1.概述

Parsley 是一个为 Flex 和 Flash 设计的 IOC 容器和消息框架，使用它可以创建高解耦的构架，它允许你用 Metadata, MXML, XML 或者 ActionScript 来配置 object,而这些 object 由一个容器来管理，而且 Parsley 是非常容易扩展的.

许多其它 Flash 平台的框架，有的是纯 Flex 框架而不能在没有 Flex SDK 时使用，有的就是不能与 Flex 深度集成的 Flash 框架，而 Parsley 两者的功能都具有. 本框架的核心 (IOC 容器与消息子系统) 并不依赖于 Flex SDK , 但有一些额外的模块是专为 Flex 设计的, 为 MXML 配置、视力注入和 Flex 模块提供了支持.

1.1 特性列表本部分提及了开发手册中其它章节的概要. 此外第 2 章 开始学习也包括了一些用例.

IOC 容器
Parsley 是一个优秀的 IOC 容器. 它提供了依赖注入、对象生命周期的管理和消息的支持. 它与其它框架最主要的不同是框架的作用域: 它像其它小型的 IOC 框架一样为简单程序提供了易用性,而且也为大型、复杂和模块化的程序提供了许多特性，如：支持 Flex 模块、对象清理和大量的扩展点.

依 赖注入
这是任何 IOC 容器的核心功能. 现在 Parsley 2 的依赖注入功能可以非常方便地使用 AS3 Metadata 标签 ([Inject]) （或者 MXML、XML） 来对属性、方法或者构造函数进行声明. 查看第 4 章 依赖注入了解详细.

消息
Parsley 包含一个消息框架允许对象以完全解耦的方式发送和接收消息. 你可以在发送对象上使用[ManagedEvents]标签来声明事件, 这样可以通过 Parsley 传递给注册该事件的处理器. 接收对象可以使用几种 metadata 标签如：[MessageHandler]来声明对特定消息类型的关注. 消息的传递是基于消息的类型（class）的，此外还可以增加消息的选择器如：事件的类型, 而不是像传统的事件处理器只基于纯字符串.这样你无需关心事件类型的常量在整个系统中是否唯一的问题. 查看第 5 章 消息了解细节.

高级 IOC 容器特性
6.5 内容生命周期: 对象使用[PostConstruct]标记的方法 会在对象实例化和配置后调用，而带有[PreDestroy]标记的方法会在容器销毁后调用. 对于模块化的程序，你需要创建动态的加载子内容，而当它不再被使用时则可以清除整个模块.

6.2 异步对象的初始化: 异步初始化对象时的可选配置 (例如：在使用对象前需要加载数据). 容器会在配置成异步的对象准备好后才初始化其它对象.

Flex 视图的注入
这是特别为 Flex 设计的整合模块并解决了你不想在 IOC 容器配置文件中声明 Flex 组件的问题, 就像平常一样在 MXML 视图中添加组件. Parsley 2 允许当组件添加到 Stage 后就交给 IOC 容器来处理. 查看第 7 章 Flex 组件的注入了解细节.

对 Flex 模块的支持
作为 Flex 的特别整合的模块，它可以关联到在特定内容需要的配置类、Flex 模块文件、services, actions, mediators 等等.当 Flex 模块被卸载时，与该模块相关的整个子系统、对象也会被销毁. 查看第 8 章 使用 Flex Modules 了解详细.

本地化
使用 [ResourceBinding]标签将属性与资源文件绑定. 对于 Flex 程序，整合到 Flex ResourceManager，对于 Flash 程序也有相关的本地化模块. 查看第 10 章 本地化了解详细.

扩展性

Parsley 可以作为创建更高级框架的基础. 你也可以根据你的需要为你的程序添加一些配置标签进行相应的开发. Parsley 是很容易进行扩展的. 一个接口的实现就可以用来创建自定义配置标签，如： Metadata, MXML 或者 XML 标签. 查看第 11 章 扩展本框架和第 6.1 章 使用工厂类了解细节.

1.2 Parsley 和 Spicelib 模块的 SWC 文件 Parsley 和底层的 Spicelib 是非常模块化的, 你只需选择你需要的 SWC 文件. 下载后的 release 文件夹包含了 Parsley 和 Spicelib 的 SWC 文件以及它们对应的 Flex 和 Flash 版本.

module 文件夹包含了实现不同子功能的 SWC 文件，你可以根据你需要添加到你的库路径中去. 使用子功能的 SWC 文件时，你需要注意每个 SWC 文件的依赖特性.

以下的图表展示了不同 SWC 之间的依赖特性. 名字省略了版本号和后缀名. 为了图表的清晰略去了几个可选的依赖文件

以下的表格总结了每个 SWC 的内容:

spicelib-core	核 心 utilities, errors 和 events.
spicelib-reflect	Spicelib Reflection API. 查看 第16章 Reflection 了解细节.
spicelib-xml	XML-to-Object 映射器. 查看 第17章 XML 到 Object 的映射器了解细节.
spicelib-task	异 步操作的 Task 框架. 查看 第18章 Task 框架了解细节.
spicelib-flash	Flash 日志框架. 查看 第19章 Flash 程序的日志了解细节.
parsley-core	核 心 IOC 容器 - 依赖注入,消息, Metadata 或者 ActionScript 配置.
parsley-flex	Flex 整合特性:第3.2章 MXML 配置,第7章 Flex 组件注入,第8章 使用 Flex 模块.
parsley-xml	提 供 XML 文件进行配置的支持 . 查看 第3.3章 XML 配置文件了解细节.
parsley-flash	Flash 整合特性: 第10.3章 本地化 Flash 程序,第13.2章 Flash 的日志配置.
parsley-pimento	为 Pimento 和 Cinnamon 配置标签. 查看 第12.2章 Pimento Data Services 和 第12.3章 Cinnamon Remoting.

2.入门

本章简单介绍如何使用 Parsley 创建程序,你可以查看其它章节了解详细。大多数的描述都适用于开发 Flex 或者 Flash 程序. 只适用于 Flex 开发的例子会作特别说明.

2.1 Hello world 应用程序举例

如果你想一边读手册，一边看例子，你可以查看 Hello World 例子。

本章不涉及例子应用程序，以下可能会在更多的例子上加入变化的功能。

2.2 添加框架的 SWC 文件

使用 Parsley 之前你需要将它的 SWC 文件添加到你的库路径中.你可以根据你的 Flex 或者 Flash 程序的需要添加相应的 SWC 文件.
最快的方式是将包含整个框架的 SWC 添加到库路径. 你可以从下载的 Zip 文件的 release 文件夹找到.
Flex 程序需要的 SWC

- parsley-complete-flex-${version}.swc
- spicelib-complete-flex-${version}.swc

Flash 程序需要的 SWC

- parsley-complete-flash-${version}.swc
- spicelib-complete-flash-${version}.swc

选择单独的框架 SWC

如果你只想选择你正在使用的框架模块，你可以选择 release 文件夹下的 modules 文 件夹中的相应的 SWC 文件 . 你需要注意每个 SWC 文件的依赖特性，在 第 1.2章 Parsley 和 Spicelib Module SWC 会有详细说明.

2.3 定义对象的依赖

这是 Parsley 最常用的功能. 在 Parsley 2 中我们可以非常方便地使用 Metadata 或者 MXML 或者 XML 文件来定义依赖.

使用 [Inject] metadata 标签

你可以在属性上使用标签:

```
[Inject]
 public var loginService:LoginService;

[Inject]
 public function set loginService (service:LoginService) : void {
```

或者你可以在方法上一次性声明多个依赖:

```
[Inject]
 public function init (loginService:LoginService, cartService:CartService = null) : void {
```

以上例子的好处是，Parsley 会根据参数类型进行反射并对可选的参数进行可选地依赖注入. 在本例子中如果容器中没有 LoginService 的话会抛出一个错误，但没有 CartService 的 话则不会.

最后，你可以在类定义上使用相似的标签([InjectConstructor]) 来进行构造器注入 (因为 Flex 编译器会忽略构造器上 metadata 标签，所以要把它放在类定义上):

```
[InjectConstructor]
 public class LoginAction {

 function LoginAction (loginService:LoginService, cartService:CartService = null) : void {
```

最佳实践

在本章展示的所有的例子中，依赖会根据类型进行注入，因为 Parsley 会根据属性 和方法参数的类型进行反射. 要想从一个 IOC 容器中得到解耦的好处，你应该偏向于在注入点处使用接口类型进行声明. 这样，你可以在修改不同的实现时不用更改注入点.

当然这只有在你往容器中只添加一个匹配类型的对象时有效. 如果你的同一个接口有多个实现，你需要使用 id（只有在属性上可用）进行注入:

```
[Inject(id="mainLoginService")]
 public var loginService:LoginService;
```

你可以根据自己的需要选择属性、方法或者构造器注入. 有人会偏好于构造器注入，因为它有更好的封装性，允许你创建持久类. 但 Flex 中的构造器注入也有一些局限性，特别是如果你使用 MXML 配置来装配对象，查看 第 四章 4 依赖注入 了解细节.

2.4 发送和接收消息

你在注入点使用接口类型的依赖注入的方式提供了一定的解耦，如果你想用更松的解耦，你可以使用消息进行通信，而发送者和接收者无需知道对方.

如果你使用 MVC 架构来创建你的应用程序，你可能**创建一个 Mediator 类来将底层的视图事件转换成消息，而该消息会通过程序中的消息子系统发送出去，然后所有动作、控制器可以注册所有它们关注的消息类型**.

查看 5 Messaging 和 9 Building MVC Architectures.了解细节.

使用 Parsley 2 你可以使用 metadat 标签来配置发送方和接收方. **发送方**可能只需在类声明处使用一个[ManagedEvents]标 签列出这个类发送出的哪些事件需要被托管:

```
[Event(name="loginSuccess",type="com.bookstore.events.LoginEvent")]
[Event(name="loginFailed",type="com.bookstore.events.LoginEvent")]
[Event(name="stateChange",type="flash.events.Event")]
[ManagedEvents("loginSuccess,loginFailure")]

public class LoginServiceImpl extends EventDispatcher implements LoginService {

    [...]

      private function handleLoginResult (user:User) : void {
          dispatchEvent(new LoginEvent("loginSuccess", user));
    }

}
```

这里我们告诉容器一旦这个类发送一个 loginSuccess 类型或者 loginFailed 类型的事件，容器都会把它发送到所有注册该事件的消息处理器. 而其它的事件将会被忽略, 这个类依然可以发送底层的事件到注册了对应事件的直接依赖于该类的对象.

对于接收方，你可以使用[MessageHandler] 标签来通知关注某个类型的消息:

```
[MessageHandler]
 public function login (event:LoginEvent) : void {
```

如果你在同一个事件类中有几个不同的消息，你可以用事件类型进行区分:

```
[MessageHandler(selector="loginSuccess)]
 public function login (event:LoginEvent) : void {
```

依赖是根据类型注入的，这确保了系统的安全性，你也无需处理整个系统中事件类型的常量唯一性问题.关于接收方还有 [MessageBinding] 或者 [MessageInterceptor]等标签. 这会在 5 Messaging 进 行介绍.

另外，因为这个模块叫做消息系统而不是事件系统，你可以使用任何类作为一个消息，不单单是那些继承 flash.events.Event 的类. 查看 5.4 Injected MessageDispatchers.

2.5 装配对象

依赖注入和消息系统还有许多的配置标签. 但以上的是最常用的。现在我们要教你如何为 IOC 容器装配对象. 在你在类上面添加 metadata 标签后，你需要告诉容器：它需要管理哪些对象.

Flex Applications

在 Flex 中你可能偏好于使用 MXML 配置. 你可以创建一个简单的 MXML 类 (使用 mx:Object 作为根标签) 并添加你要 Parsley 进行管理的类:

```
<mx:Object
    xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:services="com.bookstore.services.*"
    xmlns:actions="com.bookstore.actions.*">

    <services:LoginServiceImpl timeout="3000"/>
    <services:CartServiceImpl timeout="3000"/>
```

```
        <actions:LoginAction/>
        <actions:AddToCartAction/>
        <actions:DeleteCartAction/>

  </mx:Object>
```

当然你也可以使用这个配置文件来设置其它属性值. 容器会处理所有类中的 metadata 标签的同时也会对它们进行处理.

本例子中我们没有指定任何 id 属性. 如果你是使用根据类型注入的你就不必需要 id. 如果你在注入点使用 id 注入的话，你可以像下面那样指定:

```
          <services:AdminLoginServiceImpl id="adminLogin" timeout="3000"/>
            <services:UserLoginServiceImpl id="userService" timeout="3000"/>
```

然后你可以在注入点使用那些 id:

```
          [Inject(id="adminLogin")]
            public var adminLogin:LoginService;

            [Inject(id="userLogin")]
            public var userLogin:LoginService;
```

但一般情况下你应该尽量不要用 id 注入，只有在同一个类型有多个实例的情况下才用.

Flash Applications

在这种情况下 MXML 并不可用. 所以最好使用 XML 文件进行配置.如果你不想把配置编译到你的程序中去的话你也可以使用外部 XML 文件

与上面的 MXML 例子对应的 XML 配置如下所示:

```
<objects
     xmlns="http://www.spicefactory.org/parsley"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.spicefactory.org/parsley
         http://www.spicefactory.org/parsley/schema/2.0/parsley-core.xsd"
    >

     <object type="com.bookstore.services.LoginServiceImpl">
         <property name="timeout" value="3000"/>
    </object>

     <object type="com.bookstore.services.CartServiceImpl">
         <property name="timeout" value="3000"/>
    </object>

     <object type="com.bookstore.actions.LoginAction"/>

    <object type="com.bookstore.actions.AddToCartAction"/>

    <object type="com.bookstore.actions.DeleteCartAction"/>
</objects>
```

像 MXML 例子那样，我们无需指定对象的 id.如果需要的话，你也可以指定 .

以上就是装配对象最常用的选项. 你可以在 3 Configuration and Initialization 找到更多例子.

2.6 初始化框架

现在你可以使用 metadata 标签配置你的类，然后将它们装配到 MXML 或者 XML 文件, 是时候初始化所有东西了，大多数情况下只需使用一行命令.

设我们使用一个叫 BeanConfig.mxml 进行配置, 那一行命令会如下所示:

```
FlexContextBuilder.build(BeanConfig);
```

就是这么简单，你应该尽早执行该命令, 一般是在主程序的 preinitialize 事件执行.

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    preinitialize="FlexContextBuilder.build(BeanConfig)"
    addedToStage="dispatchEvent(new Event('configureIOC', true));"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600">
```

注：

如果你想将 MXML 定义的视图注入到容器的话，这是非常重要的。正如 第 7章 Flex 组件的注入所描述的那样.

使用 XML 文件的话则使用以下命令:

```
XmlContextBuilder.build("config.xml");
```

我们会在此讨论所常用的用例. 在同一个程序中我们可能使用多个文件进行配置，也可能同时使用 MXML 和 XML 进行配置. 这些内容会在 第 3.5章 整合多个配置文件讨论.

对于模块化的程序你不会把所有的东西都整合到一块然后在程序开始时就把它们全部加载进来.对于创建模块化程序和根据需要进行加载、卸载配置的内容，请查看 第 6.4章 Modular Contexts 和 第 8章 使用 Flex Modules.

2.7 增加更多的 Services

由于这是刚开始学习阶段，我们只提及常用的用例. 希望可以使你开始编写简单的程序. 如果你需要更多知识，你应该阅读跟着的章节.

如果你想了解 Parsley 提供的功能，你最好从 第 1.1章 特性列表开始.

最有趣而本章没有提到的部分可能是:

- 3.4 ActionScript 配置: 你并不局限于使用 Metadata, MXML 或者 XML 进行配置,第四个选择是 ActionScript, 它给予你另外的扩展性.
- 5.7 MessageInterceptors（消 息拦截器）: 该章讨论了 MessageHandlers 另外的选择，使你可以干涉消息发送的处理过程 (例如：暂挂、恢复和取消消息的处理).
- 6.1 使 用 Factories 类: 你可以往 IOC 容器中添加创建对象的工厂类来取代装配对象, 这为你如何将对象实例化提供了伸展性
- 6.2 异 步对象的初始化: 异步地初始化对象的配置选项(例如：在对象可以被操作时先加载其它数据). 这种情况下，容器会推迟其它对象的初始化而先初始化那些被设置成异步的对象.
- 7 Flex 组件的注入: 我们已经简单地提了一下这方面. 这令你可以将在 MXML 中定义的组件注入到 IOC 容器中声明的对象中去 r.
- 8 使 用 Flex Modules: 关联配置的 services, actions, mediators。使用 Flex Module 可以在自动地加载和清除单独的模块.
- 10.1 ResourceBindings（资 源文件的绑定）:将被管理对象的属性与资源文件绑定, 当资源文件发生改变时自动更新相应的属性.
- 11.2 创 建自定义的配置标签: 你可以添加你自己的 Metadata, MXML 或者 XML 配置标签. 在简化配置任务或者在 Parsley 基础上创建你自己的框架时非常有用.

3.配置与初始化

配置和初始化 Parsley 框架通常包括以下步骤:

- 第 1 步: 告诉 IOC 容器哪些类需要被它管理. 这可以使用 MXML, XML 文件或者 ActionScript 来做.这三种机制会在跟着下来的章节中说明.
- 第 2 步:配置容器的服务，如：依赖注入或者每个单独类的消息 .这可以在第 1 步中你选择的配置机制中完成(例如：使用 MXML 或者 XML 配置标签) 或者 - 通常情况下在类中使用 AS3 Metadata 标签.
- 第 3 步: 初始化 IOC 容器(通常在程序开始时). 在 Parsley 2 中通常只需一行代码. 可以查看下面例子.

3.1 使用 AS3 Metadata 进行配置

AS3 Metadata 标签可以用来配置像依赖注入或者消息等服务. 它们可以放到由 Parsley 管理的任何类中去.使用 metadata 标签的配置可以与 XML 或者 MXML 的配置相结合. 几乎每个 metadata 标签每有与它相对应的 MXML 和 XML 配置标签. 为了避免冲突， MXML 和 XML 配置会优于 Metadata 配置, 因此你可以覆盖 metadata 配置而不必对类的代码进行修改.
每个 metadata 标签和它们的属性会在相应的章节的相关例子中说明:

- 4 依 赖注入
- 5 消 息系统
- 6.1 使 用 Factories 类
- 6.2 异 步对象的初始化
- 6.5 Context 生命周期

3.2 MXML 配置

这种配置只能被 Flex 程序使用. 其它的配置在 Flex 和 Flash 程序中都可用.
假定你要配置以下两个类:

```
package com.bookstore.service {
            class LoginServiceImpl implements LoginService {
                  public var timeout:int;
                  public function login (username:String, password:String) : void {
                          // execute service
                    }
            }
    }

package com.bookstore.actions {
           class LoginAction {
               [Inject]
               public var service:LoginService

                 [MessageHandler]
               public function handleLoginEvent (event:LoginEvent) : void {
                     service.login(event.username, event.password);
                 }
            }
    }
```

你可以见到使用 metadata 已经配置了几个功能. 查看 第 4章 依赖注入 和 第 5章 消息系统 了解这些功能的

细节.

MXML 配置文件

我们需要告诉容器管理这些类并创建以下 MXML file:

```
<mx:Object
         xmlns:mx="http://www.adobe.com/2006/mxml"
        xmlns:services="com.bookstore.services.*"
        xmlns:actions="com.bookstore.actions.*">

         <services:LoginServiceImpl timeout="3000"/>

         <actions:LoginAction/>

     </mx:Object>
```

请注意我们并不需要导入 Parsley . 只需要在 MXML 中添加普通的 object 标签. 除了 metadata 标签你还可以使用 MXML 标签进行其它配置 , 比如上面例子的 timeout 值.

框架的初始化

最后你需要初始化容器. 假设你将配置保存在 BookstoreConfig.mxml, 你可以使用以下命令将它初始化:

```
FlexContextBuilder.build(BookStoreConfig);
```

在许多程序中，以上的一行代码便是你唯一直接使用的 Parsley API.你可以在某些类的方法上添加 [PostConstruct] 标签让程序开始时执行这些方法. 查看 第 6.5章 Context 周期 了解细节.

理论上你也可以使用 Parsley 的 API:

```
var context:Context = FlexContextBuilder.build(BookStoreConfig);
    var initializer:BookStoreInitializer = context.getObjectByType(BookStoreInitializer) as BookStoreInitializer;
    initializer.execute();
```

但这种用法并不推荐使用. 在一般的程序中都无需直接使用 Parsley 的 API. 只有在 Parsley 基础创建你自己的框架时才使用.

使用 Parsleys MXML 标签

以上的 MXML 配置还可以使用 Parsley 标签来代替 object 标签:

```
<mx:Object
     xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns="http://www.spicefactory.org/parsley">

     <mx:Script>
        <![CDATA[
            import com.bookstore.services.*;
             import com.bookstore.actions.*;
         ]]>
    </mx:Script>

     <Object type="{LoginServiceImpl}">
        <Property name="timeout" value="3000"/>
    </Object>
```

```
        <Object type="{LoginAction}"/>

  </mx:Object>
```

使用这些特殊的标签比普通的标签有其它的功能. 两种方式都有优缺点:

普通 MXML 标签的优点:

- 简单、可直接使用.
- 无需知道 Parsley 的配置标签.
- 编译器会检查属性值的类型.

Parsley MXML 标签的优点:

- 允许使用构造器注入 . 从封装性方面来看，这是依赖注入的最干净的方式. 使用普通标签时你不能使用构造器注入的方式,因为在那种情况下 MXML 编译器生成了对象的创建代码并且 Parsley 只能在对象实例化后才得到该对象. 当使用普通标签时，你的类需要一个没有参数的构造方法.
- 允许定义对象为 lazy 的 (<Object lazy="true" type="..."/>) ，也就是说在你第一次使用该对象之前它不会被初始化和配置.
- 允许将一个对象配置成非单例的(<Object singleton="false" type="..."/>). 也就是说，每次从容器取这个对象或者将这个对象注入其它对象时，容器都会创建一个新的实例.
- 允许使用自定义标签. Parsley 2 是非常容易扩展的. 你可以使用一个类来创建一个自定义的扩展，然后将它作为自定义的 metadata 标签, MXML 或者 XML. 你可以查看 第11 章 扩展本框架中的例子.

3.3 XML 配置文件

在以下情况中，外部 XML 文件是 MXML 之外的另一个合适的选择:

- 你的配置经常发生改变而你又不想再进行编译.
- 配置文件可以被不懂编程的人进行修改.
- 你没有在使用 Flex.

当然你也可以在某部分配置中使用 XML 文件而仍然使用 MXML 配置来注入你的核心服务. 查看 第 3.5章 将多个配置机制整合 了解细节.

上面例子的 MXML 配置也可以配置成以下的 XML 配置:

```
<objects
    xmlns="http://www.spicefactory.org/parsley"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.spicefactory.org/parsley
         http://www.spicefactory.org/parsley/schema/2.0/parsley-core.xsd"
   >
   <object type="com.bookstore.services.LoginServiceImpl">
        <property name="timeout" value="3000"/>
   </object>

    <object type="com.bookstore.services.LoginServiceImpl"/>
</objects>
```

一般的 XML 配置跟使用 Parsley MXML 标签的 MXML 配置非常类似. 在底层它们实现的作用是一样的.唯一的不同是 XML 配置需要使用 XML 规范，标签不可以大写并且属性名需要使用-，如： targetProperty 要写成 target-property .

假定你的配置文件是 config.xml，初始化还是使用一行代码:

```
XmlContextBuilder.build("config.xml");
```

编译在 XML 中配置的类

你需要注意的是：与在 MXML 中配置的类不同，XML 中配置的某些类如果没有被使用的话，它不会被编译到你的 SWC 或者 SWF 文件.

解决这个问题有三种方法（即：你想要把没有使用的类也要编译到 SWC 或者 SWF 文件）：

- 在你的代码中添加这些类的引用，即使你并不需要它.
- 如果你想将这些类作为库使用，将它们编译到一个 SWC (使用 compc 你可以将整个源代码文件夹包含到 SWC 文件) 然后使用 mxmlc 编译器中的-include-libraries 选 项将整个 SWC 包含到你的 SWF 文件.
- 你可以使用 mxmlc 编译器的-includes 选 项来包含某些类.

3.4 ActionScript 配置

如果你只认识 Flex IOC 容器的话，你可能不熟悉这种方式. 它跟 Java 的 Spring 框架中的 JavaConfig 有点相似. 它允许你使用代码的方式来创建要给 Parsley 管理的对象. 我们使用之前例子的那两个类并将它们添加到 IOC 容器中:

```
package com.bookstore.config {

  class BookStoreConfig {

      public const action:LoginAction = new LoginAction();

        public function get service () : LoginServiceImpl {
            var service:LoginServiceImpl = new LoginServiceImpl();
            service.timeout = 3000;
            return service;
        }

  }
}
```

我们使用了 ActionScript 来设置 timeout 属性. 无论你使用 var, const 或者一个 getter 函数来声明对象，这些对象都会被添加到 IOC 容器中.

初始化还是只用一行代码:

```
ActionScriptContextBuilder.build(BookStoreConfig);
```

这个配置模式允许你增加 metadata 标签:

```
[ObjectDefinition(singleton="false")]
public function get service () : LoginServiceImpl {
    var service:LoginServiceImpl = new LoginServiceImpl();
    service.timeout = 3000;
    return service;
 }
```

在上面的例子中，每次这个对象被请求时，容器都会调用 getter 函数. 默认的 singleton 属性是 true, 所以如果没有 metadata 标签的话 Parsley 只调用这个方法一次并把这个实例缓存起来，对于所有的注入都是重复使用同一个实例.

3.5 将多个配置机制整合

自 Version2.2后，配置管理机制允许指定实例在运行期间，作为容器的一部分：

```
<parsley:ContextBuilder>
            <parsley:FlexConfig type="{ServiceConfig}"/>
            <parsley:FlexConfig type="{ControllerConfig}"/>
            <parsley:XmlConfig file="logging.xml"/>
            <parsley:RuntimeConfig instances="{[instance1, instance2]}"/>
     </parsley:ContextBuilder>
```

如果你需要指定 ID，也可标注子标签：

```
<parsley:ContextBuilder>
     <parsley:FlexConfig type="{ServiceConfig}"/>
     <parsley:FlexConfig type="{ControllerConfig}"/>
     <parsley:XmlConfig file="logging.xml"/>
     <parsley:RuntimeConfig>
          <parsley:Instance id="obj1" instance="{instance1}"/>
           <parsley:Instance id="obj2" instance="{instance2}"/>
           <parsley:Instance id="obj3" instance="{instance3}"/>
      </parsley:RuntimeConfig>
</parsley:ContextBuilder>
```

在 RuntimeConfig 标签中添加的对象与之后添加的 DynamicObjects 不同之处在于，前面的作为根对象，可注入到其它对象中，因为它们 在上下文构建时已指定。

你甚至可以使用普通的<Object>为内部标签

```
<parsley:ContextBuilder>
     <parsley:FlexConfig type="{ServiceConfig}"/>
     <parsley:FlexConfig type="{ControllerConfig}"/>
     <parsley:XmlConfig file="logging.xml"/>
     <parsley:RuntimeConfig>
          <parsley:Instance id="obj1" instance="{instance1}"/>
           <parsley:Instance id="obj2" instance="{instance2}"/>
           <parsley:Object id="obj3" type="{LoginInterceptor}"/>
                <parsley:MessageInterceptor method="intercept" type="{LoginEvent}"/>
          </parsley:Object>
      </parsley:RuntimeConfig>
</parsley:ContextBuilder>
```

你也可以用编码方式进行配置：

```
var rcp:RuntimeConfigurationProcessor = new RuntimeConfigurationProcessor();
rcp.addInstance(instance1, "id1");
rcp.addInstance(instance2, "id2");
rcp.addClass(MyClass, "id3");

var builder:CompositeContextBuilder = new DefaultCompositeContextBuilder(viewRoot);
```

```
        FlexContextBuilder.merge(MainConfig, builder);
        builder.addProcessor(rcp);
        builder.build();
```

3.6 混合多个配置方式

虽然你可能在大多数程序中偏好于只用一个配置, 你不必一定要这样做. 你可以将本章所讲的配置进行组合使用. 对于一种配置，你也可以将它分成几个文件. 然后使用 buildAll 方法:

```
FlexContextBuilder.buildAll([BookStoreServices, BookStoreActions]);

XmlContextBuilder.buildAll(["services.xml", "actions.xml"]);

ActionScriptContextBuilder.buildAll([BookStoreServices, BookStoreActions]);
```

你也可以使用 CompositeContextBuilder 类将它们合并在一起:

```
var builder:CompositeContextBuilder = new CompositeContextBuilder();
    FlexContextBuilder.merge(BookStoreConfig, builder);
    XmlContextBuilder.merge("logging.xml", builder);
    builder.build();
```

这种方式也是很简单的: 你只需创建一个 CompisiteContextBuilder 类的实例并把它放到各个 context builder 类的 merge 方法中.

以上所有的例子最终的结果都是一个 Parsley Context. 无论你有多少个配置文件，结果都是一样的.

对于大型和复杂的程序，你想要创建模块化 Contexts, 即多个配置不是合并到一个 **Context**, 以便它们可以根据需要进行加载和卸载. 对于模块化的程序你可以阅读 第 8章 使用 Flex Modules 和 第 6.4章 Modular Contexts.

最后，如果你想要创建你自己的配置模式并将它与现有的配置方式无缝地结合的话，你可以创建 ObjectDefinitionBuilder 接口的实例并将它们传入到 CompositeContextBuilder.addBuilder. 查看 第 11章 扩展本框架 了解细节 .

4.依赖注入

定义你的类的依赖是配置你程序的服务与动作的其中一个核心任务 . 本章会介绍 Parsley 提供给你进行依赖注入的各种选项.
依赖注入的一般方式是使用 AS3 Metadata 标签方式. 因为一个类的依赖是该类的定义的一个重要方面，所以在 ActionScript 类本身进行依赖的定义最好不过. 在某些情况下你可能偏好于使用外部的依赖声明, 相关的选项会在本章的最后部分 4.5 在 MXML 或者 XML 中声明依赖进行讨论.

4.1 构造器方式的注入

许多人会认为这是从封装方面来说最干净的注入, 因为它让你创建了不能改变的类. 由于 Flash Player 目前会忽略放在构造方法上的 metadata 标签，你需要将 [InjectConstructor] 标签放到类的声明上面来告诉 Parsley 产生构造器方式的注入:

```
package com.bookstore.actions {

[InjectConstructor]
class LoginAction {

    private var service:LoginService;
    private var manager:UserManager;

    function LoginAction (service:LoginService, manager:UserManager = null) {
        this.service = service;
        this.manager = manager;
     }

}
}
```

需要注意以上的例子中的 manager 参数是可选的. Parsley 会根据这个信息进行反射并决定依赖是必需的还是可选的. 所以在本例子中如果容器中没有 LoginService 的话会抛出一个错误, 但没有 CartService 的话则不会..
构造器注入是基于参数类型进行注入. 这意味着容器中对于某个类型最多会包含一个对象. 你最好习惯于使用接口作为参数的类型，这样在改变接口的实现时就无需修改类的代码.
正如 第 3.2章 MXML 配置 所讲的，如果你想用构造器注入的话你不可以使用简单 MXML 标签来配置，因为该情况下 MXML 编译器会生成对象的创建代码并且 Parsley 只会在对象实例化后才能得到它. 所以不应该像下面那样定义一个类:

```
<mx:Object
     xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:actions="com.bookstore.actions.*">

     <actions:LoginAction/>

 </mx:Object>
```

你应该像下面那样来定义:

```
<mx:Object
     xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns="http://www.spicefactory.org/parsley">

     <mx:Script>
        <![CDATA[
            import com.bookstore.actions.*;
         ]]>
    </mx:Script>

   <Object type="{LoginAction}"/>

 </mx:Object>
```

在使用 Parsley 的 Object 标签时，对象是由容器来初始化的，这样构造器注入才可以实现.
使用 XML 配置时没有什么限制.
但 Flash Player 有个讨厌的 bug，就是在构造函数参数类型进行反射时使用的 describeType 并不能正确地工作 (这种情况下 Flash Player 总是把类型认为 '*'). 解决这个 bug 的一个方法是在 Parsley 初始化前创建这些类的实例:

```
new LoginAction();
new ShoppingCartAction();
```

You can simply throw away these instances, just creating an instance "fixes" describe type for the parameter types of that class.

4.2 方法注入

方法注入与构造器注入相似. 你可以将 [Inject] metadata 标签放到任何方法上:

```
package com.bookstore.actions {

class LoginAction {


    private var service:LoginService;
    private var manager:UserManager;

    [Inject]
    public function init (service:LoginService, manager:UserManager = null) : void {
        this.service = service;
        this.manager = manager;
     }

}
}
```

与构造器注入一样，Parsley 会知道方法的参数是可选的还是必需的并相应地将注入设定成可选的或必需的. 注入的对象会根据类型来选择,所以你需要确保你的配置中每种类型最多有一个对象. 方法注入的 MXML 配置并没有限制, 相比于构造器注入你只需使用简单 MXML 标签来往容器添加对象.

4.3 根据类型的属性注入

这种注入与方法注入相似, 用于属性的注入:

```
package com.bookstore.actions {

class LoginAction {


    private var manager:UserManager;

     [Inject]
    public var service:LoginService;

     [Inject(required="false")]
    public function set manager (manager:UserManager) : void {
        this.manager = manager;
    }

}
}
```

你可以将 [Inject] 标签放到一个 var 声明或者一个 setter 方法上. 对于属性， Parsley 不能检测出该注入是必需的还是可选的，所以你要显性地设置它的 required 属性. 如果忽略该属性的话默认值是 true.
像构造器注入和方法注入一样，这种注入也是根据类型注入的. 所以你需要确保你的配置中每种类型最多有一个对象.

4.4 根据 ID 的属性注入

如果你不想容器根据类型来注入，你可以显性地设置需要注入对象的 id 属性:

```
[Inject(id="defaultLoginService")]
public var service:LoginService;
```

这种情况下 Parsley 会根据 id 来注入，所以配置文件也需要包含一个对应的 id:

```
<mx:Object
      xmlns:mx="http://www.adobe.com/2006/mxml"
     xmlns:actions="com.bookstore.services.*">

      <services:LoginServiceImpl id="defaultLoginService"/>

 </mx:Object>
```

一般不推荐这种使用 id 的注入.最好使用根据类型来注入. 如果你需要显性地设置依赖的 id 的话，最好在像下一节那样的 MXML 或者 XML 中进行配置.

4.5 在 MXML 或者 XML 中声明依赖

最后，你可以在 MXML 或者 XML 中声明依赖.

MXML 例子:

```
<mx:Object
      xmlns:mx="http://www.adobe.com/2006/mxml"
     xmlns="http://www.spicefactory.org/parsley">

      <mx:Script>
         <![CDATA[
             import com.bookstore.actions.*;
              import com.bookstore.services.*;
          ]]>
     </mx:Script>

     <Object id="loginService" type="{LoginServiceImpl}">
         <Property name="timeout" value="3000"/>
     </Object>

     <Object id="userManager" type="{UserManager}"/>

      <Object type="{LoginAction}">
         <ConstructorArgs>
             <ObjectRef idRef="userManager"/>
         </ConstructorArgs>
         <Property name="service" idRef="loginService"/>
     </Object>

 </mx:Object>
```

XML 例子:
```
<objects
    xmlns="http://www.spicefactory.org/parsley">

     <object id="loginService" type="com.bookstore.services.LoginServiceImpl">
         <property name="timeout" value="3000"/>
    </object>

    <object id="userManager" type="com.bookstore.services.UserManager"/>

     <object type="com.bookstore.actions.LoginAction">
         <constructor-args>
             <object-ref id-ref="userManager"/>
         </constructor-args>
         <property name="service" id-ref="loginService"/>
    </object>

 </objects>
```

你可以见到 MXML 和 XML 配置几乎是一样的，细小的不同是标记的不同(capitalized camel-case vs. lower case names with dashes). 你可以为构造器的参数或者属性设置依赖. 对于构造器的参数你可以将它放到简单属性的标签里面:
```
<constructor-args>
    <object-ref id-ref="userManager"/>
    <string>http://www.bookstore.com/services/</string>
    <uint>3000</uint>
</constructor-args>
```

如果对象的属性是依赖于另一个对象的，应使用 id-ref 属性指向那个对象而不是使用 value 属性.
在对象内容声明依赖
如果一个依赖只是被一对象使用的话，你可以在这个对象内部对它进行声明:
```
<objects
    xmlns="http://www.spicefactory.org/parsley">

     <object type="com.bookstore.actions.LoginAction">
         <constructor-args>
             <object type="com.bookstore.services.UserManager"/>
         </constructor-args>
         <property name="service">
             <object type="com.bookstore.services.LoginServiceImpl">
                 <property name="timeout" value="3000"/>
             </object>
         </property>
    </object>

 </objects>
```

内部对象的声明不可以设置其 id. MXML 的例子也是类似的.

5.消息

parsley2引入了一个新的通用消息框架，允许你以完全解耦的方式在对象间 传递消息。解耦意味着发送者与接收者可以相互不知道，同样重要的是，发送和 接收对象也完全从框架本身脱钩。这个优势被其它 FLEX 框架所忽略(包括 parseley1)，因此你不得不使用框架的对 象或静态方法去发送程序的事件或 消息。如果你的对象与框架完全脱钩，你可以在不同的上下文中复用他们，那些上下文中可能你想使用不同的框架或根本就不想使用框架。比如你可能想在单元测试 中注入派发对象和接收一个实例，而不是进行额外的初始化。

Parsley 消息框架很通用，它没有规定一个特殊的使用情况。这与一些现有 的 FLEX MVC 框架不同,这些框架主张使用一定的结构和使用模式,甚至提供了具体的 controller,model 和 view 部分。parsley2 可以让你自由的设计你的应用框架。如果你打算使用消息框架去建立一个经典的 MVC 架构，你可以去读一下 9 Building MVC Architectures。

本章描述了如何配置对象去发送和接收 消息。对于每一个为 AS3的元数据，MXML 和 XML 配置的配置选项的例子也包括在内。

5.1 派发消息(Dispatching Messages)

对于一个由 parsley 管 理的想派发消息的对象，你可以阅读下列选项：

- 5.3 托管事件(managed events)如果派发对象是一个普通的 EventDispatcher
- 5.4 注入消息派发器(Injected MessageDispatchers)当你的消息不是事件的子类
- 5.13 使用消息 API(Using the Messaging API)特殊情况下，你需要使用框架的 API

5.2 接收消息(Receiving Messages)

对于一个由 parsley 管 理的想接收和处理消息的对象，你可以阅读下列选项：

- 5.5 消息处理器(MessageHandlers)当派发一个特殊的消息，则调用该方法
- 5.6 消息绑定器(MessageBindings)当派发一个特殊的消息，则设置属性
- 5.7 消息拦截器(MessageInterceptors)在消息被处理或绑定之前， 进行拦截，有选择地取消或推迟并重新派发消息
- 5.8 错误处理器(Error Handlers)处理由其它处理器、绑定器、或拦截器引发的错误
- 5.9 异步命令方法(Asynchronous Command Methods)处理异步操作和它们的结果和错误
- 5.10 短生命周期命令对象(Short-lived Command Objects)通过一个对象创建去执行一个异步命令,执行完后销毁.
- 5.13 使用消息 API(Using the Messaging API)特殊情况下，你可以使用框架的 APE 去 进行各种注册

5.3 托管事件

如果你想派发由 parsley 托 管的消息是一个普通的 EventDispatcher，只需要下面二步：

- 用[Event]标 签声明你想派发的事件。这是一个非常好的最佳实践，不管你是否使用 parsley，因为这样提高了 类的可读性，ASDoc 将输出所有用这个标签声明的事件。
- 告诉 parsley 声明的事件中哪些应该被"管理"，这对消息接收者十分有用。你可以使用 AS3 Metadata, MXML 或 XML tags 编 写。

Metadata Example

```
[Event(name="loginSuccess",type="com.bookstore.events.LoginEvent")]
[Event(name="loginFailed",type="com.bookstore.events.LoginEvent")]
[Event(name="stateChange",type="flash.events.Event")]
[ManagedEvents("loginSuccess,loginFailure")]
public class LoginServiceImpl extends EventDispatcher implements LoginService {
    [...]

      private function handleLoginResult (user:User) : void {
          dispatchEvent(new LoginEvent("loginSuccess", user));
    }

 }
```

这个例子中，service 声 明了三个事件。其中的两个(loginSuccess 和 loginFailure)是应用程序事件，应该由 parsley 托 管，派发这些事件有关的所有对象。第三个事件是一个低级事件，仅关注于与那个 service 直接交 互的对象，这些对象只要注册一个普通的事件监听器。

这个例子只有一个方法，显示了如何将 结果转换成一个事件并派发它。因为 loginSuccess 被声明为了托管事件，它将通过 parsley 传递给所有的 MessageHandlers .

MXML Example

```
<Object type="{LoginServiceImpl}">
    <ManagedEvents names="['loginSuccess','loginFailure']"/>
</Object>
```

如果你在 MXML 中声明了一个托管事件，你可以忽略这里的[ManagedEvents]元 标签。注意，你所写的代码中必须包含[Event]元数据标签，因为那是用于表示 flash API 的。

XML Example

```
    <object type="com.bookstore.services.LoginServiceImpl">
            <managed-events names="loginSuccess,loginFailure"/>
        </object>
```

5.4 注入消息派发器(Injected MessageDispatchers)

有时候你不想用通过事件调用模式开发。有时一些特殊场景的事件，parsley 并不能"bubble"，stopPropagation 在 parsley 消息 时序处理过程中没有任何作用。你甚至想避开消息接收者通过 event.target 捕获住消息的派 发器。

在这些情况下，parsley 提供了使用任何一个类作为应用程序消息，而不管这个类是否继承了 flash.event.Evnet。你可以请求这个框架注入一个消息派发器去派发自定义的应用程序消息。假设 你创建了下列简单消息类：

```
class LoginMessage {

    public var user:String;

      public var role:String;

      function LoginMessage (user:String, role:String) {
          this.user = user;
          this.role = role;
    }

 }
```

你可以在 service 类中使用如下代码:

```
public class LoginServiceImpl implements LoginService {

        [MessageDispatcher]
        public var dispatcher:Function;

        public function login(user:String, role:String) : void {
            dispatcher(new LoginMessage(user));
        }

}
```

注意你的 service 类不用扩展 EventDispatcher。 这里只需要声明一个类型属性变量，并用[MessageDispatcher]标识，它将指示 parsley 在对象创建时，注入一个消息派发函数。你可以传递任何类型的对象给这个派发函数。

MXML Example

```
<Object type="{LoginServiceImpl}">
            <MessageDispatcher property="dispatcher"/>
    </Object>
```

XML Example

```
<object type="com.bookstore.services.LoginServiceImpl">
            <message-dispatcher property="dispatcher"/>
    </object>
```

如果你不想使用元数据标签，你可以通过 MXML 或 XML 配 置。

5.5 消息处理器(MessageHandlers)

消息处理器是接收端最常见的方法。当一个特殊的应用程序消息被捕获，你可以声明方法进行调用。通常，这个方法仅通过参数类型选择：

Metadata Example

```
[MessageHandler]
   public function handleLogin (message:LoginMessage) : void {
```

无论何时一个匹配的消息类(包括子类)被捕获，将调用这个方法。

MXML Example

```
<Object type="{LoginAction}">
        <MessageHandler method="handleLogin"/>
    </Object>
```

XML Example

```
<object type="com.bookstore.actions.LoginAction">
          <message-handler method="handleLogin"/>
    </object>
```

这里还可以处理一个特殊情况，你分解一个消息的属性作为不同的消息处理器的参数：

```
[MessageHandler(type="com.bookstore.events.LoginMessage",messageProperties="user,role"]
   public function handleLogin (user:User, role:String) : void {
```

注意：你必须声明消息类型，因为它不能由参数类型检查

最后，你可能遇到消息类的选择不足的问题。如果你在不同的场景和应用程序派发同样的消息类，你可能想更深入的定义消息选择过程。参见5.11 Using Selectors 详细描述。

5.6 消息绑定器(MessageBindings)

消息绑定一般用于以下场景：

直接把一个消息类中的属性赋给目标类的属性，一般过程是先声明一个消息处理函数，通过 MessageHandler 绑定该函数，然后再函数内 将消息类的属性赋给相应属性，过程太麻烦。

如果你要将一个类的属性和一个消息属性绑定，消息绑定就很方便，即当一个匹配的消息类被捕获，框架自动根据消息属性更新类的属性。在下面的例子为例，用户 属性与 LoginMessage 类的属性绑定，如下例：

Metadata Example

```
  [MessageBinding(messageProperty="user",type="com.bookstore.events.LoginMessage")]
      public var user:User;
```

MXML Example

```
<Object type="{LoginServiceImpl}">
        <MessageBinding
            targetProperty="user"
            messageProperty="user"
           type="{LoginMessage}"/>
    </Object>
```

XML Example

```
<object type="com.bookstore.services.LoginServiceImpl">
        <message-binding
            target-property="user"
            message-property="user"
           type="com.bookstore.events.LoginMessage"/>
    </object>
```

如果你想在 MessageHandlers 中使用 MessageBindings 的选择功能，参见5.11 Using Selectors

5.7 消息拦截器(MessageInterceptors)

这是接收端的第三种选择。当应用程序状态或用户自己决定消息是否传给处理器和绑定器时， 可通过拦截器实现。拦截器具有下列特征：

- 所有注册的拦截器在处理器或绑定器之前执行
- 拦截器可以有选择的中止消息处理和在之后的某个时刻恢复

例：当你有未登录就可以访问的应用程序，一些行动没有登录则不让操作。在这种情况下可以暂停拦截消息处理，显 示一个登录对话框，并成功登录后恢复的消息处理。

另一个例子：在任何一个删除操作之前显示一个简单的警告，通常如下：

```
public class DeleteItemInterceptor {

    [MessageInterceptor(type="com.bookstore.events.ShoppingCartDeleteEvent")]
```

```
        public function interceptDeleteEvent (processor:MessageProcessor) : void {
            var listener:Function = function (event:CloseEvent) : void {
                if (event.detail == Alert.OK) {
                    processor.proceed();
                }
             };
             Alert.show("Do you really want to delete this item?", "Warning",
                 Alert.OK | Alert.CANCEL, null, listener);
        }

 }
```

当用户点击取消按钮时，MessageProcessor 将不会恢复执 行，但处理器或绑定器将被执行。

正如 MessageBindings，你必须声明消息类型，因为它不能根据方法签名检测。拦截方法总是有一个简单的 MessageProcessor 类型参数。你也可以使用 MXML 或 XML 代替元数据标签进行声明。

MXML Example

```
<Object type="{LoginServiceImpl}">
     <MessageInterceptor
           method="interceptDeleteEvent"
           type="{ShoppingCartDeleteEvent}"/>
</Object>
```

XML Example

```
<object type="com.bookstore.services.LoginServiceImpl">
          <message-interceptor
              method="interceptDeleteEvent"
              type="com.bookstore.events.ShoppingCartDeleteEvent"/>
     </object>
```

拦截作为消息框架的一个特征，框架必须传递 MessageProcessor 实例给你，你就能反 控或处理消息

5.8 错误处理器(Error Handlers)

在2.1版本后，你可以设置一个方法去处理或拦截消息引发的错误：

```
[MessageError(type="com.bookstore.LoginEvent")]
   public function handleError (processor:MessageProcessor, error:IOError) : void;
```

上例中，当 LoginEvent 类型的消息抛出一个 IOError 错误，这个方法将进行处理。你也可以创建一个全局的错误处理方法，接收任何消息引发的错 误类型

```
[MessageError]
   public function handleError (processor:MessageProcessor, error:Error) : void;
```

最后，因为错误处理器通过标签只能监听单一的范围，你可以添加一个错误处理器自动附加到每个应用程序中。你可以通过 GlobalFactoryRegistry 实现：

```
var provider:ObjectProvider = Provider.forInstance(this);
     var handler:MessageErrorHandler
```

```
                        = new DefaultMessageErrorHandler(provider, "handleError", Object);
GlobalFactoryRegistry.instance.messageRouter.addErrorHandler(handler);
```

5.9 异步命令方法(Asynchronous Command Methods)

2.2版本的 parsley 框架也包含了几个由 cairngorm3项目提出的设计思想。详见 Integration Library.

标签集不仅不包括[MessageHandler]标签，还包括多个不同类型的标签进行消息处理。通过[Command] 标签，你可以让框架去管理异步命令的执行：

```
[Inject]
     public var service:RemoteObject;

     [Command(selector="save")]
     public function saveUser (event:UserEvent) : AsyncToken {
           return service.saveUser(event.user);
     }
```

[Command]标签支持与[MessageHandler] 标签相同的属性集。匹配消息的逻辑也一样。上例中，捕获 UserEvent(包括子类)后，命令将执行，选择器 save 通常是指事件类型。

所不同的是，命令不要求有一个像普通消息处理器需要一个 void 返回类型。相反，它可以指定一个类型，该框架可以用它来为你管理异步命令执行。在这种情况 下，它是这是由远程调用返回 AsyncToken。现在，该框架还将为你管理的最终结果或错误。现在，任何对象都可以包含一个方法，用于处理调用的结果：

```
     [CommandResult(selector="save")]
        public function handleResult (user:User, event:UserEvent) : void {
```

上例中，User 实例 由远程调用返回，与触发动作的原始消息一起传给结果处理器。第二个参数是可选的，如果你忽略它，你必须用标签描述消息类型，以进行精确匹 配：

```
     [CommandResult(type="com.foo.events.UserEvent" selector="save")]
        public function handleResult (user:User) : void {
```

这是必要的，就象普通消息处理器一样，parsley 独立依赖于目标选择器的字符串标识符(事件类型)。它总是消息类型的混合体和带选项的选择器。

结果参数也是可选的，可以简单地提交：

```
        [CommandResult(type="com.foo.events.UserEvent" selector="save")]
           public function afterUserSaved () : void {
```

或者它可能是 ResultEvent 而不是实际结果值，如果你需要访问从服务器端返回的头部信息，可写成下面代码：

```
       [CommandResult(selector="save")]
          public function handleResult (result:ResultEvent, trigger:UserEvent) : void {
```

为了接收错误事件，其它标签也可以使用：

```
       [CommandError(selector="save")]
```

```
            public function handleResult (fault:FaultEvent, trigger:UserEvent) : void {
```

[CommandError]参数也是可选的，匹配规则也[CommandResult]标签一样。

最后，你也可以观察执行命令的状态：

```
    [CommandStatus(type="com.foo.events.UserEvent" selector="save")]
        public var isSaving:Boolean;
```

如果一个或多个异步命令匹配描述的类型，并且选择器正在执行，这个 Boolean 类型总是 True，否则为 false。这个对于任务处理很方便，比如在命令执行期间将按钮置灰。

可支持的命令类型

上面的例子展示了执行远程调用的例子和返回一个 AsyncToken。命令模式也可用于 Cinnamon Remoting，返回 ServiceRequest 实例而不是 FLEX 的 RemoteObjects。最后，命令模式与 spicelib 任务框架集成，一个命令也可将任务作为返回类型。

你可以通过你自己的 CommandFactories 扩展框架的限制，创建更多的命令标签，接口如下：

```
public interface CommandFactory {

    function createCommand (returnValue:Object,
              message:Object, selector:* = undefined) : Command;

 }
```

方法 createCommand 的返回值是一个方法，该方法用于执行命令(比如 AsyncToken)，选择器触发命令。它也返回一个 Command 实例，大多数是 AbstractCommand 的 子类，工厂实现类必须进行注册：

```
 GlobalFactoryRegistry.instance.messageRouter
      .addCommandFactory(MyReturnType, new MyCommandFactory());
```

addCommandFactory 的声明如下：

```
  function addCommandFactory (type:Class, factory:CommandFactory) : void;
```

它描述的命令的返回类型和工厂实例本身。

5.10 短生命周期命令对象(Short-lived Command Objects)

上节命令模式已展示了对象执行命令不会发生任何改变。标签 [Command], [CommandResult] 和 [CommandError] 可以用于任何一个托管对象的方法。它可以是 一个单例对象或者一个视图。每个对象可以包含多个命令方法和多个结果处理器。结果处理器可能是相同对象或不同对象。总之，这些标签仅控制单个方法或属性。

框架允许定义一个命令。如果不应用标签给个单个方法，你可以对一个完整的命令对象进行配置。这个对象仅是单个方法，用于执行命令，或者是一个接收结果的方 法和一个事件型的错误方法。结果处理器仅被执行同一个实例的命令所调用，这个命令对象可能象这样：

```
public class SaveUserCommand {

    [Inject]
    public var service:RemoteObject;
```

```
        public function execute (event:UserEvent) : AsyncToken {
            return service.saveUser(event.user);
        }

        public function result (user:User) : void {
            // do something with the result
        }

         public function error (fault:Fault) : void {
            // do something with the result
        }

    }
```

在配置类中定义[DynamicCommand]标签:

```
            <DynamicCommand type="{SaveUserCommand}" selector="save"/>
```

这是一个最小的配置，描述了选择器和消息类型。而且我们不一定在命令对象中使用常规的命令元数据标签。上节展示了选择方法中命令处理器、结果处理器和错误 处理器的默认名，你也可以用 DynamicCommand 标签重写。下例我们重写这三个方法名：

```
<DynamicCommand
         type="{SaveUserCommand}"
         selector="save"
        execute="saveUser"
        result="handleUser"
        error="handleFault"/>
```

标签也支持[MessageHandler] 或 [Command] 标签的所有属性：

```
<DynamicCommand
         type="{SaveUserCommand}"
         selector="save"
      scope="local"
        order="1"/>
```

上例中，当触发消息被同一上下文的对象捕获，命令将先执行。order 用于决定执行的顺序，但不能决定结果处理和错误处理的顺序。

除了特殊语法，DynamicCommand 标 签也允许象正常对象标签一样，拥有相同的子标签集：

```
 <DynamicCommand type="{SaveUserCommand}" selector="save">
            <ConstructorArgs>
                <ObjectRef idRef="someDependency"/>
            </ConstructorArgs>
        </DynamicCommand>
```

当然，命令对象可以由其它对象的结果处理器组成。如果你使用了[CommandResult] 标签，这与在一个方法上标识[Command]标签，或者 DynamicCommand 标签声明过的命令对象，其效果是一样的。尽管同一个命令对象拥有 private 结果处理器，任何一个对象都可以接收到结果。

命令对象生命周期

默认是容器为每个匹配消息创建一个新的命令实例。多个命令对象可以并发执行，并且不进行任何交互。如果只想使用一个实例，通过设置 stateful 属性修 改：

```
<DynamicCommand type="{SaveUserCommand}" selector="save" stateful="true"/>
```

现在，当第一个匹配消息被捕获，实例才被实例化，随后，该实例将作为一个容器托管对象，当捕获新的消息后，该实例被恢复，以节省对象创建和初始化时间。对 于有状态命令，你不能用私有属性去保存信息，因为多个命令将相互交互，并且覆盖那个值。

5.11自定义选择器(Using Selectors)

本例主要与 MessageHandlers 相关，MessageBindings 和 MessageInterceptors 匹配方法或属性总是由消息类型 决定。有时候，如果你在不同的场景或应用程序 states 中派发相同的消息，就出出现问题。你必须重新定义选择过程。

如果你使用了 Event 类的类型属性作为选择器：

```
[MessageHandler(selector="loginSuccess")]
    public function handleLogin (message:LoginEvent) : void {
          [...]
    }

    [MessageHandler(selector="loginFailure")]
    public function handleError (message:LoginEvent) : void {
        [...]
    }
```

上例中，当 LoginEvent 实例的类型属 性值为 loginSuccess，handleLogin 方法将被调用。

对于自定义类型，不用扩展 flash.events.Event。默认的选择器属性可以在消息类的属性上，用[Selector]元标签声 明。

```
class LoginMessage {

    public var user:User;

     [Selector]
    public var role:String;

     [...]
}
```

现在，你可以根据用户的角色，选择消息处理器：

Metadata 例子

```
   [MessageHandler(selector="admin")]
        public function handleAdminLogin (message:LoginMessage) : void {
```

MXML 例子

```
  <Object type="{AdminLoginAction}">
          <MessageHandler method="handleAdminLogin" selector="admin"/>
      </Object>
```

XML 例子

```
    <object type="com.bookstore.actions.AdminLoginAction">
                <message-handler method="handleAdminLogin" selector="admin"/>
        </object>
```

5.12 使用范围(Using Scopes)

Parsley 2.0中，每个子上下文可以共享父上下文的消息路由。消息总是全局进行派发。这种做法将简化模块化开发和弹出窗体等。scope 的引入将更灵活，允许派发 上下文的子集。

全局和局部的范围

对于 Parsley 2.1的默认范围是全局的范围，应用程序启动只创建一个根上下文，每个上下文根据全局范围创建，并没有父上下文，但可以被子上下文共享。另外，每个上下文 将创建自己的局部上下文，这些可以不被子上下文共享，如下图所示：

因为全局范围唯一性，它是所有的配置标签的默认范围。

```
        [MessageHandler(selector="save")]
            public function save (event:ProductEvent) : void {
```

处理器监听任何一个上下文派发的 ProductEvents 事件。

```
        [MessageHandler(selector="save", scope="local")]
            public function save (event:ProductEvent) : void {
```

现在，处理器仅监听相同上下文派发的事件。当然，scope 属性可用于所有的消息接收类型，比如 MessageInterceptor 和 MessageErrorHandler。

对于发送端，默认行为是不同的。对于每个[ManagedEvents]标 签，scope 并没有精确描述消息发送过程中经历的上下文。接收端可以决定它想监听的范围，如果你想限制发送端的范围，你可以在[ManagedEvents] 标签内使用属性 scope:

```
  [ManagedEvents("save,delete", scope="local")]
```

自定义范围

如果你即不想用全局消息，也不想用局部消息，你也能创建你自义的范围。下面假设你开发一个大型的 AIR 项目。根窗体组件可能以根应用上下文作为基础，创建 一个新的上下文，这个窗体的部分组件也可以包含一些子上下文。你可以象下图所示，创建自定义上下文：

窗体范围是自定义的，存在两个并列的默认范围。现在，如何构建框架去创建范围？还是应该根据根上下文创建，可以用 MXML 描述如下：

```
<parsley:ContextBuilder>
    <parsley:FlexConfig type="{ServiceConfig}"/>
    <parsley:FlexConfig type="{ControllerConfig}"/>
    <parsley:XmlConfig file="logging.xml"/>
```

```
        <parsley:Scope name="window" inherited="true"/>
</parsley:ContextBuilder>
```

或者添加 scope 到 CompositeContextBuilder：

```
var builder:CompositeContextBuilder = new DefaultCompositeContextBuilder(viewRoot);
    builder.addScope("window", true);
    FlexContextBuilder.merge(MyWindowConfig, builder);
    builder.build();
```

addScope 的第一个参数是 scope 的名称，该名称必须唯一。这就允许你在消息处理器上定义 scope 的名称。

```
            [MessageHandler(selector="save", scope="window")]
            public function save (event:ProductEvent) : void {
```

第二个 boolean 参数表示子上下文是否能共享 scope。

5.13 使用消息 API(Using the Messaging API)

通常应用代码应避免直接与 parsley API 交互。但是特殊情况下或你想扩展框架，建立另一个新框架时，你可能需要注册消息处理器或自动绑定。MessageReceiverRegistry 接口包含下列方法：

```
function addTarget (target:MessageTarget) : void;
    function addInterceptor (interceptor:MessageInterceptor) : void;
    function addErrorHandler (handler:MessageErrorHandler) : void;
    function addCommand (command:CommandTarget) : void;
    function addCommandObserver (observer:CommandObserver) : void;
```

消息接收器分成五类：MessageTarget 是普通接收器，实现类包 括 MessageHandler 和 MessageBinding。MessageInterceptor 与同名的标签对应，MessageErrorHandler 与标签[MessageError]对应。CommandTarget 是 一个异步命令的 target，CommandObserver 监听结果或错误。接口包括五个方法，用于添加这五类接收器。

为了获取 MessageReceiverRegistry 实例，你可以注 入上下文实例到你的类中。你必须先确定你的接收器的 scope，下面的例子是注册一个消息处理器为全局 scope：

```
class SomeExoticClass {
    [Inject]
    public var context:Context;

     [Init]
    public function init () : void {
        var registry:MessageReceiverRegistry
                = context.scopeManager.getScope(ScopeName.GLOBAL).messageReceivers;
        var target:MessageTarget
                = new MessageHandler(Provider.forInstance(this), "onLogin");
        registry.addMessageTarget(target);
     }
}
```

当你在 Context 上设置了[Inject]标签，parsley 将注入上下文实例给 这个类。

最后，你可以使用 ScopeManager 派 发消息：

```
  context.scopeManager.dispatchMessage(new LoginMessage(user, "admin"));
```

当直接通过 ScopeManager 派发，消息将经过该 context 管 理的所有 scope。这样接收端可以决定哪个 scope 去监听。

万一你不想让发送端选择，你必须通过个体的 scope 派发：

```
 var scope:Scope = context.scopeManager.getScope(ScopeName.LOCAL);
        scope.dispatchMessage(new LoginMessage(user, "admin"));
```

6-对象生命周期

本章对配置文件进行额外说明。

6.1使用工厂(Using Factories)

有时候，直接配置目标对象不可能有效。可能你在对象创建时想执行一些复杂的逻辑，以至于你想用工厂模式代替现在模式。AS3中有一个 factory 类，你 可以象其它类一样配置它，使用 Metadata, MXML 或 XML 标签。不同之处在于方法被标识成了 factory 方法：

```
class CategoryFactory {
        public var categoryName:String;

          [Inject]
        public var bookManager:BookManager;

          [Factory]
        public function createCategory () : Category {
                var books:Array = bookManager.getBooksForCategory(categoryName);
                return new Category(categoryName, books);
        }
}
```

你也能使用这个 factory 在你的配置文件中。

```
<mx:Object
            xmlns:mx="http://www.adobe.com/2006/mxml"
          xmlns:model="com.bookstore.model.*">

        <model:BookManager/>
      <model:CategoryFactory id="historyCategory" categoryName="history"/>
      <model:CategoryFactory id="politicsCategory" categoryName="politics"/>
      <model:CategoryFactory id="artsCategory" categoryName="arts"/>

    </mx:Object>
```

我们上面定义的每一个 factories 将获得 BookManager 实例注入，并且产生 Category 实例。

The special thing about using factories is that under the hood Parsley actually creates **two** object definitions for each factory: One for the factory itself and one for the type the factory method produces. This means that you can also place metadata tags on the target type (in this case the Category class) if you want the object that the factory produces to send and receive application messages managed by Parsley for example.

This also means that you can use the factory **and** the objects it creates at injection points, although in most cases you'll be interested in the objects produced by the factory:

```
[Inject(id="historyCategory")]
```

```
public var historyBooks:Category;
```

It is recommended not to use factory methods with a return type of Object like this:

```
[Factory]
public function createInstance () : Object {
```

It would work, Parsley would happily process this kind of factory method. But you'll lose some of Parsley's useful capabilities, since it uses the return type of the method for producing the object definition for the target type. If the target type is just Object, the metadata tags on the objects this factory actually produces would not be processed, since this happens **before** the factory method will be invoked for the first time. Furthermore you cannot use objects produced by such a factory for Dependency Injection by Type, since the type can only be determined dynamically. You would then be constrained to Injection by Id.
Of course, like with most other Parsley features, you may also declare the factory method in MXML or XML. This may come in handy in some edge cases, for example for a factory that has more than one method that produces objects. In this case placing metadata tags in the class itself would not be possible (only one [Factory] tag is allowed).
MXML Example

```
<Object id="monkey" type="{ZooFactory}">
    <Factory method="createMonkey"/>
 </Object>
<Object id="polarBear" type="{ZooFactory}">
    <Factory method="createPolarBear"/>
 </Object>
```

XML Example

```
<object id="monkey" type="com.example.ZooFactory">
    <factory method="createMonkey"/>
 </object>
<object id="polarBear" type="com.example.ZooFactory">
    <factory method="createPolarBear"/>
 </object>
```

6.2 Asynchronous Object Initialization

A lot of operations in the Flash Player execute asynchronously. So it might happen that you want an object configured in the Parsley IOC Container to load some data or assets first, before the rest of the Context gets initialized and before this asynchronously initializing object gets injected into other objects. In this cases you can use the [AsyncInit] tag on any EventDispatcher that fires events when the initialization is completed (or if it has failed):

```
[AsyncInit]
public class FileLoader extends EventDispatcher {

    public var filename:String;

    [Init]
    public function init () : void {
        var loader:URLLoader = new URLLoader();
        loader.addEventListener(Event.COMPLETE, fileLoaded);
        loader.addEventListener(IOErrorEvent.IO_ERROR, handleError);
        loader.addEventListener(SecurityErrorEvent.SECURITY_ERROR, handleError);
        loader.load(new URLRequest(filename));
    }
    private function fileLoaded (event:Event) : void {
        // handle loaded file
        dispatchEvent(new Event(Event.COMPLETE));
    }

    private function handleError (event:ErrorEvent) : void {
        dispatchEvent(new ErrorEvent(ErrorEvent.ERROR, false, false, event.text));
    }

}
```

In the example above the [AsyncInit] tag tells the framework that this is an asynchronously initializing object. In the method annotated with [Init] which will be invoked after configuration and dependency injection has been processed for that object (see 6.4 Object Lifecycle Methods for details) we start the loading process. Depending on whether the loading succeeds or not we then dispatch either an Event.COMPLETE or an ErrorEvent.ERROR. The container will listen for those two events. In case of Event.COMPLETE it will proceed with Context initialization. In case of ErrorEvent.ERROR the whole Context initialization process will be cancelled.
Switching event types
Event.COMPLETE and ErrorEvent.ERROR are the default event types to signal whether initialization has completed or failed. They can be switched with attributes of the [AsyncInit] tag:

```
[AsyncInit(completeEvent="myCustomCompleteEvent",errorEvent="myCustomErrorEvent")]
```

Initialization Order
AsyncInit objects will always be initialized before regular objects unless you define an order attribute for them explicitly which always has a higher precedence. But if you have more than one object marked with [AsyncInit] you may want to declare the initialization order explicitly as explained in the next section. The order will not be determined by analyzing the dependencies, as they are processed on the fly during initialization and also allow for bidirectional or circular dependencies which would make it hard to determine the order automatically. But this really is only necessary for AsyncInit objects and only if you wish to guarantee that those are ready when they are injected into other objects, otherwise everything will be resolved automatically.

6.3 Object Initialization Order

In case you want to explicitly specify the initialization order you can do that with the order attribute:
MXML Example

```
<Object id="welcomeText" type="{FileLoader}" order="1">
    <AsyncInit/>
    <Init method="init"/>
    <Property name="filename" value="welcome.txt"/>
</Object>
```

XML Example

```
<object id="welcomeText" type="com.example.FileLoader" order="1">
    <async-init/>
    <init method="init"/>
    <property name="filename" value="welcome.txt"/>
</object>
```

The default value if you omit this attribute is int.MAX_VALUE so that all objects without an order attribute will execute last and in arbitrary order. The attribute can be set to any positive or negative integer value.

The order attribute can also be used for objects that initialize synchronously. For any asynchronously initializing object in the sequence the container will wait for that object to complete its initialization before starting with the next one.

6.4 Object Lifecycle Methods

If you want the Parsley Container to invoke methods on your object when it is created or when it is destroyed, you can add the [Init] or [Destroy] metadata tags to the corresponding methods:

```
[Init]
public function init () : void {
	[...]
}
[Destroy]
public function dispose () : void {
	[...]
}
```

The methods marked with [Init] get invoked after the object has been instantiated and all injections have been processed.
The methods marked with [Destroy] get invoked after the Context instance they belong to has been destroyed with Context.destroy() or when the object was removed from the Context. See 6.4 Object Lifecycle Methods for details.
For Flex Components declared in regular MXML files and wired to the Context as described in 7 Dynamic View Wiring the lifecycle is different: For those objects the methods get invoked whenever the object is added to or removed from the stage respectively. Of course the [Destroy] method additionally gets invoked if the Context the object was wired to was destroyed.

6.5 Lifecycle Observer Methods

Added in 2.2 this functionality opens some new doors for convenient ways to observe or modify other objects without the need to add something to their configuration. This might be particularly useful for short-lived objects like views or commands which might enter and leave the Context at any point in time and thus are not valid sources for regular injection points. By observing these instances you can still get hold of a reference to such a short-lived object.

```
[Observe]
public function enhance (panel:ShoppingCartPanel) : void;
```

Now this method will be invoked whenever a new instance of ShoppingCartPanel (or any subclass) has been fully configured.
The default without attributes like shown above is equivalent to:

```
[Observe(phase="postInit", scope="global")]
```

So you could also use a different phase of the lifecycle (like preDestroy to get notified when an object leaves the Context) and can also control the scope and listen only for matching types in the same Context with scope="local" for example. Scoping works the same way like scopes for messaging as explained in 5.12 Using Scopes.
With this mechanism you simply plug in some new class that contains such a tag and suddenly existing classes can be enhanced without the need to touch their configuration. This is somewhat analogous to the existing [Factory] tag which can be used to customize the object instantiation. With this tag you can customize the object configuration after it has been instantiated. In both cases you do not even depend on the Parsley API in any way.
Supported lifecycle phases

preConfigure	Invokes the observer right after the object was instantiated but before any dependency injection was performed.
preInit	Invokes the observer after dependency injection has been performed but before the init method of the object (if available) gets invoked.
postInit	The default if the phase attribute is omitted. Invokes the observer after dependency injection has been performed and the init method of the object (if available) has been

	invoked.
preDestroy	Invoked when the object is removed from the Context but before the destroy method on the object (if available) gets invoked.
postDestroy	Invoked when the object is removed from the Context and after the destroy method on the object (if available) was invoked.

6.6 Dynamic Objects

Since version 2.1 there is a new interface DynamicContext. Such a Context allows you to dynamically add and remove objects from the Context. In most cases this is not intended to be used by normal application code, but you can easily build extensions on top of that functionality that require some more flexibility than the regular Context. Internally the DynamicContext will be used for wiring views which are added and removed from the Context depending on the time they are placed on the stage and (starting from version 2.2) for short-lived Command objects which are only added to the Context to perform their asynchronous operation and then immediately get removed again.
Dynamic Objects almost behave the same like regular objects. In particular this means:

- You can inject any regular object into the dynamic object.
- The dynamic object can send and receive messages to and from any of the available scopes.
- The dynamic object can have lifecycle methods like [Init] and [Destroy].

There is only one significant restriction for using dynamic objects:

- You cannot inject a dynamic object into another object.

This restriction is natural, since dependency injection comes with validation which would not be possible if the set of objects available for injection could change at any point in time. This is no real limitation anyway since you can get hold of these dynamic objects through their lifecycle methods, in case you want to get notified whenever an object of a particular type enters the Context (or any scope of Contexts).
You can create a dynamic Context from a regular one:

```
var dynamicContext:DynamicContext = context.createDynamicContext();
```

This way the dynamic Context is connected to the regular one and all objects living in that Context or one of its parents can be injected into the dynamic objects. You can then simply add any object to that Context:

```
var instance:Object = ...;
var dynamicObject:DynamicObject = dynamicContext.addObject(instance);
```

In this case an ObjectDefinition will be created for the existing instance and metadata will be processed for the type of that instance, performing any required dependency injection, message receiver registration or lifecycle method invocation.
The object can be removed from the Context at any point in time:

```
dynamicObject.remove();
```

At this point the method marked with [Destroy] will be invoked on that object (if existent) and any message receiver registrations will be terminated. The object is then fully removed from the Context.
For building extensions which talk to a DynamicContext instance from within a ObjectDefinitionDecorator or ObjectDefinitionFactory implementation there are two interesting variants of the addObject method shown above. First it is possible to pass an additional ObjectDefinition to the method:

```
var definition:ObjectDefinition = ...;
var instance:Object = ...;
var dynamicObject:DynamicObject = dynamicContext.addObject(instance, definition);
```

In this case the definition will not be created by the dynamic Context like in the preceding example. Instead the specified definition will be used. In version 2.2 this mechanism will be used internally to support the new option to configure dynamically wired views in MXML. An existing instance will have to be configured by an ObjectDefinition then which has been created elsewhere.

Finally you can also just pass a definition to the dynamic Context and it will create a new instance based on that definition:

```
var definition:ObjectDefinition = ...;
var dynamicObject:DynamicObject = dynamicContext.addDefinition(definition);
```

The instance created by the Context can then be accessed like this:

```
var instance:Object = dynamicObject.instance;
```

In all these different use cases removal of the object happens in the way already demonstrated:

```
dynamicObject.remove();
```

7-动态视图注入

由于 parsley 可以在配置文件内定义托管对象，包括视图元素。对于 FLEX 应用程序，这种方法并不理想，因为你可能想在你的 MXML 文件的组件中声明 你的组件，而不是 parsley 配置文件中声明组件。因此，你需要一个机制将 parsley 配置文件内的对象与 MXML 文件中的组件联系起来。

parsley 在本章提供了使用案例。

7.1 初始化视图注入(Initializing View Wiring Support)

对于视图注入，每个上下文需要一个或多个"**视图根**"，视图根是一个 DisplayObject 对象，用来监听来相关上下文中组件的事件。你可以用 MXML 标签初始化上下文或通过编程实现。

用 MXML 标签初始化上下文

在 FLEX 应用程序中，你可以使用 ContextBuilder 标签 (version 2.2版本引入)。这将用自动在**视图根**上使用**文档对象**(document object)。

```
<parsley:ContextBuilder config="{BookStoreConfig}"/>
```

你自己想直接指定视图根的情况很罕见，可以如下配置：

```
<parsley:ContextBuilder config="{BookStoreConfig}" viewRoot="{someOtherDisplayObject}"/>
```

上下文编码初始化

如果你通过程序初始化框架，视图根必须描述：

XML 配置(XML configuration)

```
var viewRoot:DisplayObject = ...;
    XmlContextBuilder.build("bookStoreConfig.xml", viewRoot);
```

多个配置机制

```
    var viewRoot:DisplayObject = ...;
var builder:CompositeContextBuilder = new DefaultCompositeContextBuilder(viewRoot);
FlexContextBuilder.merge(BookStoreConfig, builder);
XmlContextBuilder.merge("logging.xml", builder);
builder.build();
```

7.2 明确组件注入(Explicit Component Wiring)

如果你想将一个组件直接注入到组件自身所在的上下文中，你可以有二个选择。第一，你可以使用<Configure>标签：

```
<mx:Panel
             xmlns:mx="http://www.adobe.com/2006/mxml"
            xmlns:parsley="http://www.spicefactory.org/parsley">

            <parsley:Configure/>
            <!-- ... -->

         </mx:Panel>
```

在 FLEX4中，这将需要使用 <fx:Declarations>标签。 例如上例组件 Panel 将注入到上下文。你可以精确描述准备注入的对象，这个对象甚至可以不是组件：

```
<mx:Panel
              xmlns:mx="http://www.adobe.com/2006/mxml"
             xmlns:parsley="http://www.spicefactory.org/parsley"
             xmlns:view="myproject.view.*">

              <view:MyPanelPM id="model"/>
             <parsley:Configure target="{model}" />

              <!-- ... -->

         </mx:Panel>
```

本例中，我们声明一个表示层 model，并将这个 model 注入到上下文中。万一视图组件自身不是由 parsley 容器管理,你就不能在该视图组件内设置任何 parsley 元标签。这个 model 主要用于性能优化。参见下一节的第二部分将重点谈谈如何避免大型组件类的过多反射.

在 **flash** 应用程序中注入

在 flash 应用程序中，本章的注入方式并不能用。没有 MXML 去定义视图，我们经常直接在 XML 配置文件或 actionscript 配置文件中声明与视 图相关的对象。因此，这里并不需要明确描述如何注入他们。

万一你不想在上下文中定义视图，而是想使用动态注入上下文方式，这样你就能派发一个事件，就歇脚在 FLEX 中使用了<Configure/> 标签一样。为了达到这种效果，你首先应该描述视图根，去捕获这些 bubbing 事件。

```
    var viewRoot:DisplayObject = ...;
       ActionScriptContextBuilder.build(MyConfig, viewRoot);
```

这样，每个在视图根中描述的 DisplayObject 对象，当它被添加到 stage 中，就可以派发一个注入的事件。

```
public class MyView extends Sprite {

    function MyView () {
        addEventListener(Event.ADDED_TO_STAGE, configure);
    }
```

```
     private function configure () : void {
     dispatchEvent(new Event("configureView", true));
    }

     /* ... */

 }
```

7.3 无反射组件注入(Component Wiring without Reflection)

Parsley 对所有管理对象和组件反射操作代价很大，主要是由于其属性、方法和事件太多。一个组件可能花费30毫秒。parsley 维护一个内部反射缓 存，如果你使用了大多的不同组件，缓存的意义就不大了。一个小的应用程序，这种花费很少，但是对于大型系统，就要重点关注了。本节内容通过一些案例让你避 免性能大幅下降。

为了允许性能优化，Parsley 2.2引入标签 FastInject：

```
<mx:Panel
            xmlns:mx="http://www.adobe.com/2006/mxml"
           xmlns:parsley="http://www.spicefactory.org/parsley"
           xmlns:view="myproject.view.*">

            <mx:Script>
               <![CDATA[
                    import com.bookstore.model.MyPanelPM;

                     public var model:MyPanelPM;
               ]]>
           </mx:Script>

           <parsley:FastInject property="model" type="{MyPanelPM}" />

            <!-- ... -->

        </mx:Panel>
```

这有点类似与<Configure>标记和属性的目标的例子。这个模型将被注入到上下文中。但不同的是，在前一个例子中，模型是由内部的组件内的标签创建，然后由框架配置已实例化的模型对象。在这个例子中，容器 panel 创建模型对象，然后注入到组件中。这种方法的主要优点是，该模型可以在组件之间共享，而前例不行。这只是解决性能是一个 方法。

FastInject 标签也允许通过子标签定义多个注入：

```
   <parsley:FastInject injectionComplete="init();">
           <parsley:Inject property="model" type="{MyPanelPM}"/>
            <parsley:Inject property="status" type="{MyApplicationStatus}"/>
        </parsley:FastInject>
```

如果你想执行所有的注入的话，必须将它们合并为单一的父标签。如果你使用独立的标签，你需要手动检查哪些已经注入已处理。在上面的例子可以保证，只有所有 的注入都执行后，init 方法只会被调用。

<FastInject>标签的事件 creationComplete 或 addedToStage 与组件事件类似，但是并不相同。当相应的组件事件被

捕获且注入已经执行完，才找一个合适的机会初始化组件。

7.4 自动组件注入(Automatic Component Wiring)

自 Version2.2后，你还可以通过自动视图注入，避免为单个组件类添加具体配置。但是 parsley 存在多上下文环境，自动注入会导致不知道哪些上 下文中。目前只能基于他们所处的视图层次的位置。一个组件被注入到最接近的上下文中。

因为这个功能很新，没有严格测试，因此，必须注意 parsley 的版本号：

MXML

```
<parsley:ContextBuilder>
            <parsley:Settings autowireViews="true"/>
            <parsley:FlexConfig type="{MyConfig}"/>
    </parsley:ContextBuilder>
```

ActionScript

```
  GlobalFactoryRegistry.instance.viewManager.autowireFilter.enabled = true;
```

在第一个上下文创建之前，上一行语句必须执行。正如你所期望的，autowireFilter 可是选 的。你可以实现自己的选择逻辑，告诉框架哪个组件应该注入。默认仅注入 MXML 或 XML 文件中<view>标签的组件。这种方式真正做到了统 一配置。你在一个配置文件中就可以看到托管的视图和对象。一个示例配置片段可能是这样的：

```
 <View type="{MainView}"/>
  <Object type="{MainViewPM}"/>

  <View type="{StatusBar}"/>
  <Object type="{StatusBarPM}"/>

  <View type="{ImageTileList}"/>
  <Object type="{ImageTileListPM}"/>

  <View type="{ImagePreviewWindow}"/>
  <Object type="{ImagePreviewWindowPM}"/>
```

这里简单的展示了视图定和相关视图模型。仅仅这里的视图将注入到上下文中。本例中，我们还使用了空视图标签，你也可以象对象标签一样隐藏标签，参见7.6 MXML and XML Configuration。
你也可以实现你自定义的 autowire 过滤器，最简单的方式是继承 DefaultViewAutowireFilter 并 覆盖 filter 方法：

```
public class MyAutowireFilter extends DefaultViewAutowireFilter {

    public override function filter (view:DisplayObject) : ViewAutowireMode {
        if (... someCondition ...) {
            return ViewAutowireMode.ALWAYS;
        }
        else {
            return ViewAutowireMode.NEVER;
        }
    }
}
```

最终，使用 filter 如下所示：

```
GlobalFactoryRegistry.instance.viewManager.autowireFilter = new MyAutowireFilter();
```

7.5 元数据配置(Metadata Configuration)

大多数情况下，元数据是够用的，组件上的元数据标签与容器创建的对象具有相同的效果。

```
<mx:Panel
         xmlns:mx="http://www.adobe.com/2006/mxml"
        addedToStage="dispatchEvent(new Event('configureView', true));">

         <mx:Script>
            <![CDATA[
              import com.bookstore.events.*;
              import com.bookstore.model.*;

                [Bindable]
              private var user:User;

                [Inject]
              public var model:LoginPanelPM;

                [MessageHandler]
              public function handleLogin (event:LoginEvent) : void {
                     this.user = event.user;
                }
           ]]>
       </mx:Script>

        <mx:text text="Current User: {user.name}"/>
       <!-- some more components ... -->

   </mx:Panel>
```

一些开发人员要避免添加过多的逻辑到组件中。大部分情况下，在组件的表示模型类上使用 parsley 标签[Inject]封装所有的逻辑和数据。

7.6 MXML 和 XML 配置配置(MXML and XML Configuration)

自 Version2.2，你可以在 MXML 或 XML 上配置视图。在两种情况下会很方便：一、你可以在不同的脚本中使用相同的组件类；二、你可能想使用配置 机制去描述如同7.4 Automatic Component Wiring 演示的自动注入。

这个功能已用在了 Flicc 中，当它被注入到上下文中，容器将配置信息作用到视图上。作用于注入组件的配置信息

将根据 ID 或 type 匹配：

```
<View id="myPanel" type="{MyPanel}">
            <Property name="url" value="http://www.somewhere.com/"/>
            <MessageHandler method="execute"/>
      </View>
```

这看上去象一般对象的配置信息。视图标签的使用说明，parsley 等到任何匹配的组件动态地获取注入的上下文，应用配置后，再创建该类的实例。

Parsley 首先根据 ID 匹配。ID 既可以是 FLEX 组件的名字，也可以是配置标签的 ID

```
<parsley:Configure configId="someId"/>
```

如果根据 ID 没有匹配，框架将根据类型匹配。如果定义含糊，则抛出错误。如果没有匹配类型，parsley 将使用传统机制去处理元数据。

7.7 组件生命周期(Component Lifecycle)

由于 FLEX 组件与 IOC 容器有关联，组件生命周期与直接定义在容器内的对象是不同的。这个由标签 [Init] 和 [Destroy]决定。

方法标签[Init] Methods annotated with[Init]

对于一个声明在 parsley 配置文件中的对象，容器创建并配置好对象后才开始执行这些方法。对于一个动态注入的 FLEX 组件，容器捕获组件派发的事件和 所有的注入都注入后，才开始执行这些方法。

方法标签[Destroy]Methods annotated with[Destroy]

对于一个直接在 parsley 配置文件内声明的对象，这些方法仅在容器被销毁并且调用 Context.destroy() 后执行。对于一个动态注入的 FLEX 组件，destroy 方法将在组件从 stage 移出后调用。

最后如果该组件的生命周期应该不依赖于它在 stage 上的时间花费，您可以切换此默认行为。你可以设置 ViewManagerFactory 属性，无论是 在 GlobalFactoryRegistry 或 CompositeContextBuilder.factories 设置上下文。对于 Flex 这也可以 配置的设置标签

```
<parsley:ContextBuilder>
            <parsley:Settings stageBoundLifecycle="false"/>
            <parsley:FlexConfig type="{MyConfig}"/>
      </parsley:ContextBuilder>
```

当组件从 stage 中移除后，以上设置使组件不会从上下文中移除。当需要不再注入时，替代这样一个组件必须派发 "removeView"事件。

最后，在单个组件上添加以下元数据：

```
<mx:Metadata>
            [RemovedEvent("removeView")]
      </mx:Metadata>
```

7.8 Flex Popups and Native AIR Windows

For Flex Popups and Native AIR Windows some extra step is required to tell the framework about them. This is because those are views which are disconnected from the view hierarchy below the main Application component. A Flex Popup usually sits somewhere right below the SystemManager. A Native AIR Window even comes with its own SystemManager. So you have to connect both manually to a Parsley ViewManager if you want to use view wiring in popups and windows. This is quite easy:

Flex Popup

```
[Inject]
public var context:Context;

private function showPopup () : void {
    var win:TitleWindow = new TitleWindow();
    // set properties
    context.viewManager.addViewRoot(win);
    PopUpManager.addPopUp(win, this);
}
```

AIR Window

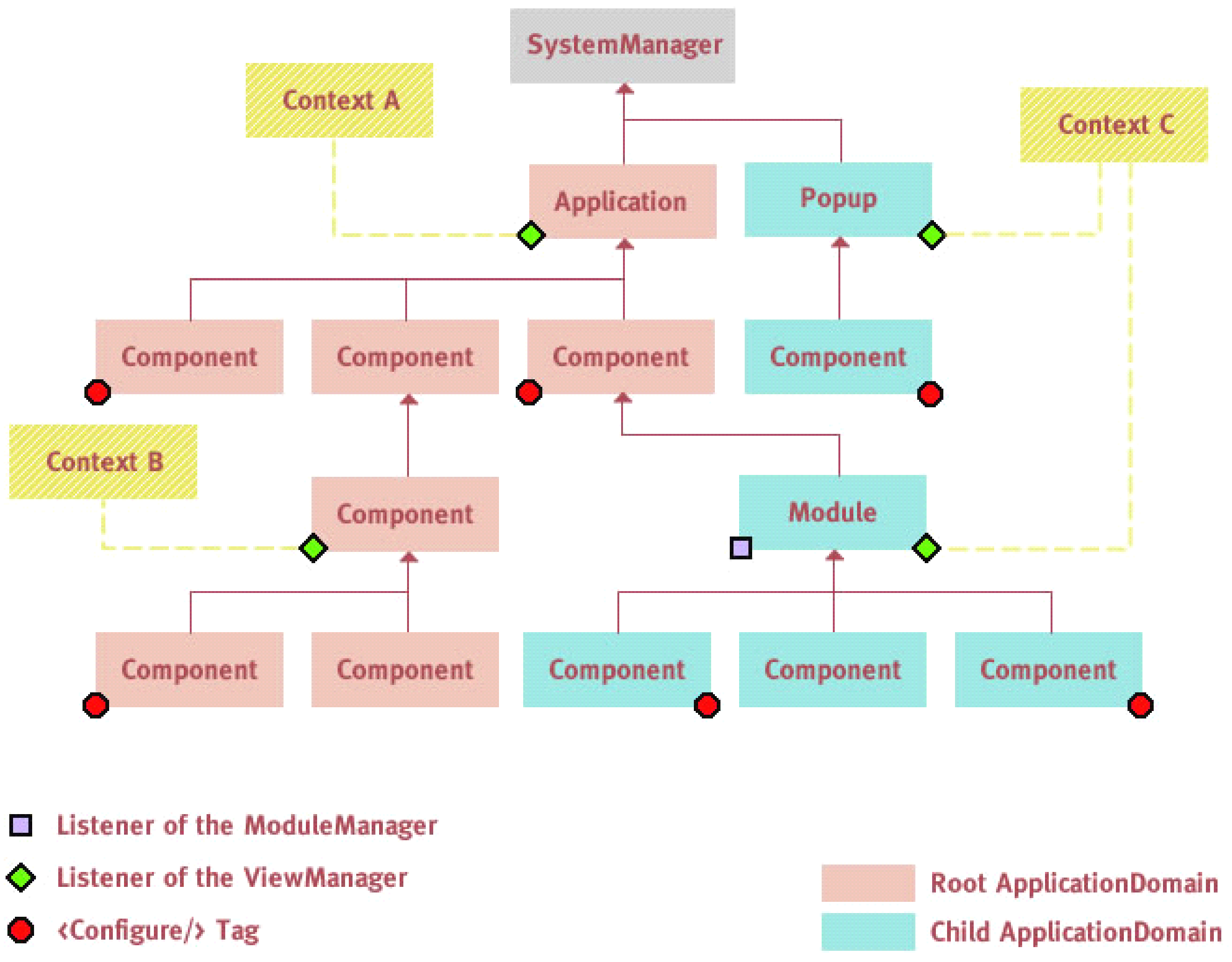
```
[Inject]
public var context:Context;

private function openWindow () : void {
    var win:Window = new Window();
    // set properties
    context.viewManager.addViewRoot(win);
    win.open();
}
```

The removeViewRoot method of the ViewManager interface will not be called by application code directly in most cases, as the default implementation of the interface will automatically remove a view root when it is removed from the stage.

7.9 在模块化系统中视图注入(View Wiring in Modular Applications)

下图展示了如何在模块化系统中进行视图注入，最重要的是如何人工连接 FLEX 的 Popups 和本地 AIR 容器到上下文。



包含<Configure/>标签的 Flex 的组件，派遣一个冒泡事件得到了一个高于该层次结构中的下一个组件 ViewManager 侦听捕 获。通过这种方式找到它的每个组件匹配上下文，这是重要的，因为一个组件加载到模块通常也需要访问该模块的配置。如果它注入到根上下文，它只会“看”上下 文中的对象。

图中的上下文 C 你会看到上下文有两个视图根：一个是加载模块的根组件，另一个是为弹出窗体打开的视图根。很明显，为什么手工连接 popup 是必要的：如果 一个组件的级别比 popup 低，派遣一个冒泡事件在 SystemManager 结束，不能达到 ViewManager 在根坐在模块组成部分，所以我们必须 设置第二个监听器。

最后，这种机制是针对应用程序领域的：parsley 的内部 ModuleManager 在根模块组件设置监听器，捕获 ContextBuilders 发出的事件，告诉应用程序如何使用。

在某些罕见情况下，您可能要加载的 Flex 模块只包含了一些注入的视图类，但没有控制器，指挥，演示或域模式。你也许会跳过创建上下文，在这种情况会导致 部分注入到根上下文。所以，你还是应该建立在模块中的根组件上下文，即使它是空的。这可以简单地透过根组件放入一个 空<parsley:ContextBuilder/>标签

8 构建模块化系统(Building Modular Applications)

Parsley 允许你建立一个可以动态加载和卸载上下文的层次结构。无论你是否使用 FLEX 模块，甚至在 AS3系统中，这个层次都可以建立。对于 Flex 模块，parsley 只是提供了一个额外的集成接口，可以更容易地处理多种不同 ApplicationDomains。

8.1 模块上下文(Modular Contexts)

对于较大的应用程序，你可能想将程序划分为模块。在这种情况下，最不幸的是，上下文作为一个整体，在应用程序启动时加载和初始化。即使分割成多个文件的配 置如3.6所示结合多种配置机制也不会有任何帮助，因为这些文件会被合并成一个单一的上下文与加载和初始化，就好像它是一个大的配置文件。

另一种功能就派上用场：当创建一个上下文，它可以连接到父上下文。父上下文可能是主上下文，用于在应用程序启动时加载。当加载一个模块时，就得创建子上下 文。子上下文的配置文件，你可以通过依赖注入使用父上下文的任何一个对象（但不反之亦然父对象的子上下文任何配置）。消息还可通过上下文边界，取决于你的 范围，通过调度。您可以创建嵌套的上下文深层次结构，但往往是结构将是相当平坦，即根上下文和任何子上下文处于同一级。

初始化子上下文可以以二种不同方式：

连接上下文树和视图树 Connecting a Context hierarchy to a view hierarchy

当您以 ContextBuilder 为切入点描述一个视图根，或者当您使用的 MXML ContextBuilder，自动使用它们放置在一个视图的根，然后 ViewManager 将上下文与视图根相关联，这里有二个目的：监听注入到上下文 组件的冒泡事件，监听来自其它 ContextBuilders 的视图低层的冒泡事件。这样，您就不必手动指定父上下文或 ApplicationDomain 中，该框架将利用这一关心你

```
<parsley:ContextBuilder config="{BookStoreConfig}"/>
```

或者:

```
var viewRoot:DisplayObject = ...;
    FlexContextBuilder.build(BookStoreConfig, viewRoot);
```

In the examples above the new Context will use the view root to automatically look for a parent Context and to listen for child Contexts within the view hierarchy.

本例中，新的上下文将视图根自动查找父上下文，并且在视图层次树中监听子上下文。

In some scenarios you may want to keep the Context hierarchy separate from the view hierarchy. This is where the second option for building the hierarchy comes into play:

在某些情况下，你可能要保持上下文层次伤残人与视图层次伤残人分开。这是采用第二个方案：

手工描述父上下文 Manually specifying the parent Context

You can define an existing Context as the parent for a new Context with an optional parameter of all the various ContextBuilder methods shown in [3 Configuration and Initialization](#):

你能定义一个存在的上下文作为一个新上下文的父类，代码如下：

```
var parent:Context = ...;
FlexContextBuilder.build(BookStoreConfig, null, parent);
```

In this case we pass null for the view root parameter and instead specify the parent manually. You must also be aware of the fact that without a view root the Context is not able to automatically detect the ApplicationDomain for reflection. So if you are not working in the root ApplicationDomain but instead in a child domain of a loaded Flex Module for example, you also have to pass the domain to the builder method:

在这种情况下，我们设置视图根参数为空，并且手工指定父类。你也必须意识到，没有视图根，上下文不能自动反射到 ApplicationDomain 中。所 以，如果你没有使用根 ApplicationDomain 中，而是在一个模块加载的 Flex 例如子域，你还必须通过域的方法

```
var parent:Context = ...;
var domain:ApplicationDomain = ...;
FlexContextBuilder.build(BookStoreConfig, null, parent, domain);
```

Again this is not necessary when specifying a view root, since the builder is then able to automatically detect the parent Context and the ApplicationDomain.

另外，没有必要去描述视图根，因为构建器能自动检测父上下文和 ApplicationDomain。

8.2 上下文生命周期(Context Lifecycle)

If you load multiple Context instances as modules like described in the previous section, you may want to get rid of them when you unload a module - without affecting the parent. Actually that is quite easy, just do this:

如果你加载多个上下文实例作为模块，当你卸载一个模块时，你可能想去掉他们--而不影响父上下文。这很容易，如下所示：

```
context.destroy();
```

When connecting a Context hierarchy to a view hierarchy it is even easier. You don't have to explicitly destroy the Context then, it will happen automatically when the last view root associated with the Context gets removed from the stage. This is the default behaviour of the ViewManager which can be adjusted in case you do not want the Context lifecycle to depend on the time the view is on the stage. You can finetune the behaviour with properties of the ViewManagerFactory.

连接上下文层次结构和视图层次结构很容易。你不用 明确销毁上下文，它会自动发生当最后一个视图根与上下文关联的获取从 stage 删除。ViewManager 默认行为是调 整上下文生命周期依赖视图的时间。你可以仔细调整 ViewManagerFactory 的属性去调整这种行为。

When a Context gets destroyed the following actions occur:

当一个上下文销毁，以下行为将出现：

- 所有在上下文中配置的对象将销毁，这将影响所有的标签 MessageHandler, MessageBinding, MessageInterceptor or MessageError
- 如果任 何一个在销毁上下文中声明的对象声明了 ManagedEvents，它们将被忽略，不再派发任何 parsley 消息。
- 所 有在销毁上下文中标识[Destroy]的方法将被调用
- 销毁的上下文将移除所有对配置对象的内部引用，让他们可以作为垃圾回收。
- 调 用 destroy 后，上下文将不再使用。任何再调用该上下文的方法将抛出错误。被销毁上下文的父上下文并不受影响，可以继续正常使用。
- All objects configured in the destroyed Context stop receiving messages dispatched by Parsleys central MessageRouter. This affects all elements annotated with MessageHandler, MessageBinding, MessageInterceptor or MessageError.
- If any objects declared in the destroyed Context declared ManagedEvents they will be ignored from now on and no longer dispatched trough Parsleys messaging system.
- All methods annotated with [Destroy] (or the corresponding MXML or XML tags) on any object of the destroyed Context get invoked.
- The destroyed Context will remove all internal references to the configured objects so they are eligible for garbage collection. (Of course you have to make sure that your application does not retain any references to those objects).
- The Context may no longer be used after invoking destroy. Any subsequent method invocations on that Context throw Errors. The parent of the destroyed Context (if any) is not affected and may continue to operate normally.

8.3 使用 FLEX 模块(Using Flex Modules)

In version 2.0.x the framework offered a special ContextModule MXML tag to specify the configuration for the module. This is no longer needed. The framework specific ContextModule and ModuleLoader tags have been removed. Instead the support for Flex Modules is now fully transparent. You load a Module either with the regular Flex ModuleLoader tag or with the Flex ModuleManager. You can then create child Contexts anywhere within that Module, it does not have to happen in the root Module component. If you connect the Context hierarchy to the view hierarchy like demonstrated in the preceding sections the child Context will automatically determine the parent Context and the ApplicationDomain of the Module.

在 version2.0.x 中，框架提供了一个特殊的 ContextModuleMXML 标签去描述模块的配置信息。这个不再需要了。框架中的 ContextModule 和 ModuleLoader 标签已经移除了。现在已完全支持 FLEX 模块 化。你可以用正常的 FLEX ModuleLoader 加载一个模块或者用 Flex ModuleManager。你既可以在模块内部创建子上下文，而不是在模块组件的根部。如果你象前面演示的那样连接上下文层次树和视图层次树，框架自动 选择父上下文和模块的 ApplicationDomain。

9 Building MVC Architectures

Parsley is different from several other Flex or Flash MVC Frameworks in that it does not provide very much explicit support for the various pieces of an MVC architecture. This was a side effect of our primary goal being to allow to design a fully decoupled architecture. Decoupled in a sense that not only the parts of your application are decoupled from each other, but also decoupled from the framework itself. For example providing base classes for controllers or mediators in the framework that application classes have to extend is a bad idea if you want to support such a decoupled architecture.

parsley 与几个其他 Flex 或 Flash 的 MVC 框架的不同，它没有提供一个非常明确的 MVC 架构。这是为了设计出完全分离的架构目标的副作用。解 耦不仅是您的应用程序部分是彼此分离，而且与框架分离。对于提供了控制器 或 mediators 的框架，如果你想支持这种分离的架构，应用程序的类必须扩展基类的例子是一个坏主意。

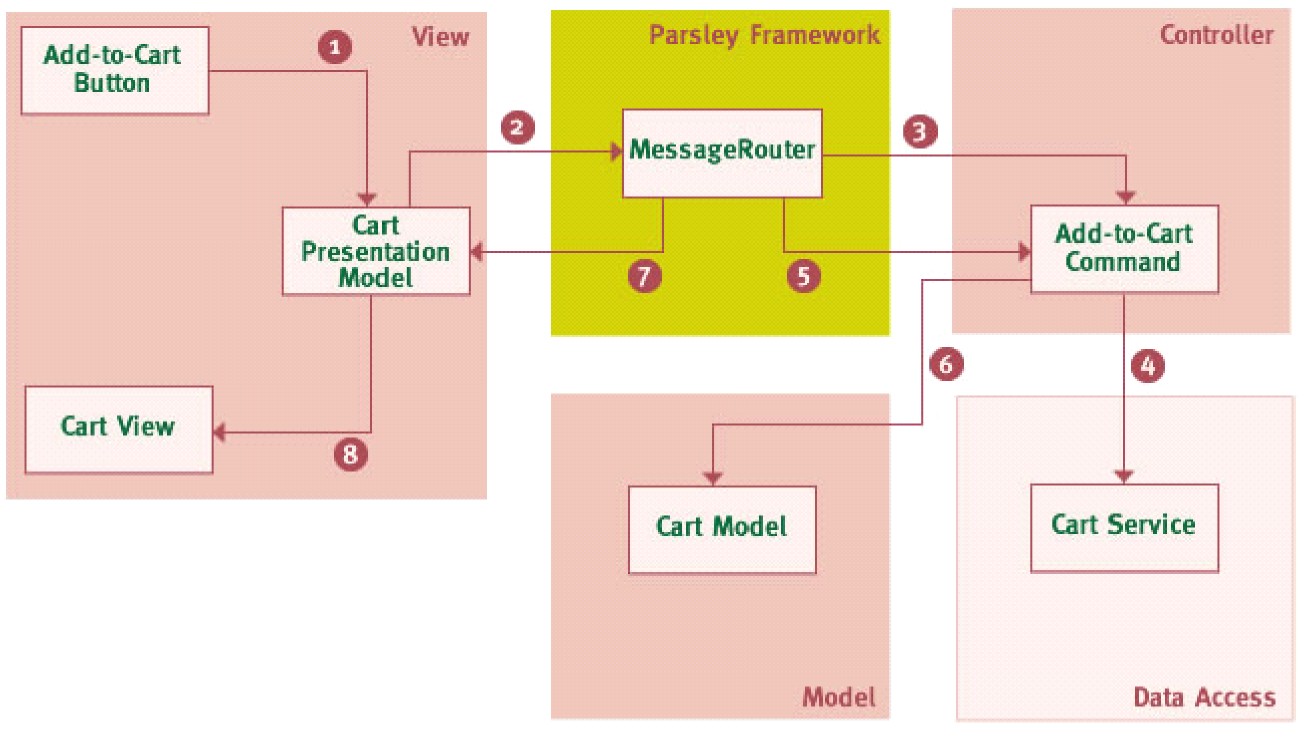
For that reason the Messaging Framework in Parsley which we already described in 5 Messaging is pretty generic. It does not assume a particular programming style or architectural pattern. Nevertheless Parsley still falls into the same category as many other MVC Frameworks since the Messaging Framework **can** easily be used in a way that helps building an application based on the MVC architectural pattern. In this chapter we will provide some examples on how to use the Messaging Framework in such a context.

出于这个原因，在 parsley 消息框架，我们已经在5 Messaging 描述过了。它并不是一个编程风格和框架模式。虽然 parsley 与许多其他 MVC 框架属于同一范畴，但消息框架可以很容易地帮助建立基于 MVC 架构模式为 基础的应用程序。在这一章中，我们将提供关于如何在这种情况下使用的消息框架的一些例子。

9.1 MVC 事件流实例(MVC Event Flow Example)

The following diagram shows an example flow for a particular use case (user adds item to shopping cart) in the context of a Parsley MVC application:

下图显示了一个在 parsley MVC 系统的特殊应用例子(添加货物到购物车内)。



Explanations for the numbered steps from the diagram:
对于图的步骤解释如下：

- 1：用户点击"Add to cart"按钮。这将产生一个低层次的 UI 事件。组件内的事件处理器调用表示层 model 实例中的一个方法(通常被注入到视图中)，用来处理视图与控制器 之间的通信。
- 2：model 产生一个系统级的消息(封装了从视图中收集的多种信息)，这个消息将通过 Parsleys 的 MessageRouter 派发；
- 3：MessageRouter 将对所有 MXML，XML 文件中的 MessageHandlers, MessageBindings, MessageInterceptors 或 Commands 进行匹配，本例中，AddToCartCommand 被调用。
- 4：命令将调用远程服务方法 (该服务将被注入)，异步操作由 parsley 框架管理。没有一个命令也没有任何应用程序部分必须明确添加结果处理器去接收 AsyncToken 返回的远 程调用结果。
- 5：框架接收了结果。首先返回命令实例，包含有结果处理器。
- 6：命令中的结果处理器在将 model 传递 给其它结果处理器之前，可以修改和缓存 model
- 7：框架调用注册的其它结果处理器，本例中 CartPM 包含一个结果处理器，通过 model 实例更新内部状态。
- 8：通过与表示层的 model 绑定，将视图进行刷新。
- 1: The user clicks the "Add to cart" button. With Flex for example this will generate a low level UI event. The event handler in the component invokes a method on the presentation model instance (usually injected into the view) which handles communication between View and Controller.
- 2: The model creates an application message possibly gathering any further information from the view and encapsulating this information in the message. This message will then be dispatched through Parsleys MessageRouter
- 3: The MessageRouter will then process any MessageHandlers, MessageBindings, MessageInterceptors or Commands which were registered for this message type (usually with Metadata tags on methods or properties, alternatively in MXML or XML). In this case the AddToCartCommand will be invoked.
- 4: The command will invoke a remote service method (the service usually being injected). The asynchronous operation will be managed by the framework which allows to avoid some of the usual plumbing code. Neither the command nor any

other part of the application has to explicitly add a result handler to the AsyncToken returned by the remote call.

- 5: The framework receives the result. It is first returned to the command instance itself if it contains a result handler. But this is purely optional.
- 6: The result handler in the command might modify or cache the model before it gets passed to other result handlers.
- 7: Next the framework will invoke any other registered result handlers in the application. In this case the CartPM contains a result handler which will now be invoked causing the model instance to update its internal state.
- 8: The view is refreshed, usually through binding to properties of the presentation model.

From the example above many core elements will have been setup and wired in a Parsley Context including the view as described in 7 Dynamic View Wiring. This usually includes all controller actions, their dependencies like remote services, and the registration for particular messages in the MessageRouter.

从本例中，一些核心元素将建立和注入到 parsley 上下文中，包括7 Dynamic View Wiring。这通常包括所有控制行为，远程服务和 MessageRouter 中的消息注册。

If you think this diagram smells like over-engineering, it merely serves as an example for a full blown MVC architecture of a large and complex application. In smaller and simpler applications you can of course simplify the design presented above. For example the Mediator may be omitted in simpler use cases where the view itself dispatches and receives application messages.

这个图仅作为大型复杂 MVC 框架的一个例子。在小型和简单的系统中，你当然可以简化上面的设计模式。例如，对于小型项目，视图派发和接收系统消息，可以忽 略 Mediator。

9.2 命令执行例子(Example Command Implementation)

Finally we'll pick one of the pieces presented in the diagram above and show how the implementation and configuration of such an application part may look like. We'll chose the AddToCartCommand that participates in the application flow shown in the diagram.

最后，我们展示系统的配置信息如何执行。我们选择 AddToCartCommand：

```
package com.bookstore.actions {

import mx.rpc.AsyncToken;
import mx.rpc.Fault;
import mx.rpc.remoting.RemoteObject;
import com.bookstore.events.CartEvent;
import com.bookstore.services.CartService;
import com.bookstore.model.LineItem;

public class AddToCartCommand {

    [Inject(id="cartService")]
    public var cartService:RemoteObject;

     public function execute (event:CartEvent) : AsyncToken {
        return cartService.addItem(event.item);
    }
    public function result (item:LineItem) : void {
        /* modify or cache the result */
    }
    public function error (fault:Fault) : void {
        /* handle fault */
    }
}

}
```

And this is how you can configure such a command in a Parsley configuration class (in this case MXML):

这里展示一个如何在 parsley 配置类(比如 MXML)中配置这个命令：

```
<DynamicCommand
        type="{AddToCartCommand}"
        selector="addItem"/>
```

We'll now examine the various parts of this class in detail. First there is the method that executes the command:

我们将详细看一下这个类的细节，首先有一个方法执行命令：

```
public function execute (event:CartEvent) : AsyncToken {
```

In this example we are using regular Flash Events as application messages, but you are not forced to do so as described in 5.1 Dispatching Messages.

这个例子使用 flash 事件作为应用程序消息，不会追使你按5.1 Dispatching Messages 的做。

The method parameter tells the framework that we are interested in CartEvents. MessageHandler selection happens based on message type, so in a large and complex application you won't run into problems having to maintain event type constants which are unique across the whole system.

这个方法的参数告诉框架，我们对 CartEvents 感兴趣。MessageHandler 的选择器可以过滤消息类型。因此， 对于大型和复杂应用程序，你不会遇到维护事件类型常量，保证惟一性的困难

But since we are not interested in all CartEvents that get dispatched we specify a selector attribute in the MXML configuration which in case of Events points to the type property of the Event. In this case we are intereseted in CartEvents with the type addItem which was dispatched by the View or the presentation model as a result of an UI event.

但是，因为我们不是对所有的 CartEvents 感兴趣，我们在 MXML 配置信息中描述了一个选择属性。本例中，我们只对 CartEvents 事件中的 addItem 感 兴趣。CartEvents 事件可以由视图派发或者是一个 UI 事件的表示层模型作为结果。

In the body of that method we are using the injected CartService:

对于方法内部，我们注入了 CartService：

```
[Inject(id="cartService")]
   public var cartService:RemoteObject;
```

In this example we are using a RemoteObject. We usually prefer to use injection by type but this is not applicable for RemoteObjects which can only be distinguished by id. For details see 12 Remoting.

本例中，我们只使用了一个远程对象。我们通常通过类型注入，通过 ID 进行注入并不是很好。详细参见12 Remoting。

Finally there are the result and error handlers:

最后，大家看一下 result 和 error 处理器：

```
public function result (item:LineItem) : void {

    public function error (fault:Fault) : void {
```

We are following a naming convention here so we can avoid any metadata configuration altogether. Based on the method names the framework knows which method executes the command and which handles results or faults. The result and error handler are both optional.

我们遵守常规命名习惯，这样可以避免任何元数据配置。框架也可以知道哪个方法执行命令，哪个处理结果或错

误。结果和错误处理器是可选的。

12 远程调用(Remoting)

Parsley is a client-side application framework and does not require any particular server-side technology to be useful. Nevertheless in this chapter we want to describe some common scenarios when integrating remoting solutions with Parsley.

Parsley 是一个客户端应用程序的框架，并不需要任何特定的服务器端技术。不过，在本章中，我们想描述一些远程调用解决方案。

For a general discussion on how to integrate services into the application architecture see 9 Building MVC Architectures.

如何集成服务到应用系统框架中参见9 Building MVC Architectures.

Of course apart from AMF based remoting solutions presented in this chapter you can also integrate HTTP Services or WebServices into your controller actions. The approach is similar to that presented in the MVC chapter: You write a controller action class that registers handlers for messages or events dispatched from the View or from Mediators, transform them into any kind of asynchronous service invocation, wait for the result, and finally dispatch an application message containing the result through Parsleys Messaging Framework.

除了基础的 AMF 的远程解决方案，在本章中您也可以融入 HTTP 服务或 WebServices 到控制器。该方法类似于在 MVC 章介绍的：你写一个控制器动 作类，用于处理从视图中派发的消息，等待结果，最后发送消息，其中包含一个 parsley 消息。

12.1 FLEX 远程服务(Flex Remoting)

For remoting the framework can help you in two respects. First you can declare your RemoteObjects in a Parsley configuration class alongside other objects and inject them into your commands. Second (since version 2.2) you can use the convenient support for asynchronous commands to also route result and faults of remote invocations in a decoupled manner.

对于远程服务，框架提供了二类帮助，首先你可以在 parsley 配置类中声明你的 RemoteObjects，并注入这些类到你的命令类中。其次，你可以 很方便地使用异步命令发送结果和错误。

Configuration

Since Parsley offers MXML-based container configuration you can easily integrate the existing MXML tags for RemoteObjects with other Parsley configuration tags:

因为 parsley 提供了 MXML 的配置信息，你可以为 RemoteObjects 简单集成现有的 MXML 标签，如下所示：

```
<mx:Object
         xmlns:mx="http://www.adobe.com/2006/mxml">

        <mx:RemoteObject
             id="loginService"
             destination="loginService"
             showBusyCursor="true"/>

        <!-- other object definitions -->

    </mx:Object>
```

You can then chose to inject those RemoteObjects into controller actions. But due to the nature of RemoteObjects not being type-safe you'd need to stick with id-based injection then:

你可以选择注入那些 RemoteObjects 到控制器的方法。但由于 RemoteObjects 不是类型案例的，你需要用 ID 注入方式。

```
        [Inject(id="loginService")]
          public var service:RemoteObject;
                [Command]
                public function login (event:LoginEvent) : AsyncToken {
                        return service.login(event.username, event.password);
          }
```

For Commands the framework will manage the AsyncToken for you. Other objects (or the same instance) can then listen for the results:

对于命令，框架负责为你管理 AsyncToken。其它对象可以监听结果：

```
[CommandResult]
        private function loginResult (user:User, trigger:LoginEvent) : void {
                [...]
        }
        [CommandError]
        private function loginFault (fault:FaultEvent, trigger:LoginEvent) : void {
                [...]
        }
```

使用商业代码(Using BusinessDelegates)

If you prefer to use delegates instead of injecting the RemoteObjects directly you could define both the RemoteObjects and the delegates in Parsley MXML:

如果你想通过代理，而不是直接注入 RemoteObjects，你可以在 Parsley MXML 中定义 RemoteObjects 和 delegates。

```
<mx:Object
     xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns="http://www.spicefactory.org/parsley">

    <mx:Script>
        <![CDATA[
            import com.bookstore.services.*;
        ]]>
    </mx:Script>

     <mx:RemoteObject
         id="loginService"
         destination="loginService"
         showBusyCursor="true"
    />

  <Object id="loginDelegate" type="{LoginDelegate}">
        <ConstructorArgs>
         <ObjectRef idRef="loginService"/>
     </ConstructorArgs>
    </Object>

     <!-- other objects -->

 </mx:Object>
```

With delegates you can then return to injection by type:

通过代理，你可以用类型注入方式：

```
[Inject]
 public var loginDelegate:LoginDelegate;

 [Command]
 public function login (event:LoginEvent) : AyncToken {
         return loginDelegate.login(event.username, event.password);
 }
```

12.2 Pimento 数据服务(Pimento Data Services)

Pimento integrates JPA/Hibernate and Spring with Flex, Flash and AIR clients. It is another Open Source Project under the Spicefactory umbrella. See the Pimento Info Page for more details.

Pimento 集成 JPA/Hibernate 和 Flex, Flash 和 AIR 的 Spring。它是另一个开源项目。参见 Pimento Info Page。

Parsley includes custom configuration tags for Pimento for MXML and XML that allow you to define the Pimento configuration and custom services.

Parsley 包含为 Pimento 提供了自定义配置标签，允许你定义 Pimento 配置信息和自定义服务。

MXML Example

```
<mx:Object
   xmlns:mx="http://www.adobe.com/2006/mxml"
 xmlns:pimento="http://www.spicefactory.org/parsley/pimento">

   <pimento:Config
      url="http://localhost:8080/test/service/"
      timeout="3000"
 />

   <!-- other objects -->

 </mx:Object>
```

This minimal setup is all that is required to be able to inject Pimentos AS3 EntityManager into any object:

这是简单的设置，将 Pimento AS3 EntityManager 注入到任何一个对象：

```
[Inject]
   public var entityManager:EntityManager;

   [Command]
   public function deleteCart (message:DeleteCartMessage) : ServiceRequest {
           return entitiyManager.remove(message.cart);
   }
```

You can additionally configure custom services with parameters and return values managed by Pimento:

你可以额外配置带参数的自定义服务，并返回由 Pimento 管理的数值。

```
 <pimento:Service
        name="loginService"
        type="{LoginServiceImpl}"/>
```

The service interfaces and remote stubs are usually generated by Pimentos Ant task. These services can then of course be injected into other objects, too.

这种服务接口和远程桩通常由 Pimentos Ant task 生成。这些服务当然可以注入到任何对象。

XML Example

```
<objects
     xmlns="http://www.spicefactory.org/parsley"
    xmlns:pimento="http://www.spicefactory.org/parsley/pimento"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.spicefactory.org/parsley
        http://www.spicefactory.org/parsley/schema/2.0/parsley-core.xsd
     http://www.spicefactory.org/parsley/pimento
        http://www.spicefactory.org/parsley/schema/2.0/parsley-pimento.xsd"
    >
<pimento:config
     url="http://localhost/test/service/"
    timeout="3000"
/>
<pimento:service
    name="loginService"
    type="com.bookstore.services.LoginServiceImpl"
/>
```

Since this is an XML extension it has to be initialized explicitly before using the XmlContextBuilder:

因为这是一个 XML 扩展，所以必须在使用 XmlContextBuilder 前进行初始化：

```
PimentoXmlSupport.initialize();
```

12.3Cinnamon 远程服务(Cinnamon Remoting)

If you don't need Pimentos data management capabilities and just want to use AMF-based Flex/Flash-to-Java Remoting you can stick with Cinnamon instead. See the Pimento/Cinnamon Info Page for more details.

如果你不想用 Pimentos 的数据管理功能，而想用基于 AMF 的 FLEX/Flash-to-java 远程服务，你可以使用 Cinnamon。参见 Pimento/Cinnamon Info Page。

Parsley includes custom configuration tags for Cinnamon for MXML and XML that allow you to define the channel and services.

Parsley 提供自定义配置标签，允许你定义管道和服务。

MXML Example

```
<mx:Object
     xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:cinnamon="http://www.spicefactory.org/parsley/cinnamon">

    <mx:Script>
        <![CDATA[
            import com.bookstore.services.*;
        ]]>
    </mx:Script>

     <cinnamon:Channel
        url="http://localhost:8080/test/service/"
        timeout="3000"
    />
    <cinnamon:Service
        name="loginService"
        type="{LoginServiceImpl}"
    />
    <cinnamon:Service
        name="cartService"
        type="{CartServiceImpl}"
    />

     <!-- other objects -->

  </mx:Object>
```

If you define only a single channel (like in most use cases) you don't have to explicitly refer to it in the service definitions. Parsley will automatically wire the single channel to all services then. In case of multiple channels you'd have to set the id property for the channel and reference it in service definitions:

如果你仅定义了一个简单的管道，你不用明确在服务定义中指定它。parsley 将自动注入简单管道到所有的服务。对于多个管道，你必须设置管理的 ID 属 性，并在服务定义中指定它。

```
<cinnamon:Channel
        id="mainChannel"
        url="http://localhost:8080/test/service/"
        timeout="3000"/>

    <cinnamon:Service
        name="loginService"
        type="{LoginServiceImpl}"
        channel="mainChannel"/>
```

You can then inject services into your actions:

你也可以注入服务到你的方法中：

```
[Inject]
    public var loginService:LoginService;

    [Command]
    public function login (event:LoginEvent) : ServiceRequest {
        return loginService.login(event.username, event.password);
    }
```

With Cinnamon there is no need for BusinessDelegates: The remote services implement business interfaces themselves, so you can directly inject them into actions. These interfaces are usually generated by Cinnamons Ant Task, automatically porting existing Java service interfaces to AS3.

通过 Cinnamon，我们就不需要 BusinessDelegates：远程服务执行商业接口。因此，你可以直接注入他们到方法上。这些接口通常由 Cinnamons Ant Task 生成，自动将存在的 JAVA 服务接口与 AS3关联。

XML Example

```
<objects
     xmlns="http://www.spicefactory.org/parsley"
    xmlns:pimento="http://www.spicefactory.org/parsley/cinnamon"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.spicefactory.org/parsley
        http://www.spicefactory.org/parsley/schema/2.0/parsley-core.xsd
     http://www.spicefactory.org/parsley/cinnamon
        http://www.spicefactory.org/parsley/schema/2.0/parsley-cinnamon.xsd"
    >
    <cinnamon:channel
        url="http://localhost:8080/test/service/"
        timeout="3000"
    />
    <cinnamon:service
        name="loginService"
        type="com.bookstore.services.LoginServiceImpl"
    />
    <cinnamon:service
        name="cartService"
        type="com.bookstore.services.CartServiceImpl"
    />
</objects>
```

Since this is an XML extension it has to be initialized explicitly before using the XmlContextBuilder:
因为这是一个 XML 扩展，所以必须初始化 XmlContextBuilder：

```
CinnamonXmlSupport.initialize();
```

14 配置标签说明(Configuration Tag Reference)

This chapter serves as a pure reference for all configuration tags, listing all their attributes. For examples or more detailed explanations we will link to the corresponding chapters.
本章只做纯配置标签说明，列出它们所有的属性。举例或详细解释则链接到相关章节。

The reference covers Metadata, MXML and XML tags. Many of the tags are available for all three configuration modes. Some only for one or two of them. The corresponding section will list which of these options are available.
说明包括 Metadata, MXML 和 XML 标签。一些标签对三种设置模式都有效。一些仅对一个或二个有效。相关章节将列出哪些设置是有效的。

14.1 依赖注入(Dependency Injection)

14.1.1 属性注入(Inject for Properties)

Metadata Tag	[Inject]
may be placed on	property declaration
MXML and XML Namespace	N/A
MXML Tag	N/A
XML Tag	N/A
Detailed Explanation	4.3 Property Injection by Typeand 4.4 Property Injection by Id
Comment	For MXML or XML see 14.7.6 Property

Attributes

id	optional	Specifies the id of the object to inject, if omitted injection by type is performed.

		描述了注入对象的 ID，如果不采用类型方式注入，则采用 ID 方式注入
required	optional	Indicates whether the dependency is required, default if omitted: true. 标 明是否需要依赖，默认为忽略:true

14.1.2 Inject for Methods

Metadata Tag	[Inject]
may be placed on	method declaration
MXML and XML Namespace	N/A
MXML Tag	N/A
XML Tag	N/A
Detailed Explanation	4.2 Method Injection

Attributes

None

14.1.3 InjectConstructor

Metadata Tag	[InjectConstructor]
may be placed on	class declaration
MXML and XML Namespace	N/A
MXML Tag	N/A
XML Tag	N/A
Detailed Explanation	4.1 Constructor Injection
Comment	For MXML or XML see 14.7.5 ConstructorArgs

Attributes

None

14.2 Messaging

14.2.1 ManagedEvents

Metadata Tag	[ManagedEvents]
may be placed on	class declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<ManagedEvents/>
XML Tag	<managed-events/>
Detailed Explanation	5.3 Managed Events

Attributes

names	required	The names (types) of the events dispatched from the annotated object that the container should manage.
scope	optional	The scope through which the event should be dispatched. If this attribute is omitted the event will be dispatched through all scopes associated with the Context the dispatching object lives in.

14.2.2 MessageDispatcher

Metadata Tag	[MessageDispatcher]
may be placed on	property declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley

MXML Tag	<MessageDispatcher/>
XML Tag	<message-dispatcher/>
Detailed Explanation	5.4 Injected MessageDispatchers
Attributes	

property	required for MXML/XML	The property the injected dispatcher function should be injected into. Will be automatically set when used as a Metadata Tag.
scope	optional	The scope through which the message should be dispatched. If this attribute is omitted the message will be dispatched through all scopes associated with the Context the dispatching object lives in.

14.2.3 MessageHandler

Metadata Tag	[MessageHandler]
may be placed on	method declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<MessageHandler/>
XML Tag	<message-handler/>
Detailed Explanation	5.5 MessageHandlers
Attributes	

type	optional	The type (class) of the message that the annotated method wishes to receive. If omitted all messages will be received (rarely useful).
messageProperties	optional	If specified not the message itself will be passed to the method, but instead the listed properties of the message as method parameters.
selector	optional	An selector value for filtering in addition to message selection by type. See 5.11 Using Selectors.
scope	optional	The scope from which the message should be received. The default is the global scope.
order	optional	The order in which handlers should be invoked. The default value is int.MAX_VALUE.
method	required for MXML/XML	The method the message instance should be passed to. Will be automatically set when used as a Metadata Tag.

14.2.4 MessageBinding

Metadata Tag	[MessageBinding]
may be placed on	property declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<MessageBinding/>
XML Tag	<message-binding/>
Detailed Explanation	5.6 MessageBindings
Attributes	

type	required	The type (class) of the message that the annotated property should be bound to.
messageProperty	required	The name of the property of the message class whose value should be bound to the targetProperty.
selector	optional	An selector value for filtering in addition to message selection by type. See 5.11 Using Selectors.
scope	optional	The scope from which the message should be received. The default is the global scope.

order	optional	The order in which bindings should be executed. The default value is int.MAX_VALUE.
targetProperty	required for MXML/XML	The name of the property that the message property value should be bound to. Will be automatically set when used as a Metadata Tag.

14.2.5 MessageInterceptor

Metadata Tag	[MessageInterceptor]
may be placed on	method declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<MessageInterceptor/>
XML Tag	<message-interceptor/>
Detailed Explanation	5.7 MessageInterceptors
Attributes	

type	optional	The type (class) of the message that the annotated method wishes to intercept. If omitted all messages will be received (rarely useful).
selector	optional	An selector value for filtering in addition to message selection by type. See 5.11 Using Selectors.
scope	optional	The scope from which the message should be received. The default is the global scope.
order	optional	The order in which interceptors should be invoked. The default value is int.MAX_VALUE.
method	required for MXML/XML	The interceptor method. Will be automatically set when used as a Metadata Tag.

14.2.6 MessageError

Metadata Tag	[MessageError]
may be placed on	method declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<MessageError/>
XML Tag	<message-error/>
Detailed Explanation	5.8 Error Handlers
Attributes	

type	optional	The type (class) of the message that was passed to the handler or interceptor that threw an Error. If omitted the error handler applies to all message types.
selector	optional	An selector value for filtering in addition to message selection by type. See 5.11 Using Selectors.
scope	optional	The scope from which the message was dispatched to the handler or interceptor that threw an Error. The default is the global scope.
order	optional	The order in which error handlers should be invoked. The default value is int.MAX_VALUE.
method	required for MXML/XML	The interceptor method. Will be automatically set when used as a Metadata Tag.

14.2.7 Command

Metadata Tag	[Command]
may be placed on	method declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley

MXML Tag	<Command/>
XML Tag	<command/>
Detailed Explanation	5.9 Asynchronous Command Methods
Attributes	

type	optional	The type (class) of the message that the annotated method wishes to receive. If omitted the type will be deduced from the method parameter.
messagePr operties	optional	If specified not the message itself will be passed to the method, but instead the listed properties of the message as method parameters.
selector	optional	An selector value for filtering in addition to message selection by type. See 5.11 Using Selectors.
scope	optional	The scope from which the message should be received. The default is the global scope.
order	optional	The order in which handlers should be invoked. The default value is int.MAX_VALUE.
method	required for MXML/XML	The method the message instance should be passed to. Will be automatically set when used as a Metadata Tag.

14.2.8 CommandResult

Metadata Tag	[CommandResult]
may be placed on	method declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<CommandResult/>
XML Tag	<command-result/>
Detailed Explanation	5.9 Asynchronous Command Methods
Attributes	

type	optional	The type (class) of the message that the annotated method wishes to receive. If omitted the type will be deduced from the second method parameter (if available).
sele ctor	optional	An selector value for filtering in addition to message selection by type. See 5.11 Using Selectors.
scop e	optional	The scope from which the message should be received. The default is the global scope.
orde r	optional	The order in which result handlers should be invoked. The default value is int.MAX_VALUE.
met hod	required for MXML/XML	The method the result should be passed to. Will be automatically set when used as a Metadata Tag.

14.2.9 CommandError

Metadata Tag	[CommandError]
may be placed on	method declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<CommandError/>
XML Tag	<command-error/>
Detailed Explanation	5.9 Asynchronous Command Methods
Attributes	

type	optional	The type (class) of the message that the annotated method wishes to receive. If omitted the type will be deduced from the second method

		parameter (if available).
selector	optional	An selector value for filtering in addition to message selection by type. See 5.11 Using Selectors.
scope	optional	The scope from which the message should be received. The default is the global scope.
order	optional	The order in which error handlers should be invoked. The default value is int.MAX_VALUE.
method	required for MXML/XML	The method the error should be passed to. Will be automatically set when used as a Metadata Tag.

14.2.10 CommandStatus

Metadata Tag	[CommandStatus]
may be placed on	Boolean property declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<CommandStatus/>
XML Tag	<command-status/>
Detailed Explanation	5.9 Asynchronous Command Methods
Attributes	

type	optional	The type (class) of the message that the annotated method wishes to receive. If omitted the type will be deduced from the second method parameter (if available).
selector	optional	An selector value for filtering in addition to message selection by type. See 5.11 Using Selectors.
scope	optional	The scope from which the message should be received. The default is the global scope.
order	optional	The order in which error handlers should be invoked. The default value is int.MAX_VALUE.
property	required for MXML/XML	The property which will act as a boolean flag for the matching command type. Will be automatically set when used as a Metadata Tag.

14.2.11 DynamicCommand

Metadata Tag	N/A
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<DynamicCommand/>
XML Tag	<dynamic-command/>
Detailed Explanation	5.10 Short-lived Command Objects
Attributes	

type	required	The type (class) of the command to create.
messageType	optional	The type (class) of the message that the command wishes to receive. If omitted the type will be deduced from the method parameter (if available).
selector	optional	An selector value for filtering in addition to message selection by type. See 5.11 Using Selectors.
scope	optional	The scope from which the message should be received. The default is the global scope.
order	optional	The order in which commands should be invoked. The default value is int.MAX_VALUE.
messageProp	optio	If specified not the message itself will be passed to the method, but instead

erties	nal	the listed properties of the message as method parameters.
stateful	optional	If false (the default) the framework will create a new instance of this command for each matching message. If true the command instance will be reused for subsequent command invocations.
execute	optional	The name of the method that executes the command. The default is execute.
result	optional	The name of the method that handles the result. The default is result.
error	optional	The name of the method that handles any errors. The default is error.

Child Elements

Any tag listed in 14.1 through 14.5, ConstructorArgs, Property, All Custom Configuration Tags that implement the ObjectDefinitionDecorator interface.

14.2.12 MessageConfirmation

Metadata Tag	N/A
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<MessageConfirmation/>
XML Tag	<message-confirmation/>
Detailed Explanation	Simple utility tag that opens an Alert for matching messages and only continues with message processing if the user clicks OK. This is a Flex-only feature.

Attributes

type	optional	The type (class) of the message that the alert should be shown for.
selector	optional	An selector value for filtering in addition to message selection by type. See 5.11 Using Selectors.
scope	optional	The scope from which the message should be received. The default is the global scope.
text	required	The text to display in the Alert.
title	required	The title to display in the Alert.

14.2.13 Selector

Metadata Tag	[Selector]
may be placed on	property declaration
MXML and XML Namespace	N/A
MXML Tag	N/A
XML Tag	N/A
Detailed Explanation	5.11 Using Selectors

Attributes

None

14.3 Object Lifecycle

14.3.1 AsyncInit

Metadata Tag	[AsyncInit]
may be placed on	class declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<AsyncInit/>
XML Tag	<async-init/>
Detailed Explanation	6.2 Asynchronous Object Initialization
Attributes	

completeEvent	optional	The event type signalling that the object is fully initialized. Default Event.COMPLETE.
errorEvent	optional	The event type signalling that the object failed to initialize correctly. Default ErrorEvent.ERROR.

14.3.2 Init

Metadata Tag	[Init]
may be placed on	method declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<Init/>
XML Tag	<init/>
Detailed Explanation	6.4 Object Lifecycle Methods
Note	The old name of this tag (PostConstruct) has been deprecated.
Attributes	

method	required for MXML/XML	The name of the method to invoke when the object is fully initialized. Will be automatically set when used as a Metadata Tag.

14.3.3 Destroy

Metadata Tag	[Destroy]
may be placed on	method declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<Destroy/>
XML Tag	<destroy/>
Detailed Explanation	6.4 Object Lifecycle Methods
Note	The old name of this tag (PreDestroy) has been deprecated.
Attributes	

method	required for MXML/XML	The name of the method to invoke when the object is fully initialized. Will be automatically set when used as a Metadata Tag.

14.3.4 Observe

Metadata Tag	[Observe]
may be placed on	method declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<Observe/>
XML Tag	<observe/>
Detailed Explanation	6.5 Lifecycle Observer Methods
Attributes	

phase	optional	The lifecycle phase to observe. The default is postInit. The other permitted values are preConfigure, preInit, preDestroy and postDestroy.
objectId	optional	The id of the object to observe. If omitted the observer method will be invoked for all matching types (polymorphically).

method	required for MXML/XML	The name of the method to invoke when the object is fully initialized. Will be automatically set when used as a Metadata Tag.

14.4 Localization

14.4.1 ResourceBinding

Metadata Tag	[ResourceBinding]
may be placed on	property declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<ResourceBinding/>
XML Tag	<resource-binding/>
Detailed Explanation	10.1 ResourceBindings

Attributes

bundle	required	The bundle name of the resource.
key	required	The resource key.
property	required for MXML/XML	The name of the property that the resource should be bound to. Will be automatically set when used as a Metadata Tag.

14.5 Extensibility

14.5.1 Factory

Metadata Tag	[Factory]
may be placed on	method declaration
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<Factory/>
XML Tag	<factory/>
Detailed Explanation	6.1 Using Factories

Attributes

method	required for MXML/XML	The name of the method that produces the obejcts. Will be automatically set when used as a Metadata Tag.

14.5.2 Target

Metadata Tag	[Target]
may be placed on	property declaration
MXML and XML Namespace	N/A
MXML Tag	N/A
XML Tag	N/A
Detailed Explanation	11.2.5 Using it as a Metadata Tagin Creating Custom Configuration Tags (final section titled "Metadata on Properties or Methods")

Attributes

None

14.6 ActionScript Configuration

14.6.1 ObjectDefinition

Metadata Tag	[ObjectDefinition]
may be placed on	property declaration
MXML and XML Namespace	N/A
MXML Tag	N/A
XML Tag	N/A
Detailed Explanation	3.4 ActionScript Configuration

Attributes

lazy	optional	When set to false (default) the object will be instantiated when the container initializes (but only if the singleton attribute keeps the default value (true). When set to true the object will be instantiated on demand.
singleton	optional	When set to true (default) the container will only create a single instance of the object, cache it internally and return the same instance upon each request. When set to false it will create a new instance upon each request.
id	optional	The id the object should be registered with. If omitted the name of the property that produces (or holds) the object will be used as the id.
order	optional	The initialization order. Only considered for non-lazy singletons, processed in ascending order. Default int.MAX_VALUE.

14.6.2 Internal

Metadata Tag	[Internal]
may be placed on	property declaration
MXML and XML Namespace	N/A
MXML Tag	N/A
XML Tag	N/A
Detailed Explanation	May be placed on properties in an AS3 Configuration Class that should not be included in the IOC Container.

Attributes

None

14.7 MXML and XML Configuration

14.7.1 Variants of Metadata Tags

This section lists tags that can only be used in MXML and XML configuration and not as Metadata in classes. Nevertheless most of the other tags documented in previous sections can be used as MXML or XML tags, too (their tag names are listed if this is the case). But for the tags in previous sections the most common use case is to use them as Metadata.

14.7.2 Object

Metadata Tag	N/A
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<Object/>
XML Tag	<object/>
Detailed Explanation	3.2 MXML Configurationand 3.3 XML Configuration Files

Attributes

type	required	The type of the object to create. 创建一个对象
lazy	option	When set to false (default) the object will be instantiated when the container

	al	initializes (but only if the singleton attribute keeps the default value (true). When set to true the object will be instantiated on demand. Can only be used in root object definitions, not in inline definitions. 当设置为 false(默认)，对象在容器初始化时创建(当 singleton 属性为默认值 true 时)。当设置为 true，在你第一次使用该对象之前，对象不会被初始化和配置。该标签仅能用于根对象的定义，不能用于内部嵌套对象定义。
singleton	optional	When set to true (default) the container will only create a single instance of the object, cache it internally and return the same instance upon each request. When set to false it will create a new instance upon each request. Can only be used in root object definitions, not in inline definitions. 当设置为 true(默认)时，容器将创建惟一的对象实例，每次 请求返回相同的实例。当设置为 false，对于每次请求，容器将创建一个新实例。这仅能用于根对象定义，不能用于内部嵌套对象定义。
id	optional	The id the object should be registered with. If omitted the name of the property that produces (or holds) the object will be used as the id. Can only be used in root object definitions, not in inline definitions. 对象最好声明 ID。 如果忽略属性的名字，对象可通过 ID 访问。这仅能用于根对象定义，不能用于内部嵌套对象定义。
order	optional	The initialization order. Only considered for non-lazy singletons, processed in ascending order. Default int.MAX_VALUE. 初始化 次序。仅应用于非 lazy singletons，按升序处理，默认为 int.MAX_VALUE

Child Elements

Any tag listed in 14.1 through 14.5, ConstructorArgs, Property, All Custom Configuration Tags that implement the ObjectDefinitionDecorator interface.

14.7.3 NestedObject

Metadata Tag	N/A
MXML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<NestedObject/>
XML Tag	N/A (simply use <object>)
Detailed Explanation	Needed in MXML for inline object definitions.

Attributes

type	required	The type of the object to create.

Child Elements

Any tag listed in 14.1 through 14.5, ConstructorArgs, Property, All Custom Configuration Tags that implement the ObjectDefinitionDecorator interface.

14.7.4 View

Metadata Tag	N/A
MXML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<View/>
XML Tag	<view>
Detailed Explanation	7.6 MXML and XML Configuration

Attributes

type	required	The type of the view that should be configured.

id	optional	The id of the view that should be configured.

Child Elements

Any tag listed in 14.1 through 14.5, all Custom Configuration Tags that implement the ObjectDefinitionDecorator interface except those that deal with object creation as the View tag configures objects which already have been created by the Flex framework.

14.7.5 ConstructorArgs

Metadata Tag	N/A
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<ConstructorArgs/>
XML Tag	<constructor-args/>
Description	May be used to explicitly specify the constructor arguments in MXML or XML as a list of child tags

Attributes

None

Child Elements

Any tag listed in 14.7.10 Simple Values, Object, Array, StaticPropertyRef or any tag that is mapped to an object or an ObjectDefinitionFactory.

14.7.6 Property

Metadata Tag	N/A
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<Property/>
XML Tag	<property/>
Description	May be used to explicitly specify a property value in MXML or XML. The value may be a simple value or a dependency injected by type or by id.

Attributes

name	required	The name of the property.
value	optional	May be used for simple values (instead of a nested child node specifying the value).
idRef	optional	Specifies a dependency for this property, referring to another object in the container by id.
typeRef	optional	Specifies a dependency for this property, referring to another object in the container by type.
required	optional	Indicates whether the dependency is required (only applied if idRef or typeRef is set).

The attributes value, idRef and typeRef are mutually exclusive.

Child Elements

A single Child Elements may be used to specify the value for the property if neither value, idRef nor typeRef have been set as an attribute. Allowed are the same tags that are allowed as children of the ConstructorArgs tag: Any tag listed in 14.7.10 Simple Values, Object, Array, StaticPropertyRef or any tag that is mapped to an object or an ObjectDefinitionFactory.

14.7.7 ObjectRef

Metadata Tag	N/A
MXML and XML Namespace	http://www.spicefactory.org/parsley

MXML Tag	<ObjectRef/>
XML Tag	<object-ref/>
Description	May be used to specify a reference to another object in the container by id or by type.

Attributes

idRef	optional	Specifies a dependency for this property, referring to another object in the container by id.
typeRef	optional	Specifies a dependency for this property, referring to another object in the container by type.
required	optional	Indicates whether the dependency is required (only applied if idRef or typeRef is set).

The attributes idRef and typeRef are mutually exclusive.

Child Elements

None.

14.7.8 Array

Metadata Tag	N/A
MXML and XML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<Array/>
XML Tag	<array/>
Description	May be used to specify Array values in MXML or XML, specifying the elements as a list of child tags. Should be preferred over mx:Array if it contains special Parsley tags as children (they won't be processed within a regular mx:Array tag).

Attributes

None

Child Elements

Any tag listed in 14.7.10 Simple Values, Object, Array, StaticPropertyRef or any tag that is mapped to an object or an ObjectDefinitionFactory.

14.7.9 StaticPropertyRef

Metadata Tag	N/A
MXML Tag	N/A
XML Namespace	http://www.spicefactory.org/parsley
XML Tag	<static-property-ref/>
Description	May be used to refer to a static property in XML configuration files. Not necessary in MXML configuration, use normal binding syntax in MXML (e.g. value="{Environment.CONSTANT}").

Attributes

type	required	The class the static property value should be fetched from.
property	required	The name of the static property.

Child Elements

None

14.7.10 Simple Values

For MXML configuration you can use the builtin tags to specify simple values like mx:String or mx:int. For XML configuration Parsley includes a number of tags that can be used for simple values:

- string
- boolean
- int
- uint
- number
- date
- class
- null

Example:

```
<object type="com.example.FooManager">
    <constructor-args>
        <string>Hello</string>
        <int>7</int>
        <boolean>true</boolean>
        <object-ref id-ref="barManager"/>
    </constructor-args>
</object>
```

14.7.11 Include

Metadata Tag	N/A
MXML Tag	N/A
XML Namespace	http://www.spicefactory.org/parsley
XML Tag	<include/>
Description	May be used to include additional XML configuration files. You can place any number of include tags in XML configuration files.

Attributes

filename	required	The name of the file to include.

Child Elements

None

14.7.12 Variable

Metadata Tag	N/A
MXML Tag	N/A
XML Namespace	http://www.spicefactory.org/parsley
XML Tag	<variable/>
Description	May be used to declare variables which may be used in XML attributes or text nodes with the notation ${variableName}.

Attributes

name	required	The name of the variable.
value	required	The value of the variable.

Child Elements

None

14.8 Flex Component Wiring

14.8.1 Configure

Metadata Tag	N/A
MXML Namespace	http://www.spicefactory.org/parsley
MXML Tag	<Configure/>
XML Tag	N/A
Detailed Explanation	7.2 Explicit Component Wiring

Attributes

target	optional	The object that should be wired to the Context. If omitted the document object the tag was placed in will be wired.
repeat	optional	Indicates whether the wiring should happen repeatedly whenever the object is added to the stage. When set to false it is only wired once. The default is true.

14.9 Flex Logging

14.9.1 Target

Metadata Tag	N/A
MXML Tag	N/A (the builtin MXML tags for LogTargets can be used instead)
XML Namespace	http://www.spicefactory.org/parsley/flex/logging
XML Tag	<target/>
Required Initialization	FlexLoggingXmlSupport.initialize()
Detailed Explanation	13.1 Logging Configuration for Flex

Attributes

id	optional	The id of the LogTarget, usually not needed.
type	optional	The type (class) of the LogTarget, defaults to TraceTarget if omitted.
level	optional	The minimum level that this target should log.
attributes of LineFormattedTarget	optional	If the specified type extends LineFormattedTarget all properties of that class can be set: field-separator, include-category, include-level, include-date, include-time.

14.10 Flash Logging

14.10.1 LogFactory

Metadata Tag	N/A
MXML Tag	N/A
XML Namespace	http://www.spicefactory.org/parsley/flash/logging
XML Tag	<factory/>
Required Initialization	FlashLoggingXmlSupport.initialize()
Detailed Explanation	13.2 Logging Configuration for Flash

Attributes

id	optional	The id of the LogFactory, usually not needed.
type	optional	The type (class) of the LogFactory, defaults to DefaultLogFactory which is usually sufficient.
context	optional	Boolean attribute (defaults to true). Indicates whether this factory should be set as the factory for the LogContext class. Should only be set to false for special requirements.

root-level	optional	The default log level for loggers for which no level has been specified explicitly. Defaults to LogLevel.TRACE.
Child Elements		
appender	optional	One or more Appenders that handle the log output. Usually at least one is required to see any output.
logger	optional	Configuration for individual loggers. Only required for loggers which should not use the root-level.

14.10.2 Logger

Metadata Tag	N/A
MXML Tag	N/A
XML Namespace	http://www.spicefactory.org/parsley/flash/logging
XML Tag	<logger/>
Required Initialization	FlashLoggingXmlSupport.initialize()
Detailed Explanation	13.2 Logging Configuration for Flash

Attributes		
name	required	The name of the Logger.
level	required	The minum level this Logger should log. Valid values are trace, debug, info, warn, error and fatal.
Child Elements		
None		

14.10.3 Appender

Metadata Tag	N/A
MXML Tag	N/A
XML Namespace	http://www.spicefactory.org/parsley/flash/logging
XML Tag	<appender/>
Required Initialization	FlashLoggingXmlSupport.initialize()
Detailed Explanation	13.2 Logging Configuration for Flash

Attributes		
ref	required	The id of the Appender class configured in the Parsley Context. For configuring the Appender itself you can use regular Parsley <object> tags.
threshold	required	The minum level this Appender should log. May be used as an additional filter mechanism in addition to the level settings for individual loggers. Valid values are trace, debug, info, warn, error and fatal.
Child Elements		
None		

14.11 Flash Localization

14.11.1 ResourceManager

Metadata Tag	N/A
MXML Tag	N/A
XML Namespace	http://www.spicefactory.org/parsley/flash/resources
XML Tag	<resource-manager/>
Required Initialization	FlashResourceXmlSupport.initialize()
Detailed Explanation	10.3 Localized Flash Applications

Attributes

id	optional	The id of the ResourceManager, usually not needed as you should only configure one ResourceManager instance and thus are able to inject it by type.
type	optional	The type (class) of the ResourceManager, defaults to DefaultResourceManager which is usually sufficient.
cacheable	optional	Boolean attribute (defaults to false). Indicates whether the ResourceManager should cache loaded bundles.
persistent	optional	Boolean attribute (defaults to false). Indicates whether the ResourceManager should store the last active locale in a Local Shared Object and restore it on the next application start.

Child Elements

locale	optional	The locales supported by this ResourceManager.

14.11.2 Locale

Metadata Tag	N/A
MXML Tag	N/A
XML Namespace	http://www.spicefactory.org/parsley/flash/resources
XML Tag	<locale/>
Required Initialization	FlashResourceXmlSupport.initialize()
Detailed Explanation	10.3 Localized Flash Applications

Attributes

language	optional	The language code for this locale, a lowercase ISO 639 code (e.g. en or fr).
country	optional	The country/region code for this locale, an uppercase ISO 3166 2-letter code (e.g. US or FR).

Child Elements

None

14.11.3 ResourceBundle

Metadata Tag	N/A
MXML Tag	N/A
XML Namespace	http://www.spicefactory.org/parsley/flash/resources
XML Tag	<resource-bundle/>
Required Initialization	FlashResourceXmlSupport.initialize()
Detailed Explanation	10.3 Localized Flash Applications

Attributes

id	required	The id of the bundle.
basename	required	The basename of files containing messages for this bundle. For the locale en_US for example the basename messages/tooltips will instruct Parsley to load the following files: messages/tooltips_en_US.xml, messages/tooltips_en.xml and messages/tooltips.xml.
type	optional	The type (class) of the ResourceBundle, defaults to DefaultResourceBundle which is usually sufficient.
loaderFactory	optional	The type (class) of the BundleLoaderFactory, responsible for loading the bundle files. Defaults to DefaultResourceBundle which is usually sufficient.
localized	optional	Boolean attribute (defaults to true). If set to false the framework will only load the resources for the basename like messages/tooltips.xml and not look for files with localized messages.

ignore-country	optional	Boolean attribute (defaults to false). If set to true the framework will ignore the country code of the active locale for this bundle.

Child Elements

None

14.12 Pimento Data Services

14.12.1 Config

Metadata Tag	N/A
MXML and XML Namespace	http://www.spicefactory.org/parsley/pimento
MXML Tag	<Config/>
XML Tag	<config/>
Required Initialization for XML	PimentoXmlSupport.initialize()
Detailed Explanation	12.2 Pimento Data Services

Attributes

id	optional	The id of the configuration instance.
url	required	The URL that Pimento should connect to.
timeout	optional	Specifies the request timeout in milliseconds.

Child Elements

None

14.12.2 Service

Metadata Tag	N/A
MXML and XML Namespace	http://www.spicefactory.org/parsley/pimento
MXML Tag	<Service/>
XML Tag	<service/>
Required Initialization for XML	PimentoXmlSupport.initialize()
Detailed Explanation	12.2 Pimento Data Services

Attributes

id	optional	The id that this service will be registered with in the Parsley IOC Container.
name	required	The name of the service as registered on the server-side.
type	required	The AS3 service implementation (usually generated with Pimentos Ant Task).
config	optional	The id of the PimentoConfig instance to use for this service. Only required if you have more than one config tag in your Context. If there is only one (like in most use cases) it will be automatically detected.
timeout	optional	Specifies the request timeout in milliseconds.

14.13 Cinnamon Remoting

14.13.1 Channel

Metadata Tag	N/A
MXML and XML Namespace	http://www.spicefactory.org/parsley/cinnamon
MXML Tag	<Channel/>
XML Tag	<channel/>

Required Initialization for XML	CinnamonXmlSupport.initialize()
Detailed Explanation	12.3 Cinnamon Remoting

Attributes

id	optional	The id of the channel instance.
url	required	The URL that the channel should connect to.
type	optional	The type (class) of the channel, defaults to NetConnectionServiceChannel.
timeout	optional	Specifies the request timeout in milliseconds.

Child Elements

None

14.13.2 Service

Metadata Tag	N/A
MXML and XML Namespace	http://www.spicefactory.org/parsley/cinnamon
MXML Tag	<Service/>
XML Tag	<service/>
Required Initialization for XML	CinnamonXmlSupport.initialize()
Detailed Explanation	12.3 Cinnamon Remoting

Attributes

id	optional	The id that this service will be registered with in the Parsley IOC Container.
name	required	The name of the service as registered on the server-side.
type	required	The AS3 service implementation (usually generated with Pimentos Ant Task).
channel	optional	Reference to the id of the ServiceChannel instance. Only required if you have more than one channel tag in your Context. If there is only one (like in most use cases) it will be automatically detected.
timeout	optional	Specifies the request timeout in milliseconds.