Disco 开发者指南

(v_2.0)

ksmwly@gmail.com

Disco 综述

Disco 是基于 java 技术，用于企业级 Java Web 应用程序快速开发的 MVC 框架。框架设计构思来源于国内众多项目实践，框架的设计及实现借鉴当前主要流行的开源 Web 框架，如 Rails、Struts、JSF、Tapestry 等，吸取其优点及精华，是一个完全由来自的中国开源爱好者开发，文档及注释全部为中文的开源框架。

Disco 由主要由四个部分组成：

1、核心 MVC。Disco 的核心是一个基于模板技术实现的 MVC 框架；他能让我们用非常简洁的代码写基于 Java 的 Web 应用。

2、容器及通用业务逻辑封装。作为一个旨在让基于 Java 的 Web 应用程序开发变得直接、快速、简易的框架，Disco 提供了一个 IoC 容器，并对企业级应用中的一些通用业务逻辑(如分页、查询、DAO 等)进行了抽象及封装，提供了一套可以直接操作、应用企业资源的组件及 API。

3、代码生成引擎及工具。仅仅依靠一个灵活、简易的 MVC 核心引擎还不能最大限度的提高开发速度，因此 Disco 还提供了一个灵活、易用的代码生成引擎及工具，通过使用代码生成引擎，可以快速完成基于 JavaEE 平台的企业级应用程序生成。如数据库添删改查（CRUD）代码生成、自动页面模版生成、自动配置文件管理等。

4、Disco 插件体系，项目中的各种实用功能的扩展，可以灵活地通过基于插件的形式安装到 Disco 中，提供各种针对性的功能。如 ajax 实用插件、代码生成插件等。

Disco 的特点：

1、快速开发支持

Disco 是首要目标是实现基于 JavaEE 的 Web 应用程序快速开发。通过 Disco 的核心 MVC、通用业务逻辑抽象及封装、代码自动生成、插件体系等几个部分有机组合，能实现企业级的 Java Web 应用程序开发。

2、零配置及约定配置

通过配置可以让程序变得更加的灵活、易维护及扩展，然而配置的滥用会造成维护配置文件过于麻烦。因此，Disco 基于尽可能简化配置的原则，实现了零配置支持，同时为了保证系统的灵活性及可扩展性，还提供了很多的约定配置支持。

3、优雅的视图支持，页面及程序完全分离。

Disco 提供了非常优雅的视图支持能力，不但实现了视图页面模板与程序逻辑的完全分离，克服了传统 jsp 页面难于维护的问题，而且还实现了对页面纯天然的支持能力，使得非常适用于企业级应用中的页面制作人员与程序的分工合作。

4、超级 IoC 容器

作为一个主要用于 Java 企业级应用程序开发的框架，Disco 实现了 IoC 容器，提供非常灵活的注入方式，并能支持 Spring、Guice 等异构容器实现。

5、Ajax 支持

Disco 内置了对远程 javascript 脚本调用功能，可以使用 javascript 直接访问服务端的业务组件。另外 Disco 通过使用 prototype.js 及其它一些来自开源社区 ajax 特效工具，提供了丰富的 Ajax 支持。

MVC 部分

MVC 中，M-Model 是指模型层，V-View 是指视图层，C-Controller 是指控制器。作为一个旨在提高开发效率、使 Java 代码与页面模板完全分离，增强系统的可维护性及可扩展性的 MVC 框架，Disco 中同样有这三个基本的概念。在 Disco 应用程序中，Model 层位于系统后台，一般是 POJO 对象，可以通过使用<beans>标签把后台的业务模型对象配置到容器中，让其它层的对象调用。Controller 用于控制转发，Disco 中所有的请求都由 ActionServlet 来负责处理，ActionServlet 再调用相应的模块的 Action，来实现具体的处理，Disco 中的 ActionServlet 与模块 Action 一起共同担当了 Controller 的角色。最后是视图 View，View 是用来显示数据，Model 层的处理结果将交由 View 层来，Disco 中的 View 由单独的模板文件担当，视图模板可以是任何文本格式的信息，如 html、xml、java、sql 等。

请求分发

ActionServlet 解析了请求，并从配置文件中得到了适当的控制器，就将包装好了的请求对象发送给该控制器的 execute 方法。而控制器的作用就在于处理请求并返回一个用户展示处理结果的页面模板对象 Page。

ActionServlet

ActionServlet 是一个前端控制器。用于在第一次请求到来时初始化框架，并接受每一个浏览器请求，解析请求 URL，包装表单属性，并分发给具体的控制器处理。

请求按照 Disco 配置文件中定义或者约定配置来寻找处理器。比如，可以在配置文件中，使用 <module path="/userManage" action="cn.disco.action.userManageAction" defaultPage="list"> 定义了一个请求为 /userManage.java 的请求样式都由 userManageAction 处理，关于控制器更详细的配置在配置详解一节讲述，discoCommand 等在 AbstractCmdAction 中介绍。在按照约定定位控制器的策略中，控制器需要放在 cn.disco.action 包中。比如，我们需要写一个用户登录的 Action，可以在 cn.disco.action 这个包下面，添加一个名为 LoginAction 的类，这样即可通过/login.java 来请求这个控制器。

对于提交的表单，ActionServlet会调用Disco中的核心处理器，将表单中的所有文字属性，都包装在WebForm中的textElement中，而将表单中包含的上传文件使用FileCommUpload包装为`FileItem`并放置在`WebForm`的`FileElement`中。关于表单的处理将在`WebForm`一节详细讲述。

请求 Url

基本请求 url 模式：

当一个请求为 http://xxx.xxx.com/abc.java 的请求到来时，解析得到的控制器名字为 abc。在这里，在框架内部是去寻找 path 属性设置为/abc 的那个 Module，并调用该 Module 对象对应的 IWebAction 对象来处理。而对于开发者，他们只需要知道用来处理/abc.java 这个 path 的 IWebAction 是 AbcAction。下面是一些请求 url 的样式例子：

1、 /module.java?discoCommand=command&name=xxx
2、 /module.java
3、 /module.java?discoCommand=command
4、 /module.java?discoCommand=command&cid=1234
5、/module.java?discoCommand=edit&cid=123455&title=111
6、/module.java?title=测试

第一个样式的意思是请求一个名字为 module 的模块(Module 来)处理请求。如果改 Action 是 AbstractCmdAction 的子类的话，则会调用该类中对应的 command 方法，并且传入一个名为 name，值为 xxx 的参数。关于 AbstractCmdAction 在后面章节有详细介绍。下面几个 url 请求类似。

高级请求 url 模式：

在Disco中，有一个URL路由映射处理器，通过配置这个映射处理器可以非常简单地实现Web应用中类似URLRewrite的需求。映射处理器代码如下所示：

```
public interface IPathMappingRuler ... {
    //对请求路径的解析；
     public String getModuleName();//得到模板的名称
    public Map getParams(); //得到模板缺省参数
    public String getCommand();//得到模板命令
}
```

该接口的默认实现是cn.disco.web.core.PathMappingRulerImpl。在基于Disco的应用中，每一个交由Disco框架处理的请求url，都会通过这个映射处理器进行转换。通过使用Disco缺省URL映射转换器，客户端请求路径/module/command/params将按以下请求规则，作如下的映射处理.

映射处理前的URL：

1、 /ejf/module/command/name=xxx

2、 /ejf/module

3、 /ejf/module/command

4、 /ejf/module/command/12345

5、/ejf/module/edit/12345/title=1111

6、/ejf/moduel/title=测试

这些url分别对应上面的6中样式，这种url样式比较直观和规范。

URL 映射转换器的一个最典型的应用示例，就是 Disco 中的远程 Web 脚本处理支持引擎，也即 Ajax 的部分功能。远程 JS 脚本调用功能只是 Disco 中的一个小小插件，Disco 的 Ajax 实现只是一个普通的 Disco Module(Action)，这个 Action 即 cn.disco.web.ajax.AjaxEngineAction。

在Disco对Ajax的支持中，可以直接通过下面的URL来动态生成远程javascript调用脚本。

```
<script type='text/javascript' src="ejf/discoajax/prototype.js"></script>
<script type='text/javascript' src='ejf/discoajax/engine.js'></script>
<script type='text/javascript' src='ejf/discoajax/UserService.js'></script>
```

这里只作为一个演示示例，更多关于 Ajax 的信息参见 Ajax 一章。

核心处理

在这一章中将详细讲解 Disco 中核心 MVC 部分的主要部件，包括 IWebAction、WebForm、Page 等。

IWebAction

IWebAction 就是 Disco 中的控制器接口。凡是实现了 IWebAction 接口的类都可以作为 Disco 的控制器。在该接口的代码为：

```
public interface IWebAction {
    public Page execute(WebForm form, Module module) throws Exception;
}
```

execute 方法返回一个 Page 对象，该对象包装了返回的模板等信息，关于 Page 对象更详细的信息参见 Page 对象一节。该方法并没有牵涉到任何的 Http 环境对象，如 HttpServletRequest、HttpServletResponse，使得 Action 及其容易测试。与 Http 环境相关的对象都包装在一个 ActionContext 对象中，可以通过 ActionContext.getContext().getRequest()等方法来得到需要的对象，ActionContext 更多的用法请参看 API doc。

execute 方法第一个参数：WebForm，包装了请求中的参数信息和提交的表单中的信息。同时，在处理器中包含了在处理器中得到了，并且需要合成到视图中的数据。第二个参数 Module，module 对象中包含了很多有用的信息及资源，最常用的一个是 Page，可以使用 Module.findPage(name)在 Action 中找到特定名称的 Page；另外较重要的一个就是控制器，简单说一个 Module 对象对应了一个 IWebAction 对象；另外，Module 中还包括了控制器级别的拦截器、所有在一个 Module 对应 Action 的注入信息等等，关于 Module 更详细的内容参见 Module 一节。

IWebAction 在整个 MVC 流程中就充当了一个控制器的角色，所有的请求都由 ActionServlet 分发到一个指定的控制器中处理。一般来说，一个控制器的处理流程为：使用 form.get()方法从表单或者请求中得到一些参数或者属性，调用业务对象完成指定的功能，调用 form.addResult()方法向 Velocity 上下文中填入要合成（展示）的数据，最后返回一个 Page 对象，完成一个流程的处理。

下面是一个标准的注册的流程示意代码：

```
public Page execute(WebForm form, Module module) {
    SystemUser suc = new SystemUser();
    form.toPo(suc);
    su.setLastLoginIP(cn.disco.web.ActionContext.getContext()
            .getRequest().getRemoteAddr());
    Long id = this.service.addSystemUser(su);
    if (id != null) {
        form.addResult("msg","注册成功");
    } else {
        form.addResult("msg","注册失败");
        return new Page("registerSu","/user/registerSU.html");
    }
```

```
        return module.findPage("registerSuccess");
    }
```

该段代码演示了一个完整的控制器处理流程，注意在这里并没有演示出 form.get("")来取得请求参数或者提交的表单属性，而使用了 form.toPo(obj)方法来合成参数，这是 Disco 的一个很重要的方法，其包含了 VO 到 PO 的拷贝，数据的验证等功能，关于改方法更多的内容请参见 PO 一节。

Page

Page 对象充当的就是 View 的角色，它包装了返回的视图信息。返回的视图可以是任何文本数据格式，比如 Html、XML、java 代码乃到 Velocity 模板等。

Page 对象的角色：

在处理器完成一个请求后，返回一个视图对象，Disco 框架通过该 Page 对象来处理视图模板与程序数据的合成，从而输出动态的内容，用于页面的展示。

Page 的创建：

一个 Page 对象可以有两种创建方式，一种是在 Disco 的配置文件中配置，如下所示：

```
<module name="SystemUser"
   path="/systemUser" action="cn.disco. demo.mvc.userAction">
<page name="reg" url="/user/registerSU.html" type="template"/>
<page name="succ" url="/user/success.html" type="html"/>
<page name="xml" url="/user/xml.xml"  contentType="xml"/>
<page name="view" url="/user/preview.html" type="template"/>
</module>
```

Page 对象有三种：template、html 和 null。template 是指模板，需要使用 Velocity 来合成的；而 html 则不需要合成，其代表一个页面转向，框架使用 redirect 来直接导向该页面；null 两者都不是，他既不合成模板，也不导向，用于在 action 中直接通过 response 来给客户端发送数据或者不对客户端作返回的情况。另外，还可以通过 contentType 来设置模板的输出内容格式，比如"text/xml"、"text/script"等，甚至能直接导向一个流文件。在这种情况下，Disco 不光会使用 Velocity 来合成视图，还会设置 response 的类型为设置类型。

配置好页面后，就可以直接在代码中使用：module.findPage("reg")等来返回一个 Page 对象了。另外，该项配置+代码等同于在控制器中直接返回一个 Page 对象，比如:return new Page("succ","/user/success.html","html")，第一个参数为 Page 的名称，第二个是模板或者 html 页面的路径，最后一个表示模板的类型，如果省略则默认为 template。在 IWebAction 小节的演示注册的代码中，演示了这两种使用方法。

Page 模板/文件的定位

无论使用 new Page()方法还是使用配置 Page 对象，page 对象都需要一个模板路径 path，该路径是相对于框架的模版根路径的.

在默认的情况下，Disco 都会从/WEB-INF/ view/文件夹中加载资源，意思是如果页面的配置为：<page name="succ" url="/user/success.html" type="template"/>，那么该模板文件应该放在/WEB-INF/ view/user/success.html 下。

同时也可以通过置 template-base-path 来修改模板加载的根目录。比如下面的配置代码：

<property name="template-base">/WEB-INF/template/</property>

就可以把模板的根路径定为/webapp/WEB-INF/template/，而比如一个模板的 path 设置为/user/register.html，那么 Disco 就会在/webapp/WEB-INF/template/user/register.html 处加载该模板文件。注意这点很重要，如果在应用中出现了加载模板文件出错的提示，那么就要检查一下自己的模板文件的位置是否正确。

页面流重定向：

假如你希望 Action 的执行结果不要直接输出的浏览器上，而是直接输出到服务器上的一个文件中保存起来(生成静态 html 是我们在建大型网站中所必须的)，或者是输出到互联网上的某一个终端或结点。Disco 给你提供了最简单解决方案，你随时可以根据需要对 Action 执行结果进行重定向，比如下面的 Action 中，我们可以把输出结果指定到服务器上 d:\myapp\news.html 文件中。

```
public void doNews(){
java.io.Writer writer= new OutputStreamWriter(new
FileOutputStream(new File("d:\\myapp\\news.html")),"UTF-8");
ActionContext.getContext().setCustomWriter(writer);
//执行 news 命令的一系列逻辑
page("news");//使用 news 模板来输出结果
}
```

关于 ActionContext 的更多用法，请参看 API doc。

WebForm

在 Disco 中，WebForm 起到了一个 VO 和 TO 的作用，将表单数据或者请求中的参数都包装其中，并且返回页面上需要合成的数据也在其中。有了 WebForm，让 Disco 相对于其它 MVC 框架，在开发的简易和方便性上有了显著的提升。

从 WebForm 得到请求参数/表单属性：

首先介绍 WebForm 对请求参数和表单参数的包装。前面已经提到过，在 WebForm 中存在两个域：`textElement` 和 `fileElement`，这两个域都是 `Map` 对象，分别用来包装文本参数和二进制参数。可以直接使用 `WebForm.get("poropertyName")` 来得到文本参数，`propertyName` 表示的是表单或者请求中的参数名，即类似于使用 `HttpServletRequest.getPrameter` 方法得到的参数值。而对于二进制的内容，直接使用 `WebForm.getFileElement` 来处理。其中的每一个参数都是 `FileItem` 对象。关于 `Common-FileUpload`，请参见：`http://commons.apache.org/fileupload/`。

下面是一个处理上传的例子：

```
public static List<CFile> saveFile(WebForm form, String path){
    List<CFile> cfiles = new ArrayList<CFile>();
    Set keys = form.getFileElement().keySet();
    for (Iterator it = keys.iterator(); it.hasNext();) {
        CFile cfile=new CFile();
        String fieldName = (String) it.next();
        FileItem file = (FileItem) form.getFileElement().get(fieldName);
```

```
            if (file.getSize() == 0) {
                break;
            }
            cfile.setName(file.getName());
            //得到InputStream来构造一个File对象；
            cfile.setFile(getFile(file.getInputStream()));
            Urls.add(cfile);
        }
        return cfiles ;
    }
```

向 WebForm 添加页面合成属性：

在 Disco 中，需要在页面上合成的数据也是使用 WebForm 包装，只需要使用 WebForm.addResult(“name”,value)方法即可以向 Velocity 上下文中添加一个名为 name，值为 value 的属性。另外，WebForm 也提供了一个简便的 WebForm.addPo(obj)来处理多属性的对象的值的添加，该方法配合@FormPO 标签，可以控制对对象属性的输出，比如有一个简单的 User 对象：

```
Public class User{
    private String userName;
    private Long id;
}
```

使用 form.addPo(user)方法相当于调用了：

```
form.addResult("userName",user.getUserName());
form.addResult("id",user.getId());
```

并且可以控制某些属性不能通过 addPo 方法添加。关于@FormPO 标签，请参见 PO 一节。

WebForm 的生命周期：

首先，当一个请求到达时，框架首先解析出请求的 IWebAction，并从 Action 中得到对应的 WebForm 的名字。然后直接调用 FrameworkEngine.creatWebForm(request, formName)来得到需要的 WebForm. 当 WebForm 中填满了表单数据后，框架会解析出 discoCommand 值和传入的参数，也放入 WebForm 中。到此，框架对 WebForm 的处理已经结束，然后，WebForm 就会传入到 IWebAction 中，接着就能对其进行适当的操作。在 IWebAction 中处理完并返回了 Page 后，框架会得到 WebForm 中的那个合成页面值的 Map，并添加到 Velocity 的上下文对象中进行页面模版的合成，然后被丢弃。至此，WebForm 完成一次生命周期。了解生命周期的主要目的是在于对在 WebForm 中存在的数据的生命周期的理解。每一次都会有一个新的 WebForm 产生，但是值得注意的是，WebForm 在将数据保存到 Velocity 上下文时，不光要保存使用 addResult 方法和 addPo 方法放入的值，还要向 Velocity 上下文中保存 textElement 中的数据，意味着重复的数据可以不用在 Action 中重复的添加，这在应用中需要特别注意。该项特性和带环境的页面导向、不带环境的页面导向配合使用，会大大的简化一些情况下的操作。

WebForm.toPo：

上文已经提到过，WebForm.toPo 方法是 Disco 中很重要的一个方法，将请求和表单中的数据直接拷贝到命令对象（Command Object)中，并完成验证等功能，关于该方法，在 PO 一节会有更完整的讲解。

Module

前面介绍了在 Disco 中，IWebAction 对象实际上起着控制器的作用，但是，同时 Disco 也提供了 Module 来包装一个 IWebAction，及其相关的资源和操作等。总的来说，Module 就是一个完成一个具体的请求处理所需要的所有资源的集合。

Module 的作用：

一个 Module 对象包括了一个 IWebAction，对该 Action 所做的拦截器链以及在该 Action 中可能使用到的所有模版或者页面资源。我们知道 ActionServlet 解析请求后，会定位到一个控制器上，前面说到这个控制器就是 IWebAction，其实这是对于开发者来说的，在 Disco 框架内部，这个控制器其实是 Module。只是我们在编写控制器代码的时候，控制器是实现的 IWebAction 接口而已。换句话说，一个 Module 是包含着 IWebAction，而在 IWebAction 中，却是在调用 Module 的资源。

Module 还提供了对于一个 Action 的拦截器链的配置。该拦截器链有别于应用级别的拦截器，它只作用于配置的那个 Module 的 Action 上。

最后，Module 为其中的 Action 所需要的所有 Page 对象作了配置，在 Action 中需要哪个页面模版只需要使用 module.findPage("pageName")就可以了。

Module 和 Action:

上面已经说到，Module 实际上是包含了 Action，而 Action 中可以通过 Module 来获得相关的资源，如 Page。

Module 为 Action 提供了拦截器机制，并且为 Action 准备好了页面模版资源。在 Module 为 IWebAction 配置的拦截器分为两种，前拦截器和后拦截器，这将在 AOP 和拦截器一章中详细介绍。

在 IWebAction 中，如果要返回一个 Page 模版页面，我们前面已经说了，只需要调用 module.findPage("pageName")就能得到了。

Module 和页面模版:

Module 为其中的 IWebAction 对象提供页面模版资源。在上面介绍 Page 对象时介绍了 Page 在 Module 对象中的配置问题。这里再强调一定要注意模板的加载位置的正确性。

Module 的配置

Module 的配置是整个框架配置的关键，特别在引入了容器的概念之后。在这里就只说明 Module 的普通配置，关于 Module 和容器的关系和配置放在容器一章中详细介绍。

下面给出了一个 Module 的完整配置示例：

```
<module name="user" path="/user"
                action="cn.disco.action.helloAction" scope="request">
    <property name="person">
        <ref value="person" />
    </property>
```

```
    <interceptor ref="timeInterceptor" />
    <page name="view" path="/user/view.html" type="template" />
</module>
```

其中的 property 元素是使用属性注入一个对象，scope 是该 module 作为 bean 时的生命周期类型，这两个元素在容器一章详细介绍。

Interceptor 是该 module 中定义的拦截器链，必须实现 cn.disco.web.interceptor.BeforeInterceptor 或者 cn.disco.web.interceptor.AfterInterceptor 接口，用于前拦截和后拦截。关于拦截及 AOP 请看相关章节。

AbstractCmdAction 和 AbstractPageCmdAction

前面讲解了 IWebAction 是控制器需要实现的接口。当然，每一个控制器都实现该接口是比较麻烦的，所以 Disco 提供了 AbstractCmdAction 和 AbstractPageCmdAction 来提供最大限度的开发简易性和灵活性。

AbstractCmdAction：

AbstractCmdAction 是很类似于 spring 的 MultiActionController 的 Action，即在一个 Action 中提供多个处理方法，其使用某一种请求方式来匹配某一个处理方法来处理请求。与 MultiActionController 稍微不同一点的是，该类没有提供过多可选映射方法的策略和参数的配置，而直接固定了这些策略。使用 AbstractCmdAction 可以将具有相同或者相关的请求处理方法放在一个 Action 中，从而达到一种模块化(Module)的效果。

在请求中使用 discoCommand 或者 cmd 传递的参数直，会作为请求执行的方法名的构成。构成的规则为 doXxxx()。其中 Xxxx 表示的是首字母大写后的该值。比如下面给出了一个继承了该类的代码片断：

```
public Page doExit(WebForm form, Module module){
ActionContext.getContext().getSession().removeAttribute("user");
return new Page("goto","/exit.htm","html");
}

public Page doShowLogin(WebForm form, Module module) {
return module.findPage("login");
}
```

如果请求的是 xxx.java?discoCommand=exit 或者/ejf/xxx/exit，则会调用 doExit 方法并返回适当的页面，如果请求的是 xxx.java?discoCommand=showLogin 或者/ejf/xxx/showLogin 或者 xxx.java?cmd=showLogin，则会调用 doShowLogin 方法并返回适当的页面。

在 1.0 版本中，Command 类型的 Action 基类即 AbstractCmdAction 中，提供了足够的灵活特性。可以根据不同的应用场景，使用不动的方法签名来写 Action 中的 command 方法，使用起来非常灵活。

假如我们在一个模块中要执行一个名为 create 操作，下面的方法签名都是合法的：

```
public Page doCreate(WebForm form,Module module)
public Page doCreate(WebForm form)
public Page doCreate(Module module)
public Page doCreate();
public void doCreate(WebForm form,Module module)
```

public void doCreate(WebForm form)
public void doCreate(Module module)
public void doCreate();
另外，把方法名称改成 create，系统也一样能识别。如：
public Page create(WebForm form,Module module)
public Page create(WebForm form)
public Page create(Module module)
public Page create();
public void create(WebForm form,Module module)
public void create(WebForm form)
public void create(Module module)
public void create();

我们可以使用 xxx.java?discoCommand=create 的方式来调用这个方法，也可以使用 xxx.java?cmd=create 的方式来调用，还可以使用/ejf/xxx/create 的形式来调用。

这种灵活的 Action 中的命名方法，一方面可以使代码更加简洁，易维护，同时也使我们的代码看起来更 cool。另外一个主要原因是其使我们可以非常容易书写这些方法的测试代码，不需要任何 Web 容器，就能运行 Disco 的单元测试。

在 Disco 提供的所有的演示应用中，几乎所有的 Action 都是继承了 AbstractCmdAction。同样，该类也提供了如 AbstractCrudAction 的 doInit，doBefore，doAfter 方法，这里就不赘述。

参见 Disco 附带的示例，就能更深入的理解 AbstractCmdAction 带来的简易性。

AbstractPageCmdAction：

在 AbstractCmdAction 的基础上，Disco 提供了另外一个对视图具有自动识别能力的 AbstractPageCmdAction。该类能通过应用“惯例代替配置”来智能判断程序中要使用的视图模板，甚至不需要书写方法签名也可以调用视图模板。

如一个空签名的 command 方法：

public void doEdit(){}

该方法中虽然没有一句代码，也没有明确的视图切换或转向。但 AbstractPageCmdAction 会根据惯例原则，自动选择 edit.html 文件作为视图模板。

另外针对下面的 url:person.java?cmd=new 或/ejf/person/new

假如 PersonAction 中没有 doNew 方法，则 AbstractPageCmdAction 会根据惯例原则，优先考虑/views/person/目录中是否存在 new.html 文件，若存在，则将直接返回该模板，即不用写 java 方法，也能产生动态页面。

作为一个应用实践之一，在我们开发一般的 WEB 应用中，我们习惯于把前台及后台分开，比如所有前台展示的放在一个模块 Module(即 Action)中。网站前台经常会有非常多的页面，而这些页面或多或少都会包含相同或相似逻辑的动态内容，此时若针对每一个页面都写一个 Command 来处理，显得很麻烦。可以借助 Velocity 的标签，以及 AbstractPageCmdAction 中的视图界面智能选择功能，不用写 java 方法，即加载各种动态页面。

下面是 Disco 官方网站 NewsAction 中的代码，类似这样的代码也用于很多 Disco 开源团队所开发的开源及商业项目中：

```
public class NewsAction extends AbstractPageCmdAction {
    private Map utils = new HashMap();
    public void setUtils(java.util.Map utils) {
```

```
        this.utils = utils;
    }
    public Object doAfter(WebForm form, Module module) {
        java.util.Iterator it = utils.entrySet().iterator();
        if (it != null) {
            while (it.hasNext()) {
                Map.Entry en = (Map.Entry) it.next();
                form.addResult((String) en.getKey(), en.getValue());
            }
        }
        return super.doAfter(form, module);
    }
    }
    /news.java?cmd=index
```
/news.java?cmd=technic

/news.java?cmd=download

可以这么说，前台诸多的动态页面，都不需要书写 Java 代码，就能实现动态内容生成。如果你借助一些基础引擎，比如 Disco 开发的 CMS 引擎或者是自己构造一些引擎，完全可以实现不用写 Java 代码，就能开发出功能比较强的 Java Web 应用。比如建一个站点、建论坛、百科、Blog、全文检索等。

配置文件

Disco 作为一个 MVC 框架，使用一个 Servlet 负责转发所有的请求。跟使用其它的任何 MVC 应用框架一样，要让 Disco 工作，首先必须做一定的配置工作。除了要配置 web.xml 以外，Disco 还自己专门的配置文件，在 Disco 专用配置文件中，可以配置 Disco 的工作模式、模块(Module)信息、表单(Weboform)信息、Bean 信息，另外还可以配置拦截器、异常处理、权限处理、AJAX 等相关信息。当然，Disco 尽量使配置文件变得简单，甚至在特定应用场景中，只要遵循一定的规则，Disco 的配置可以趋向于零，也就是前面介绍中所说的零配置。即只配置 web.xml，而不再需要配置其它的 Disco 配置信息。

web.xml

要让 Disco 工作，首先需要配置 Web 应用程序中的 web.xml 文件,在 web.xml 文件中配置一个 servlet，并用这个 servlet 来处理特定或所有的 url 请求。我们推荐的 web.xml 配置如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4">
 <!--定义 Disco 的主控 Servlet   -->
 <servlet>
  <servlet-name>disco</servlet-name>
  <servlet-class>cn.disco.web.ActionServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
```

```
 </servlet>
 <servlet-mapping>
  <servlet-name>disco</servlet-name>
  <url-pattern>*.java</url-pattern><!--所有.java 的扩展名都由 disco 来处理-->
 </servlet-mapping>
  <servlet-mapping>
  <servlet-name>disco</servlet-name>
  <url-pattern>/ejf/*</url-pattern><!--所有/ejf/*样式的 url 都交由 Disco 来处理-->
 </servlet-mapping> <!-- 定义字符处理 Filter -->
 <filter>
  <filter-name>CharsetFilter</filter-name>
  <filter-class>cn.disco.web.CharsetFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
  <init-param>
    <param-name>ignore</param-name>
    <param-value>true</param-value>
  </init-param>
 </filter>
 <filter-mapping>
  <filter-name>CharsetFilter</filter-name>
  <servlet-name>disco</servlet-name>
 </filter-mapping>
</web-app>
```

还可以在 web.xml 文件中使用下面的信息来指定 Disco 专用配置文件的位置：

```
      <!--定义 disco 的配置文件位置 -->
        <context-param>
        <param-name>discoConfigLocation</param-name>
        <param-value>
           /WEB-INF/disco-web1.xml,/WEB-INF/disco-web2.xml
         </param-value>
        </context-param>
```

上面的配置参数将使得 Disco 加载的时候，将会从/WEB-INF/disco-web1.xml 及/WEB-INF/disco-web2.xml 两个 Disco 配置文件中加载配置信息。

Tip：

另外,也可以在这里只配置一个主 Disco 配置文件,并在该配置文件中使用<import>来包含其他的配置文件。在实际应用中，我们建议分模块的使用 Disco 配置文件，并在主配置文件中包含这些文件。主文件中配置一些基础的框架配置或者容器配置，比如和 spring 结合的配置等。这样便于模块的管理和分模块开发集成。我们推荐的配置文件分布：一个主文件 mvc.xml，各个模块有各自的配置文件，比如 blog-mvc.xml；bbs-mvc.xml 等，一个 ajax 的配置文件 ajax.xml，都使用<import>包含到 mvc.xml 中。

零配置

当然，Disco 尽量使配置文件变得简单，甚至在特定应用场景中，只要遵循一定的规则，Disco 的配置可以趋向于零，也就是前面介绍中所说的零配置。即只配置 web.xml，而不再需要配置其它的 Disco 配置信息，Disco 就能很好地的工作，并让您的 Web 应用程序拥有纯 MVC 框架带来的好处。

disco-web.xml

disco-web.xml 文件是 Disco 的默认配置文件，存放在 WEB-INF 目录下。一个 Disco 配置文件的结构大致如下：

```
<?xml version="1.0" encoding="utf-8"?>
<disco-web>
 <frame-setting>
  <property name="cn.disco.debug">true</property>
  <property name=.../>
  ...
 </frame-setting>
 <modules>
  <!-- 用户添删改查模板配置信息-->
  <module path="/userManage" scope="singleton"
action="cn.disco.action.userManageAction"
   defaultPage="list">
   <page name="edit" url="/userEdit.html" type="template" />
   <page name="list" url="/userList.html" type="template" />
  </module>
  <module .../> </modules>
 <forms>
  <form name="loginForm" bean="cn.disco.site.form.LoginForm"
   event="" clientValidate="yes" serverValidate="yes"
   alertType="javascript:page"> </form> ...
 </forms><!--用于配置 Bean 信息-->
 <beans>
  <bean name="person" class="cn.disco.business.Person"
   scope="singleton">
   <property name="name" value="五小平" />
   <property name="age" value="15" />
   <property name="sex" value="男" />
  </bean>
  ...
 </beans>
 <!--用于设定 Ajax 可供 ajax 用的服务-->
 <ajax>
```

```
  <services allowName="*Service" denyName="">
   <service name="UserService">
   <include method=""></include>
   <exclude method=""></exclude>
   </service>
   ...
  </services>
  <!-- 用一设值可转换的 Bean-->
  <convert name="cn.disco.business.SystemBasicData">
   <!-- <include property=".*"></include> -->
   <!-- <exclude property="children"></exclude> -->
  </convert>
  ...
  <signatures>
  UserService.queryUser(String scope,Collection[String] paras,int begin,int max)
  </signatures>...
 </ajax>
</disco-web>
```

通过上面的结构可以看见，每一个 Disco 配置文件的根元素均为<disco-web>，在<disco-web>元素下面可以包含 0 或一个<frame-setting>元素、0 或 1 个<modules>元素、0 或 1 个<forms>元素、0 或 1 个<beans>元素、0 或 1 个<ajax>元素，根据项目的性质可以选择使用这些元素，下面将分别对这些元素作简单的介绍。关于 Disco 的配置文档的详细定义，请参看 DTD 和 schemle。

<frame-setting>

这个元素主要用来定义 Disco 的一些基本信息及全局信息，包括工作模式、模板位置、附件最大值、初始应用程序，全局拦截器等。其下\包括<property><init-app><interceptors>等子元素。<property>用来定义 Disco 的基本运行参数,如下面的配置信息：

```
  <?xml version="1.0" encoding="utf-8"?>
  <disco-web>
 <frame-setting>
  <property name="cn.disco.debug">true</property><!--调试模式，每次加载都会重新初始化Disco，只适合开发时少数几台机器使用-->
  <property name="cn.disco.maxUploadFileSize">1024</property><!-- 最大上传文件为1024kb-->
  <property name="cn.disco.uploadSizeThreshold">512</property><!-- 上传文件缓存值--></frame-setting></disco-web>
```

<init-app>用来配置伴随 Disco 启动而启动执行的应用程序,这些程序一般可以作额外的初始工作，或者是以一个新线程的形式在系统中运行。如下面是 Disco 开源 Blog 系统中，自动生成静态 html 文件的程序：

```
<frame-setting>
<property name="cn.disco.debug">false</property>
<init-app>
<app-class class="cn.disco.blog.logic.AutoHtmlGenerater" init-method="start" />
```

</init-app>
</frame-setting>
<modules>

<modules>元素下面是 Disco 的模块信息，在 Disco 中，可以把功能相关或形式的相近的请求处理程序封装到一个模块 Module 中，<modules>元素下面包含 0 或多个<module>元素，在<module>中将包含具体的 Module 配置信息，Module 是 Disco 应用程序中重要的概念，我们将在后面详细介绍<module>元素的详细配置。

<forms>

<forms>元素下面包含 0 或多个<form>元素，<form>元素用来定义前台页面表单的信息。在普通的应用中，一般不使用这个配置选项。

<beans>

<beans>元素下面包含 0 或多个<bean>元素，<bean>元素一般为业务层对象，关于<bean>的配置将在后面专门介绍。

<ajax>

<ajax>元素主要用来定义应用程序中支持远程 javascript 脚本访问的业务对象，下面包含 0 或多个<service>元素，0 或多个<convert>元素，0 或多个<signatures>元素，我们将在 Ajax 一章中专门介绍。

<module>

Module 标签定义了一个 Module 对象。根据前文对 Module 对象的讲解，应该能够很直观的了解 Module 的配置。Module 中包含了 Page 对象（Page 一节），控制器（Module 一节），拦截器（AOP 和拦截器一章），注入的 bean（容器一章）分别介绍其配置。

Tip

在一个配置文件中，并不要求包含所有的以上的元素，请根据实际情况配置最小需要的元素。比如如果按照我们推荐的 mvc 文件分布，则在主文件 mvc.xml 中，只需要配置<import>包含文件、<frame-setting>、错误处理器以及和 Spring 等容器集成的基础配置。而在各个子配置文件中只需要<modules>或者<ajax>标签。

PO 和 WebForm

WebForm 是 Disco 的一个重要的组件，在这一小节中，将介绍 PO 和 WebForm 的 toPo 方法。在验证一小节中介绍 toPO 方法中的验证。

PO

PersistenceObject，持久化对象，VO value Object，值对象，TO Translate Object 传输对象，在 Disco 中，WebForm 扮演了 VO 和 TO，而 PO 直接由 Domain 扮演。

toPo 方法

在一个带参数的请求中，或者是在一个有表单域的请求中，如何从中取得值，用于保存对象或者处理？在 Struts1.x 中，使用了一个 ActionForm 来承载这些数据；struts2.x 中，使用 Controller 作为 Command 对象来承载这些数据；springMVC 中，可以使用带 Command 对象 Controller，使用一个 Command 对象来承载这些数据；而在 Disco 中，是 WebForm 承载了这些数据，并且使用一个 toPo 方法来直接将这些数据拷贝到需要持久化的对象中。比如下面一个代码片段演示了保存一个论坛帖子的功能：

```
public Page doSave(WebForm form, Module module) {
    BBSDoc doc = form.toPo(BBSDoc.class);
    List<CFile> attachs = ActionUtil.saveFile(form, webConfig
            .getImageDirPath(), false);
    doc.setIp(ActionContext.getContext().getRequest().
getRemoteAddr());
    List<BinaryResoure> attachList = this.binaryService
            .addAttachMent(attachs);
    doc.setFiles(attachList);
    this.service.addBBSDoc(doc);
    return this.doBbsList(form, module);
}
```

在WebForm中，有两种toPo方法：

WebForm.toPo(Object obj)：传入一个对象，直接从WebForm中拷贝值到改对象中。

WebForm.toPo(Class clazz)：传入一个类型，WebForm会初始化这个类型的实例再将WebForm中的值拷贝。关于属性的拷贝,有几点注意:1，只会拷贝属性相同的值,比如提交的表单中包括一个name，那么，在持久化对象或者Command对象中，只会拷贝String、Integer的name属性。

使用@FormPO标签来规定拷贝的规则：

inject:

指定可通过toPo自动注入的属性，默认为全部可自动注入;如果设置了该值，则表示除指定可注入的属性以外，其它所有属性都为不能自动注入；在Disco中，可以直接通在模型(域)对象上通过标签来指定只允许注入哪些属性值。如果一个属性值为可注入的，则可以使用WebForm.toPo(obj)方法，把WebForm中与obj属性名称相同的属性值设置到obj中。需要注入的属性使用逗号(,)作为分逗符。比如：

```
@FormPO(inject="name,bornDate")
public class Person
```

即在Person类（或者Command对象）中，当使用toPO方法拷贝值的时候，只有name和bornDate属性被拷贝，其余如果有符合的属性，都不准被拷贝。这在很大的层度上提高了应用的安全性。

disInject:

指定不可通过toPO自动注入的属性，当试图通过toPO更新该属性时，将会被忽略，并在日志中提示相关信息。在Disco中，一个模型(域)对象的属性默认情况下全部都是可注入的，我们可以通过disInject来指定不可注入的属性。多个不可注入的属性使用逗号(,)作为分隔。 如上面inject示例，也可以写成下面的形式：

```
@FormPO(disInject="id,loginTimes")
public class Person
```

Tip:

在实际应用中，可以根据实际情况选择使用 inject 及 disInject，如果需要拷贝的属性较多，则选择 disInject；而拷贝的属性较少，则选择 inject。

控制 addPo 属性的可见性：

前文在介绍 WebForm 的时候，提到了 addPo 方法，在这个方法中，其实也有控制标签：

disRead:

指定为 disRead 的属性在 addPo 方法中不会被添加到 Velocity 上下文中。

下面，给一个这些标签的使用的一个范例：

```
@Entity
@FormPO(name                                                    =
"person",inject="name,sex,mail,intro",disInject="age",disRead="se
rialVersionUID,id")
public class Person implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private Long id;
    @Column(length = 32)
    private String name;
    @Column(length = 6)
    private String sex;
    @Column(length = 40)
    private String mail;
    private int age;
    @Column(length = 200)
    public String intro;
}
```

在这个例子中，如果调用WebForm.toPo(Person.class)，那么Person中的name、sex、mail将会拷贝，而age不会被拷贝，并且调用WebForm.addPo()方法时，Person的serialVersionUID和id将不会被添加到Velocity上下文中。关于更多@FormPO标签的用法，请参见API doc。

验证

在 Disco 中，最常用的方法是通过 WebForm.toPo 方法在拷贝属性的过程中执行验证，同时，也可以调用 FramewokEngine.form2Obj 方法执行验证。

Annotation

Disco 不使用传统的 XML 配置来完成验证，我们认为，验证应该是和 PO 或者 Command 对象紧密相连的，为了避免大量的 XML 文件的存在，采用了 Annotation 来完成验证。Disco 同时也不推荐使用过多的 Annotation，从而造成 Annotation 泛滥，因此 Disco 的验证标签只有一个，即@Validator。

Disco 中的验证

Disco 完善了验证系统，使验证变得更加容易、灵活、统一控制。你可以使用非常简单的标签或配置就能使系统拥有服务器端及客户端验证的功能。

下面以一个简单的示例：

有一个 Person 模型，如下所示：

包含 id、name、sex、borndate、height、mail、homepage 等几个属性。

假如我们要让 name、sex、heigth、borndate 必填，并且 borndate 必须在 1908 到 2008 年之间， mail 属性只接收正确的 email 信息，homepage 必须接收 url 信息。则我们只需要在 Domain 对象（可以是 Entity 对象，也可以是传输 Command 对象）中加入下面的配置信息，即可。

@FormPO(name="person",validators={@Validator(name="required",field="name,sex,heigth,borndate"),@Validator(name="range",field="borndate",value="min:1908-01-01;max:2008-01-01"),@Validator(name="email",field="mail"),@Validator(name="url",field="homepage")})

public class Person{

private Long id;

private String name;

private String sex;

private String mail;

private Integer heigth;

private Date borndate;

public String homepage;

…//setter 及 getter 方法

}

不需要进行复杂的配置，只需要使用符合人类语言习惯的简单标注，就能实现所需要的验证业务逻辑。

@Validator 验证标签的使用非常灵活，你只要具有充分的想像力，就能描述出符合特定需要的验证逻辑。比如上面的例子中，我们规定 name 不允许为空，字符数最小不能少于 5 个，最大不能超过 10 个，在进行字符验证前需要清除掉前后的空格。则我们可以使用下面的验证标签：

@Validator(name=”string”,value=”blank;trim;required;min:5;max:10;minMsg: 最少不能少

于 5 个字符;maxMsg:最大不能超过 10 字符”)

private String name;

统一的验证标签@Validator

Disco 中只有一个用于验证的 Annotation，即@Validator，通过一个@Validator，就能标签任意类型的验证，使用起来非常简单。@Validator 的代码如下所示：

```
public @interface Validator {
    public String name();// 验证器的名称，如required，string，range等
    public String value() default "";// 验证器的值，使用;号作为分隔符存放各个参数。如value="required;min:5;max:20"
    public String msg() default "";// 默认错误提示信息，当验证无法通过时显示的提示信息
    public String field() default "";// 字段名称，对于property及field类型的校验均可用，也是错误对象的主属性名称。可以用于多个字段，此时需要使用,隔开
    public String displayName() default "";// 定义对象的显示名称,默认情况下为field的名称，可以通过@Field中的name属性定义。
    public ValidateType type() default ValidateType.Property;// 校验类型，默认是对属性进行校验
    public boolean required() default false;// 是否必填字段，每个验证器都可以通过设置属性required=true来指定该属性为必填项
    public String key() default "";// 多国语言显示值的编码
}
```

Disco 内置的 Validator 验证器

Validators.RequiredValidator－用来定义必填属性，预定义名称 required。

Validators.StringValidator－字符串验证器，定义字符串的属性，预定义名称 string。

Validators.URLValidator－URL 字符串验证器，匹配一个合法的 URL，预定义名称 url。

Validators.RegexpValidator－正则表达式验证器，匹配指定条件的正则表达式，预定义名称 regex。

Validators.EmailValidator－Email 字符验证器，匹配正确的 email 字符串，预定义名称 email。

Validators.RangeValidator－范围验证器，用来限制属性必须在指定的范围之内，预定义名称 range。

验证器引擎的触发也是非常灵活的，如果是普通的 CRUD 应用或者是基于普通 CRUD 应用基础上扩展的应用，则在基本的添删改查中会自动调用验证逻辑。

如果是自定义的 Action，可通过调用 form.toPo 等方法触法验证逻辑。如果是使用 IDAO 接口进行的调用，则在进行数据持久化之前会调用验证逻辑。

关于内置验证器的更多细节，请参看 API doc。

实现自己的验证器

要自定义验证器很简单，只需要实现 Validator 接口，即可注册到系统中使用。在实际应用中，一般通过使用继承抽象类 `AbstractValidator` 来实现自定义的验证器。如下面是最简单的验证器 `Required` 的实现：

```
public class RequiredValidator extends AbstractValidator {
    public RequiredValidator() {}
    public void validate(TargetObject obj, Object value, Errors errors)
{
        if (value == null)
            addError(obj, value, errors);
        else if (value instanceof String) {
        }
    }
    public String getDefaultMessage() {
        return "{%0}不能为空!";
    }
}
```

在完成了自定义的验证器后，只需要在容器中声明该 bean，并将 bean 的名字设置为需要调用的验证器的名字即可。比如如果要将上面的 RequiredValidator 作为 notNull 验证器使用，则在容器中配置：

<bean name="notNull" Class="com.xxx.xxx.vaildator.RequiredValidator"/>即可，然后在 Domain 或 Command 对象中可以通过@Validator(name="notNull")来标注使用该验证器。

验证错误获彼此

Disco 中使用了 Errors 对象来包装在验证过程中出现的错误，该对象提供了三个有用的方法：

int *getErrorCount*()：用来显示出错的个数；

boolean *hasError*()：用来显示是否有错；

String *getMessage*(String fieldName)：用来显示某一个属性的错误；

在页面中显示错误：

$!errors－显示全部验证错误信息。

$!errors.name－显示 name 属性(字段)的错误信息。

如下面的的 Form

```
<form name="person_new_form" id="person_new_form" method="post"
action="/ejf/person/create">
    请输入姓名：<input name="name" type="text" id="person_name"
value="$!name" />$!errors.name
    电子邮箱：<input name="mail" type="text" id="person_mail"
value="$!mail" />$!errors.mail
```

在代码中获得错误：

在 Action 中，如果要捕获错误，即得到 Errors 对象，使用：

FrameworkEngine.getValidatorManager().getErrors()方法来得到一个验证错误对象 Errors。通过 Errors 可以检查是否包含未通过的验证逻辑，从则作相应的处理。

匹配错误的处理：

如果发生匹配错误，有三种处理方式：

1，忽略，在 toPo 方法中，设置 validateRollback 为 false，则发生匹配错误继续匹配下面的属性。

2，回滚，在 toPo 方法中，设置 validateRollback 为 true，那么在发生匹配错误时，将已经拷贝了的属性还原到拷贝之前。这在使用 JPA 作为持久层的时候特别有用，避免了设置 flushMode。

3，强制性出错，在<module>中设置 validate 为 true，则在匹配出错时，马上抛出一个框架错误，不会继续向下匹配。

关于错误显示的更多细节，请参看相关 API doc。

Disco 的错误处理

在 Disco 中提供了自定义错误处理机制。如果在控制器中发生了错误，那么可以根据配置，来完成对特定错误的特定处理过程。

要实现错误处理器，需要实现 cn.disco.interceptor.ExceptionInterceptor 接口，该接口定义了一个方法：

```
boolean handle(Throwable e, Object target, Method method, Object[] args) throws Exception;
```

其中，e 是出现的异常，在自定义的错误处理中，可以判断这个 e 是否 instanceof 你需要处理的 Excepion，而选择是否处理。Target 是异常出现的对象，一般来说会是控制器对象。Method 是异常抛出时调用的方法，args 是相关参数，这几个参数都是 AOP 拦截器规定的参数。该方法返回了一个 boolean 值，如果返回 true，则表示异常已经成功处理，不再需要作其它的处理，返回 false 则，表示把异常交给下一级异常处理器进行处理，如果抛出异常，则表示将直接把异常交给外部程序或发给用户。可以在自定义的错误处理器中完成自定义的错误处理规则。比如根据错误的不同，显示不同的错误页面。如果配合 target 和 method 参数，甚至可以将出错页面的导向细化到每一个 Action 的每一个 Method 上。

在完成了自定义的错误处理器后，可以直接配置一个 Bean，这样 Disco 会自动使用该异常处理器来处理在系统运行过程中出现的特定的异常。所有实现 ExceptionInterceptor 的异常处理器会会构成一个错误处理器链，来协作处理 Action 中抛出的错误。在这条错误处理器链的最后，是系统级别的 **DefaultExceptionHandle**，可以在容器中配置该处理器，在这个处理器中，如果不配置这个处理器，或者在容器中没有对这个处理器做特别的配置，将显示默认的错误提示页面。在这个处理器中，可以通过配置 errorPage 属性，来定义自己的出错页面。

工具类

Disco 提供了一些有用的工具类来辅助开发，提高开发效率。

CommUtil

CommUtil 提供了很多常用的方法，其中分页是重要的一个。CommUtil 不光只用在后台控制器中，而且 Disco 内置的将 CommUtil 的一个实例放入了 Velocity 上下文中，使得可以直接在前台使用$CommUtil 来调用。下面介绍 CommUtil 的几个常用方法：

- $!CommUtil.*formatDate*(formatStr,Date)：

  按照规则格式化时间。Date 为传入的类 型为 Date 的时间对象，formatStr 是格式化样式，使用的是 SimpleDateFormat 格 式 化 ， 比 如 $!CommUtil.formatDate("yyyy-mm-dd",$date) 就 可 以 将 date 显 示 为 2000-12-29 类似的样式，关于格式化字符请参见 SimpleDateFormat。

- $!CommUtil.longDate(Date)：

  完整的时间格式化，样式为 yyyy-MM-dd H:m:s。

  还有更多的时间格式化样式，比如显示中文时间的等等，请参见 CommUtil 的 API 文档。

- CommUtil.null2String(nameStr)：

  在 Action 中常用的方法，一般用于从 WebForm 取得字符串性的属性：

  ```
  String idStr=CommUtil.null2String(form.get("id"));
  if(!"".equals(idStr)){
      //...
  ```

  该方法避免对 null 的验证。如果是 null，返回"";

  同时，也提供了一个 null2int 方法，如果属性为 null，返回 0。

- $CommUtil.toRowChildList($List,sizePerRow)：

  将一个 List 转换成 sizePerRow 为每行的一个 List 的 List：List<List>

  在页面显示的时候特别有用，我们经常遇到在页面上需要把一个 list 分成几行来显示，不需要太多的 VTL,也不需要 vmacro，只需要：

```
$foreach($subList in $!CommUtil.toRowChildList($!infoList,num))
$foreach($info in $!subList)
$info.show()
$end
$end
```

更多的 CommUtil 方法请参见 API 文档。

分页

Disco 提供了一个比较完整的分页接口和相关工具。IPageList 接口定义了一个分页对象的信息，IQuery 接口定义了分页查询的方法。

在 IPageList 中定义了比如当前页，所有页面数，每页大小等信息。

常用的分页组件：

Disco 提供了一个为 List 对象分页的组件，只需要象下面这样使用：

```
IPageList pList=new PageList(new ListQuery(resultList));
pList.doList(pageSize,currentPage,null,null);
```

这样就得到了一个 IPageList 对象了，这里面有完整的分页信息。同时 Disco 提供了

CommUtil.saveIPageList2WebForm(IPageList,WebForm)，使用该方法可以快速的将分页信息放入 Velocity 上下文中，并且自动生成翻页需要的 html。通用的在页面的分页使用为：

在列表页面上放入一个 Form，添加两个 hidden 域：

```
<input type="hidden" name="pageSize" value="$!pageSize" />
<input type="hidden" name="currentPage" value="$!currentPage"/>
```

第一个是用于保存每页的记录数，第二个保存当前页面，

在需要放置分页的位置添加：

```
共$!rows条 第$!page/$!pages页 每页$!pageSize条 $!pagination
```

其中，$rows 是所有的条数，$page 是当前页数，$pages 是总共条数,$pageSize 是每页的条数，$pageination 是 Disco 生成的分页 HTML，如下图：

同时，Disco为JPA+Spring+Disco的应用提供了一个JPA的分页组件：QueryUtil。只需要把查询的条件包装为IQueryObject的子类，就可以使用

`public static IPageList query(IQueryObject queryObject,Class entityType,GenericDAO dao)`来完成整个分页过程。

关于更多分页和实现自定义分页请参见相关文档。

tagUtil

tagUtil 也是 Disco 内置的加入到了 Velocity 上下文中的用于生成页面标签的工具类。下面列举几个常用的方法：

- `String options(final List list,final String valuePoperty,final String titlePoperty,final String value):`

用于根据一个 `list` 生成多个选择项，其中，`list` 是需要用于生成选择项的对象的列表，比如 `List<User>`；`valueProperty` 是用于生成选择项的 `value` 的属性，比如要使用 `User` 的 `id` 生成，那么就填 `id`；`titleProperty`：用于生成选择项的名字的属性，比如要使用 `User` 的 `name` 生成，这里就填 `name`；`value`，根据 `value` 来设置选择项，比如在选择总经理的一个页面中，已经选择了某个 `user` 为总经理，那么在修改的时候，要选中这个 `user`，这里只需要传入已经选中的 `user` 的 `id` 就可以在确定的 `user` 的选择项中添加 `selected="selected"`。

在 `tagUtil` 里面还有生成分页 `HTML` 的方法，在自定义分页 `HTML` 样式时，可以参考这些方法。

更多的关于 `tagUtil` 的用法请参看 `API doc`。

验证码

Disco 内置了一个验证码生成和检查器，如果你的应用需要加入一个验证码，只需要：

1，在 mvc.xml 里面添加：

```
<modules>
                    <module path="/randomCode" form="" scope="request"
action="cn.disco.web.tools.RandomImgCode" defaultPage=""/>
```

```
</modules>
```

2，在页面上需要显示验证码的位置添加类似：

```
<div>验证码：</div>
<div>
<input name="randCode" id="randCode" />
<img border="0" src="randomCode.java"
       style="cursor:pointer;" alt="点击这里换一个"
       onclick="this.src='randomCode.java?'+Math.random()";>
</div>
```

3,在接受这个验证码的 Action 中添加类似代码：

```
String randomCode = (String) ActionContext.getContext().getSession()
              .getAttribute("rand");
if (!randomCode.equals((String) form.get("randCode"))) {
    form.addResult("msg", "验证码不正确，请重新输入！");
    return this.doRegisterPU(form, module);
}
```

即可完成整个验证。

如果要生成自定义的验证码，可以参看 ImageCode 和 RandomImgCode 的实现。

容器部分

Disco 提供了一个简单，但功能完整的 IoC 容器，并且该容器能很好的和其他容器并肩作战，形成一个超级 IoC 容器。在本章中，我们将详细介绍 Disco 容器的使用，并介绍容器间的整合。

Disco 的容器

Disco 提供了一个内置的 IoC 容器，该容器不但能做为一个 IoC 容器配合 Disco 一起使用，而且还能通过简单的配置集成 Spring、Guice 等其他容器，借用其他容器强大的功能，比如事务管理等。同时，Disco 使用@Inject 标签等，使得使用容器中的 bean 更加简单和透明。

IoC

IoC(Inversion of Control)中文译为控制反转，业界最初的定义是指框架所拥有的一种管理业务对象的能力，重点在于管理组件中和其依赖的业务对象。随着大家对 IoC 认识的提高，IoC 的概念也更加清晰：IoC 作为一种设计模式，意味着在系统开发过程中，组件中业务对象以及组件之间的关系将交由容器去控制，而不是由类自己控制，实现了 IoC 模式的框架有一种从组件外部把所依赖的业务对象注入到组件中的能力，即"don't call us, we'll call you"，也即“不用你找,让我们(框架)来提供给你”；另外，IoC 容器还提供定位器 Locater 功能，使得在应用程序中可以通过统一规范的名称从容器中查找并取出所需要的组件。因此，准确地说，IoC 是一种容器框架模式，具有以统一的方式负责组件及业务对象的注册、贮存及查询功能，另外还具有往组件中注入所需要的业务对象的能力。前者叫做定位器，也可以称为依赖查询，后者称为依赖注入 DI。

关于 IoC 的更多资料，可以参见《J2EE without EJB》和 Disco 的《深入 Spring》。

Disco 中的容器

Disco 提供了一个简约而不简单的 IoC 容器。在本小节中，我们首先来看作为一个 IoC 容器的基本使用方法，在下一小节中将看到该容器更高级的使用。

Bean

在应用开发中，经常会使用到组件、服务或业务对象等词，由于这些词的理解可大可小，不同的角度可以用不同的名称代表同一个东西，为了与传统混乱的定义区别开来，统一使用“Bean”这个名称来标识系统中的部件，这些部件可能是一个功能简单的对象，甚至仅仅是一个 String 字符串对象，也可以是一个功能强大的组件或服务。

Bean 是应用程序中的基本元素，应用程序，就是由一个又一个的 Bean 组合在一起、像搭积木一样堆出来的。所有的 Bean 都由核心容器负责管理、创建、销毁，同时 Bean 之间的相互依赖也由容器中的依赖注入功能自动管理。对于一个具体的应用来说，Bean 就是一个 DAO 对象，一个 Service 对象，Module 对象，工具类等等所有构成应用的对象。

配置一个 Bean

在 Disco 中配置一个 bean 是很简单的事情，下面是配置 Disco 配置 bean 的基本样式：

```
<?xml version="1.0" encoding="UTF-8"?>
<disco-web>
    <frame-setting>

    </frame-setting>
    <modules>

    </modules>
    <!-- 配置bean开始：-->
    <beans>
    <bean name="exceptionHandler"
          class="cn.disco.web.exception.DefaultExceptionHandle"
          scope="singleton">
       <property name="errorPage" value="/error.html"/>
    </bean>
    </beans>
    </disco-web>
```

Bean 都包含在一个<beans>标签内。

一个 bean 由名字，类型，生命周期，bean 属性等构成。在上面的配置文件中，我们配置了一个全局默认错误处理器（请参见错误处理一章），该 bean 的名字为 exceptionHandler，类型为 DefultExceptionHandle，类型为 singleton（单例）。其中为该 bean 配置了一个属性，该属性是 DefultExceptionHandle 中的 String errorPage 属性。

Bean 的属性注入类型

Disco 中可以通过属性注入，也可以通过构造方法注入。

使用属性注入时，需要设置属性的 setter 方法，比如：

```
public class Person implements IPerson {
private String name;
private int age;
private IPerson father;
private IPerson mother;
public Person(String name,int age,IPerson father,IPerson mother){
    this.father=father;
    this.mother=mother;
    this.name=name;
    this.age=age;
}
public IPerson getFather() {
    return father;
}
public void setFather(IPerson father) {
    this.father = father;
}
```

```
public int getHeight() {
    return height;
}
public void setHeight(int height) {
    this.height = height;
}
public IPerson getMother() {
    return mother;
}
public void setMother(IPerson mother) {
    this.mother = mother;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
}
```

就可以通过属性注入，配置文件如下：

```
<beans>
<bean name="son" class="Person">
    <property name="name" value="son"/>
    <property name="age" value=10 />
    <property name="father" ref="father" />
    <property name="mother" ref="mother" />
</bean>
<bean name="father" class="Person">
    <property name="name" value="father"/>
    <property name="age" value=38 />
</bean>
<bean name="mother" class="Person">
    <property name="name" value="mother"/>
    <property name="age" value=37 />
</bean>
</beans>
```

这样，在通过容器得到一个son对象的时候，其name、age、father和mother属性都被注入了配置的值了。

同样，该对象申明了构造方法，我们也可以通过构造方法来注入，配置文件如下：

```
<beans>
<bean name="son" class="Person">
    <constructor-arg type=Integer value=10 />
    <constructor-arg type=String value="son" />
    <constructor-arg type=IPerson ref="father" />
```

```
    <constructor-arg type=IPerson ref="mother" />
</bean>
<bean name="father" class="Person">
    <property name="name" value="father"/>
    <property name="age" value=38 />
</bean>
<bean name="mother" class="Person">
    <property name="name" value="mother"/>
    <property name="age" value=37 />
</bean>
</beans>
```

在实际开发中，我们建议最好使用属性注入。

Bean 的属性注入

向 bean 中注入属性，有几种情况，1 注入一个简单对象；2 注入一个复杂对象；3 注入一个 bean。

- **注入一个简单对象：**

直接在 bean 中使用<property name="" value=xxx />即可。

- **注入一个复杂对象：**

在 Disco 的容器中可以注入 List、Map、数组等。

比如注入 List：

```
<bean name="springContainer"
   class="org.springframework.web.context.support.XmlWebApplicati
onContext">
          <property name="configLocations">
             <list>
                <value>WEB-INF/classes/application.xml</value>
             </list>
          </property>
</bean>
```

在类 org.springframework.web.support.XmlWebApplicationContext 里，configLocations 声明为一个 List 对象。

- **注入一个 bean**

在上面的例子中已经看到，注入一个 bean 使用标签 ref，指向一个名字为该 bean 的对象。比如在 Person 例子中，son 这个 bean 在初始化（从容器中）时，father 属性就会去匹配名字为 father 的 bean，然后在将该 father bean 初始化，并注入。

Bean 的类型

在 Disco 中，bean 有四种类型：singleton、prototype、session 和 request。Bean 的类型通过<bean>标签的 type 来指定。其中：

singleton:表示在一个应用上下文中，只存在一个该 bean 的实例。主要用于服务等对象。

prototype：每次请求都重新生成一个新的对象。

session:在一个 session 生命周期内都存在，主要用于比如购物车等对象。

request:在一个请求生命周期内存在，每一次请求重新创建一个新的 bean。

Bean 的注入

Bean 可以通过设置自动注入来自动的完成注入。在<bean>标签中，设置属性 auto 或者 inject 来设置是否自动注入或者手动注入。如果是自动注入，不用配置 property 属性，将根据对象的类型从整个容器中寻找匹配 bean。如果是 inject（默认），则需要通过手动的配置 property 来完成依赖注入。

同时，Disco 提供了@Inject 和@disInject 标签来简化注入配置。这两个标签在应用中大量使用。

@Inject 使用在 bean 之内，比如刚才的 Person 例子，使用@Inject 我们可以重写为:

```
public class Person implements IPerson {
private String name;
private int age;
@Inject(name="father")
private IPerson father;
@Inject(name="mother")
private IPerson mother;
//setter
```

那么，配置文件中就不需要再为 father 或者 mother 配置属性了。

另外，如果我们在配置文件中设置了 auto，那么，在 bean 中不需要注入的对象，就一定要加上@disInject 标签，来表明该属性不需要注入，否则会抛出类型 bean 没有找到的错误。

容器和 Module

在 Disco 中，Module 也是 bean。在其中可以直接添加<property>标签来为 Module 包装的那个 Action 注入服务对象。比如：

```
@Inject(name="personDao")
private GenericDAO<Person> dao;
public void setDao(GenericDAO<Person> dao) {
    this.dao = dao;
}
```

在上面的 CrudAction 示例中，PersonAction 需要一个 DAO 才能工作，在这里我们声明使用的是 GenericDAO<Person>，通过@Inject 标签，Disco 会从超级 IoC 容器中查找名为 personDao 的对象，并注入到这个 Action 中，从而使得我们的 Action 能正常工作。

注意一点，使用 Disco 的容器，一定要在 Disco 框架启动的环境下才能正常使用。

集成其他容器

在 Disco 中，包括中央处理器 RequestProcessor、验证器 Validator、异常处理器 ExceptionHandler 在类的这些底层核心组件，都是通过 Disco 的容器来管理的。因此，你可以非常容易地根据自己的需要，更换 Disco 的一些部件。

下面是在 Disco 容器中加入 Spring 容器的配置：

```
<bean name="springContainer"

    class="org.springframework.web.context.support.XmlWebApplicationC
ontext">
```

```
            <property name="configLocations">
                <list>
                    <value>WEB-INF/classes/application.xml</value>
                </list>
            </property>
        </bean>
        <bean name="innerSpringContainer"
            class="cn.disco.container.impl.SpringContainer">
            <property name="factory" ref="springContainer" />
            </bean>
```

可以在 Spring 容器中配置 Disco 的中央处理器，甚至可以配置事务等，如下面的 Spring 配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
      http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">
    <import resource="jpa-base.xml"/>
    <import resource="service.xml"/>
    <import resource="dao.xml"/>
    <aop:config>
        <aop:pointcut id="discoProcessor"
            expression="execution(*
cn.disco.web.RequestProcessor.process(..))" />
        <aop:advisor advice-ref="txDiscoProcessorAdvice"
            pointcut-ref="discoProcessor" />
    </aop:config>
    <tx:advice id="txDiscoProcessorAdvice"
        transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="*" propagation="REQUIRED" read-only="true"
/>
        </tx:attributes>
    </tx:advice>
    <bean name="Disco-Processor"
class="cn.disco.web.core.DefaultRequestProcessor"/>
    </beans>
```

完成容器的兼容主要是通过适配器模式，有了 Disco 提供了 InnerContainer 接口，你可以实现和你能想到的容器之间的关联。自定义的容器的关联，可以参见 SpringInnerContainer 的实现。

Disco 中的 AOP

AOP 和拦截器

Disco 中的拦截器

Ajax 支持

Disco 内置了对远程 javascript 脚本调用功能，可以使用 javascript 直接访问服务端的业务组件。另外 Disco 通过使用 prototype.js 及其它一些来自开源社区 ajax 特效工具，提供了丰富的 Ajax 支持。

Ajax 概述

Ajax 不是一个技术，它实际上是几种技术，每种技术都有其独特这处，合在一起就成了一个功能强大的新技术。Ajax 包括：

XHTML 和 CSS

使用文档对象模型(Document Object Model)作动态显示和交互 d

使用 XML 和 XSLT 做数据交互和操作

使用 XMLHttpRequest 进行异步数据接收

使用 JavaScript 将它们绑定在一起

传统的 web 应用模型工作起来就象这样：大部分界面上的用户动作触发一个连接到 Web 服务器的 HTTP 请求。服务器完成一些处理---接收数据，处理计算，再访问其它的数据库系统，最后返回一个 HTML 页面到客户端。这是一个老套的模式，自采用超文本作为 web 使用以来，一直都这样用, 但看过《The Elements of User Experience》的读者一定知道，是什么限制了 Web 界面没有桌面软件那么好用。

这种旧的途径让我们认识到了许多技术，但它不会产生很好的用户体验。当服务器正在处理自己的事情的时候，用户在做什么？没错，等待。每一个动作，用户都要等待。
很明显，如果我们按桌面程序的思维设计 Web 应用，我们不愿意让用户总是等待。当界面加载后，为什么还要让用户每次再花一半的时间从服务取数据？实际上，为什么老是让用户看到程序去服务器取数据呢？ Ajax 如何不同凡响.通过在用户和服务器之间引入一个 Ajax 引擎，可以消除 Web 的开始－停止－开始－停止这样的交互过程. 它就像增加了一层机制

到程序中，使它响应更灵敏，而它的确做到了这一点。

不像加载一个页面一样，在会话的开始，浏览器加载了一个 Ajax 引擎---采用 JavaScript 编写并且通常在一个隐藏 frame 中。这个引擎负责绘制用户界面以及与服务器端通讯。Ajax 引擎允许用异步的方式实现用户与程序的交互－－不用等待服务器的通讯。所以用户再不不用打开一个空白窗口，看到等待光标不断的转，等待服务器完成后再响应。

通常要产生一个 HTTP 请求的用户动作现在通过 JavaScript 调用 Ajax 引擎来代替. 任何用户动作的响应不再要求直接传到服务器--例如简单的数据校验，内存中的数据编辑，甚至一些页面导航-引擎自己就可以处理它. 如果引擎需要从服务器取数据来响应用户动作-假设它提交需要处理的数据，载入另外的界面代码，或者接收新的数据--引擎让这些工作异步进行，通常使用 XML, 不用再担误用户界面的交互。

远程脚本调用

web 脚本远程调用：在基于 Web2.0 的程序中，在用户注册的时候，我们希望用户在输入完注册用户名后，假如其输入的用户已经存在，则立即显示相应的提示，这样的交互会使得应用程序交互界面变得更加友好。要实现这种功能，可以通过在用户输入完用户名时，触发一个事件，这个事件执行一个程序，自动到服务器端检测这个用户名是否存在，若用户已经存在，则给予相应的提示，让用户可以及时选择其它用户名继续操作。假如我们在服务器端有一个用户处理组件 UserService，这个组件中有一个检测用户是否存在的方法 boolean checkUserExists(String userName)，这个方法用来检测用户名是否存在，若存在则返回 true，否则返回 false。

引入远程脚本调用，则可以直接在注册页面中使用下面的 javascript 脚本来判断用户是否存在:

```
function checkUserExist(username)
{
UserService.checkUserExists(username,function(ret){
if(ret)Element.setValue('userName_Msg',"用户名已存在,请选择其它用户名\!");
})
}
```

而调用这个函数的是用户名录入框的 onChange 事件，大致如下：

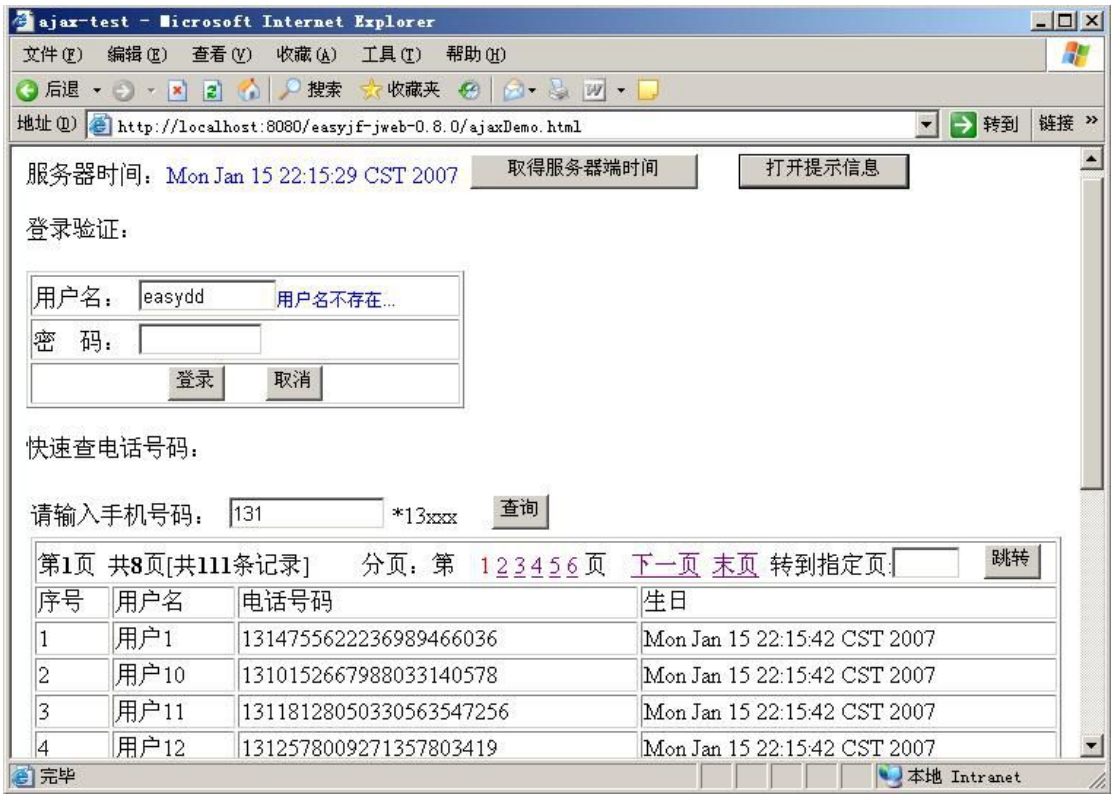
```
<input name="userName" type="text" id="userName" size="10"
onFocus="$('userName_Msg').innerText='';"
onChange="checkUserExist(this.value);">
<span id="userName_Msg" style="color:#0000FF; font-size:12px"></span>
```

这种模式即为远程脚本调用。在上面的代码中，在 checkUserExist 函数中，调用了服务器端的 UserService.checkUserExists(userName)方法，来判断用户名是否存在，若返回的结果为 true，则在 id 为 userName_Msg 的 span 中显示用户存在的提示。

在 Disco 中，内置了一个把服务器业务组件暴露给客户端的通过 Javascript 远程调用的引擎，因此可以像上面的方式轻松在 web 界面中通过 javascript 调用服务器组件，实现特定的功能，这就是我们要说的远程脚本调用。

快速上手

在 Disco 的 demo 中，提供了一个关于 Disco 中使用 Ajax 应用的 Demo，名为 ajaxDemo.html。你只需要下载最新的 Disco 源代码，然后执行 bin 目录中的 build war，即可得到一个可运行的 Web 应用包，把这个 war 包拷到 Tomcat 的 webapps 目录下，启动 web 服务器。然后在地址栏中输入 http://localhost:8080/disco-jweb-0.8.0/ajaxDemo.html,即可看到 Disco 中 Ajax 运用的一些效果。大致如下图所示，详情参考视频教程:



要在 Disco 应用程序中使用 Ajax 功能，需要下面几个步骤：

1、在 web.xml 文件添加如下的 mapping;

```
<servlet-mapping>
<servlet-name>disco</servlet-name>
<url-pattern>/ejf/*</url-pattern><!--所有/ejf/*样式的 url 都交由 Disco 来处理-->
</servlet-mapping>
```

2、在模板页面(或客户端 html 页面)中加入下面的两行：

```
<script type='text/javascript' src="ejf/discoajax/prototype.js"></script>
<script type='text/javascript' src='ejf/discoajax/engine.js'></script>
```

3、在 disco-web.xml 文件中配置需要暴露给客户端的业务对象;

```
<ajax>
<services allowName="*Service" denyName="">
  <service name="UserService">
  <include method=""></include>
  <exclude method=""></exclude>
  </service>
  <service name="ServerDate">
  </service>
```

```
    </services>
</ajax>
```

4、在模板页面(或客户端 html 页面)中通过下面的方式引用服务器端支持远程脚本访问的业务对象

```
<script type='text/javascript' src='ejf/discoajax/UserService.js'></script>
<script type='text/javascript' src='ejf/discoajax/ServerDate.js'></script>
```

5、在模板页面(或客户端 html 页面)中书写支持无刷新的远程脚本调用代码，如下所示：

```
<input type="submit" name="Submit2" value="登录" onClick="login();">function login()
{
UserService.login($('userName').value,$('password').value,function(ret)
{
if(ret)alert("登录成功!");
});
}
```

配置 Ajax

安全控制

Ajax 工具

Ajax 验证