Beetle J2EE Application Framework

开发指南

（Version **1.4.0**）

网站：www.beetlesoft.net

邮件：hdyu@beetlesoft.net

yuhaodong@gmail.com

版本信息			
日期	**版本**	**作者**	**版本说明**
2007-10-20	1.0	余浩东	创建
2009-3-14	1.3.5	余浩东	修改、完善
2009-5-31	1.3.6	余浩东	更新1.3.6内容
2009-10-1	1.3.7	余浩东	更新1.3.7内容
2009-11-24	1.3.8	余浩东	更新1.3.8内容

2010-07-06	1.4.0	余浩东	更新1.4.0内容

目录

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。

BJAF 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。

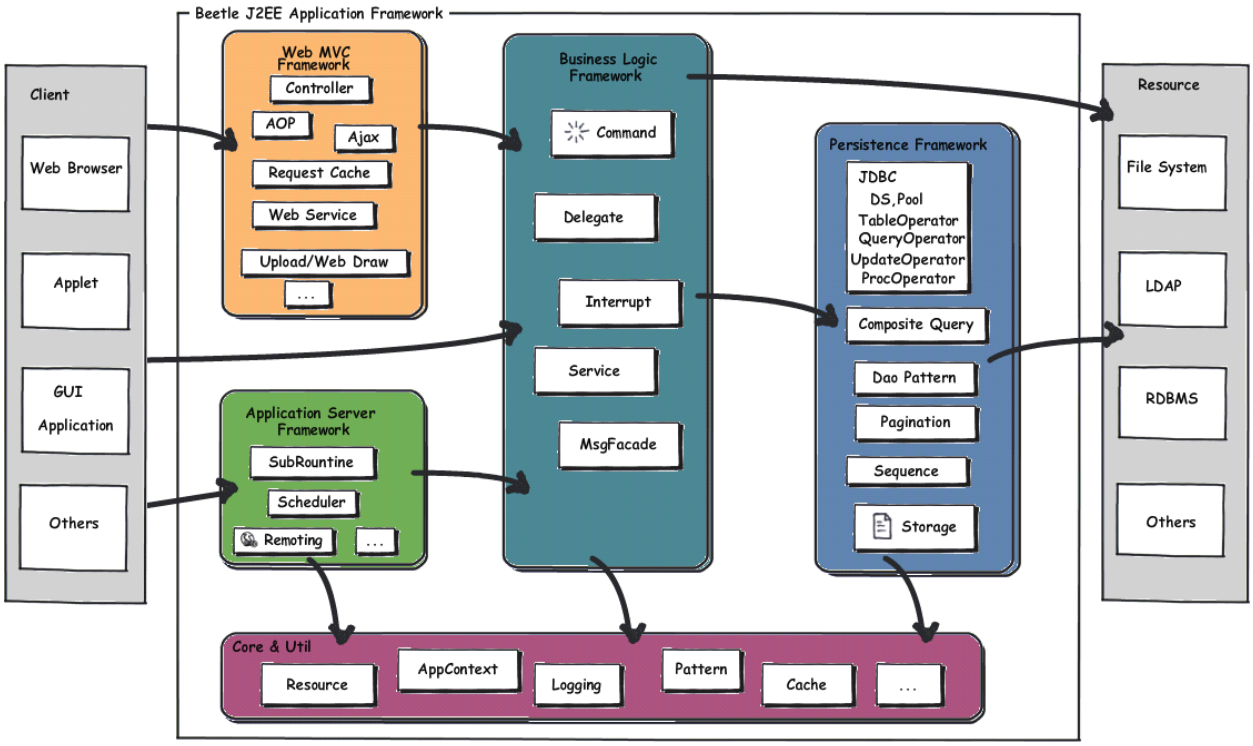


图 1-1 BJAF 整体架构

运行环境要求

BJAF 要求使用 JDK1.4 或以上。

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：

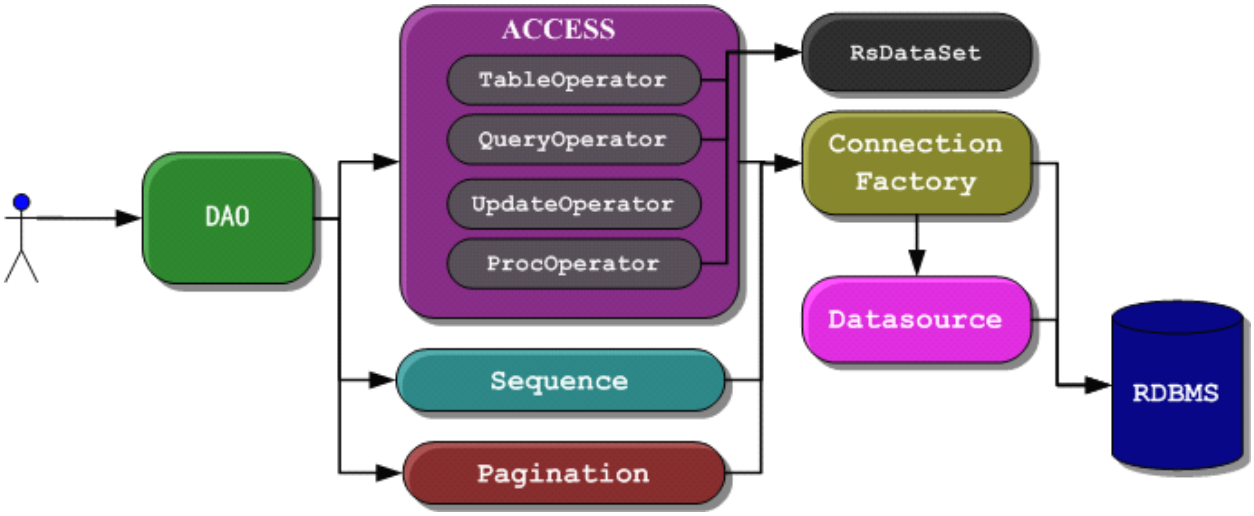


图 2-1 数据存取框架组件示意图

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。

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 3 种数据源：

自带数据源种类	说明
DriverPool	driver本地驱动连接池，框架自实现，功能简单但性能优秀，不支持XA；便于开发测试；开发本地应用，建议采用
GenePool	实现javax.sql.DataSource接口。
XaPool	实现javax.sql.XADataSource接口，支持分布式事务。
ProXool	ProXool普通连接池，性能优秀，较流行

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：

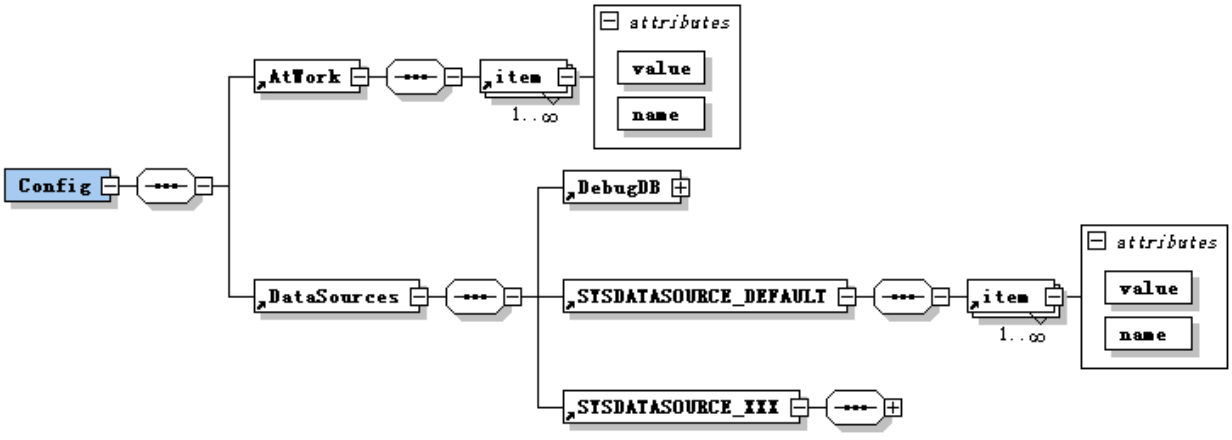


图 2-2 DBConfig.xml 文件格式

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

```
<Config>
	<AtWork>
   <!--
	1-使用 J2EE 容器数据源;
	2-使用本框架自带数据源;
	3-使用调试数据源
    -->
		<item name="DATASOURCE_USE_FLAG" value="2"/>
<!--
	  框架序列号实现类型
		1-通用实现；2-Oracle 实现；3-DB2 实现；4-POSTGRESQL 实现；10-其它
	  -->
		<item name="SEQUENCE_IMP_TYPE" value="1"/>
    <!--
	[使用分页的数据库标记，分页实现会根据此标记自动匹配相应数据库的实现。]
	1-oracle;2-sqlserver;3-mysql;4-sybase;5-db2;6-PostgreSql;7-
InterBase(Firebird)
    -->
		<item name="PAGINATION_DBMS_FLAG" value="2"/>
    <!--
	系统数据源定义
	*必须拥有一个[SYSDATASOURCE_DEFAULT]默认数据源
	*对于使用 J2EE 容器数据源（DATASOURCE_USE_FLAG=1）的情况，value 应为此数据源的
JNDI 名称;
	*对于使用本框架自带数据源（DATASOURCE_USE_FLAG=2）的情况，value 随便定义（此时
value 值没有意义）
	*多个数据源的情况，直接定义多项就可以了，命名应该符合[SYSDATASOURCE_XXXX]的形式。
    -->
		<item name="SYSDATASOURCE_DEFAULT" value="sys_db_default"/>
		<!--  添加别的数据源 -->
		<item name="SYSDATASOURCE_XXX" value="xxxx1"/>
```

```
        </AtWork>
        <DataSources>
                <DebugDB>
                    <item name="connection-url"
                    value="jdbc:oracle:thin:@127.0.0.1:1521:yhdsid"/>
                <item name="driver-class"
 value="oracle.jdbc.driver.OracleDriver"/>
                <item name="user-name" value="scott"/>
                <item name="password" value="tiger"/>
                </DebugDB>
                <SYSDATASOURCE_DEFAULT>
        <!--
            数据库连接池类型
            1-
driver[com.beetle.framework.persistence.access.datasource.DriverPool]本地驱动连
接池，框架自实现，功能简单但性能优秀；
            2-general[com.beetle.framework.persistence.access.datasource.GenePool]
普通连接池，为 jotm 库自带；
            3-jotm 自带 Xa 连接池
[com.beetle.framework.persistence.access.datasource.XaPool];
            4-
ProXool[com.beetle.framework.persistence.access.datasource.ProxoolPool]普通连接
池，性能优秀，较流行
            此属性在 beetle1.3.x 版本已经不推荐使用，用 pool-imp 属性代替
            保留此属性只为兼容以前版本；若此属性与 pool-imp 同时配置，以
        pool-imp 属性为主
            -->
                        <item name="pool-type" value="1"/>
                        <!--连接池实现具体实现类 -->
                        <item name="pool-imp"
value="com.beetle.framework.persistence.access.datasource.DriverPool"/>
                        <item name="pool-minsize" value="10"/>
                        <item name="pool-maxsize" value="15"/>
```

```
                    <!-- 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 CURRENT_DATE"/>
                    <item name="connection-url"
value="jdbc:mysql://127.0.0.1:3306/ikbs?useUnicode=true&amp;characterEncoding
=GBK"/>
                    <item name="driver-class" value="org.gjt.mm.mysql.Driver"/>
                    <item name="user-name" value="root"/>
                    <item name="password" value="iamhenry"/>
            </SYSDATASOURCE_DEFAULT>
            <SYSDATASOURCE_XXX>
                    <item name="pool-type" value="1"/>
                    <item name="pool-imp"
value="com.beetle.framework.persistence.access.datasource.DriverPool"/>
                    <item name="pool-minsize" value="10"/>
                    <item name="pool-maxsize" value="50"/>
                    <item name="test-sql" value="select CURRENT_DATE"/>
                    <item name="connection-url"
value="jdbc:oracle:thin:@10.25.10.175:1555:rmtest"/>
                    <item name="driver-class"
value="oracle.jdbc.driver.OracleDriver"/>
                    <item name="user-name" value="rmdata"/>
                    <item name="password" value="hhxxttxs"/>
            </SYSDATASOURCE_XXX>
      </DataSources>
</Config>
```

配置文件中，分为两大部分：

<AtWork>在此标记内的数据源，为当前工作数据源，ConnectionFacatory 组件只识别这些数据源，标签内容说明见下表：

标记符号	意义说明
DATASOURCE_USE_FLAG	数据源使用标记，其值含义如下： **1**-使用j2ee容器数据源;

	2-使用本框架数据源; **3**-使用调试数据源
PAGINATION_DBMS_FLAG	使用分页的数据库标记，分页实现会根据此标记自动匹配相应数据库的实现。其值含义如下： **1**-oracle;**2**-sqlserver;**3**-mysql;**4**-sybase; **5**-db2;**6**-PostgreSql;**7**-InterBase(Firebird)
SYSDATASOURCE_DEFAULT	系统默认数据源，一个应用必须有一个默认数据源，其标签名称不能改动。其值含义： 如果此数据源使用的是j2ee容器数据源，则其值为此数据源在这个容器里面配置的JNDI名称；如果使用非j2ee容器数据源，则此值随便定义。关于数据源的详细属性配置，在文件的**<DataSources>**的同名标记下
SYSDATASOURCE_XXX	添加多一个或多个数据源的话，可以自定义标记，例如： <item name="SYSDATASOURCE_XXX" value="xxxx1"/>，SYSDATASOURCE_XXX的名字可以随自己喜欢定义；其值的定义和上面SYSDATASOURCE_DEFAULT一样。同时，如果此数据源为非j2ee容器数据源，则需要在**<DataSources>**下定义同名的属性配置。

<DataSources>为本框架自己的各种数据源属性定义标签，其包含了一个**<DebugDB>**，定义了框架代码调试的数据源，其它标签为各种数据源的定义。数据源标签的名称，可以自定义，如果定义好此数据源，需要在系统框架里面使用，则必须在**<AtWork>**标签里面标明。标签内容说明如下：

标记符号	意义说明
pool-type	连接池类型； 为本框架上面实现的数据库连接池类型，其值含义为： **1-**driver[com.beetle.framework.persistence.access.datasource.DriverPool]本地驱动连接池，框架自实现，功能简单但性能优秀; **2-**general[com.beetle.framework.persistence.access.datasource.GenePool]普通连接池，为 jotm 库自带; **3-**jotm 自带 Xa 连接池[com.beetle.framework.persistence.access.datasource.XaPool]; **4-**ProXool[com.beetle.framework.persistence.access.datasource.ProxoolPool]普通连接池，性能优秀，较流行 此属性在 beetle1.3.x 版本已经不推荐使用，用 pool-imp 属性代替 保留此属性只为兼容以前版本；若此属性与 pool-imp 同时配置，以 pool-imp 属性为主

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

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）和存储过程操作器（ProcOperator），它们组件关系类图如下：

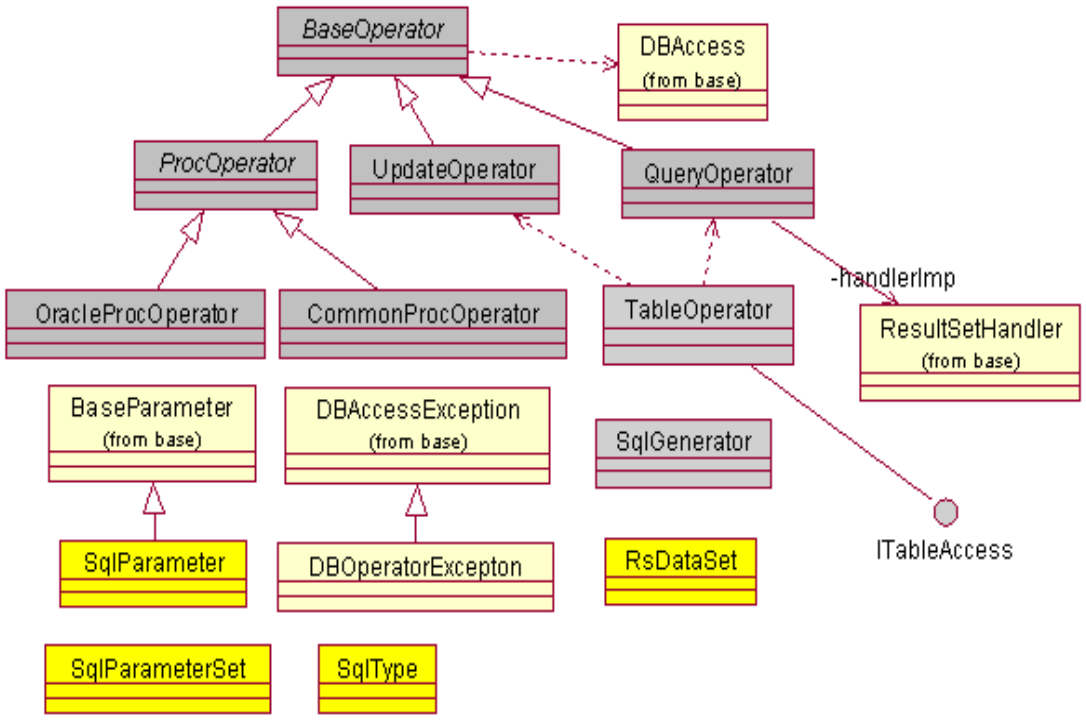


图 2-3 数据存取组件类图

下面就各组件进行详细介绍说明：

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）。其 UML 类图如下：

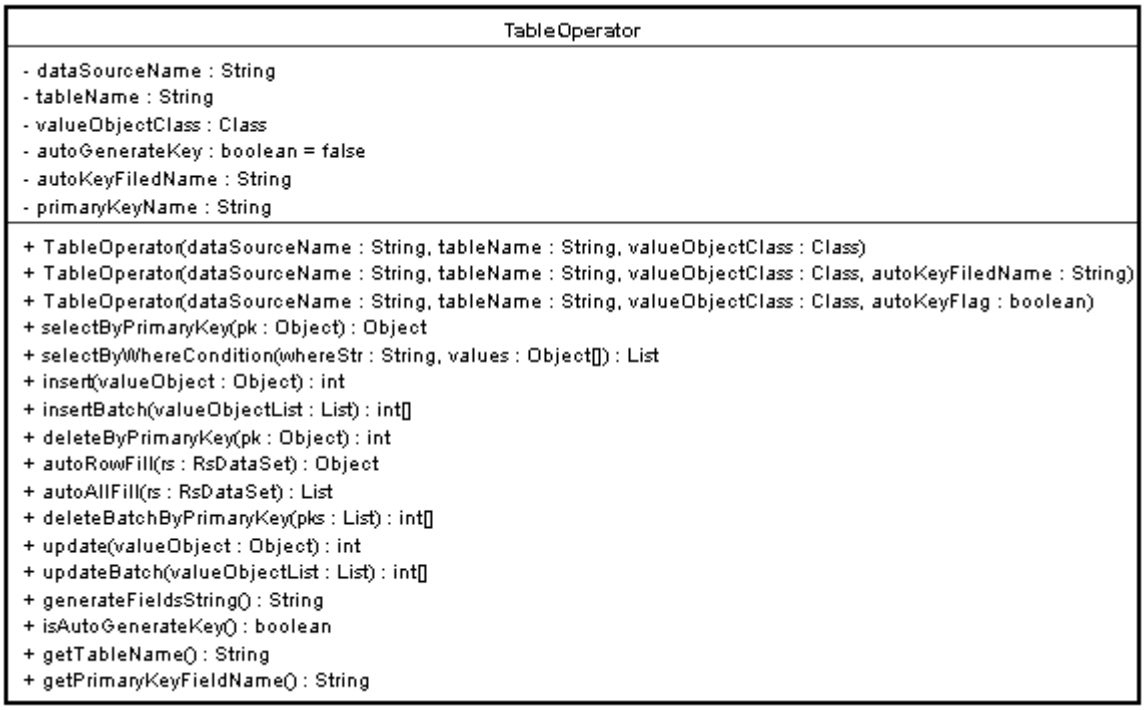


图 2-4 单表操作器类图

从上图可知，TableOperator 提供以下主要功能方法：

方法与属性	功能说明
TableOperator(dataSourceName: String, tableName: String, valueObjectClass: Class)	构造函数，适用于表的主键为非自动增量的情况。 参数： dataSourceName-- 数据源名称 tableName--数据库表名称（大小写敏感） valueObjectClass--表对应的值对象
TableOperator(dataSourceName: String, tableName: String,	构造函数，适用于表的主键为自动增量的情况。 参数：

valueObjectClass: Class, autoKeyFlag: boolean)	dataSourceName-- 数据源名称 tableName--数据库表名称（大小写敏感） valueObjectClass--表对应的值对象 autoKeyFlag--主键是否为自动增量的，true为是
TableOperator(dataSourceName: String, tableName: String, valueObjectClass: Class, autoKeyFiledName: String)	构造函数，适用于表的唯一标识（此唯一标识不是主键）为自动增量的情况。 参数同上，autoKeyFiledName为自动增长字段名。
selectByPrimaryKey(pk :Object): Object	根据主键查找单表中所对应此主键的记录，并封装成此单表值对象返回。 参数： pk--主键 返回： 单表值对象
selectByWhereCondition(whereStr: String, values: Object []):List	根据where条件，查询此单表的记录。 参数： whereStr—Sql中where条件子语句 values—Where条件语句参数，无参数则为null 返回： 单表值对象列表，如果记录为空，则列表的size为0
insert(valueObject: Object): int	插入一条记录。 参数： valueobject—表值对象 返回： int—操作影响条数
insertBatch(valueObjectList: List): int[]	批量插入记录。封装jdbc批量插入接口 参数： valueObjectList–表值对象列表 返回： int[]–每个操作影响条数数组

deleteByPrimaryKey(pk:Object):int	根据主键删除记录 参数： pk—主键 返回： int—操作影响条数
deleteBatchByPrimaryKey(pks: List):int[]	根据主键批量删除记录 参数： pks—主键列表 返回： int[]-每个操作影响条数数组
update(valueObject:Object):int	更新表记录 参数： valueobject—表值对象 返回： int—操作影响条数
updateBatch(valueObjectList: List): int[]	批量更新记录 参数： valueObjectList–表值对象列表 返回： int[]–每个操作影响条数数组
autoRowFill(rs:RsDataSet):Object[1]	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） 参数： rs—框架结果处理对象 返回： 已填充的表值对象
autoAllFill(rs:RsDataSet):List [2]	自动填充所有的结果 参数： rs—框架结果处理对象 返回： 已填充的表值对象列表

[1] [2] 为单表操作的记录集辅助处理方法。另外注意的是：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 雇员表为示例

scott 用户数据表 E-R 关系图如下：

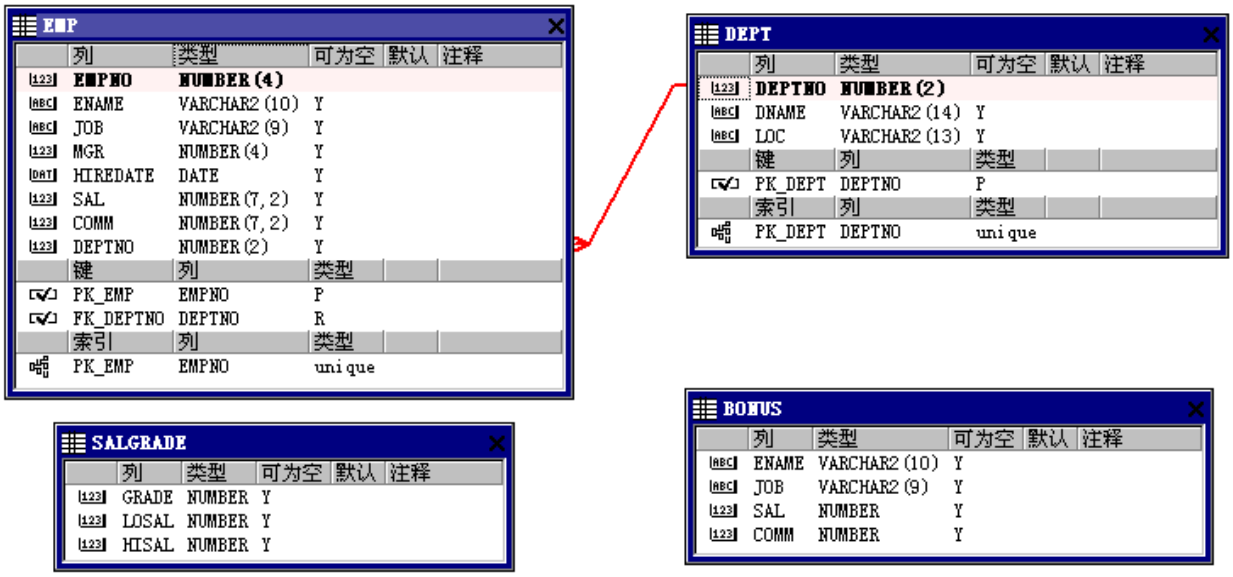


图 2-5scott 用户数据表 E-R 关系图

EMP 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      300.00    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
```

图 2-6emp 表数据截图

示例代码：

（1），与 EMP 表相对应建立一个 java 值对象：

```
package psdemo.valueobject;

import java.io.Serializable;

public class Emp implements Serializable {
	private static final long serialVersionUID = 323837418188386358L;

	private Integer empNo;

	private String ename;

	private String job;

	private Integer mgr;

	private java.sql.Timestamp hireDate;

	private Float sal;

	private Float comm;

	private Integer deptNo;

	public Float getComm() {
		return comm;
	}

	public void setComm(Float comm) {
```

```
		this.comm = comm;
	}

	public Integer getDeptNo() {
		return deptNo;
	}

	public void setDeptNo(Integer deptNo) {
		this.deptNo = deptNo;
	}

	public Integer getEmpNo() {
		return empNo;
	}

	public void setEmpNo(Integer empNo) {
		this.empNo = empNo;
	}

	public String getEname() {
		return ename;
	}

	public void setEname(String ename) {
		this.ename = ename;
	}

	public java.sql.Timestamp getHireDate() {
		return hireDate;
	}
```

```
	public void setHireDate(java.sql.Timestamp hireDate) {
		this.hireDate = hireDate;
	}

	public String getJob() {
		return job;
	}

	public void setJob(String job) {
		this.job = job;
	}

	public Integer getMgr() {
		return mgr;
	}

	public void setMgr(Integer mgr) {
		this.mgr = mgr;
	}

	public Float getSal() {
		return sal;
	}

	public void setSal(Float sal) {
		this.sal = sal;
	}
}
```

（2），利用 TableOperator 对表 EMP 进行存取操作：

```
public static void main(String[] args) {
	TableOperator empOptr = new TableOperator(Const.dataSourceName, "EMP",
			Emp.class);
	// 查找编号（empNo）为 7499 的员工信息
	Emp emp = (Emp) empOptr.selectByPrimaryKey(new Integer(7499));
	System.out.println(emp.getSal());// 打印查到的数据
	// 把此员工的薪水修改为 300.25
	emp.setSal(new Float(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 Integer(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 empList = empOptr.selectByWhereCondition("where sal>=?", params);
	if (!empList.isEmpty()) {
		for (int i = 0; i < empList.size(); i++) {
			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();

}
```

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。

查询操作器

查询操作器 QueryOperator 的使用很简单，其 UML 公共方法类图如下：

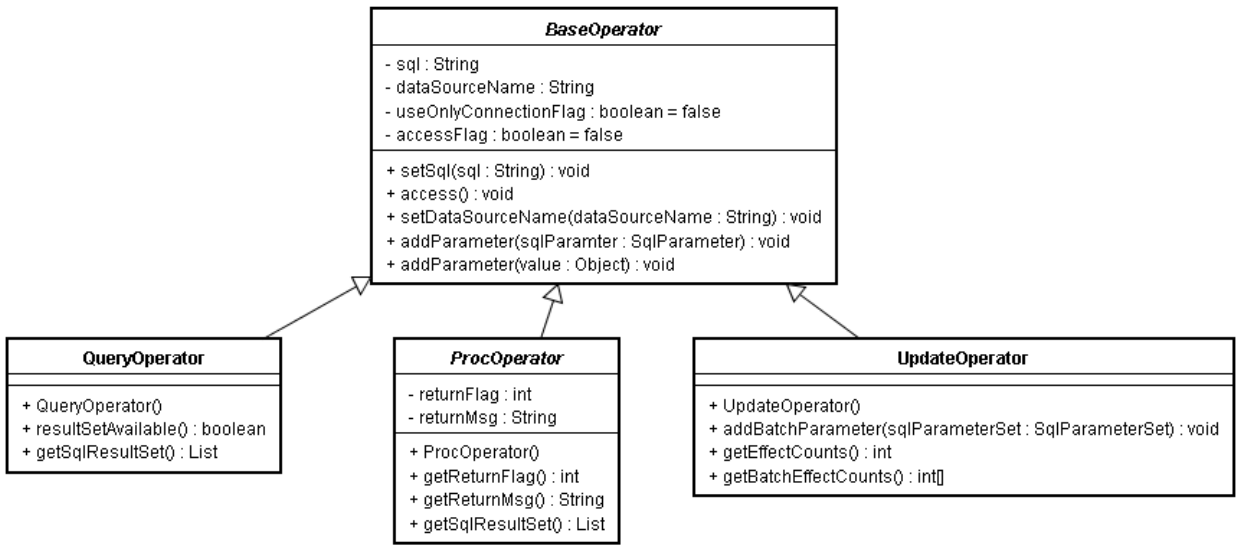


图 2-7 操作器类关系图

上图可知 QueryOperator 功能方法如下表所列：

方法与属性	功能说明

setDataSourceName(dataSourceName: String): void	设置数据源的名称，（此名称与DBConfig.xml文件定义要保持一致）
setSql(sql: String): void	设置查询的sql语句，推荐使用的“?”参数通配符的sql语句
addParameter(sqlParamter: SqlParameter): void	设置带“?”sql语句的参数。根据通配符的顺序来添加sql语句参数。SqlParameter为参数对象
addParameter(value: Object): void	设置带“?”sql语句的参数。根据通配符的顺序来添加sql语句参数。无需指定构造sqlParamter对象，即不指定参数类型，直接填入参数值。依赖具体的jdbc驱动
access():void	执行查询（每个操作器原则上只能执行一次查询）
resultSetAvailable():boolean	检查结果集是否可用（即是否查询存在返回结果集） 返回： true—存在结果集；false—不存在结果集
getSqlResultSet():List	获取结果集列表

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：

```
public static List findEmpBySal(Float sal) {
      List empList = new ArrayList();
      QueryOperator query = new QueryOperator();
      query.setDataSourceName(Const.dataSourceName);// 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));
      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());
            for (int i = 0; i < rs.rowCount; i++) {
```

```
                    Emp emp = new Emp();
                    // 利用 getFieldValueAsXXX 方法取值
                    emp.setComm(rs.getFieldValueAsFloat("comm"));
                    emp.setDeptNo(rs.getFieldValueAsInteger("deptno"));
                    emp.setEmpNo(rs.getFieldValueAsInteger("empno"));
                    emp.setEname(rs.getFieldValueAsString("ename"));
                    emp.setHireDate(rs.getFieldValueAsTimestamp("hiredate"));
                    emp.setJob(rs.getFieldValueAsString("job"));
                    emp.setMgr(rs.getFieldValueAsInteger("mgr"));
                    emp.setSal(rs.getFieldValueAsFloat("sal"));
                    empList.add(emp);
                    rs.next();// 移动记录位置指向下一条
            }
            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;
}
List empList = findEmpBySal(new Float(2000));
…
```

另外，对上面代码黄色加亮部分采取 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，RsDataSet 结果处理对象提供了 autoFillRow()自动填充数据方法，如上面代码黄色结果处理部分可简化为：

```
...
if (query.resultSetAvailable()) {
            RsDataSet rs = new RsDataSet(query.getSqlResultSet());
            for (int i = 0; i < rs.rowCount; i++) {
                    Emp emp = new Emp();
                    rs.autoFillRow(emp);
                    empList.add(emp);
                    rs.next();
            }
```

```
            rs.clearAll();

        }

}

...
```

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。

更新操作器

从图 3-7 可知，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。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(Const.dataSourceName);
        update.setSql("update emp set sal=? where empno=?");
        update.addParameter(new SqlParameter(SqlType.FLOAT, sal));
        update.addParameter(new SqlParameter(SqlType.INTEGER, 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：

```
public static int[] updateSalBatch(Float sals[], Integer empNos[]) {
      UpdateOperator update = new UpdateOperator();
      update.setDataSourceName(Const.dataSourceName);
      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);
```

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。两种操作其 UML 关系图如下：

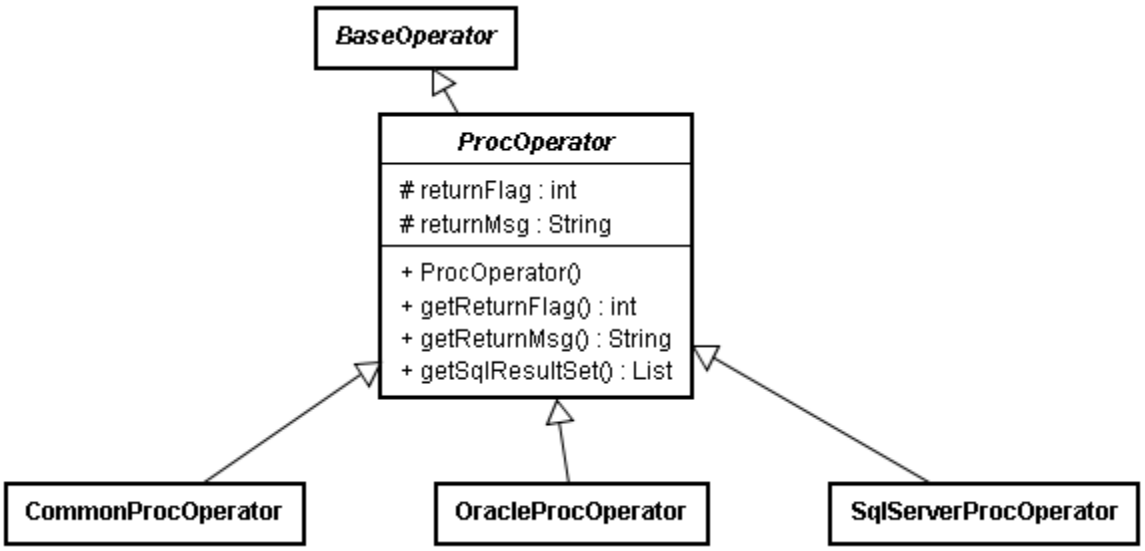


图 2-8 存储过程操作器类图

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：

```
ProcOperator proc = new OracleProcOperator();
proc.setDataSourceName(Const.dataSourceName);// 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());
            for (int i = 0; i < rs.rowCount; i++) {// 打印结果
                  Emp emp = new Emp();
                  rs.autoFillRow(emp);
                  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("--");
                emp = null;
                rs.next();
            }
            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。

唯一标识生成组件

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。

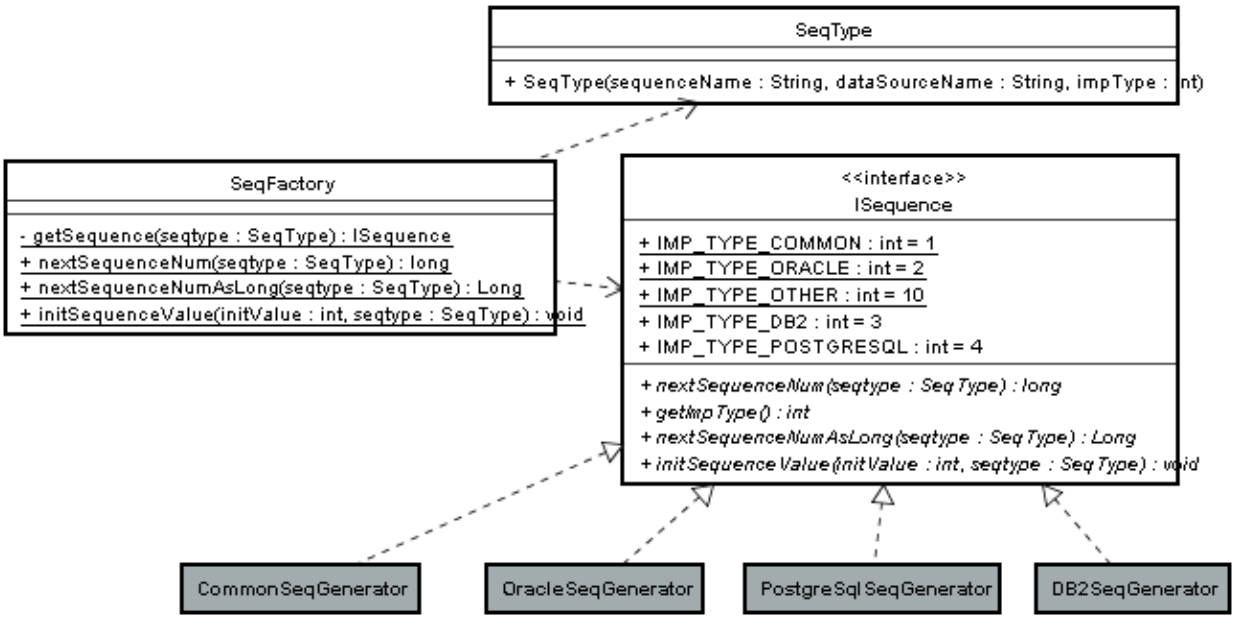


图 2-9 唯一标识生成组件类图

上图可知，ISequence 定义以下属性和方法：

方法与属性	功能说明
IMP_TYPE_COMMON	实现标记，采取NamedSequenceTable 策略的通用Sequence实现，具体对应CommonSeqGenerator实现类
IMP_TYPE_ORACLE	实现标记，对应OracleSeqGenerator专有实现
IMP_TYPE_DB2	实现标记，对应DB2SeqGenrator专有实现
IMP_TYPE_POSTGRESQL	实现标记，对应PostgreSqlSeqGenerator专有实现
nextSequenceNum(seqtype:SeqType): long	从数据库系统获取下一个序列号码 参数： SeqType序列号输入参数对象 返回： 基本long的序列号码
nextSequenceNumAsLong(seqtype: SeqType): Long	同上，只是返回是Long对象
initSequenceValue(initValue: int,	初始化一个Sequence，目前只支持CommonSeqGenerator实现。

seqtype: SeqType): void	

从图 3-9 我们知道，ISequence 接口有 4 个实现。除了 CommonSeqGenerator [1]外，其它 3 个都是分别针对 Oracle、DB2 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），通过封装 ISequence 编程接口的 SeqFactory 工厂类来获取下一位序列号：

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

  SeqType st=new
    SeqType("customerId",Const.dataSourceName,SeqType.IMP_TYPE_COMMON);

  long customerId=SeqFactory.nextSequenceNum(st);

  System.out.println(customerId);

}
```

若是换成其它实现，只需修改 SeqType 参数对象即可，例如换成 OracleSeqGenerator 实现，SeqType 参数为：

```
SeqType st=new
    SeqType("customerId",Const.dataSourceName,SeqType.IMP_TYPE_ORACLE);
```

此时“customerId”为 Oracle 自身建立的 Sequence 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。

数据库分页查询组件

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，组件 UML 结构图如下：

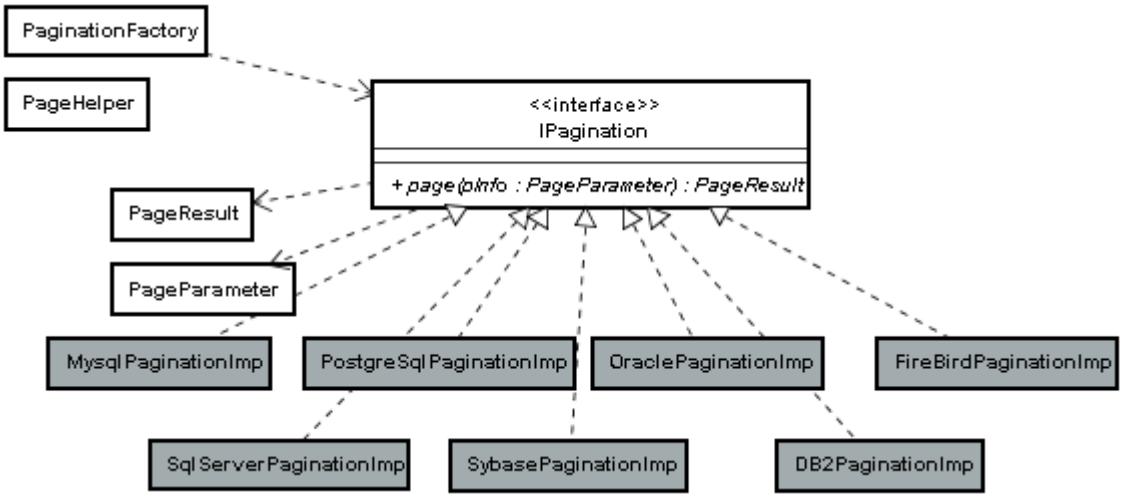


图 2-10 分页查询组件类图

IPagination 接口定义了 ***page(pInfo: PageParameter): PageResult*** 分页方法，PaginationFactory 用来解耦客户端与各个实现类之间的调用关系；PageParameter 是分页参数对象，负责分页请求参数的构建和组织；PageResult 是分页查询结果信息对象；PageHelper 是计算分页的助手类。

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，其实现代码如下：

```
public static void main_testPagination(String[] args) {

      // 建立一个分页参数对象
```

```
        PageParameter param = new PageParameter();
        param.setDataSourceName(Const.dataSourceName); // 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）
        // 利用 PaginationFactory 获取 Oracle 分页实现的实例
        IPagination pageOpt = PaginationFactory
                    .getPaginationInstance(PaginationFactory.ORACLE_ID);
        // 执行分页查询，并返回查询结果信息对象
        PageResult result = pageOpt.page(param);
        // 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
        for (int i = 0; i < rs.rowCount; i++) {
              Emp emp = new Emp();
              rs.autoFillRow(emp);
              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("--");
            emp = null;
            rs.next();
        }
        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。

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。

CompositeQueryOperator 的类图如下：

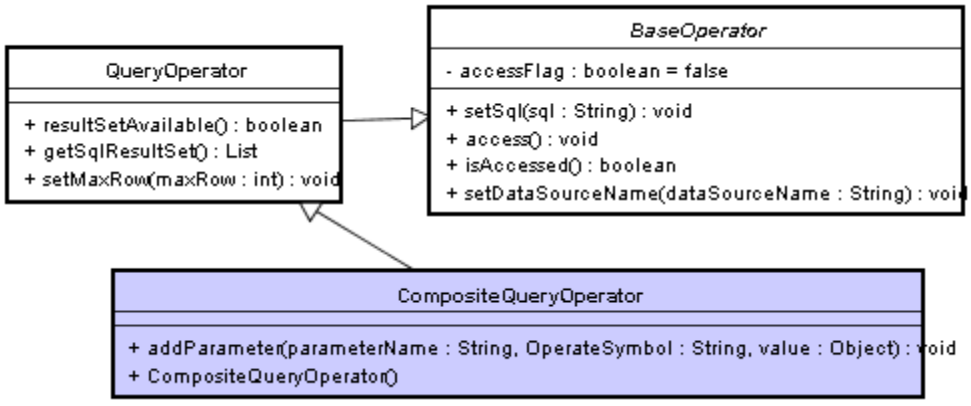


图 2-11 复杂条件组合查询器类图

从图可见，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：

以图 2-6 的 EMP 表为例，从图可知表 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：

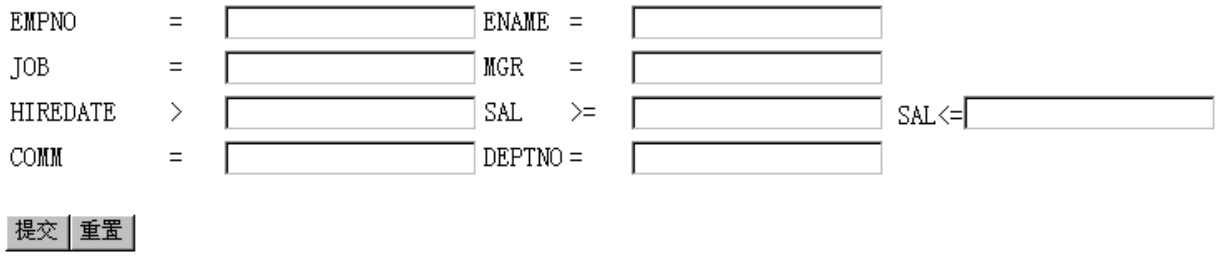


图 2-12 条件组合界面示意图

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：

```
package test.persistence;


import java.math.BigDecimal;
import java.sql.Timestamp;
import com.beetle.framework.persistence.access.operator.RsDataSet;
import com.beetle.framework.persistence.composite.CompositeQueryOperator;


public class TestSQL {


      public static void main(String[] args) {
            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 = null;
            CompositeQueryOperator cqo = new CompositeQueryOperator();
            cqo.setDataSourceName("SYSDATASOURCE_DEFAULT");// 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
            if (cqo.resultSetAvailable()) {// 处理结果，处理结果请参考 QueryOperator，这里只是打印出来
                  RsDataSet rs = new RsDataSet(cqo.getSqlResultSet());
                  for (int i = 0; i < rs.rowCount; i++) {
                        for (int j = 0; j < rs.colCount; j++) {
                              System.out.println(rs.getFieldValue(j));
                        }
                        rs.next();
                        System.out.println("--");
                  }
            }
      }
}
```

可见，一切变得很简单。

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。

DAO 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。

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。

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：

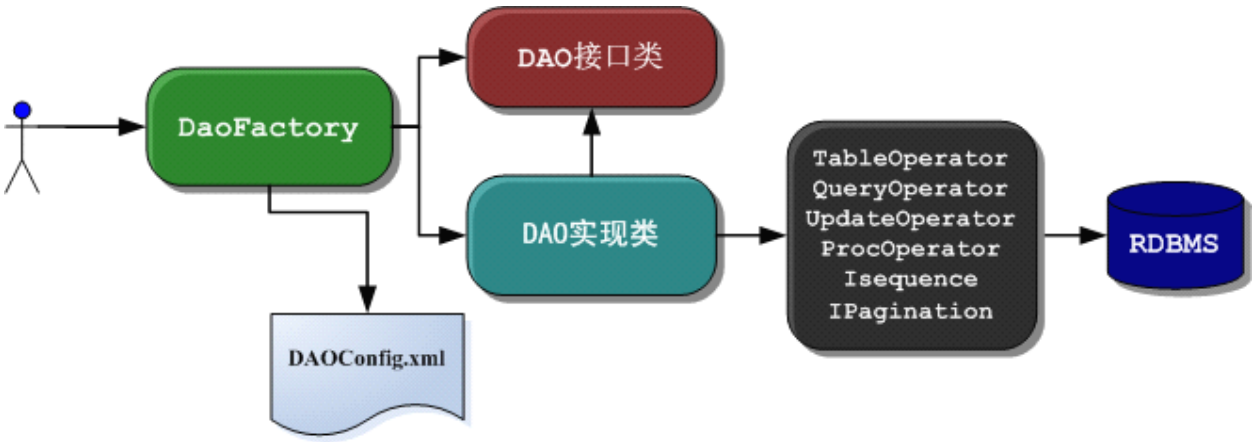


图 2-13DAO 模式工作原理图

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);
}
```

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"?>
<DAO>
  <!-- 参数设置 -->
  <PARAMETERS>
    <!-- 是否在系统初始化时候，缓存数据访问对象 -->
    <item name="initialCache" value="false"/>
  </PARAMETERS>
  <!-- 装配实现类 -->
  <IMPLEMENT>
    <item interfaceName="IDeptDao" impClass="psdemo.dao.imp.PsDept"/>
    <item interfaceName="IEmpDao" impClass="psdemo.dao.imp.PsEmp"/>
  </IMPLEMENT>
</DAO>
```

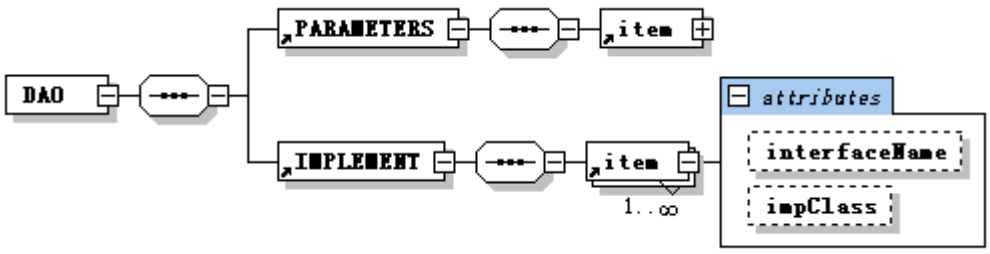


图 2-14 DAOConfig.xml 文件结构

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");
	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();
}
```

采取 IEmpDao empDao = new PsEmp()岂不是更直接？通过 DaoFactory 来装配的除了解藕接口与实现外，另外的带来的好处是工厂可以管理这些实现，将其单例化，达到内存及性能的优化目的。

本地存储器

持久层除了常见的关系数据库操作外，很多时候，我们需要把数据以 java 值对象形式保存在本地磁盘。如何存储、检索和管理这些数据呢？BJAF 框架提供了一个本地存储器的文件系统来解决这些问题。本地存储器涉及类关系图如下：

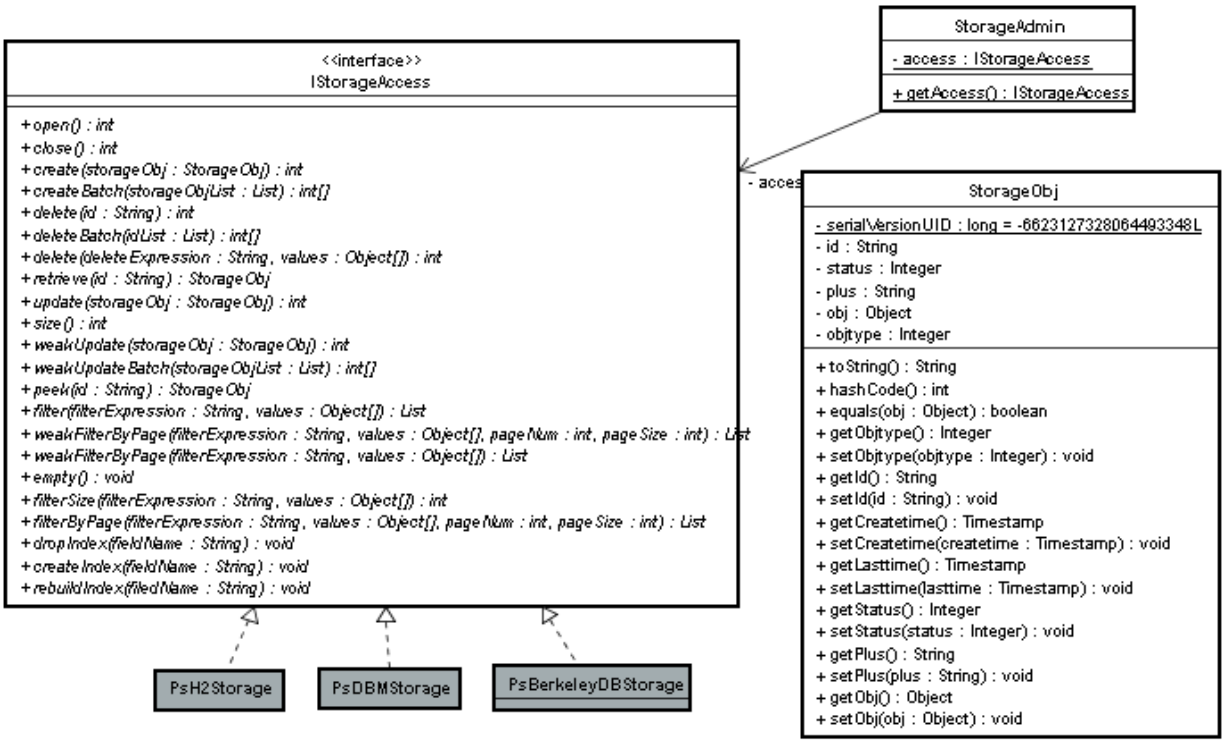


图 2-15 本地存储器类关系图

其具体实现思路是，构建一个 StorageObj 的存储对象，把需要存储数据放入此对象中；然后调用 StorageAdmin 存储管理者，返回这个操作本地磁盘的 StorageAccess 实例来保存到本地；当然，对于存储在本地的 StorageObj 的其它操作，如：检索、删除、更新等等，也是通过 StorageAccess 存储访问接口来实现。从上图可以，我们提供了 H2、DBM 和 BerkeleyDB 三种实现。

StorageObj 对象包括以下属性：

属性	说明
`String` **`id`**	对象唯一标识（要保证在整个存储文件系统中唯一）
`Timestamp` **`createtime`**	对象创建时间
`Timestamp` **`lasttime`**	对象最后修改时间
`Integer` **`status`**	对象操作状态位（数值可根据需要自行定义）
`String` **`plus`**	对象附加信息
`Object` **`obj`**	要存储的具体数据对象
`Integer` **`objtype`**	对象类型

从图 2-15 可见 IStorageAccess 提供了一系列操作方法，其详细说明参见以下接口定义：

```java
package com.beetle.framework.persistence.storage;

import java.util.List;
/**
 * 存储器操作接口
 *
 *
 * @author YUHAODONG yuhaodong@gmail.com
 *
 */
public interface IStorageAccess {
	/**
	 * 打开存储器
	 *
	 * @return 0--成功，-1 失败
	 */
	int open() throws StorageAccessException;
	/**
	 * 关闭存储器
	 *
	 * @return 0--成功，-1 失败
	 */
	int close() throws StorageAccessException;
	/**
	 * 把一个 storageobj 对象存储在文件中
	 *
	 * @param storageObj
	 * @return 1 为成功
	 */
	int create(StorageObj storageObj) throws StorageAccessException;
	/**
```

```
     * 批量建立
     *
     * @param storageObjList
     *              存储对象列表,执行完毕后此列表不会清空
     * @return
     */
    int[] createBatch(List storageObjList) throws StorageAccessException;
    /**
     * 从文件中，根据 id 把 storage 对象删除
     *
     * @param id
     * @return 1 为成功；0 为不存在此对象
     */
    int delete(String id) throws StorageAccessException;
    /**
     * 批量删除
     *
     * @param idList
     *             id 列表
     * @return
     */
    int[] deleteBatch(List idList) throws StorageAccessException;
    /**
     * 根据删除表达式删除文件系统数据
     *
     *
     * @param deleteExpression
     * @param values
     * @return 成功删除的记录数
     */
```

```
    int delete(String deleteExpression, Object values[])
                    throws StorageAccessException;
    /**
     * 根据 id，从文件中找回此对象
     *
     * @param id
     * @return
     */
    StorageObj retrieve(String id) throws StorageAccessException;
    /**
     * 从文件中更新此对象
     *
     *
     * @param storageObj
     * @return 1 为成功，0 为不存在此对象
     */
    int update(StorageObj storageObj) throws StorageAccessException;
    /**
     * 返回当前文件中多少个对象
     *
     * @return
     */
    int size() throws StorageAccessException;
    /**
     * 弱更新，只根系 StorageObj 对象中除了 obj 属性的属性
     *
     *
     * @param storageObj
     * @return
     */
```

```
        int weakUpdate(StorageObj storageObj) throws StorageAccessException;
        /**
         * 批量 弱更新，只根系 StorageObj 对象中除了 obj 属性的属性
         *
         *
         * @param storageObjList
         * @return
         */
        int[] weakUpdateBatch(List storageObjList) throws
StorageAccessException;
        /**
         * 根据 id，找出此对象，但不包括 obj 属性数据
         *
         *
         * @param id
         * @return
         */
        StorageObj peek(String id) throws StorageAccessException;
        /**
         * 根据表达式，过滤文件中的对象记录集
         *
         *
         * @param filterExpression
         *              --表达式与参见的 SQL 条件语言语法相一致 [WHERE
         *              expression[id,createtime,lasttime,status]] [ORDER BY
order
         *              [,...]] [LIMIT expression [OFFSET expression]
[SAMPLE_SIZE
         *              rowCountInt]]
         * @param values
         * @return 返回 StorageObj 对象列表
```

```
     */
    List filter(String filterExpression, Object values[])
                throws StorageAccessException;
    /**
     * 分页过滤(obj 属性不会返回)
     *
     * @param filterExpression
     * @param values
     * @param pageNum
     *             --页号
     * @param pageSize
     *             --页大小
     *
     * @return
     */
    List weakFilterByPage(String filterExpression, Object values[],
                int pageNum, int pageSize) throws StorageAccessException;

    List weakFilterByPage(String filterExpression, Object values[])
                throws StorageAccessException;
    /**
     * 清空存储文件内容
     */
    void empty() throws StorageAccessException;
    /**
     * 根据表达式，过滤文件中的对象记录集的大小
     *
     * @param filterExpression
     *             --表达式与参见的 SQL 条件语言语法相一致 [WHERE
     *             expression[id,createtime,lasttime,status]] [ORDER BY
```

```
order
         *             [,...]] [LIMIT expression [OFFSET expression]
[SAMPLE_SIZE
         *             rowCountInt]]
         * @param values
         * @return 对象记录集的大小
         */
        int filterSize(String filterExpression, Object values[])
                        throws StorageAccessException;
        /**
         * 分页过滤
         *
         * @param filterExpression
         * @param values
         * @param pageNum
         *             --页号
         * @param pageSize
         *             --页大小
         *
         * @return
         */
        List filterByPage(String filterExpression, Object values[], int pageNum,
                        int pageSize) throws StorageAccessException;
        /**
         * 删除索引
         *
         * @param fieldName
         *             -[createtime,lasttime,status,objtype]
         */
        void dropIndex(String fieldName) throws StorageAccessException;
        /**
```

```
        * 建议索引，优化查询
        *
        *
        * @param fieldName
        *             -[createtime,lasttime,status,objtype]
        */
       void createIndex(String fieldName) throws StorageAccessException;
       /**
        * 重建索引
        *
        * @param filedName
        *             -[createtime,lasttime,status,objtype]
        */
       void rebuildIndex(String filedName) throws StorageAccessException;
}
```

例如，把一个 User 的 java 对象存储在本地文件，并对它进行查询、更新操作，示例代码如下：

1， 建立一个 User 值对象：

```
package example.persistence;


import java.io.Serializable;


public class User implements Serializable {
       private static final long serialVersionUID = 1L;
       private String name;
       private String pwd;
       private String address;
       private int age;
       private String phone;
       public String getName() {
              return name;
```

```
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getPwd() {
		return pwd;
	}
	public void setPwd(String pwd) {
		this.pwd = pwd;
	}
	public String getAddress() {
		return address;
	}
	public void setAddress(String address) {
		this.address = address;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public String getPhone() {
		return phone;
	}
	public void setPhone(String phone) {
		this.phone = phone;
	}
}
```

2， 对此对象赋值，并存储到本地磁盘

```
package example.persistence;

import com.beetle.framework.persistence.storage.IStorageAccess;

import com.beetle.framework.persistence.storage.StorageAdmin;

import com.beetle.framework.persistence.storage.StorageObj;

public class TestClient {

	public static void main(String[] args) {

		// 构建一个 user 值对象并赋值

		User user = new User();

		user.setName("Henry");

		user.setAge(32);

		user.setAddress("中国深圳");

		user.setPhone("13501583576");

		user.setPwd("888888");

		// 构建一个存储对象，以便保存

		StorageObj sobj = new StorageObj();

		sobj.setId("100001");// 指定一个 id，保证文件系统能够唯一标示此存储对象

		sobj.setCreatetime(new
java.sql.Timestamp(System.currentTimeMillis()));// 指定创建时间

		sobj.setObj(user);// 把 user 值对象加入存储对象中

		sobj.setObjtype(new Integer(1));// 设置对象类型，类型值含义自己定义

		sobj.setStatus(new Integer(0));// 设置存储对象在文件系统状态，状态值含义
自己定义

		sobj.setPlus("");

		// 获取一个 StorageAdmin 实例，并从实例获取磁盘操作实例

		IStorageAccess access = StorageAdmin.getAccess();

		access.create(sobj);// 建立一个存储对象（把存储对象保存在磁盘中）

	}

}
```

此时，框架会自动在当前目录下初始化一个文件系统，并把存储对象保存在文件系统的文件中，见下示意图：

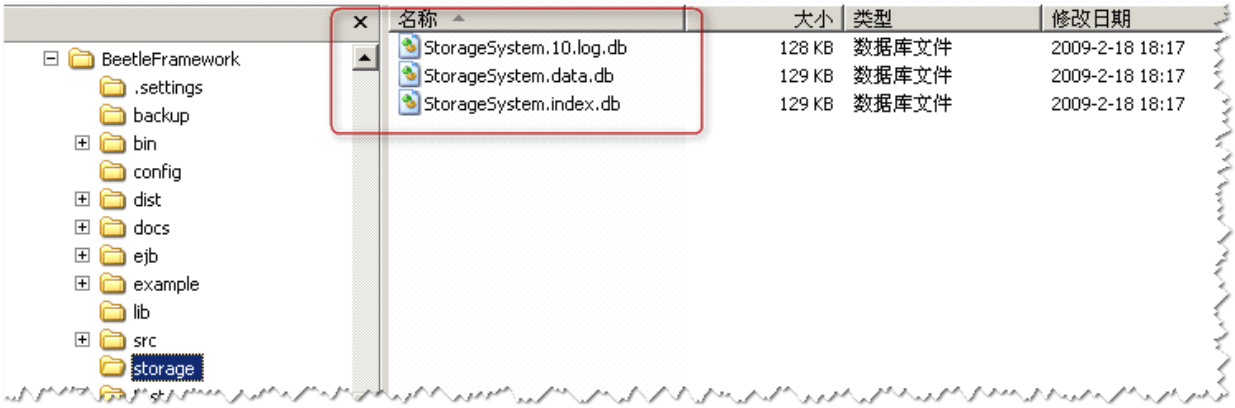


图 2-16 本地存储数据文件示意图

3， 查找并更新这个存储对象

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

	IStorageAccess access = StorageAdmin.getAccess();

	StorageObj sobj=access.retrieve("100001");

	System.out.println(sobj.getCreatetime());

	User user=(User)sobj.getObj();

	System.out.println(user.getName());

	System.out.println(user.getAge());

	System.out.println(user.getAddress());

	System.out.println(user.getPhone());

	System.out.println(user.getPwd());

	user.setAddress("中国广东深圳罗湖");

	sobj.setObj(user);

	access.update(sobj);

}
```

执行结果如下：

```
Henry

32

中国深圳

13501583576

888888
```

本地存储器对存储内容提供了功能强大的检索功能，并检索性能优秀；存储文件的大小最大的限制是 256GB，特别地，对于 FAT/FAT32 的文件系统，其大小限制是 4GB。

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，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，只提供了浏览器端验证框架，具体实现为 validatorEx.js 库。

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：

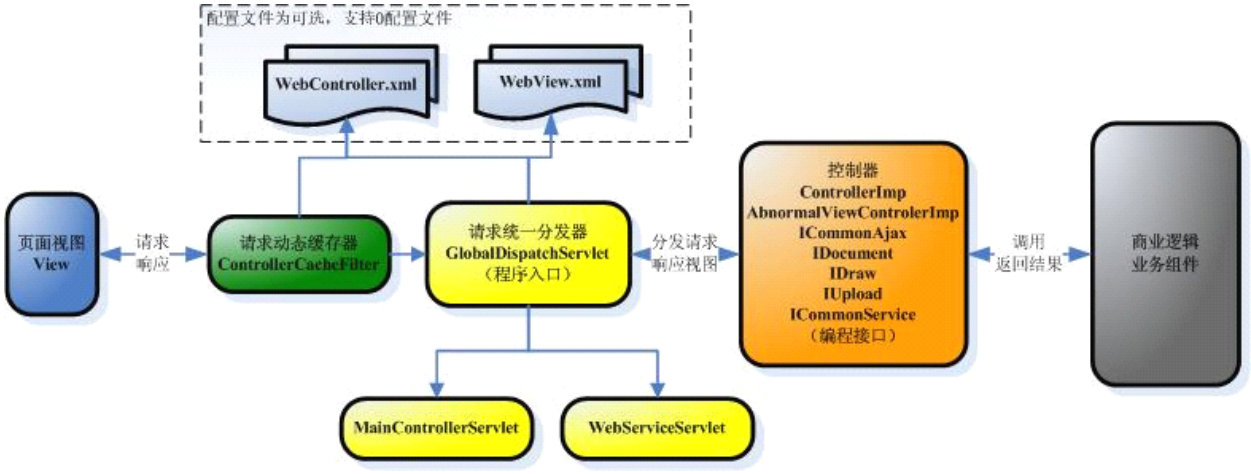


图 3 1 BJAF Web 框架对 HTTP 请求的处理基本流程

Web 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> [12]
		<url-pattern>*.ajax</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。

控制器

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：

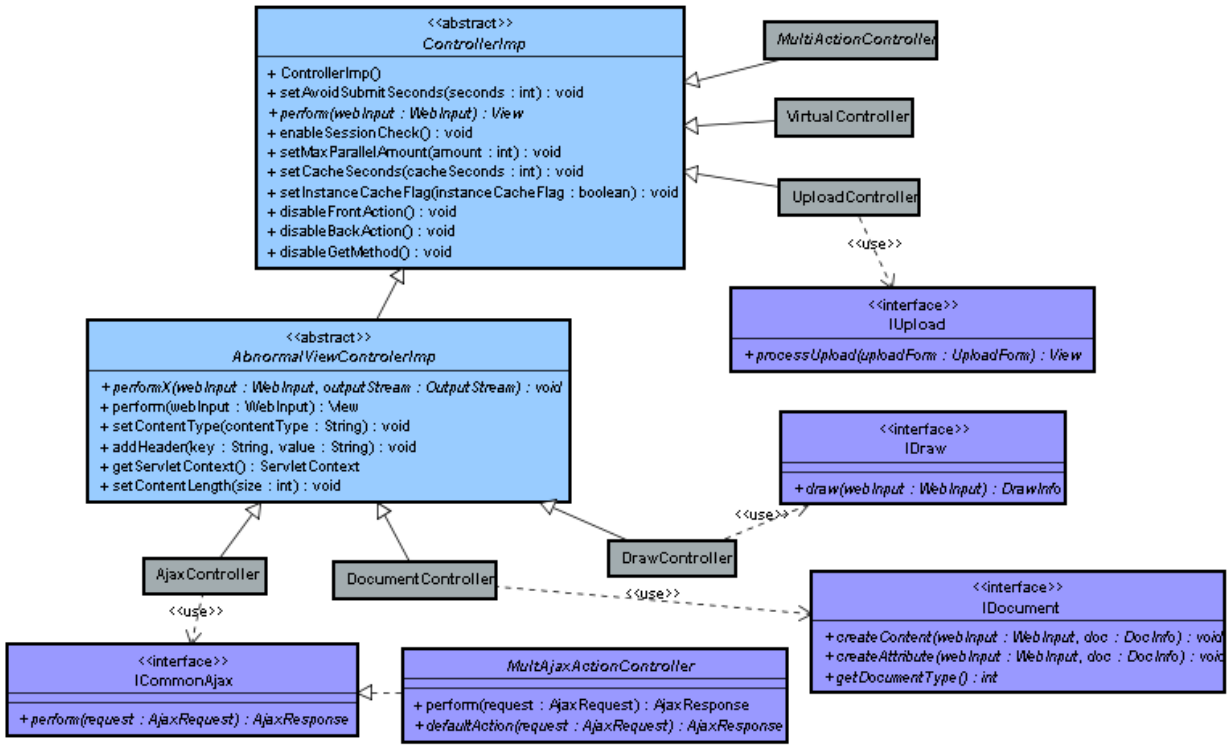


图 3-2 控制器组件及编程接口关系类图

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。

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。

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>
        <!--Ajax 特例控制器横切动作<ajaxFrontAction>为前置点横切；<ajaxBackAction>为后置点横切 -->
      <ajaxFrontAction>Example.GlobalPreActionImp</ajaxFrontAction>
      <ajaxBackAction>Example.GlobalPreActionImp</ajaxBackAction>
    </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>
    <ajax>
      <aItem name="PieStatistics.Draw"
class="com.beetle.jIKBS.web.StatisticsDraw"/>
        <!-- 添加更多的 item... -->
    </ajax>
    <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：

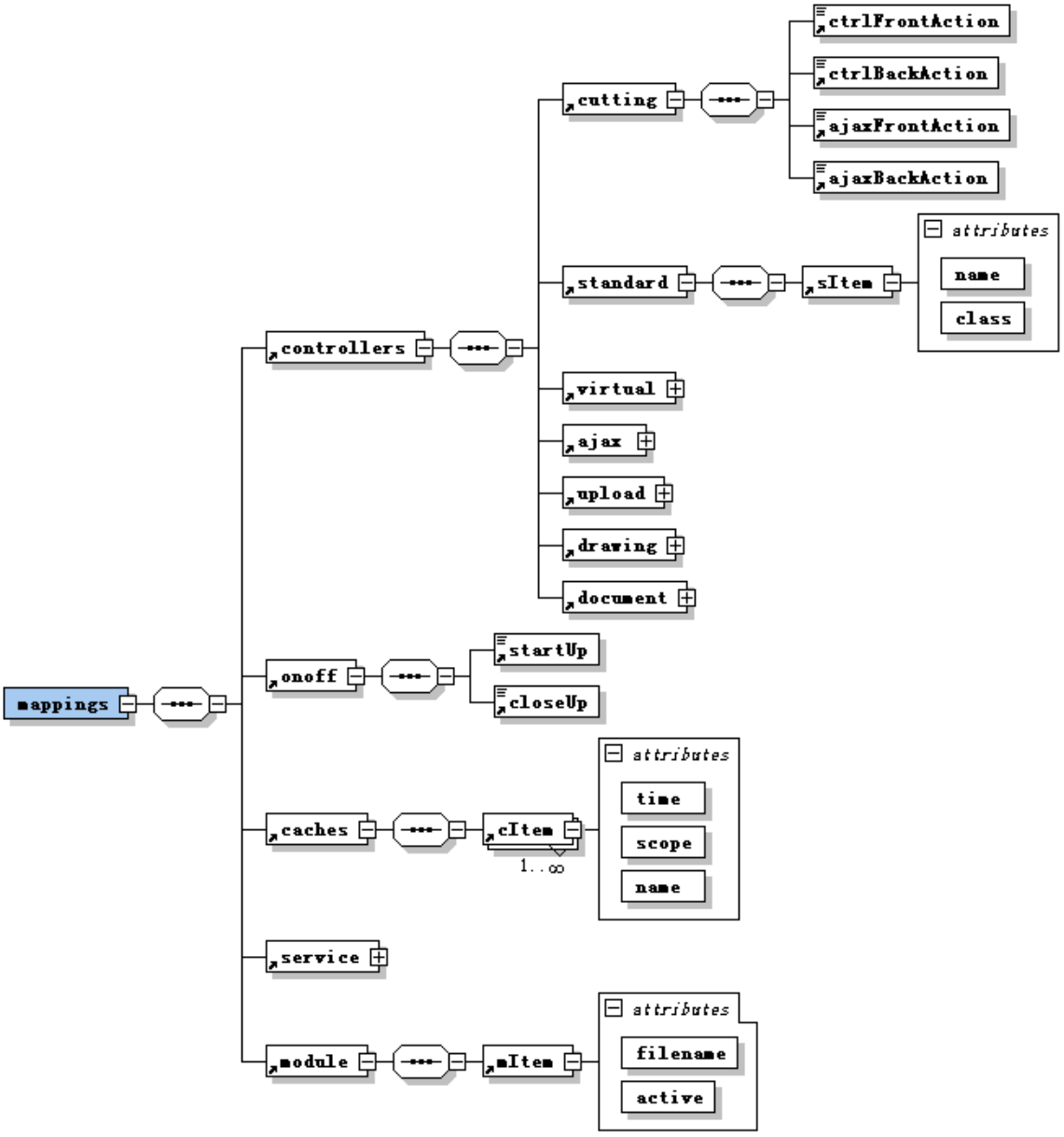


图 3-3WebController.xml 配置文件结构图

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。

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。

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：

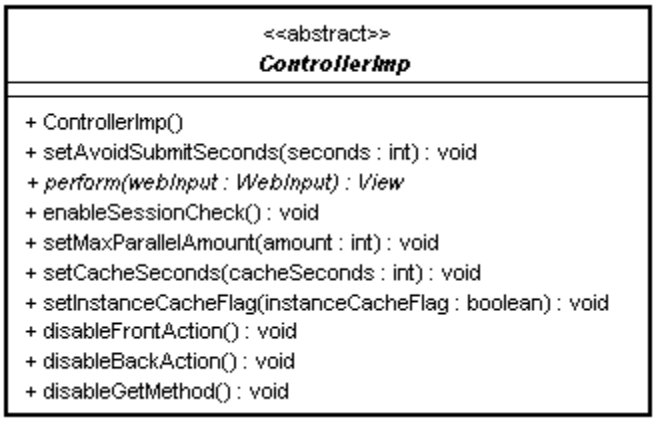


图 3-4ControllerImp 类图

从图 4-4 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。	

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，从图 4-2 是可知，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) {
                  }
            }
      }
}
```

从图 3-2 可知，DoucumentController、AjaxController、DrawController 都是继承自这个非标准视图控制器的特例实现。

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。

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：

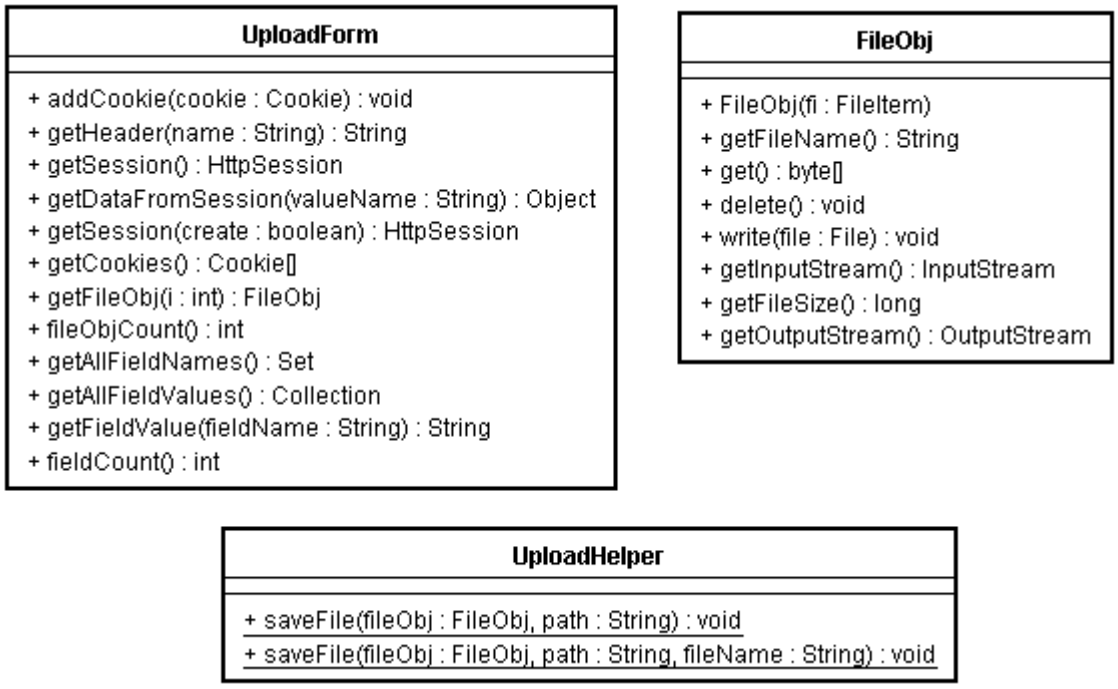


图 3-5 UploadForm 等组件类图

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.SysLogger;
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 SysLogger logger =
SysLogger.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>
```

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。

Ajax 控制器

Ajax 是当前流行的 Web 开发技术，BJAF Web 框架在原有框架结构上对 Ajax 进行了强而有力的封装，是开发 Ajax 就想开发一个普通的传统标准控制器那么简单。Ajax 框架结构如下：

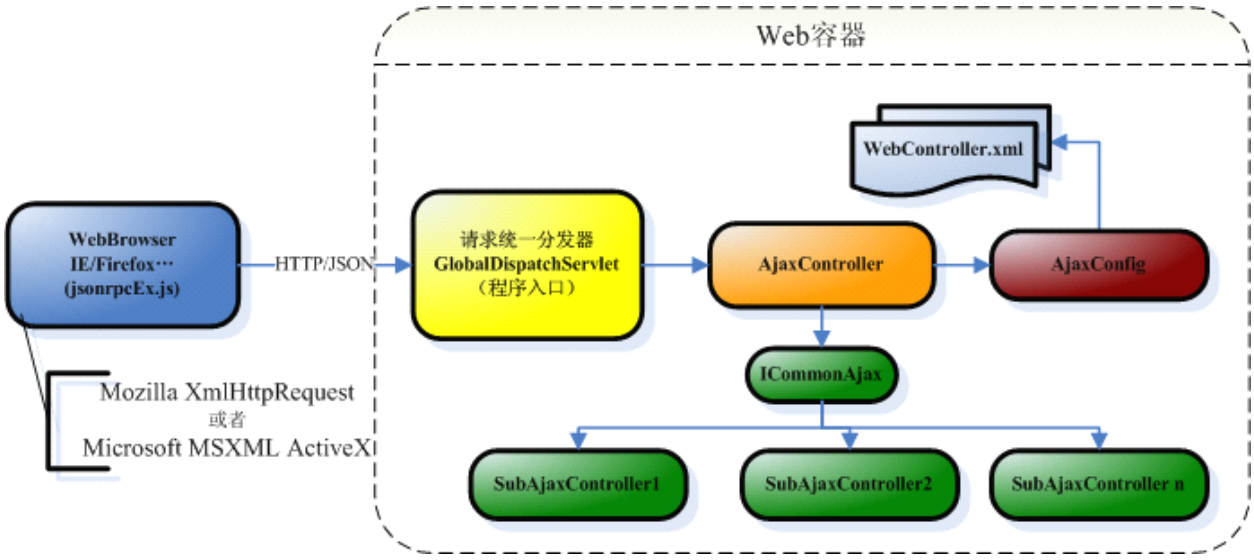


图 3-6BJAF Web Ajax 框架结构图

浏览器采取 Javascript 调用 XMLHttpRequest（Mozilla、Microsoft 各有实现）后台底层 Http 请求组件与远程的 J2EE Web Servlet 容器通信：发出的请求交给 GlobalDispatchServlet 服务程序去负责统一处理，它适配给 AjaxController 控制器再通过 AjaxConfig 配置类读取部署在 WebController.xml 文件中的 Ajax 子控制器配置信息。然后，根据浏览器提交请求的子控制器名称来匹配其对应的处理程序，完成请求逻辑并把结果数据返回给浏览器客户端。最后，浏览器再利用 Javascript 来解析并展现这些结果数据。

ICommonAjax 是请求处理程序的入口，所有的 Ajax 子控制器都必须实现此接口，其定义如下：

```
package com.beetle.framework.web.controller.ajax;

public interface ICommonAjax {

  /**

   * ajax 控制逻辑执行方法，GlobalDispatchServlet 会根据请求的名称来找到此接口的实现类，

   * 并执行此方法完成任务

   * @param AjaxRequest 浏览器客户端提交的参数对象，参照传统的 Http Request 设计。提供一些方法便于获取参数。

   * @return AjaxResponse 返回一个 AjaxResponse 结果响应的数据对象，其数据格式为 json。因为 ajax 是后台刷新，也就是说返回的视图就是它自己本身。

   * @throws ControllerException

   */

  AjaxResponse perform(AjaxRequest request) throws ControllerException;

}
```

可见，其执行过程很明显：在 perform 方法体内，首先从输入参数 AjaxRequest 对象中获取页面提

交上来的数据，然后编程处理请求的逻辑，最后生成结果的数据放在 AjaxResponse 对象中，并返回。

下面例子为用 Ajax 改写标准控制器的登录实现：

```
package com.beetle.WebDemo.presentation;
import java.util.*;
import com.beetle.WebDemo.common.*;
import com.beetle.framework.web.controller.ajax.* ;
public class LoginAjaxController implements ICommonAjax {
  public LoginAjaxController() {
  }
  public AjaxResponse perform(AjaxRequest ajaxRequest) throws
ControllerException {
    AjaxResponse response = new AjaxResponse();
   //获取页面输入的参数
    String userName = ajaxRequest.getParameter("username");
    int password = ajaxRequest.getParameterAsInt("password");
    //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) {
      User user = new User();
      user.setUserId(new Integer(10001));
      user.setName("余浩东");
      user.setPhone("13501583576");
      user.setSex("男");
      user.setYear(30);
      response.setValue("UserInfo", user);
      response.setValue("LoginTime", new Date(System.currentTimeMillis()));
      response.setReturnFlag(0);
      response.setReturnMsg("登陆成功！");
    }
    else {
      response.setReturnFlag(-1);
```

```
      response.setReturnMsg("登陆失败！");
    }
    return response;
  }
}
```

Javascript 客户端请求（同步）代码为：

```
<SCRIPT LANGUAGE="JavaScript">
<!--
function doLogin()
{
      var req=new Request();//创建一个请求对象
      req.setControllerName("LoginAjaxController.ajax");//设置处理控制器名称
      req.put("username",form1.username.value);//输入请求参数
      req.put("password",form1.password.value);
      var r=req.synchroExecute();//同步执行请求并处理响应结果
      if(r.returnFlag==0){
        var userInfo=r.getValueByName("UserInfo");
        var user=userInfo.name;
        var userid=userInfo.userId;
        var useryear=userInfo.year;
        var sex=userInfo.sex;
        var phone=userInfo.phone;
        var loginTime=new Date(r.getValueByName("LoginTime").time);
      alert(r.returnMsg+'\n-->用户信息<--\n 编号：'+userid+'\n 姓名：'+user+'\n 性
别：'+sex+'\n 年龄：'+useryear+'\n 登陆时间：'+loginTime.toString());
      }else{
        alert(r.returnMsg);
      }
}
//-->
```

```
</SCRIPT>
```

当然，框架也支持异步请求，其代码如下：

```
function login(){
    var req=new Request();
      req.setControllerName("LoginAjaxController.ajax");
      req.put("username",f2.username.value);
      req.put("pwd",f2.pwd.value);
      req.put("flag",0);
      req.asynchroExecute( //异步执行，并利用回调函数处理结果
       function resultHandle(r){
            if(r.returnFlag==0){
              alert(r.returnMsg);
              var userInfo=r.getValueByName("UserInfo");
              var username=userInfo.username;
              var pwd=userInfo.pwd;
              var sex=userInfo.sex;
              var email=userInfo.email;
              var registTime=new Date(userInfo.registTime.time);
              alert(r.returnMsg+'\n-->用户信息<--\n 姓名：'+username+'\n 性别：
'+sex+'\n 邮件：'+email+'\n 注册时间：'+registTime.toString());
            }else{
              alert(r.returnMsg);
            }
      });
}
```

最后，需要在 WebController.xml 文件指定 Ajax 控制器与实现类的映射关系：

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

```
        <aItem name="LoginAjaxController.ajax" class="
              com.beetle.WebDemo.presentation.LoginAjaxController"/>
    </ajax>
  </controllers>
</mappings>
```

上面整个 ajax 请求访问过程的各组件的调用关系如下：

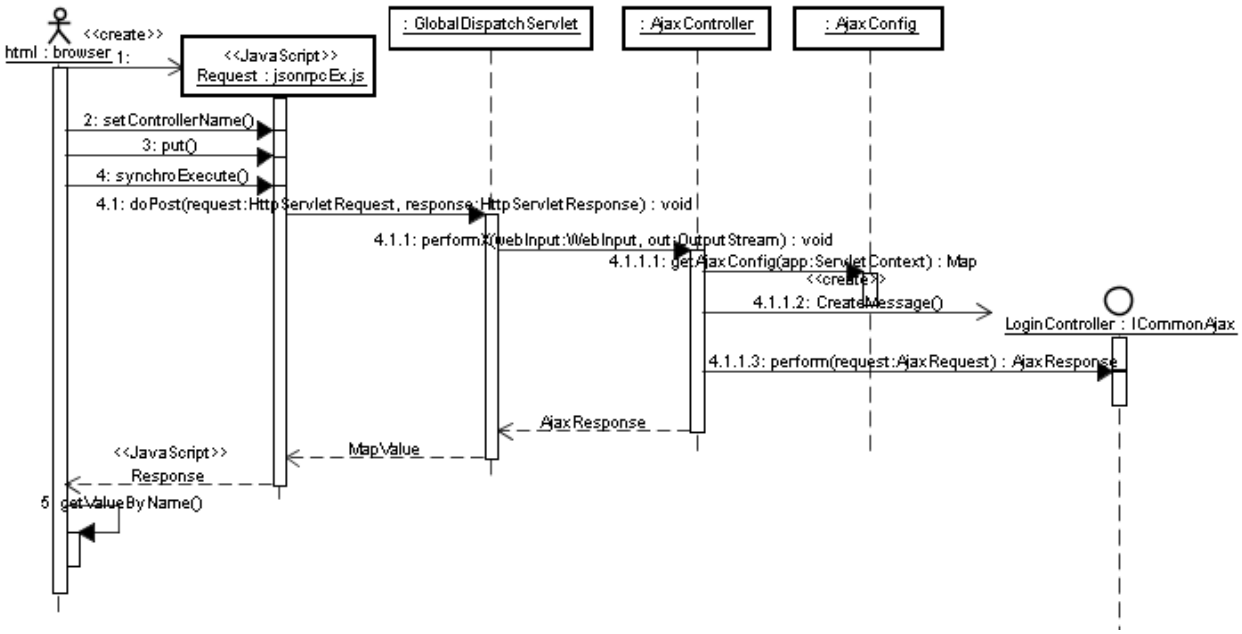


图 3-7 UML 顺序图

首先，浏览器 html 页面引入 jsonrpcEx.js 库，在登录按钮事件内创建一个 Request 请求对象，调用其 setControllerName()方法指定服务器处理此请求的子控制器名称，调用 put()方法输入请求参数，调用 synchroExecute()方法提交请求。

其次，GlobalDispatchServlet 主控器接收到通过 http 协议提交上来的请求，交给 AjaxController 控制器去处理，它根据控制器名称找到其对应的 LoginController 实现类，并实例化它，调用其 perform()方法执行处理请求的逻辑，并返回 AjaxResponse 结果对象给主控器，主控制将此结果进行序列化，以 MapValue 数据形式返回到浏览器的 javascript 客户端。

最后，javacirpt 端 request 对象把传回的数据封装成一个 Response 对象交回给 html 页面，再通过 Response 对象提供的 getValueByName 方法读取返回的数据解析显示在页面视图上。

BJAF Web 框架对 Ajax 技术进行有力封装，详细请参考 Henry Yu 编写的《J2EE Web 开发使用 Ajax 技术的核心所在》文章。

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：

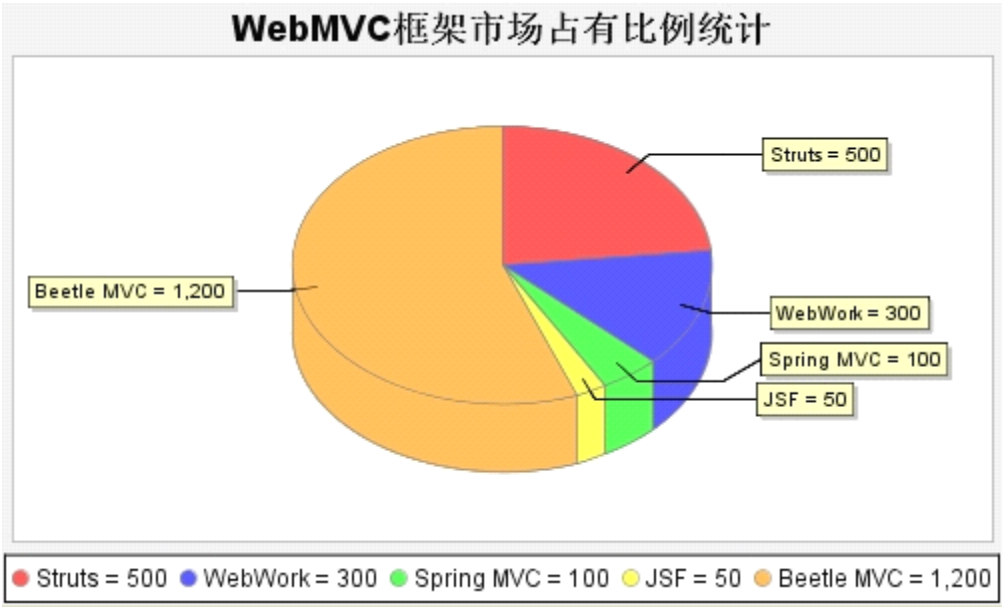


图 3-8 Web 框架统计饼图

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）。

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。从图 4-2 可知，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。

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。

视图显示

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。）

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：

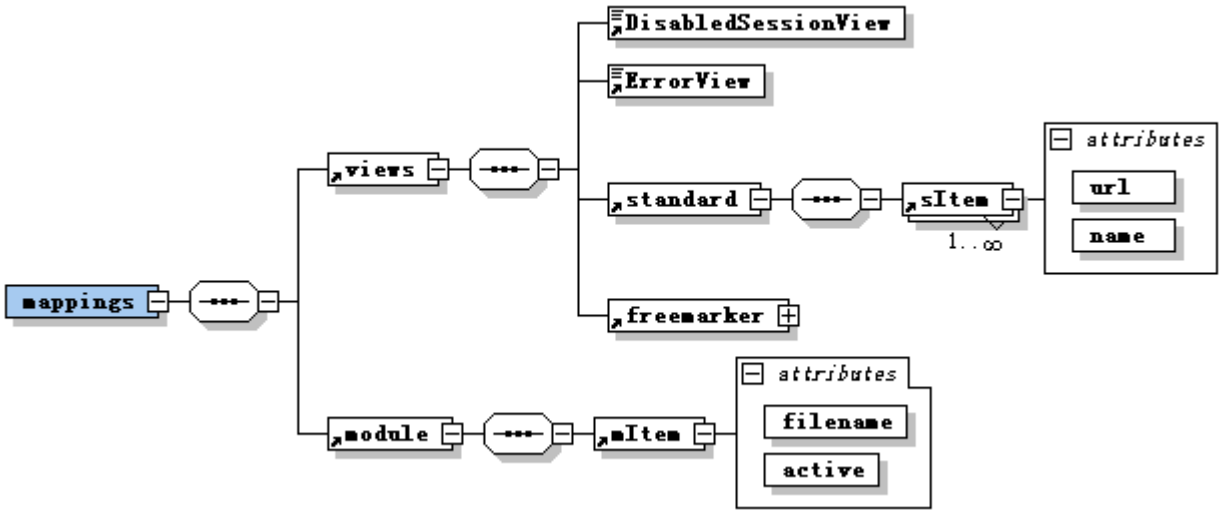


图 3-9 WebView.xml 文件结构

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。

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。

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：

图 3-10 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>
```

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。

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：

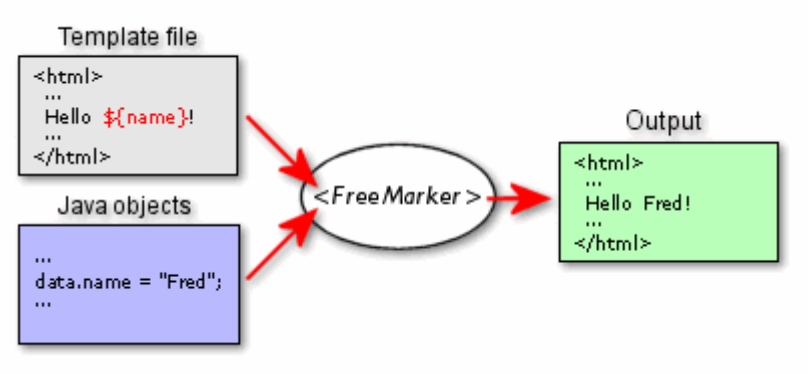


图 3-11 freemarker 工作机制

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。

页面布局

在开发 Web 应用时候，经常会用到页面布局技术，下图就是一种典型的页面布局：

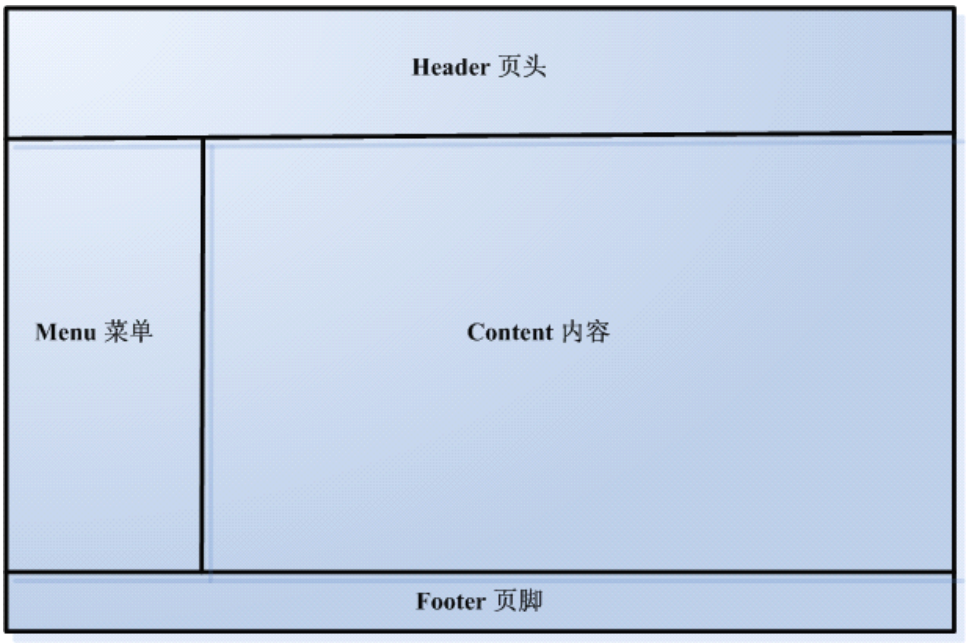


图 3-12 典型的页面布局

在上图中，页面被划分为四个部分：header、menu、footer 和 content。对于同一应用中所有 Web 页面，header、menu 和 footer 三部分的内容相同，仅 content 部分的内容不相同而已。

传统的页面布局一般采取 incluede（包含）公共基本页面方式进行，但是这种方式不够灵活，维护的工作量较大。

BJAF 框架没有对页面布局技术进行封装，理由有二：首先，采取 freemarker 模板技术实现页面布局是很容易的；其次，已存在成熟页面布局方案—SiteMesh，它采取 Decorator 设计模式实现，最大的特点是对其它框架技术没有入侵性，完全实现套接插件式的页面布局。它具备以下特点：

可重用性，基本页面可以被重用，可组合成各种不同的复合式页面。

可扩展性，可以方便的扩展基本页面，从而创建更丰富的页面。

可维护性，每个基本页面之间相互独立，当复合页面中的某局部区域发生变化，不会影响其它区域 。

若需要页面布局，BJAF Web 框架推荐使用 SiteMesh 组件，关于其使用详细资料请参考其官方网站：http://www.opensymphony.com/sitemesh。

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。

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。

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="">
    </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。

Web Service 开发

BJAF Web 框架对基于 Http 协议的 Web Service 开发提供了轻量级的支持，为了传输的高效性，我们没有封装 SOAP 标准工业协议，而且采取 Http-Hessian 协议作为实现，其架构示意图如下：

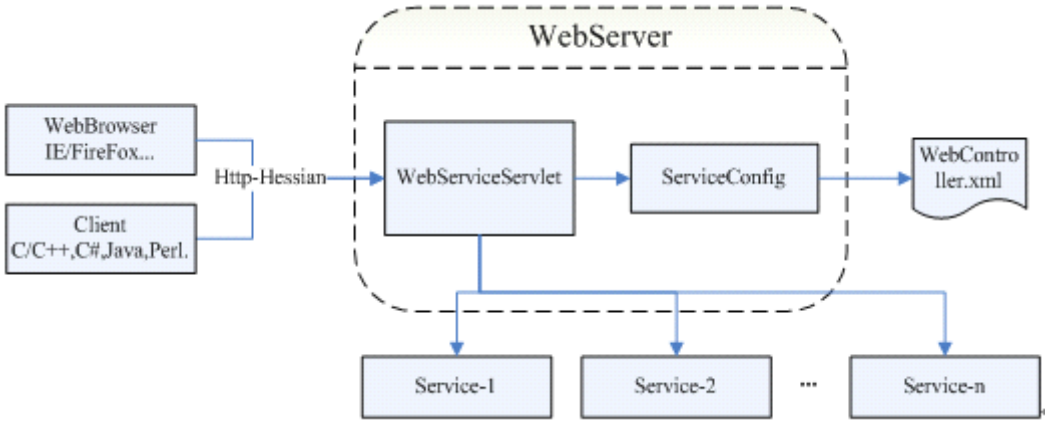


图 3-13 Web Service 架构示意图

客户端利用 Http-Hessian 协议和 WebServer 通信，发出的请求交给 WebServiceServlet 去负责统一处理。它通过 ServiceConfig 配置类读取部署在 WebController.xml 文件中所有服务信息，然后，根据客户端请求服务的名称来匹配其对应的服务。再利用 caucho 公司的 Hessian 协议实现包来传送此服务的根对象（服务接口）到客户端，然后客户端利用此对象调用业务方法工作，实现远程调用，返回计算结果给客户端。

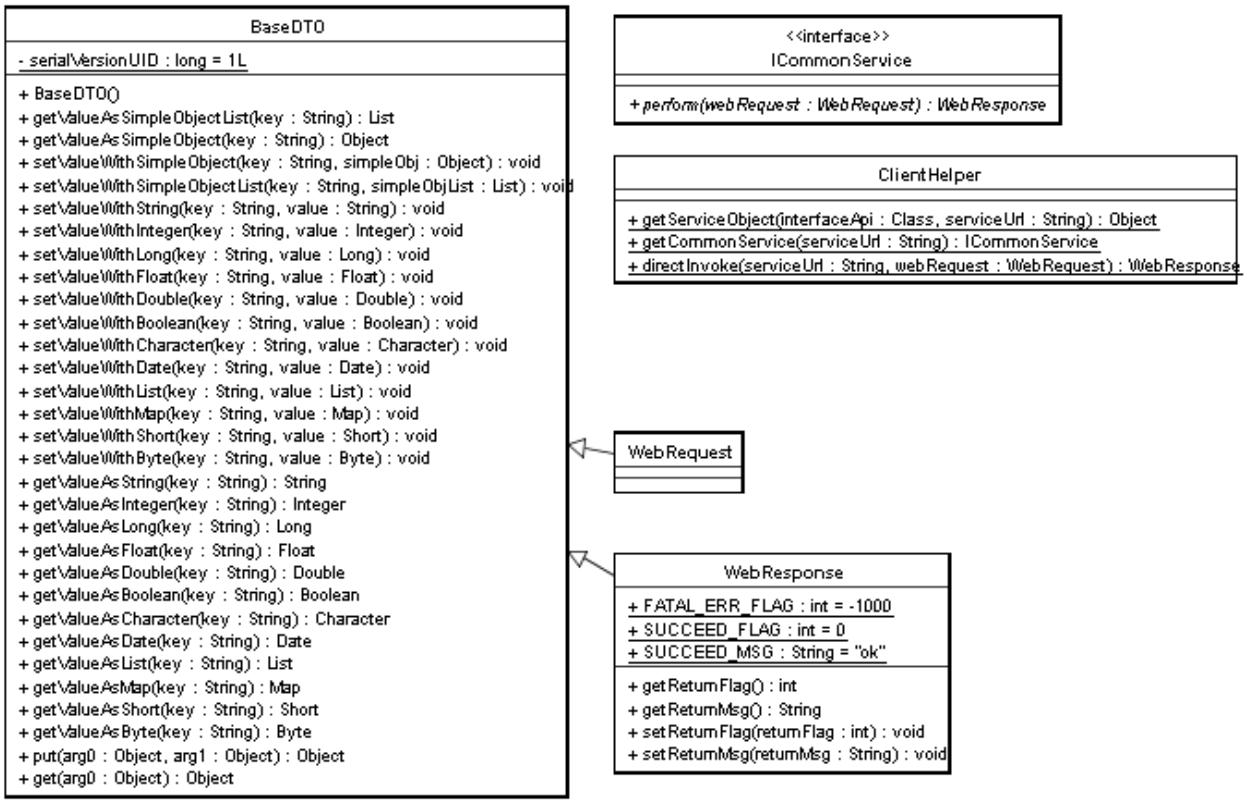


图 4-14 WebService 编程组件类图

上图可知，Web Service 的编程接口是 ICommonService，perform(Req:WebRequest):WebResponse 方法体内完成服务逻辑。WebRequest 是输入参数，WebResponse 为结果数据对象，它们都是扩展自 BaseDTO 值对象；客户端通过结果数据对象 WebResponse 的属性 returnFlag 和 returnMsg 标识服务端的运行状况。另外，对于框架还提供一个 ClientHelper 的 Java 客户端以便开发人员方便调用服务。其它语言的客户端实现请参考 Http-Hessian 协议的实现官方网站。

下面改写前面简单登录验证例子说明一下 Web Service 开发：

（1），实现 ICommonService 接口，完成登录逻辑：

```
package com.beetle.WebDemo.presentation;


import java.util.Date;


import com.beetle.WebDemo.common.LoginInfo;
import com.beetle.framework.web.service.ICommonService;
import com.beetle.framework.web.service.WebRequest;
import com.beetle.framework.web.service.WebResponse;
```

```
import com.beetle.framework.web.service.WebServiceException;


public class LoginWebService implements ICommonService {


	public WebResponse perform(WebRequest req) throws WebServiceException {
		WebResponse response = new WebResponse();
		String userName = req.getValueAsString("UserName");
		int password = req.getValueAsInteger("Password").intValue();
		// 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()));
			response.setValueWithSimpleObject("Login_Info", loginInfo);
			response.setReturnFlag(0);
			response.setReturnMsg("登录成功！");
		} else {
			response.setReturnFlag(-1);
			response.setReturnMsg("用户名不存在或者密码不正确，请重新输入，谢
谢！");
		}
		return response;
	}
}
```

（2），在 WebController.xml 的<service>标签内注册此服务:

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings>
	<service>
		<srvItem name="LoginWebService.service"
```

```
                    class="com.beetle.WebDemo.presentation.LoginWebService"
/>
      </service>
</mappings>
```

（3），编程客户端，我们这里使用 java 的 awt 界面程序作为客户端：

```
...
public void actionPerformed(java.awt.event.ActionEvent e) {
      ta_info.setText(null);
      WebRequest input = new WebRequest();
      input.setValueWithString("UserName", tf_user.getText()      .trim());
      input.setValueWithInteger("Password", new
Integer(tf_password.getText().trim()));
      try {
      WebResponse output = ClientHelper.directInvoke(tf_url.getText().trim(),
input);
            StringBuffer sb = new StringBuffer();
            sb.append("==========结果=========\n");
            sb.append("----服务执行状态：\n");
            sb.append("flag:[" + output.getReturnFlag() + "]\n");
            sb.append("msg:[" + output.getReturnMsg() + "]\n");
            if (output.getReturnFlag() == 0) {
                  sb.append("----数据：\n");
      LoginInfo loginInfo = (LoginInfo)
output.getValueAsSimpleObject("Login_Info");
                  sb    .append("user:" + loginInfo.getLoginUser()      +
"\n");
                  sb.append("pass:" + loginInfo.getPassword() + "\n");
                  sb.append("logintime:"+
loginInfo.getLoginTime().toString());
            }
            ta_info.setText(sb.toString());
      } catch (RuntimeException e1) {
```

```
            e1.printStackTrace();
            ta_info.setText(e1.getMessage());
        }
}
...
```

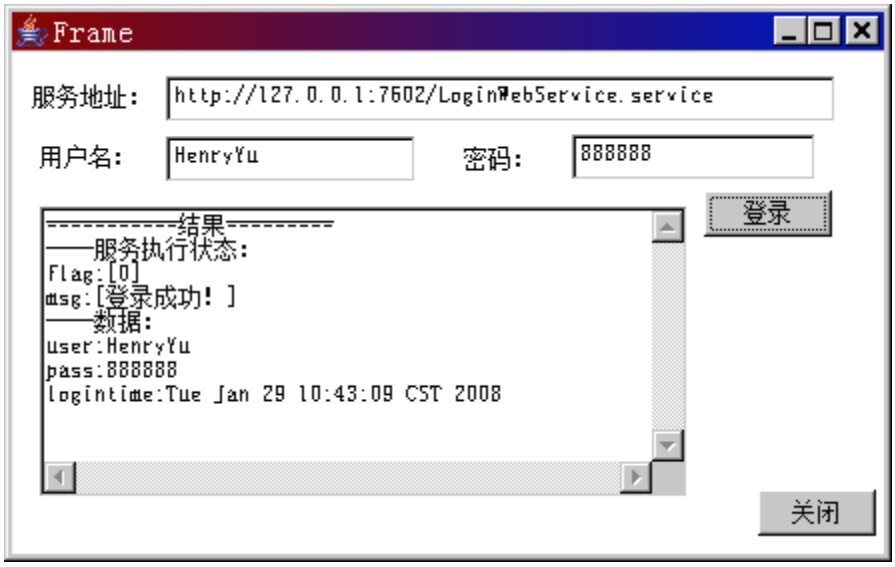


图 3-15 执行效果截图

分布式应用中有关程序间远程通信的问题一直是我们所关注的。最为原始也是最为有效率的通信方式是直接利用 Socket，通过自定义数据的格式来通信。一般地，一个完整的数据包，分为包头和包体两部分，头部分定义了数据的格式和长度等属性，而包体则为具体的数据载体。例如：移动的 CMPP 短信协议，但是，这种方式扩展性和灵活性都很差。特别是面对复杂数据格式数据包的时候，其打包/解包的算法会相应变得格外复杂。除非为了公司私有的通信协议，这种方式一般不推荐使用。

为了解决上面的问题，实现直接输送数据对象，出现了不少解决方案，例如，在 J2EE 世界我们熟悉的 RMI 就是其中一种。可是，由于防火墙的限制，我们就希望能否在 HTTP 协议基础上，来实现这种远程调用的通信方式？这样又出现了我们上面讨论 Web Services 的概念。面对形形色色的通信协议和方式，我们一定要根据自己的需求特点分析清楚，避免人云亦云，才能作出正确的选择。

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：

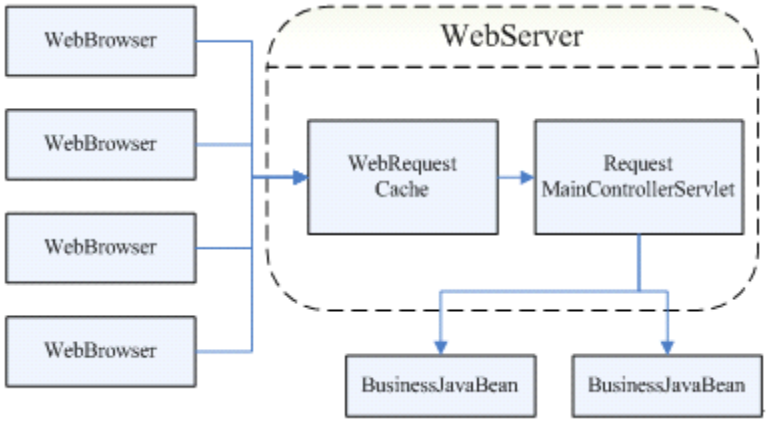


图 3-16 请求动态缓存原理示意图

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。

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。

AOP 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。

非 Ajax 横切

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。

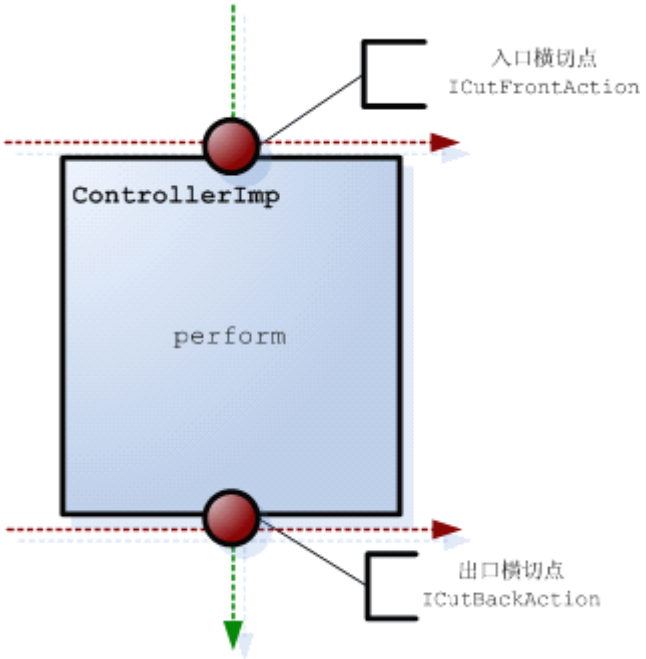


图 3-17 控制器横切编程示意图

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：

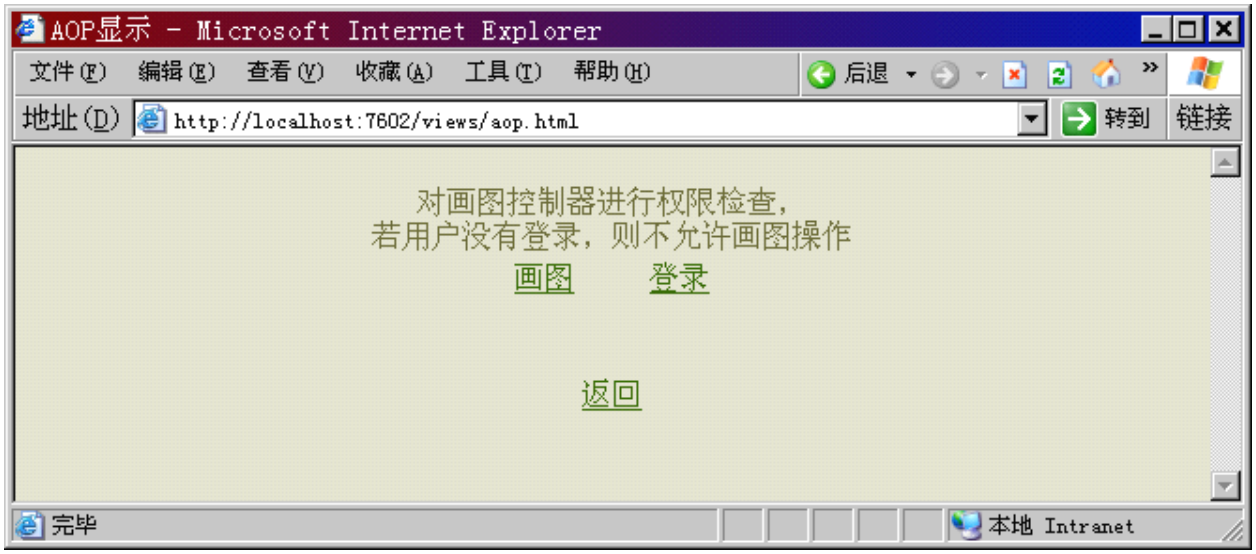


图 3-18 AOP 测试页面 1

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：

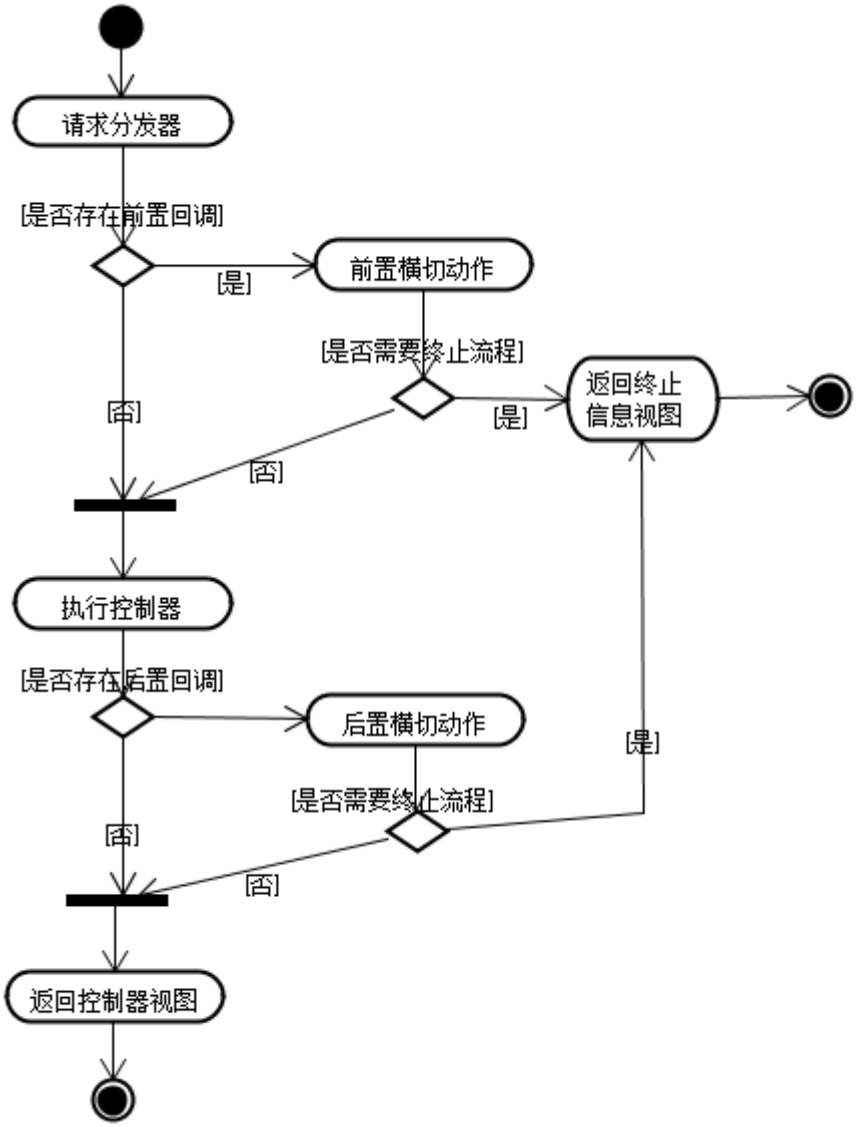


图 3-20 框架处理请求活动图

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。

Ajax 横切

BJAF 框架同样支持 Ajax 控制器横切动作编程，其实现机制和处理流程与上面非 Ajax 控制器是类似的。只是 Ajax 请求的不需要返回视图（由于无需刷新，视图就是其自身的页面），为了解决中断流程信息提示问题，框架采用 javascript 的 alert 函数来实现，因而图 3-20 的请求流程在 Ajax 中，活动图就调整为：

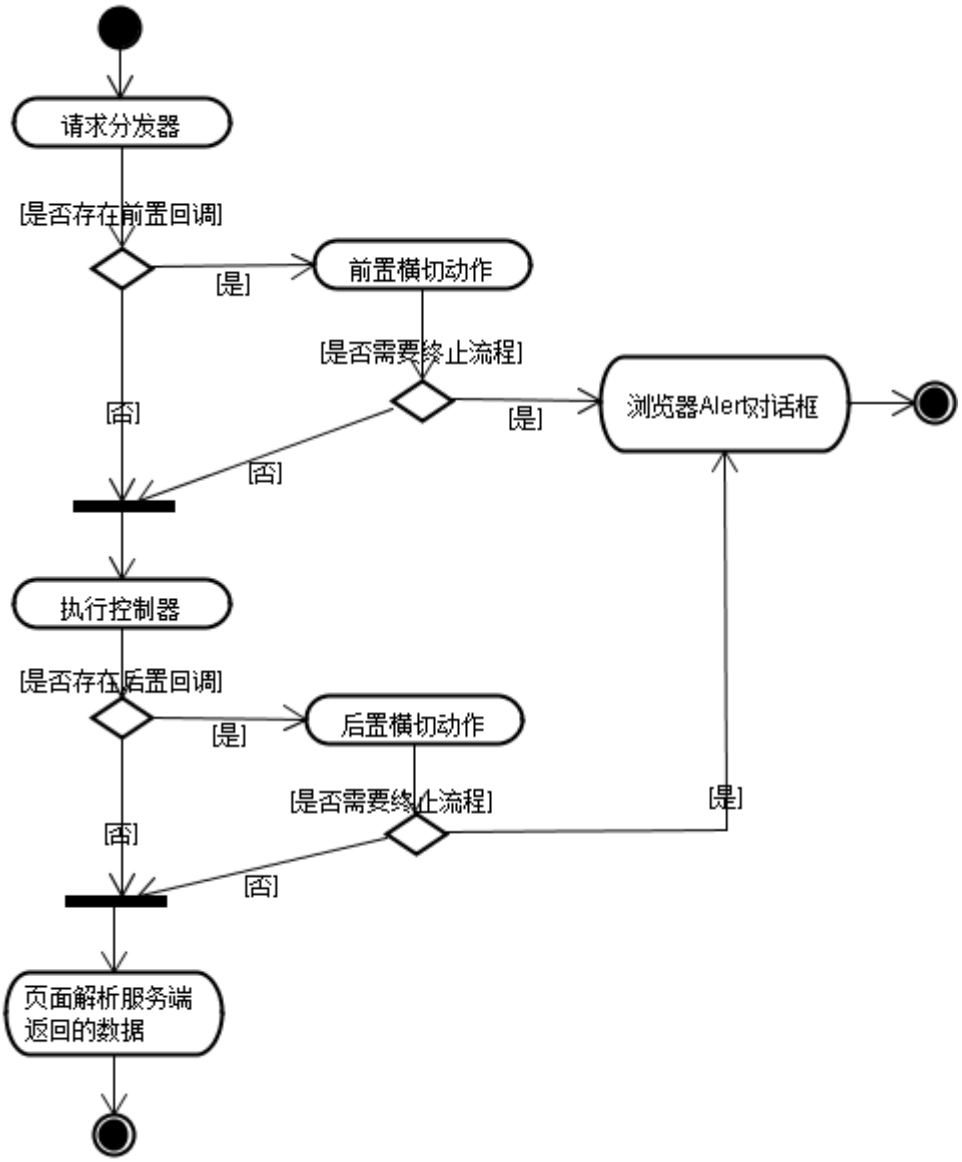


图 3-21 Ajax 请求过程活动图

与非 Ajax 横切一样，Ajax 也相应提供了前置、后置两个接口：

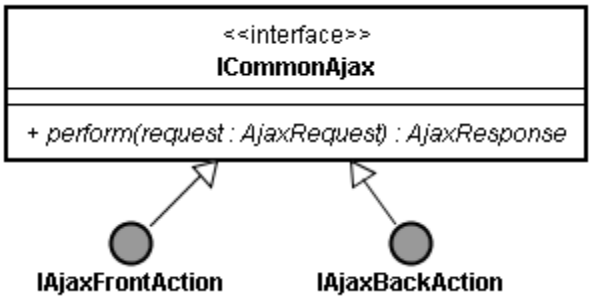


图 3-22 前、后置接口类图

可见，IAjaxFrontAction 和 IAjaxBackAction 都是继承自 ICommonAjax 编程接口，说明前、后置接口只是为了清晰语义而定义的，开发横切动作，实际上与开发一个 Ajax 控制器没有区别。

例如，添加一个前置横切动作，对 ShowDataAjaxController.ajax 获取用户列表 Ajax 控制器进行权限访问权限限制，代码如下：

```
package com.beetle.WebDemo.presentation.aop;


import com.beetle.WebDemo.common.LoginInfo;

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

import com.beetle.framework.web.controller.ajax.AjaxRequest;

import com.beetle.framework.web.controller.ajax.AjaxResponse;

import com.beetle.framework.web.controller.ajax.IAjaxFrontAction;


public class AjaxControllerFrontAction implements IAjaxFrontAction {


      public AjaxResponse perform(AjaxRequest req) throws ControllerException
{

            AjaxResponse res = new AjaxResponse();

            String ctrlName = req.getControllerName();

            if (ctrlName.equalsIgnoreCase("ShowDataAjaxController.ajax")) {

                  LoginInfo info = (LoginInfo)
req.getDataFromSession("LoginInfo");

                  if (info == null || !info.getLoginUser().equals("HenryYu"))
{

                        res.setBreakFlag(true);

                        res.setReturnMsg("你无权访问此功能，请先登录，谢谢！");

                  }

            }

            return res;

      }


}
```

注意：非 Ajax 横切通过返回一个 View 视图来终止流程，而 Ajax 横切是通过 setBreakFlag 方法设

置中断标记来中断流程。上面代码运行效果如下：

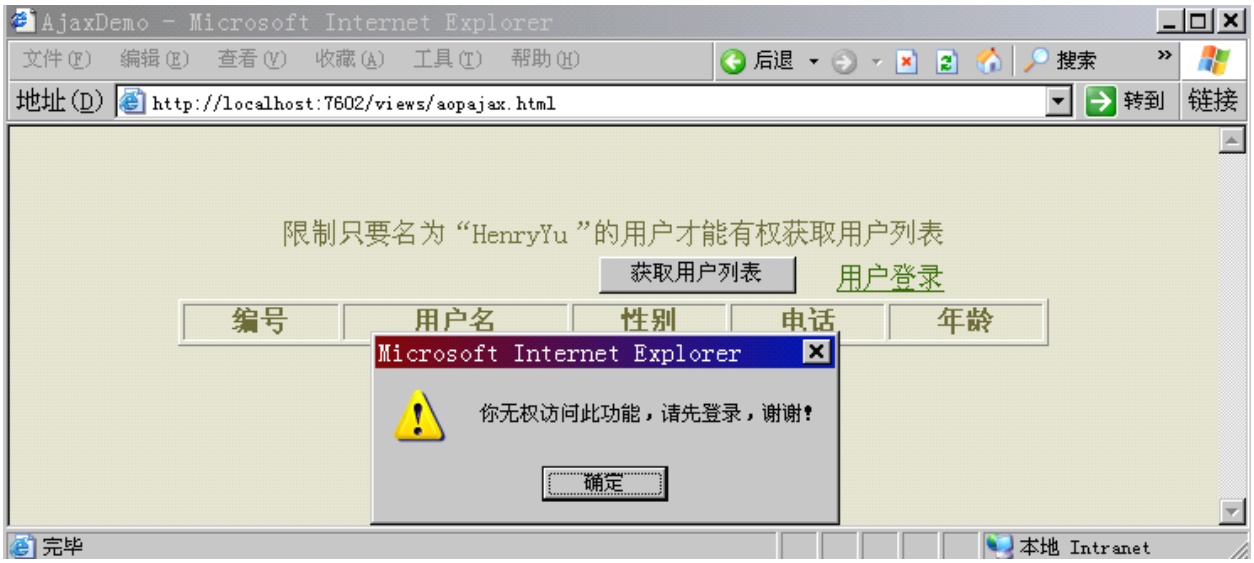


图 3-23 Ajax 横切运行效果

其它功能与特性

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>
```

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：

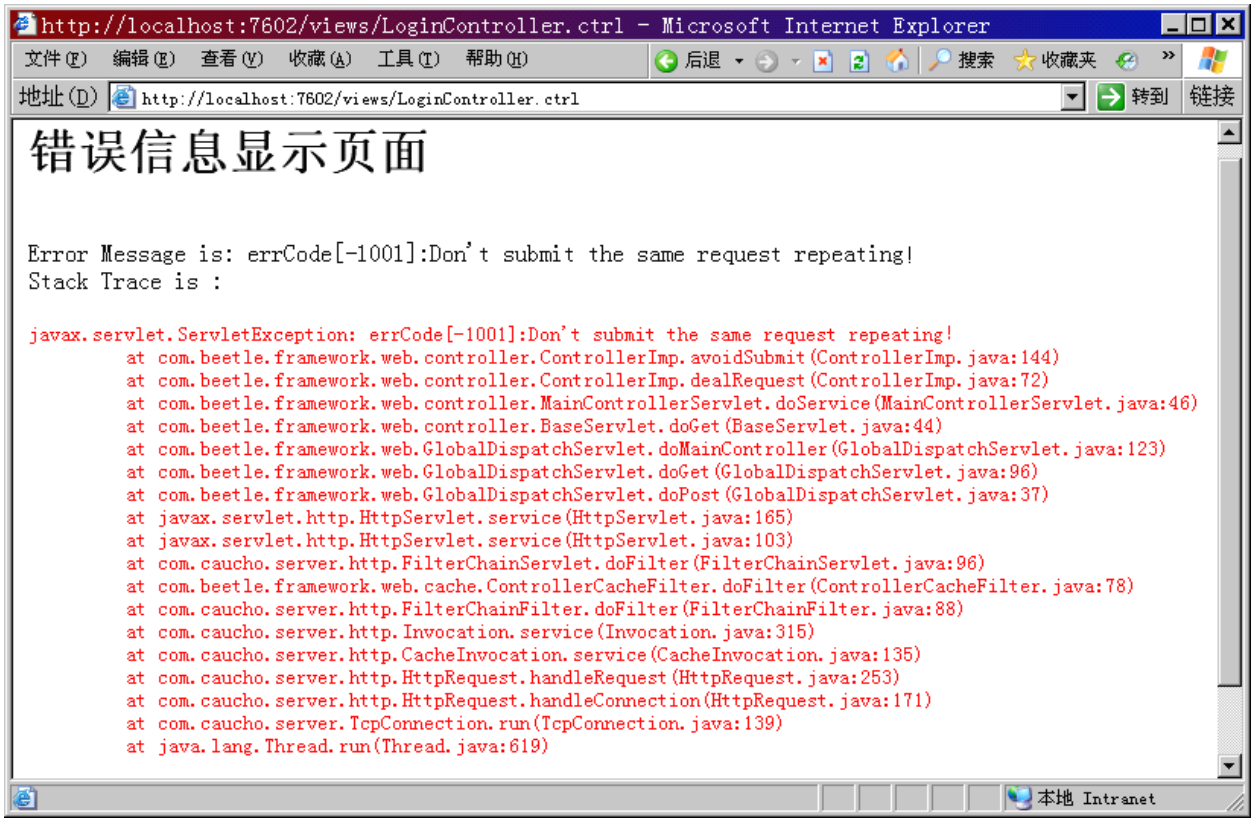


图 3-24 重复提交异常页面截图

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。

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();
      }
      ...
}
```

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。

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。

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。

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。

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：

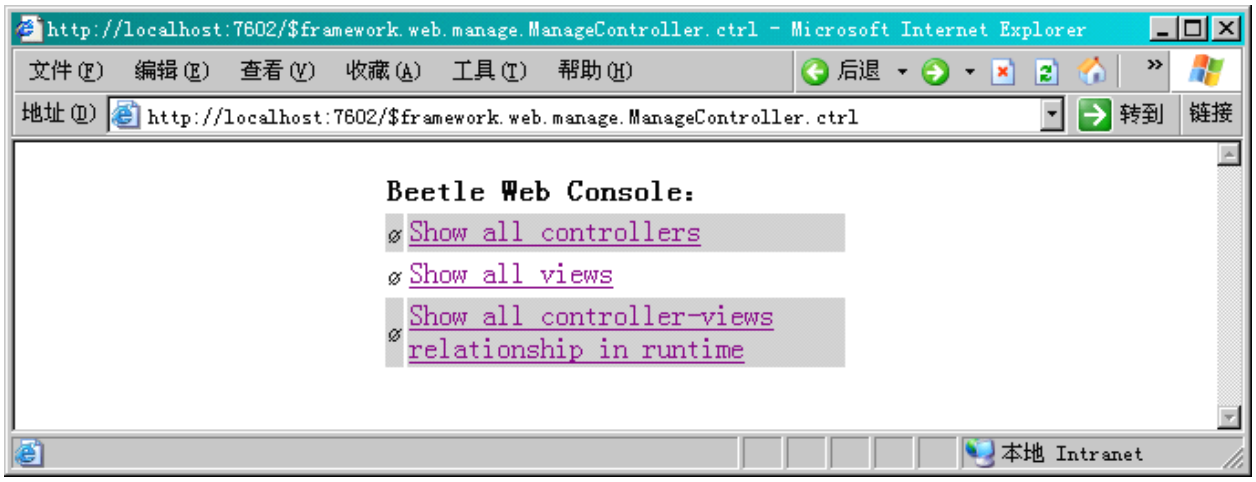


图 3-26 内置控制器管理界面截图

点击“Show all controllers”链接，显示所有控制器列表，见下图：

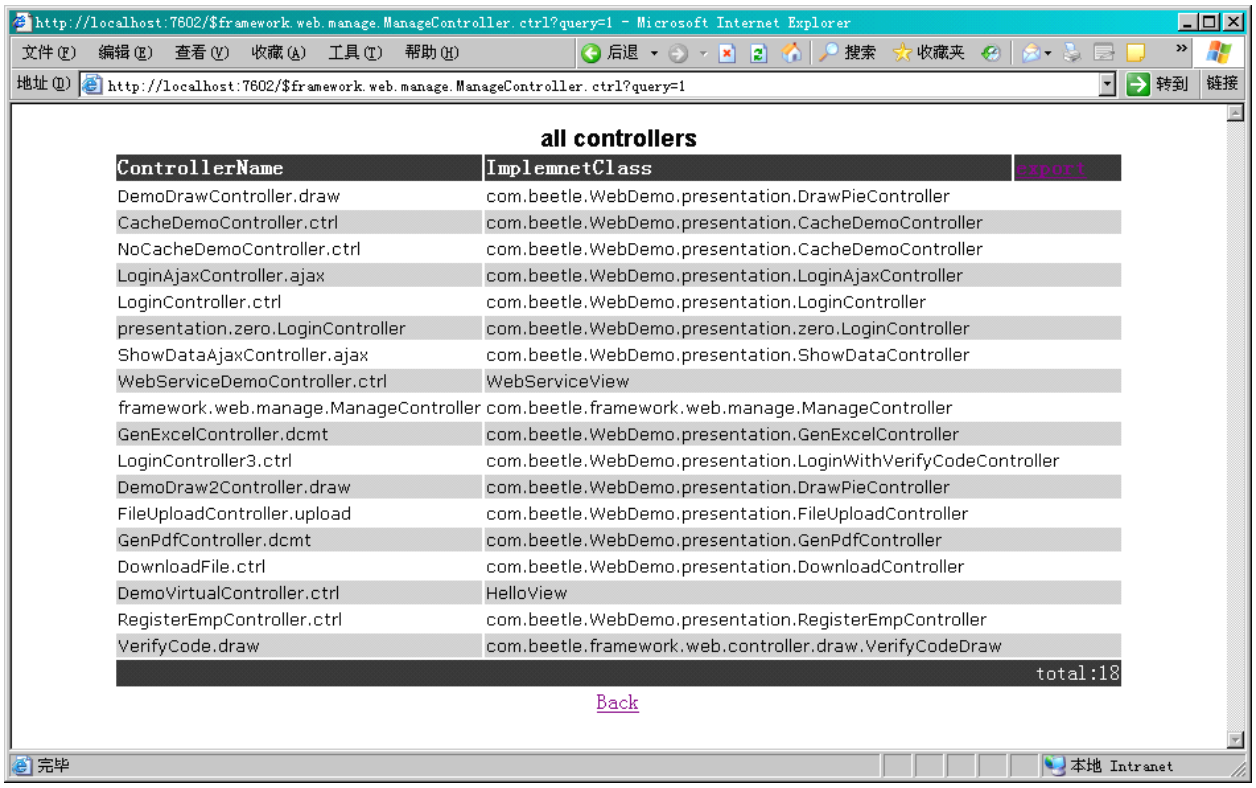


图 3-27 控制器列表截图

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：

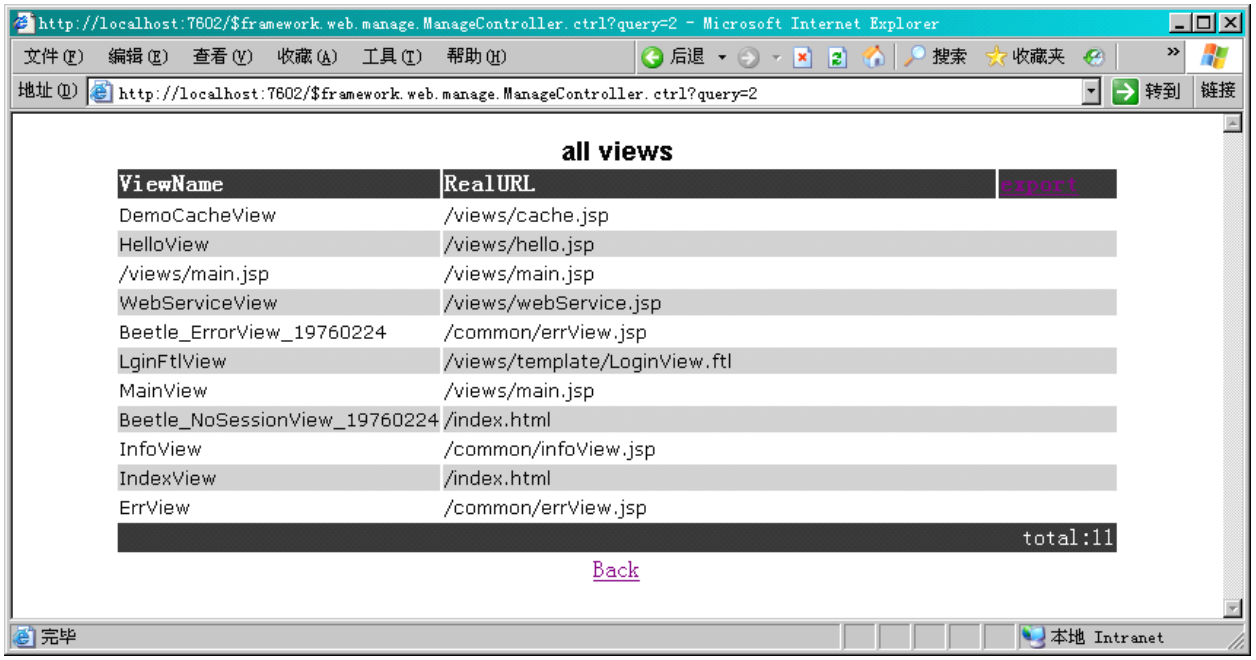


图 3-28 视图列表截图

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：

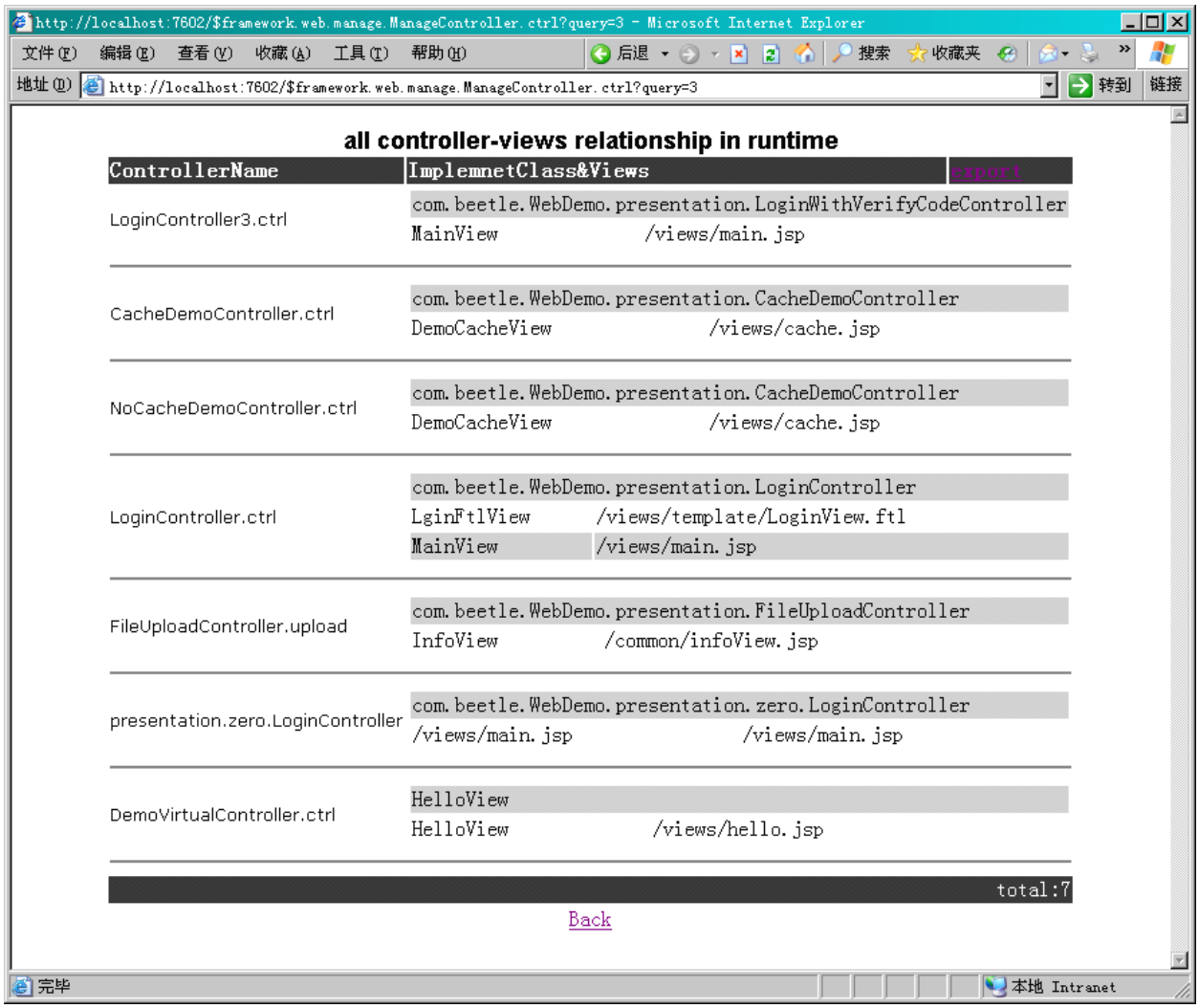


图 3-29 控制器视图引用关系列表截图

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。

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。

Web 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。

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。

业务层

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 Delegate 两种模式的同步框架、异步消息处理框架和 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 WebService 技术，相对以往臃肿不堪 WebService 方法，它开发简单、易于部署。

支持业务组件的 AOP 编程，提供前后回调编程接口。支持切入的的动态配置。

支持同步（Command、Delegate）和异步（MessageQueue）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（例如：在一个 Command 对象内所有的数据 DAO 接口可共享一条数据库连接）。

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。

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。

框架模型

BJAF Command 业务框架的模型如下：

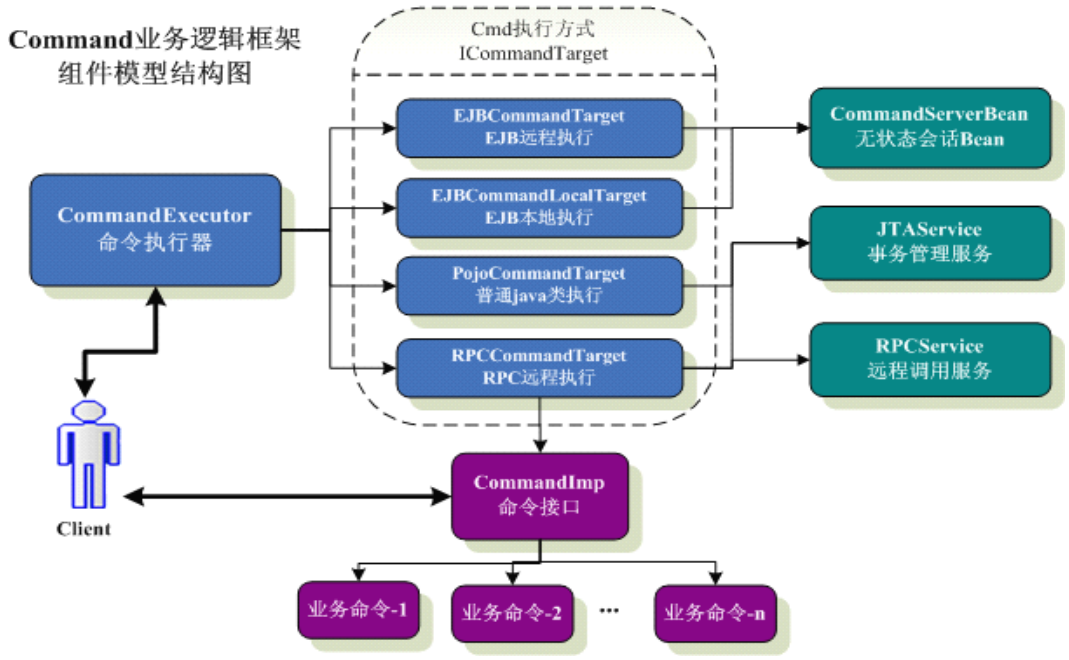


图 4-1 Command 业务组件模型

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：

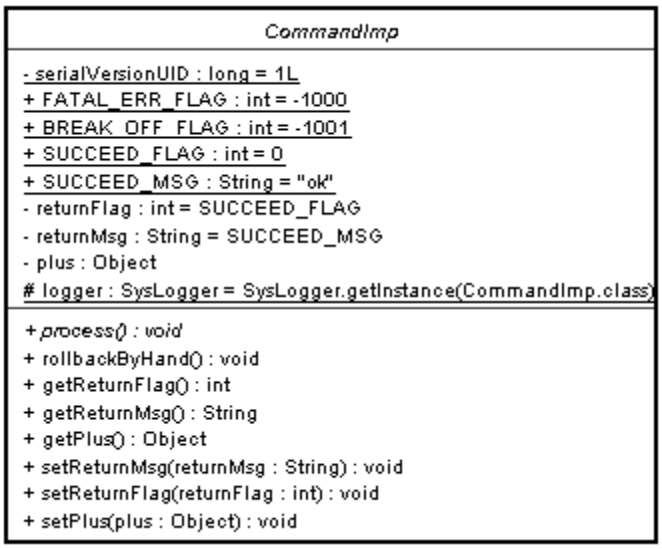


图 4-2CommandImp 编程接口类图

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：

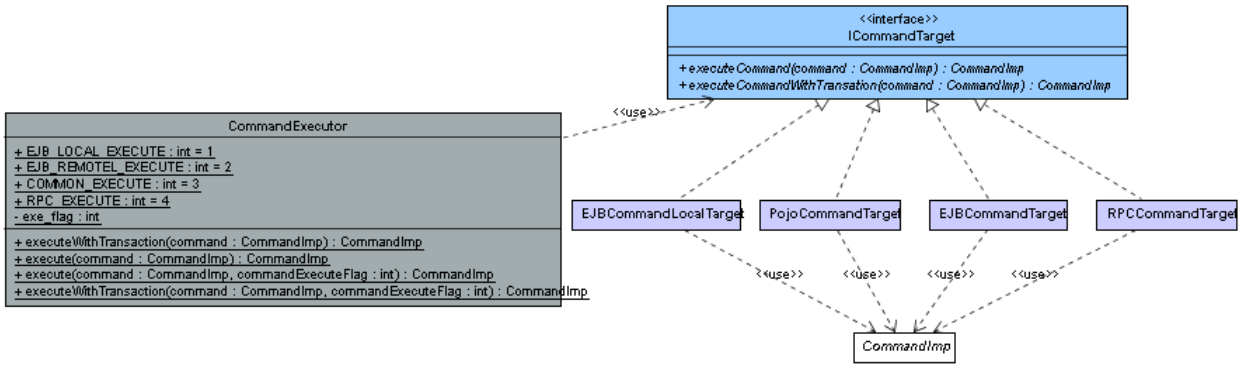


图 4-3 多种命令执行实现

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。

从图 4-3 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	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	为事务模式执行，其命令执行方法由参数commandExecuteFlag决定。

[1]ExeMode.xml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：

```
<?xml version="1.0" encoding="UTF-8"?>
<Mode>
	<Command>
		<!--
			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 设用于拥有 EJB 容器的应用服务器，1 适合集中式应用，2 适合分布式；
			*方式 3 和 4 设置适合没有 EJB 容器的应用服务器。
		-->
		<item name="COMMAND_EXECUTE_FLAG" value="2" />
		<!--
			CommandServerBean EJB 远程接口 JNDI 名称
			框架默认为名称为“ CommandServerBean”
		-->
		<item name="JNDI_NAME" value="CommandServerBean" />
		<!--
			CommandServerBean EJB 本地接口 JNDI 名称
```

```
                框架默认为名称为“ CommandServerBeanLocal”
            -->
            <item name="LOCAL_JNDI_NAME" value="CommandServerBeanLocal" />
            <!-- 是否缓存远程接口商业对象，默认为 false -->
            <item name="REMOTE_OBJECT_CACHE" value="false" />
            <!-- PRC 服务的 URL 地址，COMMAND.service 为固定服务命名 -->
            <item name="RPC_SERVICE_URL"
                  value="http://127.0.0.1:8080/COMMAND.service" />
      </Command>
</Mode>
```

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JOTM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hessian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。

从图 4-1 可知，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。

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。

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();
```

```
        }
}
```

2， 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 {
                cmd = (CreateAccountCmd) CommandExecutor
                                .executeWithTransaction(cmd);// 保证数据完整性，使用事务执
行
                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：

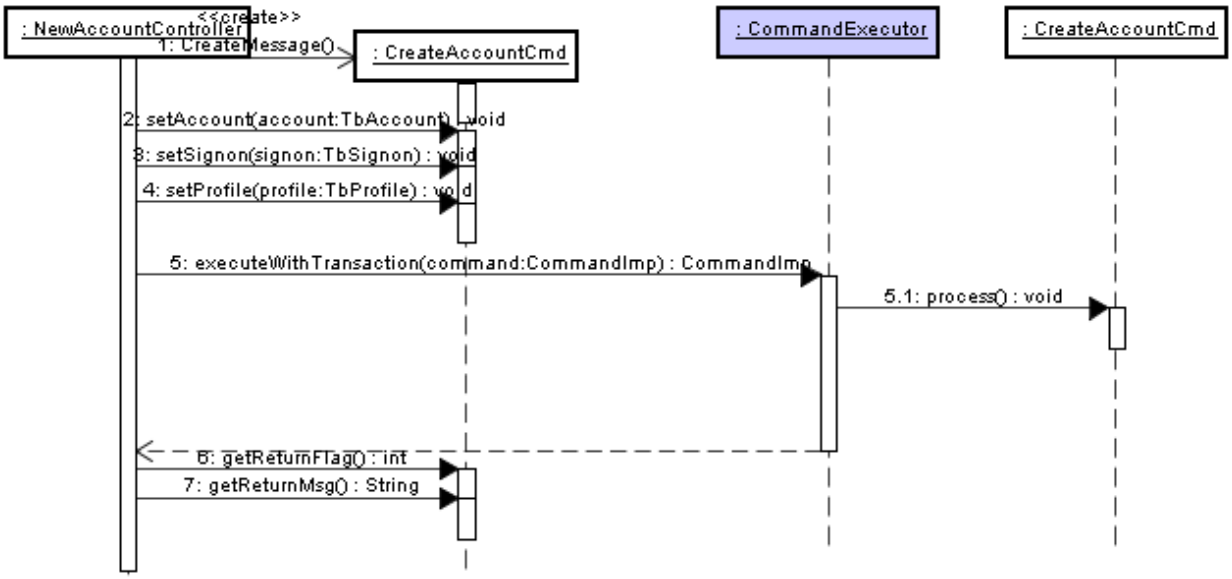


图 4-4 执行顺序图

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。

Delegate 框架

Delegate 业务框架目标与 Command 业务框架是一致的。唯一不同的是，它不再需要命令对象，而是把拆分上面讨论 3 种对象，把业务逻辑通过委托的方式提交给 EJB 容器或本框架所提供的相应服务来执行，其框架组件结构图如下：

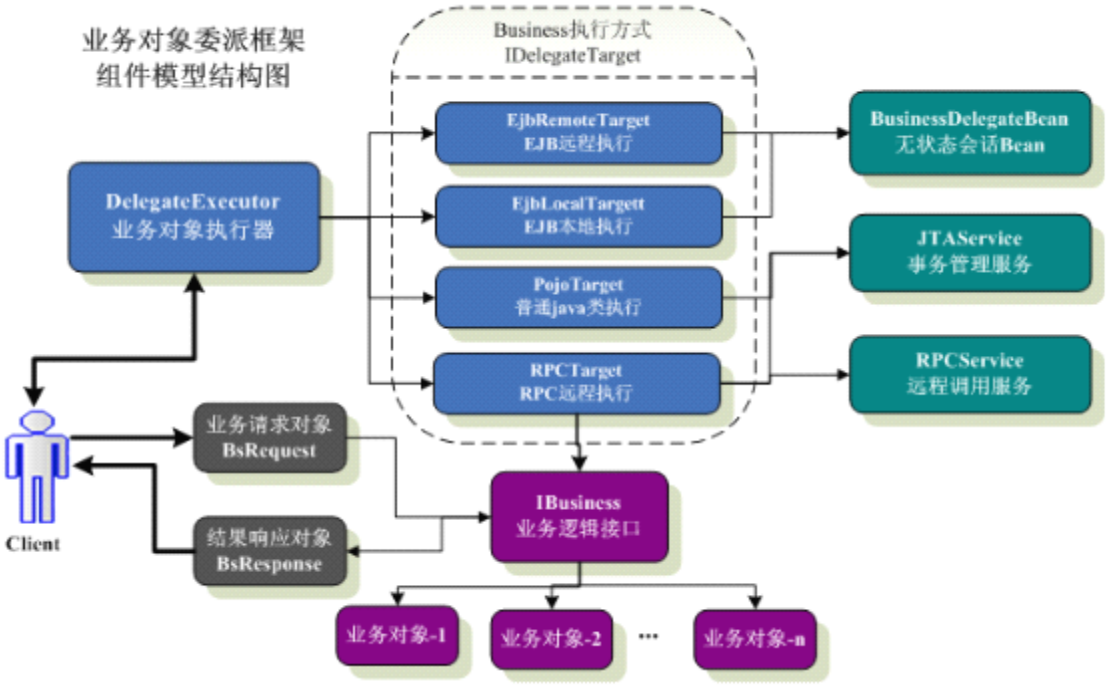


图 4-5Delegate 模型结构图

可见，其基本和 Command 框架一致，不同的是，现在客户端 Client 不再直接和业务对象交互。它只需发出请求输入业务参数（**BsRequest**），然后调用执行器（DelegateExecutor）委派给相关的 IDelegateTarget 去执行，就可以通过业务对象（**IBusiness**）结果响应对象（**BsResponse**）获取执行结果。至于业务对象如何执行，完全有不同类型 IDelegateTarget 决定，客户端无需关心。这一点，与同 Command 框架如出一辙。框架主要类的类图如下：

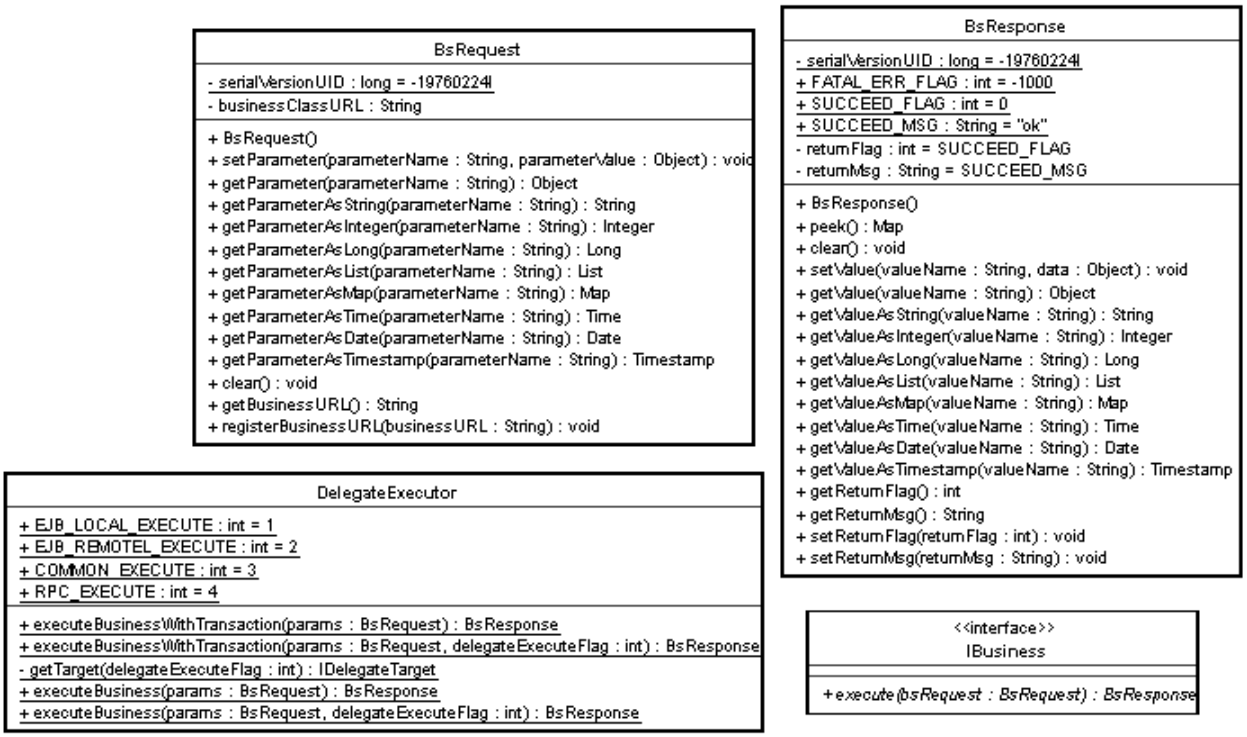


图 4-6Delegate 框架主要类图

上面的 Command 框架的例子如果换成 Delegate 框架，代码如下：

1，业务对象代码：

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


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

import com.beetle.PetStore.business.storefront.account.QueryAccount;

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.TbProfile;

import com.beetle.PetStore.valueobject.TbSignon;

import com.beetle.framework.business.delegate.BsRequest;

import com.beetle.framework.business.delegate.BsResponse;

import com.beetle.framework.business.delegate.IBusiness;
```

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


public class CreateAccountBs implements IBusiness {


   public BsResponse execute(BsRequest req) {
      BsResponse rs = new BsResponse();
      rs.setReturnFlag(0);
      rs.setReturnMsg("ok");
      int stepNo = req.getParameterAsInteger("stepNo").intValue();
      if (stepNo == 1) {// 查询 TbBannerdata
            QueryAccount qa = new QueryAccount(null, 2);
            try {
                  qa.query();
                  rs.setValue("bannerDatas", qa.getBannerDatas());
            } catch (BusinessObjectCommonException e) {
                  rs.setReturnFlag(-1);
                  rs.setReturnMsg(e.getMessage());
            }
      } else if (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");
            TbSignon signon = (TbSignon) req.getParameter("signon");
            signonDao.insertSignon(signon);
            TbAccount account = (TbAccount) req.getParameter("account");
            TbProfile profile = (TbProfile) req.getParameter("profile");
            accountDao.insertAccount(account);
```

```
            profileDao.insertProfile(profile);
        }
        return rs;
    }
}
```

2，执行此业务对象代码：

```
public static void main(String arg[]) {
        // 构建请求参数对象
        BsRequest req = new BsRequest();
        req.setParameter("stepNo", new Integer(2));
        req.setParameter("signon", new TbSignon());
        req.setParameter("account", new TbAccount());
        req.setParameter("profile", new TbProfile());
        // 注册业务类
        req.registerBusinessURL
("com.beetle.PetStore.business.storefront.account.CreateAccountBs");
        // 执行
        BsResponse rs = DelegateExecutor.executeBusinessWithTransaction(req);
        if (rs.getReturnFlag() >= 0) {
            // 成功。。。
        } else {
            // 处理失败代码
        }
}
```

Command 业务框架和业务对象委派框架，它们各有特点，虽然 Command 框架在部署上比较笨拙，网络传输数据可能冗余，但是从开发业务对象的语义上是很清晰、容易理解，而且编写 Command 业务对象的范式也是非常一致的。

根据笔者项目团队开发的经验，开发人员对 Command 对象比 Bs 业务对象似乎更受落。业务对象委派框架解决了 Command 框架缺陷，但是又引入了两个输入输出对象，参数无法做到类型的有效性检查，使开发不够利索。不过，Bs 业务对象可以通过编写方式来保证本身的线程安全，这样带来的另

一个好处就是可以缓存它们，从而改善系统性能并提高效率。所以，理论上委派框架比 Command 框架来得有效率，在这一点上笔者更喜欢业务对象委派框架来开发业务系统。当然，它们并不是矛盾的，在具体项目中，根据需求的不同，Command 框架和委派框架是可以同时并用、互为补充的。

值得探讨的是，有种观点认为 Command 框架和业务委派框架的使用会导致代码臃肿和代码重要性不高的风险。因为一般地，我们推荐一个业务对象封装一个业务用例（Use Case），这往往改变了团队的工作方式，不必再编写共用的代码。这其实引发了“用例能不能重用的问题”。显然，从实际的业务角度来说，用例怎么能重用呢？！当然，实现每个用例的部分代码是可以重用的，如 DAO 层，还有一些用例的共通部分，这些都可以抽象成一个通用类。这其实已经不再框架能所控制的范围，这更多体现在业务需求的理解、业务粒度粗细划分、分析和设计等领域上面。

横切编程

待续。。。

异步消息框架

BJAF 框架 1.3.x 版本对异步消息框架会进行重新设计，对使用模式进行很多增强。目前版本还在调试中，我们将于 1.4.x 版本中发布。

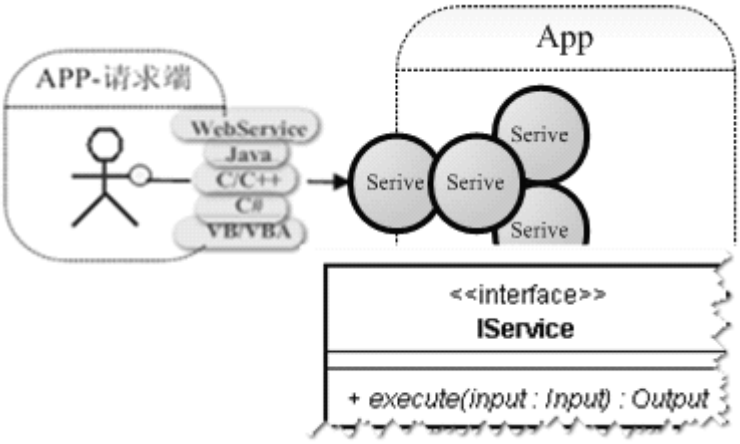
待续。。。

Service 服务组件

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，系统内部服务的定义和模型我们是没有介入的。

BJAF 框架 Service 组件的目标是实现服务暴露，标准接口，一种实现，多种不同形式的暴露，来解决系统间交互的技术难点。其架构示意图如下：



服务暴露示意图

Service 编程

待续。。。

服务程序开发

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。

支持子程序（SubRoutine）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）

基于消息对象网络 NIO 通信模块。

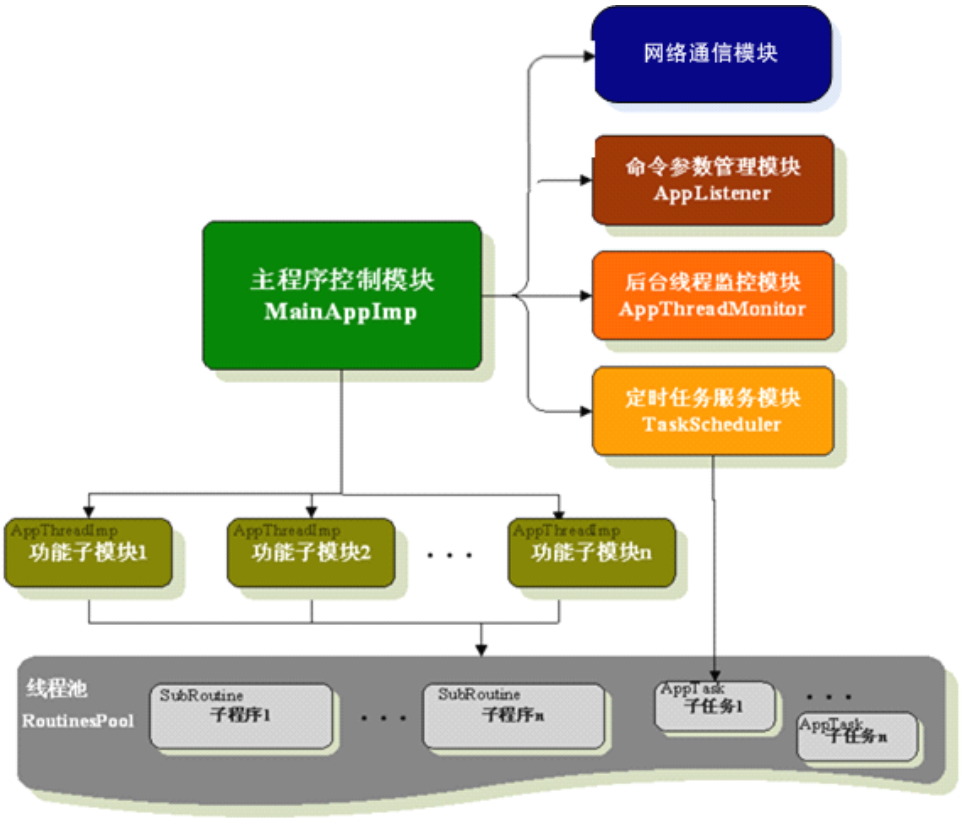


图 5-1 应用服务程序框架模块结构图

应用程序模块

从图 5-1 结构图可以知道：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：

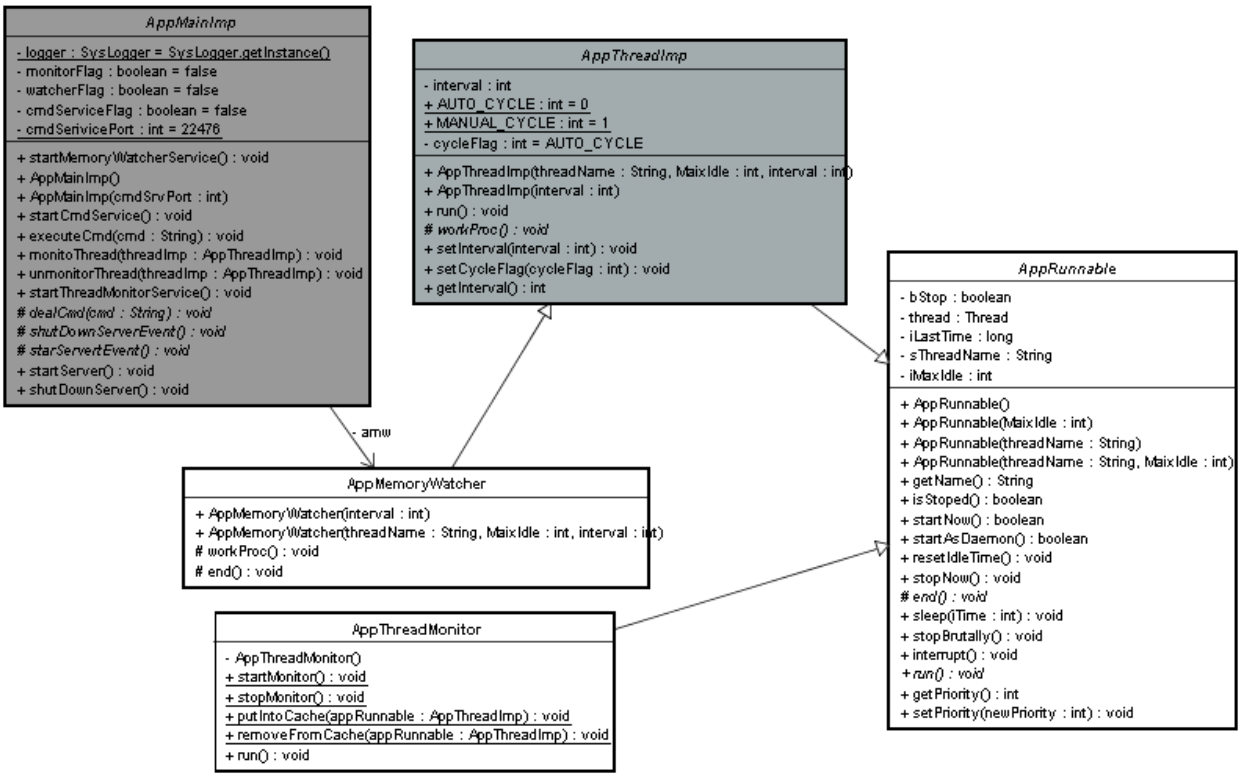


图 5-2 应用程序模块类图

从图 5-2 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：（建立一个 ModuleDemo 功能子模块，建议一个 AppDemo 应用服务程序）

（1），功能子模块，本质上为一个工作线程，继承自 AppThreadImp 抽象类：

```
package test.appsrv.app;

import com.beetle.framework.appsrv.AppThreadImp;

public class ModuleDemo extends AppThreadImp {

    public ModuleDemo(String threadName, int MaixIdle, int interval) {
```

```
            super(threadName, MaixIdle, interval);
      }
      protected void workProc() {
            System.out.println("module work...");
      }
      protected void end() {
            System.out.println("end");
      }
}
```

（2），编写一个 AppDemo 应用服务程序

```
package test.appsrv.app;

import com.beetle.framework.appsrv.AppMainImp;

public class AppDemo extends AppMainImp {
      private ModuleDemo md;

      public static void main(String arg[]) {
            AppDemo app = new AppDemo();
            if (arg.length == 0 || arg[0].equalsIgnoreCase("start")) {
                  app.startServer();// 启动服务
            } else {
                  app.executeCmd(arg[0]);// 发送命令参数
            }
      }
      protected void dealCmd(String cmd) {
            // 处理参数命令
            if (cmd == null) {
                  return;
            }
```

```
            if (cmd.equalsIgnoreCase("help")) {
                  System.out.println("--help");
                  System.out.println("--shutdown");
            } else if (cmd.equalsIgnoreCase("shutdown")) {// 关闭服务程序命令
                  this.shutDownServer();// 关闭 AppDemo 服务程序
            }
      }
      protected void shutDownServerEvent() {
            System.out.println("shutDownServerEvent called!");
      }
      protected void starServertEvent() {
            // 在启动事件中，做相关初始化工作
            this.startCmdService();// 启动参数名称服务
            this.startThreadMonitorService();// 启动后台功能模块线程监控服务
            this.startMemoryWatcherService();// 启动内存监控服务
            md = new ModuleDemo("ModuleDemo", 30, 3000);// 构建 ModuleDemo 功能
模块
            this.monitoThread(md);// 监控此模块线程运行
            md.startNow();// 启动此模块
            // ...
      }
}
```

（3）执行此程序：（java test.appsrv.app.AppDemo start）

```
loaded /config/log4j.properties from file
0    INFO  [main]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMainImp$CmdListener]CmdListener started and
the port:22476
0    INFO  [main]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMainImp]CmdService started!
0    INFO  [main]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMainImp]ThreadMonitorService started!
31   INFO  [main]    com.beetle.framework.log.SysLogger     -
```

```
[com.beetle.framework.appsrv.AppMainImp]MemoryWatcherService started!

31   DEBUG [main]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMainImp]monitor:ModuleDemo

31   DEBUG [Thread-2]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMemoryWatcher]gcRate=0.8,memoryUsedRate=0.286
4951

31   INFO  [main]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMainImp]Application Server started!

module work...

module work...

3031 DEBUG [Thread-2]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMemoryWatcher]gcRate=0.8,memoryUsedRate=0.292
4096

module work...

6031 DEBUG [Thread-2]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMemoryWatcher]gcRate=0.8,memoryUsedRate=0.294
65017

module work...

9031 DEBUG [Thread-2]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMemoryWatcher]gcRate=0.8,memoryUsedRate=0.296
94194
```

（4），通过“shutdown”命令关闭此服务程序（java test.appsrv.app.AppDemo shutdown）

```
276039 DEBUG [Thread-2]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMemoryWatcher]gcRate=0.8,memoryUsedRate=0.242
39226

278398 INFO  [Thread-0]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMainImp$CmdListener]parameter:shutdown

278680 INFO  [Thread-4]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMainImp]Application Server shutdown..

278680 INFO  [Thread-4]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppMainImp$CmdListener]stop the CmdListener

278680 DEBUG [Thread-4]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppThreadMonitor]ModuleDemo

end

end
```

```
278867 INFO  [Thread-4]    com.beetle.framework.log.SysLogger     -
[com.beetle.framework.appsrv.AppThreadMonitor]all the threads by monitored
had stopped!

278883 INFO  [Thread-4]    com.beetle.framework.log.SysLogger     -
[AppMemoryWatcher]stopped.

shutDownServerEvent called!
```

从控制台信息可知，服务程序正常关闭。

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

其对外线程编程接口是 AppThreadImp 抽象类，参考下面的类图：

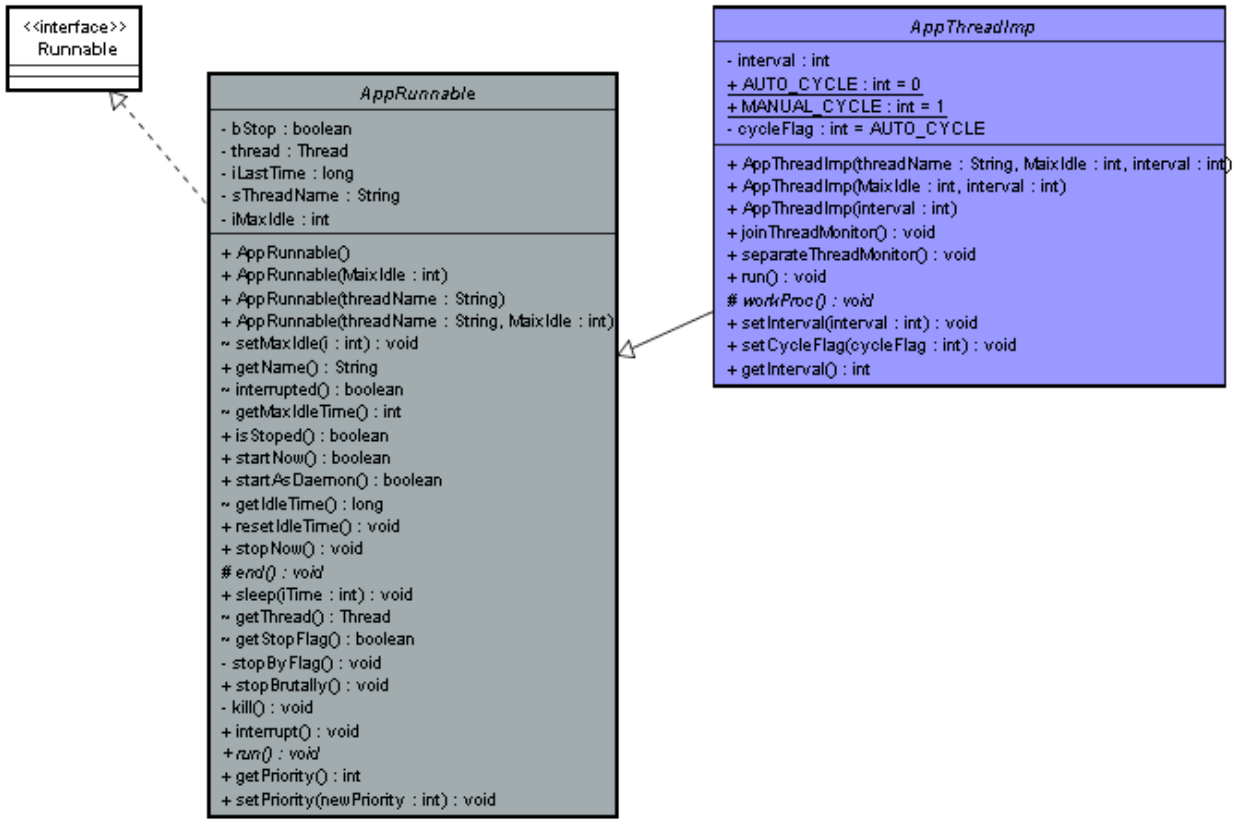


图 5-3 线程类图

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.appsrv;


import com.beetle.framework.appsrv.AppThreadImp;


public class SimpleThread extends AppThreadImp {


	public SimpleThread(int interval) {
		super(interval);
	}
	protected void workProc() {// 工作方法，会自动循环执行，间隔时间 interval 毫秒
		System.out.println(System.currentTimeMillis() + "-->do
something...");
	}
	protected void end() {// 线程结束时，触发的事件
		System.out.println(System.currentTimeMillis() + "-->end");
	}
}
```

编写执行客户端端，代码如下：

```
package example.appsrv;


public class TestClient {


	public static void main(String[] args) throws Throwable {
		SimpleThread st=new SimpleThread(2000);//2 秒钟循环执行一次
		//st.joinThreadMonitor();//加入线程监控（若执行超时，监控器会将其重新拉
起）
		//st.startAsDaemon();//以守护线程模式启动
		st.startNow();
		Thread.sleep(10000);
```

```
            //st.separateThreadMonitor();//脱离监控
            //st.stopBrutally();//粗暴结束线程
            st.stopNow();//停止线程
      }
}
```

执行结果：

```
1235526139007-->do something...
1235526141009-->do something...
1235526143011-->do something...
1235526145012-->do something...
1235526147014-->do something...
1235526149016-->end
```

上面例子，由客户端主动调用 stopNow()结束线程；我们常常需要线程自身根据某些条件自己结束，这种情况，上面 SimpleThread 可改为：

```
package example.appsrv;

import com.beetle.framework.appsrv.AppThreadImp;

public class SimpleThread extends AppThreadImp {

      private static final long serialVersionUID = 1L;
      public SimpleThread(int interval) {
            super(interval);
      }
      private int i = 0;
      protected void workProc() {// 工作方法，会自动循环执行，间隔时间 interval 毫秒
            System.out.println(System.currentTimeMillis() + "-->do
something...");
            i++;
            if (i > 4) {
                  this.stopNow();
```

```
            }
        }
        protected void end() {// 线程结束时，触发的事件
            System.out.println(System.currentTimeMillis() + "-->end");
        }
}
```

执行代码：

```
package example.appsrv;

public class TestClient {

    public static void main(String[] args) throws Throwable {
        new SimpleThread(2000).startNow();
        Thread.sleep(10000);
    }
}
```

执行结果：

```
1235526856865-->do something...
1235526858866-->do something...
1235526860867-->do something...
1235526862868-->do something...
1235526864868-->do something...
1235526864868-->end
```

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 AppThreadImp 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，我们定义为子程序（SubRoutine），我们下一小节介绍。

子程序及其执行方式

子程序（SubRoutine）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，针对子程序实际运行情况，还实现了一个针对子程序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。

对于子程序，BJAF 框架提供了线程池来优化其执行效率，同时对子程序的执行方式做了封装，（RoutineExecutor）提供做多不同执行方式。参见下面类图：

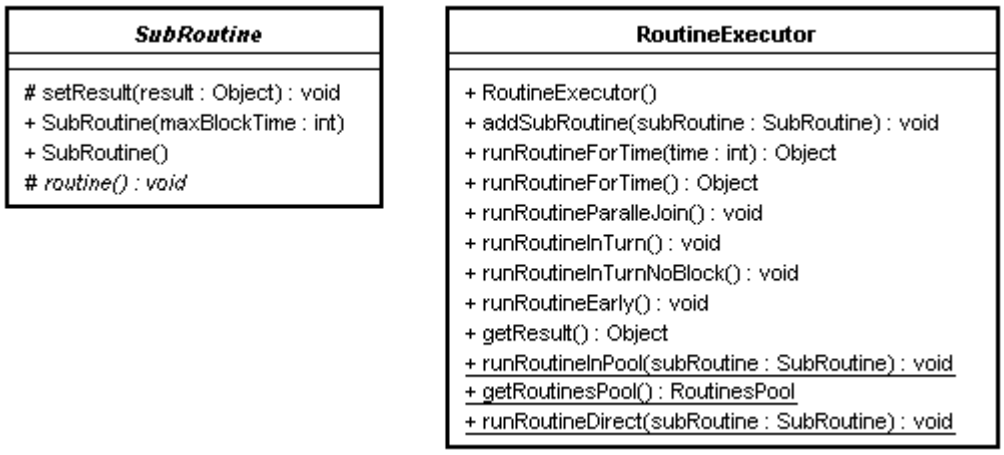


图 5-4 子程序类图

子程序说明示例

SubRoutine 类说明如下：

方法与属性	功能说明
SubRoutine(maxBlockTime : int)	构造函数，调用此构造函数实例化一个子程序，代表此子程序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，框架会对这个子程序进行超时中断处理。
SubRoutine()	构造函数，调用此构造函数实例化一个子程序，不参与后台线程超时机制监控。
routine() : void	子程序运行，为抽象方法，子程序执行任务技术所必须实现。
setResult(result : Object) : void	设置子程序返回结果 routine方法体内调用才有效。适合需要返回结果的子程序（带超时执行机制） 由RoutineExecutor.runRoutineForResult方法执行

例如，建一个简单子程序，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.SubRoutine;


public class DemoRoutine extends SubRoutine {


	public DemoRoutine() {
		super();
	}
	protected void routine() throws InterruptedException {
		System.out.println(System.currentTimeMillis() + "->do
something...");
	}
}
```

利用 RoutineExecutor 子程序执行：

```
package example.appsrv.routine;


import com.beetle.framework.appsrv.RoutineExecutor;


public class TestClient {


	public static void main(String[] args) {
		RoutineExecutor.runRoutineInPool(new DemoRoutine());
	}
}
```

执行结果：

```
loaded /config/log4j.properties from file
1235541073513->do something...
```

下面演示一下，子程序超时回收功能。假设，上面的 DemoRoutine 在开发的时候不小心写了个死

循环，导致子程序长时间吊死。此时，我们能够及时回收此子程序线程就显得很有必要了。

```
package example.appsrv.routine;

import com.beetle.framework.appsrv.SubRoutine;

public class DemoRoutine extends SubRoutine {

	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...");
		}
	}
}
```

执行此子程序：

```
package example.appsrv.routine;

import com.beetle.framework.appsrv.RoutineExecutor;

public class TestClient {

	public static void main(String[] args) {
		RoutineExecutor.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.SubRoutine.run(SubRoutine.java:114)

      at
com.beetle.framework.appsrv.RoutinesPool$ThreadPoolExecutor$Worker.run(Routin
esPool.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程序（线程）给及时中断回收。

执行方式说明及示例

从图 5-4 类图中，可知 RoutineExecutor 提供了以下执行方式：

方法与属性	功能说明
runRoutineDirect(subRoutine: SubRoutine) : 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程序放在线程池内执行
runRoutineInPool(subRoutine: SubRoutine) : void	在线程池[1]中，执行此子程序。
addSubRoutine(subRoutine: SubRoutine) : void	往执行器中，添加一个子程序。（执行器，支持多个子程序一起执行）
runRoutineEarly() : void	提早执行子程序，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	获取此子程序的处理结果（此方法会阻塞）
runRoutineForTime() : Object	执行子程序并等待返回其计算结果。 根据此子程序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程序 并触发子程序的terminate()事件

	（方法） @return 子程序结果
runRoutineForTime(time : int) : Object	同上。 只是支持自定义等待时间
runRoutineParalleJoin() : void	并行执行子程序，并等待所有的子程序结束后再退出（针对一组子程序，此方法会阻塞）（没有超时处理机制，即使子程序设置最大阻塞时间也无效）
runRoutineInTurn() : void	依次执行子程序（按照顺序前一个子程序运行完毕才接着运行下一个，直到所有的子程序执行完毕）（针对一组子程序）

[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。

常规执行一个子程序前面代码已经演示过，下面示例一下其它特别执行方式：

针对一组子程序，并行执行，并等待所有子程序结束后再返回

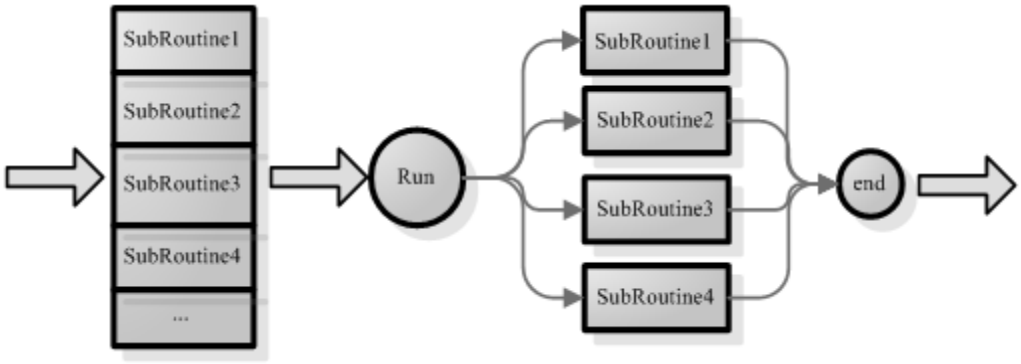


图 5-5 并行执行示意图

构建 3 个子程序，代码分别为，SR1 代码：

```
package example.appsrv.routine;
import com.beetle.framework.appsrv.SubRoutine;
public class SR1 extends SubRoutine {
```

```
	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.SubRoutine;
public class SR2 extends SubRoutine {
	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.SubRoutine;
public class SR3 extends SubRoutine {
	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.RoutineExecutor;
public class TestParalleClient {
```

```
    public static void main(String[] args) {
        // 构建一个子程序执行器
        RoutineExecutor re = new RoutineExecutor();
        re.addSubRoutine(new SR1());// 添加各个子程序到执行器队列
        re.addSubRoutine(new SR2());
        re.addSubRoutine(new SR3());
        re.runRoutineParalleJoin();// 并行执行，并阻塞，等待队列中所有子程序都结束才返回
        System.out.println("ok");
    }
}
```

执行结果：

```
sr1-begin
sr3-begin
sr2-begin
sr3-end
sr2-end
sr1-end
ok
```

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。

针对一组子程序，依次串行执行，并等待所有子程序结束后再返回

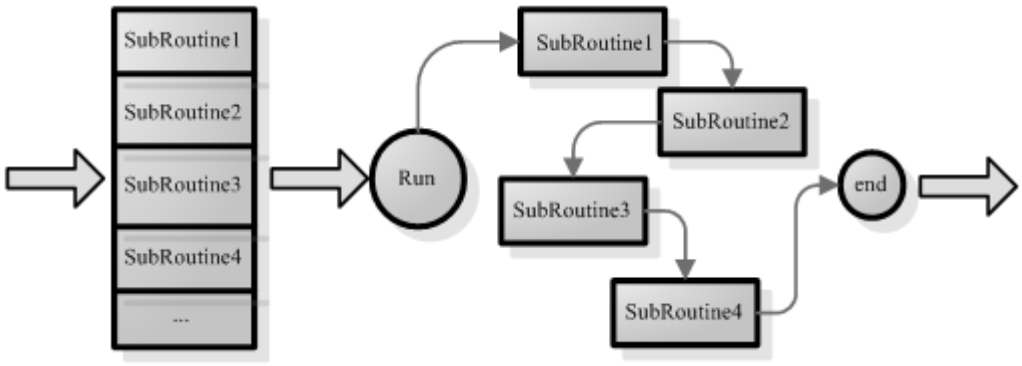


图 5-6 串行执行示意图

沿用前面的 3 个子程序，串行执行的客户端代码如下：

```
package example.appsrv.routine;
import com.beetle.framework.appsrv.RoutineExecutor;
public class TestInTurnClient {


	public static void main(String[] args) {
		// 构建一个子程序执行器
		RoutineExecutor re = new RoutineExecutor();
		re.addSubRoutine(new SR1());// 添加各个子程序到执行器队列
		re.addSubRoutine(new SR2());
		re.addSubRoutine(new SR3());
		re.runRoutineInTurn();// 串行执行，并阻塞，等待队列中所有子程序都结束才返回
		System.out.println("ok");
	}
}
```

执行结果：

```
sr1-begin
sr1-end
sr2-begin
sr2-end
sr3-begin
```

```
sr3-end

ok
```

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程序，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 子程序：

```
package example.appsrv.routine;

import java.util.ArrayList;

import java.util.List;

import com.beetle.framework.appsrv.SubRoutine;

public class HardWorkSR extends SubRoutine {

      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.RoutineExecutor;


public class TestEarlyClient {

	public static void main(String[] args) {

		RoutineExecutor re = new RoutineExecutor();

		re.addSubRoutine(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：

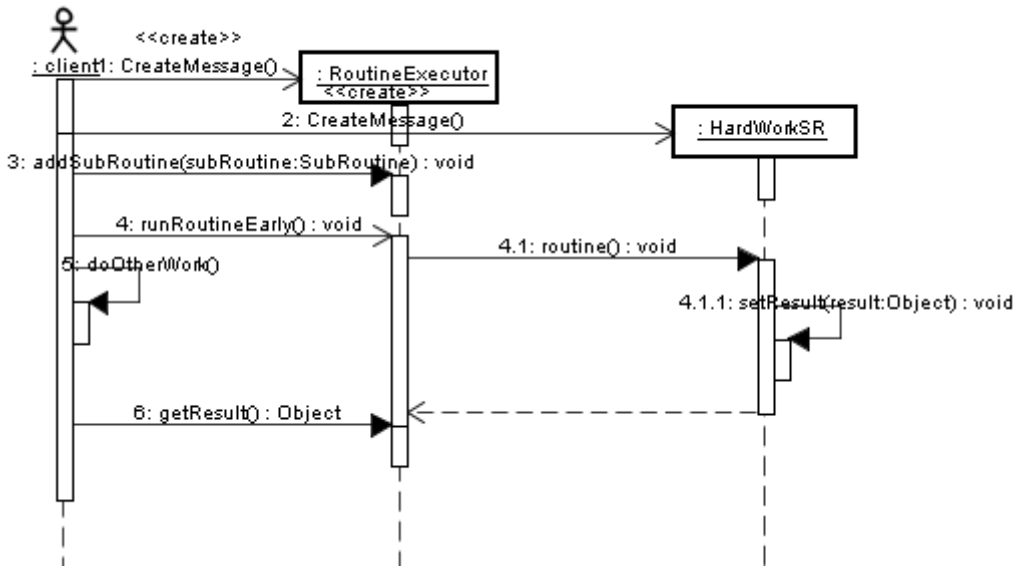


图 5-7 先执行后取结果示例顺序图

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 子程序（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.RoutineExecutor;


public class TestTimeoutClient {
	public static void main(String[] args) {
		RoutineExecutor re = new RoutineExecutor();
		re.addSubRoutine(new HardWorkSR());
		List result = (List) re.runRoutineForTime(11);//最大允许子程序执行 11
秒
		System.out.println(result);
		System.out.println("ok");
	}
}
```

从代码可知，超时设置为 11 秒，HardWorkSR 子程序会正常执行，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);// 最大允许子程序执行 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.RoutineExecutor.runRoutineForTime(RoutineExecutor.
java:104)

      at
example.appsrv.routine.TestTimeoutClient.main(TestTimeoutClient.java:12)
```

线程超时，执行中断。

简易锁

锁在线程编程经常使用到，通常我们会使用 Object 对象的 wait 和 notify 方法来操控。为了简化编程，BJAF 框架提供了一个简易锁 Locker，见下图：

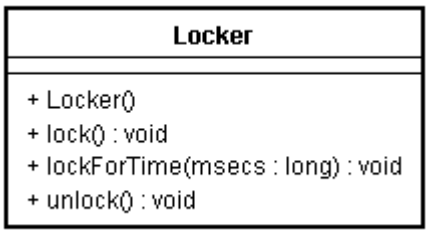


图 5-8 锁类图

利用这个 Locker，我们可以在主流程中加锁阻塞等待，通过另外一个子程序或线程进行解锁，从而来恢复主流程进行执行。示例代码如下：

```
package example.appsrv.routine;
```

```
import com.beetle.framework.appsrv.Locker;
import com.beetle.framework.appsrv.RoutineExecutor;
import com.beetle.framework.appsrv.SubRoutine;

public class TestLockClient {
	private static class SR extends SubRoutine {
		private Locker locker;
		public SR(Locker locker) {
			this.locker = locker;
		}
		protected void routine() throws InterruptedException {
			System.out.println("sr-begin");
			sleep(5000);
			System.out.println("sr-end");
			locker.unlock();// 释放锁
		}
	}

	public static void main(String[] args) {
		Locker locker = new Locker();
		try {
			RoutineExecutor.runRoutineInPool(new SR(locker));
			locker.lock();// 加锁，阻塞主线程
			//locker.lockForTime(2000);// 最多锁住 2000ms
			System.out.println("ok");
		} catch (InterruptedException e) {
			e.printStackTrace();
		} finally {
			locker.unlock();
		}
```

```
        }

}
```

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”我们叫做任务 Task（或者工作 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 对著名开源包 quartz 进行了封装，提供了一个简易定时计划任务模块。其相关类关系图如下：

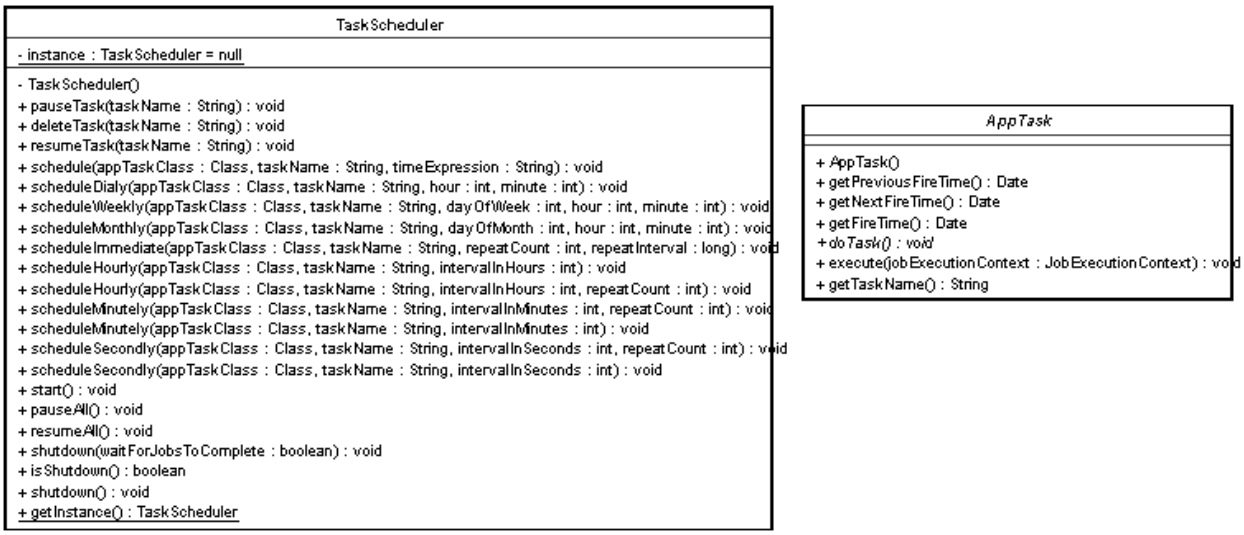


图 5-9 定时计划任务类图

从上图可知，AppTask 就是任务，而 TaskScheduler 就是这个任务计划执行器。示例代码如下：

（1），编写一个任务（扩展 AppTask 抽象类，实现其 doTask 抽象方法即可），如：

```
package example.appsrv.task;

import com.beetle.framework.appsrv.AppTask;

public class DemoTask extends AppTask {

      public void doTask() {

            System.out.println("do task...");

      }

}
```

（2），编写一个客户端，调用 TaskScheduler 按照需求来执行这个任务，代码如下：

```
package example.appsrv.task;


import com.beetle.framework.appsrv.TaskScheduler;


public class TestClient {
	public static void main(String[] args) {
		// 获取计划执行器
		TaskScheduler scheduler = TaskScheduler.getInstance();
		// 定义计划任务（每日 23 时 59 分执行此任务）
		scheduler.scheduleDialy(DemoTask.class, "DemoTask", 23, 59);
		// 启动这个计划
		scheduler.start();
	}
}
```

这样 DemoTask 在每天 23 时 59 分都会被触发执行。

远程通信模块

现在很多应用服务器都是网络通信的服务器，提供一个远程通信模块是 Beetle 应用服务程序框架的一直心愿。虽然目前市场有很多这方面的框架产品，如：Grizzly、Mina 和 Netty 等，但它们定位为大而全，结构复杂，学习周期长。Beetle1.3.6 版本开始，提供一个 remoting 远程通信模块，它完全基于消息且结构和使用都十分简单。其主要的功能特点为：

基于 NIO 的多线程服务器实现，性能优秀

完全基于网络消息对象传递，没有数据格式限制，便于扩展

支持基于用户名和密码的网络服务器验证

客户端支持同步、异步方法调用，消息发送和接收

服务器 Session 管理，向客户端消息推送

服务器端支持 Session 创建和掉失事件

服务器端支持服务器终止和客户端连接中断事件

客户端支持连接中断事件和消息到达事件

远程通信模块相关主要类图如下：

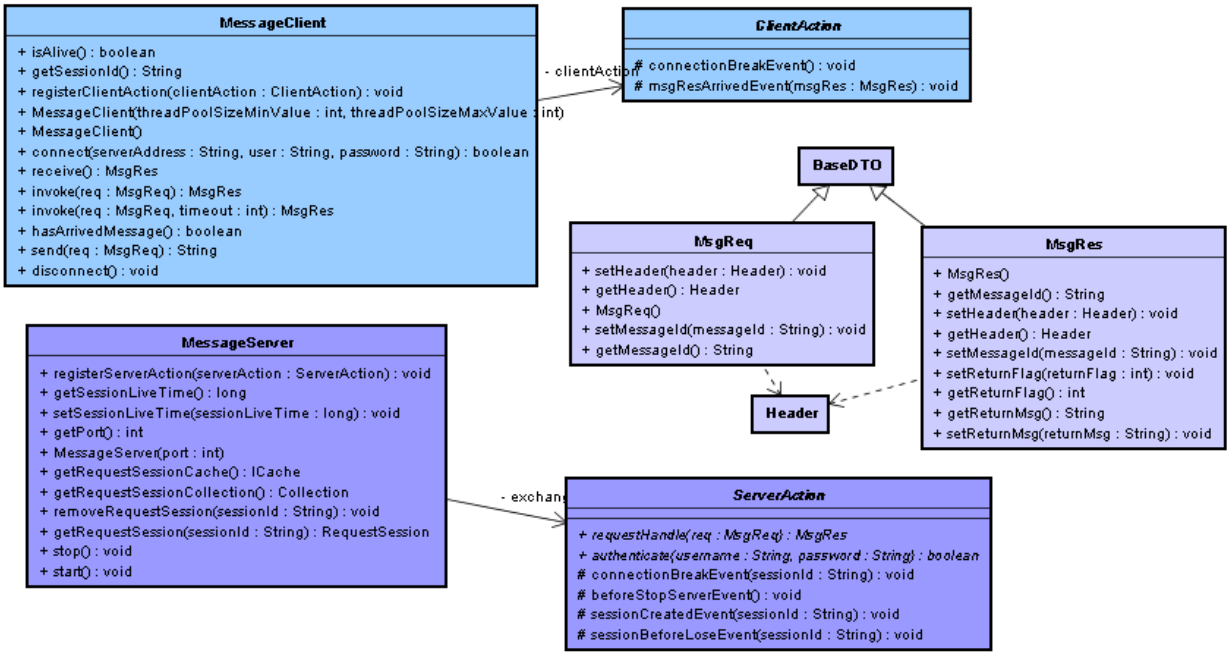


图 5-10Remoting 通信组件类关系图

从上图可知，MessageServer 是服务器实现，ServerAction 是服务器处理相关动作的抽象类，是通信组件服务端编程入口，其包含 2 个抽象方法及 4 个服务器触发事件：

1，requestHandle (req : MsgReq) : MsgRes，负责处理请求，其输入为 MsgReq 消息，返回为 MsgRes 消息，它们共同继承自 BaseDTO 基础类。

2，authenticate(username : String, password : String) : boolean，客户端验证方法。参数为客户端发送上来的用户名和密码，通过此方法开发人员可以根据自己需要对其做验证处理。

3， connectionBreakEvent(sessionId : String) : void，客户端断开在服务端触发的此事件 (连接中断)，需要处理此事件，重载此方法即可。

4， beforeStopServerEvent() : void，在此服务器停止之前触发此事件。

5， sessionCreatedEvent(sessionId : String) : void，Session 建立成功后触发此事件。

6， sessionBeforeLoseEvent(sessionId : String) : void，Session 长时间没有会话，掉线之前（及系统自动回收此会话之前）触发此事件。

MessageClient 是客户端实现，提供了创建连接、关闭连接、消息发送、接收，方法调用等功能。相对于 MessageServer，客户端同样提供了一个 ClientAction 的抽象类，支持客户端扩展动作编程。它目前支持两个事件：

1， connectionBreakEvent() : void，连接中断触发的事件。

2， msgResArrivedEvent(msgRes : MsgRes) : void，服务端响应消息到达事件。注意：如果此 msgRes 被消费掉（清空消息体或设置为 null），那么 receive 方法不会检索到此消息。send 方式才注册事件

可见，Beetle 的 remoting 通信模块相对哪些动不动几百类实现的框架来说，其结构十分简单，但

功能却十分强大。下面实现一个简单的 Echo 服务器例子以便说明：

（1），编写服务端动作及事件处理动作类 EchoServerAction

```
package example.appsrv.remoting;


import com.beetle.framework.appsrv.remoting.MessageCommunicateException;
import com.beetle.framework.appsrv.remoting.MessageServer;
import com.beetle.framework.appsrv.remoting.MsgReq;
import com.beetle.framework.appsrv.remoting.MsgRes;


public class EchoServerAction extends MessageServer.ServerAction {


      public boolean authenticate(String username, String password) {
            if (username.equals("Henry") && password.equals("888888")) {
                  return true;
            }
            return false;
      }


      public MsgRes requestHandle(MsgReq req) throws
MessageCommunicateException {
            String word = req.getValueAsString("word");
            System.out.println("[" + System.currentTimeMillis() + "]:" +
word);
            MsgRes res = new MsgRes();
            res.setValueWithString("echo", "echo:" + word);
            return res;
      }


      protected void beforeStopServerEvent() throws
MessageCommunicateException {
            System.out.println("beforeStopServerEvent");
```

```
	}

	protected void connectionBreakEvent(String sessionId)
			throws MessageCommunicateException {
		System.out.println("connectionBreakEvent[" + sessionId + "]");
	}

	protected void sessionBeforeLoseEvent(String sessionId)
			throws MessageCommunicateException {
		System.out.println("sessionBeforeLoseEvent[" + sessionId + "]");
	}

	protected void sessionCreatedEvent(String sessionId)
			throws MessageCommunicateException {
		System.out.println("");
	}
}
```

（2），编写 EchoServer 服务器：

```
package example.appsrv.remoting;

import com.beetle.framework.appsrv.remoting.MessageServer;

public class EchoServer {

	public static void main(String[] args) {
		MessageServer ms = new MessageServer(8080);
		ms.registerServerAction(new EchoServerAction());
		ms.start();
	}
```

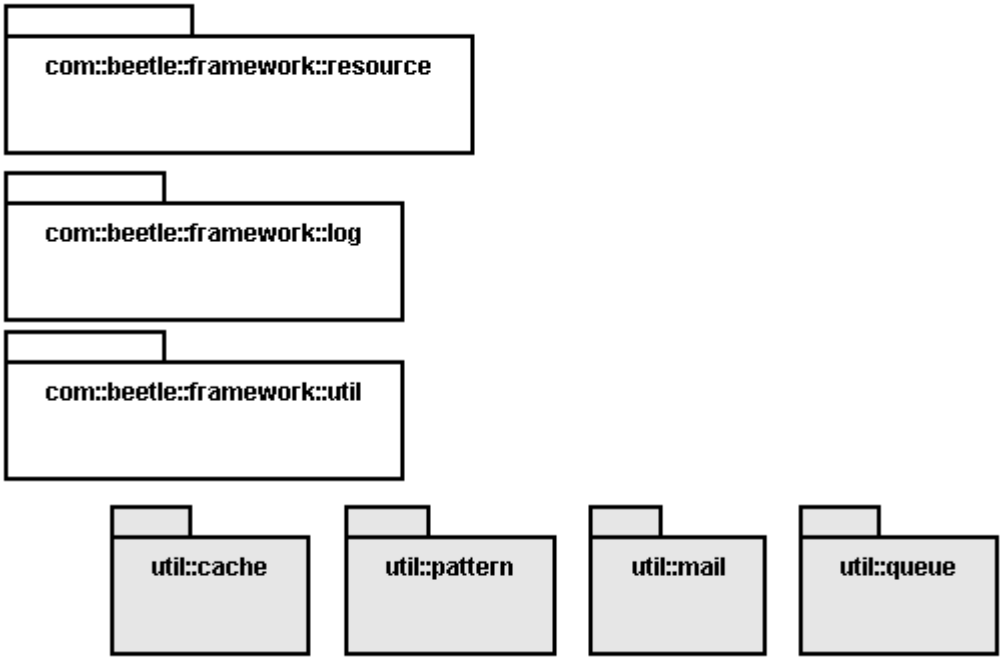
```
}
```

（3），编写 EchoClient 客户端：

```
package example.appsrv.remoting;


import com.beetle.framework.appsrv.remoting.MessageClient;
import com.beetle.framework.appsrv.remoting.MessageCommunicateException;
import com.beetle.framework.appsrv.remoting.MsgReq;
import com.beetle.framework.appsrv.remoting.MsgRes;


public class EchoClient {


      public static void main(String[] args) throws
MessageCommunicateException {
            MessageClient client = new MessageClient();
            boolean f = client.connect("127.0.0.1:8080", "Henry", "888888");
            if (f) {
                  MsgReq req = new MsgReq();
                  req.setValueWithString("word", "Hi,I am Henry.");
                  MsgRes res = client.invoke(req);
                  if (res != null) {
                        String echo = res.getValueAsString("echo");
                        System.out.println(echo);
                  }
                  client.disconnect();
            }
            System.exit(0);
      }
}
```

Core 层

Core 层是 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：



6-1 BJAF 核心层组件

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。

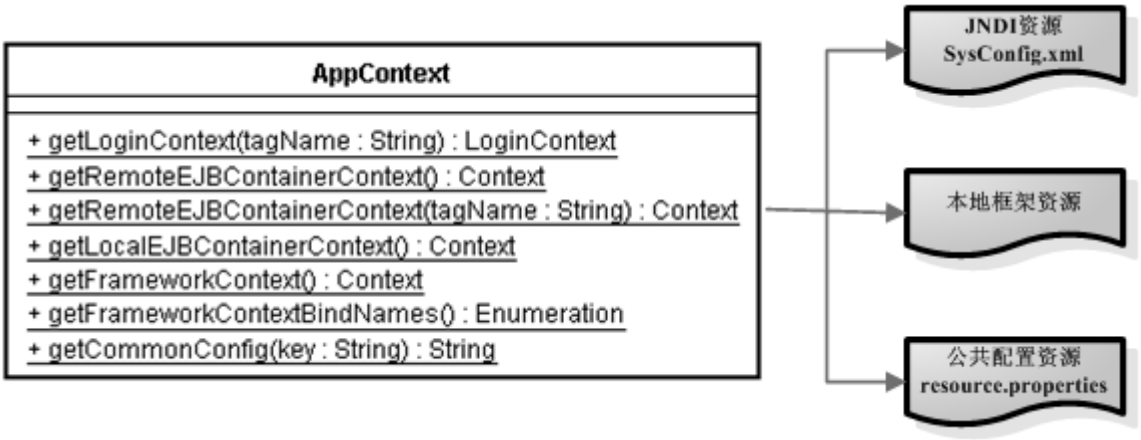


图 6-2 资源统一访问 API

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 SysConfig.xml 文件中，其格式如下：

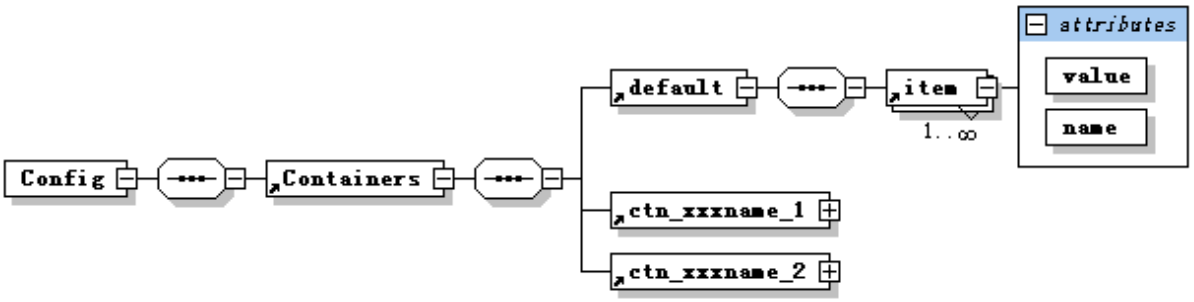


图 6-3 SysConfig.xml 格式

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) {

            }

    }

}
```

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，对外统一为 SysLogger 类作为面板，让日志处理更为简便，其类图如下：

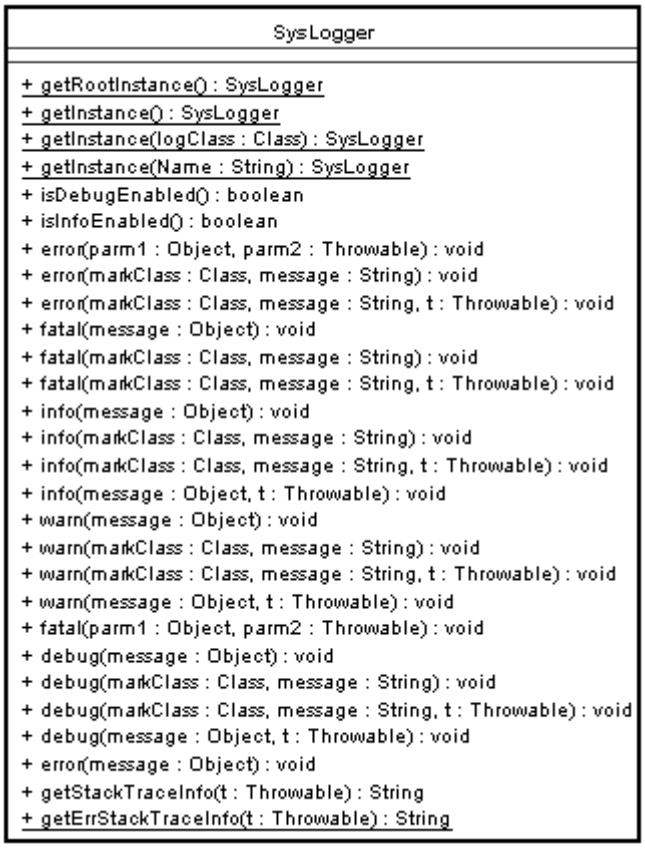


图 6-4 日志处理面板

上图可知，SysLogger 提供了 3 个记录器实例的获取方法，它们的区别在于其前缀 Pattern 有所不同。值得主要的是，getInstance()的日志前缀是“com.beetle.framework.log.SysLogger”也就是此 SysLoogger 本身，类似与 System.out，若对日志定位要求不是很严谨的话，可采取此实例记录器，由于此记录器只创建一个实例，采取它可以节省内存。

使用日志记录器十分简单，代码如下：

```
package test.core;


import com.beetle.framework.log.SysLogger;


public class Test {
	//注册一个日志记录器
	private static SysLogger logger = SysLogger.getInstance();


	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
```

log4j 的详细使用及配置说明，请参考其官方网站相关资料。

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。

常见设计模式组件

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：

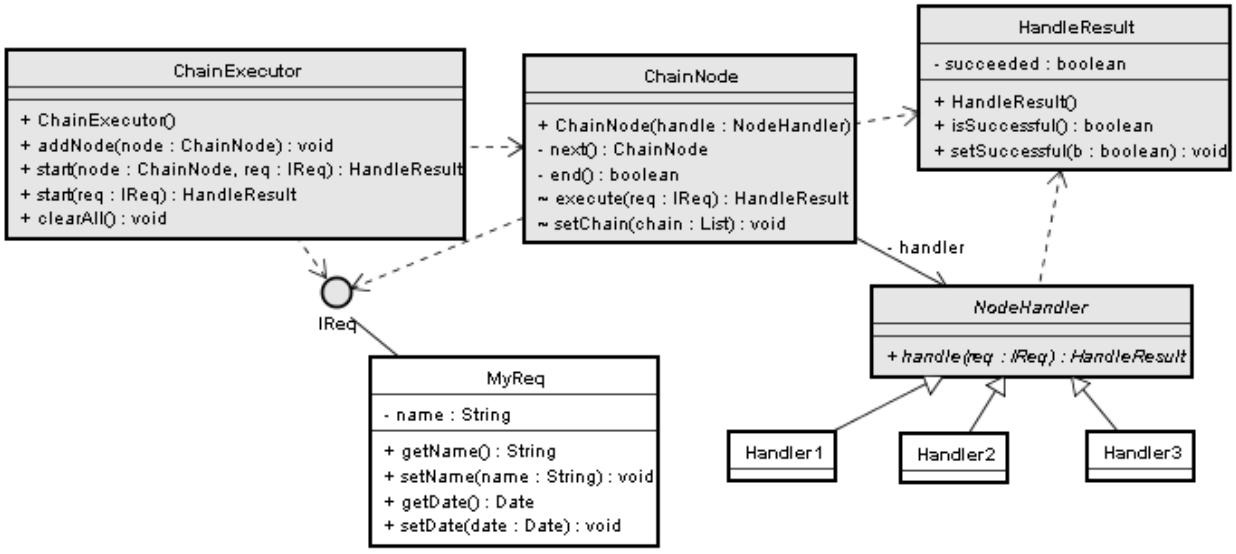


图 6-5 COR 模式类结构图

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;
	}
}
```

2，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) {
		}
	}
}
```

3， 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
```

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

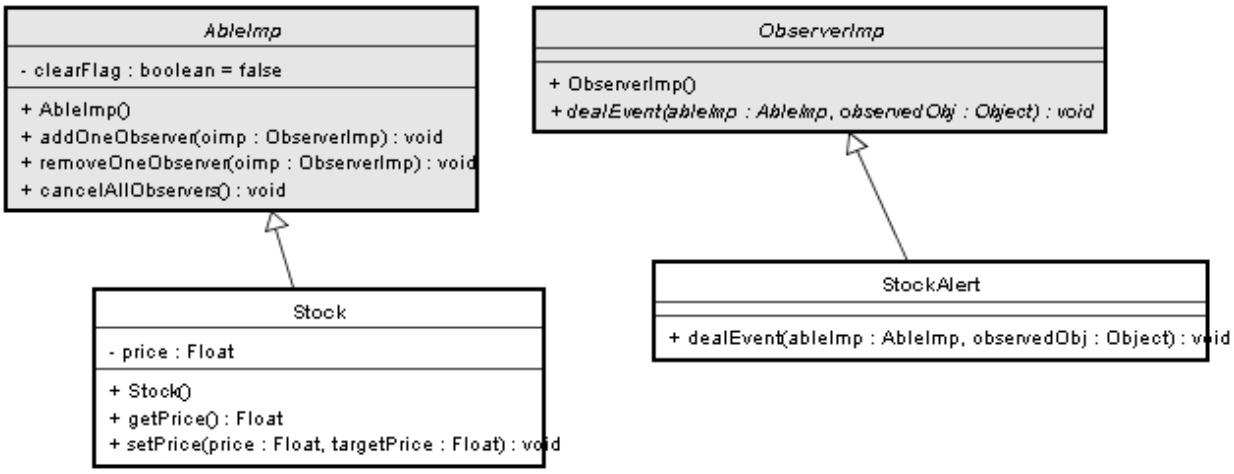


图 6-6 观察者模式类图

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
		}
	}
}
```

2， 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
	}

}
```

3， 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!
---
```

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：

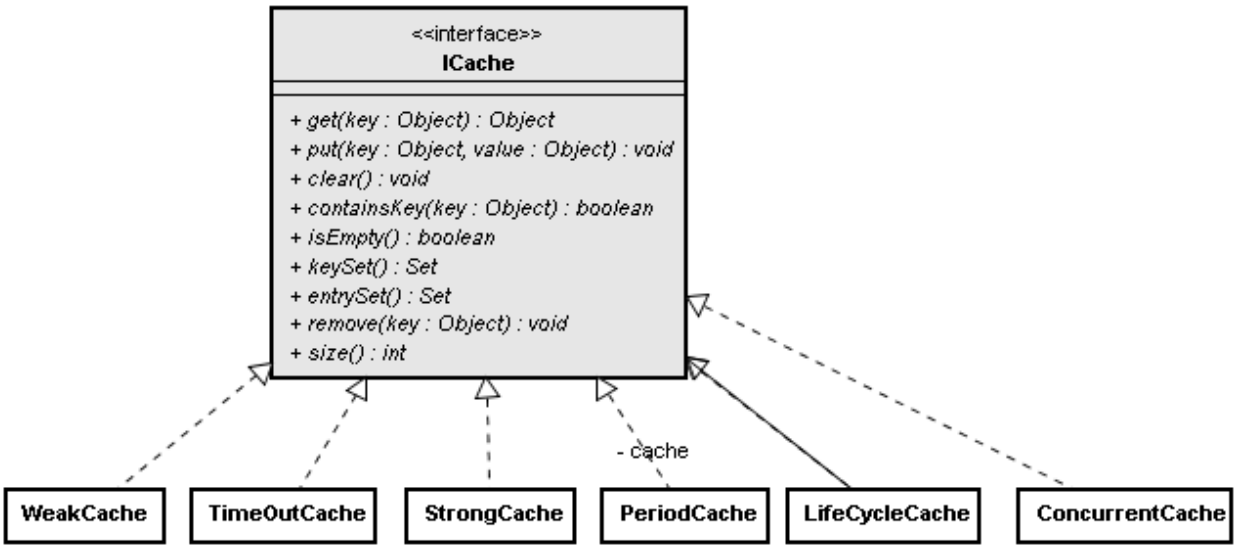


图 6-7 各种缓存器类图

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

队列/堆栈

BJAF 实现了队列/堆栈两个简单的数据结构集合，见下面类图：

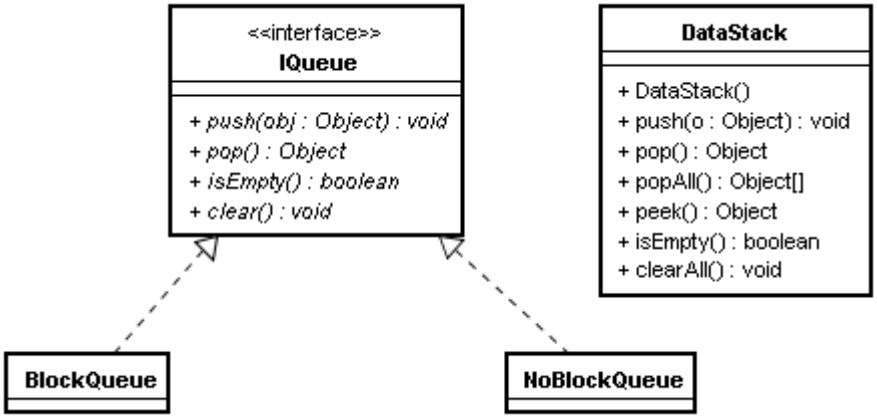
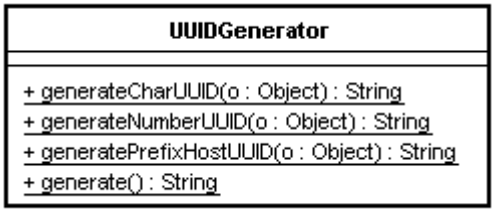
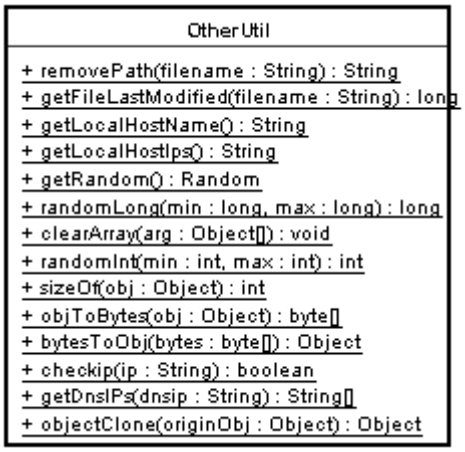


图 6-8 队列、堆栈集合类图

队列、堆栈	应用说明
BlockQueue	阻塞队列，当调用pop()方法从队列弹出一个对象的时候，如果此时队列为空，则此pop方法会阻塞，直到队列有新的对象，才返回。
NoBlockQueue	相对BlockQueue队列，调用pop方法若此时队列对空，不会阻塞，返回为null
DataStack	数据堆栈，为非阻塞的

常用工具类

工具类	应用说明
AesEncrypt	采取Aes加密算法对字符串进行加密/解密 
UUIDGenerator	UUID唯一编号生成器 

ResourceLoader	资源加载器，方便应用访问资源文件
OtherUtil	其它工具类，包含：对象深度克隆、对象大小计算、随机数、IP地址合法性、DNS解析IP等 

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。

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.