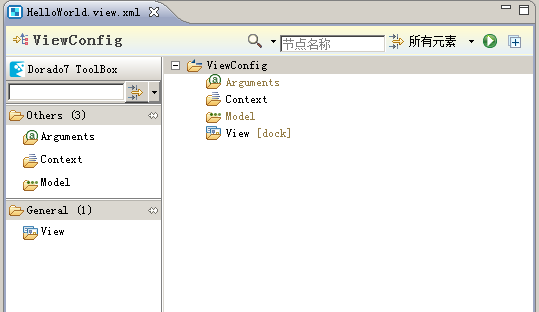
## 1、View配置文件基本结构



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

<ViewConfig>

<Arguments/>

<Context/>

<Model/>

<View/>

</ViewConfig>

一个空白的视图配置文件共有四个节点。

### 1）Arguments

用来定义参数，可在View的其他地方来引用这个参数，在特定情况下可以简化配置，如提取公用信息配置在arguments中。另外arguments的值允许从外部传入。这种参数可在当前的View配置文件中通过EL表达式引用。引用方法：通过${arguments.foo}的形式引用。

### 2）Context

对应为DoradoContext的View范围的上下文对象，它在View的整个生命周期中无论在Browser还是在Server端，都可以很方便的访问和存储其中的数据。

代码编写方式与map类似，使用时我们可以在网页中通过js将一个字符串存入Context，之后我们再发送一个ajax请求，服务器端就可以根据其key值取到刚才存入的字符串。同时我们也可以在这次ajax请求的服务器端java代码中将一个字符串存入Context，然后在这次ajax请求结束后，我们可以在Browser端对应这次ajax请求的回调函数或对应执行成功的事件中通过java端设定的key获取到返回的值。

使用方法：

* Server端通过context.getAttribute(DoradoContext.View,”foo”);读取。
* Client端通过view.get(“context”).get(“foo”)或者view.get(“context.foo”);读取

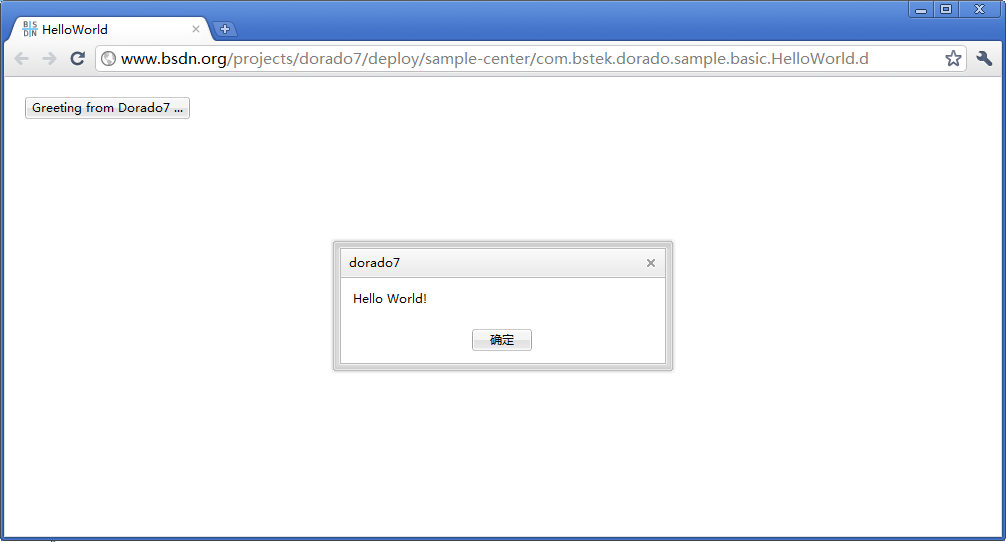
### 3）Model

Model是View的私有模型对象，View可以在此处定义自己私有的模型对象供当前页面使用，或通过这个私有的模型对象覆盖models中的某个全局的模型对象，并进行个性化设定。

### 4）View

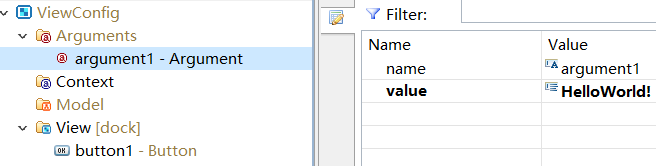
View节点是最常使用的一个节点，dorado7中的空间都是添加到这个节点下的。

## 2、实做HelloWorld



### 1）Arguments用法

在此基础上，使用一下View中的argument。首先在Arguments节点下添加一个新的argument节点，并设置value为“HelloWorld!”，同时设置按钮的caption为${argument.argument1}，这是EL表达式的用法。



### 2）URL与视图配置文件之间的映射关系

URL路径与View文件的资源文件的对应关系。

<http://localhost:8080/sample-center/com.bstek.dorado.sample.training.HelloWorld.d>

以上可以通过如下URL访问：

<http://localhost:8080/sample-center/com/bstek/dorado/sample/training/HelloWorld.d>

很多情况下，如果所有View都处于相对集中的位置，例如所有的View都在com.bstek.dorado.sample目录下，则可以通过调整dorado-home下context.xml中的dorado.viewConfigFactoryRegister的配置简化上面的URL的访问：

<bean parent="dorado.viewConfigFactoryRegister">

<property name="viewNamePattern" value="\*\*" />

<property name="viewConfigFactory">

<bean parent="dorado.xmlViewConfigDefinitionFactory">

<property name="pathPrefix"

value="${view.root}/com/bstek/dorado/sample" />

<property name="pathSubfix" value=".view.xml" />

</bean>

</property>

</bean>

按照上面的规则调整后就可以通过如下的方式访问：

<http://localhost:8080/sample-center/training.HelloWorld.d>

## 3、EL表达式

Dorado7提供了2种EL表达式：静态表达式、动态表达式。

|  |  |
| --- | --- |
| 表达式类型 | 表达式格式 |
| 静态表达式 | ${expression} |
| 动态表达式 | $${expression} |

差别：

静态EL：在创建目标对象的过程中一次性的求值。

动态EL：以动态代理的方式创建目标对象，并在外界每一次读取相应属性时都进行求值。

（注：动态EL表达式通过动态代理实现，使用的时候需要注意隐式对象的选择）

### 1）隐式对象

Dorado提供14个隐式对象，不需要预先定义就可以直接使用这些隐式对象。

与线程无关：

|  |  |
| --- | --- |
| 标识符 | 描述 |
| null | 表示null |
| env | 表示获取系统环境变量。${env.HOME}或${env[“HOME”]} |
| system | Java的系统属性，即System.getProperties()对象。${system.property1}或${system[“property1”]} |
| configure | 用于获取Dorado7的配置信息。${configure[“core.runMode”]} |

与线程有关：

|  |  |
| --- | --- |
| argument | 获取当前配置文件中的参数值，仅可用于View.xml |
| context | 当前线程中的DoradoContext |
| ctx | 简化对DoradoContext的getAttribute方法的访问。${ctx[“foo”]}相当于${context.getAttribute(“foo”)} |
| util | 基本工具类，${util.getToday()}获取当前日期，${util.getDate(‘yyyy年MM月dd日 HH时mm分ss秒’)}指定日期格式 |
| request | 当前的HttpServletRequest对象，${request.getAttribute(“foo”)} |
| req | 简化对request的getAttribute方法的访问。${req["foo"]}相当于${request.getAttribute("foo")} |
| param | 简化对request的getParameter方法的访问。${param["foo"]}相当于${request.getParameter("foo")} |
| session | 当前的HttpSession对象，与会话作用域属性的名称和值相关联的Map类。${session.getAttribute("foo")} |
| servletContext | 当前的SerlvetContext对象。  ${servletContext.getAttribute("foo")} |
| web | 基本工具类，见  com.bstek.dorado.web.WebExpressionUtilsObject的javadoc |

（注：可扩展，开发人员可自行扩展出更多的隐式变量。）

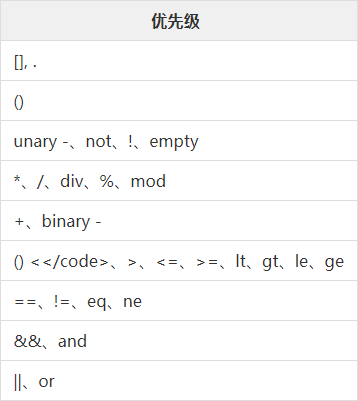
### 2）基本语法

EL可以通过使用标识符和存取器（.和[]），遍历包含应用程序数据（通过限制了作用域的变量公开）或关于环境的信息（通过EL隐式对象）的对象层次结构。

EL还包括了几个用来操作和比较EL表达式所访问数据的运算符。

|  |  |
| --- | --- |
| 类别 | 运算符 |
| 算数运算符 | +、-、\*、/（或div）、%（或mod） |
| 关系运算符 | ==（或eq）、!=（或ne）、<（或lt）、>（或gt）、<=（或le）、>=（或ge） |
| 逻辑运算符 | &&（或and）、||（或or）、!（或not） |
| 验证运算符 | empty，${empty input}返回一个布尔值，单个表达式作为其变量，求值结果为null被认为是空（即无元素的集合或数组） |

运算符优先级：

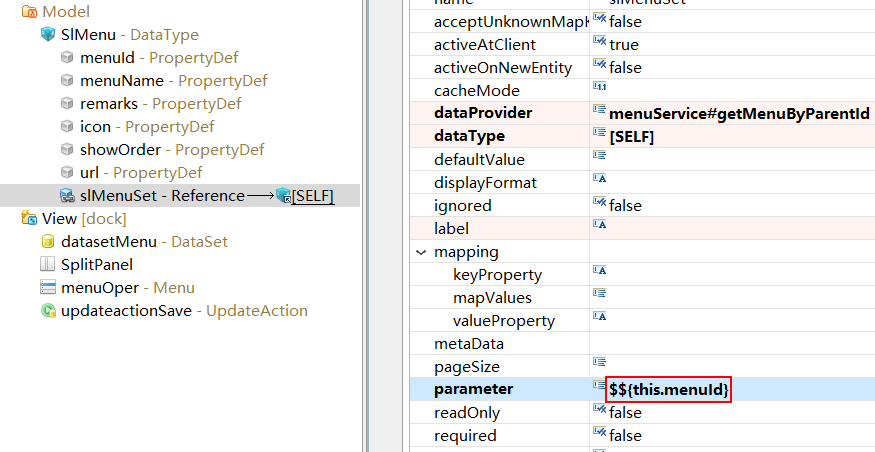


文字：在EL表达式中，数字、字符串、布尔值和null都可以被指定为文字值。字符串可以用单引号或双引号定界。布尔值被指定为true或false。

### 2）特殊EL隐式变量

$${this}或$${this.foo}。虽然写法上像动态EL，但实际运行的机制与一般EL并不一样，要点：

* 此种隐式变量通常只用在DataType的Reference属性中。this表示当前正在处理的数据实体对象，即系统正在尝试获取该数据实体对象的Reference属性的值。
* 此种EL表达式的实际求值范围是在Client端的JavaScript环境中。



## 4、控件的基础知识

### 1）控件的基本类别（不可见控件、可见控件）

不可见控件：

如DataSet（数据的容器）、Action（代表了一个动作）、FormProfile（用于设定表单的格式和风格）。这种控件不参与页面布局，不会在页面显示出来，但一般来说可以在页面上使用它，例如通过js代码存取它的属性，或调用它的方法。其中最重要的是DataSet，Dorado将数据模型和展现形式分开后，DataSet就是用来描述其中的数据模型的。

可见控件：

如Button，它会直接显示在浏览器中，参与页面的布局，课件控件可以将它分为：

* 独立控件：Button、TextEditor（表单中常用），Grid
* 容器控件：内部可以包含其他各种类型控件（包括容器控件），Container、Panel、GroupBox
* 复合控件：TabControl、SplitPanel、ToolBar

容器控件与复合控件的区别：

* 最大的外在区别在于是否支持布局管理器。
* 容器控件中总是有一块空白的矩形区域用于显示隶属于容器的子控件，且子控件的显示方式受容器控件当前选用的布局管理器控制。
* 复合控件虽然也可以包含一到多个子控件，但这些子控件如何被显示是由复合控件自身的逻辑决定的。如TabController控件，它也可以包含多个控件，但是一次只能显示一个控件，它无法同时展现多个控件，无法随意更改。

### 2）控件的其他分类方式（浮动控件、数据感知控件）

浮动控件：

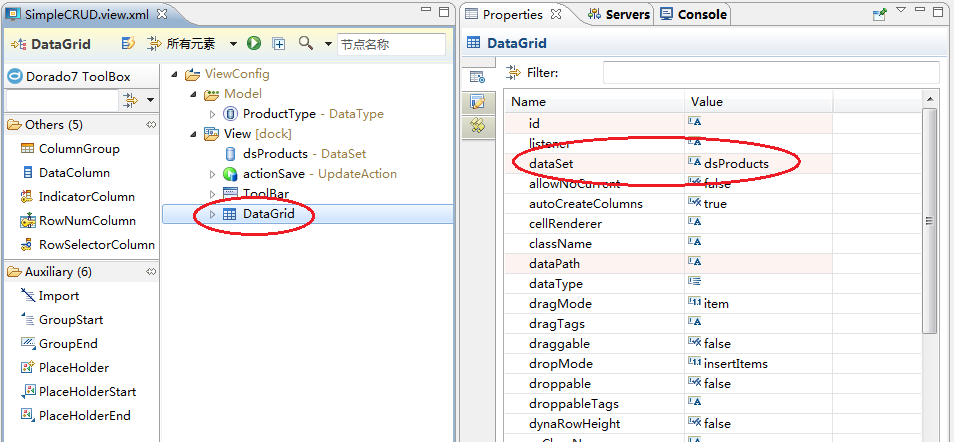
默认不接受布局管理器的管理，且显示时浮动于所有非浮动控件之上。例如：Dialog、Menu、FloatContainer。



如果将浮动控件Menu的floating属性设置为false，则浮动控件将失去浮动特性且将接受布局管理器的控制。

数据感知控件：

可以与DataSet进行绑定用于显示、修改或控制DataSet中的数据的空间。例如：DataGrid、TextEditor、AutoForm。



## 5、布局管理器

布局就是指容器组件中子元素分布、排列组合方式。Dorado的所有容器组件都支持布局操作，每一个容器都会有一个对应的布局，布局负责管理容器组件中子元素的排列、组合以及渲染方式等。

Dorado7目前提供的布局管理器：

|  |  |
| --- | --- |
| 布局名称 | 说明 |
| Dock Layout | 停靠式布局 |
| Anchor Layout | 锚定式布局 |
| Form Layout | 表单式布局 |
| Hbox Layout | 水平箱式布局 |
| Vbox Layout | 垂直箱式布局 |
| Native Layout | （其实什么都不做的布局） |

### 1）Dock Layout停靠式布局

Dorado中最常用的布局方式（作为页面的总体布局）。

* 停靠式布局将空间分为5种区域：left、top、right、bottom、center
* 每个控件选择一种区域由布局管理器按照顺序进行排列。
* 选择center区域的控件总是到最后才被放置，它总是占尽所有剩余的空间。
* 除center区域之外，其他4种区域都可以出现0到n个，center区域最多只能出现一次。
* 如果某控件没有显式的声明选择哪种区域，那么布局管理器会尝试按照center区域对其进行布局。

定义View的布局如下：

<View title="DockLayout" layout="Dock">

<HtmlContainer content="1) left" layoutConstraint="left" />

<HtmlContainer content="2) top" layoutConstraint="top" />

<HtmlContainer content="3) right" layoutConstraint="right" />

<HtmlContainer content="4) bottom" layoutConstraint="bottom" />

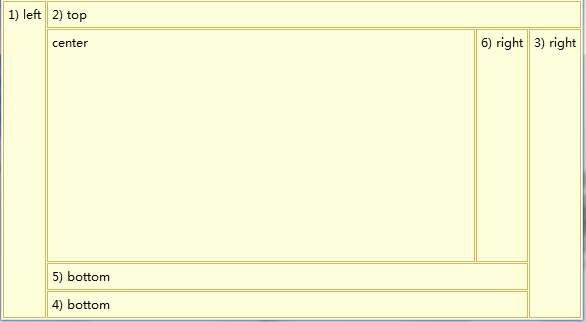
<HtmlContainer content="5) bottom" layoutConstraint="bottom" />

<HtmlContainer content="6) right" layoutConstraint="right" />

<HtmlContainer content="center" />

</View>

设置布局的layout为Dock，不同的HtmlContainer的layoutConstraint属性分别设置为：left、top、right、bottom和center中的一种，则得到的布局效果如图：



实际使用时并不一定要包含所有的区域。

<View layout="Dock" title="DockLayout">

<ToolBar layoutConstraint="top">

<ToolBarLabel>

<Property name="text">菜单栏</Property>

</ToolBarLabel>

</ToolBar>

<ToolBar layoutConstraint="bottom">

<ToolBarLabel>

<Property name="text">状态栏</Property>

</ToolBarLabel>

</ToolBar>

<Panel layoutConstraint="left">

<Property name="caption">导航树</Property>

<Buttons/>

<Children/>

<Tools/>

</Panel>

<Panel layoutConstraint="center">

<Property name="caption">工作空间</Property>

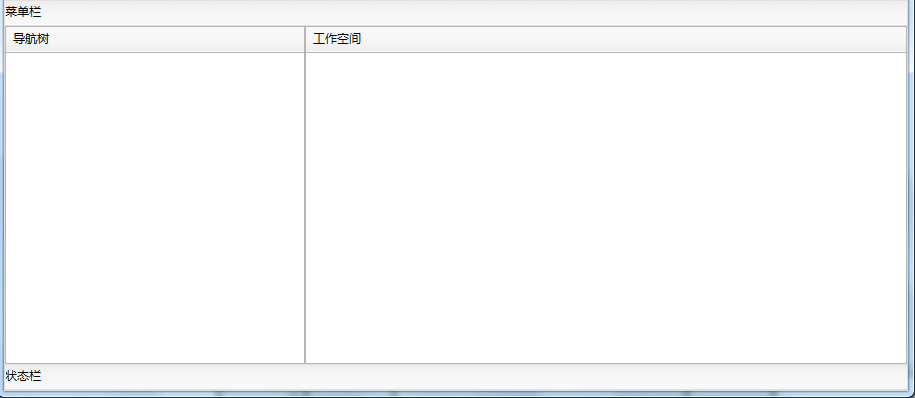
<Buttons/>

<Children/>

<Tools/>

</Panel>

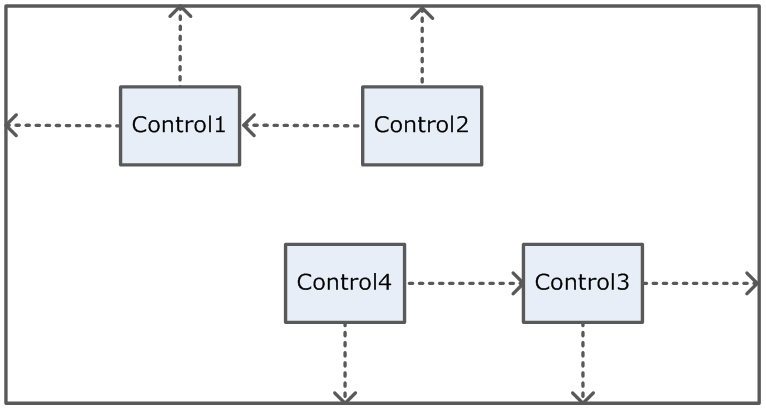
</View>

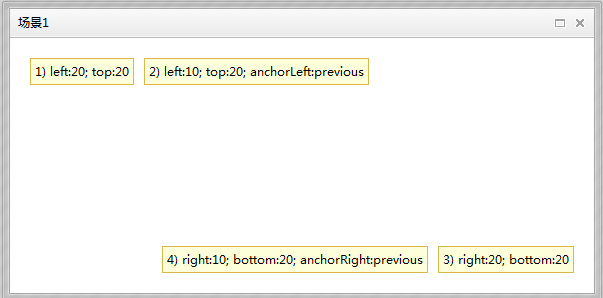


### 2）Anchor Layout锚定式布局

Anchor布局的基本思路是设定某控件的四周边框与容器或其他控件之间的锚定关系，以此来确定该控件实际所处的位置和尺寸。

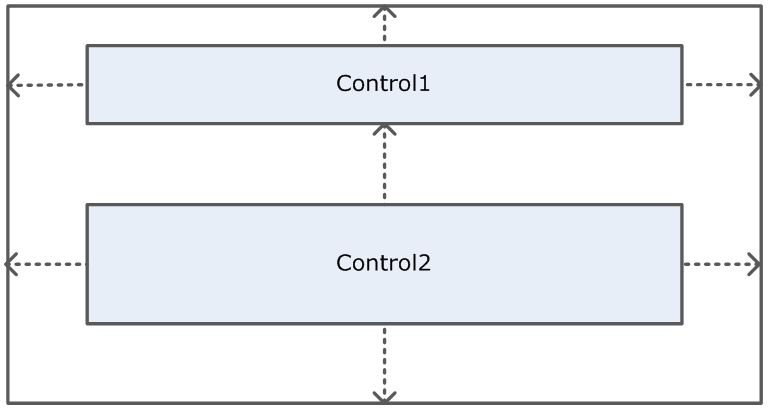
场景1（简单的边界锚定）

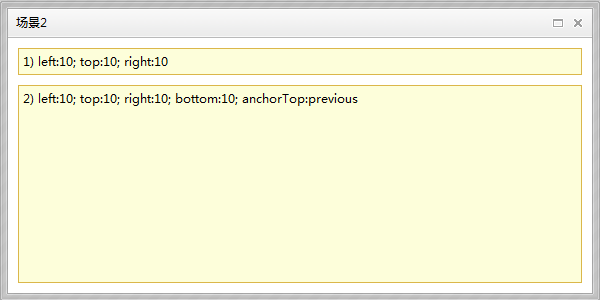




Control1设定与容器的left与top为一个具体值，因此在页面渲染的时候Control1会处在容器中一个固定的位置，而Control2与Control1有所差别，其top锚定到容器的一个具体值，使得它始终距离容器的上边界有一个固定值，但是其left没有直接锚定容器，而是相对的锚定到Control1，其left值表示相对Control1的一个相对距离。这样一旦我们调整Control1距离容器的left值的时候，会看到Control2跟着Control1走的效果。

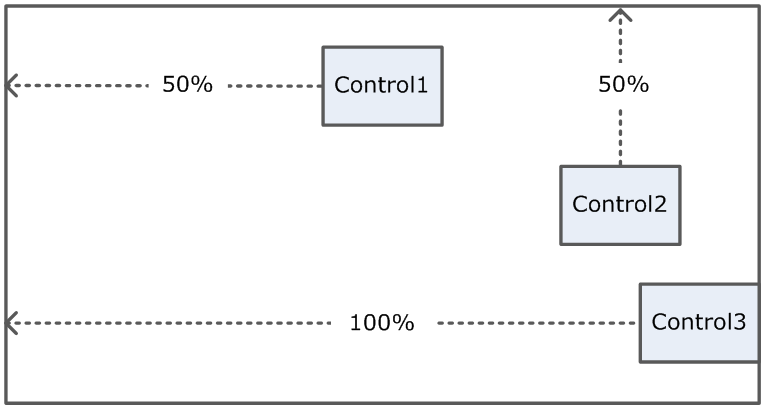
场景2（通过锚定关系来控制控件的尺寸）

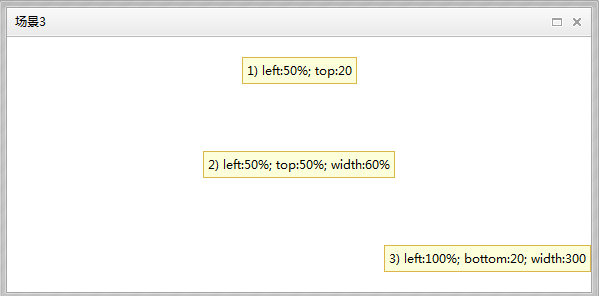




将Dialog的大小做调整，发现Control1的宽度会自适应调整，而Control2始终保持四个方向锚定距离不变。其实这个场景可以注意到它非常像DockLayout布局中Control1为top布局，Control2为Center布局。DockLayout其实是AnchorLayout的一种特例，事实上Dorado的DockLayout就是从AnchorLayout继承下来的。

场景3（利用百分比锚定实现居中等效果）





设定Control1的left的锚定值为50%，这样AnchorLayout最终运行的时候会使Control1产生一个居中的效果（与场景2居中效果不同的是，该处的控件的高度和宽度始终保持在一个固定值），产生这种居中效果的实现需要特别加以说明，50%并不是让Control1举例左边的锚定距离为50%，而是将容器宽度减去自身宽度后剩余宽度的50%，这样就恰好能实现控件居中的效果。当我们定义图中Control3的左边锚定距离为100%的话，其实际产生的效果恰好是贴在整个容器的最右边。同样Control2如果我们设容器top的锚定距离为50%，其实际产生的效果就是Control2上下居中。即使我们动态调整整个容器的大小，Control1,Control2,Control3的显示效果都不会有变化。

<View layout="anchor">

<Property name="title">AnchorLayout</Property>

<ToolBar layoutConstraint="top:0;left:0;right:0">

<ToolBarLabel>

<Property name="text">菜单栏</Property>

</ToolBarLabel>

</ToolBar>

<Panel layoutConstraint="anchorTop:previous;left:0;bottom:26">

<Property name="caption">导航树</Property>

<Buttons/>

<Children/>

<Tools/>

</Panel>

<Panel layoutConstraint="type:center;anchorLeft:previous;left:0;bottom:26;right:0;anchorTop:container;top:26">

<Property name="caption">工作空间</Property>

<Buttons/>

<Children/>

<Tools/>

</Panel>

<ToolBar layoutConstraint="bottom:0;left:0;right:0">

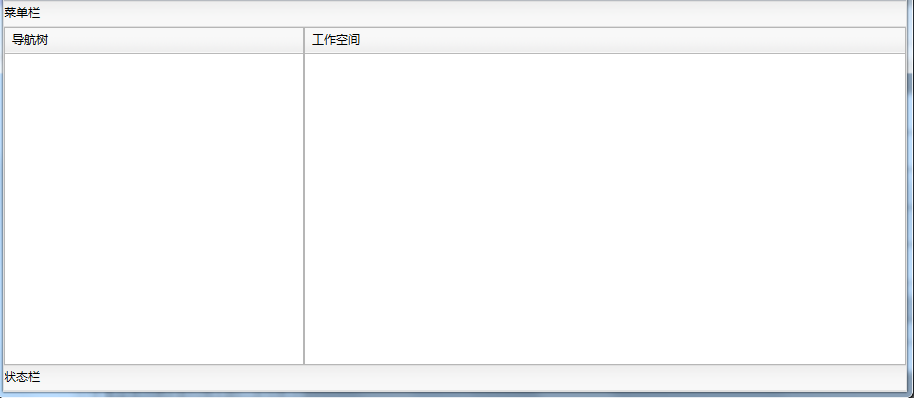
<ToolBarLabel>

<Property name="text">状态栏</Property>

</ToolBarLabel>

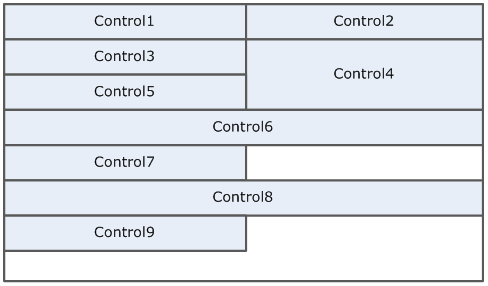
</ToolBar>

</View>



效果与DockLayout布局中一样，验证了DockLayout是AnchorLayout的子类。

### 3）FormLayout表单式布局



* 表单式布局首先把空间分成1到n列，然后由上往下依次排列各个子控件。
* 每个子控件可以选择自己要占用几行或几列。

当Control4需要占用两行时，可以看到Control3，Control4，Control5的展现效果，其中Control5和Control4的左边。而Control6需要占用两列，就会把容器的整行都占用了。然后Control7占用了第一列，当容器要放置Control8的时候发现Control7这一行的空间不够，它需要占用两列的控件，这样容器会把Control7控件后面的空间空出来，并将Control8放置在下一行中。这就是表单布局的基本特点。

### 4）HBoxLayout水平箱式布局和VBoxLayout垂直箱式布局

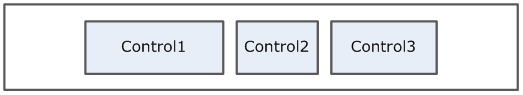
按照先后顺序进行横向布局或垂直布局。

这两种布局也提供了pack属性支持，设置内部元素的位置：start，center，end

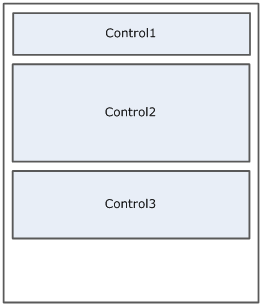
使HBoxLayout内部的元素靠左，居中还是靠右；

使VBoxLayout内部的元素顶天，居中还是立地。

HboxLayout



VBoxLayout

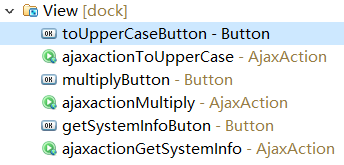


## 6、实做AJAX

AJAX技术问题：

* 如何调用Spring中bean中指定的业务方法；
* 如何传入一个单值给后台，并将调用结果为单值的值如何返回到前台；
* 如何传入一个map到后台，并将运算结果返回到前台；
* 如何调用后台的业务方法，并将计算结果为map的值返回到前台；

添加3个Button和3个AjaxAction：



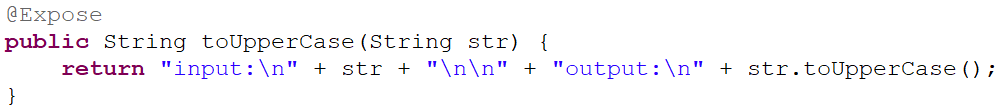
1）单击第1个按钮，调用spring中的一个bean的方法toUpperCase，并将我们传入的字符串转为大写，并返回到前台浏览器。

为ajaxaction设置属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 值 | 说明 |
| id | ajaxactionToUpperCase | 控件Id |
| caption | To UpperCase | 显示名称 |
| parameter | Hello World！ | 需要传给后台参数 |
| service | ajaxService#toUpperCase | 用于调用后台服务，ajaxService为注册到beanFactory中的beanId，getSystemInfo为方法名。 |

给按钮绑定action，表示点击按钮就会触发AjaxAction动作，执行后台方法。

准备后台方法（@Expose用于定义可暴露服务）：



AjaxAction执行成功后，应该获取后台方法的返回值展示到前台，给ajaxactionToUpperCase绑定onSuccess事件：

// @Bind #ajaxactionToUpperCase.onSuccess

!**function**(self) {

dorado.MessageBox.alert(self.get("returnValue"));

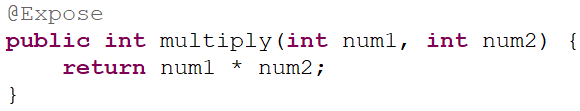
};

2）单击第2个按钮，调用spring中的一个bean的方法multiply，并将传入Map对象（含多个key）中两个数字相乘，将计算结果返回到前台浏览器。

为ajaxaction设置属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 值 | 说明 |
| id | ajaxactionMultiply | 控件Id |
| executingMessage | Cumputing... | ajaxaction执行中提示信息 |
| service | ajaxService#multiply | 调用后台服务 |

准备后台方法：



当后台方法mutiply()被调用时需要2个参数，希望给用户一个自定义输入参数的机会，给按钮绑定onClick事件：

首先获取ajaxactionMultiply这个动作，然后使用提示框，输入数字，中间用“，”隔开，将输入的参数定义为一个map，然后使用ajaxactionMultiply设置参数，并执行，含有一个回调函数（将后台返回的结果展示到前台）。

// @Bind #multiplyButton.onClick

!**function**() {

**var** action = view.get("#ajaxactionMultiply");

dorado.MessageBox.prompt("Please input two numbers here", {

defaultText : "3,5",

callback : **function**(text) {

**var** nums = text.split(",");

**var** parameter = {

num1 : nums[0],

num2 : nums[1]

};

action.set("parameter", parameter).execute(

**function**(result) {

dorado.MessageBox.alert(nums[0]+"\*"+nums[1]+"="

+result);

});

}

});

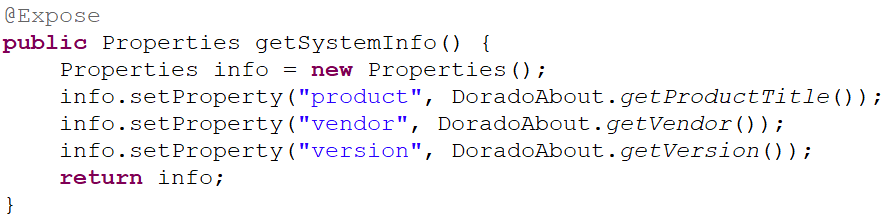
};

3）单击第3个按钮，调用spring中的一个bean的方法getSystemInfo，并将Java中的一个Map对象返回到前台浏览器中。

为ajaxaction设置属性：

|  |  |  |
| --- | --- | --- |
| 属性 | 值 | 说明 |
| id | ajaxactionGetSystemInfo | 控件Id |
| caption | Retrive Dorado7 System Information | 显示的名称 |
| service | ajaxService#getSystemInfo | 调用后台服务 |

准备后台方法（将信息存入Properties中）：



为按钮绑定action，表示点击按钮就会触发AjaxAction动作，执行后台方法。

AjaxAction执行成功后，应该获取后台方法的返回值展示到前台，给ajaxactionToUpperCase绑定onSuccess事件：

// @Bind #ajaxactionGetSystemInfo.onSuccess

!**function**(self) {

**var** info = self.get("returnValue");

dorado.MessageBox.alert("product："+ info.product + "\n" + "vendor："

+ info.vendor + "\n" + "version：" + info.version);

};

## 7、服务定位表达式

AJAX中提到服务定位表达式的概念，用来描述服务的查找方式，即Dorado7通过服务定位表达式来查找到具体服务，以便对服务进行调用。

协议:服务名[#方法名]

目前支持的协议（可扩展）：

* spring：表示其后台服务名是某个Spring上下文中的Bean的id。例如：spring:ajaxService#toUpperCase
* classpath；表示其后的服务名一个类的完全限定名，当Dorado7需要调用此服务时会自动按照单例模式来实例化该类。（此种方式不推荐使用）

可暴露服务：在Dorado7中所有的Ajax调用只能调用到那些明确声明为可暴露服务的Java方法。

在服务类中利用annotation机制实现了spring的bean的声明：

@Component

**public** **class** AjaxService {

另外在app-context.xml中配置组件扫描器：

<context:component-scan base-package=*"com.bstek.dorado.sample"* />

这样系统默认就会自动扫描到AjaxService类，并注册到Spring的BeanFactory中。而对于toUpperCase方法，采用@Expose标记：

@Expose

**public** String toUpperCase(String str) {

**return** "input:\n" + str + "\n\n"

+ "output:\n" + str.toUpperCase();

}

Dorado7会将这个方法自动注册到ExposedServiceManager中，其中服务名为ajaxService.toUpperCase，服务表达式为：spring:ajaxService.toUpperCase。annotation机制会自动将AjaxService的A自动转换为小写字母。

对于不采用annotation的系统可以手动的注册可暴露服务，在spring的配置文件中添加配置：

<bean parent="dorado.exposedServiceRegister">

<property name="services">

<map>

<entry key="ajax#toUpperCase" value="spring:ajax#toUpperCase" />

</map>

</property>

</bean>

如果有多个服务，在map中添加。

## 8、智能方法适配

智能方法适配是基于“约定优于配置”的设计思路而提供。这样在很多情况下不需要方法具有Dorado的特性，也不需要引入Dorado的开发规范。由Dorado7自动判断这个方法中的各个参数以及返回值的用途。这是一种职能化的处理逻辑。

优点：

* 几乎不需要引入任何Dorado7的API。
* 按照业务的需要而不是Dorado7的需要来为表现层编写代码。
* 可以令Dorado7与后台业务逻辑之间的粘合层代码更加简洁、易读。

例子中toUpperCase，Browser向服务器端传递的参数是我们在action的parameter属性中定义的，因为传过来的是一个字符串，智能方法在适配的时候会发现toUpperCase方法的参数就是一个字符串类型的，这样它就自动认为你想调用的方法就是toUpperCase，并将parameter中的值作为str中的值传入。这个方法调用结束之后返回一个字符串给外界，而ajax调用就自动的将这个返回值作为ajax调用的返回值，如ajaxactionToUpperCase的onSuccess方法：

// @Bind #ajaxactionToUpperCase.onSuccess

!**function**(self) {

dorado.MessageBox.alert(self.get("returnValue"));

};

在multiply范例中前台传入的是一个JSON对象，可以就认为是一个Map对象：

**var** parameter = {

num1 : nums[0],

num2 : nums[1]

};

action.set("parameter", parameter).execute(

**function**(result) {

dorado.MessageBox.alert(nums[0] + "\*"

+ nums[1] + "="+ result);

});

这样后台就拿到一个Map参数，然后对ajaxService这个bean进行方法匹配，会发现其中没有完全匹配的，这个时候智能方法适配会将parameters的参数拆开为num1，num2，这样就可以发现存在对应的multiply方法，并进行调用。并将结果作为ajax调用的结果返回到前台，直接将这个result显示出来。multiply范例体现的是智能方法适配过程中map自动拆解的功能。

### 1）适配算法

* 首先智能方法适配按照参数名进行适配：

即Dorado根据参数名来确定每个参数的作用。相对严谨的适配规则，但有时候不够业务化。例如对于AjaxAction的后端方法，其参数名必须为parameter。

如按照上面的规则，ajaxactionToUpperCase对应的Ajax方法就必须为如下的形式：

@Expose

**public** String toUpperCase(String parameter) {

**return** "input:\n" + str + "\n\n"

+ "output:\n" + str.toUpperCase();

}

这种方式有时不满足我们对业务方法的命名习惯，不利于Java代码的阅读。

* 按照参数类型进行适配：

Dorado根据参数类型来确定每个参数的作用。它存在的问题是：当参数较为复杂时Dorado可能无法准确的判断个参数的作用。

* Map：

对于Map类的参数自动方法适配会尝试将其拆解，用其中的键值对与方法参数进行适配。

* 混合式的适配规则（尚未提供）：

即根据名称规则进行一次适配，对于那些无法匹配的参数继续尝试通过类型规则进行适配。自动方法适配不会强行调用目标方法，当其认为适配可能存在歧义时会报错而不会继续执行方法调用。

## 9、HTML模板

### 1）Velocity页面模板

从示例的页面风格看出是一种开放式布局页面，其页面嵌入了一些Dorado的控件，如：

* 场景1中的按钮；
* 场景2中的一个在Dorado中叫ListBox的控件；
* 场景3中的一个面板控件。

Velocity页面模板VelocityTemplate.html：

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"

"http://www.w3.org/TR/html4/strict.dtd">

<html>

<head>

#outputPageHeader()

<style>

h1 {

font-size: 14pt;

}

h3 {

font-size: 11pt;

}

hr {

border: none;

border-bottom: 1px #77BDFD solid;

}

.place {

border: 3px red solid;

padding: 16px;

}

</style>

</head>

<body style="margin:24px; background:white url(images/summary-bg.png) no-repeat">

<h1>Velocity模板</h1>

<h3>场景1 - renderTo模式:</h3>

<div id="place1" class="place"></div>

<h3>场景2 - renderOn模式:</h3>

<div id="place2" class="place"></div>

<h3>场景3 - renderOn容器模式:</h3>

<div id="place3" class="place">

这里的内容是通过HTML模板定义的！

<p>

<img src="images/dorado7-logo-big.png">

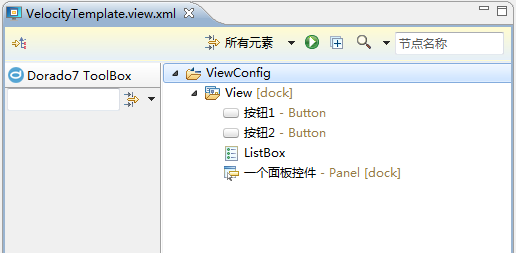
</p>

</div>

</body>

这是个普通的html，唯一不同之处是“#outputPageHeader()”这行代码，这样这个html就比较容易被美工人员在DreamWeaver或FrongPage中进行编辑。

再看下同目录下的VelocityTemplate.view.xml：



由于采用的是Velocity布局，因此在这个View中我们可以看到其中的空间是一个个罗列在View节点下的。View不再进行布局管理。

Velocity是怎么与View配合工作的，先了解View的一个特性：Dorado默认的资源文件加载的约定：与View同名同目录下的html、js、css文件可以被自动装载。

如VelocityTemplate.html和VelocityTemplate.view.xml文件名字是一样的，这样Dorado就默认认为VelocityTemplate.html是VelocityTemplate.view.xml的HTML模板页面，并自动装载。同理相关的js和css文件也是这样，如果我们希望加载不同文件名或不同资源路径的文件，则我们也可以通过View的几个属性进行设定：

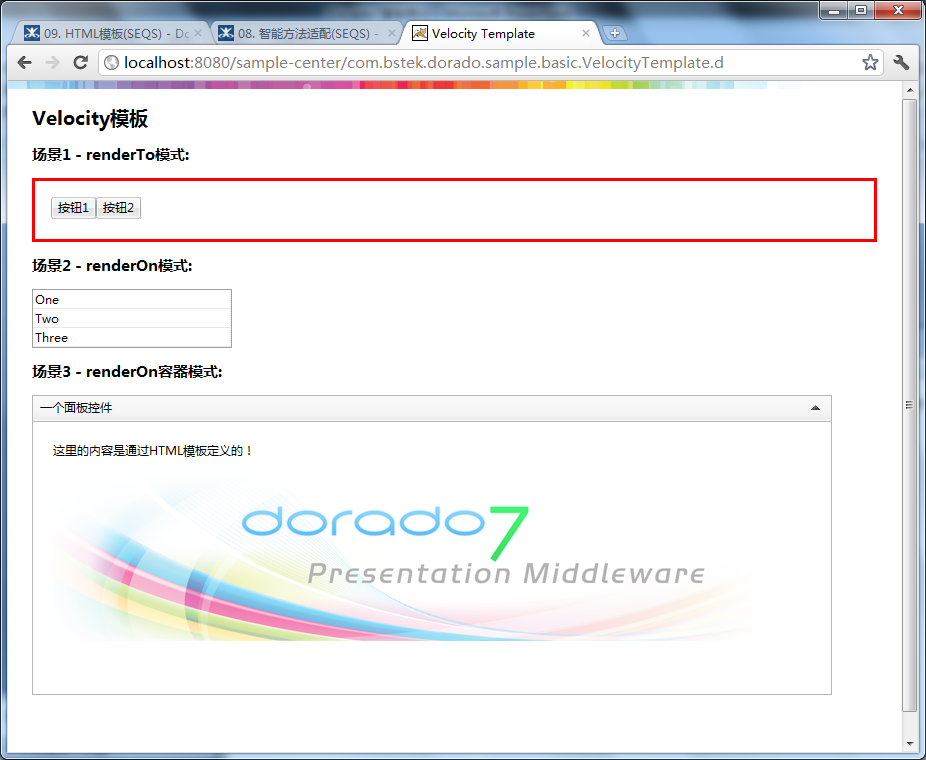
|  |  |
| --- | --- |
| 属性名 | 说明 |
| pageTemplate | HTML页面模板 |
| javascriptFile | js资源文件，多个js资源逗号隔开 |
| styleSheetFile | css文件，多个css文件逗号隔开 |

以上3个属性的值都支持协议的写法，如classpath:xxx或file:xxx

关于控件渲染到HTML模板的模式，Dorado设计了3种：

* renderTo模式
* renderOn模式
* renderOn容器模式

dorado中的所有控件都提供了renderTo和renderOn的两个属性。



renderTo模式

其中的Button1和Button2被渲染到了页面中带红边框的一个DIV中，在XML定义的时候设置其renderTo属性：

<Button caption="按钮1" renderTo="#place1"/>

<Button caption="按钮2" renderTo="#place1"/>

“#place1”，这是css中的一种写法：在dorado中目的是表示要查找HTML模板中的id为“place1”的DOM对象，并将当前控件渲染进去。在对应的HTML模板中可以找到这个DIV：

<div id="place1" class="place"></div>

这个DIV本身设置了css：

.place {

border: 3px red solid;

padding: 16px;

}

显示效果为一带红色边框的DIV对象。renderTo功能就是把button1和button2渲染到这个DIV的内部。place1这个DIV还会被保留。

renderOn模式

通过对renderTo的了解，应该还有一种可以把目标DIV替换的渲染方式，如页面上的ListBox控件：

<ListBox items="One,Two,Three" renderOn="#place2"/>

与Button控件采用了renderTo的方式不同，ListBox采用了renderOn的方式进行渲染。HTML模板的设定：

<div id="place2" class="place"></div>

place2与place1的class值相同，正常来将它们在浏览器中显示的时候都应该有一个红色边框，但实际运行的时候由于ListBox采用了rednerOn的模式，导致整个DIV被替换了，也就无法看到红色边框。

renderOn容器模式

当在某些场景下美工人员可能希望他可以定义DIV的内部元素，也就是说希望在renderOn的运行处理过程中只替换外部的DIV，这种模式已经被renderOn支持了：

<div id="place3" class="place">

这里的内容是通过HTML模板定义的！

<p>

<img src="images/dorado7-logo-big.png">

</p>

</div>

上面的place3中包含了一段文字说明以及一个图片的定义。而在View中定义的控件如下：

<Panel layout="padding:20" caption="一个面板控件" collapseable="true"

height="300" renderOn="#place3" width="800"/>

这儿用一个Panel控件替换place3最外面的那个DIV，这样在浏览器中我们最终看到的效果中，那段文字和图片都得到了保留。不过需要注意的是这儿只能使用容器类的控件才能实现这种效果。

### 2）JSP页面模板示例

基本的使用方法与Velocity模板一致。包括三种控件的渲染方式：renderTo、renderOn、renderOn容器。

与Velocity的唯一差别是View与模板的关系是通过View的pageUri属性设定的。

JSP模板页面需要引入Dorado的标签库：

<%@ taglib uri="http://www.bstek.com/dorado/taglib-7.0" prefix="d" %>

在<head>标签中加入：

<d:PageHeader />

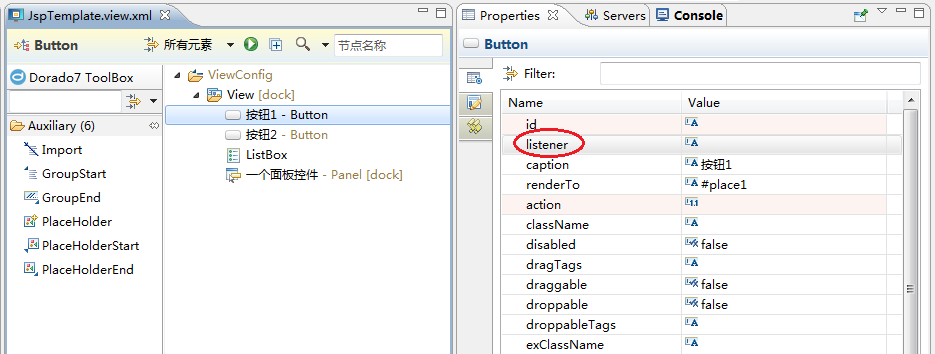
而Velocity中用的是#outputPageHeader()

（通过pageUri的运行规则，也可以使用其它模板，例如Velocity Tools、FreeMarker等，都可以将它配置到View的pageUri中作为Dorado的View的HTML模板）。

## 10、对象监听器

在某些场景下，我们可能希望在视图的初始化过程中通过自己的Java代码动态的改变视图的内容，例如：动态的添加控件或动态的调整已有控件中的部分属性。这样便于我们结合服务器端的某些外部条件进行页面显示的控制，例如个性化控制，权限控制等。要实现这些功能就有必要了解Dorado提供的对象监听器。

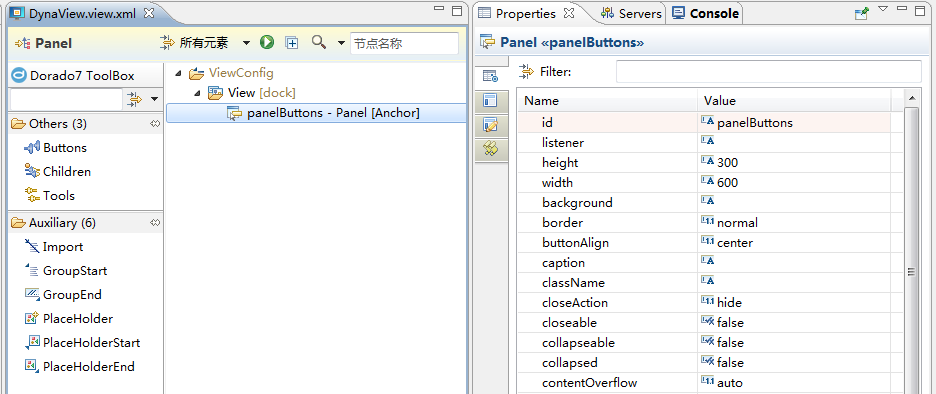
对象监听器并不是专门为View设计的，事实上Dorado中的所有控件都可以提供了一个listener属性，用以支持对象监听器功能。如前面我们看到的JspTemplate.view.xml在IDE视图中我们就能看到这个属性：



在listener中可以定义1到多个对象监听器，下面我们根据一个实例了解对象监听器的基本用法：



在这个例子中通过循环动态的创建了8个Button对象，首先看DynaView.view.xml：



定义了一个空的Panel对象，View的listener属性：

spring:dynaView#onViewInit

这是一个服务定位表达式，对于对象监听器而言，这表示通过Spring中id为dynaView的onViewInit方法监听这个View的创建过程。DynaView.java：

（用for循环创建8个Button对象，并利用Panel的addChild方法添加到Panel中）



这当中的@Component方法的声明主要的目的是将当前的类注册到Spring的上下文当中。由于此处不是一个Ajax调用,因此就不需要@Expose声明。该处方法名可以自由定义，只要保证与listener中的定位表达式一致就可以，参数也将采用智能方法适配规则，可以根据需要定义。例如本例我们希望向panelButtons对象中动态的添加8个Button，我们就可以直接将onViewInit方法写成这种样子：

public void onViewInit(Panel panelButtons) {

}

智能方法适配规则会自动将当前View中id为panelButtons且类型为Panel的对象注入进来。

（注：不要试图在对象监听器中定义两个名称一样，但是参数不一样的方法，它会导致对象监听器拒绝服务，直接报错！）

### 1）onInit监听和beforeInit监听

对象监听器有两种监听模式，一种是onInit，另一种是beforeInit。对一个控件而言，这表示我们如果要在控件创建的过程中添加监听，即可以在初始化之前监听它，也可以在这个控件初始化之后去监听它。初始化是指dorado读取View中的XML信息初始化到控件对象上，无论是beforeInit还是onInit，其中的控件都是已经被实例化好的，差别在于是否进行过初始化。在前面的范例中我们在View中的listener中定义的服务表达式是：

spring:dynaView#onViewInit

应该注意到这个地方并没有使用onInit方法，这是在Listener处理机制中有一种默认约定：一个监听器究竟使用哪一种监听方式，取决于该方法的方法名中是否使用before前缀，例如beforeInit。因此我们要理解onInit和beforeInit并不是具体的方法名，它是监听模式的含义。

另外注意几点：

* Dorado中绝大多数的可配置对象都有listener属性，可以定义一到多个监听器用以监听对象的创建过程。多个对象监听器之间用逗号隔开配置到listener属性中；
* 定义监听器时可以使用自动方法适配。

### 2）了解ViewConfig和View

对于一个View配置文件而言，ViewConfig是最顶层的节点。在Dorado的设计中，View节点主要用于定义最终需要输出到浏览器中的可见和不可见的对象。但是对于一个视图而言(不是指View节点)它的生命周期有几种不同的形式，例如我们访问一个视图的时候和这个视图向后台发出一个AJAX请求的时候，对于视图而言这是不同的物理状态，在Dorado内部处理过程中，View节点只会在前一个生命周期中被创建，在AJAX请求中它没有必要创建，而ViewConfig对象则会在所有的生命周期中创建。知道了这个区别之后，我们在使用对象监听器时，我们就要清楚的知道同样一个监听器把它添加在View的listener属性中和添加到ViewConfig的listener属性中，被激活的频率是不一样的，很显然ViewConfig中的listener中触发的频率更高，因此在实际使用中我们就要注意，如果只是希望对View的可见不可见组件进行属性定制，则把这个对象监听器添加到View或View的控件对象上就可以，而如果希望在AJAX请求过程中也触发的逻辑则应该添加到ViewConfig的listener中。

View监听器中控件的自动注入：  
对于View的监听器，它提供了控件自动注入的功能，在提供控件自动注入功能之前，我们在View的监听器中如果要获取View中的其他控件，可能就会采用如下较为笨拙的方法：

public void onInit(View view) {

Button buttonOK = (Button) view.getComponent("buttonOK");

buttonOK.setDisabled(true);

Panel panelMain = (Panel) view.getComponent("panelMain");

panelMain.setCollapsed(true);

}

但是View监听器提供了自动注入功能，就可以将上面的代码改写为：

public void onInit(Button buttonOK, Panel panelMain) {

buttonOK.setDisabled(true);

panelMain.setCollapsed(true);

}

如果希望在代码编程中针对某一个控件进行编码，可以直接通过参数定义，只要保证控件的类型正确和控件的ID与View中的ID保持一致就可以。自动注入功能会帮助你找到这个控件对象并在调用onInit方法的时候自动传入。

### 3）全局对象拦截器

对象监听器，它一次只能监听一个具体的View或一个控件对象。并且要在View中配置这个监听器，当在某些场景下我们希望做全局监听，便于做统一的设定，如：统一定制某一种控件的特性或通过这个入口做权限设定等。

为此dorado提供了全局对象监听器的配置，全局对象监听器可以监听所有对象的创建过程。并且不需要在View中做任何配置，我们只要直接在dorado-home的app-context.xml中这么配置：

<bean parent="dorado.genericObjectListenerRegister">

<property name="listener">

<bean class="com.xxx.xxx.ViewInterceptor" > //自定义的Java类

<property name="order" value="3" />

<property name="pattern" value="\*" /> //设定过滤规则

</bean>

</property>

</bean>

基本代码结构：

public class ViewInterceptor extends GenericObjectListener<View> {

@Override

public boolean beforeInit(View view) throws Exception {

}

@Override

public void onInit(View view) throws Exception {

}

}

全局监听器必须继承自GenericObjectListener，与对象监听器一样，它支持两种监听模式：beforeInit和onInit。这样这个类能监听所有的View，除了上面介绍的pattern属性使用通配符进行过滤外，还可以在Java中通过泛型进行过滤，如上述范例中使用了View通配符，表示监控所有的View，如果改为Button则表示监控所有的Button：

public class ViewInterceptor extends GenericObjectListener<View> {

@Override

public boolean beforeInit(Button button) throws Exception {

}

@Override

public void onInit(Button button) throws Exception {

}

}

### 4）全局监听器范例

调整DataGrid表格列默认操作功能的实例：将默认单击表个头代开表格列选择的功能关闭，并调整为表格列排序的功能。

public class DataGridListener extends GenericObjectListener<DataGrid> {

public boolean beforeInit(DataGrid grid) throws Exception {

return true;

}

public void onInit(DataGrid grid) throws Exception {

List<Column> columns = grid.getColumns();

for (Column column:columns){

if (column instanceof ColumnGroup){

List<Column> subColumns = ((ColumnGroup) column).getColumns();

for (Column subColumn:subColumns){

if (supportSort(subColumn)) attachSortEvent((DataColumn)subColumn);

}

}else{

if (supportSort(column)) attachSortEvent((DataColumn)column);

}

}

}

private void attachSortEvent(DataColumn column){

List<ClientEvent> listeners = column.getClientEventListeners("onHeaderClick");

if (listeners!=null && listeners.size()>=1){

//do nothing

}else{

String script = "var sortState = self.get('sortState');" +

"if (sortState=='asc'){"+

"self.get('grid').sort(self, true);"+

"}else{"+

"self.get('grid').sort(self, false);"+

"};";

column.addClientEventListener("onHeaderClick", new DefaultClientEvent(script));

}

}

private boolean supportSort(Column column){

return (column instanceof DataColumn) ;

}

}