金蝶Apusis Portal Server前端梳理

前言

本文档所面向的对象是APS全组人员。

文档主要讲解了APS前端相关的核心模块，目的是知识共享，让APS组内成员对前端核心模块有个大致了解，以后维护或扩展这些模块时，能够很快的入门和定位，从而节减学习成本。

如果你是组内新成员，首先说句：“感谢你的加盟”，你应该按照章节顺序去阅读。如果你是测试人员，只需要阅读前两章，对前端常见概念有个大概理解即可。如果你是开发人员，你在阅读完前两章后一定要认真学习一下第三章，这里记录了一些代码细节与相关实现。

版本历史

|  |  |  |  |
| --- | --- | --- | --- |
| **时间** | **更新人** | **版本** | **备注** |
| 2012-12-18 | 刘笑天 | 1.0 | 创建文档模板及目录规划 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

[前言 2](#_Toc344380905)

[版本历史 3](#_Toc344380906)

[目录 4](#_Toc344380907)

[第1章 基本概念 7](#_Toc344380908)

[1.1 页面 7](#_Toc344380909)

[1.2 页面装饰 8](#_Toc344380910)

[1.3 布局 9](#_Toc344380911)

[1.4 Portlet 10](#_Toc344380912)

[第2章 页面是怎么生成 12](#_Toc344380913)

[2.1 创建页面 12](#_Toc344380914)

[2.1.1 新建页面步骤 12](#_Toc344380915)

[2.1.2 底层剖析 12](#_Toc344380916)

[2.2 请求页面 12](#_Toc344380917)

[2.2.1 请求步骤 12](#_Toc344380918)

[2.2.2 底层剖析 13](#_Toc344380919)

[2.3 聚合渲染 14](#_Toc344380920)

[2.4 添加一个portlet 16](#_Toc344380921)

[2.4.1 添加portlet步骤 16](#_Toc344380922)

[2.4.2 底层剖析 16](#_Toc344380923)

[第3章 深入了解技术内幕 19](#_Toc344380924)

[3.1 页面 19](#_Toc344380925)

[3.1.1 Folder 类型 20](#_Toc344380926)

[3.1.2 Page 类型 21](#_Toc344380927)

[3.1.3 Link 类型 24](#_Toc344380928)

[3.2 片段 25](#_Toc344380929)

[3.2.1 Fragment 25](#_Toc344380930)

[3.2.2 ContentFragment 26](#_Toc344380931)

[3.2.3 片段与页面 27](#_Toc344380932)

[3.3 布局 28](#_Toc344380933)

[3.3.1 概述 28](#_Toc344380934)

[3.3.2 NestedLayoutPortlet 31](#_Toc344380935)

[3.4 主题 34](#_Toc344380936)

[3.4.1 Decoration 36](#_Toc344380937)

[3.4.2 DecorationValve 38](#_Toc344380938)

[3.5 聚合渲染 40](#_Toc344380939)

[3.5.1 AggregatorValve 40](#_Toc344380940)

[3.5.2 PageAggregator 41](#_Toc344380941)

[3.5.3 PortletAggregator 42](#_Toc344380942)

[3.6 Ajax API 43](#_Toc344380943)

[3.6.1 Ajax相关Pipeline 43](#_Toc344380944)

[3.6.2 ajax-pipeline 44](#_Toc344380945)

[3.6.3 ajax-direct-pipeline 54](#_Toc344380946)

# 基本概念

## 页面



上图是Portal的页面，它由以下几个部分组成：

1. 主题(标记1的桔黄色部分)

主题（Theme），是美化页面的作用，例如页面中的图片、背景色、色调搭配、文字样式、操作控件样式、导航样式等等。页面和主题的关系你可以简单的理解为人和衣服的关系。

1. 布局(标记2的灰色部分)

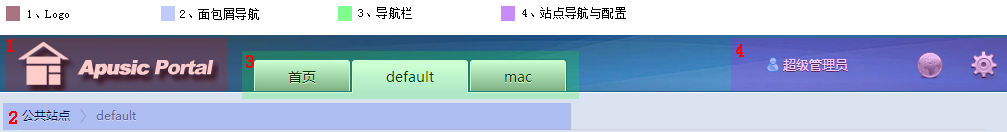
布局（Layout）是用于控制Portlet的位置与间距，例如上图是一个一行三列的布局。

1. Portlet（标记3的浅绿色部分）

它就是一个个的功能组件，页面和Portlet可以简单的理解“苹果操作系统”和“App应用”的关系。在页面中它占据了一块块的小区域。

## 页面装饰

主题由装饰（Layout Decoration）和布局组成。装饰分为页面装饰和Portlet 装饰。如下图展现的就是页面装饰：



上图是一个页面的组成部分，通常由以下几部分构成：

1. Logo(标记1的褐色部分)

企业的徽标

1. 面包屑(标记2的蓝色部分)

它是一种位置导航，可以清楚的让用户知道自己所在的网站的位置

1. 主导航栏(标记3的绿色部分)

用于跳转到目标的页面

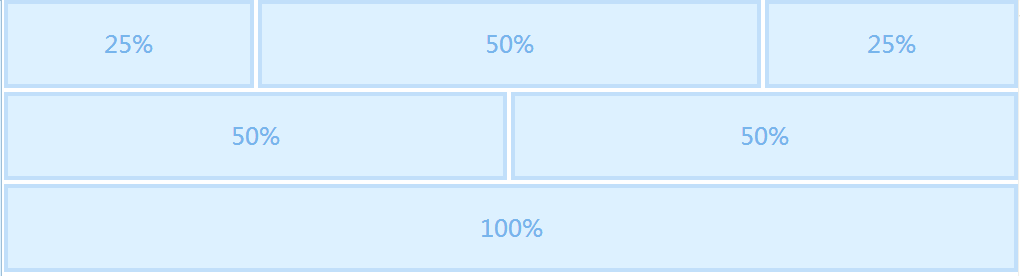
1. 其他(标记4的紫色部分)

用户展现用户信息。球状的按钮是可以切换站点的，齿轮转的按钮是操作控件，可以进行添加Portlet、注销等。

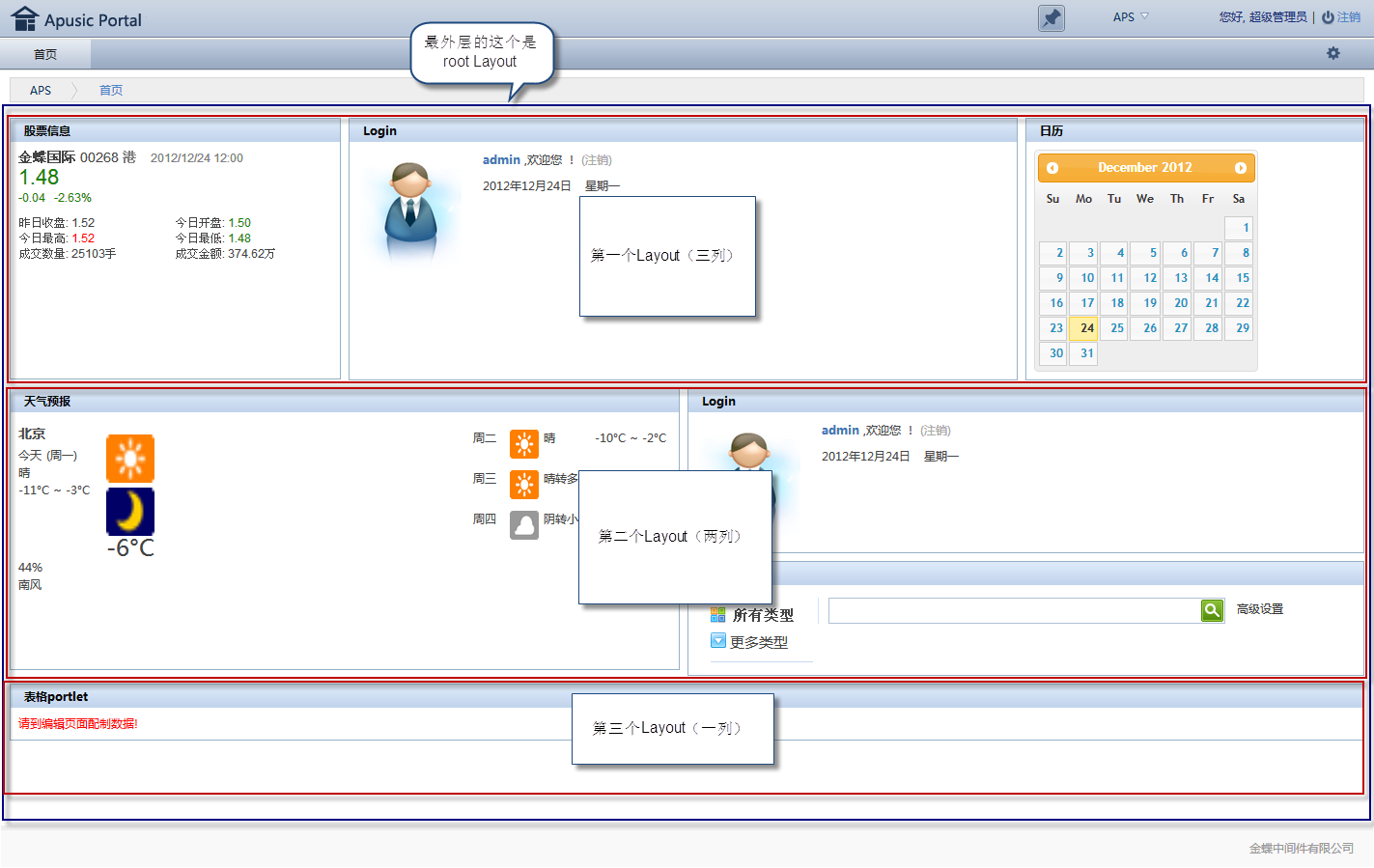
Portlet 装饰将在1.4中详细介绍。

## 布局

布局是所有Portlet的容器，它的作用就是控制Portlet的位置与间距。如下图：



设计视图



实际视图

如图我们了解到，布局是由树状结构的片段组成：

1. 根片段

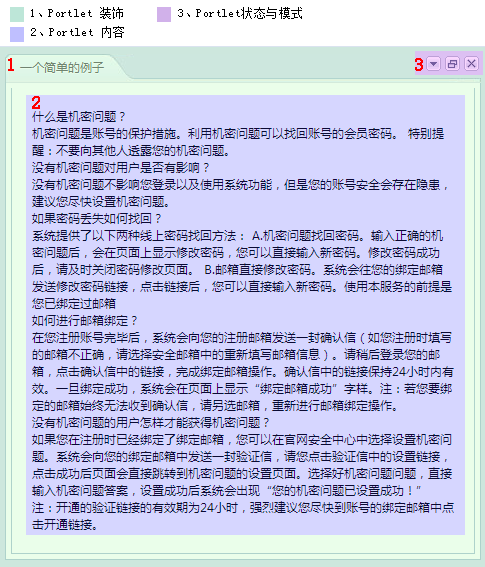
根片段（Root Fragment）

1. Layout 片段

Layout片段(Layout Fragment）。一个片段就是一行，一个片段里面放入无限个Portlet，例如上图中的机构是：

第一行放了a Portlet，第二行放了 b、c Portlet，第三行没有放入Portlet。

## Portlet



上图就是一个Portlet，它由以下几个部分组成：

1. Portlet装饰（标记1的绿色部分）

Portlet装饰（Portlet Decoration）。它环绕着Portlet内容并进行美化，它和页面装饰是一样是可以自己开发的。

1. Portlet内容（标记2的紫色部分）

Portlet 内容。

1. Portlet状态与模式（标记3的粉色部分）

Portlet状态与模式全称是 Portlet 状态与窗口模式 (portlet state and window state )，用于Portlet的基本操作，例如切换Portlet窗口的“最小化”、最大化”、编辑模式等。

1. Portlet 片段

Portlet 片段(Portlet Fragment)。它是Portlet载体，这个载体是唯一的，即使你在页面看到两个相同的Portlet，但是他们的载体是不一样的，当然从视觉上是看不出来的。

# 页面是怎么生成

## 创建页面

### 新建页面步骤

此处省略操作步骤，详细请查看“用户手册，管理控制台-页面管理模块”。

### 底层剖析

从上一章节中我们了解页面是装载了主题、布局、装饰等信息，那么它在后台是怎么被存储的呢？

这里引入的一个概念----PSML。它是XML格式的文件，描述了一个页面所应该拥有的信息。

（[点击进入](http://portals.apache.org/jetspeed-2/devguide/guide-psml.html)详细了解PSML概念）

完成新建后，一个页面就以XML的形式保存下来了。如果你使用数据库存储的，只不过是将PSML 转换成表的形式，概念是一样的。

## 请求页面

### 请求步骤

通过导航或者输入URL（例如<http://192.168.6.205:6888/aps/portal/default-page.psml>），就可以访问一个portal页面了，由于页面具有权限控制，所以不同权限的人可能被导航到不同的页面，也可能导航到相同的页面后看到的内容不同。

### 底层剖析

Portal系统接受到请求后，会进入一个[jetspeed-pipeline](http://portals.apache.org/jetspeed-2/devguide/guide-pipeline.html)，通过下图了解整个请求过程：



通过URL 进入Portal系统，系统后台是通过jetspeed-pipeline来处理请求的。Jetspeed-pipeine 有很多Vavle，这里简单列举了与渲染相关的Valve：

1. [PageProfilerValve](http://portals.apache.org/jetspeed-2/devguide/guide-profiler.html)

根据自定义规则和权限，计算出当前用户可访问的页面。

1. DecorationValve

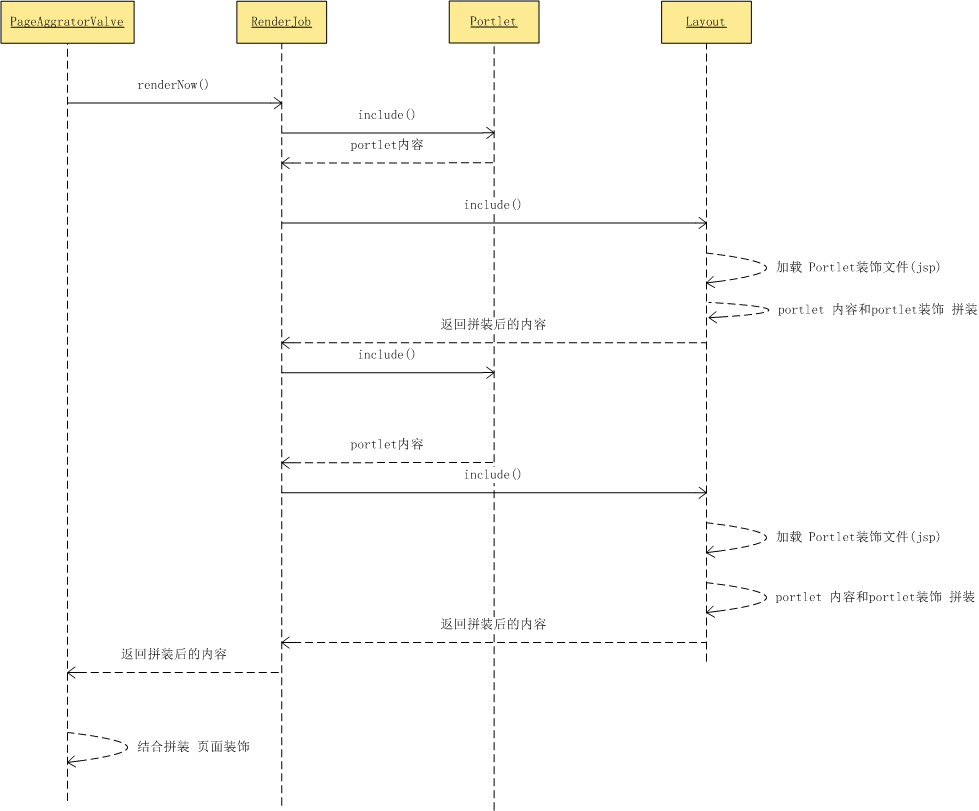
上一节我们了解到，页面就是一个PSML，里面记录了需要什么样的装饰数据，所以这个阶段是获取相应的页面装饰的数据模型：LayoutDecoration 和 PortletDecoration

1. PageAggratorVavle

对页面进行聚合渲染，也就是将页面装饰、布局、Portlet装饰和Portlet内容等拼装到一起，合并的结果是HTML页面。

## 聚合渲染

我们从上一节了解到，渲染的结果是HTML文件。下图是一个页面有两个Portlet的时候的渲染时序图，通过它进一步了解页面是怎么渲染出来的：



* 首先PageAggratorValve会对每一个Portlet创建一个RenderJob，RenderJob专门用于渲染Portlet。
* RenderJob渲染Portlet的结果就是将Portlet对应的页面加载并解析为HTML片段，然后将片段交给Layout处理。
* Layout 获得HTML 片段，通过PSML预先存储好的装饰和布局位置，进行拼接渲染。
* 最后将生成好的HTML片段与页面装饰进行拼接，就是一个完整的页面了。

生成的页面就会由Portal系统返回给用户。

## 添加一个portlet

### 添加portlet步骤

此处省略操作步骤，详细请查看“用户手册，页面添加portlet 操作”。

### 底层剖析

从上一小节中我们了解到，页面聚合是所有的HTML片段组成页面。但是用户在页面添加一个portlet 的时候，使用局部刷新是更好的用户体验。

这里引入目前局部刷新使用的两个技术概念：

1. AJAX

异步的页面请求（这里不做详细解释），这个是前端的技术解决方案。

1. AJAX-piepline

类是前面的piepline，区别在于前者计算出来一个完整的页面，后者返回一个portlet 片段的页面。这个是后端技术的解决方案。

通过下面的流程图，简单地了解一下整个AJAX-piepline的流程：



当你拖拉一个portlet 到相应位置的时候，会触发局部刷新功能，首先计算出将要添加到layout的位置，然后开始与后台交互，后台的两个关键点：

1. layoutValve

AJAX-piepline有许多的valve，这里我们只需要关心layoutValve就可以了，它是将请求映射具体的AJAX-API的。

1. AJAX-API

后台暴露的AJAX-API，我们简称为ajaxAction。URL中action参数用于映射相应的AJAX-API，（例如：<http://192.168.6.205:6888/aps/ajaxapi?action=addPortlet> 就是调用添加portlet 的请求）。当然AJAX-API返回的格式是可以自己定义HTML或则JSON等。

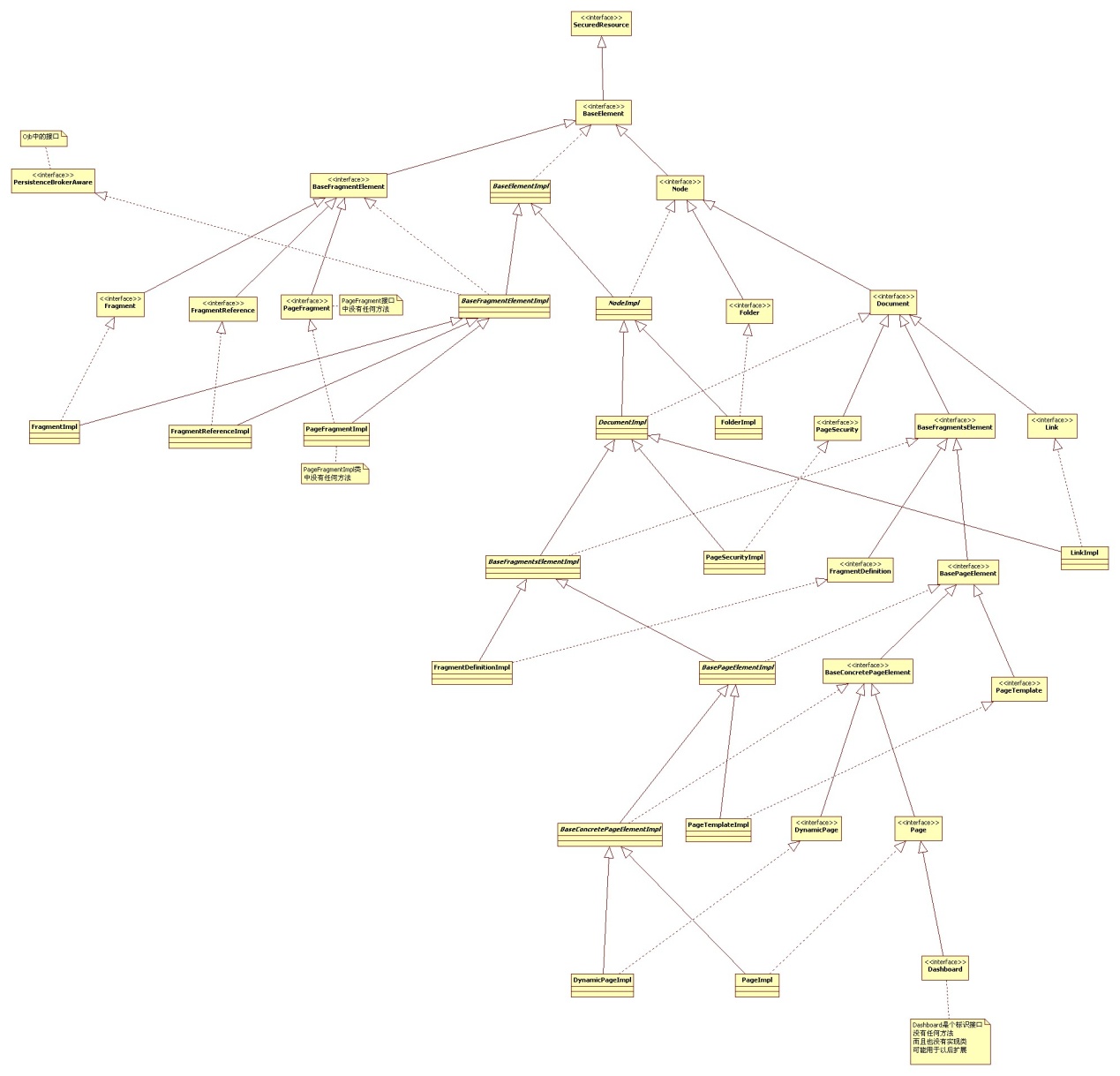
最后，浏览器接受到AJAX成功的状态后，就会做出相应处理了。整个过程就完成了。

# 深入了解技术内幕

## 页面

在第二章中我们已经了解了什么是页面，并且已知页面在后台是以PSML进行描述的。在系统代码中，页面其实只是“基本元素”家族的一个分支，如下图（原图已经上传至wiki）：

<http://wiki.apusic.net/pages/viewpageattachments.action?pageId=20848223&sortBy=date&highlight=BaseElement.jpg&>



我们主要关注3个接口：

* Folder
* Page
* Link

### Folder 类型

Folder 可以直译为“目录”。它可以包含目录、页面、链接。例如：<http://192.168.6.205:6888/aps/portal/default-page.psml>，portal就是目录，default-page.psml就是页面。

在程序中，目录被抽象为Folder接口，通过类图关系，我们进一步了解Folder的结构：



从图中可以了解到，Folder 下面可以有多个Folder、Page、Link.

Folder存储在folder表，其常用字段如下表：

|  |  |
| --- | --- |
| 字段字段名 | 作用 |
| Path | 路径 |
| Name | 名称 |
| Title | 标题 |
| Is\_hidden | 是否隐藏 |
| Skin | **功能未确定，可以考虑用于目录中页面的高级样式定制** |
| Default\_layout\_decorator | 页面装饰 |
| Default\_portlet\_decorator | Portlet装饰 |
| Default\_page\_name | 页面标题 |
| Subsite | 页面所处站点 |
| User\_principal |  |
| Owner\_principal | 创建者 |

### Page 类型

系统代码中，页面被映射为两个接口：

* Page
* ContentPage

#### Page

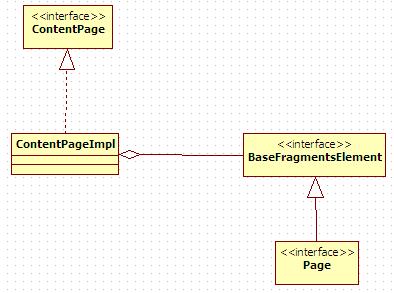
Page接口是PO （Persistant Object持久化对象），它被用于呈现一个门户页面。APS使用PageManager接口调用Page接口对数据库的page表读写数据。

页面相关数据存储在page表中，常用字段见下表：

|  |  |
| --- | --- |
| 字段字段名 | 作用 |
| Path | 路径 |
| Name | 名称 |
| Title | 标题 |
| Is\_hidden | 是否隐藏 |
| Skin | **功能未确定，可以考虑用于页面的高级样式定制** |
| Default\_layout\_decorator | 页面装饰 |
| Default\_portlet\_decorator | Portlet装饰 |
| Subsite | 页面所处站点 |
| User\_principal |  |
| Owner\_principal | 创建者 |

#### ContentPage

ContentPage接口是BO （Business Object 业务对象）。它与Page接口是聚合关系，如下图：



从ContentPage接口的注释来翻译它们的关系：

ContentPage是一个只读版的Page。ContentPage对象用于渲染，而Page对象是用于持久化的、单实例的metadata对象，它不能用于保持每次请求内容。ContentPage接口提供了与Page相似的接口，并且能够通过当前用户的请求来渲染请求内容。

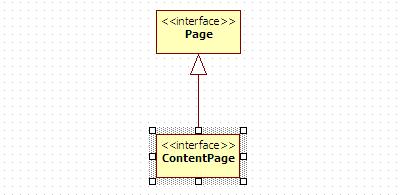
#### Page与ContentPage转化

我们经常遇到需要对这两个接口进行转化的场景。

在实际转化中，绝大部分的场景是需要将ContentPage转化为Page。

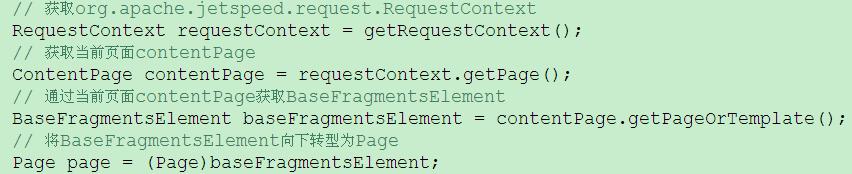
在开始讲明它们的转化方式前，需要声明的一点是：

在Jetspeed2.2.0中，ContengPage继承自Page，如下图：

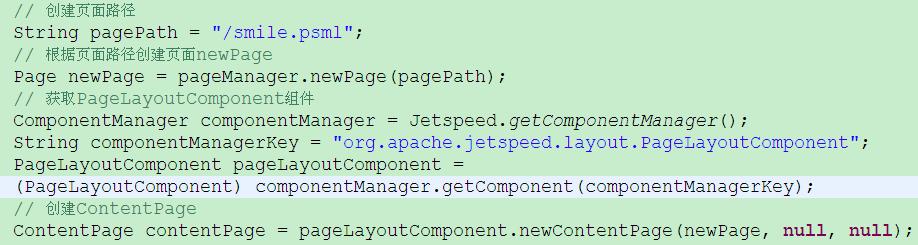


所以我们在Jetspeed2.2.0版（对应APS8.0之前的版本）中，只需要通过代码(ContentPage)page即可轻松转化。但是在Jetspeed2.2.2版（对应APS8.0版）中，由于接口改变的关系，我们不能像以前一样轻松转化了。

ContentPage转化为Page的代码如下图：



Page转化为ContentPage的代码如下图：



在实际逻辑中很少有用到Page转为ContentPage的，而且也应该尽量避免这种代码逻辑。因为它比较“笨重”，而且容易出错。笨重表现在我们需要通过Spring获取一些组件才能完成逻辑。容易出错表现在ContentPage被创建之后大部分属性被设置为null，在为Jetspeed容器调用的过程中必定报NullPointerException。

### Link 类型

Link直译为“链接”。它用于导航到某个页面，例如链接可以指向某个目录下的某页面(/aps/other/other.psml)或其他网络中某个页面（http://www.google.com）。

Link 的打开方式是可以配置为新窗口与当前窗口两种方式。

在程序中，链接在系统中被抽象成Link接口。其映射的表结构主要的字段如下：

|  |  |
| --- | --- |
| 字段字段名 | 作用 |
| Path | 路径 |
| Name | 名称 |
| Title | 标题 |
| Is\_hidden | 是否隐藏 |
| Skin | **功能未确定，可以考虑用于页面的高级样式定制** |
| target | 打开方式 |
| URL | 链接 |
| User\_principal | 用户权限 |
| Owner\_principal | 创建者 |

## 片段

片段分为layout与portlet两种，通过Fragment.LAYOUT与Fragment.PORTLET来标识，简单理解它们的区别就是：

layout类型的片段可以嵌入layout片段或者portlet片段，但是portlet类型的片段无法再嵌入任何片段。layout片段就像一个容器，它可以盛放layout片段也可以盛放portlet片段。

片段的最终用途是生成页面的HTML片段的，所以片段离不开页面，它必须寄托于页面才可以展现给用户。同样，页面也离不开片段，页面正是有了片段才能形成有实际意义的完整的内容展现出来。

类似于页面，片段也被抽象为了两个接口：

* Fragment
* ContentFragment

### Fragment

Fragment接口是PO （Persistant Object持久化对象），它用于表示页面上被聚集渲染的片段。APS通过Page、PageManager等接口调用它与数据库的fragment表同步，fragment表中的常用字段见如下表：

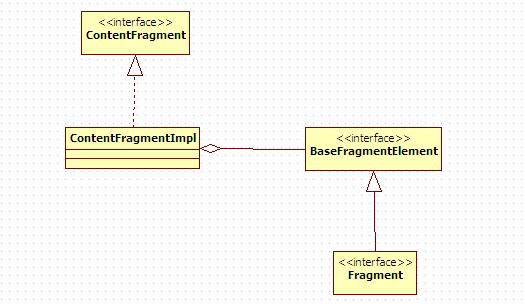
|  |  |
| --- | --- |
| 字段名 | 作用 |
| Fragment\_id | 自增主键 |
| Parent\_id | 父片段的主键，fragment\_id与parent\_id将各个片段建立起一种树形关系 |
| Page\_id | 当前片段所在页面的id |
| Fragment\_string\_id | 用于唯一标识一个片段，唯一字段 |
| Name | 片段的名称，唯一字段 |
| Type | 片段类型，分Portlet、Layout两种 |
| Skin | **功能未确定，可以考虑用于Portlet装饰的高级样式定制，但是需要确认它与page表的skin字段是否具有覆盖关系** |
| Decorator | Portlet装饰 |
| Layout\_row | Portlet在布局片段的行索引 |
| Layout\_column | Portlet所布局片段的列索引 |
| Layout\_sizes | 如果该片段是个布局片段，该字段显示了改布局片段的列数 |

### ContentFragment

ContentFragment接口是BO （Business Object 业务对象）。在ContentFragment中的注释与ContentPage的注释几乎是一模一样的：

ContentFragment是一个只读版的Fragment。ContentFragment对象用于渲染，而Fragment对象是用于持久化的、单实例的metadata对象，它不能用于保持每次请求内容。ContentFragment接口提供了与Fragment相似的接口，并且能够通过当前用户的请求来渲染请求内容。

所以我们不难理解， ContentFragment与ContentPage是类似的东西，它们用于封装Fragment、Page这种PO，并呈现于页面。我们可以通过ContentPage获取到ContentFragment，通过Page获取到Fragment。同样，与ContentPage和Page接口的关系一样，Jetspeed2.2.0版，ContentFragment继承自Fragment，但是Jetspeed2.2.2中变成了聚合关系，如下图：



在Jetspeed2.2.2版中，本身ContentFragment与Fragment是无法直接相互转化的，我们只能依靠ContentPage与Page转化后，然后通过它们来获取对应的ContentFragment或Fragment。

### 片段与页面

一个页面被创建之后，会被嵌入唯一一个根片段（Root Fragment），且该根片段被设置为layout类型。通过下图进一步了说明：



* 根片段可以拥有自己的子片段，这些子片段都是layout类型的，每一个子片段在页面中将被渲染为代表“行”的片段。
* 子片段也可以拥有自己的子片段，它们是portlet类型的，最终被页面渲染为代表Portlet的Div片段。

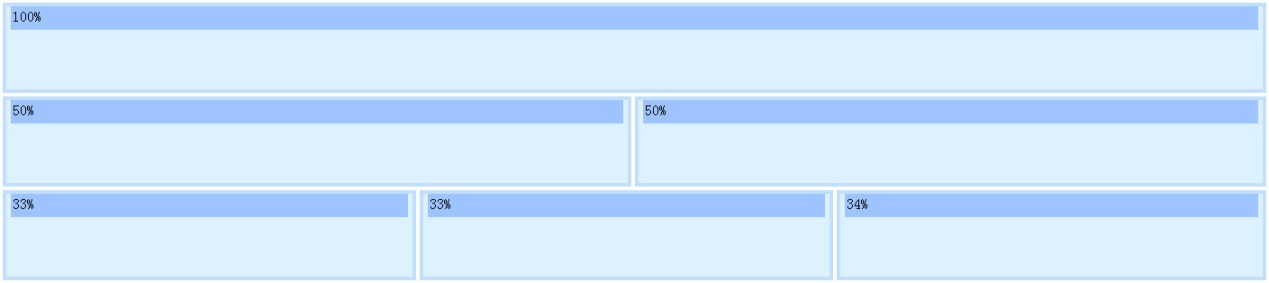
laylout类型的片段中可以放置layout片段，但是portlet类型的片段中无法放置任何片段，所以Portlet就是整棵“片段树”的叶节点，而根片段是这棵树的根。

## 布局

### 概述

所谓的布局就是一种layout片段，它可以盛放Portlet，因此Portlet的排列及间距等都是由布局决定的。而布局正是名为[NestedLayoutPortlet](#_NestedLayoutPortlet)的抽象，NestedLayoutPortlet就是一个系统级别的Portlet，它作为layout类型片段存在，每一个layout类型的片段都是一个NestedLayoutPortlet。我们约定一下，下文中有些地方会将NestedLayoutPortlet叫做“布局Portlet”，其他的所有Portlet叫做“普通Portlet”。

当一个页面被创建时，会被嵌入唯一一个根片段（Root Fragment，，并将属性type设置为layout。它又拥有自己的子片段（至少一个，type同样也被设置为layout类型），这些子片段就是作为“行”布局存在。“行”布局又拥有自己的子片段（type被设置为portlet类型）。举例来说，如下图：



图中页面的分为3行，总共有4个layout片段，分别是:根片段

* 第一行的layout片段

列数为1，宽度为100%

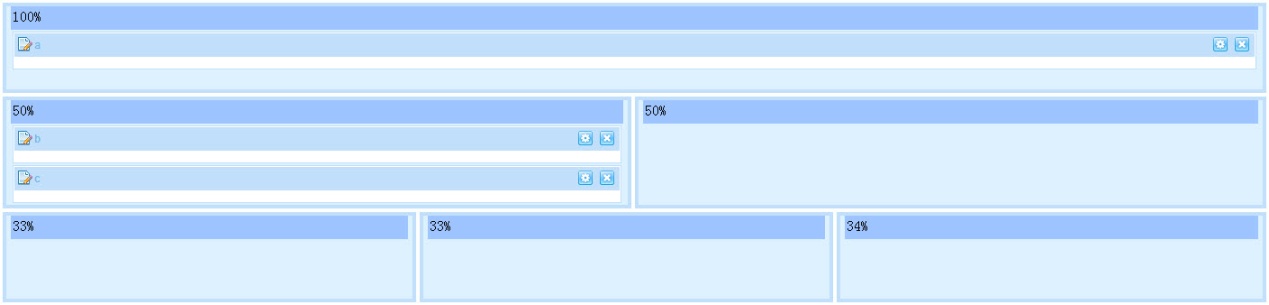
* 第二行的layout片段

列数为2，宽度分为50%、50%

* 第三行的layout片段

第三行分为三列，宽度为33%、33%、34%

如果我们添加了一些Portlet，如下图：



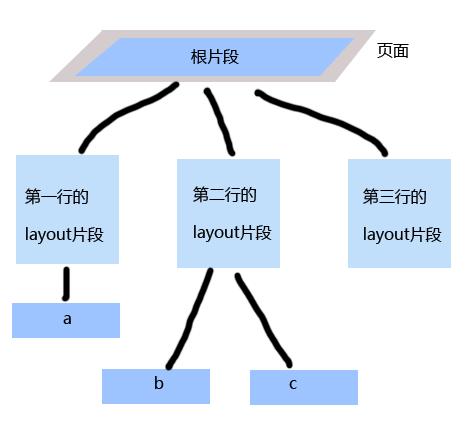
第一行中添加了1个名为a的Portlet，第二行的第二列添加了名为b和c的Portlet各一个，此时，页面上总共有7个片段，它们分别是：

* 根片段
* 第一行的layout片段
* 代表a的portlet片段
* 第二行的layout片段
* 代表b的portlet片段
* 代表c的portlet片段
* 第三行的layout片段

以上各个layout类型的片段（也就是布局Portlet）嵌套在一起，形成了行、列，它

们可以盛放普通Portlet。

它们的组织关系如下图：



从图中可以看出，页面中片段的组织方式是一棵树，它以根片段为根，行级layout片段为枝干，各个portlet片段为叶。根据布局的约束，根片段下一定是代表行的layout片段，行layout片段下面一定只能是portlet片段（即Portlet）。所以，页面上的每一行是由根片段下的一级layout片段来表示，每一行的列是通过行layout片段的layoutSizes属性来表示（即fragment表中的Layout\_sizes字段），例如第一行的值为100%，第二行的值为50%,50%，第三行的值为33%,33%,34%。所有片段都是存储在fragment表中的，每一个片段都对应了一条数据。各个片段之间的关系在fragment表中通过id、parent\_id、type字段体现，例如：

在fragment表中我们找到任意一个type值为portlet的片段，通过它的parent\_id可以找到它父容器（一个代表“行”的布局Portlet），然后通过父容器的parent\_id找到的片段就是“根片段”，而根片段的page\_id指向了当前页面。

### NestedLayoutPortlet

之前说过作为layout片段的接口是NestedLayoutPortlet。在创建页面的时候，APS会自动创建一个根片段,然后将它的layoutSizes属性设置为100%，并将名称设置为apusic-ps-layout::NestedLayoutPortlet，type的值设置为”layout”。页面的模式（查看模式、编辑模式）的切换，就是作为“根片段“的NestedLayoutPortlet的模式。我们在查看模式下看到的页面也就是NestedLayoutPortlet的ViewPage（APS8.0中是nested-layout-view.jsp），而我们用于设计布局的编辑页面就是它的EditPage（APS8.0中是nested-layout-edit.jsp）。编辑页面主要有三块功能组成：

* 选择模板
* 设计布局
* 页面信息

#### 选择模板

该功能提供了一些默认常用模板，用户只需要点击“应用”按钮后就能直接将页面上划分成新布局。这些模板是展现在designLayout.jsp中的，数据的准备也是在NestedLayoutPortlet中进行的。这些模板是通过一些XML文件定义出结构的，这些定义布局的模板文件被存放在apusic-ps-engine/webapp/WEB-INF/layouts下，它的结构如下图：



<description>定义布局描述。

<name>定义布局名称。

<img>定义布局的图片。

<rows>由一组<row>组成用于

<row>定义布局的行

<column>定义布局的列宽

布局模板文件还通过layouts.xsd进行了约束，如果语法有错误，会被编辑器显示异常。

这些文件如何被加载并显示的呢？

点击“选择模板”的按钮，会展开选择模板的面板。这些面板在访问编辑页面的时候已经在Edit模式所对应的edit()方法中准备好了数据。在edit()方法中通过LayoutUtil.getLayoutTemplates()方法读取apusic-ps-engine/webapp/WEB-INF/layouts下的所有XML文件并解析为JSON数组，然后传递给前台页面，前台页面的JS代码就调用这个JSON数组数据生成各个布局模板：



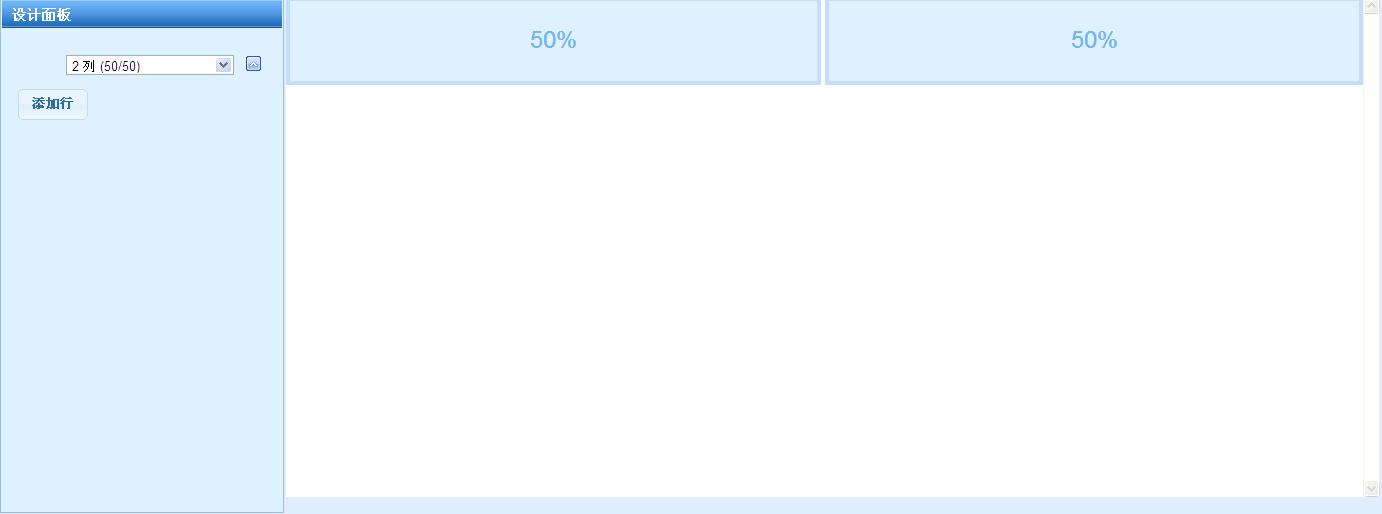
各个布局模板是使用基于jQuery代码生成，在创建之初，表示布局模板的JSON数据已经被存储在模板内部。同时，“应用”按钮绑定了事件：

点击，通过Ajax触发Resource请求将保存的代表布局模板的JSON数据向后台传递。数据会被NestedLayoutPortlet的updateDesignLayoutInfo()方法接受并处理，处理的逻辑是调用LayoutUtil. updateDesignLayoutInfo()方法完成，核心代码就在该方法中。处理逻辑：将当前传递过来的JSON数据解析后，对应生成新的片段（以Fragment），并组装在一起，然后被设置到页面（Page），最后调用PageManager同步数据库。

#### 设计布局

如果提供的默认布局模板无法满足用户，还可以设计布局。

点击“设计布局”按钮后跳转到设计页面，可以添加行，也可以对选中的行进行分列。点击“设计布局”按钮，会触发Resource请求，被NestedLayoutPortlet的getDesignLayoutPage()方法所处理。该方法调用LayoutUtil.getLayoutDesignInfo()方法完成逻辑，getLayoutDesignInfo()将当前页面的根片段解析，生成JSON格式的数据，该数据描述了当前页面上所有的片段。作为Resource请求的结果，它被传递到前台，前台会使用JS代码调用该结果生成布局的设计页面：



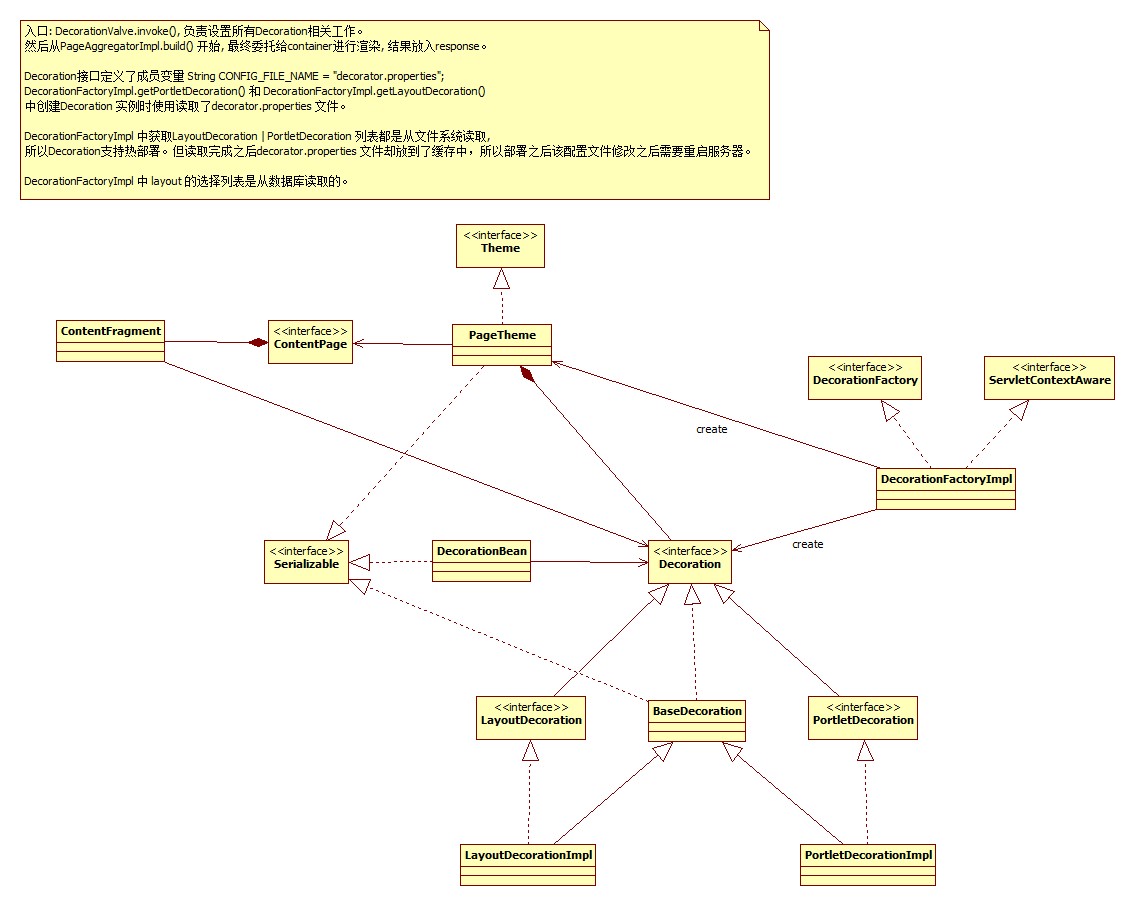
设计完毕后，按“保存”按钮后会同步到数据库（将片段同步到fragment表），并将页面跳转到查看模式。点击“保存”按钮的后台处理逻辑与[选择模板](#_选择模板)的“应用”按钮的处理逻辑是一模一样的，不再赘述。

#### 页面信息

用于编辑当前页面的信息，如标题、选择页面装饰或则portlet 装饰

## 主题

在1.2中我们了解到主题是装饰和布局组成的。在程序中，主题被抽象成一个Theme，通过下面类图关系了解Theme家族：



从图中可以看出，Theme持有各种数据模型，例如Decoration、Fragment等，而这些数据在渲染的时候是需要到的，所以Theme就是一个为渲染准备的数据模型，其中：

* ContentPage 是整个类图的的起始点
* ContentFragment 持有对Decoration 的引用
* Theme 持有对每个的 Decoration 的引用
* DecorationFactory 持有对Theme 和 Decoration 的引用

在1.2我们了解到，装饰分为：

* 页面装饰
* Portlet装饰

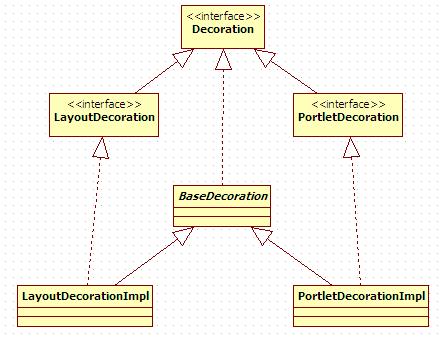
装饰被定义为环绕于fragment周围的静态或半静态的标记（html，xhtml,jsp，velocity等），装饰通常使用Velocity或JSP模板技术来渲染。可以认为fragment提供数据，decoration 提供样式。

在程序中,“装饰“被抽象为Decoration接口。

### Decoration

对应于[片段](#_片段)的两种类型，Decoration也分为layout和portlet两种。

类图如下：



从图可以看到Decoration分为layout类型的LayoutDecoration与portlet类型的PortletDecoration两种。它们有各自的实现类，而且它们的实现类都继承自BaseDecoration，BaseDecoration定义了装饰相关的“算法骨架”。

下面是它们的主要作用：

* Portlet 类型的 Decoration

在页面上，它会以HTML（XHTML、VXML 等）的方式包裹住portlet。

在程序中，被描述成PortletDecoration，为 Portlet 类型的Decoration渲染的时候提供获得样式、标题、操作按钮的接口。

在持久化上，目前以 skin.jsp + css 进行存储。

* Layout 类型的 Decoration

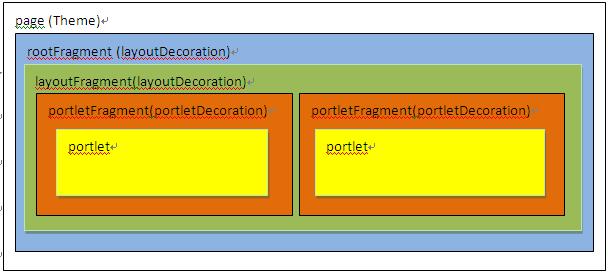
在页面上，它会以HTML（XHTML、VXML 等）的方式包裹住layoutFragment，它为一layout中的portlet提供了公用的样式，但是可以被portletDecoration中的样式覆盖掉。

在程序中，被描述为LayoutDecoration，为Layout类型的Decoration渲染的时候提供样式、标题、操作按钮。

在持久化上，目前以theme.jsp + css 进行存储。

装饰是可以自定义开发的，如果想进一步了解开发方式，请参考“皮肤开发手册”。

Decoration 在一个页面中的关系如图：



从图可以看出：

* 一个页面有且仅有一个rootFragment（根片段），而且被theme所包含
* rootFragment（根片段）下可有零到多个layoutFragment，持有一个layoutDecoration
* 一个layoutFragment可包含零个到多个portletFragment，持有一个layoutDecoration
* 一个portletFragment只能包含一个portlet，持有一个portletDecoration

装饰在页面上还体现出两个重要部分：

* 状态
* 模式

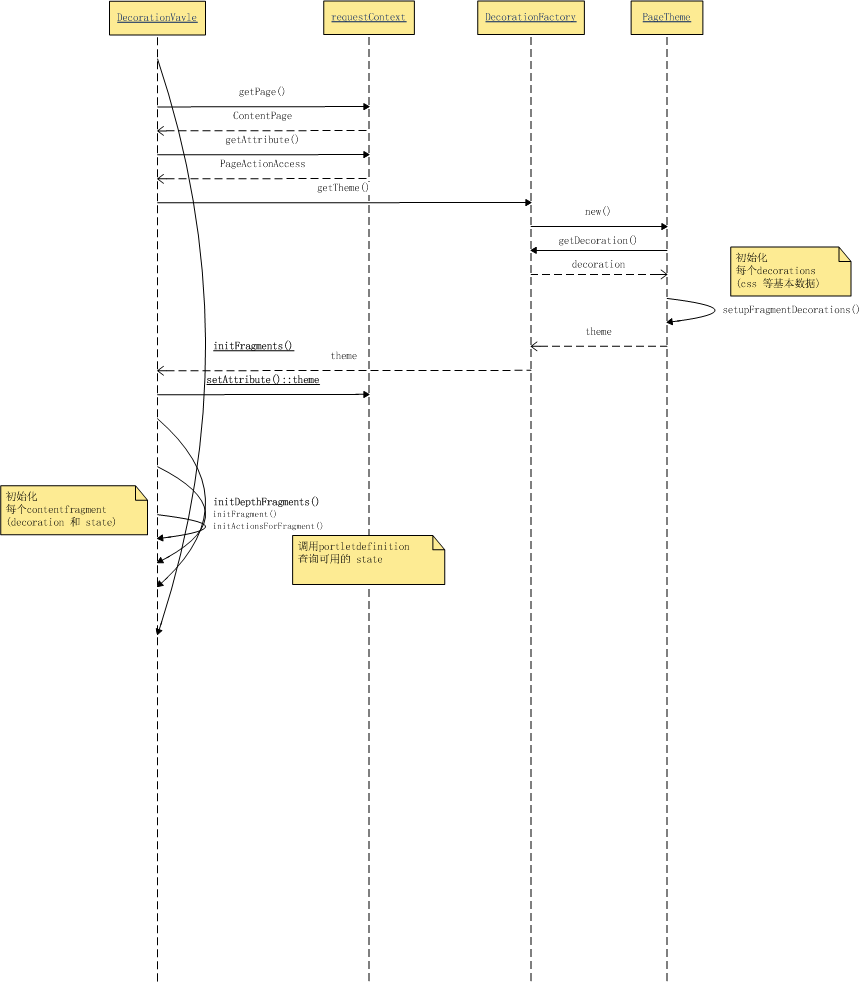
页面的“根片段”具有模式，它被用作生成页面的模式链接，即“查看页面”与“编辑页面”。但是页面上其他的layout片段不提供模式，页面上portlet类型的片段（Portlet）提供了模式操作的同时也提供了状态的操作控件，如下图：



页面上所有的Portlet类型Decoration与根片段的Layout类型Decoration共同组成了一个叫做Theme的概念，也就是“主题”。

### DecorationValve

整个Decoration 的初始化是在pipleline 中 DecorationValve 中完成，其目的是为后面的pageAggragtionValve 准备数据模型。通过下面时序图，我们进一步了解是如何构建出数据模型的：



整个流程完成贯穿着一个 initFragments()的调用：

* 从 requestContext 获得 page 对象，此page 在profileValve 中生成
* 从 requestContext 获得 pageAccessAction 对象，此对象包含了整个页面的权限
* 通过DecorationFactory 生成当前页面的theme
* 再通过DecorationFactory 为decoration生成相对应的decoration并注入到theme 中
* 完成theme 的初始化，将其注入到 requestContext 以待后面进行处理
* 为每个Portlet初始化它所对应的ContentFragment和操作按钮
* 在初始化操作按钮的时候，需要结合portletDefinition和pageActionAccess 才能生成 (只有根片段和portlet类型的片段才会注入操作按钮)

## 聚合渲染

聚合渲染就是由Portal容器将一个个片段“翻译”为html片段并最终拼凑成整个HTML文档的过程。这个过程主要涉及到以下2个关键接口:

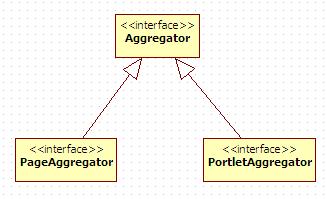
* AggregatorValve
* Aggregator

### AggregatorValve

AggregatorValve作为jetspeed-pipeline的一个Valve（阀门）存在，对于普通的访问，都会进入该Valve。该Valve的代码逻辑很简洁，就是调用Aggregator接口的build()方法。因此该接口仅仅是作为聚合渲染的入口而存在。对于聚合渲染本身而言，APS中又将它分为2种，它们都继承自Aggregator，是各种Aggregator实现类的超接口。

* 页面聚合渲染 PageAggregator
* 单个Portlet聚合渲染 PortletAggregator

继承图如下：



Aggregator被注入Valve，在注入的时候根据实际的功能选择PageAggregator或PortletAggregatord的实现类。PageAggregator主要是作为一个Valve的关联接口，用于完成页面的渲染。PortletAggregator同样也是作为一个Valve的关联接口，通过Ajax请求，来渲染Portlet并将渲染出的html片段返回给前台，以供Js代码调用。

### PageAggregator

该接口用于将页面上所有的片段进行聚合渲染，它从根片段开始递归渲染，并将渲染后返回的html片段聚合在一起设置到RequestContext中，最终会由theme.jsp使用。theme.jsp中的${APS\_PAGE\_CONTENT}就是被渲染并聚合在一起的结果，它最终以html代码的形式被嵌入整个theme.jsp中被解析。

PageAggregator主要做了以下几件事：

1. 设置参数，从而决定是否对Js、Css、图片等静态资源压缩
2. 判断是否当前页面是否最大化状态页面，并相应选择是按照最大化状态还是普通状态来渲染。

所谓的渲染就是通过pluto容器将Jsp或者Velocity等动态页面加载并解析为html的过程。

1. 为前端准备数据
2. 载入theme.jsp
3. 释放片段资源

流程图如下：

开始

是否有最大化状态Portlet

按普通状态渲染

按最大化状态渲染

否

是

准备前端使用的数据，例如装饰，渲染后的html片段等，并向前端传递

将当前页面，以根片段开始解析，将它作为根布局解析为JSON格式的数据并向前端传递

加载theme.jsp文件，使用上述准备的数据，展现出整个页面

结束

### PortletAggregator

该接口用于Portlet的渲染。它与PageAggregator类似，被注入到某个Valve，通过特定的Pipeline就可以访问到它。它做的工作内容比较简单：

根据前台传递的参数（parentId）获取Portlet，渲染后将内容以html片段的形式返回给前台页面。

APS只使用到它的子接口APSPortletAggregator。添加Portlet的时候，发送Ajax请求的URL中带有/apsportlet，从而会走apsportlet-pipeline。apsPortletValve是apsportlet-pipeline中一个Valve，它正是使用APSPortletAggregator完成功能，也就是上述：将指定Portlet渲染成html片段返回给前台页面。前台Js使用这些页面生成了页面上可见的Portlet窗口（Div）。

## Ajax API

Ajax API是由Jetspeed所提供的，用于处理AJAX请求的接口，它基于XML数据格式与部分JSON数据格式。APS中的经典应用场景有：

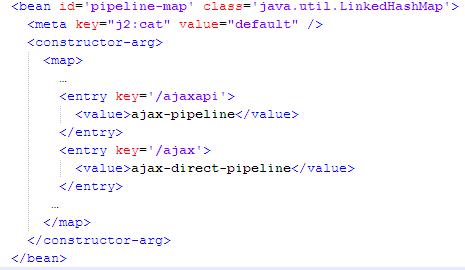
* Portlet的添加、移动、删除
* Portlet模式、状态的获取
* 装饰的获取、修改
* Portlet面板

Ajax API也是通过Pipeline被访问到的，访问Ajax API的Pipeline有两种：

* ajax-pipeline
* ajax-direct-pipeline

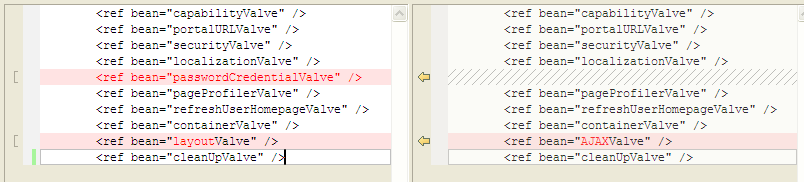
### Ajax相关Pipeline

请求一个URL时，后端根据URL的不同选择Pipeline，Pipeline和URL的对应关系是配置在pipelines.xml中，代码片段如下图：



对ajax-pipeline的访问正是通过/ajaxapi，对ajax-direct-pipeline的访问是通过/ajax。

这两个Pipeline的区别在Valve的配置，如下图红色标记部分:



### ajax-pipeline

与/ajaxapi访问相关所使用的AjaxAction实现类的相关配置在ajax-layout.xml中。而与/ajax访问相关的配置在ajax.xml中。

ajax-pipeline中用于处理Ajax请求的入口是LayoutValve。对于ajax-pipeline而言，涉及到以下几个核心接口：

* LayoutValve
* APS中，我们只能通过Pipeline访问后台，LayoutValve就是/ajaxapi的入口
* AjaxRequestService
* 它定义了/ajaxapi的后台处理逻辑的“骨架（模板）”
* AjaxAction
* 用于处理前台通过/ajaxapi发送的Ajax请求
* AjaxBuilder
* 上述Ajax请求处理完毕后的后续工作，主要是封装数据并返回前台页面
* Constant

没有提供任何方法，仅仅作为一个常量接口存在。提供了前后台传递所约定的参数

下图是访问ajax-pipeline的时序图：

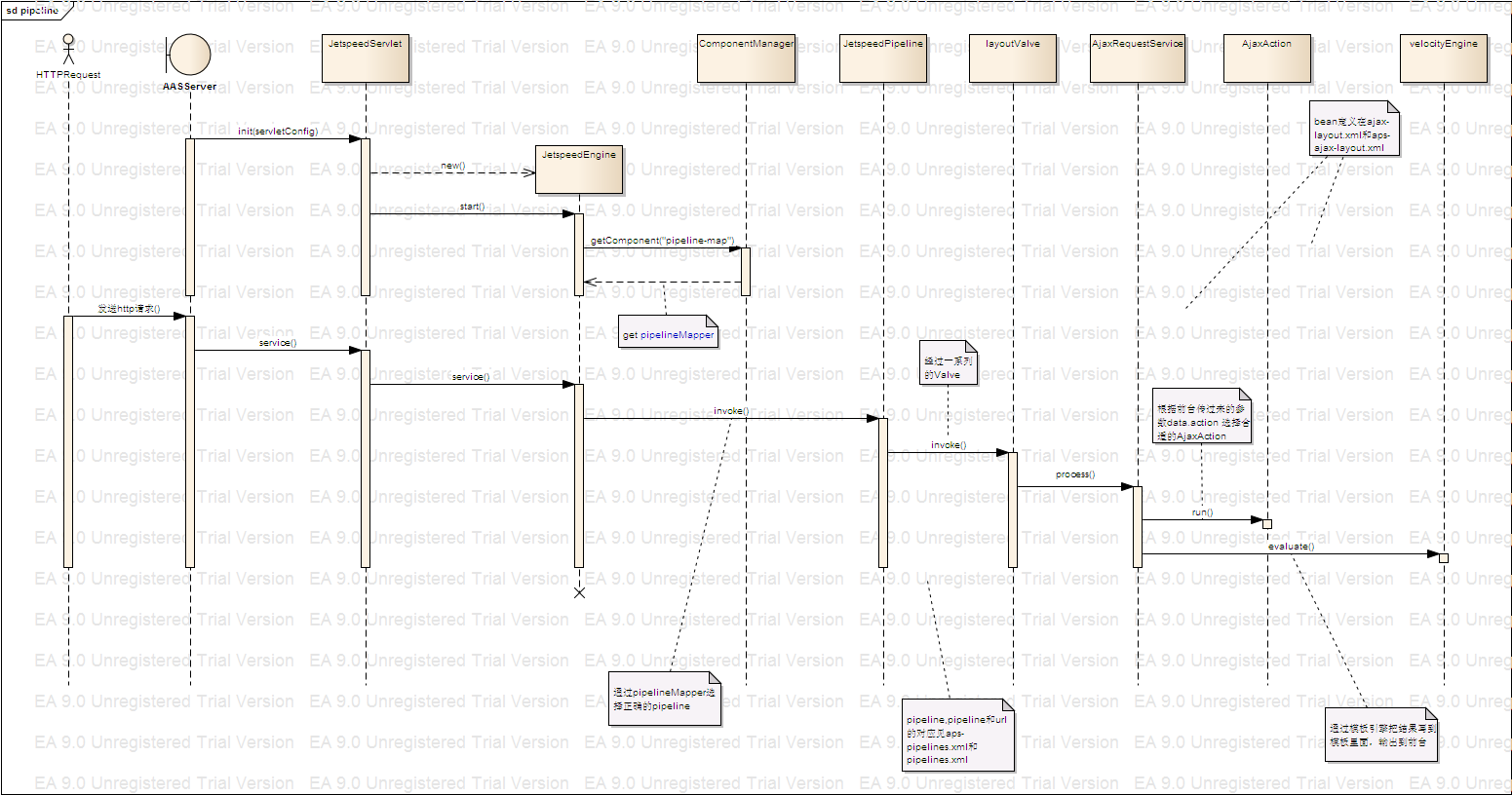


图3-22 访问ajax-pipeline的时序图

#### LayoutValve

LayoutValve的逻辑很简单，它仅仅是调用了AjaxRequestService的process()方法，调用完毕后将控制权交给下一个Valve。

#### AjaxRequestService

AjaxRequestService的实现类是AjaxRequestServiceImpl，它提供了处理Ajax请求的模板。核心的处理逻辑在process()方法，主要功能是：

* 根据前台传入的action参数的不同而调用不同的AjaxAction实现类的run(RequestContextrequestContext, Map resultMap)方法来处理请求,在run()方法里把存入resultMap
* 执行的结果，被存到resultMap中，这些结果将会被AjaxBuilder再次处理
* 处理的最终结果为Velocity引擎所用，填充到Velocity模板中作为结果返回给前台页面
* 根据format选择模板格式，模板格式Jetspeed默认支持XML格式，APS扩展了某些action的JSON格式，具体可参见前台Js API
* 调用velocity模板引擎把结果写入模板中，模板文件位于jetspeed-portal-2.2.2.jar/org/apache/jetspeed/layout/ajax-xml/，后缀为\*.vm

前面提到根据前台传入action参数来决定调用AjaxAction的实现类。这一功能的实现原理是：

AjaxRequestServiceImpl持有一个Map，它在Spring容器中配置，在AjaxRequestServiceImpl被实例化时被注入进来。它被配置在ajax-layout.xml中，如下图：



AjaxRequestServiceImpl在处理Ajax请求时，会获取前台传递的action参数，以它为Key获取值，例如add为Key，AjaxAddPortlet为值，从而找到配置id为AjaxAddPortlet的类AddPortletAction。

AjaxAction的实现类也同时实现了AjaxBuilder，所以它们既是AjaxAction也是AjaxBuilder。

#### AjaxAction

所有AjaxAction的实现类都实现了AjaxAction，AjaxBuilder接口，一部分实现类也实现了Constant。

我们看一下官网上对它的定义：



图3-24 AjaxAction

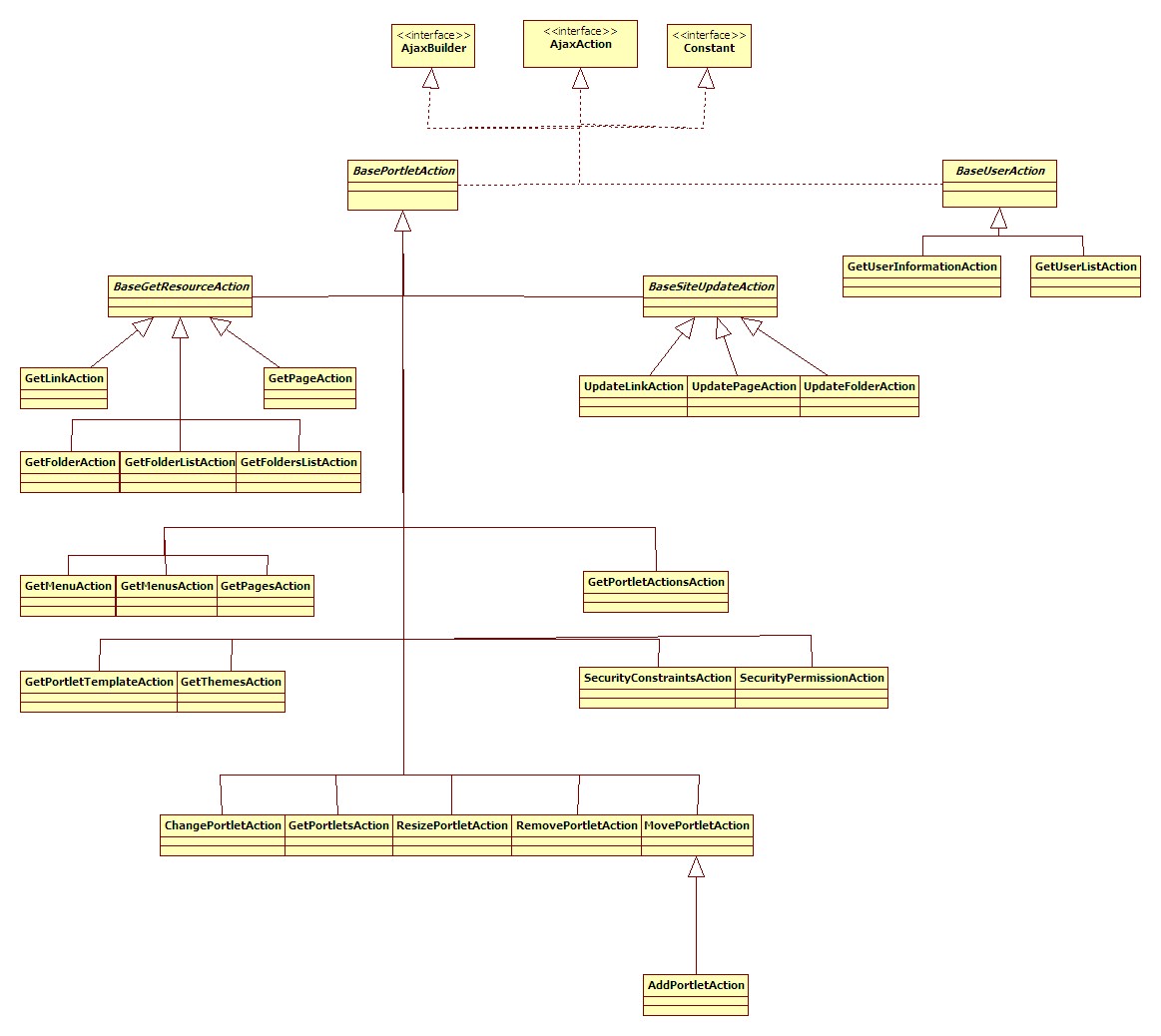
An Ajax request is made up of an action and builder phases.

Implement AjaxAction interface for the Ajax action phase.

The action should put any parameters or results it wants

passed on to the builders in the resultMap.

继承关系如下图：



上面的继承图，不需要刻意的记忆，我们下面对它进行了分块。

AjaxAction实现类分为以下几个功能模块：

* 目录相关

1. GetFolderAction
2. GetFolderListAction
3. GetFoldersListAction
4. UpdateFolderAction

* 页面相关

1. GetPageAction
2. GetPagesAction
3. UpdatePageAction

* 菜单相关

1. GetMenuAction
2. GetMenusAction

* 链接相关

1. GetLinkAction
2. UpdateLinkAction

* Portlet相关

1. GetPortletsAction
2. GetPortletActionsAction
3. ChangePortletAction
4. AddPortletAction
5. MovePortletAction
6. RemovePortletAction

* 主题相关

1. GetThemesAction

* 用户相关

1. GetUserInformationAction
2. GetUserListAction

* 安全相关

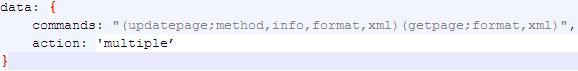
1. SecurityConstraintsAction
2. SecurityPermissionAction

* MultipleAction

这个接口比较特殊，他的作用是一次提交多个action让后台处理，可以减少http请求，但也没有保证action之间的事务，前端发起的请求格式如下图：

C:\Documents and Settings\smile\桌面\1.jpg

访问它所需要请求参数的代码如下图：



#### AjaxBuilder

它的作用是将AjaxAction处理的结果resultMap进一步处理，这里我们也给出官网上对它的定义：



Implement AjaxBuilder interface for the Ajax builder phase.

The builder can add additional information to the requiest context,

and it also provides the template used for building the result stream

sent back over the Ajax request.

作为AjaxBuilder，会被注入进vm模板文件，这个配置是在ajax-layout.xml中进行的，如下图：



与其他AjaxAction实现类一样，AddPortletAction也实现了AjaxBuilder，在AjaxRequestService中会调用AjaxBuilder的getTemplate()与getErrorTemplate()提供模板名，供Velocity加载并使用。

需要再次着重声明的是，这些模板极少部分使用到JSON，绝大多数都是用了XML格式，我们将此处进行扩展，从而令已经使用到的AjaxBuilder对应的Vm模板返回JSON格式的数据。

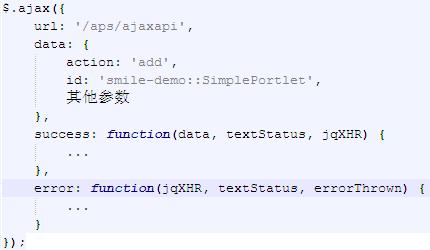
#### Constant

Constant没有提供任何方法，仅仅作为一个常量接口存在，提供了前后台传递所约定的参数。在上述的AjaxAction的实现类中，大部分都实现了它，但是由于Jetspeed的这个模块开发者的不同而导致有些实现类没有实现Constant。

#### ajax-pipeline的实现原理

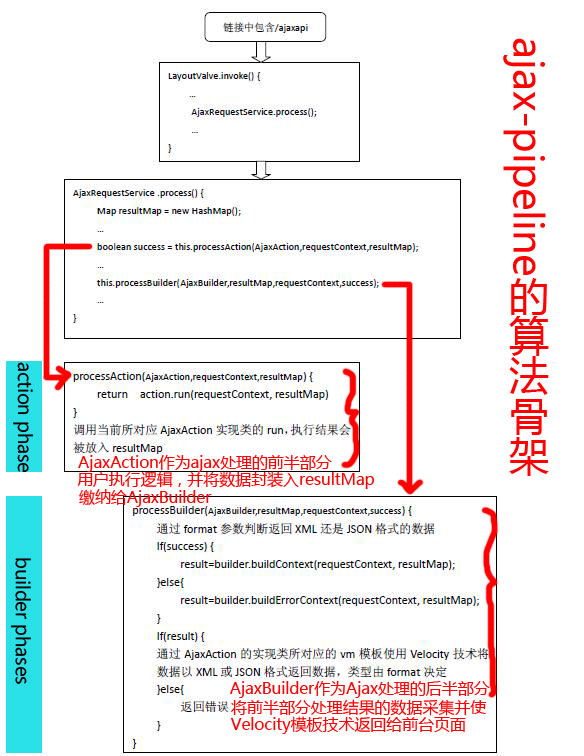
前面已经将ajax-pipeline各个核心接口讲解了一下，本节将简要介绍一下它的实现原理。所有AjaxAction的实现类的执行时序与流程都是大致类似的，我们拿AddPortletAction举例：

前台页面通过下图的代码发起Ajax请求：



图中url指向了/ajaxapi，从而会进入LayoutValve。名为“action”的参数，指向了AddPortletAction。名为“id”的参数，是AddPortletAction进行逻辑操作所需要的参数，这种为AjaxAction的实现类所需要的参数有些是必须的有些是可选的，得根据代码逻辑来判断。执行的结果是给当前页面添加了一个新的Portlet，并有一些结果以XML或者JSON格式反馈给前台页面供Js代码回调。

大致的执行逻辑用下图表示：



从上图可以分析出，Ajax的处理，分为两个阶段：

* action phase
* builder phase

#### APS对Ajax API的扩展

根据实际的需要，我们对Ajax API模块进行了一些扩展。我们将一些用到的AjaxAction返回数据的格式由XML修改为JSON。

除了覆盖了一些接口外，我们还添加了几个AjaxAction（这些覆盖与扩展都在aps-ajax-layout.xml中进行了对应的配置）：

* APSGetPortletsAction

返回当前用户可用的Portlet列表

* APSThemeOperatorAction

操作主题配置项

* APSGetPortletModelAction

获取新增portlet的模式和状态

* APSResizePortletAction

调整Portlet高度

* APSUpdatePortletAction

更新指定的Portlet

* APSGetAvailableSites

获得可用的站点

* APSSimpleAddPortletAction

添加Portlet

### ajax-direct-pipeline

目前没有用到，只做简单的介绍。

ajax-direct-pipeline独有的AjaxValve的功能是直接解析HTTP请求对后台的bean进行方法调用（通过反射）,需要传给后台的自然有对象名，方法名，参数。

访问它所需要请求参数的代码如下图：



参数说明：

* "portletRegistry.getPortletApplication"配置于ajax.xml中，是Map的Key，用于跟通过Spring配置的服务进行映射
* getPortletApplication是需要调用的方法名。
* ajax\_param\_0\_int:44 是参数位置，类型，和具体值，只支持String与int两种数据类型