

# JdonFramework 使用开发指南

板桥里人(banq J 道 <http://www.jdon.com>)

6.2.2 版本: 2010 年 3 月 12 日

JdonFramework 下载地址: <http://www.jdon.com/jdonframework/download.html>

技术支持论坛: <http://www.jdon.com/jivejdon/forum.jsp?forum=61>



## Jdon 框架能解决什么问题？

搞过数据库系统的人都知道，数据库系统中大量的基本功能无非是数据表的 CRUD 增删改查和批量分页查询，Jdon 框架结合对象设计和 J2EE/JEE 设计理念将这个看似简单功能开发过程抽象出来，放在框架中，并且随着应用程序一起运行，提供优化性能提升等。

日常企业信息化过程中，经常会一些表单数据输入数据库，使用 Jdon 框架可以在一两个小时内完成这样的应用，当你只需要快速建立一个软件应用系统，Jdon 框架正好适合你，而且拥有灵活的拓展性和方便的维护性。

如果你希望马上手一个 Hello World 的案例，可直接到“[开发一个简单 Jdon 应用系统](#)”章节。

请注意：Jdon 框架不是面向数据库的；而是面向模型分析设计（OOA/OOD）。Jdon 框架是基于如下一个软件生产环节中：

需求 Use case 分析；使用 Rose 或 Together 等 UML 工具；

面向对象的模型提取，域建模的确立，可使用 UML 的类图表达。

模型 CRUD 等基础功能快速完成以及构件/组件库组装；如果在这一过程中使用 Jdon 框架可提前保质保量完成，Jdon 之类快速开发框架是基于真正松耦合的设计架构，这样在以后迭代过程中才能应需求改变。软件松耦合和快速性讨论见：

<http://www.jdon.com/articheckt/coupling.htm>

反馈给用户确认；根据用户需求修改模型或界面或数据库或逻辑，因为有 Jdon 等框架形成架构的松耦合性，修改一处不会影响太多无关部分，具有好的可维护和可拓展，再次给用户确认；进入下一个循环。

快速开发框架和代码生成工具的区别：

软件生产是一个产品生成过程，只要是产品就要有质量要求，如果质量单纯靠人工保证是不行的，必须靠自动化的生产设备，而开发框架和组件或构件库则就是可以保证质量的生产设备；当然只有生产设备也是不行的，必须有良好的产品设计，必须符合客户要求，符合市场需求，那么产品设计就是域建模设计，域建模专家就象工厂的产品设计师或工艺师一样。

那么除了生产设备和产品图纸是不是就可以生长出高质量软件？不是，必须有不同级别软件人员，以及对他们的管理，也即软件工程管理，软件工程管理是和生产设备是互补的，如果生产设备自动化程度高，对人员要求低，管理就相对容易一些。

所以，目前对于软件生产这样一个产业，最重要的是摆脱依赖人工的局面，形成生产设备高度自动化。

当然，过分重视生产设备自动化的极端是必须指出的，这也是代码生成工具的产生原因，在目前，第一个阶段“松耦合”还没有完成，就想进入共产主义，显然有些急躁。

开发框架是基于软件最大追求松耦合基础上诞生，实现真正高质量；而代码生成工具则可能生成好或坏的代码。

软件产品生产 = 产品设计图（域建模）+ 自动化生产设备（开发框架或组件库）+ 不同级别的开发人员 + 软件工程管理。开发框架需要人员参与编程；而代码生成工具夸大生产设备作用。

如果你希望马上手看一个 Hello World 的案例，可直接到“[开发一个简单 Jdon 应用系统](#)”章节。

## Jdon 框架安装说明

在 Jdon 框架源码包中的 dist 目录下，有下列几个包，版本不同可能不相同，以实际 dist 目录下包为主：

|                       |                         |      |
|-----------------------|-------------------------|------|
| jdonFramework.jar     | Jdon 框架核心包              | 必须需要 |
| aopalliance.jar       | AOPAlliance 包           | 必须需要 |
| jdom.jar              | 读取 XML 的 JDOM 包         | 必须需要 |
| picocontainer-1.1.jar | Picocontainer 包         | 必须需要 |
| commons-pool-1.2.jar  | Apache 对象池包             | 必须需要 |
| log4j.jar             | Log4j 调试记录跟踪包           | 可选   |
|                       |                         |      |
| struts.jar ...        | struts 驱动包，支持 struts1.2 | 可选   |

## 在 JBoss 中安装

安装步骤：

1. 确保已经安装 J2SE 1.4 以上版本，然后设置操作系统的环境变量 JAVA\_HOME=你的 J2SE 目录
2. 下载 JBoss 3.X/JBoss 4.x
3. 安装 Jdon 框架驱动包：将 Jdon 框架源码包中的 dist 目录下除 log4j.jar 和 src 或 classes 目录下 log4j.properties 以外的包拷贝到 jboss/server/default/lib 目录下。
4. 安装 struts 驱动包，下载 struts 1.2，将 jar 包拷贝到 jboss/server/default/lib。  
或者使用 Jdon 框架例程 samples 中 SimpleJdonFrameworkTest 项目的 lib 目录。将该目录下 jar 包拷贝到 jboss/server/default/lib

对于具体 J2EE 应用系统，需要配置 Jboss 的数据库连接池 JNDI：

- 1.配置 JBoss 的数据库连接：

将数据库驱动包如 MYSQL 的 mysql-connector-java-3.0.14-production-bin.jar 或 Oracle 的 class12.jar 拷贝到 jboss/server/default/lib。

2. 选择 JBoss 的数据库 JNDI 定义文件：

在 jboss 的 docs 目录下寻找你的数据库的配置文件，如果是 MySQL，则是 mysql-ds.xml；如果是 Oracle；则是 oracle-ds.xml。下面以 MySQL 为例子。

将 mysql-ds.xml 拷贝到 jboss/server/default/deploy 目录下。

### 3. 修改配置数据库定义文件:

打开 jboss/server/default/deploy/mysql-ds.xml, 如下:

```
<datasources>
  <local-tx-datasource>
    <jndi-name>DefaultDS</jndi-name> <!--JNDI 名称 应用程序中使用 java:/DefaultDS 调用
-->

<connection-url>jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=UTF-8
      </connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>root</user-name><!-- MySQL 数据库访问用户和密码, 缺省是 root -->
    <password></password>
  </local-tx-datasource>
</datasources>
```

### 4.启动 JBoss

打开 jboss/server/default/log/server.log 如果没有错误, 一切 OK, 一般可能是数据库连接错误, 检查 mysql-ds.xml 配置, 查取相关资料, 弄懂每行意义。

至此, 可以将基于 Jdon 框架开发的 J2EE 应用程序部署到 JBoss 中。

一般是将\*.ear 或\*.war 拷贝到 jboss/server/default/deploy 目录下即可。

日志输出:

当运行具体应用系统时, 打开 jboss/server/default/log/server.log 时, 会看到很多 Jdon 框架本身的输出, 如果你觉得非常混乱, 可以关闭这些输出, 在 jboss/server/default/conf/log4j.xml 中加入如下配置行:

```
<category name="com.jdon.aop">
  <priority value="ERROR"/>
</category>
<category name="com.jdon.container">
  <priority value="ERROR"/>
</category>
<category name="com.jdon.model">
  <priority value="ERROR"/>
</category>
<category name="com.jdon.controller">
  <priority value="ERROR"/>
</category>
<category name="com.jdon.security">
  <priority value="ERROR"/>
</category>
<category name="com.jdon.bussinessproxy">
  <priority value="ERROR"/>
</category>
<category name="com.jdon.strutsutil">
  <priority value="ERROR"/>
</category>
```

## 在 Weblogic 等服务器中安装

只要将 Jdon 框架包和 struts 1.2 包安装到服务器的库目录下即可，或者配置在系统的 classpath 中即可。如果你的服务器没有 log4j 包，那么还需要 log4j.jar，并将 log4j.properties 放置在系统 classpath 中。

1. 将 struts 和 JdonFramework 所有驱动包拷贝到 Weblogic 的 common/lib 目录下。
2. 在 weblogic 的启动文件中加入如下命令：将 jar 包加入系统的 classpath。

```
set CLASSPATH=%CLASSPATH%;%WL_HOME%\common\lib\log4j.jar;  
    %WL_HOME%\common\lib\mysql-connector-java-3.0.14-production-bin.jar;  
set CLASSPATH=%CLASSPATH%;%WL_HOME%\common\lib\jdonFramework.jar;  
    %WL_HOME%\common\lib\jdom.jar;%WL_HOME%\common\lib\commons-pool-1.2.jar;  
    %WL_HOME%\common\lib\apollance.jar;%WL_HOME%\common\lib\picocontainer-1.1.jar  
set CLASSPATH=%CLASSPATH%;%WL_HOME%\common\lib\struts.jar;  
    %WL_HOME%\common\lib\jakarta-oro.jar;%WL_HOME%\common\lib\commons-validator.jar;  
    %WL_HOME%\common\lib\antlr.jar;%WL_HOME%\common\lib\commons-beanutils.jar;  
    %WL_HOME%\common\lib\commons-collections.jar;%WL_HOME%\common\lib\commons-digester.jar
```

3. 在具体 Web 项目打包部署时，需要将 log4j.properties 加到 WEB-INF/classes 目录下，更改 log4j.properties 中配置，使之日志输出到你指定一个文件中，注意：当时部署 log4j 日志不会激活 log4j，必须重新启动 Weblogic 即可（项目必须在 Weblogic 中）。

## 在 Tomcat 中安装

Jdon 框架在 Tomcat 下安装主要问题是 log4j 问题，下面是安装步骤：

将 struts 驱动包和 Jdon 框架包（包括 log4j.jar）拷贝到 tomcat/ common/lib 目录下。

将 Jdon 框架源码包 dist 目录下的 log4j. properties 拷贝到 tomcat/common/classes 目录。

配置 Tomcat 中运行 log4j 的关键是：检查 commons-logging.jar 和 log4j.jar 文件在 common/lib 目录，struts 驱动包中已经包含 commons-logging.jar 包。

上面步骤都正常了，可以启动 Tomcat，但是你会发现 tomcat/logs 下没有输出记录，因为我们已经使用新的 log4j，所以为了使得 tomcat 运行信息输出文件便于调试，编辑 /common/classes 下 log4j. properties：

1. 将 log4j.rootLogger=INFO, A1 前面加#，log4j.rootLogger=DEBUG, R 前去除#  
log4j.rootLogger=DEBUG, R  
#log4j.rootLogger=INFO, A1

2. 将 #log4j.appender.R.File=D:/jvaserver/jakarta-tomcat-5.0.28/logs/tomcat.log 一行前面的#删除，文件目录是绝对路径，更改为你自己的目录和文件。

3. 配置 Jdon 框架运行过程输出，在 log4j. properties 中下面一行：

log4j.logger.com.jdon=DEBUG

该配置将会显示 Jdon 框架的主要运行信息，如果你要关闭，只要更改如下：

log4j.logger.com.jdon=ERROR

重新启动 Tomcat，这时可以从 tomcat.log 看到输出记录。

## 如何判断安装启动成功

当启动 Jdon 框架任何一个应用程序，后台日志出现：

```
<===== Jdon Framework started successfully! =====>
```

表示你的系统配置和启动 Jdon 框架成功。

## 快速入门 ABC

### 一步到位

如果使用 Jdon 框架 6.0 以上，可以直接使用 Jdon 框架的 Annotation 一步到位。

举例如下：

在 HelloServiceImpl 加入注解：

```
@Service("helloService")
public class HelloServiceImpl implements HelloService {
    private UserRepository userRepository;

    public HelloServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public String hello(String name) {
        User user = userRepository.findUser(name);
        System.out.print("call ok");
        return "Hello, " + user.getName();
    }
}
```

HelloServiceImpl 中并没有对接口 UserRepository 实例化，只需在接口 UserRepository 的子类中加入注解@Component，如下：

```
@Component
public class UserRepositoryInMEM implements UserRepository {

}
```

客户端调用代码：

```
HelloService helloService = (HelloService) WebAppUtil.getService("helloService", req);
String result = helloService.hello(myname);
```

输出：Hello XXXX

原理图：

客户端调用代码:

```
HelloService helloService = (HelloService) WebAppUtil.getService("helloService");  
String result = helloService.hello(myname);
```

```
@Service("helloService")  
public class HelloServiceImpl implements HelloService {  
    UserRepository userRepository;  
    public HelloServiceImpl(UserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
    ..  
}  
@Component  
public class UserRepositoryInMEM implements UserRepository {
```



客户端代码中也没有将接口 `UserRepository` 子类 `UserRepositoryInMEM` 实例化创建后，赋给 `HelloServiceImpl`，这一切都是由 Jdon 框架悄悄在背后实现了。

具体全部代码见 JdonFramework 源码包目录下 `examples/testWeb` 项目。

更详细的 Annotation 注解使用见 [第 6 版更新说明](#)。

关于 AOP 拦截 Annotation 使用见 [6.2 版本更新说明](#)

## 详细讲解

如果你完全使用 Annotation 来使用 Jdon 框架，可能无需看这一篇，但是 XML 和 Annotation 各有优缺点，XML 有时更像水泥，可以将很多砖头联接在一起，无需重新编译源码，这样，在运行现场就可以通过更改 XML 中一些参数，达到修改程序运行功能的目的。

Jdon 框架有一个配置文件叫 `jdonframework.xml`，其中配置的是我们编写的 Java 类，格式如下：

```
<pojoService name="给自己类取的名称" class="完整类的名称"/>  
相当于  
@Service("给自己类取的名称")
```

配置有两个基本项：`name` 和 `class`，`class` 中写全 POJO 的全名；`name` 是供代码中调用这个服务的名称。



假如我们编写了一个类 `TestServicePOJOImp`，代码简要如下：

```
public class TestServicePOJOImp implements TestService{
    private JdbcDAO jdbcDao;
    public TestServicePOJOImp(JdbcDAO jdbcDao) {
        this.jdbcDao = jdbcDao;
    }
    public void createUser(EventModel em) {
        ....
    }
}
```

接口 `TestService` 代码:

```
public interface TestService {
    void createUser(EventModel em);
}
```

上面 `TestServicePOJOImp` 代码创建完成后，我们在源码目录需要创建一个叫 `jdonframework.xml` 配置文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE app PUBLIC "-//JDON//DTD Framework //EN" "http://www.jdon.com/jdonframework.dtd">
<app>
    <services>
        <pojoService name="testService" class="com.jdon.framework.test.service.TestServicePOJOImp"/>
    </services>
</app>
```

这样，在 `servlet` 或 `jsp` 或 `struts` 的 `action` 等客户端代码中，我们可以使用如下代码调用 `TestServicePOJOImp`，注意：以下代码没有具体 `TestServicePOJOImp` 类：

```
TestService testService = (TestService) WebAppUtil.getService("testService ", request);
testService.createUser(em);
```

以上步骤，只是简单展示框架的一个简要步骤，你可能没有觉得多了一个 `jdonframework.xml` 以后，好像比平常代码没什么不同，关键是：如果我们需要使用 `AnotherTestServicePOJOImp` 替换原来的 `TestServicePOJOImp` 类，只需要更改 `jdonframework.xml` 文件，而无须更改客户端代码，也无须重新编译项目了。

当然，还有另外一个优点，就是 `Ioc/DI` 依赖注射，细心的人已经注意到 `TestServicePOJOImp` 有一个构造参数如下：

```
public TestServicePOJOImp(JdbcDAO jdbcDao) {
    this.jdbcDao = jdbcDao;
}
```

如果不传入 `JdbcDAO` 实例，我们如何能在客户端代码中直接创建

TestServicePOJOImp 实例呢？原来只要我们在 jdonframework.xml 中再配置一个 JdbcDAO 类，概时框架就会自动帮我们创建 JdbcDAO 实例，并且传入 TestServicePOJOImp 实例中。

新的 jdonframework.xml 内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE app PUBLIC "-//JDON//DTD Framework //EN" "http://www.jdon.com/jdonframework.dtd">
<app>
    <services>
        <pojoService name="testService" class="com.jdon.framework.test.service.TestServicePOJOImp"/>

        <!-- 新增加的配置: jdbcDAO 是被 TestServiceImp 调用的，是为其服务的。 -->
        <component name="jdbcDAO" class="com.jdon.framework.test.dao.JdbcDAO"/>
        <!-- 相当于@Component -->
    </services>
</app>
```

再进一步，如果我们经常遇到一些类中需要一些常量或参数定义，那么可以使用如下配置：

```
<component name="jdbcDAO" class="com.jdon.framework.test.dao.JdbcDAO">
    <constructor value="java:/TestDS"/>
</component>
<!-- 如果组件带有配置参数，就不能用@Component 必须用XML配置，体现XML优势 -->
```

这时，要求 JdbcDAO 构造参数有一个字符串和参数，这样 constructor 的值 java:/TestDS 就在 JdbcDAO 被创建时注射到它的实例中。 JdbcDAO 代码如下：

```
public class JdbcDAO{

    .....
    public JdbcDAO(String jndiName){
        System.out.println("jndiname" = jndiName);
        .....
    }

    .....
}
```

### 启动 Jdon 框架

如果我们没有使用 XML 配置，全部使用 Annotation，就无需启动 Jdon 框架，这一章可跳过。

当配置了 `jdframework.xml`，我们需要在项目中启动它，有两种启动方式：一个是结合 struts 的 `struts-config.xml` 配置方式；另外一个是不结合 struts 的 `web.xml` 配置方式。

第一：web.xml 配置方式：

如果你不使用 Struts，可以通过 `web.xml` 下列配置来启动 Jdon 框架。

```
<context-param>
    <param-name> modelmapping-config </param-name>
    <param-value> jdframework.xml </param-value>
</context-param>
.....
<listener>
    <listener-class>com.jdon.container.startup.ServletContainerListener</listener-class>
</listener>
```

第二：结合 struts 配置方式(需要 struts 基础知识)：

在 `struts-config.xml` (或其它 struts 模块配置文件如 `struts-config-admin.xml` 等等) 中配置 Plugin 实现子类：

```
<plug-in className="com.jdon.strutsutil.InitPlugIn">
    <set-property property="modelmapping-config" value="jdframework.xml" />
</plug-in>
```



## 技术背景

### Java EE/J2EE 基本概念

J2EE 可以说指 Java 在数据库信息系统上实现，数据库信息系统从早期的 dBase、到 Delphi/VB 等 C/S 结构，发展到 B/S（Browser 浏览器/Server 服务器）结构，而 J2EE 主要是指 B/S 结构的实现。

J2EE 又是一种框架和标准，框架类似 API、库的概念，但是要超出它们。

J2EE 是一个虚的大概念，J2EE 标准主要有三种子技术标准：WEB 技术、EJB 技术和 JMS，谈到 J2EE 应该说最终要落实到这三个子概念上。

这三种技术的每个技术在应用时都涉及两个部分：容器部分和应用部分，Web 容器也是指 Jsp/Servlet 容器，你如果要开发一个 Web 应用，无论是编译或运行，都必须要有 Jsp/Servlet 库或 API 支持（除了 JDK/J2SE 以外）。

Web 技术中除了 Jsp/Servlet 技术外，还需要 JavaBeans 或 Java Class 实现一些功能或者包装携带数据，所以 Web 技术最初简称为 Jsp/Servlet+JavaBeans 系统。

谈到 JavaBeans 技术，就涉及到组件构件技术（component），这是 Java 的核心基础部分，很多软件设计概念（设计模式）都是通过 JavaBeans 实现的。

JavaBeans 不属于 J2EE 概念范畴中，如果一个 JavaBeans 对象被 Web 技术（也就是 Jsp/Servlet）调用，那么 JavaBeans 就运行在 J2EE 的 Web 容器中；如果它被 EJB 调用，它就运行在 EJB 容器中。

EJB（企业 JavaBeans）是普通 JavaBeans 的一种提升和规范，因为企业信息系统开发中需要一个可伸缩的性能和事务、安全机制，这样能保证企业系统平滑发展，而不是发展到一种规模重新更换一套软件系统。

J2EE 集群原理: <http://www.jdon.com/jive/article.jsp?forum=121&thread=22282>

至此，JavaBeans 组件发展到 EJB 后，并不是说以前的那种 JavaBeans 形式就消失了，这就自然形成了两种 JavaBeans 技术：EJB 和 POJO，POJO 完全不同于 EJB 概念，指的是普通 JavaBeans，而且这个 JavaBeans 不依附某种框架，或者干脆可以说：这个 JavaBeans 是你为这个应用程序单独开发创建的。

J2EE 应用系统开发工具有很多：如 JBuilder、Eclipse 等，这些 IDE 首先是 Java 开发工具，也就是说，它们首要基本功能是可以开发出 JavaBeans 或 Java class，但是如果要开发出 J2EE 系统，就要落实到要么是 Web 技术或 EJB 技术，那么就有可能要一些专门模块功能，最重要的是，因为 J2EE 系统区分为容器和应用两个部分，所以，在任何开发工具中开发 J2EE 都需要指定 J2EE 容器。

J2EE 容器分为 WEB 容器和 EJB 容器，Tomcat/Resin 是 Web 容器；JBoss 是 EJB 容器+Web 容器等，其中 Web 容器直接使用 Tomcat 实现的。所以你开发的 Web 应用程序可以在上面两种容器运行，而你开发的 Web+EJB 应用则只可以在 JBoss 服务器上运行，商业产品 Websphere/Weblogic 等和 JBoss 属于同一种性质。

J2EE 容器也称为 J2EE 服务器，大部分时它们概念是一致的。

如果你的 J2EE 应用系统的数据库连接是通过 JNDI 获得，也就是说是从容器中获得，那么你的 J2EE 应用系统基本与数据库无关，如果你在你的 J2EE 应用系统耦合了数据库 JDBC 驱动的配置，那么你的 J2EE 应用系统就有数据库概念色彩，作为一个成

熟需要推广的 J2EE 应用系统，不推荐和具体数据库耦合，当然这其中如何保证 J2EE 应用系统运行性能又是体现你的设计水平了。

## 高质量的 Java 企业系统

衡量 J2EE 应用系统设计开发水平高低的标准就是：解耦性；你的应用系统各个功能是否能够彻底脱离？是否不相互依赖，也只有这样，才能体现可维护性、可拓展性的软件设计目标。

为了达到这个目的，诞生各种框架概念，J2EE 框架标准将一个系统划分为 WEB 和 EJB 主要部分，当然我们有时不是以这个具体技术区分，而是从设计上抽象为表现层、服务层和持久层，这三个层次从一个高度将 J2EE 分离开来，实现解耦目的。

因此，我们实际编程中，也要将自己的功能向这三个层次上靠，做到大方向清楚，泾渭分明，但是没有技术上约束限制要做到这点是很不容易的，因此我们还是必须借助 J2EE 具体技术来实现，这时，你可以使用 EJB 规范实现服务层和持久层，Web 技术实现表现层；

EJB 为什么能将服务层从 Jsp/Servlet 手中分离出来，因为它对 JavaBeans 编码有强制的约束，现在有一种对 JavaBeans 弱约束，使用 Ioc 模式实现的（当然 EJB 3.0 也采取这种方式），在 Ioc 模式诞生前，一般都是通过工厂模式来对 JavaBeans 约束，形成一个服务层，这也是 Jive 这样开源论坛设计原理之一。

由此，将服务层从表现层中分离出来目前有两种可选架构选择：管理普通 JavaBeans（POJO）框架（如 Spring、JdonFramework）以及管理 EJB 的 EJB 框架，因为 EJB 不只是框架，还是标准，而标准可以扩展发展，所以，这两种区别将来是可能模糊，被纳入同一个标准了。

但是，通常标准制定是为某个目的服务的，总要牺牲一些换取另外一些，所以，这两种架构会长时间并存。

前面谈了服务层框架，使用服务层框架可以将 JavaBeans 从 Jsp/Servlet 中分离出来，而使用表现层框架则可以将 Jsp 中剩余的 JavaBeans 完全分离，这部分 JavaBeans 主要负责显示相关，一般是通过标签库（taglib）实现，不同框架有不同自己的标签库，Struts 是应用比较广泛的一种表现层框架。

这样，表现层和服务层的分离是通过两种框架达到目的，剩余的就是持久层框架了，通过持久层的框架将数据库存储从服务层中分离出来是其目的，持久层框架有两种方向：直接自己编写 JDBC 等 SQL 语句（如 iBatis）；使用 O/R Mapping 技术实现的 Hibernate 和 JDO 技术；当然还有 EJB 中的实体 Bean 技术。

持久层框架目前呈现百花齐放，各有优缺点的现状，所以正如表现层框架一样，目前没有一个框架被指定为标准框架，当然，表现层框架现在又出来了一个 JSF，它代表的页面组件概念是一个新的发展方向，但是复杂的实现让人有些忘而却步。

最后，你的 J2EE 应用系统如果采取上面提到的表现层、服务层和持久层的框架实现，基本可以在无需深刻掌握设计模式的情况下开发出一个高质量的应用系统了。

还要注意的是：开发出一个高质量的 J2EE 系统还需要正确的业务需求理解，那么域建模提供了一种比较切实可行的正确理解业务需求的方法，相关详细知识可从 UML 角度结合理解。

当然，如果你想设计自己的行业框架，那么第一步从设计模式开始吧，因为设计模式提供你一个实现 JavaBeans 或类之间解耦参考实现方法，当你学会了系统基本单元 JavaBeans 或类之间解耦时，那么系统模块之间的解耦你就可能掌握，进而你就可以实

现行业框架的提炼了，这又是另外一个发展方向了。

以上理念可以总结为一句话：

J2EE 开发三件宝: Domain Model (域建模)、patterns (模式) 和 framework (框架)。

## 组件框架比较

|              | EJB2/EJB3                           | Spring Framework                            | <a href="#">Jdon Framework</a>                                  |
|--------------|-------------------------------------|---|---|
| 灵活性<br>(松耦合) | EJB3 比 EJB2 更具灵活性, EJB3 支持应用系统 POJO | 支持应用系统 POJO, 框架基础功能不能替换                     | 支持应用系统 POJO, 框架本身可分离配置, 定制性更强                                   |
| 功能完整性        | 全面, 支持异步 JMS 分布式事务                  | 较为全面。有自己的表现层和持久层模板, 可支持异步                   | 基本完整, 表现层借助 Struts 实现。有自己简单的持久层模板                               |
| 单台性能         | 一般, 批量查询等大数据量业务处理须小心, 存在本地不透明缺陷。    | 一般, 框架本身组件无性能提升极致, 应用程序可配置 cache/Pool       | 好, 框架本身组件使用缓存提升性能, 应用程序可配置 cache/Pool, 批量查询专门优化, 适合实时性并发性要求较高应用 |
| 可伸缩性         | 可支持多台服务器分布式计算。                      | 不支持, 可依靠 EJB 实现                             | 不支持, 可依靠 EJB 实现   |
| 开发效率         | 学习曲线长, 导致熟练掌握难。借助商业开发工具可加快熟练者的开发速度。 | 较为复杂, 可挑选只适合自己的功能实现。当组件很多时, 需要照顾这些组件之间调用关系。 | 简单快速, 表现层编码很少。当组件个数很多时, 无需照顾它们之间的调用关系                           |
| 系统规模         | EJB2 适合大型系统; EJB3 适合中大型系统           | 适合中小型系统                                     | 适合小中型系统, 和 EJB 无缝结合, 可借助 EJB 支持中大型系统                            |
| 重量级别         | 重量, 正在减肥                            | 轻量偏重, 有可能继续增肥                               | 最轻量, 恪守简单快速原则   |

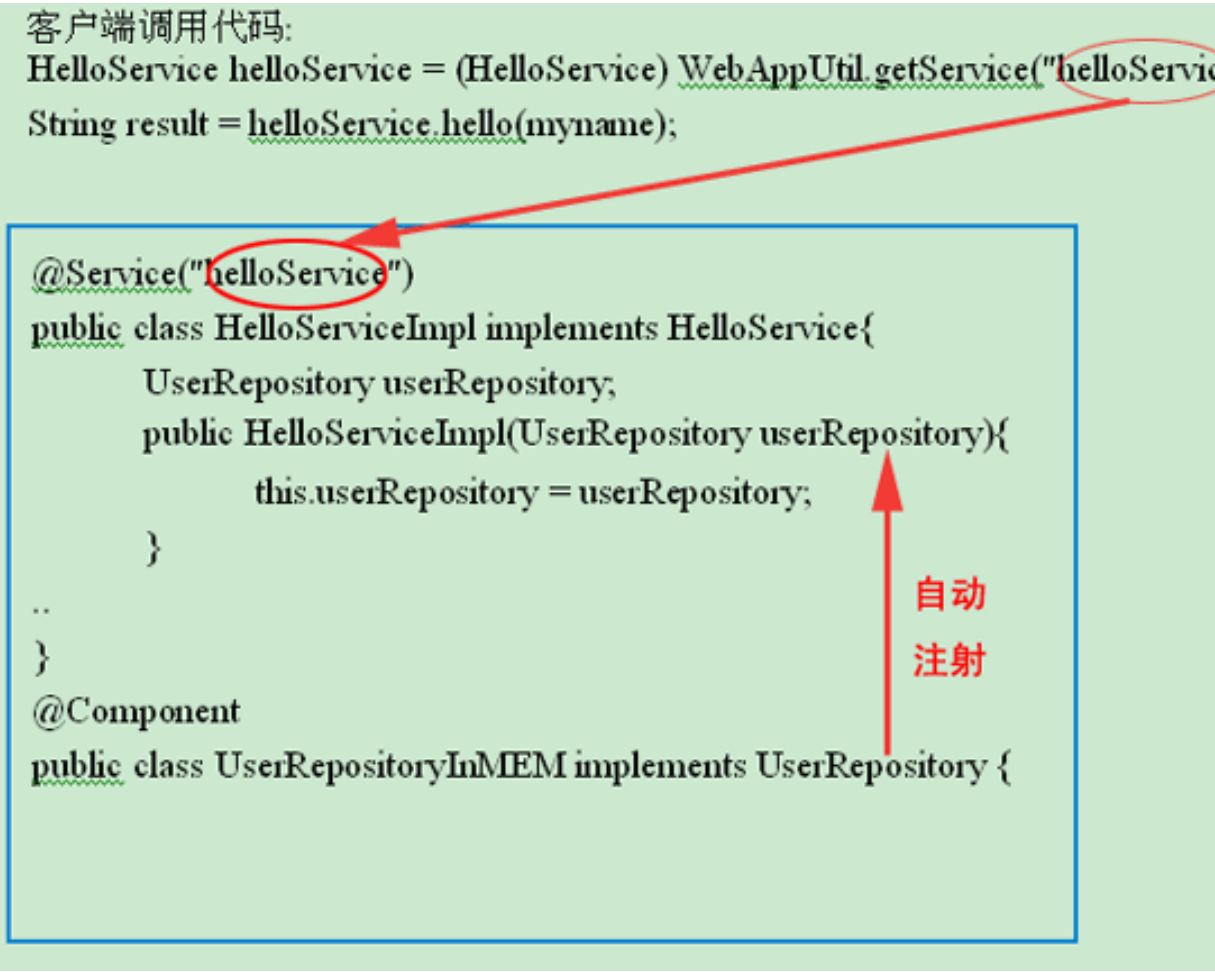
详细文章见: [http://www.jdon.com/articheckt/java\\_ee\\_architecture.htm](http://www.jdon.com/articheckt/java_ee_architecture.htm)

## 自动注射的 loc 框架

客户端调用代码:

```
HelloService helloService = (HelloService) WebAppUtil.getService("helloService");  
String result = helloService.hello(myname);
```

```
@Service("helloService")  
public class HelloServiceImpl implements HelloService {  
    UserRepository userRepository;  
    public HelloServiceImpl(UserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
    ..  
}  
@Component  
public class UserRepositoryInMEM implements UserRepository {
```



自动  
注射

代码案例

假设有调用者 B 和被调用者 A 代码如下:

调用者 B 类

```
package test;  
public class B {  
    AInfterface a;  
    public B(AInfterface a) {  
        this.a = a;  
    }  
    public void invoke() {  
        a.myMethod();  
    }  
}
```

被调用者 A 类:

```
package test;  
public class A implements AInfterface {  
    public void myMethod() {  
        System.out.println("hello");  
    }  
}
```



```
    }  
}
```

生成 B 类实例代码如下：

```
B b = new B(new A());
```

创建 B 的实例要逐个照顾到 B 类中涉及到所有其他类（如 A 类）的实例化，给编程者带来代码编写的琐碎工作，无法提高效率。

使用 Jdon 框架的 Ioc 模式后，B 类生成实例代码如下：

```
B b = (B) WebAppUtil.getService("b");  
b.invoke();
```

无需首先照顾其他类如 A 类的实例生成。B 的实例生成再也与其他类如 A 类没有任何关系了，实现松耦合。如果 B 类中涉及不只是 A 类，还有 C、D、E、F 等类，那么生成 B 类实例时，我们就无需关心这些类的创建了。

实现上述调用效果，需另外进行一下 jdonframework.xml 配置如下：

```
<app>  
  <services>  
    <pojoService name="b" class="test.B"/>  
    <pojoService name="a" class="test.A"/>  
  </services>  
  .....  
</app>
```

革命性优点

两个革命性优点：

1. 颠覆对象使用之前必须创建的基本定律，正象无需关心对象销毁一样，您可以无需关心对象创建。

Java 编程中类创建成实例的过程简化：（Class -> Instance）

使用 Jdon 等 Ioc 框架前：

编程者需要自己逐个解决这个 Class 相关涉及的其他 Class 的实例化。

使用 Jdon 等 Ioc 框架后：

无需编程者自己实现这种级联式、琐碎的实例化过程。

2. 松耦合；更换各种子类方便。面向对象编程之父 Grady Booch 说：对象最伟大之处在于其可被替代。而 Jdon 框架伟大之处是帮助你替代这些对象，甚至包括 Jdon 框架本身。

上例中，如果 Ainterface 有另外一个实现子类 AA 类，只要将 jdonframework.xml 中：

```
<pojoService name="a" class="test.A"/>
```

更换为：

```
<pojoService name="a" class="test.AA"/>
```

## Jdon 框架架构

Jdon 框架是一个真正轻量级别的开发框架，设计简单巧妙，适合快速开发各种架构的 J2EE 应用系统。它是一套符合当前国际水平的、面向构件开发的、国人拥有自主知识产权的中间件产品。

面向对象编程之父 Grady Booch 说：对象最伟大之处在于其可被替代。The great thing about objects is they can be replaced.

而 Jdon 框架伟大之处是帮助你替代这些对象，甚至包括 Jdon 框架本身。另外一个优点是：颠覆对象使用之前必须创建的基本定律，正象无需关心对象销毁一样，您可以无需关心对象创建。

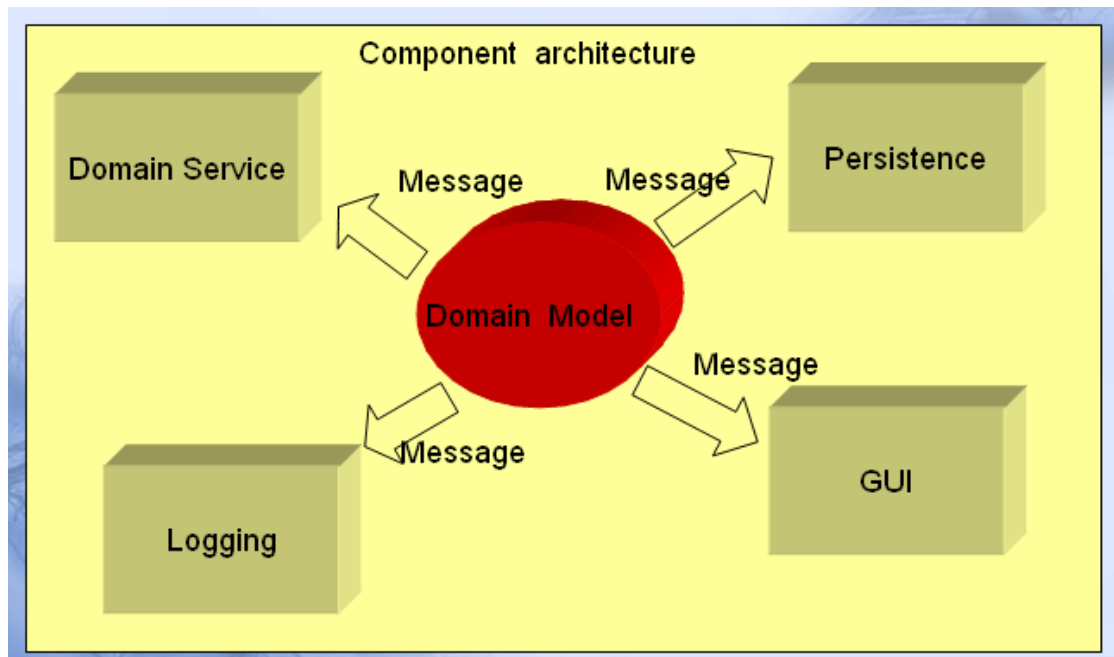
## 面向 DDD 的架构

为了更好地突出应用需求为主，Jdon 框架采取面向 DDD 的主要设计思想，将应用架构分为业务和技术两个部分：

@Model 领域模型，包括实体模型 值对象和领域服务，与技术架构无关。相当于鱼；生存空间是缓冲器中

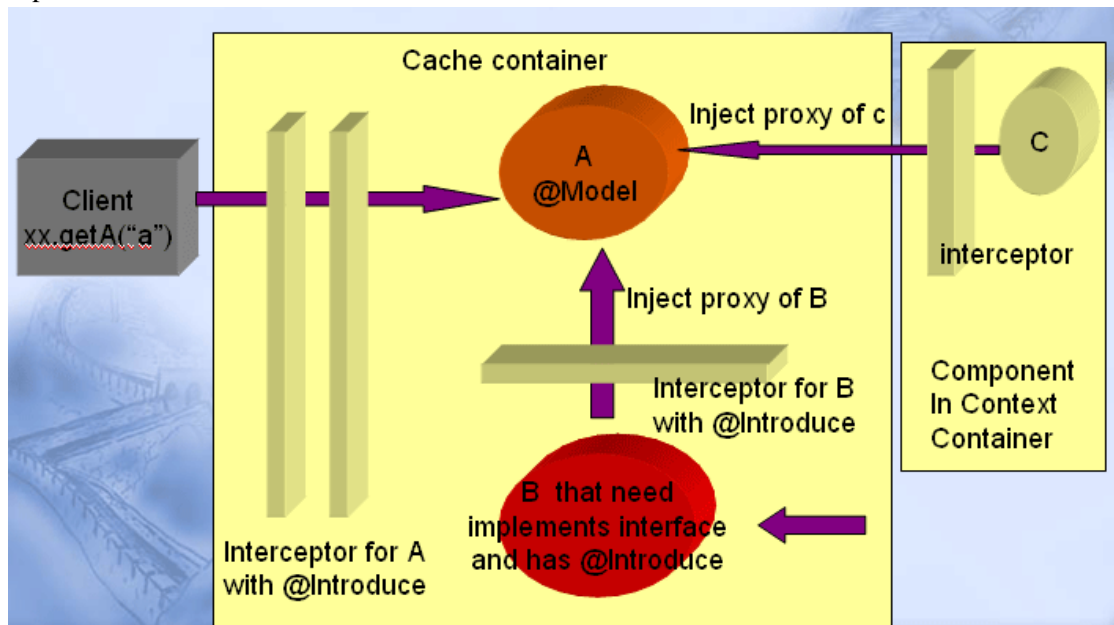
@Component 技术组件架构，用以支撑领域模型在计算机系统运行的支撑环境，相当于鱼生活的水。空间在 Context container,例如 ServletContext 中。

两者以 Domain Events 模式交互:异步 命令。



## 注射和拦截器 AOP 特点

(一) @Model: 模型中可以通过字段的 @Inject 将其他类注射进入, 包括 @Component 类。被注射的类如果有 @Introduce, 将再次引入拦截器。



```
@Model
public class MyModel {

    private Long id;
    private String name;

    @Inject
    private MyModelDomainEvent myModelDomainEvent;

    @Inject
    private MyModelService myModelServiceCommand;

    @Introduce("message")
    public class MyModelDomainEvent {

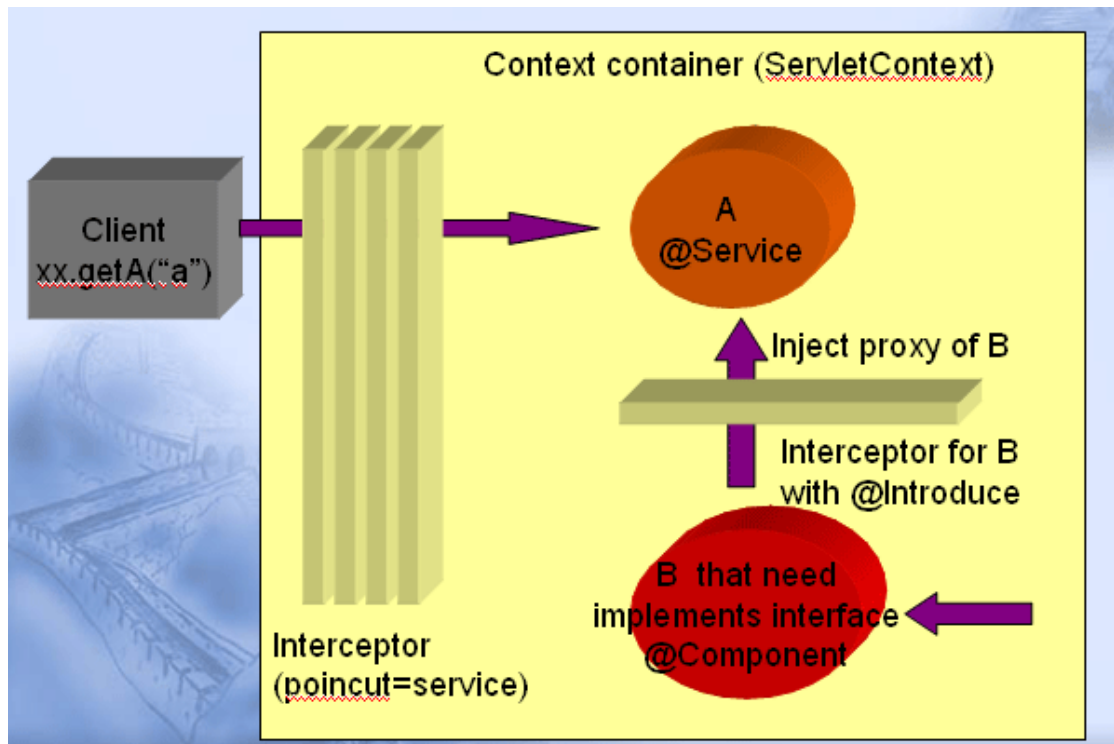
        @Send("MyModel.findName")
        public ModelMessage asyncFindName(MyModel myMod
            return new ModelMessage(myModel);
        }
    }
}
```

Inject

Introduce a interceptor

invoke MyModel --> @Introduce(message) --> MyModelDomainEvent

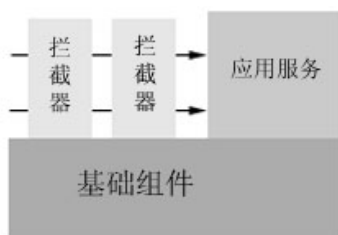
(二) @Component: 技术架构中组件可以通过构造器直接注射, 被注射的类如果有 @Introduce, 将再次引入拦截器。



## 可彻底分离的组件管理

Jdon 框架可以实现几乎所有组件可配置、可分离的管理，这主要得益于 Ioc 模式的实现，Jdon 框可以说是一个组件（JavaBeans）管理的微容器。

在 Jdon 框架中，有三种性质的组件（JavaBeans）：框架基础组件；AOP 拦截器组件和应用服务组件。三种性质的组件都是通过配置文件实现可配置、可管理的，框架应用者替换这三种性质组件的任何一个。



框架基础组件是 Jdon 框架最基本的组件，是实现框架基本功能的组件，如果框架应用者对 Jdon 框架提供的功能不满意或有意替换，可以编写自己的基础功能组件替代，从而实现框架的可彻底分离或管理。Jdon 框架功能开发基本思路是：当有新的功能加入 Jdon 框架时，总是让该功能组件实现可配置和可更换，以使得该功能代表的一类一系列其他功能有加入拓展的余地。

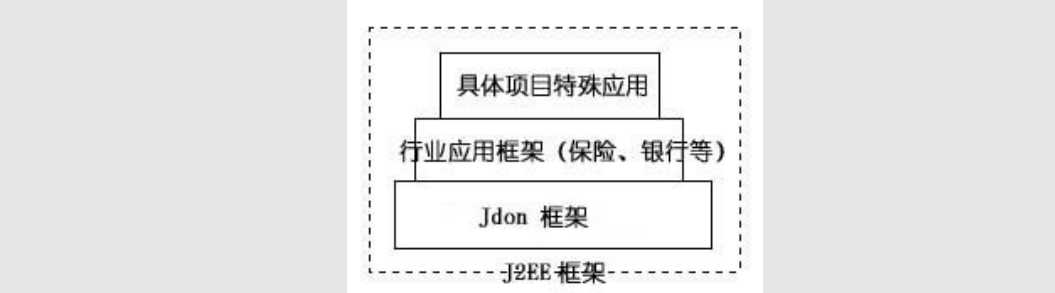
应用服务组件是框架应用者针对具体项目设计的组件，如用户管理 `AccountService`、订单服务 `OrderService` 等都属于应用服务组件。

AOP 拦截器组件主要是指一些应用相关的通用基础功能组件，如缓存组件、对象池组件等。相当于应用服务组件前的过滤器（Filter），在客户端访问应用服务组件之前，

必须首先访问的组件功能。

这三种性质组件基本涵盖了应用系统开发大部分组件，应用服务组件是应用系统相关组件，基本和数据库实现相关，明显特征是一个 DAO 类；当应用服务组件比较复杂时，我们就可以从中重整 Refactoring 出一些通用功能，这些功能可以上升为框架基础组件，也可以抽象为 AOP 拦截器组件，主要取决于它们运行时和应用服务组件的关系。当然这三种性质框架组件之间可以相互引用（以构造方法参数形式），因为它们注册在同一个微容器中。

使用 Jdon 框架，为应用系统开发者提炼行业框架提供了方便，框架应用者可以在 Jdon 框架基本功能基础上，添加很多自己行业特征的组件，从而实现了框架再生产，提高应用系统的开发效率。



在 JdonFramework 中，所有组件都是在配置文件中配置的，框架的组件是在 container.xml 和 aspect.xml 中配置，应用系统组件是在 jdonframework.xml 中配置，应用系统组件和框架内部或外部相关组件都是在应用系统启动时自动装载入 J2EE 应用服务器中，它们可以相互引用（以构造器参数引用，只要自己编写的普通 JavaBeans 属于构造器注射类型的类就可以），好似是配置在一个配置文件中一样。

因此，组件配置主要有三个配置文件：应用服务组件配置 container.xml、AOP 拦截器组件 aspect.xml 和应用服务组件配置 jdonframework.xml 另外也有一套 Annotation 来起到和这些 XML 同样作用的配置。



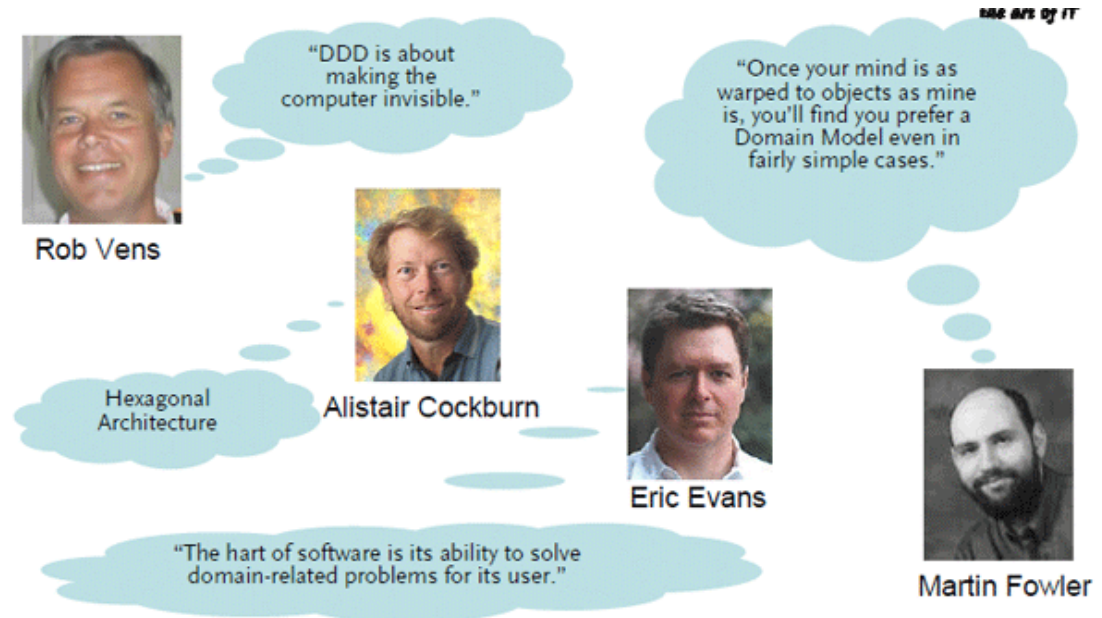
# 适合哪些人

Jdon 框架主要是面向 J2EE 程序员，对于程序员要求并不很高，只要具备以下技术背景之一就可以尝试学习使用 Jdon 框架：

- 拥有 Jsp/Servlet JavaBeans J2EE 的 Web 编程经验的程序员
- 最好有一些 Struts 感性认识和少量编程经验。

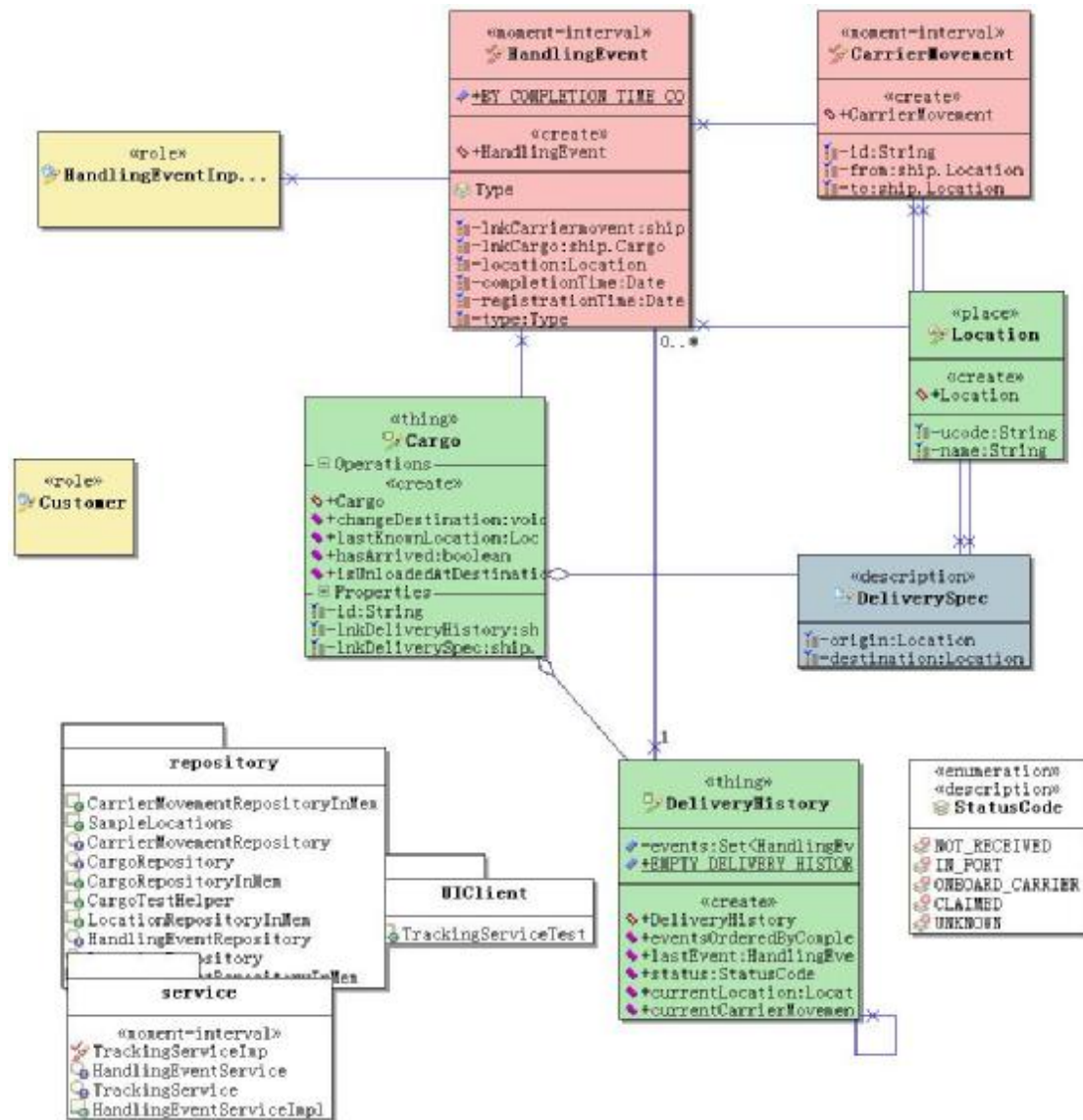
## 领域模型应用

Evans DDD 是目前软件分析设计的主要方法，看如下各位大师的解释：



Jdon.com 关于 DDD 的专题讨论: <http://www.jdon.com/jivejdon/tags/272>

当我们接到一个新项目时，使用 UML 工具，通过面向对象 DDD 的分析设计方法将其变成领域模型图，如下：



这是一个典型的 DDD 建模图,这个模型图可以直接和 Java 代码对应,比如其中 Cargo 模型的代码如下,两者是完全一一对应,可以使用 together 等建模工具直接转换, Jdon 框架的 @Model 就是针对 Cargo 这样模型, 将其运行在 Java 平台中,:

```
package ship;

/**
 * @stereotype thing
 */

public class Cargo {

    private String id;

    /**
     * @link aggregation
```

```

    */
private ship.DeliveryHistory lnkDeliveryHistory;
/**
 * @link aggregation
 */
private ship.DeliverySpec lnkDeliverySpec;

public Cargo(String trackingId, DeliverySpec deliverySpec) {
    this.id = trackingId;
    this.lnkDeliverySpec = deliverySpec;
}

public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

public void changeDestination(final Location newDestination) {
    lnkDeliverySpec.setDestination(newDestination);
}

//跟踪货物位置
public Location lastKnownLocation() {
    final HandlingEvent lastEvent = this.getLnkDeliveryHistory().lastEvent();
    if (lastEvent != null) {
        return lastEvent.getLocation();
    } else {
        return null;
    }
}

//当货物在运输抵达目的地点时
public boolean hasArrived() {
    return lnkDeliverySpec.getDestination().equals(lastKnownLocation());
}

//跟踪顾客货物的关键装卸事件
public boolean isUnloadedAtDestination() {
    for (HandlingEvent event : this.getLnkDeliveryHistory().eventsOrderedByCompletionTime())
    {
        if (HandlingEvent.Type.UNLOAD.equals(event.getType())
            && hasArrived()) {
            return true;
        }
    }
}

```



```

        }
    }
    return false;
}

public ship.DeliveryHistory getLnkDeliveryHistory() {
    return lnkDeliveryHistory;
}

public void setLnkDeliveryHistory(ship.DeliveryHistory lnkDeliveryHistory) {
    this.lnkDeliveryHistory = lnkDeliveryHistory;
}

public ship.DeliverySpec getLnkDeliverySpec() {
    return lnkDeliverySpec;
}

public void setLnkDeliverySpec(ship.DeliverySpec lnkDeliverySpec) {
    this.lnkDeliverySpec = lnkDeliverySpec;
}

}

```

当领域模型 **Cargo** 出来以后，下一步就是使用 **Jdon** 框架来将其运行起来，因为 **Jdon** 框架分为领域模型和组件技术等两个部分，**Cargo** 无疑属于 **@Model** 模型架构，我们只要给模型加上 **@Model**，就能让 **Cargo** 的对象生活在内存缓存中。

```

@Model
public class Cargo {
}

```

## 没有领域设计的坏设计

以订单为例子，如果不采取 **DDD** 设计，而是通常朴素的数据表库设计，将订单设计为订单数据表，那么带来的问题是：

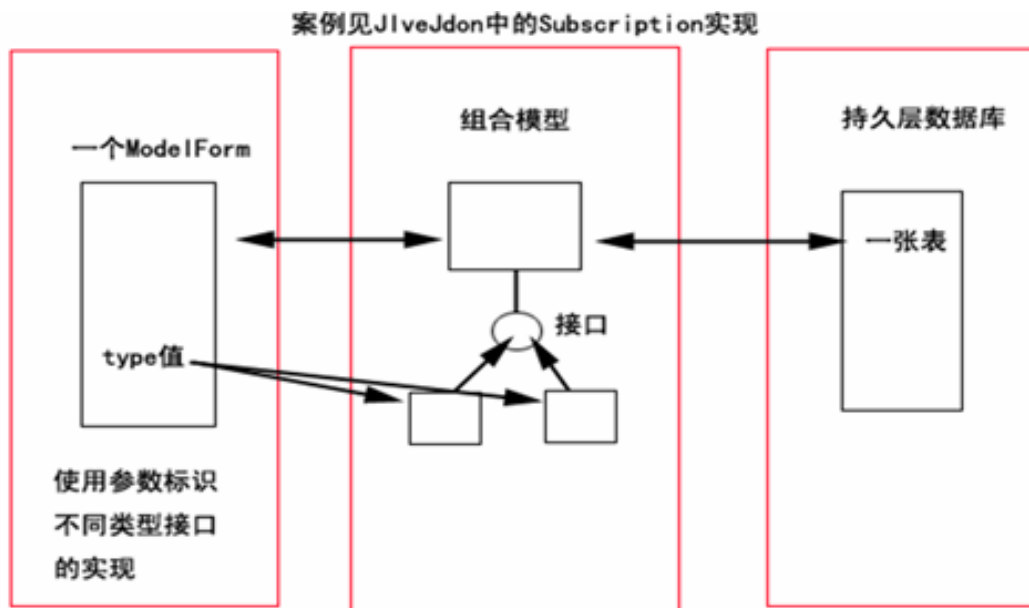
将实体的职责分离到不同限定场景，比如订单中有 **OrderItemId**, **OrderId**, **ProductId** 和 **Qty**，这是合乎逻辑的最初订单，后来有 **MinDeliveryQty** 和 **PendingQty** 字段，是和订单交货有关，这其实是两个概念，订单和订单的交货，但是我们把这些字段都混合在一个类中了。

混淆在一起的问题就是将来难以应付变化，因为实体的职责是各自变化的。

领域不是把实体看成铁板一块，一开始就把它分解到各种场景。下订单和订单交货交付是两个场景，它们应该有彼此独立的接口，由实体来实现。这就能够让实体和很多场景打交道，而彼此不影响，这就是组合模型 **composite model** 的一个关键优点。

在数据库中它们是一个，也就是说，从 **ER** 模型上看，它们是一个整体，但是从 **domain model** 领域模型角度看，它们是分离的。

下图是基于 Jdon 框架开发的 JiveJdon 中领域模型，中间的领域模型可能是继承等关系组成的对象群，或者成为聚合群 Aggregation，但是对应的数据表可能是一张表，由此可见，领域模型能够更准确反映业务需求。



## 实体模型

根据 DDD 方法，需求模型分为实体模型 值对象和领域服务三种，实际需求经常被划分为不同的对象群，如 Cargo 对象群就是 Cargo 为根对象，后面聚合了一批与其联系非常紧密的子对象如值对象等，例如轿车为根对象，而马达则是轿车这个对象群中的一个实体子对象。

在 Jdon 框架中，实体根模型通常以 @Model 标识，并且要求有一个唯一的标识主键，你可以看成和数据表的主键类似，它是用来标识这个实体模型为唯一的标志，也可以使用 Java 对象的 hashCode 来标识。

Jdon 框架是实体模型的主键标识有两个用处：

首先是用来缓存实体模型，目前缓冲器是使用 EHcache，可以无缝整合分布式云缓存 Terracotta 来进行伸缩性设计。

只要标识 @Model 的实体，在表现层 Jsp 页面再次使用时，将自动直接从缓存中获得，因为在中间业务层和表现层之间 Jdon 框架存在一个缓存拦截器 CacheInterceptor，见框架的 aspect.xml 中配置。

为了能够在业务层能够使用缓存中的模型，需要在业务层后面的持久层或 Repository 中手工加入缓存的 Annotation 标签，如下：

```

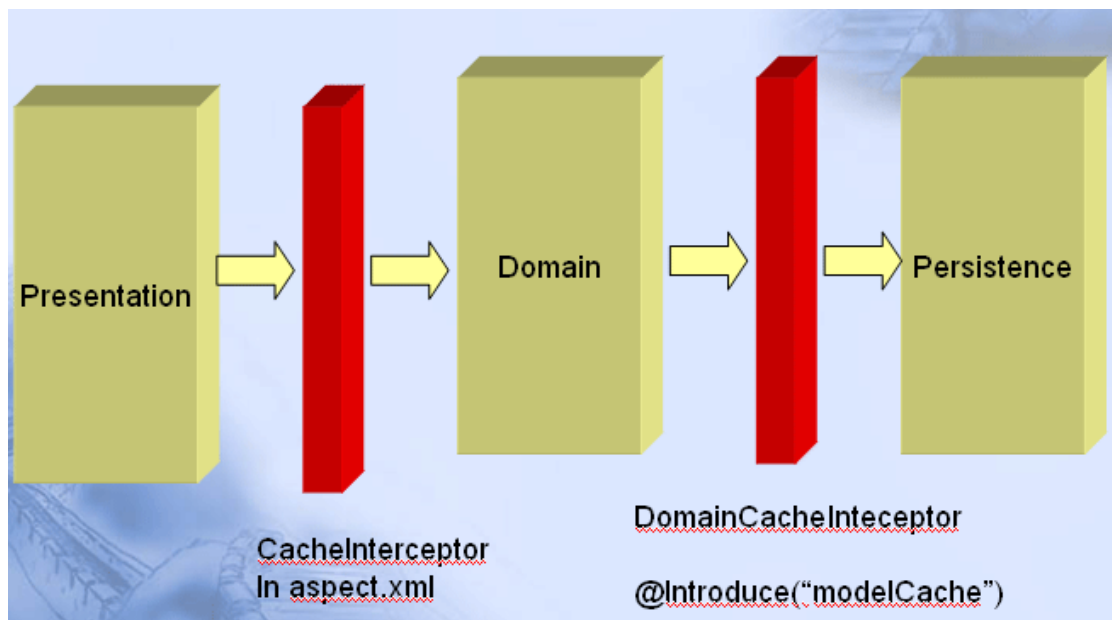
@Component("mymrepository")
@Introduce("modelCache")
public class RepositoryImp implements MyModel {

    @Around
    public MyModel getModel(Long key) {
        MyModel mym = new MyModel();
        mym.setId(key);
        return mym;
    }
}

```

Auto cache MyModel

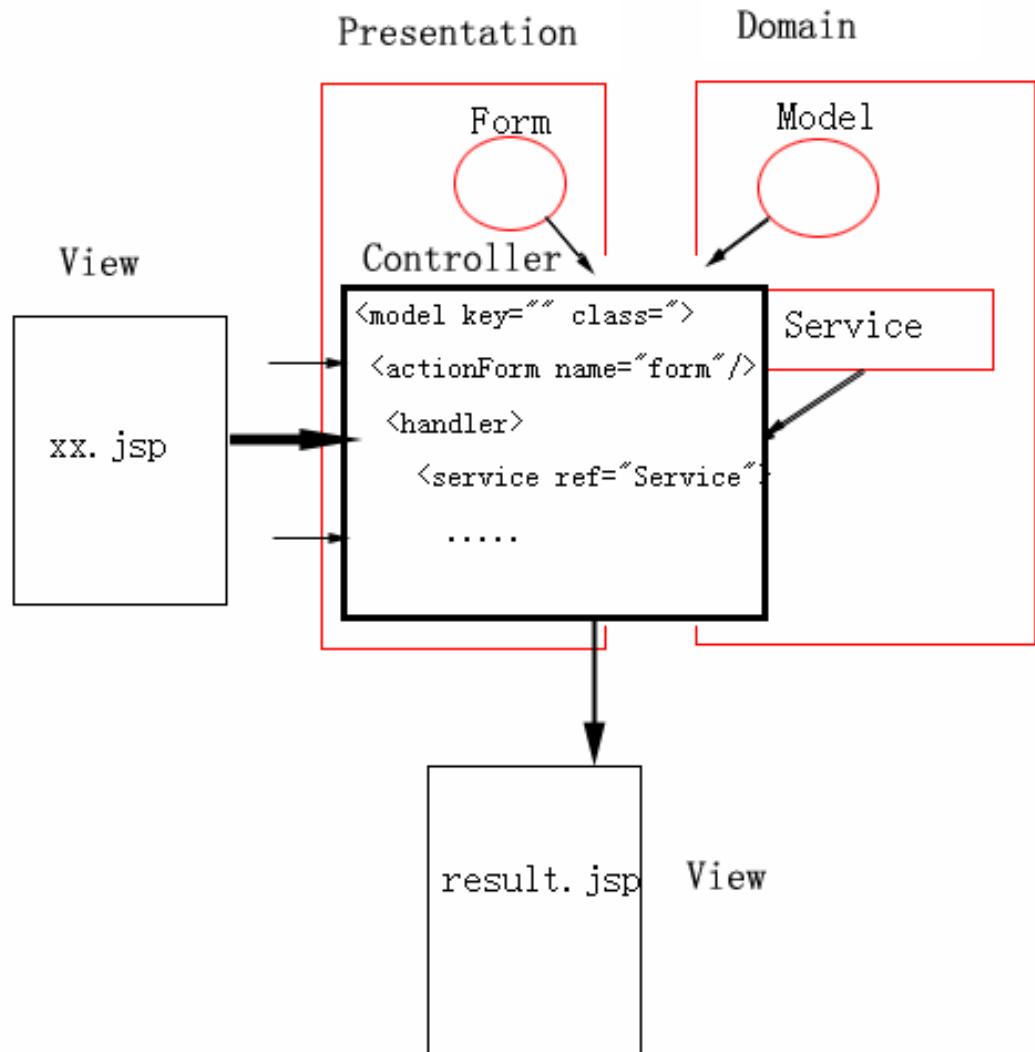
实体模型的缓存架构如下：



注意：这里可能是 Jdon 框架 6.2 的一个创新或重点，在 JF6.2 之前实际上没有对模型进行突出的支持，就象画个圈，圈子外面基本都就绪，圈子里面留白，这个圈子就是 Domain Model，原来因为考虑持久层 Hibernate 等 ORM 巨大影响力，就连 Spring 也是将 Model 委托给 Hibernate 处理，自己不做模型层，但是当 NoSQL 运动蓬勃发展，DDD 深入理解，6.2 则找到一个方式可以介入模型层，同时又不影响任一持久层框架的接入。

Jdon 框架 6.2 通过在持久层的获得模型对象方法上加注释的办法，既将模型引入了内存缓存，又实现了向这个模型注射必要的 Domain Events（见下一章）。

Jdon 框架使用实体模型的主键标识另外一个用处就是实现实体模型的表现层增删改查简化开发，见下面 [CRUD 快速开发专门章节](#)，架构图如下，图中业务层就是 Domain 层：



## Domain Events

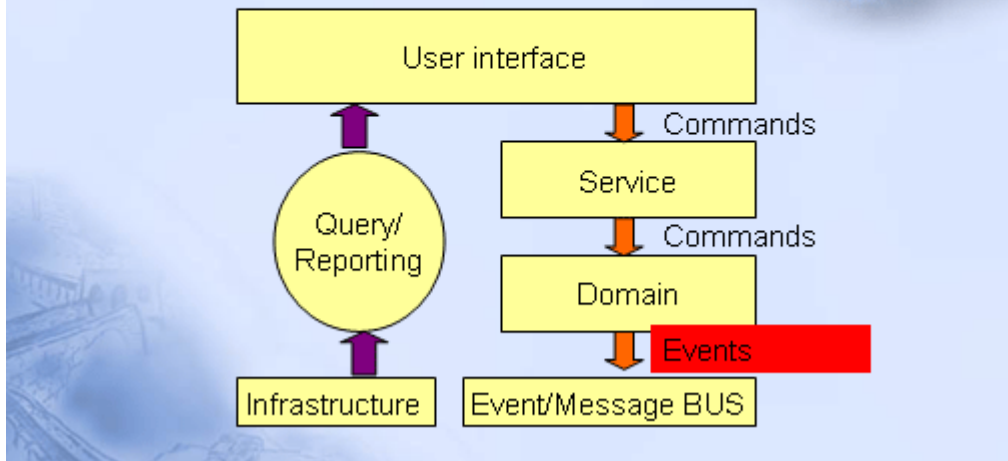
在 Evans DDD 实现过程中，经常会碰到实体和服务 **Service** 以及 **Repository** 交互过程，这个交互过程的实现是一个难点，也是容易造成失血贫血模型的主要途径。

领域模型中只有业务，没有计算机软件架构和技术。不要将和技术相关的服务和功能组件注射到实体模型中，例如数据库 **Dao** 等操作。由领域模型通过 **Domain Events** 机制指挥 **Domain** 服务和其他功能组件，包括数据库操作。

<http://jonathan-oliver.blogspot.com/search/label/DDD>

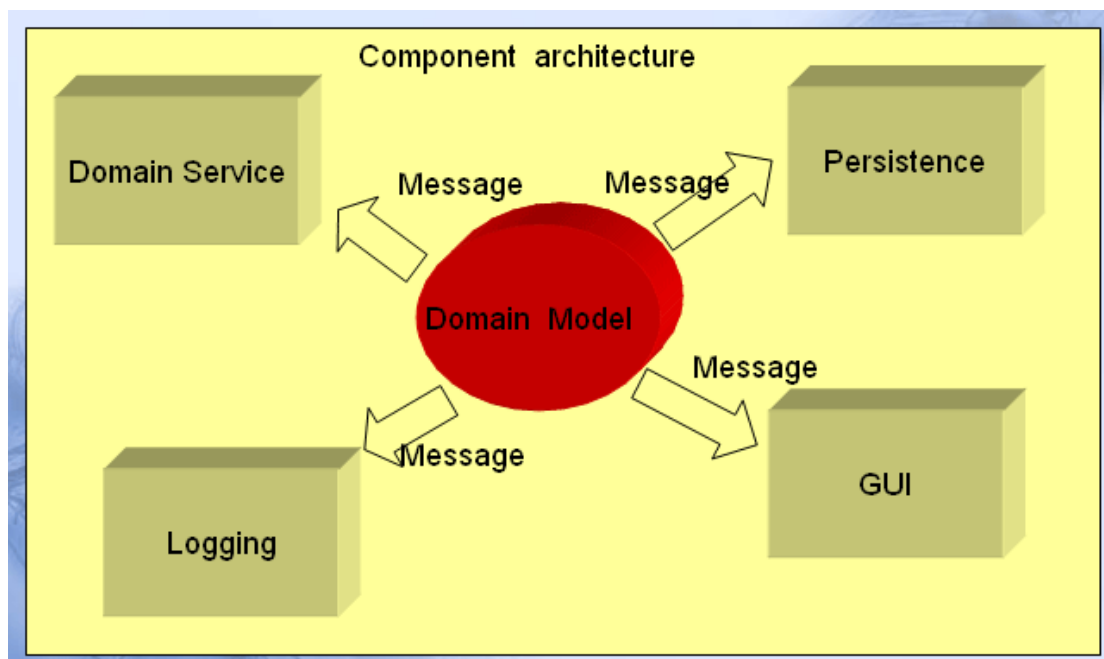
使用 DDD，常常需要[命令查询分离模式](#) CQS 架构，如下图：

# CQRS(查询命令分离) Architecture



图中 Domain 发出的 Events 就称为 Domain Events, 如果说 CQS 架构提出将 Command 命令和查询分离的模式, 那么随着 Command 命令产生 Domain Events 则提出业务模型和技术架构分离的解决方案。

JF 提供的异步观察者模式为 Domain Event 实现提供更优雅的方案。详细文章见:  
<http://www.jdon.com/jivejdon/thread/37289>



Domain Events 异步消息的使用如下, 分三步:

1. 创建模型类如 `UserModel`, 在 `UserModel` 引入自己的 `UserDomainEvents`  
@Inject  
private UserDomainEvents userDomainEvents;

2. 创建 UserDomainEvents 类
3. 创建 com.jdon.domain.message.MessageListener 实现子类

如下图所示步骤:

该案例代码可在 <http://www.jdon.com/jdonframework/SimpleJdonFrameworkTest.rar> 下载。

```
@Model
public class UserModel {

    private String userId;
    private String name;

    private UserCountValueObject ageVO;

    @Inject
    private UserDomainEvents userDomainEvents;

    public void update(UserModel userParameter) {
        this.name = userParameter.getName();

        ageVO.preloadData();
        userDomainEvents.save(this);
    }

    @Introduce("message")
    public class UserDomainEvents {

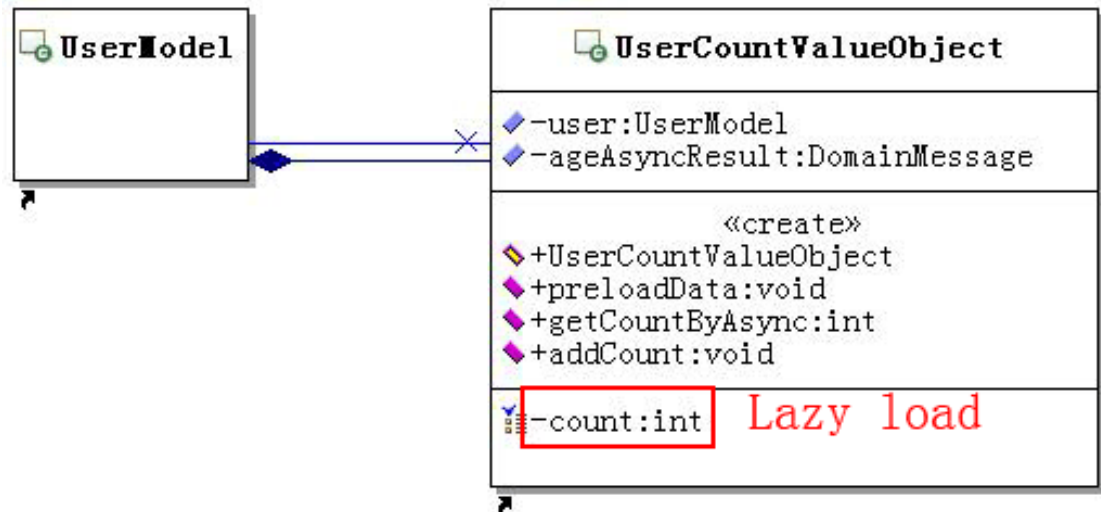
        @Send("saveUser")
        public DomainMessage save(UserModel user) {
            return new DomainMessage(user);
        }
    }

    @Component("computeCount")
    public class ComputeCountListener implements MessageListener {

        public void action(DomainMessage message) {
            UserModel user = (UserModel) message.getEvent();
            int age = userRepository.getAge(user.getUserId());
        }
    }
}
```

使用 Domain Events 可实现异步懒加载机制,对模型中任何字段值根据需从数据库加载,即用即取,这种方式有别于 Hibernate 的惰加载 lazy load 机制,更加灵活。

如下图,我们为了实现 UserCount 值对象中 count 数据的懒加载,将其封装到一个值对象中,这个值对象其他字段和方法都是为懒加载服务的。



我们看看 getCountByAsync 方法内部:

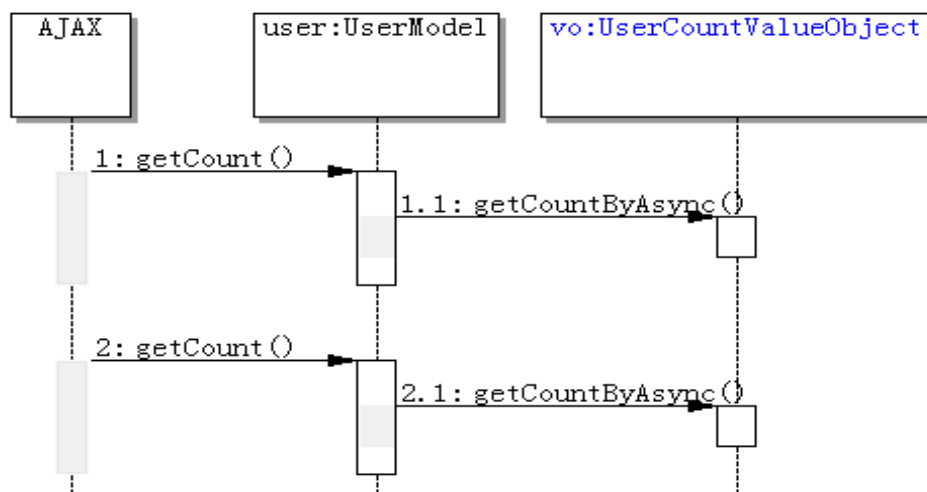
```

public int getCountByAsync() {
    if (count == -1) { // lazy load
        if (ageAsyncResult == null)
            ageAsyncResult = user.getUserDomainEvents().computeCount(user);
        else
            count = (Integer) ageAsyncResult.getEventResult();
    }
    return count;
}
  
```

Count 初始值是-1, 如果该方法第一次调用, count 必为-1, 那么 ageAsyncResult 也应该为空, 这时就激活 computeCount 这个耗时耗 CPU 的任务, 当第二次访问 getCountByAsync 时, ageAsyncResult 应该不为空, 因为耗时任务应该计算完毕, 如果还没有, 会在这里等待。一旦获得计算结果, count 就不会为-1。

UserCount 值对象随同 UserModel 生活在内存中, 因此, 这样耗时任务计算只需要一次, 以后, 基本直接从对象的字段 count 直接获得。

调用顺序图如下:



## 深入 Dmain Events

领域模型中只有业务，没有计算机软件架构和技术。

不要将和技术相关的服务和功能组件注射到实体模型中，例如数据库 Dao 等操作。容易造成多点多处耦合。由领域模型通过 Domain Events 机制指挥 Domain 服务和其他功能组件，包括数据库操作。

在 Evans DDD 实现过程中，经常会碰到实体和服务 Service 以及 Repository 交互过程，这个交互过程的实现是一个难点，也是容易造成失血贫血模型的主要途径。

因为实体的业务方法需要和服务或 Reposirtoy 打交道，如果把这个业务方法放入服务，就容易造成实体的贫血；但是如果把服务注射到实体中，也非常丑陋。这里提出一个中间处理模式：Domain Event，领域事件模式，这个模式也曾经被 MF 在文章 Domain Event(<http://martinfowler.com/eaDev/DomainEvent.html>)专门章节提到。

详细文章见：<http://www.jdon.com/jivejdon/thread/37289>  
<http://www.jdon.com/jivejdon/thread/37712>

- § 将领域模型和技术架构通过松耦合的异步消息解耦。
- § 可以避免 Hibernate 的懒加载 LazyInitializationExceptions，无需使用 Open Session in View 或 Close Session in View
- § 可以在 MessageListener 处集成入 JMS，JMS 是可伸缩性并带有分布式事务方案，可以保证持久化等事务精确完成。

## 案例

Account 中有一个计算该用户发帖总数字段 messageCount:

```
public class Account{  
  
    private int messageCount;  
  
    public int getMessageCount(){  
        return messageCount;  
    }  
  
}
```

这个 messageCount 是通过查询该用户所有发帖数查询出来的，也许你会问，为什么不将用户发帖总数设置为数据表一个字段，这样用户发帖时，更新这个字段，这样只要直接查询这个字段就可以得到 messageCount？

没有设立专门持久字段的原因如下：

1. 从模型设计角度看：messageCount 字段归属问题，messageCount 其实是一个表示 Account 和 Message 两个实体的关系关联统计字段，但是这种关联关系不属于高



聚合那种组合关系的关联，不是 Account 和 Message 的必备属性，根据 DDD 的高聚合低关联原则，能不要的关联就不要，因此放在哪里都不合适。

2.从性能可伸缩性角度看：如果我们将 messageCount 放在第三方专门关联表中，那么用户增加新帖子，除了对 Message 对应的表操作外，还需要对这个关联表操作，而且必须是事务的，事务是反伸缩性的，性能差，如果象 messageCount 各种统计字段很多，跨表事务边界延长，这就造成整体性能下降。

3.当然，如果将 messageCount 硬是作为 Account 表字段，由于整个软件的业务操作都是 Account 操作的，是不是将其他业务统计如 threadCount 等等都放到 Account 表中呢？这会造成 Account 表变大，最终也会影响性能。

那么 messageCount 每次都通过查询 Message 对应表中用户所有发帖数获得，这也会导致性能差，表中数据越多，这种查询就越费 CPU。

使用缓存，因为 Account 作为模型被缓存，那么其 messageCount 值将只有第一次创建 Account 执行查询，以后就通过缓存中 Account 可直接获得。

所以，根据 DDD，在 AccountRepository 或 AccountFactory 实现数据表和实体 Account 的转换，Account 中的值都是在这个类中通过查询数据表获得的。

当访问量增加时，这里又存在一个性能问题，虽然一个 Account 创建时，messageCount 查询耗时可能觉察不出，但是如果是几十几百个 Account 第一次创建，总体性能损耗也是比较大的，鉴于我们对可伸缩性无尽的追求，这里还是有提升余地。

从设计角度看，由于 messageCount 不是 Account 的必备字段，因此，不是每次创建 Account 时都要实现 messageCount 的赋值，可采取即用即查方式。所以，我们需要下面设计思路：

```
public class Account{

    private int messageCount = -1;

    public int getMessageCount(){
        if(messageCount == -1)
            //第一次使用时即时查询数据表
        return messageCount;
    }

}
```

怎么实现这个功能呢？使用 Hibernate 的懒加载？使用 Lazy load 需要激活 Open Session In View，否则如果 Session 关闭了，这时客户端需要访问 messageCount，就会抛 lazy Exception 错误，但是 OSIV 只能在一个请求响应范围打开，messageCount 访问可能不是在这次请求中访问，有可能在后面请求或其他用户请求访问，所以，这个懒加载必须是跨 Session，是整个应用级别的。

实际上，只要 Account 保存在缓存中，对象和它的字段能够跨整个应用级别，这时，只要在 messageCount 被访问即时查询数据表，就能实现我们目标，其实如此简单问题，因为考虑 Hibernate 等 ORM 框架特点反而变得复杂，这就是 DDD 一直反对的技术框架应该为业务设计服务，而不能成为束缚和障碍，这也是一山不容二虎的一

个原因。

这带来一个问题，如何在让 Account 这个实体对象中直接查询数据库呢？是不是直接将 AccountRepository 或 AccountDao 注射到 Account 这个实体呢？由于 AccountDao 等属于技术架构部分，和业务 Account 没有关系，只不过是支撑业务运行的环境，如果将这么多计算机技术都注射到业务模型中，弄脏了业务模型，使得业务模型必须依赖特定的技术环境，这实际上就不是 POJO 了，POJO 定义是不依赖任何技术框架或环境。

POJO 是 Martin Fowler 提出的，为了找到解决方式，我们还是需要从他老人家方案中找到答案，模型事件以及 Event 模式也是他老人家肯定的，这里 Account 模型只需要向技术环境发出一个查询 Event 指令也许就可以。

那么，我们就引入一个 Domain Events 对象吧，以后所有与技术环境的指令交互都通过它来实现，具体实现中，由于异步 Message 是目前我们已知架构中最松耦合的一种方案，所以，我们将异步 Message 整合 Domain Events 实现，应该是目前我们知识水平能够想得到的最好方式之一，当然不排除以后有更好方式，目前 JdonFramework 6.2 已经整合了 Domain Events + 异步消息机制，我们就可以直接来使用。

这样，Account 的 messageCount 即用即查就可以使用 Domain Events + 异步消息实现：

```
public int getMessageCount(){
    if (messageCount == -1) {
        if (messageCountAsyncResult == null) {
            //向技术环境发出查询获得 messageCount 值的命令，
            //这个命令是在另外新线程实现，因此结果不一定立即返回

            messageCountAsyncResult =

                domainEvents.computeAccountMessageCount(account.getUserIdLong());
        } else {
            //当客户端再次调用本方法时，可以获得查询结果，
            //如果查询过程很慢，还是没有完成，会在这里堵塞等待，但概率很小
            messageCount = (Integer) messageCountAsyncResult.getEventResult();
        }
    }
}
```

messageCount 最后获得，需要通过两次调用 getMessageCount 方法，第一次是激活异步查询，第二次是获得查询结果，在 B/S 架构中，一般第二次查询是由浏览器再次发出请求，这浏览器服务器一来一回的时间，异步查询一般基本都已经完成，这就是充分利用 B/S 架构的时间差，实现高效率的并行计算。

所以，并不是一味死用同步就能提高性能，可伸缩性不一定是单点高性能，而是指整个系统的高效率，利用系统之间传送时间差，实现并行计算也是一种架构思路。这种思考思路在实际生活中也经常会发生。

最后，关于 messageCount 还有一些有趣结尾，如果浏览器不再发第二次请求，那么浏览器显示 Account 的 messageCount 就是-1，我们可以做成不显示，也就看不

到 Account 的发帖总数，如果你的业务可以容忍这个情况，比如目前这个论坛就可以容忍这种情况存在，Account 的 messageCount 第一次查询会看不到，以后每次查询就会出现，因为 Account 一直在缓存内存中。

如果你的业务不能容忍，那么就在浏览器中使用 AJAX 再次发出对 getMessageCount 的二次查询，那么用户就会每次

都会看到用户的发帖总数，JiveJdon 这个论坛的标签关注人数就是采取这个技术实现的。这样浏览器异步和服务器端异步完美结合在一起，整个系统向异步高可伸缩性迈进一大步。

更进一步，有了 messageCount 异步查询，如何更新呢？当用户发帖时，直接对内存缓存中 Account 更新加一就可以，这样，模型操作和数据表操作在 DDD + 异步架构中完全分离了，数据表只起到存储作用(messageCount 甚至没有专门的存储数据表字段)，这和流行的 NoSQL 架构是同一个思路。

由于针对 messageCount 有一些专门操作，我们就不能直接在 Account 中实现这些操作，可以使用一个专门值对象实现。如下::

```
public class AccountMessageVO {

    private int messageCount = -1;

    private DomainMessage messageCountAsyncResult;

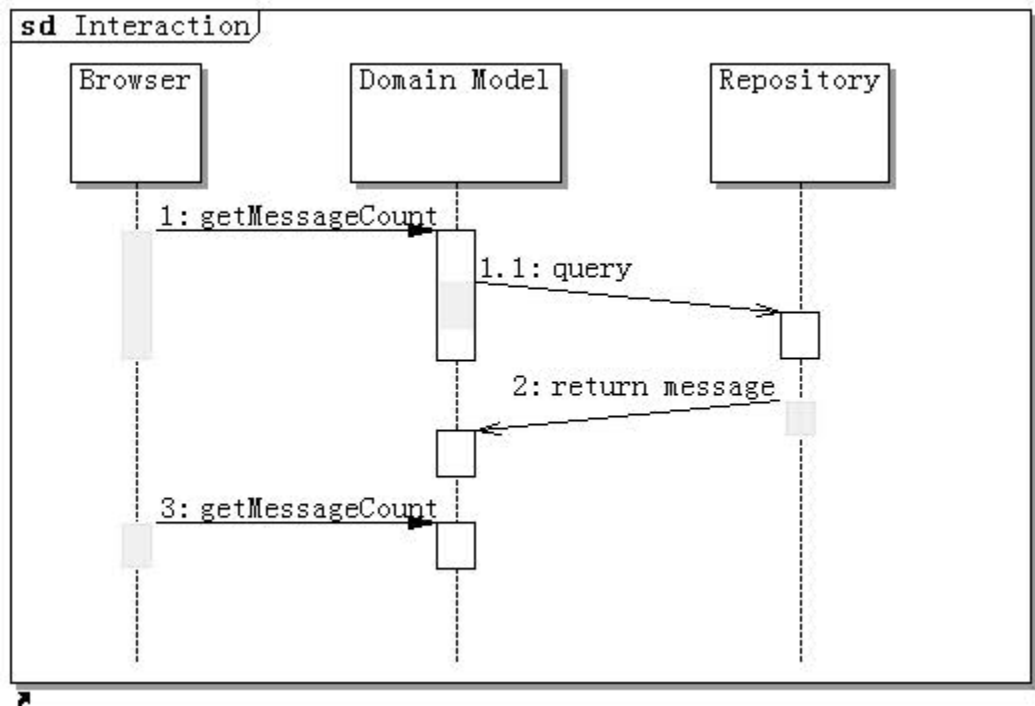
    private Account account;

    public AccountMessageVO(Account account) {
        super();
        this.account = account;
    }

    public int getMessageCount(DomainEvents domainEvents) {
        if (messageCount == -1) {
            if (messageCountAsyncResult == null) {
                messageCountAsyncResult =
                    domainEvents.computeAccountMessageCount(account.getUserIdLong());
            } else {
                messageCount = (Integer) messageCountAsyncResult.getEventResult();
            }
        }
        return messageCount;
    }

    public void update(int count) {
        if (messageCount != -1) {
            messageCount = messageCount + count;
        }
    }
}
```

调用顺序图如下：



## 即用即加载

比如图片，只有显示图片时才会需要图片的二进制数据，这个数据比较大，所以，一般从持久层加载图片时，只加载图片其他核心文字信息，如图片 ID，名称等，当需要显示时，再加载二进制数据输出真正图片。

```
public byte[] getData() {
    byte[] bytes = super.getData();
    if (bytes != null)
        return bytes;
    preloadData();
    bytes = (byte[]) imgDataAsyncResult.getEventResult();
    this.setData(bytes);
    return bytes;
}

//预加载 可在 JSP 即将显示图片之前发出事件激活该方法
public void preloadData() {
    if (imgDataAsyncResult == null && domainEvents != null)
        imgDataAsyncResult = this.domainEvents.loadUploadEntity(this);
}
```

传统意义上，懒加载和[异步](#)都是好像不被人接受的，会带来比较差的性能，高延迟性，属于边缘技术，这其实是被误导了：[并发策略可以解决延迟](#)

懒加载和[异步](#)代表的并发策略实际是一种潮流趋势，特别是作为并行计算语言 Scala 和 erlang 的新亮点：[函数式编程 functional programming 的特点](#)

而最新发布 [JiveJdon](#)3.9 使用传统 Java，基于 Jdon 框架 6.2 实现了领域模型层的懒加载和[异步](#)，可以完全克服 [Hibernate](#) 等 ORM 框架带来的 Lazy 懒加载错误问题。

## 模型注射

领域模型中不应该含有技术组件架构的元素，这是一种理想目标境界，但是在实际操作过程中，可能还是需要一些技术组件功能需要引入到模型中，Jdon 框架也提供了这种模型注射功能，一般可以将 DomainEvents 对象注射进入当前模型，这样，可以在当前模型的方法中，直接使用 DomainEvents 中的消息发送方法，反之，如果将当前模型和 DomainEvents 合二为一，也是可以的，但是本模型中方法就不能调用标注 @Send 的方法，这个方法只能供缓存以外的访问者调用了。

```

@Model
public class MyModel {

    private Long id;
    private String name;

    @Inject
    private MyModelDomainEvent myModelDomainEvent;

    @Inject
    private MyModelService myModelServiceCommand;
}

```

Inject

```

@Introduce("message")
public class MyModelDomainEvent {

    @Send("MyModel.findName")
    public ModelMessage asyncFindName(MyModel myModel) {
        return new ModelMessage(myModel);
    }
}

```

Introduce a interceptor

MyModel  $\xrightarrow{\text{invoke}}$  @Introduce(message)  $\xrightarrow{\text{invoke}}$  MyModelDomainEvent

上图中，不但可以注射 MyModelDomainEvent 到模型 MyModel 中，偶尔也可以将一个领域服务 MyModelService 注射到 MyModel 中，这种方式相比 DomainEvents 的异步，这是同步方式。

## 组件应用

本章节讲述 Jdon 框架的基本配置和一些基本设计，是使用 Jdon 框架之前需要了解的章节，通过这些基本概念说明，可以让程序员对 Jdon 框架有一个大体概念上的轮廓。

本章节对于初学者不是必读，其中很多用法可在后面案例中学习，因此，本章节也可以直接跳过去，进入下一个章节阅读。

## 应用服务配置

jdonframework.xml 是应用服务组件配置文件，文件名可自己自由定义，

jdframework.xml 中主要是定义 Model（模型）和 Service（服务）两大要素。

jdframework.xml 最新定义由 <http://www.jdon.com/jdonframework.dtd> 规定。

<models>段落是定义应用系统的建模，一个应用系统有哪些详细具体的模型，可由 Domain Model 分析设计而来。<models>中的详细配置说明可见 数据模型增、删、改、查章节。

<services>段落是定义服务组件的配置，目前有两种主要服务组件配置：EJB 和 POJO。

EJB 服务组件配置如下：

```
<ejbService name="newsManager">
    <jndi name="NewsManager" />
    <ejbLocalObject class="news.ejb.NewsManagerLocal"/>
</ejbService>
```

每个 ejbService 组件有一个全局唯一的名字，如 newsManager，有两个必须子定义：该 EJB 的 JNDI 名称和其 Local 或 remote 接口类。

POJO 服务组件配置如下：

```
<pojoService name="userJdbcDao" class="news.container.UserJdbcDao">
    <constructor value="java:/NewsDS"/>
</pojoService>
```

POJO 服务也必须有一个全局唯一名称，如 userJdbcDao，以及它的 class 类定义。如果该 POJO 构造器有字符串变量，可在这里定义其变量的值，目前 Jdon 框架只支持构造器字符串变量注射。

如果该 POJO 服务需要引用其它服务，例如 UserPrincipalImp 类的构造器如下：

```
public UserPrincipalImp(UserDao userDao){

    this.userDao = userDao;

} .....
```

UserPrincipalImp 构造器需要引用 UserDao 子类实现，只需在 jdframework.xml 中同时配置这两个服务组件即可，Jdon 框架会自动配置它们之间的关系：

```
<pojoService name="userJdbcDao" class="news.container.UserJdbcDao">
    <constructor value="java:/NewsDS"/>
</pojoService>
<pojoService name="userPrincipal" class="news.container.UserPrincipalImp"/>
```

上面配置中 news.container.UserJdbcDao 是接口 UserDao 的子类实现，这样，直接通过 userPrincipal 这个名称可获得 UserPrincipalImp 的实例。

6.0 版本以后可使用@Service(“myName”)替代以上 XML 配置。

## 基础组件配置说明

container.xml 是 Jdon 框架基础组件配置文件，container.xml 中包含的组件将由 Jdon 框架在启动时向微容器（PicoContainer）中注册，至于这些组件之间的依赖关系由微容器解决，称为 Ioc 模式。

container.xml 内容主要由每行配置组成，每行格式如下：

```
<component name="组件名称" class="POJO 类名称" />
```

如

```
<component name="modelHandler" class="com.jdon.model.handler.XmlModelHandler" />
```

代表组件 `com.jdon.model.handler.XmlModelHandler`，其名称为 `modelHandler`，如果需要在程序中调用 `XmlModelHandle` 实例，只需要以 `modelHandler` 为名称从微容器中获取即可。

组件配置也可以带有参数，例如下行：

```
<component name="cache" class="com.jdon.controller.cache.LRUCache" >
    <constructor value="cache.xml"/>
</component>
```

而 `LRUCache` 的类代码如下：

```
public class LRUCache implements Cache {
    public LRUCache(String configFileName) {
        PropsUtil propsUtil = new PropsUtil(configFileName);
        cache = new UtilCache(propsUtil);
    }
    .....
}
```

这样 `LRUCache` 中的 `configFileName` 值就是 `cache.xml`，在 `cache.xml` 中定义了有关缓存的一些设置参数。目前 `Jdon` 框架只支持构造器是纯字符串型，可多个字符串变量，但不能字类型和其它类型混淆在一起作为一个构造器的构造参数。如果需要多个类型作为构造参数，可新建一个包含字符串配置的类，这个类就可和其它类型一起作为一个构造器的构造参数了。

一般在 `container.xml` 中的组件是框架基本功能的类，不涉及到具体应用系统。

6.0 版本以后可使用 `@Component("myName")` 替代以上 XML 配置。

## 拦截器组件配置说明

本篇为高级应用，初学者可以以后再深入研究。

`aspect.xml` 是关于拦截器组件配置，有两个方面：`advice`（拦截器 `Interceptor`）和 `pointcut`（切入点）两个方面配置，有关 AOP 的基本概念可见：<http://www.jdon.com/AOPdesign/aspectJ.htm>。

`Jdon AOP` 的设计目前功能比较简单，不包括标准 AOP 中的 `Mixin` 和 `Introduction` 等功能；`Pointcut` 不是针对每个 `class` 和方法，而是针对一组 `class`（如 `POJO` 组或 `EJB` 组），拦截粒度最粗，与许多复杂完整的 AOP 框架（如 `AspectJ`、`Spring`）不同的是：`Jdon AOP` 在粒度方面是最粗的，`AspectJ` 最细，`Spring` 中等，如果你需要粒度细腻的 AOP 功能，还是推荐使用 `Spring` 或 `AspectJ`。目前这样设计是主要有两个原因：

每个类在运行时刻都实现动态拦截，在性能上有所损失，这如同职责链模式缺点一样。

在实际应用中，可以通过代理模式 `Proxy`、装饰模式 `Decorator` 实现一些细腻度拦截，结合容器的 `Ioc` 特性，这两个代理模式使用起来非常方便，运行性能有一定提高。

`Jdon AOP` 主要针对拦截器 `Interceptor` 设计，它可以为所有 `jdonframework.xml` 中定义的 `Service` 提供拦截器；所有的拦截器被放在一个拦截器链 `InterceptorsChain` 中。

`Jdon AOP` 并没有为每个目标实例都提供拦截器配置的功能，在 `JdonAOP` 中，目标对象是以组为单位，而非每个实例，类似 `Cache/Pool` 等这些通用拦截器都是服务于所有目标对象。

`JdonAOP` 拦截器目标对象组有三种：全部目标服务；`EJB` 服务；`POJO` 服务（`EJB` 服务和 `POJO` 服务是在 `JdonFramework.xml` 中定义的 `ejbService` 和 `pojoService`）。从而也



决定了 Pointcut 非常简单。以下是 aspect.xml 中的配置：

```
<interceptor name="cacheInterceptor"
    class="com.jdon.aop.interceptor.CacheInterceptor"
    pointcut="services" />
```

其中 pointcut 有三种配置可选：services ; pojoServices 和 ejbServices

拦截器配置也可以如组件配置一样，带有 constructor 参数，以便指定有关拦截器设置的配置文件名。

拦截器的加入不只是通过配置自己的 aspect.xml 可以加入，也可以通过程序实现，调用 WebAppUtil 的 addInterceptor 方法即可，该方法只要执行一次即可。

## 如何实现自己的拦截器？

本篇为高级应用，初学者可以以后再深入研究。

以对象池拦截器 PoolInterceptor 为例，对象池是使用 Aapche Commons Pool 开源产品，对象池主要是为了提高 POJO Service 的运行性能，在没有对象池的情形下，POJO Service 每次被访问时，要产生一个新的实例，如果并发访问用户量很大，JVM 将会频繁创建和销毁大量对象实例，这无疑是耗费性能的。

使用对象池则可以重复使用一个先前以前生成的 POJO Service 实例，这也是 Flyweight 模式一个应用。

对象池如何加入到 Jdon 框架中呢？有两种方式：1.替代原来的 POJO Service 实例创建方式，属于 PojoServiceFactory 实现（Spring 中 TargetSource 实现）；2.通过拦截器，拦截在原来的 PojoServiceFactory 实现之前发生作用，同时屏蔽了原来的 PojoServiceFactory 实现。Jdon 框架采取的这一种方式。

首先，为拦截器准备好基础组件。对象池拦截器有两个：对象池 com.jdon.controller.pool.CommonsPoolAdapter 和对象池工厂 com.jdon.controller.pool.CommonsPoolFactory，这两个实现是安装 Apache Pool 要求实现的。

第二步，需要确定拦截器的 Pointcut 范围。前面已经说明，在 Jdon 框架中有三个 Pointcut 范围：所有服务、所有 EJB 服务和所有 POJO 服务，这种划分目标的粒度很粗糙，我们有时希望为一些服务群指定一个统一的拦截器，例如，我们不想为所有 POJO 服务提供对象池，想为指定的一些目标服务（如访问量大，且没有状态需要保存的）提供对象池，那么如何实现呢？这实际是如何自由划分我们自己的目标群（或单个目标实例）的问题

我们只要制作一个空接口 Poolable，其中无任何方法，只要将我们需要对象池的目标类实现这个接口即可，例如 com.jdon.security.web.AbstractUserPrincipal 多继承一个接口 Poolable，那么 AbstractUserPrincipal 所有的子类都将被赋予对象池功能，所有子类实例获得是从对象池中借入，然后自动返回。

5.6 版本以后，增加 Annotation @Poolable 替代接口 Poolable。

这种通过编程而不是配置实现的 Pointcut 可灵活实现单个目标实例拦截或一组目标实例拦截，可由程序员自由指定划分，非常灵活，节省了琐碎的配置，至于 Pointcut 详细到类方法适配，可在拦截器中代码指定，如缓存 com.jdon.aop.interceptor.CacheInterceptor 只拦截目标服务类中的 getXXXX 方法，并且该方法的返回结果类型属于 Model 子类，为了提高性能，CacheInterceptor 将符合条件的所有方法在第一次检查合格后缓存起来，这样，下次无需再次检查，省却每次检查。

第三步，确定拦截器在整个拦截器链条中的位置。这要根据不同拦截器功能决定，对象池拦截器由于是决定目标服务实例产生方式，因此，它应该最后终点，也就是在拦截器链中最后一个执行，aspect.xml 中配置拦截器是有先后的：

```
<interceptor name="cacheInterceptor"
            class="com.jdon.aop.interceptor.CacheInterceptor" pointcut="services" />

<interceptor name="poolInterceptor"
            class="com.jdon.aop.interceptor.PoolInterceptor" pointcut="pojoServices" />
```

拦截器链中排序是根据 `Interceptor` 的 `name` 值排序的，`cacheInterceptor` 第一个字母是 `c`，而 `poolInterceptor` 第一个字母是 `p`，按照字母排列顺序，`cacheIntercepotr` 排在 `poolInterceptor` 之前，在运行中 `cacheInterceptor` 首先运行，在以后增加新拦截器时，要注意将 `poolInterceptor` 排在最后一个，`name` 值是可以任意指定的，如为了使 `PoolInterceptor` 排在最后一个，可命名为 `zpoolInterceptor`，前面带一个 `z` 字母。

第四步，确定拦截器激活行为是在拦截点之前还是之后，或者前后兼顾，这就是 `advice` 的三种性质：`Before`、`After` 或 `Around`，这分别是针对具体拦截点 `jointcut` 位置而言。

虽然 `Jdon` 框架没有象 `Spring` 那样提供具体的 `BeforeAdvice` 和 `AfterReturningAdvice` 等接口，其实这些都可以有程序员自己直接实现。

`Jdon` 框架的拦截器和 `Spring` 等遵循 `aopalliance` 的 `AOP` 框架继承同一个接口 `MethodInterceptor`，也就是说，一个拦截器在这些不同 `AOP` 框架之间可以通用，具体实现方式不同而已。

例如，一个 `AroundAdvice` 实现如下代码，它们都只需要完成 `invoke` 方法内尔：

```
public class AroundAdvice    implements MethodInterceptor
{
    public Object invoke( MethodInvocation invocation)    throws Throwable
    {
        System.out.println("Hello world! (by " + this.getClass().getName() + ")");
        invocation.proceed();
        System.out.println("Goodbye! (by " + this.getClass().getName() + ")");
        return null;
    }
}
```

`beforeAdvice` 实现代码如下：

```
public class BeforeAdvice    implements MethodInterceptor
{
    public Object invoke( MethodInvocation invocation)    throws Throwable
    {
        System.out.println("Hello world! (by " + this.getClass().getName() + ")");
        invocation.proceed();
    }
}
```

由此可以注意到，`invocation.proceed()` 类似一个 `jointcut` 点，这个方法类似 400 米接力比赛中的传接力棒，将接力棒传到下一个拦截器中，非常类似与 `servletFilter` 中的 `chain.doFilter(request, response)`；拦截器一些更详细说明可参考 `Spring` 相关文档，如下面网址：<http://www.onjava.com/pub/a/onjava/2004/10/20/springaop2.html>，和 `Jdon` 拦截器原理基本一致。

考察对象池拦截器功能，它实际是一个 `around advice`，在 `jointcut` 之前需要从对象

池借用一个目标服务实例，然后需要返回对象池。`com.jdon.aop.interceptor.PoolInterceptor` 主要核心代码如下：

```
Pool pool = commonsPoolFactory.getPool(); //获得对象池
Object poa = null;
Object result = null;
try {
    poa = pool.acquirePoolable(); //借用一个服务对象
    Debug.logVerbose(" borrow a object:" + targetMetaDef.getClassName()
        + " from pool", module);
    //set the object that borrowed from pool to MethodInvocation
    //so later other Interceptors or MethodInvocation can use it!
    proxyMethodInvocation.setThis(poa); //放入 invocation， 以便供使用
    result = invocation.proceed();
} catch (Exception ex) {
    Debug.logError(ex, module);
} finally {
    if (poa != null) {
        pool.releasePoolable(poa); //将服务对象归还对象池
        Debug.logVerbose(" realease a object:" + targetMetaDef.getClassName()
            + " to pool", module);
    }
}
return result;
```

经过上述四步考虑，我们基本可以在 Jdon 框架动态 plugin 自己的拦截器，关于对象池拦截器有一点需要说明的，首先 EJB 服务因为有 EJB 容器提供对象池功能，因此不需要对象池了，在 POJO 服务中，如果你的 POJO 服务设计成有状态的，或者你想让其成为单例，就不能使用对象池，只要你这个 POJO 服务类不继承 `Poolable` 或没有 `@Poolable` 注释，它的获得是通过组件实例方式获得，参考后面“组件实例和服务实例”章节。

自己实现的拦截器是在 `myaspect.xml` 中配置，至于如何将 `myaspect.xml` 告诉 Jdon 框架，可见[启动自定义框架组件](#)一节。

6.2 版本以后可使用 `@Interceptor("myName")` 替代以上 XML 配置。

## 组件替代

我们前面说过，Jdon 框架的一个最大特点是实现了对象的可替代性，Jdon 框架应用系统组件和框架本身组件都是可以更换的，这样，程序员可以根据自己开发的具体特点，通过编制自己的组件，丰富 Jdon 框架的一些功能。

那么自己编制的组件如何放入 Jdon 框架，让 Jdon 框架识别呢？

通过前面介绍已经可能知道，通过修改 `container.xml` 或 `aspect.xml` 这样的配置文件即可。

一般情况下，`container.xml` 是包含在 Jdon 框架的压缩包 `jdonFramework.jar` 中，不便于修改，我们可以用 `winrar` 等解压工具将 `jdonFramework.jar` 中 `META-INF` 目录下的 `container.xml` 或 `aspect.xml` 解压出来，解压出来的 `container.xml` 也必须放置在系统的 `classpath` 中，当然，你修改 `container.xml` 以后，再将修改后的 `container.xml` 通过 `winrar`

工具再覆盖 jdonFramework.jar 中原来的 container.xml 也可以。

具体修改办法，以 container.xml 下面一句为例子：

```
<component name="sessionContextSetup" class="com.jdon.security.web.HttpRequestUserSetup" />
```

其中 com.jdon.security.web.HttpRequestUserSetup 是 SessionContextSetup 接口的子类，如果我们要实现自己的 SessionContextSetup 子类，例如为 MyHttpRequestUserSetup，那么编制好 MyHttpRequestUserSetup 后，将配置中替代为 MyHttpRequestUserSetup 即可：

```
<component name="sessionContextSetup" class="com.mylib.MyHttpRequestUserSetup" />
```

注意，我们自己子类 MyHttpRequestUserSetup 也必须放置在系统的 classpath，最简单的办法是：将 MyHttpRequestUserSetup 等类也打包成.jar 包，和 jdonFramework.jar 一起部署。

如果你需要增加自己的组件，那么只要定义 mycontainer.xml 和 myaspect.xml 即可，然后将这两个配置通过[启动配置](#)告诉 Jdon 框架。

## 组件实例和服务实例

在 Jdon 框架中， POJO 实例分为两种性质：组件实例和服务实例。

组件实例是指那些注册在容器中的普通 Javabeans，它们是单例的，也就是你每次获得的组件实例都是同一个实例，也就是说是单态的。

组件 POJO 获得方法是通过 WebAppUtil 的 getComponentInstance 方法，例如在 container.xml 中有如下组件定义：

```
<component name="modelManager" class="com.jdon.model.ModelManagerImp" />
```

在程序中如果要获得 ModelManagerImp 组件实例的方法是：

```
ModelManager modelManager =  
    (ModelManager)WebAppUtil.getComponentInstance("modelManager", sc);
```

组件实例获得的原理实际是直接在微容器中寻找以前注册过的那些 POJO，相当于直接从 XML 配置中直接读取组件实例配置。

服务实例是指在 jdonframework.xml 中定义的 ejbService 和 pojoService，当然它们也可以组件实例方式获得，但是如果以组件实例方式获得，AOP 功能将失效；而以服务实例方式获得的话，在 aspect.xml 中定义的拦截器功能将激活。

EJB 服务一定是通过服务实例方式获得，只有普通 JavaBeans (POJO) 才可能有这两种方式。

因此，除非特殊需要，一般推荐在应用系统中，通过获得服务实例方式来获得 jdonframework.xml 中定义的服务实例。

如在 jdonframework.xml 中有如下服务组件定：

```
<pojoService name="userJdbcDao" class="news.container.UserJdbcDao">  
    <constructor value="java:/NewsDS"/>  
</pojoService>
```

获得服务实例的方法代码如下：

```
UserDao ud = (UserDao)WebAppUtil.getService("userJdbcDao", request);
```

UserDao 是 UserJdbcDao 的接口。注意，这里必须 getService 结果必须下塑为接口类型，不能是抽象类或普通类，这也是与 getComponentInstance 不同所在。

如果你在 aspect.xml 将.pojoServices 都配置以对象池 Pool 拦截器，那么上面代码将是从小对象池中获取一个已经事先生成的实例。

## 如何获得一个 POJO 服务实例？

在上面章节说明了服务实例和组件实例的区别，在 jdonframework.xml 中配置 POJO 既可以组件实例获得，也可以服务实例获得。

首先，我们需要在 jdonframework.xml 中定义自己的 POJO 服务，例如在 jdonframework.xml 有如下两行定义：

```
<pojoService name="userJdbcDao" class="news.container.UserJdbcDao">
```

那么在应用程序中需要访问 UserJdbcDao 实例有以下两种方式：

第一. 通过 WebAppUtil 工具类的 getService 方法获得服务实例，如：

```
UserDao ud = (UserDao)WebAppUtil.getService("userJdbcDao", request);
```

getService 方法每次返回的一个新的服务实例对象，相当于 new 一对象。如果对象池拦截器被配置，那么这里返回的就是从小对象池中借用的一个对象。

第二. 通过 WebAppUtil 工具类的 getComponentInstance 方法获得组件实例，这也是获得 UserJdbcDao 一个实例，与服务实例不同的是，每次获得组件实例是同一个对象，因此，如果这个服务中如果包含上次访问的状态或数据，下次访问必须使用到这些状态和数据，那么就必须使用 getComponentInstance 方法获得服务实例。

注意，以上方式是假定你获得一个 POJO 实例，是为了使用它，也就是说，是为了访问它的方法，如访问 userJdbcDao 的 getName 方法，就要使用上述方式。

如果你不是为了使用它，而是作为别的 POJO 服务的输入参数，如构造器的输入参数，那么完全不必通过上述方式，你只要直接使用上述方式获得那个 POJO 服务的实例就可以，因为容器自动完成它们的匹配。

还有一点要求注意的是：使用 getService 获得服务实例，必须该服务类有一个接口，这样才能将 getService downcasting 下塑为其接口，否则只能是一个普通 Object，你获得后无法使用它。当然 getComponentInstance 没有这样限制。

## 如何编写一个 POJO 类？

既然在 Jdon 框架中获得 POJO 服务这么方便，那么 POJO 服务类的编写是否有特殊规定，回答是没有，就是一个普通的 Java 类，当然如果你需要在这个类引用其他类，最好将其他类作为构造器参数，如 A 类中引用 B 类，A 类的写法如下：

```
class A {  
    private B b;  
    public A(B b){  
        this.b = b;  
    }  
    ....  
}
```

这样，在 jdonframework.xml 中配置如下两行：

```
<pojoService name=" a" class="A">
<pojoService name=" b" class="B">
```

如果你希望你的 POJO 服务能够以对象池形式被访问，那么你的类需要 implements `com.jdon.controller.pool.Poolable` 或者在类前加入 `@Poolable` 元注解 Annotation

例如 上面的 A 继承了 `Poolable` 或加入了 `@Poolable` 元注解 Annotation，那么当有并发两个以上客户端访问服务器时，对象池将同时提供两个以上 A 实例为客户端请求服务，提高了并发访问量。

注意，当 A 的实例同时为两个以上时，A 引用的 B 是否也是每个 A 实例拥有一个 B 实例呢？也就是说，B 实例是否也是两个以上？

这取决于 A 对 B 的调用方式，如果 A 和 B 都在 `jdonframework.xml` 中配置，也就是都被注册到 Jdon 容器中，那么此时 B 永远只有一个，也就是单例的。

如果 B 的创建是在 A 内部，如下：

```
class A {
    private B b = new B();
    ....
}
```

这样，B 实例的创建是跟随 A 实例一起创建，因此，A 有多少个，B 就有多少个。

## 如何获得一个 POJO 服务的运行结果

在应用系统中，我们不但可以通过[“如何获得一个 POJO 服务实例”](#)获得一个 POJO 服务实例，然后在通过代码调用其方法，获得其运行结果，例如写入代码：

```
UserDao ud = (UserDao)WebAppUtil.getService("userJdbcDao", request);
userJdbcDao.getName(); //获得一个 POJO 服务的运行结果
```

除此之外，Jdon 框架还可以直接获得 POJO 服务的运行结果，只要你告诉它 POJO 类名、需要调用的方法名和相关方法参数类型和值，借用 Java Method Relection 机制，Jdon 框架可以直接获得运行结果。

实现这个功能，只要和接口 `com.jdon.controller.service.Service` 打交道即可：

```
public interface Service {

    public Object execute(String name,
                           MethodMetaArgs methodMetaArgs,
                           HttpServletRequest request) throws Exception;

    public Object execute(TargetMetaDef targetMetaDef,
                           MethodMetaArgs methodMetaArgs,
                           HttpServletRequest request) throws Exception;

}
```

`Service` 提供了两种获得某个 POJO 服务运行结果的方法。一个是以

Jdonframework.xml 中配置的 POJO 服务名称为主要参数，这是经常使用的一个情况。

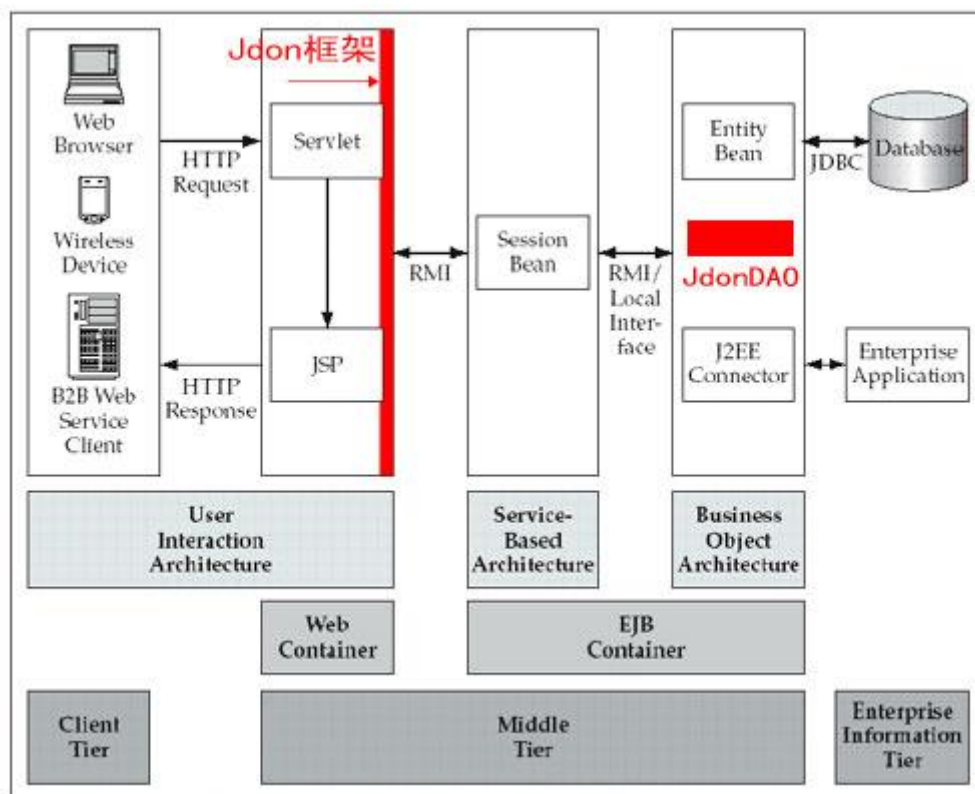
MethodMetaArgs 是包含调用的方法名、方法参数类型和方法参数值。

这种调用方式适合于 POJO 服务配置式调用，也就是说，通过编写自己的 XML 配置文件也可以实现如同写代码一样的服务调用和运行。

## 如何获得一个 EJB 服务实例？

Jdon 框架不只可以支持普通 JavaBeans，也就是 POJO 结构，也支持 EJB 架构，使用 EJB 的好处是能够获得可伸缩的、分布式的强大计算性能，当然 EJB 的开发需要借助 JBuilder 之类商业开发工具的图形开发功能才能方便快速实现。

使用 Jdon 框架可以开发出基于 Struts+Jdon+EJB 的标准 J2EE 架构系统，Jdon 框架在标准的 J2EE 架构中所处位置如下图的红色标记：



从上图底部向上看，是 J2EE 架构从抽象到具体技术的演变划分，J2EE 是一个中间件系统，那么中间层包括哪些部分呢？主要区分 Web 层和 EJB 层，Web 层主要是完成用户交互操作功能，Jdon 框架主要部分就位于这个 Web 层最末端，直接和 EJB 层打交道；同时 Jdon 框架有一部分 JdonDAO 运行在 EJB 容器中。

如何在表现层如 Struts 中获得一个 EJB 实例呢？

如同获得 POJO 实例一样，也是通过 WebAppUtil 工具类的 getService 方法获得，如果在 jdonframework.xml 中有如下配置：

```
<ejbService name="newsManager">
  <jndi name="NewsManager" />
  <ejbLocalObject class="news.ejb.NewsManagerLocal"/>
</ejbService>
```

那么，通过下面代码就可以访问 NewsManagerLocal:

```
NewsManagerLocal nm = (NewsManagerLocal)WebAppUtil.getService("newsManager", request);  
这样就访问 nm 这个 EJB 对象的方法了。
```

## 如何编写一个 EJB 类？

Jdon 框架对于一个 EJB 类的编写没有任何约束和规定。

但是，如果你原来使用 POJO 服务实现你的服务层，想无缝迁移到 EJB 服务，那么此时你的 Session Bean 需要继承 implements 原来 POJO 服务的接口，同时在 jdonframework.xml 的 ejbService 加入 interface 配置，指定原来 POJO 服务接口类，这样才能保证原来代码中通过 getService 方法获得服务实例的调用代码无需改变：

```
<ejbService name="testService2" >  
    <jndi name="TestEJB" />  
    <ejbLocalObject class="com.jdon.framework.test.ejb.TestEJBLocal"/>  
    <interface class="com.jdon.framework.test.service.TestService" />  
</ejbService>
```

## 如何获得 EJB 服务运行结果？

使用方式和获得 POJO 服务一样。

这种方式运行原理简要如下：当知道一个 EJB/POJO 的接口，通过 Proxy.newProxyInstance 生成一个动态代理实例（InvocationHandler 的实现子类）即可，以后对 EJB/POJO 的调用，实际由这个动态代理实例的 invoke 自动激活，从而使用 Method Reflection 实现 EJB/POJO 的调用。



## AOP 拦截器

Jdon 框架拦截器在 6.2 版本以后得到加强和细化，不过和 AspectJ 等 AOP 框架有些区别，主要是方便性方面，这些 AOP 框架有专门的复杂 Ponitcut 表达脚本，需要再学习，而 Jdon 框架则还是使用 Java 完成复杂的拦截方式。

有两种拦截器写法，一种是写在被拦截的类代码中，而拦截器则无特别写法，解放了拦截器；另外一种拦截语法写在拦截器中，解放了被拦截者，实际中可根据自己的情况灵活选用。

### @Introduce

@Introduce(名称) 为当前类引入拦截器，可以用在 @Model 或 @Component 中，名称是拦截器的 @Component(名称)。 例如 @Introduce("c") --- @Component("c")

在当前类具体方法上使用 @Before @After 和 @Around 三种拦截器激活的方式。注意使用 @Around 对拦截器有特定写法要求，其他无。

```
@Component("a")
@Introduce("c") // Introduce a interceptor that component name is "c"
public class A implements AInterface {

    @Before("testOne") // Interceptor will action before "myMethod "
    public Object myMethod(@Input() Object inVal, @Returning() Object ret) {
        System.out.println("this is A.myMethod is active!!!! ");
        int i = (Integer) inVal + 1;
        return i;
    }

    @After("testWo") // Interceptor will action after "MyMethod2"
    public Object myMethod2(Object inVal) {
        System.out.println("this is A.myMethod2 is active!!!! ");
        int i = (Integer) inVal + 1;
        return i;
    }
}
```

Annotations and their targets in the code above:

- `@Introduce("c")`: Introduces an interceptor named "c".
- `@Before("testOne")`: Interceptor will action before "myMethod".
- `@Input()`: @input parameter will inject into the inteceptor's testOne method.
- `@Returning()`: testOne return result inject this.
- `@After("testWo")`: Interceptor will action after "MyMethod2".

被引入的拦截器 c 的代码如下，没有特别要求，只要配置 @Component("c") 就可以，或者使用 XML 配置：<component name="c" class="xxxxxx" />也可以。

```
@Component("c")
public class C {

    //被拦截器的@Input 参数将注射到 inVal 中
    public Object testOne(Object inVal) {
        ....
    } //testOne 方法 return 结果将被注射到被拦截器的@return 中
}
```

```

//被拦截器的方法 myMethod2 返回值将被引入此方法的 inVale
public Object testWo(Object inVal) {
    .....
}
}

```

@Introduce 的 @Around 则是一个特殊情况，因为 around 表示在被拦截点周围，实际是 Before + After 综合写法，@Around 需要对拦截器写法有一些特殊要求，如下：

```

@Interceptor(“aroundAdvice”)//1. 需要表明自己是 Inteceptor
public class AroundAdvice implements MethodInterceptor {

    //2.需要实现 org.aopalliance.intercept.MethodInterceptor 接口
    //3.完成接口的 Invoke 方法
    public Object invoke(MethodInvocation invocation) throws Throwable
        Object o = invocation.proceed();//在此方法前后写包围 around 代码。
}

```

## @Interceptor

@Interceptor 是将拦截定义在拦截器这边的语法，和 AspectJ Spring 写法类似，但是没有他们的复杂 pointcut 专门表达式。

```

//1. 指定被拦截的类是@Component(“a”)和@Component(“c”)
@Interceptor(name = "myInterceptor", pointcut = "a,c")
public class MyInterceptor implements MethodInterceptor {

    //2.需要实现 org.aopalliance.intercept.MethodInterceptor 接口
    //3.完成接口的 Invoke 方法
    public Object invoke(MethodInvocation methodInvocation) throws java.lang.Throwable {
        ...
    }
}

```

如果直接写 @Interceptor(“myName”) 那么就表示该拦截器的拦截点 pointcut 是针对所有 Service 或 POJO Services，相当于在 aspectj.xml 中的配置

@Interceptor 中的 pointcut 是指被拦截的类名称，可以是多个类一起拦截。

## 增删改查(CRUD)快速开发

Jdon 框架提供了基于 Struts 的快速开发,可以快速开发出数据的增删改查(CRUD)的 MVC 配置以及批量查询等功能。

本章是与 Jdon 框架其它功能分离的,你可以只使用 Jdon 框架的组件管理和 AOP 功能,而不必使用基于 Struts 的快速开发功能,随着新的表现层技术出现,Jdon 框架将推出基于的技术如 JSF 的快速开发框架。

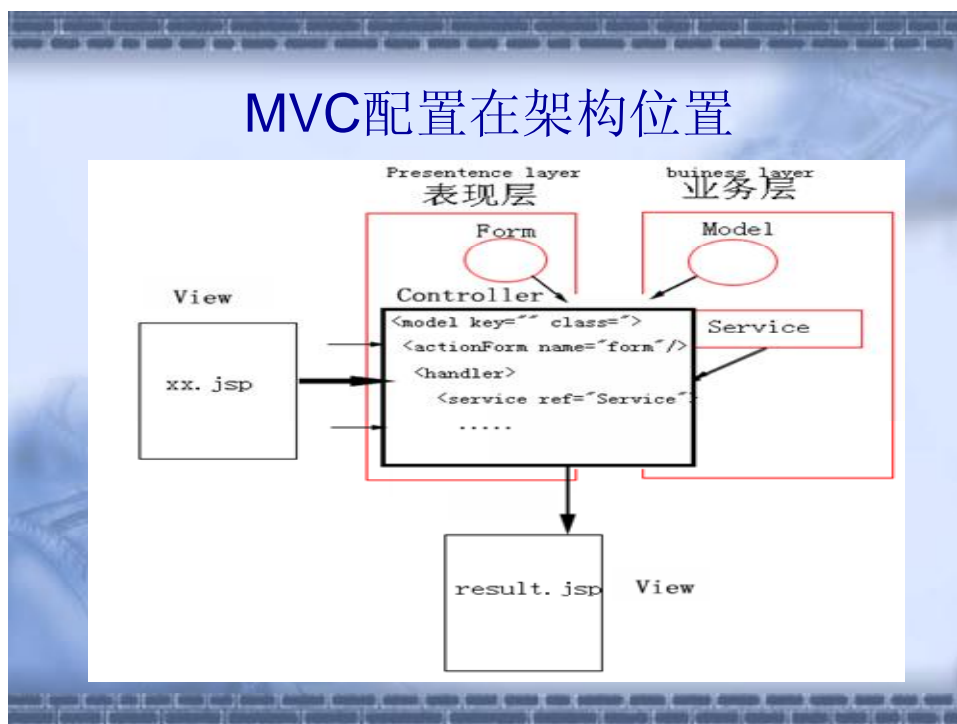
关于为什么使用 jdon 框架的 CRUD 功能,除了在设计速度上有极大提高以外,在设计上也有很多优点:

边界类,控制类和业务接口的关系:

<http://www.jdon.com/jive/article.jsp?forum=91&thread=21245>

J2EE 中几种业务代理模式的实现和比较:

<http://www.jdon.com/articheckt/businessproxy.htm>



## struts 基础简述

Jdon 框架的 CRUD 功能是基于 struts 实现,主要工作是表现层的配置,那么是否需要程序员熟悉 struts 呢?其实完全不必。

但是,需要了解 struts 的一些基础:struts 主要有一个配置文件,在 Web 目录的 WEB-INF 下 struts-config.xml,其中主要有两部分配置: ActionForm 和 Action。

ActionForm 配置:主要是配置你设计的 ActionForm 子类;例如:

```

<struts-config>
  <form-beans>
    <form-bean name="accountForm"
      type="com.jdon.framework.samples.jpstore.presentation.form.AccountForm"/>
  </form-beans>
  ....
</struts-config>

```

Action 配置：相当于一个 servlet，也需要在 struts-config.xml 中 action-mappings 配置，例如：

```

<action-mappings>
  <action name="accountForm" path="/shop/newAccountForm"
    type="com.jdon.strutsutil.ModelViewAction" scope="request">
    <forward name="create" path="/account/NewAccountForm.jsp" />
  </action>
</action-mappings>

```

action 中配置简要说明：

name 是 ActionForm 名称，是前面 ActionForm 名称；

path 是该 action 在浏览器中调用的 url 名称，使用 [http://xxx/web 名称/shop/newAccountForm.do](http://xxx/web名称/shop/newAccountForm.do) 就可以调用这个 action，在 path 值后面加一个.do 即可，.do 是在 web.xml 中配置的：

```

<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>

```

你可以改变\*.do 配置，改为\*.shtml，那么上面 action(Servlet)调用变成 [http://xxx/web 名称/shop/newAccountForm.shtml](http://xxx/web名称/shop/newAccountForm.shtml)

type 是你继承 struts 的 Action 实现子类：如 com.jdon.strutsutil.ModelViewAction 是 Jdon 框架的实现子类。

Scope 是值你的 ActionForm 生存周期，一般有 request 或 session，相当于 Jsp 中 useBean 中的 scope，表示 ActionForm 实例有效范围，request 表示用户发出一个请求周期；session 表示是从用户登陆到退出为止，这个 ActionForm 实例一直存在，你可以引用同一个实例。

```

<forward name="create" path="/account/NewAccountForm.jsp" />

```

这表示 Action 执行完成后，需要推出的 Jsp 页面，name 值是你在 Action 执行子类中定义的名称；path 是你的 jsp 页面相对路径。

所以，编写一个传统意义上的 Jsp 页面，在 struts 变成下面几步：

编写一个 ActionForm 实现子类

编写一个 Action 实现子类，完成其中方法 execute 内容。

配置 struts-config.xml

使用 struts 标签编写 Jsp 页面，这时的 Jsp 页面和传统的 Jsp 相比：已经没有 Java 代码，完全 Html 语法。

由以上可见，struts 在实现设计优化目的同时，实际增加编程复杂性，Jdon 框架试图保证 struts 优点的前提下，简化编程复杂性，CRUD 的功能对上述步骤优化如下：

ActionForm 是 Model 代码复制；

无需编写 Action 实现子类；

模板化配置 struts-config.xml，参考别的配置，稍微修改即可。

简化 Jsp 页面编写数量；四个功能只需编写一个 Jsp。

struts 学习资源：<http://www.jdon.com/idea/strutsapp/04005.htm>

## CRUD 流程原理

传统的直接基于 Struts CRUD 实现方式：增删改查是四个操作在传统表现层中需要分四次流程，每次流程中有大部分开发或配置类似，这种低层次的重复劳动降低开发者的积极性。

Jdon 框架抽象出传统增删改查四个流程中的共同部分，并将其配置固化，实现一定开发模式限制，从而大大提高开发效率。具体设计原理如下：

(1) 一般情况下，一个系统的操作用户（以下简称用户）新增或修改数据，首先要推送给他一个 Jsp 页面，如果是新增页面，就是一个空白表单的 Jsp 页面；如果是修改页面，则先到数据库中查询获得原来的数据，然后推出一个有数据表单的 Jsp 页面，用户才能在原来的数据上修改或编辑。

由于在 MVC 模式中，Jsp 页面只是一个页面输出，或者说不能有任何 Java 功能实现，因此上面修改页面推出前需要查询数据库这个需要 Java 实现的功能不能在 Jsp 页面中实现，只能在 Jsp 页面前加一个 Action，这样，修改页面的推出流程变为不是直接调用 Jsp 页面，而是：action.do ---> Jsp 页面，首先调用 Action；然后才由 Action 推出 Jsp 页面。

这个 Action 实现我们称为 ViewAction，专门用于控制输出 Jsp 界面，新增 Jsp 页面的推出前我们也加上这个 ViewAction，这样无论是新增 Jsp 页面或修改 Jsp 页面，都是由 ViewAction 推出，那么到底是推出新增 Jsp 页面还是修改 Jsp 页面呢？

关键是 ViewAction 的输入参数 Action 的值，根据 Action 的值来判断是新增还是修改。如果设置 Action 值如为空或为 create 值（如 <http://xxx/viewAction.do?action=create>），则直接输出新增性质的 JSP 页面；而 Action 值如为 edit（如 <http://xxx/viewAction.do?action=edit>），则是要求输出进行编辑页面，根据 ID 查询数据库获得存在的数据，然后输出编辑修改性质的 JSP 页面。

当然，在 ViewAction 中还有一些具体参数的检查，如果是编辑，则关键字 ID 不能为空，因为后台要根据此 ID 为主键，查询数据库其相应的记录，如果数据库未查询到该主键的记录，则需要显示无此记录等信息。

(2) 创建有关该数据的 JSP 页面，既用于新增页面，也用于修改页面。将该 Jsp 页面作为 ViewAction 的输出页面。该 Jsp 页面结构如下：

```
<html:form action="/XXSaveAction.do">
  <html:hidden property="action" />
  <!--该 action 的值是调用 viewAction.do?action 参数的值 -->
  .....
</html:form>
```

以上是 ViewAction 的表单 Jsp 页面流程

(3) 创建 SaveAction，

当用户填写完该 Jsp 页面中的表单数据将提交给一个新 Action 子类实现：专门用于接受表单数据并保存持久化它们。SaveAction 用来接受提交表单的数据，不同于 ViewAction 专门用于输出 Jsp 表单页面，该 Action 专门用于接受 Jsp 表单页面提交的数

据。

SaveAction 中主要是调用业务层 Service 实现数据持久化操作，调用 Service 之前，需要将表单的数据 ActionForm 转为一个对象（DTO 对象），然后作为方法参数传送给 Service 具体方法，Service 处理完成后，返回结果对象，SaveAction 还需要检查 Service 是否操作成功等。

一个数据的增删改查流程有上面总结的流程组成：

1. ViewAction -> 表单 Jsp 页面
2. 用户填写表单页面提交 -> SaveAction --> 结果 Jsp 页面

一个数据新增删除修改流程需要创建两个 Action，一个 Jsp 页面；当然 Struts 1.2 中已经通过 DispatchAction 解决了需要创建两个 Action 问题，但是还是需要 Action 编码，而 Jdon 框架提供了 Struts 缺省 Action 实现(也就是 ViewAction 和 SaveAction 代码实现)，就不需要另外 Action 编码，只要直接进行 struts-config.xml 配置即可。

## Model 和 ModelForm

Model 是域模型，是采取领域模型分析法从系统需求分析中获得的，反映了应用系统的本质，Model 是一个简单的 POJO，属于数据类型的 POJO，区别于 POJO 服务。从传统意义上理解，Model 设计相当数据表设计，在传统的过程化编程中，一个数据库信息系统设计之前我们总是从数据表设计开始，而现在我们提倡是从域模型提炼开始。

Jdon 框架对模型对象建立有两个要求，继承 com.jdon.controller.model.Model，每个 Model 有一个主键。

每个 Model 需要一个主键

每个 Model 必须有一个主键，就如同每个数据表设计都有主键一样，Model 的主键是和其对应的数据表主键在类型上是一致的。例如一个 Model 为 Account 如下：

```
public class Account extends Model {  
  
    private String username;  
    private String password;  
    private String email;  
    private String firstName;  
    private String lastName;  
    .....  
}
```

其持久化数据表 account 的表结构：

```
CREATE TABLE ACCOUNT (  
    username varchar(80) NOT NULL default "",  
    email varchar(80) NOT NULL default "",  
    .....  
    PRIMARY KEY (username)  
)
```

account 表的主键是 username，数据类型是 varchar，通过 JDBC 对应可知其对应的

Java 类型是 String，那么 Account 模型的主键也必须是 String，当然主键名称可以不一样，Account 是 username，而 account 的表的主键可以是 userId。下面以 MySQL 为例，说明 JDBC 缺省在对象和数据表之间的类型对应：

| Model 主键<br>类型 | 数据表主键类<br>型    |
|----------------|----------------|
| String         | varchar 或 char |
| Integer        | int            |
| Long           | BIGINT         |

主键类型（包括 Model 和数据表）一般推荐统一使用 String 或 Integer/int 或 Long/long。缺省建议使用 String（Oracle 数据库处理字符串中空格有些麻烦，需要特别注意）。下载案例包中 SimpleJdonFrameworkTest 主键是使用 String 类型，而 simpleMessage 案例包主键类型是 Long 类型。

注意：模型类中一定要提供主键的 setter 和 getter 方法，Jdon 框架内部是根据主键的 getter 方法返回类型察知主键的类型。

如果你的数据表没有主键怎么办？那么强制给它一个主键，这可以由一个专门序列器产生。

在使用本框架前提下，数据表字段不同类型选择所带来的性能等差异非常小，几乎忽略不计。相关数据类型讨论可见：  
<http://www.jdon.com/jive/article.jsp?forum=91&thread=23875>

## Model 或 ModelIF

Jdon 框架提供的 com.jdon.controller.model.Model 是一个抽象类，这需要你的 Domain Model 继承这个抽象类，由于 Java 的单继承特点，有可能使你的 Domain Model 类再不能继承其他抽象类，为此，在 Jdon 框架 1.4 版本以后，提供了 com.jdon.controller.model.ModelIF 接口，5.6 版本以后，提供了 @Model 元注解。

本说明书中谈到 Model 对象，也同时意指 ModelIF，也全部可以使用 @Model 来替代。

## Model 配置

Model 在 jdonframework.xml 中配置，如下：

```
<model key="username" class ="com.jdon.framework.samples.jpeteststore.domain.Account">
    .....
</model>
```

key 的值就是指定 Model 的主键，这里是 username，class 是 Account 的类，这样就定义了一个 Model。

Model 配置除定义 Model 自身属性以外，还需要定义了子属性：actionForm 和 handler，这两个定义在下面章节描述。

ModelForm 是 Model 的映射

Jdon 框架通过映射设计，保证表现层、Jdon 框架、服务层和持久层之间能够解耦独立，相互可插拔、脱离或组合。

ModelForm 相当于 Struts 的 ActionForm，属于界面对象，使用 Jdon 框架一个要求是：ModelForm 的内容（字段/方法）需要大于等于 Model 的内容，这样，才能将表现层的数据传送到 Model 业务层处理，ModelForm 内容可以多于 Model 内容，这些多余内容可能只与界面有关，不涉及业务逻辑。

Model 和 ModelForm 映射是通过相同字段拷贝实现的，也就是这两个类之间有相同的字段属性，那么字段属性的值可以在他们之间拷贝转移。例如 Model Account 的代码：

```
public class Account extends Model { //也可以用@Model 替代，无需继承 Model 了

    private String username;
    private String password;
    private String email;
    private String firstName;
    private String lastName;
    .....

}
```

那么其 ModelForm 的类代码如下：

```
public class AccountForm extends ModelForm {

    private String username;
    private String password;
    private String email;
    private String firstName;
    private String lastName;
    .....

}
```

这样两者代码保证它们之间的映射，这是使用 Jdon 框架的规则之一。

Jdon 框架是使用 org.apache.commons.beanutils.PropertyUtils.copyProperties 方法在这两个类的对象之间进行字段拷贝的，因此两个类对应字段类型必须一致。

### ModelForm 配置

ModelForm 就是 ActionForm，因此只需要在 struts-config.xml 的 <form-beans> 中定义 ActionForm 就可以，如下：

```
<struts-config>
  <form-beans>
    <form-bean name="accountForm"
                type="com.jdon.framework.samples.jpeteststore.presentation.form.AccountForm"/>
  </form-beans>
</struts-config>
```

注意 ModelForm 的名字是 accountForm，因为 Model 和 ModelForm 是映射对应关系，我们需要告诉 Jdon 框架这种对应关系。



那么，拓展前面的 Model 配置，在 jdonframework.xml 配置如下：

```
<model key="username" class="com.jdon.framework.samples.jpeteststore.domain.Account">
  <actionForm name="accountForm"/>
  .....
</model>
```

在 jdonframework.xml 中的配置是让 Model 和 ModelForm 有唯一的对应关系。

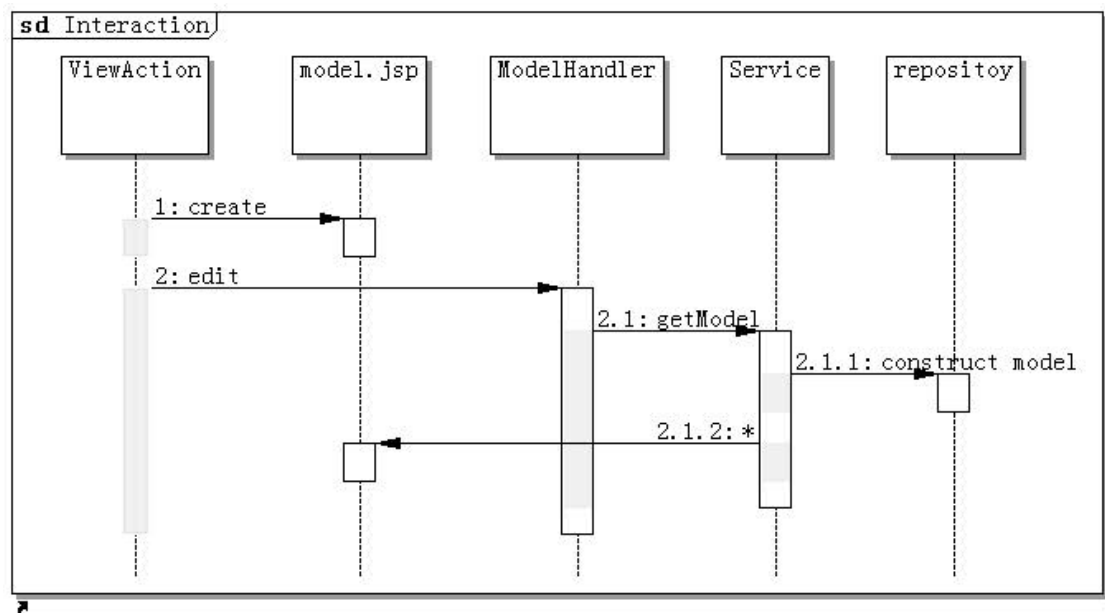
在 jdonframework.xml 中配置的 ActionForm 名称必须是全局唯一的，也就是说，如果有两个 ActionForm 名称一样的配置，就是它们的 Model class 值不一样，Jdon 框架也视为是一样的。

## CRUD 基本流程

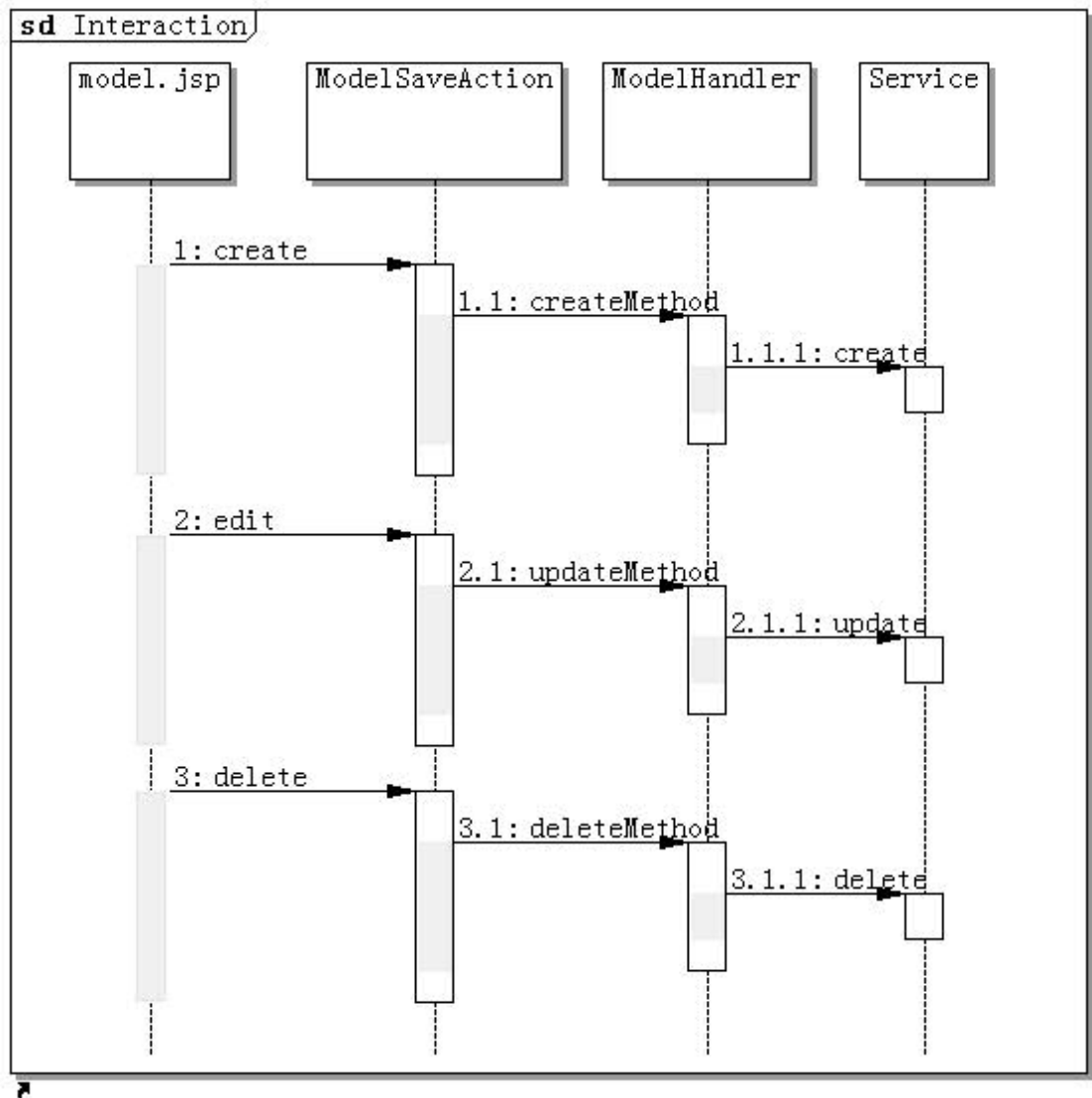
这里首先介绍一下 Jdon 框架在上面思路延伸抽象设计的思路：

一个数据的增删改查流程有上面总结的流程组成：

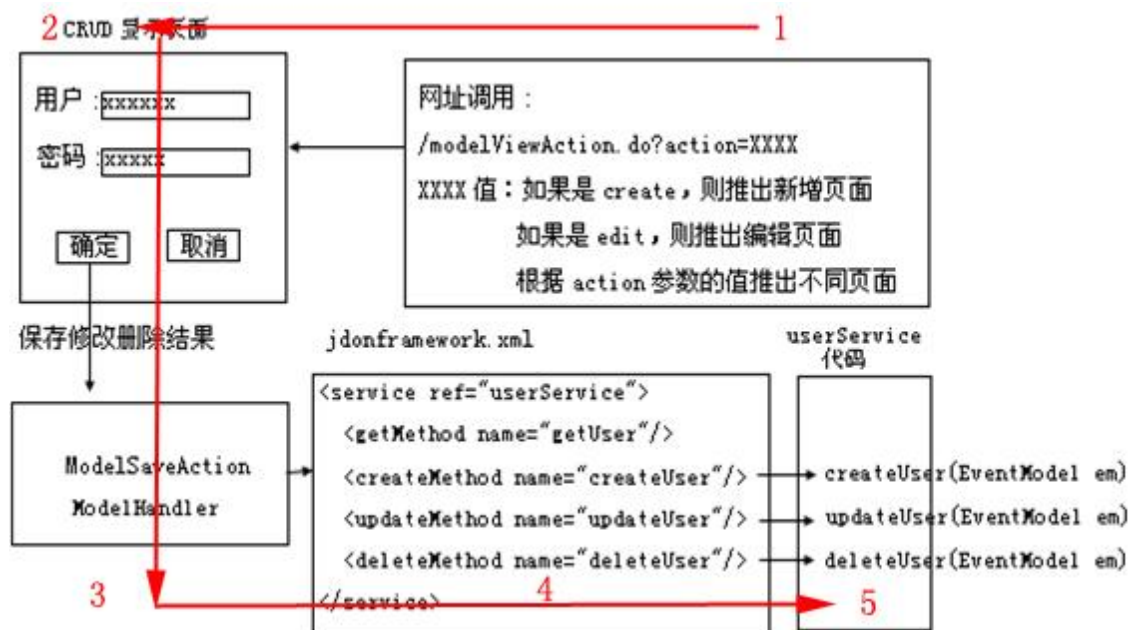
1.ViewAction -> 表单 Jsp 页面



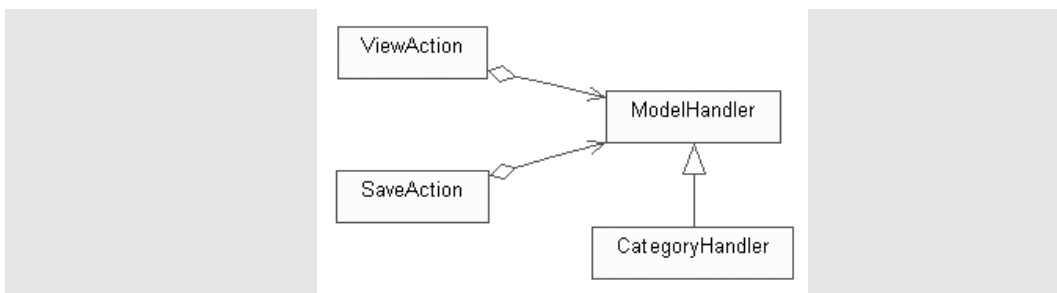
2.用户提交已经填入数据的 Jsp 页面 -> SaveAction --> 结果 Jsp 页面



如下面流程图：通过浏览器网址（或 Html 链接）调用我们推出用于新增或编辑的页面，确定保存输入的数据后，激活 ModelSaveAction，ModelSaveAction 通过 jdonframework.xml 的配置，调用相应的 Service 类，这个 Service 类是你自己编制的，根据原始网址或 Html 链接调用的参数 action 不一样，调用不同的 Service 接口中相应的方法。



上图中, 在 ModelSaveAction 步骤有一个 ModelHandler, 它其实是真正和 Service 接口打交道的类, 必要时, 你可以自己继承实现自己的 ModelHandler 子类, 它与 ModelSaveAction 的关系如下:

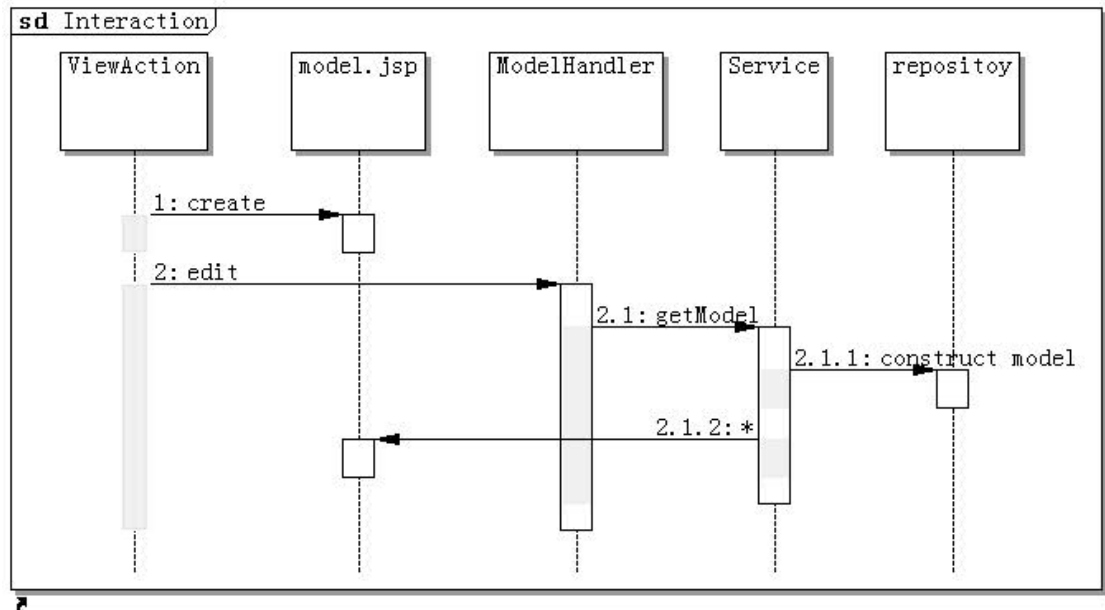


ModelViewAction 和 ModelSaveAction 中有关具体每个数据有不同的操作, 都委托给 ModelHandler 实现, 用户使用框架时, 只要继承实现 ModelHandler, 填补异性方法就可以。缺省情况下, 上图中的 jdonframework.xml 配置实际就是 ModelHandler 实现, 所以一般不必自己再编制 ModelHandler 实现了, 但是特殊情况下可能还是需要的, 提供缺省和特殊两种方便。

下面我们针对上面流程图对每一个环节详细描述如下:

## CRUD 页面显示

.ViewAction -à 表单 Jsp 页面的流程图如下:



#### 调用 ModelAndViewAction

Jdon框架中的 `com.jdon.strutsutil.ModelViewAction` 主要是实现在推出让用户新增数据或编辑数据的 Jsp 页面的之前的准备工作。它主要是做 Jsp 页面推出前的准备工作，然后推出不同的 Jsp 页面。

`ModelViewAction` 根据调用参数 `action` 的值，判断用户发出的是新增或编辑命令。

Action 有两个规定值 `create` 或 `edit`，也就是说，只有三种 URL 调用形式：

新增 URL: <http://xxx/XXXModelViewAction.do?action=create>

新增 URL: <http://xxx/XXXModelViewAction.do>

编辑 URL: <http://xxx/XXXModelViewAction.do?action=edit>

如果用户在浏览器发出这三种 URL 调用，将激活 `ModelViewAction`，`ModelViewAction` 将根据参数 `action` 的值分别推出用于创建性或编辑性的页面。下面分别详细说明这两种流程：

- 1.在创建性 Jsp 页面推出之前，要做一些准备工作，由于 Jsp 页面中表单数据（`<form>`之间数据）是由 `ModelForm`（`ActionForm`）概括的，所以，创建性 Jsp 页面推出前的准备工作主要是 `ModelForm` 的初始化创建，`ModelForm` 是 Struts 的 `ActionForm` 继承者，`ModelForm` 是抽象页面表单数据，它是 `Model` 在表现层的映射影子。

`ModelForm` 的初始化创建很重要，它决定了推出给用户创建性页面的内容，需要程序员介入度很高，Jdon 框架提供有几种初始化 `ModelForm` 方法：

首先检查 `ModelHandler` 的 `initForm` 方法，检查用户有无自己实现初始化 `ModelForm` 实现；如果无，则用 Struts 的自动创建 `ActionForm`（`ModelForm`），这时的 `ModelForm` 实例是空对象，这时还有另外一种方式提供用户初始化 `ModelForm` 实例的方法：调用 `ModelHandler` 的 `initModel` 方法，将该方法返回的 `Model` 对象拷贝到 `ModelForm` 中，这符合 `Model` 和 `ModelForm` 相互映射的关系。下面章节将描述 `ModelHandler` 的 `initForm` 和 `initModel` 方法有何区别。

- 2.推出编辑性页面之前的准备工作主要是：根据主键查询从服务层获得一个已经存

在的数据模型，简单地说：根据主键从数据库查询已经存在的一个记录，这样，推出的 Jsp 页面中包括数据的表单，用户可修改编辑。

这个工作涉及两部分：**ModelForm** 创建；然后从服务层获得一个有数据的 **Model**，将 **Model** 数据拷贝到 **ModelForm** 实例中。

注意：从 Jdon Framework 1.4 以后，参数名不一定是 action，可以是 method，也就是可以如下调用：

新增 URL: <http://xxxx/XXXModelViewAction.do?method=create>

新增 URL: <http://xxxx/XXXModelViewAction.do>

编辑 URL: <http://xxxx/XXXModelViewAction.do?method=edit>

这里的 action 或 method 参数值将传送到 **ModelViewAction** 推出的 Jsp 页面中。

## Struts 配置和 Jsp

根据不同域模型，我们有相应的不同的配置，例如：有一个 **Model** 为 **A**，那么为了推出 **A** 新增或修改或删除的页面，我们需要在 **struts-config.xml** 配置如下：

```
<action name="aActionForm" path="/aAction" type="com.jdon.strutsutil.ModelViewAction">
  <forward name="create" path="/a.jsp" />
  <forward name="edit" path="/a.jsp" />
</action>
```

这是一个标准的 **Action** 配置，**A** 的 **ActionForm** 为 **aActionForm**，新增修改删除 **A** 的 Jsp 页面为 **a.jsp**，如上面配置，这样，如果用户从浏览器网址如下调用：

<http://xxxxx/aAction.do?action=create>

这样调用将推出 **a.jsp** 用于新增；

<http://xxxxx/aAction.do?action=edit>

这样调用将推出 **a.jsp** 用于修改删除。

再举例，如果 **Model** 为 **B**，**B** 的 **ActionForm** 为 **bActionForm**，Jsp 页面为 **b.jsp**，那么 **Struts-config.xml** 只要如下配置：

```
<action name="bActionForm" path="/bAction" type="com.jdon.strutsutil.ModelViewAction">
  <forward name="create" path="/b.jsp" />
  <forward name="edit" path="/b.jsp" />
</action>
```

如果 **Model** 为 **c**，那么 **struts-config.xml** 配置如下：

```
<action name="cActionForm" path="/cAction" type="com.jdon.strutsutil.ModelViewAction">
  <forward name="create" path="/c.jsp" />
  <forward name="edit" path="/c.jsp" />
</action>
```

总结上面配置，**Model** 名称不同，只要更换一下有关 **Model** 名称即可，其余基本不变，配置具备模板化编程的特点，拷贝粘贴然后修改，修改且有规律。

通过以上配置，我们确定了 **a.jsp** 或 **b.jsp** 是 **ModelViewAction** 推出的页面，我们再看看 **a.jsp** 或 **b.jsp** 的代码：

```
<html:form action="/userSaveAction.do" method="POST" onsubmit="return checkPost();">
<html:hidden property="action"/> <!-- 这是必须的 -->

UserId: <html:text property="userId"/> <!-- 主键一般是必须的 -->
<br>
```

```
Name: <html:text property="name"/>
<br>
<html:submit property="submit" value="Submit"/>
</html:form>
```

这个是一个 CRUD 页面，如果是修改，那么其中 `userId` 和 `Name` 将有值。注意两点：

必须有 `action` 隐含字段，当然参数名称也可能是 `method`，取决于你如何调用 `ModelViewAction`。

必须有主键隐含字段，上例主键 `userId` 是可编辑的，一般主键是专门序列号产生器产生的，不能修改，但是必须包含在表单中。

纯粹显示使用 `ModelDispAction`

前面是 `ModelViewAction` 显示数据页面是为了编辑或删除，如果你纯粹是为了显示，并无需考虑后续操作结果，那么可以在配置 `Struts-config.xml` 中使用 `ModelDispAction`，其他 Jsp 页面和 `ModelViewAction` 处理尾一样的。

例如：你有一个如下类，你希望显示这个类调用的结果 `UserPaper`。

```
public UserPaper getPtime(EventModel em){
    UserPaper userPaper=(UserPaper)em.getModel();
    try{
        logger.debug("first User:"+userPaper);
        logger.debug("userDao:"+userpaperDao.getPtime());
        logger.debug("userDao.getPtime:"+userpaperDao.getPtime().getPtime());
        userPaper=userpaperDao.getPtime();
        logger.debug("object Userpaper value:"+userPaper.getPtime());

    }catch(Exception ex){
        logger.error(ex);
        em.setErrors(Constants.MESSAGE_SAVE_ERROR);
    }
    System.out.println("last user Paper:"+userPaper.getPtime());
    return userPaper;
}
```

使用[服务命令调用模式](#)是无法返回值的，所以这个 `getPtime` 不能用这个方式，因为返回值你要显示出来，显示就要有 Jsp 页面。几个步骤：

1.将 `public UserPaper getPtime(EventModel em)` 抽象为接口 如 `getPtimeIF` 并实现这个接口。

2.为 `UserPaper` 做一个 `ModelForm` 叫 `UserPaperForm`

3.在 `struts-config.xml` 中配置：

```
<action name="userPaperForm" path="/你的路径"
    type="com.jdon.strutsutil.ModelDispAction" scope="request" validate="false">
    <forward name="success" path="/你要显示 UserPaper 值的 JSP 页面" />
</action>
```

4.其他步骤参考上面 ModelViewAction 章节。

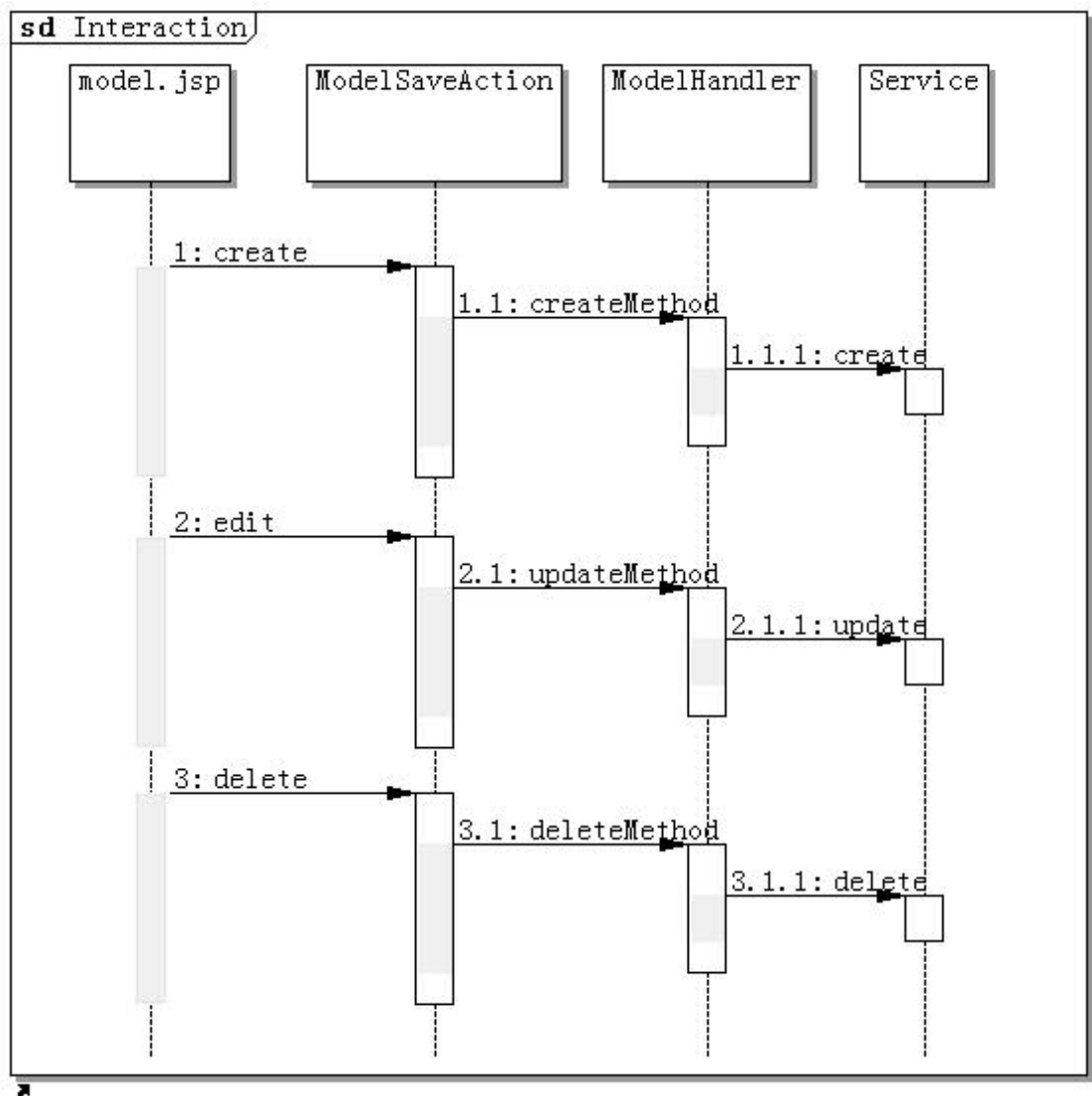
## 接受 CRUD 处理

当用户在浏览器中新增或修改页面填入数据后，将按“确定 submit”按钮提交后台进行保存或删除等处理，从 CRUD 基本流程章节的流程图可以看到看到：这主要是由 ModelSaveAction。ModelSaveAction 主要委托 ModelHandler，而 ModelHandler 则是用户自己定义的 Jdonframework.xml 的 MVC 配置实现的：

jdonframework.xml

```
<service ref="userService">
  <getMethod name="getUser"/>
  <createMethod name="createUser"/>
  <updateMethod name="updateUser"/>
  <deleteMethod name="deleteUser"/>
</service>
```

流程图如下：



## ModelSaveAction 原理

ModelSaveAction 是前面的 SaveAction 实现，它是真正的核心 Action 实现，专门将用户新增或修改后的数据提交到服务层，实现持久化如保存到数据库中。

ModelSaveAction 主要是委托 ModelHandler 实现数据提交和服务激活，这个过程比较单一有规律，用户可介入程度低，因此可以使用配置来代替代码实现。

我们看看 ModelSaveAction 是如何将 Jsp 页面提交的数据传递给后台，ModelSaveAction 从 ModelForm（ActionForm）中获得用户提交的数据，将其拷贝到相应的 Model 实例，然后将 Model 对象打包到 EventModel 对象中，将 EventModel 作为方法参数，调用 ModelHandler 的 serviceAction 方法，ModelHandler 的 serviceAction 则是调用服务层相应 Service 的对应的 create/update/delete 方法，所以 ModelSaveAction 主要工作委托给 ModelHandler 实现，我们将在下面研究重要的 ModelHandler 类。

## Struts 配置和 Jsp

ModelSaveAction 和 ModelAndViewAction 类似，根据 Model 不同，在 struts-config.xml 中配置如下：

```
<action name="aActionForm" path="/aSaveAction"
type="com.jdon.strutsutil.ModelSaveAction">
    <forward name="success" path="/aResult.jsp" />
    <forward name="failure" path="/aResult.jsp" />
</action>
```

如果 Model 为 B，那么 struts-config.xml 的配置是：

```
<action name="bActionForm" path="/bSaveAction"
type="com.jdon.strutsutil.ModelSaveAction">
    <forward name="success" path="/bResult.jsp" />
    <forward name="failure" path="/bResult.jsp" />
</action>
```

关于 ModelSaveAction 使用是在 Jsp 页面中赋值给 action：

```
<html:form action="/aSaveAction.do">
    <html:hidden property="action" />
    <html:text ....
    ....
</html:form>
```

该 Jsp 页面是新增或修改性质的页面，例如可以是前面介绍的 ModelAndViewAction 推出的 a.jsp 或 b.jsp。

至此，ModelViewAction 和 ModelSaveAction 实现了首尾衔接，实现了一个 CRUD 操作流程。

通过配置使用 Jdon 框架中的 ModelAndViewAction 和 ModelSaveAction，程序员避免了象开发普通 Struts 应用系统那样建立至少一个 Action 子类。

在这个流程中，需要和服务层服务交互，这需要由程序员根据具体程序定制，下面介绍程序员可介入定制的重要类 ModelHandler。



## CRUD 流程配置 jdonframework.xml

现在我们进入 CRUD 流程图的下面一个环节：表现层和业务层接口部分了。

前面我们说过，ModelViewAction 和 ModelSaveAction 其实是委托 ModelHandler 和 Service 接口打交道的，程序员需要定制实现的部分由继承 ModelHandler 实现。但是缺省情况下，可以通过 jdonframework.xml 配置实现。

配置实例说明

假设，我们已经编制了一个 Service 接口如下：

```
public interface AccountService {  
    Account getUser(EventModel em); //查询获得存在的 Model  
    void createUser(String username); //新增方法  
    void updateUser(EventModel em); //修改方法  
    void deleteUser(EventModel em); //删除方法  
}
```

其中 getUser 方法是由调用 ModelViewAction 调用的，其余都是由 ModelSaveAction 调用的，为了让 Service 的具体方法告诉 Jdon 框架，我们可以通过 jdonframework.xml 配置告诉它：

```
<handler>  
    <service ref="accountService">  
        <getMethod name="getUser"/> //getUser 对应 AccountService 接口的 getUser 方法  
        <createMethod name="createUser"/> //createUser 对应 AccountService 接口 createUser 方法  
        <updateMethod name="updateUser"/> // updateUser 对应 AccountService 接口 updateUser 方法  
        <deleteMethod name="deleteUser"/> // deleteUser 对应 AccountService 接口 deleteUser 方法  
    </service>  
</handler>
```

我们通过上述配置，建立了表现层 ModelViewAction/ModelSaveAction 和业务层 Service 之间的调用关系。当然这段配置应该放置在 jdonframework.xml 的 Model 配置中，以前面 Model 章节的配置为例子：

```
<model key="username" class="com.jdon.framework.samples.jpstore.domain.Account">  
    <actionForm name="accountForm">  
        <handler>  
            .....//上述配置插入在这里  
        </handler>  
    </model>
```

这样，你自己编写的 Service 接口嵌入 Jdon 框架的 CRUD 流程，非常简单。

你可能还有一个疑问，Struts 的 struts-config.xml 中的配置的 ModelVewAction 或 ModelSaveAction 是怎么定位自己的那个 Service，例如 struts-config.xml 如下配置：

```
<action path="/serviceAction" type="com.jdon.strutsutil.ModelViewAction"  
    name="accountForm" scope="request" validate="false">  
    <forward name="xxxxx" path="/result.jsp"/>  
</action>
```

然后，你定义了自己的 Service，如上面 jdonframework.xml 中的 accountService，那么 ModelViewAction 是如何找到 accountService，而不是其他定义的 accountService2

等 service 呢？

这是通过上面 `jdonframework.xml` 中的 `<actionForm name="accountForm"/>` 的这个 FormBean 名称，`accountForm` 这个 FormBean 是在 `struts-config.xml` 中定义的，它是 `struts-config.xml` 和 `jdonframework.xml` 两个配置之间的桥梁。

一个 FormBean 对应 `struts-config.xml` 中的一个 action 配置；同时对应 `jdonframework.xml` 中的一个 model CRUD 流程配置，这里的桥梁 FormBean 是指 FormBean 名称，而不是 FormBean 类，同一个 FormBean 可以配置多个 FormBean 名称，这是在 `struts-config.xml` 的 FormBeans 中配置的。

## 配置语法

对比前面的 [CRUD 基本流程](#)，`jdonframework.xml` 中的对应 CRUD 配置语法解释如下：

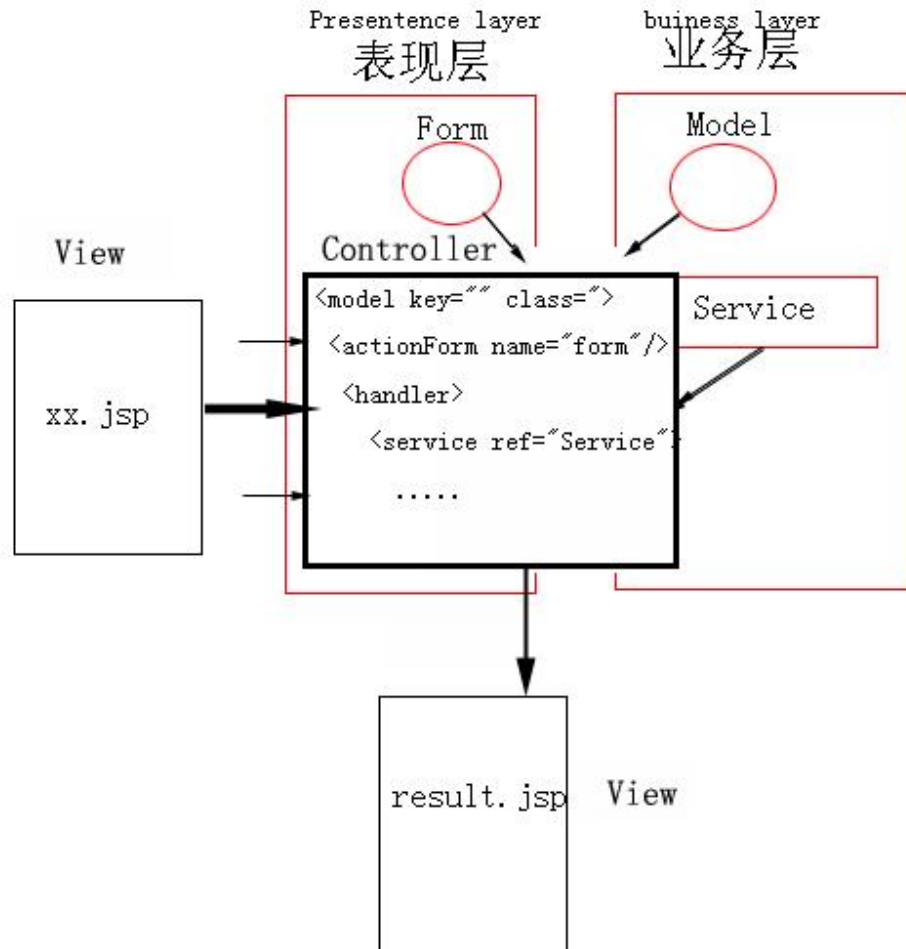
```
<handler>
  <service ref="accountService">
    <initMethod name="XXXX"/> //推出新增页面前对 Model 的初始化方法，CRUD 可选参数
    <getMethod name="XXXX"/> //推出编辑页面前对存在的 Model 进行查询
    <createMethod name=" XXXX "/> //新增页面提交后保存新增数据
    <updateMethod name=" XXXX "/> // 编辑页面提交后保存编辑数据
    <deleteMethod name=" XXXX "/> // 删除数据
  </service>
</handler>
```

`jdonframework.xml` 中这段配置其实对应 CRUD 流程两个部分：`initMethod` 和 `getMethod` 用于 [CRUD 页面显示](#)；而 `createMethod/ updateMethod/ deleteMethod` 用于 [接受 CRUD 处理](#)。说白了，`initMethod/ getMethod` 是在 `ModelViewAction` 中被调用；而 `createMethod/ updateMethod/ deleteMethod` 则是在 `ModelSaveAction` 中被调用。

除 `initMethod` 是实现 CRUD 非必须参数配置外，其他都是实现 CRUD 必须参数。

通过 `jdonframework.xml` 中 CRUD 这段配置，实际是将你的业务层 Service 接口相应的 CRUD 方法告诉 Jdon 框架，这样应用系统运行时，客户端通过调用 URL 参数，如 <http://localhost/xx.do?action=edit> 就可以直接调用你的业务层 Service 相应的 CRUD 方法，省却了表现层琐碎的编码，不只是对 CRUD 这几个方法，通过服务调用命令模式还可以对你自己定义的任何调用直接调用相应的业务层 Service 方法。见后面章节介绍。

CRUD 流程 MVC 配置在架构中位置：



## 服务 Service 编码约束

现在使用 Jdon 框架 CRUD 功能只剩下最后一步了，前面都是讲述如何配置，本节是涉及编码代码方面了。前面说过：

通过 jdonframework.xml 中 CRUD 配置，实际是将你的业务层 Service 接口相应的 CRUD 方法告诉 Jdon 框架，你的 Service 方法名告诉 Jdon 框架了，那么 Jdon 框架怎么知道你的方法参数呢？这两者 Jdon 框架都知晓了才能在运行时自动调用你指定的 Service 方法啊！

服务 Service 编码约束有两条：

方法参数类型约束

第一条：jdonframework.xml 中 CRUD 配置中涉及的所有方法除 `getMethod` 以外方法参数类型都必须是 `EventModel`。

也就是说，在 `initMethod/createMethod/ updateMethod/ deleteMethod` 中配置的 Service 方法参数都必须是 `EventModel` 类型。

```

<handler>
  <service ref="accountService">
    <!-- 对应的方法必须为    initXxx(EventModel em) -->
    <initMethod name="initXxx"/>
    <!--对应的方法必须为    getXxx(String yyyy) -->
    <getMethod name=" getXxx "/>
    <!--对应的方法必须为    void createXxx(EventModel em) -->
    <createMethod name=" createXxx "/>
    <!--对应的方法必须为    void updateXxx(EventModel em) -->
    <updateMethod name=" updateXxx "/>
    <!--对应的方法必须为    void deleteXxx (EventModel em) -->
    <deleteMethod name=" deleteXxx "/>
  </service>
</handler>

```

对应的 service 接口代码如下，注意方法参数类型：

```

public interface AccountService {
    Account initXxx(EventModel em); //被配置成<initMethod name=" getXxx "/>
    Account getXxx(String username); //被配置成<getMethod name=" getXxx "/>
    void createXxx (EventModel em); //被配置成<createMethod name=" createXxx "/>
    void updateXxx (EventModel em); //被配置成<createMethod name=" updateXxx "/>
    void deleteXxx (EventModel em); //被配置成<createMethod name=" deleteXxx "/>
}

```

上个章节的[配置实例说明](#)正好也说明了这个限制。

关于方法返回参数方面，Jdon 框架是这样假设的：

对于 initMethod/ getMethod 方法，返回参数是当前 Model；如：

```
Account getXxx(EventModel em);
```

getXxx 返回的参数 Account 为当前 Model，方法参数是 EventModel，EventModel 实际是 Model 的再次封装，这在下节会专门讲述。

对于 createMethod/ updateMethod/ deleteMethod 新增删除修改方法，方法返回类型为 void，如果方法运行时有出错信息需要返回表现层，通过将错误放入 EventModel 的 setErrors 方法中返回表现层，具体见[出错信息处理](#)章节。

getMethod 方法参数约束

注：JdonFramework 6.0 以上版本已经没有这个限制约束了。

getMethod 方法参数不是 EventModel，必须是 String 类型；方法参数值变量名没有限制。具体规定是：

String 类型 + Model 主键的值

如前面案例 getUser 方法在在 AccountService 中是这样规定编写的：

```
Account getUser(String username);
```

如果你的 Model 主键类型不是使用 String，比如是 Long/long 或 Integer/int，这里方法参数也必须是 String，然后自己的 Service 实现子类中，进行类型转换，当然如果你的 Model 主键都统一成 String 则最好了，推荐！见：

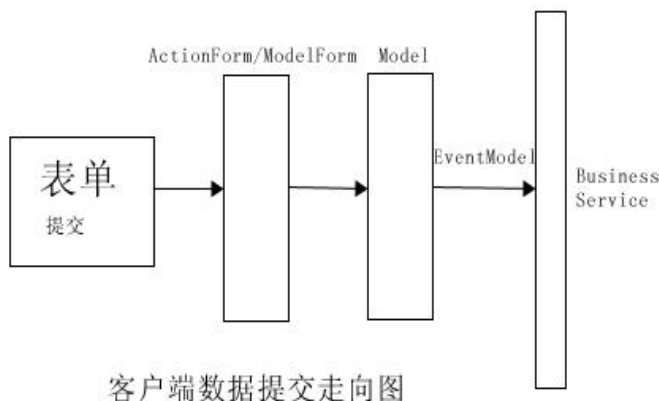
<http://www.jdon.com/jive/article.jsp?forum=91&thread=23875>

注意，以上两条限制是对你的业务层 Service 代码的限制。也就是说，为了使用 Jdon 框架的 CRUD 功能，最后一步必须按照本限制编制你的 Service 代码。当然，这个限制只是对方法参数有限制，其他都是自由的，如果你觉得这点也不方便，可以阅读下章节：

[深入定制](#)，则就完全没有限制了。

## EventModel 类

com.jdon.controller.events. EventModel 是用于表现层向业务层传递参数，相当于 DTO (Data Transfer Object J2EE 模式)，EventModel 中装载的是 Model 实例，一般 Model 实例是来自于 ModelForm 的拷贝，而 ModelForm 则来自客户端提交的表单参数，如下图：



上图展示了客户端浏览器提交的表单数据被服务器端接受后，使用 Jdon 框架处理的流程，最终被打包成 EventModel 作为 Service 方法参数传入业务层，那么我们实现业务层 Service 代码时，就需要从 EventModel 中将 Model 再取出来，如：

```
public class AccountServiceImp implements AccountService{
    .....
    public void createUser(EventModel em) {
        //取出 Model，而且确证是 Account 这个 Model。
        Account account = (Account) em.getModel();
        .....
    }
    .....
}
```

为什么上面代码取出 Model 后确证这个 Model 类型就是 Account，这和 jdonframework.xml CRUD 配置有关，

```
<model key="username" class ="com.jdon.framework.samples.jpeteststore.domain.Account">
    <actionForm name="accountForm"/>
    <handler>
        <service ref="accountService">
            <getMethod name="XXXX"/>
            <!-- 对应上面代码 AccountServiceImp 的 createUser 方法 -->
            <createMethod name=" createUser "/>
            <updateMethod name=" XXXX "/>
            <deleteMethod name=" XXXX "/>
        </service>
    </handler>
</model>
```

上面是 jdonframework.xml CRUD 完整标准配置，其中黑体部分 **com.jdon.framework.samples.jpeteststore.domain.Account** 已经表明当前 Model 的类型，

在这个 Model 配置中，有<createMethod name=" createUser "/>一行，这样我们在 AccountServiceImp 代码中的 createUser 方法中取出的 Model 可以肯定是 Account 了。

如果 createMethod/ updateMethod/ deleteMethod 方法运行时有出错信息需要返回表现层，通过将错误放入 EventModel 的 setErrors 方法中返回表现层，具体见[出错信息处理](#)章节。

需要注意的是：一定要搞清楚 EventModel 的流程来源，上图演示了 EventModel 的来源和流程，在 jdonframework.xml 中除了 getMethod 方法不使用 EventModel 作为方法参数外，其余都是使用 EventModel 作为方法参数：

```
<!-- 对应的方法必须为    initXxx(EventModel em) -->
<initMethod name="initXxx"/>
<!--对应的方法必须为    getXxx(String yyyy) -->
<getMethod name=" getXxx "/>
<!--对应的方法必须为    void createXxx(EventModel em) -->
<createMethod name=" createXxx "/>
<!--对应的方法必须为    void updateXxx(EventModel em) -->
<updateMethod name=" updateXxx "/>
<!--对应的方法必须为    void deleteXxx (EventModel em) -->
<deleteMethod name=" deleteXxx "/>
```

我们已经知道：initMethod 是用于初始化创建页面；getMethod 是用于初始化编辑页面；这两个方法是 ModelAndViewAction 调用的；createMethod/ updateMethod/ deleteMethod 则是作为接受提交页面保存处理，是 ModelSaveAction 调用的。

initMethod 方法中的 EventModel 中的 Model 是从表现层 ActionForm (ModelForm) 来的，而 ActionForm 则是从客户端调用 URL 来的，因为页面初始化工作在 initMethod 业务层方法中实现，所以这对客户端调用 URL 有一些要求，具体可见“JiveJdon3.0 源码中”的发帖功能实现，它的 URL 是这样形式：

/message/messageAction.shtml?forum.forumId=xxxx

注意，是 forum.forumId 而不是通常的直接以 forumId=xxxx 为参数形式，这样，forumId 参数值就被 messageAction 对应的 ActionForm: MessagForm 中的 Forum 的 forumId 接受，并且复制到 ForumMessage 这个 Model，最后在业务层 initMethod 中使用下列语句取出这个 forumId 参数值：

```
Long forumId = forumMessage.getForum().getForumId();
```

另外还有一个小的 Struts 技术使用细节，这时 MessagForm 这个 ActionForm 需要对其中 Forum 这个字段对象进行初始化，可在 MessagForm 构造方法中写入：

```
public MessageForm(){
    forum = new Forum(); // for parameter forum.forumId=xxx
}
```

## 深入定制

至前面章节，Jdon 框架的 CRUD 架构基本完成，可以在一般应用中使用，如果你特殊应用场景，或者上述配置可能无法满足你的应用要求，可以阅读本章节进一步深入定制，同时也会对 jdonframework.xml 中 CRUD 配置运行原理有深入了解。

简单一般以灵活牺牲为代价，如果你觉得约束太大，那么你可以自己编码实现 ModelHandler，在 ModelHandler 中自己通过代码调用 Service 接口，完全回到你的自由编

码阶段，下面针对配置和编码两种实现比较一下，这是以完成 Jdon 框架中要求的相同功能为前提的。

#### 编辑之前的数据查询

Jdon 框架的 CRUD 流程中，需要推出一个用于编辑的页面，既然是编辑，和新增页面不同的就是，编辑页面中各个字段已经有以前的数据，这样，你可在原来数据上修改。那么在编辑页面推出之前，Jdon 框架需要调用 Service 接口的 查询方法，如上面 Account 的 `getUser(String username)` 方法。

Jdon 框架是通过 ModelHandler 的 `findModelByKey` 方法实现的，两种实现方法：

1. 代码实现，继承 ModelHandler 实现，如果你的 Model 查询不是从数据库获得，如是从 HttpSession 获得的，那么你要继承实现自己的方法 `findModelByKey`，如下代码：

```
public Model findModelByKey(String keyvalue,
                            HttpServletRequest request) throws Exception{
    AccountService accountService =
        (AccountService) WebAppUtil.getService("accountService", request);
    return accountService.getUser(key);
}
```

代码实现后，记住告诉 Jdon 框架你的 ModelHandler 子类，配置 `jdonframework.xml`：

```
<model .....j>
    <handler class="你的 ModelHandler 子类" />
</model>
```

一般情况下，是通过服务层查询数据库获得的，那么涉及到服务层交互，可以采取配置实现。

有人可能奇怪：为什么上面 ModelHandler 的 `findModelByKey` 方法参数 `keyvalue` 一定是 String 型的，难道我的 Model 主键类型一定是 String 吗？

不是，这里 `keyvalue` 是主键数值，不是主键本身，主键数值一般是通过调用 ModelViewAction 的 URL 参数传入的，如下：

`/xxxViewAction.do?action=edit&key=keyvalue`

这样，`keyvalue` 一定是字符串型的，如果你的 Service 的查询方法主键参数不是 String，你可以将 String 型号转为你的主键参数。

2. 配置实现，在 `jdonframework.xml` 配置如下：

```
<model ....>
    <handler>
        <service ref="accountService">
            <getMethod name="getUser" />
            .....
        </service>
    </handler>
</model>
```

那么，要求你的服务层服务有相应的方法如下：

```
public class AccountServiceImp implements AccountService{
    private ProductManager productManager;
    public AccountServiceImp(ProductManager productManager){
        this.productManager = productManager;
    }
}
```

```

public Account getUser(String username) {
    try{
        account = accountDao.getAccount(username);
        account.setCategories(productManager.getCategoryList());
        //other business logic
    }catch(DaoException daoe){
        Debug.logError(" Dao error : " + daoe, module);
    }
    return account;
}
....
}

```

### 数据提交保存

用户在 Jsp 页面填写或修改数据后，将实现保存提交，或实现删除提交，这时 Jsp 页面的 action 是提交给 ModelSaveAction 的，ModelSaveAction 委托 ModelHandler 的 serviceAction 实现结果保存或删除。serviceAction 有两种实现：

1.代码实现，这部分工作主要是实现传递工作，代码比较有规律，一般使用配置实现，如果你需要在提交之前实现一些其他特殊实现，那么使用代码实现如下：

```

public void serviceAction(EventModel em, HttpServletRequest request) throws
    java.lang.Exception {
    try { //从 HttpSession 获得 User，保存到 News 中，这是特殊实现
        User user = (User) ContainerUtil.getUserModelAfterLogin(request);
        if (user != null) {
            News news = (News) em.getModel();
            news.setUser(user);
        }

        NewsManagerLocal newsManager = (NewsManagerLocal) WebAppUtil.
            getEJBService("newsManager", request);

        switch (em.getActionType()) {
            case Event.CREATE:
                newsManager.createNews(em); //递交 service 创建方法
                break;
            case Event.EDIT:
                newsManager.updateNews(em); //递交 service 修改方法
                break;
            case Event.DELETE:
                newsManager.deleteNews(em); //递交 service 删除方法
                break;
        }
    }
    catch (Exception ex) {
        throw new Exception(" serviceAction Error:" + ex);
    }
}

```



```
}
```

还需要配置 `jdframework.xml` 你的 `ModelHandler` 代码实现:

```
<model .....j>
    <handler class="news.web.NewsHandler" />
</model>
```

2.配置实现, 如果在递交服务处理之前没有特殊工作实现, 可使用配置, 如下:

```
<model ....>
    <handler>
        <service ref="accountService">
            <createMethod name=" createNews " />
            <updateMethod name=" updateNews " />
            <deleteMethod name=" deleteNews " />
            .....
        </service>
    </handler>
</model>
```

那么, 要求你的 `Service` 有 `insertAccount` 和 `updateAccount` 或 `deleteAccount` 方法, 而且这三个方法参数必须是 `EventModel` 类型, 当然, 方法名是可以根据你的 `service` 具体方法不同, 在配置中更改, 例如你的 `Service` 有 `insert` 方法, 那么配置就是:

```
<createMethod name="insert" />
```

方法名可以变化, 任意取, 但是方法参数必须是 `EventModel` 类型的。如:

```
public interface AccountService {
    Account initAccount(EventModel em); //初始化
    Account getAccount(String username); //查询获得存在的 Model
    void insertAccount(EventModel em); //新增方法
    void updateAccount(EventModel em); //修改方法
}
```

当然, 如果你使用代码实现, 就没有这些对 `Service` 类的编程规定了。

## 页面初始化

有些应用系统可能要求: 推出新增编辑的页面必须初始化, 实现装入一些其他数据, 例如: 我们新增人员名单的页面, 可能有一个部门下拉菜单的选择, 而这些部门数据是我们事先已经输入进入数据库的, 这时在新增人员页面时, 要装载进来。

推出新增性和编辑性页面的初始化工作是不一样的。我们知道, `Jsp` 页面的初始化其实是 `ActionForm` (`ModelForm`) 的初始化, 我们分别从新增和编辑页面的初始化两条线路说明。

如果你需要给 `ModelForm` 中属性字段简单初始化一些常量, 那么很显然这些在 `ModelForm` 的构造方法中实现, 如:

```
public class AccountForm extends ModelForm{
    private List languages;
    public AccountForm() {
        languages = new ArrayList();
        languages.add("english");
        languages.add("japanese");
    }
}
```

上面代码需要给 `languages` 加入一些初始值, 这样, 推出页面时, 用户可以进行多

项选择（使用 html 的 select/options multibox 等实现）。

如果初始化这些属性值需要从数据库中获得，那么就需要和服务层实现交互了，那么就要考虑继承实现 ModelHandler 了。

ModelForm 创建可由通过服务层后台交互实现，这种交互实现也有两种方式：

1. 通过代码实现：ModelHandler 提供的 initForm 方法，initForm 程序员只要继承 ModelHandler，在 initForm 方法中调用服务层服务，获得初始化值，再赋值入 ModelForm，如下代码：

```
public class NewsHandler extends ModelHandler {

    public ModelForm initForm(HttpServletRequest request) throws Exception {
        Debug.logVerbose("enter iniForm .", module);
        NewsActionForm nf = new NewsActionForm(); //创建 ModelForm
        NewsManagerLocal newsManager = (NewsManagerLocal)
            WebAppUtil.getEJBService("newsManager", request);
        PageIterator pi = newsManager.getNewsTypePage(0, 50);
        Collection newsTypes = new ArrayList();
        while (pi.hasNext()) {
            String id = (String) pi.next();
            NewsType newsType = newsManager.getNewsType(id);
            newsTypes.add(newsType);
        }
        nf.setNewsTypes(newsTypes); //将新闻类型列表赋值到新闻这个 ModelForm
        return nf;
    }
    .....
}
```

还需要配置一下，告诉 Jdon 框架你的 ModelHandler 实现：

```
<model .....>
    <handler class="news.web.NewsHandler" />
</model>
```

以上是代码实现直接初始化 ModelForm，还有一种初始化 ModelForm，思路是：初始化 Model，然后将 Model 值拷贝到 ModelForm 中，这是通过 ModelHandler 的 initModel 实现的。不过 initModel 和 initForm 只能选择其一实现，initModel 还可以通过配置实现：

2. 通过配置实现，配置主要是通过 initModel 方法实现，在 jdonframework.xml 中配置如下：

```
<model ....>
    <handler>
        <service ref="accountService">
            <initMethod name="initAccount" />
            .....
        </service>
    </handler>
</model>
```

上面配置 initMethod 方法是访问 accountService 的 initAccount 方法，也就是说，ModelForm 的初始化推给 Model 初始化，而 Model 初始化则由 accountService 的 initAccount 实现，程序员必须实现的 accountService 的 initAccount 方法。如：

```

public class AccountServiceImp implements AccountService{
    private ProductManager productManager;
    public AccountServiceImp(ProductManager productManager){
        this.productManager = productManager;
    }

    public Account initAccount(EventModel em){
        Account account = new Account();
        account.setCategories(productManager.getCategoryList());
        return account;
    }
    ....
}

```

这两种页面初始化选用依据, 是根据用来初始化的数据来自何处? 是来自后台或服务层, 那么选择第二个方案; 如果来自 request 相关的例如 HttpSession, 则选用第一个方案。

上述代码和配置比较中, 案例情况不一样, 实际上, 如果你将代码实现中 initForm 方法中代码移植到 Service 接口的 initAccount 方法中实现, 也就可以采取配置实现了。

## ModelHandler 原理

ModelHandler 几个重要方法是总结了表现层和服务层三种交互操作, 如果你的应用不属于这三种交互操作, 那么可能就无法使用 Jdon 框架的 CRUD 功能, 直接使用 Struts 实现。

ModelHandler 有几个重要方法是 : initForm 方法、initModel 方法、findModelByKey 方法和 serviceAction 方法。前面两个方法是和页面初始化有关, 页面初始化有可能和服务层交互访问; findModelByKey 是根据主键查询数据库存在的记录, 这是和编辑页面相关, 编辑之前需要先查询, 这个方法也需要和服务层交互访问; serviceAction 则是将用户对数据的新增修改删除决定通知服务层进一步处理, 这是和服务层主要交互访问。

上面章节中, jdonframework.xml 配置:

```

<handler>
    <service ref="accountService">
        <getMethod name="getUser"/> //getUser 对应 AccountService 接口的 getUser 方法
        <createMethod name="createUser"/> //createUser 对应 AccountService 接口 createUser 方法
        <updateMethod name="updateUser"/> // updateUser 对应 AccountService 接口 updateUser 方法
        <deleteMethod name="deleteUser"/> // deleteUser 对应 AccountService 接口 deleteUser 方法
    </service>
</handler>

```

在 Jdon 框架中是自动调用 com.jdon.model.handler.XmlModelhandler 作为代码实现, 也就是说, XmlModelhandler 根据上面配置自动生成前面的代码, 属于一种简单的代码自动生成。

注意: 有时我们可能需要代码配置混合实现, ModelHandler 几个方法中只代码实现一个方法, 其他都可以配置实现, 这也是可以的, 只是这时你的代码不是继承 ModelHandler, 而是 XmlModelHandler 了, 同时配置要写明你的子类实现:

```

<model ....>

```

```

    <handler class="你的 XmlModelHandler 子类">
        <service ref="accountService">
            <createMethod name="insertAccount" />
            <updateMethod name="updateAccount" />
            <deleteMethod name="deleteAccount" />

            .....
        </service>
    </handler>
</model>

```

## 更多配置技巧

### CRUD 功能实现标准配置

参考前面的 ModelViewAction 和 ModelSaveAction 配置章节，一个数据 Model 的标准 CRUD 功能实现无需编写任何 Action，只要有 ActionForm（还是 Model 的影子），通过 struts-config.xml 配置就可以实现：

#### 1. 推出创建性或编辑性页面：

```

<action name="aActionForm" path="/aAction" type="com.jdon.strutsutil.ModelViewAction">
    <forward name="create" path="/a.jsp" />
    <forward name="edit" path="/a.jsp" />
</action>

```

约束：

- （1）type 必须是 com.jdon.strutsutil.ModelViewAction
- （2）forward 的 name 值必须是 create 或 edit

#### 2. 接受数据提交数据并递交 Service 服务层处理。

```

<action name="aActionForm" path="/aSaveAction" type="com.jdon.strutsutil.ModelSaveAction">
    <forward name="success" path="/aResult.jsp" />
    <forward name="failure" path="/aResult.jsp" />
</action>

```

约束：

type 必须是 com.jdon.strutsutil.ModelSaveAction  
forward 的 name 值必须是 success 或 failure

### CRUD 功能实现分离配置

CRUD 可以通过两行 action 配置 ModelViewAction ModelSaveAction 合并完成，也可以将新增和修改分开配置，例如用户注册功能，用户注册（用户创建）和用户资料修改属于不同的需要分离开的两个过程，用户注册功能是针对新用户，或者说是非注册用户；而用户资料修改是针对注册用户，两者服务对象角色是不一样的，因此，不能象其他 Model 那样将 CRUD 合并在一起。

#### 1. 推出创建性页面的配置：

```

<action name="accountForm" path="/shop/newAccountForm"
        type="com.jdon.strutsutil.ModelViewAction" scope="session">
    <forward name="create" path="/account/NewAccountForm.jsp" />
</action>

```

将 forward 的 edit 值为空，没有这一行。

## 2. 接受创建数据并并递交 Service 服务层处理

```
<action name="accountForm" path="/shop/newAccount"
        type="com.jdon.strutsutil.ModelSaveAction" scope="session"
        validate="true" input="/account/NewAccountForm.jsp">
    <forward name="success" path="/shop/index.shtml" />
    <forward name="failure" path="/shop/newAccountForm.shtml" />
</action>
```

## 3. 推出编辑性页面的配置

```
<action name="accountForm" path="/shop/editAccountForm"
        type="com.jdon.strutsutil.ModelViewAction" scope="session">
    <forward name="edit" path="/account/EditAccountForm.jsp" />
</action>
```

将 forward 的 create 值为空，没有这一行，不过调用/shop/editAccountForm.shtml 形式还是必须要有参数的：/shop/editAccountForm.shtml?action=edit&username=XXX

## 4. 接受修改后的数据并并递交 Service 服务层处理

与第 2 条类似，不同的是 path 和 jsp 因为编辑页面不一样而不同：

```
<action name="accountForm" path="/shop/editAccount"
        type="com.jdon.strutsutil.ModelSaveAction" scope="session"
        validate="true" input="/account/EditAccountForm.jsp">
    <forward name="failure" path="/shop/editAccountForm.shtml" />
    <forward name="success" path="/shop/index.shtml" />
</action>
```

## 单纯输出 Jsp 页面

如果有时只是为了输出一个 Jsp 页面而做一个 Action 比较麻烦，Jdon 框架实现了一个缺省的转发 Action，可以简单重用在很多这样场合。只要在 struts-config.xml 配置如下：

```
<action path="/XXX" type="com.jdon.strutsutil.ForwardAction"
name="XXX" scope="request" validate="false">
    <forward name="forward" path="/xxx.jsp"/>
</action>
```

上述配置有三点要求如下：

type 必须是 com.jdon.strutsutil.ForwardAction

forward 的 name 值是 forward

当然，你也可以自己编制这样缺省简单 action。

## 服务命令调用模式

多层结构的好处是带来了良好的可维护性，可拓展性，但是也带来开发的复杂性，有时我们为实现一个小功能，需要做如下步骤：

创建 Domain Model;

创建界面模型，如 struts 的 ActionForm;

创建 Struts 的 Action，作为界面控制器;

配置界面流程 struts-config.xml

创建 Domain Model 的 Service 接口

创建 Domain Model 的持久化 DAO

对于复杂功能，上述过程是必须的，这样可以达到很好的分层解耦作用。但是，有时一个小功能，或者是 Domain Model 的特殊功能，只是在 Service 增加了一个小方法，是否还要整个流程都走一遍呢？

使用 Jdon 框架就不必了，Jdon 框架提供基于 URL 调用参数的服务命令调用模式，就象当初我们在 C/S 结构中，一个菜单按钮功能直接激活其功能方法一样。

Jdon 框架提供的服务命令调用模式有几种实现方式，可根据功能复杂程度灵活选择，并且提供最简单直接的缺省实现。

我们先看看 Jdon 框架提供的服务命令调用模式是什么样？如下图：



上图调用 url 形式可以有两种：/aaa.do?action=xxxxx 或/aaa.do?method=xxxxx 使用 action 或 method 都可以，推荐使用 method。

当浏览器客户端调用某个 Struts 的 Action，如 serviceAction.do 时，跟随的参数 action 或 method 值如果为 xxxxx，那么通过 Jdon 框架，就可以直接调用对应 Service 接口的方法名为 xxxxx 的方法，如果 xxxxx 是 abcde；那么就调用 service 的 abcde 方法。

如果结合 url-rewrite 框架，那么我们就可以写成 <http://website.com/aaa/xxxxx>，通过配置 urlrewrite.xml 转为/aaa.do?action=xxxxx，如下：

```
<rule>  
    <from>^/ ([0-9]+)/ ([0-9]+)$</from>  
    <to type="permanent-redirect">/$1\do?method=$2</to>  
</rule>
```

这样，我们就可以通过 URL 直接访问 Service 方法，很有 REST 的风格。

这种直接将客户端命令和后台业务层服务直接对应的关系可以大大简化我们很多小功能开发，从而达到类似 Ruby on Rails 那样的快速开发。

下面我们提供三种不同的实现方式，根据功能复杂程度进行选用，注本章节功能在 Jdon 框架 1.4 版本后可使用：

### 最简单实现

最简单实现是指命令没有任何输入参数，也就是无需 Model 或 ModelForm 的命令直接调用，调用形式如下：

<http://localhost:8080/myAction.do?service=xxx&method=xxx>

这个调用方式适合无返回值的命令调用。

其中，service 值就是 jdonframework.xml 中定义的 pojoService 名称，如：

```
<pojoService name="testService" class="com.jdon.framework.test.service.TestServicePOJOImp"/>
```

那么调用参数中 service 值就是 testService。

Method 名称为 testService 代表的那个接口的方法，如下：

```
public interface TestService{
    .....
    void xxxxx();//注意方法是无返回值 无输入参数 这是两个约束条件
    .....
}
```

上述 TestService 接口方法为 xxxxx，调用参数中的 method 值就是 xxxxx

调用 URL 中的 myAction.do 是在 struts-config.xml 中配置的，如下：

```
<action path="/myAction" type="com.jdon.strutsutil.ServiceMethodSimpleAction"
    scope="request" validate="false">
    <forward name="success" path="/result.jsp"/>
</action>
```

其中 com.jdon.strutsutil.ServiceMethodSimpleAction 是 JdonFramework 1.5 版本以后提供的一个缺省 Struts Action 实现，注意，Action 配置中需要配置一个名称为 success 的 forward 页面，表示成功后跳转页面，也可不配置。

最后，通过浏览器网址 URL 直接调用 TestService 接口的 xxxxx 方法为：

<http://localhost:8080/myAction.do?service=testService&method=xxxxx>

是不是非常简单？您无需书写任何表现层的代码如 Jsp 或 Servlet 或 Action，就可以通过浏览器客户端直接调用你的业务层服务方法，这个简单方式经常使用在系统调试维护或一些简单的功能实现上。

以上实现的缺点就是：你无法向 TestService 的方法 xxxxx 输送方法参数，因此你的业务层服务方法必须是没有方法参数的，如果必须输送方法参数，如何实现？

有两个解决方案：

1. 不写 Action 代码，将这些参数组合成一个 Model 和相应 ModelForm，如果参数简单，就可以组合到一个现成的 Model 中。无论多复杂情况，大多数情况可以采取这种方式。
2. 自己写代码 Action，就无需组合 Model 了。

下面先讨论自己写代码 Action 方式，再下一章再讨论不写 Action 代码的方式：

## Action 代码定制实现

你自己必须实现一个 Struts 的 DispatchAction 实现，DispatchAction 的特殊之处是可以根据 method 方法参数直接进入其方法的。struts-config.xml 配置如下：

```
<action path="/aaa" type="你的 DispatchAction 子类"
        scope="request" validate="false" parameter="method">
    <forward name="xxxxx" path="/xxx.jsp"/>
</action>
```

注意这里配置多了一个 parameter="method"，那么你的浏览器客户端参数调用必须符合 Struts 1.2 中这个新功能规定：

/aaa.do?method=xxxxx

其中 xxxxx 对应上面配置 parameter 的值，这样，struts 就直接调用你的 DispatchAction 子类的 xxxxx 方法，如下：

```
public class MyAction extends DispatchAction{
    .....
    //按照 struts 的 dispatchAction 规则，方法名必须为 xxxxx
    public ActionForward xxxxx (ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) throws Exception {

        //手工代码调用 Service 接口

        TestService testService = (TestService) WebAppUtil.getService(" testService ", request);
        testService. xxxxx(aa, bb)
    }
    .....
}
```

在 MyAction 中使用代码访问 Service 接口，调用 WebAppUtil 的 callService 方法即可，可返回任何对象，callService 调用参数说明如下：

服务名称：在 jdonframework.xml 中 services 定义的名称；如 testService，在 jdonframework.xml 中是这样定义的：

```
<pojoService name="testService" class="com.jdon.framework.test.service.TestServicePOJOImp"/>
```

方法名称：需要调用服务的方法名称，也就是我们的调用参数 xxxxx；

方法参数：是 Object[] 数组形式，将要传给 testService 的 xxxxx 参数值，如有两组参数 aa 和 bb，则将 aa 和 bb 参数打包入数组 Object[]。

按照这个调用要求，你的代码 testService 如下：

```
public interface TestService{
    .....
    void xxxxx(AType aa, BType bb);//这是有方法参数的 注意方法是无返回值的
    .....
}
```

以上方法适合有一些个别方法参数时，自己必须做一个 Action，将这些 URL 调用参数转换为 TestService 的方法参数。



这种方式好处，就是无需建立 Model/ModelForm，但是需要写 Action 代码，有时还要自己控制缓存的清除，缓存清除需要在 Action 代码中如下调用：

```
modelManager.removeCache(keyValue); //keyValue 为 Model 主键
```

具体可参考 ModelSaveAction 的 makeModel 方法。

还有一种更简单，无需写 Action 的方式，只要将上述参数 aa 和 bb 组合成一个 Model 就可以了，请看下一章：

不写 Action 代码方式

这是推荐大家使用的方式，无论多复杂的情况，都可以通过搞清头绪，根据参数建立好 Model，或者把参数组合到一个 Model 中，然后建立相应 ModelForm 就可以了。这样就无需写 Action 调用：

<http://localhost:8080/aaa.do?service=testService&method=xxxxx>

上面一行意思是：调用 testService 的方法 xxxxx。当然，没有返回值。就是一个简单的没有返回值的命令。

下面讨论已经建立了专门的 Model 或 ModelForm 后，服务命令调用如何简单实现？调用 URL 将变成：

<http://localhost:8080/aaa.do?method=xxxxx>

注意，其中没有指定 testService 了，因为我们已经配置，详细步骤如下：

假设你的 Model 已经建立完成，相应的 ModelForm 是 aForm：

第一步：配置 struts-config.xml 如下：

```
<action path="/aaa" type="com.jdon.strutsutil.ServiceMethodAction"
        name="aForm" scope="request" validate="false">
    <forward name="xxxxx" path="/xxx.jsp"/>
</action>
```

其中 aForm 是一个 ModelForm 的名称，必须在 struts-config.xml 中配置。

这个配置主要使用 Jdon 框架的一个现成 Action 子类：com.jdon.strutsutil.ServiceMethodAction，配置其中 forward 配置中(<forward name="xxxxx" path="/xxx.jsp"/>)的 name 值必须是参数 method 的值 xxxxx，这样，调用 Service 处理后，可返回对应的 Jsp 页面。

第二步：指定 Model 和服务的调用关系，配置 Jdon 框架的配置文件 jdonframework.xml，告诉系统，你所要调用的 Service 接口是哪个类。如下：

```
<model key="primary key of your model 主键" class="你的 model 类">
    <actionForm name="aForm"/>
    <handler>
        <service ref="testService" /> <!-- 指定对应的接口 Service 名称 -->
    </handler>
</model>

<pojoService name="testService" class="com.jdon.framework.test.service.TestServicePOJOImp"/>
```

第三步，最后就是你自己要编码实现 Service 接口和子类，接口中必须有和调用参数值 xxxxx 相同的方法名，方法参数必须是 com.jdon.controller.events.EventModel（使用下节代码定制无此限制）；在 EventModel 中可封装你的传输参数 Model 等，方法返回类型可以返回值或者为空，返回值应该是 Model 类型，返回值 Model 中的值将被复制到对应的 ModelForm，也就是 aForm 中，这里假设缺省是空 void，如下：

```
public interface TestService{
    .....
    Model  xxxxx(EventModel em);//注意，方法参数必须统一为是 EventModel 可有返回值
    .....
}
```

全部完成，这样你就可以使用下面 URL 直接调用，不必写 Action 代码：

<http://localhost:8080/aaa.do?method=xxxxx>

上面这一行中没有指定 testService，是因为我们已经在 jdonframework.xml 中定义了 model 配置。

虽然 xxxxx(EventModel em)方法没有返回值，但是你可以在这个方法实现中，对 EventModel.setErrors()设置错误，以告诉界面命令是否成功执行。

无论多复杂的情况，都可以通过搞清头绪，根据参数建立好 Model，或者把参数组合到一个 Model 中，然后建立相应 ModelForm 就可以了。也就实现开始的图示：

### 浏览器客户端参数调用



### 业务层 Service 接口

```
public interface TestService {
    ....
    public void xxxxx(EventModel em);
    ....
}
```

使用服务命令调用模式，可以节省 Struts 的 Action 代码书写，少写 Action 代码的最大好处就是：可以避免误将业务逻辑写到 Action 中，造成真正业务层空壳，因此推荐尽量使用服务命令调用模式。

下面举一个实际应用例子来说明如何启发服务命令调用模式使用，例如：用户注册功能中有一个忘记密码功能，用户通过 Jsp 提交给一个 Action，由这个 Action 转发给某个 Service，由于忘记密码功能不属于专门的 CRUD 功能，而且是个单个功能，如果我们为这些单个功能都写一个 Action 或 DispatchAction，增加系统代码的复杂性，更容易导致不懂 MVC 模式的人或不小心中将业务写入这些 Action，造成代码复杂难于维护。

其实，JdonFramework 的 CRUD 流程简化就是依据本章基本原理实现新增 删除修

改查询等服务命令的简单直接调用的。

最普遍的用法

我们已经看到，在 CRUD 流程中，最后需要保存新增或修改结果时，提交给 ModelSaveAction 的参数 action/mehtod 值非常重要，action 参数来决定整个流程是新增保存还是修改保存或者是删除，action/mehtod 的参数值是固定的，有三个 create、edit 或 delete，这是 Jdon 框架固定的，必须是这三个字符中一个。那么，参数除了这三个以外，是否可以其他值呢？或者我们是否一定取名 action/method 值为 create edit 和 delete 这样的字符呢？

有时，我们向后台提交保存结果可能不是新增、编辑或删除这么简单，可能是两者性质的删除，在我们的 Service 中有两个 delete 方法，那么如何使得前台提交到这两个 delete 方法呢？一个使用 Jdon 框架的 CRUD 方式，另外一个或更多的方式如何？

很简单，直接使用服务命令调用模式即可，例如向 ModelSaveAction 提交时，action 值为 deleteAll；那么只要你的 Service 接口中 deleteAll 方法即可；

从这里看出，提交是 Action 或 method 的值也不一定是框架规定的 create 、edit 或 delete 这些字符，可以直接是你 service 的任何方法，包括 CRUD 方法。

步骤如下： Struts-config.xml 正常配置 ModelSaveAction

```
<action path="/aaa" type="com.jdon.strutsutil. ModelSaveAction"
        name="aForm" scope="request" validate="false">
    <forward name="sucess" path="/xxx.jsp"/>
</action>
```

jdonframework.xml，告诉系统，你所要调用的 Service 接口是哪个类。如下：

```
<model key="primary key of your model 主键" class ="你的 model 类">
    <actionForm name="aForm"/>
    <handler>
        <service ref="testService" /> <!-- 指定对应的接口 Service 名称 -->
    </handler>
</model>
<pojoService name="testService" class="com.jdon.framework.test.service.TestServicePOJOImp"/>
TestService 接口代码
```

```
public interface TestService{
    .....
    void xxxxx(EventModel em);//注意，方法参数必须统一为是 EventModel
    .....
}
```

那么我们就可以进行/aaa.do?method= xxxxx 的直接调用，也无需指定 service 名称了，提交的内容是在 aForm 和其对应的"你的 model 类"中。

xxxxx 可以是 createModel 方法 或 updateModel 或 deleteModel 这些方法。

当前方式是 Jdon 框架使用最简单直接通用的方式。

上述展示了 CRUD 中 CUD 直接调用方式，关于 Read 查询，则配置 struts-config.xml 的 ModelSaveAction 为 ModelSaveAction 即可。

## 出错信息处理

在服务层一旦实现数据库操作出错，Jdon 框架通过 EventModel 的 setErrors 方法向表现层传递出错信息，因此，只要在你的服务层 Service 实现方法的出错代码调用该方法即可，如：

```
public void insertOrder(EventModel em) {  
    Order order = (Order)em.getModel();  
    try{  
        orderDao.insertOrder(order);  
    }catch(Exception daoe){  
        Debug.logError(" Dao error : " + daoe, module);  
        em.setErrors("db.error");  
    }  
}
```

当存储 orderDao 调用出错，将 db.error 保存到 EventModel 中，而 db.error 是 struts 的 Application.properties 中定义的，，需要在你的应用系统中定义，Jdon 框架应用到的信息如下：

```
id.required = you must input id  
id.notfound = sorry, not found  
db.error=sorry, database operation failure!  
system.error=sorry, the operation failure cause of the system errors.  
maxLengthExceeded=The maximum upload length has been exceeded by the client.  
notImage=this is not Image.
```

主要有 id.required id.notfound db.error 等几个，需要在 J2EE 应用系统中定义。

## 案例：UserTest 建模

本文是建立模型 UserTest 的 CRUD 和批量查询功能，本文实现功能其实和上篇是一样的，操作过程也一样，只是描述方式不一样，上篇是一种概要大体步骤描述，是依据 CRUD 和批量查询两个功能实现展开的，而本文是依据开发中业务层/表现层/持久层为主题展开的。本部分代码见 SimpleJdonFrameworkTest 源码包。

开始开发程序， 建立 Model 实现 UserTest

建立 Model 对应的 ModelForm: UserActionForm 是继承 ModelForm，而 ModelForm 实际也是继承 Struts 的 ActionForm。

将 UserTest 中方法和字段拷贝到 UserActionForm，原则上这两者内容是一致的，JdonFramework 将自动在这两个对象之间拷贝相同方法的值，但是它们服务的对象不一样，UserTest 是为了中间层或后台业务层服务的；而 UserActionForm 是为了界面服务的，两者分离是为了使得我们的业务层和界面分离，这样，当有新的可替代 Struts 界面框架出现，我们可以不修改业务层代码下更换界面框架。

业务层开发

Jdon 框架支持 Web 应用和 EJB 应用，下面两种方案中任意选一种，也就是说，是采取 POJO 服务实现或 EJB 服务实现取决你的系统架构，如果你不想使用 EJB，那么就

采取 POJO 服务，相关 EJB 服务实现及其他就都不要看了。

### POJO 服务实现

POJO 服务只要编写一个 TestService 接口和一个实现 TestServicePOJOImp 即可，在实现子类中调用持久层 jdbcDao。

### EJB 服务实现

这里使用 EJB 实现业务层功能，建立一个无态 Session Beans

为 TestEJB，主要负责 UserTest 的数据库增删改查等功能，这部分开发不在 JdonFramework 规定之类，而要遵循 EJB 相关规定。这部分开发细节可参见有关 JBuilder 开发 EJB 教程。

建立一个实体 Bean 名为 User：首先需要将 JBuilder 配置成能够直接访问数据库，我们使用 MySQL 数据库，配置后，重新启动 JBuilder。

配置 JBoss 中的数据源，数据源 JNDI 名称是 DefaultDS。在代码中调用 JBoss 的 JNDI，JBoss 有特定规定，应该是 java:/DefaultDS。

创建 CMP 时有两种方式：一种直接创建新的 CMP，当该 J2EE 部署时，将自动创建 CMP 对应的数据表，或者先创建数据表，由数据表导入 CMP。本演示采取后者方式。

完成 Session Bean TestEJB 的新增、修改和删除方法，下面完成查询和批量查询方法，查询方法建议使用 DAO+JDBC 实现。

持久层：增删改查 CRUD 实现

持久层的增删改查功能可使用 Jdon 框架的 Jdbc 模板实现：

新增操作

```
public void insert(UserTest userTest) throws Exception{
    String sql = "INSERT INTO testuser (userId , name) "+
        "VALUES (?, ?)";
    List queryParams = new ArrayList();
    queryParams.add(userTest.getUserId());
    queryParams.add(userTest.getName());
    jdbcTemp.operate(queryParams,sql);
    clearCacheOfItem();
}
```

修改操作：

```
public void update(UserTest userTest) throws Exception{
    String sql = "update testuser set name=? where userId=?";
    List queryParams = new ArrayList();
    queryParams.add(userTest.getName());
    queryParams.add(userTest.getUserId());
    jdbcTemp.operate(queryParams,sql);
    clearCacheOfItem();
}
```

删除操作：

```
public void delete(UserTest userTest) throws Exception{
    String sql = "delete from testuser where userId=?";
    List queryParams = new ArrayList();
```

```

queryParams.add(userTest.getUserId());
jdbcTemp.operate(queryParams,sql);
clearCacheOfItem();
}

```

查询操作:

```

public UserTest getUser(String Id) {
    String GET_FIELD = "select * from testuser where userId = ?";
    List queryParams = new ArrayList();
    queryParams.add(Id);

    UserTest ret = null;

    try {
        List list = pageIteratorSolverOfUser.queryMultiObject(queryParams,
            GET_FIELD);
        Iterator iter = list.iterator();
        if (iter.hasNext()) {
            Map map = (Map) iter.next();
            ret = new UserTest();
            ret.setName((String) map.get("name"));
            ret.setUserId((String) map.get("userId"));
        }
    } catch (Exception se) {
    }
    return ret;
}

```

增删改查 CRUD 配置

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE app PUBLIC "-//JDON//DTD Framework 2005 1.0 //EN"
"http://www.jdon.com/jdonframework.dtd">
<app>
    <models>
        <model key="userId" class="com.jdon.framework.test.model.UserTest">
            <actionForm name="userActionForm"/>
            <handler>
                <service ref="testService">
                    <getMethod name="getUser" />
                    <createMethod name="createUser" />
                    <updateMethod name="updateUser" />
                    <deleteMethod name="deleteUser" />
                </service>
            </handler>
        </model>
    </models>
    ..... <!-- 服 务 配 置 -->
</app>

```

注意 Handler 配置有两种: 缺省是上述配置, 如果情景复杂, 需要代码完成, 则编

写一个 Hamdler 类继承 ModelHandler 实现即可，然后在上述 handler 段配置如下：

```
<handler class="xxxx.yourHandler" />
```

### POJO 服务配置实现

```
<services>
  <pojoService name="testService" class="com.jdon.framework.test.service.TestServicePOJOImp"/>
  <pojoService class="com.jdon.framework.test.dao.JdbcDAO" name="jdbcDAO">
    <constructor value="java:/DefaultDS"/>
  </pojoService>
</services>
```

上面定义了一个 POJO 服务：com.jdon.framework.test.service.TestServicePOJOImp，因为 TestServicePOJOImp 中使用了 JdbcDAO，这里配置了 com.jdon.framework.test.dao.JdbcDAO 实现，如果你使用其他持久层技术，可以在此更换另外一个 JdbcDao。

这里的 jdbcDAO 需要数据库连接，数据库连接参数不是在应用程序中配置，而是使用 J2EE 服务器已经配置成功的 JNDI 名称，Tomcat 中配置 JNDI 数据库连接池可见：<http://www.7880.com/Info/Article-37f05fa0.html>；JBoss 的 JND 配置见本站相关文章。

特别注意：

java:/DefaultDS 是 JBoss 的数据库连接池 JNDI 写法，如果你使用其他服务器，参考他们的 JNDI 要求写法。

例如 在 JBoss 的 mysql-ds.xml 中配置一个数据库的 JNDI 为 TestDS。

那么在 J2EE 系统中(上面 JdbcDAO 配置中)，就应该前面加一个 java:/ 就成了 java:/TestDS。

如果 Tomcat 的 server.xml 配置数据库的 JNDI 成 TestDS

那么在 J2EE 中(上面 JdbcDAO 配置中)，使用该 JNDI 的名称应为：java:comp/env/TestDS。

两者写法不一样，不同的服务器写法都有规定。也可以在 web.xml 配置 JNDI 引用。

如果服务配置中存在 ejbService，如下：

```
<ejbService name="testService2" >
  <jndi name="TestEJB" />
  <ejbLocalObject class="com.jdon.framework.test.ejb.TestEJBLocal"/>
  <interface class="com.jdon.framework.test.service.TestService" />
</ejbService>
```

那么可以在 ejbService 和.pojoService 之间更换，将.pojoService 和 ejbService 的 name 值对换一下即可，将 <ejbService name="testService2" > 更换为 <ejbService name="testService" >，将 <pojoService name="testService" 改名为 <pojoService name="testService2"，这样，Model 部分配置的服务指向就从调用 POJO 服务更换到调用 EJB 服务了。

### 界面流程配置

CRUD 功能无需编制代码，代码功能已经在上节的 jdonframework.xml 中配置完成，这里只要配置 struts 的页面流程 struts-config.xml 配置和编写 Jsp 两步即可。

(1) 按照前面章节规则二：配置 `ModelViewAction` 介绍的。实现新增或编辑视图输出的 Action:

```
<action name="userActionForm" path="/userAction"
type="com.jdon.strutsutil.ModelViewAction">
    <forward name="create" path="/user.jsp" />
    <forward name="edit" path="/user.jsp" />
</action>
```

上述配置中，需要修改的跟 `Model` 名称相关的配置。

(2) 创建 `user.jsp` 页面。

在 `user.jsp` 中要增加一行关键语句：

`<html:hidden property="action"/>` 用来标识新增 修改等动作。

(3) 配置 `userSaveAction` 实现数据真正操作。

```
<action name="userActionForm" path="/userSaveAction"
        type="com.jdon.strutsutil.ModelSaveAction">
    <forward name="success" path="/result.jsp" />
    <forward name="failure" path="/result.jsp" />
</action>
```

(4) 创建数据操作结果页面 `result.jsp`，上述配置修改不大，需要修改的跟 `Model` 名称相关的配置。



# 批量分页查询快速开发

Jdon 框架的批量分页查询功能不但适合开发大批量集合数据显示，也适合一对多数据显示，例如一个对象中包含子对象集合，这个对象可以称为父对象，那么如果需要显示这个父对象和其子对象集合，使用本章节介绍的批量查询也可快速实现。

## 原理

所谓批量分页查询是指对大量数据进行分页查询，实现如下图显示效果：

| 当前有15 页 [ 1 2 3 4 5 6 ▶ ] |    |                |             |  |
|---------------------------|----|----------------|-------------|--|
| 主题名                       | 回复 | 作者             |             |  |
| ● 有关“抽象类”和“接口”            | 1  | bybsky         | 2003年08月02日 |  |
| ● 我在毕设时写的设计模式等            | 3  | wwlhp@jdon.com | 2003年08月01日 |  |
| ● 首届Java优秀译者奖励计           | 0  | rckc           | 2003年08月01日 |  |
| ● 关于value object          | 7  | SunOne         | 2003年07月31日 |  |
| ● 请问在web.xml设置的role-      | 4  | ajoke          | 2003年07月26日 |  |
| ● 综合使用抽象工厂、工厂方            | 15 | manbaum        | 2003年07月25日 |  |

批量查询由于是频繁操作，对系统性能设计要求很高，而且批量查询几乎存在每个数据库信息系统，因此，这个功能具有可重用性，Jdon 框架根据 Jive 等传统成熟系统批量查询设计原理，总结多个应用系统，在不破坏多层情况下，抽象出批量查询子框架。

批量查询设计两个宗旨：

尽量减少数据库访问，减少数据库负载和 I/O 性能损耗。

由于 J2EE 是一个多层结构，尽量减少在多个层次之间传送的数据量。减少传送损耗。

因此，批量查询设计将涉及到两个层面：表现层和持久层，Jdon 框架提供了持久层下 JDBC 查询操作模板 (JdbcDao)，这个 JdbcDao 可以被服务层不同服务类型 (POJO Service 或 Session Bean) 调用。

根据批量查询设计宗旨，有下面实现要点：

使用缓存减少数据库访问

持久层不能将满足查询条件的每页所有完整数据记录传送到表现层，试想一下，如果一条数据记录有 1000 个字节，那么一页显示 20 条，那么就有 2 万个字节传送，目前很多批量查询设计都是这样做，存在浪费内存和性能消耗大的隐患。

缓存越靠近用户界面端，性能越好，因此，持久层如果只传送满足查询条件的数据记录主键（一个字段）集合，那么表现层可以优先从缓存中根据主键获得该数据完整数据，查询越频繁使用，缓存命中率越高，各方面消耗就越小。

根据以上设计原则，Jdon 框架的批量查询设计方案如下：

对于持久层：

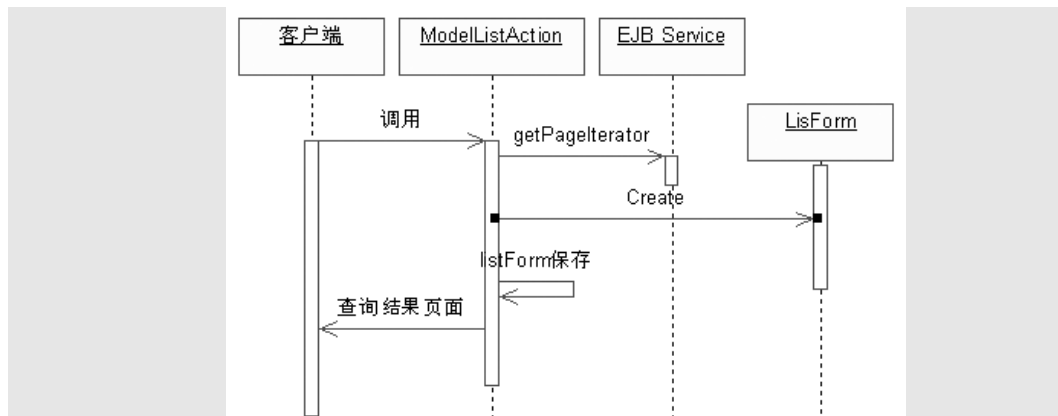
- 获取满足查询条件的所有数据表记录总个数。
- 获取当前页面所有数据记录主键集合 (ID 集合)。
- 将上述两种数据打包成一个对象 (PageIterator) 传送到前台

对于表现层：

- 获得后台满足查询条件的 PageIterator 对象。

- 遍历 PageIterator 对象中的主键集合，根据主键再查询后台获取完整数据对象 (Model)，先查询缓冲，如果没有，再访问数据库。
- 将获取的完整数据对象 (Model) 封装成一个集合，打包成 ModelListForm 对象。
- Jsp 页面再展开 ModelListForm 对象中 Model 数据集合，逐条显示出来。

批量查询的主要示意图：



图中解释：后台将满足查询条件的数据记录的主键集合 (ID 集合) 包装成 PageIterator 对象，然后由服务层返回到表现层 (通过 ModelListAction 调用服务 EJBService 或 POJO Service 的 getPageIterator 方法)，表现层的 ModelListAction 再遍历这个 ID 集合，根据每个 ID 查询获得完整 Model 数据，再打包到 ModelListForm 对象中。

更详细的实现方案可参考《Java 实用系统开发指南》一书。

批量查询框架基于一个前提是：如果你的 Model 没有主键怎么办？那么使用强制给它一个 Object ID，这可以由一个专门序列器产生。这与前面 CRUD 实现的前提是一致的。

## 实现约束

为利用 Jdon 框架提供的批量查询高性能功能，基本有以下步骤：

第一步，建立 Model，建立对于列表中每行视为一个 Model，如下图：

当前有15页 [1 2 3 4 5 6 ▶]

| 主题名                | 回复 | 作者             | 时间         |
|--------------------|----|----------------|------------|
| 有关“抽象类”和“接口”       | 1  | bybsky         | 2003年08月02 |
| 我在毕设时写的设计模式书       | 3  | wwlhp@jdon.com | 2003年08月01 |
| 首届Java优秀译者奖励计      | 0  | rckc           | 2003年08月01 |
| 关于value object     | 7  | SunOne         | 2003年07月31 |
| 请问在web.xml设置的role- | 4  | ajoke          | 2003年07月26 |
| 综合使用抽象工厂、工厂方法      | 15 | manbaum        | 2003年07月25 |

上图中每一行实际是一个 Model 对象，多行显示，实际就是多个 Model 对象显示，表结构“主题名”、“回复”“作者”“时间”则是 Model 对象的字段。

Model 的建立和上章节 CRUD 中 Model 规则是一样的，每个 Model 有一个主键，而且 Model 主键类型必须和数据表的主键数据类型是对应的，详细可参考上章节 CRUD 的“Model 和 ModelForm”中“每个 Model 需要一个主键”。

第二步，在表现层，继承 com.jdon.strutsutil.ModelListAction 实现子类，然后将该子类配置进入 struts 的 struts-config.xml，供客户端浏览器调用。细节见本章节后面说明。

第三步，在持久层，创建一个 `PageIterator` 对象，然后，将这个 `PageIterator` 对象传送到表现层供 `ModelListAction` 的子类 `getPageIterator` 方法调用。

## 重要对象 `PageIterator`

从前面批量查询原理中看出，`PageIterator` 实际是一个在多层之间穿梭的数据载体，前后台关系只是靠 `PageIterator` 来实现消息交互。

因此，无论后台持久层使用什么技术（JDBC、实体 Bean、Hibernate 或 iBatis），只要能提供一个查询结果对象 `PageIterator` 给前台就可以。

考虑到持久层技术发展极快，Jdon 框架并没有在持久层的缺省实现，但提供了 JDBC 实现的帮助 API（见包：com.jdon.model.query），具体可参考 Jdon 框架源码包 Samples 目录下 JdonNews 的 news.ejb.dao. JdbcDao。这个 JdbcDao 虽然为 EJB Session Bean 调用，也可以为普通的 POJO 服务调用，JdbcDao 开发也有模板化，一般是一个 Model 完成三个方法，具体可见“[开发一个简单 Jdon 应用系统](#)”章节的“[批量查询持久层实现](#)”。

在 Jdon-Jpetstore 中，批量查询持久层实现是使用 iBatis，然后在服务层将 iBatis 的 `PaginatedList` 类转换成 Jdon 框架的 `PageIterator` 即可。

### `PageIterator` 定义

`PageIterator` 是在持久层（Dao 类）或业务层创建的、为显示每个页面数据服务的临时对象，只要我们在持久层向表现层传递一个 `PageIterator`，前台表现层只要使用特定 Jsp 标签，也就能够出现分页标识。

其实，`PageIterator` 就是 `java.util.ListIterator` 一个子类实现，比 `ListIterator` 多了结束点和遍历个数两个定义。

使用 `PageIterator` 后，不但前台表现层代码基本无需编写，最重要的是，在后台持久层将对批量查询数据进行缓存优化，极大提高性能，这将在[下节 PageIteratorSolver](#)讲述。

从持久层产生的 `PageIterator` 主要包含符合查询条件的数据集合（是数据记录的主键集合，而非全部数据记录，可以更快地从数据库获取）；还有显示时每页开始的第一个数据记录的指针，这个指针是数据集合的指针；当然最后还有显示时每页结束的最后最后一个数据记录的指针。

注意：也就是说：`PageIterator` 包含的数据集合个数总是大于等于当前页需要显示的数据个数，当前页面显示的数据个数是根据客户端 url 调用决定的，例如 `/userListAction.do?count=20`，那么表示每页显示的数据个数是 20。当然，也并不是说，这个个数完全按客户端查询条件决定的，如果恶意查询，count 数值很大，则会对系统存在伤害。这里有一个最大值，缺省是 200，如果你需要改变，在 [PageIteratorSolver](#) 中进行修改即可。

下面介绍 `PageIterator` 缺省构造方法，第一个构造器是 Block 构造器，是缺省的构造器，如果你使用 [PageIteratorSolver](#) 来获得 `PageIterator`，框架内部就是使用这种构造器来自动构造 `PageIterator`。

第一种构造器（Block `PageIterator`）：

```
public PageIterator(int allCount, Object[] keys, int startIndex, int endIndex, int count)
```

缺省构造方法有 5 个参数，分别详细描述如下：

int allCount: 符合查询条件的所有数据记录（Model）总数。

Object[] keys: 这是一个 Block 集合，所谓 Block 就是一整块记录集合，缺省 Block 长度是 200 个，也就是说，这长 200 的一个 Block 中一定包含符合查询条件的当前页面数据记录（Model）的 ID 集合，只要在具体显示时根据 startIndex 等值从中遍历就可以。这也决定了该 PageIterator 是 Block PageIterator 性质。

int startIndex: 当前页面在 keys 中的开始点。

int endIndex: 当前页面在 keys 中的结束点。

int actualCount: 当前页面实际可显示个数，当查询条件 count 小于每页显示最大值 200 时，actualCount 其实就是 count，否则，就是 200。

第二中构造器（Page PageIterator）：

```
public PageIterator(int allCount, Object[] keys, int startIndex) {
    this.allCount = allCount;
    this.keys = keys;
    this.endIndex = keys.length;
}
```

该构造器两个参数意义如下：

int allCount: 符合查询条件的所有数据记录（Model）总数。

Object[] keys: 符合查询条件的当前页面数据记录（Model）的 ID 集合，这个 ID 集合长度一定和当前页面需要显示长度是等长，不能多也不能少，这点和上面第一个缺省构造器区别。

这种构造器称为 Page PageIterator，是因为 keys 的意义和前面 Block PageIterator 不一样，keys 参数就是符合查询条件当前页面主键 ID 集合，有多少符合查询条件，这个 keys 中就放置多少，没有多余的，不是一个长 200 的 Block 集合。

第二个构造器有两个为兼容老版本的衍生构造器，多了两个参数 int startIndex 和 boolean hasNext，这两个参在 1.4 以后版本已经丢弃不用了。

如果你使用 Hibernate 或 Ibatis 或自己实现数据库查询，然后使用得到的 ID 集合手工构造一个 PageIterator，可以使用这个构造器。如 Jpestore 的 OrderServiceImp 中：

```
public PageIterator getOrdersByUsername(String username, int start, int count) {
    PageIterator pageIterator = null;
    try {
        //查询数据库从 start 开始共 count 个记录的主键集合
        List list = orderDao.getOrderIDsByUsername(username, start, count);
        int allCount = orderDao.getOrderIDsByUsernameCount(username);
        int currentCount = start + list.size();
        //下面是使用第二个构造器(老版本)自己构造一个 PageIterator
        pageIterator = new PageIterator(allCount, list.toArray(), start,
            (currentCount < allCount)?true:false);
    } catch (Exception daoex) {
        Debug.logError(" Dao error : " + daoex, module);
    }

    return pageIterator;
}
```

## 自己构造 PageIterator

如果持久层不使用 Jdon 框架，或者通过别的数据来源产生一个数据集合，如果希望在页面能够使用 Jdon 框架的自动分页查询和性能优化，那就在业务层 Service 中自己构造一个 PageIterator，如下代码：

```
List resultSortedIDs = xxxDao.getHotThreadKeys(qc);//从数据源获得一个 ID 集合
if (resultSortedIDs.size() > 0){
    List pageIds = new ArrayList(resultSortedIDs.size());
    for (int i = start; i < start + count; i++) {
        if (i < resultSortedIDs.size()){
            pageIds.add(resultSortedIDs.get(i));
        }else
            break;
    }
    //返回自己构造的 PageIterator
    return new PageIterator(resultSortedIDs.size(), pageIds.toArray());
}
```

## 遍历 PageIterator

有时我们会根据后台持久层的 PageIterator 进行查询符合一定条件的数据，或者要将所有数据记录都获得，也就是前台页面不需要分页，但是后台持久层还使用 Jdon 框架的 PageIterator 查询，这时，我们可以如下实现：

```
int start = 0;
int count = 30;//每页 30 个元素
boolean found = false;
PageIterator pi = forumMessageQueryService.getMessage(threadId, start, count);
int allCount = pi.getAllCount();
while((start < allCount) && (!found)){//遍历所有的元素 直至满足寻找条件
    while(pi.hasNext()){
        Long messageIdT = (Long)pi.next();
        if (messageIdT.longValue() == messageIdL.longValue()){
            found = true;
            break;
        }
    }
    if (found) break;
    //继续遍历
    start = start + count ;
    logger.debug("start = " + start + " count = " + count);
    //获得下一页
    pi = forumMessageQueryService.getMessage(threadId, start, count);
}
```

## 构造 Model 类型 PageIterator

前面介绍的是构造 PageIterator 时，装载的 Model 主键集合，Jdon 框架的 ModelListAction 会遍历 PageIterator 的主键集合，根据主键再查询获得整个 Model，如果有时不能这样分两次前后查询，例如后台是 Lucene 这样搜索持久层，每次一次性获得一个 Model 完整集合，如果先获得 Model 主键集合，再逐个根据主键获得整个 Model，效率很差。

在这种情况下，我们可以直接将 Model 集合构造成 PageIterator，如下：

```
new PageIterator(resultSortedIDs.size(), modelCollection.toArray());
```

注意，需要调用 PageIterator.setElementsIsKey 为 false，这样 Jdon 框架的 ModelListAction 会识别 PageIterator 中装载的是完整 Model。

此功能适合 Jdon 框架 5.0 版本。

## 持久层 PageIteratorSolver

PageIterator 可以由开发者自己根据前台用户界面需求自己创建，当然 Jdon 框架也提供一般情况下的 PageIterator 的自动创建，开发者只需从 PageIteratorSolver 对象直接获取即可，那么 PageIteratorSolver 如何使用？

Jdon 框架提供了 PageIterator 创建的持久层创建 API：com.jdon.model.query.PageIteratorSolver，这是一个基于 SQL 的 JDBC 实现，当然你完全可以其他持久层技术实现创建 PageIterator。

使用 Jdon 框架的 PageIteratorSolver 提供 JDBC 模板操作，在查询功能上几乎无需写 JDBC 语句。使用 PageIteratorSolver 可以书写很少代码就实现一种 Model 的批量分页查询功能，PageIteratorSolver 可以被 Session Bean 调用，或者作为普通 POJO 被调用，因此，PageIteratorSolver 使用有两种情况：

配置在 Jdon 框架容器中使用，如果你不使用 EJB。

单独直接使用。

这两种使用方式和 PageIteratorSolver 提供两种构造器有关：

```
public PageIteratorSolver(DataSource dataSource, CacheManager cacheManager)
```

```
public PageIteratorSolver(DataSource dataSource)
```

第一种构造器需要指定 CacheManager，CacheManager 已经在 Jdon 框架启动时作为基础组件激活，你只要创建 DataSource 即可，而 DataSource 是和容器的数据源 JNDI 名称相关，这里给出一种方式以供参考：

第一步：创建一个 JdbcTempSource 作为 DataSource 的包装器，JdbcTempSource 构造参数如下：

```
public JdbcTempSource(String jndiname) {
    try {
        ServiceLocator sl = new ServiceLocator();
        dataSource = (DataSource) sl.getDataSource(jndiname);
        jdbcTemp = new JdbcTemp(dataSource);
    } catch (ServiceLocatorException slx) {
```

```

        logger.error(slx);
    }
}

```

在 jdonframework.xml 中配置如下：

```

<pojoService name="jdbcTempSource"
    class="com.jdon.jivejdon.dao.sql.JdbcTempSource">
    <constructor value=" java:/NewsDS "/>
</pojoService>

```

其中 java:/NewsDS 是 Jboss 的数据源在 server/default/deploy/mysql-ds.xml 中配置的 JNDI 名称。如果你使用 Tomcat 或 weblogic 其他，根据它们数据源 JNDI 说明在这里填写。

第二步，创建你自己的 DAO 类，构造参数如下：

```

public MessageDaoSql(JdbcTempSource jdbcTempSource, CacheManager cacheManager) {
    this.pageIteratorSolver =
        new PageIteratorSolver(jdbcTempSource.getDataSource(),CacheManager cacheManager);
}

```

假设 MessageDaoSql 是你自己的 Dao 类，那么我们在 jdonframework.xml 配置如下：

```

<pojoService name="messageDao"
    class="com.jdon.jivejdon.dao.sql.MessageDaoSql "/>

```

因为 CacheManager 事先已经在 Jdon 框架的 container.xml 中配置过了。

通过以上配置，由于 Jdon 框架的 autowiring 功能，为自动为你的 Dao 对象创建匹配其他的类，这样在你的 Dao 类中可以正常使用 PageIteratorSolver。

下面谈谈第二中构造器的使用，直接 new 就可以创建，适合在 EJB 这样 Jdon 框架体外调用，如下：

```

ServiceLocator sl = new ServiceLocator(); //Jdon 框架中的 ServiceLocator
DataSource dataSource = (DataSource) sl.getDataSource("java:/NewsDS");
PageIteratorSolver pageIteratorSolverOfType = new PageIteratorSolver(dataSource);

```

因为这种方式下，PageIteratorSolver 自己创建一个新缓存器，而不是象上面那样可以使用 Jdon 框架的统一缓存系统，这带来的问题是：你要自己 hold 住你的 PageIteratorSolver 对象，也就是将 PageIteratorSolver 对象引用保存在第三方处，以保证每次客户端请求时，都能找到上次操作的 PageIteratorSolver 对象，千万不能每次请求访问时，创建一个新 PageIteratorSolver 对象，那要造成内存泄漏。

建议，你可以为每个 Model 建立一个对应的 PageIteratorSolver 对象，这样，该 Model 更新时，只要刷新这个 Model 相关的缓存，具体可见下面缓存清除方法。

是不是觉得自己管理缓存非常麻烦？所以还是推荐你使用第一种 PageIteratorSolver 的构造器构造 PageIteratorSolver 对象。

有了 PageIteratorSolver 对象，我们就可以从 PageIteratorSolver 对象中获取我们最重要的分页对象 PageIterator 了。

### 获得 PageIterator

首先，根据批量查询原理，首先要查询符合查询条件的记录总数和当前页面的所有 Model 的 ID 集合，这是通过 PageIterator 的创建实现。

PageIteratorSolver 中重要方法 getPageIterator 是为了创建一个 PageIterator 对象,需要的输入参数相当比较复杂,这里详细说明一下:

getPageIterator 方法如下:

```
public PageIterator getPageIterator(String sqlqueryAllCount,  
                                   String sqlquery, String queryParam, int start, int count) throws Exception
```

注意: PageIteratorSolver 的 getDatas 与 getPageIterator 实则一样, getPageIterator 名称或参数都显得易懂正规一些。但是更要注意的是: getDatas 和 getPageIterator 方法参数顺序是不一样的,特别是第一个参数,虽然都是 String,但是意义不一样,使用时切勿混淆。

```
public PageIterator getDatas(String queryParam, String sqlqueryAllCount,  
                             String sqlquery, int start,  
                             int count) throws Exception
```

首先从 String sqlqueryAllCount 参数解释:

sqlqueryAllCount 参数其实是一个 sql 语句,该 sql 语句用来实现查询满足条件的所有记录的总数。例如查询某个表 myTable 的 sql 语句如下:

```
String sqlqueryAllCount = "select count(1) from myTable ";
```

当然这是一个标准 sql 语句,后面可写入 where 等查询条件

sqlquery 参数则是关键的查询语句,主要实现批量查询中查询符合条件的数据记录 (Model) 的主键 ID 集合。例如表 myTable 的主键是 id, 则

```
String sqlquery = "select id from T_myTable where categoryId = ?"
```

重要规定:

getPageIterator 方法两个重要参数就是上面的 sqlqueryAllCount 和 sqlquery 两条 SQL 语句,批量查询的事先对这两条 SQL 语句返回值有规定: sqlqueryAllCount 中返回的必须是满足查询条件的总数(count(1)); sqlquery 必须是满足查询条件的所有主键集合(id)。

queryParam 参数则是和 sqlquery 参数相关,如果 sqlquery 语句中有查询条件 "?",如上句 categoryId = ?中?的值就是 queryParam 参数,这样 sqlquery 和 queryParam 联合起来形成一个查询语句。如果没有查询条件,那么 queryParam 就赋值为 null 或 ""。

注意,如果查询条件有多个 "?",那么就需要将这些 queryParam 打包成一个 Collection,然后调用方法:

```
public PageIterator getPageIterator(String sqlqueryAllCount,  
                                   String sqlquery, Collection queryParams, int start, int count) throws Exception
```

这个方法与前面的 getDatas 区别就是方法参数由 String queryParam 变成集合 queryParams, queryParams 是多个查询条件 "?" 的集合,注意,查询条件目前 Jdon 框架支持常用的几种查询参数类型: String Integer Float 或 Long,例如,如果你的查询条件如下:

```
String sqlquery = select id from T_myTable where categoryId = ? and name = ?
```

这里有两个查询参数,那么将这两个查询参数打包到数组中即可,注意顺序和问号的顺序是一样,如果问号对应的变量 categoryIdValue 和 nameValue,则如下:

```
Collection queryParams = new ArrayList();  
queryParams.add(categoryIdValue);  
queryParams.add(nameValue);
```

参数 start 和 count 则是有表现层传入的, start 表示当前页面是从所有符合查询条件记录中第几个开始的,而 count 则是从 start 开始查询多少条记录个数,也就是一页显示



的记录数目。

获取数据块

在 `PageIteratorSolver` 内部是如何创建 `PageIterator` 呢？实际是根据数据块创建 `PageIterator` 的，`PageIteratorSolver` 中 `getPageIterator` 方法代码如下：

```
public PageIterator getPageIterator(String sqlqueryAllCount,
                                   String sqlquery, Collection queryParams,
                                   int startIndex, int count) {
    //every query max length must be less than blockLength
    if ((count > this.blockLength) || (count <= 0)) {
        count = this.blockLength;
    }
    Block currentBlock = getBlock(sqlquery, queryParams, startIndex, count); //关键
    startIndex = currentBlock.getStart();
    int endIndex = startIndex + currentBlock.getCount();
    Object[] keys = currentBlock.getList().toArray();
    int allCount = getDatasAllCount(queryParams, sqlqueryAllCount);
    if (endIndex < startIndex){
        return new PageIterator();
    }else{
        return new PageIterator(allCount, keys, startIndex, endIndex, count);
    }
}
```

其中 `getBlock` 方法是关键，那么数据块是怎样的概念呢？

1.4 版本以前批量查询时总是根据查询条件指定的范围从数据库获得特定范围，但是不同查询条件可能获得不同的数据结果范围，而且两者结果集合中可能会重复，这样，缓存的利用效率就不是很高。

1.4 版本以后更改数据库查询方式，不是根据查询条件指定范围查询，而是以固定范围查询，如每次读取都是读取 200 个数据，然后在内存中通过逻辑检查，客户端所要查询的指定范围是否在这 200 数据块中。

数据块都将被缓存起来，如果每页显示 20 个，这样这一个包含 200 个数据的数据块至少适合 10 页查询，进一步降低翻页对数据库的操作次数。

有关数据块的两个方法可供外界调用，以便在复杂查询中，更大程度利用缓存中的数据块：

```
public Block getBlock(String sqlquery, Collection queryParams, int startIndex, int count)
```

```
public Block locate(String sqlquery, Collection queryParams, Object locateId)
```

前者 `getBlock` 是根据查询条件获得一个 `Block` 对象，然后由程序员自己加工再自行创建 `PageIterator`，提供了更大的灵活性。

后者 `locate` 是在符合查询条件的所有 `Block` 中寻找包含主键 `locateId` 值的那个 `Block`，需要注意的是，在调用该 `locateId` 方法之前，你必须首先根据 `locateId` 直接查询一下数据库或缓存，确定存在 `locateId` 这个值，否则 `locate` 翻遍数据库或缓存都没有发现，浪费性能。

这两个方法都包含激活缓存的操作，也就是说，第一次调用 `locate` 可能需要真正操作数据库，但是第二次则会从缓存中获得，而且通过 `locate` 的操作，缓存可能有益于 `getBlock` 或 `getPageIterator` 方法，反过来也然。

有关数据块设计部分细节可见: [http://www.jdon.com/articheck/oo\\_math.htm](http://www.jdon.com/articheck/oo_math.htm)

## 批量查询缓存清除

由于 PageIteratorSolver 内置了缓存, 缓存两种情况:

符合查询条件的所有记录总数, 这个总数第一次使用后将被缓存, 不必每次都执行 select count 的数据操作, 这是很耗费数据性能的。

当前页面中所有 Model 的 ID 集合, 在没有新的 Model 被新增或删除情况下, 这个 ID 集合几乎是不变的, 因此其结果在第一次被使用后将被缓存。但是, 某个 Model 发生新增或删除情况下, 我们要主要清除这些缓存, 否则新增或删除的 Model 将不会出现在 Model 列表中, PageIteratorSolver 缓存设计前提是 Model 的新增和删除不会很频繁, 至少没有查询频繁。

PageIteratorSolver 的 clearCache() 提供这两种缓存的主动清除, 是符合查询条件的主键数据块集合。注意, 清除的不是具体一个 Model, 这是另外需要清除方式, 见 [Model 缓存使用](#)。

需要在 Model 的新增方法或删除方法中加入调用 PageIteratorSolver 的 clearCache 方法的代码。因为 Model 的新增或删除方法是由程序员自己实现, 如使用 CMP 或 Hibernate 等实现, 也需要加入 clearCache 方法。

我们建议 PageIteratorSolver 创建依据每个 Model 类型有一个对应的 PageIteratorSolver 对象, 这样, 这个 Model 类型变动只会清除它的有关缓存; 如果所有 Model 类型共用一个 PageIteratorSolver 对象, 一旦一个 Model 对象变动将引起所有缓存清除, 降低缓存效益。

## 其他形式 PageIterator 创建

上面是使用 Jdon 框架的 JDBC 模板实现 PageIterator 创建, 你有可能使用其他技术实现持久层, 这里提供两种 PageIterator 创建的参考代码:

例如, productDao 是封装了别的持久层技术(iBatis 或 Hibernate)。要求两步就可以完成 PageIterator 创建:

第一步. productDao 返回符合条件的当前页面 ID 集合:

```
List list = productDao.getProductIDsListByCategory(categoryId, start, count);
```

第二步. productDao 返回符合条件的总数:

```
int allCount = productDao.getProductIDsListByCategoryCount(categoryId);
```

创建 PageIterator 代码如下:

```
int currentCount = start + list.size(); // 计算到当前页面已经显示记录总数
PageIterator pageIterator = new PageIterator(allCount, list.toArray(), start,
    (currentCount < allCount)?true:false);
```

以上代码是在服务层实现, 一旦服务层能够返回一个 PageIterator 实例, 就可以按照下面表现层实现完成批量分页查询功能。

另外一个 PageIterator 创建实现, 假设所有数据 ID 或数据模型包装在一个 List 中, 从 List 中创建 PageIterator 的代码如下:

```
// 计算未显示记录个数
int offset = itemList.size() - start;
```

```

int pageCount = (count < offset)?count:offset;
//生成当前页面的 List
List pageList = new ArrayList(pageCount);
//从 start 开始遍历，遍历次数是一页显示个数
//将当前页面要显示的数据取出来放在 pageList 中
for(int i=start; i< pageCount + start;i++){
    pageList.add(itemList.get(i));
}
int allCount = itemList.size();
int currentCount = start + pageCount;
PageIterator pageIterator = new PageIterator(allCount, pageList.toArray(), start,
(currentCount < allCount)?true:false);

```

## 使用 JdbcTemp 实现单个查询

前面已经获得 PageIterator 对象，Jdon 框架会在表现层 ModelListAction 中遍历这个 PageIterator 对象中 ID 集合，然后根据 ID 先从缓存中获取完整 Model 数据，如果没有则从数据库获得，那么我们在持久层还需要提供单个 Model 查询的实现。

Jdon 框架已经内置查询模板 JdbcTemp，无需使用者自己实现 JDBC 语句操作，直接使用模板即可，注意：如果数量很大近百万，推荐使用专门的持久层框架，比如 iBatis 或 Hibernate 等。

以 JdonNews 中 Model : NewsType 查询一个实例为例子：

首先确定 sql 语句写法，如下：

```
String GET_TYPE = "select  * from T_NEWS_TYPE where typeid = ?";
```

其中 typeid 参数值由外界传入，因此写方法如下：

```

public NewsType getNewsType(String Id) throws Exception {
    String GET_TYPE = "select  * from T_NEWS_TYPE where typeid = ?";
    List queryParams = new ArrayList();
    queryParams.add(Id); //Id 是 GET_TYPE sql 语句中的?的值

    ....
}

```

上述代码准备了 JDBC 模板查询两个参数：GET\_TYPE 和 queryParams，

调用 PageIteratorSolver 中的两个 Model 查询方法，其实是调用 JdbcTemp.java：

```

//适合查询返回结果是单个字段，如：
// select name from user where id=?
public Object querySingleObject(Collection queryParams, String sqlquery) throws Exception {
    return jdbcTemp.querySingleObject(queryParams, sqlquery);
}

//适合查询返回结果是多个字段，而且有多项记录
//例如： select id, name, password from user where id = ?
public List queryMultiObject(Collection queryParams, String sqlquery) throws Exception {
    return jdbcTemp.queryMultiObject(queryParams, sqlquery);
}

```

这两个方法适合不同的查询语句，当你的查询 sql 语句返回不只一条数据记录，而且每条数据记录不只是一个字段，而是多个字段，使用 queryMultiObject，

queryMultiObject 经常使用。下面是获得一个 Model 查询的完整写法：

```
public NewsType getNewsType(String Id) throws Exception {
    String GET_TYPE = "select * from T_NEWS_TYPE where typeid = ?";
    List queryParams = new ArrayList();
    queryParams.add(Id);

    NewsType ret = null;
    try {
        List list = pageIteratorSolverOfType.queryMultiObject(queryParams, GET_TYPE);
        Iterator iter = list.iterator();
        if (iter.hasNext()) { //遍历查询结果，根据你的判断，决定这里使用 if 或 while
            Map map = (Map)iter.next(); //List 中每个对象是一个 Map
            ret = new NewsType();
            ret.setTypeNames((String)map.get("typename")); //根据字段名称从 Map 中获取其相应值
            ret.setTypeId(Id);
        }
    }
    catch (Exception se) {
        throw new Exception("SQLException: " + se.getMessage());
    }
    return ret;
}
```

JdbcTemp 还提供了更多数据库操作的简化模板，具体可参考：[JdbcTemp](#)

## 表现层 ModelListForm:

前面章节主要谈论了批量查询的持久层实现，Jdon 框架的批量查询主要在表现层封装了主要设计原理，让我们看看表现层 Struts 的实现步骤：

这里以查询商品类别（Category）下所有商品(Product)列表为需求，演示批量的查询的开发步骤。

使用 Jdon 框架实现批量查询时，无需使用代码创建 ModelForm( ActionForm)，Jdon 框架提供一个批量查询的缺省实现：com.jdon.strutsutil.ModelListForm

只要在 struts-config.xml 中配置这个 ActionForm 就可以：

```
<form-bean name="productListForm" type="com.jdon.strutsutil.ModelListForm"/>
```

习惯地，我们将这个 ActionForm 命名为：

model 名称+ListForm

例如，查询商品列表是显示一个商品 Prodcut 数据，那么这个 ActionForm 的名称就是 productListForm。

也就是说，我们只要在 struts-config.xml 配置 ModelListForm 就可以，然后在 Jsp 页面中使用 logic:iterator 获取就可以，见 [Jsp MultiPages 标签](#)。

ModelListForm 是一个普通的 JavaBeans，主要属性如下：

```
public class ModelListForm extends ActionForm{

    private int allCount = 0; //符合查询条件的所有记录总数
```

```

private int start = 0;    //当前页面的开始
private int count = 20;  //当前页面的可显示的记录数
private boolean hasNextPage = false; //是否有下一页

/**
 * 父 Model
 */
private Model oneModel = null;

/**
 * 批量显示的 Model 集合
 * 这是本类主要的属性
 */
private Collection list = new ArrayList();

.....
}

```

## 抽象类 ModelListAction

`com.jdon.strutsutil.ModelListAction.ModelListAction` 是一个 struts 的标准 Action，它主要实现如下功能：

从服务层获得符合查询条件 `PageIterator`，这是需要通过 `getPageIterator` 方法实现；展开遍历 `PageIterator`，根据 ID，首先从缓存中查询是否有其 `Model`（完整数据记录），如无，则调用服务层从持久层数据库获得，这是需要通过 `findModelByKey` 方法实现。

将获得的 `Model` 集合封装在 `ModelListForm` 的 `list` 字段中。

所以，`ModelListAction` 实际是将 ID 集合换算成 `Model` 集合，然后交给 `ModelListForm`，Jsp 页面的显示只和 `ModelListForm` 有关。

```

public abstract class ModelListAction extends Action {
    private ModelManager modelManager;

    //struts action 的缺省 execute 方法
    public ActionForward execute(ActionMapping actionMapping,
                                ActionForm actionForm,
                                HttpServletRequest request,
                                HttpServletResponse response) throws Exception {
        .....

        //从服务层获得符合查询条件的 PageIterator
        PageIterator pageIterator = getPageIterator(request, start, count);
        if (pageIterator == null) {
            throw new Exception(
                "getPageIterator's result is null, check your ModelListAction subclass");
        }
        //获得 ModelListForm 实例
    }
}

```

```

        ModelListForm listForm = getModelListForm(actionMapping, actionForm,
                                                    request, pageIterator);

        //赋值页面起始数等值到 ModelListForm
        listForm.setStart(start);
        listForm.setCount(count);
        listForm.setAllCount(pageIterator.getAllCount());
        listForm.setHasNextPage(pageIterator.isNextPageAvailable());

        //根据 pageIterator 获得 Model 集合
        Collection c = getModelList(request, pageIterator);
        Debug.logVerbose(" listForm 's property: getList size is " + c.size(), module);
        //将 Model 集合赋值到 ModelListForm 的 list 字段
        listForm.setList(c);
        //设置其他 Model 到 ModelListForm, 以便 jsp 页面能显示其他 Model
        listForm.setOneModel(setupOneModel(request));
        //程序员自己定义的优化 ModelListForm 其他动作, 供实现者自己优化。
        customizeListForm(actionMapping, actionForm, request, listForm);

        .....
    }

```

**ModelListAction** 是一个抽象类, 它必须有一个实现子类需要有两个方法必须实现: 获得服务层的 **Pageiterator**, 也就是 **getPageIterator** 方法  
根据 **key** 或 **ID** 获得单个 **Model** 的方法, **findModelByKey** 方法。

其他可选实现的方法有:

是否激活缓存方法 **protected boolean isEnabledCache()**

有时, 批量查询实现可能不需要缓存, 获得每个 **Model** 都一定要执行 **findModelByKey** 方法。通过覆盖 **isEnabledCache** 方法实现。

批量查询的 **Jsp** 页面, 可能不只是需要显示某一个 **Model** 的很多实例列表; 还可能  
需要其他类型 **Model** 实例显示, 可以覆盖方法:

**protected Model setupOneModel(HttpServletRequest request)**

自己再优化加工 **ModelListForm** 方法, 覆盖方法:

```

public void customizeListForm(ActionMapping actionMapping,
                              ActionForm actionForm,
                              HttpServletRequest request,
                              ModelListForm modelListForm ) throws Exception

```

通过以上方法覆盖实现, 基本使用 **ModelListAction** 可以实现批量查询的各种复杂功能实现, 如果你觉得还不行, 那么干脆自己模仿 **ModelListAction** 自己实现一个标准的 **struts** 的 **Action** 子类, 只要 **ActionForm** 还是 **ModelListForm**, 页面显示还是可以使用 **Jdon** 框架的页面标签库。

使用 **ModelListAction** 可以实现 **master details**, 使用 **setupOneModel** 或 **customizeListForm** 实现 **Master**, **ModelListForm** 则装载的多条显示的 **details**。

**com.jdon.strutsutil.ModelListAction.ModelListAction** 有两个方法需要实现, 共分下面三个步骤:

## getPageIterator 方法

```
public PageIterator getPageIterator(HttpServletRequest request, int start, int count)
```

这是从服务层获得满足查询条件的当前页面所有 Model 的 ID（主键）集合，当前页面是通过 start 和 count 两个变量说明，前者表示从第几个开始；后者表示开始后需要显示多少个 Model。

以查询商品为例子，ProductListAction 继承 ModelListAction，详细方法内容如下：

```
public class ProductListAction extends ModelListAction {
    public PageIterator getPageIterator(HttpServletRequest request, int start,
        int count) {
        //获得一个服务
        ProductManager productManager = (ProductManager) WebAppUtil.getService(
            "productManager", request);
        //从查询参数中获得 categoryId
        String categoryId = request.getParameter("categoryId");
        //调用服务的方法
        return productManager.getProductListByCategory(categoryId);
    }

    public Model findModelByKey(HttpServletRequest request, Object key) {
        .....
    }
}
```

所以，关键在于服务层的服务需要有个返回参数类型是方法，这是对服务层服务方法的约束要求，这个服务方法如何实现在后面章节介绍。

## findModelByKey 方法

这个方法主要也是从服务层获得一个 Model 实例，ProductListAction 的该方法实现如下：

```
public Model findModelByKey(HttpServletRequest request, Object key) {
    //获得一个服务
    ProductManager productManager = (ProductManager) WebAppUtil.getService(
        "productManager", request);
    return productManager.getProduct((String)key);
}
```

非常需要注意的是：需要搞清楚这里获得 Model 实例是哪个？

是商品类别 Category 还是商品 Product？

这主要取决于你的批量查询显示的是什么，如果希望显示的是商品 Product 列表，当前这个案例是需要查询某个商品类别 Category 下的所有商品 Product 列表，那么这里的 Model 类型就是 Product。

搞清楚 Model 类型是 Product 后，那么我们可能对 findModelByKey 方法参数 Object 类型的 key 有些疑问，其实这个 key 就是 com.jdon.controller.model. PageIterator 中重要属性 keys（满足查询条件的 ID 集合）中的一个元素，这里根据 key 查询获得一个完整 Model，所以我们知道，PageIterator 中封装的是 Model 主键，也是数据表的主键，以后需要根据这些 ID 能够获得唯一一个 Model。

以上述案例为例：

```
productManager.getProduct ((String)key);
```

这是将 key 类型 Object 强制转换为 String 类型，因为 PageIterator 中 keys (ID 集合) 都是 String 类型。所以， key 的类型其实是由你后台创建 PageIterator 时决定的。

其他深入定制方法

以上面例子为例，商品批量查询除了显示多个商品 Product 信息外，还需要显示商品目录 Category，也就是说在一个页面中，不但显示多个商品，也要显示商品目录的名称，比如商品目录名是“电器”，其下具体商品列表很多。

在一个页面中装入一个以上 Model 有两种方法：

1. 配置另外一个需要装入 Model 的 ActionForm，例如 Category 对应的 ActionForm 是 CategoryForm，只要在 struts-config.xml 中配置 CategoryForm 的 scope 为 session，同时注意 CategoryForm 在操作流程中要预先赋值。

2. 由于批量查询的 ActionForm 是 ModelListForm，我们可以向这个 ModelListForm 加入一个 Model，实现 ModelListAction 的 customizeListForm 方法，在这个方法中，实现将 Category 加入的语法::

```
public void customizeListForm(ActionMapping actionMapping,
                             ActionForm actionForm,
                             HttpServletRequest request,
                             ModelListForm modelListForm ) throws Exception{
    ModelListForm listForm = (ModelListForm) actionForm;
    ProductManager productManager = (ProductManager) WebAppUtil.getService(
        "productManager", request);
    String categoryId = request.getParameter("categoryId");
    Category category = productManager.getCategory(categoryId);
    listForm.setOneModel(category);
}
```

如果加入的布置一个 Model，那么可以使用 DynamicModel 将那些 Model 装载进来，然后在 Jsp 页面再一个个取出来。

## struts-config.xml 配置

批量查询与 jdonframework.xml 配置无关，也就是说 jdonframework.xml 中没有与批量查询实现相关的配置，简化了工作，这点与 CRUD 实现相反，CRUD 更多的是 jdonframework.xml 配置（当然都少不了 struts-config.xml 配置），CRUD 缺省情况下除了 Model 和 ModelForm 以外可以没有表现层编码，而批量查询主要是表现层编码，由于查询条件多种多样，只有靠更多编码才能实现查询的灵活性。

上例子中，商品查询的 ModelListForm 配置如同一般 ActionForm 一样配置：

```
<form-beans>
  <form-bean name="productListForm" type="com.jdon.strutsutil.ModelListForm"/>
  .....
</form-beans>
```

ProductListAction 配置如同一般 Action 一样：

```
<action-mappings>
  <action path="/shop/viewCategory"
    type="com.jdon.framework.samples.jpstore.presentation.action.ProductListAction"
```



```

        name="productListForm" scope="request"
        validate="false" >
        <forward name="success" path="/catalog/Category.jsp"/>
    </action>
    .....
</action-mappings>

```

剩余最后工作就是 Jsp 标签使用，通过使用 struts 的 logic:iterator 将前面配置的 productListForm 中的 Model 集合遍历出来。遍历标签语法如下：

```

<logic:iterate indexId="i" id="user" name="listForm" property="list" >
    <bean:write name="user" property="userId" />
    <bean:write name="user" property="name" />
</logic:iterate>

```

上述语法将逐个输出每行记录，如果配合 Html 的 table 语法，输出效果如下：

| <div> <div> 前页 1 2 3 4 5 后页 </div> </div> |             |           |            |
|---|-------------|-----------|------------|
| 主题  | 作者人         | 写作单位      | 日期         |
| just a notice                             | Surrry Peng | your team | 2003-11-23 |
| just a notice                             | Surrry Peng | your team | 2003-11-23 |
| just a notice                             | Surrry Peng | your team | 2003-11-23 |

## Jsp MultiPages 标签

批量查询输出还有一个重要功能：页面显示，如上图的“前页”和“后页”显示，这是使用 Jdon 框架的标签库实现的。

只要在 Jsp 页面中你希望显示“前页”和“后页”部位贴入下面代码即可：

```

<% @ taglib uri="/WEB-INF/MultiPages.tld" prefix="MultiPages" %>

.....

<MultiPages:pager actionFormName="listForm" page="/userListAction.do">
<MultiPages:prev>Prev</MultiPages:prev>
<MultiPages:index />
<MultiPages:next>Next</MultiPages:next>
</MultiPages:pager>

```

这是一个 MultiPages 标签，注意标签使用方法前提：

1. 在 Jsp 页面头部需要声明该标签：

```

<% @ taglib uri="/WEB-INF/MultiPages.tld" prefix="MultiPages" %>

```

2. 在 web.xml 中有声明解释该标签库文件所在位置：

```

<taglib>
    <taglib-uri>/WEB-INF/MultiPages.tld</taglib-uri>
    <taglib-location>/WEB-INF/MultiPages.tld</taglib-location>
</taglib>

```

这表示 Jsp 中的 uri 定义 /WEB-INF/MultiPages.tld 实际是执行本地文件 /WEB-INF/MultiPages.tld，那么在你的 Web 项目的 WEB-INF 下必须要有 MultiPages.tld 文件，可以从 Jdon 框架源码包目录 src\META-INF\tlds 下将 MultiPages.tld 拷贝过去即可，有些开发工具如 JBuilder 在配置了框架功能后，会自动在建立 Web 项目时拷贝这些标签库的。

下面说明一下批量查询的 MultiPages 标签使用方法:

```
<MultiPages:pager actionFormName="listForm" page="/userListAction.do">
```

其中 actionFormName 是你在 struts-config.xml 中配置的 ModelListForm 名称,也就是这段配置中的 name 值:

```
<form-bean name="listForm" type="com.jdon.strutsutil.ModelListForm"/>
```

MultiPages 中的 page 是指当前页面是通过哪个 action 调用产生的,这样,在显示多页其它页面也可以调用这个 action,这里是 /userListAction.do,很显然,这个 /userListAction.do 是你的 struts-config.xml 中的 action 配置中的 path 值,如下:

```
<action name="listForm" path="/userListAction" type="com.jdon.framework.test.web.UserListAction"
scope="request">
    <forward name="success" path="/userList.jsp" />
</action>
```

上述配置中的 type 值 com.jdon.framework.test.web.UserListAction 是你的 ModelListAction 实现子类。如果你的查询是条件的,也就是说 /userListAction.do 后面有查询参数,那么你也需要写在 MultiPages 后面,如:

```
<MultiPages:pager actionFormName="listForm" page="/userListAction.do"
paramId="catId" paramName="catId">
```

MultiPages 的语法类似 struts 的 html:page 标签,如果你觉得功能不够丰富,可以自己继承 MultiPages.tld 中定义的类,将自己子类写入 MultiPages.tld 配置即可。

参考一例: <http://www.jdon.com/jive/thread.jsp?forum=61&thread=24112>

MultiPages 标签的其它配置很简单,无需变化,如下:

```
<MultiPages:prev>Prev</MultiPages:prev>
<MultiPages:index />
<MultiPages:next>Next</MultiPages:next>
```

MultiPages:prev 是显示“前页”的标签,Prev 是显示“Prev”,你可以使用图标或汉字来替代 Prev。

<MultiPages:index/>是自动显示 1 2 3 ... 多个数字。目前只提供数字显示。有一个参数 displayCount,表示如果有很多页,显示前面或后面多少个页:

```
<MultiPages:index displayCount="8" />
```

表示在当前页面前面或后面如果超过 8 页,显示 8 页,例如当前页是第 14 页,如下:

```
[Prev ] 1 ... 6 7 8 9 10 11 12 13 14
```

MultiPages:next 和 MultiPages:prev 类似,显示“后页”字样。

MultiPages:prev 和 MultiPages:next 还有另外一种写法,如下:

```
<MultiPages:prev name="[Prev ]" />
```

使用了 name 属性,这样个语法的好处是:当前页面如果没有超过一页,将没有“Prev”或“Next”字样出现,只有超过一个页面时,才会有相应的字样出现,显示智能化。

如果你觉得 MultiPages 这些功能还不能满足你的需要,你可以定制自己的标签库,只要继承 com.jdon.strutsutil.taglib 包下的各个标签库,然后修改自己项目的 WEB-INF 目录下 MultiPages.tld 等 tld 文件,将自己编写的标签类替代原来 com.jdon.strutsutil.taglib 包下的标签类。



## 案例：表现层实现

CRUD 功能只需配置和 Jsp 实现就可以，而批量查询表现层需要进行简单编码，然后再需要配置和 Jsp 完成。还有一个区别是：前者无需编码，替代的是需要配置 jdonframework.xml；而后者因为编码实现，无需配置 jdonframework.xml 了。

编码：UserListAction

UserListAction 继承 ModelListAction 实现，按照前面批量查询章节，这里完成两个方法：getPageIterator 和 findModelByKey：

getPageIterator 方法如下：

```
public PageIterator getPageIterator(HttpServletRequest request, int start,
                                   int count) {
    PageIterator pageIterator = null;
    try {
        TestService testService = (TestService) WebAppUtil.getService("testService", request);
        pageIterator = testService.getAllUsers(start, count);
    } catch (Exception ex) {
        logger.error(ex);
    }
    return pageIterator;
}
```

findModelByKey 方法：

```
public Model findModelByKey(HttpServletRequest request, Object key) {
    Model model = null;
    try {
        TestService testService = (TestService) WebAppUtil.getService("testService", request);
        model = testService.getUser((String) key);
    } catch (Exception ex) {
        logger.error(ex);
    }
    return model;
}
```

代码中：

```
TestService testService = (TestService) WebAppUtil.getService("testService", request);
```

其中 testService 是 jdonframework.xml 配置中 <pojoService name="testService"> 中名称。

Struts-config.xml 配置

(1) 配置一个批量查询的 ActionForm：首先在 struts-config.xml 创建一个用于批量分页查询的 ActionForm，固定推荐名称为 listForm。

```
<form-bean name="listForm" type="com.jdon.strutsutil.ModelListForm" />
```

(2) 配置 UserListAction，按照普通 Struts 的 action 配置

```

<action name="listForm" path="/userListAction" type="jdonframeworktest.web.UserListAction"
>
    <forward name="success" path="/userList.jsp" />
</action>

```

(4) 编写 userList.jsp, 用于分页显示查询结果。

在 userList.jsp 中使用到分页的标签库:

```

<% @ taglib uri="/WEB-INF/MultiPages.tld" prefix="MultiPages" %>

<MultiPages:pager actionFormName=" Struts-config.xml 中 formBeans 名称: listForm"
page="Struts-config.xml 中当前的 action path 值: /userListAction.do">
    <MultiPages:prev>前页</MultiPages:prev>
    <MultiPages:index />
    <MultiPages:next>后页</MultiPages:next>
</MultiPages:pager>

```

我们将 userList.jsp 作为本演示首页, 将新增 修改和删除等操作的执行集中在这个页面实现, 使用 JavaScript 来实现。

### 持久层批量查询实现

这里是为了实现批量查询, 如果不需要批量查询, 可跳过此步:

持久层需要向前台提供数据库等持久层数据的查询和获得, 一般可通过 Jdon 框架的简单 JDBC 模板实现, 我们创建一个 JdbcDao 类。

### PageIterator 对象创建

首先为每个 Model 引入分页对象;

//通过 JNDI 获得 DataSource

```
dataSource = (DataSource) sl.getDataSource(JNDINames.DATASOURCE);
```

//框架使用 为每个 Model 建立一个分页对象。

```
pageIteratorSolverOfUser = new PageIteratorSolver(dataSource);
```

三个方法实现

JdbcDao 有三个缺省方法必须才能完成从数据库中获得所要求的数据, 这三个方法分别是:

User getUsers(String Id); //查询获得单个 Model 实例

PageIterator getUsers(int start, int count); //查询某页面的所有 Model 的 ID 集合

public void clearCacheOfUser(); //清除某个 Model 的缓存。

上述三个方法中, 只有一个方法需要编写代码, 其它可委托 JdonFramework 提供的工具 API 方法完成。

第一个方法完成: 编写获得单个 UserTest 的数据库查询方法: 从现成的示例拷贝如下:

```

public UserTest getUser(String Id) {
    String GET_FIELD = "select * from testuser where userId = ?";
    List queryParams = new ArrayList();
    queryParams.add(Id);
    UserTest ret = null;
    try {

```

```

        List list = pageIteratorSolverOfUser.queryMultiObject(queryParams,
            GET_FIELD);
        Iterator iter = list.iterator();
        if (iter.hasNext()) {
            Map map = (Map) iter.next();
            ret = new UserTest();
            ret.setName((String) map.get("name"));
            ret.setUserId((String) map.get("userId"));
        }
    } catch (Exception se) {
    }
    return ret;
}

```

第二个方法实现：编写分页查询 SQL 实现方法，直接使用框架内部的方法如下：

```

String GET_ALL_ITEMS_ALLCOUNT = "select count(1) from testuser ";

String GET_ALL_ITEMS = "select userId  from testuser ";

return pageIteratorSolverOfUser.getDatas("", GET_ALL_ITEMS_ALLCOUNT,
    GET_ALL_ITEMS, start, count);

```

注意 `getDatas` 的方法参数使用，参考批量查询章节

第三个方法实现：编写缓存清除方法：该方法是在该 `Model` 被新增或删除等情况下调用。这样，当前台页面新增新的数据后，批量查询能够立即显示新的数据，否则将是缓存中老的数据集合，无法立即更新。

```

public void clearCacheOfUser() {
    pageIteratorSolverOfUser.clearCache();
}

```

`JdbcDAO` 模板化编程到此结束。

下一步，使用 `EJB` 的 `Session Bean` 包装这个 `JdbcDao`，如果服务层不是使用 `EJB` 实现，直接通过一般的 `POJO` 服务包装 `JdbcDao`。

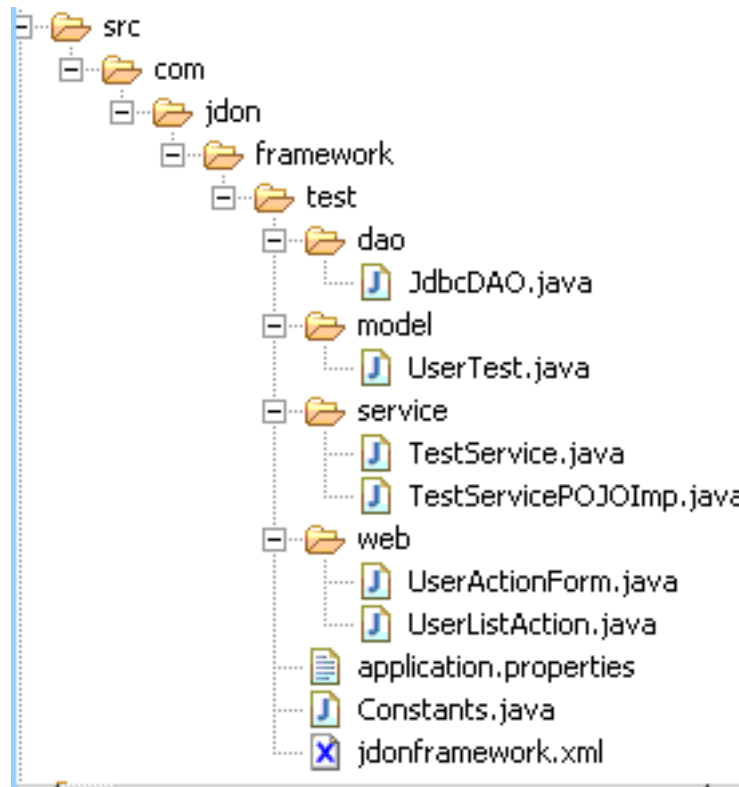
基本全部完成。

## Step By Step:Jdon 应用系统案例开发

这是一个从无到有的开发过程，如果你还追求更高的快速性，那么就使用框架应用源码包中的 `SimpleJdonFrameworkTest` 项目作为模板，将该整个项目拷贝到另外一个目录，将该目录下的 `UserTest` 字符串更换为你的应用模型名称，再修改相应字段即可（现在一些基于脚本语言的快速框架可以自动帮你完成这些）。

案例场景：每个系统都是从域建模入手设计，通过建模将业务需求转化为软件域范围的模型，本文以围绕一个模型实现该模型的基本功能：增删改查(CRUD)和批量分页查询，通过 `Jdon` 框架的迅速简化高质量的开发，建立一个复杂系统的基础部分，使得开发者将真正精力集中在每个项目系统的特殊业务处理。

源码见 Jdon 框架源码包中的 simpleMessage 项目或 SimpleJdonFrameworkTest 项目，  
以下该项目所需要的全部类和配置展示图：5 个类、一个接口和两个配置文件



案例需求：简单的留言簿，实现留言 Message 模型的新增、修改、删除和批量查询。  
增删改查(CRUD)和批量分页查询是每个系统的基本功能，下面分这两部分描述。

## CRUD 开发步骤

说明：每个应用系统中存在大量重复的 CRUD 开发流程，通过本框架可快速完成这些基本基础工作量，将精力集中在特殊功能设计上。

CRUD 快速开发主要简化了表现层的流程，将其固化，或者是模板化，以配置替代代码编制，灵活而快速。每个 Model 一套固化 CRUD 流程。

开发步骤分两个小部分：代码编写和配置。

代码：三步代码编写

代码只需要三步：

第一步：域建模：建立 sample.model.Message，如下：

```
public class Message extends Model {
    private String messageId;
    private String name;

    public String getName() {    return name;    }
```

```
public void setName(String name) {    this.name = name;    }
}
```

注意点：

模型类 `Message` 必须继承框架的 `com.jdon.controller.model.Model`，或者实现 `com.jdon.controller.model.ModelIF` 接口，5.6 版本以后可以使用 `@Model` 注解。

该模型类必须有一个能够标识其对象唯一性的主键，如 `messageId`，这个主键相当于数据表的主键。

第二步：建立 Model 组件服务：首先建立模型 `Message` 的服务接口 `sample.service.MessageService`：

```
public interface MessageService {
    public void createMessage(EventModel em);
    public void updateMessage(EventModel em);
    public void deleteMessage(EventModel em);
    public Message getMessage(String messageId);
}
```

至于 `MessageService` 的具体实现子类可以在现在或者以后建立，可见源码包中的 `sample.service.MessageServiceImp`。

第三步：建立 Model 的表现层边界模型：`sample.web.MessageForm`，必须继承框架的 `ModelForm`，如下：

```
public class MessageForm extends ModelForm {

    private String messageId;
    private String name;

    public String getName() {    return name;    }
    public void setName(String name) {    this.name = name;    }
}
```

表现层 `MessageForm` 内容基本上是从业务层模型 `Message` 类中拷贝过来的，主要是为了保持 `MessageForm` 和 `Message` 的字段一致，我们就可以通过框架内 `MessageForm` 和 `Message` 的相同字段的复制进行数据传送，将业务层的 `Message` 数据传送到表现层 `MessageForm`；或将界面表现层 `MessageForm` 传送到 `Message` 中。

一个模型 `Message` 有关 CRUD 实现的代码工作到此结束，如果有其他模型，完全按照上述三个步骤再做一次，是不是不太费脑筋？有点模板化开发味道？下面谈谈 CRUD 实现第二组成部分：配置。

配置分两个配置文件，这两个配置文件分别是：

将前面三步编写的类建立关系：`jdframework.xml`

配置界面流程：`struts-config.xml`

配置之一：Jdon 框架配置文件

首先我们将前面三步编写的三个类：模型 `Message`、服务 `MessageService` 和界面模型 `MessageForm` 建立起联系，也就是告诉 Jdon 框架这三者是解决一个模型增删改查 CRUD 功能实现的。



由于这个配置文件是告诉 Jdon 框架的，因此，我们取名为 jdonframework.xml，当然你也可以取其他名称，无论取什么名称，都要告诉 Jdon 框架，在 struts-config.xml 中配置

```
<plug-in className="com.jdon.strutsutil.InitPlugIn">
    <set-property property="modelmapping-config" value="jdonframework.xml" />
</plug-in>
```

jdonframework.xml 配置内容如下：

```
<models>
  <!-- 配置模型的类是 Message，其主键是 messageId -->
  <model key="messageId" class="sample.model.Message">
    <!-- 下行是配置界面模型 MessageForm -->
    <actionForm name="messageForm"/>
    <handler>
      <!-- 以下配置 MessageService -->
      <service ref="messageService">
        <getMethod name="getMessage" />
        <createMethod name="createMessage" />
        <updateMethod name="updateMessage" />
        <deleteMethod name="deleteMessage" />
      </service>
    </handler>
  </model>
</models>
<services>
  <!-- 以下配置 MessageService -->
  <pojoService name="messageService" class="sample.service.MessageServiceImp"/>
</services>
```

以上配置是配置模型 Message、模型服务 MessageService 和界面模型 MessageForm 三者关系的，下面详细说明三个部分的配置：

一、模型 Message 的配置：

这是通过第一行中的 class 值来指定当前 Model 是 sample.model.Message：

```
<model key="messageId" class="sample.model.Message">
```

其中，Message 模型的主键是 messageId，这个 messageId 必须是 Message 类的一个字段；同时是用来唯一标识唯一的 Message 模型对象，也就是 Object ID，或者可以认为是模型 Message 对应的数据表 message 的主键。

二、界面模型 MessageForm 配置：

```
<actionForm name="messageForm"/>
```

可能你已经注意到：这里并没有写界面模型完整类：sample.web.MessageForm，而好像是 MessageForm 类的名称 messageForm。

那么配置中 messageForm 名称是从哪里来的呢？是 struts-config.xml 中 ActionForm 定义名称，如下：

```
<struts-config>
  <form-beans>
    <form-bean name="messageForm" type="sample.web.MessageForm" />
    .....
  </form-beans>
```

```
.....
</struts-config>
```

可见我们的界面模型完整类 `sample.web.MessageForm` 是在 `struts-config.xml` 中 `form-beans` 中配置，并且命名为 `messageForm`，而这个 `messageForm` 就是 `jdonframework.xml` 中的 `messageForm`。

### 三、模型服务 `MessageService` 配置：

在 `jdonframework.xml` 中首先申明 `MessageService` 完整实现是类 `sample.service.MessageServiceImp`，并且取名为 `messageService`：

```
<pojoService name="messageService" class="sample.service.MessageServiceImp"/>
```

这样，我们就可以详细将我们自己编写的 `messageService` 的 CRUD 方法名告诉 Jdon 框架了：

```
<handler>
  <!-- 以下配置 MessageService -->
  <service ref="messageService">
    <getMethod name="getMessage" />
    <createMethod name="createMessage" />
    <updateMethod name="updateMessage" />
    <deleteMethod name="deleteMessage" />
  </service>
</handler>
```

黑体字部分正是 `messageService` 所指的类 `sample.service.MessageServiceImp` 所继承的接口 `sample.service.MessageService` 四个方法，可见前面代码步骤第二步。

### 配置之二：界面流程配置

界面流程主要是配置 CRUD 界面流程，Jdon 框架 CRUD 流程主要分两个部分：第一是推出供用户新增修改删除的页面；第二是接受用户提交新增修改过的数据，以便递交到业务层保存。

这部分配置主要是配置 `struts-config.xml`：

#### 第一、配置推出 CRUD 页面流程：

```
<action name="messageForm" path="/messageAction" type="com.jdon.strutsutil.ModelViewAction"
  scope="request" validate="false">
  <forward name="create" path="/message.jsp" />
  <forward name="edit" path="/message.jsp" />
</action>
```

其中 `com.jdon.strutsutil.ModelViewAction` 是 Jdon 框架类。只要客户端浏览器调用 <http://localhost:8080/messageAction.do>，通过上述配置将激活 `forward` 的 `name="create"` 流程，就能得到一个空白表单的页面 `message.jsp`；如果客户端浏览器调用 <http://localhost:8080/messageAction.do?action=edit&messageId=18>，通过上述配置将激活 `forward name="edit"` 流程，得到一个填满数据的表单页面，供用户修改。

#### 第二、配置：接受用户提交新增修改过的数据，以便递交到业务层保存：

```
<action name="messageForm" path="/messageSaveAction"
type="com.jdon.strutsutil.ModelSaveAction"
  scope="request" validate="true" input="/message.jsp">
  <forward name="success" path="/result.jsp" />
  <forward name="failure" path="/result.jsp" />
</action>
```

其实在上一步的 message.jsp 中已经使用到这一步的配置，在 message.jsp 的表单 action 值就是本步配置的 path 值： /messageSaveAction.do：

```
<html:form action="/messageSaveAction.do" method="POST" >
    <html:hidden property="action"/>
        MessageId: <html:text property="messageId"/>
        <br>Name: <html:text property="name"/>
        <br><html:submit property="submit" value="Submit"/>
</html:form>
```

在上面 message.jsp 中一定要有<html:hidden property="action"/>一行。

至此，模型 Message 的 CRUD 功能开发完毕。

## 批量查询实现

代码： 三步代码编写

第一步、表现层编写一个查询 Action，继承 Jdon 框架的 com.jdon.strutsutil.ModelListAction，该类名称为 sample.web. MessageListAction，完成 getPageIterator 和 findModelByKey 两个方法。

其中 getPageIterator 方法内容是业务层 MessageService 的调用：

```
MessageService messageService= (MessageService) WebAppUtil.getService("messageService",request);
return messageService.getAllMessages(start, count);
```

所以 MessageService 接口中必须有 getAllMessages 这个方法，主要功能是返回 PageIterator 对象

findModelByKey 方法内容也是业务层 MessageService 的调用：

```
MessageService messageService= (MessageService) WebAppUtil.getService("messageService", request);
return messageService.getMessage((String)key);
```

MessageService 接口中必须有 getMessage 方法。

第二步、业务层实现 MessageService 接口方法 getAllMessages 内容，一般是直接调用持久层 MessageDao 方法。

第三步、持久层实现返回 PageIterator 对象：

```
public PageIterator getMessages(int start, int count) throws Exception {
    String GET_ALL_ITEMS_ALLCOUNT = "select count(1) from testmessage ";
    String GET_ALL_ITEMS = "select messageId  from testmessage ";
    return pageIteratorSolverOfMessage. getPageIterator (GET_ALL_ITEMS_ALLCOUNT,
        GET_ALL_ITEMS, "",start, count);
}
```

如果有参数，可以如下查询：

```
public PageIterator getMessages(Long categoryId, int start, int count) {
    String GET_ALL_ITEMS_ALLCOUNT = "select count(1) from message where categoryId = ? ";
    String GET_ALL_ITEMS = "select messageId  from message where categoryId = ? ";
    Collection params = new ArrayList(1);
    params.add(categoryId);//参数放在 Collection 中
    return pageIteratorSolver.getPageIterator(GET_ALL_ITEMS_ALLCOUNT, GET_ALL_ITEMS,
params, start, count);
```

```
}
```

### 配置之一：Jdon 框架配置文件

本步骤主要是需要告诉 jdonframework.xml 我们的 MessageService 实现子类是什么，以及调用的 MessageDao 等组件，jdonframework.xml 如下：

```
<services>
  <pojoService name="messageService" class="sample.service.MessageServiceImp"/>
  <component name="messageDAO" class="sample.dao.MessageDAO"/>
  <component name="constants" class="sample.Constants">
    <constructor value="java:/TestDS"/>
  </component>
</services>
```

因为 MessageServiceImp 类中调用了 MessageDAO，MessageDAO 中又涉及 JNDI 名称，所以它们之间依赖关系靠 Jdon 框架的 IOC 容器实现。MessageServiceImp 必须有构造器如下：

```
public class MessageServiceImp implements MessageService{
    private MessageDAO messageDAO;
    public MessageServiceImp(MessageDAO messageDAO){
        this.messageDAO = messageDAO;
    }
}
```

### 配置之二：界面流程配置

这一步主要是 struts-config.xml 配置，和通常 struts 的 ActionForm 和 Action 配置类似：

```
<form-beans>
  .....
  <form-bean name="listForm" type="com.jdon.strutsutil.ModelListForm" />
</form-beans>
```

其中 com.jdon.strutsutil.ModelListForm 是框架批量查询特别使用的类。

```
<action name="listForm" path="/messageListAction"
  type="sample.web.MessageListAction" scope="request">
  <forward name="success" path="/messageList.jsp" />
</action>
```

其中 sample.web.MessageListAction 是我们前面代码编写部分编写的代码。这样，客户端浏览器通过 <http://localhost:8080/messageListAction.do> 就可以实现所有 Message 批量分页查询显示。

注意，messageList.jsp 中编码和通常 Struts 的 Jsp 编码是一样的，需要使用 logic:iterator 从 ActionForm 为 listForm 的 list 字段中获取单个的 Message 对象，然后显示这些单个 Message 对象，如下：

```
<logic:iterate indexId="i" id="message" name="listForm" property="list">
  <bean:write name="message" property="name" />
  .....
```

```
</logic:iterate
```

在 messageList.jsp 中加入下面标签库可以自动显示多页，缺省一个页面显示 30 个条目。

```
<MultiPages:pager actionFormName="listForm" page="/messageListAction.do">
```

```
<MultiPages:prev name="[Prev ]" />
```

```
<MultiPages:index displayCount="1" />
```

```
<MultiPages:next name="[Next ]" />
```

```
</MultiPages:pager>
```

模型 Message 的批量查询功能已经全部完成。

## 复杂案例: JPetstore

以 IBatis.com 的 iBATIS-Jpetstore 为例, 我们使用 Jdon 框架对其重构成为 Jdon-JPetstore, 本章开发环境是 Eclipse (本章案也适用其他开发工具), 部署运行环境是 JBoss。

Eclipse 是一个非常不错的开源开发工具, 使用 Eclipse 开发和使用 JBuilder 将有完全不同的开发方式。我们使用 Eclipse 基于 Jdon 框架开发一个完全 Web 应用, 或者说, 开发一个轻量 (lightweight) 的 J2EE 应用系统。

通过这个轻量系统开发, 说明 Jdon 框架对完全 POJO 架构的支持, 因为 EJB 分布式集群计算能力, 随着访问量提升, 可能需要引入 EJB 架构, 这时只要使用 EJB session Bean 包装 POJO 服务则可以无缝升级到 EJB。使用 Jdon 框架可实现方便简单地架构升迁。

## Eclipse 安装简要说明

1. 下载 Eclipse: 在 <http://www.eclipse.org> 的下载点中选择 tds ISP 比较快。

2. 安装免费插件:

编辑 Jsp 需要 Iomboz : <http://www.objectlearn.com/projects/download.jsp>

注意对应的 Eclipse 版本。

编辑 XML, 使用 Xmlbuddy: <http://xmlbuddy.com/>

基本上这两个插件就够了。

如果希望开发 Hibernate, 插件:

<http://www.binamics.com/hibernatesync>

代码折叠

<http://www.coffee-bytes.com/eclipse/update-site/site.xml>

3. 关键学习 ant 的编写 build.xml, 在 build.xml 将你的 jsp javaclass 打包成 war 或 jar 或 ear 就可以。都可以使用 ant 的 jar 打包, build.xml 只要参考一个模板就可以: SimpleJdonFrameworkTest.rar 有一个现成的, 可拷贝到其它项目后修改后就可用。

然后在这个模板上修改, 参考 ant 的命令参考:

<http://ant.apache.org/manual/tasksoverview.html>

网上有中文版的 ant 参考, 在 google 搜索就能找到。

关键是学习 ant 的 build.xml 编辑, SimpleJdonFrameworkTest.rar 有一个现成的, 可拷贝到其它项目后修改后就可用。

用 ant 编译替代 Eclipse 的缺省编译: 选择项目属性-->Builders ---> new --> Ant Builder --->选择本项目的 build.xml workspace 选择本项目

eclipse 开发就这些, 非常简单, 不象 Jbuilder 那样智能化, 导致项目目录很大。eclipse 只负责源码开发, 其它都由 ant 负责

## 架构设计要点

Jdon-JPetstore 除了保留 iBATIS-JPetstore 4.0.5 的域模型、持久层 ibatis 实现以及 Jsp 页面外，其余部分因为使用了 Jdon 框架而和其有所不同。

保留域模型和 Jsp 页面主要是在不更改系统需求的前提下，重构其架构实现为 Jdon 框架，通过对比其原来的实现或 Spring 的 JPetstore 实现，可以发现 Jdon 框架的使用特点。

在原来 jpetstore iBatis 包会延伸到表现层，例如它的分页查询 PaginatedList，iBatis 只是持久层框架，它的作用范围应该只限定在持久层，这是它的专业范围，如果超过范围，显得 ....。所以，在 Jdon-Jpetstore 中将 iBatis 封装在持久层（砍掉 PaginatedList 这只太长的手），Jdon 框架是一种中间层框架，联系前后台的工作应该由 Jdon 这样的中间层框架完成。

在 iBatis 4.0.5 版本中，它使用了一个利用方法映射 Reflection 的小框架，这样，将原来需要在 Action 实现方法整入了 ActionForm 中实现，ActionForm 变成了一个复杂的对象：页面表单抽象以及与后台 Service 交互，在 ActionForm 中调用后台服务是通过单态模式实现，这是在一般 J2EE 开发中忌讳的一点，关于单态模式的讨论可见：

<http://www.jdon.com/jive/article.jsp?forum=91&thread=17578>

Jdon 框架使用了微容器替代单态，消除了 Jpetstore 的单态隐患，而且也简化了 ActionForm 和服务层的交互动作（通过配置实现）。

## 用户注册登陆模块实现

### 用户域建模（Model）

首先，我们需要从域建模开始，建立正确的领域模型，以用户账号为例，根据业务需求我们确立用户账号的域模型 Account，该模型需要继承 Jdon 框架中的 com.jdon.controller.model.Model。

```
public class Account extends Model {  
  
    private String username;  
    private String password;  
    private String email;  
    private String firstName;  
    private String lastName;  
    .....  
}
```

username 是主键。

域模型建立好之后，就可以花开两朵各表一支，表现层和持久层可以同时开发，先谈谈持久层关于用户模型的 CRUD 功能实现。

### 持久层 Account CRUD 实现

主要是用户的新增和修改，主要用于注册新用户和用户资料修改。

```
public interface AccountDao {
    Account getAccount(String username); //获得一个 Account
    void insertAccount(Account account); //新增
    void updateAccount(Account account); //修改
}
```

持久层可以使用多种技术实现，例如 Jdon 框架的 JdbcTemp 代码实现比较方便，如果你的 sql 语句可能经常改动，使用 iBatis 的 sql 语句 XML 定义有一定好处，本例程使用 Jpetstore 原来的持久层实现 iBatis。见源码包中的 Account.xml

### 表现层 Account 表单创建（ModelForm）

这是在 Domain Model 建立后最重要的一步，是前台表现层 Struts 开发的起步，表单创建有以下注意点：

表单类必须继承 com.jdon.model.ModelForm

表单类基本是 Domain Model 的影子，每一个 Model 对应一个 ModelForm 实例，所谓对应：就是字段名称一致。ModelForm 实例是由 Model 实例复制获得的。

```
public class AccountForm extends ModelForm {

    private String username;
    private String password;
    private String email;
    private String firstName;
    private String lastName;
    .....

}
```

当然 AccountForm 可能有一些与显示有关的字段，例如注册时有英文和中文选择，以及类别的选择，那么增加两个字段在 AccountForm 中：

```
private List languages;
private List categories;
```

这两个字段需要初始化值的，因为在 AccountForm 对应的 Jsp 的页面中要显示出来，这样用户才可能进行选择。选择后的值将放置在专门的字段中。

有两种方式初始化这两个字段：

1. 在 AccountForm 构造方法中初始化，前提是：这些初始化值是常量，如：

```
public AccountForm() {
    languages = new ArrayList();
    languages.add("english");
    languages.add("japanese");
}
```

2. 如果初始化值是必须从数据库中获取，那么采取前面章节介绍的使用 ModelHandler 来实现，这部分又涉及配置和代码实现，缺省时我们考虑通过 jdonframework.xml 配置实现。



## Account CRUD 的 struts-config.xml 的配置

### 第一步配置 ActionForm:

上节编写了 ModelForm 代码，ModelForm 也就是 struts 的 ActionForm，在 struts-config.xml 中配置 ActionForm 如下：

```
<form-bean name="accountForm"
           type="com.jdon.framework.samples.jpetestore.presentation.form.AccountForm"/>
```

### 第二步配置 Action:

这需要根据你的 CRUD 功能实现需求配置，例如本例中用户注册和用户修改分开，这样，配置两套 ModelViewAction 和 ModelSaveAction，具体配置见源码包中的 struts-config-security.xml，这里将 struts-config.xml 根据模块划分成相应的模块配置，实现多模块开发，本模块是用户注册登陆系统，因此取名 struts-config-security.xml。

## Account CRUD 的 Jdonframework.xml 配置

```
<model key="username" class="com.jdon.framework.samples.jpetestore.domain.Account">
  <actionForm name="accountForm"/>
  <handler>
    <service ref="accountService">
      <initMethod name="initAccount" />
      <getMethod name="getAccount" />
      <createMethod name="insertAccount" />
      <updateMethod name="updateAccount" />
      <deleteMethod name="deleteAccount" />
    </service>
  </handler>
</model>
```

其中有一个 initMethod 主要用于 AccountForm 对象的初始化。其他都是增删改查的常规实现。

## Account CRUD 的 Jsp 页面实现

在编辑页面 EditAccountForm.jsp 中加入：

```
<html:hidden name="accountForm" property="action" value="create" />
```

在新增页面 NewAccountForm.jsp 加入：

```
<html:hidden name="accountForm" property="action" value="edit" />
```

所有的字段都是直接来自 accountForm。

## 整理模块配置

商品模块功能完成，struts 提供了多模块开发，因此我们可以将这一模块单独保存在一个配置中：/WEB-INF/struts-config-security.xml，这样以后扩展修改起来方便。

## 商品查询模块实现

在 iBATIS-JPetstore 中没有单独的 CategoryForm，而是将三个 Model: Category、Product、Item 合并在一个 CatalogBean 中，这样做的缺点是拓展性不强，将来这三个 Model 也许需要单独的 ActionForm。

由于我们使用 Jdon 框架的 CRUD 功能配置实现，因此，不怕细分这三个 Model 带来代码复杂和琐碎。

由于原来的 Jpetstore “偷懒”，没有实现 Category Product 等的 CRUD 功能，只实现它们的查询功能，因此，我们使用 Jdon 框架的批量查询来实现查询。

### 持久层 Product 批量查询实现

商品查询主要有两种批量查询，根据其类别 ID: CategoryId 查询所有该商品目录下所有的商品；根据关键字搜索符合条件的所有商品，下面以前一个功能为例子：

iBatis-jpetstore 使用 PaginatedList 作为分页的主要对象，该对象需要保存到 HttpSession 中，然后使用 PaginatedList 的 NextPage 等直接遍历，这种方法只适合在小数据量合适，J2EE 编程中不推荐向 HttpSession 放入大量数据，不利于 cluster。

根据 Jdon 批量查询的持久层要求，批量查询需要两种 SQL 语句实现：符合条件的 ID 集合和符合条件的总数：以及单个 Model 查询。

```
//获得 ID 集合
List getProductIDsListByCategory(String categoryId, int pagessize);
//获得总数
int getProductIDsListByCategoryCount(String categoryId);
//单个 Model 查询
Product getProduct(String productId);
```

这里我们需要更改一下 iBatis 原来的 Product.xml 配置，原来，它设计返回的是符合条件的所有 Product 集合，而我们要求是 Product ID 集合。

修改 Product.xml 如下：

```
<resultMap id="productIDsResult" class="java.lang.String">
    <result property="value" column="PRODUCTID"/>
</resultMap>

<select id="getProductListByCategory" resultMap="productIDsResult" parameterClass="string">
    select PRODUCTID from PRODUCT where CATEGORY = #value#
</select>

<select id="getProductListByCategoryCount" resultClass="java.lang.Integer"
parameterClass="string">
    select count(1) as value from PRODUCT where CATEGORY = #value#
</select>
```

ProductDao 是 IBatis DAO 实现，读取 Product.xml 中配置：

```
public List getProductIDsListByCategory(String categoryId, int start, int pagessize) {
    return sqlMapDaoTemplate.queryForList(
        "getProductListByCategory", categoryId, start, pagessize);
}

public int getProductIDsListByCategoryCount(String categoryId){
```

```

        Integer countI = (Integer)sqlMapDaoTemplate.queryForObject(
            "getProductListByCategoryCount", categoryId);
        return countI.intValue();
    }

```

这样，结合配置的 iBatis DAO 和 Jdon 框架批量查询，在 ProductManagerImp 中创建 PageIterator，当然这部分代码也可以在 ProductDao 实现，创建 PageIterator 代码如下：

```

    public PageIterator getProductIDsListByCategory(String categoryId, int start, int count)
    {
        PageIterator pageIterator = null;
        try {
            List list = productDao.getProductIDsListByCategory(categoryId, start, count);
            int allCount = productDao.getProductIDsListByCategoryCount(categoryId);
            int currentCount = start + list.size();
            pageIterator = new PageIterator(allCount, list.toArray(), start,
                (currentCount < allCount)?true:false);

        } catch (DaoException daoe) {
            Debug.logError(" Dao error : " + daoe, module);
        }

        return pageIterator;
    }

```

表现层 Product 批量查询实现

根据批量查询的编程步骤，在表现层主要是实现 ModelListAction 继承、配置和 Jsp 编写，下面分步说：

第一步，创建一个 ModelListAction 子类 ProductListAction，实现两个方法：getPageIterator 和 findModelByKey，getPageIterator 方法如下：

```

    public PageIterator getPageIterator(HttpServletRequest request, int start,
        int count) {
        ProductManager productManager = (ProductManager) WebAppUtil.getService(
            "productManager", request);
        String categoryId = request.getParameter("categoryId");
        return productManager.getProductIDsListByCategory(categoryId, start, count);
    }

```

findModelByKey 方法如下：

```

    public Model findModelByKey(HttpServletRequest request, Object key) {
        ProductManager productManager = (ProductManager) WebAppUtil.getService(
            "productManager", request);
        return productManager.getProduct((String)key);
    }

```

由于我们实现的是查询一个商品目录下所有商品功能，因此，需要显示商品目录名称，而前面操作的都是 Product 模型，所以在显示页面也要加入商品目录 Category 模型，我们使用 ModelListAction 的 customizeListForm 方法：

```

    public void customizeListForm(ActionMapping actionMapping,
        ActionForm actionForm, HttpServletRequest request,
        ModelListForm modelListForm) throws Exception {
        ModelListForm listForm = (ModelListForm) actionForm;
    }

```

```

        ProductManager productManager = (ProductManager) WebAppUtil.getService(
            "productManager", request);
        String categoryId = request.getParameter("categoryId");
        Category category = productManager.getCategory(categoryId);
        listForm.setOneModel(category);
    }

```

第二步，配置 struts-config.xml，配置 ActionForm 和 Action：

```

<form-bean name="productListForm" type="com.jdon.strutsutil.ModelListForm"/>

```

action 配置如下：

```

<action path="/shop/viewCategory"
        type="com.jdon.framework.samples.jpstore.presentation.action.ProductListAction"
        name="productListForm" scope="request"
        validate="false" >
    <forward name="success" path="/catalog/Category.jsp"/>
</action>

```

第三步，编写 Category.jsp

从 productListForm 中取出我们要显示两个模型，一个是 oneModel 中的 Category；另外一个 Product Model 集合 list，Jsp 语法如下：

```

<bean:define id="category" name="productListForm" property="oneModel" />
<bean:define id="productList" name="productListForm" property="list" />

```

我们可以显示商品目录名称如下：

```

<h2><bean:write name="category" property="name" /></h2>

```

这样我们就可以遍历 productList 中的 Product 如下：

```

<logic:iterate id="product" name="productList" >
    <tr bgcolor="#FFFF88">
        <td><b><html:link paramId="productId" paramName="product" paramProperty="productId"
page="/shop/viewProduct.shtml"><font color="BLACK"><bean:write name="product"
property="productId" /></font></html:link></b></td>
        <td><bean:write name="product" property="name" /></td>
    </tr>
</logic:iterate>

```

加上分页标签库如下：

```

<MultiPages:pager actionFormName="productListForm" page="/shop/viewCategory.do"
        paramId="categoryId" paramName="category" paramProperty="categoryId">
<MultiPages:prev></MultiPages:prev>
<MultiPages:index />
<MultiPages:next></MultiPages:next>
</MultiPages:pager>

```

至此，一个商品目录下的所有商品批量查询功能完成，由于是基于框架的模板化编程，直接上线运行成功率高。

商品搜索批量查询：

参考上面步骤，商品搜索也可以顺利实现，从后台到前台按照批量查询这条线索分别涉及的类有：

持久层实现：ProductDao 中的三个方法：

```

List searchProductIDsList(String keywords, int start, int pagesize); //ID 集合

```

```
int searchProductIDsListCount(String keywords); //总数
```

```
Product getProduct(String productId); //单个 Model
```

表现层：建立 ProductSearchAction 类，配置 struts-config.xml 如下：

```
<action path="/shop/searchProducts"
        type="com.jdon.framework.samples.jpetest.presentation.action.ProductSearchAction"
        name="productListForm" scope="request"
        validate="false">
    <forward name="success" path="/catalog/SearchProducts.jsp"/>
</action>
```

与前面使用的都是同一个 ActionForm：productListForm

编写 SearchProducts.jsp，与 Category.jsp 类似，相同的是 ActionForm；不同的是 action。

商品条目 Item 批量查询

条目 Item 批量实现与 Product 批量查询类似：

持久层：ItemDao 提供三个方法：

```
List getItemIDsListByProduct(String productId, int start, int pagesize); //ID 集合
```

```
int getItemIDsListByProductCount(String productId); //总数
```

```
Item getItem(String itemId); //单个 Model
```

表现层：创建一个 ItemListAction 继承 ModelListAction：完成 getPageIterator 和 findModelByKey，如下：

```
public class ItemListAction extends ModelListAction {

    public PageIterator getPageIterator(HttpServletRequest request, int start,
        int count) {
        ProductManager productManager = (ProductManager) WebAppUtil.getService(
            "productManager", request);
        String productId = request.getParameter("productId");
        return productManager.getItemIDsListByProduct(productId, start, count);
    }

    public Model findModelByKey(HttpServletRequest request, Object key) {
        ProductManager productManager = (ProductManager) WebAppUtil.getService(
            "productManager", request);
        return productManager.getItem((String)key);
    }

    public void customizeListForm.....
}
```

与前面的 ProductListAction 相比，非常类似，不同的是 Model 名称不一样，一个是 Product 一个是 Item；

struts-config.xml 配置如下：

```
<form-bean name="itemListForm" type="com.jdon.strutsutil.ModelListForm"/>
```

```

<action path="/shop/viewProduct"
        type="com.jdon.framework.samples.jpetestore.presentation.action.ItemListAction"
        name="itemListForm" scope="request"
        validate="false">
    <forward name="success" path="/catalog/Product.jsp"/>
</action>

```

比较前面 product 的配置，非常类似，其实 itemListForm 和 productListForm 是同一个 ModelListForm 类型，可以合并起来统一命名为 listForm，节省 ActionForm 的配置。

Product.jsp 页面与前面的 Category.jsp SearchProdcuts.jsp 类似。

```

<bean:define id="product" name="itemListForm" property="oneModel" />
<bean:define id="itemList" name="itemListForm" property="list" />

```

分页显示：

```

<MultiPages:pager actionFormName="itemListForm" page="/shop/viewProduct.do"
        paramId="productId" paramName="product" paramProperty="productId">
    .....

```

商品条目 Item 单条查询

单个显示属于 CRUD 中的一个查询功能，我们需要建立 Model 对应的 ModelForm，将 Item 的字段拷贝到 ItemForm 中。配置这个 ActionForm 如下：

```

<form-bean name="itemForm"
        type="com.jdon.framework.samples.jpetestore.presentation.form.ItemForm"/>

```

第二步：因为这个功能属于 CRUD 一种，无需编程，但是需要配置 jdonframework.xml：

```

<model key="itemId" class="com.jdon.framework.samples.jpetestore.domain.Item">
    <actionForm name="itemForm"/>
    <handler>
        <service ref="productManager">
            <getMethod name="getItem" />
        </service>
    </handler>
</model>

```

配置中只要一个方法 getMethod 就可以，因为只用到 CRUD 中的读取方式。

第三步：配置 struts-config.xml 如下：

```

<action path="/shop/viewItem" type="com.jdon.strutsutil.ModelDispAction"
        name="itemForm" scope="request"
        validate="false">
    <forward name="success" path="/catalog/Item.jsp" />
    <forward name="failure" path="/catalog/Product.jsp" />
</action>

```

第四步编辑 Item.jsp，现在开始发现一个问题，Item.jsp 中不只是显示 Item 信息，还有 Product 信息，而前面我们定义的是 Item 信息，如果使得 Item.jsp 显示 Product 信息呢，这就从设计起源 Domain Model 上考虑，在 Item 的 Model 中有 Product 引用：

```

private Product product;
public Product getProduct() { return product; }
public void setProduct(Product product) { this.product = product; }

```

Item 和 Product 的多对一关系其实应该在域建模开始就考虑到了。

那么，我们只要在持久层查询 Item 时，能够将其中的 Product 字段查询就可以。在

持久层的 iBatis 的 Product.xml 实现有下列 SQL 语句:

```
<select id="getItem" resultMap="resultWithQuantity" parameterClass="string">
  select
    I.ITEMID, LISTPRICE, UNITCOST, SUPPLIER, I.PRODUCTID, NAME,
    DESCN, CATEGORY, STATUS, ATTR1, ATTR2, ATTR3, ATTR4, ATTR5, QTY
  from ITEM I, INVENTORY V, PRODUCT P where P.PRODUCTID = I.PRODUCTID and
I.ITEMID = V.ITEMID and I.ITEMID = #value#
</select>
```

这段语法实际在查询 Item 时, 已经将 Product 查询出来, 这样 Item Model 中已经有 Product 数据, 因为 ActionForm 是 Model 映射, 因此, 前台 Jsp 也可以显示 Product 数据。

在 Item.jsp 中, 进行下面定义:

```
<bean:define id="product" name="itemForm" property="product" />
<bean:define id="item" name="itemForm" />
```

将 itemForm 中 product 属性定义为 product 即可; 这样不必大幅度修改原来的 Item.jsp 了。

整理模块配置

商品模块功能完成, struts 提供了多模块开发, 因此我们可以将这一模块单独保存在一个配置中: /WEB-INF/struts-config-catalog.xml, 这样以后扩展修改起来方便。

## 购物车模块实现

购物车属于一种有状态数据, 也就是说, 购物车的 scope 生命周期是用户, 除非这个用户离开, 否则购物车一直在内存中存在。

有态 POJO 服务

现在有两种解决方案:

第一, 将购物车状态作为数据类, 保存到 ActionForm 中, 设置 scope 为 session, 这种形式下, 对购物车的数据操作如加入条目等实现不很方便, iBatis-jpetstore 4.0.5 就采取这个方案, 在数据类 Cart 中存在大量数据操作方法, 那么 Cart 这个类到底属于数据类 Model? 还是属于处理服务类呢?

在我们 J2EE 编程中, 通常使用两种类来实现功能, 一种是数据类, 也就是我们设计的 Model; 一种是服务类, 如 POJO 服务或 EJB 服务, 服务属于一种处理器, 处理过程。使用这两种分类比较方便我们来解析业务需求, EJB 中实体 Bean 和 Session Bean 也是属于这两种类型。

iBatis-jpetstore 4.0.5 则是将服务和数据类混合在一个类中, 这也属于一种设计, 但是我们认为它破坏了解决问题的规律性, 而且造成数据和操作行为耦合性很强, 在设计模式中我们还使用桥模式来分离抽象和行为, 因此这种做法可以说是反模式的。那么我们采取数据类和服务分离的方式方案来试试看:

第二. 购物车功能主要是对购物车这个 Model 的 CRUD, 与通常的 CRUD 区别是, 数据是保存到 HttpSession, 而不是持久化到数据库中, 是数据状态保存不同而已。所以如果我们实现一个 CartService, 它提供 add 或 update 或 delete 等方法, 只不过操作对

象不是数据库,而是其属性为购物车 Cart,然后将该 CarService 实例保存到 HttpSession,实现每个用户一个 CartService 实例,这个我们成为有状态的 POJO 服务。

这种处理方式类似 EJB 架构处理,如果我们业务服务层使用 EJB,那么使用有态会话 Bean 实现这个功能。

现在问题是, Jdon 框架目前好像没有提供有状态 POJO 服务实例的获得,那么我们自己在 WebAppUtil.getService 获得实例后,保存到 HttpSession 中,下次再到 HttpSession 中获得,这种有状态处理需要表现层更多代码,这就不能使用 Jdon 框架的 CRUD 配置实现了,需要我们代码实现 ModelHandler 子类。

考虑到可能在其他应用系统还有这种需求,那么能不能将有状态的 POJO 服务提炼到 Jdon 框架中呢? 关键使用什么方式加入框架,因为这是设计目标服务实例的获得,框架主要流程代码又不能修改,怎么办?

Jdon 框架的 AOP 功能在这里显示了强大灵活性,我们可以将有状态的 POJO 服务实例获得作为一个拦截器,拦截在原来 POJO 服务实例获得之前。在 Jdon 框架设计中,目标服务实例的获得一般只有一次。

创建有状态 POJO 服务拦截器 com.jdon.aop.interceptor.StatefulInterceptor,再创建一个空接口: com.jdon.controller.service.StatefulPOJOService, 需要实现有状态实例的 POJO 类只要继承这个接口就可以。

配置 aspect.xml, 加入这个拦截器:

```
<interceptor name="statefulInterceptor" class="com.jdon.aop.interceptor.StatefulInterceptor"
    pointcut="pojoServices" />
```

这里需要注意的是: 你不能让一个 POJO 服务类同时继承 Poolable, 然后又继承 Stateful, 因为这是两种不同的类型, 前者适合无状态 POJO; 后者适合 CartService 这样有状态处理; 这种选择和 EJB 的有态/无态选择是一样的。

## Model 和 Service 设计

购物车模块主要围绕域模型 Cart 展开, 需要首先明确 Cart 是一个什么样的业务模型, 购物车页面是类似商品条目批量查询页面, 不过购物车中显示的不仅是商品条目, 还有数量, 那么我们专门创建一个 Model 来指代它, 取名为 CartItem, CartItem 是 Item 父集, 多了一个数量。

这样购物车页面就是 CartItem 的批量查询页面, 然后还有 CartItem 的 CRUD 操作, 所以购物车功能主要是 CartItem 的 CRUD 和批量查询功能。

iBatis 4.0.5 原来设计了专门 Cart Model, 其实这个 Cart 主要是一个功能类, 因为它的数据库项只有一个 Map 和 List, 这根本不能代表业务需求中的一个模型。虽然 iBatis 4.0.5 也可以自圆其说实现了购物车功能, 但是这种实现是随心所欲, 无规律性可循, 因而以后维护起来也是困难, 维护人员理解困难, 修改起来也没有章程可循, 甚至乱改一气。

CartItem 可以使用 iBatis 原来的 CartItem, 这样也可保持 Cart.jsp 页面修改量降低。删除原来的 Cart 这个 Model, 建立对应的 CartService, 实现原来的 Cart 一些功能。

```
public interface CartService {
    CartItem getCartItem(String itemId);
    void addCartItem(EventModel em);
    void updateCartItem(EventModel em);
    void deleteCartItem(EventModel em);
    PageIterator getCartItems();
}
```



CartServiceImp 是 CartService 子类，它是一个有状态 POJO 服务，代码简要如下：

```
public class CartServiceImp implements CartService, Stateful{
    private ProductManager productManager;
    //将原来 iBatis 中 Cart 类中两个属性移植到 CartServiceImp 中
    private final Map itemMap = Collections.synchronizedMap(new HashMap());
    private List itemList = new ArrayList();

    public CartServiceImp(ProductManager productManager) {
        super();
        this.productManager = productManager;
    }
    .....
}
```

itemMap 是装载 CartItem 的一个 Map，是类属性，由于 CartServiceImp 是有状态的，每个用户一个实例，那么也就是每个用户有自己的 itemMap 列表，也就是购物车。

CartServiceImp 中的 getCartItemIDs 是查询购物车当前页面的购物条目，属于批量分页查询实现，这里有一个需要考量的地方，是 getCartItems 方法还是 getCartItemIDs 方法？也就是返回 CartItem 的实例集合还是 CartItem 的 ItemId 集合？按照前面标准的 Jdon 框架批量分页查询实现，应该返回 CartItem 的 ItemId 集合，然后由 Jdon 框架的 ModelListAction 根据 ItemId 首先从缓存中获得 CartItem 实例，但是本例 CartItem 本身不是持久化在数据库，而也是内存 HttpSession 中，所以 ModelListAction 这种流程似乎没有必要。

如果将来业务需求变化，购物车状态不是保存在内存而是数据库，这样，用户下次登陆时，可以知道他上次购物车里的商品条目，那么采取 Jdon 框架标准查询方案还是有一定扩展性的。

这里，我们就事论事，采取返回 CartItem 的实例集合，展示一下如何灵活应用 Jdon 框架的批量查询功能。下面是 CartServiceImp 的 getCartItems 方法详细代码：

```
public PageIterator getCartItems(int start, int count) {
    int offset = itemList.size() - start; //获得未显示的总个数
    int pageCount = (count < offset)?count:offset;
    List pageList = new ArrayList(pageCount); //当前页面记录集合
    for(int i=start; i< pageCount + start;i++){
        pageList.add(itemList.get(i));
    }
    int allCount = itemList.size();
    int currentCount = start + pageCount;
    return new PageIterator(allCount, pageList.toArray(new CartItem[0]), start,
        (currentCount < allCount)?true:false);
}
```

getCartItems 方法是从购物车所有条目 itemList 中查询获得当前页面的条目，并创建一个 PageIterator。

注意，现在这个 PageIterator 中 keys 属性中装载的不是数据 ID 集合，而是完整的 CartItem 集合，因为上面代码中 pageList 中对象是从 itemList 中获得，而 itemList 中装载的都是 CartItem。

## 表现层购物车显示功能

由于 `PageIterator` 中封装的是完整 `Model` 集合，而不是 `ID` 集合，所以现在表现层有两种方案，继承框架的 `ModelListAction`；或重新自己实现一个 `Action`，替代 `ModelListAction`。

这里使用继承框架的 `ModelListAction` 方案，巧妙地实现我们的目的，省却编码：

```
public class CartListAction extends ModelListAction {

    public PageIterator getPageIterator(HttpServletRequest request, int arg1, int arg2) {
        CartService cartService = (CartService)WebAppUtil.getService("cartService", request);
        return cartService.getCartItems();
    }

    public Model findModelByKey(HttpServletRequest arg0, Object key) {
        return (Model)key; //因为 key 不是主键，而是完整的 Model，直接返回
    }

    protected boolean isEnabledCache(){
        return false; //无需缓存，CartItem 本身实际是在内存中。
    }

}
```

配置 `struts-config.xml`：

```
<form-beans>
    <form-bean name="listForm" type="com.jdon.strutsutil.ModelListForm"/>
</form-beans>
<action-mappings>
    <action path="/shop/viewCart"
        type="com.jdon.framework.samples.jpetestore.presentation.action.CartListAction"
        name="listForm" scope="request"
        validate="false">
        <forward name="success" path="/cart/Cart.jsp"/>
    </action>
    .....
</action-mappings>
```

上面是购物车显示实现，只要调用 `/shop/viewCart.shtml` 就可以显示购物车了。

在 `Cart.jsp` 页面插入下面标签：

```
<logic:iterate id="cartItem" name="listForm" property="list">
    ....
</logic:iterate>
```

分页显示标签如下：

```
<MultiPages:pager actionFormName="listForm" page="/shop/viewCart.shtml">
<MultiPages:prev name="<font color=green><B>&lt;&lt; Prev</B></font>" />
<MultiPages:index />
<MultiPages:next name="<font color=green><B>Next &gt;&gt;</B></font>" />
</MultiPages:pager>
```

## 购物车新增删除条目功能

前面完成了购物车显示功能，下面是设计购物车的新增和删除、修改功能。

参考 Jdon 框架的 CRUD 功能实现，Model 是 CartItem，配置 jdonframework.xml 使其完成新增删除功能：

```
<model key="workingItemId"
      class="com.jdon.framework.samples.jpststore.domain.CartItem">
  <actionForm name="cartItemForm"/>
  <handler>
    <service ref="cartService">
      <createMethod name="addCartItem"/>
      <deleteMethod name="deleteCartItem"/>
    </service>
  </handler>
</model>
```

在这个配置中，只有新增和删除方法，修改方法没有，因为购物车修改主要是其中商品条目的数量修改，它不是逐条修改，而是一次性批量修改，这里的 Model 是 CartItem，这是购物车里的一个条目，因此如果这里写修改，也只是 CartItem 一个条目的修改，不符合我们要求。下面专门章节实现这个修改。

表现层主要是配置，没有代码，代码都依靠 cartService 中的 addCartItem 和 deleteCartItem 实现：例如：

```
public void addCartItem(EventModel em) {
    CartItem cartItem = (CartItem) em.getModel();
    String workingItemId = cartItem.getWorkingItemId();
    .....
}
```

注意 addCartItem 中从 EventModel 实例中获取的 Model 是 CartItem，这与我们在 jdonframework.xml 中上述定义的 Model 类型是统一的。

Struts-config.xml 中定义是 CRUD 的标准定义，注意，这里只有 ModelSaveAction 无需 ModelViewAction，因为将商品条目加入或删除购物车这个功能没有专门的显示页面：

```
<action path="/shop/addItemToCart" type="com.jdon.strutsutil.ModelSaveAction"
      name="cartItemForm" scope="request"
      validate="false">
  <forward name="success" path="/cart/Cart.jsp"/>
</action>

<action path="/shop/removeItemFromCart" type="com.jdon.strutsutil.ModelSaveAction"
      name="cartItemForm" scope="request"
      validate="false">
  <forward name="success" path="/cart/Cart.jsp"/>
</action>
```

注意，调用删除功能时，需要附加 action 参数：

/shop/removeItemFromCart.shtml?action=delete

而/shop/addItemToCart.shtml 是新增属性，缺省后面无需跟参数。

## 购物车条目批量修改功能

上面基本完成了购物车主要功能;购物车功能一个复杂性在于其显示功能和修改功能合并在一起,修改功能是指修改购物车里所有商品条目的数量。

既然有修改功能,而且这个修改功能比较特殊,我们需要设计一个独立的 **ActionForm**, 用来实现商品条目数量的批量修改。

首先设计一个 **ActionForm (ModelForm)**, 该 **ModelForm** 主要用来实现购物车条目数量的更改, 取名为 **CartItemsForm**, 其内容如下:

```
public class CartItemsForm extends ModelForm {
    private String[] itemId;
    private int[] quantity;
    private BigDecimal totalCost;
    .....
}
```

**itemId** 和 **quantity** 设计成数组,这样,Jsp 页面可以一次性提交多个 **itemId** 和 **quantity** 数值。

现在 **CartItemsForm** 已经包含前台 jsp 输入的数据, 我们还是将其传递递交到服务层实现处理。因此建立一个 **Model**, 内容与 **CartItemsForm** 类似, 这里的 **Model** 名为 **CartItems**, 实际是一个传输对象。

```
public class CartItems extends Model{
    private String[] itemId;
    private int[] quantity;
    private BigDecimal totalCost;
    .....
}
```

表现层在 **jdonframework.xml** 定义配置就无需编码, 配置如下:

```
<model key=" "
    class="com.jdon.framework.samples.jpetestore.domain.CartItems">
    <actionForm name="cartItemsForm"/>
    <handler>
        <service ref="cartService">
            <updateMethod name="updateCartItems"/>
        </service>
    </handler>
</model>
```

上面配置中, **Model** 是 **CartItems**, **ActionForm** 是 **cartItemsForm**, 这两个是专门为批量修改设立的。只有一个方法 **updateMethod**。因为在这个更新功能中, 没有根据主键从数据库查询 **Model** 的功能, 因此, 这里 **model** 的 **key** 可以为空值。

服务层 **CartServiceImp** 的 **updateCartItems** 方法实现购物车条目数量更新:

```
public void updateCartItems(EventModel em) {
    CartItems cartItems = (CartItems) em.getModel();
    try {
        String[] itemIds = cartItems.getItemId();
        int[] qtys = cartItems.getQuantity();
        int length = itemIds.length;
        for (int i = 0; i < length; i++) {
            updateCartItem(itemIds[i], qtys[i]); //逐条更新购物车中的数量
        }
    }
}
```

```

    }
    } catch (Exception ex) {
        logger.error(ex);
    }
}

```

注意 `updateCartItems` 中从 `EventModel` 取出的是 `CartItems`，和前面 `addCartItem` 方法中取出的是 `CartItem` `Model` 类型不一样，这是因为这里我们在 `jdonframework.xml` 中定义与 `updateCartItems` 相对应的 `Model` 是 `CartItems`。

最后一步工作是 `Cat.jsp` 中加入 `CartItemsForm`，能够在购物车显示页面有一个表单提交，客户按提交按钮，能够立即实现当前页面购物车数量的批量修改。`Cat.jsp` 加入如下代码：

```

<html:form action="/shop/updateCartQuantities.shtml" method="post" >
<html:hidden property="action" value="edit" />
.....
<input type="hidden" name="itemId" value="<bean:write name="cartItem"
property="workingItemId"/>">
    <input type="text" size="3" name="quantity" value="<bean:write name="cartItem"
                                property="quantity"/>" />
.....

```

注意，一定要有 `action` 赋值 `edit` 这一行，这样提交给 `updateCartQuantities.shtml` 实际是 `ModelSaveAction` 时，框架才知道操作性质。

#### 购物车总价显示功能

最后，还有一个功能需要完成，在购物车显示时，需要显示当前购物车的总价格，注意不是显示当前页面的总价格，所以无法在 `Cart.jsp` 直接实现，必须遍历购物车里所有 `CartItem` 计算总数。

该功能是购物车显示时一起实现，购物车显示是通过 `CartListAction` 实现的，这个 `CartListAction` 实际是生成一个 `ModelListForm`，如果 `ModelListForm` 能够增加一个 `getTotalPrice` 方法就可以，因此有两种实现方式：继承 `ModelListForm` 加入自己的 `getTotalPrice` 方法；第二种无需再实现自己的 `ModelListForm`，`ModelListForm` 可以携带一个 `Model`，通过 `setOneModel` 即可，这个方法是在 `ModelListAction` 的子类 `CartListAction` 可以 `override` 覆盖实现的，在 `CartListAction` 加入下列方法：

```

protected Model setOneModel(HttpServletRequest request){
    CartService cartService = (CartService)WebAppUtil.getService("cartService", request);
    CartItems cartItems = new CartItems();
    cartItems.setTotalCost(cartService.getSubTotal());
    return cartItems;
}

```

我们使用空的 `CartItems` 作为携带价格总数的 `Model`，然后在 `Cart.jsp` 中再取出来显示：

```

<bean:define id="cartItems" name="listForm" property="oneModel" />
<b>Sub Total: <bean:write name="cartItems" property="subTotal" format="$#,##0.00" />

```

将当前页面 `listForm` 中属性 `oneModel` 定义为 `cartItems`，它实际是我们定义的 `CartItems`，

下一行取出总价即可。

## 用户喜欢商品列表功能

在显示购物车时，需要一起显示该用户喜欢的商品列表，很显然这是一个批量分页查询实现，但是它有些特殊，它首先显示的第一页不是由 URL 调用的，而是嵌入在购物车显示中，那么只能在购物车显示页面的 **ModellistForm** 中做文章。

在上节中，在 **CartListAction** 中 **setOneModel** 方法中，使用 **CartItems** 作为价格总数的载体，现在恐怕我们也要将之作为本功能实现载体。

还有一种实现载体，就是其他 scope 为 session 的 **ActionForm**，**AccountForm** 很适合做这样的载体，而且和本功能意义非常吻合，所以在 **AccountForm/Account** 中增加一个 **myList** 字段，在 **myList** 字段中，放置的是该用户喜欢的商品 **Product** 集合，注意不必放置 **Product** 的主键集合，因为我们只要显示用户喜欢商品的第一页，这一页是嵌入购物车显示页面中，所以第一页显示的个数是由程序员可事先在程序中定义。

这样在 **Account** 获得时，一起将 **myList** 集合值获得。

## 订单模块实现

我们还是从域模型开始，**Order** 是订单模块的核心实体，其内容可以确定如下：

```
public class Order extends Model {

    /* Private Fields */

    private int orderId;
    private String username;
    private Date orderDate;
    private String shipAddress1;
    private String shipAddress2;
    .....
}
```

第二步，建立与 **Model** 对应的 **ModelForm**，我们可以称之为边界模型，代码从 **Order** 拷贝过来即可。当然 **OrderForm** 还有一些特殊的字段以及初始化：

```
public class OrderForm extends ModelForm
{
    private boolean shippingAddressRequired;
    private boolean confirmed;
    static {
        List cardList = new ArrayList();
        cardList.add("Visa");
        cardList.add("MasterCard");
        cardList.add("American Express");
        CARD_TYPE_LIST = Collections.unmodifiableList(cardList);
    }

    public OrderForm(){
        this.shippingAddressRequired = false;
        this.confirmed = false;
    }
    .....
}
```

第三步，建立 Order Model 的业务服务接口，如下：

```
public interface OrderService {  
    void insertOrder(Order order);  
    Order getOrder(int orderId);  
    List getOrdersByUsername(String username);  
}
```

第四步，实现 OrderService 的 POJO 子类：OrderServiceImp。

第五步，表现层实现，本步骤可和第四步同时进行。

OrderService 中有订单的插入创建功能，我们使用 Jdon 框架的 CRUD 中 create 配置实现，配置 struts-config.xml 和 jdonframework.xml：

```
<form-bean name="orderForm"  
    type="com.jdon.framework.samples.jpeteststore.presentation.form.OrderForm"/>
```

```
<model key="orderId"  
    class="com.jdon.framework.samples.jpeteststore.domain.Order">  
    <actionForm name="orderForm"/>  
    <handler>  
        <service ref="orderService">  
            <createMethod name="insertOrder"/>  
        </service>  
    </handler>  
</model>
```

第六步：根据逐个实现界面功能，订单的第一个功能创建一个新的订单，在新订单页面 NewOrderForm.jsp 推出之前，这个页面的 ActionForm 已经被初始化，是根据购物车等 Cart 其他 Model 数据初始化合成的。

新订单页面初始化

根据 Jdon 框架中 CRUD 功能实现，初始化一个 ActionForm 有两种方法：一继承 ModelHandler 实现 initForm 方法；第二通过 jdonframework.xml 的 initMethod 方法配置。

这两个方案选择依据是根据用来初始化的数据来源什么地方。

订单表单初始化实际是来自购物车信息或用户账号信息，这两个都事先保存在 HttpSession 中，购物车信息是通过有态 CartService 实现的，所以这些数据来源是和 request 相关，那么我们选择第一个方案。

继承 ModelHandler 之前，我们可以考虑首先继承 ModelHandler 的子类 XmlModelHandler，只要继承 initForm 一个方法即可，这样节省其他方法编写实现。

```
public class OrderHandler extends XmlModelHandler {  
  
    public ModelForm initForm(HttpServletRequest request) throws  
    Exception{  
        HttpSession session = request.getSession();  
        AccountForm accountForm = (AccountForm) session.getAttribute("accountForm");  
        OrderForm orderForm = createOrderForm(accountForm);  
        CartService cartService = (CartService)WebAppUtil.getService("cartService", request);  
  
        orderForm.setTotalPrice(cartService.getSubTotal());  
  
        //below can read from the user's creditCard service;
```

```

        orderForm.setCreditCard("999 9999 9999 9999");
        orderForm.setExpiryDate("12/03");
        orderForm.setCardType("Visa");
        orderForm.setCourier("UPS");
        orderForm.setLocale("CA");
        orderForm.setStatus("P");

        Iterator i = cartService.getAllCartItems().iterator();
        while (i.hasNext()) {
            CartItem cartItem = (CartItem) i.next();
            orderForm.addLineItem(cartItem);
        }
        return orderForm;
    }

    private OrderForm createOrderForm(AccountForm account){
        .....
    }
}

```

ModelHandler 的 initForm 继承后，因为这使用了 Jdon 的 CRUD 功能实现，这里我们只使用到 CRUD 中的创建功能，因此，findModelByKey 方法就无需实现，或者可以在 jdonframework.xml 中配置该方法实现。

考虑到在 initForm 执行后，需要推出一个 NewOrderForm.jsp 页面，这是一个新增性质的页面。所以在 struts-config.xml

```

<action path="/shop/newOrderForm" type="com.jdon.strutsutil.ModelViewAction"
name="orderForm" scope="request" validate="false">
    <forward name="create" path="/order/NewOrderForm.jsp"/>
</action>

```

### 订单确认流程

新的订单页面推出后，用户需要经过两个流程才能确认保存，这两个流程是填写送货地址以及再次完整确认。这两个流程实现的动作非常简单，就是将 OrderForm 中的 shippingAddressRequired 字段和 confirm 字段赋值，相当于简单的开关，这是一个很简单的动作，可以有两种方案：直接在 jsp 表单中将这两个值赋值；直接使用 struts 的 Action 实现。后者需要编码，而且不是非有这个必要，只有第一个方案行不通时才被迫实现。

第一步：填写送货地址

使用 [Jdon 框架的推出纯 Jsp 功能的 Action](#) 配置 struts-config.xml:

```

<action path="/shop/shippingForm" type="com.jdon.strutsutil.ForwardAction"
name="orderForm" scope="session" validate="false">
    <forward name="forward" path="/order/ShippingForm.jsp"/>
</action>

```

这是实现送货地址页面的填写，使用的还是 OrderForm。更改前面流程 NewOrderForm.jsp 中的表单提交 action 值为本 action path: shippingForm.shtml:

```

<html:form action="/shop/shippingForm.shtml" styleId="orderForm" method="post" >
    .....
</html:form>

```

在 ShippingForm.jsp 中增加将 shippingAddressRequired 赋值的字段：



```
<html:hidden name="orderForm" property="shippingAddressRequired" value="false"/>
```

第二步：确认订单

类似上述步骤，配置 struts-config.xml：

```
<action path="/shop/confirmOrderForm" type="com.jdon.strutsutil. ForwardAction"
    name="orderForm" scope="session" validate="false">
    <forward name="forward" path="/order/ConfirmOrder.jsp"/>
</action>
```

将上一步 ShippingForm.jsp 的表单 action 改为本 action 的 path: confirmOrderForm.shtml:

```
<html:form action="/shop/confirmOrderForm.shtml" styleId="orderBean" method="post" >
```

修改 ConfirmOrder.jsp 中提交的表单为最后一步，保存订单 newOrder.shtml:

```
<html:link page="/shop/newOrder.shtml?confirmed=true"></html:link>
```

第三步：下面是创建数据保存功能实现：

```
<action path="/shop/newOrder" type="com.jdon.strutsutil.ModelSaveAction"
    name="orderForm" scope="session"
    validate="true" input="/order/NewOrderForm.jsp">
    <forward name="success" path="/order/ViewOrder.jsp"/>
</action>
```

ModelSaveAction 是委托 ModelHandler 实现的，这里有两种方式：配置方式：在 jdonframework.xml 中配置了方法插入；第二种是实现代码，这里原本可以使用配置方式实现，但是因为在功能上有要求：在订单保存后，需要清除购物车数据，因此只能使用代码实现方式，在 ModelHandler 中实现子类方法 serviceAction:

```
public void serviceAction(EventModel em, HttpServletRequest request) throws java.lang.Exception {
    try {
        CartService cartService = (CartService) WebAppUtil.getService("cartService", request);
        cartService.clear(); //清楚购物车数据

        OrderService orderService = (OrderService) WebAppUtil.getEJBService("orderService",
request);
        switch (em.getActionType()) {
            case Event.CREATE:
                Order order = (Order) em.getModel();
                orderService.insertOrder(order);
                cartService.clear();
                break;
            case Event.EDIT:
                break;
            case Event.DELETE:
                break;
        }
    } catch (Exception ex) {
        throw new Exception(" serviceAction Error:" + ex);
    }
}
```

## 用户订单列表

用户查询自己的订单列表功能很明显可以使用 Jdon 框架的批量查询事先。  
在 struts-config.xml 中配置 ModelListForm 如下：

```
<form-bean name="listForm" type="com.jdon.strutsutil.ModelListForm"/>
```

建立继承 ModelListAction 子类 OrderListAction：

```
public class OrderListAction extends ModelListAction {

    public PageIterator getPageIterator(HttpServletRequest request, int start, int count) {
        OrderService orderService = (OrderService) WebAppUtil.getService("orderService",
request);
        HttpSession session = request.getSession();
        AccountForm accountForm = (AccountForm) session.getAttribute("accountForm");
        if (accountForm == null) return new PageIterator();
        return orderService.getOrdersByUsername(accountForm.getUsername(), start, count);
    }

    public Model findModelByKey(HttpServletRequest request, Object key) {
        OrderService orderService = (OrderService) WebAppUtil.getService("orderService",
request);
        return orderService.getOrder((Integer)key);
    }

}
```

修改 OrderService，将获得 Order 集合方法改为：

```
public class OrderService{

    PageIterator getOrdersByUsername(String username, int start, int count)
    ....
}
```

根据 Jdon 批量查询要求，使用 iBatis 实现返回 ID 集合以及符合条件的总数。  
最后编写 ListOrders.jsp，两个语法：logic:iterator 和 MultiPages

## 配置 jdon 框架启动

目前我们有四个 struts-config.xml，前面每个模块一个配置：

/WEB-INF/struts-config.xml 主配置

/WEB-INF/struts-config-catalog.xml 商品相关配置

/WEB-INF/struts-config-security.xml 用户相关配置

/WEB-INF/struts-config-cart.xml 购物车相关配置

/WEB-INF/struts-config-order.xml 订单相关配置

本项目只有一个 jdonframework.xml，当然我们也可以创建多个 jdonframework.xml，  
然后在其 struts-config.xml 中配置。

```
<plug-in className="com.jdon.strutsutil.InitPlugIn">
    <set-property property="modelmapping-config" value="jdonframework_iBATIS.xml" />
</plug-in>
```

## 修改 iBatis 的 DAO 配置

iBatis 4.0.5 中原来的配置过于扩张（从持久层扩张到业务层），AccountDao 每个实例获得都需要通过 daoManager.getDao 这样形式，而使用 Jdon 框架后，AccountDao 等 DAO 实例获得无需特别语句，我们只要在 AccountService 直接引用 AccountDao 接口，至于 AccountDao 的具体实例，通过 Ioc 注射进入即可。

因此，在 jdonframework.xml 中有如下配置：

```
<pojoService name="accountDao"
class="com.jdon.framework.samples.jpeteststore.persistence.dao.sqlmapdao.AccountSqlMapDao"/>
<pojoService name="accountService"
class="com.jdon.framework.samples.jpeteststore.service.bo.AccountServiceImpl"/>
<pojoService name="productManager"
class="com.jdon.framework.samples.jpeteststore.service.bo.ProductManagerImp"/>
```

而 AccountServiceImpl 代码如下：

```
public class AccountServiceImpl implements AccountService, Poolable {
    private AccountDao accountDao;
    private ProductManager productManager;

    public AccountServiceImpl(AccountDao accountDao,
                              ProductManager productManager){
        this.accountDao = accountDao;
        this.productManager = productManager;
    }
}
```

AccountServiceImpl 需要两个构造方法实例，这两个中有一个是 AccountDao。

按照 iBatis 原来的 AccountDao 子类 AccountSqlMapDao 有一个构造方法参数是 DaoManager，但是我们无法生成自己的 DaoManager 实例，因为 DaoManager 是由 dao.xml 配置文件读取后生成的，这是一个动态实例；那只有更改 AccountSqlMapDao 构造方法了。

根源在于 BaseSqlMapDao 类，BaseSqlMapDao 是一个类似 JDBC 模板类，每个 Dao 都继承它，现在我们修改 BaseSqlMapDao 如下：

```
public class BaseSqlMapDao extends DaoTemplate implements SqlMapExecutor{
    ....
}
```

BaseSqlMapDao 是 XML 配置和 JDBC 模板的结合体，在这个类中，这两者搭配在一起，在其中实现 SqlMapExecutor 各个子方法。

我们再创建一个 DaoManagerFactory，专门根据配置文件创建 DaoManager 实例：主要方法如下：

```
Reader reader = Resources.getResourceAsReader(daoResource);
daoManager = DaoManagerBuilder.buildDaoManager(reader);
```

其中 daoResource 是 dao.xml 配置文件，这个配置是在 jdonframework.xml 中配置：

```
<pojoService name="daoManagerFactory"
class="com.jdon.framework.samples.jpeteststore.persistence.dao.DaoManagerFactory">
    <constructor
value="com/jdon/framework/samples/jpeteststore/persistence/dao/sqlmapdao/sql/dao.xml"/>
```

>

```
</pojoService>
```

这样，我们可以通过改变 jdonframework.xml 配置改变 dao.xml 配置。

Dao.xml 配置如下：

```
<daoConfig>
  <context>
    <transactionManager type="SQLMAP">
      <property name="SqlMapConfigResource"
        value="com/jdon/framework/samples/jpetstore/persistence/dao/sqlmapdao/sql/sql-map-config.xml"/>
    </transactionManager>

    <dao interface="com.ibatis.sqlmap.client.SqlMapExecutor"
      implementation="com.jdon.framework.samples.jpetstore.persistence.dao.sqlmapdao.BaseSqlMapDao"/>

  </context>
</daoConfig>
```

在 dao.xml 中，我们只配置一个 JDBC 模板，而不是将所有的如 AccountDao 配置其中，因为我们需要 iBatis 只是它的 JDBC 模板，实现持久层方便的持久化，仅此而已！

DaoManagerFactory 为我们生产了 DaoManager 实例，那么如何赋值到 BaseSqlMapDao 中，我们设计一个创建 BaseSqlMapDao 工厂如下：

```
public class SqlMapDaoTemplateFactory {

    private DaoManagerFactory daoManagerFactory;

    public SqlMapDaoTemplateFactory(DaoManagerFactory daoManagerFactory) {
        this.daoManagerFactory = daoManagerFactory;
    }

    public SqlMapExecutor getSqlMapDaoTemp(){
        DaoManager daoManager = daoManagerFactory.getDaomanager();
        return (SqlMapExecutor)daoManager.getDao(SqlMapExecutor.class);
    }

}
```

通过 getSqlMapDaoTemp 方法，由 DaoManager.getDao 方法获得 BaseSqlMapDao 实例，BaseSqlMapDao 的接口是 SqlMapExecutor，这样我们通过 DaoManager 获得一个 JDBC 模板 SqlMapExecutor 的实例。

这样，在 AccountDao 各个子类 AccountSqlMapDao 中，我们只要通过 SqlMapDaoTemplateFactory 获得 SqlMapExecutor 实例，委托 SqlMapExecutor 实现 JDBC 操作，如下：

```
public class AccountSqlMapDao implements AccountDao {
    //iBatis 数据库操作模板
    private SqlMapExecutor sqlMapDaoTemplate;
    //构造方法
    public AccountSqlMapDao(SqlMapDaoTemplateFactory sqlMapDaoTemplateFactory) {
```

```

        sqlMapDaoTemplate = sqlMapDaoTemplateFactory.getSqlMapDaoTemp();
    }
    //查询数据库
    public Account getAccount(String username) throws SQLException{
        return (Account)sqlMapDaoTemplate.queryForObject("getAccountByUsername", username);
    }

```

## 部署调试

当在 JBoss 或 Tomcat 控制台 或者日志文件中出现下面字样标识 Jdon 框架安装启动成功:

```
<===== Jdon Framework started successfully! =====>
```

Jdon 框架启动成功后, 以后出现的错误基本是粗心大意的问题, 仔细分析会很快找到原因, 相反, 如果编程时仔细慢一点, 则后面错误出现概率很小。

使用 Jdon 框架开发 Jpetstore, 一次性调试通过率高, 一般问题都是存在数据库访问是否正常, 一旦正常, 主要页面就出来了, 其中常见问题是 jsp 页面和 ActionForm 的字段不对应, 如 jsp 页面显示如下错误:

No getter method available for property creditCardTypes for bean under name orderForm

表示在 OrderForm 中没有字段 creditCardTypes, 或者有此字段, 但是大小写错误等粗心问题。

如果 jsp 页面或后台 log 记录显示:

System error! please call system Admin.java.lang.Exception

一般这是由于前面出错导致, 根据记录向前搜索, 搜索到第一个出错记录:

```

2005-07-07      11:55:16,671      [http-8080-Processor25]      DEBUG
com.jdon.container.pico.PicoContainerWrapper - getComponentClass: name=orderService
2005-07-07      11:55:16,671      [http-8080-Processor25]      ERROR
com.jdon.aop.reflection.MethodConstructor - no this method name:insertOrder

```

第一个出错是在 MethodConstructor 报错, 没有 insertOrder 方法, 根据上面一行是 orderService, 那么检查 orderService 代码看看有无 insertOrder:

```

public interface OrderService {
    void insertOrder(Order order);
    Order getOrder(int orderId);
    List getOrdersByUsername(String username);
}

```

OrderService 接口中是有 insertOrder 方法, 那么为什么报错没有呢? 仔细检查一下, 是不是 insertOrder 的方法参数有问题, 哦, 因为 orderService 的调用是通过 jdonframework.xml 下面配置进行的:

```

<model key="orderId"
    class="com.jdon.framework.samples.jpetstore.domain.Order">
    <actionForm name="orderForm"/>
    <handler>
        <service ref="orderService">

```

```

        <createMethod name="insertOrder"/>
    </service>
</handler>
</model>

```

而根据 Jdon 框架要求，使用配置实现 ModelHandler，则要求 OrderService 的 insertOrder 方法参数必须是 EventModel，更改 OrderService 的 insertOrder 方法如下：

```

public interface OrderService {
    void insertOrder(EventModel em);
}

```

同时，修改 OrderService 的子类代码 OrderServiceImp：

```

public void insertOrder(EventModel em) {
    Order order = (Order)em.getModel();
    try{
        orderDao.insertOrder(order);
    }catch(Exception daoe){
        Debug.logError(" Dao error : " + daoe, module);
        em.setErrors("db.error");
    }
}

```

注意 em.setErrors 方法，该方法可向 struts 页面显示出错信息，db.error 是在本项目的 application.properties 中配置的。

关于本次页面出错问题，还有更深缘由，因为我们在 jdonframework.xml 中定义了 createMethod，而根据前面已经有 ModelHandler 子类代码 OrderHandler 实现，所以，这里不用配置实现，jdonframework.xml 的配置应该如下：

```

    <model key="orderId"
        class="com.jdon.framework.samples.jpetestore.domain.Order">
        <actionForm name="orderForm"/>
        <handler
class="com.jdon.framework.samples.jpetestore.presentation.action.OrderHandler"/>
    </model>

```

直接定义了 handler 的 class 是 OrderHandler，在 OrderHandler 中的 serviceAction 我们使用代码调用了 OrderService 的 insertOrder 方法，如果使用这样代码调用，无需要求 OrderService 的 insertOrder 的参数是 EventModel 了。

## 总结

Jpetstore 整个开发大部分基于 Jdon 框架开发，特别是表现层，很少直接接触使用 struts 原来功能，Jdon 框架的表现层架构基本让程序员远离了 struts 的烦琐开发过程，又保证了 struts 的 MVC 实现。

Jpetstore 中只有 SignonAction 这个类是直接继承 struts 的 DispatchAction，这个功能如果使用基于 J2EE 容器的安全认证实现（见 JdonNews），那么 Jpetstore 全部没有用到 struts 的 Action，无需编写 Action 代码；ActionForm 又都是 Model 的拷贝，Action 和 ActionForm 是 struts 编码的两个主要部分，这两个部分被 Jdon 框架节省后，整个 J2EE

的 Web 层开发方便快捷，而且容易得多。

这说明 Jdon 框架确实是一款快速开发 J2EE 工具，而且是非常轻量的。

纵观 Jpetstore 系统，主要有三个层的配置文件组成，持久层由 iBatis 的 Product.xml 等配置文件组成；服务层由 jdon 框架的 jdonframework.xml 组成；表现层由 struts 的 struts-config.xml 和 jdonframework.xml 组成；剩余代码基本是 Model 之类实现。

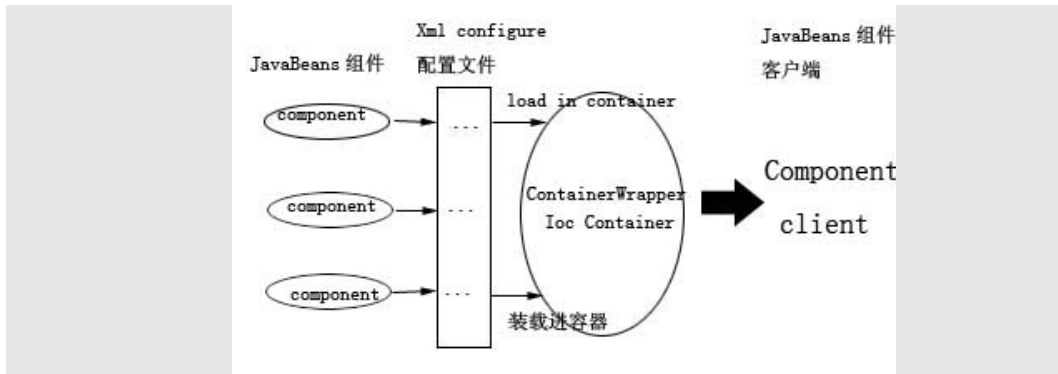




## Jdon 框架核心设计架构

### 架构设计

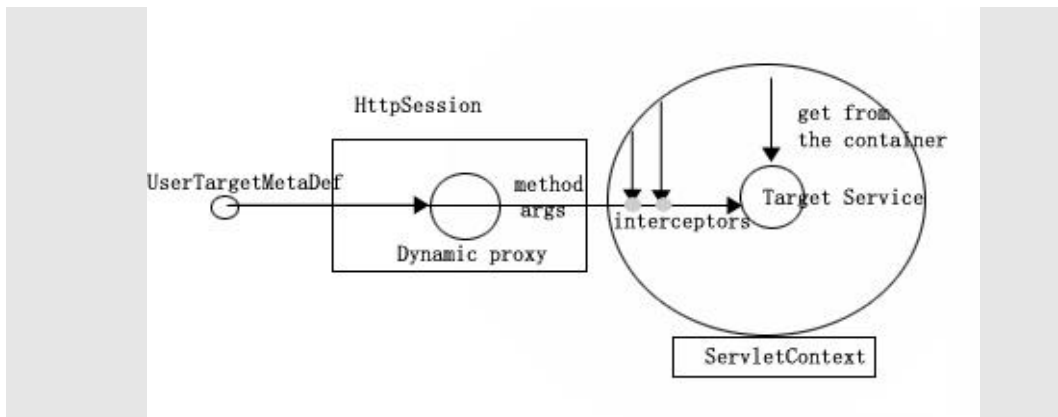
由于整个框架中组件基于 Ioc 实现，组件类之间基本达到完全解耦。



从上图中可以看出，任何 JavaBeans 组件只要在 XML 配置文件（container.xml aspect.xml 和 jdonframework.xml）中配置，由容器装载机制注册到 Ioc 容器中，这些组件的客户端，也就是访问者，只要和微容器（或者组件的接口）打交道，就可以实现组件访问。

因此，本框架中每个功能块都可从框架中分解出来，单独使用它们。用户自己的任意功能块也可以加入框架中，Jdon 框架成为一种完全开放、伸缩自如的基础平台。

运行原理图：



当客户端请求产生时，首先将访问目标服务参数包装在 userTargetMetaDef 对象中，该对象访问位于 HttpSession 中的动态代理实例，被动态代理将目标服务肢解成 Method 和方法参数 Args，然后被拦截器拦截，最后达到目标服务实例，当然，有的拦截器可能就此完全拦截（堵住去路，如 PoolInterceptor），目标服务实例是事先被注册到容器中的，在被访问时，缺省情况是每次访问产生一个新的目标服务实例。

Jdon 容器如图是被保存在 Servlet 容器的 ServletContext 中的。

### 包结构

Jdon 框架有下列包名组成：

**AOP** : 表示 AOP 相关功能的类, 其与 bussinessproxy 包关系最紧密, 两者是系统的核心包。

**Bussinessproxy**: 与动态代理、目标服务相关的类。

**Container**: 实现将组件装载入容器, 这是与 Ioc 微容器相关的类, 缺省使用了 PicoContainer, 但是可以替换的。包括容器的组件注册、配置读取、组件访问等功能。以上三个包可从 Jdon 框架中独立出来。

**Controller**: 这是一些基础功能类、以及和应用相关类的功能包, 该包是 Container 的客户端。

**Model**: 这是和数据模型增删改查 (CRUD)。批量查询、缓存优化等相关功能的类。

**Security**: 这是和用户注册登陆相关功能类,

**ServiceLocator**: 这是和 EJB 运行定位相关的类

**Strutsutil**: 与 Struts 相关类, 前面 Model 包中各种功能必须通过具体表现层技术实现。

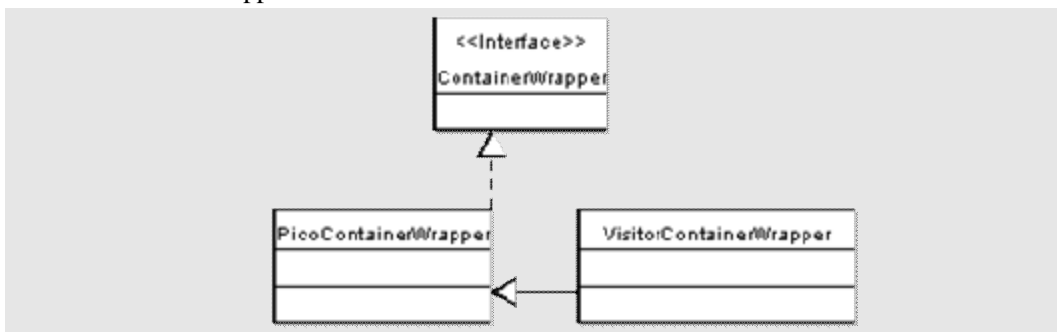
**Util**: 一些工具类。

## container 包

Container 包主要是负责容器管理方面的功能, 其他包中的组件都被写入配置文件 container.xml、aspect.xml 和 jdonframework.xml 中, 而 container 包主要负责与这些配置文件打交道, 从配置文件中获得其他包的组件, 向容器中注册, 并启动容器。

主要一个接口是 ContainerWrapper, ContainerWrapper 有两个主要方法: 向容器注册组件; 从容器查询获得组件。

ContainerWrapper 接口的缺省实现是 PicoContainerWrapper 和 VisitorContainerWrapper 两个子类, 如下图:



PicoContainerWrapper 使用了著名的 PicoContainer 实现 (<http://www.picocontainer.org>), ContainerWrapper 接口主要从其抽象出来。

访问者模式

VisitorContainerWrapper 是观察者模式的实现, 传统观察者模式中, 对于 Visitor 角色一般下面多个访问不同被访问者的方法:

visitAcomponent();

visitBcomponent();

.....

由于 Acomponent、Bcomponent 这些类已经注册到容器, 因此, 通过容器可以直接实现不同组件的访问, 只需通过下面一个方法实现:

```
public ComponentVisitor getComponentVisitor(){
    return new ComponentOriginalVisitor(this);
}
```

而访问者 **Visitor** 则可以节省为单一方法即可：

```
visit(XXX xxx);
```

使用访问者模式的原因：主要为了实现缓存，提高一些组件运行性能，如果一些组件每次访问时，都需要 **new**，例如 **Proxy.newInstance** 如果频繁反复运行，将是十分耗费性能的，因此，使用缓存尽量避免每次创建将提高系统的运行性能。

**Visitor** 有两个子类实现：**ComponentOriginalVisitor** 和 **HttpSessionProxyVisitor**，这里又使用了装饰者模式，**Decoratee** 是 **ComponentOriginalVisitor**；而 **Decorator** 是 **HttpSessionProxyVisitor**，**HttpSessionProxyVisitor** 是 **HttpSessionBindingListener**，也就是说，我们使用了 **HttpSession** 作为缓存机制，**HttpSession** 的特点是以用户为 **Key** 的缓存，符合  $\delta$  的缓存机制，当然，我们以后也可以使用更好的缓存机制替换 **HttpSession**，替换了 **HttpSession** 根本不必涉及到其他类的更好，因为这里使用模式实现了彻底的解耦。

访问者模式另外一个重要角色：**Visitable**，它则是那些运行结果需要缓存的组件必须继承的，注意，这里有一个重点：不是那些组件本省生成需要缓存，而是它的运行结果需要缓存的。继承 **Visitable** 这些组件事先必须注册在容器中。

目前 **Visitable** 有两个子类：

```
* @see com.jdon.bussinessproxy.dyncproxy.ProxyInstanceFactoryVisitable
```

```
* @see com.jdon.bussinessproxy.target.TargetServiceFactoryVisitable
```

前者主要是动态代理的创建，因为 **Proxy.newInstance** 频繁执行比较耗费性能，第一次创建后，将动态代理实例保存在 **httpSession** 中，当然每个 **Service** 对应一个动态代理实例。

**TargetServiceFactoryVisitable** 主要是为了缓存那些目标服务的实例，目前这个功能没有激活，特殊情况下才推荐激活。

## 容器的启动

容器启动主要是将配置文件中注册的组件注册到容器中，并启动容器，这涉及到 **container** 包下 **Config** 和 **Builder** 等几个子包。

**Config** 包主要负责从 **container.xml** 和 **aspect.xml** 读取组件；

**Builder** 包主要负责向 **ContainerWrapper** 注册这些组件，注册过程是首先从基础组件（**container.xml**）开始，然后是拦截器组件(**aspect.xml**)，最后是用户的服务组件(**jdonframework.xml**)。

## 容器的生命周期

容器启动后，容器本身实例是放置在 **Web** 容器的 **ServletContext** 中。

容器启动并不是在应用系统部署到 **Web** 容器时就立即启动，而是该应用系统被第一次访问时触发启动，这个行为是由 **ContainerSetupScript** 的 **startup** 触发的，而 **startup** 方法则是由 **ServletContainerFinder** 的 **findContainer** 触发。

当应用系统从 **Web** 容器中销毁或停止，**Jdon** 框架容器也就此销毁，最好将你的组件中一些对象引用释放，只要继承 **Startable**，实现 **stop** 方法即可。

## 容器的使用

客户端访问组件必需通过容器进行，这个过程分两步：

从 Web 容器中获得框架容器 ContainerWrapper 实例。

从 ContainerWrapper 容器中查询获得组件实例，有两者实例方式：单例和多例。

关于具体使用方式可见前面章节“如何获得 POJO 实例”等。

这个容器使用过程是通过 finder 包下面的类完成，主要是 ServletContainerFinder 类，ComponentKeys 类保存中一些 container.xml 中配置的组件名称，这些组件名称可能需要在框架程序中使用到，这就需要引起注意，这个类中涉及的组件名称不要在 container.xml 中随意改动（当然这个类在以后重整中争取去除）。

com.jdon.controller.WebAppUtil 是专门用于客户端对微容器的访问，这个客户端目前是 Http 客户端，主要方法：

```
public static Object getService(String name, HttpServletRequest request)
```

这是最常用的从容器中获得服务实例的方法。需要 HttpServletRequest 作为客户端，Jdon 框架将来会提供作为 Application 客户端调用的专门方法。

```
public static Object getComponentInstance(String name, HttpServletRequest request)
```

这是获得组件实例的方法。

```
public static String getContainerKey()
```

通过获得容器保存在 ServletContext 中的 Key。

## 从容器内部访问容器

上节“容器的使用”是指从容器外部（客户端）如何访问和使用容器，如果当前客户端是在容器内部，例如需要在一个 Service 服务类中访问容器的缓存机制等，该如何访问？

使用 com.jdon.container.finder.ContainerCallback，同时，该服务 POJO 类以 ContainerCallback 作为构造参数，当该 POJO 服务类注册到容器中时，容器的 Ioc 特性将会找到事先以及注册的 ContainerCallback 类。

通过 ContainerCallback 获得 ContainerWrapper 容器实例，然后通过 ContainerWrapper 下面两个方法：

```
public Object lookup(String name);
```

从容器中获得 container.xml 中注册的组件实例，这种方法每次调用获得的是同一个实例，相当于单例方式获得。

```
public Object getComponentNewInstance(String name);
```

每次从容器中获得 container.xml 中注册的组件的一个新实例，这种方法每次调用获得的是新的实例，该方法不适合一些单例资源的获得。

例如如果向在 POJOService 中访问容器的缓存机制，因为缓存整个容器只能有一处，因此必须以 lookup 方式获得，获得 ModelManager 的实例如下：

```
ModelManager modelManager = containerWrapper.lookup("modelManager");
```

其中“modelManager”字符串名称是从 Jdon 框架的 jdonFramework.jar 包中 META-INF 的 container.xml 中查询获知的。

## AOP 包

Jdon AOP 的设计目前功能比较简单，不包括标准 AOP 中的 Mixin 和 Introduction 等功能；AOP 包下有几个子包：Interceptor、joinpoint 和 reflection，分别涉及拦截器、拦截器切点和反射机制等功能。

## bussinessproxy 包

Jdon 框架从架构角度来说，它是一种业务代理，前台表现层通过业务代理层访问业务服务层，使用 AOP 实现业务代理是一种新的趋势，因此本包功能是和 AOP 包功能交互融合的。

bussinessproxy 包中的 config 子包是获取 jdonframework.xml 中的两种服务 EJB 或 POJO 配置，然后生成 meta 子包中 TargetMetaDef 定义。

target 子包封装了关于这两种服务由元定义实现对象创建的工厂功能，服务对象创建透过访问模式使用了 HttpSession 作为 EJB 实例缓存，这个功能主要是为了 EJB 中有态会话 Bean 实现，这样，客户端通过 getService 获得一个有态会话 Bean 时，无需自行考虑保存这个 Bean 引用到 HttpSession 中这一使用细节了。无态会话 bean 从实战角度发现也可以缓存，提高了性能。

Jdon 框架既然是一种业务代理，那么应该服务于各种前台表现层，Jdon 框架业务代理还适合客户端为 Java 客户端情况下的业务服务访问。这部分功能主要是 com.jdon.bussinessproxy.remote 子包实现的。使用方式参考：<http://www.jdon.com/product/ejbinvoker.htm>

### TargetMetaDef

TargetMetaDef 是目标服务的元定义，TargetMetaDef 主要实现有两种：EJBTargetMetaDef 和 POJOTargetMetaDef，分别是 EJB 服务和 POJO 服务实现。

所谓目标服务元定义也就是程序员在 jdonframework.xml 中定义的 services。

Jdon 框架提供目标服务两种形式：目标服务实例创建和目标服务本身。

### ServiceFactory

目标服务创建工厂 ServiceFactory 可以根据目标服务元定义创建一个目标服务实例，这也是 WebAppUtil.getService 获得目标服务的原理实现，

com.jdon.controller.service.DefaultServiceFactory 是 ServiceFactory 缺省实现，在 DefaultServiceFactory 中，主要是通过动态代理获得一个服务实例，使用了 DefaultServiceFactory 来实现动态代理对象的缓存。不必每次使用 Proxy.newInstance 第一次执行后，将其结果保存在 HttpSession 中。

### Service

目标服务 Service 则是通过方法 Relection 运行目标服务，只要告知 Service 目标服务的类、类的方法、方法参数类型和方法参数值这些定义，除了方法参数值，其余都是字符串定义，这是 Java 的另外一种调用方式。

## security 包

Security 包主要应用于 JdonSD（一个组件库）产品的基于容器的安全权限认证。

## strutsutil 包

本包封装了 Jdon 框架基于 Struts 的表现层实现，Jdon 框架的 CRUD 功能和批量分页查询以及树形显示功能都是在该包中实现，还有上传图片等功能（待测试）。

## controller 包

controller 包是 Jdon 框架的整体结构包，主要面向框架使用客户端，其中 WebAppUtil 是经常被调用的类。

cache 子包是框架的缓存机制，这个缓存机制主要面向数据类设计，例如 Model 类，功能类是通过 container 实现管理，功能类使用访问者模式实现结合 HttpSession 实现有状态。

Jdon 框架缺省缓存是使用了开源 OfBiz 的一个缓存，如果你希望使用更好的缓存产品。可以通过继承 Cache 或 LRUCache，然后在 container.xml 中替代如下配置即可：

```
<component name="cache" class="com.jdon.controller.cache.LRUCache" >
    <constructor value="cache.xml"/>
</component>
```

cache 子包提供的缓存不但为 Model 服务，还为批量查询的 PageIterator 创建服务，可参见 PageIteratorSolver 类；另外也为上传图片缓存服务，以后可拓展为更多数据性缓存服务。

Config 子包主要是为读取 pojoService 和 ejbService 配置实现 XML 读取功能。

Events 子包包含的是装载数据 Model 载体在各层之间穿梭，类似信使，通常表现层会将 Model 封装到 EventModel 中，传给服务层的服务，服务处理完毕，如果出错将出错信息封装在 EventModel 对象中，这样表现层可从 EventModel 中查知后台处理是否出错，并返回出错信息。

EventModel 的 setErrors 方法是用于服务层封装处理出错信息的方法，可直接将”id.notfound”这样约定出错信息赋值进去，前台如 struts 可根据 id.notfound 到 Application.properties 中查找本地化信息提示。

Model 子包是封装了框架重要的数据类 Model，Model 既是域模型中的 Model，也是一种 DTO，它是在各层之间传送数据的载体，因此 Model 可以允许嵌套 Model，DynamicModel 也许更加适合临时组装一个传送对象。

Model 是框架的 CRUD 功能重要角色；PageIterator 是框架批量查询的重要角色。

Pool 子包封装了 Apache Pool 功能，主要用于 Pool 拦截器，如果你的 POJO 服务需要池化提升性能，只要让你的 POJO 服务类继承该包下的 Poolable 或加入 @Poolable 元注解 Annotation 即可。

Service 子包封装框架两大有关服务的重要功能：创建一个服务 `ServiceFactory.create`；或直接运行一个服务 `Service.execute`。

## model 包

model 包主要是围绕 Model 的 CRUD 功能实现展开，这部分是中间层，尽量做到和表现层 `strutsutil` 包实现解耦，这样如果有新的表现层如 JSF 可直接和本包发生作用，实现 JSF 下的 CRUD 功能。

`ModelManager` 是面向表现层操作 Model 的主要客户端类，`ModelForm` 和 `ModelHandler` 都是可能需要客户端继承实现的重要类。

`Config` 子包主要是从 `jdonframework.xml` 中获得 models 相关配置，将这些加载到 `ModelMapping` 对象中。

`Handler` 子包是延续 `config` 子包，model 配置中 `Modelhandler` 部分配置被加载到 `HandlerMetaDef` 对象中，然后预先实现 `class.forName` 运行，`HandlerObjectFactory` 是创建 `ModelHandler` 实例工厂，一次性生产 20 个，如果不够用，还可以再生产 20 个，因为 `ModelHandler` 为 struts 的 action 直接调用，而 action 是多线程，所以使用这种池形式可提高性能，这里使用了自制的池功能，原理简单，池性能好，简单性能就好是 Jdon 框架设计一个原则。

`Query` 子包主要是为批量查询服务，这些类在持久层被调用。

## servicelocator 和 Util 包

这两个包都是工具，大部分从其它开源借鉴过来，正是因为这些经过实践证明运行良好的基础组件，才使得 Jdon 框架的整体基础牢固，感谢他们。

站在巨人的基础上发展是 Jdon 框架设计的另外一个原则。

更详细的描述将在专门 Jdon 框架设计文档中描述。

## Jdon 框架高级使用

从前面章节已经清楚：Jdon 框架快速开发主要体现在 CRUD 和批量查询这两个基本功能上，大量数据库系统基本都由这两个基本功能组成。

Jdon 框架的这两个功能横跨 J2EE 多个层次，主要简化工作体现在表现层，这一层主要是简化了 Struts 一些烦琐的配置和代码过程。

在这一章，主要介绍在深入使用 Jdon 框架中碰到的一些问题和解决方案，从而达到更加灵活地使用 Jdon 框架。

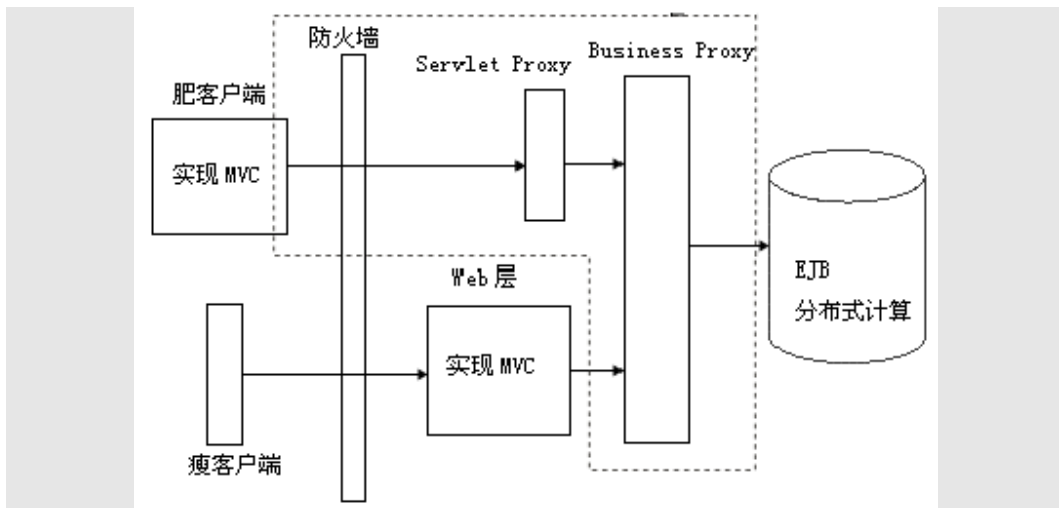
## C/S 架构下胖客户端远程调用

Jdon 框架不但是一个 B/S 架构框架，还支持 C/S 架构，客户端可以是 Java 类型的任何客户端，如 Swing、Applet 和 SWT 等。

使用 Jdon 框架，可以在无需改变业务服务层代码的情况下，将这些服务 Service 提供给远程 Java 客户端调用，这是一种基于 Http 协议的 RPC，它区别于 Web 服务的最大不同是：Jdon 框架将客户端请求打包对象传送；而不是转换成 XML；省却了两端转换解析步骤，提高了性能。

有两种使用方式，一种见下面，还有一种更方便方式，使用 Hessian，可见 [Jdon 框架 5.8 版本增加功能](#)。

架构原理图如下：



客户端配置

客户端调用远程服务器的服务核心代码很简单，如下：

```
MySessionLocal mySessionLocal = (MySessionLocal)serviceFactory.getService(
    FrameworkServices.MySessionEJB);
mySessionLocal.insert();
```

客户端调用服务就如同与服务在同一个 JVM 中一样，但是因为客户端在远程，所



以需要进行对远程服务器的一些配置，比如规定远程服务器的 IP 地址和 Servlet Prox 名称。

具体客户端代码可见框架案例 Samples 中的 remoteClient 代码。

## 服务器端配置

首先激活框架系统的远程访问，在 web.xml 加入：

```
<servlet>
  <servlet-name>EJBInvokerServlet</servlet-name>
  <display-name>接受远程客户端访问</display-name>
  <servlet-class>com.jdon.bussinessproxy.remote.http.InvokerServlet</servlet-class>
  <init-param>
    <param-name>configList</param-name>
    <param-value>jdonframework.xml</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>EJBInvokerServlet</servlet-name>
  <url-pattern>/auth/EJBInvokerServlet</url-pattern>
</servlet-mapping>
```

然后激活容器基于 HTTP 的基本验证，配置 web.xml：

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>SecurityRealm</realm-name>
</login-config>
```

在 web.xml 中定义访问权限如下：

```
<security-constraint>
  <display-name>User protected</display-name>
  <web-resource-collection>
    <web-resource-name>Collection1</web-resource-name>
    <url-pattern>/auth/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>User</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

这样，访问/auth /EJBInvokerServlet 将得到授权保护。下面在 web.xml 中分别配置该系统的角色和 JNDI 配置。

```
<security-role>
  <role-name>Admin</role-name>
</security-role>
<security-role>
```

```

    <role-name>User</role-name>
  </security-role>
  <ejb-local-ref>
    <ejb-ref-name>ejb/EJBControllerLocal</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>com.jdon.security.auth.ejb.SecurityFacadeLocalHome</local-home>
    <local>com.jdon.security.auth.ejb.SecurityFacadeLocal</local>
    <ejb-link>SecurityFacade</ejb-link>
  </ejb-local-ref>

```

在 jboss-web.xml 配置中使用容器安全机制:

```

<jboss-web>
  <security-domain>java:/jaas/SecurityRealm</security-domain>
</jboss-web>

```

然后在 JBoss 的 login-config.xml 中配置 SecurityRealm

## Struts+Jdon 表现层组件编程

通常, 一个 Jsp 页面可能显示很多各种内容, 涉及到肯定不只一个数据库, 也就是多个组件的结合, 但是 Struts 配置文件中一般又是一个 Action 一个 ActionForm 一个 Jsp, 他们之间的关系是 1:1:1。

如何在这样一个体系下实现灵活的一个页面多个组件编程模式呢?

首先, 要区分两种情况: 一个 Action 一个 ActionForm, 一个 ActionForm 中装载多个组件; 一个 Action 一个 ActionForm 一个组件;

一个 ActionForm 多个组件

第一种情况: 如果同一个 Action 就可以对付这些组件, 那么在这种情况下又有两个办法:

1.将这多个组件装入一个 ActionForm 中, 如使用 MapForm 等机制;

一般多个组件在同一个 Jsp 页面显示情况比较经常发生在批量查询中, 对于批量查询, 因为 Jdon 框架规定必须有一个 ModelListForm, 在 ModelListForm 中最多只能装载一个 Model, 如果你有多个组件, 也就是多个 Model, 如何装入 ModelListForm 呢?

可以使用 Jdon 框架的 com.jdon.controller.model.DynamicModel 装载多个 Model, 其实 DynamicModel 中有一个 Map, 然后将 DynamicModel 装入 ModelListForm 的 oneModel 中。

Struts 对于 Map 可以直接访问, 以 DynamicModel 为例子:

Action 中代码:

```

DynamicModel dynamicModel = new DynamicModel();
Test test = new Test();
test.setName("oooooooooooooooooooo");
//装入 Test 组件, 可以装入更多其他组件。”test”为 Test 实例的 key 值
dynamicModel.put("test", test);

```

```
modelListForm.setOneModel(dynamicModel);
```

Jsp 中显示代码:

```
<bean:define id="dynamicModel" name="modelListForm" property="oneModel"/>
<bean:define id="test" name="dynamicModel" property="value(test)"/>
<bean:write name="test" property="name"/>
```

其中 value(test)是这样解释: value 是 dynamicModel 方法 getValue 方法的简写, test 是之间保存 Test 实例的 key 值, 然后从 test 对象中取出其属性 name 的值。

2.手工将多个组件装入 request/session 等 scope 中, 然后根据其名称在 jsp 中获得。这种方式比较传统和原始, 类似 Jsp 编程方式, 学习难度小, 方便。

一个 ActionForm 一个组件

前面情况, 多个组件装载在一个 ActionForm 中, 隐含的意思实际是: 所有功能都可以在一个 Action 中实现, 在这个 Action 中, 我们将多个组件装入当前对应的 ActionForm, 但是, 如果这些组件必须有预先由不同的 Action 来处理, 每个组件必须经过 Action --> ActionForm 流程, 那么在这种情况下也有两种办法:

1. 使用 Tiles, 不同流程输出到同一个页面的不同区域。是一种并行处理方式。
2. 对多个流程首尾相连, 第一 Action forward 结果是第二个 Action, 最后输出一个 Jsp, 在这个 jsp 中就可以使用前面多个流程的多个 ActionForm 了, 这属于串行方式。

例如, 当前页面是使用 Jdon 框架的批量查询实现, Jdon 框架的批量查询一定是一个 ModelListAction 对应一个 ModelListForm, 这是一个比较固定的流程了。如果在当前这个批量查询页面中还要显示其他 Model 的批量查询, 也就是两个或多个 Model 的批量查询在同一个 Jsp 页面中显示, 在这种情况下有上述两个方案选择。

第一个方案前提是你必须使用 Struts+Tiles 架构, 使用 Tiles 可以将页面划分成任意块, 这样多个批量查询在页面任何位置以任何方式组合显示, 完全灵活, 缺点是加入 Tiles 概念。

在一般小型应用中, 我们只需采取第二方案, 将多个批量查询的 Action 的 struts-config.xml 配置进行首尾连接既可, 这种方式类似是按照 Jsp 页面先上后下的顺序串联的。

例如: threadPrevNexListForm 和 messageListForm 是两个批量查询的 ModelListForm, 那么 struts-config.xml 配置如下:

```
<action path="/message/messageList"
        type="com.jdon.jivejdon.presentation.action.ThreadPrevNexListAction"
        name="threadPrevNexListForm" scope="request"
        validate="false">
    <forward name="success" path="/message/messageListBody.shtml"/>
</action>

<action path="/message/messageListBody"
        type="com.jdon.jivejdon.presentation.action.MessageListAction"
        name="messageListForm" scope="request"
        validate="false">
    <forward name="success" path="/message/messageList.jsp"/>
</action>
```

这样, 当 url 调用/message/messageList.shtml 时, 最后导出/message/messageList.jsp 这个页面, 在 messageList.jsp 中可以访问 threadPrevNexListForm 和 messageListForm 两

个 ActionForm 了。

上述组件构造方法可应用在复杂的 Master-Details 编程中，将 Master 看成一个对象 Model，Details 是一系列子 Model 的集合。

## 内嵌对象(Embedded Object)缓存设计

请看下面这段代码：

```
public class Category extends Model{
    String Id;
    Product product;    //内嵌包含了一个 Product 对象
}
```

Category 这个 Model 内嵌了 Product 这个 Model，属于一种关联关系。

目前 Jdon 框架提供的缺省缓存是扁平式，不是嵌入式或树形的（当然也可以使用 JbossCache 等树形缓存替代），因此，一个 ModelB 对象（如 Product）被嵌入到另外一个 ModelA 对象（如 Category）中，那么这个 ModelB 对象随着 ModelA 对象被 Jdon 框架一起缓存。

假设实现 ModelA 已经在缓存中，如果客户端从缓存直接获取 ModelB，缓存由于不知道 ModelB 被缓存到 ModelA 中（EJB3 实体 Bean 中是通过 Annotation 标注字段），那么缓存可能告知客户端没有 ModelB 缓存。那么有可能客户端会再向缓存中加一个新的 ModelB，这样同一个 ID 可能有两份 ModelB，当客户端直接调用的 ModelB 进行其中字段更新，那么包含在 ModelA 中的 ModelB 可能未被更新（因为 ModelA 没有字段更新）。

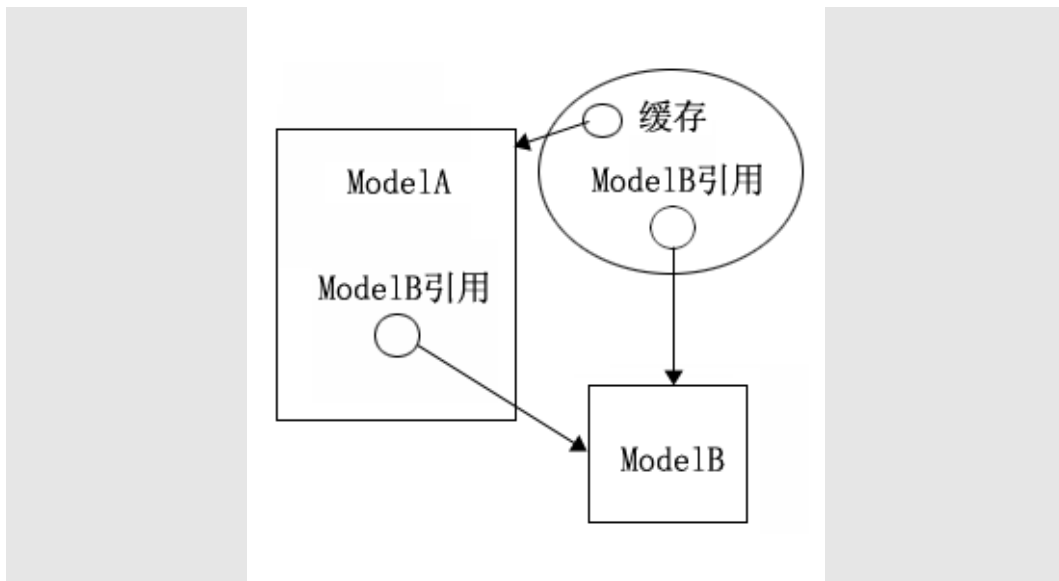
有两种解决方案：

第一. 最直接方式，通过手工清除 ModelA 缓存方式来更新，或者耐心等待 ModelA 缓存自动更新。手工清除缓存见下章。

注意：下面这种做法将也会导致不一致现象发生：

在 DAO 层读取数据库。生成 ModelA 时，直接读取数据库将 ModelB 充填。

第二.在进行 ModelA 和 ModelB 的相关操作服务设计时，就要注意保证这两种情况下 ModelB 指向的都是同一个。如下图：



为达到这个目的，只要在 Service 层和 Dao 层之间加一个缓存 Decorator，服务层向 Dao 层调用的任何 Model 对象都首先经过缓存检查，缓存中保存的 ModelA 中的 ModelB 是一个只有 ModelB 主键的空对象，在服务层 getModelA 方法中，再对 ModelA 的 ModelB 进行充实，填满，根据 ModelA 中的 ModelB 的主键，首先再到缓存查询，如果有，则将缓存中 ModelB 充填到 ModelA 的 ModelB 中，这样上图目的就可以实现了。

相关实现可参考 JiveJdon 3.0 的代码，Forum/ForumMessage 都属于这种情况。

## Model 缓存使用

Jdon 框架通过两种方式使用 Model 缓存：

CRUD 框架内部使用，如果你使用 Jdon 框架提供的 CRUD 功能，那么其已经内置 Model 缓存，而且会即时清除缓存。

通过 CacheInterceptor 缓存拦截器，如果你不使用 Jdon 框架的 CRUD 功能，缓存拦截器功能将激活，在向 Service 获取 Model 之前，首先查询当前缓存器中是否存在该 Model，如果有从缓存中获取。当你的 Model 中数值更改后，必须注意自己需要手工清除该 Model 缓存，清除方法如下介绍。

Jdon 框架除了提供单个 Model 缓存外，还在持久层 Dao 层提供了查询条件的缓存，例如如果你是根据某个字段按照如何排列等条件进行查询，这个查询条件将被缓存，这样，下次如果有相同查询条件，该查询条件将被提出，与其相关的满足查询条件一些结果（如符合条件总数等）也将被从缓存中提出，节省翻阅数据库的性能开销。

### 手工访问缓存

在一般情况下，前台表现层通过 getService 方法访问服务层一个服务，然后通过该服务 Service 获得一个 Model，这种情况 Jdon 框架的缓存拦截器将自动首先从缓存读取。

但是，有时我们在服务层编码时，需要获得一个 Model，在这种情况下，Jdon 框架

的缓存拦截器就不起作用，这时可能我们需要手工访问缓存。

因为所有服务类 POJO 都属于 Jdon 框架的容器内部组件，这实际是在容器内访问容器组件的问题。

使用 `com.jdon.container.finder.ContainerCallback`，同时，该服务 POJO 类以 `ContainerCallback` 作为构造参数，当该 POJO 服务类注册到容器中时，容器的 Ioc 特性将会找到事先以及注册的 `ContainerCallback` 类。

通过 `ContainerCallback` 获得 `ContainerWrapper` 容器实例，然后通过 `ContainerWrapper` 下面方法：

```
public Object lookup(String name);
```

从容器中获得 `container.xml` 中注册的组件实例，这种方法每次调用获得的是同一个实例，相当于单例方式获得。

```
ModelManager modelManager =
```

```
(ModelManager)containerWrapper.lookup("modelManager");
```

其中 “`modelManager`” 字符串名称是从 Jdon 框架的 `jdonFramework.jar` 包中 `META-INF` 的 `container.xml` 中查询获知的。

获得 `ModelManager` 后，我们基本可以访问 `Model` 有关的功能安排，如 `ModelManager` 的 `getCache` 方法。

下节的手工清除缓存中，我们是通过 `WebAppUtil.getComponentInstance` 获得 `ModelManager` 实例，这是一种从容器外获得容器组件的方式，本节介绍从容器内获得容器组件的方式，这两种方式可根据我们实际需要灵活使用，关键是弄清除你需要在哪里触发组件调用？

### 手工清除缓存

注意：手工清除缓存不是必要的，因为缓存中对象是有存在周期的，这在 `Cache.xml` 中设置的，过一段时间缓存将自动清除那些超过配置时间不用的对象，这样你修改的数据将被从数据库重新加载。如果你等不及这些内在变化，可以手工处理：

有两种情况需要手工清除缓存，首先，在持久层的 `Dao` 类中，总是需要手工清除查询条件的缓存（注意不是 `Model` 缓存，是查询条件的缓存），只要在相应的增删改方法中调用 `PageIteratorSolver` 的 `clearCache` 方法既可。

如果你不实行这种缓存清除，那么你更改一个 `Model` 数据或新增一个新的 `Model` 数据，你在批量查询时，将看不到任何变化：`Model` 数据没有被修改；新的 `Model` 没有出现在查询页面中。

其次，单个 `Model` 缓存在不使用 Jdon 框架的 `CRUD` 功能下也必须手工清除，如果你使用 1.2.3 以后版本，可以调用 `com.jdon.strutsutil.util.ModelUtil` 类的 `clearModelCache` 方法，该方法一般是 `Action` 中调用后台增删改服务之前被激活调用：

注意，手工清除 `Model` 缓存代码关键是：

```
modelManager.removeCache(keyValue);
```

`keyValue` 是 `Model` 的主键值，例如 `User` 的主键 `userId` 值是 “2356”，那么 `keyValue` 就是 “2356”。简化代码如下：

```
//获得 ModelManager 实例
```

```
ModelManager modelManager = (ModelManager)
```

```
WebAppUtil.getComponentInstance(ComponentKeys.MODEL_MANAGER, request);
```

```
modelManager.removeCache(keyValue);
```

上面代码是在容器外访问获得 `ModelManager`，使用上节容器内访问组件方式也可以获得 `ModelManager`；前者适合在表现层使用；后者适合在服务层使用。

最后，介绍一下清除全部缓存的方式，调用 `ModelManager` 的 `clearCache` 方法（Jdon 框架 1.3 以上版本），这样实际上将整个 Jdon 框架缓存全部清零。

前面两种清除缓存方式前提是首先获得 `ModelManager`，特别是服务层需要清除缓存时，需要以容器内访问组件方式获得 `ModelManager`，这只适合 POJOService 构成的服务层，如果我们使用 EJB 的 Session Bean 作为服务层实现，这时当前版本的 Jdon 框架容器不会在 EJB 容器中加载，因此，在 Session Bean 中无法访问到容器，无法获得 `ModelManager` 了，在这种情况下，可以通过设置 Model 的 `setModified` 属性为 `True`，表示该 Model 已经修改更新，这样当表现层获取该 Model 时，Jdon 框架缓存拦截器拦截时，发现该 Model 已经被修改，也就不会从缓存中获取。

Model 还有另外一个方法 `setCacheble`，当设置为 `false` 时，该 Model 将不会被保存到缓存中。如果你不希望某个 Model 被框架自动存入缓存，那么使用此功能，`setCacheble` 和 `setModified` 区别是，前者一旦设置为真，相当于缓存失效，以后再也不能用缓存；而后者则是当前 Model 表示被修改过，这样当有任何再次（限一次）试图从缓存中读取这个 Model 时，都会被会阻挡，从而可直接从数据库获得，然后再保存到缓存中，这样缓存中的 Model 数据就是新鲜的了。

明白 Jdon 框架 `setModified` 这个神奇作用，当你设置一个 Model 的 `setModified` 为真，那么你再读取缓存时，Jdon 框架内部将忽略你这个读取，返回一个 `null`；这样你根据返回是否为空，再从数据库直接获得，获得后，别忘记再保存到缓存中，已覆盖前次修改的旧数据，保证以后每次从缓存中读取的都是新鲜数据。

＝总结如下：手工缓存清除共有两种方式：

直接操作缓存，从缓存中清除；

如果无法操作到缓存体系，那么设置 Model 的 `setModified`。

失效 Model 缓存

如果你的系统已经对 Model 进行了缓存，那么可以失效 JF 的 Model 缓存：

用 winrar 将 JdonFramework.jar 中 META-INF 下 `aspect.xml` 解压出来，然后去除配置中 `cacheInterceptor`，再将 `aspect.xml` 拖入 winrar 的 JdonFramework.jar 中 META-INF 下覆盖原来的。

```
<interceptor name="cacheInterceptor"
    class="com.jdon.aop.interceptor.CacheInterceptor" pointcut="services" />
```

代码指定失效缓存：

`model.isCacheable isCacheable` 决定 model 是否保存到缓存。缺省是 `true`，如果设置为 `false`，该 Model 将不会保存到缓存中，但是如果之前已经在缓存中，设置为 `false` 后，不会自动立即从缓存中消除。

`Model isModified isModified` 决定 model 是否从缓存中读取。缺省是 `false`，如果设置为 `true`，该 Model 将不再从缓存中获得，但该 Model 有可能还存在缓存中，直到下次新的同样 Model 覆盖它。

这两种方法可以跳过缓存，但是不能清除缓存中原来的 Model，必须手工清除，或等该 Model 自动失效。

## 配置 encache 作为缓存

从 JdonFramework5.1 以后，在该项目目录下有一个 componenets/encache 项目，缺省 Jdon 框架的缓存是使用一个简单的 com.jdon.util.UtilCache，这种缓存是可以更换的，如果想更换为 encache，步骤简单，如下：

1. 更改 JdonFramework.jar 包中 META-INF 的 container.xml(方法可通过 winrar 打开 JdonFramework.jar，将 container.xml 解压更改后，再拖放回去覆盖原来的)：

```
<!-- 将原来缺省这行注释掉
      comment/delete this line in jdonframework.jar /META-INF/container.xml
    <component name="cache" class="com.jdon.util.LRUCache" >
      <constructor value="cache.xml"/>
    </component>
    -->

    <!--加入下面行
    active EnCache see prodject : componenets/encache: add these lines   in jdonframework.jar
/META-INF/container.xml -->
    <component name="cache" class="com.jdon.components.encache.EncacheProvider" />

    <component name="ehcacheConf" class="com.jdon.components.encache.EhcacheConf" >
      <constructor value="ehcache.xml"/>
      <constructor value="sampleCache1"/>
    </component>
```

2. 将项目目录 componenets/encache/dist 下的 jar 包文件 jdon-encache.jar ehcache-1.2.4.jar commons-logging.jar 和 JdonFramework.jar 放在一起。

3. 重新启动 JBoss 或 Tomcat

## 常用 API 说明

Jdon 框架提供了丰富的案例源码，一般简单情况下可以通过参考案例源码的用法明白一些用法，但是在深入使用过程中，需要熟悉一些 API 的用法，上节缓存的使用实际已经涉及了 Jdon 框架的 API 使用。

### JdbcTemp API

com.jdon.model.query.JdbcTemp 提供了一个简单的持久层解决方案，通过 JdbcTemp(JDBC 模板)可以完成增/删/改/查等 SQL 语句操作。

一般我们会选择 Hibernate/Ibatis 等作为持久层架构，但是这些框架需要另外学习和复杂的配置，对非常轻量的简单应用，直接使用 JdbcTemp 可节省大量时间，随着项目复杂和成熟，还是可以使用这些专门框架来替代 JdbTemp 的。

JdbcTemp 主要帮助开发者省却复杂的数据库 JDBC 操作语句（如打开/关闭数据库连接等），只要告诉它你要操作的 SQL 语句和参数数值，通过如下一句：



```
jdbcTemp. operate (queryParams, sql);
```

其中 queryParams 参数是一个 List 集合类型，里面装载的是参数集合，是有一定顺序的；sql 是一个字符串类型的 SQL 语句，如：

```
String sql = "INSERT INTO testuser (userId , name)  VALUES (?, ?)";
List queryParams = new ArrayList();
queryParams.add(userTest.getUserId());
queryParams.add(userTest.getName());
jdbcTemp.operate(queryParams,sql);
```

queryParams 中装载的是 sql 语句中两个问号参数，顺序也是和问号的顺序对应的，第一个问号是对应 testuser 的 userId 字段；第二个问号对应 testuser 的 name 字段。

数据库的 INSERT UPDATE DELETE 操作都是可以通过 operate 方法实现。

目前 JdbcTemp 的 operate 方法只支持有限几种类型:String Integer Float Long Double Bye Short，也就是说，当你需要更新数据库的参数类型符合这几种，可以使用 operate 方法，简化 JDBC 语句编写。

但是，如果这几种类型不符合你的参数类型，只能直接使用 JDBC 或使用专门的持久层框架等，Jdon 框架提供这个简单 operate 方法是为了简化一般系统的开发工作。

JdbcTemp 也提供多种查询操作语句：queryMultiObject 或 querySingleObject 方法，例如：

```
jdbcTemp. queryMultiObject (queryParams, sql);
```

```
jdbcTemp. querySingleObject (queryParams, sql);
```

querySingleObject 是查询返回一个对象，适合如下 SQL 语句：

```
select name from testuser where userId = ?
```

```
select count(1) from testuser where userId = ?
```

返回的是数据表一个字段，querySingleObject 则返回的相应类型，如字符串等，整数型则是整数型对象，如 int 则返回的是 Integer，bigInt 则返回的是 Long。

queryMultiObject 是最常用的查询方法，与 querySingleObject 返回单个字段相反，它返回的是多个字段，当然也可能是多行多个字段。类如 SQL 语句：

```
select userId, name from testuser where userId = ?
```

这个返回的两个字段，而且结果一般是一行，就是一条记录，但是下面 SQL 语句：

```
select userId, name from testuser where name = ?
```

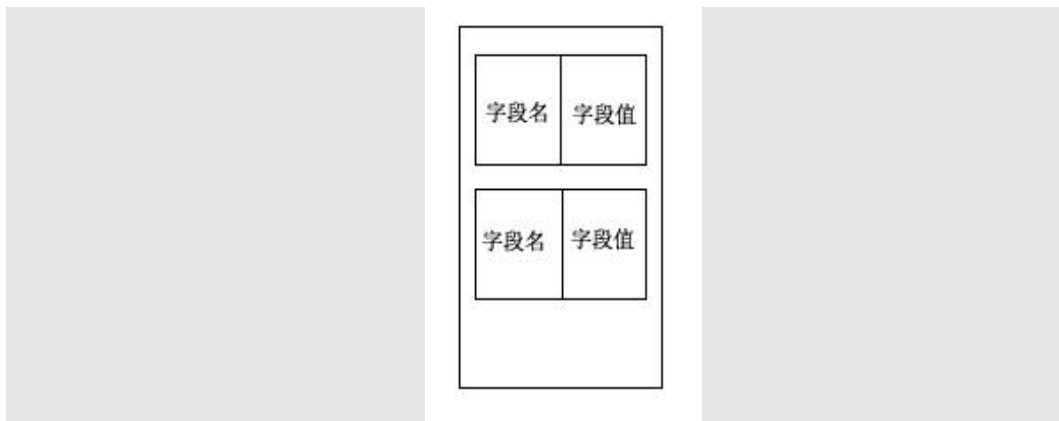
可能返回多行符合查询条件的记录，每行记录包含两个字段。

这两种形式都是可以采取 queryMultiObject 调用，queryMultiObject 返回的是一个 List 结果，在这个 List 结果中按照查询顺序装载着查询结果，这个 List 中每个元素是一个 Map 对象，Map 的 key 是数据表字段名称，Map 的 value 是数据表字段名对应的查询出来的值。

因此，从 queryMultiObject 结果中取值的写法如下：

```
List list = pageIteratorSolverOfUser.queryMultiObject(queryParams, GET_FIELD);
Iterator iter = list.iterator();
if (iter.hasNext()) { //如果有多行记录，这里是 while
    Map map = (Map) iter.next();
    ret = new UserTest();
    ret.setName((String) map.get("name"));
    ret.setUserId((String) map.get("userId"));
}
```

List 结果中装载内容结构如下：



### JdbcTemp 使用前提

只要将 DataSource 传给 JdbcTemp 即可，DataSource 可以来自容器的 JNDI；也可以来自自己编码的连接（自己编码的连接代码需要向外界提供获得 DataSource 方法）；也可来自 Hibernate 等持久层框架，在这些框架配置中配置数据库连接，然后通过这些框架的 API 获得 DataSource 赋值给 JdbcTemp 即可。

目前 JdbcTemp 的 operate 方法只支持有限几种类型：String Integer Float Long Double Byte Short，也就是说，当你需要更新数据库的参数类型符合这几种，可以使用 operate 方法，简化 JDBC 语句编写。

这样做的理由基于“将所有数据表字段尽可能都设计成字符串类型”：  
<http://www.jdon.com/jive/thread.jsp?forum=91&thread=23875>

### DaoCRUDTemplate API

本功能只有 Jdon 框架 5.1 以后才有。使用本功能可以无需你写 CRUD 的持久层代码了，大大减轻开发工作量。

在 Struts+Jdon +Hibernate3.2 的案例 samples\_hibernate.zip 中，无论任何对象的持久化，都可以使用如下一个 JDBC DAO 接口实现：

```
public void insert(Object o) throws Exception;

public void update(Object o) throws Exception;

public void delete(Object o) throws Exception;

public Object loadModelById(Class classz, Serializable Id) throws Exception;
```

这些 CRUD 方法已经在 Jdon 框架的 com.jdon.persistence. DaoCRUDTemplate 模板中已经实现，你只需直接调用就可以了。如你自己的 JdbcDaoImp 可以如下实现：

```
public class JdbcDaoImp extends DaoCRUDTemplate implements JdbcDao{
    ....
}
```

当然，如果要使用 DaoCRUDTemplate，还需要激活 CloseSessionInView，在 web.xml 配置 CloseSessionInViewFilter，如下：

```

<filter>
    <filter-name>hibernateFilter</filter-name>
    <filter-class>com.jdon.persistence.hibernate.CloseSessionInViewFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>hibernateFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

```

这样可以实现一个请求结束后再进行 Session 关闭，这样，Hibernate3.2 缺省的懒加载功能就可以正常使用。更多可见：[5.1 新功能章节](#)。

多态、有态和单态

目前在 Jdon 框架 1.3 以上版本中，提供三种接口让你的应用服务 Service 类继承：

`com.jdon.controller.pool.Poolable` //对于主要很大的类推荐使用，使用对象池提高你的服务实例运行性能

`com.jdon.controller.service.Stateful` //有状态，当你需要处理有状态数据使用，如购物车

`com.jdon.container.visitor.data.SessionContextAcceptable` //需要获得容器的登陆信息，见下节。

上述三种接口选择根据具体业务需求决，`SessionContextAcceptable` 可以和前面两种混合使用，`Poolable` 和 `Stateful` 不能并列使用，只能二选其一。

5.6 版本以后，分别提供以上三种 Annotation 元注解，这样不用你的代码实现这些接口，从而减少所谓侵略性。

假设你的一个应用服务类的接口名为 `MyServiceIF`，类实现为 `MyServiceImp`，那么当客户端进行如下服务调用：

```
MyServiceIF ms = (MyServiceIF) WebAppUtil.getService("myService ", request);
```

这时，`ms` 这个实例产生和 `MyServiceImp` 所继承的上述三种接口有关系，如果 `MyServiceImp` 实现了 `Poolable`，那么 `ms` 就是 `MyServiceImp` 多态实例池中的任何一个；如果 `MyServiceImp` 实现了 `Stateful` 接口，那么，每个客户端每次获得的 `ms` 都是同一个 `MyServiceImp` 实例；如果 `MyServiceImp` 不继承任何接口，那么每次调用上述服务获得的 `ms` 实例都是一个新的实例，相当于 `new` 一个实例。

如果希望每次获得 `ms` 实例是单态实例，也就是是一个实例，直接通过上述 `WebAppUtil.getService` 语句是不行的，但是有一个变通办法，将 `MyServiceImp` 嵌入在其他服务类中，例如嵌入一个叫 `OurService` 类中，例如：

```

public class OurService implements OurServiceIF {

    private MyServiceIF myservice;

    public OurService(MyServiceIF myservice) {
        this.myservice = myservice;
    }

    public MyServiceIF getMyservice() {
        return myservice;
    }
}

```

```
}  
}
```

这样，首先获得 `OurService` 实例：

```
OurServiceIF os = (OurServiceIF) WebAppUtil.getService("ourService ", request);  
MyServiceIF ms = os. getMyService();
```

这样，每次获得 `ms` 就是同一个实例，或者称为单态，原因是：`MyServiceImp` 是通过 `Ioc` 容器注射进入 `OurService` 中的，而 `Ioc` 容器每次注射的都是单态实例。

## 用户注册登陆

用户注册登陆系统是每个系统必须模块，主要包括两个功能块：用户资料注册和修改部分；用户登陆和访问权限 `ACL` 部分。

前一个部分主要是用户资料的增删改查 `CRUD`，可以使用 `Jdon` 框架轻易完成；如 `JPostore` 中的 `SignAction` 类。

第二个部分有两种实现方式：基于容器和手工定制（`Container-based` vs. `custom security`），这两种实现方式都可由用户自己选择，这里主要说明一下，如何将用户登陆后的信息如用户 `ID` 共享给业务模型，例如：论坛发言者登陆后，发表一个帖子，这个帖子应该有一个用户 `ID`，如何将发言者的用户 `ID` 赋值给帖子里？依据不同的实现方式有不同的捷径：

手工定制就是开发者自己编制代码完成，为需要保护的资源设立 `filter`，或者在每个 `Jsp` 页面 `include` 一个安全权限检查 `Jsp` 模块，在这个方式下，用户登陆信息一般是保存在 `HttpSession` 中，那么在表现层继承 `ModelHandler` 实现一个子类，在这个 `Handler` 中能够访问 `HttpSession`，并取出事先保存在 `HttpSession` 中的用户信息如 `ID`。

当然，基于容器的实现方式也可以采取上述办法，例如 `JdonNews` 的 `NewsHandler` 类中的 `serviceAction` 方法代码，如下：

```
User user = (User) ContainerUtil.getUserModelAfterLogin(request);  
if (user != null) {  
    News news = (News) em.getModel();  
    news.setUser(user);  
}  
else  
    Debug.logVerbose(" not found user in any scope ", module);
```

这样，将用户登陆信息 `user` 直接赋值到 `news` 实例中。

这种方式是通用方式，缺点是需要专门实现一个 `ModelHandler` 类，需要编写代码，还有一个不需要编写代码的方式：

获得基于容器登陆用户 `Principle`

下面介绍在服务层能够直接获得基于容器的登陆用户 `UserPrinciple` 名称的方式，这种方式只适合 `POJO Service` 架构，而且必须是基于容器实现方式，相关基础知识点见：

<http://www.jdon.com/idea/jaas/06001.htm>

使用本方式目的在于简化表现层代码，在 Jdon 框架中，表现层基本都是通过 jdonframework.xml 配置实现，一般情况不必编码，如果不使用本方式，那么就必须自己实现一个 ModelHandler 子类，现在则不必，可以在业务服务层服务 Service 对象中直接获得 request.getPrinciple()方法结果。

直接简单的使用方式：

1.让 Service 对象继承 com.jdon.container.visitor.data.SessionContextAcceptable，该接口目前有两个方法：

```
void setSessionContext(SessionContext sessionContext);
SessionContext getSessionContext ();
```

这样这个 Service 类必须完成接口上述两个方法，实际是 Jdon 容器在运行时，通过 com.jdon.aop.interceptor. SessionContextInterceptor 这个拦截器在运行时注射到被调用的这个 Service 对象中。

目前 com.jdon.container.visitor.data.SessionContext 中已经包含 String 类型的 principleName，这个值来源于 request.getUserPrincipal()获得的 Principal 的 Name 值，如下：

```
Principal principal = request.getUserPrincipal();
String principleName= principal.getName();
```

上述这段代码是在 Jdon 框架中执行，这样通过这种方式，在服务层服务对象中我们可以容易获得保存在 SessionContext 中用户登陆信息 Principle Name，使用这个方式虽然方便，但是必须有下列注意点：

1. 继承 com.jdon.container.visitor.data.SessionContextAcceptable 的服务层服务 Service 对象必须被客户端直接调用，调用方式是代码或配置方式都可以，客户端代码方式如下：

```
FrameworkTestServiceIF frameworkTestService = (FrameworkTestServiceIF) WebAppUtil.getService(
    "frameworkTestService", request);
frameworkTestService.getName()
```

或在 jdonframework.xml 中配置 model 部分指向 frameworkTestService 调用也可以，这是一种配置方式调用。

只有这个服务 Service 对象直接面对客户端调用，拦截器的注射功能才能将 SessionData 注射进入它的“体内”。

2.访问服务层服务 Service 对象的客户端必须在容器安全体系保护下，一般这样客户端都是 Jsp 或 Servlet/Action，都会有 Url，例如 <http://localhost:8080/MyWeb/Admin/XXXX.do>，在 web.xml 以及配置/Admin/\*都是必须被特地角色才能访问。

因为只有这样，Web 容器的 Request 对象中 getUserPrinciple 才会有值，这是 J2EE 规范规定的。载有角色值的 Request 访问/Admin/XXXX.do，XXXXX.do 是 Struts 的一个 Action，再通过 Action 访问特定的服务 Service 对象。

在上述两种情况同时满足情况下，服务层服务 Service 对象中 sessionContext 中的 principleName 才会有该用户的登陆信息，注意这个登陆信息令牌可以是用户 ID 或用户名，取决于你容器配置如 JBoss 的 login-config.xml，只有密码验证通过过 Web 容器才会有值。

一旦你在 Service 中获得 principleName，你就可以再通过查询数据库获得该用户的完整资料，从而将用户信息共享给所有业务服务层。

具体代码见 Jivejdon 3.0 的 com.jdon.jivejdon.service.imp. ForumMessageServiceImp。

## SessionContext 用处

上述容器登陆信息 Principle Name 通过保存在 SessionContext 中使得服务层服务对象能够获得用户的登陆信息。

SessionContext 是保存在 Web 容器的 HttpSession 中一个载体，当在服务层服务对象中需要缓存一些数据，这些数据的 Scope 是 Session 性质，这样可以通过 SessionContext 实现数据共享。Jdon 框架还提供 Stateful 提供另外一种与 Session 有关状态载体选择。

如果希望缓存或共享的数据的 scope 是 Request，可以使用 JDK 新语言特性 ThreadLocal，服务层的服务对象就能够和 Jdon 容器以外的 Request 请求相关发生联系，例如我们可以做一个 ServletFilter，将某些数据保存在 ThreadLocal 中，然后在服务层服务对象中从 ThreadLocal 中获取，这属于一般应用技巧，这里不详细描述，只是做一个应用提示。

如果缓存或共享的数据 scope 是 Application，就将其注册到 Jdon 容器中，因为 Jdon 容器的 scope 是 Application 的，这样整个服务层也可以访问。

目前 Jdon 框架通过 SessionContextSetup 将一些用户相关信息保存到 SessionContext，当应用系统通过 Jdon 框架调用任何服务时，将激活 SessionContextSetup，SessionContextSetup 的缺省实现是 com.jdon.security.web.HttpRequestUserSetup，

目前在 HttpRequestUserSetup 中，实现两个功能：将用户通过容器登陆的 Principle 和 remote Address 地址保存在 SessionContext，这样，在业务层中的所有服务将可以通过 SessionContext 获得这两个数值。

HttpRequestUserSetup 是通过在 container.xml 中配置激活的：

```
<component name="sessionContextSetup" class="com.jdon.security.web.HttpRequestUserSetup" />
```

如果你还有更多数值需要供业务层服务使用，可以提到这行配置，见[组件替换](#)。

例如：缺省 HttpRequestUserSetup 中是从 request 中获得用户登录的 Principle，如果你想自己实现用户登录，使用 ServletFilter 来实现登录验证，验证通过后，将验证后的用户信息保存在 Request 或 ThreadLocal 中，这样，你就需要实现自己的 SessionContextSetup，用来从 Request 或 ThreadLocal 中获取用户信息给 Jdon 框架。

## cookie 自动登录

本节是针对基于 Web 容器登录体系，login.jsp 登录页面如下：

|   |                          |  |
|---|--------------------------|--|
| 用户  | <input type="text"/>     | 自动登陆 <input checked="" type="checkbox"/> |
| 密码  | <input type="password"/> | <input type="button" value="登陆"/>        |
| <a href="#">新用户注册</a> <a href="#">忘记密码?</a> |                          |  |

通常基于容器的登录，在 login.jsp 中提交的 action 是 j\_security\_check，为了激活“自动登录”功能，我们需要使用自己的 servlet，然后再转发到 j\_security\_check，如下：

```
<form method="POST" action="%<%=request.getContextPath()%>/login"
....
</form>
```

这里/login 是一个 Servlet，在 web.xml 中配制如下：

```
<servlet>
<servlet-name>login</servlet-name>
```

```

<servlet-class>com.jdon.security.web.LoginServlet</servlet-class>
<init-param>
  <param-name>login</param-name>
  <param-value>/account/login.jsp</param-value>
</init-param>
<init-param>
  <param-name>logout</param-name>
  <param-value>/account/logout.jsp</param-value>
</init-param>
<load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>login</servlet-name>
  <url-pattern>/login</url-pattern>
</servlet-mapping>

```

com.jdon.security.web.LoginServlet 是 JdonFramework 中的一个类，主要功能：

如果是从 login.jsp 提交，LoginServlet 判断自动登录是否选中，如果是，将用户和密码加密后保存在客户端 cookie 中，然后再转发到 j\_security\_check。

如果是直接/login?logout 调用，表示退出，我们应用的退出按钮连接需要指向这个连接，退出主要是 session 失效和 cookie 失效。

如果是直接无参数/login 调用，首先判断当前 cookie 是否保存用户和密码，如果是，使用这对用户密码发往 j\_security\_check 自动登录；如果不是，推出 login.jsp 页面。

LoginServlet 需要配置两个参数：login 和 logout，表示你的应用的 login.jsp 位置和 logout.jsp 位置，LoginServlet 将根据条件推出这些页面。

以上是配置之一，还有下面的 web.xml 配置。

在 web.xml 的 form-login-page 中，不要直接将登录页面/account/login.jsp 写入，这样自动登录功能就不能激活，或者说，通过前面步骤我们配置激活 cookie，cookie 中保存了我们的用户和密码，但是具体使用时还必须到 cookie 中查询，因此需要配置 form-login-page 为/login，也就是 LoginServlet。

```

<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login</form-login-page>
    <form-error-page>/account/login_error.jsp</form-error-page>
  </form-login-config>
</login-config>

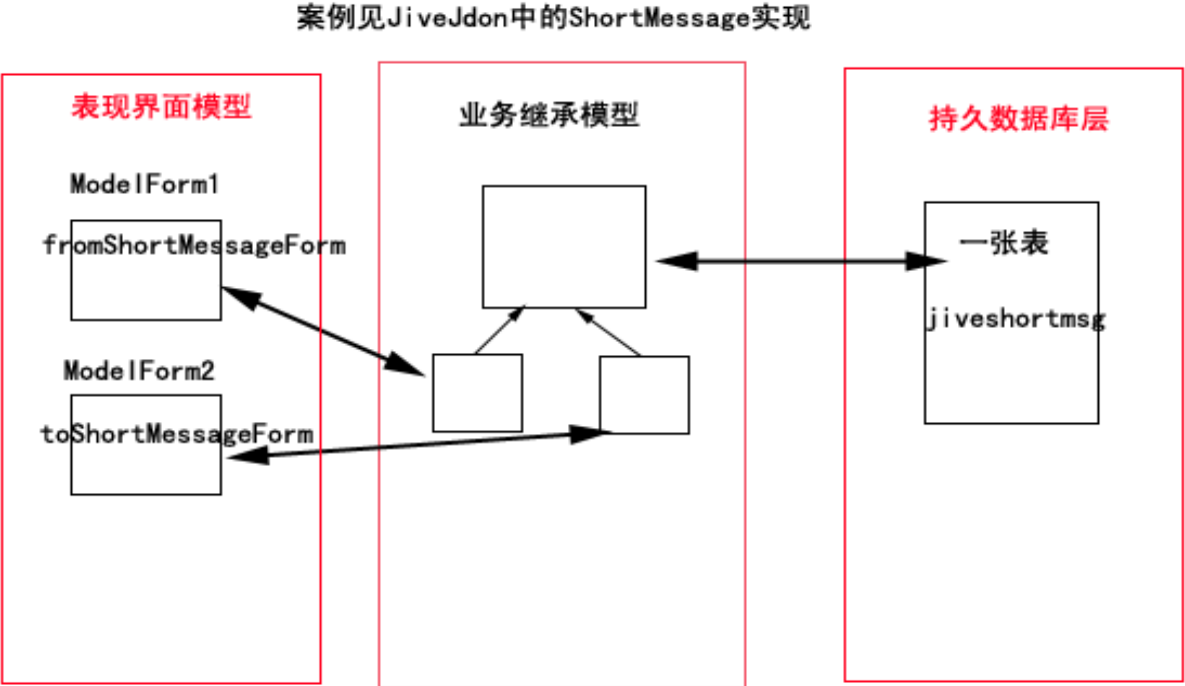
```

## 继承模型最佳实践

Jdon 框架除了可以实现根据 Evans DDD 的关联模型以外，也可以方便实现继承关系的模型，下面是来自 JiveJdon 的继承模型最佳实践，具体案例见 JiveJdon 的 ShortMessage 代码。

消息模型是一个父子继承关系，这样父子继承关系的模型如何使用 Jdon 框架来简单明了的实现，也就是不破坏领域层的继承关系，将这种继承关系延伸到非对象化的表现层

和持久层，下面根据消息模型的实践总结为最佳实践如下图：

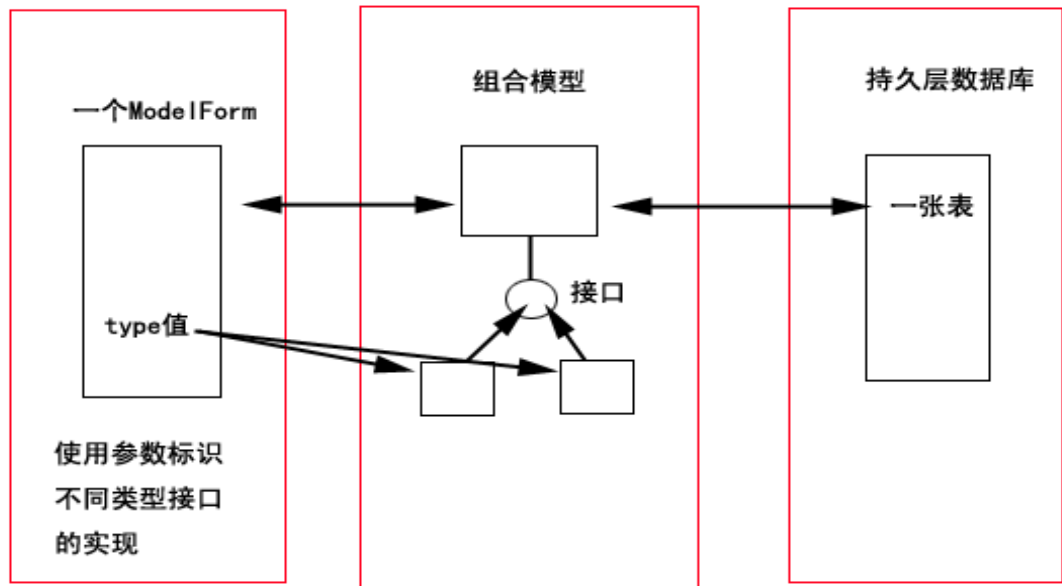


## 组合模型最佳实践

关注订阅模型是 JiveJdon 的一个功能，该模型的特点是 Subscription 根模型中有一个接口关联，而接口有不同实现，将来可能要扩展，但是在具体运行中，该接口肯定是其具体实现的一个子类，如何协调表现层模型来实现这一点，不破坏组合模型的特点和优点，JiveJdon 的关注 Subscription 模型提供一条最佳实践方式。具体见其代码。



案例见JiveJdon中的Subscription实现



struts-config.xml 的配置一个 ActionForm:

```
<form-bean name="subscriptionForm"
    type="com.jdon.jivejdon.presentation.form.SubscriptionForm"
/>
```

在 model.xml 模型的 CRUD 配置中,也配置 ActionForm 和 Model 一对一.

```
<model key="subscriptionId"

class="com.jdon.jivejdon.model.subscription.Subscription">
    <actionForm name="subscriptionForm" />
    <handler>
        <service ref="subscriptionService">
            <getMethod name="getSubscription" />
            <createMethod name="createSubscription" />
            <deleteMethod name="deleteSubscription" />
        </service>
    </handler>
</model>
```

微妙之处在 `com.jdon.jivejdon.presentation.form.SubscriptionForm` 代码中.:存在 `subscribeType` 这个参数, 而模型 `Subscription` 中没有 `subscribeType` 这个参数, 只有数据库中有 `subscribeType` 这个字段,转换关系:

表现层: `subscribeType` <----> 模型中 `Subscribe` 类型 <----> 数据表 `subscribeType`  
通过这种转换,可保证模型层使用 OO 面向对象方式设计,将数据完美表现为对象.

`SubscriptionForm` 中 `subscribeType` 和 `Subscribe` 转换主要两个方法如下:

```
/**
 * transfer Form to Model join fields into Subscribed;
 * Form拷贝到Model时,将自动执行Form的getSubscribe,将结果通过
```

调用Model的setSubscribe方法,赋值到Model中.

```
* @return
*/

public Subscribed getSubscribe() {
    return SubscribedFactory.create(subscribeType, subscribeId);
}

/**
 * transfer Model to Form separate Subscribed into fields;
 * 模型拷贝到Form时,将自动执行模型的getSubscribe方法,将其结果通过Form的
setSubscribe方法赋值其中.
 * @return
 */

public void setSubscribe(Subscribed subscribe) {
    this.subscribeType =
        SubscribedFactory.getSubscribeType(subscribe);
    this.subscriptionId = subscribe.getSubscribeId();
}
```

## Domain Event 模式

在 Evans DDD 实现过程中,经常会碰到实体和服务 Service 以及 Repository 交互过程,这个交互过程的实现是一个难点,也是容易造成失血贫血模型的主要途径。

Domain Event 提出解决方案, Jdon 框架提供的[异步观察者模式](#)为 Domain Event 实现提供更优雅的解决方案。

详细文章见: <http://www.jdon.com/jivejdon/thread/37289>

使用 [6.2 版本的领域消息](#)可以彻底实现领域模型和技术架构的耦合,领域模型任何事件只要通过消息就可以和技术架构比如持久化 领域服务进行交互,而且是异步的,没有任何事务单一线程。

## Startable 接口

框架提供 com.jdon.container.pico.Startable 接口, Startable 接口有两个方法:

start(); 当应用开始运行时,激活这个方法;

stop(); 当应用 undeploy 或容器正常 shutdown 时,激活该方法。

可以自己做一个类,实现 Startable 接口, start 方法是在类的构造方法以后才执行,可以用于启动一些重量任务,如启动线程等,而 stop 方法则是在应用从服务器中 undeploy

时自动执行，可以在其中执行一些将内存中数据写入数据库持久化的任务。

## 版本新增或改变功能

### 1.X 版本

#### 1.4 版本

增加 ModelIF 接口，其意义等同于以前 Model 类，是继承 Model 类或实现 ModelIF 接口，可由框架应用者自行决定。

服务调用命令模式，见[这里](#)。

批量分页查询，提供新的批量读取数据库算法，增加批量查询时缓存的重用效率。

#### 1.5 版本

Fixed 一些 1.4 版本的 BUG。

update 02:解决主键类型是非 String 时，可能带来的问题，在 1.4 及其以前版本，如果主键类型是非 String，Service 接口中查询模型方法如 getMessage(String messageId) 方法，该方法参数 messageId 必须为 String，这会带来初学者一些歧义，现在 1.5 以后这个方法参数 messageId 主键类型可以和模型主键类型统一了，如模型主键类型是 Long 方法，那么 Service 接口可以为 getMessage(Long messageId)。

在 JDK5.0 下编译通过，将 JF 应用案例全部在 JDK5.0 下调试通过。

暂时没有在 1.5 版本中加入 JDK5.0 特殊语法，换言之，1.5 版本还可以在 JDK1.4 平台编译通过。

1.5 版本推荐使用平台： JDK5.0 + JBoss 3.2.8

### 5.x 版本

#### 5.0 版本

升级到 JavaEE5.0 平台，本框架只能在 JDK5.0 以上平台编译和运行，引入了 5.0 高质量的并行并发概念，提高 Jdon 框架在高访问下的线程安全性和并发性。

增加对 EJB3 服务的支持，如果你的普通 JavaBean 架构需要升级到 EJB2/EJB3，无需更改客户端或表现层代码，直接修改 JdonFramework 配置即可。

#### 5.1 版本

JdonFramework 5.1 主要变化是增加了 Hibernate3 支持，并且将原来一个 jdonFramework.jar 包分为三个包：jdonFramework.jar jdon-struts1x.jar 和 jdon-hibernate3x.jar

如果使用 struts+jdon+Hibernate 架构，这三个包需要完全使用。如果只使用 Jdon 框架作为业务层框架，只需要 jdonFramework.jar，可以使用 Jdon 框架的 IOC 管理功能，不过 Jdon 框架提供的 CRUD 流程简化将无法实现。

JdonFramework 5.1 重点是增加 Hibernate3 整合，特别是 Hibernate3 的懒加载支持，每个使用 Jdon+Hibernate 项目都需要在 web.xml 配置 CloseSessionInViewFilter，如下：

```
<filter>
    <filter-name>hibernateFilter</filter-name>
    <filter-class>com.jdon.persistence.hibernate.CloseSessionInViewFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>hibernateFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

这样可以实现一个请求结束后再进行 Session 关闭，这样，Hibernate 缺省的 Lazy=true 功能就可以正常使用。

懒加载功能可以大幅度提高 Hibernate 关联性能(前提是基于 DDD 分析设计)，是 Hibernate 必须使用的功能，但是单纯使用 Hibernate 却无法激活懒加载，致使很多人关闭懒加载 lazy="false"。

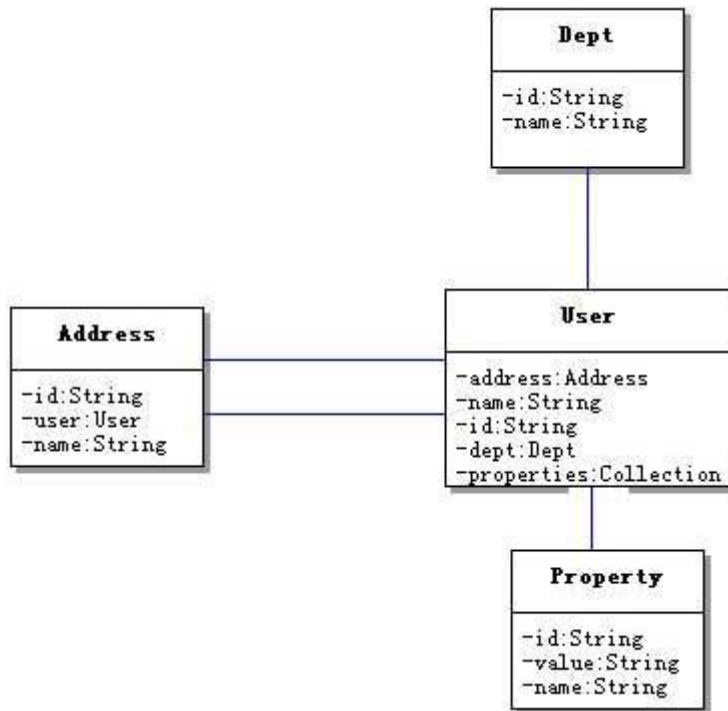
Spring 也提供了 Hibernate 的 OpenSessionInView 功能，但是它是将 Hibernate 的 Session 再表现层打开和关闭，而 JdonFramework 5.1 只是检查是否打开的 Session，如果有则关闭，这样缩短 Session 无谓开启时间，降低出错率，同时简化事务(Spring+Hibernate 架构 Spring 事务缺省是 read-only 只读，只有配置显式 create 方法事务为非 read-only 后，才能使用 Hibernate 保存创建新资料，非常不方便)；

Struts+Jdon+Hibernate(简称 SJH)和 Struts+Spring+Hibernate(简称 SSH)最大的特点是：

- 1.SJH 对 Hibernate 无侵略性，Hibernate 全局配置还是在自己的 hibernate.cfg.xml 中配置，而不似 SSH 需要在 Spring 中配置。这样从设计上减少彼此依赖性，实现真正分层松耦合。jdon-hibernate3x.jar 可以单独使用。
- 2.SJH 以更少的代码快速实现增删改查和批量分页查询。
- 3.Jdon 框架内置缓存+Hibernate 二级缓存+懒加载最大化提高缓存使用效率，性能优异，有效实现数据库零访问和必要访问。

JdonFramework 5.1 推出同时，也推出 struts+jdon+Hibernate3 的整合应用案例  
[http://www.jdon.com/jdonframework/download/samples\\_hibernate.zip](http://www.jdon.com/jdonframework/download/samples_hibernate.zip)

该案例领域模型图：



## 5.5 版本

将框架与 `HttpRequest` 和 `HttpSession` 分离，框架不一定需要用在 WEB 中，也可应用在 Application，直接运行源码包中的 `runTest.bat` 测试。

Jdon 框架可以只作为一个 IOC/DI 框架，使用在 JavaEE 或普通 Java 或 J2ME/JME 中，拓展用途。

5.5 版本经过严格的并发性能测试，解决了以往框架中可能存在的内存泄漏漏洞，使用 `ThreadLocal` 以及 `java.concurrent Callable Future` 等 JDK5.0 以上新的并行功能，增强了并行计算能力。

5.5 版本 10 分钟内发出近万次并发请求，没有任何报错和问题，测试客户端见 JiveJdon3.5 源码包中 `jmeterTest.jmx`，可以使用 Jmeter 打开，服务器端运行 Jprofiler 和 JBoss 服务器，注意：进行并发测试时，需要将 `concurrent_myaspect` 和 `concurrent_web.xml` 提到原来的 `myaspect.xml` 和 `web.xml`，失效 JiveJdon 的防 SPAM 功能。

5.5 版本的源码经过并发性能重构，增强稳健型和快速性，是目前 Jdon 框架中最稳定的版本。

限制 `sessionContext` 中最多只能放置 10 个对象，超过就自动清楚，这样，不要将重要状态放入 `sessionContext` 中，也不要放置超过 10 个对象，`sessionContext` 最大个数 10 设置是在 `container.xml` 中。

使用 ehcache 作为 Jdon 框架缺省的缓存，以增强兼容性和伸缩性，分布式 Cache Terracotta <http://www.terracotta.org/> 可以将 encache 自动进行分布到多台服务器，这样，Jdon 框架也能支持分布式缓存和计算（无需借助 EJB <http://www.jdon.com/article/34888.html>）。

Jdon 框架部署在集群环境下，由于集群环境，`HttpSession` 将被在多台服务器间同步，而 Jdon 框架利用 `HttpSession` 进行动态代理优化缓存，这时可能会引起

**NotSerializableException** 问题，采取办法可以失效优化缓存。

确认用 **winrar** 打开 **JdonFramework.jar**，将 **META-INF** 的 **container.xml** 拖到某个文件夹下。将其中下面这一行中 **true** 改为 **false**，保存后再将 **container.xml** 拖回 **winrar** 中即可。

```
<component name="httpSessionVisitorFactorySetup"
class="com.jdon.container.visitor.http.HttpSessionVisitorFactorySetup">
<constructor value="40" />
<!-- it is count of the cached instances that saved in a sessionContext instance, -->
<constructor value="true" /> <!-- disable all session cached except SessionContext -->
</component>
```

正常单机情况下，也可以将这一设置改为 **false**，可以节省内存，经过测试，稳定运行后性能差异不大。

## 5.6 版本

5.6 版本主要是增加 **Annotation** 元注释来替代原来一些接口，减少框架的侵略性。

**Jdon** 框架默认提供了三种拦截器，当然你可以自己增加定制，这三种拦截器分别是对象池 **Session** 周期和有态，以前为让你的 **service** 实现对象池等这三个功能，需要特别实现三个接口 **Poolable** **SessionContextAcceptable** **Stateful**，而现在有一个可替代方法，就是使用元注解 **Annotation**，例如：

```
@Poolable
```

```
public class ForumMessageShell implements ForumMessageService
```

表示 **ForumMessageShell** 将以对象池形式存在，这样对于一个主要代码很多的类，可以起到提高性能，防止资源无限消耗。对象池优点见：<http://www.jdon.com/jivejdon/thread/35191.html>，注意不要将小类做成 **Pool**，因为对象池本省也损耗一定性能。

另外一个元注释是 **Model**，原来所有 **Domain Model** 需要实现接口 **ModelIF** 或继承 **Model** 类，现在可以使用 **@Model** 来替代了，如：

```
@Model
```

```
public class Account{}
```

目前 **Jdon** 框架是 **Annotation** 和 **XML** 配置组合使用，**Jdon** 框架并没有使用 **Annotation** 全面替代 **XML** 配置文件，因为我们对于元注解 **Annotation** 的观点是：适当但是不可乱用，是一把双刃剑。当所有组件都可以自由拆开 **IOC/DI**，运行时也可自由指定顺序 **AOP**，那么我们就能够更快地跟随变化，甚至无需重新编译整个项目，只要带一个组件到现场，**XML** 配置改写一下，就能上线运行，注意，我这里提到了 **XML** 配置，如果你完全使用 **Annotation** 就达不到这个目的。

<http://www.jdon.com/jivejdon/forum/messageList.shtml?thread=35373&message=23119938#23119938>

## 5.8 版本

增加基于 **Http** 的远程 **remote** 远程调用，可以方便开发出基于 **RIA** 富客户端的多层架构 **C/S** 系统。见案例 **remote\_javafx**

案例 remote\_javafx 是使用 RIA 技术 JavaFX + Hessian + Jdon 框架的一个 Demo 案例，开发简要步骤：

在 web.xml 配置 Hessian Servlet Proxy 如下：

```
<servlet>
<servlet-name>Hessian2Jdon</servlet-name>
<servlet-class>com.jdon.bussinessproxy.remote.HessianToJdonServlet
</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>Hessian2Jdon</servlet-name>
<url-pattern>/remote/*</url-pattern>
</servlet-mapping>
```

这样，当远程客户端调用 <http://localhost:8080/remote/helloService>，那么这个 Servlet 将在 jdonframeowrk.xml 查找服务名称为 helloService 的服务：

```
<app>
  <services>
    <pojoService name="helloService" class="sample.HelloServiceImpl"/>
  </services>
</app>
```

远程客户端代码如下：

```
HessianProxyFactory factory = new HessianProxyFactory();
HelloService _service = (HelloService)
    factory.create(HelloService.class, _url);
_service.hello(s);
```

客户端调用 HelloService 的 hello 方法时，将激活 HelloServiceImpl.hello 方法。Demo 在线演示网址：<http://www.jdon.com:8080/jdonremote/>

## 6.x 版本

### 6.0 版本

（一）增加 Annotation 注射功能， @Service 或 @Component 替代了原来 XML 配置：

```
<pojoService name="给自己类取的名称" class="完整类的名称"/>
或者在具体类代码上加入：@Service("给自己类取的名称")

<component name="给自己类取的名称" class="完整类的名称"/>
或者在具体类代码上加入 @Component("给自己类取的名称")
```

Service 和 Component 都表示组件构建概念，可以是一个类，也可以是一个类为主的多个类，如果这个组件供客户端调用，那么就称为服务。两者在 Jdon 框架中没有区别，只是使用时称呼不同而已。

@Component(“给自己类取的名称”)也可以简化为@Component，这样，Component 的名称就是该类的完整类名称(getClass.getName());

@Service 则不可以这样，因为是供 Jdon 框架容器以外的外部客户端调用的，一定要取一个名称。

因为需要被客户端调用，那么就要指定服务的名称，所以，@Service 与@Component 的区别就是，@Service 必须规定一个名称 @Service(“给自己类取的名称”)；而@Component 则没有 name 属性，缺省是类名称 getClass().getName()名称，如果客户端有时需要临时直接调用@Component 标注的组件，也可以按类名称调用。

开发 Jdon 框架变得异常简单，只需要两步：

第一步：将存在依赖关联关系的两个类用@Service 或@Component 标注：

@Poolable

@Service("helloService")

```
public class HelloServiceImpl implements HelloService{
    UserRepository userRepository;
    public HelloServiceImpl(UserRepository userRepository){
        this.userRepository = userRepository;
    }
    ..
}
```

@Component

```
public class UserRepositoryInMEM implements UserRepository {
```

客户端调用代码：

```
HelloService helloService = (HelloService) WebAppUtil.getService("helloService", req);
String result = helloService.hello(myname);
```

无需 jdonframework.xml 配置文件和相关 XML 配置。全部演示代码见 JdonFramework 源码包目录下 examples/testWeb

(二).增加@Singleton Annotation，缺省客户端调用 getService 时，每次请求获得一个新的 Service 实例，也可以使用@Poolable 对象池来对代码比较大的 Service 类进行性能提高，现在又提供了另外一种方法，使用单例@Singleton。

单例和对象池区别主要是：对象池可以进行实例最多个数控制，这样，能够保护服务器不会被尖峰冲击造成资源耗尽。

在 Service 服务代码比较简单小的情况下，Service 实例三个创建方式：单例、对象池以及每次请求生成一个新实例，在性能上几乎没有太大差别，已经使用框架源码包中 examples 目录下的 testWeb 经过测试了。

(三).没有将有关 Action 的配置转为 Annotation，原来的配置如下继续保留：

```
<model key="username"
      class="com.jdon.jivejdon.model.Account">
  <actionForm name="accountForm"/>
```



```

        <handler>
            <service ref="accountService">
                <initMethod    name="initAccount"/>
                <getMethod     name="getAccountByName"/>
                <createMethod name="createAccount"/>
                <updateMethod name="updateAccount"/>
                <deleteMethod name="deleteAccount"/>
            </service>
        </handler>
    </model>

```

因为这个配置是有关 MVC 表现层，而表现层不是 Jdon 框架的核心，所以，Jdon 框架核心是业务层，对 Domain Model 和 Service 服务组件进行管理，所以，不能在业务层耦合具体表现层。上述配置可以认为是表现层和业务层的桥连接配置。

## 6.1 版本

增加异步事件处理功能，可以在当前处理过程同时，抛出另外一个过程处理其他事务，不影响当前主要处理过程，例如，发送邮件是一个很费时的过程，如果一个业务过程需要发送 Email，那么将发送 Email 这件事情通过异步，放到另外一个线程中处理，从而能实现处理效率性能的提供。

6.1 版本根据 Jdk 6.0 的并发 Concurrent 模型实现了异步事件处理功能 `com.jdon.async.EventProcessor`。

该类有两个构造参数，第一个是你分配几个线程同时来处理的异步事件，一般是设置一个，Queue 队列性质，一对一，如果你希望多个线程并发同时处理，发挥并发性能，那么可以选择更大数字，不过要注意，你的业务是否能够被多个线程同时执行，比如 Email 发送如果多个线程发送，那么就多发几次，一次就够了，因此设置为 1。

第二个参数，是允许同时几个线程在处理事务，因为你是将一个个事件或消息逐个放到 EventProcessor 队列中，如果每个事件或消息处理起来很费时，在之前事件消息还没处理完，你是否同意系统再启动其他线程处理后面的事件消息，这个参数需要根据你的硬件系统处理能力而定，如果处理能力不高，那么设置小一些。

6.1 版本还基于 EventProcessor 提供了观察者模式，用来实现对模型中数据变动的监测。

通常模型中数据变动，我们是通过直接关联或依赖关系，来通知另外一个模型同时变动，如下：

```

void changeValue(OtherModel om){
    this.value = "newValue";
    om.setValue("newValue too");//通过这个方法通知另外一个模型修改
}

```

这种方式适合与核心模型关系紧密的模型使用，比如同属于一个聚合模型边界内的模型对象们，但是现实中，还有一些关系并不如聚合关联那么紧密，是和核心模型有关联，但是是一种非常松散的关联，如何实现他们之间变动事件的传递？

使用观察者模式，传统的 JDK 观察者 API 实现是同步的，如下：

```

void changeValue(OtherModel om){
    this.value = "newValue";
}

```

```

        setChanged(); //设置变化点
        notifyObservers(); //通过这个方法通知观察者事件
        doothers(); //必须等上面观察者实现了其响应行为才执行本方法。
    }

```

而使用 `com.jdon.async.observer.ObservableAdapter` 和 `com.jdon.async.observer.TaskObserver` 则可以实现异步的观察者，这样，上面方法 `dooters` 就不必等待观察者 `Observer` 做完它的事情才能继续，两者同时运行。

异步观察者模式具体使用见 JiveJdon3.7 版本中的 `Subscription` 关注订阅模型，当主题有新内容时，通知关注者，我们在 `ForuThread` 的 `addNewMessage` 方法中加入观察激活点，当有人在这个主题下发新回帖时，激活这个观察点，从而异步激活订阅通知处理功能，由它来检查哪些人关注订阅了这个主题，然后给他们逐个发送消息或邮件，这个过程就不会影响发新回帖处理过程，保证发新回帖处理过程快速完成。

异步观察者模式步骤总结：

1. 继承 `TaskObserver`，实现其 `action` 方法，这是激活后所要实现的方法。
2. 将观察者 `TasjObserver` 加入 `ObservableAdapter`。
3. 将设置观察点，在被观察或监听的类中，调用 `ObservableAdapter`，在具体激活方法中调用 `ObservableAdapter` 的 `notifyObservers` 方法。

上面三个部分是松耦合，可以分别在代码三个地方执行，有时也可以合并一起：

```

private void sendComposer(EmailTask emailTask) {
    //创建一个观察者 继承TaskObserver
    EmailTaskListerner emailTaskListerner = new
        EmailTaskListerner(emailTask);

    //创建一个ObservableAdapter
    ObservableAdapter subscriptionObservable = new
        ObservableAdapter(ep);

    //将观察者TasjObserver加入ObservableAdapter
    subscriptionObservable.addObserver(emailTaskListerner);

    //激活出发
    subscriptionObservable.notifyObservers(null);
}

```

关于领域模型设计和相关监视事件，见有关 Qi4j 讨论，Jdon 框架和 Qi4j 一样已经考虑到此类实践问题：<http://www.jdon.com/jivejdon/thread/37186>

## 6.2 版本 AOP

6.2 版本架构改变比较大，最大变化是增强了 AOP 功能，具体如下：

1. 使用 `CGLIB` 替代 `JDK` 的动态代理，性能有所提高：

<http://www.jdon.com/jivejdon/thread/37330>

3. 增强了 AOP 功能，提供基于 `Annotation` 的方法拦截功能，但是 AOP 功能和 `Spring/Aspect` 的有些区别，主要是在被拦截类进行定义，如下：

```
@Introduce("c")
```

```

public class A implements AInterface {

    @Before("testOne")
    public Object myMethod(@Input() Object inVal, @Returning() Object returnVal) {
        System.out.println("this is A.myMethod is active!!!! ");
        return inVal + "myMethod" + returnVal;
    }

    @After("testWo")
    public Object myMethod2(Object inVal) {
        System.out.println("this is A.myMethod2 is active!!!! ");
        return "myMethod2" + inVal;
    }

}

```

通过元注解@Introduce 引入拦截器 c，c 是一个组件的名称，用 Component(“c”)定义的类，@Before("testOne")表示 c 这个类的 testOne 方法将在当前方法 myMehtod 之前执行，@After 则表示是以后执行。

使用这种引入式定义的拦截器 c 类不需要特别其他配置，只是一个普通组件而已，如下：

```

@Component("c")
public class C {

    public Object testOne(Object inVal) {
        System.out.println("this is C.testOne " + inVal);
        return " interceptor" + inVal;
    }

    public Object testWo(Object inVal) {
        System.out.println("this is C.testWo ");
        return inVal + " interceptor";
    }

}

```

6.2 版本还提供了一种 Around 方式的拦截，就是在包围原始方法前后都执行的方法，引入的类方式和 Before 和 After 类似：

```

@Introduce("aroundAdvice")
public class D implements DInterface {

    @Around
    public Object myMethod3(Object inVal) {
        System.out.println("this is D.myMethod3 is active!!!! ");
        return "myMethod3" + inVal;
    }

}

```

```
}
```

但是拦截器 `aroundAdvice` 写法就不是普通类的写法，需要实现接口 `org.aopalliance.intercept.MethodInterceptor`，完成其方法，同时必须有类 Annotation：`@Interceptor`，整个内容如下：

```
@Interceptor("aroundAdvice")
public class AroundAdvice implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.print("\n\n this is AroundAdvice  before \n\n");

        if (isAdviceAround(invocation.getThis().getClass()))
            System.out.print("\n\n this is AroundAdvice  myMethod3 \n\n");

        Object o = invocation.proceed(); //这句关键表示执行原始方法
        System.out.print("\n\n this is AroundAdvice after \n\n");
        return o;
    }
}
```

以上 6.2 版本新增的 **Before After Around** 拦截方式的编写规则，使用 Java 代码替代复杂的 `ponitcut` 规则表达式。简单易用。

另外还有一种和 Spring 类似，在拦截器类中配置被拦截的点 `Pointcut` 的方式，使用这个方式，就无需象上面在被拦截的地方进行注解，解放了被拦截类，而上面这种写法是解放了拦截类，灵活应用。

```
@Interceptor(name = "myInterceptor", pointcut = "a,c")
public class MyInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation methodInvocation) throws
java.lang.Throwable {
        System.out.print("\n\n this is MyInterceptor  before \n\n");
        if (methodInvocation.getMethod().getName().equals("myMethod")) {
            System.out.print("\n\n this is MyInterceptor for a.myMethod  \n\n");
        }
        Object o = methodInvocation.proceed();
        System.out.print("\n\n this is MyInterceptor after \n\n");
        return o;
    }
}
```

拦截器必须实现 `MethodInvoker` 接口，而且必须标注 `@Interceptor(name = "myInterceptor", pointcut = "a,c")`，其中名称 `name` 是拦截器自己的名称，而 `pointcut` 是被拦截类的 `@Component("a")`或`@Component("c")`。

如果拦截所有对象，那么就直接`@Interceptor()`；没有任何参数，不过这个拦截所有对象的意思不是在所有对象只要被调用就激活，而是客户端通过 `WebApp` 进行对 `Jdon` 框架

通过 Annotation 标注的这些 Component 或 @Service 时才会激活。

更多见 [AOP 章节](#)。

## 6.2 版本领域消息

见前面[领域模型章节](#)。

增加了组件生命周期，组件可以在当前 Web 应用从服务器中 undeploy，或者当前服务器也就是 Web 容器 shutdown 正常关闭时，将自动执行组件类中的 stop()方法，我们可以在 stop 方法中将一些没有保存持久化的数据进行保存持久，防止丢失。

组件类只要实现接口：com.jdon.container.pico.Startable，完成其中 start()和 stop()方法即可。

以 JiveJdon4 中的帖子浏览数为案例，每个帖子每次阅读浏览都要计数一次，如果每次技术都要保存数据库，无疑在大浏览量情况下会加载数据库负载，这时我们将浏览次数保存在内存中，也就是组件类的字段中即可，然后每隔一定时间比如一个小时保存一次数据库，如果遇到应用维护需要正常 redeploy 或服务器关闭，我们就在 stop 方法在那个，把内存中计数值一次性保存到数据库中。

见 com.jdon.jivejdon.manager.listener.ThreadViewCountManager 类代码。

## 技术支持

无论 Jdon 框架本身还是使用 Jdon 框架开发的任何系统出现问题，都可以在论坛

<http://www.jdon.com/jivejdon/forum.jsp?forum=61>

发生错误后，请按下面步骤贴出错误：

1. 需要打开 Jboss/server/default/log/server.log
2. 键入搜索" ERROR "，注意 ERROR 两边各有一个空格，找到第一个错误，那是错误起源，贴到论坛中。

### Weblogic + Oracle 环境下使用注意点

注意两点：

1. Oracle 字符串类型很特殊，它和 Java 的 JDBC 字符串类型有些特别，Oracle 字符串如果是 10 位，而 Java 传给它是两个字母 ab，你必须将剩余 8 个字母用空格填满，否则，出现“查询页面不显示记录，但是可以向数据库里插入记录”现象。

建议 oracle 数据库主键使用 Number，模型 Model 主键使用 Long 即可。

2. log4j 需要正常在 Weblogic 中显示，配置不是非常容易，按本[说明书相应章节](#)耐

心配置。

总之，试验一个 Jdon 框架技术应该分开几个环节来分别测试：

1. 使用 Jdon 框架推荐的环境测试。
2. 使用自己特殊的数据库测试。
3. 使用自己特殊的服务器测试。

从这几个环境中，可以分别了解差异产生的原因。

## 开发环境配置

为方便初学者快速入手使用 Jdon 框架开发 J2EE/Java EE 系统，现将开发环境配置和软件下载陈述如下：

下载 Java 运行环境 也就是 JDK1.4 或 JDK5.0

下载 Eclipse IDE 开发软件，因为 Eclipse 插件非常多，安装后不一定能用，可在本站 VIP 下载区下载完整立即可用的 [Eclipse 压缩包](#)。

下载运行环境，使用 Tomcat/JBoss/Weblogic 等等都可以，本站提供 JBoss + MySQL 的[现成压缩包](#)。注意，需要将最新版本的 Jdon 框架拷贝进去，方法见前面[安装章节](#)。

以上开发环境安装完成后，你可打开 Jdon 框架的案例项目进行研究学习，祝你成功。

## 配置启动 Jdon 框架

(1) 配置 jdonframework.xml 在 struts-config.xml 中。

```
<plug-in className="com.jdon.strutsutil.InitPlugIn">
  <set-property property="modelmapping-config" value="jdonframework.xml 完整路径" />
</plug-in>
```

(2) 将 jdonframework.xml 打包到你的 Web 项目。

否则，后台控制台会出现：

[InitPlugIn] looking up a config: jdonframework.xml

[InitPlugIn] cannot locate the config:: jdonframework.xml

## 打包部署

有两种方式打包成部署文件.war 或.ear。

第一：如果是 Eclipse 工具，使用 Ant 进行打包，在本系统源码中已经编制好 build.xml 文件，更改其中 J2EE 服务器为的路径后，即可直接运行 Ant 打包。

第二：通过 JBuilder 直接打包，点本项目的 Web 项目，选择 make 即可，将自动出现 xxx.war 部署包(如果你是 Tomcat 就不会出现 war 文件，只能安装 Tomcat 方式配置)。

注意，将 Web 项目中去除任何包，也就是说 WEB-INF/lib 下没有任何包，Struts 和 JdonFramework 包都放置在 JBoss 的 server/default/lib 下。

如果出现与 tld 相关的错误，如：

File "/WEB-INF/MultiPages.tld" not found

解决：在 web.xml 中应该有下面两行 tld 配置：

```
<taglib>
  <taglib-uri>/WEB-INF/MultiPages.tld</taglib-uri>
  <taglib-location>/WEB-INF/MultiPages.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/TreeView.tld</taglib-uri>
  <taglib-location>/WEB-INF/TreeView.tld</taglib-location>
</taglib>
```

并且确保/WEB-INF/MultiPages.tld 和/WEB-INF/TreeView.tld 下文件存在，这两个 MultiPages.tld 和 TreeView.tl 在 JdonFramework.jar 中的 META-INF/tlds 中，可手工拷贝过去。

使用 JBuilder2005 直接打包时，缺省是不会将所有的编译好 class 打包进去的，同时也不会将 jdonframework.xml 等资源文件打包进去，我们需要经过特别步骤处理：

选择项目的 Web 模块如 testweb，右键点选属性，选择其中的 Content，点选 include all classes and resources，在 Content 子节点中，选择 Dependencies，将所有包都 exclude all，因为这些包我们事先已经放置在 JBoss 的 server/default/lib 下，不需要在本 Web 模块中携带。

确定后，重新编译一次项目，检查 Web 模块如 testweb 下的 testweb.war 文件，展开后，检查是否有 jdonframework.xml，如果没有，接着如下步骤：

再次选择项目的 Web 模块如 testweb，右键点选属性，选择 Content，这次选择 include speified filtered file and resource，然后点按 Add Filters，输入\*\*/\*.，表示全部资源，确定。

再到 Add Filters 下面选择 Add File，将 jdonframework.xml 等资源文件加入。

再次编译整个项目，检查 war 文件，应该所有 class 和资源文件都包括其中了。

更多详细错误，打开 JBoss 日志文件：server/default/log/server.log，找到 server.log 中第一个 ERROR 信息，分析原因，或者贴到 Jdon 框架开发论坛中。

## 运行案例

打开浏览器，键入 <http://localhost:8080/testWeb/index.jsp>

我们将 index.jsp 导向真正执行/userListAction.do

网上实时演示网址：<http://www.jdon.com:8080/testWeb/>

当在 JBoss 或 Tomcat 控制台 或者日志文件中出现下面字样标识 Jdon 框架安装启动成功：

```
<===== Jdon Framework started successfully! =====>
```

## 性能压力测试

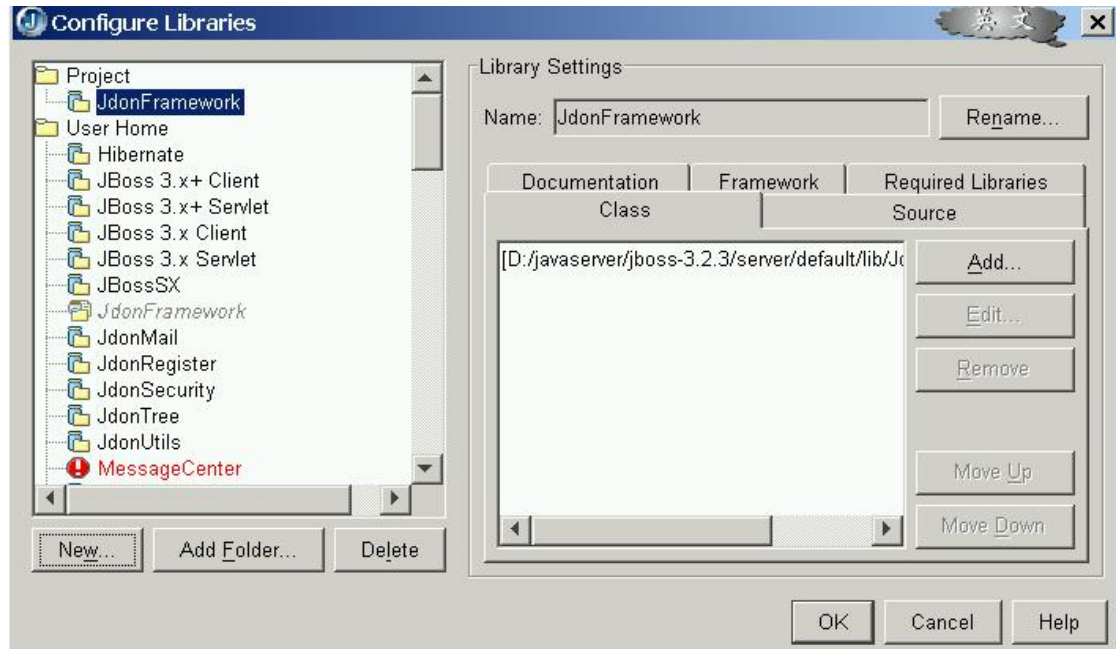
用户自己对 Jdon 框架应用案例的测试心得：

<http://www.jdon.com/jivejdon/thread/32894.html>

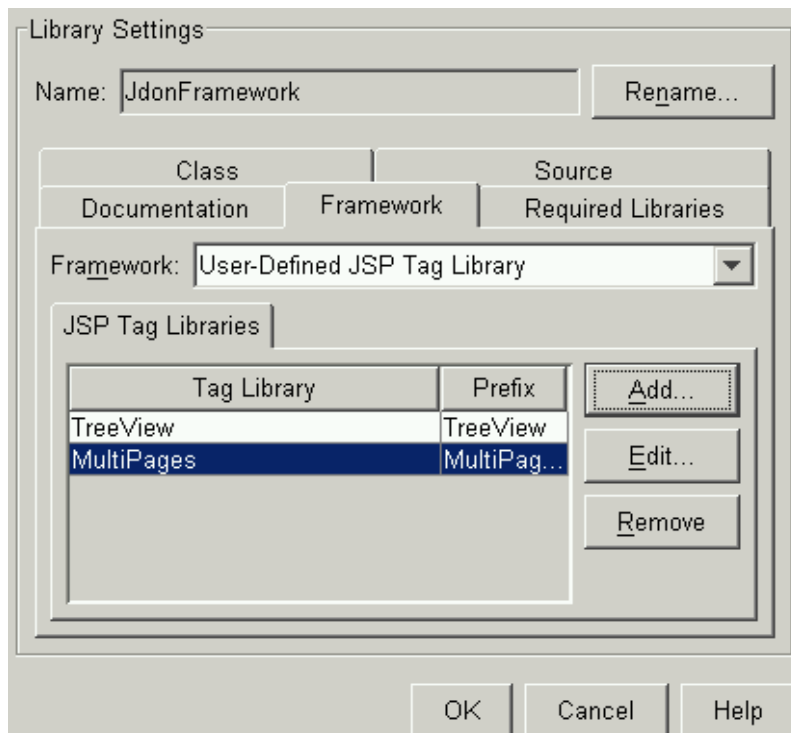
## 附：JBuilder 环境配置

如果你使用 JBuilder 开发工具，按下列步骤：

将 JdonFramework 导入项目库，或者成为整个配置库。如图：



同时，配置 Framework 配置，如下图：

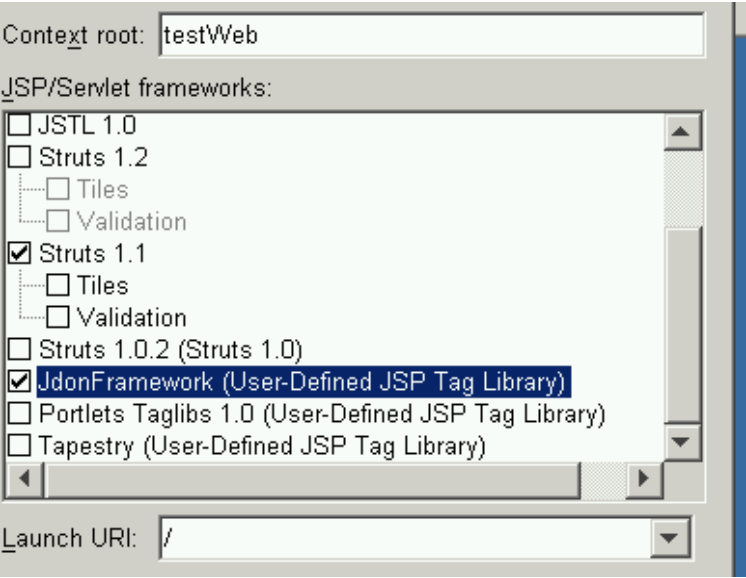


该配置是为了激活项目 Web 开发支持 JdonFramework 的标签库 taglib，使用 JdonFramework 开发应用系统时，自动支持多页批量查询和树形结构显示，这两种功能需要在 Jsp 页面导入特定的标签库，该设置就是为实现此功能。

注意：由于 JBuilder 的部分 BUG，最好将 JdonFramework.jar 的 MEAT-INF/tlds 目录下的 MultiPages.tld 等文件解压出，形成单独文件，在这里导入这些单独文件。



配置 WEB 模块时，需选中 Web 框架，如下图：



如果你使用的是 Eclipse，在将 JdonFramework.jar 作为一般的 JAR 导入项目。

另外，如果你的 JBuilder 中看不到源码包下面的 jdonframework.xml 文件，例如本项目的配置文件在 com.jdon.framework.test 包下面，但是你展开后，可能只发现 application.properties 等资源文件，但是没有发现 jdonframework.xml，解决方式：

项目属性中，Build 节点的子节点 Resource，在下拉菜单中选择 XML，然后点选右上部的 copy 即可（原来为 do not copy）。

企业培训内容

我们提供 Jdon 框架上门培训和咨询，通过 4 天培训让企业用户完全掌握使用 Jdon 框架开发 J2EE 信息系统，总报价：14400 元 由 Jdon 框架设计者彭晨阳亲赴贵公司培训，时间长短可调节。

|  |   |
|--|---|
| 第 1 天 （6 课时）<br>目标：掌握 Jdon 框架设计原理和开发特点         |   |
| 培 训<br>内容                                      | J2EE 多层架构系统简介<br>J2EE 表现层、组件层 Model 和持久层特点介绍<br>传统 J2EE 开发优缺点比较<br>Jdon 框架的总体特点介绍<br>Struts 技术特点介绍<br>Hibernate 技术特点介绍<br>Jdon 设计原理简介 |
| 第 2 天 （6 课时）<br>目标：通过简单开发演示让学员了解 Jdon 框架整体开发步骤 |   |
| 培 训<br>内容                                      | 现场命题一个简单业务需求<br>域建模分析<br>权限登录机制设计   |

|   |   |
|---|---|
|   | 业务层功能特点分类和设计<br>模型的增删改查功能实现<br>批量分页查询功能实现<br>调试运行   |
| 第3天（6课时）<br>目标：Jdon 框架开发特点和步骤详解                 |   |
| 培训<br>内容  | 开发步骤总体介绍<br>域建模过程和分析特点以及与数据库建模区别<br>Ioc/Aop 微容器原理<br>Jdon 框架组件构件配置重点和原理<br>模型 Model 编码特点<br>CRUD 实现原理<br>CRUD 复杂程度不同的开发场景设计<br>Jdon 框架批量查询特点<br>数据批量查询最优设计和缓存设计   |
| 第4天（6课时）<br>目标：通过复杂案例讲解让学员进一步掌握使用 JdonFramework |   |
| 培训<br>大纲  | <p>基于 JdonFramework 快速开发 J2EE 系统的注意点。</p> <p>以 JiveJdon3.0 或网上商店为例，讲解基于 JdonFramework 设计开发一个复杂的 J2EE 系统的全过程：包括架构设计、直至最终实现。让学员能够完全理解复杂系统源码，并能重用这些设计思想和方法。</p> <p>针对具体项目咨询，学员结合这几天所学，并结合项目特点，向老师征求一些实际应用方面的意见，根据以往类似项目的经验，提出一些具有建设性的意见。</p> |

全文完