
<action id="JDBCProcedureAction1"
implClass="com.ecc.emp.jdbc.procedure.JDBCProcedureAction" label="获取客户信息（用户合法）" isBatch="disable" procedureService="JDBCProcedure"
transactionType="TRX_REQUIRED" dataSource="DB2JDBC"
procedureDefine="userLogonGetInfo">
```

4.5.2.2. 自扩展的数据库操作Action如何遵循事务管理机制

在编写自定义的数据库操作 Action 时，可以参考下面内容：

在开始写业务逻辑之前，要注意重载声明事务管理模式的方法：

```
public EMPTransactionDef getTransactionDef() {
        return new EMPTransactionDef(trxType);
}
```

然后编写 execute 方法

首先获得数据源；然后，通过 ConnectionManager 来获得数据库连接；最后要注意，关闭数据库连接

```
dataSource =(DataSource) context.getService(dataSourceName);
connection = ConnectionManager.getConnection(dataSource);
……
ConnectionManager.releaseConnection(dataSource, connection);
```

4.6. 冲正处理

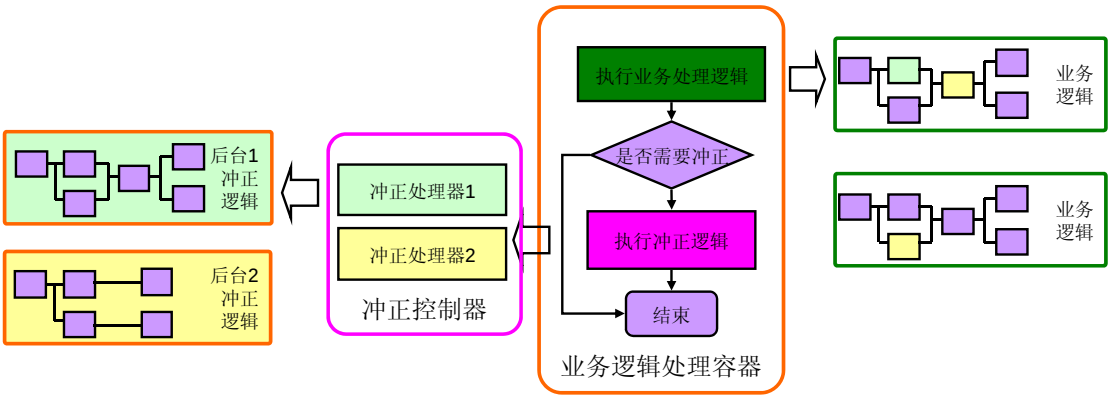
在涉及后台系统的业务逻辑处理过程中，往往会出现一些不确定的状态如响应超时，或者有时涉及多个后台系统的处理时，某个后台处理出错或不确定，需要取消已处理的其他后台系统处理，这就是冲正逻辑，即：将已完成的交易或不确定的交易取消。

在 EMP 体系中，提供了独立于业务处理逻辑的冲正处理体系，也就是具体的业务处理逻辑流程中不需要考虑冲正逻辑，只需要保留处理状态即可，由 EMP 冲正处理器来完成冲正处理。

4.6.1. 冲正处理设计

4.6.1.1. 冲正处理工作原理

冲正处理工作原理如下图所示：



我们假定每一个后台主机的冲正处理流程是可预定的和有规律的。并为每一个后台主机系统提供冲正处理逻辑对象。冲正处理逻辑对象实际上就是一个 EMP 的业务逻辑处理流程对象。

冲正处理是在业务逻辑处理容器执行业务处理逻辑时触发的。在执行业务逻辑处理时，当访问后台主机系统超时或失败时，通过检查“是否需要冲正”的条件，如果满足冲正处理的条件，则执行冲正逻辑。

业务逻辑处理容器在执行冲正逻辑时将控制权交给冲正控制器对象，在冲正控制器对象中管理着针对不同的后台主机系统所提供的冲正处理器，每一个冲正处理器包含着对应的主机系统唯一 ID ，以及该后台冲正处理的逻辑流程对象。冲正控制器根据业务逻辑处理中所判断的需要冲正的主机 ID，调用相应的冲正处理器进行冲正处理。

4.6.1.2. 冲正处理接口设计

4.6.1.2.1. 后台访问处理逻辑步骤

后台访问处理逻辑步骤是 EMP 提供了后台访问处理虚类：

com.ecc.emp.flow.reversal.HostAccessAction，它有一个缺省的参数：hostId，标志着该后台访问所对应的主机 Id，该主机 Id 在该应用系统中是唯一的标识此后台主机系统。在冲正处理中，冲正控制器对象需要根据该 Id 来决定采用哪个冲正处理器进行冲正操作。

所有需要**纳入 EMP 的冲正管理体系的后台访问处理步骤都应该继承此类**，实现其虚方法,指定其参数 hostId(访问的后台系统 ID,与冲正处理器中 HostAccessInfo 的 hostId 对应)。

4.6.1.2.2. 冲正控制器

冲正控制器是一个接口类：com.ecc.emp.flow.reversal.ReversalController。

冲正控制器是 EMP 用于管理和处理冲正，冲正处理的入口，EMP 业务逻辑处理容器如果注入了冲正管理器，则每次处理完业务逻辑后，如果需要冲正，即调用其接口方法 doReversal 方法进行冲正处理。

接口方法定义：public void doReversal(Context context, List accessList)

根据 accessList 中列出的主机访问交易结果，执行必要的冲正处理。

接口方法定义：public void doGlobalReversal(Context context, List accessList)

做全局的反交易，也就是在满足某种条件下，全部回滚原有的主机交易（这种回滚并不是主机交易引起的）。

参数 Context 为当前交易的资源结点，accessList 是当前交易所有已完成的后台主机访问结果对象，用户根据需要进行构造。至少可以包含：主机 Id，主机执行结果等等信息，在 doReversal 方法进行根据主机访问结果对象信息进行判断是否需要进行冲正处理。

4.6.1.2.3. 冲正处理器

冲正处理器是一个接口类：com.ecc.emp.flow.reversal.ReversalHandler。

冲正处理器实现具体后台的冲正判断和处理逻辑。它的接口方法定义如下：

public boolean isNeedGlobalReversal(Context context, HostAccessInfo accessInfo);

判断主机交易是否成功，是否需要对本交易及之前的其他主机交易进行冲正

public boolean isNeedReversal(Context context, HostAccessInfo accessInfo);

是否需要对本交易进行冲正,只针对引起全局反交易的交易做判断

public void doReversal(Context context, ComponentFactory factory);

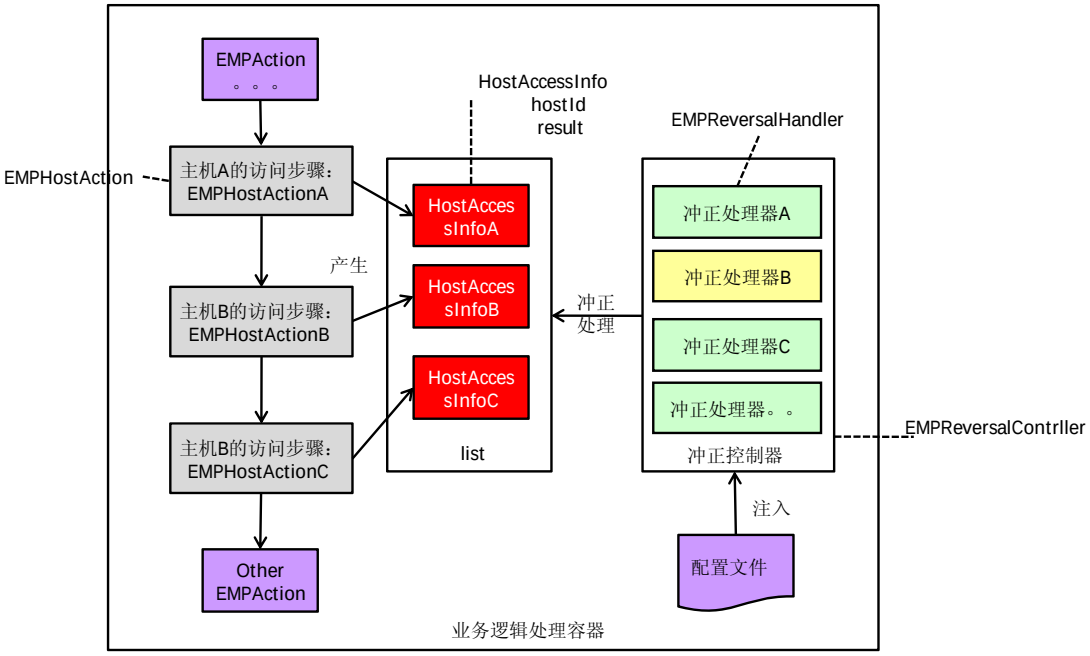
进行冲正处理

public String getHostId();

获得冲正处理器对应的后台 ID

4.6.2. EMP的冲正处理实现

EMP 平台提供了冲正处理的缺省实现。通过将冲正处理控制器注入到 EMP 业务逻辑处理容器中，在业务执行时自动进行冲正检查和根据配置条件完成冲正处理。



如上图，EMP 平台提供了冲正接口的实现。

4.6.2.1. 将冲正控制器注入到业务逻辑处理容器

通过 xml 文件配置的方式将冲正处理器注入到业务逻辑处理容器中，在运行时，也无逻辑处理容器将检查是否有冲正控制器，如果有，业务逻辑处理容器将调用冲正控制器进行冲正处理。

在 service.xml 中定义冲正处理管理器服务，并注入相应的冲正处理器，在 Context 中引用冲正管理器，定义其 ID 为“reversalController”，为每个冲正处理器定义对应的冲正处理逻辑（使用 EMP 冲正处理器时）。

```
<EMPReversalController id=“reversalController” factoryName=“empBiz”>
    <EMPReversalHandler id=“hd1” hostId=“host1” bizId=“host1Reversal” opId=
```

```
“op” globalReversalFormulaStr=“$retValue!=‘0’ reversalFormulaStr=“$retValue=‘2’ />
    <EMPReversalHandler id=“hd2” hostId=“host2” bizId=“host2Reversal” opId=“op” globalReversalFormulaStr=“$retValue!=‘0’ reversalFormulaStr=“$retValue=‘1’ />
</EMPReversalController>
```

如上表中，定义了一个冲正控制器，并在其中定义了两个冲正处理器，分别进行 host1 和 host2 主机的冲正处理，它们所对应的冲正处理流程为：*host1Reversal.op 和* host2Reversal.op。

4.6.2.2. 记录主机访问结果

EMP 平台提供了 HostAccessInfo 类用于记录主机访问结果对象，其中定义了两个关键属性：hostId 和 result，分别表示该 HostAccessInfo 对象所对应的访问主机 Id 和主机访问结果对象。

所有需要纳入到 EMP 平台冲正处理机制中的主机访问均需要继承于 HostAccessActio 类，并配置其中的 HostId，EMP 业务逻辑处理容器执行业务逻辑时在完成主机访问后，将产生一个对应的 HostAccessInfo 对象，并记录该主机访问的 Id 和结果信息。业务逻辑处理容器将 HostAccessInfo 记录在一个独立的 list 中，供冲正控制器访问。

业务逻辑流程完成后，业务逻辑处理容器将调用冲正控制器（如果注入了的话）的 doReversal 方法进行冲正处理。

4.6.2.3. 冲正处理流程

EMP 提供了 EMPReversalController 类，用于管理和执行冲正处理。业务逻辑处理容器在业务逻辑处理结束后调用 EMP 的冲正管理器，冲正控制器逐个查询每个后台业务处理逻辑的处理结果对象爱那个 HostAccessInfo，并判断是否需要做冲正，如果在某个后台处理结果处判断需要冲正，则：

对于引起冲正的后台处理步骤，判断是否需要发起冲正逻辑，如果需要则发起对这个后台冲正交易逻辑。EMP 提供了冲正处理器：EMPReversalHandler 类进行冲正处理。

从此后台开始往回逐个发起后台的冲正交易处理逻辑。

4.6.3. EMP冲正处理的扩展

EMP 冲正处理提供了实时冲正的处理机制，没有提供更多的冲正策略管理和控制，如异步冲正实现等等。EMP 冲正提供的是冲正的机制，我们可以通过继承 EMP 的冲正控制器实现类或实现冲正控制器接口类来丰富 EMP 冲正处理功能。

如果要实现异步冲正处理，则可以通过实现 EMP 的冲正控制器接口，将冲正相关信息进行存储，并配置外部的定时服务组件，定期进行冲正表扫描，根据扫描结果调用 EMPReversalHandler 进行冲正处理。

4.7. 数据类型组件

在项目开发中，很经常遇到这样一种情况：数据在页面端显示的样式和数据在后台或者数据库中存储的格式有时候是不一样的，或者说数据显示的格式可能是带着各种修饰符的字符串（如$符号等），但在运算、存储场合下用到的格式却是数值型数据。在一般的项目开发中，遇到这种情况往往是在需要显示的 JSP 页面手写 JS 代码、或者是在单个组件中手写 JAVA 代码来实现数据的转换。而现在 EMP 平台是将这一块功能单独做成了一个独立的模块，由平台来负责数据的验证和转换，而开发人员只需要在配置文件中设置为需要显示与转换的格式就可以了。

该模块主要分成两部分：前端与后台。前端主要是通过编写 JS 程序来实现，后台则是使用 Java 程序实现。当用户在页面上进行输入的时候，首先会调用相应的 JS 程序按照设定好的格式进行转化并显示，然后再对转换后的字符串进行验证，最后再传到后台。如：输入的是金额数据 12345，则可能转化为 12,345.00 格式，然后在提交时进行验证再传到后台。对于后台而言，平台在更新数据时就会根据输入数据的类型调用相应的程序对输入的数据进行校验和转换。而之所以将该模块分成前端与后台是为了安全考虑，防止了不符合格式的数据通过其它的途径转到后台时未经过校验就被使用。

4.7.1. 设计原理

在这里，我们以金额型数据为例来说明数据类型转换和校验的原理。首先，要想使用数据类型就必须定义好各种基本的数据类型的格式，如金额型数据要可以设置精度、小数点位数等，这些在平台附带的配置文件 dataType.xml 中已经有了一些基本的数据类型。其次要定义具体的数据类型的格式，如同样是金额型的数据，一般情况下所需要的精度与股票市场中所需要的精度就不一样，这就需要创建两种数据类型：normalCurrency 与 stockCurrency，而两种数据类型的精度设置不一样，对于具体的数据类型，平台附带的配置文件是 dataTypeDefine.xml。最后在页面以及交易的配置文件中针对每个具体的数据元素可以选择具体的数据类型，例如一般情况下可以选择 normalCurrency 类型，而用于处理股票金额的数据元素则使用 stockCurrency 类型。

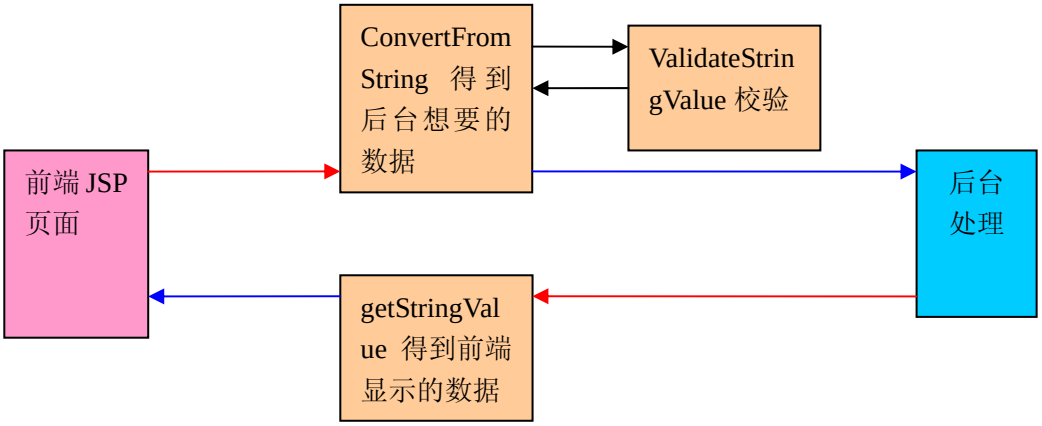
为了在页面端使用数据类型，就必须使用平台自带的 TagLib 或者是自定义的页面标签。这里以 ctp:input 标签为例，如果想要让输入的字符串显示为金额型数据（normalCurrency），则必须在 ctp:input 标签中设置 dataType=”normalCurrency”。这样，平台在解析该标签时会从 dataTypeDefine.xml 中得到 normalCurrency 类型数据是属于哪个基本数据类型（Currency）以及它的相关配置（如精度为 10 位、小数点为 2 位等），然后根据基本数据类型在 dataType.xml 文件中找到该类型的转换与校验的 JS 代码，并输出到页面中。同时在生成的 HTML 标签中会加上 onblur=“相应的 JS 转换函数”。同时在提交 ctp:form 的时候会调用相应的 JS 校验函数对输入框中的数据进行校验。

对于后台的校验与转换而言，只需要对业务逻辑中的输入数据设置相应的具体数据类型，平台在对数据模型进行更新时，会根据数据元素所设置的数据类型找到相关的基本数据类型及相关配置，然后调用该基本类型的实现类中的转换函数进行数据的转换，在数据转换函数中又会先调用相关的校验函数对数据进行校验，如果输入的数据不符合格式，则会抛出异常。

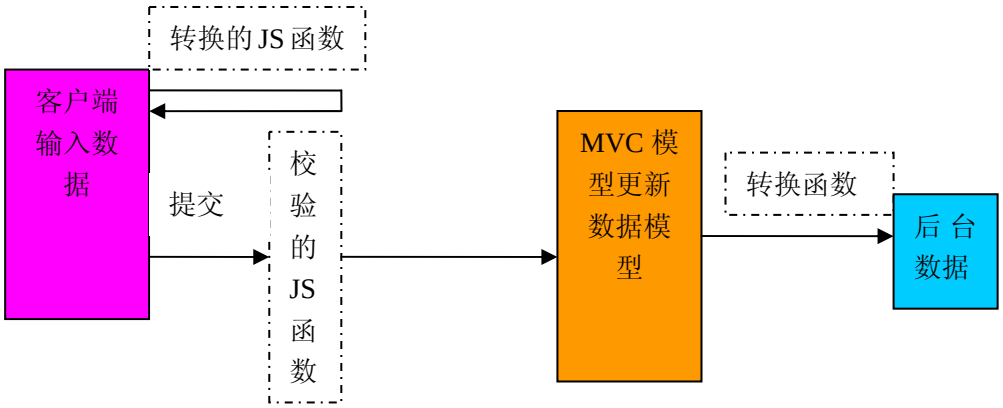
虽然 EMP 平台的数据校验与转换模块是包括前端与后台两部分的,但是这两部分不是必须关联在一起的，也就是说在前端，开发人员可以通过自己的 JS 来转换数据的格式，然后在逻辑流程的输入数据中定义相关的格式，当然也可以只用前端的显示格式而不需要后台的转换。

4.7.2. 总体框架

前端与后台数据相互转换的总体流程及后台的具体流程：



在前端，JSP 页面中输入的数据转换到后台数据的具体流程：



4.7.3. 使用方法

首先先定义一个具体的数据类型，如金额型（精度为 15，小数点后 2 位，输入\输出有小数点，后台数据也带有小数点，显示金额符号以及后台数据不是字符串），在 dataTypeDefine.xml 文件中定义这样一条配置：<Currency id="normalCurrency" name="一般余额数据" precision="15" showDot="true" scale="2" keepDot="true" showSymbol="true" keepStringValue="false"/>

如果需要在前端使用平台的数据类型及校验，则使用平台的 tagLib 标签并设置数据类型，如输入框：<ctp:input dataName=”inputCurrency” dataType=”normalCurrency” />，同时这个输入框必须是在一个 ctp:form 标签内的。这样在客户端，用户输入一条数据如：12345.00，客户端就会调用相应的 JS 函数将数据转换成￥12,345.00，在提交时，客户端又会调用相应的 JS 函数对￥12,345.00 进行校验，通过后提交到后台。

如果需要在后台使用平台的数据类型及校验，则在业务逻辑的输入数据中定义相关数据的数据格式。如前面输入的 inputCurrency 数据域，可以在输入数据中定义其数据格式为 normalCurrency：

```
<input>
    <field id="inputCurrency" dataType=”normalCurrency”/>
    ......
</input>
```

4.7.4. 具体配置说明

4.7.4.1. Currency

Currency 类型一共有 8 个属性，其中：min、max 是用于设置金额的最大值与最小值，precision 是金额的精度，即金额数据的最大长度，scale 是金额数据小数点后的位数。这两个属性合起来也可以设定金额数据的最大值与最小值，它们与 min、max 属性合起来使用时，取其中的最小范围。如最大值为 1000，而 precision 为 7，scale 为 2，而最大值也只能是 1000.00，若最大值为 100000，而 precision 为 5，scale 为 2，而最大值为 999.99。

showDot 属性定义的是在客户端显示的字符是否带有小数点，如 1234.00 数据，若是不带小数点，则显示为 123400；若带有小数点，则显示：1,234.00。

keepDot 属性定义的是后台的数据是否带有小数点，如：1234.00 数据，若是不带小数点，则后台存放的数据为：123400；若是带有小数点，则后台存放的数据为：1234.00。当然其中小数点的位数为 2。如果小数点的位数为 3，则若是不带小数点，则存放的数据为：1234000；若是带小数点，则存放的数据为：1234.000。如果小数点的位数为 1，则该数据不合法。

keepStringValue 属性定义的是后台的数据的格式是否是字符串，如果是，则以字符串的形式存放金额数据，如果不是，则是以 BigDecimal 格式存放金额数据。无论是否是字符串，存放的数据都不带有各种修饰符，即字符串也是由 BigDecimal 直接转换成的。

showSymbol 属性定义的是前端显示的字符串是否带有金额符号，即是否带有“￥”符号。

4.7.4.2. Date

Date 类型一共有 6 个属性：其中 keepStringValue 属性与其它类型一样，如果 keepStringValue 为 false，则后台存放的数据格式就为 Date 格式。

inputFormat 属性定义的是客户端日期显示的格式，如 2006 年 3 月 1 号，若 inputFormat 为“yyyy/MM/dd”，则在客户端显示的字符串为：2006/03/01。其中表示日期的字符必须是 y、M、d、H、h、K、k、a、m、s（具体代表的含义与 JAVA 程序中表示的含义一样）。如果是其它字符则看成是修饰符号，如“年”、“月”、“日”等。

valueFormat 属性定义的是后台的数据格式，这与 inputFormat 设置方法一样。

inputAmPm 属性定义的前提是 inputFormat 属性中定义中有字符 a。该属性的作用是显示 am/pm 的字符，例如设定为“上午|下午”，如果显示格式中有要求显示 am/pm（即显示格式中有字符 a），则显示的字符串会在相应的位置显示上午或者是下午。如果设定的是“am|pm”，则显示的则是 am 或者是 pm。如果在显示格式 inputFormat 中没有定义字符 a，则该属性不起作为。

valueAmPm 与 inputAmPm 属性类似，只是作用的是后台的数据。

Type 属性定义的是后台存放的数据是：单纯的时间格式、单纯的日期格式或者是同时包括日期与时间格式。该属性受到 valueFormat 和 keepStringValue 属性的影响。首先后台的数据必须是存放为字符串格式，Type 属性才会起作用。另外，Type 属性必须符合 inputFormat 属性，即 inputFormat 属性如果没有日期，则 Type 属性就不能选择日期或日期与时间，否则则抛出异常，因此，Type 属性必须与 inputFormat 合在一起使用。另外，Type 属性与 valueFormat 属性也有一定的关系，如果 valueFormat 定义为空，且存储的数据为字符串，则存储的格式就由 Type 属性决定。

4.7.4.3. MaskDataType

MaskDataType 主要是用于帐号数据的显示与校验。该类型共有 6 个属性（keeyStringValue 除外）。regExp 属性定义的是帐号的正规表达式字符串，即使用正则表达式对帐号进行判断。

Mask 属性定义的是帐号在页面端显示的格式：如 999-***-99。其中由数字代表着该位置显示的是帐号相应的号码，而“*”（该符号由另一属性 flag 设置）则代表的是显示时，在帐号的该位置显示为“*”，而不是原先的帐号数字。Mask 属性必须与 regExp 属性符合，否则是输入串无法通过校验或者是输入串显示混乱。

showSeparator 属性设定的是帐号的显示是否要包含分隔符，如 999-***-99 中的“-”字符。如果在 Mask 属性中没有分隔符，则该属性不起作用（即都当作无分隔符）。其中在 Mask 属性定义的字符串中，除了数字与 flag 定义的字符外，其它的字符都当作分隔符。

keepSeparator 属性设定的是后台存储的帐号数据是否包含分隔符，该属性与 showSeparator 一样，如果 Mask 属性中没有分隔符，则该属性也不起作用。

keepFlag 属性设定的是在页面端输入的字符串是否使用 flag 设定的字符来屏蔽相关的数字。如果需要屏蔽，则在 Mask 属性定义的字符串中在相关位置应该要有屏蔽字符出现。否则就当作都不需要屏蔽。

flag 属性定义的是用于屏蔽的字符。该属性只能是单个字符（且由于帐号的特殊性，该字符不应该是数字，缺省情况下是“*”）。

4.7.4.4. StringType

StringType 主要是用于定长字符串的相关显示与校验。该基本数据类型共有四个属性。length 属性设置的是该字符串的长度。padChar 属性定义的则是如果输入的字符串达不到设定的长度时用于补足的字符（在实际中，该属性一般只允许是单个字符）。align 属性设置的则是输入字符的对齐方式。如果长度设置为 5，输入的字符串为“123”，padChar 设置为 0，而 align 属性定义为 right，则在页面端显示为“00123”。

最后一个属性 type 用于笼统的设定输入字符串的类型，其中共有三种类型：“9”表示为输入的字符串只能是数字；“A”表示为输入的字符串只能是字母；而“X”则不限。该属性必须与 padChar 属性相配合，若 type 属性为“9”，则 padChar 属性就不能是除了数字外的其它字符。

4.7.4.5. 其它类型

另外还有几个基本数据类型：ByteType；ShortType；IntegerType；LongType；FloatType；DoubleType。这几个基本数据类型的相关设置与意义都是一样的。如果 keepStringValue 为 false，则在后台存储的数据格式就是相应的数据格式（如 ByteType 存储的为 Byte 型数据；DoubleType 存储的为 Double 型数据）。另外还有 min、max 则定义的是相应的最大值与最小值，缺省情况下是相应数据格式的最大值与最小值（如 ByteType 缺省的最大值与最小值为：Byte.MAX_VALUE 与 Byte.MIN_VALUE）。

4.7.5. 扩展

除了可以使用平台自带的基本数据类型外，平台允许开发人员按照相关的接口对数据类型进行扩展以实现更加复杂多变的数据格式。

扩展的基本数据类型也是分前端与后台两部分进行。首先需要在 dataType.xml 文件中定义新的数据类型，还是以 Currency 类型为例：定义 dataType 标签<dataType id="Currency" name="金额数据" implClass="com.ecc.emp.datatype.CurrencyType" >，其中 id 表示的是新的数据类型的 ID，implClass 定义的是后台的数据类型校验与转换的实现类。然后定义该基本数据类型的属性，这里定义的属性 ID 必须跟后台的实现类的属性名称一样：

```
<attributes>
        <attr id="precision" name="精度"/>
        ......
<attr id="keepStringValue" name="后台是否转化为字符串形式" attrType="boolean" defaultValue="false"/>
</attributes>
```

属性的名称主要是在 IDE 中对属性设定的时候显示，属性中的其它设置是为了 IDE 设定属性时更好的进行操作，其中 attrType 可以选择 boolean，那么在 IDE 中设定该属性时就是一个下拉框，可以选择 true/false。dafaultValue 是在设定该属性时的默认设置。如果未定义为两个设置则在 IDE 中设定属性时手工输入相关的配置。除了自己定义的属性外，所有的基本数据类型都有一个属性：keepStringValue，该属性表示的是后台的数据是否转

化为字符串形式，例如：Currency 类型，如果不转化为字符串形式，则后台的数据是以 BigDecimal 数据格式存放。

在定义完该基本数据类型的属性后，接着就要定义该基本数据类型校验与转换的 JS 函数。首先必须先定义校验/转换函数的函数名称，如：<validateJS function="validateCurrency(curField, $max;, $min;, $precision;, $scale;, $showSymbol;, formatErrorMsg, rangeErrorMsg)">。其中 validateJS 标签代表的就是校验的 JS 函数，函数的名称由 function 属性定义，validateJS 标签是在 dataType 标签的子标签。另外，convertorJS 标签则是表示转换的 JS 函数。

然后在 validateJS/convertorJS 标签中，定义 script 标签，并且以文本形式定义相关的 JS 代码：

```
<script><![CDATA[

function convertCurrency ( element,scale,showDot, showSymbol )

{

.....

}

]]></script>
```

这样，一个基本数据类型的定义及前端的 JS 函数就已经完成了。

完成前端 JS 后，接着就必须定义后台的相关的实现类。首先类名必须是刚才定义的实现类的类名，而且该类必须继承自 EMPDataType 虚基类。在实现类中，在 dataType.xml 中定义的属性作为它的成员变量，同时使用标准的依赖注入方式定义属性的 get/set/add 方法。在实现类中，必须包含着四个方法：1、Object convertFromString(String strValue, Locale locale) throws InvalidDataException；2、String getStringValue(Object value, Locale locale)；3、boolean validateStringValue(String value) throws InvalidDataException；4、boolean validateStringValue(String value, Locale locale) throws InvalidDataException。其中：convertFromString 是数据转换的方法，getStringValue 则是数据从后台转换成前端显示的方法，validateStringValue 则是数据校验的方法。。

其中，在方法 convertFromString 中，必须首先调用 validateStringValue 方法，如果数据未通过校验，则抛出 EMPException 异常。

4.8. 异常处理

在 EMP 平台上如何进行异常处理。EMP 平台是一个分层架构设计，基本可以分为业务逻辑层和表现逻辑层。在两个层次上都需要进行异常的处理。EMP 的异常处理设计是基于 java 的异常机制来实现的。

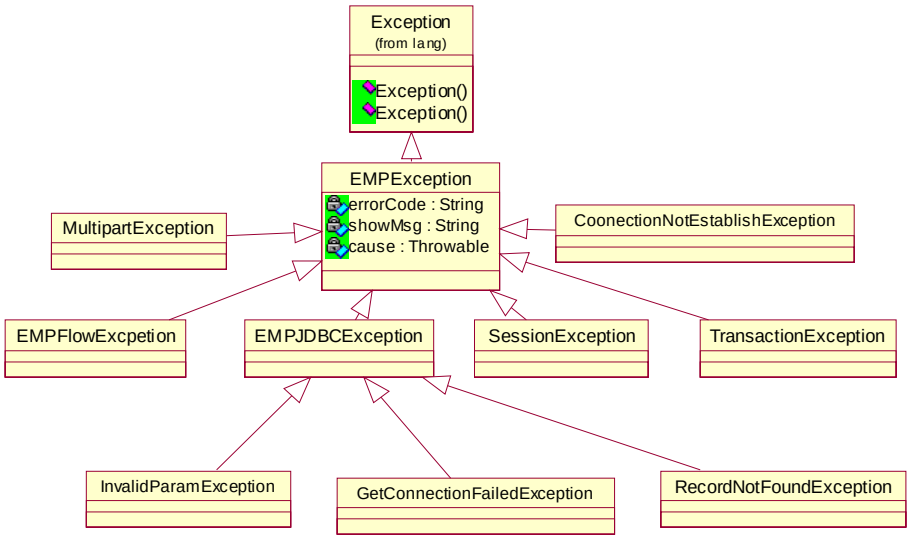
EMP 平台通过对 java 的 exception 基础类的扩展，形成了 EMP 平台的不同类型的异常扩展类。在实际应用中，通过定义和抛出 EMP 平台的异常来通知和捕获异常。

4.8.1. 基本原理

4.8.1.1. EMP平台异常类关系

EMP 平台提供了多种类型的异常对象类。这些类都是继承于 java 平台的标准异常类：Exception 类来扩展出来的。EMP 平台的异常类继承关系基本分为三层，更多继承层次用户可以在实际应用中去进行扩展。

EMP 平台的异常对象类的大致继承关系如下图所示：



EMPException 是 EMP 平台的异常基类，它继承与 java 的标准异常基类 Exception，在此基础上扩展了三个关键类属性，分别是：

属性名	属性类型	含义
errorCode	String	对应到应用的错误码
showMsg	String	对应到应用的错误显示信息

cause	Throwable	对应到具体的系统异常对象

针对 EMP 平台的各个关键框架和基础组件，EMP 平台提供了继承于 EMPException 类上的各个类型异常类。大多异常类扩展只是基于类型的封装，异常类基本属性并没有发生改变。

EMP 平台提供的类型扩展异常类主要有以下五种：

类名	类型	具体含义
EMPFlowException	业务逻辑处理异常	EMP平台的业务逻辑处理层异常
EMPJDBCException	数据库操作异常	数据库操作类型异常
SessionException	会话处理异常	用于描述web层会话管理异常
TransactionException	事务管理异常	用于描述事务一致性管理处理异常
MuliPartException	上传协议异常	专用于描述文件上传的web层处理异常

4.8.1.2. EMP的异常处理设计原则

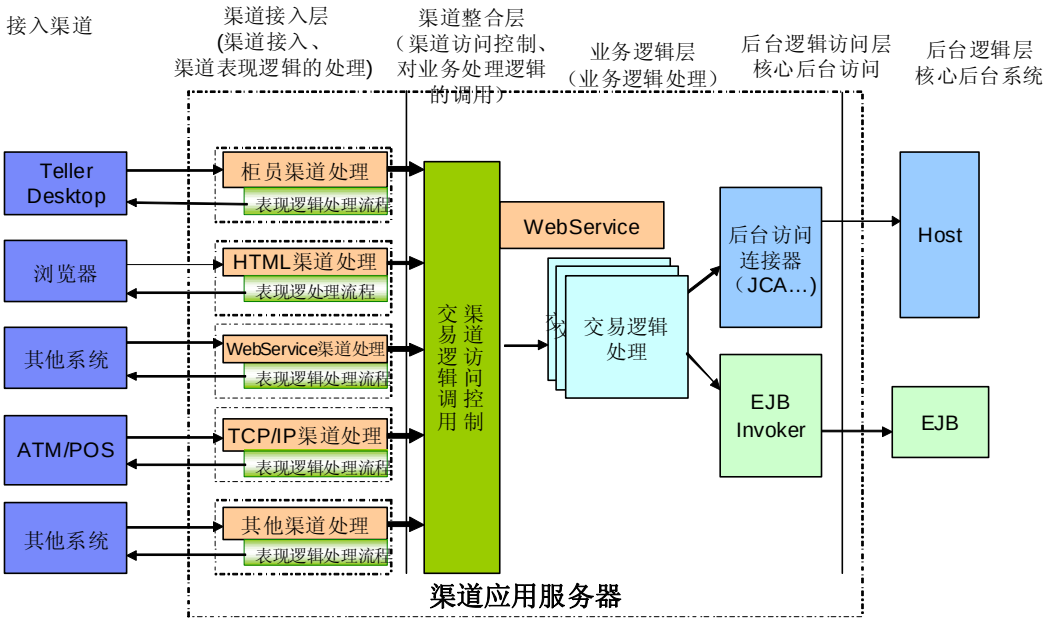
应用的异常处理的好坏往往关系到一个系统的可用性，一个平台的异常处理机制是否完善，可以从以下三个方面来看。

首先一个平台要有自己的异常处理机制才能帮助用户在使用平台建立应用功能时遵循异常处理规范，将异常处理交给平台去处理，而集中精力完成应用功能的开发。

其次是异常处理的层次，异常最终要能够提示用户操作失败的可理解原因，这意味着异常不但需要统一的处理机制，还需要灵活的展现机制。针对不同的应用需要用不同的语言去描述异常发生的原因，在提供给用户时尽量方便地转换为业务原因而不是赤裸裸的技术原因。

最后是异常的准确记录，异常在前端为操作人员可以展现为各种业务原因，但在后台应提供给技术维护人员的是清晰可见的技术原因。异常与平台日志的结合，帮助系统维护人员更块的定位异常原因并及时进行系统修复。

以上三个方面也是 EMP 平台在进行异常处理设计所遵循的原则。



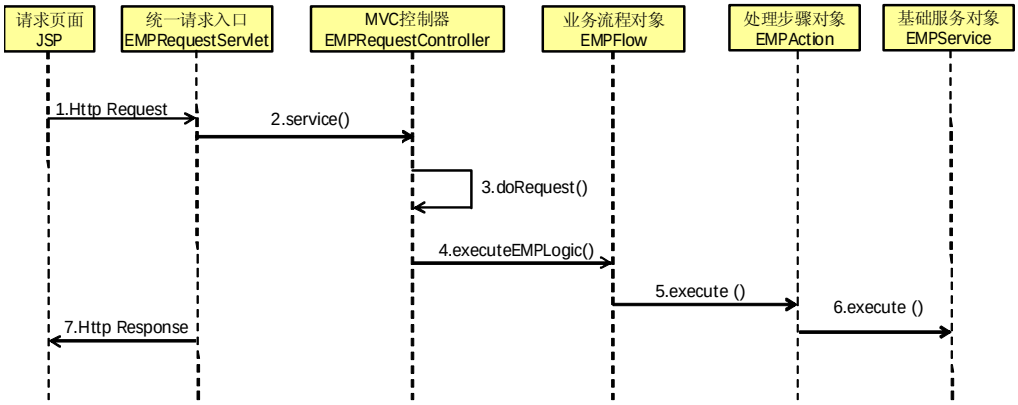
如上图，我们将银行渠道系统采用分层结构的方式进行解构，分别是接入层、渠道表现层、渠道整合层、业务逻辑层、后台逻辑访问层和后台数据层。不同的渠道需要不同的渠道表现，不同的渠道有不同的渠道逻辑，但相同的渠道交易请求所调用同样的业务逻辑处理。在分层结构中，银行多渠道接入层的设计核心在于在银行渠道系统的渠道表现层、渠道逻辑层和业务逻辑层进行完全的解藕处理。各个层面之间采用接口化扩展的标准处理进行交互，在应用上保持整个系统框架的稳定性，采用外部化配置模式进行渠道接入功能扩展的定制，提供最大化的多渠道接入的高效稳定和灵活可扩展能力。

EMP 平台所提供的多渠道接入框架正是采用这种分层设计思想，通过分层解藕的方式为用户提供的多渠道接入处理框架。

4.8.1.3. EMP的异常处理设计

EMP 平台的异常处理设计是分为业务逻辑层的异常处理设计和表现逻辑层的异常处理设计的。业务逻辑层在异常处理设计上主要考虑要能够将真实异常抛出去，以便平台的异常处理机制能够获得真实异常并进行处理。表现逻辑层在异常处理设计上主要考虑是对捕获的异常进行转化和展现的灵活配置处理。

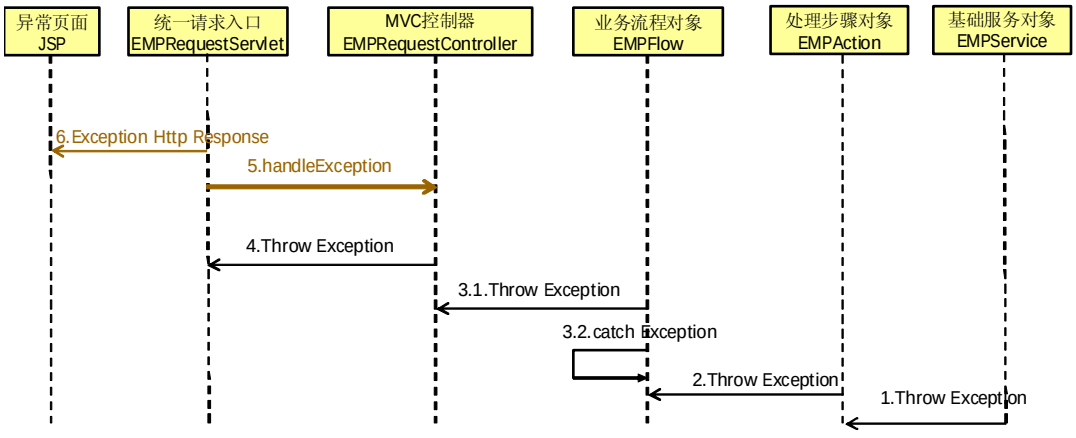
考察 EMP 平台从页面访问到最终调用业务逻辑处理的基本设计原理如下图所示：



从页面请求到最终流程执行，servlet 作为统一入口将请求数据发往请求的具体的 mvcController 对象进行处理。mvcController 根据请求所调用的业务流程调用 EMPFlow 对象进行业务流程的调用，EMPFlow 业务流程通过调用流程中的 EMPAction 对象来执行整个流程，action 对象通过对所需要的基础技术服务对象 EMPService 的调用来完成实际功能操作：如访问数据库、文件处理、数据转换等等。

这是一个对象嵌套的操作流程，统一处理的 servlet 对象和 mvcController 对象是 web 表现逻辑层的处理对象，EMPFlow、EMPAction 和 EMPService 对象是业务逻辑层的处理对象。在我们实际处理中，遵循业务逻辑层根据需要抛出异常；表现逻辑层捕获真实异常并完成展现。

EMP 平台的异常处理机制如下图所示：



步骤一：EMPService 对象所发生的异常均抛出，由它的调用者处理。

步骤二：EMPAction 对象所发生及捕获的异常均抛出，由它的调用者处理。

步骤三：EMPFlow 对象所发生及捕获的异常采用两种方式进行处理，一种是将该异常继续抛出，由它的调用者进行处理。一种是根据业务需要对该异常直接进行业务异常转换，直接在流程对象中处理完毕。

是否将抓住的异常进行业务异常转换处理，可以在业务流程外部化配置中进行配置说明，在需要对异常进行业务信息转换处理时，我们可以在该操作步骤（EMPAction）定义的跳转条件中定义该异常的下一步处理步骤，如下定义：

<transaction condition=”exception:exceptionClassName” dest=”destActionName”/>

Condition 中定义了我们需要对那一类型的异常进行直接处理，exceptionClassName 是指定的异常的类名称，根据类名称进行下一步骤跳转。这种异常处理模式的要求是根据业务流程处理时的需要要增加的。往往一个业务流程在中途出现异常时，该业务流程并不能马上直接跳出业务流程执行对象，而是要根据异常的情况进行下一步的处理，比如在最通常用到的情况是，当我们的业务流程抛出异常时，我们还需要记录该处理过程的日志，并更新日志的状态为失败。所以我们必须要求业务流程继续向下流转，而不能直接跳出。这种情况下就要求我们在业务流程处理中，针对异常信息进行业务信息的转化了。在 EMPException 类定义中，定义了 errorCode 和 showMsg 两个属性，这两个属性实质上就是异常信息向业务信息转化时的对应信息。

在具体的 action 执行语句中，在捕获异常后，均需要对这两个属性进行设置，它将给前端用户展现异常所代表的业务信息。

步骤四：mvcController 对象在发生或捕获异常后，继续抛出到 servlet 对象。

步骤五：servlet 对象在捕获异常后，将回调异常抛出者对象 mvcController 对象的 handleException()方法进行异常处理。

步骤六：根据异常处理生成的异常信息 jsp 页面，返回给请求用户前端。

4.8.1.4. ExceptionHandler对象设计

EMP 平台对异常的展现提供了灵活的处理手段，系统平台提供了缺省的异常展现方法，通过也提供了可扩展的异常处理方式。

在 EMP 平台中提供了异常处理类 ExceptionHandler，该类可以通过配置对每一种类型的异常提供独立的异常处理页面。

ExceptionHandler 类的设计如下：

Class	ExceptionHandler	Package	com.ecc.emp.web.servlet
Class Format	public Class ExceptionHandler		
Implements			

Name	Parameters	Return Value	Exceptions	Description
getExceptionView	Exception exception	ExceptionView		根据传入的异常对象，获得该异常对象定义的异常处理页面
addExceptionView	ExceptionView	void		从配置文件中获得定义的excpetion对应的exceptionView对象

配置文件的内容信息请看“ExceptionHandler 的配置处理”章节，有详细描述。在 ExceptionHandler 类中，getExceptionView 方法是一个关键方法，它从配置的对象中，根据 exception 的类型获得它所对应的 jspview 对象，并返回该对象，并由系统机制将页面返回给用户前端。

在系统的 mvc 框架设计中，当有异常出现时，我们在通过页面进行异常表现时，需要从 request 对象中拿到所有的数据对象和异常对象。系统已经将我们所需要用到 context 对象和 exception 对象分别以属性方式存放在 request 对象中。在需要获取时，这两个数据对象的属性名称都可以通过常量进行了定义：

Context 对象在 request 中的名称：EMPConstance.ATTR_CONTEXT

Exception 对象在 request 中的名称：EMPConstance.ATTR_EXCEPTION

4.8.2. EMP平台异常处理的使用方法

4.8.2.1. 定义我们自己的异常类

在一些新的应用中，我们会根据需要定义一些应用所需要的异常类，这些异常类均可以继承于 EMP 平台所提供的所有异常类。定义完成后在应用直接使用即可。

我们根据需要进行异常类的设定，一般情况遵 EMP 平台异常定义规则，异常所包含的信息是足够的，我们在定义新的异常类时，往往是为规范一个新的异常类型，我们只需要继承于 EMPException，通过新的类名来表明我们的新异常类型即可满足应用对异常的需要。如下示例：

```
Public class NewEMPException extends EMPExcetion{
    Public NewEMPException(){
```

```
            Super();
        }
    }
```

4.8.2.2. 在EMPAction编程对异常的处理

在 EMPAction 编程中，我们要根据具体的业务功能进行系统异常到应用错误信息的转化。在 EMP 平台上缺省定义了两个表现应用业务信息的数据对象，它们的名称，一个叫 ErrorCode,用户描述应用所定义系统错误码；一个叫 ErrrorMessage，用来描述系统错误码所对应的描述信息字符串。

当我们在 EMPAction 编程中，我们通过 try/catch 模块可以抓住任何异常，根据我们的业务需要，当一个异常类型可以转化为我们所需要的业务错误信息时，我们应设置我们的业务信息码，在设置完应用错误信息码后，再将错误抛出。在 action 的编程中所有的异常均需要抛出。如下举例，在一个扩展于 EMPAction 的类对象的 execute（context）方法中：

```
Try{
}catch(needDoingException e){
    setErrorInfo(e);//设置应用错误码的方法
    throw e;//将异常再次抛出
}catch(Exception e){
    Throw e;//不需要应用设置错误码的异常，直接抛出
}
```

4.8.2.3. 在EMPService扩展编程对异常的处理

同 EMPAction 编程类似，因为在 EMPService 对象中，也是可以得到 context 对象的，我们也可以根据需要在捕获异常的同时，设置错误信息码，然后再将异常抛出。

需要记住的是：无论是 EMPService 还是 EMPAction，在处理异常时，最终都需要将异常抛出，交给它的调用者进行处理。

4.8.2.4. 在EMPFlow中的异常处理配置

EMPFlow 对象是一个外部化的配置流程对象。我们可以在配置文件来表达我们是否需要将某一异常进行特殊处理，否则就直接抛出。

4.8.2.4.1. 在流程中直接处理异常

假设有一个 xxAction，在程序实现时，可能会抛出一个 newException 类异常，在流程配置中，我们需要当该步骤发生该异常时，流程跳转到 excepionAction（异常处理步骤）上去。这种情况下，我们就需要在 action 的跳转流程中对该 exception 的处理跳转进行配置了。如下：

```
<flow name=”xxxxflow”>
    ….
    <action id=”xxAction”>
        <transaction condition=”$retValue=’0’” dest=”next”/>//当正常结束时，跳转到下一步
        <transaction condition=”exception:newException” dest=”exceptionAction”/>// 当出现newException异常时，跳转到exceptionAction步骤上。
    </action>
    ….
</flow>
```

这种情况下，意味着该异常在流程中间被处理了，该异常将不会继续抛出，流程还可以继续执行。但我们可以通过系统日志看到该异常的详细信息，而在前端展现时将无法察觉到该异常的发生。

在流程中直接对异常进行处理，最关键的一点就是，异常将不会被抛出，同时流程还将按照定义的流程继续执行。

4.8.2.4.2. 从流程中直接抛出异常

当我们不定义 excpetion 的跳转条件时，或者我们抛出的异常在已定义的异常跳转条件中无法找到匹配项时，我们将直接抛出异常，并终止该流程的执行。

从流程中直接抛出异常，最关键的一点就是，异常将被抛出到流程对象的调用者，同时流程将被终止。

4.8.2.5. 在web MVC框架下的配置和使用

4.8.2.5.1. 系统缺省配置处理

EMP 平台提供的 web 处理框架 MVC 框架下，通过一个缺省的错误页面“error.jsp”页面进行错误信息展现。该页面在每一个基于 web 的应用项目中均在建立项目时自动创建。用户可对该页面进行编辑，产生符合应用要求的错误页面处理。

当整个应用中没有配置其他的异常信息处理对象时，系统将自动采用该页面进行错误信息的展现。在 mvc 页面跳转定义的处理，请参考专题文档 “MVC 框架设计专题”文档中的描述。

在我们定义了异常处理对象 ExceptionHandler 后，系统将调用该 handler 对象调用不同异常所对应的展现页面对象进行处理。

4.8.2.5.2. ExceptionHandler的配置处理

我们可以为 servlet 配置一个 handler 处理对象，也可以为不同的 controller 配置 handler 处理对象。系统在调用过程中，首先寻找 controller 对象中是否配置了 handler 对象，如果配置了则使用 conroller 对象中所配置的 handler 进行异常展现的调用页面处理，否则寻找在 servlet 的配置信息中，是否配置了 handler 对象，如果配置了，则调用该对象，如果没有找到 handler 对象的定义，则采用系统提供的默认的 error.jsp 页面进行处理。

4.8.2.5.2.1. 统一的ExceptionHandler的配置

统一的 ExceptionHandler 的配置，意味着对应用的所有的 controller 对象抛出的异常采用该统一的 exceptionhandler 进行处理。它的配置信息在基础的 mvc 配置文件：empServletContext.xml 文件中。

```
<ExceptionHandler
class="com.ecc.emp.web.servlet.mvc.ExceptionHandler">
       <ExceptionView id="sessionException" url="sessionException.jsp"
exceptionName="SessionException"/>
       <ExceptionView id="exception" url="error.jsp"
exceptionName="exception"/>
    </ExceptionHandler>
```

ExceptionView 标签配置了对于不同的异常类型，采用哪个对应的错误页面进行处理，如上边的配置：类型为 sessionException 的异常对象，将采用 sessionException.jsp 页面对象进行展现处理。

4.8.2.5.2.2. Controller定义中的ExceptionHandler的配置

在 controller 中定义 ExceptionHandler 对象，意味着该异常处理展现对象只对该 action 的处理有效。配置的信息内容与统一的 ExceptionHandler 的配置方法一样。它表现为在一个具体的 action 定义中加入对 exceptionHandler 的定义，如下：

```
<action id="showBranch" type="normal" checkSession="false">
        <jspView id="showBranch_index.jsp" url="showBranch_index.jsp"/>
    //在此加入 ExceptionHandler 的定义

        <ExceptionHandler
    class="com.ecc.emp.web.servlet.mvc.ExceptionHandler">
        <ExceptionView id="sessionException" url="sessionException.jsp"
exceptionName="SessionException"/>
        <ExceptionView id="exception" url="error.jsp"
exceptionName="exception"/>
        </ExceptionHandler>

</action>
```

4.8.3. EMP平台异常处理机制扩展

4.8.3.1. 对ExceptionHandler的扩展

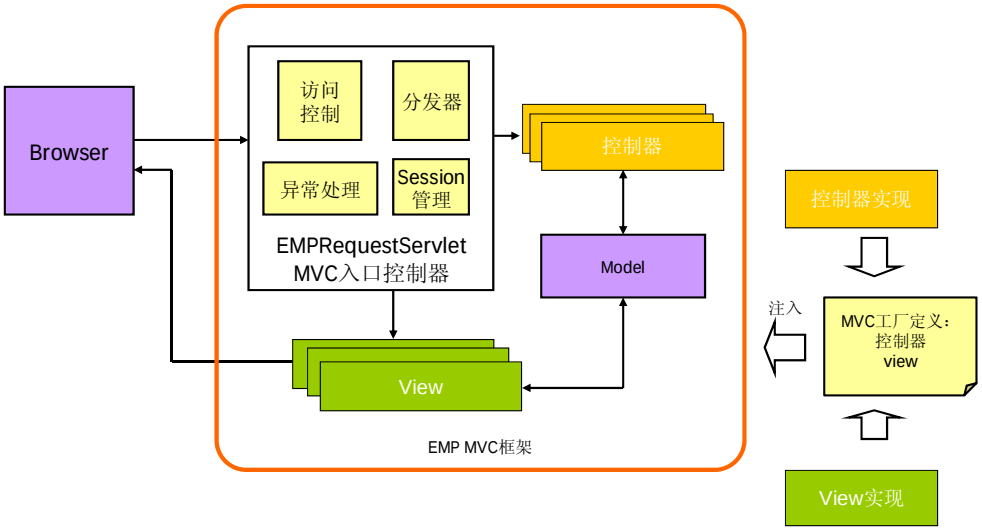
对异常的处理，EMP 提供了比较完整的机制，我们对不同的异常需要不同的展现时，通过配置 exceptionhandler 基本能满足应用的需要。当我们需要扩展 exceptionvhandler 时，只需要继承 EMP 平台的 ExceptionHandler 类，并实现类方法：getExceptionView（）。该方法的具体描述见 ExceptionHandler 对象设计章节。

5. EMP表现逻辑与MVC

5.1. MVC模型

MVC 模型是 EMP 平台提供的 Web 表现逻辑处理模型。EMP MVC 模型提供统一的 HTML 请求入口 Servlet，每一个请求都会注册到一个 EMP MVC 控制器对象，由该控制器对象进行 JSP 流程跳转控制和请求提交处理和响应。它对 Web 应用中客户浏览器端对 EMP 业务逻辑处理的请求、数据模型的更新、JSP 页面的展现以及页面间的跳转进行了抽象和简化，完全通过 XML 外部化配置文件进行定义和控制，并将页面展现与业务逻辑处理对象之间的关联进行了充分的解藕，使整个应用的结构更清晰，易于维护。

EMP MVC 模型也是基于 EMP 组件工厂框架之上设计的，通过外部化 XML 文件配置的方式将 MVC 模型实例化对象注入到运行平台完成组装运行。



MVC(Model-View-Controller，模型—视图—控制器模式)是现今十分流行的 Web 应用开发架构。它把整个应用分为了三个基本部分：模型（Model）、视图(View)和控制器(Controller)。分别对应于不同的 Java 对象，通过外部 XML 文件配置来描述三者之间的关系。

C 是控制器对象，是 EMP 平台提供的抽象类对象的继承实现，用于处理页面请求，完成数据交换和具体业务逻辑调用，并根据配置的 jsp 跳转流程逻辑，返回合适的页面对象 View。

M 是数据模型，在 EMP 平台中，Model 实质上对应于请求所要求的业务逻辑处理对象 BizLogic，控制器通过平台提供的内部接口将请求的数据更新到业务逻辑处理对象中，在业务逻辑处理完成后，根据配置要求，将业务逻辑处理对象中的数据更新到返回页面对象中。

通过配置由平台提供的控制器对象来完成页面请求和业务对象之间的数据交换。

V 是页面对象，它可以是一个提交的页面，也可以是一个返回页面，还可以根据应用需求，扩展为一个文件对象如 PDF 文件、XML 文件和图像文件对象等等。

模型—视图—控制器模式的目的是实现一种动态的程序设计，使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。除此之外此模式通过对复杂度的简化使程序结构更加直观。

MVC 模型提供了统一的入口 Servlet 来接收 Browser 的请求，并调用系统提供的奋发器将请求转交给请求 URL 所指定的控制器对象处理，控制器对象完成业务数据的交互和更新，处理完毕后，控制器将请求所指定的响应页面对象 View 转交给入口 Servlet，返回给 Browser。

5.2. EMP MVC模型实现

5.2.1. MVC入口控制器Servlet

EMPRequestServlet 继承于 Servlet 类，是 EMP MVC 模型处理的入口控制器。所有的 request 请求都将由该 Servlet 来接收并分发处理。

EMPRequestServlet 主要提供以下三个功能：

- 将该 Servlet 配置到 web.xml 文件中，配置参数在系统初始化时初始化 MVC 应用模型
- 将应用配置的请求控制器信息注册到 EMP 平台提供的 MVC 组件容器中
- 提供 MVC 模型的框架处理实现
  - 接收接收请求
  - 找到相应的控制器
  - 调用控制器的处理接口处理请求
  - 根据处理结果，分发表现页面

在 web.Xml 文件中，配置 EMPRequestServlet 对象的信息，配置内容如下表所示：

```
<servlet>
      <servlet-name>EbankmvcsServlet</servlet-name>
   <servlet-class>com.ecc.emp.web.servlet.EMPRequestServlet</serv
let-class>
      <init-param>
         <param-name>iniFile</param-name>
```

```
   <param-value>WEB-INF/bizs/Ebankbizs/settings.xml</param-value>
      </init-param>
      <init-param>
         <param-name>enableInitialize</param-name>
         <param-value>true</param-value>
      </init-param>
      <init-param>
         <param-name>password</param-name>
         <param-value>eccemp</param-value>
      </init-param>
      <init-param>
         <param-name>factoryName</param-name>
         <param-value>Ebankbizs</param-value>
      </init-param>
      <init-param>
         <param-name>rootContextName</param-name>
         <param-value>rootCtx</param-value>
      </init-param>
      <init-param>
         <param-name>servletContextFile</param-name>
   <param-value>WEB-INF/mvcs/Ebankmvcs/empServletContext.xml</par
am-value>
      </init-param>
      <init-param>
         <param-name>jspRootPath</param-name>
         <param-value>WEB-INF/mvcs/Ebankmvcs/</param-value>
      </init-param>
      <init-param>
         <param-name>resourceFileName</param-name>

   <param-value>/WEB-INF/commons/resource.xml</param-value>
      </init-param>
      <init-param>
         <param-name>contentDivId</param-name>
         <param-value>area_content</param-value>
      </init-param>
      <init-param>
         <param-name>logType</param-name>
         <param-value>log4j</param-value>
      </init-param>
      <init-param>
         <param-name>logSettingFile</param-name>
         <param-value>WEB-INF/commons/logging.xml</param-value>
```

```
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

Servlet 通过 Web 初始化时来装载这些参数进行作为 EMP 平台的初始化入口，通过 Servlet 的装载将装载 EMP 应用的组件工厂对象和一些公共配置对象，如日志处理、资源装载和 jsp 根路径指定等等。

参数名	参数说明
iniFile	业务逻辑处理容器配置入口文件，该文件中配置了业务逻辑处理容器的基础参数：如文件路径、类映射信息等等。
enableInitialize	是否进行业务逻辑处理容器的初始化
factoryName	业务逻辑组件工厂名称
rootContextName	业务逻辑处理容器的资源根节点名称
servletContextFile	MVC 模型的基础配置文件（后续描述）
resourceFileName	资源配置文件，主要是国际化信息配置资源文件
contentDivId	在 Web 页面布局中的工作区的 DivId，在 EMP MVC 中采用 Div 进行页面分层布局。
logType	日志类型
logSettingFile	日志配置文件名称

EMP 的 Servlet 装载参数可以通过 IDE 工具在加载 EMP 应用支持时自动进行参数设置，不需要用户手工处理。

5.2.2. MVC控制器

MVC 控制器对象是 MVC 模型中的核心，主要功能是实现某个具体业务请求的处理。它主要由以下几个接口功能构成：

- 数据模型处理：完成请求数据与业务逻辑对象之间的数据交换和更新。
- 业务逻辑处理：完成具体的业务逻辑的调用处理。
- 表现逻辑处理：完成请求的页面跳转流程处理。

- 异常的处理：提供异常处理的接口。

EMP 提供了控制器的抽象父类 AbstractController 类，该类主要提供了如下的接口方法：

- **处理请求**

public ModelAndView doRequest(HttpServletRequest request, HttpServletResponse response)throws Exception;

- **处理在业务逻辑处理过程中出现的异常**

public ModelAndView handleException(HttpServletRequest request, HttpServletResponse response, Exception e);

- **结束请求处理**

public void endRequest(ModelAndView mv, HttpServletRequest request, long timeUsed);

- **是否自己处理文件上传，如果返回 false,EMP 平台将根据配置的文件上传处理器处理上传文件**

public boolean isSelfProcessMultiPart();

- **启动监控信息采集**

public void startMonitor();

- **停止监控信息采集**

public void stopMonitor();

- **监控状态的获取**

public boolean getMonitorState();

- **取访问信息，用于处理监控信息的采集，实现对具体业务请求的监控信息采集**

public AccessInfo getAccessInfo();

EMP 平台提供了该抽象类的不同实现，分别用于处理不同类型的 Web 请求。当需要新的请求类型处理时，我们可以扩展该抽象类来实现新的请求处理流程。

5.2.3. Model和View

业务逻辑控制器在执行完具体的业务处理逻辑后的返回结果，结果包括：模型，和返回的表现逻辑对象用于处理表现。EMP 提供了一个类：ModelAndView，该类将 Map 对象和一个 View 对象封装在一起，用于控制器处理结束后返回给 Servlet 的一个对象。ModelAndView 中包含一个 Map 对象和一个 View 对象。

Map 对象就是 EMP MVC 提供的 Model，可以动态存放所需要的数据对象。View 对象就是返回给请求端的展现对象。

View 用于表现业务逻辑处理结果的表现页，可以是 JSP，或其他页面输出如图表、Excel, PDF。EMP 提供了 View 的接口类：View。该接口定义了一个 Render 方法，用于从数据模型中获取数据并完成展现所需要的处理，该方法的定义如下：

```
public void render(Map model, HttpServletRequest request, HttpServletResponse response,String rootPath)
```

参数 model 实际上就是 ModelAndView 对象中封装的数据模型对象。RootPath 参数就是应用所设定的 jsp 的根路径信息。

5.2.4. MVC组装和实现

5.2.4.1. MVC模型组装

EMP MVC 采用配置文件方式进行注入。EMPRequestServlet 对象的初始化时指定的 servletContextFile 文件中可以定义 MVC 的配置信息和控制器内容，通过 EMP 组件工厂注入控制器定义。同时也可以对独立的 MVC 定义文件进行注入。（独立的 MVC 定义文件存放在 servletContextFile 所在目录下即可）

5.2.4.2. 配置文件说明

EMP MVC 的配置文件由两个文件组成，一个是公共的配置文件 empServletContext.Xml 文件，用于配置一些公共组件和一些全局性控制器定义，如菜单定义等等。

一个是独立的 MVC 控制器定义，可以有若干个 MVC 控制器定义，这些控制器可能是具体的表现逻辑操作处理的定义。

5.2.4.2.1. empServletContext.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<servletContext>
    <sessionManager class="com.ecc.emp.session.EMPSessionManager"/>
    <localeResolver class="com.ecc.emp.web.servlet.CookieLocaleResolver"/>
```

```
    <initializer class="com.ecc.ebankdemo.InitContext"/>
    <action id="taskTree"
class="com.ecc.emp.web.taskInfo.TaskInfoXMLController">
        <taskInfoProvider
taskInfoFile="WEB-INF/mvcs/EBankmvcs/taskInfo.tsk"
                          class="com.ecc.emp.web.taskInfo.EMPTaskInfoPro
vider"/>
    </action>
</servletContext>
```

如上所示，empServletContext.xml 由两部分组成，除了 action 以外的项目就是 Servlet 公用组件。这些组件主要有如下几种：

- **sessionManager 会话管理器**

EMP 的会话管理器，拥有两种可替换的透明的会话管理模式：HttpSession 管理和 EMPSession 管理。在接入渠道支持 Http 协议的情况下，可以使用普通的 HttpSession 对会话进行管理；而对于特殊的接入渠道（如 TCP/IP），EMP 也提供了自己的会话管理实现，使用 URL rewrite 机制跟踪会话。

会话的超时管理完全由 EMP 平台自主进行，无需使用者干预。此外 EMP 会话管理还支持通过数据库实现的负载均衡。

HttpSessionManager 的定义方法：

<sessionManager class="com.ecc.emp.session.HTTPSessionManager"/>

EMPSessionManager 的定义方法：

<sessionManager class="com.ecc.emp.session.EMPSessionManager"/>

- **localeResolver 多语言处理器**

用于多语言 Web 应用的获取客户端语言环境的处理器，默认拥有两种实现方式：CookieLocaleResolver 和 SessionLocaleResolver。两者的区别在于 CookieLocaleResolver 能够记录用户的语言选择，并用于同一用户的下次登录；而 SessionLocaleResolver 仅在当前会话中保存用户的语言选择。

用户在提交请求时，如果传递名为 locale 的参数，localeResolver 会将其作为用户的语言选择，根据处理器实现类的不同，将其保存在 Cookie 或 Session 中。

CookieLocaleResolver 的定义方法：

<localeResolver class="com.ecc.emp.web.servlet.CookieLocaleResolver"/>

SessionLocaleResolver 的定义方法：

<localeResolver class="com.ecc.emp.web.servlet.SessionLocaleResolver"/>

- **initializer 初始化处理器**

用户自己实现的初始化处理器。在完成 ServletContext 所有的系统初始化后，如果配置了此处理器，则会调用此处理器的 initializer 方法，完成用户定义的初始化操作。实现此处理器，只需要实现 com.ecc.emp.web.servlet.Initializer 接口即可。

该接口需要实现的方法：

public void initialize(EMPFlowComponentFactory factory)throws Exception;

定义方法示例：

<initializer class="com.ecc.ebankdemo.InitContext"/>

- **multiPartResolver 文件上传处理器**

定义方法示例：

<multiPartResolver class="com.ecc.emp.web.multipart.implement.EMPMultipartResolver"/>

- **accessManager 访问控制器**

用于控制用户访问权限、及进行并发控制的控制器。当用户提交对某 action 的请求时，会由访问控制器检查该用户是否具有该 action 的权限，进而允许或拒绝用户对该 action 的访问。对于需要进行并发控制的 action，由并发访问控制器根据当前并发数以及最近一笔交易的访问时间来动态控制最大并发数量。

- **exceptionHandler 异常处理器**

用于处理在应用中出现的异常的处理器。若在执行中遇到异常，则根据异常名称找到对应的错误页面并返回。可以配置全局的异常处理器，也可以针对某一 controller 单独配置异常处理器。

异常处理器定义方法示例：

```
        <exceptionHandler>
            <exceptionView id="SQLException" url="sqlError.jsp"/>
            <exceptionView id="exception" url="error.jsp"/>
        </exceptionHandler>
```

5.2.4.2.2. 单独的MVCAction配置文件

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

<actionDefines>
    <action id="signOn" type="session"
sessionContextName="PersonalEBankSessionCtx"
```

```
                    sessionIdFieldName="sessionID" checkSession="false">
        <jspView id="index.jsp" url="index.jsp"/>
        <modelUpdater viewId="index.jsp">
            <kColl>
                <field id="userid"/>
                <field id="password"/>
            </kColl>
        </modelUpdater>
        <refFlow flowId="customerManager" op="signOn">
            <transition dest="index.jsp" condition="$retValue='failed'"/>
            <transition dest="mainLayout.jsp"
condition="$retValue='success'"/>
        </refFlow>
    </action>
    <action id="relatedAccount" type="normal">
        <jspView id="showCustomerInfo.jsp" url="showCustomerInfo.jsp"/>
    </action>
</actionDefines>
```

以上是名为 signOn.xml 的配置文件示例。在一个 MVC action 配置文件中也可以定义多个 action，通常把紧密相关的几个 action 放在同一个文件中。

action 通常由 View 和 Flow 组成。以上面 id="signOn"的 action 为例，该 action 代表了一个 JSP 流程处理，jspView 指向请求 JSP 页面，refFlow 指向该请求所调用的业务逻辑，由若干个 transition 根据不同条件转向不同响应页面。

action 必需的属性为 id 和 type。type 主要有 normal，session，endSession 等类型，它指定了该 action 对应的 controller 类型。关于 controller 在下一节有详细介绍。另外可为 action 指定 checkSession 属性，规定在执行 action 时是否检查会话超时。

根据实际需要或 controller 类型的不同，一个 action 可以只有一个 jspView，或没有 jspView 而直接执行一个 flow，或是包含其他类型的 View。

- **jspView 相关属性：**

id	JSP View 的名称
url	JSP 页面的路径

- **refFlow 相关属性/子节点：**

flowId	业务逻辑构件(EMPBusinessLogic)的名称
op	业务逻辑构件中的交易(operation)名称
transition	流转定义子节点

- **transition 相关属性：**

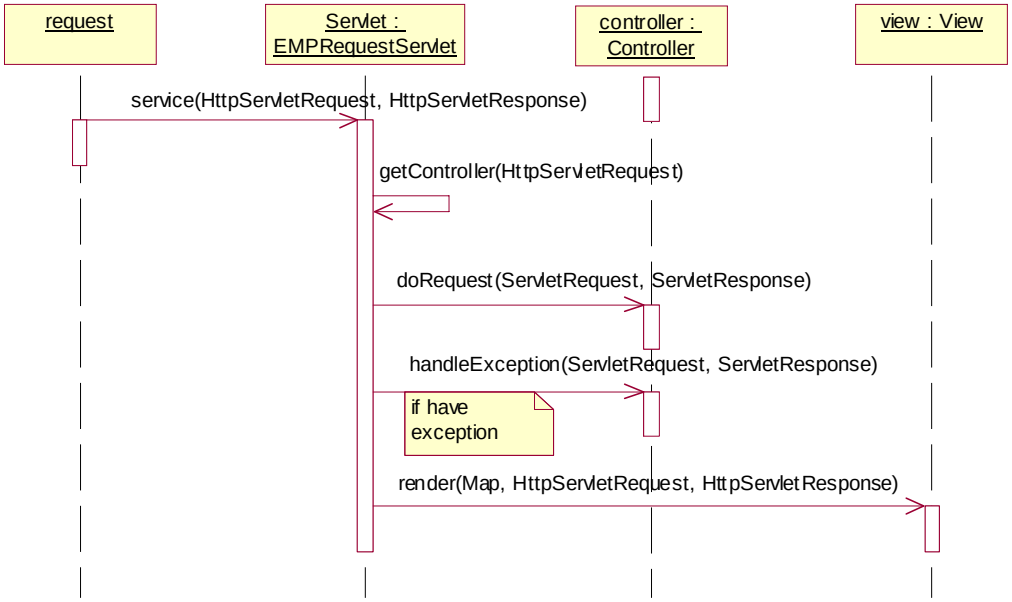
condition	流转条件，$retValue 代表业务逻辑流程返回值

dest	响应 JSP 页面的路径

modelUpdater 为数据模型更新器，它定义了从页面提交到业务逻辑的数据项（以一个 kColl 的形式定义）。如果省略 modelUpdater，则提交业务逻辑所定义的输入数据项。

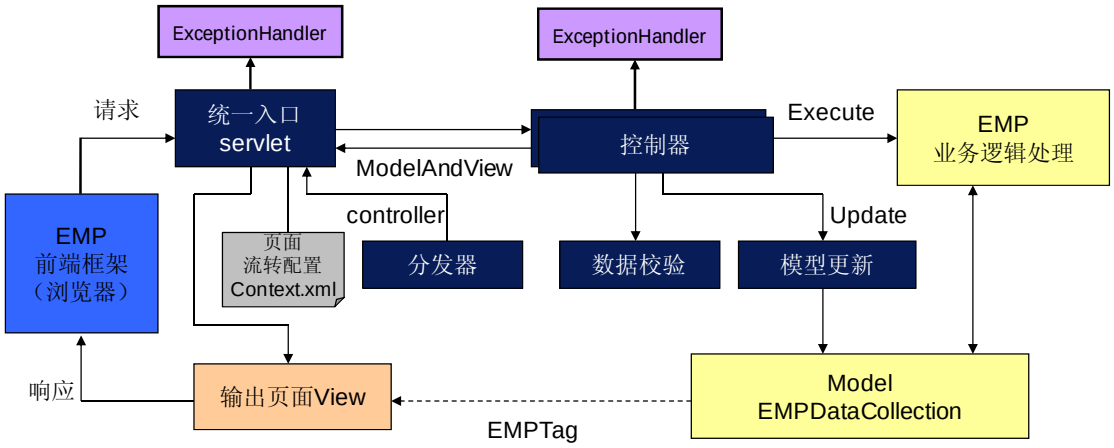
5.2.4.3. MVC运行流程

见下图，是 MVC 运行流程的示意图：



- **第一步：**一个 request 请求到达 EMPRequestServlet，EMPRequestServlet 对象调用 service 方法进行处理。
- **第二步：**根据 request 请求的 URI 字段，找到 request 请求的 ControllerID，也就是 actionName.do，根据 actionName 找到已注册的 Controller 对象。
- **第三步：**Controller 对象调用接口方法：doRequest 方法进行请求处理。
- **第四步：**如果出现异常，则进入到 Controller 对象的 handleException 方法进行异常处理。
- **第五步：**调用 action 流程中定义的跳转页面对象 View（可能是一个页面，也可能是图表或其他对象），调用 View 接口方法 render 进行输出展现处理，并将控制权交给 EMPRequestServlet 对象进行输出处理。

5.2.4.4. EMP MVC模型实现总结



如上图，我们对 EMP 平台所提供的 MVC 机制做一个总结：

- EMP MVC 是一种基于 MVC 设计模式的松藕合的 Web 处理框架。提供统一的 Servlet 请求入口，通过请求的 URI 定位到指定的控制器进行请求处理，最终通过表现对象 View 对返回进行展现处理。
- 控制器是 MVC 的核心，每一个控制器对应一段配置文件，在配置文件中定义了页面的跳转流程，以及页面所需要访问的后台业务组件的名称，控制器负责数据校验、页面跳转、异常处理和数据交换等工作，最终执行完毕，返回给 Servlet 对象一个封装好了的 ModelAndView 对象，在里面封装了完整的请求处理数据和展现 View 对象。
- 控制器通过 EMPDataCollection 对象进行数据管理。EMP MVC 提供了模型更新器对象 ModelUpdater 进行复杂的数据交换管理。在下一节中将介绍 EMP 所提供的 ModelUpdater 对象。

5.2.5. MVC的数据交换

MVC 模型更新器用于 EMP 中的 MVC 实现模型中，将 form 的请求数据更新到 EMP 业务逻辑处理器的数据模型定义中；以及在业务逻辑处理完成后，将业务逻辑处理器的数据如何更新到页面表现对象中去。在用户请求提交上来之后，controller 会把 form 的请求数据保存到表现层的数据模型（Context）中，响应页面也通过该 Context 获取数据。因此这里的模型更新器主要指负责表现层 Context 和业务逻辑 Context 进行数据交换的组件。

5.2.5.1. 模型更新器接口

EMP 平台提供了模型更新器的接口类定义，由两个 interface 组成，分别处理由表现层到业务逻辑的输入数据，和由业务逻辑到表现层的输出数据的数据交换（更新）。

```
interface InputModelUpdater
public void updateModel(Object requestData, Context opContext,
                        KeyedCollection input, Map resources) throws EMPException;
```

将表现层的数据更新到业务逻辑的数据模型 Context 中。requestData 为表现层请求数据，可以为任何类型的对象。opContext 为业务逻辑的数据模型。Input 为业务逻辑构件定义的输入项目，resources 保存其它相关资源（如数据类型定义等）。

```
interface OutputModelUpdater
public void updateModel(Object responseData, Context opContext,
                        KeyedCollection output, EMPFlowExecuteResult result, Map resources)
                        throws EMPException;
```

将业务逻辑的数据写回表现层数据模型。responseData 为表现层返回数据，可以为任何类型的对象。opContext 为业务逻辑的数据模型。output 为业务逻辑构件定义的输出项目，result 为业务逻辑执行结果封装对象（包含返回值、异常等），resources 保存其它相关资源（如数据类型定义等）。

5.2.5.2. EMP提供的缺省模型更新器

EMP 提供了简单的模型更新器实现：

com.ecc.emp.web.modelupdate.HttpInputModelUpdater

和

com.ecc.emp.web.modelupdate.HttpOutputModelUpdater

用于处理基本的 Web 表现层数据模型 Context 和业务逻辑 Context 的数据交换。因此这两个类的 requestData 和 responseData 参数实际上均为 Context 类型。两个 ModelUpdater 的默认更新规则是同名数据更新，即根据 input 的定义，将需要输入的数据按照名称从表现层 Context 取出并更新到业务逻辑 Context 中，或根据 output 的定义，将需要输出的数据按照名称从业务逻辑 Context 取出并更新到表现层 Context 中。如果需要有特殊的更新规则，则

需要在MVC Action中进行显式定义。这两个缺省ModelUpdater可以实现的特殊更新规则有：

- 在表现层和业务逻辑层的数据名称不同的情况下，可以定义映射
- 指定某个数据从 Session 中获取，或将某个数据更新回 Session
- 进行数据类型的校验和转换

以下是一个显式定义模型更新器的例子：

```
<action id="signOn" type="session" >
    <jspView id="signOn.jsp" url="signOn.jsp"/>
    <refFlow flowId="signOn" op="signOn">
        <transition dest="signOnRes.jsp"/>
        <inputModelUpdater>
            <updateList>
                <updateRule fromId="myid" toId="id" location="request" />
                <updateRule fromId="myuserinfo" toId="userinfo" location="request" />
            </updateList>
        </inputModelUpdater>
        <outputModelUpdater updateSessionCondition="$retValue='0'">
            <updateList >
                <updateRule fromId="temp" toId="session_Id" location="session" />
            </updateList>
        </outputModelUpdater>
    </refFlow>
</action>
```

两个 ModelUpdater 均定义在 refFlow（实现类：FlowInvoker）标签下，每个 ModelUpdater 下各有一个 updateList，内含一个或多个 updateRule。这些 updateRule 所涉及到的数据应该是业务逻辑的 input/output 定义的数据的子集。通过 fromId 和 toId 指定表现层和业务逻辑层不同名称数据的映射关系，通过 location 指定更新数据的来源或目标（request 或 session）。如果某个 updateRule 项目的 fromId 等于 toId，并且 location 为 request，则该项目可以省略。另外在 outputModelUpdater 上还可以定义 updateSessionCondition 属性，用于指定在何种情况下才允许更新数据到 session 中。如果不满足该表达式的条件，则 location=session 的 updateRule 失效，数据只会被更新到表现层 Context 中。如果不定义该条件，则任何情况下都会将数据更新到 session。

5.2.6. EMP提供的控制器类型

EMP 平台提供了 MVC 控制器接口和基于该接口的各种控制器实现，帮助管理各种 HTTP 请求处理和 JSP 页面的典型流程应用。这些控制器可满足大部分情况下的页面逻辑控

制需求。

5.2.6.1. normal：普通请求处理器

normal 处理器的实现类为 com.ecc.emp.web.servlet.mvc.EMPRequestController，它提供了最一般的 JSP 流程处理，根据 HTTP 请求类型返回 JSP 页面或调用业务逻辑处理。

典型的使用 normal 处理器的 action 包括一个请求页面(jspView)和一个业务逻辑处理流程引用(refFlow)。

```
    <action id="addAccount" type="normal" checkSession="true">
        <jspView id="addAccount.jsp" url="addAccount.jsp"/>
        <refFlow flowId="accountManager" op="addAccount">
            <transition dest="showCustomerInfo.jsp"
condition="$retValue='success'"/>
            <transition dest="addAccount.jsp"
condition="$retValue='failed'"/>
        </refFlow>
    </action>
```

normal 处理器的工作原理如下：

当用户发送 GET 请求时，控制器会返回 jspView 定义的请求页面，否则同 POST 提交方式。

当用户发送 POST 请求提交表单时，控制器会执行 refFlow 所指向的业务逻辑处理流程，随后根据业务逻辑的返回值转向不同的返回页面。

以上是一般的处理流程，假如未定义 jspView，则不管用户发送的是什么请求，都会直接执行业务逻辑处理流程。未定义 refFlow 的场合以此类推。

普通控制器的可配置参数较多，描述如下表所示：

参数名	描述
inputView	用户的输入 View
flowDef	empFlow 定义
Map outputViews	输出 View 定义
checkSession	是否需要检查用户的 Session 是否存在
dynamicDataModel	是否采用动态数据模型

verifyPinField	是否需要检查用户的校验码
exceptionHandler	异常处理器定义
checkVerifyPin	是否需要检查用户的校验码
verifyPinField	校验码的数据域名称
verifyPinMessage	校验码错误提示信息（可以是多语言 resource 中定义的串）
useSoftKeyPin	是否使用了软键盘输入密码
softKeyPinField	软键盘数据的数据域名称

5.2.6.2. session：创建会话处理器

session 处理器的实现类为 com.ecc.emp.web.servlet.mvc.EMPSessionController，继承于 normal 处理器，提供了创建会话功能。

```
    <action id="signOn" type="session"
                    sessionIdFieldName="sessionID" checkSession="false">
        <jspView id="index.jsp" url="index.jsp"/>
        <refFlow flowId="customerManager" op="signOn">
            <transition dest="index.jsp" condition="$retValue='failed'"/>
            <transition dest="mainLayout.jsp"
condition="$retValue='success'"/>
        </refFlow>
    </action>
```

session 处理器和 normal 处理器的工作原理基本一致，但在进入业务逻辑处理流程前会创建会话及 sessionContext。

- session 处理器相关属性：

sessionIdFieldName	保存 sessionID 的数据域名（存放在 sessionContext 中）。若无此属性则不在 EMP 数据中保存 sessionID。

5.2.6.3. endSession：结束会话处理器

endSession 处理器的实现类为 com.ecc.emp.web.servlet.mvc.EMPEndSessionController，继承于 normal 处理器，提供了结束会话功能。

```
    <action id="signOff" type="endSession" checkSession="true">
        <refFlow flowId="customerManager" op="signOff">
            <transition dest="index.jsp"/>
        </refFlow>
    </action>
```

endSession 处理器和 normal 处理器的工作原理基本一致，但在结束请求（即返回页面前）会将当前会话及 sessionContext 销毁。注意，在一个同时包含 jspView 和 refFlow 的 endSession action 中，若先执行了 GET 请求，再执行 POST 请求，此时 session 已经被销毁，无法使用 session 资源。

5.2.6.4. multiAccess：多次请求处理器

multiAccess 处理器的实现类为 com.ecc.emp.web.servlet.mvc.EMPMultiRequestController，继承于 normal 处理器，它可以暂时保存私有数据，直到用户请求其他业务处理逻辑为止。

```
    <action id="getDetailChart" type="multiAccess" checkSession="true">
         <jspView id="tranDetail.jsp" url="tranDetail.jsp" type="input"/>
        <refFlow flowId="accountManager" op="tranDetail">
            <transition dest="tranDetailRes.jsp"
condition="$retValue='success'"/>
            <transition dest="tranDetail.jsp"
condition="$retValue='failed'"/>
        </refFlow>
  <Map name="outputViews">
        <chartView id="detailChart.jpg" iCollName="tranDetailCollection"
                   valueFieldName="tranAmount" chartType="PieChart3D"
                   keyFieldName="tranType" imageTitle="@detailChart"
                   class="com.ecc.emp.web.servlet.view.PieChartView"/>
        <jspView id=”tranDetailRes.jsp”/>
  </Map>
    </action>
```

multiAccess 处理器主要用于客户端多次请求同一业务对象（非 session 数据）时使用。比如：请求提交后生成数据，返回结果页面，在页面中还有图片需要用到数据来生成。其工

作原理是会将表现层的数据模型 Context 在服务器端临时保存，用于对该处理器的多次请求。

在使用 multiAccess 控制器时，如果发送 GET 请求，并且没有指定 viewId，则返回请求页面；如果指定了 viewId（如：getDetailChart.do?viewId=detailChart.jpg），则返回相应的 View；如果是 POST 请求，则执行业务逻辑流程。

- multiAccess 处理器相关属性：

viewIdFieldName	指定请求 View ID 的参数名，默认为 viewId。

如上例：我们使用了一个 chartView 生成图表，图表的数据来源为 tranDetailCollection，而这个 iColl 是 tranDetail 流程的私有数据。如果不使用 multiAccess 处理器，当 tranDetailRes.jsp 中引用这个图表时，此时流程已经结束，私有节点已经销毁，是无法正确生成图表的。在这种情况下，就需要使用 multiAccess 处理器来暂时保存 tranDetail 流程的数据模型，以便让接下来的图表请求能够使用私有数据。

5.2.6.5. wizard：多步提交处理器

wizard 处理器的实现类为 com.ecc.emp.web.servlet.mvc.EMPWizardRequestController，继承于 normal 处理器，它和 multiAccess 有些相似，用于多页面向导形式的提交，同样将表现层数据模型 Context 在服务端临时保存，因此每个向导页面提交上来的数据都会保存在该 Context 中，最后一并提交给业务逻辑流程。wizard 处理器还支持回退到上一步，在用户最终提交后才自动清理临时数据对象。

```
<action id="applyCreditcard" type="wizzard" checkSession="true">
    <jspWizzardView pageIndex="0" url="card1.jsp"/>
    <jspWizzardView pageIndex="1" url="card2.jsp"/>
    <jspWizzardView pageIndex="2" url="card3.jsp"/>
    <jspWizzardView pageIndex="3" url="card4.jsp"/>
    <jspWizzardView pageIndex="4" url="card5.jsp"/>
    <refFlow flowId="customerManager" op="applyCreditcard">
        <transition dest="applyOK.jsp" condition="$retValue='success'"/>
        <transition dest="applyFailed.jsp" condition="$retValue='failed'"/>
    </refFlow>
</action>
```

wizard 处理器包含一系列 jspWizardView，以及一个业务处理流程。

在使用 wizard 控制器的时候，不再检测 GET 和 POST 请求，而是根据页面通过 request 传上来的 command 和 pageIndex 参数进行相应的处理。pageIndex 指定了该页面的序号，需

要和 wizard action 中定义的一致；command 取值为 next、last 或 submit，分别代表下一步、上一步和提交。

- jspWizardView 相关属性：

pageIndex	向导页面序号（从 0 开始）
url	JSP 页面路径

5.2.6.6. 其他

除了上述几种基本 Controller 之外，EMP 还提供了诸如动态菜单控制器（TaskInfoController）、标签式分页控制器（EMPTabRequestController）、系统内存监视器（MonitorController）等扩展 Controller。另外 EMP 平台还提供了控制器的编程接口，用户只需要实现该接口就可以定义自己的特有的 MVC 流程控制器。

5.2.7. MVCController的扩展

MVCController 的扩展一般采用继承于 AbstractController 或者 EMPRequestController 类，并实现其中的 doRequest 方法。

public ModelAndView doRequest(HttpServletRequest request, HttpServletResponse response)方法，返回如上格式的 XML 定义字符串即可。

- 在 controller 中获得 context 的方法：

```
EMPFlowComponentFactory factory = getEMPFlowComponentFactory();
context = factory.getContextNamed(contextName);
```

- 获得 rootContext 名称的方法：

```
String rootContextName = factory.getRootContextName();
```

- 检查并获得 sessionContext 的方法：

```
Session session = this.doSessionCheck(request, response);
Context context = null;
if (session != null) {
   context = (Context) session.getAttribute(EMPConstance.ATTR_CONTEXT);
}
```

5.3. EMP表现层多语言支持

EMP 表现层多语言支持提供了基于配置化的任意多种语言表现支持。

5.3.1. EMP表现层多语言支持工作原理

EMP 表现层多语言支持提供了静态字符串多语言翻译和动态字符串多语言翻译的功能。静态字符串多语言翻译通过外部配置字符串多语言文件来提供多语言支持；动态字符串多语言翻译需要应用系统本身提供多语言资源，EMP 平台提供 EMP 多语言数据模型提供多语言支持。

5.3.1.1. 语言选择器接口

EMP 提供了 LocaleResolver 接口类来描述用户语言选择获取器，该接口类主要有两个方法：

resolveLocale(HttpServletRequest request, HttpServletResponse response, SessionManager sessionManager)	从 request 请求中获取当前语言支持的信息，得到所支持的语言种类
setLocale(HttpServletRequest request, HttpServletResponse response, Locale locale)	将得到的语言种类保存起来，以便下次访问时获得

EMP 提供了 LocaleResolver 类的两种实现：

一种为 CookieLocaleResolver 类，从用户请求中获得 locale 信息后，存放在用户 cookie 中，下次访问将支持从 cookie 中获取语言类型。

另一种 SessionLocaleResolver 类，当用户客户端不支持 cookie 时，采用会话对象来保存 locale 信息，用户在定义时可以在 Session 数据中定义一个语言字段，SessionLocaleResolver 从该字段中获取语言类型后，存放到客户的 session 数据中，下次访问将从 session 中获取语言支持。

EMP 的语言选择器需要在 MVC 的公共逻辑定义配置中注入，在 empServletContext.xml 文件中需要增加如下的配置：

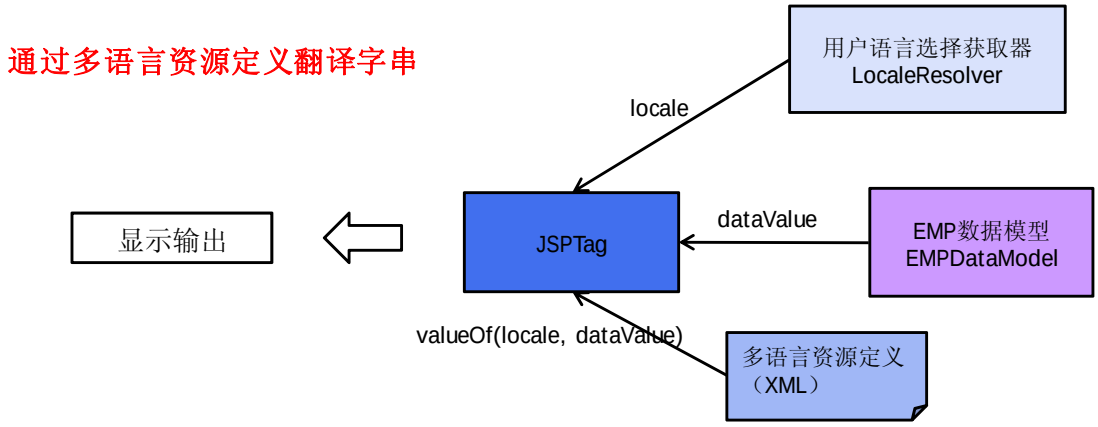
```
<localeResolver class="com.ecc.emp.web.servlet.CookieLocaleResolver"/>
```

如果是 SessionLocaleRelver 的注入，我们还必须配置一个在 context 中存放语言类型的数据域名称，SessionLocaleRelver 将通过该数据域获得应用设置的语言类型。


```
<localeResolver class="com.ecc.emp.web.servlet.SessionLocaleResolver"
localeDataName=”languageType”/>
```

5.3.1.2. 静态多语言支持设计

设计如下图所示：



EMP 提供了 JSPTag 标签来完成最终的多语言支持表现，这是 EMP 标签自动根据当前语言类型和系统可提供的语言支援进行选择的，无须用户进行干预。

LocaleResolver 提供了语言类型，EMP 数据模型提供了数据值，静态字符串的多语言配置通过外部化配置文件进行多语言资源定义。该配置文件的名称一般在 web.xml 文件的根 Context 参数中进行定义：

```
<context-param>
        <param-name>resourceFileName</param-name>
        <param-value>WEB-INF/commons/resource.xml</param-value>
</context-param>
```

该文件名为 resource.xml，位于 WEB-INF/commons 目录下。该文件内容如下所示：

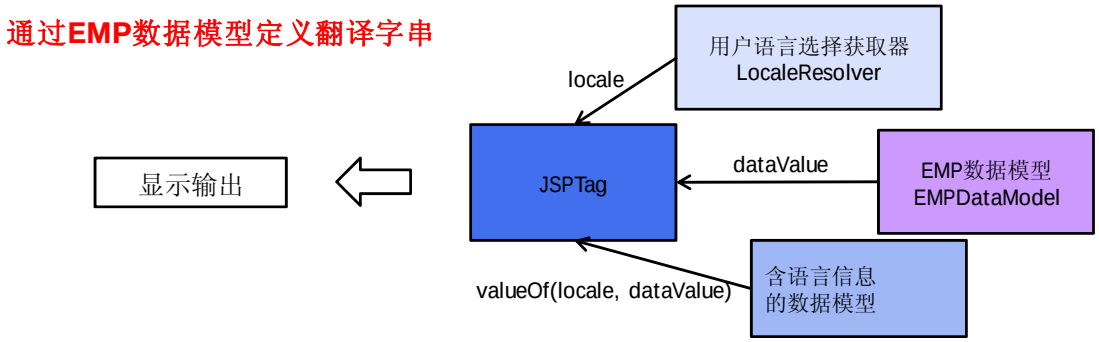
```
<?xml version="1.0" encoding="UTF-8" ?>
<IDEExternResource>
    <suportLanguage>
        <language id="zh_CN" name="中文"/>
        <language id="en_US" name="English"/>
    </suportLanguage>
    <resource id="金额">
        <resourceValue id="zh_CN">金额</resourceValue>
        <resourceValue id="en_US">Amount</resourceValue>
    </resource>
</IDEExternResource>
```

suportLanguage 指名该语言资源文件支持两种语言，分别是中文和英文。

Resource 标签中的 id 是在 Jsp 文件中的静态字符的代表 id，JSpTag 将根据该 Id 和具体的语言类型，选择对应的静态字符串来显示。

为了让页面文件具有可读性，建议将 resource 的 Id 设置为“中文”。

5.3.1.3. 动态多语言支持设计



如上图所示，通过 EMP 数据模型定义翻译字串，需要 EMP 提供含语言信息的数据模型。也就是说，动态多语言支持，需要应用提供多语言资源，EMP 将根据这些多语言资源构造含语言信息的数局模型。

应用所提供的多语言资源类似于存放在数据库表中的多语言描述或存放于文件的多语言描述，如币种的多语言说明，往往是在币种表中进行定义。

EMP 提供 text 标签来支持动态多语言的选择。text 输出显示数据模型中的数据值，将自动根据 LocaleResolver 得到的用户的 Locale 选择，通过数据模型或多语言资源定义翻译数据模型中字符串值，如果指定了数据模型映射，则通过数据模型中定义的 iColl 来翻译字符串，否则通过多语言资源定义翻译。

具体使用请参考下节的多语言支持的配置说明。

5.3.2. 多语言支持的配置说明

- 使用@符号来表明使用一个多语言支持的资源 Id

```
<ctp:label name="Label1" dataName="name" text="@userName"/>
```

上面定义了一个 label 标签，该标签的显示内容将从资源文件中获得，显示内容在资源文件中的资源 id 为‘userName’。

- 使用多语言支持的数据模型来提供多语言支持

Tag 标签的定义：

```
<emp:text dataName="acctCurrency" mapCollName="currencyInfo" mapValueName="currencyCode" mapDescName="currencyName"/>
```

数据模型定义：

```
<iColl id="currencyInfo">
  <kColl>
    <field id="currencyCode"/>
    <field id="currencyName"/>
    <field id="currencyName_en_US"/>
    <field id="currencyName_zh_CN"/>
  </kColl>
</iColl>
```

如上所示，这是一个显示帐户币种信息的 Text 标签，mapDescName 表示需要显示的数据域名称，在数据模型中定义了 currencyName，当没有多语言支持要求时，将在页面上显示 currencyName 的值，当有多语言支持时，将根据用户当前的语言类型，寻找合适的字段名称的值进行显示，如上所示，如果是中文，则寻找 currencyName_zh_CN 的值，如果是英文，则寻找 currencyName_en_US 的值。这种模式需要应用本身能够提供这种多语言资源的支持，EMP 将根据多语言支持的规则将应用所提供的多语言资源进行整理形成如上的数据模型供应用使用。

5.4. 表现层框架

5.4.1. AJAX支持

EMP 中提供基于 YUI (Yahoo UI 组件)的前端 AJAX 支持，提供基本 JS 调用，实现页面的局部更新。EMP 中的 Tag 定义和菜单处理均自动实现了 AJAX 技术，不需要用户进行 AJAX 编程处理。

EMP 中提供了两个关键的 JS 接口调用方法：

- **使用指定的 sUrl 更新指定 divId 或 iFrame 的内容**

参数为：sUrl, divId, ‘’, sUrl, divId, ‘’, ...三个一组

function updateDivContent(‘’, sUrl, divId)

- **将 form 的内容提交后，将响应内容更新到 div 或 iFrame 指定的区域**

function submitTheForm(form, div)

EMP 建议采用 DIV+iFrame 方式对页面进行布局处理。iFrame 用作应用主工作区。其他部分的布局可采用 div 组件。用户可以编写自己的 JS 函数，在该函数中调用以上两个方法进行页面的区域更新。

AJAX 框架的支持是容入到 EMP MVC 框架的表现层处理的各个部分，我们将在后续的章节中逐步介绍。

5.4.2. 布局框架Layout

EMP 表现层布局采用 DIV 分区进行布局处理。EMP 提供了 Layout 布局框架，在 IDE 开发工具中也提供了可视化的布局工具，帮助用户快速完成页面布局设计。

5.4.2.1. EMP布局框架设计

EMP 中通过提供 CSS 定义的方式实现前端 HTML 页面中的<DIV>元素的任意切分功能。一个左右切分的 DIV 组成一个基本单元，这种单元可以无限嵌套，通过这种方式提供了页面布局的无限可能。

一个基本的布局框架如下所示：

```
<div class=“yui-ge”>
    <div class=“yui-ge-ul”>
            左半边内容
    </div>
    <div class=“yui-ge-ur”>
            右半边内容
    </div>
</div>
```

在左半边内容和右半边内容中都可以无限嵌套更多的 Div 标签来进行分层设计。

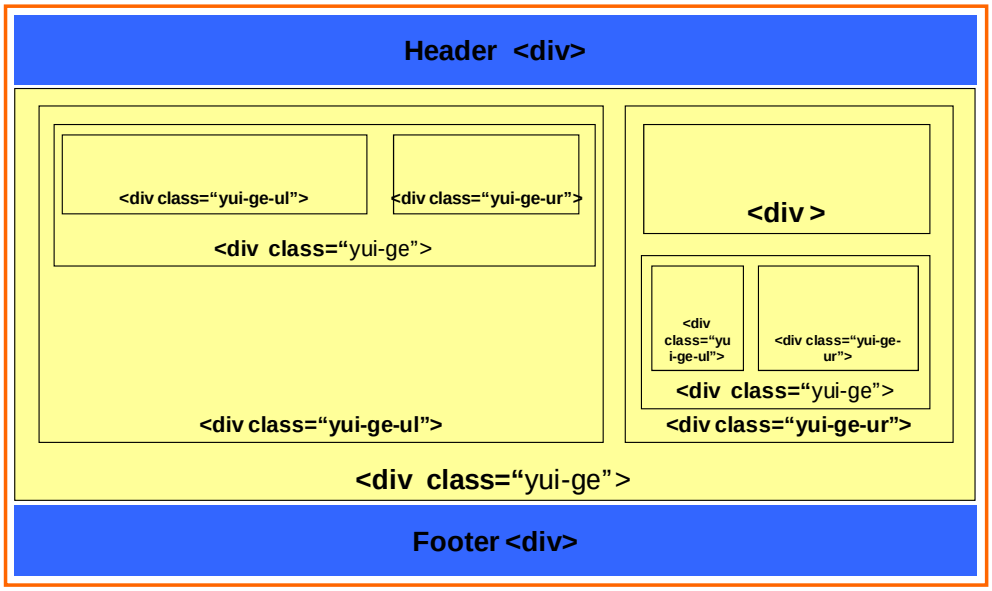
5.4.2.2. 布局框架Layout定义规则

EMP 平台中缺省提供了多种比例的 Div 切分模式，这些切分模式通过 CSS 标签的定义来进行设置。具体可用的 CSS 标签如下定义：

- yui-g 1/2 - 1/2：表示切分为父 Div 的 1/2 范围
- yui-gc 2/3 - 1/3：表示切分为父 Div 的 2/3 范围
- yui-gd 1/3 - 2/3：表示切分为父 Div 的 1/3 范围
- yui-ge 3/4 - 1/4：表示切分为父 Div 的 3/4 范围
- yui-gf 1/4 - 3/4：表示切分为父 Div 的 1/4 范围
- 其内部的左右<DIV>分别为： yui-g*-ul 和 yui-g*-ur

g*后边的*代表上面所定义的：c、d、e、f。

可以通过 IDE 工具可视化的完成布局框架的设计，在 IDE 中通过拖拽来设定 Div 的层次和大小，最终可产生如下所示的布局模型：



一个布局模型实际上产生一个*Main.jsp 文件，在布局模型中，我们可以设定每一个 Div 的用途，如菜单显示区、主工作区、标题区、页尾区和广告区等等。这些设定可绑定具体的 request 请求或 URL，在页面展现时将自动进行页面请求并更新相应区域。

如下是一个简单的布局管理器产生的布局页面内容：

```
<LINK rel="stylesheet" href='<ctp:file fileName="styles/ccb.css"/>'
type="text/css"/>
<script type="text/JavaScript" language="JavaScript">
    var empMenu;
```

```
    var empWorkBench;
    function loadContent(){
        empMenu = new EMP.widget.EMPMenu('empMenu');
            var menuBar1 = new EMP.widget.MenuBar({id:"menubar0"});
        empMenu.registMenuBar( menuBar1);
        var menuBar2 = new EMP.widget.MenuBar({id:"menubar2"});
        empMenu.registMenuBar( menuBar2);
        empMenu.registContentDiv('div0');
        empMenu.loadMenuData('<ctp:jspURL
jspFileName="taskTree.do"/>');
        }
</script>
</head>
    <body id="yahoo-com" onLoad="loadContent()">
        <div id="doc3">
            <div id="menubar0"></div>
            <div id="bd">
                <div class="yui-g">
                    <div class="yui-g-ul">
                        <div id="div0"></div>
                    </div>
                    <div class="yui-g-ur">
                        <div id="menubar2"></div>
                    </div>
                </div>
            </div>
        </div>
    </body>
```

页面装载时将执行 JS 函数：loadContent，在该函数中，将各个 Div 区进行了数据绑定和自动更新操作，如下所描述：

empMenu = new EMP.widget.EMPMenu('empMenu');

//生成一个 Menu 菜单对象

var menuBar1 = new EMP.widget.MenuBar({id:"menubar0"});

//生成一个 MenuBar 对象，并将该对象与 Div：menubar0 进行了绑定

empMenu.registMenuBar(menuBar1);

//将菜单与 Menubar 进行绑定，也就是说，一级菜单 empMenu 将在 div：menubar0 中显示

var menuBar2 = new EMP.widget.MenuBar({id:"menubar2"});

//生成一个 MenuBar 与 div：menubar2 进行绑定

empMenu.registMenuBar(menuBar2);

//将菜单与该 Menubar 进行绑定，二级菜单显示在该区域内

empMenu.registContentDiv('div0');

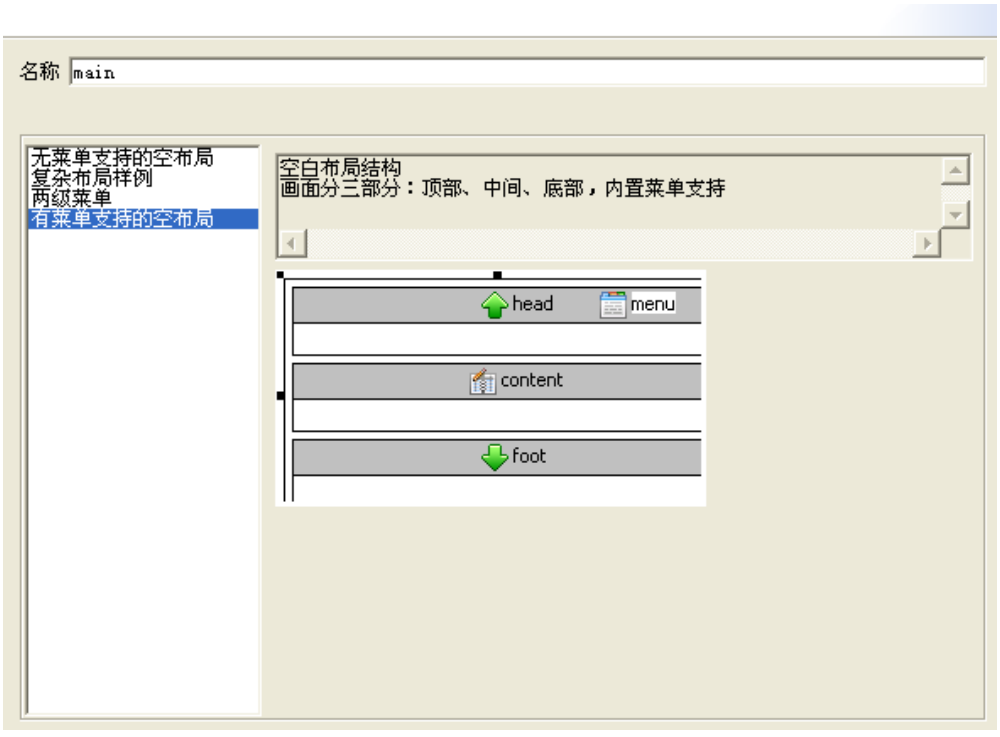
//将 div0 注册到 empMenu 的工作区，也就是说，当点击菜单时，如果是末级菜单将响应信息显示在 div0 区内。

5.4.2.3. 使用IDE来辅助布局设计

EMP IDE 提供了辅助布局工具来帮助用户完成一个 web 应用项目的布局设计。辅助布局工具提供了图形化的布局管理器，帮助用户生成页面布局的主题框架。但对于精细的实际应用，布局管理器并不能完全替代，我们需要在布局管理器的初稿之上进行手工修改，完成我们的布局设计。

首先做好我们的页面框架的规划，EMP 建议采用 DIV+IFrame 的方式进行布局设计，页面的主工作区采用 IFrame 方式，其他局部采用 DIV 方式进行设计。

可以选择布局管理器所提供的“有菜单支持的空布局”作为布局的初稿，在上面进行定制来生成我们所需要的布局样式。



这个布局模式给我们提供了最常用的布局设计，分为三级：页头、工作区和页尾。并提

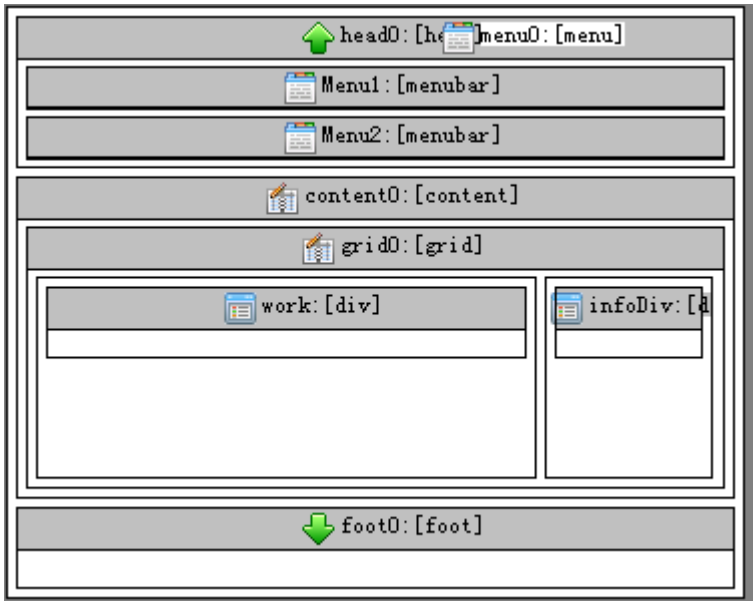
供了菜单组件。在 EMP 提供的培训教材中可找到对各个布局组件的配置使用方法。

我们一般对布局进行修改主要是针对 content 的布局进行进一步的分隔，如果需要将 content 区分为左右两个分区，我们需要使用 grid 组件来进行区分。先来认识一下布局管理器中所提供的组件的功能：

组件 Id	组件名称	功能描述	限制
Menu	菜单组件	用于定义一个菜单的总体信息，如菜单显示的一级、二级菜单与菜单栏的关联关系，菜单信息获取的操作 Id 等等	一般在选取布局模型时选择带有菜单组件的模型，不需要手工增加菜单组件
Grid	左右分栏组件	用于在一个 Div 组件中进行左右分栏布局时使用，系统默认提供了几种左右分栏比例的类型，可选择	只能在 Div 中使用
Unit	分栏单元组件	用于分栏组件之上的分栏单元表示，如果使用了分栏组件，就一定要使用 unit 组件进行分栏占位，否则分栏组件将无效	只能在分栏组件上使用
div	基本布局分区组件	一个基本布局单元，如果该单元 id 设置为 work，则表明该单元为主工作区，布局管理器将默认为其为 IFrame 对象	可放在 unit 组件之上，或 div 组件之上
Workbench	工作区组件	实质为一个 div 组件，基本不使用	不建议使用
Panel	面板组件	在页面上的小窗口，可关闭和收起	在布局中基本不建议使用
Menubar	菜单栏组件	与 Menu 组件配合使用，需要在 Menu 组件中指定一级菜单或是几级菜单的关联 Menubar，如果 Menu 有几级，则需要添加几	在有 Menu 组件的情况下使用才有效

		条 Menubar	
customContent	静态资源组件	静态资源，用于输入文字的静态内容	在布局中基本不建议使用

如下，这是通过布局管理器做出来的一个布局页面：



该页面拥有两个 Menubar，说明有两级菜单项，在 content 区，采用了一个 grid 组件，选择了“3/4 1/4”的比例进行左右分栏，在每个分栏上放置了 unit 组件，在左边的 unit 组件上放了一个 div 组件，命名为‘work’，这是我们的主工作区，在右边的 unit 组件上放了一个 div 组件，命名为‘infoDiv’，这是我们的信息显示区。

布局管理器工具会实时生成布局页面，该页面的具体内容如下所示：

```
<html>
    <head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title></title>
<ctp:yuiInclude tabType="round"/>
<!-- 缺省添加的一个yuitag，将自动加载所需要的js文件 -->
<script type="text/JavaScript" language="JavaScript">
    var empMenu;
    var empWorkBench;
<!-- 自动生成的loadContent方法，加载菜单和菜单条，定位工作区和其他定义的分区 -->
    function loadContent(){
        empMenu = new EMP.widget.EMPMenu('empMenu');
        empMenu.setDefaultMenuId('Menu');
        var menuBar1 = new EMP.widget.MenuBar({id:"Menu1"});
        empMenu.registMenuBar( menuBar1);
        var menuBar2 = new EMP.widget.MenuBar({id:"Menu2"});
```

```
        empMenu.registMenuBar( menuBar2);
        empMenu.registContentArea('work');
        empMenu.registRelativeArea('infoDiv');
        empMenu.loadMenuData('<ctp:jspURL
jspFileName="taskTree.do"/>');
        }
</script>
</head>
    <body id="yahoo-com" onLoad="loadContent()">
<!-- 根据布局管理器的定义信息，生成的div布局代码 -->
        <div id="doc3">
            <div id="Menu1"></div>
            <div id="Menu2"></div>
            <div id="bd">
                <div class="yui-ge">
                    <div class="yui-ge-ul">
                        <iframe id="work" name="work"></iframe>

                    </div>
                    <div class="yui-ge-ur">
                        <div id="infoDiv"></div>
                    </div>
                </div>
            </div>
        </div>
    </body>
</html>
```

5.4.3. Web菜单组件

5.4.3.1. 概要介绍

Web 应用的菜单通常是页面导航的重要手段。一个设计良好的菜单可以给用户带来方便快捷的体验。EMP 平台的表现层为开发者提供了自动生成和展现菜单的组件，并可以通过配置文件进行灵活配置。

通常的 Web 应用主要有如下两种菜单处理方式：①在服务端生成菜单 HTML 代码，直接输出到页面；②在服务端仅提供数据，在页面端使用 JavaScript 进行菜单展现。EMP 平台内建的菜单组件采取第二种方式，因为这种方式将内容与展现进行了分离，能够灵活满足不

同需求，同时将一部分工作转移到客户端浏览器进行，减轻服务端的压力。

5.4.3.2. 工作原理

5.4.3.2.1. 菜单数据的准备

EMP 菜单组件在服务端将数据组织成菜单定义字符串，返回给页面，由页面的 JS 脚本进行解析和展现。无论菜单数据来自静态定义还是从 EMP 数据中动态取得，只需拼装成统一格式的 XML 字符串，就可在页面端进行识别。

菜单定义的 XML 例子如下：

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

    <TaskInfo id="home" name="首页" action="home.do"/>

    <TaskGroup id="accountRel" name="关联帐户操作">

        <TaskInfo id="relatedAccount" name="查询关联帐户"
action="relatedAccount.do"/>

        <TaskInfo id="addAccount" name="添加关联帐户" action="addAccount.do">

              <advert href="advert.do"/>
              <relatedTask taskId="relatedAccount"/>

              <helpItem href="helpAccount.html" label="如何添加关联帐户？"/>

          </TaskInfo>
    </TaskGroup>

    <TaskGroup id="transferRel" name="转账操作">

        <TaskInfo id="transfer" name="本地转账" action="transfer.do"/>

        <TaskInfo id="otherBankTransfer" name="跨行转账"
action="otherTran.do"/>
    </TaskGroup>
</TaskModel>
```

其中，TaskModel 是整个菜单定义的 XML 根节点；TaskGroup 和 TaskInfo 分别代表菜单分组和菜单项。

关于菜单数据的准备，EMP 内部实现了用于静态菜单定义的 EMPTaskInfoProvider 和

TaskInfoXMLController。该 provider 从文件中读取菜单定义（仍然是 XML 格式，但有少许不同），而 controller 则将菜单定义转为 XML 字符串。页面只需发送对这个 controller 的请求，就可以获得这个 XML 字符串。

5.4.3.2.2. 菜单分组和菜单项

- 菜单分组 TaskGroup

菜单分组是具有子菜单的菜单项目，通常点击该分组会显示出下级子菜单，而不执行某个具体的功能入口。一个菜单分组可包含多个菜单项或菜单分组。如下：

```
<TaskGroup id="transferRel" name="转账操作" dftTaskId="transfer">
    <TaskInfo .../>
    ...
</TaskGroup>
```

- 菜单项 TaskInfo

菜单项是菜单中的功能入口。点击菜单项之后会向指定 URL 发送请求，并使用获得的响应内容更新工作区 DIV。如下：

```
<TaskInfo id="relatedAccount" name="查询关联帐户" action="relatedAccount.do"/>
```

5.4.3.2.3. 关联菜单项、广告和帮助

由于 EMP 菜单是与主布局页面配合使用，采用局部刷新的方式组织页面内容，因此每个菜单项除了进行主工作区的更新之外，还可同时刷新多个相关区域。EMP 内建了三种相关项目，每个菜单项可以对应多个关联菜单项或帮助项，以及一个广告信息。

- 关联菜单项 relatedTask

一般来说，菜单上的某个功能都会与其他的某些功能有所关联。为某个菜单项设置了 relatedTask 之后，当点击该菜单项时，会在关联菜单区显示这些关联菜单项的链接。一个菜单项可以对应多个关联菜单项。

```
<relatedTask taskId="relatedAccount"/>
```

- 广告 advert

若想在点击某菜单项时显示广告信息，则可为该菜单项设置 advert 项目。当点击该菜单

项时，会在广告区显示所请求 URL 的内容。

```
<advert href="advert.do"/>
```

- 帮助 helpItem

帮助信息也是一个用户体验良好的 Web 应用中经常用到的设置。可为每个菜单项设置相关的多条帮助信息，当点击该菜单项时，会在帮助区显示这些帮助信息的题目，点击其中某一项，会弹出新的窗口显示帮助信息。

```
<helpItem href="helpAccount.html" label="如何添加关联帐户？"/>
```

5.4.3.2.4. 菜单的展现

菜单的展现完全由页面端的 JavaScript 和 CSS 样式表来进行。从服务端获得菜单的 XML 字符串后，用 JavaScript 的 DOM 操作进行解析和生成相应的 HTML，并搭配 CSS 展现出多种多样的菜单。

EMP 内建的菜单页面展现封装了 Yahoo UI 的菜单，可以简单地产生普通的菜单条、下拉式菜单以及树状菜单，并内含回退功能。

5.4.3.3. 使用说明

5.4.3.3.1. 静态菜单定义的配置实例

静态的菜单配置文件 taskInfo.tsk：

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

    <TaskInfo id="home" name="首页" action="home.do"/>

    <TaskGroup id="accountRel" name="关联帐户操作">

        <TaskInfo id="relatedAccount" name="查询关联帐户"
action="relatedAccount.do"/>

        <TaskInfo id="addAccount" name="添加关联帐户" action="addAccount.do">
                <advert href="advert.do"/>
                <relatedTask taskId="relatedAccount"/>
```

```
            <helpItem href="helpAccount.html" label="如何添加关联帐户？"/>
        </TaskInfo>
    </TaskGroup>

    <TaskGroup id="transferRel" name="转账操作">

        <TaskInfo id="transfer" name="本地转账" action="transfer.do"/>

        <TaskInfo id="otherBankTransfer" name="跨行转账"
action="otherTran.do"/>
    </TaskGroup>
</TaskModel>
```

- TaskGroup 相关属性：

id(必需)	该分组的 id，用于调用及生成页面元素 id。在同一个菜单中不能出现相同的 id。
name(必需)	该分组的显示文字，支持多语言资源（用@引用资源 id）
dftTaskId	该分组的默认菜单项，即点击分组后在显示出下级菜单的同时所执行的菜单项。

- TaskInfo 相关属性：

id(必需)	该菜单项的 id，用于调用及生成页面元素 id。在同一个菜单中不能出现相同的 id。
name(必需)	该菜单项的显示文字，支持多语言资源（用@引用资源 id）
action	该菜单项所执行的 URL。
relatedTask	关联菜单项子标签（可有多个）
helpItem	帮助项目子标签（可有多个）
advert	广告信息子标签（最多 1 个）

- relatedTask 相关属性：

taskId(必需)	关联菜单项的 id。对应 taskInfo 的 id 属性

- advert 相关属性：

href (必需)	广告页面的 URL

- helpItem 相关属性：

label (必需)	帮助信息的题目
href (必需)	帮助信息的 URL

5.4.3.3.2. Controller配置实例

得到菜单定义需要在 empServletContext.xml 中定义一个 MVC Action。这一 action 不是功能入口，因此并不使用 EMP MVC 基本 Controller 类型，而是专门生成 XML 格式菜单定

义的 TaskInfoXMLController。

```
    <action id="taskTree"
class="com.ecc.emp.web.taskInfo.TaskInfoXMLController">
        <taskInfoProvider
taskInfoFile="WEB-INF/mvcs/EBankmvcs/taskInfo.tsk"
                          class="com.ecc.emp.web.taskInfo.EMPTaskInfoPro
vider"/>
    </action>
```

执行这个 action 后，就会获得菜单定义的 XML document。

5.4.3.3.3. 菜单展现实例

使用 EMP 内建的菜单展现只需在页面中加入几条 JavaScript 语句进行注册即可。如下：

```
<script type="text/JavaScript" language="JavaScript">
    var empMenu;
    function loadContent(){

        empMenu = new EMP.widget.EMPMenu('empMenu');   //生成一个EMPMenu对象

         //下面两句生成两级普通菜单，id为对应的div名，可指定平时与选中两种样式

        var menuBar1 = new
EMP.widget.MenuBar({id:"Menu1",normalClass:"menu1_off",
activeClass:"menu1_on"});
        var menuBar2 = new
EMP.widget.MenuBar({id:"Menu2",normalClass:"menu2_off",
activeClass:"menu2_on"});

         //下面两句将两级菜单注册到EMPMenu对象中

        empMenu.registMenuBar(menuBar1);
        empMenu.registMenuBar(menuBar2);

        //注册工作区和广告区

        empMenu.registContentDiv('area_content');
        empMenu.registAdvertDiv('advert');

         //从taskTree.do获得菜单内容

        empMenu.loadMenuData('<ctp:jspURL jspFileName="taskTree.do"/>');
    }
</script>
```

上面的代码在 Menu1 和 Menu2 两个 div 中生成两级普通菜单，菜单项为 taskTree.do 返回的内容。

若要生成下拉菜单，则按照如下方式定义：

```
var menuBar1 = new EMP.widget.MenuBar({id:"Menu1",style:"menuBar"});
```

若要生成树状菜单，则按照如下方式定义：

```
var menuBar1 = new EMP.widget.MenuBar({id:"Menu1",style:"tree"});
```

除了 empMenu.registContentDiv 和 empMenu.registAdvertDiv 之外，还有 registRelativeDiv 和 registHelpDiv 方法用来注册关联功能区和帮助区。

生成的菜单自动拥有页面回退功能，不需要另行配置。但有时需要在 WEB-INF 目录下放置一个名为 blank.html 的文件，内容是：

```
<html><body></body></html>
```

这是 YUI 在处理 SSL 请求的页面回退时所需要的文件。

5.4.3.4. 应用扩展

5.4.3.4.1. 调整下拉式菜单的CSS样式

由于 YUI 的下拉式菜单结构比较复杂，不能够像普通菜单条那样通过参数来设置其 CSS 样式，而是采用固定的 class 名，因此需要根据情况修改 CSS 样式文件。以下是两级下拉式菜单的 HTML 结构：（DIV-DIV-UL-LI）

```
<div class=yuimenubar>
    <div class=bd>
        <ul>
            <li class=yuimenubaritem first-of-type><a>label</a></li>
            <li class=yuimenubaritem><a>label</a></li>
            <li class=yuimenubaritem hassubmenu>
                <a class=hassubmenu>label</a>
                <em class=submenuindicator></em>
                <div class=yuimenu>
                     <div class=bd>
                          <ul>
                              <li class=yuimenuitem first-of-type><a>label</a></li>
                          </ul>
                     </div>
                </div>
             </li>
        </ul>
    </div>
</div>
```

第一级整体 DIV 的 class 为 yuimenubar，每个菜单项 LI 的 class 为 yuimenubaritem；第二级（及以后）整体 DIV 的 class 为 yuimenu，每个菜单项 LI 的 class 为 yuimenuitem。多个同级 LI 中的第一个还拥有样式 first-of-type。有子菜单的项目拥有样式 hassubmenu。被选中的项目拥有样式 selected。

5.4.3.4.2. 动态菜单的数据准备扩展

EMP 仅内建了静态菜单定义的组件，若要实现从 EMP 数据中动态生成菜单定义，可以扩展自己的 TaskInfoProvider。继承 TaskInfoProvider，实现该接口的抽象方法即可。

```
<action                                                    id="taskTree"
class="com.ecc.emp.web.taskInfo.TaskInfoXMLController">
       <taskInfoProvider
taskInfoFile="WEB-INF/mvcs/EBankmvcs/taskInfo.tsk"
                          class="com.ecc.emp.web.taskInfo.EMPTaskInfoPro
vider"/>
    </action>
```

如上为一个 TaskInfo 菜单信息处理的 Controller 的配置，EMP 提供了两个类一个为 TaskInfoXMLController 类，该类继承于 EMPRequestController 类，主要用于将菜单信息转换为标准的 xml 字符串信息返回给客户端，供客户端展现。只要提供了菜单对象是采用 EMP 平台所提供的标准的 TaskGroup+TaskInfo 方式的菜单内容，TaskInfoXMLController 控制器可以自动完成菜单信息的输出。

另一个类是 EMPTaskInfoProvider 类，该类实现了接口：TaskInfoProvider 接口。来看看 TaskInfoProvider 接口中定义的方法：

方法名	参数	说明
initializeFrom	String rootPath	初始化菜单内容，生成 TaskGroup 下的相关内容
getTaskInfos	Context sessionCtx，String userId，String parentGroupId	根据用户名和 parentGroupId，得到指定的 parentGroupId 下的所有菜单对象。如果 parentGroupId 为 null，则获取所有的 taskInfo，

		返回一个 List 对象
getTaskInfos	Context sessionCtx， String userId，String taskInfoId	根据 userId 和 taskInfoId，返回指定的 taskInfo 对象
getRelativeTasks		返回指定的 taskInfoId 的关联菜单信息对象，返回一个 List 对象
GetHelpItems		返回指定的 taskInfoId 的帮助菜单信息对象，返回一个 List 对象

EMP 所提供的 EMPTaskInfoProvider 类实现了该接口，实现的是从指定的 taskInfo.tsk 文件中去读取所有的 taskInfo 和 taskgroup 对象信息并形成 task 菜单树。

如果我们需要从数据库中去获取菜单信息或者别的什么数据源获取菜单信息，则需要实现 TaskInfoProvider 接口来提供我们自己的 TaskInfoProvider。

常规的做法是：

一、采用标准的 EMP Biz 对象从数据库中获取相应的菜单信息，并将菜单信息保存在 Biz 构造中的 context 中的数据域中。

二、新增一个 TaskInfoProvider 的实现类，如 ExtTaskInfoProvider 类。定义 TaskGroup 对象变量，用于保存该菜单信息对象的根结点对象。

三、实现 initializeFrom 方法，执行指定的 bizflow，然后从 bizFlow 中获取 KColl 数据对象，根据获取的对象生成我们需要的 TaskGroup 根对象，将该对象赋予 taskGroup 变量。

四、实现其他几个方法，getTaskInfos 和 getTaskInfo 方法必须实现，其他两个方法可根据情况选择实现。

新扩展的 taskInfoProvider 在配置上可能是如下形式：

```
<action                                                    id="taskTree"
class="com.ecc.emp.web.taskInfo.TaskInfoXMLController">
        <taskInfoProvider          bizId=”menuInfo”          opId=”getMenu”
userIdField=”userId”
                          class="com.ecc.emp.web.test.EMPExtTaskInfoProv
ider"/>
    </action>
<!-- 注意 TaskInfoXMLController 并没有改变，只是 TaskInfoProvider 的实现类和参数发
```

生了改变，其中的 EMPExtTaskProvider 是实现类名，bizId 和 opId 指向具体的 menu 信息获取的业务处理逻辑流程，userIdField 是指在需要进行权限检查时，从那个数据域中获取用户名 Id 的值 -->

<!-- 以上只是一个配置示例，可能大多数情况要生成的动态 Provider 需要这些参数的定义 -->

5.4.4. Web图表功能

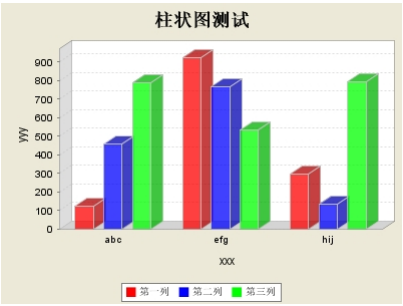
5.4.4.1. 概要介绍

在 EMP 平台的 MVC 模型中，提供了专门用于绘制展示图表的视图 chartView。使用该类视图，可以将 IndexedCollection 中以表单结构储存的数据，按照所定制的规则进行统计，然后以各种统计图表的方式加以展示。

目前 EMP 中提供了以下两种图表视图：

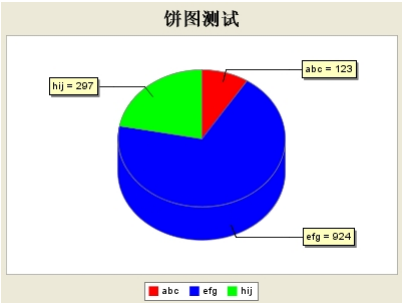
- BarChartView

柱状图表，包括：BarChart-柱状平面图，BarChart3D-柱状立体图，LineChart-折线图，StackedBarChart-叠加柱状平面图，StackedBarChart3D-叠加柱状立体图。柱状图表通常适用于展现某一组数据在特定条件下的渐进变化过程，例如统计过去一季度中，三个部门每月的盈利额度变化趋势。效果图如下：



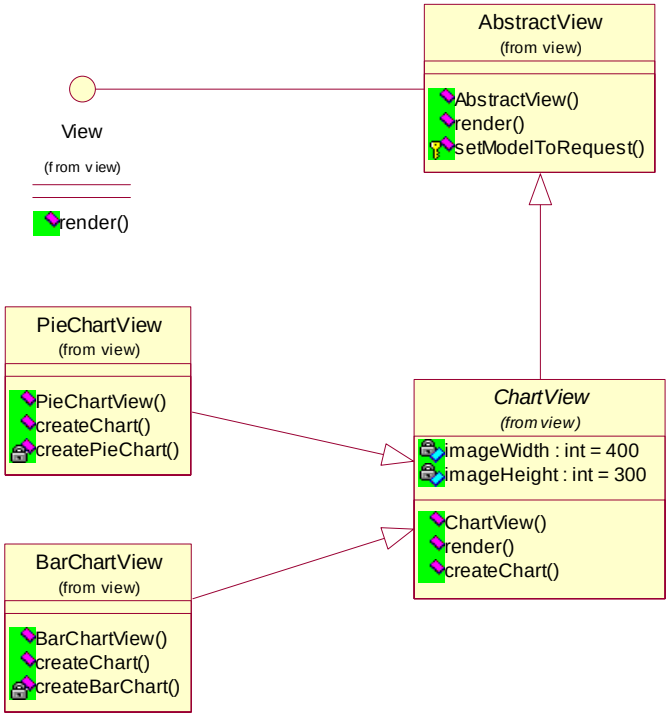
- PieChartView

饼状图表，包括：PieChart-2D 饼图，PieChart3D-3D 饼图。饼状图表通常适用于展现某一类数据的百分比例分布情况，例如统计过去一年中不同部门的盈利额度百分比例。效果图如下：



5.4.4.2. 工作原理

5.4.4.2.1. EMP图表视图类关系说明



5.4.4.2.2. EMP图表生成原理

EMP 的图表生成机制是基于 JFreeChart 提供的统计图表绘制 API 实现的。从 EMP Context 数据生成统计图表的过程可以概括为如下方流程图所示的 2 个阶段：

- 取出 EMP Context 中的数据集合 IndexedCollection，将其中的数据封装为实现 org.jfree.data.Dataset 接口类型的 JFreeChart 数据集合。其中，柱状图数据集合类型为 org.jfree.data.DefaultCategoryDataset，饼状图数据集合类型为 org.jfree.data.DefaultPieDataset。
- 调用 org.jfree.chart.ChartFactory.createXXXChart()方法，由 Datase 数据集合生成 org.jfree.chart.JFreeChart 类型的图表实例。

生成柱状图的调用方法如下：

```
JFreeChart chart = ChartFactory.createBarChart(imageTitle, //chart title
                    getImageDomain(), //domain axis label
                    getImageRange(), //range axis label
                    (DefaultCategoryDataset) dataset, //data
                    PlotOrientation.VERTICAL, //orientation
                    true, //include legend?
                    true, //tool tips?
                    false //URLs?
                );
```

生成饼状图的调用方法如下：

```
JFreeChart chart = ChartFactory.creatPieChart(imageTitle, //chart title
                    (DefaultPieDataset) dataset,//data
                    true, //include legend?
                    false, //tool tips?
                    false //URLs?
                );
```

生成的 JFreeChart 图表实例，可以调用 org.jfree.chart.ChartUtilities.writeChartAsJPEG()方法将其以 JPEG 图片格式输出到浏览器上。

5.4.4.3. 图表视图的配置实例

5.4.4.3.1. 图表视图属性

属性名	属性描述	属性类型
BarChartView		
class	对应实现类名	class
chartType	生成图形的类型	string
imageTitle	图形的标题	string
imageDomain	图形的横轴标示	string
imageRange	图形的纵轴标示	string
iCollName	生成图形的数据集合名称	string
categoryFieldName	分类的数值域名称	string
seriesNames	以’;’分割的所有序列值域名称	string
seriesFields	以’;’分割的所有序列名称	string
PieChartView		
class	对应实现类名	class
chartType	生成图形的类型	string
imageTitle	图形的标题	string
imageDomain	图形的横轴标示	string
imageRange	图形的纵轴标示	string
iCollName	生成图形的数据集合名称	string
keyFieldName	饼图中的数值代表的名称域	string
valueFieldName	饼图中的数值域名称	string

5.4.4.3.2. 图表视图的使用

使用图表视图 chartView 时，首先需要在相关的 MVC 配置文件中对视图属性进行定义，然后在 JSP 页面中通过<ctp:img>标签进行引用。

- **图表视图的定义方式**

图表视图可以被添加到已存在的 MVC 控制器模型定义中，也可以定义为独立的 MVC 模型。需要注意的是，如果图表视图处理的数据集合是属于某一业务逻辑构件的私有数据，则需要将该图表视图所处的 MVC 模型中的入口 action 的 type 属性设置为 multiAccess。

```
<?xml version="1.0" encoding="GB18030" ?>
<actionDefines>
```

```
    <action id="chartMVC" type="multiAccess" checkSession="false">
        <refFlow flowId="chartBiz" op="chartFlow">
            <transition dest="chartJSP.jsp" />
        </refFlow>
        <chartView id="barChart.jpg"
            class="com.ecc.emp.web.servlet.view.BarChartView"
            chartType="BarChart3D"
            imageTitle="柱状图测试"
            imageDomain="x轴"
            imageRange="y轴"
            iCollName="chartColl"
            categoryFieldName="name"
            seriesNames="第一列;第二列;第三列;"
            seriesFields="value01;value02;value03;" />
        <chartView id="pieChart.jpg"
            class="com.ecc.emp.web.servlet.view.PieChartView"
            chartType="PieChart3D"
            imageTitle="饼图测试"
            iCollName="chartColl"
            keyFieldName="name"
            valueFieldName="value01" />
    </action>
</actionDefines>
```

在上面的例子中，chartMVC 这个 MVC 模型中包含了 chartBiz.chartFlow 这个业务流程，以及结果页面 chartJSP.jsp。模型中添加的 2 个图表视图分别为 barChart.jpg 和 pieChart.jpg，而它们与整个 MVC 模型的执行流程无关，即只对 2 个图表视图的属性作了定义，以留待 jsp 页面进行引用和展示。

- **图表视图的引用方式**

定义在 MVC 模型中的图表视图可以视为一个动态生成的图片文件。在 jsp 页面中，通过<ctp:img>标签，同样能够以引用一般静态图片的方式对其进行引用，引用格式如下：

```
<ctp:img alt="pieChartImg" src="chartMVC.do?viewId=pieChart.jpg"/>
<ctp:img alt="barChartImg" src="chartMVC.do?viewId=barChart.jpg"/>
```

另外，直接在浏览器地址栏中输入 src 属性的参数值，也可在浏览器中得到相应的图表。

5.5. 表示层组件Taglib

在 EMP 的 MVC 模型实现中，把 Context 以 attribute 的方式保存到 httpServletRequest 中，这样在 jsp 页面中就可以访问到 EMP 的业务逻辑数据模型，从而在页面上显示经过业务逻辑处理后的数据。但 EMP 平台并不推荐直接在 jsp 页面中用 Java 代码取得 Context 及数据模型，因为这样会造成页面复杂度及与后台业务逻辑的耦合度提升，不利于页面开发和维护。为解决页面与数据模型的交互问题，EMP 平台提供了一套基于 JSP Taglib 的标签实现，用于在页面段对数据模型进行多种形式（输入框、下拉框、列表等）的展现。通过 JSP Taglib 机制，在页面上自动生成相应的 HTML 内容和 JavaScript 脚本，实现对数据模型的访问。

由于 Context 以及数据模型 KeyedCollection 和 IndexedCollection 都按照 Java 的 collection 标准实现（Context 和 KeyedCollection 实现了 Map 接口，而 IndexedCollection 实现了 List 接口），因此也可以直接在 JSP 页面上使用 JSP 标准标记库（JSTL）标签访问数据模型。

此外，对于页面上的一些常用组件或功能，如日期输入、联动下拉框、标签式浏览等，EMP 也提供了相应的 Taglib 封装，使得开发人员不必书写复杂的脚本代码，即可实现丰富多彩的页面展现。部分功能是基于 Yahoo UI 开发的，因此在运行时需要引入 Yahoo UI 的相关文件支持。

5.5.1. JSTL标签支持

使用 JSP 标准标记库（JSTL）访问数据模型，主要用到了 JSTL 中的 c:out、c:set 等标签访问单个数据，及 c:forEach 等标签访问集合数据。

- 直接输出 Context 中定义的数据 dataName 的值：

```
<c:out value="${context ['dataName']}"/>
```

- 按照 EMP 的数据类型定义进行转换后输出：

```
<c:set var="tb" value="${context['timeBalance']}" scope="request"/>
<emp:text dataName="tb" dataTypeName="Currency"/>
```

其中 emp:text 为 EMP 提供的输出数据取值的标签封装。它可以输出当前 Context 中的单个数据域，或 request 中的 attribute 值。因此这句也可以直接写成

```
<emp:text dataName="timeBalance" dataTypeName="Currency"/>
```

- 访问 Context 中的 KeyedCollection 中的数据

```
<c:out value="${context ['kCollName.dataName']}"/>
```

- 访问 Context 中的 IndexedCollection 中的数据

```
<table>
    <c:forEach var="acctItem" items="${context['balanceInfos']}" varStatus="status">
<c:choose>
<c:when test="${status.count % 2 == 0}“><tr class="tableLine1"></c:when>
<c:otherwise><tr class="tableLine2“></c:otherwise>
</c:choose>
<td align="center“><c:out value="${acctItem['month']}"/></td>
<td align="right"><c:out value="${acctItem['currentBalance']}"/></td>
<td align="right">
<c:set var="inCome" value="${acctItem['inCome']}" scope="request"/>
<ctp:text dataName="inCome" dataTypeName="Currency"/>
</td>
</tr>
    </c:forEach>
</table>
```

上例用一个 c:forEach 从名为 balanceInfos 的 iColl 里依次取得每个 kColl，命名为 acctItem，以表格形式输出它们的 3 个字段值（month, currentBalance, inCome）。c:choose 用于按照循环次数的奇偶性生成具有不同样式的行。

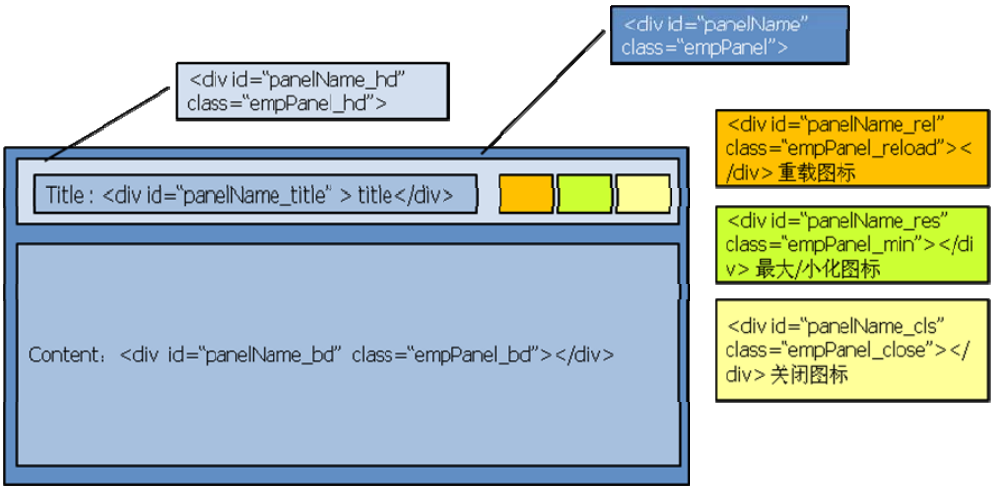
5.5.2. EMP Panel

5.5.2.1. 介绍及原理

EMP Panel 又被称为 EMP 展示区域，用于处理页面中的局部展示。它的用途多种多样，可以仅仅展示几行文字，也可以将来自其它域的网页内容嵌入当前页面。用户可以对各个区域进行收放、关闭、刷新等操作，Panel 本身也可以通过 JavaScript 定时器实现自动加载、定时刷新。普通展示内容均为 DIV，如果指定区域的来源为其他服务器域，则为了保证展示的效果及安全性，会自动转变为嵌入式框架（iFrame）方式进行处理，并通过 CSS 样式进行统一风格展现。

5.5.2.2. 表现形式

以下是一个 EMP Panel 的表现结构示意图：



从图中可以看到，一个 Panel 包含了标题区域 Head 和内容区域 Content 两部分，其中 Head 中又包含了标题 Title 和重载、最大/最小化及关闭图标。每一部分实际都是具有特定样式名的 DIV，可以方便地通过 CSS 定义 Panel 各部分的样式。

5.5.2.3. 参数定义

参数	说明
dataSrc	数据源 URL，使用此 URL 的内容填充此展示区域，如果是绝对路径（http…打头）时，将使用 iFrame 展示内容
title	标题字串，用于显示区域的标题，自动从多语言串资源定义中查找
languageRegist	是否向 EMP 全端多语言处理组件注册，如果为 true 则向多语言组件注册，在用户通过它修改语言设定时，自动刷新此区域
height	指定高度（iFrame 时）
closable	是否可以被关闭，允许在标题栏上出现关闭图标，用户点击它时，关闭这个区域
resizable	是否可以最小/大化，如果可以，则在标题栏出现图标，用户点击时自动收起/展开
reloadable	是否提供刷新图标，如果有则在标题栏出现图标，用户点击时刷新内容
autoReload	是否需要自动定时刷新
reloadDelay	定时刷新延时

5.5.2.4. 使用实例

- 使用 URL 内容填充展现区域

```
<emp:panel title="Market overview" name="market" dataSrc="stockAverage.do"
reloadable="true" resizable="true"/>
```

此组件会把 stockAverage.do 的返回内容填入 Panel 中，并且可以进行手动刷新及展开、收起操作。

如果 dataSrc 属性以绝对路径打头(http://…)，则内容区域自动生成一个 IFrame 加载其它域的内容。

- 直接定义展现区域内容

```
<emp:panel title="Recent Quotes" name="Symbole">
    <table>
        <tr><th>symbol</th><th>Name</th></tr>
        <tr><td>CTRIP</td><td>CTRIP.com ltd.</td></tr>
    </table>
</emp:panel>
```

5.5.3. Tab标签页

5.5.3.1. 介绍及原理

当需要在同一页面展示大量相关内容时，往往因为可视空间的不够造成排版困难、主次混乱等问题。EMP 的 TagLib 为此提供了标签式浏览的解决方案，开发者可将要展示的内容按需要划分为若干个子页面，分别放入不同的标签页中。用户在浏览这些信息时，只需点击相应标签即可进行切换显示内容。和 EMP Panel 一样，标签页也支持直接定义内容，以及从某个 URL 载入。

标签页组件由 emp:tabView 和 emp:tab 两种标签组成，在页面上用 tabView 嵌套多个 tab，并为每个 tab 定义其数据来源，即可实现一组标签页的展现。

Tab 标签页组件是基于 YUI 的 tabview 组件开发的，因此在使用时需要引入 YUI tabview 相关的 JS 和 CSS 文件。

5.5.3.2. 参数定义

- emp:tabView 相关属性：

参数	说明
name	tabView 名称
orientation	标签样式，默认为 top，可取值 bottom，left，right

- emp:tab 相关属性：

参数	说明
name	tab 名称
label	标签上的文字
active	若为 true 则是初始显示的分页
dataSrc	内容来源 URL
cacheData	是否缓冲内容，若为 false 则每次都会重新向内容来源发送请求

5.5.3.3. 使用实例

```
<emp:tabView name="testTabView" orientation="bottom" >
    <emp:tab label="balance" cacheData="true" name="balanceTab"
              dataSrc="balanceInfo.do"/>
    <emp:tab label="trasfer" active="true" name="transferTab">
       <DIV id="transferContent">
             <emp:form ...>
             </emp:form>
        </DIV>
     </emp:tab>
     <emp:tab label="balanceEnquiry" name="balanceEnquiry"
              cacheData="true" dataSrc="balanceEnquiry.do" />
</emp:tabView>
```

这个页面定义了一组三个标签页。第一个和第三个调用了某个 MVC Action 来取得其中的内容，而第二个标签页的内容直接在页面中进行了定义。

5.5.4. 联动选择框

5.5.4.1. 介绍及原理

在提交表单时，经常会用到多个相关的下拉选择框，比如在第一个下拉框中选择省份之后，第二个下拉框中会出现该省份对应的城市的选项。若一次性将所有组合读入，则传输量

是相当大的，会造成资源浪费、效率低下。为解决这一问题，EMP 可将多个下拉框划为一个分组，并使用事件监听机制监测下拉框的选择。一旦用户改变了某个下拉框的值，则发送 Ajax 异步请求执行某个特定的 MVC Action 去获取下一级下拉框的内容，然后将返回结果填充到其中，实现下拉选择框间的联动。具体实现如下：

设联动数据来源 Action 为 addressInfo.do，联动的三级下拉框为 country,province 和 city。当某一个下拉框（最后一个除外）的选项被改变时，会按照注册顺序将第一个到发生变化的下拉框值作为参数一并提交。即：

选择某个国家后发送：addressInfo.do?country=China

选择某个省份后发送：addressInfo.do?country=China&province=Hebei

作为数据来源的 addressInfo.do 返回“值:描述;值:描述;”格式的字符串，如 Hebei:河北;Henan:河南;Shandong:山东;Shanxi:山西;，这些选项会自动被填入下一个下拉框。用户可以扩展 controller 来生成这样的字符串；或者直接用 normal Controller 指向一个 JSP 文件，在 JSP 中生成。

联动选择框由 emp: relatedComboBoxGroup 和 emp: relatedComboBox 两个标签组成，由一个 Group 嵌套多个 ComboBox，按照顺序依次联动。注意这两个标签并不在页面上生成下拉框元素，而是将页面中已存在的下拉框注册为联动组。因此用户仍然需要使用 emp:combobox 或普通 HTML combobox 生成下拉框。此外，也可以不通过这两个标签，直接在页面上书写 JavaScript 脚本来完成同样的注册工作。

联动选择框组件用到了 YUI 的 event 和 connection 组件，在使用时需要引入相关的 JS 文件。

5.5.4.2. 参数定义

- emp: relatedComboBoxGroup 相关属性：

参数	说明
comboDataSrc	联动数据来源 URL

- emp: relatedComboBox 相关属性：

参数	说明
comboBoxId	要注册到联动组的下拉框 id

5.5.4.3. 使用实例

使用 TagLib 进行定义：

```
<emp:relatedComboBoxGroup id="grp1" comboDataSrc="addressInfo.do">
   <emp:relatedComboBox comboBoxId="country"/>
   <emp:relatedComboBox comboBoxId="province"/>
   <emp:relatedComboBox comboBoxId="city"/>
</emp:relatedComboBoxGroup>
```

使用 JavaScript 进行定义：

```
//选项数据来源URL
var dataSrc = '<ctp:jspURL jspFileName="addressInfo.do"/>';
//生成Combobox分组
var comboxGrp = new EMP.widget.ComboboxGroup(dataSrc);
//下面两句将三个comboBox注册到该分组中
comboxGrp.registCombobox(document.getElementById('country'));
comboxGrp.registCombobox(document.getElementById('province'));
comboxGrp.registCombobox(document.getElementById('city'));
//初始化country中的内容
comboxGrp.doInitComboboxContent(null,document.getElementById('country'));
```

实际上，上述 TagLib 生成的页面内容就是下面的 JavaScript。

5.5.5. 日期输入域

5.5.5.1. 介绍及原理

在填写表单或搜索条件时，经常会遇到日期型的数据，因此日期输入域自然也成为 Web 应用的一个常用页面组件。EMP 对 Yahoo UI 的 Calendar 组件进行了封装，将其与页面上的输入框对应起来，实现了输入与选择操作的统一。当输入框有初始值、或用户填写了日期后，日历中自动显示输入框中日期对应的月份；反之在日历中选择了一个日期之后，会自动将内容填充到输入框中。支持弹出或一直显示两种日历的显示方式。此外用户也可以通过 JS 脚本进行扩展，如将输入框换成下拉框等形式，实现更丰富的页面展现。

联动选择框组件用到了 YUI 的 calendar 组件，在使用时需要引入相关的 JS 和 CSS 文件。

5.5.5.2. 参数定义

- emp: calendar 相关属性：

参数	说明
id	日期域 ID
type	显示类型 popup / fixed
divId	日历所在的 DIV
autoClose	弹出情况下，选择日期后自动关闭
title	日历标题
inputYear	代表年份的输入框
inputMonth	代表月份的输入框
inputDay	代表日期的输入框

5.5.5.3. 使用实例

```
<emp:input name="year" type="text" size="4" />年
<emp:input name="month" type="text" size="2" />月
<emp:input name="day" type="text" size="2" />日
<emp:calendar type="popup" autoClose="false" title="请选择日期" inputYear="year" inputMonth="month"
inputDay="day" />
```

5.5.6. 其他Taglib

5.5.6.1. emp:label

emp:label 标签用于在页面中显示字符串。与直接在页面源码中书写字符串不同的是，该标签可以自动通过客户端的语言选择，从静态多语言资源定义中得到对应语言的字符串显示。静态多语言资源以一个 resource.xml 文件进行定义，对每个资源 id，维护了多种语言的对应描述。在页面标签中只需提供资源 id，就会按照客户端当前的语言选择（request 中提交名为 locale 的 parameter）自动显示该语言对应的描述信息。以下各种 Taglib 在显示静态字符串的时候也都支持静态多语言资源定义。

属性名称	说明
name	名称
text	字符串，系统会以此为 ID 从多语言定义中查找字符串，如果没有，则直接输出此字符串
CSSClass	指定样式，如果设定了此值，系统以： text 的方式输出字符串
cltType	客户端类别：wml,kjava,browser(默认)

5.5.6.2. emp:URL

emp:URL 标签用与在页面上生成一个超级链接。与普通的<a>锚点不同的是，它会自动为 URL 添加上 WebContext 路径的信息，并配合 EMP 框架中的 Sesssion 管理，添加必要的 Session 跟踪参数，并与前端框架配合完成用户工作区的更新。实际效果为在页面中生成一个<a>标签。

属性名称	说明
name	表单名称
href	锚点指向的 URL，允许以$dataName 的方式附加 EMP 中的数据模型数据作为参数：如:href=“abc.do?aa=$data1&bb=$data2”
label	显示的字符串，自动从多语言资源中查找
target	点击 URL 后提交的方式：parent：覆盖本窗口； newwin：打开新窗口，此属性优先于 contentDivId
contentDivId	将点击后的返回页面更新到此 id 指定的 DIV 区域
imageFile	用于显示锚点的图片，相对于根路径
needLocale	是否需要对 image 进行多语言处置，即在 imageName 后附加语言，如：a.jpg -> a_zh_CN.jpg
CSSClass	附加给锚点的 class 定义

5.5.6.3. emp:action

emp:action 标签用来对 HTML 组件参数或 JS 脚本中出现的 URL 进行处理。它和 emp:URL 一样，会为 URL 添加 Session 跟踪参数等框架必要信息，但使用更为灵活，可以使用在 JavaScript 脚本，以及 HTML 组件（Form，Href 等）的参数中。实际效果为在页面中生成处理后的 URL 字符串。

属性名称	说明
name	名称
action	指向的 URL，允许以$dataName 的方式附加 EMP 中的数据模型数据作为参数：如:href=“abc.do?aa=$data1&bb=$data2” 用例： var href = "<emp:action action="ccbHeader.do"/>"; updateDivContent(href, "hd", "");

5.5.6.4. emp:text

emp:text 标签用于在页面中输出 EMP 数据定义中的数据，支持通过 EMP 框架中定义的

数据类型处理对数值进行转换。实际效果为在页面中生成取值字符串。

此外，emp:text 还有一个重要功能：可以通过 EMP 中定义的数据集合(iColl)来翻译此数据值。所谓翻译，就是指将该数据域在后台的实际取值转换为前台显示的描述值，例如某币种数据域的实际取值为 001，前台显示人民币。要实现这种翻译功能，只需为该 text 域指定一个含有 value 和 desc 对应关系的 iColl 即可。数据翻译也支持多语言资源显示，不同于之前所述静态多语言定义，这里的多语言资源是与数据绑定的，因此不定义在 resource.xml 中，而需要在指定 iColl 中存在诸如 desc_zh_CN、desc_en_US 这样的字段即可，标签会根据用户选择的 locale 值自动取用相应字段进行显示。

属性名称	说明
name	名称
dataName	数据定义名称，通过它从 Context 中得到数据，如果从 Context 中得不到，则通过 request.getAttribute 的方法得到（便于使用 tld 进行数据处理），系统会试图通过取得 dataName_language 如 dataName_zh_CN 的方式得到数值
dataType	数据类型定义 ID，如果不定义，系统默认使用数据定义中指定的数据类型处理
mapCollName	用来翻译此值的数据集合定义 indexedCollection
mapValueName	在数据集合定义中的数值定义数据名称
mapDescName	在数据集合定义中的数值定义对应的翻译数据名称

5.5.6.5. emp:image

emp:image 标签用于在页面中引用一张图片。实际效果为在页面中生成<img>标签。它与普通<img>标签的区别在于会对图片来源 URL 进行转换（同 emp:URL）。另外，它也支持多语言对应的图片显示，它会自动在图片文件名后附加当前 locale 的值，以实现对不同图片的引用，如 a.jpg->a_zh_CN.jpg。

属性名称	说明
name	表单名称
src	图片来源，如果是以绝对路径，则不作任何变化
alt	图片的替代字符串，自动通过多语言资源进行翻译
needLocale	是否需要对 image 进行多语言处置，即在 imageName 后附加语言，如：a.jpg -> a_zh_CN.jpg
customAttr	附加的客户定义内容，直接附在 img 定义后

5.5.6.6. emp:form

emp:form 标签用于在页面中生成表单。实际效果为在页面中生成<form>标签和相关脚本。它同样会对 URL 进行转换，并与前端框架配合完成用户工作区的更新。它也会统一生成对表单中所有输入域的必填和数据类型前端校验脚本代码。

属性名称	说明
name	表单名称
action	表单提交的 Action
enctype	表单编码方式： application/x-www-form-urlencoded：标准表单 multipart/form-data：文件上传
method	表单提交方法：POST,GET
cltType	客户端类别：wml,kjava,browser(默认)
contentDivId	将表单提交后的返回页面更新到此 id 指定的 DIV 区域
target	
customAttr	客户附加的属性，会被附在 form 定义中

5.5.6.7. emp:input

emp:input 标签用于生成表单中的输入域，实际效果为在页面中生成<input>标签和相关 JavaScript。与 HTML Input 一样，它也支持 text，hidden，submit 等多种展示方式。如果将其定义在 emp:form 内，它会自动配合数据的类型定义，实现输入域的客户端 JavaScript 校验。

属性名称	说明
name	输入域名称
type	输入域的类型
dataName	输入域对应的 EMP 数据定义名称，如果不设，则直接使用 name
valueDataName	从此属性定义的数据名称中获得值，并赋值给 value
valueRequestName	获得此属性定义的请求参数(request.getParamater)，并赋值给 value
value	Value 赋值：1.valueRequestName，2.dataName 对应的 value，3.valueDataName 对应的 value，4.value 设定的值
dataType	客户端类别：wml,kjava,browser(默认)
required	是否为必输项
size	大小
customAttr	客户附加的属性，会被附在 input 定义中

CSSClass	class 值

5.5.6.8. emp:combobox

emp:combobox 标签用于在表单中生成下拉选择框，或者列表选择框，实际效果为生成<select>标签。供选择的项目可以来自用户直接定义，也可以来自 iColl，这两种方式都支持数据的值-描述翻译及多语言支持。此外，它可以配合联动选择框组件，实现多级下拉框的联动选择。

属性名称	说明
name	输入域名称
dataName	输入域对应的 EMP 数据定义名称，如果不设，则直接使用 name
dataTypeName	数据类型定义，用于格式化选项值输出
optionSrc	选项来源　0：从数据定义 1：用户自定义
iCollName	从数据定义中得到选项值，指定 IndexedCollection 的 ID
valueName	选项值的数据名称
descName	选项的显示字串的数据名称，如果不设直接使用 valueName
languageResouce	是否需要按照多语言的规则得到选项的显示字串
itemStr	用户定义的选项：value:desc;vale1;desc1;...
size	大小,如果>1 代表是按照 List 的方式定义选择框
multiple	是否允许多选，只对 List 选项时有效
customAttr	客户附加的属性，会被附在 form 定义中
cltType	客户端类别
CSSClass	class 值

5.5.6.9. emp:selection

emp:selection 标签用于在表单中生成单选(radio)或复选(check)框，实际效果为生成一组<input>标签。供选择的项目可以来自用户直接定义，也可以来自 iColl，这两种方式都支持数据的值-描述翻译及多语言支持。

属性名称	说明
name	输入域名称
dataName	输入域对应的 EMP 数据定义名称，如果不设，则直接使用 name
dataTypeName	数据类型定义，用于格式化选项值输出
optionSrc	/选项来源　0：从数据定义 1：用户自定义
iCollName	从数据定义中得到选项值，指定 IndexedCollection 的 ID
valueName	选项值
descName	选项的显示字串,如果不设，直接使用 valueName
languageResouce	是否需要按照多语言的规则得到选项的显示字串

itemStr	用户定义的选项：value:desc;vale1;desc1;...
type	类别：checkbox radiobox
label	手工设定显示串，自动通过多语言定义资源翻译
layout	“table”如果设定为 table 则按照 table 的方式输出
customAttr	客户附加的属性，会被附在 form 定义中
CSSClass	class 值

5.5.6.10. emp:textArea

emp:textArea 用于在表单中生成多行文本输入域，用于内容比较多的文本输入。实际效果为在页面中生成<textarea>标签。

属性名称	说明
name	输入域名称
dataName	输入域对应的 EMP 数据定义名称，如果不设，则直接使用 name
valueDataName	从此属性定义的数据名称中获得值，并赋值给 value
cols	列数
rows	行数
customAttr	客户附加的属性，会被附在 form 定义中
CSSClass	class 值

5.5.6.11. emp:fieldMessage

emp:fieldMessage 用于在页面中输出表单输入域的服务器端校验错误信息。在用户第一次进入表单页面时，并不存在服务端校验错误信息，该标签不生成任何内容。当用户提交表单后，若有数据在服务端的检验和未通过，则可为检验不通过的数据设置各自的错误信息，并保存在一个内置的 kColl 中。当返回原页面后，此标签会去该 kColl 中查找 dataName 对应的错误信息，并显示在页面上。

属性名称	说明
name	输入域名称
dataName	输入域对应的 EMP 数据定义名称，如果不设，则直接使用 name
CSSClass	class 值

5.5.6.12. emp:file

emp:file 类似 emp:action，但它只为指定 URL 追加 WebContext 路径，而不关心 Session 等相关信息。

属性名称	说明
name	输入域名称
fileName	文档名称

5.5.6.13. emp:dynPassword

emp:dynPassword 用于生成动态密码键盘密码输入域，它需要配合 DynamicPasswordController 使用。用户可通过软键盘方式输入密码等敏感信息，防止键盘记录工具等泄露重要信息。它可以实现输入的自动编码，避免真实密码在客户端出现。键盘样式可以通过图片方式指定。

属性名称	说明
name	输入域名称
dataName	输入域对应的 EMP 数据定义名称，如果不设，则直接使用 name
value	Value 赋值
required	是否为必输项
size	大小
keyboardType	键盘类别：NUM 数字键盘，FULL 全键盘
keyBoardTitle	键盘的显示 Title
keyBoardWidth	宽度，用于计算用户的鼠标点击点对应的键值，键盘的大小为提供的图片背景大小
keyBoardHeight	高度
nextFieldName	下一个输入域的名称
encodeKey	是否编码键盘
imgSrc	用于生成键盘图片的 URI, 需要在 MVC 中定义基于 DynamicPasswordController 的 Action
customAttr	客户附加的属性，会被附在 form 定义中
CSSClass	class 值

5.5.6.14. emp:appendPin

emp:appendPin 用于在页面中生成附加码图片，它需要配合 DynamicPasswordController 使用。它的原理是，在进入标签时调用指定 URL（对应上述 controller）生成随机附加码，保存在 session 中，同时返回给客户端相应图片。当用户提交表单后，服务端会从 session 中取得附加码，与用户所填写的内容进行比较。若不通过，则返回之前页面。

属性名称	说明
name	输入域名称
length	附加码长度
imgSrc	用于生成图片的 URI, 需要在 MVC 中定义基于 DynamicPasswordController 的 Action

customAttr	客户附加的属性，会被附在 form 定义中
CSSClass	class 值

6. EMP多渠道处理

多渠道接入处理框架是 EMP 平台提供的渠道整合体系下的多渠道接入的技术解决方案。EMP 平台不但提供了银行多渠道接入的平台分层设计方法，更是针对银行当前流行的渠道系统接入实现的技术特点，在统一的多渠道接入体系下，提供了基于 TCPIP、HTTP 和 WebService 模式下的渠道接入实现方案。

- 这里所说的多渠道处理并不是业务层面上的多渠道
- 这里所说的多渠道是指技术层面上的，而实际上是对多种访问协议的支持
- EMP 通过在框架上灵活支持不同的通信协议以及报文格式，来满足多渠道处理的需求 ，并实现业务处理逻辑在不同渠道应用中的共享

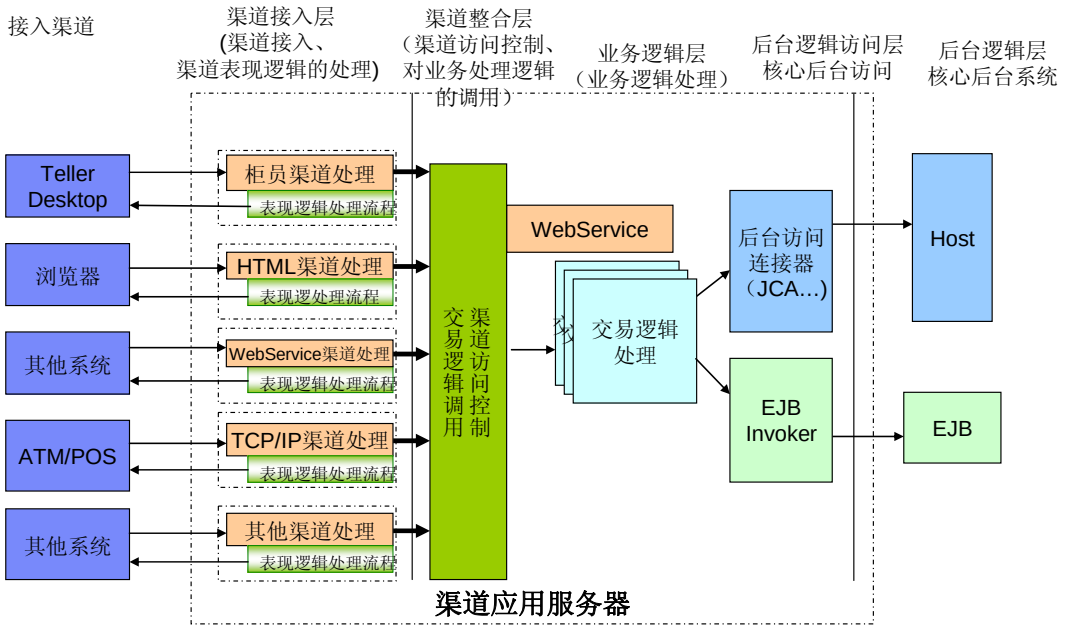
6.1. EMP多渠道处理模型

6.1.1. 渠道整合的历史分析

多渠道接入处理是银行业务系统的一个永恒话题。原因在于银行的长期发展过程中不同的渠道系统出现的时期不一致，使用的技术不一致，导致各个渠道系统建设上的系统孤立，每个渠道系统都自成一体。这种孤立导致了银行业务处理上的重复建设，客户可以在不同渠道上完成相同的业务流程，但由于不同的渠道系统设计结构的无法兼容，导致我们往往需要在不同的渠道系统上将相同的业务进行重复的设计和开发。这种长期形成的渠道各自为政的局面，不但导致银行渠道系统随着渠道建设的不断深入系统环境越来越复杂，维护成本和升级成本也越来越高。更重要的是随着银行业务的不断发展，业务要求在统一的平台上进行业务整合、客户信息的整合，而由于各个渠道系统的技术实现不一致而导致的整合困难。

6.1.2. EMP的多渠道整合与分层设计

由以上的原因我们可以看到银行在渠道整合上要求的是通过技术平台的实现带来业务信息的可整合能力。理想状态下的渠道接入的方式如下图所示：



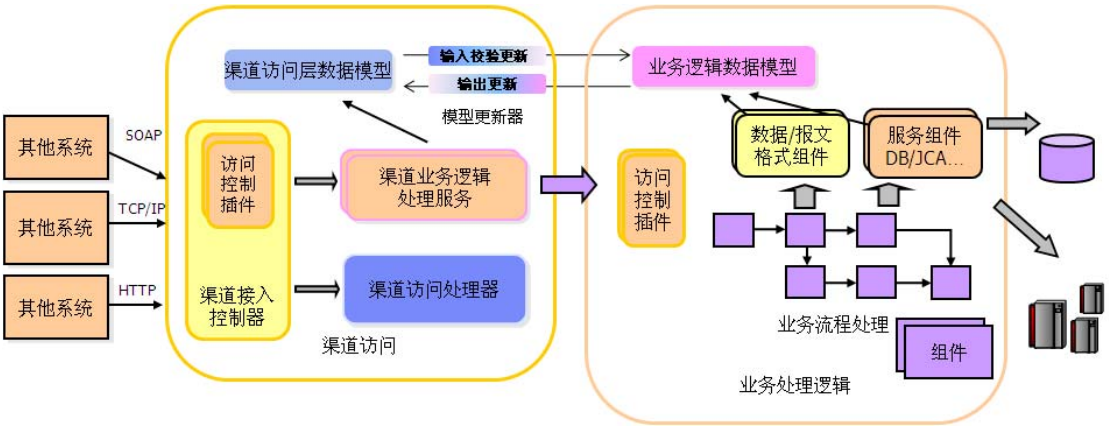
如上图，我们将银行渠道系统采用分层结构的方式进行解构，分别是接入层、渠道表现层、渠道整合层、业务逻辑层、后台逻辑访问层和后台数据层。不同的渠道需要不同的渠道表现，不同的渠道有不同的渠道逻辑，但相同的渠道交易请求所调用同样的业务逻辑处理。在分层结构中，银行多渠道接入层的设计核心在于在银行渠道系统的渠道表现层、渠道逻辑层和业务逻辑层进行完全的解藕处理。各个层面之间采用接口化扩展的标准处理进行交互，在应用上保持整个系统框架的稳定性，采用外部化配置模式进行渠道接入功能扩展的定制，提供最大化的多渠道接入的高效稳定和灵活可扩展能力。

EMP 平台所提供的多渠道接入框架正是采用这种分层设计思想，通过分层解藕的方式为用户提供的多渠道接入处理框架。

6.1.3. EMP多渠道接入框架设计

EMP 运行平台还在基础架构上，为应用系统提供了多渠道的处理架构，让应用系统支持多渠道的处理架构，透过其内建的多渠道处理组件（这一组件同时提供了灵活的扩展和重组机制），允许不同的渠道接入到 EMP 应用系统中，然后调用相应的业务处理逻辑进行业务处理，最终以渠道相关的展现逻辑将结果展现给渠道用户。

EMP 平台在渠道接入框架处理上也是采用渠道处理插件的方式提供灵活的渠道处理扩展能力。渠道接入框架基础设计如下图所示：



在 EMP 平台上，业务处理逻辑层和渠道访问逻辑层是完全分开的，业务处理逻辑提供专门的业务逻辑处理功能，并开放外部调用接口方法供外部系统访问。

渠道访问逻辑层是 EMP 平台的多渠道接入框架的基本构成，面对外部渠道的接入，渠道接入控制器是渠道接入框架的核心组件。多渠道接入框架提供了渠道访问处理器来进行渠道访问的各种通信协议的配置处理，同时多渠道访问框架中也支持插入式的访问控制插件，对不同的渠道接入可提供不同的访问控制插件进行访问控制处理。在对后台业务系统进行访问时，EMP 多渠道接入框架提供了渠道业务逻辑处理服务供用户进行扩展，针对不同渠道系统的报文处理需求和返回格式要求开发不同的处理模式。渠道访问层与业务逻辑层分别拥有各自的数据模型，之间通过模型更新器进行数据的校验、数据转换和数据更新完成数据传递。

渠道接入处理基本流程如下：

一、渠道接入控制器接收到渠道系统的访问请求，调用渠道访问处理器根据请求的协议和报文规范进行数据解包；

二、如果有访问控制插件，则渠道接入控制器调用访问控制插件根据请求数据信息进行访问控制检查，如果不满足访问条件，则抛出异常，拒绝访问；

三、渠道接入控制器调用配置的渠道业务逻辑处理服务组件将收到的请求数据组装成渠道访问层数据模型，通过输入数据模型更新器，对数据进行校验、转换，把数据设置到所调用的业务逻辑处理组件的数据模型中完成数据传递；

四、在业务逻辑层处理结束后，渠道接入控制器调用渠道业务逻辑处理服务，通过输出模型更新器对业务逻辑层数据模型中的数据进行转换，把数据更新到渠道访问层的数据模型中，渠道业务逻辑处理服务按照渠道所要求的方式进行组包返回，并由渠道访问处理器解释为渠道所能理解的表现逻辑。

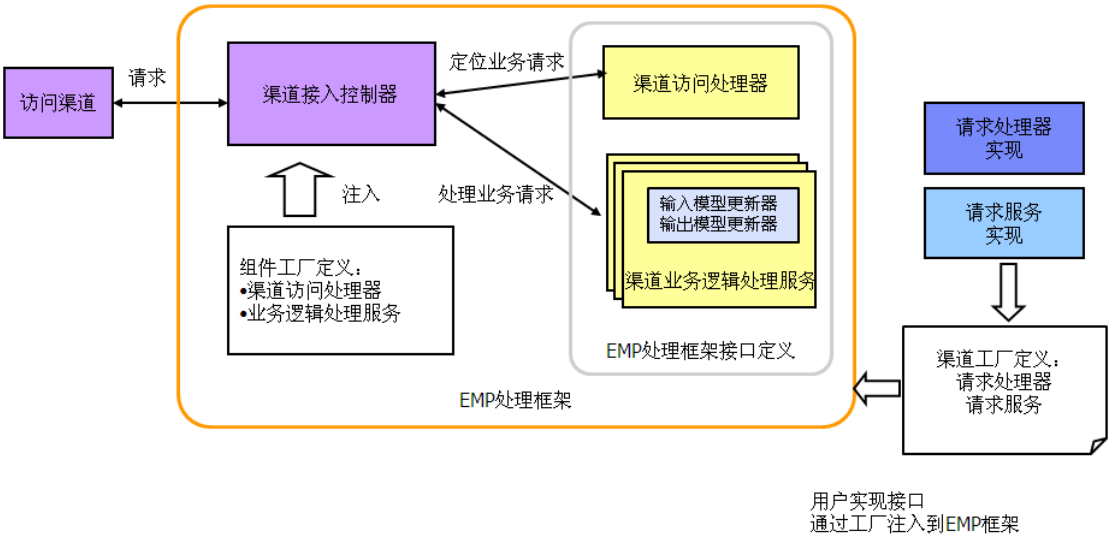
由上所述，我们知道，EMP 平台的多渠道接入框架的入口和控制器是渠道接入控制器组件，在 EMP 平台中它表现为一个继承于 servlet 的对象，我们在该对象中进行渠道接入的基础配置信息导入和接入处理组件的初始化并控制业务逻辑的调用处理。这是渠道处理的核心：通过接收渠道请求，转化渠道请求为标准的平台内部数据，调用平台业务逻辑处理，并将平台处理结果转化为渠道所能识别的渠道数据并返回给渠道。

渠道访问处理器是一个由渠道访问处理器初始化并调用的组件，它为不同的渠道提供满足渠道访问协议和报文规范要求的报文处理。

渠道业务逻辑处理服务是由渠道访问控制器初始化并调用的组件，每个服务组件提供了对某个业务逻辑的调用，包括对渠道传输数据的打包解包以及通过模型更新器和业务逻辑层的数据模型进行数据更新。

访问控制器也是一个插件，由渠道接入控制器进行初始化和配置并接收渠道访问处理器的调用和管理。

6.1.4. 渠道访问处理框架设计



如上图所示，渠道访问逻辑层主要由三大组件构成，分别是渠道接入控制器、渠道访问处理器和渠道业务逻辑处理服务组件。

- **渠道接入控制器**

在 EMP 框架内提供的渠道接入控制器，实现组件工厂式的处理逻辑注入，融入 EMP 访问控制框架。

- **渠道访问处理器**

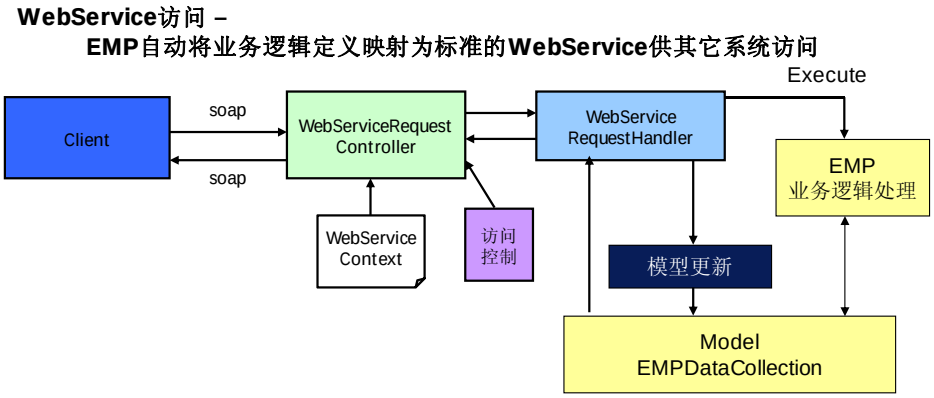
处理渠道访问通信层协议，通过请求内容定位具体的业务逻辑请求。

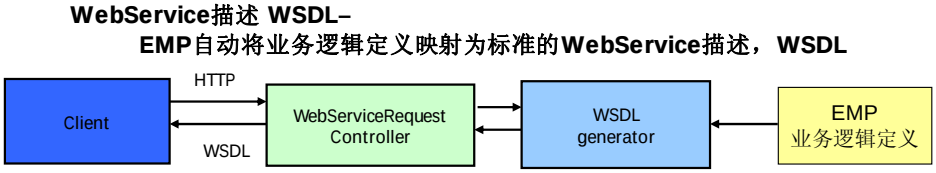
- **渠道业务逻辑处理服务**

实现渠道接入具体的业务请求的逻辑处理功能。EMP 平台分层设计将业务逻辑处理组件完全与表现渠道分离，通过模型更新器对渠道访问层数据模型和业务逻辑层数据模型之间的数据进行校验、转换和更新。渠道业务逻辑处理服务是一个针对不同渠道接入的业务逻辑封装，实际作用是将平台上的业务逻辑处理构件通过渠道业务逻辑处理服务的封装可以很方便地发布到指定的渠道上，EMP 平台通过这种方式来保证统一的业务逻辑处理组件在不同渠道上的重用。

6.2. WebService访问

6.2.1. WebService渠道访问的实现





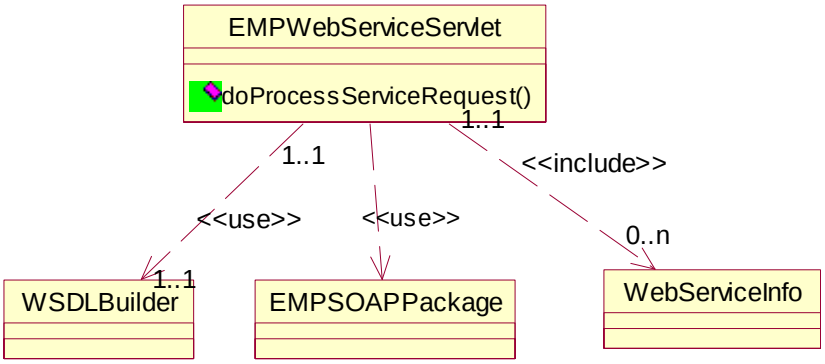
如上图，WebService 渠道访问是指 EMP 平台提供基于 SOAP 协议上的 webService 访问支持。EMP 平台上的所有的业务逻辑处理组件可以自动发布为一个 Web Service 对象，外部系统可通过标准的 SOAP 协议进行业务逻辑的访问，只需要在渠道访问配置文件中指明一个 webService 命名所对应的业务逻辑处理名称即可。webService 渠道访问组件可以提供标准的 WDSL 文件内容和标准的 webservice 协议访问功能。

EMP 平台上的 webservice 渠道访问由于是采用的业界标准的访问协议，它的组包方式

和通信方式都已经被确定下来，EMP 平台 webservice 渠道访问组件只需要提供标准的 WSDL 文件生成器和 SOAP 协议访问包文组包功能即可满足用户需要。

EMP 平台上的 webservice 渠道访问组件已经提供了完整的 webservice 访问功能，并不需要用户在此基础上再进行其他的扩展，只需要通过配置文件进行使用。

下面的类图描述了 EMP 平台是如何支持 webservice 渠道访问处理功能的。



如上图所示，webservice 渠道访问组件一共由四个类构成。

EMPWebServiceServlet 是一个 webservice 访问接入对象，提供标准的 webservice URL 访问支持。它通过调用 webservice 协议支持类进行 webservice 协议与 EMP 平台业务逻辑之间的自动转化，完成从一个 webservice 标准访问映射到一个具体的业务逻辑处理组件。

WebServiceInfo 类是一个 webservice 对象与 EMP 平台业务对象映射的关系描述类。一个 webservice 对象名称将映射到一个业务逻辑对象 BizLogic。Webservice 对象定义是提供给外部系统访问的 webservice 名称，业务逻辑对象定义是内部定义的该 webservice 对象定义所对应的业务逻辑操作流程对象。通过该类将 webservice 与平台内部业务逻辑对象关联起来。

在配置文件中，我们按照以下方式对一个 webservice 所对应的业务逻辑对象进行描述：

<WebService id="transferFlow" EMPFlowId="transferFlow" />

Id 为 webservice 的名称，EMPFlowId 为 EMP 平台中定义的业务逻辑对象。注意，一个 EMPFlowId 对应的是一个 EMP 平台中的业务逻辑结合对象，在该集合中包含多个操作流程对象 operation，在实际应用中，webservice 调用通过外部的 SOAP 协议包中的字段指定所需要调用的具体的 operation 对象名称来确定该 webservice 的请求的具体业务流程。

WSDLBuilder 类是一个将 EMP 平台的业务逻辑对象自动转化的 WSDL 文件描述对象的生成器。我们在通过 get 方法访问一个 EMP 平台上的 webservice 定义时，系统会自动将 webservice 对应的业务逻辑对象定义转化为 webservice 文件描述内容返回。

EMPSOAPPackage 类是一个对 SOAP 协议包进行解包和组包操作的处理类，它将 SOAP

协议包首先进行解包获取上送的数据，并从这些数据中获得 webservice 所要调用的具体的 operation 和所有的请求数据，并将这些数据转化为 EMP 平台所能使用的 data 数据对象，在返回时，该类将 EMP 平台的数据模型中的数据自动组织为 SOAP 协议包格式进行返回。

EMPWebServiceServlet 对象是 webservice 渠道访问处理器对象，该对象提供了 webservice 对象定义的初始化工作，由该对象统一接收 webservice 请求和统一调度 webservice 请求的业务逻辑处理和数据返回。它通过 get 和 post 方法分别提供 WSDL 文件描述返回和 webservice 操作请求处理。

Get 方法将返回一个指定的 service 的 WSDL 描述文件内容。

Post 方法将返回一个指定的 service 的操作执行结果。

6.2.2. WebService渠道访问使用说明

Web Service 渠道的配置信息最为简单，因为它是一个业界标准的访问协议，不需要自定义的插件进行处理，或者自定义的格式来进行转换，我们在定义 Web Service 渠道插件信息时，只需要在其中定义 web service 服务与 EMP 业务组件之间的映射信息即可。基本配置信息如下：

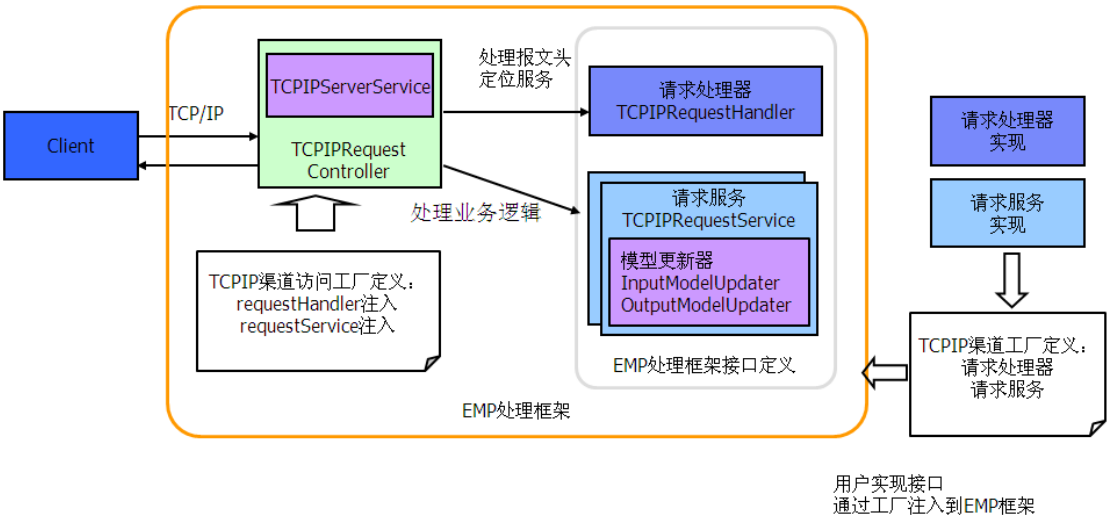
```
<WebService id="transferFlow" EMPFlowId="transferFlow" />
```

一个 webservice 的 ID，对应一个 EMPFlowId，完成从 webservice 访问到 EMP 业务组件之间的映射访问。

6.3. TCP/IP访问

6.3.1. TCP/IP渠道访问的实现

TCPIP 渠道访问组件是所有渠道访问组件里面最复杂的一个处理组件，因为 TCPIP 协议的灵活性以及报文格式的自由度要求 TCPIP 渠道访问组件拥有足够的灵活度和可扩展能力。我们在 EMP 平台所提供 TCPIP 渠道访问组件里将借助于 EMP 平台提供的 TCPIP 通信组件来提供完整的可配置的 TCPIP 通信协议的处理，通过扩展 TCPIPRequestHandler 接口来提供不同的渠道报文协议处理的特殊要求，通过扩展 TCPIPRequestService 接口实现对业务逻辑的调用。



EMPTCPIPServiceServlet 类在系统初始化时对 TCPIP 渠道接入处理功能进行初始化和配置。在 EMPTCPIPServiceServlet 所关联的配置文件中，我们需要为该 TCPIP 协议接入提供TCPIP通信服务对象。TCPIP通信服务对象需要配置报文接收处理接口：PackageProcessor，EMP 平台提供了缺省的实现类 EMPPackageProcessor，就是 TCPIP 渠道控制器，TCPIP 渠道接入框架就是在这个实现类的接口方法实现的。在实际应用中，一般不用扩展，若用户有特殊需求，则需要继承该 EMPPackageProcessor 对象，并重载实现 PackageProcessor 接口所定义的方法，或者直接实现 PackageProcessor 接口。

接口类 TCPIPRequestHandler 是一个 TCPIP 接入处理插件，用于处理 TCP/IP 请求的报文头，以及从报文头获取必要的信息。它的基本接口方法定义如下表所示：

Interface	TCPIPRequestHandler	Package	com.ecc.emp.access.tcpip
Interface Format	public Interface TCPIPRequestHandler		
Implements			

Name	Parameters	Return Value	Exceptions	Description
isRequestPackage	byte[] msg	boolean		判断该数据包是否为所需的业务请求数据包

getTCPIPRequest	byte[] reqMsg 请求数据包	EMPTCPIPRequest		获得请求的 EMPTCPIPRequest 封装对象。 从请求报文中解出报文头，从中取得 Session id 和 Service id，连同报文体一起封装为 EMPTCPIPRequest 以进行进一步处理。
getResponse	EMPTCPIPRequest request 请求封装对象, byte[] retMsg 返回报文体	byte[] retMsg		处理正常的响应报文。 将数据打包成返回报文头，并和返回报文体组装后返回。
getExceptionResponse	EMPTCPIPRequest request 请求封装对象, Exception e	byte[] retMsg		处理异常时的响应报文。 将数据和异常打包成返回报文头。

接口 TCPIPRequestService 定义了一个 TCPIP 渠道业务处理服务。它的接口方法定义如下：

Interface	TCPIPRequestService	Package	com.ecc.emp.access.tcpip
Interface Format	public Interface TCPIPRequestService		
Implements			

Name	Parameters	Return Value	Exceptions	Description
getServiceName		String		获得渠道业务逻辑处理服务名称
handleRequest	EMPTCPIPRequest request, Context sessionContext	byte[]	EMPException	具体的请求处理实现方法。返回打包后的报文体。
setComponentFactory	ComponentFactory factory			注入组件工厂
isSessionService		boolean		判断请求是否需要创建 session
isEndSessionService		boolean		判断请求是否要销毁 session
isCheckSession		boolean		判断该请求是否检查会话

接口 InputModelUpdater 定义了输入模型更新器，负责把数据从渠道访问层数据模型更

新到业务逻辑处理层数据模型中，并在更新过程中根据数据类型进行数据校验和数据转换。它的接口方法定义如下：

Interface	InputModelUpdater	Package	com.ecc.emp.access.tcpip.modelUpdater
Interface Format	public Interface InputModelUpdater		
Implements			

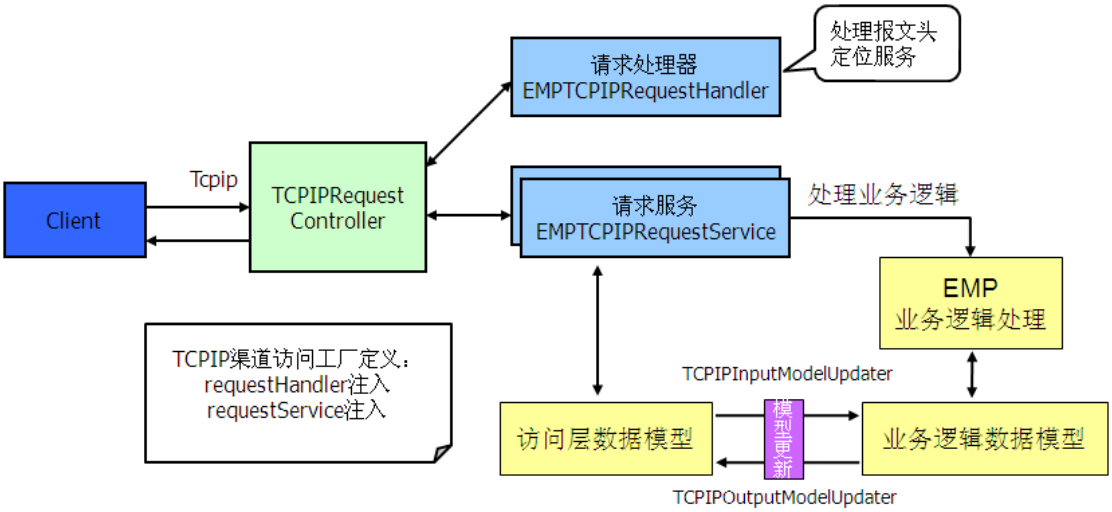
Name	Parameters	Return Value	Exceptions	Description
updateModel	Object requestData 访问层数据模型, Context opContext 业务逻辑数据模型, KeyedCollection input 业务逻辑中的输入定义, Map dataTypeDefines EMP 中的数据类型集合		EMPException	将数据按照业务逻辑构件定义更新到业务逻辑层数据模型 Context 中

接口 OutputModelUpdter 定义了输出数据模型更新器，在业务逻辑处理完成之后，负责把数据从业务逻辑层数据模型中更新到访问层数据模型中，并在更新过程中按照数据类型进行数据转换。它的接口方法定义如下：

Interface	OutputModelUpdater	Package	com.ecc.emp.access.tcpip.modelUpdater
Interface Format	public Interface OutputModelUpdater		
Implements			

Name	Parameters	Return Value	Exceptions	Description
updateModel	Object responseData 访问层数据模型, Context opContext 业务逻辑数据模型, KeyedCollection outputData 业务逻辑中的输出数据, EMPTCPIPRequest request TCPIP 请求对象, Map dataTypeDefines EMP 中的数据类型集合		EMPException	将数据按照业务逻辑输出定义更新到访问层响应数据模型中

6.3.2. EMP实现的TCPIP渠道接入处理



如上图所示，一个客户端 tcpip 请求到达渠道处理端时，首先由渠道控制器调用请求处理器（EMPTcpipRequestHandler）去处理报文头，通过报文头解析定位服务。Tcpip 渠道控制器通过调用所请求的业务逻辑处理服务对象（EMPTCPIPRequestService）来执行一个 EMP 业务逻辑处理，业务逻辑处理服务对象（EMPTCPIPRequestService）在调用业务逻辑前后，分别使用输入、输出模型更新器（TCPIPInputModelUpdater、TCPIPOutputModelUpdater）在访问层数据模型和业务逻辑数据模型之间进行数据交换，。

6.3.3. TCPIP渠道使用说明

6.3.3.1. 建立和配置TCPIP渠道

EMP 平台中一个渠道的建立是通过一个特定的 Servlet 对象来启动和初始化配置参数的。EMP 为三种接入渠道分别提供了三个 Servlet 对象用于渠道参数初始化。

需要将该渠道的初始化 Servlet 对象加载到 web.xml 文件中。（按照标准的 Servlet 对象定义方式进行定义）

察看 Servlet 定义的内容并进行参数有效性的确认，如下为 Servlet 对象的参数配置内容：

```
<servlet>
  <servlet-name>EMPTCPIPService</servlet-name>
  <servlet-class>com.ecc.emp.access.tcpip.EMPTCPIPServiceServlet</servlet-class>
  <init-param>
   <param-name>factoryName</param-name>
```

```
    <param-value>DemoBiz</param-value>
  </init-param>
  <init-param>
    <param-name>iniFile</param-name>
    <param-value>WEB-INF/bizs/DemoBiz/settings.xml</param-value>
  </init-param>
  <init-param>
   <param-name>servletContextFile</param-name>
   <param-value>WEB-INF/channels/TCPIPServiceContext.xml</param-value>
  </init-param>
  <init-param>
   <param-name>sessionIdField</param-name>
   <param-value>sessionId</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
 </servlet>
```

参数及其含义，以及对该参数可用的修改如下表所示：

参数名	含义	示例	修改参考
EMPTCPIPService	启动的 Servlet 对象类名称	**com.ecc.emp.access.tcpip.EMPTCPIPServiceServlet**	不用修改。
factoryName	为你的渠道应用所访问的业务逻辑组件所在的业务工厂名称	**DemoBiz**	与你所采用的业务逻辑分组定义时所采用的名称相同。如在建立业务逻辑分组时定义名称为“testBiz”，则该处应改为“testBiz”。
iniFile	业务逻辑处理组件工厂文件定义	**WEB-INF/httpAccessServletContext.xml**	这个参数非必须参数，如果没有，则通过 **factoryName** 共享公共工厂定义
servletContextFile	渠道信息配置文件。该文件中定义了：渠道通信参数；接入接出的报文头定义；部署到该渠道上的业务逻辑构件的定义，包括请求和返回报文体的格式定义等	**WEB-INF/channels/TCPIPServiceContext.xml**	在生成应用时，系统已经提供了该渠道的配置文件基础内容，与该文件名相符。不用修改该参数。该文件内容可根据渠道定义相关属性进行修改。
sessionIdField	会话 ID 存放的字	**sessionId**	系统缺省定义，不用

	段名称。缺省使用 sessionId		修改
load-on-startup	当前 servlet 的启动顺序号	**2**	如果 TCPIP 渠道使用的组件工厂是共享别的 Servlet 创建的组件工厂，则此参数值必须比创建组件工厂的 Servlet 的启动顺序号大，确保 TCPIP 渠道控制器初始化时组件工厂已经创建。

在配置好 web.xml 文件后，再来看看 WEB-INF/channels/TCPIPServiceContext.xml 文件，在项目的 WEB-INF/channels 目录下的 TCPIPServiceContext.xml，该文件的主要内容如下（该文件中的非本章节关注内容可先忽略，并不影响理解渠道的接入实现）

```
<!-- 会话管理器定义 -->
<sessionManager name="EMPTCPIPServiceSessionMgr" sessionTimeOut="600000" sessionCheckInterval="120000"
        class="com.ecc.emp.session.EMPSessionManager"/>

<!-- 访问控制管理器定义 -->
<accessManager class="com.ecc.emp.accesscontrol.EMPAccessManager"/>

<!-- 通信接入服务 -->
<TCPIPServerService name="aService" keepAlive="false" dual="false" class="com.ecc.emp.tcpip.TCPIPServerService">
    <ListenPort id="listenPort" keepAlive="false" port="8010" maxConnection="50" poolSize="1" poolThread="true" waitTime="2000" class="com.ecc.emp.tcpip.ListenPort"/>
    <CommProcessor name="commProcessor" class="com.ecc.emp.tcpip.EMPCommProcessor"/>
    <PackageProcessor name="PackageProcessor" class="com.ecc.emp.access.tcpip.EMPPackageProcessor"/>
</TCPIPServerService>

<!-- 渠道处理器 -->
<TCPIPRequestHandler name="empTCPIPHandler" sessionIdField="sessionId" serviceIdField="serviceId" appendReqHead="true" appendRepHead="true" class="com.ecc.emp.access.tcpip.EMPTCPIPRequestHandler">
    <RequestHeadFormat id="headReqFmt" class="com.ecc.emp.format.FormatElement">
        <record>
            <fString dataName="sessionId">
                <delim delimChar="|"/>
            </fString>
            <fString dataName="serviceId">
                <delim delimChar="|"/>
```

```
            </fString>
            <delim delimChar="#"/>
        </record>
    </RequestHeadFormat>

    <ResponseHeadFormat id="headRepFmt" class="com.ecc.emp.format.FormatElement">
        <record>
            <fString dataName="errorCode">
                <delim delimChar="|"/>
            </fString>
            <fString dataName="sessionId">
                <delim delimChar="|"/>
            </fString>
            <delim delimChar="#"/>
        </record>
    </ResponseHeadFormat>

    <Map name="serviceIdMap">

    </Map>
</TCPIPRequestHandler>
```

该部分配置分为三个小块来了解，分别是会话管理器、通信接入服务和渠道接入处理器组件三大部分。

会话管理器的配置与其他渠道的配置相同。

我们要关注的是渠道接入处理器和通信接入服务组件两个关键组件。

先看渠道接入处理器组件。该对象是用于处理 TCPIP 请求报文头，并从报文头中获取必要的信息。

可配置参数名	参数定义
requestHeadFormat	请求报文头格式定义
responseHeadFormat	响应报文头格式定义
appendReqHead	在向 httpRequestService 传递数据时，是否需要截去请求报文头。 缺省为 true，可选值： true，截去请求报文头； false，不截去请求报文头。
appendRepHead	在处理返回报文时，是否需要在附带报文头，如果需要，将使用 responseHeadFormat 格式化后附在报文头部。 缺省为 true，可选值：

	true，追加报文头； false，不追加报文头。
serviceIdField	报文头解开后的 ServiceId 数据域，必须设置
sessionIdField	报文头解开后的 SessionId 数据域，非必须设置
serviceIdMap	请求的服务 id 到指定服务 ID 的对照表，允许将多个请求 ServiceId 映射到其他定义 ServiceId 中，用于交易分发
errorCodeField	错误码数据域定义，非必须设置
encoding	请求报文头对应的字符编码，非必须设置，缺省为 null

实际上从系统提供的缺省的例子看，提供了两个头格式定义，分别是字符串分隔符模式的头格式定义。在接入头格式内容中，定义了上传数据中的请求的业务构件服务名称和该请求的会话 ID 的值。在返回的头格式内容中，定义了返回数据中的错误信息代码（如果有的话）和会话 ID 的值。

另一个关键组件是通信协议服务组件，它采用了 EMP 提供的一个基础的 TCPIP 监听服务组件的定义，我们知道在一个 TCPIP 监听服务需要配置一个 commProcessor 用于处理协议报文头，还需要配置一个 PackageProcesser 用于处理协议报文体。

```
<ListenPort id="listenPort" keepAlive="false" port="8010" maxConnection="50" poolSize="1" poolThread="true" waitTime="2000" class="com.ecc.emp.tcpip.ListenPort"/>
```

该监听服务是监听端口 8010，采用线程池的方式处理请求连接，线程池中提供 1 个线程，在 poolThread 为 false 时，maxConnection 参数生效，表示最大 50 个的并发连接限制。具体详细参数含义，可以参见 EMP 技术组件章节中 TCPIP 通讯组件部分。可以根据实际设计修改以上参数。

通过以上的配置，实际上我们已经完成了我们的 TCPIP 渠道接入的基本技术实现。有了通信协议的实现和渠道接入流程的处理器对象的实现，就可以进行渠道接入服务了。

接下来，将我们的业务服务需要发布到 TCPIP 渠道上去的进行 TCPIP 渠道的发布。

6.3.3.2. 在渠道上快速发布业务

EMP 平台作为多渠道整合平台，业务逻辑构件作为独立的构件让所有渠道共享，我们只需要将需要在某个渠道发布的业务逻辑构件通过配置文件的配置注册到指定的渠道中即可。

在 TCPIP 渠道处理中，EMP 平台缺省提供了两种业务逻辑构件快速发布的模式，一种

是将业务逻辑构件发布为传输格式采用 EMP 平台定义的数据序列化方式的服务发布；另一种是采用用户自定义的数据传输格式方式的服务发布。第一种序列化方式是不需要用户定义需要传输的数据打包格式的，只需要在客户端构建访问的 Context 数据对象，然后进行访问即可，系统自动会将该数据对象进行序列化传输。采用序列化方式的好处是发布迅速，不需要关心格式处理，缺点是客户端也需要 EMPJAR 包支持（或者需要构造 EMP 序列化方法，等同于定义打包格式了）；采用用户定义传输格式的好处是灵活性和客户端的 EMP 无关性。大多数应用还是采用自定义传输格式的服务发布模式。

EMP 自定义传输格式的服务对象类 EMPTCPIPRequestService 是 TCPIPRequestService 接口的实现类。这个类中还定义了输入、输出模型更新器。

输入模型更新器 TCPIPInputModelUpdater 用在业务逻辑执行前，把数据从渠道访问层数据模型更新到业务逻辑层数据模型中，并在更新过程中进行数据校验和转换。

输出模型更新器 TCPIPOutputModelUpdater 用在业务逻辑执行后，把数据从业务逻辑层数据模型更新到渠道访问层数据模型中，并在更新过程中进行数据转换。

EMP 自定义传输格式的服务发布对象是在 TCPIPServiceContext.xml 文件中进行配置的。

自定义传输格式的服务发布对象的实现类：com.ecc.emp.access.tcpip.EMPTCPIPRequestService，在配置时可配置的参数有以下设置：

配置参数名称	配置参数含义
requestDataFormat	请求报文格式定义（如果是序列化服务，则不需要配置该参数） Format 可以是字符串格式、xml 格式或 8583 格式，或其他任意一种扩展实现的格式定义
responseDataFormat	响应报文格式定义（如果是序列化服务，则不需要配置该参数） Format 可以是字符串格式、xml 格式或 8583 格式，或其他任意一种扩展实现的格式定义
serviceType	服务类型，默认为 normal，可选值： session：建立 Session 的服务 endSession：结束 Session 的服务 normal：一般业务服务
checkSession	服务是否需要检查 session 是否建立，默认为 true
bizId	需要执行的 EMPBizLogic id
opId	执行的 operation id

inputModelUpdater	输入模型更新器。如果不定义的话，则使用缺省的输入模型更新器。如果业务逻辑中有 Input 定义，则以 Input 定义为准，根据更新规则进行数据更新；如果业务逻辑中没有 Input 定义，则以输入请求的数据为准，根据更新规则进行数据更新。
outputModelUpdater	输出模型更新器。如果不定义的话，则使用缺省的输出模型更新器。业务逻辑中未定义 output，也未自定义更新规则，则按照业务逻辑的数据模型更新表现层的数据模型；业务逻辑中未定义 output，但渠道接入部分中定义了更新规则，则按照更新规则更新；业务逻辑中定义了 output，则按照 output 定义的数据更新

输入模型更新器 TCPIPInputModelUpdater 中可配置的参数有：

配置参数名称	配置参数含义
updateList	子元素，输入更新规则列表

输入更新列表 updateList 中定义了多条更新规则 updateRule。更新规则 updateRule 中可配置的参数有：

配置参数名称	配置参数含义
fromId	源数据 ID，访问层数据模型中的数据域的 ID
toId	目标数据 ID，业务逻辑层数据模型中的数据域 ID
location	访问层的数据域的位置，可选值： request，来自于请求数据对象中； session，来自于会话数据中。 缺省值为 request
dataType	数据类型，用以进行数据校验和数据转换，缺省为 null

如果 EMPTCPIPRequestService 中未定义 TCPIPInputModelUpdater 或 TCPIPInputModelUpdater 中的 updateList 为空，则系统会提供缺省的更新规则，即 fromId 和 toId 相同，都为需要更新的数据域的 ID；location 为 request。

输出模型更新器 TCPIPOutputModelUpdater 中可配置的参数有：定义了属性更新会话的条件表达式输出更新规则列表 updaterList。

配置参数名称	配置参数含义
updateSessionCondition	属性，判断是否更新会话数据的表达式。可以不定义。同 EMP 表达式。 如果定义了该表达式，则只在表达式结果为 true 时，才可能更新会话数据。
updaterList	子元素，输出更新规则列表

输出更新规则列表中定义了多条更新规则 updateRule。更新规则 updateRule 中可配置的参数有：

配置参数名称	配置参数含义
fromId	源数据 ID，业务逻辑层数据模型中的数据域的 ID
toId	目标数据 ID，访问层数据模型中的数据域 ID
location	要更新的访问层的数据域的位置，可选值： request，更新到请求数据对象中； session，更新到会话数据中。 缺省值为 request
dataType	数据类型，用以进行数据转换，缺省为 null
opType	对 session 数据操作类型，缺省为 0，可选值： 0，追加操作，如果会话中没有待更新数据，则增加该数据，如果会话中的数据为 IndexedCollection（iColl），而该数据类型为 KeyedCollection（kColl），则把 kColl 追加到 iColl 中； 1，普通的复制操作； 2，删除会话中的数据。

如果 EMPTCPIPRequestService 中未定义 TCPIPOutputModelUpdater 或 TCPIPOutputModelUpdater 中的 updateList 为空，则系统会提供缺省的更新规则，即 fromId 和 toId 相同，都为需要更新的数据域的 ID；location 为 request。

一个 TCPIPService 的配置示例，如下所示：

```
<TCPIPService id="signOn" bizId="testEMPLogic" serviceType="session" opId="signOn">
    <requestDataFormat    id="headReqFmt" class="com.ecc.emp.format.FormatElement">
        <record>
            <fString dataName="userId">
                <delim delimChar="|"/>
            </fString>
            <fString dataName="password">
                <delim delimChar="|"/>
            </fString>
            <delim delimChar="#"/>
        </record>
    </requestDataFormat >

    <responseDataFormat    id="headRepFmt" class="com.ecc.emp.format.FormatElement">
        <record>
            <fString dataName="errorMsg">
```

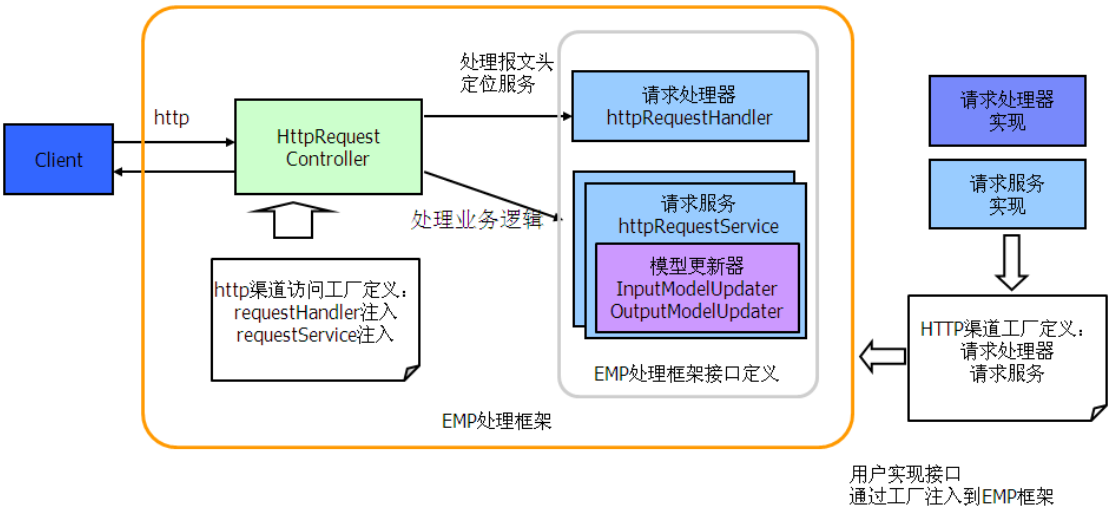
```
                    <delim delimChar="|"/>
                </fString>
                <fString dataName="name">
                    <delim delimChar="|"/>
                </fString>
                <delim delimChar="#"/>
            </record>
        </responseDataFormat    >

        <inputModelUpdater>
            <updateList>
                <updateRule toId="userId" location="request" fromId="userId"/>
                <updateRule toId="password" location="request" fromId="password"/>
            </updateList>
        </inputModelUpdater>

        <outputModelUpdater updateSessionCondition="!@isNull($name)">
            <updateList>
                <updateRule toId="name" location="session" fromId="name"/>
            </updateList>
        </outputModelUpdater>
</TCPIPService>
```

将需要配置到 TCPIP 渠道的业务逻辑构件进行分析后，配置到 TCPIPServiceContext.xml 文件中。即完成了一个业务发布到渠道上的操作。

6.4. HTTP访问

6.4.1. HTTP渠道访问的实现

HTTP 渠道访问组件与 TCPIP 渠道访问组件的处理相似，http 协议也同样有 TCPIP 协议一样的灵活性，用户的组报要求和规范是非常个性化的，在具体应用中都需要我们能够提供足够的可扩展能力。



如上图，为 EMP 平台提供的 HTTP 协议接入的设计类图。与 TCPIP 相似，我们在进行 HTTP 协议接入时，EMPHttpAccessServlet 类（HttpRequestController）是该渠道的渠道接入控制器，负责系统初始化和配置参数的读入，并实例化相关的组件，如渠道请求处理器 HttpRequestHandler，请求业务逻辑处理服务 HttpRequestService。

接口 HttpRequestHandler 是 HTTP 请求处理接口，用于处理 HTTP 请求的报文头，以及从报文头获取必要的信息。它的接口方法定义如下：

Interface	HttpRequestHandler	Package	com.ecc.emp.access.http
Interface Format	public Interface HttpRequestHandler		
Implements			

Name	Parameters	Return Value	Exceptions	Description
getRequestServiceName	HttpServletRequest request	String		从 request 中获得 Service id。
getSessionId	HttpServletRequest request	String		取得请求的 SessionId，如果是无 Session 的请求，则返回 null。
parseRequest	HttpServletRequest request	void	Exception	从请求报文中解出报文头，从中取得 Session id 和 Service id，连同报文体一起放入 request 以进行进一步处理。

handleResponse	HttpServletRequest request, HttpServletResponse response, String retMsg 返回的报文体, String reqURI 请求 URL, String sessionId 会话 ID	void		处理正常的响应报文。 将数据打包成返回报文头，并和返回报文体一起放入 response。
handleException	HttpServletRequest request, HttpServletResponse response, Exception e 异常, String reqURI 请求 URL, String sessionId 会话 ID	void		处理异常时的响应报文。 将数据打包成返回报文头，并和异常信息一起放入 response。

接口 HttpRequestService 是 Http 业务逻辑处理服务接口，它的接口定义方法为：

Interface	HttpRequestService	Package	com.ecc.emp.access.http
Interface Format	public Interface HttpRequestService		
Implements			
Name	Parameters	Return Value	Exceptions	Description
getServiceName		String		获得 Http 渠道业务逻辑处理服务名称
handleRequest	HttpServletRequest request, HttpServletResponse response, Context sessionContext, String sessionId	String	EMPException	具体的请求处理实现方法。返回打包后的报文体。
setComponentFactory	ComponentFactory factory			注入组件工厂
isSessionService		boolean		判断请求是否需要创建 session
isEndSessionService		boolean		判断请求是否要销毁 session

isCheckSession		boolean		判断该请求是否检查会话

HTTP 渠道业务逻辑处理服务实现类中会使用到输入、输出模型更新器对象。

接口 InputModelUpdater 定义了输入模型更新器，负责把数据从渠道访问层数据模型更新到业务逻辑处理层数据模型中，并在更新过程中根据数据类型进行数据校验和数据转换。它的接口方法定义如下：

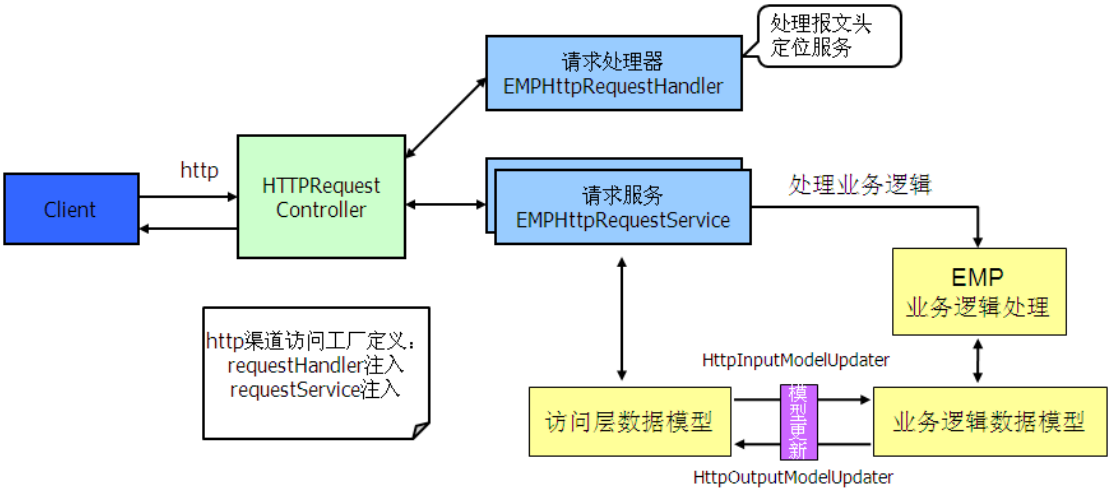
Interface	InputModelUpdater	Package	com.ecc.emp.access.http.modelUpdater
Interface Format	public Interface InputModelUpdater		
Implements			
Name	Parameters	Return Value	Exceptions	Description
updateModel	Object requestData 访问层数据模型, Context opContext 业务逻辑数据模型, KeyedCollection input 业务逻辑中的输入定义, Map dataTypeDefines EMP 中的数据类型集合		EMPException	将数据按照业务逻辑构件定义更新到业务逻辑层数据模型 Context 中

接口 OutputModelUpdter 定义了输出数据模型更新器，在业务逻辑处理完成之后，负责把数据从业务逻辑层数据模型中更新到访问层数据模型中，并在更新过程中按照数据类型进行数据转换。它的接口方法定义如下：

Interface	OutputModelUpdater	Package	com.ecc.emp.access.http.modelUpdater
Interface Format	public Interface OutputModelUpdater		
Implements			
Name	Parameters	Return Value	Exceptions	Description

updateModel	Object responseData 访问层数据模型, Context opContext 业务逻辑数据模型, KeyedCollection outputData 业务逻辑中的输出数据, HttpServletRequest request, Map dataTypeDefines EMP 中的数据类型集合		EMPException	将数据按照业务逻辑输出定义更新到访问层响应数据模型中

6.4.2. EMP实现的HTTP渠道接入处理



如上图所示，一个客户端 http 请求到达渠道处理端时，Http 渠道接入控制器调用请求处理器（EMPHttpRequestHandler）去处理报文头，通过报文头解析定位服务。Http 渠道控制器通过调用所请求的渠道业务逻辑处理服务对象（EMPHttpRequestService）来执行一个 EMP 业务逻辑处理。渠道业务逻辑处理服务对象（EMPHttpRequestService）通过输入、输出模型更新器（HttpInputModelUpdater、HttpOutputModelUpdater）与业务逻辑层数据模型之间进行数据交换。

EMP 提供的 HTTP 渠道接入控制器是由 **EMPHttpAccessServlet 实现的。**

6.4.3. HTTP渠道使用说明

6.4.3.1. 建立和配置HTTP渠道

类似于建立和配置 TCPIP 渠道，将系统提供的 HTTP 渠道建立的配置文件。（按照标准的 Servlet 对象定义方式进行定义）

察看 Servlet 定义的内容并进行参数有效性的确认，如下为 Servlet 对象的参数配置内容：

```
<servlet>
  <servlet-name> EMPHttpAccessServlet</servlet-name>
  <servlet-class>com.ecc.emp.access.http.EMPHttpAccessServlet</servlet-class>
  <init-param>
   <param-name>factoryName</param-name>
   <param-value>DemoBiz</param-value>
  </init-param>
  <init-param>
     <param-name>iniFile</param-name>
     <param-value>WEB-INF/bizs/DemoBiz/settings.xml</param-value>
  </init-param>
  <init-param>
   <param-name>servletContextFile</param-name>
   <param-value>WEB-INF/channels/httpAccessServletContext.xml</param-value>
  </init-param>
  <init-param>
   <param-name>sessionIdField</param-name>
   <param-value>sessionId</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
 </servlet>
```

参数及其含义，以及对该参数可用的修改如下表所示：

参数名	含义	示例	修改参考
EMPHttpAccessServlet	启动的 Servlet 对象类名称	**com.ecc.emp.access.http.EMPHttpAccessServlet**	不用修改。
factoryName	为你的渠道应用所访问的业务逻辑组件所在的业务工厂名称	**DemoBiz**	与你所采用的业务逻辑分组定义时所采用的名称相同。如在建立业务逻辑分组时定义名称为“testBiz”，则该处应改为“testBiz”。
iniFile	业务逻辑处理组件工厂文件定义	**WEB-INF/httpAccessServl**	这个参数非必须参数，如果没有，则通过 **factoryName** 共享

		etContext.xml	公共工厂定义
servletContextFile	渠道信息配置文件。该文件中定义渠道通信参数和部署到该渠道上的业务逻辑构件的索引	**WEB-INF/channels/ httpAccessServletContext.xml**	在生成应用时，系统已经提供了该渠道的配置文件基础内容，与该文件名相符。不用修改该参数。该文件内容可根据渠道定义相关属性进行修改。
sessionIdField	会话 ID 存放的字段名称。缺省使用 sessionId	**sessionId**	系统缺省定义，不用修改
load-on-startup	Servlet 的启动顺序号	**2**	如果 TCPIP 渠道使用的组件工厂是共享别的 Servlet 创建的组件工厂，则此参数值必须比创建组件工厂的 Servlet 的启动顺序号大，确保 TCPIP 渠道控制器初始化时组件工厂已经创建。

6.4.3.2. 在HTTP渠道上快速发布业务

HTTP 渠道接入访问的模型包括：

渠道接入控制器（HttpRequestController）：

通过 EMP 平台提供的 EMPHttpAccessServlet 类实现，已经实现，只需要通过配置来实现。该类的实现了 http 渠道逻辑工厂定义的注入，是 http 渠道请求入口，通过请求处理器定位请求服务。

请求处理器（HttpRequestHandler）：

通信层协议处理以及定位具体的 http 请求服务。

请求业务逻辑处理服务（HttpRequestService）：

调用业务逻辑处理具体的 http 渠道请求，包括使用输入、输出模型更新器（InputModelUpdater、OutputModelUpdater）进行数据交换。

这些都是定义在渠道配置文件 httpAccessServletContext.xml 中。

EMP 实现的 HTTP 渠道接入处理器（EMPHttpRequestHandler）的配置：

```
<HttpRequestHandler name="empHTTPHandler" sessionIdField="sessionId" serviceIdField="serviceId" appendReqHead="true" appendRepHead="true" errorCodeField="errorCode" class="com.ecc.emp.access.http.EMPHttpRequestHandler">
    <RequestHeadFormat id="headReqFmt" class="com.ecc.emp.format.FormatElement">
        <record>
            <fString dataName="sessionId"><delim delimChar="|"/></fString>
            <fString dataName="serviceId"><delim delimChar="|"/></fString>
```

```
            <delim delimChar="#"/>
        </record>
    </RequestHeadFormat>
    <ResponseHeadFormat id="headRepFmt" class="com.ecc.emp.format.FormatElement">
        <record>
            <fString dataName="errorCode"><delim delimChar="|"/></fString>
            <fString dataName="sessionId"><delim delimChar="|"/></fString>
            <delim delimChar="#"/>
        </record>
    </ResponseHeadFormat>
    <Map name="serviceIdMap">
        <MapEntry key="signOn" value="signOn"/>
        <MapEntry key="transfer" value="transfer"/>
        <MapEntry key="signOff" value="signOff"/>
        <MapEntry key="signOff1" value="signOff"/>
    </Map>
</HttpRequestHandler>
```

请求处理器的可配置参数如下表所示：

配置参数名	配置参数含义
sessionIdField	报文头解开后的 SessionId 数据域，非必须设置
serviceIdField	报文头解开后的 ServiceId 数据域，必须设置
errorCodeField	错误码数据域定义，非必须设置
appendReqHead	在向 httpRequestService 传递数据时，是否需要截去请求报文头。 缺省为 true，可选值： true，截去请求报文头； false，不截去请求报文头。
appendRepHead	在处理返回报文时，是否需要附带报文头，如果需要，将使用 responseHeadFormat 格式化后附在报文头部。 缺省为 true，可选值： true，追加报文头； false，不追加报文头。
RequestHeadFormat	请求报文头格式定义
ResponseHeadFormat	响应报文头格式定义
serviceIdMap	请求的服务 id 到指定服务 ID 的对照表，允许将多个请求 ServiceId 映射到其他定义 ServiceId 中，用于交易分发

EMP 实现的 Http 业务逻辑处理服务（EMPHttpRequestService）的配置：

```
<HttpService id="signOn" bizId="testEMPLogic" serviceType="session" opId="signOn" >
    <requestDataFormat>
        <record>
            <fString dataName="userId">
                <delim delimChar="#"/>
            </fString>
```

```
            <fString dataName="password">
                <delim delimChar="#"/>
            </fString>
        </record>
    </requestDataFormat>

    <responseDataFormat>
        <record>
            <fString dataName="errorMsg">
                <delim delimChar="|"/>
            </fString>
            <fString dataName="name">
                <delim delimChar="|"/>
            </fString>
            <delim delimChar="#"/>
        </record>
    </responseDataFormat>

    <inputModelUpdater>
        <updateList>
            <updateRule toId="userId" location="request" fromId="userId"/>
            <updateRule toId="password" location="request" fromId="password"/>
        </updateList>
    </inputModelUpdater>

    <outputModelUpdater updateSessionCondition="!@isNull($name)">
        <updateList>
            <updateRule toId="name" location="session" fromId="name"/>
        </updateList>
    </outputModelUpdater>
</HttpService>
```

请求服务的可配置参数如下表所示：

配置参数名称	配置参数含义
requestDataFormat	请求报文格式定义（如果是序列化服务，则不需要配置该参数） Format 可以是字符串格式、xml 格式或 8583 格式，或其他任意一种扩展实现的格式定义
responseDataFormat	响应报文格式定义（如果是序列化服务，则不需要配置该参数） Format 可以是字符串格式、xml 格式或 8583 格式，或其他任意一种扩展实现的格式定义
serviceType	服务类型，默认为 normal，可选值： session：建立 Session 的服务

	endSession：结束 Session 的服务 normal：一般业务服务
checkSession	服务是否需要检查 session 是否建立，默认为 true
bizId	需要执行的 EMPBizLogic id
opId	执行的 operation id
inputModelUpdater	输入模型更新器。如果不定义的话，则使用缺省的输入模型更新器。如果业务逻辑中有 Input 定义，则以 Input 定义为准，根据更新规则进行数据更新；如果业务逻辑中没有 Input 定义，则以输入请求的数据为准，根据更新规则进行数据更新。
outputModelUpdater	输出模型更新器。如果不定义的话，则使用缺省的输出模型更新器。业务逻辑中未定义 output，也未自定义更新规则，则按照业务逻辑的数据模型更新表现层的数据模型；业务逻辑中未定义 output，但渠道接入部分中定义了更新规则，则按照更新规则更新；业务逻辑中定义了 output，则按照 output 定义的数据更新

至于 inputModelUpdater 和 outputModelUpdater 的配置同 EMPTCPIPRequestService 下的配置，请参考相关章节。

此外，EMP 提供的 HTTP 渠道接入框架中还提供了 EMP 的访问控制、Session 管理和监控处理的部分，方便用户实现以 http 通信方式的渠道接入处理。

7. EMP技术组件

7.1. 基础运算组件

7.1.1. DES、3DES等加密算法

7.1.2. SHA等散列算法

7.1.3. 非对称加解密

7.1.4. 校验码

7.2. 数据报文处理组件

格式处理组件的主要意义在于将一系列数据（通常在 Context 空间中保存）进行格式转换，最终生成为特定格式的数据包；或者将特定格式的数据包进行反向转换，将其释放到 EMP 的 Context 空间中。这种数据格式的转换在应用系统中的应用非常广泛，如在与其他系统进行数据交换时，需要将本系统的数据转换为其他系统访问接口所要求的报文格式。

本章节将主要讨论 EMP 是如何应用格式组件来完成这些功能的。

7.2.1. 工作原理

7.2.1.1. 格式组件所完成的功能

格式组件主要完成两大功能：

- 将 EMP Context 中的数据转化为其他系统需要的数据报文，如：ISO8583 报文、XML 报文等
- 将其他系统送来的数据报文，转化为 EMP Context 中的数据

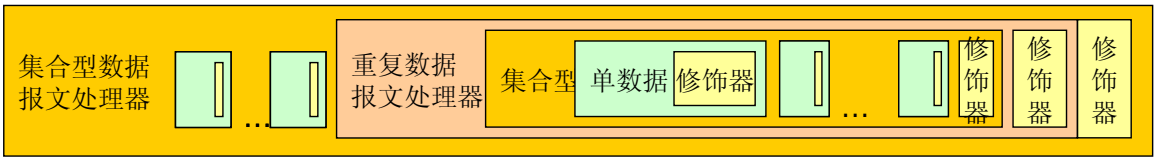
应当避免在格式组件的范围内进行逻辑运算（更理想的位置应该是在交易流程步骤中）。

7.2.1.2. 格式组件的实现

通过对报文处理功能进行抽象分析，形成类似下图的格式定义模式：

- 第一，对简单集合类型数据所定义的格式处理：

- 第二，对重复型数据所定义的格式处理：



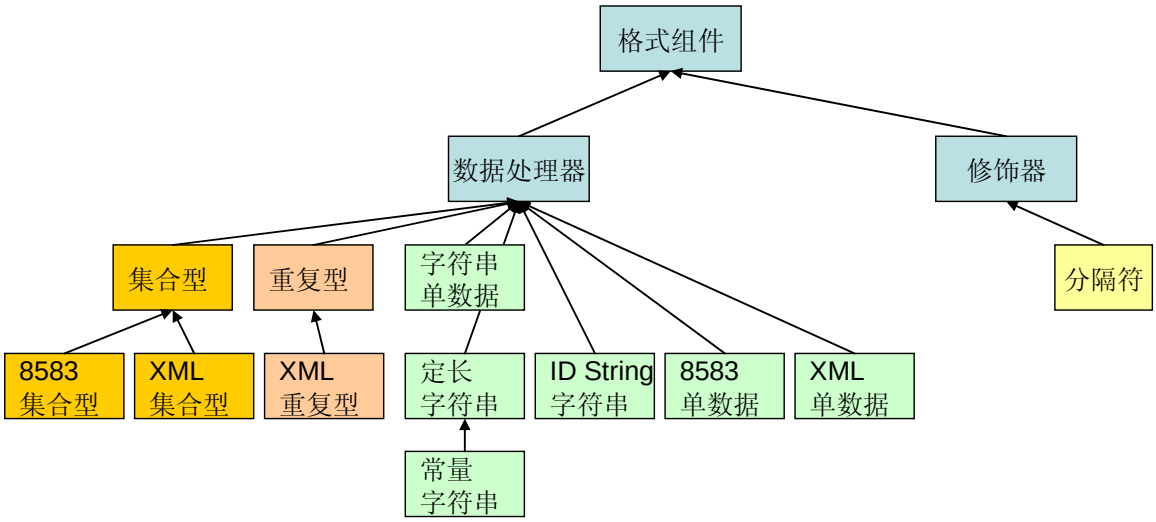
在 EMP 格式处理中，共有 4 种要素，分别是：

- 集合型数据报文处理器：集合型处理器对照 EMP 数据结构中的 kcoll 对象。一般

来说一个报文定义的顶级节点一定是一个集合型报文处理器，其下可以放置单数据报文处理组件、重复数据处理器、修饰器等。（在某些情况下，也可以嵌套放置集合型数据报文处理器，主要应用于 XML 报文处理）。

- 重复数据报文处理器：重复数据报文处理器对照 EMP 数据结构中的 icoll 对象。用于表述重复数据，其下可以放置集合型处理器和修饰器。
- 单数据报文处理组件：单数据报文处理组件对照 EMP 数据结构中的 datafield 对象，是对一个单独数据进行格式处理的组件，无论多么复杂的 format 定义，最终都是由单数据报文处理组件组成的。
- 修饰器：修饰器即对数据进行修饰的报文处理组件，例如分隔符，修饰的对象可以是上面三种中的任意一种。

EMP 提供了常用的报文格式处理组件，包括字符串处理、XML 格式处理和 8583 格式处理。格式处理组件的继承结构如下：



7.2.1.3. 报文结构的描述

在这种报文组件的实现的基础上，EMP 应用 XML 来描述报文结构。

格式定义应用一系列格式处理组件，可以将一组特定的数据从 EMP Context 中打包成一个特定的报文格式，也可以将一个特定的报文数据源，拆解到特定的 EMP Context 数据中去。

报文的定义类似如下格式：

```
<fmtDef id="gw1105SendFormat">
  <xmlWrap>
   <xmlHead version="1.0" encoding="gb2312"/>
   <xmlWrap tagName="ebank">
    <xmlIColl dataName="iAccountOverview">
     <xmlWrap tagName="kAccountOverview">
      <xmlFullTag dataName="accountNo"/>
      <xmlFullTag dataName="accountType"/>
     </xmlWrap>
    </xmlIColl>
   </xmlWrap>
  </xmlWrap>
```

```
</fmtDef>
```

7.2.2. 使用说明

在应用 EMP 格式组件定义一个报文时，首先需要确定，报文结构是什么样的，采用何种格式，如何标识一个数据，是否有特殊处理要求；然后结合 EMP 提供的格式处理组件，评估是否满足其需求，如果不满足，则需要进行格式组件的扩展。

7.2.2.1. Format的定义

1. 公共 Format

若某个格式定义要在多个业务逻辑处理构件或交易中使用，EMP 允许将这样的报文格式定义为公共 format（即在 formats.xml 文件中进行定义）。定义后，各个逻辑处理构件或交易如果需要使用这种公共报文定义，则只需在业务逻辑构建上进行引用定义即可，类似如下：

```
<EMPBusinessLogic id="demobiz" operationContext="demobizSrvCtx">
        <operation id="opname" name="交易名称">
         ……
        </operation>
        <refFormat name="formatid" refId=" formatid"/>
</EMPBusinessLogic>
```

注意，如不在业务逻辑构件中声明对公共报文定义的引用，将无法应用该公共报文定义。

2. 私有 Format

同样，即使在业务逻辑处理构件中定义的报文，使用前也需要声明对该报文定义的引用，类似如下：

```
<EMPBusinessLogic id="demobiz" operationContext="demobizSrvCtx">
        <operation id="op" name="交易名称">
            ......
        </operation>
        <refFormat name="selfFormatId" refId="selfFormatId"/>
            ......
        <refFormat name="commonFormatId" refId="commonFormatId"/>
    </EMPBusinessLogic>
    <context id="DemoBizSrvCtx" type="op">
    ......
```

```
<fmtDef id="selfFormatId">
    <record>
        <fString dataName="errorCode">
            <delim delimChar="|"/>
        </fString>
        <fString dataName="errorMsg">
            <delim delimChar="|"/>
        </fString>
    </record>
</fmtDef>
```

7.2.2.2. 程序中取得Format

在代码中（如交易步骤 Action 中）通过下边的方式，获取 Format 对象：

```
FormatElement aFormat =context.getFormat("formatName");
```

不推荐使用 EMPAction.getFormat 方法来获得 format 定义。

7.2.2.3. Format配置参数说明

类名	属性	允许内容
fmtDef com.ecc.emp.format.FormatElement 报文定义根节点	id	record<1>
record com.ecc.emp.format.KeyedFormat 报文定义数据集合	无	iColl<*>
iColl com.ecc.emp.format.IndexedFormat 重复型数据集合	dataName：对应的 icoll 的 id append:解包时是否将 icoll 中的内容清空(false 为清空，true 为附加)	record<1>
fString com.ecc.emp.format.String.StringFormat 普通字符串型报文处理组件	dataName:对应的 data field 的 id	delim<*> nullCheck<1>
nullCheck com.ecc.emp.format.String.NullCheck 空值检查	无	无
delim com.ecc.emp.format.String.Delimiter	delimChar：分隔符	无

分隔符修饰器		
FixedLenFormat com.ecc.emp.format.String.FixedLenFormat 定长字符串报文处理组件	aligment：居左、中、右 len：字段长度 padChar：填充符 dataName:对应的 data field 的 id	delim<*> nullCheck<1>
IDStringFormat com.ecc.emp.format.String.IDStringFormat 名值串报文处理组件	dataName:对应的 data field 的 id hasQuot：value 是否用引号括起来，默认为 false idName：id 名，如不填写，则用 dataName 代替	delim<*> nullCheck<1>
XMLWrapFormat com.ecc.emp.format.xml.XMLWrapFormat XML 报文定义数据集合	tagName:本节点的 XML 标签名 dataNameType:默认为 0；设置本属性后，如果其下的子元素没有设置 dataNameType，则将本属性值设定到子元素中。，该属性并不影响本节点的内容，而是对其子节点造成影响。 dataNameAttr:该属性并不影响本节点的内容，而是对其子节点造成影响。	XMLHeadTagFormat XMLIndexedFormat XMLFullTagFormat XMLTagFormat XMLConstantTagFormat
XMLHeadTagFormat com.ecc.emp.format.xml.XMLHeadTagFormat XML 报文定义头节点	version：默认为 1.0 encoding：默认为 gb2312	无
XMLIndexedFormat com.ecc.emp.format.xml.XMLIndexedFormat XML 报文定义重复型数据集合	tagName	XMLFullTagFormat XMLTagFormat XMLConstantTagFormat
XMLFullTagFormat com.ecc.emp.format.xml.XMLFullTagFormat 完整的 XML 域处理组件	tagName:本节点的 XML 标签名 dataName dataNameType:默认为-1；当设置为 2 时，解包数据域名为 dataNameAttr 所指向的属性名；1：使用 tagname；0：使用 dataname dataNameAttr:解包时使用，当 dataNameType 设定为使用 dataNameAttr 时,用于指定	无

	从哪个属性取值作为字段名； desc：描述； 其他属性：在该节点下定义任意属性，则都将出现在格式化结果中	
XMLTagFormat com.ecc.emp.format.xml.XMLTagFormat 空标签 XML 域处理组件	tagName:本节点的 XML 标签名 dataName dataNameType:默认为-1；当设置为 2 时，解包数据域名为 dataNameAttr 所指向的属性名；1：使用 tagname；0：使用 dataname dataNameAttr:解包时使用，当 dataNameType 设定为使用 dataNameAttr 时,用于指定从哪个属性取值作为字段名； desc：描述； 其他属性：在该节点下定义任意属性，则都将出现在格式化结果中	无
XMLConstantTagFormat com.ecc.emp.format.xml.XMLConstantTagFormat 常量 XML 域处理组件	tagName:本节点的 XML 标签名，没有设置或设置为“*”则将其格式化为“Unnamed” value：值，有下列保留字：TimeMillis;Time;Date;DateTime format：日期规格定义，满足 java SimpleDateFormat 的格式即可	无
ISO8583Format com.ecc.emp.format.ISO8583 8583 报文格式的根节点定义组件	codeSet：编码集，默认为 ASCII formatDefine：8583 报文字段属性定义类 msgType：报文类型	ISO8583Field

ISO8583Field com.ecc.emp.format.ISO8583.ISO8583Field	fieldIdx,域索引,使用域的编号(1-192) IOType,IO 类型,该域为接收或发送域,I=接收;O=发送;IO=发送/接收 fieldState,必需域,该域是否必需,M=必需;NotM=非必需; refFormat,引用格式,允许某些域的值通过 Format 的方式得到，此 refFmt 指向 Format 定义的 ID fieldType,域类型,该域的格式类型。L*VAR 代表变长域，前面的几个 L 代表几位长度数字；其他为定长域，区别在于对齐方式和填充字符；H 为二进制数据 fieldLength,长度,该域的固定长度 fieldValue,取值,该域的值，若无定义则使用数据域中的值 padchar,填充字符,该域在不足固定长度时的填充字符 alignment,对齐方式,该域在不足固定长度时的对齐方式,left;right;center; codeConvert,转换编码,是否需要转换编码 codeSet,编码,ASCII;EBCDIC;	

7.2.2.4. 字符串格式

7.2.2.4.1. 分隔符形式

3. 参考 1：

定义如”yucheng|12345678|”的字符串格式定义：

```
<fmtDef id="delimStringFmt">
    <record>
        <fString dataName="userid">
```

```
            <delim delimChar="|" />
        </fString>
        <fString dataName="password">
            <delim delimChar="|" />
        </fString>
    </record>
</fmtDef>
```

4. 参考 2：

定义如”yucheng|12345678|10000*0000*10001*1000*10002*2000*”的拥有多条重复数据记录的字符串格式定义：

```
<fmtDef id="delimStringICollFmt">
    <record>
        <fString dataName="userid">
            <delim delimChar="|" />
        </fString>
        <fString dataName="password">
            <delim delimChar="|" />
        </fString>
        <iColl dataName="accountList">
            <record>
                <fString dataName="accountNo">
                    <delim delimChar="*" />
                </fString>
                <fString dataName="balance">
                    <delim delimChar="*" />
                </fString>
            </record>
        </iColl>
    </record>
</fmtDef>
```

7.2.2.4.2. 定长格式

5. 参考 3：

定义如固长的字符串格式定义：

```
<fmtDef id="fixStringFmt">
    <record>
        <FixedLenFormat len="10" padChar=" " aligment="left"
            dataName="userid" />
        <FixedLenFormat len="10" padChar="*" aligment="right"
            dataName="userid" />
        <FixedLenFormat len="10" padChar="-" aligment="center"
```

```
            dataName="userid" />
        </record>
    </fmtDef>
```

6. **参考 4：**

定义如固长多条记录的字符串格式定义：

```
    <fmtDef id="fixStringICollFmt">
        <record>
            <FixedLenFormat len="10" padChar=" " aligment="left"
                dataName="userid" />
            <FixedLenFormat len="10" padChar="*" aligment="right"
                dataName="userid" />
            <FixedLenFormat len="10" padChar="-" aligment="center"
                dataName="userid" />
            <iColl dataName="accountList">
                <record>
                    <FixedLenFormat dataName="accountNo" len="10"
                        aligment="right" padChar="0" />
                    <FixedLenFormat dataName="balance" len="10"
                        aligment="right" padChar=" " />
                </record>
            </iColl>
        </record>
    </fmtDef>
```

7.2.2.4.3. 名值串格式

- **参考 5：**

定义名值串的字符串格式定义：

```
    <fmtDef id="idStringFmt">
        <record>
            <IDStringFormat hasQuot="true" idName="a" dataName="userid"
/>
            <IDStringFormat hasQuot="false" idName="b"
                dataName="password">
                <delim delimChar="*" />
            </IDStringFormat>
            <IDStringFormat hasQuot="false" idName="b"
                dataName="password" />
            <delim delimChar="|" />
        </record>
    </fmtDef>
```

7. **参考 6：**

定义拥有多条记录的名值串的字符串格式定义：

```
    <fmtDef id="idStringICollFmt">
        <record>
            <IDStringFormat hasQuot="true" idName="a" dataName="userid"
/>
            <IDStringFormat hasQuot="false" idName="b"
                dataName="password">
                <delim delimChar="*" />
            </IDStringFormat>
            <IDStringFormat hasQuot="false" idName="b"
                dataName="password" />
            <delim delimChar="|" />
            <iColl dataName="accountList">
                <record>
                    <IDStringFormat dataName="accountNo" hasQuot="true"
idName="accountNo"/>
                    <IDStringFormat dataName="balance" hasQuot="true"
idName="balance"/>
                </record>
            </iColl>
        </record>
    </fmtDef>
```

7.2.2.5. XML格式

➢ **参考 7：**

定义一般 XML 的字符串格式定义，本示例包括了 XML 包头、XML 常量处理、空标签的 XML 数据域、完整的 XML 数据域。

```
<fmtDef id="xmlStringFmt">
        <XMLWrapFormat tagName="echannels" dataNameType="1">
            <XMLHeadTagFormat version="1.0" encoding="GBK" />
            <XMLFullTagFormat tagName="a" dataName="userid" desc="11440"
w="test"/>
            <XMLFullTagFormat tagName="b" dataName="password" />
            <XMLTagFormat tagName="c" dataName="userid" desc="cfield"
                dataNameAttr="echannels" />
            <XMLConstantTagFormat tagName="d" value="abcdef" />
            <XMLConstantTagFormat tagName="e" value="Time" />
            <XMLConstantTagFormat tagName="f" value="Date" />
        </XMLWrapFormat>
    </fmtDef>
```

用上面 format 后的字符串效果如下：

```
<echannels><?xml version="1.0" encoding="GBK" ?><a desc="11440"
w="test">yucheng</a><b>12345678</b><c value="yucheng" desc="cfield"
```

```
dataNameAttr="echannels"/><d>abcdef</d><e>03:39:17</e><f>2007-07-20</
f></echannels>
```

- **参考 8：**

  定义多条及录 XML 的字符串格式定义。

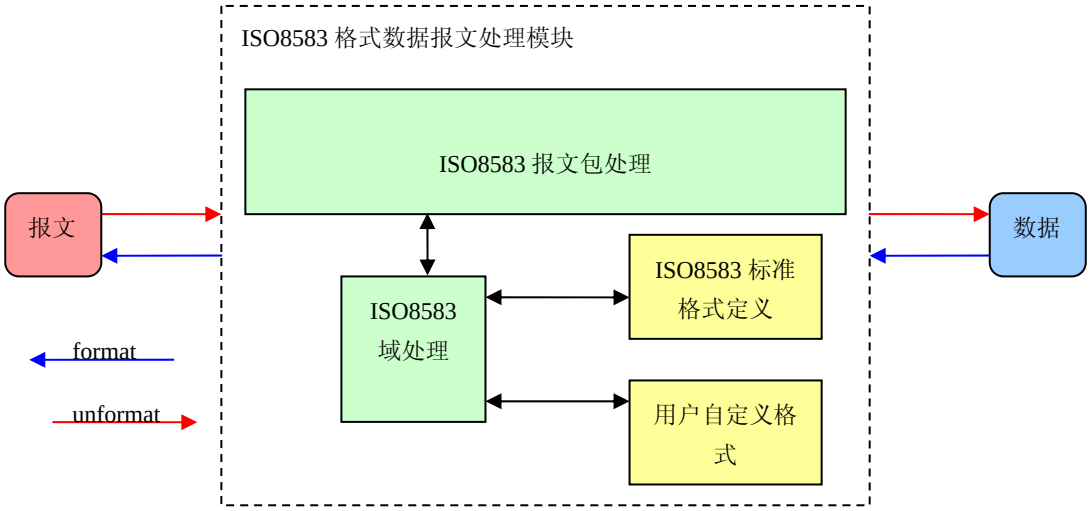
```
<fmtDef id="xmlStringICollFmt">
       <XMLWrapFormat dataNameAttr="name" dataNameType="1">
           <XMLHeadTagFormat version="1.0" encoding="GBK" />
           <XMLFullTagFormat tagName="a" dataName="userid" />
           <XMLFullTagFormat tagName="b" dataName="password" />
           <XMLTagFormat tagName="c" dataName="userid" desc="cfield"
               dataNameAttr="name" />
           <XMLConstantTagFormat tagName="d" value="abcdef" />
           <XMLConstantTagFormat tagName="e" value="Time" />
           <XMLConstantTagFormat tagName="f" value="Date" />
           <XMLIndexedFormat tagName="icoll" dataName="accountList">
               <XMLWrapFormat dataNameAttr="name" dataNameType="1">
                   <XMLFullTagFormat tagName="accountNo"
                       dataName="accountNo" />
                   <XMLFullTagFormat tagName="balance"
                       dataName="balance" />
               </XMLWrapFormat>
           </XMLIndexedFormat>
       </XMLWrapFormat>
   </fmtDef>
```

7.2.2.6. 8583 报文格式

ISO8583 格式数据报文处理模块可将 EMP 中定义的数据按照一定的规则打包成 ISO8583 格式二进制报文，或反过来将 ISO8583 格式二进制报文进行解包处理并逐个释放到 EMP 定义的数据域中。此外本模块还支持对 ISO8583 标准进行扩展，最大可用 192 个域，并允许用户对域进行自定义格式和长度。

7.2.2.6.1. 8583 报文处理功能

由下图可看到，ISO 格式数据报文处理模块分为 2 个部分，由 ISO8583 报文处理模块根据定义分别对每个域调用 ISO8583 域处理模块进行格式化和反格式化操作。根据外部 XML 定义，域处理模块会按照标准定义或用户自定义格式对报文进行处理。



ISO8583 格式报文处理模块主要有以下 3 个功能：

- **ISO8583 报文域处理：**

对 ISO8583 报文的某个域进行格式化和反格式化，其 value 与 EMP 数据对应。另外 value 也可以引用其他的 Format，此时要调用该 Format 的格式化（或反格式化）方法做深层处理。报文域的格式可以使用默认的标准参数，也可以自定义。

- **ISO8583 报文包处理：**

将所有域合成一个带有位图的完整 ISO8583 报文，或反过来进行分解，是 ISO8583 报文的处理入口。位图的生成和解析都在此进行。

- **用户自定义的扩展的 8583 定义：**

在某些场合，为了避免在 XML 配置文件中对每一域进行定义带来的麻烦和冗余，可以使用类继承的方式自行定义一整套扩展的 8583 定义。

7.2.2.6.2. EMP的 8583 报文处理实现

EMP 的 8583 报文处理实现在包 com.ecc.emp.format.iso8583 中。它通过实现 EMP 平台提供的 format 格式处理接口来实现的。

- **ISO8583Field**

可扩展的 ISO8583 报文域解析，支持自定义域类型和长度，支持域对其他 Format 的引用。从 com.ecc.emp.format.FormatField 继承。

它有三个关键方法：

Format 方法：将数据域格式化 ISO8583 域并返回。

Unformat 方法：对输入二进制串中下一个 ISO8583 域进行反格式化并将数据填回 EMP 数据域。

Extract 方法：返回从偏移量到下一个 ISO8583 域结束的长度。

- **ISO8583Format**

可扩展的 ISO8583 报文包解析，支持最多 192 个域。从 com.ecc.emp.format.KeyedFormat 继承。系统调用报文处理时的入口类。在该对象中包含通过配置定义的 ISO8583Field 对象，通过 IOS8583Format 对象的逐个调用完成打包和解包处理。

- **ISO8583Define**

ISO8583 标准格式定义。继承此类可以创建用户自定义类 ISO8583 格式。

重要的扩展类，通过扩展该类可以实现用户自定义的类 8583 处理。它主要定义了三个数组类型，分别用于标识 8583 的字段名、字段长度和字段类型。

属性名称	类型	描述
fieldNames	String[]	标准 ISO8583 定义
fieldLengths	int[]	标准 ISO8583 定义
fieldTypes	String[]	标准 ISO8583 定义

我们在定义自定义的 8583 类型时，需要在配置 ISO8583Format 属性时，定义 ISO8583Format 的“**formatDefine**”属性为自定义类的类名称。

7.2.2.6.3. 8583 配置说明

- **ISO8583Field**

参数名称	说明
fieldIdx	ISO8583 域编号（必需项，范围 1~192，其中 129~192 需自定义）
dataName	EMP 数据域名称（若无引用格式则为必需项）
IOType	输入输出类别（默认为 IO）
fieldState	是否必需项(M)
fieldType	自定义域类型：L*VAR(变长)，N，FN，AN，H(定长)
fieldLength	自定义域长度
fieldValue	自定义取值(常量)
padchar	定长域的填充字符（默认为空格）
alignment	定长域的对齐方式：left，right，center（默认为 left）
codeConvert	是否需要转码：true，false（默认为 false）
codeSet	要转换的编码类型(目前支持 EBCDIC)
refFormat	引用其他报文格式名

- **ISO8583Format**

参数名称	说明
codeSet	默认的编码类型（默认为 ASCII）
msgType	报文类型（默认为 0200）
formatDefine	自定义格式类名（必须写带完整包名的全名）

所有的 ISO8583Field 配置都要包含在一个 ISO8583Format 之内，并且不能嵌套 k/iColl。

7.3. 后台通信组件

7.3.1. TCPIP通讯组件

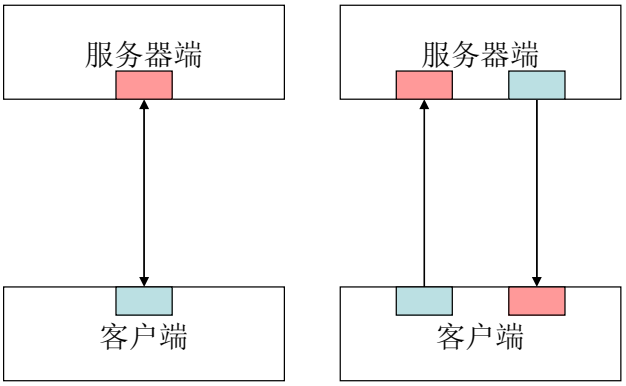
TCPIP 通讯组件是 EMP 的重要技术组件包，实现了 TCPIP 通讯的框架及扩展机制。在应用项目的开发过程中，使用该组件包，可以快速实现 TCPIP 通讯功能，通过配置和少量的编码即可实现各种通讯功能。

EMP 的 TCPIP 组件实现了 TCPIP 通讯的客户端逻辑和服务器端逻辑。在进行应用系统建设中，可以使用 TCPIP 组件来实现接出（本系统作为客户端，连接其他系统作为主机），也可以使用 TCPIP 组件来实现接入（本系统作为服务器提供业务逻辑服务，其他系统主动连接本系统）。

7.3.1.1. 工作原理

EMP 的 TCPIP 组件支持单双工通讯模式，支持长连接和短连接，并可以提供连接池功能。借助报文格式处理组件，实现通讯过程中的数据包组装和拆解。

7.3.1.1.1. 单工双工问题



双工　　单工

TCPIP 连接有多种模式，对于双工模式，一条连接既可以发送，又可以接收。对于单工模式，一条连接只能发送（不能接收），一条连接只能接收（不能发送）。所以，双工模式下，客户端只需要启动 socket 连接即可，无需启动监听端口，服务器端只需要启动监听端口，无需定义向客户端的连接；而单工模式下，客户端在启动 socket 连接的同时，还要启动端口监听，服务器端也要在定义端口监听的同时，定义客户端连接。

EMP 的 TCPIP 组件能够处理上面提到的双工和单工模式。

```
配置双工时，将 dual 属性设置为 true，单工时，设定为 false
<TCPIPService … dual=”true”…/>
或
<TCPIPServerService … dual=”true”…/>
```

7.3.1.1.2. 长连接短连接问题

TCPIP 连接，包括长连接和短连接方式。

EMP 的 TCPIP 组件支持这两种连接模式。

在 EMP 的配置中，按照下面的方式定义长短连接方式：

```
<TCPIPService … keepAlive=”true”…/>
或
<TCPIPServerService … keepAlive=”true”…/>
```

7.3.1.1.3. 通讯协议处理问题

在 TCPIP 连接中，通常会涉及到通讯协议报头的处理，例如在报文数据中，最前面几位为包长，在接收报文时要根据包长数据来进行数据接收。同样在发送报文时，也要生成包长，并放置在报文前面。

此外，TCPIP 通讯中，还存在着报文加密、报文转码等操作。这些操作与具体的业务逻辑处理无关，而是与通讯层数据包的约定有关。

EMP 的 TCPIP 组件提供了处理这些问题的扩展接口，com.ecc.emp.tcpip.CommProcessor。在应用开发中，如何需要进行上述处理，则可以开发特定的代码，实现 CommProcessor 接口，并将其配置在 TCPIPService 或 TCPIPServerService 上。

该接口实现了提供两个接口方法：

- readPackage
- wrapMessagePackage

用这两个方法，可以在读入数据包后（第一个动作），或发送数据包前（最后一个动作）时，对数据报文进行处理。

在 EMP 的配置文件中，按照下面的方式定义通讯协议处理类：

```
<TCPIPService … >
    ….
    <CommProcessor class=”…package.class”/>
</TCPIPService>
或
<TCPIPServerService … >
    ….
    <CommProcessor class=”…package.class”/>
</TCPIPServerService>
```

7.3.1.1.4. TCPIP客户端与负载均衡

在 TCPIP 应用中，客户端通常连接确定地址和端口的服务器，但是在大数量级应用中，客户端连接单一服务器可能造成该服务器压力过大，这时，可以通过部署多台服务器（或启动多个应用实例）来进行负载均衡。这种情况下，客户端需要平衡访问负担，轮流访问多台服务器。

EMP 的 TCPIP 组件（作为客户端应用时）能够定义多个主机连接，并轮流访问这些主机，将访问压力分散到多台服务器中。

在客户端的 TCPIPService 中，定义多个 ConnectToHost 即可：

```
<TCPIPService ……>
    <ConnectToHost … port=”…” hostAddr=”…” …/>
    …
    <ConnectToHost … port=”…” hostAddr=”…” …/>
</TCPIPService>
```

7.3.1.1.5. TCPIP客户端连接池

EMP 的 TCPIP 组件在客户端应用中，无论长短连接，都提供连接池功能。

对于长连接，连接池中连接对象的 socket 连接持续保持；

对于短连接，连接池中连接对象每次发送的时候进行连接，发送后关闭连接，但是对象仍然保持。

连接池的默认大小为 10 个连接，该数字可以设定。

```
<TCPIPService …>
  < ConnectToHost maxConnection=”20” …/>
  …
</TCPIPService>
```

7.3.1.1.6. TCPIP服务器端线程池

EMP 的 TCPIP 组件在服务器端应用中，无论长短连接，可以提供线程池功能（有属性开关）。

提供线程池机制主要是考虑到在一些操作系统中，新启动线程将消耗很多系统资源（例如使用进程来模拟线程），所以将线程对象缓存起来将保证系统的性能。

对于长连接，线程池中的对象….还不确定，待补充

对于短连接，线程池中的线程每次接到连接后，进行相关业务逻辑处理，返回消息后，将该连接关闭，但是线程对象仍然存在（进入 wait 状态），等待下一个连接处理（notify）。

线程池的开关是可以设置的，线程池的大小也可以设置：

```
<ListenPort … poolThread=”true” maxConnection=”20” …/>
```

7.3.1.1.7. EMP通讯组件的要素定义关系

对于客户端应用，EMP 提供接出交易步骤（TCPIP）和 TCPIP 接出服务（TCPIPService）。在交易流程中定义交易步骤，并在交易步骤定义中指向特定的 TCPIP 接出服务。

在接出服务下，可以定义若干个 ConnectToHost 标签，用于定义所要访问的主机地址、端口等；可以定义若干个 ListenPort，用于定义单工情况下，客户端的接收端口；还可以定义一个通讯协议处理器，用于指向处理通讯协议的代码类。即下面的形式：

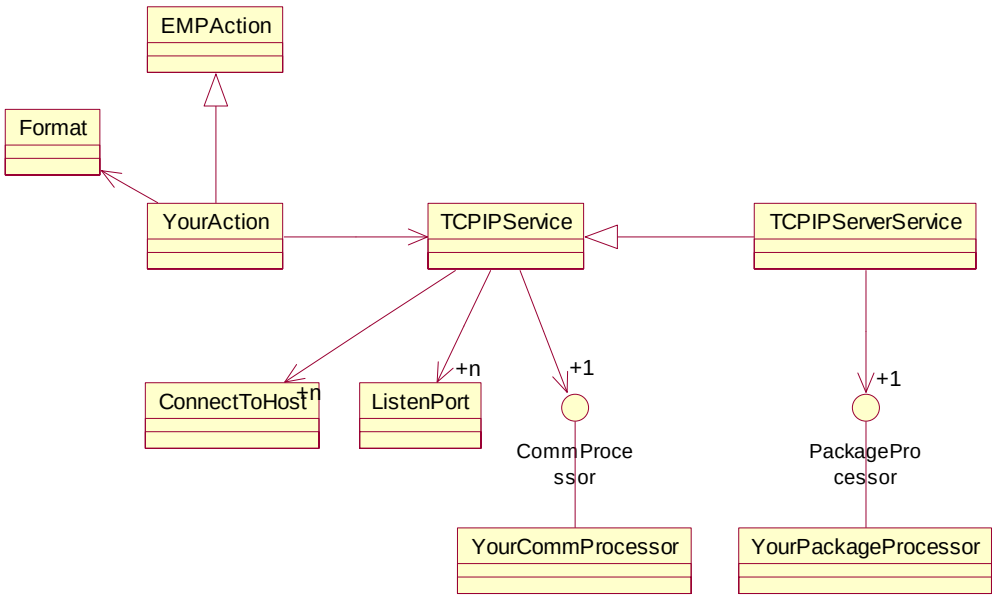
```
服务定义结构如下：
<TCPIPService…>
    <ListenPort …/>
    …
    <ListenPort …/>
    <ConnectToHost …/>
    …
    <ConnectToHost …/>
    <CommProcessor …/>
</TCPIPService>
```

对于服务器端应用，EMP 提供 TCPIP 接入服务（TCPIPServerService），以及接入处理扩展接口（PackageProcessor）。

接入服务的定义格式与上面提到的接出服务的定义格式是相同的，也可以定义 ListenPort 标签用于定义接入端口，定义 ConnectToHost 标签用于定义单工模式下的向客户端发送返回报文的连接，定义 CommProcessor 用于定义通讯协议处理器扩展类。此外，还可以定义 PackageProcessor，用于定义收报后的业务逻辑处理。

```
服务定义结构如下：
<TCPIPServerService…>
    <ListenPort …/>
    …
    <ListenPort …/>
    <ConnectToHost …/>
    …
    <ConnectToHost …/>
    <CommProcessor …/>
    <PackageProcessor …/>
</TCPIPServerService>
```

7.3.1.1.8. EMP TCPIP通讯组件类关系说明



7.3.1.2. TCPIP组件的使用

7.3.1.2.1. Action和Service属性

属性名	属性描述	属性类型
TCPIPServerService		
keepAlive	长短链接标志	boolean
dual	单双工标志	boolean
ListenPort		
port	本地端口	int
poolThread	线程池启用标志	boolean
maxConnection	线程池最大链接数	int
TCPIPService		
keepAlive	长短链接标志	boolean
dual	单双工标志	boolean
ConnectToHost		
hostAddr	主机地址	string
maxConnection	最大链接数	int
port	主机端口	int
CommProcessor		
class	对应的类名	class
PackageProcessor		

class	对应的类名	class

其中：

TCPIPServerService 下，可以定义 ListenPort（一到多个）、ConnectToHost（一到多个，待确定）、CommProcessor（一个且必须有一个）作为子元素，PackageProcessor（一个且必须有一个）作为子元素。

TCPIPService 下，可以定义 ListenPort（一到多个）、ConnectToHost（一到多个）、CommProcessor（一个且必须有一个）作为子元素。

7.3.1.2.2. TCPIP组件的XML配置（客户端）

在交易定义的XML配置中使用TCPIP组件作为后台连接器，则需要在交易流程定义中，定义特定的 TCPIP 访问 Action 组件。在 EMP 中提供了默认的 TCPIP 访问 Action（com.ecc.emp.tcpip.TCPIPAccessAction），

此外，还要定义 TCPIPService 服务（在 businesslogic 文件中或公共的 service.xml 文件中）。

参考定义：在下面的例子中，定义了一个双工、短连接的 TCPIP 服务，轮流访问后台两台主机，最大连接数都为 30，使用的通讯协议处理器为 com.ecc.emp.tcpip.EMPCommProcessor。

```
交易步骤定义按照如下：
<action id="TCPIPAccessAction0"
 implClass="com.ecc.emp.tcpip.TCPIPAccessAction" timeOut="10000"
 sendFormatName="sendFormat" serviceName=" aService "
 receiveFormatName="receFormat" >
      <transition …/>
</action>
服务定义按照如下：
<TCPIPService name="aService" keepAlive="false" dual="true"
       class="com.ecc.emp.tcpip.TCPIPServerService">
       <ConnectToHost port="12345" maxConnection="30"
    hostAddr="192.168.0.6" class="com.ecc.emp.tcpip.ConnectToHost" />
        <ConnectToHost port="12345" maxConnection="30"
    hostAddr="192.168.0.7" class="com.ecc.emp.tcpip.ConnectToHost" />
       <CommProcessor name="commProcessor"
          class="com.ecc.emp.tcpip.EMPCommProcessor" />
</TCPIPService>
```

7.3.1.2.3. 直接使用编程调用（客户端）

除了可以在交易中通过配置来进行 TCPIP 访问之外，还可以通过编码来调用 TCPIP 组件访问后台主机。编码类似如下：

下面编码实现了 TCPIP 短连接、双工、连接一个后台主机，最大连接数 20 个，这样一个连接场景。

```
        try {
            TCPIPService service = new TCPIPService();
            service.setKeepAlive(false);
            service.setDual(true);
            ConnectToHost connectToHost = new ConnectToHost();
            connectToHost.setHostAddr("127.0.0.1");
            connectToHost.setPort(12345);
            connectToHost.setMaxConnection(20);
            byte[] result = service.sendAndWait(null, getRequestData()
                    .getBytes(), 10000);
            service.addConnectToHost(connectToHost);
            service.setCommProcessor(new EMPCommProcessor());
            service.initialize();
        } catch (Exception e) {
            e.printStackTrace();
        }
```

7.3.1.2.4. TCPIP组件的XML配置（服务器端）

在服务器端，在服务定义或相关渠道访问定义中，可以定义 TCPIP 服务的访问方式如下：

在下面这段 XML 配置中，实现了短连接、双工，启动了一个监听端口（12345），该端口进行线程池管理，最大连接数为 30 个。

```
    <TCPIPServerService name="SRVService" keepAlive="false" dual="true"
        class="com.ecc.emp.tcpip.TCPIPServerService">
        <ListenPort keepAlive="false" port="12345" maxConnection="30"
            poolThread="true" class="com.ecc.emp.tcpip.ListenPort" />
        <CommProcessor name="commProcessor"
            class="com.ecc.emp.tcpip.EMPCommProcessor" />
    </TCPIPServerService>
```

7.3.1.3. TCPIP组件的扩展

7.3.1.3.1. 扩展CommProcessor

实现 com.ecc.emp.tcpip.CommProcessor 接口，实现下面两个方法：

byte[] readPackage(java.io.InputStream in) throws IOException, EMPException;

从输入流中读入一个标准数据包，它需要处理通信协议

byte[] wrapMessagePackage(byte[] msg);

根据通信协议对数据包进行打包，如加入通信报文头

7.3.1.3.2. 实现PackageProcessor（服务器端）

实现 com.ecc.emp.tcpip.PackageProcessor 接口，实现下面两个方法：

public byte[] processNewPackage(byte[] msg, TCPIPService service, Socket socket)throws EMPException;
接收到新数据包处理接口，当通信服务接收到新的数据包时，调用此接口，如果是请求报文则此接口处理数据请求报文，返回 null 否则原包返回。其中 OutputStream 是当通信服务是双工通信时的响应输出流

public boolean isRequestPackage(byte[] msg);

7.3.1.3.3. 扩展Action（客户端）

继承 com..ecc.emp.flow.EMPAction 基类即可，实现下面方法：

public String execute(Context context) throws EMPException

在客户端连接器的交易步骤中，通常要完成以下几项操作：

- 取得 TCPIP 通讯服务；
- 调用报文处理组件，将交易数据进行打包
- 调用 TCPIP 通讯服务发送交易报文，并接收返回报文
- 调用报文处理组件，将交易数据进行解包

7.3.2. HTTP通讯组件

HTTP 通讯组件的主要功能是，提供直接或间接的 HTTP 通信处理，可用于与其他系统的 HTTP 网关进行交互。

7.3.2.1. 工作原理

7.3.2.1.1. 要素定义关系

在 EMP HTTP 组件的使用中，共有三种要素需要定义：

- 交易步骤（HttpCommAction）：在 HttpCommAction 中定义了通信组件与交易数据之间的关联关系。
- 通信服务（HttpResource）：定义 HTTP 通信的若干属性，包括地址、代理服务器设置、最大连接数等全部属性值。
- 通信定义（HttpCommService）：HttpCommService 需要定义在一个 HttpResource 下，可以定义其特定的访问属性。

在 HttpResource 下，可以不定义 HttpCommService，这时组件均使用 HttpResource 的访问设置。

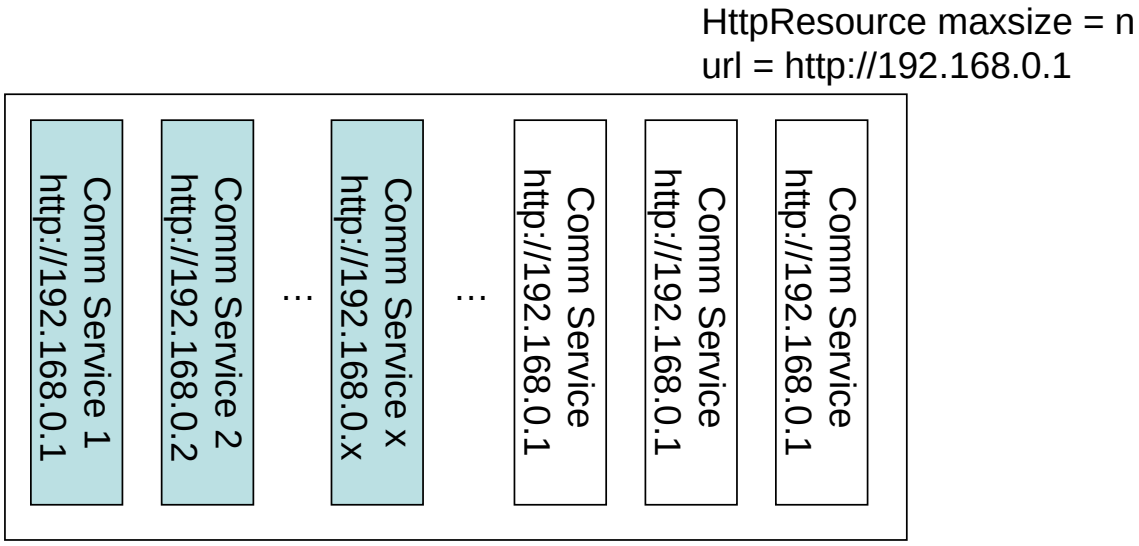
7.3.2.1.2. HTTP代理

EMP HTTP 通讯组件可以透过代理服务器与其他系统的 HTTP 网关交互。在需要透过代理时，需要为该访问定义设置如下属性：

- 使用代理服务器标志（useProxyAuthor）
- 代理服务器地址（reqProxyIP）
- 代理服务器端口（reqProxyPort）
- 代理服务器用户名（proxyUserName）
- 代理服务器用户口令（proxyUserPass）

当设置了使用代理服务器标识后，系统将根据后续四个属性的设定进行访问。如果不设置代理服务器标识，则系统将忽略后续四个属性的设置。

7.3.2.1.3. 连接策略说明



在一个 HttpResource 中，可以定义若干个 HttpCommService 定义，在每个 HttpCommService 中可以定义特定的链接参数。

上图描述的情况是，在一个 HttpResource 中，通过 XML 文件，定义了三个 HttpCommService，并且分别设定了不同的 url 属性，且在 HttpResource 中，也设定了其访问参数。

当使用该服务时，并发访问情况下，将按照下面的策略进行处理：

- 当有调用请求时，首先从第一个 HttpCommService 开始检查是否被占用，如果没有被占用，则使用该 comm service。如果已经被占用，则顺序检查下一个，直至当前 HttpResource 中的 comm service 全部检查一遍为止。
- 当将目前 HttpResource 中的 comm service 检查一遍，且没有找到可用的 comm service 时，则会判断当前的 comm service 是否超出了 resource 中设定的并发访问最大值，如果没有达到最大值，则创建一个新的 comm service（注意，这时，该 comm service 的链接属性从 http resource 的属性来复制），并使用这个新建的 comm service；如果达到了最大值，则返回错误信息。

7.3.2.2. 使用说明

7.3.2.2.1. Action和Service的属性

在使用 EMP HTTP 通信组件时，首先要定义服务，然后定义 Action 即可。

- HttpCommAction 的属性有：
  - String msgField ：发送数据域，通信时，需要发送的请求内容的存储数据域名称。如果参数 msgField 没有配置，则使用 sendFormatName 设置的 format 来得到发送的内容，若两者都没设置，则表示不需要发送任何信息。 **数据域中的值的格式应该为 name=value|name=value|name=value**
  - String recMsgField：接收数据域，用于存储接收到的响应信息的数据域名称；
  - String sendFormatName ：发送 format 的名称，发送数据的打包格式 ，**如果 msgField 参数存在，则该参数无效**。
  - String receiveFormatName ：接收 format 的名称，接收到的数据的解包格式，**如果 recMsgField 参数存在，则该参数无效**。该参数的缺省配置为 hostReceiveFormat。
  - String dataName ：在 HttpCommService 服务中，若 httpURL 设置为数据域名称（即不是以 HTTP://开头），则在这里 dataName 设置为数据域所在的 Collection 名称。如 httpURL 设置为“fielda”，则在发送请求时，会从取 fielda 的值，并将其作为 http URL。如果不设置该参数，则直接从 context 中取值，如果设置了该参数，则从该参数所指定的 data Collection 中取值。
  - String serviceName：本 action 所使用的 http resource 的名称
  - int conTimeout ：通信超时时间（缺省为-1），设定为-1 时，则选用 http resource 中的超时设定。

如果要正常使用 HttpCommAction，则必须设置的参数包括：serviceName，发送数据域或发送 format 定义，接收数据域或接收 format 定义

- HttpResource 的属性有：
  - reqMethod：HTTP 请求方式，缺省设置为 POST
  - httpURL：Http 通信 URL 地址
  - reqProxyIP：代理 IP 地址（可选参数）

- reqProxyPort：代理端口（可选参数，缺省为-1）
- useProxyAuthor：代理权限模式（可选参数）
- proxyUserName：登陆代理用户名（可选参数）
- proxyUserPass：登陆密码（可选参数）
- connectTimeout：通信超时时间（ms）（可选参数，缺省为 4 秒）
- size：最大连接数（缺省为 10）

除最大连接数和超时时间之外，HTTP 的访问属性应该设置，否则，在并发访问时，新创建出的 comm service 将没有访问属性，从而会发生异常。

- HttpCommService 的属性有：
  - reqMethod：HTTP 请求方式，一般设置为 POST
  - reqProxyIP：代理 IP 地址
  - reqProxyPort：代理端口
  - useProxyAuthor：代理权限模式
  - proxyUserName：登陆代理用户名
  - proxyUserPass：登陆密码
  - httpURL：Http 通信 URL 地址，可以是诸如 Servlet、JSP 等等可以接收参数传递的 URL。该参数可以是一个合法的地址(以 Http://开头)，也可以是一个存储了合法 URL 的数据域名称
  - reqHead：可以设置多个 http 连接的头部信息（以;结束，每一个头部信息对的格式都是："头部信息名称：内容;"。如：Accept-Language:zh-cn;）

7.3.3. WebService访问（待完成）

7.3.4. MQ通信

MQ 通讯模块是用于实现与 IBM WebSphere MQ 消息服务器进行交互的模块，可将组织好的报文上送到指定 MQ 消息队列中，或从指定 MQ 消息队列中获取消息。

7.3.4.1. 工作原理

EMP MQ 通讯组件通过 com.ibm.mq 访问 MQ 服务的标准接口，实现消息数据包的传送

与接收。其中将所有的 MQ 连接参数抽象成 MQResource 类进行管理，将 MQ 队列管理器的管理封装为带资源池的 QManagerConnectionManager 类，与服务器交互的实现类抽象为 MQConnectionImp 类。

由于 MQ 具有异步传输的特点，EMP MQ 通讯组件将消息的发送和接收分别封装为了两组交易步骤：

- 消息发送步骤（MQSendtoAction）
- 消息接收步骤（MQReceiveAction）

他们通过调用统一的 MQConnectionPoolService 服务来进行实际的操作。用户可在 Service 中定义一个或多个 MQResource 连接。以发送消息为例，用户需要将发送消息以及连接编号事先存放到 context 的数据域中；当业务逻辑执行到 MQSendtoAction 时，该步骤会调用 MQConnectionPoolService 取得相应消息队列的连接，发送消息，并返回消息编号供后续处理（由于是异步发送，并不即时获得服务器响应消息）。接收消息的原理同样如此。

7.3.4.2. 配置说明

7.3.4.2.1. MQConnectionPoolService

MQ 连接池服务。发送和接收的 Action 都需依赖该服务。

子节点名称	说明
MQResource	MQ 连接参数定义，可有多个
QManagerConnectionManager	带有连接池的 MQQueueManager 连接管理器，可有多个

```
<MQConnectionPoolService id="MQConnectionPool"
        implClass="com.ecc.emp.comm.MQ.MQConnectionPoolService " >
        <MQResource ……/>
        <QManagerConnectionManager ……/>
</ MQConnectionPoolService>
```

7.3.4.2.2. MQResource

MQResource 作为 MQConnectionPoolService 的子节点定义。一个 Service 中可以包含多个 Resource，定义不同的队列连接属性。

参数名称	说明
resourceID	连接资源 ID，在每次执行 Action 时需指定资源 ID，代表使用哪个连接定义

sendToQ	mq 发送队列名称
replyToQ	mq 返回队列名称
qManagerName	Mq 连接管理器名称，通过该名称去找到 Resource 对应的 QManagerConnectionManager
getMessageOptions	获取消息选项(缺省为 MQC.MQGMO_WAIT)
putMessageOptions	发送消息选项(缺省为 MQC.MQPMO_NO_SYNCPOINT)
replyToQOptions	接收队列选项(缺省为 MQC.MQOO_INPUT_SHARED)
sendToQOptions	发送队列选项(缺省为 MQC.MQOO_OUTPUT)
timeOut	超时时间(单位毫秒，缺省是 1000，即 1 秒)

7.3.4.2.3. QManagerConnectionManager

QManagerConnectionManager 作为 MQConnectionPoolService 的子节点定义。一个 Service 中可以包含多个 QManagerConnectionManager，定义不同的连接管理器。

参数名称	说明
qmanagerName	管理器名称
hostName	主机地址
port	端口
channelName	渠道名
charSet	字符集
maxCons	该连接管理器允许的最大连接数

7.3.4.2.4. MQSendtoAction

参数名称	说明
packMsgGetFrom	MQ 消息的发送域
msgExpirty	消息时效(缺省为-1)
msgPersitent	消息持久性(缺省为 2)
msgPirority	优先级(缺省为-1)
MQResourceIdField	resourceId 的存放域(缺省为数据域 MQResourceId)
serviceName	MQConnectionPoolService 的服务名称(缺省为 MQConnectionPool)
MessageIdField	messageID 的存放域(缺省为数据域 MessageID)

```
<action id="MQSendtoAction" packMsgGetFrom="errorCode"
        implClass="com.ecc.emp.comm.MQ.MQSendtoAction">
</action>
```

7.3.4.2.5. MQReceiveAction

参数名称	说明

packMsgSaveTo	MQ 消息的接收域
MQResourceIdField	resourceId 的存放域(缺省为数据域 MQResourceId)
MessageIdField	messageID 的存放域(缺省为数据域 MessageID)
GWRetCode	GWRetCode 网关的存放域(缺省为数据域 GWRetCode)
serviceName	MQConnectionPoolService 的服务名称(缺省为 MQConnectionPool)
hostRetCode	hostRetCode 返回码的存放域(缺省为数据域 hostRetCode)

```
<action id="MQReceiveAction" packMsgSaveTo="dataName"
        implClass= "com.ecc.emp.comm.MQ.MQReceiveAction">
</action>
```

7.3.5. CICS通信

CICS (客户信息控制系统)是 IBM 系统/370 的一个产品，可支持一个包括若干终端和终端子系统的网络，提供一个面向事务处理的联机应用环境。它使用 DCE(分布式计算环境)的安全功能，并支持 TCP/IP 通信协议。

CICS 主机通讯处理模块是基于 EMP 平台应用之上的与 CICS 主机进行报文通讯的实现，具有连接缓冲池功能，提供可配置的 CICS 主机通讯处理功能。

7.3.5.1. 工作原理

EMP CICS 通讯组件通过 com.ibm.ctg.client 包完成 CICS 通讯的客户端实现。它将所有的 CICS 连接抽象成 CICSConnection 类，并通过带有连接缓冲池的 CICSResource 服务进行统一管理。提供 SendToCICSAction 操作步骤作为 CICS 访问的业务逻辑入口。

要进行 CICS 主机访问，首先需要定义 CICSResource 服务，在其中设定连接池的各项属性，以及地址、端口、用户名、密码等连接属性。之后便可在某个业务逻辑中调用 SendToCICSAction 操作步骤进行发送。SendToCICSAction 会首先用指定 Format 对 context 中的数据进行打包，然后将打包后的报文送给 CICSResource 服务。该服务会自动从连接池中取得连接进行发送，并等待服务器响应，得到响应后的报文返回给 Action。最后在 Action 中将报文解开，数据放回 context。

7.3.5.2. 配置说明

7.3.5.2.1. CICSResource

参数名称	说明
size	缓冲池当前尺寸
maxPoolSize	缓冲池最大尺寸（默认为 0，表示不限）
timeBetweenRetries	重试时间（默认 2000 毫秒）
spare	缓冲比例
cleanupTime	清理间隔时间（默认为 0，表示不启动清理线程）
gatewayName	网关地址（以下都是 CICSConnection 的参数）
gatewayPort	端口
serverName	服务器名
userId	用户名
password	密码
programName	请求程序名
synchronousMode	同步模式
commAreaLength	COMMAREA 大小
transactionId	运行程序的事务 ID
codePage	编码
automaticConnection	自动连接标志
establishSessionRetries	连接重试次数

7.3.5.2.2. SendToCICSAction

参数名称	说明
timeout	接收超时时间（默认为 60000 毫秒）
cicsResourceName	CICS Resource 标识，指定发送所用的 cics 连接
sendFormatName	发送数据打包格式化名称（默认为 hostSendFormat）
receiveFormatName	接收数据打包格式化名称（默认为 hostReceiveFormat）
succFlag	成功标志，若等于返回信息则发送成功

7.3.6. Tuxedo通信（待完成）

7.3.7. JCA访问（待完成）

7.3.8. LU0、LU6.2 通信（待完成）

7.4. 数据库访问组件

如今几乎所有的 Web 应用都离不开数据库的支持。JDBC 数据库访问组件就是 EMP 平台所提供的用于访问数据库服务器的技术组件。使用该组件，可以只通过简单的 XML 文件外部参数化配置，实现对数据库的各种操作，并能与 EMP 数据模型进行数据交换。此外该组件可与 EMP 提供的“独立于应用、基于声明的”事务管理机制相结合，由平台自动控制事务的提交和回滚。

EMP 的 JDBC 数据库访问组件包括相关交易步骤（Action）和服务（Service），提供了如下三种基本的访问方式：

- 表映射操作方式：即提供一张数据库表和 Context data 的映射，实现对单表的增、删、改、查操作，并提供分页查询功能，也可以进行简单的多表查询；
- 执行 SQL 方式：直接执行 SQL 语句，可批量执行多条，支持动态 SQL 和模糊查询；
- 存储过程方式：访问数据库服务器端已编写好的存储过程；

7.4.1. 数据源定义

EMP 的数据源作为一个服务存在，它是对数据库对象资源描述的一个封装。EMP 提供两种数据源的定义方式（服务）。

7.4.1.1. JDBC数据源

JDBC 数据源，通过定义用户名/密码、驱动程序类、数据库名等属性产生一个标准的 JDBCResource 对象。


```
<JDBCDriverDataSource implClass="com.ecc.emp.jdbc.JDBCDriverDataSource"
                      id="dataSource" userName="ebank" password="ebank"
                      driverName="org.apache.derby.jdbc.ClientDriver"
                      dbURL="jdbc:derby://localhost:1527/EBANK">
</JDBCDriverDataSource>
```

7.4.1.2. JNDI数据源

JNDI 数据源，通过 JNDI 寻址方式应用在应用服务器中配置好的的数据源定义。

JNDI 数据源是由应用服务器进行管理的，使用 JNDI 数据源时，只需要在 EMP 应用中指明所使用数据源的 JNDI 名称即可，而数据库的用户名、密码等参数由应用服务器进行管理。以下是 JNDI 数据源服务的定义方法：

```
<JNDIDataSource id="jndiDataSource" implClass="com.ecc.emp.jdbc.JNDIDataSource"
                jndiName="java:comp/env/jdbc/derby">
</JNDIDataSource>
```

7.4.1.3. 选用适当的数据源

EMP 平台所提供的以上两种数据源的定义方式对于应用来说是透明的，可以根据部署环境互换使用。一般来说，在开发环境下采用 JDBC 数据源，而在应用运行环境下采用 JNDI 数据源定义方式。

需要注意的是，EMP 本身并不提供数据库连接池机制，而是透过使用 JNDI 数据源，应用 J2EE 应用服务器提供的数据库连接池机制来保证应用系统的效率。而在应用 JDBC 数据源时，则无法靠数据库连接池来提高应用的效率。

注意：要正确取得数据库连接，需要将相应数据库的 JDBC 驱动程序包放在应用的 WebContent/WEB-INF/lib 目录下。

7.4.1.4. Tomcat中配置数据源参考

对于 Tomcat 5.0，在 server.xml 的当前 Web 应用的 context 定义下添加下列内容：

```
<Resource name="derby" auth="Container"
    type="javax.sql.DataSource">
</Resource>
```

```
<ResourceParams name="derby">
    <parameter>
        <name>url</name>
        <value>jdbc:derby://localhost:1527/EBANK</value>
    </parameter>
    <parameter>
        <name>driverClassName</name>
        <value>org.apache.derby.jdbc.ClientDriver</value>
    </parameter>
    <parameter>
        <name>username</name>
        <value>ebank</value>
    </parameter>
    <parameter>
        <name>password</name>
        <value>ebank</value>
    </parameter>
</ResourceParams>
```

对于 Tomcat 5.5，在 server.xml 的当前 Web 应用的 context 定义下添加下列内容：

```
<Resource name="derby" auth="Container" type="javax.sql.DataSource"
    driverClassName="org.apache.derby.jdbc.ClientDriver"
    url="jdbc:derby://localhost:1527/EBANK" username="ebank"
    password="ebank" maxActive="20" maxIdle="10" maxWait="10000" />
```

此外在 web.xml 中增加下列定义：

```
<resource-ref>
        <res-ref-name>derby</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
</resource-ref>
```

对于在 Tomcat 中配置数据源，还要注意将数据库驱动 jar 文件放置到相应文件夹下，如：Tomcat5\common\lib 目录。

7.4.1.5. 数据库访问服务绑定

EMP 提供的三种数据库访问方式都有各自的访问服务调用接口（TableService、SQLExecService、ProcedureAccessService），并针对不同的数据库（如 DB2、Oracle）的差异提供了相应的访问服务实现类。在 EMP 2.1 版本之前，三种数据库访问服务需要单独定义在 RootContext 中，而在业务流程中进行引用；在 2.2 版本中，提供了一种更方便的定义方式，即将数据库访问服务绑定在数据源上。这样业务流程的某一步骤只需取得数据源，就可

以得到相应的数据库访问服务，省去了大量的冗余定义。

数据库访问服务绑定的定义方法如下：

```
<DataSource id="dataSource" implClass="com.ecc.emp.jdbc.JDBCDriverDataSource">
    <resources implClass="java.util.HashMap">
        <TableService id="tableService"
implClass="com.ecc.emp.jdbc.table.TableServiceForDB2" />
        <SQLExecService id="sqlService"
implClass="com.ecc.emp.jdbc.sql.SQLExecServiceForDB2" />
        <ProcedureAccessService id="procedureService"
implClass="com.ecc.emp.jdbc.procedure.ProcedureAccessServiceForDB2" />
    </resources>
</DataSource>
```

将三种数据库访问服务定义挂在 DataSource 的 resources 子节点下即可。各自的 id 如下：

表映射操作服务：tableService

SQL 执行服务：sqlService

存储过程服务：procedureService

7.4.2. 表映射操作功能

7.4.2.1. 表映射操作要素分析

表映射操作就是指通过 Table Data Mapping 的方式将数据表字段直接映射到 EMP 平台的数据结构定义中，在访问数据表时通过表映射字段的设置自动生成 SQL 语句完成对数据库表的操作。

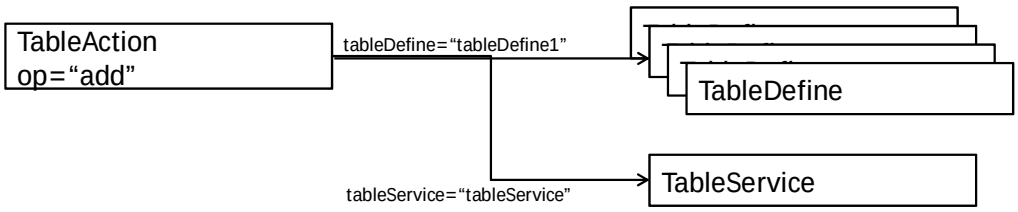
表映射操作处理实现包含在 EMP 平台产品包：com.ecc.emp.jdbc.table 包中。

EMP 在表映射操作中，全部操作要素被抽象为三种，分别是：

- 表字段映射对象定义：用于定义数据库表字段与 Context 数据的对应关系
- 数据库表访问服务：用于定义采用何种类型的数据库访问服务（对于不同种类的数据库由于存在一定的差异，所以数据库访问服务不尽相同）
- 表映射访问处理单元：用于将表映射操作嵌入处理流程

EMP 对表映射操作的设计原则是对每张数据库表提供独立的表映射定义，EMP 提供统一的公用的 TableService 服务来完成具体的针对表映射对象的增、删、改、查等数据库操作的自动实现。

在外部调用时，EMP 提供了表映射访问处理单元对象：TableAction。通过对 TableAction 的具体配置来灵活决定采用哪一个数据表映射对象和具体进行哪种数据库访问操作。



TableAction通过属性定义来指定本次访问操作所需要的
表映射定义对象：TableDefine和表映射操作服务对象TableService。
TableAction通过属性op来指定所需要操作的类型

7.4.2.2. 表映射对象定义——TableDefine

为了在 EMP 数据模型和数据库表间交换数据，JDBC 组件采取了定义数据映射的方法，即在数据库表定义服务中定义 EMP 数据域与数据库表字段的映射关系。实现类为：com.ecc.emp.jdbc.table.TableDefine。

TableDefine 对象的定义如下面的 XML 片段：

```
<TableDefine id="testTableDefine" tableName="TESTTABLE" schema="EBANK" maxLine="5">
    <column dataName="id" columnName="ID" columnType="INTEGER"/>
    <column dataName="userid" columnName="USERID" columnType="VARCHAR"/>
    <column dataName="name" columnName="NAME" columnType="VARCHAR"/>
</TableDefine>
```

如上表，tableName 表明了它是对数据库哪张表进行数据字段映射处理，schema 代表数据库访问空间，maxLine 可以控制为在查询情况下最大查询返回记录数。

在 TableDefine 定义的子节点中，是该表字段的具体映射信息定义。它通过三个属性进行设定。

dataName	映射到数据结点中的数据对象的名称
columnName	需要被映射的表字段名称
columnType	映射数据类型，目前支持的数据类型如下： Char，Varchar，LongVarchar，Decimal，Numeric，Integer，Long，Bit，Boolean，Float，Double，Real，Binary，VarBinary，LongVarBinary，Date，Time，TimeStamp 等等 目前并不支持 Blob 和 Clob 数据的映射

在数据库表映射定义对象中所定义的映射关系是公用的默认映射，若有在某一次访问中临时修改映射的需求，则可以在操作步骤(Action)中进行新的定义。

7.4.2.3. 表映射访问服务定义——TableService

一个公共服务类，用于对表映射定义对象进行解析操作完成最终对数据库表的访问。一般来说，针对不同的数据库类型可提供一个公共的映射访问服务对象的实现。在 EMP 中，提供了通用的表映射访问服务定义：com.ecc.emp.jdbc.table.TableService。并针对不同的数据库提供了不同的特殊实现，EMP 分别提供了 Oracle 和 DB2 的缺省表映射访问服务的实现，他们分别为：

Oralce： com.ecc.emp.jdbc.table.TableServiceForOracle

DB2: com.ecc.emp.jdbc.table.TableServiceForDB2

在实际应用中，应根据当前应用所采用的数据库类型替换 TableService 的实现类定义。

TableService 是一个标准的服务类，它既可以按照一般服务的定义方法挂在应用的公共根结点上，也可以照前述与某个数据源绑定。基本定义方式如下：

```
<TableService id="tableService" implClass="com.ecc.emp.jdbc.table.TableService" />
```

7.4.2.4. 表映射访问处理单元定义——TableAction

表映射操作的 Action 需要调用数据库表定义服务和数据库表访问服务来实现对单表的增删改查操作。实现类为：com.ecc.emp.jdbc.table.TableAction。

在 Action 定义中需要指定：

- 数据源定义
- 事务管理类型
- 字段映射定义
- 表映射访问服务
- 操作类型
- dataMapping 设置，重载字段映射定义中的数据对应关系。

以及相关条件设定等，参考如下：

```
<action id="TableAction6" dataSource="dataSource"
        transactionType="TRX_REQUIRED" condition=" ACCOUNTINFO.ID=$id "
```

```
        tableService="tableService" tableDefine="accountInfoTable"
op="enquiry"
        label="查询关联帐户" iCollName="accountCollection">

    <transition ……/>

</action>
```

具体参数说明：

dataSource	数据源名称
op	操作类型
transactionType	事务类型，如果为 TRX_REQUIRED，则为需要事务支持并延续已有事务； 如为 TRX_REQUIRE_NEW，则为需要事务支持并产生一个独立的新事务。
tableService	表映射访问服务，如果省略，则使用数据源上绑定的服务
tableDefine	需要访问的表映射定义对象 当操作类型为查询类操作时，tableDefine 的定义可以是一个以上的数据表映射对象定义的组合，它们之间采用‘;’号进行区隔。如： tableDefine=”tableDefine1;tableDefine2’
condition	在执行操作时的条件语句。采用标准的 SQL 语法进行编写，并可嵌入 EMP 数据对象。通过运行时解析生成标准的条件语句。 如：condition=” userId=’$UserId’ and password=’123456’” 上面是一条 TableAction 的条件语句的定义，按照标准的 Sql 语法书写，其中 userId 为数据表字段名称，如果需要引用 EMP 数据对象的值，则采用$符号+EMP 数据域名称的方式进行表示，如示例中的$UserId，在运行时，TableAction 将自动获取 UserId 的值并生成标准的 SQL 条件语句。
iCollName	当需要进行查询时，需要设置存放返回 ResultSet 对象的 iColl 数据域的名称
columnsStr	当需要处理的数据表字段不需要表映射的全部字段时，可通过

	设置该属性将需要操作的列字段采用‘;’分隔符的方式列出。TableAction 在处理时将只处理该属性定义了的列字段。 如果该属性为空，则操作 TableDefine 中定义的全映射字段。
dataMapStr	在 TableDefine 中定义了 Column 与 EMP 数据域之间的映射关系，当我们操作一个表映射时需要将表字段映射到一个非 TableDefine 定义中的数据域时，我们可以通过在该属性中进行临时表字段映射，在 TableAction 执行过程中，将采用该属性定义的字段映射信息覆盖 TableDefine 中定义的字段映射信息，为 TableAction 的使用提供灵活度。 举例： dataMapping="columnName1:dataName1;columnName2:dataName2" columnName1，columnName2 为需要临时进行映射的表列字段名称，dataName1 和 dataName2 为本操作中临时映射的 EMP 数据域名称。他们之间采用‘:’进行指定，各个映射字段之间采用‘;’进行区隔。
tableSequenceColumn	在查询时的排序字段名称
maxLine	多条查询时的最大返回记录数

单表操作交易步骤会根据参数“op”的设置，自动将 Context 数据按照字段映射定义服务中的映射关系和数据库表进行交换。具体方式如下：

- 修改（op=”0”）：对满足查询条件的记录，使用当前 Context 中的 EMP 数据更新记录。
- 查询多条记录（op=”1”）：查询满足条件的所有记录，并把结果集赋给当前 Context 中的 iColl 数据集合。该 iColl 定义必须含有要操作的那些数据项。
- 分页查询（op=”2”）：见下一章节。
- 查询一条记录（op=”3”）：查询满足条件的第一条记录，并把记录的值赋给当前 Context 中的 EMP 数据。

- 删除（op=”4”）：删除满足查询条件的记录。
- 添加（op=”5”）：将当前 Context 中的 EMP 数据（若没有在 Action 中指定 columns，则使用所有做了映射的数据，否则只使用 columns 指定的数据，下同）作为一条新记录插入到数据库表中。

7.4.2.5. 分页查询功能

数据库表映射操作支持分页查询。将 Action 的 op 参数指定为 pageEnquiry，就会按照 Service 中设定的每页最大记录数进行查询。通过设置几个固定名称的数据域值，就可以实现上一页、下一页、指定页查询等功能。这几个数据域值是：

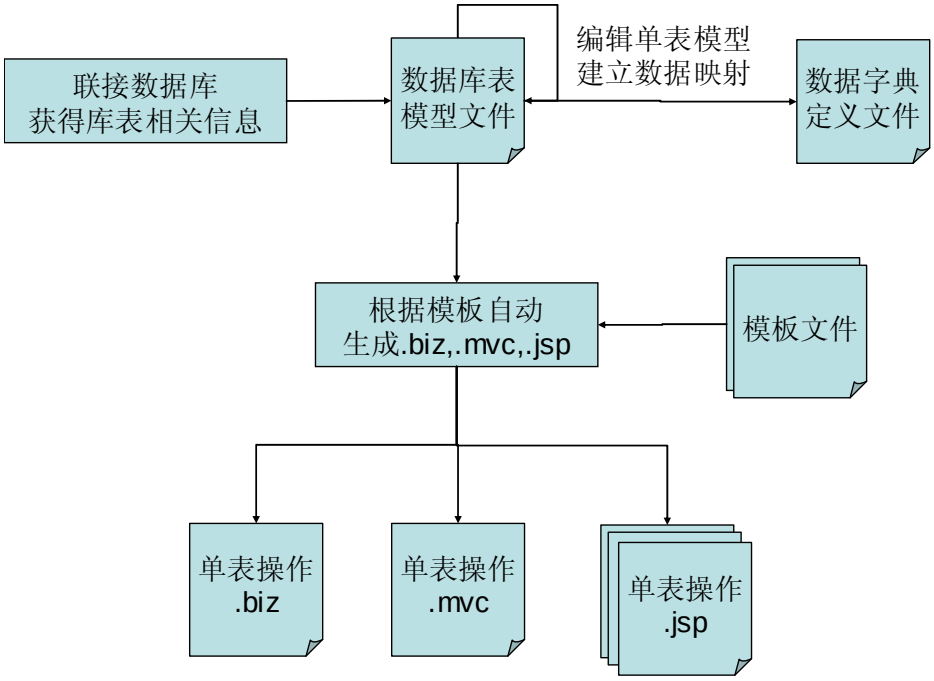
- firstKey 查询索引第一条记录
- lastKey 查询索引最后一条记录
- pageMode 查询排序方式
- recordSize 全部记录数
- currentPage 当前页面序号
- targetPage 目标页面序号

在实际应用中不需要手动设置数据域的值，可以使用 EMP 内建的 Taglib 标签支持。

7.4.2.6. 单表操作敏捷开发支持

借助 EMP IDE 的支持，我们可以通过 IDE 所提供的敏捷开发功能支持来快速定义单表操作的增删改查功能。

其操作过程类似下图：



- 借助 IDE 的支持，首先从数据库中获得某个数据库表的信息，并快速建立与 EMP 数据（即数据字典）的映射关系。
- 通过 IDE 的模版引擎，根据数据库表的模型文件自动生成 EMP 的业务逻辑定义、表现逻辑定义和页面，在这些文件中已经实现了单表增删改查的基础功能。
- 借助 IDE 用户只需要几分钟，无须编码或设置，即可完成一个单表的基础操作功能。如果有进一步需求，可以在这些文件中进行进一步的修改。

7.4.3. SQL执行功能

在单表操作不能满足需求的情况下（例如多表查询、需要用数据库内置函数生成某些字段等），则可以使用执行 SQL 语句步骤和服务来直接执行一条或多条 SQL 语句。

7.4.3.1. SQL执行操作要素分析

执行 SQL 语句服务同样分为 SQL 定义服务和 SQL 执行服务。在 Action 中可以引用多个 SQL 定义服务，在执行时会依次执行。

```
SQLExecAction:
    <action id="SQLExecAction0" sqlService="ebankEJSQL"
        transactionType="TRX_REQUIRE_NEW" dataSource="dataSource">
        <refSQL id="insertEJ"/>
        <refSQL id="enquiryEJ"/>
```

```
        <transition ……/>
    </action>
SQLDefine:
    <SQLDefine id="insertEJ" SQLStr="……" sqlType="insert">
        <input>
                <param dataName="id" idx="1"/>
                <param dataName="userid" idx="2"/>
                <param dataName="tranId" idx="3"/>
        </input>
    </SQLDefine>
     <SQLDefine id="enquiryEJ" ..../>
     ....
SQLExecService:
   <SQLExecService id="ebankEJSQL"
           implClass="com.ecc.emp.jdbc.sql.SQLExecServiceForDB2">
    </SQLExecService>
```

如上与 Table 映射操作功能实现类似，SQL 执行操作分别采用 SQLDefine、SQLExecService 和 SQLExecAction 三个对象来完成。

SQLDefine	实现类：com.ecc.emp.jdbc.sql.SQLDefine SQLDefine 定义一条独立的 SQL 语句，及相应的输入和输出与 EMP 数据域名称的映射
SQLExecService	实现类：com.ecc.emp.jdbc.sql.SQLExecService 执行 SQLDefine 对象，采用预处理 SQL 语句定义方式，将‘?’进行 EMP 数据域值注入并完成执行
SQLExecAction	实现类：com.ecc.emp.jdbc.sql.SQLExecAction 外部流程调用接口。在 action 中定义 SqlExecService 和 SQLDefine 名称，并通过调用 SQLExecService 完成对 SQL 语句的处理

7.4.3.2. SQLDefine定义

定义如下：

```
<JDBCSQLDef id="SelectAccount" iCollName="accountCollection"
SQLStr="Select
```

```
EBANK.ACCOUNTINFO.ACCOUNTNO ,EBANK.ACCOUNTINFO.ACCOUNTNAME ,EBANK.ACC
OUNTINFO.ACCOUNTTYPE ,EBANK.ACCOUNTINFO.ACCOUNTRATE ,EBANK.ACCOUNTINF
O.CURRENCYCODE ,EBANK.ACCOUNTINFO.OPENDATE ,EBANK.ACCOUNTINFO.ACCOUNT
BALANCE ,EBANK.ACCOUNTINFO.FREEZEBALANCE From
EBANK.ACCOUNTINFO,EBANK.USERINFO Where  USERINFO.USERID= ? AND
ACCOUNTINFO.ID=USERINFO.ID" sqlType="select">
    <output>
        <param idx="1" dataName="accountNo" dataType="VARCHAR"/>
        <param idx="2" dataName="accountName" dataType="VARCHAR"/>
        <param idx="3" dataName="accountType" dataType="INTEGER"/>
        <param idx="4" dataName="accountRate" dataType="DOUBLE"/>
        <param idx="5" dataName="CurrencyCode" dataType="VARCHAR"/>
        <param idx="6" dataName="openDate" dataType="VARCHAR"/>
        <param idx="7" dataName="accountBalance" dataType="DECIMAL"/>
        <param idx="8" dataName="freezeBalance" dataType="DECIMAL"/>
    </output>
    <input >
        <param idx="1" dataName="userid"/>
    </input>
</JDBCSQLDef>
```

一个 SQLDefine 定义一条 SQL 语句。SqlType 定义了 SQL 语句类型，分别为增删改查四种类型：insert，delete，update 和 select。当类型为 select 时，需要指定 iCollName 属性用来存放查询回来的数据内容。

SQLDefine 通过 input 和 Output 子结点来描述输入/输出数据与 EMP 数据域的映射关系。当操作类型为 select 时，需要进行 output 数据域的映射定义，它们映射的数据域是 iCollName 属性定义的 iColl 数据集合对象中的数据域对象。

SQLDefine 采用 param 对象来描述数据域的映射信息，如下：

```
<param idx="2" dataName="accountName" dataType="VARCHAR"/>
```

Idx 指明了输出数据的第二个输出列，dataName 指明了所对应的数据域名称，dataType 指明了它的数据类型。

7.4.3.3. SQLExecService定义

EMP 分别提供了 Oracle 和 DB2 的缺省 SQL 执行服务的实现，他们分别为：

Oralce： com.ecc.emp.jdbc.sql.SQLExecServiceForOracle

DB2: com.ecc.emp.jdbc.sql.SQLExecServiceForDB2

```
<SQLExecService id="sqlService"
        implClass="com.ecc.emp.jdbc.sql.SQLExecServiceForDB2">
```

该服务同样可以定义在根节点上，或者与某个数据源绑定。

7.4.3.4. SQLExecAction定义

SQLExecAction 的定义如下所示：

```
<action id="SQLExecAction" transactionType="TRX_REQUIRE_NEW"
dataSource="JDBCDataSource" sqlService ="SQLExecService"
refSQL="SQLDefine1;SQLDefine2;SQLDefine3">
…
</action>
```

基本定义方式很清晰，需要注意的是 SQLExecAction 可以一次执行多个 SQLDefine 对象，它们通过 refSQL 属性进行定义，采用‘;’进行分隔。若不定义 sqlService 参数，则使用数据源绑定的 SQL 访问服务。

SQLExecAction 在无论执行一个 SQLDefine 还是执行多个 SQLDefine 均在一个事务中完成。

7.4.3.5. 模糊查询

EMP SQL 执行组件支持 like 子句定义的模糊查询，当 SQL 中出现 like？这样的子句时，可以在对应的 input-SQLParameter 标签上设置查询模式。通过设置 prefix 和 postfix 两个参数，来定义模糊查询值前后的%、?等通配符，如下：

```
<SQLDefine id="queryUser" iCollName="userIColl" sqlType="select" >
    <SQLStr><![CDATA[
    SELECT * FROM USR WHERE NAME like ?
    ]]></SQLStr>
    <input>
        <SQLParameter paramIdx="1" dataType="VARCHAR" prefix="%" postfix="%"/>
    </input>
    <output>
        <SQLParameter columnName="ID" dataName="id" dataType="INTEGER" />
        <SQLParameter columnName="NAME" dataName="name" dataType="VARCHAR" />
        <SQLParameter columnName="ADDR" dataName="address" dataType="VARCHAR" />
    </output>
</SQLDefine>
```

7.4.3.6. 动态SQL定义（DynamicSQLDefine）

在某些场合下，预先写好的静态 SQL 无法满足较为复杂的需求，例如管理系统常见的条件组合查询：系统提供若干项查询条件，例如客户编号、姓名、性别、所在城市等，用户可以通过其中的一个或多个条件进行查询，而其它没有被输入的条件应该被忽视。为满足这种需求，EMP 提供了基于模板引擎的动态 SQL 定义，即 SQLDefine 的子类 DynamicSQLDefine。通过 SQL 模板的定义，在运行时根据输入条件动态产生最终的 SQL 语句，从而实现更复杂的数据库操作。DynamicSQLDefine 默认采用 velocity 作为模板处理引擎。

DynamicSQLDefine 会将 Context 以“_DATA”的名称传入模板引擎作为输入参数，因此可以通过$_DATA.get(数据名)的方法取到EMP数据的值。一个典型的条件组合查询的SQL模板如下：

```
SELECT * FROM USERINFO WHERE 1=1
     #if ($_DATA.get('id')) AND ID=?id #end
     #if ($_DATA.get('name')) AND NAME=?name #end
     #if ($_DATA.get('address')) AND ADDR=?address #end
```

该 SQL 模板在经过 velocity 处理之后，会把未输入的条件去掉，从而实现动态条件组合查询的功能。

另外，SQL 模板的输入项以“问号+数据名”的方式进行定义，在处理时会进行解析并通过 setObject 方法设置到 PreparedStatement 中，以防止 SQL 注入攻击。

以下是一个完整的 DynamicSQLDefine 定义（注意多了一个模板引擎接口的定义）：

```
    <DynamicSQLDefine id="DynamicSQL_Velocity" iCollName="userIColl" sqlType="select"
        <SQLStr><![CDATA[
SELECT * FROM USERINFO WHERE 1=1
#if ($_DATA.get('id')) AND ID=?id #end
#if ($_DATA.get('name')) AND NAME=?name #end
#if ($_DATA.get('address')) AND ADDR=?address #end
        ]]></SQLStr>
        <input>
            <SQLParameter dataName="id" dataType="INTEGER" />
        </input>
        <output>
            <SQLParameter columnName="ID" dataName="id" dataType="INTEGER" />
            <SQLParameter columnName="NAME" dataName="name" dataType="VARCHAR" />
```

```
            <SQLParameter columnName="ADDR" dataName="address" dataType="VARCHAR" />
        </output>
        <TemplateEngine class="com.ecc.emp.template.VelocityTemplateEngine" />
    </DynamicSQLDefine>
```

和普通的 SQLDefine 不同的是，由于输入和输出参数的个数不能事先确定，因此不能通过 paramIdx 属性来指定参数顺序，而是通过 dataName（输入）和 columnName（输出）来进行与 EMP 数据的对应。此外，由于输入参数实际上是定义在模板内（?数据名形式），因此<input>子标签下的定义只用于定义每个数据对应的字段类型。若没有定义某个数据对应的字段类型，则默认为 VARCHAR。

7.4.4. 访问存储过程

对于某些复杂需求、针对大量记录的操作或是经常要执行的任务，一般会在服务器上事先编写并编译好存储过程，在需要的时候进行调用，这样可以改善应用程序的性能。EMP 同样提供了访问存储过程的 Action 和 Service。使用时需定义存储过程的输入、输出与 EMP 数据域的对应关系，根据存储过程具体情况还可能有一个到多个结果集，对应 EMP 数据模型中的 iColl。

```
JDBCProcedureAction:
    <action id="JDBCProcedureAction0" isBatch="disable"
             transactionType="TRX_REQUIRED" dataSource="DB2JDBC"
             procedureService="JDBCProcedure"
procedureDefine="userLogonCheck">
        <transition ……/>
    </action>
JDBCProcedureDefine:
    <JDBCProcedureDefine id="userLogonCheck" retCodeName="procRetCode"
                         procedureName="PB_LOGIN_CHECK">
        <input>
            <param dataName="logonType" dataType="VARCHAR"/>

            ……
        </input>
        <output>
            <param dataName="procRetCode" dataType="VARCHAR"/>

            ……
        </output>
    </JDBCProcedureDefine>
```

EMP 分别提供了 Oracle 和 DB2 的缺省存储过程访问服务的实现，他们分别为：

Oralce： com.ecc.emp.jdbc.procedure.ProcedureAccessServiceForOracle

DB2: com.ecc.emp.jdbc.procedure.ProcedureAccessServiceForDB2

```
<ProcedureAccessService id="procedureService"
      implClass="com.ecc.emp.jdbc.procedure.ProcedureAccessServiceForDB2" />
```

该服务同样可以定义在根节点上，或者与某个数据源绑定。

7.4.5. 事务一致性管理机制

在使用 JDBC 数据库访问组件时，用户可以根据需要声明该访问的事务类型（应用全局事务或创建独有事务），由 EMP 的事务管理机制自动控制事务的提交和回滚。EMP 平台提供了两种事务管理机制，一种是基于 JDBC 数据源的事务管理，另一种是多数据源的 JTA/JTS 的事务管理机制。一般来说在开发阶段使用传统的 JDBC 的封装组件 DataSourceTransactionManager 来进行事务管理，而在部署到应用服务器上时，可使用标准的 JTATransactionManager 来进行事务管理。这两种事务管理器在接口上是统一标准的，对用户来说，通过配置文件的修改就可以简单完成两者的替换。

事务一致性管理机制的具体说明请参看相应文档。

7.4.6. 使用说明

7.4.6.1. 表映射操作的配置实例

- TableAction 相关属性：

id	操作步骤标识符
transactionType	事务类型，即 TRX_REQUIRE_NEW 或 TRX_REQUIRED。前者代表创建独有事务，在执行完该步骤后立刻提交或回滚；后者代表全局事务，整个流程的所有全局事务数据库访问在最后会一起提交或回滚。
op	操作类别：insert（添加一条记录）；update（更新记录）；delete（删除记录）；retrieve（查询满足条件的第一条记录）；enquiry（查询多条记录）；pageEnquiry（分页查询）
dataMapping	数据映射关系，在服务定义中定义了默认的数据映射，如果有特殊的映射定义，可以通过此属性实现
columns	需要查询的数据库表列名，格式为以’;’分割的若干字段
condition	用于除 insert 操作的查询条件设定，其格式为：

	condition="BRANCHID='$branchCode' ..."，其中$标识部分将会被替换为 EMP 数据模型中的数据
iCollName	存放查询多条记录和分页查询时的结果数据的 iColl 名称
dataSource	数据源服务定义名称
tableService	数据表访问服务定义名称，若不定义则使用数据源绑定的服务
tableDefine	数据表定义服务定义名称

- TableDefine 相关属性：

id	服务标识符
tableName	数据表名
schema	数据库模式名
tableSequenceColumn	用于翻页查询时，数据库表中的顺序索引字段名称。当调用其翻页查询时，会根据此列的值来进行查询结果的翻页控制
maxLine	翻页查询时，每次返回的最大记录条数
column	数据库表列属性定义子标签

- column 相关属性：

dataName	数据库表列对应的数据名称
columnName	数据库表列的字段名称
columnType	数据库表列的数据类型，支持的类型有：CHAR,VARCHAR,LONGVARCHAR,INTEGER,DECIMAL 等

- Action 配置实例：

```
<action id="JDBCTableAction6" dataSource="dataSource" label="查询关联帐户"
      transactionType="TRX_REQUIRED" condition="ACCOUNTINFO.ID=$id"
      op="enquiry" columns="ACCOUNTNO;ACCOUNTNAME;BALANCE;"
       iCollName="accountCollection" tableDefine="accountInfoTable">
    <transition dest="SetValueAction0"/>
</action>
```

- Service 配置实例：

```
<JDBCTableDefine id="accountInfoTable" tableName="ACCOUNTINFO"
schema="EBANK"
                 maxLine="10" tableSequenceColumn="ACCOUNTNO">
    <column dataName="id" columnName="ID" columnType="INTEGER"/>
    <column dataName="accountNo" columnName="ACCOUNTNO"
columnType="VARCHAR"/>
    <column dataName="accountName" columnName="ACCOUNTNAME"
columnType="VARCHAR"/>
    <column dataName="accountType" columnName="ACCOUNTTYPE"
columnType="INTEGER"/>
    <column dataName="accountRate" columnName="ACCOUNTRATE"
```

```
columnType="DOUBLE"/>
    <column dataName="CurrencyCode" columnName="CURRENCYCODE"
columnType="VARCHAR"/>
    <column dataName="openDate" columnName="OPENDATE"
columnType="VARCHAR"/>
    <column dataName="balance" columnName="BALANCE"
columnType="DECIMAL"/>
    <column dataName="freezeBalance" columnName="FREEZEBALANCE"
columnType="DECIMAL"/>
</JDBCTableDefine>
```

上面这套配置查询了 EBANK.ACCOUNTINFO 表，使用 ACCOUNTINFO.ID=$id 这个条件（$id 代表当前 Context 中的 id 数据域的值）。此操作步骤只查询了 ACCOUNTNO,ACCOUNTNAME,BALANCE 三个字段，并把结果放在名为 accountCollection 的 iColl 中。由于该 Define 服务定义了全部 9 个字段的映射，因此可以用于所有对该表操作的 Action。

7.4.6.2. 单表分页查询的配置示例

以 3.1 节的 Service 定义为例行分页查询。Action 配置实例：

```
<action id="JDBCTableAction6" dataSource="dataSource" label="查询关联帐户"
        transactionType="TRX_REQUIRED" condition="ACCOUNTINFO.ID=$id"
        op="pageEnquiry" columns="ACCOUNTNO;ACCOUNTNAME;BALANCE;"
        iCollName="accountCollection" tableDefine="accountInfoTable">
    <transition dest="SetValueAction0"/>
</action>
```

可以看到定义方法只是将 op 属性设为 pageEnquiry。由于 JDBCTableDefine 服务中定义了 maxLine="10" tableSequenceColumn="ACCOUNTNO"，因此此操作会以 ACCOUNTNO 字段为索引，每次查出 10 条记录。

若要实现翻页，可在页面使用 emp:pageIndex 标签。

```
<emp:pageIndex   name="pageIndex"   previous_next="  上  页  ;  下  页  "
submitFormName="page" maxPageNum="5" first_last="首页;尾页" jumpPageLabel="
跳转至#targetPage;页" enquiryPageLabel="(第#currentPage;页/共#totalPage;页)
```

```
共#recordSize;笔 每页#maxLine;笔 "/>
```

7.4.6.3. 执行SQL语句的配置实例

执行 SQL 语句需要操作步骤 SQLExecAction、执行服务 SQLExecService 和若干个定义服务 SQLDefine。在 SQLExecAction 中可以引用多个 SQLDefine，它们会按照顺序依次执行。

- SQLExecAction 相关属性：

id	操作步骤标识符
transactionType	事务类型，即 TRX_REQUIRE_NEW 或 TRX_REQUIRED。前者代表创建独有事务，在执行完该步骤后立刻提交或回滚；后者代表全局事务，整个流程的所有全局事务数据库访问在最后会一起提交或回滚。
dataSource	数据源服务定义名称
sqlService	数据库 SQL 语句执行服务名称，若不定义则使用数据源绑定的服务
refSQL	要执行的 SQL 定义，可以有多个用分号隔开，会顺序执行

- SQLDefine 相关属性：

Id	SQL 标识符
input	输入数据子标签
output	输出数据子标签

- input/output 相关属性：

param	数据项定义子标签

- param 相关属性：

idx	输入输出和EMP数据的映射中的输入输出项序号，input中对应SQL语句中的?号数顺序，output中对应查询结果的字段顺序
columnName	输出和EMP数据的映射中的输出字段名，与idx二选一使用
dataName	输入输出和EMP数据的映射中的数据名称
dataType	数据类型，包括：CHAR,VARCHAR,INT,DECIMAL等

Action 配置实例：

```
<action id="SQLExecAction0" label="记录日志" sqlService="sqlExecService"
        transactionType="TRX_REQUIRE_NEW" dataSource="dataSource">
    <refSQL id="insertEJ"/>
    <transition dest="EndAction0"/>
</action>
```

Service 配置实例：

```
<SQLExecService id="sqlExecService"/>
<SQLDefine id="insertEJ" SQLStr="Insert Into EBANK.EBANKEJ (EBANK.EBANKEJ.ID,
                EBANK.EBANKEJ.USERID,EBANK.EBANKEJ.WORKDATE,EBANK.EBANKEJ.WORKTIME,
                EBANK.EBANKEJ.SEQNO,EBANK.EBANKEJ.TRANID,EBANK.EBANKEJ.TRANSTATUS,
                EBANK.EBANKEJ.FIRSTACCOUNT,EBANK.EBANKEJ.SECONDACCOUNT,
                EBANK.EBANKEJ.TRANAMOUNT,EBANK.EBANKEJ.CURRENCYCODE,
                EBANK.EBANKEJ.COMMENT) Values (?,?,CURRENT_DATE ,CURRENT_TIME ,
                default ,?,?,?,?,?,?,?)" sqlType="insert">
    <input>
        <param dataName="id" idx="1"/>
        <param dataName="userid" idx="2"/>
        <param dataName="tranId" idx="3"/>
        <param dataName="tranStatus" idx="4"/>
        <param dataName="firstAccount" idx="5"/>
        <param dataName="secondAccount" idx="6"/>
        <param dataName="tranAmount" idx="7"/>
        <param dataName="CurrencyCode" idx="8"/>
        <param dataName="comment" idx="9"/>
    </input>
</SQLDefine>
```

上面这套配置执行了 SQLStr 中定义的那句 SQL 语句，将当前 Context 中的 id,userid,tranId....等 9 个数据域的值作为一条记录插入到 EBANK.EBANKEJ 表中，而其中的 WORKDATE 和 WORKTIME 字段由 derby 的关键字 CURRENT_DATE 和 CURRENT_TIME 产生，SEQNO 自增型字段则自动产生下一个序列号。

7.4.6.4. 访问存储过程的配置实例

访问存储过程的 Service 有两个，分别是存储过程定义服务(JDBCProcedureDefine)和存储过程访问服务(ProcedureAccessService)。和存储过程本身相关的定义如输入输出等需要定义在存储过程定义服务中，而存储过程访问服务只负责根据定义去调用响应的存储过程。此外存储过程访问服务还提供对多种数据库差异的操作支持。

- JDBCProcedureAction 相关属性：

id	操作步骤标识符
transactionType	事务类型，即 TRX_REQUIRE_NEW 或 TRX_REQUIRED。前者代表创建独有事务，在执行完该步骤后立刻提交或回滚；后者代表全局事务，整个流程的所有全局事务数据库访

	问在最后会一起提交或回滚。
dataSource	数据源服务定义名称
isBatch	是否循环执行，取值设为 batch 或 disabled
iCollName	循环执行存储过程时，输入 iColl 名称。定义了 isBatch 和 iCollName 后，会依次将 iColl 中的每一条记录对应 input 的数据项进行循环执行。
procedureDefine	存储过程定义服务名称
procedureService	存储过程访问服务名称，若不定义则使用数据源绑定的服务

- JDBCProcedureDefine 相关属性：

id	服务标识符
retCodeName	返回值数据域名
procedureName	要调用的存储过程名称
input	输入数据子标签
output	输出数据子标签
resultSet	输出结果集子标签，可有多个

- input/output 相关属性：

param	数据项定义子标签。output中默认把返回值(retCode)作为第一项定义，需与retCodeName对应。

- resultSet 相关属性：

iCollName	存放返回结果集的iColl名

- param 相关属性：

dataName	输入/输出项对应的数据名称
dataType	数据类型，包括：CHAR,VARCHAR,INT,DECIMAL等

Action 配置实例：

```
<action id="JDBCProcedureAction2" label="处理用户非法登陆（用户非法）"
        isBatch="disable" transactionType="TRX_REQUIRED" dataSource="DB2JDBC"
        procedureDefine="userLogonForbidPass">
    <transition dest="EndAction3"/>
</action>
```

Service 配置实例：

```
<JDBCProcedureDefine id="userLogonForbidPass" retCodeName="procRetCode"
                     procedureName="PB_LOGON_FORBIDPASS">
    <input>
        <param dataName="session_customerId" dataType="VARCHAR"/>
        <param dataName="session_userRemoteIP" dataType="VARCHAR"/>
    </input>
    <output>
```

```
            <param dataName="procRetCode" dataType="VARCHAR"/>
        </output>
        <resultSet iCollName="iQuickMenu">
            <param dataName="businessCode1"/>
            <param dataName="businessCode2"/>
        </resultSet>
    </JDBCProcedureDefine>
```

上面这套配置访问了 DB2 数据库中名为 PB_LOGON_FORBIDPASS 的存储过程。这个存储过程有两个 VARCHAR 型的输入项，执行时接受的数据是 EMP 数据域 session_customerId 和 session_userRemoteIP。存储过程有一个返回值，存放到 procRetCode 数据域中。存储过程还有一个返回结果集，对应 iQuickMenu 这个 iColl，该 iColl 里面的 businessCode1 和 businessCode2 用于保存返回的两列数据。

7.5. 后台定时服务

7.5.1. EMP的定时服务机制

7.5.1.1. 定时服务的概念

定时服务，又叫计划任务功能。是指在系统中将某一个操作或功能按照指定的时间或周期自动运行的服务。定时服务在系统应用中是经常使用的，如数据库的定时备份功能和应用系统的定时批量处理功能等等都是典型的定时服务应用。

7.5.1.2. J2EE的定时服务实现

7.5.1.2.1. J2EE的实现方式

- JDK 的标准实现

做为 J2EE 框架下的一个具体的功能应用，J2EE 通过 JDK 版本的更新在 1.4 版本以上的 JDK 标准版中已经提供了定时服务处理的接口和实现。它就是 java.util 包中的 Timer 类及 TimerTask 类。

我们可以直接使用 Timer 类对象并在其中加载我们所要执行的 TimerTask 对象，启动 timer 对象后，系统将自动按照 timer 对象中的时间点或循环周期定义自动的执行 timerTask 对象所代表的功能。

- 开源框架 Quartz 的实现

Quartz 是一个开源的定时服务处理组件的实现。它提供了一个轻量级的支持多线埕处理的计划任务管理和执行功能。

Quartz 和 timer 所实现的计划任务管理器组件只需要在 JDK 环境下就可以良好的运行。

- 基于应用服务器标准的 commonj 的实现

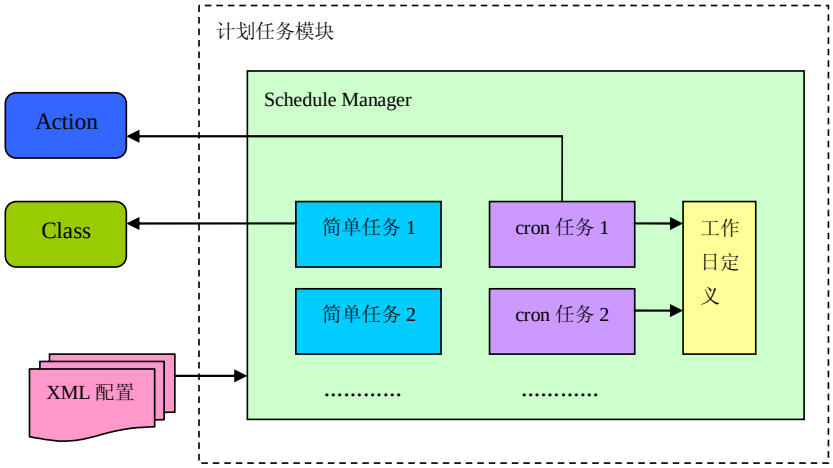
Commonj 是一个基于应用服务器标准的定时服务应用组件。它提供了一个时间管理器和一个任务管理器对象，这两个对象都是符合 WEB 容器和 EJB 容器规范的时间管理对象和后台任务管理对象。Commonj 提供的是利用应用服务器标准规范的资源来完成定时任务管理运行的功能。

所以说，commonj 是一个对应用服务器有依赖条件的定时服务组件，在应用服务器环境下，通过配置它的运行将由应用服务器来完成调度和管理，相应来说，定时服务的可靠性和效率都会有所提高。但由于它对应用服务器的依赖，很难在开发环境和非应用服务器环境下使用，所以我们需要根据我们所开发的定时服务的运行环境进行有针对性的选择。

7.5.1.2.2. EMP平台的定时服务实现方式

EMP 平台提供了自己的定时服务实现方式。EMP 平台的定时服务组件的核心还是采用了 J2EE 框架下当前比较流行的 Timer、Quartz 和 commonj 三种定时服务组件的封装。

EMP 平台的定时服务组件的结构设计图如下所示：



由上图可看到，计划任务模块由 ScheduleManager 统一对任务进行调度，而任务大致分为简单周期任务和复杂周期(cron)任务。其中复杂周期任务又可能需要一张工作日定义表。ScheduleManager 的调度工作可选择由 Timer、Quartz、commonj 等来完成。

定时服务组件主要有以下 4 个模块组成：

- Schedule Manager

是整个计划任务的管理员角色，以 EMP 服务(Service)形式根据外部 XML 配置对一系列任务进行安排和调度。实质上是一个接口，在其之上采用 JDK Timer、commonj、Quartz 等具体方式进行实现。

- 简单周期任务(simple)

按照一定周期反复执行任务，可设定周期、延时、重复次数等。

- 复杂周期任务(cron)

在每月、每周或每天的某个时刻执行任务，可设定条件、工作日等。并可根据工作日服务过滤或增加工作日管理。

- 常用任务实现

执行一个类或一个 ActionFlow 等 EMP 常用任务的实现。

执行一个类是指可调用任意一个 java 类中的方法来完成一个任务，特别注意：EMP 平台但目前还没有提供带参数的方法调用机制。

执行一个 ActionFlow 是调用一个 EMP 平台上的业务逻辑处理对象。这是 EMP 定时服务组件所建议的定时服务调用方式。

EMP 平台提供的定时服务组件想为用户提供的是一种对三种定时服务处理机制完全透明的定时服务功能。也就是说，我们在需要核心考虑怎样来设置我们的任务计划和任务计划

所需要执行的功能方法，而不用去管最终我们使用哪一种任务计划管理机制，针对 MEP 平台帮助用户屏蔽掉了三种机制上的差异。我们可以任意使用这三种机制对 EMP 平台所定义的任务计划进行管理。

7.5.2. EMP定时服务组件使用

7.5.2.1. 定义一个简单周期任务

一个简单的周期任务是指在任务计划启动时开始执行，根据间隔时间和执行次数，间隔一段时间后反复执行直到达到反复执行次数为止。在一个简单周期任务中，需要配置的是任务执行次数：repeatCount 和执行间隔时间：period。

在 EMP 定时服务组件中，一个简单任务执行的标签为：<SimpleTimerJob>，在该标签下可配置一个简单时间管理器对象：SimpleTimer 和一个可执行的操作，可执行操作分为两种：一种是一个 java 类方法的执行，配置标签为：<EMPClassMethodInvorkWork>，另一种是执行一个 EMP 平台定义的业务逻辑流程对象，配置标签为：<EMPFlowExecuteWork>。

如下是一个简单周期任务的配置使用示例：

```
<SimpleTimerJob name="simpleTimer"
implClass="com.ecc.emp.schedule.EMPSimpleTimerScheduledJob">
          <SimpleTimer
implClass="com.ecc.emp.schedule.EMPSimpleTimerSchedule"
repeatCount="2" delay="5000" period="1000"/>
          <EMPClassMethodInvorkWork
implClass="com.ecc.emp.schedule.EMPClassMethodInvorkWork"
methodName="hello" className="com.ecc.emp.test.schedule.HelloWorld"/>
</SimpleTimerJob>
```

配置的关键信息在 SimpleTimer 和 EMPClassMethodInvokrWork 标签内，每个标签的可配置属性含义如下：

SimpleTimer 标签的可配置属性		
属性名	**属性含义**	**属性可取值规则**
repeatCount	任务执行次数	不包含第一次执行次数，也就是说，如果想要总共执行 5 次，则该属性应配置为“4”
delay	第一次执行时的延迟时间	当该任务对象启动后，将在

		delay 时间后进行第一次执行，注意：第一次执行不计入 repeatCount 计算中
period	两次执行的间隔时间	以毫秒为单位计算
EMPClassMethodInvorkWork 标签的可配置属性		
属性名	**属性含义**	**属性可去取值规则**
className	所要执行的类对象的类名称	完整类名称或在应用中可找到的类名称
mothodName	所要执行的方法名称	注意：该方法目前不支持带参数的方法执行

在简单周期任务的使用描述中，我们并没有提到<EMPFlowExecuteWork>标签，该标签代表一个业务流程对象的执行，同样也可以配置到一个简单周期任务中去执行。该标签的配置使用描述将在下一节：复杂周期任务的使用配置中进行说明。

7.5.2.2. 定义一个复杂周期任务

一个复杂周期任务是指在指定的时间里启动执行，并可根据月、周、日进行反复执行设置。一个复杂周期任务提供了对工作日的管理，可根据工作日定义来增加或过滤工作日对象。

一个复杂周期任务的配置使用需要定义一个复杂周期时间管理器：CronTimer，根据要执行的任务定义一个：EMPClassMethodInvorkWork（用于执行一个类方法）或 EMPFlowExecuteWork（用于执行一个业务逻辑流程对象）。还可以根据需要定义工作日表，工作日表的定义有两种类型：一种是在把一个正常工作日做为非工作日处理；另一种是把一个非工作日做为一个工作日处理。一个复杂周期任务的工作日配置发生作用的条件是复杂周期任务的时间管理器按照工作日进行管理。

如下是一个复杂周期任务的配置使用示例：

```
<CronTimerJob name="cronTimer"
implClass="com.ecc.emp.schedule.EMPCronTimerScheduledJob">
          <CronTimer onlyWorkingDay="true"
implClass="com.ecc.emp.schedule.EMPCronTimerSchedule" cronType="0"
startTime="10:15"/>
          <WorkingDaysDefine id="workday"
          defaultWorkingDaysOfWeek="23456">
              <WorkingDaysException date="2007-03-01"/>
              <WorkingDaysException date="2007-03-02"/>
              <NonWorkingDaysException date="2007-01-01"/>
          </WorkingDaysDefine>
```

```
        <EMPFlowExecuteWork
implClass="com.ecc.emp.schedule.EMPFlowExecuteWork"
opId="setTimerSequenceNo" flowId="timerSchedules"
factoryName="misbizs"/>
</CronTimerJob>
```

配置的关键信息在 CronTimer、WorkingDaysDefine 和 EMPFlowInvokrWork 标签内，WorkingDaysDefine 标签的定义将单独进行描述，在这里描述 CronTimer 和 EMPFlowInvokrWork 两个标签的可配置属性含义如下：

CronTimer 标签的可配置属性		
属性名	**属性含义**	**属性可取值规则**
cronType	任务执行类型	0：每天执行；1：以周为单位执行；2：以月为单位执行
cronDays	可执行的日期列表 如果 cronType=1：则可执行日期列表为 1~7；如果 cronType=2:则可执行日期列表为 1~31	以','分割的日期列表 1-7 每周; 1-31 每月；L 代表最后一天，允许 2-4 代表从 2 到 4
startTime	启动时间	以 hh:mm:ss 方式设置时间，可设置多个启动时间，中间用“，”隔开，也就意味着在一天内可执行多次任务。
onlyWorkingDay	是否以工作日表来计算执行日期 以工作日表来计算执行日期是指按照 cronType 和 cronDay 设置的执行日期计算得到的结果还需要与当前工作日表之间进行匹配，如果执行日期落在非工作日内，则不执行。	True：是；false：否 当该值为 true 时，标签 <WorkingDaysDefine> 的定义才有意义。
autoDelay	非工作日是否自动顺延	True：自动顺延；false：否
EMPFlowInvorkWork 标签的可配置属性		
属性名	**属性含义**	**属性可去取值规则**
factoryName	EMP 中定义的组件工厂名称	可参考业务逻辑处理容器专题了解 factory 对象的含义
flowId	业务逻辑处理组件的名称	EMP 平台中 BizLogic 对象的 ID
opId	业务逻辑处理组件中的具体操作流程对象的名称	EMP 平台中的 operation 对象的 ID

7.5.2.3. 工作日配置使用

当 cronTimer 中的 onlyWorkdingDays 的定义为 true 时，我们需要定义工作日表。为 CronTimerJob 定义一系列工作日，可用 WorkingDaysException 定义特殊的工作日(例如周末加班)，或用 NonWorkingDaysException 定义特殊的非工作日(例如节日)。

工作日表定义示例如下：

```
<WorkingDaysDefine id="workday"
defaultWorkingDaysOfWeek="23456">
    <WorkingDaysException date="2007-03-01"/>
    <WorkingDaysException date="2007-03-02"/>
    <NonWorkingDaysException date="2007-01-01"/>
</WorkingDaysDefine>
```

工作日表是以周为单位进行定义的，也就是说，我们首先要设置我们每周的工作时间，示例中的属性 defaultWorkingDaysOfWeek 就代表着我们每周的正常工作日设置。在工作日表的定义中，是以周日为一周的开始，也就是说，1 代表周日，2 代表周一，依次类推，7 代表周六。如上我们的定义是“23456”，表明我们的每周工作日是周一到周五。

在定义了每周工作日后，我们可以为具体的工作日期例外进行定义，有两种定义类型，它们都采用属性 date 来表示一个具体的真实的日期，两种类型的配置描述如下：

- `WorkingDaysException`：工作日例外的定义

属性 date 的定义日期本身是一个工作日（按照周工作日设定的标准来看），通过 WorkingDaysException，它被标志为一个非工作日。如节假日。注意：WorkingDaysDefine 中只以周来定义工作日，并不考虑节假日的设计。所以对于节假日的过滤是需要采用 WorkingDaysException 的定义方式的。

- `NonWorkingDaysException`：非工作日例外的定义

属性 date 的定义日期本身是一个非工作日（按照周工作日设定的标准来看），通过 NonWorkingDaysException,它被标志为一个工作日。如加班的工作日。

7.5.3. EMP定时服务的配置和使用

7.5.3.1. 在EMP平台中的配置使用

在 EMP 平台应用中，定时服务是属于业务逻辑处理层的服务，我们将它定义在业务逻辑处理的 services.xml 配置文件中，基本配置文件示例如下：

```
<EMPQuartzScheduleManager id="QuartzScheduleTest"
implClass="com.ecc.emp.schedule.quartz.EMPQuartzScheduleManager">
       <CronTimerJob name="cronTimer"
implClass="com.ecc.emp.schedule.EMPCronTimerScheduledJob">
           <CronTimer onlyWorkingDay="false"
implClass="com.ecc.emp.schedule.EMPCronTimerSchedule" cronType="0"
startTime="10:08"/>
           <EMPFlowExecuteWork
implClass="com.ecc.emp.schedule.EMPFlowExecuteWork"
opId="setTimerSequenceNo" flowId="timerSchedules"
factoryName="misbizs"/>
       </CronTimerJob>
       <SimpleTimerJob name="simpleTimer"
implClass="com.ecc.emp.schedule.EMPSimpleTimerScheduledJob">
           <SimpleTimer
implClass="com.ecc.emp.schedule.EMPSimpleTimerSchedule"
repeatCount="2" delay="5000" period="1000"/>
           <!-- EMPFlowExecuteWork
implClass="com.ecc.emp.schedule.EMPFlowExecuteWork"
opId="setTimerSequenceNo" factoryName="misbizs"
flowId="timerSchedules"/-->
           <EMPClassMethodInvorkWork
implClass="com.ecc.emp.schedule.EMPClassMethodInvorkWork"
methodName="hello" className="com.ecc.emp.test.schedule.HelloWorld"/>
       </SimpleTimerJob>
</EMPQuartzScheduleManager>
```

在 EMP 平台中，应用采用一个扩展的 ScheduleManager 对象来管理定义的定时服务对象，一个 ScheduleManager 对象中可管理多个定义服务对象。

在 EMP 中，提供了三种 ScheduleManager 管理对象，分别对应到定时服务在 J2EE 上的三种实现，这三种 ScheduleManager 管理对象的标签使用分别是：EMPQuartzScheduleManager（支持 Quartz 框架的管理器）、EMPTimerScheduleManager（支持 Timer 组件的管理器）和 EMPCOMMONJScheduleManager（支持 commonj 框架

的管理器）。这三种管理器对用户来说是透明的，可替换的。如上面的示例，我们采用的是 EMPQuartzScheduleManager 管理器，如果我们想使用 Timer 管理器，我们只需要在配置文件中将标签：EMPQuartzScheduleManager 替换为：EMPTimerScheduleManager 即可。

注意：如果要使用 EMPCOMMONJScheduleManager 管理器，必须在标准的应用服务器环境下才能使用。

7.5.3.2. 扩展机制

在 EMP 平台不建议用户去扩展我们的定时服务机制，因为这些机制是属于标准的计划任务服务的实现。应该可以满足我们对定时服务功能的需求。

我们针对特殊需要可以采用扩展我们的任务执行功能来进行处理。也就是通过增加我们的 EMPFlowExecuteWork 和 EMPClassMethodInvokeWork 的功能来实现定时服务功能的增强。

在 EMP 平台上为可执行的任务提供了一个可扩展的接口：EMPWork。所有的定时服务均会调用该接口来完成可执行任务的功能。在该接口下定义了一个方法：execute（），用户通过扩展该接口就可以实现可执行任务的功能增强。

```
public interface EMPWork {
    public void execute();
}
```

7.6. 文件处理功能

7.6.1. 文件FTP功能

7.6.1.1. 概要介绍

文件传输协议（File Transfer Protocol，简称为 FTP）是用于在网络上进行文件传输的一套标准协议。EMP 的 FTP 组件提供了基于 FTP 协议的文件上传下载功能。

7.6.1.2. 工作原理

7.6.1.2.1. FTP组件构成

EMP 的 FTP 组件由一个操作步骤 FtpFileAcion 和一个服务 FtpFileService 组成。在操作步骤中定义单次交易的参数，如传输方式、要传输的文件路径等；在服务中定义 FTP 服务器的地址、端口、用户名、密码等公用参数。FTP 文件传输由 Service 实现，由 Action 调用，这样可将对同一个服务器的访问定义为一个公共服务，供所有业务使用。

以下是 FTP 组件 Action 和 Service 的定义方法：

```
Action：
<action id="FtpFileAction0" tranType="0" serviceName="ftpFileService"
        localPathField="clientPath" localFileNameField="clientFile"
        remoteFileNameField="serverFile" remotePathField="serverPath"
fileType="0"
        implClass="com.ecc.emp.comm.ftp.FtpFileAction" >

        <transition ……/>
</action>
Service:
<FtpFileService id="ftpFileService"
implClass="com.ecc.emp.comm.ftp.FtpFileService"
         ftpServer="127.0.0.1" port="4869" password="pass"
userName="user">
</FtpFileService>
```

7.6.1.2.2. FTP服务器参数

FTP 协议文件传输即在服务器端和客户端进行文件交换。FTP 服务器通常设定有控制端口、用户名和密码等参数。默认的控制端口为 21（在主动模式下传输端口为 20）。若服务器允许匿名登录，则可以使用用户名 anonymous、密码任意（通常为用户电子邮件地址）的帐号进行登录。

EMP 的 FTP 组件允许在服务中配置要连接服务器的地址、端口、用户名和密码。

7.6.1.2.3. 本地和远程的文件交换

通过 FTP 协议下载文件即将远程服务器上的文件复制到本地；上传文件即将本地文件复制到远程服务器中。EMP 的 FTP 组件通过以下四个参数来定义本地和远程文件的对应关系：

- localPath：本地路径　　取值例：c:/temp/
- localFileName：本地文件名　　取值例：foo.bar
- remotePath：远程路径　　取值例：/
- remoteFileName：远程文件名　　取值例：receive.dat

以上传为例，FTP 组件会将 c:/temp/foo.bar 文件上传到 FTP 服务器的根目录，并改名为 receive.dat。

为增强灵活性，在 Action 中实际需要配置的是存放这四个参数的 EMP 数据域。可以根据应用需要动态修改这些数据域的取值以上传/下载不同文件。

7.6.1.3. 使用说明

7.6.1.3.1. 配置实例

Action：

```
<action id="FtpFileAction0" tranType="0" serviceName="ftpFileService"
        localPathField="clientPath" localFileNameField="clientFile"
        remoteFileNameField="serverFile"      remotePathField="serverPath"
fileType="0"
        implClass="com.ecc.emp.comm.ftp.FtpFileAction" >
        <transition dest="EndAction0"/>
</action>
Service:
<FtpFileService                                          id="ftpFileService"
implClass="com.ecc.emp.comm.ftp.FtpFileService"
          ftpServer="127.0.0.1"         port="4869"         password="pass"
userName="user">
</FtpFileService>
```

- FtpFileAction 相关属性：

id	操作步骤标识符
tranType	传输方式：0=下载 1=上传

serviceName	FTP 文件传输服务定义名称
localPathField	存放本地文件路径的数据域
localFileNameField	存放本地文件名称的数据域
remoteFileNameField	存放远程文件路径的数据域
remotePathField	存放远程文件名称的数据域
fileType	文件类型：0=二进制 1=文本

- FtpFileService 相关属性：

id	服务标识符
ftpServer	FTP 服务器地址
port	FTP 服务器端口（默认为 21）
userName	登录用户名（默认为匿名用户 anonymous）
password	登录密码

7.6.2. 上传文件功能

文件上传处理组件是基于 EMP 平台实现的，应用于服务端，负责响应和处理客户端浏览器所发出的文件上传请求的功能模块。通过简单的配置，该组件就可以被 EMP 运行平台加载，从而配合 EMP 的请求处理器分析和处理文件上传的 Http 请求。

该组件提供了以下 3 项功能：

- 分析 EMP 请求处理器接收到的 Http 请求，分离出其中的文件上传请求。
- 分析文件上传请求，从中提取出文件数据流，保存到服务器端。
- 对文件上传的处理过程状态进行全程的实时监听和记录，同时提供了相应的 MVC 控制器，用于反馈即时的状态信息。

7.6.2.1. 基本原理

7.6.2.1.1. 支持文件上传的Http协议

RFC1867 协议规范，在传统的 Http 协议基础上实现了基于表单的文件上传功能，它为表单添加了扩展元素**<input type="file"/>**，其配置实例如下：

```
<form enctype="multipart/form-data" method="post" action="fileUp.do">
    文件上传测试：<br/>
    <input name="userfile1" type="file"/><br/>
    <input name="userfile2" type="file"/><br/>
    <input type="submit" value="提交"/>
</form>
```

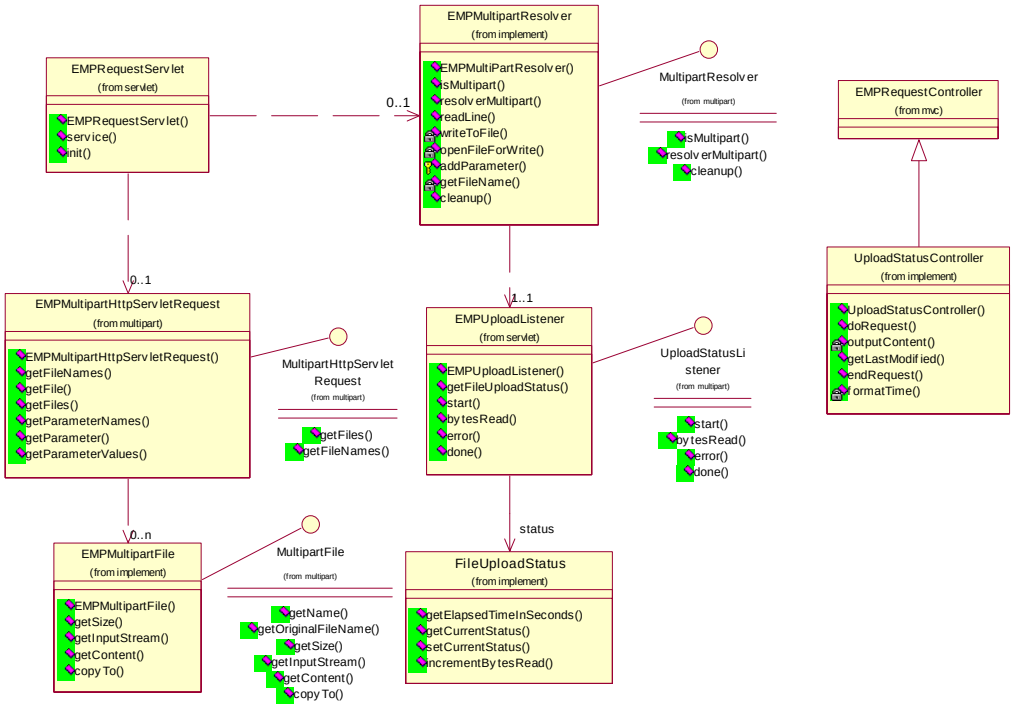
- 在 Form 中添加**<input type="file"/>**标签，浏览器在解析该标签时会自动生成一个输入框和一个按钮，输入框可填写本地的文件名和路径名，按钮可打开一个文件选择框以供选择文件。
- 将 Form 提交方式设置为 **post**。
- 为 Form 添加属性 **enctype="multipart/form-data"**。

根据 RFC1867 规范，浏览器会使用特殊的 Http 协议格式处理文件上传的任务。协议的变更如下面的实例：

```
contentType="multipart/form-data;+空格
+boundary=---------------------------7d41cca2001fc"
每一个数据域：
-----------------------------7d41cca2001fc\r\n
Content-Disposition: form-data; name=CTPKey.PAGEID\r\n
66\r\n
-----------------------------7d41cca2001fc\r\n
对文件内容：
Content-Disposition: form-data; name="uploadFile1"; filename="C:\\文本文档.txt"\r\n
Content-Type: text/plain\r\n
文件内容：
-----------------------------7d41cca2001fc\r\n
结尾：
-----------------------------7d41cca2001fc--\r\n
```

- 协议的 contentType 字段值变更为：**multipart/form-data; + 空格 + boundary=---------------------------7d41cca2001fc**。其中 boundary 参数值指定了请求输入流中所包含的数据流实体的首尾分隔符，boundary 的数字字符区由浏览器随机生成。
- 上传文件的内容将以数据流的形式被插入到 Http 协议的实体中，文件数据流被 boundary 指定的分隔符与协议中的其它数据域进行了分隔。

7.6.2.1.2. EMP文件上传处理组件的实现类图



- 文件上传处理器：**EMPMultipartResolver**
- 上传状态监听器：**EMPUploadListener**
- 上传请求封装类：**EMPMultipartHttpServletRequest**
- 上传状态报告器：**UploadStatusController**

7.6.2.1.3. EMP文件上传处理组件的功能描述

- 文件上传处理器 **EMPMultipartResolver**，配置在 empServletContext.xml 中，它会随 EMP 系统初始化，实例保存在 **EMPRequestServlet** 中。
- 根据 RFC1867 规范，浏览器会使用特殊的 Http 协议格式处理文件上传的任务。
- **EMPRequestServlet** 将通过 **EMPMultipartResolver** 分析接收到的所有 Http 请求信息，并从中分离出文件上传的任务请求，将其封装为 **EMPMultipartHttpServletRequest**，交由 **EMPMultipartResolver** 进行处理。分析的依据是文件上传协议中包含的 boundary=----------7d41cca2001fc 字段，boundary 的数字字符区由浏览器随机生成，它指定了请求输入流中所包含的文件内容数据流的首尾分隔符。

- **EMPMultipartResolver** 负责根据 boundary 分隔符，从上传请求输入流中分离出文件内容数据流，以 byte 数组作为缓冲，逐步组装为文件格式，并储存到服务器指定的物理路径之下。同时也会对文件重名、文件传输异常中断等问题进行特殊处理。
- 在 **EMPMultipartResolver** 接收文件数据流的过程中，上传状态监听器 **EMPUploadListener** 将会对整个过程的进行状态作实时的监听和记录。同时，**EMPUploadListener** 会将文件上传的状态记录封装为 **FileUploadStatus** 实例，并保存到当前请求的 session 中。
- 文件上传状态报告器 **UploadStatusController**，通过创建和访问该类型的 MVC 控制器，可以从 session 中获取 **FileUploadStatus** 实例，从而获知文件上传任务进程的当前状态。

7.6.2.2. 使用实例

7.6.2.2.1. 文件上传处理器的初始化配置

如果需要 EMP 平台在启动时加载文件上传处理器，则需要在配置文件 empServletContext.xml 中加入 **multiPartResolver** 处理器的初始化配置属性。相关配置参考如下：

```
<servletContext>
    <multiPartResolver tempFileRootPath="C:/Temp"
    class="com.ecc.emp.web.multipart.implement.EMPMultipartResolver"
/>
    ...
</servletContext>
```

目前 **EMPMultipartResolver** 仅提供了配置属性 **tempFileRootPath**，用于指定上传文件在 web 服务器端储存的物理路径。

7.6.2.2.2. 上传状态报告器的配置

当客户端发出文件上传请求后，EMP 服务平台对所上传文件的整个处理过程的即时状态，均可以通过上传状态报告器随时获得。上传状态报告器需要定义为 EMP 平台的一个 MVC 控制器，配置方案可参照下面的实例：

```
<action id="fileStatus"
    class="com.ecc.emp.web.multipart.implement.UploadStatusController
```

```
">
    ...
</action>
```

上传状态报告器可以定义在任意的 action.xml 配置文件中，亦可定义在 empServletContext.xml 中，该类型的 MVC 控制器未包含其他的配置属性。

需要说明的是，上传状态报告器所提供的功能是：获取上传文件在当前时刻的处理状态。如果需要获得文件上传的整个过程状态，则需要对该 MVC 控制器进行

7.6.3. 操作文件功能

7.6.3.1. 概要介绍

文件操作组件的开发是独立于 EMP 平台的组件体系的。文件操作组件可以当作 EMP 平台的一个服务组件提供给业务逻辑流程进行调用，同时，文件操作组件也可以由其它的程序直接进行调用。

文件操作组件实现了对文件的常用操作，其中包括了写文件、移动文件、删除文件以及文件重命名等。同时文件操作组件提供了文件名产生器接口，可以通过实现接口产生所操作文件的文件名。

通过外部配置可以在 EMP 的业务逻辑中使用文件操作组件，另外也可以将文件操作组件脱离 EMP 的业务逻辑，直接使用 JAVA 程序进行调用。

7.6.3.2. 基础原理

文件操作组件支持文件重名状态下的多种处理方式、对于文件移动等操作时文件的路径不存在时自动建立相应的文件夹目录以及对于绝对路径和相对路径的支持。

7.6.3.2.1. 文件重名问题

基本上所有的文件操作都面临着文件重名的问题，例如文件的移动。在文件操作组件中提供了三种处理方式：一种是实现文件名产生器的接口。在文件操作组件中，在进行所有的文件操作的相关动作之前，组件会调用相关的实现类产生目标文件的文件名，其中的目标文件代表的是最后真正操作的文件（如移动文件操作的目标文件名指的是移动文件到目标地址

后的文件名）。通过对文件名产生器接口的实现，可以在产生目标文件的文件名的同时对文件进行是否已存在的判断，当存在时可以采取其它的方式重新生成一个新的文件名。第二种处理方式是如果在进行文件操作之前，调用组件中文件存在判断的方法，通过判断的结果采取相应的措施，例如在移动时如果目标路径已存在同名文件，那么可以采取不移动或者将目标路径的重名文件删除然后再移动文件的策略。第三种处理方式主要用于写文件操作中，文件操作组件提供了一个 op_Type 属性，通过该属性的设置可以决定写文件时是从原有文件的头部开始写还是从原有文件的末尾继续添加。

如果在采取了上述处理三种方式之后，在进行相关的文件操行时依然发生文件重名的冲突时（除了写文件操作以外，该操作会根据配置采取上面第三种处理方式进行处理），组件会抛出 DuplicatedFileNameException 异常，而不会进行其它的操作。

7.6.3.2.2. 文件名包含路径问题

在进行文件操作时，也许文件所在的路径是在当前路径的子目录中，那么在配置文件名时，文件名的设置中就可以包含着文件所在的路径。在文件操作组件中，如果目标文件所在的路径并不存在，那么组件会自动的创建相应的文件夹目录，然后再进行相关的文件操作。例如，如果移动文件的目标路径并不存在，那么组件会根据文件名的路径自动的创建出相应的文件夹目录，然后再将文件移动到该目录下面。

7.6.3.2.3. 绝对路径与相对路径问题

文件操作组件支持两种路径格式：绝对路径与相对路径。可以通过配置 rootPathType 属性确定当前的文件路径是相对路径还是绝对路径。如果是相对路径，那么在系统初始化或者是调用文件操作组件时需要设置相应的根路径，确定了根路径后，接下来组件的所有操作都将在根路径下进行。

7.6.3.3. 文件操作组件的使用

7.6.3.3.1. 相关的属性及方法

属性介绍

属性名	属性类型	属性描述
fileRootPath	String	文件存放的路径
rootPathType	Int	是否为绝对路径，0：不是；1：是
opType	int	文件重名时的策略：0 时在文件末尾增加内容；1 时覆盖原文件；
genFileName	GenFileName	GenFileName 是一个接口定义的对象，用于产生文件名生成策略。 接口方法一个：genFileName(String filename)，需要继承实现
方法介绍		
方法名	参数	功能说明
deleteFile	String fileName	删除在 fileRootPath 路径下的名称为 fileName 的文件，删除成功则返回 true，失败返回 false 。若文件不存在，则抛出异常
deleteFile	String pathName，String fileName	删除在 pathName 路径下的名称为 fileName 的文件，删除成功则返回 true，失败返回 false 。若文件不存在，则抛出异常
editFile	String fileName，Byte[] content	将 content 中的数据保存至文件 fileName 中，路径为 fileRootPath，若路径下已有该文件，则根据 op_type 属性进行相关的操作。该方法最后返回生成的文件名
editFile	String fileName，Byte[] content，int type	将 content 中的数据保存至文件 fileName 中，路径为 fileRootPath，若路径下已有该文件，则根据 Type 值，若为 0，则在文件末尾增加内容；若为 1，则覆盖原文件。该方法最后返回生成的文件名
editFile	String pathName，String fileName，Byte[] content	将 content 中的数据保存至文件 fileName 中，路径为 pathName，若路径下已有该文件，则根据 Type 属性进行相关的操作。该方法最后返回生成的文件名
editFile	String pathName，String fileName，Byte[] content，int type	将 content 中的数据保存至文件 fileName 中，路径为 pathName，若路径下已有该文件，则根据 Type 值，若为 0，则在文件末尾增加内容；若为 1，则覆盖原文件。该方法最后返回生成的文件名
renameFile	String oldName，String newName	将 fileRootPath 路径下的 oldName 文件重命名为 newName。若是原文件不存在，则抛出异常，若新的文件名存在，则也抛出异常
renameFile	String oldName，String newName，String pathName	将 pathName 路径下的 oldName 文件重命名为 newName。若是原文件不存在，则抛出异常，若新的文件名存在，则也抛出异常
moveFile	String newPath，String fileName	将 fileRootPath 路径下的 fileName 文件转移到 newPath 路径下。若是原文件不存在，则抛出异常。若新的路径下文件名重名，则也抛出异常
moveFile	String oldPath，String newPath，String fileName	将 oldPath 路径下的 fileName 文件转移到 newPath 路径下。若是原文件不存在，则抛出异常。若新的路径下文件名重名，则也抛出异常
copyFile	String fileName，	将 fileRootPath 路径下的 fileName 文件复制到

	String newPath	newPath 路径下。若原文件不存在或者复制的文件重名，则都抛出异常
copyFile	String fileName , String oldPath , String newPath	将 oldPath 路径下的 fileName 文件复制到 newPath 路径下。若原文件不存在或者复制的文件重名，则都抛出异常
getFileContent	String fileName	得到服务器端 fileRootPath 路径下的 fileName 文件内容,，返回 byte[]类型
getFileContent	String pathName , String fileName	得到服务器端 pathName 路径下的 fileName 文件内容，返回 byte[]类型
isExistFile	String fileName	判断在 fileRootPath 路径下是否已存在 fileName 文件
isExistFile	String pathName , String fileName	判断在 pathName 路径下是否已存在 fileName 文件

文件名产生器接口：

FileNameGenerator		
方法介绍		
方法名	参数	功能说明
generateFileName	String fileName	产生文件名称，需要实现该接口。传入的参数是一个文件名称产生的种子字符串（要有完整路径）。

7.6.3.3.2. EMP平台的配置文件方式

对于在 EMP 平台中，文件操作组件是作为一个 EMP 服务的方式被使用。在使用时可以通过扩展业务逻辑操作组件在 context 中获得该服务的实例，然后再进行相关的操作。对于文件操作组件的配置如下：

```
//配置文件中的配置信息：
<FileOpService id=”fileOpService” fileRootPath=”c:\\temp” rootPathType=”1” opType=”0”>
    <FileNameGenerator id=”fileNameGet” implClass=”…”/>
</FileOpService>
//在业务逻辑操作组件中的使用：
FileOpService service = (FileOpService)context.getService(“fileOpService”);
Byte[] content = new Byte[1024];
.....
service.edit(“temp.txt”,content);
```

7.6.3.3.3. 直接调用的方式

除了在 EMP 平台中使用外，文件操作组件还可以由 JAVA 程序进行直接的调用。

首先可以通过构造函数得到一个实例，然后再进行相关的操作：

```
//暂时以 FileOpService 类名表示
FileOpService service = new FileOpService(“C:/Temp”,2);
//加载一个文件名产生器（根据情况是否需要决定）
service.setFileNameGenerator(FileNameGenerator)

Byte[] content = new Byte[1024];
//通过其它步骤，得到想要保存到文件中的数据
.....
service.edit(“temp.txt”,content);
```

7.6.3.4. 文件操作组件的扩展

文件操作组件提供的只是最基本的文件操作，当需要对文件进行复杂的操作时，可以通过组件提供的不同方法进行组合以实现相关的复杂操作。例如可以通过判断文件是否已存在来决定是采取覆盖的方式，还是不进行操作。

```
FileOpService service;
......
if(service.isExist(fileName)){
      return;
}else{
       service.remove(fileName);
       service.write(fileName,(byte[])content);
       ......
}
```

除此之后，文件操作组件提供了对文件名产生器的扩展接口，可以通过实现该接口来满足对文件重名的处理以及对文件名的特殊需求。

```
public class EMPFileNameGenerator implements FileNameGenerator{
    public String generateFileName(String fileName){
        ......
    }
}
```

8. EMP运行时装载

EMP 平台是一个标准的 J2EE 架构下的基础应用平台。通过应用服务器中进行应用部署的方式进行装载。EMP 平台通过 EMP 应用所定义的 web.xml 文件的内容进行 EMP 运行时环境装载。应用服务器启动应用时将读取 web.xml 文件内容，通过在 web.xml 文件中定义 Listener 的方式来完成 EMP 运行环境的装载。

下面详细说明 EMP 运行环境是如何通过 web.xml 文件进行配置和装载的。

8.1. 装载模式概述

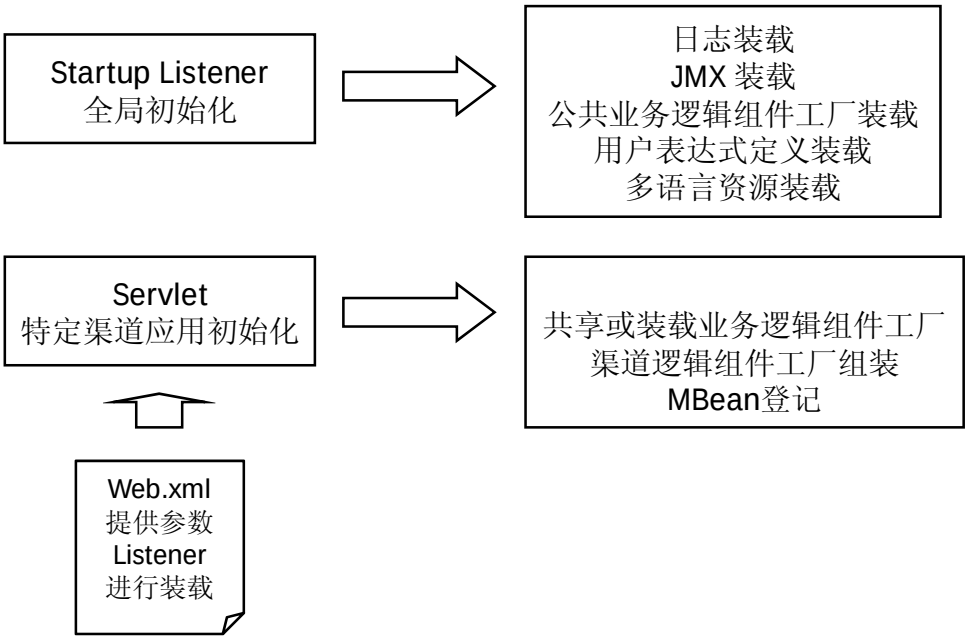
EMP 运行时环境装载分为两大部分，全局参数装载和渠道应用装载。

第一部分为全局参数装载，主要是装载 EMP 运行时日志参数、JMX 服务器定义、业务逻辑处理组件工厂对象、用户表达式定义和多语言资源定义信息等等全局化内容。

第二部分为渠道应用参数装载，是针对每一个访问渠道所定义的参数的装载。主要装载的内容包括：共享全局性组件工厂对象、渠道访问组件工厂对象的装载和渠道 MBean 对象的注册（用于监控）等等。

注：在 EMP 中，基于 Web 的 MVC 框架应用也是渠道应用的一种。

EMP 装载模式按照以下图形所示进行设计：



8.2. 全局参数装载

8.2.1. 装载入口定义

全局参数装载是 EMP 运行环境装载的关键步骤，EMP 平台提供了继承于 ServletContextListener.java 类接口的类实现来提供 EMP 运行环境的装载入口，该类的实现名为：com.ecc.emp.util.StartupListener，在 web.xml 文件中进行 Listener 对象的定义，该定义是不需要进行改变的，主要当该应用需要通过 Listener 对 EMP 运行环境加载更多的信息时可以通过继承并扩展该类的方式来提供更多装载功能。

在 Web.xml 文件中配置内容如下：

```
<listener>
    <listener-class>com.ecc.emp.util.StartupListener</listener-class>
</listener>
```

应用服务器启动时将城市化该类并调用该类方法来完成 EMP 平台运行环境的装载。

8.2.2. 全局参数定义

全局参数定义在 EMP 平台中采用在 web.xml 文件中定义 context-param 参数来进行定义。

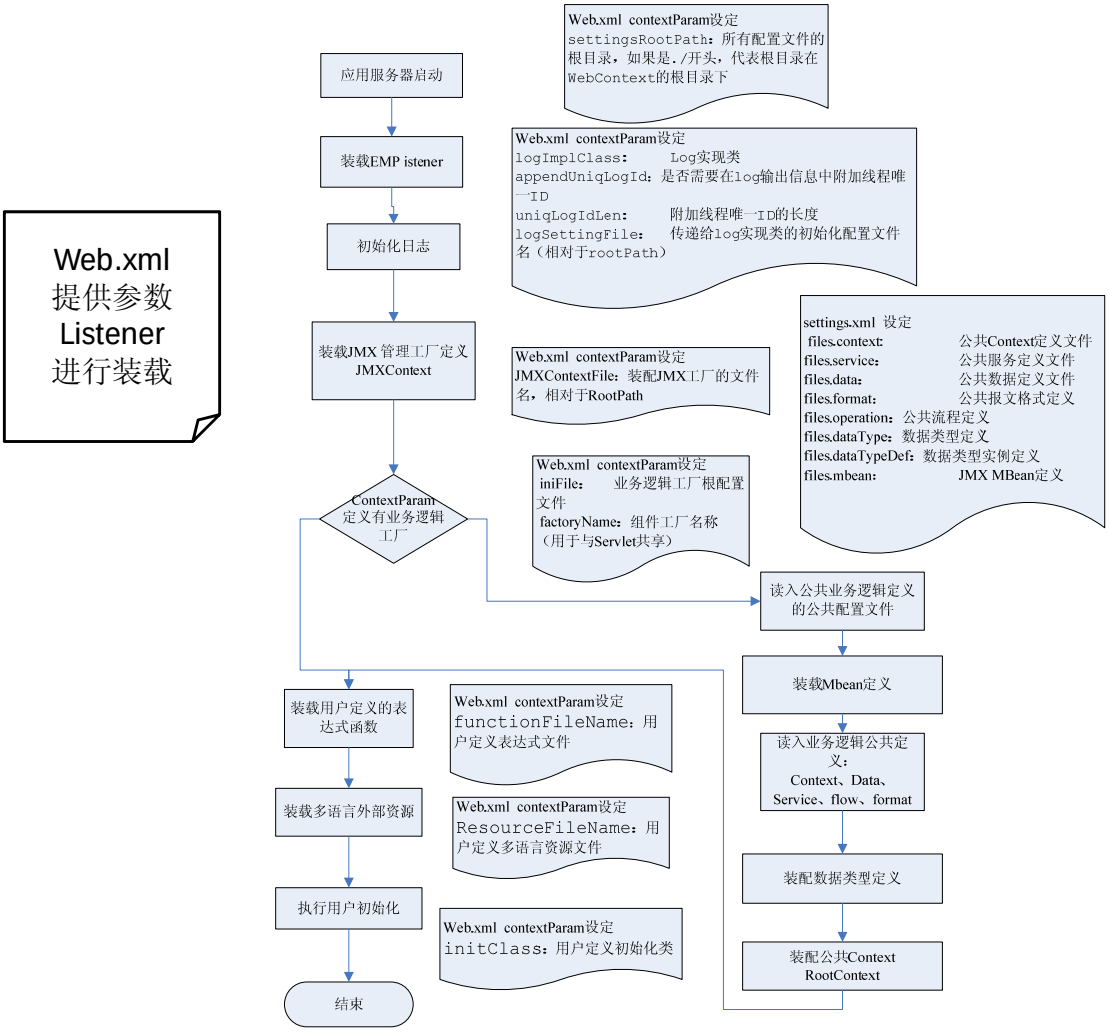
一个定义的示例如下，通过这些参数读取，EMP 运行环境将逐步建立起来。

```
<context-param>
        <param-name>settingsRootPath</param-name>
        <param-value>./</param-value>
        <!-- ./ 代表rootPath 为通过servletContext.getRealPath()得到 -->
    </context-param>
    <!--  JMX Manager setting param load by startuplistener -->
    <!--JMXContextFile指EMP平台所提供JMX服务器对象的初始化配置信息文件所在路径
-->
    <context-param>
        <param-name>JMXContextFile</param-name>
        <param-value>WEB-INF/EMPJMXContext.xml</param-value>
    </context-param>
    <!--  EMPLog setting param load by startuplistener -->
    <!--logImplClass指EMP平台所提供的日志处理的类实现，用户可以通过扩展该类来实
现自己所需要的日志处理功能 -->
    <context-param>
```

```
        <param-name>logImplClass</param-name>
        <param-value>com.ecc.emp.log.EMPLog4jLog</param-value>
    </context-param>
    <context-param>
        <param-name>appendUniqLogId</param-name>
        <param-value>true</param-value>
    </context-param>
    <context-param>
        <param-name>uniqLogIdLen</param-name>
        <param-value>6</param-value>
    </context-param>
    <!—logSettingFile参数指EMP平台所提供log日志配置相关信息文件，它采用标准的log4j的配置文件格式 -->
    <context-param>
        <param-name>logSettingFile</param-name>
        <param-value>WEB-INF/log4j.properties</param-value>
    </context-param>
    <!-— resourceFileName指EMP平台所提供多语言资源文件的路径 -->
    <context-param>
        <param-name>resourceFileName</param-name>
        <param-value>WEB-INF/commons/resource.xml</param-value>
    </context-param>
    <!-—functionFileName指EMP平台所提供表达式函数的定义配置文件名称，EMP表达式模型将通过该文件初始化表达式函数模型 -->
    <context-param>
        <param-name>functionFileName</param-name>
        <param-value>WEB-INF/commons/function.xml</param-value>
    </context-param>
    <!-—factoryName指EMP平台所提供的业务逻辑处理组件工厂名称，该工厂提供全局性的业务逻辑处理组件对象的获取和维护 -->
    <context-param>
        <param-name>factoryName</param-name>
        <param-value>Ebankbizs</param-value>
    </context-param>
    <!—如果定义了factoryName，就必须定义iniFile参数，该参数指EMP平台所提供组件工厂的初始化配置文件的路径 -->
    <context-param>
        <param-name>iniFile</param-name>
        <param-value>WEB-INF/bizs/Ebankbizs/settings.xml</param-value>
    </context-param>
    <!-- Common EMP flow context load by StartupListener -->
    <!-—如果定义了factoryName，rootContextName指EMP业务逻辑处理工厂所指定的根结点名称，不做特别修改该参数不变 -->
    <context-param>
```

```
        <param-name>rootContextName</param-name>
        <param-value>rootCtx</param-value>
    </context-param>
```

如上表所示，是 EMP 平台运行环境全局参数配置定义说明。EMP 平台运行时装载的基本流程如下图所示：



需要注意的是，EMP 在全局参数中的 factoryName、iniFile 和 rootContextName 三个参数是三位一体的，定义其中一个则必须三个全部定义，否则将不产生任何效用。在此定义了 factoryName 就是全局性的组件工厂对象，在各个渠道应用参数定义中可以只定义该 factoryName 的名称等于该处定义的参数值，意味着该渠道将采用全局组件工厂作为业务逻辑处理组件对象的容器。

我们也可以不在全局参数中定义组件工厂对象，而在渠道中进行组件工厂对象的定义，定义方式与上述方式一致。（由于渠道应用是通过 servlet 来装载的，所以 context-param 参数名称应改为 init-param 名称）

8.3. 渠道应用参数装载

渠道应用参数装载是采用在 Web.xml 文件中定义启动类 Servlet 方式进行装载的。EMP 平台针对不同的渠道提供了已实现的启动类 Servlet 实现，通过在 web.xml 文件中进行定义即可完成渠道应用参数的装载。

渠道应用参数装载请参考各个相关章节的内容，如 5.2 章节的 EMP MVC 模型实现以及第六章节 EMP 多渠道处理章节相关内容。

9. 典型应用功能的实现

9.1. 如何开始一个web应用

9.1.1. 目的

本专题主要讲述在 EMP 平台上如何创建和管理我们的 web 应用项目，在 EMP 平台上创建一个 web 应用项目需要注意些什么？那些步骤是无法回避和替代的？EMP 平台为用户提供的经典的 web 应用项目解决方案步骤是什么样的？

本专题在回答上述问题的同时，也会教会用户如何灵活的采用 EMP 平台所提供的框架，最高效率的创建自己的 web 应用。

9.1.2. Web应用设计应遵循EMP的三层设计思想

EMP 平台为用户提供了三个层面的实现框架：业务逻辑处理框架、web 应用逻辑框架和 web 页面表现框架。三个框架之间采用松耦合形式的接口定义方式进行交互，最大限度地解放了各个层面框架之间的耦合关系。EMP 平台提供了三个层面的解决框架并不意味着我们创建应用一定要使用这三个层面的框架，而是可以在任意层面上采用我们自己的框架来实现，只需要为其他框架的调用和交互提供标准的接口规范即可。

EMP 平台的三层框架结构是为了给用户一个良好的分层结构和设计模型。我们采用 EMP 平台来做一个应用，首先要考虑的是采用三层的框架结构来做设计，至于在每一层采用何种技术或是否采用 EMP 所提供的框架并不是 EMP 平台所强制的要求。当然 EMP 平台

所提供的缺省的各个层面的解决方案和框架会尽量做到最好，成为用户的首选一直是 EMP 平台努力的目标。但做为一个开放性的平台，它并不把此作为平台技术发展的一个必然束缚性要求。

EMP web 应用项目的设计也是按照 EMP 的三层框架的设计思路进行。EMP Web MVC FrameWork 是 EMP 平台提供的一套 web 表现层应用框架及解决方案。它对 Web 应用中客户浏览器端对 EMP 业务逻辑处理的请求、数据模型的更新、JSP 页面的展现以及页面间的跳转进行了抽象和简化，完全通过 XML 外部化配置文件进行定义和控制，并将页面展现与业务逻辑处理对象之间的关联进行了充分的解藕，使整个应用的结构更清晰，易于维护。

最关键的是，EMP 平台所提供的 MVC 框架及前端页面表现的 YUI 框架是开放性的，它并不要求用户一定要使用这样的框架来构建自己的 web 应用。EMP 平台遵循了一个 web 应用项目应用的基本流程，并将之封装起来，用一种很直观的方式展现给用户，同时屏蔽掉复杂的技术细节，完整的开发支持带给用户的将是高效率的开发和高质量的 web 设计框架的保证。

9.1.3. Web应用设计要素

9.1.3.1. web应用框架的选择

要做一个 web 应用项目，我们首先要做的一件事情就是选择我们所要采用的 web 应用框架。Web 应用的开发已经到了一个相对成熟的技术环境，各种 web 解决方案层出不穷，但还是很难找到一个完整的解决方案，大多数 web 框架往往只是为了解决 web 应用中某一个方面的问题。其实在应用中，选择什么框架并不是重点，重点是要理解框架的实质并遵循框架的规范才能真正发挥框架的作用。

什么是 web 应用框架？很多 web 应用框架实际上是在做页面表现的处理，在本文中认为 web 应用框架是 web 应用中页面流程交互和数据交互问题的解决方案和管理者。一个好的 web 应用框架应该帮助用户屏蔽掉 web 应用中的技术实现细节，并为用户提供一种以业务处理的模式来思考 web 应用功能的实现。

每一个 web 框架都有它的优点，在发展过程中同时也会暴露出框架的一些弱点。EMP 平台在设计时并不想将 EMP 平台的 web 框架做成一种复杂的能推而广之的 web 应用体系，考虑到框架的发展是一个逐步的，渐进的过程，EMP 平台所提供的 web 框架只是 EMP 平

台解决方案中的一个层面的问题，并不是全部。所以，EMP 平台提供的 web 框架首先是一个松散耦合的框架，它不会与平台的其他层面的设计框架产生交错的调用关系，它是通过简单的 MVC 设计思想将 web 应用框架简单化，并在此思想基础上提供了基于 EMP 平台的其他两个层面框架的缺省实现接口。

在我们为 web 应用做设计的时候，EMP 平台所提供的 MVC 框架只是一种选择，但如果要最大化的利用 EMP 平台的其他资源的话，EMP 的 MVC 框架能提供更完美的支持。我们也可以采用其他的 web 应用框架，如 struts、JSF 等等，但这些框架如果要使用 EMP 平台的前端表现框架和后端的业务逻辑框架，以及完整的开发工具支持就需要针对这些 web 框架开发 EMP 平台的支持接口。

总之，我们做 web 应用就需要选择框架，而 EMP 平台能提供一个灵活的，扩展能力强的 web 应用框架，同时在此基础上还可以提供完整的前端表现实现和后端业务逻辑实现以及完整的开发工具支持的能力。这些各个层面的框架组件的整合所产生的强大的生产力才是 EMP 平台的设计目标。

关于 EMP 平台的 MVC 框架设计的专题请参考“EMP MVC 框架设计专题”文档内容。

9.1.3.2. Web应用的页面表现能力

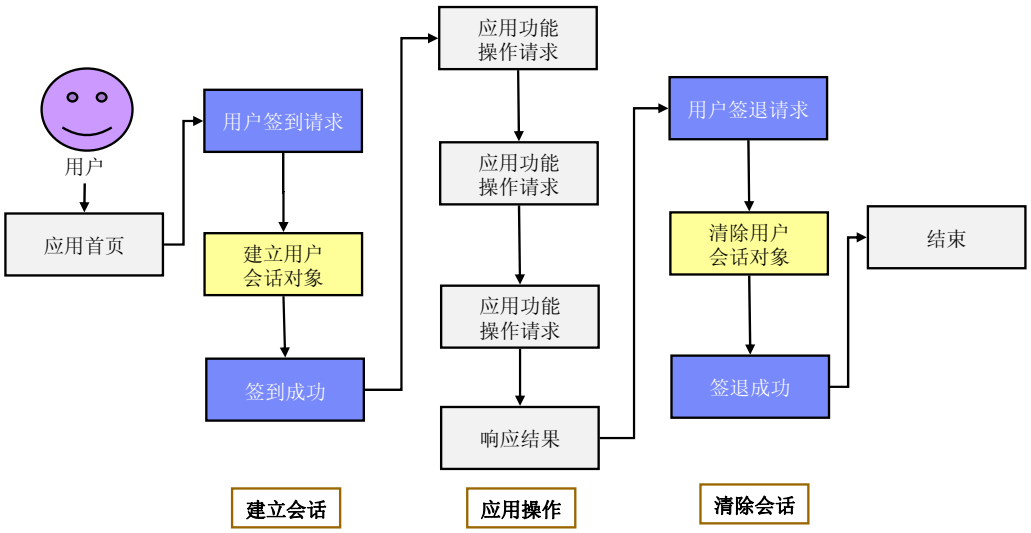
Web 应用的页面表现能力直接关系到该应用的可用性。页面表现能力与客户体验息息相关，越来越受到各个 web 应用客户的关注，各种页面表现处理的框架也越来越多。

由于 web 页面表现实现的语言 JavaScript 是一种脚本语言，它并非规范的编程化语言，在页面中的调用方式也十分灵活，这种灵活性和非规范性带来了这种脚本语言的复杂度和高维护成本。往往一个用户所写的 javascript 脚本只能由该编写者才能完全看懂！大多数页面处理框架解决的办法都是采用面向对象的思想将 javascript 函数进行封装，产生一个个可以通过参数配置并重复使用的页面组件。

EMP 平台选用的是 YUI 的页面处理框架。这是一个 yahoo 提供的免费页面处理工具库，提供了大量了菜单、选择框、表格、form 等组件的封装，并采用面向对象的事件驱动机制来传递逻辑和触发动作。在 YUI 框架中应用了大量的 AJAX 技术，并解决了 ajax 技术应用中的页面回退问题，更好的是，它能够在主流的浏览器上完美的运行，如 IE、firefox 等。能够很好的满足各种 web 应用的需求。

9.1.3.3. Web应用的会话管理设计

Web 应用的经典流程如下所示：



一个 web 应用可分为三个阶段，通常是建立会话阶段、应用操作阶段和清除会话阶段。由于 bs 结构的原因，签到用户的信息必须通过 session 对象保留在服务器端，同时也通过 session 对象的标示符（应用服务器内建的 sessionId）来识别对象的合法性。

通常，建立会话的阶段是一个用户签到的处理，用户通过身份验证后才能确认用户身份并在应用服务器上建立用户的会话对象，并通过该 session 对象的 ID 来唯一标示该用户身份。

应用操作是指用户签到建立会话成功后，可以进行应用功能处理请求了。在服务器端首先通过 session 的检查验证用户身份后，完成用户请求操作，并返回结果页面。

在用户完成了应用操作后，用户将选择退出系统，正常情况下，用户通过一个签退交易来完成退出，签退处理主要要完成的功能是清除该签退者已经建立的 session 会话对象。如果不清除会话对象，则有可能造成随着访问者的增加，系统内存消耗越来越大，最终服务器崩溃。非正常情况下，用户直接关闭浏览器退出，而没有直接清除掉用户的 session 对象。所有在所有的 web 应用中，均会有 session 超时的处理，应用服务器提供了缺省的超时处理，还有的是应用自身提供了 session 超时的检查和处理机制，通过这种超时机制的保障来维护系统内存及其他资源的稳定性。

EMP 平台在 web 应用支持开发上也遵循了这样的经典流程，通过提供不同的 controller，用户在应用中可以指定不同的动作请求做为应用建立 session 的操作（通常意义下，是签到处理请求做为建立 session 的操作请求）；在应用中常规业务请求可以指定该请

求是否需要检查 session 的合法性；可以指定清除 session 的请求。也能够很方便地定义 session 管理器服务对象的超时设置。

9.1.4. 应用EMP开发web应用

应用 EMP 开发 web 应用是 EMP 平台提供的基础功能之一。EMP 平台为 web 应用开发提供了良好的支持。这种支持具体体现在帮助解决 web 应用开发中的设计要素上的处理。EMP 平台提供了灵活可扩展的 web 表现框架，提供了灵活可配置的前端页面处理框架和 web 应用经典流程的辅助设计框架。在 web 应用开发中，当我们解决好了 web 应用的三大设计要素问题后，后续的工作就会相对轻松了，就可以将更多的精力来关注业务逻辑的实现。

应用 EMP 开发 web 应用从一个开发过程来看，要达到为用户提供快速的业务开发能力，帮助用户解决掉在 web 应用开发中碰到的技术设计问题，EMP 平台从如下几个专题对 web 应用开发提供了良好的支持。我们只需要掌握以下几个专题的内容，就可以轻松地使用 EMP 平台从无到有的来构建一个满足我们需求的 web 应用。

9.1.4.1. 定义应用的签到流程

定义应用的签到流程，实际上就是构造 web 应用经典流程中的创建会话对象的过程。一个用户只有通过了签到的身份认证后才能进行后续的业务操作。在 EMP 中，一个签到流程的处理也是一个普通的 MVC 的定义。它有登录信息页面，签到流程处理，错误页面和签到后的主页面四大部分构成，如下表所示：

对象名称	描述
signOn.jsp	签到页面，用于输入签到用户的用户名标识和认证数据（如密码等）信息
signOnEMPFlow	一个 EMP 的流程定义，用于定义用户签到的处理流程，可能包含用户身份认证，用户权限数据获取等等操作
Main.jsp	用户签到成功后的住页面，也就是主操作页面
Error.jsp	用户签到失败的响应页面

在 EMP 中，要完成一个签到流程需要注意以下两个部分的操作。

9.1.4.1.1. 定义会话数据

定义会话数据是指应用设计中需要把哪些数据放入到用户的 session 会话对象中。这些数据我们可以通过 IDE 工具在应用的业务逻辑分组中进行定义。在“应用逻辑分组设定编辑器”中，我们在应用结点分页操作编辑器中，可以为我们的 session 会话对象增加 session 会话数据，这些数据将在用户建立 session 会话之后一直保留在 sesion 对象中，直到用户的 session 被清除为止。

IDE 的基本操作界面如下图所示：

如上图，定义的数据为 session 的会话数据，注意它的访问方式列中都是“session”标志，说明这些数据都是 session 会话对象中存放的数据。

9.1.4.1.2. 定义应用的会话管理器

我们需要为我们的应用定义一个会话管理器，也就是 SessionManager 服务。在使用 IDE 工具创建一个 web 应用后，我们会为我们的表现逻辑处理创建一个表现逻辑分组，在表现逻辑分组目录下会有一个全局的 web 应用配置文件：empServletContext.xml。打开该文件，我们可以看到 EMP 平台在创建一个 web 应用时已经为我们添加了一个 sessionManager 服务了，内容如下：

<**sessionManager** name="BrowserReqSessionMgr" sessionCheckInterval="60000" sessionTimeOut="600000"/>

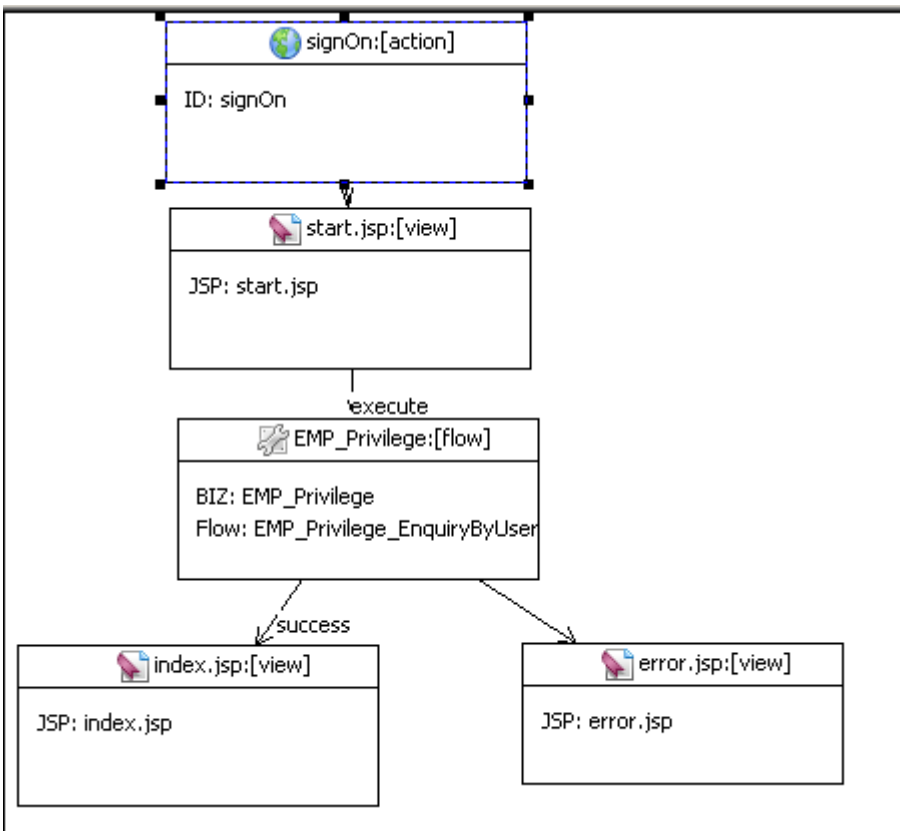
该 sessionmanager 的定义中，我们可以修改的以下参数：

参数名称	含义
sessionCheckInterval	毫秒为单位，表示 session 管理器多长时间进行一次 sesion 对象是否超时的检查，超时检查时间应小于 session 超时的时间设置
sessionTimeOut	一个 session 对象不活动的最大时间，超过该时间的 session 对象将被自动清除出内存空间

9.1.4.1.3. 定义签到流程的MVC

定义签到流程 MVC，一般在签到流程中都会调用一个指定的签到业务处理流程对象，在 EMP 平台中是一个 EMPFlow 对象，该对象的配置和定义与通常的 EMPFlow 的业务流程定义是一样的，在这里不进行描述。

在一个签到流程的 MVC 定义中，我们通常会用到本章节开始所提到的登录信息页面，签到流程处理，错误页面和签到后的主页面四个主要对象，通过 IDE 工具可以构建一个典型的签到处理 MVC 流程，如下图所示：



Start.jsp 实际上就是登录签到的提交页面，EMP_Privilege_EnquiryByUser 的 flow 定义实际上就是该签到交易所需要的业务处理流程，很明显该流程要做的是从数据库表中去查询该用户是否存在的操作，当操作返回‘seccess’时，则返回成功后的主页面，也就是‘index.jsp’页面，如果失败将返回‘error.jsp’页面。

在定义签到的 MVC 时，唯一与其他 MVC 定义不同的地方在于，在定义该 MVC 的操作类型时，需要选择该类型为“session”类型，同时是否检查 sesion 的选项要设置为“false”值。因为这是一个创建 session 的过程，而不需要去检查 session，IDE 中的操作界面如下所示：

id	signOn
类型	session
sessionContextName	PersonalSessionCtx
检查session	false
控制器类名	
检查附加码	
关联权限	false

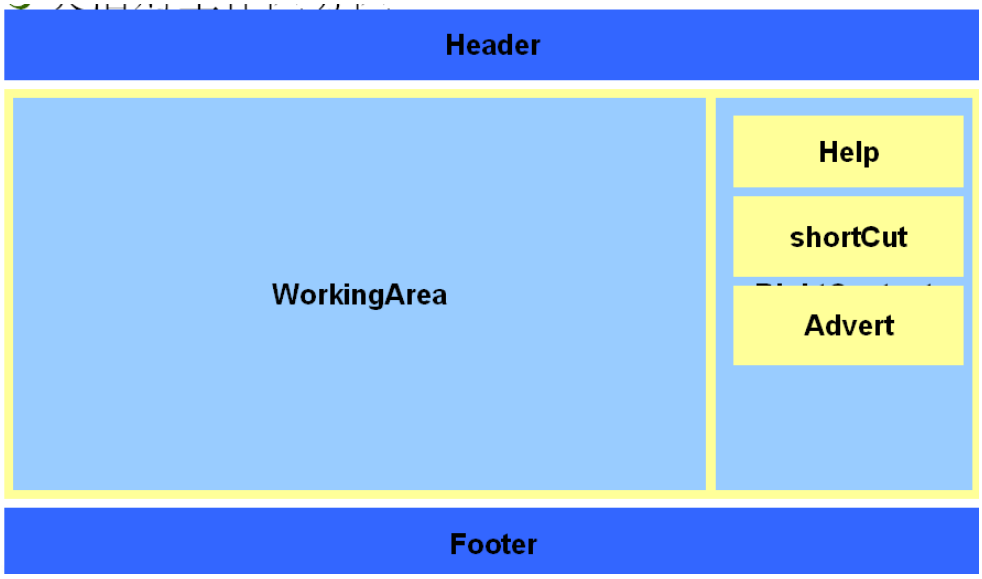
其中的 sessionContextName 是指用于存放 session 数据的数据结点对象，可通过属性选择框进行选择。

9.1.4.2. 定义应用的页面布局

定义了应用的会话建立的处理流程后，如果会话建立成功将进入主操作页面，也就是上面章节提到的 main.jsp 页面。我们在设计一个 web 应用时要考虑到主页面如何进行布局，最简单的考虑就是 web 页面的功能区划分，有主工作区、页面 title 区、菜单展示区、广告区等等。通过这些区域的设置来设计一个 web 应用的操作风格。一个 web 应用的操作风格直接关系到该应用的客户体验的好坏。功能分明、界面清新的布局总是能为一个 web 应用带来很好的用户反应。

在 EMP 平台中，为 web 应用的客户体验提供了很多新的技术支持，如 YUI 的页面组件、AJAX 技术支持等等。这些新技术都是基于 DIV 标签的布局来提供支持的。所以在 EMP 平台下的 web 应用将采用以 div 分区标签为主的页面布局功能。通过 div 分区标签进行布局处理的好处是可以通过 ajax 技术指定 div 分区进行区域更新，带来更好的服务器响应效率和客户友好体验。

在 EMP 平台下，目前对主页面布局没有提供图形化的编程支持，需要用户手工去完成一个主页面的布局管理。



如上图，这是设计的一个主页面布局。将主页面分为四个大区域，分别为 header 区、工作区、footer 区和右展现区，在右展现区上又可以嵌套一些更小的分区，如帮助区、快捷菜单区或广告区等等。通过 DIV 的分区标签可以进行分区的无限划分和嵌套。

下面是一个手工编写的简单的分区设置的主页面代码：

```
<body id="yahoo-com" onLoad="loadContent()">
```

```
<div id="doc3" class="yui-t4">
    <div id="hd">HEAD</div>

    <div id="MyMenu">
        <div id="Menu1" style="width:100%">
        </div> <!-- end Menu1 -->
        <div id="Menu2">
        </div> <!-- end Menu2 -->
    </div> <!-- end Menu -->

    <div id="bd">
              <div id="area_content"> </div>
              <div id="EMP_info"></div>
    </div> <!-- end bd -->

    <div id="ft">ccopy right(C)</div>
</div>
```

从上面的例子可以看出，整个主布局页面分为了四个大区，分别为‘hd’区，‘MyMenu’区、‘bd’区和‘ft’区。在‘MyMenu’区中定义了两及菜单区，分别是‘Menu1’和‘Menu2’。

Div 的布局定义只是描述了主页面的布局管理，还需要在页面 load 的时候装载每个区域的内容，这个操作的总入口是通过一个 JS 的动作函数来完成的。

```
<body id="yahoo-com" onLoad="loadContent()">
```

在 body 标签的定义中，定义了 onload 的响应函数为：“loadContent()”JS 函数，我们可以通过编写该函数实现来分别装载我们的主页面内容。

在 loadContent() 函数实现中，有一个很重要的内容就是要装载和注册我们的菜单结构，该函数的基本实现体如下所示：

```
    //创建一个EMPMenu对象
empMenu = new EMP.widget.EMPMenu('empMenu');
     //创建一个MenuBar对象，对应于分区“Menu1”
        var menuBar1 = new
EMP.widget.MenuBar({id:"Menu1",normalClass:"menu1_off",
activeClass:"menu1_on"});
     //创建一个MenuBar对象，对应于分区“Menu2”
        var menuBar2 = new
EMP.widget.MenuBar({id:"Menu2",normalClass:"menu2_off",
activeClass:"menu2_on"});
     //将menubar在EMPMenu上进行注册
        empMenu.registMenuBar( menuBar1 );
        empMenu.registMenuBar( menuBar2 );
    //将EMPMenu对象注册到响应工作区上，也就是说，当点击菜单时，响应的分区将是注册
```

```
  //的分区，一般来说就是住工作区
      empMenu.registContentDiv( 'area_content' );
     //下面的步骤是通过一个定义的tasktree.do的请求，向服务器端申请菜单树的数据
    //结构
      empMenu.loadMenuData('<ctp:jspURL
jspFileName="taskTree.do"/>');
```

关于菜单结构的组织和生成，将在下一节中描述。

9.1.4.3. 定义应用的菜单结构

EMP 平台对 web 应用提供的菜单处理机制是采用通过页面的菜单组件（前面章节提到的 MenuBar 组件等）对“xml 格式的菜单内容文件”在浏览器端进行解析生成的菜单结构内容。而 xml 格式的菜单内容文件则由服务器端根据需求实现接口的方式来生成。

在页面布局定义中的 loadcontent（）函数中的最后一句为：

```
empMenu.loadMenuData('<ctp:jspURL jspFileName="taskTree.do"/>');
```

该函数语句就是通过一个 emp 定义的 URL 请求连接去访问一个 tasktree.do 的操作，通过该操作从服务器端获得 xml 格式的菜单内容文件。

Tasktree.do 的 mvc 定义是需要手工进行编写的，它的编写位置也是在 EMPweb 应用的全局配置文件：empServletContext.xml 文件中，如下是一个 tasktree.do 的定义：

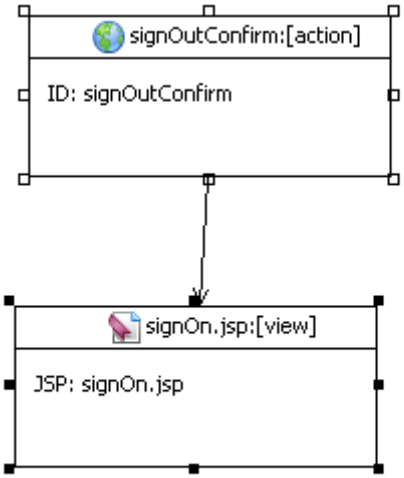
```
<action id="taskTree"
class="com.ecc.emp.web.taskInfo.TaskInfoXMLController">
       <taskInfoProvider
taskInfoFile="WEB-INF/mvcs/email/taskInfo.tsk"
class="com.ecc.emp.web.taskInfo.EMPTaskInfoProvider"/>
    </action>
```

一个 taskinfo.do 的定义中，包含一个实现了 controller 接口的 TaskInfoXMLController 类定义，由该类负责来生成前端菜单组件表现所需要的 xml 格式的菜单内容文件并返回到客户端，每一个 controller 定义下都包含一个 taskInfoProvider 的实现，通过该实现来构造一个 EMP 菜单模型。

关于 EMP 菜单内容的构造处理，请参考专题文档“《EMP 平台-菜单设计专题文档.doc》”的详细内容。

9.1.4.4. 定义应用的签退流程

应用的签退流程是很简单的操作，在 EMP 中提供了专用的签退处理类型，主要作用就是清除用户的 session 会话对象，保证用户的 session 会话对象所占用的系统资源得到释放。如下图的定义：



signOutConfirm 定义了一个签退的处理，注意该处理的属性表值的设置，如下图所示：

该操作的类型定义为“endSession”，只要设置该操作的操作类型为“endSession”即可。应用系统就会自动得在服务器端清除该用户的 session 会话对象。

添加为对应扩展步骤的子元素。