

ZK 开发手册

<http://zh.zkoss.org/doc/devguide/>

ZK 开发手册

版权 © 2008 Potix Corporation. All rights reserved.

摘要

Version 3.5.1

目录

[1. 简介](#)

[传统的Web应用程序](#)

[点对点\(Ad-hoc\)AJAX应用](#)

[ZK: 它是什么](#)

[ZK: 它不是什么](#)

[ZK: 局限](#)

[2. 让我们开始吧](#)

[Hello World!](#)

[互动性](#)

[zscript元素](#)

[脚本语言](#)

[将脚本代码放在一个单独的文件中](#)

[attribute元素](#)

[EL表达式](#)

[id属性](#)

[if_和_unless属性](#)

[forEach属性](#)

[use和_apply属性](#)

[use属性](#)

[apply_属性](#)

[以_zscript实现Java类](#)

[与forward属性一起使用](#)

[手动创建组件](#)

[不使用ZUML来开发ZK应用程序](#)

[为某一页面定义新的组件](#)

[3. 基础](#)

[架构概况](#)

[执行流](#)

[组件, 页面和桌面](#)

[组件](#)
[页面](#)
[桌面](#)
[组件树的森林](#)
[组件：视觉部分和Java对象](#)
[标识](#)
[UUID](#)
[ID空间](#)
[命名空间和ID空间](#)
[在zscript中定义变量和函数](#)
[事件](#)
[桌面和事件处理](#)
[桌面及创建组件](#)
[ZUML 和XML命名空间](#)

[4. 组件活动周期](#)

[加载页面的活动周期](#)
[页面初始阶段](#)
[组件创建阶段](#)
[事件处理阶段](#)
[响应阶段](#)
[更新页面的活动周期](#)
[请求处理阶段](#)
[事件处理阶段](#)
[响应阶段](#)
[模型\(The Molds\)](#)
[组件垃圾回收](#)

[5. 事件监听及处理](#)

[通过标记语言添加事件监听器](#)
[通过程序添加或移除事件监听器](#)
[声明一个成员](#)
[动态地添加与移除事件监听器](#)
[延期事件监听器](#)
[为页面动态地添加和移除事件监听器](#)
[调用顺序](#)
[中止调用序列](#)
[事件监听器提交,发送和回显事件](#)
[提交事件](#)
[发送事件](#)
[回显事件](#)
[线程\(Thread\)模型](#)
[挂起及恢复](#)
[长操作\(Long Operations\)](#)
[初始与清理事件处理线程](#)
[处理每个事件前的初始化](#)

[处理完每个事件后清理](#)

[6. ZK用户界面标记语言](#)

[XML](#)

[元素必须格式良好](#)

[特殊字符必须被替换](#)

[属性值必须被指定且用引号包围](#)

[注释](#)

[字符编码](#)

[命名空间](#)

[条件式流程](#)

[If 和Unless](#)

[Switch和Case](#)

[Choose 和 When](#)

[反复式流程](#)

[each变量](#)

[forEachStatus变量](#)

[如何在事件监听器中使用 each和forEachStatus 变量](#)

[随机存取\(Load on Demand\)](#)

[使用fulfill属性的随机存取](#)

[使用事件监听器的随机存取](#)

[隐含对象](#)

[隐含对象列表](#)

[关于Request和Execution的信息](#)

[进程指令](#)

[page指令](#)

[component指令](#)

[init指令](#)

[variable-resolver指令](#)

[import指令](#)

[link和meta指令](#)

[ZK属性](#)

[apply属性](#)

[use属性](#)

[if属性](#)

[unless属性](#)

[forEach属性](#)

[forEachBegin属性](#)

[forEachEnd属性](#)

[fulfill属性](#)

[forward 属性](#)

[ZK元素](#)

[zk 元素](#)

[zscript元素](#)

[attribute元素](#)

[variables元素](#)
[custom-attributes元素](#)
[组件集及XML命名空间](#)
[标准的命名空间](#)

[7. ZUML页面及XUL组件集](#)

[基本组件](#)
[标签](#)
[按钮](#)
[单选按钮和单选按钮组](#)
[图像](#)
[图像映射\(Imagemap\)](#)
[音频](#)
[输入控件](#)
[日历](#)
[进度条](#)
[Slider](#)
[计时器](#)
[分页](#)
[窗口](#)
[标题](#)
[closables属性](#)
[sizable属性](#)
[样式类](#)
[contentType属性](#)
[边框](#)
[重叠，弹出，Modal，标示和嵌入](#)
[position属性](#)
[通用对话框](#)
[布局组件](#)
[嵌套的borderlayout组件](#)
[size 和 border属性](#)
[splittable 和collapsible 属性](#)
[flex 属性](#)
[open 属性](#)
[onOpen 属性](#)
[箱式模型](#)
[spacing属性](#)
[widths 和 heights 属性](#)
[分割器](#)
[Tab箱](#)
[嵌套tab box](#)
[The Accordion Tab Boxes](#)
[orient属性](#)
[Tabs的align属性](#)

[closable属性](#)
[disabled属性](#)
[Tab面板的随机存取](#)
[网格](#)
[滚动网格](#)
[可变列宽](#)
[分页网格](#)
[排序](#)
[实况数据](#)
[辅助表头](#)
[特殊属性](#)
[更多的布局组件](#)
[Separators 和空格](#)
[Group boxes](#)
[工具栏](#)
[菜单栏](#)
[执行一个菜单命令](#)
[像复选框一样使用菜单项目](#)
[autodrop属性](#)
[onOpen事件](#)
[更多的菜单特性](#)
[上下文菜单](#)
[定制的tooltip及弹出菜单](#)
[onOpen事件](#)
[列表框](#)
[多列列表框](#)
[栏头](#)
[栏尾](#)
[下拉列表](#)
[多选](#)
[滚动列表框](#)
[可变列表头](#)
[分页列表框](#)
[排序](#)
[特殊属性](#)
[实况数据](#)
[包含按钮的列表框](#)
[树控件](#)
[open属性和onOpen事件](#)
[多选](#)
[分页](#)
[特殊属性](#)
[Tree控件的打开时创建](#)
[下拉列表框](#)

[autodrop属性](#)

[description属性](#)

[onOpen事件](#)

[onChanging事件](#)

[Bandboxes](#)

[closeDropdown方法](#)

[autodrop属性](#)

[onOpen事件](#)

[onChanging事件](#)

[图表](#)

[实况数据](#)

[向下钻取\(onClick事件\)](#)

[操作区](#)

[拖放](#)

[draggable_和_dropable属性](#)

[onDrop_事件](#)

[使用多选拖曳](#)

[可拖曳组件的多种类型](#)

[HTML相关组件](#)

[html组件](#)

[Native命名空间, <http://www.zkoss.org/2005/zk/native>](#)

[XHTML命名空间, <http://www.w3.org/1999/xhtml>](#)

[include组件](#)

[style组件](#)

[script组件](#)

[iframe组件](#)

[用HTML FORM 和Java Servlets](#)

[name属性](#)

[支持name属性的组件](#)

[丰富用户界面](#)

[客户端行为](#)

[引用一个组件](#)

[onshow和onhide_行为](#)

[CSA JavaScript工具](#)

[事件](#)

[鼠标事件](#)

[按键事件](#)

[输入事件](#)

[List和Tree_事件](#)

[Slider和Scroll事件](#)

[其它事件](#)

[8. 数据绑定](#)

[基本概念](#)

[添加一个数据源](#)

- [建立数据绑定管理器](#)
- [将UI组件关联至数据源](#)
- [何时从数据源加载数据至UI](#)
- [何时从UI组件保存数据至数据源](#)
- [将相同的数据源关联至多个UI组件](#)
- [关联UI组件和一个集合](#)
- [在数据源和UI组件间定制转换](#)
- [定义数据绑定管理的访问权限](#)

[9. 在ZUML中使用XHTML组件集](#)

- [目标](#)
- [有效的XHTML页面即为有效的ZUM页面](#)
- [以服务器为中心的交互](#)
- [像平常一样使用Servlet](#)
- [差异](#)
- [为每个标签创建一个组件](#)
- [UUID即为ID](#)
- [所有标签都有效](#)
- [大小写](#)
- [无模型支持](#)
- [浏览器端的DOM树](#)
- [TABLE和TBODY标签](#)
- [事件](#)
- [与JSF, JSP及其它的集成](#)
- [使用已存在的Servlet](#)
- [使用包含丰富](#)
- [丰富一个静态的HTML页面](#)
- [使用ZK JSP标签](#)
- [使用ZK JSF组件](#)
- [使用ZK Filter丰富动态生成的页面](#)

[10. 宏组件](#)

- [使用宏组件的三个步骤](#)
- [第一步：实现](#)
- [第二步：实现](#)
- [第三步：使用](#)
- [内联宏](#)
- [一个例子](#)
- [常规宏](#)
- [宏组件和ID空间](#)
- [增设方法](#)

[11. 高级特性](#)

- [标识页面](#)
- [表示组件](#)
- [组件路径](#)
- [排序](#)

[浏览器的信息及控制](#)
[onClientInfo事件](#)
[org.zkoss.ui.util.Clients_类](#)
[防止用户关闭窗口](#)
[浏览器的历史管理](#)
[添加合适的状态到浏览器历史](#)
[监听onBookmarkChange事件并据此操作桌面](#)
[为iframe使用书签功能](#)
[简单的事例](#)
[组件克隆](#)
[组件序列化](#)
[序列化会话](#)
[序列化监听器](#)
[跨页面通信](#)
[提交和发送事件](#)
[属性](#)
[跨Web应用程序通信](#)
[来自路径的Web资源](#)
[注释](#)
[注释ZUML页面](#)
[手动注释组件](#)
[获取注释](#)
[Richlets](#)
[实现org.zkoss.zk.ui.Richlet接口](#)
[配置web.xml 和zk.xml_](#)
[会话超时管理](#)
[错误处理](#)
[加载页面时的错误处理](#)
[更新页面时的错误处理](#)
[其它](#)
[配置ZK加载器不压缩输出](#)

[12. 性能提示](#)

[使用编译过的Java代码](#)
[使用deferred_属性](#)
[deferred属性和onCreate事件](#)
[使用forward属性](#)
[使用Servlet 线程处理事件](#)
[模态窗口](#)
[消息框](#)
[文件上传](#)
[使用本地命名空间代替XHTML命名空间](#)
[延长时期\(Prolong the Period\)检查文件是否被修改](#)
[延迟子组件的创建](#)
[为大型Listbox使用实况数据和分页](#)

[使用ZK JSP标签或ZK JSF 组件代替ZK Filter](#)

[13. 其它设备和输出格式](#)

[ZK Mobile](#)

[Mobile组件集, http://www.zkoss.org/2007/mil](#)

[XML输出](#)

[使用ZUML页面输出产生XML 输出的三步](#)

[XML组件集](#)

[14. 国际化](#)

[地域](#)

[px_preferred_locale会话属性](#)

[请求拦截器](#)

[时区](#)

[px_preferred_time_zone 会话属性](#)

[请求拦截器](#)

[标签](#)

[本地文件](#)

[浏览器和本地URI](#)

[在Java中定位浏览器与本地资源](#)

[消息](#)

[主题](#)

[改变字体大小和/或样式](#)

[使用自制主题](#)

[主题提供者](#)

[15. 数据库连接](#)

[ZK仅为表现层](#)

[使用JDBC的简单方式 \(但不推荐\)](#)

[使用连接池](#)

[打开及关闭一个连接](#)

[配置连接池](#)

[易于数据库访问的ZK特性](#)

[org.zkoss.zk.ui.event.EventThreadCleanup 接口](#)

[在EL表达式中访问数据库](#)

[事务处理和org.zkoss.zk.util.Initiator_](#)

[16. 整合Hibernate](#)

[什么是Hibernate](#)

[安装Hibernate>](#)

[配置ZK的配置文件](#)

[创建Java对象](#)

[映射Java对象](#)

[使用映射文件](#)

[使用Java注释](#)

[创建Hibernate 配置文件](#)

[创建DAO 对象](#)

[在ZUML页面访问持久对象](#)

17. 整合Spring

[什么是Spring](#)

[使用Spring的准备](#)

[将spring.jar复制到你的Web library](#)

[配置web.xml](#)

[创建Spring配置文件](#)

[创建Spring Bean类](#)

[在ZUML 页面内访问 Spring Bean](#)

[使用 variable-Resolver](#)

[使用 SpringUtil](#)

[Spring Security](#)

[运行一个简单的应用程序](#)

[使用Spring Security的准备](#)

[配置/WEB-INF/web.xml 文件](#)

[创建 /WEB-INF/applicationContext-security.xml](#)

[定义哪些服务被保护](#)

[定义那些ZK事件被保护](#)

[ZUML页面](#)

18. Portal 整合

[配置](#)

[WEB-INF/portlet.xml](#)

[WEB-INF/web.xml](#)

[使用方法](#)

[zk_page 及 zk_richlet参数和属性](#)

[事例](#)

19. ZK之外

[Logger](#)

[如何使用ZK配置日志等级](#)

[i3-log.conf的内容](#)

[i3-log.conf的位置](#)

[禁用所有日志](#)

[DSP](#)

[iDOM](#)

第 1 章 简介

目录

[传统的Web应用程序](#)

[点对点\(Ad-hoc\)AJAX应用](#)

[ZK: 它是什么](#)

[ZK: 它不是什么](#)

[ZK: 局限](#)

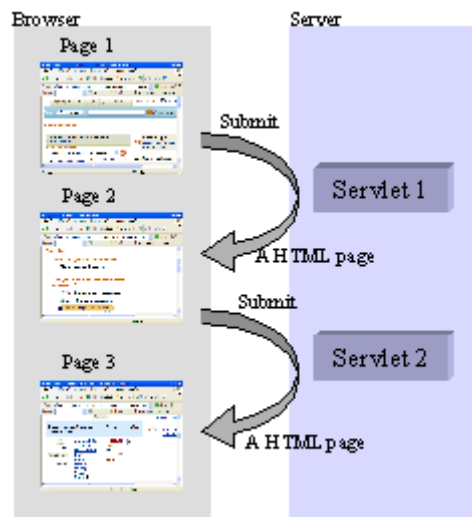
欢迎使用 ZK---- 一种丰富 Web 应用程序的最简单方式。

此开发手册描述了 ZK 的概念和功能。关于如何安装配置 ZK 的环境，请参阅 [Quick Start Guide](#)。如果想得到的各个组件(components)的属性和方法的详细描述，请参阅 [Developer's Reference](#)。

这章描述了 Web 程序，AJAX 技术和 ZK 项目的历史背景。如果你想马上了解 ZK 的功能，可以跳过这一章。

传统的Web应用程序

以简单高效交换文档为目的的 Web 技术，例如超文本传输协议(HTTP)和超文本标记语言(HTML)，都来源于单页性(page-based)和无状态(stateless-communication)的模式。在这种模式中，一个页面是自给自足(self-contained)的，并且是沟通客户端与服务器端的最小单位。

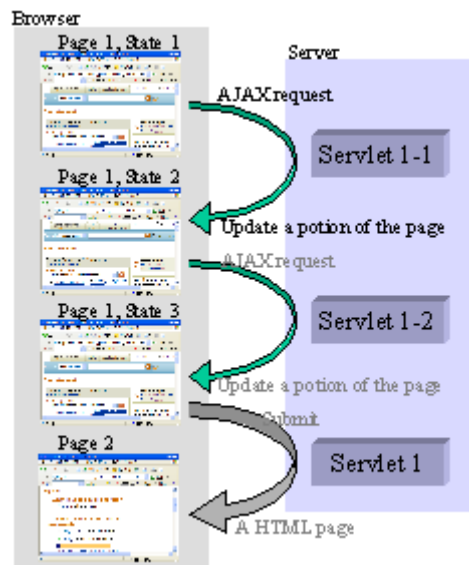


随着网络俨然成为应用开发的默认平台，这种模式面临着巨大的挑战：对于表现当今应用程序中复杂性的无能为力。举个例子，为了给客户报价，你或许必须打开另一个页面来查询此客户的交易记录，再打开一个页面来显示当前的价格，还得开一个页面来存储当前信息。用户被迫离开他正在工作的页面，并且在几个页面间来回浏览。这很容易迷失，混淆，结果是把客户弄得不愉快，销售机会的损失和低生产力。

在这种单页性(page-based)的模式上开发一个现代的应用程序也是一个极大的挑战。在这种模式中，运行在服务器上的应用程序必须处理来自从语法上分析请求，送出回应，连接用户从一个页面到另一个页面路由的一切，并且处理用户的各种错误。数十种框架，例如 **Strut**，**Tapestry** 和 **JSF**，随即出现用来简化开发过程。由于单页性(page-based)模式与现代模式之间的巨大差别，学习和使用这些框架并不是一个愉快的过程，更不要提直觉感知(intuition)和简化了。

点对点(Ad-hoc)AJAX应用

经过数十年的演变，Web应用已经从静态HTML发展到DHTML，applets，Flash，最后发展到了AJAX^[1] (Asynchronous JavaScript and XML，非同步的JavaScript和XML)。通过谷歌地图及推荐(Google Maps and Suggest)的说明，AJAX技术通过提供与桌面应用程序同等水平的互动性和反应能力给Web应用带来了新生命。不同于applets或Flash，AJAX基于标准的浏览器和JavaScript，并且不需要专门的插件。



AJAX 是新一代的 DHTML，就像 DHTML，它在很大程度上依赖于 JavaScript 监听用户活动产生的事件,然后动态的操纵浏览器中一个页面(亦称 DOM)的视觉表现。此外，它更进一步，能够使与服务器的沟通异步进行，即不需要离开或提交整个页面。它通过引入客户与服务器间轻量级的通信(light-weight communication)打破了基于页面的模式。妥善设计，AJAX 可以给 Web 应用带来丰富的桌面通用组件，而且在 Web 应用程序的活动周期内可以动态更新这些组件并通过应用程序获得对组件的更多控制。

当提供给用户需要的交互性的同时，AJAX 给已经很昂贵的 Web 应用程序开发增加了复杂性和技术先决条件。开发者不得不在浏览器中操纵 DOM，并且使用不兼容甚至是错误的 JavaScript 与服务器通信，为了更好的交互性，开发者必须重复复制应用数据和业务逻辑以便于浏览。这样就增加了维护成本且面临在服务端与客户端同步数据的挑战。

底线是在关于处理请求方面，点对点(Ad-hoc)的 AJAX 应用与传统的 Web 应用没有区别。开发者仍然必须解决由单页性(page-based)和无状态(stateless-communication)模式造成的隔阂。

^[1] Ajax是由Jesse James Garrett於Ajax: A New Approach to Web Applications中所提出的。

ZK: 它是什么

ZK 是一个事件驱动(event-driven)的, 基于组件(component-based)的, 用以丰富网络程序中用户界面的框架。ZK 包括一个基于 AJAX 事件驱动的引擎(engine), 一套丰富的 XUL 和 XHTML, 以及一种被称为 ZUML(ZK User Interface Markup Language, ZK 用户界面标记语言)的标记语言。

有了 ZK, 您可以利用 XUL 和 XHTML 的丰富特性来呈现您的 Web 应用, 操纵它们来处理因用户活动而引发的事件, 就像在桌面应用程序中那样。不同于大多数其它框架, 就 ZK 而言, AJAX 是一种幕后(behind-the-scene)技术, 组件内容的同步和流水线事件(pipelining of events)都由 ZK 引擎自动完成。

您的用户获得了如同桌面程序的互动性和反应能力, 而您的开发仍然像开发桌面应用程序那样简单。

除了简单的模型和丰富的组件, ZK也支持一种文本标记语言, 称为ZUML。ZUML, 如同HTML, 可以让开发人员设计界面而无需编程。通过XML的命名空间, ZUML无缝的集成了一套不同的标签^[2]到同一页面。目前, ZUML支持两套标签, 即XUL和HTML。

为了方便快速模型开发(prototyping)和定制, ZK允许开发人员嵌入EL表达式, 以及您喜欢的脚本语言, 包括但不限于 Java^[3], JavaScript^[4], Ruby^[5] and Groovy^[6]. 开发人员可以选择不嵌入任何脚本语言, 如果他们喜欢更严格的要求(discipline)。不同于JavaScript嵌入在HTML, ZK在服务器端执行所有的嵌入脚本。

注意一切运行在服务器端是从应用程序开发者的角度出发的。组件开发人员必须平衡互动性与简单性来决定什么任务由浏览器来完成, 而什么任务由服务器来完成。

^[2] 标签是XML元素。组件是在当ZUML网页被翻译时所产生出来的。

^[3] 使用BeanShell(<http://www.beanshell.org>)的Java interpreter。

^[4] 使用Rhino (<http://www.mozilla.org/rhino>)的JavaScript interpreter。

^[5] 使用JRuby (<http://jruby.codehaus.org/>)的Ruby interpreter。

^[6] 使用Groovy (<http://groovy.codehaus.org/>)的Groovy interpreter。

ZK: 它不是什么

ZK 并没有关注持久化(persistence)或伺服器之间的沟通(inter-server communication)。ZK 被设计的尽可能的简单,它只针对表示层(presentation tier)。他并不要求和暗示任何后端技术,所有你喜欢的中间件就像以前一样工作,如 JDBC, Hibernate, Java Mail, EJB 或 JMS。

Zk 并没有为开发人员提供(tunnel),RMI 或其他的 API 用来在客户端与服务器端通信,因为所有的代码都运行在同一服务器的同一 Java 虚拟机(JVM)上。

ZK 并没有强迫开发人员使用 MVC 或其他设计模式。是否使用它们由开发人员选择。

ZK 并不是旨在把 XUL 带入 Web 应用的框架。它的目标是把桌面编程模式引入 Web 应用。目前,它只支持 XUL 和 XHTML。将来它或许会支持 XAML, Xquery 及其它。

ZK将AJAX嵌入到了现今的应用中(implementation),但它并没有止步于AJAX结束的地方。在ZK Mobile中,您的应用程序可以到达支持J2ME的任何设备,例如PDA,手机和游戏平台。此外,您根本不用修改您的应用程序^[7]。

^[7] 根据萤幕大小有时需要做调整。

ZK: 局限

ZK 不适合在客户端运行多任务的应用程序,例如 3D 动作游戏,除非你写编写一个特殊的组件。ZK 也不适合需要大量使用客户端计算能力的应用程序。

第 2 章 让我们开始吧

目录

[Hello World!](#)

[互动性](#)

[zscript元素](#)

[脚本语言](#)

[将脚本代码放在一个单独的文件中](#)

[attribute元素](#)

[EL表达式](#)

[id属性](#)

[if 和 unless属性](#)

[forEach属性](#)
[use和 apply属性](#)
 [use属性](#)
 [apply 属性](#)
 [以 zscript实现Java类](#)
[与forward属性一起使用](#)
[手动创建组件](#)
 [不使用ZUML来开发ZK应用程序](#)
[为某一页面定义新的组件](#)

这一章的内容描述了如何写出你的第一个 ZUML 页面，如果你没时间的话建议你至少阅读这一章。

此章使用 ZUL 来说明 ZK 的功能，但是也适合于其他 ZK 支持的语言。

Hello World!

当ZK安装到你最喜爱的Web服务器^[8]后，你就可以直接编写应用程序。仅需在合适的目录新建一个名为hello.zul的文件^[9]。

```
<window title="Hello" border="normal">
    Hello World!
</window>
```

然后输入正确的URL，例如：<http://localhost/myapp/hello.zul>，得到如下页面：



在 ZUML 页面中，一个 XML 元素描述了应该创建。在这个例子中，被创建的是 `window(org.zkoss.zul.Window)`，XML 属性(attributes)用来指定 `window` 组件属性(properties)的值。在这个例子中，创建了 `window`，并指定了 `title` 和 `border` 属性的值分别为 'Hello'和'normal'。XML 元素内的文本(即 `Hello World`)也可以通过一个称为 `Label (org.zkoss.zul.Label)`的标签来展示。所以上面的例子和下面的例子是等价的：

```
<window title="Hello" border="normal">
    <label value="Hello World!"/>
</window>
```

也等价于：

```
<window title="Hello" border="normal">
  <label value="Hello World!">/label>
</window>
```

[\[8\]](#) 参考Quick Start Guide。

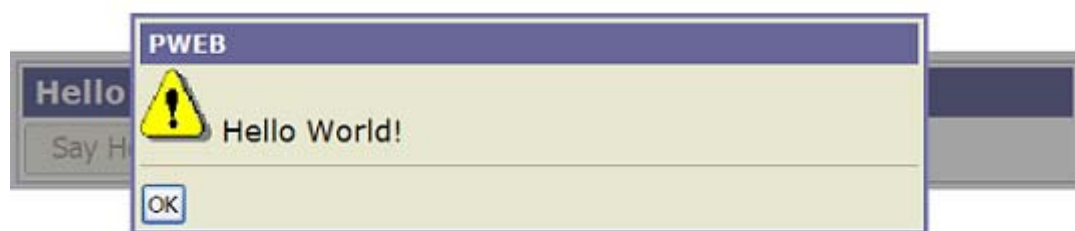
[\[9\]](#) 你也可以试试这些例子的在线示范。

互动性

让我们来添加一些互动元素：

```
<window title="Hello" border="normal">
  <button label="Say Hello" onClick="alert(&quot;Hello
World!&quot;);" />
</window>
```

点击按钮，可以看到如下效果：



onClick 是为组件添加事件监听器的一个特殊属性。这个属性的值可以是任何合法的 Java 代码。注意我们使用 `"` 来匹配双引号 (“) 以使其成为合法的 XML 文档。如果你不熟悉 XML，可以在 ZK 用户接口标记语言 (ZK User Interface Markup Language) 一章中查看有关 XML 的部分。

alert 是一个显示消息对话框的全局函数。它是调用 `org.zkoss.zul.Messagebox` 类的一个捷径。

```
<button label="Say Hello" onClick="Messagebox.show(&quot;Hello
World!&quot;);" />
```

[注]:

1. 嵌入到 ZUML 页面的脚本可以用不同的语言编写，包括但不限于 Java, JavaScript, Ruby and Groovy。此外，它们是运行在服务器上的。

2. 在运行时刻，ZK 使用 BeanShell 解释 Java，所以你可以声明全局函数，例如 `alert`。同样 它为大部分的脚本语言提供了一个简单的方式来定义全局函数，有时甚至是类。
3. 在 ZUML 页面嵌入脚本语言前，所有 `java.lang`, `java.util`,
`org.zkoss.zk.ui`, `org.zkoss.zk.ui.event` 和
`org.zkoss.zul` 包中的类都被引入。

zscript元素

`zscript` 是一个用来定义代码的元素，当 ZUML 页面被提交时被赋值(evaluated)。典型的应用包括初始化和申明全局变量与方法。

[注]: 你不可在 `zscript` 代码中使用 EL 表达式。

例如，下面的例子展示了每次按钮被按下时显示的不同信息。

```
<window title="Hello" border="normal">
  <button label="Say Hello" onClick="sayHello()" />
  <zscript>
    int count = 0;
    void sayHello() { //declare a global function
      alert("Hello World! "+ ++count);
    }
  </zscript>
</window>
```

[注]: `zscript` 仅当页面被加载时赋值(evaluated)一次，通常被用于定义函数和初始变量。

脚本语言

Java 是 ZK 默认的脚本语言，但是你可以通过指定 `language` 属性来选择不同的语言，就像下面的例子一样。`language` 属性区分大小写。

```
<zscript language="javascript">
  alert('Say Hi in JavaScript');
```

```
new Label("Hi, JavaScript!").setParent(win);
</zscript>
```

你可以像下面的例子一样使用前缀 `javascript:` 来为事件处理器指定脚本语言。注意：不要在自己指定语言的前面或后面添加空格。

```
<button onClick="javascript: do_something_in_js();" />
```

你可以在同一页面中使用不同的脚本语言。

将脚本代码放在一个单独的文件中

为了分离代码和视图(views)，开发人员可以将脚本代码放在单独的文件中，例如 `sayHello.zs`，然后使用 `src` 属性指向此文件。

```
<window title="Hello" border="normal">
  <button label="Say Hello" onClick="sayHello()" />
  <zscript src="sayHello.zs" />
</window>
```

假设 `sayHello.zs` 文件的内容如下：

```
int count = 0;
void sayHello() { //declare a global function
  alert("Hello World! " + ++count);
}
```

attribute 元素

`attribute` 元素是用来定义 XML 元素属性的元素。妥善使用，它可以使页面更具可读性。下面的例子和前面所述的 `hello.zul` 是等价的。

```
<button label="Say Hello">
  <attribute name="onClick">alert("Hello World!");</attribute>
</button>
```

你可以决定是否使用 `trim` 属性来省略属性值开头和末位的值，使用方法如下：

```
<button>
  <attribute name="label" trim="true">
    The leading and trailing whitespaces will be omitted.
  </attribute>
```

```
</button>
```

EL表达式

就像 JSP 一样，你可以在 ZUML 页面的任何部分使用 EL 表达式，但除了属性的名字 (names of attributes)，元素(elements)和处理指令(processing instruction)。

EL 表达式的语法格式为 `${expr}`，例如：

```
<element attr1="${bean.property}".../>
${map[entry]}
<another-element>${3+counter} is ${empty map}</another-element>
```

[提示]: `empty` 是用来测试一个 `map`, `collection`, `array` 或者 `string` 是否为 `null` 或空的。

[提示]: `map[entry]` 是读取 `map` 元素的一种方法，换句话说，就像 Java 中的 `map.get(entry)`。

当一个 EL 表达式作为一个属性值时，它可以返回任何类型的对象，对象的长度限制在组件可以接受的范围内。在下面的例子中，表达式被赋予一个 `Boolean` 对象的值：

```
<window if="${some > 10}">
```

[提示]: `+` 在 EL 表达式中是算数操作，并不能用于 `string` 类型。对于 `string` 可以使用 `"${expr1} is added with ${expr2}"`。

标准的隐含对象(implicit objects)，如 `param` 和 `requestScope`，还有 ZK 的隐含对象，如 `self` 和 `page`，可以很简单的使用。

```
<textbox value="${param.who} does ${param.what}"/>
```

为了引入一个方法，你可以按如下方法使用 `xel-method` 处理指令(processing instruction)。

```
<?xel-method prefix="c" name="forName"
  class="java.lang.Class"
  signature="java.lang.Class forName(java.lang.String)"?>
<textbox value="${c:forName('java.util.List')}" />
```

通过从 TLD 引入 EL 函数，你可以使用被称为 `tablib` 的指令，就像下面：

```
<?taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c" ?>
```

Developer's Reference 提供了更多关于 EL 表达式的细节。或者，你可以参考 JSP 2.0 的指南或手册来获得更多的关于 EL 表达式的信息。

id 属性

为了读取 Java 代码或 EL 表达式中的组件，你可以使用 id 属性来标识它。在下面的例子中，我们为 label 设置了一个标识，这样当一个按钮被按下时，我们就可以操纵 label 的值了。

```
<window title="Vote" border="normal">
  Do you like ZK? <label id="label"/>
  <separator/>
  <button label="Yes" onClick="label.value = self.label"/>
  <button label="No" onClick="label.value = self.label"/>
</window>
```

当按下 Yes 按钮时，可以看到如下效果：



下面是一个 EL 表达式为组件赋值的例子：

```
<textbox id="source" value="ABC"/>
<label value="${source.value}"/>
```

if 和 unless 属性

if 和 unless 属性被用于控制是否创建一个组件，在下面的例子中，两个 label 只有在请求中含有一个为 vote 的属性时才被创建：

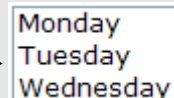
```
<label value="Vote 1" if="${param.vote}"/>
<label value="Vote 2" unless="${!param.vote}"/>
```

如果两个属性都被指定，将不会创建组件除非它们的值都被赋值为 true。

forEach属性

forEach 属性用来控制要创建多少组件，如果你为这个对象指定一个对象集合，ZK 装载机(ZK loader)将为每个被指定的集合项目创建一个组件。在下面的 ZUML 页面中，listitem 元素将被赋值三次 (分别为"Monday", "Tuesday" 和"Wednesday") 然后产生三个 list 项目。

```
<zscript>contacts = new String[] { "Monday", "Tuesday",  
"Wednesday"};</zscript>  
<listbox width="100px">  
  <listitem label="{each}" forEach="{contacts}"/>  
</listbox>
```



当使用 forEach 属性赋值时，每个变量被一个接一个的赋予来自集合的对象，即像先前接触的例子一样。因上面的 ZUML 页面和下面的是等价的：

```
<listbox>  
  <listitem label="Monday"/>  
  <listitem label="Tuesday"/>  
  <listitem label="Wednesday"/>  
</listbox>
```

另外，你也可以为 forEach 属性指定一个项目列表，通过逗号来分隔各个项目。

```
<listbox>  
  <listitem label="{each}" forEach="Monday, Tuesday,  
Wednesday"/>  
</listbox>
```

除了使用 forEach，还可以通过 forEachBegin 和 forEachEnd 来控制迭代 (iteration)，可以参考 ZK 用户标记语言一章中 ZK 属性一节获取详细信息。

use和 apply属性

在页面中嵌入代码不当会增加维护的难度，有两种途径可以从视图中分离出代码。

一种途径是你可以监听所关心的事件，然后调用合适的方法进行处理。例如，可以调用 onCreate ^[10], onOK ^[11], onCancel ^[12] 事件来完成初始化(initialize)，处理(process)和取消(cancel)等工作。

```
<window id="main" onCreate="MyManager.init(main) "
    onOK="MyManager.process(main) "
onCancel="MyManager.cancel(main)"/>
```

另外，必须有一个名称为 MyManager 的 Java 类，内容像下面一样：

```
import org.zkoss.zul.Window;

public class MyManager {
    public static void init(Window main) { //does initialization
    }
    public static void save(Window main) { //saves the result
    }
    public static void cancel(Window main) { //cancel any changes
    }
}
```

但是，上面的方法需要你在ZUML页面内嵌入一些代码。在用户界面(UI)内嵌入代码的优点是可以很容易的动态改变行为(特别是在原型阶段)，但是这仍然会展现一些维护代码且性能会有一些降低 [\[13\]](#)

use属性

若不想在 ZUML 页面内使用 Java 代码，你可以继承一个组件的实现来处理事件，如下。

```
import org.zkoss.zul.Window;

public class MyWindow extends Window {
    public void onCreate() { //does initialization
    }
    public void onOK() { //save the result
    }
    public void onCancel() { //cancel any changes
    }
}
```

然后,使用 use 属性指定类,如下。

```
<window use="MyWindow">
    ...
</window>
```

apply 属性

若你喜欢使用 MVC(模型-试图-控制者)方法, 例如, 你不想在 window(视图)内嵌入处理代码, 可以实现一个类来初始化 window。这个类必须实现 `org.zkoss.zk.ui.util.Composer` 接口。

```
import org.zkoss.zk.ui.util.Composer;
import org.zkoss.zul.Window;
public class MyComposer implements Composer {
    public void doAfterCompose(Component comp) {
        ((Window)comp).setTitle("My Title"); //do whatever
        initialization you want
        //comp is Window since we will specify it to a window later
    }
}
```

在这里我们假设你有三个监听器, `MyCreate`, `MyOK`, 和 `MyCancel`。参考下面的事件章节获取事件监听器的解释。

然后, 使用 `apply` 属性指定类, 如下。

```
<window apply="MyComposer">
...
</window>
```

`window` 仍然作为 `org.zkoss.zul.Window` 的一个实例被创建, 且作为 `comp` 参数被传递给 `doAfterCompose` 方法。然后, 你可以处理你所希望的初始化。

若你想 `apply` 多个 `composer`, 使用逗号隔开。另外, 你可以使用 EL 表达式来返回类, 它的名称, `Composer` 实例, 或 `Composer` 实例的集合。

```
<window apply="MyComposer, AnotherComposer">
    <textbox apply="${c:mycomposer()}" />
</window>
```

以 `zscript` 实现 Java 类

在 `zscript` 中继承 Java 类, 多亏了 `BeanShell`^[14] 的强大功能, Java 类的继承可以按如下的方式完成:

```
<zscript>
    public class MyWindow extends Window {
    }
```

```
</zscript>
<window use="MyWindow"/>
```

[提示]: 很多脚本语言, 例如 JRuby, 也允许开发人员定义可以被 Java 虚拟机(JVM) 存取的类, 请参考相应的手册来了解详情。

为了从视图中分离代码, 你可以把所有的 zscript 代码放到单独的文件中, 称为 mywnd.zs, 然后:

```
<zscript src="/zs/mywnd.zs"/>
<window use="MyWindow"/>
```

[提示]: 你也可以使用初始化指令(`initdirective`)来指定 zscript 文件的位置。不同的是初始化指令在所有组件被创建前被赋值(页面初始阶段)。如需更多信息请参考 ZK 用户界面标记语言一章中 `init` 指令一节。

[10] 当ZUML中window产生时onCreate事件会被送出。

[11] 使用者按下ENTER key时onOK事件被送出。

[12] 使用者按下ESC key时onCancel事件被送出。

[13] Java解释器会在运行时解释ZUML页面内指定的代码。

[14] <http://www.beanshell.org>

与forward属性一起使用

window 通常由一些按钮, 菜单项目和其他组件组成。例如,

```
<window use="MyWindow">
  ...
  <button label="OK"/>
  <button label="Cancel"/>
</window>
```

当用户点击按钮时, `onClick` 事件会被送至按钮本身。但是这些事件最好在 `window` 内处理而不是散落这些按钮。为了这样, 你可以按如下方式使用 `forward` 属性。

```
<window use="MyWindow">
  ...
  <button label="OK" forward="onOK"/>
```



```
<button label="Cancel" forward="onCancel"/>
</window>
```

在这里 **OK** 按钮的 `forward` 属性指定接收 `onClick` 事件后将其作为 `onOK` 事件转向空间所有者(例如, `window`)。同样, 针对 **Cancel** 按钮的 `onClick` 事件会转向 `onCancel` 事件。因此你可以在 `MyWindow` 命名空间内处理 `onOK` 和 `onCancel` 事件, 如下。

```
public class MyWindow extends Window {
    public void onOK() {
        //called when the OK button is clicked (or the ENTER button
is pressed)
    }
    public void onCancel() {
        //called when the Cancel button is clicked (or the ESC button
is pressed)
    }
}
```

除了将 `onClick` 事件 `forward` 至空间所有者, 你可以使用 `forward` 属性将任何事件 `forward` 至任何组件。参考 **ZK** 用户界面标记语言一章中 `forward` 属性一节。

手动创建组件

除了讲述什么组件可以在**ZUML**页面被创建外, 开发人员可以手动创建组件。所有的组件都是具体的(**concrete**), 你可以直接^[15]通过它们的构造函数(**constructors**)来创建它们。

```
<window id="main">
    <button label="Add Item">
        <attribute name="onClick">
            new Label("Added at "+new Date()).setParent(main);
            new Separator().setParent(main);
        </attribute>
    </button>
    <separator bar="true"/>
</window>
```

当一个组件被手动创建时, 它并没有自动被加到页面。换句话说, 它并不在用户的浏览器中出现。为了将它加到页面, 你可以调用 `setParent`, `appendChild` 或 `insertBefore` 方法来为其指定一个父类(**parent**), 如果父类组件是页面的一部分, 那么它也变成了页面的一部分。

组件类并没有`destroy` 或`close`方法^[16], 当一个组件从页面中被拆卸的时候就会从浏览器中内移除。它表现的就像附着在页面上一样。

```
<window id="main">
  <zscript>Component detached = null;</zscript>
  <button id="btn" label="Detach">
    <attribute name="onClick">
      if(detached != null) {
        detached.setParent(main);
        detached = null;
        btn.label = "Detach";
      } else {
        (detached = target).setParent(null);
        btn.label = "Attach";
      }
    </attribute>
  </button>
  <separator bar="true"/>
  <label id="target" value="You see this if it is attached."/>
</window>
```

在上面的例子中, 你可以用 `setVisible` 方法来产生类似的效果。但是 `setVisible(false)` 方法并没有把组件从浏览器中移除, 它只是使一个组件(及其所有的子组件(children))变得不可见。

当一个组件从页面被卸载后, 如果应用程序没有涉及到该组件, 它所占用的内存会被 Java 虚拟机的垃圾回收机制(JVM's garbage collector)所释放。

不使用ZUML来开发ZK应用程序

对于根本不习惯使用 ZUML 的开发人员来说, 他们可以使用被称为 `richlet` 的方法来手动创建所有的组件。

```
import org.zkoss.zul.*;
public class TestRichlet extends org.zkoss.zk.ui.GenericRichlet
{
    public void service(Page page) {
        page.setTitle("Richlet Test");

        final Window w = new Window("Richlet Test", "normal", false);
        new Label("Hello World!").setParent(w);
        final Label l = new Label();
        l.setParent(w);
        //...
    }
}
```

```
w.setPage(page);  
}  
}
```

请参考高级特性一章中 Richlets 一节。

[15] 为了简化，这里不用 factory design pattern。

[16] 与 W3C DOM 的观念相近。而另一方面，Windows API 需要程序员管理生命周期。

为某一页面定义新的组件

就像所展示的那样，通过使用 XML 属性为组件指定一个属性是很容易的事情。

```
<button label="OK" style="border:1px solid blue"/>
```

ZK 提供了一种强大但很简单的方式来让开发人员为某一页面定义新的组件，如果同一类型的大多数组件共享一套属性，这就非常有用。

首先使用组件指令来定义一个新的组件。

```
<?component name="bluebutton" extends="button" style="border:1px  
solid blue" label="OK"?>  
  
<bluebutton/>  
<bluebutton label="Cancel"/>
```

等价于：

```
<bluebutton style="border:1px solid blue" label="OK"/>  
<bluebutton style="border:1px solid blue" label="Cancel"/>
```

此外，你可以用下面的方式来覆盖 button 组件的定义，当然，这不会影响到其他的页面：

```
<?component name="button" extends="button" style="border:1px  
solid blue" label="OK"?>  
  
<button/>  
<button label="Cancel"/>
```

如需要更多的信息，请参考 ZK 用户界面标记语言一章中 component 指令一节。

第 3 章 基础

目录

[架构概况](#)

[执行流](#)

[组件，页面和桌面](#)

[组件](#)

[页面](#)

[桌面](#)

[组件树的森林](#)

[组件：视觉部分和Java对象](#)

[标识](#)

[UUID](#)

[ID空间](#)

[命名空间和ID空间](#)

[在zscript中定义变量和函数](#)

[事件](#)

[桌面和事件处理](#)

[桌面及创建组件](#)

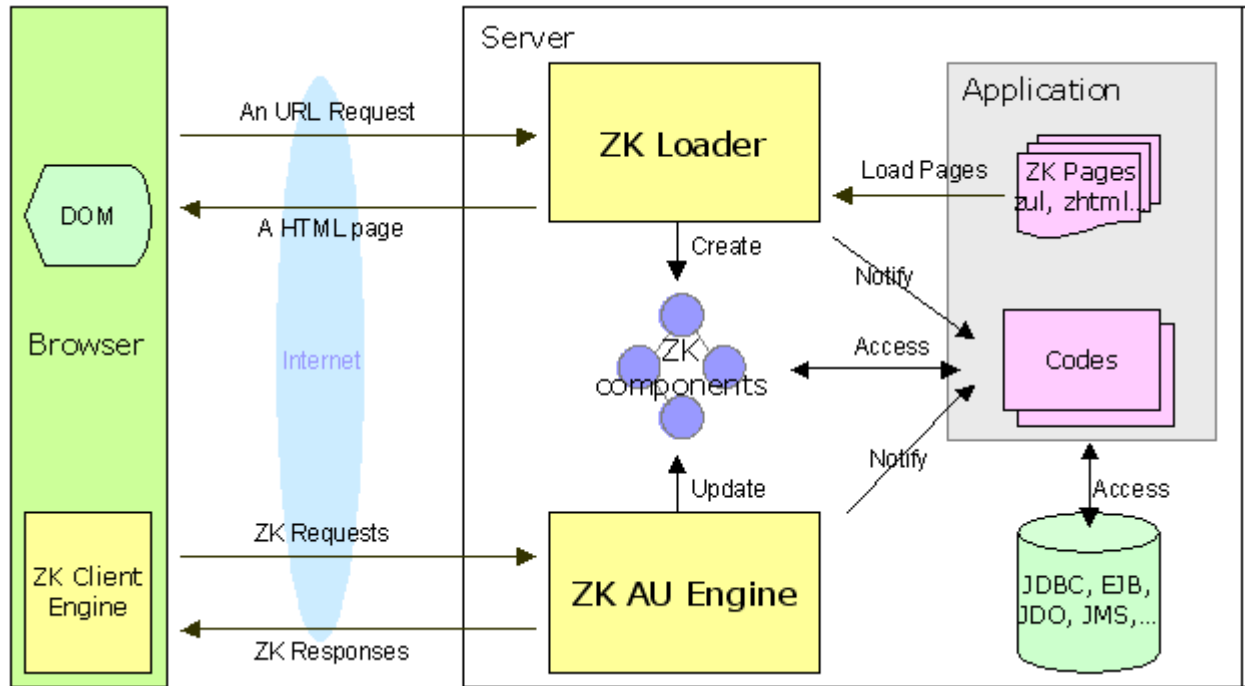
[ZUML 和XML命名空间](#)

这一章讲述了 ZK 的基础。这里使用了 XUL 来说明 ZK 的功能，但同样可用于 ZK 支持的其他标记语言。

架构概况

ZK 包括一种基于 AJAX 机制用来实现自动的交互性，一套丰富的基于 XUL 的组件用以丰富可用性，和一种的标记语言用来简化开发。

基于AJAX的机制包括三个部分，描绘如下：ZK 加载器(ZK loader)，ZK AU引擎(ZK AU Engine) ^[17]，和ZK客户端引擎(ZK Client Engine)。



基于用户的请求，ZK 加载器(ZK loader)加载一个 ZK 页面，解释它，并将结果送到 HTML 页面来响应 URI 请求。ZK 页面是用一种被称为 ZUML 的标记语言写成的。ZUML，就像 HTML，被用来描述什么组件被创建，以及如何把它们呈现出来。这些组件一旦被创建，就会一直处于可用状态知道会话超时。

然后ZK AU^[18] 引擎(ZK AU Engine)和ZK客户端引擎(ZK Client Engine)作为投手和捕手一起工作。它们将在浏览器端发生的事件送到运行在服务器端的应用程序，然后更新浏览器段的DOM树，基于组件如何被应用程序操纵。这种方式即所谓的事件驱动编程模型。

执行流

1. 当用户在浏览器中键入一个URL或点击一个超链接时，一个请求便被送到了 Web服务器，如果URI符合ZK的配置^[19]，ZK 加载器则援引担任这一要求。
2. ZK 加载器(ZK loader)加载指定的页面然后解释它，以据此创建和适的组件。
3. 当解释完整个页面后，ZK 加载器(ZK loader)将结果送到一个HTML页面。然后这个HTML页面被送回浏览器和ZK客户端引擎(ZK Client Engine)^[20]一起。
4. ZK客户端引擎(ZK Client Engine)坐落在浏览器，以监视由客户的活动触发的事件，例如挪动鼠标，或改变某个值。一旦监测到，它就通知ZK AU引擎通过发送一个ZK请求^[21]。
5. 当从客户端引擎接到 ZK 请求后，如果有需要的话 AU 引擎就更新相应组件的内容。然后，AU 引擎通过调用相关的事件处理程序(如果有的话)来通知应用程序。
6. 如果应用程序选择改变组件的内容，添加或移动组件，AU 引擎通过 ZK 响应(ZK responses)将更新后组件的新内容送至客户端引擎。
7. 这些 ZK 响应实际上是一些命令，这些命令指示客户端引擎如何更新 DOM 树的内容。

[17] 同ZK Update引擎

[18] AU 即Asynchronous Update，异步更新。

[19] 参考 the Developer's Reference中的Appendix A 。

[20] ZK客户端引擎(ZK Client Engine)是由JavaScript语言编写的。浏览器缓存ZK客户端引擎，所以通常仅需在首次读取时设置引擎。

[21] ZK请求(ZK requests)是一种特殊的AJAX 请求。 但是，对于mobile版本，ZK 请求是一种特殊的HTTP 请求。

组件，页面和桌面

组件

组件即一个用户界面(UI)对象，如标签，按钮和树。它用来定义 一个特定用户界面的视觉表现和行为。通过操纵他们， 开发 人员来控制如何在客户端展示一个应用程序。

组件必须实现 `org.zkoss.zk.ui.Component` 接口。

页面

页面(`org.zkoss.zk.ui.Page`)是一系列组件的集合，一个页面限制属于它的组件，这样它们会被展示在浏览器的特定部分。一个页面被自动创建当 ZK 加载器(ZK loader)解释完一个 ZUML 页面时。

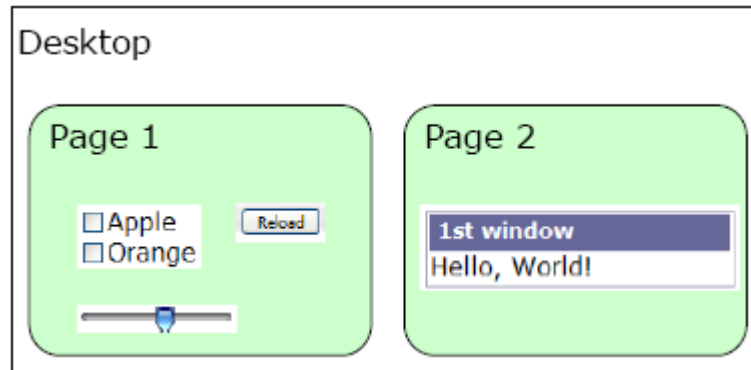
页面标题

每个页面都可以有一个标题，这个标题将被作为浏览器窗口标题(window caption)的一部分。请参考 ZK 用户界面标记语言一章中进程指令一节。

```
<?page title="My Page Title"?>
```

桌面

一个 ZUML 可以直接或间接包含另外一个 ZUML 页面。由于这些页面是为服务同样的 URL 请求而创建的,它们被统称为桌面(org.zkoss.zk.ui.Desktop)。换句话说,桌面是页面的集合,这些 页面服务于同样的 URL 请求。



为了实现 Zk 应用程序与用户的交互,更多的页面需要被加到桌面,而另一些需要从桌面移除。类似的,一个组件可能被加到页面或从页面移除。

createComponents 方法

请注意创建移除页面和桌面都是在背后达成的。并没有关于此的 API。每次 ZUML 加载一个页面时,这个页面就被创建。当 ZK 发现某个页面不再被用到时这个页面就会被移除。当第一个 ZUML 页面被加载时桌面被创建。当太多的桌面为特定的会话而创建时桌面会被移除。

org.zkoss.zk.ui.Executions 类中的 CreateComponents 方法仅能创建组件,而不是页面,即使它装载一个 ZUML 页面(亦=page)。

组件树的森林

一个组件至多有一个父组件,而可能有多个子组件。一些组件只能接受某些类型的组件作为子组件,一些组件必须为某些类型组件的子组件,一些组件根本不允许有子组件。例如,XUL 中的 Listbox 接受 Listcols 和 Listitem。没有任何父组件的组件称为根组件(root component)。一个页面可能有多个根组件,这些组件可以由 getRoots 方法获得。

组件：视觉部分和Java对象

除了在服务器端的Java对象，组件在浏览器端有一个可视部分^[22]，并且当且仅当它属于一个页面时。当一个组件附加到一个页面时，其视觉部分就会被创建^[23]。当一个组件从一个页面脱离时，其视觉部分被移除。

有两种方法将一个组件附加到一个页面。第一种，你可以调用 `setPage` 方法使一个组件成为指定页面的根组件。第二中，你可以调用 `setParent`，`insertBefore` 或 `appendChild` 方法来使其成为另外一个的子组件，那么属于同一页面的子组件将会属于父组件所属于的页面。

同样，你可以通过调用 `setPage` 方法并将其参数设为 `null` 将一个根组件从页面卸载。当一个子组件从父 组件被卸载或其父组件从页面被卸载时，此组件被卸载。

标识

每个组件都有一个标识(可利用 `getId` 方法获得)，当一个组件被创建时它被自动创建。开发人员可以在任何时候改变它。对于标识如何命名并没有限制。但是，如果一个字母标识(alphabetical identifier)被指定之后，开发人员可以通过 Java 代码或 嵌入到 ZUML 页面的 EL 表达式直接读取到它。

```
<window title="Vote" border="normal">
  Do you like ZK? <label id="label"/>
  <separator/>
  <button label="Yes" onClick="label.value = self.label"/>
  <button label="No" onClick="label.value = self.label"/>
</window>
```

UUID

一个组件有另外一个称为 UUID 的标识，但应用程序开发人员很少用到它。

UUID 是被组件和客户端引擎用来操纵浏览器端的 DOM 以及和服务器进行通信的。更确切地说，客户端 DOM 元素的 `id` 属性就是 UUID。

当一个组件被创建时，UUID 会自动产生。除了用以呈现 HTML 标签的组件标识，它是一成不变的。

相关的 HTML 组件处理 UUID 的方式不同于其他的一套组件：UUID 就像 ID 一样。如果你改变一个 HTML 相关组件的 ID，UUID 就会跟着改变。因此原来的 JavaScript 代码和 `servlets` 将会继续工作，而无需任何修改。

[22] 如果客户端为浏览器，视觉表现为一个 DOM 元素 或一个套DOM 元素。

[23] 可是部分是自动创建，更新和移除的。应用程序开发人员很少需要注意到它的存在。他们在服务器端操纵对象部分（object part）。

ID空间

将视觉表现(visual presentation)分为几个 ZUML 页面是很常见的。例如，一个页面用来展示订购单，一个对话框用于进入付款期。如果同一个桌面内所有的组件都是非常明确的，开发人员必须为这个桌面内所有页面维护所有标识的唯一性。为了解决这个问题，ID 空间的概念被引入。一个 ID 空间是一个桌面的组件的子集。唯一性只在 ID 空间的范围内有保障。

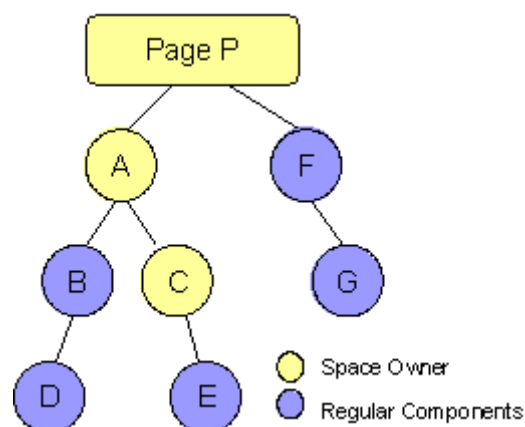
Id 空间的最简单形式是一个 window(org.zkoss.zul.Window)。所有 window 衍生出来的组件(包括 window)形成了一个独立的 ID 空间。因此，你可以将 window 作为每个页面的最高组件使用，这样，开发人员需要维护每个页面的唯一性。

更一般地说，任何组件可形成一个 ID 空间，只要它实现了 org.zkoss.zk.ui.IdSpace 接口。页面(Page)也实现了这个接口，所以它又是个空间所有者(space owner)。

一个 ID 空间的最高组件即为空间的所有者，可以使用 Component 接口中的 getSpaceOwner 方法来获得这个组件。

如果一个称为 X 的 ID 空间，从另外一个称为 Y 的 ID 空间衍生而来，那么 X 的所有者是空间 Y 的一部分，但从 X 衍生出来的部分并不是空间 Y 的一部分。

就像在图中描绘的一样，有三个空间：P，A 和 C。空间 P 包括 P，A，F 和 G。空间 A 包括 A，B，C 和 D。空间 C 包括 C 和 E。



在相同 ID 空间内的组件称为 fellows, 例如 A, B, C 和 D 就是同一 ID 空间内的 fellows。

为了获得另一个 fellow, 可以使用 IdSpace 或 Component 接口中的 getFellow 方法。

请注意可以 getFellow 方法可以被同一 ID 空间内任何组件调用, 并不仅限于空间所有者。同样, 对于在同一空间内的任何组件, getSpaceOwner 方法返回的是同样的对象, 与是否是空间所有者无关。

org.zkoss.zk.ui.Path 类提供了在 ID 空间内简化定位组件的工具。其使用凡是类似 java.io.File

```
Path.getComponent("/A/C/E");
new Path("A/C", "E").getComponent();
```

命名空间和ID空间

为了能让解释器(interpreter)直接读取到组件, 命名空间(org.zkoss.scripting.Namespace)的概念被引入。首先, 每一个 ID 空间都有一个确切的命名空间。第二, 定义在命名空间内的变量对于属于 同一个命名空间的脚本代码及 EL 表达式是可见的。

```
<window border="normal">
  <label id="l" value="hi"/>
  <zscript>
    l.value = "Hi,
namespace";
  </zscript>
  ${l.value}
</window>
```

Hi, namespace! Hi, namespace!

下面的例子有两个命名空间, 一个属于window w1, 另一个属于window w2。b1 按钮的onClick事件针对window w1 内定义的label, 而b2 按钮的onClick事件是针对窗口window w2 [\[24\]](#)内的定义的checkbox。

```
<window id="w1">
  <window id="w2">
    <label id="c"/>
    <button id="b1" onClick="c.value = &quot;OK&quot;"/>
  </window>
  <checkbox id="c"/>
  <button id="b2" onClick="c.label = &quot;OK&quot;"/>
</window>
```

请注意命名空间是有等级的。换言之，`window w2` 中的 `zscript` 可以看见 `window w1` 中的组件，除非她凌驾于 `window w2`。因此，在下面的例子中 `button b1` 将会改变标签 `c`。

```
<window id="w1">
  <window id="w2">
    <button id="b1" onClick="c.value = &quot;OK&quot;"/>
  </window>
  <label id="c"/>
</window>
```

除了 `ZK` 指定的添加到命名空间的组件，你可以指定自己的变量通过使用 `setVariable` 方法，这样 `zscript` 可以直接参考(reference)它们。

在`zscript`中定义变量和函数

除了执行代码,你可以在 `zscript` 元素中直接定义变量和函数，就像下面描绘的一样：

```
<window id="A">
  <zscript>
    Object myvar = new LinkedList();
    void myfunc() {
      ...
    }
  </zscript>
  ...
  <button label="add" onClick="myvar.add(some)"/>
  <button label="some" onClick="myfunc()"/>
</window>
```

在 `zscript` 中定义的变量和函数存储在相应脚本语言的解释器(interpreter)中。

`zscript`和EL表达式

就像命名空间^[25]一样，定义在`zscript`中的变量对于EL表达式都是可见的。

```
<window>
  <zscript>
    String var = "abc";
    self.setVariable("var2", "xyz", true);
  </zscript>
  ${var} ${var2}
</window>
```

等价于：

```
<window>
abc xyz
</window>
```

请注意，定义在 `zscript` 中的变量比定义在命名空间中的变量有更高的优先级。

```
<window>
  <zscript>
    String var = "abc";
    self.setVariable("var", "xyz", true);
  </zscript>
  ${var}
</window>
```

等价于：

```
<window>
abc
</window>
```

但如果你之后声明了一个同名组件，这就会令人困惑，就像下面展示的那样。

```
<window>
  <zscript>
    String var = "abc";
  </zscript>
  <label id="var" value="A label"/>
  ${var.value} <!-- Wrong! var is "abc", not the label -->
</window>
```

因此，建议使用一些命名方式来避免这种困惑。例如，你可以为所有的解释(interpreter)变量加上前缀 `zs_`。

另外，应该尽量使用局部变量。局部变量是和类名一起被声明的，并且只对某一范围的 `zscript` 代码可见。

```
<zscript>
Date now = new Date();
</zscript>
```

你可以通过将其放在 `{ }` 中使局部变量对于 EL 表达式不可见，如下：

```
<zscript>
{ //create a new logic scope
```

```
String var = "abc"; //visible only inside of the enclosing curly
brace
}
</zscript>
```

多范围(Multi-Scope)的解释器

依靠实现，一个解释器或许有确切的范围，或每个 ID 空间一个逻辑范围来存储这些变量和方法。出于 以上的描述，我们将它们分别称为单范围和多范围的解释器 (single-scope and multi-scope interpreters)。

Java解释器(BeanShell)是一个典型的多范围解释器。^[26]。它为每个ID空间创建一个独立的解释范围。例如，在下面的例子中分别为window A和B创建两个逻辑范围。因此在下面的例子中，var2 仅对于window B是可见的，var1 对于窗口A和B都是可见的。

```
<window id="A">
  <zscript>var1 = "abc";</zscript>
  <window id="B">
    <zscript>var2 = "def";</zscript>
  </window>
</window>
```

Java解释器(BeanShell)

通过 Java 解释器，你可以为一个最近 ID 空间(例如一个窗口 window)的逻辑范围声明一个局部解释变量(interpreter variable)通过指定类名，如下例所示：

```
<window id="A">
  <window id="B">
    <zscript>
      String b = "local to window B";
    </zscript>
  </window>
</window>
```

下面是一个更复杂的例子，可以产生 abc def。

```
<window id="A">
  <zscript>
    var1 = var2 = "abc";
  </zscript>
  <window id="B">
```

```
<zscript>
Object var1 = "123";
var2 = "def";
var3 = "xyz";
</zscript>
</window>
${var1} ${var2} ${var3}
</window>
```

Object var1 ="123"实际上是为 window B 创建了一个局部变量，对象是指定的。另一方面，var2 ="def"会使解释器(interpreter)在当前或更高的范围内寻找名称为 var2 的变量。var2 变量在 window A 内已被定义，变量在此被重定义(overridden)。var3 ="xyz "为窗口(window)B 创建了一个局部变量，而 window A 并没有定义任何叫做 var3 的变量。

单范围(Single-Scope)解释器

Ruby, Groovy 和JavaScript解释器(Interpreters)并不支持多范围^[27]。这就意味着定义的所有变量，就是说， Ruby存储在一个逻辑范围内(每一个解释器)。因此，定义在一个窗口中的解释变量(interpreter variables)会覆盖定义在另一个窗口中的变量，如果它们在同一个页面内。为了避免这种困惑，你可以为每个变量的名字加上与窗口相关的特殊前缀。

[提示]:每个页面都有它自己的解释器(interpreter)来为 zscript 代码赋值，如果一个桌面有多个页面，那么它或许有多个解释器的实例(instances of the interpreters)(每一种脚本语言)。

在一个页面中使用多种脚本语言

每种脚本语言都与一种解释器(interpreter)相关联。因此，定义在一种语言中的变量和方法对于另外一种语言是不可见的。例如在下面的例子中，变量 var 1 和 var2 属于两种不同的解释器(interpreter)。

```
<zscript language="Java">
    var1 = 123;
</zscript>
<zscript language="JavaScript">
    var2 = 234;
</zscript>
```

getVariableVS getZScriptVariable

可以通过 `getVariable` 方法获得定义在命名空间内的变量。另一方面，定义在 `zscript` 中的变量是解释它的解释的一部分，它们不是任何命名空间的一部分。换句话说，你 cannot 通过 `getVariable` 方法获取它。

你必须使用 `getZScriptVariable` 方法来获得 `zscript` 中的定义的变量。同样，可以使用 `getZScriptClass` 和 `getZScriptMethod` 方法来获取定义在 `zscript` 中的类和方法。这些方法将会遍历所有的被加载的解释器直到指定的一个被找到。

如果你想找到某个解释器，可以使用 `getInterpreter` 方法先获得解释器，就像下面一样：

```
<zscript>
    var1 = 123; //var1 belongs to the interpreter, not any namespace
    page.getVariable("var1"); //returns null
</zscript>
```

相反，你必须使用 `getZScriptVariable` 方法来获得 `zscript` 中的定义的变量。同样，可以使用 `getZScriptClass` 和 `getZScriptMethod` 方法来获取定义在 `zscript` 中的类和方法。这些方法将会遍历所有的被加载的解释器(**interpreter**)知道指定的一个被找到。

如果你想找到某个解释器，可以使用 `getInterpreter` 方法先获得解释器，就像下面一样：

```
page.getInterpreter("JavaScript").getVariable("some");
//interpreter for JavaScript
page.getInterpreter(null).getVariable("some"); //interpreter for
default language
```

^[24] `window` 实现了 `org.zkoss.zk.ui.IdSpace`，所以它形成了一个独立的ID空间和命名空间。

^[25] `org.zkoss.zk.scripting.Namespace`

^[26] Java 解释器支持多范围(**multi-scope**)在 2.3.1 (包括)之后及 2.2.1 (包括)之前

^[27] 在不久的将来我们或许会支持。

事件

事件(`org.zkoss.zk.ui.event.Event`)用来通知服务器发生了什么。每种类型的事件都由一种不同的类来表示。例如，

`org.zkoss.zk.ui.event.MouseEvent` 来代表鼠标活动，如点击。

为了响应事件，服务器需要为其注册一个或多个事件监听器。由两种方法来注册一个事件监听器。一种是通过在标记语言中指定 `onXxx` 属性。另一种方法是为你要监听的组件或页面调用 `addEventListener` 方法。

除了在浏览器端由客户活动触发的事件，一个应用程序可以

`org.zkoss.zk.ui.event.Events` 类中的 `sendEvent` 和 `postEvent` 和 `echoEvent` 方法来触发事件。

桌面和事件处理

如上所述，桌面是页面的集合，这些页面服务于同样的 URL 请求。一个桌面当然是事件监听器能读取的范围。

当一个事件被触发时，它就和桌面联系在一起。**ZK** 分离基于关联桌面及流水事件(**pipelines events**)分成单独的队列。因此，同一桌面的事件可以被顺序处理。另一方面，不同桌面的事件可以被并行处理。

一个事件监听器是被允许读取事件关联桌面内任何页面的任何组件的。它也被允许将一个组件从一个页面移到另一个页面，只要这些页面在同一桌面内。另一方面，它不能读取到其它桌面的组件。

[注]: 开发人员可以在一个事件监听器中将一个组件从一个桌面卸载，然后在另外的事件监听器中将其添加到另外一个桌面。

桌面及创建组件

当一个组件在一个事件监听器中被创建时，它就自动被分配到被处理的事件相关联的桌面。即使组件不属于一个页面这种分配也会发生。这就意味着你在事件监听器中创建的任何组件可以用于监听正在处理的同一桌面。

如果一个组件是在一个线程(**thread**)而不是任何事件监听器中创建的话，它就不属于任何桌面。在这种情况下，可以将它添加到任何一个桌面，只要添加发生在一个合适的监听程序中。当然，一旦组件被添加到一个桌面，它就一直属于这个桌面。

对于大多数应用程序而言，很少在线程(**thread**)而不是事件监听器中创建组件。然而，如果有一个长操作(**long operation**)，你或许会在后台线程(**background thread**)中

执行它。那么，你可以在后台准备一些组件树，然后在合适的事件被接收时将它们添加到桌面。关于此的详细信息，请参考事件监听及处理一章中长操作一节。

ZUML 和XML命名空间

ZK用户界面标记语言(ZK User Interface Markup Language, ZUML)是一套基于XML的语言，开发人员可以用它来描述视觉表现。ZK的目标是分离出一套独立的组件一供使用。换句话说，一套不同的组件^[28]，就像XUL和XHTML，可以同时在同一ZUML页面内使用。不同的标记语言可以被透明地添加。如果两套或更多的组件在同一页面内被使用，开发人员必须使用XML命名空间来区分它们。请参考ZK用户界面标记语言一章中组件集及XML命名空间一节，如果你想在同一页面中使用多套组件，就像XUL和XHTML。

[提示]: 在 ZUML 中使用 XML 命名空间是可选的，你只有在使用多套组件时会用到它。

^[28] 亦称为标签(tags)。 在组件与标签之间有一对一的映射。

第 4 章 组件活动周期

目录

[加载页面的活动周期](#)

[页面初始阶段](#)

[组件创建阶段](#)

[事件处理阶段](#)

[响应阶段](#)

[更新页面的活动周期](#)

[请求处理阶段](#)

[事件处理阶段](#)

[响应阶段](#)

[模型\(The Molds\)](#)

[组件垃圾回收](#)

本章描述了加载页面和更新页面的活动周期。

加载页面的活动周期

ZK 加载器(ZK loader)加载并解释页面需要经历四个阶段：页面初始阶段，组件创建阶段，事件处理阶段及响应阶段(the Page Initial Phase, the Component Creation Phase, the Event Processing Phase, and the Rendering Phase)。

如果你手动创建所有组件(使用richlet ^[29]),则没有页面初始阶段。

页面初始阶段

在这个阶段，ZK 处理处理指令，被称为初始化(init)。如果并没有定义这样的处理指令，此阶段会被跳过。

对于每个 init 处理指令都有一个 class 属性，一个指定类的实例(instance)将会被创建，然后它的 doInit 方法将会被调用。当然，这个类要做什么取决于你的应用程序的需求。

```
<?init class="MyInit"?>
```

初始处理指令的另一种形式是使用 zscript 属性指定包含脚本代码的文件如下。那么在页面初始阶段这个文件将会被解释。

```
<?init zscript="/my/init.zs"?>
```

请注意在这个阶段页面并没有被附加到桌面。

组件创建阶段

在这个阶段，ZK 加载器(ZK loader)解释一个 ZUML 页面，它创建并初始组件。这需要以下的一些步骤：

1. 对于每个元素，它检查 if 和 unless 以确定元素是否有效。如果无效，此元素及其所有的子元素将会被忽略。
2. 如果 forEach 被指定并伴随着一个项目的集合，ZK 将会为集合中的每个项目重复以下步骤。
3. 基于元素名字，或使用 use 属性指定的类(如果有的话)创建一个组件。
4. 基于在 ZUML 页面属性指定的顺序依次初始成员。
5. 解释嵌套的元素(nested elements)并重复整个过程。
6. 调用afterCompose方法如果组件实现了 org.zkoss.zk.ui.ext.AfterCompose ^[30] 接口。
7. 在所有的组件都被创建后，onCreate 事件被送到该组件，这样之后应用程序可以初始划化一些元素的内容。注意，onCreate 事件首先为子组件公布。

[注]: 开发人员可以通过监听 onCreate 事件或实现 AfterCompose 接口来完成一些特定应用程序的初始化。AfterCompose 在组件创建阶段(the Component Creation Phase)被调用, 而 onCreate 事件是由事件监听器来处理的。一个事件监听器者可以自由地挂起或恢复执行(例如创建对话框(modal dialogs)), 而由于 AfterCompose 不需要派生另一个线程, 所以它快一些。

事件处理阶段

在这个阶段, ZK 依次调用每个事件的监听器, 这些事件已经为桌面排好队列。一个独立线程开始调用监听, 这样它可以在不影响其它事件处理的情况下被挂起。

在处理过程中, 一个事件可能引发其它事件, 事件监听和处理(The Event Listening and Processing)一章来获得更多细节。

响应阶段

在所有的事件都被处理后, ZK 将这些组件组成一个规则的 HTML 页面并将这个页面送到浏览器。

为了发送一个组件, redraw 会被调用。在这个方法中, 一个组件的实现(implementation)并不会更改其它组件的内容。

^[30] step 3-5 即所谓的composing。这是此方法称为 AfterCompose的原因。

更新页面的活动周期

ZK AU 引擎处理从客户端来的 ZK 请求需要三个阶段: 请求处理阶段, 事件处理阶段及响应阶段。

ZK AU 引擎将 ZK 请求传递到队列(每个桌面一个队列)。因此, 来自相同桌面的请求可以被顺序处理, 而来自不同桌面的请求可以被并行处理。

请求处理阶段

依赖于请求, ZK AU 引擎可能会更新被影响组件的内容, 这样它们的内容就和在客户端展示的一样。

然后, 它将相应的事件提交到队列。

事件处理阶段

这个阶段和组件创建阶段中的事件处理阶段是类似的。它在独立的线程中依次处理事件。

响应阶段

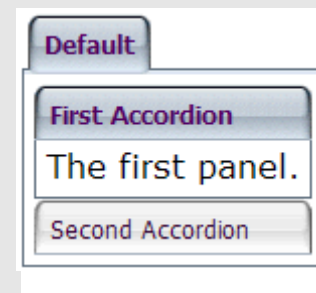
在所有的事件都被处理后，ZK 送出受影响的组件，产生相应的 ZK 响应，并将这些请求送回到客户端。然后，客户端引擎会基于这些响应更新浏览器端的 DOM 树。

是否重设一个组件的整个视觉表现或更新浏览器端的一个属性完全取决于组件的实现。平衡互动性与简易性，这是开发人员的工作。

模型(The Molds)

一个组件可以有不同的外观，甚至在同一页面内，这个概念被称为模型(aka., **template**)。通过使用 `setMold` 方法，开发人员可以动态的改变组件界面的模型。所有的组件都支持的模型(**mold**)为 `default`，即默认值。一些组件可以支持两种或更多的模型。例如，`tabbox` 同时支持 `default` 和 `accordion` 模型。

```
<tabbox><!-- if not specified, the default mold is assumed. -->
  <tabs>
    <tab label="Default"/>
  </tabs>
  <tabpanels>
    <tabpanel>
      <tabbox mold="accordion">
        <tabs>
          <tab label="First Accordion"/>
          <tab label="Second Accordion"/>
        </tabs>
        <tabpanels>
          <tabpanel>The first panel.</tabpanel>
          <tabpanel>The second panel.</tabpanel>
        </tabpanels>
      </tabbox>
    </tabpanel>
  </tabpanels>
</tabbox>
```



组件垃圾回收

不同于其它基于组件的图形用户界面(GUI)， 组件类并没有 `destroy` 或 `close` 方法。就像 W3C 的 DOM，当一个组件从页面被卸载时它就从浏览器被移除。它的表现就像附着在页面上一样。

更确切地说，一旦一个组件被卸载，如果应用程序不再涉及到它，它就不再受 ZK 的管理。它所占用的内存会被 Java 虚拟机的垃圾回收机制(JVM's garbage collector)所释放。

第 5 章 事件监听及处理

目录

[通过标记语言添加事件监听器](#)

[通过程序添加或移除事件监听器](#)

[声明一个成员](#)

[动态地添加与移除事件监听器](#)

[延期事件监听器](#)

[为页面动态地添加和移除事件监听器](#)

[调用顺序](#)

[中止调用序列](#)

[事件监听器提交,发送和回显事件](#)

[提交事件](#)

[发送事件](#)

[回显事件](#)

[线程\(Thread\)模型](#)

[挂起及恢复](#)

[长操作\(Long Operations\)](#)

[初始与清理事件处理线程](#)

[处理每个事件前的初始化](#)

[处理完每个事件后清理](#)

本章描述事件是如何被处理的。

通过标记语言添加事件监听器

添加一个事件监听器最简单的方法就是在一个 ZUML 页面内声明一个属性。用来监听的属性的值是可以被 BeanShell 解释的任何 Java 代码。

```
<window title="Hello" border="normal">
```

```
<button label="Say Hello" onClick="alert(&quot;Hello
World!&quot;);"/>
</window>
```

通过程序添加或移除事件监听器

有两种方法通过程序添加或移除事件监听器。

声明一个成员

当用你自己的类重定义(overriding)一个组件后,你可以声明一个成员函数成为事件监听器。

在一个ZUML页面中,你可以使用use属性来指定你想使用的类,即用它去替换默认类。如下所示,它使用了MyWindow来替换默认的org.zkoss.zul.Window [\[31\]](#)。

```
<window use="MyWindow">
...
</window>
```

然后你得实现 MyWindow.java 同过继承默认类,就像下面一样:

```
public class MyWindow extends org.zkoss.zul.Window {
    public void onOK() { //add an event listener
        ...//handles the onOK event (sent when ENTER is pressed)
    }
}
```

如果你想获得关于事件的更多信息,你可以按如下方式声明:

```
public void onOK(org.zkoss.zk.ui.event.KeyEvent event) {
...
}
```

或

```
public void onOK(org.zkoss.zk.ui.event.Event event) {
...
}
```

不同的事件或许与不同的事件对象相关联。参考附录 C(Append C)来获取更多的细节。

动态地添加与移除事件监听器

开发人员可以使用 `org.zkoss.zk.ui.Component` 接口中的 `addEventListener` 和 `removeEventListener` 方法来动态地添加或移除事件监听器。如下所示，动态添加的事件监听器必须实现 `org.zkoss.zk.ui.event.EventListener` 接口。

```
void init(Component comp) {
    ...
    comp.addEventListener("onClick", new MyListener());
    ...
}
class MyListener implements org.zkoss.zk.ui.event.EventListener
{
    public void onEvent(Event event) throws UiException {
        ...//processing the event
    }
}
```

延期事件监听器

默认情况下，当客户端的事件被触发时就会被送到服务器。但是，许多事件仅用于维持服务器端的现状，而不是向客户端提供视觉响应(**visual response**)。换句话说，这些监听器的事件并不需要马上被送出。相反，它们应该仅被提交一次，以降低客户端和服务器的来往，以提高服务器的性能。为求描述方便，我们称它们为延期事件监听器(**Deferrable Event Listeners**)。

为了使一个事件监听延期，必须实现 `org.zkoss.zk.ui.event.Deferrable` 接口(和 `EventListener`)并且使用 `isDeferrable` 方法返回 `true`，就像下面一样。

```
public class DeferrableListener implements EventListener,
Deferrable {
    private boolean _modified;
    public void onEvent(Event event) {
        _modified = true;
    }
    public boolean isDeferrable() {
        return true;
    }
}
```

当客户端的一个事件(例如，选择一个列表项目(list item))被触发时，如果没有为其注册事件监听器或仅有延期(deferrable)的监听器被注册，ZK 不会将事件送出。

一方面，如果至少有一个非延期(non-deferrable)监听器，事件会被马上送到服务器端，和所有的队列事件(queued events)一起。没有事件会丢失，到达顺序是保存好的。

[提示]: 当有非延期监听器为用户提供视觉响应，可以使用使用延期的(deferrable)事件监听器维持(maintaining)服务器状态。

为页面动态地添加和移除事件监听器

开发人员可以为页面(org.zkoss.zk.ui.Page)动态地添加和移除事件监听器。一旦被添加，所有被指定名字的事件会被送到指定页面的任何组件，这些页面将会被送到监听器。

所有的页面级(page-level)事件监听器都是非即时。换言之，isArap 方法被忽略了。

一个典型的例子是使用页面级事件监听器来维护修改标志(modification flag)，如下：

```
public class ModificationListener implements EventListener,
Deferrable {
    private final Window _owner;
    private final Page _page;
    private boolean _modified;

    public ModificationListener(Window owner) {
        //Note: we have to remember the page because unregister might
        //be called after the owner is detached
        _owner = owner;
        _page = owner.getPage();
        _page.addEventListener("onChange", this);
        _page.addEventListener("onSelect", this);
        _page.addEventListener("onCheck", this);
    }
    /** Called to unregister the event listener.
     */
    public void unregister() {
        _page.removeEventListener("onChange", this);
        _page.removeEventListener("onSelect", this);
        _page.removeEventListener("onCheck", this);
    }
    /** Returns whether the modified flag is set.
     */
    public boolean isModified() {
        return _modified;
    }
}
```



```
//-- EventListener --//
public void onEvent(Event event) throws UiException {
    _modified = true;
}
//-- Deferrable --//
public boolean isDeferrable() {
    return true;
}
}
```

[注]: 在事例中是否实现 `Deferrable` 接口是可选的, 因为页面的事件监听器总是假定为延期的, 与是否实现 `Deferrable` 接口无关。

调用顺序

调用事件监听器的顺序如下。假定接收的是 `onClick` 事件。

1. 如果监听器实现了 `org.zkoss.zk.ui.event.Express` 接口, 依次为添加到目标组件(**targeting component**)的 `onClick` 事件调用事件监听器。按照添加的顺序调用。
2. 调用目标组件的 `onClick` 属性指定的脚本语言。
3. 如果监听器没有实现 `org.zkoss.zk.ui.event.Express` 接口, 依次为添加到目标组件的 `onClick` 事件调用事件监听器。按照添加的顺序调用。
4. 调用目标组件的 `onClick` 成员方法。
5. 依次为添加到目标组件所属页面的 `onClick` 事件调用事件监听器。按照添加的顺序调用。

`org.zkoss.zk.ui.event.Express` 接口是一个装饰器(**decorative interface**), 用来改变调用事件监听器的优先级。注意, 如果事件监听器被添加到页面, 而不是组件, 这个接口是没有意义的。

中止调用序列

你可以通过调用 `org.zkoss.zk.ui.event.Event` 类中的 `stopPropagation` 方法来中止调用序列。一旦事件监听器调用了这个方法, 之后所有的事件监听器将会被忽略。

^[31] 默认的定义在 `in zul.jar` 中的 `lang.xml`。

事件监听器提交,发送和回显事件

除了接收事件，一个应用程序可以在事件监听器间通信通过向他们发送和提交事件。

提交事件

通过使用 `org.zkoss.zk.ui.event.Events` 类中的 `postEvent` 方法，应用程序可以将事件提交至事件队列末尾。将事件放到队列中后立即返回。当这个事件之前的所有事件都被处理完毕后，该事件就会被处理。

发送事件

通过使用 `org.zkoss.zk.ui.event.Events` 类中的 `sendEvent` 方法，应用程序可以请求 ZK 立即处理指定事件。当指定事件的事件监听器都被处理完毕后才返回。事件是在相同的线程内被处理的。

回显事件

通过使用 `org.zkoss.zk.ui.event.Events` 类的 `echoEvent` 方法，应用程序可以请求客户端回显稍后要处理的事件。在排列请求客户端回显事件的响应后，该事件会立即返回。

注意，不同于 `sendEvent` 和 `postEvent`，事件并不会在当前执行过程中被处理，而是在客户端回显事件之后。换句话说，在客户端已更新用户界面之后，事件才会被处理。因此，对于在开始长操作之前提示用户，这是很有用的。

例如，你可以打开一个标示(**highlighted**)窗口，然后在客户端显示窗口(且回显事件)之后调用 `echoEvent` 进行长操作。

例如，在下面的例子中，我们可以使用 `org.zkoss.zk.ui.util.Clients.showBusy` 方法来显示繁忙信息，这样用户就知道系统繁忙。然后用户将会看到 "Execute..."，两秒钟之后则是 "Done." 如果我们使用 `postEvent`，最后两秒钟之后用户将会同时看到 "Execute..." 和 "Done."。

```
<window id="w" title="Test echoEvent">
  <attribute name="onLater">
    Thread.sleep(2000);
    Clients.showBusy(null, false);
    new Label("Done.").setParent(w);
  </attribute>
```

```
<button label="echo">
<attribute name="onClick">
  Clients.showBusy("Execute...", true);
  Events.echoEvent("onLater", w, null);
</attribute>
</button>
</window>
```

线程(Thread)模型

对于每个桌面，事件总是被顺序处理的，所以线程模型是很简单的。就像开发桌面应用程序一样，你不需要担心 **racing** 和多线程(**multi-threading**)。所有你需要做的就是注册一个事件监听器且事件被调用时处理它。

[提示]: 当一个 **ZUML** 页面在 **servlet** 线程中被赋值时，每一个事件监听器都在一个被称为事件处理线程的独立线程中运行。

[提示]: 事件处理线程的使用可以被禁止，这样所有的线程都可以在 **Servlet** 线程中被处理。这样有更好一点的表现并且减少集成的问题。但是，你不能中止执行。参考性能提示一章中使用 **Servlet** 线程处理事件一节。

挂起及恢复

为了高级的应用，你或许必须挂起一个执行直到一些条件被满足，**org.zkoss.zk.ui.Executions** 类中的 **wait**, **notify** 和 **notifyAll** 方法都是为这样的目的而设计的。

当一个事件监听器想挂起自己，它可以调用 **wait**。如果申请的具体条件得到满足，另一个线程可以通过 **notify** 和 **notifyAll** 方法唤醒它。对话框是使用这种机制的典型例子。

```
public void doModal() throws InterruptedException {
...
    Executions.wait(_mutex); //suspend this thread, an event
processing thread
}
public void endModal() {
...
    Executions.notify(_mutex); //resume the suspended event
processing thread
}
```

它们的使用类似于 `java.lang.Object` 中的 `wait`, `notify` 和 `notifyAll` 方法。但是, 你不能使用 `java.lang.Object` 中的方法来挂起和恢复事件监听器。否则, 关联到这个桌面的所有的事件处理都会停滞。

请注意, 不同于 Java 的 `Object` 类的 `wait` 和 `notify` 方法, 是否使用同步的 `synchronized block` 来包 `Executions` 的 `wait` 和 `notify` 是可选的。在上述情况下中, 我们并不需要这样做, 因为没有可能的 `racing` 问题。但是, 如果存在这样的 `racing` 问题, 你可以使用 `synchronized block`, 就像在 Java `Object` 的 `wait` 和 `notify` 中使用那样。

```
//Thread 1
public void request() {
    ...
    synchronized (mutex) {
        ...//start another thread
        Executions.wait(mutex); //wait for its completion
    }
}

//Thread 2
public void process() {
    ... //process it asynchronously
    synchronized (mutex) {
        Executions.notify(mutex);
    }
}
```

长操作(Long Operations)

对于同一个桌面而言, 事件是被顺序处理的。最坏的情况下, 由于 `HTTP` 的限制, 用户仅会看到显示在浏览器左上方的一个小处理对话框而没有其他任何提示。

通过使用上面回显事件章节描述的回显事件和 `showBusy` 方法, 则可以提供更多描述性信息来提示用户接下来的行为, 例如, 点击其他按钮为进一步进行长操作降低性能。

但是, 阻止来自用户的访问对你的应用程序而言也许是不可能的。为了防止阻塞, 你必须在另一个线程中处理长操作, 就像桌面应用程序那样。然后, 持续将处理状态送回客户端。

使用 `ZK`, 你有四种选择: 服务器推动(server push), 挂起和恢复(suspend and resume), `timer`, 和捎带(piggyback)。

选择 1：服务器推动

服务器推动即所谓的反向 Ajax(reverse-Ajax), 允许服务器将内容动态的发至客户端。通过使用服务器推动技术, 当你预先定义的条件满足时, 则可以在工作线程内将内容发至客户端或更新客户端的内容。使用服务器推动很简单, 仅需要如下的三步,

1. 使用 `Desktop.enableServerPush(boolean bool)` 为桌面调用启用服务器推动。
2. 将需要更新的组件传递至工作线程。
3. 在桌面内调用工作线程。

[注]: 你需要安装 `zkex.jar` 或 `zkmax.jar` 来使用服务器推动, 除非你有自己 `org.zkoss.zk.ui.sys.ServerPush` 的实现。

现在让我们来看一个实际的例子。若你想使用服务器推动更新客户端的数字, 首先要为桌面启用服务器推动, 然后调用线程, 如下。

```
<window title="Demo of Server Push">
<zscript>
    import test.WorkingThread;
    WorkingThread thread;
    void startServerpush() {
        //enable server push
        desktop.enableServerPush(true);
        //invoke working thread and passing required component as
parameter
        thread= new WorkingThread(info);
        thread.start();
    }
    void stopServerpush() {
        //stop the thread
        thread.setDone();
        //disable server push
        desktop.enableServerPush(false);
    }
</zscript>
    <vbox>
        <button label="Start Server Push"
onClick="startServerpush()" />
        <button label="Stop Server Push"
onClick="stopServerpush()" />
        <label id="info"/>
    </vbox>
</window>
```

安全问题

需要注意的一件事是同步问题，即当多个线程访问同一桌面时发生的问题。因此，在访问桌面前，你必须调用 `Executions.activate(Desktop desktop)` 来获取对桌面的完全控制权，以避免这个问题，在线程完成它的工作后调用 `Executions.deactivate(Desktop desktop)` 释放对桌面的控制，如下，

```
package      test;

public class  WorkingThread extends Thread{
    private final Desktop _desktop;
    private final Label _info;
    private int _cnt;
    private boolean _ceased;
    public      WorkingThread(Label info){
        _desktop = info.getDesktop();
        _info    = info;
    }
    public      void    run(){
        try      {
            while (!_ceased){
                Threads.sleep(2000); //Update each two seconds
                Executions.activate(_desktop); //get full control of
desktop
                try {
                    _info.setValue(Integer.toString(++_cnt));
                }catch      (RuntimeException ex)      {
                    throw ex;
                }catch      (Error ex){
                    throw ex;
                }finally{
                    Executions.deactivate(_desktop); //release full
control of      desktop
                }
            }
        } catch (InterruptedException ex){
        }
    }
    public      void    setDone(){
        _ceased = true;
    }
}
```

幕后

服务器推动机制是使用客户端轮询(client-polling)技术实现的，即客户端将会反复询问服务器以调用工作线程完成其工作，询问的频率可以调用 `Executions.setDelay(int min, int max, int factor)` 手动调整。

1. `min`，为任何将要到来的服务器推动进程获得(poll)服务器的最小延迟。
2. `max`，为任何将要到来的服务器推动进程获得(poll)服务器的最大延迟。
3. `factor`，实际的延迟为多种延迟因素复合的处理时间。

最后，需要注意的一件事是频率会依赖服务器的加载而自动调整。

选择 2：线程挂起和恢复

使用服务器推动技术，你不需要关心多线程的问题。但是，若你想自己处理这个工作，由于 HTTP 的限制必须遵守如下的规则。

1. 创建工作线程后，使用 `org.zkoss.zk.ui.Executions` 类的 `wait` 方法来挂起事件处理程序本身。
2. 由于工作线程不是一个事件监听器，所以它不能读取任何组件，除非这个组件不属于任何桌面。因此在开始工作线程之前你可能需要手动添加(pass)一些必要信息。
3. 然后，若有必要的话，工作线程可以取出信息，并且创建组件。只是不引用属于任何桌面的任何组件。
4. 工作线程完成之后，在工作线程中使用 `org.zkoss.zk.ui.Executions` 类中的 `notify(Desktop desktop, Object flag)` 或

`notifyAll(Desktop desktop, Object flag)` 方法来恢复事件处理程序。

5. 直到另一个事件从客户端被发送过来，恢复的事件处理程序才会执行。为了强制发送一个组件，你可以使用 `timer` 组件(`org.zkoss.zul.Timer`)触发事件片刻之后或定期的。这个 `timer` 的事件监听器可以不做任何事情或更改进展状况。

事例：一个异步产生标签的工作线程

假定我们以异步的方式创建标签。当然，用多线程来做这么小的事是没有意义的，但是我们可以用更复杂的(sophisticated)任务来代替这个。

```
//WorkingThread
package test;
```

```

public class WorkingThread extends Thread {
    private static int _cnt;

    private Desktop _desktop;
    private Label _label;
    private final Object _mutex = new Integer(0);

    /** Called by thread.zul to create a label asynchronously.
     * To create a label, it start a thread, and wait for its
    completion.
     */
    public static final Label asyncCreate(Desktop desktop)
    throws InterruptedException {
        final WorkingThread worker = new WorkingThread(desktop);
        synchronized (worker._mutex) { //to avoid racing
            worker.start();
            Executions.wait(worker._mutex);
            return worker._label;
        }
    }

    public WorkingThread(Desktop desktop) {
        _desktop = desktop;
    }

    public void run() {
        _label = new Label("Execute "+ ++_cnt);
        synchronized (_mutex) { //to avoid racing
            Executions.notify(_desktop, _mutex);
        }
    }
}

```

然后,在一个事件监听器中,我们使用 ZUML 页面来调用这个工作线程,如在 onClick 事件中。

```

<window id="main" title="Working Thread">
    <button label="Start Working Thread">
        <attribute name="onClick">
            timer.start();
            Label label = test.WorkingThread.asyncCreate(desktop);
            main.appendChild(label);
            timer.stop()
        </attribute>
    </button>
    <timer id="timer" running="false" delay="1000"
    repeats="true"/>

```



```
</window>
```

注意到我们必须使用 `timer` 来真正地恢复被挂起的事件监听器(`onClick`)。这看起来是多余的，但是由于 HTTP 的限制：为了保持 Web 页面在浏览器端的活跃，当事件处理程序被挂起时我们必须返回响应。然后，工作线程完成了工作并唤醒了事件监听器，HTTP 请求已经不在了。因此，我们需要一种方式来”捎带(`piggyback`)”这个结果，而这就使 `timer` 为什么会被使用的原因。

更确切地说，当工作线程唤醒一个事件监听器时，ZK 只是把它加到一个等待列表。当另一个 HTTP 请求到达时，监听器才真正恢复(如上面例子中的 `onTimer` 事件)。

在这个简单的事例中，我们并没有为 `onTimer` 事件作任何事情。对于一个复杂的应用程序，你可以用它来返回处理状态。

选择 3: Timer(没有挂起/恢复)

无需挂起和恢复来实现一长操作(`long operation`)是由可能的。这在同步代码(`synchronization codes`)对于调试来说太复杂的情况下是很有用的。

注意是很简单的。工作线程将结果保存在一个临时空间，然后使用 `onTimer` 事件将结果弹到(`pops`)桌面。

```
//WorkingThread2
package test;
public class WorkingThread2 extends Thread {
    private static int _cnt;

    private final Desktop _desktop;
    private final List _result;

    public WorkingThread2(Desktop desktop, List result) {
        _desktop = desktop;
        _result = result;
    }
    public void run() {
        _result.add(new Label("Execute "+ ++_cnt));
    }
}
```

然后，在 `onTimer` 事件监听器上附加标签。

```
<window id="main" title="Working Thread2">
    <zscript>
        int numPending = 0;
        List result = Collections.synchronizedList(new LinkedList());
```

```

</zscript>
<button label="Start Working Thread">
  <attribute name="onClick">
    ++numPending;    timer.start();
    new test.WorkingThread2(desktop, result).start();
  </attribute>
</button>
<timer id="timer" running="false" delay="1000" repeats="true">
  <attribute name="onTimer">
    while (!result.isEmpty()) {
      main.appendChild(result.remove(0));
      --numPending;
    }
    if (numPending == 0) timer.stop();
  </attribute>
</timer>
</window>

```

选择 4: 捎带(piggyback)(没有挂起/恢复, 没有Timer)

当用户, 例如, 点击一个按钮或输入一些东西时, 你可以将结果捎带(piggyback)到客户端, 而不必循环地检查它们。

为了完成捎带(piggyback), 你需要为一个根组件注册一个 onPiggyback 事件监听器。然后每次 ZK 更新引擎(ZK Update Engine)处理事件时, 这个监听器将会被调用。例如, 你可以按如下的方式重写代码。

```

<window id="main" title="Working Thread3"
onPiggyback="checkResult()">
  <zscript>
    List result = Collections.synchronizedList(new LinkedList());

    void checkResult() {
      while (!result.isEmpty())
        main.appendChild(result.remove(0));
    }
  </zscript>
  <button label="Start Working Thread">
    <attribute name="onClick">
      timer.start();
      new test.WorkingThread2(desktop, result).start();
    </attribute>
  </button>
</window>

```

捎带(piggyback)方式的优点是客户端与服务器端没有额外的往来。但是, 如果用户没有任何活动(如点击或打字)的话是无法看到更新的。这种方式是否合适取决于应用程序的要求。

[注]: 一个延期的事件(deferrable)不会马上被送到客户端, 所以, 只有一个非延期的事件被触发后 onPiggyback 事件才会被触发。请参考关于延期事件监听器一节获得详细信息。

初始与清理事件处理线程

处理每个事件前的初始化

一个事件监听器是在一个事件处理线程中执行的。有时, 你必须在处理所有事件前初始该线程。

一个典型的例子是初始化认证所使用的线程。一些 J2EE 或 Web 容器将认证信息存储在局部存储器(local storage)线程中。这样, 可以在需要的时候自动进行重复验证。

为了初始化事件处理线, 必须在WEBINF/zk.xml文件^[32]的listener元素中注册一个是实现了org.zkoss.zk.ui.event.EventThreadInit接口的类。

一旦注册完毕, 一个指定类的实例就会在主线程中被创建。然后, 在处理其他事情之前, 该实例的 init 方法就会在事件处理线程的上下文中被调用。

请注意构造程序(constructor)和 init 方法是在不同的线程中被调用的, 因此开发人员可以从一个线程取得独立线程数据并传送给另外一个线程。

这里有一个 JBoss^[33]的认证机制。在这个例子中, 我们在构造程序中获取存储在Servlet线程中的信息。然后当init方法被调用时初始事件处理线程。

```
import java.security.Principal;
import org.jboss.security.SecurityAssociation;
import org.zkoss.zk.ui.Component;
import org.zkoss.zk.ui.event.Event;
import org.zkoss.zk.ui.event.EventThreadInit;

public class JBossEventThreadInit implements EventThreadInit {
    private final Principal _principal;
    private final Object _credential;
    /** Retrieve info at the constructor, which runs at the servlet
    thread. */
    public JBossEventThreadInit() {
        _principal = SecurityAssociation.getPrincipal();
    }
}
```

```

        _credential = SecurityAssociation.getCredential();
    }
    //-- EventThreadInit --//
    /** Initial the event processing thread at this method. */
    public void init(Component comp, Event evt) {
        SecurityAssociation.setPrincipal(_principal);
        SecurityAssociation.setCredential(_credential);
    }
}

```

然后，在 WEB-INF/zk.xml 文件中，你需要完成如下的配置：

```

<zk>
  <listener>
    <listener-class>JBossEventThreadInit</listener-class>
  </listener>
</zk>

```

处理完每个事件后清理

类似的，在处理完每个事件后你或许必须清理一个事件处理线程。

一个典型的例子是关闭事务处理，如果它没有被适时地关闭。

为了清理一个事件处理线程，你必须在 WEB-INF/zk.xml 文件的 listener 元素中注册一个实现 org.zkoss.zk.ui.event.EventThreadCleanup 接口的监听类。

```

<zk>
  <listener>
    <listener-class>my.MyEventThreadCleanup</listener-class>
  </listener>
</zk>

```

^[32] the Developer's Reference 的附录(Appendix B) 进行了详细描述。

^[33] <http://www.jboss.org>

第 6 章 ZK用户界面标记语言

目录

[XML](#)

[元素必须格式良好](#)

[特殊字符必须被替换](#)

[属性值必须被指定且用引号包围](#)

[注释](#)

[字符编码](#)

[命名空间](#)

[条件式流程](#)

[If 和Unless](#)

[Switch和Case](#)

[Choose 和 When](#)

[反复式流程](#)

[each变量](#)

[forEachStatus变量](#)

[如何在事件监听器中使用 each和forEachStatus 变量](#)

[随机存取\(Load on Demand\)](#)

[使用fulfill属性的随机存取](#)

[使用事件监听器的随机存取](#)

[隐含对象](#)

[隐含对象列表](#)

[关于Request和Execution的信息](#)

[进程指令](#)

[page指令](#)

[component指令](#)

[init指令](#)

[variable-resolver指令](#)

[import指令](#)

[link和meta指令](#)

[ZK属性](#)

[apply属性](#)

[use属性](#)

[if属性](#)

[unless属性](#)

[forEach属性](#)

[forEachBegin属性](#)

[forEachEnd属性](#)

[fulfill属性](#)

[forward 属性](#)

[ZK元素](#)

[zk 元素](#)

[zscript元素](#)

[attribute元素](#)

[variables元素](#)

[custom-attributes元素](#)

[组件集及XML命名空间](#)

[标准的命名空间](#)

ZK 用户界面标记语言

ZK 用户界面标记语言(ZUML)是基于 XML 的。每一个 XML 元素描述了要创建的组件。一个 XML 属性描述了被创建组件的初始值。一个 XML 处理指令(processing instruction)如何处理整个页面，如页面的标题。

不同的组件集通过XML命名空间来区分。例如，XUL的命名空间为 <http://www.zkoss.org/2005/zul>^[34]，而XHTML的命名空间为 <http://www.w3.org/1999/xhtml>。

XML

这一章节提供了和ZK一起工作的XML的最基本概念，如果你很熟悉XML，可以跳过这一章节。如果你想学习更多，在网络上有很多资源，如 http://www.w3schools.com/xml/xml_what_is.asp 和 <http://www.xml.com/pub/a/98/10/guide0.html>。

XML 是一种标记语言，就像 HTML，但是有更严格和简洁(cleaner)的语法，有几点需要特别注意。

元素必须格式良好

首先，每个元素必须关闭。有两种元素来关闭一元素，就像下面描述的一样，它们是等价的。

描述	代码
通过一个结束标签关闭:	<code><window><window/></code>
不通过一个结束标签关闭:	<code><window/></code>

其次，元素要被正确的嵌套(nested)。

结果	代码
正确:	<pre><window> <groupbox> Hello World! </groupbox> </window></pre>
错误:	<pre><window> <groupbox></pre>

结果	代码
	Hello World! </window> </groupbox>

特殊字符必须被替换

XML 使用<element-name>来表示一个元素，所以你必须替换特殊的字符。

例如，你必须使用<表示<。

特殊字符	替换字符
<	<
>	>
&	&
"	"
'	'

另外，你可以使用 CDATA 来使 XML 解析器不要解释其内的文本，如下：

```
<zscript>
<![CDATA[
void myfunc(int a, int b) {
    if (a < 0 && b > 0) {
        //do something
    }
}]>
</zscript>
```

有意思的是反斜杠(\)不是特殊字符，所以你不需要担心。

属性值必须被指定且用引号包围

结果	代码
正确:	width="100%" checked="true"
错误:	width=100% checked

注释

注释通常被用于留下一个说明或暂时屏蔽出一部分 XML 源码(leave a note or to temporarily edit out a portion of XML code)。

```
<window>
<!-- this is a comment and ignored by ZK -->
</window>
```

字符编码

尽管是可选定的，但在你的 XML 中指定编码是个好主意，这样 XML 解析器可以正确的解释文本。

注意：它必须被放置在文件的第一行。

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

除了指定正确的编码，同时也要确保你的 XML 编辑器支持这种编码。

命名空间

命名空间是区分 XML 文档中用到名字的一个简单易懂的办法。ZK 使用 XML 命名空间来区分组件名称。这样，只要不在同一个命名空间，两个组件有相同的名字是可以的。ZK 使用 XML 命名空间来表现一个组件集。这样，开发人员可以在同一个页面内混合使用两个或多个组件集，如下所示

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:x="http://www.zkoss.org/2005/zul"
xmlns:zk="http://www.zkoss.org/2005/zk">
<head>
<title>ZHTMLDemo</title>
</head>
<body>

<h1>ZHTML Demo</h1>
  <table>
    <tr>
      <td><x:textbox/></td>

<td><x:button label="Now" zk:onClick="addItem()" /></td>
    </tr>
```



```
</table>
<zk:zscript>
void addItem() {
}
</zk:zscript>
</body>
</html>
```

在这里

1. `xmlns:x="http://www.zkoss.org/2005/zul"` 指定一个称为 `http://www.zkoss.org/2005/zul` 的命名空间，并且使用 `x` 来呈现这个命名空间。
2. `xmlns:="http://www.w3.org/1999/xhtml"` 指定一个称为 `http://www.w3.org/1999/xhtml` 的命名空间，且使用它作为默认的命名空间。
3. `<html>` 从默认的命名空间中指定一个称为 `html` 的元素，在这个例子中也就是 `http://www.w3.org/1999/xhtml`。
4. `<x:textbox/>` 从 `http://www.zkoss.org/2005/zul` 命名空间中指定一个称为 `textbox` 的元素。

使用Schema 自动完成

许多 IDE，如 Eclipse，支持自动完成，如果 XML schema 被按如下方式指定。

```
<window xmlns="http://www.zkoss.org/2005/zul"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.zkoss.org/2005/zul
http://www.zkoss.org/2005/zul/zul.xsd">
```

除了可以从 <http://www.zkoss.org/2005/zul/zul.xsd> 下载到，可以在 ZK 的 binary 发行版中的 `dist/xsd` 目录找到 `zul.xsd`。

^[34] 被称为 <http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul>。但是，添加了许多非 XUL 特性，所以最好使用独立的命名空间。

条件式流程

If 和Unless

创建一个元素的流程可以条件式的。通过只指定 `if` 或 `unless` 属性，或都指定，开发人员可以控制是否创建相关的元素。

在下面的例子中，如果 `a` 为 1，且 `b` 不为 2 `window` 组件就会被创建。如果一个元素被忽略，那么它所有的组件都会忽略。

```
<window if="{a==1}" unless="{b==2}">
    ...
</window>
```

下面的例子控制什么时候解释 Java 代码。

```
<textbox id="contributor"/>
<zscript if="{param.contributor}">
    contributor.label =
    Executions.getCurrent().getParameter("contributor");
</zscript>
```

Switch和Case

使用 `zk` 元素的 `switch` 和 `case` 属性，你可以控制 `ZK` 页面在一个变量等于某个特定值时才会被执行。

```
<zk switch="{fruit}">
    <zk case="apple">
        Evaluated only if {fruit} is apple
    </zk>
    <zk case="{special}">
        Evaluated only if {fruit} equals {special}
    </zk>
    <zk>
        Evaluated only if none of the above cases matches.
    </zk>
</zk>
```

`ZK` 加载器会从第一个 `case` 执行到最后一个 `case`，直到匹配 `switch` 的条件，即 `switch` 属性指定的值。语句是互相排斥的条件(The evaluation is mutually exclusive conditional.)，只有第一个匹配的 `case` 才会被执行。

zk 元素没有任何 case 匹配, 则为 default-即, 当前面所有的 case 都不匹配时则执行 default。

复合Case

你可以为 case 属性指定一个 case 列表, 这样当其中一个匹配时, 此部分的 ZK 页面会被执行。

```
<zk switch="${fruit}">
  <zk case="apple, ${special}">
    Evaluated if ${fruit} is either apple or ${special}
  </zk>
</zk>
```

正则表达式

```
<zk switch="${fruit}">
  <zk case="/ap*.e/">
    Evaluate if the regular expression, ap*.e"., matches the switch
    condition.
  </zk>
</zk>
```

和forEach一起使用

就像其他元素, 你可以和 forEach(同样 if 和 unless 也可以)属性一起使用 case, forEach 条件会首先被执行, 就像复合 case 一样。

```
<zk case="${each}" forEach="apple, orange">
```

等价于,

```
<zk case="apple, orange">
```

Choose 和 When

choose 和 when 为相互排斥条件语句提供了一种可选的方式。

```
<zk choose="">
  <zk when="${fruit == 'apple'}">
    Evaluated if the when condition is true.
  </zk>
```

```
<zk>
  Evaluated if none of above cases matches.
</zk>
</zk>
```

你不必为 **choose** 指定任何值，它只是被用来标识相互排斥条件语句的范围。

反复式流程

创建一个元素的流程可以是反复式的。通过为 **forEach** 属性一个对象集合，开发人员可以控制相关元素要被创建多少次。为了描述方便，如果一个元素被 **forEach** 属性赋值就称其为迭代元素(**iterative element**)。

在下面的例子中，列表项目被创建了三次。每个项目的 **label** 分别为"Best", "Better" 和 "God",

```
<listbox>
  <listitem label="${each}" forEach="Best, Better, God"/>
</listbox>
```

如果你有一个存放对象集合的变量，则可以直接为 **forEach** 属性指定它。例如，假如你有一个 **grades** 变量，如下。

```
grades = new String[] { "Best", "Better", "Good" };
```

然后可以使用 **forEach** 属性来迭代此变量。注意，你必须使用 **EL** 表达式来指定这个集合。

```
<listbox>
  <listitem label="${each}" forEach="${grades}"/>
</listbox>
```

迭代(**iteration**)依赖于 **forEach** 属性指定值的类型。

1. 如果是 `java.util.Collection`，就会迭代集合(**collection**)的每个元素。
2. 如果是 `java.util.Map`，就会迭代 **map** 中的每个 `Map.Entry`。
3. 如果是 `java.util.Iterator`，就会迭代迭代器(**iterator**)中的每个元素。
4. 如果是 `java.util.Enumeration`，就会迭代 **enumeration** 中的每个元素。
5. 如果是 `Object[]`, `int[]`, `short[]`, `byte[]`, `char[]`, `float[]`
或 `double[]` 被指定了，就会迭代数组(**array**)中的每个元素。
6. 如果是 `null`，什么也不会产生(被忽略)。

7. 如果被指定的不是以上类型，相关元素仅被赋值(evaluated)一次，就好像一个集合只指定了一个单独的项目。

```
<listbox>
  <listitem label="${each}" forEach="grades"/>
</listbox>
```

each变量

在迭代中，一个变量被称为 each，通过指定集合的项目被创建并且赋值。在上面的例子中，首次迭代中，each 被赋值为"Best"，然后是"Better"，最后是"Good"。

注意 each 变量在 EL 表达式和 zscript 中都是可访问的。ZK 将会保留以前定义的所有变量，并在迭代完每个元素后将其恢复。

forEachStatus变量

forEachStatus 变量是 org.zkoss.ui.util.ForEachStatus 的一个实例(instance)，用来保存(hold)当前迭代(iteration)的信息。也主要用于取得封闭元素的项目，这些元素已通过 forEach 属性赋值。

在下面的例子中，我们使用嵌套迭代元素(nested iterative elements)来产生两个 listbox。

```
<hbox>
  <zscript>
classes = new String[] {"College", "Graduate"};
grades = new Object[] {
  new String[] {"Best", "Better"}, new String[] {"A++", "A+", "A"}
};
</zscript>
<listbox width="200px" forEach="${classes}">
  <listhead>
    <listheader label="${each}"/>
  </listhead>
  <listitem label="${forEachStatus.previous.each}: ${each}"
    forEach="${grades[forEachStatus.previous.index]}"/>
</listbox>
</hbox>
```

College	Graduate
College: Best	Better: A++
College: Better	Better: A+
	Better: A

注意 `forEachStatus` 变量在 EL 表达式和 `zscript` 中都是可访问的。

如何在事件监听器中使用 `each` 和 `forEachStatus` 变量

在事件监听器中使用 `forEach` 和 `forEachStatus` 变量有点棘手(tricky)，因为它们仅在组件创建阶段(Component Creation Phase)^[35]是可用的(available)。因此，下面的例子是不正确的。当 `onClick` 监听器被调用时，`each` 变量就不可用了。

```
<window title="Countries" border="normal" width="100%">
  <zscript><![CDATA[
    String[] countries = {
      "China", "France", "Germany", "United Kindom", "United
States"};
  ]]></zscript>

  <hbox>
    <button label="${each}" forEach="${countries}"
      onClick="alert(each)"/> <!-- incorrect!! -->
  </hbox>
</window>
```

注意按钮(button)标签(label)的赋值方式是错误的，因为它们在同一个阶段-----组件创建阶段(Component Creation Phase)。

同时也要注意不能在事件监听器中使用 EL 表达式。例如，下列代码会执行失败，因为 `onClick` 的监听器并不是 Java 代码(也就是说，EL 表达式在 `zscript` 中是被忽略的)。

```
<button label="${each}" forEach="${countries}"
  onClick="alert(${each})"/> <!-- incorrect!! -->
```

一个解决方法：定制属性

解决方法是我们必须在某处存储 `each` (和 `forEachStatus`)的内容，这样当监听器执行时其内容仍是可用的。你可以将其内容存储在任何地方，但是有一个简单的方法来处理，如下：

```
<window title="Countries" border="normal" width="100%">
  <zscript><![CDATA[
    String[] countries = {
      "China", "France", "Germany", "United Kindom", "United
States"};
  ]]></zscript>
```

```

<hbox>
  <button label="{each}" forEach="{countries}"
    onClick="alert(self.getAttribute('country'))">
    <custom-attributes country="{each}"/>
  </button>
</hbox>
</window>

```

就像按钮(button)标签(label), 定制属性(custom attributes)的属性在组件创建阶段(Component Creation Phase)被赋值(evaluated), 所以在这里你可以使用 each。然后, 它被存储在一个定制属性, 这个定制属性会在组间存在的期间一直有效(直到组件被移除)。

[\[35\]](#) 参考组件活动周期 (Component Lifecycle) 一章获取细节。

随机存取(Load on Demand)

默认情况下, 当加载页面时, ZK 基于在 ZUML 页面内定义的内容依次创建组件。但是, 我们可以推迟部分组件的创建, 直到它们可见。这个特性即为随机存取(load-on-demand)。如果在初始时有许多非可见组件, 随机存取可以提高性能。

使用fulfill属性的随机存取

延迟创建子组件的最简单方式是使用 fulfill 属性。例如, 在下面的代码片断中, comboitem 组件将不会被创建, comboitem 组件接收了 onOpen 事件, 此事件可使 comboitem 变为可见。

```

<combobox fulfill="onOpen">
  <comboitem label="First Option"/>
</combobox>

```

换句话说, 如果一个 ZUML 元素使用了 fulfill 属性, 直到 fulfill 指定的事件发生时这个组件的子组件才会被处理。

如果创建子组件创建事件的目标是另一个组件, 你可以按如下描述指定目标组件的标识。

```

<button id="btn" label="show" onClick="content.visible = true"/>
<div id="content" fulfill="btn.onClick">

```

```
Any content created automaticall when btn is clicked
</div>
```

如果组件属于不同的 ID 空间，你可以在事件名称之后指定一个路径。

```
<button id="btn" label="show" onClick="content.visible = true"/>
<window id="content" fulfill="../btn.onClick">
  Any content created automaticall when btn is clicked
</window>
```

使用事件监听器的随机存取

如果你喜欢手动创建子组件或者你需要动态地修改它们，你可以监听使这些子组件变得可见的事件，然后在监听器中操纵它们。例如：

```
<combobox id="combo" onOpen="prepare()" />
<zscript><![CDATA[
  void prepare() {
    if (event.isOpen() && combo.getItemCount() == 0) {
      combo.appendItem("First Option");
    }
  }
}]></zscript>
```

隐含对象

对于嵌入到 ZUML 页面的脚本，有一套可以使开发人员更有效地访问组件的隐含对象。这些对象对于 `zscript` 元素包含的 Java 代码及事件监听器指定的属性是可用的。当然对于 EL 表达式也是可用的。

例如，`self` 是 `org.zkoss.zk.ui.Component` 的用来代表被处理组件的一个实例。在下面的例子中，在事件监听器中使用 `self` 来标识组件。

```
<button label="Try" onClick="alert(self.label)"/>
```

同样，`event` 代表当前事件监听器正在处理的事件。因此上面语句的等价程序如下：

```
<button label="Try" onClick="alert(event.target.label)"/>
```


隐含对象列表

对象名称	描述
self	<code>org.zkoss.zk.ui.Component</code> 组件本身。
spaceOwner	<code>org.zkoss.zk.ui.IdSpace</code> 组件的空间所有者，与 <code>self.spaceOwners</code> 相同。
page	<code>org.zkoss.zk.ui.Page</code> 页面，与 <code>self.page</code> 相同。
desktop	<code>org.zkoss.zk.ui.Desktop</code> 桌面，与 <code>self.desktop</code> 相同。
session	<code>org.zkoss.zk.ui.Session</code> 会话。
application	<code>org.zkoss.zk.ui.WebApp</code> Web 应用程序。
componentScope	<code>java.util.Map</code> 在组件中定义的属性的映射(map)。与 <code>org.zkoss.zk.ui.Component</code> 接口中的 <code>getAttributes</code> 方法相同。
spaceScope	<code>java.util.Map</code> 在包含此组件的命名空间内定义的属性的映射(map)。
pageScope	<code>java.util.Map</code> 定义在页面内属性的映射(map)。与 <code>org.zkoss.zk.ui.Page</code> 接口中的 <code>getAttributes</code> 方法相同。
desktopScope	<code>java.util.Map</code> 定义在桌面内属性的映射(map)。与 <code>org.zkoss.zk.ui.Desktop</code> 接口中的

对象名称	描述
	getAttributes 方法相同。
sessionScope	<p>java.util.Map</p> <p>定义在 session 内属性的映射(map)。与 org.zkoss.zk.ui.Session 接口中的 getAttributes 方法相同。</p>
applicationScope	<p>java.util.Map</p> <p>定义在 web 程序内属性的映射(map)。与 org.zkoss.zk.ui.WebApp 接口中的 getAttributes 方法相同。</p>
requestScope	<p>java.util.Map</p> <p>在 request 内定义属性的映射(map)。与 org.zkoss.zk.ui.Execution 接口中的 getAttributes 方法相同。</p>
arg	<p>java.util.Map</p> <p>arg 参数会被传送到 org.zkoss.zk.ui.Executions 类的 createComponents 方法。不能为 null。</p> <p>注意，只有为包含页面(included page)(createComponents 方法的第一个参数)创建组件时 arg 才是可用的。另一方面，包括 onCreate 事件在内的所有事件会在之后被处理。然后，如果你想访问 onCreate 事件监听器中的 arg，可以使用 org.zkoss.zk.ui.event.CreateEvent 类的 getArg 方法。</p> <p>与 self.desktop.execution.arg 相同。</p>
each	<p>java.lang.Object</p> <p>当 ZK 为每个迭代元素赋值(evaluates)时，其代表被迭代(iterated)集合的当前项目。一个迭代元素即为使用 forEach 属性的元素。</p>
forEachStatus	<p>org.zkoss.zk.ui.util.ForEachStatus</p> <p>一个迭代器(iteration)的状态。当为迭代元素赋值时，ZK 陈</p>

对象名称	描述
	列出(exposes)迭代发生时的相关信息。
event	org.zkoss.zk.ui.event.Event 或派生类 当前事件。仅对事件监听器可用。

关于Request和Execution的信息

org.zkoss.zk.ui.Execution 接口提供了关于当前执行(execution)的信息，如请求参数。为了获取当前的 execution ，你可以选择一种方法：

1. 在组件中使用 `getDesktop().getExecution()` 。
2. 如果没有涉及到(reference)组件，页面或桌面，使用 `org.zkoss.zk.ui.Executions` 类中的

`getCurrent` 方法。

进程指令

XML 处理指令描述了如何处理 ZUML 页面。这里我们列出了最常见的指令。关于这些指令的完整信息，请参考 [Developer's Reference](#)。

page指令

```
<?page [id="..."] [title="..."] [style="..."]
[language="xul/html"] zscriptLanguage="Java"]?>
```

此指令描述了页面的属性。

[注]：你可以将 page 指令放置在 XML 文档的任何地方，但是，language 属性只有当指令位于最高层次时才是有意义的，也就是说，处于根组件的层次。

component指令

```
<?component name=" myName " macroURI=" /mypath/my.zul "
[ prop1 =" value1 " ] [ prop2 =" value2 " ]... ?>

<?component name=" myName " [class=" myPackage.myClass "]
[extends=" existentName "] [moldName=" myMoldName "]
```

```
[macroURI="/ myMoldUri "] [ prop1 =" value1 " ] [ prop2 =" value2 " ] ... ?>
```

为某一页面定义新的组件。使用此指令定义的组件，仅对于使用该指令的页面是可见的。为了定义在所有组件中可以使用的组件，可是使用附加语言插件(language addon)，即一个 XML 文件，用来

定义在Web应用程序^[36]中所有页面都可使用的组件。

有两种方式：通过宏和通过类(by-macro and by-class)。

宏格式(The by-macro Format)

```
<?component name=" myName " macroURI=" /mypath/my.zul "
[inline="true| false "] [class=" myPackage.myClass "] [ prop1
=" value1 " ] [ prop2 =" value2 " ] ... ?>
```

基于 ZUML 页面定义一个组件，被称为宏组件(macro component)。换句话说，一旦新组件的一个实例被创建，就会基于指定的 ZUML 页面(macroURI 属性指定)创建子组件。为了更多的细节，参考宏组件一章。

类格式(The by-class Format)

```
<?component name=" myName " [class=" myPackage.myClass "]
[extends=" existentName "] [moldName=" myMoldName "]
[moldURI="/ myMoldUri "] [ prop1 =" value1 " ] [ prop2 =" value2 " ] ... ?>
```

如果扩展属性被指定，则基于类创建一个新的组件，即主组件(primitive component)。这个类必须实现 org.zkoss.zk.ui.Component 接口。

为了定义一个新的组件，你至少必须定义 class 属性。ZK 用这个类属性用来实例化组件的一个新实例。

除了定义一个全新的组件，你可以通过指定 extends="existentName"来重写已存在组件的属性。换句话说，如果 extends 被指定，被指定组件的定义会作为默认值被加载，然后，只有在此指令中被指定的属性会被重写。

例如，假定你想使用 MyWindow 定义一个名字为 mywindow 的新组件，而不是默认的 window(ZUML 页面中的 org.zkoss.zul.Window)。然后声明如下：

```
<?component name="mywindow" extends="window" class="MyWindow"?>
...
<mywindow>
```

```
...  
</mywindow>
```

等价于下面的代码:

```
<window use="MyWindow">  
...  
</window>
```

同样, 在下面的例子中, 将使用 **OK** 作为按钮(**button**)默认标签(**label**), 并且按钮的默认边框为蓝色,

当然, 在仅在此页面内有效。

```
<?component name="okbutton" extends="button" label="OK"  
style="border:1px solid blue"?>
```

注意, 新组件的名字可以和已存在组件的名称一样。在这个例子中, 所有组件指定类型的实例将会使用指定的初始属性, 就好像它隐藏了存在的定义。例如, 下列代码会使所有的按钮(**button**)使用蓝色边框作为默认值。

```
<?button name="button" extends="button" style="border:1px solid  
blue"?>  
<button/> <!-- with blue border -->
```

如想获得更多信息, 请参考 [the Developer's Reference](#).

init指令

```
<?init class="..." [arg0="..."] [arg1="..."] [arg2="..."]  
[arg3="..."]?>
```

```
<?init zscript="..." [arg0="..."] [arg1="..."] [arg2="..."]  
[arg3="..."]?>
```

有两种格式。第一种格式是指定一个类用于处理具体应用(application-specific)的初始化。第二种格式是指定一个 **zscript** 文件用于处理具体应用(application-specific)的初始化。

初始化发生在页面被赋值(evaluated)前, 并且和桌面联系在一起, 因此, 当初始化时 `getDesktop`, `getId` 和 `getTitle` 方法都会返回 **null**。可以使用 `org.zkoss.zk.ui.Execution` 接口获取当前桌面。

你可以指定许多 `init` 指令。如果你选择了第一种格式，所指定的类必须实现 `org.zkoss.zk.ui.util.Initiator` 接口。一旦指定，在页面被赋值(evaluated)前，此类的实例会被创建，并且其 `doInit` 方法会被调用。

另外，页面被赋值(evaluated)完毕后，`doFinally` 方法会被调用。当例外发生时，`doCatch` 方法会被调用。因此，`init` 指令并不限于初始化，你可以将其用于清理及错误处理。

如果你选择了第二种格式，`zscript` 文件会被赋值(evaluated)，且参数(参数 0, 参数 1,.....)会作为类型为 `Object[]` 的 `args` 调用的一个变量被传递。

更多信息请参考 [the Developer's Reference](#)。

variable-resolver指令

```
<?variable-resolver class="..."?>
```

为 `zscript` 解释器指定一个变量分解器(variable resolver)用以分解未知变量。被指定的类必须实现 `org.zkoss.xel.VariableResolver` 接口。

你可以使用多个 `variable-resolver` 指令以指定多个变量分解器。声明靠后的分解器有更高的优先级。

下面是一个 ZK 结合 Spring 框架的例子，它分解了在 Spring 框架中声明的 Java Beans，这样我们可以直接访问到它们。

```
<?variable-resolver  
class="org.zkoss.zkplus.spring.DelegatingVariableResolver"?>
```

参考 [Small Talk: ZK with Spring DAO and JDBC, Part II](#) 获取更多细节。

关于此属性的详细信息,请参考 [the Developer's Reference](#)。

import指令

```
<?import uri="..."?>
```

```
<?import uri="..." directives="..."?>
```

引入定义在另一个 ZUML 页面的指令，例如组件定义(`<?component?>`)和初始器(`<?init?>`)，若省略 `directives` 属性，则只会引入 `component` 指令和 `init` 指令。

若你想引入特定的指令，则可以指定一个指令名称的列表，以逗号分隔。例如，

```
<?import uri="/template/taglibs.zul"
directives="taglib, xel-method"?>
```

可以被引入的指令包括 `component`, `init`, `meta`, `taglib`, `variable-resolver`, 和 `xel-method`。若你想引入所有的指令, 可以为 `directives` 属性指定`*`。注意 `meta` 暗含着 `meta` 和 `link` 指令。

一个典型的应用时将一套组件定义防置在一个 ZUML 页面中, 然后在另外的 ZUML 页面引入, 这样除了系统默认的外(**additional to the system default**), 可以共享一套相同的组件定义。

```
<!-- special.zul: Common Definitions -->
<?init zscript="/WEB-INF/macros/special.zs"?>
<?component name="special" macroURI="/macros/special.zuml"
class="Special"?>
<?component name="another"
macroURI="/WEB-INF/macros/another.zuml"?>
```

假定 `Special` 类定义在 `/WEB-INF/macros/special.zs` 文件内。

然后, 其他的 ZUML 页面可以按如下方式共享一套组件定义:

```
<?import uri="special.zul"?>
...
<special/><!-- you can use the component defined in special.zul
-->
```

不同与其他指令, `import` 指令必须位于最高层次, 也就是说, 处于根组件的层次。

参考 the Developer's Reference 获取更多信息。

link和meta指令

```
<?link [href="uri"] [name0="value0"] [name1="value1"]
[name2="value2"]?>
```

```
<?meta [name0="value0"] [name1="value1"] [name2="value2"]?>
```

这些即 HTML 里所谓的头元素(**header elements**)。目前仅基于 HTML 的客户端(即浏览器)支持他们。

开发人员可以为这些头指令指定任何属性。**ZK** 仅会翻译 href 属性的 URI(使用 Executions 类的 encodeURL 方法)。**ZK** 为客户端直接产生其他的所有属性。

注意, 这些头指令只对于主 **ZUL** 页面是有效的。也就是说, 如果页面是被另其他页面或 servlet 包含的话, 这些头指令会被忽略。当然, 若一个页面为 **zhtml** 文件, 这些指令也会被忽略。

```
<?link rel="alternate" type="application/rss+xml" title="RSS
feed"
  href="/rssfeed.php"?>
<?link rel="shortcut icon" type="image/x-icon"
href="/favicon.ico"?>
<?link rel="stylesheet" type="text/css"
href="~/zul/css/ext.css.dsp"?>

<window title="My App">
  My content
</window>
```

[\[36\]](#) 语言插件在 the Component Development Guide中有描述。

ZK属性

除了(other than)用于初始化数据成员, **ZK** 属性还被用于控制关联元素(associated element)。

apply属性

```
apply="a-class-name"
```

```
apply="class1, class2,..."
```

```
apply="{EL_returns_a_class_or_a_collection_of_classes}"
```

```
apply="{EL_returns_an_instance_or_a_collection_of_Compos
er_instances}"
```

它指定了一个类, 类的集合用于初始化组件。被指定的类必须实现 org.zkoss.zk.util.Composer 接口。然后, 由于在组件及其子组件初始化之后会调用 doAfterCompose 方法, 所以你可以在 doAfterCompose 方法内进行初始化。


```
<window apply="MyComposer"/>
```

此外，你可以使用 EL 表达式指定一个 Composer 实例，或 Composer 实例的集合。

[注]：若指定了 EL 表达式，其会在组件初始化之前被赋值。所以你不能够引用此组件。此外，此属性中 EL 表达式的 `self` 变量会引用父组件(如果有的话)，或者当前页面(如果没有父组件)。。

若你想获得更多的控制，如处理异常，你也可以实现 `org.zkoss.zk.util.ComposerExt` 接口。

use属性

```
forEachEnd="a-class-name"
```

```
use="${EL_returns_a_class_or_a_class_name}"
```

指定一个类来创建一个组件(代替默认的)。在下面的例子中，`MyWindow` 被用于代替默认的 `org.zkoss.zul.Window`。

```
<window use="MyWindow"/>
```

if属性

```
if="${ an-EL-expr }"
```

指定为相关元素赋值(evaluate)的条件。换句话说，如果条件值为假(false)，关联元素及其所有子元素会被忽略。

unless属性

```
unless="${ an-EL-expr }"
```

指定不为相关元素赋值(evaluate)的条件。换句话说，如果条件值为真(true)，关联元素及其所有子元素会被忽略。

forEach属性

```
forEach="${ an-EL-expr }"
```

```
forEach="${ an-EL-expr }, a-value"
```

有两种格式。第一种，你可以不使用逗号来指定一个值。通常为一个对象集合，这样关联元素可以依靠(**against**)集合中的每个对象重复被赋值。如果没有指定或为空，此属性会被忽略。如果没有集合对象被指定，仅会被赋值一次就好像有一个单元素的集合被指定。

第二种，你可以指定一个列表，使用逗号分隔各项目。然后，对于列表中的每个值，相关联的元素会被重复执行(**he associated element will be evaluated repeatedly against each value in the list**)。

forEachBegin属性

```
forEachBegin="an-interger"
```

```
forEachBegin="${an-EL-expr}"
```

被用于 **forEach** 属性，指定迭代(**iteration**)开始处索引(从 0 开始)。如果没有指定，迭代会从第一个元素开始，即 0。

如果 **forEachBegin** 大于或等于元素的数目，则不会发生迭代。

注: **forEachBegin.index** 对于基本的集合，数组和其他类型是绝对的 (**forEachStatus.index is absolute with respect to the underlying collection, array or other type**)。例如，如果

forEachBegin 为 5，**forEachStatus.index** 的第一个值为 5。

forEachEnd属性

```
for EachEnd="${ an-EL-expr }"
```

被用于 **forEach** 属性，指定迭代(**iteration**)结束处索引(包括此)(从 0 开始)。如果没有指定，迭代会在最后一个元素处结束。

如果 **forEachEnd** 大于或等于元素的数目，则迭代会在最后一个元素处结束。

fulfill属性

```
fulfill="event-expr"
```

```
fulfill="event-expr1, event-expr2, event-expr3"
```

```
fulfill="event-expr=uri-expr"
```

```
fulfill="event-expr1, event-expr2=uri-expr2"
```

```
fulfill="uri-expr"
```

此处的 `event-expr`, `event-expr1` 和其他的类似的表达式被称为事件表达式。可以是如下的一种格式:

`event-name`

`target-id.event-name`

`id1/id2/id3.event-name`

`${el-expr}.event-name`

`uri-expr` 是一个 URI 或 EL 表达式返回的 URI。例如,

`/my/super.zul`

`${my_super_zul}`

指定何时创建子组件。默认(即没有指定 `fulfill`)情况下, 子组件会在父组件之后被创建, 当 ZUML 页面被加载时。

如果你想推迟子组件的创建, 你可以通过 `fulfill` 属性指定条件。条件有事件名称, 还有, 可选的, 目标组件的标识或路径。这意味着直到目标组件指定的事件(如果指定了)发生时, 子组件才会被处理。如果标识被省略, 则假定为同一组件。

如果指定了 EL 表达式, 则返回一个组件, 标识或路径。

参考随机存取(Load on Demand)一节获取更多细节。

使用URI表达式

若指定了 URI 表达式, ZK 加载器会创建定义在 URI 内的组件, 并像子组件一样分配它们。为了创建定义在指定 URI 内的组件, ZK 实际上调用了定义在 `Executions` 内的 `createComponents` 方法。例如, 在下面的例子中, 当按钮被按下时, ZK 加载器会调用 `Executions.createComponents("/my/super.zul", d, null)` 来为 `d div` 创建子组件。

```
<button id="b" label="open"/>
<div id="d" fulfill="b.onClick=/my/super.zul">
</div>
```

若没有指定事件表达式, ZK 加载器会立刻创建组件-在分配所有的属性及创建所有的子组件之后。在下面的例子中, ZK 会首先创建 combobox, 然后创建定义在 /my/super.zul 内的组件。

```
<div fulfill="/my/super.zul">
    <combobox/>
</div>
```

onFulfill事件

在 ZK 执行了 fulfill 条件之后, 即创建了了所有的后续组件之后, 会触发一个 org.zkoss.zk.ui.event.FulfillEvent 实例的 onFulfill 事件来通知组件做后续处理, 如果有后续事件的话。

例如, 你要使用 org.zkoss.zk.ui.Components 类的 wireVariables 方法, 必须再次调用 wireVariables, 然后在 onFulfill 事件内通知新组件。

```
<div fulfill="b1.onClick, b2.onOpen"
    onFulfill="Components.wireVariables(self, controller)">
    ...
</div>
```

forward 属性

forward="target_event_expr"

forward="original_event=target_event_expr"

这里 target_event_expr 是一个事件表达式。事件表达式被用于为一个组件指定事件。可以使用下面格式中的一个:

event-name

target-id.event-name

id1/id2/id3.event-name

\${el-expr}.event-name

此属性用将目标组件一个事件以其他事件名称跳转至另一个组件。这就是所谓的跳转条件(forward condition)。

例如, 你可以将 button 的 onClick 事件跳转至 window, 如下:

```
<window id="w" use="MyWindow">
    ...
    <button lable="Submit" forward="onClick=w.onOK"/>
</window>
```

然后，你可以在 `MyWindow` 类内处理这个任务，如下：

```
public class MyWindow extends Window {
    public void onOK() {
        //handle the submission
    }
}
```

原始的事件是可选的。若忽略，则默认为 `onClick` 事件。类似的，目标 ID 也是可选的。若忽略了，则默认为空间所有者。因此，上面的代码也可以写成：

```
<window id="w" use="MyWindow">
    ...
    <button lable="Submit" forward="onOK"/>
</window>
```

若你想跳转多个事件，则可以在 `forward` 属性内指定这些条件，用逗号(,)分隔：

```
<textbox forward="onChange=onUpdating,
onChange=some.onUpdate"/>
```

Forward事件

`Forward` 事件发送的是 `org.zkoss.zk.ui.event.ForwardEvent` 类的一个实例。你可以使用 `getOrigin` 方法获取原始事件。

为Forward事件传递信息

```
forward="originalEvent=targetId1/targetId2.targetEvent(eventData)"
```

你可以使用括号将 `forward` 事件传递特定的应用程序信息，然后添加到 `forward` 条件，如下所示。可以使用 `ForwardEvent` 类的 `getData` 方法来获取这些信息。

```
<button forward="onCancel(abort)"/>
```

上面的例子中，`getData` 方法会返回 `"abort"`。当然，你可以指定 EL 表达式来传递任何数据类型。

Forward条件内的EL表达式

```
forward="originalEvent=${el-targetPath}.targetEvent(${el-eventData})"。
```

当指定目标 ID/path 和特定的应用程序信息(亦称, 事件数据)时可以使用 EL 表达式。

```
<button forward='${mainWnd}.onOK(${c:getProperty("status")})' />
```

但是不能使用 EL 表达式指定事件名称。

ZK元素

除了被用于创建组件, ZK 元素还被用于控制 ZUML 页面。

zk 元素

```
<zk>...</zk>
```

用于聚集(aggregate)其它元素的特殊元素。不同于一个真实的组件(如 hbox 或 div), 它并不是被创建的组件树的一部分。换句话说, 它并不代表任何元素。例如:

```
<window>
  <zk>
    <textbox/>    <textbox/>
  </zk>
</window>
```

等价于:

```
<window>
  <textbox/>  <textbox/>
</window>
```

那么, 它是干什么用的呢?

一个页面中的多个根元素

由于 XML 语法的限制, 我们只能指定一个文档根, 因此, 如果有多个根组件, 你必须使用 zk 作为文档根来组织起这些根组件。

```
<?page title="Multiple Root"?>
<zk>
  <window title="First">
    ...
  </window>
  <window title="Second" if="\${param.secondRequired}">
    ...
  </window>
</zk>
```

通用的迭代组件

zk 元素，像组件一样，支持 `forEach` 属性。因此，你可以使用它根据条件产生不同类型的组件。在下面的例子中，我们假定 `mycols` 是包含几个成员的对象集合

`isUseText()`，`isUseDate()` 和 `isUseCombo()`。

```
<window>
  <zk forEach="\${mycols}">
    <textbox if="\${each.useText}"/>
    <datebox if="\${each.useDate}"/>
    <combobox if="\${each.useCombo}"/>
  </zk>
</window>
```

属性名称	描述
<code>if</code>	<p>[可选][默认: <code>true</code>]</p> <p>指定条件为这个元素赋值(evaluate).</p>
<code>unless</code>	<p>[可选][默认: <code>false</code>]</p> <p>指定条件不为这个元素赋值(evaluate).</p>
<code>forEach</code>	<p>[可选][默认: <code>ignored</code>]</p> <p>指定一个对象集合，这样 zk 元素可以依靠(against)集合中的每个对象重复被赋值。如果没有指定或为空，此属性被忽略。如果没有集合对象被指定，仅会被赋值一次就好像有一个单元素的集合被指定。</p>
<code>switch</code>	<p>[可选][默认: <code>none</code>]</p> <p>为相互排斥条件语句(mutually exclusive evaluation)提供了选择(context)。在此属性内指定的值被称为开关条件。</p> <p>仅允许 zk 元素作为其子组件。</p>

属性名称	描述
case	<p>[可选][默认: none]</p> <p>在开关语句内提供一个可选的方式。</p> <p>若其值为以斜线开始结束的字符串,则会被认为是一个正则表达式,用以匹配 switch 条件。</p> <p>你可以指定复合 case, 使用逗号分隔。</p>
choose	<p>[可选][默认: none]</p> <p>为相互排斥条件语句(mutually exclusive evaluation)提供了选择(context)。</p> <p>仅允许 zk 元素作为其子组件。</p>
when	<p>[可选][默认: none]</p> <p>在 choose 语句内提供了一个可选的方式。</p> <p>仅当条件匹配时会被执行。</p>

zscript元素

```
<zscript [language="Java"]> Scripting codes
</zscript><zscript src=" uri " [language="Java"]/>
```

定义脚本代码, 如 **Java** 代码, 当页面被赋值(evaluated)时会解释此代码。默认的语言为 **Java**(见下文)。你可以使用 **language** 属性来选择不同的语言。

zscript 元素有如上所示的两种格式。第一种格式是将脚本代码直接嵌入到页面中。第二种格式是指定一个包含脚本代码的文件。

属性名称	描述
src	<p>[可选][默认: none]</p> <p>指定包含脚本代码文件的 URI。若被指定, 脚本代码会被加载, 就像直接嵌入到文件中一样。</p> <p>src 属性支持浏览器和按照地方(locale dependent)的 URI。换言之, 你可以指定 ~ 或 * 来表示不同的上下文路径, 浏览器或本地(locale-dependent) 信息。参考国际化一章来获取细节。</p> <p>注: 文件应包含指定语言的源代码, 此代码可以被直接解释。编码必须为</p>

属性名称	描述
	UTF-8。不要指定类文件(亦称字节码)。
language	[可选][默认: Java 或 page 指令指定的语言][可选值: Java JavaScript Ruby Groovy] 指定编写脚本代码的语言。
deferred	[可选][默认: false] 是否推迟这个元素的赋值, 直到第一个同种语言的非推迟 zscript 代码需要被赋值。参考下面的如何推迟赋值一节。
if	[可选][默认: true] 指定为这个组件赋值的条件
unless	[可选][默认: false] 指定不为这个组件赋值的条件

如何推迟赋值

当要为第一个 zscript 代码赋值时, ZK 会加载解释器。例如, 在下面的例子中, 当用户点击按钮时 Java 解释器会被加载。

```
<button onClick="alert('Hi');" />
```

另一方面, 由于需要在加载页面时为 zscript 元素赋值, 所以当加载下面的 ZUML 页面时, 解释器会被加载。

```
<window>
  <zscript>
    void add() {
    }
  </zscript>
  <button onClick="add()" />
</window>
```

如果你想推迟解释器的加载, 可以将 deferred 设为 true。那么, 直到用户点击按钮时, 解释器才会被加载。

```
<window>
  <zscript deferred="true">
    void add() {
    }
  </zscript>
```

```
<button onClick="add()" />
</window>
```

[注]: 在 if, unless 和 src 属性中指定的 EL 表达式的赋值也是延迟的。

[注]: 如果到解释器时一个组件被从页面移除, zscript 代码会被忽略。例如, 前面例子中的 window 不再属于页面, 延迟的 zscript 也不会被解释。

如何选择一种不同的脚本语言

一个页面可以有多种不同的脚本语言。

```
<button onClick="javascript:do_something_in_js()" />
<zscript language="groovy">
do_something_in_Groovy();
</zscript>
```

如果省略脚本语言, 则假定为 **Java**。如果你想改变默认的脚本语言, 可以按如下方式使用 page 指令

```
<?page zscript-language="Groovy"?>

<zscript>
def name = "Hello World!";
</zscript>
```

如何支持更多的脚本语言

目前, ZK 支持 Java, JavaScript, Ruby 和 Groovy。但是, 它很容易扩展:

1. 一个实现 `org.zkoss.zk.scripting.Interpreter` 接口的类。如果你想直接处理命名空间, 可以继承 `org.zkoss.zk.scripting.util.GenericInterpreter` 类。或者, 如果解释器支持 **BSF**(Bean Scripting Framework, Bean 脚本框架), 也可以继承 `org.zkoss.scripting.bsh.BSFInterpreter` 类。
2. 在 `WEB-INF/zk.xml` 或 `zk/config.xml` 文件中声明脚本语言。

```
<zscript-config>
<language-name>SuperJava</language-name><!-- case insensitive
--!>
```

```
<interpreter-class>my.MySuperJavaInterpreter</interpreter-class>
</zscript-config>
```

参考 [the Developer's Reference](#) 来获取关于 `WEB-INF/zk.xml` 的细节。参考 [the Component Development Guide](#) 来获取关于 `zk/config.xml` 的细节。

attribute元素

定义一个闭合元素的 XML 属性。元素的内容为属性值，而 `name` 属性用于指定属性名称。如果属性值比较复杂或属性为条件式的，这比较有用。

```
<window>
  <attribute name="title" if="${new}">Untitled</attribute>
  <attribute name="title" unless="${new}">${title}</attribute>
</window>
```

这里 `ol` 和 `li` 是本地内容的一部分。它们不是 **ZK** 组件。它们最终会被转换为字符串实例，并分配给指定的 `attribute`。如果这个值有三个元素，那么上面的代码等价于：

```
<html>
  <attribute name="content"><![CDATA[
    <ol>
      <li>${values[0]}</li>
      <li>${values[1]}</li>
      <li>${values[2]}</li>
    </ol>
  ]]></attribute>
</html>
```

属性名称	描述
name	[必须] 指定属性名称。
trim	[可选][默认: false] 指定是否省略属性值开头及末尾的空格。
if	[可选][默认: none] 指定为这个元素赋值的条件。
unless	[可选][默认: none] 指定不为这个元素赋值的条件。

variables元素

定义一套变量。如果有一个父组件和页面，如果在页面级声明，它等价于 Component 的 `setVariable` 方法。

如下所述，`variables` 可以很方便的指定变量而无需编程。

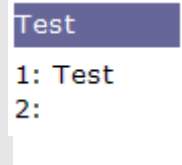
```
<window>
  <variables rich="simple" simple="intuitive"/>
</window>
```

等价于：

```
<window>
  <zscript>
    self.setVariable("rich", "simple", false);
    self.setVariable("simple", "intuitive", false);
  </zscript>
</window>
```

当然，也可以为值指定 EL 表达式。

```
<window>
  <window id="w" title="Test">
    <variables title="{w.title}"/>
    1: ${title}
  </window>
  2: ${title}
</window>
```



Test

1: Test

2:

就像 Component 的 `setVariable` 方法，你可以按如下方式控制是否为当前的 ID 空间声明局部变量。如果没有指定，则假定为 `local="false"`。

```
<variables simple="rich" local="true"/>
```

和composite属性一起使用 List 和 Map值

默认情况下，执行 EL 表达式之后，值会被直接分配给变量，如果有的话。例如，如果 `more` 为 `"orange"`，`"apple, ${more}"` 会被解释成 `"apple, orange"`，并分配给 `more` 变量。

如果你想指定一个 list 值，则可以使用 `composite` 属性，和 `list` 一起，如下。

```
<variables simple="apple, ${more}" composite="list"/>
```

然后，它会被转换为含有两个元素的 list。第一个元素是"apple"，第二个元素是"orange"。

如果你想指定一个 map 值，则可以使用 composite 属性，和 map 一起，如下。

```
<variables simple="juice=apple, flavor=${more}"
composite="map"/>
```

然后，它会被转换为一个含有两个实条目的 map。第一个为是("juice", "apple")，第二个是("flavor", "orange")。

null(The null Value)

在下面的例子中，var 是一个空字符串。

```
<variables var=""/>
```

为了为变量指定一个 null 值，可以使用下面的语句。

```
<variables var="${null}"/>
```

为一个变量分配保留名

为一个变量指定保留名，例如，forEach，你必须指定一个命名空间(除了 ZK 命名空间可以任选)，如下。

```
<variables m:forEach="a value" xmlns:m="http://whatever.com"/>
```

然后，forEach 会被认为是一个变量，而不是迭代条件。

custom-attributes元素

定义一套定制属性。定制属性为与一个特定范围相关联的对象。可接受的范围包括组件，空间，页面，桌面，会话和应用(component, space, page, desktop, session and application)。

如下所述，custom-attributes 属性可以方便的指派定制属性而无需编程。

```
<window>
  <custom-attributes main.rich="simple"
very-simple="intuitive"/>
</window>
```

等价于:

```
<window>
  <zscript>
    self.setAttribute("main.rich", "simple");
    self.setAttribute("very-simple", "intuitive");
  </zscript>
</window>
```

此外, 你可以指定定制属性的范围。

```
<window id="main" title="Welcome">
  <custom-attributes scope="desktop" shared="{main.title}"/>
</window>
```

等价于:

```
<window id="main">
  <zscript>
    desktop.setAttribute("shared", main.title);
  </zscript>
</window>
```

注意 EL 表达式依靠创建的组件被赋值。有时它是微妙的通知(it is subtle to notice)。例如, 在下面的代码中, `{componentScope.simple}` 被赋值为 `null`。为什么呢? 这是 `<label value=`

`"{componentScope.simple}"/>` 的一个快捷方式(shortcut)。换言之, 当 EL 被赋值时, 组件, `self`, 是 `lable` 而不是 `window`。

```
<window>
  <custom-attributes simple="intuitive"/>
  {componentScope.simple}
</window>
```

等价于:

```
<window>
  <custom-attributes simple="intuitive"/>
  <label value="{componentScope.simple}"/><!-- self is label not
window -->
</window>
```

[提示]：不要混淆<attribute>与<custom-attributes> 。他们毫不相干。
attribute 元素是定义闭合元素属性的一种方式，而 custom-attributes 被用于为一个特定范围指派定制属性。

属性名称	描述
scope	[可选][默认: component] 指定定制属性关联的范围。
composite	[可选][默认: none] 指定值的格式。可以为 none, list 或 map。 参考 variables 元素 章节获取更多信息。。
if	[可选][默认: none] 指定为这个元素赋值的条件。
unless	[可选][默认: none] 指定不为这个元素赋值的条件。

组件集及XML命名空间

为了在同一个 ZUML 页面内使用两套或更多套组件，XML 使用命名空间来区分不同的组件集。例如，

XUL 为 `http://www.zkoss.org/2005/zul`，而 XHTML 为 `http://www.w3.org/1999/xhtml`。

另一方面，多个页面仅使用一套组件。为了使这样的页面更容易编写，ZK 以扩展名为基础决定了默认的命名空间。例如，XUL 和 ZUL 扩展名意味着使用 XUL 命名空间。因此，开发人员只需将 ZUML 页面与合适的扩展名相关联，那么再也不用担心 XML 命名空间。

标准的命名空间

如前所述，每套组件都会和一个特定的命名空间相关联。然而，开发人员可以开发或使用第三方的组件，所以我们在这里只列出 ZK 发布的命名空间(so here we list only the namespaces that are shipped with the ZK distribution)。

命名空间

命名空间
http://www.zkoss.org/2005/zul XUL 组件集的命名空间。
http://www.w3.org/1999/xhtml XHTML 组件集的命名空间。
http://www.zkoss.org/2005/zk ZK 命名空间。指定 ZK 具体元素与属性的保留命名空间。
http://www.zkoss.org/2005/zk/native 本地命名空间。这是为内联(inline) 元素的保留命名空间。 参考 HTML 相关组件(Work with HTML Tags)一节获取细节。
native:URI-of-another-namespace 指定本地命名空间的可选方式。除了指定一个标签属于本地命名空间， native: 随后的命名空间会被生成至输出并发送至客户端。 参考 HTML 相关组件(Work with HTML Tags)一节获取细节。
http://www.zkoss.org/2005/zk/annotation 注释(Annotation)的命名空间。这是为注释保留的命名空间。 参考注释(Annotations)一节获取细节。

在 ZUML 页面中指定命名空间是可选的，直到有冲突为止。ZK 通过检测 ZUML 页面的扩展名来决定使用哪个命名空间。对于 .zul 和 .xul，使用 XUL 命名空间。对于 html，xhtml 和 zhtml，使用 XHTML 命名空间。

为了混合使用另一种标记语言，你可以使用 xmlns 来指定合适的命名空间。

```
<window xmlns:h="http://www.w3.org/1999/xhtml">
  <h:div>
    <button/>
  </h:div>
</window>
```

对于 XHTML 组件，onClick 和 onChange 属性和 ZK 的属性冲突。你可以使用保留的 <http://www.zkoss.org/2005/zk> 命名空间来解决这个问题，如下：


```
<html xmlns:x="http://www.zkoss.org/2005/zul"
xmlns:zk="http://www.zkoss.org/2005/zk">
<head>
<title>ZHTML Demo</title>
</head>
<body>
  <script type="text/javascript">
    function woo() { //running at the browser
    }
  </script>
  <zk:zscript>
    void addItem() { //running at the server
    }
  </zk:zscript>
  <x:window title="HTML App">
    <input type="button" value="Add Item"
      onClick="woo()" zk:onClick="addItem()" />
  </x:window>
</body>
```

在这个例子中，ZHTML 的 `onClick` 属性用来指定在浏览器端运行的 JavaScript 代码。另一方面，`zk:onClick` 作为保留属性来指定一个 ZK 事件处理器。

注意，命名空间的前缀，`zk`，对于 `zscript` 元素是可选的，因为 ZHTML 元素没有这个元素，而 ZK 有足够的信息来确定它。

同样也要注意你必须为 `window` 组件指定 XML 命名空间，因为其来自于一个不同的组件集。

第 7 章 ZUML 页面及 XUL 组件集

目录

[基本组件](#)

[标签](#)

[按钮](#)

[单选按钮和单选按钮组](#)

[图像](#)

[图像映射\(Imagemap\)](#)

[音频](#)

[输入控件](#)

[日历](#)

[进度条](#)

[Slider](#)

[计时器](#)

[分页](#)

[窗口](#)

[标题](#)

[closables属性](#)

[sizable属性](#)

[样式类](#)

[contentStyle属性](#)

[边框](#)

[重叠，弹出，Modal，标示和嵌入](#)

[position属性](#)

[通用对话框](#)

[布局组件](#)

[嵌套的borderlayout组件](#)

[size 和 border属性](#)

[splittable 和collapsible 属性](#)

[flex 属性](#)

[open 属性](#)

[onOpen 属性](#)

[箱式模型](#)

[spacing属性](#)

[widths 和 heights 属性](#)

[分割器](#)

[Tab箱](#)

[嵌套tab box](#)

[The Accordion Tab Boxes](#)

[orient属性](#)

[Tabs的align属性](#)

[closable属性](#)

[disabled属性](#)

[Tab面板的随机存取](#)

[网格](#)

[滚动网格](#)

[可变列宽](#)

[分页网格](#)

[排序](#)

[实况数据](#)

[辅助表头](#)

[特殊属性](#)

[更多的布局组件](#)

[Separators 和空格](#)

[Group boxes](#)

[工具栏](#)

[菜单栏](#)

- [执行一个菜单命令](#)
- [像复选框一样使用菜单项目](#)
- [autodrop属性](#)
- [onOpen事件](#)
- [更多的菜单特性](#)

[上下文菜单](#)

- [定制的tooltip及弹出菜单](#)
- [onOpen事件](#)

[列表框](#)

- [多列列表框](#)
- [栏头](#)
- [栏尾](#)
- [下拉列表](#)
- [多选](#)
- [滚动列表框](#)
- [可变列表头](#)
- [分页列表框](#)
- [排序](#)
- [特殊属性](#)
- [实况数据](#)
- [包含按钮的列表框](#)

[树控件](#)

- [open属性和onOpen事件](#)
- [多选](#)
- [分页](#)
- [特殊属性](#)
- [Tree控件的打开时创建](#)

[下拉列表框](#)

- [autodrop属性](#)
- [description属性](#)
- [onOpen事件](#)
- [onChanging事件](#)

[Bandboxes](#)

- [closeDropdown方法](#)
- [autodrop属性](#)
- [onOpen事件](#)
- [onChanging事件](#)

[图表](#)

- [实况数据](#)
- [向下钻取\(onClick事件\)](#)
- [操作区](#)

[拖放](#)

- [draggable 和 droppable属性](#)
- [onDrop 事件](#)

[使用多选拖曳](#)

[可拖曳组件的多种类型](#)

[HTML相关组件](#)

[html组件](#)

[Native命名空间, <http://www.zkoss.org/2005/zk/native>](#)

[XHTML命名空间, <http://www.w3.org/1999/xhtml>](#)

[include组件](#)

[style组件](#)

[script组件](#)

[iframe组件](#)

[用HTML FORM 和Java Servlets](#)

[name属性](#)

[支持name属性的组件](#)

[丰富用户界面](#)

[客户端行为](#)

[引用一个组件](#)

[onshow和onhide 行为](#)

[CSA JavaScript工具](#)

[事件](#)

[鼠标事件](#)

[按键事件](#)

[输入事件](#)

[List和Tree 事件](#)

[Slider和Scroll事件](#)

[其它事件](#)

本章描述了 XUL 组件集。不同于其他的实现，ZK 的 XUL 组件，经过了跨越网络合作的优化(co-operating across Internet)。有些组件可能不会完全兼容 XUL 技术标准。为方便起见，我们有时指它们为 XUL 组件。

基本组件

标签

Label 组件用来呈现一段文字。

```
<window border="normal">
  Hello World
</window>
```



如果你想为 Label 指定一个属性，并且如下方式明确指定<label>。

```
<window border="normal">
```

```
<label style="color: red" value="Hello World"/>
</window>
```

Hello World

[提示]: ZUML 为 XML 而不是 HTML, 所以并不接受` `。但是可以用` `代替。

pre, hyphen, maxlength 和multiline 属性

你可以使用 `pre` , `hyphen` , `maxlength` 和 `multiline` 属性来控制如何展示一个 `label`。例如, 如果指定 `pre` 为 `true`, 所有的空格(white spaces), 例如换行, 空白, 制表符(new line, space 和 tab)都会被保留。

hyphen	pre	maxlength	描述
false	false	positive	切去超过指定最大长度的字符
true	any	positive	如果一个单词超过了最大长度, 此单词会被截断(hypernated)
false	true	any	maxlength 被忽略
any	any	0	pyphen 被忽略

```
<window border="normal" width="100px">
<vbox id="result">

</vbox>
  <zscript><![CDATA[

    String[] s = {"this is 9", "this is ten more to show",
    "this framework", "performance is everything"};
    for (int j = 0; j < s.length; ++j) {
      Label l = new Label(s[j]);
      l.maxlength = 9;
      l.hyphen = true;
      l.parent = result;
    }
  ]]></zscript>
</window>
```

this is 9
this is
ten more
to show
this fram-
ework
performan-
ce is eve-
rything

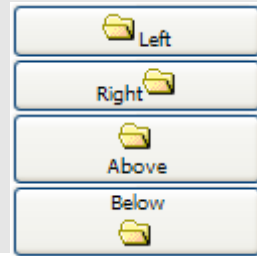
`multiline` 属性与 `pre` 类似, 除了 `multiline` 在每行的开始保留了换行和空格。

按钮

有两种类型的按钮：button 和 toolbarbutton。除了外观，它们的功能是相似的。button 组件使用 HTML BUTTON 标记，而 toolbarbutton 组件使用 HTML A 标记。

你可以使用 label 和 image 属性来为一个按钮指定标签和图像。如果这两个属性都被指定，dir 控制哪个组件显示在前方，orient 控制布局为横向或纵向。

```
<button label="Left" image="/img/folder.gif" width="125px"/>
<button label="Right" image="/img/folder.gif"
dir="reverse" width="125px"/>
<button label="Above" image="/img/folder.gif"
orient="vertical" width="125px"/>
<button label="Below" image="/img/folder.gif"
orient="vertical" dir="reverse" width="125px"/>
```



除了通过 URL 来指定图片，你可以使用 setImageContent 方法为按钮指定一个动态生成图像。参考下面的章节获取细节。

提示：所有包含 image 属性的组件都提供了 setImageContent 方法。简单来说，setImageContent 方法用于动态生成图像，而 image 用于通过 URL 指定图像。

onClick 事件 和 href 属性

有两种方式为 button 和 toolbarbutton 添加行为。首先需要为 onClick 指定一个监听器。然后为 href 属性指定一个 URL。如果都被指定，href 属性拥有更高的优先级，也就是说 onClick 事件不会被发送。

```
<button onClick="do_something_in_Java()" />
<button href="/another_page.zul"/>
```

org.zkoss.zk.ui.Execution 接口的 sendRedirect 方法

当处理一个事件时，你可以决定停止处理当前桌面，并且通过 sendRedirect 方法来转到另一个页面。换句话说，下面的例子中两个按钮是等价的(从用户的观点来看)。

```
<button
onClick="Executions.sendRedirect(&quot;another.zul&quot;)" />
<button href="another.zul"/>
```

既然 onClick 事件被送到了服务器去处理，你可以在调用 sendRedirect 前添加更多的逻辑，例如，当特定的条件被满足时转到另一个页面。

另外，href 属性在客户端方面被完全处理。当用户点击按钮时，你的应用程序并不会被察觉

单选按钮和单选按钮组

一个单选按钮是可以被打开或关闭的组件。单选按钮可以被分组，称为 radiogroup。在相同的组内，同一时间仅可以有一个按钮被选中。

```
<radiogroup onCheck="alert(self.selectedItem.label)">
  <radio label="Apple"/>
  <radio label="Orange"/>
  <radio label="Banana"/>
</radiogroup>
```

多用途设计版面

你可以混合使用 radiogroup 和 radio 来组成你想要的布局，如下所示。

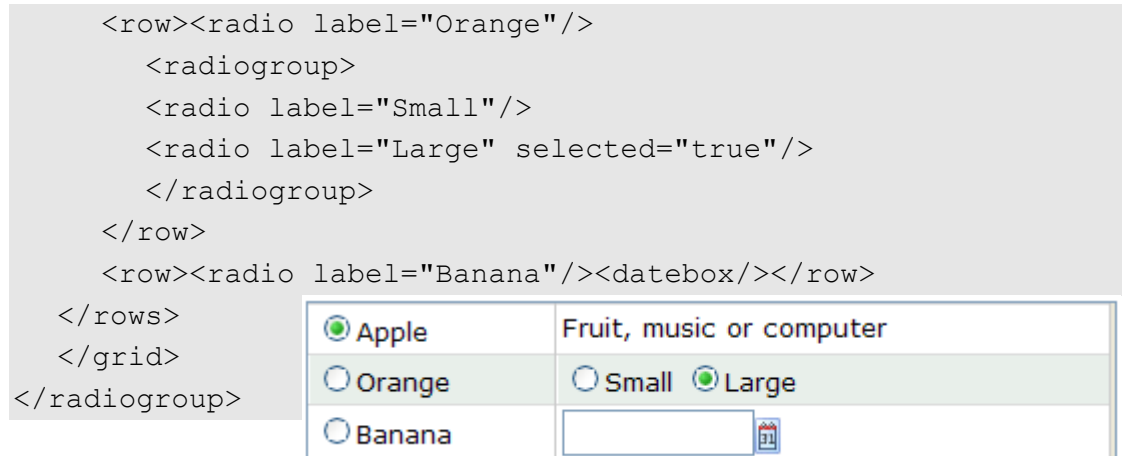
```
<radiogroup>
  <grid>
    <rows>
      <row><radio label="Apple" selected="true"/> Fruit, music or
computer</row>
      <row><radio label="Orange"/><textbox/></row>
      <row><radio label="Banana"/><datebox/></row>
    </rows>
  </grid>
</radiogroup>
```

<input checked="" type="radio"/> Apple	Fruit, music or computer
<input type="radio"/> Orange	<input type="text"/>
<input type="radio"/> Banana	<input type="text"/>

单选按钮属于离其最近的 radiogroup。你甚至你可按如下方式嵌套使用 radiogroup。每个

radiogroup 是独立的，虽然可能有某种视觉重叠(visual overlap)。

```
<radiogroup>
  <grid>
    <rows>
      <row><radio label="Apple" selected="true"/> Fruit, music or
computer</row>
```



图像

`image` 用于在浏览器端展示图像。有两种方式来为 `image` 组件指定一个图像。一种方法是使用 `src` 属性指定图像的 **URI**，这种方法与 **HTML** 支持的相似。如果你想展示一张景台图像，或任何可以通过 **URL** 定位的图像，这种方法很有用。

```
<image src="/some/my.jpg"/>
```

本地图像

就像使用其他可以接受 **URI** 的属性一样，你可以指定 `"*"` 来定位一张本地图像。例如，如果对不同的地区对应不同的图像，你可以按如下方式使用。

```
<image src="/my*.png"
```

然后假定你的用户以 `de_DE` 作为首选地区访问你的页面。`ZK` 会设法定位名称为 `/my_de_DE.png` 的图像。如果没有找到，则会尝试 `/my_de.png`，最后是 `/my.png`。

参考国际化一章中浏览器和本地化 **URI** 一节来获取细节。

第二种方法是使用 `setContent` 方法来直接为 `image` 组件指定图像的内容。一旦被指定，图像会展示在浏览器端并会被动态更新。这种方法指定适用于动态生成的图像。

例如，你可以按如下方式为用户指定的位置生成一个映射。

```
Location: <textbox onChange="updateMap(self.value)"/>
Map: <image id="image"/>
<zscript>
  void updateMap(String location) {
    if (location.length() > 0)
      image.setContent(new MapImage(location));
  }
}
```



```
</zscript>
```

在上面的例子中，我们假定有一个 `MapImage` 类用于产生指定位置的映射，即所谓的商业逻辑(business logic)。

注意，`image` 组件仅接受 `org.zkoss.image.Image` 中的内容。如果有工具生成的图像不是这个格式，可以使用 `org.zkoss.image.AImage` 类来将一个二进制阵列数据，文件或输入流包装成 `Image` 接口

在传统的 Web 应用程序中，缓存一个动态生成的图像是很复杂的。有了 `image` 组件，你就不需要担心这些了。一旦指定了一个图像的内容，它就属于 `image` 组件，而且当 `image` 组件不再被使用时，它所占用的内存会被自动释放。

提示：如果你想指定非图像音频文件的内容，例如 PDF，可以使用 `iframe` 组件。参考相关章节获取细节。

图像映射(Imagemap)

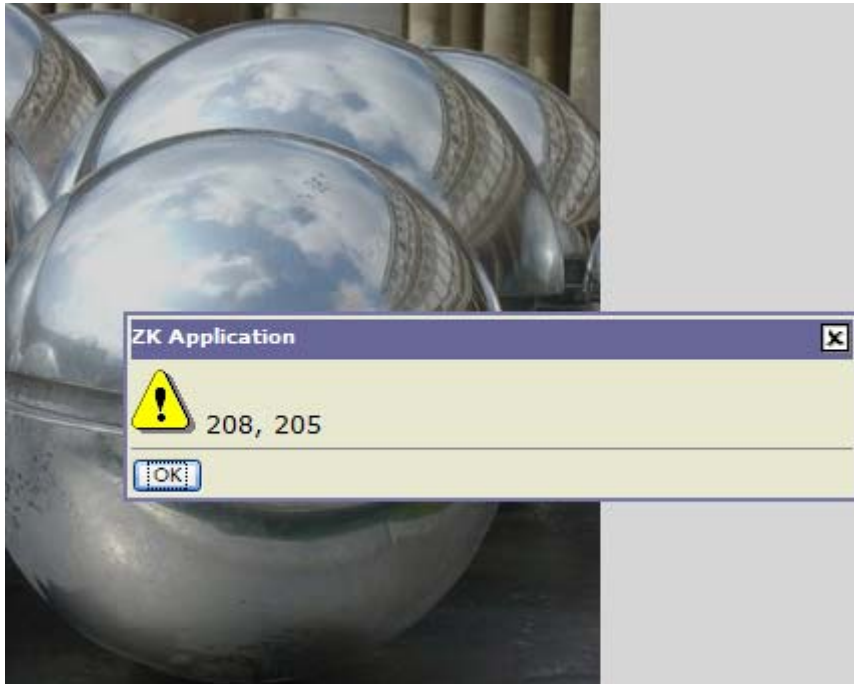
`imagemap` 是一个特殊的 `image` 组件。它接受 `image` 组件的所有属性。但是不同于 `image`，当用户点击一张图像时，`onClick` 事件及鼠标坐标的位置会一起被送回服务器。相比之下，`image` 发送的 `onClick` 事件不包含坐标。

鼠标位置的坐标是屏幕像素，从图像的左上角开始计数，始于(0, 0)。这将会作为

`org.zkoss.zk.ui.event.MouseEvent` 的实例存储。一旦应用程序接收了 `onClick` 事件，就可以通过 `getX` 和 `getY` 方法来检查鼠标位置的坐标。

例如，如果用户点击了下列语句展示图像的(137,167)像素(从左上角开始)，那么用户就会得到如下所示的结果：

```
<imagemap src="/img/sun.jpg" onClick="alert(event.x + '&quot;, &quot; + event.y)"/>
```



应用程序通常使用坐标来决定用户点击了那个部分，然后作出相应的相应。

区域

通过为 `imagemap` 然后，组件添加 `area` 子组件，开发人员可以代替使用应用程序本身处理坐标的方法。

```
<imagemap src="/img/sun.jpg" onClick="alert(event.area)">
  <area id="First" coords="0, 0, 100, 100"/>
  <area id="Second" shape="circle" coords="200, 200, 100"/>
</imagemap>
```

然后，`imagemap` 组件将鼠标的位置坐标翻译成一个逻辑名字：用户点击的区域标识。

例如，如果用户点击(150,150)，将会得到如下描绘的结果：



shape属性

area 组件支持三种形状：圆，多边形和矩形(circle, polygon and rectangle)。鼠标位置的坐标是屏幕像素，从图像的左上角开始计数，始于(0, 0)。

形状	坐标/描述
circle	<code>coords="x, y, r"</code> x , y 定义圆心坐标， r 为半径，以像素为单位。
polygon	<code>coords="x1, y1, x2, y2, x3, y3..."</code> 一对 x,y 定义多边形的一个定点。定义三角形至少需要三对坐标。多边形会自动关闭，所以不需要为了关闭区域而在列表的末尾重复第一个坐标。
rectangle	<code>coords="x1, y1, x2, y2"</code> 第一对坐标为矩形的一个角，另一对为斜对角。矩形只是为多边形指定四个顶点的简单方式。

如果一个 area 组件内的坐标覆盖了另外一个，第一个拥有更高的优先权。

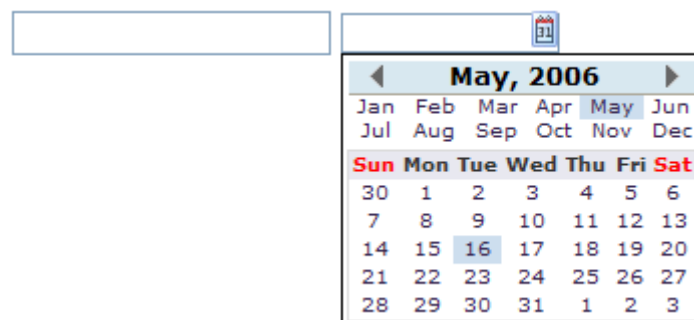
音频

audio 组件用来在浏览器端播放音频。就像 image，你可以使用 src 属性来指定音频资源的 URL，或使用 setContent 方法来指定一段动态生成的音频。

依靠浏览器和音频插件，开发人员可以使用 `play`，`stop` 和 `pause` 方法来控制播放一段音频。目前，包含多媒体的 Internet 浏览器都有这种控制机制。

输入控件

XUL 组件支持一套输入控制组件：`textbox`，`intbox`，`decimalbox`，`doublebox`，`datebox`，`combobox` 和 `bandbox`，用于输入各种类型的数据。



```
<zk>
  <textbox/>
  <datebox/>
</zk>
```

提示：`combobox` 和 `bandbox` 是特殊的输入框。他们共享这里描述的公用属性。关于它们各自独特的特点，将会在之后的 `combobox` 和 `bandbox` 章节中讨论。

type属性

你可以为 `textbox` 组件指定值为 `password` 的 `type` 属性。这样将不会显示用户输入的内容。

```
Username: <textbox/>
Password: <textbox type="password"/>
```

format属性

通过使用 `format` 域，可以控制输入控件的格式。默认为 `null`。对于 `datebox`，意味着 `yyyy/MM/dd`。而对与 `intbox` 和 `decimalbox`，则以为着根本没有格式。

```
<datebox format="MM/dd/yyyy"/>
<decimalbox format="#.##0.##"/>
```

就像其他的任何属性，你可以动态的改变格式，如下所示：

```
<datebox id="db"/><button label="set MM-dd-yyyy"
onClick="db.setFormat('&quot;MM-dd-yyyy&quot;')"/>
```

无鼠标输入 datebox

1. Alt+DOWN 弹出日历 .
2. LEFT , RIGHT , UP 和 DOWN 改变日历中选中的日期.
3. ENTER 将选中的日期复制到 date.
4. Alt+UP 或 ESC 放弃选择并关闭日历. .

约束

使用 constraint 属性可以控制输入控件接受什么值。它可以使 no positive , no negative , no zero , no empty , no future , no past , no today 和一个正则表达式的集合。前三个约束金使用于 intbox 和 decimalbox。no future , no past,和 no today 约束仅适用于 datebox。no empty 适用于任何组件。正则表达式约束仅适用于字符串类型组件，例如 textbox , combobox 和 bandbox。

使用两个或更多的约束时，用逗号翻开约束，如下：

```
<intbox constraint="no negative,no zero"/>
```

为了指定一个正则表达式，你可以使用/来包围表达式。如下：

```
<textbox constraint="/.+@.+\.[a-z]+/">
```

注：

1. 上面的语句为 XML，所以不必使用\\来表示反斜杠。另一方面，如果写在 Java 代码中，则是需要的。

```
new Textbox().setContraint("/.+@.+\.[a-z]+/");
```

1. 允许混合使用正则表达式与其他约束，但要用逗号分隔。

如果你想为应用程序展示定制的信息，而不是默认的，可以添加约束及验证失败后的信息，用

冒号隔开。

如果你想为应用程序展示定制的信息，而不是默认的，可以添加约束及验证失败后的信息，用

冒号隔开。

```
<textbox constraint="/.+@.+\.[a-z]+/: e-mail address only"/>
<datebox constraint="no empty, no future: now or never"/>
```

注：

1. 如果指定了错误信息，那么它必须为末尾元素，且以冒号开始。
2. 为了支持多语言，你可以使用 I 函数，就像在国际化(Internationalization)一章中描述的那样。

```
<textbox constraint="/.+@.+\.[a-z]+/: ${c:l('err.email.required')}}"/>
```

Datebox的约束

除了在上述章节中描述的约束(例如 no future 和正则表达式)，datebox 支持一个日期范围。例如，

```
<datebox constraint="between 20071225 and 20071203"/>
<datebox constraint="before 20071225"/>
<datebox constraint="after 20071225"/>
```

注意

1. 日期的格式被约束为 yyyyMMdd，独立于区域。
2. 约束内指定的日期为包含。例如，"before 20071225"包括 December 25, 2007 及之前的每一天。
3. 约束实际上表示 org.zkoss.zul.SimpleDateConstraint 类的一个实例。你可以使用 getBeginDate 和 getEndDate 方法分别获取开始和结束日期。
4.

```
((SimpleDateConstraint)datebox.getConstraint()).getBeginDate();
```

定制属性

如果你想使用更复杂的约束，可以指定一个实现了 org.zkoss.zul.Constraint 接口的对象。

```

<window title="Custom Constraint">
  <zscript><![CDATA[
Constraint ctt = new Constraint() {
  public void validate(Component comp, Object value) throws
WrongValueException {
    if (value == null || ((Integer)value).intValue() < 100)
      throw new WrongValueException(comp, "At least 100 must be
specified");
  }
}
  ]]></zscript>
  <intbox constraint="{ctt}"/>
</window>

```

你可以在一个 Java 类中实现你的约束，例如 my.EmailValidator，那么

```

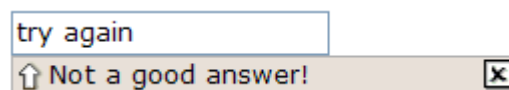
<?taglib uri="/WEB-INF/tld/web/core" prefix="c"?>
<textbox constraint="{c:new('my.EmailValidator')}" />

```

org.zkoss.zk.ui.WrongValueException

在上面的例子中，我们使用了 org.zkoss.zk.ui.WrongValueException 来表述一个错误。

就像描绘的那样，你需要指定两个参数，第一个为引起错误的组件，第二个为错误信息。你可以随时丢出这个异常，如下面，当 onChange 事件被接收时：



```

<textbox>
  <attribute name="onChange">
    if (!self.value.equals("good")) {
      self.value = "try again";
      throw new WrongValueException(self, "Not a good answer!");
    }
  </attribute>
</textbox>

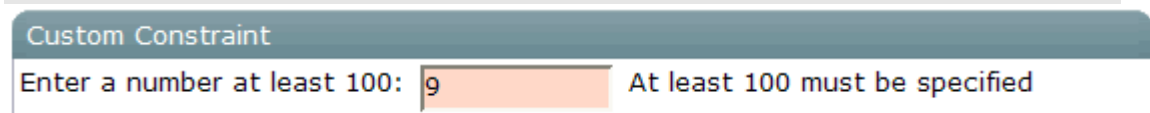
```

定制方式显示错误信息

通过实现 org.zkoss.zul.CustomConstraint 及 Constraint 接口，你可以提供定制的外观，以代替前面例子中的默认错误框。CustomConstraint 有一个

showCustomError 方法，当抛出异常或验证失败时，此方法会被调用。下面是一个例子：

```
<window title="Custom Constraint" border="normal">
  <zscript><![CDATA[
class MyConst implements Constraint, CustomConstraint {
  //Constraint//
  public void validate(Component comp, Object value) {
    if (value == null || ((Integer)value).intValue() < 100)
      throw new WrongValueException(comp, "At least 100 must be
specified");
  }
  //CustomConstraint//
  public void showCustomError(Component comp,
WrongValueException ex) {
    errmsg.setValue(ex != null ? ex.getMessage(): "");
  }
}
Constraint ctt = new MyConst();
  ]]></zscript>
  <hbox>
    Enter a number at least 100:
    <intbox constraint="{ctt}"/>
    <label id="errmsg"/>
  </hbox>
</window>
```



提高响应能力

在客户端验证更多的约束可以提高能力。你得实现 org.zkoss.zul.ClientConstraint 及 Constraint 接口来实现此功能。如果完成了客户端的所有验证，可以通

isClientComplete 方法返回 true，那么将不会有回调服务(server callback)。

你也可以使用纯 JavaScript 代码来定制一个错误信息的显示，需要提供一个叫作 Validate_errorbox 的函数。如下：

```
<script type="text/javascript"><![CDATA[
  //Running at the browser
  window.Validate_errorbox = function (id, boxid, msg) {
```



```

        var html = '<div style="display:none;position:absolute"
id="'
            +boxid+'">'+zk.encodeXML(msg, true)+'</div>';
        document.body.insertAdjacentHTML("afterbegin", html);
return $e(boxid);    }
]]></script>

```

[注]: script 指定了在浏览器端运行的脚本代码类型, 而 script 在服务器端运行的脚本代码类型.

[注]: 如果也实现了 CustomConstraint, 那么由于在服务端完成了所有的验证, 所以 ClientConstraint 会被忽略。换言之, 如果你想使用 ClientConstraint 提高响应能力, 重写 Validate_errorbox 是定制错误信息显示的唯一方式。

onChange事件

当用户已经改变了输入控件的内容, 输入控件会使用 onChange 事件来通知应用程序。

注意, 当 onChange 的事件监听器被调用时, 其值已经被设定。因此, 如果当你想为 onChange 的事件监听器注入一个非法值时已经晚了, 除非你适当地还原其值。建议使用在定制约束(Custom Constraints)章节描述的约束。

onChanging事件

当用户正在改变了输入控件的内容时, 输入控件会使用 onChanging 事件来通知应用程序

注意, 当 onChanging 的事件监听器被调用时, 其值并没有被设定。换言之, value 属性原来的值。为了获取用户输入的内容, 你得按如下方式访问事件的 value 属性。

```

<grid>
    <rows>
        <row>The onChanging textbox:
            <textbox onChanging="copy.value = event.value"/></row>
        <row>Instant copy:
            <textbox id="copy" readonly="true"/></row>
    </rows>
</grid>

```

由于用户还没有更改, 所以 onChanging 的事件监听器非法值就太早了。建议使用在定制约束(Custom Constraints)章节描述的约束。

日历

calendar 展示了一个日历并允许用户从中选择一个日期。

```
<hbox>
<calendar id="cal" onChange="in.value = cal.value"/>
<datebox id="in"
onChange="cal.value =
in.value"/>
</hbox>
```



value 属性和onChange事件

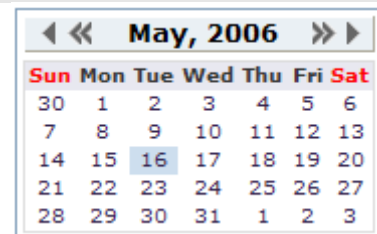
就像输入控件，calendar 提供了 value 属性来开发人员设置及获取选中的日期。此外如果有必要的话，开发人员还可以监听 onChange 事件以便立即加以处理。

compact属性

calendar 支持两种不同的布局，可以通过 compact 属性控制。

```
<calendar compact="true"/>
```

默认值为本地。



进度条

progress 为指示任务完成进度的棒形图。其 value 属性必须在 0 到 100 的范围内取值。

```
<progressmeter value="10"/>
```



Slider

Slider 以滚动(scrolling) 方式来指定值。

```
<slider id="slider" onScroll="Audio.setVolume(slider.curpos)"/>
```

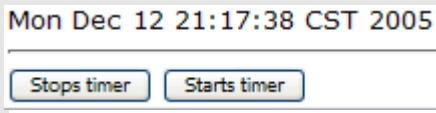


slider 接收在 0 至 100 范围内的值。你可以使用 maxpos 属性来改变允许的最大值 。

计时器

`timer` 是一个不可见的组件，用于在指定的时刻或一段时间内将 `onTimer` 事件发送到服务器。你可以使用 `start` 和 `stop` 方法来控制 `timer`。

```
<window title="Timer demo" border="normal">
  <label id="now"/>
  <timer id="timer" delay="1000" repeats="true"
    onTimer="now.setValue(new Date().toString())"/>
  <separator bar="true"/>
  <button label="Stops timer"
onClick="timer.stop()" />
  <button label="Starts timer" onClick="timer.start()" />
</window>
```



分页

`paging` 组件用于将一段很长的内容分成多个页面。例如，假定有 100 个项目，每次显示 20 个项目，那么可以按如下方式使用 `paging` 组件。

```
<paging totalSize="100" pageSize="20"/>
```

[Prev](#) [1](#) [2](#) [3](#) [4](#) [5](#) [Next](#)

然后，当用户点击一个链接时，`onPaging` 事件会和 `org.zkoss.zul.event.PagingEvent` 的一个实例被送到 `paging` 组件。可以为 `paging` 组件添加一个监听器以决定 100 个项目的哪部分是可见的。

```
<paging id="paging"/>
<zscript>
  List result = new SearchEngine().find("ZK");
  //assume SearchEngine.find() will return a list of items.
  paging.setTotalSize(result.size());
  paging.addEventListener("onPaging", new EventListener() {
    public void onEvent(Event event) {
      int pgno = event.getPaginal().getActivePage();
      int ofs = pgno * event.getPaginal().getPageSize();
      new Viewer().redraw(result, ofs, ofs +
event.getPaginal().getPageSize() - 1);
      //assume redraw(List result, int b, int e) will display
      //from the b-th item to the e-th item
    }
  });
</zscript>
```

List Boxes 和 Grids的Paging

listbox 和 grid 组件本身支持 paging, 所以不必像上面一样明确地为其指定 paging 组件, 除非你想指定不同的布局(visual layout)或者使用一个 paging 组件控制多个 listbox 和 grid。

参考网格一节获取更多的细节。

窗口

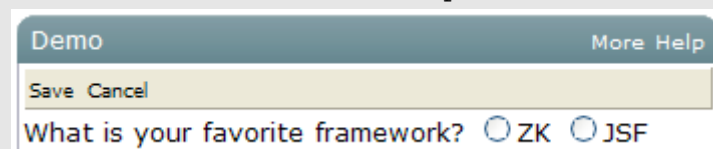
window, 就像 HTML 的 DIV 标签, 用于为组件分组。不同于其它的组件, window 有如下的特点。

1. window 是一个 ID 空间的所有者。window 可以包含任意组件, 包括其自身。如果通过标识指定, 可以使用 getFellow 方法来找到它。
2. window 可以被重叠, 弹出和嵌入(overlapped, popup, and embedded)。
3. window 可以是对话框(modal dialog)。

标题

window 可以有一个 title, 一个 caption 和一个 border。标题(title) 是 title 属性指定的。标题 (caption)是 caption 组件声明的。caption 组件的所有子组件都会出现在 title 的右边。

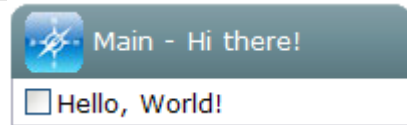
```
<window title="Demo" border="normal" width="350px">
  <caption>
    <toolbarbutton
label="More"/>
    <toolbarbutton
label="Help"/>
  </caption>
  <toolbar>
    <toolbarbutton label="Save"/>
    <toolbarbutton label="Cancel"/>
  </toolbar>
  What is your favorite framework?
  <radiogroup>
    <radio label="ZK"/>
    <radio label="JSF"/>
  </radiogroup>
</window>
```



你可以为 `caption` 指定标签和图像，其外观如下。

```
<window id="win" title="Main" border="normal" width="200px">
  <caption image="/img/coffee.gif" label="Hi there!"/>
  <checkbox label="Hello, World!"/>
</window>
```

`closables`属性

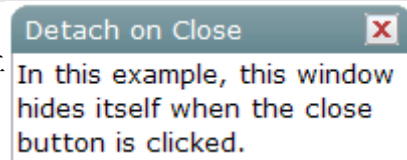


若将 `closable` 属性值设为 `true`，`close` 按钮会显示在 `window` 组件中，这样可以关闭此 `window`。一旦用户点击了 `close` 按钮，`onClose` 事件会被送到 `window`。这样此事件会被 `window` 的 `onClose` 方法处理。那么，`onClose` 在默认情况下会把 `window` 自身移除。

你也可以重定义此方法，使其做你想做的事情。或者，注册一个监听器来改变默认的行为。例如，你可以选择隐藏而不是关闭。

```
<window closable="true" title="Detach on Close" border="normal"
width="200px"
onClose="self.visible = false; event.stopPropagation();">
  In this example, this window hides itself when the close button
  is clicked.
</window>
```

注意，必须调用 `event.stopPropagation()` 阻止 `Window.onClose()` 被调用。



提示：若为一弹出 `window`，当由于用户点击 `window` 的外围或按下 `ESC` 键关闭此弹出 `window` 时，`onOpen` 事件(`open=false`) 会被送至 `window`。

这会有些困惑，但是 `onClose` 被送至服务器来询问移除还是隐藏 `window`。默认情况下，`window` 会被移除。当然，应用程序可以重定义此方法并且让其做任何事情，就像上面描述的一样。

另外，`onOpen` 是一个通知(notification)。此事件被送出以通知应用程序客户端已经隐藏了 `window`。应用程序并不能阻止 `window` 被隐藏，或改变被移除的行为。

`sizable`属性

如果你允许用户重定义 `window` 的大小，可以将 `sizable` 属性设为 `true`。这样用户可以拖曳边框来改变

`window` 的尺寸。

```
<window id="win" title="Sizable Window" border="normal"
width="200px" sizable="true">
    This is a sizable window.
    <button label="Change Sizable" onClick="win.sizable
= !win.sizable"/>
</window>
```

onSize事件

一旦用户改变了 `window` 的尺寸, `onSize` 事件及的 `org.zkoss.zul.event.SizeEvent` 一个实例会被送出。注意改变 `window` 的大小是在 `onSize` 事件被送出前发生的。换言之, 事件就像一个通知那样服务。当然, 你可以在事件监听器中做任何事情。

注: 如果用户拖曳上边框或左边框, `onMove` 事件也会被送出, 因为位置也改变了。

样式类

Zk 的 `window` 支持四种样式类: `embedded`, `overlapped`, `popup` 和 `wndcyan`。当然, 你可以添加更多。

默认情况下, `sclass` 属性与 `window` 模式是一样的, 所以不同模式 `window` 的外观是不同的。要想改变外观, 可以按下例描述的那样简单的为 `sclass` 属性指定一个值。

```
<hbox>
    <window title="Embedded Style" border="normal" width="200px">
        Hello, Embedded!
    </window>
    <window title="Cyan Style" sclass="wndcyan" border="normal"
width="200px">
        Hello, Cyan!
    </window>
    <window title="Popup Style" sclass="popup" border="normal"
width="200px">
        Hello, Popup!
    </window>
    <window title="Modal Style" sclass="modal" border="normal"
width="200px">
        Hello, Modal!
    </window>
</hbox>
```

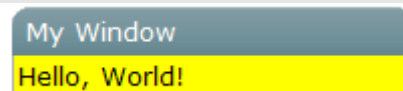


contentStyle属性

通过指定 contentStyle 属性可以改变 window 中内容块的亲自体会(look and feel)。

```
<window title="My Window" border="normal" width="200px"
contentStyle="background:yellow">
  Hello, World!
</window>
```

滚动窗口

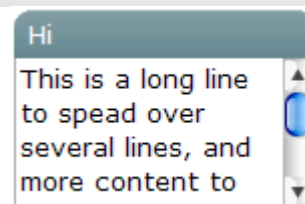


contentType 的一个典型应用是使一个 window 变得可滚动。

```
<window id="win" title="Hi" width="150px" height="100px"
contentStyle="overflow:auto" border="normal">
This is a long line to spead over several lines, and more content
to display.
Finally, the scrollbar becomes visible.
This is another line.
</window>
```

边框

border 属性是否显示 window 的边框。默认的样式仅支持 normal 和 none。默认为 none。

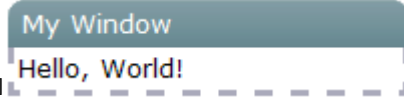


当然，你可以提供额外的样式类(style class)。例如，

```
<zk>
  <style>
    div.wc-embedded-dash {
      padding: 2px; border: 3px dashed #aab;
    }
  </style>
  <window title="My Window" border="dash" width="200px">
    Hello, World!
```

```
</window>
</zk>
```

wc-embedded-dash定义了window 内框(inner box)的样式。样式类通过连锁(concatenating)wc ^[37]命名, sclass属性和border属性是一起的, 并且以横线(-)分开它们。在此例中, 由于此窗口是嵌入式的(embedded), 所以为sclass为embedded, 而且没有明确的sclass被指定(所以使用了默认的sclass)。



重叠, 弹出, Modal, 标示和嵌入

window 有四种模式: 重叠, 弹出, Modal, 标示和嵌入。嵌入为默认模式。可以使用 doOverlapped, doPopup, doModal, doHighlighted 和 doEmbedded 方法来改变模式。如下,

```
<zk>
  <window id="win" title="Hi!" border="normal" width="200px">
    <caption>
      <toolbarbutton label="Close"
onClick="win.setVisible(false)"/>
    </caption>
    <checkbox label="Hello, Wolrd!"/>
  </window>

  <button label="Overlap" onClick="win.doOverlapped();"/>
  <button label="Popup" onClick="win.doPopup();"/>
  <button label="Modal" onClick="win.doModal();"/>
  <button label="Embed" onClick="win.doEmbedded();"/>
  <button label="Highlighted" onClick="win.doHighlighted();"/>
</zk>
```

嵌入

嵌入 window 和其它组件一起被插入在文字之间。在这种模式中, 你不能改变 window 的位置, 因为其位置是由浏览器决定的。

重叠

重叠 window 和其它组件是重叠的, 这样用户可以到处拖曳此 window, 并且开发人员可以使用 setLeft 和 setTop 方法来设置其位置。

除了 `doOverlapped`，你也可以按如下方式使用 `mode` 属性，

```
<window title="My Overlapped" width="300px" mode="overlapped">
</window>
```

弹出

弹出 `window` 与重叠 `window` 类似，除了弹出 `window` 在用户点击任一组件(不包括弹出 `window` 本身及其所有子组件)时会自动关闭。就像其名字一样，它被设计成实现弹出 `window`。

Modal

modal `window` (亦称对话框，`modal dialog`)与重叠 `window` 类似，除了 modal `window` 可以挂起执行，直到 `endModal`, `doEmbedded`, `doOverlapped`, `doHighlighted`, 和 `doPopup` 方法中的一个被调用。

除了挂起执行，modal `window` 还可以使不属于其的组件失效。

modal `window` 会自动位于浏览器的中央，你可以控制它的位置。

标示

标示 `window` 与重叠 `window` 类似，除了标示 `window` 的视觉效果与 modal `window` 相同。换言之，标示 `window` 位于浏览器的中央，并且使不属于其的组件失效。

但是，标示 `window` 不会挂起执行。就像重叠 `window`，一旦模式改变，执行会继续。例如，仅当 `win1` 被关闭后 `f1()` 才会被调用，而在 `win2` 变为标示后 `g()` 被立即调用。

```
win1.doModal(); //the execution is suspended until win1 is closed
f1();

win2.doHighlighted(); //the execution won't be suspended
g1()
```

如果你不喜欢挂起事件处理线程，可以使用标示 `window` 代替 modal `window`。参考特性提示一章中使用 `Servlet` 线程处理事件的一节。

Modal窗口和事件监听器

不同于其它的模式，你仅能在事件监听器中将 `window` 放置在 `modal` 模式中。换言之，你可以在事件监听器中调用 `doModal()` 或 `setMode("modal")`。

```
<zk>
  <window id="wnd" title="My Modal" visible="false"
width="300px">
    <button label="close" onClick="wnd.visible = false"/>
  </window>
  <button label="do it" onClick="wnd.doModal()" />
</zk>
```

另外，如果在组件创建阶段(Component Creation Phase)^[38] 执行，下面的例子是错误的。

```
//t1.zul
<window title="My Modal" width="300px" closable="true"
mode="modal">
</window>
```

如果在浏览器中直接打开，将会导致如下结果^[39]。

HTTP Status 500 -

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

```
com.potix.zk.ui.WrongValueException: doModal() and setMode("modal")
    com.potix.zul.html.Window.doModal(Window.java:249)
    sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
    java.lang.reflect.Method.invoke(Unknown Source)
    bsh.Reflect.invokeMethod(Unknown Source)
    bsh.Reflect.invokeObjectMethod(Unknown Source)
    bsh.Name.invokeMethod(Unknown Source)
    bsh.BSHMethodInvocation.eval(Unknown Source)
    bsh.BSHPrimaryExpression.eval(Unknown Source)
    bsh.BSHPrimaryExpression.eval(Unknown Source)
    bsh.Interpreter.eval(Unknown Source)
    bsh.Interpreter.eval(Unknown Source)
    com.potix.zk.ui.impl.bsh.BshInterpreter.interpret(BshInterpreter.java:100)
    com.potix.zk.ui.impl.PageImpl.interpret(PageImpl.java:524)
    com.potix.zk.ui.impl.UiEngineImpl.execCreate(UiEngineImpl.java:100)
    com.potix.zk.ui.impl.UiEngineImpl.execCreateChild(UiEngineImpl.java:100)
    com.potix.zk.ui.impl.UiEngineImpl.execCreate(UiEngineImpl.java:100)
    com.potix.zk.ui.impl.UiEngineImpl.execNewPage(UiEngineImpl.java:100)
    com.potix.zk.ui.http.DHtmlLayoutServlet.process(DHtmlLayoutServlet.java:100)
    com.potix.zk.ui.http.DHtmlLayoutServlet.doGet(DHtmlLayoutServlet.java:100)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:689)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
```

note The full stack trace of the root cause is available in the Apache Tomcat/5.5.12 logs.

Apache Tomcat/5.5.12

下面的代码会导致相同的结果。

```
//t2.zul
```

```
<window title="My Modal" width="300px" closable="true">
  <zscript>
    self.doModal();
  </zscript>
</window>
```

如果你需要在页面加载时创建一个 **modal window**，可以按如下方式提交 **onModal** 事件。

```
//t3.zul
<window title="My Modal" width="300px" closable="true">
  <zscript>
    Events.postEvent("onModal", self, null);
  </zscript>
</window>
```

注：下面的代码会正确执行，甚至 **t1.zul** 将 **window** 直接设置为 **modal** 模式(就像上面一样)。为什么呢？它在一个事件监听器中执行(**onClick**)。

```
<button label="do it">
  <attribute name="onClick">
    Executions.createComponents("t1.zul", null, null);
    //it loads t1.zul in this event listener for onClick
  </attribute>
</button>
```

position属性

除了 **left** 和 **top** 属性，你可以使用 **position** 属性来控制重叠/弹出/modal **window** 的位置。例如，下面的代码片断将 **window** 置于右下角。

```
<window width="300px" mode="overlapped" position="right,bottom">
...
```

position 属性可以是下列常量的集合，各常量之间以逗号 (,) 隔开。

常量	描述
center	将 winow 组件放置在中间。若制定了 left 和 right 属性，则为垂直中心。若指定了 top 和 buttom 属性，则为水平中心。若均未指定，则以为着在两个方向均居中。 left 和 top 属性被忽略。

常量	描述
left	将 winow 组件在左边。 left 属性被忽略。
right	将 winow 组件在右边。 left 属性被忽略。
top	将 winow 组件在顶部。 top 属性被忽略。
bottom	将 winow 组件在底部。 top 属性被忽略。

默认情况下，其值为 `null`。也就是，重叠和弹出 `window` 的位置由 `left` 和 `top` 决定，而 `modal window` 居中。

通用对话框

XUL 组件集支持下列通用对话框来简化一些通用任务。

消息框

`org.zkoss.zul.Messagebox` 类提供了一套功能来显示消息框。典型应用是当发生错误时警告用户，或促使用户作出决定。

```
if (Messagebox.show("Remove this file?", "Remove?",
Messagebox.YES | Messagebox.NO, Messagebox.QUESTION) ==
Messagebox.YES) {
    ...//remove the file
}
```

由于警告用户有误是很平常的，所以一个称为 `alert` 的全局函数被加进了 `zscript`。`alert` 函数是 `Messagebox` 类中 `show` 方法的一个捷径。换言之，下列两条语句是等价的。

```
alert("Wrong");
Messagebox.show("Wrong");
```

注意 `Messagebox` 为一个 `modal window`，所以它也受相同的约束：仅在事件监听器中是可执行的。因此，下列代码将会失败。参考上面的 **Modal** 窗口和事件监听器一节获取更多的描述。

```

<window title="MessageBox not allowed in paging loading">
  <zscript>
    //failed since show cannot be called in paging loading
    if (MessageBox.show("Redirect?", "Redirect?",
      MessageBox.YES | MessageBox.NO, MessageBox.QUESTION) ==
      MessageBox.YES)
      Executions.sendRedirect("another.zul");
  </zscript>
</window>

```

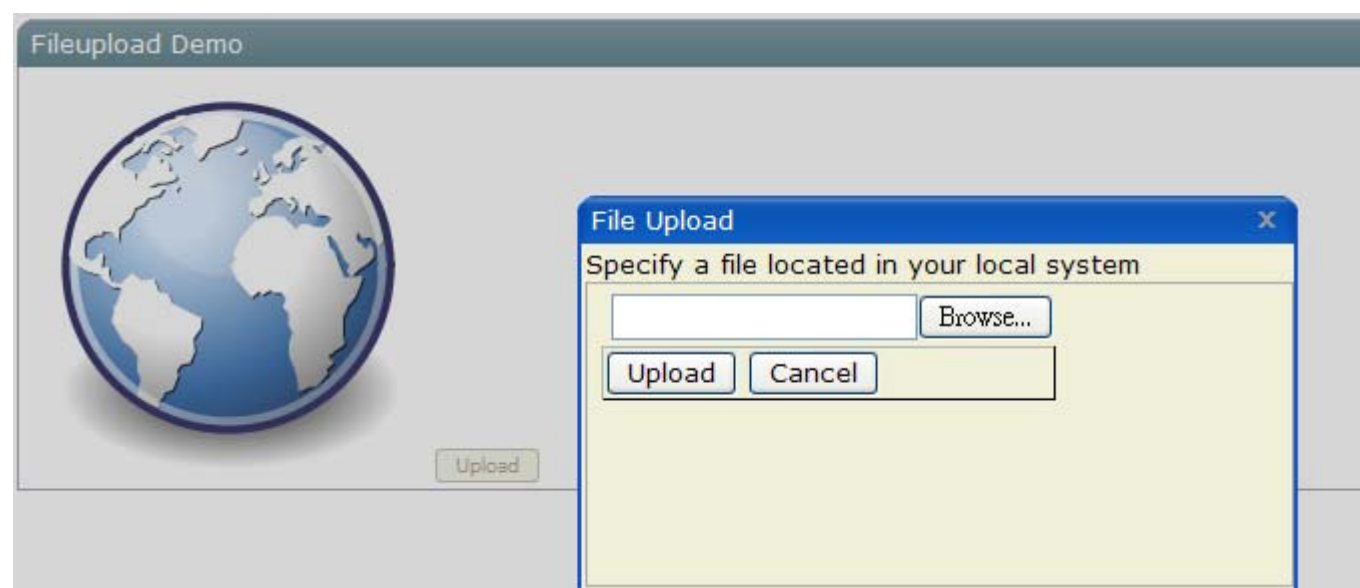
文件上传对话框

org.zkoss.zul.Fileupload 类提供了一套功能用以帮助用户向服务器上传文件。一旦调用了 `get` 方法，浏览器端会显示一个文件上传对话框来促使用户指定要上传的文件。直到用户已经上传了文件或点击了放弃按钮其才会返回。

```

<window title="Fileupload Demo" border="normal">
  <image id="image"/>
  <button label="Upload">
    <attribute name="onClick">{
      Object media = Fileupload.get();
      if (media instanceof org.zkoss.image.Image)
        image.setContent(media);
      else if (media != null)
        MessageBox.show("Not an image: "+media, "Error",
          MessageBox.OK, MessageBox.ERROR);
    }</attribute>
  </button>
</window>

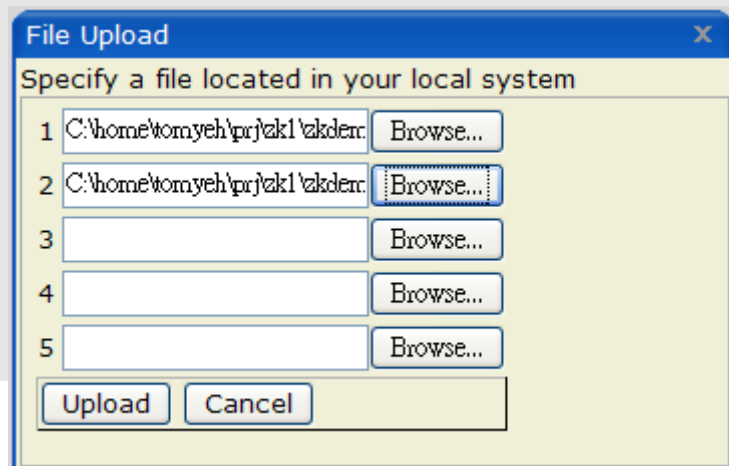
```



一次上传多个文件

如果你允许一次上传多个文件，可以按如下方式指定允许数字的最大值。

```
<window title="fileupload demo" border="normal">
  <button label="Upload">
    <attribute name="onClick"><![CDATA[{
      Object media = Fileupload.get(5);
      if (media != null)
        for (int j = 0; j < media.length; ++j) {
          if (media[j] instanceof org.zkoss.image.Image) {
            Image image = new Image();
            image.setContent(media[j]);
            image.setParent(pics);
          } else if (media[j] != null) {
            MessageBox.show("Not an image: "+media[j], "Error",
              MessageBox.OK,
              MessageBox.ERROR);
          }
        }
      }]]></attribute>
    </button>
    < vbox id="pics"/>
  </window>
```



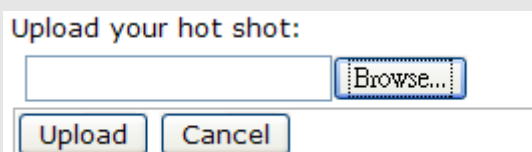
fileupload组件

fileupload 不是一个 modal 对话框。它是一个组件，所以 fileupload 可以和其它组件一起插入文字之间。

注:除了静态的 get 方法用于打开文件上传对话框，org.zkoss.zul.Fileupload 本身即为一个组件。即所谓的 fileuplod 组件。

例如，

```
<image id="img"/>
Upload your hot shot:
<fileupload
  onUpload="img.setContent(event.
media)"/>
```



onUpload事件

当按下上传按钮后，onUpload 事件及 `org.zkoss.zk.ui.event.UploadEvent` 事件的一个实例被送出。你可以使用 `getMedia` 或 `getMedias` 方法获取上传文件的内容。

注意 `getMedia` 和 `getMedias` 方法返回 `null` 即表示没有文件被指定但上传按钮被按下了。

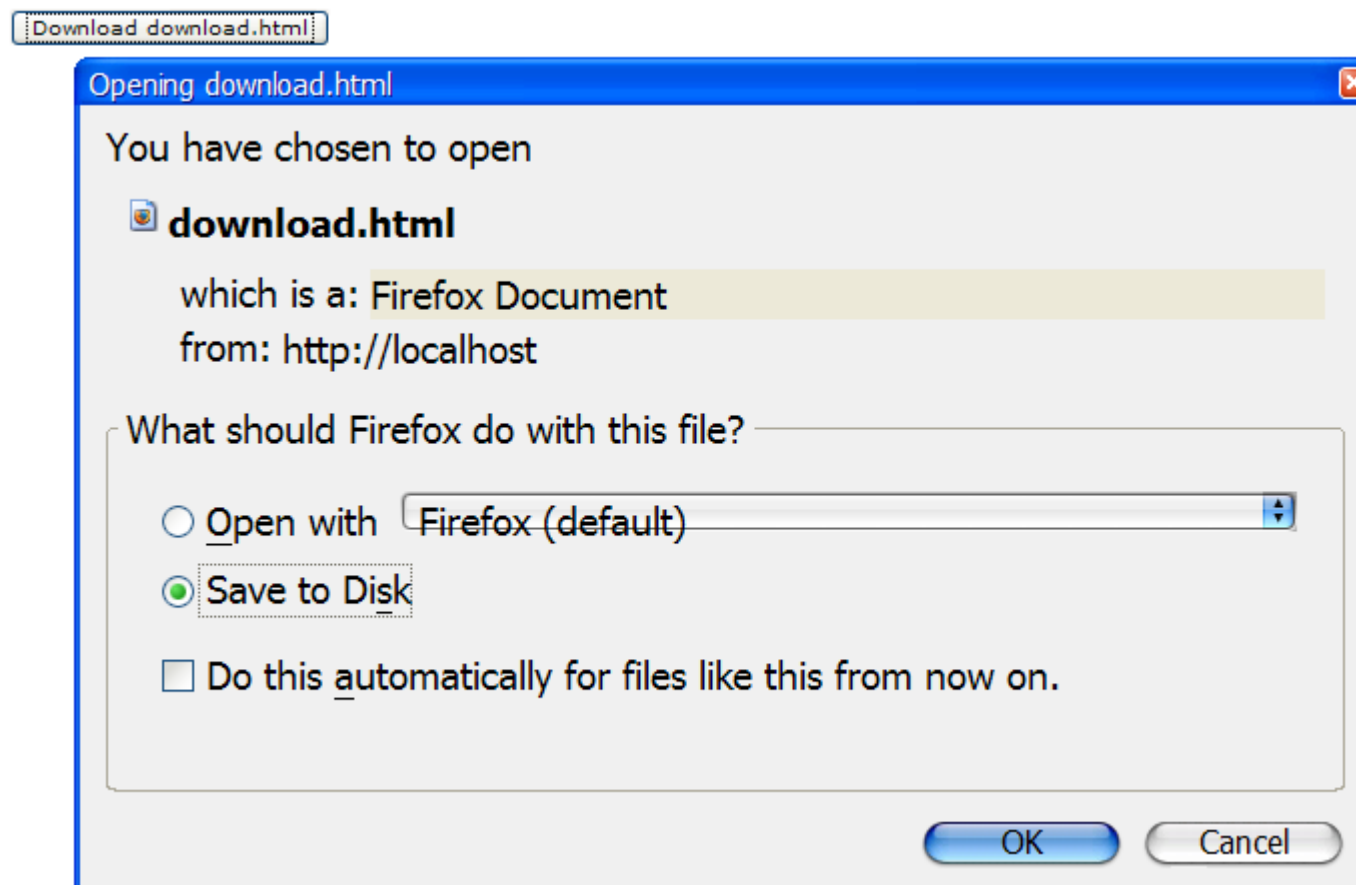
onClose事件

除了 onUpload，onClose 事件也被送出以通知上传按钮还是放弃按钮被按下。默认情况下，如果监听了此事件来实现定制行为(have the custom behavior)，fileupload 组件会失效，也就是说，所有的域(field)会被清空或重设(redraw)。

文件下载对话框

`org.zkoss.zul.Filedownload` 类提供了一套功能用以帮助用户从服务器下载文件。不同于 `iframe` 组件在浏览器窗口显示文件，如果其中的一个 `save` 方法被调用，则文件下载对话框会显示在浏览器端。然后，用户可以指定在本地文件系统中的存储路径。

```
<button label="Download download.html">
  <attribute name="onClick">{
    java.io.InputStream is =
desktop.getWebApp().getResourceAsStream("/test/download.html")
;
    if (is != null)
      Filedownload.save(is, "text/html", "download.html");
    else
      alert("/test/download.html not found");
  }</attribute></button>
```

^[37] wc 为 window 内容, 而 wt 为 window title.

^[38] 参考组件活动周期(Component Lifecycle) 一章.

^[39] 假定使用Tomcat.

布局组件

组件: `borderlayout`, `north`, `south`, `center`, `west`, `east`

布局组件是嵌套组件。父组件为 `borderlayout`, 子组件包括 `north`, `south`, `center`, `west`, 和 `east`。`borderlayout` 子组件的组合是任意的。例如, 你想将某一区域分成三个区域(竖直的), 则可以试试下面的组合,

```
<borderlayout height="500px">
  <east>
```

```

    The East
</east>
<center>
    The Center
</center>
<west>
    The West
</west>
</borderlayout>

```

或者你可以将区域水平的分为三部分，如下，

```

<borderlayour height="500px">
    <north>
        The North
    </north>
    <center>
        The Center
    </center>
    <south>
        The South
    </south>
</borderlayour>

```

并且，你可以根据需求在这些区域内嵌入任何 ZK 组件。

嵌套的borderlayout组件

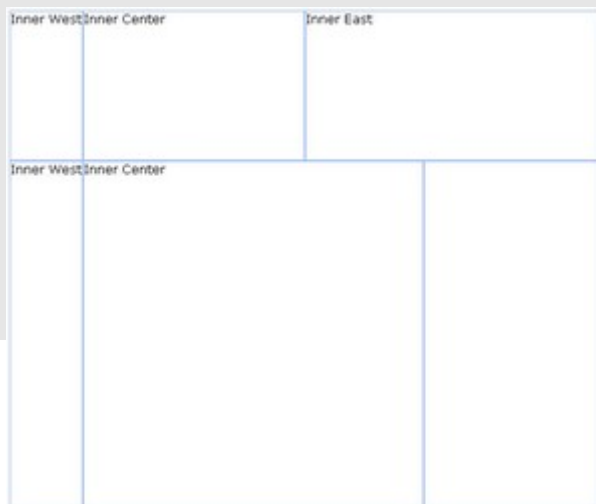
此外，你可将布局组件嵌入到另一个布局组件中，以划分更多的区域，如下，

```

<borderlayout height="500px">
    <north size="30%">
        <borderlayout height="250px">
            <west border="normal">

                Inner West
            </west>
            <center>
                Inner Center
            </center>
            <east size="50%"
border="normal">
                Inner East
            </east>

```



```

        </borderlayout>
    </north>
    <center border="normal">
        <borderlayout>
            <west border="normal">
                Inner West
            </west>
            <center border="normal">
                Inner Center
            </center>
            <east size="30%" border="normal">
            </east>
        </borderlayout>
    </center>
</borderlayout>

```

size 和 border属性

你可以为下列的自组件(north,south,east,west)指定 size 属性以决定其大小。但是，size 属性的功能依赖于子组件的类型(竖直的或水平的)。对于水平组件(north, 和 south), size 属性决定了它们的高度。而对于竖直组件， size 属性决定了它们的宽度

border 属性决定是否为这些布局组件设置边框，包括 borderlayout 的所有子组件。下面的表格指出了 border 属性的值。

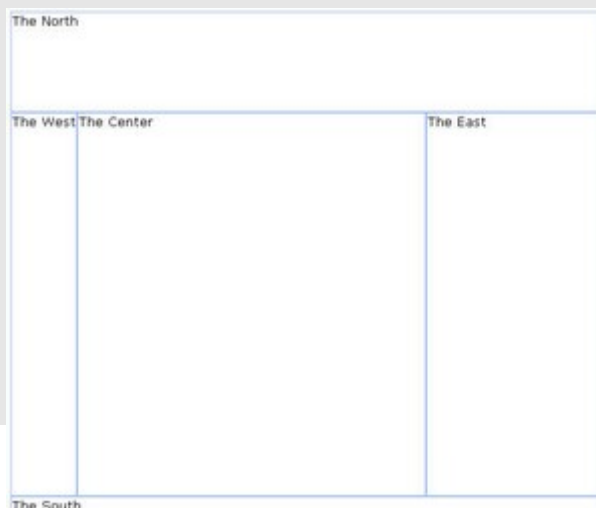
值	描述
none (default)	无边框
normal	有边框

这里有一个例子。

```

<borderlayout height="500px">
    <north size="30%"
border="normal">
        The North
    </north>
    <east size="30%"
border="normal">
        The East
    </east>
    <center border="normal">
        The Center

```



```

</center>
<west border="normal">
    The West
</west>
<south border="normal">
    The South
</south>
</borderlayout>

```

splittable 和 collapsible 属性

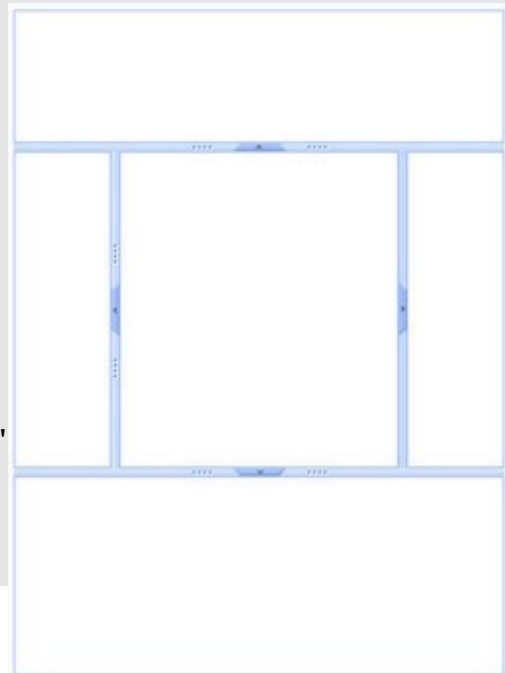
若你想使你的布局组件可拆分(splittable)，则可以将 splittable 属性设置为 true。

此外，若你想使一个组件可折叠(collapsible)，则可以将 collapsible 属性设置为 true。

```

<borderlayout height="500px">
    <north size="20%"
splittable="true"
collapsible="true"/>
    <east size="20%"
splittable="true"
collapsible="true"/>
    <center border="normal"/>
    <west size="20%"
splittable="true"
collapsible="true"/>
    <south size="30%" border="normal"
splittable="true"
collapsible="true"/>
</borderlayout>

```



maxsize和minisize 属性

当你将一个组件设置为 可拆分时，则 maxsize 和 minisize 属性会决定此组件的变动范围(re-resizing range)。

```

<north splittable="true" maxsize="500" minisize="200"/>

```

flex 属性

若留浏览器的大小改变了，则布局组件会自动调整自身的尺寸来适合浏览器的尺寸。若你想那些嵌入这些布局组件的 ZK 组件也可以自定调整自身大小，则可以将布局组件的 flex 属性设置为 true。

open 属性

为了获知一个布局组件是否已被折叠，则可以检查属性的值(也就是 isOpen 方法)。若想在程序中打开或折叠布局组件，则可以设置 open 属性的值(也就是，setOpen 方法)。

onOpen 属性

当用户打开或折叠一个布局组件时，则 onOpen 事件会被送至应用程序。

箱式模型

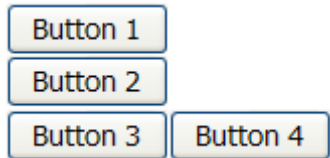
组件： vbox ， hbox 和 box 。

XUL 的箱式模型用于将显示部分分割成一系列的 box。Box 内的组件需要将它们定位成水平或垂直的。通过一系列的 box 及 separator，可以控制视觉表现的布局(visual representation)。

box 可以将其子组件布局成两种方位，水平的或垂直的。水平 box 可以将其子组件排成一条线，而垂直 box 可以将其子组件定位成垂直方向。你可以将其想象成 HTML 表格的行或里列。

下面是一些例子。

```
<zk>
  <vbox>
    <button label="Button 1"/>
    <button label="Button 2"/>
  </vbox>
  <hbox>
    <button label="Button 3"/>
    <button label="Button 4"/>
  </hbox>
</zk>
```



hbox 组件用于创建水平 **box**。放置在 hbox 内的组件会被水平排成一行。vbox 组件用于创建垂直 **box**。添加于其内的组件会被垂直的排成一行。

也有一个一般的 **box** 组件，默认水平布局，这意味着等价于 hbox。但是，可以使用 `orient` 属性用于控制 **box** 的布局方位。你可以将其赋值为 `horizontal` 以创建一个水平 **box**，或 `vertical` 以创建一个垂直 **box**。

因此下面的两行语句是等价的：

```
<vbox>
<box orient="vertical">
```

在 **box** 内可以添加任意多的组件，包括其它的 **box**。例如水平 **box**，每一个新添的组件都会被放置在前一个的右边。组件是不会包裹(`wrap`)的，所以添加多少组件，**window** 就会有多宽。相似的，每个垂直 **box** 中的组件会被置于前一个组件的正下方。

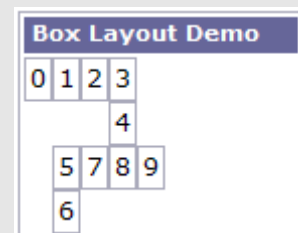
spacing 属性

box 控件组件之间的间隙是可以控制的。例如，下面的例子在上边缘与下边缘均放置了 5em 的间隙。注意：两个输入域的总间隙为 10em。

```
<vbox spacing="5em">
  <textbox/>
  <datebox/>
</vbox>
```

另一个例子是使用无间隙(`zero spacing`)的有趣布局。

```
<window title="Box Layout Demo" border="normal">
  <hbox spacing="0">
    <window border="normal">0</window>
    <vbox spacing="0">
      <hbox spacing="0">
        <window border="normal">1</window>
        <window border="normal">2</window>
        <vbox spacing="0">
          <window border="normal">3</window>
          <window border="normal">4</window>
        </vbox>
      </hbox>
    </vbox>
  </hbox>
  <hbox spacing="0">
    <vbox spacing="0">
      <window border="normal">5</window>
      <window border="normal">6</window>
    </vbox>
  </hbox>
</window>
```



```
        </vbox>
        <window border="normal">7</window>
        <window border="normal">8</window>
        <window border="normal">9</window>
    </hbox>
</vbox>
</hbox>
</window>
```

widths 和 heights 属性

使用 widths 属性可以指定 hbox 每个元素(cell)的宽度, 如下。

```
<hbox width="100%" widths="10%,20%,30%,40%">
    <label value="10%" />
    <label value="20%" />
    <label value="30%" />
    <label value="40%" /></hbox>
```

其值为宽度值的列表, 以逗号分隔。若未指定其值, 则不会为相应的元素产生宽度, 同时真实的宽度值被送至浏览器。

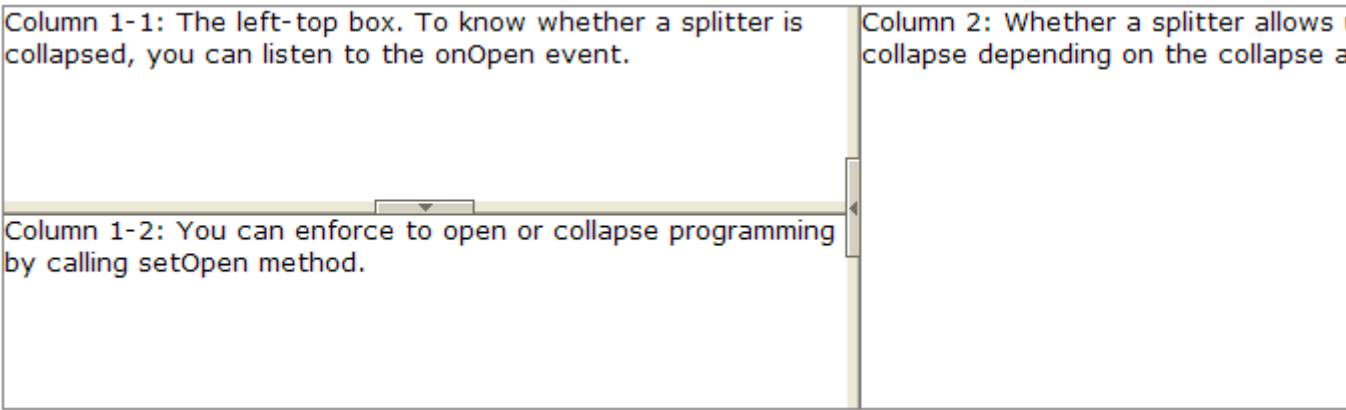
类似的, 你可以使用 heights 属性指定 vbox 每个元素的高度。实际上, 由于 box 的方向是水平或垂直的依赖于 orient 属性, 所以这两个属性是相同的。

分割器

组件: `splitter`。

或许有些时候你想把一个 window 分成两个部分, 这样用户可以改变这些部分的大小。可以使用 `splitter` 组件来实现此功能。此组件在两部分间创建一个细小的 bar, 可以允许用户修改任一边的大小。

`splitter` 必须放置在 box 内。当 `splitter` 放置在水平 box(hbox)内时, 它将会允许水平的改变大小。当 `splitter` 放置在垂直 box(vbox)内时, 它将会允许垂直的改变大小。例如,



[注]: 若你想使用原始的“os” CSS, 则可以将其 `splitter` 类名指定为“`splitter-os`”。

代码如下:

```
<hbox spacing="0" style="border: 1px solid grey" width="100%">
  <vbox height="200px">
    Column 1-1: The left-top box. To know whether a splitter is
    collapsed, you can listen to the onOpen event.
    <splitter collapse="after"/>
    Column 1-2: You can enforce to open or collapse programming
    by calling setOpen method.
  </vbox>
  <splitter collapse="before"/>
  Column 2: Whether a splitter allows users to open or collapse
  depending on the collapse attribute.
</hbox>
```

collapse属性

指定当 `splitter` 的按键(亦作,按钮(button))被点击时哪边被折叠。若未指定此属性, `splitter` 是不会引起折叠的(按键不会出现)。

允许值及它们的意义如下。

值	描述
none	无折叠发生。
before	当按下按键时, 相同组件内的元素立即在分割器(splitter)前折叠起来, 这样其宽度或高度将变为 0。
after	当按下按键时, 相同组件内的元素立即在分割器(splitter)后折叠起来, 这样其宽度或高度将变为 0。

open属性

为了知道 **splitter** 是否折叠，你可以检查 **open** 属性的值(也就是 **isOpen** 方法)。为了在程序中打开或折叠，你可以设置 **open** 属性的值(也就是 **setOpen** 方法)。

onOpen事件

当一个用户打开或折叠 **splitter** 时，**onOpen** 事件会被送至应用程序。

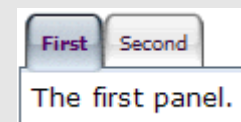
Tab箱

组件: **tabbox** , **tabs** , **tab** , **tabpanel** 和 **tabpanel** 。

Tab box 允许开发人员将大量组件分别置于几个组内，且每次只显示一组，这样用户界面不会太难于阅读。在同一时间内仅会显示一组(亦作，面板)。一旦一个隐藏组的 **tab** 被点击后，此组的组件变得可见而前一个可见组变得不可见。

tab box 的通用语法如下。

```
<tabbox>
  <tabs>
    <tab label="First"/>
    <tab label="Second"/>
  </tabs>
  <tabpanel>
    <tabpanel>The first panel.</tabpanel>
    <tabpanel>The second panel</tabpanel>
  </tabpanel>
</tabbox>
```



- **tabbox**: 外层的 **box** , 包括一系列 **tab** 和 **tab** 面板。
- **tabs**: **tab** 的容器, 也就是, **tab** 组件的集合。
- **tab**: 一个特定的 **tab**。点击 **tab** 可以将其 **tab** 面板显示在前。你可以将标签或图像置于其中。
- **tabpanel**: **tab** 面板的容器, 也就是, **tabpanel** 组件的集合。
- **tabpanel**: 单一 **tab** 面板的主体。你可以将一组组件放置于一个 **tab** 面板内。第一个 **tabpanel** 对应于第一个 **tab**, 第二个 **tabpanel** 对应于第二个 **tab**, 依次类推。

当前选中的 **tab** 组件由 `selected` 属性给出，将其设为 `true`。这样可以给当前选中 **tab** 不同的外观，这样可以容易看出被选中。在同一时间内，仅有一个 **tab** 的值为 `true`。

通过 Java 代码有两种方式可以改变选中的 **tab**。如下所示，它们是等价的。

```
tab1.setSelected(true);
tabbox.setSelectedTab(tab1);
```

当然，你可以直接将 `selected` 属性设为 `true`。

```
<tab label="My Tab" selected="true"/>
```

若没有 **tab** 被选中，则第一个自动被选中。

嵌套 **tab box**

tab 面板可以包含任何组件包括另一个 **tab box**。

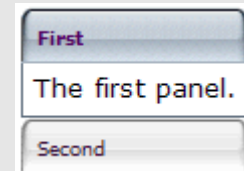
```
<tabbox>
  <tabs>
    <tab label="First"/>
    <tab label="Second"/>
  </tabs>
  <tabpanel>
    The first panel.
    <tabbox>
      <tabs>
        <tab label="Nested 1"/>
        <tab label="Nested 2"/>
        <tab label="Nested 3"/>
      </tabs>
      <tabpanel>
        <tabpanel>The first nested panel</tabpanel>
        <tabpanel>The second nested panel</tabpanel>
        <tabpanel>The third nested panel</tabpanel>
      </tabpanel>
    </tabbox>
  </tabpanel>
  <tabpanel>The second panel</tabpanel>
</tabpanels>
</tabbox>
```



The Accordion Tab Boxes

Tab box 支持两种模式(mold): default 和 accordion。accordion 模式的效果如下。

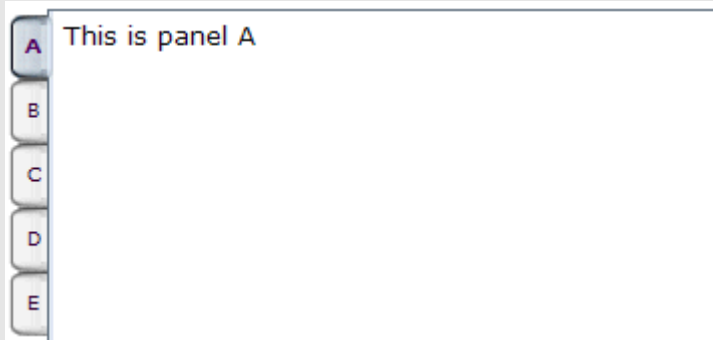
```
<tabbox mold="accordion">
  <tabs>
    <tab label="First"/>
    <tab label="Second"/>
  </tabs>
  <tabpanels>
    <tabpanel>The first panel.</tabpanel>
    <tabpanel>The second panel</tabpanel>
  </tabpanels>
</tabbox>
```



orient 属性

开发人员可以使用 orient 属性来控制 tab 的方位。默认为水平布局。你可以将其变为垂直的，效果如下。

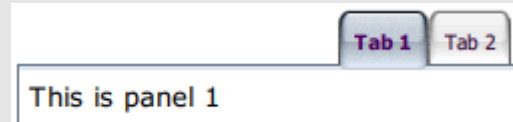
```
<tabbox width="400px" orient="vertical">
  <tabs>
    <tab label="A"/>
    <tab label="B"/>
    <tab label="C"/>
    <tab label="D"/>
    <tab label="E"/>
  </tabs>
  <tabpanels>
    <tabpanel>This is panel A</tabpanel>
    <tabpanel>This is panel B</tabpanel>
    <tabpanel>This is panel C</tabpanel>
    <tabpanel>This is panel D</tabpanel>
    <tabpanel>This is panel E</tabpanel>
  </tabpanels>
</tabbox>
```



Tabs的align属性

开发人员可以使用 **tabs** 的 **align** 属性控制 **tab** 的对齐方式。默认为 **start**(最左边或最上边)。你可以将其改为 **center** 或 **end**(最右边或最底部)，效果如下。

```
<tabbox width="250px">
  <tabs align="end">
    <tab label="Tab 1"/>
    <tab label="Tab 2"/>
  </tabs>
  <tabpanel>This is panel 1</tabpanel>
  <tabpanel>This is panel 2</tabpanel>
</tabbox>
```



closable属性

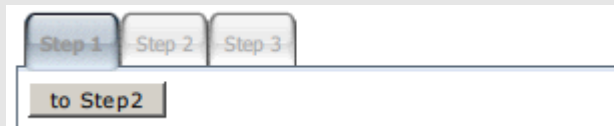
将 **closable** 属性设置为 **true**, **tab** 会显示关闭按钮, 这样用户可以通过点击按钮来关闭 **tab** 和相应的 **tab** 面板。一旦用户点击了 **close** 按钮, **onClose** 事件会被送至 **tab**。此事件会由 **Tab** 的 **onClose** 方法来处理。然后, 默认情况下, **onClose** 方法将 **tab** 本身及相应的 **tab** 面板移除。

参考 **window** 的 **closable** 属性。

disabled属性

通过将 **disabled** 属性设置为 **true**, 用户通过点击 **tab** 或 **close** 按钮不能选择或关闭相应的 **tab**。但是, 开发人员仍然可以编程控制 **tab** 的选择或关闭。

```
<tabbox width="300px" id="tbx">
  <tabs>
    <tab label="Step 1"
    id="tb1" disabled="true"/>
    <tab label="Step 2"
    id="tb2" disabled="true"/>
    <tab label="Step 3" id="tb3" disabled="true"/>
  </tabs>
  <tabpanel>
    <button label="to Step2"
    onClick="tbx.selectedTab=tb2"/>
  </tabpanel>
```



```

        <tabpanel><button label="to Step3"
onClick="tbx.selectedTab=tb3"/></tabpanel>
        <tabpanel>This is panel 3</tabpanel>
    </tabpanel>
</tabbox>

```

Tab面板的随机存取

就像其它组件，你可以仅在 **tab** 面板变为可见时加载其内容。最简单的方式是使用 `fulfill` 属性来推迟 **tab** 面板子组件的创建。

```

<tabbox>
  <tabs>
    <tab label="Preload" selected="true"/>
    <tab id="tab2" label="OnDemand"/>
  </tabs>
  <tabpanel>
    This panel is pre-loaded since no fulfill specified
  </tabpanel>
  <tabpanel fulfill="tab2.onSelect">
    This panel is loaded only tab2 receives the onSelect event
  </tabpanel>
</tabpanel>
</tabbox>

```

如果你更加喜欢手动创建子组件或自动操纵面板，你可以监听 `onSelect` 事件，然后当面板被选中时填充其内容。如下所示，

```

<tabbox id="tabbox" width="400" mold="accordion">
  <tabs>
    <tab label="Preload"/>
    <tab label="OnDemand" onSelect="load(self.linkedPanel)"/>
  </tabs>
  <tabpanel>
    This panel is pre-loaded.
  </tabpanel>
  <tabpanel>
  </tabpanel>
</tabpanel>
<zscript><![CDATA[
void load(Tabpanel panel) {
  if (panel != null && panel.getChildren().isEmpty())

```

```

        new Label("Second panel is loaded").setParent(panel);
    }
    ]]></zscript>
</tabbox>

```

网格

组件: `grid`, `columns`, `column`, `rows` 和 `row`。

`Grid` 包含排列整齐的组件就像表格一样。在 `grid` 内，你可以声明 `columns`，定义了 `header` 及 `column` 属性；还可以声明 `rows`，提供内容。

使用 `rows` 组件可以声明一套 `row`，即为 `grid` 元素的子组件。在 `rows` 内可以为每一行添加 `row` 组件。你可以在 `row` 元素内添加你想要的内容。每个子元素为指定行的一列。

类似的，`columns` 是由 `columns` 组件声明的，其将作为 `grid` 的一个子组件。不同于 `row` 用于保留每行的内容，`column` 声明了每列的通用属性，例如宽度和对齐方式，还有可选的 `headers`，也就是标签或图像。

```

<grid>
  <columns>
    <column
label="Type"/>
    <column
label="Content"/>
  </columns>
  <rows>
    <row>
      <label value="File:"/>
      <textbox width="99%"/>
    </row>
    <row>
      <label value="Type:"/>
      <hbox>
        <listbox rows="1" mold="select">
          <listitem label="Java Files, (*.java)"/>
          <listitem label="All Files, (*.*)" />
        </listbox>
        <button label="Browse..." />
      </hbox>
    </row>
  </rows>
</grid>

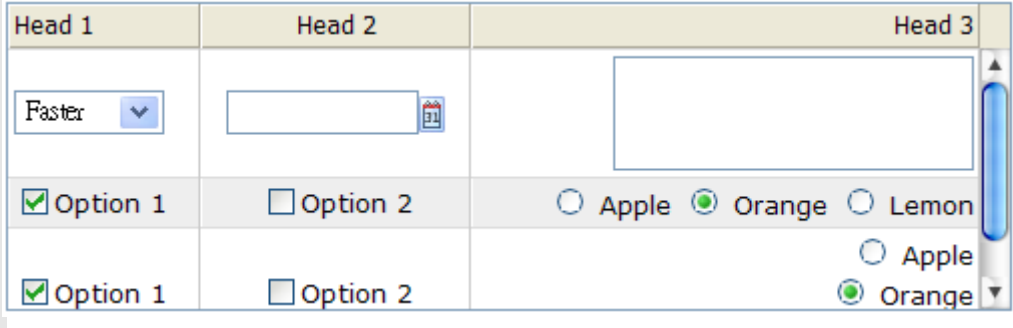
```

Type	Content
File:	<input type="text"/>
Type:	<div>Java Files (*.java) ▼ Browse...</div>

滚动网格

当指定了 height 属性且没有足够的空间来显示数据时，grid 会变为滚动的。

```
<grid width="500px" height="130px">
  <columns>
    <column label="Head 1">
      <listbox mold="select">
        <listitem label="Faster"/>
        <listitem label="Fast"/>
        <listitem label="Average"/>
      </listbox>
    </column>
    <column label="Head 2" align="center">
      <datebox/>
    </column>
    <column label="Head 3" align="right">
      <textbox rows="2"/>
    </column>
  </columns>
  <rows>
    <row>
      <checkbox checked="true" label="Option 1"/>
      <checkbox label="Option 2"/>
      <radiogroup>
        <radio label="Apple"/>
        <radio label="Orange" checked="true"/>
        <radio label="Lemon"/>
      </radiogroup>
    </row>
    <row>
      <checkbox checked="true" label="Option 1"/>
      <checkbox label="Option 2"/>
      <radiogroup orient="vertical">
        <radio label="Apple"/>
        <radio label="Orange" checked="true"/>
        <radio label="Lemon"/>
      </radiogroup>
    </row>
  </rows>
</grid>
```



```
</rows></grid>
```

可变列宽

如果你允许用户改变每列的宽度，可以将 columns 的 `sizable` 属性的设为 `true`。一旦允许用户进行此操作，用户可以通过拖动相邻列的边框来改变列宽，如下，

```
<window>
  <grid>
    <columns id="cs" sizable="true">
      <column label="AA"/>
      <column label="BB"/>
      <column label="CC"/>
    </columns>
    <rows>
      <row>
        <label value="AA01"/>
        <label value="BB01"/>
        <label value="CC01"/>
      </row>
      <row>
        <label value="AA01"/>
        <label value="BB01"/>
        <label value="CC01"/>
      </row>
      <row>
        <label value="AA01"/>
        <label value="BB01"/>
        <label value="CC01"/>
      </row>
    </rows>
  </grid>
  <checkbox label="sizeable" checked="true" onCheck="cs.sizeable
= self.checked"/>
</window>
```

onColSize事件

一旦用户改变了列宽，`onColSize` 事件及 `org.zkoss.zul.event.ColSizeEvent` 的一个实例会被送出。注意列宽是在 `onColSize` 事件被送出前被调整的。换言之，你可以忽略作为一个通知的事件。当然，你可以在事件监听器中做你想做的事。

分页网格

有两种方式在 **grid** 中处理较长的内容：滚动及分页。滚动可以由指定 `height` 属性来实现，就像在前面讨论的一样。分页可以通过将 `mold` 属性设为 `paging` 来实现的。一旦分页可用，**grid** 会将内容分为几页并且在同一时间内只显示一页。如下所示，

```
<grid width="300px" mold="paging" pageSize="4">
  <columns>
    <column label="Left"/>
    <column label="Right"/>
  </columns>
  <rows>
    <row>
      <label value="Item
1.1"/><label value="Item 1.2"/>
    </row>
    <row>
      <label value="Item 2.1"/><label value="Item 2.2"/>
    </row>
    <row>
      <label value="Item 3.1"/><label value="Item 3.2"/>
    </row>
    <row>
      <label value="Item 4.1"/><label value="Item 4.2"/>
    </row>
    <row>
      <label value="Item 5.1"/><label value="Item 5.2"/>
    </row>
    <row>
      <label value="Item 6.1"/><label value="Item 6.2"/>
    </row>
    <row>
      <label value="Item 7.1"/><label value="Item 7.2"/>
    </row>
  </rows>
</grid>
```

Left	Right
Item 1.1	Item 1.2
Item 2.1	Item 2.2
Item 3.1	Item 3.2
Item 4.1	Item 4.2
1 2 Next [1/7]	

一旦设置为分页模式，**grid** 将会创建 `paging` 组件的一个实例作为其子组件。然后它将会关注分页(It then takes care of paging for the grid it belongs to)。

pageSize属性

一旦设置为分页模式,你可以通过 pageSize 属性指定每次的可见行数(也就是页面大小, page size)。默认为 20。

paginal属性

如果你喜欢将 paging 组件置于不同的位置或你想使用同一个 paging 组件控制两个或更多的 grid, 可以明确指明 paginal 属性。注意: 如果没有明确指明, 即同于 paging 属性。

```
<vbox>
<paging id="pg" pageSize="4"/>
<hbox>
  <grid width="300px" mold="paging" paginal="{pg}">
    <columns>
      <column label="Left"/><column label="Right"/>
    </columns>
    <rows>
      <row>
        <label value="Item 1.1"/><label value="Item 1.2"/>
      </row>
      <row>
        <label value="Item 2.1"/><label value="Item 2.2"/>
      </row>
      <row>
        <label value="Item 3.1"/><label value="Item 3.2"/>
      </row>
      <row>
        <label value="Item 4.1"/><label value="Item 4.2"/>
      </row>
      <row>
        <label value="Item 5.1"/><label value="Item 5.2"/>
      </row>
      <row>
        <label value="Item 6.1"/><label value="Item 6.2"/>
      </row>
      <row>
        <label value="Item 7.1"/><label value="Item 7.2"/>
      </row>
    </rows>
  </grid>
  <grid width="300px" mold="paging" paginal="{pg}">
```

```

<columns>
  <column label="Left"/><column label="Right"/>
</columns>
<rows>
  <row>
    <label value="Item A.1"/><label value="Item A.2"/>
  </row>
  <row>
    <label value="Item B.1"/><label value="Item B.2"/>
  </row>
  <row>
    <label value="Item C.1"/><label value="Item C.2"/>
  </row>
  <row>
    <label value="Item D.1"/><label value="Item D.2"/>
  </row>
  <row>
    <label value="Item E.1"/><label value="Item E.2"/>
  </row>
  <row>
    <label value="Item F.1"/><label value="Item F.2"/>
  </row>
</rows>
</grid>
</hbox></vbox>

```

[Prev](#) [1](#) [2](#)

Left	Right
Item 5.1	Item 5.2
Item 6.1	Item 6.2
Item 7.1	Item 7.2

Left	Right
Item E.1	Item E.2
Item F.1	Item F.2

paging属性

其为一个用来呈现 paging 子组件(自动创建来处理分页)只读属性。如果你通过 paginal 属性指派了额外的分页，它的值将为空。你很少会访问到此属性。相反，使用 paginal 属性。

OnPaging事件及方法

一旦用户点击了 paging 组件的分页数字，onPaging 事件会被送至 grid。onPaging 方法会处理此事件。默情况下，此方法或使 rows 的内容无效，也就是刷新。

如果你想实现"(依要求创建)create-on-demand" 特性,你可以为 `grid` 的 `onPaging` 事件添加一个事件监听器。

```
grid.addEventListener(org.zkoss.zul.event.ZulEvents.ON_PAGING,
new MyListener());
```

排序

`grid` 支持直接的行排序。如果你想打开某一列的递增排序,可以为 `sortAscending` 属性指派一个 `java.util.Comparator` 实例。类似的,可以为 `sortDescending` 属性指派一个 `comparator` 来打开递减排序。

如下所示,首先得实现 `comparator` 接口来比较 `grid` 的任意两行,然后将其实例指派给 `sortAscending` 和 `sortDescending` 属性。注意: `compare` 方法由 `org.zkoss.zul.Row` 的实例调用。

```
<zk>
  <zscript>
    class MyRowComparator implements Comparator {
      public MyRowComparator(boolean ascending) {
        ...
      }
      public int compare(Object o1, Object o2) {
        Row r1 = (Row)o1, r2 = (Row)o2;
        ....
      }
    }
    Comparator asc = new MyRowComparator(true);
    Comparator dsc = new MyRowComparator(false);
  </zscript>
  <grid>
    <columns>
      <column sortAscending="${asc}" sortDescending="${dsc}"/>
    ...
```

sortDirection属性

`sortDirection` 属性用来控制是否在客户端显示一个图标,以显示特定列的排列顺序。如果每行元素在添加到 `grid` 前即已排好序,则需要明确设定这个属性。

```
<column sortDirection="ascending"/>
```

然后,只要你为相应的列指定比较器(`comparator`), `grid` 会自动维护此属性。

onSort事件

当你为某一列至少指定了一个比较器(**comparator**)，则若用户点击了此列 onSort 事件会被送至服务器。column 组件实现了监听器，基于指定的比较器(**comparator**)自动为行(**rows**)排序。

若你倾向于手工处理，可以为 onSort 事件将你自己的监听器添加到指定列。为了阻止默认的监听器调用 sort 方法，你必须调用 stopPropagation 方法来阻止事件被接收。另外，你可以重定义 sort 方法，见下文。

sort方法

sort 方法是 onSort 事件监听器的最底层实现。如果你使用 Java 代码为行排序，则这会非常有用。例如，你或许必须在添加行(假定未排序)后调用此方法。

```
Row row = new Row();
row.setParent(rows);
row.appendChild(...);
...
if (!"natural".column.getSortDirection())
    column.sort("ascending".equals(column.getSortDirection()));
```

默认排序算法为快速排序(**quick-sort**)(org.zkoss.zk.ui.Components 类的 sort 方法)。你可以使用自己的实现来重定义此方法。

注：sort 方法会检测排序顺序(通过调用 getSortDirection)。仅在排序顺序不同时 sort 才会为行排序。若想强制排序，可按如下方式，

```
column.setSortDirection("natural");
sort(myorder);
```

等价于：

```
sort(myorder, true);
```

实况数据

就像 listbox,grid 支持 live data 一样。使用实况数据，开发人员可以将数据从视图分离。换言之，开发人员仅需实现 rg.zkoss.zul.ListModel 接口类提供数据。而非直接操作 grid。好处有以下两点，

- 易于使用不同的视图来显示相同的数据。

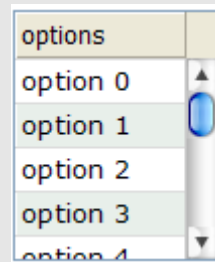
- **grid** 仅在其可见时才会将数据送至客户端。在数据量巨大时可以减少大量的网络流量(network traffic)。

使用实况数据需要经过三步，

1. 以 `ListModel` 形式准备好数据。ZK 有一个称为 `org.zkoss.zul.SimpleListModel` 的具体实现，用于显示一个数组对象。
2. 实现 `org.zkoss.zul.RowRenderer` 接口用于将数据行送至 **grid**。
 - 这是可选的。若为指定，默认的渲染器(renderer)会启，并将数据送至第一列。
 - 你可以实现不同的渲染器(renderer)，这样可以在不同的视图中显示相同的数据。
3. 在 `model` 属性中指定数据，并且可以选择在 `rowRenderer` 属性指定渲染器(renderer)。

在下面的例子中，我们准备了一个称为 `strset` 的列表模型，通过 **grid** 的 `model` 属性为其指定此值。然后，**grid** 会处理剩下的工作。

```
<window title="Live Grid" border="normal">
  <zscript>
    String[] data = new String[30];
    for(int j=0; j < data.length; ++j) {
      data[j] = "option "+j;
    }
    ListModel strset = new SimpleListModel(data);
  </zscript>
  <grid width="100px" height="100px"
model="${strset}">
    <columns>
      <column label="options"/>
    </columns>
  </grid></window>
```



实况数据的排序

如果你允许用户为提供实况数据的 **grid** 排序，则必须要实现 `org.zkoss.zul.ListModel` 及 `org.zkoss.zul.ListModelExt` 接口。

```
class MyListModel implements ListModel, ListModelExt {
  public void sort(Comparator cmpr, boolean ascending) {
    //do the real sorting
    //notify the grid (or listbox) that data is changed by use
    of ListDataEvent
```

```
}  
}
```

当用户请求 **grid** 排序时, **grid** 将会调用的用 `ListModelExt` 的 `sort` 方法来为数据排序。换言之, 排序 是由列表模型(**list mode**)完成的, 而不是 **grid**。

排序完成之后, 列表模型会调用 `org.zkoss.zul.event.ListDataListener` 实例(通过 `addListDataListener` 方法注册在 **grid**)的 `onChange` 方法来通知 **grid**。在大多数情况下, 所有的数据通常会变化, 所以列表模型通常会发送下列事件:

```
new ListDataEvent(this, ListDataEvent.CONTENTS_CHANGED, -1, -1)
```

辅助表头

除了 `columns`, 你可以使用 `auxhead` 和 `auxheader` 组件指定辅助表头, 如下。

```
<grid>  
  <auxhead>  
    <auxheader label="H1'07" colspan="6"/>  
    <auxheader label="H2'07" colspan="6"/>  
  </auxhead>  
  <auxhead>  
    <auxheader label="Q1" colspan="3"/>  
    <auxheader label="Q2" colspan="3"/>  
    <auxheader label="Q3" colspan="3"/>  
    <auxheader label="Q4" colspan="3"/>  
  </auxhead>  
  <columns>  
    <column label="Jan"/><column label="Feb"/><column  
label="Mar"/>  
    <column label="Apr"/><column label="May"/><column  
label="Jun"/>  
    <column label="Jul"/><column label="Aug"/><column  
label="Sep"/>  
    <column label="Oct"/><column label="Nov"/><column  
label="Dec"/>  
  </columns>  
  <rows>  
    <row>  
      <label value="1,000"/><label value="1,100"/><label  
value="1,200"/>  
      <label value="1,300"/><label value="1,400"/><label  
value="1,500"/>
```

```

        <label value="1,600"/><label value="1,700"/><label
value="1,800"/>
        <label value="1,900"/><label value="2,000"/><label
value="2,100"/>
    </row>
</rows>
</grid>

```

H1'07						H2'07					
Q1			Q2			Q3			Q4		
Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
1,000	1,100	1,200	1,300	1,400	1,500	1,600	1,700	1,800	1,900	2,000	2,100

辅助表头支持 `colspan` 和 `rowspan` 属性，而列表头(column header)并不支持。但是，就像名字暗示的那样，辅助表头必须和 `column` 一起使用。

不同于 `column/columns` 仅可以为 `grid` 所使用，`auhead/auxheader` 可以被用于 `grid`, `listbox` 和 `tree`。

特殊属性

spans属性

为一个整数列表，以逗号分隔，用于就控制是否将一个元素(cell)跨越几列。列表中的第一个数字表示第一个元素跨越的列数，第二个数字表示第二个元素跨越的列数，依次类推。若省略数字，则为 1。

例如，

```

<grid>
  <columns>
    <column label="Left" align="left"/><column label="Center"
align="center"/>
    <column label="Right" align="right"/><column label="Column
4"/>
    <column label="Column 5"/><column label="Column 6"/>
  </columns>
  <rows>
    <row>
      <label value="Item A.1"/><label value="Item A.2"/>
      <label value="Item A.3"/><label value="Item A.4"/>
      <label value="Item A.5"/><label value="Item A.6"/>
    </row>
  </rows>
</grid>

```



```

</row>
<row spans="1,2,2">
  <label value="Item B.1"/><label value="Item B.2"/>
  <label value="Item B.4"/><label value="Item B.6"/>
</row>
<row spans="3">
  <label value="Item C.1"/><label value="Item C.4"/>
  <label value="Item C.5"/><label value="Item C.6"/>
</row>
<row spans=",,2,2">
  <label value="Item D.1"/><label value="Item D.2"/>
  <label value="Item D.3"/><label value="Item D.5"/>
</row>
</rows>
</grid>

```

Left	Center	Right	Column 4	Column 5	Column 6
Item A.1	Item A.2	Item A.3	Item A.4	Item A.5	Item A.6
Item B.1	Item B.2		Item B.4	Item B.5	Item B.6
Item C.1	Item C.2	Item C.3	Item C.4	Item C.5	Item C.6
Item D.1	Item D.2	Item D.3		Item D.5	Item D.6

更多的布局组件

Separators 和空格

组件: `separator` 和 `space` 。

`separator` 用于在两个组件间插入一定的空间。有几种方式来定制 `separator` 。

1. 使用 `orient` 属性，你可以指定一个垂直或水平的 `separator`。默认为水平 `separator`，即插入一条水平线。而垂直的 `separator` 为插入一个空格。另外，`space` 是默认为垂直方向的 `separator` 的一种变体型(variant)。
2. 使用 `bar` 属性，你可以控制在组件间显示水平线还是垂直线。
3. 使用 `spacing` 属性，你可以控制 `spacing` 的大小。

```

<window>
  line 1 by separator
  <separator/>
  line 2 by separator
  <separator/>
  line 3 by separator | another piece
  line 4 by separator | another piece

```

```
line 3 by separator<space bar="true"/>another piece
<separator spacing="20px"/>
line 4 by separator<space bar="true" spacing="20px"/>another
piece
</window>
```

Group boxes

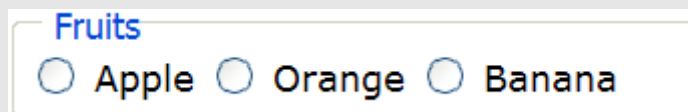
组件: groupbox 。

group box 用于将组件分组。典型情况下,被 group box 分组的组件周围会显示边框,以显示它们是相关联的。

使用 caption 组件可以创建 group box 的顶部标签。就像 HTML 的图例(legend)元素一样工作。

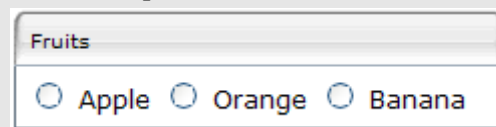
不同于 windows, group box 并不是一个 ID 空间的所有者。group box 并不能重叠或弹出。

```
<groupbox width="250px">
  <caption
label="Fruits"/>
  <radiogroup>
    <radio label="Apple"/>
    <radio label="Orange"/>
    <radio label="Banana"/>
  </radiogroup>
</groupbox>
```



除了 default 模型(mold), group box 还支持 3d 模型。如果使用了 3d 模型,group box 就像简单的 tab box 一样工作。首先,你可以使用 open 属性来控制是否显示其内容。类似的,你可以在接收 onOpen 事件时创建 group box 的内容。

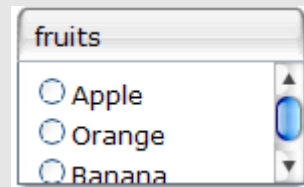
```
<groupbox mold="3d" open="true" width="250px">
  <caption label="fruits"/>
  <radiogroup>
    <radio label="Apple"/>
    <radio label="Orange"/>
    <radio label="Banana"/>
  </radiogroup>
</groupbox>
```



contentStyle属性及滚动Groupbox

contentStyle 属性用于为 groupbox 的内容块指定 CSS 样式。因此，你可以使用 overflow:auto (或 overflow:scroll)来将 groupbox 设置为滚动样式，如下，

```
<groupbox mold="3d" width="150px"
contentStyle="height:50px;overflow:auto">
  <caption label="fruits"/>
  <radiogroup onCheck="fruit.value =
self.selectedItem.label" orient="vertical">
    <radio label="Apple"/>
    <radio label="Orange"/>
    <radio label="Banana"/>
  </radiogroup>
</groupbox>
```



注：若使用 default 模型则 contentStyle 属性会被忽略。

contentStyle 属性中指定的 height 为内容块的高度，不包括 caption。因此，如果 groupbox 被忽略(dismissed)(也就是内容块不可见)了，整个 groupbox 的高度会收缩到仅能包含 caption。另外，如果你为整个 groupbox 指定了高度(使用 height 属性)，当忽略 groupbox 时，仅仅是内容块消失而整个 groupbox 的高度是不会变的。


工具栏

组件: toolbar 和 toolbarbutton.

toolbar 用于放置一系列的按钮，如 toolbar 按钮。toolbar 按钮可以不在 toolbar 内使用，所以同样 toolbar 可以不使用工具按钮。但是，放置在 toolbar 内的工具按钮会改变外观。

toolbar 有两种布局方向: horizontal 和 vertical。它们可以控制如何放置按钮。

```
<toolbar>
  <toolbarbutton
label="button1"/>
  <toolbarbutton label="button2"/>
</toolbar>
```



菜单栏

组件: menubar , menupopup , menu , menuitem 和 menuseparator 。

菜单栏包括了菜单项目及子菜单项目的集合。子菜单栏包括菜单项目及其它子菜单项目的集合。因此，它们组成了一个树状菜单项目，这样用户可以选择执行。

一个使用菜单栏的例子如下，

```
<menubar>
  <menu label="File">
    <menupopup>
      <menuitem
label="New"/>
      <menuitem
label="Open"/>
      <menuseparator/>
      <menuitem label="Exit"/>
    </menupopup>
  </menu>
  <menu label="Help">
    <menupopup>
      <menuitem label="Index"/>
      <menu label="About">
        <menupopup>
          <menuitem label="About ZK"/>
          <menuitem label="About Potix"/>
        </menupopup>
      </menu>
    </menupopup>
  </menu>
</menubar>
```



- menubar: 菜单项目(menuitem)和菜单(menu)集合的最顶层容器。
- menu: 弹出(popup)菜单的容器，menu 定义了一个用于显示的标签(the label to be displayed at part of its parent)。当用户点击标签时，弹出菜单就会出现。
- menupopup: 一个菜单项目(menuitem)和菜单(menu)集合的容器。同时它为 menu 的一个子组件并且当 menu 的标签被点击时它就会出现。
- menuitem: menu 的私有命令(individual command)。可以被放置在 a menu bar 或弹出菜单内。
- menuseparator: 一个菜单内的分割栏。需要放置在弹出菜单内。

执行一个菜单命令

菜单命令是和菜单项目相关联的。有两种方式将一个命令关联到菜单：onClickListener 事件及 href 属性。如果添加一个事件监听器到菜单项目的 onClick 事件，当此项目被点击时监听器就会被调用。

```
<menuitem onClick="draft.save()" />
```

另外，你可以为 href 属性指定一个超链接，则当用户点击菜单项目时会转向指定的 URL。

```
<menuitem href="/edit"/>  
<menuitem href="http://zkl.sourceforge.net"/>
```

若事件监听器和 href 属性都被指定了，则都会执行。但是，当事件监听器在服务器开始执行时，浏览器或许已经将当前的 URL 转向了指定的链接。因此，所有由事件监听器产生的响应都会被忽略。

像复选框一样使用菜单项目

菜单项目可以像复选框一样使用。checked 属性表示此菜单项目是否被选中。若选中，一个选中图标就会出现在菜单项目的前面。

除了编程使用 checked 属性，你还可以将 autocheck 属性设为 true，这样当用户点击此菜单项目时 checked 属性就会被自动切换(toggled)。

```
<menuitem label="" autocheck="true"/>
```

autodrop 属性

默认情况下，弹出菜单会在被点击时打开。你可以将其改变为当鼠标移动到菜单上方时它自动打开。将 autodrop 属性设为 true 即可实现。

```
<menubar autodrop="true">  
...  
</menubar>
```

onOpen 事件

当 menupopup 将要出现(或要隐藏) 时，onOpen 事件会被送至 menupopup 用于通知。对于更复杂的应用程序，你可以推迟 menupopup 内容的创建或自动操纵内容，直到接收了 onOpen 事件。参考用户界面标记语言一章中随机存取一节。

更多的菜单特性

就像 box，你可以使用 orient 属性控制菜单的方向。默认方向为 horizontal。

就像其它的组件，你可以自动的改变菜单，包括属性和创建子菜单。参考 zkdemo 中 test 目录下的 menu.zul。

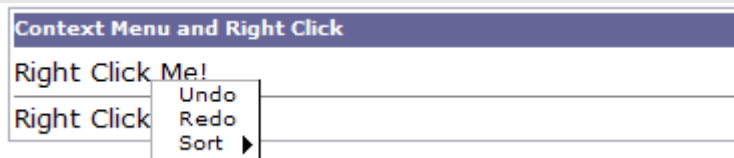
上下文菜单

组件: popup 和 menupopup。

你可以将 popup 或 menupopup 组件的 ID 指派给任一个 XUL 组件的 context 属性，这样，当用户右击此组件时 popup 或 menupopup 组件就会打开。

如下所示，通过将 ID 指派给 context 属性可以很容易的启动 context 菜单。当然，你可以为多个组件指派相同的 ID。

```
<label value="Right Click Me!" context="editPopup"/>
<separator bar="true"/><label value="Right Click Me!"
onRightClick="alert(self.value)"/>
<menupopup id="editPopup">
  <menuitem label="Undo"/>
  <menuitem label="Redo"/>
  <menu label="Sort">
    <menupopup>
      <menuitem label="Sort by Name" autocheck="true"/>
      <menuitem label="Sort by Date" autocheck="true"/>
    </menupopup>
  </menu>
</menupopup>
```



注意直到用户右击与其组件(通过 ID 关联)时 menupopup 才会变得可见。

[技巧]: 如果你只是想禁用浏览器默认的上下文菜单，可以为 context 属性指定一个并不存在的 ID。

popup 是一个比 menupopup 更通用的 popup 组件。你可以在 popup 内放置任意类型的组件。

```

<label value="Right Click Me!" context="any"/>

<popup id="any" width="300px">
  <vbox>
    It can be anything.
    <toolbarbutton label="ZK"
href="http://zk1.sourceforge.net"/>
  </vbox>
</popup>

```

定制的tooltip及弹出菜单

除了当用户右击组件时打开一个 **popup**，ZK 也可以其它情况下打开一个 **popup**。

属性	描述
context	当用户右键点击一个带有 context 属性的组件时，将会显示已指定 id 的 popup 或 menupopup 组件。
tooltip	当用户鼠标经过一个带有 tooltip 属性的组件时，将会显示已指定 id 的 popup 或 menupopup 组件。
popup	当用户点击一个带有 popup 属性的组件时，将会显示已指定 id 的 popup 或 menupopup 组件。

例如：

```

<window title="Context Menu and Right Click" border="normal"
width="360px">
  <label value="Move Mouse Over Me!" tooltip="editPopup"/>
  <separator bar="true"/>
  <label value="Tooptip for Another Popup" tooltip="any"/>
  <separator bar="true"/>
  <label value="Click Me!" popup="editPopup"/>
  <menupopup id="editPopup">
    <menuitem label="Undo"/>
    <menuitem label="Redo"/>
    <menu label="Sort">
      <menupopup>
        <menuitem label="Sort by Name" autocheck="true"/>
        <menuitem label="Sort by Date" autocheck="true"/>
      </menupopup>
    </menu>
  </menupopup>
  <popup id="any" width="300px">
    <vbox>

```

```

        ZK simply rich.
        <toolbarbutton label="ZK your killer Web application now!"
href="http://zk1.sourceforge.net"/>
    </vbox>
</popup>
</window>

```

注意，你可以在 `popup`, `tooltip` 和 `context` 属性内指定任意的标识，只要它们在同一页面。换句话说，它并不限于 `(confine)ID` 空间。

onOpen事件

当一个 `context menu`, `tooltip` 或 `popup` 将要显示(或隐藏)时，`onOpen` 事件会被送至 `ontext`, `tooltip` 或 `poup menu` 用于通知。事件为 `org.zkoss.zk.ui.event.OpenEvent` 类的一个实例，而你可以使用 `getReference` 方法来获取致使 `context menu`, `tooltip` 或 `popup` 出现的组件。

为了提高性能，你可以推迟内容的创建直到它们变得可见，也就是，直到接收 `onOpen` 事件。

推迟创建内容的最简单方式是使用 `fulfill` 属性，如下，

```

<popup id="any" width="300px" fulfill="onOpen">
    <button label="Hi"/><!-- whatever content -->
</popup>

```

然后。当加载页面时，内容(Hi 按钮)不会被创建。在接收到 `onOpen` 事件的第一时间，内容就会被创建。

若你喜欢在 `Java` 代码中动态的操纵内容，可以按如下描绘的方式监听 `onOpen` 事件。

```

<popup id="any" width="300px">
    <attribute name="onOpen">
        if (event.isOpen()) {
            if (self.getChildren().isEmpty()) {
                new Button("Hi").seParent(self);
                ...
            }
            if (event.getReference() instanceof Textbox) {
                //you can do component-dependent manipulation here
                ...
            }
        }
    </attribute>

```



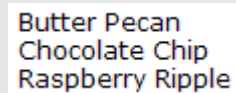
```
</popup>
```

列表框

组件: listboxlistitem , listcell , listhead 和 listheader 。

list box 用于显示列表中的若干项目。用户可以从列表选取某一项目。最简单的形式如下。这是一个单列单选择的列表框。

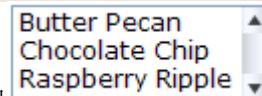
```
<listbox>
  <listitem label="Butter Pecan"/>
  <listitem label="Chocolate Chip"/>
  <listitem label="Raspberry Ripple"/>
</listbox>
```



Listbox 有两种模型: default 和 select。若使用了 select, 就会产生 HTML 的 SELECT 标签。

```
<listbox mold="select">...</listbox>
```

注意: 若为"select"模型, 则行数为"1", 且并没有项目被标明选中, 浏览器就像第一行被选中一样显示 listbox。最坏的情况事, 若用户选中了此事例中的首项目, 则不会发出 onSelect 事件。为了避免这种困惑, 开发人员至少要为 mold="select" 及 rows="1" 选择一个项目。



除了标签, 你可以使用 setValue 方法为每个项目指派一个特定应用程序值。

无鼠标输入 listbox

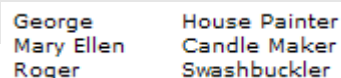
1. UP 和 DOWN, 上下移动选中的列表项目。
2. PgUp 和 PgDn , 以一页的步长上下移动选中项。
3. HOME , 选中首行, END , 选中末行。
4. Ctrl+UP 和 Ctrl+DOWN , 上下移动列表项目的聚焦但并不改变选中项目。
5. SPACE, 选中聚焦项目。

多列列表框

List box 也支持多列。当用户选中以一个项目时, 整行都会被选中。

为了指定一个多列列表, 你需要指定 listcell 组件作为每个 listitem(作为一行)的列。

```
<listbox width="200px">
  <listitem>
```



```

        <listcell label="George"/>
        <listcell label="House Painter"/>
    </listitem>
    <listitem>
        <listcell label="Mary Ellen"/>
        <listcell label="Candle Maker"/>
    </listitem>
    <listitem>
        <listcell label="Roger"/>
        <listcell label="Swashbuckler"/>
    </listitem>
</listbox>

```

栏头

通过使用 `listhead` 和 `listheader` 可以指定栏头，如下^[40]。除了标签，通过使用 `image` 属性你可以指定一张图像作为栏头。

```

<listbox width="200px">
    <listhead>
        <listheader label="Name"/>
        <listheader label="Occupation"/>
    </listhead>
    ...
</listbox>

```

Name	Occupation
George	House Painter
Mary Ellen	Candle Maker
Roger	Swashbuckler

栏尾

通过使用 `listfoot` 和 `listfooter`，你可以指定栏尾，如下。注意 `listhead` 和 `listfoot` 的顺序是不匹配的。每次 `listhead` 实例被添加到一个 `listbox` 时，它必须为第一个子组件，而 `listfooter` 实例为最后一个子组件。

```

<listbox width="200px">
    <listhead>
        <listheader label="Population"/>
        <listheader align="right"
label="%" />
    </listhead>
    <listitem id="a" value="A">
        <listcell label="A. Graduate"/>
        <listcell label="20%" />
    </listitem>
    <listitem id="b" value="B">

```

Population	%
A. Graduate	20%
B. College	23%
C. High School	40%
D. Others	17%
More or less	100%

```

        <listcell label="B. College"/>
        <listcell label="23%"/>
    </listitem>
    <listitem id="c" value="C">
        <listcell label="C. High School"/>
        <listcell label="40%"/>
    </listitem>
    <listitem id="d" value="D">
        <listcell label="D. Others"/>
        <listcell label="17%"/>
    </listitem>
    <listfoot>
        <listfooter label="More or less"/>
        <listfooter label="100%"/>
    </listfoot>
</listbox>

```

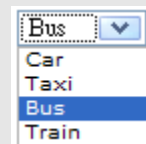
下拉列表

通过指定 `select` 模型及单行可以创建一个下拉列表。注意不能为下拉列表指定多列。

```

<listbox mold="select" rows="1">
    <listitem label="Car"/>
    <listitem label="Taxi"/>
    <listitem label="Bus" selected="true"/>
    <listitem label="Train"/>
</listbox>

```



多选

当用户点击一个列表项目时，这个项目会被选中，且 `onSelect` 事件会被送回服务器以通知应用程序。通过将 `multiple` 属性设置为 `true`，你可以控制一个 `listbox` 是否允许选中多行。默认为 `false`。

滚动列表框

若指定了 `rows` 或 `height` 属性且不足以显示所有项目时，`listbox` 会变为滚动的。

```

<listbox width="250px" rows="4">
    <listhead>
        <listheader label="Name"
sort="auto"/>

```

Name	Gender
Mary	FEMALE
John	MALE
Jane	FEMALE
Henry	MALE

```
<listheader label="Gender" sort="auto"/>
</listhead>
<listitem>
  <listcell label="Mary"/>
  <listcell label="FEMALE"/>
</listitem>
<listitem>
  <listcell label="John"/>
  <listcell label="MALE"/>
</listitem>
<listitem>
  <listcell label="Jane"/>
  <listcell label="FEMALE"/>
</listitem>
<listitem>
  <listcell label="Henry"/>
  <listcell label="MALE"/>
</listitem>
<listitem>
  <listcell label="Michelle"/>
  <listcell label="FEMALE"/>
</listitem>
</listbox>
```

rows属性

rows 属性用于控制显示多少行。若将其设置为 0，listbox 会将自身大小调整到容纳所有的项目。

可变列表头

就像 columns，你可以将 listhead 的 sizable 属性设为 true，以允许用户改变列表头的宽度。类似的当用户改变列宽时 onColSize 事件会被送出。

分页列表框

就像 grid，指定 paging 模型，你就可以使用多页来呈现 listbox 的较长内容。类似的，你可以控制每页显示多少项目，是否使用额外的 paging 组件，以及当选中某一页面时是否定制行为。参考网格一节获取细节。

排序

Listbox 直接支持列表项目的排序。有几种方式来启动某一列的排序。最简单的方式是将 listheader 的 sort 属性设为 auto，如下。然后，与 listheader 关联的列会基于指定列的每个列表元素的标签排序。

```
<zk>
  <listbox width="200px">
    <listhead>
      <listheader label="name"
sort="auto"/>
      <listheader label="gender"
sort="auto"/>
    </listhead>
    <listitem>
      <listcell label="Mary"/>
      <listcell label="FEMALE"/>
    </listitem>
    <listitem>
      <listcell label="John"/>
      <listcell label="MALE"/>
    </listitem>
    <listitem>
      <listcell label="Jane"/>
      <listcell label="FEMALE"/>
    </listitem>
    <listitem>
      <listcell label="Henry"/>
      <listcell label="MALE"/>
    </listitem>
  </listbox>
</zk>
```

name	gender
Henry	MALE
Jane	FEMALE
John	MALE
Mary	FEMALE

sortAscending 和 sortDescending 属性

若你想用不同的方式为列表项目排序，可以为 sortAscending 和/或 sortDescending 属性指派一个 java.util.Comparator 实例。一旦指定了，列表项目会使用你指派的比较器以升序或降序存储。

调用值为 auto 的 sort 属性实际上是自动为 sortAscending 和 sortDescending 指定了两个比较器。你可以重定义它们中的任意一个通过为其指派另一个比较器。

例如，假定你想基于列表项目的值排序，而不是列表元素的标签，可以按如下方式为这些属性指派以一个 `ListitemComparator` 的实例。

```
<zscript>
    Comparator asc = new ListitemComarator(-1, true, true);
    Comparator dsc = new ListitemComarator(-1, false, true);
</zscript>
<listbox>
    <listhead>
        <listheader sortAscending="${asc}"
sortDescending="${dsc}"/>
    ...
```

sortDirection属性

`sortDirection` 属性控制是否在客户端显示一个图标，以指明特定列的排列顺序。若列表项目在添加到 `listbox` 前已排好序，你需要明确设置这个属性。

```
<listheader sortDirection="ascending"/>
```

然后，只要你为相应的 `listheader` 指派了比较器 `listbox` 会自动维护。

onSort事件

当你为 `listheader` 至少指派了一个比较器时，若用户点击了它，`onSort` 事件就会被送至服务器。`listheader` 实现了一个监听器来自动处理排序。

若你喜欢受手动处理，可以将你的监听器添加到 `listheader` 的 `onSort` 事件。为了阻止默认的监听器调用 `sort` 方法，你必须调用 `stopPropagation` 方法来阻止接收事件。另外，你可以重定义 `sort` 方法，见下文。

sort方法

`sort` 方法是默认的 `onSort` 事件监听器的最底层实现。若你想使用 Java 代码为列表项目排序这也是很有用的。例如，你可以在添加项目(假定并未排好序)之后调用此方法。

```
new Listem("New Stuff").setParent(listbox);
if (!"natural".header.getSortDirection())
    header.sort("ascending".equals(header.getSortDirection()));
```

默认的排序算法为快速排序(使用 `org.zkoss.zk.ui.Components` 类的 `sort` 方法)。你可以将其用自己的实现来重定义，或像前一章节描述的那样监听 `onSort` 事件。

提示：为大量实况数据(live data)排序或许会显著的降低性能。最好是侦听(intercept) onSort 事件或 sort 方法以有效处理排序。参考下面的为实况数据排序一节。

特殊属性

checkmark属性

checkmark 属性控制是否在每个 listitem 前显示一个 checkbox 或 radio 按钮。

在下面的例子中，当你将一个 listitem 从左边的 listbox 移到右边的 listbox 时，会自动添加一个 checkbox。反过来 checkbox 会被移去。

```
<hbox>
  <listbox
id="src"
rows="0"
multiple="true"
width="200px">
  <listhead>
    <listheader label="Population"/>
    <listheader label="Percentage"/>
  </listhead>
  <listitem id="a" value="A">
    <listcell label="A. Graduate"/>
    <listcell label="20%"/>
  </listitem>
  <listitem id="b" value="B">
    <listcell label="B. College"/>
    <listcell label="23%"/>
  </listitem>
  <listitem id="c" value="C">
    <listcell label="C. High School"/>
    <listcell label="40%"/>
  </listitem>
  <listitem id="d" value="D">
    <listcell label="D. Others"/>
    <listcell label="17%"/>
  </listitem>
</listbox>
<vbox>
  <button label="=>" onClick="move(src, dst)"/>
  <button label="<=" onClick="move(dst, src)"/>
```

Population	Percentage
A. Graduate	20%
C. High School	40%

=>

<=

Population	Percentage
<input type="checkbox"/> E. Supermen	21%
<input checked="" type="checkbox"/> D. Others	17%
<input type="checkbox"/> B. College	23%

```

</vbox>
<listbox id="dst" checkmark="true" rows="0" multiple="true"
width="200px">
  <listhead>
    <listheader label="Population"/>
    <listheader label="Percentage"/>
  </listhead>
  <listitem id="e" value="E">
    <listcell label="E. Supermen"/>
    <listcell label="21%"/>
  </listitem>
</listbox>
<zscript>
void move(Listbox src, Listbox dst) {
  Listitem s = src.getSelectedItem();
  if (s == null)
    MessageBox.show("Select an item first");
  else
    s.setParent(dst);
}
</zscript>
</hbox>

```

注意若 `multiple` 属性为 `false`，则会显示 `radio` 按钮，如右图所示。

Population	Percentage
<input type="radio"/> A. Graduate	20%
<input checked="" type="radio"/> B. College	23%
<input type="radio"/> C. High School	40%
<input type="radio"/> D. Others	17%

vflex属性

`vflex` 属性控制是否在垂直方向增长或缩小以适合指定空间。即所谓的垂直柔性 (`vertical flexibility`)。例如，如果列表太长以至于不适合浏览器窗口，此属性会缩小列表的高度以控制整个列表在浏览器内可见。

若指定了 `rows` 属性则此属性会被忽略。

maxlength 属性

maxlength 属性定义了浏览器端可见字符的最大允许字节数。通过设置这个属性，你可以将 listbox 变窄。

实况数据

就像grid^[41]，listbox支持live data。使用了实况数据，开发人员可以将数据从视图分离。换句话说，开发人员仅需要实现org.zkoss.zul.ListModel接口提供数据，而不用直接操作listbox。好处有以下两点，

- 易于使用不同的视图来显示相同的数据。
- listbox 仅在其可见时才会将数据送至客户端。在数据量巨大时可以减少大量的网络流量(network traffic)。

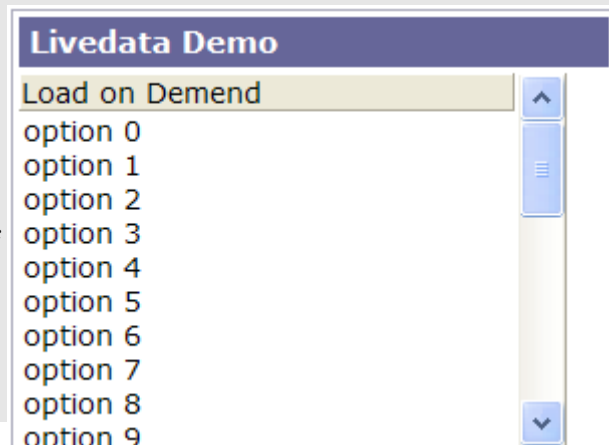
使用实况数据需要经过三步，

1. 以 ListModel 形式准备好数据。ZK 有一个称为 org.zkoss.zul.SimpleListModel 的具体实现，用于显示一个数组对象。
2. 实现 org.zkoss.zul.ListitemRenderer 接口用于将一个数据项目送至 listbox 的一个列白哦项目。
 - 这是可选的。若为指定，默认的渲染器(renderer)会启，并将数据送至第一列。
 - 你可以实现不同的渲染器(renderer)，这样可以在不同的视图中显示相同的数据。
3. 在 model 属性中指定数据，并且可以选择在 itemRenderer 属性指定渲染器(renderer)。

在下面的例子中,我们准备了一个 strset 列表模型(list model)，通过 model 属性将其指派给一个 listbox。然后，listbox 会处理余下的工作。

```
<window title="Livedata Demo" border="normal">
```

```
    <zscript>
        String[] data = new
String[30];
        for(int j=0; j <
data.length; ++j) {
            data[j] = "option "+j;
        }
        ListModel strset = new
SimpleListModel(data);
    </zscript>
```



```
<listbox width="200px" rows="10" model="${strset}">
  <listhead>
    <listheader label="Load on demend"/>
  </listhead>
</listbox>
</window>
```

为实况数据排序

若你允许用户为一个提供实况数据的 **listbox** 排序，你可以实现 `org.zkoss.zul.ListModel` 和 `org.zkoss.zul.ListModelExt` 接口。

```
class MyListModel implements ListModel, ListModelExt {
    public void sort(Comparator cmpr, boolean ascending) {
        //do the real sorting
        //notify the listbox (or grid) that data is changed by use
        of ListDataEvent
    }
}
```

当用户向 **listbox** 发出排序请求时，**listbox** 将会调用 `ListModelExt` 的 `sort` 方法为数据排序。换句话说，排序是由列表模型处理的，而不是 **listbox**。

排好序之后，列表模型会调用 `org.zkoss.zul.event.ListDataListener` 实例(通过 `addListDataListener` 方法注册到 **listbox**)的 `onChange` 方法通知 **listbox**。在大多数情况下，所有的数据通常会改变，所以列表模型通常发出下列事件：

```
new ListDataEvent(this, ListDataEvent.CONTENTES_CHANGED, -1,
-1)
```

注：`ListModel` 和 `ListModelExt` 的实现与视觉表现是独立的。换言之，它可以被用于 **grid** ,**listbox** 及其它支持 `ListModel` 的任意组件。

换言之，为了获得最大的灵活性，你应该假定不使用组件，而使用 `ListDataEvent` 通信。

包含按钮的列表框

理论上，**listcell** 可以包含任意的其它组件，如下所述。

```
<listbox width="250px">
  <listhead>
```

Population	Percentage
A. Graduate	20%
<input type="checkbox"/> B. College	<input type="text" value="23%"/>
C. High School	40%

```

    <listheader label="Population"/>
    <listheader label="Percentage"/>
</listhead>
<listitem value="A">
    <listcell><textbox value="A. Graduate"/></listcell>
    <listcell label="20%"/>
</listitem>
<listitem value="B">
    <listcell><checkbox label="B. College"/></listcell>
    <listcell><button label="23%"/></listcell>
</listitem>
<listitem value="C">
    <listcell label="C. High School"/>
    <listcell><textbox cols="8" value="40%"/></listcell>
</listitem></listbox>

```

注:

1. 若使用 **grid** 更好, 则不要使用 **listbox**。它们的外观类似, 但 **listbox** 应仅用于呈现可选项目的列表。
2. 若 **listbox** 包括可编辑的组件, 例如 **textbox** 和 **checkbox**, 则会引起用户的困惑。一个普遍的问题使, 用户在一个未选中的项目内输入文本(A common question is what the text, that a user entered in a unselected item, means).
3. 由于浏览器的限制, 用户不能从文本框(text box)内选择一段文字。

[40] 在使用 **listhead** 和 **listheader** 的地方, 此特性与 XUL 有一些不同。

[41] 此概念类似于 **Swing(javax.swing.ListModel)**。

树控件

组件: **tree**, **treechildren**, **treeitem**, **treerow**, **treecell**, **treecols** 和 **treecol**。

tree 包含了两部分: 一套专栏 (**column**), 及树体(**tree body**)。专栏由若干 **treecol** 组件定义, 一个对应一个专栏。每个专栏都会作为页眉(**header**)出现在树顶。第二部分, 树体, 包含了出现在树中的数据, 由 **treechildren** 组件创建。

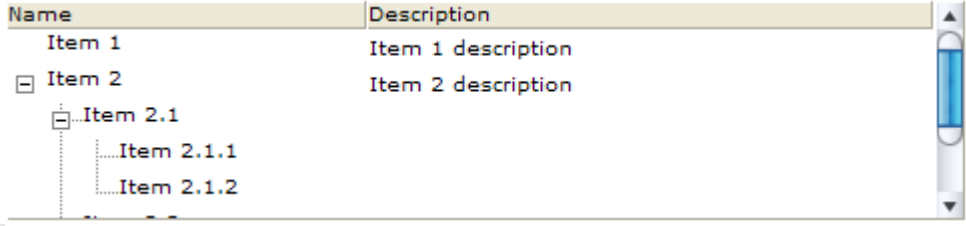
一个数控件的例子如下:

```
<tree id="tree" rows="5">
```

```

<treecols>
<treecol
label="Name" />
<treecol label="Description" />
</treecols>
<treechildren>
<treeitem>
<treerow>
<treecell label="Item 1" />
<treecell label="Item 1 description" />
</treerow>
</treeitem>
<treeitem>
<treerow>
<treecell label="Item 2" />
<treecell label="Item 2 description" />
</treerow>
<treechildren>
<treeitem>
<treerow>
<treecell label="Item 2.1" />
</treerow>
<treechildren>
<treeitem>
<treerow>
<treecell label="Item 2.1.1" />
</treerow>
</treeitem>
<treeitem>
<treerow>
<treecell label="Item 2.1.2" />
</treerow>
</treeitem>
</treechildren>
</treeitem>
</treechildren>

```



```
        </treeitem>
      </treechildren>
    </treeitem>
    <treeitem label="Item 3"/>
  </treechildren>
</tree>
```

- **tree**: 这是 **tree** 组件的外层。
- **treecols**: **treecol** 组件的容器。
- **treecol**: 用于声明 **tree** 的一列(column)。使用此组件, 你可以指定额外的信息, 如列的标头。
- **treechildren**: 包含了 **tree** 的主体, 此组件包含了一个 **treeitem** 的集合。
- **treeitem**: 此组件包含了一行数据(**treerow**), 及可选的 **treechildren**。
 - 若组件没有包含 **treechildren**, 则它为一个不接受任何子项目的叶子节点。
 - 若组件包含了 **treechildren**, 则它为一个可以包含其它项的分支节点。
 - 对于一个分支节点, 一个+/-按钮将会出现在此行的开始, 这样用户可以通过点击+/- 按钮来打开或关闭此项。
- **treerow**: **tree** 中的单独行, 需要放置在 **treeitem** 组件内。
- **treecell**: **treerow** 中的单独元素(cell) 。此元素将要放在 **treerow** 组件内。

无鼠标输入 tree

- UP 和 DOWN , 上下移动 **tree** 项目中的选中项。
- PgUp 和 PgDn , 以一页的步长上下移动选中项。
- HOME , 选中首行, END , 选中末行。
- RIGHT , 打开一个 **tree** 项目, LEFT 关闭一个 **tree** 项。
- Ctrl+UP 和 Ctrl+DOWN, 上下移动一个 **tree** 项的聚焦, 但并不改变选中项。
- SPACE 选中聚焦项。

open属性和onOpen事件

每一个 **tree** 项都有一个 **open** 属性来控制是否显示其子项目。默认值为 **true**。通过将此属性设置为 **false**, 你可以控制 **tree** 的哪部分不可见。

```
<treeitem open="false">
```

当用户点击+/-按钮时, 他会打开 **tree** 项并使其子项目可见。onOpen 事件会被发送至服务器通知应用程序。




对于更复杂的应用程序，你可以延迟 **tree** 项的创建或动态操作 **tree** 项的内容，直到接收了 `onOpen` 事件。参考 **ZK 用户标记语言** 一章中随机存取一节获取细节。


多选

当用户点击一个 **tree** 项时，整个项目会被选中且 `onSelect` 事件会被送回至服务器通知应用程序。通过将 `multiple` 属性设置为 `true`，你可以控制 **tree** 控件是否允许多选。默认值为 `false`。

分页




`pageSize` 属性控制一次显示的 **tree** 项目的个数。默认为 10。也就是在客户端每一层最多显示 10 个 **tree** 项目，如右图所示。

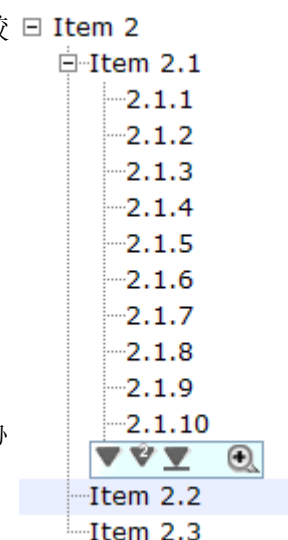
用户可以通过点击  来显示更多的 **tree** 项目(即加大 `pageSize`)，或点击  或  来上下滚动。

若你想显示所有的 **tree** 项目，只需简单的将 `pageSize` 设为 -1。但是，由于浏览器处理含有大量项目的 **tree** 时是非常慢的，所以若 **tree** 控件比较大  **Item 2** 的话不推荐这样做。

除了 **tree** 控件的 `pageSize` 属性，你可以通过每个 `treechildren` 相应的 `pageSize` 属性改变每个 `treechildren` 实例的页大小。

`onPaging`和 `onPageSize`事件

当用户点击  或  来上下滚动某一页时，`onPaging` 事件会协同 `org.zkoss.zul.event.PagingEvent` 实例被发出。类似的，当用户点击  时，`onPageSize` 事件会协同 `org.zkoss.zul.event.PageSize` 实例被发出。



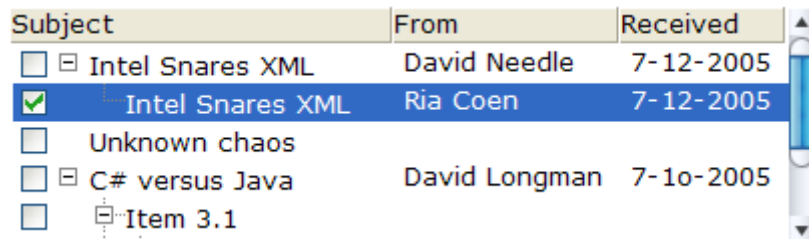
特殊属性

`rows`属性

`rows` 属性控制多少行可见。通过将其设置为 `zero`，**tree** 控件会改变其自身大小来容纳尽可能多的项目。

checkmark属性

checkmark 属性控制是否在每个 tree 项的前面显示一个复选框或一个单选按钮。



Subject	From	Received
<input type="checkbox"/> Intel Snares XML	David Needle	7-12-2005
<input checked="" type="checkbox"/> Intel Snares XML	Ria Coen	7-12-2005
<input type="checkbox"/> Unknown chaos		
<input type="checkbox"/> C# versus Java	David Longman	7-10-2005
<input type="checkbox"/> Item 3.1		

vflex属性

vflex 属性控制是否在垂直方向增长或缩小以适合指定空间。即所谓的垂直柔性 (vertical flexibility)。例如，如果 tree 太大以至于不适合浏览器窗口，此属性会缩小 tree 的高度以控制整个 tree 在浏览器内可见。

若指定了 rows 属性则此属性会被忽略。

maxlength属性

maxlength 属性定义了浏览器端可见字符的最大允许字节数。通过设置这个属性，你可以将 tree 变窄。

可变列宽

就像 columns，你可以将 treecols 的 sizable 属性设置为 true 来允许用户改变 tree 的列宽。类似的，当用户改变宽度时 onColSize 事件被发出。

Tree控件的打开时创建

如下所述，你可以监听 onOpen 事件，然后加载一个 tree 项目的子组件。类似的，你也可以 groupbox 为这样做。

```
<tree width="200px">
  <treecols>
    <treecol label="Subject"/>
    <treecol label="From"/>
  </treecols>
  <treechildren>
```

```

<treeitem open="false" onOpen="load()">
  <treerow>
    <treecell label="Intel Snares XML"/>
    <treecell label="David Needle"/>
  </treerow>
  <treechildren/>
</treeitem>
</treechildren>
<zscript>
void load() {
  Treechildren tc = self.getTreechildren();
  if (tc.getChildren().isEmpty()) {
    Treeitem ti = new Treeitem();
    ti.setLabel("New added");
    ti.setParent(tc);
  }
}
</zscript>
</tree>

```

下拉列表框

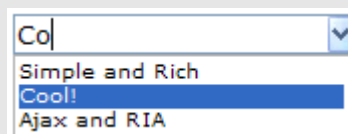
组件： `combobox` 和 `comboitem`。

`combobox` 是一个内嵌下拉列表的特殊文本框。使用 `combobox`，除了直接输入文本，用户还可以从下拉列表中选择一项。

```

<combobox>
  <comboitem label="Simple and Rich"/>
  <comboitem label="Cool!"/>
  <comboitem label="Ajax and RIA"/>
</combobox>


```



无鼠标输入 `combobox`

- Alt+DOWN ，弹出列表。
- Alt+UP 或 ESC 关闭列表 。
- UP 和 DOWN 改变列表项目的选种项。

autodrop属性

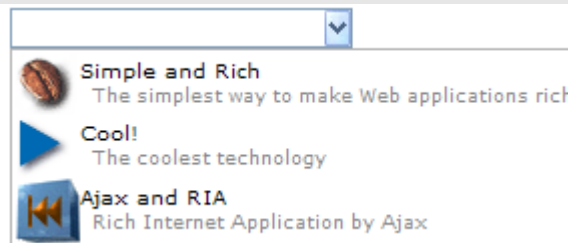
默认情况下，直到用户点击，或按下 Alt+DOWN 后，列表才会打开。但是，你可以将 autodrop 属性设为 true，这样当用户键入任何字符时下拉列表就会打开。这对于初学者来说是有帮助的，但是这或许会令有经验的用户感到厌烦。

```
<combobox autodrop="true"/>
```

description属性

你可以为每个 comboitem 添加一个描述使其易于理解。此外，你可以为每个 comboitem 指派一张图像。

```
<combobox>
  <comboitem label="Simple and Rich" image="/img/coffee.gif"
    description="The simplest way to make Web applications rich"/>
  <comboitem label="Cool!" image="/img/corner.gif"
    description="The coolest technology"/>
  <comboitem label="Ajax and RIA" image="/img/cubfirs.gif"
    description="Rich Internet Application by Ajax"/>
</combobox>
```

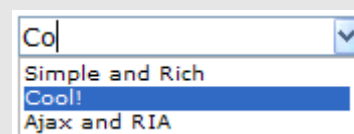


就像其它的支持图像的组件，你可以使用 setImageContent 方法来为 comboitem 组件指派一个动态生成的图像。参考图像(Image)一节来获取细节。

onOpen事件

当用户打开下拉列表时，onOpen 事件被发送至应用程序。为了延迟 comboitem 的创建，你可以按如下所示的方法使用 fulfill 属性。

```
<combobox fulfill="onOpen">
  <comboitem label="Simple and Rich"/>
  <comboitem label="Cool!"/>
  <comboitem label="Ajax and RIA"/>
</combobox>
```



或者，你可以监听 `onOpen` 事件，然后在事件监听器中准备下拉列表或动态的改变列表，如下所示，

```
<combobox id="combo" onOpen="prepare()" />
<zscript>
    void prepare() {
        if (event.isOpen() &&& combo.getItemCount() == 0) {
            combo.appendItem("Simple and Rich");
            combo.appendItem("Cool!");
            combo.appendItem("Ajax and RIA");
        }
    }
</zscript>
```

`appendItem` 方法等价于创建一个 `comboitem`，然后将其父组件指派为 `comobox`。

onChanging事件

由于 `combobox` 也是一个文本框，如果你为它添加了一个事件监听器，`onChanging` 事件会被发出。通过监听此事件，你可以像 [Google Suggests^{\[42\]}](#) 那样操纵下拉列表。此特性有时被称为文本框的自动填充(`autocomplete`)。

如下所示，你可以基于用户的输入来填充下拉列表。


```
<combobox id="combo" autodrop="true" onChanging="suggest()" />
<zscript>
    void suggest() {
        combo.getItems().clear();
        if (event.value.startsWith("A")) {
            combo.appendItem("Ace");
            combo.appendItem("Ajax");
            combo.appendItem("Apple");
        } else if (event.value.startsWith("B")) {
            combo.appendItem("Best");
            combo.appendItem("Blog");
        }
    }
</zscript>
```

注意，当接收 `onChanging` 事件时，`combobox` 的内容并不会改变。因此，你不能使用 `combobox` 的 `value` 属性。而要使用事件 (`org.zkoss.zk.ui.event.InputEvent`) 的 `value` 属性。

[42] <http://www.google.com/webhp?complete=1&hl=en>

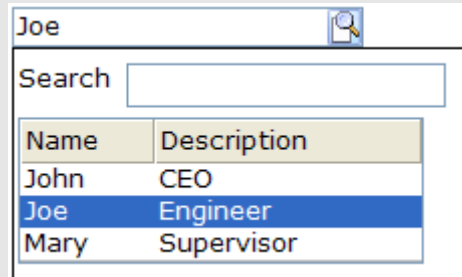
Bandboxes

组件: `bandbox` 和 `bandpopup`。

`bandbox` 是一个内嵌定制弹出 `window` (亦称下拉 `window`) 的特殊文本框, `bandbox` 由一个输入文本框和一个弹出 `window` 组成。当用户按下 `Alt+DOWN` 或点击  按钮时, 弹出 `window` 会自动打开。

不同于 `combobox`, `bandbox` 的弹出 `window` 可以包含任何组件。这样的设计给了开发人员最大的自由。一个典型的应用是将弹出 `window` 作为一个查询对话框。

```
<bandbox id="bd">
  <bandpopup><vbox>
    <hbox>Search <textbox/></hbox>
    <listbox width="200px"
      onSelect="bd.value=self.selectedItem.label;
bd.closeDropdown();">
      <listhead>
        <listheader label="Name"/>
        <listheader
label="Description"/>
      </listhead>
      <listitem>
        <listcell label="John"/>
        <listcell label="CEO"/>
      </listitem>
      <listitem>
        <listcell label="Joe"/>
        <listcell
label="Engineer"/>
      </listitem>
      <listitem>
        <listcell label="Mary"/>
        <listcell label="Supervisor"/>
      </listitem>
    </listbox></vbox>
  </bandpopup>
</bandbox>
```



无鼠标输入 `bandbox`

- Alt+DOWN ，弹出列表。
- Alt+UP 或 ESC 关闭列表。
- UP 和 DOWN 改变列表的选中项。

closeDropdown方法

弹出 window 可以包含任意类型的组件，所以当列表的某个项目被选中时，复制出选中值或关闭弹出 window 是开发人员的工作。

在上面的例子中，我们复制选中项目的标签到 **bandbox** ，然后使用下面的语句关闭弹出 window。

```
<listbox width="200px"
onSelect="bd.value=self.selectedItem.label;
bd.closeDropdown();">
```

autodrop属性

默认情况下，直到用户点击  按钮或按下 Alt+DOWN 时，弹出 window 才会打开。

但是，你可以将 autodrop 属性设置为 true，这样当用户键入任意字符时弹出 window 就会打开。这对于初学者来说是有帮助的，但是这或许会令有经验的用户感到厌烦。

```
<bandbox autodrop="true"/>
```

onOpen事件

若用户打开了弹出 window，onOpen 事件会被发送至应用程序。通过将 fulfill 属性设置为 onOpen，你可以延迟弹出 window 的创建。

```
<bandbox fulfill="onOpen">
  <bandpopup>
    ...
  </bandpopup>
</bandbox>
```

或者，通过监听 onOpen 事件你可以在 Java 代码中准备弹出 window，如下所示。

```
<bandbox id="band" onOpen="prepare()" />
<zscript>
  void prepare() {
    if (event.isOpen() &&& band.getPopup() == null) {
      ...//create child elements
```

```
    }  
  }  
</zscript>
```

onChanging事件

由于 **bandbox** 也是一个文本框，如果你为它添加了一个事件监听器，**onChanging** 事件会被发出。通过监听此事件，你可以随意操纵弹出 **window**。

如下所示，你可以基于用户的输入来填充下拉列表。

```
<bandbox id="band" autodrop="true" onChanging="suggest()" />  
<zscript>  
  void suggest() {  
    if (event.value.startsWith("A")) {  
      ...//do something  
    } else if (event.value.startsWith("B")) {  
      ...//do another  
    }  
  }  
</zscript>
```

注意，当接收 **onChanging** 事件时，**bandbox** 的内容并不会改变。因此，你不能使用 **bandbox** 的 **value** 属性。而使用事件 (`org.zkoss.zk.ui.event.InputEvent`) 的 **value** 属性。

图表

组件: **chart**

chart 用于显示以图形方式一组数据。它可以帮助用户以快照(**snapshot**)形式判断某事物。

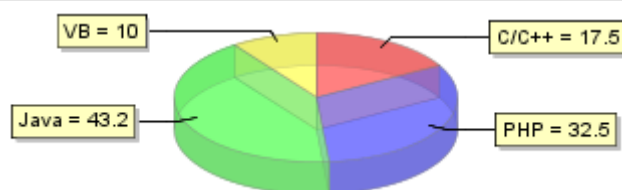
使用 **chart** 组件非常直观。在数据模型中准备合适的数数据且将其送入 **chart**。下面为一个饼图(**pie chart**)的例子。

```
<chart id="mychart" type="pie" width="400" height="200"  
threeD="true" fgAlpha="128">  
  <zscript><![CDATA[  
    PieModel model = new SimplePieModel();  
    model.setValue("C/C++", new Double(17.5));  
    model.setValue("PHP", new Double(32.5));  
    model.setValue("Java", new Double(43.2));
```

```

model.setValue("VB", new Double(10.0));
mychart.setModel(model);
]]></zscript>
</chart>

```



不同类型的 **chart** 用于展示不同类型的数据；因此，**chart** 必须提供合适的数据模型。对于饼图，开发人员必须提供 **PieModel** 作为它们的数据模型，而柱状图，线图，面积图和瀑布图 (**bar chart**, **line chart**, **area chart**, and **waterfall chart**) 需要 **CategoryModel** 和 **XYModel** 。

实况数据

上面的例子有一点点令人误解。实际上，在将数据填充到 **chart** 前，开发人员不一定必须准备真实的数据，因为 **chart** 组件支持实况数据机制。使用实况数据，开发人员可以从视图分离出数据。换言之，开发人员可以从数据模型添加，更改及移除数据，而 **chart** 会根据此刷新。对于一些高级实现，开发人员甚至可以实现 **org.zkoss.zul.ChartModel** 接口来提供他们自己的 **chart** 模型。

向下钻取(**onClick**事件)

当用户查看一个 **chart**，且发现了一些有趣的东西，很自然的用户会想了解关于兴趣点的详细信息。兴趣点通常会 是饼图中的一块，柱状图中的一个条形柱，线图中的一点。**chart** 组件支持这样的向下钻取(**drill down**)功能，通过自动将 **chart** 分割成 **area** 组件，然后用户可以点击 **chart** 来触发一个 **onClick** 鼠标事件。然后，开发人员可以定位 **area** 组件，并处理相应的向下钻取(**do whatever appropriate drill down**)。

在 **area** 组件的 **componentScope** 内，有一些开发人员可以使用的有用信息。

名称	描述
entity	区域的实体类型。(例如 TITLE , DATA , CATEGORY , LEGEND)
series	关联数据的连续名称(CategoryModel , XYModel , or HiLoModel)。
category	关联数据的类别名称 (PieModel or CategoryModel)。
url	字符串类型的 url ，用于向下钻取一个遗留(legacy)页面。
value	关联数据的数值(PieModel or CategoryModel)。
x	关联数据的 x 值 (XYModel)。
y	关联数据的 y 值 (XYModel)。

名称	描述
date	关联数据的 date 值 (HiLoModel)。
open	关联数据的 open 值(HiLoModel)。
high	关联数据的 high 值 (HiLoModel)。
low	关联数据的 low 值 (HiLoModel)。
close	关联数据的 close 值 (HiLoModel)。
volume	关联数据的 volume 值 (HiLoModel)。

在下面的例子中，在 chart 中我们提供了一个 onClick 事件监听器。它定位 area 组件并且显示 area 组件(也就是 pie)的分类(category)。

```
<chart id="mychart" type="pie" width="400" height="250">
  <attribute name="onClick">

alert(self.getFellow(event.getArea()).getAttribute("category")
);
  </attribute>
  <zscript><![CDATA[
    PieModel model = new PieModel();
    model.setValue("C/C++", new Double(17.5));
    model.setValue("PHP", new Double(32.5));
    model.setValue("Java", new Double(43.2));
    model.setValue("VB", new Double(10.0));
    mychart.setModel(model);
  ]]></zscript>
</chart>
```

操作区

Chart 组件也提供了一个 area 渲染器(renderer) 机制，开发人员可以操纵 chart 的 area 组件。

使用 area 渲染器仅需两步。

1. 实现 org.zkoss.zul.event.ChartAreaListener 接口操纵 area 组件，例如，改变 area 的 tooltipText。
2. 将 chart 的 areaListener 属性设置为监听对象或监听类的名称。

这样，开发人员取得了以一个机会，改变 area 组件属性或将更多的信息插入打到 area 组件的 componentScope 属性，因此可以通过 onClick 事件监听器。

拖放

ZK 允许用户在用户界面内拖曳特定的组件。例如，将文件拖至其它的目录，或将商品拖至购物车。

若一个组件可以被拖曳则它是可拖曳的。若用户可以将一个可拖曳的组件放入到某一组件内，则称该组件是可放下的(droppable)。

注：在放下后，ZK 并不假定关于发生什么的任何行为。这由应用程序开发人员编写 `onDrop` 事件监听器来决定。

如果应用程序什么也不做，被拖曳的组件只是简单的移回它的初始位置。

draggable 和 droppable 属性

使用 ZK，通过指派 `draggable` 属性除了"false"外的任何值，你可以使一个组件变为可拖曳的。若想禁用，则将其设为"false"。

```
<image draggable="true"/>
```

类似的，可以将 `droppable` 属性赋值为"true"来将一个组件变为可放下的。

```
<hbox droppable="true"/>
```

然后，用户可以拖曳一个可拖曳的组件，并将其放入一个可放下的组件。

onDrop 事件

一旦用户拖曳一个组件并将其放入可放下的另一个组件，`onDrop` 事件会通知用户放入组件的组件。`onDrop` 事件是 `org.zkoss.ui.event.DropEvent` 的一个实例。调用 `getDragged` 方法，你可以获取什么被拖曳(及放下)了。

注意 `onDrop` 事件的目标是可放下的组件，而不是被拖曳的组件。

下面是一个简单的例子，允许用户使用拖放对列表项目重新排序。

Reorder by Drag-and-Drop		
Unique Visitors of ZK:		
Country/Area	Visits	%
United States	5,093	19.39%
China	4,274	16.27%
France	1,892	7.20%
Germany	1,846	7.03%
(other)	13,162	50.01%
Total 132	26,267	100.00%

```

<window title="Reorder by Drag-and-Drop" border="normal">
  Unique Visitors of ZK:
  <listbox id="src" multiple="true" width="300px">
    <listhead>
      <listheader label="Country/Area"/>
      <listheader align="right" label="Visits"/>
      <listheader align="right" label="%"/>
    </listhead>
    <listitem draggable="true" droppable="true"
onDrop="move(event.dragged)">
      <listcell label="United States"/>
      <listcell label="5,093"/>
      <listcell label="19.39%"/>
    </listitem>
    <listitem draggable="true" droppable="true"
onDrop="move(event.dragged)">
      <listcell label="China"/>
      <listcell label="4,274"/>
      <listcell label="16.27%"/>
    </listitem>
    <listitem draggable="true" droppable="true"
onDrop="move(event.dragged)">
      <listcell label="France"/>
      <listcell label="1,892"/>
      <listcell label="7.20%"/>
    </listitem>
    <listitem draggable="true" droppable="true"
onDrop="move(event.dragged)">
      <listcell label="Germany"/>
      <listcell label="1,846"/>
      <listcell label="7.03%"/>
    </listitem>
    <listitem draggable="true" droppable="true"
onDrop="move(event.dragged)">
      <listcell label="(other)"/>
      <listcell label="13,162"/>

```

```

        <listcell label="50.01%"/>
    </listitem>
    <listfoot>
        <listfooter label="Total 132"/>
        <listfooter label="26,267"/>
        <listfooter label="100.00%"/>
    </listfoot>
</listbox>
<zscript>
void move(Component dragged) {
    self.parent.insertBefore(dragged, self);
}
</zscript>
</window>

```

使用多选拖曳

当用户拖放一个列表项或 **tree** 项时，这些项目的选择状态不会改变。仅当移动拖曳项时是可视的，但是，通过查询所有选中项的集合，你可以处理所有的选中项，如下所示。

```

public void onDrop(DropEvent evt) {
    Set selected =
    ((Listitem)evt.getDragged()).getListbox().getSelectedItems();
    //then, you can handle the whole set at once
}

```

注意，被拖曳项允许并未被选中。因此对于此事例，你或许更喜欢将选中项改变为拖曳项，如下所示。

```

Listitem li = (Listitem)evt.getDragged();
if (li.isSelected()) {
    Set selected =
    ((Listitem)evt.getDragged()).getListbox().getSelectedItems();
    //then, you can handle the whole set at once
} else {
    li.setSelected(true);
    //handle li only
}

```

可拖曳组件的多种类型

可放下组件不接受所有的可拖曳组件是和平常的。例如，一个 e-mail 文件夹只接受 e-mail 且拒绝联系(contacts)或其它。当调用 onDrop 时，你可以默默地忽略不可接受的组件，或是发出警告消息。

为了获得更好的视觉效果，你可以使用一个标识来确定每种类型的可拖曳组件，然后将标识赋予 draggable 属性。

```
<listitem draggable="email"/>
...
<listitem draggable="contact"/>
```

然后，你可以为 droppable 属性指定一个标识列表来限制可被放下的组件。例如，下面的图像仅接受 email 和 contact。

```
<image src="/img/send.png" droppable="email, contact"
onDrop="send(event.dragged)"/>
```

若想接受任何类型的组件，你可以将 droppable 属性的值设为 "true"。例如，下面的图像可以接受任意类型的可拖曳组件。

```
<image src="/img/trash.png" droppable="true"
onDrop="remove(event.dragged)"/>
```

另外，如果 draggable 属性的值为 "true"，则意味着此组件属于匿名类型。此外，只有 droppable 属性值为 "true" 的组件接受此组件。

HTML相关组件

在同一 ZUML 页面内和 XUL 组件一起使用 HTML 标签有几种方式。你可以根据需求选择任意一个。首先，你可以使用 html 组件嵌入 HTML 标签。使用此方法，HTML 标签仅简单的为 html 组件的内容。它们会直接被送至客户端，对于 ZK 来说它们没有具体的意义。

第二，你可以使用 XHTML(<http://www.w3.org/1999/xhtml>) 命名空间来从 XHTML 组件集内指定一个组件。换句话说，XHTML 命名空间代表的相关 XML 元素为 XHTML 组件集内的组件。就像 ZUL 组件集(<http://www.zkoss.org/2005/zul>)，ZK 为 ZUML 页面内每个 XML 元素创建一个实例。

第三，你可以使用本地命名空间(<http://www.zkoss.org/2005/zk/native>) 来代表将一个 HTML 标签直接送至客户端，而不是为它们创建 ZK 组件。这种方法更有效，但是不能动态改变。

最后一种但并非最不重要，你可以使用 `inclusion (include)` 和 `inline frames (iframe)` 嵌入一个 ZUL 页面，理论上可以是任何内容(不限于 HTML 标签)。

html 组件

最简单的方式是使用称为 `html` ^[43] 的 XUL 组件，在其中嵌入你想直接发送至浏览器的 HTML 标签。为了避免 ZK 解释 HTML 标签，通常要使用 `<![CDATA[and]]>` 将 HTML 标签围入其中。换言之，他们并不是子组件。而是被存储在 `content` 属性 ^[44] 内。注意在其中可以使用 EL 表达式。

```
<window title="Html Demo">
  <html><![CDATA[
    <h4>Hi ${parent.title}</h4>
    <p>It is the content of the html component. </p>
  ]]></html>
</window>
```

在此处将会成为元素的内容(又见 `org.zkoss.zul.Html` 类的 `getContent` 方法)。

提示：你可以使用 `attribute` 元素来指定 XHTML 片段，以代替 `CDATA`,如下。

```
<html
  <attribute name="content">
    <h4>Hi, ${parent.title}</h4>
    <p>It is the content of the html component.</p>
  </attribute>
</html>
```

参考 ZK 用户界面标记语言一章 `attribute` 元素一节。

`html` 组件会产生 `HTML SPAN` 标签来包围这些内容。换句话说，当提交到浏览器是时会产生下列的 HTML 标签。

```
<span id="z_4a_3">
  <h4>Hi, Html Demo</h4>
  <p>It is the content of the html component.</p>
</span>
```

`html` 组件会产生 `HTML SPAN` 标签来包围这些内容。因此，你可以像使用其它的 XUL 组件一样使用它。例如，我们可以指定 CSS 样式，动态的改变它的内容。

```
<html id="h" style="border: 1px solid blue;background:
yellow"><![CDATA[
  <ul>
    <li>Native browser content</li>
  </ul>
]]></html>
<button label="change" onClick="l.setContent(&quot;Hi
Update&quot;)" />
```

注意，由于 SPAN 用于包围嵌入的 HTML 标签，下面的语句是错误的。

```
<html><![CDATA[
  <ul>
    <li> <!-- incorrect since <ul><li> is inside <span> -->
  </li>
]]></html>

<textbox/>

<html><![CDATA[
  </li>
</ul>
]]></html>
```

如果你需要直接产生嵌入的 HTML 标签，而不用 SPAN 包围，你可以使用 **native** 命名空间，如下所示。

Native命名空间, <http://www.zkoss.org/2005/zk/native>

使用 **Native** 命名空间，表示 ZUML 页面内的 XML 元素会被直接送至浏览器而不会成为 ZK 组件。例如，

```
<n:ul xmlns:n="http://www.zkoss.org/2005/zk/native">
  <n:li>
    <textbox/>
  </n:li>
  <n:li>
    <textbox/>
  </n:li>
</n:ul>
```

会产生下面的 HTML 标签送至浏览器：

```
<ul>
  <li>
```

```

<input id="z_a3_2"/>
</li>
<li>
<input id="z_a3_5"/>
</li>
</ul>

```

此处，`<input>`为由`textbox`产生的HTML标签。不同于上例中的`textbox`，ZK加载器并不会为每个`ul`和`li` [\[45\]](#) 创建一个组件。而是，它们直接被送至客户端。当然，它们必须是被客户端可识别的。对于HTML浏览器，它们必须为合法的HTML标签。

由于与 **Native** 关联的元素会直接被送至客户端，所以它们不是 **ZK** 组件，且在客户端没有对应部分。优点是根据内存和处理事件来看都有更好的性能。而缺点是你不能够动态的改变它们。例如，下面的代码片断是错误的，因为没有名为 `x` 的组件。

```

<n:ul id="x" xmlns:n="http://www.zkoss.org/2005/zk/native"/>
<button label="add" onClick="new Li().setParent(x)"/>

```

若你想动态改变它们，则可以按如下章节描述的那样指定 **XHTML** 命名空间。

使用Native输出另外的命名空间

若你想生成另一个命名空间至输出，则可以使用另一种格式作为 **Native** 命名空间的URI。

```
native:URI-of-another-namespace
```

例如，若你想直接输出 **SVG** 标签之客户端，则可以指定 `native:native:http://www.w3.org/2000/svg`，如下。

```

<window>
  <svg width="100%" height="100%" version="1.1"
    xmlns="native:http://www.w3.org/2000/svg">
    <ellipse cx="240" cy="100" rx="220" ry="30"
style="fill:purple"/>
  </svg>
</window>

```

那么，客户端接受到的内容如下：

```

<div id="z_lx_0c" z.type="zul.wnd.Wnd">
  <svg width="100%" height="100%" version="1.1"
    xmlns="http://www.w3.org/2000/svg">

```

```
<ellipse cx="240" cy="100" rx="220" ry="30"
style="fill:purple"/>
</svg>
</div>
```

XHTML命名空间, <http://www.w3.org/1999/xhtml>

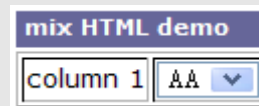
XHTML表示XHTML组件集,就像ZUL命名空间(<http://www.zkoss.org/2005/zul>)代表ZUL组件集一样。因此,使用XHTML命名空间指定的一个XML元素表示此组件的创建将会基于定义在XHTML组件集内的组件。例如,下面的代码基于XHTML组件集创建了一个ul实例:

```
<h:ul xmlns:h="http://www.w3.org/1999/xhtml">
```

换言之,ZK 加载器将会在 XHTML 组件集内寻找组件定义 ul,然后基于它创建一个实例。

下面是一个更完整的例子。

```
<window title="mix HTML demo"
xmlns:h="http://www.w3.org/1999/xhtml">
  <h:table border="1">
    <h:tr id="row1">
      <h:td>column 1</h:td>
      <h:td>
        <listbox id="list" mold="select">
          <listitem label="AA"/>
          <listitem label="BB"/>
        </listbox>
      </h:td>
    </h:tr>
  </h:table>
  <button label="add" onClick="new
org.zkoss.zhtml.Td().append(row1)"/>
</window>
```



不同于html组件,ZK加载器会为存储在content属性内HTML标签创建ZK组件。优点是你可以动态的操作每个HTML标签,就像在上面例子中描述的那样(add按钮)。缺点是需要花费更长的处理事件及更多的空间来维护。

[提示]: 不同于XHTML命名空间,Native并不代表另一种组件集。它只是一个保留的命名空间,用于告诉ZK加载器将它们直接送至客户端,以取得更好的性能。

include组件

include 组件用于包含由另一个 `servlet` 产生的输出。`servlet` 可以是任何页面，包括 JSF, JSP, 甚至另一个 ZUML 页面。

```
<window title="include demo" border="normal" width="300px">
  Hello, World!
  <include src="/userguide/misc/includedHello.zul"/>
  <include src="/html/frag.html"/>
</window>
```

就像所有的其它属性，你可以在运行时动态的改变 `src` 属性包括来自于不同 `servlet` 的输出。

若被包括的输出是另一个 ZUML，开发人员被允许访问此 ZUML 的组件，就好像它们是包含(containing)页面的一部分。

将值传递至包含页面

有两种方式将值传递至包含页面。一，你可以使用查询字符串。

```
<include src="mypage?some=something"/>
```

然后，在包含页面内，你可以使用 `Execution` 或 `ServletRequest` 接口的 `getParameter` 方法来访问它们。在 EL 表达式(包含页面内的)中，你可以使用 `param` 变量来访问它们。但是，使用查询字符串，你仅能传递字符串类型的值。

```
${param.some}
```

另一种方式，你可以利用 `setDynamicProperty` 方法所谓的动态属性，或者 ZUL 内的一个动态属性来传递任意值，如下：

```
<include src="mypage" some="something" another="${expr}"/>
```

使用动态属性，你可以传递非字符串类型的值。在包含页面内，可以使用 `Execution` 或 `ServletRequest` 接口的 `getAttribute` 方法来访问它们。在 EL 表达式(包含页面内的)中，你可以使用 `requestScope` 变量来访问它们。

```
${requestScope.some}
```


包含ZUML页面

如果 `include` 组件被用于包含一个 ZUML 页面，那么被包含的页面会成为桌面的一部分。但是，直到请求被完全处理被包含页面才会可见。换言之，仅当下列的事件被用户或计时器(timer)触发时被包含页面才会可见。

理由是`include`组件到页面响应阶段^[46](the Rendering phase)才会包含一个页面。另外`zscript` 发生在组件创建阶段(the Component Creation phase)， `onCreate` 发生在事件处理阶段(the Event Processing Phase)。它们都在包含前执行。

```
<window onCreate="desktop.getPages()"> <!-- the included page not
available -->
  <include src="/my.zul"/>
  <zscript>
    desktop.getPages(); //the included page not available yet
  </zscript>
  <button label="Hit" onClick="desktop.getPages()" />
  <!-- Yes, the included page is available when onClick is
received -->
</window>
```

若你想浏览被包含页面的组件，宏(macro)组件通常是更好的选择。参考 ZK 用户界面标记语言一章中宏组件一节。

style组件

`style` 组件被用于在 ZUML 页面中指定 CSS 样式。最简单的格式如下。

```
<style>
.blue {
color: white; background-color: blue;
}
</style>
<button label="OK" sclass="blue"/>
```

OK

提示：为整个应用程序配置样式表，可以在 `zk.xml` 内指定 `theme-uri`，参考国际化一章中主题一节或 the Developer's Reference 中的附录 B(Appendix B) 获取细节。为一中语言配置样式表，可以使用语言插件(language addon)，参考 the Component Development Guide。

有时将所有的 CSS 定义放在一个单独的文件中是更好的选择，例如 `my.css`。然后，我们可以使用 `style` 组件来引用它，如下。

```
<style src="/my.css"/>
```

上面的语句实际下列的HTML标签^[47]送至服务器，所以指定的文件必须是可被浏览器访问的。

```
<link rel="stylesheet" href="/css/mystyles.css"/>
```

换言之，你不能指定"/WEB-INF/xx" 或 "C:/xx/yy"。

就像其它的 URI, style 接受 "*" 来加载浏览器和本地化(Locale dependent)样式表。参考国际化一章中浏览器和本地化 URI 一节获取细节。

script组件

script 组件用于指定运行在浏览器的脚本代码。注意，不同于 zscript, 脚本代码运行在浏览器。通常由大多数浏览器支持的 JavaScript 编写。最简单的格式如下。

```
<script type="text/javascript">
function myfunc() {
    $e("${win.uuid}").style.backgroundColor = "blue";
}
</script>
```

如上所示，你可以在脚本代码内使用 EL 表达式 (\${win.uuid})。

当然，你可以使用 src 属性引用额外的 JavaScript 文件，如下。

```
<script src="/js/super.js" type="text/javascript"/>
```

由于 ZK 应用程序运行在服务器端(使用你最喜欢的语言执行)，开发人员很少需要使用 JavaScript 代码来执行。它们通常定制 ZK 客户端引擎的行为，或运行遗留下来的 JavaScript 库。

iframe组件

iframe 组件使用 HTML IFRAME 标签将显示的一部分委托(delegate)给另一个 URL。尽管外观看起来与 include 组件相似，但是 iframe 组件的概念及意义是不同的。

被 include 组件包含的内容是整个 HTML 页面的片断。由于内容是 HTML 页面的一部分，所以也是桌面的一部分，你可以访问 include 组件内的任何组件。包含是在服务器进行的，浏览器并不知道。这意味着 src 属性指定的 URL 可以是任何内部资源。

iframe 组件的内容是由浏览器加载的，作为一个单独的页面。由于是作为单独的页面被加载，所以内容的格式可以不同于 HTML。例如，你可以嵌入一个 PDF 文件。

```
<iframe src="/my.pdf"/>
...other HTML content
```

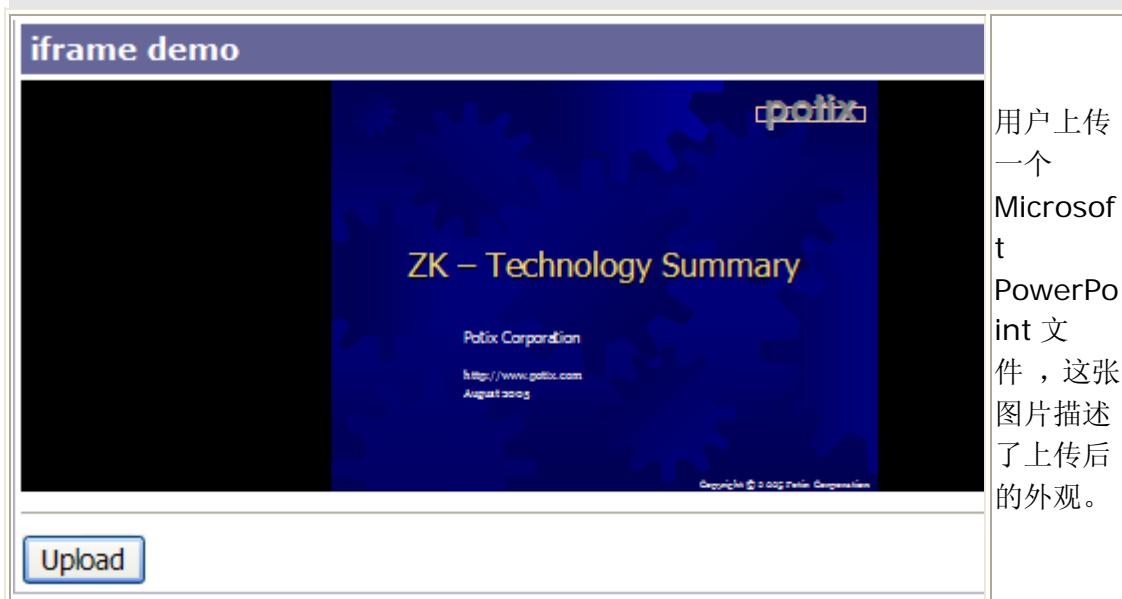
提示：默认没有边框。若想启用，使用 style 属性指定，例如，`<iframe style="border:1px inset" src="http://www.zkoss.org"/>`

当解释包含 IFRAME 的标签时，embedding 是被浏览器处理的。这也暗示着 URL 必须为从浏览器可访问的资源。

就像 image 和 audio 组件^[48]，你可以动态指定生成的内容。典型的例子是使用 JasperReport^[49] 生成一个 PDF 报告，以二进制数组或流格式，然后将结果包装成 org.zkoss.util.media.AMedia 类传递给 iframe 组件。

在下面的例子中，我们说明，你可以使用 iframe 嵌入任何内容，只要客户端支持内容的格式。

```
<window title="iframe demo" border="normal">
  <iframe id="iframe" width="95%"/>
  <separator bar="true"/>
  <button label="Upload">
    <attribute name="onClick">{
      Object media = Fileupload.get();
      if (media != null)
        iframe.setContent(media);
    }</attribute>
  </button>
</window>
```



onURICHange事件

当用户操作 `iframe` 组件指向另一个 URL(或 bookmark 签)时, `org.zkoss.zk.ui.event.URIEvent` 类的一个对象会被发送至 `iframe` 组件。这个事件通常被用于标记(bookmark)`iframe` 组件的状态, 这样之后正确的内容会被修复(be restored)。

与其他技术的整合

若 `iframe` 组件包含一个非 ZK 页面, 则 `onURICHange` 事件不会被发送。例如, 若包含一个 PDF 页面, 此事件就不会被发送。

另一方面, 如果你使用了其它技术将 ZK 页面置于一个 `iframe` 内, 可以编写一个叫做 `onIframeChange` 的 JavaScript 方法监视 URL。

```
//Part of your, say, PHP page
<script type="text/script">
function onIframeChange(uuid, url) {
    do_whatever_you_need_in_the_technology_you_use(uuid, url);
}
</script>
```

这里 `uuid` 是你使用 `document.getElementById` 获取的元素的 ID, `url` 是 `iframe` 要浏览的一个新 URL。注意, `url` 包含上下文路径(context path), 而 `URIEvent.getURI()` 不包含。

^[43] html元素内的文本实际上是由html组件的content属性赋值的(而不是成为一个子标签)。

^[44] 若你不熟悉XML, 参考ZK用户界面标记语言一章中XML一节。

^[45] ZK实际上创建了一个特殊的组件来表示尽可能多的使用Native命名空间的XML元素。

^[46] 参考组件活动周期(Component Lifecycle)一章获取细节。

^[47] 实际的结果取决于Web应用程序的配置。

^[48] 在许多方面, `iframe`类似于`image` 和 `audio`。你可以将`iframe`当成一个可以包含任意内容的组件。

^[49] <http://jasperreports.sourceforge.net>


用HTML FORM 和Java Servlets

事件驱动模型是简单但强大的，但是用事件监听器重写所有的 servlets 是不实际的。

name属性

为了与遗留的(legacy)Web 应用程序一起工作，你可以指定 name 属性，就像在 HTML 页面那样。例如，

```
<window xmlns:h="http://www.w3.org/1999/xhtml">
  <h:form method="post" action="/my-old-servlet">
    <grid>
      <rows>
        <row>When <datebox name="when"/> Name <textbox name="name"/>
Department
        <combobox name="department">
          <comboitem label="RD"/>
          <comboitem label="Manufactory"/>
          <comboitem label="Logistics"/>
        </combobox>
      </row>
      <row>
        <h:input type="submit" value="Submit"/>
      </row>
    </rows>
  </grid>
</h:form>
</window>
```

When	2006/03/01 	Name	Bill Gates	Department	Manufactory
<input type="button" value="Submit"/>					

一旦用户按下了 submit 按钮，一个如下查询字符串(query string)请求被提交到 my-old-servlet servlet 。

```
/my-old-servlet?when=2006%2F03%2F01&name=Bill+Gates&department=Manufactory
```

因此，只要在 name 和 value 间保持适当的关联，servlet 就可以如平常一样工作而无需修改。

支持name属性的组件

所有的输入型组件都支持 name 属性，如 textbox, datebox, decimalbox, intbox, combobox, bandbox, slider 和 calendar。

此外，listbox 和 tree 控件也支持 name 属性。若 multiple 属性为 true，且用户选择了多项，那么多对 name/value 被提交。

```
<listbox name="who" multiple="true" width="200px">
  <listhead>
    <listheader label="name"/>
    <listheader label="gender"/>
  </listhead>
  <listitem value="mary">
    <listcell label="Mary"/>
    <listcell label="FEMALE"/>
  </listitem>
  <listitem value="john">
    <listcell label="John"/>
    <listcell label="MALE"/>
  </listitem>
  <listitem value="jane">
    <listcell label="Jane"/>
    <listcell label="FEMALE"/>
  </listitem>
  <listitem value="henry">
    <listcell label="Henry"/>
    <listcell label="MALE"/>
  </listitem>
</listbox>
```

name	gender
Mary	FEMALE
John	MALE
Jane	FEMALE
Henry	MALE

若选中了 John 和 Henry，那么查询字符串将包含：

```
who=john&who=henry
```

注意，为了使用 listbox 和 tree 控件的 name 属性，你必须分别为 listitem 和 treeitem 指定 value 属性。它们即为被提交到 servlet 的值。

丰富用户界面

因为 form 组件可以包含任意类型的组件，丰富的用户界面可以独立地实现，独立于已存在的 servlet。例如，你可以监听 onOpen 事件，填充(fulfill)一个 tab 面板，就如前一章节所示。另一个例子，你可以动态的为一个 grid 控件添加更多的行，每行可

以控制一个含有 name 属性的输入框。一旦用户提交了表单，最新更新的内容会被提交到 servlet 。

客户端行为

某些行为在客户端用 JavaScript 代码处理更为合适，例如动画和图像转滚(image rollovers)。为了在客户端执行 JavaScript 代码，ZK 引入了客户端行为(CSA)的概念。使用 CSA，开发人员可以监听任何 JavaScript 事件且在客户端执行 JavaScript 代码。

CSA 类似于事件监听器，除了 CSA 的行为是用 JavaScript 编写的且在客户端执行。ZK 允许开发人员为任何 JavaScript 事件指定行为，例如 onfocus, onblur, onmouseover 和 onmouseout，只要你的目标浏览器支持它们。

CSA 的语法如下。

```
action="[onfocus|onblur|onmouseover|onmouseout|onclick|onshow|onhide...]: javascript;"
```

注意 CSA 是完全独立于 ZK 事件监听器的，尽管它们或许有相同的名字，例如 onFocus。不同点包括：

1. CSA 在客户端执行，且在服务器端的 ZK 事件将监听器被调用前发生。
2. CSA 代码用 JavaScript 编写，而 ZK 事件将监听器用 Java 编写。
3. CSA 可以注册你的目标浏览器允许的任何事件，而 ZK 近仅支持在事件(Events)章节中列出的事件。

引用一个组件

在 JavaScript 代码中，你可以通过晚捆绑(late-binding)EL 表达式来引用一个组件或其它对象。晚捆绑 EL 表达式以 #{ 开始且以 } 结束，如下所示。

```
<button action="onmouseover: action.show("#{parent.tip})" />
```

晚捆绑 EL 表达式在响应阶段被赋值。另外，如果一个 EL 表达式以 \${ 开始，则它会在组件创建阶段被赋值，在为 action 属性赋值前。例如，

```
<button action="onfocus: action.show(${tip}); onblur: action.hide(${tip})" />
<div id="tip" visible="false">...</div>
```

将等价于

```
<button action="onfocus: action.show(); onblur: action.hide()" />
```

```
<div id="tip" visible="false">...</div>
```

由于 tip 组件在为 action 属性赋值时才会被创建。

由于 ZUML 加载器(loader)并不知道 CSA，它调用 toString 方法将组件转换成一个字符串，所以即使在为 action 属性赋值前创建了被引用组件仍是不正确的。


当然，这并不妨碍你在某一 action 内使用\${}，如下所述。仅需记住它是在为 action 属性赋值前赋值。

```
<variables myaction="onfocus: action.show("#{tip}"); onblur:
action.hide("#{tip}");"
<button action="${myaction} onmouseover:
action.show("#{parent.parent.tip}))/>
```

一个onfocus和的onblur例子

在下面的例子中，我们展示了如何使用 CSA 来提供在线帮助。当用户改变了任一个文本框的聚焦时，一条帮助消息就会据此显示。

```
<grid>
  <columns>
    <column/>
    <column/>
    <column/>
  </columns>
  <rows>
    <row>
      <label value="text1: "/>
      <textbox action="onfocus: action.show("#{help1}"); onblur:
action.hide("#{help1}))/>
      <label id="help1" visible="false" value="This is help for
text1."/>
    </row>
    <row>
      <label value="text2: "/>
      <textbox action="onfocus: action.show("#{help2}"); onblur:
action.hide("#{help2}))/>
      <label id="help2" visible="false" value="This is help for
text2."/>
    </row>
  </rows>
</grid>
```



强制规则

ZUL 组件实际上是将一个 EL 表达式 (`{}`) 转换为合适的 JavaScript 代码, 基于结果对象的类。

1. 若结果为 `null`, 则用 `null` 替换。
2. 若结果为一个组件, 则用 `{uuid}` 替换, `{}` 处为一个返回 HTML 标签引用的 JavaScript 函数, `uuid` 为组件的 UUID。
3. 若结果为一个 `Date` 对象, 则用 `new Date(milliseconds)` 替换。
4. 否则。结果调用 `toString` 方法将结果转换为一个字符串, 然后用 `'result in string'` 替换。

onshow和onhide 行为

onshow 和 onhide 行为被用于控制显示及隐藏一个组件的视觉效果。

改变window如何出现的例子

```
<zkc>
  <button label="Show Overlapped"
onClick="win.doOverlapped();" />
  <window id="win" border="normal" width="200px"
mode="overlapped"
action="onshow:anima.appear({self});onhide:anima.fade({self}
)" visible="false">
    <caption image="/img/inet.png" label="Hi there!" />
    <checkbox label="Hello, Effect!" />
  </window>
</zkc>
```

CSA JavaScript工具

为节简化 CSA 编程, ZK 提供了一些你可以利用的工具对象。

action对象

可用于任何对象的基本工具。

函数	描述
----	----

函数	描述
<code>action.show(cmp)</code>	使一个组件可见。 cmp – 组件。 使用 <code>#{ EL-expr }</code> 标识它。
<code>action.hide(cmp)</code>	使一个组件不可见。 cmp – 组件。 Use <code>#{EL-expr}</code> 标识它。

提示: 对于 JavaScript 程序员, 直接操纵样式用于显示是很平常的。但是, 这并不是一个好主意。而是使用 `action.show` and `action.hide` 代替, 因为 ZK 客户端引擎必须处理视觉效果, **bug workaround**, 等等。

comm对象

用于与服务器通信。

函数	描述
<code>comm.onClick(cmp, info)</code>	发送 onClick 事件至服务器。 cmp – 组件, 使用 <code>#{EL-expr}</code> 或 <code>this</code> 指定它。 info – 一个字符串或 <code>null</code> 用于提供额外的信息。它会成为 <code>MouseEvent</code> 的 <code>getArea</code> 的返回值。
<code>comm.onUser(cmp, ...)</code>	将 onUser 事件发送至服务器。 cmp – 组件, 使用 <code>#{EL-expr}</code> 或 <code>this</code> 指定它。 other – 你可以提供很多参数。
<code>comm.onEvent(cmp, evt, ...)</code>	将指定的事件发送至服务器。 cmp – 组件, 使用 <code>#{EL-expr}</code> 或 <code>this</code> 指定它。 evt – 事件名称 例如, <code>onUser</code> 。 other – 你可以提供很多参数。

例如,

```
<window title="Test of JavaScript Utilities">
```

```

    <html onClick='l.value = "onClick "+event.area'
      onUser='l.value ="onUser '
+org.zkoss.lang.Objects.toString(event.data)'><![CDATA[
    <a href="javascript:;" onclick="comm.sendClick(this,
'Hi')">onClick with Hi</a>
    <a href="javascript:;"
onClick="comm.sendClick(this)">onClick with null</a>
    <a href="javascript:;" onclick="comm.sendUser(this)">onUser
with null</a>
    <a href="javascript:;" onclick="comm.sendUser(this,
'One')">onUser with One</a>
    <a href="javascript:;" onclick="comm.sendUser(this, 'One',
'Two')">onUser with [One, Two]</a>
    <a href="javascript:;" onclick="comm.sendEvent(this,
'onUser', 'XYZ')">onUser with XYZ</a>
  ]]></html>
  <separator/>
  <label id="l"/>
</window>

```

anima对象

动画般的视觉效果。基于 script.aculo.us ^[50] 提供的 [Effect](#) 类。API被简化了。若你想要更多的视觉效果或空间，可以直接访问 [Effect](#)。

注：[Effect](#)要求组件使用DIV标签包围。并不是所有的ZUL组件都以这种方式被实现。若有疑问，可以与div组件嵌套使用，如下。

```

<window>
  <div id="t" visible="false"
    action="onshow: anima.slideDown({self}); onhide:
anima.slideUp({self})">
    <div><!-- the 2nd div is optional but sometimes it looks better
with it -->
      <groupbox>
        <caption label="slide down"/>
        Hi <textbox/>
      </groupbox>
      When? <datebox/>
    </div>
  </div>
  <button label="toggle" onClick="t.visible = !t.visible"/>
</window>

```

当然，你加载的其它库并没有这个限制。

函数	描述
<code>anima.appear(cmp)</code> <code>anima.appear(cmp, dur)</code>	通过增加透明度(opacity)来使组件可见。 <code>cmp</code> – 组件。使用 <code>#{EL-expr}</code> 标识它。 <code>dur</code> – 以毫秒为单位的持续时间(duration)。默认： 800。
<code>anima.slideDown(cmp)</code> <code>anima.slideDown(cmp, dur)</code>	以滑盖式(slide-down)效果来使组件可见。 <code>cmp</code> – 组件。使用 <code>#{EL-expr}</code> 标识它。 <code>dur</code> – 以毫秒为单位的持续时间。默认： 400。
<code>anima.slideUp(cmp)</code> <code>anima.slideUp(cmp, dur)</code>	以滑起来(slide-up)效果来使组件不可见。 <code>cmp</code> – 组件。使用 <code>#{ EL-expr }</code> 标识它。 <code>dur</code> – 以毫秒为单位的持续时间。默认： 400。
<code>anima.fade(cmp)</code> <code>anima.fade(cmp, dur)</code>	通过褪色组件不可见。 <code>cmp</code> – 组件。使用 <code>#{ EL-expr }</code> 标识它。 <code>dur</code> – 以毫秒为单位的持续时间。默认： 550。
<code>anima.puff(cmp)</code> <code>anima.puff(cmp, dur)</code>	通过膨胀(puff out)使组件不可见。 <code>cmp</code> – 组件。使用 <code>#{ EL-expr }</code> 标识它。 <code>dur</code> – 以毫秒为单位的持续时间。默认： 700。
<code>anima.dropOut(cmp)</code> <code>anima.dropOut(cmp, dur)</code>	通过褪色及放下(drop)使组件不可

函数	描述
	<p>见。</p> <p>cmp – 组件。使用 <code>{ EL-expr }</code> 标识它。</p> <p>dur – 以毫秒为单位的持续时间。默认： 700。</p>

例如，

```
<window title="Animation Effects">
  <style>.ctl {
    border: 1px outset #777; background:#ddeecc;
    margin: 2px; margin-right: 10px; padding-left: 2px;
padding-right: 2px; }
  </style>
  <label value="Slide" sclass="ctl"action="onmouseover:
anima.slideDown({t}); onmouseout: anima.slideUp({t})"/>
  <label value="Fade" sclass="ctl"action="onmouseover:
anima.appear({t}); onmouseout: anima.fade({t})"/>
  <label value="Puff" sclass="ctl"action="onmouseover:
anima.appear({t}); onmouseout: anima.puff({t})"/>
  <label value="Drop Out" sclass="ctl"action="onmouseover:
anima.appear({t}); onmouseout: anima.dropOut({t})"/>
  <div id="t" visible="false">
    <div>
      <groupbox>
        <caption label="Dynamic Content"/>
        Content to show and hide dynamically.
        <datebox/>
      </groupbox>
      Description <textbox/>
    </div>
  </div>
</window>
```

^[50] <http://script.aculo.us> provides easy-to-use, cross-browser user interface JavaScript libraries

事件

注意，是否支持一个事件取决于组件。此外，在组件内容更新之后会发送出事件。

鼠标事件

事件名称	组件	描述
onClick	button caption column div groupbox image imagemap label listcell listfooter listheader menuitem tabpanel toolbar toolbarbutton treecell treecol window	事件： org.zkoss.zk.ui.event.MouseEvent 表示用户点击了组件。
onRightClick	button caption checkbox column div groupbox image imagemap label listcell listfooter listheader listitem radio slider tab tabbox tabpanel toolbar toolbarbutton treecell	事件： org.zkoss.zk.ui.event.MouseEvent 表示用户右击了组件。

事件名称	组件	描述
	treecol treeitem window	
onDoubleClick	button caption checkbox column div groupbox image label listcell listfooter listheader listitem tabpanel toolbar treecell treecol treeitem window	<p>事件: org.zkoss.zk.ui.event.MouseEvent</p> <p>表示用户双击了组件。</p>

按键事件

事件名称	组件/描述
onOK	<p>window textbox intbox longbox doublebox decimalbox datebox timebox combobox bandbox</p> <p>事件:org.zkoss.zk.ui.event.KeyEvent</p> <p>表示用户按下了 ENTER 键。</p>
onCancel	<p>window textbox intbox longbox doublebox decimalbox datebox timebox combobox bandbox</p> <p>事件:org.zkoss.zk.ui.event.KeyEvent</p> <p>表示用户按下了一个特殊键，例如 PgUp，Home，以及和 Ctrl 或 Alt 组合键。参考下面的 ctrlKeys 属性(ctrlKeys Property)一节获取细节。</p>
onCtrlKey	<p>window textbox intbox longbox doublebox decimalbox datebox timebox combobox bandbox</p>

事件名称	组件/描述
	事件:org.zkoss.zk.ui.event.KeyEvent 表示用户按下了 ESC 键。

按键事件会被送至最近的 window，此 window 已经为指定的事件注册了事件监听器。此事件实现了 submit，cancel 和 shortcut 功能。

如下所示，当 T1 获得聚焦时，若用户按下了 ENTER 键则会调用 doA()方法。当 T2 获得聚焦时，若用户按下了 ENTER 键则会调用 doB()方法。

```
<window id="A" onOK="doA()">
  <window id="B" onOK="doB()">
    <textbox id="T1"/>
  </window>
  <textbox id="T2"/>
</window>
```

注意， window 并不接收发送至内 window 的按键事件，除非你手动提交它们。在上面的例子中，若 T1 获得了聚焦，事件不会被送至 window A，不管是否为 window B 声明了 onOK 处理器。

ctrlKeys属性

为了接收 onCtrlKey 事件，你必须指定什么按键被 ctrlKeys 属性拦截。换言之，仅在 ctrlKeys 属性内指定的按键会被送回服务器。例如，若用户按下了 Alt+C，Ctrl+A，F10，或 Ctrl+F3，则 onCtrlKey 事件会被发送出。

```
<window ctrlKeys="@c a#10 #3">
...
```

下面为 ctrlKeys 属性的语法。

键	描述					
k	一个控制键， 即 Ctrl+k， 这里 k 可以为 a~z, 0~9, #n 和 ~n。					
@k	alt 键，即 Alt+k, 这里 k 可以为 a~z, 0~9, #n 和 ~n。					
\$k	shift 键，即 Shift+k, 这里 k 可以为#n 和 ~n。					
#n	一个特殊键，如下。					
	#home	Home	#end	End	#ins	Ins ert
	#del	Del ete	#left	←	#right	→

#up	↑	#down	↓	#pgup	PgUp
#pgdn	PgDn				
#f n	功能键。 #f1, #f2, ... #f12 对应 F1, F2,... F12。				

输入事件

事件名称	组件	描述
onChange	textbox datebox decimalbox doublebox intbox combobox bandbox	事件: <code>org.zkoss.zk.ui.event.InputEvent</code> 表示输入组件的内容已经被用户修改。
onChanging	textbox datebox decimalbox doublebox intbox combobox bandbox	事件: <code>org.zkoss.zk.ui.event.InputEvent</code> 表示用户正在改变输入组件的内容。注意, 直到接收了 onChange 事件, 组件的内容(在服务器)才会改变。因此, 你必须调用 <code>InputEvent</code> 类的 <code>getValue</code> 方法才能获取到临时值。
onSelection	textbo datebox decimalbox doublebox intbox combobox bandbox	事件: <code>org.zkoss.zk.ui.event.SelectionEvent</code> 表示用户正在选择输入组件的部分文本。你可以使用 <code>getStart</code> 和 <code>getEnd</code> 方法获取选中文本的开始及结束位置。
onFocus	textbox datebox decimalbox doublebox intbox combobox bandbox button toolbarbutton checkboxradio	事件: <code>org.zkoss.zk.ui.event.Event</code> 表示一个组件获得了聚焦。 事件监听器是在服务器端执行的, 所以当 onFocus 的事件监听器执行时, 客户端的聚焦或许已经改变。

事件名称	组件	描述
onBlur	textbox datebox decimalbox doublebox intbox combobox bandbox button toolbarbutton checkbox radio	事件: <code>org.zkoss.zk.ui.event.Event</code> 表示一个组件失去了聚焦。 事件监听器是在服务器端执行的, 所以当 onBlur 的事件监听器执行时, 客户端的聚焦或许已经改变。

List和Tree 事件

事件名称	组件	描述
onSelect	listbox tabbox tab tree	事件: <code>org.zkoss.zk.ui.event.SelectEvent</code> 表示用户选择了一个或多个子组件。对于 listbox 为一套 listitem 。对于 tree 为一套 treeitem 。对于 tabbox 为一个 tab。 注: onSelect 被发送至 tab 和 tabbox。
onOpen	north east west south groupbox treeitem combobox bandbox menu popup window	事件: <code>org.zkoss.zk.ui.event.OpenEvent</code> 表示用户打开或关闭了一个组件。注: 不同于 onClose, 该事件仅是一个通知。在打开或关闭一个组件后, 客户端会发出此事件。 这对于当首次打开组件, 通过监听 onOpen 事件, 创建组件来实现随机存取(load-on-demand)是很有用的。

Slider和Scroll事件

事件名称	组件	描述
onScroll	slider	事件: <code>org.zkoss.zk.ui.event.ScrollEvent</code> 表示用户已经滚动了滚动组件的内容。

事件名称	组件	描述
onScrolling	slider	<p>事件: <code>org.zkoss.zk.ui.event.ScrollEvent</code></p> <p>表示用户正在滚动一个滚动组件的内容。注意直到接收了 <code>onScroll</code> 事件, 组件的内容(在服务器端)才会改变。因此, 你必须调用 <code>ScrollEvent</code> 类的 <code>getPos</code> 方法才能获得临时位置。</p>

其它事件

事件名称	组件	描述
onCreate	all	<p>事件: <code>org.zkoss.ui.zk.ui.event.CreateEvent</code></p> <p>表示当送出(rendering)一个 ZUML 页面时组件被创建。参考组件活动周期一章。</p>
onClose	window tab fileupload	<p>事件: <code>org.zkoss.ui.zk.ui.event.Event</code></p> <p>表示用户按下了 <code>close</code> 按钮, 然后组件会移除其自身。</p>
onDrop	all	<p>事件: <code>org.zkoss.ui.zk.ui.event.DropEvent</code></p> <p>表示另一个组件被放入了接收此事件的组建。参考拖放一节。</p>
onCheck	checkbox radio radiogroup	<p>事件: <code>org.zkoss.zk.ui.event.CheckEvent</code></p> <p>表示用户已经改变了一个组件的状态。</p> <p>注: <code>onCheck</code> 被发送至 <code>radio</code> 和 <code>radiogroup</code>。</p>
onMove	window	<p>事件: <code>org.zkoss.zk.ui.event.MoveEvent</code></p> <p>表示用户已经移动了一个组件。</p>
onSize	window	事件:

事件名称	组件	描述
		<code>org.zkoss.zk.ui.event.SizeEvent</code> 表示用户已经改变了一个组件的大小。
<code>onZIndex</code>	<code>window</code>	Event: <code>org.zkoss.zk.ui.event.ZIndexEvent</code> 表示用户已经改变了一个组件的 z-index 。
<code>onTimer</code>	<code>timer</code>	事件: <code>org.zkoss.zk.ui.event.Event</code> 表示你指定的计时器已经触发了一个事件。要想知道是哪个计时器，可以调用 <code>Event</code> 类的 <code>getTarget</code> 方法。
<code>onNotify</code>	<code>any</code>	事件: <code>org.zkoss.zk.ui.event.Event</code> 表示一个独立的应响用程序事件。它的意义取决于应用程序。目前，还没有组件发送此事件。
<code>onClientInfo</code>	<code>root</code>	事件: <code>org.zkoss.zk.ui.event.ClientInfoEvent</code> 通知一个根组件关于客户端的信息，例如时区和分辨率(resolutions)。
<code>onPiggyback</code>	<code>root</code>	事件: <code>org.zkoss.zku.ui.event.Event</code> 通知根组件客户端已经向服务器发出了一个请求。通常用于捎带非紧急的 UI 更新到客户端 (piggyback non-emergent UI updates to the client)。
<code>onBookmarkChanged</code>	<code>root</code>	事件: <code>org.zkoss.zk.ui.event.BookmarkEvent</code> 通知程序用户按下了 BACK , FORWARD 或其它引起标签(bookmark)改变的行为。
<code>onColSize</code>	<code>columns</code> <code>listhead</code> <code>treecols</code>	事件: <code>org.zkoss.zul.event.ColSizeEvent</code> 通知一组页眉(a group of headers)的父组件，用户改变了它的两个子组件的宽度。(Notifies the parent of a group of headers that the

事件名称	组件	描述
		widths of two of its children are changed by the user)
onPaging	grid listbox paging	事件： org.zkoss.zul.event.PagingEvent 通知用户选中了多页面组件的某一页面。
onUpload	fileupload	事件： org.zkoss.zul.event.UploadEvent 通知文件已被上传，应用程序可以使用 getMedia 或 getMedias 方法获取已上传的文件。
onFulfill	all	事件： org.zkoss.zul.event.FulfillEvent 通知 fulfill 事件已经被应用于目标组件。当所有的后续组件被创建后此事件会被提交。

Radio和radiogroup的事件流

为方便开发人员，onCheck事件首先被发送至radio，然后是radiogroup ^[51]。因此，你可以为radiogroup或每个radio 按钮添加监听器。

```
<radiogroup onCheck="fruit.value = self.selectedItem.label">
  <radio label="Apple"/>
  <radio label="Orange"/>
</radiogroup>
You have selected : <label id="fruit"/>
```

上面的事例和下面的有相同的效果。

```
<radiogroup>
  <radio label="Apple" onCheck="fruit.value = self.label"/>
  <radio label="Orange" onCheck="fruit.value = self.label"/>
</radiogroup>
You have selected : <label id="fruit"/>
```

^[51] 当一个radio被添加至一个radiogroup时，内部的实现是通过添加一个监听器处理的。

第 8 章 数据绑定

目录

[基本概念](#)

[添加一个数据源](#)

[建立数据绑定管理器](#)

[将UI组件关联至数据源](#)

[何时从数据源加载数据至UI](#)

[何时从UI组件保存数据至数据源](#)

[将相同的数据源关联至多个UI组件](#)

[关联UI组件和一个集合](#)

[在数据源和UI组件间定制转换](#)

[定义数据绑定管理的访问权限](#)

基本概念

数据绑定是一种机制，在 UI 组件和数据源之间自动完成数据复制的底层基础实现。

(Data binding is a mechanism that automates the data-copy plumbing codes between UI components and the data source)。应用程序开发人员只需要告诉数据绑定管理器(data -binding manager)关于 UI 组件和数据源的联系即可。然后，数据绑定管理器会自动完成加载(将数据从数据源加载至 UI 组件)并保存(将数据从 UI 组件保存至数据源)工作。

添加一个数据源

首先，我们需要定义一个数据源作为数据 bean。在这个例子中，我们使用 Person 类来存储一个人的信息，例如 first name 和 last name。

```
Person.java

public class Person {
    private String _firstName="";
    private String _lastName="";

    //getter and setters
    public void setFirstName(String firstName) {
        _firstName = firstName;
    }
    public String getFirstName() {
        return _firstName;
    }
}
```

```

    }
    public void setLastName(String lastName) {
        _lastName = lastName;
    }
    public String getLastName() {
        return _lastName;
    }
    public void setFullName(String f) {
        //do nothing.
    }
    public String getFullName() {
        return _firstName + " " + _lastName;
    }
}

```

然后在页面内声明一个数据源，如下，

```

<zscript>
    //prepare the example person object
    Person person = new Person();
    person.setFirstName("Tom");
    person.setLastName("Hanks");
</zscript>

```

建立数据绑定管理器

通过在页面顶部定义页面初始化来建立数据绑定管理器。

```

<?init
class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>

```

然后初始器类会做如下事情：

1. 创建一个 AnnotateDataBinder 实例。
2. 使用存储在组件内的 **"binder"** 将 AnnotateDataBinder 实例设置为一个变量，此组件在 arg0 "component-path"被指定(若 arg0 未指定，则使用 Page 代替)。
3. 调用 `DataBinder.loadAll()` 通过关联的数据源初始化所有的 UI 组件。

将UI组件关联至数据源

在添加数据源，创建数据绑定管理器之后，你需要定义需要的 UI 对象，并将它们关联至数据源。使用 ZUML 注释表达式来告诉数据绑定管理器数据源与 UI 组件的关系。它的使用非常简单，直接为组件的属性声明注释。

```
<component-name  
attribute-name="@{bean-name.attribute-name}"/>
```

1. component-name 描述一个 UI 组件。
2. attribute-name 描述 UI 组件或数据源的一个属性。
3. bean-name 描述一个数据源。

我们使用 Grid 作为一个例子，将它与一个 Person 实例的数据源相关联。在这个例子中，数据绑定管理器会自动同步 UI 组件与数据源。

```
<?init  
class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>  
  
<window>  
<zscript>  
    //prepare the example person object  
    Person person = new Person();  
    person.setFirstName("George");  
    person.setLastName("Bush");  
</zscript>  
  
<grid width="400px">  
    <rows>  
        <row> First Name: <textbox  
value="@{person.firstName}"/></row>  
        <row> Last Name: <textbox value="@{person.lastName}"/></row>  
        <row> Full Name: <label value="@{person.fullName}"/></row>  
    </rows>  
</grid>  
</window>
```

何时从数据源加载数据至UI

数据绑定管理器是被事件或者用户的活动触发的。因此，你必须使用 load-when 标签表达式在 ZUML 注释表达式内指定事件，以告诉数据绑定管理器何时从数据源加载数据至组件的属性。


```
<component-name attribute-name="@{bean-name.attribute-name,
load-when='component-id.event-name' }"/>
```

1. component-id 描述一个 UI 组件的 ID。
2. event-name 描述事件名称。

允许多重定义但需要依次调用。

在下面的例子中，一旦一个人的 first name 或 last name 被修改，则 fullname 会被自动更新。

当 id 为 firstName 或 lastName 的 Textbox 的值被更改时，数据绑定管理器会从 person.fullName 重新加载数据至 id 为 fullName 的 Label。换句话说，onChange 事件会被触发。

```
<?init
class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>

<window>
<zscript>
    //prepare the example person object
    Person person = new Person();
    person.setFirstName("George");
    person.setLastName("Bush");
</zscript>

<grid width="400px">
    <rows>
        <row> First Name:
            <textbox id="firstName" value="@{person.firstName}"/>
        </row>
        <row> Last Name:
            <textbox id="lastName" value="@{person.lastName}"/>
        </row>
        <row> Full Name:
            <label id="fullName" value="@{person.fullName,
                load-when='firstName.onChange,lastName.onChange' }"/>
        </row>
    </rows>
</grid>

</window>
```

何时从UI组件保存数据至数据源

数据绑定管理器是被事件或者用户的活动触发的。因此，你必须使用 `save-when` 标签表达式在 ZUML 注释表达式内指定事件，以告诉数据绑定管理器何时将组件属性值保存至数据源。

```
<component-name
attribute-name="@{bean-name.attribute-name,load-when='com
ponent-id.event-name' }"/>
```

1. `component-id` 描述一个 UI 组件的 ID。
2. `event-name` 描述事件名称。

允许多重定义但需要依次调用。

在下面的例子中，当 Textbox 自身触发了 `"onChange"` 事件时，数据绑定管理器会将 Textbox `"firstName"` 的属性 `"value"` 保存至 `"person.firstName"`。

```
<?init
class="org.zkoss.zkplus.databind.AnnotateDataBinderInit"?>

<window width="500px">
<zscript>
    Person person = new Person();
    person.setFirstName("Bill");
    person.setLastName("Gates");
</zscript>

<listbox>
    <listhead>
        <listheader label="First Name" width="100px"/>
        <listheader label="Last Name" width="100px"/>
        <listheader label="Full Name" width="100px"/>
    </listhead>
    <listitem>
        <listcell>
            <textbox id="firstName" value="@{person.firstName,
save-when='self.onChange' }"/>
        </listcell>
        <listcell>
            <textbox id="lastName" value="@{person.lastName,
save-when='self.onChange' }"/>
        </listcell>
        <listcell label="@{person.fullName}"/>
    </listitem>
</listbox>
```

```
</listitem>
</listbox>
</window>
```

将相同的数据源关联至多个UI组件

一个数据源可以和多个 UI 组件相关联。一旦数据源被修改，数据绑定管理器会自动更新这些被关联的组件。

在下面的例子中，我们使用 ZUML 注释表达式将一个数据源 Person 的实例

"selected"和多个UI组件关联,包括Listbox 和Grid。一旦用户选择了Listbox 的一个项目，Grid 就会展示被选中 person 相应的信息。

```
<?init
class="org.zkoss.zkplus.databind.AnnotateDataBinderInit"?>

<window width="500px">
<zscript>
    //prepare the example person
    Person selected = new Person();
</zscript>

<listbox rows="4" selectedItem="@{selected}">
<listhead>
    <listheader label="First Name" width="100px"/>
    <listheader label="Last Name" width="100px"/>
    <listheader label="Full Name" width="100px"/>
</listhead>
<listitem>
    <listcell label="George"/>
    <listcell label="Bush"/>
</listitem>
<listitem>
    <listcell label="Bill"/>
    <listcell label="Gates"/>
</listitem>
</listbox>

    <!-- show the detail of the selected person -->
<grid>
    <rows>
        <row>First Name: <textbox
value="@{selected.firstName}"/></row>
```

```
<row>Last Name: <textbox  
value="@{selected.lastName}"/></row>  
</rows>  
</grid>  
</window>
```

关联UI组件和一个集合

将一个集合和一个组件关联是非常有用的，数据绑定管理器会将集合转换为相应的 UI 组件。

1. 准备集合的数据源。
2. 将集合和那些支持的 UI 组件，例如 Listbox, Grid, 和 Tree 的 model 属性相关联。
3. 定义一个 UI 组件模板。
 - a. 使用 self 属性定义一个你想要的变量，用来呈现集合内的每个实例。

```
<component-name self="@{each='variable-name'}"/>
```

variable-name 仅对于 component-name 和它的子组件是可见的。

- b. 将 UI 组件和变量关联。

```
<component-name  
attribute-name="@{variable-name.attribute-name}  
"/>
```

在下面的例子中，我们将展示如何将一个集合和 Listbox 关联来显示 person 列表。

```
<?init  
class="org.zkoss.zkplus.databind.AnnotateDataBinderInit"?>  
  
<window width="500px">  
<zscript>  
    //prepare the example persons List  
    int count = 30;  
    List persons = new ArrayList();  
    for(int j= 0; j < count; ++j) {  
        Person personx = new Person();  
        personx.setFirstName("Tom"+j);  
        personx.setLastName("Hanks"+j);  
  
        persons.add(personx);  
    }  
</zscript>  
</window>
```

```

</zscript>

<listbox rows="4" model="@{persons}">
<listhead>
  <listheader label="First Name" width="100px"/>
  <listheader label="Last Name" width="100px"/>
  <listheader label="Full Name" width="100px"/>
</listhead>
<listitem self="@{each='person'}">
  <listcell>
    <textbox value="@{person.firstName}"/>
  </listcell>
  <listcell>
    <textbox value="@{person.lastName}"/>
  </listcell>
  <listcell label="@{person.fullName}"/>
</listitem>
</listbox>
</window>

```

在数据源和UI组件间定制转换

如果你想自己处理数据源和 UI 组件间的转换，可以在 `converter` 标签表达式中指定转换器的类名，以告诉数据绑定管理器使用你自己的方式来处理数据源和 UI 组件间的转换。

```

<component-name
attribute-name="@{bean-name.attribute-name,converter='class-name' }"/>

```

不允许多重定义，后面的定义会重写前面的定义。

1. 定义一个实现了 `TypeConverter` 的类，并实现下面的方法
 - a. `coerceToUI`，将一个值对象转换成 UI 组件属性类型。
 - b. `coerceToBean`，将一个值对象转换成 `bean` 属性类型。
2. 在 `converter` 标签表达式内指定转换器的类名。

在下面的例子中，我们将展示如何将一个 `boolean` 值转换成不同的图像以代替纯文本。

`myTypeConverter` 将 `boolean` 转换成相应的不同图像。

```

import org.zkoss.zkplus.databind.TypeConverter;
import org.zkoss.zul.Listcell;

```

```

public class myTypeConverter implements TypeConverter {
public Object coerceToBean(java.lang.Object val,
org.zkoss.zk.ui.Component comp)    {
    return null;
}

public Object coerceToUi(java.lang.Object val,
org.zkoss.zk.ui.Component comp)

{
    boolean married = (Boolean) val;
    if (married)
        ((Listcell) comp).setImage("/img/true.png");
    else
        ((Listcell) comp).setImage("/img/false.png");
    return null;
}
}

```

使用 `convert` 标签表达式指定 `myTypeConverter` 与 `Person` 实例的 `married` 属性关联。

```

<?init
class="org.zkoss.zkplus.databind.AnnotateDataBinderInit"?>

<window width="500px">
<zscript><![CDATA[
    //prepare the example persons List
    List persons = new ArrayList();
    persons.add(new Person("Tom", "Yeh", true));
    persons.add(new Person("Henri", "Chen", true));
    persons.add(new Person("Jumper", "Chen", false));
    persons.add(new Person("Robbie", "Cheng", false));
]]>
</zscript>

<listbox rows="4" model="@{persons}">
    <listhead>
        <listheader label="First Name" width="100px" />
        <listheader label="Last Name" width="100px" />
        <listheader label="Married" width="100px" />
    </listhead>
    <listitem self="@{each=person}">
        <listcell label="@{person.firstName}" />
        <listcell label="@{person.lastName}" />
    </listitem>
</listbox>

```

```
<listcell label="@{person.married,
converter='myTypeConverter' }"/>
</listitem>
</listbox>
</window>
```

定义数据绑定管理的访问权限

为了更好的控制数据绑定管理器，你可以设置的 `attribute-name` 的模式为

`both` (加载/保存), `load` (仅加载), `save` (仅保存), 或 `none` (都不可)。

```
<component-name
attribute-name="@{bean-name.attribute-name,access='type-name' }"/>
```

1. `type-name` 描述特定的访问模式。

不允许多重定义，后面的定义会重写前面的定义。

在下面的例子中，如果 `Textbox`, `"firstName"`, 和 `"lastName"` 的值被修改，`Listcell`, `"fullname"` 的值仍不会改变，因为数据绑定管理器被设置为禁止修改。

```
<?init
class="org.zkoss.zkplus.databind.AnnotateDataBinderInit"?>

<window width="500px">
<zscript>
    Person person = new Person();
    person.setFirstName("Bill");
    person.setLastName("Gates");
</zscript>

<listbox>
    <listhead>
        <listheader label="First Name" width="100px"/>
        <listheader label="Last Name" width="100px"/>
        <listheader label="Full Name" width="100px"/>
    </listhead>
    <listitem>
        <listcell>
```

```
<textbox id="firstName" value="@{person.firstName}"/>
</listcell>
<listcell>
<textbox id="lastName" value="@{person.lastName}"/>
</listcell>
<listcell id="fullName" label="@{person.fullName,
access='none'}"/>
<listitem>
</listbox>
</window>
```

第 9 章 在ZUML中使用XHTML组件集

目录

目标

[有效的XHTML页面即为有效的ZUM页面](#)
[以服务器为中心的交互](#)
[像平常一样使用Servlet](#)

差异

[为每个标签创建一个组件](#)
[UUID即为ID](#)
[所有标签都有效](#)
[大小写](#)
[无模型支持](#)

浏览器端的DOM树

[TABLE和TBODY标签](#)

事件

与JSF, JSP及其它的集成

[使用已存在的Servlet](#)
[使用包含丰富](#)
[丰富一个静态的HTML页面](#)
[使用ZK JSP标签](#)
[使用ZK JSF组件](#)
[使用ZK Filter丰富动态生成的页面](#)

本章描述了 XHTML 组件集。

目标

介绍 XHTML 组件集的目的是，使已存在的 Web 页面可以简单的移植(port)到 ZUML。最终目标是，所有有效的 XHTML 页面即为有效的 ZUML 页面。所有的 servlet 就像平常一样处理提交的表单。

因此，已存在的 XHTML 页面可以享受 ZUML 页面拥有的最强大优点，丰富用户界面。这种丰富性可以用两种方式实现。第一种，你可以嵌入 Java 代码来动态操纵 XHTML 组件。第二种，你可以为已存在的页面添加 off-of-shelf XUL 组件，就像为 XUL 页面添加 XHTML。

[性能考虑]：如果 HTML 标签部分是静态的，则最好使用 Native 命名空间，就像在性能提示(Performance Tips)一章中描述的那样。

有效的XHTML页面即为有效的ZUM页面

下面为一个简单但典型的 Web 页面。

```
<html>
<head>
<title>ZHTML Demo</title>
</head>
<body>
  <h1>ZHTML Demo</h1>
  <ul id="ul">
    <li>The first item.</li>
    <li>The second item.</li>
  </ul>
  <input type="button" value="Add Item" />
  <br/>
  <input id="inp0" type="text" /> +
  <input id="inp1" type="text" /> =
  <text id="out" />
</body>
</html>
```

将其以 zhtml 为扩展名^[52]，则 ZK 加载器会将此页面作为 ZUML 页面解释。然后，org.zkoss.zhtml.Html, org.zkoss.zhtml.Head 及其它的实例被创建。换言之，我们在服务器端创建了一个 XHTML 组件树。然后 ZK 将它们成为 (renders) 规则的 XHTML 页面并送回浏览器，就像我们为任何 ZUML 页面做的那样。

以服务器为中心的交互

作为一个 ZUML 页面，它可以嵌入任何 Java 代码，且在服务器执行它们，如下。

```
<html xmlns:zk="http://www.zkoss.org/2005/zk">
<head>
<title>ZHTML Demo</title>
</head>
```

```

<body>
  <h1>ZHTML Demo</h1>
  <ul id="ul">
    <li>The first item.</li>
    <li>The second item.</li>
  </ul>
  <input type="button" value="Add Item" zk:onClick="addItem()" />
  <br/>
  <input id="inp0" type="text" zk:onChange="add()" /> +
  <input id="inp1" type="text" zk:onChange="add()" /> =
  <text id="out" />
  <zscript> void addItem() {
    Component li = new Raw("li");
    li.setParent(ul);
    new Text("Item "+ul.getChildren().size()).setParent(li);
  }
  void add() {
    out.setValue(inp0.getValue() + inp1.getValue());
  }
</zscript>
</body></html>

```

在上面的例子中，我们使用了 **ZK** 命名空间指定 `onClick` 属性。这是必须的，因为 **XHTML** 本身有一个相同名称的属性。

注意 **Java** 代码是运行在服务器的。因此，不同于嵌入 **HTML** 页面的 **JavaScript**，你可以直接在服务器访问任何资源。例如，你可以打开一个数据库的连接，并获取数据来填充特定的组件。

```

<zscript>
import java.sql.*;
void addItem() {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  String url = "jdbc:odbc:Fred";
  Connection conn = DriverManager.getConnection(url,"myLogin",
"myPassword");
  ...
  conn.close();
}
</zscript>

```

像平常一样使用Servlet

在传统的 Web 应用程序中，XHTML 页面通常会提交一个表单到指定的 servlet 进行处理。你不需要修改 servlet 来将页面融入 ZK(port the page to ZK)。

^[52] 如果你想所有的XHTML页面变为ZUML页面,可以将.html扩展名映射到DHtmlLayoutServlet。参考the Developer's Reference的附录A(Appendix A)获取细节。

差异

除了作为ZK组件，XHTML组件集的实现与其它的组件集^[53]有某些不同，这样会更容易将传统的XHTML页面融入ZK。

为每个标签创建一个组件

ZK 加载器会为在 ZUML 页面内声明的每个标签创建一个 ZK 组件。例如，在下面的 ZUML 页面内创建了四个组件(html, body, p 和 label)。

```
<html>
  <body>
    <p>Hi</p>
  </body>
</html>
```

优点是你动态改变任何组件的内容：

```
<p id="info">Hi</p>
<z:button onClick="info.detach()"
xmlns:z="http://www.zkoss.org/2005/zk"/>
```

但是，这会消耗更多的处理时间，且会需要更多的内存来容纳这些组件，所以，如果页面的一部分是静态的，你可以按如下方式使用 **Native** 命名空间。

```
<n:html xmlns:n="http://www.zkoss.org/2005/zk/native">
  <n:body>
    <p id="info">Hi</p>
    <z:button onClick="info.detach()"
xmlns:z="http://www.zkoss.org/2005/zk"/>
  </n:body>
```

```
</n:html>
```

参考性能提示一章获取更多信息。

UUID即为ID

传统的 servlets 和 JavaScript 代码通常依赖于 id 属性，所以 XHTML 组件的 UUID 与 ID 是相同的。因此，开发人员不需要更改已存在的代码来适应 ZK，如下所示。

```
<img id="which"/>
<script type="text/javascript"><![CDATA[
//JavaScript and running at the browser
function change() {
    var el = document.getElementById("which");
    el.src = "something.gif";
}
]]></script>
<zscript><!-- Java and running at the server -->
void change() {
    which.src = "another.gif";
}
</zscript>
```

注意，UUID 是不可变的，并且与组件的 ID 没有任何关系，而不是 XHTML(Notice that UUID is immutable and nothing to do with ID for components other than XHTML)。因此，在上面的例子中，若使用了 XUL 组件则会失败。若你真想在 JavaScript 中使用 XUL 组件，则必须使用 EL 表达式获取正确的 UUID。

```
<input id="which"/>
<script type="text/javascript">//Running at the browser
    var el = document.getElementById("${which.uuid}");
    el = $e("${which.uuid}"); // $e() is an utility of ZK Client Engine
</script>
```

副作用

由于 UUID 即为 ID，所以同一桌面内的两个组件不能有相同的 ID。

所有标签都有效

不同于XUL或其它组件集，在XHTML组件集中没有无效的XML元素。ZK使用 `org.zkoss.zhtml.Raw` 类用于构造任何未经认证的xml元素^[54]。因此，开发人员可以使用目标浏览器支持的任意标签，不管ZK组件是否实现了它们。

类似的，你可以使用 `Raw` 组件来创建任何在 `XHTML` 组件集中没有定义的组件，如下。

```
new Raw("object"); //object could be any tag name the target browser supports
```

大小写

不同于 XUL 或其它组件集，XHTML 的组件名是区分大小写的。下面的 XML 元素都被映射到 `org.zkoss.zhtml.Br` 组件。

```
<br/>
<BR/>
<bR/>
```

无模型支持

XHTML 组件直接输出其内容。它们不支持模型。换言之，`model` 属性会被忽略。

^[53] 这些差异是由于实现了特定的接口，所以你可以为你的组件应用 (`apply`) 相同的效果，若果你喜欢的话。

^[54] 注：这由 `org.zkoss.zk.ui.ext.DynamicTag` 接口处理。

浏览器端的DOM树

在将 XHTML 移植到 ZK 后，你不需要使用 JavaScript 操纵浏览器端的 DOM 树，尽管 ZK 并不阻止你那么做。相反，你在服务器操纵 XHTML 组件，然后 ZK 引擎会为你更新浏览器的 DOM 树。

这很方便但是有一个 `catch`。ZK 假定浏览器端的 DOM 树与服务器端的组件树是一样的。在大多数情况下，这是事实。但并非总是如此。

TABLE和TBODY标签

浏览器总是在 TABLE 和 TR 间创建 TBODY。因此，下面的两个表格拥有相同的结构。

```
<table>
  <tr><td>Hi</td></tr>
</table>
<table>
  <tbody>
    <tr><td>Hi</td></tr>
  </tbody>
</table>
```

不幸的是，在 ZK 中它们的组件树是不同的。因此，若你想动态的操纵一个表格，就必须在 TABLE 和 TR 间声明 TBODY 。当然，你不需要为静态表格担心这个。

事件

所有的XHTML组件都支持下列组件，但是是否适用还得依赖于浏览器。例如，onChange对于非输入组件是没有意义的，例如body和div。你必须考虑HTML标准[\[55\]](http://www.w3c.org)。

事件名称	组件	描述
onChange	all	事件: org.zkoss.zk.ui.event.InputEvent 表示用户修改了输入组件的内容。
onClick	all	事件: org.zkoss.zk.ui.event.MouseEvent 表示用户点击了组件。

[\[55\]](http://www.w3c.org) <http://www.w3c.org>

与JSF, JSP及其它的集成

当与已存在的 Web 页面集成时，你或许会问自己几个问题。

1. 已存在的页面是静态的还是动态生成的？
2. 若丰富已存在的页面，是轻微的增强(minor enhancement)？或是，重写页面的一部分？

3. 当添加一个新页面时，使用 XUL 作为默认的组件集？

依你的请求，有几种途径可以选择。

使用已存在的Servlet

通过将使用 FORM 包裹 ZUL 组件，你可以将 ZUML 组件的内容作为请求参数提交到一个已存在的 servlet。参考 ZUML 页面及 XUL 组件集一章中用 HTML FORM 和 Java Servlets 一节获取细节。

使用此方法，你可以设计丰富的用户界面而无需修改依存在的 servlet，仅是简单的使用 ZUL 组件替换一些内容。

使用包含丰富

若你想重写已存在页面的一部分，把重写的部分放入一个单独的 ZUML 文件或许会更好。然后，你可以在已存在的页面内包含此 ZUML 文件。例如，若使用了 JSP 技术则可以使用 `jsp:include`。

```
<jsp:include page="/my/ria.zul"/>
```

丰富一个静态的HTML页面

若你想通过添加丰富的内容来直接修改一个静态的HTML页面，你可以使用Native命名空间(<http://www.zkoss.org/2005/native>) 来指定静态内容，然后为需要东逃改变的内容使用XUL 或XHTML组件集 (<http://www.zkoss.org/2005/zul> 和 <http://www.w3.org/1999/xhtml>) 。

```
<!-- test.zhtml -->
<html xmlns="http://www.zkoss.org/2005/native"
xmlns:h="http://www.w3.org/1999/xhtml"
      xmlns:z="http://www.zkoss.org/2005/zul">
<head>
  <title>Hi ZK</title>
</head>
<body>
  <h:ul id="list"><!-- dynamically changeable -->
    <h:li><z:textbox/></h:li>
  </h:ul>
  <p>Static content</p>
</body>
</html>
```

使用ZK JSP标签

ZK 为每个 ZUL 组件都提供了等价的 JSP 标签。若你喜欢在 JSP 页面使用 ZUL 组件，仅需简单的跟随下面的步骤。

1. 在 JSP 页面内使用 taglib 指令指定 `http://www.zkoss.org/jsp/zul` 作为 TLD 文件的 URI。
2. 尽管是可选的，最好指定 DOCTYPE 为 XHTML 1.0 Transitional，因为 ZK 客户端引擎默认使用它。
3. ZK JSP 页面的必须以 `page` 标签(ZK 的 `org.zkoss.zul.jsp.PageTag`)开始，即表示此为一个 JSP 文件。

下面为一个简单的例子。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<%@ taglib uri="/WEB-INF/tld/zul/jsp/zul2jsp.tld" prefix="z" %>
<html>
  <head>
    <title>Test of ZUL on JSP</title>
  </head>
  <body>
    <h1>1. Header outside z:page</h1><z:page>
    <h2>2. Header in z:page</h2>
    <z:window title="Test" border="normal">
      <p>3. Content in z:window</p>
    </z:window>
    <p>4. Content in z:page after z:window</h2></z:page>
    <p>5. Content after z:page</p>
  </body>
</html>
```

每个 ZUL 组件都被一个 JSP 标签包装，且每个属性都被一个标签属性包装。此外，混合使用 JSP 标签与其它标签是你的自由。因此，使用 ZK JSP 标签更直接。

使用ZK JSF组件

ZK为每个ZUL组件提供了对应的JSF组件。使用方法类似于ZK JSP标签，只是taglib URI 为 <http://www.zkoss.org/jsf/zul>。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```



```

        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.d
d">
<html>
<head>
<title>Validator Example</title>
</head>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://www.zkoss.org/jsf/zul" prefix="z"%>
<body>
<f:view>
    <h:form id="helloForm">
        <z:page>
            <z:window z:title="Validator Example" width="500px"
border="normal">
                --Validate input day must in weekend--<br/>
                <z:datebox id="dbox" format="yyyy/MM/dd"
                    f:value="#{ValidatorBean.value}"
                    f:validator="#{ValidatorBean.validateDate}"/>
                <h:message
                    style="color: red; font-style: oblique;"
                    for="dbox" />
                <br/>
                <h:commandButton id="submit"
action="#{ValidatorBean.doSubmit}" value="Submit" />
            </z:window>
            <h:messages/>
        </z:page>
    </h:form>
    <a href="../index.html">Back</a>
</f:view>
</body>
</html>

```

使用ZK Filter丰富动态生成的页面

若你想 ajax-ize 一个动态生成的 HTML 页面(例如, 输出一个 JSP 页面), 可以使用 ZK Filter 处理生成的页面。你必须使用 web.xml 启用 ZK filter, 如下所示。

```

<filter>
    <filter-name>zkFilter</filter-name>

```

```

<filter-class>org.zkoss.zk.ui.http.DHtmlLayoutFilter</filter-c
lass>
  <init-param>
    <param-name>extension</param-name>
    <param-value>html</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>zkFilter</filter-name>
  <url-pattern>/my/dyna.jsp</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>zkFilter</filter-name>
  <url-pattern>/my/dyna/*</url-pattern>
</filter-mapping>

```

此处，url-pattern 是独立于应用程序的。extension 参数(init-param)定义了动态输出的语言。默认为 html。若为 xul/html，则指定 xul 作为扩展名。

提示：在大多数情况下，ZK JSP 标签更易于使用且比 ZK filter 消耗更少的内存。参考性能提示一章中。

注意，若你想过滤来自 include 和/或 forward 的输出，要使用 REQUEST 和/或 INCLUDE 指定适配器(dispatcher)元素。查阅 Java Servlet 规范获取细节。例如，

```

<filter-mapping>  <filter-name>zkFilter</filter-name>
<url-pattern>/my/dyna/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>ERROR</dispatcher>
</filter-mapping>

```

第 10 章 宏组件

目录

[使用宏组件的三个步骤](#)

[第一步：实现](#)

[第二步：实现](#)

[第三步：使用](#)

[内联宏](#)

[一个例子](#)

[常规宏](#)

[宏组件和ID空间](#)

[增设方法](#)

有两种方式实现一个组件。一种方法是实现一个来自 `org.zkoss.zk.ui.AbstractComponent` 的类。另一种方法是使用其它组件实现它。

第一种方式更为灵活。这需要深入 ZK，所以通常由组件开发人员来完成。这在 [Component Development Guide](#) 中有讨论。

另外，使用其它组件来实现一个新组件更为直接。就像 `composition`，`macro expansion`，或 `inline replacement` 一样工作。为了描述方便，我们将这类组件成为宏组件(`macro components`)，而其它的组件称为本地组件(`native components`)。

[提示]：从应用程序开发人员的角度来看，宏组件与本地组件是没有区别的，除了宏组件是被实现的。

使用宏组件的三个步骤

使用宏组件需要以下三步。

1. 通过 ZUML 页面实现一个宏组件。
2. 在将要使用它的页面内声明宏组件。
3. 使用宏组件，就像使用其它组件一样。

[提示]：除了在页面内定义宏组件，你可以将其定义放入一个语言插件(`language addon`)，这样所有的页面都可以访问此宏组件。

第一步. 实现

所有你要做的就是准备一个 ZUML 页面来描述宏组件的组成。也就是，宏模板。

例如，假定你想将一个 `label` 和一个 `textbox` 包装成一个宏组件。我们可以创建页面，例如 `/WEB-INF/macros/username.zul`，如下。

```
<hbox>
  Username: <textbox/>
</hbox>
```

完成！

实现宏组件的页面和其它页面是一样的，所以任何页面都可以做为宏组件被使用。

第二步 . 实现

在实例化一个宏组件之前，你必须首先声明。一个简单的方式就是使用指令声明它。

```
<?component name="username"
macroURI="/WEB-INF/macros/username.zul"?>
```

如上所示，你必须声明 **name**(name 属性)和页面的 **URI**(macroURI 属性)。

其它属性

除了name, macroURI和class [\[56\]](#)属性，你可以指定一个初始属性的列表，用于实例化时初始一个组件。

```
<?component name="mycomp" macroURI="/macros/mycomp.zul"
myprop="myval" another="anotherval"?>
```

因此，

```
<mycomp/>
```

等价于

```
<mycomp myprop="myval1" another="anotherval"/>
```

第三步. 使用

使用宏组件与使用其它组件一样。

```
<window>
  <username/>
</window>
```

传递属性

就像一个普通的组件，当使用宏组件时你可以指定属性(properties)(亦=attributes) ， 如下。

```
<?component name="username"
macro-uri="/WEB-INF/macros/username.zul"?>
<window>
  <username who="John"/>
```

```
</window>
```

所有这些被指定的属性被存储在一个 `map` 中，然后通过 `arg` 变量被传递到模板 (`template`)。然后在模板中，你可以按如下方式访问这些属性。

```
<hbox>
  Username: <textbox value="{arg.who}"/>
</hbox>
```

[注]: 仅当送出(`rendering`)宏页面时 `arg` 是可用的。为了访问事件监听器，你必须使用 `getDynamicProperty` 代替。细节请参考增设方法一节。

arg.includer

除了指定的属性(`properties`)(亦=`attributes`)，`arg.includer` 属性也被传出用以呈现定义在宏模板中组件的父组件。

若一个常规的宏被创建，`arg.includer` 即为宏组件本身。若创建了一个内联(`inline`)宏，`arg.includer` 为父组件，若果有的话。更多信息请参考内联宏一节。

在上面的例子中，`arg.includer` 表示常规的宏组件，`<username who="John"/>`，且为的父组件(定义在 `username.zul` 中)。

[\[56\]](#) 后面会讨论`class` 属性。

内联宏

有两种类型的宏组件：内联^{[\[57\]](#)}和常规。默认为常规宏。为了指定内联宏，你必须在 `component` 指令内指定`inline="true"`。

内联宏的行为就像内联扩展(`inline-expansion`)。若遇见了内联宏，则 `ZK` 不会创建宏组件。相反，它会内联扩展定义在宏 `URI` 内的组件。换言之，就好象直接将内联组件的内容的嵌入(`type`)到目标页面。

use.zul: (目标页面)	等价页面:
<pre><?component name="username" inline="true" macro-uri="username.zul"?> <grid> <rows></pre>	<pre><grid> <rows> <row> Username</pre>

<pre> <username id="ua" name="John"/> </rows> </grid> username.zul: (宏定义) <row> Username <textbox id="\${arg.id}" value="\${arg.name}"/> </row> </pre>	<pre> <textbox id="ua" value="John"/> </row> </rows> </grid> </pre> <p>所有的属性，包括 id，都会被传递到内联宏。</p>
--	---

另外，ZK 将会创建一个真实的组件(称为宏组件)来显示常规的宏。也就是说，宏组件将会作为定义在宏内组件的父组件被创建。

内联宏更易整合到复杂的页面。例如，由于 rows 仅接受 row，所以在前一个例子中，你不能使用常规的组件，而不是宏组件(not macro components)。由于在同一 ID 空间内，所以更易访问定义在宏内的所有组件。这也意味着开发人员必须清楚如何实现以避免名字冲突。

常规宏允许开发人员提供额外的 API，并且对组件用户隐藏实现。每个常规宏组件均是一个 ID 空间所有者，所以不会有名字冲突。通常假定常规宏的用户不知道如何实现。相反，他们使用定义好的 API 访问宏组件。

一个例子

inline.zul: (宏定义)

```

<row>
  <textbox value="${arg.col1}"/>
  <textbox value="${arg.col2}"/>
</row>

```

useinline.zul: (目标页面)

```

<?component name="myrow" macro-uri="inline.zul" inline="true"?>
<window title="Test of inline macros" border="normal">
  <zscript><![CDATA[
    import org.zkoss.util.Pair;
    List infos = new LinkedList();
    for (int j = 0; j < 10; ++j) {
      infos.add(new Pair("A" + j, "B" + j));
    }
  ]]></zscript>

```

```
<grid>
  <rows>
    <myrow col1="{each.x}" col2="{each.y}"
forEach="{infos}" />
  </rows>
</grid>
</window>
```

[\[57\]](#) 自ZK 2.3 版本后添加了内联宏组件。

常规宏

ZK 创建一个真实的组件(称为宏组件)来显示常规宏，即前一章节描述的。

为了描述方便，当在此章节中讨论的宏组件时，即意味着为常规宏组件。

宏组件和ID空间

就像 window，宏组件为一个 ID 空间所有者。换言之，在实现宏组件(亦=宏组件的子组件)的页面内使用什么标识来标识组件是自由的。它们不会和定义在使用宏组件的同一页面内的组件相冲突的。

例如，假定我们有如下一个宏定义。

```
<hbox>
  Username: <textbox id="who" value="{arg.who}" />
</hbox>
```

那么下面的代码将会正常工作。

```
<?component name="username"
macro-uri="/WEB-INF/macros/username.zul"?>
<zkl>
  <username/>
  <button id="who"/> <!-- no conflict because it is in a different
ID space -->
</zkl>
```

但是，下列的代码将不会工作。

```
<?component name="username"
macro-uri="/WEB-INF/macros/username.zul"?>
```

```
<username id="who"/>
```

为什么呢？就像任何 ID 空间所有者，宏组件本身和其子组件在相同的 ID 空间内。有两种可选的解决方案：

1. 为宏组件的子组件标识使用一个特殊的前缀。例如使用"mc_who"代替"who"。

```
<hbox>
  Username: <textbox id="mc_who" value="${arg.who}"/>
</hbox>
```

1. 使用 window 组件创建一个额外的 ID 空间。

```
<window>
  <hbox>
    Username: <textbox id="who" value="${arg.who}"/>
  </hbox>
</window>
```

将使用第一种方案 T，简单方便。

从外部访问子组件

就像其它的 ID 空间所有者，你可以调用 `getFellow` 方法或使用 `org.zkoss.zk.ui.Path` 来访问其子组件 (by use of two `getFellow` method invocations or `org.zkoss.zk.ui.Path`)。

例如，假定你有一个 ID 为 "username" 的宏组件，那么可以按如下方式访问 `textbox`。

```
comp.getFellow("username").getFellow("mc_who");
new Path("/username/mc_who");
```

访问定义在 **Ancestor** 的变量

宏组件像内联扩展一样工作。因此，就像其它组件，(一个宏组件的)子组件可以访问任何定义在父组件 ID 空间内的变量。

例如，`username` 的子组件可以直接访问 `v`。

```
<zscript>
  String v = "something";
</zscript>
<username/>
```


但是，并不推荐这样使用(it is not recommended to utilize such visibility)，因为这或许会限制宏的使用范围。

运行时改变macro-uri

你可以动态的改变宏的 URI，如下。

```
<username id="ua"/>
<button onClick="ua.setMacroURI(&quot;another.zul&quot;);" />
```

增设方法

宏组件实现了 `org.zkoss.zk.ui.ext.DynamicPropertie` 接口，所以你可以按如下方式使用 `getDynamicProperty` 方法访问其属性。

```
<username id="ua" who="John"/>
<button label="what?"
onClick="alert(ua.getDynamicProperty(&quot;who&quot;))" />
```

显然使用 `DynamicPropertied` 是很繁琐的。更糟的是，若你使用 `setDynamicProperty` 改变一个属性，却不会被改变宏的子组件。例如，下面的代码会显示 John 作为 **username**，而不是 Mary。

```
<username id="ua" who="John"/>
<zscript>
    ua.setDynamicProperty("who", "Mary");
</zscript>
```

为什么呢？一个宏组件的所有子组件都是在创建宏组件时被创建的，除非你手动操作这些子组件^[58]，否则是不会改变它们的。调用 `setDynamicProperty` 仅会影响存储在宏组件中的属性(可以使用 `getDynamicProperties` 获取的)。textbox的内容仍没有改变。

因此，最好增设一个方法，例如 `setWho`，用以直接操作宏组建组件。为了增设你自己的方法，必须为宏组件实现一个 `class`，然后使用 `component` 指令的 `class` 属性指定此类。

[提示]：可以使用 `recreate` 方法来再创建(recreate) 子组件(包括其当前的属性)。实际上此方法是移除了所有的子组件，然后再重新创建它们。

有两种方式实现一个类。下面的章节描述了细节。

在java中增设方法

为宏组件增设方法需要两个步骤。

```
1. 继承 org.zkoss.zk.ui.HtmlMacroComponent 实现一个类。
2. //Username.java
3. package mypack;
4. public class Username extends HtmlMacroComponent {
5.     public void setWho(String name) {
6.         setDynamicProperty("who", name); //arg.who requires
        it
7.         final Textbox tb = (Textbox)getFellow("mc_who");
8.         if (tb != null) tb.setValue(name); //correct the child
        if available
9.     }
10.    public String getWho() {
11.        return (String)getDynamicProperty("who");
12.    }
13. }
```

- 正如上面所描述的，你必须在 setWho 内调用 setDynamicProperty，因为在宏页面({arg.who})内引用了 {arg.who}，{arg.who}被用于宏组件创建其子组件时。
- 由于 setWho 方法或许会在宏组件创建其子组件之前被创建，因此你必须检查 mc_who 是否存在。
- 由于调用了 mc_who 的 setValue，当调用 setWho 时，客户端的内容及视觉表现(visual presentation)都会被自动更新。

14.使用 class 属性在宏声明内声明类。

```
<?component name="username"
macro-uri="/WEB-INF/macros/username.zul"
class="mypack.Username"?>
```

在zscript中增设方法

除了使用 Java 文件实现，你也可以在 zscript 中实现 Java class(es)。优点是不需要编译(compilation)，并且可以动态的修改其内容(无需重新部署 Web 应用程序)。缺点是降低了性能且容易出现打字错误(prone to typos)。

需要几个步骤在 zscript 中实现 Java class 。

1. 你需要为要实现的类准备一个 zscript 文件，例如/zs/username.zs。
注意你可以在相用的 zscript 文件内放置任意数量的类及函数。

```
2. //username.zs
```

```

3.  package mypack;
4.  public class Username extends HtmlMacroComponent {
5.      public void setWho(String name) {
6.          setDynamicProperty("who", name);
7.          Textbox tb = getFellow("mc_who");
8.          if (tb != null) tb.setValue(name);
9.      }
10.     public String getWho() {
11.         return getDynamicProperty("who");
12.     }
13. }

```

14. 使用 `init` 指令加载 `zscript` 文件，然后声明组件。

```

15. <?init zscript="/zs/username.zs"?>
16. <?component name="username"
    macro-uri="/WEB-INF/macros/username.zul"
17.     class="mypack.Username"?>

```

实现类(前一个例子中的 `mypack.Username`)直到宏组件被使用时才会被决定(resolved)，所以使用 `zscript` 元素为 `zscript` 文件赋值(evaluate)是没有问题的。

```

<?component name="username"
macro-uri="/WEB-INF/macros/username.zul"
    class="mypack.Username"?>
<zk>
    <zscript src="/zs/username.zs"/>
    <username/>
</zk>

```

尽管主观(subjective)，`init` 指令仍更具有可读性。

当实例化时重写实现类

就像其它的任何组件，你可以为任何特定的实例使用 `use` 属性重写用于实现宏组件的类。

```

<?component name="username"
macro-uri="/WEB-INF/macros/username.zul"
    class="mypack.Username"?>

<username use="another.MyAnotherUsername/>

```

当然在上面的例子中，你必须提供一个 `another.MyAnotherUsername` 的实现。然后再一次，可以使用单独的 `Java` 文件或 `zscript` 来实现此类。

手动创建一个宏组件

为了手动创建一个宏组件，你必须在所有的初始化完成后调用 `afterCompose` 方法，如下。

```
HtmlMacroComponent ua = (HtmlMacroComponent)
    page.getComponentDefinition("username",
false).newInstance(page);
ua.setParent(wnd);
ua.applyProperties(); //apply properties defined in the component
definition
ua.setDynamicProperty("who", "Joe");
ua.afterCompose(); //then the ZUML page is loaded and child
components are created
```

[注]: `getComponentDefinition` 方法被用于在一个页面内查到组件定义。

若你为宏实现了一个类，例如 `Username`，那么可按如下方式处理。

```
Username ua = new Username();
ua.setWho("Joe");
ua.setParent(wnd);
ua.afterCompose();
```

[58] 另外，在提交(rendering)阶段，`include`组件包含的子组件被创建。而且，每一次使`include`组件无效时，所有的子组件都会被移除并且创建。

第 11 章 高级特性

目录

[标识页面](#)

[表示组件](#)

[组件路径](#)

[排序](#)

[浏览器的信息及控制](#)

[onClientInfo事件](#)

[org.zkoss.ui.util.Clients 类](#)

[防止用户关闭窗口](#)

[浏览器的历史管理](#)

[添加合适的状态到浏览器历史](#)

- [监听onBookmarkChange事件并据此操作桌面](#)
 - [为iframe使用书签功能](#)
 - [一个简单的事例](#)
- [组件克隆](#)
- [组件序列化](#)
 - [序列化会话](#)
 - [序列化监听器](#)
- [跨页面通信](#)
 - [提交和发送事件](#)
 - [属性](#)
- [跨Web应用程序通信](#)
 - [来自路径的Web资源](#)
- [注释](#)
 - [注释ZUML页面](#)
 - [手动注释组件](#)
 - [获取注释](#)
- [Richlets](#)
 - [实现org.zkoss.zk.ui.Richlet接口](#)
 - [配置web.xml 和zk.xml](#)
- [会话超时管理](#)
- [错误处理](#)
 - [加载页面时的错误处理](#)
 - [更新页面时的错误处理](#)
- [其它](#)
 - [配置ZK加载器不压缩输出](#)

这一章将会描述关于组件和页面的高级主题。

标识页面

所有同一 `desktop` 内的页面都可以在一个事件监听器内被访问到。你可以使用 `org.zkoss.zk.ui.Component` 接口内的 `getPage` 方法获取组件的当 `page`。

为了获取另一 `page` 的引用，你必须为 `page` 指定一个标识，用以查找。

```
<?page id="another"?>
...
```

然后可以使用 `org.zkoss.zk.ui.Desktop` 接口内的 `getPage` 方法，如下。

```
<zscript>
    Page another = self.getDesktop().getPage("another");
</zscript>
```

表示组件

组件通过 ID 空间被分组。page 本身即为一个 ID 空间。window 组件是另外一个 ID 空间。假定你有一个名为 P 的 page，此 page 有一个名为 A 的窗口，A 窗口有一个子窗口 B。那么，若你想获取窗口 B 的一个子组件，例如 C，可以按如下方式处理。

```
comp.getDesktop().getPage("P").getFellow("A").getFellow("B").getFellow("C");
```

getFellow 方法用于获取如同一空间内的任何伙伴。ID 空间的概念请参考基础一章中 ID 空间一节。

组件路径

就像文件系统中的路径，组件路径为沿着 ID 空间排列的组件 ID(a component path is a catenation of IDs of components along ID spaces)。在上面的例子中，路径为 "/A/B/C"。换言之，组件路径的根为当前 page。若你想标识另一个 page，必须使用 "/"。在上面的例子中，路径也可以被表示为 "//P/A/B/C"。

org.zkoss.zk.ui.Path 类，就像 java.io.File，被用来简化操纵组件路径。因此，下列的语句的等价于上面的例子。

```
Path.getComponent("/A/B/C"); //assume the current page is P
Path.getComponent("//P/A/B/C");
```

除了静态方法，你可以初始化一个 Path 实例。

```
Path parent = new Path("//P/A");
new Path(parent, "B/C").getComponent();
```

提示: Path.getComponent("//xyz") 总会返回 null，因为//之后的标识为页面 ID。而页面不是组件。

排序

org.zkoss.zk.ui.Component 接口中的 getChildren 方法返回的列表是实况的(live)。org.zkoss.zul.ListBox 中的 getItems 方法及其它接口中的对应方法也是如此。换言之，你可以动态操纵其内容。例如，下面的语句是等价的：

```
comp.getChildren().remove(0);
((Component)comp.getChildren().get(0)).setParent(null);
```

但是，你不能够使用 `java.util.Collections` 类的 `sort` 方法为它们排序。原因是很微妙的(subtle)：当你将列表添加到另一位置时，`children` 列表会自动从原始的位置移除一个 `child`。例如，下面的代码实际上移动了第一个 `child` 前的第二个 `child`。

```
comp.getChildren().add(0, comp.getChildren().get(1));
```

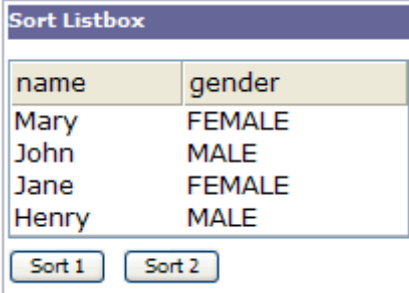
这和平常的列表(例如 `LinkedList`)有些不同，所以 `Collections` 的 `sort` 方法是不会工作的。

为了简化组件的排序，我们提供了 `org.zkoss.zk.ui.Components` 类的 `sort` 方法来处理 `children` 列表。

在下面的例子中，我们使用 `sort` 方法和 `org.zkoss.zul.ListitemComparator` 来为一个 `listbox` 排序。

注意，这仅是用来说明 `listbox` 直接支持 `listitem` 的排序。详情请参考 [ZUML](#) 页面及 [XUL 组件集一章中列表框一节](#)。

```
<window title="Sort Listbox" border="normal" width="200px">
  <vbox>
    <listbox id="l">
      <listhead>
        <listheader label="name"/>
        <listheader label="gender"/>
      </listhead>
      <listitem>
        <listcell label="Mary"/>
        <listcell label="FEMALE"/>
      </listitem>
      <listitem>
        <listcell label="John"/>
        <listcell label="MALE"/>
      </listitem>
      <listitem>
        <listcell label="Jane"/>
        <listcell label="FEMALE"/>
      </listitem>
      <listitem>
        <listcell label="Henry"/>
        <listcell label="MALE"/>
      </listitem>
    </listbox>
    <hbox>
      <button label="Sort 1" onClick="sort(l, 0)"/>
      <button label="Sort 2" onClick="sort(l, 1)"/>
    </hbox>
  </vbox>
</window>
```



name	gender
Mary	FEMALE
John	MALE
Jane	FEMALE
Henry	MALE

```

        </hbox>
    </vbox>
    <zscript>
    void sort(Listbox l, int j) {
        Components.sort(l.getItems(), new ListitemComparator(j));
    }
    </zscript>
</window>

```

浏览器的信息及控制

若要获取客户端的信息,你可以在根组件为 **onClientInfo** 事件注册一个事件监听器。为了控制客户端的行为,你可以使用 `org.zkoss.zk.ui.util.Clients` 类的功能。

onClientInfo事件

有时,一个应用程序需要知道客户端的信息,例如时区。那么,你可以为 **onClientInfo** 事件添加一个事件监听器。一旦添加了事件,客户端将会寄回一个 `org.zkoss.zk.ui.event.ClientInfoEvent` 类的实例,从此类你可以获取客户端的信息。

```

<grid onClientInfo="onClientInfo(event)">
    <rows>
        <row>Time Zone <label id="tm"/></row>
        <row>Screen <label id="scrn"/></row>
    </rows>
    <zscript>
    void onClientInfo(ClientInfoEvent evt) {
        tm.setValue(evt.getTimeZone().toString());
        scrn.setValue(

evt.getScreenWidth()+"x"+evt.getScreenHeight()+"x"+evt.getColorDepth());
    }
    </zscript>
</grid>

```

[注]: **onClientInfo** 事件仅对于根组件(亦=没有任何父组件的组件)是有意义的。

ZK 不会存储客户端的信息,所以若有需要的话你需要手动存储。由于会话(session)是与相同的客户端相关联的,所以你可以将客户端信息存储在会话属性中。


```
session.setAttribute("px_preferred_time_zone",
event.getTimeZone());
```

注意，若你将时区信息存储在名为 `px_preferred_time_zone` 的会话变量中，那么此后此值将会成为默认的时区。详情请参考国际化一章中关于时区一节。

注意，在页面被提交(rendered)(且送至客户端)后，`onClientInfo` 事件会从客户端被发出。因此，若你的部分组件数据依赖于客户端的信息，例如时区，你可以要求客户端重新发送请求，如下。

```
import org.zkoss.util.TimeZones;
...
if (!TimeZones.getCurrent().equals(event.getTimeZone()))
    Executions.sendRedirect(null);
```

org.zkoss.ui.util.Clients 类

用于控制客户端视觉表现(visual presentation)(更确切的说，为浏览器窗口)的功能(utilities)被集中放置于 `org.zkoss.ui.util.Clients`。例如，你可以滚动浏览器窗口(亦=桌面)，如下。

```
Clients.scrollBy(100, 0);
```

防止用户关闭窗口

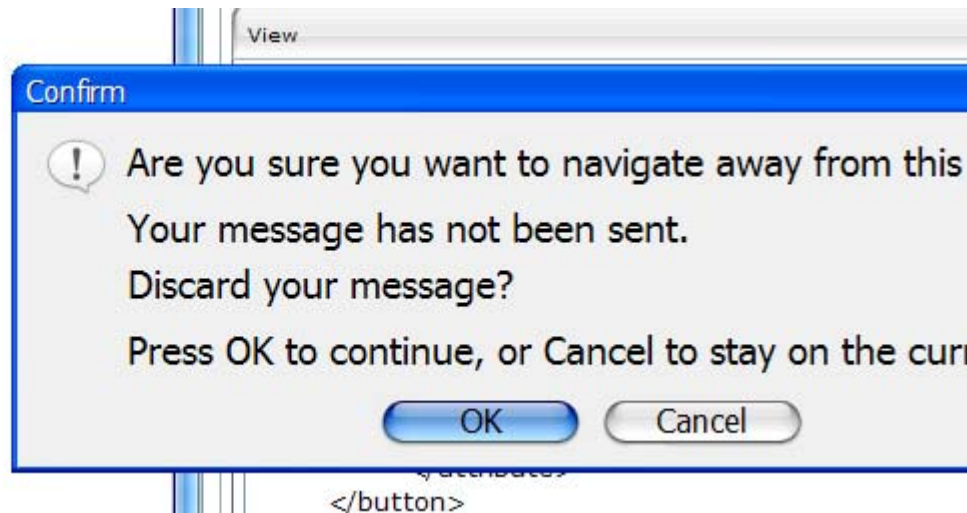
某些情况下，你或许想防止，或至少，警告用户，当他试图关闭窗口或浏览另一个网页时。例如，当用户正在编写一封未保存的邮件时。

```
if (mail.isDirty()) {
    Clients.confirmClose("Your message has not been sent.\nDiscard your message?");
} else {
    Clients.confirmClose(null);
}
```

一旦调用了 `confirmClose` 方法且参数为非空字符串(with a non-empty string)，当用户试图关闭浏览器窗口，重载(reload)，或浏览另一个 URL 时时就会显示一个确认对话框。

浏览器的历史管理

-Input
-The onChange event
-Radio and Checkboxes
-Comboboxes
-More comboboxes
-Bandboxes
-Sliders
-The onScrolling event
- Modal Dialogs
-Messagebox
-Fileupload
-Modal dialog
- Layout Elements
-The box model



在传统的多页面 Web 应用程序中，用户通常会使用 BACK 或 FORWARD 按钮来在多页面间冲浪(surf)，并将他

们制作成书签(bookmark)以备日后使用。使用 ZK，你仍然可以使用多页面来呈现一套不同的特性及信息，就像在传统的 Web 应用程序中那样。

但是，对于 ZK 应用程序来说，在一个桌面内显示很多特性是很平常的，而在传统 Web 应用程序中则通常需要多个 Web 页面。为使用户冲浪更简单，ZK 支持浏览器的历史管理，ZK 应用程序可以简单的在服务器端管理浏览器的历史。

概念很简单。将合适的桌面状态项目(items for appropriate states of a desktop)添加到浏览器的历史中，那么用户可以在同一桌面的不同状态间使用 BACK 和 FORWARD 按钮冲浪。当用户在这些状态间冲浪时，onBookmarkChanged 事件会被发出以通知应用程序。

从应用程序的角度来看，管理浏览器的历史需要两步：

1. 为桌面每个合适的状态添加一个项目到浏览器的历史。
2. 监听 onBookmarkChanged 事件并据此操作桌面。

添加合适的状态到浏览器历史

你的应用程序必须决定把什么合适的状态添加到浏览器历史。例如，在一个多步操作中，每个状态都是一个很好的候选，用以添加到浏览器历史，这样用户可以在这些状态间跳转或将他们制作成书签以备日后使用。

一旦决定了将一个状态添加到浏览器历史，当合适时你可以简单的调用 `org.zkoss.zk.ui.Desktop` 接口的 `setBookmark` 方法。将一个状态添加到浏览器历史称为 **bookmarking**。注意并不是用户添加到浏览器的书签(亦=IE 中我的最爱)。

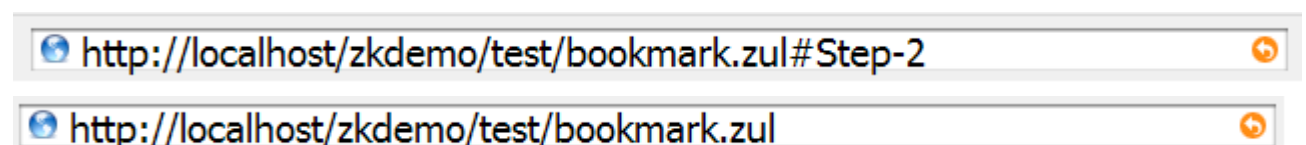
[提示]: 你可以将在服务器端添加书签成为服务器书签，以区别于浏览器书签。

例如，假定你想当点击 **Next** 按钮时将状态制为书签(bookmark the state)，可以按如下方式处理。

```
<button label="Next"
onClick="desktop.setBookmark('&quot;Step-2&quot;');" />
```

若你关注 URL，会发现 ZK 为 URL 添加了 `#Step-2`。

若你按下按钮，则会看到，



监听 `onBookmarkChange` 事件并据此操作桌面

添加一个状态到浏览器历史后，用户可以在这些状态间冲浪，例如按下 **BACK** 按钮返回前一状态。当状态改变时，ZK 会通过广播 `onBookmarkChange` 事件 (`org.zkoss.zk.ui.event.BookmarkEvent` 类的一个实例)到桌面内所有的根组件来通知应用程序。

不同于传统的多页面应用程序，当状态改变时，你必须手动操作 ZK 桌面。反映展示标签的状态是应用程序开发人员的工作。

为监听 `onBookmarkChange` 事件，你可以为桌面内的任何页面添加事件监听器，或任一根组件。

```
<window onBookmarkChange="goto(event.bookmark)">
  <zscript>
    void goto(String bookmark) {
      if ("Step-2".equals(bookmark)) {
        ...//create components for Step 2
      } else { //empty bookmark
        ...//create components for Step 1
      }
    }
  </zscript>
</window>
```

就像处理其它的事件，当接收 `onBookmarkChange` 事件后，你可以按你想的方式操纵桌面。典型的方法是使用 `org.zkoss.zk.ui.Executions` 类的 `createComponents` 方法。换言之，你可以使用 ZUML 页面呈现每个状态，然后当接收 `onBookmarkChange` 时，使用 `createComponents` 在其中创建所有的组件。

```
if ("Step-2".equals(bookmark)) {
    //1. Remove components, if any, representing the previous state
    try {
        self.getFellow("replacable").detach();
    } catch (ComponentNotFoundException ex) {
        //not created yet
    }

    //2. Creates components belonging to Step 2
    Executions.createComponents("/bk/step2.zul", self, null);
}
```

为iframe使用书签功能

如果一个页面包含一个或多个 `iframe` 组件，则有时标记 `iframe` 的状态是比较好的选择。例如，当被包含的 `iframe` 被转向另一个 URL 时，你可以改变页面的书签(容器)，这样你可以恢复 `iframe` 的内容。为了达到这样的效果，你需要监听 `onURICHange` 事件，如下。

```
<window onURICHange="desktop.bookmark =
storeURI(event.getTarget(), event.getURI())">
    <iframe src="{uri_depends_on_bookmark}"
forward="onURICHange"/>
</window>
```

一个简单的事例

在这个例子中，我们为没一个 `tab` 选项制作一个书签。

```
<window id="wnd" title="Bookmark Demo" width="400px"
border="normal">
    <zscript>
        page.addEventListener("onBookmarkChange",
            new EventListener() {
                public void onEvent(Event event) throws UiException
            {
                try {
```

```

wnd.getFellow(wnd.desktop.bookmark).setSelected(true);
        } catch (ComponentNotFoundException ex) {
            tab1.setSelected(true);
        }
    }
});
</zscript>

<tabbox id="tbox" width="100%" onSelect="desktop.bookmark =
self.selectedTab.id">
    <tabs>
        <tab id="tab1" label="Tab 1"/>
        <tab id="tab2" label="Tab 2"/>
        <tab id="tab3" label="Tab 3"/>
    </tabs>
    <tabpaneles>
        <tabpanel>This is panel 1</tabpanel>
        <tabpanel>This is panel 2</tabpanel>
        <tabpanel>This is panel 3</tabpanel>
    </tabpaneles>
</tabbox>
</window>

```

组件克隆

所有的组件都是可克隆的。换言之，它们都实现了 `java.lang.Cloneable` 接口。因此，复制组件是很简单的，如下。

```

<vbox id="vb">
    <listbox id="src" multiple="true" width="200px">
        <listhead>
            <listheader label="Population"/>
            <listheader align="right" label="%"/>
        </listhead>
        <listitem value="A">
            <listcell label="A. Graduate"/>
            <listcell label="20%"/>
        </listitem>
        <listitem value="B">
            <listcell label="B. College"/>
            <listcell label="23%"/>
        </listitem>
        <listitem value="C">

```

```

        <listcell label="C. High School"/>
        <listcell label="40%"/>
    </listitem>
</listbox>
<zscript>
int cnt = 0;
</zscript>
<button label="Clone">
    <attribute name="onClick">
        Listbox l = src.clone();
        l.setId("dst" + ++cnt);
        vb.insertBefore(l, self);
    </attribute>
</button>
</vbox>

```

1. 一旦组件被克隆了，所有它的子组件也都会被克隆(all its children and descendants)。
2. 被克隆的组件并不属于任何页面和父组件。换言之，`src.clone().getParent()` 会返回 `null`。
3. ID 并未改变，若你想将被克隆的组件添加回相同的 ID 空间，要记住更改 ID。

组件序列化

所有组件都是可序列化的(serializable)，所以你可以为内存或其它存储器序列化组件，之后再拆解它们(so you can serialize components to the memory or other storage and de-serialize them later)。就像克隆，被拆解的组件不属于另一个页面(和桌面)(another page (and desktop))。它们也独立于被序列化的组件。如下所述，序列化可以被用于实现相似的克隆功能。

```

<vbox id="vb">
    <listbox id="src" multiple="true" width="200px">
        <listhead>
            <listheader label="Population"/>
            <listheader align="right" label="%" />
        </listhead>
        <listitem value="A">
            <listcell label="A. Graduate"/>
            <listcell label="20%"/>
        </listitem>
        <listitem value="B">
            <listcell label="B. College"/>
            <listcell label="23%"/>
        </listitem>
    </listbox>
</vbox>

```

```

        <listitem value="C">
            <listcell label="C. High School"/>
            <listcell label="40%"/>
        </listitem>
    </listbox>
    <zscript>
    int cnt = 0;
    </zscript>
    <button label="Clone">
        <attribute name="onClick">
            import java.io.*;
            ByteArrayOutputStream boa = new ByteArrayOutputStream();
            new ObjectOutputStream(boa).writeObject(src);
            Listbox l = new ObjectInputStream(

                new
ByteArrayInputStream(boa.toByteArray())) .readObject();
            l.setId("dst" + ++cnt);
            vb.insertBefore(l, self);
        </attribute>
    </button>
</vbox>

```

当然，使用 clone 方法克隆会有更好的性能，而序列化组件可以被用于不同的机器之间(crossing different machines)。

序列化会话

默认情况下，一个非序列化的实现会被用于表示一个会话 (org.zkoss.zk.ui.Session)。使用非序列化实现的好处是不需要担心存储在一个组件内的值，例如 Listitem's setValue，是否为可序列化的。

但是，若你想确认存储在组件内的所有值都是可序列化的，可以使用一个序列化的实现表示一个会话。

为了配置 ZK 使用序列化实现，你需要在 WEB-INF/zk.xml 内配置

factory-class 元素，细节请参考 the Developer's Reference 的附录 B(Appendix B)。

序列化监听器

存储在一个组件，页面，桌面或会话内的属性，变量和监听器也会被序列化，如果它们是可序列化的(且相应的组件，页面，桌面或会话会被序列化)。

为简化可序列化对象的实现，ZK 会在序列化前和拆解之后调用序列化监听器，若实现了特定的接口。例如，你可以按如下方式为一个组件实现事件监听器。

```
public MyListener
implements EventListener, java.io.Serializable,
ComponentSerializationListener {
    private transient Component _target; //no need to serialize it

    //ComponentSerializationListener//
    public willSerialize(Component comp) {
    }
    public didDeserialize(Component comp) {
        _target = comp; //restore it back
    }
}
```

org.zkoss.zk.ui.util.ComponentSerializationListener 接口被用于序列化一个组件时。类似的，PageSerializationListener，DesktopSerializationListen 和 SessionSerializationListener 被分别用于序列化一个页面，桌面和会话时。

跨页面通信

在同一桌面内不同页面间通信是很直接的。首先，可以使用事件来互相通知。其次，可以使用属性共享数据。

提交和发送事件

你可以在同一桌面的不同页面间通信。通信方式是使用 postEvent 或 sendEvent 通知目标页面的组件。

```
Events.postEvent(new Event("SomethingHappens",
    comp.getDesktop().getPage("another").getFellow("main"));
```


属性

每个组件，页面，桌面，会话和 Web 应用程序都有一个独立的属性映射。这是一个在组件，页面，桌面，甚至会话间共享数据的好地方。

在 zscript 和 EL 表达式中，你可以使用隐含对象：`componentScope`，`pageScope`，`desktopScope`，`sessionScope`，`requestScope` 和 `applicationScope`。

在一个 Java 文件中，你可以使用相应类中的属性相关方法来访问它们。你也可以使用作用域(scope)参数来标识你想访问的作用域。下面的两条语句是等价的，假定 `comp` 为一个组件。

```
comp.getAttribute("some", comp.DESKTOP_SCOPE);  
comp.getDesktop().getAttribute("some");
```

跨Web应用程序通信

一个 EAR 文件可以包含多个 WAR 文件。每个 WAR 都为 Web 应用程序。在两个 Web 应用程序间通信没有标准的方法。

但是，ZK 支持从另一个 Web 应用程序引用文件。例如，假定你想从另一个 Web 应用程序，例如 `app2`，包含一个资源，例如 `/foreign.zul`。那么，你可以按如下方式处理。

```
<include src="~app2/foreign.zul"/>
```

类似的，你可以从另一个 Web 应用程序引用一个样式表。

```
<style src="~app2/foreign.css"/>
```

[注]：是否能够访问到另一个 Web 应用程序指定位置的资源依赖于 Web 服务器的配置。例如，若使用 Tomcat，你必须在 `conf/context.xml` 内指定

`crossContext="true"`。

来自路径的Web资源

使用 ZK，你可以引用由 `classpath` 定位的资源。这样做的好处是可以将 Web 资源嵌入 JAR 文件，简化了部署。

```

```

然后，它会通过从 `classpath` 寻找资源在 `/web` 目录定位资源，`/my/jar.gif`。

注释

注释提供了关于一个组件的数据，这些数据并不属于组件本身。它们在所注释组件的操作上并没有直接的影响。而是，它们在运行时被检测(examine)，主要由工具或管理者(a tool or a manager)使用。注释的内容和意义完全取决于开发人员使用的工具或管理者。例如，一个数据绑定(data-binding)管理者可以检测注释，以知道组件值要被存储的数据源。

注释ZUML页面

注释可被用于 ZUML 页面中组件和属性的声明。有两种注释方式：标准的方式和简单的方式(the classic way and the simple way)。选择哪种看你的喜好。若喜欢的话可以在同一页面内混合使用它们。

注释组件声明的标准方式

注释要放在你想注释的元素声明之前：

```
<window xmlns:a="http://www.zkoss.org/2005/zk/annotation">
  <vbox>
    <a:author name="John Magic" date="3/17/2006"/>
    <listbox>
    </listbox>
    ...
  </vbox>
</window>
```

annotation为 <http://www.zkoss.org/2005/zk/annotation>空间内的一个元素。元素的名字和属性可以是依赖于你所使用工具的一切。你可以使用几个注释为相同的组件声明注释：

```
<a:author name="John Magic"/>
<a:editor name="Mary White" date="4/11/2006"/>
<listbox/>
```

author 和 editor 为注释的名称，而 name 和 date 为属性名称。换言之，注释由名称和属性映射组成。

若两个注释有相同的名称，则它们会被合并为一个注释。例如，

```
<a:define var1="auto"/>
```

```
<a:define var2="123"/>
<listbox/>
```

等价于

```
<a:define var1="auto" var2="123"/>
<listbox/>
```

[注]: 注释不支持 EL 表达式。

注释属性(Property)声明的标准方式

为注释一个属性声明, 你可以将注释放置于属性声明之前, 如下所示。

```
<listitem a:bind="datasource='author',name='name'"
value="${author.name}"/>
```

或者, 你可以使用 `attribute` 元素, 然后简单的注释属性声明, 类似于为组件声明注释。换言之, 上面的注释等价于下面的注释:

```
<listitem>
  <a:bind datasource="author" name="name"/>
  <attribute name="value">${author.name}</attribute>
</listitem>
```

注: 若忽略了属性名称, 则名称为 `value`。例如,

```
<listitem a:bind="value='selected'" value=""/>
```

等价于

```
<listitem a:bind="selected" value=""/>
```

注释属性声明的简单方式

除了如上描述的使用特定 XMI 命名空间进行注释, 有一种简单的注释属性的方法: 为要注释的属性使用注释表达式指定一个值, 如下所示。

```
<listitem label="@{bind(datasource='author',selected)}"/>
```

注释表达式的格式为 `@{ annot-name (attr-name1 = attr-value1, attr-name2=attr-value2) }`。换言之, 若属性值为一个注释表达式, 则会被认

为是为相应属性的注释，而不是其值。在上面的例子中，名为 bind 的注释为 label 属性注释。因此，其等价于

```
<listitem a:bind=" datasource='author',selected" label="" />
```

若没有指定注释名称，则假定名称为 default。例如，下面的代码片断使用了 default 作为注释名称为 label 属性注释，且注释有一个属性，名称和值分别为 value 和 selected.name。

```
<listitem label="@{selected.name}" />
```

换言之，等价于下面的代码片断：

```
<listitem label="@{default(value='selected.name')}" />
```

注：你可以使用多个注释为相同的属性注释，如下所示。

```
<listitem label="@{ann1(selected.name)  
ann2(attr2a='attr2a',attr2b)}" />
```

注释组件声明的简单方式

类似的，你可以通过为一个名为 self 的特定属性指定注释表达来注释一个组件。

```
<listitem self="@{bind(each=person)}" />
```

self 为一个关键字，表示注释被用于注释组件声明，而不是任何属性。换言之，等价于

```
<a:bind each="person" />  
<listitem />
```

手动注释组件

通过使用 org.zkoss.zk.ui.sys.ComponentCtrl 接口的 addAnnotation 方法，你可以在运行时注释一个组件。

```
Listbox listbox = new Listbox();  
listbox.addAnnotation("some", null);
```

获取注释

可以在运行时取回(retrieved back)注释。通常由工具获取，例如数据绑定管理者，而不是应用程序。换言之，为某一特定目的，应用程序注释 ZUML 页面以告诉工具如何处理组件。

下面为转储(dump)一个组件所有注释的例子：

```
void dump(StringBuffer sb, Component comp) {
    ComponentCtrl compCtrl = (ComponentCtrl)comp;
    sb.append(comp.getId()).append(": ")
      .append(compCtrl .getAnnotations()).append('\n');

    for (Iterator it =
compCtrl.getAnnotatedProperties().iterator(); it.hasNext();) {
        String prop = it.next();
        sb.append(" with ").append(prop).append(": ")
          .append(compCtrl .getAnnotations(prop)).append('\n');
    }
}
```

Richlets

richlet 为一个小型的 Java 程序，可以创建所有必须的组件以响应用户的请求。

当用户请求一个 URL 内容时，ZK 加载器(loader)会检查指定的 URL 资源是一个 ZUML 页面，还是一个 richlet。若为 ZUML 页面， ZK 加载器会基于 ZUML 页面的内容自动创建组件，就像在前面章节描述的那样。

若资源为一个 richlet， ZK 加载器会将处理交给 richlet。创建什么组件与如何创建组件都由 richlet 处理。换言之，编程创建所有需要的组件以响应用户请求是开发人员的工作。

选择 ZUML 页面还是 richlet 取决于你的偏好。对于大多数开发人员，ZUML 页面更具有可读性和简洁性。

实现一个 richlet 是很直接的。首先，实现 org.zkoss.zk.ui.Richlet 接口，然后声明 richlet 的注释及一个 URL(association of the richlet with an URL)。

实现org.zkoss.zk.ui.Richlet接口

所有的 richlet 都必须实现 org.zkoss.zk.ui.Richlet 接口。为将实现所有方法的影响降至最低,可以继承 org.zkoss.zk.ui.GenericRichlet 类代替。然后,当请求指定的 URL 时, service 方法会被调用,然后你可创建用户界面。

```
package org.zkoss.zkdemo;import org.zkoss.zk.ui.Page;import
org.zkoss.zk.ui.GenericRichlet;
import org.zkoss.zk.ui.event.*;
import org.zkoss.zul.*;
public class TestRichlet extends GenericRichlet
{
    //Richlet//
    public void service(Page page) {
        page.setTitle("Richlet Test");
        final Window w = new Window("Richlet Test", "normal", false);
        new Label("Hello World!").setParent(w);
        final Label l = new Label();
        l.setParent(w);
        final Button b = new Button("Change");
        b.addEventListener(Events.ON_CLICK,
            new EventListener() {
                int count;
                public void onEvent(Event evt) {
                    l.setValue("" + ++count);
                }
            });
        b.setParent(w);
        w.setPage(page);
    }
}
```

就像 servlet, 你可以实现 init 和 destroy 方法以在加载时初始化和销毁 richlet。就像 servlet, richlet 被加载一次, 且为关联 URL 的所有请求服务(serves all requests for the URL it is associated with)。

每个URL一个Richlet

就像 servlet, 一个 richlet 被创建且为相同的 URL 所共享。换言之, richlet(至少为 service 方法)必须为线程安全的。另外, 组件是不共享的。每个桌面都一套独立的组件。因此, 将组件作为一个 richlet 的数据成员来存储通常并不是一个好主意。

有许多方式来解决这个问题。一个典型的方法是使用另一个类来处理每个桌面的组件, 如下所述。

```

class MyApp { //one per desktop
    Window _main;
    MyApp(Page page) {
        _main = new Window();
        _main.setPage(page);
    }
}

class MyRichlet extends GenericRichlet {
    public void service(Page page) {
        new MyApp(page); //create and forget
    }
}

```

配置web.xml 和zk.xml

在实现 richlet 之后，你可以在 zk.xml 中使用下列语句定义 richlet。

```

<richlet>
    <richlet-name>Test</richlet-name>
    <richlet-class>org.zkoss.zkdemo.TestRichlet</richlet-class>
</richlet>

```

一旦声明了一个 richlet，你可以使用 richlet-mapping 将其映射到任意数量的 URL，如下所示。

```

<richlet-mapping>
    <richlet-name>Test</richlet-name>
    <url-pattern>/test</url-pattern>
</richlet-mapping>
<richlet-mapping>
    <richlet-name>Test</richlet-name>
    <url-pattern>/some/more/*</url-pattern>
</richlet-mapping>

```

默认情况下，richlet 是不可用的。为启用 richlet，你必须在 web.xml 中添加下列声明。一旦启用了 richlet，你可以添加任意多的 richlet 而无需再修改 web.xml。

```

<servlet-mapping>
    <servlet-name>zkLoader</servlet-name>
    <url-pattern>/zk/*</url-pattern>
</servlet-mapping>

```

然后，你可以访问 <http://localhost/zk/test> 来请求 richlet。

在 `url-pattern` 元素内指定的 URL 必须以 / 开始。若 URL 以 /* 开始，那么它会匹配所有相同前缀的请求。为获取真实的请求，你可以检查由当前页面的 `getRequestPath` 方法返回的值。

```
public void service(Page page) {
    if ("/some/more/hi".equals(page.getRequestPath())) {
        ...
    }
}
```

[提示]: 通过为 `url-pattern` 指定 /*，你可以将所有不匹配的 URL 映射到被映射的 richlet (you can map all unmatched URL to the mapped richlet)。

会话超时管理

当会话超时之后，所有它属于的桌面都会被移除。若用户继续访问已经不存在的桌面，浏览器端会显示一条错误信息来提醒用户。

有时最好转到另一个页面，这样可以给用户更易理解的描述，引导他们转向其它资源，或要求他们重新登录。你可以在 `WEB-INF` 目录下的 `zk.xml` 文件内指定目标 URI，即超时的时候重定向的路径。例如，目标 URI 为 `/timeout.zul`，那么你可以将下列几行添加到 `zk.xml`。

```
<device-config>
  <device-type>ajax</device-type>
  <timeout-uri>/timeout.zul</timeout-uri>
</device-config>
```

[提示]: 每个设备(device)都有一个确切的超时 URI。更多关于 `zk.xml` 信息请参考 [the Developer's Reference](#) 的附录 B(Appendix B)。

除了配置 `zk.xml`，你可以手动更改重定向的 URI，如下。

```
Devices.setTimeoutURI("ajax", "/timeout.zul");
```

关于 Device: `device` 代表客户端 device。每个桌面都关联一个 device，反之亦然 (A device represents the client device. Each desktop is associated with one device, and vice versa)。

若你更喜欢重载页面，而不是重定向到其它的 URI，可以指定一个空的 URI，如下。

```
<device-config>
  <device-type>ajax</device-type>
  <timeout-uri></timeout-uri>
</device-config>
```


错误处理

ZK Web 应用程序可以指定错误发生时该做什么。错误是由异常引起的，而应用程序没有捕获到这个异常。

在两种情况下可能会抛出异常：加载页面和更新页面^[59]。

加载页面时的错误处理

若加载 ZUML 页面时抛出了一个未捕获的异常，此异常会直接由 Web 服务器处理。换言之，这和其它页面的处理没有区别，例如 JSP。

默认情况下，Web 服务器会使用一个错误页面来显示错误信息及栈跟踪(stack trace)。

HTTP Status 500 -

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

```
com.potix.zk.ui.UiException: Recursive import: /test/import.zul
    com.potix.zk.ui.metainfo.Parser.parse(Parser.java:200)
    com.potix.zk.ui.metainfo.Parser.parse(Parser.java:90)
    com.potix.zk.ui.metainfo.PageDefinitions$MyLoader.parse(Pag
    com.potix.web.util.resource.ResourceLoader.load(ResourceLoa
    com.potix.util.resource.ResourceCache$Info.load(ResourceCac
    com.potix.util.resource.ResourceCache$Info.<init>(ResourceC
    com.potix.util.resource.ResourceCache.get(ResourceCache.jav
```

通过在 WEB-INF/web.xml 文件内指定错误页面，你可以定制错误处理，如下。细节请参考 Java Servlet Specification。

```
<!-- web.xml -->
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/WEB-INF/sys/error.zul</location>
</error-page>
```

那么，当加载页面时发生一个错误，Web 服务器会转向你指定的错误页面，`/error/error.zul`。转发之后(Upon forwarding)，Web 服务器会立即将一套请求属性传递到错误页面以描述发生了什么。这些属性如下。

请求属性	类型
<code>javax.servlet.error.status_code</code>	<code>java.lang.Integer</code>
<code>javax.servlet.error.exception_type</code>	<code>java.lang.Class</code>
<code>javax.servlet.error.message</code>	<code>java.lang.String</code>
<code>javax.servlet.error.exception</code>	<code>java.lang.Throwable</code>
<code>javax.servlet.error.request_uri</code>	<code>java.lang.String</code>
<code>javax.servlet.error.servlet_name</code>	<code>java.lang.String</code>

然后，在错误页面内，通过使用这些属性，你可以显示你的定制信息。例如，

```
<window title="Error
${requestScope['javax.servlet.error.status_code']}">
    Cause: ${requestScope['javax.servlet.error.message']}
</window>
```

[提示]：错误页面可以为任何类型的 `servlet`。除了 `ZUL`，可以使用 `JSP` 或任何你喜欢的页面。

[提示]：转发之后， 错误页面会被作为主页面展示，所以你不需要指定主窗口指定模态(modal)或 overlapped 模式(mode)(如果有的话)。

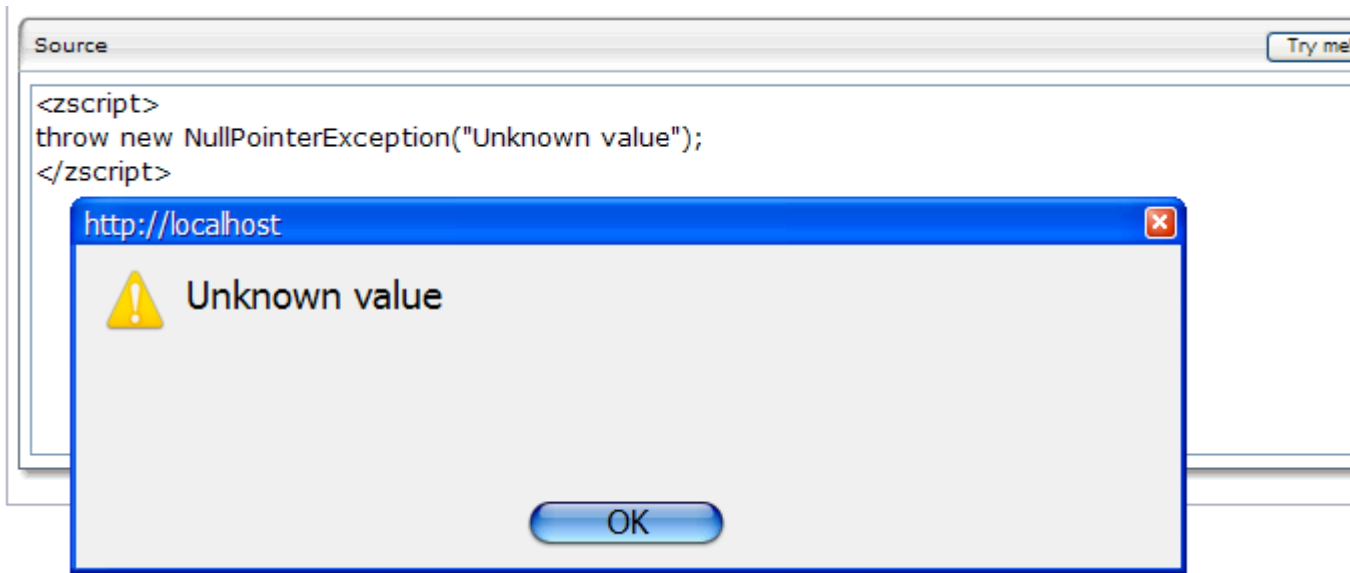
ZK Mobile错误处理

`Servlet 2.x (web.xml)`没有 `device` 类型的概念。因此，若想在相同的服务器端同时支持 `Ajax` 浏览器和移动设备，你必须转向正确的页面。这里有一个例子：

```
//error.zul
<zk>
    <zscript>
        if (Executions.getCurrent().isMilDevice())
            Executions.forward("error.mil");
    </zscript>
    <window>
        ....error message in ZUL
    </window>
</zk>
```

更新页面时的错误处理

若当更新 ZUML 页面(亦=一个事件监听器正在执行时)时抛出了一个未被捕获的异常，此异常会由 ZK 更新引擎(ZK Update Engine)处理。默认情况下，它只会要求浏览器端显示一个警告对话框来告诉用户。



你可以在 WEB-INF/zk.xml 文件内指定错误页面以定制错误处理，如下。参考 the Developer's Reference 的附录 B(Appendix B)。

```
<!-- zk.xml -->
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/WEB-INF/sys/error.zul</location>
</error-page>
```

那么，当在事件监听器内发生一个错误时，ZK 更新引擎会使用你指定的错误页面，/error/error.zul，创建一个对话框。

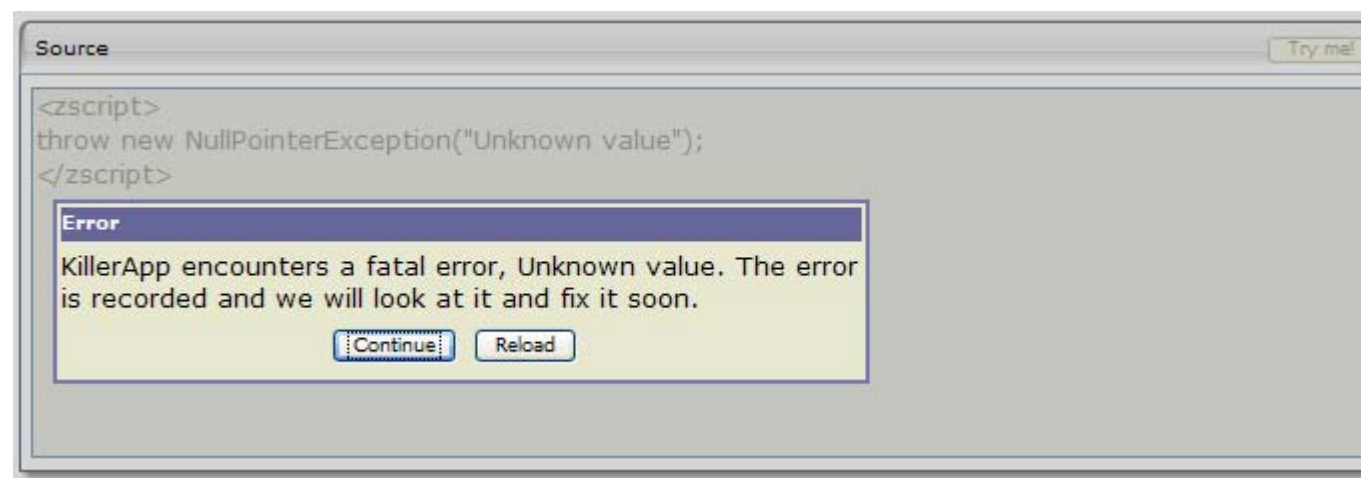
就像加载一个页面时的错误处理，你可以指定多个<error-page>元素。其中的每个元素都与一个不同的异常类型(<exception-type>元素的值)相关联。当发生一个错误时，ZK 将会逐个寻找错误页面，直到异常类型匹配。

另外，ZK 将一套请求属性传递到错误页面来描述发生了什么。这些属性如下。

请求属性	类型
javax.servlet.error.exception_type	java.lang.Class
javax.servlet.error.message	java.lang.String
javax.servlet.error.exception	java.lang.Throwable

例如，你可以指定下列的内容作为错误页面。

```
<window title="Error
${requestScope['javax.servlet.error.status_code']}"
width="400px" border="normal" mode="modal">
    <vbox>
        KillerApp encounters a fatal error,
        ${requestScope['javax.servlet.error.message']}.
        The error is recorded and we will look at it and fix it soon.
        <hbox style="margin-left:auto; margin-right:auto">
            <button label="Continue" onClick="spaceOwner.detach()" />
            <button label="Reload"
onClick="Executions.sendRedirect(null)" />
        </hbox>
    </vbox>
    <zscript>
        org.zkoss.util.logging.Log.lookup("Fatal").log(
            requestScope.get("javax.servlet.error.exception"));
    </zscript>
</window>
```



[提示]： 错误页面是在引起错误的相同桌面内被创建的，所以你可以从其中获取相关的信息。

[提示]： 从 2.3.1 开始，ZK 不会自动将根窗口作为模态(modal)，因为一些应用程序或许并不倾向于使用 modal 窗口。若你倾向于使用模态窗口，可以指定为模态 模式，就像前面所示的例子那样。

更新页面时ZK Mobile错误

每个 device 类型都有其自己的一套错误页面。为了为 ZK mobile 设备指定一个错误页面(支持 MIL 的移动设备)，你必须使用 mil 指定 device-type 元素，如下所示。

```
<!-- zk.xml -->
<error-page>

    <device-type>mil</device-type>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/WEB-INF/sys/error.zul</location>
</error-page>
```

[提示]: 若忽略了 `device-type` 元素,则假定为 `ajax`。换言之,为 `Ajax` 浏览器指定了一个错误页面。

```
<device-type>ajax</device-type> <!-- ajax is the default -->
```

[\[59\]](#) 细节请参考组件活动周期(the Component Lifecycle)一章。

其它

配置ZK加载器不压缩输出

默认情况下,若浏览器支持内容压缩^{[\[60\]](#)}(且输出不被servlets其它所包含),ZK加载器和过滤器(filter)的输出是被压缩的。通过压缩输出,在慢速网络上的传输时间被大幅降低。

但是,若你想使用过滤器来后处理(post-process)输出,压缩未必适用。在这种情况下,你可以将 `compress` 参数(init-param)指定为 `true` 以禁用压缩,WEB-INF/web.xml 文件中的 ZK 加载器配置如下。

```
<servlet>
    <servlet-name>zkLoader</servlet-name>

    <servlet-class>org.zkoss.zk.ui.http.DHtmlLayoutServlet</servlet-class>
    <init-param>
        <param-name>update-uri</param-name>
        <param-value>/zkau</param-value>
    </init-param>
    <init-param>
        <param-name>compress</param-name>
        <param-value>>false</param-value>
    </init-param>
```

```
</servlet>
```

注意：你可以在一个应用程序中配置多个 ZK 加载器。每个拥有不同的选项。

```
<servlet>
  <servlet-name>zkLoader1</servlet-name>

  <servlet-class>org.zkoss.zk.ui.http.DHtmlLayoutServlet</servle
t-class>...
</servlet>
<servlet>
  <servlet-name>zkLoader2</servlet-name>

  <servlet-class>org.zkoss.zk.ui.http.DHtmlLayoutServlet</servle
t-class>...
</servlet>
```

类似的，你也可以配置 ZK 过滤器
(org.zkoss.zk.ui.http.DHtmlLayoutFilter)来禁止压缩。

```
<filter>
  <filter-name>zkFilter</filter-name>

  <filter-class>org.zkoss.zk.ui.http.DHtmlLayoutFilter</filter-c
lass>
  <init-param>
    <param-name>extension</param-name>
    <param-value>html</param-value>  </init-param>
  <init-param>
    <param-name>compress</param-name>
    <param-value>>false</param-value>
  </init-param>
</filter>
```

^[60] 参考<http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>中的 14.3, Accept-Encoding。

第 12 章 性能提示

目录

[使用编译过的Java代码](#)

[使用deferred 属性](#)
[deferred属性和onCreate事件](#)
[使用forward属性](#)
[使用Servlet 线程处理事件](#)
[模态窗口](#)
[消息框](#)
[文件上传](#)
[使用本地命名空间代替XHTML命名空间](#)
[延长时期\(Prolong the Period\)检查文件是否被修改](#)
[延迟子组件的创建](#)
[为大型Listbox使用实况数据和分页](#)
[使用ZK JSP标签或ZK JSF 组件代替ZK Filter](#)

本章将要讲述使你的 ZK 应用程序运行的更快的提示。

使用编译过的Java代码

在 ZUML 中使用 zscript 很方便，但是这需要付出代价：性能较低。降级随应用程序不同而不同(The degradation varies from one application from another)。对于大型的网站，建议尽可能的不使用 zscript。

使用deferred 属性

若你仍然需要编写 zscript 代码，可以指定 deferred 属性来延迟 zscript 代码的赋值(evaluated)，如下。

```
<zscript deferred="true">
...
```

通过指定 deferred 属性，它包含的 zscript 代码在 ZK 提交(render)一个页面时不会被赋值(evaluated)。这意味着当 ZK 提交一个页面时，解释其不会被加载。这节省了内存且加速了页面的提交。

在下面的例子中，仅当按钮被点击时解释器才会被加载：

```
<window id="w">
  <zscript deferred="true">
    void addMore() {
      new Label("More").setParent(w);
    }
  </zscript>
  <button label="Add" onClick="addMore()" />
</window>
```

deferred属性和onCreate事件

这值得注意，若 onCreate 事件监听器由 **zscript** 编写，在前面提到的 **deferred** 选项会变得无效。这是因为 onCreate 事件是在加载页面时被送出的。换言之，若 onCreate 事件监听器由 **zscript** 编写，所有被延迟的 **zscript** 会在加载页面时被赋值，如下所示。

```
<window onCreate="init()">
...
```

最好重写为

```
<window use="my.MyWindow">
...
```

然后准备 `MyWindow.java`，如下。

```
package my;
public class MyWindow extends Window {
    public void onCreate() { //to process the onCreate event
    ...
```

若你喜欢在创建了组件(及该组件所有的子组件)之后马上进行初始化，可以是实现 `org.zkoss.zk.ui.ext.AfterCompose` 接口，如下。注：`AfterCompose` 接口的 `afterCompose` 方法是在组件创建阶段被赋值的，而 `onCreate` 事件是在事件处理阶段被赋值的。

```
package my;
public class MyWindow extends Window implements
org.zkoss.zk.ui.ext.AfterCompose {
    public void afterCompose() { //to initialize the window
    ...
```

使用forward属性

为简化事件流，**ZK** 经常会将事件送至组件本身，而不是父组件或其它目标。例如，当用户点击一个按钮时，`onClick` 事件会被送至 **button**。开发人员通常使用 `onClick` 事件监听器将事件转发至 **window**，如下。

```
<window id="w">
    <button label="OK" onClick="w.onOK"/>
```

正如在前面章节建议的那样，不使用 **zscript** 可以提高性能。因此，你可以使用 `EventListener` 或按如下方式指定 `forward` 属性来重写代码片断。


```
<window>
  <button label="OK" forward="onOK"/>
```

使用Servlet 线程处理事件

默认情况下，ZK 在一个被称为事件处理线程的独立线程中处理事件。因此，开发人员可以在任何时间挂起和恢复执行，而无需阻塞将响应送回服务器的 servlet 线程 (without blocking the servlet thread from sending back the responses to the browser.)。

但是，这消耗了更多的内存，尤其是有许多被挂起的线程的时，且可能会在整合其它系统时引起一些挑战(challenge)，这些系统将信息存储在 Servlet 的本地存储器中 (Servlet thread's local storage)。

ZK 提供了一个选择，用来禁用事件处理线程。换言之，你可以强制 ZK 在 Servlet 线程中处理所有的事件，就像其它的传统框架。当然，若 Servlet 线程已被使用，你可以挂起执行。

为禁用事件处理线程，你必须在 WEB-INF/zk.xml 文件中指定下面的代码。

```
<system-config>
  <disable-event-thread/>
</system-config>
```

这里是使用 Servlet 线程处理事件的优点和局限。在下面的章节中，我们将会更多的讨论关于使用 Servlet 线程时的限制和工作区时(workaround)。

	使用 Servlet 线程	使用事件处理线程
整合(Integration)	很少的整合问题。 很多容器假定 HTTP 请求在 servlet 内被处理。	你或许必须实现 EventThreadInit 和/或 EventThreadCleanup 来解决整合问题。 ZK 和社会(community)会不断提供更多的实现来解决整合问题。
挂起 恢复 (SuspendResume)	没有办法挂起事件监听器的执行。 例如，你可以创建一个模态窗口。	根本没有限制。

模态窗口

你可以不再使用模态窗口。你可以使用 highlighted 模式创建相同的视觉效果。但是，在服务器端，它就像 overlapped 模式一样工作-立即返回而无需等待用户的响应。

```
win.doHighlighted(); //returns once the mode is changed; not
suspended
```

消息框

消息框会立即返回，所以总是返回 `MessageBox.OK`。因此，对于显示 `OK` 按钮以外的按钮，它是没有意义的。例如，下面例子中的 `if` 子句(`clause`)就不会为 `true`。

```
if (MessageBox.show("Delete?", "Prompt",
    MessageBox.YES|MessageBox.NO, MessageBox.QUESTION) ==
    MessageBox.YES) {
    this_never_executes();
}
```

这样，你必须要提供一个事件监听器，如下：

```
MessageBox.show("Delete?", "Prompt",
    MessageBox.YES|MessageBox.NO,
    MessageBox.QUESTION,
    new EventListener() {
        public void onEvent(Event evt) {
            switch (((Integer)evt.getData()).intValue()) {
                case MessageBox.YES: doYes(); break; //the Yes button is
pressed
                case MessageBox.NO: doNo(); break; //the No button is
pressed
            }
        }
    }
);
```

你提供的事件监听器在用户点击其中一个按钮时会被调用。然后，你可以通过测试数据(事件的 `getData` 方法)来确定哪个按钮被按下。数据为一个整数，其值为按钮的标识，例如 `MessageBox.YES`。

```
public void onEvent(Event evt) {
    if ("onYes".equals(evt.getName())) {
        doYes(); //the Yes button is pressed
    } else if ("onNo".equals(evt.getName())) {
        doNo(); //the No button is pressed
    }
}
```

注:OK 按钮的事件名称为 onOK, 而不是 onOk。

文件上传

文件上传对话框不再适用。而你需要使用 fileupload 组件代替。fileupload 组件并不是一个模态对话框。它和其它组件一起被内联置于页面内(it is placed inline with other components)。更多信息请参考 fileupload 组件一节。

```
<fileupload onUpload="handle(event)"/>
```

使用本地命名空间代替XHTML命名空间

就像在 ZUML 及 XUL 组件集(ZUML with the XUL Component Set)一章中与 HTML 标签一起工作(Work with HTML Tags) 一节描述的那样, ZK 使用 XHTML 命名空间为每个指定的 XML 元素创建一个 ZK 组件。换言之, ZK 必须维护它们在服务器端的状态。由于 HTML 标签的数量通常很大, 所以若你使用本地命名空间代替可以显著提高性能。

例如, 下面的代码片断会创建五个组件(一个 table, tr, textbox 和两个 td)。

```
<h:table xmlns:h="http://www.w3.org/1999/xhtml">
  <h:tr>
    <h:td>Name</h:td>
    <h:td>
      <textbox/>
    </h:td>
  </h:tr>
</h:table>
```

另外, 下面的代码片断会创建两个组件(一个特殊的组件, 为客户端产生 table, tr 和 td, 和一个 textbox)。

```
<n:table xmlns:n="http://www.zkoss.org/2005/zk/native">
  <n:tr>
    <n:td>Name</n:td>
    <n:td>
      <textbox/>
    </n:td>
  </n:tr>
</n:table>
```

注意，table，tr 和 td 会直接为客户端产生，所以它们在客户端没有副本(counterpart)。因此，你不能够动态改变它。例如，下面的代码片断是错误的。

```
<n:ul id="x" xmlns:n="http://www.zkoss.org/2005/zk/native"/>
<button label="add" onClick="new Li().setParent(x)"/>
```

而是，你若想动态改变它们，你必须使用 html 组件或 XHTML 命名空间。

延长时期(Prolong the Period)检查文件是否被修改

Zk 缓存(caches)ZUML 页面解析的结果，且仅当页面被修改时才重新编译。在一个生产(production)系统中，ZUML 页面很少被修改，所以通过在 WEB-INF/zk.xml 文件内指定 file-check-period，你可以延长时期以检查页面是否被修改，如下所示。默认为 5 秒。

```
<desktop-config>
  <file-check-period>600</file-check-period><!-- unit: seconds
-->
</desktop-config>
```

延迟子组件的创建

对于更复杂的页面，若我们将子组件延迟到它们变得可见时才创建它们，会显著提高性能。最简单的方式为使用 fulfill 属性。在下面的例子中，第二个 tabpanel 的子组件仅当它变得可见时才会被创建。参考 ZK 用户界面标记语言一章中随机存取一节。

```
<tabbox>
  <tabs>
    <tab label="Preload" selected="true"/>
    <tab id="tab2" label="OnDemand"/>
  </tabs>
  <tabpaneles>
    <tabpanel>
      This panel is pre-loaded since no fulfill specified
    </tabpanel>
    <tabpanel fulfill="tab2.onSelect">
      This panel is loaded only tab2 receives the onSelect event
    </tabpanel>
  </tabpaneles>
</tabbox>
```

为大型Listbox使用实况数据和分页

将一个拥有大量项的 listbox 发送至客户端的代价是很昂贵的。此外，浏览器的 JavaScript 引擎并不擅于使用大量项初始化一个 listbox。更好的解决方法是使用实况数据(live data)，也就是说，为它指派一个 list model。然后，列表项仅在可见时才会被送至客户端。

若使用分页模式，性能会进一步提升。

细节请参考 ZUML 及 XUL 组件集一章中列表框一节。

使用ZK JSP标签或ZK JSF 组件代替ZK Filter

ZK filter 实际上将每个 HTML 标签映射到相应的 XHTML 组件。就像在前一章节描述的那样，由于 ZK 必须维护所有 ZK 组件的状态(包括 XUL 和 XHTML 组件)，这比实际需求消耗了更多的内存。

引入 ZK JSP 标签是为了消除(eliminate)JSP 页面对于 ZK filter 的需要。使用 ZK JSP 标签，会为每个 ZK JSP 标签创建一个 ZUL 组件。所有的其它 HTML 标签都会被作为一个特殊组件被封装。

```
<!-- a JSP page -->
<z:page>
  <table>
    <tr>
      <td>Name</td>
      <td><z:textbox/></td>
    </tr>
  </table>
</z:page>
```

若使用了 ZUL 页面，则等价于下面的代码片断，

```
<!-- a ZUL page -->
<n:table xmlns:n="http://www.zkoss.org/2005/zk/native">
  <n:tr>
    <n:td>Name</n:td>
    <n:td><textbox/></n:td>
  </n:tr>
</n:table>
```

第 13 章 其它设备和输出格式

目录

[ZK Mobile](#)

[Mobile组件集](#), <http://www.zkoss.org/2007/mil>

[XML输出](#)

[使用ZUML页面输出产生XML 输出的三步](#)

[XML组件集](#)

除了 Ajax 浏览器, ZK 也支持移动设备和 XML 输出。本章简要描述它们。对于细节信息, 你可以参考相应文档。

ZK Mobile

ZK Mobile Computing 是 ZK 的扩展, 可以使 ZK 应用程序扩展到移动设备, 而只需少量的编程。

ZK Mobile Computing 由两部分组成。在移动设备方面为 ZK Mobile, 一个 JavaMe Midlet 瘦客户端, 与 ZK 服务器互动, 且作为客户端方面的用户界面工作。在服务器端方面为一套 MIL (移动交互式语言, Mobile Interactive Language)组件, 使用它们你可以控制操作何时结束来自于移动设备的用户触发行为(you can control and manipulate when end users trigger actions from the mobile device)。

开发 ZK Mobile 应用程序是很直接的。只需下载 ZK Mobile 发行版(release), 且跟随 the ZK Mobile Quick Start Guide, 你就可以准备开始了。

Mobile组件集, <http://www.zkoss.org/2007/mil>



你可以像处理 ZUL 和 ZHTML 组件那样编写你的 ZK 应用程序。这次不同的仅是你需要使用 MIL 组件。如下面这个标准的(classic) Hello World 程序(hello.mil)。

```
<frame title="My First Window" visible="true">
    Hello World!
</frame>
```

这几乎和 ZUL 的"Hello World"例子一样。<frame>标签表示 mobile 展示的框架，且"Hello World!"为框架内的文本。

你可以将在 ZUL 组件方面的编程经验应用到 MIL 组件，不会有问题。仅有的不同是前者(ZUL 组件)是为在 web Ajax 浏览器上显示而设计的，而后者(MIL 组件)是为在 ZK Mobile 客户端显示设计的。你仍然可以使用 ZK 的模版(template) 属性，例如 if，

unless，orEach，each，等等。你可以在<zscript>标签内使用多种脚本语言。

你可以使用 EL 表达式和注释的数据绑定。你也可以选择使用 ZUML 页面编写代码或纯的 Java Richlet 方法编写代码。

总之，你可能不仅是编写一个"纯的"ZK Mobile 应用程序。而是，你或许要使用桌面 web 浏览器视图和移动设备视图。为不同的客户端编写不同的视图是很平常的，而所有的视图仍然共享相同的后台(backend)后台业务逻辑和数据库模型。

XML输出

如今 XML 已经成为许多设备和协议的标准格式，如 RSS 和 SVG。使用 ZK 输出 XML 是很直接的。

使用ZUML页面输出产生XML 输出的三步

1. 使用XML组件集(<http://www.zkoss.org/2007/xml>)。
2. 将文件扩展映射到 ZK 加载器。
3. 将文件扩展映射到 XML 组件集。

使用XML组件集， <http://www.zkoss.org/2007/xml>

XML组件集(亦=XML语言，在ZK术语中)被用于产生XML输出。不同于XUL或XHTML组件集，所有在ZUML页面内未知的^[61] 标签都被假定属于本地命名空间(<http://www.zkoss.org/2005/native>)，而不是抛出一个异常。ZK会直接产生它们并输出，而不会为它们创建ZK组件(instantiating a ZK component for each of them)。

下面为一个产生 SVG 输出的例子。这看起来和你想产生的 XML 输出非常相似，除了 (except) 你可以使用 EL 表达式，宏组件和其它 ZK 特性。

```
<?page contentType="image/svg+xml; charset=UTF-8"?>
<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg"
xmlns:z="http://www.zkoss.org/2005/zk">
  <z:zscript><![CDATA[
    String[] bgnds = {"purple", "blue", "yellow"};    int[] rads =
{30, 25, 20};
  ]]></z:zscript>
  <circle style="fill:${each}" z:forEach="${bgnds}"
    cx="${50+rads[forEachStatus.index]}"
    cy="${20+rads[forEachStatus.index]}"
    r="${rads[forEachStatus.index]}" />
</svg>
```



产生的输出将会为

```
<svg xmlns="http://www.w3.org/2000/svg" width="100%"
height="100%" version="1.1">
<circle style="fill:purple" cx="80" cy="50" r="30">
</circle>
<circle style="fill:blue" cx="75" cy="45" r="25">
</circle>
<circle style="fill:yellow" cx="70" cy="40" r="20">
</circle>
</svg>
```

此处

1. 内容类型(content type)由 page 指令指定。对于 SVG，为 image/svg+xml。输出的 xml 处理指令(<?xml?>)和 DOCTYPE 也是由 page 指令指定的。更多关于 page 指令的信息请参考 Developer's Reference。
2. 此例中所有的标签，例如svg和circle，都与命名空间(<http://www.w3.org/2000/svg>)关联，而ZK加载器并不知道此命名空间。因此，它们被假定属于本地命名空间。它们会被直接输出，而不会为它们创建ZK组件。关于更多关于本地命名空间的信息请参考ZUML及XUL组件集 (ZUML with the XUL Component Set)一章中关于本地命名空间(Native Namespace)的部分。
3. 为使用zscript, forEach和其它ZK特性，你必须要指定ZK命名空间(<http://www.zkoss.org/2005/zk>)。

将文件扩展映射到ZK加载器

为让ZK加载器处理文件，你需要在WEB-INF/web.xml内将其与ZK加载器相关联。在此例中，我们将所有的.svg扩展文件映射到ZK加载器^[62]：

```
<servlet-mapping>
  <servlet-name>zkLoader</servlet-name>
  <url-pattern>*.svg</url-pattern>
</servlet-mapping>
```

将文件扩展映射到XML组件集

除非文件扩展名为.xml，否则你必须在WEB-INF/zk.xml内明确的将其与XML组件集(亦=XML语言)相关联。在这个例子中，我们将.svg映射到XML组件集：

```
<language-mapping>
  <language-name>xml</language-name>
  <extension>svg</extension>
</language-mapping>
```

此处，xml为XML组件集(<http://www.zkoss.org/2007/xml>)的名称。因此，当ZK加载器解析一个.svg文件时，它就知道了默认的语言为XML组件集。

XML组件集

除了直接产生XML标签并输出，XML组件集提供了几个组件用以简化复杂的任务，如XSLT。

The XML Transformer

为将一个XML文件翻译为另外一个，你可以使用transformer组件，如下。

```
<?page contentType="text/html; charset=UTF-8"?>
<x:transformer xsl="book.xsl"
xmlns:x="http://www.zkoss.org/2007/xml">
  <book>
    <title>ZK - Ajax without the JavaScript Framework</title>
    <for-who>Web application designers and programmers who wish
to implement rich Ajax web applications in the simplest
way.</for-who>
    <author>Henri Chen and Robbie Cheng</author>
```

```
</book>
</x:transformer>
```

此处， `transformer` 为 XML 组件集的一个组件，所以我们必须要指定命名空间。否则假定为本地命名空间。

那么，让我们假定 `book.xsl` 的内容如下。

```
<xsl:stylesheet
version="1.0"xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head>
        <title>Book Info</title>
      </head>
      <body>
        <h1>Book Info</h1>
        <xsl:apply-templates select="book"/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="book">
    <dl>
      <dt>Title:</dt>
      <dd><xsl:value-of select="title"/></dd>
      <dt>Who is this book for:</dt>
      <dd><xsl:value-of select="for-who"/></dd>
      <dt>Authors</dt>
      <dd><xsl:value-of select="author"/></dd>
    </dl>
  </xsl:template>
</xsl:stylesheet>
```

那么，产生的 XML 输出将会为 XHTML ， 如下。

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>Book Info</title>
</head>
<body>
<h1>Book Info</h1>
<dl>
```

```
<dt>Title:</dt>
<dd>
ZK - Ajax without the JavaScript Framework</dd>
<dt>Who is this book for:</dt>
<dd>
Web application designers and programmers who wish to implement
rich Ajax web applications in the simplest way.</dd>
<dt>Authors</dt>
<dd>
Henri Chen and Robbie Cheng</dd>
</dl>
</body>
</html>
```

[61] 说到未知标签，我们指未与命名空间相关联，或命名空间未知。

[62] 我们假定ZK加载器(zkLoader) 被映射到org.zkoss.zk.ui.http.DHtmlLayoutServlet。

第 14 章 国际化

目录

[地域](#)

[px_preferred_locale](#)会话属性
[请求拦截器](#)

[时区](#)

[px_preferred_time_zone](#) 会话属性
[请求拦截器](#)

[标签](#)

[本地文件](#)

[浏览器和本地URI](#)
[在Java中定位浏览器与本地资源](#)

[消息](#)

[主题](#)

[改变字体大小和/或样式](#)
[使用自制主题](#)
[主题提供者](#)

本章描述了如何使 ZK 应用程序足够简单的运行在任何地域(in any locale)。

首先，ZK 允许开发人员以任何喜欢的方式嵌入 Java 代码或 EL 表达式。你可以使用国际化(Internationalization)方法，例如 java.util.ResourceBundle。

但是，ZK 对于国际化有一些内置(built-in)的支持，你或许会发现它们很有用。

地域

地域(locale)用于处理请求和事件，默认情况下，由浏览器的首选项(preferences)(通过使用 `javax.servlet.ServletRequest` 的 `getLocale` 方法)决定。

但是，这是可以配置的。例如，你或许想为所有的用户使用相同的地域(locale)，而不管浏览器是如何配置的。另一个例子，若在服务器端维护用户的 `profiles`，你或许想使用用户在他或她的 `profile` 内指定的首选(preferred)地域(locale)。

`px_preferred_locale`会话属性

在检查浏览器的首选项之前，ZK 会首先检查是否定义了一个名为 `px_preferred_locale` 的会话属性。若定义了，ZK 会使用它作为会话默认的地域(locale)，同时代替浏览器的首选项。因此，通过为此属性中的首选 `locale` 排序，你可以控制一个会话的地域(locale)。

例如，当用户登录时你可以这样做。

```
void login(String username, String password) {
    //check password
    ...
    Locale preferredLocale = ...; //decide the locale (from, say,
    database)
    session.setAttribute("px_preferred_locale",
    preferredLocale);
    ...
}
```

[提示]: 为避免 `typo`，你或许可以使用定义在 `org.zkoss.web.Attributes` 类内的 `PREFERRED_LOCALE` 常量。

请求拦截器

在用户登录之后才决定地域(locale)，对于一些应用程序已经有些晚了。例如，在用户登录之前，你或许想使用与前一个会话相同的地域(locale)。对于一个 Web 应用程序，这通常由 `cookies` 处理。使用 ZK，你可以注册一个请求拦截器，并且当拦截器被调用时维护 `cookies`。

请求拦截器被用于拦截由 ZK 加载器和 ZK 更新引擎处理的每个请求。它必须实现 `org.zkoss.zk.ui.util.RequestInterceptor` 接口。例如，

```

public class MyLocaleProvider implements
org.zkoss.zk.ui.util.RequestInterceptor {
    public void request(org.zkoss.zk.ui.Session sess,
        Object request, Object response) {
        final Cookie[] cookies =
((HttpServletRequest)request).getCookies();
        if (cookies != null) {
            for (int j = cookies.length; --j >= 0;) {
                if (cookies[j].getName().equals("my.locale")) {
                    //determine the locale
                    String val = cookies[j].getValue();
                    Locale locale =
org.zkoss.util.Locales.getLocale(val);
                    sess.setAttribute(Attributes.PREFERRED_LOCALE,
locale);
                    return;
                }
            }
        }
    }
}

```

为使其生效，你必须按如下方式在 WEB-INF/zk.xml 内进行注册。一旦注册，每次 ZK 加载器或 ZK 更新引擎接收一个请求时都会调用 request 方法。关于配置的详细信息，请参考 the Developer's Reference 的附录 B(Appendix B)。

```

<listener>
    <listener-class>MyLocaleProvider</listener-class>
</listener>

```

[注]：当注册时，一个拦截器的实例会被初始化。然后，相同应用程序内的所有请求会共享此拦截器。因此，你必须确保拦截器可以被同时访问(也就是说，线程安全)。

[注]： request 方法会在很早的阶段被调用，在请求参数被解析之前。因此，推荐在此方法中访问它们，除非你为请求合适的配置了地域(locale)和字符编码。

时区

用于处理请求和事件，默认情况下，由 Java 虚拟机(JVM)的首选项 (java.util.TimeZone 的 getDefault 方法)决定。

[注]： 不同于地域(locale)，没有标准的方法为每种浏览器决定时区。

就像地域(locale)，一个给定会话的时区是可以配置的。例如，若在服务器端维护用户的 profiles，你或许想使用用户在他或她的 profile 内指定的首选(preferred)时区。

px_preferred_time_zone 会话属性

ZK 会检查是否定义了一个名为 px_preferred_time_zone 的会话属性。若定义了，ZK 会使用它作为会话默认的时区，同时代替系统默认选项。因此，通过为此属性中的首选地域(locale)排序，你可以控制一个会话的时区，之后，例如，在前一章节中描述的用户登录。

[提示]: 为避免 typo，你可以使用 org.zkoss.web.Attributes 类内定义的 PREFERRED_TIME_ZONE 常量。

请求拦截器

就像地域(locale)，你可以使用请求拦截器的 px_preferred_time_zone 属性为给定的会话准备时区。

标签

开发人员可以将本地(Locale-dependent)数据从 ZUML 页面(和 JSP 页面)分离出来，只需将它们在 WEB-INF 目录下的 i3-label_lang_CNTY.properties 内排序。此处 lang 为一种语言，如 en 和 fr，CNTY 为国家，如 US 和 FR。

为获得本地(Locale-dependent)属性，你可以在 Java 中使用 org.zkoss.util.resource.Labels，或在 EL 表达式中使用 \${c:l('key')}。为了在 EL 中使用它，你可以按如下方式包含 TLD 文件。

```
<%@ taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c" %>

<window title="${c:l('app.title')}">
...
</window>
```

文件位置: core.dsp.tld 文件位于 dist/WEB-INF 目录。 你不需要将它复制到你的 Web 应用程序中。

当将要获得一个本地标签时，i3-label_lang_CNTY.properties 中的一个文件会被加载。例如，若地域(locale) 为 de_DE，那么 WEB-INF/i3-label_de_DE.properties 将会被加载。若没用这个文件，ZK 将会尝试加载 WEB-INF/i3-label_de.properties 和 WEB-INF/i3-label.properties 并返回(in turn)。

为了在 Java 代码(包括 `zscript`)中访问到标签, 可以使用 `org.zkoss.util.resource.Labels` 类的 `getLabel` 方法。

此外, 你可以继承标签加载器(label loader), 以从别的位置, 例如数据库, 加载标签, 需要注册一个实现了 `org.zkoss.util.resource.LabelLocator` 接口的 `locator`。

本地文件

浏览器和本地URI

许多资源依赖于地域(locale), 有时依赖于用户用于访问 Web 页面的浏览器。例如, 你需要为汉字显示大字体, 以获得更好的可读性。

若你使用 "*" 指定了样式表的 URL, ZK 可以为你自动处理这些。算法(algorithm)如下。

1. 若在一个 URI 内指定了 "*", 例如 `/my*.css`, 那么 "*" 将会被一个合适的地域(Locale)代替, 在浏览器的首选项内指定。

例如, 用户的首选项为 `de_DE`, 那么 ZK 将会从你的 Web 站点依次查找 `/my_de_DE.css`, `/my_de.css`, 和 `/my.css`, 直到任一个被找到。若没有找到, 则仍然使用 `/my.css`。

2. 若在一个 URI 内指定了两个或更多的 "*", 例如 `"/my*/lang*.css"`, 那么首个 "*", 对于 IE 会被 "ie" 代替, Safari 为 "saf", 而其它浏览器^[63]为 "moz"。此外, 最后一个星号(asterisk)会被一个合适的地域(Locale)代替, 就像上一布描述的那样。综上所述, 最后一个星号(the last asterisk)表示地域(Locale), 而第一个星号表示浏览器类型。
3. 所有其它的 "*" 会被忽略。

[注]: 表示地域(Locale)的最后那个星号必须被放置于首个点(".")之前且中间无任何字符, 或若没有点则置于最后。另外, 下列的斜杠是不允许的(no following slash (/) is allowed), 也就是说, 它必须为文件名, 而不是目录。若最后一个星号不满足此约束, 它就会被除去(eliminated)(而不是忽略)。

例如, `"/my/lang.css*" 等价于 "/my/lang.css"`。

换言之, 你可以认为它是中立于地域的(you can consider it as neutral to the Locale)。

[提示]: 我们可以将此规则应用于指定一个依赖于浏览器类型的 URI, 而不是依赖于地域(Locale)。例如, 若用户使用的为 IE, 则 `"/my/langie.css"` 会代替 `"/my/lang*.css"`。

在下面的例子中，我们假定选中的地域(Locale)为 de_DE，且浏览器为 IE。

URI	被搜索的资源
/css/norm*.css	1. /norm_de_DE.css 2. /norm_de.css 3. /norm.css
/css-*/norm*.css	1. /css-ie/norm_de_DE.css 2. /css-ie/norm_de.css 3. /css-ie/norm.css
/img*/pic*/lang*.png	1. /imgie/pic*/lang_de_DE.png 2. /imgie/pic*/lang_de.png 3. /imgie/pic*/lang.png
/img*/lang.gif	1. /img/lang.gif
/img/lang*.gif*	1. /img/langie.gif
/img*/lang*.gif*	1. /imgie/lang*.gif

在Java中定位浏览器与本地资源

除了组件属性和 ZUML 属性，你可以使用 Java 编程来处理浏览器和本地资源。这里为你可以使用的方法列表。

- `org.zkoss.zk.ui.Exection` 中的 `encodeURL` , `forward` 和 `include` 方法，分别用于编码 URL，转向另一个页面，及包含一个页面。在大多数情况下，这些方法可以满足你的需求。
- `org.zkoss.web.servlet.Servlets` 中的 `locate` , `forward` 和 `include` 方法，用于定位 Web 资源。当开发 ZK 应用程序时，你很少需要它们，但是对于编写 `servlet`, `portlet` 或 `filter` 是很有用的。
- `org.zkoss.web.servlet.http.Encodes` 中 `encodeURL` 的方法，用于编码 URL。当开发 ZK 应用程序时，你很少需要它们，但是对于编写 `servlet`, `portlet` 或 `filter` 是很有用的。
- `org.zkoss.util.resource.Locators` 中的 `locate` 方法，用于定位类资源。

[63] 在未来的版本中，除了Internet Explorer， Firefox 和 Safari，我们将为浏览器使用不同的代码。

消息

消息被存储于 `properties` 文件,位于类路径(classpath)的 `/metainfo/mesg` 目录。每一个单元(module)对应一个唯一的名字。地域(Locale)也被添加(appended)到了 `property` 文件。例如, `zk.jar` 中 `Germany` 的消息文件为 `msgzk_de_DN.properties` 或 `msgzk_de.properties`。目前, `zk.jar` 仅能使用(is only shipped with) `English` 和 `Chinese` 版本。你可以为不同的地域(Locales)添加你自己的 `property` 文件,并将它们置于类路径(classpath)的 `/metainfo/mesg` 目录。

主题

XUL 组件集为每种浏览器提供了几套样式表: 较小(smaller), 较大(larger) 和 普通(normal)。默认使用 `normal` 集。你可以配置 `WEB-INF/zk.xml` 来选择一个不同的主题。

改变字体大小和/或样式

有两种方式改变字体的大小和样式(family): 使用 `library` 属性和使用内置的(built-in)CSS 文件。

使用Library 属性

内置的它们使用了下面的变量控制字体大小。

名称	默认	描述
<code>org.zkoss.zul.theme.fontSizeM</code>	12px	默认字体大小。它被用于多大数组件
<code>org.zkoss.zul.theme.fontSizeS</code>	11px	对于要求小字体的组件使用的小一号字体, 如 <code>toolbar</code> 。
<code>org.zkoss.zul.theme.fontSizeXS</code>	10px	非常小的字体, 很少使用。
<code>org.zkoss.zul.theme.fontSizeMS</code>	11px	menu 项目使用的字体。

若想改变默认的字体大小, 你可以在 `WEB-INF/zk.xml` 内指定 `library` 属性, 如下。

```

<library-property>
  <name>org.zkoss.zul.theme.fontSizeM</name>
  <value>12px</value>
</library-property>
<library-property>
  <name>org.zkoss.zul.theme.fontSizeS</name>
  <value>10px</value>
</library-property>
  <library-property>
    <name>org.zkoss.zul.theme.fontSizeXS</name>
    <value>9px</value>
  </library-property>
</library-property>

```

下面的内置变量控制字体样式。

名称	描述
org.zkoss.zul.theme.fontSizeT	默认: Verdana, Tahoma, Arial, Helvetica, sans-serif 此字体样式被用于 titles 和 caption
org.zkoss.zul.theme.fontSizeC	默认: Verdana, Tahoma, Arial, serif 此字体样式被用于 contents。

使用内置的小字体主题

较小字体的主题文件为 normsm*.css.dsp。你可以配置 WEB-INF/zk.xml 来使用它，如下。

```

<desktop-config>

<disable-theme-uri>~/zul/css/norm*.css.dsp*</disable-theme-uri>

  <theme-uri>~/zul/css/normsm*.css.dsp*</theme-uri>
</desktop-config>

```

参考 [Developer's Reference](#) 来获取更多关于如何配置 WEB-INF/zk.xml 的信息。

使用内置的大字体主题

较大字体的主题文件为 `normlg*.css.dsp`。你可以配置 `WEB-INF/zk.xml` 来使用它，如下。

```
<desktop-config>

<disable-theme-uri>~/zul/css/norm*.css.dsp</disable-theme-uri>

  <theme-uri>~/zul/css/normlg*.css.dsp</theme-uri>
</desktop-config>
```

根据Locale使用主题

对于中国地区使用 `larger` 字体时很平常的.你可以按如下方式配置 `WEB-INF/zk.xml` 来启用此功能。

```
<desktop-config>

<disable-theme-uri>~/zul/css/norm*.css.dsp</disable-theme-uri>

  <theme-uri>~/zul/css/norm**.css.dsp</theme-uri>
</desktop-config>
```

使用自制主题

通过提供你自制的样式表(CSS)来定制 ZK XUL 组件的外观是很简单的。你可以参考 ZUL 工程中的 `norm.css.dsp` 和 `normie.css.dsp`。若你只想改变字体，则可以参考 `normsm.css.dsp`。

```
<desktop-config>
  <theme-uri>/css/my-fine-font.css</theme-uri>
</desktop-config>
```

主题提供者

如你想在运行时依赖当前用户，`cookie`，`locale` 或其它决定主题，则可以实现一个主题提供者(theme provider)。主题提供者为实现了 `org.zkoss.zk.ui.util.ThemeProvider` 接口的类。

```
package my;

public class MyThemeProvider implements ThemeProvider {
```

```

    public Collection getThemeURIs(Execution exec, List uris) {
        for (Iterator it = uris.iterator(); it.hasNext();) {
            if ("~/zul/css/norm*.css.dsp*".equals(it.next()))
                it.remove(); //remove the default theme
        }
        HttpServletRequest req =
        (HttpServletRequest)exec.getNativeRequest();
        uris.add(getMyThemeURI(req));
        return uris;
    }
}

```

然后，配置 WEB-INF/zk.xml 添加如下几行。

```

<desktop-config>

<theme-provider-class>my.MyThemeProvider</theme-provider-class>
</desktop-config>

```

又见 zkdemo.war 内的
org.zkoss.zkdemo.userguide.FontSizeThemeProvider 决定了基于
cookie 的主题。

第 15 章 数据库连接

目录

[ZK仅为表现层](#)

[使用JDBC的简单方式 \(但不推荐\)](#)

[使用连接池](#)

[打开及关闭一个连接](#)

[配置连接池](#)

[易于数据库访问的ZK特性](#)

[org.zkoss.zk.ui.event.EventThreadCleanup 接口](#)

[在EL表达式中访问数据库](#)

[事务处理和org.zkoss.zk.util.Initiator](#)

本章将要讲述如何连接数据库。

ZK仅为表现层

Zk的目的是像表现层一样瘦(thin)。此外,使用中心服务器(server-centric)技术,ZK在服务器端运行所有代码,所以,连接数据库和任何桌面应用程序没有区别。换言之,ZK并不改变你访问数据库的方式,不管你使用JDBC或其它持久层框架,例如Hibernate^[64]。

^[64] <http://www.hibernate.org>

使用JDBC的简单方式 (但不推荐)

最简单的方式是使用JDBC,就像任何JDBC教程描述的那样,使用java.sql.DriverManager。这里有一个例子,将姓名和电子邮件存入MySQL^[65]数据库。

```
<window title="JDBC demo" border="normal">
  <zscript><![CDATA[
    import java.sql.*;
    void submit() {
      //load driver and get a database connetion
      Class.forName("com.mysql.jdbc.Driver");
      Connection conn = DriverManager.getConnection(
"jdbc:mysql://localhost/test?user=root&password=my-password");
      PreparedStatement stmt = null;
      try {
        stmt = conn.prepareStatement("INSERT INTO user
values(?, ?)");

        //insert what end user entered into database table
        stmt.set(1, name.value);
        stmt.set(2, email.value);

        //execute the statement
        stmt.executeUpdate();
      } finally { //cleanup
        if (stmt != null) {
          try {
            stmt.close();
          } catch (SQLException ex) {
```

```

        log.error(ex); //log and ignore
    }
}
if (conn != null) {
    try {
        conn.close();
    } catch (SQLException ex) {
        log.error(ex); //log and ignore
    }
}
}
}
</zscript>
<vbox>
    <hbox>Name : <textbox id="name"/></hbox>
    <hbox>Email: <textbox id="email"/></hbox>
    <button label="submit" onClick="submit()"/>
</vbox>
</window>

```

尽管简单，但并不推荐这样做。毕竟 ZK 应用程序是基于 Web 的，加载是难以预测且宝贵的资源，例如数据库连接，必须要有效的管理。

很幸运，所有的 J2EE 框架和 Web 服务器都支持一种称为连接池(connection pooling)的功能。很容易使用连接池，且用于管理数据库连接更好。在下一章节我们会讨论更多。

[提示]: 不同与其它 Web 应用程序，可以和 ZK 使用 DriverManager，尽管不推荐这样做。

首先，你需要在桌面缓存连接，为每一个事件重复使用此连接，当桌面失效时关闭连接。就像传统的客户端/服务器应用程序一样工作。 就像客户端/服务器应用程序，这样会有有效的运作，仅当最多有数十个并发用户时。

为了得知桌面何时失效，你需要通过
org.zkoss.zk.ui.util.DesktopCleanup 实现一个监听器。

[65] <http://www.mysql.com>

使用连接池

连接池是一种创建和管理多个(a pool of)连接的技术，这些连接是为任何需要它们的线程准备的。连接池不会马上关闭连接，而是将连接保存在一个 pool 中，这样可以很高效地服务下一个连接。除此之外，连接池还有很多好处，例如，控制资源的使用。

当开发基于 Web 的应用程序时，没有理由不使用连接池，包括 ZK 应用程序。

连接池的概念很简单：配置，连接及关闭。打开和关闭一个连接与点对点(ad-hoc)方法很相似，而配置依赖于使用的 Web 服务器和数据库。

打开及关闭一个连接

配置好连接池(将会在下方的章节讨论)之后，你可以使用 JNDI 获取一个连接，如下。

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

import javax.naming.InitialContext;
import javax.sql.DataSource;

import org.zkoss.zul.Window;

public class MyWindows extends Window {
    private Textbox name, email;
    public void onCreate() {
        //initial name and email
        name = getFellow("name");
        email = getFellow("email");
    }
    public void onOK() throws Exception {
        DataSource ds = (DataSource)new InitialContext()
            .lookup("java:comp/env/jdbc/MyDB");
        //Assumes your database is configured and
        //named as "java:comp/env/jdbc/MyDB"

        Connection conn = null;
        Statement stmt = null;
        try {
            conn = ds.getConnection();
            stmt = conn.prepareStatement("INSERT INTO user
values(?, ?)");
```

```

        //insert what end user entered into database table
        stmt.set(1, name.value);
        stmt.set(2, email.value);

        //execute the statement
        stmt.executeUpdate();
        stmt.close(); stmt = null;
        //optional because the finally clause will close it
        //However, it is a good habit to close it as soon as done,
        especially
        //you might have to create a lot of statement to complete
        a job
    } finally { //cleanup
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException ex) {
                //(optional log and) ignore
            }
        }
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException ex) {
                //(optional log and) ignore
            }
        }
    }
}
}
}

```

[注]:

1. 使用之后关闭语句和连接是很重要的。
2. 使用多个连接, 你可以同时访问多个数据库。依赖于配置和 J2EE/Web 服务器, 这些连接甚至可以形成一个分布式事务处理(distributed transaction)。

配置连接池

连接池的配置随 J2EE/Web/数据库服务器的不同而不同。这里我们说明其中一些的配置。你需要参考(consult)你使用的服务器的文档。

Tomcat 5.5 + MySQL

为配置Tomcat 5.5 配置连接池，你必须编辑\$TOMCAT_DIR/conf/context.xml^[66]，在<Context>元素的下添加下面的内容。标记为蓝色的的信息依赖于你的安装，且通常需要修改。

```
<!-- The name you used above, must match _exactly_ here!
    The connection pool will be bound into JNDI with the name
    "java:/comp/env/jdbc/MyDB"
-->
<Resource name="jdbc/MyDB" username="someuser"
password="somepass"
    url="jdbc:mysql://localhost:3306/test"
    auth="Container" defaultAutoCommit="false"
    driverClassName="com.mysql.jdbc.Driver" maxActive="20"
    timeBetweenEvictionRunsMillis="60000"
    type="javax.sql.DataSource" />
</ResourceParams>
```

然后在 web.xml 内，你需要在<web-app>元素下添加下面的内容。

```
<resource-ref>
    <res-ref-name>jdbc/MyDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

JBoss + MySQL

下列的指令基于 MySQL 5.0 参考手册的 23.3.4.3 章节(section 23.3.4.3 of the reference manual of MySQL 5.0)。

为 JBoss 配置连接池，你必须在 deploy (\$JBoss_DIR/server/default/deploy)目录下添加一个新文件。文件名要以 "-ds.xml"结尾，即告诉 JBoss 将此文件部署为 JDBC 数据源。文件必须包含下列内容。标记为蓝色的的信息依赖于你的安装，且通常需要修改。

```
<datasources>
    <local-tx-datasource>
        <!-- This connection pool will be bound into JNDI with the
name
        "java:/MyDB" -->
        <jndi-name>MyDB</jndi-name>
```

```

<connection-url>jdbc:mysql://localhost:3306/test</connection-url>

    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>someser</user-name>
    <password>somepass</password>

    <min-pool-size>5</min-pool-size>

    <!-- Don't set this any higher than max_connections on your
    MySQL server, usually this should be a 10 or a few 10's
    of connections, not hundreds or thousands -->

    <max-pool-size>20</max-pool-size>

    <!-- Don't allow connections to hang out idle too long,
    never longer than what wait_timeout is set to on the
    server...A few minutes is usually okay here,
    it depends on your application
    and how much spikey load it will see -->

    <idle-timeout-minutes>5</idle-timeout-minutes>

    <!-- If you're using Connector/J 3.1.8 or newer, you can use
    our implementation of these to increase the robustness
    of the connection pool. -->

<exception-sorter-class-name>com.mysql.jdbc.integration.jboss.
ExtendedSQLExceptionSorter</exception-sorter-class-name>

<valid-connection-checker-class-name>com.mysql.jdbc.integratio
n.jboss.MySqlValidConnectionChecker</valid-connection-checker-
class-name>

    </local-tx-datasource>
</datasources>

```

JBoss + PostgreSQL

```

<datasources>
    <local-tx-datasource>
        <!-- This connection pool will be bound into JNDI with the name
        "java:/MyDB" -->

```

```
<jndi-name>MyDB</jndi-name>

<!-- jdbc:postgresql://[servername]:[port]/[database name] -->

<connection-url>jdbc:postgresql://localhost/test</connection-url>

<driver-class>org.postgresql.Driver</driver-class>
<user-name>someuser</user-name>
<password>somepass</password>
<min-pool-size>5</min-pool-size>
<max-pool-size>20</max-pool-size>
<track-statements>false</track-statements>
</local-tx-datasource>
</datasources>
```

[66] 多谢 Thomas Muller (<http://asconet.org:8000/antville/oberinspector>) 更正。

又见 <http://tomcat.apache.org/tomcat-5.5-doc/jndi-resources-howto.html>
和 [http://en.wikibooks.org/wiki/ZK/How-Tos/HowToHandleHibernateSessions#Working with the Hibernate session](http://en.wikibooks.org/wiki/ZK/How-Tos/HowToHandleHibernateSessions#Working_with_the_Hibernate_session) 获取细节。

易于数据库访问的ZK特性

`org.zkoss.zk.ui.event.EventThreadCleanup` 接口

就像以前强调过的，在 `finally` 子句中关闭连接是很重要的，这样每个连接都会被正确的返回连接池。

为使你的应用程序更加健壮(**robust**)，你可以实现 `org.zkoss.zk.ui.event.EventThreadCleanup` 接口来关闭任何处在等待之际的连接和语句(**any pending connections and statements**)，以防你的一些应用程序代码忘记了在 `finally` 子句中关闭它们。

但是，关闭处在等待之际的连接和语句实际上依赖于你使用的服务器。你需要参考服务器的文档来得知如何编写关闭代码。

[提示]：在许多情况下，并不需要(且并不容易)提供这样的方法，因为多数连接池的实现可以循环一个连接若调用了连接池的 `finalized` 方法(**because most implementation of connection pooling be recycled a connection if its finalized method is called**)。

在EL表达式中访问数据库

除了在事件监听器中访问数据库，使用 EL 表达式访问数据库填充一个属性也是很常见的。在下面的例子中，我们从数据库取出数据，并使用 EL 表达式将它们用 listbox 展示出来。

```
<zscript>
    import my.CustomerManager;
    customers = new CustomerManager().findAll(); //load from
database
</zscript>
<listbox id="personList" width="800px" rows="5">
    <listhead>
        <listheader label="Name"/>
        <listheader label="Surname"/>
        <listheader label="Due Amount"/>
    </listhead>
    <listitem value="${each.id}" forEach="${customers}">
        <listcell label="${each.name}"/>
        <listcell label="${each.surname}"/>
        <listcell label="${each.due}"/>
    </listitem>
</listbox>
```

有几种方式来实现 findAll 方法。

读取所有数据并将其拷贝到链表

最简单的方式是在 findAll 方法内获取所有的数据，将它们拷贝到一个列表，然后关闭连接。

```
public class CustomerManager {
    public List findAll() throws Exception {
        DataSource ds = (DataSource)new InitialContext()
            .lookup("java:comp/env/jdbc/MyDB");

        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        List results = new LinkedList();
        try {
            conn = ds.getConnection();
            stmt = conn.createStatement();
```

```

        rs = stmt.executeQuery("SELECT id, name, surname FROM
customers");
        while (rs.next()) {
            long id = rs.getInt("id");
            String name = rs.getString("name");
            String surname = rs.getString("surname");
            results.add(new Customer(id, name, surname));
        }
        return results;
    } finally {
        if (rs != null) try { rs.close(); } catch (SQLException ex)
[]
        if (stmt != null) try { stmt.close(); } catch (SQLException
ex) []
        if (conn != null) try { conn.close(); } catch (SQLException
ex) []
    }
}
}

```

实现org.zkoss.zk.ui.util.Initiator接口

你可以使用 `init` 指令来加载数据，以代替在试图中混合使用 `Java` 代码。

```

<?init class="my.AllCustomerFinder" arg0="customers"?>

<listbox id="personList" width="800px" rows="5">
    <listhead>
        <listheader label="Name"/>
        <listheader label="Surname"/>
        <listheader label="Due Amount"/>
    </listhead>
    <listitem value="${each.id}" forEach="${customers}">
        <listcell label="${each.name}"/>
        <listcell label="${each.surname}"/>
        <listcell label="${each.due}"/>
    </listitem>
</listbox>

```

然后，使用 `org.zkoss.zk.ui.util.Initiator` 接口实现 `my.CustomerFindAll` 类。

```

import org.zkoss.zk.ui.Page;
import org.zkoss.zk.ui.util.Initiator;

```

```

public class AllCustomerFinder implements Initiator {
    public void doInit(Page page, Object[] args) {
        try {
            page.setVariable((String)args[0], new
CustomerManager().findAll());
            //Use setVariable to pass the result back to the page
        } catch (Exception ex) {
            throw UiException.Aide.wrap(ex);
        }
    }
    public void doCatch(Throwable ex) { //ignore
    }
    public void doFinally() { //ignore
    }
}

```

事务处理和org.zkoss.zk.util.Initiator

度与复杂的应用程序(例如分布式事务处理),你或许需要明确的控制一个事务处理的活动周期。若所有的数据库访问均是在事件监听器中被处理的,那么在 ZK 中为使其工作并不需要改变什么。你开始,提交或回滚(start, commit or rollback)一个事务处理,与你的 J2EE/Web 服务器文档建议的一样。

但是,若你想将整个 ZUML 页面(组件创建阶段)的赋值(evaluation)在相同的事务处理里被执行,如上一章节描述的那样,你可以实现
org.zkoss.zk.util.Initiator 接口控制给定页面的事务处理活动周期。

实现的框架如下所示。

```

import org.zkoss.zk.ui.Page;
import org.zkoss.zk.ui.util.Initiator;

public class TransInitiator implements Initiator {
    private boolean _err;    public void doInit(Page page, Object[]
args) {
        startTrans(); //depending the container, see below
    }
    public void doCatch(Throwable ex) {
        _err = true;
        rollbackTrans(); //depending the container, see below
    }    public void doFinally() {
        if (!_err)
            commitTrans(); //depending the container, see below
    }
}

```

```
}  
}
```

如上所示，事务处理在 `doInit` 方法内开始，且在 `org.zkoss.zk.util.Initiator` 接口的 `doFinally` 方法内结束。

如何开始，提交和回滚(`start`, `commit` and `rollback`)一个事务处理取决于你使用的容器。

J2EE事务处理及Initiator

若你使用的为一个 J2EE 容器，你可以找到事务管理器(transaction manager)(`javax.transaction.TransactionManager`)，然后调用它的 `begin` 方法开始一个事务处理。调用 `rollback` 方法回滚。调用 `commit` 方法提交。

Web 容器和Initiator

若你使用的为一个没有事务管理器的 Web 容器，可以通过构造(`constructing`)一个数据库连接来开始事务处理。然后，据此调用 `commit` 和 `rollback` 方法。

```
import java.sql.*;  
import javax.sql.DataSource;  
import javax.naming.InitialContext;  
import org.zkoss.util.logging.Log;  
import org.zkoss.zk.ui.Page;  
import org.zkoss.zk.ui.util.Initiator;  
  
public class TransInitiator implements Initiator {  
    private static final Log log =  
Log.lookup(TransInitiator.class);  
    private Connection _conn;  
    private boolean _err;  
  
    public void doInit(Page page, Object[] args) {  
        try {  
            DataSource ds = (DataSource)new InitialContext()  
                .lookup("java:comp/env/jdbc/MyDB");  
            _conn = ds.getConnection();  
        } catch (Throwable ex) {  
            throw UiException.Aide.wrap(ex);  
        }  
    }  
  
    public void doCatch(Throwable t) {
```

```

        if (_conn != null) {
            try {
                _err = true;
                _conn.rollback();
            } catch (SQLException ex) {
                log.warning("Unable to roll back", ex);
            }
        }
    }

    public void doFinally() {
        if (_conn != null) {
            try {
                if (!_err)
                    _conn.commit();
            } catch (SQLException ex) {
                log.warning("Failed to commit", ex);
            } finally {
                try {
                    _conn.close();
                } catch (SQLException ex) {
                    log.warning("Unable to close transaction", ex);
                }
            }
        }
    }
}

```

第 16 章 整合Hibernate

目录

[什么是Hibernate](#)

[安装Hibernate>](#)

[配置ZK的配置文件](#)

[创建Java对象](#)

[映射Java对象](#)

[使用映射文件](#)

[使用Java注释](#)

[创建Hibernate 配置文件](#)

[创建DAO 对象](#)

[在ZUML页面访问持久对象](#)

什么是Hibernate

Hibernate 是一个对象关系映射(ORM)解决方案,专门针对 Java 语言。Hibernate 的主要特点是简化了对于数据库的访问。

安装Hibernate>

在使用 Hibernate 前,你首先要将它安装到你的应用程序。

1. 下载Hibernate (<http://www.hibernate.org>)
2. 将所有的 jar 文件放到\$myApp/WEB-INF/lib/下

\$myApp 表示你的应用程序的名称。例如, Event

配置ZK的配置文件

为使 ZK 和 Hibernate 顺利的工作,你需要完成下面的工作。

1. 在\$myApp/WEB-INF/下创建 zk.xml (如果没有的话)
2. 将下面的内容复制到你的 zk.xml

```
<!-- Hibernate SessionFactory lifecycle -->
<listener>
  <description>Hibernate SessionFactory
lifecycle</description>

<listener-class>org.zkoss.zkplus.hibernate.HibernateSessionFac
toryListener</listener-class>
</listener>

<!-- Hibernate OpenSessionInView Pattern -->
<listener>
  <description>Hibernate Open Session In View
life-cycle</description>

<listener-class>org.zkoss.zkplus.hibernate.OpenSessionInViewLi
stener</listener-class>
</listener>

<!-- Hibernate thread session context handler -->
<listener>
```

```
<description>Hibernate thread session context
handler</description>

<listener-class>org.zkoss.zkplus.hibernate.HibernateSessionCon
textListener</listener-class>
</listener>
```

创建Java对象

你需要创建简单的 **JavaBean** 类，并添加一些属性。

```
package events;

import java.util.Date;

public class Event {
    private Long id;

    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    public Date getDate() {
        return date;
    }
    public void setDate(Date date) {
        this.date = date;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
}
```

1. 创建你的首个 Java 类(Event.java)

2. 你需要编译 Java 源文件，然后将类文件放到 Web 部署文件夹的 classes 目录下，要保证包名正确。(例如，\$myApp/WEB-INF/classes/event/Event.class)

下一步是告诉 Hibernate 如何将这个持久类和数据库映射。

映射Java对象

有两种方式告诉 Hibernate 如何加载和存储持久类对象，第一种方式是使用 Hibernate 映射文件，另一种方式是使用 Java 注释。

使用映射文件

1. 简单的为持久类 Event.java 创建一个 Event.hbm.xml 文件。
2.

```
<?xml version="1.0"?>
```
3.

```
<!DOCTYPE hibernate-mapping PUBLIC
```
4.

```
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
```
5.

```
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```
- 6.
7.

```
<hibernate-mapping>
```
- 8.
- 9.
- 10.
11.

```
<class name="events.Event" table="EVENTS">
```
- 12.
13.

```
<id name="id" column="EVENT_ID">
```
- 14.
15.

```
<generator class="native"/>
```
- 16.
17.

```
</id>
```
- 18.
19.

```
<property name="date" type="timestamp"
```
20.

```
column="EVENT_DATE"/>
```
21.

```
<property name="title"/>
```
- 22.
23.

```
</class>
```
- 24.
25.

```
</hibernate-mapping>
```
26. 将这个 Event.hbm.xml 文件放到部署文件夹的目录 src 下，并且保证正确的包名。(例如,\$myApp/WEB-INF/src/event/Event.hbm.xml)

使用Java注释

使用 Java 注释的好处是不用创建额外的 Hibernate 配置文件。为你的 Java 类简单的添加 Java 注释来告诉如何 Hibernate 如何关联。

```
package events;

import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="EVENTS")
public class Event {

    private Long id;

    private String title;

    private Date date;

    @Id
```

```
@GeneratedValue(strategy=GenerationType.SEQUENCE)

@Column(name = "EVENT_ID")

public Long getId() {

    return id;

}


private void setId(Long id) {

    this.id = id;

}


@Column(name = "EVENT_DATE")

public Date getDate() {

    return date;

}


public void setDate(Date date) {

    this.date = date;

}


public String getTitle() {

    return title;

}
```

```

public void setTitle(String title) {

    this.title = title;

}

}

```

1. `@Entity` 声明这个类为一个持久对象
2. `@Table(name = "EVENTS")` 注释表示这个实体是和数据库的 `EVENTS` 表映射的。
3. `@Column` 元素备用与映射实体属性和数据库表的字段。
4. `@Id` 元素定义了映射主键字段的属性。

创建Hibernate 配置文件

下一步是安装Hibernate 来使用一个数据库。HSQL DB ,一个基于Java的SQL 数据库管理系统(DBMS), 可以在HSQL DB 网站(<http://hsqldb.org/>)下载到。

1. 解压其到一个目录, 例如 `c:/hsqldb`。
2. 打开一个命令框(command box), 切换到 `c:/hsqldb` 目录。
3. 在命令提示(command prompt)下, 执行 `java -cp lib/hsqldb.jar org.hsqldb.Server`。

安装完数据库之后, 我们需要安装 `Hibernate` 配置文件。在部署文件夹的目录 `src` 下创建

`hibernate.cfg.xml`

文件(例如, `$myApp/WEB-INF/src/hibernate.cfg.xml`)。将下面的内容复制到你的 `hibernate.cfg.xml`。这依赖于你如何映射 `Java` 对象。

使用映射文件

```

<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.
dtd">

```

```
<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property
name="connection.driver_class">org.hsqldb.jdbcDriver</property
>
        <property
name="connection.url">jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property
name="dialect">org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable Hibernate's automatic session context
management -->
        <property
name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache -->
        <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvide
r</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

        <mapping resource="events/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

使用Java注释

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.
dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property
name="connection.driver_class">org.hsqldb.jdbcDriver</property
>
        <property
name="connection.url">jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property
name="dialect">org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable Hibernate's automatic session context
management -->
        <property
name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache -->
        <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvide
r</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
```



```
        <property name="hbm2ddl.auto">create</property>

        <mapping class="events.Event"/>
    </session-factory>

</hibernate-configuration>
```

我们继续，下面是如何创建一个类来处理数据库访问工作。

创建DAO 对象

为了简化维护，我们常常创建另一个类来处理数据库访问工作。

```
1. Create EventDAO.java
2.  package events;
3.
4.  import java.util.Date;
5.  import java.util.List;
6.
7.  import org.hibernate.Session;
8.  import org.zkoss.zkplus.hibernate.HibernateUtil;
9.
10. public class EventDAO {
11.
12.     Session currentSession() {
13.         return HibernateUtil.currentSession();
14.     }
15.
16.     public void saveOrUpdate(Event anEvent, String title,
17.         Date date) {
18.         Session sess = currentSession();
19.         anEvent.setTitle(title);
20.         anEvent.setDate(date);
21.
22.         sess.saveOrUpdate(anEvent);
23.     }
24.
25.     public void delete(Event anEvent) {
26.         Session sess = currentSession();
27.
28.         sess.delete(anEvent);
29.     }
30. }
```

```

31.     public Event findById(Long id) {
32.         Session sess = currentSession();
33.
34.         return (Event) sess.load(Event.class, id);
35.     }
36.
37.     public List findAll() {
38.         Session sess = currentSession();
39.
40.
41.         return sess.createQuery("from Event").list();
42.     }
43. }

```

44. 你需要编译 Java 源文件，然后将类文件放到 Web 部署文件夹的 classes 目录下，要保证包名正确。(例如，
\$myApp/WEB-INF/classes/event/EventDAO.class)

在ZUML页面访问持久对象

为了在 ZUML 页面访问持久对象，需要简单的声明一个持久对象，并且使用该对象从数据库获取数据。

1. 在 web 部署的根目录创建一个 event.zul 文件。(例如，
\$myApp/event.zul)

```

2.     <zk>
3.     <zscript><![CDATA[
4.     import java.util.Date;
5.     import java.text.SimpleDateFormat;
6.     import events.Event;
7.     import events.EventDAO;
8.
9.     //fetch all allEvents from database
10.    List allEvents = new EventDAO().findAll();
11.
12.    ]]></zscript>
13.    <listbox id="lbxEvents">
14.
15.        <listhead>
16.            <listheader label="Title" width="200px"/>
17.            <listheader label="Date" width="100px"/>
18.        </listhead>
19.        <listitem forEach="${allEvents}" value="${each}">
20.            <listcell label="${each.title}"/>

```

```
21.      <zscript>String datestr = new
        SimpleDateFormat("yyyy/MM/dd").format(each.date);</zscrip
        t>
22.      <listcell label="${datestr}"/>
23.    </listitem>
24.  </listbox>
25. </zk>
```

26. 打开浏览器访问ZUML页面(例如, <http://localhost:8080/event/event.zul>)。

第 17 章 整合Spring

目录

[什么是Spring](#)

[使用Spring的准备](#)

[将spring.jar复制到你的Web library](#)

[配置web.xml](#)

[创建Spring配置文件](#)

[创建Spring Bean类](#)

[在ZUML 页面内访问 Spring Bean](#)

[使用 variable-Resolver](#)

[使用 SpringUtil](#)

[Spring Security](#)

[运行一个简单的应用程序](#)

[使用Spring Security的准备](#)

[配置/WEB-INF/web.xml 文件](#)

[创建 /WEB-INF/applicationContext-security.xml](#)

[定义哪些服务被保护](#)

[定义那些ZK事件被保护](#)

[ZUML页面](#)

什么是Spring

Spring 是构建(building)Java 应用程序的一个平台,它包含许多易于使用的解决方案,用来构建基于 web 的应用程序。

使用Spring的准备

首先你需要完成下列工作:

将spring.jar复制到你的Web library

在使用 Spring 之前，需要下载它，并将 jar 文件放到应用程序的目录下。

1. 下载 Spring library (<http://www.springframework.org/download>)
2. 将 spring.jar 放到 \$myApp/WEB-INF/lib/下

\$myApp 代表你的应用程序的名字。

配置web.xml

在 web.xml 内，你需要定义

org.springframework.web.context.ContextLoaderListener，并指定配置文件的位置来加载 bean 定义。

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
<listener-class>org.springframework.web.context.ContextLoaderL
istener</listener-class>
</listener>
```

创建Spring配置文件

在 applicationContext.xml 文件内定义 bean，并将此文件放到你的 WEB-INF 目录。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="DataSource" class="test.DataSourceImpl"/>
</beans>
```

创建Spring Bean类

在面向对象编程(OOP)中，你需要定义一个 DataSource 接口：

DataSource.java

```
package test;

public interface DataSource
{
    java.util.List getElementsList();
}
```

它的实现类为:

DataSourceImpl.java

```
package test;

import java.util.*;

public class DataSourceImpl implements DataSource
{
    public List getElementsList()
    {
        List list = new ArrayList();
        list.add("Tom");
        list.add("Henri");
        list.add("Jim");

        return list;
    }
}
```

在ZUML 页面内访问 Spring Bean

在 ZUML 页面内有两种方式来访问基于 Spring 的 bean。第一种方式是 variable-resolver，另一种方式是 SpringUtil。使用哪个要看你的习惯，在 ZUML 页面内，我们建议你使用 variable-resolver。

使用 variable-Resolver

在 ZUML 页面的顶部简单的为

org.zkoss.zkplus.spring.DelegatingVariableResolver 声明

variable-resolver，然后，在下面的页面里，你可以使用 bean 的 id 来访问任何 Spring-managed bean。

```

<?variable-resolver
class="org.zkoss.zkplus.spring.DelegatingVariableResolver"?>
  <window>
    <grid>
      <rows>
        <row forEach="${DataSource.elementsList}">
          <label value="${each}"/>
        </row>
      </rows>
    </grid>
  </window>

```

variable-resolver 将会为你自动查找名字为 DataSource 的 bean，然后返回一个 list 到 forEach 循环

。

使用 SpringUtil

org.zkoss.zkplus.spring.SpringUtil 是一个实用类，它允许你使用简单的方式来获取 Spring-managed bean。

```

<window>
  <zscript>
    import org.zkoss.zkplus.spring.SpringUtil;
    import test.*;

    DataSource dataSource = SpringUtil.getBean("DataSource");
    List list = dataSource.getElementsList();
  </zscript>

  <grid>
    <rows>
      <row forEach="${list}">
        <label value="${each}"/>
      </row>
    </rows>
  </grid>
</window>

```

在这里 forEach 循环集合来打印集合内每个对象的 \${each} 属性。

Spring Security

Spring Security 2.0 是 Spring 框架的下一代安全系统。它在上一代 Acegi 安全系统上又添加了许多新特性。

运行一个简单的应用程序

1. 下载 war 文件

<http://downloads.sourceforge.net/zkforge/zkspringsec2.war>

该示例程序来自于 Spring Security 2.0, 已经做了修改以保证和 ZK 一起顺利工作。

2. 部署 war 文件

Tomcat 5.5 或更高版本。

将 zkspringsec2.war 复制到 Tomcat 服务器的
\$Tomcat_Home/webapps/文件夹, 然后重启 Tomcat。

3. 使用你的浏览器访问:

<http://localhost:8080/zkspringsec2/>

username/password 为 rod/koala

使用Spring Security的准备

为使 Spring Security 和 ZK 一起工作, 你需要复制下面的 jar 到 /WEB-INF/lib。

1. ZK Spring Library jar 文件
2. zkspring.jar
3. Spring Security library jar 文件
4. aopalliance-1.0.jar
5. aspectjrt-1.5.4.jar
6. commons-codec-1.3.jar
7. commons-collections-3.2.jar
8. commons-lang.jar
9. commons-logging-1.1.1.jar
10. jstl-1.1.2.jar
11. log4j-1.2.14.jar
12. spring-security-acl-2.0.3.jar

```
13. spring-security-core-2.0.3.jar
14. spring-security-core-tiger-2.0.3.jar
15. spring-security-taglibs-2.0.3.jar
16. spring.jar
17. standard-1.1.2.jar
```

配置/WEB-INF/web.xml 文件

为使 Tomcat 和 Spring Security 一起工作，你需要指明 spring-security 配置文件的位置，然后定义 spring 的 listener 和 spring-security 的 filter，如下：

```
<!--
    - Location of the XML file that defines the root application
context
    - Applied by ContextLoaderListener.
-->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        ...
        /WEB-INF/applicationContext-security.xml
    </param-value>
</context-param>

<!--
    - Loads the root application context of this web app at startup.
    - The application context is then available via
    -
WebApplicationContextUtils.getWebApplicationContext(servletCon
text).
-->
<listener>

<listener-class>org.springframework.web.context.ContextLoaderL
istener</listener-class>
</listener>

<!--
    - Spring Security Filter Chains
-->
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterP
roxy</filter-class>
```



```
</filter>
```

创建 /WEB-INF/applicationContext-security.xml

在\$myApp/WEB-INF/目录下创建 application-security.xml。此文件包含了 spring-security 所需要的配置(dentitions)。

```
<!--
- Spring namespace-based configuration
-->
<beans:beans
xmlns="http://www.springframework.org/schema/security"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-2.0.1.xsd">

    <!--
    - Enable the @Secured annotation to secure service layer
    methods
    -->
    <global-method-security secured-annotations="enabled">
    </global-method-security>

    <!--
    - Secure the page per the URL pattern
    -->
    <http auto-config="true">
        <intercept-url pattern="/secure/extreme/**"
        access="ROLE_SUPERVISOR"/>
        <intercept-url pattern="/secure/**"
        access="IS_AUTHENTICATED_REMEMBERED" />
        <intercept-url pattern="/**"
        access="IS_AUTHENTICATED_ANONYMOUSLY" />
```

```

        <!-- use own login page rather than the default one -->
        <form-login login-page="/login.zul"/>
    </http>

    <!--
    Usernames/Passwords are
        rod/koala
        dianne/emu
        scott/wombat
        peter/opal
    -->
    <authentication-provider>
        <password-encoder hash="md5"/>
        <user-service>
            <user name="rod"
password="a564de63c2d0da68cf47586ee05984d7"
authorities="ROLE_SUPERVISOR, ROLE_USER, ROLE_TELLER" />
            <user name="dianne"
password="65d15fe9156f9c4bbffd98085992a44e"
authorities="ROLE_USER,ROLE_TELLER" />
            <user name="scott"
password="2b58af6dddbd072ed27ffc86725d7d3a"
authorities="ROLE_USER" />
            <user name="peter"
password="22b5c9accc6e1ba628cedc63a72d57f8"
authorities="ROLE_USER" />
        </user-service>
    </authentication-provider>
</beans:beans>

```

定义哪些服务被保护

使用 Spring Security 注释@Secured 来保护商业服务的调用，如下：

```

public interface BankService {

    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    public Account readAccount(Long id);

    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    public Account[] findAccounts();
}

```

```
@Secured("ROLE_TELLER")
public Account post(Account account, double amount);
}
```

定义那些ZK事件被保护

在/WEB-INF/applicationContext-security.xml 内定义 zk 命名空间需要的配置，并指定哪个 ZK 事件需要被保护，如下，

```
<!--
- Spring namespace-based configuration
-->
<beans:beans
xmlns="http://www.springframework.org/schema/security"
xmlns:zksp="http://www.zkoss.org/2008/zkspring"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-2.0.1.xsd
http://www.zkoss.org/2008/zkspring
http://www.zkoss.org/2008/zkspring/zkspring.xsd">

    <http ...>
        ...
    </http>
    ...

<!--
- Secure the ZK event processing per the event name and ZK component
path pattern
-->
<zksp:zk-event login-template-close-delay="5">
<zksp:intercept-event event="onClick" path="//**/btn_*"
access="ROLE_TELLER"/>
<zksp:intercept-event path="//**"
access="IS_AUTHENTICATED_ANONYMOUSLY"/>
```

```
</zksp:zk-event>
```

```
</beans:beans>
```

1. `xmlns:zksp="http://www.zkoss.org/2008/zkspring"` 告诉 Spring Security 引擎我们将要使用 ZK Spring 命名空间配置，并定义命名空间为 zksp。
2. `http://www.zkoss.org/2008/zkspring`

`http://www.zkoss.org/2008/zkspring/zkspring.xsd` 告诉 Spring Security 引擎哪里可以找到这个 ZK Spring 命名空间的配置 schema。
3. `<zksp:zk-event>` 告诉 Spring Security 引擎我们要保护 ZK 事件处理。这将会自动配置需要的 listeners, filters 和 Spring bean。在这个例子中，`login-template-close-delay="5"` 告诉 ZK 登录成功 5 秒后自动关闭这个登录窗口；`zero(0)` 意味着登录成功后立即关闭登录窗口；而一个负数值则意味着要等待用户的操作。
4. `<zksp:intercept-event event="onClick" path="//**/btn_*" access="ROLE_TELLER"/>` 告诉 Spring Security 引擎我们想要保护哪些 ZK 事件和组件。在这个例子中，例如任何一个 id 以 btn_ 开头的组件的 onClick 事件被触发后，它都会被 ROLE_TELLER authority 检查。
5. `<zksp:intercept-event path="/" access="IS_AUTHENTICATED_ANONYMOUSLY"/>` 表示所有的匿名用户可以访问所有的事件和组件。

ZUML页面

在下面的例子中，当他/她企图该改变帐户余额(the balance of account)时，ZK 会提示终止用户登录，因为 Button 组件的 onClick 事件已经被 Spring-Security 保护了。

```
<?variable-resolver
class="org.zkoss.spring.DelegatingVariableResolver"?>
<zk>
<window title="Accouts" border="normal" width="500px">
  <zscript><![CDATA[
    void adjBalance(Button btn) {
      double bal = new
Double((String)btn.getAttribute("bal")).doubleValue();
      //get the account object
      bigbank.Account a =
bankService.readAccount(btn.getAttribute("aid"));
      //change the account balance
      bankService.post(a, bal);
```

```

        //update the account balance on the browser

btn.getFellow("bal_"+a.getId()).setValue(""+a.getBalance());
    }
]]>
</zscript>
<grid>
    <rows>
        <row forEach="${accounts}">
            <label value="${each.id}"/>
            <label value="${each.holder}"/>
            <label id="bal_${each.id}"
value="${each.balance}"/>
            <button id="btn_m20_${each.id}" label="- $20"
onClick="adjBalance(self)">
                <custom-attributes aid="${each.id}" bal="-20"/>
            </button>
            <button id="btn_m5_${each.id}" label="- $5"
onClick="adjBalance(self)">
                <custom-attributes aid="${each.id}" bal="-5"/>
            </button>
            .....
        </row>
    </rows>
</grid>
</window>
<button label="Home" href="/index.zul"/>
<button label="Logout" href="/j_spring_security_logout"/>
</zk>

```

第 18 章 Portal 整合

目录

[配置](#)

[WEB-INF/portlet.xml](#)

[WEB-INF/web.xml](#)

[使用方法](#)

[zk_page 及 zk_richlet 参数和属性](#)

[事例](#)

ZK 提供了一个 portlet 来为 JSR 168 compliant portal 加载 ZUML 页面。这个 portlet 被称为 ZK portlet 加载器(ZK portlet loader)，它就像 `org.zkoss.zk.ui.http.DHtmlLayoutPortlet` 一样被实现。

配置

WEB-INF/portlet.xml

为使用 ZK portlet 加载器，首先你需要将下列定义添加至 `WEB-INF/portlet.xml`。注意，`expiration-cache` 必须被设置为 0，以阻止 portal 缓存结果。

```
<portlet>
  <description>ZK loader for ZUML pages</description>
  <portlet-name>zkPortletLoader</portlet-name>
  <display-name>ZK Portlet Loader</display-name>

  <portlet-class>org.zkoss.zk.ui.http.DHtmlLayoutPortlet</portlet-class>

  <expiration-cache>0</expiration-cache>
  <supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>VIEW</portlet-mode>
  </supports>
  <supported-locale>en</supported-locale>
  <portlet-info>
    <title>ZK</title>
    <short-title>ZK</short-title>
    <keywords>ZK, ZUML</keywords>
  </portlet-info>
</portlet>
```

WEB-INF/web.xml

ZK portlet 加载器实际上将 ZUML 页面的加载委派(delegate)给 ZK 加载器 (`org.zkoss.zk.ui.http.DHtmlLayoutServlet`)。因此，你必须配置 `WEB-INF/web.xml`，将像在 [the Developer's Reference](#) 的附录 A(Appendix A) 中指定的那样，即使你仅想使用 portlet。

使用方法

zk_page 及 zk_richlet 参数和属性

ZK portlet 加载器是一个通用的加载器。为加载一个特定的 ZUML 页面，你需要指定一个请求参数，一个 portlet 属性或一个被称为 zk_page 的首选项，若你想加载一个 ZUML 页面，或 zk_richlet，若 你想加载一个 richlet。

更确切地说，ZK portlet 加载器为 ZUML 页面或 richlet 的路径检查下列的位置。号码越小，优先级越高。

1. 被称为 zk_page 的请求参数(RenderRequest's getParameter)。若找到了，则它为 ZUML 页面的路径。
2. 被称为 zk_page 的请求属性(RenderRequest's getAttribute)。
3. 被称为 zk_page 的请求首选项(RenderRequest's getPortletPreferences's getValue)。若找到了，则它为 ZUML 页面的路径。
4. 被称为 zk_richlet 的请求参数(RenderRequest's getParameter)。若找到了，则它为 richlet 的路径。
5. 被称为 zk_richlet 的请求属性(RenderRequest's getAttribute)。若找到了，则它为 richlet 的路径。
6. 被称为 zk_richlet 的请求首选项(RenderRequest's getPortletPreferences's getValue)。若找到了，则它为 richlet 的路径。
7. 被称为 zk_page 的初始参数(PortletConfig's getInitParameter)。若找到了，则它为 ZUML 页面的路径。

事例

如何将一个请求参数或属性传递到一个 portlet 取决于 portal。你需要参考你最喜欢 portal 的用户指南获取细节。下面为使用 Potix Portal 的一个例子。

```
<layout contentType="text/html">
  <title>ZK Porlet Demo</title>
  <header name="Cache-Control" value="no-cache"/>
  <header name="Pragma" value="no-cache"/>
  <vbox>
    <hbox>
      <servlet page="sample1.zul"/>
      <portlet name="zkdemo.zkLoader">
        <attribute name="zk_page" value="/test/sample2.zul"/>
      </portlet>
    </hbox>
  </vbox>
</layout>
```

```

    </hbox>
  </vbox>
  <molds uri="~/pxp/html/molds.xml"/>
</layout>

```

Population	Percentage
Graduate	20%
College	23%
High School	40%
Others	17%

Subject	From	Received
<input type="checkbox"/> Intel Snares XML	David Needle	7-12-2005
<input type="checkbox"/> Intel Snares XML	Ria Coen	7-12-2005
Unknown chaos		
<input type="checkbox"/> C# versus Java	David Longman	7-10-2005

第 19 章 ZK 之外

目录

[Logger](#)

[如何使用 ZK 配置日志等级](#)

[i3-log.conf 的内容](#)

[i3-log.conf 的位置](#)

[禁用所有日志](#)

[DSP](#)

[iDOM](#)

除了处理 ZUML 页面，ZK 发行版包括了许多技术和工具。本章提供了一些关于它们的基本信息。感兴趣的读者可以参考 Javadoc 获取详细的 API。

Logger

Package: `org.zkoss.util.logging.Log`

ZK 使用的 `logger` 基于标准的 `logger`, `java.util.Logger`。但是，我们将其包装成 `org.zkoss.util.logging.Log`，以使其更高效。典型的使用方法如下。

```

import org.zkoss.util.logging.Log;
class MyClass {
    private static final Log log = Log.lookup(MyClass.class);
    public void f(Object v) {
        if (log.isDebugEnabled()) log.debug("Value is "+v);
    }
}

```


由于 ZK 使用了标准的 logger 记录消息，通过配置你使用 Web 服务器的 logging，可以控制记录什么。如何配置 Web 服务器的 logging 依服务器的不同而不同。请参考手册。或者，你可以使用 ZK 提供的 logging 配置机制，如下所述。

[注]: 默认情况下，所有的 ZK log 实例都会被映射到同一个名为 `org.zkoss` 的 Java logger，主要为了获得更好的性能。若你想控制日志级别以适于个别类(*If you want to control the log level up to individual class*)，则必须要调用下面的语句来打开对于等级(hierarchy)的支持。

```
Log.setHierarchy(true);
```

[注]: 若你使用 `WEB-INF/zk.xml` 配置了日志级别，就像在下面章节描述的那样，等级支持会被自动禁用。

如何使用ZK配置日志等级

除了配置 Web 服务器的 logging，你可以使用 ZK 提供的 logging 配置机制。默认是禁用的。为启用它，你必须在 `WEB-INF/zk.xml` 内指定下列内容。更多细节请参考 the Developer's Reference 的附录 B(Appendix B)。

```
<zk>
  <log>
    <log-base>org.zkoss</log-base>
  </log>
</zk>
```

另外，通过调用 `LogService` 的 `init` 方法，你可以手动启用 logging 配置机制，如下。

```
org.zkoss.util.logging.LogService.init("org.zkoss", null);
```

若你不仅想记录 `org.zkoss` 而是所有，可以为 `log-base` 指定空。

一旦启用了此机制，通过在启动时搜索类路径(classpath)和一些特定的位置(见下面)，ZK 会搜寻 `i3-log.conf`。若找到了，ZK 会加载它的内容并初始化日志级别。然后，ZK 会一直监视此文件，若文件被修改则会重新加载此文件内容。

i3-log.conf的内容

`i3-log.conf` 的一个事例如下。

```
org.zkoss.zk.ui.impl.UiEngineImpl=FINER
```

```

#Make the log level of the specified class to
FINERorg.zkoss.zk.ui.http=DEBUG
#Make the log level of the specified package to DEBUG
org.zkoss.zk.au.http.DHtmlUpdateServlet=INHERIT
#Clear the log level of a specified class such that it inherits
what

#has been defined above (Default: INFO)
org.zkoss.zk.ui=OFF

#Turn off the log for the specified package
org.zkoss=WARNING

#Make all log levels of ZK classes to WARNING except those
specified here

```

被允许的级别

级别	描述
OFF	表示没有消息。
ERROR	表示提供错误信息。
WARNING	表示提供警告信息。也隐含着 ERROR。
INFO	表示提供信息性的消息。也隐含着 ERROR 和 WARNING。
DEBUG	表示提供用于调试目的的跟踪信息。也隐含着 ERROR, WARNING 和 INFO。
FINER	表示提供用于调试目的的相当详细的跟踪信息。也隐含着 ERROR, WARNING, INFO 和 DEBUG。
INHERIT	表示明确任何为指定包或类设置的等级。换言之，日志等级与其父结点相同。

i3-log.conf的位置

首先，ZK 会在类路径(classpath)查找此文件。若未找到，则会在 conf 目录查找。

应用程序 服务器	位置
Tomcat	将 i3-log.conf 置于\$TOMCAT_HOME/conf 目录。
其它	首先试试 conf 目录。若不工作，你可以设置被称为

应用程序 服务器	位置
	org.zkoss.io.conf.dir 的系统属性目录作为放置 i3-log.conf 的目录(you could set the system property called the org.zkoss.io.conf.dir directory to be the directory where i3-log.conf resides)。

禁用所有日志

一些日志会在加载i3-log.conf之前产生。若你想完全禁用所有日志，则必须配置Web服务器^[67]的 logging，或当配置WEB-INF/web.xml内的 DHtmlLayoutServlet时指定log-level。细节请参考the Developer's Reference。

```
<servlet>
  <servlet-name>zkLoader</servlet-name>

  <servlet-class>org.zkoss.zk.ui.http.DHtmlLayoutServlet</servle
t-class>
  <init-param>
    <param-name>log-level</param-name>
    <param-value>OFF</param-value>
  </init-param>
  ...

```

^[67] 记住ZK使用标准的logging功能。除非你在 i3-log.conf内指定了一些内容，默认的 logging级别取决于Web服务器(通常为INFO)。

DSP

Package: org.zkoss.web.servlet.dsp

一种类似于 JSP 模板的技术。它的语法和 JSP 相同。不同于 JSP， DSP 在运行时被解释，所以很容易部署 DSP 页面。在运行环境中并不需要 Java 编译器。此外，你可以将 DSP 页面发布为 jar 文件。这就是 ZK 发布的方式。

但是，你不能在 DSP 页面内嵌入 Java 代码。DSP 的行为，尽管可以通过 TLD 文件扩展，与 JSP 标签还是不同的。

若你想在你的 Web 应用程序中使用 DSP，需要在 WEB-INF/web.xml 内添加下列几行。

```
<!-- ////////////////// -->
<!-- DSP (optional) -->
<servlet>
    <description><![CDATA[
The servlet loads the DSP pages.
    ]]></description>
    <servlet-name>dspLoader</servlet-name>

<servlet-class>org.zkoss.web.servlet.dsp.InterpreterServlet</servlet-class>

    <!-- Specify class-resource, if you want to access TLD defined
in jar files -->
    <init-param>
        <param-name>class-resource</param-name>
        <param-value>true</param-value>    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>dspLoader</servlet-name>
    <url-pattern>*.dsp</url-pattern>
</servlet-mapping>
```

[注]：DSP 加载器的映射是可选的。仅当你想使用 DSP 语法编写 Web 页面时再指定它。

尽管标准的 ZK 组件使用 DSP 做为模板技术，但是它们直接由 ZK 加载器处理。

详细信息请参考 Developer's Reference 。

iDOM

Package: org.zkoss.idom

一个W3C DOM 的实现。它由JDOM^[68]驱使(inspired)，对于所有的XML对象都有具体的类，例如元素和属性。但是，iDOM 实现了W3C API，例如 org.w3c.dom.Element。因此，你可以无缝使用iDOM与仅接受W3C DOM的XML功能。

典型的例子为 XSLT 和 XPath。你可以与 iDOM 使用任何喜欢的 XSL 处理器和 XPath 功能。

[68] <http://www.jdom.org>