技术文档

第一部分、ofbiz 表现

一、理解 MVC 模式

当涉及大量商业逻辑项目的时候,我们需要考虑什么?如何分离用户界面和后台操作？如何避免将商业逻辑混淆于一般的流程控制中？作为企业信息系统， 就需要考虑很多类似的问题。

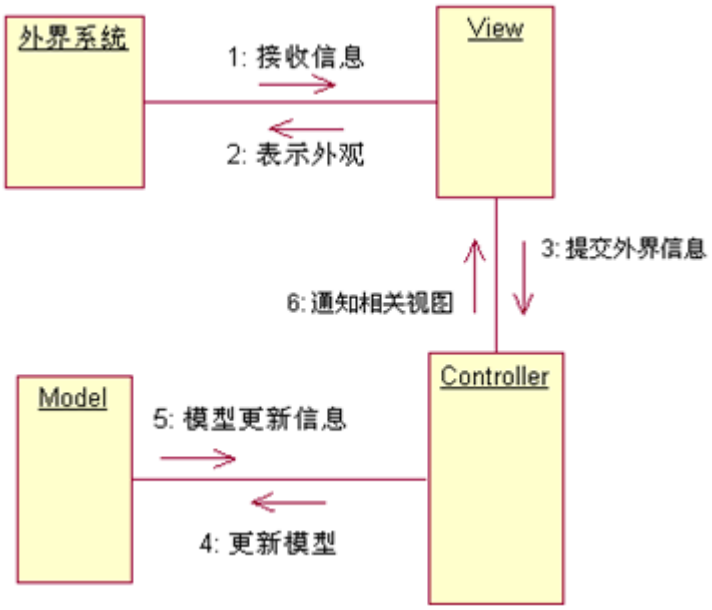
源源不断的客户新需求, 要进行功能修改和扩充, 但是因为程序的高耦合，改动将变得非常困难，导致项目成本何风险增加。 而且，往往维护人员与开发人员不是同一个人， 即使有详尽的文档，也很难理清程序里纵横交错的联系。 所以贯彻 Model-View-Controller（MVC）模式的设计， 在设计阶段首先杜绝此类问题, 是一个非常好的方法。

1、MVC 理论描述

所谓 MVC 模式，指的是一种划分系统功能的方法，它将一个系统划分为三个部分：

- 模型（Model）：封装的是数据源和所有基于对这些数据的操作。在一个组件中，Model 往往表示组件的状态和操作状态的方法。
- 视图（View）：封装的是对数据源 Model 的一种显示。一个模型可以由多个视图，而一个视图理论上也可以同不同的模型关联起来。
- 控制器（Control）：封装的是外界作用于模型的操作。通常，这些操作会转发到模型上，并调用模型中相应的一个或者多个方法。一般 Controller 在 Model 和 View 之间起到了沟通的作用，处理用户在 View 上的输入，并转发给 Model。这样 Model 和 View 两者之间可以做到松散耦合，甚至可以彼此不知道对方，而由 Controller 连接起这两个部分。

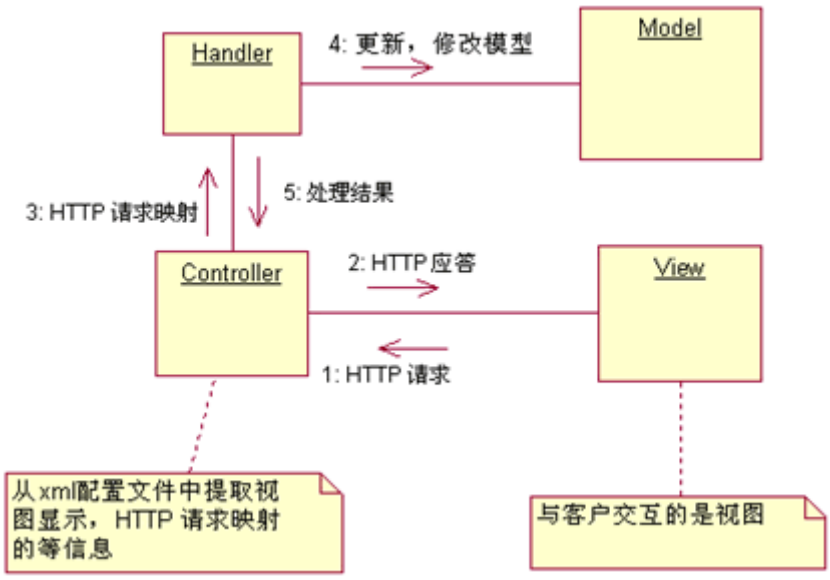
模型，即相关的数据，它是对象的内在属性；视图是模型的外在表现形式，一个模型可以对应一个或者多个视图，视图还具有与外界交互的功能； 控制器是模型与视图的联系纽带，控制器提取通过视图传输进来的外部信息转化成相应事件，然后由对应的控制器对模型进行更新； 相应的，模型的更新与修改将通过控制器通知视图，保持视图与模型的一致性。 下图（图 1.1）描述了这三者之间的关系：



2、系统设计

系统属于浏览器/服务器模型(Browser/Server)。一般的，客户通过浏览器发送 HTTP 请求给服务器端 Web 服务器，Web 服务器接收该请求并且进行相应处理，然后将处理后的结果返回到客户的浏览器中。在客户端，浏览器中呈现的正是该系统的视图部分。视图的作用就是提供给客户一个可视化的界面，并且允许客户输入一些数据来与服务器端程序交互。

对客户来说，他只能看到视图，而模型和控制器对他则是透明的。在这里 Web 服务器仅仅起到提供 HTTP 服务的作用。Web 服务器将客户提交的 HTTP 请求交给后方的 Jsp、Servlet 引擎，并且进一步交给其中的控制器来处理。控制器按照从 xml 配置文件中提取的请求映射表将该请求映射到相应的处理器（Handler）；处理器对模型进行更新、修改等操作，处理完后返回结果给控制器；控制器根据结果通知视图做相应变化，并且选择相应视图返回给客户。下图（图 1。2）说明了这一协作过程。



3、OFBiz 中 MVC 模式体现

OFBIZ 的 Web 应用框架严格遵循 MVC 模式。 OFBizMVC 中 Model 有它的封装业务逻辑的事件和服务承担. Control 有 controller 承担， View 有传统的 jsp, 和 FreeMarker,JPublish, Beanshell 承担。这里我主要说明 Control (Model, View将在相应技术的模块阐述). 在OFBiz框架中, Controller是一组管理 web 表示层对象, 其目的是将业务逻辑和表示层完全地分离开来。

1）过滤器(Context Security Filter)

Servlet API 2.3 中最重大的改变是增加了 filters，filters 能够传递 request 或者修改 response。Filters 并不是 servlets；它们不能产生 response。你可以把过滤器看作是还没有到达 servlet 的 request 的预处理器，或从 servlet 发出的 response 的后处理器。一句话，filter 代表了原先的"servlet chaining"概念，且比"servlet chaining"更成熟。

filter 能够:

在 servlet 被调用之前截取 servlet

在 servlet 被调用之前查看 request

提供一个自定义的封装原有 request 的 request 对象，修改 request 头和 request 内容

提供一个自定义的封装原有 response 的 response 对象，修改 response 头和 response 内

Context Security Filter (CSF) 在 /WEB-INF/web.xml 定义，用来限制访问 web 应用程序的文件. 具体参看下面的例子.

filter 配置如下:

```
    <filter>
  <filter-name>ContextSecurityFilter</filter-name>
  <display-name>ContextSecurityFilter</display-name>
  <filter-class>org.ofbiz.content.webapp.control.ContextSecurityFilter</filter-class>
  <init-param>
    <param-name>allowedPaths</param-name>
    <param-value>/control:/index.html:/index.jsp:/default.html:/default.jsp:/images</param-value>
  </init-param>
  <init-param>
    <param-name>errorCode</param-name>
    <param-value>403</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>ContextSecurityFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

2）Control Servlet 控制程序流程

Control Servlet 是所有请求过程的核心. 当收到一个请求时，servlet 首先设置系统环境信息, 初始化 request, session，实体代表（采用了"业务代表"的 设计模式, 我们将在关于实体引擎文章中介

绍）, Service Dispatcher, and Security Handler 存放在 ServletContext. 随后 control server 将 request 交给了 Request Handler 处理. Request Handler 处理完成后，返回给 control servlet ,整个请求过程就结束了。

3）Handler 对请求的处理过程

Request Handler 利用 RequestManager 帮助类收集在定义在/WEB-INF/controller.xml 中的请求映射(mapping) . 映射建立了 URI 和 VIEW 的对应关系。 URI 又和 Event 紧密相连，Events 用来处理可以直接通过实体引擎，或者调用服务的方式处理业务逻辑。

当 Request Handler 收到一个请求后，首先察看请求映射 (mapping)，如果没有发现对应的当前请求定义或者请求映射，返回错误。 如果察看成功，调用 Security Handler 检查当前的请求是否需要验证和使用这个 请求的用户身份验证。如果当前的用户没有验证，Request Handler 直接将请求转向到登录页面。 成功验证之后， Request Handler 寻找当前请求的事件 Event。并将请求转交给 EventHandler. 当事件处理完成之后， 又将 reuqest 移交给 ViewHandler. 如果 view 类型是 url，直接重定向到 url, 否则根据不同的 view 类型。调用不同的 view handler 处理(如:JPublishViewHandler, RegionViewHandler, FreeMarkerViewHandler)。

请求映射定义实例:

```
<request-map uri="checkLogin" edit="false">
<description>Verify a user is logged in.</description>
<security https="false" auth="false"/>
<event type="java" path="org.ofbiz.commonapp.security.login.LoginEvents"
                                    invoke="checkLogin" />
<response name="success" type="view" value="main" />
<response name="error" type="view" value="login" />
</request-map>
```

如上所示， 事件返回" success" 调用 view: main 。如果这里的 type="request", 返回成功以后，自动调用另外的请求。 这就是所谓的"请求嵌套"。

二、JPublish 合成表示层

1、经典的合成器--JPublish

JPublish 是一个功能强大的 web 发布系统. 它的目的也是最大的特点是将 web 开发人员的角色严格并清晰的区分开来。JPublish 支持多种模版引擎，包括 Apache Velocity, FreeMarker 和 WebMacro. JPublish 也支持各种脚本语言，如: Python, BeanShell, and JavaScript。 JPublish 是绝对是一个经典的"*合成器*", 平滑的扩展性让人赞叹。

JPublish 定义了多种角色用户：

- 应用程序设计师（Application designer）-- 主要是通过模板来为一个网站和这个网站的不同部分进行统一的设计，构架。
- 内容策划（content producer）-- 负责处理内容库（content repository）中的文档或其他信息。内容策划在板面设计中通常使用一套有限的设计元素（比如粗体或斜

体)。

◆ 特定业务领域的程序员（Domain-specific programmer）-- 负责 build 特定领域的 Java 代码，这可能是 JavaBeans，或者与 Web 应用程序相关的普通 Java 类。

◆ 集成程序员（Integration programmer ） -- 负责新建一个将特定代码和 Web site 连接起来的逻辑，该逻辑体现为 JPublish 的用于 scripts 自动化任务的 action system。

有了这种用户角色的清晰分离，实际上做到了 内容，程序逻辑，和表示逻辑的分离。

大多数开发者往往会将所有的任务放在一起，而没有将它们更好地组织起来。笔者之前所在的项目组也遇到了此类的问题：项目开始时，似乎已经做了很好的设计。然后将功能化成很多块，每个人负责一块。但是到了项目后期发现，需求无奈变更，代码变得很糟糕。 程序员往往为了解决一个问题，乱塞代码。导致原有的设计变形。结果项目难以维护，不停地 refactor。导致这样的后果是因为在开发阶段没有将开发人员的职能清晰的分离开来。调试程序时，总是将表示逻辑和程序逻辑，整合过程一同考虑。 这样代码的质量必然下降。导致最后为解决问题牺牲了代码的质量。

而使用这种**基于角色**的工具会使你不得不将这些任务分开并重新合成，我相信这一点一定会使你的程序变得更好更高效。

"Now everyone can do their specific task and put it all together。"

2、template + script

我们知道 JPublish 将 template, script 合成在一起。OFBiz 选择了 FreeMarker 和 Beanshell. view的表示逻辑写在FreeMarker, 程序逻辑写在BeanShell中，中间还有一个pagedef 文件，将 ftl 和 bsh 结合起来。 ***这里还有一个好处要提一下***，就是通过 pagedef 配置一个 ftl 对应多个 bsh, 那么就可以做到 bsh 文件封装的程序逻辑的重用，换一句话说就是一个 bsh 文件可以在多个地方使用。 所以完成一个 jpublish 类型的 view 实现，至少需要编写三个文件. ftl 文件, pagedef, bsh, 在不同的目录切换，某种意义上增加了工作量。

3、JPublish 和 MVC

JPublish 遵循 MVC 吗？ 回答是否定的.

笔者曾在网上看到“JPublish 提供了 MVC 架构” ,其实这是错误的。 从 MVC 模式的构成来，JPublish 完全不是。 JPublish 没有 control, pagedef 只是将 template 和 srcipt 合成起来。每次在 template web page 被执行之前首先执行对应的 script. 不存在控制流程的功能。在 OFBiz 中， JPublish 担任着 view 的

三、区块（Region）指南

1、简介

区块工具（或叫框架）是一个复合视图模式（在 Enterprise Architecture Patterns 描述的各种视图）的实现，同时也是基于 David Geary 著作的 Advanced Java Server Pages 的一种实现。在 www.ofbiz.org 的 Docs & Books 页面可以得到有关这本书更多的信息。

区块由许多命名节（准备插入到一个模板）组成的一个区域。区块和节在每个 webapp 的 **WEB-INF/regions.xml** 文件里进行配置。通过使用一个类型为 **region** 的视图，区块能与基于 web 应用的 OFBizControl Servlet 协同工作。

在 JSP 文件中，可以对区块进行声明，并且通过使用 regions.tld 文件里的标记把区块包含进来。这些标记也可以用在区块模板中，用来在模板的指定位置包含命名节。

2、定义区块

1）一个基本的区块定义

首先让我们看一下在 **WEB-INF/regions.xml** 文件中的一个实际区块的定义：

```
<define id='MAIN_REGION' template='/templates/main_template.jsp'>
    <put section='title'>Application Page</put> <!-- this is a default and is meant to overridden -->
    <put section='header' content='/includes/header.jsp'/>
    <put section='error' content='/includes/errormsg.jsp'/>
    <put section='content' content='/main.jsp'/> <!-- this is a default and is meant to overridden -->
    <put section='footer' content='/includes/footer.jsp'/>
    <put section='remote' content='http://www.yahoo.com' type="http"/>
</define>
```

define 标记用来声明一个命名区块。区块名在 id 属性指定。这是使用 JSP 模板的基本区块。你也可以定义一个从其他区块继承来的区块，并可以从下一节开始使用。在模板属性中指定模板全路径，并且是相对 webapp 的根路径的一个 webapp 资源，就象一个 webapp 里的 JSP 或 Servlet 一样。

模板文件可以插入的内容是常规文本，或其他文本，这些文本被插入在不同的地方。在一个区块中内容就是作为一个节而被插入。在区块模板中，节由名称进行标识，并且在配置一个区块时，通过 **put** 标记对准备插入到每个节的内容进行定义。

put 标记的 section 属性保存节名称。content 属性的值用来确定准备插入到节中的内容。内容还可以嵌入到 put 标记的 body 里，类似于上面示例的标题节。这里指的是直接内容。缺省地，content 属性值是用来定位当前 webapp 的某个资源的。

节类型在 type 属性中设定，并且缺省类型是 default。这些类型将在下面讨论。

注意：如果一个节被包含到一个模板中，却没有在区块定义中为它指定内容，那么当模板被使用时不会插入任何东西。

2）一个继承的区块

下面是一个通过继承其他区块来定义区块的例子。

```
<define id='login' region='MAIN_REGION'>
    <put section='title'>Login Page</put>
    <put section='content' content='/login.jsp'/>
</define>
```

这个定义没有使用定义标记中的模板属性，而是使用区块属性和指定被继承区块来代替。使用的模板与父区块模板相同，并且继承节的 put 标记中所有设定。子区块的 put 标记设定将覆盖父区块的设定。

在这种情况，login 区块从 MAIN_REGION 继承而来。MAIN_REGION 区块采用这种格式命名表示不直接使用，而是用来被 login 继承。我们发现采用这样的约定很有用。

Login 区块与 MAIN_REGION 布局相同，而"content"节内容将被覆盖，因此，可以在主内容区域（ main content area）使用 login.jsp 页面。为了给页面指定一个自定义标题，title 也要被覆盖。

3）在一个区块内的区块

下面的示例是一个主区块（main region）某节的内容，为了编写这种新风格的页面，通过继承主区块定义了一个区块。下面还包括一个区块示例，它从新风格的页面继承而来，并且覆盖了主内容区域和 title 节内容。这个示例能反映真实世界的状况。

```
<define id='LEFTBAR_REGION' template='/templates/leftbar_template.jsp'>
    <put section='first' content='/catalog/choosecatalog.jsp'/>
    <put section='second' content='/catalog/keywordsearchbox.jsp'/>
    <put section='third' content='/catalog/sidedeepcategory.jsp'/>
    <!-- <put section='fourth' content='http://www.yahoo.com' type="http"/> -->
    <put section='fourth' content='/catalog/minireorderprods.jsp' type="resource"/>
    <!-- <put section='fifth' content='/fifth.jsp'/> -->
</define>

<define id='LEFT_ONLY_REGION' region='MAIN_REGION'>
    <put section='leftbar' content='LEFTBAR_REGION'/>
</define>

<define id='showcart' region='LEFT_ONLY_REGION'>
    <put section='title'>Show Cart</put>
    <put section='content' content='/cart/showcart.jsp'/>
</define>
```

4）不同类型的节

只有几种类型的节总是可用。除了这些类型外，你还可以使用 controller.xml 文件里定义的任何类型的 Control Servlet View。在示例中包括有 http、velocity 等。

总是可用类型是：default、region、resource、direct。

direct 是最简单的类型。put 标记的 content 属性值采用这种类型，或直接把 put 标记之 body 中的字符串插入到模板中。

使用 region 类型可以使得内容能既可以是区块，也可以是作为节内容包含到别的区块中的区块。

resource 类型用来指定当前 webapp 相关资源（一个 JSP、Servlet 或其他 webapp 资源）的 content 属性值。

default 类型是唯一不具 direct 意义的类型。当类型设定为 default（或者没有指定类型）时，将在区块名列表中查找 content 属性值，如果找到就把该区块作为节的内容。如果 content 属性值不是一个区块名，那么 content 属性值就被假定是一个资源，并且区块框架会在当前 webapp 中查找该命名资源。

3、创建区块模板

1）简介

创建一个区块模板最简单的方式是：创建一个使用区块标记库的 JSP（在区块中使用节）。JSPs 模板能象任何其他 JSP 一样使用，但是最好保持他们的简单，而把更复杂内容放到 webapp 资源（webapp 资源当作节内容包含）里。

正象一个边注，"stack"区块保存在一个 request 属性里，理论上，通过它可以创建一个 servlet 来作模板，或者为 velocity 开发一个工具，或者把其他模板机制作为区块模板。

2）标签

使用区块的第一步是，象下面的 JSP 一样用标记声明准备使用区块标记库：

```
<%@ taglib uri='regions' prefix='region' %>
```

"regions" uri 是来自 web.xml 文件的一个声明（采用标准方式声明）。

象下面一样，你可以指定在哪里插入每节内容，或在哪里使用 region:render 标记：

```
<region:render section='header'/>
```

在区块标记库中还有其他一些标记，能用来定义区块（象 regions.xml 文件里 XML 元素一样），并且可以指定到区块的每节内容

四、JSP 标签库指南

1、简介

OFBiz 的 JSP 标记库由一组自定义标记组成，利用这些标记可以很容易地使用 OFBiz 核心框架的其他构件。

然而有一些基本的条件性、循环控制、i18n 和其他通用标记也能在更通用的标记库如 Jakarta 中找到。我们也包含这些标记主要是为了使用方便，并且在缺省情况下 OFBiz 的这些标记对用户更友好。

下面是 OFBiz 的标记表和一些基本参考信息。在文档的其他内容将详细解释每个标记。

标记名	Body 内容	标记类	TEI 类
url	JSP	UrlTag	
contenturl	JSP	ContentUrlTag	
if	JSP	IfTag	
unless	JSP	UnlessTag	
iterator	JSP	IteratorTag	IteratorTEI
iteratorNext	JSP	IterateNextTag	IterateNextTEI
iteratorHasNext	JSP	IteratorHasNextTag	
format	JSP	FormatTag	
print	empty	PrintTag	

field	empty	EntityFieldTag	
entityfield	empty	EntityFieldTag	
inputvalue	empty	InputValueTag	
i18nBundle	JSP	I18nBundleTag	
i18nMessage	JSP	I18nMessageTag	
i18nMessageArgument	empty	I18nMessageArgumentTag	
service	JSP	ServiceTag	
param	empty	ParamTag	
object	JSP	ObjectTag	ObjectTEI

2、URL 标记

1）ofbiz:url

如果有必要，可以添加控制路径和 URL 编码。这是为了使让同一个 webapp 和 body 可以动态地链接到页面，意味着包含"/"符号，然后请求名在 controller.xml 文件中定义。参数也将包含在这个标记里。

这个标记将自动地插入当前 webapp 和控制 servlet 的安装点。

这个标记也可以创建一个安全的（HTTPS）URL（一个指到服务器安全端口的 URL）自动地从不安全区域进入到安全区域，并且将创建一个简单或非安全的（HTTP）URL 来自动地从站点的安全区域转到不安全区域。这将在 url.properties 文件里进行配置。

这个标记没有属性。

2）ofbiz:contenturl

为这个页面引用的内容建立一个 URL。这些通常是些静态内容，如存储在不同服务器上的图象。所以，如果必要的话，可以为静态内容创建一个完整的 URL，同时要考虑内容是被安全地（用 HTTPS）引用，还是不安全地（用 HTTP）引用。

这个标记这也在 url.properties 文件里进行配置，象核心配置向导里描述那样。这个标记没有属性

3、条件性标记

1）ofbiz:if

如果该属性有值存在，并且满足设定的 size 和 value 条件时，本标记就可以用来处理标记的 body 内容。

属	必	Exp	描述

性名	须?	rVal?	
name	Y	Y	页面、请求、会话的名称，或用来对比的应用属性名（本属性里面的值叫做命名
type	N	N	对象的类型。不必要指定这个属性，因为标记在运行时会测试对象的类型。只有本属性
value	N	Y	如果设定本属性，那么命名属性值将与这个值进行比较。可以使用<%=...%>语法
size	N	N	如果指定这个值，那么将与命名属性值的大小比较。这适用于大部分类型应用，

2）ofbiz:unless

与 if 标记作用正好相反。换句话说，如果命名属性值不存在，或者命名属性值存在但不满足 size 和 value 条件时，那么 body 内容是被估计出来的。

unless 标记的属性与 if 标记的属性相同。这些属性的详细信息参阅上面 if 标记内容。

4、循环控制标记

1）ofbiz:iterator

Iterator 标记用来在一个命名属性的 collection 对象之元素集合中进行循环。独立元素将放在 pageContext 属性里，并且在 JSP 的 Java 上下文（用在 scriptlets 中）中可用。

属性名	必须?	ExprVal?	描述
name	Y	N	元素集合名。可以用在页面、请求、会话或应用上下文（本属性里面的值叫做命
property	N	N	把独立元素的属性名和 Java 变量名作为 iterator 进行循环
type	N	N	集合的元素类型。必须指定才能创建 Java 上下文变量。缺省值为 org.ofbiz.core.e 更常见的对象之一
expandMap	N	N	如果 expandMap 为真，那么 iterator 元素必须实现 Map 接口。Map 入口的关键字 性值，那么每个 map 入口将放在 pageContext 内。值必须是 true 或 false。缺省为 fal
offset	N	Y	如果设定本属性，那么循环将从设定的偏移值开始，而不是从集合的启始点开始
limit	N	Y	如果设定本属性，那么只作有限次循环。换句话说，这限制了从 iterator 提取元

子元素名	多少	描述
iterateNext	0 对多	下面说明

iteratorHasNext	0 对多	下 面 说明

2）ofbiz:iteratorNext

在 iterator 标记的集合里取得下一个元素。

属 性名	必须?	ExprVal?	描述
name	N	N	用来放元素。缺省是 iterator 标记的 property 属性指定的名
type	N	N	集合的元素类型。必须指定才能创建 Java 上下文变量。缺省值为 org.ofbiz.core.e 更常见的对象之一
expandMap	N	N	如果 expandMap 为真，那么 iterator 元素必须实现 Map 接口。Map 入口的关键字 性值，那么每个 map 入口将放在 pageContext 内。值必须是 true 或 false。缺省为 fal

3）ofbiz:iteratorHasNext

当 iterator 标记还有下一个入口时，才对 body 进行处理。这个标记没有属性。

5、数据表示标记

1)ofbiz:format

根据本标记的 body 值对日期、数值、货币进行格式化。它有一个属性用来指定文本将如何被格式化。

属性名	必须?	ExprVal?	描述
type	N	N	确定可用作 body 值的类型将如何格式化。值必须是“货币”、“数值”或“日期”。

2)ofbiz:print

打印来自 pageContext 的一个属性。设定为 null，则打印缺省值；如果连缺省值也没指定，那么什么都不打印。

属性名	必须?	ExprVal?	描述
attribute	Y	Y	准备打印的属性名
def	N	Y	找不到指定的属性时，缺省打印的值

ault			

3）ofbiz:field

在指定属性里，根据字段显示一个正确格式化的字符串。

属性名	必须?	ExprVal?	描述
attribute	Y	N	打印包含这个字段的属性名
type	N	N	（可选地）指定打印字段的类型。可以是“货币”或任何 Java 字类型
default	N	Y	找不到指定的字段时，缺省打印的值
prefix	N	N	如果结果信息不为空，那么在字段值前面打印这个字符串
suffix	N	N	如果结果信息不为空，那么在字段值后面打印这个字符串

4）ofbiz:entityfield

在指定属性的一个实体中，根据字段显示一个本地化正确的字符串。当 prefix 和 suffix 属性值不为空时，其内容将在字段值之前/之后打印。

属性名	必须?	ExprVal?	描述
attribute	Y	N	打印包含字段的实体的属性名
field	Y	N	打印命名实体的字段名
type	N	N	（可选地）指定打印字段的类型。可以是“货币”或任何 Java 字段类型。如果没指引擎的字段定义得到附加类型信息。通过这些信息，货币就会自动地格式化成货币
default	N	Y	找不到指定的字段时，缺省打印的值
prefix	N	N	如果结果信息不为空，那么在字段值前面打印这个字符串
suffix	N	N	如果结果信息不为空，那么在字段值后面打印这个字符串

5）ofbiz:inputvalue

从实体字段或请求参数给输入框输出的一个字符串。检查 entityAttr 是否存在，根据指定的属性值来确定使用实体字段还是请求参数。

如果被 tryEntityAttr 属性引用的 Boolean 对象值为 false，那么总是试图使用请求参数，忽略实体字段。

采用一个非常简单的 toString 来进行格式化。

在上下文属性中根据 entityAttr 里的名称查找到的对象，可以是一个 GenericValue 对象或 Map 接口的一个实现。

属性名	必须?	ExprVal?	描述
field	Y	N	打印命名实体的字段名
param	N	N	与这个值相应的 URL 参数名。如果为 null 或空，那么参数将与字段相同
entityAttr	Y	N	打印包含字段的实体的属性名
tryEntityAttr	Y	N	包含一个布尔值的上下文属性名，指定是否试图使用指定的实体。如果属性名为 false，则使用请求参数；如果属性名为 true，将使用实体字段值
default	N	Y	指定实体字段或参数没找到时，打印的缺省值
fullattrs	N	N	如果 fullattrs 属性设置为 true，那么将用[name="{param}" value="{value}"]代替[value]进行输出。fullattrs 缺省为 false

6、国际化标记

1）ofbiz:i18nBundle

用来设定和加载一个国际化信息包。如果指定 id，那么包将通过 id 的值来确定。缺省地，所有信息都从包（由 i18nBundle 标记指定）取得，换句话说 i18nMessage 标记将嵌套在 bundle 标记内。

属性名	须?	ExprVal?	描述
Id		Y	一个 ID，随后可以用在 i18nMessage 标记的 bundleId 属性中，用来引用包
baseName		Y	信息包的资源基础名。通常是一个.properties 文件的名，但指定没有.properties 扩展名

2）ofbiz:i18nMessage

从一个包或特定包（由主键标识）打印一个国际化信息。使用嵌套的 i18nMessageArgument 标记来接受信息参数。

属性名	须?	ExprVal?	描述
Key		Y	包中信息的主键或 ID

bundleId		Y	从中提取信息的包的 ID。如果不指定，则使用 i18nBundle 包

3）ofbiz:i18nMessageArgument

给 i18nMessage 标记指定一个参数。arguments 将根据指定的顺序编号。

属性名	须?	ExprVal?	描述
value		Y	插入到信息的参数值。可以使用<%=...%>语法来使用 Java 方法上下文变量（Java method context variable）

7、服务标记

1）ofbiz:service

Service 标记采用同步或异步模式调用命名服务。根据 resultTo 属性的设定，结果可以放在页面、请求、会话或应用里。缺省为页面。

为了设定参数和返回值有关信息，可以嵌套使用 param 标记。将在下面进行描述。

关于服务的更多信息，请看服务引擎指南和服务引擎配置指南。

属性名	须?	ExprVal?	描述
name		Y	调用的服务名
mode		Y	调用服务的方式-同步或异步。必须是 sync 或 async，缺省为 sync
resultTo		Y	指定结果输出的的范围。把服务的 OUT 属性/参数放进本属性中。值必须是 page、request、session 或 application。缺省是 page

2）ofbiz:param

param 标记用来设定传递给服务的参数的相关信息，并且可以用来指定返回值的某些信息。

属性名	须?	ExprVal?	描述
name		Y	服务参数名
value		Y	分配给 IN 参数的值。可以使用<%=...%>语法来使用 Java 方法上下文变量（Java method context variable）

mode		Y	指定参数的方式。取值必须是 IN、OUT 或 INOUT。缺省为 IN。如果设置为 OUT 或 INOUT，那么 alias 属性可以用来覆盖在 service 标记指定范围里上下文属性的缺省关键字
map		Y	为一个 Map 指定一个名字，用来在上下文属性里进行查找。如果设定了这个属性，那么它将用来在 map 里查找一个值。如果不设定这个属性，那么属性将用来在上下文属性里查找一个值
attribute		Y	指定一个名称供查找之用。它是用来代替 value 属性的，value 属性用来直接为这个参数指定传递给服务的值。找到的属性值将传递给满足这个参数的服务
alias		Y	用来指定主键（把 OUT 或 INOUT 参数放进上下文属性时使用）。缺省为 name 属性的值

8、其他标记

1）ofbiz:object

使上下文属性（页面、请求、会话或应用）里的对象在 JSP 的 Java 方法上下文里可用，这样它可以用在 Java scriptlets 中。

属性名	必须?	ExprVal?	描述
name	Y	N	要创建的 Java 方法上下文变量（Java method context variable）名
property	N	N	带值的上下文属性名，如果不指定，则与 name 属性相同
type	N	N	对象的 Java 类型。如果不指定，缺省为 org.ofbiz.core.entity.GenericValue

第二部分、ofbiz 服务

一、服务引擎指南

1、简介

服务框架是 OFBiz 2.0 新增加的功能。服务定义为一段独立的逻辑程序，当多个服务组合在一起时可完成不同类型的业务需求。服务有很多类型：Workflow, Rules, Java, SOAP, BeanShell 等。Java 类型的服务更像一个静态方法实现的事件，然而使用服务框架就不会局限在 Web 应用程序中。服务需要使用 Map 传入参数，结果同样从 Map 中返回。这样很妙，因为 Map 可以被

序列化并保存或者通过 HTTP(SOAP)传输。服务通过服务定义来定义并指派给具体的服务引擎。每个服务引擎 通过适当的方式负责调用服务定义。因为服务没有和 web 应用程序帮定在一起，就允许在没有响应对象可用时仍可以执行。这就允许在指定时间由 工作调度程序 在后台调用服务。

服务能调用其他服务。因此，将多个小的服务串联起来实现一个大的任务使重用更容易。在不同应用程序中使用的服务可以通过创建全局服务定义文件(只能创建一个)或者一个应用程序的特定服务(这样的服务受限制且只能用于这个应用程序)。

当在 web 应用程序中使用时，服务可以用于 web 事件，这允许时间在服务框架中(stay small?)并重用现成的逻辑。同样，服务可以定义成 'exportable'，允许外部程序访问。目前，SOAP EventHandler 允许服务通过 SOAP 来产生。其他形式的远程访问将来会加入到框架中。

2、Service Dispatcher

Service Dispatcher 将需要处理的服务分配给适当的服务引擎，在那里服务被调用。每个 Entity Delegator 都有一个具体的 ServiceDispatcher 。应用程序中如果有多个 delegator 也应该有多个 ServiceDispatcher 。通过 LocalDispatcher 访问 ServiceDispatcher 。可能有多个 LocalDispatcher 关联到一个 ServiceDispatcher 上。每个 LocalDispatcher 都是唯一命名的并包含自己的一些服务定义。当创建 LocalDispatcher 的一个实例，一个 DispatchContext 实例也会被创建并被传给服务引擎。

一个 LocalDispatcher 和一个应用程序关联。应用程序永远不会直接和 ServiceDispatcher 对话。LocalDispatcher 包含一个 API 用来调用服务，这些服务通过 ServiceDispather 发送。然而，应用程序可能运行于不同的线程和实际的 ServiceDispatcher 中，

，However, applications may be running in different threads then the actual ServiceDispatcher, so it is left to the LocalDispatcher to keep a DispatchContext which among other things keeps a reference to the applications classloader.

3、Dispatch Context

DispatchContext 在 LocalDispatcher 实例之上由他创建的。这是运行时 dispatcher 上下文环境。它包含每个 dispatcher 处理服务的必须信息。这个上下文包含到每个服务定义文件的引用，

The DispatchContext is created by the LocalDispatcher upon instantiation. This is the runtime dispatcher context. It contains necessary information to process services for each dispatcher. This context contains the reference to each of the service definition files, the classloader which should be used for invocation, a reference to the delegator and its dispatcher along with a 'bag' of user defined attributes. This context is passed on to each service when invoked and is used by the dispatcher to determine the service's model.

4、服务引擎

是服务实际被调用的地方。每个服务定义都要定义一个引擎名。引擎名定义在 servicesengine.xml 文件中当调用时通过 GenericEngineFactory 实现。支持第三方引擎，但是必须实现 GenericEngine 接口。参照实体引擎配置指南看定义引擎的详细信息。调用同步和异步

服务是引擎的工作。使用 Job Scheduler 调用异步服务的引擎可以从 GenericAsyncEngine 派生得到。

5、服务定义

服务定义在服务定义文件中。有全局(global)定义文件，所有服务派遣者都可以调用，同时也有只和单一服务派遣者相关联单独服务定义文件。当 LocalDispatcher 被创建，他会传递指向服务定义文件的 Arils 的一个集合。这些文件由 XML 写成，并定义了调用一个服务的必须信息。请参照相关 DTD 文件。

Services are defined in *Service Definition Files*. There are global definition files used for all service dispatchers as well as individual files associated only with a single dispatcher. When a LocalDispatcher is created it is passed a Collection of Arils which point to these definition files. These files are composed using XML and defined the necessary information needed to invoke a service. The DTD of this file can be found here.

服务定义有一个唯一名字，明确的服务引擎名，明确定义的输入输出参数。下面是个例子。

```
<service name="userLogin" engine="java"
location="org.ofbiz.commonapp.security.login.LoginServices" invoke="userLogin">
<description>Authenticate a username/password; create a UserLogin object</description>
<attribute name="login.username" type="String" mode="IN"/>
<attribute name="login.password" type="String" mode="IN"/>
<attribute name="userLogin" type="org.ofbiz.core.entity.GenericValue" mode="OUT"
optional="true"/>
</service>
```

SERVICE 元素:

- **name** - 服务的唯一名字。
- **engine** - 服务引擎的名字 (在 servicesengine.xml 中定义)
- **location** - 服务类的包或其位置。
- **invoke** - 服务的方法名。
- **auth** - 服务是否需要验证(true/false)
- **export** - 是否通过 SOAP/HTTP/JMS (true/false) 访问。
- **validate** - 是否对下面属性的名字和类型进行验证(true/false)
- **IMPLEMENTS 元素:**
- **sevice** - 这个服务实现的服务的名字。所有属性都被继承。
- **ATTRIBUTE 元素:**
- **name** - 这个属性的名字
- **type** - 对象的类型 (String, java.util.Date, 等。)
- **mode** - 这个参数是输入、输出或输入输出类型。(IN/OUT/INOUT)
- **optional** - 这个参数是否可选(true/false)

*下划线标注的值是默认值。

由上面可以看出服务名是 *userLogin*，使用 *java* 引擎。这个服务需要两个必须的输入参数：*login.username* 和 *login.password*。必须的参数在服务调用之前会做检验。如果参数和名字及对象类型不符服务就不会被调用。参数是否应该传给服务定义为 **optional**。服务调用后，输出参数也被检验。只有需要的参数被检验，但是，如果传递了一个没有定义为可选的参数或者必须的参数没有通过校验，将会导致服务失败。这个服务没有要求输出参数，因此只是简单返回。

6、用法

服务框架内部的用法非常简单。在 Web 应用程序中，LocalDispatcher 被保存在 ServletContext 中，ServletContext 可以在事件中通过访问 Session 对象来访问。对不是基于 web 的应用程序仅仅创建了一个 GenericDispatcher。(在 web.xml 中可以找到)

GenericDelegator delegator = GenericDelegator.getGenericDelegator("default");
LocalDispatcher dispatcher = new GenericDispatcher("UniqueName", delegator);

现在我们有了 dispatcher ，可以用来调用服务。为了调用这个服务，为 context 创建一个 Map 包含一个输入参数 *message*，然后调用这个服务：

```
Map context = UtilMisc.toMap("message","This is a test.");
Map result = null;
      try {
        result = dispatcher.runSync("testScv", context);
}
catch (GenericServiceException e) {
  e.printStackTrace();
}
if (result != null)
  System.out.println("Result from service: " + (String) result.get("resp"));
```

现在查看控制台看测试服务的回复信息。

*** The test service is located in core/docs/examples/ServiceTest.java you must compile this and place it in the classpath.

安排一个服务在稍晚点时间运行或者重复使用：

```
      // This example will schedule a job to run now.
Map context = UtilMisc.toMap("message","This is a test.");
try {
  long startTime = (new Date()).getTime();
  dispatcher.schedule("testScv", context, startTime);
}
catch (GenericServiceException e) {
  e.printStackTrace();
}

// This example will schedule a service to run now and repeat once every 5 seconds a total of
10 times.
Map context = UtilMisc.toMap("message","This is a test.");
try {
  long startTime = (new Date()).getTime();
  int frequency = RecurrenceRule.SECONDLY;
  int interval = 5;
  int count = 10;
  dispatcher.schedule("testScv", context, startTime, frequency, interval, count);
}
```

```
catch (GenericServiceException e) {
  e.printStackTrace();
}
```

二、高级特性

服务引擎中加入了很多'高级'特性，在下面有例子、定义及信息。

1、接口

interface 服务引擎实现了在定义服务时可以共享同样的参数。一个接口服务不能被调用，而是为其他服务继承而定义的。每个接口服务都需要用 interface 引擎来定义：

```
<service name="testInterface" engine="interface" location="" invoke="">
<description>A test interface service</description>
<attribute name="partyId" type="String" mode="IN"/>
<attribute name="partyTypeId" type="String" mode="IN"/>
<attribute name="userLoginId" type="org.ofbiz.core.entity.GenericValue" mode="OUT" optional="true"/>
</service>
```

**注意到 location 和 invoke 和在 DTD 中定义为必须的，因此用做 interface 时使用空串。

现在定义一个服务来实现这个接口

```
<service name="testExample1" engine="simple"
    location="org/ofbiz/commonapp/common/SomeTestFile.xml"
                                                invoke="testExample1">
    <description>A test service which implements testInterface</description>
    <implements service="testInterface"/>
</service>
```

testExample1 服务将会和 testInterface 服务拥有完全一样的需要或可选的属性。任何实现 testInterface 的服务都将继承其参数/属性。如果需要给指定的服务增加附加属性，可以在 implements 标签后面跟上 attribute 标签。可以在 implements 标签后面重定义某个属性达到重写一个属性的目的。

2、ECAs

ECA (Event Condition Action) 更象是一个触发器。当一个服务被调用时，会执查看是否为这个事件定义任何 ECAs 。在验证之前，检验之前，事件在实际调用之前，在输出参数校验之前，在事务提交之前或者在服务返回之前包含进来。然后每个条件都会进行验证，如果全部返回为真，定义的动作就会执行。一个动作就是一个服务，该服务的参数必须已经存在于服务的上下文中。每个 ECA 可以定义的条件数或者动作数没有限制。

```
<service-eca>
    <eca service="testScv" event="commit">
```

```
            <condition field-name="message" operator="equals" value="12345"/>
            <action service="testBsh" mode="sync"/>
        </eca>
    </service-eca>
```

eca 标签：

属性名	需要?	描述
service	Y	ECA 关联的服务名
event	Y	ECA 在哪个事件上或之前执行。事件有:auth, in-validate, out-validate, invoke, commit, 或者 return。
run-on-error	N	当有错误时是否执行 ECA (默认为 false)

eca 元素应该有 0 或多个 condition/condition-field 元素，1 或多个 action 元素。

condition 标签

属性名	需要?	描述
map-name	N	本服务上下文属性的名字，包含要检验的字段名字组成的 Map。如果没有指定这个域名，就会使用服务上下文环境（env-name）。
field-name	Y	要比较的 map 的名字。
operator	Y	指定比较操作符，必须为 less, greater, less-equals, greater-equals, equals, not-equals, 或 contains 当中的一个。
value	Y	字段要比较的值。必须为 String,但是可以转换成其他类型。
type	N	用来进行比较的数据类型。必须是 String, Double, Float, Long, Integer, Date, Time, 或 Timestamp 当中的一个。
format	N	指定当将 String 转换成其他类型(主要是 Date, Time 和 Timestamp)时使用的格式说明。

condition-field 标签

属性名	需要?	描述
map-name	N	本服务上下文属性的名字，包含要检验的字段名字组成的 Map。如果没有指定这个域名，就会使用服务上下文环境（env-name）。
field-name	Y	要比较的 map 的名字。
operator	Y	指定比较操作符，必须为 less, greater, less-equals, greater-equals, equals, not-equals, 或 contains 当中的一个。
to-map-name	N	本服务上下文属性的名字，包含要与之比较的字段名字组成的 Map。如果为空就会使用上面的 map-name，如果 map-name 也为空，就会使用服务下文环境（env-name）。
to-field-name	N	要与之比较的 Map 中要比较的字段名，如果为空默认为上面 field-name。

type	N	用来进行比较的数据类型。必须是 String, Double, Float, Long, Integer, Date, Time, 或 Timestamp 当中的一个。如果没指定默认为 String。
format	N	指定当将 String 转换成其他类型(主要是 Date, Time 和 Timestamp)时使用的格式说明。

action 标签

属性名	需要?	描述
service	N	本动作(action)要调用的服务名。
mode	Y	调用服务的方式，可以是 sync 或 async。async actions 将不会更新 context 即使 result-to-context 设置为 true.
result-to-context	N	action 服务的执行结果是否更新服务的上下文(context)，默认为 true。
ignore-error	N	是否忽略本 action 服务导致的错误，如果否原始服务就会失败。默认为 true。
persist	N	action 服务 store/run。可以为 true 或 false。 只有当 mode 属性为 async 时才生效。默认为 false。

3、服务组

服务组是由多个服务组成的服务集，当调用初始化组服务时应该运行。使用组服务定义文件定义一个组服务，包含这个组服务所有服务需要的参数/属性。location 属性不需要，invoke 属性定义了要运行的组服务名。当这个组服务被调用时，组中定义的所有服务都会被调用。

组的定义很简单，他包含一个拥有多个 service 元素的 group 元素。group 元素包含一个 name 属性和一个 mode 属性， mode 用来定义服务怎么执行。service 元素更像 ECA 中的 action 元素，不同之处在于 result-to-context 属性的默认值。

```
<service-group>
<group name="testGroup" send-mode="all">
    <service name="testScv" mode="sync"/>
    <service name="testBsh" mode="sync"/>
</group>
</service-group>
```

group 标签

属性名	需要?	描述
name	Y	要调用的服务组的名字。
send-mode	N	这些服务被调用的方式。可为：none, all, first-available, random, 或 round-robin。默认为 all。

service 标签

属性名	需要?	描述
service	N	这个服务组要调用的服务名。

mode	Y	这个服务被调用的方式。可为 sync 或 async。async 将不会更新 context 即使 result-to-context 设置为 true。
result-to-context	N	组服务执行的结果是否更新服务的上下文(context)，默认为 false.。

4、路由服务(Route services)

路由服务使用路由服务引擎定义。当一个路由服务被调用时，不会执行调用，但是所有定义的 ECA 会在适当事件中运行。这种类型的服务不常用，但是通过利用 ECA 服务选项可以路由('route') 到其他服务。

5、HTTP 服务

使用 HTTP 服务是调用定义在其他系统上远程服务的一种方法。本地定义应该和远程定义一致，但是引擎应该是 http，location 应该是 httpService 事件在远程系统上运行的完全 URL，方法应该是远程系统上被调用运行的服务名。远程系统必须有挂在 HTTP 服务上公允的 httpService 事件。默认情况下，commonapp web 应用程序有用来接收服务请求的这样的事件。在远程系统上的服务必须将 export 属性设为 true 允许远程调用。HTTP 服务本质就是同步的。

6、JMS 服务

JMS 服务和 HTTP 服务很相似，除了服务请求被发送到 JMS topic/queue。engine 属性应该设置为 jms，location 属性因该设置为在 serviceengine.xml 文件中定义的 JMS 服务名(服务配置)。方法应该是你请求要执行的远程系统上的 JMS 服务名。本质就是异步的。

三、服务引擎配置指南

1、简介

本篇文章描述服务引擎的设置。开始介绍总体思想，然后介绍 serviceengine.xml 的每部分并结识可用的元素及其用法。serviceengine.xml 文件为不同用途提供有例子，文件位于 **ofbiz/commonapp/etc/serviceengine.xml**。
服务引擎的设置通过一个叫做 **serviceengine.xml** 的简单 XML 文件来完成，必须位于 classpath 某处。

2、验证

authorization 标签设置服务授权需要调用的服务。这个标签只有一个属性 service-name；属性值应该是用来授权的服务名。默认定义为使用通用 OFBiz userLogin 服务。

3、线程池

工作调度器(job scheduler)异步调用工作/服务。它包含池化的线程和几个请求线程。**thread-pool** 标签用来配置每个线程怎么操作。有如下属型可用。

属性名	需要?	描述
Ttl	Y	每个请求线程的存活时间。达到时间线程将被销毁。
wait-millis	Y	每个请求线程在检查通过运行前休眠的时间。
jobs	Y	每个请求线程在销毁之前可运行的工作数。
min-threads	Y	线程池中保持的请求线程的最小数。
max-threads	Y	线程池中将会创建请求线程的最大数。
poll-enabled	Y	为'true'scheduler 就会 poll 数据库来调度工作。
poll-db-millis	Y	如果线程池可用，本属性用来定义池化线程运行的频率。

4、引擎定义

每一个 GenericEngine 接口的实现都需要在服务定义中定义，**engine** 标签有如下属性：

属性名	需要?	描述
name	Y	服务引擎的名字。必须唯一。
class	Y	GenericEngine 接口的实现类。

5、资源加载器

resource-loader 标签用来设置一个指定的资源加载器以在其他地方加载 XML 文件和其他资源。有如下属性：

属性名	需要?	描述
name	Y	资源加载器的名字。用于其他标签的 'loader' 属性。
class	Y	通用抽象类 org.ofbiz.core.service.config.ResourceLoader 的扩展类。可用类包括

		FileLoader, UrlLoader, 和 ClasspathLoader，同类 ResourceLoader 位于同一个包中。
prepend-env	N	Java 环境属性的名称。用来放在全部路径(full location)比较靠前的地方，在前缀前面。可选。
prefix	N	当拼装全部路径(full location)时，放在前面的字符串。可选。如果使用了 prepended 环境属性，将会置于其后并位于每个指定资源位置的前面。

6、全局服务

global-services 标签用来定义服务定义文件的位置。有如下属性：

属性名	需要?	描述
loader	Y	前面 **resource-loader** 标签定义的资源加载器。
location	Y	指明资源加载器加载资源要使用的文件的位置。

7、服务组

service-groups 标签用来定义服务组定义文件的位置。有如下属性：

属性名	需要?	描述
loader	Y	前面 **resource-loader** 标签定义的资源加载器。
location	Y	指明资源加载器加载资源要使用的文件的位置。

8、ECAs

service-ecas 标签用来定义服务条件触发动作定义文件的位置。有如下属性：

属性名	需要?	描述
loader	Y	前面 **resource-loader** 标签定义的资源加载器。
location	Y	指明资源加载器加载资源要使用的文件的位置。

9、JMS

jms-service 标签为 JMS 定义服务的位置。

属性名	需要?	描述
name	Y	JMS 服务的名字，在服务定义中作为 location 的值。
send-mode	Y	向定义的服务发送的模式有：none, all, first-available, random, round-robin, 或 least-load。

jms-service 可以包含一个或多个 **server** 标签， **server** 标签有如下属性：

属性名	需要?	描述
jndi-server-name	Y	在 jndiservers.xml 文件中定义的 JNDI 服务名字。
jndi-name	Y	在 JNDI 中为 JMS 工厂定义的名字。
topic-queue	Y	主题或队列(topic or queue)的名字。
type	Y	JMS 类型可能为主题或队列(topic or queue)。
username	Y	连接主题/队列(topic/queue)的用户名。
password	Y	连接主题/队列(topic/queue)的密码。
listen	Y	设置是否对主题/队列(topic/queue)起用监听。

在 jndiservers.xml 文件中定义的 jndi-server 应该指出 JMS 客户端 APIs 的位置。根据定义的 JMS 类型来决定使用 TopicConnectionFactory 或 QueueConnectionFactory 。JNDI 名字应该指出在 JNDI 中包含连接工厂实例的对象名字。

三、编程指南

四、编程参考

第三部分、ofbiz 实体

一、实体引擎配置指南

1、介绍

本篇文章描述实体引擎配置。先介绍整体思想，然后分别 entityengine.xml 文件各部分的可用元素及用法。这个文件为不同的用途提供了一些例子，文件位于 **ofbiz/commonapp/etc/entityengine.xml**。

通过一个简单的 XML 文件 **entityengine.xml** 来配置实体引擎。该文件必须位于 classpath 某处。这个文件用来定义维持服务需要的参数，比如 EJB 服务参数、JDBC 服务参数。他也用来指定使用那个实体模型、实体组及服务用到的字段类型模型文件，OFBiz 发版默认的配置文件在 **ofbiz/commonapp/etc/entityengine.xml** 处可找到。

使用实体引擎的每个应用程序都需要通过一个 **GenericDelegator** 类的实例来达到目的。如果新产生一个实例,delegator 的名字必须传给静态工厂方法以传给 GenericDelegator 构造函数。delegator 名称用来在 **entityengine.xml** 中查找相关设置。**entityengine.xml** 包含将每个实体组映射到这个实体组 **GenericHelper** 的配置信息。**GenericHelper** 是一个接口，必须为每种数据源(JDBC,EJB,SOAP,HTTP 等等)实现这个借口的一个 Helper 类。

每个特定的 **GenericHelper** 的设置信息都在 **entityengine.xml** 文件的 **datsource** 元素指定。对 JDBC 帮助类，包括数据库连接参数，比如 JNDI 数据源参数或包括数据库驱动程序名、JDBCURL、用户名、密码的 JDBC 参数。一个 EJB 帮助类将会包含 JNDI 参数，比如上下文相关的 URL、初始的上下文工厂，及 URL 包前缀(suchasthecontextproviderURL,theinitialcontextfactory,andtheURLpackageprefixes)。

每个 delegator 使用一个实体模型和一个实体模型加载器(entitymodelreader)和一个实体组加载器(entitygroupreader)。

GenericDelegator 是实体引擎服务中最主要的访问途经。每个服务请求被分派到和实体对应的帮助类，服务被请求用来和这个实体的实体组通信。OFBiz 实体模型默认的实体组 XML 文件可以在 **ofbiz/mmonapp/entitydef/entitygroup.xml**.中找到。

2、资源加载器

resource-loader 标签用来配置指定的资源加载器以在其他地方加载 XML 文件和其他资源。有如下属性：

属性名	需要?	描述
name	Y	资源加载器的名称。在其他标签的'loader'属性中使用。
class	Y	要使用的类，从抽象类 org.ofbiz.core.entity.config.ResourceLoader 派生来。可用的有 FileLoader、UrlLoader、ClasspathLoader，和 ResourceLoader 在同一个包里。
prepend-env	N	Java 环境属性的名称。用来放在全部路径(fulllocation)比较靠前的地方，在前缀前面。可选。
prefix	N	当拼装全部路径(fulllocation)时，放在前面的字符串。可选。如果使用了 prepended 环境属性，将会置于其后并位于每个指定资源位置的前面。

3、JTA 元素

为支持 JTA，实体引擎需要访问 **javax.transaction.UserTransaction**，也可以访问 **javax.transaction.TransactionManager** 接口，为使用 TXManager 实现的。这些通过类 **org.ofbiz.core.entity.TransactionFactory** 实现。

ForJTAsupporttheEntityEngineneedsaccesstothe**javax.transaction.UserTransaction**andoptionallythe**javax.transaction.TransactionManager**interfaceimplementationsfortheTXManagerbeingused.The

seareretrievedthroughtheorg.ofbiz.core.entity.TransactionFactoryclass.Thisclassusestheclassspecifiedwiththeclassattributeofthetransaction-factory.这个类可能会依据 OFBiz 运行使用的应用程序服务器或事务管理器的不同而改变。

默认的 TX 管理器使用 Tyrex，来自 Exolab(www.exolab.org)。Tyrex 的工厂类是 **org.ofbiz.core.entity.transaction.TyrexFactory**。同样 Weblogic 有一个专门的工厂类：**org.ofbiz.core.entity.transaction.WeblogicFactory**,他需要修改以包含 Weblogic 详细准确的代码(stuff)并和位于 classpath 上的 weblogic.jar 一起编译。

最有用处的事务工厂类是 **org.ofbiz.core.entity.transaction.JNDIFactory** 类。这个类使用 entityengine.xml 中附加元素来定位 JNDI 中的 UserTransaction 和 TransactionManager 对象。**user-transaction-jndi** 和 **transaction-manager-jndi** 标签用来指定对象在 JNDI 中的名称以及要使用 JNDI 服务的名称。两个标签都有两个必须的属性：**jndi-server-name** 和 **jndi-name**。一个例子：UserTransaction 对象的 jndi-name 值为 **java:comp/UserTransaction**，TransactionManager 对象的 jndi-name 值为 **java:comp/TransactionManager**。

4、Delegator 元素

GenericDelagator 通过一个使用一个包含 delegator 名称的 String 类型的参数的工厂类方法产生。delegator 名称用来在 entityengine.xml 文件中查找 **delegator** 标签。

属性名	需要?	描述
name	Y	Delegator 名称，使用这个名称用来查找这个标签。
entity-model-reader	Y	delegator 使用的 entity-model-reader 名称。
entity-group-reader	Y	delegator 使用的 entity-group-reader 名称。

delegator 标签必须包含一个或多个 **group-map** 标签来为 delegator 从实体引擎加载器知道每组实体指定一个数据源。delegator 使用这个文件为所有实体分组。当对一个实体的操作发生时，delegator 会查找实体组及和实体组通信的数据源帮助类来执行低层数据源操作。采用这种技术实现应用程序，就不需要知道对给定的实体由哪个帮助类负责访问，由此可以实现使用一个简单的配置将一个实体或很多组实体指派给不同的数据源。

5、实体模型 XML 文件

entity-model-reader 标签用来为每个指定的实体模型加载器。标签的 name 属性用来指定实体模型加载器的名字。每个加载器可能从多个资源中加载，其中每个资源使用标签中的 **resource** 标签来指定。

每个 **resource** 标签有两个必须的标签：**loader** 用来指定使用哪个资源加载器，**location** 指定了资源加载器内部加载资源使用的位置。

6、实体组 XML 文件

entity-group-reader 标签用来设置每个指定的实体组加载器。标签的 **name** 属性用来指名实体组加载器的名字。实体组加载器使用一个单独的 XML 文件来获取实体组的映射信息。
这个标签有两个需要的属性：**loader** 用来指定使用哪个资源加载器，**location** 指定了资源加载器内部加载资源使用的位置。

7、字段类型 XML 文件

field-type 标签用来配置每个指定的字段类型。标签的 **name** 属性用来指名字段类型的名字。实体组加载器使用一个单独的 XML 文件来获取字段类型信息。

这个标签有两个需要的属性：**loader** 用来指定使用哪个资源加载器，**location** 指定了资源加载器内部加载资源使用的位置。

8、数据源元素

很多数据源都可以使用 **datasource** 标签设置，每个数据源对应一个数据源。这个标签有如下属性，很多包含如下字属性。

Manydatasourcescanbeconfiguredusingone**datasource**tagforeachdatasource.Thistaghasthefollowingattributes,andmaycontainthefollowingsub-elements:

属性名	需要?	描述
name	Y	数据源的名字
helper-class	Y	可能有许多类型的数据源帮助类，主要的一个是 JDBC/DAO 帮助类。可以实现 **org.ofbiz.core.entity.GenericHelper** 接口编写自己的帮助类。JDBC/DAO 帮助类就是 **org.ofbiz.core.entity.GenericHelperDAO**。
field-type-name	Y	要使用的字段类型的名字。必须和前面 field-type 标签定义的字段类型名字匹配。
check-on-start	N	启动时是否检查数据源？必须为 true 或 false，默认为 true。
add-missing-on-start	N	当启动时检查数据源时，是否要添加不存在的实体？必须为 true 或 false，默认为 false。
use-foreign-keys	N	对"one"类型的关系，是否使用/创建外键？必须为 true 或 false，默认为 true。
use-foreign-key-indices	N	是否对外键使用/创建索引(也就是外键上的索引)?注意到这个属性起作用并不需要创建外键，索引只为定义为"one"类型的关系创建。必须为 true 或 false，默认为 true。
check-fks-on-start	N	启动时是否检查外键，当需要时是否添加？必须为 true 或 false，默认为 false。有些数据库会花费很长时间，并且不会返回所有外键列表，结果导致重复的外键会添加到数据库中。
check-fk-indices-on-start	N	启动时是否检查外键索引，当需要时是否添加？必须为 true 或 false，默认为 false。

use-pk-constraint-names	N	是否为主键使用约束名字？有些数据库对此有问题，但是如果给个名字就回工作正常。必须为 true 或 false，默认为 true。
constraint-name-clip-length	N	指定约束名的最大长度。超出的将被裁掉。这样做时当心重复的约束名。必须为整数，默认为 30。
fk-style	N	指定外键语法(syntax)类型：命名为外键约束或仅仅外键。大多数数据库使用 name_constraint 语法，但是 SAPDB 这样做会出现异常，可能还有其他数据库也会这样。必须为"name_constraint"或"name_fk"，默认为"name_constraint"。
use-fk-initially-deferred	N	指定在许多数据库中，当创建外键时使用 INITIALLYDEFERRED 选项是否可用。不是所有数据库都支持这个选项。当可用且数据库支持时，外键检查只有在事务提交时才会进行，这和在事务中检查外键恰恰相反。必须为 true 或 false，默认为 true。
join-style	N	指定当在 view-entity 操作中做表联接时使用的语法。许多数据库采用标准的 ANSIJOIN，但是在此之前 theta 联接更通用。支持两个 theta 连接类型：Oracle 和 MSSQL。必须为"ansi","theta-oracle"或"theta-mssql"。默认为"ansi"。

子元素名	少?	描述
sql-load-path	到多个	用来指定目录的完整路径列表，用来查找 XML 和 SQL 文件在 **install** 页导入到数据源中 UsedtospecifyalistoffullpathstodirectoriesthatwillbesearchedforXMLandSQLfilestoimportintothedatasourceb
inline-jdbc	或1	用来指定由 Tyrex 使用的 JDBC 参数或者当 Tyrex 不可用时直接加载驱动程序(非常慢)。必须为 DAOHelper 指定
jndi-jdbc	或1	用来指定指定 jndi-server 和 jndi-name 来从 JNDI 获取一个连接或 XAConnection。必须为 DAOHelper 指定 in
ANY	或1	datasource 标签中的任意标签，来为其他 GenericHelper 的实现类指定参数。只有在 DTD 文件中作了修改来描述

inline-jdbc 标签有如下属性：

属性名	需要?	描述
jdbc-driver	Y	数据库 JDBC 驱动程序类。
jdbc-uri	Y	用来指定数据库的类型和位置的 URI。
jdbc-username	Y	连接数据库用的用户名。
jdbc-pass	Y	用户名的密码。

word		
isolation-level	N	用来指定 Tyrex 事物隔离级别(isolationlevel)，标准 JDBC 事务隔离级别有： 'ReadCommitted' 'ReadUncommitted' 'RepeatableRead' 'Serializable'(default)

jndi-jdbc 标签有如下属性：

属性名	需要?	描述
jndi-server-name	Y	要用的 JNDI 服务的名字。
jndi-name	Y	连接或 XAC 连接对象在 JNDI 中的名字。

从 JNDI 重新得到的数据源应该使用连接池技术并可使用事务，如果 JTA 事物启动的话。这可能是 DataSource 对象或者 XADataSource 对象。

如果没有指定 JNDI 元素，连接工厂就会从 entityengine.xml 文件中取到，并试图使用 Tyrex 作为事物管理和连接池。如果 Tyrex 不可用实体引擎将会在每次接收到请求时创建一个连接，这时将会有警告信息。

二、实体应用指南

ofbiz 实体引擎是管理实体和实体数据的一系列工具和模式。在这里实体是指用 field（字段）和 relation（与其他实体的关系）定义的一组数据。实体定义依据标准 RDBMS(关系数据库管理系统)中的 Entity-Relation 模型来建立。实体引擎的目标是简化实体数据的使用，包括实体的定义、维护、优化、及和实体有关的功能。

实体引擎使用了一些能够被大多数应用系统所识别的应用模式、中间模式（集成），ofbiz 中也使用了许多表现层的模式，这些模式只在 servlet 控制器中使用，实体引擎不使用。实体引擎中可使用的模式包括：业务委派（Business Delegate），数值对象(Value Object),复合实体(Composite Entity), 数值对象集合器(Value Object Assembler）,服务定位器 Service Locator, 数据访问对象（Data Access Object），这些模式的不断完善和其他模式的实现都在计划之当中。

这些模式在 sun 公司 Alur, Crupi,和 Malks 所写的 “Core J2EE Patterns”中有描述，你也可以在网上，通过查找这两位作者发表的文章获取相关信息。

另外TheServerSide公司的Floyd Marinescu将要发表的"EJB Design Patterns"书中，也有几种可用的模式。它包括数据传递HashMap和概括属性访问，对于产生唯一关键字，我们采用一种称之为"Ethernet Key Generation"的模式，作为我们的初试模式，它具有冲突检测机制，可以保证多台服务器使用同一个数据库以一个数据库独立的方式获取一组唯一性的关键字。

实体引擎的主要目标是，尽量除去交易性应用系统中许多地方要使用的、特殊的实体持久化代码。我们承认对于报表类，或者与之相似的系统，提取出的持久化的方法有所不同，但对于每天都要做的交易性应用系统，实体引擎可以节省大量的开发工作量，并且大量地减少系统中与数据持久化有关的错误。这类应用系统包括电子商务中的帐务系统、配送系统、库存管理系统、人力资源系统等等所有系统。这些工具可用于报表系统和分析系统，但该工具不包含适合于，经常出现的由用户定义的、使用方便的查询。

为了得到尽量小而精的数据实体代码，所有的数值对象都为数据汇集 generic，generic 是一个将数据实体的域值按照名称进行存取的 Map 对象。

依据字符串名称，使用域（fields）方法对实体域进行存取，以的得到或存储所需要的数据，并鉴别字符串名称是否实体中的一个域。这种灵活的坏处是减少了实体引擎和应用程序之间的配合。这种配合在特定的 XML 文件中进行说明。

取代编写特定的代码，实体的定义将从 entitymodel*.XML 文件中读入，并由实体引擎强制规定应用系统和数据源之间的访问规则，这里所指的数据源可以是数据库，也可以是其他的数据源。使用 XML 定义的实体中，每一个实体都有唯一的名称和域，他们和数据库的表和字段一一对应。在实体的定义中，每个域都有一个与数据库字段相类似的类型说明，根据这个类型可以查处它的 java 类型和数据库类型。在实体文件 entitymodel*.XML 中还定义了实体和实体的关系。关系定义中包含一个相关连的数据库表和关系类型（one or many），及关系的关键字值对。关系还可以定义一个标识，以区别实体关系中的其他关系。

在一个抽象层次上使用实体引擎，实体的应用代码可以很容易的建立和修改。这些代码使用实体接口 APIs 与实体进行交互，这些实体可以有多种方式进行持久化，而不需要修改应用代码。比如只要修改应用程序的实体配置文件，就可以将通过 JDBC 的数据库连接，转换到 EJB 服务器和实体 Bean 来进行数据的持久化。也可在同一个框架中使用用户自己特定的数据源，比如 legacy systems over HTTP,或者使用一些自定义的消息服务。

实体模型

The first thing to do when starting work with a new entity is to define or model that entity. In the OFBiz Entity Engine this is done through two XML files, one for entity modeling and the other for field type modeling. There are links to the XML DTDs for these documents in the Related Documents section above. The reason that these two files are separated is that field type definitions can be database specific. In other words, for a given entity model definition various field type definitions may exist for different databases. When a persistence server is defined the field type model XML file to be used for that server is specified.

The main entity model XML files for Open For Business can be found in ofbiz/commonapp/entitydef/. Originally all of the entities were in the entitymodel.xml file, but now they are separated into various files in addition to the entitymodel.xml file. They are all named after the following pattern: entitymodel_*.xml

The MySQL field type model XML file for Open For Business can be found in ofbiz/commonapp/entitydef/fieldtypemysql.xml. There are other database specific field type files for Postgres, Hypersonic, Oracle, et cetera. From the entity model files and the field type files database tables can be created automatically through the checkDataSource routine on the GenericHelper inteface. This can be done automatically on startup or through the tools in WebTools.

While tables can be created automatically, data must be loaded from data files. These files can be either SQL scripts or XML Entity Engine files. All of the type information and other pre-loaded information such as statuses, enumerations, geo data, etc., are located in XML Entity Engine files in ofbiz/commonapp/db/. These files can be located and loaded automatically by the install.jsp page in WebTools. This page looks in the directories specified in the entityengine.xml file for a given entity group name and finds all .xml and .sql files. These are listed and confirmation is requested by the page. Clicking on the Yes, Load Now link will cause these files to attempt to be loaded. Error messages will appear in the page as well as on the console or in log files. Data files can also be loaded one at a time by specifying the full path of the .sql or .xml file in the load a single file form. While on the topic, XML Entity Engine files can also be imported and exported through the import & export pages in WebTools.

As mentioned above an entity consists of a set of fields and a set of relationships to other entities. In the XML entity definitions each of these are specified in addition to attributes of the entity itself such as the entity name, the corresponding table name, the package name for the entity, and meta data about the entity such as the author, a copyright notice, a description, and so forth. Here is an example of an XML entity definition:

```
        <entity title="Sample Entity"
                copyright="Copyright (c) 2001 John Doe Enterprises"
                author="John Doe" version="1.0"
                package-name="org.ofbiz.commonapp.sample"
                entity-name="SampleEntity"
                table-name="SAMPLE_ENTITY">
          <field name="primaryKeyFieldOne" col-name="PRIMARY_KEY_FIELD_ONE"
type="id-ne"></field>
          <field name="primaryKeyFieldTwo" type="id-ne"></field>
```

```
    <field name="fieldOne" type="long-varchar"></field>
    <field name="fieldTwo" type="long-varchar"></field>
    <field name="foreignKeyOne" type="id"></field>
    <prim-key field="primaryKeyFieldOne" />
    <prim-key field="primaryKeyFieldTwo" />
    <relation type="one" rel-entity-name="OtherSampleEntity">
      <key-map field-name="foreignKeyOne" rel-field-name="primaryKeyOne" />
    </relation>
    <relation type="one" title="Self" rel-entity-name="SampleEntity">
      <key-map field-name="primaryKeyFieldOne" />
      <key-map field-name="primaryKeyFieldTwo" />
    </relation>
    <relation type="many" title="AllOne" rel-entity-name="SampleEntity">
      <key-map field-name="primaryKeyFieldOne" />
    </relation>
  </entity>
```

This is a pretty simple entity that demonstrates a few small points. The meta data at the top is all optional, and if left unspecified will default to meta data defined for the entire entity model file. For example the "description" element was left out, so the ModelReader will use the description specified at the top of the XML file if one exists. The package-name is used to organize the entities, and specify a default location for any code that would be entity specific. This becomes extremely useful when you have an entity model with hundreds of entities.

Notice that while the field primaryKeyFieldOne has a column name specified, none of the other fields do. The col-name and the table-name elements are optional. They can be derived from the field name or entity name through widely used conventions. These conventions dramatically simplify the definition of entities. Table and column names are written in caps with underscores separating the words like SAMPLE_ENTITY. Entity and field names are written using the Java conventions for class and field names where all letters are lowercase except for the first letter of each word, which is uppercase. Entity names correspond to Java classes to the first letter is upper case but field names correspond to member fields of a class so the first letter is lower case. For example: SampleEntity and fieldOne correspond to SAMPLE_ENTITY and FIELD_ONE.

Multiple primary key columns can be specified using multiple <prim-key> tags specifying the names of the primary key fields.

Field types are specified using a type string, which is defined in a fieldtypemodel XML file specified by the fieldtypemodel.dtd XML Data Type Definition. Each type maps to a Java type and an SQL type. Because of this

separation different fieldtypemodel XML files can be specified for different databases allowing an entitymodel XML file to work with the various databases. In addition, validators can be specified for a field type or for any field which denote that the named validator should be called when data is input. These validators are defined in the class org.ofbiz.core.util.UtilValidate and follow the definition pattern: [boolean isValid(String in);].

Multiple relations can exist for each entity. Each relation is either of type 'one' or of type 'many' depending on the cardinality of the relation. If the type is 'one' then the key-map elements must fully specifiy the primary key of the related entity. If the type is many, the key-map elements do not need to have any relation to the primary key of the related entity. If multiple relations to the same related entity are used for a given entity, a title must be specified to make the relation name unique. By this convention the relation name is defined as [title][rel-table-name]. For the two SampleEntity relations their names are SelfSampleEntity and AllOneSampleEntity. For OtherSampleEntity there is no title, so the relation name is simply OtherSampleEntity, or the name of the related entity.

Key Maps are used to define how the related entity will be looked up. Each key map specified a field name (field-name) and a related field name (rel-field-name). If the two column names are the same, the rel-field-name does not have to be specified.

View Entity Modeling

特别地，当一个实体只有一个关系的时候，我们可以定义一个虚拟的视图 virtual，view。这个视图和 otalce 及其他常用的关系数据库的视图一致，视图实体允许你联合、连接两个实体去建立一个新的实体。新视图实体中域的名称使用原来实体中域的别名。视图实体中所使用的实体，使用 view-links 根据实体的关键字将实体连接成 key_map，这和上面的关系定义类似。

视图数据和常规的实体一致，他们包含有名称、包、描述、版本、作者等。但不包含数据库表的名称，因为他们没有对应的实体源。视图实体和其他的实体一样也必须在实体组 XML 文件中进行定义。

```
<view-entity title="Sample View Entity"
        copyright="Copyright (c) 2001 John Doe Enterprises"
        author="John Doe" version="1.0"
        package-name="org.ofbiz.commonapp.sample"
```

```
        entity-name="SampleViewEntity">
    <member-entity entity-alias="SE" entity-name="SampleEntity" />
    <member-entity entity-alias="OSE" entity-name="OtherSampleEntity" />
    <alias entity-alias="SE" name="primaryKeyFieldOne" />
    <alias entity-alias="SE" name="primaryKeyFieldTwo" />
    <alias entity-alias="SE" name="fieldOne" />
    <alias entity-alias="SE" name="fieldTwo" />
    <alias entity-alias="OSE" name="primaryKeyOne" />
    <alias entity-alias="OSE" name="otherFieldOne" field="fieldOne" />
    <view-link entity-alias="SE" rel-entity-alias="OSE">
      <key-map field-name="foreignKeyOne" rel-field-name="primaryKeyOne" />
    </view-link>
    <relation type="one" rel-entity-name="OtherSampleEntity">
      <key-map field-name="primaryKeyOne" />
    </relation>
    <relation type="many" title="AllOne" rel-entity-name="SampleEntity">
      <key-map field-name="primaryKeyFieldOne" />
    </relation>
  </view-entity>
```

Normal entities that will become part of the view entity are referred to as member entities. Each member entity is given an alias and is referred to by that alias for the rest of the definition. This allows a single entity to be linked in multiple times.

Rather than specifying fields with a view entity you specify aliases. Each alias is effectively a field in usage and is defined by being mapped to a field on an aliases member entity. If the field name is that same as the alias name, there is no need to specify it. In other words if no field name is specified the alias name will be used as the field name to map to on the member entity with the specified entity-alias.

View links are used to specify the links between the member-entities of the view. They link one entity-alias to another and use key-maps just like relations. Here the field-name specifies the name of the field on the aliased entity and the rel-field-name the field on the related aliased entity. As with many other things, the rel-field-name is optional if it is the same as the field-name.

View Entities have primary keys just like normal entities. The Entity Engine automatically determines which aliases are primary keys by looking at the field that they alias.

The primary key for a view entity should include all primary key fields of each member entity of the view. This means that all of the primary key

fields must be aliased, but fields that are used to link the entities need only be aliased once. For example, if an OrderHeader entity has an orderId primary key and an OrderLine entity has an orderId primary key and an orderLineSeqId primary key, and the orderId was the mapped key that links the two entities, then the view entity would have two primary keys: orderId and orderLineSeqId. In this case orderId would only be aliased once since by definition the orderIds from each entity will always be the same since that is how the entities are linked (or joined).

Relationships are specified the same way with view entities as they are with normal entities. That key-map attributes are still called field-name and rel-field-name but in the case of view entities the field name is actually the alias name that will be looked up.

The Entity Engine API

The Entity Engine classes in the package **org.ofbiz.core.entity** define the API used to interact with Entity Engine entity data. From a users point of view only three classes really need to be understood. They are **GenericDelegator**, **GenericValue** and **GenericPK**. The **GenericDelegator** class, usually used with the instance name '**delegator**', is used to do create, find, store and other operations on a **GenericValue** object. Once a **GenericValue** object is created it will contain a reference to the delegator that created it and through this reference it knows how to store, remove and do other operations without requiring a program to invoke methods on the delegator itself.

I've been trying to think of how best to present information about this API, but short of writing a number of documents about the specific usage of each piece, there is not much that is useful that I could write here. I recommend reading through the JavaDocs for the core module, specifically the **org.ofbiz.core.entity** package, and browsing through the eCommerce and other applications which make heavy use of this API.

A few quick notes to help you get started might be in order. Rather than trying to construct a GenericValue or a GenericPK yourself, you should use the **makeValue** and **makePK** methods on the **GenericDelegator**. These create an object without persisting it and allow you to add to it and create or store it later using their own **create** method, or calling the **create** method on the **delegator** object.

Value instances can be retrieved from the database with the **findByPrimaryKey** methods, or a collection can be retrieved using the **findAll** or **findByAnd** methods.

There are two main types of findByAnd methods. Each type has a number of variations that may include the use of a cache, and may accept an orderBy field list. The two main types accept different field lists. One accepts a Map of fields and finds entities by anding together expressions where each named field must equal the corresponding value in the map. The other type of findByAnd accepts a list of **EntityExpr** objects which are used to specify small expressions that will be anded together. Each EntityExpr specifies a field name, an operation, and a value for the field.

To create (or insert) values into the database, use the **create** method on the **GenericValue** or **GenericDelegator** objects. To store (or update) existing values, use the **store** method on the **GenericValue** or **GenericDelegator** objects.

For storing multiple entities the **GenericDelegator** class has a method called **storeAll** which takes many **GenericValue** instances and stores them in the same transaction. Actually, to say that it stores them is incorrect. It checks to see if they exist and if so does an update, if not it does an insert. This may be optimized in the future for speed by allowing you to specify whether it should be inserted or updated based on prior knowledge of the existence if that entity. Note that this is a DIFFERENT behavior that the store method, which just does an update.

Removal of entities is done through the **remove** method on either the **delegator**, or the **GenericValue**.

The EntityCondition object

Originally the EntityExpr object was meant to be nestable to allow for more flexible queries, but was never completed. Also, even if you could next it the types of queries you could run would be limited because you couldn't have two ANDs (for instance) inside a set of parenthesis. To address these issues, and complete the EntityExpr implementation the EntityCondition abstract object has been introduced along with the EntityExprList and EntityFieldMap objects which both extend EntityCondition. The EntityExpr object has also been changed to extend EntityCondition.

The EntityExprList and EntityFieldMap objects are pretty simple, they are created with a List or Map, respectively, and an EntityOperator to specify operator used to join the members of these containers, generally AND or OR.

The EntityExpr class now has two primary constructors: one for comparing a field to a value (the String, EntityOperator, Object constructor), and one for comparing two EntityCondition objects (the EntityCondition, EntityOperator, EntityCondition constructor).

A findByCondition method is now available that accepts an EntityCondition argument (as well as some other useful arguments) and this EntityCondition can be an EntityExpr, EntityExprList or EntityFieldMap.

The code inside the Entity Engine has changed somewhat because these EntityCondition objects now created their own WHERE clauses. This is a nice architectural point because other custom EntityConditions could also be created and used as desired by a savvy developer.

The EntityListIterator object

The EntityListIterator class implements the ListIterator interface for convenience, but also has other methods that are necessary for it's operation, like a close() method for when you are finished.

This object allows you to iterate through query results efficiently in both directions by keeping a reference to the ResultSet that comes back from the query. This makes is possible to use the cursor feature in the database and especially for large queries uses memory much more efficiently. This object constructs GenericValue objects on the fly rather than creating a bunch all at once, so if you need to export the results of a huge query to a file or something, it can be done without a massive amount of memory.

JTA Support

The Entity Engine JTA Support is simple to use, but has a few complications in configuration. The support runs through an API and a Factory class so that no direct contact with the particular JTA implementation is necessary. The TransactionFactory class can be used to get the two main objects needed for JTA use: UserTransaction and TransactionManager. The current implementation supports the Tyrex JTA/JTS implementation. To use a different implementation simply change the TransactionFactory class, everything else uses that. That's the tricky configuration part, if you aren't using Tyrex. For Tyrex make sure a domain configuration XML file called tyrexdomain.xml must be on the running classpath.

To demarcate transactions you can use the TransactionUtil class which wraps the UserTransaction class and only throws GenericEntityExceptions, well, and runtime exceptions. The basic methods needed are begin(), commit(), and rollback(), but the rest are included and can be very useful. A transaction is attached to the current thread so it is not necessary to pass it around all over the place. After beginning a transaction make sure it is always either committed or rolled back. This is normally done by committing at the end of a try block and and rolling back in each catch block. You can also use the standard UserTransaction object by getting one from the TransactionFactory.

Core Web Tools

The WebTools web application contains a number of useful tools for working with the entity engine. These include a cross linked reference to the entity definitions, a tool for editing entity and relation definitions, and a JSP which acts as an XML template and also saves the XML entity definitions to their corresponding files.

There is also a JSP that acts as a front end to the routines which checks the current state of the database table definitions and reports any differences. Where possible tables or columns which are missing can be added to the database. This is the same routine which optionally runs when the server loads and optionally creates missing tables and columns.

The entity code generator has been removed from the project, or deprecated if you will, because of the next generation entity tool which is the entity engine described herein. There are still some occasions where the use of templates to create entity specific code or other text is still useful, and in fact, necessary. One example of this is the JSP which creates the XML for the entity definitions from those definitions, and is used for writing out the XML after entity definitions have changed. Other uses for templates include manually generating database specific table creation SQL and quick start JSPs and event handlers which allow for finding, viewing and editing entity specific data. These can be used as starting points for task specific applications.

Where entity data editing is not task specific, but instead is entity specific, the Entity Data Maintenance pages in WebTools can be used. They are dynamic pages that rely on the in memory entity definitions to create

forms for the entering of data, and event for handing the entered data (including validators specified in the entity definition), and finding specific entities by any of the fields on the entity, or finding any relation entity instances for a given entity instance. For instance, when viewing the OrderHeader entity all of the relations to that entity can be viewed as well, including links to edit and view them. These related entities would include OrderType (one relation), OrderLine (many relation), and many others.

Data in the database can be imported from and exported to Entity Engine XML files in the import and export pages. Importing data causes corresponding entity instances to either be created or updated, depending on whether or not they already exist. The Export page allows you to specify which entities you want to export the data for by using a big list of check boxes, one for each entity. For more granular control over exported data, the Entity Data Maintenance pages mentioned above would be the place to look (not yet finished though...).

1、介绍

二、编程指南

在 ofbiz 中对实体的访问（数据库）是通过 delegate 对象来进行的，而 delegate 对象是 GenericDelegator 类的一个实例，他包含有关实体操作的方法和属性。

1、delegator 对象的获取

- 在 JSP 中使用

<jsp:useBeanid="delegator"type="org.ofbiz.core.entity.GenericDelegator"scope="request"/>

- 在 severlet 或 event 中使用

GenericDelegatordelegator=(GenericDelegator)request.getAttribute("delegator");

- 通过一个已知的数值对象获取 delegator，方法为

GenericDelegatordelegator=userLogin.getDelegator();

- 手工建立

GenericDelegatordelegator=GenericDelegator.getGenericDelegator("default")

2、 数据访问

1)以 delegate 对象进行数据访问

- 插入使用 create 方法，在插入之前一般要用 makeValue 方法，建立一个数值对象，然后进行插入，典型的语句为

GenericValuepartyRole=delegator.makeValue("PartyRole",UtilMisc.toMap("partyId",partyId,"roleTypeId",context.get("roleTypeId")));

partyRole.create();

- 删除 remove 方法，remove 一般的用法为

partyRole=delegator.findByPrimaryKey("PartyRole",UtilMisc.toMap("partyId",partyId,"roleTypeId",context.get("roleTypeId")));

partyRole.remove();

- 使用 store 方法，包括(store 和 storeall)典型的语句为

delegator.storeAll(storeAll);

store 存储一个数值对象，而 storeall 存储用 List 组织起来的一组数值对象。

Storeall 的用法说明：

通过 findByPrimaryKey 在一个实体中查找出符合条件的一条数值对象

```
GenericValue orderHeader = delegator.findByPrimaryKey
                        ("OrderHeader", UtilMisc.toMap("orderId", orderId));
```

对数值对象中的某个值进行修改

```
orderHeader.set("statusId", statusId);
```

创建另外一个不同实体的数值对象（当然也可以采用相同的实体）。

```
changeFields.put("orderStatusId",
                delegator.getNextSeqId("OrderStatus").toString());
changeFields.put("statusId", statusId);
changeFields.put("orderId", orderId);
changeFields.put("orderItemSeqId", orderItem.getString("orderItemSeqId"));
changeFields.put("statusDatetime", UtilDateTime.nowTimestamp());
GenericValue orderStatus = delegator.makeValue("OrderStatus", changeFields);
```

使用 setPKFields ， setNonPKFields 建立一个数值对象

```
roleType = delegator.makeValue("RoleType", null);
 roleType.setPKFields(context);
 roleType.setNonPKFields(context);
 roleType = delegator.create(roleType);
```

将数值对象，放到 List 中

```
List toBeStored = new ArrayList();
toBeStored.add(orderHeader);
toBeStored.add(orderStatus);
```

将数值对象，存储到数据实体中

```
delegator.storeAll(toBeStored);
```

- 查找使用 find 方法，包括 findall 、 findAllByPrimaryKeys 、 findByAnd ，findByCondition、findByLike、findByOr、findByPrimaryKey、findListIteratorByCondition

- 依据某数值对象的关系，查找关联信息，可以使用 getRelated 方法。包括 getRelated、getRelatedByAnd、getRelatedDummyPK、getRelatedOne、getRelatedOrderBy

2）依据数值对象进行访问

在现有的数值对象上可以进行下列操作，

- 根据关系查找关联信息 getRelated，包括 getRelated、getRelatedByAnd、getRelatedDummyPK、getRelatedMulti、getRelatedOrderBy。
- 刷新本数值对象 refresh
- 保存本数值对象 store，主要用于修改后的保存
- 删除数值对象 remove,包括删除本数值对象 remove 和删除某个关联的数值对象 removeRelated

在现有数值对象上的操作是通过调用

三、编程参考

第四部分、工作流

一、工作流引擎指南

1、简介

OFBiz 工作流引擎基于 WfMC 和 OMG 规范（看相关文档可以了解这些规范的信息）。它是服务框架的成员之一，与 EntityEngine 紧密集成。工作流引擎把 entitymodel_workflow.XML 文件找到的实体用作定义信息，而把 entitymode_workeffort 文件找到的实体用作运行时刻存储。一个流程或任务（activity）都是实时的。因此，工作流引擎不是运行在一个线程上，而只是简单的一组 API 和通用对象在处理流程。当工作流发生改变时，引擎马上就处理这个变化，处理结束后，引擎返回结果。因此，当一个应用垮了（或系统重启），重启时工作流接着从停下的位置继续执行。

工作流引擎不是为一个 web 站点的处理流程而设计的。这是一个普遍的错误概念。web 站点的流转由控制 Servlet 处理。工作流是为了达到一个目标而进行的手动和自动任务（activitie）处理。

OFBiz 工作流引擎把 XPDL 作为自己的流程定义语言。这是一个开放的标准，十分灵活。在 XPDL 规范没有明确或留给厂商实现的地方，我们在 XPDL 扩展节说明。

2、XPDL 扩展

工作流使用 XPDL 进行设计。本文档不讨论 XPDL 细节，这些内容在 WfMC 有详细解释，请看相关文档。

在 WfMC 规范里，留了很多属性（attribute）给厂商使用。下面是我们在使用的 XPDL 扩展：

任务（ACTIVITY）扩充属性

- **acceptAllAssignments**-这个扩展属性告诉工作流引擎在一个任务开始之前的所有委派都必须接受。当一个任务有多个参与者，并且要求在任务开始前每个参与者必须接受自己收到的委派时，本属性将会很有用。（缺省值：NO）

- **completeAllAssignments**-类似于 **acceptAllAssignments** 属性，告诉工作流引擎在一个任务完成之前的必须完成所有的委派。例如，在某个薪资任务中，要求所有的员工必须提交自己的时间表后才能进行下一步动作。（缺省值：NO）
- **limitService**-当定义一个任务时，可以设定一个时间限制，表示分配给该任务时间总和。一个服务可以被设定成当某个任务没有在规定的时间完成时启动起来。（缺省值：不设定）
- **limitAfterStart**-这个属性告诉工作流引擎，限制检查在任务开始之后发生。当把属性设定为 NO 时，限制从委派创建开始检查，而不是从任务开始。（缺省值：YES）
- **canStart**-告诉工作流引擎，如果本扩展属性设定为 yes 的任务都能够初始化工作流。因此，工作流不但可以从第一个任务开始启动，而且可以从任何本属性设定为 yes 的任务开始启动。

实现/工具扩充属性和实际参数

- **runAsUser**-这个扩展属性将告诉 TOOL（如果是过程）以这个用户来运行定义的服务。取用户的 userLoginId 作为属性值。缺省情况下不传递用户登录对象。
- **ActualParameters:**
  - 表达式：可以使用 **ActualParameters** 里的表达式对上下文属性与服务参数进行映射。你必须用 *expr:*来注释一个表达式。例如：
  - <ActualParameter>expr:orderNumber=orderId</ActualParameter> 先将上下文属性 *orderId* 映射到一个叫 *orderNumber* 的内存缓冲变量，然后，如果你接着写：
  - <ActualParameter>orderNumber</ActualParameter>表示参数 *orderNumber* 将被传递到值为 *orderId* 的服务中。
  - 工作任务 ID：你可以把 *workEffortId* 作为一个实际参数，这个参数映射为当前任务的主关键字。

缺省启动任务（ACTIVITY）

缺省地，一个工作流从任务列表的第一个任务开始启动，因为列表中的第一个任务为缺省启动任务。这意味着当一个工作流正常启动时，将首先运行第一个任务，然后在根据路由（transition）继续流转。最常见的方式是调用一个工作流时，可能从许多不同的起点开始调用。在这种情况下，采用客户端 API 来启动合适的任务。路由（transition）将从这个起点继续下去。只有任务的'canStart'扩展属性设定为 yes 的任务才可以初始化一个工作流。

3、客户端 API

OFBiz 工作流引擎的客户端 API 由一组全局服务和 factory 类组成。这个 factory 类叫 org.ofbiz.core.workflow.WfFactory，能用来创建一个工作流构件（process/activity/assignment）的新实例，或者定位一个已存在的实例。实例被储存在工作任务实体中，并且在必要时装入内存。在 org.ofbiz.core.workflow.WorkflowServices 里的这组全局服务被设置能够很容易生成、接受和完成委派/任务。虽然使用这些服务不是必须的，但是强烈建议你使用。复习 JavaDocs 可以让你了解到这些类。

工作任务（*workeffort*）的 web 应用就是一个工作流 web 客户端的例子。通过这个界面，你可以接受到当前委派给你的组别/角色的工作委派，并可以编辑其状态。如果你想尝试一下，请单击 http://localhost:8080/workeffort.。

4、工作流任务

手动和自动任务的组合让一个工作流变得功能十分强大。OFBiz 工作流引擎当前已经实现无（NO）、路由（ROUTE）、工具：过程（TOOL:Procedure）、子流程等类型任务。工作：应用（TOOL:Application）类型任务目前仍未实现，但是将来版本会实现。

- **无（NO）**-如字面意义显示一样，表示没有任务。这用来描述一个'手动'任务。
- **路由（ROUTE）**-一个路由活动是利用路由（transition）简单地转移到其他任务中。
- **工具（TOOL）**-
  - 应用（Application）：调用一个外部应用程序-目前暂未实现。
  - 过程（Procedure）：OFBiz 工作流引擎把过程当作一个内部服务调用来实现。
  - **子流程（SUB-FLOW）**-同步或异步地创建和运行一个子流程。
  - 当前 OFBiz 工作流引擎版本并没有实现 LOOPS，但将来版本会实现。

5、用法

在使用工作流引擎之前，你必须先准备一个预计调用的流程，流程要按 XPDL 格式进行设计。一旦流程设计完成，需要把 XPDL 文件导入到工作流实体中。你可以通过 web 界面的 *webtools* 工具来完成这些工作。工作流被导入后，就可以被执行了。调用一个工作流最容易的方法是把它定义成一个服务。为工作流创建一个服务，并把引擎类型设定为'workflow'，然后就可以象运行任何其他服务一样启动一个工作流。

6、文档注释

我们知道本文档不够详细，为此我们向你道歉。然而，我们大部分时间都在写代码，因此，不能全力写文档。虽然我们也认识到好文档是有必要的，但是我们的资源终究有限。如果你或了解的其他任何人愿意花时间来为 OFBiz 写文档，请根据本文档顶部的 e-mail 地址联系我们。多谢。

二、编程指南

1、 建立流程状态实体

开发一个流程首先要建立一个，记录流程状态的实体，比如在 ofbiz 的流程例子中建立了一个 OrderHeader 的实体，这个实体最起码要有三个基本属性描述，一个是 partyId，另外一个是 roletypeId，还有一个是被控对象的 Id。

PartyId 和 RoletypeId 指明流程中某个流程的执行者，被控对象 Id 指明要控制的对象。比如在公文流转中控制的是公文。在 ofbiz 的定单管理中控制的是定单。

2、 编写流程活动的服务

流程在执行过程中，依据 PartyId、RoletypeId 和被控对象的 Id，并根据环境参数，执行特定的动作，这个动作可以由开发人员具体编写。并将执行的状态保存到流程状态实体中。

3、 对于 MANUAL 类型的服务，开发人员编写特定的应用，在应用执行完成后重新执行流

程。

三、编程参考

本部分结合 ofbiz 中定单管理中的例子来阐述

1、在定单管理中，若一个定单结束则定单进入流程控制，

1）、流程的触发

定单流程是在在清空购物车后由 clearcart 请求触发的，其定义在 Controll.xml 描述，结构如下：

```
-------------------------------------------------------------------------------------
<request-map uri="clearcart">
      <security https="true" auth="true"/>
      <event type="java" path="org.ofbiz.commonapp.order.shoppingcart.ShoppingCartEvents"
                  invoke="destroyCart"/>
       <response name="success" type="request" value="initiateOrderWorkflow"/>
       <response name="error" type="view" value="confirm"/>
</request-map>
<request-map uri="initiateOrderWorkflow">
       <security https="true" auth="true" direct-request="false"/>
       <event type="java" path="org.ofbiz.commonapp.order.shoppingcart.CheckOutEvents"
                  invoke="initiateOrderWorkflow"/>
       <response name="success" type="view" value="confirm"/>
       <response name="error" type="view" value="confirm"/>
  </request-map>
     ------------------------------------------------------------------------
```

清空购物车后发出 initiateOrderWorkflow 请求,而 initiateOrderWorkflow 请求触发 CheckOutEvents 类中 initiateOrderWorkflow 方法的执行 initiateOrderWorkflow 启动流程.

2)、流程的启动

流程启动是由 initiateOrderWorkflow 类中 initiateOrderWorkflow 的方法来启动的，具体过程如下

```
-------------------------------------------------------------------
public static String initiateOrderWorkflow(HttpServletRequest request,HttpServletResponse response){
     GenericDelegator    delegator=(GenericDelegator)request.getAttribute("delegator");
     LocalDispatcher    dispatcher=(LocalDispatcher)request.getAttribute("dispatcher");
     StringorderId=(String)request.getAttribute("order_id");
     GenericValue    orderHeader=null;
     try{
```

```
            orderHeader=delegator.findByPrimaryKey("OrderHeader",UtilMisc.toMap("orderId",orderId));
        }catch(GenericEntityExceptione){
            Debug.logError(e,"Problemsgettingorderheader",module);
            request.setAttribute(SiteDefs.ERROR_MESSAGE,"<li>Problemsgettingorderheader.WFnotstarted!");
            return"error";
          }
        if(orderHeader!=null){
              try{
                  dispatcher.runAsync("processOrder",UtilMisc.toMap("orderId",orderId,"orderStatusId",orderHe
ader.getString("statusId")));
              }catch(GenericServiceExceptione){
                  Debug.logError(e,"Cannotinvokeprocessingworkflow",module);
                  request.setAttribute(SiteDefs.ERROR_MESSAGE,"<li>ProblemsstartingorderWF!");
                  return"error";
              }
        }
        return"success";
    }
```

该方法的主要工作为

- 准备服务的参数

服务的参数有两个，一个是 orderId，另外一个是 orderStatusId，这两个参数是从 orderHeader 实体中获取的，orderHeader 实体记录了定单状态和参数

OrderHeader 在 entitymodel_order.xml 中定义：其结构如下

```
---------------------------------------------------
<entityentity-name="OrderHeader"
package-name="org.ofbiz.commonapp.order.order"
never-cache="true"
title="OrderHeaderEntity">
<fieldname="orderId"type="id-ne"></field>
<fieldname="orderTypeId"type="id"></field>
<fieldname="orderDate"type="date-time"></field>
<fieldname="entryDate"type="date-time"></field>
<fieldname="visitId"type="id"></field>
<fieldname="statusId"type="id"></field>
<fieldname="createdBy"type="id-vlong"></field>
<fieldname="syncStatusId"type="id"></field>
<fieldname="billingAccountId"type="id"></field>
<fieldname="originFacilityId"type="id"></field>
<fieldname="webSiteId"type="id"></field>
<fieldname="grandTotal"type="currency-amount"></field>
<prim-keyfield="orderId"/>
-----------------------------------------------------
```

- 调用流程服务

initiateOrderWorkflow 调用的服务为 processOrder，processOrder 在 services_order.xml 中描述，其内容为：

<servicename="processOrder"engine="workflow"location="org.ofbiz.commonapp.order.order"invoke="ProcessOrder">

<description>Servicefortestingtheworkflowengine</description>

<attributename="orderId"type="String"mode="IN"optional="false"/>

<attributename="orderStatusId"type="String"mode="INOUT"optional="false"/>

</service>

---

engine="workflow"说明这个服务是流程服务。流程服务是由 wfmc 的标准语言 xpdl 来描述的程序脚本。

location="org.ofbiz.commonapp.order.order"指的流程脚本中的包(package)

invoke="ProcessOrder"指的是流程脚本中的 WorkflowProcess，

3）、活动的执行

a）参数传入

上面已经说明，ofbiz 在调用服务时要给流程传入两个参数，这两个参数在流程脚本中的要有描述：

```
<FormalParameters>
        <FormalParameterId="orderId"Index="1"Mode="IN">
            <DataType>
               <BasicTypeType="STRING"/>
            </DataType>
            <Description>Theordernumber</Description>
        </FormalParameter>
        <FormalParameterId="orderStatusId"Index="2"Mode="INOUT">
            <DataType>
                <BasicTypeType="STRING"/>
            </DataType>
            <Description>Theorderstatus</Description>
      </FormalParameter>
</FormalParameters>
```

---

b）活动的执行

在流程脚本中，活动 tool 说明要调用的过程

- 如果 Type="PROCEDURE"，则服务名称缺省为 ToolId，若存在类似于 ExtendedAttributeName 扩展属性中存在 serviceName 的定义，则服务名称为 ExtendedAttributeName 中定义的名称
- 如果 Type="APPICATION"则服务名称为 wfActivateApplication
- 其他所有情况都使用 ToolId 作为服务的名称

<ToolId="orderNotReadySuspend" Type="PROCEDURE">

<ActualParameters>

<ActualParameter>workEffortId</ActualParameter>

</ActualParameters>

<ExtendedAttributes>

<ExtendedAttribute Name="serviceName"Value="wfSuspendActivity"/>

<ExtendedAttribute Name="runAsUser"Value="admin"/>

</ExtendedAttributes>

</Tool>

- 其中<ExtendedAttributeName="serviceName"Value="wfSuspendActivity"/>

  说明该活动调用 ofbiz 定义的一个标准服务 wfSuspendActivity，这个服务在 services_workflow.xml 中定义结构如下

  <servicename="wfSuspendActivity"engine="java"

  location="org.ofbiz.core.workflow.client.WorkflowServices"invoke="suspendActivity">

  <description>Suspendaworkflowactivity</description>

  <attributename="workEffortId"type="String"mode="IN"/>

  </service>
- 使用的 ActualParameter 参数为 workEffortId，<ActualParameter>workEffortId</ActualParameter>,workEffortId 的值是内部自动获取的。
- <ExtendedAttributeName="runAsUser"Value="admin"/>

  说明该活动是由 admin 来建立,如果没有指明则说明该活动的建立是由 workflowOwnerId 来指定，workflowOwnerId 指的是登录用户

  这里说明（活动的创建者和活动的执行者不是一回事，也即一个是 workflow 的 owner，一个是 workflow 的 performer

c) Type="APPICATION"时的服务

当 Type="APPICATION"时调用 wfActivateApplication 服务，该服务是向 ApplicationSandbox 实体中记录当前活动的状态信息，其中最重要的是 workEffortId，通过它可以获取当前的流程、当前的处理、当前的活动等。以便由应用程序进行处理。一般情况下他是流程引擎处理的结果。

GetApplicationContext 获取信息

CompleteApplication 完成应用

d)活动中 ActualParameters 参数的获取方法和步骤

活动中可以使用的参数，包括

- 服务运行传入的参数
- 运行环境参数，是从根据 workEffortId 从 WorkEffort 中获取 runtimeDataId，由 runtimeDataId 从 RuntimeData 中读取
- extendedAttributes 定义的参数（runAsUser，workflowOwnerId 等）
- userLogin，
- workEffortId

- assignedPartyId（由 Performer 定义）
- assignedRoleTypeId，（由 Performer 定义）

参数值的获取

活动依据 ActualParameters 的定义从上述可运行的参数中取得或计算出参数值

- 如果只列出变量名称，则直接从上述环境中取值
- 如果变量以：expr：打头则变量值用 bsh 计算出来
- 如果变量以：name：打头则变量值从环境中取值并影射到新的名称上
  比如：流程中的服务，需要的变量名称为 statusId,而环境中实际的名称为 currentStatusId,则 ActualParameters 可以描述成 name:statusId=currentStatusId

综合起来 ActualParameters 可能是这样的格式

<ActualParameter>orderId</ActualParameter>
<ActualParameter>name:statusId=currentStatusId</ActualParameter>
<ActualParameter>expr:orderId</ActualParameter>

e)在给一个例子和说明

<ToolId="approveOrderChangeOrderStatus"Type="PROCEDURE">
<ActualParameters>
<ActualParameter>orderId</ActualParameter>
<ActualParameter>statusId</ActualParameter>
</ActualParameters>
<ExtendedAttributes>
<ExtendedAttributeName="statusId"Value="ORDER_APPROVED"/>
<ExtendedAttributeName="serviceName"Value="changeOrderStatus"/>
<ExtendedAttributeName="runAsUser"Value="admin"/>
</ExtendedAttributes>

orderId,statusId 从运行环境中获取参数值，runAsUser 将运行环境改为 admin 的运行环境 userLogin，并将 userLogin 作为隐含参数输入给流程的活动。

2、流程的服务

1)、流程服务

<ExtendedAttributeName="serviceName"Value="changeOrderStatus"/>

说明该活动调用的是 changeOrderStatus，该服务在 services_order.xml 进程了说明，其描述如下

<servicename="changeOrderStatus"engine="java"location="org.ofbiz.commonapp.order.order.OrderServices" invoke="setOrderStatus">
<description>Changethestatusofanexistingorder</description>
<attributename="orderId"type="String"mode="IN"/>
<attributename="statusId"type="String"mode="IN"/>
<attributename="orderStatusId"type="String"mode="OUT"optional="true"/>

- 在流程中只指明了 2 个参数，另外一个参数采用缺省值"true"，这些参数要在流程描述

中用 ActualParameters 进行，其中 ActualParameters

- 参数中 statusId 是由<ExtendedAttributeName="statusId"Value="ORDER_APPROVED"/>获取的，值为 ORDER_APPROVED。
- setOrderStatus 服务

  setOrderStatus 做两个工作一是修改实体 OrderHeader 的状态（statusId）二是在 OrderStatus 实体中增加一条记录了 orderStatusId，statusId，orderId，statusDatetime 状态的记录。

2）、改变流程的用户环境

一般情况下 userLogin, 指的是登录用户的用户环境，若使用<ExtendedAttributeName="runAsUser"Value="admin"/>则流程引擎将流程的用户环境设置成 admin

3、扩充属性

任务（ACTIVITY）的扩充属性有 acceptAllAssignments，completeAllAssignments，limitService，limitAfterStart，canStart 共五个这些属性在“4.1.2 中已经介绍”，这里主要介绍 limitService 的执行。

LimitService 触发服务的执行，这个服务在流程脚本中没有明显地列出参数，而是隐含地使用在服务定义中所指明的参数，这些参数值的获取，与活动 tool 中服务获取参数值的方法一致。比如

在 ofbiz 的流程例子中 orderNotReady 活动的扩充属性为：

```
<ExtendedAttributes>
    <ExtendedAttribute Name="canStart" Value="N"/>
    <ExtendedAttribute Name="runAsUser" Value="admin"/>
    <ExtendedAttribute Name="limitAfterStart" Value="N"/>
    <ExtendedAttribute Name="limitService" Value="sendProcessNotification"/>
    <ExtendedAttribute Name="inheritPriority" Value="Y"/>
</ExtendedAttributes>
```

其中 sendProcessNotification 为服务，没有指明它使用的参数。该服务在 services_order.xml 中的描述为

```
<service name="sendProcessNotification" engine="java"
location="org.ofbiz.commonapp.order.order.OrderServices" invoke="sendNotification">
    <description>Limit Service for order processing workflow; sends activitiy notifications</description>
    <attribute name="workEffortId" type="String" mode="IN"/>
    <attribute name="adminEmailList" type="String" mode="IN"/>
    <attribute name="assignedPartyId" type="String" mode="IN" optional="true"/>
    <attribute name="assignedRoleTypeId" type="String" mode="IN" optional="true"/>
</service>
```

他有 4 个参数，因此该服务使用四个参数，而这四个参数，是从与 tool 中服务相同的运行环境中获得的。

AdminEmailList 是流程定义的常量

AssignedPartyId 和 assignedRoleTypeId 在 performer 定义
WorkEffortId 是运行环境中取得

4、runtimeData 怎样生成和调用

runtimeData 是通过 WfActivityImpl 中的 processContext 形成环境，并用 setProcessContext 将环境存到 runtimeData 实体中,这个环境是一个 Map，存入的时候自动转换成 XML

5、workflow 是怎样执行的

使用 xpdl 描述的脚本相当于 Ofbiz 的一个服务，这个服务的执行和通常的服务类似，但它是怎样启动和调用有关服务的方法，这里我们给出大致的分析

1、 服务调用 WorkflowEngine 中的 runAsync 方法
该方法建立流程的运行环境和参数然后调用 WorkflowRunner 启动流程，WorkflowRunner 包含三个参数 process, requester, startActivityId。

6、程中控制操作

1）、控制方法

ofbiz 流程是异步执行的，流程执行的每一步都在 workeffort 中记录没执行的状态和结果，对流程的控制是通过 WorkflowServices 中的方法来进行的，而 WorkflowServices 是通过 WorkflowClient 中的方法来实现的。

WorkflowServices 中封装了有关服务的多种操作，包括：
acceptAssignment，接受付值并启动响应流程的活动
acceptRoleAssignment 接受角色付值并启动响应流程的活动
appendActivityContext 向流程中追加信息
assignActivity 将活动连接到指定的 PartyId
cancelWorkflow，取消流程服务
changeActivityState，修改活动的状态
checkActivityState 检查活动的状态
completeAssignment 完成付值
delegateAcceptAssignment，delegate 并接受付值
delegateAssignment ，delegate 付值
getActivityContext 获取活动的环境
getOwner 获取流程的属主
hasPermission 检查用户是否有权访问流程的数据
resumeActivity 继续执行活动
suspendActivity 暂停活动
这些操作可以对流程进行控制
当活动执行到一个活动时，这个活动可以自动执行（AUTOMATIC）也可以手动执行（MANUAL），自动执行由流程引擎自动将执行状态设置为 ACCEPTED 并执行活动，

若为手动执行则有用户通过调用 WorkflowServices 的方法来手动启动流程服务。一般情况下在手动的执行之前都要做特定的工作，比如公文管理中修改文档，修改完成后再继续流程操作。特定工作要使用该活动有关参数，该参数从 workeffort 中取得。

2）、在 ofbiz 定单流程中,tasklist 的控制方式

在 tasklist 中发出 ceptAssignment 请求，该请求在 controller.xml 中的定义为：

```
-------------------------------------------------------------
    <request-map uri="acceptAssignment">
      <security https="true" auth="true"/>
        <event type="simple"
               path="org/ofbiz/commonapp/workeffort/workeffort/WorkflowSimpleEvents.xml"
               invoke="acceptAssignment"/>
        <response name="success" type="view" value="mytasks"/>
        <response name="error" type="view" value="mytasks"/>
    </request-map>
-----------------------------------------
```

说明调用 Mini_language 编写的方法 WorkflowSimpleEvents.xml，该方法为

```
-----------------------------------------
    <simple-method method-name="acceptAssignment" short-description="Create Work Effort">
        <call-map-processor xml-resource=
                "org/ofbiz/commonapp/workeffort/workeffort/WorkflowMapProcessors.xml"
                processor-name="assignmentMap" in-map-name="parameters"
                out-map-name="context"/>
        <check-errors/>
        <call-service service-name="wfAcceptAssignment" in-map-name="context">
            <default-message>Work Effort successfully created.
            </default-message>
          </call-service>
    </simple-method>
-----------------------------------------
```

说明该 mini_language 写的方法调用服务 wfAcceptAssignment，它的定义在 services_workflow 中的描述为：

```
-------------------------
 <service name="wfAcceptAssignment" engine="java"
                location="org.ofbiz.core.workflow.client.WorkflowServices"
                invoke="acceptAssignment">
        <description>Accept a workflow user assignment</description>
        <implements service="wfAssignmentInterface"/>
    </service>
-------------------------------------------------
```

说明 wfAcceptAssignment 调用 WorkflowServices 类中的方法 acceptAssignment，

7、流程服务的编写方法

流程的服务和其他的服务编写方法完全一样，流程服务的参数中除了包含，delegate,security, userlogin、等传统的参数外还包含从流程带入的相关参数。传统的参数是在执行服务时生成的。

流程服务一般用于活动的 Tool 中或 LimitService 中。

8、动态执行者

流程的执行者（Performer）有三种，RESOURCE，ROLE，HUMAN，其中 RESOURCE，ROLE 的执行者是动态获取的。RESOURCE，是有应用开发人员自己定义的。比如 Ofbiz 的流程定义中有：

```
<Participant Id="user" Name="Dynamic User">
    <ParticipantType Type="RESOURCE"/>
    <Description>Dynamic user assignment</Description>
    <ExtendedAttributes>
        <ExtendedAttribute Name="partyId" Value="expr:partyId"/>
    </ExtendedAttributes>
</Participant>
```

执行者 user 是在流程执行的过程中自动获取的，这个执行者就是环境 partyId 所指的会员。

四、公文流转流程的做法

1、 设计一个 workflowHead，这个 workflowHead 中包含四个基本信息
wfId 流程号　wfType 流程类型，partyId,roleTypeId.其中 wfType 指明流程的类型，比如文档流程，车辆审批流程，请假流程

2、 编写流程中的应用：比如公文管理中的流程应用有查看文档，修改文档。

3、 编写服务，比如在公文管理中，若一个公文流转到某人处，他没有在规定的时间里完成则可以通过编写服务发出电子邮件、及时通知，或将延迟信息记录在一个实体中以便以后考核使用等。也可以将文档进行加密等

4、公文流转中的状态：接受，完成，暂停，取消，退回等这和 ofbiz 的状态是一致的。

第五部分、特殊服务

一、JMS 消息服务引擎

1、JMS 消息服务引擎

JMS 消息服务引擎是 OFBiz 服务框架中的一种。之所以把它单独成文，是因为时下 JMS 是 J2EE 的重要规范之一。

Java2 平台，企业版（J2EE）满足了用一种健壮的、安全的、事务性的方法在 Web 上提供现有的应用程序和业务流程的需要。J2EE 环境下已经建立了几个规范，最值得注意的是 Java 消息传递服务（JMS）和 Java2 连接器体系结构（JCA），这些规范主要用来把 J2EE 应用程序与非 J2EE 环境集成在一起。两个接口使我们能够使用运行在 J2EE 应用程序服务器内的解决方案来集成非 J2EE 环境。一种方案解决了松散耦合的、基于 JMS 的异步集成，而另一种方案描述了耦合更紧密且同步的模型，这种方案使用的是 JCA。（我在这里只介绍 OFBiz 中配置使用 JMS。刚接触 JMS 的朋友，请先熟悉一下 JMS 本身的知识。）

OFBiz 消息引擎是在 jms 标准 api 的基础之上，将 JMS 整合到自己的服务框架之中，使 JMS 应用部署到不同的消息服务器是更灵活，更方便。

2、消息服务引擎配置和使用

这里我们一步一步来看看在 OFBiz 中是如何使用 JMS 的。

1.jndiservers.xml

路径:/commonapp/etc/这个文件中需要配置 jndi-server.指定 jms 客户端使用的 api.

例如：<jndi-servername="OpenJMS"

context-provider-url="rmi://111.11.11.1:1099/JndiServer"

initial-context-factory="org.exolab.jms.jndi.rmi.RmiJndiInitialContextFactory"/>

又如:<jndi-servername="WeblogicJMS"

context-provider-url="t3://111.11.11.1:7001/JndiServer"

initial-context-factory="weblogic.jndi.WLInitialContextFactory"/>

2.serviceengine.xml 路径:/commonapp/etc/

<jms-servicename="serviceMessager"send-mode="all">

<serverjndi-server-name="WeblogicJMS"jndi-name="JmsQueueConnectionFactory"

topic-queue="queue1"type="queue"username="system"password="12345678"

listen="true"/>

<serverjndi-server-name="OpenJMS"jndi-name="JmsQueueConnectionFactory"

topic-queue="topic1"type="topic"username=""password=""

listen="false"/>

</jms-service>

jms-servicetag 说明:

属性	必须?	描述
name	Y	JMS 服务的名称,在服务定义文件中作为 location 属性.参考 **3**
send-mode	N	发送模式。指：对应 server 发送采取的方式。 none:不发送。all:全部发送。如示例表示一次消息向 WeblogicJMS,OpenJMS 两个消息服务器发送。 以下的模式，还没有实现:first-available, random, round-robin, orleast-load.

servertag 说明:

属性	必须?	描述
jndi-server-name	Y	在文件 jndiservers.xml 中配置的 jndiserver.
jndi-name	Y	JMSFactory 的 jndi 名称.
topic-queue	Y	主题 topic 或者通道 queue 名称.
type	Y	topic 或者 queue
username	Y	建立 jmsconnection 需要提供的用户名.
password	Y	建立 jmsconnection 需要提供的密码.
listen	Y	是否启动监听 Job, 即是否接受该 topic, 或者 queue 的消息。(true/false)

3.services_xxxx.xml 服务定义文件

<servicename="testJMSQueue"engine="jms"
location="serviceMessenger"invoke="testScv">
<description>TestJMSQueueservice</description>
<attributename="message"type="String"mode="IN"/>
</service>

servicetag 说明:

属性	必须?	描述
name	Y	服务名称.
engine	Y	使用的服务引擎
location	Y	指的是 **2** 中提到的 name
invoke	Y	接受方的服务名称。注意：示例"testScv"是服务的名称，当 jmslistener 收到消息后调用此服务。

以上是在 OFBiz 中使用 JMS 引擎的方法。

二、时间服务（job）

1、JobScheduler

Theoverhauledjobschedulerisnowintegratedwiththeservicesframework.Thisisthemostappropriateplaceforascheduler.SinceitcannotbeguaranteethataHttpServletRequestandHttpServletResponseobjectwillbeavailablewhenajobisreadytorunitdoesnotmakesensetointegratewiththewebcontroller.Plus,thisfeatureismostusefulifitwerenotlimitedtowebenvironments.

Theschedulerisamultithreadedcomponentwithasinglethreadusedforjobmanaging/schedulingandseparatethreadsusedforinvocationofeachservice.Whenajobisscheduledtorun,theschedulerwillcalltheservicedispatcherassociatedwiththejobtoinvoketheserviceinitsownthread.Thiswillpreventlongortimeconsumingjobsfromslowingdownotherjobsinthequeue.

TheschedulernowsupportstheiCalendarrulestructureforrecurrence.Thejobsarenolongerstoredinan XMLfileandispartofaServiceDispatcher.ThereisoneJobSchedulerforeachServiceDispatcher(whichmeansthereisonlyoneperGenericDelegatoraswell).

如何工作的:

JobScheduler 最好的用法例子是异步服务调用。当一个异步服务被调用时，他被传递给 JobScheduler 排队运行。

Thebestusageexampleoftheschedulerisanasynchronousservicecall.Whenanasynchronousserviceisinvoked,itispassedtotheJobSchedulertobequeuedtorun.Arecurrenceentryiscreated(RecurrenceInfoandRecurrenceRuleentitiesarecreated),thejobisstored,(JobSandboxentityiscreated)andthecontext(Map)isserializedandstored(RuntimeDataentityiscreated).Theschedulerthenaddsthejobtothetopofthelistofscheduledjobs(asynchronousservicesdonothaveanydelaytime)andinvoked.

JobsarenolongerdefinedinanXMLfile.ThishasbeenmovedtotheJobSandboxentity.Thereisawebbasedclientinplanningforaddingpredefinedjobstothequeue,butcurrentlytheentitieswillhavetobecreatedbyhand.

三、利用 OFBiz 实现 SingleSignOn 单点登录

1、SSO 简介

SingleSignOn(SSO)通常被翻译为单点登陆，最常见被使用在企业内部网络应用，当用户访问企业内部应用系统时候只需作一次身份认证，随后就可以对所有应用系统进行访问，而不需要多次输入认证信息。SSO 可以提高普通用户工作效率，避免了各个系统都有自己独立的用户帐户和密码，减少了系统维护人员工作。本文只介绍一种利用 OFBiz 现有的架构建立一个简单而有效的 SSO 机制，对于更完善，复杂 SSO 实现，请参考相关文档。

2、利用 OFBiz 建立简单有效的 SSO 机制

首先看一下 OFBiz 的登陆机制，打开任意 webapp 的 controller.xml，会找到这一段：

```
<request-mapuri="checkLogin"edit="false">
<description>Verifyauserisloggedin.</description>
<securityhttps="false"auth="false"/>
<eventtype="java"path="org.ofbiz.securityext.login.LoginEvents"invoke="checkLogin
"/>
<responsename="success"type="view"value="main"/>
<responsename="error"type="view"value="login"/>
</request-map>
```

对于 controller.xml 里面定义任何的 request-map,如果其 auth="true"，那么 OFBiz 都会调用这个 checkLogin 来检测发出这个 request 的 user 是否已经在这个 webapp 登陆过，如果没有登陆过则把当前 request 名称和 post 参数 save 在 session 里面，然后分发到 login 的 view，让 user 来做登陆。在 user 登陆完以后，会读取 session 里前面 save 内容，重定向到前面的请求。

接下来谈一下我们实现的 SSO，首先必须要有统一登陆入口和统一存储 Useraccount 和 password 的数据库。假设我们现在统一登陆入口为一个跑在一台叫 SSOServer 服务器上的 ssoweb 应用。我们在这个 SSODB 里面利用 OFBiz 自带的 UserLogin 再扩展一个 field:ssoId，用来作为登陆以后的 user 一个独一无二标识。

修改 checkLogin 对于 user 没有登陆过这个 webapp 时候的处理，让他重定向到 sso 登陆入口：

```
<responsename="error"type="url"value="http://SSOServer/sso/control/main?redire
ct=http://thisserver/thisapp"/>
```

这样就可以达到统一到 SSO 入口认证的目的了。

现在我们来看一下一个实例：

- Step1.一个 user 打开浏览器，发出对 AServer 的 abc 应用某个 request。

abc 判断到这个 user 对于本身来说是一个新 session，是没有登陆过的，重定向到 sso 的登陆入口。

- Step2.user 在 sso 登陆入口输入用户名，密码。

ssowebapp 进行用户名，密码验证，生成一个独一无二的 ssoId，save 对应的 UserLogin，利用前面获得 redirect 参数构造一个这样的链接:“**http://AServer/abc/control/autoLogin?ssoId=generatedSsoId**”，进行重定向。

- Step3.自动登陆 AServer 的 abc 应用

autoLogin 是我们实现的一个 request-map，其作用是利用 ssoId 去统一存储 UserLoginSSODB 获得对应的 UserLogin，做一个自动登陆。登陆完以后，还是和原来的 OFBiz 一样的处理机制，会读取 session 里前面 save 的内容，重定向到第一步发出的请求。这个时候看起来还和原来的登陆方式没有什么区别，但是如果这个 user 继续操作：

- Step4.发出对 BServer 的 efg 应用某个 request。

efg 判断到这个 user 对于本身来说是一个新的 session，是没有登陆过的，重定向到 sso 的登陆入口。

- Step5.ssowebapp 判断

ssowebapp 会得知这个 user 前面已经在 SSO 登陆过，所有不用再次要求 user 登陆，利用 ssoId,redirect 参数构造一个这样的链接:“**http://BServer/efg/control/autoLogin?ssoId=generatedSsoId**”，进行重定向。

- Step6.自动登陆 BServer 的 efg 应用

OK，4~6 就是 SSO，对于 user 来说，完全是无缝的操作。

总结一下在 OFBiz 上做一个 SSO 实现的工作和要注意的地方：

1. 写一个 SSO 的 webapp，跑在一个统一的 SSOServer，使用统一的 SSO 数据库。
2. 任何需要集成 SSO 的应用，修改 webappscheckLogin 方法，新增加一个 autoLogin 的方法。
3. 在实际实施的时候，我们写的 SSO 登陆的 webapp 会把用户登陆信息写在一个 cookie 中，这样用户就算是打开新的浏览器访问的话，ssowebapp 可以从 cookie 中得到用户是否登陆过的信息。
4. 在实际实施的时候，考虑到有可能会被构造 ssoId 来做多次尝试登陆的问题，ssoId 长度可以考虑加大（255 个字符），这样被暴力尝试破解的机会几乎为 0。
5. 这种方法不仅仅限制在 ofbiz，完全可以适用在其他 javawebapp，asp,php 等等。只要其他 web 应用的登陆统一到 ssowebapp 通过 ssoId 验证，就可以一样集成 sso，以 asp 为例子：我们只需要重定向到 http://CServer/hij/autoLogin.asp?ssoId=xxx，即可。(在实际应用中，我们利用这种方式集成了旧有 notes，asp,php 等系统。)

4、备注

- 补充几点对于本文的解释：
- Q1.如果一个用户在原有的几个系统中都有不同的用户名？SSO 如何处理这个情况？

我们的这个简单实现不能解决这个问题。

- Q2.原系统的登录是否都要在 SSODB 维护，应用系统的 db 中是还要维护 userLogin?

各个应用系统 DB 里面是没有 UserLogin 这样的表的，但是这样会带来一些问题，如果应用里面有和 UserLogin 建立 viewentity，需要进行修改，但是这样的 viewentity 很少，通常都是为了获取 partyId,但是由于 party 的数据还是在各个应用系统 db 里面，通常修改到 Party 既可以解决，另外有和 UserLogin 建立外键的 entity 定义，也需要修改，因为大部分数据库不支持跨 DB 的外键。

- Q3.如果统一在 SSODB 维护。userLogin 对应的权限，维护在哪里？

UserLogin 对应的 securitygroup 维护还是在各自的应用系统里面进行，因为一个 user 在各个系统里面的权限设置肯定有各种变化，不能在 SSO 中统一作。

第六部分、安全控制

ofbiz 的安全是建立在 party 之上的，是与 party 关联的

一、认证

1、实体

ofbiz 的登录与认证使用实体 UserLogin，UserLoginHistory，UserLoginSession，

- UserLogin 为基础实体，他记录用户的登录名，口令、party 等信息，
- UserLoginHistory 用于记录用户登录的信息，以便系统跟踪查阅，提高系统的安全性，
- UserLoginSession，用于保存当前的用户 Session，在 ofbiz 中若用户 session 超时，则系统将失效当前的 session，在失效之前先保存当前的 session 到 UserLoginSession 中。这在 ofbiz 表现为：当网页过一个阶段没有用时，若用户继续使用则出现注册对话框（说明 session 已经失效），当登录完成后界面又回到原来的状态（说明原来的 session 已经恢复）

2、应用

有关认证的应用在 org/ofbiz/commonapp/security/login 下，有关认证的类在 LoginEvents 中描述，有关维护在 LoginServices 中描述。

1）LoginEvents 中主要是 login、logout、checkLogin 等

- checkLogin，在总控程序(controlServlet,sitservlet)中要使用，用于检查请求是否当前的用户，这样提高了用户的安全性，在 Controller.xml 的脚本描述中，比如<security https="true" auth="true"/>，说明该请求要经过认证。
- login：用于登录并进行响应的初始化，在 ofbiz 中扩充、集成其他应用基本上都需要在这里做一定的工作，比如我们在 ofbiz 增加的电子邮件和论坛在这里都要进行响应的处理（因为一个独立的应用大多都有自己的认证系统）。
- logout，退出之前要失效当前的 session，session.invalidate();

2)LoginServices 主要是 createUserLogin，updatePassword、userLogin 等

二、权限

有关权限的服务，操作在/org/ofbiz/core/security 之下，最主要的是 OFBizSecurity，有关权限的操作都在这里。主要有三类： hasPermission， hasRolePermission，hasEntityPermission， hasRole。OFBizSecurity 是 ofbiz 中的基础之一，他在用户登录后用总控程序保存在整个应用环境中。这样用户在编写 servlet,Jsp，服务等都可以直接使用。

1、权限的定义方法

action 是指对某一对象的权限操作如：creat,update,delete 等

- 实体权限：实体权限是由实体+Action 构成，比如对 product 实体可以定义 product_admin、product_update、product_create 等

- 角色权限：角色权限要用到应用(application)的概念，比如在 ofbiz 中定义了三种应用 ORDERMGR、FACILITY、MARKETING，角色权限的定义方法为：ORDERMGR_ROLE_ADMIN、ORDERMGR_ROLE_CLERK 等

2、权限的访问

- hasPermission，检查当前用户是否有指定的权限
- hasRolePermission。他的使用比较复杂，他与应用程序有关，与角色有关没，与实体有关
- hasEntityPermission
- hasRole：为了和大多数的应用一致，我们另外增加了 hasRole，hasRole 和 tomcat 中的使用方法一致，在我们额外增加 Menu 功能中使用，和我们的 E-office 也一致。

2、

- 审记
- 权限分配
- 权限组
- 角色
- 资源

1、 ldap 安全的融入
4、 ca 系统的溶入

三、CA 认证

1、 在 url.properties 中定义 Https 的端口等

port.https.enabled=Y

port.https=8443

force.https.host=localhost

第七部分、会员管理

ofbiz 中的会员管理是根据一定的理论来建立模型的，具有很强的通用性。在会员管理中主要有两个表，一个是会员表，一个是关系表，还有一个是角色表，其他的表都是辅助的表

一、会员

会员可以是一个人，可以是一个长设机构，临时机构，可以是内部的也可以是外部的。
在会员管理中我们可以定义以下会员

2、 员工，本单位的员工
3、 内部机构，就是本单位的结构设置，包括公司、事业部，部门等。
4、 临时机构，比如根据项目组成的项目组等。公关小组等
5、 团队：
6、 客户
7、 客户单位

三、关系

会员都是有关系的，比如员工与公司是雇佣关系，员工与部门是从属关系，部门与公司是从属关系，

二、角色

角色是会员的一种属性，比如员工，可以是管理人员、技术人员、工程人员等。

第八部分、应用说明

一、 环境

1、全局 request 环境变量

request 中包含下列环境变量，这些环境有些是 ServletContext 取到的放到 request 中，以变在 view 中可以随时取到。

webSiteId
servletContext
SiteDefs.CONTROL_PATH
SiteDefs.CONTEXT_ROOT
SiteDefs.SERVER_ROOT_URL
SiteDefs.CURRENT_VIEW
从 ServletContext 放到 request 中
delegator
dispatcher
MailServletContext
security
从 session 取出，放到 request 中
userLogin

2、全局 session 变量

主要为系统登录后使用的环境，包括：
delegatorName
webSiteId
SiteDefs.LOGIN_PASSED
userLoginSession
SiteDefs.USER_LOGIN
appContext
loginName=username
sake.mail.username
sake.mail.password
sake.mail.smtpserver
sake.mail.incomingserver
sake.mail.storeprotocol
displayName
visit
USERNAME
PASSWORD

SiteDefs.PREVIOUS_REQUEST

3、全局 ServletContext 变量

ServletContext 环境是在应用初始化的时候建立的，系统启动以后是不变的，包括：dispatcher、delegator、security、MailServletContext，在应用的时候用户可以使用这些环境中的变量和方法。

二、 环境加载

1、JSP 客户端

1）安全及实体加载

在 JSP 文件中使用 javabean 来引用有关安全（security）和实体（delegator）的函数
<jsp:useBeanid="security"type="org.ofbiz.core.security.Security"scope="request"/>
<jsp:useBeanid="delegator"type="org.ofbiz.core.entity.GenericDelegator"scope="request"/>

2）登录环境加载

GenericValueuserLogin=(GenericValue)session.getAttribute(SiteDefs.USER_LOGIN);
AppContextappContext=(AppContext)session.getAttribute(“AppContext”);

3）环境变量

```
session.setAttribute("webSiteId",getWebSiteId());
```

2、Servlet 及 Event 环境加载

```
GenericValueuserLogin=(GenericValue)session.getAttribute(SiteDefs.USER_LOGIN);
LocalDispatcherdispatcher=(LocalDispatcher)request.getAttribute("dispatcher");
GenericDelegatordelegator=(GenericDelegator)request.getAttribute("delegator");
```

3、环境的引用。

第九部分、设计规范

一、设计规范

1、使用 html 设计用户界面，

然后通过 controller 完成界面的组合，完成初步的界面设计，界面设计是应用开发的一个基础，它有以下作用：

关于需求

- 协助用户理解需求。
- 方便于开发商与用户达成双方可以认可的需求。
- 用户界面是用户最容易发生变化的部分，采用界面设计，使用户参与到了应用系统的开发过程中，有利于化解开发人员和用户的关系，有利于开发出适合与用户的应用系统。

关于设计

- 具有逻辑关系的界面设计，可以使设计人员了解整个项目的全貌，了解整个业务之间的关系。方便设计人员对应用系统进行全面的分析和设计。界面设计是需求的一部分，也是设计的一部分，对需求和设计有比较大的促进作用。
- 应用界面对应用的设计是有影响的，因此界面有利于以后的总体设计少走弯路，使应用开发能够符合业务的需求。

2、应用分析与总体设计（用例，对象分析）

根据需求分析，调研报告、和应用界面，完成应用系统的总体设计。总体设计使用 UML 方法。

工作产品：应用关系图、实例图、数据实体图

3、整体设计完成后进行详细设计

这些设计包括

- 按照 ofbiz 的服务的格式，提取公用部分设计成为服务
- 按照 ofbiz 事件的模式，设计复杂的企业逻辑。
- 按照 ofbizminilanguange 的模式实际简单的企业逻辑

将企业逻辑设计成 ofbiz 的服务、按照 minilanguange 的方式设计事件，具有以下优点：

- 有利于应用开发人员、集成开发人员，界面开发人员，按照角色进行分工，提高开发的效率。以便使应用开发可以量化，比如界面设计人员，可以按照界面的数量核定工作量、集成开发人员可以按照开发的事件核定工作量。软件工程师可以按照设计出的服务核定工作量。
- 有利于应用系统开发规范化。服务和事件都有标准的格式和用法，减少程序设计人员的随意性，因此有利于应用开发过程规范化。
- 有利于实现（服务）应用之间的程序共享，有利于应用程序的修改。

4、应用集成

- 使用 controller 绑定企业逻辑和应用界面。

绑定企业逻辑和应用界面有多种方法。目前我们建议使用 ofbiz 的 region 结构，以后根据情况决定是否采用模板语言的方式（freemarker 和 velocity）

应用集成是由集成开发人员来实施的。

二、开发规范

1、命名原则

按照 ofbiz 的方式进行命名（和 java 命名规则是一致的）

2、界面显示

界面的显示不论是采用模板语言还是采用 JSP 都有明确的要求：在显示语言中只允许有以下语句：

- 显示条件
- 显示循环控制
- 显示数据引用
- 有关权限和角色的控制（标准化的）

对与数据库操作，业务逻辑等不允许在 JSP 中出现。数据库操作，业务逻辑这些操作在服务 service 和事件 event 中由 java 语言和 mini-Languange 来实现。

3、接口

在 ofbiz 中服务，Event 等接口已经规定。

第十部分、Ofbiz 基础应用说明

一、介绍

OFBIZ 是建立在普通框架上的一套企业应用，它使用普通的数据，逻辑和处理组件。这个应用的松散连接使得这些组件容易理解，扩展和自定义。

OFBIZ 的工具和框架使高效的开发和维护企业应用变得容易。对于这个项目的开发者和维护者来说，不需要过多的努力就可以很容易快速发行新的功能和维护现有的功能。当你有特殊需要的时候，自定义和扩展现有的功能模块也变得十分容易。

这个框架可以按照你的需要很容易的自定义应用，但是如果这系统不被作为开源软件发放，那么系统中的一些最重要的灵活性将变得没有意义甚至不可能。MITOpenSourceLicense 有权授予 OFBIZ 的相应的权利，如自定义、扩展、修改、重新包装、转售和一些此系统的其他潜在的使用。

这些行为没有约束因为我们认为他们需要有效的使用这种类型的软件。不象起它的开源许可，如 GPL，你的改变不可以作为开源发行。但是一些涉及所有权或保密信息的改变不应该公开发行。出于这原因 OFBIZ 使用 MIT 认证。想要了解更多的有关开源认证的信息可以浏览 theOpenSourceInitiative(OSI)网站 www.opensource.org

这个开源模型的另一个好处是我们从这个软件的使用者那里得到不断的反馈。我们已经从 OFBIZ 的使用者和潜在使用者那里得到无数的 bug，开发建议和更实际的商务意见。这个项目的一些重要的特性是受到通过信件和这个项目联系的意见和建议所启发的。

为了确保我们的功能具有实时行和有效性，对于任何一个我们编写的组件都从研究公共标准和普遍用途开始。这帮助我们支持和使用普通的词汇，给我们一个选择的即时宽度，这些只有通过对标准的处理和其他组织的努力才能够取得。在未来，围绕相同标准建立、组织内部和合伙人或其它组织内的系统间的相互灵活通讯之门被打开。

二、主要的应用组件

通过在一个大的社区里的合作和正在进行的开发努力，OFBIZ 希望包括的特性能够自动的应用于企业信息和知识的每一个方面。在项目的开发和设计的初期，我们为这个系统的基础查找好的初始数据模型。

在发现 TheDataModelResourceBook,RevisedEdition,Volumes1and2byLenSilverston 前，我们研究了其它的 ERP 和 CRM 系统并查阅各种各样的概要和系统说明书。我们花了几周的时间将这些书中描述的逻辑数据模型翻译成规范化的物理数据模型，这样我们就得到了最初的系统组织。自从那时起，这个数据模型在基于现实系统的使用和许多现存公共标准的兼容性上经历了无数次的修改。由于灵活的系统构架使得很容易改变数据模型。这对于此项目的继续发展是必需的并且使得根据自己的特殊要求改变数据模型变得相当容易。

高层的应用和应用逻辑组件几乎都被组织在叫做数据模型（datamodel）的相同结构中。一旦系统某个层的组织被理解，那么就很容易理解系统中的其它层。

下面是对系统主要功能区的简要概述。一些功能区的功能已经存在，所有的功能区都有满足那部分系统需要所定义的数据元素。当更多的功能被执行的时候，我们继续精炼数据模型，但是所有主要的数据元素定义都很恰当了。因此，要记住这里描述的一些而不是全部功能都是在这一点上被执行的。

1、CommonData

这个系统里的 CommonData 包括一些实体（entities），例如：GeographicBoundaries,UnitsofMeasure,StatusCodes,Enumerations 等等。大部分这种数据是输入数据（seeddata），当系统被安装和随时间的流逝系统几乎不发生什么改变时，这种数据被输入。Geographicboundary 和其它可应用的输入数据是基于 ISO 和其它的一些标准。

2、Content

Content 实体被用于追踪数据源并把它们组成常规的内容和知识（contentandknowledge）。它们包括一些概念，例如：信息的隔离和允许一个数据源被用到一些内容结构中的组织；内容的灵活组织包括自由来自 graphs 中的联系或在 trees,lists,namedmaps,templates 中的更限制的组织等；对于内容和数据源，meta-data 的说明可被用于暗中组织，明确描述信息。单独的内容块能够存在于各种文本中并且被标准的 MIME 类型和字符集所描述的二进制格式。

一但对于这个信息的一般维护工具适当，那么更高级的工具，例如：基于关键字，基于 meta-data，智能搜索或发掘自动创建额外结构的文本或 meta-data 被用于使企业扩大文档和知识管理。

内容实体也包括关于基于 web 内容的信息，例如：页面，和访问网站（或应用）的交互，关于发给这个网站的信息。出于安全，市场，可用性和其它原因，对于跟踪用户利用应用所作的操作是有用的。

3、Security

安全实体用于控制系统各部分的访问，包括用户登录账户，登录审计数据、许可数据等等。在特殊的规则中，额外的限制性安全通过 Party 实体和与 Parties 联系的各种其它数据被完成。例如：一个用户可能有权限浏览和修改所有的产品数据，而另一个用户仅仅可以浏览和修改一种产品的数据。

4、Party

Party 可能是一个人或一个团体。一个 PartyGroup 可能是一个公司，公司内部的一个组织，一个供应商，一个客户等等。描述 Parties 的信息或直接与 Parties 有关的信息都被包含在这些实体中。

一类相关的数据是联系机制，例如：邮政地址，电话号码，电子邮件地址，internet 链接。

另一类数据是 Party 所扮演的角色，例如：客户、供应商、雇员、管理者、商人等。一般情况下一个单一 party 和系统不同部分不同角色间有相关的联系。

Partycategory 里的另一类数据是有关 Parties 间通讯和协定的信息。它进入关系型管理区域，包括 Party 可能有的关于 issues 或 troubletickets 的信息。这些实体连同 WorkEffort 实体设计跟踪这种问题的研究和决定。

5、Product

产品实体包含有关出售或公司内部使用的产品的信息。产品可能是货物或是服务，不同种类的货物包括原材料、半成品、成品。产品信息包括有关产品的描述性信息，不包括描述的真实实例或实物，这些在存货项目中起作用。存货项目包含有关特定货物的位置、项目的状况、连载项目的序号或手头中的数量的信息，可用于答应无连载存货的数量。

产品可能被管理进产品类别。一个单一产品可能是多重类别的一个成员，甚至类别本身也可能是多重类别的子类别并且可以有多重子类别。产品可能和第三方联系制定概念，例如：variants,cross-sells,up-sells,marketingpackages 等。

种类（categories）可能和不同的目录（Catalogs）相联系。产品目录（aProductCatalog）本质上是特定一套被出售产品的所有信息的出发点。Promotionsandinventorymanagementoptionsareassociatedwitheachcatalogsothatdifferentsaleschannelscanbehavedifferentlyevenwiththesamesetofunderlyingproducts

为了建模产品经常有的不同种类的特性，有一些实体用来定义各种类型和实际能够用到产品中的类型。例如：你可能说这件衬衫是大号，蓝色的衬衫。尺寸和颜色是特性的类型，大号和蓝色是特定衬衫产品的实际特性。

多重价格可能和单一产品相联系，正如多重成本。对于不同的货币，不同的设备（或商店），不同的期限都可以定义不同的价格。

在 OFBIZ 中有一个好的方面是大范围的使用有效的日期测定。有两个字段用于表达有效的日期：起始日期和终止日期。在产品价格例子中，起始日期和终止日期表示价格在某日某时生效，在某日某时失效。这用于与价格变化的历史保持一致，有效的管理暂时的价格上涨。

类外在明确的表达价格中有一些实体和逻辑，逻辑按照价格规则使用实体。例如：你能够创建一个规则在特定的时间生效，按照这个规则出售某一种类的所有产品。或者，按照一个简单的规则对特定的客户或一类客户给出特定的价格。

6、Order

订单实体用于管理有关买卖和作为订单准备信息的信息。例如：对于特定产品或特性的请求可能由客户提交，这个请求将会被订单包中的请求实体所跟踪。

请求可能被跟踪并被转变成一个要求，此要求被用于创建一个 WorkEffort（一个任务），此任务将会满足要求，执行请求。一旦要求产生，一个引用就会随之产生，如果客户接受这个引用，那么这个引用就可以产生一个订单。一旦订单被执行，来自订单的发票就会产生。发票是下面所要描述的账目实体包的一部分。

订单由订单头、任何数目的订单条款、描述订单详情的调节结构所组成。和订单相关信息的不同部分和订单的头或单一或多重列条款联系。例子包括运送目的地和订单的运送参数，这些参数对于所有的列条款可能是相同的也可能对每个都是不同的。

调节结构包含使订单价格改变的情况的信息，而这种订单不是实际的出售或购买的货物或服

务。例子包括税金，运送，折扣，额外费用等等。Anadjustmentcanbeeitheraflatamount,aflatamountperquantity,orapercentageofthesubtotaloftheentireorderorthelineitemitisassociatedwith.税金和运送作为特殊事件被处理，每个调节结构指定是否它应该被包含在 sub-total 中对于税金来说或对于运送来说。

一旦发票被创建，订单被创建给自动付款时付款参数就会被跟踪。这对于应用信用卡或其它电子方式付款是非常有用的。如果没有指定付款参数，那么一个标准的开发票清单的处理将会使用。

7、Facility

设备（Facility）是建筑物或其它的物理场所。例子包括：仓库、商店、办公楼、大建筑里的单间、装在码头及其它等等。通常设备有联系机制，例如通信地址或电话号码。

设备能包含在设备组中（FacilityGroups），反过来，设备足也能够被其它的设备组所包含。组的例子包括连锁商店、地区、区域和对于产品销售和定价使用的特定组。

库存项目可以和一个设备甚至和设备里的一个特定地点联系。为了便于管理仓库空间和特殊库存项目的位置，设备位置能够被跟踪和管理它们包含的单独的库存项目。

Parties 能够和设备联系已表示人在什么地方工作，哪个组织控制或操作设备，谁管理设备等等。

8、Shipment

发货实体用于跟踪引入和流出的货物，从库存发布条款或接收条款进库存。

一个发货有多个发货条款组成，每个条款，象订单条款，表示一定数量的特定产品。当发货被接收时，它们与一个订购单连接，此信息在发货收据实体中被跟踪。

当为一个出货发布一个存货条款时，它和一个挑选列表联系，这个列表为从多重出货到更高效的通道准备摘选的路径在仓库或其它设备中。

出货包能够被创建表示一个单独的箱子或运送单元。一个单独的箱子能够包含多重出货条款，甚至来自于不同的出货假设（它们发往相同的目的地）的条款。

出货通道实体把一个出货行程分裂成多重的通道部分。一个通道部分对于出货来说是一个贸易载体，而其它的可能是一个私人载体或公司拥有的交易。

Accounting

账目实体根据年代和普遍接受的规则组织起来，例如：double-entry 账目，普通的带有等级账号、分类的分类账，交易的置入和相应的报关手续。这结构主要基于 OMGGL 标准和 OMGGL 标准的 AR/AP 扩展。这个标准很好的与其它诸如 ebXML 和 OAGIS 标准相关联。

账目实体被组织起来，以至于对多重组织的账目能够被管理。多重组织可能是多个公司或一个公司内部的部门或组织。每个组织有各种 GL 帐目与它联系，以至于它能够操作它自己账目控制图表的子集。为了灵活每个组织有它自己的一套分类账，即使分类账目的使用不利于允许系统自动生成和基于被标准过程和文档激发的商务事件的交易，例如：买卖订单，发票，存货调动，付款，收据等等。

有一些实体用来做预算和在特定的结算期反对现行的 GL 账目平衡的预算调和。也有一些实体用来跟踪客户结算期，另一些实体在特定时期为了账目保持概要的结果。

跟踪固定资产的实体也是账目实体包的一部分。它包括出固定资产的维护、时序安排之外的折旧信息等等。

9、Marketing

行销实体跟踪和行销活动有关的信息，例如：联系方式（邮件，电邮，或电话单）和跟踪代码。跟踪代码主要用于自动系统中跟踪客户来自哪里，并用于除分析包含广告、合伙人的特殊行销活动的分析效力外的代理用途。

10、WorkEffort

AWorkEffort 可能是下列事情中的一种，任务、工程、工程段、要做的项目、日历项目甚至一个工作流或一个工作流活动。

工作流（Workflows）是一个特殊的事件，它由 OFBIZ 工作流引擎管理。工作流实质上是一个半自动化的处理，此处理包括由电脑执行的自动行为和由人力或由人力控制的一些其它的外部系统执行手动行为。工作流引擎处理在特殊实体模型中按照工作流管理联盟的规范（WFMC.org）定义的工作流。XPDL 文件被输入进这些实体，然后通过各种方法调用在那里定义的工作流。工作流知道每件已经被做的事和需要做的事。

WorkEffort 实体包中也有一些实体知道时间片和特定 Parties 执行 WorkEfforts 的工资率。例外，各种其它的资源也能被诸如固定资产，设备等所跟踪。

当 WorkEfforts 用于产品的生产和修改时，被特定 WorkEffort 消耗和生产的产品和存货条款能够被跟踪。

11、HumanResources

人力资源实体用于跟踪职位、职责、技能、雇佣、终止、利益、培训、工资等级、工资册参数、成绩评论、履历和应用和与其它人力资源相关的信息。

12、ArchitectureandSystemOrganization

构架真正是一个对应用组件的管理和合成来说富有想象力的词。有一些作为 Java,J2EEandOFBIZCoreFramework 的一部分的不同的有用的工具，它们联合在一起使用充分高效地组织数据和商业逻辑，为其它系统提供接口，为人员创建用户接口来与系统交互。

13、EntitiesandServices

OFBIZ 中最基本的组件就是实体和服务。实体是包含一些字段并能够和其它实体联系的关系型数据结构。基本的实体符合目前的数据库结构。有一种叫做“view-entity”的实体能够创建来自其它实体的虚拟实体通过联合其它实体一起结合一些字段。这些结构能够用于总结，分组数据和通常在程序中为观察或使用而准备。

在一些构架中，数据或持久层单独由成百上千或数以百万计行代码组成，当系统被开发和自定义时这些代码被维护。随着实体引擎的引入，仅仅几千行的 XML 数据定义就可以容易的完成驱动动态的 API。甚至在几天之内毫无经验的程序员也可能成为一个高手，不必学习任何 SQL。

她一切为你着想！

你也许应经听到了一些“WebServices”的嗡嗡声，它已经传遍了软件业的所有角落。OFBIZ 系统不但使用这种服务模型与其它系统通讯，而且在系统内部使用这种服务模型提供给 acleanandeasy 使用设备来创建和运行商业逻辑组件。

服务是一个简单执行特定操作的过程。服务定义用于定义服务消耗和产生的输入和输出参数。在实际逻辑使用这个定义被调用之前，传递给服务的数据自动被校验。服务运行以后，结果也使用相同的方法进行校验。

通过使用 Event-Condition-Action(ECA)规则其它服务能够在一个服务运行的不同点上自动运行。这个规则表示哪个服务应该被调用和在什么环境下调用。这样就允许逻辑在不修改原始逻辑的情况下被扩展，并允许系统被干净的组织以至于每个服务执行一个简单特定的任务。

服务能够以一些不同的方式被执行，这样就使工程师们利用手头现有的工具开放变得更容易。也很容易跟踪系统中的逻辑组件，这些逻辑组件存在于几百个不同的文件中，甚至在公司内部使用的不同计算机中或合作公司的计算机中。

14、Workflow and Rules

工作流引擎用于自动商业处理，包括手动（人员执行）和自动（电脑执行）行为。工作流能被作为服务和自动行为调用，在工作流中可以调用服务允许无限制的灵活组合这些小的逻辑组件。

工作流工具的作用是自动操作和跟踪复杂的商业处理。它们能帮助确保你的方针被遵循，在混乱中重要的任务不会丢失，甚至适当尺度的管理经常操作。在 OFBIZ 中基于工作流的一个好例子就是订单管理。当订单制定时，一个工作流被启动并通过批准跟踪执行订单。利用这个工具容易看到对于一个特定的订单什么已经发生，根据订单需要做什么。

有一些其它的处理例子通过这例管理受益。例如：内容创建和发布，客户发行，抱怨和服务管理，销售环管理，其它一些包括在不同角色的不同人的合作的商业处理。

在 OFBIZ 中的规则引擎本质上是一个选择性的编程语言，它所基于的概念完全不同于过程编程语言象：Java,Basic,C,Cobol,等等。事实和规则被声明，然后以两种方法中的一种使用：1）寻问是否明确的主张是真的，显示所有主张的可能论据；2）介绍所有有用的信息并寻问规则引擎产生所有的它所知道的关于事实和规则给定的可能信息。

规则引擎被应用于一些不同的应用。它们对于决定支持和帮助人们分析复杂情况。使用规则作为约束和发现问题的所有有用的解决方案，例如：机场大门的分配或货车递送和 pickup 安排能够节省大量的时间和虑及规则的一致的应用，不管人的经验如何人们都有可能遗漏或忘记。

15、Web Framework and XML Mini-Languages

在 OFBIZ 内核框架中有一些有用的工具在于 web,client-serverand 基于企业应用的 peer-to-peer 的困难点上。一个为简化平坦数据使用的工具使整合遗留系统更容易。各种各样的工具存在用于管理和构件基于 web 的具有逻辑和表现灵活分离特性的内容和应用。这些工具遵循 J2EE 标准，其它的 Java 工具是系统可以容易的与人员拥护和其它系统间相互通讯。

为了使使用这些内核框架工具更容易，另一个叫做 XMLMini-Language 的组件已经被创建，这个组件允许工程师或其它用户创建简单限定结构的 XML 文件，这种文件能够被系统理解和执行。利用 Mini-Language“simple-method”表达逻辑需要仅仅用 Java 方法的三分之一代码，对于半技术用户可以容易的阅读和修改。处理形成输入，将服务组合成更大的服务，利用这个工具可以很容易的与

数据库进行数据交互。

16、webtools

ofbiz 非常适合做企业开发的应用平台,因为它提供了企业应用系统所需要常用的各种工具，除了核心的：MVC 实现，entityengine,serviceengine 以外，还有其他很实用的工具，这些对于我们开发,发布,维护系统都是非常有实用价值。这里介绍一下它的工具,这些工具都在 webtools 这个组件内，在你安装完毕以后，你可以通过 http://localhost:8080/webtools，使用默认帐户 admin 和密码 ofbiz 登陆即可。

- CacheTools

UtilCache 是 ofbiz 里提供的一种 Cache 机制，对于提供系统性能有很大的帮助。

  - CacheMaintenance

这里是维护各种 Cache 的设置，通过 clearcache 可以让你不用重启 server，就可以使一些修改立刻生效。比如说你修改了某个 Request 对应的 event 处理以后返回的 view，那么你只用 remove 对应的元素在 webapp.ConfigXMLReader.Request 里，就可以立刻生效了。在开发环境下由于一些文件经常被修改，所以我们可以设置 cache 的失效时间为 1 毫秒(如 cache 脚本文件的 script.BshBsfParsedCache)，这样就不用频繁手工 clearcache 了。

- DebugTools

OFBiz 利用 log4j 作为它的 debug/log 工具

  - AdjustDebuggingLevels

这里是维护 Debug 的输出等级，可以让你在不用重起 server 的情况下，修改等级，对于快速得找到已经生产化的 server 上出现的问题会很有帮助。

- EntityEngineTools

EntityEngine 是 ofbiz 的核心之一，这里提供的工具也最多。

  - EntityDataMaintenance

Entity 数据维护的 web 界面，可以在这里查找，修改数据，对于一些常用的数据维护是非常方便的。

  - EntityReference&EditingTools

用来修改 Entity 定义的几个工具，点击这个链接，会打开一个新窗口包含下列所有工具，如果你的 entity 定义上有什么问题，在这个窗口右面的 frame，会列出一些警告，可以参考这些警告做修改。而且这个工具对于理解 DataModel 会有非常大的帮助。

  - Check/UpdateDatabase
  - GenerateEntityModelXML(allinone)
  - SaveEntityModelXMLtoFiles
  - GenerateEntityGroupXML
  - SaveEntityGroupXMLtoFile

以上这几个工具，使用一下就明白它的具体应用了。

  - EditEntityDefinitions

提供了一个 web 界面供你编辑 entitymodel 的定义文件。（在 3.0 上目前有 bug，建议不要使用）

  - InduceModelXMLfromDatabase

如果你的需要从已有的数据库获取 entitymodel，可以使用这个工具。

- ServiceEngineTools
  - JobList
  - ScheduleJob

通过 web 界面提供一种 schedule 某个 service 的途径。

- WorkflowEngineTools
  - ReadXPDLFile

读取写好的 xpdl 文件，将其 import 到数据库中，供 workflowengine 使用。

- ServerHitStatisticsTools
  - StatsSinceServerStart

统计应用系统的状态，通过分析这些数据，可以方便的找出需要进行性能优化的 event,view。

第十一部分、OFBiz 项目-特征列表

一、概述

- 免费开源软件
  - 不须许可或许可维护费用
  - 没有卖主、服务供应商或应用锁住
  - 积极支持的团队
  - 能获得全部源代码
    - 可以了解每件事是如何工作
    - 可以快速地捕捉到问题
    - 可以改变任何你想改变的
  - MIT 开源许可
    - 不必开源你所改变的代码
    - 可以重打包、分发、甚至出售派生出来的软件
    - 可以说它是基于 OFBiz
- 基于的标准
  - 熟悉类似软件的人容易上手
  - 容易重用基于相同标准的现有软件
  - 容易与内部或协作系统集成
  - 基于 SunJava、J2EE；W3CXML、HTML、SOAP；WfMCXPDL；OMGGL、Party、Product、Workflow
- 所有应用在相同框架、工具和组件上构建
  - 不需学习和使用许多不同的技术
  - 不需进行应用集成
  - 不需处理由于集成不同技术带来的特点限制
  - 因为构件的一致性和易维护，因此节省大量的费用
- 基于灵活的和通用的数据模型

- 覆盖在商业里使用的所有主要实体
- 提供一个简化自定义数据的结构
- 实体名使用通用的术语，容易理解和使用

- 灵活有效地使用数据层
  - 无数据库限定；支持许多不同的数据库
  - 无需写多余的持久层代码和配置
  - 容易使用 XML 数据定义
  - 强大的 API 提供基于数据定义的不同行为的一般操作
  - 许多操作能用一行代码来完成，无需写支持代码
- 松耦合多层组件体系结构
  - 容易定制和重用组件
  - 容易通过现有组件的组合构建新应用
  - 容易查找基于一致模式的代码和其它组件
  - 因为依赖性定义和管理得好，所以不用暂停其它组件，就可替换组件
- 分布式体系
  - 容易扩充多服务器或服务器池
  - 能与其它系统无缝集成和通信
- 以逻辑层为基础的服务
  - 所有逻辑被设定为服务
  - 重用逻辑容易
  - 服务能自动作为 Web 服务公布
  - 使添加定制用户接口容易
  - 在多服务器上部署分布式系统容易
  - 与其它系统通信容易
- 高级 web 应用框架
  - 分离输入处理逻辑、查看数据准备逻辑、查看表示层模板
  - 支持许多不同逻辑类型，包括脚本语言和服务
  - 支持许多不同的表示层模板类型，包括 XML/XSLT、FreeMarker、Velocity、JSP 和其它
  - 为了安全和销售目的，记录所有浏览和页面点击
  - 从服务器启动以后及时统计通讯和性能数据

二、应用

- eComerce
  - 完美的 B2C 和 B2BeCommerce
  - 容易配置成安全或公开发布
  - 支持从 HTTP（不安全的）到 HTTPS（安全的）的自动转换，并返回受到保护的页面
  - 产品搜索
    - 关键字搜索
    - 根据建立了索引的产品关键字进行搜索
    - 在给定类别的产品里进行搜索；这样就使得有可能在搜索结果里只

显示活动类别里的产品

- 可以用所有或单个关键字进行搜索
- 搜索结果按索引进行排序
- 在建立索引和搜索时，可以配置成移除终止符，这样才能不影响搜索结果的目的
- 可配置成建立索引时移除后缀(如-y、-ies 等等)，以达到不影响搜索结果的目的
- 建立索引时，产品不同的域有不同的权重

- 产品类别浏览
  - 可展开的浏览数显示选择的类别，而在相关页面显示类别的具体内容
  - 不同的类别可以有不同布局模板，不同产品可以有不同的子模板
  - 缺少地一次显示 10 个产品，可以定义前一页和下一页（这些设置很容易改变）
  - 产品可以属于多个类别
  - 子类别可以属于多个父类别
  - 基于选择的目录的相关设置自动改变浏览的根类别
  - 所有产品、类别和目录都有 from 日期和 thru 日期
  - 产品、类别和目录数无限制
- 产品细节浏览
  - 显示大的产品图（如果有指定的话），图片是一个指向详细图片的链接（如果有指定的话）
  - 显示所有相应的产品信息，包括名称、长短描述、价格、是否有可用库存等
  - 显示所有跨区销售、向上交易、本产品的目标竞争产品、以本产品为目标的竞争产品、其他希望通过简单模板改变的相关产品
  - 有变量的产品（“虚拟”产品）为每一个选择类型特征显示一个下拉框，列有相关特征;为了能够处理可用的特征组合，在第一个下拉框显示所有可用特征，选择一个特征后又会产生新的下拉框
  - 对于有变量的产品能为第一个选择的特征类型的每个特征显示一个小图片;例如，第一个选择特征类型是颜色，那么对于每种不同颜色显示有不同的图片;当一个大图片与有变量的产品相关联时，根据下拉框的选择显示不同的图片
  - 为了完全浏览给定类别的详情，在当前类别里显示前一个、下一个产品链接
  - 有灵活的产品属性和特征附加构造信息，可以容易地按你的要求的方式添加和显示
- 特殊的类别
  - 在特殊页面上显示，象主页面
  - 例子包括头 10 个最畅销的、促销品、新品等
  - 这些类别隶属于当前活动的产品录
- 跨区销售和向上销售
  - 建模成产品之间联系的特殊类型
  - 可以包含竞争产品、行销等包
  - 为每个产品显示产品细节

- 购物车随意跨区销售
  - 购物车里当前所有选择产品可以随意跨区
  - 每次只显示三个；当超过三个产品时，在每个新页面选择一个组
  - 当一个产品添加到购物车后，就不再出现
  - 显示购物车清单页面
  - 显示在相应页面购物车下边的右边栏的小盒子（缺省地）
- 快速的再次定购
  - 再次定购列表从以前定购的产品生成
  - 根据订购数量和频率确定权重表
  - 缺省地，再次定购数量是所有该产品以前订购数量的平均值
  - 在任何特定时间只显示头五个
  - 当一个产品添加到购物车后，就不再出现
  - 显示在相应页面购物车下边的右边栏的小盒子（缺省地）

- 基于规则的促销
- 基于规则的价格
- 客户喜好
- 购物车和检验流程
- 历史订单
- 会员和行销数据

- 参与者管理器
- 参与者类型:人和组
- 搜索参与者
  - 参与者数据维护
  - 个人数据
  - 组织数据
  - 用户登录安全数据
  - 联系机制：电话号码、通信地址、邮件地址、Web 页面地址等等
  - 付款机制：信用卡、EFT 帐号
  - 参与者角色
  - 参与者关系
- 安全数据维护
  - 安全许可
  - 安全组
  - 用户登录组成员资格
  - 组权限的联系
- 行销管理器

- 种类管理器（Catalog）
- 设备管理器
- 订单管理器
- 会计管理器
- 还没有...
- 工作任务管理器
  - 追踪工作任务
    - 任务和待办事宜
      - 项目的分等级项、阶段、任务、子任务等等
      - 分配工作项给多个参与者
      - 追踪优先级、成本估算等等
      - 追踪估算和实际：启动时间、结束时间、持续时间
    - 日历事件
      - 管理共享和私有计划的日历事件
    - 工作流任务
      - 查看分配给你的所有任务
      - 查看分配给你所属角色或参与者组的所有任务
      - 更新任务里与你相关的状态
      - 基于你的状态更新，系统将自动更新任务状态
      - 可以添加自定义模板和视图，显示来自工作流过程上下文的每个任务或其它数据库数据的相关信息
    - 工作任务与成本效益分析的成本相关；为了管理成本和效益两者，工作任务又与要求或请求相关
  - 追踪请求
    - 目的是为了技术支持、特征、修复、信息、举证、建议
    - 把请求与需求联系起来
    - 把请求与工作任务（任务、项目等）联系起来
    - 每个请求由多项内容组成，每项内容包含要求的细节
    - 把多个项目内容打包成请求
    - 每个请求包含所需要的内容
    - 在成本效益分析中，请求是与效益关联的一方；成本与工作任务相关，因此你能追踪并管理着两个方面
  - 追踪要求
    - 用于内部管理产品的需要具有的特征
    - 基于请求或请求项目的通常更准确
    - 为了更能规范的描述将会创建什么，每个要求包含一个“用例‘
    - 需求也是是与效益关联的一方，但通常效益比相关请求更好理解
- 内容管理器

还没有...

- Web 工具

- Cache 工具
  - Cache 维护
    - 查看 cache 大小和命中/未命中统计
    - 清除所有 caches、单个 cache、甚至 cache 的某行
    - 清除所有过期 cache 入口
    - 管理 cache 参数如大小限制、失效时间、软参数等等
    - 查看每个 cache 里的每个元素
- 调试工具
  - 调整调试级别
    - 应用运行时，调整调试日志信息级别
    - 直至服务器关闭才改变
    - 要永久地改变，请使用 debug.properties 文件完成
- 实体引擎工具
  - 实体数据维护
    - 查找、浏览、创建、更新和移除实体里的数据
    - 依照实体定义动态工作
    - 用灵活的权限，控制所有实体或特定实体组的访问
  - 实体参考和编辑
    - 显示所有实体定义细节，包括字段、类型、表和列名、关系等
    - 在主要浏览帧里，实体按包的字母顺序排序
    - 在左边帧里，有包的字母顺序列表和所有实体的字母顺序列表
    - 用链接显示实体的关系，使得浏览数据模型容易
    - 编辑页面可以用来创建和修改内存中的实体定义
    - 比较实体定义与数据库（正象启动时做的那样）差异的页面，（可选的）能自动添加缺少的表和列到数据库中
    - 为了比较容易，实体和实体组定义的 XML 文件的写法模板是一致性（注意：这些也必须用于保存内存里的实体定义的改变）；这些模板也可用于输出信息到浏览器
    - 读数据库元数据和创建第一个传递的 XML 实体定义的模板，可以根据需要进行修改
  - XML 数据导出
    - 以 XML 文件方式从实体导出数据
    - XML 构成有：每个实体实例、域（attribute）对应一个元素，或者每个实体派生的域对应一个子元素
    - XML 文件可以保存在服务器上或通过浏览器查看或保存到客户端
    - 高性能、可扩充的流式输出技术在每个交付里能导出无数实体实例
  - XML 数据导入
    - 从 XML 文件的实体定义导入数据
    - XML 构成有：每个实体实例、域（attribute）对应一个元素，或者每个实体派生的域对应一个子元素

        - XML 文件可以从服务器的磁盘加载或通过浏览器的表单上载
        - 高性能、可扩充的流式和 SAX 基本输入处理技术使得每次导入的实体实例无限
- 服务引擎工具
    - 工作列表
        - 查看所有被调度的“工作”服务
        - 显示工作 ID、开始日期/时间、完成日期/时间、调用的服务名
    - 调度工作
        - 允许命名服务的手工调度
        - 可指定时间间隔大小和数量
        - 可指定绝对开始和结束日期/时间
        - 可手工添加数据到运行服务使用的永久上下文
- 工作流引擎工具
    - 工作流监视器
        - 查看所有正在运行的过程
        - 显示每个过程的包版本、过程版本、状态、优先级、开始日期等等信息
        - 可以向下钻取，了解所有作为过程组成部分任务实例
        - 显示每个任务的任务 ID、优先级、状态、开始日期、完成日期和委派等信息
        - 在任务管理器里，为每个任务链接一个任务管理页面
        - 在参与者管理器里，为分配到每个任务的参与者链接一个参与者管理页面
    - 读 XPDL 文件
        - 读、校验、并显示 XPDL 文件
        - 读的可以是服务器或任何 URL 上的文件
        - 能够只校验，或为工作流过程执行写数据到数据库
- 规则引擎工具
    - Logikus-运行规则集（RunRulesets）
        - 提供一个 web 化的用户界面来查询包含论据和归纳规则的规则集
        - 现在仅支持反向链接
        - 能一次确定一个结果或所有结果
        - 包括许多示例规则集供实践
- 数据文件工具
    - 查看数据文件
        - 从符合格式文件规定的平面文件显示数据
        - 可以回写数据文件，并校验格式定义；即读/写具有重现性
        - 可从 URL 或服务器上的文件加载数据和格式定义文件
- 杂项设置工具
    - 编辑自定义时间段
        - 查看、创建、更新、删除分曾的自定义时间段
        - 时间段能与一个组织参与者相关联，并且查看的时间段时可以通过 partyID 进行过滤

- 管理财政年、季、月、两周、周和任何自定义的周期类型
- 追踪期间号、期间名、与每个时间期间的开始（from）日期和结束（thru）日期

- 编辑枚举

- 编辑状态选项

- 服务器点击率统计工具
  - 服务器启动后统计数据
    - 显示对于每个资源、资源组或所有资源分配的服务器负载和性能的统计数据
    - 追踪关于不同类型资源（包括请求、事件和视图）的数据
    - 显示服务器启动后积聚的统计数据
    - 为 timebin，链接到一个页面显示同一份数据
    - 保留 timebin 数据供将来分析之用

设置运行环境

startWLS.cmd

在你下载的 OFBiz-app 包中./setup/bea 下面有对这部分知识的说明，但是他的 readme 说的非常简单，同事配置文件得例子也是用于 UNIX 环境下的。

正常安装你的 weblogic7.0.2（你可以选择自己得 weblogic 得版本，在 bea 得官方网站上能下载到一个月的限时版本，bea 这方面做得很好）；正常安装你得数据库，当然你可以选择自己的数据库形式，在 OFBiz 中对 sybase、orical、和 mysql 等数据库得结合和支持都比较好，我们这里选择的是微软的 MSSQL2000，并建立自己得数据库（createdatabaseOFBiz）。

把你下载好的 OFBiz-2.1.1-apps 包解压并且放入 bea\service_2.1.1（这是我自己得选择路径，你当然可以按照自己的需要改变）。

startWLS.cmd 是 weblogic 真正启动服务的脚本，放置在你自己建立的域的目录正下方的 startWebLogic.cmd 会在最后启动这个脚本。

要运行 OFBiz 必须在 weblogic 中设置 OFBiz 运行环境和用到得类库。weblogic 和 tomcat 不同 tomcat 的运行只需要将运行包放到制定目录就可以，然而 weblogic 必须将所有类包的位置加入到 weblogic 的 CALSSPATCH 中，这样在 weblogic 启动时就会自动将 OFBiz 运行必须的类库加载。OFBiz 运行必备的类库放置在 OFBiz_home\core 下面的所有 jar 文件。

具体做法是，打开 weblogic 的启动脚本%bea_home%\weblogic700\server\bin\startWLS.cmd 文件，在这个文件中 weblogic 定义了运行环境变量以及使用类库。我们对这个文件的改动有两处:

设置环境 OFBiz_home,形如 setOFBiz_HOME=C:\OFBiz

设置 CALSSPATH,CALSSPATCH 的设置是重要的环节，它需要将 OFBiz 运行时需要的类和方法全部包括进来。OFBiz 中有很多自己定义的类，所以你的 CALSSPATCH 会变得很长。我在定义的时候尽量将 OFBiz 可能用到的包全部加入到 CALSSPATCH 中了，结果由于一条命令语句过长导致在 win2000Server 上运行时报错。

下面继续设置 CALSSPATCH，CALSSPATCH 得配置主要是根

据%OFBiz_HOME%\setup\bea\wlserver6.1\config\mydomain\startOFBiz.sh 作为参考，改造成为windows 脚本如下：

```
setclasspath=%OFBiz_HOME%\core\lib\ofbcore-share.jar;%classpath%
setclasspath=%OFBiz_HOME%\core\lib\ofbcore-entity.jar;%classpath%
setclasspath=%OFBiz_HOME%\core\lib\ofbcore-service.jar;%classpath%
setclasspath=%OFBiz_HOME%\core\lib\ofbcore-extutil.jar;%classpath%
setclasspath=%OFBiz_HOME%\core\lib\ofbcore-workflow.jar;%classpath%
setclasspath=%OFBiz_HOME%\core\lib\ofbcore-rules.jar;%classpath%
setclasspath=%OFBiz_HOME%\core\lib\ofbcore-datafile.jar;%classpath%
setclasspath=%OFBiz_HOME%\core\lib\ofbcore-minilang.jar;%classpath%
setclasspath=%OFBiz_HOME%\commonapp\lib\commonapp.jar;%classpath%
    setclasspath=%OFBiz_HOME%\commonapp\etc;%classpath%
    setclasspath=%OFBiz_HOME%\lib\common\castor-0.9.3.9.jar;%OFBiz_HOME%\lib\common\hsqldb.jar;%OFBiz_HOME%\lib\common\postgresql.jar;%classpath%
    setclasspath=%OFBiz_HOME%\lib\common\mm.mysql-2.0.8-bin.jar;%OFBiz_HOME%\lib\common\log4j.jar;%classpath%
    setclasspath=%OFBiz_HOME%\lib\share\axis.jar;%OFBiz_HOME%\lib\share\clutil.jar;%OFBiz_HOME%\lib\share\wsdl4j.jar;%classpath%
    setclasspath=%OFBiz_HOME%\lib\scripting\bsh.jar;%OFBiz_HOME%\lib\share\jakarta-oro-2.0.6.jar;%OFBiz_HOME%\lib\share\velocity-1.3.jar;%classpath%
    setclasspath=%OFBiz_HOME%\lib\datavision\DataVision.jar;%OFBiz_HOME%\lib\datavision\MinML.jar;%OFBiz_HOME%\lib\datavision\jcalendar.jar;%classpath%
    setclasspath=%OFBiz_HOME%\lib\weka\weka.jar;%classpath%
```

上面给出的是根据 OFBiz 的 setup 中的提示写的脚本，但是这个文档不是为 OFBiz2.1.1 所写的，所以你会发现在运行时很多需要的类其实还没有被加载如你的 classpath。你还需要到 OFBiz 的文档中搜寻需要加入的类库包。我这里把 OFBiz 中的 core 目录和 lib 目录和 commapp\lib 目录下的所有 jar 包全部放入到 classpath 中以防止疏漏。

但是你会遇到问题了，如果你在使用 win2000Server 同时又和我一样不太清楚 OFBiz 的类包哪些是必须的哪些是可选的（因为 OFBiz 网站上要求加入的类库包很多），就要听听解决方案了。其实解决这个问题很简单就是将 OFBiz 的 jar 包重新打过，简单的方法就是将 OFBiz 的一个目录下的包用 WinRAR 解压工具打开，将整个目录中的其他包中的文件和文件夹一起拖进去就好了，或者将包解压并且打成一个大包。然后在你设置 CALSSPATCH 的时候只需要将这个你自己重新捆绑的包加入就好了，这样大大减少了 CALSSPATH 的长度。上面是个小插曲，你可以尝试采用其他方法或者目的是让你启动得时候脚本执行的一条语句不要过长。以上的问题在 WinXP 中不不会出现，你可以将原本 OFBiz 中的包一一加载。

我这里给出一个我写的脚本例子，其中你在 OFBiz 中找不到的 jar 包其实就是我重新将一个目录中的 jar 包打的大包，你可以自己重新来过:

```
    @rem**************************************************************************
@remOFBizclass_pathseting

setclasspath=%JAVA_HOME%\lib\tools.jar;%WL_HOME%\server\lib\weblogic_sp.jar;%WL_HOME%\server\lib\weblogic.jar;%classpath%
setclasspath=%OFBiz_HOME%\core\lib\ofbcore-share.jar;%OFBiz_HOME%\core\lib\core-lib.jar;%OFBiz_HOME%\commonapp\etc;%OFBiz_HOME%\lib\datavision\datavision.jar;%OFBiz_HOME%\lib\datavision\MinML.jar;%OFBi
```

```
z_HOME%\lib\datavision\jcalendar.jar;%OFBiz_HOME%\lib\compile\activation.jar;%OFBiz_HOME%\lib\compile\jboss-j2ee.jar;%OFBiz_HOME%\lib\compile\jdbc2_0-stdext.jar;%OFBiz_HOME%\lib\compile\mail.jar;%OFBiz_HOME%\lib\compile\servlet.jar;%OFBiz_HOME%\lib\compile\xerces.jar;%classpath%
setclasspath=%OFBiz_HOME%\lib\jotm\jotm.jar;%classpath%
    @rem**********%OFBiz_HOME%\lib\common
setclasspath=%OFBiz_HOME%\lib\common\lib-common.jar;%classpath%

@rem**********%OFBiz_HOME%\lib\jotm
setclasspath=%OFBiz_HOME%\lib\jotm\lib-jotm.jar;%classpath%

@rem**********%OFBiz_HOME%\lib\scripting
setclasspath=%OFBiz_HOME%\lib\scripting\lib-scripting.jar;%classpath%

@rem**********%OFBiz_HOME%\lib\share

setclasspath=%OFBiz_HOME%\lib\share\lib-share.jar;%classpath%
setclasspath=%OFBiz_HOME%\lib\share\junit.jar;%classpath%
setclasspath=%OFBiz_HOME%\lib\share\lucene-1.2.jar;%classpath%
setclasspath=%OFBiz_HOME%\lib\share\saaj.jar;%classpath%
setclasspath=%OFBiz_HOME%\lib\share\velocity-1.3.jar;%classpath%
setclasspath=%OFBiz_HOME%\lib\share\velocity-dep-1.3.jar;%classpath%
setclasspath=%OFBiz_HOME%\lib\share\wsdl4j.jar;%classpath%
setclasspath=%OFBiz_HOME%\lib\share\xalan.jar;%classpath%

@rem**********%OFBiz_HOME%\lib\weka
setclasspath=%OFBiz_HOME%\lib\weka\remoteExperimentServer.jar;%classpath%
setclasspath=%OFBiz_HOME%\lib\weka\weka.jar;%classpath%

@rem**********%OFBiz_HOME%\lib\worldpay
setclasspath=%OFBiz_HOME%\lib\worldpay\cryptix.jar;%classpath%
setclasspath=%OFBiz_HOME%\lib\worldpay\select.jar;%classpath%
setclasspath=%OFBiz_HOME%\lib\worldpay;%classpath%

@rem**********%OFBiz_HOME%\commonapp\lib
setclasspath=%OFBiz_HOME%\commonapp\lib\commonapp.jar;%classpath%
@rem*************************************************************************
```

以上是 OFBiz 用到的类库，你必须核对文件并且确保文件在制定的位置。这样才能将包中的类库准确的加载到 weblogic 的运行环境中。同时如果你需要其他的类库比如说你的数据库驱动程序你也要在这段中加入到 CALSSPATCH 中去，我们选择了 microsoft 的 JDBCforSQLServer2000Driver（你可以到微软得官方网站得到下载），同样我们假如要到 CALSSTH 中去，这个驱动包括三个类库包，下面是我加载的脚本语句，你可以看到我得驱动程序是放到那里了，你可以选择自己得路径。

```
    setclasspath=%OFBiz_HOME%\server\lib\JDBCDrivers\msbase.jar;%classpath%
setclasspath=%OFBiz_HOME%\server\lib\JDBCDrivers\mssqlserver.jar;%classpath%
```

setclasspath=%OFBiz_HOME%\server\lib\JDBCDrivers\msutil.jar;%classpath%

简单说,就是在 startWLS.cmd 中增加 OFBiz_HOME 变量和增加 OFBiz 的类库包到 weblogic 的 classpath 中

config.xml

config.xml 是在 domain 的目录中，例如我自己的 omain 是在 user_projects 的 mydomian 目录中。config.xml 是 weblogic 定义自己的域的配置文件，其中定义了 weblogic 的数据库连接池和应用发布的目录等，OFBiz 的应用就是在这里发布。我们对 config.xml 修改的地方主要有两个，第一是发布 OFBiz 的一些应用，我们为发布 OFBiz 的几个重要的应用就要在 xml 文件中加入发布的路径等信息。可把%OFBiz_home%\setup\bea\wlserver6.1\config\mydomain\startOFBiz.sh 文档作为参考，不过这个脚本是在 unix 的系统上运行的，所以你要更改成在 win 上的脚本格式，发布好后就是要配置运行时需要的数据库连接池。

我们在 OFBiz 中用到得数据库连接是 weblogic 提供得连接池，所以实际配置数据库连接池的工作是在 weblogic 中配置的，你可以通过 weblogic 的 console 来配置，也可以通过直接更改这个 XML 配置文件来实现。在本文档中主要介绍 MicrosoftSqlServer2000 数据库的配置。首先你要选择驱动程序，我选择的是微软的 JDBC 驱动（好像 weblogic 自带的驱动在 OFBiz 启动时不太正常，不过我不确定，你可以自己去尝试），上面我们已经将驱动程序的类库包加载到了 classpath 中了，现在可以直接通过 weblogic 的 console 直观的配置这连接池这部分（具体配置不在本文的介绍范围，请参看其他文档）。

配置好了数据库连接池后我们要将 OFBiz 中必要得应用和应用实例发布到你的 weblogic 上。下面是我整理的发布脚本语言，你要加入到你的 config.xml 中，注意这里要求写详细准确的路径，我的安装路径你可以更改。

```
<ApplicationDeployed="true"Name="OFBizWebTools"Path="C:\OFBiz\webtools\webapp">
<WebAppComponentName="webtools"Targets="myserver"URI="."/>
</Application>
<ApplicationDeployed="true"Name="OFBizCommonApp"Path="C:\OFBiz\commonapp\webapp">
<WebAppComponentName="commonapp"Targets="myserver"URI="."/>
</Application>
<ApplicationDeployed="true"Name="OFBizeCommerce"Path="C:\OFBiz\ecommerce\webapp">
<WebAppComponentName="ecommerce"Targets="myserver"URI="."/>
</Application>
<ApplicationDeployed="true"Name="OFBizImages"Path="C:\OFBiz\ecommerce\images">
<WebAppComponentName="images"Targets="myserver"URI="."/>
</Application>
<ApplicationDeployed="true"Name="OFBizCatalog"Path="C:\OFBiz\catalog\webapp">
<WebAppComponentName="catalog"Targets="myserver"URI="."/>
</Application>
<ApplicationDeployed="true"Name="OFBizOrderMgr"Path="C:\OFBiz\ordermgr\webapp">
<WebAppComponentName="ordermgr"Targets="myserver"URI="."/>
</Application>
<ApplicationDeployed="true"Name="OFBizPartyMgr"Path="C:\OFBiz\partymgr\webapp">
<WebAppComponentName="partymgr"Targets="myserver"URI="."/>
```

</Application>
<ApplicationDeployed="true"Name="OFBizWorkEffortMgr"Path="C:\OFBiz\workeffort\webapp">
<WebAppComponentName="workeffort"Targets="myserver"URI="."/>
</Application>
<ApplicationDeployed="true"Name="OFBizAccountingMgr"Path="C:\OFBiz\accounting\webapp">
<WebAppComponentName="accounting"Targets="myserver"URI="."/>
</Application>
<ApplicationDeployed="true"Name="OFBizFacilityMgr"Path="C:\OFBiz\facility\webapp">
<WebAppComponentName="facility"Targets="myserver"URI="."/>
</Application>
<ApplicationDeployed="true"Name="OFBizMarketingMgr"Path="C:\OFBiz\marketing\webapp">
<WebAppComponentName="marketing"Targets="myserver"URI="."/>
</Application>
<ApplicationDeployed="true"Name="OFBizContentMgr"Path="C:\OFBiz\content\webapp">
<WebAppComponentName="content"Targets="myserver"URI="."/>
</Application>

startWebLogic.cmd

startweblogic.cmd 位于%bea_home%\user_projects\mydomain\startweblogic.cmd，其中的 mydomain 是你自己定义的域的名字。在定义 JAVA_OPTIONS 变量的时候在后面加入 -ms64m-mx64m-DOFBiz.home=%OFBiz_HOME%

---

entityengine.xml 配置

这部分的配置是属于实体引擎的配置，在这里只要配置 default 使用的数据源。需要配置的地方有两处，本文中是针对 weblogic 与 OFBiz 的配置说明.

1）配置事务，我们使用 jndi 去调用事务，把文件下面部分的注释去掉

<transaction-factoryclass="org.OFBiz.core.entity.transaction.JNDIFactory">
<user-transaction-jndijndi-server-name="default"jndi-name="java:comp/UserTransaction"/>
<transaction-manager-jndijndi-server-name="default"jndi-name="java:comp/UserTransaction"/>
</transaction-factory>

以上的部分是说明使用事务的位置，不用讲你要给默认的事务说明部分注释掉。

2）配置数据库连接信息，entityengine.xml 文件中列出了很多 datasource 的配置例子，你可以选择其中的一个作为你的数据库连接。方法是将 datasource 的名字属性填写到上面对 delegator 的定义中。默认是<group-mapgroup-name="org.OFBiz.commonapp"datasource-name="localhsql"/>我们将它改造成<group-mapgroup-name="org.OFBiz.commonapp"datasource-name="LocalMsSQL"/>（注意下面我们将编写一段对 LocalMsSQL 数据源的说明）。

3）配置数据源，主要是以上定义的 default 对应的 datasource。我们应用 weblogic 提供的数据库连接池。所以要应用到 jndi 去连接 weblogic 的数据库连接池。在这里会遇到一个问题，在将来全部对 OFBiz 的更改全部完成后虽然可以正常启动了，但是你在调用服务的时候还是会有问题出现。所以在 weblogic 配置数据库连接池的时候要特别注意一下说明（我选用的是微软提供的 sqlserver2000JDBC 驱动程序）。

在配置 URI 的时候必须增加一下参数的设置：
jdbc:microsoft:sqlserver://myServer:1433;DatabaseName=myDatabase;**SelectMethod=Cursor**
数据库连接方式**必须是指针形式**
这里我们只给出一个现成的例子：

```
<datasourcename="LocalMsSQL"
helper-class="org.OFBiz.core.entity.GenericHelperDAO"
field-type-name="sybase"
check-on-start="true"
add-missing-on-start="true"
use-foreign-keys="true"
use-foreign-key-indices="true"
use-fk-initially-deferred="false"
join-style="theta-oracle">
<sql-load-pathpath="commonapp/db"prepend-env="OFBiz.home"/>
<jndi-jdbcjndi-server-name="localweblogic"jndi-name="isdp"isolation-level="ReadUncommitted"/>
</datasource>
```

FixOFBiz-core

由于在 OFBiz 的帮助文档中的介绍紧紧局限于以上的部分，我按照以上办法配置完全后发现还有错误的出现。当我通过浏览器访问服务器的 7001 端口时候系统出现 500 错误。于是需要更改 OFBiz 中的部分代码来改善以上的问题。
这是主要的报错信息：

```
<2003-6-19????12 ê±37??46??><Error><HTTP><101002>
<[ServletContext(id=4266158,name=webtools,context-path=/webtools)]Could
notdeserializecontextattribute
java.io.NotSerializableException:org.OFBiz.core.entity.GenericDelegator
at
java.io.ObjectOutputStream.outputObject(ObjectOutputStream.java:1143)
```

以上是 weblogic 报出的错误信息，其中大致意思是说找不到上下文信息。导致以上错误的原因是 weblogic 信息容器是敏感的本地 class 容器中的，GenericDelegator 的生存期是在共享的 classpath 中不能在每个应用中都起作用。解决方法是注释掉 ControlServlet.java 中的申请 thecachedclassloader 的语句，

```
//ClassLoaderloader=Thread.currentThread().getContextClassLoader();
//localCachedClassLoader=newCachedClassLoader(loader,getWebSiteId());
//Thread.currentThread().setContextClassLoader(localCachedClassLoader);
```

以后重新打包入 ofbcore-webapp.jar，然后在每个应用的 lib 文件夹下都拷贝一份，这样对 OFBiz 改造一番后 OFBiz 应该可以正常启动了.

OFBiz 安装与设置

简介

文档简要介绍 OFBiz 的初试化过程和配置选项。

简单快速的设置

你可以下载 OFBiz 完整发布版并遵循如下简单指令快速开始并运行。

从 Sun'sJavasite 下载并安装 Java2v1.3 或更高版本 JDK(不是 JRE),并设置 JAVA_HOME 环境变量。

下载 OFBiz 压缩档并解压缩到指定目录。这样会创建子目录 **ofbiz**.这就是 OFBIZ_HOME 环境变量需指向的目录。

进入 **ofbiz** 目录,Windows 系统，运行**"%JAVA_HOME%\bin\java-jarofbiz.jar"**；Linux/Unix 系统，运行**"$JAVA_HOME/bin/java-jarofbiz.jar"**,来运行应用 OFBiz.

ecommerce 应用程序可以在浏览器中输入 **http://127.0.0.1:8080/ecommerce**，WebToolsapplication 应用程序可以输入 **http://127.0.0.1:8080/webtools**，或者 Catalog 管理应用程序可以输入 **http://127.0.0.1:8080/catalog**.可以看到不同应用程序.

现在只是演示，详细设置请继续往下看。

外部包设置

同样需要 Java2v1.3or1.4JDK(不仅仅 JRE)java.sun.com.

OpenForBusiness 可以运行在一个简单的 servletengine(likeTomcat)上,也可以运行在完全 J2EE 容器(比如：Orion 或 JBoss 与 Jetty 一起)上.完全(full)J2EE 应用程序服务器不仅支持基本 EJB 容器,比如:JNDI,连接池,事务管理,不同资源和配置管理等等,更支持服务.很多其他 J2EE 应用程序服务器对这些不同特征都可用.OpenForBusiness 应该能够作到易于适应(adaptableto)不同容器.

现在,Jetty 是 OFBiz 的默认应用程序服务器,使用 JOTM 提供 JTA(分布式事务)支持,但是也可以用 JBoss 或其他 J2EE 应用程序服务器.

可以在 jetty.mortbay.com 下载 Jetty,从 sf.net/projects/jboss 下载 JBoss.从 jakarta.apache.org 下载 Ant.

要运行 Ant 脚本,和 Jetty,确保如下环境变量设置正确.

路径

最后一步就是确保所有目录都是正确的，特别是 JDK 路径。要确保不同的环境变量都设置了，如下：

环境变量名	目录	描述
JAVA_HOME	/usr/local/javaorc:\jdk1.4.1_02	Java2SDK 安装目录。
ANT_HOME	/usr/local/antorc:\tools\ant	Ant 安装目录,应该包含 bin,docs,lib,等;可选，只有你想 Build 才需要。
OFBIZ_HOME	/ofbiz/work/ofbiz	OFBiz 安装目录;包含 OFBizCVS 模块象 core,commonapp,webtools,lib,ecommerce,等;可选，只有当你不在 OFBiz 主目录运行它时才需要。

CATALINA_HOME	/ofbiz/work/catalina	Tomcat4/Catalina 安装目录,应该包含 bin,lib,common,work, 等;可选，只有当对 Tomcat 实现自动设置时才需要，因为 Tomcat 需要拷贝一些特殊文件文件。

文件位置

如果您从 CVS 取出这些模块，就需要从 OFBIZ_HOME/setup/ofbiz/build.xml 拷贝 build.xml 到 OFBIZ_HOME.buildOFBiz 需要的所有 jar 都包括在'lib'模块中.有些在运行时需要包含在 ClassPath 中，但不是全部.有些只是为了使项目 build 更容易并提供一个一致的平台。

如下：

位置
ofbiz/lib/common
ofbiz/lib/datavision
ofbiz/lib/jasperreports
ofbiz/lib/jotm
ofbiz/lib/scripting
ofbiz/lib/share
ofbiz/lib/weka
ofbiz/lib/worldpay
ofbiz/core/lib
ofbiz/commonapp/lib
ofbiz/commonapp/etc

另外，对这个还没完成的 appserver 来说，还需要 ofbiz/lib/compile 目录中额外的类.举个例子,Jetty 需要:xerces.jar,activation.jar,mail.jar,andjdbc2_0-stdext.jar.

如果有些疑虑，也不要太担心，OFBiz 提供了脚本来自动为不同应用程序服务器配置文件。一切都可自动完成。

你可能需要将额外的 JARs 和资源放在 classpath 中，比如 JDBC 驱动，OFBiz 已经为 Postgres,MySQL,SAPDB，HypersonicSQL 提供了 JDBC 驱动.

跑起来

ofbiz.sh 和 ofbiz.bat 在调用 catalina.sh 和 catalina.bat 前分别作了设置，这些脚本应该放在 catalian/bin 目录下，你可以在命令行下使用 run,start,stop,等参数。

初始化数据可以使用 URL:http://localhost:8080/webtools/control/main,然后点连接导入,或者 http://localhost:8080/webtools/control/install 直接导入.因为可以使用 entitygorup 的名字来支持多数据库，你必须输入 entitygroup 名字，用以决定那些数据需要导入到那个数据库，正如 entityengine.xml 文件设置那样。JSP 程序会把在 entityengine.xml 文件中指定目录下为 groupname 指定的所有.sql 和.xml 文件导入。所有实体的默认 groupname 是"org.ofbiz.commonapp"。点"LoadData",看到下面消息后点"Yes,LoadNow",等一会数据就会导入。

导入程序可以导入 XML 实体引擎文件，或者 SQL 文件，对 SQL 文件需要用";"分开 SQLINSERT 语句，SQL 语句需要列出所有列。INSERT 语句非常通用，所以应该能够在大多数数据库上工作。这些语句通过 JDBC 运行。使用 JDBC 数据映射达到通用。

默认的 admin 帐户信息在初试化文件中，导入后可以使用 admin,ofbiz 登录。登录后可以到 http://localhost:8080/webtools/control/main 使用 WebTools.

数据库配置

数据库配置在 entityengine.xml 文件中完成.

OFBiz 应用程序文档中带有 HypersonicSQL 数据库，运行于内存中的一个 Java 数据库。这个数据库很容易配置和使用，有利于演示用。并不适合处理大量数据，也不支持大型数据库的性能和容量。

很多开源数据库都达到不错的水准，提供很好的性能。最常用的是 MySQL 和 PostgreSQL.Postgres 支持所有数据库的特性：事物，外键，存储过程等，MySql 性能也不错，但是事物支持有限（还在改进），并且不支持存储过程。存储过程对 OFBiz 来说不是大问题，因为 OFBiz 不使用存储过程。但是事务支持对商业系统非常重要。另一个好的开源选择就是 SAPDB，并值得特别提到。SAP 决定使用户使用商业数据库更容易，而不是自己的开源数据库 SAPDB.虽然比不上 Oracle 或其它商业数据库，他仍然有很多优秀的特性。

OFBiz 也支持商业数据库。OFBiz 已经在 Oracle 上作过测试，应该能够在其它使用 JDBC 驱动程序的数据库上正常工作。

不需要为创建表或修改默认数据而写 SQL 脚本.当服务启动时实体引擎会创建不存在的表或者字段,如果在 entityengine.xml 文件中作了适当设置.

为使数据库表创建工作正常.字段类型必须在相应 fieldtypeXXX.xml(XXX 是数据库名)中设置.fieldtypeXXX.xml 在 **ofbiz/commonapp/entitydef 目录下,**还有包含实体定义 entitymodel*.xml 文件.

第十二部分：Soap 的用法

soap 在 ofbiz 中有两个方向的用法，一是 ofbiz 调用其他 webservice 中的服务，二是其他 soap 客户端调用 ofbiz 的服务，这两项使 ofbiz 可以很方便地与其他应用系统融合。

一、soap 应用的两种情况

1、ofbiz 将其他的 webservice 封装成服务，这时 ofbiz 的应用系统可以直接调用该服务，以访问其他的 Webservice：

- 将 webservice 封装为服务的例子如下

```
<service name="testSoap" engine="soap" export="true"
        location="http://nagoya.apache.org:5049/axis/servlet/AxisServlet"
            invoke="echoString">
    <description>Test SOAP service; calls echoString service on Axis server</description>
    <namespace>http://soapinterop.org/</namespace>
    <attribute name="message" type="String" mode="IN"/>
    <attribute name="result" type="String" mode="OUT"/>
</service>
```

说明该服务是一个通过调用 http://nagoya.apache.org:5049/axis/servlet/AxisServlet 中的 echoString 来产生。

2、其他的 soap 客户端可以通过 soap 访问 ofbiz，这时 ofbiz 相当于 Webservice 服务器

外部 soap 客户端，发出 soap 请求调用，这个请求调用中的调用程序为 SOAPService。SOAPService 通过 SOAPEventHandler，将封装的 ofbiz 服务分离出来并执行。SOAPService 是一个 request 请求，在 controller.xml 中的定义如下：

```
<request-map uri="SOAPService">
    <event type="soap"/>
    <response name="error" type="none"/>
    <response name="success" type="none"/>
</request-map>
```

因为 soap 的输入输出响应都由 soap 客户端来解释执行，因此这里没有 response 响应。由此可以看出 SOAPEventHandle，和 ofbiz 的服务进行绑定（要求 ofbiz 的服务 export 为 true）

- 在服务定义中加入 expor=”true” 说明该服务可以通过被外部的 soap 客户端调用

```
<service name="testScv" engine="java" export="true" validate="false"
        require-new-transaction="true"
        location="org.ofbiz.commonapp.common.CommonServices" invoke="testService">
    <description>Test service</description>
    <attribute name="message" type="String" mode="IN" optional="true"/>
    <attribute name="resp" type="String" mode="OUT"/>
</service>
```

二、Soap 应用例子

本例子在 ofbiz 和 delphi7 中调试通过，delphi7 中使用的是 Demo 下的 EchiService.

1、ofbiz 调用 delphi7 中的 soap 服务

- 编写调用请求程序并在 controller.xml 中注册，请求程序在 org.ofbiz.core.event.testenvent 中，具体如下：

```
public static String testSoap(HttpServletRequest request, HttpServletResponse response) {
        request.setAttribute("MESSAGE", "Test Event Ran Fine.");
        HttpSession session = request.getSession();
        LocalDispatcher dispatcher = (LocalDispatcher) request.getAttribute("dispatcher");
        Debug.log("Test Event Ran Fine.");
        Map serviceContext = new HashMap();
        String serviceName="testSoap";
        Map result=null;
        try{
              serviceContext.put("value","String");
              result = dispatcher.runSync(serviceName, serviceContext);
              }catch(Exception e){};
     return "success";
}
```

- 其中 result = dispatcher.runSync(serviceName, serviceContext)调用的是访问其他 soap 服务器中的服务。并在 services_test.xml 注册该服务：该服务的定义如下：

```
<service name="testSoap" engine="soap" export="true"
              location="http://localhost:1024/EchoService_WAD.EchoService/soap/IEchoService"
                    invoke="echoString">
        <description>Test SOAP service; calls echoString service on Axis server</description>
        <attribute name="value" type="String" mode="IN"/>
        <attribute name="result" type="String" mode="OUT"/>
    </service>
```

该 ofbiz 服务实际上调用的是 http://localhost:1024 服务器上的 Soap 服务 IEchoService。方法是 echoString。Value 是输入参数，result 是返回参数。

2、delphi7 调用 ofbiz 中的服务

- delphi 中使用 EchiService 例子中的 client 端程序，在地址栏敲入 ofbiz 的 SoapService 请求格式为：http://localhost:8080/E-office/sit/SOAPService

delphi7 中 EchiService 的 client 中，按 connect 发出 echoBoolean 请求，按 String 发出 echoString 请求，按 WideString 发出 echoWString 请求，这些请求实际上是 ofbiz 中的服务。SoapService 可以将这些服务从 soap 序列中分离出来。

- echoBoolean、echoString、echoWString 服务的在 org.ofbiz.core.event.testServices 中具体写法如下：

```
public static Map echoBoolean(DispatchContext dctx, Map context) {
        Map response = new HashMap();
        Boolean aa=new Boolean ("true");
        response.put("return", aa);
        return response;
}
public static Map echoString(DispatchContext dctx, Map context) {
        Map response = new HashMap();
        Object value = context.get("value");
        response.put("return",value);
        return response;
}
public static Map echoWString(DispatchContext dctx, Map context) {
        Map response = new HashMap();
        Object value = context.get("value");
        response.put("return",value);
        return response;
}
```

3、ofbiz 中 Soap 类型映射

当以电子方法交换数据时，进行交换的端点需要预先在两方面达成一致：*交互模式*和*类型系统*。前者与通信通道的体系结构（例如，点对点和多对一，或分块对异步）有关。而另一方面，后者是要在对消息进行编码和解码的过程中使用的达成一致的数据格式。

除术语之外，让我来介绍一下整个系列中使用的用来表示 QName（qualified name，限定名）的名称空间前缀（适用于 SOAP 和 W3C 的 XML Schema）和约定。

2001 年发布了一个 SOAP 1.2 工作草案，但我们将只讨论 SOAP 1.1 的细节问题。这样，前缀 SOAP-ENV 和 SOAP-ENC 将分别引用名称空间 http://schemas.xmlsoap.org/soap/envelope/ 和 http://schemas.xmlsoap.org/soap/encoding/。

除非另有明确声明，一般我们假定前缀 xsd 和 xsi 分别引用名称空间

http://www.w3.org/2001/XMLSchema 和 http://www.w3.org/2001/XMLSchema-instance。

我们将以下面的格式编写带外部 URI 的 QName：

```
{uri}localpart.
```

一个示例：{http://xml.apache.org/xml-soap}Map

SOAP 和 RPC

当用 SOAP 执行 RPC 调用时，在 Web 服务请求者和提供者之间交换两种类型的有效负载：

远程方法的参数（RPC 请求）

返回值（RPC 响应）

从技术上来说，SOAP 附件和 SOAP 报头也可以组成 SOAP RPC 调用中的有效负载，但现在我们

可以安全地忽略它们。通常是使用俗称 *Section 5* 的方法对有效负载编码。虽然 SOAP 规范没把它指定为缺省的编码方法，但目前可用的所有 SOAP 框架都支持这个编码方法。

Section 5 编码描述了将对象图转换为 XML 的方法。如果一直坚持使用这种方法，就可以将 SOAP XML 重新构造回它的原始形态。另一种编码方法通过使用模式语言（如 W3C 的 XML Schema）约束有效负载。在前一个方法学中，事务中感兴趣的各方对序列化规则都持一致意见，而在后一个方法学中，他们是对消息的文字格式意见一致。在这个序列的第 2 部分中，您会看到一个示例，它演示由模式约束的 SOAP。

清单 1 和 2 详细演示并深入研究了 Section 5 编码。清单 1 中的 Foo JavaBean 在清单 2 中被序列化。

清单 1. Foo JavaBean

```
class Foo{
  int i;
  String s;
  public Foo() {}
  ....  /* property mutators and accessors not shown for brevity */
}
```

清单 2. Foo 对象的 SOAP 表示

```
<SOAP-ENV:Body>
  <ns1:eatFoo
    xmlns:ns1="urn:ibmdw:myservice"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <fooParam  xmlns:ns2="http://foo/type" xsi:type="ns2:Foo">
      <i xsi:type="xsd:int">1000</i>
      <s xsi:type="xsd:string">Hello World</s>
    </fooParam>
  </ns1:eatFoo>
</SOAP-ENV:Body>
```

清单 2 中的 SOAP XML 实例表示一个 RPC 调用，该调用将方法调用 `eatFoo` 调度到由 URI `urn:ibmdw:myservice` 标出的 Web 服务。元素 `fooParam`、`i` 和 `s` 被称为*访问器（accessor）*；它们是值的容器。多引用（Multireference）访问器不在这个定义的范围内；这些是一些空元素，使用“XML 指针语言”（XML Pointer Language）之类的机制（关于更多信息，请参阅下面的参考资料）来引用其它包含实际值的元素。

清单 2 中遍布的 `xsi:type` 属性提供访问器的子类型。通常，Web 服务提供者和请求者已经预先就每个 RPC 调用的参数数据类型达成了一致。要对 SOAP XML 实例进行正确的反序列化，有了这个预先达成的一致就够了，即使没有 `xsi:type` 属性也可以。但请考虑一下这样一个用例，在其中有一个特殊的方法既接受字符串也接受整数（例如可以将参数定型为接受 `java.lang.Object` 参数）。在这个案例中，为使对象正确进行反序列化，需要一个显式的 `xsi:type` 声明所包含值的类型。带有显式的 `xsi:type` 属性的访问器被称为*多态访问器（polymorphic accessor）*。

Apache SOAP 被设计为把所有的访问器都当作多态访问器看待。版本 2.2 继续使用多态访问器生成 SOAP XML 实例，但编程时它被设计为不用多态访问器就可以进行数据编出。稍后，在本文中我将简要说明如何执行这种操作。

xsi:type 属性将 QName 作为它的值。xsi:type 属性可接受的值有：

“XML 模式第 2 部分：数据类型”规范中的内置类型。

Apache SOAP 支持大多数内置类型并向后兼容该模式第 2 部分规范的所有版本的类型，这些版本中包括：

推荐版 （2001）

候选推荐版 （2000 年 10 月）

工作草案 （1999）

准确地说，Apache SOAP 并没有涵盖所有的内置类型，但它的确支持那些最常用的类型。在下一部分，我将看一下 Apache SOAP 支持的 Section 5 编码类型的详细清单。

表示一个用户定义的类型的任意 QName

这些用于表示混合类型如清单 2 中的 ns2:Foo。这种类型的序列化格式可以由模式文档来定义，或者，象 Apache SOAP 中那样，由定制的（反）序列化器来定义。

SOAP 1.1 扩展类型。

这些包括 SOAP-ENC:Array（表示数组成员的一个有序序列）和 SOAP-ENC:base64（表示一个 base-64 编码的字符串）。

清单 2 中的 encodingStyle 属性被用于指出所用的序列化模式。例如，Section 5 编码由 URI http://schemas.xmlsoap.org/soap/encoding/ 指出。Apache SOAP 有三个 encodingStyle 的内置支持：Section 5、XMI 和文字 XML（literal XML）。同时也支持定制的 encodingStyle。

但是，Apache SOAP 的 Section 5 编码支持并不全面。它仍然缺少一些功能，如稀疏（sparse）支持、部分传输支持和多维数组支持。

直到现在，我一直都在暗示进行数据交换的端点将能够魔术般地（反）序列化以 Section 5 编码的类型；但前提条件实际上依赖于对被来回传送的实体的数据模型要有某些非常规的一致。这基本上意味着，端点需要在一些“众所周知”的类型上达成一致。然而，这里有一个替代方案。不想受 Section 5 限制的软件开发者可以求助于*基于模式的序列化（schema-based serialization）*。将 SOAP 服务的接口和请求 / 响应消息的模式（一个或多个）一起发布，这种方法就可以生效。“Web 服务定义语言”（Web Services Definition Language，WSDL）是当前用于这种目的的实际标准。Apache SOAP 是无 WSDL 意识的，但 Apache SOAP 的后继工具箱 Axis（请参阅参考资料）有这种意识。

Section 5 编码

SOAP 没有为 Section 5 编码描述的 SOAP 类型定义任何语言绑定。相反，它的类型已足够通用，能够为 Java 编程语言和大多数其它主流编程语言中的一些典型数据类型建模。在这部分中，我们将深入研究 Apache SOAP 对编码和解码 Java 基本数据类型和任意 Java 类的支持。

但首先让我们来定义一些术语。*序列化（Serialization）*是将 Java 对象转换为 XML 实例的过程，而*反序列化（deserialization）*是从 XML 重新构造 Java 对象的过程。Apache SOAP 利用被称为*序列化器（serializer）*和*反序列化器（deserializer）*的构造来执行这些操作。在整篇文章中，我将使用简称*（反）序列化器*表示短语*反序列化器和序列化器*。

包含在 Apache SOAP 工具箱内的（反）序列化器的基集（base set）可以处理三组数据类型：*简单类型（simple type）*、*混合类型（compound type）*和*特殊类型（special type）*。术语*简单类型*和*混合类型*直接引自 SOAP 1.1 规范，它们在本文中的意思和在 SOAP 1.1 规范中的意思相同。我已经杜撰了*特殊类型*这个术语来表示与前面两组类型不太相符的 Java 类型。

简单类型

Apache SOAP 直接将这些类型作为访问器来序列化，只有文本节点作为子节点。一般情况下，XML 实例具有下列格式：

```
<accessor xsi:type="qname">
```

```
    <!-- data -->
</accessor>
```

表 1. Apache SOAP 中支持的简单类型

Java	SOAP	序列化器	反序列化器
String	xsd:string	内置*	StringDeserializer*
Boolean	xsd:boolean	内置*	BooleanObjectDeserializer*
boolean	xsd:boolean	内置*	BooleanDeserializer*
Double	xsd:double	内置*	DoubleObjectDeserializer*
double	xsd:double	内置*	DoubleDeserializer*
Long	xsd:long	内置*	LongObjectDeserializer*
long	xsd:long	内置*	LongDeserializer*
Float	xsd:float	内置*	FloatObjectDeserializer*
float	xsd:float	内置*	FloatDeserializer*
Integer	xsd:int	内置*	IntObjectDeserializer*
int	xsd:int	内置*	IntDeserializer*
Short	xsd:short	内置*	ShortObjectDeserializer*
short	xsd:short	内置*	ShortDeserializer*
Byte	xsd:byte	内置*	ByteObjectDeserializer*
byte	xsd:byte	内置*	ByteDeserializer*
BigDecimal	xsd:decimal	内置	DecimalDeserializer
GregorianCalendar	xsd:timeInstant!	CalendarSerializer	CalendarSerializer
Date	xsd:date	DateSerializer	DateSerializer
QName	xsd:QName	QNameSerializer	QNameSerializer

内置：org.apache.soap.encoding.SOAPMappingRegistry 中实现的内部类。

*：从 Apache SOAP 版本 2.1 以后。

!：在 2001 年 1 月，当“W3C XML 模式数据类型”规范（请参阅参考资料）转变为提议推荐状态时，timeInstant 被重命名为 dateTime。Apache 仍不支持 xsd:dateTime （请参阅参考资料）。

在继续往下讨论之前，让我们更仔细地检查一下表 1 中三点。首先，机灵的读者会注意到 char 不受支持（请参阅参考资料）。幸运的是，为它写一个定制的（反）序列化器很容易。其次，对 BooleanDeserializer 的更新使它能够分辨出 { "1", "t", "T" } 这组值表示布尔文字 true，而 { "0", "f", "F" } 表示 false。第三，缺省情况下 RPC 调用的接收方将反序列化为基本数据类型。为演示这一点，让我们假设 SOAP 服务器公开一个单独的方法，如下所示：

```
```

```
echoBoolean( Boolean b ) { .. }
```

当您使用 Boolean 参数调用 echoBoolean 时，将生成一个 SOAP Fault。

```
Exception while handling service request:
com.raverun.bool.Service.echoBool(boolean) -- no signature match
```

幸运的是，有一组反序列化器（xxxObjectDeserializer）将返回相应的包装器对象。为使它返回相应的包装器对象，您需要覆盖它的缺省反序列化行为。稍后，在本文中您将看到如何执行这种操作。

混合类型

从 Java 的角度来说，混合类型是带有成分（constituent）元素的类型。这些元素要么是只通过名称识别（例如，带有几个成员属性的 Java 类），要么是通过顺序的位置来识别（例如，象 Array 或 Vector 这样的 List 数据结构）。清单 3 显示了混合类型的一个 XML 实例。

清单 3. java.util.Hashtable 的 XML 实例

```
<hash xmlns:ns2="http://xml.apache.org/xml-soap" xsi:type="ns2:Map">
    <item>
          <key xsi:type="xsd:string">2</key>
          <value xsi:type="xsd:string">two</value>
    </item>
    <item>
          <key xsi:type="xsd:string">1</key>
          <value xsi:type="xsd:string">one</value>
    </item>
</hash>
```

表 2. 混合类型的 Section 5 编码

Java	SOAP	序列化器 / 反序列化器
Java 数组	SOAP-ENC:Array	ArraySerializer
java.util.Vector java.util.Enumeration	{http://xml.apache.org/xml-soap}Vector*	VectorSerializer
java.util.Hashtable	{http://xml.apache.org/xml-soap}Map	HashtableSerializer
java.util.Map	{http://xml.apache.org/xml-soap}Map	MapSerializer#
任意 JavaBean	用户定义的 Qname	BeanSerializer

*：Apache 2.1 中的 VectorSerializer 序列化为一个 SOAP 数组。但在 2.2 中，是把它作为它自己的 Apache SOAP 所属类型引入的。

#：MapSerializer 实际上把调用委托给 HashtableSerializer。

在表 2 中您可以看到，多复杂的 Java 类都可以用 BeanSerializer 来发送。但被序列化的 Java 类必须遵守 JavaBeans 设计模式；特别是，对于每个您想序列化的属性，它都必须有一个公有的、无参数的构造函数和 getter/setter 方法。在您的 getter/setter 方法不满足 JavaBeans 命名约定的情况下，您可以将它们通过 BeanInfo 类公开给 BeanSerializer。例如，如果您需要序列化类 Foo，请创建一个名为 FooBeanInfo 的类，它实现接口 java.beans.BeanInfo。请参阅清单 4 中的示例。

清单 4. FooBeanInfo 类

```
import java.lang.reflect.*;
import java.beans.*;
/*
 * Foo has a single String property by the name of "s".
 * And its setter and getter methods are called "_setS" and "_getS"
 * respectively, thus violating the JavaBean spec. Passing Foo to BeanSerializer
 * will cause this exception to be thrown:
 *        Error: Unable to retrieve PropertyDescriptor for property 's'
 *               of class "Foo".
 */
public class FooBeanInfo extends SimpleBeanInfo{
  private final static Class c = Foo.class;
  public PropertyDescriptor[] getPropertyDescriptors()
  {
    try{
      Class[] sp = new Class[] {String.class};
      Method s_setter = c.getMethod( "_setS", sp );
      Method s_getter = c.getMethod( "_getS", null );
      PropertyDescriptor spd = new PropertyDescriptor("s", s_getter, s_setter);
      PropertyDescriptor[] list = { spd };
      return list;
    }catch (IntrospectionException iexErr){
      throw new Error(iexErr.toString());
    }catch (NoSuchMethodException nsme){
      throw new Error(nsme.toString());
    }
  }
}
```

特殊类型

有许多 Java 类型不是十分符合上述类别的任何一种；SOAP 也可以处理这些类型。

Null 和 void

Apache SOAP 将通过添加 xsi:null 属性，将其设为 "true" 来序列化带空值的对象引用。它的缺省行为是忽略其它可能的 xsi:null 属性变体，例如：

xsi:nil="true"（按 XML Schema 第 1 部分）（请参阅参考资料）

xsi:null="1"（按 SOAP 1.1）（请参阅参考资料）

清单 5 包含一个带空值的、被序列化的对象引用的示例。

清单 5. 稀疏向量的 SOAP 表示

```
<myvec xmlns:ns2="http://xml.apache.org/xml-soap" xsi:type="ns2:Vector">
<item xsi:type="xsd:string">Hello World</item>
<item xsi:type="xsd:anyType" xsi:null="true"/>
</myvec>
```

在 Apache SOAP 2.1 中，发送带空成员的向量会导致 SOAPException 异常，因为当时不存在反序列化空引用的反序列化器。为实现这个目的，引入了 UrTypeDeserializer。VectorSerializer 将一个

空引用映射到 xsd:anyType（在 2001 XML Schema 推荐版中不推荐 xsd:ur-type）— XML Schema 内置数据类型层中的根类。

在处理字符串参数时，如果访问器是一个不带 xsi:null 属性的空元素，那么 Apache SOAP 将不会反序列化为一个空对象引用。例如，如果一个远程方法需要一个 String 参数：

```
public void eat(String s) //1
```

并且您假设这个请求被发送到一个 Web 服务，如：

```
<ns1:eat>
    <s xsi:type="xsd:string"/>
</ns1:eat>
```

它将会反序列化为一个空 String。

二进制数据

如果您需要传输二进制大对象（blob），您有三个选择：

字节数组

十六进制字符串

MIME 附件

表 3. 发送二进制大对象

Java	SOAP	序列化器 / 反序列化器
字节数组	SOAP-ENC:base64[#]	Base64Serializer
十六进制字符串	xsd:hexBinary	HexDeserializer
javax.activation.DataSource	xsd:anyType	MimePartSerializer
javax.activation.DataHandler	xsd:anyType	MimePartSerializer

[#]：在反序列化期间，SOAPMappingRegistry 的最新版本支持 {http://www.w3.org/2001/XMLSchema}base64Binary — SOAP-ENC:base64 的一个超类型（supertype）。

包装器类 org.apache.soap.encoding.Hex 被用于封装十六进制字符串。这个就是被（反）序列化的类。这个包装器类将接受十六进制值从 a 到 f 的小写和大写字母。只需确保您的十六进制字符串包含的字符数目是偶数。

Apache SOAP 支持将 SOAP 信封嵌入 MIME/multipart related 消息。所以二进制数据可作为 MIME 附件传送。这一点与“带附件的 SOAP 消息”（SOAP Messages with Attachment，SwA）规范一致（请参阅参考资料）。

文字 XML

如果将一个 XML 段作为字符串发送，内置的字符串序列化器将转义所有作为元素内容不合法的字符，其手段是用它们预定义的字符串实体替换它们 — 例如 < 将代替 <。要发送嵌入在 SOAP RPC 构造中的文字 XML，您有两个选择。您可以将 XML 字符串包在 CDATA 部分中。这样还允许您发送格式不好的 XML。第二种方法是将 XML 段加载到一个 DOM 中并将文档元素作为参数发送。Apache SOAP 将通过 XMLParameterSerializer 序列化 org.w3c.dom.Element 实例。

类型映射模式

在使用 Apache SOAP 编写 SOAP 客户机和服务器程序时，您很可能会遇到可怕的 java.lang.IllegalArgumentException 异常，并伴随有表 4 的列表中的消息。

表 4. 常见类型映射器错误消息

消息
No Serializer found to serialize a 'xxx' using encoding style 'yyy'
No Deserializer found to deserialize a ':Result' using encoding style 'yyy'
No mapping found for 'xxx' using encoding style 'yyy'

Apache SOAP 中的序列化和反序列化技术要依赖于注册表中定义的*类型映射*的可用性。类型映射定义如何在 Java 类（或基本数据类型）和 XML 之间进行来回转换。注册表是类 org.apache.soap.encoding.SOAPMappingRegistry 的一个实例。缺省情况下，SOAP 客户机和 SOAP 服务器将继承相同的类型映射集。

图 1. 上下文中的序列化 / 反序列化

对于下面的讨论，我将使用术语*入站（inbound）*和*出站（outbound）*来描述 SOAP 消息处理的方向。例如，在 SOAP 服务器的上下文中，序列化指的是生成*出站*消息的技术（mechanic）。这个简单的模型忽略了中间的 SOAP 服务器，但对我们的讨论没什么影响。图 1 描述了正在讨论的体系结构。

在描述与类型映射注册表交互的编程语法之前，了解一些关于注册表内部实现的知识会比较有帮助。开发者与之交互的注册表实际上是四个内部 Hashtable 的虚包（请参阅下面的列表）。

序列化器的注册表

反序列化器的注册表

Java2XML 的注册表

XML2Java 的注册表

序列化和反序列化是对称的操作；因此我将只讨论如何存储序列化类型映射，希望您能够根据我的说明推知如何对反序列化类型映射进行相应的操作。序列化类型映射由上面列表中的三个散列表 — A、C 和 D 中的条目表示。将一个序列化器 FooSerializer 作为示例考虑一下，它将 Foo.class 对象序列化为 SOAP 类型 "{http://someuri}foo"。如果您考虑用字符串 k 代表这个类型映射的唯一键，下面这些就是为序列化目的而创建的映射：

散列表 A：k 映射到 FooSerializer.class

散列表 C:k 映射到 {http://someuri}foo

散列表 A 和 C 都是以 Java 运行时类型和 encodingStyleURI 连接在一起生成的共享字符串作为键。散列表 A 维护实现接口 org.apache.soap.util.xml.Serializer 的 Java 对象集合，而散列表 C 管理一个 QName 集合。在序列化过程中，使用两个 API 调用查询注册表：

```
Serializer querySerializer( javaType, encodingStyleURI )
QName queryElementType (javaType, encodingStyleURI )
```

当 Apache SOAP 被定向为序列化一个 Foo 实例时，它调用 querySerializer 方法，并得到一个 FooSerializer。然后调用 FooSerializer 对对象进行数据编入，这时它将调用 queryElementType 来检索 xsi:type 属性值。

如果 querySerializer 无法返回任何 Serializer 实例，您会得到表 4 中的出错消息 1。另一方面，如果 queryElementType 不返回一个匹配的 QName，出错消息 3 将被返回。

在 SOAP 客户机上定义类型映射

您可以加强注册表，通过调用 SOAPMappingRegistry 实例的 mapTypes() 方法来处理特殊类型。方法说明如下所示：

```
mapTypes( String, QName, Class, Serializer, Deserializer );
```

如果您希望映射被同时注册到散列表 C 和 D，那么就要强制使用 QName 和 Class 参数。清单 6 包含 Apache SOAP 中缺省类型映射的示例。

清单 6. 示例客户机映射类型

```
HashtableSerializer hashtableSer = new HashtableSerializer();
mapTypes("http://schemas.xmlsoap.org/soap/encoding/",
         new QName("http://xml.apache.org/xml-soap", "Map"),
         Hashtable.class,
         hashtableSer,
         hashtableSer);
mapTypes("http://schemas.xmlsoap.org/soap/encoding/",
         new QName("http://schemas.xmlsoap.org/soap/encoding/", "Array"),
         null,
         null,
         new ArrayDeserializer());
mapTypes( encStyleURI, Qname, Foo.class, null, null );
```

清单 6 中的 Hashtable 映射是*对称映射（symmetric mapping）*的一个示例，在这种映射中入站和出站处理都已注册。第二个映射是*不对称映射（asymmetric mapping）*；它只将 ArrayDeserializer 注册到散列表 B。数组序列化发生在 SOAPMappingRegistry.querySerializer() 内，它用内省指出 Java 类型是不是一个数组，然后当时就返回 ArraySerializer 的一个新实例。最后的映射只是一个关联。

在 SOAP 服务器上定义类型映射

在服务器端，类型映射是在一个名为*部署描述符*的 XML 文件中定义的。Apache SOAP 实际上是根据部署描述符的内容构建一个 SOAPMappingRegistry 实例。 随本文一起提供的源代码分发包提供了一个 XML 语法模式（请参阅参考资料）。在清单 7 中您可以看到从这个模式抽取的内容，它显示了 map 元素的属性；它的属性反映了 SOAPMappingRegistry 的 mapTypes 方法的参数。唯一的不同是对于 map，qname 是必需的。

清单 7. Apache SOAP 部署描述符中映射元素的模式片段

```
<complexType name="mapType">
  <attribute name="encodingStyle" type="xsd:anyURI"/>
  <attribute name="qname" type="xsd:string"/>
  <attribute name="javaType" type="xsd:string" use="optional"/>
  <attribute name="java2XMLClassName" type="xsd:string" use="optional"/>
  <attribute name="xml2JavaClassName" type="xsd:string" use="optional"/>
</complexType>
```

被编译的定制类型映射

通过将单调乏味的类型映射手工声明固定在 SOAPMappingRegistry 的子类中，就有可能绕过它们。于是这个子类既可以用于客户机也可以用于服务器。清单 8 包含 SOAPMappingRegistry 子类的代码样本，这个子类提供了自己的数组（反）序列化器。

清单 8. 为继承 SOAPMappingRegistry 的公有类 MySmr 建立子类

```
  public MySmr()
  {
    super();
    registerMyMappings();
  }

  public MySmr(String schemaURI)
  {
    super(schemaURI);
    registerMyMappings();
  }

  private void registerMyMappings()
  {
    MyArraySerializer arraySer = new MyArraySerializer();
    mapTypes(Constants.NS_URI_SOAP_ENC,
             new QName(Constants.NS_URI_SOAP_ENC, "Array"),
             null,
             arraySer,
          arraySer);
  }
}
```

要使用调用者端的这个定制的 SOAPMappingRegistry，请将其作为参数发送到 Call 对象的 setSOAPMappingRegistry() 方法。在提供者端，删除所有的独立映射元素，并按清单 9 所示修改部署描述符。

清单 9. 部署描述符的更改

```
<isd:service ...>
  // other elements here
  <isd:mappings defaultRegistryClass="com.raverun.array.MySmr" />
</isd:service>
```

Apache SOAP 如何处理不同的模式名称空间？

在清单 9 中，您会注意到 SOAPMappingRegistry 类有一个重载的构造函数，这个构造函数使用一个 String 参数。该参数实际上是一个 URI，指出 W3C 的 XML Schema 语言的特定版本。Apache SOAP 接受的版本是在 org.apache.soap.Constants 中预定义的最终版本：

Constants.NS_URI_1999_SCHEMA_XSD = http://www.w3.org/1999/XMLSchema

Constants.NS_URI_2000_SCHEMA_XSD = http://www.w3.org/2000/10/XMLSchema

Constants.NS_URI_2001_SCHEMA_XSD = http://www.w3.org/2001/XMLSchema

如果您用 SOAPMappingRegistry 的无参数构造函数对它进行了实例化，缺省情况下，Apache SOAP 将把类型序列化为 1999 名称空间。调用重载的构造函数实际上只覆盖了 xsd 名称空间；SOAP 信封中声明的 xsi 名称空间仍未改变。这是因为该信封直接从 Constants.NS_URI_CURRENT_SCHEMA_XSD 和 Constants.NS_URI_CURRENT_SCHEMA_XSI（它们指向 1999 名称空间）获取名称空间前缀映射。这个缺点在最新的 CVS 源代码发行版中已经被修正（请参阅参考资料）。但从反序列化的角度来看，Apache SOAP 能够适当地处理列出的三个版本中任何一个的模式类型。

编码风格和类型映射

根据我们对类型映射表内部工作机制的了解，很明显，encodingStyleURI 在确定应该如何进行序列化和反序列化方面起着很大的作用。在这一部分，我将重点讨论一些影响 encodingStyleURI 的使用的编程问题。

首先，SOAP 1.1 规范明确声明“没有为 SOAP 消息定义任何缺省编码”。于是，在 Apache SOAP 中缺省 encodingStyleURI 为空。如果您没有在 Call 对象或者每个参数的声明中把它初始化为一个值，您可以通过调用 SOAPMappingRegistry 的 setDefaultEncodingStyle 方法做到这一点。这相当于向您的所有参数访问器添加本地范围的 encodingStyleURI 属性。

第二，考虑一下需要 SOAP 响应和 SOAP 请求的 encodingStyle 不一样这种情形。为便于讨论，请考虑清单 10 中的方法 countKids()。它的入站和出站 encodingStyles 分别是文字 XML 和 Section 5。

清单 10. 示例 SOAP 服务器的代码段

```
public int countKids( Element el ){
  return( DOMUtils.countKids(el, Node.ELEMENT_NODE) );
}
```

如果您想用清单 11 中的代码调用 countKids()，就会返回一个 SOAP 错误，同时伴随着一条消息：java.lang.IllegalArgumentException: I only know how to serialize an 'org.w3c.dom.Element'。

清单 11. RPC 调用的代码段

```
Call call = new Call();
Vector params = new Vector();
params.addElement( new Parameter("xmlfile",
                                 Element.class,
                                 e,
                                 Constants.NS_URI_LITERAL_XML) );
call.setParams( params );
call.invoke ( url, "" );
```

要理解发生这种错误的原因，您需要理解 Apache SOAP（在 SOAP 服务器上）用来确定在出站消息上使用何种 encodingStyle 的算法：

如果方法包装器访问器的 encodingStyle 属性可用的话，请使用它。

如果没有这个属性，请从第一个参数推出 encodingStyle。

所以，如果您的方法接受多个参数，当没有显式设置 Call 的 encodingStyle 属性时，您可能希望重新组织它们的顺序。为保证绝对安全，在 Call 对象中设置 encodingStyle 来响应远程方法的返回类型。

SOAP 消息中混合的 encodingStyle

首先，也是最重要的，SOAP 规范不会不接受参数内的混合 encodingStyle；encodingStyle 属性的作

用域很像名称空间声明。这种混合编码类型的一个有用的案例是一个 SVG 段数组。首先，使用 Section 5 对该数组进行序列化和使用文字 XML 序列化 SVG 成员看起来好像是合理的。不幸的是，这样是行不通的，因为所有的 Section 5 序列化器都被硬编码为只处理在 Section 5 encodingStyle 注册的类型。换句话说就是以 Section 5 编码的 XML 流中的任何内容都不可变，并且无法包含使用不同 encodingStyle 编码的 XML 段。而且，最近的错误修正（请参阅参考资料）也确保了混合数据结构（如 Array、Hashtable 和 Vector）内的值不可以是 Section 5 之外的任何 encodingStyle。

混合类型映射问题

我们将用您在 SOAP 类型映射中可能遇到的几个问题和怪现象结束本文。

反序列化期间缺少 xsi:type

在处理 Microsoft 的 SOAP 工具箱生成的入站消息时，您很可能会遇到下列错误：

```
No Deserializer found to deserialize a ':Result' using encoding style 'yyy'
```

这个典型的互操作性问题是由于访问器中缺少 xsi:type 属性引起的。Apache SOAP 2.1 发行版实现的解决方案是按照这个规则动态创建 xsi:type 值： 如果参数的访问器不在任何名称空间的范围内，就使用下面的 QName：

```
{""}/X
```

其中 X 代表访问器的 tagName，"" 代表一个空 URI。如果反序列化的访问器的名称空间 URI 存在的话，请使用它来代替这个空 URI。这项工作完成后，就会生成类型映射将这些特别的 QName 与（反）序列化器匹配。清单 12 演示了客户机和服务器的样本类型映射。

清单 12. RPC 调用的代码段

```
    [Client]
    SOAPMappingRegistry smr = new SOAPMappingRegistry ();
    StringDeserializer sd = new StringDeserializer ();
    smr.mapTypes (Constants.NS_URI_SOAP_ENC,
                  new QName ("", "Result"), null, null, sd);
    [Server]
    <isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:x="" qname="x:Book"
             javaType="com.raverun.Book"
             xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:x="" qname="x:price"
xml2JavaClassName="org.apache.soap.encoding.soapenc.FloatObjectDeserializer"/>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:x="" qname="x:isbn"
xml2JavaClassName="org.apache.soap.encoding.soapenc.StringDeserializer"/>
    </isd:mappings>
```

类型映射需要遵守下列约定：

对于基本数据类型以及它们的包装器对象（例如，int 和 Integer），不要在类型映射中指定 javaType。

对于混合类型，如果您的反序列化器在执行它的任务时要求了解 javaType（例如 org.apache.soap.encoding.soapenc.BeanSerializer），那么您可以指定 javaType。

关于这种行为的更多详细信息可以从 Sanjiva Weeraweena 发到 soap-dev 的帖子列表中收集，也可以从 Jim Snell 写的一篇文章中收集（请参阅参考资料获得这两者的链接）。

多引用类型

多引用类型并不是一种新的类型分组，但应该看作是简单和混合类型的一个方面。对于保持参数（特别是那些显示 DCE 风格的 [ptr] 语义的参数）间的一致性，这种类型很有用。Apache SOAP 能够反序列化多引用类型，但不能序列化为多引用类型。

多引用类型被序列化为两部分：*参数访问器（parameter accessor）*（一个空元素）和*值访问器（value accessor）*。 在清单 13 中，元素 varString 是参数访问器，而 greeting 是值访问器。注意，值访问器是方法包装器访问器 ns1:echoString 的同级元素。这一点很重要，因为 Apache SOAP 假设序列化图的根（方法包装器访问器）是 SOAP-ENV:Body 的直接子元素。如果值访问器排在第一位，那么您将遇到一个 Server.BadTargetObjectURI 错误。SOAP 规范提供了一个属性 SOAP-ENC:root 指出一个元素不是序列化的根，但 Apache SOAP 无法识别它。

清单 13. 多引用参数的一个示例

```
<SOAP-ENV:Body>
  <ns1:echoString
    xmlns:ns1="http://soapinterop.org"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <varString href="#String-0" />
  </ns1:echoString>
  <greeting
    xsi:type="xsd:string"
    id="String-0">Hello World</greeting>
</SOAP-ENV:Body>
```

XMI 编码

XMI（XML Metadata Interchange，XML 元数据交换）是一个在应用程序间共享 UML 模型的 OMG 标准。它在 UML 建模应用程序中最常用，这种应用程序要从 XMI 文件导进自己的模型，还要把自己的模型导出到 XML 文件。“IBM XMI 框架”（IBM XMI Framework）中的“Java 对象桥”（Java Object Bridge，JOB）允许把 Java 对象（反）序列化为 XMI 格式。这样，就可以把 XMI 看作 Section 5 编码的备选方案，虽然 XMI 很少用到。

第十三部分：JMX 的用法