深圳四方精创资讯股份有限公司

Java EE开发平台——数据访问层

使用手册

公司标志 copy

|  |  |  |  |
| --- | --- | --- | --- |
| **文档名称** | Java EE开发平台——数据访问层使用手册 | **文档编号** |  |
| **文档类别** | 使用手册 | **版本信息** | V1.0 |
| **内部密级** | 公开 | **外部密级** | 自用 |
| **创 建 人** | 林济松 | **创建日期** | 2016-6-2 |
| **审 批 人** |  | **审批日期** |  |

修改历史

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本号 | 日期 | \*状态 | 修订人 | 摘要 |
| V1.0 | 2016-6-2 | C | 林济松 | 创建文档 |
| V1.0 | 2016-6-8 | A | 林济松 | 添加数据访问层说明 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

\*状态：C – 创建 A – 增加 M – 修改 D – 删除

# 目录

[目录 I](#_Toc456799087)

[第1章 数据访问层 1](#_Toc456799088)

[§1.1 数据访问组件相关配置 1](#_Toc456799089)

[§1.1.1 单数据源 1](#_Toc456799090)

[§1.1.1.1 数据源配置 1](#_Toc456799091)

[§1.1.1.2 事务配置 2](#_Toc456799092)

[§1.1.1.3 mybatis-spring集成配置 2](#_Toc456799093)

[§1.1.2 多数据源 3](#_Toc456799094)

[§1.1.3 Mybatis全局配置 3](#_Toc456799095)

[§1.1.4 平台配置中的数据访问组件参数 4](#_Toc456799096)

[§1.2 IDaoTemplate接口 6](#_Toc456799097)

[§1.2.1 查询API 6](#_Toc456799098)

[§1.2.1.1 查询单笔 6](#_Toc456799099)

[§1.2.1.2 查询列表 6](#_Toc456799100)

[§1.2.1.3 流式查询 7](#_Toc456799101)

[§1.2.2 修改API 10](#_Toc456799102)

[§1.2.2.1 新增 10](#_Toc456799103)

[§1.2.2.2 修改 11](#_Toc456799104)

[§1.2.2.3 删除 11](#_Toc456799105)

[§1.2.3 批量执行API 12](#_Toc456799106)

[§1.2.4 存储过程API 14](#_Toc456799107)

[§1.3 Mybatis动态代理接口用法 16](#_Toc456799108)

[§1.3.1 问题1：确定需要执行的sqlId 17](#_Toc456799109)

[§1.3.2 问题2：确定需要执行的方法名 17](#_Toc456799110)

[§1.3.3 问题3：确定执行SQL时的参数 19](#_Toc456799111)

[§1.4 Beneform4j动态代理接口用法 20](#_Toc456799112)

[§1.4.1 问题1：确定需要执行的sqlId 20](#_Toc456799113)

[§1.4.2 问题2：确定需要执行的方法名 22](#_Toc456799114)

[§1.4.3 问题3：确定执行SQL时的参数 23](#_Toc456799115)

[§1.5 批量执行动态代理接口用法 24](#_Toc456799116)

[§1.5.1 准备工作 24](#_Toc456799117)

[§1.5.1.1 可转换为集合类型的参数 24](#_Toc456799118)

[§1.5.1.2 批量类型 25](#_Toc456799119)

[§1.5.2 批量类型1：单SQL批量 26](#_Toc456799120)

[§1.5.2.1 确定需要执行的sqlId 26](#_Toc456799121)

[§1.5.2.2 确定需要执行的方法名 26](#_Toc456799122)

[§1.5.2.3 确定执行SQL时的参数 26](#_Toc456799123)

[§1.5.3 批量类型2：多SQL批量 29](#_Toc456799124)

[§1.5.3.1 确定需要执行的sqlId组 29](#_Toc456799125)

[§1.5.3.2 确定需要执行的方法名 29](#_Toc456799126)

[§1.5.3.3 确定执行SQL时的参数 29](#_Toc456799127)

[§1.5.4 批量类型3：混合批量 30](#_Toc456799128)

[§1.5.4.1 确定需要执行的sqlId组 30](#_Toc456799129)

[§1.5.4.2 确定需要执行的方法名 32](#_Toc456799130)

[§1.5.4.3 确定执行SQL时的参数 32](#_Toc456799131)

[§1.5.5 回顾与总结 33](#_Toc456799132)

[§1.6 事务用法说明 34](#_Toc456799133)

[§1.6.1 直接使用JDBC时的事务控制 34](#_Toc456799134)

[§1.6.2 Spring声明式事务 35](#_Toc456799135)

[§1.6.2.1 Spring事务管理的局限性约束 36](#_Toc456799136)

[§1.6.2.2 Spring声明式事务的典型用法 37](#_Toc456799137)

[§1.6.3 Beneform4j平台事务 40](#_Toc456799138)

[第2章 WEB控制层 42](#_Toc456799139)

[§2.1 Spring MVC概述 42](#_Toc456799140)

[§2.1.1 动态WEB技术发展历程 42](#_Toc456799141)

[§2.1.2 Servlet API简要说明 44](#_Toc456799142)

[§2.1.3 Spring MVC概述 48](#_Toc456799143)

[§2.1.3.1 DispatcherServlet继承体系 48](#_Toc456799144)

[§2.1.3.2 DispatcherServlet九大组件概述 50](#_Toc456799145)

[§2.1.3.2.1 MultipartResolver：上传解析器 50](#_Toc456799146)

[§2.1.3.2.2 LocaleResolver：国际化解析器 51](#_Toc456799147)

[§2.1.3.2.3 ThemeResolver：主题解析器 52](#_Toc456799148)

[§2.1.3.2.4 HandlerMapping：客户请求与后台处理器的映射 53](#_Toc456799149)

[§2.1.3.2.5 HandlerAdapter：执行后台处理器的适配器 54](#_Toc456799150)

[§2.1.3.2.6 HandlerExceptionResolver：处理器异常的解析器 55](#_Toc456799151)

[§2.1.3.2.7 RequestToViewNameTranslator：默认视图名的解析器 55](#_Toc456799152)

[§2.1.3.2.8 ViewResolver：视图解析器 56](#_Toc456799153)

[§2.1.3.2.9 FlashMapManager：FlashMap管理器 56](#_Toc456799154)

[§2.1.3.3 DispatcherServlet组件初始化策略 56](#_Toc456799155)

[§2.1.3.4 DispatcherServlet请求处理流程 58](#_Toc456799156)

[§2.1.3.5 总结 61](#_Toc456799157)

[§2.2 Beneform4j中Spring MVC组件配置 61](#_Toc456799158)

[§2.3 请求映射HandlerMapping 61](#_Toc456799159)

[§2.3.1 RequestMappingHandlerMapping 62](#_Toc456799160)

[§2.3.2 ViewMappingHandlerMapping 62](#_Toc456799161)

[§2.3.3 ProxyHandlerMapping 63](#_Toc456799162)

[§2.4 处理器调用适配器HandlerAdapter 65](#_Toc456799163)

[§2.4.1 RequestMappingHandlerAdapter 65](#_Toc456799164)

[§2.4.1.1 参数绑定HandlerMethodArgumentResolver 66](#_Toc456799165)

[§2.4.1.2 结果处理HandlerMehtodReturnValueHandler 68](#_Toc456799166)

[§2.4.1.3 Http消息转换器HttpMessageConverter 69](#_Toc456799167)

[§2.4.2 ViewMappingHandlerAdapter 71](#_Toc456799168)

[§2.4.3 ProxyHandlerAdapter 71](#_Toc456799169)

[§2.5 服务端数据校验 71](#_Toc456799170)

[§2.5.1 数据校验示例 71](#_Toc456799171)

[§2.5.1.1 步骤一：环境配置 71](#_Toc456799172)

[§2.5.1.2 步骤二：编写领域模型类 72](#_Toc456799173)

[§2.5.1.3 步骤三：编写国际化提示信息 73](#_Toc456799174)

[§2.5.1.4 步骤四：在控制层添加校验注解 73](#_Toc456799175)

[§2.5.1.5 步骤五：处理校验结果 74](#_Toc456799176)

[§2.5.2 校验注解 75](#_Toc456799177)

[§2.5.3 其它用法 76](#_Toc456799178)

[§2.6 JsonBody注解族 76](#_Toc456799179)

[§2.6.1 目标 76](#_Toc456799180)

[§2.6.2 Jackson配置 77](#_Toc456799181)

[§2.6.3 @JsonBody注解 78](#_Toc456799182)

[§2.6.4 @JsonField注解 79](#_Toc456799183)

[§2.6.5 属性筛选模式 81](#_Toc456799184)

[§2.6.6 属性转换器接口IJsonConverter 82](#_Toc456799185)

[§2.6.7 数据包装器接口IJsonWrapper 82](#_Toc456799186)

[§2.6.8 @JsonBody子注解族 83](#_Toc456799187)

[§2.6.8.1 示例说明 84](#_Toc456799188)

[§2.6.8.2 @JsonBody子注解及其包装器一览 84](#_Toc456799189)

[§2.6.8.3 注意事项 85](#_Toc456799190)

[§2.7 视图解析 85](#_Toc456799191)

[§2.7.1 视图解析器 85](#_Toc456799192)

[§2.7.2 视图 86](#_Toc456799193)

[§2.8 异常处理 87](#_Toc456799194)

[§2.9 国际化 87](#_Toc456799195)

[§2.10 主题切换 87](#_Toc456799196)

[§2.11 上传支持 87](#_Toc456799197)

# 数据访问层

## 数据访问组件相关配置

示例项目的数据访问组件配置主要在spring-orm.xml文件中，可参考该文件。

### 单数据源

#### 数据源配置

* 容器管理
* 在Web容器中配置好数据源，如在tomcat容器中的context.xml文件、websphere里面的连接池等。
* 在spring-orm.xml文件中如下配置（需导入jee命名空间）：

<jee:jndi-lookup id="dataSource" jndi-name="jdbc/beneform4j"/>

* 连接池，可以选用dbcp、druid等开源包，具体配置大同小异，下面使用druid做示例说明。
* 在properties属性文件中配置数据源相关的属性。
* 在spring-commons.xml中配置好属性文件路径。
* 在spring-orm.xml中如下配置数据源：

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*

init-method=*"init"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"${beneform4j.db.driver}"* />

<property name=*"url"* value=*"${beneform4j.db.url}"* />

<property name=*"username"* value=*"${beneform4j.db.username}"* />

<property name=*"password"* value=*"${beneform4j.db.password}"* />

</bean>

其中${beneform4j.db.driver}等是引用属性文件中的配置值。

#### 事务配置

在spring-orm.xml文件中，添加如下配置：

<!-- 开启注解事务 -->

<tx:annotation-driven transaction-manager=*"txManager"*/>

<!-- 事务管理 -->

<bean id=*"txManager"* p:dataSource-ref=*"dataSource"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*/>

#### mybatis-spring集成配置

在spring-orm.xml文件中，添加如下配置：

<bean id=*"DialectDatabaseIdProvider"* class=*"com.forms.beneform4j.core.dao.mybatis.provider.DialectDatabaseIdProvider"*/>

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*

p:sqlSessionFactoryBeanName=*"sqlSessionFactory"* p:basePackage=*"com.forms"*

p:annotationClass=*"org.springframework.stereotype.Repository"*/>

<bean id=*"sqlSessionFactory"* p:dataSource-ref=*"dataSource"* class=*"com.forms.beneform4j.core.dao.mybatis.SqlSessionFactoryBeanForSpring"*

p:configLocation=*"classpath:mybatis/mybatis-config.xml"*

p:databaseIdProvider-ref=*"DialectDatabaseIdProvider"*>

<property name=*"mapperLocations"*>

<array>

<value>classpath\*:sql-mapper/beneform4j/\*.xml</value>

<value>classpath\*:com/forms/beneform4j/\*\*/\*.beneform4j.xml</value>

</array>

</property>

</bean>

如上所示，主要包括三个bean，其中DialectDatabaseIdProvider是数据库ID提供者的实现类，主要用于SqlMapper配置文件中区分是哪种数据库产品，这是Mybatis接口的一个实现类，严格来说，不算是和Spring集成相关的配置，也可以在mybatis全局文件中自行配置；MapperScannerConfigurer则是配置自动扫描，为哪些接口动态生成代理子类，如上例，只为com.forms包下面的含有Repository注解的接口生成代理子类；真正集成配置必须的是SqlSessionFactory，这是Sql会话工厂，里面可以添加Mybatis的全局配置文件、需要扫描的SqlMapper配置文件等等。

### 多数据源

多数据源的配置和单数据源大同小异：

* 数据源配置：和单数据源一样，添加数据源配置，可以使用容器配置，也可以使用数据源连接池，需要注意的一个地方是：由于有多个数据源，为了API的简单，需要设置一个默认的数据源，而这个默认数据源的规则如下：如果数据源中含有beanId为dataSource的，设置为默认数据源，否则设置第一个数据源为默认数据源。
* 事务配置：事务配置需要先分析清楚业务的事务特性，如果需要多个数据源处于同一个事务，那就使用JPA事务，否则就按单数据源的方式配置事务。
* Mybatis-spring集成配置：和单数据源类似，不同之处在于可以为每一个数据源配置各自的SqlMapper扫描模式。

### Mybatis全局配置

见mybatis/mybatis-config.xml文件。

这里强调其中两个设置，其它的可参看Mybatis官方文档中的说明。

<setting name=*"logPrefix"* value=*"mybatis."*/>

<settingname=*"logImpl"*value=*"com.forms.beneform4j.core.dao.mybatis.MybatisLog"*>

* 日志前缀：我们知道，使用LogFactory获取Log实现类时，可以传入一个名称，这里所谓的日志前缀就是在获取Log实现类时的名称前缀，添加前缀后，可以方便统一控制Mybatis日志的级别。
* 日志实现类：这里的日志实现类是指实现Mybatis框架本身提供的日志接口。这里配置的实现类具体功能下面再结合平台配置说明。

### 平台配置中的数据访问组件参数

平台配置可参考配置文档。

平台配置中和数据访问组件相关的参数配置项如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 分类 | 配置项 | 配置项名称 | 类型 | 默认值 |
| 数据访问 | beneform4jTablePrefix | 平台表的前缀 | String | 空格 |
| 数据访问 | sqlIdMapping | sqlId映射 | Map |  |
| 数据访问 | sqlInterceptors | Sql语句拦截器 | List |  |
| 数据访问 | statementParameterResolvers | Sql参数解析器 | List |  |
| 数据访问 | databaseProductNameDialectMapping | 产品名关键字和方言的映射关系 | Map |  |
| 数据访问 | mybatisLogTypeMapping | Mybatis日志输出模式和类别映射 | Map |  |
| 数据访问 | sqlMapperStrategy | sqlId的查找策略 | Object |  |

其中：

* beneform4jTablePrefix：平台表前缀，默认为空格
* sqlIdMapping：sqlId的映射，可以使用该配置将sqlId指向新的真正需要执行的sqlId
* sqlInterceptors：sql语句拦截器，在执行前进行sql拦截，默认配置为com.forms.beneform4j.core.dao.sql.interceptor.impl.PropertiesSqlInterceptor,

com.forms.beneform4j.core.dao.sql.interceptor.impl.SpelSqlInterceptor

利用beneform4jTablePrefix和sqlInterceptors就可以将SqlMapper文件中的{{BF}}解析为真正的表前缀。

* statementParameterResolvers：语句参数解析器，在Mybatis解析SqlMapper配置中的#{property}时的拦截，默认配置为com.forms.beneform4j.core.dao.mybatis.parameter.impl.SpelStatementParameterResolver

通过该配置，可以实现在#{property}中使用SpEL表达式的功能

* databaseProductNameDialectMapping：数据库产品名关键字与数据库方言之间的映射关系，默认配置如下：

|  |  |
| --- | --- |
| 产品名中含关键字（不区分大小写） | 数据库方言实现类 |
| oracle | Oracle |
| mysql | MySQL |
| db2 | Db2 |
| h2 | H2 |
| ADAPTIVE SERVER ENTERPRISE | SybaseASE |
| IQ | SybaseIQ |

* mybatisLogTypeMapping：Mybatis日志输出的正则模式和日志分类的映射关系。比如如下配置：

<property name=*"mybatisLogTypeMapping"*>

<map>

<entry key=*"==> Preparing:"* value=*"mybatis.preparing"*/>

<entry key=*"==> Parameters:"* value=*"mybatis.parameters"*/>

<entry key=*"&lt;== Total:"* value=*"mybatis.total"*/>

</map>

</property>

什么意思呢？就是根据Mybatis输出的日志内容划分类别，如果输出内容包含“==> Preparing:” 就归为“mybatis.preparing”这一类，至于这一类是否需要输出，则是根据log4j里面的配置而定，如下配置：

log4j.logger.mybatis=DEBUG

log4j.logger.mybatis.preparing=OFF

log4j.logger.mybatis.parameters=OFF

log4j.logger.mybatis.total=DEBUG

就表示不输出“mybatis.preparing”类型的日志。由于Mybatis内核写日志的时候这些这些级别已经确定，我们无法更改，因此可以通过该配置来进行二级分类，从而自主控制是否输出。

* sqlMapperStrategy：查找sqlId的策略，Mybatis原生用法是包名、类名、方法名的组合，Beneform4j平台则添加了@SqlRef等辅助注解（详见下一节介绍）。

## IDaoTemplate接口

Mybatis中数据访问最核心的接口是SqlSession，类似地Beneform4j平台的数据访问组件也有一个核心接口：IDaoTemplate。

### 查询API

#### 查询单笔

查询单笔是直接调用SqlSession中对应方法，有两个重载方法：有执行和没有执行参数。

/\*\*

\* 查询单笔数据

\* **@param** sqlId SQL-ID

\* **@return** 单个对象

\*/

**public** <T> T selectOne(String sqlId);

/\*\*

\* 查询单笔数据

\* **@param** sqlId SQL-ID

\* **@param** parameter 参数对象

\* **@return** 单个对象

\*/

**public** <T> T selectOne(String sqlId, Object parameter);

#### 查询列表

查询列表分为简单列表和分页查询列表，共有四个重载方法：

* 查询列表：执行sqlId，执行参数
* 查询列表：执行sqlId，执行参数
* 分页查询列表：执行sqlId，分页对象
* 分页查询列表：执行sqlId，分页对象，执行参数

/\*\*

\* 查询列表数据

\* **@param** sqlId SQL-ID

\* **@return** 对象列表

\*/

**public** <E> List<E> selectList(String sqlId);

/\*\*

\* 查询列表数据

\* **@param** sqlId SQL-ID

\* **@param** parameter 参数对象

\* **@return** 对象列表

\*/

**public** <E> List<E> selectList(String sqlId, Object parameter);

/\*\*

\* 查询分页列表数据

\* **@param** sqlId SQL-ID

\* **@param** page 分页对象

\* **@return** 指定页的对象列表

\*/

**public** <E> List<E> selectList(String sqlId, IPage page);

/\*\*

\* 查询分页列表数据

\* **@param** sqlId SQL-ID

\* **@param** parameter 参数对象

\* **@param** page 分页对象

\* **@return** 指定页的对象列表

\*/

**public** <E> List<E> selectList(String sqlId, Object parameter, IPage page);

#### 流式查询

流式查询可以传入三个参数：执行sqlId、执行参数、每次读取的记录条数，其中执行sqlId是必须的参数，因此，流式查询也有四个重载方法，返回值都是流式操作接口：

* 流式查询：执行sqlId
* 流式查询：执行sqlId，执行参数
* 流式查询：执行sqlId，每次读取的记录数
* 流式查询：执行sqlId，执行参数，每次读取的记录数

/\*\*

\* 流式查询

\* **@param** sqlId SQL-ID

\* **@return** 流式操作接口

\*/

**public** <E>IListStreamReader<E> selectListStream(String sqlId);

/\*\*

\* 流式查询

\* **@param** sqlId SQL-ID

\* **@param** parameter 参数对象

\* **@return** 流式操作接口

\*/

**public** <E>IListStreamReader<E> selectListStream(String sqlId, Object parameter);

/\*\*

\* 流式查询

\* **@param** sqlId SQL-ID

\* **@param** fetchSize 每次读取的记录条数(0, 5000]

\* **@return** 流式操作接口

\*/

**public** <E>IListStreamReader<E> selectListStream(String sqlId, **int** fetchSize);

/\*\*

\* 流式查询

\* **@param** sqlId SQL-ID

\* **@param** parameter 参数对象

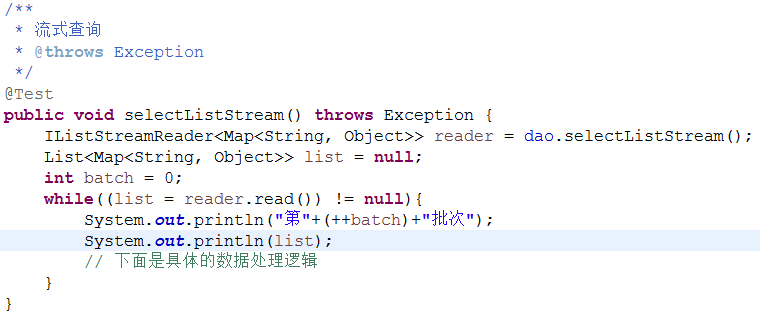
\* **@param** fetchSize 每次读取的记录条数(0, 5000]

\* **@return** 流式操作接口

\*/

**public** <E>IListStreamReader<E> selectListStream(String sqlId, Object parameter, **int** fetchSize);

关于流式查询，说明如下：

* 流式查询在Mybatis中并没有对应方法，是Beneform4j平台扩展的
* 流式查询的返回值都是IListStreamReader接口，这个接口定义如下：
* 流式查询返回接口IListStreamReader时并没有真正执行数据库查询，而只是把执行数据库查询需要的信息（sqlId、执行参数、以及根据fetchSize內建的分页对象等）保存起来，等到调用者调用返回接口中的read()方法时，才真正执行数据库分页查询，每调用一次read()，当前分页数加1，如果数据读取完成，返回null，否则返回数据列表，此外，还可以使用reset()方法重置当前页数，从第1页开始读取。
* 流式查询中有一个参数fetchSize，表示每次读取的记录条数（分页大小），这个参数的取值范围是(0, 5000]，传入负数和0、没有传入参数，都使用默认值1000。
* 流式查询适用于数据量比较大，需要分批次处理的情况。可以按如下方式使用流式查询：

### 修改API

严格来说，修改API包括新增、修改、删除，也应该包括批量执行，但由于批量执行的复杂性，将批量执行单独说明。

新增、修改、删除，比较简单，直接调用SqlSession中的同名方法，各自包括有执行参数和无执行参数的两个重载方法。

#### 新增

/\*\*

\* 新增

\* **@param** sqlId SQL-ID

\* **@return** 影响的记录条数

\*/

**public** **int** insert(String sqlId);

/\*\*

\* 新增

\* **@param** sqlId SQL-ID

\* **@param** parameter 参数对象

\* **@return** 影响的记录条数

\*/

**public** **int** insert(String sqlId, Object parameter);

#### 修改

/\*\*

\* 修改

\* **@param** sqlId SQL-ID

\* **@return** 影响的记录条数

\*/

**public** **int** update(String sqlId);

/\*\*

\* 修改

\* **@param** sqlId SQL-ID

\* **@param** parameter 参数对象

\* **@return** 影响的记录条数

\*/

**public** **int** update(String sqlId, Object parameter);

#### 删除

/\*\*

\* 删除

\* **@param** sqlId SQL-ID

\* **@return** 影响的记录条数

\*/

**public** **int** delete(String sqlId);

/\*\*

\* 删除

\* **@param** sqlId SQL-ID

\* **@param** parameter 参数对象

\* **@return** 影响的记录条数

\*/

**public** **int** delete(String sqlId, Object parameter);

### 批量执行API

Mybatis原生API中，没有提供直接的批量执行API，而是通过提供ExecutorType.BATCH方式来对批量执行进行支持，使用起来非常不方便，因此Beneform4j平台在面向开发人员的接口中添加了批量执行的API。

/\*\*

\* 执行批量：一个SQL执行多次

\* **@param** sqlId SQL-ID

\* **@param** parameters 参数对象数组

\* **@return** 批量执行的影响记录数组

\*/

**public** **int**[] executeBatch(String sqlId, List<?> parameters);

/\*\*

\* 执行批量：一次执行多个SQL

\* **@param** sqlIds 要执行的一组SQL-ID

\* **@return** 批量执行的影响记录数组

\*/

**public** **int**[] executeBatch(List<String> sqlIds);

/\*\*

\* 执行批量：一次执行多个SQL

\* **@param** sqlIds 要执行的一组SQL-ID

\* **@param** parameters 参数对象数组

\* **@return** 批量执行的影响记录数组

\*/

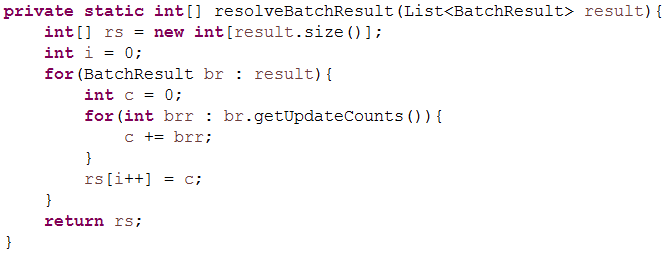
**public** **int**[] executeBatch(List<String> sqlIds, List<?> parameters);

针对批量执行的返回值说明一下：Mybatis原生的批量执行返回结果是List<BatchResult> 类型，首先是一个List，然后List中元素类型是BatchResult，而BatchResult有一个updateCounts属性，是整型数组，表示影响的记录条数。他们的关系如下所示：



也就是说会返回一组数组，表示影响的记录条数。可能有同事对此会感到困惑，为什么是一组数组，而不是一个数组呢？其实是这样的，批量执行并不止是可以执行一个SQL，而可以执行多个SQL，而每一个SQL都可以是JDBC内置的批量，因此需要返回一组数组。比如一次批量执行两条sql：（1）delete from table1 where keyId = ? （2）delete from table2 where keyId = ?，而这两条sql都可以根据参数执行n次，从而也就需要返回每条sql语句对应的整型数组构成的数组了。

但是在Beneform4j平台中对此做了简化，没有直接返回int[][]的整型数组，而是返回int[]，具体简化逻辑是：将相同sql的批量执行影响的记录条数数组累加成一个整数，然后再将所有不同sql的整数组合在一个构成int[]并返回给调用者。看下面的处理代码：



### 存储过程API

Mybatis原生API也没有直接提供存储过程的调用，在Beneform4j中添加了如下两个重载方法：

/\*\*

\* 调用存储过程

\* **@param** sqlId SQL-ID

\* **@return** 存储过程返回结果接口

\*/

**public** ICallResult call(String sqlId);

/\*\*

\* 调用存储过程

\* **@param** sqlId SQL-ID

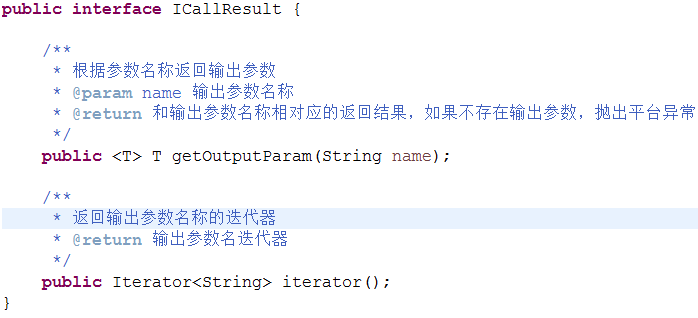
\* **@param** parameter 参数对象

\* **@return** 存储过程返回结果接口

\*/

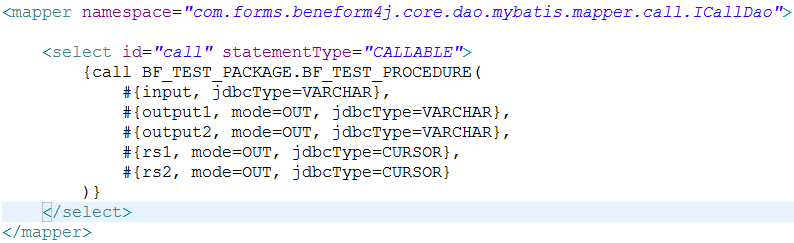
**public** ICallResult call(String sqlId, Object parameter);

参数很简单，sqlId和执行参数，其中执行参数是可选的。需要说明的是，这里引入了一个ICallResult接口，表示调用存储过程的返回结果。这个接口定义如下：

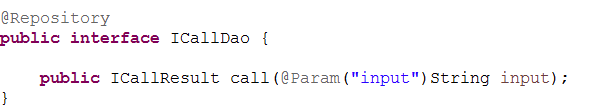


具体用法如下所示：

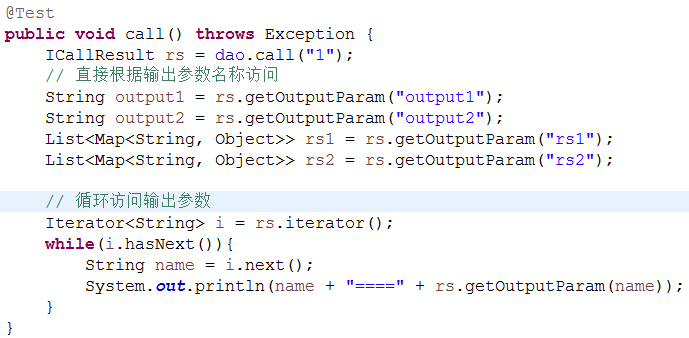
（1）SqlMapper配置



（2）Dao接口



（3）测试类



## Mybatis动态代理接口用法

目前同事习惯于动态代理的方式使用Mybatis，而Beneform4j平台的数据访问层是基于Mybatis进行封装扩展而来的，因此有必要先介绍一下Mybatis的动态代理接口原生用法。不过先声明：这里介绍的只是Mybatis的其中一种用法，如需要更深入了解Mybatis，还是要参考官方文档，或者研读源码。

使用Mybatis的方式有很多种，从是否集成上分，可以单独使用，也可以和Spring集成使用；从使用方式上分，可以编写静态工具类，在静态工具中调用SqlSession，也可以直接注入SqlSession/ SqlSessionTemplate，还可以编写Dao接口，让mybatis自动生成代理子类（对于只编写接口，不写实现类就可以运行有疑问的同事，可以先了解一下JDK中的动态代理技术，这里就不展开了）；在sql脚本的编写上，可以使用xml——也就是通常所说的SqlMapper，可以使用注解，还可以使用Mybatis提供的API生成等等。

根据新平台运行环境和之前同事的开发习惯，我们的选择组合是（姑且称之为动态代理接口方式或代理接口方式）：

集成Spring + Dao接口/自动动态代理子类 + SqlMapper配置

代理接口方式，姑且不管是怎么样代理的，但能够运行起来，肯定是有一段真正的可以运行的代码——实际上就是调用SqlSession接口中的方法。因此需要解决的就是下面三个基本问题：

### 问题1：确定需要执行的sqlId

从SqlSession接口的方法签名可以知道，所有的数据访问方法，都必须有一个参数“statement”，也就是通常所说的sqlId。那么，sqlId是怎么确定的？SqlMapper中那么多配置，Mybatis怎么知道调用哪一个？

Mybatis的解决方法很简单，就是将接口类加上方法作为sqlId，可以用下面一个公式来描述：



### 问题2：确定需要执行的方法名

不管是用哪种方式，归根结底还是调用SqlSession接口中的方法，那么问题来了，SqlSession接口中那么多不同方法，怎么知道调用哪一个呢？比如Dao接口中的一个查询方法，是调用SqlSession的selectOne？还是selectList？还是有回调的select？还是分页查询呢？

确定需要执行的方法，可以分为两步，第一步确定方法名称，第二步确定方法参数，如果有同名的重载方法，Java的反射机制会自动选择最匹配的方法。

第2个问题就是要确定需要执行的方法名称，怎么确定呢？

首先想到的，可以根据sqlId读取到SqlMapper中的配置（不用担心每次都去解析xml，Mybatis会有缓存的），然后就可以知道是select|insert|update|delete中的哪一种，因此，可以先确定sql语句的类型。对于insert、update、delete，因为方法名只有一个，所以也就确定了调用方法名，但是对于select，因为SqlSession中的方法还是很多，还需要进一步区分。

进而想到的是，可以根据dao接口方法的签名来区分，签名包括返回值和形参，比如返回值是void，并且形参中包含处理查询结果的回调接口ResultHandler，那么，就可以大致确定需要调用void select(String statement, ResultHandler handler)这族重载方法中的一种，进一步可以分析形参中是否包含分页参数RowBounds、是否包含其它执行参数从而选择重载方法中的其中一种；又比如可以根据返回值是否为集合（Collection）类型来区分是查询单笔（selectOne）还是查询列表（selectList）。

如果上面两种方法还确定不了，Mybatis还有第三种方式：在dao接口方法上添加特定的注解，比如使用@Flush表示批量执行，使用@MapKey表示selectMap等。

简单总结一下，确定调用方法名的依据：

* SqlMapper中的元素名称，如select | insert | update | delete
* 接口方法签名中特定类型或特定类型的子类型，如分页参数类型RowBounds、结果回调处理器类型ResultHandler
* 接口方法的返回值，如是否集合Collection类型或数组类型
* 接口方法的注解，如@Flush、@MapKey等
* 调用接口时的运行时参数

### 问题3：确定执行SQL时的参数

确定了方法名和sqlId，还需要确定执行SQL时的参数，从SqlSession接口可以知道，所有的执行参数都将包装成为一个对象，那么，怎么包装呢？

首先有个原则：特殊类型的参数不参与执行对象的包装。也就是说RowBounds类型和ResultHandler类型的参数不参与执行对象包装。

然后再看Mybatis的具体包装规则：

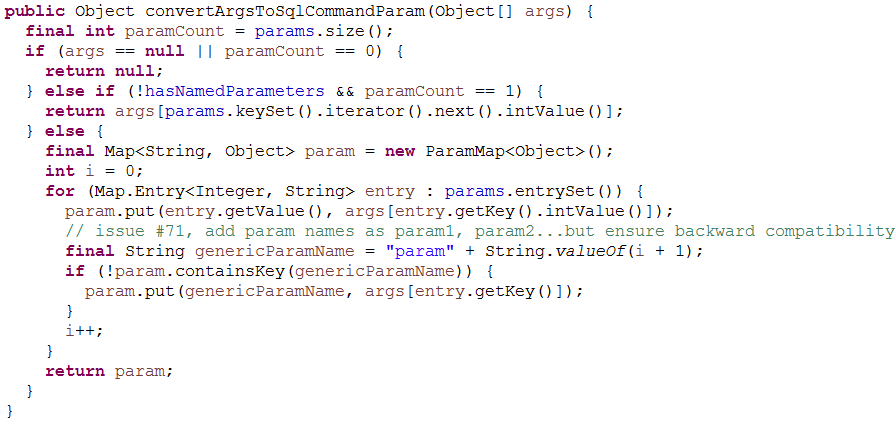
* 如果实参为null或者除了特殊类型参数外没有形参，那就直接返回null

如果没有标注了@Param注解的形参，并且形参的个数只有1个（不计算特殊类型参数），那么返回实参中的这个参数对象，不做任何包装，但是为了使用清晰明确，建议添加@Param注解。

* 其它情形，先创建一个Map结构，然后
* 将含@Param注解的参数以该注解的值为键，对应索引处的实际参数为值存入Map结构
* 将其它没有@Param注解的参数，就使用"param"加上索引序号为键，对应实际参数为值存入Map结构

最后返回这个Map结构。

可能看这段描述还不是很好理解，将这段源码放在这里：



方法名确定了，方法参数确定了，使用反射机制直接调用就可以执行具体的业务逻辑了，相当于Dao接口背后的实现已经有了，至于怎么样把这个实现和接口关联起来，那就是Java中的语法问题了。

## Beneform4j动态代理接口用法

Mybatis的动态代理接口背后起作用的是SqlSession接口，而Beneform4j动态代理接口背后起作用的则是IDaoTemplate接口了，虽然接口不同，但是需要解决的问题却是相同的，也需要确定执行的sqlId/sqlId组、执行的方法名和执行的参数。

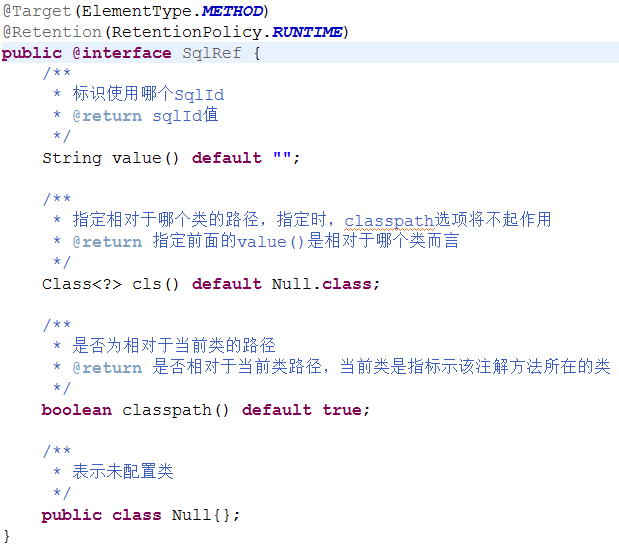
由于批量执行的复杂性，将批量执行单独作为一小节说明，这一节中主要说明非批量执行的动态代理接口用法。

### 问题1：确定需要执行的sqlId

Mybatis原生的确定sqlId的方法虽然简单，但是也有几个缺陷，比如以下几种情形都无法处理：

* 两个重载的方法(方法名相同，参数不同)，需要执行不同的sqlId
* 两个不同名的方法，需要引用相同的sqlId配置(参数不同，比如查询列表和查找元素，就常常可以共用一个<select>配置)
* 当前Dao中的方法需要引用命名空间和全限定类名不一致的sqlId

为此，引入@SqlRef注解来处理上述问题：



这个注解可以添加到Dao接口的方法上面，用来重定向需要执行的sqlId，共有三个属性方法和一个内部类：

* value:表示sqlId值，如果配置为空，则使用注解所在方法的方法名
* cls:表示value是相对于哪个类，SqlRef.Null表示没有配置，如果不是SqlRef.Null，解析sqlId的时候就在前面添加cls.getName()+”.”。
* classpath:和cls作用类似，表示value是否相对于注解所在的类，默认为true，如果为false，则忽略注解所在的类，如果同时配置cls和classpath，以cls为准。
* Null：内部类，只用于表示cls是否配置。

有了@SqlRef，我们就可以自由定义单个sqlId了，很轻送的就解决了前面的三个问题，但是对于需要一次性执行多个sqlId的批量，只使用@SqlRef还是无能为力，需要借助于其它方法，具体方法在批量执行相关章节详细说明。

### 问题2：确定需要执行的方法名

和SqlSession比较，IDaoTemplate接口添加了分页查询（物理分页）、流式查询、存储过程调用、批量执行等方法，因此需要在Mybatis原生方法之前先设别出新添加的方法。

具体规则如下：

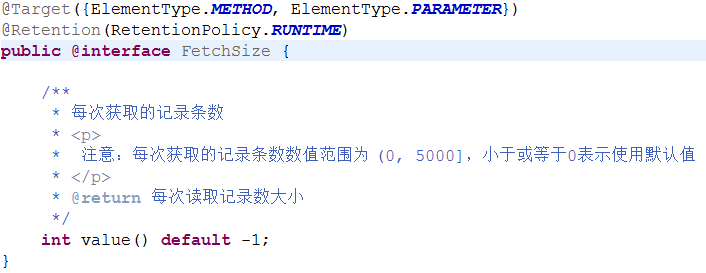
* 如果返回值为整型数组int[]，作为批量执行
* 如果返回值为ICallResult类型，作为存储过程调用
* 如果返回值为IListStreamReader，作为流式查询
* 如果方法参数中包含IPage类型参数，作为分页查询（物理分页）
* 其它情形按Mybatis原生方法确定需要执行的方法名

### 问题3：确定执行SQL时的参数

剔除批量执行，其它接口方法所需的执行参数和Mybatis原生方式大同小异，基本相同。

不同的主要有两点：

* 分页参数IPage：和Mybatis中的RowBounds类型等同处理，作为特殊参数，不参与执行参数的包装
* 流式查询的每次读取记录条数：因为这个参数是一个整型，和一般的sql执行参数无法区分开来，因此，引入一个新的注解@FetchSize：



这个注解有两种使用方法：

* 添加至整型形参前：将整型参数的实参作为fetchSize值，忽略value()属性方法值，具有局部性，每次调用方法，这个值随着实参变化而变化
* 添加至非整型形参前或者添加在接口方法上面：将注解的value()属性方法值作为fetchSize值，具有全局性，每次调用方法，这个值都相同

## 批量执行动态代理接口用法

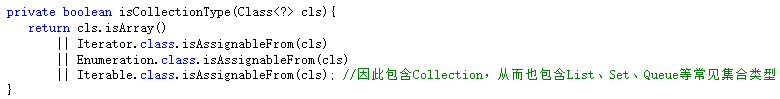
由于批量执行的复杂性，单独使用一小节来说明其用法。

### 准备工作

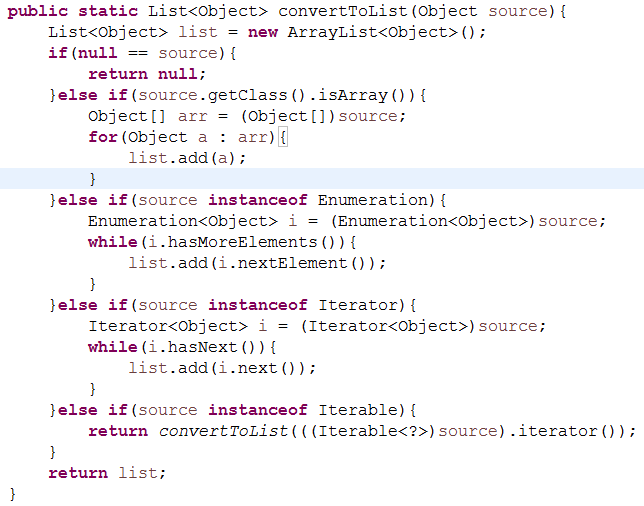
为了更容易描述批量执行的使用，先做一点准备工作：

#### 可转换为集合类型的参数

引进一个概念：可转换为集合类型的参数，什么意思呢？其实很简单，就是说这个参数可以转换成集合类型，比如数组类型的参数、迭代器类型的、本身就是集合类型的等等。看下面的代码可能更清晰：



至于具体怎么转换为集合类型，以转换为List为例，大致如下（剔除了异常处理、简单类型数组的处理等）：



#### 批量类型

根据批量执行的sqlId个数、不同参数等将批量类型分为如下三大类四小类：

* 批量类型1（单sql）：一个sqlId，不同参数，执行多次
* 批量类型2（多sql）：一组sqlId，相同参数，执行多次，这里又分为两种：
* 批量类型2.1（多sql单参数）：一组sqlId，对应相同的参数，每次执行不同sqlId，传入数据库的真正的执行参数完全相同
* 批量类型2.2（多sql多参数）：一组sqlId，对应一个可转换为集合类型的参数，并且sqlId的个数和集合大小相同，一一对应的关系，每次执行不同sqlId，传入数据库的真正的执行参数也不相同
* 批量类型3（混合批量类型）：一组sqlId，有的sqlId本身不是批量类型，有的sqlId本身又是一个批量类型（这个子批量类型就可以限制为单sql批量类型：即一个sqlId，不同参数多次执行，可以想想为什么？），他们的参数也不尽相同

下面就按照批量类型的分类来逐一描述怎么处理动态代理接口中的三个基本问题。

### 批量类型1：单SQL批量

#### 确定需要执行的sqlId

单SQL批量，因为是一个sqlId，所以很简单，和其它方法一样，可以使用Mybatis原生用法，也可以使用@SqlRef注解来确定执行的sqlId。

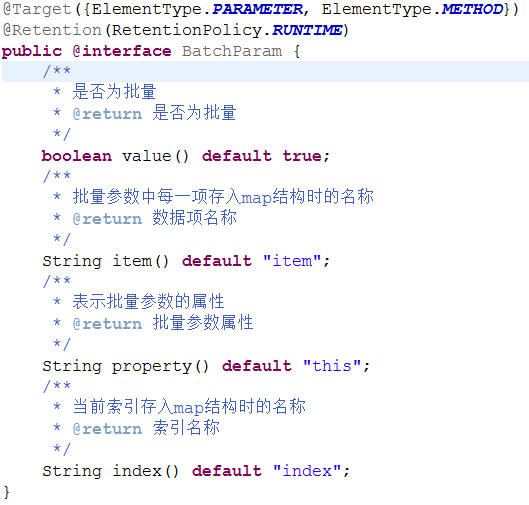
#### 确定需要执行的方法名

IDaoTemplate接口中批量执行只有一个方法名，三个不同参数的重载方法，所以执行的方法名自然也就不用再确定了，直接就等于executeBatch。

#### 确定执行SQL时的参数

一个sqlId执行多次，当然需要不同的参数，也就从本质上要求有一个可转换为集合类型的参数，然后将这个参数转换为真正的集合类型，从而可以循环迭代这个集合类型，每次使用其中的一个参数去执行。

另一方面，如果有多个可转换为集合类型的参数怎么办？使用哪一个呢？为了防止歧义，引入@BatchParam注解来明确指定可转换为集合类型的参数：



从@BatchParam定义可以看到，@BatchParam可以加到方法上，也可以加到方法参数前。但是对于批量类型1——单SQL批量类型，@BatchParam注解只有添加到方法参数前才生效，并且value属性方法不生效，这两个特性（加在方法参数，value()属性方法）是给混合批量类型使用的。

现在再来看看怎么组装批量类型1——单SQL批量类型的执行参数：

1、将所有执行参数按照Mybatis原生用法的方式组装成一个参数对象，记为P1（可能是Map，也可能是单个参数）

2、找出可转换为集合类型的参数，并实际转换为集合类型的参数，记为C2

具体算法：找到第一个标有@BatchParam参数的入参，如果property()属性方法等于“this”，就将这个入参作为集合类型的参数，否则的话，就解析这个入参的对应属性值作为集合类型的参数（属性可以是任意ognl表达式），如果入参或者从入参解析出的参数不是可转换为集合类型的参数，就抛出异常

3、循环迭代C2，每次循环创建一个Map对象PM，然后如下处理：

* 如果P1是一个Map结构，直接将Map结构合并到PM对象中，否则将整个P1参数以"param1"为key存入PM
* 将C2中当前迭代的对象以@BatchParam.item()为key存入PM
* 将当前循环索引以@BatchParam.index()为key存入PM

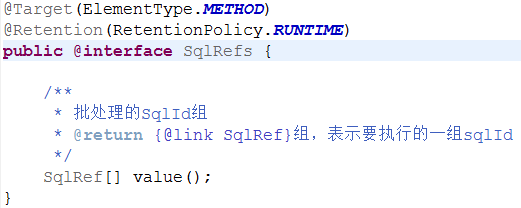
最终，返回Map对象PM组成的集合作为执行SQL时的参数。至此， sqlId找到了，集合类型的参数也准备好了，直接调用IDaoTemplate中public int[] executeBatch(String sqlId, List<?> parameters)就可以了。

注：批量类型1——单SQL批量的关键在于组装批量执行的参数，如果对该过程仍然不清楚的同事，可以多阅读几遍这节中的说明，然后实际跟踪调试几个测试案例。批量类型1也是其它批量类型的基础，请务必彻底弄明白来龙去脉。

### 批量类型2：多SQL批量

#### 确定需要执行的sqlId组

多SQL批量，需要多个sqlId，单纯靠原生用法和@SqlRef注解已经无能为力，为此，引入@SqlRefs注解（比@SqlRef多一个s，表示复数）：



因为每一个@SqlRef可以确定一个sqlId，因此@SqlRefs就可以确定一组sqlId。

#### 确定需要执行的方法名

同单SQL批量，直接等于executeBatch。

#### 确定执行SQL时的参数

批量类型2——多SQL批量可以按是否有含@BatchParam注解的方法形参分为两种情况：

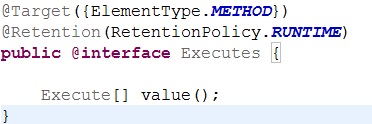
* 批量类型2.1——多SQL单参数：Dao接口方法的方法入参中全部不包含@BatchParam注解，这种情况下，直接按Mybatis原生用法组装执行参数对象P，然后循环sqlId，每次传入P执行，每次数据库接收到的参数都相同
* 批量类型2.2——多SQL多参数：Dao接口方法的方法入参中至少有一个包含@BatchParam注解，这种情况下，取第一个含@BatchParam的方法入参，转换为集合类型的参数（如果不能转换，抛出异常），然后检查sqlId数组的长度和集合参数的大小是否一致（如果不一致，抛出异常），最后再循环sqlId，每次传入集合参数相应位置处的参数执行，每次数据库接收到的参数都是不同对象

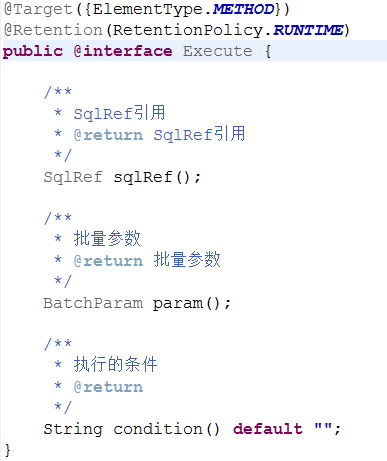
### 批量类型3：混合批量

首先明确一下，批量类型3——混合批量并不仅仅是指批量类型1和批量类型2的混合，还可以混合非批量类型的执行语句。

#### 确定需要执行的sqlId组

和批量类型2一样，需要确定一组sqlId，使用原生方法或@SqlRef注解都无法解决问题，那使用@SqlRefs是否可以呢？如果只是确定一组sqlId，就像批量执行2中那样是没有问题的，但问题是，批量类型3不但需要确定一组sqlId，而且还需要确定和每一个sqlId对应的参数，需要知道其中每一个sqlId本身是不是批量，因此不能简单实用@SqlRefs，为此，引入了@Execute和@Executes注解：

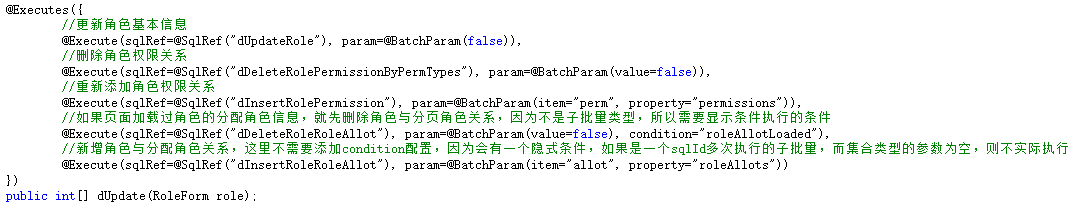




关于@Execute和@Executes说明如下：

* @Execute表示一次执行所需的信息，包括：
* sqlId：由sqlRef()确定
* 执行参数：由param()确定
* 执行条件：由condition()确定，如果为空，视作true，否则视作ognl/spel表达式求值（根对象为执行参数）
* 是否批量：由param()中的value()确定，默认为true
* @Executes表示一组执行，需要说明的是，一次执行并不是在物理上和数据库交互一次，而只是一个逻辑上的划分，实际上，整个批量类型3只会和数据库交互一次
* @Execute不能单独使用，必须嵌套在@Executes中

可以参考下面的例子熟悉一下@Execute和@Executes的使用：



#### 确定需要执行的方法名

同单SQL批量，直接等于executeBatch。

#### 确定执行SQL时的参数

相对于批量类型2——多SQL批量，批量类型3的参数反而简单一些，不过就是循环处理@Execute注解，而每一个@Execute的处理，和批量类型1非常类似，先根据sqlRef属性方法确定执行的sqlId，然后根据@BatchParam确定对应的执行参数，所不同的是：

* @BatchParam注解的value()属性方法起作用了，表示当前sqlId是否为一个子批量类型，默认为true，可以使用false表示一个简单执行
* 求子批量类型的集合类型参数时，根对象不同了，批量类型1的根对象就是@BatchParam注解所在的入参，而批量类型3的根对象，是整个入参按照Mybatis原生方式组装的包装对象
* @Execute注解还包含condition()的属性方法，表示需要执行的条件表达式，如果为空，表示需要执行，这个条件表达式的求值根对象也是整个入参按照Mybatis原生方式组装的包装对象

最终也是返回一个集合类型的参数。

### 回顾与总结

回顾一下批量类型动态代理接口的三个基本问题处理，发现有一个前提，那就是事先已经知道是什么类型的批量了，而事实上，事先我们并不知道，那怎么确定是什么类型的批量呢？

如果一开始就讲怎么确定批量类型，很难说的清，但现在回过头来看就已经再简单不过了：

* 如果接口方法有@Executes注解，则视作批量类型3——混合批量处理
* 如果接口方法有@SqlRefs注解，则视作批量类型2——多SQL批量处理，进一步：
* 如果没有@BatchParam注解参数，则视作批量类型2.1——多SQL单参数
* 否则视作批量类型2.2——多SQL多参数
* 如果接口方法既没有@Executes，也没有@SqlRefs，则视作批量类型1——单SQL批量处理

再总结一下批量执行动态代理接口方法中三个基本问题的处理：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **批量类型** | **批量类型1 （单SQL批量）** | **批量类型2 （多SQL批量）** | | **批量类型3 （混合批量）** |
| **2.1单参数** | **2.2多参数** |  |
| **特征** | 无@Executes 无@SqlRefs | 含@SqlRefs 不含@BatchParam | 含@SqlRefs 含@BatchParam | 含@Executes |
| **sqlId/sqlId组** | 原生用法 @SqlRef | @SqlRefs @SqlRef | | @Executes @Execute中@SqlRef |
| **执行方法名** | executeBatch | | | |
| **执行参数** | @BatchParam | 原生用法 | @BatchParam | @Executes @Execute中@BatchParam |

## 事务用法说明

### 直接使用JDBC时的事务控制

直接使用JDBC时的事务控制如下所示：



可以看到：事务性代码主要有5步：

1. 打开数据库连接
2. 设置不要自动提交，开启事务
3. 执行成功完成后提交，提交事务
4. 执行失败后回滚事务
5. 关闭数据库连接

这5个步骤和业务逻辑没有太大关系，并且都是模板式代码，而这正为Spring AOP提供了用武之地。

### Spring声明式事务

Spring的声明式事务是Spring的一大亮点，也是Spring AOP的最成功案例，其本质上也很简单，无非就是做我们上一小节说的5个步骤。

注：这里只是简单介绍Spring的事务管理，感兴趣的同事可以自行查找相关资料。

#### Spring事务管理的局限性约束

Spring内部使用Bean容器和ThreadLocal变量来处理声明式事务，这同时也就带来了两个约束：

* Spring事务管理不能跨线程：Spring在事务中，将获取到的数据库连接Connection对象存放到ThreadLocal变量中，因此只有当前线程才能获取到这个Connection对象，从而导致Spring管理的事务不能跨线程。
* Spring事务管理必须是Bean方法：Spring决定是否进行事务拦截的前提是调用方法所在类必须是Spring容器管理的Bean，也即是说必须要求是Bean方法，而我们在bean内部互相调用的方法，并没有经过Spring的拦截，因此即便加上了事务注解也不起作用。

注意：

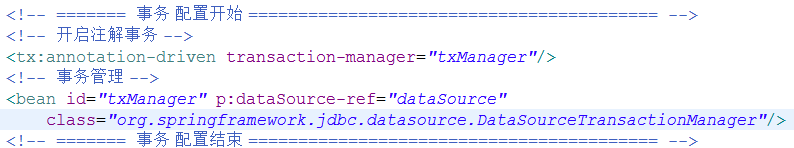
事务注解Transactional只是一个注解，它本身对程序运行不会有任何影响，只有当其它程序（这里就是Spring事务模块）设别这个注解并做相应的处理之后，才真正起作用。一定要明白这一点，才不会有明明加了Transactional注解却没有起作用的困惑，关键是看这个事务注解有没有被Spring设别——这里再重复强调：Spring只会设别Bean容器中的bean方法，不会设别bean内部的互相调用。

说明：

如果想在不被Spring容器管理的类上添加事务，可以使用Spring的编程式事务管理，但写起来比较麻烦。

#### Spring声明式事务的典型用法

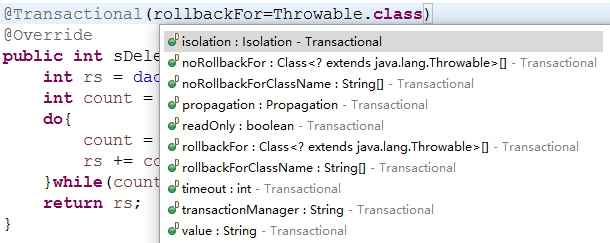
* 开启注解事务、配置事务管理器



* 在需要事务的方法上面添加事务注解，比如：



事务注解有很多选项，如下图：

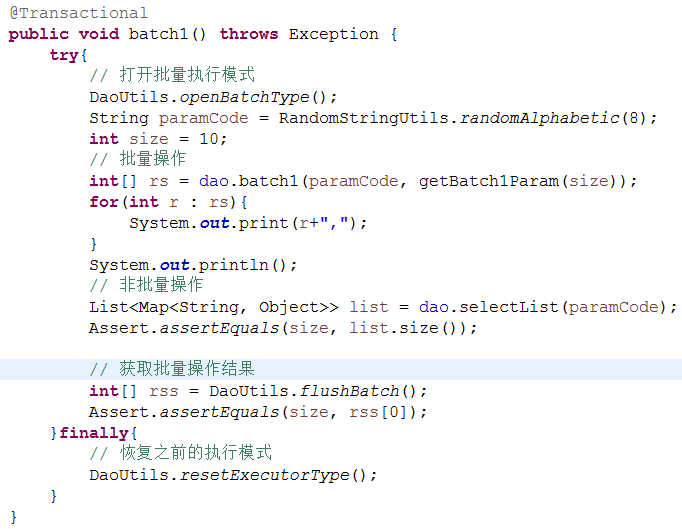


说明如下：

* 隔离级别（isolation）:
* ISOLATION\_DEFAULT:使用数据库默认的事务隔离级别
* ISOLATION\_READ\_UNCOMMITTED:允许读取尚未提交的修改，可能导致脏读、幻读和不可重复读。
* ISOLATION\_READ\_COMMITTED:允许从已经提交的事务读取，可防止脏读、但幻读，不可重复读仍然有可能发生。
* ISOLATION\_REPEATABLE\_READ：对相同字段的多次读取的结果是一致的，除非数据被当前事务自生修改。可防止脏读和不可重复读，但幻读仍有可能发生
* ISOLATION\_SERIALIZABLE:完全服从ACID隔离原则，确保不发生脏读、不可重复读、和幻读，但执行效率最低。
* 事务回滚机制：
* 不需要回滚的异常类/类名称(noRollbackFor/noRollbackForClassName)：
* 需要回滚的异常类/类名称(rollbackFor/rollbackForClassName)：
* 默认情况下，只在方法抛出运行时异常的时候才回滚(runtime exception)。而在出现受阻异常(checked exception)时不回滚事务，这个ejb的回滚行为一致。
* 传播级别(progagation):
* PROPAGATION\_REQUIRED（XML文件中为REQUIRED)：表示当前方法必须在一个具有事务的上下文中运行，如客户端有事务在进行，那么被调用端将在该事务中运行，否则就重新开启一个事务。（如果被调用端发生异常，那么调用端和被调用端事务都将回滚），这也是默认的事务传播级别。
* PROPAGATION\_SUPPORTS(XML文件中为SUPPORTS）:表示当前方法不必需要具有一个事务上下文，但是如果有一个事务的话，它也可以在这个事务中运行。
* PROPAGATION\_MANDATORY(XML文件中为MANDATORY）:表示当前方法必须在一个事务中运行，如果没有事务，将抛出异常。
* PROPAGATION\_NESTED(XML文件中为NESTED):表示如果当前方法正有一个事务在运行中，则该方法应该运行在一个嵌套事务中，被嵌套的事务可以独立于被封装的事务中进行提交或者回滚。如果封装事务存在，并且外层事务抛出异常回滚，那么内层事务必须回滚，反之，内层事务并不影响外层事务。如果封装事务不存在，则同PROPAGATION\_REQUIRED的一样。
* PROPAGATION\_NEVER（XML文件中为NEVER):表示当前方法不应该在一个事务中运行，如果存在一个事务，则抛出异常。
* PROPAGATION\_REQUIRES\_NEW(XML文件中为REQUIRES\_NEW)：表示当前方法必须运行在它自己的事务中。一个新的事务将启动，而且如果有一个现有的事务在运行的话，则这个方法将在运行期被挂起，直到新的事务提交或者回滚才恢复执行。
* PROPAGATION\_NOT\_SUPPORTED（XML文件中为NOT\_SUPPORTED）:表示该方法不应该在一个事务中运行。如果有一个事务正在运行，他将在运行期被挂起，直到这个事务提交或者回滚才恢复执行。
* 是否只读(readOnly): 只读的含义是指，如果后端数据库发现当前事务为只读事务，那么就会进行一系列的优化措施。它是在后端数据库进行实施的，因此，只有对于那些有可能启动一个新事务的传播行为（REQUIRED,REQUIRES\_NEW,NESTED）的方法来说，才有意义。（测试表明，当使用JDBC事务管理器并设置当前事务为只读时，并不能发生预期的效果，即能执行删除，更新，插入操作）。
* 超时时间(timeout):
* 有的时候为了系统中关键部分的性能问题，它的事务执行时间应该尽可能的短。因此可以给这些事务设置超时时间，以秒为单位。我们知道事务的开始往往都会发生数据库的表锁或者被数据库优化为行锁，如果允许时间过长，那么这些数据会一直被锁定，影响系统的并发性。
* 因为超时时钟是在事务开始的时候启动，因此只有对于那些有可能启动新事务的传播行为（REQUIRED,REQUIRES\_NEW,NESTED）的方法来说，事务超时才有意义。
* 事务管理器(value/transactionManager):Spring容器中配置的事务管理器ID

### Beneform4j平台事务

* 在不涉及到批量操作时，按上述示例添加事务注解即可
* 批量类型的操作已经内置事务，无需再单独添加事务注解
* 若服务层中既有非批量调用操作，又有批量调用操作，同时，这个两个操作还处于同一个事务中。这种情况下，由于Mybatis本身不支持在同一个事务中切换执行方式（执行方式见Mybatis中的枚举类ExecutorType），因此不做其他处理的情况下，会抛出异常。
* 若想实现在同一个事务中即调用批量操作，又调用非批量操作，由于批量操作一定只能用批量执行方式执行，所以可以在调用所有操作方法之前，先打开批量执行模式，在finally块中恢复之前的执行模式。如下例所示：



# WEB控制层

## Spring MVC概述

控制层是基于Spring MVC封装、扩展而来，有必要将Spring MVC涉及的一些基础性内容介绍一下，这对于新同事来说，应该有些帮助，对于老同事，就当是一次复习。但是Spring MVC博大精深，不可能也没有必要在这里讲的面面俱到，如果需要更全面的掌握、更深入的了解，可以参考官方文档，或者研读Spring MVC的源码。

### 动态WEB技术发展历程



* 静态HTML：早期的Web应用全部是静态的HTML页面，用于将一些文本信息呈现给浏览者，但不具备与用户交互的能力，没有动态显示的功能。
* CGI（通用网关接口）： CGI(common gateway interface)是最早使用的动态WEB技术，通过为不同的平台，不同的web server 编写插件编写应用接口，来满足通过 web 方式编写应用的需求。当时流行的方式包含 NSAPI/ISAPI，使用 Perl 来编写 CGI 程序。CGI 最大的问题就是线程并发的问题，当时给很多人的感觉是 CGI 访问速度慢，其主要原因是应用程序所编写的 CGI 没有考虑多线程。
* Servlet：1997年，Java广泛使用，Servlet技术迅速成为动态Web应用的主要开发技术。作为一种跨平台语言的服务器端技术，其一经产生就备受瞩目，采用 Servlet 开发的应用，不用考虑平台，多  
  线程等让人头疼的问题，使得开发人员专注于业务逻辑的实现，大大解放了生产力。但是，在 Servlet 中嵌入 html无疑是开发人员的噩梦，开发效率让人不敢恭维。
* ASP、JSP（Java Server Pages）：1998年，微软发布了ASP 2.0，随后，由Sun带领的Java阵营，立即发布了JSP标准。JSP必须先编译成Servlet，实质上还是Servlet，但是JSP编写比Servlet简单，也将业务逻辑和显示逻辑分离开来了，但随着企业应用复杂度不断增加，维护成本也越来越大。
* 企业应用规范MVC模式：随后就是Java阵营发布了一套完整的企业开发规范：J2EE（现在已经更名为Java EE），紧跟着微软也发布了ASP.NET技术，它们都采用一种优秀的分层思想，力图解决Web应用维护困难的问题。
* JSP MODEL1：用户通过浏览器之间访问 web 应用的 JSP 页面， JSP 提供 UI 显示，JavaBeans 处理数据库访问和业务逻辑。这种开发方式最大的优势是直接、简单，对于小型应用，可以很方便、快速地进行开发。
* JSP MODEL2：JSP Model 2 和 JSP Model 1 最大的区别是引入了 MVC 模式的概念，即 M(Model:业务逻辑)，V(View:系统UI)，C(Controller:控制)分离，用户的所有请求提交给 Controller，由 Controller 进行统一分配，并且采用推的方式将不同的 UI 显示给用户。

### Servlet API简要说明

下面是Servlet API主要接口、类的一个结构图（做了部分简化，没有加入异步等API）：



如上图，可以将这些API分成四种类型：

* Servlet公共API
* ServletContext：Servlet上下文环境对象，也是JSP内置对象中的应用对象Application，全局唯一，所有Servlet共享
* Servlet、ServletConfig：Servlet及其配置对象，Servlet是一个具有生命周期的接口，定义的方法如下：

|  |  |  |
| --- | --- | --- |
| 接口/方法名称 | | 接口/方法签名 |
| Servlet | | public interface Servlet |
|  | init | public void init(ServletConfig config)  throws ServletException; |
| service | public void service(ServletRequest req, ServletResponse res)  throws ServletException, IOException; |
| destroy | public void destroy(); |
| getServletConfig | public ServletConfig getServletConfig(); |
| getServletInfo | public String getServletInfo(); |
| ServletConfig | | public interface ServletConfig |
|  | getInitParameter | public String getInitParameter(String name); |
| getInitParameterNames | public Enumeration<String> getInitParameterNames(); |
| getServletName | public String getServletName(); |
| getServletContext | public ServletContext getServletContext(); |

* GenericServlet：Servlet的一个抽象实现类，是所有Servlet实现的类父类，需要强调的是GenericServlet不但实现了Servlet接口，同时还实现了ServletConfig接口，使用起来非常方便。
* Filter、FilterConfig、FilterChain：过滤器和Servlet几乎完全相同，也是一个有生命周期的接口，定义的方法如下：

|  |  |  |
| --- | --- | --- |
| Filter | | public interface Filter |
|  | init | public void init(FilterConfig filterConfig) throws ServletException; |
| doFilter | public void doFilter(ServletRequest request, ServletResponse response,FilterChain chain)  throws IOException, ServletException; |
| destroy | public void destroy(); |
| FilterConfig | | public interface FilterConfig |
|  | getInitParameter | public String getInitParameter(String name); |
| getInitParameterNames | public Enumeration<String> getInitParameterNames(); |
| getServletName | public String getFilterName(); |
| getServletContext | public ServletContext getServletContext(); |
| FilterChain | | public interface FilterChain |
|  | doFilter | public void doFilter(ServletRequest request, ServletResponse response)  throws IOException, ServletException; |

* ServletRequest、ServletInputStream：请求对象和请求输入流
* ServletResponse、ServletOutputStream：响应对象和响应输出流
* Servlet Http API：
* HttpServlet：继承于GenericServlet，主要是在service方法中根据请求方法分发至对应的服务方法，比如将GET请求转发至doGet、POST请求转发至doPost等。
* HttpServletRequest：继承于ServletRequest，添加了获取Cookies、请求头等相关方法。也是JSP内置对象中的request。
* HttpServletResponse：继承于ServletResponse，添加了设置Cookies、响应头等相关方法。也是JSP内置对象中的response。
* HttpSession：会话对象。
* Part：上传对象，Servlet 3 新添加。
* 侦听器：对Servlet对象（应用对象、请求对象、会话对象等）生命事件、属性事件等进行侦听，侦听器全部继承自标志接口（只是一个标志，没有实际方法）java.util.EventListener。这些侦听器和事件都比较简单，一般看名称就知道什么意思，可以参照下表：

|  |  |  |  |
| --- | --- | --- | --- |
| Servlet对象 | 侦听器 | 事件 | 事件对象 |
| ServletContext | ServletContextListener | contextInitialized | ServletContextEvent |
| contextDestroyed | ServletContextEvent |
| ServletContextAttributeListener | attributeAdded | ServletContextAttributeEvent |
| attributeRemoved | ServletContextAttributeEvent |
| attributeReplaced | ServletContextAttributeEvent |
| ServletRequest | ServletRequestListener | requestInitialized | ServletRequestEvent |
| requestDestroyed | ServletRequestEvent |
| ServletRequestAttributeListener | attributeAdded | ServletRequestAttributeEvent |
| attributeRemoved | ServletRequestAttributeEvent |
| attributeReplaced | ServletRequestAttributeEvent |
| HttpSession | HttpSessionListener | sessionCreated | HttpSessionEvent |
| sessionDestroyed | HttpSessionEvent |
| HttpSessionActivationListener | sessionDidActivate | HttpSessionEvent |
| sessionWillPassivate | HttpSessionEvent |
| HttpSessionAttributeListener | attributeAdded | HttpSessionBindingEvent |
| attributeRemoved | HttpSessionBindingEvent |
| attributeReplaced | HttpSessionBindingEvent |
| HttpSessionBindingListener | valueBound | HttpSessionBindingEvent |
| valueUnbound | HttpSessionBindingEvent |

* 事件：侦听器侦听的事件对象，实际上就是侦听器方法的参数，这些参数全部继承自java.util.EventObject。

### Spring MVC概述

初步了解了动态WEB技术和Servlet API之后，接下来熟悉一下Spring MVC。Spring MVC的核心是DispatcherServlet，我们可以从该类入手。

#### DispatcherServlet继承体系

首先，DispatcherServlet也是一个Servlet，其继承体系结构如图所示：

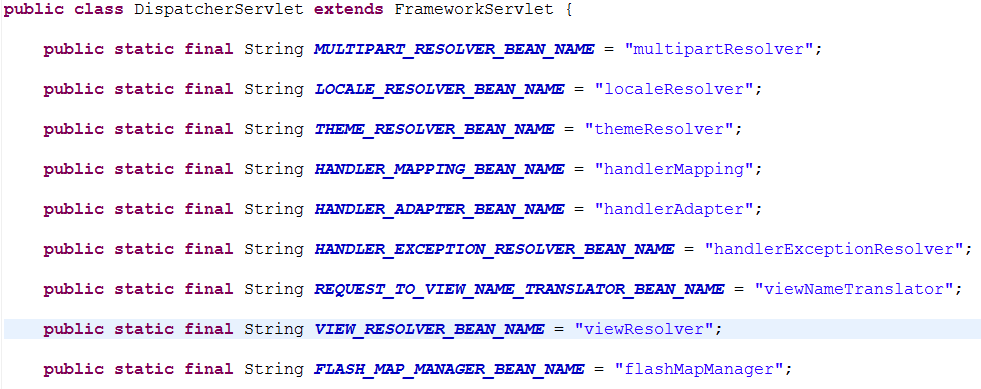


可以看到，DispatcherServlet体系中有5个类、6个接口，其中：

* 来自于的Servlet API的2个接口、2个类前面已经做过说明；
* 来自于Spring本身的四个接口也很简单，Aware接口是一个标志接口，没有任何方法，其含义就是具备感知Spring中组件的能力，比如ApplicationContextAware接口就可以感知Spring容器，而EnvironmentAware则可以感知当前运行的环境，最后一个接口EnvironmentCapable则表示环境适配，也就是可以获取当前运行的环境。
* HttpServletBean：将Servlet作为Spring的Bean处理，提供了一个空的初始化方法。
* FrameworkServlet：使用Spring中WebApplicationContext容器的Servlet，添加了WebApplicationContext容器的初始化处理。
* DispatcherServlet：Spring MVC的核心控制器。

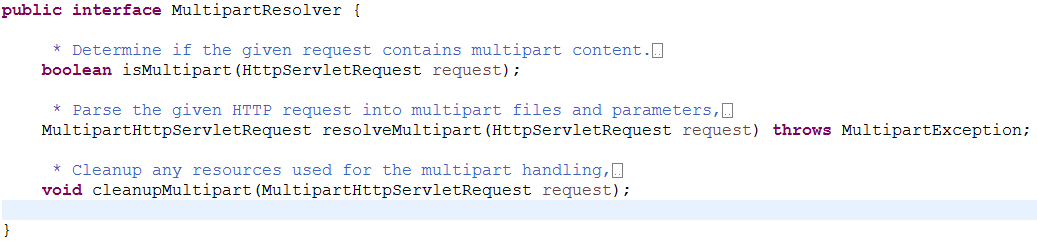
#### DispatcherServlet九大组件概述

下面是DispatcherServlet的一个源码截图：



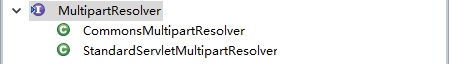
可以看到，其中有九大组件：

##### MultipartResolver：上传解析器



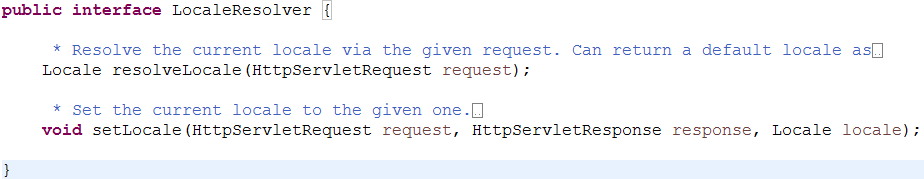
* isMultipart：判断是否上传请求
* resolveMultipart：转换为上传请求
* cleanMultipart：清除上传过程中产生的临时文件

实现类继承体系图：



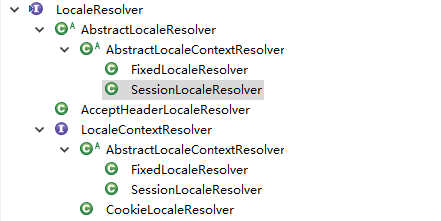
可以看到有两个实现类，分别使用apache-commons工具包和Servlet 3中的Part实现。

##### LocaleResolver：国际化解析器



* resolveLocale：根据请求解析国际化对象
* setLocale：设置国际化对象，保存国际化对象

实现类继承体系图：



从Spring 4开始，添加了LocaleContext接口，用于延迟Locale对象的获取，也可方便开发人员自己扩展Locale对象的解析：

**public** **interface** LocaleContext {

Locale getLocale();

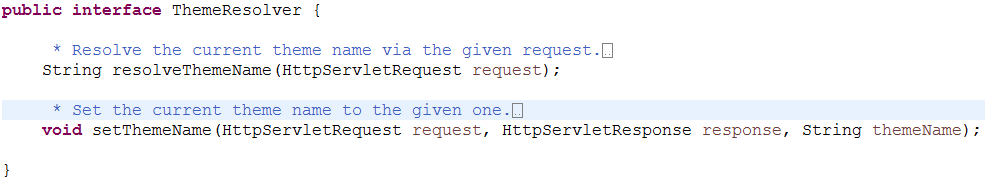
}

然后可以看到如下实现类：

* FixedLocaleResolver：返回固定的Locale对象
* SessionLocaleResolver：返回存储于Session中的Locale对象
* AcceptHeaderLocaleResolver：根据请求头获取Locale对象，即调用request.getLocale()
* CookieLocaleResolver：根据Cookies中特定名称对象解析成Locale对象，默认名称为CookieLocaleResolver.class.getName() + ".LOCALE"

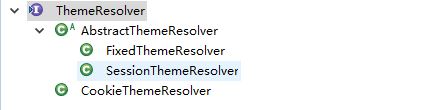
这五个实现类都没有持久化，所以在实际使用中，可以自己实现一个持久化的解析器，然后装配使用。

##### ThemeResolver：主题解析器



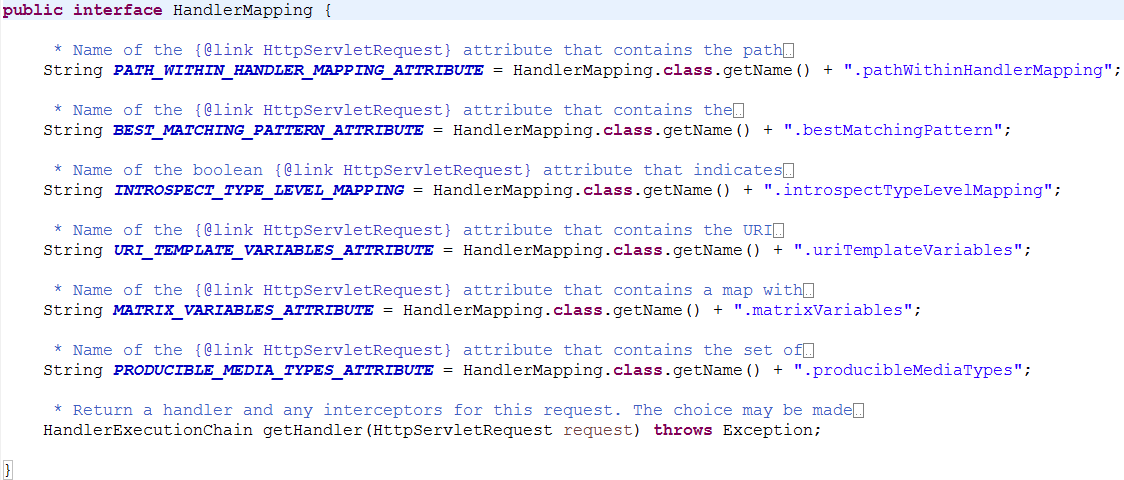
* resolveThemeName：根据请求解析主题名称
* setThemeName：设置主题名称，保存主题名称

实现类继承体系图：



和LocaleResolver类似，就不赘述了。

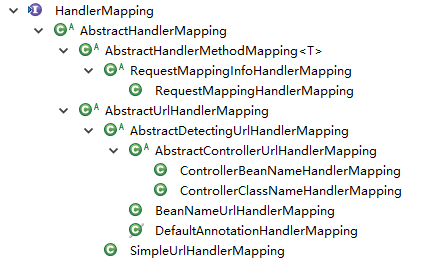
##### HandlerMapping：客户请求与后台处理器的映射



除了几个常量之外，只有一个方法：

* getHandler：获取处理器执行链，包括处理器及其拦截器

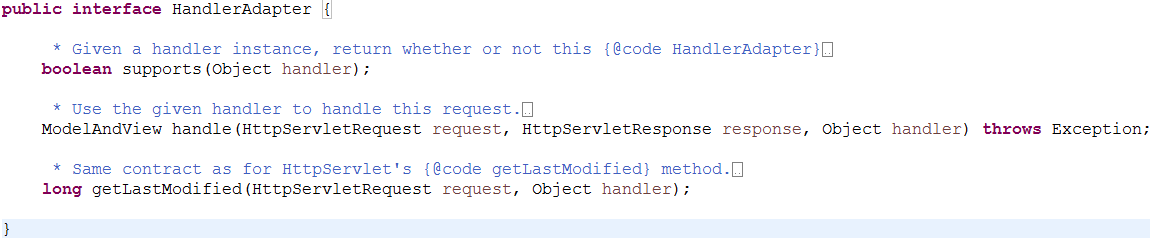
HandlerMapping是Spring MVC非常关键的一个接口，其实现类继承体系图如下：



可以看到，实现类主要有两大体系：

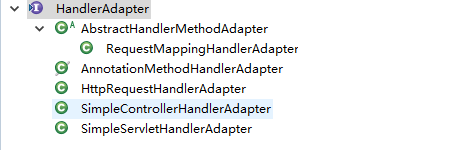
* 根据Url查找处理器AbstractUrlHandlerMapping：主要逻辑是将请求中的url依据某种策略（如果处理器的bean名称，处理器的类名）映射为处理器
* 根据处理器方法查找处理器AbstractHandlerMethodMapping：主要逻辑是根据处理器中的处理方法来查找最匹配的处理器，用的最广泛就是该体系中的实现类RequestMappingHandlerMapping。

##### HandlerAdapter：执行后台处理器的适配器



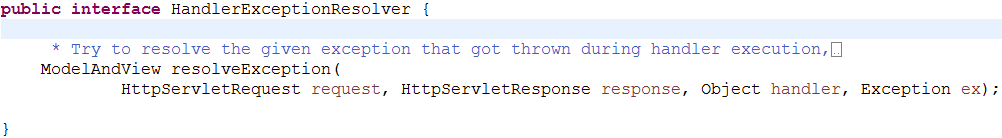
* supports：是否支持找到的处理器
* handler：实际执行处理器，返回ModelAndView对象
* getLastModified：获取最后修改时间，判断是否使用缓存等

HandlerAdapter是Spring MVC中另外一个核心接口，其实现类继承体系图如下：



其中AnnotationMethodHandlerAdapter已废弃，可以不用考虑，而HttpRequestHandlerAdapter、SimpleControllerHandlerAdapter、Simple ServletHandlerAdapter三个实现类都是针对特定类型的Handler处理，非常简单。RequestMappingHandlerAdapter则是和RequestMappingHandler Mapping对应，用于适配一般的处理器方法，这个实现类非常复杂，功能非常强大，也非常灵活，给开发人员提供了很多扩展点，是整个Spring MVC的核心实现类。

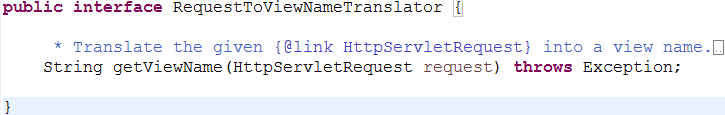
##### HandlerExceptionResolver：处理器异常的解析器



* resolveException：处理异常，返回ModelAndView对象

需要特别强调的是：这里只是处理器的异常，不包括视图渲染过程中的异常

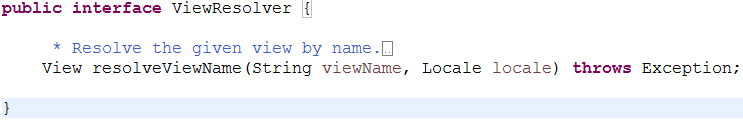
##### RequestToViewNameTranslator：默认视图名的解析器



* getViewName：根据请求获取默认视图名称

注意：默认视图名的解析器，只有找到了处理器，并且程序执行正常，但没有返回逻辑视图的情况下才生效，如果没有找到处理器，或者处理器执行异常，都不会使用该组件。

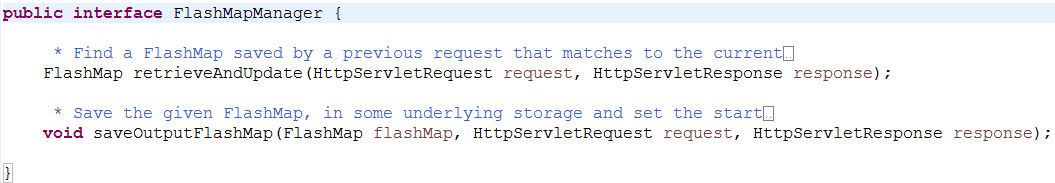
##### ViewResolver：视图解析器



* resolveViewname：将逻辑视图名称解析为物理视图

##### FlashMapManager：FlashMap管理器

FlashMap是什么？因为Spring MVC支持redirect请求，而一般地redirect请求之后，相关的请求参数也就失效了，而Spring MVC则通过FlashMap，在转发前保存请求参数，在转发后恢复参数，从而达到redirect也能够传递参数的作用。



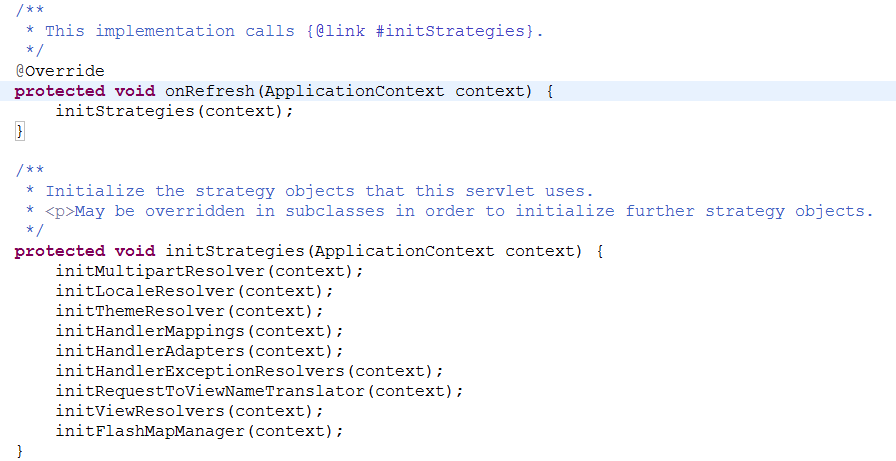
* retrieveAndUpdate：恢复FlashMap
* saveOutputFlashMap：保存FlashMap

这些组件职责清晰，分工明确，非常容易理解，组合在一起就构成了Spring MVC的整个架构体系。Beneform4j平台在WEB控制层几乎所有的封装和扩展都是围绕这九大组件而产生的。

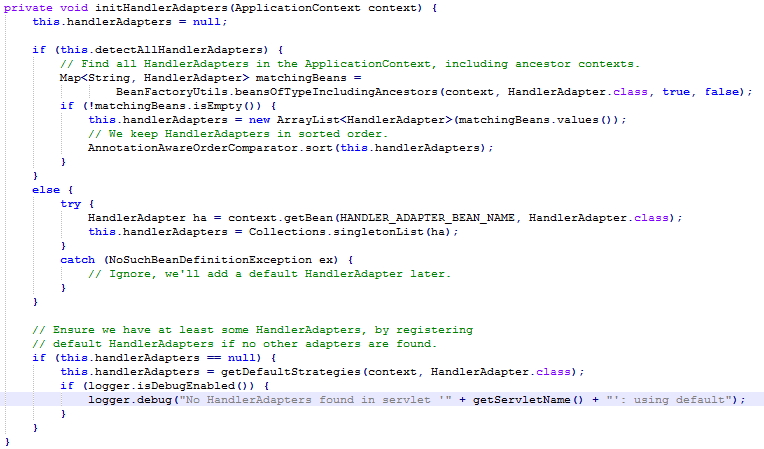
#### DispatcherServlet组件初始化策略

回到DispatcherServlet，看一下组件是怎初始化的，下面以最核心的HandlerAdapter为例：

* 初始化入口：实现父类提供的初始化方法

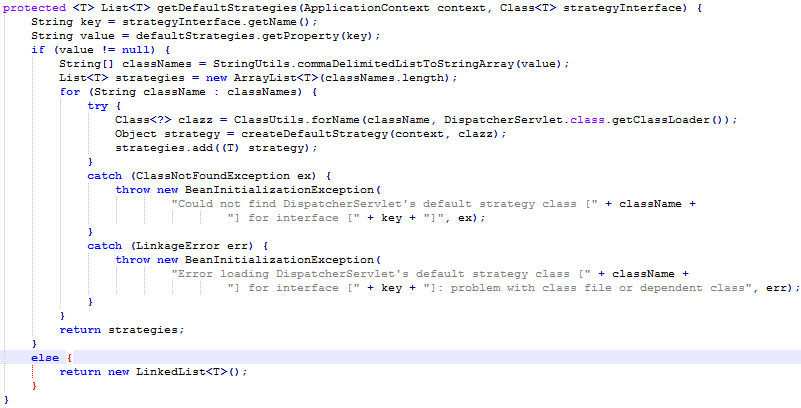


* 初始化：先侦测Spring容器中的HandlerAdapter类，如果有配置，则按优先级排序，否则使用默认策略获取实现类

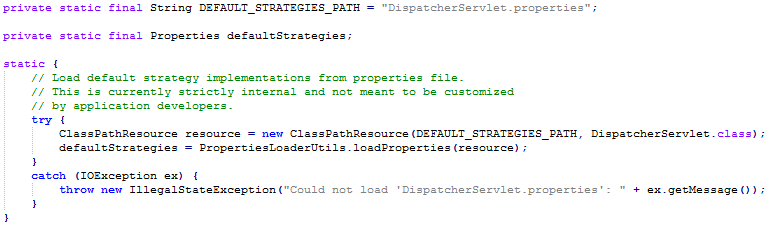


注意：如果使用mvc命名空间的注解驱动，会自动在Spring容器中添加实现类RequestMappingHandlerAdapter。

* 默认初始化策略：根据组件接口从配置文件中查找默认实现类



配置文件即DispatcherServlet类路径下的DispatcherServlet.properties文件。



#### DispatcherServlet请求处理流程

了解DispatcherServlet的九大组件及其初始化策略后，再看一下他们是怎么互相协作完成客户端请求的：



1. 用户发送Http请求，根据web.xml配置文件，确定是否由Spring MVC的总控制器接受请求。
2. 总控制器接受到请求后，根据请求信息（包括URL、GET/POST方法、请求头、请求参数等）和装配的HandlerMapping组件，找到处理请求的第一个非空Handler，并添加相应的HandlerInterceptor拦截器，组成一个HandlerExecutionChain。
3. 找到Handler后，找到第一个可以适配该处理器的HandlerAdapter组件，然后使用HandlerAdapter组件来调用Handler，这里会先执行拦截器的执行前处理preHander()，再执行Handler，执行过程中出现异常，就使用异常处理器处理，如果正确执行但没有返回逻辑视图，则应用默认请求视图设置默认视图，最后再执行拦截器的执行后处理postHander()。
4. 处理链执行完后返回一个ModelAndView对象，其中包含了逻辑视图名和模型数据。需注意的是，处理器Handler可以返回表示逻辑视图的字符串，可以返回模型数据，也可以返回ModelAndView对象，最终都由HandlerAdapter适配为一个ModelAndView对象。
5. 调用视图解析器ViewResolver将逻辑视图解析成物理视图。
6. 将模型数据渲染至物理视图。
7. 将物理视图作为响应给用户，最终响应可以根据不同的解析器返回HTML页面、XML或JSON数据，或者文档下载的二进制流。

下面再画一个图来帮助大家理解：



#### 总结

* Spring MVC相当强大，这里只是简要提及，要想深入了解，可参考官方文档，或者深入源码（推荐）
* Spring MVC核心控制器DispatcherServlet，包含九大组件，每一个组件都是一个扩展点（称之为一级扩展点），其中HandlerAdapter组件的实现类RequestMappingHandlerAdapter更是有着基本的重要性，在其中提供了参数解析、结果处理、消息转换等二级扩展点
* Spring MVC组件从Spring容器初始化，如果Spring容器中不包括某类组件，则使用DispatcherServlet.properties属性文件使用默认组件。同时，需要特别强调的是，使用MVC命名空间的注解驱动，会向Spring容器添加默认组件（注意，这里的默认组件和DispatcherServlet.properties中的默认组件并不完全相同）

## Beneform4j中Spring MVC组件配置

Beneform4j中Spring MVC组件或者扩展组件配置主要有二个文件：

* spring-view.xml：比如MVC的注解驱动、主题解析器、异常处理器等
* spring-beneform4j：比如默认请求视图映射等

应用开发人员想找到组件或扩展组件的入口，可以从这个两个文件中找出相关的配置。

## 请求映射HandlerMapping

HandlerMapping作用是根据请求查找处理器。

### RequestMappingHandlerMapping

MVC注解驱动会自动注入RequestMappingHandlerMapping，该实现类综合使用请求中的各种条件（url、参数、请求头、请求方法等等）来查找使用@RequestMapping注解的方法处理器，具体用法可参照Spring MVC相关文档。

### ViewMappingHandlerMapping

实现默认请求映射。

Beneform4j默认已经装配该实现类，优先级是所有HandlerMapping中最低。这里的默认请求映射和DispatcherServlet中的RequestToViewNameTranslator不同，后者只能在找到处理器的情况下起作用，而前者是在找不到其它处理器的情况下起作用。这里的默认请求映射返回的视图优先级低于用户返回的视图。具体映射关系由平台装配的IViewMapping组件确定。

**public** **interface** IViewMapping {

/\*\*

\* 根据请求获取视图名称

\* **@param** requestInfo 请求信息

\* **@return** 默认视图名称

\*/

**public** String getViewname(IRequestInfo requestInfo);

/\*\*

\* 是否覆盖用户视图

\* **@return** 是否覆盖用户返回的视图

\*/

**public** **boolean** overrideUserView();

}

IViewMapping实现类可以在spring-beneform4j.xml中配置，根据方法overrideUserView的返回值分成两组，一组优先级高于处理器返回视图（即使开发人员返回了viewname，也会被覆盖），一组优先级低于处理器返回视图（只在没有找到处理器或处理器没有返回viewname下生效）。目前Beneform4j平台中两种优先级各有一个实现类：

* MimeParamViewMapping：根据请求参数中特定参数确定返回视图，优先级高于用户视图，默认特定参数名为mime，比如请求参数包含mime= download，可以配置为返回下载视图，包含mime=json，可以配置为返回JsonView。
* RequestUrlViewMapping：根据请求url的模式确定返回视图，优先级低于用户视图，比如请求url满足\*\*/search{suffix}，可以配置为返回\*\*/list{suffix}，这里的模式支持类似{suffix}的零个或多个命名占位符和\*\*/、/\*\*两个通配符（目前只支持开头和结尾两个通配符），但不是严格意义上的正则表达式。

### ProxyHandlerMapping

代理请求映射。

默认注解驱动情况下，所有的HandlerMapping已经按顺序装配好，但是某些特殊情形下，需要自己实现自定义优先级的HandlerMapping，用于拦截一些公共处理，比如拦截所有的\*.page请求，返回\*.jsp视图，又比如判断请求是否已经通过安全认证等。在实现HandlerMapping的同时，需要有一个Handler执行具体业务逻辑，并且还需要实现一个适配类HandlerAdapter来调用Handler，显得比较麻烦，因此Beneform4j中实现了一个公共的HandlerMapping（也即是ProxyHandlerMapping）、Handler和HandlerAdapter，并提供一个接口用于业务逻辑处理，从而通过代理的方式来处理所有需要自定义的情形。

简单的说，就是ProxyHandlerMapping及其处理器和适配类负责协调Spring MVC和业务逻辑接口，而具体业务逻辑由接口实现类实现，这个接口就是IProxyHandler：

**public** **interface** IProxyHandler {

/\*\*

\* 是否支持该请求

\* **@param** request 请求对象

\* **@return**

\*/

**public** **boolean** isSupport(HttpServletRequest request);

/\*\*

\* 实际处理，返回MVC视图

\* **@param** request 请求对象

\* **@return**

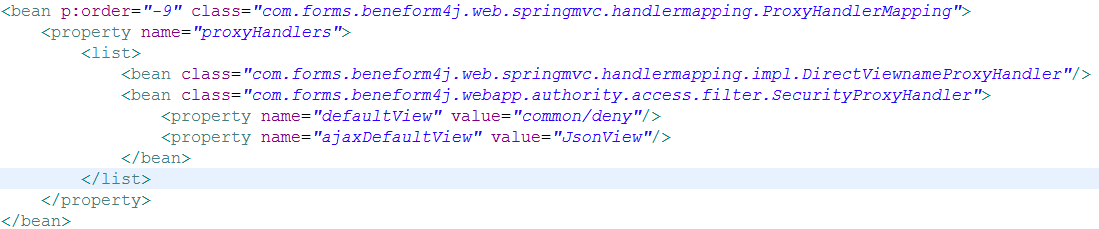
\*/

**public** ModelAndView handler(HttpServletRequest request);

}

该接口有两个方法，第一个用于判断是否支持当前请求，第二个在支持当前请求的情况下用于执行业务逻辑并返回ModelAndView对象以响应当前请求。Beneform4j平台提供一个实现类DirectViewnameProxy Handler，用于将\*.page请求转换为\*.jsp，Beneform4j示例项目中提供了一个权限认证实现类SecurityProxyHandler，主要逻辑是先在Servlet过滤器SecurityFilter中进行权限认证，如果权限认证不通过，则将认证失败的信息设置到当前请求对象Request中（属性名SecurityFilter.**class**.getName()+".AUTHORIZATION\_INFO\_ATTRIBUTE\_NAME"），然后在SecurityProxyHandler的isSupport方法中通过检查当前请求对象中是否包含认证失败信息来判断是否需要使用ProxyHandlerMapping（可能有同事会问，为什么权限认证失败时不直接在SecurityFilter中直接返回？是因为如果直接返回，就不能使用Spring MVC的视图解析组件带来的便利了，而通过ProxyHandlerMapping，则可以伪装成请求完全执行正常，只是返回认证失败的视图，从而可以享用视图解析、视图渲染等便利）。

默认情形下，是没有装配ProxyHandlerMapping，如需使用，开发人员需要自行配置（推荐在spring-view.xml中配置），同时可以配置优先级，优先级的数值越小，优先级越高，例如：



这里因为SecurityProxyHandler需要先执行，所以优先级配置成-9。

## 处理器调用适配器HandlerAdapter

HandlerMapping虽然可以找到Handler以及相应的拦截器，然而Handler并不是一个接口，无法使用统一的方式去调用，因此还需要能够统一调用Handler的一个适配接口，这个接口就是HandlerAdapter了。一般来说，每种类型的Handler都有相应的HandlerAdapter。

### RequestMappingHandlerAdapter

MVC注解驱动会默认注入RequestMappingHandlerAdapter，用于调用RequestMappingHandlerMapping找到的处理器，也就是含有@RequestMapping注解的方法处理器。

我们知道，注解方法处理器非常灵活，可以是任意形式签名的公共方法，这也给调用该处理器增加了难度，可以想象，有两个最基本的问题要处理：

* 根据方法签名，绑定实际参数，如有需要，进行参数校验
* 处理方法返回值

下面是一个简化截图：



#### 参数绑定HandlerMethodArgumentResolver

方法处理器的参数解析器，其中默认装配的参数解析器有：



可以看到，这些参数解析器可以分成四类：

* 基于注解的参数解析器
* 基于参数类型的参数解析器
* 用户自定义参数解析器
* 其它通用型参数解析器

具体解析器这里就不展开了，可以参考Spring MVC的官方文档，这里说明一下Beneform4j中自定义的参数解析器：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 实现类 | 适用情况 | 绑定值 | 备注 |
| 分页参数 | PageArgumentResolver | 类型为IPage | PageUtils.createPage() |  |
| 类型为RowBounds | new PageAdapter(PageUtils.createPage()) |  |
| 上传参数 | UploadFileArgumentResolver | 类型为IUploadFile | ServletHelp.getUploadFile()中第一个值 | 非上传请求 返回null |
| 类型为IUploadFile[] | ServletHelp.getUploadFile() |
| 请求信息 | RequestInfoArgumentResolver | 类型为IRequestInfo | ServletHelp.getRequestInfo() |  |
| 用户信息 | UserArgumentResolver | 类型为IUser | 由子类获取IUser实例 | 会话超时 返回null |
| [含有@User注解](mailto:含有@User注解) | 由子类获取IUser实例 @User注解值为空、id或userId时 返回userId 否则返回IUser中对应注解值的属性 |

注意：

在beneform4j-demo项目中，有一个WebTool工具类，可以获取用户信息，但是在实际项目开发中，不建议直接使用该工具类，而应该使用UserArgumentResolver，在控制器的方法签名中直接添加IUser类型的参数即可。

#### 结果处理HandlerMehtodReturnValueHandler

方法处理器的返回结果处理器，默认装配的结果处理器有：



可以分成如下几类：

* 单一目标型的结果处理
* 基于注解的结果处理
* 多目标的结果处理
* 自定义的结果处理
* 通用型结果处理

其中Beneform4j自定义的结果处理器有两个：

* PageReturnValueHandler：分页查询的结果处理器，和jQuery EasyUI适配的结果处理
* JsonBodyReturnValueHandler：含有JsonBody注解的结果处理器

注意：由于这两个结果处理器的作用有更好的实现，因此建议不再使用。

#### Http消息转换器HttpMessageConverter

Http消息转换器是什么？简单一点说，HttpMessageConverter就是Http请求/Http响应与Java对象之间的转换器。为什么需要这个接口呢？我们知道，Spring MVC可以使用ResponseBody或RequestBody注解，直接返回JSON对象或从JSON参数中绑定参数，而其中关键的一点就是使用HttpMessageConverter接口，将不同的返回值类型转换为JSON字符串后直接输出至HttpServletResponse。

MVC注解驱动装配的消息转换器逻辑可以参考下面的伪源码：



虽然ResponseBody可以直接将返回值作为JSON字符串处理，非常便利，但是也有几个局限性，比如不能够灵活的取舍数据对象中的属性，不能方便的对数据对象进一步包装等，为此，Beneform4j平台引入@JsonBody注解和JsonBodyHttpMessageConverter转换器进行补充，这也是目前Beneform4j平台引入的唯一一个消息转换器。由于@JsonBody用法很基础，涉及内容也较多，将在下一节单独说明。

### ViewMappingHandlerAdapter

和ViewMappingHandlerMapping对应。

### ProxyHandlerAdapter

和ProxyHandlerMapping对应。

## 服务端数据校验

参数绑定过程中，如果有需要，则会进行输入参数的校验，称之为服务端的数据校验，以区别于前端使用JS的客户端数据校验。

Spring MVC支持JSR303的校验规范，Beneform4j平台直接使用Spring MVC校验框架，集成Hibernate-validator实现，只是在校验信息的提示方面做了一些修改，以便于在前端和Spring的标签解绑，便于使用jQuery EasyUI等Ajax框架。

### 环境配置

* 添加依赖包，因为beneform4j平台包已经添加该依赖，因此应用添加平台依赖即可

<dependency>

<groupId>org.hibernate</groupId>

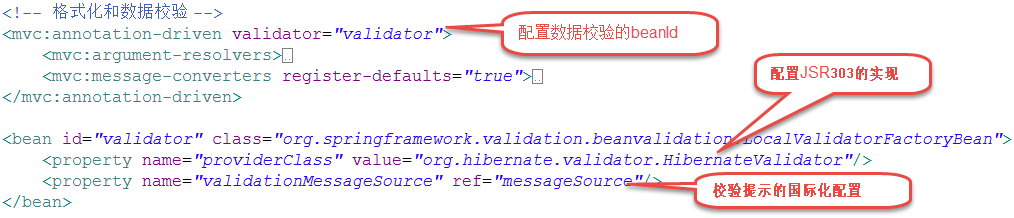
<artifactId>hibernate-validator</artifactId>

<version>5.2.4.Final</version>

</dependency>

* 添加spring配置，指定使用Hibernate-validator校验框架，同时指定校验提示的国际化配置

spring-view.xml



spring-common.xml：



### 数据校验示例

先看一个数据校验示例，这个示例选自beneform4j-demo中的机构维护，包括三个功能操作，输入参数要求分别如下：

* 新增机构：机构号、机构名称、父机构号不能为空
* 修改机构：机构号、机构名称不能为空，（父机构号的变更通过移动机构实现，因此这里的父机构可以为空）
* 移动机构：机构号、父机构号不能

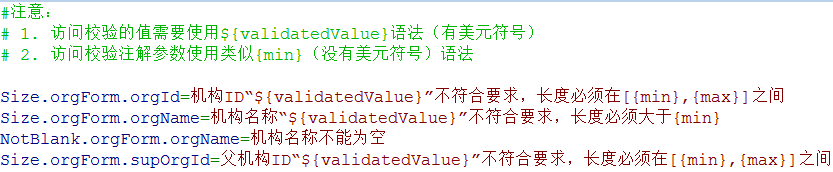
下面看一下怎么实现：

#### 步骤一：编写领域模型类



#### 步骤二：编写国际化提示信息

在locale/validator/validator\_zh\_CN.properties（中文）中添加如下配置：



如需要其它语言，比如繁体中文，可在locale/validator/validator\_zh\_TW.properties中添加相应配置。

配置中可以访问属性值，使用含美元符号的语法，也可以访问注解参数，比如@Size注解中的min值，使用不含美元符号的语法。

#### 步骤三：在控制层添加校验注解



如截图中说明，添加分组校验的注解，如果不需要分组，可以简单使用@Valid注解。

#### 步骤四：处理校验结果

Spring MVC原生方式：

* 可以在被校验的参数后面，添加BindingResult类型的参数，然后通过这个参数访问校验结果
* 如果没有添加BindingResult类型的参数，并且校验不通过，Spring MVC会抛出BindingException异常。

由于直接抛出异常，对于Ajax请求来说，不够友好，因此Beneform4j平台针对数据校验抛出的绑定错误进行了拦截，封装成JSON格式输出至前台。最终输出格式类似：

{

success:false,

data:{

errors:[

{name:’orgId’, type: ‘field’, message:’机构号不能为空’},

{name:’orgId’, type: ‘field’, message:’机构号不能为空’},

……

{name:’org’, type:’global’, message:’机构表单校验提示’}

]

}

}

其中，type表示校验信息的种类，field表示为某个属性的校验提示信息，global表示整个对象的校验提示信息。

因此，在UI端，解析返回的JSON字符串后，可以根据需要提示校验信息。

### 校验注解

由于Beneform4j使用Hibernate-validator校验框架，因此所谓的校验注解，也就是JSR303提供的标准注解和Hibernate-validator提供的扩展注解。这些注解具体怎么使用，可以参考更详细的Hibernate-validator文档，这里简单列举一下这些注解，以做索引之用：

* @AssertFalse
* @AssertTrue
* @DecimalMax(value=,inclusive=)
* @DecimalMin(value=,inclusive=)
* @Digits(integer=,fraction=)
* @Future
* @Max(value=)
* @Min(value=)
* @NotNull
* @Null
* @Past
* @Pattern(regex=,flag=)
* @Size(min=, max=)
* @Valid
* @CreditCardNumber(ignoreNonDigitCharacters=)
* @EAN
* @Email
* @Length(min=,max=)
* @LuhnCheck(startIndex= ,endIndex=,checkDigitIndex=,ignoreNonDigitCharacters=)
* @Mod10Check(multiplier=,weight=,startIndex=,endIndex=,checkDigitIndex=,ignoreNonDigitCharacters=)
* @Mod11Check(threshold=,startIndex=,endIndex=,checkDigitIndex=,ignoreNonDigitCharacters=,treatCheck10As=,treatCheck11As=)
* @NotBlank
* @NotEmpty
* @Range(min=,max=)
* @SafeHtml(whitelistType= ,additionalTags=,additionalTagsWithAttributes=)
* @ScriptAssert(lang=,script=,alias=)
* @URL(protocol=,host=,port=,regexp=,flags=)

### 其它用法

Spring MVC的校验还支持更多用法，具体可参考相关资料，这里简要提及一下：

* 自定义校验注解
* 硬编码提示信息：在校验提示信息中，可以直接使用中文，但是不推荐这么做
* 使用默认国际化信息的Key值
* 多个属性的联合校验
* ……

## JsonBody注解族

### 目标

Beneform4j平台借鉴Spring MVC原生的ResponseBody注解，添加JsonBody注解来序列化Java对象，最重要的目标是：

在控制层的返回视图需要变更的情况下，仅仅通过修改控制层的注解即可实现，而不需要修改服务层的返回结果，从而实现控制层和服务层的高度解耦。

这个目标又可以分解为下面几个更细化的功能目标：

* 重命名属性对象：将数据中的属性使用别名输出，比如表示树型数据节点的打开/关闭状态的属性，可能在jQuery EasyUI视图中为state，而在bootstrap视图中可能为status，这种情况下，就可以通过配置@JsonBody中属性别名来实现，而不需要修改服务层返回的ITree树对象
* 忽略属性对象：忽略不需要序列化的属性
* 筛选属性对象：选择需要序列化的属性
* 属性值的转换：将原始属性值转换为另外一个简单值或另外一个嵌套对象，或者将数据对象中的一个属性或多个属性值进行再次封装等等，比如需要将数据对象中的用户新添加的所有属性值组装成为一个userData属性，最后序列化成userData:{property1:userData1,property2:userData2}这种形式。

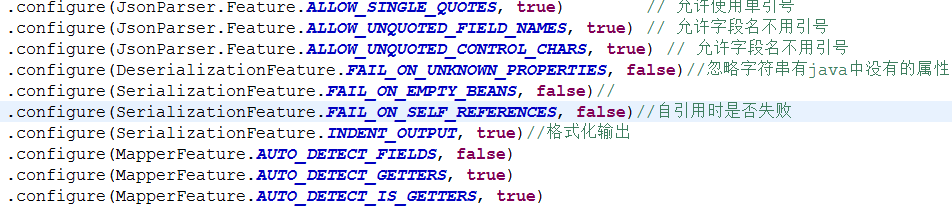
可能有同事会问，Jackson不是已经提供了@Ignore、@Alias、@JsonView等一系列注解来实现属性筛选、属性别名等功能吗？为什么Beneform4j平台还需要自己实现呢？不错，Jackson是有@Ignore等功能注解，但是这些注解灵活度不够，只能加入到数据模型的定义类里面去，而有时候，数据模型的定义类可能根本就不是项目组本身开发的，也就无从添加注解，即便是项目组自己开发，可以加注解，也不能灵活的适应于不同视图的序列化。

### Jackson配置

正常情况下，JSON序列化会输出对象的所有属性值，Jackson包则提供了是否需要序列化null属性值的选项，Beneform4j平台默认配置为不输出null值，如需要改变，可在程序初始化过程中调用如下代码：



Beneform4j常用默认配置还有：



至于这些配置具体什么含义，还有其它的哪些配置，可以参考Jackson的官方文档或者研读相关源码。需要特别指出的是，平台中这些配置是全局共享的，一旦修改，整个系统就会全部修改。

### @JsonBody注解

定义如下：

@Target({ElementType.***METHOD***})

@Retention(RetentionPolicy.***RUNTIME***)

@ResponseBody

**public** **@interface** JsonBody {

/\*\*

\* 属性组配置：同一属性多次配置以前面的为准

\* 1.property或property#1或property#true 配置属性名称

\* 2.property#false或property#0 表示忽略属性

\* 3.property#alias 表示将属性property使用别名alias输出，隐式表示是需要输出，但不影响属性筛选模式

\*/

String[] fields() **default** {};

/\*\*

\* 使用@JsonField注解配置属性组，优先级整体比fields要高，但jsonFields中同一属性多次配置以前面的为准

\*/

JsonField[] jsonFields() **default** {};

/\*\*

\* JsonWrapper对象的beanId

\*/

String value() **default** "";

/\*\*

\* JsonWrapper对象的Class，优先级比beanId要低

\*/

Class<? **extends** IJsonWrapper> wrapperClass() **default** DefaultJsonWrapper.**class**;

}

说明：总共有四个属性方法，分为两组

* 属性组配置
* fields：以字符串数组的形式配置属性组，同一属性多次配置以前面的为准，其中每一个字符串又可以按照如下的格式配置，其含义分别是：

1、property或property#1或property#true 配置输出的属性

2、property#false或property#0 配置忽略的属性

3、property#alias 表示将属性property使用别名alias输出

* jsonFields：以@JsonField注解数组的形式配置属性组，优先级比fields高，但是jsonFields内部同一属性多次配置，则以前面的为准。
* 包装器配置
* value：Spring容器中表示包装器对象的beanId，优先级比wrapperClass要高，生命周期由Spring容器管理（单列、原型等），必须实现IJsonWrapper而接口，默认值为空格，表示没有配置。
* wrapperClass：表示包装器对象的实现类，每次使用都重新创建一个实例，该实现类必须有一个无参的构造器，优先级比value表示的beanId低，默认值表示没有配置。

### @JsonField注解

@JsonBody注解中的jsonFields是一个@JsonField类型的数组，@JsonField定义如下：

@Target({ElementType.***METHOD***})

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** JsonField {

/\*\*

\* 属性名称

\*/

String value();

/\*\*

\* 属性别名

\*/

String alias() **default** "";

/\*\*

\* 是否忽略

\*/

**boolean** ignore() **default** **false**;

/\*\*

\* 转换类的beanId

\*/

String convertBean() **default** "";

/\*\*

\* 转换类的实现类，优先级低于convertBean

\*/

Class<? **extends** IJsonConverter> convertCls() **default** DefaultJsonConverter.**class**;

}

说明：

* value：属性名称，必须配置
* alias：属性别名，默认值空格，表示没有配置
* ignore：是否忽略该属性，默认为false，不忽略
* 转换器配置
* converBean：Spring容器中表示转换器对象的beanId，优先级比convertCls要高，生命周期由Spring容器管理（可以单列，可以原型等），必须实现IJsonConverter而接口，默认值为空格，表示没有配置。
* convertCls：表示转换器器对象的实现类，每次使用都重新创建一个实例，该实现类必须有一个无参的构造器，优先级比convertBean表示的beanId低，默认值表示没有配置。

从@JsonField的定义可以看到，@JsonBody中的fields是没有转换器的@JsonField的简化，@JsonField比字符串分隔模式的fields功能更强大，这也是为什么@JsonField优先级要高于fields的一个原因。

### 属性筛选模式

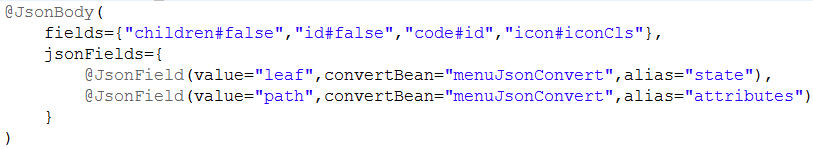
从@JsonBody及@JsonField注解的定义看，很自然的就产生了一个问题：如果一个JavaBean有10个对象，其中配置了2个需要输出的属性，忽略了2个属性，那对于没有配置的其它6个属性，是输出呢？还是不输出？

为了清楚说明这个问题，引入一个“属性筛选模式”的概念。

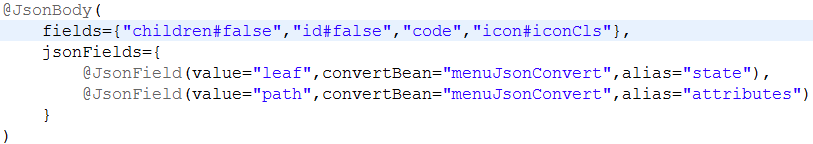
根据有没有明确指定需要序列化的属性，分成如下两种：

* 否定模式：没有明确指定任意一个需要序列化的属性，就进入否定模式，也即只要没有明确指定忽略，就输出该属性
* 肯定模式：只要有一个明确指定需要序列化的属性，就进入肯定模式，也即只有被明确指定需要序列化的属性才会输出，其它默认属性全部不输出

关于属性筛选模式，需要特别注意的是：属性别名和属性转换器不影响属性筛选模式的切换，虽然这两个配置隐式表示需要输出该属性。比如下面的配置：



虽然有属性别名，有属性转换器，但是没有仅仅表示需要输出的属性配置，因此属性筛选模式为否定模式，也就是没有配置忽略的就输出。如果修改一下：



这里code属性没有其它指意，只是表示需要输出该属性，这种情况下，属性筛选模式变成肯定模式，未配置属性或配置忽略的属性，统统忽略，也就是只输出code、icon、leaf、path属性。

### 属性转换器接口IJsonConverter

看一下前面反复提及的属性转换器接口IJsonConverter，定义如下：

**public** **interface** IJsonConverter {

/\*\*

\* 转换

\* **@param** container 容器，一般为JavaBean或Map对象

\* **@param** property 属性名称

\* **@param** value 属性值

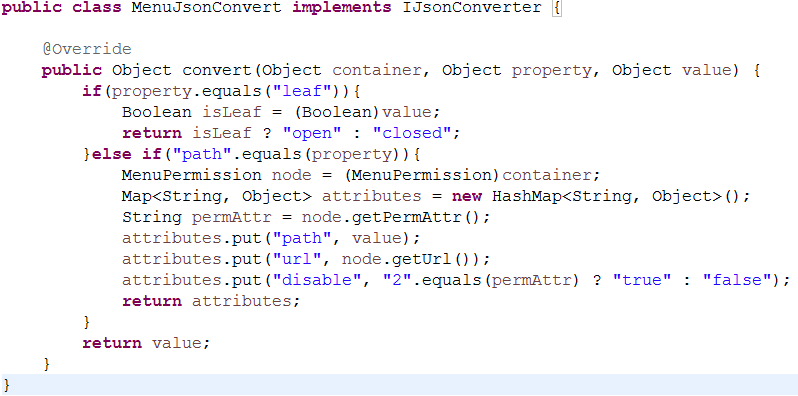
\* **@return** 转换后的值

\*/

**public** Object convert(Object container, Object property, Object value);

}

其中只有一个方法，就是根据数据对象、属性名称、属性值转换为需要输出的值。如果结合alias别名配置，属性转换器可以实现非常强大的功能，比如将数据对象中的多个属性组合成新的属性：



### 数据包装器接口IJsonWrapper

在@JsonBody注解中，还有一个数据包装器接口IJsonWrapper，定义如下：

**public** **interface** IJsonWrapper {

/\*\*

\* 包装对象

\* **@param** original 原始对象

\* **@return** 包装后的对象

\*/

**public** Object wrap(Object original);

}

作用就是在序列化之前，对原始数据对象进行包装。比如在分页查询中，服务层返回的是List对象的数据列表，但是视图层不仅需要数据列表，还需要类似总记录数、分页大小等的分页信息。而这些分页信息的获取可以使用统一的方式获得，因此就可以实现一个分页包装器来统一处理，简化开发：

**public** **class** PageJsonWrapper **implements** IJsonWrapper{

@Override

**public** Object wrap(Object original) {

Map<String, Object> wrapper = **new** HashMap<>();

wrapper.put("data", original);

IPage page = PageUtils.*getPage*();

wrapper.put("rows", page.getTotalRecords());

**return** wrapper;

}

}

当然，实际代码中为了灵活性考虑，应将data、rows参数化。

### @JsonBody子注解族

像上例中的分页包装器，如果直接使用@JsonBody注解，需要每一个分页方法的注解中都要添加PageJsonWrapper的配置（beanId或PageJsonWrapper.class），这是很繁琐的，也不利于后期维护。

因此Beneform4j平台针对一些常见的情形，实现了一组包装器，并且每个包装器都配之以相应的@JsonBody子注解。

#### 示例说明

仍然以分页包装器为例说明：

* 定义分页包装器PageJsonWrapper
* 定义分页注解@PageJsonBody

**public** **@interface** PageJsonBody {

/\*\*

\* 同JsonBody中的fields

\*/

String[] value() **default** {};

/\*\*

\* 同JsonBody中的jsonFields

\*/

JsonField[] jsonFields() **default** {};

}

* 在Controller的方法中添加@PageJsonBody注解

@RequestMapping("list")

@PageJsonBody

**public** List<UserBean> list(UserForm user, IPage page){

**return** service.sList(user, page);

}

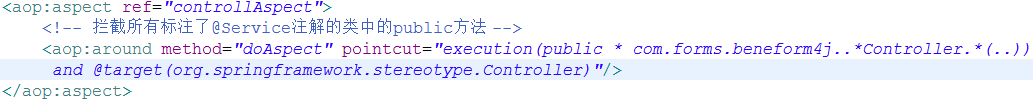
剩下的就是将@PageJsonBody和PageJsonWrapper关联，这个在Beneform4j中也没有写死，可以通过配置的形式（spring-beneform4j）修改默认值。

#### @JsonBody子注解及其包装器一览

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 注解 | 默认包装器 | 功能说明 | 配置名称 | 数据示例 |
| @JsonBody | MapJsonWrapper | 添加执行状态 | defaultJsonWrapper | {  success:true,  data: … } |
| @PageJsonBody | PageJsonWrapper | 添加总记录条数 | pageJsonWrapper | {  total:100,  rows: [{},{}] } |
| @ListJsonBody | NoWrapJsonWrapper | 不包装 |  |  |
| @NoWrapJsonBody | NoWrapJsonWrapper | 不包装 |  |  |
| @TreeJsonBody | TreeJsonWrapper | 包装为树型结构 |  |  |
|  | ExceptionJsonWrapper | 包装异常信息 | exceptionWrapper | {  success:false,  data:{  code : …,  trackId: …,  level: …,  message:…,  detail:...  } } |

#### 注意事项

* @JsonBody相关功能的实现借助了控制层的AOP，因此spring配置里面需要将相应的控制器添加到控制层AOP中。



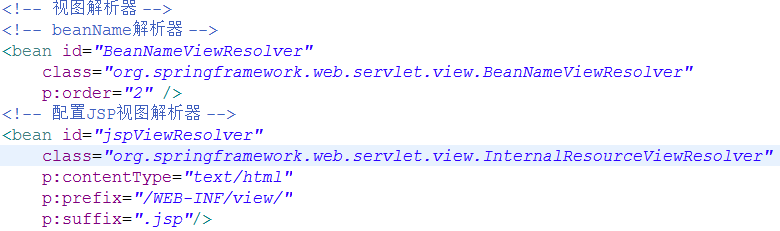
曾经有同事将这里的com.forms.beneform4j..\*配置修改，导致分页查询出现异常的问题。

* @JsonBody实现中，如果有别名、输出属性、忽略属性或者属性转换器的配置（也就是和默认输出不同），ObjectMapper就不是全局共享的，所以如果没有特殊要求，基于性能考虑，@JsonBody等注解尽量不要添加参数。

## 视图解析

### 视图解析器

Beneform4j平台暂未添加新的视图解析器，示例项目中使用的视图解析器都是Spring MVC原生提供的，配置在spring-view.xml中：



说明：

* jspViewResolver是默认视图解析器，其优先级最低，作用是将逻辑视图加上前后缀，然后查找对应jsp页面，进而渲染。
* BeanNameViewResolver视图的作用是将逻辑视图名作为beanId，在Spring容器中查找bean，查到之后再使用该bean进行渲染。比如@Controller中返回“JsonView”，然后在Spring容器中可以找到对应的bean，所以就按该bean渲染。如返回“JsonView2”，在Spring容器中找不到对应的bean，然后就使用下一个视图解析器（这里就是jspViewResolver），解析成/WEB-INF/view/JsonView2.jsp进行渲染，如果没有这个页面，就报渲染异常（注意：是渲染异常，不是处理器处理异常）。
* 如果还需要其它的视图解析器，也可以在spring-view.xml中一并配置，注意器order属性，数值越小，优先级越高。
* 其它更多相关知识参考Spring MVC官方文档，或者研读相关源码。

### 视图

在beneform4j-demo项目中，spring-view.xml中定义了两个JSON视图：

* JsonView：以默认格式序列化输出
* TextJsonView：因为在IE低版本浏览器中，默认格式序列化输出时，会弹出一个下载的对话框，因此这里添加了一个输出内容格式的配置contentType=*"text/html"*。

应用开发人员可以在Controller中直接返回这两个视图。

后续会考虑添加诸如文件下载、JasperReport报表等视图。

## 异常处理

### 异常捕获与抛出

参考异常工具类使用说明。

### 视图渲染异常处理

Spring MVC中没有对视图渲染异常做处理，我们可以在web.xml中配置相应错误页面。

### 处理器异常处理

## 国际化

## 主题切换

## 上传支持