
DWR 中文文档 v0.9

DWR 2.0

方佳玮 编著

部分原创/部分整理/部分翻译

版权声明

本书目前仅发行网络版，完全免费，转载请注明作者信息。任何出版社或个人未经作者允许不得出版印刷。

另外如果发现本人的部分内容有所抄袭，请不要来找我理论，我本来做的就是公益事情。

版权所有，侵权必究。

参考网站及资料

DWR 官方网站	http://getahead.ltd.uk/dwr/
JavaScud Wiki	http://wiki.javascud.org/display/dwrcn/Home
IBM 中国	http://www.ibm.com/developerworks/cn/

前言

Ajax 向我们袭来时，很多写代码的程序员看到了 Ajax 的发展前景，但并不是每一个程序员都能将页面与代码完美整合在一起，DOM、CSS、javascript 让人眼花缭乱，不知从何下手。

本书的读者必须有一定的 Jsp, JavaScript, 应用服务器（比如 Tomcat）的基础和使用经验，否则请在看此教程前先了解一下。当然附录里有一些 JavaScript 的教程。

本书可以当作一本 DWR 完整的教程，也可以当作一本详细介绍 DWR 的“词典”，我的目的只是通过本书，希望您能够了解一些 DWR 的基本知识、常用的用户界面组件、远程方法调用等。并能够搭建 DWR 开发环境，实现 DWR 的快速开发。

非常感谢 [JavaScud Wiki 网站](#)，省去了我很多翻译时间，同时感谢网站的几位翻译人员。

由于本人也刚刚接触 DWR，书中难免会有一些错误和表达不是太好的地方，请读者谅解并提出您的宝贵意见。我很希望得到读者对本书的评价和建议。您可以把您在学习本书的过程中所遇到的问题和建议发送到我的邮箱 jorwen.fang@gmail.com，以便我对本书下一个版本的更新，我会在第一时间给您回复。

感谢您阅读本书！希望这本书对您来说是一本有用的书。我是上海华东理工大学 03 届计算机（金山）专业的方佳玮。

方佳玮
2007-01-22

目录

第 1 章. DWR 入门.....	8
1.1 简介.....	8
1.2 第一个 DWR 程序: Hello World.....	9
1.2.1 将 DWR 放入你的工程	9
1.2.2 编辑配置文件	9
1.2.3 编写 service	10
1.2.4 测试 DWR	10
1.2.5 编写一个 jsp.....	11
1.3 本章总结.....	12
第 2 章. web.xml 配置	13
2.1 主要配置.....	13
2.2 常用<init-param>参数列表	14
2.2.1 安全参数	14
2.2.2 Ajax 服务器加载时保护参数	14
2.2.3 其他参数	15
2.3 日志配置.....	16
2.4 多个 dwr.xml 配置和 J2EE 角色定义	16
2.5 插件配置.....	18
2.6 测试模式配置	19
第 3 章. dwr.xml 配置	20
3.1 纵览.....	20
3.2 <init>标签	20
3.3 <allow>标签	21
3.3.1 Creator.....	21
3.3.2 Converter	25
3.4 <signatures>标签	30
第 4 章. 整合	32
4.1 DWR 与 Servlet.....	32
4.1.1 使用 webContext 的方法:	32
4.1.2 方法选择	32
4.2 DWR 与 Spring.....	34
4.2.1 让 DWR 和 Spring 一起工作的检查列表.....	34
4.2.2 Spring Creator	34
4.2.3 找到 Spring 配置文件.....	34
4.2.4 使用 Spring 配置 DWR.....	35
4.3 DWR 与 JSF.....	36
4.3.1 JSF Creator	36
4.3.2 Servlet Filter	36
4.4 DWR 与 Struts.....	37
4.4.1 Struts creator	37

4.4.2	开始顺序	37
4.5	DWR 与 Weblogic 或 PageFlow	38
4.6	DWR 与 Hibernate	39
4.6.1	让 DWR 和 Hibernate 一起工作的检查列表.....	39
4.6.2	HibernateBeanConverter	39
4.6.3	Session 管理.....	39
4.7	DWR 与 WebWork	40
4.7.1	配置 dwr.xml	40
4.7.2	在 JSP 中导入脚本.....	40
4.7.3	高级	41
4.8	DWR 与 Acegi	42
4.8.1	问题提出	42
4.8.2	解决方案	42
第 5 章.	DWR 中的 JavaScript 简介	44
5.1	简单的回调函数	44
5.2	调用元数据对象	45
5.3	查找回调函数	45
5.4	创建一个与 Java 对象匹配的 Javascript 对象	46
第 6 章.	engine.js 功能	47
6.1	使用选项.....	47
6.2	选项索引.....	48
6.2.1	处理器(Handler)	48
6.2.2	调用处理器(Call Handler)	48
6.2.3	Hooks (一个 batch 中可以注册多个 hook)	49
6.2.4	全局选项(在单次调用或者批量调用中不可用)	49
6.2.5	废弃的选项.....	49
6.2.6	未来版本的选项	49
6.3	选项说明.....	50
6.3.1	批量调用	50
6.3.2	顺序调用	50
6.3.3	错误警告和超时	50
6.3.4	远程调 Hooks	51
6.3.5	远程调用选项	51
第 7 章.	util.js 功能	54
7.1	\$()	54
7.2	addOptions and removeAllOptions	54
7.3	addRows and removeAllRows	55
7.4	getText	56
7.5	getValue.....	56
7.6	getValues	57
7.7	onReturn	57
7.8	selectRange	57
7.9	setValue.....	58
7.10	setValues	58

7.11	toDescriptiveString.....	58
7.12	useLoadingMessage.....	58
7.13	Submission box.....	61
第 8 章.	DWR 进阶.....	63
8.1	DWR Annotations.....	63
8.1.1	初始配置	63
8.1.2	远程访问类.....	63
8.1.3	对象转换	64
8.2	错误和异常处理.....	64
8.2.1	错误处理	64
8.2.2	异常	65
8.2.3	找出更多的信息	65
8.3	传递额外的数据到 callback 函数	66
8.4	从其他的 URL 读取数据	67
8.5	安全.....	68
第 9 章.	范例精讲.....	71
9.1	购物车	71
9.1.1	介绍	71
9.1.2	实现目录	72
9.1.3	测试部署	74
9.1.4	调用远程对象	75
9.1.5	实现购物车.....	77
9.1.6	调用远程的 Cart 方法	79
9.1.7	演示结果	81
9.1.8	总结	81
第 10 章.	附录.....	83
10.1	常见问题.....	83
10.1.1	TransformerFactoryConfigurationError	83
10.1.2	XML 解析错误	83
10.1.3	使用 weblogic 的类路径问题	83
10.1.4	没有 cookies 的情况下用 DWR	84
10.2	JavaScript 高级应用	85
10.2.1	用变量操纵函数	85
10.2.2	高阶函数	86
10.2.3	动态类型	87
10.2.4	灵活的对象模型	89
10.2.5	本节总结	91

第1章. DWR 入门

1.1 简介

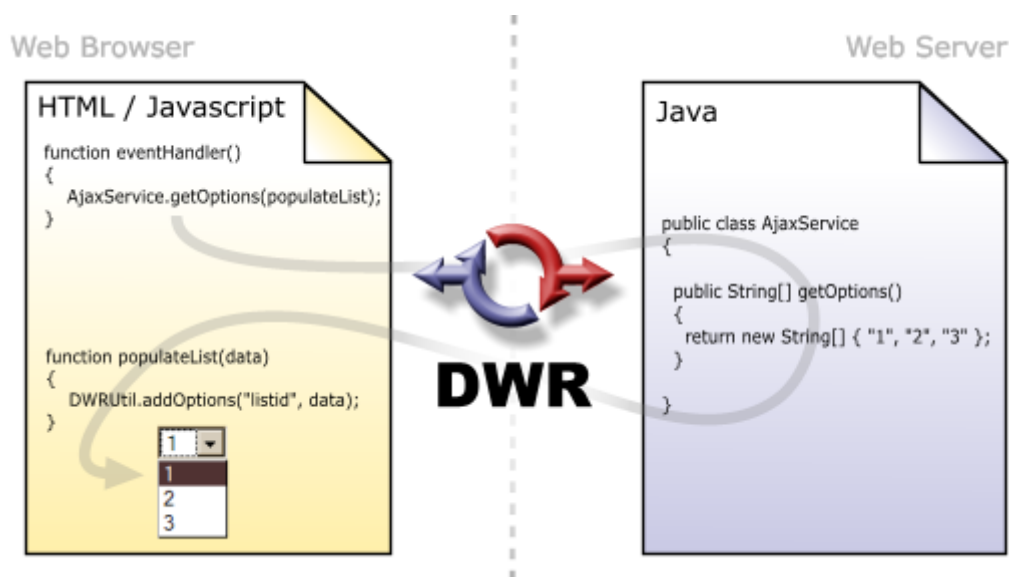
DWR 是一个可以允许你去创建 AJAX WEB 站点的 JAVA 开源库。它可以让你在浏览器中的 Javascript 代码调用 Web 服务器上的 Java 代码，就像在 Java 代码就在浏览器中一样。

DWR 包含 2 个主要部分：

- 一个运行在服务器端的 Java Servlet，它处理请求并且向浏览器发回响应。
- 运行在浏览器端的 JavaScript，它发送请求而且还能动态更新网页。

DWR 工作原理是通过动态把 Java 类生成为 Javascript。它的代码就像 Ajax 魔法一样，你感觉调用就像发生在浏览器端，但是实际上代码调用发生在服务器端，DWR 负责数据的传递和转换。这种从 Java 到 JavaScript 的远程调用功能的方式使 DWR 用起来有种非常像 RMI 或者 SOAP 的常规 RPC 机制，而且 DWR 的优点在于不需要任何的网页浏览器插件就能运行在网页上。

Java 从根本上来讲是同步机制，然而 AJAX 却是异步的。所以你调用远程方法时，当数据已经从网络上返回的时候，你要提供有反调（callback）功能的 DWR。



这个图片显示了 DWR 如何选择一下拉列表的内容作为 JavaScript 的 onclick 事件的结果。

DWR 动态在 JavaScript 里生成一个 AjaxService 类，去匹配服务端代码。由 eventHandler 去调用它，然后 DWR 处理所有的远程细节，包括倒置（converting）所有的参数以及返回 Javascript 和 Java 之间的值。在示例中，先在 eventHandler 方法里调用 AjaxService 的 getOptions() 方法，然后通过反调（callback）方法 populateList(data) 得到返回的数据，其中 data 就是 String[]{"1", "2", "3"}，最后再使用 DWR utility 把 data 加入到下拉列表。

好了，DWR 介绍完了，现在大家肯定很想知道如何做出第一个 DWR 吧！然后我们在下一章节以一个 HelloWorld 示例带领大家入门。

1.2 第一个 DWR 程序：Hello World

有 2 中方法可以帮助你入门 DWR，一个方法是去[下载](#) WAR 文件并且去完整看一下代码，但是这样并不能帮助你发现 DWR 是如何简单地集成到你当前地 WEB 应用，所以以下几个简单地步骤推荐看一下：

1.2.1 将 DWR 放入你的工程

- 1) 从官方网站[下载](#) dwr.jar 包。然后将它放在你 webapp 的 WEB-INF/lib 目录下。
- 2) 将下载的 dwr-版本号-src.zip \java\org\directwebremoting 内的 engine.js 和 util.js 放入 WEB 应用中，比如 js 文件夹下。

1.2.2 编辑配置文件

1. web.xml

以下几行代码必须被添加到 WEB-INF/web.xml 文件中。注意，要把<servlet>和其他<servlet>放在一起，<servlet-mapping>要和其他<servlet-mapping>放在一起

```
<servlet>
  <servlet-name>dwr-invoker</servlet-name>
  <servlet-class> org.directwebremoting.servlet.DwrServlet </servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>true</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>dwr-invoker</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

2. dwr.xml

在 web.xml 的同一目录下，创建 dwr.xml，并且将要被调用的 java 类写入其中。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting
2.0//EN" "http://www.getahead.ltd.uk/dwr/dwr20.dtd">

<dwr>
  <allow>
    <create creator="new" javascript="service">
      <param name="class" value="helloWorld.Service" />
    </create>
  </allow>
</dwr>
```

1.2.3 编写 service

就像没有 dwr 一样，写一个简单类并加一个方法是

```
package helloWorld;

public class Service {
    public String sayHello(String yourName) {
        //可以是访问数据库的复杂代码
        return "Hello World " + yourName;
    }
}
```

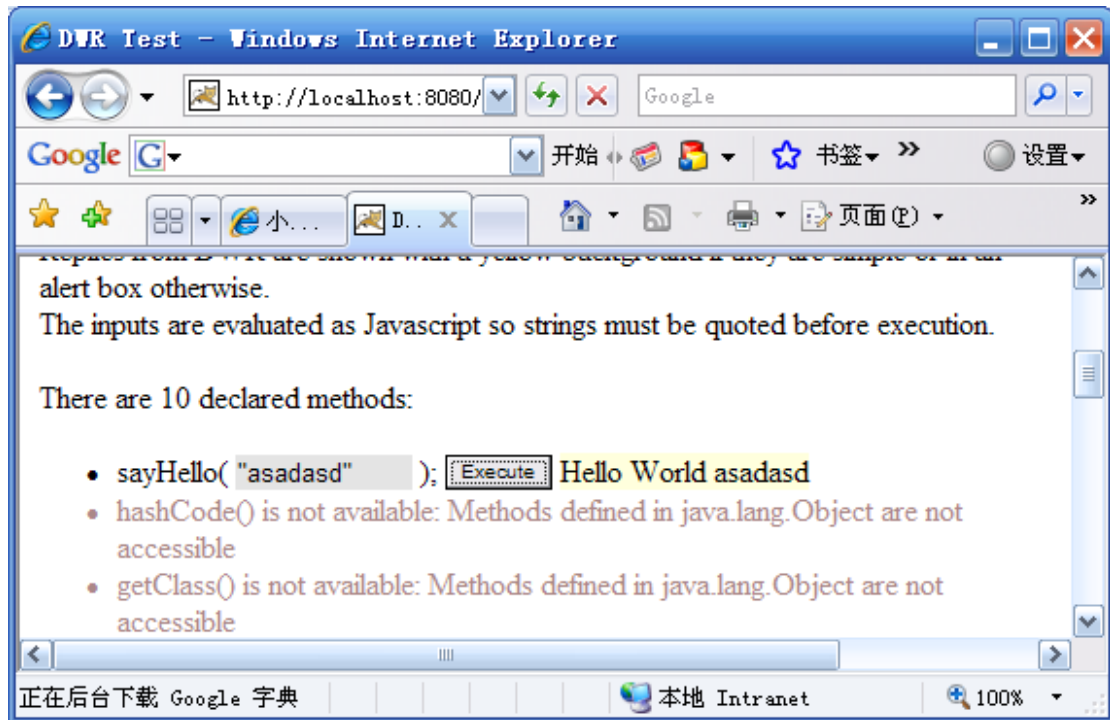
1.2.4 测试 DWR

将代码放入应用服务器（比如 Tomcat），启动。

然后在地址栏输入 <http://localhost:8080/你的工程/dwr>



然后点击 service，会看到刚才写的 sayHello()的方法，输入自己的名字然后点击“Execute”，如果发现确实是正确的返回结果，说明测试通过了，可以进入下一步了。



1.2.5 编写一个 jsp

接下来最后一步就是创建一个 jsp 文件

- 1) 要引用几个 dwr 的 js, 其中 engine.js 必须要, 如果需要用到 dwr 提供的一些方便的工具要引用 util.js
- 2) 然后还要引用 dwr 自动生成的 js, dwr/interface/service.js, 注意 js 名字要和 dwr.xml 配置的一样
- 3) js 里的 service.sayHello 和 java 类的那个有一点区别, 多了个参数, 用来 callback 返回的数据

```
<%@ page language="java" pageEncoding="UTF-8"%>

<html>
<head>
  <title>My JSP 'first_dwr.jsp' starting page</title>
  <script type='text/javascript' src='js/util.js'></script>
  <script type='text/javascript' src='js/engine.js'></script>
  <script type='text/javascript' src='dwr/interface/service.js'>
</script>
  <script type="text/javascript">
    function firstDwr(){
      service.sayHello("Jorwen",callBackHello);
    }
    function callBackHello(data){
      alert(data);
    }
  </script>
</head>
```

```
<body>
<input type="button" name="button" value="测试" onclick="firstDwr()">
</body>
</html>
```

地址栏输入 http://localhost:8080/你的工程/first_dwr.jsp

显示的结果如下:



1.3 本章总结

相信看了此章节,大家一般都能做出这个实例来,也算是 DWR 刚入门了,在以后的教程里将详细介绍 DWR 各个功能。帮助大家能开发出任何 Ajax 需求的功能来。更多进阶的例子可以参考[范例精讲](#),您也可以通过看范例学习 DWR 然后有疑问再查看该文档的相关章节。

第2章. web.xml 配置

2.1 主要配置

要加入到你的 web.xml 最少的代码就是简单地去申明 DWR servlet，没有它 DWR 就不起作用。

```
<servlet>
  <servlet-name>dwr-invoker</servlet-name>
  <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>dwr-invoker</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

在 DWR2.x 里,DwrServlets 是类 org.directwebremoting.servlet.DwrServlet ，尽管 uk.ltd.getahead.dwr.DWRServlet 仍然可以用。在 DWR 1.x 你不得不使用后者。

有些额外的 servlet 参数，在有些地方很重要。尤其 debug 参数

这个扩展 DWR 的标准结构是使用<init-params>。放在<servlet>内，就像如下使用

```
<servlet>
  <servlet-name>dwr-invoker</servlet-name>
  <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>true</param-value>
  </init-param>
</servlet>
```

另外，启动服务时，如果报如下错。

```
java.lang.IllegalArgumentException: DefaultContainer can't find a
classes
```

是 DWR2.0 加入了 JDK5 的注释(annotations).DwrServlet 初始化的时候会去检查注释的类,找不到就抱错了。如果你不用 annotations 也可以忽略掉这个错误。不过看起来总是不爽。解决方案如下

```
<servlet>
  <servlet-name>dwr-invoker</servlet-name>
  <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
  <init-param>
```

```
<param-name>debug</param-name>
<param-value>true</param-value>
</init-param>
<init-param>
  <param-name>classes</param-name>
  <param-value>java.lang.Object</param-value>
</init-param>
</servlet>
```

2.2 常用<init-param>参数列表

2.2.1 安全参数

allowGetForSafariButMakeForgeryEasier

开始版本: 2.0

默认值: false

描述: 设置成 true 使 DWR 工作在 Safari 1.x, 会稍微降低安全性。

crossDomainSessionSecurity

开始版本: 2.0

默认值: true

描述: 设置成 false 使能够从其他域进行请求。注意, 这样做会在安全性上有点冒险, 参考一下这篇[文章](#), 在没有理解这个后果前不要设置成为 false。

debug

开始版本: 1.0

默认值: false

描述: 设置成 true 使 DWR 能够 debug 和进入测试页面

scriptSessionTimeout

开始版本: 2.0

默认值: 1800000(30 分钟)

描述: script session 的超时设置

maxCallCount

开始版本: 2.0rc2 和 1.1.4

默认值: 20

描述: 一次批量(batch)允许最大的调用数量。(帮助保护 Dos 攻击)

2.2.2 Ajax 服务器加载时保护参数

pollAndCometEnabled

开始版本: 2.0

默认值: false

描述: 设置成 true 能增加服务器的加载能力, 尽管 DWR 有保护服务器过载的机制。

maxWaitingThreads

开始版本: 2.0

默认值: 100

描述: 最大等待线程数量。

preStreamWaitTime

开始版本: 2.0

默认值: 29000(单位: 毫秒)

描述: 对一个打开流前的反应, 等待的最大时间

postStreamWaitTime

开始版本: 2.0

默认值: 1000(单位: 毫秒)

描述: 对一个打开流后的反应, 等待的最大时间

2.2.3 其他参数

ignoreLastModified

开始版本: 2.0

默认值: false

描述: 默认值支持最后修改, 这样就允许服务器端对客户端请求较少资源。设置为 true 就能屏蔽支持。

scriptCompressed

开始版本: 1.1

默认值: false

描述: DWR 能够执行简单的压缩, 设置为 true 可以激活此功能。另外还有一个未公开的有关系的重要参数 “compressionLevel”, 此参数允许你配置压缩类型。查看[这里](#)得到更多详细资料。

sessionCookieName

开始版本: 2.0

默认值: JSESSIONID

描述: DWR 通过检查文档和提取当前 session ID 支持 URL 重写。一些 servlet 引擎使用非标准的 cookie 名。参数允许你改变默认值。

welcomeFiles

开始版本: 2.0

默认值: index.html, index.htm, index.jsp

描述: 类似于 web.xml 的<welcome-file-list>标签

2.3 日志配置

DWR 工作在 JDK1.3 中不支持 `java.util.logging`，但我们并不强迫任何人都去使用 `commons-logging` 或者 `log4j`，所以在使用 `HttpServlet.log()` 方法时 DWR 将正常工作，如果没有日志类的话。然而如果 DWR 可以使用，那么它将使用日志。

Commoms-Logging

由于大多数 `servlet` 容器都使用它，几乎每个人都将使用 `commons-logging`。所以如果你的 `webapp` 不明确使用 `commons-logging`，它将被默认设为可以使用。

在这些日志将被一些配置文件所约束，比如 `java.util.logging` 或者 `log4j`，可以去查看他们各自的文档获得详情。

HttpServlet.log()

如果你正在使用 `HttpServlet.log()`，以下的代码用来控制 DWR 日志

```
<init-param>
  <param-name>logLevel</param-name>
  <param-value>DEBUG</param-value>
</init-param>
```

值可以是：FATAL，ERROR，WARN(默认)，INFO，DEBUG

2.4 多个 `dwr.xml` 配置和 J2EE 角色定义

一般来说只需要一个 `dwr.xml` 文件，并且会被设为默认位置 `WEB-INF/dwr.xml`。所以你不需要配置。

有 3 个原因说明你为何需要指定不同位置的 `dwr.xml` 文件：

- 你想保持 `dwr.xml` 的文件请参照下面的例子

```
<param-value>WEB-INF/classes/com/yourco/dwr/dwr.xml</param-value>
```

- 你可以有很多的远程方法类并且希望指定文件。在这个例子里将指定不同文件开始配置，不同的 `param-name` 将重复多次 DWR 将轮流读取它们。
- DWR 能够使 J2EE URL 具有给与不同用户组不同权限的安全机制。通过起不同名字,URL 和权限。

就像如下例子去使用

```
<init-param>
  <param-name>config*****</param-name>
  <param-value>WEB-INF/dwr.xml</param-value>
  <description>What config file do we use?</description>
</init-param>
```

用一个字符串"config" 作为开始，设置 param-name，每个 param-name 必须不同。

```
<servlet>
  <servlet-name>dwr-user-invoker</servlet-name>
  <servlet-class>
    org.directwebremoting.servlet.DwrServlet
  </servlet-class>
  <init-param>
    <param-name>config-user</param-name>
    <param-value>WEB-INF/dwr-user.xml</param-value>
  </init-param>
</servlet>
<servlet>
  <servlet-name>dwr-admin-invoker</servlet-name>
  <servlet-class>
    org.directwebremoting.servlet.DwrServlet
  </servlet-class>
  <init-param>
    <param-name>config-admin</param-name>
    <param-value>WEB-INF/dwr-admin.xml</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>dwr-admin-invoker</servlet-name>
  <url-pattern>/dwradmin/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>dwr-user-invoker</servlet-name>
  <url-pattern>/dwruser/*</url-pattern>
</servlet-mapping>

<security-constraint>
  <display-name>dwr-admin</display-name>
  <web-resource-collection>

    <web-resource-name>dwr-admin-collection</web-resource-name>
    <url-pattern>/dwradmin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <display-name>dwr-user</display-name>
```

```
<web-resource-collection>
  <web-resource-name>dwr-user-collection</web-resource-name>
  <url-pattern>/dwruser/*</url-pattern>
</web-resource-collection>
<auth-constraint>
  <role-name>user</role-name>
</auth-constraint>
</security-constraint>
```

2.5 插件配置

大多数 DWR 的功能是可以作为插件的，所以就通过替换默认类可以改变 DWR 的功能。你可以通过包含 `<init-param>` 去覆盖默认的实现。举个例子：

```
<init-param>
  <param-name>
    org.directwebremoting.extend.ServerLoadMonitor
  </param-name>
  <param-value>com.example.MyCustomServerLoadMonitor</param-value>
</init-param>
```

没有规定我们越过主要的版本丢弃这些接口，但是我们将尝试提供简单的升级路径。

DWR2.0 插件

- org.directwebremoting.Container
- org.directwebremoting.WebContextFactory.WebContextBuilder
- org.directwebremoting.ServerContextFactory.ServerContextBuilder
- org.directwebremoting.servlet.UrlProcessor
- org.directwebremoting.extend.AccessControl
- org.directwebremoting.extend.AjaxFilterManager
- org.directwebremoting.extend.ConverterManager
- org.directwebremoting.extend.CreatorManager
- org.directwebremoting.extend.DebugPageGenerator
- org.directwebremoting.extend.HtmlCallMarshaller
- org.directwebremoting.extend.HtmlPollHandler
- org.directwebremoting.extend.PageNormalizer
- org.directwebremoting.extend.PlainCallMarshaller
- org.directwebremoting.extend.PlainPollHandler
- org.directwebremoting.extend.Remoter
- org.directwebremoting.extend.ScriptSessionManager
- org.directwebremoting.extend.ServerLoadMonitor

默认的实现大多数在 org.directwebremoting.impl 包，细节是在 ContainerUtil.setupDefaults() 指定的。

DWR1.1 插件

- uk.ltd.getahead.dwr.AccessControl
- uk.ltd.getahead.dwr.Configuration
- uk.ltd.getahead.dwr.ConverterManager
- uk.ltd.getahead.dwr.CreatorManager
- uk.ltd.getahead.dwr.Processor
- uk.ltd.getahead.dwr.ExecutionContext

默认的实现大多数在 `uk.ltd.getahead.dwr.impl` 包

2.6 测试模式配置

通过添加如下参数，设置 `debug` 测试模式

```
<init-param>
  <param-name>debug</param-name>
  <param-value>true</param-value>
</init-param>
```

在 `debug` 模式里，DWR 将为每个 `allow` 的类(请看下面的 `dwr.xml` 配置章节)生成测试页面。这些能变得非常有用帮助了解 DWR 能做什么和如何工作。这个模式也能警告你以防止 `javascript` 的保留字，或者重载问题。

然而这个模式不应该被用在现场部署，因为他能给黑客或者攻击者许多关于服务器的详细信息。

第3章. dwr.xml 配置

3.1 纵览

dwr.xml 是你用来配置 DWR 的文件，默认是将其放入 WEB-INF 文件夹。

创建一个 dwr.xml 文件

dwr.xml 有如下的结构：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting
2.0//EN" "http://www.getahead.ltd.uk/dwr/dwr20.dtd">

<dwr>
  <!-- 仅当需要扩展DWR时才需要 -->
  <init>
    <creator id="..." class="..." />
    <converter id="..." class="..." />
  </init>

  <!-- 没有它DWR什么也做不了 -->
  <allow>
    <create creator="..." javascript="..." />
    <convert converter="..." match="..." />
  </allow>

  <!-- 有必要告诉DWR方法签名 -->
  <signatures>...</signatures>
</dwr>
```

3.2 <init>标签

这个初始化部分申明被用来创建远程 **beans** 而且这个类能被用来以某种过程转换。大多数例子你将不需要用它，如果你想去定义一个新的 **Creator** 或者 **Converter**，就要在此被申明。

在 **init** 部分里有了定义只是告诉 DWR 这些扩展类的存在，给出了如何使用的信息。这时他们还没有被使用。这种方式很像 Java 中的 **import** 语句。多数类需要在使用前先 **import** 一下，但是只有 **import** 语句并不表明这个类已经被使用了。每一个 **creator** 和 **converter** 都用 **id** 属性，以便后面使用。

3.3 <allow>标签

allow 部分定义了 DWR 能够创建和转换的类。

3.3.1 Creator

每一个在类中被调用的方法需要一个<create ...>有若干类型的 creator，使用“new”关键字或者 Spring 框架等。

create 元素是如下的结构

```
<allow>
  <create creator="..." javascript="..." scope="...">
    <param name="..." value="..." />
    <auth method="..." role="..." />
    <exclude method="..." />
    <include method="..." />
  </create>
  ...
</allow>
```

1. creator 属性

1). new: Java 用“new”关键字创建对象

是 DWR 默认的 creator，如下所示

```
<create id="new" class="org.directwebremoting.create.NewCreator"/>
```

没有必要把它加入 dwr.xml，它已经在 DWR 内部文件了。

这个 creator 将使用默认构造器创建类的实例，以下是用 new 创建器的好处

- 安全:DWR 创造的对象生存的时间越短，多次调用中间的值不一致的错误机会越少。
- 内存消耗低: 如果你的站点用户量非常大，这个创造器可以减少 VM 的内存溢出。

2). none: 它不创建对象，看下面的原因。(v1.1+)

none 创建器不创建任何对象，它会假设你不须要创建对象。有 2 个使用的原因:

- 你可能在使用的 scope 不是"page"(看上面)，并在在前面已经把这个对象创建到这个 scope 中了，这时你就不需要再创建对象了。
- 还有一种情况是要调用的方法是静态的，这时也不需要创建对象。DWR 会在调用创建器之前先检查一下这个方法是不是静态的。

对于上述两种情况，你仍然需要 class 参数，用来告诉 DWR 它是在操作的对象类型是什么。

3). **scripted**: 通过 BSF 使用脚本语言创建对象, 例如 BeanShell 或 Groovy。
要使用这个创造器, 你需要把一些辅助库放到 WEB-INF/lib 文件夹下: 比如 [BSF 的 jar 包](#), 你要用的 [脚本语言的 jar 包](#)。

new 创造器在 DWR 中已经默认声明了:

```
<creator id="script" class="uk.ltd.getahead.dwr.create.ScriptedCreator"/>
```

这个创造器用 BSF 来执行脚本得到 Bean, 例如:

```
<allow>
...
<create creator="script" javascript="EmailValidator">
  <param name="language" value="beanshell" />
  <param name="script">
    import org.apache.commons.validator.EmailValidator;
    return EmailValidator.getInstance();
  </param>
</create>
...
</allow>
```

script 创造器有如下参数:

参数	DWR 版本	描述
language	1.0	脚本语言, 字符串, 例如'beanshell'. (必需)
script	1.0	要执行的脚本。 (必需, 除非 scriptPath 参数存在)
scriptPath	1.1	脚本文件路径。 (必需, 除非 script 参数存在)
reloadable	1.1	是否检测脚本文件的改动, 以重新加载 (可选, 默认 true)
class	1.0	创造出对象的类型(可选). 如果没有 DWR 通过创造器得到类型。

注意:

当一个类是用 **script** 创造出来的, 并且 **scope** 是 **session** 或 **application**, 如果你的脚本改变, **session** 中的类和 **script** 中的类就不一致了。这样会出现错误。虽然 web 容器不用重启, 但是用户需要先登出(或以某种方式清空 **session**), 然后再登录。

当 **clazz** 参数不为空, 并且用来创造新实例, DWR 简单的调用 **class.newInstance()** 方法。这种方法是没问题的, 除非脚本正在用某个参数创建一个类, 或者调用某个函数来配置这个类。不幸的是, 每次请求都要重新运行 **script** 并造成上面的问题。

4). spring: 通过 Spring 框架访问 Bean。

[详情请见 DWR 与 Spring 整合](#)

5). jsf: 使用 JSF 的 Bean。 (v1.1+)

[详情请见 DWR 与 JSF 整合](#)

6). struts: 使用 Struts 的 FormBean。 (v1.1+)

[详情请见 DWR 与 Struts 整合](#)

7). pageflow: 访问 Weblogic 或 Beehive 的 PageFlow。 (v1.1+)

[详情请见 DWR 与 Weblogic 或 Beehive 的 PageFlow 整合](#)

8). ejb3: 使用 EJB3 session bean。 (v2.0+)

一个正在实验的创造器，用来访问 EJB Session beans。直到进行更多的测试和正式的维护，否则还不能作为产品被使用。

如果你想写自己的 creator，你必须在<init>里注册它。

2. javascript 属性

在浏览器里给你创建的对象命名。避免使用 JavaScript 保留字。这个名字将在页面里作为 js 被导入，就像第 2 章节的那个 jsp:

dwr.xml

```
<create creator="new" javascript="service">
  <param name="class" value="helloWorld.Service" />
</create>
```

html / jsp

```
<html>
  <head>
  ...
    <script type='text/javascript' src='dwr/interface/service.js'>
  ...
```

3. scope 属性

和定义在 servlet 的 scope 一样大的范围，它允许你指定哪个 bean 是可以获得的。选项可以是: application, session, request 和 page。这些值应该已经被开发者们熟悉了。

scope 选项是可选的，默认为 page，使用 session 请求 cookies。目前，DWR 还不支持 URL 重写。

4. param 元素

被用来指定创造器的其他参数，每种构造器各有不同。例如，"new"创造器需要知道要创建的对象类型是什么。每一个创造器的参数在各自的文档中能找到。

5. include 和 exclude 元素

允许一个创造器去限制进入类的方法。一个创造器必须指定 **include** 列表或 **exclude** 列表之一。如果是 **include** 列表则暗示默认的访问策略是"拒绝"，**include** 中的每个方法就是允许访问的方法；如果是 **exclude** 列表则暗示默认的访问策略是"允许"，**exclude** 中的每个方法就是拒绝访问的方法。

比如：

```
<create creator="new" javascript="Fred">
  <param name="class" value="com.example.Fred" />
  <include method="setWibble" />
</create>
```

说明你只能在 DWR 中使用 Fred 的是 setWibble 方法。

6. auth 元素

允许你指定一个 J2EE 的角色作为将来的访问控制检查：

```
<create creator="new" javascript="Fred">
  <param name="class" value="com.example.Fred" />
  <auth method="setWibble" role="admin" />
</create>
```

7. 使用静态方法

DWR 会在调用创建器之前先检查一下这个方法是不是静态的，如果是那么创造器不会被调用。很显然这个逻辑适用于所有创造器，尽管如此"null"创造器是最容易配置的。

8. 使用单例类

对于单例类的创建，最好适用 BeanShell 和 BSF 来实例化对象。请参考 [scripted](#) 创造器。

9. DWR 与 HttpSessionBindingListeners

DWR1.x 中存贮已经创造的 Bean 的方法需要注意，它在每次请求时都会调用相同的 setAttribute() 方法。就是说，如果一个 Bean 在 dwr.xml 中的声明周期设置为 session，再每次调用 bean 中的方法时，DWR 都会执行一次 session.setAttribute(yourBean)。这看上去没有什么危害，但是如果你要使用 servlet 的事件机制的，就是说用了 HttpSessionBindingListener 接口，你就会发现 valueBound 和 valueUnbound 事件在每次调用时都会发生，而不是你想像的在 bean 被创建时以及 session 过期时。

DWR2 只在第一次创建对象时调用 setAttribute() 。

3.3.2 Converter

我们需要确认所有的参数能被转换。许多 JDK 提供的类型使你能够使用，但是你如果要转换你自己的代码，就必须告诉 DWR。一般是指 JavaBean 的参数需要一个<convert...>标签作为入口。



你不需要在 dwr.xml 中<allow>部分的<convert>中定义。它们默认支持。

- 所有主要的类型，boolean, int , double 等等。
- 包装类，Boolean, Integer 等等。
- java.lang.String
- java.util.Date 和 java.sql.Timestamp, java.sql.Date。
- 数组(存放以上类型的)
- 集合类型 (List, Set, Map, Iterator 等等) (存放以上类型的)
- DOM 对象(来自于 DOM, XOM, JDOM 和 DOM4J)

1. 日期转换器

如果你有一个 String(例如: “2001-02-11”)在 Javascript, 你想把它转换成 Java 日期。那么你有 2 种选择，一是使用 Date.parse()然后使用 DataConverter 传入服务器端，还有一种选择是把该 String 传入，然后用 java 的 SimpleDateFormat(或者其他的)来转换。

同样，如果你有个 Java 的 Date 类型并且希望在 HTML 使用它。你可以先用 SimpleDateFormat 把它转换成字符串再使用。也可以直接传 Date 给 Javascript，然后用 Javascript 格式化。第一种方式简单一些，尽管浪费了你的转换器，而且这样做也会是浏览器上的显示逻辑受到限制。其实后面的方法更好，也有一些工具可以帮你，例如：

- [The Javascript Toolbox Date formatter](#)
- [Web Developers Notes on Date formatting](#)

2. 数组转换器

数组实体不太容易理解。默认情况下 DWR 能转换所有原生类型的数组，还有所有 marshallable 对象的数组。这些 marshallable 对象包括前面介绍的 String 和 Date 类型。match 属性看上去很怪。

```
<convert converter="array" match="[Z"/>
<convert converter="array" match="[B"/>
<convert converter="array" match="[S"/>
<convert converter="array" match="[I"/>
<convert converter="array" match="[J"/>
<convert converter="array" match="[F"/>
<convert converter="array" match="[D"/>
<convert converter="array" match="[C"/>
<convert converter="array" match="[L*"/>
```

上面没有解释 * 的作用 - 它是通配符，表示匹配接下来的所有字符串。这也是 DWR 可以转换任意类型的数组的原因。

3. bean 和对象转换器

两个没有默认打开的转换器是 **Bean** 和 **Object** 转换器。**Bean** 转换器可以把 **POJO** 转换成 **Javascript** 的接合数组(类似与 **Java** 中的 **Map**)，或者反向转换。这个转换器默认情况下是没打开的，因为 **DWR** 要获得你的允许才能动你的代码。

Object 转换器很相似，不同的是它直接应用于对象的成员，而不是通过 **getter** 和 **setter** 方法。下面的例子都是可以用 **object** 来替换 **bean** 的来直接访问对象成员。

如果你有一个在 `<create ...>` 中声明的远程调用 **Bean**。它有个一参数也是一个 **bean**，并且这个 **bean** 有一个 **setter** 存在一些安全隐患，那么攻击者就可能利用这一点。

你可以为某一个单独的类打开转换器：

```
<convert converter="bean" match="your.full.package.BeanName"/>
```

如果要允许转换一个包或者子包下面的所有类，可以这样写：

```
<convert converter="bean" match="your.full.package.*"/>
```

显而易见，这样写是允许转换所有的 **JavaBean**：

```
<convert converter="bean" match="*" />
```

- **BeanConverter 和 JavaBeans 规范**

用于被 **BeanConverter** 转换的 **Bean** 必须符合 **JavaBeans** 的规范，因为转换器用的是 **Introspection**，而不是 **Reflection**。这就是说属性要符合一下条件：有 **getter** 和 **setter**，**setter** 有一个参数，并且这个参数的类型是 **getter** 的返回类型。**setter** 应该返回 **void**，**getter** 应该没有任何参数。**setter** 没有重载。以上这些属于常识。就在 **eclipse** 里自动为每个属性添加 **setter**，**getter** 那种类型，如果你用的不是 **JavaBean**，那么你应该用 **ObjectConverter**。

- **设置 Javascript 变量**

DWR 可以把 **Javascript** 对象(又名 **maps**，或联合数组)转换成 **JavaBean** 或者 **Java** 对象。例子：

```
public class Remoted {
    public void setPerson(Person p) {
        // ...
    }
}

public class Person {
    public void setName(String name) { ... }
    public void setAge(int age) { ... }
    // ...
}
```

如果这个 **Remoted** 已经被配置成 **Creator** 了, **Persion** 类也定义了 **BeanConverter**, 那么你可以通过下面的方式调用 **Java** 代码:

```
var p = { name:"Fred", age:21 };
Remoted.setPerson(p);
```

- **限制转换器**

就像你可以在 **creator** 的定义中剔出一些方法一样, **converter** 也有类似的定义。

限制属性转换仅仅对于 **Bean** 有意义, 很明显原生类型是不要需要这个功能的, 所以只有 **BeanConverter** 及其子类型(**HibernateBeanConverter**)有这个功能。

语法是这样的:

```
<convert converter="bean" match="com.example.Fred">
  <param name="exclude" value="property1, property2" />
</convert>
```

这就保证了 **DWR** 不会调用 **fred.getProperty1()** 和 **fred.getProperty2** 两个方法。另外如果你喜欢"白名单"而不是"黑名单"的话:

```
<convert converter="bean" match="com.example.Fred">
  <param name="include" value="property1, property2" />
</convert>
```

安全上比较好的设计是使用"白名单"而不是"黑名单"。

- **访问对象的私有成员**

通过'**object**'转换器的参数的一个名为 **force** 的参数, 可以让 **DWR** 通过反射来访问对象私有成员。

语法是这样的:

```
<convert converter="object" match="com.example.Fred">
  <param name="force" value="true" />
</convert>
```

直到 **DWR1.1.3**, 这里有一个 **bug**, **public** 的 **field** 反而不能被发现, 所以你需要在 **public** 成员上设置 **force=true**。

4. 集合类型转换器

有个两个默认转换器，针对 `Map` 和 `Collection`：

```
<convert converter="collection" match="java.util.Collection"/>

<convert converter="map" match="java.util.Map"/>
```

一般来说这些转换器可以递归转换它们的内容。

但是也有两点不足之处：

- 仅仅用反射机制是没有方法明确集合里面是什么类型的。所以这两个转换器不能把集合里面的东西转换成有意义的 `Javascript` 对象。
- 不能明确是那种类型的集合。 虽然我们不能让他们自动的起作用，我们可以在 `dwr.xml` 中用 `signatures` 语法声明它们类型，使之正确转换。

5. 枚举类型转换器

枚举类型转换器默认是没有打开的。它在 `Java5` 中的 `Enum` 和 `Javascript` 的 `String` 之间进行转换。这个转换器默认关闭是因为 `DWR` 要在转换你的代码之前得到你的同意。

枚举类型转换器是 `DWR 1.1` 版以后才支持的。

你可以这样设置来打开这个转换器：

```
<convert converter="enum" match="your.full.package.EnumName"/>
```

设置 `Javascript`，一个简单的例子。假设你有下面的 `Java` 代码：

```
public class Remoted {
    public void setStatus(Status p) {
        // ...
    }
}

enum Status {
    PASS, FAIL,
}
```

如果 `Remoted` 类已经配置好 `Creator`，并且 `Status` 枚举类型已经设置了 `EnumConverter`。那么你就可以在 `javascript` 中这样调用：

```
Remoted.setStatus("PASS");
```

6. DOM 对象

DWR 可以自动转换来之 DOM,DOM4J,JDOM 和 XOM 的 DOM 树。你可以简单得用上面这些类库返回一个 Document、Element 或者 Node，DWR 会把他们自动转换成浏览器的 DOM 对象。

在程序启动的时候会有一个常见的关于 JDOM 转换器的警告，你可以放心的忽略它，除非你要用 JDOM：

```
INFO: Missing classdef for converter 'jdom'. Failed to load  
uk.ltd.getahead.dwr.convert.JDOMConverter. Cause: org/jdom/Document
```

因为 DWR 没有办法知道你是否想用 JDOM，所以这个信息设在 INFO 级别的。

如果你曾经尝试过使用 JDOM，你会意识到在这种情况下这个转换器不可用的 - 这也是我们显示这个信息的原因。

exist-db.org，我相信 DWR 能同 exist-db 很好的工作，因为它是建立在 W3C DOM 之上的，而 DWR 也支持这个。

3.4 <signatures>标签

DWR 使用反射机制在转换过程中找到它应该使用的类型。有时候类型的信息无法获得，在这种情况下你要在此处用方法签名给予暗示。

`signatures` 段使 DWR 能确定集合中存放的数据类型。例如下面的定义中我们无法知道 `list` 中存放的是什么类型。

```
public class Check {  
    public void setLotteryResults(List nos)  
    {  
        ...  
    }  
}
```

`signatures` 段允许我们暗示 DWR 应该用什么类型去处理。格式对以了解 JDK5 的泛型的人来说很容易理解。

```
<signatures>  
  <![CDATA[  
    import java.util.List;  
    import com.example.Check;  
    Check.setLotteryResults(List<Integer> nos);  
  ]]>  
</signatures>
```

DWR 中又一个解析器专门来做这件事，所以即便你的环境是 JDK1.3 DWR 也能正常工作。

解析规则基本上会和你预想规则的一样(有两个例外)，所以 `java.lang` 下面的类型会被默认 `import`。

第一个是 DWR1.0 中解析器的 `bug`，某些环境下不能返回正确类型。所以你也不用管它了。

第二个是这个解析器是“阳光(sunny day)”解析器。就是说它非常宽松，不想编译器那样严格的保证你一定正确。所以有时它也会允许你丢失 `import`：

```
<signatures>  
  <![CDATA[  
    import java.util.List;  
    Check.setLotteryResults(List<Integer>);  
  ]]>  
</signatures>
```

将来的 DWR 版本会使用一个更正式的解析器，这个编译器会基于官方 Java 定义，所以你最好不要使用太多这个不严格的东西。

signatures 段只是用来确定泛型参数中的类型参数。DWR 会自己使用反射机制或者运行时类型确定类型，或者假设它是一个 **String** 类型。所以：

不需要 **signatures** - 没有泛型参数：

```
public void method(String p);  
public void method(String[] p);
```

需要 **signatures** - DWR 不能通过反射确定：

```
public void method(List<Date> p);  
public void method(Map<String, WibbleBean> p);
```

不需要 **signatures** - DWR 能正确的猜出：

```
public void method(List<String> p);  
public void method(Map<String, String> p);
```

不需要 **signatures** - DWR 可以通过运行时类型确定：

```
public List<Date> method(String p);
```

没有必要让 Javascript 中的所有对象的 **key** 都是 **String** 类型 - 你可以使用其他类型作为 **key**。但是他们在使用之前会被转换成 **String** 类型。DWR1.x 用 Javascript 的特性把 **key** 转换成 **String**。DWR2.0 可能会用 **toString()** 方法，在服务段进行这一转换。

第4章. 整合

4.1 DWR 与 Servlet

有 2 个 Java 类你一般需要用在 DWR 中，是 `webContext` 和 `WebContextFactory`。在 DWR 1.x 它们在 `uk.ltd.getahead.dwr` 包，DWR 2.0+ 在 `org.directwebremoting` 包。这 2 个类给你访问标准 `Http servlet` 对象的入口。这些对象是：

- `HttpServletRequest`
- `HttpServletResponse`
- `HttpSession`
- `ServletContext`
- `ServletConfig`

4.1.1 使用 `webContext` 的方法：

```
import uk.ltd.getahead.dwr.WebContext;
import uk.ltd.getahead.dwr.WebContextFactory;
//
WebContext ctx = WebContextFactory.get();
req = ctx.getHttpServletRequest();
```

处理 `Http request` 和 `response` 做为只读是非常重要的。因为，当 `Http headers` 也许会通过，那么有些浏览器会忽略它们（比如 `IE` 忽略缓存参数）。任何尝试改变 `Http body` 将会导致 `DWR` 错误。

`WebContext` 使用一个本地线程变量，所以你能使用以上的代码放在任何地方。

也可以看一下 `DWR` 的 [Java 文档](#)，或者详细看一下 [WebContext](#)。

`WebContext` 代替了 `DWR1.1` 中的 `ExecutionContext`。

4.1.2 方法选择

在没有写依赖于 `DWR` 的代码时，要能够访问 `Http servlet` 对象是可以做到的（比如 `HttpServletRequest`, `HttpServletResponse`, `HttpSession`, `ServletContext` or `ServletConfig`）。`DWR` 将自动填充它。

举个例子：

```
public class Remote {
    public void method(int param, ServletContext cx, String s) { ... }
}
```

然后你将可以从 Javascript 中通访问它尽管没有 ServletContext 参数:

```
Remote.method(42, "test", callback);
```

DWR 将为你填充这个参数。

对这个方法这里有个小小的警告，你要保证你的没有把'callback function'作为第一个参数，而应该把它作为最后一个参数，或者作为[元数据对象](#)

4.2 DWR 与 Spring

4.2.1 让 DWR 和 Spring 一起工作的检查列表

1. 确认你用的是最新版的 DWR。Spring 创造器已经有了变化, 所以你最好检查一下 DWR 的[最新版本](#)。
2. 确认你的 Spring 的 Bean 在 DWR 外面运行良好。
3. 配置 DWR 和 Spring 一起工作。(看下面)
4. 查看演示页面: [http://localhost:8080/\[YOUR-WEBAPP\]/dwr](http://localhost:8080/[YOUR-WEBAPP]/dwr) , 检查 spring 的 Bean 是否出现。

DWR 对于 Spring 没有运行期依赖, 所以如果你不使用 Spring 那么 Spring 的支持不会产生任何影响到。

4.2.2 Spring Creator

这个创造器会在 `spring beans.xml` 里查询 beans, 并且会使用 Spring 去创建它们。如果你已经使用 Spring, 这个创造器会非常有用。否则将完全没有任何用处。

要让 DWR 使用 Spring 创造器去创建和远程调用 beans, 要像如下所示:

```
<allow>
    ...
    <create creator="spring" javascript="Fred">
        <param name="beanName" value="Shiela" />
    </create>
</allow>
```

4.2.3 找到 Spring 配置文件

有 3 个方法可以找到 Spring 配置文件

1. ContextLoaderListener

最简单的用法使从 Spring-MVC 里使用

```
org.springframework.web.context.ContextLoaderListener
```

你不必使用整个 Spring-MVC 去确认这个普遍的解决方案使多么好。只要再你的 `web.xml` 里配置:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/classes/beans.xml</param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

2. 使用 location 参数

如果你喜欢指定的 `beans.xml` 在你的 `dwr.xml` 文件中，那么你能使用 `location` 参数。你可以指定你希望数量的，必须有唯一的以“`location`”开头的命名。比如：`location-1`, `location-2`。这些 `locations` 是作为参数传到 Spring 的 `ClassPathXmlApplicationContext`:

```
<allow>
    ...
    <create creator="spring" javascript="Fred">
        <param name="beanName" value="Shiela" />
        <param name="location" value="beans.xml" />
    </create>
</allow>
```

3. 设置 `beenFactory` 目录

Spring 创造器有一个静态的方法：`setOverrideBeanFactory(BeanFactory)`，这个方法提供一个可编程的方式去覆盖任何 `BeanFactories`。

4.2.4 使用 Spring 配置 DWR

Bram Smeets 写了一个[有意思的 blog](#)，教你配置 DWR 使用 `beans.xml` 代替 `WEB-INF/web.xml`。我也对于如何在 `beans.xml` 中指定 `dwr.xml` 很感兴趣，尽管这看上去有些 Spring 传染病的感觉。

4.3 DWR 与 JSF

DWR 包括两个 JSF 的扩展点，一个创造器和一个 `ServletFilter`。

4.3.1 JSF Creator

DWR1.1 中有一个体验版的 `JsfCreator`。你可以在 `dwr.xml` 中这样使用：

```
<allow>
    ...
    <create creator="jsf" javascript="ScriptName">
        <param name="managedBeanName" value="beanName" />
        <param name="class" value="your.class" />
    </create>
    ...
</allow>
```

这将允许你通过 DWR 调用 `ManagedBean`。

4.3.2 Servlet Filter

DWR/Faces 过滤器允许你不在 JSF 的生命周期里调用 `FacesContext` 中的 `Bean`。

要使用 `JsfCreator`，你应该把 DWR/Faces 过滤器加到 `web.xml` 中。

```
<filter>
    <filter-name>DwrFacesFilter</filter-name>
    <filter-class>
        uk.ltd.getahead.dwr.servlet.FacesExtensionFilter
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>DwrFacesFilter</filter-name>
    <url-pattern>/dwr/*</url-pattern>
</filter-mapping>
```

这两个需要放在 `web.xml` 中与其他 `filter` 和 `filter-mapping` 放在一起。

4.4 DWR 与 Struts

DWR 能够 and 任何框架结合。这个网站是个极端巧妙的证据，因为它把 DWR 使用在 [Drupal](#)(PHP)。

DWR 和 Struts 整合有 2 个层次。最基础的层次就是同时使用这两个框架，这是非常容易的，但是这样就不允许在 DWR 和 Struts 之间共享 Action 了。

DWR 能够调用任何方法，所以你能从调用一个 Struts action，除非你不想那么做。ActionForm 的内容是什么，当返回 ActionForward 时 DWR 怎么做？

一个比较好方法是重构你想调用的 Action，提取出 Action 的逻辑。DWR 和你的 Action 就可以同时调用相同的方法了。

4.4.1 Struts creator

DWR1.1 增加了一个 StrutsCreator。你可以在 dwr.xml 中这样使用：

```
<allow>
    ...
    <create creator="struts" javascript="ScriptName">
        <param name="formBean" value="formBeanName" />
    </create>
    ...
</allow>
```

这样你就能从 DWR 调用 FormBeans。

4.4.2 开始顺序

如果正在使用 Struts 创造器，那么你应该确保 Struts 初始化在 DWR 之前。你要保证你在 web.xml 里有一个 <load-on-startup> 的值，其中 Struts 的值比 DWR 设置地要低。

4.5 DWR 与 Weblogic 或 PageFlow

DWR 中有一个创造器可以和 Weblogic 或者 Beehive 中的 PageFlow 一起工作。

PageFlow creator

DWR1.1 中加入了一个 PageFlowCreator。你可以这样使用：

```
<allow>
...
<create creator="pageflow" javascript="ScriptName"/>
...
</allow>
```

4.6 DWR 与 Hibernate

4.6.1 让 DWR 和 Hibernate 一起工作的检查列表

1. 确保你使用的是最新的 DWR。Hibernate 转换器是新东西，所以你需要下载[最新版本](#)
2. 确保你的 Hiberante 在没有 DWR 的时候工作正常。
3. 如果是 Spring 和 Hibernate 一起使用，那么你最好先了解一下如何将[整合 Spring](#)。
4. 配置 DWR，使之与 Hibernate 一起工作。（看下面）。
5. 查看演示页面：<http://localhost:8080/YOUR-WEBAPP/dwr>，确定 Spring 的 Bean 可以出现。

4.6.2 HibernateBeanConverter

除了我们可以决定是否要使用 lazy loaded 属性，这个转换器非常类似于标准的 BeanConverter。

在 DWR1.1 Hibernate 转换器被称为“hibernate”并且作用在 hibernate2，在 DWR2.x 有 2 个转换器被称为“hibernate2”并且作用在 hibernate3。

使用 HibernateBeanConverter 也许有点风险，原因如下：

- 结构: HibernateBeanConverter 不符合 MVC 模式，所以不能把对象在数据层和表现层之间进行隔离。这个风险可以通过在上面加上独立的 bean 来减轻。
- 性能: DWR 试图通过相同的序列化方式来转换所有可以得到的属性(除了 DWR 仅仅读 JavaBean 属性的时候)。所以可能会出现通过 HTTP 序列化了你的整个数据的情况。通常这并不是你想要的。要减少这一风险可以使用 BeanConverter(HibernateBeanConverter 衍生于它)的排除某些属性的功能，如下所示：

```
<param name="exclude" value="propertyToExclude1, propertyToExclude2"/>
```

HibernateBeanConverter 会尝试不去读取没有初始化的属性。如果你只是想读取所有的东西那么应该使用 BeanConverter。

建议使用 Hibernate3，实际上 Hibernate2 的情况，你会发现你得到的都是空的 Bean。

4.6.3 Session 管理

如果你使用 Hibernate 对象，你需要知道每一个 DWR 请求都是一个新的 Servlet 请求，所以你需要保证为每个请求打开一个 Hiberante 的 Session。

如果你用 Spring，那么可以很方便的使用 Spring 里面的 OpenSessionInViewFilter，它可以保证为每个请求打开一个 Hiberante 的 Session。类似的解决方案在其它 Framework 中也存在。

4.7 DWR 与 WebWork

WebWork 支持在 DWR2.0m3 以后才有。

要可以通过 DWR 调用 WW 的 Action，要做两件事。

4.7.1 配置 dwr.xml

你必须在 dwr 的配置文件中加入这样的配置：

```
<create creator="none" javascript="DWRAction">
  <param name="class"
value="org.directwebremoting.webwork.DWRAction" />
  <include method="execute" />
</create>

<convert converter="bean"
  match="org.directwebremoting.webwork.ActionDefinition">
  <param name="include"
value="namespace,action,method,executeResult" />
</convert>

<convert converter="bean"
  match="org.directwebremoting.webwork.AjaxResult" />
```

这样你 AjaxWebWork Action 调用返回一个 action 实例(而不是文字)。然后你必须包括 action 对象的转换器定义(package 级别或单独 action)。

```
<convert converter="bean" match="your_action_package.*"/>
```

4.7.2 在 JSP 中导入脚本

下面这些代码开启 DWR 调用 Action 的功能。你还要导入 DWRActionUtil.js 脚本(在你的 web 脚本路径中)

像这样在 JS 中调用 Action：

```
DWRActionUtil.execute(id, params, callback, [displayMessage]);
```

1. id 参数

- actionUri: 要调用 action 的 URI(没有 .action)。例如：

```
DWRActionUtil.execute('/ajax/TestFM', 'myform', 'doOnTextResult');
```


- **actionDefinitionObject**: 在 `xwork.xml` 中定义的 **action** 对象。必须指定下面的内容:
 - ✧ **namespace**: `xwork.xml` 中 **action** 的名称空间
 - ✧ **action**: `xwork.xml` 中 **action** 的名字
 - ✧ **executeResult**: `true|false` (是否执行 **action** 的结果, 如果 `false` 直接返回 **action** 实例)

例如:

```
DWRActionUtil.execute({
    namespace: '/ajax',
    action: 'TestJS',
    executeResult: 'true'
}, 'data', doOnJSResult, "stream...");
```

2. **params** 参数

- **emptyParams**: 传递{}忽略任何参数。例子:

```
DWRActionUtil.execute('/ajax/TestFM', {}, doOnJSResult, "stream...");
```

- **fieldId**: 被转换为 **action** 调用参数的字段的 id。

例子:

```
<input id="mytext" name="mytext" value="some value" type="text"/>
DWRActionUtil.execute('/ajax/TestFM', 'mytext', doOnJSResult,
"stream...");
```

- **formId**: 表单的 id。所有的 `input` 值被转换为 **action** 调用参数。

Note: 如果你的 **action** 使用了 **parameter** 拦截器, 那么你的 **action** 会得到正确的参数值, 请参考 **WebWork** 的文档。

3. **callback** 参数

- **callbackFunction**: 在 **DWR** 中, 这个函数在请求完毕后调用。
- **callbackObject**: 在 **DWR** 中, **callback** 对象。

4. **displayMessage** 参数

displayMessage 是可选参数, 当请求完毕后显示的消息(参考 **DWR** 文档)

4.7.3 高级

你可以声明一个 **pre/post Action** 处理器, 在 `web.xml` 中的一个 **context-wide** 初始化参数 (**dwrActionProcessor**)。处理器必须实现 `org.directwebremoting.webwork.IDWRActionProcessor` 接口。这个处理器将会在 **action** 之前和之后被调用, 所以你可以做一些预处理或改变结果。

4.8 DWR 与 Acegi

整合 DWR 和 Acegi 是为了保护 bean 方法调用

在此之前有必须熟悉 spring 和 acegi, 我就不解释 acegi 的配置了, 您可以从 [acegi](#) 网站下载最新版本。

4.8.1 问题提出

我们必须保护一个 bean, 通过 DWR 设置 “exposed”。

Acegi 是一个基于 Spring 的安全机制的框架, 这个例子告诉你如何去阻止通过网页访问一个 bean 方法, 如果它没有权限的。

在 dwr.xml 里, 在没有 acegi 保护下, 我们这样声明 bean

```
<create creator="spring" javascript="loanDWR" beanName="loanDWR">
    <include method="addLoan" />
</create>
```

在 Spring 配置文件里, 这个 addLoan 方法被网页调用, 这个 loanDWR bean 是一个 Spring 管理的 bean

```
<bean id="loanDWR" class="fr.iremia.jlab.web.dwr.LoanDWR">
    <property name="personDAO">
        <ref bean="personDAO" />
    </property>
    <property name="loanDAO">
        <ref bean="loanDAO" />
    </property>
</bean>
```

4.8.2 解决方案

然后我们怎么去阻止没有权限的访问 loanDWR.addLoan 这个方法呢? 如下所示:

- 创建一个安全机制的拦截器

```
<bean id="loanDWRSecurityInterceptor"
    class="net.sf.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor"
    <property name="authenticationManager" ref="authenticationManager" />
    <property name="accessDecisionManager"
        ref="httpRequestAccessDecisionManager" />
    <property name="objectDefinitionSource">
        <value>fr.iremia.jlab.web.dwr.LoanDWR.addLoan=edit,admin</value>
    </property>
</bean>
```

LoanDWR.addLoan 方法, 只能被管理员或者有修改角色权限的用户 edit 调用

想要了解 acegi 的属性: authenticationManager 和 httpRequestAccessDecisionManager, 请参考 <http://acegisecurity.org/docbook/acegi.html>

- **为 bean 创建一个代理**

给原有的 loanDWR Spring bean 添加一个代理, 使用 Spring proxyfactorybean:

```
<bean id="loanDWRSecure"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="loanDWR" />
  <property name="interceptorNames">
    <idref local="loanDWRSecurityInterceptor" />
  </property>
</bean>
```

现在, loadDWRSecure 是个代理被 loanDWR 调用, 每个调用 addLoan 方法将被 Acegi 拦截, 只有拥有权限的用户才能调用

- **修改 dwr 配置文件**

我们现在需要修改

```
<create creator="spring" javascript="loanDWR"
        beanName="loanDWRSecure">
  <include method="addLoan" />
</create>
```

第5章. DWR 中的 JavaScript 简介

DWR 根据 `dwr.xml` 生成和 Java 代码类似的 Javascript 代码。

相对而言 Java 同步调用，创建与 Java 代码匹配的 Ajax 远程调用接口的最大挑战来至与实现 Ajax 的异步调用特性。

DWR 通过引入回调函数来解决这个问题，当结果被返回时，DWR 会调用这个函数。

有两种推荐的方式来使用 DWR 实现远程方法调用。可以通过把回调函数放在参数列表里，也可以把回调函数放到元数据对象里。

当然也可以把回调函数做为第一个参数，但是不建议使用这种方法。因为这种方法在处理自动处理 http 对象时(查看"Alternative Method")上会有问题。这个方法主要是为向下兼容而存在的。

5.1 简单的回调函数

假设你有一个这样的 Java 方法：

```
public class Remote {  
    public String getData(int index) { ... }  
}
```

我们可以在 Javascript 中这样使用：

```
<script type='text/javascript' src='[WEBAPP]/dwr/engine.js'></script>  
<script type='text/javascript'  
src='[WEBAPP]/dwr/interface/Remote.js'></script>  
.....  
    <script type="text/javascript">  
        function handleGetData(str) {  
            alert(str);  
        }  
  
        Remote.getData(42, handleGetData);  
    </script>
```

42 是 Java 方法 `getData()` 的一个参数。

此外你也可以使用这种减缩格式：

```
Remote.getData(42, function(str) { alert(str); });
```

5.2 调用元数据对象

另外一种语法时使用"调用元数据对象"来指定回调函数和其他的选项。上面的例子可以写成这样：

```
Remote.getData(42, {
  callback:function(str) { alert(str); }
});
```

这种方法有很多优点：易于阅读，更重要的指定额外的调用选项。

在回调函数的元数据中你可以指定超时和错误的处理方式。例如：

```
Remote.getData(42, {
  callback:function(str) { alert(str); },
  timeout:5000,
  errorHandler:function(message) { alert("Oops: " + message); }
});
```

5.3 查找回调函数

有些情况下我们很难区分各种回调选项(记住，Javascript 是不支持函数重载的)。例如：

```
Remote.method({ timeout:3 }, { errorHandler:somefunc });
```

这两个参数之一是 **bean** 的参数，另一个是元数据对象，但是我们不能清楚的告诉 DWR 哪个是哪个。为了可以跨浏览器，我们假定 `null == undefined`。所以当前的情况，规则是：

- 如果第一个或最后一个是一个函数，那么它就是回调函数，没有元数据对象，并且其他参数都是 **Java** 的方法参数。
- 另外，如果最后一个参数是一个对象，这个对象中有一个 **callback** 成员，并且它是个函数，那么这个对象就是元数据对象，其他的都是 **Java** 方法参数。
- 另外，如果第一个参数是 `null`，我们就假设没有回调函数，并且其他的都是 **Java** 方法参数。尽管如此，我们会检查最后一个参数是不是 `null`，如果是就发出警告。
- 最后如果最后一个参数是 `null`，那么就没有 **callback** 函数。
- 另外，发出错误信号是个糟糕的请求格式。

5.4 创建一个与 Java 对象匹配的 Javascript 对象

假设你有这样的 Java 方法:

```
public class Remote {  
    public void setPerson(Person p) {  
        this.person = p;  
    }  
}
```

Person 对象的结构是这样的:

```
public Person {  
    private String name;  
    private int age;  
    private Date[] appointments;  
    // getters and setters ...  
}
```

那么你可以在 Javascript 中这样写:

```
var p = {  
    name: "Fred Bloggs",  
    age: 42,  
    appointments: [ new Date(), new Date("1 Jan 2008") ]  
};  
Remote.setPerson(p);
```

在 Javascript 没有出现的字段, 在 Java 中就不会被设置。

因为 setter 都是返回'void', 我们就不需要使用 callback 函数了。如果你想要一个返回 void 的服务端方法的完整版, 你也可以加上 callback 函数。很明显 DWR 不会向它传递任何参数。

第6章. engine.js 功能

engine.js 对 DWR 非常重要，因为它是用来转换来至动态生成的接口的 javascript 函数调用的，所以只要用到 DWR 的地方就需要它。

每一个页面都需要下面这些语句来引入主 DWR 引擎。

```
<script type="text/javascript" src="/[YOUR-WEB-APP]/dwr/engine.js">
</script>
```

6.1 使用选项

下面这些选项可以通过 `DWREngine.setX()` 函数来设置全局属性。例如：

```
DWREngine.setTimeout(1000);
```

或者在单次调用级别上(假设 `Remote` 被 DWR 暴露出来了)：

```
Remote.singleMethod(params, {
  callback:function(data) { ... },
  timeout:2000
});
```

远程调用可以批量执行来减少反应时间。`endBatch` 函数中可以设置选项。

```
DWREngine.beginBatch();
Remote.methodInBatch1(params, callback1);
Remote.methodInBatch2(params, callback2);
DWREngine.endBatch({
  timeout:3000
});
```

可以混合这几种方式，那样的话单次调用或者批量调用级别上的设置可以复写全局设置(就像你希望的那样)。当你在一个批量处理中多次设置了某个选项，DWR 会保留最后一个。所以如果 `Remote.singleMethod()` 例子在 `batch` 里面，DWR 会使用 `3000ms` 做为超时的时间。

`callback` 和 `exceptionHandler` 两个选项只能在单次调用中使用，不能用于批量调用。

`preHook` 和 `postHook` 选项两个选项是可添加的，就是说你可以为每一次调用添加多个 `hook`。全局的 `preHook` 会在批量调用和单次调用之前被调用。同样全局的 `postHook` 会在单次调用和批量调用之后被调用。

如果以上叙述让你感到混乱，不用担心。DWR 的设计往往和你想象中的一样，所以其实这些并不复杂

6.2 选项索引

下面是可用选项列表。

Option	Global	Batch	Call	Summary
async	1.1	1.1	1.1	设置是否为异步调用，不推荐同步调用
headers	2.0	2.0	2.0	在 XHR 调用中加入额外的头信息
parameters	2.0	2.0	2.0	可以通过 <code>Meta-datarequest.getParameter()</code> 取得的元数据
httpMethod	2.0	2.0	2.0	选择 GET 或者 POST. 1.x 中叫 “verb”
rpcType	2.0	2.0	2.0	选择是使用 xhr, iframe 或者 script-tag 来实现远程调用. 1.x 中叫 “method”
skipBatch	1.0*	2.1?	-	某个调用是否应该设置为 batch 中的一部分或者直接的。这个选项和上面都有些不同。 *没有 <code>setSkipBatch()</code> 方法，批量调用是通过 <code>beginBatch()</code> 和 <code>endBatch()</code> 来控制的。
timeout	1.0	1.1	1.1	设定超时时长，单位 ms

6.2.1 处理器(Handler)

Option	Global	Batch	Call	Summary
errorHandler	1.0	1.1	1.1	当出了什么问题时的动作。1.x 中还包括服务端的异常。从 2.0 开始服务端异常通过 “exceptionHandler” 处理
warningHandler	1.0	2.0	2.0	当因为浏览器的 bug 引起问题时的动作，所以默认这个设置为 null(关闭)
textHtmlHandler	2.0	2.0	2.0	当得到不正常的 text/html 页面时的动作(通常表示超时)

6.2.2 调用处理器(Call Handler)

Option	Global	Batch	Call	Summary
callback	-	-	1.0	调用成功以后的要执行的回调函数，应该只有一个参数：远程调用得到的数据
exceptionHandler	-	-	2.0	远程调用失败的动作，一般是服务端异常或者数据转换问题。

6.2.3 Hooks (一个 batch 中可以注册多个 hook)

Option	Global	Batch	Call	Summary
preHook	1.0	1.1	1.1	远程调用前执行的函数
postHook	1.0	1.1	1.1	远程调用后执行的函数

6.2.4 全局选项(在单次调用或者批量调用中不可用)

Option	Global	Batch	Call	Summary
ordered	1.0	-	-	DWR 是否支持顺序调用
pollType	2.0	-	-	选择 xhr 或者 iframe 的反转 Ajax
reverseAjax	2.0	-	-	是否查找 inbound 调用

6.2.5 废弃的选项

Option	Global	Batch	Call	Summary
verb	1.0	1.1	1.1	2.0 废弃。使用 “httpMethod” 代替
method	1.0	1.1	1.1	2.0 废弃。使用 “rpcType” 代替

6.2.6 未来版本的选项

Option	Global	Batch	Call	Summary
onBackButton	2.1?	2.1?	-	用户按了 back 按钮后的动作
onForwardButton	2.1?	2.1?	-	用户按了 forward 按钮的动作

6.3 选项说明

6.3.1 批量调用

你可以使用 `batch` 来批量的执行远程调用。这样可以减少与服务器的交互次数，所以可以提交反应速度。

一个 `batch` 以 `DWREngine.beginBatch()` 开始，并以 `DWREngine.endBatch()` 结束。当 `DWREngine.endBatch()` 被调用，我们就结束了远程调用的分组，这样 DWR 就在一次与服务器的交互中执行它们。

DWR 会小心的处理保证所有的回调函数都会被调用，所以你可以明显的打开和关闭批处理。只要别忘了调用 `endBatch()`，否则所有的远程调用永远的处于列队中。

警告：很明显，把一些远程调用放在一起执行也会产生一些影响。例如不能在 `batch` 里面执行同步调用。所有的元数据选项，例如 `hooks`, `timeouts` 和 `errorHandlers` 都在 `batch` 级别的，而不是单次调用级别上的。所以如果一个 `batch` 中有两个调用设置了不同的超时，除了最后一个其他的都被忽略。

6.3.2 顺序调用

因为 Ajax 一般是异步调用，所以远程调用不会按照发送的顺序返回。`DWREngine.setOrdered(boolean)` 允许结果严格按照发送的顺序返回。DWR 在旧的请求安全返回以后才去发送新的请求。

我们一定需要保证请求按照发送的顺序返回吗？（默认为 `false`）

警告：把这个设置为 `true` 会减慢你的应用程序，如果一个消息丢失，浏览器就会没有响应。很多时候即使用异步调用也有更好的解决办法，所以在用这一功能之前先好好考虑一下。

6.3.3 错误警告和超时

● 处理错误和警告

当因为一些原因调用失败，DWR 就会调用错误和警告 `handler`(根据错误的激烈程度)，并传递错误消息。

你可以用这种方法来在 `alert` 窗口或状态来中显示错误信息。

你可以使用 `DWREngine.setErrorHandler(function)`来改变错误处理方式，同样通过 `DWREngine.setWarningHandler(function)`来改变警告处理方式。

● 设置超时

`DWREngine.setTimeout()`，单次调用和批量调用级别的元数据选项，允许你设置一个超时值。全局的 `DWREngine.setTimeout()`函数设置全局超时。如果设置值为 `0`(默认)可以将超时关掉。

`setTimeout()`的单位是毫秒。如果调用超时发生，错误处理器就会被调用。

- 一个例子:

```
Remote.method(params, {
    callback:function(data) { alert("it worked"); },
    errorHandler:function(message) { alert("it broke"); },
    timeout:1000
});
```

如果 Remote.method()调用超过了 1 分钟还没有返回, "it broke"消息就会被显示。

6.3.4 远程调 Hooks

DWREngine.setPreHook(function) 和 DWREngine.setPostHook(function) 。

如果你想在 DWR 调用之前出现一个提示, 你可以设置 pre-hook 函数。它将会被调用, 但是没有参数传递到这个函数。当你希望让一些按钮在调用期间变灰来防止被再次使用, 这一功能将会很有用。

post-hook 用来和 pre-hook 一起使用来逆转 pre-hook 产生的一些改变。

一个使用 pre 和 post hook 的例子就是 DWRUtil.useLoadingMessage() 函数。

6.3.5 远程调用选项

DWR 有一些选项用来控制远程调用的处理方式。method 和 verb 对于用户应该时透明的, 但是不同的浏览器效果的可能不一样。一般来说 DWR 会选择正确方法来处理, 但是这些选项对于在不同效果的浏览器之间开发很有用。

- DWREngine.setAsync(flag)

DWR1.0 不支持。

我们指定让 XHR 异步调用了吗? 默认为 true。警告如果你使用的时 IFrame 或者 ScriptTag 这一选项被忽略。一般来说把他变成 false 是个糟糕的做法。因为那样会使你的浏览器变慢。

要设置全局同步机制:

```
DWREngine.setAsync(true);
```

或者设置单次调用同步:

```
Remote.method(params, {
    callback:function(data) { ... },
    async:true
});
```

或者在 `batch` 里面:

```
DWREngine.beginBatch();
    Remote.method1(params, callback1);
    Remote.method2(params, callback2);
    DWREngine.endBatch({
        async:true
    });
```

- **DWREngine.setMethod(newmethod)**

用来设置恰当的方法。`setMethod()`不能把正使用你选择的方法，它只是保证首先尝试使用那个方法。`newmethod` 必须是 `DWREngine.XMLHttpRequest` 或者 `DWREngine.IFrame`，或者 2.0 以后的 `DWREngine.ScriptTag`。

`XMLHttpRequest` 时默认的，并且大多数情况下可用。当 `ActiveX` 禁用 `IFrame` 就有用了，尽管 DWR 能自动检测出这种情况并切换到 `IFrame`。当你要突破跨域调用的限制，`ScriptTag` 就很有用了。

例如，要设置全局的远程调用方法:

```
DWREngine.setMethod(DWREngine.IFrame);
```

或者设置单次调用同步:

```
Remote.method(params, {
    callback:function(data) { ... },
    method:DWREngine.IFrame
});
```

或者在 `batch` 里面:

```
DWREngine.beginBatch();
    Remote.method1(params, callback1);
    Remote.method2(params, callback2);
    DWREngine.endBatch({
        method:DWREngine.IFrame
    });
```

- **DWREngine.setVerb(verb)**

这个选项允许你选择 `POST` 和 `GET`，无论时用 `iframe` 还是 `XMLHttpRequest` 方法。一些浏览器(例如，旧版的 `Safari`)不支持 `XHR-POST` 所以 DWR 就自动切换到 `GET`，即使你设置 `POST` 为 `verb`。所以 `setVerb()`应当被仅仅做为一个提示。

如果使用 `ScriptTag` 来远程调用，设置 `verb` 时没有的。

例如，设置全局远程调用的 **verb**：

```
DWREngine.setVerb("GET");
```

或者设置单次调用同步：

```
Remote.method(params, {  
    callback: function(data) { ... },  
    verb: "GET"  
});
```

或者在 **batch** 里面：

```
DWREngine.beginBatch();  
    Remote.method1(params, callback1);  
    Remote.method2(params, callback2);  
    DWREngine.endBatch({  
        verb: "GET"  
    });
```

第7章. util.js 功能

util.js 包含了一些工具函数来帮助你用 javascript 数据(例如从服务器返回的数据)来更新你的 web 页面。

你可以在 DWR 以外使用它，因为它不依赖于 DWR 的其他部分。你可以下载整个 DWR 或者单独下载。

4 个基本的操作页面的函数：`getValue[s]()`和 `setValue[s]()`可以操作大部分 HTML 元素除了 `table`，`list` 和 `image`。`getText()`可以操作 `select list`。

要修改 `table` 可以用 `addRows()`和 `removeAllRows()`。要修改列表(`select` 列表和 `ul,ol` 列表)可以用 `addOptions()`和 `removeAllOptions()`。

还有一些其他功能不是 `DWRUtil` 的一部分。但它们也很有用，它们可以用来解决一些小问题，但是它们不是对于所有任都通用的。

7.1 `$()`

`$()` 函数(它是合法的 Javascript 名字) 是从 `Prototype` 偷来的主意。

大略上的讲：`$ = document.getElementById`。因为在 `Ajax` 程序中，你会需要写很多这样的语句，所以使用 `$()` 会更简洁。

通过指定的 `id` 来查找当前 `HTML` 文档中的元素，如果传递给它多个参数，它会返回找到的元素的数组。所有非 `String` 类型的参数会被原封不动的返回。这个函数的灵感来至于 `prototype` 库，但是它可以在更多的浏览器上运行。

可以看看 `DWRUtil.toDescriptiveString` 的演示。

从技术角度来讲他在 `IE5.0` 中是不能使用的，因为它使用了 `Array.push`，尽管如此通常它只是用来同 `engine.js` 一起工作。如果你不想要 `engine.js` 并且在 `IE5.0` 中使用，那么你最好为 `Array.push` 找个替代品。

7.2 `addOptions` and `removeAllOptions`

DWR 的一个常遇到的任务就是根据选项填充选择列表。下面的例子就是根据输入填充列表。

下面将介绍 `DWRUtil.addOptions()` 的几种是用方法。

如果你希望在你更新了 `select` 以后，它仍然保持运来的选择，你要像下面这样做：

```
var sel = DWRUtil.getValue(id);
DWRUtil.removeAllOptions(id);
DWRUtil.addOptions(id, ...);
DWRUtil.setValue(id, sel);
```

如果你想加入一个初始的"Please select..." 选项那么你可以直接加入下面的语句:

```
DWRUtil.addOptions(id, \["Please select ..."]);
```

DWRUtil.addOptions 有 5 种模式

- ✧ **数组:** DWRUtil.addOptions(selectid, array) 会创建一堆 option, 每个 option 的文字和值都是数组元素中的值。
- ✧ **对象数组 (指定 text):** DWRUtil.addOptions(selectid, data, prop) 用每个数组元素创建一个 option, option 的值和文字都是在 prop 中指定的对象的属性。
- ✧ **对象数组 (指定 text 和 value 值):** DWRUtil.addOptions(selectid, array, valueprop, textprop) 用每个数组元素创建一个 option, option 的值是对象的 valueprop 属性, option 的文字是对象的 textprop 属性。
- ✧ **对象:** DWRUtil.addOptions(selectid, map, reverse)用每个属性创建一个 option。对象属性名用来作为 option 的值, 对象属性值用来作为属性的文字, 这听上去有些不对。但是事实上却是正确的方式。如果 reverse 参数被设置为 true, 那么对象属性值用来作为选项的值。
- ✧ **对象的 Map:** DWRUtil.addOptions(selectid, map, valueprop, textprop) 用 map 中的每一个对象创建一个 option。用对象的 valueprop 属性做为 option 的 value, 用对象的 textprop 属性做为 option 的文字。
- ✧ **ol 或 ul 列表:** DWRUtil.addOptions(ulid, array) 用数组中的元素创建一堆 li 元素, 他们的 innerHTML 是数组元素中的值。这种模式可以用来创建 ul 和 ol 列表。

这是网上的[例子](#)

7.3 addRows and removeAllRows

DWR 通过这两个函数来帮你操作 table: DWRUtil.addRows() 和 DWRUtil.removeAllRows()。这个函数的第一个参数都是 table、tbody、thead、tfoot 的 id。一般来说最好使用 tbody, 因为这样可以保持你的 header 和 footer 行不变, 并且可以防止 Internet Explorer 的 bug。

● DWRUtil.removeAllRows()

```
DWRUtil.removeAllRows(id);
```

描述:

通过 id 删除 table 中所有行。

参数:

id: table 元素的 id(最好是 tbody 元素的 id)

- **DWRUtil.addRows()**

```
DWRUtil.addRows(id, array, cellfuncs, [options]);
```

描述:

向指定 **id** 的 **table** 元素添加行。它使用数组中的每一个元素在 **table** 中创建一行。然后用 **cellfuncs** 数组中的没有函数创建一个列。单元格是依次用 **cellfunc** 根据没有数组中的元素创建出来的。

DWR1.1 开始, **addRows()** 也可以用对象做为数据。如果你用一个对象代替一个数组来创建单元格, 这个对象会被传递给 **cell** 函数。

参数:

id: table 元素的 id(最好是 tbody 元素的 id)

array: 数组(DWR1.1 以后可以是对象), 做为更新表格数据。

cellfuncs: 函数数组, 从传递过来的行数据中提取单元格数据。

options: 一个包含选项的对象(见下面)

选项包括:

rowCreator: 一个用来创建行的函数(例如, 你希望个 tr 加个 css). 默认是返回一个 `document.createElement("tr")`

cellCreator: 一个用来创建单元格的函数(例如, 用 th 代替 td). 默认返回一个 `document.createElement("td")`

这是网上的[例子](#)

7.4 getText

getText(id)和 **getValue(id)**很相似。出了它是为 **select** 列表设计的。你可能需要取得显示的文字, 而不是当前选项的值。

这是网上的[例子](#)

7.5 getValue

DWRUtil.getValue(id)是 **setValue()**对应的"读版本"。它可以从 **HTML** 元素中取出其中的值, 而你不用管这个元素是 **select** 列表还是一个 **div**。

这个函数能操作大多数 **HTML** 元素包括 **select**(去处当前选项的值而不是文字)、**input** 元素(包括 **textarea**)、**div** 和 **span**。

这是网上的[例子](#)

7.6 getValues

`getValues()`和 `getValue()`非常相似，除了输入的是包含 `name/value` 对的 javascript 对象。`name` 是 HTML 元素的 ID，`value` 会被更改为这些 ID 对象元素的内容。这个函数不会返回对象，它只更改传递给它的值。

从 DWR1.1 开始 `getValues()`可以传入一个 HTML 元素(一个 DOM 对象或者 id 字符串)，然后从它生成一个 `reply` 对象。

这是网上的[例子](#)

7.7 onReturn

当按下 `return` 键时，得到通知。

当表单中有 `input` 元素，触发 `return` 键会导致表单被提交。当使用 `Ajax` 时，这往往不是你想要的。而通常你需要的触发一些 `Javascript`。

不幸的是不同的浏览器处理这个事件的方式不一样。所以 `DWRUtil.onReturn` 修复了这个差异。如果你需要一个同表单元素中按回车相同的特性，你可以用这样代码实现：

```
<input type="text"
onkeypress="DWRUtil.onReturn(event,submitFunction)"/>
<input type="button" onclick="submitFunction()"/>
```

你也可以使用 `onkeypress` 事件或者 `onkeydown` 事件，他们做同样的事情。

一般来说 `DWR` 不是一个 `Javascript` 类库，所以它应该试图满足这个需求。不管怎样，这是在使用 `Ajax` 过程中一个很有用函数。

这个函数的工作原理是 `onSubmit()`事件只存在于`<FORM ...>`元素上

这是网上的[例子](#)

7.8 selectRange

选择一个输入框中的一些范围的文字。

你可能为了实现类似"Google suggest"类型的功能而需要选择输入框中的一些范围的文字，但是不同浏览器间选择的模型不一样。这 `DWRUtil` 函数可以帮你实现。

```
DWRUtil.selectRange(ele, start, end)
```

这是网上的[例子](#)

7.9 setValue

DWRUtil.setValue(id, value)根据第一个参数中指定的 id 找到相应元素，并根据第二个参数改变其中的值。

这个函数能操作大多数 HTML 元素包括 select(去处当前选项的值而不是文字)、input 元素(包括 textarea)、div 和 span。

这是网上的[例子](#)

7.10 setValues

setValues()和 setValue()非常相似，除了输入的是包含 name/value 对的 javascript 对象。name 是 HTML 元素的 ID，value 是你想要设置给相应的元素的值。

这是网上的[例子](#)

7.11 toDescriptiveString

DWRUtil.toDescriptiveString()函数比默认的 toString()更好。第一个参数是要调试的对象，第二个参数是可选的，用来指定内容深入的层次：

0: 单行调试

1: 多行调试，但不深入到子对象。

2: 多行调试，深入到第二层子对象

以此类推。一般调试到第二级是最佳的。

还有第三个参数，定义初始缩进。这个函数不应该被用于调式程序之外，因为以后可能会有变化。

这是网上的[例子](#)

7.12 useLoadingMessage

这个方法将来可能被废弃，因为这个实现实在太专断了。为什么是红色，为什么在右上角，等等。唯一的答案就是：抄袭 Gmail。这里的建议是以本页面中的代码为模板，根据你的需求自定义。

你必须在页面加载以后调用这个方法(例如，不要在 onload()事件触发之前调用)，因为它要创建一个隐藏的 div 来容纳消息。

最简单的做法时在 `onload` 事件中调用 `DWRUtil.useLoadingMessage`，像这样：

```
<head>
  <script>
    function init() {
      DWRUtil.useLoadingMessage();
    }
  </script>
  ...
</head>
<body onload="init();" >
  ...
```

可能有些情况下你是不能容易的编辑 `header` 和 `body` 标签(如果你在使用 `CMS`，这很正常)，在这样的情况下你可以这样做：

```
<script>
function init() {
  DWRUtil.useLoadingMessage();
}

if (window.addEventListener) {
  window.addEventListener("load", init, false);
}
else if (window.attachEvent) {
  window.attachEvent("onload", init);
}
else {
  window.onload = init;
}
</script>
```

下面这些是这个函数的代码，它对于你要实现自己的加载消息很有用。这个函数的主要内容是动态创建一个 `div`(`id` 是 `disabledZone`)来容纳消息。重要的代码是当远程调用时使它显示和隐藏：

```
DWREngine.setPreHook(function() {
  $('disabledZone').style.visibility = 'visible';
});
DWREngine.setPostHook(function() {
  $('disabledZone').style.visibility = 'hidden';
});
This is fairly simple and makes it quite easy to implement your own
"loading" message.
```

```
function useLoadingMessage(message) {
    var loadingMessage;
    if (message) loadingMessage = message;
    else loadingMessage = "Loading";

    DWREngine.setPreHook(function() {
        var disabledZone = $('disabledZone');
        if (!disabledZone) {
            disabledZone = document.createElement('div');
            disabledZone.setAttribute('id', 'disabledZone');
            disabledZone.style.position = "absolute";
            disabledZone.style.zIndex = "1000";
            disabledZone.style.left = "0px";
            disabledZone.style.top = "0px";
            disabledZone.style.width = "100%";
            disabledZone.style.height = "100%";
            document.body.appendChild(disabledZone);
            var messageZone = document.createElement('div');
            messageZone.setAttribute('id', 'messageZone');
            messageZone.style.position = "absolute";
            messageZone.style.top = "0px";
            messageZone.style.right = "0px";
            messageZone.style.background = "red";
            messageZone.style.color = "white";
            messageZone.style.fontFamily = "Arial,Helvetica,sans-serif";
            messageZone.style.padding = "4px";
            disabledZone.appendChild(messageZone);
            var text = document.createTextNode(loadingMessage);
            messageZone.appendChild(text);
        }
        else {
            $('messageZone').innerHTML = loadingMessage;
            disabledZone.style.visibility = 'visible';
        }
    });

    DWREngine.setPostHook(function() {
        $('disabledZone').style.visibility = 'hidden';
    });
}
```

下面的做法能简单的使用有加载消息图片：

```
function useLoadingImage(imageSrc) {
    var loadingImage;
    if (imageSrc) loadingImage = imageSrc;
    else loadingImage = "ajax-loader.gif";
    DWREngine.setPreHook(function() {
        var disabledImageZone = $('disabledImageZone');
        if (!disabledImageZone) {
            disabledImageZone = document.createElement('div');
            disabledImageZone.setAttribute('id', 'disabledImageZone');
            disabledImageZone.style.position = "absolute";
            disabledImageZone.style.zIndex = "1000";
            disabledImageZone.style.left = "0px";
            disabledImageZone.style.top = "0px";
            disabledImageZone.style.width = "100%";
            disabledImageZone.style.height = "100%";
            var imageZone = document.createElement('img');
            imageZone.setAttribute('id', 'imageZone');
            imageZone.setAttribute('src', imageSrc);
            imageZone.style.position = "absolute";
            imageZone.style.top = "0px";
            imageZone.style.right = "0px";
            disabledImageZone.appendChild(imageZone);
            document.body.appendChild(disabledImageZone);
        }
        else {
            $('imageZone').src = imageSrc;
            disabledImageZone.style.visibility = 'visible';
        }
    });
    DWREngine.setPostHook(function() {
        $('disabledImageZone').style.visibility = 'hidden';
    });
}
```

然后你就可以这样使用:useLoadingImage("images/loader.gif");

7.13 Submission box

h1 非 util.js 中的功能

这里有一些功能不适合加入到 DWRUtil 中。它们在解决一下特殊问题是很用，但是他们还不够通用以适用任何场合。

修补浏览器事件

如果你创建了一个 **DOM** 元素，然后用 **addAttribute** 在这个元素上创建了一个事件，那么他们不能被正常的触发。你可以使用下面的脚本来遍历一个 **DOM** 树，并重新为他们绑定事件，这样他们就能正常的触发了。

把'click'改成你希望的事件。

```
DWREngine._fixExplorerEvents = function(obj) {
  for (var i = 0; i < obj.childNodes.length; i++) {
    var childObj = obj.childNodes[i];
    if (childObj.nodeValue == null) {
      var onclickHandler = childObj.getAttribute('onclick');
      if (onclickHandler != null) {
        childObj.removeAttribute('onclick');
        // If using prototype:
        //   Event.observe(childObj, 'click', new
Function(onclickHandler));
        // Otherwise (but watch out for memory leaks):
        if (element.attachEvent) {
          element.attachEvent("onclick", onclickHandler);
        }
        else {
          element.addEventListener("click", onclickHandler,
useCapture);
        }
      }
      DWREngine._fixExplorerEvents(childObj);
    }
  }
}
```

第8章. DWR 进阶

8.1 DWR Annotations

DWR 标注是用来代替 `dwr.xml` 或者与其一同工作的。

8.1.1 初始配置

要使用 DWR 的标注，你需要在 `web.xml` 中配置不同的 DWR 控制器。

```
<servlet>
  <description>DWR controller servlet</description>
  <servlet-name>DWR controller servlet</servlet-name>
  <servlet-class>
    org.directwebremoting.servlet.DwrServlet
  </servlet-class>
  <init-param>
    <param-name>classes</param-name>
    <param-value>
      com.example.RemoteFunctions, com.example.RemoteBean
    </param-value>
  </init-param>
</servlet>
```

`servlet` 参数 `classes` 定义的时标注的类的全名，这些名字用逗号分割。

8.1.2 远程访问类

要使一个简单的 `class` 可以成为远程访问类，你需要使用 `@Create` 和 `@RemoteMethod` 标注。

```
@Create
public class RemoteFunctions {
  @RemoteMethod
  public int calculateFoo() {
    return 42;
  }
}
```

没有被 `@RemoteMethod` 标注的方法不能被远程访问。

要在 Javascript 使用不同于类型的名字，使用 `@Create` 标注的 `name` 属性。

```
@Create(name = "Functions")
public class RemoteFunctions {
```

```
}
```

8.1.3 对象转换

要使一个简单的 bean 类可以被远程访问，使用 `@Convert` 和 `@RemoteProperty` 标注：

```
@Convert
public class Foo {
    @RemoteProperty
    private int foo;

    public int getFoo() {
        return foo;
    }

    @RemoteProperty
    public int getBar() {
        return foo * 42;
    }
}
```

要使用复杂的转换器，使用 `@Convert` 标注的 `converter` 属性。

8.2 错误和异常处理

8.2.1 错误处理

在 1.0 版中错误处理规则有些 bug，1.1 修复了这些错误。

DWR 中有一些全局的处理器(一个错误相关的，叫做 `errorHandler`，另一个警告相关的，叫做 `warningHandler`)。DWR 会默认指定一些全局处理器。你可以这样的改变全局级别的处理器：

```
DWREngine.setErrorHandler(handler);
```

你也可以指定单次调用和批量调用的错误和警告处理。例如，在调用元数据中：

```
Remote.method(params, {
    callback:function(data) { ... },
    errorHandler:function(errorString, exception) { ... }
});
```

或者,在批量元数据中:

```
DWREngine.beginBatch();
Remote.method(params, function(data) { ... });
```



```
// 其他的远程调用
DWREngine.endBatch({
  errorHandler:function(errorString, exception) { ... }
});
```

8.2.2 异常

DWR 可以转换异常，这样他们会变成 Javascript 中的错误(他们可以被抛出，因为这可能在异步调用中发生)。

例如，如果我们远程调用下面的 Java 类：

```
public class Remote {
    public String getData() {
        throw new NullPointerException("message");
    }
}
```

那么在 Javascript 中我们加入下面这些：

```
function eh(msg) {
    alert(msg);
}
{
    DWREngine.setErrorHandler(eh);

    Remote.getData(function(data) { alert(data); });
}
```

结果会通过 eh()错误处理器调用 alert 窗口的，显示消息 - 例如调用异常的 getMessage()得到的消息。

8.2.3 找出更多的信息

我们可以把整个异常传地到 Javascript 中。如果在 dwr.xml 中加入转换异常本身的能力：

```
<convert converter="bean" match="my.special.FunkyException"/>
```

在这里例子中 FunkyException 被指定，因为它不仅仅包括一个消息，它还包括一些关于异常的额外数据。例如，SQLException 包含错误号，SAX 异常包含错误的行和列等等。所以我们可以把上面的例如改为：

```
public class Remote {
    public String getData() {
        Date when = new Date();
        throw new FunkyException("message", when);
        // FunkyException有一个getWhen()方法
    }
}
```

```
}  
}
```

然后在 Javascript 中是这样的:

```
function eh(msg, ex) {  
    alert(msg + ", date=" + ex.when);  
}  
  
DWREngine.setErrorHandler(eh);  
  
Remote.getData(function(data) { alert(data); });
```

结果会是一个 `eh()` 错误处理器调用的 `alert` 框, 上面有这些信息: "message, date=Mon Jan 01 2008 10:00:00 GMT+0100"

被传递到错误处理器的 `ex` 对象会包含异常在服务端的所有属性, 但是异常栈信息没有。

8.3 传递额外的数据到 **callback** 函数

通常我们需要传递额外的数据到 **callback** 函数, 但是因为所有的回调函数都只有一个参数(远程方法的返回结果), 这就需要一些小技巧了。

解决方案就是使用 Javascript 的闭包特性。

例如, 你的回调函数原本需要像这个样子:

```
function callbackFunc(dataFromServer, dataFromBrowser) {  
    // 用dataFromServer和dataFromBrowser做些事情.....  
}
```

那么你可以像这个组织你的函数:

```
var dataFromBrowser = ...;  
  
// 定义一个闭包函数来存储dataFromBrowser的引用, 并调用dataFromServer  
var callbackProxy = function(dataFromServer) {  
    callbackFunc(dataFromServer, dataFromBrowser);  
};  
  
var callMetaData = { callback:callbackProxy };  
  
Remote.method(params, callMetaData);
```

换句话说，现在你作为 **callback** 函数传递过来的不是一个真正的 **callback**，他只是一个做为代理的闭包，用来传递客户端的数据。

你可以用更简介的形式：

```
var dataFromBrowser = ...;
Remote.method(params, {
  callback:function(dataFromServer) {
    callbackFunc(dataFromServer, dataFromBrowser);
  }
});
```

8.4 从其他的 URL 读取数据

如果你需要读取其他 **web** 应用程序生成的页面，并返回到 **Javascript** 中。非常简单。只要在你的 **Java** 类里面包括下面这写代码：

```
public String getInclude() throws ServletException, IOException{
    return WebContextFactory.get().forwardToString("/forward.jsp");
}
```

很明显你应该把 `"/forward.jsp"` 替换成你要 **forward** 到的页面的 **URL**。这个 **URL** 必须以一个斜杠开始，因为它只是调用 `HttpRequest.forward()`。

你可以用这个方法在传递之前任何定制你的页面。

8.5 安全

我们很谨慎的对待 DWR 的安全问题，并且认为有必要解释一下避免错误要做的事情。

首先 DWR 让你明确哪些是被远程调用的，是如何被远程调用。原则就是 DWR 必须调用那些你明确允许的代码。

dwr.xml 要求你为每一个远程类定义一个'create'项。你还可以通过指定 include 和 exclude 元素来更精确的控制远程调用 Bean 中可以被调用的方法。

除此之外如果你希望允许 DWR 在转换你的 JavaBean 到 Javascript 或者从 Javascript 转换到 JavaBean 时有一定的许可限制，同样可以精确控制哪些 Bean 的属性可以被转换。

一个很明显但又必须指出的 - 不要在生产环境中打开 test/debug 模式控制台。如何打开或关闭 debug 控制台在配置 web.xml 部分可以找到详细描述。

- **审查 - DWR 带来的最大好处**

很值得对比一下 DWR 和 Servlet、JSP 或周围的其他 web 框架。

如果你要审查基于 DWR 的功能，那是非常简单的。看看 dwr.xml 你就能得到一个哪些方法被暴露到外面的描述了。你也可以俯视全局用 DWR 可以访问哪些资源。

但是要在其他系统里做这件事可不是这么容易。如果是 Servlet 你需要检查 WEB-INF/web.xml 文件，然后检查写在 Servlet 中的 request.getParameter(...)。如果是 Struts 和其他 Framework 你需要检查配置文件，然后顺着流程检查代码，看请求信息出了什么问题。

- **访问控制**

DWR 允许你通过两种基于 J2EE 的机制来进行访问控制。首先你可以基于 [J2EE 角色定义 DWR 的访问](#)。其次你可以在 DWR 里面[定义访问方法的角色](#)。

- **其他方面**

DWR 不允许你定义任何内部类的 create 和 convert。这样设计是为了不出现意外的攻击来操作 DWR 的核心文件以提升访问权限。

- **风险**

有什么机会可以让攻击者窥视你的系统呢？使用 DWR 你攻击者可以使服务器创建任何你在 dwr.xml 中指定的 Java 对象的实例。并且(如果你用 BeanConverter)Java 类的任何方法、以及方法任何参数都是可见的。这些类的任何一个属性都有可能是攻击者需要的。

如果你知道 DWR 是怎么工作的，这些都是很显而易见的结论，但是往往粗心会造成问题。如果你创建了一个有 appendStringToFile()方法的 FileBean 的类，而且用 DWR 把它暴露出去，那么你就给了攻击者一个机会来填满你的文件系统。

你必须时刻注意了 DWR 以后，有没有给攻击者什么机会。

一般来说这样的情景让人感觉使用 DWR 是有风险的，但是这样的问题在所有的传统 web 架构中都存在，只是在那些架构中这些不明显，所以就很难被修复。

- **保证更加安全**

这已经很安全了，那么你还能做什么来保证更加安全了？首先记住上面这些关于审查的内容，当 web 应用的其他地方不安全时，即使它看上去很安全，过多的关注 DWR 是很愚蠢的。如果 DWR 让人感觉恐惧，那是因为它的问题都在明处。所以第一步是多检查几遍传统架构的问题。

你可以通过把可远程访问的类放到不同的包里，并且使用代理来防止 DWR 访问机制出问题。如果你愿意还可以再次检查基于角色的安全控制。这些内容只是在检查 DWR 已经为你做的事情。

比多检查几次更好的方法是检查 DWR 的源码，保证它是在正确的工作。这些代码已经被很多人检查过了，但多双眼睛总是有好处的。

- **整合 Acegi**

DWR 可以整合 Acegi security framework。更多的信息见整合 [DWR 和 Acegi](#)。

第9章. 范例精讲

9.1 购物车

该文选自于 [IBM 中国](#)

作者: Philip McCarthy (软件开发顾问)

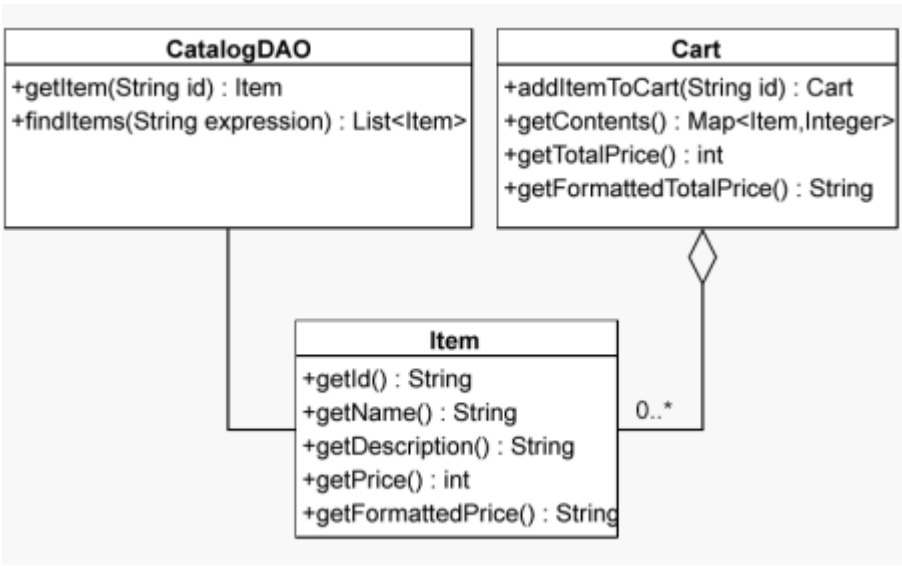
代码下载

描述	名字	大小	下载方法
DWR source code	j-ajax3dwr.zip	301 KB	FTP

9.1.1 介绍

我将采用一个基于在线商店的最小模型，这次包含一个基本的产品表示、一个可以包含产品商品的用户购物车以及一个从数据存储查询产品的数据访问对象（DAO）。Item 类与前一篇文章中使用的一样，但是不再实现任何手工序列化方法。图 1 说明了这个简单的设置：

图 1. 说明 Cart、CatalogDAO 和 Item 类的类图



在这个场景中，我将演示两个非常简单的用例。第一，用户可以在目录中执行文本搜索并查看匹配的商品。第二，用户可以添加商品到购物车中并查看购物车中商品的总价。

9.1.2 实现目录

DWR 应用程序的起点是编写服务器端对象模型。在这个示例中，我从编写 DAO 开始，用它提供对产品目录数据存储的搜索功能。`CatalogDAO.java` 是一个简单的无状态的类，有一个无参数的构造函数。清单 1 显示了我想要公开给 Ajax 客户的 Java 方法的签名：

清单 1. 通过 DWR 公开的 `CatalogDAO` 方法

```
/**
 * Returns a list of items in the catalog that have names or descriptions
 * matching the search expression
 *
 * @param expression
 *         Text to search for in item names and descriptions
 * @return list of all matching items
 */
public List<Item> findItems(String expression);

/**
 * Returns the Item corresponding to a given Item ID
 *
 * @param id The ID code of the item
 * @return the matching Item
 */
public Item getItem(String id);
```

接下来，我需要配置 DWR，告诉它 Ajax 客户应当能够构建 `CatalogDAO` 并调用这些方法。我在清单 2 所示的 `dwr.xml` 配置文件中做这些事：

清单 2. 公开 `CatalogDAO` 方法的配置

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting
2.0//EN" "http://www.getahead.ltd.uk/dwr/dwr20.dtd">
<dwr>
    <allow>
        <create creator="new" javascript="catalog">
            <param name="class"
                value="developerworks.ajax.store.CatalogDAO" />
            <include method="getItem" />
            <include method="findItems" />
        </create>
```



```
<convert converter="bean"
    match="developerworks.ajax.store.Item">
    <param name="include"
        value="id,name,description,formattedPrice" />
    </convert>
</allow>
</dwr>
```

dwr.xml 文档的根元素是 **dwr**。在这个元素内是 **allow** 元素，它指定 DWR 进行远程的类。**allow** 的两个子元素是 **create** 和 **convert**。

create 元素

create 元素告诉 DWR 应当公开给 Ajax 请求的服务器端类，并定义 DWR 应当如何获得要进行远程的类的实例。这里的 **creator** 属性被设置为值 **new**，这意味着 DWR 应当调用类的默认构造函数来获得实例。其他的可能有：通过代码段用 **Bean** 脚本框架（Bean Scripting Framework, BSF）创建实例，或者通过与 IOC 容器 **Spring** 进行集成来获得实例。默认情况下，到 DWR 的 Ajax 请求会调用 **creator**，实例化的对象处于页面范围内，因此请求完成之后就不再可用。在无状态的 **CatalogDAO** 情况下，这样很好。

create 的 **javascript** 属性指定从 JavaScript 代码访问对象时使用的名称。嵌套在 **create** 元素内的 **param** 元素指定 **creator** 要创建的 Java 类。最后，**include** 元素指定应当公开的方法的名称。显式地说明要公开的方法是避免偶然间允许访问有害功能的良好实践——如果漏了这个元素，类的所有方法都会公开给远程调用。反过来，可以用 **exclude** 元素指定那些想防止被访问的方法。

convert 元素

creator 负责公开用于 Web 远程的类和类的方法，**convertor** 则负责这些方法的参数和返回类型。**convert** 元素的作用是告诉 DWR 在服务器端 Java 对象表示和序列化的 JavaScript 之间如何转换数据类型。

DWR 自动地在 Java 和 JavaScript 表示之间调整简单数据类型。这些类型包括 Java 原生类型和它们各自的类表示，还有 **String**、**Date**、数组和集合类型。DWR 也能把 **JavaBean** 转换成 JavaScript 表示，但是出于安全性的原因，做这件事要求显式的配置。

清单 2 中的 **convert** 元素告诉 DWR 用自己基于反射的 **bean** 转换器处理 **CatalogDAO** 的公开方法返回的 **Item**，并指定序列化中应当包含 **Item** 的哪个成员。成员的指定采用 **JavaBean** 命名规范，所以 DWR 会调用对应的 **get** 方法。在这个示例中，我去掉了数字的 **price** 字段，而是包含了 **formattedPrice** 字段，它采用货币格式进行显示。

现在，我准备把 **dwr.xml** 部署到 Web 应用程序的 **WEB-INF** 目录，在那里 DWR servlet 会读取它。但是，在继续之前，确保每件事都按照希望的那样运行是个好主意。

9.1.3 测试部署

如果 `DWRServlet` 的 `web.xml` 定义把 `init-param debug` 设置为 `true`，那么就启用了 DWR 非常有帮助的测试模式。导航到 `/your-web-app/dwr/` 会把 DWR 配置的要进行远程的类列表显示出来。在其中点击，会进入指定类的状态屏幕。`CatalogDAO` 的 DWR 测试页如图 2 所示。除了提供粘贴到 Web 页面的 `script` 标记（指向 DWR 为类生成的 JavaScript）之外，这个屏幕还提供了类的方法列表。这个列表包括从类的超类继承的方法，但是只有在 `dwr.xml` 中显式地指定为远程的才标记为可访问。

Methods For: Catalog (developerworks.ajax.store.CatalogDAO)

To use this class in your javascript you will need the following script includes:

```
<script type='text/javascript' src='/dwr-app/dwr/interface/Catalog.js'></script>
<script type='text/javascript' src='/dwr-app/dwr/engine.js'></script>
```

In addition there is an optional utility script:

```
<script type='text/javascript' src='/dwr-app/dwr/util.js'></script>
```

Replies from DWR are shown with a yellow background if they are simple or in an alert box otherwise.

The inputs are evaluated as Javascript so strings must be quoted before execution.

There are 12 declared methods:

- `getAllItems()` is not available: Method access is denied by rules in dwr.xml
- `findItems(""), Execute`
- `getItem(""), Execute`
- `hashCode()` is not available: Method access is denied by rules in dwr.xml
- `getClass()` is not available: Method access is denied by rules in dwr.xml
- `wait()` is not available: Method access is denied by rules in dwr.xml
- `wait()` is not available: Method access is denied by rules in dwr.xml
- `wait()` is not available: Method access is denied by rules in dwr.xml
- `equals()` is not available: Method access is denied by rules in dwr.xml
- `notify()` is not available: Method access is denied by rules in dwr.xml
- `notifyAll()` is not available: Method access is denied by rules in dwr.xml
- `toString()` is not available: Method access is denied by rules in dwr.xml

可以在可访问的方法旁边的文本框中输入参数值并点击 **Execute** 按钮调用方法。服务器的响应将在警告框中用 JSON 标注显示出来，如果是简单值，就会内联在方法旁边直接显示。这个测试页非常有用。它们不仅允许检查公开了哪个类和方法用于远程，还可以测试每个方法是否像预期的那样工作。

如果对远程方法的工作感到满意，就可以用 DWR 生成的 JavaScript 存根从客户端代码调用服务器端对象。

9.1.4 调用远程对象

远程 Java 对象方法和对应的 JavaScript 存根函数之间的映射很简单。通用的形式是 `JavaScriptName.methodName(methodParams ..., callBack)`，其中 `JavaScriptName` 是 `creator` 的 `javascript` 属性指定的名称，`methodParams` 代表 Java 方法的 `n` 个参数，`callback` 是要用 Java 方法的返回值调用的 JavaScript 函数。如果熟悉 Ajax，可以看出这个回调机制是 XMLHttpRequest 异步性的常用方式。

在示例场景中，我用清单 3 中的 JavaScript 函数执行搜索，并用搜索结果更新用户界面。这个清单还使用来自 DWR 的 `util.js` 的便捷函数。要特别说明的是名为 `$()` 的 JavaScript 函数，可以把它当作 `document.getElementById()` 的加速版。录入它当然更容易。如果您使用过 JavaScript 原型库，应当熟悉这个函数。

清单 3. 从客户机调用远程的 `findItems()`

```
/*
 * Handles submission of the search form
 */
function searchFormSubmitHandler() {
    // Obtain the search expression from the search field
    var searchexp = $("#searchbox").value;

    // Call remoted DAO method, and specify callback function
    catalog.findItems(searchexp, displayItems);

    // Return false to suppress form submission
    return false;
}
/*
 * Displays a list of catalog items
 */
function displayItems(items) {
    // Remove the currently displayed search results
    DWRUtil.removeAllRows("items");
    if (items.length == 0) {
        alert("No matching products found");
        $("#catalog").style.visibility = "hidden";
    } else {
        DWRUtil.addRows("items", items, cellFunctions);
        $("#catalog").style.visibility = "visible";
    }
}
```

在上面的 `searchFormSubmitHandler()` 函数中，我们感兴趣的代码当然是 `catalog.findItems(searchexp, displayItems);`。这一行代码就是通过网络向 DWR servlet 发送 `XMLHttpRequest` 并用远程对象的响应调用 `displayItems()` 函数所需要的全部内容。

`displayItems()` 回调本身是由一个 `Item` 数组表示调用的。这个数组传递给 `DWRUtil.addRows()` 便捷函数，同时还有要填充的表的 ID 和一个函数数组。表中每行有多少单元格，这个数组中就有多个函数。按照顺序使用来自数组的 `Item` 逐个调用每个函数，并用返回的内容填充对应的单元格。

在这个示例中，我想让商品表中的每一行都显示商品的名称、说明和价格，并在最后一列显示商品的 `Add to Cart` 按钮。清单 4 显示了实现这一功能的单元格函数数组：

清单 4. 填充商品表的单元格函数数组

```
/*
 * Array of functions to populate a row of the items table
 * using DWRUtil's addRows function
 */
var cellFunctions = [
    function(item) { return item.name; },
    function(item) { return item.description; },
    function(item) { return item.formattedPrice; },
    function(item) {
        var btn = document.createElement("button");
        btn.innerHTML = "Add to cart";
        btn.itemId = item.id;
        btn.onclick = addToCartButtonHandler;
        return btn;
    }
];
```

前三个函数只是返回 `dwr.xml` 中 `Item` 的 `converter` 包含的字段内容。最后一个函数创建一个按钮，把 `Item` 的 ID 赋给它，并指定在点击按钮时应当调用名为 `addToCartButtonHandler` 的函数。这个函数是第二个用例的入口点：向购物车中添加 `Item`。

9.1.5 实现购物车

用户购物车的 Java 表示基于 Map。当 Item 添加到购物车中时，Item 本身作为键被插入 Map。Map 中对应的值是一个 Integer，代表购物车中指定 Item 的数量。所以 Cart.java 有一个字段 contents，声明为 Map<Item,Integer>。

使用复杂类型作为哈希键给 DWR 带来一个问题 —— 在 JavaScript 中，数组的键必须是标量的。所以，DWR 无法转换 contents Map。但是，对于购物车用户界面来说，用户需要查看的只是每个商品的名称和数量。所以我向 Cart 添加了一个名为 getSimpleContents() 的方法，它接受 contents Map 并根据它构建一个简化的 Map<String,Integer>，只代表每个 Item 的名称和数量。这个用字符串作为键的 map 表示可以由 DWR 的转换器转换成 JavaScript。

客户对 Cart 感兴趣的其他字段是 totalPrice，它代表购物车中所有商品的金额汇总。使用 Item，我还提供了一个合成的成员叫作 formattedTotalPrice，它是金额汇总的格式化好的 String 表示。

转换购物车

为了不让客户代码对 Cart 做两个调用（一个获得内容，一个获得总价），我想把这些数据一次全都发给客户。为了做到这一点，我添加了一个看起来有点儿怪的方法，如清单 5 所示：

清单 5. Cart.getCart() 方法

```
/**
 * Returns the cart itself - for DWR
 *
 * @return the cart
 */
public Cart getCart() {
    return this;
}
```

虽然这个方法在普通的 Java 代码中可能完全是多余的（因为在调用这个方法时，已经有对 Cart 的引用），但它允许 DWR 客户让 Cart 把自己序列化成 JavaScript。

除了 getCart()，需要远程化的另一个方法是 addItemToCart()。这个方法接受目录 Item 的 ID 的 String 表示，把这个商品添加到 Cart 中并更新总价。方法还返回 Cart，这样客户代码在一个操作中就能更新 Cart 的内容并接收购物车的新状态。

清单 6 是扩展的 dwr.xml 配置文件，包含 Cart 类进行远程所需要的额外配置：

清单 6. dwr.xml 加入 Cart 类

```
<createcreator="new" scope="session" javascript="Cart">
  <param name="class"
    value="developerworks.ajax.store.Cart"/>
  <include method="addItemToCart"/>
  <include method="getCart"/>
</create>
<convertconverter="bean"
  match="developerworks.ajax.store.Cart">
  <param name="include"
    value="simpleContents,formattedTotalPrice"/>
</convert>
```

在这个版本的 `dwr.xml` 中，我添加了 `Cart` 的 `creator` 和 `converter`。`create` 元素指定应当把 `addItemToCart()` 和 `getCart()` 方法远程化，而且重要的是，生成的 `Cart` 实例应当放在用户的会话中。所以，购物车的内容在用户的请求之间会保留。

`Cart` 的 `convert` 元素是必需的，因为远程的 `Cart` 方法返回的是 `Cart` 本身。在这里我指定在 `Cart` 的序列化 JavaScript 形式中应当存在的成员是 `simpleContents` 这个图和 `formattedTotalPrice` 这个字符串。

如果对这觉得有点儿不明白，那么只要记住 `create` 元素指定的是 DWR 客户可以调用的 `Cart` 服务器端方法，而 `convert` 元素指定在 `Cart` 的 JavaScript 序列化形式中包含的成员。

现在可以实现调用 `Cart` 的远程方法的客户端代码了。

9.1.6 调用远程的 **Cart** 方法

首先，当商店的 **Web** 页首次装入时，我想检查保存在会话中的 **Cart** 的状态，看是否已经有一个购物车了。这是必需的，因为用户可能已经向 **Cart** 中添加了商品，然后刷新了页面或者导航到其他地方之后又返回来。在这些情况下，重新载入的页面需要用会话中的 **Cart** 数据对自己进行同步。我可以在页面的 **onload** 函数中用一个调用做到这一点，就像这样：**Cart.getCart(displayCart)**。请注意 **displayCart()** 是一个回调函数，由服务器返回的 **Cart** 响应数据调用。

如果 **Cart** 已经在会话中，那么 **creator** 会检索它并调用它的 **getCart()** 方法。如果会话中没有 **Cart**，那么 **creator** 会实例化一个新的，把它放在会话中，并调用 **getCart()** 方法。

清单 7 显示了 **addToCartButtonHandler()** 函数的实现，当点击商品的 **Add to Cart** 按钮时会调用这个函数：

清单 7. **addToCartButtonHandler()** 实现

```
/*
 * Handles a click on an Item's "Add to Cart" button
 */
function addToCartButtonHandler() {
    // 'this' is the button that was clicked.
    // Obtain the item ID that was set on it, and
    // add to the cart.
    Cart.addItemToCart(this.itemId, displayCart);
}
```

由 **DWR** 负责所有通信，所以客户上的添加到购物车行为就是一个函数。清单 8 显示了这个示例的最后部分 —— **displayCart()** 回调的实现，它用 **Cart** 的状态更新用户界面：

清单 8. displayCart() 实现

```
/*
 * Displays the contents of the user's shopping cart
 */
function displayCart(cart) {
    // Clear existing content of cart UI
    var contentsUL = $("contents");
    contentsUL.innerHTML="";
    // Loop over cart items
    for (var item in cart.simpleContents) {
        // Add a list element with the name and quantity of item
        var li = document.createElement("li");
        li.appendChild(document.createTextNode(
            cart.simpleContents[item] + " x " + item
        ));
        contentsUL.appendChild(li);
    }

    // Update cart total
    var totalSpan = $("totalprice");
    totalSpan.innerHTML = cart.formattedTotalPrice;
}
```

在这里重要的是要记住，`simpleContents` 是一个把 `String` 映射到数字的 `JavaScript` 数组。每个字符串都是一个商品的名称，关联数组中的对应数字就是购物车中该商品的数量。所以表达式 `cart.simpleContents[item] + " x " + item` 可能就会计算出 “2 x Oolong 128MB CF Card” 这样的结果

9.1.7 演示结果

图 3 显示了这个基于 DWR 的 Ajax 应用程序的使用情况：显示了通过搜索检索到的商品，并在右侧显示用户的购物车：

图 3. 基于 DWR 的 Ajax 商店应用程序的使用情况

Ajax Store using DWR

<input type="text" value="camera"/> <input type="button" value="Search"/>			
Name	Description	Price	
Fujak Superpix172 Camera	7.2 Megapixel digital camera featuring six shooting modes and 3x optical zoom. Silver.	\$299.00	<input type="button" value="Add to cart"/>
Fujak Superpix158 Camera	5.8 Megapixel digital camera featuring six shooting modes and 2.5x optical zoom. Silver.	\$249.00	<input type="button" value="Add to cart"/>
Fujak Superpix130 Camera	3.0 Megapixel digital camera featuring six shooting modes and 2x optical zoom. Silver.	\$149.00	<input type="button" value="Add to cart"/>
Fujak Superpix145 Camera	4.5 Megapixel digital camera featuring six shooting modes and 2x optical zoom. Silver.	\$199.00	<input type="button" value="Add to cart"/>

Your Shopping Cart

2 x Bel-link USB Bluetooth dongle
1 x Fujak Superpix172 Camera
1 x Maxigate HD1200L

Total price: \$419.00

9.1.8 总结

DWR 的利弊

调用批处理

在 DWR 中，可以在一个 HTTP 请求中向服务器发送多个远程调用。调用 `DWREngine.beginBatch()` 告诉 DWR 不要直接分派后续的远程调用，而是把它们组合到一个批请求中。`DWREngine.endBatch()` 调用则把批请求发送到服务器。远程调用在服务器端顺序执行，然后调用每个 JavaScript 回调。

批处理在两方面有助于降低延迟：第一，避免了为每个调用创建 `XMLHttpRequest` 对象并建立相关的 HTTP 连接的开销。第二，在生产环境中，Web 服务器不必处理过多的并发 HTTP 请求，改进了响应时间。

现在可以看出用 DWR 实现由 Java 支持的 Ajax 应用程序有多么容易了。虽然示例场景很简单，我实现用例的手段也尽可能少，但是不应因此而低估 DWR 引擎相对于自己设计 Ajax 应用程序可以节约的工作量。在前一篇文章中，我介绍了手工设计 Ajax 请求和响应、把 Java 对象图转化成 JSON 表示的全部步骤，在这篇文章中，DWR 替我做了所有这些工作。我只编写了不到 50 行 JavaScript 就实现了客户机，而在服务器端，我需要做的所有工作就是给常规的 `JavaBean` 加上一些额外方法。

当然，每种技术都有它的不足。同任何 RPC 机制一样，在 DWR 中，可能很容易忘记对于远程对象进行的每个调用都要比本地函数调用昂贵得多。DWR 在隐藏 Ajax 的机械性方面做得很好，但是重要的是要记住网络并不是透明的——进行 DWR 调用会有延迟，所以应用程序的架构应当让远程方法的粒度比较粗。正是为了这个目的，`addItemToCart()` 才返回 `Cart` 本身。虽然让 `addItemToCart()` 作为一个 `void` 方

法可能更自然，但是这样的话对它的每个 DWR 调用后面都必须跟着一个 `getCart()` 调用以检索修改后的 `Cart` 状态。

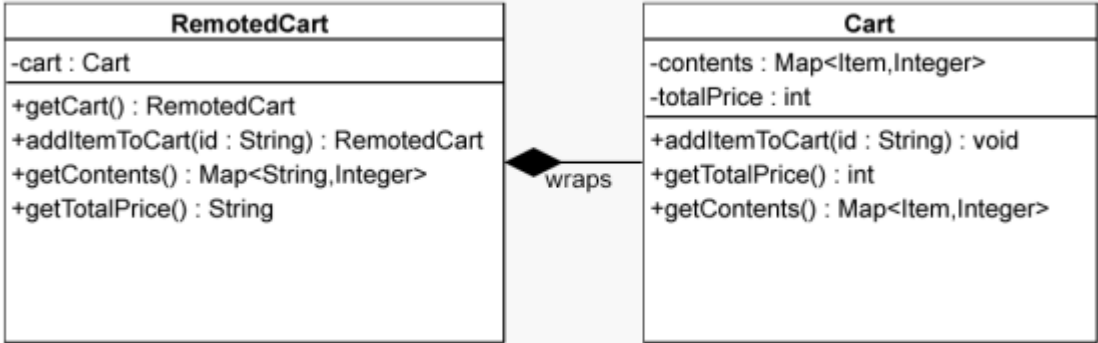
对于延迟，DWR 在调用的[批处理](#)中有自己的解决方案。如果不能为应用程序提供适当粗粒度的 `Ajax` 接口，那么只要有可能把多个远程调用组合到一个 `HTTP` 请求中，就请使用调用批处理。

分离的问题

从实质上看，DWR 在客户端和服务端代码间形成了紧密的耦合，这有许多含义：首先，远程方法 API 的变化需要在 DWR 存根调用的 `JavaScript` 上反映出来。第二（也是最明显的），这种耦合会造成对客户端的考虑会渗入服务器端代码。例如，因为不是所有 `Java` 类型都能转化成 `JavaScript`，所以有时有必要给 `Java` 对象添加额外方法，好让它能够更容易地远程化。在示例场景中，我通过把 `getSimpleContents()` 方法添加到 `Cart` 来解决这个问题。我还添加了 `getCart()` 方法，它在 DWR 场景中是有用的，但在其他场景中则完全是多余的。由于远程对象粗粒度 API 的需要以及把某些 `Java` 类型转化成 `JavaScript` 的问题，所以可以看到远程 `JavaBean` 会被那些只对 `Ajax` 客户有用的方法“污染”。

为了克服这个问题，可以使用包装器类把额外的特定于 DWR 的方法添加到普通 `JavaBean`。这意味着 `JavaBean` 类的 `Java` 客户可能看不到与远程相关联的额外的毛病，而且也允许给远程方法提供更友好的名称——例如用 `getPrice()` 代替 `getFormattedPrice()`。图 4 显示的 `RemoteCart` 类对 `Cart` 进行了包装，添加了额外的 DWR 功能：

图 4. `RemoteCart` 为远程功能对 `Cart` 做了包装



最后，需要记住：DWR `Ajax` 调用是异步的，所以不要期望它们会按照分派的顺序返回。在示例代码中我忽略了这个小问题，但是在这个系列的第一篇文章中，我演示了如何为响应加时间戳，以此作为保证数据到达顺序的一种简单手段。

第10章. 附录

10.1 常见问题

10.1.1 TransformerFactoryConfigurationError

当应用服务器启动时，报出如下错误

```
javax.xml.transform.TransformerFactoryConfigurationError:  
  Provider org.apache.xalan.processor.TransformerFactoryImpl not found  
    javax.xml.transform.TransformerFactory.newInstance(Unknown Source)
```

这个并非 DWR 导致的，是 Tomcat 没有设置属性导致的。最快的回答是[下载 Xalan](#) 并且把 xalan.jar 替换掉 \$Tomcat-HOME/common/lib 目录下的 xalan.jar。DWR2.0 比 DWR1.x 能更好地应付这个问题，但核心问题仍然是 DWR 的 XML 串行化对象需要一个 XSLT 的解析。

10.1.2 XML 解析错误

这是一开始用 DWR 会产生的常见的错误，无论对 DWR 做什么都不会有用，主要是因为 Tomcat 里面自带的 Xerces 的问题，要么是该有的时候没有，要么是不该有的时候有了。

- JDK 1.3 自身没有 XML 解析器，所以你需要 xercesImpl.jar 和 xml-apis.jar.
- JDK 1.4.0 和 JDK 1.4.1 虽然有了 XML 解析器，但是有很多 bug，所以你还是需要把 xercesImpl.jar 放到 tomcat\common\endorsed 目录下。
- JDK 1.4.2 和 JDK 5 自带的 XML 解析器工作的很好，你就不需要再加其他的了。
另外要提的一点是，不同版本的 Tomcat 需要的 XML 解析器不一样。所以要注意检查它和 JDK 的版本兼容性。

10.1.3 使用 weblogic 的类路径问题

Weblogic 8.1(有可能其他版本同样)可能找不到 DWR 的类。

这大多出现在 dwr.jar 放在 APP-INF 目录下(APP-INF/lib)的情况。在这种情况下 DWR 依然可以工作，例如 debug 页面可以看见，但是 DWR 找不到你的类。

解决办法是把 dwr.jar 放到 WEB-INF/lib 目录下。

10.1.4 没有 cookies 的情况下用 DWR

当不能用 cookies 时, servlet 规范通过 URL 重写来支持 HttpSession。DWR 2.x 通过它生成的 URL 来支持这项功能。但是 DWR 1.x 没有这个功能。你可以通过以下办法让 DWR 1.x 也支持 cookies:

- 从 dwr.jar 中提取 engine.js, 保存到你的文件系统中, 就像 jsp 文件一样.
- 修改"DWREngine._sendData = function(batch)" 方法, 加入一行:

```
statsInfo += ";jsessionid=" + <%= ""+session.getId()+" "%>
```

这样就可以让 DWR 1.x 支持 url 重写了。DWR 2+默认支持。

10.2 JavaScript 高级应用

本节选自于 IBM [developerWorks 中国](#) 以及《ajax in action》部分章节，并非本人原创，仅整理献给大家。

JavaScript 常被人们认为是编程语言中无足轻重的一员。这种观点的形成可以“归功”于其开发工具、复杂且不一致的面向 HTML 页面的文档对象模型以及不一致的浏览器实现。但 JavaScript 绝对不仅仅是一个玩具这么简单。在本文中，Bruce Tate 向您介绍了 JavaScript 的语言特性。

几乎每个 Web 开发人员都曾有过诅咒 JavaScript 的经历。这个备受争议的语言受累于其复杂的称为文档对象模型 (DOM) 的编程模型、糟糕的实现和调试工具以及不一致的浏览器实现。直到最近，很多开发人员还认为 Javascript 从最好的方面说是无可避免之灾祸，从最坏的方面说不过是一种玩具罢了。

然而 JavaScript 现在开始日益重要起来，而且成为了广泛应用于 Web 开发的脚本语言。JavaScript 的复苏使一些业界领袖人物也不得不开始重新审视这种编程语言。诸如 Ajax (Asynchronous JavaScript + XML) 这样的编程技术让 Web 网页更加迷人。而完整的 Web 开发框架，比如 Apache Cocoon，则让 JavaScript 的应用越来越多，使其不只限于是一种用于制作 Web 页面的简单脚本。JavaScript 的一种称为 ActionScript 的派生物也推动了 Macromedia 的 Flash 客户端框架的发展。运行在 JVM 上的实现 Rhino 让 JavaScript 成为了 Java™ 开发人员所首选的一类脚本语言。

我的好友兼同事 Stuart Halloway 是 Ajax 方面的专家，曾在其教授的 JavaScript 课程中做过这样的开场白：“到 2011 年，JavaScript 将被公认为是一种拥有开发现代应用程序所需的一整套新特性的语言”。他继而介绍说 JavaScript 程序要比类似的 Java 程序紧密十倍，并继续展示了使其之所以如此的一些语言特性。

在这篇文章中，我将带您探究 JavaScript 的一些特性，看看这些特性如何让它如此具有吸引力：

10.2.1 用变量操纵函数

使用 JavaScript 时，我会经常在变量或数组中存储函数，例如，查看下一例

```
<head>
  <script type='text/javascript'>
    hot = function hot() {
      alert('Sweat.')
    }
    cold = function cold() {
      alert('Shiver.')
    }

    function swap() {
      temp = hot
      hot = cold
      cold = temp
    }
  </script>
</head>
```

```

        alert('Swapped.')
    }
</script>
</head>
<body>
    <button onclick="hot();">Hot</button>
    <button onclick="cold();">Cold</button>
    <button onclick="swap();">Swap</button>
</body>

```

函数是 JavaScript 中的一类对象，可以自由地操纵它们。首先我声明两个函数：hot 和 cold。并分别在不同的变量存储它们。单击 Hot 或 Cold 按钮会调用对应的函数，生成一个告警。接下来，声明另一个函数用来交换 Hot 和 Cold 按钮的值，将此函数与第三个按钮关联，该按钮显示如图 3 所示的告警：



这个例子说明可以像处理其他变量一样处理函数。C 开发人员很容易将此概念看作是函数指针功能，但 JavaScript 的高阶函数的功能更为强大。该特性让 JavaScript 程序员能够像处理其他变量类型一样轻松处理动作或函数。

10.2.2 高阶函数

将函数用作函数的参数，或将函数作为值返回，这些概念属于高阶函数的领域。

```

<head>

<script type='text/javascript'>

    function temperature() {
        return current
    }

    hot = function hot() {
        alert('Hot.')
    }

    cold = function cold() {

```

```

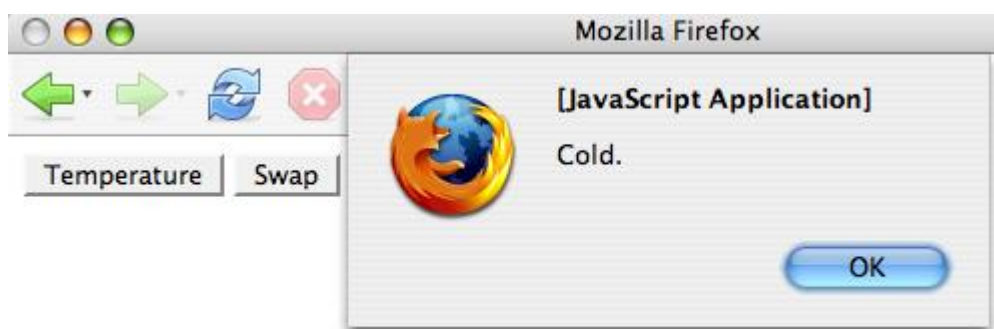
        alert('Cold.')
    }

    current = hot

    function swap() {
        if(current == hot) {
            current = cold
        } else {
            current = hot
        }
    }
}
</script>
</head>
<body>
    <button onclick="funct = temperature() ();">Temperature</button>
    <button onclick="swap();">Swap</button>
</body>

```

这个例子解决了一个常见问题：如何将更改中的行为附加到用户接口事件？通过高阶函数，这很容易做到。`temperature` 高阶函数返回 `current` 的值，而 `current` 又可以有 `hot` 或 `cold` 函数。看一下这个有些陈旧的函数调用：`temperature()()`。第一组括号用于调用 `temperature` 函数。第二组括号调用由 `temperature` 返回的函数。如下显示了输出：



高阶函数是函数式编程的基础，对比面向对象编程，函数式编程代表了更高级别的抽象。但 JavaScript 的实力并不仅限于高阶函数。JavaScript 的动态类型就极为适合 UI 开发。

10.2.3 动态类型

通过静态类型，编译器可以检查参数和变量的值或针对一个给定操作所允许的返回值。其优势是编译器可以做额外的错误检查。而且静态类型还可以为诸如 IDE 这样的工具提供更多信息，带来其他一些特性，比如更好的代码完成功能。但静态类型也存在着如下一些劣势：

- 必须提前声明意图，这常常会导致灵活性降低。例如，更改一个 **Java** 类就会更改类的类型，因而必须重新编译。对比之下，**Ruby** 允许开放的类，但更改一个 **Java** 类还是会更改类的类型。
- 要实现相同的功能，必须输入更多的代码。例如，必须用参数形式包括进类型信息，必须用函数形式返回值和所有变量的类型。另外，还必须声明所有变量并显式地转化类型。
- 静态语言的编译-部署周期要比动态语言的部署周期长，尽管一些工具可被用来在某种程度上缓解这一问题。

静态类型更适合用于构建中间件或操作系统的语言中。UI 开发常常需要更高的效率和灵活性，所以更适合采用动态类型。我深知这种做法存在危险。相信使用过 **JavaScript** 的 **Web** 开发人员都曾经为编译器本应检测到的错误类型的变量而绞尽脑汁。但它所带来的优势同样不可否认。下面将举例加以说明。

首先，考虑一个对象的情况。在清单 5 中，创建一个新对象，并访问一个不存在的属性，名为 `color`：

```
<script type='text/javascript'>
  blank_object = new Object();
  blank_object.color = 'blue'
  alert('The color is ' + blank_object.color)
</script>
```

当加载并执行此应用程序时，会得到如所示的结果：



JavaScript 并不会报告 `blue` 属性不存在的错误。静态类型的拥护者大都会被本例所吓倒，因为本例中的错误被很好地隐匿了。虽然这种做法多少会让您感觉有些不正当，但您也不能否认它巨大的诱惑力。您可以很快引入属性。如果将本例和本文之前的例子结合起来，还可以引入行为。记住，变量可以保存函数！所以，基于动态类型和高阶函数，您可以在任何时候向类中引入任意的行为。

```
<script type='text/javascript'>
  blank_object = new Object();
  blank_object.color = function() { return 'blue' }
  alert('The color is ' + blank_object.color())
</script>
```

从上例可以看出，在 **JavaScript** 的不同概念之间可以如此轻松地来回变换，其含义上的变化很大 —— 比如，是引入行为还是引入数据 —— 但语法上的变化却很小。该语言很好的延展性是它的一种优势，但同样也是其缺点所在。实际上，该语言本身的对象模型就是 **JavaScript** 延展程度的一种体现。

10.2.4 灵活的对象模型

到目前为止，您应该对 JavaScript 有一个正确的评价了，它绝非只如一个玩具那么简单。事实上，很多人都使用过其对象模型创建过极为复杂、设计良好的面向对象软件。但对象模型尤其是用于继承的对象模型又非您一贯认为的那样。

Java 语言是基于类的。当构建应用程序时，也同时构建了可以作为所有对象的模板的新类。然后调用 new 来实例化该模板，创建一个新对象。而在 JavaScript 中，所创建的是一个原型，此原型是一个实例，可以创建所有未来的对象。

现在先暂且放下这些抽象的概念，去查看一些实际代码。比如，创建了一个简单的 Animal，它具有 name 属性和 speak 动作。其他动物会从这个基础继承。

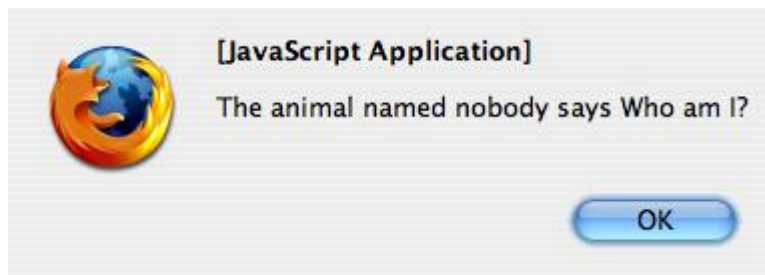
1. 构造函数

创建一个构造函数

```
<script type='text/javascript'>
Animal = function() {
    this.name = "nobody"
    this.speak = function () {
        return "Who am I?"
    }
}

myAnimal = new Animal();
alert('The animal named ' + myAnimal.name +
      ' says ' + myAnimal.speak());

</script>
```



对于 Java 开发人员而言，清单 7 中的代码看起来多少有点生疏和奇怪。实际上对于没有亲自构建过对象的许多 JavaScript 开发人员来说，这些代码同样看起来有点生疏和奇怪。也许，下面的解释可以让大家能够更好地理解这段代码。

实际上，您只需重点关注其中三段信息。首先，JavaScript 用嵌套函数表示对象。这意味着清单 7 中的 Animal 的定义是一种有效的语法。第二，JavaScript 基于原型或现有的对象的实例来构造对象，而非基于类模板。func() 是一种调用，但 new Animal() 却基于 Animal 内的原型构造一个对象。最后，在 JavaScript 中，对象只是函数和变量的集合。每个对象并不与类型相关，所以可以自由地修改这种结构。

如您所见，JavaScript 基于在 `Animal` 中指定的原型定义一个新对象：`myAnimal`。继而可以使用原型中的属性和函数，甚或重定义函数和属性。这种灵活性可能会让 `Java` 开发人员受不了，因为他们不习惯这种行为，但它的确是一种十分强大的模型。

2. 继承

现在我还要更深入一步。您还可以使用名为 `prototype` 实例变量来指定对象的基础。方法是设置 `prototype` 实例变量使其指向继承链的父。如此设置 `prototype` 之后，您所创建的对象会为未指定的那些对象继承属性和函数。这样一来，您就可以模仿面向对象的继承概念。

```
<script type='text/javascript'>

Animal = function() {
    this.name = "nobody"
    this.speak = function () {
        return "Who am I?"
    }
}

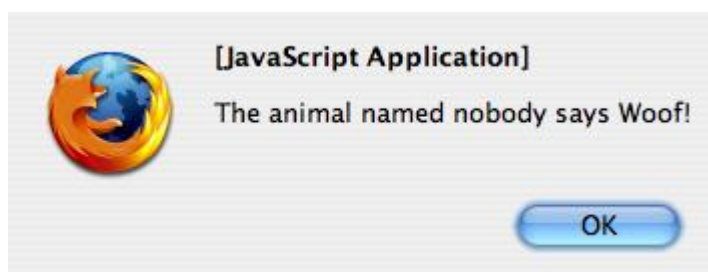
Dog = function() {
    this.speak = function() {
        return "Woof!"
    }
}

Dog.prototype = new Animal();

myAnimal = new Dog();
alert('The animal named ' + myAnimal.name +
      ' says ' + myAnimal.speak());

</script>
```

创建了一个 `Dog` 原型。此原型基于 `Animal`。`Dog` 重定义 `speak()` 方法但却不会对 `name()` 方法做任何改动。随后，将原型 `Dog` 设置成 `Animal`。图 7 显示了其结果：



这也展示了 **JavaScript** 是如何解决到属性或方法的引用问题的：

- **JavaScript** 基于原始的原型创建实例，该原型在构造函数中定义。任何对方法或属性的引用都会使用所生成的原始副本。
- 您可以在对象内像定义其他任何变量一样重新定义这些变量。这样做必然会更改此对象。所以您显式定义的任何属性或函数都将比在原始的原型中定义的那些属性或函数优先级要高。
- 如果您显式设置了名为 `prototype` 的实例变量，**JavaScript** 就会在此实例中寻找任何未定义的实例变量或属性。这种查找是递归的：如果在 `prototype` 内定义的实例不能找到属性或函数，它就会在其原型中查找，依此类推。

那么，**JavaScript** 的继承模型到底是什么样的？这取决于您如何对它进行定义。您需要定义继承行为以便可以覆盖它。然而，从本质上讲，**JavaScript** 更像是一种函数式语言，而非面向对象的语言，它使用一些智能的语法和语义来仿真高度复杂的行为。其对象模型极为灵活、开放和强大，具有全部的反射性。有些人可能会说它太过灵活。而我的忠告则是，按具体作业的需要选择合适的工具。

10.2.5 本节总结

JavaScript 对象模型构建在该语言的其他功能之上来支持大量的库，比如 **Dojo**。这种灵活性让每个框架能够以一种精细的方式更改对象模型。在某种程度上，这种灵活性是一种极大的缺点。它可以导致可怕的互操作性问题（尽管该语言的灵活性可以部分缓解这些问题）。

而另一方面，灵活性又是一种巨大的优势。**Java** 语言一直苦于无法充分增强其灵活性，原因是它的基本对象模型还未灵活到可以被扩展的程度。一个典型的企业级开发人员为能够成功使用 **Java** 语言必须要学习很多东西，而新出现的一些优秀的开放源码项目和新技术，比如面向方面编程、**Spring** 编程框架和字节码增强库，则带来了大量要学的代码。

最后，**JavaScript** 优秀的灵活性的确让您体会到了一些高阶语言的强大功能。当然您无需选择为每个项目或大多数项目都做这样的权衡和折衷。但了解一种语言的优势和劣势 —— 通过参考大量信息，而不仅仅基于广告宣传或公众意见 —— 会让您可以更好地控制何时需要使用以及何时不能使用这种语言。当您在修改 **JavaScript Web** 小部件时，您至少知道该如何让此语言发挥它最大的优势。
