Beetle J2EE Application Framework

开发指南

01278576189468.png

（Version **3.0.1**）

Copyright © 2001-2016

BeetleSoft, ALL RIGHTS RESERVED.

甲壳虫软件版权所有

网站：<http://www.mc2e.cc>

<https://github.com/jbeetle/BJAF3.x>

邮件：[yuhaodong@gmail.com](mailto:yuhaodong@gmail.com)

[henryyu@163.com](mailto:henryyu@163.com)

目录

[1.BJAF概述 9](#_Toc358126485)

[1.1BJAF是什么 9](#_Toc358126486)

[1.1.1运行环境要求 11](#_Toc358126487)

[2.持久层数据访问 12](#_Toc358126488)

[2.1数据源配置 13](#_Toc358126489)

[2.2数据存取 22](#_Toc358126490)

[2.2.1单表操作器 23](#_Toc358126491)

[2.2.2查询操作器 40](#_Toc358126492)

[2.2.3更新操作器 51](#_Toc358126493)

[2.2.4存储过程操作器 56](#_Toc358126494)

[2.2.5序列操作器 61](#_Toc358126495)

[2.2.6分页操作器 65](#_Toc358126496)

[2.2.7复杂条件组合查询器 71](#_Toc358126497)

[2.3DAO模式支持 75](#_Toc358126498)

[2.3.1什么是DAO 75](#_Toc358126499)

[2.3.2模式使用 76](#_Toc358126500)

[2.4持久层代码自动生成 84](#_Toc358126501)

[3.业务层 87](#_Toc358126502)

[3.1业务层介绍 87](#_Toc358126503)

[3.2DIC依赖注入容器 88](#_Toc358126504)

[3.2.1 InjectField注解 90](#_Toc358126505)

[3.2.2 Aop注解 91](#_Toc358126506)

[3.2.3 ServiceTransaction注解 94](#_Toc358126507)

[3.3 Command框架 95](#_Toc358126508)

[3.2.1框架模型 95](#_Toc358126509)

[3.2.2使用示例 101](#_Toc358126510)

[3.4 Service服务框架 111](#_Toc358126511)

[3.4.1 Service界定 111](#_Toc358126512)

[3.4.2 Service模型 111](#_Toc358126513)

[3.4.3使用示例 114](#_Toc358126514)

[4.Web表示层 119](#_Toc358126515)

[4.1Web框架配置 120](#_Toc358126516)

[4.2控制器 126](#_Toc358126517)

[4.2.1控制器介绍 126](#_Toc358126518)

[4.2.2配置控制器 128](#_Toc358126519)

[4.2.3标准视图控制器 133](#_Toc358126520)

[4.2.4非标准视图控制器 138](#_Toc358126521)

[4.2.5虚拟控制器 141](#_Toc358126522)

[4.2.6文件上传控制器 143](#_Toc358126523)

[4.2.7页面绘图控制器 152](#_Toc358126524)

[4.2.8文档视图控制器 157](#_Toc358126525)

[4.2.9多动作控制器 164](#_Toc358126526)

[4.2.10 Web Service控制器 173](#_Toc358126527)

[4.2.11面板控制器 177](#_Toc358126528)

[4.3视图显示 189](#_Toc358126529)

[4.3.1标准JSP视图 189](#_Toc358126530)

[4.3.2视图配置 190](#_Toc358126531)

[4.3.3标准视图Model数据解析 193](#_Toc358126532)

[4.3.4 freemarker模板支持 200](#_Toc358126533)

[4.4数据绑定与校验 202](#_Toc358126534)

[4.4.1页面数据绑定 202](#_Toc358126535)

[4.4.2页面数据校验 203](#_Toc358126536)

[4.5请求动态缓存功能 208](#_Toc358126537)

[4.6AOP横切编程 210](#_Toc358126538)

[4.7其它功能与特性 220](#_Toc358126539)

[4.7.1 Web应用启动/关闭接口支持 220](#_Toc358126540)

[4.7.2防止表单重复提交 223](#_Toc358126541)

[4.7.3防止手工URL绕过验证进行请求访问 224](#_Toc358126542)

[4.7.4设置视图的缓存机制 225](#_Toc358126543)

[4.7.5页面验证码支持 226](#_Toc358126544)

[4.7.6特定请求并发控制 229](#_Toc358126545)

[4.7.7错误处理视图 230](#_Toc358126546)

[4.7.8控制器、视图及两者引用关系分析 232](#_Toc358126547)

[4.8Web应用零配置编程 237](#_Toc358126548)

[5.应用服务程序开发 243](#_Toc358126549)

[5.1简介 243](#_Toc358126550)

[5.2应用程序模块 244](#_Toc358126551)

[5.3线程的简化编程模型 253](#_Toc358126552)

[5.4子任务及其执行方式 257](#_Toc358126553)

[5.4.1子任务说明示例 257](#_Toc358126554)

[5.4.2执行方式说明及示例 262](#_Toc358126555)

[5.5插件开发和管理 274](#_Toc358126556)

[5.6定时计划任务 277](#_Toc358126557)

[6.CoreUtil 287](#_Toc358126558)

[6.1资源管理 287](#_Toc358126559)

[6.1.1公共配置资源属性 288](#_Toc358126560)

[6.1.2 JNDI资源查找 293](#_Toc358126561)

[6.2日志组件 297](#_Toc358126562)

[6.3邮件发送组件 301](#_Toc358126563)

[6.4常见设计模式组件 303](#_Toc358126564)

[6.4.1责任链设计模式 303](#_Toc358126565)

[6.4.2观察者设计模式 313](#_Toc358126566)

[6.5简单缓存器 318](#_Toc358126567)

[6.6其它工具类 319](#_Toc358126568)

[7.BJAF应用部署 320](#_Toc358126569)

[7.1灵活部署方案 320](#_Toc358126570)

[7.1.1 集中式 320](#_Toc358126571)

[7.1.2 Web层和业务层分离 321](#_Toc358126572)

[7.1.3 多Web层 322](#_Toc358126573)

[7.1.4 多web层多业务层 322](#_Toc358126574)

[7.2独立打包配置文件存放目录截图示意 323](#_Toc358126575)

[7.2.1 WAR应用截图 324](#_Toc358126576)

[7.2.2 EAR应用截图 325](#_Toc358126577)

[7.2.3 独立进程应用截图 329](#_Toc358126578)

[8.后记 331](#_Toc358126579)

# 1.BJAF概述

当前J2EE技术虽然说十分成熟，但可惜的是在实际中很多采取J2EE技术开发的信息系统效果不尽人意，其投资与回报往往令人失望。为了解决J2EE开发领域中所遇到的各种难题，市场上涌现了不少J2EE应用开发框架，BJAF甲壳虫J2EE应用框架便是其中的优秀一员。

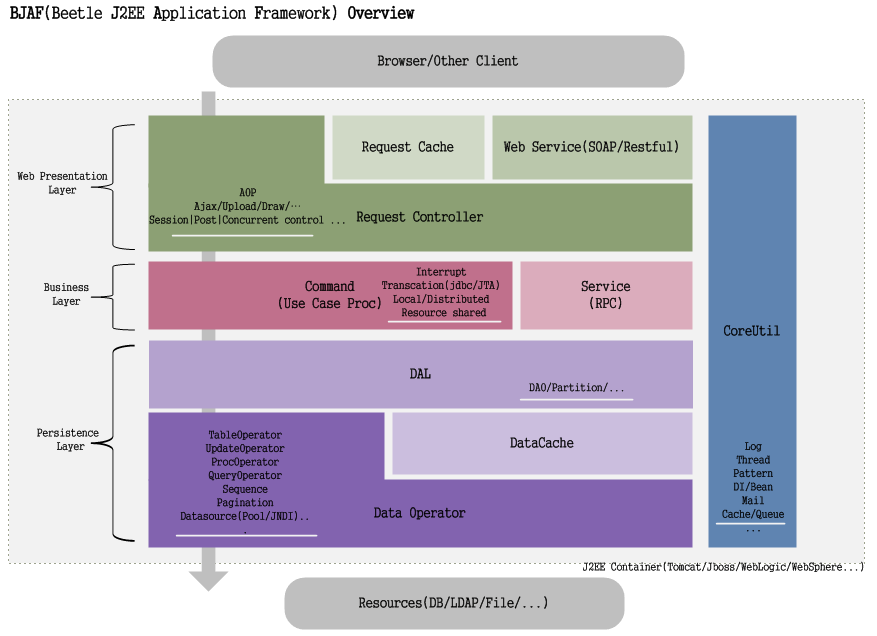
BJAF对J2EE体系结构中各层次的技术进行了透明的封装，为开发人员提供了一套灵活缩放、高可靠、可扩展、高性能的J2EE企业级应用开发的解决方案。

在本开发指南中，我们会向读者展现和介绍BJAF框架中各个功能模块及其具体的技术细节，以便读者使用BJAF框架快速投入具体项目开发中。

## 1.1BJAF是什么

BJAF 是甲壳虫 J2EE 应用框架（**B**eetle **J**2EE **A**pplication **F**ramework）的简写，它并不是一个可以即时看见和运行的应用系统，它为构建于 J2EE 之上的应用系统定义了一个固定而有效的设计开发框架，简化 J2EE 应用，尤其加速了 J2EE 应用的开发过程。

BJAF 的最终目标是为开发人员提供一个填空式的开发框架。让开发人员在 BJAF 架构下，只需关注编写和具体业务逻辑相关的程序，而将业务无关的需求（非功能需求，non-functional requirement）交给 BJAF 来完成。



BJAF整体架构

### 1.1.1运行环境要求

* BJAF要求使用JDK1.6或以上。
* Servlet容器：Tomcat、Resin、Jetty等
* J2EE容器：JBoss、Weblogic、Websphere等
* 如果不需要Web、则可不依赖于任何容器，自身带了基于RPC的Service容器

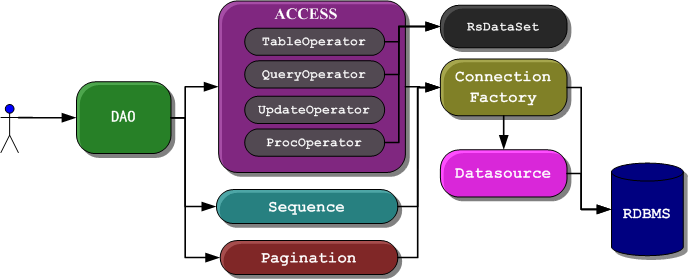
# 2.持久层数据访问

在J2EE应用中，持久层负责企业资源持久化处理。这些企业数据源包括：文件系统数据、邮件服务器、企业遗留应用及数据库管理系统（DBMS）等等。关系数据库管理系统（RDBMS）又是其中最主要、最重要、应用最为广泛的数据源，对其数据访问的策略将是决定一个J2EE应用成败的关键因素。因而，BJAF框架在持久层重点关注是RDBMS数据存取的解决方案，并实现了一个基于JDBC数据存取框架。

与Hibernate、JPA等O/R映射框架不同，它的设计理念脱胎于微软的ADO数据存取框架，它关注的是数据存取的效率、实用和易用性。它具备以下主要功能特征：

* 封装JDBC低级API，类似于微软的ADO框架，结构简单、便于开发、调试和维护。
* 支持数据的批量更新，支持数据库存储过程的操作，支持标准化的结果集处理。预编译SQL 语句处理。
* 支持单个数据表CRUD（create增、retrieve查、update改、delete删）等方便操作，无需编写SQL语句。
* 支持结果集与值对象的自动装配。
* 可动态配置数据源，支持通过JNDI服务J2EE容器DataSource，框架本身自带数据源。
* 无须依赖J2EE容器，支持数据库连接池，支持XA数据源。
* 支持常见数据库（如：oracle、sqlserver、sybase、mysql等）的无状态数据分页查询。
* 支持数据库Sequence特性，提供自己维护序列的接口。
* 支持DAO设计模式。
* 保证数据访问对象线程安全，防止并发访问危及数据完整性。
* 高速缓存只读数据，提高系统性能。

其框架组件结构示意图如下：



数据存取框架组件示意图

从上图可知，BJAF数据存取框架与RDBMS关系数据库系统交互主要提供三大组件：

Access数据访问操作组件，包括：TableOperator单表操作器、QueryOperator查询操作器、UpdateOperator更新操作器、ProcOperator存储过程操作器。

Sequence序列号（唯一号）生成器组件。Pagination分页查询器组件。

其中，查询结果统一由RsDataSet结果集处理器负责处理；Datasource数据源组件负责协调各种不同数据库访问方式；ConnectionFactory组件负责管理数据库连接的产生和销毁。

最后，结合DAO设计模式思想，统一通过DAO组件把具体的数据访问接口暴露给用户。

## 2.1数据源配置

在进行数据库编程，首要问题是解决数据源的问题。BJAF数据存取框架支持多种数据源访问形式。主要体现为J2EE应用服务器容器数据源、开发调试数据源和BJAF框架自带数据源3种情形：

J2EE容器数据源

像WebLogic、JBoss、WebSphere等J2EE应用服务器都有各自的标准数据源的实现，对这些J2EE容器，BJAF框架提供了标准的JDNI方式来访问其数据源。而且，在生产环境，如果应用部署在J2EE容器下，那么我们建议采用J2EE容器本身的数据源。

开发调试数据源

开发调试数据源是框架为了在IDE环境下方便程序调试而建立的单连接数据源。

框架自带数据源

BJAF针对没有J2EE数据源容器的情况，实现了以下各种数据源（连接池）：



自实现数据源，如果觉得BJAF框架提供现有数据源不满足要求，可以自己定制开发一个数据源（或自己写代码集成其它第三方数据库连接池），BJAF对外提供了IConnPool接口，开发人员只要编写其实现类，然后把这个实现在< DataSources >标签里面的< pool-imp >值注册一下就可以了。

数据源统一在项目工程config目录下DBConfig.xml文件中配置。其定义格式如下：

|  |
| --- |
| <?xml version=**"1.0"** encoding=**"UTF-8"**?>  <Config>  <DataSources>  <SYSDATASOURCE\_DEFAULT>  <!--  使用模式  1-使用J2EE容器数据源模式;  2-使用本框架自带数据源模式;  3-使用调试模式  -->  <item name=**"use-mode"** value=**"2"** />  <!--  模式1下，j2ee容器所配置的数据源名称，其他模式下可删除此项  -->  <item name=**"jndi-name"** value=**"xxxDs"** />  <!--  模式2下，连接池实现具体实现类（类型），其他模式下可删除此项  1-driver[com.beetle.framework.persistence.access.datasource.DriverPool]框架自实现，功能简单，兼容性，用于测试环境;  2-CommonPool[com.beetle.framework.persistence.access.datasource.CommonPool]通用连接池，使用各个数据库厂商驱动自实现的连接池，性能良好，用于兼容性强的场景;  3-BoneCP[com.beetle.framework.persistence.access.datasource.BoneCPPool]号称当前性能最好的连接池[实际环境发现兼容性不够好，问题挺多，要根据自己具体情况而定]。  4-XaPool[com.beetle.framework.persistence.access.datasource.XaPool]atomikos开源Xa数据源连接池，事务跨越2个或以上数据源的场景，采取此连接池,兼容性及性能良好  5-NonXaPool[com.beetle.framework.persistence.access.datasource.NonXaPool]atomikos开源NonXa数据源连接池，兼容性及性能良好  6-TomcatJdbcPool[com.beetle.framework.persistence.access.datasource.TomcatJdbcPool]tomcat7.x版本实现的连接池，性能良好，超越commons-dbcp和c3p0[推荐]  内部性能测试显示3>6>5>2>1  -->  <item name=**"pool-imp"**  value=**"com.beetle.framework.persistence.access.datasource.TomcatJdbcPool"** />  <item name=**"pool-minsize"** value=**"5"** />  <item name=**"pool-maxsize"** value=**"10"** />  <!-- 测试sql语句，空隙时，有利于连接池检查连接状态 -->  <item name=**"test-sql"** value=**"select 1"** />  <item name=**"connection-url"** value=**"jdbc:mysql://127.0.0.1:3306/demodb"** />  <item name=**"driver-class"** value=**"com.mysql.jdbc.Driver"** />  <item name=**"user-name"** value=**"root"** />  <!--  密码掩码实现类，可选配置，不配置代表密码不加密，同时密码[password]的值必须采取"mask{掩码}"格式  框架提供了一个默认掩码加密实现[com.beetle.framework.persistence.access.DefaultPassworkMask]  用户也可以通过实现[com.beetle.framework.resource.mask.IPasswordMask]接口来扩展  -->  <item name=**"mask-imp"**  value=**"com.beetle.framework.resource.mask.DefaultPassworkMask"** />  <!-- 用户密码，采取mask{xxx}格式代表值是掩码，此时必须提供mask-imp实现类 -->  <item name=**"password"** value=**"760224"** />  <!--  <item name="password" value="mask{7b55ef6a3c0c4be172e29384b6e6c5f1ca9d620eb1580489e3c888f49cec9904}" />  -->  <!--连接池特有属性配置-->  <item name=**"specific-attribute"** value=**"key1=value1#key2=value2#key3=value3"** />  </SYSDATASOURCE\_DEFAULT>  <h2ds>  <item name=**"use-mode"** value=**"2"** />  <item name=**"pool-imp"**  value=**"com.beetle.framework.persistence.access.datasource.TomcatJdbcPool"** />  <item name=**"pool-minsize"** value=**"5"** />  <item name=**"pool-maxsize"** value=**"10"** />  <item name=**"test-sql"** value=**"select 1"** />  <item name=**"connection-url"** value=**"jdbc:h2:tcp://localhost/~/demo"** />  <item name=**"driver-class"** value=**"org.h2.Driver"** />  <item name=**"user-name"** value=**"sa"** />  <item name=**"mask-imp"**  value=**"com.beetle.framework.resource.mask.DefaultPassworkMask"** />  <item name=**"password"** value=**"mask{9bf99f38dd0ec8d861ed984a7a9890a8ca9d620eb1580489e3c888f49cec9904}"** />  </h2ds>  <oracleds>  <item name=**"use-mode"** value=**"2"** />  <item name=**"pool-imp"**  value=**"com.beetle.framework.persistence.access.datasource.CommonPool"** />  <item name=**"pool-minsize"** value=**"5"** />  <item name=**"pool-maxsize"** value=**"10"** />  <item name=**"test-sql"** value=**"select sysdate from dual"** />  <item name=**"connection-url"** value=**"jdbc:oracle:thin:@10.79.11.169:1521:npdb"** />  <item name=**"driver-class"** value=**"oracle.jdbc.driver.OracleDriver"** />  <item name=**"user-name"** value=**"dm\_ord"** />  <item name=**"password"** value=**"dm\_ord"** />  </oracleds>  <SYSDATASOURCE\_ORACLE\_XA>  <item name=**"use-mode"** value=**"2"** />  <item name=**"pool-imp"**  value=**"com.beetle.framework.persistence.access.datasource.XaPool"** />  <item name=**"pool-minsize"** value=**"10"** />  <item name=**"pool-maxsize"** value=**"50"** />  <item name=**"test-sql"** value=**"select sysdate from dual"** />  <item name=**"connection-url"** value=**"jdbc:oracle:thin:@10.25.10.175:1555:rmtest"** />  <item name=**"driver-class"** value=**"oracle.jdbc.xa.client.OracleXADataSource"** />  <item name=**"user-name"** value=**"rmdata"** />  <item name=**"password"** value=**"hhxxttxs"** />  </SYSDATASOURCE\_ORACLE\_XA>  <SYSDATASOURCE\_SQLSERVER>  <item name=**"use-mode"** value=**"3"** />  <item name=**"pool-imp"**  value=**"com.beetle.framework.persistence.access.datasource.BoneCPPool"** />  <item name=**"pool-minsize"** value=**"2"** />  <item name=**"pool-maxsize"** value=**"5"** />  <item name=**"test-sql"** value=**"select CURRENT\_DATE"** />  <item name=**"connection-url"**  value=**"jdbc:sqlserver://10.25.17.68:52188;databaseName=demo"** />  <item name=**"driver-class"** value=**"com.microsoft.sqlserver.jdbc.SQLServerDriver"** />  <item name=**"user-name"** value=**"sa"** />  <item name=**"password"** value=**"asdF1234"** />  </SYSDATASOURCE\_SQLSERVER>  <SYSDATASOURCE\_postgreSql>  <item name=**"use-mode"** value=**"2"** />  <item name=**"pool-imp"**  value=**"com.beetle.framework.persistence.access.datasource.CommonPool"** />  <item name=**"pool-minsize"** value=**"12"** />  <item name=**"pool-maxsize"** value=**"50"** />  <item name=**"test-sql"** value=**"select CURRENT\_DATE"** />  <item name=**"connection-url"** value=**"jdbc:postgresql://localhost:5432/demodb"** />  <item name=**"driver-class"** value=**"org.postgresql.Driver"** />  <item name=**"user-name"** value=**"postgres"** />  <item name=**"password"** value=**"yhd@1976"** />  </SYSDATASOURCE\_postgreSql>  <SYSDATASOURCE\_demodb>  <item name=**"use-mode"** value=**"2"** />  <item name=**"pool-imp"**  value=**"com.beetle.framework.persistence.access.datasource.CommonPool"** />  <item name=**"pool-minsize"** value=**"2"** />  <item name=**"pool-maxsize"** value=**"5"** />  <item name=**"test-sql"** value=**"select CURRENT\_DATE"** />  <item name=**"connection-url"** value=**"jdbc:postgresql://localhost:5432/demodb"** />  <item name=**"driver-class"** value=**"org.postgresql.Driver"** />  <item name=**"user-name"** value=**"postgres"** />  <item name=**"password"** value=**"yhd@1976"** />  </SYSDATASOURCE\_demodb>  </DataSources>  <Extensions>  <SYSDATASOURCE\_DEFAULT>  <!-- 序列号实现类型 Common, Oracle, DB2, PostgreSql,MySql,H2, Other; -->  <item name=**"seq-type"** value=**"MySql"** />  <!-- 查询分页实现类  1-mysql[com.beetle.framework.persistence.pagination.imp.MysqlPaginationImp]  2-sysbase[com.beetle.framework.persistence.pagination.imp.SybasePaginationImp]  3-sqlserver[com.beetle.framework.persistence.pagination.imp.SqlServerPaginationImp]  4-oracle[com.beetle.framework.persistence.pagination.imp.OraclePaginationImp]  5-db2[com.beetle.framework.persistence.pagination.imp.DB2PaginationImp]  6-postgresql[com.beetle.framework.persistence.pagination.imp.PostgreSqlPaginationImp]  7-firebird[com.beetle.framework.persistence.pagination.imp.FireBirdPaginationImp]  8-H2[com.beetle.framework.persistence.pagination.imp.H2PaginationImp]  -->  <item name=**"pagination-imp"**  value=**"com.beetle.framework.persistence.pagination.imp.MysqlPaginationImp"** />  </SYSDATASOURCE\_DEFAULT>  <SYSDATASOURCE\_SQLSERVER>  <item name=**"seq-type"** value=**"Common"** />  <item name=**"pagination-imp"**  value=**"com.beetle.framework.persistence.pagination.imp.MysqlPaginationImp"** />  </SYSDATASOURCE\_SQLSERVER>  <SYSDATASOURCE\_postgreSql>  <item name=**"seq-type"** value=**"PostgreSql"** />  <item name=**"pagination-imp"**  value=**"com.beetle.framework.persistence.pagination.imp.PostgreSqlPaginationImp"** />  </SYSDATASOURCE\_postgreSql>  </Extensions>  </Config> |

配置文件中，分为两大部分：

**<DataSources>**为本框架自己的各种数据源属性定义标签，内容说明如下：

|  |  |
| --- | --- |
| **标记符号** | **意义说明** |
| use-mode | 使用模式[1-使用J2EE容器数据源模式; 2-使用本框架自带数据源模式; 3-使用调试模式 |
| jndi-name | 模式1下，j2ee容器所配置的数据源名称 |
| pool-imp | 连接池具体实现类名称 |
| pool-minsize | 连接池中，最小的连接数 |
| pool-maxsize | 连接池中，最大可达到的连接数 |
| connection-url | 数据库连接的URL，和jdbc定义的一致 |
| driver-class | 数据库驱动类 |
| user-name | 用户名 |
| password | 密码，用户密码，采取mask{xxx}格式代表值是掩码，此时必须提供mask-imp实现类；如：  <item name="password" value="760224" />  <item name="password" value="mask{7b55ef6a3c0c4be172e29384b6e6c5f1ca9d620eb1580489e3c888f49cec9904}" /> |
| test-sql | 测试sql语句，空隙时，有利于连接池检查连接状态 |
| mask-imp | 密码掩码实现类，可选配置，不配置代表密码不加密，同时密码[password]的值必须采取"mask{掩码}"格式 框架提供了一个默认掩码加密实现[com.beetle.framework.persistence.access.DefaultPassworkMask]  用户也可以通过实现[com.beetle.framework.resource.mask.IPasswordMask]接口来扩展 |

**<Extensions>**在此标记内为某个对应数据源的扩展属性定义标签，内容说明见下表：

|  |  |
| --- | --- |
| **标记符号** | **意义说明** |
| seq-type | 序列号实现类型 Common, Oracle, DB2, PostgreSql,MySql, Other; |
| pagination-imp | 查询分页实现类 1-mysql[com.beetle.framework.persistence.pagination.imp.MysqlPaginationImp]  2-sysbase[com.beetle.framework.persistence.pagination.imp.SybasePaginationImp]  3-sqlserver[com.beetle.framework.persistence.pagination.imp.SqlServerPaginationImp]  4-oracle[com.beetle.framework.persistence.pagination.imp.OraclePaginationImp]  5-db2[com.beetle.framework.persistence.pagination.imp.DB2PaginationImp] 6-postgresql[com.beetle.framework.persistence.pagination.imp.PostgreSqlPaginationImp]  7-firebird[com.beetle.framework.persistence.pagination.imp.FireBirdPaginationImp] |

## 2.2数据存取

BJAF框架针对数据库常见的交互方式特点，对底层的JDBC进行抽象封装，提供了一套高级的数据存取组件，包括：单表操作器（TableOperator）、查询操作器（QueryOperator）、更新操作器（UpdateOperator）、存储过程操作器（ProcOperator）、序列操作器（SeqOperator）和分页操作器（PaginationOperator）等，其类关系为：

**com.beetle.framework.persistence.access.operator   
类 BaseOperator**

java.lang.Object 4638364261348736729914.gif**com.beetle.framework.persistence.access.operator.BaseOperator**

**直接已知子类：**

[ProcOperator](file:///D:\com\beetle\framework\persistence\access\operator\ProcOperator.html), [QueryOperator](file:///D:\com\beetle\framework\persistence\access\operator\QueryOperator.html), [UpdateOperator](file:///D:\com\beetle\framework\persistence\access\operator\UpdateOperator.html)

下面就各组件进行详细介绍说明：

### 2.2.1单表操作器

在持久层编程中，针对数据库单个数据表的增、删、查、改的操作代码在整个持久层代码中占有相当比例。虽然这些CRUD代码十分简单，但是针对每张数据表重复编写雷同的代码显然是一件乏味的事情，另外，如果数据表的字段发现变动，那么这意味着CRUD代码将需要重新修改，增加了代码维护的难度。

为了简化编程，提高工作效率，我们可以针对单数据表的CRUD操作封装成TableOperator通用单表操作器（类）。

**com.beetle.framework.persistence.access.operator   
类 TableOperator<T>**

java.lang.Object 4638364261348736729914.gif**com.beetle.framework.persistence.access.operator.TableOperator<T>**

|  |
| --- |
| **构造方法摘要** |
| [**TableOperator**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.lang.String dataSourceName, java.lang.String tableName, java.lang.Class<[T](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)> valueObjectClass)            TableOperator（默认为此表的主键为非自动增量） |
| [**TableOperator**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.lang.String dataSourceName, java.lang.String tableName, java.lang.Class<[T](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)> valueObjectClass, boolean autoKeyFlag)            TableOperator |
| [**TableOperator**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.lang.String dataSourceName, java.lang.String tableName, java.lang.Class<[T](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)> valueObjectClass, java.lang.String autoKeyFiledName)            TableOperator |

|  |  |
| --- | --- |
| **方法摘要** | |
| java.util.List<[T](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)> | [**autoAllFill**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)([RsDataSet](file:///D:\com\beetle\framework\persistence\access\operator\RsDataSet.html) rs)            自动填充所有的结果 |
| [T](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html) | [**autoRowFill**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)([RsDataSet](file:///D:\com\beetle\framework\persistence\access\operator\RsDataSet.html) rs)            自动填充一行（将sql返回的字段值与其对应的值对象自动匹配起来） |
| int[] | [**deleteBatchByPrimaryKey**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.util.List pks)            根据主键批量删除 |
| int | [**deleteByPrimaryKey**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.lang.Object pk)            根据主键删除记录 |
| java.lang.String | [**generateFieldsString**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)()            生成此表字段名字符串 |
| java.lang.String | [**getPrimaryKeyFieldName**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)()            获取主键名称 |
| java.lang.String | [**getTableName**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)()            获取表名 |
| int | [**insert**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)([T](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html) valueObject)            插入一条记录 |
| int[] | [**insertBatch**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.util.List<[T](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)> valueObjectList)            批量插入记录 |
| boolean | [**isAutoGenerateKey**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)()            此表是否为自动生成id |
| [T](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html) | [**selectByPrimaryKey**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.lang.Object pk)            根据主键查找记录 |
| java.util.List<[T](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)> | [**selectByWhereCondition**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.lang.String whereStr, java.lang.Object[] values)            根据条件查找 |
| [MasterDetailDTO](file:///D:\com\beetle\framework\resource\define\MasterDetailDTO.html) | [**selectDetailTable**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.lang.Object masterTablePK, java.lang.String detailTableName, java.lang.Class<?> detailTableVOClass)            查询明细表记录 |
| [MasterDetailDTO](file:///D:\com\beetle\framework\resource\define\MasterDetailDTO.html) | [**selectDetailTableByWhereCondition**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.lang.Object masterTablePK, java.lang.String detailTableName, java.lang.Class<?> detailTableVOClass, java.lang.String whereStrForDetailTable, java.lang.Object[] sqlvalues)            按条件查询明细表的记录 |
| int | [**update**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.util.Map<java.lang.String,java.lang.Object> fieldValues) |
| int | [**update**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)([T](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html) valueObject)            更新记录 |
| int[] | [**updateBatch**](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)(java.util.List<[T](file:///D:\com\beetle\framework\persistence\access\operator\TableOperator.html)> valueObjectList)            批量更新记录 |

注意的是：单表操作器只适用于表的主键是单个字段的数据表，对组合主键的数据表不支持。

以Oracle9i数据库scott用户的EMP雇员表和DEPT表为示例

scott用户数据表E-R关系图如下：

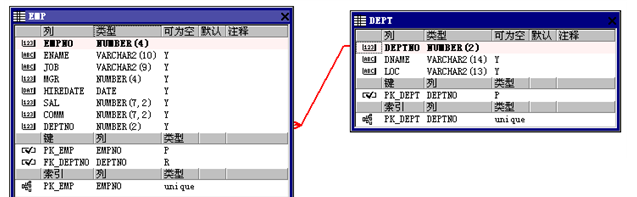


图2-3scott用户数据表E-R关系图

两个表的数据内容如下：

SQL> select \* from emp;

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

----- ---------- --------- ----- ----------- --------- --------- ------

7369 SMITH CLERK 7902 1980-12-17 800.00 20

7499 ALLEN SALESMAN 7698 1981-2-20 1600.00 300.00 30

7521 WARD SALESMAN 7698 1981-2-22 1250.00 500.00 30

7566 JONES MANAGER 7839 1981-4-2 2975.00 20

7654 MARTIN SALESMAN 7698 1981-9-28 1250.00 1400.00 30

7698 BLAKE MANAGER 7839 1981-5-1 2850.00 30

7782 CLARK MANAGER 7839 1981-6-9 2450.00 10

7788 SCOTT ANALYST 7566 1987-4-19 3000.00 20

7839 KING PRESIDENT 1981-11-17 5000.00 10

7844 TURNER SALESMAN 7698 1981-9-8 1500.00 0.00 30

7876 ADAMS CLERK 7788 1987-5-23 1100.00 20

7900 JAMES CLERK 7698 1981-12-3 950.00 30

7902 FORD ANALYST 7566 1981-12-3 3000.00 20

7934 MILLER CLERK 7782 1982-1-23 1300.00 10

14 rows selected

SQL> select \* from dept;

DEPTNO DNAME LOC

------ -------------- -------------

10 ACCOUNTING NEW YORK

20 RESEARCH DALLAS

30 SALES CHICAGO

40 OPERATIONS BOSTON

示例代码：

1. ，每个表建立一个相应属性（字段）的java值对象：

package demo.valueobject;

import java.math.BigDecimal;

public class DEPT extends VOBase {

private static final long serialVersionUID = 1L;

private BigDecimal DEPTNO;

private String DNAME;

private String LOC;

public DEPT() {

}

public BigDecimal getDEPTNO() {

return this.DEPTNO;

}

public String getDNAME() {

return this.DNAME;

}

public String getLOC() {

return this.LOC;

}

public void setDEPTNO(BigDecimal DEPTNO) {

this.DEPTNO = DEPTNO;

}

public void setDNAME(String DNAME) {

this.DNAME = DNAME;

}

public void setLOC(String LOC) {

this.LOC = LOC;

}

}

package demo.valueobject;

import java.math.BigDecimal;

import java.sql.Timestamp;

public class EMP extends VOBase {

private static final long serialVersionUID = 1L;

private BigDecimal DEPTNO;

private BigDecimal COMM;

private Timestamp HIREDATE;

private BigDecimal MGR;

private BigDecimal SAL;

private String JOB;

private String ENAME;

private BigDecimal EMPNO;

public EMP() {

}

public BigDecimal getDEPTNO() {

return this.DEPTNO;

}

public BigDecimal getCOMM() {

return this.COMM;

}

public Timestamp getHIREDATE() {

return this.HIREDATE;

}

public BigDecimal getMGR() {

return this.MGR;

}

public BigDecimal getSAL() {

return this.SAL;

}

public String getJOB() {

return this.JOB;

}

public String getENAME() {

return this.ENAME;

}

public BigDecimal getEMPNO() {

return this.EMPNO;

}

public void setDEPTNO(BigDecimal DEPTNO) {

this.DEPTNO = DEPTNO;

}

public void setCOMM(BigDecimal COMM) {

this.COMM = COMM;

}

public void setHIREDATE(Timestamp HIREDATE) {

this.HIREDATE = HIREDATE;

}

public void setMGR(BigDecimal MGR) {

this.MGR = MGR;

}

public void setSAL(BigDecimal SAL) {

this.SAL = SAL;

}

public void setJOB(String JOB) {

this.JOB = JOB;

}

public void setENAME(String ENAME) {

this.ENAME = ENAME;

}

public void setEMPNO(BigDecimal EMPNO) {

this.EMPNO = EMPNO;

}

}

1. 利用TableOperator对表EMP进行存取操作：

package test;

import java.math.BigDecimal;

import java.util.List;

import com.beetle.framework.persistence.access.operator.TableOperator;

import demo.valueobject.EMP;

public class TestMe {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

TableOperator<EMP> empOptr = new TableOperator<EMP>(

"SYSDATASOURCE\_oracle", "EMP", EMP.class);

// 查找编号（empNo）为7499的员工信息

EMP emp = empOptr.selectByPrimaryKey(new BigDecimal(7499));

System.out.println(emp);// 打印查到的数据

// 把此员工的薪水修改为300.25

emp.setSAL(new BigDecimal(300.25));

empOptr.update(emp);

// 建立一个员工编号为9999，名为Henry的新员工

emp.setEMPNO(new BigDecimal(9999));

emp.setENAME("Henry");

empOptr.insert(emp);

// 删除编号为9999的新员工

empOptr.deleteByPrimaryKey(new Integer(9999));

// 查找薪水在2000元以上的员工

Object params[] = new Object[1];

params[0] = new Float(2000);

List<EMP> empList = empOptr.selectByWhereCondition("where sal>=?",

params);

System.out.println(empList);

empList.clear();

}

}

可见，单个表的常见增、删、查、改操作使用TableOperator操作时无需写任何SQL代码的，而且完全实现了数据表与Java值对象的自动的数据装配的。另外TableOperator还可以实现更高级的功能，例如：批量插入、批量更新、局部更新等，详细可参考其上面的API。TableOperator同样支持主-从表（1：n）的操作，例如，我们通过DEPT来获取部门编号为“10”的所有月薪大于“2000”的员工的代码如下：

TableOperator<DEPT> deptOptr = new TableOperator<DEPT>(

"SYSDATASOURCE\_oracle", "DEPT", DEPT.class);

MasterDetailDTO mdt = deptOptr.selectDetailTableByWhereCondition(

new BigDecimal(10), "EMP", EMP.class, "where sal>=?",

new Object[] { new Float(2000) });

System.out.println((DEPT) mdt.getMaster());

List<?> emps = mdt.getDetail();

for (Object emp : emps) {

System.out.println((EMP) emp);

}

emps.clear();

执行结果：

DEPT [DEPTNO=10, DNAME=ACCOUNTING, LOC=NEW YORK]

EMP [DEPTNO=10, COMM=null, HIREDATE=1981-06-09 00:00:00.0, MGR=7839, SAL=2450, JOB=MANAGER, ENAME=CLARK, EMPNO=7782]

EMP [DEPTNO=10, COMM=null, HIREDATE=1981-11-17 00:00:00.0, MGR=null, SAL=5000, JOB=PRESIDENT, ENAME=KING, EMPNO=7839]

81278576189468.jpg实际上，TableOperator实现思路与当前大多数所谓O/R框架底层核心技术的基本原理是一致的。涉及到结果集与值对象的自动装配问题，当前解决方法通常有两种方式：

（1），建立一个配置文件把值对象属性与表结构的字段属性一一对应起来。例如：iBatis和Hibernate等框架就是采取此方法，好处是以不变应万变，不受表结构与值对象定义格式的任何约束。其缺点是，需要维护多一份配置文件，将问题复杂化。另外，对于我们仅仅的单表操作来说，显得过“重”。

（2），采取值对象属性名称与表结构字段名称必须相同（可不区分大小写）约定关系。其好处是简单明了，不依赖于任何的约束关系。缺点是不能做到值对象与表字段的任意名称定义。我们在这里采取约定方法。

### 2.2.2查询操作器

查询操作器QueryOperator的使用很简单，其API为：

com.beetle.framework.persistence.access.operator   
类 QueryOperator

java.lang.Object

继承者 [com.beetle.framework.persistence.access.operator.BaseOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html)

继承者 **com.beetle.framework.persistence.access.operator.QueryOperator**

**直接已知子类：**

[CompositeQueryOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\composite\CompositeQueryOperator.html)

public class **QueryOperator**

extends [BaseOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html)

|  |
| --- |
| **字段摘要** |

|  |
| --- |
| **从类 com.beetle.framework.persistence.access.operator.**[**BaseOperator**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html)**继承的字段** |
| [logger](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#logger) |

|  |  |
| --- | --- |
| **构造方法摘要** | |
| [**QueryOperator**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\QueryOperator.html#QueryOperator())() |  |

|  |  |
| --- | --- |
| **方法摘要** | |
| protected  void | [**accessImp**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\QueryOperator.html#accessImp())() |
| [CompositeDTO](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\resource\define\CompositeDTO.html) | [**getResult**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\QueryOperator.html#getResult(java.lang.Class...))(java.lang.Class<?>... dtoClass)            针对单个数据dto对象无法包含所有的结果数据，而又不想重新定向其对应的dto对象的情况。 |
| java.util.List<java.util.Map<java.lang.String,java.lang.Object>> | [**getResultList**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\QueryOperator.html#getResultList())()            获取查询结果集，每个记录对应一个key-value的map数据对象，key为查询语句的字段名称，value为数据 |
| |  | | --- | | <T> java.util.List<T> | | [**getResultList**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\QueryOperator.html#getResultList(java.lang.Class))(java.lang.Class<T> dtoClass)            根据输入的数据对象定义类，自动装配，返回其结果数据列表 |
| java.util.List<java.util.Map<java.lang.String,java.lang.Object>> | [**getSqlResultSet**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\QueryOperator.html#getSqlResultSet())()            返回sql语句返回的结果列表 |
| boolean | [**resultSetAvailable**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\QueryOperator.html#resultSetAvailable())()            查询返回的结果集是否可用 true--结果可用，拥有记录 false--结果不可用，没有记录集 |
| void | [**setMaxRow**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\QueryOperator.html#setMaxRow(int))(int maxRow)            设置结果集返回的最大条数。 |

|  |
| --- |
| **从类 com.beetle.framework.persistence.access.operator.**[**BaseOperator**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html)**继承的方法** |
| [access](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#access()), [addParameter](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#addParameter(java.lang.Object)), [addParameter](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#addParameter(com.beetle.framework.persistence.access.operator.SqlParameter)), [closePresentConnection](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#closePresentConnection()), [getPresentConnection](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#getPresentConnection()), [getSql](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#getSql()), [isAccessed](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#isAccessed()), [isPresentConnectionUsable](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#isPresentConnectionUsable()), [isUseOnlyConnectionFlag](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#isUseOnlyConnectionFlag()), [setDataSourceName](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#setDataSourceName(java.lang.String)),[setPresentConnection](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#setPresentConnection(java.sql.Connection)), [setSql](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#setSql(java.lang.String)), [setUseOnlyConnectionFlag](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#setUseOnlyConnectionFlag(boolean)) |

例如使用QueryOperator来查询前面示例EMP表中所有薪水>=2000元的雇员，其代码如下：

static List**<**EMP**>** findEmpBySal\_Ver1**(**Float sal**)** **{**

List**<**EMP**>** empList **=** **new** ArrayList**<**EMP**>();**

QueryOperator query **=** **new** QueryOperator**();**

query**.**setDataSourceName**(**"SYSDATASOURCE\_oracle"**);**// 设置数据源

// 设置sql查询语句

query**.**setSql**(**"select empno,ename,job,mgr,hiredate,sal,comm,deptno from emp where sal>=?"**);**

// 设置sql参数

query**.**addParameter**(new** SqlParameter**(**SqlType**.**FLOAT**,** sal**));**[1]

query**.**access**();**// 执行

**if** **(**query**.**resultSetAvailable**())** **{** // 检查是否存在结果集

// 利用结果集处理对象解析查询结果数据

RsDataSet rs **=** **new** RsDataSet**(**query**.**getSqlResultSet**());**

**try** **{**

**for** **(**int i **=** 0**;** i **<** rs**.**rowCount**;** i**++)** **{**

EMP emp **=** **new** EMP**();**

// 利用getFieldValueAsXXX方法取值

emp**.**setCOMM**(**rs**.**getFieldValueAsBigDecimal**(**"comm"**));**

emp**.**setDEPTNO**(**rs**.**getFieldValueAsBigDecimal**(**"deptno"**));**

emp**.**setEMPNO**(**rs**.**getFieldValueAsBigDecimal**(**"empno"**));**

emp**.**setENAME**(**rs**.**getFieldValueAsString**(**"ename"**));**

emp**.**setHIREDATE**(**rs**.**getFieldValueAsTimestamp**(**"hiredate"**));**

emp**.**setJOB**(**rs**.**getFieldValueAsString**(**"job"**));**

emp**.**setMGR**(**rs**.**getFieldValueAsBigDecimal**(**"mgr"**));**

emp**.**setSAL**(**rs**.**getFieldValueAsBigDecimal**(**"sal"**));**

empList**.**add**(**emp**);**

rs**.**next**();**// 移动记录位置指向下一条

**}**

**}** **finally** **{**

rs**.**clearAll**();**// 清空结果集（本方法为可选，在释放rs对象会自动清空）

**}**

**}**

**return** empList**;**

**}**

[1]可以直接简化成

query.addParameter(sal);

即无需指定参数的数据类型，框架会自动给以识别。当框架无法识别（执行报错或不正确）时，再按标准指定参数类型的写法。

另外，对上面代码黄色加亮部分是对结果集的基本操作，这种方式比较灵活但也比较低级，虽然EMP对象属性逐个设值的方式虽然很清晰，但是如果字段过多的话，显得编程繁琐。对此，QueryOperator对结果集的处理（数据的装配）作了进一步优化操作，可以直接简化为：

static List**<**EMP**>** findEmpBySal\_Ver2**(**Float sal**)** **{**

QueryOperator query **=** **new** QueryOperator**();**

query**.**setDataSourceName**(**"SYSDATASOURCE\_oracle"**);**

query**.**setSql**(**"select empno,ename,job,mgr,hiredate,sal,comm,deptno from emp where sal>=?"**);**

query**.**addParameter**(**sal**);**

query**.**access**();**

**return** query**.**getResultList**(**EMP**.**class**);**

**}**

有时候，你不希望使用一个Java的值对象（如此例的EMP对象）去装配，你希望直接返回一个原始的查询数据，此时，你可以：

**static** List<Map<String, Object>> findEmpBySal\_Ver3(Float sal) {

QueryOperator query = **new** QueryOperator();

query.setDataSourceName("SYSDATASOURCE\_oracle");

query.setSql("select empno,ename,job,mgr,hiredate,sal,comm,deptno from emp where sal>=?");

query.addParameter(sal);

query.access();// 执行

**return** query.getResultList();

}

它返回是一个元素为Map的列表，这个Map里面，key为查询语句的字段名称，value为数据。

为了方面查询结果集的处理，我们采取的策略是每个查询结果集建立一个对应的Java值对象去装配数据，这样产生一个问题就是，如果查询语句的的返回的数据形式较多的话（特别是多表关联查询的情况下），我们有可能建立众多的值对象，这显然有点。。。不够优雅O(∩\_∩)O~，QueryOperator提供了：

|  |  |
| --- | --- |
| [CompositeDTO](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\resource\define\CompositeDTO.html) | [**getResult**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\QueryOperator.html#getResult(java.lang.Class...))(java.lang.Class<?>... dtoClass)            针对单个数据dto对象无法包含所有的结果数据，而又不想重新定向其对应的dto对象的情况。 |

方法来解决这个问题，其思路就是，就是重用现有的已定义值对象，利益现有的值对象去装配数据，例如：我们现在DEPT和EMP做关联查询，返回某个部门下所有的员工及此部门的信息。其查询语句为：

select dept.deptno,

dept.dname,

dept.loc,

emp.empno,

emp.ename,

emp.job,

emp.mgr,

emp.hiredate,

emp.sal,

emp.comm

from dept, emp

where dept.deptno = emp.deptno and dept.deptno=?

查询结果包含了DEPT和EMP值对象所有属性，如果我们再建立一个X值对象去装配就显得啰嗦了，此时我们可以采取

|  |  |
| --- | --- |
| [CompositeDTO](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\resource\define\CompositeDTO.html) | [**getResult**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\QueryOperator.html#getResult(java.lang.Class...))(java.lang.Class<?>... dtoClass)            针对单个数据dto对象无法包含所有的结果数据，而又不想重新定向其对应的dto对象的情况。 |

方法来处理，上面例子的代码如下：

**static** **void** findDeptEmps(Long deptNo) {

QueryOperator query = **new** QueryOperator();

query.setDataSourceName("SYSDATASOURCE\_oracle");

query.setSql("select dept.deptno,dept.dname,dept.loc,emp.empno,emp.ename,emp.job,emp.mgr,emp.hiredate,emp.sal,emp.comm from dept,emp where dept.deptno = emp.deptno and dept.deptno=?");

query.addParameter(deptNo);

query.access();

CompositeDTO ct = query.getResult(DEPT.**class**, EMP.**class**);

List<DEPT> deptList = ct.getObjList(DEPT.**class**);

System.*out*.println(deptList);

List<EMP> empList = ct.getObjList(EMP.**class**);

**for** (EMP emp : empList) {

System.*out*.println(emp);

}

deptList.clear();

empList.clear();

}

执行：

*findDeptEmps*(10L);

结果：

[DEPT [DEPTNO=10, DNAME=ACCOUNTING, LOC=NEW YORK], DEPT [DEPTNO=10, DNAME=ACCOUNTING, LOC=NEW YORK], DEPT [DEPTNO=10, DNAME=ACCOUNTING, LOC=NEW YORK]]

EMP [DEPTNO=10, COMM=null, HIREDATE=1981-06-09 00:00:00.0, MGR=7839, SAL=2450, JOB=MANAGER, ENAME=CLARK, EMPNO=7782]

EMP [DEPTNO=10, COMM=null, HIREDATE=1981-11-17 00:00:00.0, MGR=null, SAL=5000, JOB=PRESIDENT, ENAME=KING, EMPNO=7839]

EMP [DEPTNO=10, COMM=null, HIREDATE=1982-01-23 00:00:00.0, MGR=7782, SAL=1300, JOB=CLERK, ENAME=MILLER, EMPNO=7934]

可见，返回是一个组合的对象，然后从这个对象里面通过getObjList方法再获取各个装配好的值对象。值得注意的是，我们采取这个方式算是取巧的方法，如果值对象中有多出的属性，我们是不会装配的，这些多出的属性在获取时，应该会保持其值对象的默认值，这些不是数据库返回的，所以使用时，开发人员脑中一定要有这样的认识，要不很容易导致误会，从而导致数据不一致的。另外，每个返回的Lis的它们的记录的顺序是一致的。

81278576189468.jpg采取QueryOperator等数据存取操作器来与数据库交互，开发人员无须关心数据库连接的创建和释放，这些都是框架后台根据情况自动管理的。另外，从上面代码可知，一个简单的QueryOperator查询操作对象，就可以代替原来很多底层的JDBC接口操作，消除了操作数据库一切可能发生的危险操作，同时，又让代码条理清晰，容易理解。使用了QueryOperator等数据存取操作框架的程序代码，你会发现代码编写流程、编写方式几乎都是“千篇一律”的，这对一个团体开发，以及以后代码的维护是极其重要的。

### 2.2.3更新操作器

UpdateOperator提供功能方法基本与QueryOperator一致。更新操作器与查询操作器相对应，更新操作器主要负责没有结果集返回的sql语句访问数据库的情形。其支持单句更新和批量更新两种方式。

com.beetle.framework.persistence.access.operator   
类 UpdateOperator

java.lang.Object

继承者 [com.beetle.framework.persistence.access.operator.BaseOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html)

继承者 **com.beetle.framework.persistence.access.operator.UpdateOperator**

public final class **UpdateOperator**

extends [BaseOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html)

|  |
| --- |
| **字段摘要** |

|  |
| --- |
| **从类 com.beetle.framework.persistence.access.operator.**[**BaseOperator**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html)**继承的字段** |
| [logger](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#logger) |

|  |  |
| --- | --- |
| **构造方法摘要** | |
| [**UpdateOperator**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\UpdateOperator.html#UpdateOperator())() |  |

|  |  |
| --- | --- |
| **方法摘要** | |
| protected  void | [**accessImp**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\UpdateOperator.html#accessImp())() |
| void | [**addBatchParameter**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\UpdateOperator.html#addBatchParameter(com.beetle.framework.persistence.access.operator.SqlParameterSet))([SqlParameterSet](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\SqlParameterSet.html) sqlParameterSet)            添加一行sql语句的参数（批处理时候使用） |
| int[] | [**getBatchEffectCounts**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\UpdateOperator.html#getBatchEffectCounts())()            an array of update counts containing one element for each command in the batch. |
| int | [**getEffectCounts**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\UpdateOperator.html#getEffectCounts())()            获取执行后影响的记录条数,不支持批量更新语句 对于 INSERT、UPDATE 或 DELETE 语句，返回行数; 对于什么都不返回的 SQL 语句，返回 0 |

|  |
| --- |
| **从类 com.beetle.framework.persistence.access.operator.**[**BaseOperator**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html)**继承的方法** |
| [access](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#access()), [addParameter](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#addParameter(java.lang.Object)), [addParameter](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#addParameter(com.beetle.framework.persistence.access.operator.SqlParameter)), [closePresentConnection](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#closePresentConnection()), [getPresentConnection](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#getPresentConnection()), [getSql](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#getSql()), [isAccessed](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#isAccessed()), [isPresentConnectionUsable](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#isPresentConnectionUsable()), [isUseOnlyConnectionFlag](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#isUseOnlyConnectionFlag()), [setDataSourceName](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#setDataSourceName(java.lang.String)),[setPresentConnection](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#setPresentConnection(java.sql.Connection)), [setSql](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#setSql(java.lang.String)), [setUseOnlyConnectionFlag](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html#setUseOnlyConnectionFlag(boolean)) |

下面分别给以示例：

单句更新

把EMP表中雇员编号为7499的薪水修改为350元，代码如下：

**public** **static** **int** updateSal(Float sal, Integer empNo) {

UpdateOperator update = **new** UpdateOperator();

update.setDataSourceName("SYSDATASOURCE\_oracle");

update.setSql("update emp set sal=? where empno=?");

update.addParameter(sal);

update.addParameter(empNo);

update.access();

**return** update.getEffectCounts();

}//调用

updateSal(new Float(350),new Integer(7499));

批量更新

修改多个雇员的薪水（7499🡪350 ;7521🡪1450 ;7902🡪5000），为了提高执行效率，此时可以使用批量更新，代码如下：

**static** **int**[] updateSalBatch(Float sals[], Integer empNos[]) {

UpdateOperator update = **new** UpdateOperator();

update.setDataSourceName("SYSDATASOURCE\_oracle");

update.setSql("update emp set sal=? where empno=?");

**for** (**int** i = 0; i < empNos.length; i++) {

SqlParameterSet sps = **new** SqlParameterSet();

sps.addParameter(**new** SqlParameter(SqlType.*FLOAT*, sals[i]));

sps.addParameter(**new** SqlParameter(SqlType.*INTEGER*, empNos[i]));

update.addBatchParameter(sps);

}

update.access();

**return** update.getBatchEffectCounts();

}

//调用

Integer empNos[] = new Integer[3];

Float sals[] = new Float[3];

empNos[0] = new Integer(7499);

sals[0] = new Float(350);

empNos[1] = new Integer(7521);

sals[1] = new Float(1450);

empNos[2] = new Integer(7902);

sals[2] = new Float(5000);

updateSalBatch(sals, empNos);

### 2.2.4存储过程操作器

大多数商业的数据库，如Oracle、SqlServer、Sysbase、DB2等等，都提供存储过程支持。其实，在J2EE开发中，关于存储过程的使用问题一直存在很大的争论。故此，在我们使用存储过程操作器的之前，我们还是有必要讨论一下存储过程使用这个问题。

J2EE开发往往不推荐使用存储过程，究其原因，有以下几点：

存储过程是不可以跨数据库移植的。对存储过程的支持，各个数据库之间没有一个统一的标准，使用的语句也差别很大（如：oracle是pl/sql；Sqlserver是TansactSQL），很难互相迁移。

在一个J2EE项目中，存储过程的引入容易导致系统变得越来越复杂，减低了系统的可维护性。

存储过程，基本上都是过程语言，不是面向对象的。重用性比较差。

如果把业务逻辑都在存储过程里面处理，会对数据库要求很高，容易出现性能问题。而且，使得系统难以扩展。

存储过程的引入对Java程序员要求提高，必须掌握某种数据库的专有语言。

虽然存储过程会带来以上这些问题，那我们是不是一概而论，在J2EE项目中拒绝任何存储过程的使用呢？笔者认为，在J2EE中全面否定存储过程是错误的。在某些情况下，存储过程对我们解决问题带来明显的好处，甚至是唯一的方法。存储过程的好处有：

对于处理持久性逻辑，存储过程更灵活，可以多个数据表的更新。存储过程的运行速度比同价的Java业务对象要来得快速。

对于那些需要多个sql语句交互才能完成的持久性数据逻辑，存储过程可以很完美地合并它们，提高效率。

存储过程可以访问数据库独有的特性，这在很多时候，普通的sql语句是做不到的。这在某些情况下，将会明显提高应用的效率。

存储过程有时候是整合已有旧系统的必要工具。

如果我们能很清晰地区分开持久性逻辑与业务性逻辑，使用存储过程将不会破坏我们J2EE的体系结构。记住，尽量不要让存储过程实现业务逻辑，这应该是Java业务对象的责任。

BJAF数据存取框架对存储过程使用提供了支持，但是这种支持不是任意的，而是根据我们对存储过程的开发方面的设计策略进行了界定，因为我们相信：存储过程也是一种语言，如果不加以编程规范化，其编写方式将是五花八门，其结果将会使得项目变得难以维护。从以往经验来看，一个项目如果要引入存储过程，我们必须以一种简单明了的方式将其编写规范化，否则的话，存储过程代码的维护将是一个恶梦。

BJAF框架认为储存过程类似一个Command，通过调用一个存储过程的名称，让数据库在服务端运行，并返回相关结果。因而，框架参考Command设计模式，将存储过程的调用模式化，达到最终便于开发和维护的目的。故此，利用BJAF进行存储过程开发需要遵循以下规定：

|  |  |
| --- | --- |
| **约定** | **说明** |
| （1）存储过程必须有两个输出参数：  **returnFlag**――返回标识；大于或等于0（>=0）为存储过程按正常逻辑运行成功，其中等于1（=1）表示存储过程有结果集返回；小于0（<0）为失败。  **returnMsg**――返回说明。主要是对返回标识的补充说明。 | 我们编写存储过程让其在数据库服务器中运行，我们在客户端是很需要知道存储过程的运行结果是否按照我们思路完成的，因为我们需要根据这样的结果来指导我们下一步的操作。引入输出参数returnFlag，我们就可以根据这个标识来判别存储过程的执行情况了。 |
| （2）存储过程除了上面两个输出参数外，不能再要其他输出参数。（Oracle除外，因为我们需要一个输出游标来返回结果集） | 这一点很重要，如果我们不规定这一点。程序员就会根据自己的需要，利用输出参数来任意返回结果，对维护不利。如果需要输出结果，我们可以统一通过结果集这种方式返回。 |
| （3）一个存储过程每次只能返回一个结果集 | 从存储过程返回结果集，是很常见的需求。存储过程功能强大，可以一次返回多个结果集。如果这样的话，这对我们编程很不利，难以模式化和维护。而且，这一点也是保证和SQL查询语言返回习惯一致。 |

由于存储过程与各数据库特性的依赖性，BJAF框架实现了两种类型的存储过程操作器，分别是：

CommonProcOperator普通操作器适合不依赖于显式游标返回数据结果集的数据库，如：MySql、DB2、Sybase等；SqlServerProcOperator为针对sqlserver实现；OracleProcOperator针对Oracle数据库显式游标返回数据结果集的专门实现。

com.beetle.framework.persistence.access.operator   
类 ProcOperator

java.lang.Object

继承者 [com.beetle.framework.persistence.access.operator.BaseOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html)

继承者 **com.beetle.framework.persistence.access.operator.ProcOperator**

**直接已知子类：**

[CommonProcOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\CommonProcOperator.html), [OracleProcOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\OracleProcOperator.html), [SqlServerProcOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\SqlServerProcOperator.html)

public abstract class **ProcOperator**

extends [BaseOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html)

下面以操作Oracle数据库存储过程进行示例说明：

前面查找EMP表中所有薪水>=2000元的雇员的查询例子，我们将它改成用存储过程方式

（1），编写Oracle数据库findEmpBySal存储过程：

--建立一个包，并定义一个参考游标

create or replace package beetle as type aCur is ref cursor; end;

/

--编写具体存储过程

create or replace procedure findEmpBySal(

i\_sal number,--输入参数

o\_flag out number,--存储过程返回标记

o\_msg out varchar2,--返回补充说明

o\_cur in out beetle.aCur --记录集输出游标

)

is

begin

o\_flag:=1;

o\_msg:='ok';

open o\_cur for select \* from emp where sal>=i\_sal;

exception

when others then

begin

o\_flag:=sqlcode;

o\_msg:=sqlerrm;

end;

end;

/

（2），编写调用此存储过程代码：

**static** **void** testProc() {

ProcOperator proc = **new** OracleProcOperator();

proc.setDataSourceName("SYSDATASOURCE\_oracle");// 设置数据源

proc.setSql("findEmpBySal");// 设置存储过程名称

// 设置输入参数

proc.addParameter(**new** SqlParameter(SqlType.*FLOAT*, **new** Float(2000)));

proc.access();// 执行

**int** f = proc.getReturnFlag();// 获取执行状态标记

**if** (f == ProcOperator.*PROC\_HAVE\_RESULT*) {// 是否存在结果集

RsDataSet rs = **new** RsDataSet(proc.getSqlResultSet());

**try** {

**for** (**int** i = 0; i < rs.rowCount; i++) {// 打印结果

EMP emp = **new** EMP();

rs.autoFillRow(emp);

System.*out*.println(emp);

rs.next();

}

} **finally** {

rs.clearAll();

}

} **else** { // 其它情况（成功但没有结果集返回或执行异常）

System.*out*.println(proc.getReturnFlag());

System.*out*.println(proc.getReturnMsg());

}

}

可见代码十分简单，没有任何JDBC底层调用存储过程的API，在存储过程结果集的时候和QueryOperator保持一致的方式，代码的风格也一样。

### 2.2.5序列操作器

我们在进行关系数据库设计的时候，对于每一张表我们基本都要指定一个主键。而这个主键是一个唯一标识。主键常与外键构成参照完整性约束，保证数据的一致性，而这一切都归功于唯一标识。

常见唯一标识生成有以下基本方式：

数据库系统自己生成、维护。

几乎每一个数据库系统都支持唯一标识的生成和管理。例如，有的使用自动增长字段、有的使用序列等等，只是各自的方式不尽相同罢了。读者根据需要，自己参考各自的数据库系统就可以了，这里对此方式就不详细讨论了。这种方式的好处有：使用简单、安全，可以省略掉我们很多繁琐的工作。它的缺点就是：它没法满足我们所有的业务需求，特别是在主－从关系表的处理上面，我们更多的时候需要自己去维护这个唯一标识。

开发人员手工生成、维护。

开发人员手工生成和维护唯一标识，可以看作是数据库自动生成方式的一个补充。特别当唯一标识具备业务属性的时候，这个需求更加强烈。手工生成唯一标识的技术也是各种各样，为了便于系统的维护和迁移，BJAF框架对唯一标识生成的方式抽象出来，形成ISequence统一的编程接口。

com.beetle.framework.persistence.seq   
接口 ISequence

**所有已知实现类：**

[CommonSeqGenerator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\seq\imp\CommonSeqGenerator.html), [DB2SeqGenerator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\seq\imp\DB2SeqGenerator.html), [MySqlSeqGenerator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\seq\imp\MySqlSeqGenerator.html), [OracleSeqGenerator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\seq\imp\OracleSeqGenerator.html), [PostgreSqlSeqGenerator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\seq\imp\PostgreSqlSeqGenerator.html)

ISequence接口有5个实现。除了CommonSeqGenerator[1]外，其它3个都是分别针对Oracle、DB2、Mysql和PostgreSql三个自身支持Sequence数据库的实现，使用它们需要在它们各自系统建立Sequence对象；CommonSeqGenerator是通用实现，支持所有关系数据库，它需要在相应的数据库中建立一张名为“SEQUENCE”存放序列的数据表：

create table SEQUENCE (

NAME varchar(30) not null,

NEXTID DECIMAL(20,0),

primary key (NAME)

);

例如，使用CommonSeqGenerator实现来获取数据库序列号示例如下：

（1），往SEQUENCE表插入一条序列记录：

insert into sequence values('customerId',1000);

序列名为：customerId，初始化值：1000

（2），在其对应的DBConfig.xml数据源的扩展属性配置中，指定其具体实现类

<Extensions>

<SYSDATASOURCE\_DEFAULT>

<!-- 序列号实现类型 Common, Oracle, DB2, PostgreSql,MySql, Other; -->

<item name=**"seq-type"** value=**"Common"** />

</SYSDATASOURCE\_DEFAULT>

</Extensions>

（2），通过封装SeqOperator序列操作器来获取下一位序列号：

**static** **void** testSeq() {

**long** id = SeqOperator.*nextSequenceNum*("SYSDATASOURCE\_oracle",

"customerId");

System.*out*.println(id);

}

若是换成其它实现，只需修改配置文件seq-type参数即可，例如换成OracleSeqGenerator实现：

<Extensions>

<SYSDATASOURCE\_oracle>

<item name=**"seq-type"** value=**"Oracle"** />

<item name=**"pagination-imp"**

value=**"com.beetle.framework.persistence.pagination.imp.OraclePaginationImp"** />

</SYSDATASOURCE\_oracle>

</Extensions>

代码无效改动，同时在Oracle中建立一个序列对象：

create sequence customerId start with 1000;

[1]由于CommonSeqGenerator的获取种子方法在一个完整的事务内完成，为了避免嵌套事务的发生，建议专门建立一个"no-tx-datasource"数据源（或不参与全局事务的数据源）为此类服务。

### 2.2.6分页操作器

当我们为了完成某个业务逻辑，需要从数据库系统返回记录过多的结果集的时候，往往会引发多方面的问题，这些问题包括：性能优化设计策略、结果集的分页技术等等。对于开发人员来说，都是无法避免的。

先看看数据库查询大结果集可能带来的问题：

（1），内存占有过大。如果我们一次性把成千上万条记录保存在内存的话，内存势必会消耗过大，这使得JVM虚拟机能运用的内存空间减少，数据交换的速度减低，导致服务性能明显降低；而且，很容易引发内存不足不可恢复异常，最后导致整个服务瘫痪。

（2），网络传输阻塞。一般情况下，数据库服务都是独立一台机器，如果我们一次性从数据库返回大量数据的话，很容易瞬间导致网络阻塞，特别是当用户并发访问很大的时时候，情况更加明显。

（3），数据显示问题。如果一次性把成千上万条记录用一个页面或窗口呈献给查询用户，容易造成客户视觉疲劳，所以我们要考虑分页显示。

针对不同的问题，我们解决问题的方法也不尽相同。对待大结果集数据分页的处理上，我们一般的设计策略有以下3种方式：

（一），把结果集所有的数据一次性查询出来，放在应用服务器内存中，然后再分页处理。

这是一种很常见的思路，它的优点是：解藕了ResultSet操作类和数据库查询游标的关系，使得数据库连接得到有效的释放，提高了数据库并发访问的性能。同时，把数据导在本地，处理十分简单。由于数据都在内存中，翻页查询没有必要再从数据库请求数据，直接从内存获取就可以了，因而，响应更快。其缺点有：由于数据缓存在内存中，客户查询有可能看到的是过期数据；如果查询结果集数据量非常大的话，第一次查询遍历结果集会很耗时间，而且缓存这样大的数据也会造成内存消耗过大，使得JVM的效率明显降低。所以，这种方式对于查询结果在1000条以内的“小”结果集，而且数据完整性要求不高的情况下，可以采用，而且也十分简单有效。

（二），对查询结果集进行业务分析，针对查询客户的需要，把客户最为关心的数据首先呈献给客户，对于一些老数据，放入历史库，另外提供一个历史查询模块给客户使用。

这其实是把查询结果集按照客户的需要拆分开，避免了问题的主要矛盾，当然，前提是假设了用户实时关心的数据并不多。例如，对新闻数据查询，客户可能最关心的就是近3天的新闻信息，其它过期新闻可能1个月也不会查询一次。这个方式主要基于业务上设计策略，实现的技术也是甚为灵活。

（三），查询结果集所有的数据仍然留在数据库服务器端。每次翻页，我们都根据页面大小只从数据库服务器里面检索并返回块区数据，直到检索完所有的记录。

由于每次查询的记录数很少，网络传输数据量不大，不会造成阻塞。而且，数据都是由数据库统一提供，不会存在数据过期问题。应该说，这种方式是解决数据分页的理想方式。

实现方式（三）的技术常见有：

直接使用JDBC底层的ResultSet接口，利用游标来处理。

ResultSet本质是在数据库服务器上建立游标，然后通过行位置定位数据记录。这种方法，当客户第一请求数据时，就执行查询语句，获得ResultSet对象，保存在会话中，以后分页获取数据时都是通过ResultSet对象来定位和获取指定位置的记录。最后，当客户不再进行分页查询的时候，就关闭会话，释放数据库连接和ResultSet等等数据访问资源。

优点：减少sql语句查询的次数，利用标准API实现，便于系统移植。

缺点：每一个用户查询都要占用一个数据库连接和结果集游标。当并发用户量大的时候，会浪费数据库连接资源，使数据库响应变慢，性能降低；另外，从系统设计的角度考虑，开发人员需要操作ResultSet和维护其状态，使得无法从结果集的处理独立出来，对系统框架的抽象设计很是不利；而且，ResultSet在数据定位上的操作也很是笨拙。

所以这方法基本上不值得提倡。

利用各种数据库服务器自己提供的对查询结果集的可定位行范围的接口技术来实现。

客户发出请求，数据库就根据查询请求的行范围参数，检索出所需的记录返回给客户端。客户获取记录集后就释放相关的数据访问资源，从而避免了对并发访问的影响。

优点：技术简单，直接利用了数据库产品特性实现，java开发人员无需关心其具体的实现原理；充分利用了数据库的连接资源；系统查询响应速度快；解藕了ResultSet与数据结果集的关系，便于系统框架的抽象设计。

缺点：需要涉及专有的数据库技术，对java开发人员要求高。每翻页一次，就要查询数据库一次；通用性不强，因为每种数据库的行范围定位技术都不一样，需要重新抽象。

虽然对开发人员数据库知识要求高，但这种方法往往是最为有效的解决方法。我们只要根据经验，把各种数据库的行范围技术总结一次，我们就重复利用，数据分页的问题因而也就变得十分简单了。BJAF数据存取框架总结来各个数据库系统分页特有技术，封装成Pagination查询分页组件，有效地解决来持久层数据库查询分页编程难题。其API见下：

com.beetle.framework.persistence.pagination   
接口 IPagination

**所有已知实现类：**

[DB2PaginationImp](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\pagination\imp\DB2PaginationImp.html), [FireBirdPaginationImp](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\pagination\imp\FireBirdPaginationImp.html), [MysqlPaginationImp](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\pagination\imp\MysqlPaginationImp.html), [OraclePaginationImp](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\pagination\imp\OraclePaginationImp.html), [PostgreSqlPaginationImp](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\pagination\imp\PostgreSqlPaginationImp.html), [SqlServerPaginationImp](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\pagination\imp\SqlServerPaginationImp.html),[SybasePaginationImp](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\pagination\imp\SybasePaginationImp.html)

假设前面查询EMP表薪水>2000的雇员记录数超过3000条，现在采取每页20返回条记录进行分页查询，其实现如下：

（1），在DBConfig.xml文件其数据源扩展属性中指定其具体实现类：

<Extensions>

<SYSDATASOURCE\_DEFAULT>

<!-- 序列号实现类型 Common, Oracle, DB2, PostgreSql,MySql, Other; -->

<item name=**"seq-type"** value=**"Oracle"** />

<!-- 查询分页实现类

1-mysql[com.beetle.framework.persistence.pagination.imp.MysqlPaginationImp]

2-sysbase[com.beetle.framework.persistence.pagination.imp.SybasePaginationImp]

3-sqlserver[com.beetle.framework.persistence.pagination.imp.SqlServerPaginationImp]

4-oracle[com.beetle.framework.persistence.pagination.imp.OraclePaginationImp]

5-db2[com.beetle.framework.persistence.pagination.imp.DB2PaginationImp]

6-postgresql[com.beetle.framework.persistence.pagination.imp.PostgreSqlPaginationImp]

7-firebird[com.beetle.framework.persistence.pagination.imp.FireBirdPaginationImp] -->

<item name=**"pagination-imp"**

value=**"com.beetle.framework.persistence.pagination.imp.OraclePaginationImp"** />

</SYSDATASOURCE\_DEFAULT>

<SYSDATASOURCE\_SQLSERVER>

<item name=**"seq-type"** value=**"Common"** />

<item name=**"pagination-imp"**

value=**"com.beetle.framework.persistence.pagination.imp.MysqlPaginationImp"** />

</SYSDATASOURCE\_SQLSERVER>

<SYSDATASOURCE\_postgreSql>

<item name=**"seq-type"** value=**"PostgreSql"** />

<item name=**"pagination-imp"**

value=**"com.beetle.framework.persistence.pagination.imp.PostgreSqlPaginationImp"** />

</SYSDATASOURCE\_postgreSql>

</Extensions>

（2），使用PaginationOperator分页操作器

**static** **void** testPagination() {

// 建立一个分页参数对象

PageParameter param = **new** PageParameter();

param.setDataSourceName("SYSDATASOURCE\_oracle"); // 设置数据源

param.setUserSql("select empno,ename,job,mgr,hiredate,sal,comm,deptno from emp where sal>=?"); // 查询语句

param.addParameter(**new** SqlParameter(SqlType.*FLOAT*, **new** Float(2000)));// 设置sql参数

param.setPageNumber(1);// 设置查询页号码（现返回第1页数据）

param.setPageSize(20);// 设置每页返回数量大小（条数）

PageResult result = PaginationOperator.*access*(param);// 执行并获取结果

// 打印此结果信息对象

System.*out*.println(result.getCurPageNumber());// 当前页号

System.*out*.println(result.getCurPageSize());// 当前页数量大小

System.*out*.println(result.getCurPos());// 当前记录集位置

System.*out*.println(result.getFirstPageNumber());// 首页号码

System.*out*.println(result.getLastPageNumber());// 最后页号码

System.*out*.println(result.getNextPageNumber());// 下一页号码

System.*out*.println(result.getPrePageNumber());// 前一页号码

System.*out*.println(result.getPageAmount());// 总页数

System.*out*.println(result.getRecordAmount());// 总记录数

// 利用RsDataSet解析查询记录集

RsDataSet rs = **new** RsDataSet(result.getSqlResultSet());// 记录数据

**try** {

**for** (**int** i = 0; i < rs.rowCount; i++) {

EMP emp = **new** EMP();

rs.autoFillRow(emp);

System.*out*.println(emp);

rs.next();

}

} **finally** {

rs.clearAll();

}

}

注：SqlServerPaginationImp和SybasePaginationImp实现依赖与Sp\_Pagination数据库存储过程，具体请参考java doc的api文档。

### 2.2.7复杂条件组合查询器

在做数据库统计分析开发过程中，我们经常会碰到复杂条件组合查询的问题，其特点是：检索字段是固定（就是显示结果集的字段），但查询的条件是多变的，体现在各个字段的条件任意自由组合。如果为每一次组合情况编写一个SQL查询语句的话，显然是一件费时费力的乏味活。BJAF框架为了解决这样一个问题，实现了一个复杂条件组合查询器CompositeQueryOperator。它支持查询条件的任意组合，动态构建SQL语句，让你乏味而啰嗦的复杂条件查询开发过程解脱出来。

com.beetle.framework.persistence.composite   
类 CompositeQueryOperator

java.lang.Object

继承者 [com.beetle.framework.persistence.access.operator.BaseOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\BaseOperator.html)

继承者 [com.beetle.framework.persistence.access.operator.QueryOperator](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\QueryOperator.html)

继承者 **com.beetle.framework.persistence.composite.CompositeQueryOperator**

|  |  |
| --- | --- |
| **构造方法摘要** | |
| [**CompositeQueryOperator**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\composite\CompositeQueryOperator.html#CompositeQueryOperator())() |  |

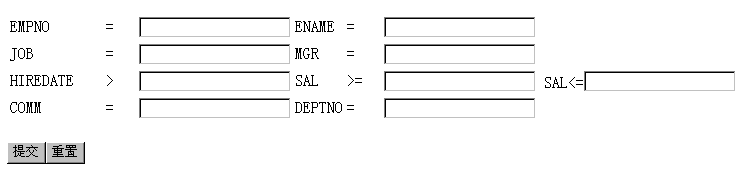
|  |  |
| --- | --- |
| **方法摘要** | |
| protected  void | [**accessImp**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\composite\CompositeQueryOperator.html#accessImp())() |
| void | [**addParameter**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\composite\CompositeQueryOperator.html#addParameter(java.lang.Object))(java.lang.Object value)            添加sql语句参数 |
| void | [**addParameter**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\composite\CompositeQueryOperator.html#addParameter(com.beetle.framework.persistence.access.operator.SqlParameter))([SqlParameter](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\access\operator\SqlParameter.html) sqlParamter)            添加sql语句参数 |
| void | [**addParameter**](file:///D:\Study\code\JAF\dist\apidoc\com\beetle\framework\persistence\composite\CompositeQueryOperator.html#addParameter(java.lang.String, java.lang.String, java.lang.Object))(java.lang.String parameterName, java.lang.String OperateSymbol, java.lang.Object value)            添加组合查询参数 |

可见，CompositeQueryOperator继承自QueryOperator查询器，所以它使用方式基本上与QueryOperator保持一致。它只是多了一个addParameter方法来构建各种组合的条件。

示例如下：

EMP表为例，EMP的字段有（EMPNO、ENAME、JOB、MGR、 HIREDATE、SAL、COMM 和DEPTNO）假设需要按以上各个字段自由组合来查询EMP表的数据。

典型的条件组合界面类似下图所示：



如果按照传统作法，对上面的每种组合编写一个SQL语句，显然会脑瘫的:-)，复杂组合查询器CompositeQueryOperator的引入，只要下面简单代码即可解决问题：

**static** **void** testCompositeQuery() {

String EMPNO = **null**;// 设置参数（从页面获取输入参数）

String ENAME = **null**;

String JOB = **null**;

BigDecimal MGR = **null**;

Timestamp HIREDATE = **null**;

BigDecimal SAL\_1 = **null**;

BigDecimal SAL\_2 = **null**;

BigDecimal COMM = **null**;

BigDecimal DEPTNO = **new** BigDecimal(10);

CompositeQueryOperator cqo = **new** CompositeQueryOperator();

cqo.setDataSourceName("SYSDATASOURCE\_oracle");// 数据数据源

cqo.setSql("select \* from emp");// 数据查询语句（只填显示字段select部分，无需写where子句）

cqo.addParameter("EMPNO", "=", EMPNO);// 添加各种组合条件字段

cqo.addParameter("ENAME", "=", ENAME);

cqo.addParameter("JOB", "=", JOB);

cqo.addParameter("MGR", "=", MGR);

cqo.addParameter("HIREDATE", ">", HIREDATE);

cqo.addParameter("SAL", ">=", SAL\_1);

cqo.addParameter("SAL", "<=", SAL\_2);

cqo.addParameter("COMM", "=", COMM);

cqo.addParameter("DEPTNO", "=", DEPTNO);

cqo.access();// 执行查询

List<EMP> empList = cqo.getResultList(EMP.**class**);

System.*out*.println(empList);

}

可见，一切变得很简单。

81278576189468.jpg值得注意的是，当某个条件参数不参与检索查询，则此参数应设置为null而不是空字符””，当然，你也可以把这个参数注释掉。

## 2.3DAO模式支持

前面的数据存取组件解决了java访问关系数据库的编程问题。接下来，我们面临的问题是：在一个过程中如何让持久性逻辑操作与业务性逻辑操作分离，解藕成数据持久层和业务逻辑层呢？解决问题的答案就是DAO模式。

### 2.3.1什么是DAO

什么是DAO模式？DAO也就是Data Access Object数据访问对象（接口），它介于数据库资源和业务逻辑之间，其意图是将底层数据访问操作与高层业务逻辑完全分离开。一般情况下，构成一个典型的DAO模式有以下组件：

值对象（数据传输对象，DTO）

DAO接口，把对数据库所有的操作定义成一个个抽象方法。

DAO接口具体实现类，针对不同数据库（或不同的技术）来具体实现DAO接口定义的各种操作。

DAO工厂，为了统一维护、优化和管理DAO数据访问对象，采用Factory设计模式建立工厂类。

在BJAF数据存取框架中，DAO接口定义，开发人员要根据具体需要分析来决定；DAO接口实现代码中，用到数据存取部分的代码用本数据存取组件来完成。

### 2.3.2模式使用

BJAF数据存取框架实现DAO设计模式。其工作原理如下图所示：

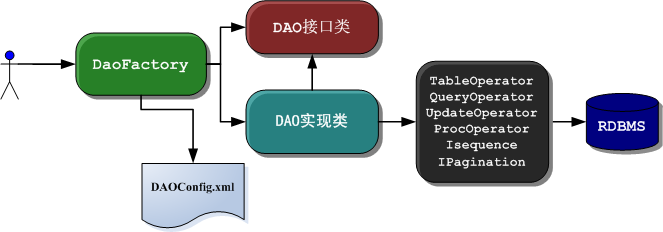


图2-3DAO模式工作原理图

如上图所示，采取DAO模式开发过程如下：

首先，根据设计需要定义数据访问DAO接口，我们一般以一个数据表为最小单元定义一个数据访问接口，例如对EMP数据表定义一个IEmpDao接口：

**package** psdemo.dao;

**import** java.util.List;

**import** psdemo.valueobject.Emp;

**public** **interface** IEmpDao {

**int** insert(Emp emp);

**int** update(Emp emp);

**int** delete(Integer id);

Emp select(Integer id);

**int**[] updateBatch(List empList);

List findEmpBySal(Float sal);

}

81278576189468.jpg当然，数据接口定义界定要根据具体设计来考虑。一般，为了重用，数据访问接口（方法）定义越细越好，体现持久化数据存取逻辑，不能涉及到业务逻辑。一般地，一个数据接口用一个简单的非嵌套SQL语句实现来作为衡量标准。总之，数据接口为一个细粒度对象。

其次，编写DAO接口的实现类，在实现类中，引用TableOperator、QueryOperator等数据存取操作组件来具体完成与数据库系统的交互，例如，实现IEmpDao接口的PsEmp类代码如下：

**package** psdemo.dao.imp;

**import** java.util.List;

**import** psdemo.dao.Const;

**import** psdemo.dao.IEmpDao;

**import** psdemo.valueobject.Emp;

**import** com.beetle.framework.persistence.access.operator.TableOperator;

**public** **class** PsEmp **implements** IEmpDao {

**private** TableOperator tableOperator;

**public** PsEmp() {

**this**.tableOperator = **new** TableOperator(Const.*dataSourceName*, "EMP",

Emp.**class**);

}

**public** **int** delete(Integer id) {

**return** tableOperator.deleteByPrimaryKey(id);

}

**public** **int** insert(Emp emp) {

**return** tableOperator.insert(emp);

}

**public** Emp select(Integer id) {

**return** (Emp) tableOperator.selectByPrimaryKey(id);

}

**public** **int** update(Emp emp) {

**return** tableOperator.update(emp);

}

**public** **int**[] updateBatch(List empList) {

**return** tableOperator.updateBatch(empList);

}

**public** List findEmpBySal(Float sal) {

Object params[] = **new** Object[1];

params[0] = sal;

List empList = tableOperator.selectByWhereCondition("where sal>=?",

params);

**return** empList;

}

}

然后，在DAOConfig.xml配置文件装配实现类：

<?xml version="1.0" encoding="GBK"?>

<binder>

<item interface="psdemo.dao.IDeptDao" implement="psdemo.dao.imp.PsDept"/>

<item interface="psdemo.dao.IEmpDao" implement="psdemo.dao.imp.PsEmp"/>

</binder>

最后，客户端使用DaoFactory作为统一面板（Facade）来获取数据接口，调用具体方法完成任务：

**public** **static** **void** main\_(String[] args) {

IEmpDao empDao = (IEmpDao) DaoFactory.*getDaoObject*(IEmpDao.class);

List empList = empDao.findEmpBySal(**new** Float(2000));

**if** (!empList.isEmpty()) {

**for** (**int** i = 0; i < empList.size(); i++) {

Emp emp = (Emp) empList.get(i);

System.*out*.println(emp.getEmpNo());

System.*out*.println(emp.getEname());

System.*out*.println(emp.getHireDate());

System.*out*.println(emp.getJob());

System.*out*.println(emp.getMgr());

System.*out*.println(emp.getSal());

System.*out*.println(emp.getComm());

System.*out*.println(emp.getDeptNo());

System.*out*.println("--");

}

}

empList.clear();

}

另外，开发人员也可以不采取DAOConfig.xml文件来装配（一般出现在只有一种实现的情况下），此时，调用代码如下：

**public** **static** **void** main(String[] args) **throws** ConnectionException,

SQLException {

IEmpDao empDao = (IEmpDao) DaoFactory.*getDaoObject*(PsEmp.**class**);

List empList = empDao.findEmpBySal(**new** Float(2000));

**if** (!empList.isEmpty()) {

**for** (**int** i = 0; i < empList.size(); i++) {

Emp emp = (Emp) empList.get(i);

System.*out*.println(emp.getEmpNo());

System.*out*.println(emp.getEname());

System.*out*.println(emp.getHireDate());

System.*out*.println(emp.getJob());

System.*out*.println(emp.getMgr());

System.*out*.println(emp.getSal());

System.*out*.println(emp.getComm());

System.*out*.println(emp.getDeptNo());

System.*out*.println("--");

}

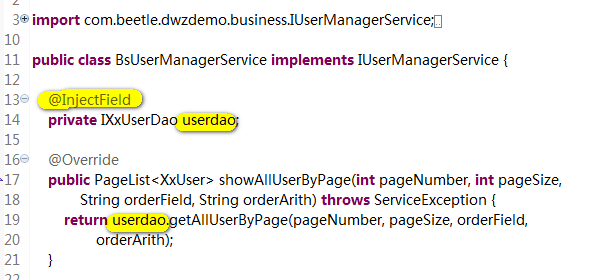
}

empList.clear();

}

对于业务层的Service引用Dao的接口可采取字段注入的方式，例如：





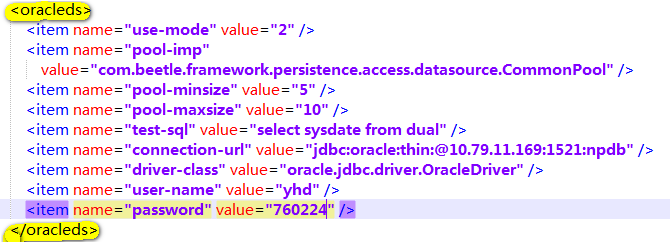
详见业务层介绍。

## 2.4持久层代码自动生成

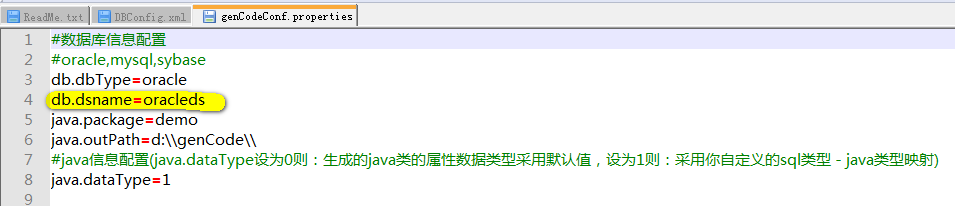
在持久层开发中，我们一般的开发过程是，先设计数据库，建立各种数据表；然后针对这些表创建一个以之相对应的Java值对象；再针对表的基本操作，创建Dao接口及其实现类。这个比较啰嗦，为了方便开发，提高效率。BJAF提供一个持久层代码生成工具。

其原理是读取数据库中表结构的元数据信息，结合BJAF的DAO开发模式，生成相应的Java类。其操作步骤如下：

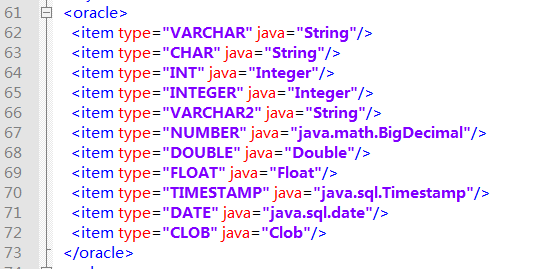
1. 配置sdk开发包\dist\config的DBConfig.xml文件，配置表所在的数据源，例如：



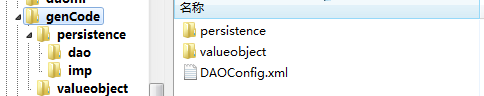
1. 配置sdk开发包\dist\config的genCodeConf.properties文件，设置数据类型、所用的数据源、Java包名、生成文件所在目录等属性



如果java.dataType=0则代表程序会自动根据数据库的元数据信息与Jdbc驱动信息来生成Java值对象属性对应的Java数据类型；如果java.dataType=1则可以通过\dist\config的genCodeConf.xml文件类手动指定相应的字段数据类型。例如：



1. 最后，执行” **ant genDbCode**”命令即可。例如：代码生成结果目录



注：生成代码后，各个Java对象的命名或字段与标准Java命名规范不太一致，手动调整一下即可。

# 3.业务层

## 3.1业务层介绍

业务层是J2EE应用的核心，它负责处理应用的业务数据与逻辑。在J2EE规范中，业务层属于EJB容器的技术范畴。这样界定的缘由是EJB容器本身提供了各种开发业务逻辑的功能服务，如：事务管理服务、消息服务、远程调用服务、目录命名服务等等。这样，开发人员就可以把精力集中在处理业务逻辑上面，而无需关心各种功能的底层技术实现，从而加快应用系统的开发。

BJAF框架在业务层针对业务模型的特点，实现了一套完整的业务层业务框架的解决方案，包括：Command和Service服务组件。本章将在设计理念与使用模式上对上面框架、组件进行详细说明。

BJAF业务框架具备以下主要功能特征：

* 结构简单、便于开发、调式和维护。
* 框架具备控制或限制开发人员出现越界设计情况的能力。
* 支持开发分布式和集中式应用，无需修改代码，即可支持分布式和集中式应用相互切换，使得应用系统具备良好的缩放性。而且，这些对开发人员是透明的。
* 完全支持和兼容各种J2EE的EJB容器。提供本地接口和远程接口。但并不依赖于EJB容器，没有EJB容器也能在框架的封装本地微容器里面运行。
* 模式化编程，保障所有的业务逻辑的开发方式和风格的一致性和代码的质量的可控性。
* 完全支持事务处理，保证业务数据完整性。而且，事务的处理方式对开发人员是透明的。
* 支持业务组件的AOP编程。
* 支持同步/异步业务逻辑过程。
* 自动识别线程安全对象进行缓存处理，在框架业务对象体内，支持资源连接共享，提高资源利用率和性能效率。
* 业务层DIC（**D**ependency **I**njection **C**ontainer）依赖注入容器支持，减少代码依赖，简化编程。

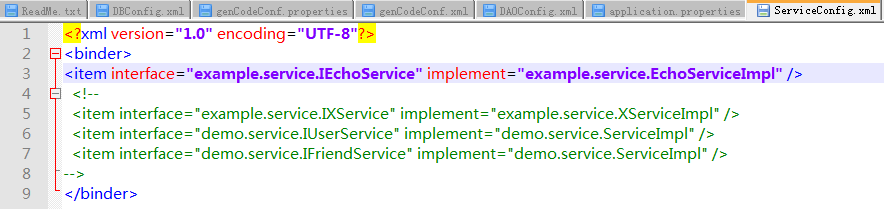
81278576189468.jpg由于EJB容器提供众多功能服务，由于其技术复杂性和容器的依赖性，往往给开发和调试带来困难与不便，使EJB容器显得有点“重”，因而，市场出现一些“轻”量级别Java Bean容器（如：Spring）的替代方案。但无论“重”与“轻”，当前市场上没有一套真正针对业务层模型的业务开发框架，很多所谓的J2EE框架产品只提供了Web层和数据持久层，在业务层却含糊不轻，暧昧称为整合层，确没有明确的编程模型。BJAF框架出现，填补这个空白。

BJAF业务框架提供明确的编程模型，溶合重轻容器（BJAF内置自己轻量容器实现），解耦Pojo业务组件与EJB容器的依赖关系，便于开发与调试，以及支持事务处理透明性和集中式/分布式应用部署的透明性。

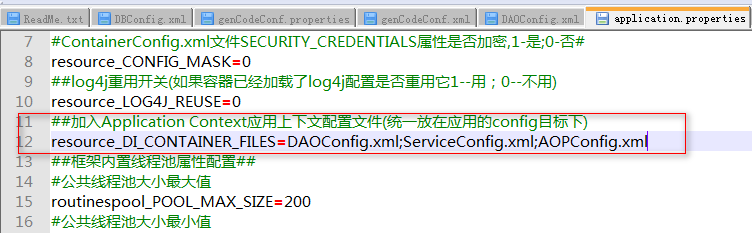
## 3.2DIC依赖注入容器

DI（**D**ependency **I**njection）依赖注入模式引入最大的好处是可以替换到传统的Factory模式，有利降低代码的耦合度，简化编程。BJAF2.x版本通过内置的DIC来支持依赖注入的编程模式。在BJAF框架中，DIC的作业范围只涵盖业务层和持久层，Web层不在其管理范围。这是因为在BJAF中Web层与业务层是可以物理分开的（本地与分布式透明），而让DIC透明支持本地/分布式我们认为是没有必要的，所以Web层要获取业务层的业务对象（Service）只能通过DIC的Context而不能采取注入的方式。

要使用DIC进行管理，必须采取Interface-Implement形式，而且要通过配置文件来注册。注册文件的格式如下：



在<binder>标签内一项一项的绑定接口与其实现类的关系。框架将默认DAOConfig.xml;ServiceConfig.xml;AOPConfig.xml即持久层Dao组件服务层Service组件和Service的AOP拦截器注册到DIC（即这些组件纳入容量来管理），对于自定义的，文件名根据自己需要命名，然后在application.properties的resource\_DI\_CONTAINER\_FILES属性里面设置即可。如：



在BJAF3.x中已经实现了DIC容器零配置，也就是说可以不配置binder的配置xml文件，只需要指定这些接口及其实现类的包路径，框架会自动扫描加载，方面的编程。指定的参数在application.properties的文件中：

|  |
| --- |
| ##指定DAO包路径后台自动扫描加载无需DAOConfig.xml文件配置  resource\_DI\_DAO\_PACK\_PATH=  ##指定Service包路径后台自动扫描加载无需ServiceConfig.xml文件配置  resource\_DI\_SERVICE\_PACK\_PATH= |

另外，需要说明的是包路径自动扫描虽然方便，但是只是支持接口-实现这中类型的自动绑定，对于AOP还有纯实现类目前是不支持自动扫描的。所以配置文件还是推荐使用的，因为它最灵活。配置文件和自动扫描可以混合使用，不会冲突。

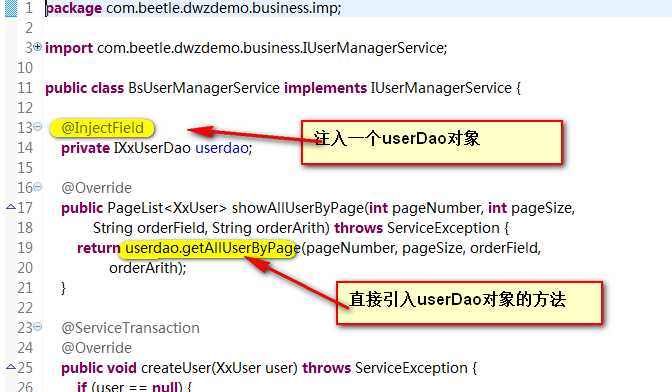
DIC容器的实现类是DIContainer，其类图如下：



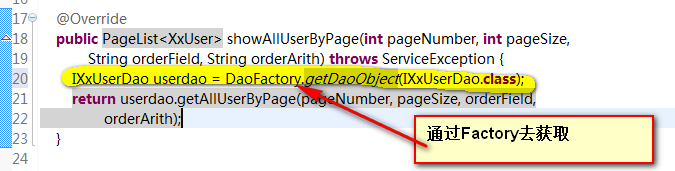
DIC容器可以调用init方法进行容器的初始化工作（主要是读取相关配置，完成各个组件类的依赖关系的装配及实例缓存过程）；如果没有显性调用init方法，容器会在首次执行retrieve获取对象实例时进行初始工作。另外，业务层内置的BusinessAppSrv的RPC服务器在其启动事件里面已经加入DIC容器的初始化动作；对于集中式的Web应用可以在BJAF的Web启动事件也可以通过设置application.properties配置文件web\_businessAppSrv\_enabled=true来在Web容器启动时完成DIC容器的初始化动作。DIC容器管理的对象，通过retrieve方法获取。

### 3.2.1 InjectField注解

InjectField注解用于业务层Service之间或Service引用Dao接口，不可以用于Dao与Dao之间（因为这样不符合我们框架的编程范式），使用InjectField代替原有通过Factory方式获取对象实例，有利于降低代码的耦合度。



如果不使用InjectField，则代码是：



两者从功能上来说是等价的。

### 3.2.2 Aop注解

AOP注解可以实现针对Service方法的拦截，实现AOP编程。针对Service方法拦截其过程如下：

（1），编写一个拦截器，通过扩展AopInterceptor类，实现其Before和After方法，如果拦截前后都不能满足要求，则可以重载interrupt和interruptResult两个方法，将interrupt返回为true，在interruptResult里面完全实现所需逻辑和返回结果，这种情况，被拦截的方法的逻辑将不会再执行。

**package** example.aop;

**import** java.lang.reflect.Method;

**import** com.beetle.framework.resource.dic.aop.AopInterceptor;

**public** **class** EchoServiceInerceptor **extends** AopInterceptor {

@Override

**protected** **void** before(Method method, Object[] args) **throws** Throwable {

System.*out*.println("before:arg[" + print(args) + "]methodName["

+ method + "]");

}

@Override

**protected** **void** after(Object returnValue, Method method, Object[] args)

**throws** Throwable {

System.*out*.println("after:args[" + print(args) + "]returnvalue["

+ returnValue + "]methodName[" + method + "]");

}

}

（2），将这个拦截器在AOPConfig.xml中注册，如：

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

<binder>

<item aopid=**"echo"** interceptor=**"example.aop.EchoServiceInerceptor"** />

</binder>

（3），在要拦截方法上使用AOP注解来声明此方法需要被拦截和使用哪个拦截器，如：

**package** example.aop;

**import** com.beetle.framework.resource.dic.def.Aop;

**public** **class** EchoServiceImpl **implements** EchoService {

@Aop(id = "echo")

**public** String echo(String word) {

**return** "echo:" + word;

}

@Aop(id = "printEcho")

**public** **void** printEcho(String word) {

System.*out*.println("printEcho:" + word);

}

**public** **void** other() {

System.*out*.println("OK");

}

}

（4），编写客户端执行

**public** **static** **void** main(String[] args) {

DIContainer di = DIContainer.*getInstance*();

di.init();

EchoService service = di.retrieve(EchoService.**class**);

String e = service.echo("hi,Henry");

System.*out*.println(e);

service.printEcho("hi,tom");

service.echo("hi,Henry");

}

执行结果：

before:arg[hi,Henry]methodName[public abstract java.lang.String example.aop.EchoService.echo(java.lang.String)]

after:args[hi,Henry]returnvalue[echo:hi,Henry]methodName[public abstract java.lang.String example.aop.EchoService.echo(java.lang.String)]

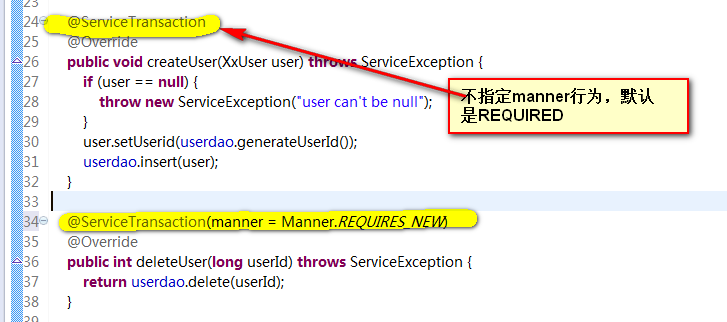
echo:hi,Henry

### 3.2.3 ServiceTransaction注解

在BJAF框架中，事务属于业务层的范畴。通过ServiceTransaction注解来保证一个方法的事务（事务支持本地的JDBC事务和多数据源的JTA事务）的完整性，ServiceTransaction支持两种事务行为：

* **REQUIRED**，需要事务，如果已存在事务，则参与，没有则开一个新的事务
* **REQUIRES\_NEW**，无论是否有事务存在，都会开启一个独立的新事务，不会参与原来的事务

示例如下：



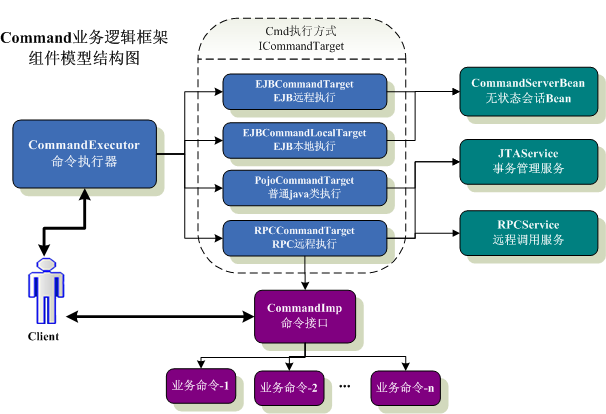
注：注解的ServiceTransaction实现事务的回滚是依赖于异常的，就是说对于一个声明了ServiceTransaction注解的方法，在执行过程中，发生了任何类型的异常，这个方法启动的事务就会回滚，如果在一个方法内部吃掉异常，手动回滚事务的话，那可以使用**ServiceTransactionRollback.rollbackByHand()**方法回滚事务。

## 3.3 Command框架

Command业务框架模型最早出现于IBM的San Francisco业务框架中，在BJAF框架中，我们结合EJB Command模式的思想和我们在业务逻辑开发经验，实现了面向业务逻辑开发Command业务框架模型。

### 3.2.1框架模型

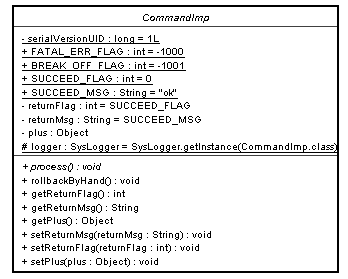
BJAF Command业务框架的模型如下：



从上图可知：整个框架主要由三个不同模块组成，分别是：

**业务命令接口（CommandImp）**

定义一个统一对外的业务命令接口CommandImp，其定义如下类图所示：



可见，CommandImp提供了一个抽象方法process()，这是每一个具体业务命令对象都必须实现的方法，在process()方法体内完成***Use Case***[1]（用例）业务逻辑执行。在设计策略上，通常一个命令业务对象与一个*Use Case*相对应，这样，系统各个功能*Use Case*业务逻辑，最终对外体现为一个个POJO业务命令。

[1] Use Case是UML的术语，UML经过多年的发展和普及，已经成为系统分析、开发的必备建模语言。如果读者还不熟悉的话，最好马上恶补一下了。关于UML的资料网上很多，笔者在这里就不再介绍了。

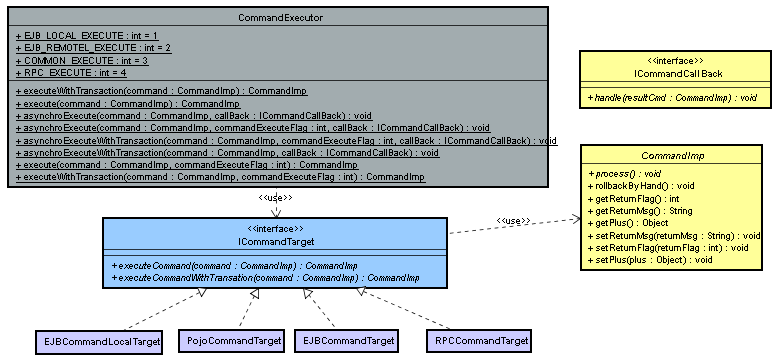
由于在EJB Command设计模式中，笨拙的异常处理，使得调用的客户端不能方便利用异常来跟进命令对象的运行情况。所以需要我们从设计上改变这个情况，returnFlag属性用来标识命令对象运行的各种可能遇到的结果（或状况）；而returnMsg属性则来辅助描述发生这些结果时的情况。这样，客户端都可以根据returnFlag和returnMsg来判别命令运行的情形，指导表示层的下一步操作。

针对命令执行失败的情形，CommandImp专门提供了一个FATAL\_ERR\_FLAG致命错误常量，用来标识执行命令抛出CommandException异常的结果。因为CommandExecutor执行命令对象时如果抛出了CommandException，我们认为这个命令执行失败了，returnFlag这时候应该为FATAL\_ERR（致命错误）标记。

在一个Command命令体内实现业务逻辑的时候，虽然开发人员无需关心事务的界定问题，当执行逻辑过程发现异常时，框架会自动回滚事务，但是对于一些非程序异常的逻辑性错误，框架是无法捕捉的，这个时候，开发人员需要手工调用rollbackByHand()方法进行事务回滚。

**命令执行（CommandExecutor）模块**

开发人员只负责业务命令对象的实现，而无需关心此命令对象的执行，这是我们本框架要实现的目标之一。命令执行模块担负了命令执行的责任，它提供了多种的执行方式：



|  |  |
| --- | --- |
| **方式** | **说明** |
| EJBCommandTarget | 利用EJB远程接口，把命令提交到远程的EJB容器运行 |
| EJBCommandLocalTarget | 使用EJB本地接口，把命令提交到本地EJB容器执行 |
| PojoCommandTarget | 普通的java对象调用，把命令提到本框架的JTAService微容器里面执行 |
| RPCCommandTarget | 同PojoCommandTarget方式一样，只是利用本框架PRCServer服务，令其具备远程提交的能力，实现不依赖EJB容器也能支持分布式开发的目的。 |

上面执行方式中，前两种是和后两种相对应的。4种方式，最后统一由CommandExecutor命令执行器利用commandExecuteFlag标记来支配决定。

从上图可知，执行方式对应CommandExecutor所提供EJB\_LOCAL\_EXECUTE等4个静态属性标记。另外，针对命令执行过程的数据完整性，执行提供了事务执行和非事务执行两种模式。详细说明见下表：

|  |  |
| --- | --- |
| **方法** | **含义说明** |
| execute(command:CommandImp)  :CommandImp | 此方法同步为非事务模式执行，命令执行方式由系统配置文件ExeMode.xml[1]决定。  若命令所处理的业务逻辑过程只涉及到数据的只读查询，而不涉及数据修改，又或即使会修改数据，但对此数据完整性并没有要求的情景下，则可采取非事务模式执行。  由于少了事务管理，节约系统资源，执行效率相对采取事务模式执行要高。 |
| executeWithTransaction (command:CommandImp):CommandImp | 为同步事务模式执行，其命令执行方式同上，也由配置文件决定。  若命令所处理的业务逻辑过程涉及到数据修改，并对数据完整性有严格要求的情景，则必须采取事务模式执行来保证数据的完整性。 |
| execute(command:CommandImp,  commandExecuteFlag:int)  :CommandImp | 同步非事务模式执行，其命令执行方法由参数commandExecuteFlag决定。可供选择参数为上面提到的4个静态属性。提供此方法的目的是，面对复杂应用，命令执行方式是多种方式并存的，命令如何执行，调用代码根据具体的需求决定，而不是通过配置文件来硬性规定。 |
| executeWithTransaction  (command:CommandImp,  commandExecuteFlag:int)  :CommandImp | 同步为事务模式执行，其命令执行方法由参数commandExecuteFlag决定。 |
| asynchroExecute(command:CommandImp,callBack:ICommandCallBack) : void | 异步非阻塞非事务方式执行；调用此方式，命令以任务形式会放入队列，由框架后台执行，同时，此命令执行的结果将有ICommandCall回调接口的实现类中处理。可以通过ExeMode.xml文件中的“ASYNCHRO\_CALL\_QUEUE\_SIZE”参数来调整此命令任务队列的消费线程数以调整异步并发处理速度。 |
| asynchroExecute(command : CommandImp, commandExecuteFlag : int, callBack : ICommandCallBack) : void | 功能同上，其命令执行方法由参数commandExecuteFlag决定。可供选择参数为上面提到的4个静态属性。提供此方法的目的是，面对复杂应用，命令执行方式是多种方式并存的，命令如何执行，调用代码根据具体的需求决定，而不是通过配置文件来硬性规定。 |
| asynchroExecuteWithTransaction(command : CommandImp, callBack : ICommandCallBack) : void | 异步非阻塞事务方式执行，实现机制与asynchroExecute方法一样 |
| asynchroExecuteWithTransaction(command : CommandImp, commandExecuteFlag : int, callBack : ICommandCallBack) : void | 功能同上，其命令执行方法由参数commandExecuteFlag决定。可供选择参数为上面提到的4个静态属性。提供此方法的目的是，面对复杂应用，命令执行方式是多种方式并存的，命令如何执行，调用代码根据具体的需求决定，而不是通过配置文件来硬性规定。 |
| [1]application.properties文件是本业务逻辑框架相关属性的配置文件，它包含了，执行器的执行方式，核心EJB的JNDI名称，PRC远程请求链接等配置信息。有关Command框架部分的的配置项如下： | |

**组件容器服务模块。**

本模块包括3个基本的服务，如下：

|  |  |
| --- | --- |
| **服务名称** | **含义说明** |
| CommandServerBean | 利用SLSB来获取EJB容器相应的功能服务。这样，事务管理和分布式的支持完全由CommandServerBean利用EJB容器来实现。同时，还可以利用了EJB容器现有的对象池、线程池，使得框架具备良好的并发性能和高效的执行效率。 |
| JTAService | 本框架利用开源包Atomikos实现的事务管理服务。本框架利用它，脱离了对EJB容器的事务管理服务的依赖。 |
| PRCService | 本框架利用内置RPC协议实现远程调用服务。利用它，使本框架无需EJB容器也可以支持分布式应用。 |

客户端（Client）只需与命令执行器（CommandExecutor）以及具体业务实现命令类（Command）打交道。客户端把业务命令类实例化成对象，输入相关参数后，再提交给命令执行器去运行；而命令执行器根据配置按照指定的执行方式执行命令后，又把此执行后的命令对象（包含结果数据）返回给客户端，最后，客户端获得命令执行的结果标记（returnFlag）来判别或指引表示层逻辑的下一步操作。

81278576189468.jpg在利用EJB容器开发的应用中，本框架通过封装命令执行器让业务命令与EJB容器完全解藕，使得可以利用一个普通java对象来开发命令，其编写和调试完全不依赖于EJB，开发变得更加快速、容易。而且，整个框架只需一个现成的SLSB就可以了，程序员无需再编写其它的SLSB，大大降低了对开发人员能力的要求，从某种程度来说，这意味着节约开发的成本。

另外，即使没有EJB容器，Pojo命令也可以在本框架微容器内运行，而这种容器切换，通过配置文件动态设置，对开发人员来说完全透明。

对于EJB容器与本框架微容器选择，我们建议在生产环境使用EJB容器，在开发和调试环境，使用微容器，当然，在不采取EJB容器，开发服务应用程序时，微容器也是一个不错选择。

### 3.2.2使用示例

例如PetStore系统里面用户开户例子，开户过程涉及数据库登录信息表、帐户信息表、个人配置信息表、栏目配置信息表等。这个用例过程是，首先把相关配置数据显示到视图页面，用户挑选，并输入个人信息提交保存到数据库，完成开户的过程。这个用例用Command业务框架实现如下：

1. 编写CreateAccountCmd创建账户业务命令：

**package** com.beetle.PetStore.business.storefront.account;

**import** com.beetle.PetStore.business.storefront.BusinessObjectCommonException;

**import** com.beetle.PetStore.persistence.dao.IAccountDao;

**import** com.beetle.PetStore.persistence.dao.IProfileDao;

**import** com.beetle.PetStore.persistence.dao.ISignonDao;

**import** com.beetle.PetStore.valueobject.TbAccount;

**import** com.beetle.PetStore.valueobject.TbBannerdata;

**import** com.beetle.PetStore.valueobject.TbProfile;

**import** com.beetle.PetStore.valueobject.TbSignon;

**import** com.beetle.framework.business.command.CommandException;

**import** com.beetle.framework.business.command.CommandImp;

**import** com.beetle.framework.persistence.dao.DaoFactory;

**public** **class** CreateAccountCmd **extends** CommandImp {

**private** **static** **final** **long** *serialVersionUID* = 3809472631884829872L;

**private** **int** stepNo;

**private** TbBannerdata bannerDatas[];

**private** TbProfile profile;

**private** TbAccount account;

**private** TbSignon signon;

**public** **void** setAccount(TbAccount account) {

**this**.account = account;

}

**public** **void** setProfile(TbProfile profile) {

**this**.profile = profile;

}

**public** **void** setSignon(TbSignon signon) {

**this**.signon = signon;

}

**public** TbBannerdata[] getBannerDatas() {

**return** bannerDatas;

}

**public** **void** setStepNo(**int** stepNo) {

**this**.stepNo = stepNo;

}

**public** **void** process() **throws** CommandException {

**if** (**this**.stepNo == 1) {// 查询TbBannerdata

queryBanners();

} **else** **if** (**this**.stepNo == 2) {// 保存新帐号信息到数据库

IAccountDao accountDao = (IAccountDao) DaoFactory

.*getDaoObject*("IAccountDao");

IProfileDao profileDao = (IProfileDao) DaoFactory

.*getDaoObject*("IProfileDao");

ISignonDao signonDao = (ISignonDao) DaoFactory

.*getDaoObject*("ISignonDao");

signonDao.insertSignon(**this**.signon);

accountDao.insertAccount(**this**.account);

profileDao.insertProfile(**this**.profile);

**this**.setReturnFlag(0);

}

}

**private** **void** queryBanners() **throws** BusinessObjectCommonException {

QueryAccount qa = **new** QueryAccount(**null**, 2);

qa.query();

**this**.bannerDatas = qa.getBannerDatas();

}

}

1. 从表示层编写一个控制器来调用这个业务层的业务命令：

**private** View doRegist(WebInput webInput) **throws** ControllerException {

View view;

TbSignon signon = (TbSignon) webInput

.getParameterValuesAsFormBean(TbSignon.**class**);

TbAccount account = (TbAccount) webInput

.getParameterValuesAsFormBean(TbAccount.**class**);

account.setStatus("OK");

account.setUserid(signon.getUsername());

TbProfile profile = (TbProfile) webInput

.getParameterValuesAsFormBean(TbProfile.**class**);

profile.setUserid(signon.getUsername());

**if** (profile.getMylistopt() == **null**) {

profile.setMylistopt(**new** Integer(0));

}

**if** (profile.getBanneropt() == **null**) {

profile.setBanneropt(**new** Integer(0));

}

CreateAccountCmd cmd = **new** CreateAccountCmd();

cmd.setStepNo(2);

cmd.setAccount(account);

cmd.setProfile(profile);

cmd.setSignon(signon);

**try** {

// 保证数据完整性，使用事务执行

cmd = (CreateAccountCmd) CommandExecutor

.*executeWithTransaction*(cmd);

**if** (cmd.getReturnFlag() == 0) {

// after registed,then sign on the user;

//call signoncontroller to implement it;

view = DoormanController.*signOn*(webInput);

} **else** {

ModelData datas = **new** ModelData();

datas.put(Const.*RETURN\_MSG*, cmd.getReturnMsg());

view = **new** View(Const.*ERR\_VIEW*, datas);

}

} **catch** (CommandExecuteException cee) {

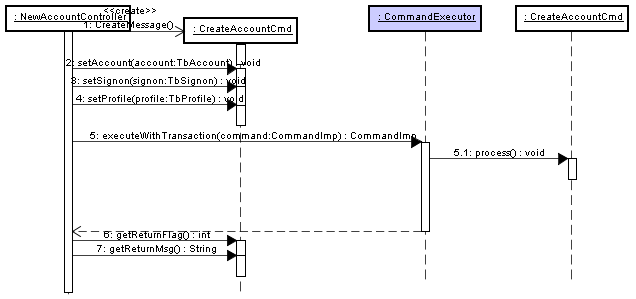
**throw** **new** ControllerException(cee);

}

**return** view;

}

上面执行过程的顺序图如下：



可见，控制器负责对Cmd填充参数（此时Cmd当值对象使用），然后提交给执行器；执行器调用Cmd（此时Cmd当业务逻辑处理对象使用）的process()方法完成业务逻辑过程的处理。在整个过程中，事务处理及在哪里执行，完全有执行器来决定，这些逻辑对开发人员完全透明。

另外，上面Cmd是采取同步阻塞的方式执行的，这是由此用例的需求决定的。假设我们现在需求变成了用户提交的账户信息是需要经过后台另外流程审核后才能决定是否创建账户成功，用户在页面提交账户后，我们只需告诉用户信息已受理的，成功与否另行通知。这样的话，我们在这里就可以采取异步非阻塞方式来执行这个Cmd了，见下面示例代码：

private View doRegist(WebInput webInput) throws ControllerException {

View view;

TbSignon signon = (TbSignon) webInput

.getParameterValuesAsFormBean(TbSignon.class);

TbAccount account = (TbAccount) webInput

.getParameterValuesAsFormBean(TbAccount.class);

account.setStatus("OK");

account.setUserid(signon.getUsername());

TbProfile profile = (TbProfile) webInput

.getParameterValuesAsFormBean(TbProfile.class);

profile.setUserid(signon.getUsername());

if (profile.getMylistopt() == null) {

profile.setMylistopt(new Integer(0));

}

if (profile.getBanneropt() == null) {

profile.setBanneropt(new Integer(0));

}

CreateAccountCmd cmd = new CreateAccountCmd();

cmd.setStepNo(2);

cmd.setAccount(account);

cmd.setProfile(profile);

cmd.setSignon(signon);

try {

// 采取异步非阻塞事务执行

CommandExecutor.asynchroExecuteWithTransaction(cmd,

new ICommandCallBack() {

public void handle(CommandImp resultCmd) throws CommandException {

AccountCmd cmd2 = (AccountCmd) resultCmd;

if (cmd2.getReturnFlag() == 0) {

//审核处理流程...

}

}

});

ModelData datas = new ModelData();

datas.put(Const.RETURN\_MSG, "您的请求已经受理，结果另行通知！");

view = new View(Const.ERR\_VIEW, datas);

} catch (CommandExecuteException cee) {

throw new ControllerException(cee);

}

return view;

}

在高并发请求、计算响应缓慢等场景采取异步非阻塞执行方式是个理想选择。

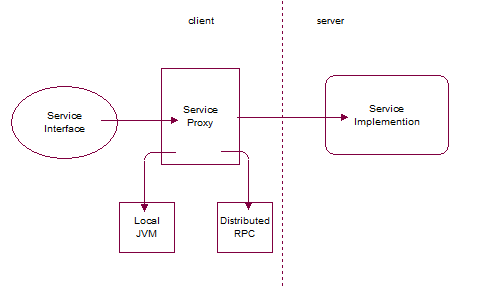
## 3.4 Service服务框架

### 3.4.1 Service界定

Service是应用系统中可重用的组件，它封装可重复使用的业务逻辑；一般地Service应为粗粒度的服务。作为SOA（Service Oriented Architecture，面向服务架构）的关键组成部分，Service是作为向外部系统提供服务。对于BJAF框架来说，Service组件模块只界定为外部系统通信接口（统一在**ServiceConfig.xml**文件中注册），系统内部服务的定义（或叫Component）和模型我们是没有介入的。

### 3.4.2 Service模型

BJAF业务层的Service模型采取Java标准的Interface-Implemente（接口-实现）编程模型。其实现模型如下图所示：

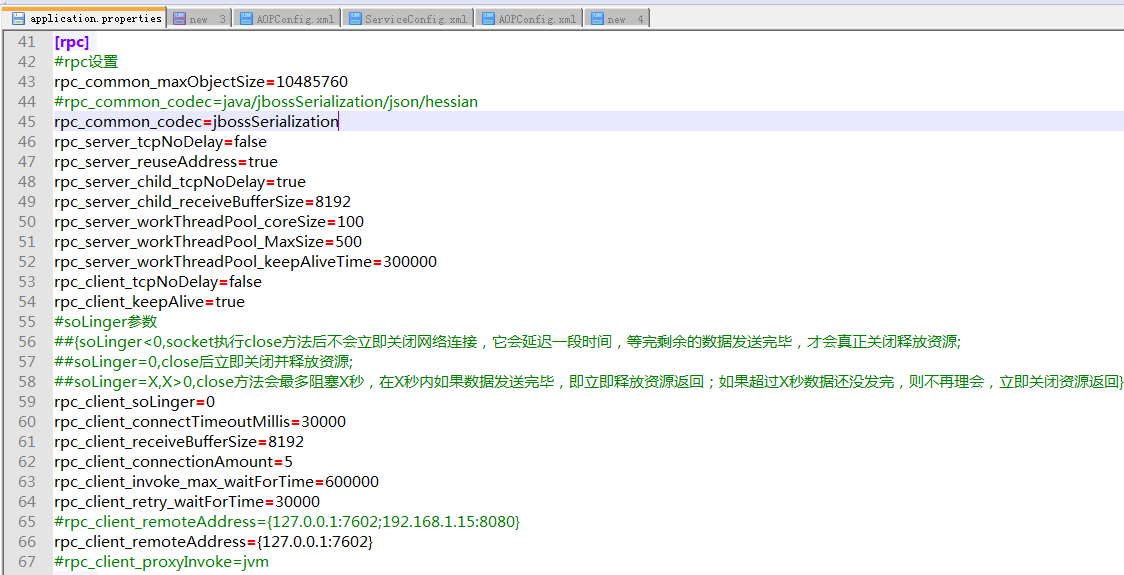


整个Service框架从概念上分为Client端和Server端两个部分。Client端暴露Service的接口，Server端是Service接口的具体实现；Client通过ServiceProxy来访问Server端。方式有两种：

* Local本地，即Proxy会在本地的JVM去获取Service实现实例。
* Distributed分布式，即Proxy会通过框架内置的RPC去远程的服务端获取Service实现实例。同时，支持通信的长连接和短连接，客户端支持负载均衡等特性。

本地和分布式方式通过application.properties应用配置文件设置“**rpc\_client\_proxyInvoke=jvm**”参数来完成，设置本参数即代表使用本地访问，注释掉本参数即代表使用分布式方式访问。

另外，RPC相关调优参数请参考application.properties的RPC设置部分。

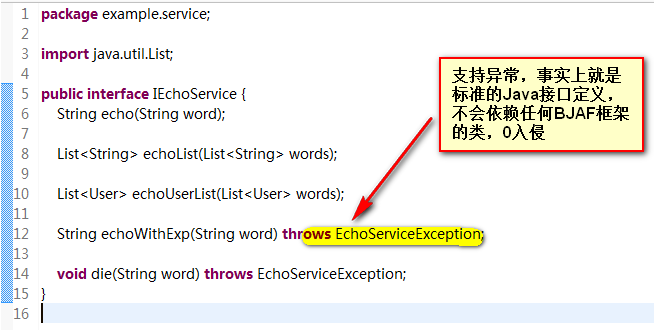


### 3.4.3使用示例

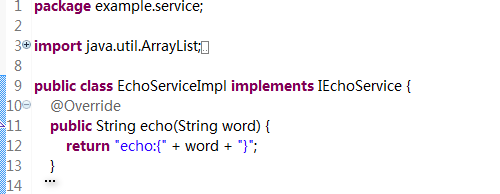
#### 3.4.3.1同步调用

例如，开发一个Echo的服务

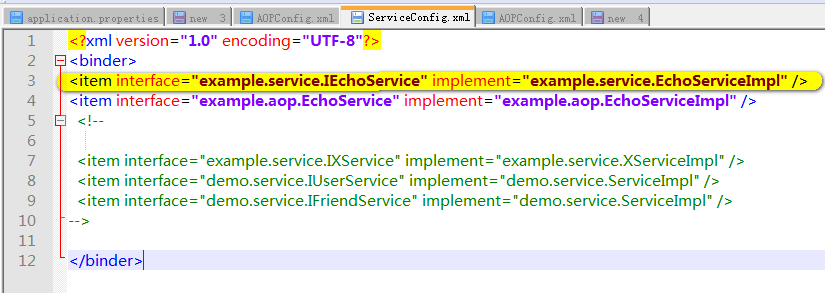
（1），定义接口IEchoService



（2），编写实现类



（3），在ServiceConfig.xml配置注册服务（3.x版本可以省略配置，参考前面章节说明）



（4），使用Proxy代理执行

**public** **static** **void** main(String[] args) {

IEchoService echoService = ServiceFactory

.*serviceLookup*(IEchoService.**class**);

String x = echoService.echo("xxxxx");

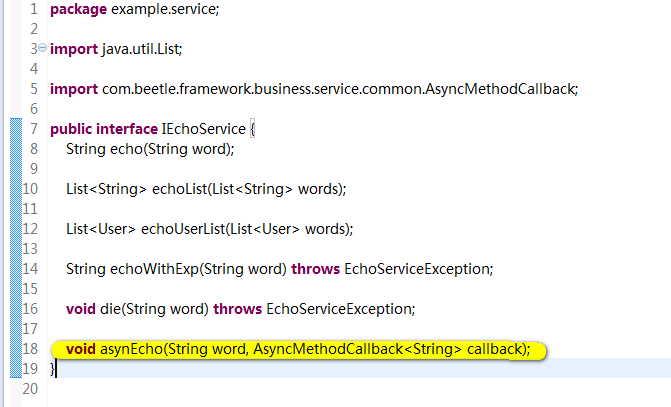
System.*out*.println(x);

}

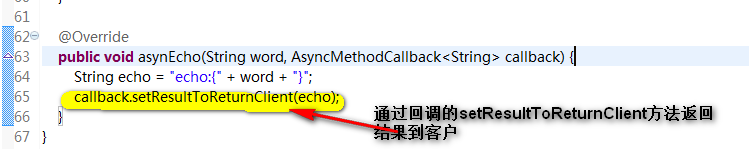
#### 3.4.3.2异步调用

异步调用就是主线程调用方法不会阻塞等待结果，调用后就马上返回，服务端计算的结果会传到本地后自动触发回调函数执行。例子如下：

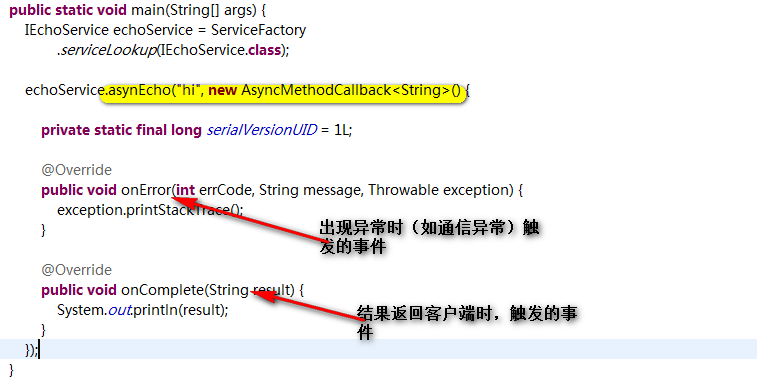
（1），定义回调方法



（2），编写实现



（3），编写客户端



81278576189468.jpg业务层是使用Command框架还是Service服务框架来开发？Command是面向于过程的，Service是面向于服务的。如果你的业务逻辑是相对封闭无需暴露在外部，给其它第3方来调用的可以考虑使用Command；同样，如果你的业务需要暴露给外部系统或者以后有可能暴露给外部系统，考虑到SOA模式扩展（如：引入ESB服务总线）则采取Service比较合适。

在BJAF框架中，Service的@ServiceTransaction注解是通过command框架提供的事务执行来实现的，当然这一点对开发人员来说是透明的。

# 4.Web表示层

Web表示层在J2EE应用中的角色可简单概括为：接收来自客户端的HTTP请求，将请求分发给业务层进行处理，将处理结果返回给客户端显示。在J2EE体系架构中，它属于Web Servlet 容器的技术范畴。BJAF在Web表示层实现了一个基于J2EE标准Servlet技术构建的请求驱动（Request Driven）的MVC快速开发框架。它具备以下功能特征：

* 标准MVC模式实现，明确的控制器、模型、视图界定。
* 结构清晰、配置简单、便于扩展和维护。
* 框架组件语义清晰、具备控制开发人员出现越界设计的能力。
* 不依赖Web容器，支持控制器单元测试。
* 控制器支持Session透明检查、http协议缓存、防止重复提交功能。
* 支持多种功能控制器：如：文件上传、动态页面绘图、验证码等等。
* 以HTML、JSP为标准视图，支持freemarker页面模板、PDF、Excel等视图。
* 提供Application和Session两个级别页面请求访问动态缓存功能，所有的缓存动作都是可以根据运行时需求而动态配置，无需更改任何代码。
* 支持Web Service技术。
* 支持当前流行的AJAX技术。
* 支持零配置（控制器和视图配置可选）
* 控制器与视图关系可视分析

BJAF Web框架包括以下几个主要组件：

**全局分发器（GlobalDispatchServlet）**：全局分发器是整个web应用的入口，所有客户端（如：浏览器）请求和响应都有它来分发处理。其本身为一个标准的多线程Servlet实现，为了提高响应性能，可以装配多个实例。

**请求缓存过滤器（ControllerCacheFilter）**：提供运行时页面请求动态缓存的能力。

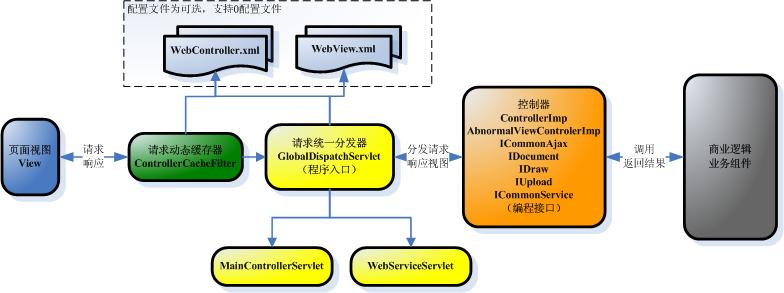
**控制器（Controller）：**处理由全局分发器分发的请求的具体控制逻辑。Web应用的开发主要是控制器逻辑的开发。

**数据绑定（DataBinder）：**负责将Web页面的输入元素封装成一个数据对象。在本框架中，数据绑定无须依赖任何配置与标签库，它的绑定对开发人员是透明的。

**数据校验（Validator）：**校验页面输入数据的合法性。我们推荐使用浏览器端javascript进行数据校验（如：使用jQuery的validate.js插件），而不是服务端用java代码进行；所以BJAF没有提供服务端验证框架的支持。

**网页视图（View）：**视图的职责是显示模型中数据，不应该直接处理请求，所以在一个视图中只能包含处理数据显示逻辑，这也是Web应用开发除控制器外的另一项重要工作。各个视图之间应该是独立的，不能耦合关系。

BJAF Web MVC框架处理Http请求基本流程及组件间的调用关系如下图所示：



BJAF Web框架对HTTP请求的处理基本流程

## 4.1Web框架配置

BJAF的 Web MVC框架中的全局分发器（GlobalDispatchServlet）和请求缓存过滤器（ControllerCacheFilter）都需要在Web容器的*web.xml*文件中装配才能使用。BJAF Web框架典型web.xml配置如下：

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

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app\_2\_3.dtd">

<web-app>

<display-name>webdemo</display-name>

<context-param>

<param-name>WEB\_ENCODE</param-name> [1]

<param-value>gb2312</param-value>

</context-param>

<context-param>

<param-name>WEB\_SERVICE\_SUFFIX</param-name> [2]

<param-value>service</param-value>

</context-param>

<context-param>

<param-name>CTRL\_PREFIX</param-name> [3]

<param-value>com.beetle.WebDemo</param-value>

</context-param>

<context-param>

<param-name>CTRL\_VIEW\_MAP\_ENABLED</param-name> [4]

<param-value>true</param-value>

</context-param>

<filter>

<filter-name>CacheFilter</filter-name> [5]

<filter-class>

com.beetle.framework.web.cache.ControllerCacheFilter

</filter-class>

</filter>

<filter-mapping>

<filter-name>CacheFilter</filter-name> [6]

<servlet-name>GlobalDispatchServlet</servlet-name>

</filter-mapping>

<servlet>

<servlet-name>GlobalDispatchServlet</servlet-name> [7]

<servlet-class>

com.beetle.framework.web.GlobalDispatchServlet

</servlet-class>

<load-on-startup>6</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>GlobalDispatchServlet</servlet-name> [8]

<url-pattern>\*.ctrl</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>GlobalDispatchServlet</servlet-name> [9]

<url-pattern>\*.upload</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>GlobalDispatchServlet</servlet-name> [10]

<url-pattern>\*.draw</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>GlobalDispatchServlet</servlet-name> [11]

<url-pattern>\*.service</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>GlobalDispatchServlet</servlet-name> [13]

<url-pattern>\*.dcmt</url-pattern>

</servlet-mapping>

<session-config>

<session-timeout>30</session-timeout> [14]

</session-config>

<mime-mapping>

<extension>jar</extension>

<mime-type>application/x-java-archive</mime-type>

</mime-mapping>

<mime-mapping>

<extension>jnlp</extension>

<mime-type>application/x-java-jnlp-file</mime-type>

</mime-mapping>

<error-page>

<exception-type>java.lang.Exception</exception-type> [15]

<location>/common/err.jsp</location>

</error-page>

</web-app>

[1] WEB\_ENCODE参数为指定Web应用请求处理的编码，如果不设置此参数，框架会以当前操作系统的默认编码处理。

[2] WEB\_SERVICE\_SUFFIX参数为WebService服务后缀名称，作用类似于[7]，默认为service。

[3]CTRL\_PREFIX参数代表控制器前缀，BJAF在1.3.x版本中实现零配置功能，主要实现思想是采取规定约定来代替通过配置文件寻找具体控制器实现。例如：某个控制器实现类为：

**com.beetle.WebDemo.presentation.zero. LoginController**

显然这个包路径过长，为了简约编写，我们可以把公共部分提出来，作为CTRL\_PREFIX常量，如：

**CTRL\_PREFIX=com.beetle.WebDemo**

零配置详细说明请参考零配置实现章节

[4]CTRL\_VIEW\_MAP\_ENABLED参数为是否启动运行时控制器与视图依赖关系记录分析功能。

[5]装配动态请求缓存过滤器，[6]指定过滤器的作用范围，这里配置的意思是：对所有提交给全局分发器（GlobalDispatchServlet）的请求都经过过滤器处理。用户也可以根据自己需要，合理地规划设置。

[7]装配全局分发器，本质上就是配置一个标准Servlet，命名为GlobalDispatchServlet，指定其实现类（com.beetle.framework.web.GlobalDispatchServlet），并指定其初始化的实例数量。

[8]指定应用下所有的后缀为：“\*.ctr”的请求控制器交给全局分发器GlobalDispatchServlet分发处理；“ctr”是框架默认的标准控制器后缀命名，相当于struts的“do”，当然，开发人员也可以根据自己需求重新定义此后缀。[9] [10] [11] [12] [13] 与[8]相类似分别定义上传、画图、Web服务、ajax、xls电子表格、pdf文档等各功能子控制器的请求都交由GlobalDispatchServlet处理。在此配置中，设置了整个Web应用（\*.\*）都由GlobalDispatchServlet处理，用户也可以根据实际需要加入请求路径进行合理划分，有关web.xml的详细解释请参考J2EE的web.xml规范。

[14]设置Session失效时间。

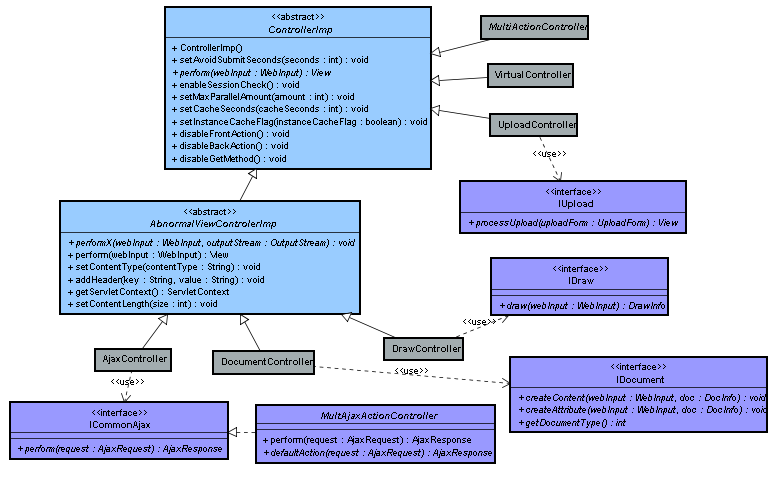
[15]指定Web应用错误处理（显示）视图页面。

## 4.2控制器

### 4.2.1控制器介绍

控制器是MVC设计模式的重要组成部分。控制器定义了应用系统中处理页面请求的逻辑动作，它负责解释用户的输入，调用业务对象进行处理并将其返回的结果数据转换成相应的数据模型，再通过视图展示给用户。

BJAF Web框架实现来一套灵活控制器编程机制，并针对不同情况功能需求，派生出一系列功能控制器，它们之间层次关系及编程接口UML关系类图如下：



控制器组件及编程接口关系类图

**ControllerImp**：为标准控制器，是控制器抽象基类（父类），所有功能控制器都由它派生而来，它适用于视图为：常见的html、jsp、freemarker模板等传统页面的开发情形。

**AbnormalViewControlerImp**：为非标准视图控制器。相对ControllerImp而言，它负责处理返回给客户的视图为：文件、流、图片等非html、jsp传统视图的开发情形。

**UploadController**：文件上传控制器，用于页面文件上传的情况。其编程接口是IUpload，需要进行页面文件上传，子控制器实现此接口完成控制逻辑就可以了。

**VirtualController**：虚拟控制器，它没有编程接口，它直接跳转到定义的视图中。适用于简单页面跳转逻辑情况，也常用于代替直接暴露视图后缀名，保持应用一贯风格的情况。

**~~AjaxController~~**~~：ajax控制器，扩展自非标准视图控制器。BJAF Web框架对ajax提供了深入的强而有力的支持，此控制器为BJAF Web Ajax框架的具体实现。ICommonAjax为其统一编程接口，所有的Ajax开发的子控制器都必须实现此接口。（注：BJAF2.x版本已经删除此控制器）~~

**DocumentController**：文档视图控制器，扩展自非标准视图控制器。相对页面视图而言，适用于不使用页面显示数据而采取文档格式返回的情况。支持：pdf、xls电子表格等文档格式。其对外编程接口为IDocument。

**DrawController**：页面绘图控制器，扩展自非标准视图控制器。适用于在页面上绘制数据统计图（如：饼图、曲线图、柱状图等等）的情况。其对外编程接口为IDraw。

MultiActionController:多动作控制器，可以用一个控制器处理多个页面提交的动作。

~~MultAjaxActionController：针对Ajax的多动作控制器，用一个控制器处理多个页面提交的动作~~。~~（注：BJAF2.x版本已经删除此控制器）~~

FacadeController：面板控制器，将框架支持的所有控制器类型都融合在一个统一的控制器中， 以方便编程。

### 4.2.2配置控制器

如前所述，GlobalDispatchServlet将web请求分发给相应的控制器处理，为了正确的分发请求，就必须告知分发器处理此请求的控制器具体的实现类在什么地方。这样就需要建立一个请求与控制器处理实现类映射关系，此映射关系在本框架体现为*WebController.xml*配置文件。其内容格式如下：

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

<mappings>

<!--控制器 【注：在<controllers>标签内，控制器的名称是唯一的，而且无需带路径】 -->

<controllers>

<!--横切动作 -->

<cutting>

<!--控制器横切动作<ctrlFrontAction>为前置点横切；<ctrlBackAction>为后置点横切 -->

<ctrlFrontAction>Example.GlobalPreActionImp</ctrlFrontAction>

<ctrlBackAction>Example.GlobalPreActionImp</ctrlBackAction>

</cutting>

<standard>

<sItem name="SearchController.ctrl" class="com.beetle.jIKBS.web.SearchController"/>

<!-- 添加更多的item... -->

</standard>

<virtual>

<vItem name="NoDataFound.ctrl" view="InputView"/>

<!-- 添加更多的item... -->

</virtual>

<upload>

<uItem name="Login.Upload" class="com.beetle.SmsCRM.presentation.LogicCtrl"/>

<!-- 添加更多的item... -->

</upload>

<drawing>

<dItem name="PieStatistics.ajax" class="com.beetle.jIKBS.web.StatisticsDraw"/>

<!-- 添加更多的item... -->

</drawing>

<document>

<docItem name="TestPdf.dcmt" class="Example.TestPdf"/>

<!-- 添加更多的item... -->

</document>

</controllers>

<!--启动开关 -->

<onoff>

<startUp>Example.GlobalPreCallImp</startUp>

<closeUp>Example.GlobalPreCallImp</closeUp>

</onoff>

<!--控制器缓存 -->

<caches>

<cItem name="ShowStatisticsController.ctrl" scope="application" time="60"/>

<cItem name="ShowContentController.ctrl" scope="session" time="30"/>

<!-- 添加更多的item... -->

</caches>

<!--Web服务 -->

<service>

<srvItem name="LoginWebService.service" class="com.beetle.WebDemo.presentation.LoginWebService" />

<!-- 添加更多的item... -->

</service>

<!--控制器模块【针对系统控制器<controllers>标签，如果系统控制器比较多，那么可以建立多个控制器模块】 -->

<module>

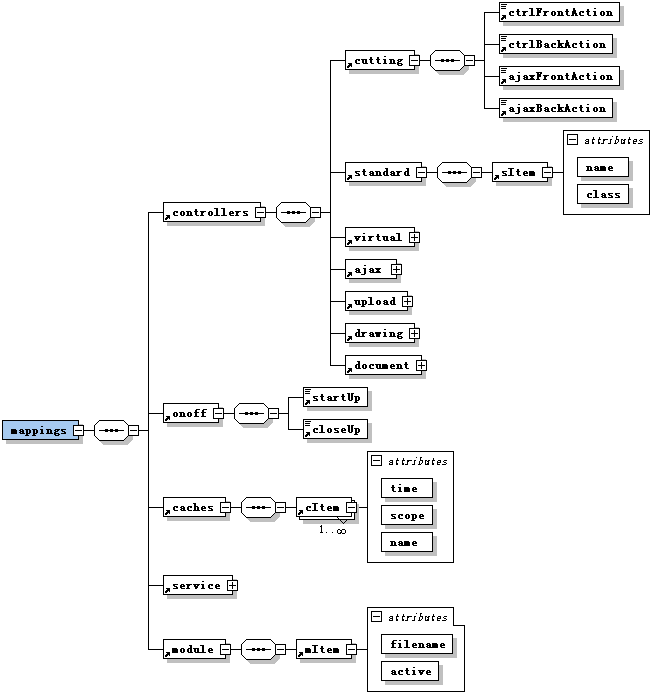
<mItem filename="xxx\_controller.xml" active="true"/>

<!-- 添加更多的item... -->

</module>

</mappings>

WebController.xml配置文件结构示意图如下：



WebController.xml配置文件结构图

如上所示，配置控制器很简单，只要在相应的类型的控制器标签下，添加其对应的项目即可。项目一般包括两个属性：

name：控制器的名称，与页面发起请求动作所设置的名称一致。（例如：页面form表单<FORM METHOD=POST ACTION="**SearchController.ctrl**">...</FORM>）**在BJAF Web框架中控制器采取实名制，即所有的控制器名称必须唯一**，在指定控制器的时候无须指定访问路径，消除了Web开发中常见绝对路径与相对路径容易混乱所带来的不便。

class：控制器实现类的名称，包含此类包路径。

另外，框架支持多控制器配置文件装配，如果一个Web应用中控制器数目过多，可以考虑分模块各自建立一个“xxx\_controller.xml”文件，格式与WebController.xml一致，然后在WebController.xml文件<module>标签内装配就可以了。注意的是自建立的模块文件要与WebController.xml文件放在统一目录下。

【注】：WebController.xml必须放置在Web应用根目录下***config***目录内，config目录为BJAF Web框架所依赖的固定目录。

81278576189468.jpg实际上，大多数请求驱动的MVC框架（例如Struts、Webwork、Spring MVC等），都是需要依靠“请求-控制器”映射关系来分发请求的。包括哪些所谓的零配置文件框架，它们只是把映射的关系用一种默认约定代替了。BJAF Web框架在1.3.x版本以后也可以通过约定来实现零配置；在是否采取配置的问题上，虽然BJAF支持零配置、文件配置和两者混合使用3种模式，但是，为了管理和维护的方便，笔者推荐采取统一文件配置模式。

### 4.2.3标准视图控制器

所有采取页面（html／jsp／freemarker）作为视图的请求动作都属于标准控制器开发范畴，*ControllerImp*是标准控制器编程统一入口，它为抽象类，它对外提供了一个简单的抽象方法：

/\*\*

\* 控制逻辑执行方法，系统框架主控制器（MainControllerServlet）会根据请求的url来找到此控制类，并执行此方法完成任务

\* @param webInput Web页面输入参数对象，对request对象封装，基本上保留request的方法，屏蔽到一些不利于开发的方法

\* @return 视图对象（视图的名称[WebView.xml]，以及相关的数据）

\* @throws ControllerException

\*/

**public abstract View perform(WebInput webInput) throws ControllerException;**

通常用户在写自己的控制器时，只需扩展ControllerImp抽象类在perform方法体内完成请求控制逻辑就可以了。

下面例子就是一个简单登录验证控制器实现：

**package** com.beetle.WebDemo.presentation;

**import** java.util.Date;

**import** javax.servlet.http.HttpSession;

**import** com.beetle.WebDemo.common.Const;

**import** com.beetle.WebDemo.common.LoginInfo;

**import** com.beetle.framework.web.controller.ControllerException;

**import** com.beetle.framework.web.controller.ControllerImp;

**import** com.beetle.framework.web.controller.WebInput;

**import** com.beetle.framework.web.view.ModelData;

**import** com.beetle.framework.web.view.View;

**public** **class** LoginController **extends** ControllerImp {

**public** LoginController() {

}

**public** View perform(WebInput webInput) **throws** ControllerException {

View view = **null**;

String userName = webInput.getParameter("username"); // 获取页面输入的参数

**int** password = webInput.getParameterAsInt("password");

**int** veiwFlag = webInput.getParameterAsInt("veiwFlag");// 为了视图演示的标记

// 调用业务对象处理业务逻辑，本示例在这里只是简单地作了一个字符串比较

**if** (userName.equals("HenryYu") && password == 888888) {

LoginInfo loginInfo = **new** LoginInfo();

loginInfo.setLoginUser(userName);

loginInfo.setPassword(password);

loginInfo.setLoginTime(**new** Date(System.*currentTimeMillis*()));

HttpSession session = webInput.getSession(**true**);// 创建会话，保存登录数据(操作Session演示)

session.setAttribute("LoginInfo", loginInfo);// 以便后用。。。

ModelData vd = **new** ModelData();

vd.put("Login\_Info", loginInfo);

**if** (veiwFlag == 0) {// 采取标准的JSP视图显示数据

view = **new** View("MainView", vd); // 返回MainView视图（连同LoginInfo数据对象一起返回）

} **else** **if** (veiwFlag == 1) {// 采取freemarker模板作为视图显示数据

view = **new** View("LginFtlView", vd);

}

} **else** {

ModelData vd = **new** ModelData();

vd.put(Const.*WEB\_FORWARD\_URL*, "login.html");

vd.put(Const.*WEB\_RETURN\_MSG*, "用户名不存在或者密码不正确，请重新输入，谢谢！");

view = **new** View("InfoView", vd); // 返回InfoView视图

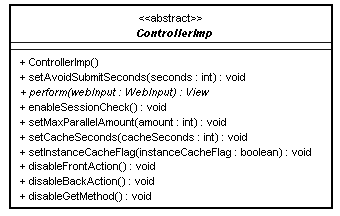
}

**return** view;

}

}

显然，标准控制器开发过程十分简单明了，从WebInput页面参数对象中获取页面传递过来的参数值，进行业务逻辑运算，获取的结果数据封装成ModelData，最后通过View视图对象返回。标准控制器编程抽象类ControllerImp类图如下：



ControllerImp类图

从上图可见，ControllerImp还具体下表方法、属性：

|  |  |
| --- | --- |
| **方法与属性** | **功能说明** |
| setAvoidSubmitSeconds(seconds:int):void [1] | 设置避免页面重复提交的秒数。框架支持防止页面重复提交的功能，在此秒数时间内，重复提交的请求无效。默认为0，即对提交不加以控制。 |
| disableFrontAction():void [2] | 禁止此控制器参与前置回调，前置横切动作对此控制器无效。控制器默认参与前置回调。 |
| disableBackAction():void[3] | 禁止此控制器参与后置回调，机制同上。 |
| setInstanceCacheFlag(instanceCacheFlag:boolean):void[4] | 设置此控制器是否需要缓存，默认，所有的控制器都被缓存；参数instanceCacheFlag为false，此控制器实例不被缓存，默认为true，如果你的控制器为线程不安全的，则设置为false。 |
| setCacheSeconds(cacheSeconds:int):void[5] | 设置浏览器客户端对视图内容缓存的秒数，参数应大于0；默认为不缓存。 |
| enableSessionCheck():void[6] | 启动框架对此控制器进行Session检查的功能，框架默认不对控制器进行Session检测。  如果启动了检查，那么当Session不存在时，则主控制器会不不处理此控制器，直接挑转到NoSessionView视图。 |
| setMaxParallelAmount(amount:int):void[7] | 设置此控制器最大支持并发请求数，默认为负数，即无限制。 此方法在对此控制器做并发控制时候，才需要设置，其它情况，框架不会对控制器进行任何并发数量限制。 |
| disableGetMethod() : void[8] | 禁止http的get方式的请求。此方法针对某些要求控制器只支持post方式的请求，禁止get方式，有利于防止那些通过地址栏输入参数发起请求的恶性攻击。 |
| perform(webInput:WebInput):View | 子控制器必须实现的抽象方法，子控制器在此方法体内完成请求的控制逻辑。  参数：  webInput--Web页面输入参数对象，对request对象封装，基本上保留request的方法，屏蔽到一些不利于开发的方法  返回：  视图对象（视图的名称[WebView.xml]，以及相关的数据） |
| [1] [2] [3] [4] [5] [6] [7]标记的方法必须在控制器构造函数内调用才有效，才起作用。 | |

### 4.2.4非标准视图控制器

非标准视图控制器是相对ControllerImp标准视图控制器而言的，它负责处理返回给客户的视图为：文件、流、图片等非html、jsp传统视图的开发情形。AbnormalViewControlerImp为其编程接口，AbnormalViewControlerImp继承自ControllerImp，也就是说非标准视图控制器是标准视图控制器的特别实现。

AbnormalViewControlerImp为抽象类，类似于ControllerImp对外提供一个简单的抽象方法：

public abstract void performX(WebInput webInput, OutputStream outputStream) throws ControllerException;

其参数：WebInput为Web页面输入参数对象；OutputStream为页面输出流，也就是说对于非标准视图内容的返回都是以流的形式响应给客户端浏览器。

例如，通过浏览器从服务器下载一个文件，可以利用非标准视图控制器来实现：

package com.beetle.WebDemo.presentation;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import com.beetle.framework.web.controller.AbnormalViewControlerImp;

import com.beetle.framework.web.controller.ControllerException;

import com.beetle.framework.web.controller.WebInput;

public class DownloadController extends AbnormalViewControlerImp {

public void performX(WebInput wi, OutputStream out)

throws ControllerException {

InputStream is = null;

try {

is = this.getServletContext()

.getResourceAsStream("images/logo.zip");

this.setContentType("application/octet-stream");

this.addHeader("Content-Disposition",

"attachment; filename=logo.zip");

byte[] buff = new byte[2048];

while (is.read(buff) != -1) {

out.write(buff);

}

} catch (Exception e) {

throw new ControllerException(e);

} finally {

try {

if (is != null) {

is.close();

}

out.flush();

out.close();

} catch (IOException e) {

}

}

}

}

DoucumentController、AjaxController、DrawController都是继承自这个非标准视图控制器的特例实现。

### 4.2.5虚拟控制器

在我们开发Web应用时候，经常遇到直接点击链接显示jsp视图（页面）的时候，而我们不想把这个jsp页面链接直接暴露给用户，这时候可以采取虚拟控制器来代替直接链接。

例如：

<A HREF="**/user/register.jsp**">注册表格</A>

使用虚拟控制器

<A HREF="**userRegister.ctrl**">注册表格</A>

在config目录下WebController.xml和WebView.xml分别注册一下控制器和视图的对应关系。

WebController.xml文件配置：

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

<mappings>

<controllers>

<virtual>

<vItem name="**userRegister.ctrl**" view="**registerView**"/>

</virtual>

<!-- ... -->

</mappings>

WebView.xml文件配置：

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

<mappings>

<views>

<standard>

<sItem name="**registerView**" url="**/user/register.jsp**" />

</standard>

<!-- ... -->

</views>

</mappings>

另外，虚拟控制器还适用于简单页面跳转逻辑而又不向编程一个标准控制器的情况。

### 4.2.6文件上传控制器

BJAF Web框架利用控制器技术封装了Apache Commons FileUpload 组件包提供了文件上传的支持。其编程接口是：

package com.beetle.framework.web.controller.upload;

import javax.servlet.\*;

import com.beetle.framework.web.controller.ControllerException;

import com.beetle.framework.web.view.\*;

public interface IUpload {

/\*\*

\* 上传文件的大小，单位为byte，默认为10M

\*/

long sizeMax = 10485760; //10M

int sizeThreshold = 4096; //4k

/\*\*

\* 执行上传

\*

\* @param uploadForm 上传的form参数对象

\* @return 返回视图对象

\* @throws ServletException

\*/

View processUpload(UploadForm uploadForm) throws ControllerException;

}

其参数UploadForm对象是页面上传<form>表单内容请求参数的封装，它提供了一系列的方法来获取上传文件的信息，详见下图：

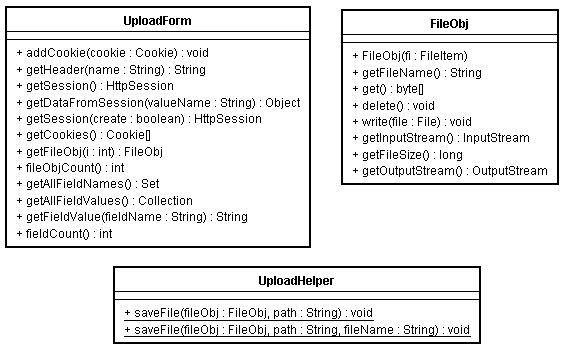


图3-5 UploadForm等组件类图

其基本处理过程为：通过getFileObj()方法获取上传文件FileObj对象，根据具体逻辑对此上传文件进行处理，例如，利用UploadHelper助手类把此文件对象保存在服务器磁盘上。

一个简单文件上传例子如下：

（1），编写上传页面

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=GBK">

<title>文件上传</title>

<link href="t-1.css" rel="stylesheet" type="text/css">

</head>

<body>

<form action="**FileUploadController.upload**" method="**post**" enctype="**multipart/form-data**" name="form1">[1]

<table width="60%" border="0" align="center">

<tr>

<td width="31%">

<div align="right">请选择要上传的文件</div>

</td>

<td width="69%">**<input type="file" name="file">**[2]</td>

</tr>

<tr>

<td>

<div align="right">新的文件名:</div>

</td>

<td><input name="fileName" type="text" id="fileName"> （不填为默认文件名）</td>

</tr>

<tr>

<td colspan="2">

<div align="center"><input type="submit" name="Submit" value="提交"></div>

</td>

</tr>

<tr>

<td colspan="2">

<div align="center">(注：上传的文件默认保存在系统的C盘的根目录下)</div>

</td>

</tr>

</table>

</form>

<div align="center"><a href="./index.html">返回</a></div>

</body>

</html>

[1]必须指明是一个multipart请求，而且method必须为post方法。

[2]通过一个file类型输入对话框选择指定要上传的文件。

（2），实现IUpload接口编写一个文件上传处理子控制器：

package com.beetle.WebDemo.presentation;

import com.beetle.WebDemo.common.Const;

import com.beetle.framework.log.AppLogger;

import com.beetle.framework.web.controller.ControllerException;

import com.beetle.framework.web.controller.upload.FileObj;

import com.beetle.framework.web.controller.upload.IUpload;

import com.beetle.framework.web.controller.upload.UploadForm;

import com.beetle.framework.web.controller.upload.UploadHelper;

import com.beetle.framework.web.view.ModelData;

import com.beetle.framework.web.view.View;

public class FileUploadController implements IUpload {

private static AppLogger logger = AppLogger.getInstance(FileUploadController.class);

public View processUpload(UploadForm upForm) throws ControllerException {

View view = null;

String newFileName = upForm.getFieldValue("fileName");

FileObj fileObj = upForm.getFileObj(0);//获取上传文件对象

if (logger.isDebugEnabled()) {

logger.debug(newFileName);

logger.debug(fileObj);

}

if (fileObj == null || fileObj.getFileName().equals("")) {

ModelData vd = new ModelData();

vd.put(Const.WEB\_FORWARD\_URL, "file.html");

vd.put(Const.WEB\_RETURN\_MSG, "文件输入不能为空，请重新输入，谢谢！");

view = new View("InfoView", vd);

}

else {

ModelData vd = new ModelData();

vd.put(Const.WEB\_FORWARD\_URL, "file.html");

StringBuffer sb = new StringBuffer();

sb.append(fileObj.getFileName());

if (newFileName != null && !newFileName.equals("")) {

sb.append("改成新的名字：" + newFileName);

UploadHelper.saveFile(fileObj, Const.UPLOAD\_FILE\_SAVE\_PATHE,

newFileName); //保存到服务器磁盘

}

else {

UploadHelper.saveFile(fileObj, Const.UPLOAD\_FILE\_SAVE\_PATHE);

}

sb.append("文件上传成功！");

sb.append("保存在服务器的位置为：");

sb.append(Const.UPLOAD\_FILE\_SAVE\_PATHE);

vd.put(Const.WEB\_RETURN\_MSG, sb.toString());

view = new View("InfoView", vd);

}

return view;

}

}

（3），最后在WebController.xml的<upload>标签中注册刚才编写的控制器即可：

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

<mappings>

<controllers>

<upload>

<uItem name="**FileUploadController.upload**" class="**com.beetle.WebDemo.presentation.FileUploadController**" />

</upload>

</controllers>

</mappings>

81278576189468.jpg文件控制器默认对文件上传大小最大限制是10MB，如果你需要修改此参数，可以在你的HTML的提交表单里面添加sizeMax参数，设置新的大小（单位为byte字节），如：

<INPUT TYPE="hidden" NAME="sizeMax" value="20971520">

把最大限制改成20MB。另外，还涉及sizeThreshold参数，此参数默认是4096，调整此参数方式与sizeMax是一样的。参数详细说明请参考：

<http://commons.apache.org/fileupload/>

上传组件主页。

### 4.2.7页面绘图控制器

在Web应用开发中，我们经常需要在页面动态绘制各种统计图，如：实时的股票行情，市场调研分析等等。这些图片数据需要我们在服务器内存中动态生成，然后通过HTTP协议传送到客户端，最终在浏览器上显示出来。

BJAF Web框架封装了著名的开源JfreeChart包，支持了页面动态绘制包括：饼图、柱状图(普通柱状 图以及堆栈柱状图)、线图、区域图、分布图、混合图、甘特图、仪表盘等统计图功能。

页面绘图控制器DrawController的编程接口是：

package com.beetle.framework.web.controller.draw;

import com.beetle.framework.web.controller.\*;

public interface IDraw {

/\*\*

\* 执行，画图

\* @param webInput 页面参数输入对象

\* @return DrawInfo－－返回画图属性信息对象

\*/

DrawInfo draw(WebInput webInput)throws ControllerException;

}

接口定义很明显，根据WebInput获取页面请求参数，调用jFreeChart组件生成一个DrawInfo交给框架处理，生成图形响应回浏览器端。可见，在这里DrawController控制器负责封装接受请求、分析数据、生成并返回图像格式数据流的整个页面绘图过程，而这些对开发人员是透明的，开发人员关注如何利用JfreeChart提供各种绘图功能完成业务绘图的需求，而不需关心图像在HTTP传输和输出的问题。

例如，在页面绘制一个Web框架市场占有率统计饼图，如下图：

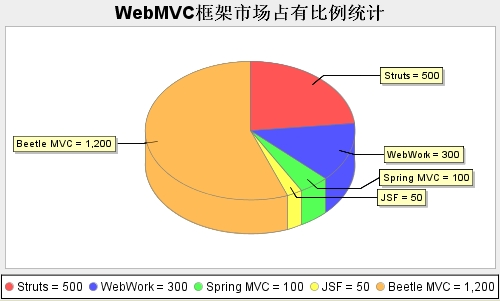


图3-8 Web框架统计饼图

绘图子控制器代码如下：

package com.beetle.WebDemo.presentation;

import org.jfree.chart.ChartFactory;

import org.jfree.chart.JFreeChart;

import org.jfree.data.general.DefaultPieDataset;

import com.beetle.framework.web.controller.WebInput;

import com.beetle.framework.web.controller.draw.DrawInfo;

import com.beetle.framework.web.controller.draw.IDraw;

public class DrawPieController implements IDraw {//实现IDraw接口

public DrawInfo draw(WebInput wi) throws ControllerException{

int width = wi.getParameterAsInt("width");//获取页面输入参数

int height = wi.getParameterAsInt("height");

DefaultPieDataset pie = new DefaultPieDataset();//调用jfreechart组件工作

pie.setValue("Struts", 500);

pie.setValue("WebWork", 300);

pie.setValue("Spring MVC", 100);

pie.setValue("JSF", 50);

pie.setValue("Beetle MVC", 1200);

JFreeChart chart = ChartFactory.createPieChart3D("WebMVC框架市场占有比例统计",

pie, true, false, false);

return new DrawInfo(100, chart, width, height, null);//返回画图信息对象

}

}

html页面通过<img>标签输出图像：

<img src="**DemoDrawController.draw?width=500&height=300**">

同时，在WebController.xml文件<drawing>标签中注册此绘图子控制器：

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

<mappings>

<controllers>

<drawing>

<dItem name="**DemoDrawController.draw**"

class="**com.beetle.WebDemo.presentation.DrawPieController**" />

<!-- ... -->

</drawing>

</controllers>

</mappings>

关于jfreechart组件的使用说明，请参考其官方网站（<http://www.jfree.org/jfreechart/index.php>）。

81278576189468.jpg实际上，由于web浏览器和javascript技术的发展，现在比较流行的方案是在浏览器端使用Javascript来绘制统计图，目前成熟的Javascript方案很多，例如flot等，如果采取这样的方案只要利于我们的WebServiceController控制器配合使用即可。

### 4.2.8文档视图控制器

有些时候，客户要求服务器处理Web请求结果数据以pdf、excel等文档视图返回。为此，BJAF Web框架实现了一个文档视图控制器，目前支持pdf、excel两种文件格式的生成返回。文档视图控制器DocumentController也是扩展自AbnormalViewControlerImp非标准视图控制器抽象类的一个特例实现，其编程接口为：

package com.beetle.framework.web.controller.document;

import com.beetle.framework.web.controller.\*;

public interface IDocument {

public static final int TYPE\_PDF = 10;

public static final int TYPE\_MS\_EXCEL = 11;

public static final int TYPE\_MS\_WORD = 12;

/\*\*

\* 建立文档内容

\* @param webInput 页面参数输入对象

\* @param doc 文档信息对象

\*/

void createContent(WebInput webInput, DocInfo doc)throws ControllerException;

/\*\*

\* 建立文档属性信息(例如：Author/Title/CreationDate)

\* @param webInput 页面参数输入对象

\* @param doc 文档信息对象

\*/

void createAttribute(WebInput webInput, DocInfo doc)throws ControllerException;

/\*\*

\* 获取此控制器生成文档的类型

\* @return[TYPE\_PDF\TYPE\_MS\_EXCEL\TYPE\_MS\_WORD...]

\*/

int getDocumentType();

}

例如，生成一个pdf返回的简单例子如下：

package com.beetle.WebDemo.presentation;

import com.beetle.framework.web.controller.WebInput;

import com.beetle.framework.web.controller.ControllerException;

import com.beetle.framework.web.controller.document.DocInfo;

import com.beetle.framework.web.controller.document.IDocument;

import com.lowagie.text.Document;

import com.lowagie.text.DocumentException;

import com.lowagie.text.Font;

import com.lowagie.text.Paragraph;

import com.lowagie.text.pdf.BaseFont;

public class GenPdfController implements IDocument {

public void createAttribute(WebInput wi, DocInfo di) throws ControllerException{//建立pdf文档属性信息

String auther = wi.getParameter("auther");//获取web输入参数

Document pdfDoc = di.getPdfDocument();//从DocInfo对象获取一个pdf文档对象

pdfDoc.addAuthor(auther);

}

public void createContent(WebInput wi, DocInfo di) throws ControllerException {

Document pdfDoc = di.getPdfDocument();//从DocInfo对象获取一个pdf文档对象

try {

pdfDoc.add(new Paragraph("Hello World!"));//构建文档内容

try {

BaseFont bf = BaseFont.createFont("STSong-Light",

"UniGB-UCS2-H", BaseFont.NOT\_EMBEDDED);

Font FontChinese = new Font(bf, 12, Font.NORMAL);

String info=wi.getParameter("info");

Paragraph p0 = new Paragraph(info, FontChinese);

pdfDoc.add(p0);

Paragraph p1 = new Paragraph("Beetle Web Framework 页面生成PDF文件演示!", FontChinese);

pdfDoc.add(p1);

} catch (Exception ex1) {

throw new ControllerException(ex1);

}

} catch (DocumentException ex) {

throw new ControllerException(ex);

}

}

public int getDocumentType() {

return IDocument.TYPE\_PDF;

}

}

注：BJAF Web框架利用itext.jar包生成pdf文件，所以开发子控制器也依赖与itext.jar包。

还需把这个GenPdfController子控制器注册在WebController.xml的<document>标签下：

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

<mappings>

<controllers>

<document>

<docItem name="GenPdfController.dcmt"

class="com.beetle.WebDemo.presentation.GenPdfController" />

<docItem name="GenExcelController.dcmt"

class="com.beetle.WebDemo.presentation.GenExcelController" />

</document>

</controllers>

</mappings>

类似，生成一个Excel文件的子控制器代码如下：

package com.beetle.WebDemo.presentation;

import org.apache.poi.hssf.usermodel.HSSFCell;

import org.apache.poi.hssf.usermodel.HSSFRichTextString;

import org.apache.poi.hssf.usermodel.HSSFRow;

import org.apache.poi.hssf.usermodel.HSSFSheet;

import org.apache.poi.hssf.usermodel.HSSFWorkbook;

import com.beetle.framework.web.controller.ControllerException;

import com.beetle.framework.web.controller.WebInput;

import com.beetle.framework.web.controller.document.DocInfo;

import com.beetle.framework.web.controller.document.IDocument;

public class GenExcelController implements IDocument {

public void createAttribute(WebInput wi, DocInfo di) throws ControllerException{

// ...

}

public void createContent(WebInput wi, DocInfo di)throws ControllerException {

HSSFWorkbook wb = di.getExcelDocument();

// 创建HSSFSheet对象

HSSFSheet sheet = wb.createSheet("sheet0");

// 创建HSSFRow对象

HSSFRow row = sheet.createRow((short) 0);

// 创建HSSFCell对象

HSSFCell cell = row.createCell((short) 0);

// 用来处理中文问题

// cell.setEncoding(HSSFCell.ENCODING\_UTF\_16);

// 设置单元格的值

// cell.setCellValue("Hello World! 你好，中文世界");

String info = wi.getParameter("info");

HSSFRichTextString rts = new HSSFRichTextString(info);

cell.setCellValue(rts);

HSSFCell cell2 = row.createCell((short) 1);

cell2.setCellValue(new HSSFRichTextString(

"Beetle Web Framework 页面生成PDF文件演示!"));

}

public int getDocumentType() {

return IDocument.TYPE\_MS\_EXCEL;

}

}

注：框架依赖于Apache的poi包来生成Excel文件。

### 4.2.9多动作控制器

在一般情况下，我们在一个页面中，一个动作会对应一个控制器，假设这个页面要处理10个动作，那么就需要建立10控制器，虽然从技术上来说没有什么，但是从管理角度来考量，肯定是不合理的。我们能否从设计层面上做到，一“类”或“组”动作，专门由一个控制器处理？答案是肯定的，BJAF在1.4.0版本推出MultiActionController多动作控制器就是专门针对这类场景的。

例如：我们从一个控制器来处理系统用户登陆和登出两个动作，代码如下：

**package** com.beetle.WebDemo.presentation;

**import** java.util.Date;

**import** javax.servlet.http.HttpSession;

**import** com.beetle.WebDemo.common.Const;

**import** com.beetle.WebDemo.common.LoginInfo;

**import** com.beetle.framework.web.controller.ControllerException;

**import** com.beetle.framework.web.controller.MultiActionController;

**import** com.beetle.framework.web.controller.WebInput;

**import** com.beetle.framework.web.view.ModelData;

**import** com.beetle.framework.web.view.View;

**public** **class** LoginAndOutController **extends** MultiActionController {

**public** View defaultAction(WebInput arg0) **throws** ControllerException {

// **TODO** Auto-generated method stub

**return** **null**;

}

**public** View loginAction(WebInput webInput) **throws** ControllerException {

View view = **null**;

String userName = webInput.getParameter("username");

**int** password = webInput.getParameterAsInt("password");

**if** (userName.equals("HenryYu") && password == 888888) {

LoginInfo loginInfo = **new** LoginInfo();

loginInfo.setLoginUser(userName);

loginInfo.setPassword(password);

loginInfo.setLoginTime(**new** Date(System.*currentTimeMillis*()));

HttpSession session = webInput.getSession(**true**);// 创建会话，保存登录数据(操作Session演示)

session.setAttribute("LoginInfo", loginInfo);// 以便后用。。。

ModelData vd = **new** ModelData();

vd.put("Login\_Info", loginInfo);

view = **new** View("LoginoutView", vd); // 返回MainView视图（连同LoginInfo数据对象一起返回）

} **else** {

ModelData vd = **new** ModelData();

vd.put(Const.*WEB\_FORWARD\_URL*, "multiActionController.html");

vd.put(Const.*WEB\_RETURN\_MSG*, "用户名不存在或者密码不正确，请重新输入，谢谢！");

view = **new** View("InfoView", vd); // 返回InfoView视图

}

**return** view;

}

**public** View logoutAction(WebInput wi) **throws** ControllerException {

HttpSession session = wi.getSession(**false**);

session.removeAttribute("LoginInfo");

ModelData vd = **new** ModelData();

vd.put(Const.*WEB\_FORWARD\_URL*, "multiActionController.html");

vd.put(Const.*WEB\_RETURN\_MSG*, "成功退出！");

**return** **new** View("InfoView", vd);

}

}

其对应的视图页面代码为：

multiActionController.html

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=GBK">

<title></title>

<link href="t-1.css" rel="stylesheet" type="text/css">

</head>

<body>

<form name="form1" method="post" action="LoginAndOutController.ctrl">

<table width="34%" border="0" align="center">

<tr>

<td><div align="right">用户名:</div></td>

<td><input name="username" type="text" id="username" value="HenryYu"></td>

</tr>

<tr>

<td><div align="right">密 码:</div></td>

<td><input name="password" type="text" id="password" value="888888"></td>

</tr>

<tr>

<td colspan="2"><div align="center">

<INPUT TYPE="hidden" NAME="$action" value="loginAction">

<input type="submit" name="Submit" value="提交">

</div></td>

</tr>

</table>

</form>

</body>

</html>

loginoutView.jsp

<%@page contentType="text/html; charset=gb2312"%>

<%@page session="false"%>

<%@page import="com.beetle.framework.web.view.ViewHelper"%>

<%@page import="com.beetle.WebDemo.common.\*"%>

<html>

<head>

<title> </title>

<meta http-equiv="Content-Type" content="text/html; charset=gb2312">

<link href="t-1.css" rel="stylesheet" type="text/css">

</head>

<body>

<!-- 方式1 -->

<%

ViewHelper helper=new ViewHelper(request);

LoginInfo loginInfo=(LoginInfo)helper.getDataValue("Login\_Info");

%>

<table width="75%" border="1" align="center">

<tr>

<td width="20%">登陆用户名：</td>

<td width="80%"><%=loginInfo.getLoginUser() %></td>

</tr>

<tr>

<td>密码：</td>

<td><%=loginInfo.getPassword() %></td>

</tr>

<tr>

<td>登陆时间：</td>

<td><%=loginInfo.getLoginTime() %></td>

</tr>

<tr>

<td>...</td>

<td>

<A HREF="LoginAndOutController.ctrl?$action=logoutAction">退出系统</A>

</td>

</tr>

</table><BR><BR>

<p align="center"><a href="../index.html">返回</a></p>

</body>

</html>

从上面例子代码可知，每一个动作对应一个自定义的方法。

方法定义满足以下原则：

1--在符合java规范条件下，随便定义。

2--方法的输入参数只能是WebInput，返回类型必须是View

3--方法必须抛出ControllerException异常

4--方法必须使用public修饰，

eg:public View xxxAction(WebInput webInput) throws ControllerException

注意：在页面提交表单中，必须用'$action'关键字指定具体的方法名称。

eg:<INPUT TYPE="hidden" NAME="$action" value="xxxAction">

若'$action'不设置，则会执行defaultAction方法

另外，Ajax编程也是支持多动作控制器（MultAjaxActionController）的，Java端开发模式和约定与传统的多动作控制器是一样的，Html页面端，使用ajax的客户端，采取诸如：

var req=new Request();

req.setControllerName("$job.CreateJobAjaxController.ajax");

req.put("$action","checkCronAction");

req.put("cronExpression",$.trim($('#tab5\_worktime').val()));

var r=req.synchroExecute();

if(r.returnFlag<0){

alert(r.returnMsg);

return;

}

这里就不足赘述了，详细请参考开发包的例子。

### 4.2.10 Web Service控制器

Web Service控制器（WebServiceController）是专门针对返回的是非视图（如：jsp页面），返回是对象数据。数据的格式支持Json和XML数据，以及这两种数据格式的透明转换。同样，也支持多动作。同时，也支持restful访问的形式。另外，BJAF2.x版本删除原来的Ajax控制器，对于Ajax支持，改用Web Service控制器来支持，Ajax前端Javascript，我们推荐使用jQuery。

例子如下：

package com**.**beetle**.**WebDemo**.**presentation**.**webservice**;**

**import** java**.**util**.**ArrayList**;**

**import** com**.**beetle**.**framework**.**web**.**controller**.**ControllerException**;**

**import** com**.**beetle**.**framework**.**web**.**controller**.**WebInput**;**

**import** com**.**beetle**.**framework**.**web**.**controller**.**WebServiceController**;**

**import** com**.**beetle**.**framework**.**web**.**view**.**ModelData**;**

public class Hello **extends** WebServiceController **{**

public ModelData defaultAction**(**WebInput wi**)** **throws** ControllerException **{**

ModelData md **=** **new** ModelData**();**

md**.**put**(**"path"**,** wi**.**getServletPath**());**

md**.**put**(**"name"**,** "Henry"**);**

md**.**put**(**"session"**,** wi**.**getSession**());**

ArrayList myList **=** **new** ArrayList**();**

myList**.**add**(**"aaaa"**);**

myList**.**add**(new** java**.**sql**.**Timestamp**(**System**.**currentTimeMillis**()));**

md**.**put**(**"myList"**,** myList**);**

**return** md**.**asXML**(); //转换成XML格式返回**

**}**

public ModelData jsonTestService**(**WebInput wi**)** **throws** ControllerException **{**

ModelData md **=** **new** ModelData**();**

md**.**put**(**"path"**,** wi**.**getServletPath**());**

md**.**put**(**"name"**,** "Henry"**);**

md**.**put**(**"phone"**,** wi**.**getParameter**(**"phone"**));**

md**.**put**(**"session"**,** wi**.**getSession**());**

ArrayList myList **=** **new** ArrayList**();**

myList**.**add**(**"aaaa"**);**

myList**.**add**(new** java**.**sql**.**Timestamp**(**System**.**currentTimeMillis**()));**

md**.**put**(**"myList"**,** myList**);**

**return** md**.**asJSON**();//转换成json格式返回**

**}**

public ModelData testAction(WebInput wi) throws ControllerException {

ModelData md = new ModelData();

md.put("path", wi.getServletPath());

md.put("name", "Henry");

md.put("session", wi.getSession());

ArrayList myList = new ArrayList();

myList.add("aaaa");

myList.add(new java.sql.Timestamp(System.currentTimeMillis()));

md.put("myList", myList);

return md;// 无需显式表明返回数据类型，由web.xml的WEB\_SERVICE\_DATA\_DEFAULT\_FORMAT参数决定

}

**}**

详细请参考SDK中的WebDemo例子。另外，BJAF还提供了一个Web Service客户端，com.beetle.framework.web.client. RestClient，其示例代码如下：

RestClient rc **=** **new** RestClient**(**"henry"**,** "123456"**);**

rc**.**connect**();**

**try** **{**

RestRequest rreq **=** **new** RestRequest**(** "http://127.0.0.1:9090/webdemo/service/presentation/webservice/BeanDataDemoService/"**,** RestRequest**.**InvokeMethod**.**GET**);**

rreq**.**addParameter**(**"$dataFormat"**,** "json"**);**

rreq**.**addParameter**(**"name"**,** "余浩东"**);**

rreq**.**addParameter**(**"home"**,** "shenzhen,china"**);**

RestResponse rres **=** rc**.**invoke**(**rreq**);**

**if** **(**rres**.**getStatusCode**()** **==** 200**)** **{**

System**.**out**.**println**(**rres**.**getContent**());**

**}** **else** **{**

//...

**}**

**}** **catch** **(**RestInvokeException e**)** **{**

e**.**printStackTrace**();**

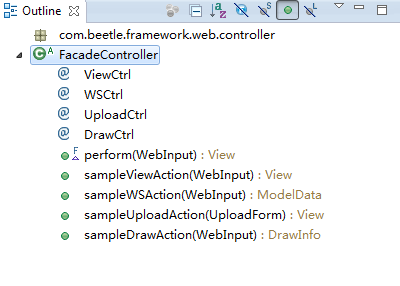
**}** **finally** **{**

rc**.**close**();**

**}**

### 4.2.11面板控制器

面板控制器，是在多动作控制器的基础上，将框架支持的所有控制器类型都融合在一个统一的控制器中， 统一控制器编程接口，以简化开发。其类图如下：



从上图可知，Façade控制器包括4个注解，分别是：

* ViewCtrl，视图控制，标注这个方法（或动作）返回视图，是标准视图控制器的实现。
* WSCtrl，Web Service视图控制器。
* UploadCtrl，标注这个动作是文件上传的控制器。
* DrawCtrl，标注这个动作是后台绘图控制器。

同时，面板控制台还提供了针对上面4个注解的Sample方法，在具体开发过程中，如果不熟悉，可以先重载相应的Sample方法再改动作名称。例子如下：

package com**.**beetle**.**dwzdemo**.**web**.**user**;**

**import** org**.**apache**.**catalina**.**User**;**

**import** com**.**beetle**.**dwzdemo**.**business**.**IUserManagerService**;**

**import** com**.**beetle**.**dwzdemo**.**valueobject**.**XxUser**;**

**import** com**.**beetle**.**dwzdemo**.**web**.**common**.**WsStatus**;**

**import** com**.**beetle**.**framework**.**log**.**AppLogger**;**

**import** com**.**beetle**.**framework**.**resource**.**define**.**PageList**;**

**import** com**.**beetle**.**framework**.**util**.**encrypt**.**AesEncrypt**;**

**import** com**.**beetle**.**framework**.**web**.**controller**.**ControllerException**;**

**import** com**.**beetle**.**framework**.**web**.**controller**.**FacadeController**;**

**import** com**.**beetle**.**framework**.**web**.**controller**.**WebInput**;**

**import** com**.**beetle**.**framework**.**web**.**view**.**ModelData**;**

**import** com**.**beetle**.**framework**.**web**.**view**.**View**;**

public class UserManagerController **extends** FacadeController **{**

private static final AppLogger logger **=** AppLogger

**.**getInstance**(**UserManagerController**.**class**);**

public UserManagerController**()** **{**

**super();**

**this.**enableSessionCheck**();**// 验证用户是否登录

**}**

@ViewCtrl

public View pageUserAction**(**WebInput wi**)** **throws** ControllerException **{**

// 分页显示用户数据

int pagesize **=** wi**.**getParameterAsInteger**(**"numPerPage"**,** 20**);**

int pagenum **=** wi**.**getParameterAsInteger**(**"pageNum"**,** 1**);**

String orderField **=** wi**.**getParameter**(**"orderField"**,** "userid"**);**

String orderArith **=** wi**.**getParameter**(**"orderDirection"**,** "desc"**);**

logger**.**debug**(**"orderField:{}"**,** orderField**);**

logger**.**debug**(**"orderArith:{}"**,** orderArith**);**

IUserManagerService umSrvc **=** **this**

**.**serviceLookup**(**IUserManagerService**.**class**);**

PageList**<**XxUser**>** pl **=** umSrvc**.**showAllUserByPage**(**pagenum**,** pagesize**,**

orderField**,** orderArith**);**

ModelData md **=** **new** ModelData**();**

md**.**put**(**"userPageList"**,** pl**);**

md**.**put**(**"orderField"**,** orderField**);**

md**.**put**(**"orderDirection"**,** orderArith**);**

View view **=** **new** View**(**"/userListview.jsp"**,** md**);**

**return** view**;**

**}**

@ViewCtrl

public View nameSearchAction**(**WebInput wi**)** **throws** ControllerException **{**

String username **=** wi**.**getParameter**(**"username"**);**

int pagesize **=** wi**.**getParameterAsInteger**(**"numPerPage"**,** 1000**);**

int pagenum **=** wi**.**getParameterAsInteger**(**"pageNum"**,** 1**);**

String orderField **=** wi**.**getParameter**(**"orderField"**,** "userid"**);**

String orderArith **=** wi**.**getParameter**(**"orderDirection"**,** "desc"**);**

logger**.**debug**(**"orderField:{}"**,** orderField**);**

logger**.**debug**(**"orderArith:{}"**,** orderArith**);**

IUserManagerService umSrvc **=** **this**

**.**serviceLookup**(**IUserManagerService**.**class**);**

PageList**<**XxUser**>** pl **=** umSrvc**.**searchByName**(**username**,** pagenum**,** pagesize**,**

orderField**,** orderArith**);**

ModelData md **=** **new** ModelData**();**

md**.**put**(**"userPageList"**,** pl**);**

md**.**put**(**"orderField"**,** orderField**);**

md**.**put**(**"orderDirection"**,** orderArith**);**

View view **=** **new** View**(**"/userListview.jsp"**,** md**);**

**return** view**;**

**}**

@ViewCtrl

public View findUserAction**(**WebInput wi**)** **throws** ControllerException **{**

long userid **=** wi**.**getParameterAsLong**(**"uid"**);**

IUserManagerService umSrvc **=** **this**

**.**serviceLookup**(**IUserManagerService**.**class**);**

XxUser user **=** umSrvc**.**findUser**(**userid**);**

**if** **(**user **==** **null)** **{**

**throw** **new** ControllerException**(**"用户不存在"**);**

**}**

ModelData md **=** **new** ModelData**();**

md**.**put**(**"user"**,** user**);**

View view **=** **new** View**(**"/updateUser.jsp"**,** md**);**

**return** view**;**

**}**

private String getForwardUrl**(**WebInput wi**)** **{**

// $user.UserManagerController.ctrl?$action=pageUserAction&pageNum=1&numPerPage=20

StringBuilder sb **=** **new** StringBuilder**();**

sb**.**append**(**wi**.**getControllerName**()).**append**(**"?"**)**

**.**append**(**"$action=pageUserAction&pageNum=1&numPerPage=20"**);**

**return** sb**.**toString**();**

**}**

@WSCtrl

public ModelData updateUserAction**(**WebInput wi**)** **throws** ControllerException **{**

ModelData md **=** **new** ModelData**();**

WsStatus wss **=** **new** WsStatus**();**

**try** **{**

IUserManagerService umSrvc **=** **this**

**.**serviceLookup**(**IUserManagerService**.**class**);**

XxUser user **=** wi**.**getParameterValuesAsFormBean**(**XxUser**.**class**);**

XxUser userDB **=** umSrvc**.**findUser**(**user**.**getUserid**());**

**if** **(**user**.**getPasswd**()** **==** **null**

**||** user**.**getPasswd**().**trim**().**length**()** **==** 0**)** **{**

user**.**setPasswd**(**userDB**.**getPasswd**());**

**}** **else** **{**

user**.**setPasswd**(**AesEncrypt**.**encrypt**(**user**.**getPasswd**()));**

**}**

umSrvc**.**updateUser**(**user**);**

// {"statusCode":"200", "message":"操作成功", "navTabId":"navNewsLi",

// "forwardUrl":"", "callbackType":"closeCurrent"}

// {"statusCode":"300", "message":"操作失败"}

wss**.**setStatusCode**(**200**);**

wss**.**setMessage**(**"更新用户成功！"**);**

wss**.**setNavTabId**(**"userListview"**);**

wss**.**setForwardUrl**(**getForwardUrl**(**wi**));**

wss**.**setCallbackType**(**"forward"**);**

**}** **catch** **(**Exception e**)** **{**

logger**.**error**(**e**);**

wss**.**setStatusCode**(**300**);**

wss**.**setMessage**(**"更新用户失败！"**);**

**}**

md**.**setData**(**wss**);**

**return** md**.**asJSON**();**

**}**

@WSCtrl

public ModelData delUserAction**(**WebInput wi**)** **throws** ControllerException **{**

ModelData md **=** **new** ModelData**();**

WsStatus wss **=** **new** WsStatus**();**

**try** **{**

IUserManagerService umSrvc **=** **this**

**.**serviceLookup**(**IUserManagerService**.**class**);**

int r **=** umSrvc**.**deleteUser**(**wi**.**getParameterAsLong**(**"uid"**));**

wss**.**setStatusCode**(**200**);**

**if** **(**r **>** 0**)** **{**

wss**.**setMessage**(**"删除用户成功！"**);**

**}** **else** **{**

wss**.**setMessage**(**"此用户不存在，无法执行删除动作！"**);**

**}**

wss**.**setNavTabId**(**"userListview"**);**

wss**.**setForwardUrl**(**getForwardUrl**(**wi**));**

wss**.**setCallbackType**(**"forward"**);**

**}** **catch** **(**Exception e**)** **{**

logger**.**error**(**e**);**

wss**.**setStatusCode**(**300**);**

wss**.**setMessage**(**"删除用户失败！"**);**

**}**

md**.**setData**(**wss**);**

**return** md**.**asJSON**();**

**}**

@WSCtrl

public ModelData addUserAction**(**WebInput wi**)** **throws** ControllerException **{**

ModelData md **=** **new** ModelData**();**

WsStatus wss **=** **new** WsStatus**();**

**try** **{**

XxUser user **=** wi**.**getParameterValuesAsFormBean**(**XxUser**.**class**);**

user**.**setPasswd**(**AesEncrypt**.**encrypt**(**user**.**getPasswd**()));**

logger**.**debug**(**"user:{}"**,** user**);**

IUserManagerService umSrvc **=** **this**

**.**serviceLookup**(**IUserManagerService**.**class**);**

umSrvc**.**createUser**(**user**);**

// {"statusCode":"200", "message":"操作成功", "navTabId":"navNewsLi",

// "forwardUrl":"", "callbackType":"closeCurrent"}

// {"statusCode":"300", "message":"操作失败"}

wss**.**setStatusCode**(**200**);**

wss**.**setMessage**(**"添加用户成功！"**);**

wss**.**setNavTabId**(**"userListview"**);**

wss**.**setForwardUrl**(**getForwardUrl**(**wi**));**

wss**.**setCallbackType**(**"forward"**);**

**}** **catch** **(**Exception e**)** **{**

logger**.**error**(**e**);**

wss**.**setStatusCode**(**300**);**

wss**.**setMessage**(**"创建用户失败！"**);**

**}**

md**.**setData**(**wss**);**

**return** md**.**asJSON**();**

**}**

**}**

详细请参考SDK开发包的DWZ的例子。

## 4.3视图显示

### 4.3.1标准JSP视图

当前可利用的视图技术很多，各有特点。然而，BJAF Web框架从技术成熟性、视图的性能和项目团队的技能等因素考量，框架把JSP作为标准主要的视图来使用。

JSP从98年发展到现在，已经十分成熟，容易接受，而且支持开发工具多；从性能来考量，JSP在众多的视图技术测试中，性能优秀排名前列；特别地，在几乎所有的Web应用中，页面布局和美化工作都是有非技术的美工来负责，JSP作为以HTML为基础的最为简单的视图，美工更是驾轻就熟；而且，JSP也是J2EE规范中标准的视图技术。

虽然JSP技术为Web表现层技术提供了灵活、丰富的功能支持，但是过于凌乱的JSP Scriplet也成为系统维护的头号大敌。JSP代码中若是业务逻辑、数据逻辑、表现逻辑代码混杂，那么代码重用性、可维护性都极低，为系统的开发带来极大的隐患。所以，JSP的使用应遵循以下原则：

JSP只能作为视图来使用

JSP功能强大，它几乎允许在一个页面里面编写任何逻辑。如访问数据库、业务逻辑代码等等。然而，从设计角度看，这是绝对不允许的，JSP仅仅只能作为显示模型数据的视图使用，只能拥有数据显示逻辑。

JSP不应该具备访问Session的能力

为了不破坏设计，视图所有的数据都应该统一由模型来提供。我们应该使用**<%@page session=”false”%>**语句关闭JSP使用Session功能，而且能提高性能。操纵Session是控制器而不是视图的权利。

JSP不应该使用request对象直接访问请求参数

处理请求参数是控制器责任，视图不能越权。

JSP应避免异常处理

没有必要在一个页面中使用try/catch，因为视图运行时不应该遇到可恢复性的错误。

不应该在一个JSP页面中，定义内部类、函数等等

理论上视图只处理显示逻辑，不太可能有太复杂逻辑需要用到内部类。即使有可能，也应该抽象成助手类、通用函数，放在java的包中，而不应该定义在一个JSP文件内部。

不应该使用out.println()语句

这只会给页面美化人员带来不必要的麻烦，没有任何好处。

尽量不要使用自定义标签库

如果没有充分的理由，不要实现自己的标签库。即使有需要，应该首先考虑JSTL标签库。（虽然笔者一直是各种标签库的反对者，但是对JSTL却是一个例外，因为合理使用JSTL可以使得页面变得简单，关键是JSTL很简单易懂，是一个J2EE支持标准技术。JSTL可以作为JSP视图辅助工具来使用。）

### 4.3.2视图配置

控制器需要根据视图的名称来定位视图页面在服务器上存放位置，这就需要建立一个视图名称与位置路径的映射关系。框架通过*WebView.xml*配置文件来定义此映射关系，它与WebController.xml一起存放在Web应用Config目录下。其内容格式如下：

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

<mappings>

<!-- 视图的名称是唯一的，url是视图文件在Web服务器上存放位置 -->

<views>

<!-- Session失效时，请求会自动跳转到此标签定义的视图 -->

<DisabledSessionView>IndexView</DisabledSessionView>

<!-- 应用错误统一处理视图，当系统发生异常自动跳转到此标签定义的视图 -->

<ErrorView>ErrView</ErrorView>

<standard>

<sItem name="IndexView" url="/index.html"/>

<sItem name="MainView" url="/views/main.jsp"/>

<sItem name="ErrView" url="/errdeal/errView.jsp"/>

<sItem name="ContentView" url="/views/content.jsp"/>

<sItem name="InputView" url="/views/input.jsp"/>

<!-- 添加更多的Item ...-->

</standard>

<freemarker>

<fItem name="wapInputView" url="/views/wap\_input.ftl"/>

<!-- 添加更多的Item ...-->

</freemarker>

</views>

<!--视图模块【针对视图<views>标签，如果系统视图数量比较多，那么可以建立多个视图模块】 -->

<module>

<mItem filename="xxx\_view.xml" active="true"/>

<!-- 添加更多的Item ...-->

</module>

</mappings>

其文件结构如下下图所示：

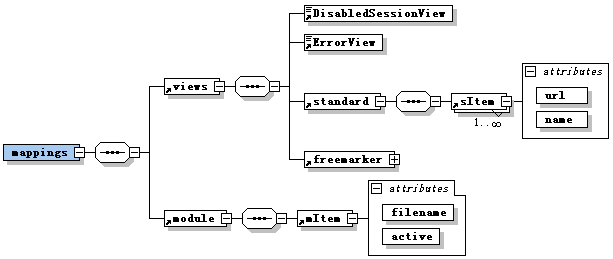


图3-9 WebView.xml文件结构

可见，配置视图是简单，在其对应的功能标签中定义其名称与其对应的url路径即可。

81278576189468.jpg值得注意的是，在一个Web应用中，**视图的命名是唯一的**，如果配置文件出现多个名称相同的定义，则框架会以最后面一个定义为准；建议视图名称定义采取一定命名规则（如 ：[模块名]\_[意思名称]）以防止视图过多导致含糊不清的情况；如果视图过多，建议按功能模块建立多个视图文件存放；另外，视图不会隶属于某个控制器的，多个控制器可以重用（引用）一个视图定义。

BJAF在1.3.x以后版本支持视图零配置，也就是说可以不依赖于WebView.xml配置文件来装配视图，只要在编程的返回视图的时候，**直接返回视图的具体物理路径和文件名**就可以了，如：

view = **new** View("/views/main.jsp", vd);

为了便于管理和维护，我们推荐使用WebView.xml配置文件来装配视图。

### 4.3.3标准视图Model数据解析

在Web MVC框架中，Model运算产生的结果数据ModelData最终会传递到视图页面中，然后通过解析成html再显示给客户端。

BJAF Web框架对页面Model数据解析支持以下两种方式：

**使用标准JSTL标签库方式**

JSTL（JavaServer Pages Standard Tag Library）是Java标准标签库，可以使用JSTL1.0或更高版本进行ModelData解析。JSTL1.0主要包括以下四类标签。

|  |  |  |  |
| --- | --- | --- | --- |
| **功能** | **URI** | **前缀** | **举例** |
| 核心功能 | *http://java.sun.com/jstl/core* | c | <c:tarname ...> |
| I18N格式化 | *http://java.sun.com/jstl/xml* | fmt | <fmt:tagname ...> |
| 处理XML | *http://java.sun.com/jstl/fmt* | x | <x:tagname ...> |
| 数据库操作 | *http://java.sun.com/jstl/sql* | sql | <sql:tagname...> |

在BJAF Web应用中，常用的是**<c>**和**<fmt>**这两类标签，由于BJAF的数据库操作必须通过持久成来完成，所以数据库操作的<sql>标签不推荐使用的。

为了使用JSTL标签库，要在JSP页面中引入标签定义，例如列出了常用的JSTL标签：

<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>

常用JSTL标签：

|  |  |
| --- | --- |
| **标签** | **作用** |
| <c:out> | 输出一个变量 |
| <c:if> | 判断一个变量 |
| <c:forEach> | 循环遍历一个数组或collection类型的变量 |
| <c:url> | 输出相对于应用context的URL |
| <fmt:message> | 格式化一个消息字符串 |
| <fmt:formatDate> | 格式化一个日期 |

关于JSTL的详细用法，请参考以下资料：<http://java.sun.com/products/jsp/jstl/>

例如，前面标准控制器例子LoginController简单登录验证控制器，返回ModelData包含一个用户登录信息LoginInfo对象，在页面中用JSTL解析展示如下：

<%@page contentType="text/html; charset=gb2312"%>

<%@page session="false"%>

<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>

<html>

<head>

<title></title>

<meta http-equiv="Content-Type" content="text/html; charset=gb2312">

<link href="./t-1.css" rel="stylesheet" type="text/css">

</head>

<body>

<table width="75%" border="1" align="center">

<tr>

<td width="20%">登陆用户名：</td>

<td width="80%"><c:out value="${**Login\_Info.loginUser**}"/></td>

</tr>

<tr>

<td>密码：</td>

<td><c:out value="${**Login\_Info.password**}"/></td>

</tr>

<tr>

<td>登陆时间：</td>

<td><c:out value="${**Login\_Info.loginTime**}"/></td>

</tr>

</table>

<p align="center"><a href="./index.html">返回</a></p>

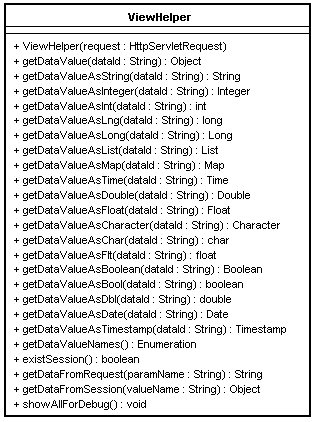
</body>

</html>

可见，采取JSTL结合EL表达式来解析Model数据是很简洁的。

**使用框架ViewHelper助手类方式**

JSTL虽然简单，但很多时候在处理页面逻辑的时候不够灵活，特别对于习惯了java编码方式的开发人员来说，一切都没有手动编码来得舒畅。对此，框架提供了ViewHelper视图助手类来方便Model数据在页面的获取和解析。ViewHelper类图如下：



ViewHelper类图

可见，ViewHelper助手类提供了一系列getDataValueAsXXX方法来简化ModelData处理，例如，用ViewHelper改写上面JSTL的例子，代码如下：

<%@page contentType="text/html; charset=gb2312"%>

<%@page session="false"%>

<!-- 引入ViewHelper类 -->

**<%@page import="com.beetle.framework.web.view.ViewHelper"%>**

**<%@page import="com.beetle.WebDemo.common.\*"%>**

<html>

<head>

<title></title>

<meta http-equiv="Content-Type" content="text/html; charset=gb2312">

<link href="./t-1.css" rel="stylesheet" type="text/css">

</head>

<body>

<%

ViewHelper helper=new ViewHelper(request); //创建一个ViewHelper实例

LoginInfo loginInfo=(LoginInfo)helper.getDataValue("Login\_Info"); //获取LoginInfo登录信息对象

%>

<table width="75%" border="1" align="center">

<tr>

<td width="20%">登陆用户名：</td>

<td width="80%"><%=loginInfo.getLoginUser() %></td>

</tr>

<tr>

<td>密码：</td>

<td><%=loginInfo.getPassword() %></td>

</tr>

<tr>

<td>登陆时间：</td>

<td><%=loginInfo.getLoginTime() %></td>

</tr>

</table>

<p align="center"><a href="./index.html">返回</a></p>

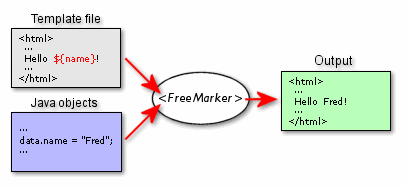
</body>

</html>

81278576189468.jpg笔者自己习惯采取ViewHelper而不是JSTL标签来操作视图，因为这样可以把java语言应用到视图上而不用再学习其它EL表达式，语言上一脉相承；虽然可能带来视图出现逻辑代码不够简洁情况，但是这些逻辑应该是视图展示逻辑，还是可以接受的。

### 4.3.4 freemarker模板支持

freemarker是一个十分优秀的html模板引擎，它功能强大而且简单易用。由于它以html为模板，这与JSP本质是一致的，所以它可作为JSP视图的替代方案，也可以作为JSP的辅助工具与JSP结合起来继续视图开发。freemarker的工作机制如下图所示：



freemarker工作机制

BJAF Web框架对freemarker进行透明封装，让freemarker模块与框架有机结合起来。开发好的视图freemarker模板主要在WebView.xml文件的<freemarker>注册一下就可以了。如：

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

<mappings>

<views>

<freemarker>

<fItem name="**LginFtlView**" url="**/views/template/LoginView.ftl**"/>

<!-- ... -->

</freemarker>

</views>

</mappings>

例如，我们采取freemarker模块来代替前面LoginController简单登录验证控制器JSP视图的开发，LoginView.ftl模板代码如下：（控制器代码无须改变）

<html>

<head>

<title>freemarker模板视图</title>

<link href="/views/t-1.css" rel="stylesheet" type="text/css">

</head>

<body>

<CENTER><B>#使用freemarker模板视图</B></CENTER><BR>

<table width="75%" border="1" align="center">

<tr>

<td width="20%">登陆用户名：</td>

<td width="80%">${Login\_Info.loginUser}</td>

</tr>

<tr>

<td>密码：</td>

<td>${Login\_Info.password}</td>

</tr>

<tr>

<td>登陆时间：</td>

<td>${Login\_Info.loginTime?string("yyyy-MM-dd HH:mm:ss zzzz")}</td>

</tr>

</table>

<p align="center"><a href="/index.html">返回</a></p>

</body>

</html>

freemarker模板提供了十分强大功能，其详细介绍，请参数<http://freemarker.sourceforge.net/>官方主页。

## 4.4数据绑定与校验

BJAF Web框架提供了强大的数据绑定和数据校验功能，使得传统繁杂的页面输入及校验功能变得简单明了。

### 4.4.1页面数据绑定

页面数据绑定是指把web页面提交的参数转换为一个数据值对象的操作。传统的数据绑定是通过手工完成的，典型代码如下：

LoginInfo loginInfo = new LoginInfo();

loginInfo.setLoginUser(webInput.getParameter("userName"));

loginInfo.setPassword(webInput.getParameterAsInt("password"));

...

当页面输入参数（或对象属性）数量较多的时候，逐个抽取参数再填充对象属性的代码就会冗长繁琐。为此，BJAF Web 框架提供的数据绑定的功能方法，能够自动将参数赋予指定的对象，将用户从手工绑定的工作中解放出来。上面的代码在BJAF的实现中，变成下面的一行。

//自动数据绑定

LoginInfo loginInfo=(LoginInfo)webInput.**getParameterValuesAsFormBean**(LoginInfo.class);

可见，通过WebInput参数对象提供的getParameterValuesAsFormBean方法相对其它同类型的框架利用JSP标签库来的轻量、易用和简单。

当然，没有免费的午餐，要成功地进行数据绑定，必须**遵循页面<form>表单<input>输入框参数名称与Java 值对象的属性名称一致（大小写不敏感）**的约定。

### 4.4.2页面数据校验

对于页面输入数据校验，BJAF框架采取浏览器端使用javascript进行数据输入有效性校验的设计策略，不提供服务端校验机制。因为服务端进行输入数据校验显得过于笨重繁琐，相对在页面进行校验则简洁、交互性好。而且，经过多年Web开发，相信很多开发人员都积累了丰富js函数库；事实上，在页面进行数据校验，已经是一个默认事实编程原则了。

BJAF Web 框架提供了一个页面<form>表单校验框架，封装在*formValidator.js*库文件中，下面简单介绍一下其使用：

（1），在html页面<head>标签内引入formValidator.js库：

<script language="JavaScript" src="**/config/formValidator.js**" type="text/JavaScript">

</script>

（2），在<form>表单内定义<input>输入框属性：

其定义格式为：

<input type="text" name="mail" **required**="1" **pattern**="EMAIL" **message**="xxxxxx">

其实就是在标准的<input>参数输入框标签内，扩展了isReq、regex、message三个功能属性，它们含义见下表：

|  |  |
| --- | --- |
| **属性** | **含义说明** |
| required | 是否检查为必填框，值为“1”，则必须输入，非“1”或不显性注明此属性，则不作检查。 |
| pattern | 检查模式，支持模式有：  NUMBER—输入只能为数字  EMAIL—邮件合法性检查  PHONE—电话号码合法性检查  POSTCODE—邮政编码合法性检查  CURRENCY—货币／浮点数合法性检查  DATE—日期格式检查[YYYY-MM-DD]  TIME—时间格式检查[HH:MM:SS]  DATETIME—日期+时间格式检查[YYYY-MM-DD HH:MM:SS]  regex—自定义正则表达式，以便扩展 |
| message | 检查不通过，信息说明。若不显性注明此属性，则框架会提示默认信息。 |

例如：

<form name="form1" method="post" action="RegisterEmpController.ctrl" >

<table width="34%" border="0" align="center">

<tr>

<td><div align="right">empNo</div></td>

<td>**<input type="text" name="empNo" required=1 pattern='NUMBER' message="请输入数字">**</td>

</tr>

<tr>

<td><div align="right">ename</div></td>

<td>**<input type="text" name="ename" required=1 >**</td>

</tr>

<tr>

<td><div align="right">phone</div></td>

<td>**<input type="text" name="phone" pattern='PHONE' >**</td>

</tr>

<tr>

<td><div align="right">job</div></td>

<td>**<input name="job" type="text" value="">**</td>

</tr>

<tr>

<td><div align="right">hireDate</div></td>

<td>**<input name="hireDate" type="text" required=1 pattern='DATETIME' value="">**(yyyy-mm-dd hh:mm:ss)</td>

</tr>

<tr>

<td><div align="right">sal</div></td>

<td>**<input type="text" name="sal" required=1 pattern='CURRENCY'>**</td>

</tr>

<tr>

<td><div align="right">email</div></td>

<td>**<input type="text" name="email" pattern='EMAIL' message="请输入一个合法邮件地址">**</td>

</tr>

<tr>

<td colspan="2"><div align="center">

<input type="button" name="Submit" value="提交" onclick="return submitForm()">

</div></td>

</tr>

</table>

</form>

（3），在提交动作事件中调用框架formValidate函数校验此<form>表单：

<SCRIPT LANGUAGE="JavaScript">

<!--

function submitForm(){

var f=**formValidate**(form1);

if(!f){

return;

}

form1.submit();

}

//-->

</SCRIPT>

当然，开发人员可以修改formValidate页面校验库添加自己pattern或者直接自定义正则表达式进行功能扩展。

现在页面开发基本上离不开jQuery，对应页面校验，笔者在这里推荐jquery.validate.js开发包。

## 4.5请求动态缓存功能

BJAF Web框架利用Servlet的Filter技术在请求和具体执行业务控制器之间加多一层，这是一个缓存策略管理层。它可以根据请求的特点来动态决定是否缓存，以及缓存内容及时返归回用户。其结构示意图如下：

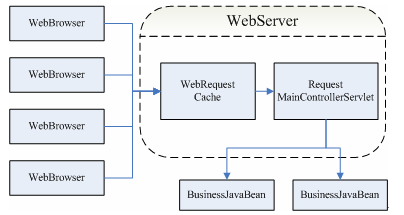


图3-16 请求动态缓存原理示意图

可见，我们在WebBrowser请求到达Request MainControllerServlet（请求主控制器的Servlet）之间，我们增加多一层：Web Request Cache，此缓存组件模块负责的具体工作如下：

缓存策略的定义。即：缓存的作用领域是多宽。是针对整个Web应用（Application级）的缓存还是只针对某个用户的会话（Session级）的缓存。它们的缓存时间又是多少？

缓存控制器的配置及读取。在软件的开发过程中，我们有时很难界定哪些控制输出需要作缓存处理，即使知道，我们编写的代码也只能是硬代码，很难根据以后具体的运行情况而变动。所以，所有的缓存处理必须是透明的、可以根据具体的需求而灵活配置的。哪些请求结果需要作缓存，其缓存策略如何，完全由配置文件来描述。

BJAF Web框架的请求缓存描述都在WebController.xml的**<caches>**标签内装配，例如：

<?xml version="1.0" encoding="gb2312"?>

<mappings>

<caches>

<cItem name="shop-viewCategory.ctrl" scope="application" time="30"/>

<cItem name="shop-viewProduct.ctrl" scope="application" time="30"/>

<cItem name="shop-listOrders.ctrl" scope="session" time="30"/>

<cItem name="shop-viewOrder.ctrl" scope="session" time="30"/>

</caches>

</mappings>

上面每一项为一条缓存策略的定义，例如第一项（条）表示为：

名称为：“shop-viewCategory.ctrl”的控制器输出的结果视图内容需要作缓存处理；其缓存的作用域是整个应用（Application），缓存时间的周期是30秒。

81278576189468.jpg动态缓存是BJAF Web框架又一个功能亮点，为Web应用的性能优化提供了一个优秀方案。但是值得注意是，缓存换来请求响应性能的提高的同时它也是占用内存的，如果不加分析什么都缓的话，那很容易造成OutOfMemory的内存溢出错误，特别是在session这一级别上应用。

所以，在准备采取缓存策略进行优化的时候，要在系统内存与请求性能之间找一个平衡，合理的设计才是解决问题的根本。

## 4.6AOP横切编程

AOP是一种编程技术，更是一种编程思想。支持对请求-处理过程进行横切编程有时候会给我们开发带来很大的灵活性和扩展性。例如：横切常见的应用场景是权限控制和操作日志。

在J2EE Web容器的编程中，我们一般可采取Servlet Filter技术对一个请求进行过滤来实现AOP横切编程。然而，对于普通开发人员来说Filter技术还是过于低级。BJAF Web框架在现有控制器技术的基础上，有机地封装了AOP编程思想在请求流程中设置了前、后两个回调函数，实现了所有请求子控制器的前、后置两个横切点的操作。

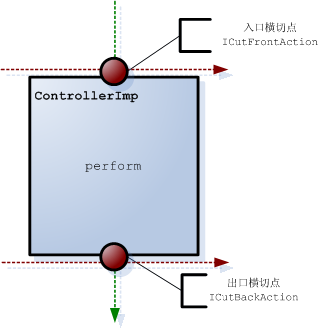


图3-17 控制器横切编程示意图

前置横切点ICutFrontAction接口定义如下：

package com.beetle.framework.web.controller;

import javax.servlet.\*;

import com.beetle.framework.web.view.\*;

public interface ICutFrontAction {

/\*\*

\* 执行全局前置回调动作

\*

\* @param webInput WebInput

\* @return 返回一个扩展View视图

\* @throws ServletException

\*/

View act(WebInput webInput) throws ServletException;

}

act动作方法其输入与输出都与标准控制器ControllerImp的perform方法是一样的。当然，流程不满足条件需要中断时，返回一个View视图告诉客户端中断的信息；当流程没必要中断，则View为null（即act方法返回为null即可）。后置横切点ICutBackAction的定义与前置接口机制一样。

对当前所有的控制器添加一个前置（或后置）动作开发过程如下：编写一个ICutFrontAction的实现类，然后在WebController.xml文件的**<cutting>**标签中注册，如：

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

<mappings>

<controllers>

<cutting>

<**ctrlFrontAction**>

com.beetle.WebDemo.presentation.aop.ControllerFrontAction

</**ctrlFrontAction**>

<**ctrlBackAction**>

com.beetle.WebDemo.presentation.aop.ControllerBackAction

</**ctrlBackAction**>

<**ajaxFrontAction**>Example.GlobalPreActionImp</**ajaxFrontAction**>

<**ajaxBackAction**>Example.GlobalPreActionImp</**ajaxBackAction**>

</cutting>

</controllers>

</mappings>

注意：一个Web应用中只能注册一个前置（或后置）的横切动作。

例如，我们建立一个前置动作检查用户必须为“HenryYu”才能允许浏览市场统计图信息。编写一个ICutFrontAction接口ControllerFrontAction类，代码如下：

package com.beetle.WebDemo.presentation.aop;

import javax.servlet.ServletException;

import javax.servlet.http.HttpSession;

import com.beetle.WebDemo.common.Const;

import com.beetle.WebDemo.common.LoginInfo;

import com.beetle.framework.web.controller.\*;

import com.beetle.framework.web.view.ModelData;

import com.beetle.framework.web.view.View;

public class ControllerFrontAction implements ICutFrontAction {

public View act(WebInput wi) throws ServletException {

String ctrlName = wi.getControllerName();

if (ctrlName.equalsIgnoreCase("DemoDraw2Controller.draw")) {

HttpSession session = wi.getSession(false);

if (session == null) {

return errView();

} else {

LoginInfo info = (LoginInfo) session.getAttribute("LoginInfo");

if (!info.getLoginUser().equals("HenryYu")) {

return errView();

}

}

}

return null;

}

private View errView() {

ModelData vd = new ModelData();

vd.put(Const.WEB\_FORWARD\_URL, "aop.html");

vd.put(Const.WEB\_RETURN\_MSG, "游客不能进行此操作，必须先登录，谢谢！");

View view = new View("InfoView", vd);

return view;

}

}

运行效果如下：

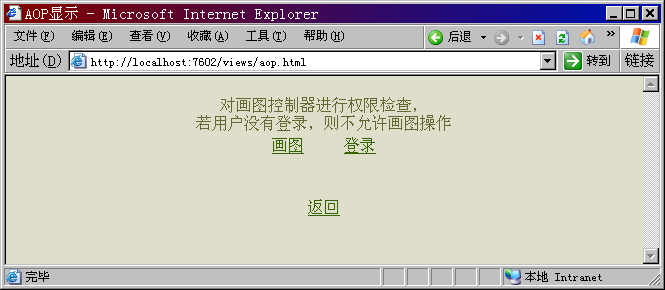


图3-18 AOP测试页面1

直接点击画图链接，由于预先没有登录，横切动作会返回一个提示视图：

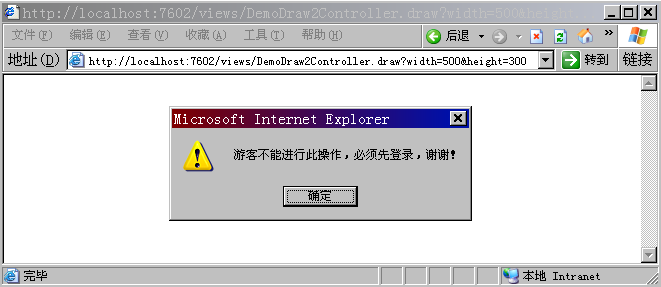


图3-19 测试页面2

用户登录后，再点击画图链接，就可以看到统计图像信息了。

同样，如果我们需要对所有的控制器作一个日志记录器，编写一个ICutBackAction后置接口实现类，再注册一下就可以了。

package com.beetle.WebDemo.presentation.aop;

import javax.servlet.ServletException;

import com.beetle.framework.web.controller.\*;

import com.beetle.framework.web.view.\*;

public class ControllerBackAction implements ICutBackAction{

public View act(WebInput wi) throws ServletException {

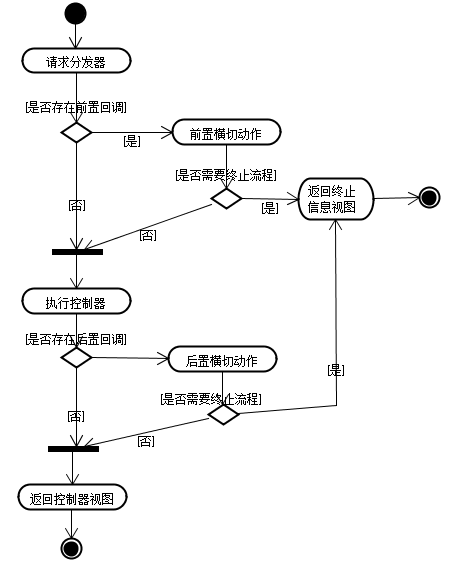
System.out.println("log->["+wi.getControllerName()+"]...");

return null;

}

}

BJAF Web框架处理请求的活动图如下：



框架处理请求活动图

可见，在整个流程中前、后置动作都是可选的，它们作用范围是整个流程，并不是作用于单个控制器；如果想对单个控制器进行横切，需要在动作中手工编程；另外，如果某个控制器想不参与前、后的横切动作，可以通过disableFrontAction()和disableBackAction()方法来取消；以返回视图的形式来终止流程。

## 4.7其它功能与特性

### 4.7.1 Web应用启动/关闭接口支持

有时候，我们需要在一个Web应用刚启动的时候，进行一个Web应用的数据初始化工作。例如，把一些常量性数据装载进内存等等；在一个Web应用销毁时进行一些资源回收或保存日志等工作。虽然Servlet规范中提供了javax.servlet. ServletContextListener接口可以解决上面问题，但是BJAF Web框架作为一个Web 应用框架解决方案，不推荐开发人员使用Servlet规范中低级的API接口，在本框架下Web应用所有的请求编程入口是控制器，为了框架的完整性和易用性，BJAF Web定义了启动和关闭Web应用两个接口，方便开发人员进行Web应用初始化和资源回收的工作。

Web启动接口IStartUp的定义如下：

package com.beetle.framework.web.onoff;

public interface IStartUp {

void startUp();

}

值得注意的是IStartUp接口依赖于框架Globaldispatchservlet总指派servlet，需要在Web.xml配置文件<load-on-startup>标签配置初始化线程数（至少大于1）才能工作。例如：

<servlet>

<servlet-name>Globaldispatchservlet</servlet-name>

<servlet-class>com.beetle.framework.web.GlobalDispatchServlet</servlet-class>

<load-on-startup>3</load-on-startup>

</servlet>

接口的实现类需要在WebController.xml文件的<onoff>标签内注册后才能使用。

Web应用关闭接口定义如下：

package com.beetle.framework.web.onoff;

public interface ICloseUp {

void closeUp();

}

一个简单例子如下：

package com.beetle.WebDemo.presentation.onoff;

import com.beetle.framework.web.onoff.\*;

public class WebAppOnOff implements IStartUp,ICloseUp {

public void startUp() {

System.out.println("初始化。。。。");

//...

}

public void closeUp() {

System.out.println("回收资源。。。");

//...

}

}

WebController.xml文件注册内容如下：

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

<mappings>

<**onoff**>

<**startUp**>com.beetle.WebDemo.presentation.onoff.WebAppOnOff</**startUp**>

<**closeUp**>com.beetle.WebDemo.presentation.onoff.WebAppOnOff</**closeUp**>

</**onoff**>

</mappings>

### 4.7.2防止表单重复提交

表单重复提交指的是，在输入数据并提交表单后，通过刷新浏览器或者回退页面再次提交表单，表单数据被重复处理的情况。

BJAF Web控制器提供了一个在固定时间段内只允许请求只允许提交一次的机制来防止表单重复提交的情况，在ControllerImp控制器抽象类提供了setAvoidSubmitSeconds(seconds :int) :void方法，参数seconds为秒数。若用户在此规定时间内重复提交的话，则会抛出一个ServletException异常。

例如：限制登录控制器在3秒，在LoginController控制器的构造函数内添加一句：

package com.beetle.WebDemo.presentation;

import java.util.Date;

...

public class LoginController extends ControllerImp {

public LoginController() {

this.setAvoidSubmitSeconds(3);

}

....

}

这样，在3秒内，如果客户端再次提出请求的话，则抛出不允许重复提交的异常信息：

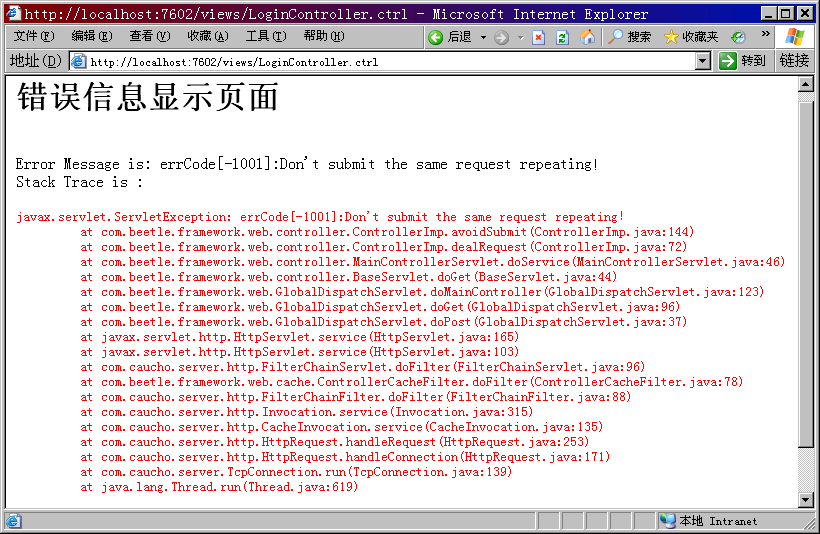


图3-24 重复提交异常页面截图

开发人员可以在错误处理页面捕捉此异常，提供更友好的提示信息。

### 4.7.3防止手工URL绕过验证进行请求访问

手工URL绕过验证是指用户记住某个功能服务URL，不按照系统登录验证等既定流程，而是直接在浏览器地址栏输入进行访问。为了防止此种情况的发生，BJAF Web框架提供了一个Session会话检查机制：启动控制器Session检查，当用户发起请求，而框架又检查系统没有Session存在时候，框架会把此请求跳转到WebView.xml文件内**<DisabledSessionView>**标签定义的视图。

由于在常见的设计策略中，用户验证后一般都会采取Session来保存用户会话数据，故此，在此情况下可以采取BJAF Web框架提供的控制器Session检查机制来防止用户手工输入URL绕过验证的情况发生。控制器默认是不会进行Session会话检查的，你需要在此控制器的构造函数内显式地调用enableSessionCheck()方法才能启动Session检查。例如启动DownloadController控制器的Session检查，代码如下：

package com.beetle.WebDemo.presentation;

...

public class DownloadController extends AbnormalViewControlerImp {

public DownloadController() {

this.enableSessionCheck();

}

...

}

### 4.7.4设置视图的缓存机制

我们知道，视图最终以html数据格式返回给浏览器。如果视图数据时效性不是很强，利用浏览器适当地缓存页面内容，是提高整个系统处理性能的一个很好的策略。由于http协议的header提供了一个Cache-Control参数，我们可以利用它来通知浏览器缓存页面的时效。

BJAF Web框架在控制器中封装了一个 “setCacheSeconds(cacheSeconds : int) : void”设置页面在浏览器端缓存秒数的方法。可以在控制器构造函数进行设置，例如：

package com.beetle.WebDemo.presentation;

...

public class LoginController extends ControllerImp {

public LoginController() {

this.setAvoidSubmitSeconds(3);

this.setCacheSeconds(4);//通知浏览器缓存此控制器返回视图页面4秒

}

...

}

注意的是，所有的控制器默认不对浏览器提出缓存视图页面的要求。

### 4.7.5页面验证码支持

BJAF Web框架通过实现一个特例VerifyCodeDraw页面绘图控制器来支持Web应用中常见的页面验证码功能。只需要在WebController.xml的<drawing>标签中，显式注册一下就可以了，如：

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

<mappings>

<controllers>

<drawing>

<dItem name="DemoDrawController.draw"

class="com.beetle.WebDemo.presentation.DrawPieController" />

<dItem name="DemoDraw2Controller.draw"

class="com.beetle.WebDemo.presentation.DrawPieController" />

<dItem name="VerifyCode.draw"

class="**com.beetle.framework.web.controller.draw.VerifyCodeDraw**" />

</drawing>

</controllers>

</mappings>

然后，在视图html页面添加以下html代码即可：

<div align="right">

验证码:

<img border="0" src="**VerifyCode.draw**"

style="cursor:pointer;" alt="看不清？点击这里换一个"

onclick="this.src='**VerifyCode.draw**'";>

</div>

其运行效果如下图所示：



图3-25验证码截图

另外，VerifyCodeDraw验证码控制来支持多种功能参数，见下表：

|  |  |
| --- | --- |
| **参数** | **含义说明** |
| width | 图片宽 |
| height | 图片高 |
| digit | 验证码位数 |
| type | 内容类型：1-纯数字；2-纯英文字母；3-数字和英文字母混合 |
| font | 验证码字体名称，如果指定为操作系统默认字体 |
| 例子：  VerifyCode.draw**?width=60&height=20&digit=4&type=3&font=Comic Sans MS** | |
| 说明：  宽-60，高-20，4位数验证码，数字和英文混合， 字体为：Comic Sans MS | |

VerifyCodeDraw与其它同类型采取Session缓存状态的验证码的实现不同，它采取Cookie来保存验证码的状态，由于不占用内存，VerifyCodeDraw的使用不会给系统造成Session资源的浪费。

值得注意的是，在同一个视图页面中，只能拥有一个验证码校验。

### 4.7.6特定请求并发控制

在Web应用开发过程中，有时候我们希望对某些请求处理任务量很重的控制器进行并发请求控制，以保护整个系统或通信网络不至于此类控制器由于并发量过大而导致迟缓或瘫痪。

框架为每个控制器内置一个计数器，可以通过设置此计数器大小来限制并发请求数。它通过控制器的“setMaxParallelAmount(amount:int):void”方法进行设置。

例如，提供数百兆文件下载的控制器，若不进行并发控制器的话，万一同时有1000用户下载数据的话，很容易造成网络阻塞，其它用户无法再访问服务器其它功能页面；此时，我们设计上有必要对此下载文件控制器进行并发控制，下面的代码设置控制器同时只允许5个并发下载。

package com.beetle.WebDemo.presentation;

...

public class DownloadController extends AbnormalViewControlerImp {

public DownloadController() {

this.setMaxParallelAmount(5);

}

...

}

值得注意的是，框架默认不会对控制器的并发访问进行任何控制，setMaxParallelAmount在控制器构造函数内显性调用后才会进行并发限制，而且，参数amount必须为大于0的整数才能有效。

### 4.7.7错误处理视图

BJAF Web框架针对Web应用异常处理机制实现了错误处理视图功能，它与Servlet规范的错误页面是相兼容的。开发人员可以选择Servlet的错误页面来处理和显示应用的异常信息，也可以采取BJAF Web框架提供的错误处理视图来完成。

错误处理视图定义在WebView.xml文件的**<ErrorView>**标签中，框架会判断此定义是否存在，若错误视图已经在标签内定义好，那么凡是控制器在执行过程发生的异常错误最终会传递到此错误视图，再显示处理。

一个典型的errView.jsp错误视图代码如下：

<%@page contentType="text/html; charset=GBK"%>

<%@page session="false"%>

<%@page import="com.beetle.framework.web.view.ViewHelper"%>

<html>

<link href="/views/t-1.css" rel="stylesheet" type="text/css">

<body bgcolor="#ffffff">

<h1>错误信息显示页面</h1>

**<%**

**ViewHelper helper=new ViewHelper(request);**

**%>**

<br>Error Code is: **<%= helper.getErrCode() %>**<br>

<br>Error Message is: **<%= helper.getErrMessage() %>**<br>

Stack Trace is : <pre><font color="red">**<%=helper.getErrStackTraceInfo() %>**</font></pre>

<br>

<p align="center"><a href="/index.html">返回</a></p>

</body>

</html>

WebView.xml定义如下：

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

<mappings>

<views>

<ErrorView>**ErrView**</ErrorView>

<standard>

<sItem name="**ErrView**" url="**/common/errView.jsp**" />

<!-- ... -->

</standard>

</views>

</mappings>

开发人员可以根据错误代码（**ErrCode**）来自定义更为友好的错误提示信息。

### 4.7.8控制器、视图及两者引用关系分析

BJAF Web框架内置一个ManageController的管理控制器，实现应用控制器、视图以及两者引用关系的列表展示，方便开发人员了解当前系统有多少个控制器，有多个视图，它们在系统的物理位置如何，某个控制器究竟引用了多少个视图等等。

这个内置控制器入口是：

**$framework.web.manage.ManageController.ctrl**

进入后显示以下页面：

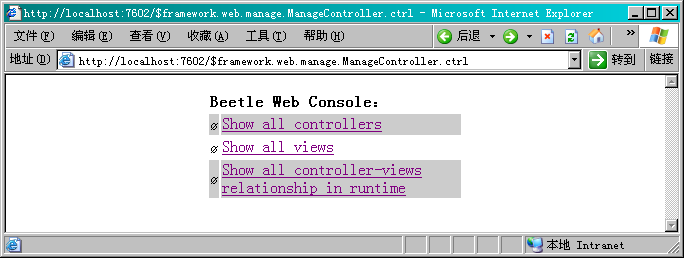


图3-26内置控制器管理界面截图

点击“Show all controllers”链接，显示所有控制器列表，见下图：

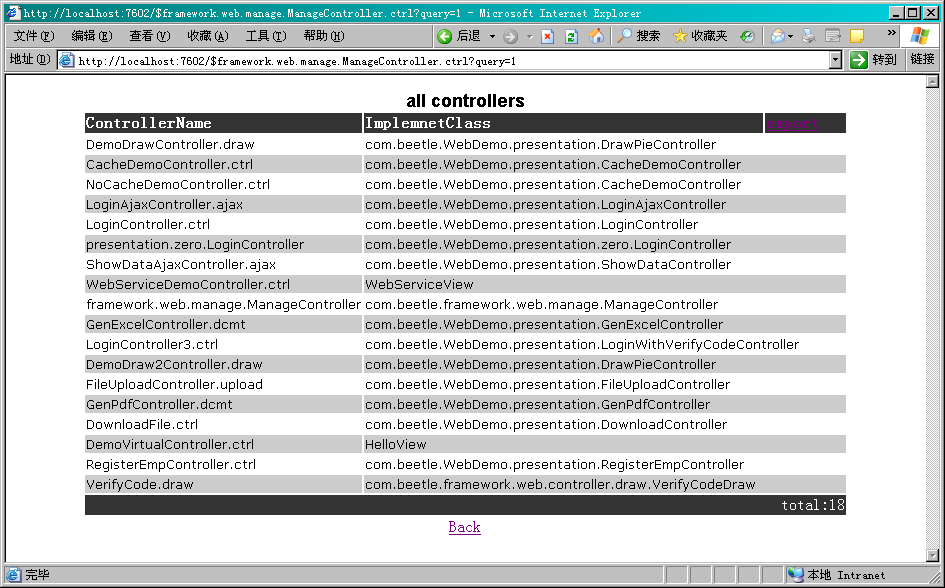


图3-27控制器列表截图

可见，在列表中清楚见到当前系统使用控制器名称及其具体实现类。点击“Show all views”进入以下视图列表：

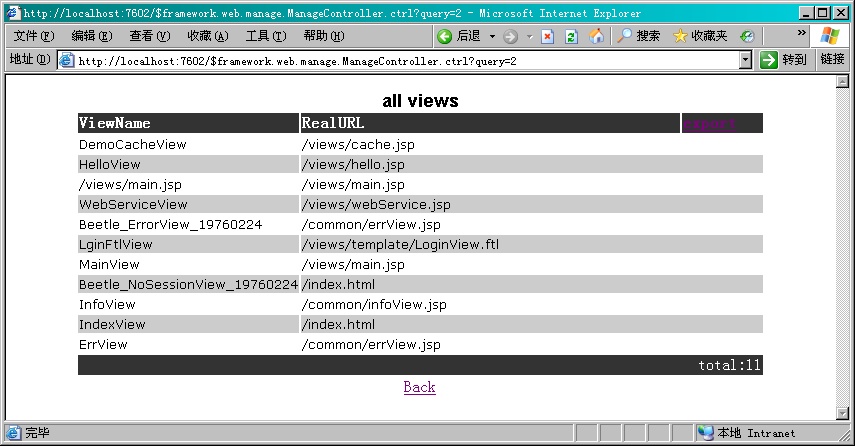


图3-28视图列表截图

可见，在列表中可以清楚了解当前系统使用的视图的名称以及此视图所对应的具体文件及其所在位置信息。点击“Show all controller-views relationship in runtime”，进入控制器与视图关系列表：



图3-29控制器视图引用关系列表截图

可见，通过上面列表很清晰知道控制器与视图的各种关系，例如：对于LoginController.ctrl控制器，可知其具体实现类是：“com.beetle.WebDemo.presentation.LoginController”；其引用了两个视图：“LginFtlView”和“MainView”；并且这两个视图的具体文件和路径一目了然。

81278576189468.jpg若想把这个功能禁止，可以通过在Web.xml文件中把CTRL\_VIEW\_MAP\_ENABLED参数设置为false即可。

## 4.8Web应用零配置编程

在一个标准的BJAF框架Web应用中，一共涉及到3个配置文件，分别是：**web.xml**、**WebController.xml**和**WebView.xml**。其中，web.xml是Servlet标准规范所必须的，不能省略；WebController.xml是控制器的配置文件，主要作用是在页面请求的URL与具体此请求的消费控制器类之间建立一个映射关系，为了省去这个配置文件，我们可以采取一种统一的编程约定来代替。我们BJAF框架采取的约定是：“**控制器名称=$+实现类名称（含包路径）+后缀**”；“**视图名称=视图所在物理路径+视图文件名称**”。

例如：WebDemo示例项目（参考框架开发包例子）中，login.html登录页面的控制器定义是：

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=GBK">

<title></title>

<link href="t-1.css" rel="stylesheet" type="text/css">

</head>

<body>

<form name="form1" method="post" action="**LoginController.ctrl**">[1]

<table width="34%" border="0" align="center">

<tr>

<td><div align="right">用户名:</div></td>

<td><input name="username" type="text" id="username" value="HenryYu"></td>

</tr>

<tr>

<td><div align="right">密 码:</div></td>

<td><input name="password" type="text" id="password" value="888888"></td>

</tr>

<tr>

<td colspan="2"><div align="center">

<INPUT TYPE="hidden" NAME="veiwFlag" value="0">

<input type="submit" name="Submit" value="提交">

</div></td>

</tr>

</table>

</form>

</body>

</html>

在WebController.xml配置文件中的定义是：

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

<mappings>

<controllers>

<standard>

**<sItem name="LoginController.ctrl" class="com.beetle.WebDemo.presentation.LoginController" />**

</standard>

</controllers>

</mappings>

那么，换成零配置模式，则[1]按照约定改成：

<form name="form1" method="post"

action="**$com.beetle.WebDemo.presentation.LoginController.ctrl**">[2]

即可。关于视图，LoginController控制器返回视图时候，标准模式写法是：

view = **new** View("MainView", vd); [3] // 返回MainView视图

而名为“MainView”的视图是需要定义在配置文件WebView.xml中的，如：

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

<mappings>

<views>

<standard>

**<sItem name="MainView" url="/views/main.jsp" />**

</standard>

</views>

</mappings>

那么换成零配置模式，则[3]按照约定，代码改成：

view = **new** View("/views/main.jsp", vd); // 直接返回视图的具体物理路径和文件名

即可。

可能读者注意到[2]把完整的实现类暴露在外，一，不利于安全，别人很容易就知道你的代码结构；二，写法上也不够优化，显得太长，太啰嗦，显然“com.beetle.WebDemo”包名称是公共的；那么有没有优化点定义方法呢？

当然，此时我们只要在web.xml应用配置中，添加一个“CTRL\_PREFIX”参数，把公共包名部分定义在此参数值里面，同时在控制器名称上把公共部分删除即可，如：

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

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app\_2\_3.dtd">

<web-app>

<context-param>

<param-name>CTRL\_PREFIX</param-name>

<param-value>com.beetle.WebDemo</param-value>

</context-param>

</web-app>

[2]改成：

<form name="form1" method="post" action="**$presentation.LoginController.ctrl**">

详细例子请参考BJAF框架开发的ZeroConfigWebDemo示例项目。

在BJAF1.4.0版本中，特别地针对原来的[虚拟控制器](#_虚拟控制器)场景也进行零配置的约定开发，一个物理视图文件[/path/filename]可以按照[$|path|filename.ctrlsuffix]形式来直接暴露。例如：

视图文件在web服务器上的位置是：**/views/main.jsp**，那么其直接在页面通过以下url作为一个虚拟控制器暴露：**$|views|main.ctrl**

可见其规则是，零配置控制器前面要加“$”；所有的 “/”使用“|”代替；原来的路径名保持不变；原来的文件名保持不变，但文件的后缀要换成web.xml文件中配置的控制器后缀名称。

81278576189468.jpg零配置带来了编码上方便，但是其配置上灵活性消除了。同时，控制器及视图维护上带来一定的挑战，为此，BJAF框架提供了一个的[控制器、视图及两者引用关系分析](#_控制器、视图及两者引用关系分析)内置管理模块来减低由于没有配置文件后带来的不便利性。笔者建议，在开发之前，最好对应用的结构，包定义，及视图所存放目录结构要事先有一个统一的规划，以便整个开发团队都必须遵从这个规范。

## 4.9关于请求重写urlRewrite

BJAF Web框架是基于请求驱动的，所有的请求都是\*.ctrl如：

<http://localhost:7070/eaat/$ArticleController.ctrl?$action=queryArticleById&rid=5>

现在很多流行基于restful风格以及便于搜索引擎检索，我们希望是这样的：

<http://localhost:7070/eaat/notice/query/4>

这个如何实现？目前框架没有内置实现方案，其实这本质上是url重写问题，当前可采取的方案有：

1. tomcat5.x版本其内置了url重写功能，可参考tomcat相关文档
2. 在服务器前面加多一个代理层，如：nginx，使用nginx的URLRewrite功能
3. 使用开源包：[UrlRewriteFilter](http://tuckey.org/urlrewrite/)，简单好用，例如上面的url改写要求，我们只需要在urlrewrite.xml文件中配置一个规则即可：

|  |
| --- |
| <rule>  <from>/notice/query/([A-Za-z0-9]+)</from>  <to type=*"forward"*>%{context-path}/$ArticleController.ctrl?$action=queryArticleById&amp;rid=$1</to>  </rule> |

又如，一个post请求通用点规则：

|  |
| --- |
| <rule>    <from>^/([\_a-zA-Z]+[\_0-9a-zA-Z-/]\*[\_0-9a-zA-Z]+)/?$</from>  <to type="forward">/$1.ctrl</to>  </rule> |

所以，关于请求重写我们这里推荐使用UrlRewriteFilter。详情如何安装配置，请参考其网站文件，这里就不啰嗦了。

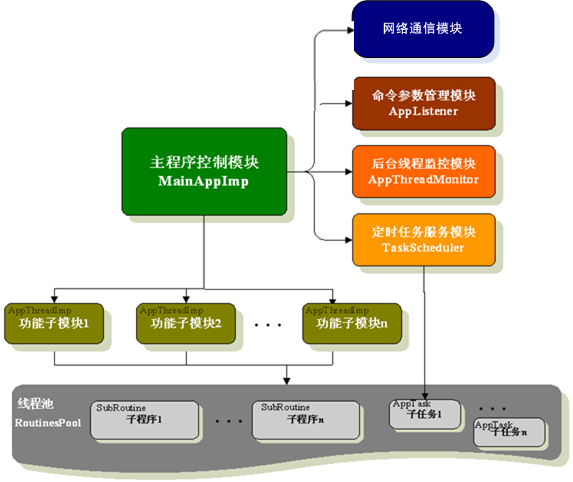
# 5.应用服务程序开发

## 5.1简介

我们在进行J2EE系统开发过程中，除了在现有的应用服务器（如：WebLogic、Jboss等）上构造企业应用外，很多时候我们需要自己编写应用服务程序来处理一些特殊的业务需要。如：编写一个集中交易的网络服务器、一个短信网关、一个客户端数据采集器等等，甚至，我们想编写自己的业务逻辑处理中间件。在J2EE规范中，类似这种应用服务器框架模型，是没有定义的，或许因为它可能不属于J2EE范畴，但在开发J2EE项目的时候，这些情况往往是很常见的。

为了解决自己编写应用服务程序中所面临的各个问题，BJAF框架特意提供了一个应用服务器程序子框架（主要实现在com.beetle.framework.appsrv包下）。它主要的功能特性有：

* 框架结构合理，清晰明了，开发简单方便。
* 多线程简易并发编程、线程池、简易锁、线程监控机制和恢复机制、线程超时事件回收机制。
* 支持子任务（TaskImp）及多种执行方式（串行、并行、超时等待）。
* 支持参数命令。
* 内存监控模块
* 定时计划任务的管理（触发/执行/暂停/删除等）
* 基于RPC、消息对象网络NIO通信模块。
* 支持插件开发



应用服务程序框架模块结构图

## 5.2应用程序模块

从上图可以知道：一个应用服务器程序一般都应包括一个命令参数管理模块，一个后台线程监控模块，定时计划任务管理模块和若干功能子模块。显然，我们需要一个主程序模块，其主要责任就是负责管理和协调好这些相关的模块，控制好程序的主流程。

在BJAF的应用服务程序子框架中，这个主程序模式是AppMainImp抽象类，其相关核心类关系图如下：



应用程序模块类图

从图可知，AppMainImp主要有以下功能：

1. 参数命令模块（启动或关闭）
2. 功能子模块（AppThreadImp）监控模型
3. 内存监控模块（防止内存溢出）
4. 服务程序的启动、关闭
5. 服务启动和关闭事件支持

示例如下：

**package** demo.XXXApp;

**import** com.beetle.framework.AppProperties;

**import** com.beetle.framework.appsrv.AppMainImp;

**import** com.beetle.framework.appsrv.plugin.PluginManager;

**import** com.beetle.framework.business.service.server.ServiceServer;

**import** com.beetle.framework.persistence.access.ConnectionFactory;

**import** com.beetle.framework.persistence.dao.DaoFactory;

**import** demo.XXXApp.plugin.IXXXPlugin;

**public** **class** XXXMain **extends** AppMainImp {

**private** **final** ServiceServer rpcSrv;// 支持RPC对外服务暴露的Server

**public** XXXMain(**int** cmdSrvPort) {

**super**(cmdSrvPort);

**int** rpcSrvPort = AppProperties.*getAsInt*("app\_rpcSrvPort", 9090);

rpcSrv = **new** ServiceServer(rpcSrvPort);

}

@Override

**protected** String dealCmd(String cmd) {

// 接受外部命令，并处理命令（外部命令一般是指shell脚本发出的）

// 例如：

StringBuilder sb = **new** StringBuilder();

**if** (cmd.equalsIgnoreCase("shutdown")) {

**this**.shutDownServer();

sb.append("shutdownOK");

} **else** **if** (cmd.equalsIgnoreCase("restartRpcServer")) {

**this**.rpcSrv.stop();

**this**.rpcSrv.start();

} **else** **if** (cmd.equalsIgnoreCase("help")) {

sb.append("-->help info:\n");

sb.append("-->cmd[shutdown]--shutdown this server\n");

sb.append("-->cmd[restartRpcServer]--restart rpc server\n");

} **else** **if** (cmd.equalsIgnoreCase("gc")) {

System.*gc*();

} **else** {

sb.append("sorry,err cmd!\n");

}

**return** sb.toString();

}

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

**int** localCmdSrvPort = AppProperties.*getAsInt*("app\_localCmdSrvPort",

22476);

XXXMain xm = **new** XXXMain(localCmdSrvPort);

**if** (args != **null** && args.length == 1) {

xm.executeCmd(args[0]);

} **else** {

xm.startServer();// 启动服务器

xm.startCmdService();// 启动外部命令接受服务

xm.startMemoryWatcherService();// 启动内存监控服务

xm.startThreadMonitorService();// 启动线程监控服务

}

}

@Override

**protected** **void** shutdownServerEvent() {

// 关闭服务器时，触发的动作，在此事件中释放整个应用资源，并处理应用状态数据的持久化保护工作

rpcSrv.stop();// 关闭rpc服务

PluginManager.*getInstance*().clear();// 清空插件容器资源

// ...

}

@Override

**protected** **void** starServerEvent() {

// 启动服务时出发的事件，在此事件中初始化整个应用的相关资源

ConnectionFactory.*initializeAllDataSources*();// 初始化数据源（连接池）

DaoFactory.*initialize*();// 初始化dao对象

PluginManager.*getInstance*().initialize(IXXXPlugin.**class**);// 初始化插件容器

rpcSrv.start();// 启动rpc服务

// ...

}

}

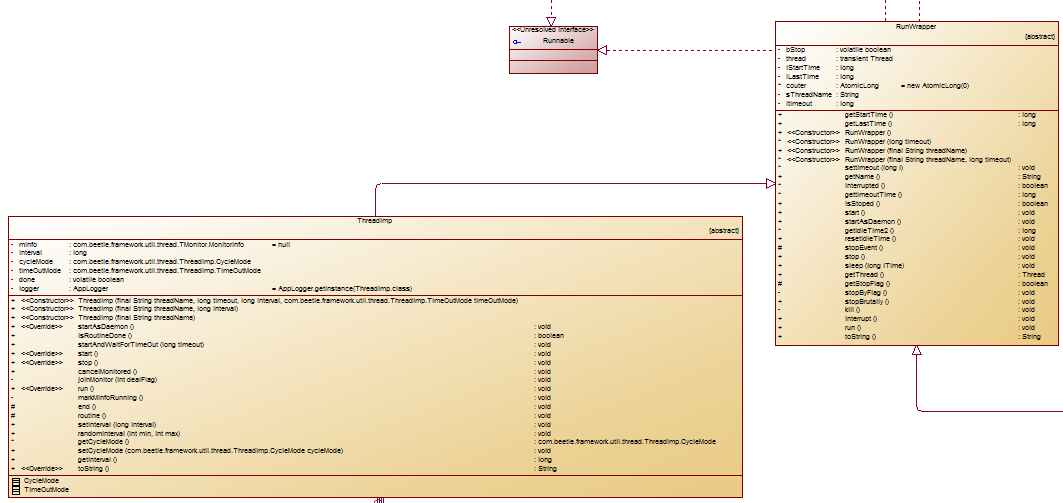
详细示例请参考SDK开发中的AppDemo的例子。

## 5.3线程的简化编程模型

应用程序服务子框架的实现了一个简单的线程编程模型来简化传统的线程开发，主要提供了以下功能：

* 线程自身的启动（常规启动或作为守护线程启动）、关闭（常规关闭或粗暴关闭）
* 线程自动循环执行，无需手工构建循环逻辑（也可手工构建）
* 支持线程结束事件
* 支持线程运行时阻塞状态监控及线程重新拉起
* 支持线程超时处理
* 支持线程震荡执行
* 消息发布订阅编程模型
* 消息流量管道消费控制编程模型
* 简易线程锁
* 并发计数器等等

其对外线程编程接口是ThreadImp抽象类，参考下面的类图：



线程类图

类图中相关方法的描述说明，请参考Java API文档。简单例子如下：

**package** example.thread;

**import** com.beetle.framework.util.thread.ThreadImp;

**public** **class** SimpleThread **extends** ThreadImp {

**public** SimpleThread(String threadName, **long** interval) {

**super**(threadName, interval);

}

@Override

**protected** **void** routine() **throws** Throwable {

System.*out*.println(System.*currentTimeMillis*());

}

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

**new** SimpleThread("SimpleThread", 3000l).start();

}

}

执行结果如下：

find [config/application.properties] and use it!

loaded {config/log4j.properties} from file

[DEBUG] 2013-05-30 16:25:45,763 [SimpleThread] -begin info- [com.beetle.framework.util.thread.ThreadImp.run(ThreadImp.java:221) ]

[DEBUG] 2013-05-30 16:25:45,817 [SimpleThread] -ThreadImp [mInfo=null, interval=3000, cycleMode=AUTO, done=false]- [com.beetle.framework.util.thread.ThreadImp.run(ThreadImp.java:222) ]

1369902345818

1369902348819

1369902351819

1369902354819

1369902357820

1369902360820

关于ThreadImp使用例子，也可以参考框架源代码的JobScheduler类实现。

81278576189468.jpg可见，相对传统的java线程编程，上面代码确实简便和简化不少。在我们应用服务程序框架中，图5-1中的功能子模块编程模型，对应就是ThreadImp抽象类。功能模块具体指利用线程实现功能在后台长期执行；如果执行一次就结束，完成一个任务计算，在BJAF框架中，我们定义为子任务（TaskImp），我们下一小节介绍。

## 5.4子任务及其执行方式

子任务（TaskImp）在BJAF框架中的定义是专门用来处理某一次的任务计算，处理完就结束。它本质上也是一个线程，只是这个线程执行一次就结束。另外，BJAF框架了，针对子任务实际运行情况，还实现了一个针对子任务任务执行的超时处理机制，用来解决由于某个任务长时间运行（超过预估的时间，或者死循环，阻塞挂起等）而无法及时线程回收的技术难题。

对于子任务，BJAF框架提供了线程池来优化其执行效率，同时对子任务的执行方式做了封装，（TaskExecutor）提供做多不同执行方式。

### 5.4.1子任务说明示例

TaskImp类说明如下：

|  |  |
| --- | --- |
| **方法与属性** | **功能说明** |
| TaskImp(maxBlockTime : int) | 构造函数，调用此构造函数实例化一个子任务，代表此子任务自动参与后台线程超时机制的监控。参数：maxBlockTime就是允许最大阻塞时间，单位为秒（S）；如果任务执行超过这个时间，框架会对这个子任务进行超时中断处理。 |
| TaskImp() | 构造函数，调用此构造函数实例化一个子任务，不参与后台线程超时机制监控。 |
| routine() : void | 子任务运行，为抽象方法，子任务执行任务技术所必须实现。 |
| setResult(result : Object) : void | 设置子任务返回结果 routine方法体内调用才有效。适合需要返回结果的子任务（带超时执行机制）  由TaskExecutor.runRoutineForResult方法执行 |

例如，建一个简单子任务，处理任务是打印一下当前时间戳。代码如下：

**package** example.appsrv.routine;

**import** com.beetle.framework.appsrv.TaskImp;

**public** **class** DemoRoutine **extends** TaskImp {

**public** DemoRoutine() {

**super**();

}

**protected** **void** routine() **throws** InterruptedException {

System.*out*.println(System.*currentTimeMillis*() + "->do something...");

}

}

利用TaskExecutor子任务执行：

**package** example.appsrv.routine;

**import** com.beetle.framework.appsrv.TaskExecutor;

**public** **class** TestClient {

**public** **static** **void** main(String[] args) {

TaskExecutor.*runRoutineInPool*(**new** DemoRoutine());

}

}

执行结果：

loaded /config/log4j.properties from file

1235541073513->do something...

下面演示一下，子任务超时回收功能。假设，上面的DemoRoutine在开发的时候不小心写了个死循环，导致子任务长时间吊死。此时，我们能够及时回收此子任务线程就显得很有必要了。

**package** example.appsrv.routine;

**import** com.beetle.framework.appsrv.TaskImp;

**public** **class** DemoRoutine **extends** TaskImp {

**public** DemoRoutine(**int** maxBlockTime) {

**super**(maxBlockTime);// 采取超时参数构造函数

}

**protected** **void** routine() **throws** InterruptedException {

**while** (**true**) {// 死循环

System.*out*.println(System.*currentTimeMillis*() + "->do something...");

}

}

}

执行此子任务：

**package** example.appsrv.routine;

**import** com.beetle.framework.appsrv.TaskExecutor;

**public** **class** TestClient {

**public** **static** **void** main(String[] args) {

TaskExecutor.*runRoutineInPool*(**new** DemoRoutine(5));// 最大阻塞时间为5秒

}

}

执行结果：

loaded /config/log4j.properties from file

1235551942465->do something...

.....//省略重复输出

1235551942465->do something...

1235551942465->do something...

com.beetle.framework.appsrv.RoutineRunException: thread interrpting

at com.beetle.framework.appsrv.TaskImp.run(TaskImp.java:114)

at com.beetle.framework.appsrv.RoutinesPool$ThreadPoolExecutor$Worker.run(RoutinesPool.java:795)

at java.lang.Thread.run(Thread.java:534)

5190 DEBUG [Thread-1] com.beetle.framework.appsrv.RoutinesPool$RoutineMonitor - Thread:[gcaHS0gnMy]killed!--

可见超过5秒，框架后台监控服务就会把这个死循环的子任务（线程）给及时中断回收。

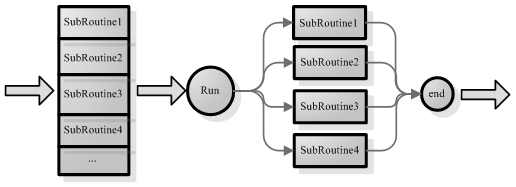
### 5.4.2执行方式说明及示例

TaskExecutor提供了以下执行方式：

|  |  |
| --- | --- |
| **方法与属性** | **功能说明** |
| runRoutineDirect(TaskImp: TaskImp) : void | 直接执行，为静态方法。不采取线程池，直接创建线程执行。除了直接执行方法外，执行器所有的执行方法都会将子任务放在线程池内执行 |
| runRoutineInPool(TaskImp: TaskImp) : void | 在线程池[1]中，执行此子任务。 |
| addTaskImp(TaskImp: TaskImp) : void | 往执行器中，添加一个子任务。（执行器，支持多个子任务一起执行） |
| runRoutineEarly() : void | 提早执行子任务，后调用getResult方法获取运算结果 特别适合在主流程中提前先处理任务重部分，再处理其它任务，最后再获取重任务计算结果的场景。这样处理的最大好处是优化和节约主流程的执行时间 |
| getResult() : Object | 获取此子任务的处理结果（此方法会阻塞） |
| runRoutineForTime() : Object | 执行子任务并等待返回其计算结果。 根据此子任务设置最大阻塞时间来防止线程超时，则超出此时间会中断此子任务 并触发子任务的terminate()事件（方法）  @return 子任务结果 |
| runRoutineForTime(time : int) : Object | 同上。  只是支持自定义等待时间 |
| runRoutineParalleJoin() : void | 并行执行子任务，并等待所有的子任务结束后再退出 （针对一组子任务，此方法会阻塞） （没有超时处理机制，即使子任务设置最大阻塞时间也无效） |
| runRoutineInTurn() : void | 依次执行子任务（按照顺序前一个子任务运行完毕才接着运行下一个，直到所有的子任务执行完毕） （针对一组子任务） |
| [1]线程池相关的参数配置在RoutinesPool.properties属性文件里面，默认的配置为：  #设置线程池的属性  pool-max-size=200  pool-min-size=20  #m  idlesse-minute=1  可以根据需求通过修改这个文件优化设置。 | |

常规执行一个子任务前面代码已经演示过，下面示例一下其它特别执行方式：

针对一组子任务，并行执行，并等待所有子任务结束后再返回



并行执行示意图

构建3个子任务，代码分别为，SR1代码：

**package** example.appsrv.routine;

**import** com.beetle.framework.appsrv.TaskImp;

**public** **class** SR1 **extends** TaskImp {

**protected** **void** routine() **throws** InterruptedException {

System.*out*.println("sr1-begin");

*sleep*(3000);

System.*out*.println("sr1-end");

}

}

SR2代码：

**package** example.appsrv.routine;

**import** com.beetle.framework.appsrv.TaskImp;

**public** **class** SR2 **extends** TaskImp {

**protected** **void** routine() **throws** InterruptedException {

System.*out*.println("sr2-begin");

*sleep*(2000);

System.*out*.println("sr2-end");

}

}

SR3代码：

**package** example.appsrv.routine;

**import** com.beetle.framework.appsrv.TaskImp;

**public** **class** SR3 **extends** TaskImp {

**protected** **void** routine() **throws** InterruptedException {

System.*out*.println("sr3-begin");

*sleep*(1000);

System.*out*.println("sr3-end");

}

}

编写执行客户端代码：

**package** example.appsrv.routine;

**import** com.beetle.framework.appsrv.TaskExecutor;

**public** **class** TestParalleClient {

**public** **static** **void** main(String[] args) {

// 构建一个子任务执行器

TaskExecutor re = **new** TaskExecutor();

re.addTaskImp(**new** SR1());// 添加各个子任务到执行器队列

re.addTaskImp(**new** SR2());

re.addTaskImp(**new** SR3());

re.runRoutineParalleJoin();// 并行执行，并阻塞，等待队列中所有子任务都结束才返回

System.*out*.println("ok");

}

}

执行结果：

sr1-begin

sr3-begin

sr2-begin

sr3-end

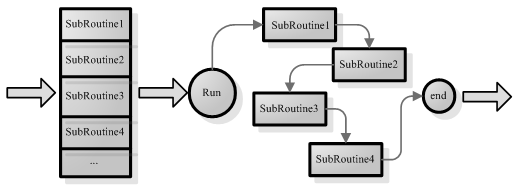
sr2-end

sr1-end

ok

81278576189468.jpg此模型对于哪些计算量很巨大任务的处理很有帮助，我们可以把此任务按照一定的条件，分解成多个子任务，并行处理，从而加快任务处理速度。

针对一组子任务，依次串行执行，并等待所有子任务结束后再返回



串行执行示意图

沿用前面的3个子任务，串行执行的客户端代码如下：

**package** example.appsrv.routine;

**import** com.beetle.framework.appsrv.TaskExecutor;

**public** **class** TestInTurnClient {

**public** **static** **void** main(String[] args) {

// 构建一个子任务执行器

TaskExecutor re = **new** TaskExecutor();

re.addTaskImp(**new** SR1());// 添加各个子任务到执行器队列

re.addTaskImp(**new** SR2());

re.addTaskImp(**new** SR3());

re.runRoutineInTurn();// 串行执行，并阻塞，等待队列中所有子任务都结束才返回

System.*out*.println("ok");

}

}

执行结果：

sr1-begin

sr1-end

sr2-begin

sr2-end

sr3-begin

sr3-end

ok

81278576189468.jpg执行器还提供了一个runRoutineInTurnNoBlock()方法，不会阻塞主流程，让其在后台串行执行。

先执行，后拿结果

当我们在主流程中处理多个任务，若这些任务中，有某个计算量很大，十分消耗时间，为了提高主流程的处理速度，我们可以把这个任务封装成子任务，先执行，主流程处理完其它任务后，再获取这个任务的结果。

示例代码如下：

编写一个HardWorkSR子任务：

**package** example.appsrv.routine;

**import** java.util.ArrayList;

**import** java.util.List;

**import** com.beetle.framework.appsrv.TaskImp;

**public** **class** HardWorkSR **extends** TaskImp {

**protected** **void** routine() **throws** InterruptedException {

System.*out*.println("work-begin");

List data = **new** ArrayList();

*sleep*(10000);// 假设此任务要计算10秒

data.add("AAA");

data.add("BBB");

**this**.setResult(data);// 设置结果以便返回

System.*out*.println("word-end");

}

}

编写客户端：

**package** example.appsrv.routine;

**import** java.util.List;

**import** com.beetle.framework.appsrv.TaskExecutor;

**public** **class** TestEarlyClient {

**public** **static** **void** main(String[] args) {

TaskExecutor re = **new** TaskExecutor();

re.addTaskImp(**new** HardWorkSR());

re.runRoutineEarly();// 提早执行

// ...继续处理其它任务

System.*out*.println("do other work...");

// 处理完其它任务后，再来拿结果

List result = (List) re.getResult();// 会阻塞一定等到任务处理完毕有结果返回为止

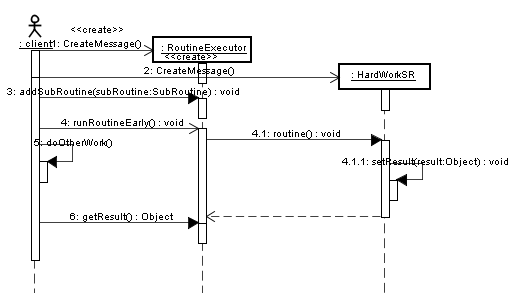
System.*out*.println(result);

System.*out*.println("ok");

}

}

执行过程，参考一下顺序图：



先执行后取结果示例顺序图

执行结果如下：

do other work...

work-begin

word-end

[AAA, BBB]

ok

具备超时保护并能获取结果的执行

沿用上面的HardWorkSR子任务（其计算时间为10s），参考以下执行方式的区别：

**package** example.appsrv.routine;

**import** java.util.List;

**import** com.beetle.framework.appsrv.TaskExecutor;

**public** **class** TestTimeoutClient {

**public** **static** **void** main(String[] args) {

TaskExecutor re = **new** TaskExecutor();

re.addTaskImp(**new** HardWorkSR());

List result = (List) re.runRoutineForTime(11);//最大允许子任务执行11秒

System.*out*.println(result);

System.*out*.println("ok");

}

}

从代码可知，超时设置为11秒，HardWorkSR子任务会正常执行，不会做超时处理，此时，其运行结果如下：

work-begin

word-end

[AAA, BBB]

ok

若把上面的时间设置为5秒，如：

List result = (List) re.runRoutineForTime(5);// 最大允许子任务执行5秒

由于HardWorkSR本身需要10秒才能计算完成，而我们5秒就要回收，所以其会被超时处理，此时，其运行结果如下：

work-begin

com.beetle.framework.appsrv.RoutineRunException: thread timeout; cause exception is:

EDU.oswego.cs.dl.util.concurrent.TimeoutException

EDU.oswego.cs.dl.util.concurrent.TimeoutException

at EDU.oswego.cs.dl.util.concurrent.FutureResult.timedGet(FutureResult.java:128)

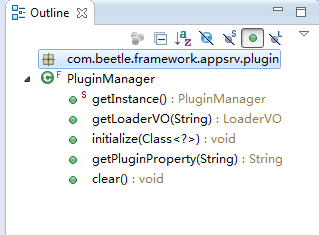
at com.beetle.framework.appsrv.TaskExecutor.runRoutineForTime(TaskExecutor.java:104)

at example.appsrv.routine.TestTimeoutClient.main(TestTimeoutClient.java:12)

线程超时，执行中断。

## 5.5插件开发和管理

插件是应用的一种功能动态扩展的机制，特别是当应用在运行时，动态添加功能特别有用。BJAF对此做了封装，提供了一个简单的插件管理功能，以Jar包形式发布插件，支持Jar动态加载和执行。由插件管理器（PluginManager）类完成整个插件生命周期的管理工作。



其开发过程如下：

（1），根据自己功能扩展需要，定一个插件功能接口，例如：

**package** demo.XXXApp.plugin;

**public** **interface** IXXXPlugin {

**void** deal(String word);

}

（2），调用PluginManager初始化这个接口（就是注册）

PluginManager.getInstance().initialize(IXXXPlugin.class);// 初始化插件容器

（3），在application.properties配置插件管理参数

appsrv\_plugin\_DirPath=plugin/ //指定插件目录，存放jar

appsrv\_plugin\_PropertiesFileName=plugin/plugin.properties //指定插件属性文件

（4），使用PluginManager来执行插件，例如：

**public** **void** callPluginDemo(String moduleId) {

PluginManager pm = PluginManager.*getInstance*();

String jarname = pm.getPluginProperty(moduleId);

LoaderVO lvo = pm.getLoaderVO(jarname);

System.*out*.println("-->");

System.*out*.println(lvo);

IXXXPlugin xp = (IXXXPlugin) lvo.getHandler();

xp.deal("hi,plugin!");

}

此时插件容器编程完成，启动应用让其处于运行状态。

（5），编写插件，例如，我们根据上面的接口编写了1个插件

package plugin;

import demo.XXXApp.plugin.IXXXPlugin;

import java.io.PrintStream;

public class AModulePlugin

implements IXXXPlugin

{

public void deal(String word)

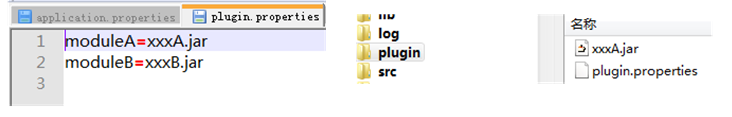
{

System.out.println("AModulePlugin[" + word + "]");

}

}

配置好属性文件，并打包成xxxA.jar放入Plugin插件目录内

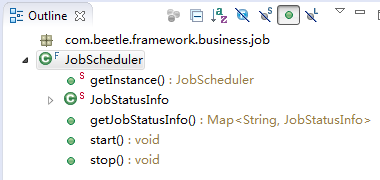


这样插件容器会自动发现插件并加载，当请求触发callPluginDemo函数时，会根据输入模块参数找到相应的插件去执行IXXXPlugin接口定义的deal方法。

## 5.6定时计划任务

在我们日常系统维护、开发中，经常需要制定一些工作计划，例如：在某月的某个时间，备份生产系统的交易数据。在此计划中，“备份生产系统交易数据”我们叫做工作（Job）；显然，计划定义出来以后，接下来我们就要考虑如何去执行它，进一步，我们还要思考去管理它。

BJAF框架实现了一个支持cron-时间表达式的Job任务计划调度管理器（JobScheduler）：



其实现原理是，所有的Job配置在JobConfig.xml文件，其格式如下：

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

<jobs>

<!--

\*每个Job根据cron时间表达式串行执行，也就是说对同一个Job而已，执行完一次，才会接着执行下一次

\*id-唯一标识，代表不同的job

\*implement-Job接口的实现类

\*enabled-有效启动标记，true代表后天会按时启动；false代表不会按时启动，如果一个Job已经启动再修改此参数，则代表执行完毕后，下次不再执行

\*cron-时间表达式

\*timeout-超时参数，为可选参数，代表某个Job执行超过这个时间是否需要强行回收；单位为毫秒ms，如果为0或负数代表参数无效，不会被强行回收

\*starttime-Job开始计算参数，可选参数，代表在此时间之前，即使时间表达式满足触发条件，Job也不会触发，格式为【yyyy-MM-dd HH:mm:ss】，不设置此参数，默认为配置时间

-->

<!--

<item id="hardTestJob" implement="example.job.HardJob" enabled="true" cron="0/10 \* \* \* \* ?" timeout="30000" starttime="2012-11-24 12:30:01" />

-->

<item id=**"dynamicToStaticHtmlJob"** implement=**"com.flyman.AdPlatform.jobserver.jobs.DynamicToStaticHtmlJob"** enabled=**"true"** cron=**"0 0/1 \* \* \* ?"** />

</jobs>

配置文件支持热部署，即可以在运行时状态下动态添加Job和更改Job状态（例如：停止Job执行），每个Job根据cron时间表达式串行执行，也就是说对同一个Job而已，执行完一次，才会接着执行下一次；支持Job执行超时终止功能，即Job执行超过预设的时间后还没有结束，调度器会自动强行终止此Job，回收资源。

JobScheduler会定时扫描JobConfig.xml配置文件，根据Job定义的时间表达式的语意来执行Job。JobScheduler的Start方法来启动调度器，Stop方法来停止调度器。

开发一个Job必须实现JobImp接口的run方法即可。例如，定时生成静态文件的DynamicToStaticHtmlJob实现代码如下：

**package** com.flyman.AdPlatform.jobserver.jobs;

**import** java.io.BufferedReader;

**import** java.io.File;

**import** java.io.FileOutputStream;

**import** java.io.InputStream;

**import** java.io.InputStreamReader;

**import** java.io.OutputStreamWriter;

**import** java.io.PrintWriter;

**import** java.net.URL;

**import** java.util.List;

**import** com.beetle.framework.AppProperties;

**import** com.beetle.framework.business.job.JobImp;

**import** com.beetle.framework.log.AppLogger;

**public** **class** DynamicToStaticHtmlJob **implements** JobImp {

**private** **final** **static** AppLogger *logger* = AppLogger

.*getInstance*(DynamicToStaticHtmlJob.**class**);

**private** **static** **void** genPage(**final** String filename, String url\_addr)

**throws** Exception {

PrintWriter fileOut = **null**;

InputStream ins = **null**;

**try** {

URL url = **new** URL(url\_addr);

ins = url.openStream();

BufferedReader bReader = **new** BufferedReader(**new** InputStreamReader(

ins));

FileOutputStream out = **new** FileOutputStream(filename);

OutputStreamWriter writer = **new** OutputStreamWriter(out);

fileOut = **new** PrintWriter(writer);

String rLine = **null**;

**while** ((rLine = bReader.readLine()) != **null**) {

String tmp\_rLine = rLine;

**int** str\_len = tmp\_rLine.length();

**if** (str\_len > 0) {

fileOut.println(tmp\_rLine);

fileOut.flush();

}

tmp\_rLine = **null**;

}

} **finally** {// 关闭资源

**if** (fileOut != **null**) {

fileOut.close();

}

**if** (ins != **null**) {

ins.close();

}

}

}

**private** **static** **void** dealOneRequest(**final** String filename, String url\_addr) {

String tmpfile = filename + ".tmp";

**try** {

*genPage*(tmpfile, url\_addr);

File tf = **new** File(tmpfile);

**if** (tf.exists()) {

**boolean** f = tf.renameTo(**new** File(filename));

**if** (!f) {

*logger*.warn("file({}) rename to {} faild! ", tmpfile,

filename);

}

}

} **catch** (Exception e) {

*logger*.error(*logger*.strFormat("gen req({}) to file({}) err!",

url\_addr, filename), e);

}

}

@Override

**public** **void** run() {

List<String> tasks = AppProperties

.*getByPrefixName*("DynamicToStaticHtmlJob\_");

**if** (!tasks.isEmpty()) {

**try** {

**for** (String fn : tasks) {

*logger*.info("deal {} ...", fn);

String a[] = fn.split("->");

*dealOneRequest*(a[1], a[0]);

*logger*.info("deal {} OK", fn);

}

} **finally** {

tasks.clear();

}

}

}

}

另外可以通过application.properties文件中**job\_FileMonitor\_interval**参数来调整检查JobConfig.xml的时间频率，默认30000毫秒，**job\_Scheduler\_interval**参数来调整调度逻辑的检查时间间隔，默认是250毫秒。

针对没有EJB而又要使用RPC分布式场景的Servlet容器，如：Tomcat、Jetty等，BJAF使用本节介绍的应用服务器程序开发框架实现了一个**BusinessAppSrv**，业务应用服务器（可单独执行的进程也可以在Tomcat等容器内作为一个内置组件去运行）。其包括：RPC Service、Job调度、数据库连接池初始化、DIC容器初始化和后台交互命令等功能模块，同时指出服务器启动事件扩展。



其相关参数配置application.properties文件中，如下：

**[BusinessAppSrv]**

**#框架内置业务层RPC服务器，针对没有容器的分布式场景**

**businessAppSrv\_dbPool\_initialized=true**

**businessAppSrv\_daoObject\_initialized=false**

**businessAppSrv\_jobScheduler\_initialized=true**

**businessAppSrv\_rpcServer\_port=7802**

**businessAppSrv\_localCmdServer\_port=22478**

**businessAppSrv\_rpcServer\_ipBind=192.168.1.115**

**businessAppSrv\_eventImp=**

如果你希望在Tomcat等servlet容器中启动时，同时启动**BusinessAppSrv**服务器，你可以通过设置application.properties文件的参数web\_businessAppSrv\_enabled=true这样容器启动时就会一起将**BusinessAppSrv**启动。如果你要脱离容器启动，你可以直接执行com.beetle.framework.business.server.BusinessAppSrv程序。

# 6.CoreUtil

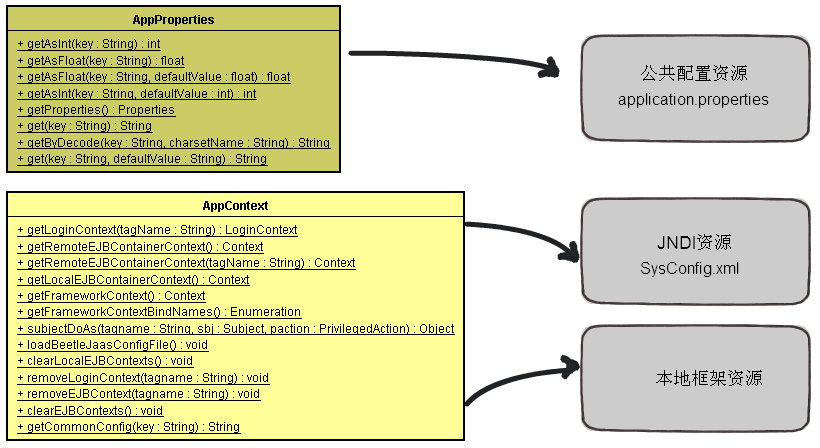
CoreUtil是BJAF框架核心公共组件层，它主要包括：日志、异常处理、系统配置、系统资源管理、邮件发送和对象缓存等公共组件，是BJAF其它子框架构建基础工具包。其相关包UML类图如下：

651278576189468.emf

BJAF核心层组件

## 6.1资源管理

资源管理主要对EJB容器的JNDI资源、框架本地资源及框架本身公共信息配置资源作了封装，对于提供统一的API来获取。



资源统一访问API

### 6.1.1公共配置资源属性

为了便于BJAF各个子框架模块相关资源属性的管理以及开发人员自定义属性资源统一管理，BJAF框架在1.4.0版本其引入了“***application.properties***”属性配置文件，对BJAF而言，一个BJAF应用必须对应一个***application.properties***配置文件（配置文件内容参数可以根据应用的特点和需求进行裁剪，例如：如果你这个应用属于RPC客户端应用，那么你可以将配置文件中相关的RPC服务端配置删除掉）。并封装了**AppProperties**类统一对外编程api。

**AppProperties**类对“***application.properties***”属性文件读取遵循以下规则：

1. 若通过jvm参数[***-Dbeetle.application.properties.file.path*=d://xxx//yyy**]指定其存放目录，则优先从此指定目录读取；
2. 若不显性指定路径，则首先从AppContext里面查找定义，如果存在则按照定义路径加载（提供在程序设置路径的接口），否则在当前应用的工作目录下（相对路径）的config子目录下寻找并加载；
3. 若[2]读取不到，则会从当前应用的classpath的config子目录下寻找并加载。
4. 若以上资源路径都获取不到，则会抛出异常。

BJAF框架在 “***application.properties***”文件已定义的属性如下：

|  |
| --- |
| #BJAF框架配置  #---------------------------------------------------------------------------------------------------------------  **[resource]**  ##框架相关资源配置##  #1-启动；0-不启动#  resource\_BUSINESS\_WATCH**=**1  #ContainerConfig.xml文件SECURITY\_CREDENTIALS属性是否加密,1-是;0-否#  resource\_CONFIG\_MASK**=**0  ##log4j重用开关(如果容器已经加载了log4j配置是否重用它1--用；0--不用)  resource\_LOG4J\_REUSE**=**0  ##加入Application Context应用上下文配置文件(统一放在应用的config目标下)  resource\_DI\_CONTAINER\_FILES**=**DAOConfig.xml;ServiceConfig.xml;AOPConfig.xml  ##框架内置线程池属性配置##  #公共线程池大小最大值  routinespool\_POOL\_MAX\_SIZE**=**200  #公共线程池大小最小值  routinespool\_POOL\_MIN\_SIZE**=**20  #线程池空闲回收时间，单位分钟min  routinespool\_IDLESSE\_MINUTE**=**5  #线程后台健康线程监控时间间隔，单位毫秒ms  routinespool\_MONITOR\_INTERVAL**=**250  **[command]**  #command设置  #执行方式标记  ##{1-EJB容器本地执行[Local接口],2-EJB容器远程执行[Remote接口],3-本地微容器POJO方式执行,4-RPC远程调用执行.  ##方式1和2设置拥有EJB容器的应用服务器，方式1适合集中式应用，方式2适合分布式应用；  ##方式3和4设置适合没有EJB容器的应用服务器。}  command\_execute\_flag**=**3  #异步调用线程池队列大小，值越大并发处理能力越强  command\_asynchro\_call\_queue\_size**=**10  command\_ejb\_jndi\_name**=**CommandServerBean  command\_ejb\_local\_jndi\_name**=**CommandServerBeanLocal  #是否缓存远程接口商业对象，默认为false  command\_ejb\_remote\_object\_cache**=**false  **[appsrv]**  #appsrv设置  appsrv\_memGuard\_allocateNum**=**524288  appsrv\_plugin\_DirPath**=**plugin/  appsrv\_plugin\_PropertiesFileName**=**plugin/plugin.properties  appsrv\_cmd\_waitForTime**=**10000  **[rpc]**  #rpc设置  rpc\_common\_maxObjectSize**=**10485760  #rpc\_common\_codec=java/jbossSerialization/json/hessian  rpc\_common\_codec**=**jbossSerialization  rpc\_server\_tcpNoDelay**=**false  rpc\_server\_reuseAddress**=**true  rpc\_server\_child\_tcpNoDelay**=**true  rpc\_server\_child\_receiveBufferSize**=**8192  rpc\_server\_workThreadPool\_coreSize**=**100  rpc\_server\_workThreadPool\_MaxSize**=**500  rpc\_server\_workThreadPool\_keepAliveTime**=**300000  rpc\_client\_tcpNoDelay**=**false  rpc\_client\_keepAlive**=**true  #soLinger参数  ##{soLinger<0,socket执行close方法后不会立即关闭网络连接，它会延迟一段时间，等完剩余的数据发送完毕，才会真正关闭释放资源;  ##soLinger=0,close后立即关闭并释放资源;  ##soLinger=X,X>0,close方法会最多阻塞X秒，在X秒内如果数据发送完毕，即立即释放资源返回；如果超过X秒数据还没发完，则不再理会，立即关闭资源返回}  rpc\_client\_soLinger**=**0  rpc\_client\_connectTimeoutMillis**=**30000  rpc\_client\_receiveBufferSize**=**8192  rpc\_client\_connectionAmount**=**5  rpc\_client\_invoke\_max\_waitForTime**=**600000  rpc\_client\_retry\_waitForTime**=**30000  #rpc\_client\_remoteAddress={127.0.0.1:7602;192.168.1.15:8080}  rpc\_client\_remoteAddress**=**{127.0.0.1:7602}  #rpc\_client\_proxyInvoke=jvm  **[job]**  #JobConfig.xml热部署检查时间间隔  job\_FileMonitor\_interval**=**30000  job\_Scheduler\_interval**=**250  **[BusinessAppSrv]**  #框架内置业务层RPC服务器，针对没有容器的分布式场景  businessAppSrv\_dbPool\_initialized**=**true  businessAppSrv\_daoObject\_initialized**=**true  businessAppSrv\_jobScheduler\_initialized**=**false  businessAppSrv\_rpcServer\_port**=**7602  businessAppSrv\_rpcServer\_ipBind**=**0.0.0.0  businessAppSrv\_localCmdServer\_port**=**22476  businessAppSrv\_eventImp**=**  **[web]**  #web设置  #ws处理包Jackson/XStream,不设置默认Jackson  web\_ws\_jsonProcessor**=**Jackson  #针对控制器设置了session检查功能，若session不存在或失效，  #则会跳转到此视图（与WebView.xml的<DisabledSessionView>一样，  #只是为了统一配置而已，此配置比WebView.xml优先）  web\_view\_DisabledSessionView**=**/common/loseSession.jsp  web\_view\_ErrorView**=**/common/err.jsp  #是否在web启动的时候同时启动BusinessAppSrv业务层的RPC服务，默认不启动  web\_businessAppSrv\_enabled**=**false  **[extend]**  #自定义扩展配置  #---------------------------------------------------------------------------------------------------------------  author**=**余浩东  xxxKey**=**xxxValue |

使用**AppProperties**类读取此属性文件示例如下：

**public** **static** **void** main(String arg[]) {

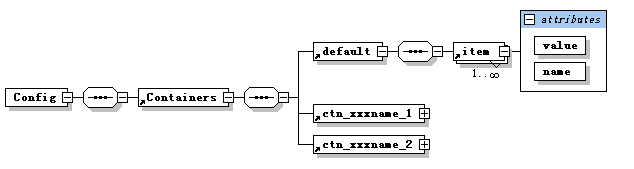
**int** size=AppProperties.*getAsInt*("routinespool\_POOL\_MAX\_SIZE");

String test=AppProperties.*get*("test");

}

### 6.1.2 JNDI资源查找

从图6-2可知，通过AppContext类的getRemoteEJBContainerContext方法来获取EJB容器的JNDI目录服务的上下文，EJB容器相关地址信息配置在ContainerConfig.xml文件中，其格式如下：



ContainerConfig.xml格式

文件支持多个容器的配置，若只要一个容器，则配置在*<default>*默认标签中；多于一个容器，则自行定义标签的名称，如：<ctn\_xxxname\_1>。标签体内包含容器具体属性数据，通过name-value的形式配置：

|  |  |
| --- | --- |
| **属性（name）** | **说明** |
| INITIAL\_CONTEXT\_FACTORY | J2EE规范中EJB容器验证标准配置，各项属性说明请参考相关EJB编程文档。 |
| PROVIDER\_URL |
| URL\_PKG\_PREFIXES |
| SECURITY\_PRINCIPAL |
| SECURITY\_CREDENTIALS |
| VALIDATE\_TYPE | 容器登录验证类型：其值为："JNDI"或"JAAS"默认为JNDI |
| CONTAINER\_PRODUCT | 容器产品：其值为：jboss,weblogic,websphere |
| GROUP\_NAMES[\*] | 群组定义，当访问此容器失败时， 框架会判断是否存在群组定义，若存在，则从此定义中逐个尝试获取其它的EJB容器。  存在多个容器，名称用‘#’分隔 |
| INVOKE\_LIMIT[\*] | 容器调用的并发控制 |
| TIME\_OUT[\*] | 容器超时时间，单位：ms 默认300000ms(5min) |
| DELEGATE\_JNDI\_NAME[\*] | 异步消息框中默认Delegate的JNDI名称 |
| ROUND-ROBIN-AFFINITY[\*] | ROUND-ROBIN-AFFINITY负载均衡算法是否启动 (仅对weblogic有效，其它容器必须为false) |
| CLUSTER\_ENABLED[\*] | 是否配置集群，true为是，false为否 |
| CLUSTER\_HOSTS[\*] | 集群机器地址(hostname1,192.168.1.10:7001#hostname2,10.168.1.101:7001)可选配置 |
| [\*] 为可选，只要才使用基于异步消息的ESB总线框架中才起作用。 | |

例如：jboss容器的配置如下：

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

<Config>

<!-- EJB容器远程访问设置 -->

<Containers>

<!-- 默认容器，系统必须拥有一个<default>默认EJB容器设置 -->

<default>

<item name="INITIAL\_CONTEXT\_FACTORY"

value="org.jnp.interfaces.NamingContextFactory"/>

<item name="PROVIDER\_URL" value="127.0.0.1:2099"/>

<item name="URL\_PKG\_PREFIXES" value="org.jboss.naming:org.jnp.interfaces"/>

<item name="SECURITY\_PRINCIPAL" value="test\_user"/>

<item name="SECURITY\_CREDENTIALS" value="test\_pass"/>

<!-- 容器登录验证类型：其值为："JNDI"或"JAAS"默认为JNDI -->

<item name="VALIDATE\_TYPE" value="JNDI"/>

<!-- 容器产品：其值为：jboss,weblogic,websphere -->

<item name="CONTAINER\_PRODUCT" value="jboss"/>

</default>

</Containers>

</Config>

这样获取此jboss容器的JNDI上下文的代码如下：

**package** test.core;

**import** javax.naming.Context;

**import** javax.naming.NamingException;

**import** com.beetle.framework.resource.AppContext;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

**try** {

Context ctx = AppContext.*getRemoteEJBContainerContext*("default");

// ...

} **catch** (NamingException e) {

}

}

}

## 6.2日志组件

日志可以在开发和测试阶段帮助查找和修正错误，也可用于追踪生产环境的问题发生原因。BJAF框架封装了著名的log4j日志工具包（主要是做了性能、功能及易用性的优化，详细请参考BJAF的Api文档），对外统一为AppLogger类作为面板，让日志处理更为简便。

使用日志记录器十分简单，代码如下：

**package** test.core;

**import** com.beetle.framework.log.AppLogger;

**public** **class** Test {

//注册一个日志记录器

**private** **static** AppLogger *logger* = AppLogger.*getInstance*(Test.class);

**public** **static** **void** main(String[] args) {

*logger*.debug("test...");

//...

}

}

另外，框架日志的配置文件为log4j.properties，部署在应用的config目录或classpath目录下即可，其配置内容如下：

##log4j设置DEBUG < INFO < WARN < ERROR < FATAL

log4j.logger.com.beetle=DEBUG,A1,A2

log4j.logger.esbAccessLogger=INFO,A3

#log4j.logger.com.beetle=INFO,A1,A2

#log4j.rootLogger=DEBUG,A1,A2

#log4j.rootLogger=ERROR,A1,A2

#log4j.rootLogger=INFO,A1,A2

#############

# %m 输出代码中指定的消息

# %p 输出优先级，即DEBUG，INFO，WARN，ERROR，FATAL

# %r 输出自应用启动到输出该log信息耗费的毫秒数

# %c 输出所属的类目，通常就是所在类的全名

# %t 输出产生该日志事件的线程名

# %n 输出一个回车换行符，Windows平台为“\r\n”，Unix平台为“\n”

# %d 输出日志时间点的日期或时间，默认格式为ISO8601，也可以在其后指定格式，

# 比如：%d{yyy MMM dd HH:mm:ss,SSS}，输出类似：2002年10月18日 22：10：28，921

# %l 输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数。

#############

#打印到屏幕

log4j.appender.A1=org.apache.log4j.ConsoleAppender

log4j.appender.A1.layout=org.apache.log4j.PatternLayout

log4j.appender.A1.layout.ConversionPattern=%-4r %-5p [%t] %37c %3x - %m%n

#A2--打印到文件config/beetlesyslog.log中。这个文件每天备份一次

log4j.appender.A2=org.apache.log4j.DailyRollingFileAppender

log4j.appender.A2.file=config/beetle.log

log4j.appender.A2.DatePattern='.'yyyy-MM-dd

log4j.appender.A2.layout=org.apache.log4j.PatternLayout

log4j.appender.A2.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} [%t] %37c %3x - %m%n

#A3--打印到文件esbaccess.log中。频繁的日志记录，这个文件每天备份一次

log4j.appender.A3=org.apache.log4j.DailyRollingFileAppender

log4j.appender.A3.ImmediateFlush=false

log4j.appender.A3.BufferedIO=true

log4j.appender.A3.BufferSize=8192

log4j.appender.A3.file=esbaccess.log

log4j.appender.A3.DatePattern='.'yyyy-MM-dd

log4j.appender.A3.layout=org.apache.log4j.PatternLayout

log4j.appender.A3.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS}-- %m%n

#A4--打印到邮件中--只有发生严重错误时才发邮件提醒

log4j.appender.A4=org.apache.log4j.net.SMTPAppender

log4j.appender.A4.Threshold=ERROR

log4j.appender.A4.SMTPHost=

log4j.appender.A4.From=

log4j.appender.A4.To=hdyu@beetlesoft.net

log4j.appender.A4.Subject=error report from beetle System

log4j.appender.A4.layout=org.apache.log4j.PatternLayout

log4j.appender.A4.layout.ConversionPattern=[%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} method:%l%n%m%n

81278576189468.jpglog4j的详细使用及配置说明，请参考其[官方网站](http://logging.apache.org/log4j/)相关资料。

## 6.3邮件发送组件

BJAF框架封装了javax.mail包，简化了邮件发送，支持html格式及附件的发送，例子如下：

**package** test.core;

**import** javax.mail.MessagingException;

**import** com.beetle.framework.util.mail.Letter;

**import** com.beetle.framework.util.mail.SendLetter;

**import** com.beetle.framework.util.mail.SmtpServerInfo;

**public** **class** SendMail {

**public** **static** **void** main(String[] args) {

// 构建smtp服务器信息

SmtpServerInfo ssi = **new** SmtpServerInfo("smptserverhost", "username",

"password");

// 构建一个邮件

Letter letter = **new** Letter();

letter.setSubject("hi,test");

letter.setFrom("from@xx.com");

letter.setTo("to@xxx.com");

letter.setMessage("xxxxx");

// 邮件内容，若发html格式可用letter.setHtmlMessage(arg0)

**try** {

letter.addAttachment("c:\\xxx.doc");// 添加附件

SendLetter.*send*(letter, ssi);

} **catch** (MessagingException e1) {

e1.printStackTrace();

}

}

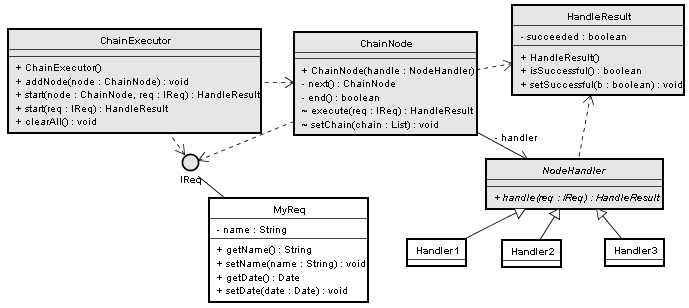
}

目前只支持SMTP类型的邮件服务器。

## 6.4常见设计模式组件

### 6.4.1责任链设计模式

责任链设计模式（Chain of Responsibility）在Java开发中是一种常见又实用的设计模式，其设计意图是：同一个请求，需要按照一定顺序经过不同节点处理，因而连成一条链，并沿着这条链传递此请求，直到处理完它为止。BJAF封装了此设计模式，其结构如下下图：



COR模式类结构图

如图所示，IReq为请求，ChainNode为请求处理链的节点，NodeHandler为节点上此请求的处理器，HandleResult顾名思义是处理器的结果，其决定此请求是否进行传递，ChainExecutor为链的执行器，请求和各个节点都有添加到链中再执行。

示例代码如下：

1. MyReq请求

**package** example.core.cor;

**import** java.util.Date;

**import** com.beetle.framework.util.pattern.cor.IReq;

**public** **class** MyReq **implements** IReq {

**private** String name;

**private** Date date;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Date getDate() {

**return** date;

}

**public** **void** setDate(Date date) {

**this**.date = date;

}

}

1. 编写3个处理器

**package** example.core.cor;

**import** java.util.Date;

**import** com.beetle.framework.util.pattern.cor.HandleResult;

**import** com.beetle.framework.util.pattern.cor.IReq;

**import** com.beetle.framework.util.pattern.cor.NodeHandler;

**public** **class** Handler1 **extends** NodeHandler {

**public** HandleResult handle(IReq req) {

System.*out*.println("-->handler1");

MyReq my = (MyReq) req;

System.*out*.println("-->" + my.getName());

System.*out*.println("-->" + my.getDate());

sleep();

my.setDate(**new** Date(System.*currentTimeMillis*()));

System.*out*.println("-->end");

**return** **new** HandleResult(**false**);

}

**private** **void** sleep() {

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

}

}

}

**package** example.core.cor;

**import** java.util.Date;

**import** com.beetle.framework.util.pattern.cor.HandleResult;

**import** com.beetle.framework.util.pattern.cor.IReq;

**import** com.beetle.framework.util.pattern.cor.NodeHandler;

**public** **class** Handler2 **extends** NodeHandler{

**public** HandleResult handle(IReq req) {

System.*out*.println("-->handler2");

MyReq my = (MyReq) req;

System.*out*.println("-->" + my.getName());

System.*out*.println("-->" + my.getDate());

sleep();

my.setDate(**new** Date(System.*currentTimeMillis*()));

System.*out*.println("-->end");

**return** **new** HandleResult(**false**);

}

**private** **void** sleep() {

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

}

}

}

**package** example.core.cor;

**import** java.util.Date;

**import** com.beetle.framework.util.pattern.cor.HandleResult;

**import** com.beetle.framework.util.pattern.cor.IReq;

**import** com.beetle.framework.util.pattern.cor.NodeHandler;

**public** **class** Handler3 **extends** NodeHandler {

**public** HandleResult handle(IReq req) {

System.*out*.println("-->handler3");

MyReq my = (MyReq) req;

System.*out*.println("-->" + my.getName());

System.*out*.println("-->" + my.getDate());

sleep();

my.setDate(**new** Date(System.*currentTimeMillis*()));

System.*out*.println("-->end");

**return** **new** HandleResult(**false**);

}

**private** **void** sleep() {

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

}

}

}

1. 添加节点并执行

**package** example.core.cor;

**import** com.beetle.framework.util.pattern.cor.ChainExecutor;

**import** com.beetle.framework.util.pattern.cor.ChainNode;

**public** **class** TestClient {

**public** **static** **void** main(String[] args) {

MyReq req = **new** MyReq();

req.setName("Henry");

req.setDate(**new** java.util.Date(System.*currentTimeMillis*()));

ChainNode node1 = **new** ChainNode(**new** Handler1());

ChainNode node2 = **new** ChainNode(**new** Handler2());

ChainNode node3 = **new** ChainNode(**new** Handler3());

ChainExecutor executer = **new** ChainExecutor();

executer.addNode(node1);

executer.addNode(node2);

executer.addNode(node3);

executer.start(req);

executer.clearAll();

}

}

运行结果如下：

-->handler1

-->Henry

-->Fri Oct 31 17:58:18 CST 2008

-->end

-->handler2

-->Henry

-->Fri Oct 31 17:58:19 CST 2008

-->end

-->handler3

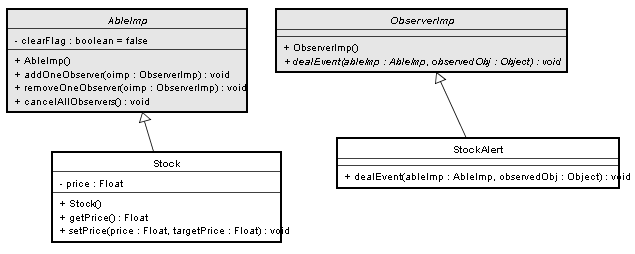
-->Henry

-->Fri Oct 31 17:58:20 CST 2008

-->end

### 6.4.2观察者设计模式

观察者模式意图是：定义对象间一对多的依赖关系，当某个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。BJAF对此模式进行封装



观察者模式类图

例如：股民设置某只股票，当股票跌到某个价位，系统会自动给此股民发一个报警短信。采用观察者模式实现如下：

1. 建立股票对象Stock

**package** example.core.observer;

**import** com.beetle.framework.util.pattern.observer.AbleImp;

**public** **class** Stock **extends** AbleImp {// 扩展AbleImp以便Stock对象的属性可监控

**public** Stock() {

**super**();

}

**private** Float price;// 股票价格

**public** Float getPrice() {

**return** price;

}

**public** **void** setPrice(Float price, Float targetPrice) {

**this**.price = price;

**if** (**this**.price.floatValue() <= targetPrice.floatValue()) {// 报警条件（股价跌到目标价）

**this**.observeOneObject(**this**.price);// 设置price字段对象为报警对象

}

}

}

1. 建立股票报警对象StockAlert处理报警事件

**package** example.core.observer;

**import** com.beetle.framework.util.pattern.observer.AbleImp;

**import** com.beetle.framework.util.pattern.observer.ObserverImp;

**public** **class** StockAlert **extends** ObserverImp {// 股票报警类，主要处理报警事件

**public** **void** dealEvent(AbleImp ableImp, Object observedObj) {

Stock stock = (Stock) ableImp;

System.*out*.println("price:" + stock.getPrice());

System.*out*.println("send a sms to the stock's owner!");// 发短信通知客户

}

}

1. 模拟执行

**package** example.core.observer;

**public** **class** TestClient {

**public** **static** **void** main(String[] args) {

StockAlert alert=**new** StockAlert();

Stock stock=**new** Stock();

stock.addOneObserver(alert);//注册报警事件

stock.setPrice(**new** Float(12.5f), **new** Float(6.5f));//股票价位正常

System.*out*.println("---");

stock.setPrice(**new** Float(4.5f), **new** Float(6.5f));//股票价位跌破目标价，自动触发事件

System.*out*.println("---");

}

}

执行结果：

---

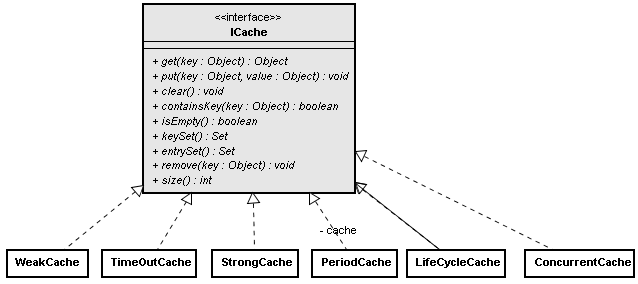
price:4.5

send a sms to the stock's owner!

---

## 6.5简单缓存器

BJAF针对常见应用场景，封装了几个简单缓存器实现方便编程，其类图如下：



各种缓存器类图

|  |  |
| --- | --- |
| **缓存器** | **应用说明** |
| StrongCache | 缓存器内所有存储的值需要手工清除； |
| WeakCache | 缓存器内的值如果没有人为清除的话，在此值不再引用是，由jvm自己清除。清除时间由jvm来决定。 |
| TimeOutCache | 类似于WeakCache，缓存器内的值如果没有人为清除的话，在此值不再引用是，由jvm自己清除。清除时间由jvm来决定；其另外一个特性是，在给定时间内，通过get方法获取值都如常返回，若超过此时间，缓存器在调用此方法是会清空此值，返回一个null值 |
| LifeCycleCache | 会自己管理缓存器内值的生命周期，值超过给定的时间会自动清除，无需人工干预；此缓存器适合多线程访问场景。不允许保存null值 |
| ConcurrentCache | 功能上类似于StrongCache，此缓存器适合多线程访问场景。不允许保存null值 |

## 6.6其它工具类

框架还包括其它的工具类，如：文件操作，Class操作，动态数组，字符表达式，文档模板，流量管道控制模型，消息发布订阅模型等等。都统一在Util包下。这里就不再逐一赘述。

# 7.BJAF应用部署

采用BJAF框架开发出来的应用系统是标准的、典型的三层架构应用[1]，也是标准的J2EE规范应用；所以，BJAF应用部署支持标准J2EE应用部署模式，同时，也可以适应用户不同部署设计方案，十分灵活。

## 7.1灵活部署方案

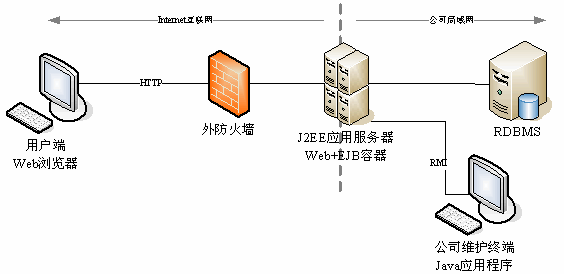
关于BJAF应用部署的说明及部署方案的详细设计及讨论，请参考《J2EE应用框架设计与项目开发》[2]第10章应用部署小节。

[1]指把逻辑分别表示层、业务层和持久层来设计的应用系统。

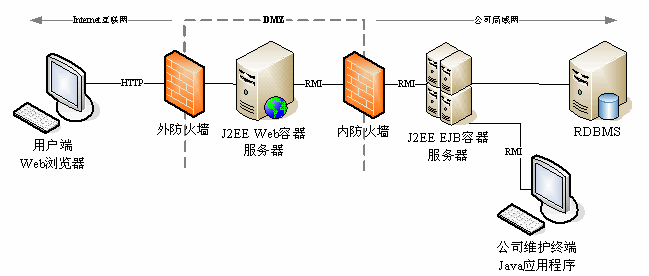
[2]余浩东，J2EE应用框架设计与项目开发，清华大学出版社，2007.

下面给出示意：

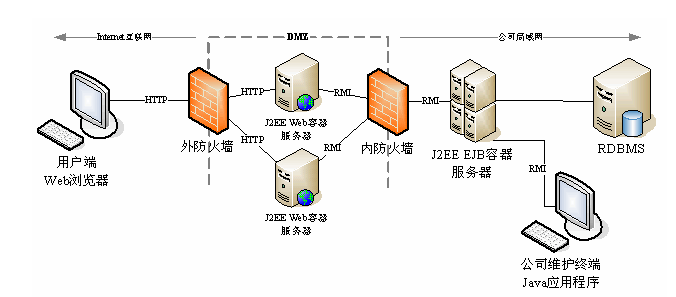
### 7.1.1 集中式



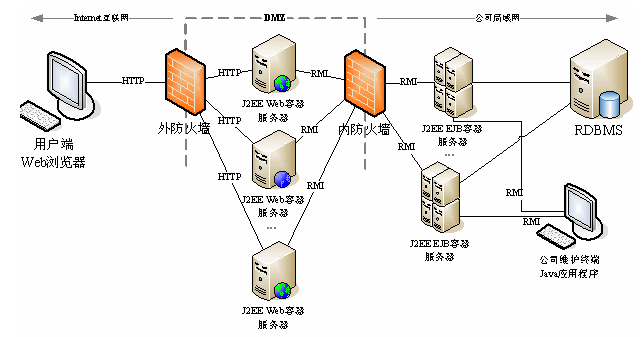
### 7.1.2 Web层和业务层分离



### 7.1.3 多Web层



### 7.1.4 多web层多业务层



对于以上部署方法，使用BJAF框架开发的应用只需设置一个配置文件参数即可支持，无需改动任何代码。

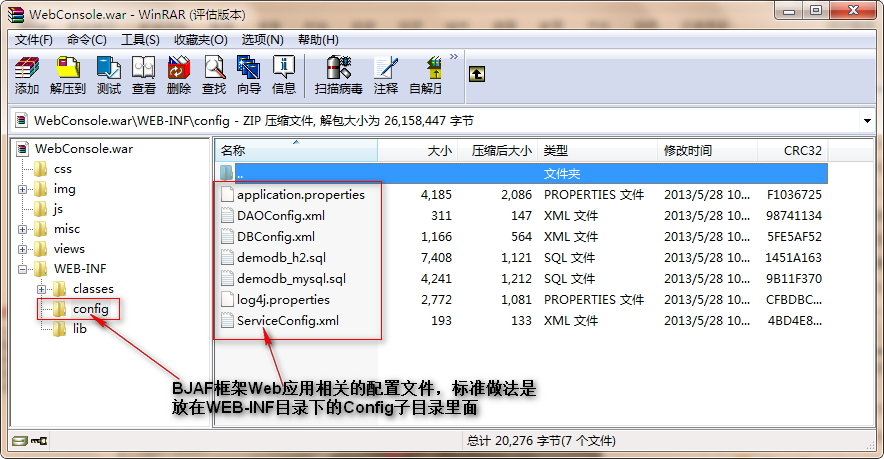


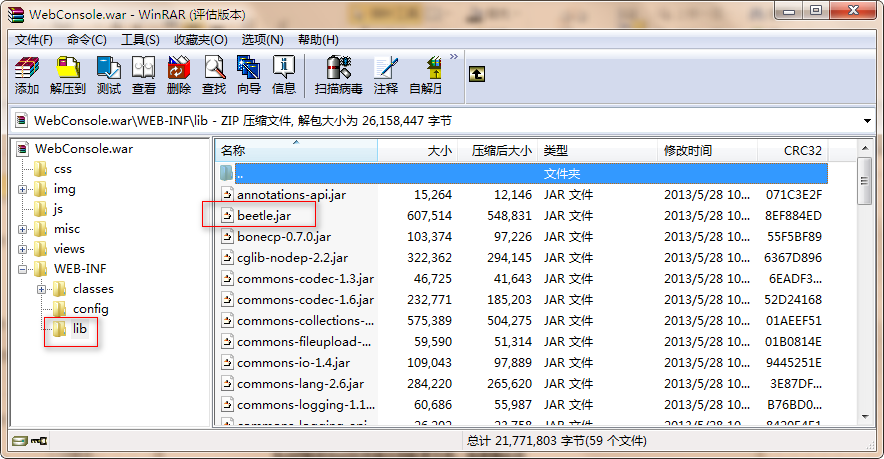
## 7.2独立打包配置文件存放目录截图示意

附注：关于BJAF框架开发应用打包相关配置文件存放路径的问题，事实上，BJAF框架配置文件较为灵活的，基本上以放置在对应逻辑层的classpath路径下为原则就能够正确读取到。另外，如果想手动指定配置的文件路径，则可以通过jvm参数[***-Dbeetle.application.properties.file.path*=d://xxx//yyy**]指定其存放目录，详细请参考6.1.1章节。

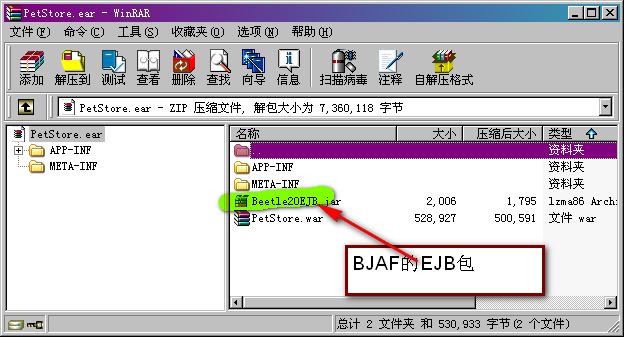
下面分别针对WAR应用、EAR应用和独立进程部署包截图，以示参考：

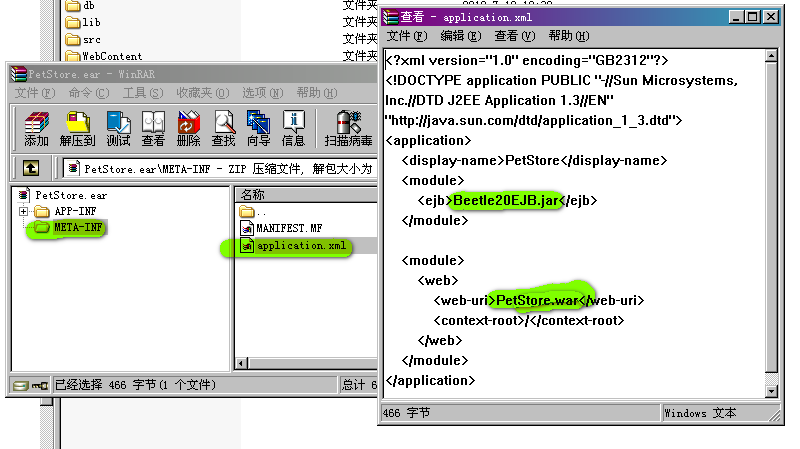
### 7.2.1 WAR应用截图

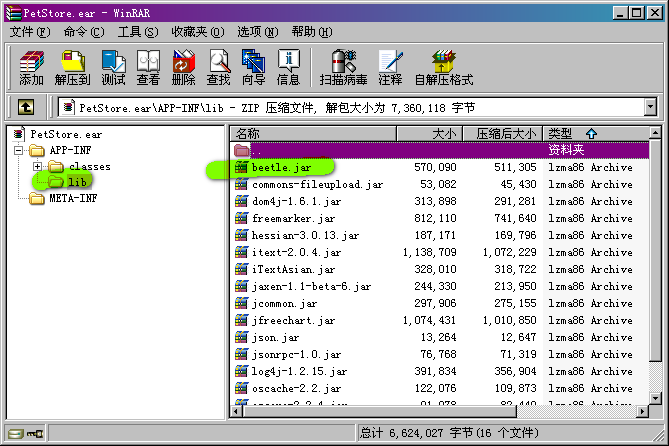


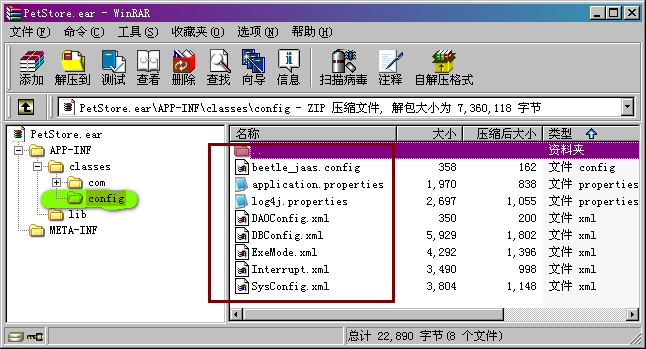


### 7.2.2 EAR应用截图

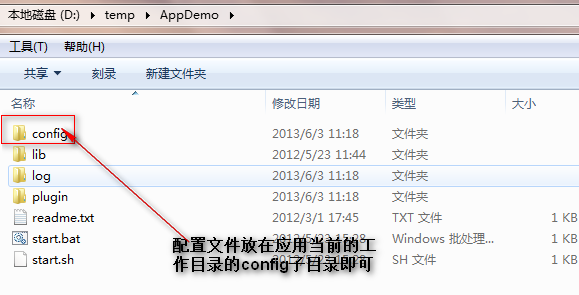








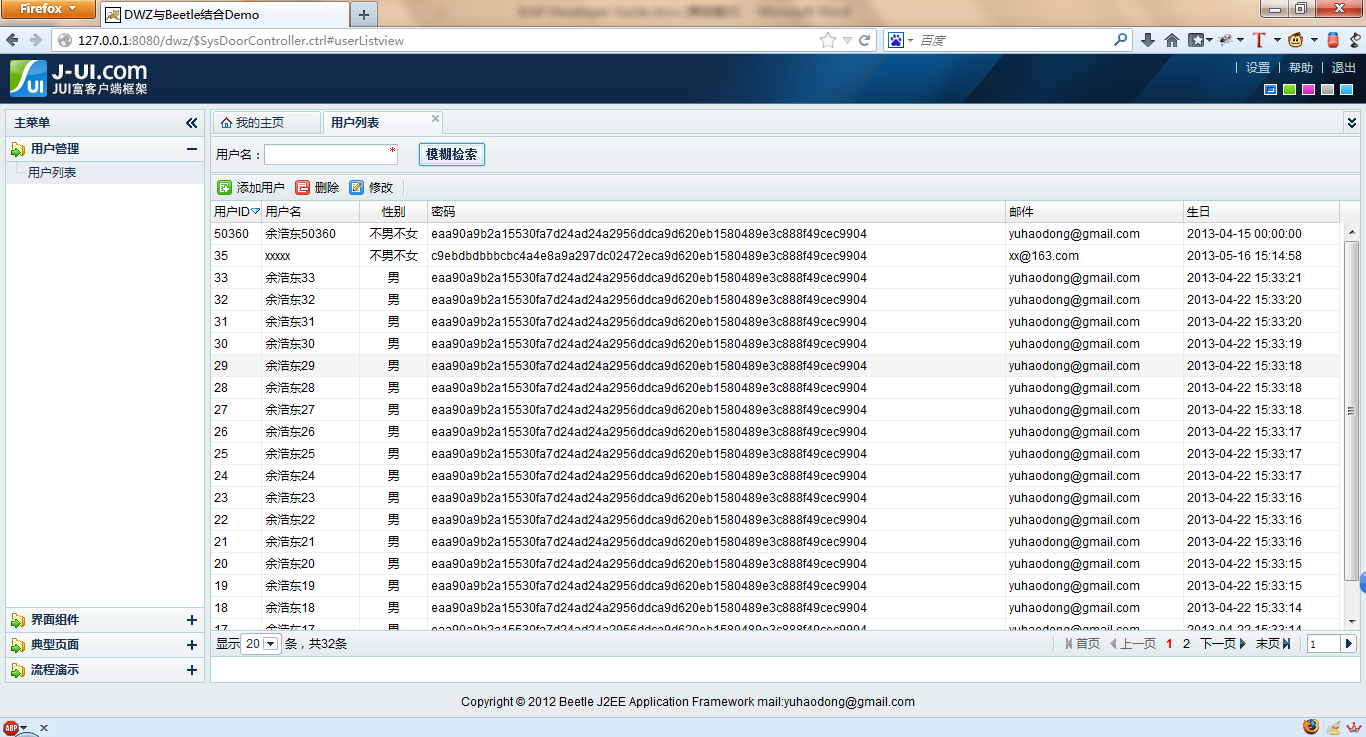
### 7.2.3 独立进程应用截图



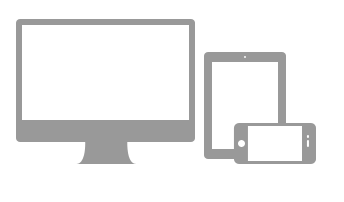
# 8.后记

如果你只是开发一个XXX信息管理系统，那么恭喜你，BJAF框架可以轻松解决你的问题。你可以直接参考SDK里面的例子模板（例如DWZ）在例子基础上直接开发。例如其界面截图：

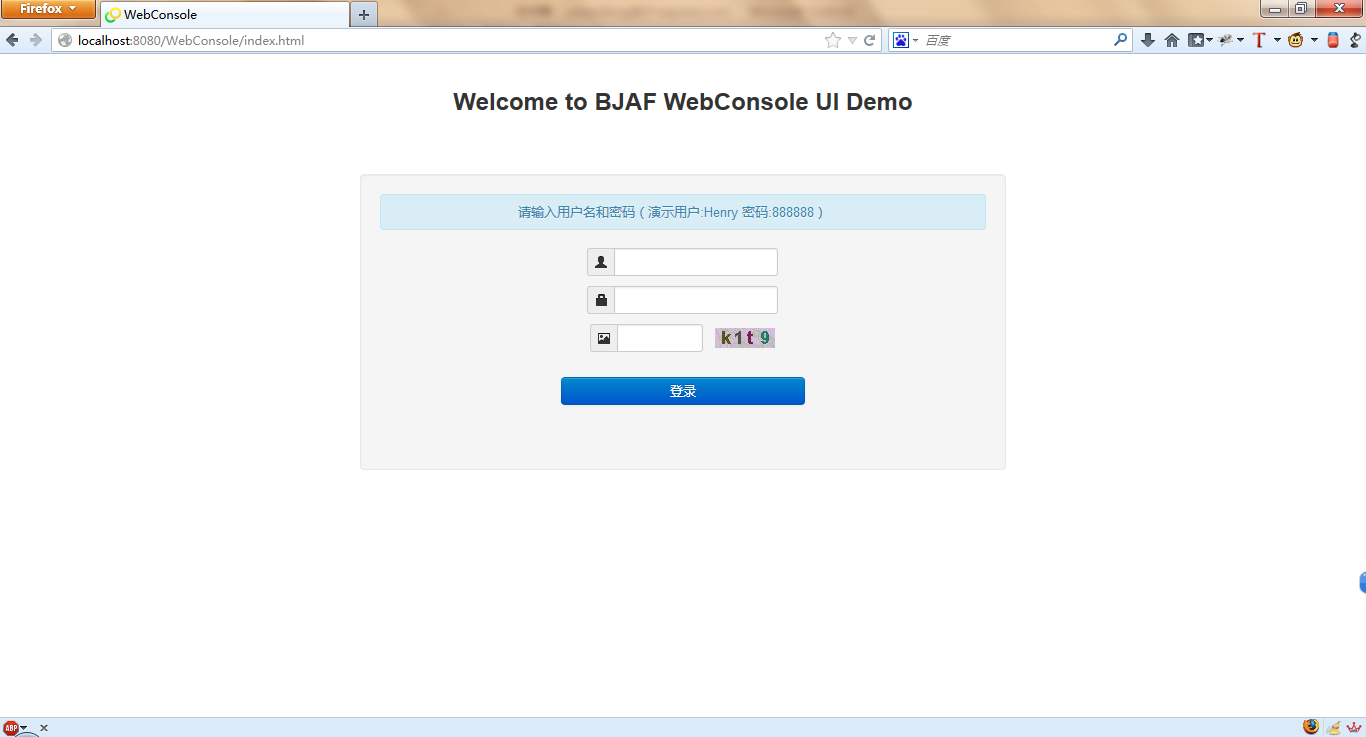


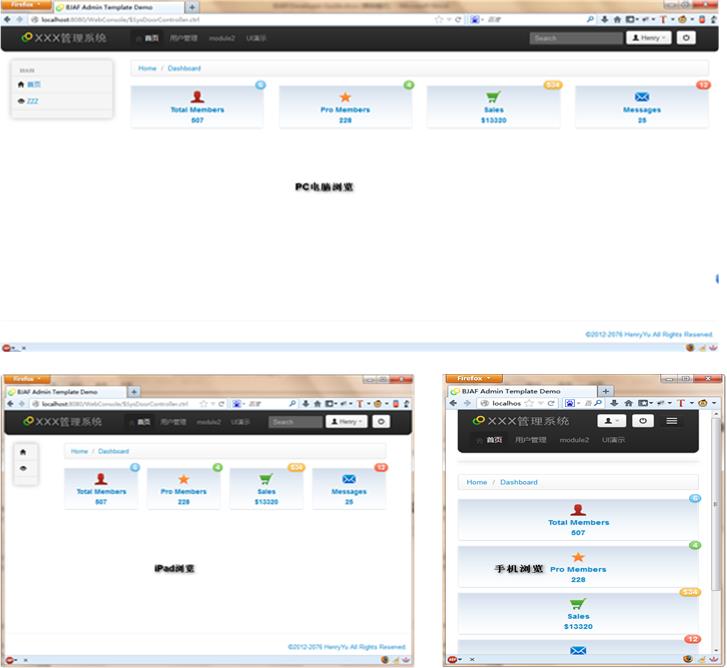


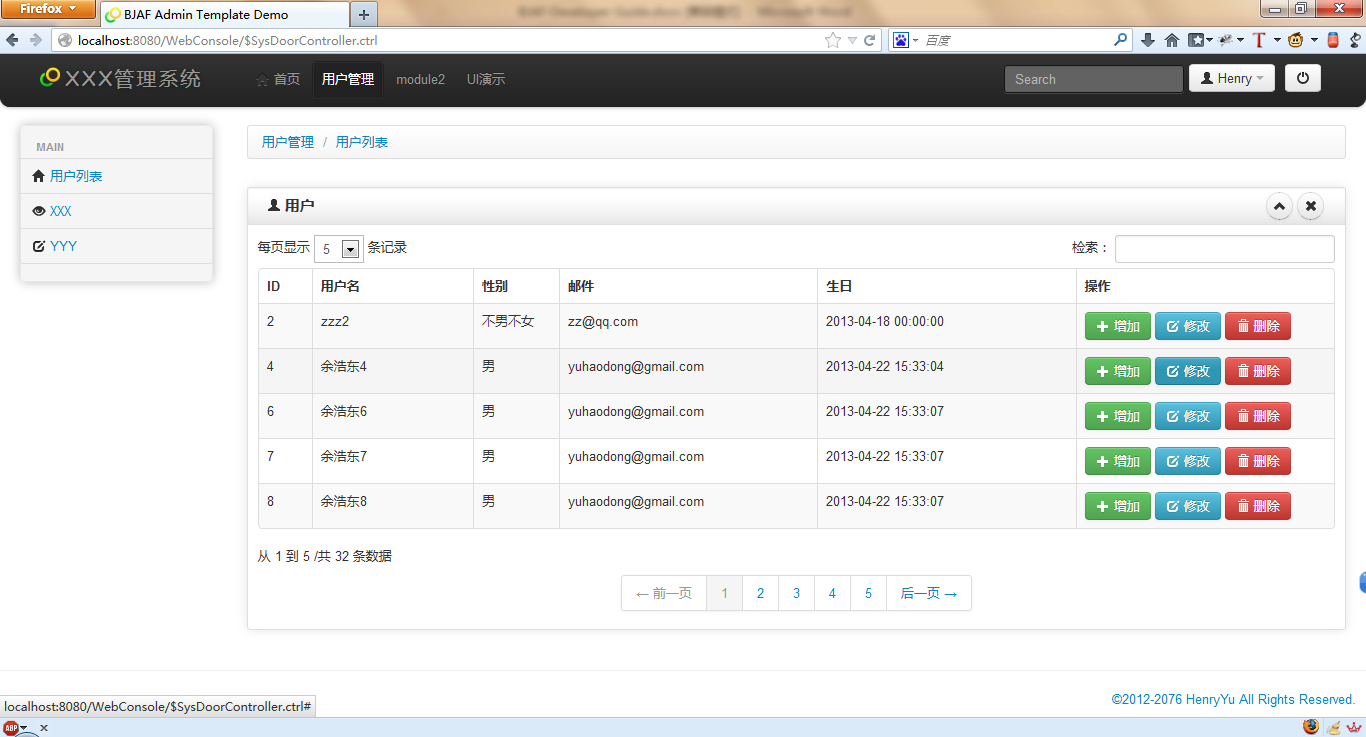
如果你要拥抱HTML5技术，支持响应式系统页面设计（即自动根据浏览设备的尺寸来响应页面内容，达到不同设备，如电脑、平板和手机都有最佳的浏览体验），如：



BJAF SDK中的WebConsole例子是基于Bootstrap技术开发后台管理控制台模板。其截图如下



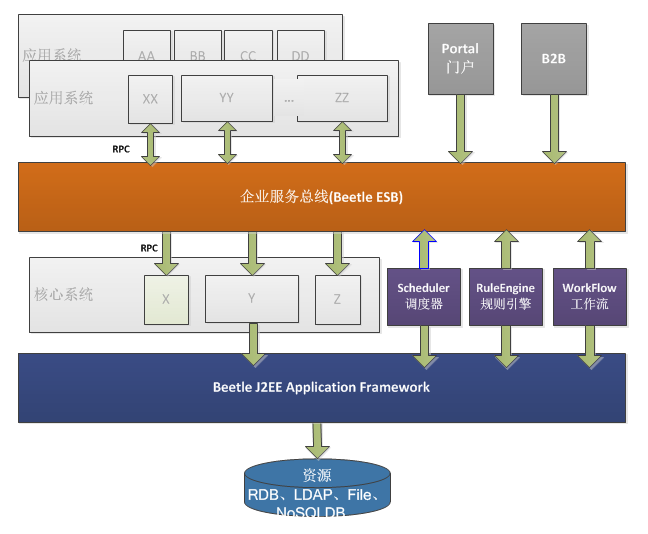




81278576189468.jpgDWZ和WebConsole开发模板对Java开发人员来说的最大的好处在于：开发人员无需熟悉（或要求懂太多）Javascript即可进行Ajax开发，如果开发人员原来习惯使用 Jsp来开发视图页面的话，这里可以直接援用原有的JSP视图技术，因为BJAF框架已经在前端利用js透明封装了。

如果你不仅仅是开发1个系统或产品，你面临的是十几个，甚至成百上千个子系统，那么恭喜你，故事才刚刚开始O(∩\_∩)O~。你要解决它们之间的通信的健壮性、梳理它们的调用关系；面对众多的服务，你还要考虑如何是治理它们？你可能想到SOA，但面对市场众多厂商的SOA的解决方案和相关的产品，哪个适合你，能解决你具体的问题？最悲屈的事情，花了大价钱却还解决不了问题。。。

BJAF给出的参考架构是：

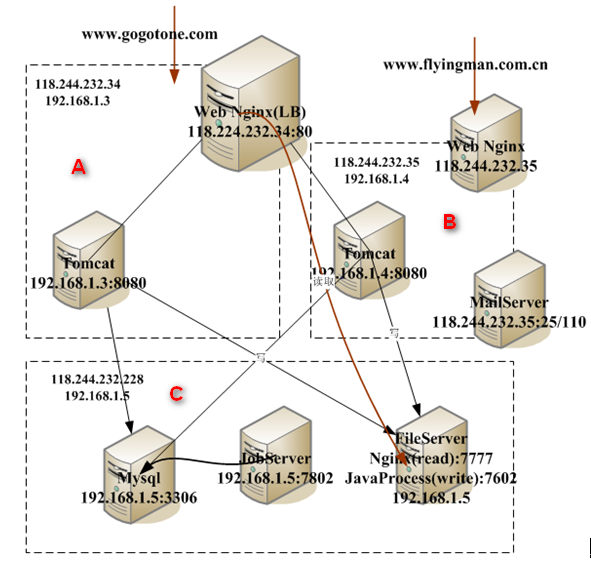


当然BJAF框架不能解决你所有的问题，但我们的RPC可以有效帮你解决通信的可靠性、可用性和通信效率问题；我们的基于RPC的服务总线ESB（不包括在BJAF的SDK中，另外提供）可以有效解耦各个子系统互相调用的依赖关系，提供诸如：服务注册、监控和通信规则（控制服务并发量、数据大小等）等常见治理的能力。同时，我们在BPM、IAM、CEP、Scheduler、RuleEngine等企业级应用组件解决方案方面也累积了丰富的实施经验。

如果你开发的是一个网站，那么恭喜你。BJAF可以说为互联网而生的实用性框架。因为其针对网站高性能、高可用和灵活扩展性等方面的技术架构挑战做了很多针对性的设计和优化（例如：缓存、多线程并发编程等）。我们对网站技术上挑战和架构设计的理解请参考：

<http://wenku.baidu.com/view/a66efd23dd36a32d737581d6?fr=prin>

也就是说使用BJAF框架来开发，无论是基于现有的开源容器（如：Tomcat）还是自己根据需要构建自己的应用服务器（参考第5章节）都是很容易，框架的基础保证了你网站架构灵活的扩展性。例如：下面是我们近期完成小型网站案例（架构示意图）



我们使用3台VPS（4核4GB内存）轻松通过1W并发用户压力测试。

除了开发上的实现问题，我们平时还遇到哪些烦心事？Tomcat性能上不去？动不动就出现OM内存泄露错误？如何定位这样问题？如何对一个Java进程进行调优？请参考SDK附件文档《Java进程JVM参数调优指导》。

对我们的框架你想了解更多或对我们的方案有兴趣？

需要我们提供技术支持或咨询服务，请联系我们：[yuhaodong@gmail.com](mailto:yuhaodong@gmail.com)