JFramework 开发指南

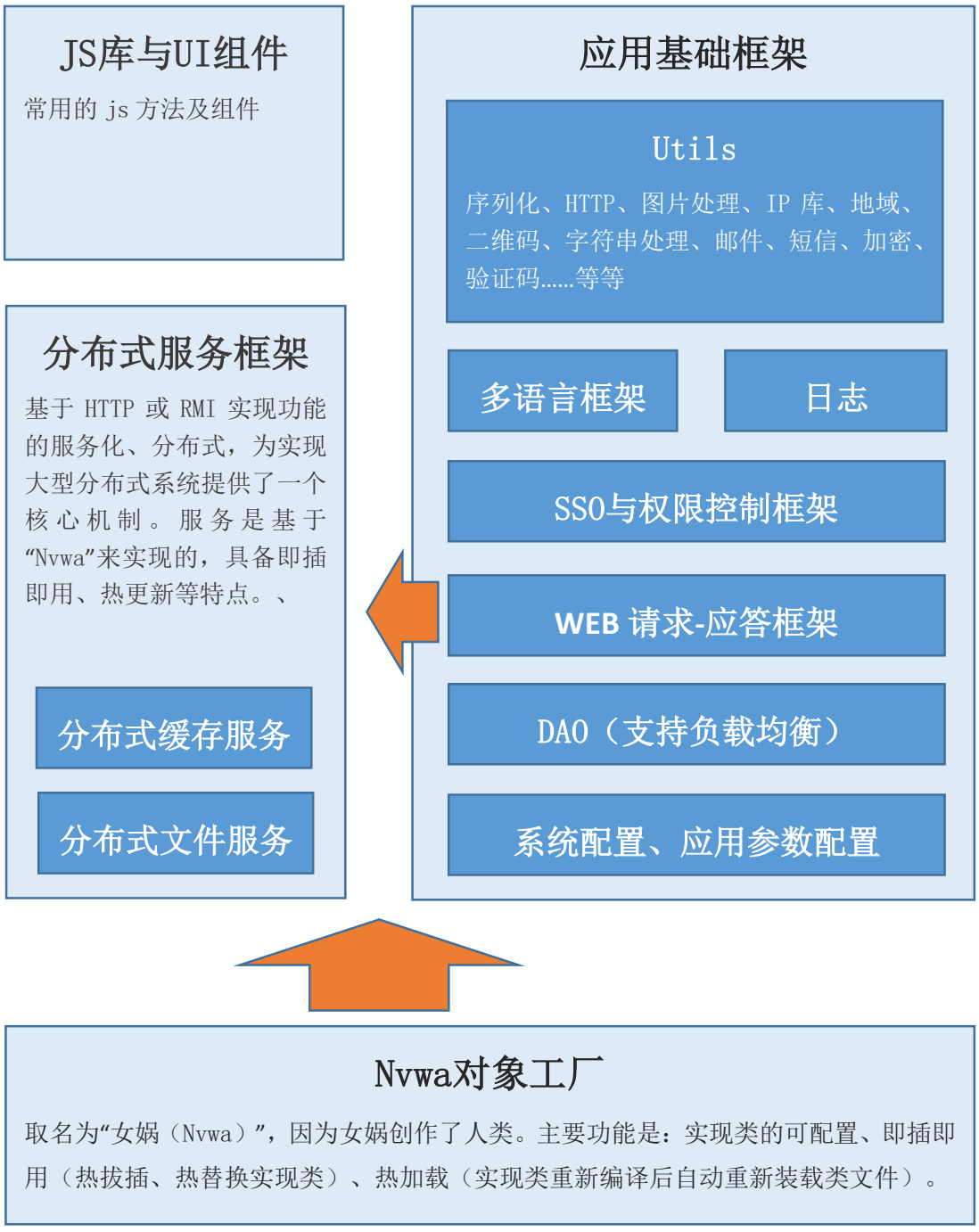
目录

概述

JFramework——J 取自本人名字拼音第一个字母，同时也取 JAVA 之意。框架定位于简单、高效、灵活，采用最基本的 java 技术，去除各种“流行的”、“高大上的”、“不明觉厉的”概念。

整体架构

架构简图



模块间依赖关系

服务的实现类基于 Nvwa；WEB 请求-应答框架的处理类默认是托管给 Nvwa 的（以便实现热

更新），也可通过声明“non-nvwa-object”脱离 Nvwa。

开发指南

Get Started

起始配置文件

位于 config 包中的 jframework.properties 是整个框架启动的入口，它包含关键的路径信息和自定义的其它属性。J.Properties 类负责加载并提供相关 Getter 方法。一个典型的 jframework.properties 内容如下：

```
###关键路径###
WebRoot=JFRAMEWORK_HOME/jframework/ROOT/
ConfigPath=JFRAMEWORK_HOME/ jframework/ROOT/WEB-INF/classes/config/
ClassPath=JFRAMEWORK_HOME/ jframework/ROOT/WEB-INF/classes/
JarPath=JFRAMEWORK_HOME/ jframework/ROOT/WEB-INF/lib/
I18NPath=JFRAMEWORK_HOME/ jframework/ROOT/WEB-INF/I18N/

###日志级别###
LogLevel=LEVEL_ERROR
LogDatabase=jframework
Loggers=1

###JHttp 启用实例个数###
JHttpInstances=1

###rmi###
[rmi]java.naming.factory.initial=com.sun.jndi.rmi.registry.RegistryContextFactory

##sqlite###
#org.sqlite.lib.path=d:/tomcat/webapps/vselected_sso/ROOT/WEB-INF/lib/
#org.sqlite.lib.name=d:/tomcat/webapps/vselected_sso/ROOT/WEB-INF/lib/sqlitejdbc.dll

[group]key1=value1
[group]key2=value2
```

- **关键路径**

1， JFRAMEWORK_HOME 指应用服务器的根路径（不包含\或/），比如 D:\tomcat\webapps，

可以直接写真实路径，也可写 JFRAMEWORK_HOME，但必须设置系统环境变量 JFRAMEWORK_HOME，其值为真实路径（这在安装路径由用户动态决定的情况下很有用，比如打包成 exe 安装时，无需手动更改路径，而通过自动设置环境变量即可）。

2， WebRoot 表示应用的根路径，如 D:\tomcat\webapps\jframework\ROOT。

3， ConfigPath 表示配置文件存放目录，框架及其它模块都将从改目录寻找相应的配置文件。

4， ClassPath 类存放路径、jar jar 包存放路径，Nvwa 对象工厂将成此两目录加载类。

5， I18NPath 多语言配置及资源文件存放目录。

➢ 日志处理

1， LogLevel j.log.Logger 类输出日志的级别，如果日志级别低于 level 将不输出。

2， LogDatabase 请求-应答框架中的常规日志处理类 j.app.webserver.ActionLogger 及业务日志处理类 j.log.JLogger 保存日志的数据库名称。

3， Loggers j.log.JLogger 处理日志的线程数。

➢ HTTP 模块

1， JHttpInstances j.http.JHttp 实例数。

```
1， [group]key1=value1
2， [group]key2=value2
```

➢ 设置一组 key-value

1， 如下所示，[]之间表示分组名，然后在后面紧跟这正常设置 key-value 对。

➢ j.Properties 类提供 Getter

1， 功能如其名，不再赘述，如下：

2， public static String getProperty(String propertyName)

3， public static String getProperty(String groupName,String propertyName)

4， public static java.util.Properties getProperties(String group)

5， public static JUtilKeyValue[] getPropertiesAsArray(String group)

6， public static String getConfigPath()

7， public static String getI18NPath()

8， public static String getWebRoot()

9， public static String getClassPath()

10， public static String getJarPath()

11， ……

系统核心配置文件 sys.xml

Sys.xml 配置了系统最基本、最核心的配置信息，一个典型的 sys.xml 配置文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<sys-config>
    <!--系统 ID，如果作为 sso client，也是 client 的 id-->
    <sys-id>sso</sys-id>

    <!--物理服务器 ID-->
    <machine-id>MY-PC</machine-id>

    <!--系统使用的字符编码格式-->
    <sys-encoding>UTF-8</sys-encoding>

    <!--哪些页面需调用
response.setContentType("text/html;charset="+SysConfig.sysEncoding)-->
    <!--|-|为通配符，代表 0 或多个任意字符-->
    <responseEncodingPages>|-|.handler;|-|.service;|-|.jsp</responseEncodingPages>

    <!--错误信息页面-->
    <error-page>/common/error.htm</error-page>

    <!—系统默认数据库配置-->
    <database name="jframework" min-uuid="0" max-uuid="10000000"/>

    <!--初始化系统-->
    <Initializers>
        <!-- <command retries="1">d:/tomcat/webapps/jservice/WEB-
INF/classes/config/startup.bat</command> -->
        <Initializer init-handler="j.I18N.I18N"/>
        <Initializer init-handler="j.service.Manager"/>
    </Initializers>
</sys-config>
```

上述范例文件中对各项做了注释，不再赘述。

WEB 应用开始的地方

在 web.xml 配置自启动的 servlet（j.sys. Initializers），它将依次调用 sys.xml 中< Initializers>中配置的命令（cmd 命令、bat、sh 等）和初始化模块（j.sys. Initializer 的实现类）。

简单、好用的应用参数配置文件 para.xml 和 para.*.xml

j.sys.AppConfig 类加载 para.xml 和文件名格式形如 para.*.xml 的应用参数配置文件，一个简单的范例文件如下：

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

<root>
  <group name="SYSTEM">
    <para name="time-zone-diff">0</para>
    <para name="http-get-parameter-encoding-convert">false</para>
  </group>

  <group name="HTTP">
    <para name="User-Agent">Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)</para>
    <para name="retries">2</para>
    <para name="retry-interval">3000</para>
  </group>

  <group name="TEST">
    <para name="user-id">user</para>
    <para name="user-pwd">password</para>
    <para name="admin-id">admin</para>
    <para name="admin-pwd">password</para>
  </group>
</root>
```

如上所示，group 表示一组参数值，name 表示分组名，在所有 para..xml 文件中必须唯一，para 表示一个参数，其 name 属性表示参数名（key），标签内文本表示参数值（value）。

特别情况：上述范例文件中所示的 SYSTEM 组内两个参数和 HTTP 组内的三个参数为 Jframework 必须的几个参数——time-zone-diff 表示应用所使用时间与服务器时间的调整（有时懒得去调服务器时间）、http-get-parameter-encoding-convert 指示在用 j.sys.SysUtil.getHttpParameter 方法获取 http 参数时是否进行参数转换（因为应用服务器版本的不同，get 请求传送的中文值有的会乱码，有的则不会，会乱码的则需要进行转换）、HTTP 组内的 参数这在用这在 JHttp 进行 http 请求时设置的 几个关键字（作用如其名，不赘述）。

配置文件修改后系统会自动重新加载

上述几节中提到的配置文件及其它模块所用到的配置文件都绝大部分都实现了修改后自动重新加载的功能（除非被认为确实极少在系统运行时会修改的）。

之所以称之为工厂并不是使用所谓的工厂模式，而是因为它“像工厂一样生成东西”。

开始前先看看 Nvwa 存在的问题

1， 每个使用 Nvwa 来管理的类，都需要编写一个 Interface，这看起来很麻烦，先看一下代码：

```
//SomeClass
SomeNvwaClassImpl impl=（SomeNvwaClassImpl）Nvwa.create(….);
```

如上所示， SomeClass 是一个普通的类，它的上下文中使用的是默认 ClassLoader，而 Nvwa.create 时使用的是实现了热加载的 自定义 ClassLoader，由于 ClassLoader 之间是隔离的，所以 Nvwa create 的 SomeNvwaClassImpl 和 SomeClass 中声明的 SomeNvwaClassImpl 并不是同一个东西，所以上述代码是无法运行的（不能转换的错误）——如果使用一个接口，则由于接口都是默认 ClassLoader 负责加载，就解决上述问题了。

2，静态变量的问题
如果在 Nvwa 管理的类中声明静态变量，如下：

```
// SomeNvwaClassImpl
Public    static    int    someInt=0;

Public void hello(){
      someInt++;
      System.out.print(someInt+” “);
}

//SomeClass
Public static void main(String[] args){
      SomeNvwaClassInterface    objA=Nvwa.create(….);
      objA.hello();

      SomeNvwaClassInterface    objB=new SomeNvwaClassImpl();
      objB.hello();
}
```

objA 和 objB 是由不同 ClassLoader 创建的 ，他们对应的类只是名字和方法相同而已，在系统看来是完全不同的东西，静态变量也是安全没关联的，所以上述 main 方法的运行结果还是：1 1，而不是：1 2

配置文件 nvwa.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
	<!--自定义 ClassLoader，必须是 j.nvwa.NvwaClassLoader 的子类，NvwaClassLoaderAutoRenew 实现了自动热加载功能-->
	<!--Nvwa 有托管机制，无需在下方配置<object>就能实现类的热加载、热拔插，只需调用 Nvwa.entrust()即可将类托管给 Nvwa，并调用 Nvwa.entrustCreate()创建对象-->
	<custom-classloader class="j.nvwa.NvwaClassLoaderAutoRenew">
		<!--哪些类由自定义类加载器进行加载，当 class 的值为__all_implementation 时表示 nvwa.xml 中的所有实现类-->
		<responsible-for class="__all_implementation"/>
		<responsible-for class="j.log.LoggerDefault"/>
	</custom-classloader>
	<object>
		<code>/j.service.hello.words</code>
		<name>测试服务的初始值</name>
		<implementation>j.service.hello.JHelloWords</implementation>
		<singleton>true</singleton>
		<field name="words" type="String" init-value="I love oooo" keep="false"/>
		<parameter key="someParameter" value="someValue"/>
	</object>
	<object>
		<code>/j.service.hello</code>
		<name>测试服务</name>
		<implementation>j.service.hello.JHelloImpl</implementation>
		<singleton>true</singleton>
		<!-- 基本类型初始值，只支持 String,int,long,double,Timestamp,boolean -->
		<!--
		String 字符串型
		int 整型
		long 长整型
		double 双精度浮点型（由于 java 处理 Float 型的精度问题，禁止使用 FLoat 型）
		Timestamp 时间日期型 格式必须为 0000-00-00 00:00:00[.000]  []里的表示可选
		boolean 布尔型 取值为 true / false
		-->
		<field name="counter" type="int" init-value="0" keep="true"/>
		<!-- ref 表示引用另外一个 nvwa.xml 中配置的对象，值为该对象配置的 code -->
		<!-- 所引用对象必须有不带参数的构造函数 -->
		<field name="words" type="ref" init-value="/j.service.hello.words" keep="false"/>
	</object>
</root>
```

通过配置文件应该可大概了解，且不详述。

使用 Nvwa 管理的类

1， 构造函数不带参数的 public static Object create(String code)
2， 构造函数带参数的 public static Object create(String code,Class[] parameterTypes,Object[] parameters)

托管（无需配置文件）

1， 构造函数不带参数的 public static Object entrustCreate(String code,String className,boolean)
2， 构造函数带参数的 public static Object entrustCreate(String code,String className,boolean singleton,Class[] parameterTypes,Object[] parameters)
3， 创建对象前需要先托管，如下：

```
//OneInterface
.....
//OneImpl
....
//OneClass
Nvwa.entrust("someCode", "j.nvwa.lab.OneObjectImpl", true);
OneObject o=(OneObject)Nvwa.entrustCreate("someCode ", "j.nvwa.lab.OneObjectImpl", false);
o.hello();
```

WEB 请求-应答框架

没有 mvc 之类

设计时没有迎合风行的一些概念，在保证简单、易用、高效的同时给开发者更多灵活性和发挥空间。

配置入口文件 actions.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<root >
    <loggers>1</loggers>
    <action-timeout>60000</action-timeout>

    <!--action 请求路径模式-->
    <action-path-pattern>.handler</action-path-pattern>
    <action-path-pattern>.service</action-path-pattern>

    <!--action 配置文件-->
    <actions>
        <module desc="sso">actions.sso.xml</module>
        <module desc="范例">actions.demo.xml</module>
    </actions>
</root >
```

Loggers 表示用来记录 action 日志（常规日志）的线程数。

action-timeout action 处理超时时间，超过这个时间 ActionLogger 会认为本次请求处理超时。

action-path-pattern 表示可使用的请求路径后缀。

actions 中的 module 表示一个 action 配置文件，系统将从这些文件中加载 action 配置，并监控各文件的更新，如有更新则重新加载。

action 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <handler path="/demo" path-pattern=".handler"
class="j.app.webserver.demo.DemoHandler" request-by="request" non-nvwa-obj="true"
singleton="false">
        <action name="测试 1" id="test1" method="test11" respond-with-string="true">
            <log>
                <p>param_a</p>
                <p>param_b</p>
            </log>
        </action>
        <action name="测试 2" id="test2" method="test22">
                <log save-all-parameters="true"></log>
                <navigate condition="ok" type="forward">/demo/ok.jsp</navigate>
                <navigate condition="err" type="redirect">/demo/err.jsp</navigate>
        </action>
    </handler>

    <handler path="/upload" class="j.test.TestUpload" request-by="request">
        <action name="测试 Pipe" id="pipe" method="pipe" respond-with-string="true"/>
        <action name="测试文件上传" id="upload" method="upload" respond-with-
string="true"/>
    </handler>

    <global-navigate condition="error" name="系统错误"
type="redirect">/global/error.html</global-navigate>
    <global-navigate condition="noRight" name="没有权限"
type="redirect">/global/noRight.html</global-navigate>
</root>
```

- **handler**

  表示一个业务处理类，可能包含一个或多个 action。

  Path 表示请求路径（系统使用该值作为 code 将处理类托管给 Nvwa 以实现热更新）。

  path-pattern 表示路径后缀，一般取值.handler 或.service，默认为.handler。

  class 表示处理类名，必须是 j.app.webserver.Jhandler 的子类。

  request-by 表示用什么参数来指示请求哪个 action。

  non-nvwa-obj 表示是否托管给 Nvwa，默认为 false（托管）。

  singleton 是否单例，默认为 true。

  请求一个 action 的 url：/demo.handler?request=test1

- **action**

  name 名称。

  Id 同一个 handler 内必须唯一。

  method 表示调用 handler 的哪个方法，每个方法必须形如：

public void test(JSession jsession,
HttpSession session,
HttpServletRequest request,
HttpServletResponse response) throws Exception;

respond-with-string 表示用 response 的 print 方法直接输出内容（而不是 forward 或 redirect，一般用于前段使用 ajax 方式的情况）。输出内容通过使用 jsession.resultString=…来设置。

- **log**

  如果为 action 配置了 log 项，系统会自动记录请求该 action 的日志，其中 p 表示记录那些参数，如果 save-all-parameters 属性值设为 true 则记录全部参数。

- **navigate**

  可以为每个 action 配置多个 navigate 项，即页面跳转，其中 condition 表示处理结果为该值时跳转到该 navigate 指定的页面，跳转方式包括 redirect 和 forward（forward 可以通过 request.setAttribute 向页面传递对象）。Action 的处理结果由 jsession.result=…来设置。

- **global-navigate**

  全局范围的导航配置，通过 jsession 的 setIsBackToGlobalNavigation 方法设置使用全局到导航。

转换 jsp 为.jhtml 后缀名

为了不暴露 jsp 所处的实际目录，将 jsp 统一放在 WEB-INF/pages/目录下，并将.jhtml 后缀的路径映射到 jsp，比如/aa/index.jhtml 映射到 WEB-INF/pages/aa/index.jsp

开发步骤

1，在 web.xml 中配置 Filter(j.app.webserever.Router)，它负责拦截用户请求，如果请求的是换一个 action，则找到对应的 Handler 进行处理，同时负责 jsp 路径映射。如下：

```
<filter>
      <filter-name>ActionRouter</filter-name>
      <filter-class>
          j.app.webserver.Router
      </filter-class>
   </filter>

   <filter-mapping>
      <filter-name>ActionRouter</filter-name>
      <url-pattern>*.handler</url-pattern>
   </filter-mapping>
   <filter-mapping>
```

```
        <filter-name>ActionRouter</filter-name>
        <url-pattern>*.service</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>ActionRouter</filter-name>
        <url-pattern>*.jhtml</url-pattern>
</filter-mapping>
```

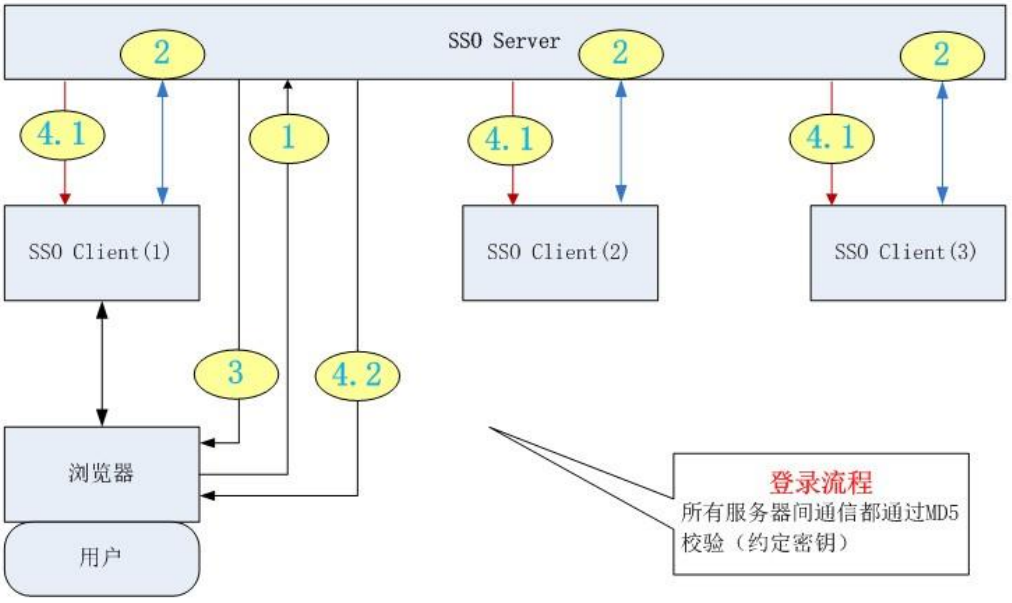
2， 编写一个继承 j.app.webserver.Jhandler 的类，编写你所需要的业务处理方法，方法格式必须如下：`public void test(JSession jsession,HttpSession session,HttpServletRequest request,HttpServletResponse response) throws Exception;`

3， 配置对应的handler和action。

4， 使用类似 xxx.handler?request=…的地址请求相关 action

SSO 与权限控制框架

sso 实现机制



图例说明 **黑色**箭头表示通过浏览器进行的交互（form提交、url跳转），**红色**箭头表示服务器之间的通讯（不通过浏览器），蓝色箭头表示服务器之间可能存在（也可能不存在）的通讯。

第1步 用户通过浏览器将包含用户名、密码、验证码、认证方式等的登录请求提交至SSO Server。

第2步 SSO Server接收到请求后对用户身份验证。用户身份验证分为几种情况——由于现实业务中可能存在这几种情况：（A）各个系统使用统一的用户库（可以是集中的用户数据库、也可以是各个系统的用户库保持同步），这个时候用户身份可由SSO Server来验证，（B）各个系统拥有各自独立的用户库，这时根据用户登录时使用的哪个系统的账号来决定通过哪个系统进行验证，即SSO Server需调用某个SSO Client的接口来验证用户身份，称之为“代理验证”。最复杂的情况是：用户在SSO Client（1）选择用SSO Client（2）的账号登录，则SSO Server需调用SSO Client(2)的接口对用户身份进行验证。

第3步 登录失败，跳转至用户登录请求中指定的登录页面。

第4.1步 登录成功，SSO Server通过调用各系统的登录接口（通过HTTP请求），通知该用户已经登录（包含从哪登录、用的哪个SSO Client用户库、登录时间、登录IP等），各系统收到该通知后须记录下该用户已经登录（此时用户是“在SSO Server已登录、在SSO Client未登录”状态）。

第4.2步 SSO Server通过浏览器，跳转至用户登录时所在系统的登录接口地址，这时登录接口应该查找该用户是否确实登录过（根据4.1步中的记录），如未登录则跳转至用户登录请求中指定的登录页面；否则进行用户信息加载等与该系统业务相关的其它操作，然后跳转至用户登录请求中指定的返回页面（此时用户是“在SSO Server已登录、在本SSO Client已登录，在其它SSO Client未登录”状态）。

注【一】 各系统间是独立用户库、集中单一用户库、用户库保持同步或者兼而有之，这些都没有关系，上述方案都可以适用。用户库如何处理完全取决于实际业务需求，与该SSO方案无关，这保证了方案的通用性和灵活性。

注【二】 各登录验证码需要使用SSO Server提供的URL获得，验证码是有SSO Server来验证的。

用户登录状态获取

用户已经登录过系统 A（SSO Client A），然后在同一会话（同一个浏览器窗口）中访问系统 B，这时系统 B 应该自动登入用户并加载与本系统业务相关的用户信息。有三种方式可以让

B 系统获得用户登录状态并加载用户信息：

1， 访问需要登录才能访问的地址。

2， 主动点击用户信息查询链接。

3， 无刷新的：在用户访问的 B 系统的当前页面内嵌入 iframe，网址为：/sso_info_getter.htm，这些 B 系统就会自动获得用户登录状态，然后不管是否已经登录都会跳转到 /sso_info_getter_login.htm，sso_info_getter_login.htm 的内容为：

<script>try{if(top.onSsoInfoGot) top.onSsoInfoGot();}catch(e){}</script>

所以，可以看出，访问 sso_info_getter.htm 可以获得用户登录状态，然后我们可以定义一个 js 方法 onSsoInfoGot，在里面调用 B 系统的某个地址查询用户登录状态。

获取当前用户

每个业务系统需要指定代表当前登录用户的 User 对象（j.app.sso.User）的实现类，通过(User)session.getAttribute(Constants. SSO_USER)可获得当前登录的用户。

SSO 配置文件 sso.xml

各项含义请见文件中注释。

```
<?xml version="1.0" encoding="UTF-8"?>
<sso>
    <!-- 是否 sso server 端 -->
    <is-server>true</is-server>

    <!-- 单点登录服务器地址 -->
    <server>http://sso.vselected.com/</server>

    <!-- 登录验证类，必须是 j.app.sso.Authenticator 的子类 -->
    <authenticator>j.app.sso.AuthenticatorImpl</authenticator>

    <!-- 是否启用登录验证码 -->
    <verifier-code-enabled>true</verifier-code-enabled>

    <!-- 超时时间,以秒为单位 -->
    <session-time-out>7200</session-time-out>

    <!-- 多长时间用户没有活动表示用户为"离线",以秒为单位 -->
    <online-active-time>300</online-active-time>

    <!-- 对每个 sso client，sso server 启用多少个通知线程 -->
    <notifiers-per-client>2</notifiers-per-client>

    <!-- 如果 SSO Client 或合作站点很多，可能选择将配置信息存于数据库等，使用 j.app.sso.SSOConfiger 接口的实现类进行加装 -->
```

```
    <clients-conf-loader>j.app.sso.SSOConfigLoaderXMLImpl</clients-conf-loader>

    <!-- SSO Client，isssoserver 表示是否就是 sso server 这个应用，true 表示是-->
    <client isssoserver="true">
        <!-- SSO Client ID，sso 框架内唯一 -->
        <id>sso</id>

        <!-- SSO Client 名称 -->
        <name>sso</name>

        <!-- 可用网址（default="true"表示为与 SSO Server 通信的默认网址 -->
        <url default="true">http://sso.vselected.com/</url>

        <!-- 默认登录地址，也可使用绝对地址 -->
        <login-page>/login.htm</login-page>

        <!-- 默认主页，也可使用绝对地址 -->
        <home-page>/index.htm</home-page>

        <!-- 与 SSO Server 通信进行 MD5 校验所使用的约定密钥 -->
        <passport>de6ced-1304bcb8d25-80f8cac1e092ca14301921951cf77f91</passport>

        <!-- SSO Client 登录接口，用于接收来自 SSO Server 的登录通知（包括服务器间通知和浏览器端 URL 通知） -->
        <login-interface>ssoclient.handler?request=ssologin</login-interface>

        <!-- SSO Client 登出接口，用于接收来自 SSO Server 的登出通知（服务器间通知） -->
        <logout-interface>ssoclient.handler?request=ssologout</logout-interface>

        <!-- SSO Client 代理认证接口，SSO Server 调用它来验证用户身份 -->
        <!-- avail 表示该 client 是否向 sso server 提供用户验证功能 -->
        <!--
        for-other-clients 表示该 client 是否向其它 client 提供用户验证功能(即其它 client 是否可用该 client 账号登录）
        _DENY_ALL 表示全不允许
        _DENY:client id,client id... 表示允许这几个 client，多个用逗号分隔
        _ALLOW_ALL 表示全允许
        _ALLOW:client id,client id... 表示允许这几个 client，多个用英文逗号,分隔
        四个策略可同时使用，中间用英文分号;分隔优先级从高到低分别为_DENY,_ALLOW,_DENY_ALL,_ALLOW_ALL
        -->
        <!--(SSO Client 的默认 java 实现中)authenticator 表示向 sso server 提供用户验证功能的认证类，必须是 j.app.sso.Authenticator 的子类-->
```

```
        <login-agent avail="true" for-other-clients="_ALLOW_ALL"
authenticator="j.app.sso.AuthenticatorImpl"
interface="ssoclient.handler?request=ssologinagent"/>

        <!--SSO Client 的默认 java 实现中，用来在 Client 加载用户信息的类（必须是
j.app.sso.User 的子类）-->
        <user-class>j.app.sso.UserImpl</user-class>

        <!--自定义业务相关属性，可多个（比如下面的 syn-interface 表示数据同步接口）-
->
        <property key="syn-interface" value="syn.handler?request=syn"/>
    </client>

    <client isssoserver="false">
        <id>pay</id>
        <name>支付中心</name>
        <url default="true">http://pay.vselected.com/</url>
        <login-page>/index.jhtml</login-page>
        <home-page>/index.jhtml</home-page>
        <passport>de6ced-1304bcb8d25-80f8cac1e092ca14301921951cf77f91</passport>
        <login-interface>ssoclient.handler?request=ssologin</login-interface>
        <logout-interface>ssoclient.handler?request=ssologout</logout-interface>
        <login-agent avail="true" for-other-clients="_ALLOW_ALL"
authenticator="j.pay.user.JPayAuthenticator"
interface="ssoclient.handler?request=ssologinagent"/>

        <user-class>j.pay.user.JPayUser</user-class>
    </client>

    <client isssoserver="false">
        <id>shop</id>
        <name>商城</name>
        <url default="true">http://www.vselected.com/</url>
        <url default="false">http://vselected.com/</url>
        <login-page>/index.jhtml</login-page>
        <home-page>/index.jhtml</home-page>
        <passport>de6ced-1304bcb8d25-80f8cac1e092ca14301921951cf77f91</passport>
        <login-interface>ssoclient.handler?request=ssologin</login-interface>
        <logout-interface>ssoclient.handler?request=ssologout</logout-interface>
        <login-agent avail="true" for-other-clients="_ALLOW_ALL" authenticator=""
interface="ssoclient.handler?request=ssologinagent"/>

        <user-class>j.shop.user.JShopUser</user-class>
    </client>
```

```
</sso>
```

用户登录认证类 j.app.sso. Authenticator

当 SSO Server 进行用户身份验证时，如果是在 Server 端进行统一验证则调用 Server 端配置的验证类（Authenticator 的子类），否则访问指定 SSO Client 的登录验证接口，这时该 SSO Client 会调用配置的验证类，然后将验证结果返回给 SSO Server。

一个简单的实现类如下：

```
package j.app.sso;

import j.app.Constants;
import j.log.Logger;
import j.sys.AppConfig;
import j.sys.SysUtil;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

/**
 *
 * @author JFramework
 *
 */
public class AuthenticatorImpl implements Authenticator{
	private static final long serialVersionUID = 1L;
	private static Logger log=Logger.create(AuthenticatorImpl.class);

	/*
	 *	(non-Javadoc)
	 *	@see	j.framework.sso.Authenticator#login(javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpSession)
	 */
	public LoginResult login(HttpServletRequest request, HttpSession session) throws Exception {
		/*
		 * 登录流程
		 * 1，判断验证码是否正确
		 * 2，登录信息是否完整
		 * 3，用户是否存在（实际应用中可能需要判断用户状态是否有效等）
		 * 4，密码是否正确
		 */
		String userId=SysUtil.getHttpParameter(request,Constants.SSO_USER_ID);
		String userPwd=SysUtil.getHttpParameter(request,Constants.SSO_USER_PWD);
```

```
        LoginResult result=new LoginResult();
        result.setUserId(userId);

        //2，登录信息是否完整
        if(userId==null||userPwd==null){
            result.setResult(LoginResult.RESULT_BAD_REQUEST);
            return result;
        }

        userId=userId.toLowerCase().intern();
        synchronized(userId){
            User user=null;
            try{
                if(userId.equals(AppConfig.getPara("TEST","user-id"))
                        ||userId.equals(AppConfig.getPara("TEST","admin-id"))){
                    user=new UserImpl();
                    user.setUserId(userId);
                }

                if(user==null){//3，用户不存在
                    result.setResult(LoginResult.RESULT_USER_NOT_EXISTS);
                }else{
                    String pwd="";
                    if(userId.equals(AppConfig.getPara("TEST","user-id"))){
                        pwd=AppConfig.getPara("TEST","user-pwd");
                    }else{
                        pwd=AppConfig.getPara("TEST","admin-pwd");
                    }

                    if(pwd.equals(userPwd)){//登录成功
                        result.setResult(LoginResult.RESULT_PASSED);
                    }else{//密码不对
                        result.setResult(LoginResult.RESULT_PASSWORD_INCORRECT);
                    }
                }
            }catch(Exception ex){
                log.log(ex,Logger.LEVEL_ERROR);
                result.setResult(LoginResult.RESULT_ERROR);
            }
        }
        return result;
    }
```

```
    /*
     * (non-Javadoc)
     * @see j.framework.sso.Authenticator#logout(javax.servlet.http.HttpSession)
     */
    public void logout(HttpSession session) throws Exception {
    }
}
```

与业务相关的用户对象 j.app.sso. User

当 SSO Client 首次登入用户时，会加载与业务相关的用户信息和用户所具有的角色（当然，可以什么都不做），角色用于访问权限控制，权限控制系统根据用户具有的角色确定其能否请求某个 url 或 action。一个简单的用户实现类如下：

```
package j.app.sso;

import j.app.permission.Role;
import j.sys.AppConfig;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class UserImpl extends User{
    private static final long serialVersionUID = 1L;
    protected String userId;
    protected String userName;

    /**
     * constructor
     *
     */
    public UserImpl() {
        super();
    }

    /*
     *    (non-Javadoc)
     * @see j.framework.sso.User#getUserId()
     */
    public String getUserId() {
        return this.userId;
    }
```

```
	/*
	 *	(non-Javadoc)
	 * @see j.app.sso.User#setUserId(java.lang.String)
	 */
	public void setUserId(String userId) {
		this.userId=userId;
	}

	/*
	 *	(non-Javadoc)
	 * @see j.app.sso.User#getUserName()
	 */
	public String getUserName() {
		return this.userName;
	}

	/*
	 *	(non-Javadoc)
	 * @see j.app.sso.User#setUserName(java.lang.String)
	 */
	public void setUserName(String userName) {
		this.userName=userName;
	}

	/*
	 *	(non-Javadoc)
	 *		@see		j.app.sso.User#load(javax.servlet.http.HttpSession,
javax.servlet.http.HttpServletRequest)
	 */
	public boolean load(HttpSession _session,HttpServletRequest _request) throws Exception {
		loadRoles();
		return true;
	}

	/**
	 *
	 * @throws Exception
	 */
	public void loadRoles() throws Exception {
		if(this.userId.equals(AppConfig.getPara("TEST","user-id"))){
			this.roles.add(Role.getInstance("ROLE_USER"));
		}else{
			this.roles.add(Role.getInstance("ROLE_USER"));
			this.roles.add(Role.getInstance("ROLE_ADMIN"));
```

```
        }
    }
}
```

权限控制配置文件 permission.xml

权限配置文件 permission.xml 格式如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <!--用户权限不够时返回的默认页面-->
    <no-permission-page>/common/forbidden.htm</no-permission-page>

    <!--需要具备某角色权限（不一定需要登录）才能访问的 URL-->
    <!--match 表示 url 匹配该字符串时需要执行该项权限验证，可使用通配符*，代表一个或多个任意字符-->
    <!--roles 表示那些角色可访问该资源，多个角色用|分隔-->
    <!--no-permission-page 表示无权限时转向的页面，不配置表示转向默认的 no-permission-page-->
    <!--login-page 表示无权限时转向的页面，不配置表示使用 sso 中的配置-->
    <urls>
        <!--
        <url pattern="/user/" roles="ROLE_USER|ROLE_ADMIN" no-permission-page="/common/forbidden.html">
            <exclude>/user/reg.html</exclude>
        </url>
        <url pattern="/admin/" roles="ROLE_ADMIN" no-permission-page="/common/forbidden.html" login-page="/sso/demo/sso_login.htm"/>
        -->
    </urls>

    <!--需要具备某角色权限（不一定需要登录）才能访问的 Action-->
    <!--id 不填表示 path 下所有 action 都使用该规则-->
    <!--roles 表示那些角色可访问该资源，多个角色用|分隔-->
    <!--no-permission-page 表示无权限时转向的页面，不配置表示转向默认的 no-permission-page-->
    <!--login-page 表示无权限时转向的页面，不配置表示使用 sso 中的配置-->
    <actions>
        <action path="/ssoserver" id="ssoupdate" roles="only_sso_communication" no-permission-page="/common/forbidden.html"/>
        <action path="/ssoserver" id="ssologoutuser" roles="only_sso_communication" no-permission-page="/common/forbidden.html"/>
        <action path="/ssoserver" id="ssoaddurl" roles="only_sso_communication" no-permission-page="/common/forbidden.html"/>
```

```
        <action path="/ssoserver" id="ssodelurl" roles="only_sso_communication" no-permission-page="/common/forbidden.html"/>
        <action path="/ssoserver" id="ssologinauto" roles="only_sso_communication" no-permission-page="/common/forbidden.html"/>
        <action path="/AdminTypeOfService" roles="ROLE_SYS|ROLE_ADMIN" no-permission-page="/common/forbidden.html"/>
        <action path="/AdminStaff" roles="ROLE_SYS|ROLE_ADMIN" no-permission-page="/common/forbidden.html"/>
    </actions>

    <!--应用之间进行通信的 passport(持有效通行证的请求不需要进行权限认证)-->
    <passports>db0[)/3,31^.^Zde%3/c2cY`1e-92[a,,1(072.5,.Z_/6^31
dckswasafaswdfsfdfsffdsxdaspcksxxscdaspxsddzsdfzsxfzsdxvsaxvsaascdsdksxcdszffsfcpsccvscxfspfaskdaspxwsxxzszccsvdzsfxpszaszdaskfcskdsacfspdaswczsvwszxasadwskccsvcfsfpswxzsvfxswfdskxcscfk</passports>
</root>
```

权限控制之 filter 配置（j.app.sso.SSOClient）

在 web.xml 中配置，如下：

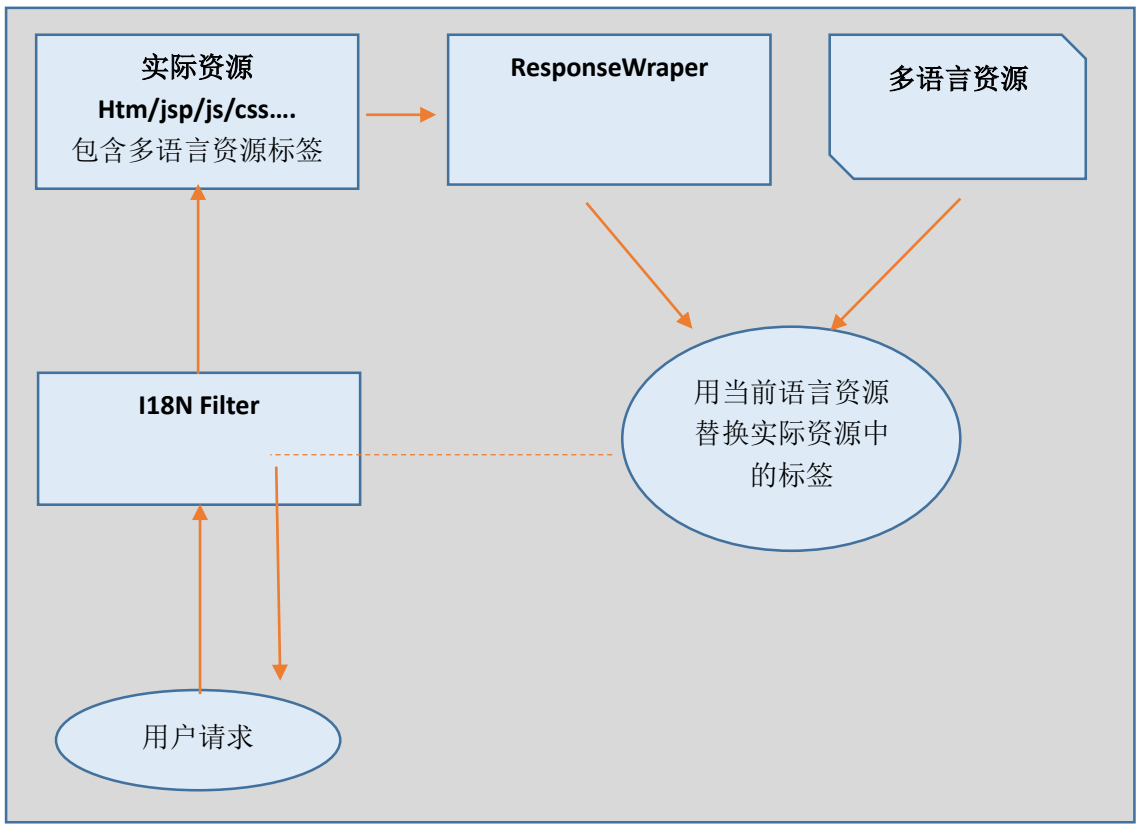
```
    <filter>
        <filter-name>GateKeeper</filter-name>
        <filter-class>j.app.sso.SSOClient</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>GateKeeper</filter-name>
        <url-pattern>*.jsp</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>GateKeeper</filter-name>
        <url-pattern>*.htm</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>GateKeeper</filter-name>
        <url-pattern>*.html</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>GateKeeper</filter-name>
        <url-pattern>*.jhtml</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>GateKeeper</filter-name>
```

```
        <url-pattern>*.handler</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>GateKeeper</filter-name>
        <url-pattern>*.sql</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>GateKeeper</filter-name>
        <url-pattern>/</url-pattern>
    </filter-mapping>
```

多语言框架

实现原理

如下图所示，I18NFilter（处理多语言的 Filter）拦截用户的请求，根据配置判断该请求是否需要进行多语言转换，I18NFilter 将用户请求 forward 到 ResponseWraper，即不是直接返回给用户，而是将返回内容输出到 ResponseWrapper（详见 servlet 相关的技术细节），I18Nfilter 获取输出到 ResponseWrapper 中的内容，并将其中的多语言标签替换成当前语言资源，然后在通过 response.print 类似方法返回给用户。



方案优点

1， 不需要模板之类的机制，直接在文件中写如多语言标签，不管是 jsp、css、js 都可以，不需要经过显式的转换和处理，随时更新随时生效。
2， 多语言资源文件随时编辑、即时生效，可动态增删语言，增删资源。

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <enabled>true</enabled>

    <!--多语言配置-->
    <languages default="zh-cn">
        <language code="zh-tw" name="中文（繁体）"/>
        <language code="zh-cn" name="中文（简体）"/>
        <language code="en-us" name="English(USA)"/>
    </languages>

    <!--多种语言文本的存放文件-->
    <modules>
        <module remark="js 中的文字">string[js].xml</module>
        <module remark="用户中心">string[1].xml</module>
    </modules>

    <!--哪些 url 需要进行 i18n 处理-->
    <urls>
        <url extension="" pattern="/" match="equals">
        </url>
        <url extension=".htm" pattern="/help/*.htm" match="alike">
            <exclude>tech.htm</exclude>
        </url>
        <url extension=".handler" pattern="/AdminUser.handler" match="alike">
            <exclude>tech.htm</exclude>
        </url>
    </urls>
</root>
```

多语言资源

为方便管理和识别，资源分组管理，系统根据分组名和 key 来寻找多语言资源，当分组名为空时表示是全局资源。比如配置中的 string[js].xml 内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <!-- group 为分组，可自由分组，以好管理、好识别为原则，分组名不能使用逗号，当找不到与指定分组名、key 对应的资源时，尝试查找全局资源-->
    <group name="" desc="全局资源">
        <!-- key 不要使用逗号 -->
        <string key="您好">
            <language code="zh-cn">您好</language>
            <language code="zh-tw">您好</language>
            <language code="en-us">HELLO</language>
        </string>
    </group>
    <group name="js" desc="">
        <string key="notice">
            <language code="zh-cn">提示</language>
            <language code="zh-tw">提示</language>
            <language code="en-us">Notice</language>
        </string>
        <string key="close">
            <language code="zh-cn">关闭</language>
            <language code="zh-tw">關閉</language>
            <language code="en-us">Close</language>
        </string>
        <string key="clear">
            <language code="zh-cn">清除</language>
            <language code="zh-tw">清除</language>
            <language code="en-us">Clear</language>
        </string>
        <string key="wait">
            <language code="zh-cn">请稍候...</language>
            <language code="zh-tw">請稍候...</language>
            <language code="en-us">Please wait...</language>
        </string>

        <string key="time">
            <language code="zh-cn">时间</language>
            <language code="zh-tw">時間</language>
            <language code="en-us">Time</language>
        </string>
```

```
        <string key="请选择省份或国家">
            <language code="zh-cn">请选择省份或国家</language>
            <language code="zh-tw">請選擇省份或國家</language>
            <language code="en-us">Select Province Or Country</language>
        </string>
        <string key="请选择城市">
            <language code="zh-cn">请选择城市</language>
            <language code="zh-tw">請選擇城市</language>
            <language code="en-us">Select City</language>
        </string>
        <string key="请选择区县">
            <language code="zh-cn">请选择区县</language>
            <language code="zh-tw">請選擇區縣</language>
            <language code="en-us">Select County</language>
        </string>
        <string key="请选择乡镇/街道">
            <language code="zh-cn">请选择乡镇/街道</language>
            <language code="zh-tw">請選擇鄉鎮/街道</language>
            <language code="en-us">Select Street</language>
        </string>
        <string key="国内">
            <language code="zh-cn">国内</language>
            <language code="zh-tw">國內</language>
            <language code="en-us">China</language>
        </string>
        <string key="全球">
            <language code="zh-cn">全球</language>
            <language code="zh-tw">全球</language>
            <language code="en-us">Worldwide</language>
        </string>
    </group>
</root>
```

全局资源：I{key}　　如 I{您好 }

分组资源：I{group,key}　　如 I{js,请选择城市}

当前访问页的资源：在 key 前加一点，如 I{.备份数据}，这是为了避免编写代码时候需要去记忆、查找资源分组名——用当前资源的网址（requestURI）作为 group 名，如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <group name="/manage/_index.jhtml" desc="首页">
        <string key="备份数据">
```

```
            <language code="zh-cn">备份数据</language>
            <language code="zh-tw">備份數據</language>
            <language code="en-us">Backup Data</language>
        </string>
        <string key="更新文件">
            <language code="zh-cn">更新文件</language>
            <language code="zh-tw">更新文件</language>
            <language code="en-us">Update File</language>
        </string>
</root>
```

你可能想不到的用法

假设某个页面的英文版和中文版背景色不同，那么我们可以这样做：
Css 中如：background-color:I{bg,color};
对应资源：

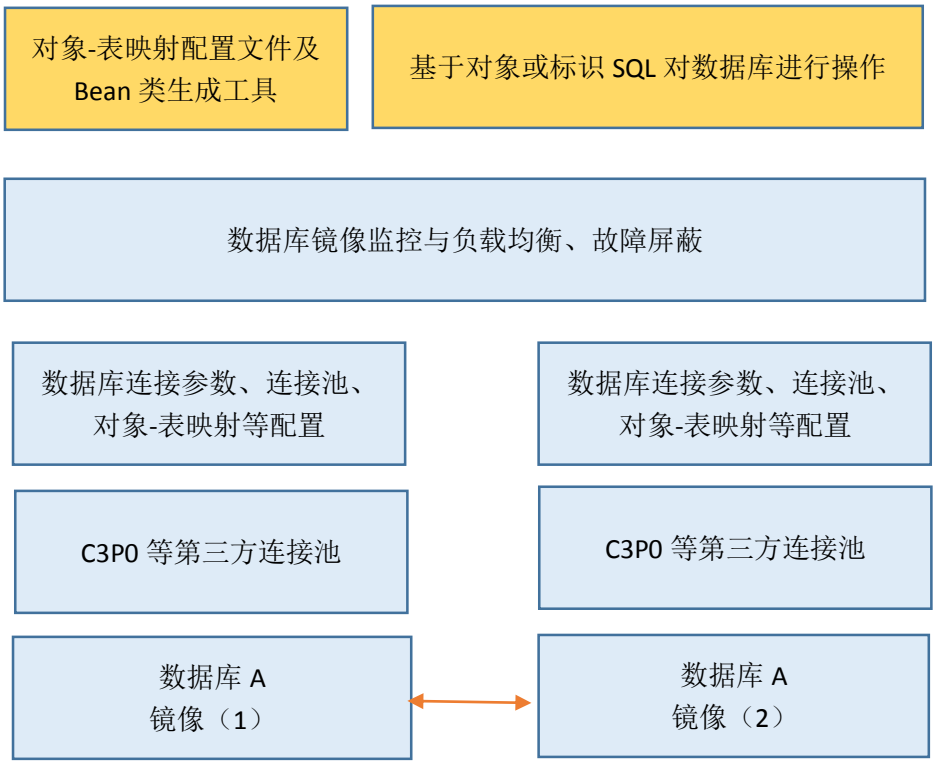
```
<group name=”bg” desc=””>
    <string key=”color”>
        <language code="zh-cn">#eee</language>
        <language code="zh-tw">#ddd</language>
        <language code="en-us">#ccc</language>
    </string>
</group>
```

类似的，不同语言使用不同 logo，不同背景图片等，也可以如此处理。

数据库处理模块（DAO）

架构简图

说明：理论上，这里并不关注各数据库镜像间如何进行数据同步，这应该是别的解决方案去做的事，架构设计应该关注本层次的问题。但实际中，我们可能使用mysql的semisynchronized Replication（半同步复制）进行双向同步，但由于做不到实时同步，所以对实时性要求高的数据，可用共享缓存进行操作：一个应用节点更新数据成功后立即更新缓存，所有应用节点都从缓存读取数据，这样即使数据同步稍有延时也不影响数据一致性。



配置文件 JDAO.xml

如下所示，大部分很好理解，其中 database 的 name 必须与实际数据库名保持一致，service-channel 是指定通过数据库主键自增服务自动生成数据库主键时使用 http 还是 rmi 通信方式（详见分布式服务）。

ignoreColWhileUpdateViaBean 表示只能直接通过 sql 语句更新，不能通过对象进行操作，已避免错误覆盖。

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <database name="jframework" desc="框架核心数据库（Mysql）" service-channel="http">
        <mirror uuid="1" config="config/datasource.jframework.xml" read="true" insert="true"
update="true"

priority="2"></mirror>
        <mirror uuid="2" config="config/datasource.jframework.xml" read="true" insert="false"
update="false"

priority="1"></mirror>

        <!-- 保证安全性和篡改 -->
        <ignoreColWhileUpdateViaBean>table,column1</ignoreColWhileUpdateViaBean>
        <ignoreColWhileUpdateViaBean>table,column2</ignoreColWhileUpdateViaBean>

        <!-- 某些表使用其它表的元数据 -->
        <meta selector="table*">table</meta>

        <!-- 基于文件系统的长字段库,未实现 -->
        <fsstore table="same_table1" column="some_column" load-on-query="false"/>
        <fsstore table="same_table2" column="some_column" load-on-query="false"/>
    </database>
    <database name="Region" desc="行政区划数据库（SQLite）" service-channel="http">
        <mirror uuid="3" config="config/datasource.region.xml"></mirror>
    </database>
    <database name="IP" desc="IP 数据库（SQLite）" service-channel="http">
        <mirror uuid="4" config="config/datasource.ip.xml"></mirror>
    </database>
    <database name="jshop" desc="" service-channel="http">
        <mirror uuid="5" config="config/datasource.jshop.xml"></mirror>
    </database>
    <database name="jpayment" desc="" service-channel="http">
        <mirror uuid="6" config="config/datasource.jpay.xml"></mirror>
```

```
    </database>
</root>
```

数据库连接与对象-表映射配置

对每一个数据库都会生成一个连接配置文件和一组与各个表对应的 bean 类和字段映射文件。连接配置文件，如下：

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- <!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">-->
<hibernate-configuration>
    <session-factory>
        <!-- <property name="plugin">j.dao.DAOPlugin4PrintSQL</property> -->
        <property name="dialect">j.dao.dialect.MysqlDialect</property>
        <!--                                                    <property name="connection.provider_class">j.dao.connection.DriverManagerConnectionProvider</property> -->
        <property name="connection.provider_class">j.dao.connection.C3P0ConnectionProvider</property>
        <property name="connection.pool_size">15</property>

        <property name="c3p0.initialPoolSize">15</property>
        <property name="c3p0.maxPoolSize">300</property>

        <!-- <property name="dbcp.maxActive">300</property> -->

        <!-- <property name="connection.datasource">java:comp/env/jdbc/test</property> -->

        <!-- <property name="proxool.xml">j/db/proxool.xml</property> -->
        <!-- <property name="proxool.properties">j/db/proxool.properties</property> -->
        <!-- <property name="proxool.pool_alias">DBPool</property> -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost/jframework?useUnicode=true&amp;characterEncoding=utf-8</property>
        <property name="connection.username">root</property>
        <property name="connection.password">20081016</property>
        <property name="show_sql">false</property>
        <mapping resource="j/db/JactionLog.xml" />
        <mapping resource="j/db/Japp.xml" />
        <mapping resource="j/db/JappMirror.xml" />
        <mapping resource="j/db/Jappserver.xml" />
```

```
        <mapping resource="j/db/Jblacklist.xml" />
        <mapping resource="j/db/Jcity.xml" />
        <mapping resource="j/db/Jcontinent.xml" />
        <mapping resource="j/db/Jcountry.xml" />
        <mapping resource="j/db/Jcounty.xml" />
        <mapping resource="j/db/Jdatabase.xml" />
        <mapping resource="j/db/Jdb.xml" />
        <mapping resource="j/db/JdbMirror.xml" />
        <mapping resource="j/db/JfsTask.xml" />
        <mapping resource="j/db/Jip.xml" />
        <mapping resource="j/db/Jlog.xml" />
        <mapping resource="j/db/Jprovince.xml" />
        <mapping resource="j/db/Jserver.xml" />
        <mapping resource="j/db/Jservice.xml" />
        <mapping resource="j/db/JserviceMirror.xml" />
        <mapping resource="j/db/JserviceRouter.xml" />
        <mapping resource="j/db/Jtest.xml" />
        <mapping resource="j/db/Jwebserver.xml" />
        <mapping resource="j/db/JwebserverToAppserver.xml" />
        <mapping resource="j/db/Jzone.xml" />
    </session-factory>
</hibernate-configuration>
```

映射文件，比如 Jzone.xml：

```
<?xml version="1.0"?>
<!-- <!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >-->
<hibernate-mapping>
<class name="j.db.Jzone"    table="j_zone">
<id name="zoneId" type="java.lang.String" column="ZONE_ID">
    <generator class="assigned" />
</id>

<property name="zoneId" type="java.lang.String" column="ZONE_ID" not-null="true"
length="16"/>

<property name="countyId" type="java.lang.String" column="COUNTY_ID" not-null="false"
length="16"/>

<property name="cityId" type="java.lang.String" column="CITY_ID" not-null="false" length="16"/>

<property name="provinceId" type="java.lang.String" column="PROVINCE_ID" not-null="false"
length="16"/>
```

```
<property name="countryId" type="java.lang.String" column="COUNTRY_ID" not-null="false" length="16"/>

<property name="continentId" type="java.lang.String" column="CONTINENT_ID" not-null="true" length="16"/>

<property name="zoneName" type="java.lang.String" column="ZONE_NAME" not-null="false" length="60"/>

<property name="zoneNameTw" type="java.lang.String" column="ZONE_NAME_TW" not-null="false" length="60"/>

<property name="zoneNameEn" type="java.lang.String" column="ZONE_NAME_EN" not-null="false" length="150"/>

<property name="areaCode" type="java.lang.String" column="AREA_CODE" not-null="false" length="8"/>

<property name="timeZone" type="java.lang.Double" column="TIME_ZONE" not-null="false" length="3"/>

<property name="postalCode" type="java.lang.String" column="POSTAL_CODE" not-null="false" length="16"/>

<property name="isAvail" type="java.lang.String" column="IS_AVAIL" not-null="false" length="1"/>

</class>
</hibernate-mapping>
```

Bean 类，比如 Jzone.java：

```
/*
 * Created on 2014-04-23
 *
 */
package j.db;


import java.io.Serializable;


/**
 * @author JFramework-BeanGenerator
```

```
 *
 */
public class Jzone implements Serializable{

	private java.lang.String zoneId;
	private java.lang.String countyId;
	private java.lang.String cityId;
	private java.lang.String provinceId;
	private java.lang.String countryId;
	private java.lang.String continentId;
	private java.lang.String zoneName;
	private java.lang.String zoneNameTw;
	private java.lang.String zoneNameEn;
	private java.lang.String areaCode;
	private java.lang.Double timeZone;
	private java.lang.String postalCode;
	private java.lang.String isAvail;

	public java.lang.String getZoneId(){
		return this.zoneId;
	}
	public void setZoneId(java.lang.String zoneId){
		this.zoneId=zoneId;
	}

	public java.lang.String getCountyId(){
		return this.countyId;
	}
	public void setCountyId(java.lang.String countyId){
		this.countyId=countyId;
	}

	public java.lang.String getCityId(){
		return this.cityId;
	}
	public void setCityId(java.lang.String cityId){
		this.cityId=cityId;
	}

	public java.lang.String getProvinceId(){
		return this.provinceId;
	}
	public void setProvinceId(java.lang.String provinceId){
		this.provinceId=provinceId;
```

```
    }

    public java.lang.String getCountryId(){
        return this.countryId;
    }
    public void setCountryId(java.lang.String countryId){
        this.countryId=countryId;
    }

    public java.lang.String getContinentId(){
        return this.continentId;
    }
    public void setContinentId(java.lang.String continentId){
        this.continentId=continentId;
    }

    public java.lang.String getZoneName(){
        return this.zoneName;
    }
    public void setZoneName(java.lang.String zoneName){
        this.zoneName=zoneName;
    }

    public java.lang.String getZoneNameTw(){
        return this.zoneNameTw;
    }
    public void setZoneNameTw(java.lang.String zoneNameTw){
        this.zoneNameTw=zoneNameTw;
    }

    public java.lang.String getZoneNameEn(){
        return this.zoneNameEn;
    }
    public void setZoneNameEn(java.lang.String zoneNameEn){
        this.zoneNameEn=zoneNameEn;
    }

    public java.lang.String getAreaCode(){
        return this.areaCode;
    }
    public void setAreaCode(java.lang.String areaCode){
        this.areaCode=areaCode;
    }
```

```
    public java.lang.Double getTimeZone(){
        return this.timeZone;
    }
    public void setTimeZone(java.lang.Double timeZone){
        this.timeZone=timeZone;
    }

    public java.lang.String getPostalCode(){
        return this.postalCode;
    }
    public void setPostalCode(java.lang.String postalCode){
        this.postalCode=postalCode;
    }

    public java.lang.String getIsAvail(){
        return this.isAvail;
    }
    public void setIsAvail(java.lang.String isAvail){
        this.isAvail=isAvail;
    }

    public boolean equals(Object obj){
        return super.equals(obj);
    }

    public int hashCode(){
        return super.hashCode();
    }

    public String toString(){
        return super.toString();
    }

}
```

一个简单的数据操作范例

```
DAO dao=null;
        try{
            dao=DB.connect("Region",Region.class); //参数为数据库名、调用者类

            List temp=dao.find("j_zone","");//查询所有记录，参数为表名、查询条件
            for(int i=0;i<temp.size();i++){
```

```
                    Jzone o=(Jzone)temp.get(i);
                    //……
                }
                temp.clear();
                temp=null;

                dao.close();
                dao=null;
            }catch(Exception e){
                log.log(e,Logger.LEVEL_ERROR);
                try{
                    dao.close();
                    dao=null;
                }catch(Exception ex){}
            }
```

表联查

```
List    list=dao.find(new String[]{table1,table2},"table1.id=table2.id");
for(int i=0;i<list.size();i++){
    Object[] objs=(Object[])list.get(i);
    Bean1 bean1=(Bean1)objs[0];
    Bean2 bean1=(Bean2)objs[1];
}
```

查询单条记录

```
SomeBean bean=(SomeBean)dao.findSingle(table,"someId=1");
```

分页查询

List list=dao.find(table,查询条件,10,2);//每页 10 条，第二页

查询（返回 ResultSet）

```
StmtAndRs sr=dao.find(sql);
ResultSet rs=sr.resultSet();
….
```

执行 sql

```
dao.executeSQL(sql);
```

更新记录

```
dao. updateByKeys (bean,new String[]{name});//更新 name 字段等于 bean 中 name 值的数据库记录为 bean 中各字段的值。

dao. updateByKeysIgnoreNulls(bean,new String[]{name});//同上，但忽略 bean 中为 null 的字段

dao. updateByKeys(bean);//根据主键更新
dao. updateByKeys(bean) ;//同上，但忽略 bean 中为 null 的字段
```

更多请参考下面的“所有方法”。

插入记录

```
SomeBean bean;
……
dao.insert(bean);
//dao.insertIfNotExists(bean);
```

更多请参考下面的“所有方法”。

主键自增

```
public String autoIncreaseKey(String tableName,String colName) throws Exception;
```

所有方法

```
package j.dao;

import java.sql.Connection;
import java.util.List;
import java.util.Map;

/**
 *
 * @author JFramework
 *
```

```
 */
public interface DAO{
	public static final String DB_TYPE_MYSQL="MYSQL";
	public static final String DB_TYPE_SQLITE="SQLITE";
	public static final String DB_TYPE_DB2="DB2";
	public static final String DB_TYPE_ORACLE="ORACLE";
	public static final String DB_TYPE_SQLSERVER="SQLSERVER";
	public static final String DB_TYPE_HSQL="HSQL";

	/**
	 *
	 * @return
	 */
	public Mirror getMirror();

	/**
	 *
	 * @return
	 */
	public String getCaller();

	/**
	 *
	 * @param caller
	 */
	public void setCaller(String caller);

	/**
	 *
	 * @return
	 */
	public long getTimeout();

	/**
	 *
	 * @param timeout
	 */
	public void setTimeout(long timeout);

	/**
	 *
	 * @return
	 */
	public void begin();
```

```
    /**
     *
     *
     */
    public void finish();

    /**
     *
     * @return
     */
    public long getLastUsingTime();

    /**
     *
     * @return
     */
    public boolean isUsing();

    /**
     * 提交
     * @throws Exception
     */
    public void commit()throws Exception ;

    /**
     * 回滚
     * @throws Exception
     */
    public void rollback()throws Exception ;

    /**
     * 开始事务
     * @throws Exception
     */
    public void beginTransaction()throws Exception;

    /**
     * 是否处于事务状态
     * @return
     * @throws Exception
     */
    public boolean isInTransaction();
```

```
    /**
     * 关闭
     * @throws Exception
     */
    public void close()throws Exception;

    /**
     * 在任何调用前
     * @throws Exception
     */
    public void beforeAnyInvocation() throws Exception ;

    /**
     * 在任何调用后
     * @throws Exception
     */
    public void afterAnyInvocation() throws Exception ;

    /**
     * 异常发生时候
     * @throws Exception
     */
    public void onException();

    /**
     * 是否已经关闭
     * @return boolean
     */
    public boolean isClosed() ;

    /**
     *
     * @return
     */
    public Connection getConnection();

    /**
     *
     * @return
     */
    public DAOFactory getFactory();
```

```
    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////

    /**
     * 查询
     * @param sql 标准 sql
     * @return see StmtAndRs
     * @throws Exception
     */
    public StmtAndRs find(String sql)throws Exception;

    /**
     * 查询
     * @param sql 标准 sql
     * @param RPP 每页多少条，大于 0 的整数
     * @param PN 第几页，大于 0 的整数
     * @return see StmtAndRs
     * @throws Exception
     */
    public StmtAndRs find(String sql,int RPP,int PN)throws Exception;

    /**
     * get the start(inclusive) to end(exclusive)
     * @param sql
     * @param start
     * @param end
     * @return
     * @throws Exception
     */
    public StmtAndRs findScale(String sql,int start,int end)throws Exception;

    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////

    /**
     *
     * @param sql
     * @param cls
     * @param except 某些不读取的列，格式：{列名 1}{列名 2}
     * @return
     * @throws Exception
     */
    public List find(String sql,Class cls,String except)throws Exception;
```

```
    /**
     *
     * @param sql
     * @param cls
     * @param except 某些不读取的列，格式：{列名 1}{列名 2}
     * @param RPP
     * @param PN
     * @return
     * @throws Exception
     */
    public List find(String sql,Class cls,String except,int RPP,int PN)throws Exception;

    /**
     * get the start(inclusive) to end(exclusive)
     * @param sql
     * @param cls
     * @param except 某些不读取的列，格式：{列名 1}{列名 2}
     * @param start
     * @param end
     * @return
     * @throws Exception
     */
    public List findScale(String sql,Class cls,String except,int start,int end)throws Exception;


    /**
     *
     * @param sql
     * @param cls
     * @param except
     * @return
     * @throws Exception
     */
    public Object findSingle(String sql,Class cls,String except)throws Exception;


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////


    /**
     * 根据表名和查询条件进行单表查询
     * @param tableName 表名，不区分大小写
     * @param condition 标准 sql 的查询条件，不包含"where"，即 where 后的部分
```

```
     * @return List  与表对应的 Bean 的列表
     * @throws Exception
     */
    public List find(String tableName,String condition) throws Exception;

    /**
     *  根据表名和查询条件进行单表查询
     * @param tableName  表名，不区分大小写
     * @param condition  标准 sql 的查询条件，不包含"where"，即 where 后的部分
     * @param RPP  每页多少条，大于 0 的整数
     * @param PN  第几页，大于 0 的整数
     *
     * @return List  与表对应的 Bean 的列表
     * @throws Exception
     */
    public List find(String tableName,String condition,int RPP,int PN) throws Exception;

    /**
     * get the start(inclusive) to end(exclusive)
     * @param tableName
     * @param condition
     * @param start
     * @param end
     * @return
     * @throws Exception
     */
    public List findScale(String tableName,String condition,int start,int end) throws Exception;

    /**
     *  返回符合条件的第一条记录(if exists)
     * @param tableName  表名，不区分大小写
     * @param condition  标准 sql 的查询条件，不包含"where"，即 where 后的部分
     *
     * @return  与表对应的 Bean
     * @throws Exception
     */
    public Object findSingle(String tableName,String condition)throws Exception;


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
```

```
    /**
     *
     * @param tableName
     * @param condition
     * @param cls
     * @return
     * @throws Exception
     */
    public List find(String tableName,String condition,Class cls)throws Exception;


    /**
     *
     * @param tableName
     * @param condition
     * @param cls
     * @param RPP
     * @param PN
     * @return
     * @throws Exception
     */
    public List find(String tableName,String condition,Class cls,int RPP,int PN)throws Exception;


    /**
     *
     * @param tableName
     * @param condition
     * @param cls
     * @param start
     * @param end
     * @return
     * @throws Exception
     */
    public List findScale(String tableName,String condition,Class cls,int start,int end)throws Exception;


    /**
     *
```

```
     * @param tableName
     * @param condition
     * @param cls
     * @return
     * @throws Exception
     */
    public Object findSingle(String tableName,String condition,Class cls)throws Exception;


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////


    /**
     * 根据表名和查询条件进行多表查询
     * @param tableNames 参与联查的表名的数组
     * @param condition 标准 sql 的查询条件，不包含"where"，即 where 后的部分
     *
     * @return List 对象数组（Object[]）的列表，对象数组中的对象是与表对应的对象，
     * 其顺序与 tblNames 中指定的表名顺序一致。例如：
     * List lst=find(new String[]{"TBL_A","TBL_B"},"TBL_A.ID=TBL_B.ID");
     * for(int i=0;i<lst.size();i++){
     *   Object[] objs=(Object[])lst.get(i);
     *   TblA a=(TblA)objs[0];
     *   TblB b=(TblB)objs[1];
     * }
     * @throws Exception
     */
    public List find(String[] tblNames,String condition) throws Exception;


    /**
     * 根据表名和查询条件进行多表查询（翻页）
     * @param tableNames 参与联查的表名的数组
     * @param condition 标准 sql 的查询条件，不包含"where"，即 where 后的部分
     * @param RPP 每页多少条，大于 0 的整数
     * @param PN 第几页，大于 0 的整数
     *
     * @return List 对象数组（Object[]）的列表，对象数组中的对象是与表对应的对象，
     * 其顺序与 tblNames 中指定的表名顺序一致。例如：
     * List lst=find(new String[]{"TBL_A","TBL_B"},"TBL_A.ID=TBL_B.ID");
     * for(int i=0;i<lst.size();i++){
     *   Object[] objs=(Object[])lst.get(i);
     *   TblA a=(TblA)objs[0];
```

```
     *    TblB b=(TblB)objs[1];
     * }
     * @throws Exception
     */
    public List find(String[] tblNames,String condition,int RPP,int PN) throws Exception;


    /**
     * for the numbers of records 0 to n ,get the start(inclusive) to end(exclusive)
     * @param tableNames
     * @param condition
     * @param start
     * @param end
     * @return
     * @throws Exception
     */
    public List findScale(String[] tblNames,String condition,int start,int end) throws Exception;


    /**
     *
     * @param tableNames
     * @param condition
     * @return
     * @throws Exception
     */
    public Object findSingle(String[] tableNames,String condition)throws Exception;


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////


    /**
     *
     * @param tableNames
     * @param CLSs
     * @param condition
     * @return
     * @throws Exception
     */
    public List find(String[] tblNames,Class[] CLSs,String condition) throws Exception;

    /**
```

```
     *
     * @param tableNames
     * @param CLSs
     * @param condition
     * @param RPP
     * @param PN
     * @return
     * @throws Exception
     */
    public List find(String[] tblNames,Class[] CLSs,String condition,int RPP,int PN) throws Exception;


    /**
     *
     * @param tableNames
     * @param CLSs
     * @param condition
     * @param start
     * @param end
     * @return
     * @throws Exception
     */
    public List findScale(String[] tblNames,Class[] CLSs,String condition,int start,int end) throws Exception;



    /**
     *
     * @param tableNames
     * @param CLSs
     * @param condition
     * @return
     * @throws Exception
     */
    public Object findSingle(String[] tblNames,Class[] CLSs,String condition) throws Exception;


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////


```

```
    /**
     * 插入对象
     * @param vo 与数据表记录对应的 value object
     * @throws Exception
     */
    public void insert(Object vo)throws Exception;
    public void insertIfNotExists(Object vo)throws Exception;
    public void insertIfNotExists(Object vo,String[] conditionKeys)throws Exception;


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////


    /**
     * 插入对象
     * @param vo 与数据表记录对应的 value object
     * @throws Exception
     */
    public void insert(String tblName,Object vo)throws Exception;
    public void insertIfNotExists(String tblName,Object vo)throws Exception;
    public void insertIfNotExists(String tblName,Object vo,String[] conditionKeys)throws Exception;


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////

    /**
     * 更新符合条件的记录
     * @param tableName 表名，不区分大小写
     * @param colsBeUpdated 要更新的列（Map），key: 列名（不区分大小写），value：列的值
     * @param condition 标准 sql 的查询条件，不包含"where"，即 where 后的部分
     * @throws Exception
     */
    public void update(String tableName,Map colsBeUpdated,String condition)throws Exception;

    /**
     * 根据 vo 中指定字段（0 个或多个）所组成的条件，将符合条件的记录更新成 vo 所表示的状态（不包括作为条件的自段）
     * @param vo
     * @param conditionKeys 组成的条件的字段名数组，不区分大小写
```

```
     * @throws Exception
     */
    public void updateByKeys(Object vo,String[] conditionKeys)throws Exception;
    public void updateByKeys(Object vo)throws Exception;

    public void updateByKeys(String tblName,Object vo,String[] conditionKeys)throws Exception;
    public void updateByKeys(String tblName,Object vo)throws Exception;

    /**
     * 根据 vo 中指定字段（0 个或多个）所组成的条件，将符合条件的记录更新成 vo 所表示的状态（不包括作为条件的自段和为 null 的字段）
     * @param vo
     * @param conditionKeys 组成的条件的字段名数组，不区分大小写
     * @throws Exception
     */
    public void updateByKeysIgnoreNulls(Object vo,String[] conditionKeys)throws Exception;
    public void updateByKeysIgnoreNulls(Object vo)throws Exception;

    public void updateByKeysIgnoreNulls(String tblName,Object vo,String[] conditionKeys)throws Exception;
    public void updateByKeysIgnoreNulls(String tblName,Object vo)throws Exception;


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////

    /**
     * 执行标准 sql
     * @param sql 标准 SQL
     * @throws Exception
     */
    public void executeSQL(String sql)throws Exception;
    public void executeSQLList(List sqls)throws Exception;
    /**
     *
     * @param sql 标准 SQL
     * @throws Exception
     */
    public void executeBatchSQL(List sqls)throws Exception;


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
```

```
    /**
     * 得到符合指定标准 SQL 的记录条数
     * @param sql 标准、完整的 SQL
     *
     * @return
     * @throws Exception
     */
    public int getRecordCnt(String sql) throws Exception;

    /**
     * 得到指定表名的，符合 condition 所指定条件的记录条数
     * @param tableName 表名，不区分大小写
     * @param condition 标准 sql 的查询条件，不包含"where"，即 where 后的部分
     *
     * @return
     * @throws Exception
     */
    public int getRecordCnt(String tableName,String condition) throws Exception;


    /**
     * 多表联查时，得到符合 condition 所指定条件的记录数
     * @param tableNames 参与联查的表名的数组
     * @param condition 标准 sql 的查询条件，不包含"where"，即 where 后的部分
     *
     * @return
     * @throws Exception
     */
    public int getRecordCnt(String[] tblNames,String condition) throws Exception;

    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////

    /**
     * 得到指定表、指定列的，符合 condition 所指定条件的最小值
     * @param tableName 表名，不区分大小写
     * @param colName 列名
     * @param condition  标准 sql 的查询条件，不包含"where"，即 where 后的部分
     *
     * @return 如果没有最小值，返回空字符串
     * @throws Exception
     */
```

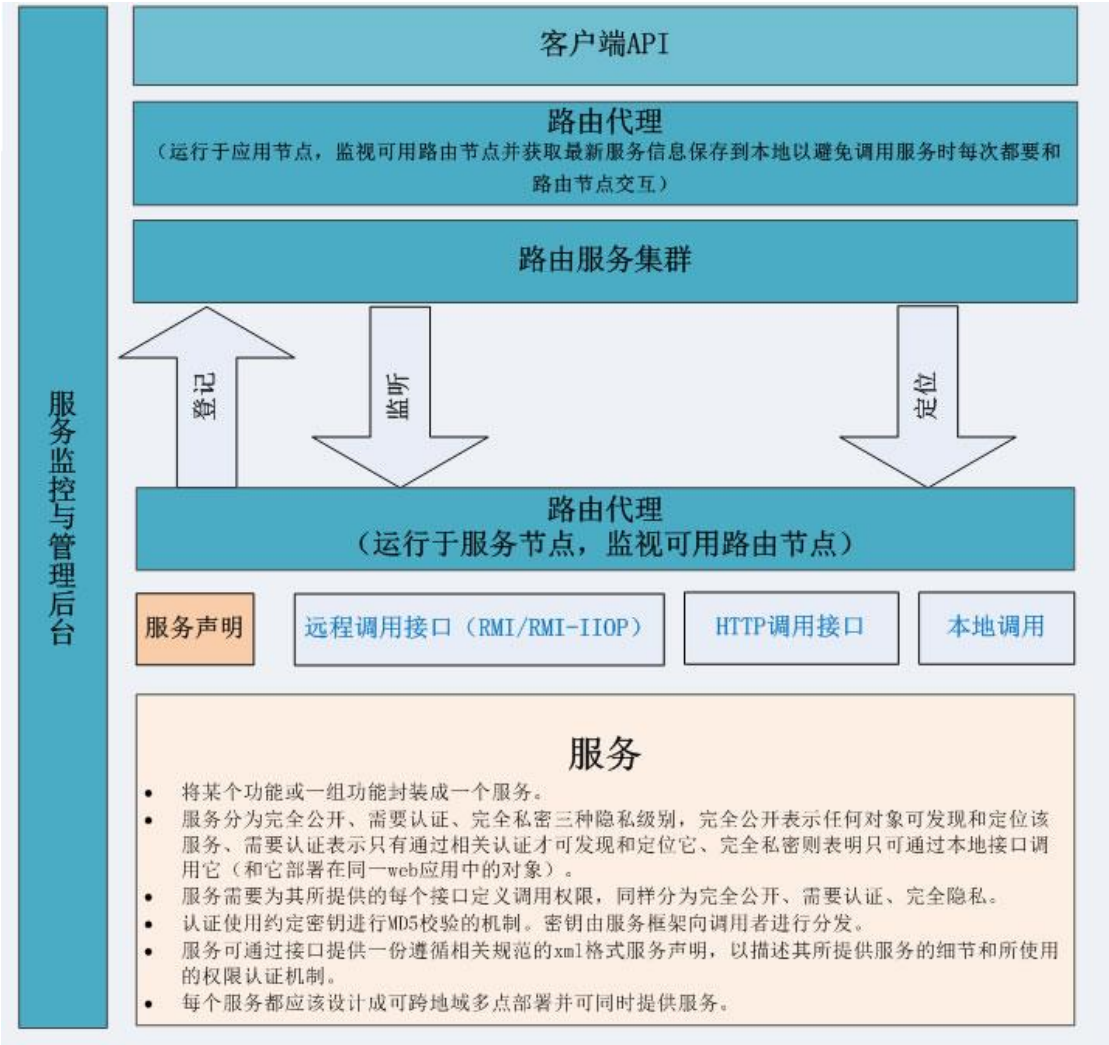
```
    public String getMinValue(String tableName,String colName,String condition) throws Exception;

    /**
     * 得到指定表、指定列的，符合 condition 所指定条件的最小值
     * @param tableName 表名，不区分大小写
     * @param colName 列名
     * @param condition  标准 sql 的查询条件，不包含"where"，即 where 后的部分
     *
     * @return 如果没有最小值，返回空字符串
     * @throws Exception
     */
    public String getMinNumber(String tableName,String colName,String condition) throws Exception;

    /**
     * 得到指定表、指定列的，符合 condition 所指定条件的最大值
     * @param tableName 表名，不区分大小写
     * @param colName 列名
     * @param condition  标准 sql 的查询条件，不包含"where"，即 where 后的部分
     *
     * @return 如果没有最大值，返回空字符串
     * @throws Exception
     */
    public String getMaxValue(String tableName,String colName,String condition) throws Exception;


    /**
     * 得到指定表、指定列的，符合 condition 所指定条件的最大值
     * @param tableName 表名，不区分大小写
     * @param colName 列名
     * @param condition  标准 sql 的查询条件，不包含"where"，即 where 后的部分
     *
     * @return 如果没有最大值，返回空字符串
     * @throws Exception
     */
    public String getMaxNumber(String tableName,String colName,String condition) throws Exception;


    /**
     * 得到指定表、指定列的，符合 condition 所指定条件的值的和
     * @param tableName 表名，不区分大小写
```

```
     * @param colName  列名
     * @param condition    标准 sql 的查询条件，不包含"where"，即 where 后的部分
     *
     * @return  如果没有相关值，返回空字符串
     * @throws Exception
     */
    public String getSum(String tableName,String colName,String condition) throws Exception;

    /**
     *
     * @param tableName
     * @param colName
     * @return
     * @throws Exception
     */
    public String autoIncreaseKey(String tableName,String colName) throws Exception;


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////

    /**
     *  得到数据库编目列表
     * @return List  编目名称的列表
     * @throws Exception
     */
    public List getCatalogs()throws Exception;

    /**
     *  得到数据库模式列表
     * @return List  模式名称的列表
     * @throws Exception
     */
    public List getSchemas()throws Exception;

    /**
     *  得到数据库中的表、视图等对象
     * @param catalog  编目名（可包含通配符?和*，null 表示全部）
     * @param schemaPattern  编目名（可包含通配符?和*，null 表示全部）
     * @param tableNamePattern  表名（可包含通配符?和*，null 表示全部）
     * @param tableNameTypes  取值范围：TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY,
LOCAL TEMPORARY, ALIAS, SYNONYM
     *
     * @return  相关对象名称的列表
```

```
     * @throws Exception
     */
    public List getTables(String catalog, String schemaPattern,String tblPattern, String[] tblTypes)throws Exception;

    /**
     * 得到指定表的列
     * @param tableName 表名，不区分大小写
     *
     * @return Column 对象的列表
     * @throws Exception
     */
    public List getColumns(String tblName)throws Exception;

    /**
     * 得到指定表的主键列
     * @param tableName 表名，不区分大小写
     *
     * @return Column 对象的数组
     * @throws Exception
     */
    public Column[] getPrimaryKeyColumns(String tblName)throws Exception;

    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////

    /**
     *
     *
     */
    public void disablePlugin();
    public void enablePlugin();
    public boolean isPluginEnabled();
}
```

分布式服务框架

架构简图



开发服务第一步：编写服务类

编写接口类

继承 j.service.ServiceBase，每个接口方法需要定义 http 通信版本和 rmi 通信版本，并且都需要抛出 RemoteException。Rmi 接口第一、二个参数必须是：String clientUuid, String md54Service，其中 clientUuid 是调用该服务的应用（称为客户端应用）的 id，md54Service

则是用于验证客户端应用的 md5 校验码，可通过 j.service.client.Client.md54Service(String code,String methodName)获得。http 版必须是 WEB 请求-应答框架中规定的形式，例如：

```
package j.service.hello;

import j.app.webserver.JSession;
import j.service.server.ServiceBase;

import java.rmi.RemoteException;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 * @author JFramework
 *
 */
public interface JHello extends ServiceBase{
	/**
	 *  http 接口实现
	 * @param jsession
	 * @param session
	 * @param request
	 * @param response
	 * @throws Exception
	 */
	public void hello(JSession jsession,HttpSession session,HttpServletRequest request,HttpServletResponse response) throws Exception;

	/**
	 * 每个方法的前两个参数是必须且固定不变的
	 * @param clientUuid  客户节点传过来的它的 uuid
	 * @param md54Service 客户节点传过来的 md5 校验串
	 * @param words
	 * @param times
	 * @return
	 * @throws RemoteException
	 */
	public String hello(String clientUuid, String md54Service,String words,int times) throws RemoteException;
}
```

编写一个抽象类

形式必须如下：public abstract class JHelloAbstract extends ServiceBaseImpl implements JHello,Serializable；如下：

```
package j.service.hello;

import j.service.server.ServiceBaseImpl;

import java.io.Serializable;
import java.rmi.RemoteException;

/**
 *
 * @author JFramework
 *
 */
public abstract class JHelloAbstract extends ServiceBaseImpl implements JHello,Serializable {
	/**
	 *
	 * @throws RemoteException
	 */
	public JHelloAbstract() throws RemoteException {
		super();
	}
}
```

编写实现类

如下：

```
package j.service.hello;

import j.app.webserver.JSession;
import j.service.Constants;
import j.sys.SysUtil;

import java.rmi.RemoteException;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
```

```
 * @author JFramework
 *
 */
public class JHelloImpl extends JHelloAbstract {
	private static final long serialVersionUID = 1L;
	private int counter=0;
	private JHelloWords words;


	/**
	 *
	 * @throws RemoteException
	 */
	public JHelloImpl() throws RemoteException {
		super(); // invoke rmi linking and remote object initialization
	}

	/**
	 *
	 * @param counter
	 */
	public void setCounter(int counter){
		this.counter=counter;
	}

	/**
	 *
	 * @return
	 */
	public int getCounter(){
		return this.counter;
	}

	/**
	 *
	 * @param words
	 */
	public void setWords(JHelloWords words){
		this.words=words;
	}

	/**
	 *
	 * @return
```

```
	*/
	public JHelloWords getWords(){
		return this.words;
	}

	/*
	 *  (non-Javadoc)
	 *  @see j.service.hello.HelloInterface#hello(j.app.webserver.JSession, javax.servlet.http.HttpSession, javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)
	 */
	public void hello(JSession jsession,HttpSession session,HttpServletRequest request,HttpServletResponse response) throws Exception {
		//如果需要认证，必须包含 jservice_client_uuid,jservice_md5_4service 两个参数
		String clientUuid=SysUtil.getHttpParameter(request,Constants.JSERVICE_PARAM_CLIENT_UUID);
		String md54Service=SysUtil.getHttpParameter(request,Constants.JSERVICE_PARAM_MD5_STRING_4SERVICE);

		//每个服务方法都必须以这段代码开头
		try{
			auth(clientUuid,"hello",md54Service);//hello 为本方法的名称
		}catch(RemoteException e){
			jsession.resultString=Constants.AUTH_FAILED;
		}
		//每个服务方法都必须以这段代码开头  end

		String pwords=SysUtil.getHttpParameter(request,"words");
		String ptimes=SysUtil.getHttpParameter(request,"times");
		int times=Integer.parseInt(ptimes);

		counter++;
		System.out.println(counter);
		for(int i=0;i<times;i++){
			System.out.println(pwords);
		}
		jsession.resultString="9999 http got your words:"+pwords;
	}

	/*
	 *  每个方法的前两个参数是必须且固定不变的
	 *  (non-Javadoc)
	 * @see j.service.hello.HelloInterface#hello(java.lang.String, java.lang.String, java.lang.String,
```

```
java.lang.String, int)
     */
    public String hello(String clientUuid, String md54Service,String words,int times) throws RemoteException {
        //每个服务方法都必须以这段代码开头
        try{
            auth(clientUuid,"hello",md54Service);//hello 为本方法的名称
        }catch(RemoteException e){
            throw new RemoteException(Constants.AUTH_FAILED);
        }
        //每个服务方法都必须以这段代码开头  end

        counter++;
        System.out.println(counter);
        for(int i=0;i<times;i++){
            System.out.println(words);
        }
        return "rmi got your words:"+words+" and I said - "+this.words.getWords();
    }
}
```

开发服务第二步：生成 RMI 相关类

rmic -classpath F:\work\JFramework\WebContent\WEB-INF\lib*;. -d F:\work\JFramework\src -keep -nowrite j.service.hello.JHelloAbstract
如果仅使用 http 通信方式，则不需要生成这些类。

开发服务第三步：配置服务

在 Service.server.xml 中配置服务，如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <!-- 一个节点上可运行多个服务 -->
    <!--
    服务框架流程：
    1，服务节点上的 ServiceManager 读取 service.server.xml 配置文件，初始化、启动服务。
    2，在服务节点上运行着与各个路由节点对应的路由代理，服务节点通过路由代理将服务注册到路由节点。
    3，在应用节点上同样运行着与各个路由节点对应的路由代理，路由代理会监测对应路由节点的状态。
    4，应用节点通过 Client 提供的方法获取并获得服务入口——
```

```
        a，RouterManager 通过路由代理提供的信息维护着可用路由节点集合。
        b，Client 是通过 RouterManager 获取服务入口的，RouterManager 通过某种策略使用可用路由节点中的一个，并通过它获得服务入口。
        c，应用节点成功获得服务入口后，再通过 Client 提供的方法调用服务的某个方法，也可通过获得的远程对象获 http 接口直接调。
    *  服务节点、应用节点与路由节点交互时，都是要经过 MD5 校验的。 应用节点获取服务入口，调用服务的方法时，也是需要经过 MD5 校验的。
    -->
    <service>
        <!--
        同一个服务多点部署时，以 code 值来标示同一服务。
        客户端调用服务时，路由器会通过某种策略将同组服务中的一个实例返回给客户应用，以达到均衡负载的效果。
        -->
        <code>HelloService</code>

        <!-- 服务名称 -->
        <name>测试服务</name>

        <!-- 是否需要校验及校验机制 -->
        <privacy>MD5</privacy>

        <property key="j.service.interface" value="j.service.hello.JHello"/>
        <property key="j.service.class" value="j.service.hello.JHelloImpl"/>
        <property key="j.service.relatedHttpHandlerPath" value="/j.service.hello"/>


        <!-- 服务提供的方法，可以多个，可以不配置 -->
        <method>
            <!-- 方法名 -->
            <name>hello</name>

            <!-- 是否需要校验及校验机制 -->
            <privacy>MD5</privacy>
        </method>

        <!-- 当服务或方法需要 MD5 校验时，指明那些应用节点可访问，并配置相应的密钥 -->
        <client uuid="j.service.node.ser:00" name="服务节点" key="any"/>
        <client uuid="j.service.node.app:sso:00" name="应用节点" key="any"/>
        <client uuid="j.service.node.app:pay:00" name="应用节点" key="any"/>
        <client uuid="j.service.node.app:shop:00" name="应用节点" key="any"/>

        <!-- server-uuid 与 j.service.uuid(service.xml 中配置)相同时，表示是当前机器上的节
```

```
点，需要启动 -->
        <node server-uuid="j.service.node.ser:00">
            <!-- 全局范围内运行的唯一服务实例的 ID -->
            <uuid>j.service.HelloService:00</uuid>

            <!-- rmi 或者 rmi-iiop 服务接口配置信息，属性值可为 rmi 或者 rmi-->
            <rmi available="false">
                <property key="java.naming.provider.url"
value="rmi://sso.vselected.com:1099"/>
            </rmi>

            <!-- http 服务接口配置信息 -->
            <http available="true">
                <property key="j.service.http"
value="http://sso.vselected.com/j.service.hello.service"/>
            </http>
        </node>

        <node server-uuid="j.service.node.ser:01">
            <!-- 全局范围内运行的唯一服务实例的 ID -->
            <uuid>j.service.HelloService:01</uuid>

            <!-- rmi 或者 rmi-iiop 服务接口配置信息，属性值可为 rmi 或者 rmi-->
            <rmi available="false">
                <property key="java.naming.provider.url"
value="rmi://fs.vselected.com:1099"/>
            </rmi>

            <!-- http 服务接口配置信息 -->
            <http available="true">
                <property key="j.service.http"
value="http://fs.vselected.com/j.service.hello.service"/>
            </http>
        </node>
    </service>
</root>
```

开发服务第四步：配置 HTTP 通道

服务的 http 通道是基于 web 请求-应答框架来实现的，需要配置相关的 action，如下：

注：其中的`/j.service.router`*是服务路由的* `http` *通道，是必不可少的* 。

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <handler path="/j.service.router" path-pattern=".service" class="j.service.router.JRouterImpl" request-by="request">
        <action id="auth" method="auth" respond-with-string="true"/>
        <action id="heartbeat" method="heartbeat" respond-with-string="true"/>
        <action id="register" method="register" respond-with-string="true"/>
        <action id="unregister" method="unregister" respond-with-string="true"/>
        <action id="service" method="service" respond-with-string="true"/>
    </handler>

    <handler path="/j.service.hello" path-pattern=".service" class="j.service.hello.JHelloImpl" request-by="request">
        <action id="auth" method="auth" respond-with-string="true"/>
        <action id="heartbeat" method="heartbeat" respond-with-string="true"/>
        <action id="hello" method="hello" respond-with-string="true"/>
    </handler>
</root>
```

开发服务第五步：在客户端配置

在 serveice.client.xml 中定义可用的服务，如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <!-- 如果服务存在于本应用中，是否直接调用本地服务 -->
    <!-- true: 是  false:否  random:随机 -->
    <callLocalServiceIfExists>false</callLocalServiceIfExists>

    <!-- 应用节点必须有各服务的基本信息和与服务节点进行 MD5 校验的密钥（如果需要的话） -->
    <service>
```

```
        <code>HelloService</code>
        <name>测试服务</name>
        <property key="j.service.key" value="any"/>
    </service>
</root>
```

开发服务第六步：调用服务

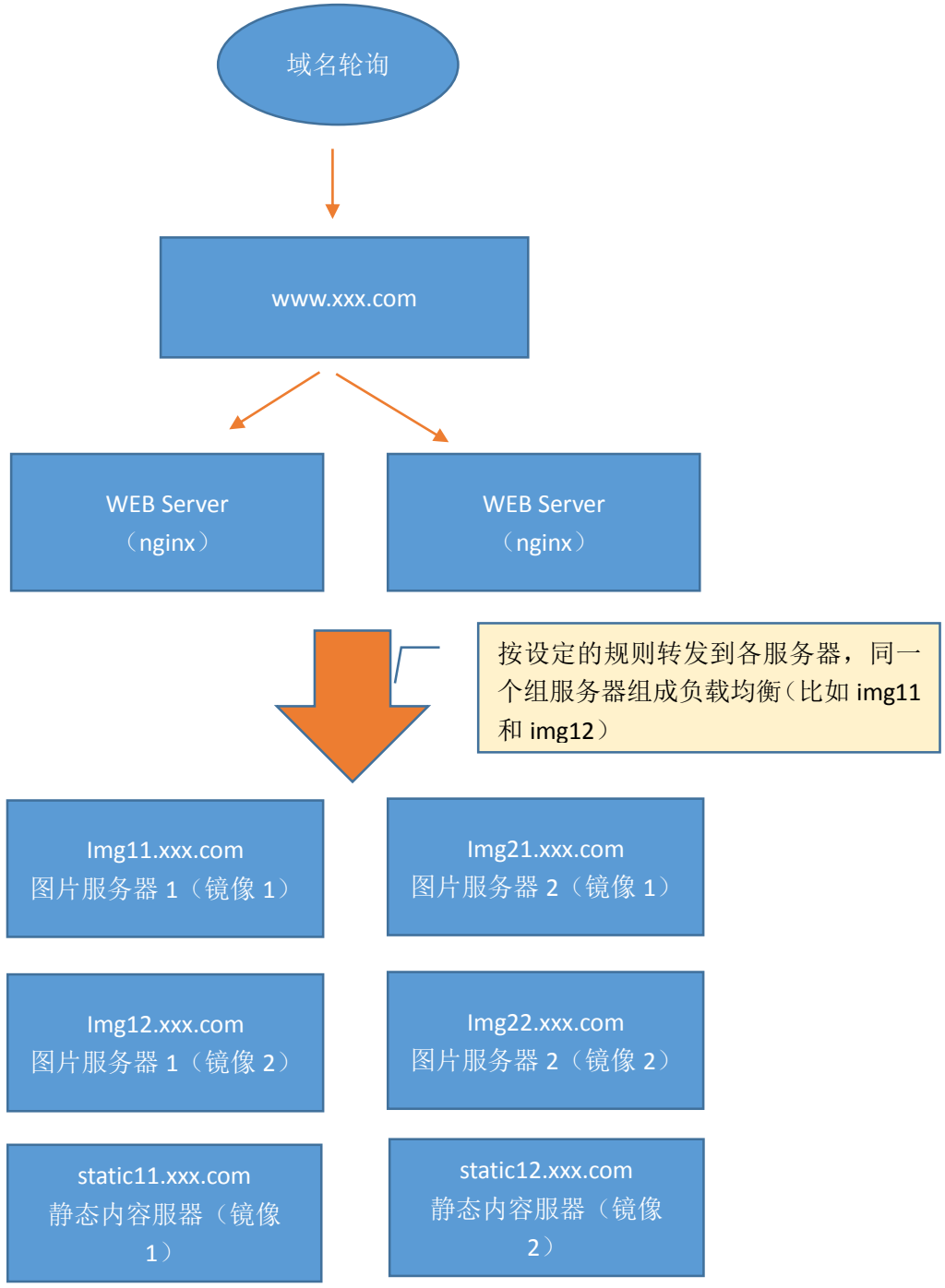
如下：

```
//通过http调用
        try{
            Map paras=new HashMap();
            paras.put("words","uuid -
"+JUtilUUID.genUUID());
            paras.put("times","4");

            String
entrance=Client.httpGetService(null,null,"HelloService",t
rue);
            String
result=Client.httpCallPost(null,null,"HelloService",entra
nce,"hello",paras);
            System.out.println("result2:"+result);
        }catch(Exception e){
            e.printStackTrace();
        }

        //通过rmi调用
        try{
            JHello
h=(JHello)Client.rmiGetService("HelloService",true);
            String
result=h.hello(Manager.getClientNodeUuid(),Client.md54Ser
vice("HelloService","hello"),"hooooooo",2);
            System.out.println("result3:"+result);
        }catch(Exception e){
            e.printStackTrace();
        }
```

分布式文件系统

设计意图

假设有以下架构：



如上图所示，如何将图片、静态文件按照 Nginx 转发的规则存储到各个服务器，并且同一组服务器的镜像节点间保持同步、相互备份，并且这个过程对于应用来说是透明的，应用就像

操作的是本地文件。这就是该分布式文件系统设计的主要动机。

实现原理

文件系统是基于分布式服务框架来实现的，首先系统实现了一个分布式服务，使应用可以透明的操作远程或本地文件，然后根据业务需求，将图片、静态文件分组，每一组文件对应一个分布式文件服务，而每一个服务可以有多个节点，节点间实现文件的同步。

配置文件 JFS.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <!-- 默认通信方式 -->
    <service-channel>http</service-channel>

    <!-- 最大文件（K） -->
    <max-file-size>102400</max-file-size>

    <!-- service-code 为对应的服务编码，service-channel 表示调用服务的方式（http 或 rmi），os 表示目标机器操作系统类型 windows/linux-->
    <!-- 目录分割一律使用/（系统会进行处理）-->
    <mapping service-code="JFS.jshop" service-channel="http" os="windows">
        <rule selector="^JDFS://jshop/[ig]{1}/[\S ]{0,}$" virtual-root="JDFS://jshop/" physical-root="d:/tomcat/webapps/vselected/ROOT/"></rule>
        <rule selector="^JDFS://jshop/goods/[\S ]{0,}$" virtual-root="JDFS://jshop/" physical-root="d:/tomcat/webapps/vselected/ROOT/"></rule>\
    </mapping>
</root>
```

一个 mapping 表示一个分布式文件服务（比如对应 img11 和 img12 这一组），service-code 表示对应的服务编号，service-channel 表示服务通信方式（http 或 rmi）。

Rule 则表示虚拟路径和实际路径之间的映射关系，两个对等节点间物理路径是可以不同的，比如在 windows 是 d:/tomcat/webapps/vselected/ROOT/，linux 上是 /usr/local/tomcat/webapp/vselected/ROOT/，这在两个节点上的 JFS.xml 中填写不同物理路径即可。

一个简单的实例

```
JFile
jfile=JFile.create("JDFS://jshop/i/"+album.getAlbumId()+"/"+photoId+"."+upFile.getFileExt_Saved
().toLowerCase());
jfile.save(file);
jfile=null;
```

分布式缓存系统

设计意图

当一个应用能够跨地域多点部署多个实例时，有几个关键点：文件同步、数据库同步、缓存数据同步，文件同步可用本框架的分布式文件系统解决，数据库同步可结合数据库同步方案+本框架 DAO 来实现，而缓存数据同步、共享则是本模块设计的目标。

再举一个简单的例子，比如应用部署到两台服务器，服务器上都有一个数据库，相互同步，由于数据同步存在延时或事务问题，当我们在两台服务器上分别获取自增主键时，是可能取得重复值的——于是我实现了一个数据库主键自增服务，服务只在初始时获取各表的最大主键值，并保存于共享缓存中，然后应用通过服务从缓存中获取自增主键并将其值加 1。

实现原理与配置文件 Jcache.xml

缓存系统基于分布式服务实现，其原理类似分布式文件系统，不再赘述。

创建一个缓存单元需要指定一个 ID，系统根据 ID 来进行分布式（即配置文件中的 selector），这不同服务器上使用同一个 ID 来操作缓存就是操作的同一个缓存对象（这就实现了共享）。

匹配 selector 的这些 cache 单元是对应一个缓存服务的，每个服务下可以有多个对等节点并实现同步（同步缓存）。

所以，本缓存系统是实现了分布式和节点间备份的。

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <!-- 默认通信方式 -->
    <service-channel>http</service-channel>

    <!-- 对于临时缓存，对象多久没被使用将从缓存中移除 -->
    <cache-timeout>30000</cache-timeout>
```

```
    <!-- 向每个镜像节点同步数据所使用的同步线程数 -->
    <synchronizers>2</synchronizers>

    <!-- service-code 为对应的服务编码，service-channel 表示调用服务的方式（http 或 rmi），os 表示目标机器操作系统类型 windows/linux-->

<mapping selector="^jshop[\S ]{0,}$" service-code="JCache.jshop" service-channel="http" os="windows"></mapping>
    <mapping selector="^jpay[\S ]{0,}$" service-code="JCache.jpay" service-channel="http" os="windows"></mapping>
    <mapping selector="^[\S ]{0,}$" service-code="JCache" service-channel="http" os="windows"></mapping>
</root>
```

常用缓存单元

目前已经实现的缓存单元有 CachedList 和 CachedMap，分别实现对 list 和 map 的缓存。

```
CachedList cache1=new CachedList(“someCache1”);
c ache1.addOne(obj1);

Object obj=cache1.get(index);

CachedMap cache2=new CachedMap(“someCache2”);
cache2.addOne(key,value);
Object obj=cache2.get(new JcacheParams(key));
```

提高缓存操作效率和便捷性

为了让对缓存内对象的获取、查询、更新、移除更方便、高效，系统提供了 JcacheRemover、JcacheUpdater，JcacheFilter 等接口，应用可通过实现这些接口来批量、方便的查询、操作缓存。这些对象都通过 JcacheParams 进行传递。

实用模块： 地域信息 j.tool.region.Region

载入内存的地域信息

Web 应用中，特别是电商，需要详尽、准确的地域信息，为了避免花大量时间去做这项工作，我采集了淘宝网目前使用的地域信息（含邮编、区号），包括从国家、省份、城市、区县、乡镇/街道的各级数据。数据采用嵌入式数据库 SQLite 存储（当然，也可以采用其它数据库），并在启动时载入内存。Region 类提供了查询这些信息的方法。

实现四级地域信心联动的 js

另外，框架提供了一个包含国家、省、市、县和一组 js 方法的 js，已经各个区县下乡镇/街道信息的 js（每个区县对应一个）。Js 提供了从国家到街道 4 级联动（外国与国内省份并列）的方法和邮编、区号查询功能。

地域信息联动范例

Region.js 片段

```
//国家列表
function getCountries(){
	return countries;
}

//区号和邮编
function getCodes(id,areaCodeInput,postalCodeInput){
	var c=codes[id];
	if(areaCodeInput){
		if(areaCodeInput.tagName=='INPUT') areaCodeInput.value=c[0];
		else areaCodeInput.innerHTML=c[0];
	}

	if(postalCodeInput){
		if(postalCodeInput.tagName=='INPUT') postalCodeInput.value=c[1];
		else postalCodeInput.innerHTML=c[1];
	}

	return c;
}

//初始化省份列表
function initProvince(provinceSelector){
```

```
    while(provinceSelector.options&&provinceSelector.options.length>0){
        provinceSelector.options.remove(provinceSelector.options.length-1);
    }
    provinceSelector.options.add(new Option('l{js,请选择省份或国家}',''));

    provinceSelector.options.add(new Option('l{js,国内}',''));
    for(var i=0;i<regions.list.length;i++){
        var province=regions.list[i];
        provinceSelector.options.add(new Option(province[1],province[0]));
    }

    provinceSelector.options.add(new Option('l{js,全球}',''));
    for(var i=0;i<countries.length;i++){
        var country=countries[i];
        if(Lang.l()=='en-us') provinceSelector.options.add(new Option(country[2],country[0]));
        else provinceSelector.options.add(new Option(country[1],country[0]));
    }
}

//选择省份
function changeProvince(provinceSelector,citySelelector,countySelector,zoneSelector){
    if(!provinceSelector||!citySelelector) return;

    while(citySelelector.options&&citySelelector.options.length>0){
        citySelelector.options.remove(citySelelector.options.length-1);
    }
    citySelelector.options.add(new Option('l{js,请选择城市}',''));

    if(countySelector){
        while(countySelector.options&&countySelector.options.length>0){
            countySelector.options.remove(countySelector.options.length-1);
        }
        countySelector.options.add(new Option('l{js,请选择区县}',''));
    }

    if(zoneSelector){
        while(zoneSelector.options&&zoneSelector.options.length>0){
            zoneSelector.options.remove(zoneSelector.options.length-1);
        }
        zoneSelector.options.add(new Option('l{js,请选择乡镇/街道}',''));
    }

    if(provinceSelector.value=='') return;
```

```
    var cities=regions.cities[provinceSelector.value].list;
    for(var i=0;i<cities.length;i++){
        var city=cities[i];
        citySelelector.options.add(new Option(city[1],city[0]));
    }
}

//选择城市
function changeCity(provinceSelector,citySelelector,countySelector,zoneSelector){
    if(!provinceSelector||!citySelelector||!countySelector) return;

    while(countySelector.options&&countySelector.options.length>0){
        countySelector.options.remove(countySelector.options.length-1);
    }
    countySelector.options.add(new Option('I{js,请选择区县}',''));

    if(zoneSelector){
        while(zoneSelector.options&&zoneSelector.options.length>0){
            zoneSelector.options.remove(zoneSelector.options.length-1);
        }
        zoneSelector.options.add(new Option('I{js,请选择乡镇/街道}',''));
    }

    if(citySelelector.value=='') return;

    var counties=regions.cities[provinceSelector.value].counties[citySelelector.value].list;
    for(var i=0;i<counties.length;i++){
        var county=counties[i];
        countySelector.options.add(new Option(county[1],county[0]));
    }
}

//获取街道
var thisZoneSelector=null;
function changeCounty(countySelector,zoneSelector){
    if(!countySelector||!zoneSelector) return;
    thisZoneSelector=zoneSelector;

    while(zoneSelector.options&&zoneSelector.options.length>0){
        zoneSelector.options.remove(zoneSelector.options.length-1);
    }
    zoneSelector.options.add(new Option('I{js,请选择乡镇/街道}',''));

    if(countySelector.value=='') return;
```

```
    zones=new Array();
    loadJS({src:'zones/'+countySelector.value+'.js?_t='+Math.random(),          charset:'utf-8',
callback:showZones});
    //loadJS({src:'/js/region/zones.jhtml?county_id='+countySelector.value,      charset:'utf-8',
callback:showZones});
}

//显示街道
function showZones(){
    for(var i=0;i<zones.length;i++){
        thisZoneSelector.options.add(new Option(zones[i][1],zones[i][0]));
    }
}
```

页面：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<title></title>
<script language="javascript" src="../lib/jframework.js"></script>
<script language="javascript" src="../region/region.js"></script>
</head>
<body>
<select name="province_id" id="province_id"
onchange="changeProvince($('province_id'),$('city_id'),$('county_id'),$('zone_id'));">
</select>
<select name="city_id" id="city_id"
onchange="changeCity($('province_id'),$('city_id'),$('county_id'),$('zone_id'));
getCodes(this.value,$('areaCode'),$('postalCode'));">
</select>
<select name="county_id" id="county_id" onchange="changeCounty($('county_id'),$('zone_id'));
getCodes(this.value,$('areaCode'),$('postalCode'));">
</select>
<select name="zone_id" id="zone_id">
</select>

<br/>
区号    <input type="text" id="areaCode"/>
<br/>
```

```
邮编    <input type="text" id="postalCode"/>
</body>
</html>
<script language="javascript">
initProvince($('province_id'));
changeProvince($('province_id'),$('city_id'),$('county_id'),$('zone_id'));
</script>
```

实用模块： IP 库

j.tool.ip.IP 实现 IP 对应地理信息的查询，本 ip 库采用纯真 ip 库数据。

实用模块：二维码

j.tool.tdc.TDC 提供了二维码生成和解析的方法。

实用模块：基于分布式缓存的验证码分发

在单一的应用中验证码往往基于 session 来实现，但复杂架构下可能是跨应用的，session 机制不再可行，所以基于分布式缓存来实现一个验证码分发服务（在淘宝网这样的大规模下，可能没个功能都需要一个分布式的集群来承载其压力，所以这也是分布式服务架构的重要性）。

实用模块：邮件发送

j.mail.* 实现了简单的邮件发送功能（采用线程池）。

实用模块：短信发送

j.sms.*实现通过短信网关、短信猫等收发短信的功能（暂未实现）。

实用模块：HTTP

基于 commons httpclient 实现的 HTTP 处理模块，最初目的主要用于支撑本人开发的“数据采集解析平台”，后来广泛应用于应用间通信、分布式服务（http 通道）等，是非常重要和基础的一个组件。

实用模块：对象序列化遇反序列化 j.common.JObject

j.common.Jobject 实现了相关功能，分布式服务如何通过 http 来进行对象的传输？就是使用

了 Jobject 提供的功能。

Js 库与 web ui 组件

框架提供了一个包含大部分常用前段操作的 jframework.js，包括窗口、日历、ajax、进度框、跨浏览器剪贴板等等。另外还有图片轮播、图片预览等组件。

结尾

开发规范

在本人工作的团队，我们严禁开发人也擅自将第三方程序、工具加入工程，包括所谓的“框架”、jar 包、js、所谓的 ajax 框架等——不管是使用或不使用第三方功能，都需要经过架构设计人也进行权衡，如果使用则需负责人员将其整合进系统，以符合本团队的风格来使用。

联系作者

QQ 1691969900
微信 1691969900
邮件 xiaojiong@ymail.com

范例工程下载

http://www.vselected.com/downloads/JDemo.zip

源码下载 s

http://www.vselected.com/downloads/JFramework.zip

开发指南下载

http://www.vselected.com/downloads/JFrameworkDoc.zip

Bean 生成工具下载

http://www.vselected.com/downloads/BeanGen.zip