Documentation for Robotlegs v1.0RC1

目录

Robotlegs 是什么

Robotlegs 是一个用来开发Flash, Flex, 和 AIR 应用的纯 AS3 微架构(框架). Robotlegs 专注于将应用程序各层排布在一起并提供它们相互通讯的机制. Robotlegs 试图通过提供一种解决常见开发问题的经过时间检验的架构解决方案来加速开发. Robotlegs 无意锁定你到框架, 你的类就是你的类的样子, 而且应该很容易地切换到其他框架.

框架提供一个基于 Model-View-Controller 元设计模式的默认实现. 这个实现提供一个针对应用程序结构和设计的强烈建议. 虽然它确实轻微减低了你的应用程序的便携性, 不过它依然以最低限度影响你的具体类为目标. 通过扩展 MVCS 实现类, 你可以获得很多有用的方法和属性.

你不必使用Robotlegs的标准 MVCS 实现.你可以使用它的任意部分, 或者完全不使用它, 或者使用自己的实现来适应你的需求. 它是为了提供合适的参考实现和快速开始使用 Robotlegs 而被包含进来。

依赖注入

Robotlegs 围绕 依赖注入 设计模式展开.

> 最简单地, 依赖注入是为对象提供实例变量或属性的行为. 当你传递一个变量到一个类的构造函数, 你在使用依赖注入. 当你设置一个类的属性, 你在使用依赖注入. 如果你不是使用严格的过程或线性方式编写AS3, 很可能你现在就在使用依赖注入。

Robotlegs 使用基于元数据的自动依赖注入. 这是为了方便开发而提供, 而且在排布应用程序并提供类和它所需要的依赖时，可以减少很多代码量. 虽然完全可以手动提供这些依赖, 但是允许框架来履行这些职责可以减少出错的机会，并且通常可以加快编码进程。.

使用 **Injectors**

Robotlegs 采用一种适配器(adapter)机制来为框架提供依赖注入机制. 默认地, 框架配备了 SwiftSuspenders 注入/反射库来适合这个要求. 另有 SmartyPants-IoC 和 Spring Actionscript 的适配器可以使用. 可能有潜在的特定需求来使用其它的依赖注入适配器, 但是如果没有特别的理由, 建议使用默认的SwiftSuspenders, 因为它为 Robotlegs 做了一些特别调整.

SwiftSuspenders 适配器注入语法

SwiftSuspenders 支持三种类型的依赖注入

- 属性(域)注入
- 参数(方法/设值) 注入
- 构造注入

鉴于此文档的目的, 我们将特别介绍属性注入, 以及在 Robotlegs 里如何使用. 将属性注入类有两种选择. 你可以使用未命名，或命名的注入:

```
[Inject]
public var myDependency:Depedency; //未命名注入
```

```
[Inject(name="myNamedDependency")]
public var myNamedDependency:NamedDepedency; //命名注入
```

Robotlegs 里三处提供了注入映射. MediatorMap, CommandMap, 和直接通过 Injector . MediatorMap 和 CommandMap 也都是使用 Injector, 但它们同时做了一些各自层(tier)所需要的额外工作. 顾名思义, MediatorMap 用来映射 Mediator, CommandMap 用来映射 Command, 其它所有需要被注入的内容 (包括但不限于 Model) 都要直接使用 Injector 映射.

Injector 类的映射注入

具体 Injector 类的适配器都遵照 IInjector 接口. 这个接口为不同的依赖注入解决方案提供了统一的API. 本文档专注于 SwiftSuspenders, 但这些语法同样适应于其它任何遵照 Iinjector 接口的 Injector.

injector 是你应用程序里所发生的所有依赖注入的生产车间. 它用来注入框架 actor, 同时也可以用来执行你的应用程序所需要的任何其它注入. 这包括但不限于 RemoteObjects, HTTPServices, 工厂类, 或者事实上任何有可能成为你的对象所需要的依赖的类/接口.

下面是实现 IInjector 接口的类所提供的四个映射方法:

mapValue

mapValue 用来映射一个对象的特定实例到一个 injector. 当请求一个特定的类，使用类的这个特定实例来注入.

```
//你的应用程序中某个映射/配置发生的地方
var myClassInstance:MyClass = new MyClass();
injector.mapValue(MyClass, myClassInstance);
```

```
//在接收注入的类中
[Inject]
public var myClassInstance:MyClass
```

```
mapValue(whenAskedFor:Class, instantiateClass:Class, named:String = null)
```

MyClass 的实例被创建和保留，并在被请求的时候被注入. 当此类被请求的时候，这个实例被用来满足这个注入请求. 请注意很重要的一点，因为你已经手动创建并通过 mapValue 映射了一个类实例, 这个实例所需要的依赖将不会被自动注入. 你需要手动或通过 injector 注入这些依赖:

```
injector.injectInto(myClassInstance);
```

这将立即为此实例提供已映射的可以注入的属性。.

mapClass

mapClass 为每一个注入请求提供这个被映射的类的一个 特有(*unique*) 实例.

```
//你的应用程序中某个映射/配置发生的地方
injector.mapClass(MyClass);
```

```
//在第一个接收注入的类里
[Inject]
public var myClassInstance:MyClass
```

```
//在第二个接收注入的类里
[Inject]
public var myClassInstance:MyClass
```

为上面的每一个注入提供MyClass的 特有(*unique*) 实例来完成请求.

```
mapClass(whenAskedFor:Class, named:String = null)
```

injector 提供了一个方法来实例化被映射的对象:

```
injector.mapClass(MyClass);
var myClassInstance:MyClass = injector.instantiate(MyClass);
```

这提供你的对象的一个实例，并填充了此对象包含的所有被映射的注入点(injection points).

mapSingleton

mapSingleton 为每一个注入请求提供类的一个 单一_(*_single*) 实例. 为所有映射提供类的单一实例确保你维护一个一致的状态并且不用担心创建被此类的多余实例. 这是一个被框架强制和管理的单一实例, 而不是一个在类内部强制的单例(Singleton).

```
//你的应用程序中某个映射/配置发生的地方
injector.mapSingleton(MyClass);
```

```
//在第一个接收注入的类里
[Inject]
public var myClassInstance:MyClass
```

```
//在第二个接收注入的类里
[Inject]
public var myClassInstance:MyClass
```

在上面的例子里, 两个注入请求都将由被请求类的相同实例填充. 这个注入是被延迟的, 即对象直到第一次被请求才被实例化.

```
mapSingletonOf(whenAskedFor:Class, useSingletonOf:Class, named:String = null)
```

mapSingletonOf

mapSingletonOf在功能上非常像 mapSingleton. 它对映射抽象类和接口很有用, 而 mapSingleton 用来映射具体类实现.

```
//你的应用程序中某个映射/配置发生的地方  injector.mapSingletonOf(IMyClass, MyClass); //MyClass
implements IMyClass
```

```
//在第一个接收注入的类里  [Inject]  public var myClassInstance:IMyClass
```

```
//在第二个接收注入的类里  [Inject]  public var myClassInstance:IMyClass
```

这个注入方法对创建使用多态的更具可测性的类非常有用. 在下面的 Service 实例 章节可以找到一个例子.

MediatorMap 类的依赖注入

MediatorMap 实现 IMediatorMap 接口. IMediatorMap 提供两个方法来将你的 mediators 映射到 view 并注册它们以便用来注入.

```
mapView(viewClassOrName:*, mediatorClass:Class, injectViewAs:Class = null, autoCreate:Boolean
= true, autoRemove:Boolean = true):void
```

mapView 接受一个视图类, MyAwesomeWidget, 或者一个视图的类全名, *com.me.app.view.components::MyAwesomeWidget* 作为第一个参数. 第二个参数是将要作为视图组件中介的 Mediator 类. **[injectViewAs** 内容未完成**]**, 最后的两个参数 autoCreate 和 autoRemove 提供方便的自动管理 mediator 的布尔值开关.

```
//在你的程序里某个映射/配置发生的地方
mediatorMap.mapView(MyAwesomeWidget, MyAwesomeWidgetMediator);
```

```
//在conntextView 的显示列表里的某个地方
var myAwesomeWidget:MyAwesomeWidget = new MyAwesomeWidget();
this.addChild(myAwesomeWidget); // ADDED_TO_STAGE 事件被抛出, 触发这个视图组件的中介机制
```

这个方法使用了自动中介机制. 手动中介，以及对此方面更深入的内容将稍后在 Mediators 章节中介绍.

CommandMap 类的依赖注入

CommandMap 类实现 ICommandMap 接口, 提供一个用来将 command 映射到到触发它们的框架事件的方法.

```
mapEvent(eventType:String, commandClass:Class, eventClass:Class = null, oneshot:Boolean =
false)
```

你要提供给 commandMap 一个可以执行的类, 执行它的事件类型, 可选的这个事件的强类型, 以及这个 command 是否只被执行一次并且随即取消映射的布尔值开关.

这个可选的强类型事件类用来对Flash平台的"magic string"事件类型系统做额外的保护.以避免采用相同事件类型字符串的不同事件类之间的冲突.

```
//在你的程序里某个映射/配置发生的地方
commandMap.mapEvent(MyAppDataEvent.DATA_WAS_RECEIVED, MyCoolCommand, MyAppDataEvent);
```

```
//在事件被广播的另外一个框架actor里
//这触发了随后被执行的被映射的 command
dispatch(new MyAppDataEvent(MyAppDataEvent.DATA_WAS_RECEIVED, someTypedPayload))
```

The Context

Context 是所有 Robotlegs 具体实现的中心. 一个 Context, 或者也许多个 Context, 提供其它层进行通讯的机制. 一个应用程序并非只能使用一个 Context, 但大多情况下一个 Context 就足够了. 如果在 Flash 平台上创建模块化应用程序, 多个 Context 就是必须的了. Context 在一个应用程序里有三个功能: 提供初始化,**[de-initialization –** 非初始化**?]**, 和用来通讯的事件中心bus.

```
package org.robotlegs.examples.bootstrap
```

```
{
        import flash.display.DisplayObjectContainer;

        import org.robotlegs.base.ContextEvent;
        import org.robotlegs.core.IContext;
        import org.robotlegs.mvcs.Context;

        public class ExampleContext extends Context implements IContext
        {
                public function UnionChatContext(contextView:DisplayObjectContainer)
                {
                        super(contextView);
                }

                override public function startup():void
                {
                        //这个 Context 只映射一个 command 到 ContextEvent.STARTUP 事件.
                        //这个 StartupCommand 将映射其它将在应用程序里使用的的 command,
                        //mediator, service, 和 model.
                        commandMap.mapEvent( ContextEvent.STARTUP, StartupCommand,
ContextEvent, true );

                        //启动应用程序 (触发 StartupCommand)
                        dispatch(new ContextEvent(ContextEvent.STARTUP));
                }
        }
}
```

MVCS 参考实现

Robotlegs 装备了一个参考实现. 这个实现遵照经典的 Model-View-Controller (MVC) 元设计模式, 另外增加了第四个叫做 Service 的 actor. 这些层在本文档中通称为"核心 actor", 或者简称为"actor".

MVCS 提供一个应用程序的框架概况. 通过将几个经过时间检验的设计模式整合到一个具体实现, Robotlegs 的 MVCS 实现可以用做创建你的应用程序的一致方案. 通过这些架构概念着手一个应用程序, 你甚至可以在开始你的设计之前避免很多常见的障碍:

- 分离
- 组织
- 解耦

分离

MVCS 提供一种将你的应用程序分离到提供特定功能的无关联的层的很自然的方法. view 层处理用户交互. model 层处理用户创建的或从外部获取的数据. controller 提供一种封装各层之间复杂交互的机制. 最后, service 层提供一种和外界(比如远程服务 API 或文件系统)交互的独立机制.

组织

通过这种分离我们自然获得一个组织水平. 每个项目都需要某个组织水平. 是的, 有人可以把他们所有的类都扔到顶级包下完事, 但即使是最小的项目这也是不可接受的. 当一个项目有了一定的规模就需要开始组织类文件的结构了. 当向同一个应用程序开发中增加团队成员的时候问题就更加严重了. RobotLegs 的 MVCS 实现为项目描绘出一个分为四层的优雅的组织结构.

解耦

Robotlegs 的MVCS实现将应用程序解耦为4层. 每层都与其它层隔离, 使分离类和组件分别测试变得非常容易. 除了简化测试进程, 通常也使类更具便携性以在其它项目中使用. 比如, 一个连接到远程 API 的 Service 类可能在多个项目中都很有用. 通过解耦这个类, 它可以不需重构便从一个项目转移到另一个中使用.

这个默认实现只是充作最佳实践建议的一个例子. Robotlegs 并不打算以任何方式束缚你到这个例子, 它只

是一个建议. 你可以随意开发自己的实现来适应你喜欢的命名规范和开发需求. 如果这正是你所追求的, 请一定告诉我们, 因为我们一直对新方案很感兴趣, 而且它有可能作为一种替代实现包含到 RobotLegs 的代码仓库中.

Context

像 RobotLegs 中的其它实现一样, MVCS 实现也是围绕一个或多个 Context. 这个 context 提供一个中心的事件 bus 并且处理自己的启动和关闭. 一个 context 定义了一个范围. 框架 actor 们处在 context 之内, 并且在 context 定义的范围之内进行相互间的通讯. 一个应用程序是可以有多个 context 的. 这对想要加载外部模块的应用程序很有用. 因为在一个 context 里的 actor 只能在他们的 context 定义的范围之内相互通讯, 所以在一个模块化的应用程序里, 不同 context 之间的通讯是完全可能的.

本文档不讨论模块化编程的内容. 之后本文档内所有提到的 Context 都指在一个应用程序里的单一的 context.

Controller & Commands

Controller 层由 Command 类体现. Command 是用来执行应用程序单一单位工作的, 无状态的, 短生命周期的对象. Command 用于应用程序各层之间相互通讯, 也可能用来发送系统事件. 这些系统事件既可能发动其它的 Command, 也可能被一个 Mediator 接收,然后对一个 View Component 进行对应这个事件的工作. Command 是封装你的应用程序业务逻辑的绝佳场所.

View & Mediators

View 由 Mediator 类体现. 继承 Mediator 的类用来处理框架和 View Component 之间的交互. 一个 Mediator 将会监听框架事件和 View Component 事件, 并在处理所负责的 View Component 发出的事件时发送框架事件. 这样开发者可以将应用程序特有的逻辑放到 Mediator, 而避免把 View Component 耦合到特定的应用程序.

Model, Service and the Actor

MVCS 架构里的 service 和 model 在概念上有着非常多的相似之处. 因为这种相似性, model 和 service 继承了同样的 Actor 基类. 继承 Actor 基类可以获得很多应用程序架构内的功能. 在 MVCS 的 context 里, 我们通过利用继承 Actor 基类来定义应用程序所需要用来管理数据以及和外界通讯的 model 和 service 类. 本文档将把 model 和 service 类分别叫做 Model 和 Service.

澄清一点, 本文档把体现应用程序四个层的所有类都称为"framework actor"或"actor". 请不要和本例中包含的只被 Model 和 Service 类继承的 MVCS 类 Actor 混淆.

Model

Model 类用来在 model 层对数据进行封装并为其提供 API. Model 会在对数据模型进行某些工作之后发出事件通知. Model 通常具有极高的便携性.

Service

一个 service 层的 Service 用来和"外面的世界"进行通讯. Web service, 文件存取, 或者其它任何应用程序范围之外的行为对 service 类都很适合. Service 类在处理外部事件时会广播系统事件. 一个 service 应该封装和外部服务的交互且具有非常高的便携性.

框架事件

Robotlegs 使用Flash的原生事件用于框架 actor 之间的通讯. 自定义事件类通常用于此用途, 虽然使用现有的 Flash 事件同样可行. Robotlegs 不支持事件冒泡, 因为它并不依赖 Flash 显示列表作为 event bus. 使用自定义类允许开发者通过给事件添加属性来为框架 actor 之间通讯所用的系统事件提供强类型的负载.

所有的框架 actor 都可以发送事件: Mediator, Service, Model, 和 Command. Mediator 是唯一接收框架事件的actor. Command 是在对框架事件的处理中被触发. 一个事件既可以被一个 Mediator 接收, 也可以触发一个 command.

model 和 service 不应该监听和处理事件. 这样做会把它们紧耦合到应用程序特有逻辑而降低潜在的便携性和复用性.

Command

Command 是短生命周期的无状态对象. 它们在被实例化和执行之后立即释放. Command 应该只在处理框架事件时被执行, 而不应该被任何其他框架 actor 实例化或执行.

Command 职责

Command 被 Context 的 CommandMap 注册到 Context. CommandMap 在 Context 和 Command 类里默认可用. Command 类被注册到 Context 时接收4个参数: 一个事件类型; 响应这个事件时执行的 Command 类; 可选的事件类; 一个是否该 Command 只被执行一次随即被取消注册而不响应后续事件触发的一次性设置.

触发 **Command**

Command 被 Mediators, Services, Models, 和其它 Command 广播的框架事件触发. 典型的, 触发这个 Command 的事件会被注入到这个 Command, 以提供对其属性/负载的访问:

```
public class MyCommand extends Command
{
        [Inject]
        public var event:MyCustomEvent;

        [Inject]
        public var model:MyModel;

        override public function execute():void
        {
                model.updateData( event.myCustomEventPayload )
        }
}
```

一个被映射的 command 在响应一个框架事件时被实例化, 所有已被映射, 并被 [Inject] 元数据标签标记过的依赖都会被注入到这个 Command. 另外, 触发这个 Command 的事件实例也会被注入. 当这些依赖被注入完毕, Command 的执行方法会被自动调用, Command 便会进行它的工作. 你不需要, 而且不应该直接调用 execute() 方法. 这是框架的工作.

链接 **Command**

链接 command 也是可行的:

```
public class MyChainedCommand extends Command
{
        [Inject]
        public var event:MyCustomEvent;

        [Inject]
        public var model:MyModel;
```

```
        override public function execute():void
        {
                model.updateData( event.myCustomEventPayload )

                //UPDATED_WITH_NEW_STUFF 触发一个 command 的同时被
                //一个 mediator 接收然后更新一个View Component, 但是只在需要这个响应的时候
                if(event.responseNeeded)
                    dispatch( new MyCustomEvent( MyCustomEvent.UPDATED_WITH_NEW_STUFF,
model.getCalculatedResponse() ) )
        }
}
```

使用这种方法可以把需要的任意多的 Command 链接在一起. 上面的例子使用了一个条件语句. 如果条件不满足 Command 就不会被链接. 这为你的 Command 执行应用程序工作提供了极大的灵活性.

应用程序层的解耦

Command 是解耦一个应用程序里各个 actor 的非常有用的机制. 因为一个 Command 永远不会被 Mediator, Model 或者 Service 实例化或执行, 这些类也就不会被耦合到 command, 甚至都不知道 command 的存在.

为了履行它们的职责, Command 可能:

- 映射 Mediator, Model, Service, 或者 Context 里的其它 Command
- 广播可能被 Mediator 接收或者触发其它 Command 的事件.
- 被注入Model, Service, 和Mediator 以直接进行工作.

> 需要注意的是, 不建议在一个 Command 里直接和 Mediator 交互. 虽然这是可行的, 但会将这个 Mediator 耦合到这个 Command. 因为 Mediator 不像 Service 和 Model, 它可以接受系统事件, 更好的做法是让 Command 广播事件, 然后让需要响应这些事件的 Mediator 监听它们.

Mediator

Mediator 类用来作为用户交互和系统的 View Component 之间的中介. 一个 Mediator 可以在多个级别的粒度上履行它的职责, 中介一个应用程序整体和它的子组件, 或者一个应用程序的任何和所有子组件.

Mediator 职责

Flash, Flex 和 AIR 应用程序为富视觉用户界面组件提供了无限的可能. 所有这些平台都提供了一套组件, 像 DataGrid, Button, Label 和其它常用的UI组件. 也可以继承这些基本的组件来创建自定义组件, 创建复合组件, 或者完全重写新的组件.

一个 View Component 是任何的UI组件和/或它的子组件. 一个 View Component 是已被封装的, 尽可能多地处理自己的状态和操作. 一个 View Component 提供一个包含了事件, 简单方法和属性的API, . Mediators负责代表它所中介的View Component和框架交互. 这包括监听组件及其子组件的事件, 调用其方法, 和读取/设置组件的属性.

一个 Mediator 监听它的 View Component 的事件, 通过 View Component 暴露的 API 访问其数据. 一个 Mediators 通过响应其它框架 actor 的事件并对自己的 View Component 进行相应修改来代表它们. 一个 Mediator 通过转发 View Component 的事件或自己向框架广播合适的事件来通知其它的框架 actor.

映射一个 **Mediator**

任何可以访问到 *mediatorMap* 实例的类都可以映射 Mediator. 这包括 Mediator, Context, 和 Command 类.

这是映射一个 mediator 的语法:

```
mediatorMap.mapView( ViewClass, MediatorClass, autoCreate, autoRemove );
```

View Component 的自动中介

当映射一个 view component 类以获得中介时, 你可以指定是否自动为它创建 Mediator. 当此项为 *true* 时 context 将监听这个 view component 的 ADDED_TO_STAGE 事件. 当收到这个事件这个 view component 会被自动中介, 它的 mediator 就可以开始发送和接收框架事件了.

View Component 的手动中介

有时候可能不希望或者不可能使用 view component 的自动中介. 在这种情况下可以手动创建一个 Mediator 类的实例:

```
mediatorMap.createMediator(viewComponent);
```

这里假设这个 viewComponent 之前已经被 *mediatorMap* 的 *mapView()* 方法映射过了.

映射主程序 (*contextView*) Mediator

映射 contextView 到一个 mediator 是一个常见的模式. 这是个特殊情况, 因为自动中介对 contextView 不起作用, 因为它已经被添加到舞台上, 而不会再发出*mediatorMap* 自动中介所需要的事件了. 这个映射通常在持有 *contextView* 引用的 Context 的 *setup()* 方法里完成:

```
override public function startup():void
{
        mediatorMap.mapView(MediateApplicationExample, AppMediator);
        mediatorMap.createMediator(contextView);
}
```

contextView 并没有被完全中介, 还可以发送和接受框架事件.

访问一个 Mediator 的 View Component

当一个 View Component 在一个 Context 的 contextView 里被添加到舞台上的时候, 它默认地会被根据 MediatorMap 做映射时的配置被自动中介. 在一个基本的 mediator 里, *viewComponent* 会被注入为被中介的 view component. 一个 Mediator 的 *viewComponent* 属性是 Object 类型的. 在大多数情况下, 我们希望访问一个强类型的对象以从中获益. 为此目的, 我们注入被中介的 view component 的强类型实例:

```
public class GalleryLabelMediator extends Mediator implements IMediator
{
        [Inject]
        public var myCustomComponent:MyCustomComponent;

        /**
        * 覆写 onRegister 是添加此 Mediator 关心的任何系统或 View Component 事件的好机会.
        */
        override public function onRegister():void
        {
                //添加一个事件监听器到 Context 来监听框架事件
                eventMap.mapListener( eventDispatcher, MyCustomEvent.DO_STUFF, handleDoStuff
);
                //添加一个事件监听器到被中介的 view component
                eventMap.mapListener( myCustomComponent, MyCustomEvent.DID_SOME_STUFF,
handleDidSomeStuff)
        }

        protected function handleDoStuff(event:MyCustomEvent):void
        {
```

```
                //把事件的强类型负载设置到 view component 的属性.
                //View component 很可能基于这个新数据管理自己的状态.
                myCustomComponent.aProperty = event.payload
        }

        protected function handleDidSomeStuff(event:MyCustomEvent):void
        {
                //把这个事件转发到框架
                dispatch(event)
        }
}
```

通过这种方法我们现在可以很方便地访问被中介的 view component 的公开属性和方法.

给一个 **Mediator** 添加事件监听

事件监听器是 Mediator 的眼睛和鼻子. 因为框架内的所有通讯都通过原生的Flash事件, Mediator 可以通过添加事件监听器来响应感兴趣的事件. 除了框架事件, Mediator同时监听所中介的 view component 的事件.

通常在 Mediator 的 onRegister 方法里添加事件监听. 在 Mediator 生命周期中的这个阶段, 它已经被注册并且它的 view component 和其它依赖也都已被注入. 具体的 Mediator 类必须覆写 onRegister 方法. 也可以在其它方法里添加事件监听, 比如响应框架事件和 view component 事件的事件处理方法里.

Mediators 装备了一个有 mapListener() 方法的 EventMap. 这个方法注册每个被添加到 Mediator 的事件监听, 并且确保 mediator 被框架取消注册时删除这些事件监听. Flash 里删除事件监听是很重要的, 因为如果一个类里添加了事件监听而没有删除, Player将无法对此类进行运行时垃圾回收(GC, Garbage Collection). 也可以使用传统的 Flash 语法添加事件监听器, 但要注意也要手动把它们删除.

监听框架事件

所有框架里的actor在实例化时都会被注入一个 *eventDispatcher* 属性. 这个 *eventDispatcher* 就是 Mediator 发送和接受框架事件的机制.

```
eventMap.mapListener(eventDispatcher, SomeEvent.IT_IS_IMPORTANT, handleFrameworkEvent)
```

通过此语法, 一个 Mediator 现在监听了 *SomeEvent.IT_IS_IMPORTANT* 事件并在 *handleFrameworkEvent* 方法里处理它.

广播框架事件

Mediator的一个很重要的职责就是向框架发送其它 actor 可能感兴趣的事件. 这些事件通常是在响应应用程序用户和被中介的 view component 之间的一些交互时发出的. 这里同样有一个可以减少发送事件到框架的代码输入的很有用的方法:

```
dispatch(new SomeEvent(SomeEvent.YOU_WILL_WANT_THIS, myViewComponent.someData))
```

这个事件现在可以被其它 Mediator 接收或者执行一个 command 了. 发出事件的 Mediator 并不关心其它的 actor 如何回应这个事件, 它只是简单地广播一条有事发生的消息. 一个 mediator 也可以监听自己发出的事件, 然后据此作出回应.

监听 **View Component** 事件

Mediator 负责所中介的 view component 发出的事件. 这可以是个独立组件, 比如 TextField 或者 Button, 也可以是有嵌套层级的复杂组件. 当 mediator 收到 view component 发出的事件会使用指定的方法处理它. 和框架事件一样, EventMap 的 mapListener 方法是给一个 mediator 添加事件监听的首选.

```
eventMap.mapListener(myMediatedViewComponent, SomeEvent.USER_DID_SOMETHING,
handleUserDidSomethingEvent)
```

响应一个 view component 的事件时, 一个 mediator 可能:

- 考察事件的负载 (如果有)
- 考察 view component 的当前状态
- 对 view component 进行需要的工作
- 发送系统事件以通知其它actor有事发生

通过 **Mediator** 访问 **Model** 和 **Service**

你的 mediator 可以监听 Service 和 Model 类派出的系统事件来提高松耦合性. 通过监听事件, 你的 mediator 不需要关心事件来源, 而只需直接使用事件携带的强类型的负载. 因此, 多个 mediator 可以监听相同的事件然后根据所收到的数据调整自己的状态.

在一个 mediator 里直接访问 service 可以提供很大便利而不会带来严重的耦合性问题. 一个 service 并不存储数据, 只是简单地提供一个向外部service发送请求并接受响应的API. 能够直接访问这个API可以避免在你的应用程序中增加不需要的 command 类来达到同样目的. 如果这个 service API 在很多 mediator 中通过相同的方式访问, 将此行为封装到一个 command 里有益于保持此行为的一致性并减少对此 service 的反复调用以及在你的 mediator 里的直接访问.

建议通过 model 和 service 实现的接口将 model 和 service 注入 mediator. 下面的 Service 实例 章节可以找到一个这样的例子.

访问其它 **Mediator**

如同 Service 和 Model,在一个 Mediator 里也可以注入和访问其它的 Mediator. 这种做法是 强烈不建议的 因为这种紧耦合可以简单地通过使用框架事件进行通讯而避免.

Model

Model 类用来管理对应用程序的数据模型的访问. Model 为其它框架actor提供一个 API 来访问, 操作和更新应用程序数据. 这个数据包括但不限于原生数据类型比如 String, Array, 或者像 ArrayCollection 一样的域特有对象或集合.

Model 有时被当做简单的 Model 比如 UserModel, 有时也被当做 Proxy 比如 UserProxy. 在 Robotlegs 里, 这两种命名都是用作相同的目的, 为应用程序数据提供一个 API. 不管采用哪种命名 model 都继承提供了核心框架依赖和一些有用方法的 Actor 基类. 本文档将这些类当做 Model.

Model 职责

Model类封装了应用程序数据模型并为其提供一个 API. 一个 Model 类是你的应用程序数据的看门人. 应用程序里的其它 actor 通过 Model 提供的 API 请求数据. 因为数据是通过 Model 更新, Model 装备了向框架广播事件的机制以向其它 actor 通知数据模型的变化使它们得以据此调整自己的状态.

除了控制对数据模型的访问, Model 通常也被用来保证数据状态的有效性. 这包括对数据进行计算, 或域特有逻辑的其它领域. Model 的这个职责非常重要. Model 是应用程序中最有潜力具有便携性的层. 通过把域逻辑放入 Model, 以后的 model 实现就不再需要像把域逻辑放入 View 或 Controller 层那样重复这些相同的逻辑,

作为一个例子, 你的 Model 里可能执行购物车数据的计算. 一个 Command 将会访问这个方法, 最终的计算结果将会被作为被某个 Mediator 监听的事件派发出去. 这个 mediator 将会根据这个被更新的数据更新自己

的 view component, 应用程序的第一个迭代是个典型的 Flex 程序. 这个计算也很容易在一个 Mediator 甚至视图里进行. 应用程序的第二个迭代是一个需要全新视图元素的移动设备 Flash 应用程序. 因为这个逻辑在 Model 里, 所以可以很容易被两个完全不同元素的视图复用.

映射一个 **Model**

Injector 有几个方法可以用来将你的 Model 类映射到你的框架actor. 另外, 这些方法事实上可以用来注入任何类到你的类里.

将一个已存在的实例当做一个单例注入映射, 使用下面的语法:

```
injector.mapValue(MyModelClass, myModelClassInstance)
```

为每个注入映射一个类的新实例, 使用下面的语法:

```
injector.mapClass(MyModelClass, MyModelClass)
```

另外, 这也可以用来使用被注入的实现某接口的合适的类来映射这个用来注入的接口.

```
injector.mapClass(IMyModelClass, MyModelClass)
```

为某个接口或类映射一个单例实例, 使用下面的语法:

```
injector.mapSingleton(MyModelClass, MyModelClass)
```

需要注意重要的一点, 当提及上面的一个单例时, 它并不是一个单例模式的单例. 在这个 Context 之外并不强制它作为一个单例. Injector 简单地确保这个类的唯一一个实例被注入. 这对处理你的应用程序数据模型的 Model 非常重要.

从一个**Model**里广播事件

Model 类提供一个方便的 *dispatch* 方法用来发送框架事件:

```
dispatch( new ImportantDataEvent(ImportantDataEvent.IMPORTANT_DATA_UPDATED))
```

有很多理由派发一个事件, 包括但不限于:

- 数据已被初始化并准备好被其它 actor 使用
- 一些数据片被添加到 Model
- 数据被从 Model 中删除
- 数据已改变或者更新
- 数据相关的状态已改变

在一个 **Model** 里监听框架事件

虽然技术上可能, 但 强烈不建议 这样做. 不要这样做. 只是为了说清楚: 不要这样做. 如果你这样做了, 不要说你没被警告过.

Service

Service 用来访问应用程序范围之外的资源. 这包括但当然不限于:

- web services
- 文件系统
- 数据库
- RESTful APIs
- 通过 localConnection 的其它 Flash 应用程序

Service 封装了这些和外部实体的交互, 并管理这个交互产生的 result , fault 或其它事件.

你可能注意到 Service 和 Model 的基类非常相像. 事实上, 你可能注意到除了类名, 它们其实是一样的. 那么为什么用两个类呢? Model 和 Service 类在一个应用程序里有完全不同的职责. 这些类的具体实现将不再相像. 如果没有这个分离, 你将经常发现 Model 类在访问外部服务. 这让 Model 有很多职责, 访问外部数据, 解析结果, 处理失败, 管理应用程序数据状态, 为数据提供一个 API, 为外部服务提供一个 API, 等等. 通过分离这些层有助于缓解这个问题.

Service 职责

一个 Service 类为你的应用程序提供一个和外部服务交互的 API. 一个 service 类将连接外部服务并管理它收到的响应. Service 类通常是无状态的实体. 他们并不存储从外部服务收到的数据, 而是发送框架事件来让合适的框架 actor 管理响应数据和失败.

映射一个 **Service**

有 injector 的多个可用的方法 可以用来映射你的 Service 类以注入你的其它框架 actor. 另外, 这些方法也可以用来注入事实上任何类到你的类里.

将一个已存在的实例当做一个单例注入映射, 使用下面的语法:

```
injector.mapValue(MyServiceClass, myServiceClassInstance)
```

为每个注入映射一个类的新实例, 使用下面的语法:

```
injector.mapClass(MyServiceClass, MyServiceClass)
```

另外, 这也可以用来使用被注入的实现某接口的合适的类来映射这个用来注入的接口.

```
injector.mapClass(IMyServiceClass, MyServiceClass)
```

为某个接口或类映射一个单例实例, 使用下面的语法:

```
injector.mapSingleton(MyServiceClass, MyServiceClass)
```

需要注意重要的一点, 当提及上面的一个单例时, 它并不是一个单例模式的单例. 在这个 Context 之外并不强制它作为一个单例. Injector 简单地确保这个类的唯一一个实例被注入.

在一个 **Service** 里监听框架事件

虽然技术上可能, 但 强烈不建议 这样做. 不要这样做. 只是为了说清楚: 不要这样做. 如果你这样做了, 不要说你没被警告过.

广播框架事件

Service 类提供一个方便的 *dispatch* 方法用来发送框架事件:

```
dispatch( new ImportantServiceEvent(ImportantServiceEvent.IMPORTANT_SERVICE_EVENT))
```

Service 示例

下面是来自 Image Gallery demo 的 Flickr service 类 . The Flickr API AS3 Library 做了很多连接到 Flickr 的底层处理. 这个例子使用了它并为在这个例子范围内使用提供了一个简单的抽象.

```
package org.robotlegs.demos.imagegallery.remote.services
{
        import com.adobe.webapis.flickr.FlickrService;
        import com.adobe.webapis.flickr.Photo;
        import com.adobe.webapis.flickr.events.FlickrResultEvent;
        import com.adobe.webapis.flickr.methodgroups.Photos;
        import com.adobe.webapis.flickr.methodgroups.helpers.PhotoSearchParams;

        import org.robotlegs.demos.imagegallery.events.GalleryEvent;
        import org.robotlegs.demos.imagegallery.models.vo.Gallery;
        import org.robotlegs.demos.imagegallery.models.vo.GalleryImage;
        import org.robotlegs.mvcs.Actor;

    /**
     *  这个类使用了 Adobe 提供的 Flickr API 来连接到
     *  Flickr 并获取图片. 它最开始加载当前最"有趣"的
     *  的照片, 同时也提供了搜索其它关键词的能力.
     */
        public class FlickrImageService extends Actor implements IGalleryImageService
        {
                private var service:FlickrService;
                private var photos:Photos;

                protected static const FLICKR_API_KEY:String =
"516ab798392cb79523691e6dd79005c2";
                protected static const FLICKR_SECRET:String = "8f7e19a3ae7a25c9";

                public function FlickrImageService()
                {
                        this.service = new FlickrService(FLICKR_API_KEY);
                }

                public function get searchAvailable():Boolean
                {
                        return true;
                }

                public function loadGallery():void
                {
                        service.addEventListener(FlickrResultEvent.INTERESTINGNESS_GET_LIST,
handleSearchResult);
                        service.interestingness.getList(null,"",20)
                }

                public function search(searchTerm:String):void
                {
                        if(!this.photos)
                                this.photos = new Photos(this.service);
                        service.addEventListener(FlickrResultEvent.PHOTOS_SEARCH,
handleSearchResult);
                        var p:PhotoSearchParams = new PhotoSearchParams()
                        p.text = searchTerm;
                        p.per_page = 20;
                        p.content_type = 1;
                        p.media = "photo"
                        p.sort = "date-posted-desc";
                        this.photos.searchWithParamHelper(p);
                }

                protected function handleSearchResult(event:FlickrResultEvent):void
                {
                        this.processFlickrPhotoResults(event.data.photos.photos);
                }

                protected function processFlickrPhotoResults(results:Array):void
                {
```

```
                        var gallery:Gallery = new Gallery();

                        for each(var flickrPhoto:Photo in results)
                        {
                                var photo:GalleryImage = new GalleryImage()
                                var baseURL:String = 'http://farm' + flickrPhoto.farmId +
'.static.flickr.com/' + flickrPhoto.server + '/' + flickrPhoto.id + '_' + flickrPhoto.secret;
                                photo.thumbURL = baseURL + '_s.jpg';
                                photo.URL = baseURL + '.jpg';
                                gallery.photos.addItem( photo );
                        }

                        dispatch(new GalleryEvent(GalleryEvent.GALLERY_LOADED, gallery));
                }

        }
}
```

FlickrGalleryService 提供了一个连接到一个 gallery 服务的非常简单的接口. 应用程序可以 *loadGallery*, *search*, 并查询 *searchAvailable* 是 true 还是 false. IGalleryService 接口定义的接口:

```
package org.robotlegs.demos.imagegallery.remote.services
{
        public interface IGalleryImageService
        {
                function loadGallery():void;
                function search(searchTerm:String):void;
                function get searchAvailable():Boolean;
        }
}
```

Services 应该实现一个接口

通过创建实现了接口的 service, 为了测试在运行时切换它们, 或者对应用程序的最终用户提供对其它 service 的访问将会很简单. 比如, FlickrGalleryService 可以很容易替换为 XMLGalleryService:

```
package org.robotlegs.demos.imagegallery.remote.services
{
        import mx.rpc.AsyncToken;
        import mx.rpc.Responder;
        import mx.rpc.http.HTTPService;

        import org.robotlegs.demos.imagegallery.events.GalleryEvent;
        import org.robotlegs.demos.imagegallery.models.vo.Gallery;
        import org.robotlegs.demos.imagegallery.models.vo.GalleryImage;
        import org.robotlegs.mvcs.Actor;

        public class XMLImageService extends Actor implements IGalleryImageService
        {
                protected static const BASE_URL:String = "assets/gallery/";

                public function XMLImageService()
                {
                        super();
                }

                public function get searchAvailable():Boolean
                {
                        return false;
                }

                public function loadGallery():void
                {
                        var service:HTTPService = new HTTPService();
                        var responder:Responder = new Responder(handleServiceResult,
handleServiceFault);
                        var token:AsyncToken;
                        service.resultFormat = "e4x";
                        service.url = BASE_URL+"gallery.xml";
                        token = service.send();
                        token.addResponder(responder);
                }

                public function search(searchTerm:String):void
```

```
                {
                        trace("search is not available");
                }

                protected function handleServiceResult(event:Object):void
                {
                        var gallery:Gallery = new Gallery();

                        for each(var image:XML in event.result.image)
                        {
                                var photo:GalleryImage = new GalleryImage()
                                photo.thumbURL = BASE_URL + "images/" + image.@name +
'_s.jpg';
                                photo.URL = BASE_URL + "images/" + image.@name + '.jpg';
                                gallery.photos.addItem( photo );
                        }

                        dispatchEvent(new GalleryEvent(GalleryEvent.GALLERY_LOADED,
gallery));
                }

                protected function handleServiceFault(event:Object):void
                {
                        trace(event);
                }
        }
}
```

XML gallery 提供了和 Flickr 一样的方法可以访问并可在任何 IGalleryService 接口被调用的地方进行替换. 这些 service 派发相同的事件并且在最终的应用程序里很难区分. 在这个例子里, 搜索并没有被实现, 但搜索功能在这个 service 里也同样可以很容易实现,

建议所有的 service 都实现一个定义了它们 API 的接口. 在框架 actor 里接收一个 service 作为依赖注入时可以请求这个接口, 而不是具体的实现.

```
injector.mapSingletonOf(IGalleryService, FlickrGalleryService);
```

```
[Inject]
public var galleryService:IGalleryService
```

你可以通过简单地改变注入来使用你的类代替这个 gallery service:

```
injector.mapSingletonOf(IGalleryService, XMLGalleryService);
```

这种方式可以为一个应用程序提供健壮性, 灵活性, 和增强的可测试性.

在一个 **Service** 里解析数据

在上面的例子里 service 类或者外部服务提供了不符合应用程序域的对象. Flickr service 提供强类型的 Photo 对象而 XML service 提供 xml. 这些数据类型都很好用, 但是并不符合我们应用程序的 context. 它们是外来者. 可以围绕外部数据类型对应用程序进行建模, 或者更可取地, 转换这些数据以符合应用程序.

应用程序里有两处可以进行这项操作/转换. Service 和 Model 都很适合. Service 是进入外部数据的第一个点, 所以它是操作一个外部服务返回的数据的更好的选择. 外来数据应该在第一个机会转换到应用程序域.

提供一个使用工厂类而不是在 *service* 里生成应用程序域对象的例子... 适当的

当数据被转换为应用程序域特有的对象之后发出带有强类型负载的事件以被对此关心的 actor 立即使用.

Service 事件

service 组合的最后一个部分是自定义事件. 没有事件的 service 只是哑巴. 他们可能做的任何工作都不会被

其它框架成员注意到. 一个 service 将会使用自定义事件来向应用程序发出声音. 事件并不一定是唯一的意图. 如果这个 service 正在转换数据它可以使用一个普通的事件来派发强类型的数据给感兴趣的应用程序 actor.